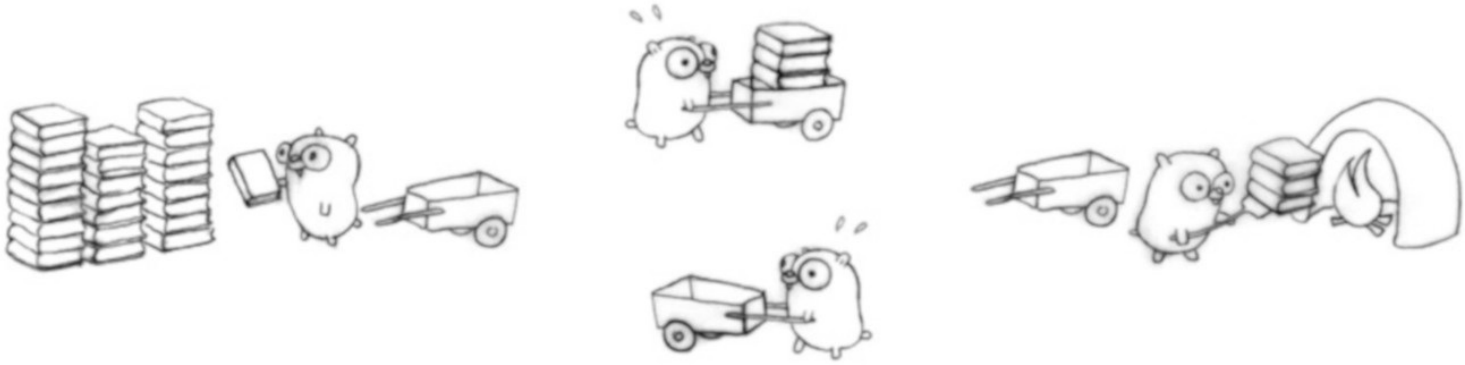
Go语言101

-= v1.23.b-3d9e142 =-



老貘 著

目录

- 第0章：关于《Go语言101》 – *为什么写这本书*
- 第1章：致谢

- 第2章：Go语言简介 – *为什么Go语言值得学习*
- 第3章：Go官方工具链 – *如何编译和运行Go程序*

- Go编程入门
  - 第4章：程序源代码基本元素介绍
  - 第5章：关键字和标识符
  - 第6章：基本类型和它们的字面量表示
  - 第7章：常量和变量 – *顺便介绍了类型不确定值和类型推断*
  - 第8章：运算操作符 – *顺便介绍了更多的类型推断规则*
  - 第9章：函数声明和调用
  - 第10章：代码包和包引入
  - 第11章：表达式、语句和简单语句
  - 第12章：基本流程控制语法
  - 第13章：协程、延迟函数调用、以及恐慌和恢复

- Go类型系统
  - 第14章：Go类型系统概述 – *精通Go编程必读*
  - 第15章：指针
  - 第16章：结构体
  - 第17章：值部 – *为了更容易和更深刻地理解Go中的各种值*
  - 第18章：数组、切片和映射 – *Go中的首要容器类型*
  - 第19章：字符串
  - 第20章：函数 – *函数类型和函数值，以及变长参数个数函数*
  - 第21章：通道 – *Go特色的并发同步方式*
  - 第22章：方法
  - 第23章：接口 – *通过包裹不同具体类型的非接口值来实现反射和多态*
  - 第24章：类型内嵌 – *不同于继承的类型扩展方式*
  - 第25章：非类型安全指针
  - 第26章：泛型 – *如何使用和解读组合类型*
  - 第27章：反射 – *`reflect` 标准库包中提供的反射支持*

- 一些专题
  - 第28章：代码断行规则
  - 第29章：更多关于延迟函数调用的知识点
  - 第30章：一些恐慌/恢复用例
  - 第31章：详解panic/recover原理 – *也解释了什么是“函数退出阶段”*
  - 第32章：代码块和标识符作用域

    - 第33章：表达式估值顺序规则
    - 第34章：值复制成本
    - 第35章：边界检查消除

- 并发编程
    - 第36章：并发同步概述
    - 第37章：通道用例大全
    - 第38章：如何优雅地关闭通道
    - 第39章：其它并发同步技术 - *如何使用`sync`标准库包*
    - 第40章：原子操作 - *如何使用`sync/atomic`标准库包*
    - 第41章：Go中的内存顺序保证
    - 第42章：一些常见并发编程错误

- 内存相关
    - 第43章：内存块
    - 第44章：关于Go值的内存布局
    - 第45章：一些可能的内存泄漏场景

- 一些总结
    - 第46章：一些简单的总结
    - 第47章：关于Go中的`nil`
    - 第48章：类型转换、赋值和值比较规则大全
    - 第49章：Go中的一些语法/语义例外
    - 第50章：Go细节101
    - 第51章：Go问答101
    - 第52章：Go技巧101

- 第53章：更多关于Go的知识

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

关于《Go语言101》

我觉得很难用通常的描述方式来描述本文中的内容。 所以本文将采用采访的形式来进行描述。

你好，老貘，你是什么时候开始写这本书的?

大概在2016年七月份左右，不是很密集地使用了Go两年后，我觉得Go是一门简单的语言，我感觉我已经精通了Go编程。 在那个时候，我在日常编程中已经搜集了不少关于Go编程的细节。 我觉得我可以将这些细节写成一本书。 我觉得这应该是一件容易的事。

但是我发现我错了。我太过于自信了。 在试图解释一些细节时，我发现我无法把它们解释清楚。 我对Go编程中的很多细节的底层原因产生了困惑。 随着越来越多的困惑的积攒，我觉得我对Go的领悟非常有限。 我感觉我仍然是一个Go新手程序员。

我放弃了写那本书。

放弃? 《Go语言101》现在不是已经完成了吗?

那本曾经计划要写的书不是《Go语言101》。 放弃那本书的写作计划后，我通过阅读很多官方Go文档和网络中的各种Go文章、关注Go官方项目的问题跟踪列表和一些Go论坛、查看一些代码等途径， 逐渐地，我几乎消除了我所有关于Go编程细节中的困惑。

我大概花了大约一年时间来消除这些困惑。 在这个时期，每当我消除了某个主题的困惑，我就以该主题写一篇博客文章。 最后，我写了大约20篇Go文章。 与此同时，我收集到了比以前更多的Go编程中的细节。 此时到了重启编写一本新的Go编程书籍的时候了。

我写了另外大约十篇Go基础教程和另外大约二十篇Go语言中关于各种其它主题的文章。 所以现在《Go语言101》大约有50篇文章。

你曾经的困惑主要包括哪些方面?

一些困惑是关于一些Go语法和语义设计细节的。 一些困惑涉及到某些类型的值，主要是切片，接口和通道类型。 另外一些涉及到标准包API的使用细节。

你认为造成你曾经的困惑的主要原因是什么?

我觉得最主要的原因是我当时抱着Go是一门非常简单的语言的态度去学习和使用Go。 持有这种态度阻止了我更深刻地理解Go。

Go是一门特性丰富的语言。它的语法集虽然不大，但我们也不能说它很小。 Go中的一些语法和语义设计很简单明了，但也有一些设计略微反直觉，甚至自相矛盾。 Go语法和语义设计中有很多折衷和权衡。一个Go程序员需要相当的Go编程经验和感悟才能理解这些权衡。

Go提供了几种基本但非必需的类型，比如切片，接口和通道。 Go编译器和运行时在实现这些类型的时候，进行了必要的封装。 一方面，这些封装为Go编程带来了许多便利，使我们不用从头实现这些类型。 但另一方面，这些封装隐藏了这些类型的内部结构， 从而对我们更深入地理解这些类型的值的行为带来了一些障碍。

许多官方和非官方的Go教程都非常简单。 这些教程只涵盖了一般典型用例，而忽略了许多细节。 这对鼓励新手Go程序员学习和使用Go非常有好处， 但另一方面，这也使许多Go程序员对他们的Go知识掌握程度过度自信。 从长远看，这不利于一个Go程序员更好地理解和使用Go。

标准库包中声明的某些函数和类型没有得到详细的解释。很多时候这是可以理解的。 因为很多细节解释起来很拗口和微妙，有时很难找到适当的措词来清楚地解释它们。 少量但准确的描述比大量但不准确的描述要好。但这确实也给Go程序员们留下了一些困惑。

所以你认为简单不是Go的卖点吗?

我认为，至少简单不是Go的主要卖点。毕竟有一些其它语言确实比Go简单。 另一方面，虽然Go是一门特性丰富的语言，但是它却也不是一门复杂的语言。 一个持有积极的学习态度的Go新手程序员可以在一年内精通Go编程。

那你觉得Go的卖点是什么呢?

我个人的观点是，做为一门静态语言，Go却和很多动态脚本语言一样得灵活是Go的主要卖点。

节省内存、程序启动快、代码执行速度快和编译速度快合在一块儿是Go的另一个主要卖点。 虽然这三项是C家族语言的共同特征，但是在Web开发领域，很少有语言同时拥有这四个特征。 事实上，这就是我当初从Java转到Go进行Web开发的原因。

内置并发编程支持也算是Go的卖点，虽然我个人认为它不是Go的主要卖点。

良好的代码可读性是Go的另一个重要卖点。 我感觉可读性是Go在设计的时候考虑的最重要的一个因素。

良好的跨平台支持也应该算是Go的一个卖点，尽管如今这个卖点并不是很新鲜。

一个稳定的Go核心设计和开发团队以及一个活跃的社区也可以被视为Go的一个卖点。

《Go语言101》做了什么来消除Go编程中可能出现的困惑?

《Go语言101》做了以下这些方面来试图清除很多Go编程中可能遇到的困惑。

1. 着重于基本概念和术语的解释。如果不理解或者不熟悉这些基本概念和术语，就很难完全理解Go中的很多规则和高层次的概念。
2. 使用了值部（value part）这个术语并单独用一篇文章来解释值部。 这篇文章揭示了某些类型的底层结构，从而使得Go程序员可以更深入地理解这些类型的值。 我认为知道一些可能的底层实现对于清除某些Go编程中的困惑非常有帮助。
3. 详细地解释了内存块（memory block）。 了解Go值和内存块之间的关系对于理解垃圾收集器是如何工作的以及如何避免内存泄漏非常有帮助。
4. 将接口值视为用于包裹非接口值的盒子。 我发现将接口值视为用于包裹非接口值的盒子对于清除很多和接口相关的困惑非常有帮助。
5. 澄清了Go白皮书中的一些含糊描述，包括内嵌规则、提升方法估值和恐慌/恢复机制。
6. 汇总了许多知识点和细节，从而可以帮助Go程序员节省很多学习时间。

有什么其它值得一提吗?

本书不涵盖自定义泛型相关内容。 请阅读《Go自定义泛型101》（https://gfw.go101.org/generics/101.html）一书来了解使用自定义泛型。

另外，在阐述值类型转换、值赋值和值比较规则时，自定义泛型中频繁使用的类型参数类型被特意忽略掉了。 也就是说，本书不考虑自定义泛型中的情形。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

致谢

首先，感谢整个Go社区。 如果没有一个活跃和交流顺畅的社区，本书很难完成。

特别感谢Ian Lance Taylor。 Ian十分耐心地解答了我在go-nuts群组提出的无数枯燥的问题。 Ian的解答帮助我清除了很多曾经在Go编程中遇到的困惑。

感谢下面这些直接给予了帮助的社区成员： Axel Wagner、 Robert Griesemer、 Keith Randall、 Brad Fitzpatrick、 Matthew Dempsky、 Russ Cox、 Jan Mercl、 Brad Fitzpatrick、 Joe Tsai、 Bryan C。 Mills、 Martin Möhrmann、 Konstantin Khomoutov、 Alan Donovan、 Minux Ma、 Dave Cheney、 Josh Bleecher Snyder、 Volker Dobler、 Caleb Spare、 Matt Harden、 Roger Peppe、 Michael Jones、 peterGo、 Pietro Gagliardi、 Paul Jolly、 Daniel Martí、 Andrew Bonventre、 Damian Gryski、 Alberto Donizetti、 Emmanuel T Odeke、 Filippo Valsorda、 Dominik Honnef、 和 Rob 'Commander' Pike 等。

感谢直接参与本书写作和改进的Go社区成员，包括： Amir Khazaie、 Ziqi Zhao、 Artur Kirillov、 Arinto Murdopo、 Andreas Pannewitz、 Jason-Huang、 Joel Rebello、 Julia Wong、 Wenbin Zhang、 Farid Gh、 Valentin Deleplace、 nofrish、 Peter Matseykanets、 KimMachineGun、 Yoshinori Kawasaki、 mooncaker816、 Sangwon Seo、 Shaun (wrfly)、 Stephan Renatus、 Yang Shu、 stemar94、 berkant (yadkit)、 Thomas Bower、 Patryk Małek、 kucharskim、 Rodrigue Koffi、 Jhuliano Skittberg Moreno、 youmoo、 ahadc、 Kevin、 jojo (hjb912)、 Gordon Wang、 Steve Zhang、 cerlestes、 Gleb Sozinov、 Jake Montgomery、 Erik Dubbelboer、 Afriza N. Arief、 Maxim Neverov、 Asim Himani、 sulinehk、 Grachev Mikhail、 halochou、 GeXiao、 Knight Young、 binderclip、 sunznx、 haozibi、 Tony Bai、 lanastasov、 liukaifei、 Abirdcfly、 Tahir Raza、 lniwn、 GFZRZK、 Jiadas、 黎显图、 Raymon Zhang、 Crazy Yang、 sdjdd、 NOmansky、 bestgopher、 9r0k、 wieghx、 SourceLink、 Will Xiang、 Cholerae Hu、 ilvsx、 Ibrahim Mohammed、 n374、 Genaro-Chris、 Ying-Han Chen、 Sina-Ghaderi、 Acehi、 yngwiewang、 chenlujjj、 Phan Phu Thanh、 Frank Meyer、 moveurbody、 Shane、 V0idk、 LiuHe、 yikakia、 ZhangJuChang、 Jugg.Gao、 KONY、 toywei、 yao、 luozhiyun、 Huang Chao、 Jiangwei Lu、 lisgroup、 Cuong Manh Le、 NeoFive、 Roman Ilchyshyn、 Muhan Li、 zhangshuai、 ch0ngsheng、 Yussif Mohammed、 W.T. Chang、 cui fliter、 LGiki、 CooperLi、 Aleksandr Shalimov、 Eric (kuchaguangjie)、 iridescentGray、 zhaoshouhang、 zunlongzhou、 Wu ZhaoYan、 Li ShuMin、 Bai Kai、 modood、 xmwilldo、 TheChalice和zonesan等。

特别感谢白凯同学帮助我翻译了《细节101》、《问答101》和《技巧101》三篇文章。

我很抱歉如果上述列表遗漏了某个曾经给予我帮助的成员。 Go社区有如此多友善和富有创造性的成员，以至于上述列表肯定遗漏了某些成员。 感谢所有曾经直接或者间接，有意或者无意帮助过我完成这本书的Go社区成员。

另外也要感谢Bootstrap CSS框架（https://getbootstrap.com/docs/3.3/）、jQuery（https://jquery.com）/code prettify（https://github.com/google/code-prettify）/prism（https://prismjs.com/）JavaSript库、go-epub（https://github.com/bmaupin/go-epub）库和calibre（https://calibre-ebook.com/）软件的作者。 这些软件用于构建g101.org网站和《Go语言101》电子书。

感谢Adam Chalkley对电子书制作过程提出的改进建议。

特别感谢Renee French（http://reneefrench.blogspot.com/）和Rob Pike。 本书电子版和纸质版封面中的生动的图片来源于Rob Pike的一张幻灯片（https://talks.golang.org/2012/waza.slide#19）。 Renee French是这张图片中的地鼠卡通形象的设计者。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

Go语言简介

Go是一门编译型的和静态的编程语言。 Go诞生于谷歌研究院。 Go的核心设计成员中包括很多有着数十年编程语言研究领域经验的研究者。

Go有很多特性，有一些是独特的，有一些借鉴于一些其它编程语言：

- 内置并发编程支持：
  - 使用协程（goroutine）做为基本的计算单元。轻松地创建协程。
  - 使用通道（channel）来实现协程间的同步和通信。
- 内置了映射（map）和切片（slice）类型。
- 支持多态（polymorphism）。
- 使用接口（interface）来实现装盒（value boxing）和反射（reflection）。
- 支持指针。
- 支持函数闭包（closure）。
- 支持方法。
- 支持延迟函数调用（defer）。
- 支持类型内嵌（type embedding）。
- 支持类型推断（type deduction or type inference）。
- 内存安全。
- 自动垃圾回收。
- 良好的代码跨平台性。
- 自定义泛型（从Go 1.18开始）。

除了以上特性，Go还有如下亮点：

- Go的语法很简洁并且和其它流行语言相似。 这使得具有一定编程经验的程序员很容易上手Go编程。 当然，对于没有编程经验的初学者，Go也比很多其它流行编程语言更容易上手一些。
- Go拥有一个比较齐全的标准库。这个标准库提供了很多常用的功能。
- Go拥有一个活跃和回应快速的社区。 社区贡献了大量高质量的第三方库包和应用（https://github.com/avelino/awesome-go）。

Go程序员常常被称为地鼠（gopher）。

上面已经提到，Go属于编译型的静态语言。但是Go的很多特性使得用Go编程像使用动态脚本语言一样的灵活。 一般很难将静态语言的严格性和动态语言灵活性的优点合二为一。但是Go做到了这一点。 当然，合二为一也会不可避免地带来一些弊端。但是，对于Go来说，合二为一带来的好处远多于合二为一的带来的弊端。

可读性是在Go语言的设计中一个非常重要的考虑因素。 一个Go程序员常常可以轻松读懂其他Go程序员写的代码。 甚至对于一个没有Go编程经验但具有其它语言编程经验的程序员来说，读懂一份Go源码也不是一件难事。

目前，使用最广泛的Go编译器由Go官方设计和开发团队维护。 以后我们将称此编译器为标准编译器。标准编译器也常常称为gc（是Go compiler的缩写，不是垃圾回收garbage collection的缩写）。 Go官方设计和开发团队也维护着另外一个编译器，gccgo。 gccgo是gcc编译器项目的一个子项目。 gccgo的使用广泛度大不如gc， 它的主要作用是做为一个参考，来保证gc的实现正确性。 目前两个编译器的开发都很活跃，尽管Go开发团队在gc的开发上花费的精力更多。

gc编译器是Go官方工具链中一个组件。 Go官方工具链的使用将在下一篇文章中介绍。 Go官方工具链1.0发布于2012年三月。 Go语言规范的最新版本和Go官方工具链的最新版本总是保持一致。 每年Go官方工具链发行两个主版本。

自从Go语言正式发布后，Go的语法变化很小。 但是标准编译器gc却在不断地改进。 使用早期的gc编译的程序在运行的时候在每次垃圾回收的结尾常常会有明显的停顿。 但是自从Go 1.8，使用gc编译的程序在运行时刻已经基本消除了停顿现象。

gc支持跨平台编译。 比如，我们可以在Linux平台上编译出Windows程序，反之亦然。

使用Go编写的程序常常编译得非常快。 编译时间的长短是开发愉悦度的一个重要因素。 编译时间短是很多程序员喜欢Go的一个原因。

Go程序生成的二进制可执行文件常常拥有以下优点：

- 内存消耗少
- 执行速度快
- 启动快

很多C家族语言，比如C/C++/Rust等，也拥有上述的优点。 但它们缺少Go语言的几个重要优点：

- 程序编译时间短
- 像动态语言一样灵活
- 内置并发支持

上面所有提到的优点使得Go成为一个出众的编程语言。 对于很多项目来说，Go是一个相当不错的选择。 目前，Go主要用于网络开发、系统工具开发、数据库开发和区块链开发。 随着从Go 1.18开始支持自定义泛型，预期Go会在更多开发领域流行起来，比如图形界面、游戏、大数据和人工智能等。

最后，我们应该知道，没有一门语言是完美的。Go也一样。Go的设计中有很多折衷和各种权衡。Go 1确实有一些不足。 比如，目前Go不支持任意类型的不变量。这导致很多标准库中一些希望永不被更改的值目前被声明为变量。这是Go程序中的一个潜在安全隐患。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

Go官方工具链

目前，Go官方工具链中提供的工具是使用得最广泛的Go开发工具。 《Go语言101》所有中所有的实例代码都使用Go官方工具链中提供的标准编译器编译验证过。

本文将介绍如何配置Go开发环境和如何使用Go官方工具链中提供的`go`命令。 一些非官方工具链中的工具也将简单提及。

安装Go官方工具链

请从国际官网（https://go.dev/doc/install）或者国内官网（https://golang.google.cn/doc/install）下载Go官方工具链， 并按照下载页面中的说明安装Go官方工具链。

我们也可以使用工具GoTV（https://go101.org/apps-and-libs/gotv.html）来安装多个Go官方工具链版本，并方便和谐地使用它们。

Go官方工具链的版本和其所支持的最高Go语言版本是一致的。 比如，Go官方工具链1.23.x版本支持从1.0到1.23的所有Go语言版本。

为了从任意目录运行Go官方工具链中工具命令（通过`go`命令）， Go官方工具链安装目录下的`bin`子目录路径必须配置在`PATH`环境变量中。 当使用安装程序安装Go官方工具链时，安装程序很可能已经自动地将此配置好了。

Go官方工具链近来的版本均支持一个称为Go模块（Go modules）的特性，用来管理项目依赖。 此特性在版本1.11中被试验性地引入并从版本1.16开始被默认支持。

我们应该了解一个环境变量： `GOPATH`。 此环境变量的默认值为当前用户的HOME目录下的名为`go`文件夹对应的目录路径。 `GOPATH`环境变量可以被手动地配置多个路径。 以后，当`GOPATH`文件夹被提及的时候，它表示`GOPATH`环境变量中的第一个路径对应的文件夹。

- `GOPATH`文件夹中的`pkg`子文件夹用来缓存被本地项目所依赖的Go模块（一个Go模块为若干Go库包的集合）的版本。
- `GOBIN`环境变量用来指定`go install`子命令产生的Go应用程序二进制可执行文件应该存储在何处。 它的默认值为`GOPATH`文件夹中的`bin`子目录所对应的目录路径。 `GOBIN`路径需配置在`PATH`环境变量中，以便从任意目录运行这些Go应用程序。

最简单的Go程序

让我们写一个简单的Go程序，并且学习如何运行之。

下面的程序应该是最简单的Go程序。

```
1| package main
2|
3| func main() {
4| }
```

在此程序中，单词`package`和`func`是两个关键字。 两个`main`是两个标识符。 标识符和关键字将在后续的一篇文章中讲解。

此程序的第一行指定了当前源代码文件所处的包的包名（此处为`main`）。 第二行是一个空行，用来增强可读性。 第三和第四行声明了一个名为`main`的函数。 此函数为程序的入口函数。

运行一个Go程序

Go官方工具链工具要求所有的Go源代码文件必须以`.go`后缀结尾。 这里，我们假设上面展示的最简单的Go程序存放在一个名叫`simplest-go-program.go`的文件中。

打开一个终端（控制台）并进入上述源文件所在的目录，然后运行

```
$ go run simplest-go-program.go
```

什么也没输出？是的，此程序不做什么有实质意义的事儿。

如果代码中有语法错误，这些错误将输出在终端中。

如果一个程序的`main`包中有若干Go源代码文件，我们也可以使用下面的命令运行此程序。

```
$ go run .
```

注意：

- `go run`子命令并不推荐在正式的大项目中使用。`go run`子命令只是一种方便的方式来运行简单的Go程序。 对于正式的项目，最好使用`go build`或者`go install`子命令构建可执行程序文件来运行Go程序。
- 支持Go模块特性的Go项目的根目录下需要一个`go.mod`文件。此文件可以使用`go mod init`子命令来生成（见下）。
- 名称以`_`和`.`开头的源代码文件将被Go官方工具链工具忽略掉。

更多go子命令

上面提到的三个go子命令（go run、go build和go install） 将只会输出代码语法错误。它们不会输出可能的代码逻辑错误（即警告）。 go vet子命令可以用来检查可能的代码逻辑错误（即警告）。

我们可以（并且应该常常）使用go fmt子命令来用同一种代码风格格式化Go代码。

我们可以使用go test子命令来运行单元和基准测试用例。

我们可以使用go doc子命令来（在终端中）查看Go代码库包的文档。

强烈推荐让你的Go项目支持Go模块特性来简化依赖管理。对一个支持Go模块特性的项目：

- go mod init example.com/myproject命令可以用来在当前目录中生成一个go.mod文件。 当前目录将被视为一个名为example.com/myproject的模块（即当前项目）的根目录。 此go.mod文件将被用来记录当前项目需要的依赖模块和版本信息。 我们可以手动编辑或者使用go子命令来修改此文件。
- go mod tidy命令用来通过扫描当前项目中的所有代码来添加未被记录的依赖至go.mod文件或从go.mod文件中删除不再被使用的依赖。
- go get命令用拉添加、升级、降级或者删除单个依赖。此命令不如go mod tidy命令常用。

从Go官方工具链1.16版本开始，我们可以运行go install example.com/program@latest来安装一个第三方Go程序的最新版本（至GOBIN目录）。 在Go官方工具链1.16版本之前，对应的命令是go get -u example.com/program（现在已经被废弃而不再推荐被使用了）。

我们可以运行go help aSubCommand来查看一个子命令aSubCommand的帮助信息。

运行不带参数的go命令将会列出所有支持的go子命令。

《Go语言101》系列文章将不再对各种go子命令做更多的解释。 请阅读官方文档（https://golang.google.cn/cmd/go/） （墙外版（https://golang.org/cmd/go/））以获取更多信息。

查看Go代码库文档

我们可以使用Go项目文档和代码阅读工具**Golds**（https://github.com/go101/golds）（**go101.org/golds**，由本书作者老貘开发）来阅读代码库文档。 此工具将列出类型的实现关系，并且支持良好的代码阅读体验。 安装**Golds**完成之后，我们可以运行`golds std ./...`来阅读Go标准库和当前文件夹下的所有库包的文档。

我们可以在国内官网（https://golang.google.cn）或者国际官网（https://go.dev）阅读Go官方文档。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

程序源代码基本元素介绍

相比很多其它流行语言，Go的语法相对简洁。 此篇文章将介绍编程中常用的代码元素，并展示一份简单的Go程序代码，以便让刚开始学Go编程的程序员对Go代码结构有一个大概的印象。

编程和程序代码元素

简单来讲，编程可以看作是以各种方式控制和组合计算机运行中的各种操作，以达到各种各样的目的。 一个操作可能从一个硬件设备读取、或者向一个硬件设备写入一些数据，从而完成一个特定的任务。 对于现代计算机来说，最基本的操作是底层计算机指令，比如CPU和GPU指令。 常见的硬件设备包括内存、磁盘、网卡、显卡，显示器、键盘和鼠标等。

直接操控底层计算机指令进行编程是非常繁琐和容易出错的。 高级编程语言通过对底层指令进行一些封装和对数据进行一些抽象，从而使得编程变得直观和易于理解。

在流行高级编程语言中，一个操作通常是通过**函数**（function）调用或者使用**操作符**（operator）运算来完成的。 大多数高级编程语言都支持一些条件和循环控制语句。 这些条件和循环控制语句可以看作是特殊的操作。 它们的语法接近于人类语言，因此一个人写的代码很容易被其他人理解。

在大多数高级编程语言中，数据通常被抽象为各种**类型**（type）和**值**（value）。 一个类型可以看作是值的模板。一个值可以看作是某个类型的实例。 大多数编程语言支持自定义类型和若干预声明类型（即内置类型）。 一门语言的类型系统可以说是这门语言的灵魂。

编程中常常会使用大量的值。 一些在编码阶段可确定的值可以用它们的**字面形式**（literal，即字面量）来表示。 为了编程灵活和不易出错，其它的值一般使用**变量**（variable）和**（具名）常量**（named constant）来表示。

在《Go语言101》中，具名的函数、具名的值（包括变量和具名常量）、以及定义类型和类型别名将被统称为代码要素。 代码要素名必须为[标识符（identifier）](#)（**第5章**）。

高级编程语言代码将被编译器或者解释器转换为底层机器码进行执行。 为了帮助编译器和解释器解析高级语言代码，一些单词将被用做**关键字**（keyword）。 这些单词不能被当做标识符使用。

很多现代高级语言使用**包**（package）来组织代码。 一个包必须**引入**（import）另一个包才能使用另一个包中的公有（导出的）代码要素。 包名和包的引入名也都必

须是标识符。

尽管高级编程语言代码比底层机器指令友好和易懂，我们还是需要一些**注释**来帮助自己和其他程序员理解我们所写的代码。 在下一节的程序示例中，我们可以看到很多注释。

一个简单的Go示例程序

为了对各种代码元素有一个更清楚的认识，让我们来看一个简短的Go示例程序。 和很多其流行语言一样，Go使用`//`来起始一个行注释，使用一个`/*`和`*/`对来包裹一个块注释。

下面是这个Go示例程序。请注意阅读其中的注释。程序之后有更多解释。

```
 1| package main // 指定当前源文件所在的包名
 2|
 3| import "math/rand" // 引入一个标准库包
 4|
 5| const MaxRand = 16 // 声明一个具名整型常量
 6|
 7| // 一个函数声明
 8| /*
 9|  StatRandomNumbers生成一些不大于MaxRand的非负
10|  随机整数，并统计和返回小于和大于MaxRand/2的随机数
11|  个数。输入参数numRands指定了要生成的随机数的总数。
12| */
13| func StatRandomNumbers(numRands int) (int, int) {
14|    // 声明了两个变量（类型都为int，初始值都为0）
15|    var a, b int
16|    // 一个for循环代码块
17|    for i := 0; i < numRands; i++ {
18|       // 一个if-else条件控制代码块
19|       if rand.Intn(MaxRand) < MaxRand/2 {
20|          a = a + 1
21|       } else {
22|          b++ // 等价于：b = b + 1
23|       }
24|    }
25|    return a, b // 此函数返回两个结果
26| }
27|
28| // main函数，或主函数，是一个程序的入口函数。
29| func main() {
30|    var num = 100
```

```
31|    // 调用上面声明的StatRandomNumbers函数，
32|    // 并将结果赋给使用短声明语句声明的两个变量。
33|    x, y := StatRandomNumbers(num)
34|    // 调用两个内置函数（print和println）。
35|    print("Result: ", x, " + ", y, " = ", num, "? ")
36|    println(x+y == num)
37| }
```

将上面的程序代码存盘到一个名为basic-code-element-demo.go 的文件中并使用下列命令运行此程序：

```
$ go run basic-code-element-demo.go
Result: 46 + 54 = 100? true
```

在上面的示例程序中，单词package、import、const、func、var、for、if、else和return均为关键字。 其它大多数单词均为标识符。 请阅读关键字和标识符（第5章）以获得更多关于关键字和标识符的信息。

四个int（一个在第15行，另三个在第13行） 表示内置基本类型int。int类型是Go中的内置基本整数类型之一。 第5行中的16、第17行中的0、 第20行中的1以及第30行的100均为整型字面量。 第35行的"Result: "是一个字符串字面量。 请阅读基本类型和它们的字面量表示（第6章）以获取更多关于基本类型和它们的字面量的信息。 Go中的非基本类型（均为组合类型）将在以后的其它文章中介绍和解释。

第20行是一个赋值语句。 第5行声明了一个具名常量，叫做MaxRand。 第15行和第30行使用标准变量声明语句声明了三个变量。 第17行的变量i以及第33行的变量x和y是使用变量短声明语句声明的。 变量a和b在声明的时候被指定为int类型。编译器会自动推导出变量i、num、x和y的类型均为int类型，因为它们的初始值都是整型字面量表示的。 请阅读常量和变量（第7章）以获取什么是类型不确定值、类型推导、赋值、以及如何声明变量和具名常量。

上面的示例程序中使用了很多操作符，比如第17和19行的小于比较符<，第36行的等于比较符==，还有第20和36行的加法运算符+。 第35行中的+不是一个运算符，它是一个字符串字面量中的一个字符。 一个使用操作符的操作中涉及到的值称为操作值（有时也可称为运算数）。 请阅读常用操作符（第8章）以获取更多关于操作符的信息。 更多操作符将在后续其它文章中介绍。

第35和36行调用了两个内置函数print和println。 从第13行到第26行声明的函数StatRandomNumbers在第33行被调用。 第19行也调用了一个函数 Intn。 这个函数声明在math/rand标准库包中。 请阅读函数声明及函数调用（第9章）以获取更多关于函数声明及函数调用的信息。

（注意，一般`print`和`println`这两个内置函数并不推荐使用。 在正式的项目中，我们应该尽量使用`fmt`标准库包中声明的相应函数。 《Go语言101》只在开始的几篇文章中使用了这两个函数。）

第1行指定了当前源文件所处的包的名称。 一个Go程序的主函数（`main`函数）必须被声明在一个名称为`main`的包中。 第3行引入了`math/rand`标准库包，并以`rand`做为引入名。 在这个包中声明的`Intn`函数将在第19行被调用。 请阅读代码包和包引入（第10章），以获取更多关于代码包和包引入的信息。

表达式、语句和简单语句（第11章）一文中介绍了什么是表达式和语句。特别地，此文列出了所有的简单语句类型。 在Go代码中，各种流程控制代码块中的某些部分必须为简单语句，某些部分必须为表达式。

`StatRandomNumbers`函数的声明体中使用了两个流程控制代码块。 其中一个是`for`循环代码块，它内嵌了另外一个代码块。 另外一个代码块是一个`if-else`条件控制代码块。 请阅读基本流程控制语法（第12章）以获取更多关于流程控制代码块的信息。 更多的特殊的流程控制代码块将在以后的其它文章中介绍。

空行常常用来增加代码的可读性。 上面的程序中也包涵了很多注释，但它们大多是为了Go初学者快速理解的目的而加入的。 我们应该尽量使代码自解释，只在确实需要解释的地方进行注释。

关于代码断行

像很多其它流行编程语言一样，Go也使用一对大括号`{ and }`来形成一个显式代码块。但是在Go代码中，编码样式风格有一些限制。 比如，很多左大括号`{`不能被放到下一行。 如果，上面的`StatRandomNumbers`被修改成如下所示，则上面的示例程序将编译不通过。

```
 1| func StatRandomNumbers(numRands int) (int, int)
 2| { // 编译错误：语法错误
 3|     var a, b int
 4|     for i := 0; i < numRands; i++
 5|     { // 编译错误：语法错误
 6|         if rand.Intn(MaxRand) < MaxRand/2
 7|         { // 编译错误：语法错误
 8|             a = a + 1
 9|         } else {
10|             b++
11|         }
12|     }
13|     return a, b
14| }
```

一些程序员不是很喜欢这些限制。但是这些限制有两个好处：

1. 它们使得Go程序编译得非常快。
2. 它们使得不同的Go程序员编写的代码风格类似，从而一个Go程序员写的代码很容易被另一个程序员看懂。

我们可以阅读代码断行规则（第28章）一文以获取更多关于代码换行规则的细节。在目前，我们最好避免将左大括号放在下一行。 或者说，每行的非空起始字符不能是左大括号（但是，请记住，这不是一个普适的规则）。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

关键字和标识符

此篇文章将介绍Go中的关键字和标识符。

关键字

关键字是一些特殊的用来帮助编译器理解和解析源代码的单词。

截至目前（Go 1.23），Go中共有25个关键字。

```
1| break     default      func    interface  select
2| case      defer        go      map        struct
3| chan      else         goto    package    switch
4| const     fallthrough  if      range      type
5| continue  for          import  return     var
```

这些关键字可以分为四组：

- `const`、`func`、`import`、`package`、`type`和`var`用来声明各种代码元素。
- `chan`、`interface`、`map`和`struct`用做一些组合类型的字面表示中。
- `break`、`case`、`continue`、`default`、`else`、`fallthrough`、`for`、`goto`、`if`、`range`、`return`、`select`和`switch`用在流程控制语句中。详见基本流程控制语法（第12章）。
- `defer`和`go`也可以看作是流程控制关键字，但它们有一些特殊的作用。详见协程和延迟函数调用（第13章）。

这些关键字将在后续文章中得到详细介绍。

标识符

一个标识符是一个以Unicode字母或者_开头并且完全由Unicode字母和Unicode数字组成的单词。

- Unicode字母是定义在Unicode标准（http://www.unicode.org）中的*Lu*、*Ll*、*Lt*、*Lm*和*Lo*分类中的字符。
- Unicode数字是定义在Unicode标准中的*Nd*数字字符分类中的字符。

注意：**关键字不能被用做标识符。**

标识符_是一个特殊字符，它叫做**空标识符**。

以后，我们将知道所有的类型名、变量名、常量名、跳转标签、包名和包的引入名都必须是标识符。

一个由Unicode大写字母（http://www.fileformat.info/info/unicode/category/Lu/list.htm）开头的标识符称为**导出标识符**。 这里***导出***可以被理解为***公开***（public）。 其它（即非Unicode大写字母开头的）标识符称为**非导出标识符**。 ***非导出***可以被理解为***私有***（private）。 截至目前（Go 1.23），东方字符都被视为非导出字符。 **非导出**有时候也被称为***未导出***。

下面是一些合法的导出标识符：

```
 1| Player_9
 2| DoSomething
 3| VERSION
 4| Ĝo
 5| Π
```

下面是一些合法的未导出标识符：

```
 1| _
 2| _status
 3| memStat
 4| book
 5| π
 6| 一个类型
 7| 변수
 8| エラー
```

下面这些不能被用做标识符：

```
 1| // Unicode数字开头
 2| 123
 3| 3apples
 4|
 5| // 含有不符合要求的Unicode字符
 6| a.b
 7| *ptr
 8| $name
 9| a@b.c
10|
11| // 这两个是关键字
12| type
13| range
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

基本类型和它们的字面量表示

类型（type）可以被看作是值（value）的模板，值可以被看作是类型的实例。 这篇文章将介绍内置（或称为预声明的）基本类型和它们字面量的表示形式。 本篇文章不介绍组合类型。

基本内置类型

Go支持如下内置基本类型：

- 一种内置布尔类型： `bool`。
- 11种内置整数类型： `int8`、`uint8`、`int16`、`uint16`、`int32`、`uint32`、`int64`、`uint64`、`int`、`uint`和`uintptr`。
- 两种内置浮点数类型： `float32`和`float64`。
- 两种内置复数类型： `complex64`和`complex128`。
- 一种内置字符串类型： `string`。

内置类型也称为预声明类型。

这17种内置基本类型（type）各自属于一种Go中的类型种类（kind）。 尽管所有的内置基本类型的名称都是非导出标识符（第5章）， 我们可以不用引入任何代码包而直接使用这些类型。

除了`bool`和`string`类型，其它的15种内置基本类型都称为数值类型（整型、浮点数型和复数型）。

Go中有两种内置类型别名（type alias）：

- `byte`是`uint8`的内置别名。 我们可以将`byte`和`uint8`看作是同一个类型。
- `rune`是`int32`的内置别名。 我们可以将`rune`和`int32`看作是同一个类型。

以`u`开头的整数类型称为无符号整数类型。 无符号整数类型的值都是非负的。 一个数值类型名称中的数字表示每个这个类型的值将在内存中占有多少二进制位（以后简称位）。二进制位常称为比特（bit）。 比如，一个`uint8`的值将占有8位。 我们称`uint8`类型的值的尺寸是8位。 因此，最大的`uint8`值是`255`（2^8-1）， 最大的`int8`值是`127`（2^7-1）， 最小的`int8`值是`-128`（-2^7）。

任一个类型的所有值的尺寸都是相同的，所以一个值的尺寸也常称为它的类型的尺寸。

更多的时候，我们使用字节（byte）做为值尺寸的度量单位。 一个字节相当于8个比特。所以`uint32`类型的尺寸为4，即每个`uint32`值占用4个字节。

uintptr、int以及uint类型的值的尺寸依赖于具体编译器实现。 通常地，在64位的架构上，int和uint类型的值是64位的；在32位的架构上，它们是32位的。编译器必须保证uintptr类型的值的尺寸能够存下任意一个内存地址。

一个complex64复数值的实部和虚部都是float32类型的值。 一个complex128复数值的实部和虚部都是float64类型的值。

在内存中，所有的浮点数都使用IEEE-754格式（https://en.wikipedia.org/wiki/IEEE_754）存储。

一个布尔值表示一个真假。在内存中，一个布尔值只有两种可能的状态。 这两种状态使用两个预声明（或称为内置）的常量（false和true）来表示。 关于常量声明，下一篇文章（第7章）将做详细解释。

从逻辑上说，一个字符串值表示一段文本。 在内存中，一个字符串存储为一个字节（byte）序列。 此字节序列体现了此字符串所表示的文本的UTF-8编码形式。 我们可以从Go中的字符串（第19章）一文中获取更多关于字符串的知识。

尽管布尔和字符串类型分类各自只有一种内置类型， 我们可以声明定义更多自定义布尔和字符串类型。 所以，Go代码中可以出现很多布尔和字符串类型（数值类型也同样）。 下面是一个类型声明的例子。 在这些例子中，type是一个关键字。

```
 1| // 一些类型定义声明
 2| type status bool     // status和bool是两个不同的类型
 3| type MyString string // MyString和string是两个不同的类型
 4| type Id uint64       // Id和uint64是两个不同的类型
 5| type real float32    // real和float32是两个不同的类型
 6|
 7| // 一些类型别名声明
 8| type boolean = bool // boolean和bool表示同一个类型
 9| type Text = string  // Text和string表示同一个类型
10| type U8 = uint8     // U8、uint8和 byte表示同一个类型
11| type char = rune    // char、rune和int32表示同一个类型
```

我们将上面定义的real类型和内置类型float32都称为float32类型 （注意这里的第二个float32是一个泛指，而第一个高亮的float32是一个特指）。 同样地，MyString和string都被称为字符串（string）类型， status和bool都被称为布尔（bool）类型。

我们将在Go类型系统概述（第14章）一文中学习到更多关于自定义类型的知识。

零值

每种类型都有一个零值。一个类型的零值可以看作是此类型的默认值。

- 一个布尔类型的零值表示真假中的假。
- 数值类型的零值都是零（但是不同类型的零在内存中占用的空间可能不同）。
- 一个字符串类型的零值是一个空字符串。

基本类型的字面量表示形式

一个值的字面形式称为一个字面量，它表示此值在代码中文字体现形式（和内存中的表现形式相对应）。一个值可能会有很多种字面量形式。

布尔值的字面量形式

Go白皮书没有定义布尔类型值字面量形式。 我们可以将`false`和`true`这两个预声明的具名常量当作布尔类型的字面量形式。 但是，我们应该知道，从严格意义上说，它们不属于字面量。具名常量声明将在下一篇文章中介绍和详细解释。

布尔类型的零值可以使用预声明的`false`来表示。

整数类型值的字面量形式

整数类型值有四种字面量形式：十进制形式（decimal）、八进制形式（octal）、十六进制形式（hex）和二进制形式（binary）。 比如，下面的三个字面量均表示十进制的15：

```
0xF // 十六进制表示（必须使用0x或者0X开头）
0XF

017 // 八进制表示（必须使用0、0o或者0O开头）
0o17
0O17

0b1111 // 二进制表示（必须使用0b或者0B开头）
0B1111

15  // 十进制表示（必须不能用0开头）
```

（注意：二进制形式和以`0o`或`0O`开头的八进制形式从Go 1.13开始才支持。）

下面的程序打印出两个`true`。

```
1| package main
2|
3| func main() {
4|     println(15 == 017) // true
5|     println(15 == 0xF) // true
6| }
```

注意这里的==是一个等于比较操作符。 操作符将在后续的文章常用操作符（第8章）一文中详细解释。

整数类型的零值的字面量一般使用0表示。 当然，00和0x0等也是合法的整数类型零值的字面量形式。

浮点数类型值的字面量形式

一个浮点数的完整十进制字面量形式可能包含一个十进制整数部分、一个小数点、一个十进制小数部分和一个以10为底数的整数指数部分。 整数指数部分由字母e或者E带一个十进制的整数字面量组成（xEn表示x乘以10^n的意思，而xE-n表示x除以10^n的意思）。 常常地，某些部分可以根据情况省略掉。一些例子：

```
1.23
01.23 // == 1.23
.23
1.
// 一个e或者E随后的数值是指数值（底数为10）。
// 指数值必须为一个可以带符号的十进制整数字面量。
1.23e2  // == 123.0
123E2   // == 12300.0
123.E+2 // == 12300.0
1e-1    // == 0.1
.1e0    // == 0.1
0010e-2 // == 0.1
0e+5    // == 0.0
```

从Go 1.13开始，Go也支持另一种浮点数字面量形式：十六进制浮点数字面量。 在一个十六进制浮点数字面量中，

- 一个十六进制浮点数字面量必须以一个以2为底数的整数指数部分。 这样的一个整数指数部分由字母p或者P带一个十进制的整数字面量组成（yPn表示y乘以2^n的意思，而yP-n表示y除以2^n的意思）。
- 和整数的十六进制字面量一样，一个十六进制浮点数字面量也必须使用0x或者0X开头。 和整数的十六进制字面量不同的是，一个十六进制浮点数字面量可以包括一个小数点和一个十六进制小数部分。

一些合法的浮点数的十六进制字面量例子：

```
0x1p-2     // == 1.0/4 = 0.25
0x2.p10    // == 2.0 * 1024 == 2048.0
0x1.Fp+0   // == 1+15.0/16 == 1.9375
0X.8p1     // == 8.0/16 * 2 == 1.0
0X1FFFP-16 // == 0.1249847412109375
```

而下面这几个均是不合法的浮点数的十六进制字面量。

```
0x.p1    // 整数部分表示必须包含至少一个数字
1p-2     // p指数形式只能出现在浮点数的十六进制字面量中
0x1.5e-2 // e和E不能出现在十六进制浮点数字面量的指数部分中
```

注意：下面这个表示是合法的，但是它不是浮点数的十六进制字面量。事实上，它是一个减法算术表达式。其中的e为是十进制中的14，0x15e为一个整数十六进制字面量，-2并不是此整数十六进制字面量的一部分。（算术运算将在后续的文章常用操作符（第8章）一文中详细介绍。）

```
0x15e-2 // == 0x15e - 2 (整数相减表达式)
```

浮点类型的零值的标准字面量形式为0.0。 当然其它很多形式也是合法的，比如0.、.0、0e0和0x0p0等。

虚部字面量形式

一个虚部值的字面量形式由一个浮点数字面量或者一个整数字面量和其后跟随的一个小写的字母i组成。 在Go 1.13之前，如果虚部中i前的部分为一个整数字面量，则其必须为并且总是被视为十进制形式。 一些例子：

```
1.23i
1.i
.23i
123i
0123i   // == 123i（兼容性使然。见下）
1.23E2i // == 123i
1e-1i
011i   // == 11i（兼容性使然。见下）
00011i // == 11i（兼容性使然。见下）
// 下面这几行从Go 1.13开始才能编译通过。
0o11i    // == 9i
0x11i    // == 17i
0b11i    // == 3i
0X.8p-0i // == 0.5i
```

注意：在Go 1.13之前，虚部字面量中字母i前的部分只能为浮点数字面量。 为了兼容老的Go版本，从Go 1.13开始，一些虚部字面量中表现为（不以0o和0O开头的）八进制形式的整数字面量仍被视为浮点数字面量。 比如上例中的011i、0123i和00011i。

虚部字面量用来表示复数的虚部。下面是一些复数值的字面量形式：

```
1 + 2i       // == 1.0 + 2.0i
1. - .1i     // == 1.0 + -0.1i
1.23i - 7.89 // == -7.89 + 1.23i
1.23i        // == 0.0 + 1.23i
```

复数零值的标准字面表示为0.0+0.0i。 当然0i、.0i、0+0i等表示也是合法的。

数值字面表示中使用下划线分段来增强可读性

从Go 1.13开始，下划线_可以出现在整数、浮点数和虚部数字面量中，以用做分段符以增强可读性。 但是要注意，在一个数值字面表示中，一个下划线_不能出现在此字面表示的首尾，并且其两侧的字符必须为（相应进制的）数字字符或者进制表示头。

一些合法和不合法使用下划线的例子：

```
// 合法的使用下划线的例子
6_9           // == 69
0_33_77_22    // == 0337722
0x_Bad_Face   // == 0xBadFace
0X_1F_FFP-16 // == 0X1FFFP-16
0b1011_0111 + 0xA_B.Fp2i

// 非法的使用下划线的例子
_69         // 下划线不能出现在首尾
69_         // 下划线不能出现在首尾
6__9        // 下划线不能相连
0_xBadFace // x不是一个合法的八进制数字
1_.5        // .不是一个合法的十进制数字
1._5        // .不是一个合法的十进制数字
```

rune值的字面量形式

上面已经提到，rune类型是int32类型的别名。 因此，rune类型（泛指）是特殊的整数类型。 一个rune值可以用上面已经介绍的整数类型的字面量形式表示。 另一方面，很多各种整数类型的值也可以用本小节介绍的rune字面量形式来表示。

在Go中，一个rune值表示一个Unicode码点。 一般说来，我们可以将一个Unicode码点看作是一个Unicode字符。 但是，我们也应该知道，有些Unicode字符由多个Unicode码点组成。 每个英文或中文Unicode字符值含有一个Unicode码点。

一个rune字面量由若干包在一对单引号中的字符组成。 包在单引号中的字符序列表示一个Unicode码点值。 rune字面量形式有几个变种，其中最常用的一种变种是将一个rune值对应的Unicode字符直接包在一对单引号中。比如：

```
'a' // 一个英文字符
'π'
'众' // 一个中文字符
```

下面这些rune字面量形式的变种和'a'是等价的 （字符a的Unicode值是97）。

```
'\141'   // 141是97的八进制表示
'\x61'   // 61是97的十六进制表示
'\u0061'
'\U00000061'
```

注意：\之后必须跟随三个八进制数字字符（0-7）表示一个byte值， \x之后必须跟随两个十六进制数字字符（0-9，a-f和A-F）表示一个byte值， \u之后必须跟随四个十六进制数字字符表示一个rune值（此rune值的高四位都为0）， \U之后必须跟随八个十六进制数字字符表示一个rune值。 这些八进制和十六进制的数字字符序列表示的整数必须是一个合法的Unicode码点值，否则编译将失败。

下面这些println函数调用都将打印出true。

```
 1| package main
 2|
 3| func main() {
 4|    println('a' == 97)
 5|    println('a' == '\141')
 6|    println('a' == '\x61')
 7|    println('a' == '\u0061')
 8|    println('a' == '\U00000061')
 9|    println(0x61 == '\x61')
10|    println('\u4f17' == '众')
11| }
```

事实上，在日常编程中，这四种rune字面量形式的变种很少用来表示rune值。 它们多用做字符串的双引号字面量形式中的转义字符（详见下一小节）。

如果一个rune字面量中被单引号包起来的部分含有两个字符， 并且第一个字符是\，第二个字符不是x、 u和U，那么这两个字符将被转义为一个特殊字符。 目前支持的转义组合为：

```
\a   (rune值：0x07) 铃声字符
\b   (rune值：0x08) 退格字符（backspace）
\f   (rune值：0x0C) 换页符（form feed）
\n   (rune值：0x0A) 换行符（line feed or newline）
\r   (rune值：0x0D) 回车符（carriage return）
\t   (rune值：0x09) 水平制表符（horizontal tab）
\v   (rune值：0x0b) 竖直制表符（vertical tab）
\\   (rune值：0x5c) 一个反斜杠（backslash）
\'   (rune值：0x27) 一个单引号（single quote）
```

其中，\n在日常编程中用得最多。

一个例子：

```
 1|    println('\n') // 10
 2|    println('\r') // 13
 3|    println('\'') // 39
 4|
 5|    println('\n' == 10)     // true
 6|    println('\n' == '\x0A') // true
```

rune类型的零值常用 '\000'、'\x00'或'\u0000'等来表示。

字符串值的字面量形式

在Go中，字符串值是UTF-8编码的， 甚至所有的Go源代码都必须是UTF-8编码的。

Go字符串的字面量形式有两种。 一种是解释型字面表示（interpreted string literal，双引号风格）。 另一种是直白字面表示（raw string literal，反引号风格）。 下面的两个字符串表示形式是等价的：

```
// 解释形式
"Hello\nworld!\n\"你好世界\""

// 直白形式
`Hello
world!
"你好世界"`
```

在上面的解释形式（双引号风格）的字符串字面量中，每个\n将被转义为一个换行符，每个\"将被转义为一个双引号字符。 双引号风格的字符串字面量中支持的转义字符和rune字面量基本一致，除了一个例外：双引号风格的字符串字面量中支持\"转义，但不支持\'转义；而rune字面量则刚好相反。

以\、\x、\u和\U开头的rune字面量（不包括两个单引号）也可以出现在双引号风格的字符串字面量中。比如：

```
// 这几个字符串字面量是等价的。
"\141\142\143"
"\x61\x62\x63"
"\x61b\x63"
"abc"

// 这几个字符串字面量是等价的。
"\u4f17\xe4\xba\xba"
      // “众”的Unicode值为4f17，它的UTF-8
      // 编码为三个字节：0xe4 0xbc 0x97。
"\xe4\xbc\x97\u4eba"
      // “人”的Unicode值为4eba，它的UTF-8
      // 编码为三个字节：0xe4 0xba 0xba。
"\xe4\xbc\x97\xe4\xba\xba"
"众人"
```

在UTF-8编码中，一个Unicode码点（rune）可能由1到4个字节组成。 每个英文字母的UTF-8编码只需要一个字节；每个中文字符的UTF-8编码需要三个字节。

直白反引号风格的字面表示中是不支持转义字符的。 除了首尾两个反引号，直白反引号风格的字面表示中不能包含反引号。 为了跨平台兼容性，直白反引号风格的字面表示中的回车符（Unicode码点为0x0D） 将被忽略掉。

字符串类型的零值在代码里用 ""或``表示。

基本数值类型字面量的适用范围

一个数值型的字面量只有在不需要舍入时，才能用来表示一个整数基本类型的值。比如，1.0可以表示任何基本整数类型的值，但1.01却不可以。 当一个数值型的字面量用来表示一个非整数基本类型的值时，舍入（或者精度丢失）是允许的。

每种数值类型有一个能够表示的数值范围。 如果一个字面量超出了一个类型能够表示的数值范围（溢出），则在编译时刻，此字面量不能用来表示此类型的值。

下表是一些例子：

字面表示	此字面表示可以表示哪些类型的值（在编译时刻）

字面表示	此字面表示可以表示哪些类型的值（在编译时刻）
256	除了int8和uint8类型外的所有的基本数值类型。
255	除了int8类型外的所有的基本数值类型。
-123	除了无符号整数类型外的所有的基本数值类型。
123	所有的基本数值类型。
123.000	
1.23e2	
'a'	
1.0+0i	
1.23	所有浮点数和复数基本数值类型。
0x10000000000000000 (16 zeros)	
3.5e38	除了float32和complex64类型外的所有浮点数和复数基本数值类型。
1+2i	所有复数基本数值类型。
2e+308	无。

注意几个溢出的例子：

- 字面量`0x10000000000000000`需要65个比特才能表示，所以在运行时刻，任何基本整数类型都不能精确表示此字面量。
- 在IEEE-754标准中，最大的可以精确表示的float32类型数值为`3.40282346638528859811704183484516925440e+38`，所以`3.5e38`不能表示任何float32和complex64类型的值。
- 在IEEE-754标准中，最大的可以精确表示的float64类型数值为`1.797693134862315708145274237317043567981e+308`，因此`2e+308`不能表示任何基本数值类型的值。
- 尽管`0x10000000000000000`可以用来表示float32类型的值，但是它不能被任何float32类型的值所精确表示。上面已经提到了，当使用字面量来表示非整数基本数值类型的时候，精度丢失是允许的（但溢出是不允许的）。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

常量和变量

这篇文章将介绍常量和变量相关的知识。 类型不确定值、类型推断和值的显式类型转换等概念也将被介绍。

上一章中提到的基本类型的字面量表示（第6章） （除了`false`和`true`）都属于无名常量（unnamed constant），或者叫字面常量（literal constant）。 `false`和`true`是预声明的两个具名常量。本文将介绍如何声明自定义的具名常量。

类型不确定值（untyped value）和类型确定值（typed value）

在Go中，有些值的类型是不确定的。换句话说，有些值的类型有很多可能性。 这些值称为类型不确定值。对于大多数类型不确定值来说，它们各自都有一个默认类型， 除了预声明的`nil`。`nil`是没有默认类型的。 我们在后续的文章中将了解到很多关于`nil`的知识。

与类型不确定值相对应的概念称为类型确定值。

上一章提到的字面常量（无名常量）都属于类型不确定值。 事实上，Go中大多数的类型不确定值都属于字面常量和本文即将介绍的具名常量。 少数类型不确定值包括刚提到的`nil`和以后会逐步接触到的某些操作的布尔返回值。

一个字面（常）量的默认类型取决于它为何种字面量形式：

- 一个字符串字面量的默认类型是预声明的`string`类型。
- 一个布尔字面量的默认类型是预声明的`bool`类型。
- 一个整数型字面量的默认类型是预声明的`int`类型。
- 一个rune字面量的默认类型是预声明的`rune`（亦即`int32`）类型。
- 一个浮点数字面量的默认类型是预声明的`float64`类型。
- 如果一个字面量含有虚部字面量，则此字面量的默认类型是预声明的`complex128`类型。

类型不确定常量的显式类型转换

和很多语言一样，Go也支持类型转换。 一个显式类型转换的形式为`T(v)`，其表示将一个值`v`转换为类型`T`。 编译器将`T(v)`的转换结果视为一个类型为`T`的类型确定值。 当然，对于一个特定的类型`T`，`T(v)`并非对任意的值`v`都合法。

下面介绍的规则同时适用于上一章介绍的字面常量和即将介绍的类型不确定具名常量。

对于一个类型不确定常量值v，有两种情形显式转换T(v)是合法的：

1. v可以表示为（第6章）T类型的一个值。 转换结果为一个类型为T的类型确定常量值。
2. v的默认类型是一个整数类型（int或者rune） 并且T是一个字符串类型。转换T(v)将v看作是一个Unicode码点。 转换结果为一个类型为T的字符串常量。 此字符串常量只包含一个Unicode码点，并且可以看作是此Unicode码点的UTF-8表示形式。 对于不在合法的Unicode码点取值范围内的整数v， 转换结果等同于字符串字面量"\uFFFD"（亦即"\xef\xbf\xbd"）。 0xFFFD是Unicode标准中的（非法码点的）替换字符值。 （但是请注意，今后的Go版本可能只允许rune或者byte整数被转换为字符串（https://github.com/golang/go/issues/3939）。 从Go官方工具链1.15版本开始， go vet命令会对从非rune和非byte整数到字符串的转换做出警告。）

事实上，第二种情形并不要求v必须是一个常量。 如果v是一个常量，则转换结果也是一个常量。 如果v不是一个常量，则转换结果也不是一个常量。

一些合法的转换例子：

```
// 结果为complex128类型的1.0+0.0i。虚部被舍入了。
complex128(1 + -1e-1000i)
// 结果为float32类型的0.5。这里也舍入了。
float32(0.49999999)
// 只要目标类型不是整数类型，舍入都是允许的。
float32(17000000000000000)
float32(123)
uint(1.0)
int8(-123)
int16(6+0i)
complex128(789)

string(65)          // "A"
string('A')         // "A"
string('\u68ee')    // "森"
string(-1)          // "\uFFFD"
string(0xFFFD)      // "\uFFFD"
string(0x2FFFFFFFF) // "\uFFFD"
```

下面是一些非法的转换：

```
int(1.23)     // 1.23不能被表示为int类型值。
uint8(-1)     // -1不能被表示为uint8类型值。
float64(1+2i) // 1+2i不能被表示为float64类型值。

// -1e+1000不能被表示为float64类型值。不允许溢出。
float64(-1e1000)
// 0x10000000000000000做为int值将溢出。
int(0x10000000000000000)

// 字面量65.0的默认类型是float64（不是一个整数类型）。
string(65.0)
// 66+0i的默认类型是complex128（不是一个整数类型）。
string(66+0i)
```

从上面的例子可以看出，一个类型不确定数字值所表示的值可能溢出它的默认类型的表示范围。 比如上例中的`-1e1000`和`0x10000000000000000`。 一个溢出了它的默认类型的表示范围的类型不确定数字值是不能被转换到它的默认类型的（将编译报错）。

注意，有时一个显式转换形式必须被写成`(T)(v)`以免发生歧义。 这种情况多发生在`T`不为一个标识符的时候。

我们以后将在其它章节学到更多的显式类型转换规则。

类型推断介绍

Go支持类型推断（type deduction or type inference）。 类型推断是指在某些场合下，程序员可以在代码中使用一些类型不确定值， 编译器会自动推断出这些类型不确定值在特定情景下应被视为某些特定类型的值。

在Go代码中，如果某处需要一个特定类型的值并且一个类型不确定值可以表示为此特定类型的值， 则此类型不确定值可以使用在此处。Go编译器将此类型不确定值视为此特定类型的类型确定值。 这种情形常常出现在运算符运算、函数调用和赋值语句中。

有些场景对某些类型不确定值并没有特定的类型要求。在这种情况下，Go编译器将这些类型不确定值视为它们各自的默认类型的类型确定值。

上述两条类型推断规则可以被视为隐式转换规则。

本文下面的章节将展示一些类型推断的例子。 后续其它文章将会展示更多类型推断的例子和规则。

（具名）常量声明（constant declaration）

和无名字面常量一样，具名常量也必须都是布尔、数字或者字符串值。 在Go中，关键字const用来声明具名常量。 下面是一些常量声明的例子。

```
 1| package main
 2|
 3| // 声明了两个单独的具名常量。（是的，
 4| // 非ASCII字符可以用做标识符。）
 5| const π = 3.1416
 6| const Pi = π // 等价于：const Pi = 3.1416
 7|
 8| // 声明了一组具名常量。
 9| const (
10|     No         = !Yes
11|     Yes        = true
12|     MaxDegrees = 360
13|     Unit       = "弧度"
14| )
15|
16| func main() {
17|     // 声明了三个局部具名常量。
18|     const DoublePi, HalfPi, Unit2 = π * 2, π * 0.5, "度"
19| }
```

Go白皮书把上面每行含有一个等号=的语句称为一个常量描述（constant specification）。 每个const关键字对应一个常量声明。一个常量声明中可以有若干个常量描述。 上面的例子中含有4个常量声明。除了第3个，其它的常量声明中都各自只有一个常量描述。 第3个常量声明中有4个常量描述。

在上面的例子中，符号*是一个乘法运算符， 符号!是一个布尔取否运算符。 运算符将在下一篇文章（第8章）中详述。

常量声明中的等号=表示“绑定”而非“赋值”。 每个常量描述将一个或多个字面量绑定到各自对应的具名常量上。 或者说，每个具名常量其实代表着一个字面常量。

在上面的例子中，具名常量π和Pi都绑定到（或者说代表着）字面常量3.1416。这两个具名常量可以在程序代码中被多次使用，从而有效避免了字面常量3.1416在代码中出现在多处。 如果字面常量3.1416在代码中出现在多处， 当我们以后欲将3.1416改为3.14的时候，所有出现在代码中的3.1416都得逐个修改。 有了具名常量的帮助，我们只需修改对应常量描述中的3.1416即可。 这是常量声明的主要作用。当然常量声明也可常常增加代码的可读性（代码即注释）。

以后，我们使用非常量这一术语表示不是常量的值。 下一节将要介绍的变量就属于非常量。

注意，常量可以直接声明在包中，也可以声明在函数体中。 声明在函数体中的常量称为局部常量（local constant），直接声明在包中的常量称为包级常量（package-level constant）。 包级常量也常常被称为全局常量。

包级常量声明中的常量描述的顺序并不重要。比如在上面的例子中， 常量描述`No`和`Yes`的顺序可以掉换一下。

上面例子中声明的所有常量都是类型不确定的。 它们各自的默认类型和它们各自代表的字面量的默认类型是一样的。

类型确定具名常量

我们可以在声明一些常量的时候指定这些常量的确切类型。 这样声明的常量称为类型确定具名常量。 在下面这个例子中，所有这4个声明的常量都是类型确定的。 `X`和`Y`的类型都是`float32`， `A`和`B`的类型都是`int64`。

```
1| const X float32 = 3.14
2|
3| const (
4|     A, B int64   = -3, 5
5|     Y    float32 = 2.718
6| )
```

如果一个常量描述中包含多个类型确定常量，则这些常量的类型必然是一样的， 比如上例中的`A`和`B`。

我们也可以使用显式类型转换来声明类型确定常量。 下面的例子和上面的例子是完全等价的。

```
1| const X = float32(3.14)
2|
3| const (
4|     A, B = int64(-3), int64(5)
5|     Y    = float32(2.718)
6| )
```

欲将一个字面常量绑定到一个类型确定具名常量上，此字面常量必须能够表示为此常量的确定类型的值。 否则，编译将报错。

```
1| const a uint8 = 256             // error: 256溢出uint8
2| const b = uint8(255) + uint8(1) // error: 256溢出uint8
```

```
3| const c = int8(-128) / int8(-1) // error: 128溢出int8
4| const MaxUint_a = uint(^0)      // error: -1溢出uint
5| const MaxUint_b uint = ^0       // error: -1溢出uint
```

在上面的例子中，符号^为位反运算符，符号+为加法运算符，符号/为除法运算符。

下面这个类型确定常量声明在64位的操作系统上是合法的，但在32位的操作系统上是非法的。 因为一个uint值在32位操作系统上的尺寸是32位， (1 << 64) - 1将溢出uint。（这里，符号<<为左移位运算符。）

```
1| const MaxUint uint = (1 << 64) - 1
```

那么如何声明一个代表着最大uint值的常量呢？ 我们可以用下面这个常量声明来替换上面这个。下面这个声明在64位和32位的操作系统上都是合法的。

```
1| const MaxUint = ^uint(0)
```

类似地，我们可以使用下面这个常量声明来声明一个具名常量来表示最大的int值。（这里，符号>>为右移位运算符。）

```
1| const MaxInt = int(^uint(0) >> 1)
```

使用类似的方法，我们可以声明一个常量来表示当前操作系统的位数，或者检查当前操作系统是32位的还是64位的。

```
1| const NativeWordBits = 32 << (^uint(0) >> 63) // 64 or 32
2| const Is64bitOS = ^uint(0) >> 63 != 0
3| const Is32bitOS = ^uint(0) >> 32 == 0
```

这里，符号!=和==分别为不等于和等于比较运算符。

常量声明中的自动补全

在一个包含多个常量描述的常量声明中，除了第一个常量描述，其它后续的常量描述都可以只包含标识符列表部分。 Go编译器将通过照抄前面最紧挨的一个完整的常量描述来自动补全不完整的常量描述。 比如，在编译阶段，编译器会将下面的代码

```
1| const (
2|     X float32 = 3.14
3|     Y                // 这里必须只有一个标识符
4|     Z                // 这里必须只有一个标识符
5|
6|     A, B = "Go", "language"
```

```
 7|    C, _
 8|    // 上一行中的空标识符是必需的（如果
 9|    // 上一行是一个不完整的常量描述的话）。
10| )
```

自动补全为

```
 1| const (
 2|    X float32 = 3.14
 3|    Y float32 = 3.14
 4|    Z float32 = 3.14
 5|
 6|    A, B = "Go", "language"
 7|    C, _ = "Go", "language"
 8| )
```

在常量声明中使用iota

iota是Go中预声明（内置）的一个特殊的具名常量。 iota被预声明为0，但是它的值在编译阶段并非恒定。 当此预声明的iota出现在一个常量声明中的时候，它的值在第n个常量描述中的值为n（从0开始）。 所以iota只对含有多个常量描述的常量声明有意义。

iota和常量描述自动补全相结合有的时候能够给Go编程带来很大便利。 比如，下面是一个使用了这两个特性的例子。 请阅读代码注释以了解清楚各个常量被绑定的值。

```
 1| package main
 2|
 3| func main() {
 4|    const (
 5|       k = 3 // 在此处，iota == 0
 6|
 7|       m float32 = iota + .5 // m float32 = 1 + .5
 8|       n                     // n float32 = 2 + .5
 9|
10|       p = 9             // 在此处，iota == 3
11|       q = iota * 2      // q = 4 * 2
12|       _                 // _ = 5 * 2
13|       r                 // r = 6 * 2
14|       s, t = iota, iota // s, t = 7, 7
15|       u, v              // u, v = 8, 8
16|       _, w              // _, w = 9, 9
17|    )
```

```
18|
19|    const x = iota // x = 0 （iota == 0）
20|    const (
21|       y = iota // y = 0 （iota == 0）
22|       z        // z = 1
23|    )
24|
25|    println(m)             // +1.500000e+000
26|    println(n)             // +2.500000e+000
27|    println(q, r)          // 8 12
28|    println(s, t, u, v, w) // 7 7 8 8 9
29|    println(x, y, z)       // 0 0 1
30| }
```

上面的例子只是展示了一下如何使用`iota`。 在实际编程中，我们应该用有意义的方式使用之。比如：

```
 1| const (
 2|    Failed = iota - 1 // == -1
 3|    Unknown           // == 0
 4|    Succeeded         // == 1
 5| )
 6|
 7| const (
 8|    Readable = 1 << iota // == 1
 9|    Writable             // == 2
10|    Executable           // == 4
11| )
```

在上面这段代码中，`-`是一个减法运算符。

变量声明和赋值操作语句

变量可以被看作是在运行时刻存储在内存中并且可以被更改的具名的值。

所有的变量值都是类型确定值。当声明一个变量的时候，我们必须在代码中给编译器提供足够的信息来让编译器推断出此变量的确切类型。

在一个函数体内声明的变量称为局部变量。 在任何函数体外声明的变量称为包级或者全局变量。

Go语言有两种变量声明形式。一种称为标准形式，另一种称为短声明形式。 短声明形式只能用来声明局部变量。

标准变量声明形式

每条标准变量声明形式语句起始于一个var关键字。 每个var关键字跟随着一个变量名。 每个变量名必须为一个标识符（第5章）。

下面是几条完整形式的标准变量声明语句。 这些声明确地指定了被声明的变量的类型和初始值。

```
1| var lang, website string = "Go", "https://golang.org"
2| var compiled, dynamic bool = true, false
3| var announceYear int = 2009
```

我们可以看到，和常量声明一样，多个同类型的变量可以在一条语句中被声明。

完整形式的标准变量声明使用起来有些罗嗦，因此很少在日常Go编程中使用。 在日常Go编程中，另外两种变种形式用得更广泛一些。 一种变种形式省略了变量类型（但仍指定了变量的初始值），这时编译器将根据初始值的字面量形式来推断出变量的类型。 另一种变种形式省略了初始值（但仍指定了变量类型），这时编译器将使用变量类型的零值做为变量的初始值。

下面是一些第一种变种形式的用例。在这些用例中，如果一个初始值是一个类型确定值，则对应声明的变量的类型将被推断为此初始值的类型； 如果一个初始值是一个类型不确定值，则对应声明的变量的类型将被推断为此初始值的默认类型。 注意在这种变种中，同时声明的多个变量的类型可以不一样。

```
1| // 变量lang和dynamic的类型将被推断为内置类型string和bool。
2| var lang, dynamic = "Go", false
3|
4| // 变量compiled和announceYear的类型将被推断
5| // 为内置类型bool和int。
6| var compiled, announceYear = true, 2009
7|
8| // 变量website的类型将被推断为内置类型string。
9| var website = "https://golang.org"
```

上例中的类型推断可以被视为隐式类型转换。

下例展示了几个省略了初始值的标准变量声明。每个声明的变量的初始值为它们各自的类型的零值。

```
1| var lang, website string      // 两者都被初始化为空字符串。
2| var interpreted, dynamic bool // 两者都被初始化为false。
3| var n int                     // 被初始化为0。
```

和常量声明一样，多个变量可以用一对小括号组团在一起被声明。

```
 1| var (
 2|    lang, bornYear, compiled     = "Go", 2007, true
 3|    announceAt, releaseAt    int = 2009, 2012
 4|    createdBy, website       string
 5| )
```

上面这个变量声明语句已经被`go fmt`命令格式化过了。 这个变量声明语句包含三个变量描述（variable specification）。

一般来说，将多个相关的变量声明在一起将增强代码的可读性。

纯赋值语句

在上面展示的变量声明的例子中，等号`=`表示赋值。 一旦一个变量被声明之后，它的值可以被通过纯赋值语句来修改。 多个变量可以同时在一条赋值语句中被修改。

一个赋值语句等号左边的表达式必须是一个可寻址的值、一个映射元素或者一个空标识符。 内存地址（以及指针）和映射将在以后的文章中介绍。

常量是不可改变的（不可寻址的），所以常量不能做为目标值出现在纯赋值语句的左边，而只能出现在右边用做源值。 变量既可以出现在纯赋值语句的左边用做目标值，也可以出现在右边用做源值。

空标识符也可以出现在纯赋值语句的左边，表示不关心对应的目标值。 空标识符不可被用做源值。

一个包含了很多（合法或者不合法的）纯赋值语句的例子：

```
 1| const N = 123
 2| var x int
 3| var y, z float32
 4|
 5| N = 789 // error: N是一个不可变量
 6| y = N   // ok: N被隐式转换为类型float32
 7| x = y   // error: 类型不匹配
 8| x = N   // ok: N被隐式转换为类型int
 9| y = x   // error: 类型不匹配
10| z = y   // ok
11| _ = y   // ok
12|
13| z, y = y, z              // ok
14| _, y = y, z              // ok
15| z, _ = y, z              // ok
16| _, _ = y, z              // ok
17| x, y = 69, 1.23          // ok
```

```
18| x, y = y, x                 // error: 类型不匹配
19| x, y = int(y), float32(x) // ok
```

上例中的最后一行使用了显式类型转换，否则此赋值（见倒数第二行）将不合法。数字非常量值的类型转换规则将在后边的章节介绍。

Go不支持某些其它语言中的连等语法。下面的赋值语句在Go中是不合法的。

```
 1| var a, b int
 2| a = b = 123 // 语法错误
```

短变量声明形式

我们也可以用短变量声明形式来声明一些局部变量。比如下例：

```
 1| package main
 2|
 3| func main() {
 4|     // 变量lang和year都为新声明的变量。
 5|     lang, year := "Go language", 2007
 6|
 7|     // 这里，只有变量createdBy是新声明的变量。
 8|     // 变量year已经在上面声明过了，所以这里仅仅
 9|     // 改变了它的值，或者说它被重新声明了。
10|     year, createdBy := 2009, "Google Research"
11|
12|     // 这是一个纯赋值语句。
13|     lang, year = "Go", 2012
14|
15|     print(lang, "由", createdBy, "发明")
16|     print("并发布于", year, "年。")
17|     println()
18| }
```

每个短声明语句中必须至少有一个新声明的变量。

从上面的例子中，我们可以看到短变量声明形式和标准变量声明形式有几个显著的区别：

1. 短声明形式不包含 var 关键字，并且不能指定变量的类型。
2. 短变量声明中的赋值符号必须为 := 。
3. 在一个短声明语句的左侧，已经声明过的变量和新声明的变量可以共存。 但在一个标准声明语句中，所有出现在左侧的变量必须都为新声明的变量。

注意，相对于纯赋值语句，目前短声明语句有一个限制：出现在一个短声明左侧的项必须都为纯标识符。 以后我们将学习到在纯赋值语句的左边可以出现结构体值的字段，指针的解引用和容器类型值的元素索引项等。 但是这些项不能出现在一个变量短声明语句的左边。

关于“赋值”这个术语

以后，当“赋值”这个术语被提到的时候，它可以指一个纯赋值、一个短变量声明或者一个初始值未省略的标准变量声明。 事实上，一个更通用的定义包括后续文章将要介绍的函数传参（第9章）。

当`y = x`是一条合法的赋值语句时，我们可以说`x`可以被赋给`y`。 假设`y`的类型为`Ty`，有时为了叙述方便，我们也可以说`x`可以被赋给类型`Ty`。

一般来说，如果`x`可以被赋给`y`，则`y`应该是可修改的，并且`x`和`y`的类型相同或者`x`可以被隐式转换到`y`的类型。 当然，`y`也可以是空标识符`_`。

每个局部声明的变量至少要被有效使用一次

注意，当使用目前的主流Go编译器编译Go代码时，一个局部变量被声明之后至少要被有效使用一次，否则编译器将报错。 包级变量无此限制。 如果一个变量总是被当作赋值语句中的目标值，那么我们认为这个变量没有被有效使用过。

下面这个例子编译不通过。

```
 1| package main
 2|
 3| var x, y, z = 123, true, "foo" // 包级变量
 4|
 5| func main() {
 6|     var q, r = 789, false
 7|     r, s := true, "bar"
 8|     r = y // r没有被有效使用。
 9|     x = q // q被有效使用了。
10| }
```

当编译上面这个程序的时候，编译器将报错（这个程序代码存在一个名为`example-unused.go`的文件中）：

```
./example-unused.go:6:6: r declared and not used
./example-unused.go:7:16: s declared and not used
```

避免编译器报错的方法很简单，要么删除相关的变量声明，要么像下面这样，将未曾有效使用过的变量（这里是r和s）赋给空标识符。

```
 1| package main
 2|
 3| var x, y, z = 123, true, "foo"
 4|
 5| func main() {
 6|     var q, r = 789, false
 7|     r, s := true, "bar"
 8|     r = y
 9|     x = q
10|
11|     _, _ = r, s // 将r和s做为源值使用一次。
12| }
```

若干包级变量在声明时刻的依赖关系将影响它们的初始化顺序

下面这个例子中的声明的变量的初始化顺序为`y = 5`、`c = y`、`b = c+1`、`a = b+1`、`x = a+1`。

```
1| var x, y = a+1, 5          // 8 5
2| var a, b, c = b+1, c+1, y // 7 6 5
```

包级变量在初始化的时候不能相互依赖。比如，下面这个变量声明语句编译不通过。

```
1| var x, y = y, x
```

值的可寻址性

在Go中，有些值是可以被寻址的。上面已经提到所有变量都是可以寻址的，所有常量都是不可被寻址。 我们可以从后面的指针（第15章）一文了解更多关于内存地址和指针的知识。

非常量数字值相关的显式类型转换规则

在Go中，两个类型不一样的基本类型（第6章）值是不能相互赋值的。 我们必须使用显式类型转换将一个值转换为另一个值的类型之后才能进行赋值。

前面某节已经提到了整数（不论常量还是非常量）都可以被显式转换为字符串类型。 这里再介绍两个不同类型数字值之间的转换规则。

- 一个非常量浮点数和整数可以显式转换到其它任何一个浮点数和整数类型。
- 一个非常量复数可以显式转换到其它任何一个复数类型。

上面已经提到，常量数字值的类型转换不能溢出。此规则不适用于非常量数字值的类型转换。 非常量数字值的类型转换中，溢出是允许的。 另外当将一个浮点数非常量值（比如一个变量）转换为一个整数类型的时候，舍入（或者精度丢失）也是允许的。 具体规则如下：

- 当从一个比特位数多的整数类型的非常量整数值向一个比特位数少的整数类型转换的时候，高位的比特将被舍弃，低位的比特将被保留。我们称这种处理方式为截断（truncated）。
- 当从一个非常量的浮点数向一个整数类型转换的时候，浮点数的小数部分将被舍弃（向零靠拢）。
- 当从一个非常量整数或者浮点数向一个浮点数类型转换的时候，精度丢失是可以发生的。
- 当从一个非常量复数向另一个复数类型转换的时候，精度丢失也是可以发生的。
- 当一个显式转换涉及到非常量浮点数或者复数数字值时，如果源值溢出了目标类型的表示范围，则转换结果取决于具体编译器实现（即行为未定义）。

在下面的例子中，第*7*行和第*15*行的隐式转换是不允许的，第*5*行和第*14*行的显式转换也是不允许的。

```
 1| const a = -1.23
 2| // 变量b的类型被推断为内置类型float64。
 3| var b = a
 4| // error: 常量1.23不能被截断舍入到一个整数。
 5| var x = int32(a)
 6| // error: float64类型值不能被隐式转换到int32。
 7| var y int32 = b
 8| // ok: z == -1，变量z的类型被推断为int32。
 9| //     z的小数部分将被舍弃。
10| var z = int32(b)
11|
12| const k int16 = 255
13| var n = k             // 变量n的类型将被推断为int16。
14| var f = uint8(k + 1) // error: 常量256溢出了uint8。
15| var g uint8 = n + 1  // error: int16值不能隐式转换为uint8。
16| var h = uint8(n + 1) // ok: h == 0，变量h的类型为uint8。
17|                      // (n+1)溢出uint8，所以只有低8位
18|                      // bits（都为0）被保留。
```

第3行的隐式转换中，a被转换为它的默认类型（float64）；因此b的类型被推断为float64。

变量和常量的作用域

在Go中，我们可以使用一对大括号来显式形成一个（局部）代码块。一个代码块可以内嵌另一个代码块。 最外层的代码块称为包级代码块。 一个声明在一个内层代码块中的常量或者变量将遮挡另一个外层代码块中声明的同名变量或者常量。 比如，下面的代码中声明了3个名为x的变量。 内层的x将遮挡外层的x， 从而外层的x在内层的x声明之后在内层中将不可见。

```
 1| package main
 2|
 3| const y = 70
 4| var x int = 123 // 包级变量
 5|
 6| func main() {
 7|    // 此x变量遮挡了包级变量x。
 8|    var x = true
 9|
10|    // 一个内嵌代码块。
11|    {
12|       x, y := x, y-10 // 这里，左边的x和y均为新声明
13|                        // 的变量。右边的x为外层声明的
14|                        // bool变量。右边的y为包级变量。
15|
16|       // 在此内层代码块中，从此开始，
17|       // 刚声明的x和y将遮挡外层声明x和y。
18|
19|       x, z := !x, y/10 // z是一个新声明的变量。
20|                         // x和y是上一句中声明的变量。
21|       println(x, y, z) // false 60 6
22|    }
23|    println(x) // true
24|    println(y) // 70 （包级变量y从未修改）
25|    /*
26|    println(z) // error: z未定义。
27|               // z的作用域仅限于上面的最内层代码块。
28|    */
29| }
```

刚提到的作用域是指一个标识符的可见范围。 一个包级变量或者常量的作用域为其所处于的整个代码包。 一个局部变量或者常量的作用域开始于此变量或者常量的声

明的下一行，结束于最内层包含此变量或者常量的声明语句的代码块的结尾。 这解释了为什么上例中的`println(z)`将编译不通过。

后面的代码块和作用域（第32章）一文将详述代码块和标识符的作用域。

更多关于常量声明

一个类型不确定常量所表示的值可以溢出其默认类型

比如，下例中的三个类型不确定常量均溢出了它们各自的默认类型，但是此程序编译和运行都没问题。

```
 1| package main
 2|
 3| // 三个类型不确定常量。
 4| const n = 1 << 64           // 默认类型为int
 5| const r = 'a' + 0x7FFFFFFF // 默认类型为rune
 6| const x = 2e+308            // 默认类型为float64
 7|
 8| func main() {
 9|     _ = n >> 2
10|     _ = r - 0x7FFFFFFF
11|     _ = x / 2
12| }
```

但是下面这个程序编译不通过，因为三个声明的常量为类型确定常量。

```
 1| package main
 2|
 3| // 三个类型确定常量。
 4| const n int = 1 << 64           // error: 溢出int
 5| const r rune = 'a' + 0x7FFFFFFF // error: 溢出rune
 6| const x float64 = 2e+308        // error: 溢出float64
 7|
 8| func main() {}
```

每个常量标识符将在编译的时候被其绑定的字面量所替代

常量声明可以看作是增强型的C语言中的`#define`宏。 在编译阶段，所有的标识符将被它们各自绑定的字面量所替代。

如果一个运算中的所有运算数都为常量，则此运算的结果也为常量。或者说，此运算将在编译阶段就被估值。 下一篇文章将介绍Go中的常用运算符（第8章）。

一个例子：

```
 1| package main
 2|
 3| const X = 3
 4| const Y = X + X
 5| var a = X
 6|
 7| func main() {
 8|     b := Y
 9|     println(a, b, X, Y)
10| }
```

上面这段程序代码将在编译阶段被重写为下面这样：

```
 1| package main
 2|
 3| var a = 3
 4|
 5| func main() {
 6|     b := 6
 7|     println(a, b, 3, 6)
 8| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

运算操作符

本文将介绍适用于基本类型值的各种运算操作符。

关于本文中的内容和一些解释

本文只介绍算术运算符、位运算符、比较运算符、布尔运算符和字符串衔接运算符。 这些运算符要么是二元的（需要两个操作数），要么是一元的（需要一个操作数）。 所有这些运算符运算都只返回一个结果。操作数常常也称为操作值。

本文中的解释不追求描述的完全准确性。 比如，当我们说一个二元运算符运算需要其涉及的两个操作数类型必须一样的时，这指：

- 如果这两个操作数都是类型确定值，则它们的类型必须相同，或者其中一个操作数可以被隐式转换到另一个操作数的类型。
- 如果其中只有一个操作数是类型确定的，则要么另外一个类型不确定操作数可以表示为此类型确定操作数的类型的值，要么此类型不确定操作数的默认类型的任何值可以被隐式转换到此类型确定操作数的类型。
- 如果这两个操作数都是类型不确定的，则它们必须同时都为两个布尔值，同时都为两个字符串值，或者同时都为两个基本数字值。

类似的，当我们说一个运算符（一元或者二元）运算要求其涉及的某个操作数的类型必须为某个特定类型时，这指：

- 如果这个操作数是类型确定的，则它的类型必须为所要求的特定类型，或者此操作数可以被隐式转换为所要求的特定类型。
- 如果一个操作数是类型不确定的，则要么此操作数可以表示为所要求的特定类型值，要么此操作数的默认类型的任何值可以被隐式转换为所要求的特定类型。

常量表达式

在继续下面的章节之前，我们需要知道什么叫常量表达式和关于常量表达式估值的一个常识。 表达式的概念将在表达式和语句（第11章）一文中得到解释。 目前我们只需知道本文中所提及的大多数运算都属于表达式。 当一个表达式中涉及到的所有操作数都是常量时，此表达式称为一个常量表达式。 一个常量表达式的估值是在编译阶段进行的。一个常量表达式的估值结果依然是一个常量。 如果一个表达式中涉及到的操作数中至少有一个不为常量，则此表达式称为非常量表达式。

算术运算符

Go支持五个基本二元算术运算符：

字面形式	名称	对两个运算数的要求
+	加法	两个运算数的类型必须相同并且为基本数值类型。
-	减法	
*	乘法	
/	除法	
%	余数	两个运算数的类型必须相同并且为基本整数数值类型。

Go支持六种位运算符（也属于算术运算）：

字面形式	名称	对两个操作数的要求以及机制解释
&	位与	两个操作数的类型必须相同并且为基本整数数值类型。 机制解释（下标2表明一个字面量为二进制）： • 1100_2 & 1010_2 得到 1000_2 • 1100_2 | 1010_2 得到 1110_2 • 1100_2 ^ 1010_2 得到 0110_2 • 1100_2 &^ 1010_2 得到 0100_2
|	位或	
^	（位）异或	
&^	清位	
<<	左移位	左操作数必须为一个整数，右操作数也必须为一个整数（如果它是一个常数，则它必须非负），但它们的类型可以不同。（注意：在Go 1.13之前，右操作数必须为一个无符号整数类型的类型确定值或者一个可以表示成uint值的类型不确定（第7章）常数值。） 一个负右操作数（非常数）将在运行时刻造成一个恐慌。 机制解释： • 1100_2 << 3 得到 1100000_2（低位补零） • 1100_2 >> 3 得到 1_2（低位被舍弃） 注意，在右移运算中，左边空出来的位（即高位）全部用左操作数的最高位（即正负号位）填充。 比如如果左操作数-128

字面形式	名称	对两个操作数的要求以及机制解释
>>	右移位	的类型为int8（二进制表示以及机制解释），则 10000000_2 >> 2的二进制补码结果为 11100000_2（即-32）。

Go也支持三个一元算术运算符：

字面形式	名称	解释
+	取正数	+n等价于0 + n.
−	取负数	-n等价于0 - n.
^	位反（或位补）	^n等价于m ^ n，其中m和n同类型并且它的二进制表示中所有比特位均为1。 比如如果n的类型为int8，则m的值为-1；如果n的类型为uint8，则m的值为255。

注意：

- 在很多其它流行语言中，位反运算符是用~表示的。
- 和一些其它流行语言一样，加号运算符+也可用做字符串衔接运算符（见下）。
- 和C及C++语言一样，*除了可以当作乘号运算符，它也可以用做指针解引用运算符； &除了可以当作位与运算符，它也可以用做取地址运算符。 后面的指针（第15章）一文将详解内存地址和指针类型。
- 和Java不一样，Go支持无符号数，所以Go不需要无符号右移运算符>>>。
- Go不支持幂运算符， 我们必须使用math标准库包中的Pow函数来进行幂运算。 下一篇文章将详解包和包引入（第10章）。
- 清位运算符&^是Go中特有的一个运算符。 m &^ n等价于m & (^n)。

一些运算符的使用示例：

```
 1| func main() {
 2|     var (
 3|         a, b float32 = 12.0, 3.14
 4|         c, d int16   = 15, -6
 5|         e   uint8    = 7
 6|     )
 7|
 8|     // 这些行编译没问题。
 9|     _ = 12 + 'A' // 两个类型不确定操作数（都为数值类型）
10|     _ = 12 - a   // 12将被当做a的类型（float32）使用。
11|     _ = a * b    // 两个同类型的类型确定操作数。
12|     _ = c % d
```

```
13|    _, _ = c + int16(e), uint8(c) + e
14|    _, _, _, _ = a / b, c / d, -100 / -9, 1.23 / 1.2
15|    _, _, _, _ = c | d, c & d, c ^ d, c &^ d
16|    _, _, _, _ = d << e, 123 >> e, e >> 3, 0xF << 0
17|    _, _, _, _ = -b, +c, ^e, ^-1
18|
19|    // 这些行编译将失败。
20|    _ = a % b   // error: a和b都不是整数
21|    _ = a | b   // error: a和b都不是整数
22|    _ = c + e   // error: c和e的类型不匹配
23|    _ = b >> 5  // error: b不是一个整数
24|    _ = c >> -5 // error: -5不是一个无符号整数
25|
26|    _ = e << uint(c) // 编译没问题
27|    _ = e << c       // 从Go 1.13开始，此行才能编译过
28|    _ = e << -c      // 从Go 1.13开始，此行才能编译过。
29|                     // 将在运行时刻造成恐慌。
30|    _ = e << -1      // error: 右操作数不能为负（常数）
31| }
```

关于溢出

上一篇文章提到了

- 一个类型确定数字型常量所表示的值是不能溢出它的类型的表示范围的。
- 一个类型不确定数字型常量所表示的值是可以溢出它的默认类型的表示范围的。 当一个类型不确定数字常量值溢出它的默认类型的表示范围时，此数值不会被截断（亦即回绕）。
- 将一个非常量数字值转换为其它数字类型时，此非常量数字值可以溢出转化结果的类型。 在此转换中，当溢出发生时，转化结果为此非常量数字值的截断（亦即回绕）表示。

对于一个算数运算的结果，上述规则同样适用。

示例：

```
1| // 结果为非常量
2| var a, b uint8 = 255, 1
3| var c = a + b  // c==0。a+b是一个非常量表达式，
4|                // 结果中溢出的高位比特将被截断舍弃。
5| var d = a << b // d == 254。同样，结果中溢出的
6|                // 高位比特将被截断舍弃。
7|
8| // 结果为类型不确定常量，允许溢出其默认类型。
```

```
 9| const X = 0x1FFFFFFFF * 0x1FFFFFFFF // 没问题，尽管X溢出
10| const R = 'a' + 0x7FFFFFFF          // 没问题，尽管R溢出
11|
12| // 运算结果或者转换结果为类型确定常量
13| var e = X                 // error: X溢出int。
14| var h = R                 // error: R溢出rune。
15| const Y = 128 - int8(1)  // error: 128溢出int8。
16| const Z = uint8(255) + 1 // error: 256溢出uint8。
```

关于算术运算的结果

除了移位运算，对于一个二元算术运算，

- 如果它的两个操作数都为类型确定值，则此运算的结果也是一个和这两个操作数类型相同的类型确定值。
- 如果只有一个操作数是类型确定的，则此运算的结果也是一个和此类型确定操作数类型相同的类型确定值。 另一个类型不确定操作数的类型将被推断为（或隐式转换为）此类型确定操作数的类型。
- 如果它的两个操作数均为类型不确定值，则此运算的结果也是一个类型不确定值。 在运算中，两个操作数的类型将被设想为它们的默认类型中一个（按照此优先级来选择：`complex128`高于`float64`高于`rune`高于`int`）。 结果的默认类型同样为此设想类型。 比如，如果一个类型不确定操作数的默认类型为`int`，另一个类型不确定操作数的默认类型为`rune`， 则前者的类型在运算中也被视为`rune`，运算结果为一个默认类型为`rune`的类型不确定值。

对于移位运算，结果规则有点小复杂。首先移位运算的结果肯定都是整数。

- 如果左操作数是一个类型确定值（则它的类型必定为整数），则此移位运算的结果也是一个和左操作数类型相同的类型确定值。
- 如果左操作数是一个类型不确定值**并且右操作数是一个常量**，则左操作数将总是被视为一个整数。 如果它的默认类型不是一个整数（`rune`或`int`），则它的默认类型将被视为`int`。 此移位运算的结果也是一个类型不确定值并且它的默认类型和左操作数的默认类型一致。
- 如果左操作数是一个类型不确定值**并且右操作数是一个非常量**，则左操作数将被首先转化为运算结果的期待设想类型。 如果期待设想类型并没有被指定，则左操作数的默认类型将被视为它的期待设想类型。 如果此期待设想类型不是一个内置整数类型，则编译报错。 当然最终运算结果是一个类型为此期待设想类型的类型确定值。

一些非移位算术运算的例子：

```
 1| func main() {
 2|     // 三个类型不确定常量。它们的默认类型
```

```
 3|    // 分别为：int、rune和complex64.
 4|    const X, Y, Z = 2, 'A', 3i
 5|
 6|
 7|    var a, b int = X, Y // 两个类型确定值
 8|
 9|    // 变量d的类型被推断为Y的默认类型：rune（亦即int32）。
10|    d := X + Y
11|    // 变量e的类型被推断为a的类型：int。
12|    e := Y - a
13|    // 变量f的类型和a及b的类型一样：int。
14|    f := a * b
15|    // 变量g的类型被推断为Z的默认类型：complex64。
16|    g := Z * Y
17|
18|    // 2 65 (+0.000000e+000+3.000000e+000i)
19|    println(X, Y, Z)
20|    // 67 63 130 (+0.000000e+000+1.950000e+002i)
21|    println(d, e, f, g)
22| }
```

一个移位算术运算的例子：

```
 1| const N = 2
 2| // A == 12，它是一个默认类型为int的类型不确定值。
 3| const A = 3.0 << N
 4| // B == 12，它是一个类型为int8的类型确定值。
 5| const B = int8(3.0) << N
 6|
 7| var m = uint(32)
 8| // 下面的三行是相互等价的。
 9| var x int64 = 1 << m  // 1的类型将被设想为int64，而非int
10| var y = int64(1 << m) // 同上
11| var z = int64(1) << m // 同上
12|
13| // 下面这行编译不通过。
14| /*
15| var _ = 1.23 << m // error: 浮点数不能被移位
16| */
```

上面提到的移位运算结果的最后一点类型推断规则有点反常。 这条规则的主要目的是为了防止一些移位运算在32位架构和64位架构的机器上的运算结果出现不一致但不一致却没有被及时发现的情况。 比如如果上面一段代码中第*10*行（或第*9*行）的`1`的类型被推断为它的默认类型`int`， 则在32位架构的机器上，`x`的取值在运行时刻将被截断为*0*，而在64位架构的机器上，`x`的取值在运行时刻将为2^{32}。 因为`m`是

一个变量，在32位架构的机器上，第*9*行和第*10*行并不会在编译时刻报错。 这将导致Go程序员在不经意间写出没有料到的和难以觉察的bug。 因此，第*9*行和第*10*行中的`1`的类型被推断为`int64`（最终的设想结果类型），而不是它们的默认类型`int`。

下面这段代码展示了对于左操作数为类型不确定值的移位运算，编译结果因右操作数是否为常量而带来的不同结果：

```
 1| const n = uint(2)
 2| var m = uint(2)
 3|
 4| // 这两行编译没问题。
 5| var _ float64 = 1 << n
 6| var _ = float64(1 << n)
 7|
 8| // 这两行编译失败。
 9| var _ float64 = 1 << m  // error
10| var _ = float64(1 << m) // error
```

上面这段代码最后两行编译失败是因为它们都等价于下面这两行：

```
 1| var _ = float64(1) << m
 2| var _ = 1.0 << m // error: shift of type float64
```

另一个例子：

```
 1| package main
 2|
 3| const n = uint(8)
 4| var m = uint(8)
 5|
 6| func main() {
 7|    println(a, b) // 2 0
 8| }
 9|
10| var a byte = 1 << n / 128
11| var b byte = 1 << m / 128
```

上面这个程序打印出`2 0`，因为最后两行等价于：

```
 1| var a = byte(int(1) << n / 128)
 2| var b = byte(1) << m / 128
```

关于除法和余数运算

假设两个操作数x和y的类型为同一个整数类型， 则它们通过除法和余数运算得到的商q（= x / y）和余数r（= x % y）满足x == q*y + r（|r| < |y|）。如果余数r不为零，则它的符号和被除数x相同。商q的结果为x / y向零靠拢截断。

如果除数y是一个常量，则它必须不为0，否则编译不通过。 如果它是一个整数型非常量，则在运行时刻将抛出一个恐慌（panic）。 恐慌类似与某些其它语言中的异常（exception）。 我们将在以后的文章中了解到Go中的恐慌和恐慌恢复机制。如果除数y非整数型的非常量，则运算结果为一个无穷大（Inf，当被除数不为0时）或者NaN（not a number，当被除数为0时）。

示例：

```
 1| println( 5/3,   5%3)  // 1 2
 2| println( 5/-3,  5%-3) // -1 2
 3| println(-5/3,  -5%3)  // -1 -2
 4| println(-5/-3, -5%-3) // 1 -2
 5|
 6| println(5.0 / 3.0)     // 1.666667
 7| println((1-1i)/(1+1i)) // -1i
 8|
 9| var a, b = 1.0, 0.0
10| println(a/b, b/b) // +Inf NaN
11|
12| _ = int(a)/int(b) // 编译没问题，但在运行时刻将造成恐慌。
13|
14| // 这两行编译不通过。
15| println(1.0/0.0) // error: 除数为0
16| println(0.0/0.0) // error: 除数为0
```

op=运算符

对于一个二元算数运算符op，语句x = x op y可以被简写为x op= y。 在这个简写的语句中，x只会被估值一次。

示例：

```
 1| var a, b int8 = 3, 5
 2| a += b
 3| println(a) // 8
 4| a *= a
 5| println(a) // 64
 6| a /= b
```

```
 7| println(a) // 12
 8| a %= b
 9| println(a) // 2
10| b <<= uint(a)
11| println(b) // 20
```

自增和自减操作符

和很多其它流行语言一样，Go也支持自增（++）和自减（--）操作符。 不过和其它语言不一样的是，自增（aNumber++）和自减（aNumber--）操作没有返回值，所以它们不能当做表达式（第11章）来使用。 另一个显著区别是，在Go中，自增（++）和自减（--）操作符只能后置，不能前置。

一个例子：

```
 1| package main
 2|
 3| func main() {
 4|     a, b, c := 12, 1.2, 1+2i
 5|     a++ // ok. <=> a += 1 <=> a = a + 1
 6|     b-- // ok. <=> b -= 1 <=> b = b - 1
 7|     c++ // ok.
 8|
 9|     // 下面这些行编译不通过。
10|     /*
11|     _ = a++
12|     _ = b--
13|     _ = c++
14|     ++a
15|     --b
16|     ++c
17|     */
18| }
```

字符串衔接运算符

上面已经提到了，加法运算符也可用做字符串衔接运算符。

字面形式	名称	对两个操作数的要求
+	字符串衔接	两个操作数必须为同一类型的字符串值。

+=运算符也适用于字符串衔接。

示例：

```
1| println("Go" + "lang") // Golang
2| var a = "Go"
3| a += "lang"
4| println(a) // Golang
```

如果一个字符串衔接运算中的一个操作值为类型确定的，则结果字符串是一个类型和此操作数类型相同的类型确定值。 否则，结果字符串是一个类型不确定值（肯定是一个常量）。

布尔（又称逻辑）运算符

Go支持两种布尔二元运算符和一种布尔一元运算符。

字面形式	名称	对操作值的要求
&&	布尔与（二元）	两个操作值的类型必须为同一布尔类型。
||	布尔或（二元）	
!	布尔否（一元）	唯一的一个操作值的类型必须为一个布尔类型。

我们可以用下一小节介绍的不等于操作符 != 来做为布尔异或操作符。

机理解释：

```
// x     y       x && y   x || y   !x      !y
true     true    true     true     false   false
true     false   false    true     false   true
false    true    false    true     true    false
false    false   false    false    true    true
```

如果一个布尔运算中的一个操作值为类型确定的，则结果为一个和此操作值类型相同的类型确定值。 否则，结果为一个类型不确定布尔值。

比较运算符

Go支持6种比较运算符：

字面形式	名称	对两个操作值的要求
==	等于	如果两个操作数都为类型确定的，则它们的类型必须一样，或者其中一个操作数可以隐式转换为另一个操作数的类型。 两者的类型必须都为可比较类型（将在以后的文章中介绍）。

字面形式	名称	对两个操作值的要求
!=	不等于	如果只有一个操作数是类型确定的，则另一个类型不确定操作数必须可以隐式转换到类型确定操作数的类型。 如果两个操作数都是类型不确定的，则它们必须同时为两个类型不确定布尔值、两个类型不确定字符串值或者两个类型不确定数字值。
<	小于	两个操作值的类型必须相同并且它们的类型必须为整数类型、浮点数类型或者字符串类型。
<=	小于或等于	
>	大于	
>=	大于或等于	

比较运算的结果总是一个类型不确定布尔值。 如果一个比较运算中的两个操作数都为常量，则结果布尔值也为一个常量。

以后，如果我们说两个值可以比较，我们的意思是说这两个值可以用==或者!=运算符来比较。 我们将在以后的文章中，我们将了解到某些类型的值是不能比较的。

注意，并非所有的实数在内存中都可以被精确地表示，所以比较两个浮点数或者复数的结果并不是很可靠。 在编程中，我们常常比较两个浮点数的差值是否小于一个阈值来检查两个浮点数是否相等。

操作符运算的优先级

Go中的操作符运算的优先级和其它流行语言有一些差别。 下面列出了本文介绍的操作符的优先级。 同一行中的操作符的优先级是一样的。优先级逐行递减。

```
1| *    /    %    <<   >>   &    &^
2| +    -    |    ^
3| ==   !=   <    <=   >    >=
4| &&
5| ||
```

一个和其它流行语言明显的差别是，移位运算<<和>>的优先级比加减法+和-的优先级要高。

一个表达式（做为一个子表达式）可以出现在另一个表达式中。 这个子表达式的估值结果将成为另一个表达式的一个操作数。 在这样的复杂表达式中，对于相同优先级的运算，它们将从左到右进行估值。 和很多其它语言一样，我们也可用一对小括号()来提升一个子运算的优先级。

更多关于常量表达式

常量子表达式的顺序有可能影响到最终的估值结果。

下面这个声明的变量将被初始化为`2.2`，而不是`2.7`。 优先级更高的子表达式`3/2`是一个常量表达式，所以它将在编译阶段被估值。 根据上面介绍的规则，在运算中，`3`和`2`都被视为`int`，所以`3/2`的估值结果为`1`。 在常量表达式`1.2 + 1`的运算中，两个操作数的类型被视为`float64`，所以最终的估值结果为`2.2`。

```
1| var x = 1.2 + 3/2
```

再比如下例，在一个常量声明中，`3/2`先被估值，其结果为`1`，所以最终的估值结果为`0.1`。 在第二个常量声明中，`0.1*3`先被估值，其结果为`0.3`，所以最终的估值结果为`0.15`。

```
1| package main
2|
3| const x = 3/2*0.1
4| const y = 0.1*3/2
5|
6| func main() {
7|    println(x) // +1.000000e-001
8|    println(y) // +1.500000e-001
9| }
```

更多其它操作符

Go中还有一些其它操作符。它们将在后续其它适当的文章中介绍。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

函数声明和调用

除了上一篇文章介绍的运算符操作，函数操作是另一种在编程中常用的操作。 函数操作常被称为函数调用。此篇文章将介绍如何在Go中声明和调用函数。

函数声明

让我们来看一个函数声明：

```
 1| func SquaresOfSumAndDiff(a int64, b int64) (s int64, d
int64) {
 2|    x, y := a + b, a - b
 3|    s = x * x
 4|    d = y * y
 5|    return // <=> return s, d
 6| }
```

从上面的例子中，我们可以发现一个函数声明从左到右由以下部分组成：

1. 第一部分是`func`关键字。
2. 第二部分是函数名称。函数名称必须是一个标识符。 这里的函数名称是`SquareOfSumAndDiff`。
3. 第三部分是输入参数列表。输入参数声明列表必须用一对小括号括起来。 输入参数声明有时也称为形参声明（对应后面将介绍的函数调用中的实参）。
4. 第四部分是输出结果声明列表。在Go中，一个函数可以有多个返回值。 比如上面这个例子就有两个返回值。 当一个函数的输出结果声明列表为空或者只包含一个匿名结果声明时，此列表可以不用一对小括号括起来（见下面的示例）；否则，小括号是必需的。
5. 最后一部分是函数体。函数体必须用一对大括号括起来。 一对大括号和它其间的代码形成了一个显式代码块。 在一个函数体内，`return`关键字可以用来结束此函数的正常向前执行流程并进入此函数的退出阶段（详见下下节中的解释）。

在上面的例子中，每个函数参数和结果声明都由一个名字和一个类型组成（变量名字在前，类型在后）。 我们可以把一个参数和结果声明看作是一个省略了`var`关键字的标准变量声明。 上面这个函数有两个输入参数（`a`和`b`）以及两个输出结果（`x`和`y`）。 它们的类型都是`int64`。

输出结果声明列表中的所有声明中的结果名称可以（而且必须）同时出现或者同时省略。 这两种方式在实践中都使用得很广泛。 如果一个返回结果声明中的结果名

称没有省略，则这个返回结果称为具名返回结果。否则称为匿名返回结果。

如果一个函数声明的所有返回结果均为匿名的，则在此函数体内的返回语句`return`关键字后必须跟随一系列返回值，这些返回值和此函数的各个返回结果声明一一对应。比如，下面这个函数声明和上例中的函数声明是等价的。

```
1| func SquaresOfSumAndDiff(a int64, b int64) (int64, int64) {
2|     return (a+b) * (a+b), (a-b) * (a-b)
3| }
```

事实上，如果一个函数声明中的所有输入参数在此函数体内都没有被使用过，则它们也可以都同时是匿名的。 不过这种情形在实际编程中很少见。

尽管一个函数声明中的输入参数和返回结果看上去是声明在这个函数体的外部，但是在此函数体内，这些输入参数和输出结果被当作局部变量来使用。 但输入参数和输出结果和普通局部变量还是有一点区别的：目前的主流Go编译器不允许一个名称不为_的普通局部变量被声明而不有效使用。

Go不支持输入参数默认值。每个返回结果的默认值是它的类型的零值。 比如，下面的函数在被调用时将打印出（和返回）`0 false`。

```
1| func f() (x int, y bool) {
2|     println(x, y) // 0 false
3|     return
4| }
```

和普通的变量声明一样，如果若干连续的输入参数或者返回结果的类型相同，则在它们的声明中可以共用一个类型。 比如，上面的两个`SquaresOfSumAndDiff`函数声明和下面这个是完全等价的。

```
1| func SquaresOfSumAndDiff(a, b int64) (s, d int64) {
2|     return (a+b) * (a+b), (a-b) * (a-b)
3|     // 上面这行等价于下面这行：
4|     // s = (a+b) * (a+b); d = (a-b) * (a-b); return
5| }
```

注意，尽管在上面这个函数声明的返回结果都是具名的，函数体内的`return`关键字后仍然可以跟返回值。

如果一个函数声明只包含一个返回结果，并且此返回结果是匿名的，则此函数声明中的返回结果部分不必用小括号括起来。 如果一个函数声明的返回结果列表为空，则此函数声明中的返回结果部分可以完全被省略掉。 一个函数声明的输入参数列表部分总不能省略掉，即使此函数声明的输入参数列表为空。

下面是更多函数声明的例子：

```
 1| func CompareLower4bits(m, n uint32) (r bool) {
 2|     // 下面这两行等价于：return m&0xFF > n&0xff
 3|     r = m&0xF > n&0xf
 4|     return
 5| }
 6|
 7| // 此函数没有输入参数。它的结果声明列表只包含一个
 8| // 匿名结果声明，因此它不必用()括起来。
 9| func VersionString() string {
10|     return "go1.0"
11| }
12|
13| // 此函数没有返回结果。它的所有输入参数都是匿名的。
14| // 它的结果声明列表为空，因此可以被省略掉。
15| func doNothing(string, int) {
16| }
```

在前面的《Go语言101》文章中，我们已经知道一个程序的`main`入口函数必须不带任何输入参数和返回结果。

注意，在Go中，所有函数都必须直接声明在包级代码块中。 或者说，任何一个函数都不能被声明在另一个函数体内。 虽然匿名函数（将在下面的某节中介绍）可以定义在函数体内，但匿名函数定义不属于函数声明。

函数调用

一个声明的函数可以通过它的名称和一个实参列表来调用之。 一个实参列表必须用小括号括起来。 实参列表中的每一个单值实参对应着（或称被传递给了）一个形参。

注意：函数传参也属于赋值操作。在传参中，各个实参被赋值给各个对应形参。

一个实参值的类型不必一定要和其对应的形参声明的类型一样。 但如果一个实参值的类型和其对应的形参声明的类型不一致，则此实参必须能够隐式转换到其对应的形参的类型。

如果一个函数带有返回值，则它的一个调用被视为一个表达式。如果此函数返回多个结果，则它的每个调用被视为一个多值表达式。 一个多值表达式可以被同时赋值给多个目标值（数量必须匹配，各个输出结果被赋值给相对应的目标值）。

下面这个例子完整地展示了如何调用几个已经声明了的函数。

```
 1| package main
 2|
```

```
 3| func SquaresOfSumAndDiff(a int64, b int64) (int64, int64) {
 4|     return (a+b) * (a+b), (a-b) * (a-b)
 5| }
 6|
 7| func CompareLower4bits(m, n uint32) (r bool) {
 8|     r = m&0xF > n&0xf
 9|     return
10| }
11|
12| // 使用一个函数调用的返回结果来初始化一个包级变量。
13| var v = VersionString()
14|
15| func main() {
16|     println(v) // v1.0
17|     x, y := SquaresOfSumAndDiff(3, 6)
18|     println(x, y) // 81 9
19|     b := CompareLower4bits(uint32(x), uint32(y))
20|     println(b) // false
21|     // "Go"的类型被推断为string；1的类型被推断为int32。
22|     doNothing("Go", 1)
23| }
24|
25| func VersionString() string {
26|     return "v1.0"
27| }
28|
29| func doNothing(string, int32) {
30| }
```

从上例可以看出，一个函数的声明可以出现在它的调用之前，也可以出现在它的调用之后。

一个函数调用可以被延迟执行或者在另一个协程（goroutine，或称绿色线程）中执行。 后面的一文（第13章）将对这两个特性进行详解。

函数调用的退出阶段

在Go中，当一个函数调用返回后（比如执行了一个`return`语句或者函数中的最后一条语句执行完毕）， 此调用可能并未立即退出。一个函数调用从返回开始到最终退出的阶段称为此函数调用的退出阶段（exiting phase）。 函数调用的退出阶段的意义将在讲解延迟函数（第13章）的时候体现出来。

函数调用的退出阶段将在后面的一篇文章（第31章）中详细解释。

匿名函数

Go支持匿名函数。定义一个匿名函数和声明一个函数类似，但是一个匿名函数的定义中不包含函数名称部分。 注意匿名函数定义不是一个函数声明。

一个匿名函数在定义后可以被立即调用，比如：

```
 1| package main
 2|
 3| func main() {
 4|    // 这个匿名函数没有输入参数，但有两个返回结果。
 5|    x, y := func() (int, int) {
 6|       println("This function has no parameters.")
 7|       return 3, 4
 8|    }() // 一对小括号表示立即调用此函数。不需传递实参。
 9|
10|    // 下面这些匿名函数没有返回结果。
11|
12|    func(a, b int) {
13|       println("a*a + b*b =", a*a + b*b) // a*a + b*b = 25
14|    }(x, y) // 立即调用并传递两个实参。
15|
16|    func(x int) {
17|       // 形参x遮挡了外层声明的变量x。
18|       println("x*x + y*y =", x*x + y*y) // x*x + y*y = 32
19|    }(y) // 将实参y传递给形参x。
20|
21|    func() {
22|       println("x*x + y*y =", x*x + y*y) // x*x + y*y = 25
23|    }() // 不需传递实参。
24| }
```

注意，上例中的最后一个匿名函数处于变量x和y的作用域内，所以在它的函数体内可以直接使用这两个变量。 这样的函数称为闭包（closure）。事实上，Go中的所有的自定义函数（包括声明的函数和匿名函数）都可以被视为闭包。 这就是为什么Go中的函数使用起来和动态语言中的函数一样灵活。

在后面的文章中，我们将了解到一个匿名函数可以被赋值给某个函数类型的值，从而我们不必在定义完此匿名函数后立即调用它，而是可以在以后合适的时候再调用它。

内置函数

Go支持一些内置函数，比如前面的例子中已经用到过多次的`println`和`print`函数。 我们可以不引入任何库包（见下一篇文章）而调用一个内置函数。

我们可以使用内置函数`real`和`imag`来得到一个复数的实部和虚部（均为浮点数类型）。 注意，如果这两个函数的任何一个调用的实参是一个常量，则此调用将在编译时刻被估值，其返回结果也是一个常量。 此调用将被视为一个常量表达式。特别地，如果此实参是一个类型不确定值，则返回结果也是一个类型不确定值。

一个例子：

```
 1| // c是一个类型不确定复数常量。
 2| const c = complex(1.6, 3.3)
 3|
 4| // 函数调用real(c)和imag(c)的结果都是类型
 5| // 不确定浮点数值。在下面这句赋值中，它们都
 6| // 被推断为float32类型的值。
 7| var a, b float32 = real(c), imag(c)
 8|
 9| // 变量d的类型被推断为内置类型complex64。
10| // 函数调用real(d)和imag(d)的结果都是
11| // 类型为float32的类型确定值。
12| var d = complex(a, b)
13|
14| // 变量e的类型被推断为内置类型complex128。
15| // 函数调用real(e)和imag(e)的结果都是
16| // 类型为float64的类型确定值。
17| var e = c
```

更多内置类型将在很多后面其它文章中介绍。

更多函数相关的概念

本文是一篇Go函数入门的文章，很多其它函数相关的概念并未在此文中解释。 今后，我们可以从函数类型和函数值（第20章）一文中了解到和函数相关的其它概念。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

代码包和包引入

和很多现代编程语言一样，Go代码包（package）来组织管理代码。 我们必须先引入一个代码包（除了builtin标准库包）才能使用其中导出的代码要素（比如函数、类型、变量和具名常量等）。 此篇文章将讲解Go代码包和代码包引入（import）。

包引入

下面这个简短的程序（假设它存在一个名为simple-import-demo.go的源文件中）引入了一个标准库包。

```
1| package main
2|
3| import "fmt"
4|
5| func main() {
6|     fmt.Println("Go has", 25, "keywords.")
7| }
```

对此程序的一些解释：

- 第一行指定了源文件simple-import-demo.go所处的包名为main。 程序入口main函数必须处于一个名为main的代码包中。
- 第三行通过使用import关键字引入了fmt标准库包。 在此源文件中，fmt标准库包将用fmt标识符来表示。 标识符fmt称为fmt标准库包的引入名称。（后续某节将详述代码包的引入名称）。
- fmt标准库包中声明了很多终端打印函数供其它代码包使用。 Println函数是其中之一。 它可以将不定数量参数的字符串表示形式输出到标准输出中。第六行调用了此Println函数。 注意在此调用中，函数名之前需要带上前缀fmt.，其中fmt是Println函数所处的代码包的引入名称。aImportName.AnExportedIdentifier这种形式称为一个限定标识符（qualified identifier（https://golang.google.cn/ref/spec#Qualified_identifiers））。
- fmt.Println函数调用接受任意数量的实参并且对实参的类型没有任何限制。 所以此程序中的此函数调用的三个实参的类型将被推断为它们各自的默认类型：string、int和string。
- 对于一个fmt.Println函数调用，任何两个相邻的实参的输出之间将被插入一个空格字符，并且在最后将输出一个空行字符。

下面是上面这个程序的运行结果：

```
$ go run simple-import-demo.go
Go has 25 keywords.
```

当一个代码包被引入一个Go源文件时，只有此代码包中的导出（第5章）代码要素（名称为大写字母的变量、常量、函数、定义类型和类型别名等）可以在此源文件被使用。 比如上例中的`Println`函数即为一个导出代码要素，所以它可以在上面的程序源文件中使用。

前面几篇文章中使用的内置函数`print`和`println`提供了和`fmt`标准库包中的对应函数相似的功能。 内置函数可以不用引入任何代码包而直接使用。

注意：`print`和`println`这两个内置函数不推荐使用在生产环境，因为它们不保证一定会出现在以后的Go版本中。

我们可以访问Go官网（https://golang.org/pkg/）（墙内版（https://golang.google.cn/pkg/））来查看各个标准库包的文档， 我们也可以开启一个本地文档服务器（第3章）来查看这些文档。

一个包引入也可称为一个包声明。一个包声明只在当前包含此声明的源文件内可见。

另外一个例子：

```
1| package main
2|
3| import "fmt"
4| import "math/rand"
5|
6| func main() {
7|     fmt.Printf("下一个伪随机数是%v。\n", rand.Uint32())
8| }
```

这个例子多引入了一个`math/rand`标准库包。 此包是`math`标准库包中的一个子包。 此包提供了一些函数来产生伪随机数序列。

一些解释：

- 在此例中，`math/rand`标准库包的引入名是`rand`。 `rand.Uint32()`函数调用将返回一个`uint32`类型的随机数。
- `Printf`函数是`fmt`标准库包中提供的另外一个常用终端打印函数。 一个`Printf`函数调用必须带有至少一个实参，并且第一个实参的类型必须为`string`。 此第一个实参指定了此调用的打印格式。此格式中的`%v`在打印结果将被对应的后续实参的字符串表示形式所取代。 比如上列中的`%v`在打印结

果中将被`rand.Uint32()`函数调用所返回的随机数所取代。 打印格式中的`\n`表示一个换行符，这在基本类型和它们的字面量表示（第6章）一文中已经解释过。

上面这个程序的输出如下：

```
下一个伪随机数是2596996162。
```

注意：在Go 1.20之前，如果我们希望上面的程序每次运行的时候输出一个不同的随机数，我们需要在程序启动的时候调用`rand.Seed`函数来设置一个不同的随机数种子。

多个包引入语句可以用一对小括号来合并成一个包引入语句。比如下面这例。

```
 1| package main
 2|
 3| // 一条包引入语句引入了三个代码包。
 4| import (
 5|     "fmt"
 6|     "math/rand"
 7|     "time"
 8| )
 9|
10| func main() {
11|     // 设置随机数种子（仅在Go 1.20之前需要）。
12|     rand.Seed(time.Now().UnixNano())
13|     fmt.Printf("下一个伪随机数是%v。\n", rand.Uint32())
14| }
```

一些解释：

- 此例多引入了一个`time`标准库包。 此包提供了很多和时间相关的函数和类型。 其中`time.Time`和`time.Duration`是两个最常用的类型。
- 函数调用`time.Now()`将返回一个表示当前时间的类型为`time.Time`的值。
- `UnixNano`是类型`time.Time`的一个方法。 我们可以把方法看作是特殊的函数。方法将在Go中的方法（第22章）一文中详述。 方法调用`aTime.UnixNano()`将返回从UTC时间的1970年一月一日到`aTime`所表示的时间之间的纳秒数。 返回结果的类型为`int64`，这也是`rand.Seed`函数的参数类型（注意：`rand.Seed`函数从Go 1.20开始被声明为废弃了）。 在上例中，此方法调用的结果用来设置随机数种子。

更多关于`fmt.Printf`函数调用的输出格式

从上面的例子中，我们已经了解到`fmt.Printf`函数调用的第一个实参中的`%v`在输出中将替换为后续的实参的字符串表示形式。 实际上，这种百分号开头的占位字符组合还有很多。下面是一些常用的占位字符组合：

- `%v`：将被替换为对应实参字符串表示形式。
- `%T`：将替换为对应实参的类型的字符串表示形式。
- `%x`：将替换为对应实参的十六进制表示。实参的类型可以为字符串、整数、整数数组（array）或者整数切片（slice）等。 （数组和切片将在以后的文章中讲解。）
- `%s`：将被替换为对应实参的字符串表示形式。实参的类型必须为字符串或者字节切片（byte slice）类型。
- `%%`：将被替换为一个百分号。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     a, b := 123, "Go"
 7|     fmt.Printf("a == %v == 0x%x, b == %s\n", a, a, b)
 8|     fmt.Printf("type of a: %T, type of b: %T\n", a, b)
 9|     fmt.Printf("1%% 50%% 99%%\n")
10| }
```

输出：

```
a == 123 == 0x7b, b == Go
type of a: int, type of b: string
1% 50% 99%
```

请阅读[fmt标准库包的文档](https://golang.google.cn/pkg/fmt/)（https://golang.google.cn/pkg/fmt/）以了解更多的占位字符组合。 我们也可以运行`go doc fmt`命令来在终端中查看`fmt`标准库包的文档。 运行`go doc fmt.Printf`命令可以查看`fmt.Printf`函数的文档。

代码包目录、代码包引入路径和代码包依赖关系

一个代码包可以由若干Go源文件组成。一个代码包的源文件须都处于同一个目录下。 一个目录（不包含子目录）下的所有源文件必须都处于同一个代码包中，亦即这些源文件开头的`package pkgname`语句必须一致。 所以，一个代码包对应着一个目录（不包含子目录），反之亦然。 对应着一个代码包的目录称为此代码包的目录。 一个代码包目录下的每个子目录对应的都是另外一个独立的代码包。

对于Go官方工具链来说，一个引入路径中包含有internal目录名的代码包被视为一个特殊的代码包。 它只能被此internal目录的直接父目录（和此父目录的子目录）中的代码包所引入。 比如，代码包.../a/b/c/internal/d/e/f和.../a/b/c/internal只能被引入路径含有.../a/b/c前缀的代码包引入。

当一个代码包中的某个文件引入了另外一个代码包，则我们说前者代码包依赖于后者代码包。

Go不支持循环引用（依赖）。 如果一个代码包a依赖于代码包b，同时代码包b依赖于代码包c，则代码包c中的源文件不能引入代码包a和代码包b，代码包b中的源文件也不能引入代码包a。

当然，一个代码包中的源文件不能也没必要引入此代码包本身。

今后，我们称一个程序中含有main入口函数的名称为main的代码包为程序代码包（或者命令代码包），称其它代码包为库代码包。 程序代码包不能被其它代码包引入。一个程序只能有一个程序代码包。

代码包目录的名称并不要求一定要和其对应的代码包的名称相同。 但是，库代码包目录的名称最好设为和其对应的代码包的名称相同。 因为一个代码包的引入路径中包含的是此包的目录名，但是此包的默认引入名为此包的名称。 如果两者不一致，会使人感到困惑。

另一方面，最好给每个程序代码包目录指定一个有意义的名字，而不是它的包名main。

init函数

在一个代码包中，甚至一个源文件中，可以声明若干名为init的函数。 这些init函数必须不带任何输入参数和返回结果。

注意：我们不能声明名为init的包级变量、常量或者类型。

在程序运行时刻，在进入main入口函数之前，每个init函数在此包加载的时候将被（串行）执行并且只执行一遍。

下面这个简单的程序中有两个init函数：

```
1| package main
2|
3| import "fmt"
4|
5| func init() {
```

```
 6|    fmt.Println("hi,", bob)
 7| }
 8|
 9| func main() {
10|    fmt.Println("bye")
11| }
12|
13| func init() {
14|    fmt.Println("hello,", smith)
15| }
16|
17| func titledName(who string) string {
18|    return "Mr. " + who
19| }
20|
21| var bob, smith = titledName("Bob"), titledName("Smith")
```

此程序的运行结果：

```
hi, Mr. Bob
hello, Mr. Smith
bye
```

程序代码要素初始化顺序

一个程序中所涉及到的所有的在运行时刻要用到的代码包的加载是串行执行的。 在一个程序启动时，每个包中总是在它所有依赖的包都加载完成之后才开始加载。 程序代码包总是最后一个被加载的代码包。每个被用到的包会被而且仅会被加载一次。

在加载一个代码包的过程中，所有的声明在此包中的`init`函数将被串行调用并且仅调用执行一次。 一个代码包中声明的`init`函数的调用肯定晚于此代码包所依赖的代码包中声明的`init`函数。 所有的`init`函数都将在调用`main`入口函数之前被调用执行。

在同一个源文件中声明的`init`函数将按从上到下的顺序被调用执行。 对于声明在同一个包中的两个不同源文件中的两个`init`函数，Go语言白皮书推荐（但不强求）按照它们所处于的源文件的名称的词典序列（对英文来说，即字母顺序）来调用。所以最好不要让声明在同一个包中的两个不同源文件中的两个`init`函数存在依赖关系。

在加载一个代码包的时候，此代码包中声明的所有包级变量都将在此包中的任何一个`init`函数执行之前初始化完毕。

在同一个包内，包级变量将尽量按照它们在代码中的出现顺序被初始化，但是一个包级变量的初始化肯定晚于它所依赖的其它包级变量。 比如，在下面的代码片段中，四个包级变量的初始化顺序依次为`y`、`z`、`x`、`w`。

```
 1| func f() int {
 2|     return z + y
 3| }
 4|
 5| func g() int {
 6|     return y/2
 7| }
 8|
 9| var (
10|     w       = x
11|     x, y, z = f(), 123, g()
12| )
```

关于更具体的包级变量的初始化顺序，请阅读表达式估值顺序规则（第33章）一文。

完整的引入声明语句形式

事实上，一个引入声明语句的完整形式为：

```
import importname "path/to/package"
```

其中引入名`importname`是可选的，它的默认值为被引入的包的包名（不是目录名）。

事实上，在本文上面的例子中的包引入声明中，`importname`部分都被省略掉了，因为它们都分别和引入的代码包的包名相同。 这些引入声明等价于下面这些：

```
import fmt "fmt"         // <=> import "fmt"
import rand "math/rand" // <=> import "math/rand"
import time "time"       // <=> import "time"
```

如果一个包引入声明中的`importname`没有省略，则限定标识符使用的前缀必须为`importname`，而不是被引入的包的名称。

引入声明语句的完整形式在日常编程中使用的频率不是很高。 但是在某些情况下，完整形式必须被使用。 比如，如果一个源文件引入的两个代码包的包名一样，为了防止使编译器产生困惑，我们至少需要用完整形式为其中一个包指定一个不同的引入名以区分这两个包。

下面是一个使用了完整引入声明语句形式的例子。

```
 1| package main
 2|
 3| import (
 4|     format "fmt"
 5|     random "math/rand"
 6|     "time"
 7| )
 8|
 9| func main() {
10|     random.Seed(time.Now().UnixNano())
11|     format.Print("一个随机数:", random.Uint32(), "\n")
12|
13|     // 下面这行编译不通过，因为rand不可识别。
14|     /*
15|     fmt.Print("一个随机数:", rand.Uint32(), "\n")
16|     */
17| }
```

一些解释：

- 我们必须使用`format`和`random`，而不是`fmt`和`rand`，来做为限定标识符的前缀。
- `Print`是`fmt`标准库包中的另外一个函数。 和`Println`函数调用一样，一个`Print`函数调用也接受任意数量实参。 它将逐个打印出每个实参的字符串表示形式。如果相邻的两个实参都不是字符串类型，则在它们中间会打印一个空格字符。

一个完整引入声明语句形式的引入名`importname`可以是一个句点(`.`)。 这样的引入称为句点引入。使用被句点引入的包中的导出代码要素时，限定标识符的前缀必须省略。

例子：

```
 1| package main
 2|
 3| import (
 4|     . "fmt"
 5|     . "time"
 6| )
 7|
 8| func main() {
 9|     Println("Current time:", Now())
10| }
```

在上面这个例子中，Println和Now函数调用不需要带任何前缀。

一般来说，句点引入不推荐使用，因为它们会导致较低的代码可读性。

一个完整引入声明语句形式的引入名importname可以是一个空标识符(_)。 这样的引入称为匿名引入。一个包被匿名引入的目的主要是为了加载这个包，从而使得这个包中的代码要素得以初始化。 被匿名引入的包中的init函数将被执行并且仅执行一遍。

在下面这个例子中，net/http/pprof标准库包（https://golang.google.cn/pkg/net/http/pprof/）中的所有init函数将在main入口函数开始执行之前全部执行一遍。

```
 1| package main
 2|
 3| import _ "net/http/pprof"
 4|
 5| func main() {
 6|     ... // 做一些事情
 7| }
```

每个非匿名引入必须至少被使用一次

除了匿名引入，其它引入必须在代码中被使用一次。 比如，下面的程序编译不通过。

```
 1| package main
 2|
 3| import (
 4|     "net/http" // error: 引入未被使用
 5|     . "time"   // error: 引入未被使用
 6| )
 7|
 8| import (
 9|     format "fmt"  // okay: 下面被使用了一次
10|     _ "math/rand" // okay: 匿名引入
11| )
12|
13| func main() {
14|     format.Println() // 使用"fmt"包
15| }
```

模块

一个模块（module）为的若干代码包的集合。当被下载至本地后，这些代码包处于同一个目录（此模块的根目录）下。 一个模块可以有很多版本（版本号遵从Semantic Versioning（https://semver.org/）规范）。 更多关于模块的概念和使用，请阅读官方文档（https://golang.google.cn/ref/mod）。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

表达式、语句和简单语句

此篇文章将简单介绍一下Go语言中的表达式和语句，为后面的（特别是下一篇）文章做一个铺垫。

简单说来，一个表达式表示一个值，而一条语句表示一个操作。 但是在实际中，有些个表达式可能同时表示多个值，有些语句可能是由很多更基本的语句组成的。 另外，根据场合不同，某些语句也可以被视为表达式。

Go中，某些语句被称为简单语句。Go中各种流程控制语句的某些部分可能会被要求必须为简单语句或者表达式。 详见下一篇文章对Go中基本流程控制语句的介绍和解释。

本篇文章将不对表达式和语句作出详尽的解释。 详尽的解释需要大量的篇幅。 本文只是列出一些表达式和语句的例子，并非包括所有的表达式和语句的种类，但是简单语句的所有种类都会被列出来。

一些表达式的例子

Go中大多数的表达式都是单值表达式。一个单值表达式只表示一个值。 某些表达式可以表示多个值，它们被称为多值表达式。

以后（不包括本文），如果没有特殊说明，当表达式这个词被提及的时候，它表示一个单值表达式。

前面的几篇文章介绍的字面量、变量和具名常量等均属于单值表达式。它们可称为基本表达式。

前面的运算操作符（第8章）一文中介绍的运算符操作（不包括赋值部分）也都属于单值表达式。

如果一个函数至少返回一个值，则它的调用属于表达式。 特别的，如果此函数返回两个或两个以上的值，则对它的调用称为多值表达式。 不返回任何结果的函数的调用不属于表达式。

以后的某篇文章中介绍的方法（第22章）可以看作是特殊的函数。 所以上述对函数的解释同样适用于方法。

事实上，以后我们将会了解到自定义函数（包括方法）本身都属于函数类型的值，所以它们都是单值表达式。

通道的接收数据操作（不包括赋值部分）也属于表达式。通道（第21章）将在以后详解。

Go中的一些表达式，包括刚提及的通道的接收数据操作，可能会表示可变数量的值。 根据不同的场景，这样的表达式可能呈现为单值表达式，也可能呈现为多值表达式。 我们将在以后的文章中了解到这样的表达式。

简单语句类型列表

Go中有六种简单语句类型：

1. 变量短声明语句。
2. 纯赋值语句，包括`x op= y`这种运算形式。
3. 有返回结果的函数或方法调用，以及通道的接收数据操作。 上一节已经提到了，这些语句也可以用做表达式。
4. 通道的发送数据操作。上面已经提到过一次，通道以后将在此文中（第21章）详解。
5. 空语句。在下一篇文章我们将看到一些空语句的应用。
6. 自增（`x++`）和自减（`x--`）语句。

注意：和C/C++不一样，在Go中，自增和自减语句不能被当作表达式使用。

简单语句这个概念在Go中比较重要，所以请牢记这六种简单语句类型。

一些非简单语句

下面是一个非简单语句的不完整列表：

- 标准变量声明语句。是的，短声明语句属于简单语句，但是标准变量声明语句不属于。
- （具名）常量声明语句。
- 类型声明语句。
- （代码）包引入语句。
- 显式代码块。一个显式代码块起始于一个左大括号`{`，终止于一个右大括号`}`。 一个显式代码块中可以包含若干子语句。
- 函数声明。 一个函数声明中可以包含若干子语句。
- 流程控制跳转语句。详见下一章。
- 函数返回（`return`）语句。
- 延迟函数调用和协程创建语句。下下篇文章将会介绍。

一些表达式和语句的例子

```
1| // 一些非简单语句：
2| import "time"
```

```
 3| var a = 123
 4| const B = "Go"
 5| type Choice bool
 6| func f() int {
 7|    for a < 10 {
 8|       break
 9|    }
10|
11|    // 这是一个显式代码块。
12|    {
13|       // ...
14|    }
15|    return 567
16| }
17|
18| // 一些简单语句的例子：
19| c := make(chan bool) // 通道将在以后讲解
20| a = 789
21| a += 5
22| a = f() // 这是一个纯赋值语句
23| a++
24| a--
25| c <- true // 一个通道发送操作
26| z := <-c  // 一个使用通道接收操作
27|           // 做为源值的变量短声明语句
28|
29| // 一些表达式的例子：
30| 123
31| true
32| B
33| B + " language"
34| a - 789
35| a > 0 // 一个类型不确定布尔值
36| f     // 一个类型为“func ()”的表达式
37|
38| // 下面这些即可以被视为简单语句，也可以被视为表达式。
39| f() // 函数调用
40| <-c // 通道接收操作
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

基本流程控制语法

Go中的流程控制语句和其它很多流行语言很类似，但是也有不少区别。 本篇文章将列出所有这些相似点和不同点。

Go中的流程控制语句简单介绍

Go语言中有三种基本的流程控制代码块：

- `if-else`条件分支代码块；
- `for`循环代码块；
- `switch-case`多条件分支代码块。

Go中另外还有几种和特定种类的类型相关的流程控制代码块：

- 用来遍历整数、各种容器（第18章）、通道（第21章）和某些函数（第20章）的`for-range`循环代码块。
- 接口（第23章）相关的`type-switch`多条件分支代码块。
- 通道（第21章）相关的`select-case`多分支代码块。

和很多其它流行语言一样，Go也支持`break`、`continue`和`goto`等跳转语句。 另外，Go还支持一个特有的`fallthrough`跳转语句。

Go所支持的六种流程控制代码块中，除了`if-else`条件分支代码块，其它五种称为可跳出代码块。 我们可以在一个可跳出代码块中使用`break`语句以跳出此代码块。

我们可以在`for`和`for-range`两种循环代码块中使用`continue`语句提前结束一个循环步。 除了这两种循环代码块，其它四种代码块称为分支代码块。

请注意，上面所提及的每种流程控制块的一个分支都属于一条语句。这样的语句常常会包含很多子语句。

上面所提及的流程控制语句都属于狭义上的流程控制语句。 下一篇文章中将要介绍的协程、延迟函数调用、以及恐慌和恢复（第13章），以及今后要介绍的并发同步技术（第36章）属于广义上的流程控制语句。

本文余下的部分主要解释三种基本的流程控制语句和各种代码跳转语句。 Go 1.22引入的`for range anInteger {...}`循环也将被介绍。 其它上面提及的流程控制块将在后面其它文章中逐渐介绍。

if-else条件分支控制代码块

一个if-else条件分支控制代码块的完整形式如下：

```
1| if InitSimpleStatement; Condition {
2|     // do something
3| } else {
4|     // do something
5| }
```

if和else是两个关键字。 和很多其它编程语言一样， else分支是可选的。

在一个if-else条件分支控制代码块中，

- InitSimpleStatement部分是可选的，如果它没被省略掉，则它必须为一条简单语句（第11章）。 如果它被省略掉，它可以被视为一条空语句（简单语句的一种）。 在实际编程中， InitSimpleStatement常常为一条变量短声明语句。
- Condition必须为一个结果为布尔值的表达式（第11章）（它被称为条件表达式）。 Condition部分可以用一对小括号括起来，但大多数情况下不需要。

注意，我们不能用一对小括号将InitSimpleStatement和Condition两部分括在一起。

在执行一个if-else条件分支控制代码块中，如果InitSimpleStatement这条语句没有被省略，则此条语句将被率先执行。 如果InitSimpleStatement被省略掉，则其后跟随的分号;也可一块儿被省略。

每个if-else流程控制包含一个隐式代码块，一个if分支显式代码块和一个可选的else分支代码块。 这两个分支代码块内嵌在这个隐式代码块中。 在程序运行中，如果Condition条件表达式的估值结果为true，则if分支式代码块将被执行；否则， else分支代码块将被执行。

一个例子：

```
1| package main
2|
3| import (
4|     "fmt"
5|     "math/rand"
6|     "time"
7| )
```

```
 8|
 9| func main() {
10|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
11|
12|    if n := rand.Int(); n%2 == 0 {
13|       fmt.Println(n, "是一个偶数。")
14|    } else {
15|       fmt.Println(n, "是一个奇数。")
16|    }
17|
18|    n := rand.Int() % 2 // 此n不是上面声明的n
19|    if n % 2 == 0 {
20|       fmt.Println("一个偶数。")
21|    }
22|
23|    if ; n % 2 != 0 {
24|       fmt.Println("一个奇数。")
25|    }
26| }
```

如果`InitSimpleStatement`语句是一个变量短声明语句，则在此语句中声明的变量被声明在外层的隐式代码块中。

可选的`else`分支代码块一般情况下必须为显式的，但是如果此分支为另外一个`if-else`块，则此分支代码块可以是隐式的。

另一个例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6| )
 7|
 8| func main() {
 9|    if h := time.Now().Hour(); h < 12 {
10|       fmt.Println("现在为上午。")
11|    } else if h > 19 {
12|       fmt.Println("现在为晚上。")
13|    } else {
14|       fmt.Println("现在为下午。")
15|       // 左h是一个新声明的变量，右h已经在上面声明了。
16|       h := h
17|       // 刚声明的h遮掩了上面声明的h。
```

```
18|         _ = h
19|     }
20|
21|     // 上面声明的两个h在此处都不可见。
22| }
```

for循环代码块

for循环代码块的完整形式如下：

```
1| for InitSimpleStatement; Condition; PostSimpleStatement {
2|     // do something
3| }
```

其中for是一个关键字。

在一个for循环代码块中，

- InitSimpleStatement（初始化语句）和PostSimpleStatement（步尾语句）两个部分必须均为简单语句，并且PostSimpleStatement不能为一个变量短声明语句。
- Condition必须为一个结果为布尔值的表达式（它被称为条件表达式）。

所有这三个刚提到的部分都是可选的。和很多其它流行语言不同，在Go中上述三部分不能用小括号括在一起。

每个for流程控制包括至少两个子代码块。 其中一个是隐式的，另一个是显式的（花括号起始和终止的部分，又称循环体）。 此显式代码块内嵌在隐式代码块之中。

在一个for循环流程控制中，初始化语句（InitSimpleStatement）将被率先执行，并且只会被执行一次。

在每个循环步的开始，Condition条件表达式将被估值。如果估值结果为false，则循环立即结束；否则循环体（即显式代码块）将被执行。

在每个循环步的结尾，步尾语句（PostSimpleStatement）将被执行。

下面是一个使用for循环流程控制的例子。此程序将逐行打印出0到9十个数字。

```
1| for i := 0; i < 10; i++ {
2|     fmt.Println(i)
3| }
```

在一个for循环流程控制中，如果InitSimpleStatement和PostSimpleStatement两部分同时被省略（可将它们视为空语句），则和它们相邻的两个分号也可被省略。 这样的形式被称为只有条件表达式的for循环。只有条件表达式的for循环和很多其它语言中的while循环类似。

```
 1| var i = 0
 2| for ; i < 10; {
 3|    fmt.Println(i)
 4|    i++
 5| }
 6| for i < 20 {
 7|    fmt.Println(i)
 8|    i++
 9| }
```

在一个for循环流程控制中，如果条件表达式部分被省略，则编译器视其为true。

```
 1| for i := 0; ; i++ { // 等价于：for i := 0; true; i++ {
 2|    if i >= 10 {
 3|       break
 4|    }
 5|    fmt.Println(i)
 6| }
 7|
 8| // 下面这几个循环（大多时候）是等价的。
 9| for ; true; {
10| }
11| for true {
12| }
13| for ; ; {
14| }
15| for {
16| }
```

在一个for循环流程控制中，如果初始化语句InitSimpleStatement是一个变量短声明语句，则在此语句中声明的循环变量被声明在外层的隐式代码块中。 我们可以在内嵌的循环体（显式代码块）中声明同名变量来遮挡在InitSimpleStatement中声明的变量。 比如下面的代码打印出012，而不是0。

```
 1| for i := 0; i < 3; i++ {
 2|    fmt.Print(i)
 3|    i := i // 这里声明的变量i遮挡了上面声明的i。
 4|           // 右边的i为上面声明的循环变量i。
 5|    i = 10 // 新声明的i被更改了。
```

```
6|     _ = i
7| }
```

注意：Go 1.22修改了for循环流程控制的语义：

- 在Go 1.22之前，每一个声明的循环变量在整个循环的执行过程中只会被实例化一次。此唯一的实例将被所有循环步共享。
- 从Go 1.22开始，每一个声明的循环变量将会在每个循环步被实例化一次。每个实例只作用于当前循环步。

对于大多数情形，此语义改变不会造成代码的行为改变。 但是，有时候，它会（第13章）。 所以，此语义改变破坏了向后兼容性。 为了将此语义改变造成的破坏减至做小，从Go 1.22开始，每个Go源文件都应该被指定一个Go版本号（https://go101.org/blog/2024-03-01-for-loop-semantic-changes-in-go-1.22.html）。

一条break语句可以用来提前跳出包含此break语句的最内层for循环。 下面这段代码同样逐行打印出0到9十个数字。

```
1| i := 0
2| for {
3|     if i >= 10 {
4|         break
5|     }
6|     fmt.Println(i)
7|     i++
8| }
```

一条continue语句可以被用来提前结束包含此continue语句的最内层for循环的当前循环步（步尾语句仍将得到执行）。 比如下面这段代码将打印出13579。

```
1| for i := 0; i < 10; i++ {
2|     if i % 2 == 0 {
3|         continue
4|     }
5|     fmt.Print(i)
6| }
```

for-range流程控制代码块用来遍历整数

for-range流程控制代码块可以用来遍历整数、各种容器（第18章）和通道（第21章）等。 本文只介绍如何使用for-range流程控制代码块来遍历整数。

注意：使用for-range流程控制代码块来遍历整数是从Go 1.22才开始支持的。

下面的代码

```
1| // 变量 i 已在前面声明过了
2| for i = range anInteger {
3|     ...
4| }
```

等价于

```
1| for i = 0; i < anInteger; i++ {
2|     ...
3| }
```

同样，

```
1| for i := range anInteger {
2|     ...
3| }
```

等价于（Go 1.22+语义下的）

```
1| for i := 0; i < anInteger; i++ {
2|     ...
3| }
```

比如，上一节的最后一个例子等价于：

```
1| for i := range 10 {
2|     if i % 2 == 0 {
3|         continue
4|     }
5|     fmt.Print(i)
6| }
```

switch-case流程控制代码块

switch-case流程控制代码块是另外一种多分支代码块。

一个switch-case流程控制代码块的完整形式为：

```
1| switch InitSimpleStatement; CompareOperand0 {
2| case CompareOperandList1:
3|     // do something
```

```
 4| case CompareOperandList2:
 5|    // do something
 6| ...
 7| case CompareOperandListN:
 8|    // do something
 9| default:
10|    // do something
11| }
```

其中switch、case和default是三个关键字。

在一个switch-case流程控制代码块中，

- InitSimpleStatement部分必须为一条简单语句，它是可选的。
- CompareOperand0部分必须为一个表达式（如果它没被省略的话，见下）。此表达式的估值结果总是被视为一个类型确定值。如果它是一个类型不确定值，则它被视为类型为它的默认类型的类型确定值。 因为这个原因，此表达式不能为类型不确定的nil值。 CompareOperand0常被称为switch表达式。
- 每个CompareOperandListX部分（X表示1到N）必须为一个用（英文）逗号分隔开来的表达式列表。 其中每个表达式都必须能和CompareOperand0表达式进行比较。 每个这样的表达式常被称为case表达式。 如果其中case表达式是一个类型不确定值，则它必须能够自动隐式转化为对应的switch表达式的类型，否则编译将失败。

每个case CompareOperandListX:部分和default:之后形成了一个隐式代码块。 每个这样的隐式代码块和它对应的case CompareOperandListX:或者default:形成了一个分支。 每个分支都是可选的。

每个switch-case流程控制代码块中最多只能有一个default分支（默认分支）。

除了刚提到的分支代码块，每个switch-case流程控制至少包括其它两个代码块。其中一个是隐式的，另一个是显式的。此显式的代码块内嵌在隐式的代码块之中。所有的分支代码块都内嵌在此显式代码块之中（因此也间接内嵌在刚提及的隐式代码块中）。

switch-case代码块属于可跳出流程控制。 break可以使用在一个switch-case流程控制的任何分支代码块之中以提前跳出此switch-case流程控制。

当一个switch-case流程控制被执行到的时候，其中的简单语句InitSimpleStatement将率先被执行（只执行一次）。 随后switch表达式CompareOperand0将被估值（仅一次）。上面已经提到，此估值结果一定为一个类型确定值。 然后此结果值将从上到下从左到右和各个CompareOperandListX表达式列表中的各个case表达式逐个依次比较（使用==运算符）。 一旦发现某个表达

式和CompareOperand0相等，比较过程停止并且此表达式对应的分支代码块将得到执行。 如果没有任何一个表达式和CompareOperand0相等，则default默认分支将得到执行（如果此分支存在的话）。

一个switch-case流程控制的例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "math/rand"
 6|     "time"
 7| )
 8|
 9| func main() {
10|     rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
11|     switch n := rand.Intn(100); n%9 {
12|     case 0:
13|         fmt.Println(n, "is a multiple of 9.")
14|
15|         // 和很多其它语言不一样，程序不会自动自动从
16|         // 当前分支代码块跳到下一个分支代码块去执行。
17|         // 所以，这里不需要一个break语句。
18|     case 1, 2, 3:
19|         fmt.Println(n, "mod 9 is 1, 2 or 3.")
20|         break // 这里的break语句可有可无的，效果
21|               // 是一样的。执行不会跳到下一个分支。
22|     case 4, 5, 6:
23|         fmt.Println(n, "mod 9 is 4, 5 or 6.")
24|     // case 6, 7, 8:
25|         // 上一行可能编译不过，因为6和上一个case中的
26|         // 6重复了。是否能编译通过取决于具体编译器实现。
27|     default:
28|         fmt.Println(n, "mod 9 is 7 or 8.")
29|     }
30| }
```

在上例中，rand.Intn函数将返回一个从0到所传实参之间类型为int的随机数。

注意，编译器可能会不允许一个switch-case流程控制中有任何两个case表达式可以在编译时刻确定相等。 比如，当前的官方标准编译器（1.23版本）认为上例中的case 6, 7, 8一行是不合法的（如果此行未被注释掉）。但是其它编译器未必这么认为。 事实上，当前的官方标准编译器允许重复的布尔case表达式在同一个switch-case流程控制中出现

（https://github.com/golang/go/issues/28357），而gccgo（v8.2）允许重复的布尔和字符串类型的case表达式在同一个switch-case流程控制中出现。

上面的例子中的前两个case分支中的注释已经解释了，和很多其它语言不一样，每个分支代码块的结尾不需要一条break语句就可以自动跳出当前的switch-case流程控制。那么如何让执行从一个case分支代码块的结尾跳入下一个分支代码块？Go提供了一个fallthrough关键字来完成这个任务。比如，在下面的例子中，所有的分支代码块都将得到执行（从上到下）。

```
 1| rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
 2| switch n := rand.Intn(100) % 5; n {
 3| case 0, 1, 2, 3, 4:
 4|    fmt.Println("n =", n)
 5|    fallthrough // 跳到下个代码块
 6| case 5, 6, 7, 8:
 7|    // 一个新声明的n，它只在当前分支代码快内可见。
 8|    n := 99
 9|    fmt.Println("n =", n) // 99
10|    fallthrough
11| default:
12|    // 下一行中的n和第一个分支中的n是同一个变量。
13|    // 它们均为switch表达式"n"。
14|    fmt.Println("n =", n)
15| }
```

请注意：

- 一条fallthrough语句必须为一个分支代码块中的最后一条语句。
- 一条fallthrough语句不能出现在一个switch-case流程控制中的最后一个分支代码块中。

比如，下面代码的几个fallthrough使用是不合法的。

```
 1| switch n := rand.Intn(100) % 5; n {
 2| case 0, 1, 2, 3, 4:
 3|    fmt.Println("n =", n)
 4|    // 此整个if代码块为当前分支中的最后一条语句
 5|    if true {
 6|       fallthrough // error: 不是当前分支中的最后一条语句
 7|    }
 8| case 5, 6, 7, 8:
 9|    n := 99
10|    fallthrough // error: 不是当前分支中的最后一条语句
11|    _ = n
12| default:
```

```
13|    fmt.Println(n)
14|    fallthrough // error: 不能出现在最后一个分支中
15| }
```

一个switch-case流程控制中的InitSimpleStatement语句和CompareOperand0表达式都是可选的。 如果CompareOperand0表达式被省略，则它被认为类型为bool类型的true值。 如果InitSimpleStatement语句被省略，其后的分号也可一并被省略。

上面已经提到了一个switch-case流程控制中的所有分支都可以被省略，所以下面的所有流程控制代码块都是合法的，它们都可以被视为空操作。

```
 1| switch n := 5; n {
 2| }
 3|
 4| switch 5 {
 5| }
 6|
 7| switch _ = 5; {
 8| }
 9|
10| switch {
11| }
```

上例中的后两个switch-case流程控制中的CompareOperand0表达式都为bool类型的true值。 同理，下例中的代码将打印出hello。

```
1| switch { // <=> switch true {
2| case true: fmt.Println("hello")
3| default: fmt.Println("bye")
4| }
```

Go中另外一个和其它语言的显著不同点是default分支不必一定为最后一个分支。比如，下面的三个switch-case流程控制代码块是相互等价的。

```
 1| switch n := rand.Intn(3); n {
 2| case 0: fmt.Println("n == 0")
 3| case 1: fmt.Println("n == 1")
 4| default: fmt.Println("n == 2")
 5| }
 6|
 7| switch n := rand.Intn(3); n {
 8| default: fmt.Println("n == 2")
 9| case 0: fmt.Println("n == 0")
```

```
10| case 1: fmt.Println("n == 1")
11| }
12|
13| switch n := rand.Intn(3); n {
14| case 0: fmt.Println("n == 0")
15| default: fmt.Println("n == 2")
16| case 1: fmt.Println("n == 1")
17| }
```

goto跳转语句和跳转标签声明

和很多其它语言一样，Go也支持goto跳转语句。 在一个goto跳转语句中， goto关键字后必须跟随一个表明跳转到何处的跳转标签。 我们使用LabelName:这样的形式来声明一个名为LabelName的跳转标签，其中LabelName必须为一个标识符。一个不为空标识符的跳转标签声明后必须被使用至少一次。

一条跳转标签声明之后必须立即跟随一条语句。 如果此声明的跳转标签使用在一条goto语句中，则当此条goto语句被执行的时候，执行将跳转到此跳转标签声明后跟随的语句。

一个跳转标签必须声明在一个函数体内，此跳转标签的使用可以在此跳转标签的声明之后或者之前，但是此跳转标签的使用不能出现在此跳转标签声明所处的最内层代码块之外。

下面这个例子使用跳转标签声明和goto跳转语句来实现了一个循环：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     i := 0
 7|
 8| Next: // 跳转标签声明
 9|     fmt.Println(i)
10|     i++
11|     if i < 5 {
12|         goto Next // 跳转
13|     }
14| }
```

上面刚提到了一个跳转标签的使用不能出现在此跳转标签声明所处的最内层代码块之外，所以下面的代码片段中的跳转标签使用都是不合法的。

```
 1| package main
 2|
 3| func main() {
 4| goto Label1 // error
 5|    {
 6|       Label1:
 7|       goto Label2 // error
 8|    }
 9|    {
10|       Label2:
11|    }
12| }
```

另外要注意的一点是，如果一个跳转标签声明在某个变量的作用域内，则此跳转标签的使用不能出现在此变量的声明之前。 关于变量的作用域，请阅读后面的文章代码块和作用域（第32章）

下面这个程序编译不通过：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    i := 0
 7| Next:
 8|    if i >= 5 {
 9|       // error: goto Exit jumps over declaration of k
10|       goto Exit
11|    }
12|
13|    k := i + i
14|    fmt.Println(k)
15|    i++
16|    goto Next
17| Exit: // 此标签声明在k的作用域内，但
18|       // 它的使用在k的作用域之外。
19| }
```

刚提到的这条规则可能会在今后放宽（https://github.com/golang/go/issues/26058）。 目前，有两种途径可以对上面的程序略加修改以使之编译通过。

第一种途径是缩小变量k的作用域：

```
 1| func main() {
 2|    i := 0
 3| Next:
 4|    if i >= 5 {
 5|       goto Exit
 6|    }
 7|    // 创建一个显式代码块以缩小k的作用域。
 8|    {
 9|       k := i + i
10|       fmt.Println(k)
11|    }
12|    i++
13|    goto Next
14| Exit:
15| }
```

第二种途径是放大变量k的作用域：

```
 1| func main() {
 2|    var k int // 将变量k的声明移到此处。
 3|    i := 0
 4| Next:
 5|    if i >= 5 {
 6|       goto Exit
 7|    }
 8|
 9|    k = i + i
10|    fmt.Println(k)
11|    i++
12|    goto Next
13| Exit:
14| }
```

包含跳转标签的break和continue语句

一个goto语句必须包含一个跳转标签名。 一个break或者continue语句也可以包含一个跳转标签名，但此跳转标签名是可选的。 包含跳转标签名的break语句一般用于跳出外层的嵌套可跳出流程控制代码块。 包含跳转标签名的continue语句一般用于提前结束外层的嵌套循环流程控制代码块的当前循环步。

如果一条break语句中包含一个跳转标签名，则此跳转标签必须刚好声明在一个包含此break语句的可跳出流程控制代码块之前。 我们可以把此跳转标签名看作是其

后紧跟随的可跳出流程控制代码块的名称。 此break语句将立即结束此可跳出流程控制代码块的执行。

如果一条continue语句中包含一个跳转标签名，则此跳转标签必须刚好声明在一个包含此continue语句的循环流程控制代码块之前。 我们可以把此跳转标签名看作是其后紧跟随的循环流程控制代码块的名称。 此continue语句将提前结束此循环流程控制代码块的当前步的执行。

下面是一个使用了包含跳转标签名的break和continue语句的例子。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func FindSmallestPrimeLargerThan(n int) int {
 6| Outer:
 7|    for n++; ; n++{
 8|       for i := 2; ; i++ {
 9|          switch {
10|          case i * i > n:
11|             break Outer
12|          case n % i == 0:
13|             continue Outer
14|          }
15|       }
16|    }
17|    return n
18| }
19|
20| func main() {
21|    for i := 90; i < 100; i++ {
22|       n := FindSmallestPrimeLargerThan(i)
23|       fmt.Print("最小的比", i, "大的素数为", n)
24|       fmt.Println()
25|    }
26| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

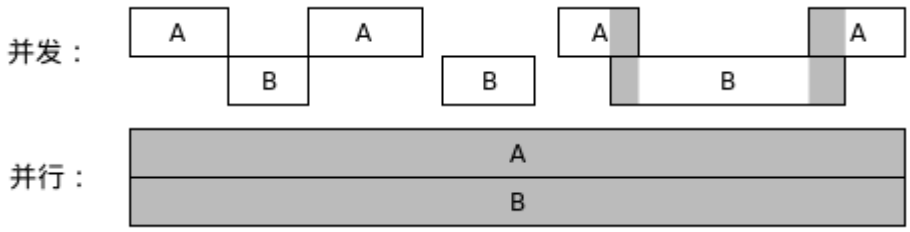
协程、延迟函数调用、以及恐慌和恢复

此篇文章将介绍协程和延迟函数调用。协程和延迟函数调用是Go中比较独特的两个特性。 恐慌和恢复也将在此篇文章中得到简单介绍。本文并非全面地对这些特性进行介绍，后面的其它文章会陆续补全本文的未介绍的内容。

协程（goroutine）

现代CPU一般含有多个核，并且一个核可能支持多线程。换句话说，现代CPU可以同时执行多条指令流水线。 为了将CPU的能力发挥到极致，我们常常需要使我们的程序支持并发（concurrent）计算。

并发计算是指若干计算可能在某些时间片段内同时运行的情形。 下面这两张图描绘了两种并发计算的场景。在此图中，A和B表示两个计算。 在第一种情形中，两个计算只在某些时间片段同时运行。 第二种情形称为并行（parallel）计算。在并行计算中，多个计算在任何时间点都在同时运行。并行计算属于特殊的并发计算。



并发计算可能发生在同一个程序中、同一台电脑上、或者同一个网络中。 在《Go语言101》中，我们只谈及发生在同一个程序中的并发计算。 在Go编程中，协程是创建计算的唯一途径。

协程有时也被称为绿色线程。绿色线程是由程序的运行时（runtime）维护的线程。一个绿色线程的内存开销和情景转换（context switching）时耗比一个系统线程常常小得多。 只要内存充足，一个程序可以轻松支持上万个并发协程。

Go不支持创建系统线程，所以协程是一个Go程序内部唯一的并发实现方式。

每个Go程序启动的时候只有一个对用户可见的协程，我们称之为主协程。 一个协程可以开启更多其它新的协程。在Go中，开启一个新的协程是非常简单的。 我们只需在一个函数调用之前使用一个 go 关键字，即可让此函数调用运行在一个新的协程之中。 当此函数调用退出后，这个新的协程也随之结束了。我们可以称此函数调用为一个协程调用（或者为此协程的启动调用）。 一个协程调用的所有返回值（如果存在的话）必须被全部舍弃。

在下面的例子程序中，主协程创建了两个新的协程。在此例中，`time.Duration`是一个在`time`标准库包中定义的类型。 此类型的底层类型为内置类型`int64`。 底层类型这个概念将在下一篇文章（第14章）中介绍。

```
 1| package main
 2|
 3| import (
 4|    "log"
 5|    "math/rand"
 6|    "time"
 7| )
 8|
 9| func SayGreetings(greeting string, times int) {
10|    for i := 0; i < times; i++ {
11|       log.Println(greeting)
12|       d := time.Second * time.Duration(rand.Intn(5)) / 2
13|       time.Sleep(d) // 睡眠片刻（随机0到2.5秒）
14|    }
15| }
16|
17| func main() {
18|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
19|    log.SetFlags(0)
20|    go SayGreetings("hi!", 10)
21|    go SayGreetings("hello!", 10)
22|    time.Sleep(2 * time.Second)
23| }
```

非常简单！我们编写了一个并发程序！ 此程序在运行的时候在某一时刻将很可能会有三个协程并存。 运行之，可能会得到如下的结果（也可能是其它结果）：

```
hi!
hello!
hello!
hello!
hello!
hi!
```

当一个程序的主协程退出后，此程序也就退出了，即使还有一些其它协程在运行。

和前面的几篇文章不同，上面的例子程序使用了`log`标准库而不是`fmt`标准库中的`Println`函数。 原因是`log`标准库中的打印函数是经过了同步处理的（下一节将解释什么是并发同步），而`fmt`标准库中的打印函数却没有被同步。 如果我们在上例

中使用fmt标准库中的Println函数，则不同协程的打印可能会交织在一起。（虽然对此例来说，交织的概率很低。）

并发同步（concurrency synchronization）

不同的并发计算可能共享一些资源，其中共享内存资源最为常见。 在一个并发程序中，常常会发生下面的情形：

- 在一个计算向一段内存写数据的时候，另一个计算从此内存段读数据，结果导致读出的数据的完整性得不到保证。
- 在一个计算向一段内存写数据的时候，另一个计算也向此段内存写数据，结果导致被写入的数据的完整性得不到保证。

这些情形被称为数据竞争（data race）。并发编程的一大任务就是要调度不同计算，控制它们对资源的访问时段，以使数据竞争的情况不会发生。 此任务常称为并发同步（或者数据同步）。Go支持几种并发同步技术，这些并发同步技术将在后面的章节中逐一介绍。

并发编程中的其它任务包括：

- 决定需要开启多少计算；
- 决定何时开启、阻塞、解除阻塞和结束哪些计算；
- 决定如何在不同的计算中分担工作负载。

上一节中这个并发程序是有缺陷的。我们本期望每个新创建的协程打印出10条问候语，但是主协程（和程序）在这20条问候语还未都打印出来的时候就退出了。 如何确保主协程在这20条问候语都打印完毕之后才退出呢？我们必须使用某种并发同步技术来达成这一目标。

Go支持几种并发同步技术（第36章）。 其中， 通道（第21章）是最独特和最常用的。 但是，为了简单起见，这里我们将使用sync标准库包中的WaitGroup来同步上面这个程序中的主协程和两个新创建的协程。

WaitGroup类型有三个方法（特殊的函数，将在以后的文章中详解）： Add、Done和Wait。 此类型将在后面的某篇文章中详细解释，目前我们可以简单地认为：

- Add方法用来注册新的需要完成的任务数。
- Done方法用来通知某个任务已经完成了。
- 一个Wait方法调用将阻塞（等待）到所有任务都已经完成之后才继续执行其后的语句。

示例：

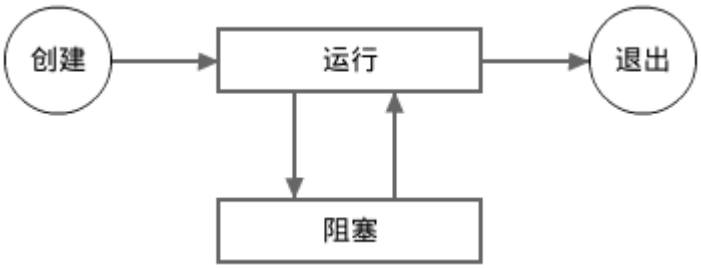
```
 1| package main
 2|
 3| import (
 4|    "log"
 5|    "math/rand"
 6|    "time"
 7|    "sync"
 8| )
 9|
10| var wg sync.WaitGroup
11|
12| func SayGreetings(greeting string, times int) {
13|    for i := 0; i < times; i++ {
14|       log.Println(greeting)
15|       d := time.Second * time.Duration(rand.Intn(5)) / 2
16|       time.Sleep(d)
17|    }
18|    wg.Done() // 通知当前任务已经完成。
19| }
20|
21| func main() {
22|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
23|    log.SetFlags(0)
24|    wg.Add(2) // 注册两个新任务。
25|    go SayGreetings("hi!", 10)
26|    go SayGreetings("hello!", 10)
27|    wg.Wait() // 阻塞在这里，直到所有任务都已完成。
28| }
```

运行这个修改后的程序，我们将会发现所有的20条问候语都将在程序退出之前打印出来。

协程的状态

从上面这个的例子，我们可以看到一个活动中的协程可以处于两个状态：**运行状态**和**阻塞状态**。一个协程可以在这两个状态之间切换。 比如上例中的主协程在调用`wg.Wait`方法的时候，将从运行状态切换到阻塞状态；当两个新协程完成各自的任务后，主协程将从阻塞状态切换回运行状态。

下面的图片显示了一个协程的生命周期。



注意，一个处于睡眠中的（通过调用`time.Sleep`）或者在等待系统调用返回的协程被认为是处于运行状态，而不是阻塞状态。

当一个新协程被创建的时候，它将自动进入运行状态，一个协程只能从运行状态而不能从阻塞状态退出。 如果因为某种原因而导致某个协程一直处于阻塞状态，则此协程将永远不会退出。 除了极个别的应用场景，在编程时我们应该尽量避免出现这样的情形。

一个处于阻塞状态的协程不会自发结束阻塞状态，它必须被另外一个协程通过某种并发同步方法来被动地结束阻塞状态。 如果一个运行中的程序当前所有的协程都出于阻塞状态，则这些协程将永远阻塞下去，程序将被视为死锁了。 当一个程序死锁后，官方标准编译器的处理是让这个程序崩溃。

比如下面这个程序将在运行两秒钟后崩溃。

```
 1| package main
 2|
 3| import (
 4|     "sync"
 5|     "time"
 6| )
 7|
 8| var wg sync.WaitGroup
 9|
10| func main() {
11|     wg.Add(1)
12|     go func() {
13|         time.Sleep(time.Second * 2)
14|         wg.Wait() // 阻塞在此
15|     }()
16|     wg.Wait() // 阻塞在此
17| }
```

它的输出：

```
fatal error: all goroutines are asleep - deadlock!

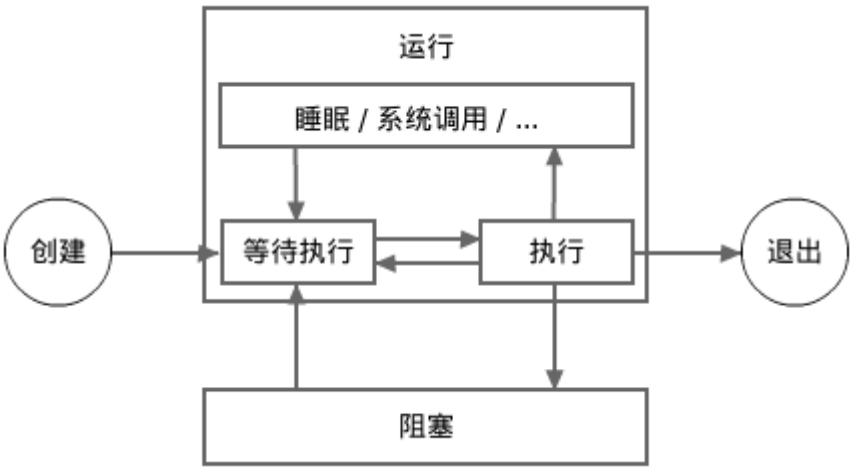
...
```

以后我们将学习到更多可以让一个协程进入到阻塞状态的操作。

协程的调度

并非所有处于运行状态的协程都在执行。在任一时刻，只能最多有和逻辑CPU数目一样多的协程在同时执行。 我们可以调用runtime.NumCPU（https://golang.google.cn/pkg/runtime/#NumCPU）函数来查询当前程序可利用的逻辑CPU数目。 每个逻辑CPU在同一时刻只能最多执行一个协程。Go运行时（runtime）必须让逻辑CPU频繁地在不同的处于运行状态的协程之间切换，从而每个处于运行状态的协程都有机会得到执行。 这和操作系统执行系统线程的原理是一样的。

下面这张图显示了一个协程的更详细的生命周期。在此图中，运行状态被细分成了多个子状态。 一个处于排队子状态的协程等待着进入执行子状态。一个处于执行子状态的协程在被执行一会儿（非常短的时间片）之后将进入排队子状态。



请注意，为了解释的简单性，在以后其它的《Go语言101》文章中，上图中所示的子状态将不会再提及。 重申一下，睡眠和等待系统调用返回子状态被认为是运行状态，而不是阻塞状态。

标准编译器采纳了一种被称为M-P-G模型（https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw）的算法来实现协程调度。 其中，M表示系统线程，P表示逻辑处理器（并非上述的逻辑CPU），G表示协程。 大多数的调度工作是通过逻辑处理器（P）来完成的。 逻辑处理器像一个监工一样通过将不同的处于运行状态协程（G）交给不同的系统线程（M）来执行。 一个协程在同一时刻只能在一个系统线程中执行。一个执行中的协程运行片刻后将自发地脱离让出一个系统线程，从而使得其它处于等待子状态的协程得到执行机会。

在运行时刻，我们可以调用runtime.GOMAXPROCS（https://golang.google.cn/pkg/runtime/#GOMAXPROCS）函数来获取和设置逻辑处理器的数量。 对于官方标准编译器，在Go官方工具链1.5之前，默认初始逻辑处

理器的数量为1；自从Go官方工具链1.5之后，默认初始逻辑处理器的数量和逻辑CPU的数量一致。 此新的默认设置在大多数情况下是最佳选择。但是对于某些文件操作十分频繁的程序，设置一个大于`runtime.NumCPU()`的`GOMAXPROCS`值可能是有好处的。

我们也可以通过设置`GOMAXPROCS`环境变量来设置一个Go程序的初始逻辑处理器数量。

延迟函数调用（deferred function call）

在Go中，一个函数调用可以跟在一个`defer`关键字后面，成为一个延迟函数调用。此`defer`关键字和此延迟函数调用一起形成一个延迟调用语句。 和协程调用类似，被延迟的函数调用的所有返回值（如果存在）必须全部被舍弃。

当一个延迟调用语句被执行时，其中的延迟函数调用不会立即被执行，而是被推入由当前协程维护的一个延迟调用队列（一个后进先出队列，原理等同于栈）。 当一个函数调用返回（此时可能尚未完全退出）并进入它的退出阶段（第9章）后，所有在执行此函数调用的过程中已经被推入延迟调用队列的调用将被按照它们被推入的顺序逆序被弹出队列并执行。 当所有这些延迟调用执行完毕后，此函数调用也就完全退出了。

下面这个例子展示了如何使用延迟调用函数。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     defer fmt.Println("The third line.")
 7|     defer fmt.Println("The second line.")
 8|     fmt.Println("The first line.")
 9| }
```

输出结果：

```
The first line.
The second line.
The third line.
```

下面是另一个略微复杂一点的使用了延迟调用的例子程序。此程序将按照自然数的顺序打印出0到9十个数字。

```
 1| package main
 2|
```

```
 3| import "fmt"
 4|
 5| func main() {
 6|    defer fmt.Println("9")
 7|    fmt.Println("0")
 8|    defer fmt.Println("8")
 9|    fmt.Println("1")
10|    if false {
11|       defer fmt.Println("not reachable")
12|    }
13|    defer func() {
14|       defer fmt.Println("7")
15|       fmt.Println("3")
16|       defer func() {
17|          fmt.Println("5")
18|          fmt.Println("6")
19|       }()
20|       fmt.Println("4")
21|    }()
22|    fmt.Println("2")
23|    return
24|    defer fmt.Println("not reachable")
25| }
```

一个延迟调用可以修改包含此延迟调用的最内层函数的返回值

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func Triple(n int) (r int) {
 6|    defer func() {
 7|       r += n // 修改返回值
 8|    }()
 9|
10|    return n + n // <=> r = n + n; return
11| }
12|
13| func main() {
14|    fmt.Println(Triple(5)) // 15
15| }
```

延迟函数调用的必要性和好处

事实上，上面的几个使用了延迟函数调用的例子中的延迟函数调用并非绝对必要。但是延迟调用对于下面将要介绍的恐慌/恢复特性是必要的。

另外延迟函数调用可以帮助我们写出更整洁和更鲁棒的代码。我们可以在后面的更多关于延迟调用（第29章）一文中读到这样的例子。

协程和延迟调用的实参的估值时刻

一个延迟调用的实参是在此调用对应的延迟调用语句被执行时被估值的。 或者说，它们是在此延迟调用被推入延迟调用队列时被估值的。 这些被估值的结果将在以后此延迟调用被执行的时候使用。

一个匿名函数体内的表达式是在此函数被执行的时候才会被逐渐估值的，不管此函数是被普通调用还是延迟/协程调用。

一个例子：

```
 1| // eval-moment.go
 2| package main
 3|
 4| import "fmt"
 5|
 6| func main() {
 7|    func() {
 8|       var x = 0
 9|       for i := 0; i < 3; i++ {
10|          defer fmt.Println("a:", i + x)
11|       }
12|       x = 10
13|    }()
14|    fmt.Println()
15|    func() {
16|       var x = 0
17|       for i := 0; i < 3; i++ {
18|          defer func() {
19|             fmt.Println("b:", i + x)
20|          }()
21|       }
22|       x = 10
```

```
23|    }()
24| }
```

使用不同版本的Go编译器运行之（gotv（https://gfw.go101.org/apps-and-libs/gotv.html）是一个管理运行多个Go官方工具链版本的工具），将得到如下输出结果：

```
$ gotv 1.21. run eval-moment.go
[Run]: $HOME/.cache/gotv/tag_go1.21.8/bin/go run eval-moment.go
a: 2
a: 1
a: 0

b: 13
b: 13
b: 13
$ gotv 1.22. run eval-moment.go
[Run]: $HOME/.cache/gotv/tag_go1.22.1/bin/go run eval-moment.go
a: 2
a: 1
a: 0

b: 12
b: 11
b: 10
```

请注意Go 1.22对for循环流程控制做出的语义修改（第12章）而导致的代码行为变化。

我们可以对第二个循环略加修改（使用两种方法），使得它和第一个循环打印出相同的结果。

```
 1|       for i := 0; i < 3; i++ {
 2|          defer func(i int) {
 3|             // 此i为形参i，非实参循环变量i。
 4|             fmt.Println("b:", i)
 5|          }(i)
 6|       }
```

或者

```
 1|       for i := 0; i < 3; i++ {
 2|          i := i // 在下面的调用中，左i遮挡了右i。
 3|                  // <=> var i = i
 4|          defer func() {
 5|             // 此i为上面的左i，非循环变量i。
```

```
 6|             fmt.Println("b:", i)
 7|         }()
 8|     }
```

同样的估值时刻规则也适用于协程调用。下面这个例子程序将打印出`123 789`。

```
 1| package main
 2|
 3| import "fmt"
 4| import "time"
 5|
 6| func main() {
 7|    var a = 123
 8|    go func(x int) {
 9|       time.Sleep(time.Second)
10|       fmt.Println(x, a) // 123 789
11|    }(a)
12|
13|    a = 789
14|
15|    time.Sleep(2 * time.Second)
16| }
```

顺便说一句，使用`time.Sleep`调用来做并发同步不是一个好的方法。 如果上面这个程序运行在一个满负荷运行的电脑上，此程序可能在新启动的协程可能还未得到执行机会的时候就已经退出了。 在正式的项目中，我们应该使用并发同步技术（第36章）一文中列出的方法来实现并发同步。

恐慌（panic）和恢复（recover）

Go不支持异常抛出和捕获，而是推荐使用返回值显式返回错误。 不过，Go支持一套和异常抛出/捕获类似的机制。此机制称为恐慌/恢复（panic/recover）机制。

我们可以调用内置函数`panic`来产生一个恐慌以使当前协程进入恐慌状况。

进入恐慌状况是另一种使当前函数调用开始返回的途径。 一旦一个函数调用产生一个恐慌，此函数调用将立即进入它的退出阶段。

通过在一个延迟函数调用之中调用内置函数`recover`，当前协程中的一个恐慌可以被消除，从而使得当前协程重新进入正常状况。

如果一个协程在恐慌状况下退出，它将使整个程序崩溃。

内置函数`panic`和`recover`的声明原型如下：

```
1| func panic(v interface{})
2| func recover() interface{}
```

接口（interface）类型和接口值将在以后的文章接口（第23章）中详解。 目前，我们可以暂时将空接口类型`interface{}`视为很多其它语言中的`any`或者`Object`类型。 换句话说，在一个`panic`函数调用中，我们可以传任何实参值。

一个`recover`函数的返回值为其所恢复的恐慌在产生时被一个`panic`函数调用所消费的参数。

下面这个例子展示了如何产生一个恐慌和如何消除一个恐慌。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     defer func() {
 7|         fmt.Println("正常退出")
 8|     }()
 9|     fmt.Println("嗨！")
10|     defer func() {
11|         v := recover()
12|         fmt.Println("恐慌被恢复了：", v)
13|     }()
14|     panic("拜拜！") // 产生一个恐慌
15|     fmt.Println("执行不到这里")
16| }
```

它的输出结果：

```
嗨！
恐慌被恢复了： 拜拜！
正常退出
```

下面的例子在一个新协程里面产生了一个恐慌，并且此协程在恐慌状况下退出，所以整个程序崩溃了。

```
1| package main
2|
3| import (
4|     "fmt"
5|     "time"
6| )
7|
```

```
 8| func main() {
 9|    fmt.Println("hi!")
10|
11|    go func() {
12|       time.Sleep(time.Second)
13|       panic(123)
14|    }()
15|
16|    for {
17|       time.Sleep(time.Second)
18|    }
19| }
```

运行之，输出如下：

```
hi!
panic: 123

goroutine 5 [running]:
...
```

Go运行时（runtime）会在若干情形下产生恐慌，比如一个整数被0除的时候。下面这个程序将崩溃退出。

```
 1| package main
 2|
 3| func main() {
 4|    a, b := 1, 0
 5|    _ = a/b
 6| }
```

它的输出：

```
panic: runtime error: integer divide by zero

goroutine 1 [running]:
...
```

一般说来，恐慌用来表示正常情况下不应该发生的逻辑错误。 如果这样的一个错误在运行时刻发生了，则它肯定是由于某个bug引起的。 另一方面，非逻辑错误是现实中难以避免的错误，它们不应该导致恐慌。 我们必须正确地对待和处理非逻辑错误。

更多可能由Go运行时产生的恐慌将在以后其它文章中提及。

以后，我们可以了解一些恐慌/恢复用例（第30章）和更多关于恐慌/恢复机制的细节（第31章）。

一些致命性错误不属于恐慌

对于官方标准编译器来说，很多致命性错误（比如栈溢出和内存不足）不能被恢复。它们一旦产生，程序将崩溃。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

Go类型系统概述

本文将介绍Go中的各个类型种类。Go类型系统中的各种概念也将被介绍。如果不熟知这些概念，则很难精通Go编程。

概念：基本类型（basic type）

内置基本类型已经在前面的文章基本类型和它们的字面量表示（第6章）一文中介绍过了。 为了本文的完整性，这些内置类型重新被列在这里：

- 内置字符串类型： `string`.
- 内置布尔类型： `bool`.
- 内置数值类型：
  - `int8`、`uint8`（`byte`）、`int16`、`uint16`、`int32`（`rune`）、`uint32`、`int64`、`uint64`、`int`、`uint`、`uintptr`。
  - `float32`、`float64`。
  - `complex64`、`complex128`。

注意，`byte`是`uint8`的一个内置别名，`rune`是`int32`的一个内置别名。 下面将要提到如何声明自定义的类型别名。

除了字符串类型（第19章），《Go语言101》后续其它文章将不再对其它基本类型做详细讲解。

这17个内置基本类型属于预声明类型（predeclared type）。

概念：组合类型（composite type）

Go支持下列组合类型：

- 指针类型（第15章） - 类C指针
- 结构体类型（第16章） - 类C结构体
- 函数类型（第20章） - 函数类型在Go中是一种一等公民类别
- 容器类型（第18章），包括：
  - 数组类型 - 定长容器类型
  - 切片类型 - 动态长度和容量容器类型
  - 映射类型（map） - 也常称为字典类型。在标准编译器中映射是使用哈希表实现的。
- 通道类型（第21章） - 通道用来同步并发的协程
- 接口类型（第23章） - 接口在反射和多态中发挥着重要角色

无名组合类型可以用它们各自的字面表示形式来表示。 下面是一些各种不同种类的无名组合类型字面表示形式的例子（具名和无名类型将在下面解释）：

```
 1| // 假设T为任意一个类型，Tkey为一个支持比较的类型。
 2|
 3| *T          // 一个指针类型
 4| [5]T        // 一个元素类型为T、元素个数为5的数组类型
 5| []T         // 一个元素类型为T的切片类型
 6| map[Tkey]T // 一个键值类型为Tkey、元素类型为T的映射类型
 7|
 8| // 一个结构体类型
 9| struct {
10|    name string
11|    age  int
12| }
13|
14| // 一个函数类型
15| func(int) (bool, string)
16|
17| // 一个接口类型
18| interface {
19|    Method0(string) int
20|    Method1() (int, bool)
21| }
22|
23| // 几个通道类型
24| chan T
25| chan<- T
26| <-chan T
```

支持和不支持比较的类型（#types-not-support-comparison）将在下面介绍。

事实：类型的种类

每种上面提到的基本类型和组合类型都对应着一个类型种类（kind）。除了这些种类，今后将要介绍的非类型安全指针类型属于另外一个新的类型种类。

所以，目前（Go 1.23），Go有26个类型种类。

语法：类型定义（type definition declaration）

*（**类型定义**又称类型定义声明。在Go 1.9之前，类型定义被称为类型声明并且是唯一的一种类型声明形式。 但是自从Go 1.9，类型定义变成了两种类型声明形式之一。另一种新的类型声明形式为后面的一节中将要介绍的类型别名声明。）*

在Go中，我们可以用如下形式来定义新的类型。在此语法中，`type`为一个关键字。

```
// 定义单个类型。
type NewTypeName SourceType

// 定义多个类型（将多个类型描述合并在一个声明中）。
type (
    NewTypeName1 SourceType1
    NewTypeName2 SourceType2
)
```

新的类型名必须为标识符。但是请注意：包级类型（以及下一节将要介绍的类型别名）的名称不能为[init](#)（第10章）。

上例中的第二个类型声明中包含两个类型描述（type specification）。 如果一个类型声明包含多于一个的类型描述，这些类型描述必须用一对小括号`()`括起来。

每个类型描述创建了一个全新的定义类型（defined type）。

注意：

- 一个新定义的类型和它的源类型为两个不同的类型。
- 在两个不同的类型定义中所定义的两个类型肯定是两个不同的类型。
- 一个新定义的类型和它的源类型的底层类型（将在下面介绍）一致并且它们的值可以相互显式转换。
- 类型定义可以出现在函数体内。

一些类型定义的例子：

```
// 下面这些新定义的类型和它们的源类型都是基本类型。
// 它们的源类型均为预声明类型。
type (
    MyInt int
    Age   int
    Text  string
)

// 下面这些新定义的类型和它们的源类型都是组合类型。
// 它们的源类型均为无名类型（见下下节）。
type IntPtr *int
type Book struct{author, title string; pages int}
type Convert func(in0 int, in1 bool)(out0 int, out1 string)
```

```
14| type StringArray [5]string
15| type StringSlice []string
16|
17| func f() {
18|     // 这三个新定义的类型名称只能在此函数内使用。
19|     type PersonAge map[string]int
20|     type MessageQueue chan string
21|     type Reader interface{Read([]byte) int}
22| }
```

请注意：从Go 1.9到Go 1.17，Go白皮书曾经把预声明类型视为定义类型。 但是从Go 1.18开始，Go白皮书明确说明预声明类型不再属于定义类型。

概念：自定义泛型类型和实例化类型（generic type and instantiated types）

从Go 1.18开始，Go开始支持自定义泛型类型（和函数）。 一个泛型类型必须被实例化才能被用做值类型。

一个泛型类型是一个定义类型；它的实例化类型为具名类型。具名类型将在下一节解释。

自定义泛型中的另外两个重要的概念为类型约束（constarint）和类型参数（type parameter）。

本书不详细阐述自定义泛型。关于如何声明和使用泛型类型和函数，请阅读《Go自定义泛型101》（https://gfw.go101.org/generics/101.html）。

概念：具名类型和无名类型（named type and unnamed type）

在Go 1.9之前，**具名类型**这个术语在Go白皮书中是精确定义的。 在那时，一个具名类型被定义为一个可以用标识符表示的类型。 随着在Go 1.9中引入了自定义类型别名（见下一节），**具名类型**这个术语被从白皮书中删除了；取而代之的是**定义类型**。 随着Go 1.18中引入了自定义泛型，**具名类型**这个术语又被重新加回到白皮书。

一个具名类型可能为

- 一个预声明类型（不包括类型别名（#type-alias））；
- 一个定义（非自定义泛型）类型；
- 一个（泛型类型的）实例化类型；
- 一个类型参数类型（使用在自定义泛型中）。

其它类型称为无名类型。一个无名类型肯定是一个组合类型（反之则未必）。

语法：类型别名声明（type alias declaration）

从Go 1.9开始，我们可以使用下面的语法来声明自定义类型别名。此语法和类型定义类似，但是请注意每个类型描述中多了一个等号=。

```
1| type (
2|    Name = string
3|    Age  = int
4| )
5|
6| type table = map[string]int
7| type Table = map[Name]Age
```

类型别名也必须为标识符。同样地，类型别名可以被声明在函数体内。

在上面的类型别名声明的例子中，Name是内置类型string的一个别名，它们表示同一个类型。 同样的关系对下面的几对类型表示也成立：

- 别名Age和内置类型int。
- 别名table和映射类型map[string]int。
- 别名Table和映射类型map[Name]Age。

事实上，文字表示形式map[string]int和map[Name]Age也表示同一类型。 所以，table和Table一样表示同一个类型。

注意：尽管一个类型别名有一个名字，但是它可能表示一个无名类型。 比如，table和Table这两个别名都表示同一个无名类型map[string]int。

概念：底层类型（underlying type）

在Go中，每个类型都有一个底层类型。规则：

- 一个内置类型的底层类型为它自己。
- unsafe标准库包中定义的Pointer类型的底层类型是它自己。 （至少我们可以认为是这样。事实上，关于unsafe.Pointer类型的底层类型，官方文档中并没有清晰的说明。我们也可以认为unsafe.Pointer类型的底层类型为*T，其中T表示一个任意类型。） unsafe.Pointer也被视为一个内置类型。

- 一个无名类型（必为一个组合类型）的底层类型为它自己。
- 在一个类型声明中，新声明的类型和源类型共享底层类型。

一个例子：

```
 1| // 这四个类型的底层类型均为内置类型int。
 2| type (
 3|     MyInt int
 4|     Age   MyInt
 5| )
 6|
 7| // 下面这三个新声明的类型的底层类型各不相同。
 8| type (
 9|     IntSlice   []int   // 底层类型为[]int
10|     MyIntSlice []MyInt // 底层类型为[]MyInt
11|     AgeSlice   []Age   // 底层类型为[]Age
12| )
13|
14| // 类型[]Age、Ages和AgeSlice的底层类型均为[]Age。
15| type Ages AgeSlice
```

如何溯源一个声明的类型的底层类型？规则很简单，在溯源过程中，当遇到一个内置类型或者无名类型时，溯源结束。 以上面这几个声明的类型为例，下面是它们的底层类型的溯源过程：

```
MyInt → int
Age → MyInt → int
IntSlice → []int
MyIntSlice → []MyInt → ~~[]int~~
AgeSlice → []Age → ~~[]MyInt~~ → ~~[]int~~
Ages → AgeSlice → []Age → ~~[]MyInt~~ → ~~[]int~~
```

在Go中，

- 底层类型为内置类型`bool`的类型称为**布尔类型**；
- 底层类型为任一内置整数类型的类型称为**整数类型**；
- 底层类型为内置类型`float32`或者`float64`的类型称为**浮点数类型**；
- 底层类型为内置类型`complex64`或`complex128`的类型称为**复数类型**；
- 整数类型、浮点数类型和复数类型统称为**数字值类型**；
- 底层类型为内置类型`string`的类型称为**字符串类型**。

底层类型这个概念在类型转换、赋值和比较规则（第48章）中扮演着重要角色。

概念：值（value）

一个类型的一个实例称为此类型的一个值。一个类型可以有很多不同的值，其中一个为它的零值。 同一类型的不同值共享很多相同的属性。

每个类型有一个零值。一个类型的零值可以看作是此类型的默认值。 预声明的标识符`nil`可以看作是切片、映射、函数、通道、指针（包括非类型安全指针）和接口类型的零值的字面量表示。 我们以后可以在Go中的nil（第47章）一文中了解到关于`nil`的各种事实。

在源代码中，值可以呈现为若干种形式，包括字面量（第6章）、具名常量（第7章）、变量（第7章）和表达式（第11章）。前三种形式可以看作是最后一种形式的特例。

值分为类型确定的和类型不确定的（第7章）。

基本类型和它们的字面量表示已经在前面一文（第6章）中介绍过了。 另外，Go中还有另外两种的字面量表示形式：函数字面量表示形式和组合字面量表示形式（composite literal）。

函数字面量表示形式用来表示函数值。事实上，一个函数声明（第9章）是由一个标识符（函数名）和一个函数字面量表示形式组成。

组合字面量表示形式用来表示结构体类型值和容器类型（数组、切片和映射）值。详见结构体（第16章）和容器类型（第18章）两文。

指针类型、通道类型和接口类型的值没有字面量表示形式。

概念：值部（value part）

在运行时刻，很多值是存储在内存的。每个这样的值都有一个直接部分，但是有一些值还可能有一个或多个间接部分。每个值部分在内存中都占据一段连续空间。 通过安全（第15章）或者非安全（第25章）指针，一个值的间接部分被此值的直接部分所引用。

值部（第17章）这个术语并没有在Go白皮书中定义。它仅使用在《Go语言101》这本书中，用来简化一些解释并帮助Go程序员更好地理解Go类型和值。

概念：值尺寸（value size）

一个值存储在内存中是要占据一定的空间的。此空间的大小称为此值的尺寸。值尺寸是用字节数来衡量的。 在Go中，当我们谈及一个值的尺寸，如果没有特殊说明，我们一般是指此值的直接部分的尺寸。 某个特定类别的所有类型的值的尺寸都是一样的。因为这个原因，我们也常将一个值的尺寸说成是它的类型的尺寸（或值尺寸）。

我们可以用unsafe标准库包中的Sizeof函数来取得任何一个值的尺寸。

Go白皮书没有规定非数值类型值的尺寸。对数值类型值的尺寸的要求已经在基本类型和它们的字面量表示（第6章）一文中提及了。

概念：指针类型的基类型（base type）

如果一个指针类型的底层类型表示为*T，则此指针类型的基类型为T所表示的类型。

指针类（第15章）一文详细解释了指针类类型和指针值。

概念：结构体类型的字段（field）

一个结构体类型由若干成员变量组成。每个这样的成员变量称为此结构体的一个字段。 比如，下面这个结构体类型含有三个字段： author、title和pages。

```
1| struct {
2|     author string
3|     title  string
4|     pages  int
5| }
```

结构体（第16章）一文详细解释了结构体类型和结构体值。

概念：函数类型的签名（signature）

一个函数和其类型的签名由此函数的输入参数和返回结果的类型列表组成。 函数名称和函数体不属于函数签名的构成部分。

函数（第20章）一文详细解释了函数类型和函数值。

概念：类型的方法（method）和方法集（method set）

在Go中，我们可以给满足某些条件的类型声明方法（第22章）。方法也常被称为成员函数。 一个类型的所有方法组成了此类型的方法集。

概念：接口类型的动态类型和动态值

接口类型的值称为接口值。一个接口值可以包裹装载一个非接口值。包裹在一个接口值中的非接口值称为此接口值的动态值。此动态值的类型称为此接口值的动态类型。 一个什么也没包裹的接口值为一个零值接口值。零值接口值的动态值和动态类型均为不存在。

一个接口类型可以指定若干个（可以是零个）方法，这些方法形成了此接口类型的方法集。

如果一个类型（可以是接口或者非接口类型）的方法集是一个接口类型的方法集的超集，则我们说此类型实现（第23章）了此接口类型。

接口（第23章）一文详细解释了接口类型和接口值。

概念：一个值的具体类型（concrete type）和具体值（concrete value）

对于一个（类型确定的）非接口值，它的具体类型就是它的类型，它的具体值就是它自己。

一个零值接口值没有具体类型和具体值。 对于一个非零值接口值，它的具体类型和具体值就是它的动态类型和动态值。

概念：容器类型

数组、切片和映射是Go中的三种正式意义上的内置容器类型。

有时候，字符串和通道类型也可以被非正式地看作是容器类型。

（正式和非正式的）容器类型的每个值都有一个长度属性。

数组、切片和映射（第18章）一文详细解释了各种正式容器类型和它们的值。

概念：映射类型的键值（key）类型

如果一个映射类型的底层类型表示为`map[Tkey]T`，则此映射类型的键值类型为`Tkey`。 `Tkey`必须为一个可比较类型（见下）。

概念：容器类型的元素（element）类型

存储在一个容器值中的所有元素的类型必须为同一个类型。此同一类型称为此容器值的（容器）类型的元素类型。

- 如果一个数组类型的底层类型表示为`[N]T`，则此数组类型的元素类型为`T`所表示的类型。
- 如果一个切片类型的底层类型表示为`[]T`，则此切片类型的元素类型为`T`所表示的类型。
- 如果一个映射类型的底层类型表示为`map[Tkey]T`，则此映射类型的元素类型为`T`所表示的类型。
- 如果一个通道类型的底层类型表示为`chan T`、`chan<- T`或者`<-chan T`，则此通道类型的元素类型为`T`所表示的类型。
- 一个字符串类型的元素类型总是内置类型`byte`（亦即`uint8`）。

概念：通道类型的方向

一个通道值可以被看作是先入先出（first-in-first-out，FIFO）队列。一个通道值可能是可读可写的、只读的（receive-only）或者只写的（send-only）。

- 一个可读可写的通道值也称为一个双向通道。 一个双向通道类型的底层类型可以被表示为`chan T`。
- 我们只能向一个只写的通道值发送数据，而不能从其中接收数据。 只写通道类型的底层类型可以被表示为`chan<- T`。
- 我们只能从一个只读的通道值接收数据，而不能向其发送数据。 只读通道类型的底层类型可以被表示为`<-chan T`。

通道（第21章）一文详细解释了通道类型和通道值。

事实：可比较类型和不可比较类型

目前（Go 1.23），下面这些类型的值不支持（使用`==`和`!=`运算标识符）比较。这些类型称为不可比较类型。

- 切片类型
- 映射类型
- 函数类型
- 任何包含有不可比较类型的字段的结构体类型和任何元素类型为不可比较类型的数组类型。

其它类型称为可比较类型。

映射类型的键值类型必须为可比较类型。

我们可以在类型转换、赋值和值比较规则大全（第48章）一文中了解到更详细的比较规则。

事实：Go对面向对象编程（object-oriented programming）的支持

Go并不全面支持面向对象编程，但是Go确实支持一些面向对象编程的元素。请阅读以下几篇文章以获取详细信息：

- 方法（第22章）
- 实现（第23章）
- 类型内嵌（第24章）

事实：Go对泛型（generics）的支持

在1.18版本以前，Go中泛型支持只局限在内置类型和内置函数中。 从1.18版本开始，Go也支持自定义泛型。 请阅读泛型（第26章）一文来了解内置泛型和《Go自定义泛型101》（https://gfw.go101.org/generics/101.html）一书来了解自定义泛型。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go101/go101获取本书最新版）

指针

虽然Go吸收融合了很多其语言中的各种特性，但是Go主要被归入C语言家族。其中一个重要的原因就是Go和C一样，也支持指针。 当然Go中的指针相比C指针有很多限制。本篇文章将介绍指针相关的各种概念和Go指针相关的各种细节。

内存地址

在编程中，一个内存地址用来定位一段内存。

通常地，一个内存地址用一个操作系统原生字（native word）来存储。 一个原生字在32位操作系统上占4个字节，在64位操作系统上占8个字节。 所以，32位操作系统上的理论最大支持内存容量为4GB（1GB == 2^{30}字节），64位操作系统上的理论最大支持内存容量为2^{64}Byte，即16EB（EB：艾字节，1EB == 1024PB，1PB == 1024TB，1TB == 1024GB）。

内存地址的字面形式常用整数的十六进制字面量来表示，比如`0x1234CDEF`。

以后我们常简称内存地址为地址。

值的地址

一个值的地址是指此值的直接部分占据的内存的起始地址。在Go中，每个值都包含一个直接部分，但有些值可能还包含一个或多个间接部分，下下章（第17章）将对此详述。

什么是指针?

指针是Go中的一种类型分类（kind）。 一个指针可以存储一个内存地址；从地址通常为另外一个值的地址。

和C指针不一样，为了安全起见，Go指针有很多限制，详见下面的章节。

指针类型和值

在Go中，一个无名指针类型的字面形式为`*T`，其中`T`为一个任意类型。类型`T`称为指针类型`*T`的基类型（base type）。 如果一个指针类型的基类型为`T`，则我们可以称此指针类型为一个`T`指针类型。

虽然我们可以声明具名指针类型，但是一般不推荐这么做，因为无名指针类型的可读性更高。

如果一个指针类型的底层类型（第14章）是*T，则它的基类型为T。

如果两个无名指针类型的基类型为同一类型，则这两个无名指针类型亦为同一类型。

一些指针类型的例子：

```
1| *int  // 一个基类型为int的无名指针类型。
2| **int // 一个多级无名指针类型，它的基类型为*int。
3|
4| type Ptr *int // Ptr是一个具名指针类型，它的基类型为int。
5| type PP *Ptr  // PP是一个具名多级指针类型，它的基类型为Ptr。
```

指针类型的零值的字面量使用预声明的nil来表示。一个nil指针（常称为空指针）中不存储任何地址。

如果一个指针类型的基类型为T，则此指针类型的值只能存储类型为T的值的地址。

关于引用（reference）这个术语

在《Go语言101》中，术语“引用”暗示着一个关系。比如，如果一个指针中存储着另外一个值的地址，则我们可以说此指针值引用着另外一个值；同时另外一个值当前至少有一个引用。 本书对此术语的使用和Go白皮书是一致的。

当一个指针引用着另外一个值，我们也常说此指针指向另外一个值。

如何获取一个指针值?

有两种方式来得到一个指针值：

1. 我们可以用内置函数new来为任何类型的值开辟一块内存并将此内存块的起始地址做为此值的地址返回。 假设T是任一类型，则函数调用new(T)返回一个类型为*T的指针值。 存储在返回指针值所表示的地址处的值（可被看作是一个匿名变量）为T的零值。
2. 我们也可以使用前置取地址操作符&来获取一个可寻址的值的地址。 对于一个类型为T的可寻址的值t，我们可以用&t来取得它的地址。&t的类型为*T。

一般说来，一个可寻址的值是指被放置在内存中某固定位置处的一个值（但放置在某固定位置处的一个值并非一定是可寻址的）。 目前，我们只需知道所有变量都是

可以寻址的；但是所有常量、函数返回值和强制转换结果都是不可寻址的。 当一个变量被声明的时候，Go运行时将为此变量开辟一段内存。此内存的起始地址即为此变量的地址。

更多可被（或不可被）寻址的值将在以后的文章中逐渐提及。 如果你已经对Go比较熟悉，你可以阅读此条总结（第46章）来了解在Go中哪些值可以或不可以被寻址。

下一节中的例子将展示如何获取一些值的地址。

指针（地址）解引用

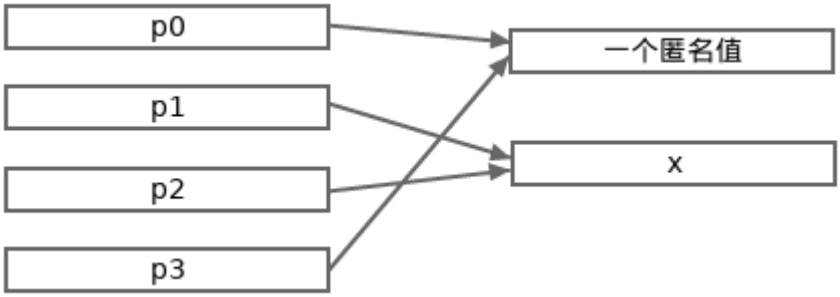
我们可以使用前置解引用操作符 * 来访问存储在一个指针所表示的地址处的值（即此指针所引用着的值）。 比如，对于基类型为 T 的指针类型的一个指针值 p，我们可以用 *p 来表示地址 p 处的值。 此值的类型为 T。 *p 称为指针 p 的解引用。解引用是取地址的逆过程。

解引用一个nil指针将产生一个恐慌（第13章）。

下面这个例子展示了如何取地址和解引用。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     p0 := new(int)   // p0指向一个int类型的零值
 7|     fmt.Println(p0)  //  (打印出一个十六进制形式的地址)
 8|     fmt.Println(*p0) // 0
 9|
10|     x := *p0                // x是p0所引用的值的一个复制。
11|     p1, p2 := &x, &x        // p1和p2中都存储着x的地址。
12|                             // x、*p1和*p2表示着同一个int值。
13|     fmt.Println(p1 == p2) // true
14|     fmt.Println(p0 == p1) // false
15|     p3 := &*p0             // <=> p3 := &(*p0)
16|                             // <=> p3 := p0
17|                             // p3和p0中存储的地址是一样的。
18|     fmt.Println(p0 == p3) // true
19|     *p0, *p1 = 123, 789
20|     fmt.Println(*p2, x, *p3) // 789 789 123
21|
22|     fmt.Printf("%T, %T \n", *p0, x) // int, int
23|     fmt.Printf("%T, %T \n", p0, p1) // *int, *int
24| }
```

下面这张图描绘了上面这个例子中各个值之间的关系。



我们为什么需要指针?

让我们先看一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func double(x int) {
 6|     x += x
 7| }
 8|
 9| func main() {
10|     var a = 3
11|     double(a)
12|     fmt.Println(a) // 3
13| }
```

我们本期望上例中的`double`函数将变量`a`的值放大为原来的两倍，但是事实证明我们的期望没有得到实现。 为什么呢? 因为在Go中，所有的赋值（包括函数调用传参）过程都是一个值复制过程。 所以在上面的`double`函数体内修改的是变量`a`的一个副本，而没有修改变量`a`本身。

当然我们可以让`double`函数返回输入参数的两倍数，但是此方法并非适用于所有场合。 下面这个例子通过将输入参数的类型改为一个指针类型来达到同样的目的。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func double(x *int) {
 6|     *x += *x
 7|     x = nil // 此行仅为讲解目的
 8| }
```

```
 9|
10| func main() {
11|    var a = 3
12|    double(&a)
13|    fmt.Println(a) // 6
14|    p := &a
15|    double(p)
16|    fmt.Println(a, p == nil) // 12 false
17| }
```

从上例可以看出，通过将`double`函数的输入参数的类型改为`*int`，传入的实参`&a`和它在此函数体内的一个副本`x`都引用着变量`a`。 所以对`*x`的修改等价于对`*p`（也就是变量`a`）的修改。 换句话说，新版本的`double`函数内的操作可以反映到此函数外了。

当然，在此函数体内对传入的指针实参的修改`x = nil`依旧不能反映到函数外，因为此修改发生在此指针的一个副本上。 所以在`double`函数调用之后，局部变量`p`的值并没有被修改为`nil`。

简而言之，指针提供了一种间接的途径来访问和修改一些值。 虽然很多语言中没有指针这个概念，但是指针被隐藏其它概念之中。

在Go中返回一个局部变量的地址是安全的

和C不一样，Go是支持垃圾回收的，所以一个函数返回其内声明的局部变量的地址是绝对安全的。比如：

```
1| func newInt() *int {
2|    a := 3
3|    return &a
4| }
```

Go指针的一些限制

为了安全起见，Go指针在使用上相对于C指针有很多限制。 通过施加这些限制，Go指针保留了C指针的好处，同时也避免了C指针的危险性。

Go指针不支持算术运算

在Go中，指针是不能参与算术运算的。比如，对于一个指针`p`， 运算`p++`和`p-2`都是非法的。

如果p为一个指向一个数值类型值的指针，*p++将被编译器认为是合法的并且等价于(*p)++。 换句话说，解引用操作符*的优先级都高于自增++和自减--操作符。

例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     a := int64(5)
 7|     p := &a
 8|
 9|     // 下面这两行编译不通过。
10|     /*
11|     p++
12|     p = (&a) + 8
13|     */
14|
15|     *p++
16|     fmt.Println(*p, a)   // 6 6
17|     fmt.Println(p == &a) // true
18|
19|     *&a++
20|     *&*&a++
21|     **&p++
22|     *&*p++
23|     fmt.Println(*p, a) // 10 10
24| }
```

一个指针类型的值不能被随意转换为另一个指针类型

在Go中，只有如下某个条件被满足的情况下，一个类型为T1的指针值才能被显式转换为另一个指针类型T2：

1. 类型T1和T2的底层类型必须一致（忽略结构体字段的标签）。 特别地，如果类型T1和T2中只要有一个是无名类型（第14章）并且它们的底层类型一致（考虑结构体字段的标签），则此转换可以是隐式的。 关于结构体，请参阅下一篇文章（第16章）。
2. 类型T1和T2都为无名类型并且它们的基类型的底层类型一致（忽略结构体字段的标签）。

比如，

```
 1| type MyInt int64
 2| type Ta    *int64
 3| type Tb    *MyInt
```

对于上面所示的这些指针类型，下面的事实成立：

1. 类型*int64的值可以被隐式转换到类型Ta，反之亦然（因为它们的底层类型均为*int64）。
2. 类型 *MyInt的值可以被隐式转换到类型Tb，反之亦然（因为它们的底层类型均为*MyInt）。
3. 类型*MyInt的值可以被显式转换为类型*int64，反之亦然（因为它们都是无名的并且它们的基类型的底层类型均为int64）。
4. 类型Ta的值不能直接被转换为类型Tb，即使是显式转换也是不行的。 但是，通过上述三条事实，通过三层显式转换Tb((*MyInt)((*int64)(ta)))，一个类型为Ta的值ta可以被间接地转换为类型Tb。

这些指针类型的任何值都无法被转换到类型*uint64。

一个指针值不能和其它任一指针类型的值进行比较

Go指针值是支持（使用比较运算符==和!=）比较的。 但是，两个指针只有在下列任一条件被满足的时候才可以比较：

1. 这两个指针的类型相同。
2. 其中一个指针可以被隐式转换为另一个指针的类型。换句话说，这两个指针的类型的底层类型必须一致并且至少其中一个指针类型为无名的（考虑结构体字段的标签）。
3. 其中一个并且只有一个指针用类型不确定的nil标识符表示。

例子：

```
 1| package main
 2|
 3| func main() {
 4|    type MyInt int64
 5|    type Ta    *int64
 6|    type Tb    *MyInt
 7|
 8|    // 4个不同类型的指针：
 9|    var pa0 Ta
10|    var pa1 *int64
11|    var pb0 Tb
12|    var pb1 *MyInt
```

```
13|
14|    // 下面这6行编译没问题。它们的比较结果都为true。
15|    _ = pa0 == pa1
16|    _ = pb0 == pb1
17|    _ = pa0 == nil
18|    _ = pa1 == nil
19|    _ = pb0 == nil
20|    _ = pb1 == nil
21|
22|    // 下面这三行编译不通过。
23|    /*
24|    _ = pa0 == pb0
25|    _ = pa1 == pb1
26|    _ = pa0 == Tb(nil)
27|    */
28| }
```

一个指针值不能被赋值给其它任意类型的指针值

一个指针值可以被赋值给另一个指针值的条件和这两个指针值可以比较的条件（见上一小节）是一致的。

上述Go指针的限制是可以被打破的

`unsafe`标准库包（第25章）中提供的非类型安全指针（`unsafe.Pointer`）机制可以被用来打破上述Go指针的安全限制。 `unsafe.Pointer`类型类似于C语言中的`void*`。 但是，通常地，非类型安全指针机制不推荐在Go日常编程中使用。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

结构体

和C语言类似，Go也支持结构体类型。此篇文章将介绍Go中的结构体类型和结构体值做一个详细的解释。

结构体类型和结构体字面量表示形式

每个无名结构体类型的字面形式均由`struct`关键字开头，后面跟着用一对大括号`{}`，其中包裹着的一系列字段（field）声明。 一般来说，每个字段声明由一个字段名和字段类型组成。一个结构体类型的字段数目可以为0。下面是一个无名结构体类型的字面形式：

```
1| struct {
2|     title  string
3|     author string
4|     pages  int
5| }
```

上面这个结构体类型含有三个字段。前两个字段（`title`和`author`）的类型均为`string`。 最后一个字段`pages`的类型为`int`。

有时字段也称为成员变量。

相邻的同类型字段可以声明在一起。比如上面这个类型也可表示成下面这样：

```
1| struct {
2|     title, author string
3|     pages         int
4| }
```

一个结构体类型的尺寸为它的所有字段的（类型）尺寸之和加上一些填充字节的数目。 常常地，编译器（和运行时）会在一个结构体值的两个相邻字段之间填充一些字节来保证一些字段的地址总是某个整数的倍数。 我们可以在后面的内存布局（第44章）一文中了解到字节填充（padding）和内存地址对齐（memory address alignment）。

一个零字段结构体的尺寸为零。

每个结构体字段在它的声明中可以被指定一个标签（tag）。从语法上讲，字段标签可以是任意字符串，它们是可选的，默认为空字符串。 但在实践中，它们应该被表

示成用空格分隔的键值对形式，并且每个标签尽量使用直白字面形式（`...`）表示，而键值对中的值使用解释型字面形式（"..."）表示。 比如下例：

```
1| struct {
2|     Title  string `json:"title" myfmt:"s1"`
3|     Author string `json:"author,omitempty" myfmt:"s2"`
4|     Pages  int    `json:"pages,omitempty" myfmt:"n1"`
5|     X, Y   bool   `myfmt:"b1"`
6| }
```

注意：上例中的X和Y字段的标签是一样的（尽管在实践中基本上从不会这样使用字段标签）。

我们可以使用反射（第27章）来检视字段的标签信息。

每个字段标签的目的取决于具体应用。上面这个例子中的字段标签用来帮助encoding/json标准库包来将上面这个结构体类型的某个值编码成JSON数据或者从一份JSON数据解码到上面这个结构体类型的某个值中。在编码和解码过程中，encoding/json标准库包中的函数将只考虑导出的结构体字段。这是为什么上面这个结构体的字段均为导出的。

把字段标签当成字段注释来使用不是一个好主意。

和C语言不一样，Go结构体不支持字段联合（union）。

上面的例子中展示的结构体类型都是无名的。在实践中，具名结构体类型用得更流行。

只有导出字段可以被使用在其它代码包中。非导出字段类以于很多其它语言中的私有或者保护型的成员变量。

一个结构体类型中的字段标签和字段的声明顺序对此结构体类型的身份识别很重要。 如果两个无名结构体类型的各个对应字段声明都相同（按照它们的出现顺序），则此两个无名结构体类型是等同的。 两个字段声明只有在它们的名称、类型、标签以及内嵌性（第24章）都等同的情况下才相同。 注意：**两个声明在不同的代码包中的非导出字段将总被认为是不同的字段。**

一个结构体类型不能（直接或者间接）含有一个类型为此结构类型的字段。

结构体字面量表示形式和结构体值的使用

在Go中，语法形式T{...}称为一个组合字面量形式（composite literal），其中T必须为一个类型名或者类型字面形式。 组合字面量形式可以用来表示结构体类型和内置容器类型（将在后面的文章中介绍）的值。

注意：组合字面量`T{...}`是一个类型确定值，它的类型为`T`。

假设`S`是一个结构体类型并且它的底层类型为`struct{x int; y bool}`，`S`的零值可以表示成下面所示的组合字面量两种变种形式：

1. `S{0, false}`。在此变种形式中，所有的字段名称均不出现，但每个字段的值必须指定，并且每个字段的出现顺序和它们的声明顺序必须一致。
2. `S{x: 0, y: false}`、`S{y: false, x: 0}`、`S{x: 0}`、`S{y: false}`和`S{}`。 在此变种形式中，字段的名称和值必须成对出现，但是每个字段都不是必须出现的，并且字段的出现顺序并不重要。 没有出现的字段的值被编译器认为是它们各自类型的零值。`S{}`是最常用的类型`S`的零值的表示形式。

如果`S`是声明在另一个代码包中的一个结构体类型，则推荐使用上面所示的第二种变种形式来表示它的值。 因为另一个代码包的维护者今后可能会在此结构体中添加新的字段，从而导致当前使用的第一种变种形式在今后可能编译不通过。

当然，上面所示的结构体值的组合字面量也可以用来表示结构体类型的非零值。

对于类型`S`的一个值`v`，我们可以用`v.x`和`v.y`来表示它的字段。 `v.x`（或`v.y`）这种形式称为一个选择器（selector）。其中的`v`称为此选择器的属主。 今后，我们称一个选择器中的句点`.`为属性选择操作符。

一个例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5| )
 6|
 7| type Book struct {
 8|     title, author string
 9|     pages         int
10| }
11|
12| func main() {
13|     book := Book{"Go语言101", "老貘", 256}
14|     fmt.Println(book) // {Go语言101 老貘 256}
15|
16|     // 使用带字段名的组合字面量来表示结构体值。
17|     book = Book{author: "老貘", pages: 256, title: "Go语言
101"}
18|     // title和author字段的值都为空字符串""，pages字段的值为0。
19|     book = Book{}
20|     // title字段空字符串""，pages字段为0。
```

```
21|    book = Book{author: "老貘"}
22|
23|    // 使用选择器来访问和修改字段值。
24|    var book2 Book // <=> book2 := Book{}
25|    book2.author = "Tapir"
26|    book2.pages = 300
27|    fmt.Println(book2.pages) // 300
28| }
```

如果一个组合字面量中最后一项和结尾的`}`处于同一行，则此项后的逗号`,`是可选的；否则此逗号不可省略。 我们可以阅读后面的Go代码断行规则（第28章）一文了解更多断行规则。

```
 1| var _ = Book {
 2|    author: "老貘",
 3|    pages: 256,
 4|    title: "Go语言101", // 这里行尾的逗号不可省略
 5| }
 6|
 7| // 下行}前的逗号可以省略。
 8| var _ = Book{author: "老貘", pages: 256, title: "Go语言
101",}
```

关于结构体值的赋值

当一个（源）结构体值被赋值给另外一个（目标）结构体值时，其效果和逐个将源结构体值的各个字段赋值给目标结构体值的各个对应字段的效果是一样的。

```
 1| func f() {
 2|    book1 := Book{pages: 300}
 3|    book2 := Book{"Go语言101", "老貘", 256}
 4|
 5|    book2 = book1
 6|    // 上面这行和下面这三行是等价的。
 7|    book2.title = book1.title
 8|    book2.author = book1.author
 9|    book2.pages = book1.pages
10| }
```

如果两个结构体值的类型不同，则只有在它们的底层类型相同（要考虑字段标签）并且其中至少有一个结构体值的类型为无名类型（第14章）时（换句话说，只有它们可以被隐式转换为对方的类型的时候，见下）才可以互相赋值。

结构体字段的可寻址性

如果一个结构体值是可寻址的，则它的字段也是可寻址的；反之，一个不可寻址的结构体值的字段也是不可寻址的。 不可寻址的字段的值是不可更改的。所有的组合字面量都是不可寻址的。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    type Book struct {
 7|       Pages int
 8|    }
 9|    var book = Book{} // 变量值book是可寻址的
10|    p := &book.Pages
11|    *p = 123
12|    fmt.Println(book) // {123}
13|
14|    // 下面这两行编译不通过，因为Book{}是不可寻址的，
15|    // 继而Book{}.Pages也是不可寻址的。
16|    /*
17|    Book{}.Pages = 123
18|    p = &Book{}.Pages // <=> p = &(Book{}.Pages)
19|    */
20| }
```

注意：选择器中的属性选择操作符 . 的优先级比取地址操作符 & 的优先级要高。

组合字面量不可寻址但可被取地址

一般来说，只有可被寻址的值才能被取地址，但是Go中有一个语法糖（语法例外）：虽然所有的组合字面量都是不可寻址的，但是它们都可被取地址。

例子：

```
 1| package main
 2|
 3| func main() {
 4|    type Book struct {
 5|       Pages int
 6|    }
```

```
 7|    // Book{100}是不可寻址的，但是它可以被取地址。
 8|    p := &Book{100} // <=> tmp := Book{100}; p := &tmp
 9|    p.Pages = 200
10| }
```

在字段选择器中，属主结构体值可以是指针，它将被隐式解引用

比如，在下面的例子中，为了简洁，(*bookN).pages可以被写成bookN.pages。换句话说，在这种简写形式中，bookN将被隐式解引用。

```
 1| package main
 2|
 3| func main() {
 4|    type Book struct {
 5|       pages int
 6|    }
 7|    book1 := &Book{100} // book1是一个指针
 8|    book2 := new(Book)  // book2是另外一个指针
 9|    // 像使用结构值一样来使用结构体值的指针。
10|    book2.pages = book1.pages
11|    // 上一行等价于下一行。换句话说，上一行
12|    // 两个选择器中的指针属主将被自动解引用。
13|    (*book2).pages = (*book1).pages
14| }
```

关于结构体值的比较

如果一个结构体类型是可比较的，则它肯定不包含不可比较类型（第14章）的字段（这里不忽略名为空标识符_的字段）。

和结构体值的赋值规则类似，如果两个不同类型的结构体值均为可比较的，则它们仅在它们的底层类型相同（要考虑字段标签）并且其中至少有一个结构体值的类型为无名类型时（换句话说，只有它们可以被隐式转换为对方的类型的时候，见下）才可以互相比较。

如果两个结构体值可以相互比较，则它们的比较结果等同于逐个比较它们的相应字段（按照字段在代码中的声明顺序）。 两个结构体值只有在它们的相应字段都相等的情况下才相等；当一对字段被发现不相等的或者在比较中产生恐慌（第23章）的时候，对结构体的比较将提前结束。 在比较中，名为空标识符_的字段将被忽略掉。

关于结构体值的类型转换

两个类型分别为S1和S2的结构体值只有在S1和S2的底层类型相同（忽略掉字段标签）的情况下才能相互转换为对方的类型。 特别地，如果S1和S2的底层类型相同（要考虑字段标签）并且只要它们其中有一个为无名类型，则此转换可以是隐式的。

比如，对于下面的代码片段中所示的五个结构体类型： S0 、 S1 、 S2 、 S3和S4：

- 类型S0的值不能被转换为其它四个类型中的任意一个，原因是它与另外四个类型的对应字段名不同（因此底层类型不同）。
- 类型S1 、 S2 、 S3和S4的任意两个值可以转换为对方的类型。

特别地，

- S2表示的类型的值可以被隐式转化为类型S3，反之亦然。
- S2表示的类型的值可以被隐式转换为类型S4，反之亦然。

但是，

- S2表示的类型的值必须被显式转换为类型S1，反之亦然。
- 类型S3的值必须被显式转换为类型S4，反之亦然。

```
 1| package main
 2|
 3| type S0 struct {
 4|    y int "foo"
 5|    x bool
 6| }
 7|
 8| type S1 = struct { // S1是一个无名类型
 9|    x int "foo"
10|    y bool
11| }
12|
13| type S2 = struct { // S2也是一个无名类型
14|    x int "bar"
15|    y bool
16| }
17|
18| type S3 S2 // S3是一个定义类型（因而具名）。
19| type S4 S3 // S4是一个定义类型（因而具名）。
20| // 如果不考虑字段标签，S3（S4）和S1的底层类型一样。
21| // 如果考虑字段标签，S3（S4）和S1的底层类型不一样。
```

```
22|
23| var v0, v1, v2, v3, v4 = S0{}, S1{}, S2{}, S3{}, S4{}
24| func f() {
25|    v1 = S1(v2); v2 = S2(v1)
26|    v1 = S1(v3); v3 = S3(v1)
27|    v1 = S1(v4); v4 = S4(v1)
28|    v2 = v3; v3 = v2 // 这两个转换可以是隐式的
29|    v2 = v4; v4 = v2 // 这两个转换也可以是隐式的
30|    v3 = S3(v4); v4 = S4(v3)
31| }
```

事实上，两个结构体值只有在它们可以相互隐式转换为对方的类型的时候才能相互赋值和比较。

匿名结构体类型可以使用在结构体字段声明中

匿名结构体类型允许出现在结构体字段声明中。匿名结构体类型也允许出现在组合字面量中。

一个例子：

```
 1| var aBook = struct {
 2|    author struct { // 此字段的类型为一个匿名结构体类型
 3|       firstName, lastName string
 4|       gender              bool
 5|    }
 6|    title string
 7|    pages int
 8| }{
 9|    author: struct {
10|       firstName, lastName string
11|       gender              bool
12|    }{
13|       firstName: "Mark",
14|       lastName: "Twain",
15|    }, // 此组合字面量中的类型为一个匿名结构体类型
16|    title: "The Million Pound Note",
17|    pages: 96,
18| }
```

通常来说，为了代码可读性，最好少使用匿名结构体类型。

更多关于结构体类型

Go中有一些和结构体类型相关的进阶知识点。这些知识点将后面的类型内嵌（第24章）和内存布局（第44章）两篇文章中介绍。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

值部

此篇文章后续的若干文章将介绍Go中更多的类型。为了更容易和更深刻地理解那些类型，最好先阅读一下本文。

Go类型分为两大类别（category）

Go可以被看作是一门C语言血统的语言，这可以通过此前的指针（第15章）和结构体（第16章）两篇文章得以验证。 Go中的指针和结构体类型的内存结构和C语言很类似。

另一方面，Go也可以被看作是C语言的一个扩展框架。 在C中，值的内存结构都是很透明的；但在Go中，对于某些类型的值，其内存结构却不是很透明。 在C中，每个值在内存中只占据一个内存块（第43章）（一段连续内存）；但是，一些Go类型的值可能占据多个内存块。

以后，我们称一个Go值分布在不同内存块上的部分为此值的各个值部（value part）。 一个分布在多个内存块上的值含有一个直接值部和若干被此直接值部引用着（第15章）的间接值部。

上面的段落描述了两个类别的Go类型。下表将列出这两个类别（category）中的类型（type）种类（kind）：

每个值在内存中只分布在一个内存块上的类型	每个值在内存中会分布在多个内存块上的类型
单直接值部	直接值部 → 底层间接值部
布尔类型 各种数值类型 指针类型 非类型安全指针类型 结构体类型 数组类型	切片类型 映射类型 通道类型 函数类型 接口类型 字符串类型

表中列出的很多类型将在后续文章中逐一详细讲解。本文的目的就是为了给后续的讲解做一个铺垫。

注意：

- 接口类型和字符串类型值是否包含间接部分取决于具体编译器实现。 如果不使用今后将介绍的非类型安全途径，我们无法从这两类类型的值的外在表现来

判定它们的值是否含有间接部分。 在《Go语言101》中，我们认为这两类类型的值是可能包含间接值部的。
- 同样地，函数类型的值是否包含间接部分几乎也是不可能验证的。 在《Go语言101》中，我们认为函数值是可能包含间接值部的。

通过封装了很多具体的实现细节，第二个类别中的类型给Go编程带来了很大的便利。 不同的编译器实现会采用不同的内部结构来实现这些类型，但是这些类型的值的外在表现必须满足Go白皮书中的要求。 此分类中的类型对于编程来说并非是很基础的类型。 我们可以使用第一个分类中的类型来实现此分类中的类型。 但是，通过将一些常用或者很独特的功能封装到此第二个分类中的类型里，使用Go编程的效率将得到大大提升，体验将得到大大增强。

另一方面，这些封装同时也隐藏了这些类型的值的内部结构，使得Go程序员不能对这些类型有一个更全局更深刻的认识。有时候这会对更好地理解Go带来了一些障碍。

为了帮助Go程序员更好的理解第二个分类中的类型和它们的值，本文余下的内容将对这些类型的内在实现做一个简单介绍。 这些实现的细节将不会在本文中谈及。本文的介绍主要基于（但并不完全符合）官方标准编译器的实现。

Go中的两种指针类型

在继续下面的内容之前，我们先了解一下Go中的两种指针类型并明确一下“引用”这个词的含义。

我们已经在上上篇文章（第15章）中了解了Go中的指针。 那篇文章中所介绍的指针属于类型安全的指针。事实上，Go还支持另一种称为非类型安全的指针类型（第25章）。 非类型安全的指针类型提供在`unsafe`标准库包中。 非类型安全指针类型通常使用`unsafe.Pointer`来表示。 `unsafe.Pointer`类似于C语言中的`void*`。

在《Go语言101》中的大多数文章中，如果没有特别说明，当一个指针类型被谈及，它表示一个类型安全指针。 但是在本文的余下内容中，当一个指针被谈及，它可能表示一个类型安全指针，也可能表示一个非类型安全指针。

一个指针值存储着另一个值的地址，除非此指针值是一个`nil`空指针。 我们可以说此指针引用着（第15章）另外一个值，或者说另外一个值正被此指针所引用。 一个值可能被间接引用，比如

- 如果一个结构体值`a`含有一个指针字段`b`并且这个指针字段`b`引用着另外一个值`c`，那么我们可以说结构体值`a`也引用着值`c`。
- 如果一个值`x`（直接或者间接地）引用着另一个值`y`，并且值`y`（直接或者间接地）引用着第三个值`z`，则我们可以说值`x`间接地引用着值`z`。

以后，我们将一个含有（直接或者间接）指针字段的结构体类型称为一个**指针包裹类型**，将一个含有（直接或者间接）指针的类型称为**指针持有者类型**。 指针类型和指针包裹类型都属于指针持有者类型。元素类型为指针持有者类型的数组类型也是指针持有者类型（数组将在下一篇文章中介绍）。

第二个分类中的类型的（可能的）内部实现结构定义

为了更好地理解第二个分类中的类型的值的运行时刻行为，我们可以认为这些类型在内部是使用第一个分类中的类型来定义的（如下所示）。 如果你以前并没有很多使用过Go中各种类型的经验，目前你不必深刻地理解这些定义。 对这些定义拥有一个粗糙的印象足够对理解后续文章中将要讲解的类型有所帮助。 你可以在今后有了更多的Go编程经验之后再重读一下本文。

映射、通道和函数类型的内部定义

映射、通道和函数类型的内部定义很相似：

```
1| // 映射类型
2| type _map *hashtableImpl // 目前，官方标准编译器是使用
3|                          // 哈希表来实现映射的。
4|
5| // 通道类型
6| type _channel *channelImpl
7|
8| // 函数类型
9| type _function *functionImpl
```

从这些定义，我们可以看出来，这三个种类的类型的内部结构其实是一个指针类型。 或者说，这些类型的值的直接部分在内部是一个指针。 这些类型的每个值的直接部分引用着它的具体实现的底层间接部分。

切片类型的内部定义

切片类型的内部定义：

```
1| type _slice struct {
2|     elements unsafe.Pointer // 引用着底层的元素
3|     len      int            // 当前的元素个数
4|     cap      int            // 切片的容量
5| }
```

从这个定义可以看出来，一个切片类型在内部可以看作是一个指针包裹类型。 每个非零切片值包含着一个底层间接部分用来存储此切片的元素。 一个切片值的底层元素序列（间接部分）被此切片值的`elements`字段所引用。

字符串类型的内部结构

```
1| type _string struct {
2|     elements *byte // 引用着底层的byte元素
3|     len      int   // 字符串的长度
4| }
```

从此定义可以看出，每个字符串类型在内部也可以看作是一个指针包裹类型。 每个非零字符串值含有一个指针字段 `elements`。 这个指针字段引用着此字符串值的底层字节元素序列。

接口类型的内部定义

我们可以认为接口类型在内部是如下定义的：

```
1| type _interface struct {
2|     dynamicType  *_type         // 引用着接口值的动态类型
3|     dynamicValue unsafe.Pointer // 引用着接口值的动态值
4| }
```

从这个定义来看，接口类型也可以看作是一个指针包裹类型。一个接口类型含有两个指针字段。 每个非零接口值的（两个）间接部分分别存储着此接口值的动态类型和动态值。 这两个间接部分被此接口值的直接字段`dynamicType`和`dynamicValue`所引用。

事实上，上面这个内部定义只用于表示空接口类型的值。空接口类型没有指定任何方法。 后面的接口（第23章）一文详细解释了接口类型和值。 非空接口类型的内部定义如下：

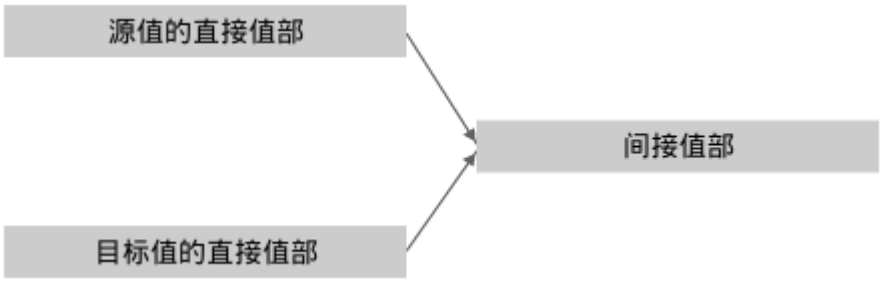
```
1| type _interface struct {
2|     dynamicTypeInfo *struct {
3|         dynamicType *_type       // 引用着接口值的动态类型
4|         methods     []*_function // 引用着动态类型的对应方法列表
5|     }
6|     dynamicValue unsafe.Pointer // 引用着动态值
7| }
```

一个非空接口类型的值的dynamicTypeInfo字段的methods字段引用着一个方法列表。 此列表中的每一项为此接口值的动态类型上定义的一个方法，此方法对应着此接口类型所指定的一个的同描述的方法。

在赋值中，底层间接值部将不会被复制

现在我们了解了第二个分类中的类型的内部结构是一个指针持有（指针或者指针包裹）类型。 这对于我们理解Go中的值复制行为有很大帮助。

在Go中，每个赋值操作（包括函数调用传参等）都是一个值的浅复制过程（假设源值和目标值的类型相同）。 换句话说，在一个赋值操作中，只有源值的直接部分被复制给了目标值。 如果源值含有间接部分，则在此赋值操作完成之后，目标值和源值的直接部分将引用着相同的间接部分。 换句话说，两个值将共享底层的间接值部，如下图所示：



事实上，对于字符串值和接口值的赋值，上述描述在理论上并非百分百正确。 官方FAQ（https://golang.google.cn/doc/faq#pass_by_value）明确说明了在一个接口值的赋值中，接口的底层动态值将被复制到目标值。 但是，因为一个接口值的动态值是只读的，所以在接口值的赋值中，官方标准编译器并没有复制底层的动态值。这可以被视为是一个编译器优化。 对于字符串值的赋值，道理是一样的。所以对于官方标准编译器来说，上一段的描述是100%正确的。

因为一个间接值部可能并不专属于任何一个值，所以在使用unsafe.Sizeof函数计算一个值的尺寸的时候，此值的间接部分所占内存空间未被计算在内。

关于术语“引用类型”和“引用值”

“引用”这个术语在Go社区中使用得有些混乱。很多Go程序员在Go编程中可能由此产生了一些困惑。 一些文档或者网络文章，包括一些官方文档（https://golang.google.cn/doc/faq#references），把“引用”（reference）看作是“值”（value）的一个对立面。 《Go语言101》强烈不推荐这种定义。在这一点上，本人不想争论什么。这里仅仅列出一些肯定错误地使用了“引用”这个术语的例子：

- 在Go中，只有切片、映射、通道和函数类型属于***引用类型***。 （如果我们确实需要***引用类型***这个术语，那么我们不应把其它指针持有者类型排除在引用类型之外。）
- 一些函数调用的参数是通过引用来传递的。 （对不起，在Go中，所有的函数调用的参数都是通过值复制**直接值部**的方式来传递的。）

我并不是想说***引用类型***这个术语在Go中是完全没有价值的， 我只是想表达这个术语是完全没有必要的，并且它常常在Go的使用中导致一些困惑。我推荐使用指针持有者类型来代替这个术语。 另外，我个人的观点是最好将***引用***这个词限定到只表示值之间的关系，把它当作一个动词或者名词来使用，永远不要把它当作一个形容词来使用。 这样将在使用Go的过程中避免很多困惑。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

数组、切片和映射

在严格意义上，Go中有三种一等公民容器类型：数组、切片和映射。 有时候，我们可以认为字符串类型和通道类型也属于容器类型。 但是，此篇文章只谈及数组、切片和映射类型。

Go中有很多和容器类型相关的细节，本文将逐一列出这些细节。

容器类型和容器值概述

每个容器（值）用来表示和存储一个元素（element）序列或集合。一个容器中的所有元素的类型是相同的。此相同的类型称为此容器的类型的元素类型（或简称此容器的元素类型）。

存储在一个容器中的每个元素值都关联着一个键值（key）。每个元素可以通过它的键值而被访问到。 一个映射类型的键值类型必须为一个可比较类型（第14章）。数组和切片类型的键值类型均为内置类型`int`。 一个数组或切片的一个元素对应的键值总是一个非负整数下标，此非负整数表示该元素在该数组或切片所有元素中的顺序位置。此非负整数下标亦常称为一个元素索引（index）。

每个容器值有一个长度属性，用来表明此容器中当前存储了多少个元素。 一个数组或切片中的每个元素所关联的非负整数索引键值的合法取值范围为左闭右开区间`[0, 此数组或切片的长度)`。 一个映射值类型的容器值中的元素关联的键值可以是任何此映射类型的键值类型的任何值。

这三种容器类型的值在使用上有很多的差别。这些差别多源于它们的内存结构的差异。 通过上一篇文章值部（第17章），我们得知每个数组值仅由一个直接部分组成，而一个切片或者映射值是由一个直接部分和一个可能的被此直接部分引用着的间接部分组成。

一个数组或者切片的所有元素紧挨着存放在一块连续的内存中。一个数组中的所有元素均存放在此数组值的直接部分，一个切片中的所有元素均存放在此切片值的间接部分。 在官方标准编译器和运行时中，映射是使用哈希表算法来实现的。所以一个映射中的所有元素也均存放在一块连续的内存中，但是映射中的元素并不一定紧挨着存放。 另外一种常用的映射实现算法是二叉树算法。无论使用何种算法，一个映射中的所有元素的键值也存放在此映射值（的间接部分）中。

我们可以通过一个元素的键值来访问此元素。 对于这三种容器，元素访问的时间复杂度均为$O(1)$。 但是一般来说，映射元素访问消耗的时长要数倍于数组和切片元素访问消耗的时长。 但是映射相对于数组和切片有两个优点：

- 映射的键值类型可以是任何可比较类型。
- 对于大多数元素为零值的情况，使用映射可以节省大量的内存。

从上一篇文章中，我们已经了解到，在任何赋值中，源值的底层间接部分不会被复制。 换句话说，当一个赋值结束后，一个含有间接部分的源值和目标值将共享底层间接部分。 这就是数组和切片/映射值会有很多行为差异（将在下面逐一介绍）的原因。

无名容器类型的字面表示形式

无名容器类型的字面表示形式如下：

- 数组类型：`[N]T`
- 切片类型：`[]T`
- 映射类型：`map[K]T`

其中，

- `T`可为任意类型。它表示一个容器类型的元素类型。某个特定容器类型的值中只能存储此容器类型的元素类型的值。
- `N`必须为一个非负整数常量。它指定了一个数组类型的长度，或者说它指定了此数组类型的任何一个值中存储了多少个元素。 一个数组类型的长度是此数组类型的一部分。比如`[5]int`和`[8]int`是两个不同的类型。
- `K`必须为一个可比较类型（第14章）。它指定了一个映射类型的键值类型。

下面列出了一些无名容器类型的字面表示：

```
 1| const Size = 32
 2|
 3| type Person struct {
 4|    name string
 5|    age  int
 6| }
 7|
 8| // 数组类型
 9| [5]string
10| [Size]int
11| [16][]byte  // 元素类型为一个切片类型：[]byte
12| [100]Person // 元素类型为一个结构体类型：Person
13|
14| // 切片类型
15| []bool
16| []int64
17| []map[int]bool // 元素类型为一个映射类型：map[int]bool
```

```
18| []*int          // 元素类型为一个指针类型：*int
19|
20| // 映射类型
21| map[string]int
22| map[int]bool
23| map[int16][6]string     // 元素类型为一个数组类型：[6]string
24| map[bool][]string       // 元素类型为一个切片类型：[]string
25| map[struct{x int}]*int8 // 元素类型为一个指针类型：*int8；
26|                         // 键值类型为一个结构体类型。
```

所有切片类型的尺寸（第14章）都是一致的，所有映射类型的尺寸也都是一致的。一个数组类型的尺寸等于它的元素类型的尺寸和它的长度的乘积。长度为零的数组的尺寸为零；元素类型尺寸为零的任意长度的数组类型的尺寸也为零。

容器字面量的表示形式

和结构体值类似，容器值的文字表示也可以用组合字面量形式（composite literal）来表示。比如对于一个容器类型`T`，它的值可以用形式`T{...}`来表示（除了切片和映射的零值外）。下面是一些容器字面量：

```
 1| // 一个含有4个布尔元素的数组值。
 2| [4]bool{false, true, true, false}
 3|
 4| // 一个含有三个字符串值的切片值。
 5| []string{"break", "continue", "fallthrough"}
 6|
 7| // 一个映射值。
 8| map[string]int{"C": 1972, "Python": 1991, "Go": 2009}
```

映射组合字面量中大括号中的每一项称为一个键值元素对（key-value pair），或者称为一个条目（entry）。

数组和切片组合字面量有一些微小的变种：

```
 1| // 下面这些切片字面量都是等价的。
 2| []string{"break", "continue", "fallthrough"}
 3| []string{0: "break", 1: "continue", 2: "fallthrough"}
 4| []string{2: "fallthrough", 1: "continue", 0: "break"}
 5| []string{2: "fallthrough", 0: "break", "continue"}
 6|
 7| // 下面这些数组字面量都是等价的。
 8| [4]bool{false, true, true, false}
 9| [4]bool{0: false, 1: true, 2: true, 3: false}
10| [4]bool{1: true, true}
```

```
11| [4]bool{2: true, 1: true}
12| [...]bool{false, true, true, false}
13| [...]bool{3: false, 1: true, true}
```

上例中最后两行中的`...`表示让编译器推断出相应数组值的类型的长度。

从上面的例子中，我们可以看出数组和切片组合字面量中的索引下标（即数组和切片的键值）是可选的。 在一个数组或者切片组合字面量中：

- 如果一个索引下标出现，它的类型不必是数组和切片类型的键值类型`int`，但它必须是一个可以表示为int值的非负常量； 如果它是一个类型确定值，则它的类型必须为一个内置整数类型。
- **在一个数组或切片组合字面量中，如果一个元素的索引下标缺失，则编译器认为它的索引下标为出现在它之前的元素的索引下标加一。**
- 如果出现的第一个元素的索引下标缺失，则它的索引下标被认为是0。

映射组合字面量中元素对应的键值不可缺失，并且它们可以为非常量。

```
1| var a uint = 1
2| var _ = map[uint]int {a : 123} // 没问题
3| var _ = []int{a: 100}          // error: 下标必须为常量
4| var _ = [5]int{a: 100}         // error: 下标必须为常量
```

一个容器组合字面量中的常量键值（包括索引下标）不可重复（第50章）。

容器类型零值的字面量表示形式

和结构体类似，一个数组类型`A`的零值可以表示为`A{}`。 比如，数组类型`[100]int`的零值可以表示为`[100]int{}`。 一个数组零值中的所有元素均为对应数组元素类型的零值。

和指针一样，所有切片和映射类型的零值均用预声明的标识符`nil`来表示。

顺便说一句，除了刚提到的三种类型，以后将介绍的函数、通道和接口类型的零值也用预声明的标识符`nil`来表示。

在运行时刻，即使一个数组变量在声明的时候未指定初始值，它的元素所占的内存空间也已经被开辟出来。 但是一个nil切片或者映射值的元素的内存空间尚未被开辟出来。

注意：`[]T{}`表示类型`[]T`的一个空切片值，它和`[]T(nil)`是不等价的。 同样，`map[K]T{}`和`map[K]T(nil)`也是不等价的。

容器字面量是不可寻址的但可以被取地址

我们已经了解到结构体（组合）字面量是不可寻址的但却是可以被取地址的（第16章）。 容器字面量也不例外。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    pm := &map[string]int{"C": 1972, "Go": 2009}
 7|    ps := &[]string{"break", "continue"}
 8|    pa := &[...]bool{false, true, true, false}
 9|    fmt.Printf("%T\n", pm) // *map[string]int
10|    fmt.Printf("%T\n", ps) // *[]string
11|    fmt.Printf("%T\n", pa) // *[4]bool
12| }
```

内嵌组合字面量可以被简化

在某些情形下，内嵌在其它组合字面量中的组合字面量可以简化为`{...}`（即类型部分被省略掉了）。 内嵌组合字面量前的取地址操作符`&`有时也可以被省略。

比如，下面的组合字面量

```
 1| // heads为一个切片值。它的类型的元素类型为*[4]byte。
 2| // 此元素类型为一个基类型为[4]byte的指针类型。
 3| // 此指针基类型为一个元素类型为byte的数组类型。
 4| var heads = []*[4]byte{
 5|    &[4]byte{'P', 'N', 'G', ' '},
 6|    &[4]byte{'G', 'I', 'F', ' '},
 7|    &[4]byte{'J', 'P', 'E', 'G'},
 8| }
```

可以被简化为

```
 1| var heads = []*[4]byte{
 2|    {'P', 'N', 'G', ' '},
 3|    {'G', 'I', 'F', ' '},
 4|    {'J', 'P', 'E', 'G'},
 5| }
```

下面这个数组组合字面量

```
type language struct {
    name string
    year int
}

var _ = [...]language{
    language{"C", 1972},
    language{"Python", 1991},
    language{"Go", 2009},
}
```

可以被简化为

```
var _ = [...]language{
    {"C", 1972},
    {"Python", 1991},
    {"Go", 2009},
}
```

下面这个映射组合字面量

```
type LangCategory struct {
    dynamic bool
    strong  bool
}

// 此映射值的类型的键值类型为一个结构体类型，
// 元素类型为另一个映射类型：map[string]int。
var _ = map[LangCategory]map[string]int{
    LangCategory{true, true}: map[string]int{
        "Python": 1991,
        "Erlang": 1986,
    },
    LangCategory{true, false}: map[string]int{
        "JavaScript": 1995,
    },
    LangCategory{false, true}: map[string]int{
        "Go":   2009,
        "Rust": 2010,
    },
    LangCategory{false, false}: map[string]int{
        "C": 1972,
    },
}
```

可以被简化为

```
 1| var _ = map[LangCategory]map[string]int{
 2|    {true, true}: {
 3|       "Python": 1991,
 4|       "Erlang": 1986,
 5|    },
 6|    {true, false}: {
 7|       "JavaScript": 1995,
 8|    },
 9|    {false, true}: {
10|       "Go":   2009,
11|       "Rust": 2010,
12|    },
13|    {false, false}: {
14|       "C": 1972,
15|    },
16| }
```

注意，在上面的几个例子中，最后一个元素后的逗号不能被省略。原因详见后面的断行规则（第28章）一文。

容器值的比较

在Go类型系统概述（第14章）一文中，我们已经了解到映射和切片类型都属于不可比较类型。 所以任意两个映射值（或切片值）是不能相互比较的。

尽管两个映射值和切片值是不能比较的，但是一个映射值或者切片值可以和预声明的`nil`标识符进行比较以检查此映射值或者切片值是否为一个零值。

大多数数组类型都是可比较类型，除了元素类型为不可比较类型的数组类型。

当比较两个数组值时，它们的对应元素将按照逐一被比较（可以认为按照下标顺序比较）。这两个数组只有在它们的对应元素都相等的情况下才相等；当一对元素被发现不相等的或者在比较中产生恐慌（第23章）的时候，对数组的比较将提前结束。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
```

```
 6|    var a [16]byte
 7|    var s []int
 8|    var m map[string]int
 9|
10|    fmt.Println(a == a)   // true
11|    fmt.Println(m == nil) // true
12|    fmt.Println(s == nil) // true
13|    fmt.Println(nil == map[string]int{}) // false
14|    fmt.Println(nil == []int{})          // false
15|
16|    // 下面这些行编译不通过。
17|    /*
18|    _ = m == m
19|    _ = s == s
20|    _ = m == map[string]int(nil)
21|    _ = s == []int(nil)
22|    var x [16][]int
23|    _ = x == x
24|    var y [16]map[int]bool
25|    _ = y == y
26|    */
27| }
```

查看容器值的长度和容量

除了上面已提到的容器长度属性（此容器中含有有多少个元素），每个容器值还有一个容量属性。 一个数组值的容量总是和它的长度相等；一个非零映射值的容量可以被认为是无限大的。切片值的容量的含义将在后续章节介绍。 一个切片值的容量总是不小于此切片值的长度。在编程中，只有切片值的容量有实际意义。

我们可以调用内置函数`len`来获取一个容器值的长度，或者调用内置函数`cap`来获取一个容器值的容量。 这两个函数都返回一个`int`类型确定结果值或者一个默认类型为`int`的类型不确定结果，具体取决于传递给它们的实参是否为常量表达式。 因为非零映射值的容量是无限大，所以`cap`并不适用于映射值。

一个数组值的长度和容量永不改变。同一个数组类型的所有值的长度和容量都总是和此数组类型的长度相等。 切片值的长度和容量可在运行时刻改变（一般只能通过被赋值的途径来修改，两者一般不可单独被修改）。 因为此原因，切片可以被认为是动态数组。 切片在使用上相比数组更为灵活，所以切片（相对数组）在编程用得更为广泛。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var a [5]int
 7|    fmt.Println(len(a), cap(a)) // 5 5
 8|    var s []int
 9|    fmt.Println(len(s), cap(s)) // 0 0
10|    s, s2 := []int{2, 3, 5}, []bool{}
11|    fmt.Println(len(s), cap(s), len(s2), cap(s2)) // 3 3 0 0
12|    var m map[int]bool
13|    fmt.Println(len(m)) // 0
14|    m, m2 := map[int]bool{1: true, 0: false}, map[int]int{}
15|    fmt.Println(len(m), len(m2)) // 2 0
16| }
```

上面这个特定的例子中的每个切片值的长度和容量都相等，但这并不是一个普遍定律。 我们将在后面的章节中展示一些长度和容量不相等的切片值。

读取和修改容器的元素

一个容器值v中存储的对应着键值k的元素用语法形式v[k]来表示。 今后我们称v[k]为一个元素索引表达式。

假设v是一个数组或者切片，在v[k]中，

- 如果k是一个常量，则它必须满足上面列出的对出现在组合字面量中的索引的要求（#value-literals）。 另外，如果v是一个数组，则k必须小于此数组的长度。
- 如果k不是一个常量，则它必须为一个整数。 另外它必须为一个非负数并且小于len(v)，否则，在运行时刻将产生一个恐慌。
- 如果v是一个零值切片，则在运行时刻将产生一个恐慌。

假设v是一个映射值，在v[k]中，k的类型必须为（或者可以隐式转换为）v的类型的元素类型。另外，

- 如果k是一个动态类型为不可比较类型的接口值，则v[k]在运行时刻将造成一个恐慌；
- 如果v[k]被用做一个赋值语句中的目标值并且v是一个零值nil映射，则v[k]在运行时刻将造成一个恐慌；

- 如果v[k]用来表示读取映射值v中键值k对应的元素，则它无论如何都不会产生一个恐慌，即使v是一个零值nil映射（假设k的估值没有造成恐慌）；
- 如果v[k]用来表示读取映射值v中键值k对应的元素，并且映射值v中并不含有对应着键值k的条目，则v[k]返回一个此映射值的类型的元素类型的零值。 一般情况下，v[k]被认为是一个单值表达式。但是在一个v[k]被用为唯一源值的赋值语句中，v[k]可以返回一个可选的第二个返回值。 此第二个返回值是一个类型不确定布尔值，用来表示是否有对应着键值k的条目存储在映射值v中。

一个展示了容器元素修改和读取的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    a := [3]int{-1, 0, 1}
 7|    s := []bool{true, false}
 8|    m := map[string]int{"abc": 123, "xyz": 789}
 9|    fmt.Println (a[2], s[1], m["abc"])    // 读取
10|    a[2], s[1], m["abc"] = 999, true, 567 // 修改
11|    fmt.Println (a[2], s[1], m["abc"])    // 读取
12|
13|    n, present := m["hello"]
14|    fmt.Println(n, present, m["hello"]) // 0 false 0
15|    n, present = m["abc"]
16|    fmt.Println(n, present, m["abc"]) // 567 true 567
17|    m = nil
18|    fmt.Println(m["abc"]) // 0
19|
20|    // 下面这两行编译不通过。
21|    /*
22|    _ = a[3]  // 下标越界
23|    _ = s[-1] // 下标越界
24|    */
25|
26|    // 下面这几行每行都会造成一个恐慌。
27|    _ = a[n]         // panic: 下标越界
28|    _ = s[n]         // panic: 下标越界
29|    m["hello"] = 555 // panic: m为一个零值映射
30| }
```

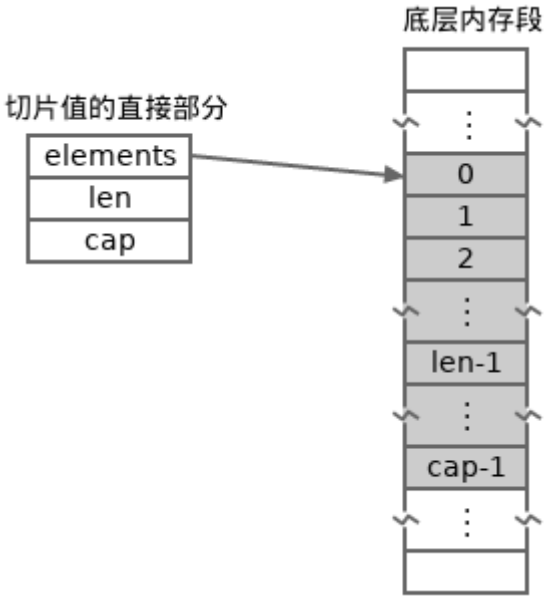
重温一下切片的内部结构

为了更好的理解和解释切片类型和切片值，我们最好对切片的内部结构有一个基本的印象。 在上一篇文章值部（第17章）中，我们已经了解到官方标准编译器对切片类型的内部定义大致如下：

```
1| type _slice struct {
2|     elements unsafe.Pointer // 引用着底层存储在间接部分上的元素
3|     len      int            // 长度
4|     cap      int            // 容量
5| }
```

虽然其它编译器中切片类型的内部结构可能并不完全和官方标准编译器一致，但应该大体上是相似的。 下面的解释均基于官方标准编译器对切片类型的内部定义。

上面展示的切片的内部定义为切片的直接部分的定义。直接部分的len字段表示一个切片当前存储了多少个元素；直接部分的cap表示一个切片的容量。 下面这张图描绘了一个切片值的内存布局。



尽管一个切片值的底层元素部分可能位于一个比较大的内存片段上，但是此切片值只能感知到此内存片段上的一个子片段。 比如，上图中的切片值只能感知到灰色的子片段。

在上图中，从下标len（包含）到下标cap（不包含）对应的元素并不属于图中所示的切片值。 它们只是此切片之中的一些冗余元素槽位，但是它们可能是其它切片（或者数组）值中的有效元素。

下一节将要介绍如何通过调用内置append函数来向一个基础切片添加元素而得到一个新的切片。 这个新的结果切片可能和这个基础切片共享起始元素，也可能不共享，具体取决于基础切片的容量（以及长度）和添加的元素数量。

当一个切片被用做一个append函数调用中的基础切片，

- 如果添加的元素数量大于此（基础）切片的冗余元素槽位的数量，则一个新的底层内存片段将被开辟出来并用来存放结果切片的元素。 这时，基础切片和结果切片不共享任何底层元素。
- 否则，不会有底层内存片段被开辟出来。这时，基础切片中的所有元素也同时属于结果切片。两个切片的元素都存放于同一个内存片段上。

下下一节将展示一张包含了上述两种情况的图片。

一些其它切片操作也可能会造成两个切片共享底层内存片段的情况。这些操作将在后续章节逐一介绍。

注意，一般我们不能单独修改一个切片值的某个内部字段，除非使用反射（#modify-slice-length-and-capacity）或者非类型安全指针（第25章）。 换句话说，一般我们只能通过将其它切片赋值给一个切片来同时修改这个切片的三个字段。

容器赋值

当一个映射赋值语句执行完毕之后，目标映射值和源映射值将共享底层的元素。 向其中一个映射中添加（或从中删除）元素将体现在另一个映射中。

和映射一样，当一个切片赋值给另一个切片后，它们将共享底层的元素。它们的长度和容量也相等。 但是和映射不同，如果以后其中一个切片改变了长度或者容量，此变化不会体现到另一个切片中。

当一个数组被赋值给另一个数组，所有的元素都将被从源数组复制到目标数组。赋值完成之后，这两个数组不共享任何元素。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    m0 := map[int]int{0:7, 1:8, 2:9}
 7|    m1 := m0
 8|    m1[0] = 2
 9|    fmt.Println(m0, m1) // map[0:2 1:8 2:9] map[0:2 1:8 2:9]
10|
11|    s0 := []int{7, 8, 9}
12|    s1 := s0
13|    s1[0] = 2
14|    fmt.Println(s0, s1) // [2 8 9] [2 8 9]
15|
```

```
16|    a0 := [...]int{7, 8, 9}
17|    a1 := a0
18|    a1[0] = 2
19|    fmt.Println(a0, a1) // [7 8 9] [2 8 9]
20| }
```

添加和删除容器元素

向一个映射中添加一个条目的语法和修改一个映射元素的语法是一样的。 比如，对于一个非零映射值`m`，如果当前`m`中尚未存储条目`(k, e)`，则下面的语法形式将把此条目存入`m`；否则，下面的语法形式将把键值`k`对应的元素值更新为`e`。

```
m[k] = e
```

内置函数`delete`用来从一个映射中删除一个条目。比如，下面的`delete`调用将把键值`k`对应的条目从映射`m`中删除。 如果映射`m`中未存储键值为`k`的条目，则此调用为一个空操作，它不会产生一个恐慌，即使`m`是一个`nil`零值映射。

```
delete(m, k)
```

下面的例子展示了如何向一个映射添加和从一个映射删除条目。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    m := map[string]int{"Go": 2007}
 7|    m["C"] = 1972     // 添加
 8|    m["Java"] = 1995  // 添加
 9|    fmt.Println(m)    // map[C:1972 Go:2007 Java:1995]
10|    m["Go"] = 2009    // 修改
11|    delete(m, "Java") // 删除
12|    fmt.Println(m)    // map[C:1972 Go:2009]
13| }
```

注意，在`Go 1.12`之前，映射打印结果中的条目顺序并不固定，两次打印结果可能并不相同。

一个数组中的元素个数总是恒定的，我们无法向其中添加元素，也无法从其中删除元素。但是可寻址的数组值中的元素是可以被修改的。

我们可以通过调用内置`append`函数，以一个切片为基础，来添加不定数量的元素并返回一个新的切片。 此新的结果切片包含着基础切片中所有的元素和所有被添加的

元素。 注意，基础切片并未被此append函数调用所修改。 当然，如果我们愿意（事实上在实践中常常如此），我们可以将结果切片赋值给基础切片以修改基础切片。

Go中并未提供一个内置方式来从一个切片中删除一个元素。 我们必须使用append函数和后面将要介绍的子切片语法一起来实现元素删除操作。 切片元素的删除和插入将在后面的更多切片操作（#slice-manipulations）一节中介绍。 本节仅展示如何使用append内置函数。

下面是一个如何使用append内置函数的例子。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     s0 := []int{2, 3, 5}
 7|     fmt.Println(s0, cap(s0)) // [2 3 5] 3
 8|     s1 := append(s0, 7)      // 添加一个元素
 9|     fmt.Println(s1, cap(s1)) // [2 3 5 7] 6
10|     s2 := append(s1, 11, 13) // 添加两个元素
11|     fmt.Println(s2, cap(s2)) // [2 3 5 7 11 13] 6
12|     s3 := append(s0)         // <=> s3 := s0
13|     fmt.Println(s3, cap(s3)) // [2 3 5] 3
14|     s4 := append(s0, s0...)  // 以s0为基础添加s0中所有的元素
15|     fmt.Println(s4, cap(s4)) // [2 3 5 2 3 5] 6
16|
17|     s0[0], s1[0] = 99, 789
18|     fmt.Println(s2[0], s3[0], s4[0]) // 789 99 2
19| }
```

注意，内置append函数是一个变长参数函数（第20章）（下下篇文章中介绍）。它有两个参数，其中第二个参数（形参）为一个变长参数（第20章）。

变长参数函数将在下下篇文章中解释。目前，我们只需知道变长参数函数调用中的实参有两种传递方式。 在上面的例子中，第*8*行、第*10*行和第*12*行使用了同一种方式，第*14*行使用了另外一种方式。 在第一种方式中，零个或多个实参元素值可以传递给append函数的第二个形参。 在第二种方式中，一个（和第一个实参同元素类型的）实参切片传递给了第二个形参，此切片实参必须跟随三个点...。 关于变长参数函数调用，详见下下篇文章（第20章）。

在上例中，第*14*行等价于

```
    s4 := append(s0, s0[0], s0[1], s0[2])
```

第*8*行等价于

```
   s1 := append(s0, []int{7}...)
```

第*10*行等价于

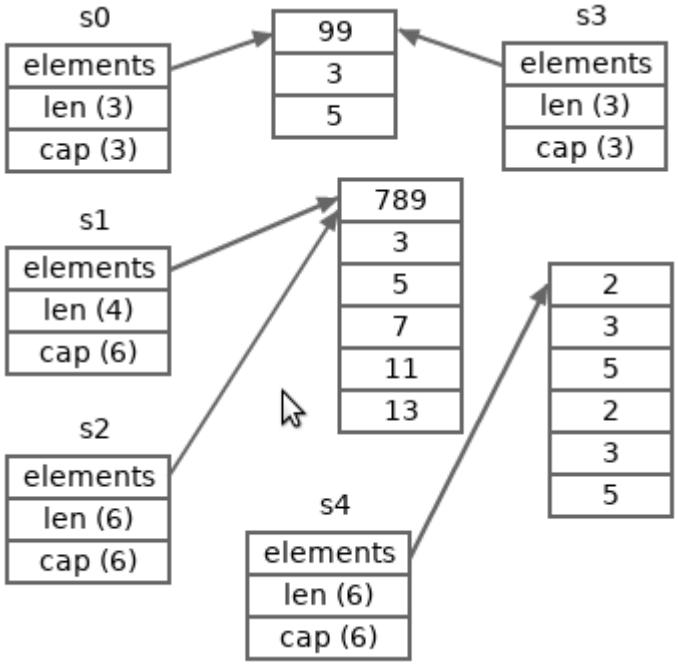
```
   s2 := append(s1, []int{11, 13}...)
```

对于三个点方式，append函数并不要求第二个实参的类型和第一个实参一致，但是它们的元素类型必须一致。 换句话说，它们的底层类型（第14章）必须一致。

在上面的程序中，

- 第*8*行的append函数调用将为结果切片s1开辟一段新的内存。 原因是切片s0中没有足够的冗余元素槽位来容纳新添加的元素。 第*14*行的append函数调用也是同样的情况。
- 第*10*行的append函数调用不会为结果切片s2开辟新的内存片段。 原因是切片s1中的冗余元素槽位足够容纳新添加的元素。

所以，上面的程序中在退出之前，切片s1和s2共享一些元素，切片s0和s3共享所有的元素。 下面这张图描绘了在上面的程序结束之前各个切片的状态。



请注意，当一个append函数调用需要为结果切片开辟内存时，结果切片的容量取决于具体编译器实现。 在这种情况下，对于官方标准编译器，如果基础切片的容量较小，则结果切片的容量至少为基础切片的两倍。 这样做的目的是使结果切片有足够多的冗余元素槽位，以防止此结果切片被用做后续其它append函数调用的基础切片时再次开辟内存。

上面提到了，在实际编程中，我们常常将append函数调用的结果赋值给基础切片。比如：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var s = append([]string(nil), "array", "slice")
 7|    fmt.Println(s)      // [array slice]
 8|    fmt.Println(cap(s)) // 2
 9|    s = append(s, "map")
10|    fmt.Println(s)      // [array slice map]
11|    fmt.Println(cap(s)) // 4
12|    s = append(s, "channel")
13|    fmt.Println(s)      // [array slice map channel]
14|    fmt.Println(cap(s)) // 4
15| }
```

截至目前（Go 1.23），append函数调用的第一个实参不能为类型不确定的nil。

使用内置make函数来创建切片和映射

除了使用组合字面量来创建映射和切片，我们还可以使用内置make函数来创建映射和切片。 数组不能使用内置make函数来创建。

顺便说一句，内置make函数也可以用来创建以后将要介绍的通道（第21章）值。

假设M是一个映射类型并且n是一个整数，我们可以用下面的两种函数调用来各自生成一个类型为M的映射值。

```
 1| make(M, n)
 2| make(M)
```

第一个函数调用形式创建了一个可以容纳至少n个条目而无需再次开辟内存的空映射值。 第二个函数调用形式创建了一个可以容纳一个小数目的条目而无需再次开辟内存的空映射值。此小数目的值取决于具体编译器实现。

注意：第二个参数n可以为负或者零，这时对应的调用将被视为上述第二种调用形式。

假设S是一个切片类型， length和capacity是两个非负整数，并且length小于等于capacity，我们可以用下面的两种函数调用来各自生成一个类型为S的切片

值。length和capacity的类型必须均为整数类型（两者可以不一致）。

```
1| make(S, length, capacity)
2| make(S, length) // <=> make(S, length, length)
```

第一个函数调用创建了一个长度为length并且容量为capacity的切片。 第二个函数调用创建了一个长度为length并且容量也为length的切片。

使用make函数创建的切片中的所有元素值均被初始化为（结果切片的元素类型的）零值。

下面是一个展示了如何使用make函数来创建映射和切片的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     // 创建映射。
 7|     fmt.Println(make(map[string]int)) // map[]
 8|     m := make(map[string]int, 3)
 9|     fmt.Println(m, len(m)) // map[] 0
10|     m["C"] = 1972
11|     m["Go"] = 2009
12|     fmt.Println(m, len(m)) // map[C:1972 Go:2009] 2
13|
14|     // 创建切片。
15|     s := make([]int, 3, 5)
16|     fmt.Println(s, len(s), cap(s)) // [0 0 0] 3 5
17|     s = make([]int, 2)
18|     fmt.Println(s, len(s), cap(s)) // [0 0] 2 2
19| }
```

使用内置new函数来创建容器值

在前面的指针（第15章）一文中，我们已经了解到内置new函数可以用来为一个任何类型的值开辟内存并返回一个存储有此值的地址的指针。 用new函数开辟出来的值均为零值。因为这个原因，new函数对于创建映射和切片值来说没有任何价值。

使用new函数来用来创建数组值并非是完全没有意义的，但是在实践中很少这么做，因为使用组合字面量来创建数组值更为方便。

一个使用new函数创建容器值的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     m := *new(map[string]int)   // <=> var m map[string]int
 7|     fmt.Println(m == nil)       // true
 8|     s := *new([]int)            // <=> var s []int
 9|     fmt.Println(s == nil)       // true
10|     a := *new([5]bool)          // <=> var a [5]bool
11|     fmt.Println(a == [5]bool{}) // true
12| }
```

容器元素的可寻址性

一些关于容器元素的可寻址性的事实：

- 如果一个数组是可寻址的，则它的元素也是可寻址的；反之亦然，即如果一个数组是不可寻址的，则它的元素也是不可寻址的。 原因很简单，因为一个数组只含有一个（直接）值部（第17章），并且它的所有元素和此直接值部均承载在同一个内存块（第43章）上。
- 一个切片值的任何元素都是可寻址的，即使此切片本身是不可寻址的。 这是因为一个切片的底层元素总是存储在一个被开辟出来的内存片段（间接值部）上。
- 任何映射元素都是不可寻址的。原因详见此条问答（第51章）。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     a := [5]int{2, 3, 5, 7}
 7|     s := make([]bool, 2)
 8|     pa2, ps1 := &a[2], &s[1]
 9|     fmt.Println(*pa2, *ps1) // 5 false
10|     a[2], s[1] = 99, true
11|     fmt.Println(*pa2, *ps1) // 99 true
12|     ps0 := &[]string{"Go", "C"}[0]
13|     fmt.Println(*ps0) // Go
14|
15|     m := map[int]bool{1: true}
16|     _ = m
```

```
17|    // 下面这几行编译不通过。
18|    /*
19|    _ = &[3]int{2, 3, 5}[0]
20|    _ = &map[int]bool{1: true}[1]
21|    _ = &m[1]
22|    */
23| }
```

一般来说，一个不可寻址的值的直接部分是不可修改的。但是映射元素是个例外。映射元素虽然不可寻址，但是每个映射元素可以被整个修改（但不能被部分修改）。 对于大多数做为映射元素类型的类型，在修改它们的值的时候，一般体现不出来整个修改和部分修改的差异。 但是如果一个映射的元素类型为数组或者结构体类型，这个差异是很明显的。

在上一篇文章值部（第17章）中，我们了解到每个数组或者结构体值都是仅含有一个直接部分。所以

- 如果一个映射类型的元素类型为一个结构体类型，则我们无法修改此映射类型的值中的每个结构体元素的单个字段。 我们必须整体地同时修改所有结构体字段。
- 如果一个映射类型的元素类型为一个数组类型，则我们无法修改此映射类型的值中的每个数组元素的单个元素。 我们必须整体地同时修改所有数组元素。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    type T struct{age int}
 7|    mt := map[string]T{}
 8|    mt["John"] = T{age: 29} // 整体修改是允许的
 9|    ma := map[int][5]int{}
10|    ma[1] = [5]int{1: 789} // 整体修改是允许的
11|
12|    // 这两个赋值编译不通过，因为部分修改一个映射
13|    // 元素是非法的。这看上去确实有些反直觉。
14|    /*
15|    ma[1][1] = 123      // error
16|    mt["John"].age = 30 // error
17|    */
18|
19|    // 读取映射元素的元素或者字段是没问题的。
20|    fmt.Println(ma[1][1])       // 789
```

```
21|     fmt.Println(mt["John"].age) // 29
22| }
```

为了让上例中的两行编译不通过的两行赋值语句编译通过，欲修改的映射元素必须先存放在一个临时变量中，然后修改这个临时变量，最后再用这个临时变量整体覆盖欲修改的映射元素。比如：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     type T struct{age int}
 7|     mt := map[string]T{}
 8|     mt["John"] = T{age: 29}
 9|     ma := map[int][5]int{}
10|     ma[1] = [5]int{1: 789}
11|
12|     t := mt["John"] // 临时变量
13|     t.age = 30
14|     mt["John"] = t // 整体修改
15|
16|     a := ma[1] // 临时变量
17|     a[1] = 123
18|     ma[1] = a // 整体修改
19|
20|     fmt.Println(ma[1][1], mt["John"].age) // 123 30
21| }
```

注意：刚提到的这个限制[可能会在以后被移除](https://github.com/golang/go/issues/3117)（https://github.com/golang/go/issues/3117）。

从数组或者切片派生切片（取子切片）

我们可以从一个基础切片或者一个可寻址的基础数组派生出另一个切片。此派生操作也常称为一个取子切片操作。 派生出来的切片的元素和基础切片（或者数组）的元素位于同一个内存片段上。或者说，派生出来的切片和基础切片（或者数组）将共享一些元素。

Go中有两种取子切片的语法形式（假设`baseContainer`是一个切片或者数组）：

```
1| baseContainer[low : high]       // 双下标形式
2| baseContainer[low : high : max] // 三下标形式
```

上面所示的双下标形式等价于下面的三下标形式：

```
baseContainer[low : high : cap(baseContainer)]
```

所以双下标形式是三下标形式的特例。在实践中，双下标形式使用得相对更为广泛。

（注意：三下标形式是从Go 1.2开始支持的。）

上面所示的取子切片表达式的语法形式中的下标必须满足下列关系，否则代码要么编译不通过，要么在运行时刻将造成恐慌。

```
// 双下标形式
0 <= low <= high <= cap(baseContainer)

// 三下标形式
0 <= low <= high <= max <= cap(baseContainer)
```

不满足上述关系的取子切片表达式要么编译不通过，要么在运行时刻将导致一个恐慌。

注意：

- 只要上述关系均满足，下标`low`和`high`都可以大于`len(baseContainer)`。但是它们一定不能大于`cap(baseContainer)`。
- 如果`baseContainer`是一个零值nil切片，只要上面所示的子切片表达式中下标的值均为`0`，则这两个子切片表达式不会造成恐慌。 在这种情况下，结果切片也是一个nil切片。

子切片表达式的结果切片的长度为`high - low`、容量为`max - low`。 派生出来的结果切片的长度可能大于基础切片的长度，但结果切片的容量绝不可能大于基础切片的容量。

在实践中，我们常常在子切片表达式中省略若干下标，以使代码看上去更加简洁。省略规则如下：

- 如果下标`low`为零，则它可被省略。此条规则同时适用于双下标形式和三下标形式。
- 如果下标`high`等于`len(baseContainer)`，则它可被省略。此条规则同时只适用于双下标形式。
- 三下标形式中的下标`max`在任何情况下都不可被省略。

比如，下面的子切片表达式都是相互等价的：

```
baseContainer[0 : len(baseContainer)]
baseContainer[: len(baseContainer)]
baseContainer[0 :]
baseContainer[:]
baseContainer[0 : len(baseContainer) : cap(baseContainer)]
baseContainer[: len(baseContainer) : cap(baseContainer)]
```

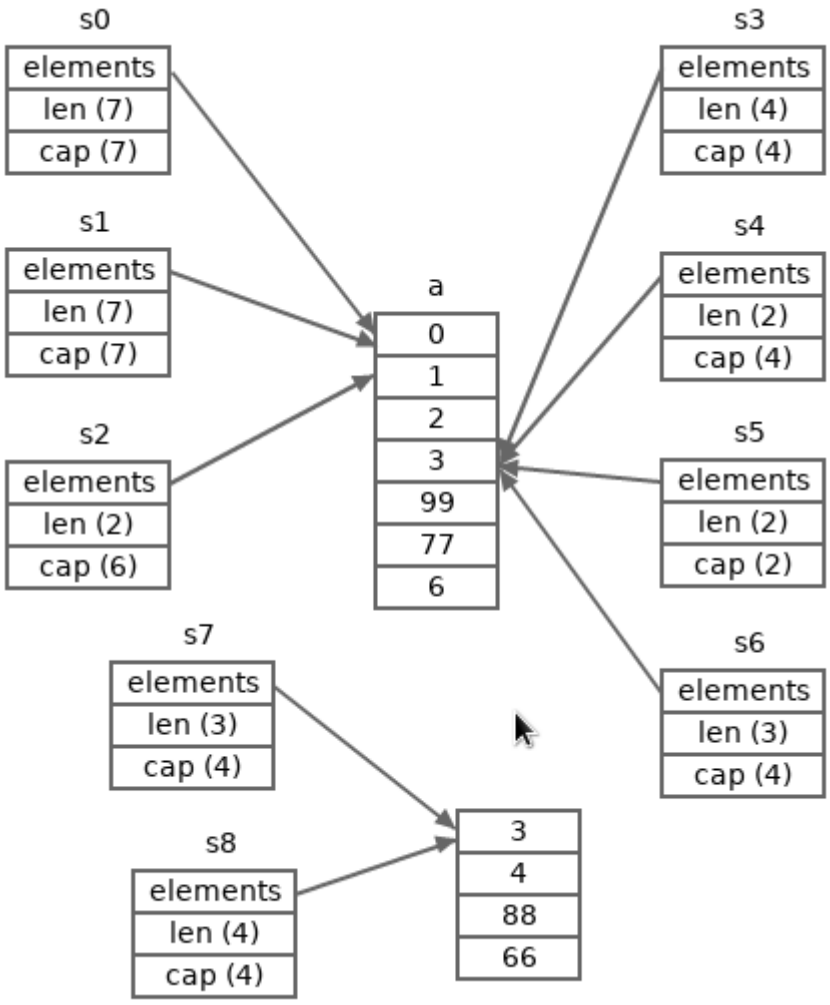
一个使用了子切片语法的例子：

```
package main

import "fmt"

func main() {
    a := [...]int{0, 1, 2, 3, 4, 5, 6}
    s0 := a[:]     // <=> s0 := a[0:7:7]
    s1 := s0[:]    // <=> s1 := s0
    s2 := s1[1:3]  // <=> s2 := a[1:3]
    s3 := s1[3:]   // <=> s3 := s1[3:7]
    s4 := s0[3:5]  // <=> s4 := s0[3:5:7]
    s5 := s4[:2:2] // <=> s5 := s0[3:5:5]
    s6 := append(s4, 77)
    s7 := append(s5, 88)
    s8 := append(s7, 66)
    s3[1] = 99
    fmt.Println(len(s2), cap(s2), s2) // 2 6 [1 2]
    fmt.Println(len(s3), cap(s3), s3) // 4 4 [3 99 77 6]
    fmt.Println(len(s4), cap(s4), s4) // 2 4 [3 99]
    fmt.Println(len(s5), cap(s5), s5) // 2 2 [3 99]
    fmt.Println(len(s6), cap(s6), s6) // 3 4 [3 99 77]
    fmt.Println(len(s7), cap(s7), s7) // 3 4 [3 4 88]
    fmt.Println(len(s8), cap(s8), s8) // 4 4 [3 4 88 66]
}
```

下面这张图描绘了上面的程序在退出之前各个数组和切片的状态。



从这张图片可以看出，切片`s7`和`s8`共享存储它们的元素的底层内存片段，其它切片和数组`a`共享同一个存储元素的内存片段。

请注意，子切片操作有可能会造成暂时性的内存泄露。 比如，下面在这个函数中开辟的内存块中的前50个元素槽位在它的调用返回之后将不再可见。 这50个元素槽位所占内存浪费了，这属于暂时性的内存泄露。 当这个函数中开辟的内存块今后不再被任何切片所引用，此内存块将被回收，这时内存才不再继续泄漏。

```
1| func f() []int {
2|     s := make([]int, 10, 100)
3|     return s[50:60]
4| }
```

请注意，在上面这个函数中，子切片表达式中的起始下标（`50`）比`s`的长度（`10`）要大，这是允许的。

切片转化为数组指针

从Go 1.17开始，一个切片可以被转化为一个相同元素类型的数组的指针类型。 但是如果数组的长度大于被转化切片的长度，则将导致恐慌产生。 转换结果和被转化

切片将共享底层元素。 一个例子：

```
 1| package main
 2|
 3| type S []int
 4| type A [2]int
 5| type P *A
 6|
 7| func main() {
 8|    var x []int
 9|    var y = make([]int, 0)
10|    var x0 = (*[0]int)(x) // okay, x0 == nil
11|    var y0 = (*[0]int)(y) // okay, y0 != nil
12|    _, _ = x0, y0
13|
14|    var z = make([]int, 3, 5)
15|    var _ = (*[3]int)(z) // okay
16|    var _ = (*[2]int)(z) // okay
17|    var _ = (*A)(z)      // okay
18|    var _ = P(z)         // okay
19|
20|    var w = S(z)
21|    var _ = (*[3]int)(w) // okay
22|    var _ = (*[2]int)(w) // okay
23|    var _ = (*A)(w)      // okay
24|    var _ = P(w)         // okay
25|
26|    var _ = (*[4]int)(z) // 会产生恐慌
27| }
```

切片转化为数组

从Go 1.20开始，一个切片可以被转化为一个相同元素类型的数组。 但是如果数组的长度大于被转化切片的长度，则将导致恐慌产生。 转换过程中将复制所需的元素，因此结果数组和被转化切片不共享底层元素。 一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var s = []int{0, 1, 2, 3}
 7|    var a = [3]int(s[1:])
 8|    s[2] = 9
```

```
 9|    fmt.Println(s) // [0 1 9 3]
10|    fmt.Println(a) // [1 2 3]
11|
12|    _ = [3]int(s[:2]) // panic
13| }
```

使用内置copy函数来复制切片元素

我们可以使用内置copy函数来将一个切片中的元素复制到另一个切片。 这两个切片的类型可以不同，但是它们的元素类型必须相同。 换句话说，这两个切片的类型的底层类型必须相同。 copy函数的第一个参数为目标切片，第二个参数为源切片。 传递给一个copy函数调用的两个实参可以共享一些底层元素。 copy函数返回复制了多少个元素，此值（int类型）为这两个切片的长度的较小值。

结合上一节介绍的子切片语法，我们可以使用copy函数来在两个数组之间或者一个数组与一个切片之间复制元素。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    type Ta []int
 7|    type Tb []int
 8|    dest := Ta{1, 2, 3}
 9|    src := Tb{5, 6, 7, 8, 9}
10|    n := copy(dest, src)
11|    fmt.Println(n, dest) // 3 [5 6 7]
12|    n = copy(dest[1:], dest)
13|    fmt.Println(n, dest) // 2 [5 5 6]
14|
15|    a := [4]int{} // 一个数组
16|    n = copy(a[:], src)
17|    fmt.Println(n, a) // 4 [5 6 7 8]
18|    n = copy(a[:], a[2:])
19|    fmt.Println(n, a) // 2 [7 8 7 8]
20| }
```

注意，做为一个特例，copy函数可以用来将一个字符串中的字节复制到一个字节切片（第19章）。

截至目前（Go 1.23），copy函数调用的两个实参均不能为类型不确定的nil。

遍历容器元素

在Go中，我们可以使用下面的语法形式来遍历一个容器中的键值和元素：

```
for key, element = range aContainer {
   // 使用key和element ...
}
```

在此语法形式中，for和range为两个关键字，key和element称为循环变量。 如果aContainer是一个切片或者数组（或者数组指针，见后），则key的类型必须为内置类型int。

上面所示的for-range语法形式中的等号=也可以是一个变量短声明符号:=。 当短声明符号被使用的时候，key和element总是两个新声明的变量，这时如果aContainer是一个切片或者数组（或者数组指针），则key的类型被推断为内置类型int。

和传统的for循环流程控制一样，每个for-range循环流程控制形成了两个代码块，其中一个是隐式的，另一个是显式的（花括号之间的部分）。 此显式的代码块内嵌在隐式的代码块之中。

和for循环流程控制一样，break和continue也可以使用在一个for-range循环流程控制中的显式代码块中。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    m := map[string]int{"C": 1972, "C++": 1983, "Go": 2009}
 7|    for lang, year := range m {
 8|       fmt.Printf("%v: %v \n", lang, year)
 9|    }
10|
11|    a := [...]int{2, 3, 5, 7, 11}
12|    for i, prime := range a {
13|       fmt.Printf("%v: %v \n", i, prime)
14|    }
15|
```

```
    s := []string{"go", "defer", "goto", "var"}
    for i, keyword := range s {
        fmt.Printf("%v: %v \n", i, keyword)
    }
}
```

for-range循环代码块有一些变种形式：

```
// 忽略键值循环变量。
for _, element = range aContainer {
    // ...
}

// 忽略元素循环变量。
for key, _ = range aContainer {
    element = aContainer[key]
    // ...
}

// 舍弃元素循环变量。此形式和上一个变种等价。
for key = range aContainer {
    element = aContainer[key]
    // ...
}

// 键值和元素循环变量均被忽略。
for _, _ = range aContainer {
    // 这个变种形式没有太大实用价值。
}

// 键值和元素循环变量均被舍弃。此形式和上一个变种等价。
for range aContainer {
    // 这个变种形式没有太大实用价值。
}
```

遍历一个nil映射或者nil切片是允许的。这样的遍历可以看作是一个空操作。

一些关于遍历映射条目的细节：

- 映射中的条目的遍历顺序是不确定的（可以认为是随机的）。或者说，同一个映射中的条目的两次遍历中，条目的顺序很可能是不一致的，即使在这两次遍历之间，此映射并未发生任何改变。
- 如果在一个映射中的条目的遍历过程中，一个还没有被遍历到的条目被删除了，则此条目保证不会被遍历出来。

- 如果在一个映射中的条目的遍历过程中，一个新的条目被添加入此映射，则此条目并不保证将在此遍历过程中被遍历出来。

如果可以确保没有其它协程操纵一个映射`m`，则下面的代码保证将清空`m`中所有条目（除了那些键值为`NaN`的条目）。

```
1| for key := range m {
2|     delete(m, key)
3| }
```

（Go 1.21引入了一个[clear内置函数](#clear)，此函数可以用来清空一个映射中所有条目，包括那些键值为`NaN`的条目。）

当然，数组和切片元素也可以用传统的`for`循环来遍历。

```
1| for i := 0; i < len(anArrayOrSlice); i++ {
2|     // ... 使用 anArrayOrSlice[i]
3| }
```

对一个`for-range`循环代码块（不论在`range`前面的是`=`还是`:=`）

```
1| for key, element = range aContainer {...}
```

有两个重要的事实存在：

1. 被遍历的容器值是`aContainer`的**一个副本**。 注意，只有`aContainer`的直接部分被复制了（第17章）。 此副本是一个匿名的值，所以它是不可被修改的。
    - 如果`aContainer`是一个数组，那么在遍历过程中对此数组元素的修改不会体现到循环变量中。 原因是此数组的副本（被真正遍历的容器）和此数组不共享任何元素。
    - 如果`aContainer`是一个切片（或者映射），那么在遍历过程中对此切片（或者映射）元素的修改将体现到循环变量中。 原因是此切片（或者映射）的副本和此切片（或者映射）共享元素（或条目）。
2. 在遍历中的每个循环步，`aContainer`副本中的一个键值元素对将被赋值（复制）给循环变量。 所以对循环变量的直接部分的修改将不会体现在`aContainer`中的对应元素中。 （因为这个原因，并且`for-range`循环是遍历映射条目的唯一途径，所以最好不要使用大尺寸的映射键值和元素类型，以避免较大的复制负担。）

下面这个例子验证了上述两个事实。

```
1| package main
2|
```

```
 3| import "fmt"
 4|
 5| func main() {
 6|    type Person struct {
 7|       name string
 8|       age  int
 9|    }
10|    persons := [2]Person {{"Alice", 28}, {"Bob", 25}}
11|    for i, p := range persons {
12|       fmt.Println(i, p)
13|       // 此修改将不会体现在这个遍历过程中，
14|       // 因为被遍历的数组是persons的一个副本。
15|       persons[1].name = "Jack"
16|
17|       // 此修改不会反映到persons数组中，因为p
18|       // 是persons数组的副本中的一个元素的副本。
19|       p.age = 31
20|    }
21|    fmt.Println("persons:", &persons)
22| }
```

输出结果：

```
0 {Alice 28}
1 {Bob 25}
persons: &[{Alice 28} {Jack 25}]
```

如果我们将上例中的数组改为一个切片，则在循环中对此切片的修改将在循环过程中体现出来。 但是对循环变量的修改仍然不会体现在此切片中。

```
 1| ...
 2|
 3|    // 数组改为切片
 4|    persons := []Person {{"Alice", 28}, {"Bob", 25}}
 5|    for i, p := range persons {
 6|       fmt.Println(i, p)
 7|       // 这次，此修改将反映在此次遍历过程中。
 8|       persons[1].name = "Jack"
 9|       // 这个修改仍然不会体现在persons切片容器中。
10|       p.age = 31
11|    }
12|    fmt.Println("persons:", &persons)
13| }
```

输出结果变成了：

```
0 {Alice 28}
1 {Jack 25}
persons: &[{Alice 28} {Jack 25}]
```

下面这个例子验证了上述第二个事实。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    m := map[int]struct{ dynamic, strong bool } {
 7|       0: {true, false},
 8|       1: {false, true},
 9|       2: {false, false},
10|    }
11|
12|    for _, v := range m {
13|       // This following line has no effects on the map m.
14|       v.dynamic, v.strong = true, true
15|    }
16|
17|    fmt.Println(m[0]) // {true false}
18|    fmt.Println(m[1]) // {false true}
19|    fmt.Println(m[2]) // {false false}
20| }
```

复制一个切片或者映射的代价很小，但是复制一个大尺寸的数组的代价比较大。 所以，一般来说，`range`关键字后跟随一个大尺寸数组不是一个好主意。 如果我们要遍历一个大尺寸数组中的元素，我们以遍历从此数组派生出来的一个切片，或者遍历一个指向此数组的指针（详见下一节）。

对于一个数组或者切片，如果它的元素类型的尺寸较大，则一般来说，用第二个循环变量来存储每个循环步中被遍历的元素不是一个好主意。 对于这样的数组或者切片，我们最好忽略或者舍弃`for-range`代码块中的第二个循环变量，或者使用传统的`for`循环来遍历元素。 比如，在下面这个例子中，函数`fa`中的循环效率比函数`fb`中的循环低得多。

```
 1| type Buffer struct {
 2|    start, end int
 3|    data       [1024]byte
 4| }
 5|
 6| func fa(buffers []Buffer) int {
 7|    numUnreads := 0
```

```
 8|    for _, buf := range buffers {
 9|       numUnreads += buf.end - buf.start
10|    }
11|    return numUnreads
12| }
13|
14| func fb(buffers []Buffer) int {
15|    numUnreads := 0
16|    for i := range buffers {
17|       numUnreads += buffers[i].end - buffers[i].start
18|    }
19|    return numUnreads
20| }
```

在Go 1.22之前，对一个如下`for-range`循环代码块（注意`range`前面是`:=`）

```
 1| for key, element := range aContainer {...}
```

所有被遍历的键值元素对将被赋值给**同一对**循环变量实例。 但是从Go 1.22版本开始，每组键值元素对将被赋值给一对**与众不同**的循环变量实例（既循环变量在每个循环步都会生成一份新的实例）。

下面这个例子展示了Go 1.21-和Go 1.22+之间的行为差异。

```
 1| // forrange1.go
 2| package main
 3|
 4| import "fmt"
 5|
 6| func main() {
 7|    for i, n := range []int{0, 1, 2} {
 8|       defer func() {
 9|          fmt.Println(i, n)
10|       }()
11|    }
12| }
```

使用不同版本的Go编译器运行之（[gotv](https://gfw.go101.org/apps-and-libs/gotv.html)（https://gfw.go101.org/apps-and-libs/gotv.html）是一个管理运行多个Go官方工具链版本的工具；未来的Go 1.22版本将来从tip版本开出来），将得到不同的输出：

```
 1| $ gotv 1.21. run forrange1.go
 2| [Run]: $HOME/.cache/gotv/tag_go1.21.8/bin/go run
forrange1.go
 3| 2 2
 4| 2 2
```

```
 5| 2 2
 6| $ gotv 1.22. run forrange1.go
 7| [Run]: $HOME/.cache/gotv/tag_go1.22.1/bin/go run
forrange1.go
 8| 2 2
 9| 1 1
10| 0 0
```

另一个例子：

```
 1| // forrange2.go
 2| package main
 3|
 4| import "fmt"
 5|
 6| func main() {
 7|    var m = map[*int]uint32{}
 8|    for i, n := range []int{1, 2, 3} {
 9|       m[&i]++
10|       m[&n]++
11|    }
12|    fmt.Println(len(m))
13| }
```

使用不同版本的Go编译器运行之，得到如下输出：

```
 1| $ gotv 1.21. run forrange2.go
 2| [Run]: $HOME/.cache/gotv/tag_go1.21.8/bin/go run
forrange2.go
 3| 2
 4| $ gotv 1.22. run forrange2.go
 5| [Run]: $HOME/.cache/gotv/tag_go1.22.1/bin/go run
forrange2.go
 6| 6
```

因此，这是一个破坏了向后兼容性的语义改变。 但是新的语义更符合人们的直觉；并且从理论上，到目前还没有发现在逻辑上正确的旧代码因为此改变而导致行为变化的情况。

把数组指针当做数组来使用

对于某些情形，我们可以把数组指针当做数组来使用。

我们可以通过在range关键字后跟随一个数组的指针来遍历此数组中的元素。 对于大尺寸的数组，这种方法比较高效，因为复制一个指针比复制一个大尺寸数组的代价低得多。 下面的例子中的两个循环是等价的，它们的效率也基本相同。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var a [100]int
 7|
 8|    for i, n := range &a { // 复制一个指针的开销很小
 9|       fmt.Println(i, n)
10|    }
11|
12|    for i, n := range a[:] { // 复制一个切片的开销很小
13|       fmt.Println(i, n)
14|    }
15| }
```

如果一个for-range循环中的第二个循环变量既没有被忽略，也没有被舍弃，并且range关键字后跟随一个nil数组指针，则此循环将造成一个恐慌。 在下面这个例子中，前两个循环都将打印出5个下标，但最后一个循环将导致一个恐慌。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var p *[5]int // nil
 7|
 8|    for i, _ := range p { // okay
 9|       fmt.Println(i)
10|    }
11|
12|    for i := range p { // okay
13|       fmt.Println(i)
14|    }
15|
16|    for i, n := range p { // panic
17|       fmt.Println(i, n)
18|    }
19| }
```

我们可以通过数组的指针来访问和修改此数组中的元素。如果此指针是一个nil指针，将导致一个恐慌。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    a := [5]int{2, 3, 5, 7, 11}
 7|    p := &a
 8|    p[0], p[1] = 17, 19
 9|    fmt.Println(a) // [17 19 5 7 11]
10|    p = nil
11|    _ = p[0] // panic
12| }
```

我们可以从一个数组的指针派生出一个切片。从一个nil数组指针派生切片将导致一个恐慌。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    pa := &[5]int{2, 3, 5, 7, 11}
 7|    s := pa[1:3]
 8|    fmt.Println(s) // [3 5]
 9|    pa = nil
10|    s = pa[0:0] // panic
11|    // 如果下一行能被执行到，则它也会产生恐慌。
12|    _ = (*[0]byte)(nil)[:]
13| }
```

内置len和cap函数调用接受数组指针做为实参。 nil数组指针实参不会导致恐慌。

```
 1| var pa *[5]int // == nil
 2| fmt.Println(len(pa), cap(pa)) // 5 5
```

memclr优化

假设t0是一个类型T的零值字面量，并且a是一个元素类型为T的数组或者切片，则官方标准编译器将把下面的单循环变量for-range代码块优化为一个内部的

memclr调用（https://github.com/golang/go/issues/5373）。 大多数情况下，此memclr调用比一个一个地重置元素要快。

```
 1| for i := range a {
 2|     a[i] = t0
 3| }
```

此优化在官方标准编译器1.5版本中被引入。

从Go官方工具链1.19开始，此优化也适用于a为一个数组指针的情形。

注意：Go 1.21引入了一个clear内置函数，用来清空一个映射中的所有条目或者重置一个切片中的所有元素。 我们应该尽量使用此内置函数而不是依赖于此memclr优化来重置切片或者数组的元素。 clear内置函数见下一节。

使用内置clear函数来清空映射条目或者重置切片元素

Go 1.21引入了一个clear内置函数。 此函数可以用来清空映射条目或者重置切片元素。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     s := []int{1, 2, 3}
 7|     clear(s)
 8|     fmt.Println(s) // [0 0 0]
 9|
10|     a := [4]int{5, 6, 7, 8}
11|     clear(a[1:3])
12|     fmt.Println(a) // [5 0 0 8]
13|
14|     m := map[float64]float64{}
15|     x := 0.0
16|     m[x] = x
17|     x /= x // x变成了NaN
18|     m[x] = x
19|     fmt.Println(len(m)) // 2
20|     for k := range m {
```

```
21|       delete(m, k)
22|    }
23|    fmt.Println(len(m)) // 1
24|    clear(m)
25|    fmt.Println(len(m)) // 0
26| }
```

从上例中，我们可以发现此clear函数甚至可以清除那些键值为NaN的映射条目.

内置函数len和cap的调用可能会在编译时刻被估值

如果传递给内置函数len或者cap的一个调用的实参是一个数组或者数组指针，则此调用将在编译时刻被估值。 此估值结果是一个类型为内置类型int的类型确定常量值。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| var a [5]int
 6| var p *[7]string
 7|
 8| // N和M都是类型为int的类型确定值。
 9| const N = len(a)
10| const M = cap(p)
11|
12| func main() {
13|    fmt.Println(N) // 5
14|    fmt.Println(M) // 7
15| }
```

单独修改一个切片的长度或者容量

上面已经提到了，一般来说，一个切片的长度和容量不能被单独修改。一个切片只有通过赋值的方式被整体修改。 但是，事实上，我们可以通过反射的途径来单独修改一个切片的长度或者容量。 反射将在后面的一篇文章（第27章）中详解。

一个例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "reflect"
 6| )
 7|
 8| func main() {
 9|     s := make([]int, 2, 6)
10|     fmt.Println(len(s), cap(s)) // 2 6
11|
12|     reflect.ValueOf(&s).Elem().SetLen(3)
13|     fmt.Println(len(s), cap(s)) // 3 6
14|
15|     reflect.ValueOf(&s).Elem().SetCap(5)
16|     fmt.Println(len(s), cap(s)) // 3 5
17| }
```

传递给函数`reflect.SetLen`调用的第二个实参值必须不大于第一个实参切片值的容量。 传递给函数`reflect.SetCap`调用的第二个实参值必须不小于第一个实参切片值的长度并且须不大于第一个实参切片值的容量。 否则，在运行时刻将产生一个恐慌。

此反射方法的效率很低，远低于一个切片的赋值。

更多切片操作

Go不支持更多的内置切片操作，比如切片克隆、元素删除和插入。 我们必须用上面提到的各种内置操作来实现这些操作。

在下面当前大节中的例子中，假设`s`是被谈到的切片、`T`是它的元素类型、`t0`是类型`T`的零值字面量。

切片克隆

对于当前的Go版本（1.23），最简单的克隆一个切片的方法为：

```
sClone := append(s[:0:0], s...)
```

我们也可以使用下面这种实现。但是和上面这个实现相比，它有一个不完美之处：如果源切片`s`是一个空切片（但是非nil），则结果切片是一个nil切片。

```
sClone := append([]T(nil), s...)
```

上面这两种append实现都有一个缺点：它们开辟的内存块常常会比需要的略大一些从而可能造成一点小小的不必要的性能损失。 我们可以使用这两种方法来避免这个缺点：

```
// 两行make+copy实现：
sClone := make([]T, len(s))
copy(sClone, s)

// 或者下面的make+append实现。
// 对于目前的Go官方工具链1.23版本来说，
// 这种实现比上面的make+copy实现略慢一点。
sClone := append(make([]T, 0, len(s)), s...)
```

上面这两种make方法都有一个缺点：如果s是一个nil切片，则使用此方法将得到一个非nil切片。 不过，在编程实践中，我们常常并不需要追求克隆的完美性。如果我们确实需要，则需要多写几行：

```
var sClone []T
if s != nil {
   sClone = make([]T, len(s))
   copy(sClone, s)
}
```

删除一段切片元素

前面已经提到了切片的元素在内存中是连续存储的，相邻元素之间是没有间隙的。所以，当切片的一个元素段被删除时，

- 如果剩余元素的次序必须保持原样，则被删除的元素段后面的每个元素都得前移。
- 如果剩余元素的次序不需要保持原样，则我们可以将尾部的一些元素移到被删除的元素的位置上。

在下面的例子中，假设from（包括）和to（不包括）是两个合法的下标，并且from不大于to。

```
 1| // 第一种方法（保持剩余元素的次序）：
 2| s = append(s[:from], s[to:]...)
 3|
 4| // 第二种方法（保持剩余元素的次序）：
 5| s = s[:from + copy(s[from:], s[to:])]
 6|
 7| // 第三种方法（不保持剩余元素的次序）：
 8| if n := to-from; len(s)-to < n {
```

```
 9|    copy(s[from:to], s[to:])
10| } else {
11|    copy(s[from:to], s[len(s)-n:])
12| }
13| s = s[:len(s)-(to-from)]
```

如果切片的元素可能引用着其它值，则我们应该重置因为删除元素而多出来的元素槽位上的元素值，以避免暂时性的内存泄露：

```
 1| // "len(s)+to-from"是删除操作之前切片s的长度。
 2| temp := s[len(s):len(s)+to-from]
 3| for i := range temp {
 4|    temp[i] = t0 // t0是类型T的零值字面量
 5| }
```

前面已经提到了，上面这个`for-range`循环将被官方标准编译器优化为一个`memclr`调用。

删除一个元素

删除一个元素是删除一个元素段的特例。在实现上可以简化一些。

在下面的例子中，假设`i`将被删除的元素的下标，并且它是一个合法的下标。

```
 1| // 第一种方法（保持剩余元素的次序）：
 2| s = append(s[:i], s[i+1:]...)
 3|
 4| // 第二种方法（保持剩余元素的次序）：
 5| s = s[:i + copy(s[i:], s[i+1:])]
 6|
 7| // 上面两种方法都需要复制len(s)-i-1个元素。
 8|
 9| // 第三种方法（不保持剩余元素的次序）：
10| s[i] = s[len(s)-1]
11| s = s[:len(s)-1]
```

如果切片的元素可能引用着其它值，则我们应该重置刚多出来的元素槽位上的元素值，以避免暂时性的内存泄露：

```
 1| s[len(s):len(s)+1][0] = t0
 2| // 或者
 3| s[:len(s)+1][len(s)] = t0
```

条件性地删除切片元素

有时，我们需要删除满足某些条件的切片元素。

```
 1| // 假设T是一个小尺寸类型。
 2| func DeleteElements(s []T, keep func(T) bool, clear bool)
[]T {
 3|    // result := make([]T, 0, len(s))
 4|    result := s[:0] // 无须开辟内存
 5|    for _, v := range s {
 6|       if keep(v) {
 7|          result = append(result, v)
 8|       }
 9|    }
10|    if clear { // 避免暂时性的内存泄露。
11|       temp := s[len(result):]
12|       for i := range temp {
13|          temp[i] = t0 // t0是类型T的零值
14|       }
15|    }
16|    return result
17| }
```

注意：如果T是一个大尺寸类型，请慎用（第34章）T做为参数类型和使用双循环变量for-range代码块遍历元素类型为T的切片。

将一个切片中的所有元素插入到另一个切片中

假设插入位置i是一个合法的下标并且切片elements中的元素将被插入到另一个切片s中。

```
 1| // 第一种方法:单行实现。
 2| s = append(s[:i], append(elements, s[i:]...)...)
 3|
 4| // 上面这种单行实现把s[i:]中的元素复制了两次，并且它可能
 5| // 最多导致两次内存开辟（最少一次）。
 6| // 下面这种繁琐的实现只把s[i:]中的元素复制了一次，并且
 7| // 它最多只会导致一次内存开辟（最少零次）。
 8| // 但是，在当前的官方标准编译器实现中（1.23版本），此
 9| // 繁琐实现中的make调用将会把部分刚开辟出来的元素清零。
10| // 这其实是没有必要的。所以此繁琐实现并非总是比上面的
11| // 单行实现效率更高。事实上，它仅在处理小切片时更高效。
12|
13| if cap(s) >= len(s) + len(elements) {
14|    s = s[:len(s)+len(elements)]
15|    copy(s[i+len(elements):], s[i:])
```

```
16|    copy(s[i:], elements)
17| } else {
18|    x := make([]T, 0, len(elements)+len(s))
19|    x = append(x, s[:i]...)
20|    x = append(x, elements...)
21|    x = append(x, s[i:]...)
22|    s = x
23| }
24|
25| // Push（插入到结尾）。
26| s = append(s, elements...)
27|
28| // Unshift（插入到开头）。
29| s = append(elements, s...)
```

插入若干独立的元素

插入若干独立的元素和插入一个切片中的所有元素类似。 我们可以使用切片组合字面量构建一个临时切片，然后使用上面的方法插入这些元素。

特殊的插入和删除：前推/后推，前弹出/后弹出

假设被推入和弹出的元素为e并且切片s拥有至少一个元素。

```
1| // 前弹出（pop front，又称shift）
2| s, e = s[1:], s[0]
3| // 后弹出（pop back）
4| s, e = s[:len(s)-1], s[len(s)-1]
5| // 前推（push front）
6| s = append([]T{e}, s...)
7| // 后推（push back）
8| s = append(s, e)
```

请注意：使用append函数来插入元素常常是比较低效的，因为插入点后的所有元素都要向后挪，并且当空余容量不足时还需要开辟一个更大的内存空间来容纳插入完成后所有的元素。 对于元素个数不多的切片来说，这些可能并不是严重的问题；但是在元素个数很多的切片上进行如上的插入操作常常是耗时的。所以如果元素个数很多，最好使用链表来实现元素插入操作。

关于上面各种切片操控的例子

在实践中，需求是各种各样的。对于某些特定的情形，上面的例子中的代码实现可能并非是最优化的，甚至是不满足要求的。 所以，请在实践中根据具体情况来实现代码。或许，这就是Go没有支持更多的内置切片操作的原因。

用映射来模拟集合（set）

Go不支持内置集合（**set**）类型。但是，集合类型可以用轻松使用映射类型来模拟。在实践中，我们常常使用映射类型`map[K]struct{}`来模拟一个元素类型为`K`的集合类型。 类型`struct{}`的尺寸为零，所以此映射类型的值中的元素不消耗内存。

上述各种容器操作内部都未同步

请注意，上述所有各种容器操作的内部实现都未进行同步。如果不使用今后将要介绍的各种并发同步技术，在没有协程修改一个容器值和它的元素的时候，多个协程并发读取此容器值和它的元素是安全的。但是并发修改同一个容器值则是不安全的。 不使用并发同步技术而并发修改同一个容器值将会造成数据竞争。请阅读以后的并发同步概述（第36章）一文以了解Go支持的各种并发同步技术。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

字符串

和很多其它编程语言一样，字符串类型是Go中的一种重要类型。本文将列举出关于字符串的各种事实。

字符串类型的内部结构定义

对于标准编译器，字符串类型的内部结构声明如下：

```
1| type _string struct {
2|     elements *byte // 引用着底层的字节
3|     len      int   // 字符串中的字节数
4| }
```

从这个声明来看，我们可以将一个字符串的内部定义看作为一个字节序列。 事实上，我们确实可以把一个字符串看作是一个元素类型为`byte`的（且元素不可修改的）切片。

注意，前面的文章已经提到过多次，`byte`是内置类型`uint8`的一个别名。

关于字符串的一些简单事实

从前面的若干文章，我们已经了解到下列关于字符串的一些事实：

- 字符串值（和布尔以及各种数值类型的值）可以被用做常量。
- Go支持两种风格的字符串字面量表示形式：双引号风格（解释型字面表示）和反引号风格（直白字面表示）。具体介绍请阅读前文（第6章）。
- 字符串类型的零值为空字符串。一个空字符串在字面上可以用`""`或者``` `` ```来表示。
- 我们可以用运算符`+`和`+=`来衔接字符串。
- 字符串类型都是可比较类型。同一个字符串类型的值可以用`==`和`!=`比较运算符来比较。 并且和整数/浮点数一样，同一个字符串类型的值也可以用`>`、`<`、`>=`和`<=`比较运算符来比较。 当比较两个字符串值的时候，它们的底层字节将逐一进行比较。如果一个字符串是另一个字符串的前缀，并且另一个字符串较长，则另一个字符串为两者中的较大者。

一个例子：

```
1| package main
2|
3| import "fmt"
```

```
 4|
 5| func main() {
 6|     const World = "world"
 7|     var hello = "hello"
 8|
 9|     // 衔接字符串。
10|     var helloWorld = hello + " " + World
11|     helloWorld += "!"
12|     fmt.Println(helloWorld) // hello world!
13|
14|     // 比较字符串。
15|     fmt.Println(hello == "hello")   // true
16|     fmt.Println(hello > helloWorld) // false
17| }
```

更多关于字符串类型和值的事实：

- 和Java语言一样，字符串值的内容（即底层字节）是不可更改的。 字符串值的长度也是不可独立被更改的。 一个可寻址的字符串只能通过将另一个字符串赋值给它来整体修改它。
- 字符串类型没有内置的方法。我们可以
  - 使用`strings`标准库（https://golang.google.cn/pkg/strings/）提供的函数来进行各种字符串操作。
  - 调用内置函数`len`来获取一个字符串值的长度（此字符串中存储的字节数）。
  - 使用容器元素索引（第18章）语法`aString[i]`来获取`aString`中的第`i`个字节。 表达式`aString[i]`是不可寻址的。换句话说，`aString[i]`不可被修改。
  - 使用子切片语法（第18章）`aString[start:end]`来获取`aString`的一个子字符串。 这里，`start`和`end`均为`aString`中存储的字节的下标。
- 对于标准编译器来说，一个字符串的赋值完成之后，此赋值中的目标值和源值将共享底层字节。 一个子切片表达式`aString[start:end]`的估值结果也将和基础字符串`aString`共享一部分底层字节。

一个例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "strings"
 6| )
 7|
 8| func main() {
```

```
 9|    var helloWorld = "hello world!"
10|
11|    var hello = helloWorld[:5] // 取子字符串
12|    // 104是英文字符h的ASCII（和Unicode）码。
13|    fmt.Println(hello[0])         // 104
14|    fmt.Printf("%T \n", hello[0]) // uint8
15|
16|    // hello[0]是不可寻址和不可修改的，所以下面
17|    // 两行编译不通过。
18|    /*
19|    hello[0] = 'H'         // error
20|    fmt.Println(&hello[0]) // error
21|    */
22|
23|    // 下一条语句将打印出：5 12 true
24|    fmt.Println(len(hello), len(helloWorld),
25|          strings.HasPrefix(helloWorld, hello))
26| }
```

注意：如果在`aString[i]`和`aString[start:end]`中，`aString`和各个下标均为常量，则编译器将在编译时刻验证这些下标的合法性，但是这样的元素访问和子切片表达式的估值结果总是非常量（这是Go语言设计之初的一个失误，但因为兼容性的原因导致难以弥补（https://github.com/golang/go/issues/28591））。比如下面这个程序将打引出`4 0`。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| const s = "Go101.org" // len(s) == 9
 6|
 7| // len(s)是一个常量表达式，但len(s[:])却不是。
 8| var a byte = 1 << len(s) / 128
 9| var b byte = 1 << len(s[:]) / 128
10|
11| func main() {
12|    fmt.Println(a, b) // 4 0
13| }
```

a和b两个变量估值不同的具体原因请阅读移位操作类型推断规则（第8章）和哪些函数调用在编译时刻被估值（第46章）。

字符串编码和Unicode码点

Unicode标准为全球各种人类语言中的每个字符制定了一个独一无二的值。 但Unicode标准中的基本单位不是字符，而是码点（code point）。大多数的码点实际上就对应着一个字符。 但也有少数一些字符是由多个码点组成的。

码点值在Go中用rune值（第6章）来表示。 内置`rune`类型为内置`int32`类型的一个别名。

在具体应用中，码点值的编码方式有很多，比如UTF-8编码和UTF-16编码等。 目前最流行编码方式为UTF-8编码。在Go中，所有的字符串常量都被视为是UTF-8编码的。 在编译时刻，非法UTF-8编码的字符串常量将导致编译失败。 在运行时刻，Go运行时无法阻止一个字符串是非法UTF-8编码的。

在UTF-8编码中，一个码点值可能由1到4个字节组成。 比如，每个英语码点值（均对应一个英语字符）均由一个字节组成，而每个中文码点值（均对应一个中文字符）均由三个字节组成。

字符串相关的类型转换

在常量和变量（第7章）一文中，我们已经了解到整数可以被显式转换为字符串类型（但是反之不行）。

这里介绍两种新的字符串相关的类型转换规则：

1. 一个字符串值可以被显式转换为一个字节切片（byte slice），反之亦然。一个字节切片类型是一个元素类型的底层类型为内置类型`byte`的切片类型。
2. 一个字符串值可以被显式转换为一个码点切片（rune slice），反之亦然。一个码点切片类型是一个元素类型的底层类型为内置类型`rune`的切片类型。

在一个从码点切片到字符串的转换中，码点切片中的每个码点值将被UTF-8编码为一到四个字节至结果字符串中。 如果一个码点值是一个不合法的Unicode码点值，则它将被视为Unicode替换字符（码点）值`0xFFFD`（Unicode replacement character）。 替换字符值`0xFFFD`将被UTF-8编码为三个字节`0xef 0xbf 0xbd`。

当一个字符串被转换为一个码点切片时，此字符串中存储的字节序列将被解读为一个一个码点的UTF-8编码序列。 非法的UTF-8编码字节序列将被转化为Unicode替换字符值`0xFFFD`。

当一个字符串被转换为一个字节切片时，结果切片中的底层字节序列是此字符串中存储的字节序列的一份深复制。 即Go运行时将为结果切片开辟一块足够大的内存来容纳被复制过来的所有字节。当此字符串的长度较长时，此转换开销是比较大的。同样，当一个字节切片被转换为一个字符串时，此字节切片中的字节序列也将被深复制到结果字符串中。 当此字节切片的长度较长时，此转换开销同样是比较大的。在这两种转换中，必须使用深复制的原因是字节切片中的字节元素是可修改的，但

是字符串中的字节是不可修改的，所以一个字节切片和一个字符串是不能共享底层字节序列的。

请注意，在字符串和字节切片之间的转换中，

- 非法的UTF-8编码字节序列将被保持原样不变。
- 标准编译器做了一些优化，从而使得这些转换在某些情形下将不用深复制。这样的情形将在下一节中介绍。

Go并不支持字节切片和码点切片之间的直接转换。我们可以用下面列出的方法来实现这样的转换：

- 利用字符串做为中间过渡。这种方法相对方便但效率较低，因为需要做两次深复制。
- 使用unicode/utf8（https://golang.google.cn/pkg/unicode/utf8/）标准库包中的函数来实现这些转换。 这种方法效率较高，但使用起来不太方便。
- 使用bytes标准库包中的Runes函数（https://golang.google.cn/pkg/bytes/#Runes）来将一个字节切片转换为码点切片。 但此包中没有将码点切片转换为字节切片的函数。

一个展示了上述各种转换的例子：

```
 1| package main
 2|
 3| import (
 4|     "bytes"
 5|     "unicode/utf8"
 6| )
 7|
 8| func Runes2Bytes(rs []rune) []byte {
 9|     n := 0
10|     for _, r := range rs {
11|         n += utf8.RuneLen(r)
12|     }
13|     n, bs := 0, make([]byte, n)
14|     for _, r := range rs {
15|         n += utf8.EncodeRune(bs[n:], r)
16|     }
17|     return bs
18| }
19|
20| func main() {
21|     s := "颜色感染是一个有趣的游戏。"
22|     bs := []byte(s) // string -> []byte
23|     s = string(bs)  // []byte -> string
24|     rs := []rune(s) // string -> []rune
```

```
25|    s = string(rs)  // []rune -> string
26|    rs = bytes.Runes(bs) // []byte -> []rune
27|    bs = Runes2Bytes(rs) // []rune -> []byte
28| }
```

字符串和字节切片之间的转换的编译器优化

上面已经提到了字符串和字节切片之间的转换将深复制它们的底层字节序列。 标准编译器做了一些优化，从而在某些情形下避免了深复制。 至少这些优化在当前（Go官方工具链1.23版本）是存在的。 这样的情形包括：

- 一个`for-range`循环中跟随`range`关键字的从字符串到字节切片的转换；
- 一个在映射元素读取索引语法中被用做键值的从字节切片到字符串的转换（注意：对修改写入索引语法无效）；
- 一个字符串比较表达式中被用做比较值的从字节切片到字符串的转换；
- 一个（至少有一个被衔接的字符串值为非空字符串常量的）字符串衔接表达式中的从字节切片到字符串的转换。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var str = "world"
 7|    // 这里，转换[]byte(str)将不需要一个深复制。
 8|    for i, b := range []byte(str) {
 9|       fmt.Println(i, ":", b)
10|    }
11|
12|    key := []byte{'k', 'e', 'y'}
13|    m := map[string]string{}
14|    // 这个string(key)转换仍然需要深复制。
15|    m[string(key)] = "value"
16|    // 这里的转换string(key)将不需要一个深复制。
17|    // 即使key是一个包级变量，此优化仍然有效。
18|    fmt.Println(m[string(key)]) // value
19| }
```

注意：在最后一行中，如果在估值`string(key)`的时候有数据竞争的情况，则这行的输出有可能并不是`value`。 但是，无论如何，此行都不会造成恐慌（即使有数据竞争的情况发生）。

另一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "testing"
 5|
 6| var s string
 7| var x = []byte{1023: 'x'}
 8| var y = []byte{1023: 'y'}
 9|
10| func fc() {
11|    // 下面的四个转换都不需要深复制。
12|    if string(x) != string(y) {
13|       s = (" " + string(x) + string(y))[1:]
14|    }
15| }
16|
17| func fd() {
18|    // 两个在比较表达式中的转换不需要深复制，
19|    // 但两个字符串衔接中的转换仍需要深复制。
20|    // 请注意此字符串衔接和fc中的衔接的差别。
21|    if string(x) != string(y) {
22|       s = string(x) + string(y)
23|    }
24| }
25|
26| func main() {
27|    fmt.Println(testing.AllocsPerRun(1, fc)) // 1
28|    fmt.Println(testing.AllocsPerRun(1, fd)) // 3
29| }
```

使用for-range循环遍历字符串中的码点

for-range循环控制中的range关键字后可以跟随一个字符串，用来遍历此字符串中的码点（而非字节元素）。 字符串中非法的UTF-8编码字节序列将被解读为Unicode替换码点值0xFFFD。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
```

```
 5| func main() {
 6|    s := "é    aπ囧"
 7|    for i, rn := range s {
 8|       fmt.Printf("%2v: 0x%x %v \n", i, rn, string(rn))
 9|    }
10|    fmt.Println(len(s))
11| }
```

此程序的输出如下：

```
 0: 0x65 e
 1: 0x301 ´
 3: 0x915
 6: 0x94d
 9: 0x937
12: 0x93f
15: 0x61 a
16: 0x3c0 π
18: 0x56e7 囧
21
```

从此输出结果可以看出：

1. 下标循环变量的值并非连续。原因是下标循环变量为字符串中字节的下标，而一个码点可能需要多个字节进行UTF-8编码。
2. 第一个字符é由两个码点（共三字节）组成，其中一个码点需要两个字节进行UTF-8编码。
3. 第二个字符　　由四个码点（共12字节）组成，每个码点需要三个字节进行UTF-8编码。
4. 英语字符a由一个码点组成，此码点只需一个字节进行UTF-8编码。
5. 字符π由一个码点组成，此码点只需两个字节进行UTF-8编码。
6. 汉字 囧 由一个码点组成，此码点只需三个字节进行UTF-8编码。

那么如何遍历一个字符串中的字节呢？使用传统for循环：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    s := "é    aπ囧"
 7|    for i := 0; i < len(s); i++ {
 8|       fmt.Printf("第%v个字节为0x%x\n", i, s[i])
 9|    }
10| }
```

当然，我们也可以利用前面介绍的编译器优化来使用for-range循环遍历一个字符串中的字节元素。 对于官方标准编译器来说，此方法比刚展示的方法效率更高。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     s := "é    aπ囧"
 7|     // 这里，[]byte(s)不需要深复制底层字节。
 8|     for i, b := range []byte(s) {
 9|        fmt.Printf("The byte at index %v: 0x%x \n", i, b)
10|     }
11| }
```

从上面几个例子可以看出，len(s)将返回字符串s中的字节数。 len(s)的时间复杂度为*O*(1)。 如何得到一个字符串中的码点数呢？使用刚介绍的for-range循环来统计一个字符串中的码点数是一种方法，使用unicode/utf8标准库包中的RuneCountInString（https://golang.google.cn/pkg/unicode/utf8/#RuneCountInString）是另一种方法。 这两种方法的效率基本一致。第三种方法为使用len([]rune(s))来获取字符串s中码点数。标准编译器从1.11版本开始，对此表达式做了优化以避免一个不必要的深复制，从而使得它的效率和前两种方法一致。 注意，这三种方法的时间复杂度均为*O*(n)。

更多字符串衔接方法

除了使用+运算符来衔接字符串，我们也可以用下面的方法来衔接字符串：

- fmt标准库包中的Sprintf/Sprint/Sprintln函数可以用来衔接各种类型的值的字符串表示，当然也包括字符串类型的值。
- 使用strings标准库包中的Join函数。
- bytes标准库包提供的Buffer类型可以用来构建一个字节切片，然后我们可以将此字节切片转换为一个字符串。
- 从Go 1.10开始，strings标准库包中的Builder类型可以用来拼接字符串。和bytes.Buffer类型类似，此类型内部也维护着一个字节切片，但是它在将此字节切片转换为字符串时避免了底层字节的深复制。

标准编译器对使用+运算符的字符串衔接做了特别的优化。 所以，一般说来，当所有被衔接的字符串都可以呈现在一条+字符串衔接语句中的情况下，使用+运算符进行字符串衔接是比较高效的。

语法糖：将字符串当作字节切片使用

在上一篇文章（第18章）中，我们了解到内置函数copy和append可以用来复制和添加切片元素。 事实上，做为一个特例，如果这两个函数的调用中的第一个实参为一个字节切片的话，那么第二个实参可以是一个字符串。 （对于append函数调用，字符串实参后必须跟随三个点...。） 换句话说，在此特例中，字符串可以当作字节切片来使用。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    hello := []byte("Hello ")
 7|    world := "world!"
 8|
 9|    // helloWorld := append(hello, []byte(world)...) // 正常
的语法
10|    helloWorld := append(hello, world...)            // 语法
糖
11|    fmt.Println(string(helloWorld))
12|
13|    helloWorld2 := make([]byte, len(hello) + len(world))
14|    copy(helloWorld2, hello)
15|    // copy(helloWorld2[len(hello):], []byte(world)) // 正常
的语法
16|    copy(helloWorld2[len(hello):], world)            // 语法
糖
17|    fmt.Println(string(helloWorld2))
18| }
```

更多关于字符串的比较

上面已经提到了比较两个字符串事实上逐个比较这两个字符串中的字节。 Go编译器一般会做出如下的优化：

- 对于==和!=比较，如果这两个字符串的长度不相等，则这两个字符串肯定不相等（无需进行字节比较）。
- 如果这两个字符串底层引用着字符串切片的指针相等，则比较结果等同于比较这两个字符串的长度。

所以两个相等的字符串的比较的时间复杂度取决于它们底层引用着字符串切片的指针是否相等。 如果相等，则对它们的比较的时间复杂度为*O*(1)，否则时间复杂度为*O*(n)。

上面已经提到了，对于标准编译器，一个字符串赋值完成之后，目标字符串和源字符串将共享同一个底层字节序列。 所以比较这两个字符串的代价很小。

一个例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "time"
 6| )
 7|
 8| func main() {
 9|     bs := make([]byte, 1<<26)
10|     s0 := string(bs)
11|     s1 := string(bs)
12|     s2 := s1
13|
14|     // s0、s1和s2是三个相等的字符串。
15|     // s0的底层字节序列是bs的一个深复制。
16|     // s1的底层字节序列也是bs的一个深复制。
17|     // s0和s1底层字节序列为两个不同的字节序列。
18|     // s2和s1共享同一个底层字节序列。
19|
20|     startTime := time.Now()
21|     _ = s0 == s1
22|     duration := time.Now().Sub(startTime)
23|     fmt.Println("duration for (s0 == s1):", duration)
24|
25|     startTime = time.Now()
26|     _ = s1 == s2
27|     duration = time.Now().Sub(startTime)
28|     fmt.Println("duration for (s1 == s2):", duration)
29| }
```

输出如下：

```
duration for (s0 == s1): 10.462075ms
duration for (s1 == s2): 136ns
```

1ms等于1000000ns！所以请尽量避免比较两个很长的不共享底层字节序列的相等的（或者几乎相等的）字符串。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

函数

函数声明和调用（第9章）已经在前面的文章中解释过了。当前这篇文章将介绍更多关于函数的概念和细节。

事实上，在Go中，函数是一种一等公民类型。换句话说，我们可以把函数当作值来使用。尽管Go是一门静态语言，但是Go函数的灵活性宛如甚至超越了很多动态语言。

Go中有一些内置函数，这些函数展示在builtin（https://docs.go101.org/std/pkg/builtin.html）和unsafe（https://docs.go101.org/std/pkg/unsafe.html）标准包中。内置函数和自定义函数有很多差别。这些差别将在下面逐一提及。

函数签名（function signature）和函数类型

刚已经提到了，在Go中，函数是一种一等公民类型。一个函数类型的字面表示形式由一个 func 关键字和一个函数签名字面表示表示形式组成。一个函数签名由一个输入参数类型列表和一个输出结果类型列表组成。参数名称和结果名称可以出现函数签名的字面表示形式中，但是它们并不重要。

func 关键字可以出现在函数签名的字面形式中，也可以不出现。鉴于此，我们常常混淆使用函数类型（见下）和函数签名这两个概念。

下面是一个函数类型的字面形式：

```
func (a int, b string, c string) (x int, y int, z bool)
```

从前面的函数声明和调用（第9章）一文中，我们了解到连续的同类型参数和结果可以声明在一块儿。所以上面的字面形式等价于：

```
func (a int, b, c string) (x, y int, z bool)
```

参数名称和结果名称并不重要，只要它们不重名即可。上面两个字面形式等价于下面这个：

```
func (x int, y, z string) (a, b int, c bool)
```

参数名和结果名可以是空标识符 _ 。上面的字面形式等价于：

```
func (_ int, _, _ string) (_, _ int, _ bool)
```

函数参数列表中的参数名或者结果列表中的结果名可以同时省略（即匿名）。上面的字面形式等价于：

```
func (int, string, string) (int, int, bool) // 标准函数字面形式
func (a int, b string, c string) (int, int, bool)
func (x int, _ string, z string) (int, int, bool)
func (int, string, string) (x int, y int, z bool)
func (int, string, string) (a int, b int, _ bool)
```

所有上面列出的函数类型字面形式表示同一个（无名）函数类型。

参数列表必须用一对小括号`()`括起来，即使此列表为空。如果一个函数类型一个结果列表为空，则它可以在函数类型的字面形式中被省略掉。当一个结果列表含有最多一个结果，则此结果列表的字面形式在它不包含结果名称的时候可以不用括号`()`括起来。

```
// 这三个函数类型字面形式是等价的。
func () (x int)
func () (int)
func () int

// 这两个函数类型字面形式是等价的。
func (a int, b string) ()
func (a int, b string)
```

变长参数和变长参数函数类型

一个函数的最后一个参数可以是一个变长参数。一个函数可以最多有一个变长参数。一个变长参数的类型总为一个切片类型。变长参数在声明的时候必须在它的（切片）类型的元素类型前面前置三个点`...`，以示这是一个变长参数。两个变长函数类型的例子：

```
func (values ...int64) (sum int64)
func (sep string, tokens ...string) string
```

一个变长函数类型和一个非变长函数类型绝对不可能是同一个类型。

后面的一节将展示几个变长函数声明和使用的例子。

所有的函数类型都属于不可比较类型

Go类型系统概述（第14章）一文已经提到了函数类型属于不可比较类型。但是，和映射值以及切片值类似，一个函数值可以和类型不确定的`nil`比较。（函数值将在

本文最后一节介绍。）

因为函数类型属于不可比较类型，所以函数类型不可用做映射类型的键值类型。

函数原型（function prototype）

一个函数原型由一个函数名称和一个函数类型（或者说一个函数签名）组成。它的字面形式由一个func关键字、一个函数名和一个函数签名字面形式组成。

一个函数原型的例子：

```
func Double(n int) (result int)
```

换句话说，一个函数原型可以看作是一个不带函数体的函数声明；或者说一个函数声明由一个函数原型和一个函数体组成。

变长函数声明和变长函数调用

普通非变长函数的声明和调用已经在函数声明和调用（第9章）一文中介绍过了。本节将介绍变长函数的声明和调用。

变长函数声明

变长函数声明和普通函数声明类似，只不过最后一个参数必须为变长参数。一个变长参数在函数体内将被视为一个切片。

```
 1| // Sum返回所有输入实参的和。
 2| func Sum(values ...int64) (sum int64) {
 3|     // values的类型为[]int64。
 4|     sum = 0
 5|     for _, v := range values {
 6|         sum += v
 7|     }
 8|     return
 9| }
10|
11| // Concat是一个低效的字符串拼接函数。
12| func Concat(sep string, tokens ...string) string {
13|     // tokens的类型为[]string。
14|     r := ""
15|     for i, t := range tokens {
16|         if i != 0 {
```

```
17|           r += sep
18|       }
19|       r += t
20|    }
21|    return r
22| }
```

从上面的两个变长参数函数声明可以看出，如果一个变长参数的类型部分为`...T`，则此变长参数的类型实际为`[]T`。

事实上，在前面的文章中多次使用过的`fmt`标准库包中的`Print`、`Println`和`Printf`函数均为变长参数函数。它们的声明大致如下：

```
 1| func Print(a ...interface{}) (n int, err error)
 2| func Printf(format string, a ...interface{}) (n int, err
error)
 3| func Println(a ...interface{}) (n int, err error)
```

这三个函数中的变长参数的类型均为`[]interface{}`。此类型的元素类型为`interface{}`，这是一个接口类型。接口类型和接口值将在后面的接口（第23章）一文中详述。

变长参数函数调用

在变长参数函数调用中，可以使用两种风格的方式将实参传递给类型为`[]T`的变长形参：

1. 传递一个切片做为实参。此切片必须可以被赋值给类型为`[]T`的值（或者说此切片可以被隐式转换为类型`[]T`）。此实参切片后必须跟随三个点`...`。
2. 传递零个或者多个可以被隐式转换为`T`的实参（或者说这些实参可以赋值给类型为`T`的值）。这些实参将被添加入一个匿名的在运行时刻创建的类型为`[]T`的切片中，然后此切片将被传递给此函数调用。

注意，这两种风格的方式不可在同一个变长参数函数调用中混用。

下面这个例子展示了一些变长参数函数调用：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func Sum(values ...int64) (sum int64) {
 6|    sum = 0
```

```
 7|    for _, v := range values {
 8|       sum += v
 9|    }
10|    return
11| }
12|
13| func main() {
14|    a0 := Sum()
15|    a1 := Sum(2)
16|    a3 := Sum(2, 3, 5)
17|    // 上面三行和下面三行是等价的。
18|    b0 := Sum([]int64{}...) // <=> Sum(nil...)
19|    b1 := Sum([]int64{2}...)
20|    b3 := Sum([]int64{2, 3, 5}...)
21|    fmt.Println(a0, a1, a3) // 0 2 10
22|    fmt.Println(b0, b1, b3) // 0 2 10
23| }
```

另一个展示了一些变长参数函数调用的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func Concat(sep string, tokens ...string) (r string) {
 6|    for i, t := range tokens {
 7|       if i != 0 {
 8|          r += sep
 9|       }
10|       r += t
11|    }
12|    return
13| }
14|
15| func main() {
16|    tokens := []string{"Go", "C", "Rust"}
17|    langsA := Concat(",", tokens...)        // 风格1
18|    langsB := Concat(",", "Go", "C","Rust") // 风格2
19|    fmt.Println(langsA == langsB)           // true
20| }
```

下面这个例子编译不通过，因为两种调用风格混用了。

```
 1| package main
 2|
```

```
 3| // 这两个函数的声明见前面几例。
 4| func Sum(values ...int64) (sum int64) {......}
 5| func Concat(sep string, tokens ...string) string {......}
 6|
 7| func main() {
 8|     // 下面两行报同样的错：实参数目太多了。
 9|     _ = Sum(2, []int64{3, 5}...)
10|     _ = Concat(",", "Go", []string{"C", "Rust"}...)
11| }
```

更多关于函数声明和函数调用的事实

同一个包中可以同名的函数

一般来说，同一个包中声明的函数的名称不能重复，但有两个例外：

1. 同一个包内可以声明若干个原型为`func ()`的名称为`init`的函数（第10章）。
2. 多个函数的名称可以被声明为空标识符`_`。这样声明的函数不可被调用。

某些函数调用是在编译时刻被估值的

大多数函数调用都是在运行时刻被估值的。但`unsafe`标准库包中的函数的调用都是在编译时刻估值的。另外，某些其它内置函数（比如`len`和`cap`等）的调用在所传实参满足一定的条件的时候也将在编译时刻估值。详见在编译时刻估值的函数调用（第46章）。

所有的函数调用的传参均属于值复制

再重申一次，和赋值一样，传参也属于值（浅）复制。当一个值被复制时，只有它的直接部分（第17章）被复制了。

不含函数体的函数声明

我们可以使用Go汇编（Go assembly）（https://golang.google.cn/doc/asm）来实现一个Go函数。Go汇编代码放在后缀为`.a`的文件中。一个使用Go汇编实现的函数依旧必须在一个`*.go`文件中声明，但是它的声明必须不能含有函数体。换句话说，一个使用Go汇编实现的函数的声明中只含有它的原型。

某些有返回值的函数可以不必返回

如果一个函数有返回值，则它的函数体内的最后一条语句必须为一条终止语句（https://golang.google.cn/ref/spec#Terminating_statements）。Go中有多种终止语句，return语句只是其中一种。所以一个有返回值的函数的体内不一定需要一个return语句。比如下面两个函数（它们均可编译通过）：

```
1| func fa() int {
2|     a:
3|     goto a
4| }
5|
6| func fb() bool {
7|     for{}
8| }
```

自定义函数的调用返回结果可以被舍弃，但是某些内置函数的调用返回结果不可被舍弃

自定义函数的调用结果都是可以被舍弃掉的。但是大多数内置函数（除了recover和copy）的调用结果都是不可被舍弃的。调用结果不可被舍弃的函数是不可以被用做延迟调用函数和协程起始函数的，比如append函数。

有返回值的函数的调用是一种表达式

一个有且只有一个返回值的函数的每个调用总可以被当成一个单值表达式使用。比如，它可以被内嵌在其它函数调用中当作实参使用，或者可以被当作其它表达式中的操作数使用。

如果一个有多个返回结果的函数的一个调用的返回结果没有被舍弃，则此调用可以当作一个多值表达式使用在两种场合：

1. 此调用可以在一个赋值语句中当作源值来使用，但是它不能和其它源值掺和到一块。
2. 此调用可以内嵌在另一个函数调用中当作实参来使用，但是它不能和其它实参掺和到一块。

一个例子：

```
1| package main
2|
3| func HalfAndNegative(n int) (int, int) {
```

```
 4|    return n/2, -n
 5| }
 6|
 7| func AddSub(a, b int) (int, int) {
 8|    return a+b, a-b
 9| }
10|
11| func Dummy(values ...int) {}
12|
13| func main() {
14|    // 这几行编译没问题。
15|    AddSub(HalfAndNegative(6))
16|    AddSub(AddSub(AddSub(7, 5)))
17|    AddSub(AddSub(HalfAndNegative(6)))
18|    Dummy(HalfAndNegative(6))
19|    _, _ = AddSub(7, 5)
20|
21|    // 下面这几行编译不通过。
22|    /*
23|    _, _, _ = 6, AddSub(7, 5)
24|    Dummy(AddSub(7, 5), 9)
25|    Dummy(AddSub(7, 5), HalfAndNegative(6))
26|    */
27| }
```

注意，在目前的标准编译器的实现中，有几个内置函数破坏了上述规则的普遍性（第49章）。

函数值

本文开头已经介绍了函数类型是Go中天然支持的一种类型。函数类型的值称为函数值。在字面上，函数类型的零值也使用预定义的`nil`来表示。

当我们声明了一个函数的时候，我们实际上同时声明了一个不可修改的函数值。此函数值用此函数的名称来标识。此函数值的类型的字面表示形式为此函数的原型刨去函数名部分。

注意：内置函数和`init`函数不可被用做函数值。

任何函数值都可以被当作普通声明函数来调用。调用一个nil函数来开启一个协程将产生一个致命的不可恢复的错误，此错误将使整个程序崩溃。在其它情况下调用一个nil函数将产生一个可恢复的恐慌。

从值部（第17章）一文，我们得知，当一个函数值被赋给另一个函数值后，这两个函数值将共享底层部分（内部的函数结构）。换句话说，这两个函数值表示的函数可以看作是同一个函数。调用它们的效果是相同的。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func Double(n int) int {
 6|    return n + n
 7| }
 8|
 9| func Apply(n int, f func(int) int) int {
10|    return f(n) // f的类型为"func(int) int"
11| }
12|
13| func main() {
14|    fmt.Printf("%T\n", Double) // func(int) int
15|    // Double = nil // error: Double是不可修改的
16|
17|    var f func(n int) int // 默认值为nil
18|    f = Double
19|    g := Apply
20|    fmt.Printf("%T\n", g) // func(int, func(int) int) int
21|
22|    fmt.Println(f(9))         // 18
23|    fmt.Println(g(6, Double)) // 12
24|    fmt.Println(Apply(6, f))  // 12
25| }
```

在上例中，`g(6, Double)`和`Apply(6, f)`是等价的。

在实践中，我们常常将一个匿名函数赋值给一个函数类型的变量，从而可以在以后多次调用此匿名函数。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    // 此函数返回一个函数类型的结果，亦即闭包（closure）。
 7|    isMultipleOfX := func (x int) func(int) bool {
 8|       return func(n int) bool {
```

```
 9|           return n%x == 0
10|       }
11|    }
12|
13|    var isMultipleOf3 = isMultipleOfX(3)
14|    var isMultipleOf5 = isMultipleOfX(5)
15|    fmt.Println(isMultipleOf3(6))  // true
16|    fmt.Println(isMultipleOf3(8))  // false
17|    fmt.Println(isMultipleOf5(10)) // true
18|    fmt.Println(isMultipleOf5(12)) // false
19|
20|    isMultipleOf15 := func(n int) bool {
21|       return isMultipleOf3(n) && isMultipleOf5(n)
22|    }
23|    fmt.Println(isMultipleOf15(32)) // false
24|    fmt.Println(isMultipleOf15(60)) // true
25| }
```

Go中所有的函数都可以看作是闭包，这是Go语言编程常常给人一种和动态脚本语言一样灵活的一个重要原因。

遍历函数值（从Go 1.23开始）

从Go 1.23开始，底层类型为下列函数类型的函数值可以使用`for-range`循环来遍历。

```
// K和V是特定类型

func(yield func() bool)

func(yield func(V) bool)

func(yield func(K, V) bool)
```

这样的函数值称为推遍历器（push iterator），常简称为遍历器。

当使用一个`for-range`循环遍历这样的一个遍历器函数值时，这个遍历器函数值将被调用（一次）并被传入一个隐式创建的`yield`回调函数。此`yield`回调函数返回一个`bool`结果。当它返回`false`时，这个遍历器函数的调用应该（但并不强求一定）立即退出；否则（当此`yield`回调函数返回`true`时），这个遍历器函数应该继续执行，直到自然退出。

下面是一些使用遍历器函数的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func Loop3(yield func() bool) {
 6|    for range 3 {
 7|       if (!yield()) {
 8|          return
 9|       }
10|    }
11| }
12|
13| func OneDigitNumbers(onValue func(int) bool) {
14|    for i := range 10 {
15|       if (!onValue(i)) {
16|          return
17|       }
18|    }
19| }
20|
21| func SquareLessThan50(onKeyValue func(int, int) bool) {
22|    for i := range 8 {
23|       if (!onKeyValue(i, i*i)) {
24|          return
25|       }
26|    }
27| }
28|
29| func main() {
30|    var n = 0
31|    for range Loop3 {
32|       fmt.Print(n)
33|       n++
34|    }
35|    fmt.Println()
36|    // 输出：012
37|
38|    for i := range OneDigitNumbers {
39|       fmt.Print(i)
40|    }
41|    fmt.Println()
42|    // 输出：0123456789
43|
44|    for i, ii := range SquareLessThan50 {
45|       fmt.Printf("%v:%v ", i, ii)
```

```
46|    }
47|    fmt.Println()
48|    // 输出：0:0 1:1 2:4 3:9 4:16 5:25 6:36 7:49
49| }
```

上面这些for-range循环和下面这些相应的函数调用是等价的：

```
 1| func main() {
 2|    var n = 0
 3|    Loop3(func() bool {
 4|       fmt.Print(n)
 5|       n++
 6|       return true
 7|    })
 8|    fmt.Println()
 9|
10|    OneDigitNumbers(func(i int) bool {
11|       fmt.Print(i)
12|       return true
13|    })
14|    fmt.Println()
15|
16|    SquareLessThan50(func(i, ii int) bool {
17|       fmt.Printf("%v:%v ", i, ii)
18|       return true
19|    })
20|    fmt.Println()
21| }
```

是的，我们可以认为Go 1.23引入的遍历函数特性是一个语法糖。

请阅读这篇官方博客文章（https://go.dev/blog/range-functions）了解更多关于遍历器的内容。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

通道

通道是Go中的一种一等公民类型。它是Go的招牌特性之一。 和另一个招牌特性协程（第13章）一起，这两个招牌特性使得使用Go进行并发编程（concurrent programming）变得十分方便和有趣，并且大大降低了并发编程的难度。

通道的主要作用是用来实现并发同步（第13章）。 本篇文章将列出所有的和通道相关的概念、语法和规则。为了更好地理解通道，本文也对通道的可能的内部实现略加介绍。

本篇文章中的信息量对于Go初学者来说可能有些密集。本文的某些段落可能需要反复阅读几遍才能有效吸收、消化和理解。

通道（channel）介绍

Go语言设计团队的首任负责人*Rob Pike*对并发编程的一个建议是**不要让计算通过共享内存来通讯，而应该让它们通过通讯来共享内存**。 通道机制就是这种哲学的一个设计结果。（在Go编程中，我们可以认为一个计算就是一个协程。）

通过共享内存来通讯和通过通讯来共享内存是并发编程中的两种编程风格。 当通过共享内存来通讯的时候，我们需要一些传统的并发同步技术（比如互斥锁）来避免数据竞争。

Go提供了一种独特的并发同步技术来实现通过通讯来共享内存。此技术即为通道。我们可以把一个通道看作是在一个程序内部的一个先进先出（FIFO：first in first out）数据队列。 一些协程可以向此通道发送数据，另外一些协程可以从此通道接收数据。

随着一个数据值的传递（发送和接收），一些数据值的所有权从一个协程转移到了另一个协程。 当一个协程发送一个值到一个通道，我们可以认为此协程释放了（通过此发送值可以访问到的）一些值的所有权。 当一个协程从一个通道接收到一个值，我们可以认为此协程获取了（通过此接受值可以访问到的）一些值的所有权。

当然，在通过通道传递数据的时候，也可能没有任何所有权发生转移。

所有权发生转移的值常常被传递的值所引用着，但有时候也并非如此。 在Go中，数据所有权的转移并非体现在语法上，而是体现在逻辑上。 Go通道可以帮助程序员轻松地避免数据竞争，但不会防止程序员因为犯错而写出错误的并发代码的情况发生。

尽管Go也支持几种传统的数据同步技术，但是只有通道为一等公民。 通道是Go中的一种类型，所以我们可以无需引进任何代码包就可以使用通道。 几种传统的数据同

步技术提供在sync和sync/atomic标准库包中。

实事求是地说，每种并发同步技术都有它们各自的最佳应用场景，但是通道的应用范围更广（第37章）。 使用通道来做同步常常可以使得代码看上去更整洁和易于理解。

通道的一个问题是通道的编程体验常常很有趣以至于程序员们经常在并非是通道的最佳应用场景中仍坚持使用通道。

通道类型和值

和数组、切片以及映射类型一样，每个通道类型也有一个元素类型。 一个通道只能传送它的（通道类型的）元素类型的值。

通道可以是双向的，也可以是单向的。

- 字面形式chan T表示一个元素类型为T的双向通道类型。 编译器允许从此类型的值中接收和向此类型的值中发送数据。
- 字面形式chan<- T表示一个元素类型为T的单向发送通道类型。 编译器不允许从此类型的值中接收数据。
- 字面形式<-chan T表示一个元素类型为T的单向接收通道类型。 编译器不允许向此类型的值中发送数据。

双向通道chan T的值可以被隐式转换为单向通道类型chan<- T和<-chan T，但反之不行（即使显式也不行）。 类型chan<- T和<-chan T的值也不能相互转换。

每个通道值有一个容量属性。此属性的意义将在下一节中得到解释。 一个容量为0的通道值称为一个非缓冲通道（unbuffered channel），一个容量不为0的通道值称为一个缓冲通道（buffered channel）。

通道类型的零值也使用预声明的nil来表示。 一个非零通道值必须通过内置的make函数来创建。 比如make(chan int, 10)将创建一个元素类型为int的通道值。 第二个参数指定了欲创建的通道的容量。此第二个实参是可选的，它的默认值为0。

通道值的比较

所有通道类型均为可比较类型。

从值部（第17章）一文，我们了解到一个通道值可能含有底层部分。 当一个通道值被赋给另一个通道值后，这两个通道值将共享相同的底层部分。 换句话说，这两个

通道引用着同一个底层的内部通道对象。 比较这两个通道的结果为true。

通道操作

Go中有五种通道相关的操作。假设一个通道（值）为ch，下面列出了这五种操作的语法或者函数调用。

1. 调用内置函数close来关闭一个通道：

```
close(ch)
```

传给close函数调用的实参必须为一个通道值，并且此通道值不能为单向接收的。

2. 使用下面的语法向通道ch发送一个值v：

```
ch <- v
```

v必须能够赋值给通道ch的元素类型。 ch不能为单向接收通道。 <-称为数据发送操作符。

3. 使用下面的语法从通道ch接收一个值：

```
<-ch
```

如果一个通道操作不永久阻塞，它总会返回至少一个值，此值的类型为通道ch的元素类型。 ch不能为单向发送通道。 <-称为数据接收操作符，是的它和数据发送操作符的表示形式是一样的。

在大多数场合下，一个数据接收操作可以被认为是一个单值表达式。 但是，当一个数据接收操作被用做一个赋值语句中的唯一的源值的时候，它可以返回第二个可选的类型不确定的布尔值返回值从而成为一个多值表达式。 此类型不确定的布尔值表示第一个返回值是否是在通道被关闭之前被发送的。 （从后面的章节，我们将得知我们可以从一个已关闭的通道中接收到无穷个值。）

数据接收操作在赋值中被用做源值的例子：

```
v = <-ch
v, sentBeforeClosed = <-ch
```

4. 查询一个通道的容量：

```
cap(ch)
```

其中cap是一个已经在容器类型（第18章）一文中介绍过的内置函数。 cap的返回值的类型为内置类型int。

5. 查询一个通道的长度：

```
len(ch)
```

其中len是一个已经在容器类型（第18章）一文中介绍过的内置函数。 len的返回值的类型也为内置类型int。 一个通道的长度是指当前有多少个已被发送到此通道但还未被接收出去的元素值。

Go中大多数的基本操作都是未同步的。换句话说，它们都不是并发安全的。 这些操作包括赋值、传参、和各种容器值操作等。 但是，上面列出的五种通道相关的操作都已经同步过了，因此它们可以在并发协程中安全运行而无需其它同步操作。

注意：通道的赋值和其它类型值的赋值一样，是未同步的。 同样，将刚从一个通道接收出来的值赋给另一个值也是未同步的。

如果被查询的通道为一个nil零值通道，则cap和len函数调用都返回0。 这两个操作是如此简单，所以后面将不再对它们进行详解。 事实上，这两个操作在实践中很少使用。

通道的发送、接收和关闭操作将在下一节得到详细解释。

通道操作详解

为了让解释简单清楚，在本文后续部分，通道将被归为三类：

1. 零值（nil）通道；
2. 非零值但已关闭的通道；
3. 非零值并且尚未关闭的通道。

下表简单地描述了三种通道操作施加到三类通道的结果。

操作	一个零值nil通道	一个非零值但已关闭的通道	一个非零值且尚未关闭的通道
关闭	产生恐慌	产生恐慌	成功关闭(C)
发送数据	永久阻塞	产生恐慌	阻塞或者成功发送(B)
接收数据	永久阻塞	永不阻塞(D)	阻塞或者成功接收(A)

对于上表中的五种未打上标的情形，规则很简单：

- 关闭一个nil通道或者一个已经关闭的通道将产生一个恐慌。

- 向一个已关闭的通道发送数据也将导致一个恐慌。
- 向一个nil通道发送数据或者从一个nil通道接收数据将使当前协程永久阻塞。

下面将详细解释其它四种被打了上标（A/B/C/D）的情形。

为了更好地理解通道和为了后续讲解方便，先了解一下通道类型的大致内部实现是很有帮助的。

我们可以认为一个通道内部维护了三个队列（均可被视为先进先出队列）：

1. 接收数据协程队列（可以看做是先进先出队列但其实并不完全是，见下面解释）。此队列是一个没有长度限制的链表。 此队列中的协程均处于阻塞状态，它们正等待着从此通道接收数据。
2. 发送数据协程队列（可以看做是先进先出队列但其实并不完全是，见下面解释）。此队列也是一个没有长度限制的链表。 此队列中的协程亦均处于阻塞状态，它们正等待着向此通道发送数据。 此队列中的每个协程将要发送的值（或者此值的指针，取决于具体编译器实现）和此协程一起存储在此队列中。
3. 数据缓冲队列。这是一个循环队列（绝对先进先出），它的长度为此通道的容量。此队列中存放的值的类型都为此通道的元素类型。 如果此队列中当前存放的值的个数已经达到此通道的容量，则我们说此通道已经处于满槽状态。 如果此队列中当前存放的值的个数为零，则我们说此通道处于空槽状态。 对于一个非缓冲通道（容量为零），它总是同时处于满槽状态和空槽状态。

每个通道内部维护着一个互斥锁用来在各种通道操作中防止数据竞争。

通道操作情形A：当一个协程R尝试从一个非零且尚未关闭的通道接收数据的时候，此协程R将首先尝试获取此通道的锁，成功之后将执行下列步骤，直到其中一个步骤的条件得到满足。

1. 如果此通道的缓冲队列不为空（这种情况下，接收数据协程队列必为空），此协程R将从缓冲队列取出（接收）一个值。 如果发送数据协程队列不为空，一个发送协程将从此队列中弹出，此协程欲发送的值将被推入缓冲队列。此发送协程将恢复至运行状态。 接收数据协程R继续运行，不会阻塞。对于这种情况，此数据接收操作为一个**非阻塞操作**。
2. 否则（即此通道的缓冲队列为空），如果发送数据协程队列不为空（这种情况下，此通道必为一个非缓冲通道）， 一个发送数据协程将从此队列中弹出，此协程欲发送的值将被接收数据协程R接收。此发送协程将恢复至运行状态。接收数据协程R继续运行，不会阻塞。对于这种情况，此数据接收操作为一个**非阻塞操作**。
3. 对于剩下的情况（即此通道的缓冲队列和发送数据协程队列均为空），此接收数据协程R将被推入接收数据协程队列，并进入阻塞状态。 它以后可能会被另一个发送数据协程唤醒而恢复运行。 对于这种情况，此数据接收操作为一个**阻塞操作**。

通道操作情形B：当一个协程S尝试向一个非零且尚未关闭的通道发送数据的时候，此协程S将首先尝试获取此通道的锁，成功之后将执行下列步骤，直到其中一个步骤的条件得到满足。

1. 如果此通道的接收数据协程队列不为空（这种情况下，缓冲队列必为空），一个接收数据协程将从此队列中弹出，此协程将接收到发送协程S发送的值。此接收协程将恢复至运行状态。发送数据协程S继续运行，不会阻塞。对于这种情况，此数据发送操作为一个**非阻塞操作**。
2. 否则（接收数据协程队列为空），如果缓冲队列未满（这种情况下，发送数据协程队列必为空），发送协程S欲发送的值将被推入缓冲队列，发送数据协程S继续运行，不会阻塞。对于这种情况，此数据发送操作为一个**非阻塞操作**。
3. 对于剩下的情况（接收数据协程队列为空，并且缓冲队列已满），此发送协程S将被推入发送数据协程队列，并进入阻塞状态。它以后可能会被另一个接收数据协程唤醒而恢复运行。对于这种情况，此数据发送操作为一个**阻塞操作**。

上面已经提到过，一旦一个非零通道被关闭，继续向此通道发送数据将产生一个恐慌。注意，向关闭的通道发送数据属于一个**非阻塞操作**。

通道操作情形C：当一个协程成功获取到一个非零且尚未关闭的通道的锁并且准备关闭此通道时，下面两步将依次执行：

1. 如果此通道的接收数据协程队列不为空（这种情况下，缓冲队列必为空），此队列中的所有协程将被依个弹出，并且每个协程将接收到此通道的元素类型的一个零值，然后恢复至运行状态。
2. 如果此通道的发送数据协程队列不为空，此队列中的所有协程将被依个弹出，并且每个协程中都将产生一个恐慌（因为向已关闭的通道发送数据）。这就是我们在上面说并发地关闭一个通道和向此通道发送数据这种情形属于不良设计的原因。事实上，在数据竞争侦测编译选项（`-race`）打开时，Go官方标准运行时将很可能会对并发地关闭一个通道和向此通道发送数据这种情形报告成数据竞争。

注意：当一个缓冲队列不为空的通道被关闭之后，它的缓冲队列不会被清空，其中的数据仍然可以被后续的数据接收操作所接收到。详见下面的对情形D的解释。

通道操作情形D：一个非零通道被关闭之后，此通道上的后续数据接收操作将永不会阻塞。此通道的缓冲队列中存储数据仍然可以被接收出来。伴随着这些接收出来的缓冲数据的第二个可选返回（类型不确定布尔）值仍然是`true`。一旦此缓冲队列变为空，后续的数据接收操作将永不阻塞并且总会返回此通道的元素类型的零值和值为`false`的第二个可选返回结果。上面已经提到了，一个接收操作的第二个可选返回（类型不确定布尔）结果表示一个接收到的值是否是在此通道被关闭之前

发送的。 如果此返回值为false，则第一个返回值必然是一个此通道的元素类型的零值。

知道哪些通道操作是阻塞的和哪些是非阻塞的对正确理解后面将要介绍的select流程控制机制非常重要。

如果一个协程被从一个通道的某个队列中（不论发送数据协程队列还是接收数据协程队列）弹出，并且此协程是在一个select控制流程（#select）中推入到此队列的，那么此协程将在下面将要讲解的select控制流程的执行步骤（#select-implementation）中的第9步中恢复至运行状态，并且同时它会被从相应的select控制流程中的相关的若干通道的协程队列中移除掉。

根据上面的解释，我们可以得出如下的关于一个通道的内部的三个队列的各种事实：

- 如果一个通道已经关闭了，则它的发送数据协程队列和接收数据协程队列肯定都为空，但是它的缓冲队列可能不为空。
- 在任何时刻，如果缓冲队列不为空，则接收数据协程队列必为空。
- 在任何时刻，如果缓冲队列未满，则发送数据协程队列必为空。
- 如果一个通道是缓冲的，则在任何时刻，它的发送数据协程队列和接收数据协程队列之一必为空。
- 如果一个通道是非缓冲的，则在任何时刻，一般说来，它的发送数据协程队列和接收数据协程队列之一必为空， 但是有一个例外：一个协程可能在一个select流程控制（#select）中同时被推入到此通道的发送数据协程队列和接收数据协程队列中。

一些通道的使用例子

来看一些通道的使用例子来加深一下对上一节中的解释的理解。

一个简单的通过一个非缓冲通道实现的请求/响应的例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "time"
 6| )
 7|
 8| func main() {
 9|     c := make(chan int) // 一个非缓冲通道
10|     go func(ch chan<- int, x int) {
11|         time.Sleep(time.Second)
```

```
12|       // <-ch    // 此操作编译不通过
13|       ch <- x*x  // 阻塞在此，直到发送的值被接收
14|    }(c, 3)
15|    done := make(chan struct{})
16|    go func(ch <-chan int) {
17|       n := <-ch      // 阻塞在此，直到有值发送到c
18|       fmt.Println(n) // 9
19|       // ch <- 123   // 此操作编译不通过
20|       time.Sleep(time.Second)
21|       done <- struct{}{}
22|    }(c)
23|    <-done // 阻塞在此，直到有值发送到done
24|    fmt.Println("bye")
25| }
```

输出结果：

```
9
bye
```

下面的例子使用了一个缓冲通道。此例子程序并非是一个并发程序，它只是为了展示缓冲通道的使用。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    c := make(chan int, 2) // 一个容量为2的缓冲通道
 7|    c <- 3
 8|    c <- 5
 9|    close(c)
10|    fmt.Println(len(c), cap(c)) // 2 2
11|    x, ok := <-c
12|    fmt.Println(x, ok) // 3 true
13|    fmt.Println(len(c), cap(c)) // 1 2
14|    x, ok = <-c
15|    fmt.Println(x, ok) // 5 true
16|    fmt.Println(len(c), cap(c)) // 0 2
17|    x, ok = <-c
18|    fmt.Println(x, ok) // 0 false
19|    x, ok = <-c
20|    fmt.Println(x, ok) // 0 false
21|    fmt.Println(len(c), cap(c)) // 0 2
22|    close(c) // 此行将产生一个恐慌
```

```
23|    c <- 7   // 如果上一行不存在，此行也将产生一个恐慌。
24| }
```

一场永不休场的足球比赛：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6| )
 7|
 8| func main() {
 9|    var ball = make(chan string)
10|    kickBall := func(playerName string) {
11|       for {
12|          fmt.Print(<-ball, "传球", "\n")
13|          time.Sleep(time.Second)
14|          ball <- playerName
15|       }
16|    }
17|    go kickBall("张三")
18|    go kickBall("李四")
19|    go kickBall("王二麻子")
20|    go kickBall("刘大")
21|    ball <- "裁判"   // 开球
22|    var c chan bool // 一个零值nil通道
23|    <-c             // 永久阻塞在此
24| }
```

请阅读通道用例大全（第37章）来查看更多通道的使用例子。

通道的元素值的传递都是复制过程

在一个值被从一个协程传递到另一个协程的过程中，此值将被复制至少一次。 如果此传递值曾经在某个通道的缓冲队列中停留过，则它在此传递过程中将被复制两次。 一次复制发生在从发送协程向缓冲队列推入此值的时候，另一个复制发生在接收协程从缓冲队列取出此值的时候。 和赋值以及函数调用传参一样，当一个值被传递时，只有它的直接部分被复制（第17章）。

对于官方标准编译器，最大支持的通道的元素类型的尺寸为65535。 但是，一般说来，为了在数据传递过程中避免过大的复制成本，我们不应该使用尺寸很大的通道元素类型。 如果欲传送的值的尺寸较大，应该改用指针类型做为通道的元素类型。

关于通道和协程的垃圾回收

注意，一个通道被其发送数据协程队列和接收数据协程队列中的所有协程引用着。因此，如果一个通道的这两个队列只要有一个不为空，则此通道肯定不会被垃圾回收。 另一方面，如果一个协程处于一个通道的某个协程队列之中，则此协程也肯定不会被垃圾回收，即使此通道仅被此协程所引用。 事实上，一个协程只有在退出后才能被垃圾回收。

数据接收和发送操作都属于简单语句

数据接收和发送操作都属于简单语句（第11章）。 另外一个数据接收操作总是可以被用做一个单值表达式。 简单语句和表达式可以被用在一些控制流程（第12章）的某些部分。

在下面这个例子中，数据接收和发送操作被用在两个`for`循环的初始化和步尾语句。

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "time"
 6| )
 7|
 8| func main() {
 9|     fibonacci := func() chan uint64 {
10|         c := make(chan uint64)
11|         go func() {
12|             var x, y uint64 = 0, 1
13|             for ; y < (1 << 63); c <- y { // 步尾语句
14|                 x, y = y, x+y
15|             }
16|             close(c)
17|         }()
18|         return c
19|     }
20|     c := fibonacci()
21|     for x, ok := <-c; ok; x, ok = <-c { // 初始化和步尾语句
22|         time.Sleep(time.Second)
23|         fmt.Println(x)
24|     }
25| }
```

for-range应用于通道

for-range循环控制流程也适用于通道。 此循环将不断地尝试从一个通道接收数据，直到此通道关闭并且它的缓冲队列为空为止。 和应用于数组/切片/映射的for-range语法不同，应用于通道的for-range语法中最多只能出现一个循环变量，此循环变量用来存储接收到的值。

```
1| for v := range aChannel {
2|     // 使用v
3| }
```

等价于

```
1| for {
2|     v, ok = <-aChannel
3|     if !ok {
4|         break
5|     }
6|     // 使用v
7| }
```

当然，这里的通道aChannel一定不能为一个单向发送通道。 如果它是一个nil零值，则此for-range循环将使当前协程永久阻塞。

上一节中的例子中的最后一个for循环可以改写为下面这样：

```
1|     for x := range c {
2|         time.Sleep(time.Second)
3|         fmt.Println(x)
4|     }
```

select-case分支流程控制代码块

Go中有一个专门为通道设计的select-case分支流程控制语法。 此语法和switch-case分支流程控制语法很相似。 比如，select-case流程控制代码块中也可以有若干case分支和最多一个default分支。 但是，这两种流程控制也有很多不同点。在一个select-case流程控制中，

- select关键字和{之间不允许存在任何表达式和语句。
- fallthrough语句不能被使用.
- 每个case关键字后必须跟随一个通道接收数据操作或者一个通道发送数据操作。 通道接收数据操作可以做为源值出现在一条简单赋值语句中。 以后，一

个case关键字后跟随的通道操作将被称为一个case操作。

- 所有的非阻塞case操作中将有一个被随机选择执行（而不是按照从上到下的顺序），然后执行此操作对应的case分支代码块。
- 在所有的case操作均为阻塞的情况下，如果default分支存在，则default分支代码块将得到执行； 否则，当前协程将被推入所有阻塞操作中相关的通道的发送数据协程队列或者接收数据协程队列中，并进入阻塞状态。

按照上述规则，一个不含任何分支的select-case代码块select{}将使当前协程处于永久阻塞状态。

在下面这个例子中，default分支将铁定得到执行，因为两个case分支后的操作均为阻塞的。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var c chan struct{} // nil
 7|    select {
 8|    case <-c:              // 阻塞操作
 9|    case c <- struct{}{}: // 阻塞操作
10|    default:
11|       fmt.Println("Go here.")
12|    }
13| }
```

下面这个例子中实现了尝试发送（try-send）和尝试接收（try-receive）。 它们都是用含有一个case分支和一个default分支的select-case代码块来实现的。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    c := make(chan string, 2)
 7|    trySend := func(v string) {
 8|       select {
 9|       case c <- v:
10|       default: // 如果c的缓冲已满，则执行默认分支。
11|       }
12|    }
13|    tryReceive := func() string {
14|       select {
```

```
15|        case v := <-c: return v
16|        default: return "-" // 如果c的缓冲为空，则执行默认分支。
17|        }
18|    }
19|    trySend("Hello!") // 发送成功
20|    trySend("Hi!")    // 发送成功
21|    trySend("Bye!")   // 发送失败，但不会阻塞。
22|    // 下面这两行将接收成功。
23|    fmt.Println(tryReceive()) // Hello!
24|    fmt.Println(tryReceive()) // Hi!
25|    // 下面这行将接收失败。
26|    fmt.Println(tryReceive()) // -
27| }
```

下面这个程序有50%的几率会因为恐慌而崩溃。 此程序中select-case代码块中的两个case操作均不阻塞，所以随机一个将被执行。 如果第一个case操作（向已关闭的通道发送数据）被执行，则一个恐慌将产生。

```
 1| package main
 2|
 3| func main() {
 4|    c := make(chan struct{})
 5|    close(c)
 6|    select {
 7|    case c <- struct{}{}: // 若此分支被选中，则产生一个恐慌
 8|    case <-c:
 9|    }
10| }
```

select-case流程控制的实现机理

select-case流程控制是Go中的一个重要和独特的特性。 下面列出了官方标准运行时中select-case流程控制的实现步骤（https://github.com/golang/go/blob/master/src/runtime/select.go）。

1. 将所有case操作中涉及到的通道表达式和发送值表达式按照从上到下，从左到右的顺序一一估值。 在赋值语句中做为源值的数据接收操作对应的目标值在此时刻不需要被估值。
2. 将所有分支随机排序。default分支总是排在最后。 所有case操作中相关的通道可能会有重复的。
3. 为了防止在下一步中造成（和其它协程互相）死锁，对所有case操作中相关的通道进行排序。 排序依据并不重要，官方Go标准编译器使用通道的地址顺

序进行排序。 排序结果中前N个通道不存在重复的情况。 N为所有case操作中涉及到的不重复的通道的数量。 下面，***通道锁顺序***是针对此排序结果中的前N个通道来说的，***通道锁逆序***是指此顺序的逆序。

4. 按照上一步中的生成通道锁顺序获取所有相关的通道的锁。
5. 按照第*2*步中生成的分支顺序检查相应分支：
    1. 如果这是一个case分支并且相应的通道操作是一个向关闭了的通道发送数据操作，则按照通道锁逆序解锁所有的通道并在当前协程中产生一个恐慌。 跳到第*12*步。
    2. 如果这是一个case分支并且相应的通道操作是非阻塞的，则按照通道锁逆序解锁所有的通道并执行相应的case分支代码块。 （此相应的通道操作可能会唤醒另一个处于阻塞状态的协程。） 跳到第*12*步。
    3. 如果这是default分支，则按照通道锁逆序解锁所有的通道并执行此default分支代码块。 跳到第*12*步。

    （到这里，default分支肯定是不存在的，并且所有的case操作均为阻塞的。）
6. 将当前协程（和对应case分支信息）推入到每个case操作中对应的通道的发送数据协程队列或接收数据协程队列中。 当前协程可能会被多次推入到同一个通道的这两个队列中，因为多个case操作中对应的通道可能为同一个。
7. 使当前协程进入阻塞状态并且按照通道锁逆序解锁所有的通道。
8. ...，当前协程处于阻塞状态，等待其它协程通过通道操作唤醒当前协程，...
9. 当前协程被另一个协程中的一个通道操作唤醒。 此唤醒通道操作可能是一个通道关闭操作，也可能是一个数据发送/接收操作。 如果它是一个数据发送/接收操作，则（当前正被解释的select-case流程中）肯定有一个相应case操作与之配合传递数据。 在此配合过程中，当前协程将从相应case操作相关的通道的接收/发送数据协程队列中弹出。
10. 按照第*3*步中的生成的通道锁顺序获取所有相关的通道的锁。
11. 将当前协程从各个case操作中对应的通道的发送数据协程队列或接收数据协程队列中（可能以非弹出的方式）移除。
    1. 如果当前协程是被一个通道关闭操作所唤醒，则跳到第*5*步。
    2. 如果当前协程是被一个数据发送/接收操作所唤醒，则相应的case分支已经在第*9*步中知晓。 按照通道锁逆序解锁所有的通道并执行此case分支代码块。
12. 完毕。

从此实现中，我们得知

- 一个协程可能同时多次处于同一个通道的发送数据协程队列或接收数据协程队列中。
- 当一个协程被阻塞在一个select-case流程控制中并在以后被唤醒时，它可能会从多个通道的发送数据协程队列和接收数据协程队列中被移除。

更多

我们可以在通道用例大全（第37章）一文中找到更多通道的使用例子。

尽管通道可以帮助我们轻松地写出正确的并发代码（第38章），和其它并发同步技术一样，通道并不会阻止我们写出不正确的并发代码（第42章）。

通道并非在任何场合总是最佳的并发同步方案，请阅读其它并发同步技术（第39章）和原子操作（第40章）来了解Go中支持的更多的并发同步技术。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

方法

Go支持一些面向对象编程特性，方法是这些所支持的特性之一。 本篇文章将介绍在Go中和方法相关的各种概念。

方法声明

在Go中，我们可以为类型T和*T显式地声明一个方法，其中类型T必须满足四个条件：

1. T必须是一个定义类型（第14章）；
2. T必须和此方法声明定义在同一个代码包中；
3. T不能是一个指针类型；
4. T不能是一个接口类型。接口类型将在下一篇文章（第23章）中讲解。

类型T和*T称为它们各自的方法的属主类型（receiver type）。 类型T被称作为类型T和*T声明的所有方法的属主基类型（receiver base type）。

注意：我们也可以为满足上列条件的类型T和*T的别名（第14章）声明方法。 这样做的效果和直接为类型T和*T声明方法是一样的。

如果我们为某个类型声明了一个方法，以后我们可以说此类型拥有此方法。

从上面列出的条件，我们得知我们不能为下列类型（显式地）声明方法：

- 内置基本类型。比如int和string。 因为这些类型声明在内置builtin标准包中，而我们不能在标准包中声明方法。
- 接口类型。但是接口类型可以拥有方法。详见下一篇文章（第23章）。
- 除了满足上面条件的形如*T的指针类型之外的无名组合类型。

一个方法声明和一个函数声明很相似，但是比函数声明多了一个额外的参数声明部分。 此额外的参数声明部分只能含有一个类型为此方法的属主类型的参数，此参数称为此方法声明的属主参数（receiver parameter）。 此属主参数声明必须包裹在一对小括号()之中。 此属主参数声明部分必须处于func关键字和方法名之间。

下面是一个方法声明的例子：

```
1| // Age和int是两个不同的类型。我们不能为int和*int
2| // 类型声明方法，但是可以为Age和*Age类型声明方法。
3| type Age int
4| func (age Age) LargerThan(a Age) bool {
```

```
 5|    return age > a
 6| }
 7| func (age *Age) Increase() {
 8|    *age++
 9| }
10|
11| // 为自定义的函数类型FilterFunc声明方法。
12| type FilterFunc func(in int) bool
13| func (ff FilterFunc) Filte(in int) bool {
14|    return ff(in)
15| }
16|
17| // 为自定义的映射类型StringSet声明方法。
18| type StringSet map[string]struct{}
19| func (ss StringSet) Has(key string) bool {
20|    _, present := ss[key]
21|    return present
22| }
23| func (ss StringSet) Add(key string) {
24|    ss[key] = struct{}{}
25| }
26| func (ss StringSet) Remove(key string) {
27|    delete(ss, key)
28| }
29|
30| // 为自定义的结构体类型Book和它的指针类型*Book声明方法。
31|
32| type Book struct {
33|    pages int
34| }
35|
36| func (b Book) Pages() int {
37|    return b.pages
38| }
39|
40| func (b *Book) SetPages(pages int) {
41|    b.pages = pages
42| }
```

从上面的例子可以看出，我们可以为各种种类（kind）的类型声明方法，而不仅仅是结构体类型。

在很多其它面向对象的编程语言中，属主参数名总是为隐式声明的`this`或者`self`。这样的名称不推荐在Go编程中使用。

指针类型的属主参数称为***指针类型属主***，非指针类型的属主参数称为***值类型属主***。在大多数情况下，我个人非常反对将***指针***和***值***这两个术语用做对立面，但是在这里，我并不反对这么用，原因将在下面谈及。

方法名可以是空标识符_。一个类型可以拥有若干名可以是空标识符的方法，但是这些方法无法被调用。 只有导出的方法才可以在其它代码包中调用。 方法调用将在后面的一节中介绍。

每个方法对应着一个隐式声明的函数

对每个方法声明，编译器将自动隐式声明一个相对应的函数。 比如对于上一节的例子中为类型`Book`和`*Book`声明的两个方法，编译器将自动声明下面的两个函数：

```
 1| func Book.Pages(b Book) int {
 2|    return b.pages // 此函数体和Book类型的Pages方法体一样
 3| }
 4|
 5| func (*Book).SetPages(b *Book, pages int) {
 6|    b.pages = pages // 此函数体和*Book类型的SetPages方法体一样
 7| }
```

在上面的两个隐式函数声明中，它们各自对应的方法声明的属主参数声明被插入到了普通参数声明的第一位。 它们的函数体和各自对应的显式方法的方法体是一样的。

两个隐式函数名`Book.Pages`和`(*Book).SetPages`都是`aType.MethodName`这种形式的。 我们不能显式声明名称为这种形式的函数，因为这种形式中的函数名不属于合法标识符。这样的函数只能由编译器隐式声明。 但是我们可以在代码中调用这些隐式声明的函数：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Book struct {
 6|    pages int
 7| }
 8|
 9| func (b Book) Pages() int {
10|    return b.pages
11| }
12|
13| func (b *Book) SetPages(pages int) {
14|    b.pages = pages
```

```
15| }
16|
17| func main() {
18|    var book Book
19|    // 调用这两个隐式声明的函数。
20|    (*Book).SetPages(&book, 123)
21|    fmt.Println(Book.Pages(book)) // 123
22| }
```

事实上，在隐式声明上述两个函数的同时，编译器也将改写这两个函数对应的显式方法（至少，我们可以这样认为），让这两个方法在体内直接调用这两个隐式函数：

```
 1| func (b Book) Pages() int {
 2|    return Book.Pages(b)
 3| }
 4|
 5| func (b *Book) SetPages(pages int) {
 6|    (*Book).SetPages(b, pages)
 7| }
```

为指针类型属主隐式声明的方法

对每一个为值类型属主`T`声明的方法，一个相应的同名方法将自动隐式地为其对应的指针类型属主`*T`而声明。 以上面的为类型`Book`声明的`Pages`方法为例，一个同名方法将自动为类型`*Book`而声明：

```
 1| // 注意：这不是合法的Go语法。这里这样表示只是
 2| // 为了解释目的。它表明表达式(&aBook).Pages
 3| // 将被估值为aBook.Pages（见随后几节）。
 4| func (b *Book) Pages = (*b).Pages
```

正因为如此，我并不排斥使用值类型属主这个术语做为指针类型属主这个术语的对立面。 毕竟，当我们为一个非指针类型显式声明一个方法的时候，事实上两个方法被声明了。 一个方法是为非指针类型显式声明的，另一个是为指针类型隐式声明的。

上一节已经提到了，每一个方法对应着一个编译器隐式声明的函数。 所以对于刚提到的隐式方法，编译器也将隐式声明一个相应的函数：

```
 1| func (*Book).Pages(b *Book) int {
 2|    return Book.Pages(*b)
 3| }
```

换句话说，对于每一个为值类型属主显式声明的方法，同时将有一个隐式方法和两个隐式函数被自动声明。

方法描述（method specification）和方法集（method set）

一个方法描述可以看作是一个不带 func 关键字的函数原型（第20章）。 我们可以把每个方法声明看作是由一个 func 关键字、一个属主参数声明部分、一个方法描述和一个方法体组成。

比如，上面的例子中的 Pages 和 SetPages 的描述如下：

```
1| Pages() int
2| SetPages(pages int)
```

每个类型都有个方法集。一个非接口类型的方法集由所有为它声明的（不管是显式的还是隐式的，但不包含方法名为空标识符的）方法的描述组成。 接口类型将在下一篇文章（第23章）详述。

比如，在上面的例子中， Book 类型的方法集为：

```
1| Pages() int
```

而 *Book 类型的方法集为：

```
1| Pages() int
2| SetPages(pages int)
```

方法集中的方法描述的次序并不重要。

对于一个方法集，如果其中的每个方法描述都处于另一个方法集中，则我们说前者方法集为后者（即另一个）方法集的子集，后者为前者的超集。 如果两个方法集互为子集（或超集），则这两个方法集必等价。

给定一个类型 T，假设它既不是一个指针类型也不是一个接口类型，因为上一节中提到的原因，类型 T 的方法集总是类型 *T 的方法集的子集。 比如，在上面的例子中， Book 类型的方法集为 *Book 类型的方法集的子集。

请注意：**不同代码包中的同名非导出方法将总被认为是不同名的。**

方法集在Go中的多态特性中扮演着重要的角色。多态将在下一篇文章（第23章）中讲解。

下列类型的方法集总为空：

- 内置基本类型；
- 定义的指针类型；
- 基类型为指针类型或者接口类型的指针类型；
- 无名数组/切片/映射/函数/通道类型。

方法值和方法调用

方法事实上是特殊的函数。方法也常被称为成员函数。 当一个类型拥有一个方法，则此类型的每个值将拥有一个不可修改的函数类型的成员（类似于结构体的字段）。 此成员的名称为此方法名，它的类型和此方法的声明中不包括属主部分的函数声明的类型一致。 一个值的成员函数也可以称为此值的方法。

一个方法调用其实是调用了一个值的成员函数。假设一个值`v`有一个名为`m`的方法，则此方法可以用选择器语法形式`v.m`来表示。

下面这个例子展示了如何调用为`Book`和`*Book`类型声明的方法：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Book struct {
 6|    pages int
 7| }
 8|
 9| func (b Book) Pages() int {
10|    return b.pages
11| }
12|
13| func (b *Book) SetPages(pages int) {
14|    b.pages = pages
15| }
16|
17| func main() {
18|    var book Book
19|
20|    fmt.Printf("%T \n", book.Pages)       // func() int
21|    fmt.Printf("%T \n", (&book).SetPages) // func(int)
22|    // &book值有一个隐式方法Pages。
23|    fmt.Printf("%T \n", (&book).Pages)    // func() int
24|
25|    // 调用这三个方法。
26|    (&book).SetPages(123)
27|    book.SetPages(123)           // 等价于上一行
```

```
28|    fmt.Println(book.Pages())    // 123
29|    fmt.Println((&book).Pages()) // 123
30| }
```

（和C语言不同，Go中没有->操作符用来通过指针属主值来调用方法。(&book)->SetPages(123)在Go中是非法的。）

等一下，上例中的`(&book).SetPages(123)`一行为什么可以被简化为`book.SetPages(123)`呢？ 毕竟，类型`Book`并不拥有一个`SetPages`方法。 啊哈，这可以看作是Go中为了让代码看上去更简洁而特别设计的语法糖。此语法糖只对可寻址的值类型的属主有效。 编译器会隐式地将`book.SetPages(123)`改写为`(&book).SetPages(123)`。 但另一方面，我们应该总是认为`aBookExpression.SetPages`是一个合法的选择器（从语法层面讲），即使表达式`aBookExpression`被估值为一个不可寻址的`Book`值（在这种情况下，`aBookExpression.SetPages`是一个无效但合法的选择器）。

如上面刚提到的，当为一个类型声明了一个方法后，每个此类型的值将拥有一个和此方法同名的成员函数。 此类型的零值也不例外，不论此类型的零值是否用`nil`来表示。

一个例子：

```
 1| package main
 2|
 3| type StringSet map[string]struct{}
 4| func (ss StringSet) Has(key string) bool {
 5|    _, present := ss[key] // 永不会产生恐慌，即使ss为nil。
 6|    return present
 7| }
 8|
 9| type Age int
10| func (age *Age) IsNil() bool {
11|    return age == nil
12| }
13| func (age *Age) Increase() {
14|    *age++ // 如果age是一个空指针，则此行将产生一个恐慌。
15| }
16|
17| func main() {
18|    _ = (StringSet(nil)).Has   // 不会产生恐慌
19|    _ = ((*Age)(nil)).IsNil    // 不会产生恐慌
20|    _ = ((*Age)(nil)).Increase // 不会产生恐慌
21|
22|    _ = (StringSet(nil)).Has("key") // 不会产生恐慌
```

```
23|    _ = ((*Age)(nil)).IsNil()       // 不会产生恐慌
24|
25|    // 下面这行将产生一个恐慌，但是此恐慌不是在调用方法的时
26|    // 候产生的，而是在此方法体内解引用空指针的时候产生的。
27|    ((*Age)(nil)).Increase()
28| }
```

属主参数的传参是一个值复制过程

和普通参数传参一样，属主参数的传参也是一个值复制过程。 所以，在方法体内对属主参数的直接部分（第17章）的修改将不会反映到方法体外。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Book struct {
 6|    pages int
 7| }
 8|
 9| func (b Book) SetPages(pages int) {
10|    b.pages = pages
11| }
12|
13| func main() {
14|    var b Book
15|    b.SetPages(123)
16|    fmt.Println(b.pages) // 0
17| }
```

另一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Book struct {
 6|    pages int
 7| }
 8|
 9| type Books []Book
10|
```

```
11| func (books Books) Modify() {
12|     // 对属主参数的间接部分的修改将反映到方法之外。
13|     books[0].pages = 500
14|     // 对属主参数的直接部分的修改不会反映到方法之外。
15|     books = append(books, Book{789})
16| }
17|
18| func main() {
19|     var books = Books{{123}, {456}}
20|     books.Modify()
21|     fmt.Println(books) // [{500} {456}]
22| }
```

有点题外话，如果将上例中`Modify`方法中的两行代码次序调换，那么此方法中的两处修改都不能反映到此方法之外。

```
 1| func (books Books) Modify() {
 2|     books = append(books, Book{789})
 3|     books[0].pages = 500
 4| }
 5|
 6| func main() {
 7|     var books = Books{{123}, {456}}
 8|     books.Modify()
 9|     fmt.Println(books) // [{123} {456}]
10| }
```

这两处修改都不能反映到`Modify`方法之外的原因是`append`函数调用将开辟一块新的内存来存储它返回的结果切片的元素。 而此结果切片的前两个元素是属主参数切片的元素的副本。对此副本所做的修改不会反映到`Modify`方法之外。

为了将此两处修改反映到`Modify`方法之外， `Modify`方法的属主类型应该改为指针类型：

```
 1| func (books *Books) Modify() {
 2|     *books = append(*books, Book{789})
 3|     (*books)[0].pages = 500
 4| }
 5|
 6| func main() {
 7|     var books = Books{{123}, {456}}
 8|     books.Modify()
 9|     fmt.Println(books) // [{500} {456} {789}]
10| }
```

方法值的正规化

在编译阶段，编译器将正规化各个方法值表达式。简而言之，正规化就是将方法值表达式中的隐式取地址和解引用操作均转换为显式操作。

假设值`v`的类型为`T`，并且`v.m`是一个合法的方法值表达式，

- 如果`m`是一个为类型`*T`显式声明的方法，那么编译器将把它正规化`(&v).m`；
- 如果`m`是一个为类型`T`显式声明的方法，那么`v.m`已经是一个正规化的方法值表达式。

假设值`p`的类型为`*T`，并且`p.m`是一个合法的方法值表达式，

- 如果`m`是一个为类型`T`显式声明的方法，那么编译器将把它正规化`(*p).m`；
- 如果`m`是一个为类型`*T`显式声明的方法，那么`p.m`已经是一个正规化的方法值表达式。

提升方法值的正规化将在随后的类型内嵌（第24章）一文中解释。

方法值的估值

假设`v.m`是一个已经正规化的方法值表达式，在运行时刻，当`v.m`被估值的时候，属主实参`v`的估值结果的一个副本将被存储下来以供后面调用此方法值的时候使用。

以下面的代码为例：

- `b.Pages`是一个已经正规化的方法值表达式。 在运行时刻对其进行估值时，属主实参`b`的一个副本将被存储下来。 此副本等于`b`的当前值：`Book{pages: 123}`，此后对`b`值的修改不影响此副本值。 这就是为什么调用`f1()`打印出`123`。
- 在编译时刻，方法值表达式`p.Pages`将被正规化为`(*p).Pages`。 在运行时刻，属主实参`*p`被估值为当前的`b`值，也就是`Book{pages: 123}`。 这就是为什么调用`f2()`也打印出`123`。
- `p.Pages2`是一个已经正规化的方法值表达式。 在运行时刻对其进行估值时，属主实参`p`的一个副本将被存储下来，此副本的值为`b`值的地址。 当`b`被修改后，此修改可以通过对此地址值解引用而反映出来，这就是为什么调用`g1()`打印出`789`。
- 在编译时刻，方法值表达式`b.Pages2`将被正规化为`(&b).Pages2`。 在运行时刻，属主实参`&b`的估值结果的一个副本将被存储下来，此副本的值为`b`值的地址。 这就是为什么调用`g2()`也打印出`789`。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Book struct {
 6|    pages int
 7| }
 8|
 9| func (b Book) Pages() int {
10|    return b.pages
11| }
12|
13| func (b *Book) Pages2() int {
14|    return (*b).Pages()
15| }
16|
17| func main() {
18|    var b = Book{pages: 123}
19|    var p = &b
20|    var f1 = b.Pages
21|    var f2 = p.Pages
22|    var g1 = p.Pages2
23|    var g2 = b.Pages2
24|    b.pages = 789
25|    fmt.Println(f1()) // 123
26|    fmt.Println(f2()) // 123
27|    fmt.Println(g1()) // 789
28|    fmt.Println(g2()) // 789
29| }
```

一个定义类型不会获取为它的源类型显式声明的方法

举个例子，在下面的代码中，定义类型`Age`并不像它的源类型`MyInt`一样拥有一个`IsOdd`方法。

```
 1| package main
 2|
 3| type MyInt int
 4| func (mi MyInt) IsOdd() bool {
 5|    return mi%2 == 1
 6| }
 7|
```

```
 8| type Age MyInt
 9|
10| func main() {
11|    var x MyInt = 3
12|    _ = x.IsOdd() // okay
13|
14|    var y Age = 36
15|    // _ = y.IsOdd() // error: y.IsOdd undefined
16|    _ = y
17| }
```

如何决定一个方法声明使用值类型属主还是指针类型属主?

首先，从上一节中的例子，我们可以得知有时候我们必须在某些方法声明中使用指针类型属主。

事实上，我们总可以在方法声明中使用指针类型属主而不会产生任何逻辑问题。 我们仅仅是为了程序效率考虑有时候才会在函数声明中使用值类型属主。

对于值类型属主还是指针类型属主都可以接受的方法声明，下面列出了一些考虑因素：

- 太多的指针可能会增加垃圾回收器的负担。
- 如果一个值类型的尺寸太大，那么属主参数在传参的时候的复制成本将不可忽略。 指针类型都是小尺寸类型。 关于各种不同类型的尺寸，请阅读值复制代价（第34章）一文。
- 在并发场合下，同时调用值类型属主和指针类型属主方法比较易于产生数据竞争。
- sync 标准库包中的类型的值不应该被复制，所以如果一个结构体类型内嵌（第24章）了这些类型，则不应该为这个结构体类型声明值类型属主的方法。

如果实在拿不定主意在一个方法声明中应该使用值类型属主还是指针类型属主，那么请使用指针类型属主。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

接口

接口类型是Go中的一种很特别的类型。接口类型在Go中扮演着重要的角色。 首先，在Go中，接口值可以用来包裹非接口值；然后，通过值包裹，反射和多态得以实现。

自从1.18版本开始，Go已经支持自定义泛型。 在自定义泛型中，接口类型总可以被用做类型约束。 事实上，所有的类型约束都是接口类型。 在Go 1.18版本之前，所有的接口类型均可用做值类型。 但是从Go 1.18版本开始，有些接口类型只能被用做类型约束。 可被用做值类型的接口类型称为基本接口类型。

本文大体是在Go支持自定义泛型之前写成的，所以本文主要讲述基本接口类型。 关于非基本接口类型（只能用做类型约束的接口类型），请阅读《Go自定义泛型101》（https://gfw.go101.org/generics/101.html）一书以了解详情。

接口类型介绍和类型集（Type Set）

一个接口类型定义了一些类型条件。 所有满足了全部这些条件的非接口类型形成了一个类型集合。 此类型集合称为此接口类型的类型集。

接口类型是通过内嵌若干接口元素来定义类型条件的。 目前（Go 1.23）支持两种接口元素：方法元素和类型元素。

- 一个方法元素呈现为一个方法描述（第22章）（method specification）。内嵌在接口类型中的方法描述不能使用空标识符_命名。
- 一个类型元素可以是一个类型名称、一个类型字面表示形式、一个近似类型或者一个类型并集。 本文不过多介绍后两者。对于前两者，也只谈及当它们表示接口类型的情况。

举个例子，预声明的error接口类型（https://golang.org/pkg/builtin/#error）的定义如下。 它内嵌了一个方法描述Error() string。 在此定义中，interface{...}称为接口类型的字面表示形式，其中interface为一个关键字。

```
1| type error interface {
2|         Error() string
3| }
```

我们可以说此error接口类型（直接）指定了一个方法（描述）：Error() string。 它的类型集由所有拥有此同样描述的方法（第22章）的非接口类型组成。 理论上，此类型集是一个无限集。当然对于一个具体的Go项目，此集合是有限的。

下面是一些其它接口类型定义和别名声明。

```
// 此接口直接指定了两个方法和内嵌了两个其它接口。
// 其中一个为类型名称，另一个为类型字面表示形式。
type ReadWriteCloser = interface {
    Read(buf []byte) (n int, err error)
    Write(buf []byte) (n int, err error)
    error                        // 一个类型名称
    interface{ Close() error } // 一个类型字面表示形式
}

// 此接口类型内嵌了一个近似类型。
// 它的类型集由所有底层类型为[]byte的类型组成。
type AnyByteSlice = interface {
    ~[]byte
}

// 此接口类型内嵌了一个类型并集。它的类型集包含6个类型：
// uint、uint8、uint16、uint32、uint64和uintptr。
type Unsigned interface {
    uint | uint8 | uint16 | uint32 | uint64 | uintptr
}
```

将一个接口类型（无论呈现为类型名称还是类型字面表示形式）内嵌到另一个接口类型中等价于将前者中的元素（递归）展开放入后者。 比如，别名`ReadWriteCloser`表示的接口类型等价于下面这个类型字面表示形式表示的直接指定了4个方法的接口类型。

```
interface {
    Read(buf []byte) (n int, err error)
    Write(buf []byte) (n int, err error)
    Error() string
    Close() error
}
```

上面这个接口类型（即别名`ReadWriteCloser`表示的接口类型）的类型集由所有拥有全部这4个指定方法的非接口类型组成。 从理论上，这也是一个无限集。它肯定是`error`接口类型的类型集的子集。

请注意：在Go 1.18之前，只有接口类型名称可以内嵌在接口类型中。

下面的代码片段中展示的接口类型都称为空接口类型。它们什么也没有内嵌。

```
// 一个无名空接口类型。
interface{}
```

```
3|
4| // Nothing是一个定义空接口类型。
5| type Nothing interface{}
```

事实上，Go 1.18引入了一个预声明的类型别名`any`，用来表示空接口类型`interface{}`。

一个空接口类型的类型集由所有由非接口类型组成。

类型的方法集

每个类型有一个方法集。

- 对于一个非接口类型，它的方法集由为此类型（无论显式还是隐式）声明 所有方法（第22章）的方法描述组成。
- 对于一个接口类型，它的方法集由此接口类型（无论直接还是间接）指定的所有方法描述组成。

对于上一节中提到的接口类型，

- 别名`ReadWriteCloser`表示的接口类型的方法集包含4个方法（描述）。
- 预声明的`error`接口类型的方法集包含一个方法（描述）。
- 一个空接口类型的方法集为空。

为了方便起见，一个类型的方法集常常也称为此方法的任何一个值的方法集。

基本接口类型

基本接口类型是指可以用做值类型的接口类型。 一个非基本接口类型只能为用做（自定义泛型中使用的）约束接口类型（即类型约束）。

目前（Go 1.23），每一个基本接口类型都可以使用一个方法集来完全定义。 换句话说，一个基本接口类型不需要内嵌任何类型元素。

在上上一节中的例子中，别名`ReadWriteCloser`表示的接口类型为一个基本接口类型， 但是`Unsigned`接口类型和别名`AnyByteSlice`表示的接口类型均不是基本接口类型。 后两者均只能用做约束接口类型。

空接口类型和预声明的`error`接口类型也都是基本接口类型。

如果两个无名基本接口类型的方法集是相同的，则这两个类型肯定为同一个类型。但是请注意：不同代码包中的同名非导出方法名将总被认为是不同名的。

类型实现（implementation）

如果一个非接口类型处于一个接口类型的类型集中，则我们说此非接口类型实现了此接口类型。 如果一个接口类型的类型集是另一个接口类型的类型集的子集，则我们说前者实现了后者。

因为一个类型集的总是它自己的子集，一个接口类型总是实现了它自己。 类似地，如果两个接口类型的类型集相同，则它们相互实现了对方。 事实上，两个拥有相同类型集的无名接口类型为同一个接口类型。

如果一个（接口或者非接口）类型`T`实现了一个接口类型`X`，那么类型`T`的方法集肯定是接口类型`X`的方法集的超集。 一般说来，反之并不成立。但是如果`X`是一个基本接口类型，则反之也成立。 比如，在前面的例子中，别名`ReadWriteCloser`表示的接口类型实现了预声明的`error`接口类型。

在Go中，实现关系是隐式的。 两个类型之间的实现关系不需要在代码中显式地表示出来。 Go中没有类似于`implements`的关键字。 Go编译器将自动在需要的时候检查两个类型之间的实现关系。

比如，在下面的例子中，类型`*Book`、`MyInt`和`*MyInt`都拥有一个描述为`About() string`的方法，所以它们都实现了接口类型`Aboutable`。

```
 1| type Aboutable interface {
 2|    About() string
 3| }
 4|
 5| type Book struct {
 6|    name string
 7|    // 更多其它字段……
 8| }
 9|
10| func (book *Book) About() string {
11|    return "Book: " + book.name
12| }
13|
14| type MyInt int
15|
16| func (MyInt) About() string {
17|    return "我是一个自定义整数类型"
18| }
```

隐式实现关系的设计使得一个声明在另一个代码包（包括标准库包）中的类型可以被动地实现一个用户代码包中的接口类型。 比如，如果我们声明一个像下面这样的

接口类型，则database/sql（https://golang.google.cn/pkg/database/sql/）标准库包中声明的DB和Tx类型都实现了这个接口类型，因为它们都拥有此接口类型指定的三个方法。

```
 1| import "database/sql"
 2|
 3| ...
 4|
 5| type DatabaseStorer interface {
 6|     Exec(query string, args ...interface{}) (sql.Result,
error)
 7|     Prepare(query string) (*sql.Stmt, error)
 8|     Query(query string, args ...interface{}) (*sql.Rows,
error)
 9| }
```

注意：因为空接口类型的类型集包含了所有的非接口类型，所以所有类型均实现了空接口类型。 这是Go中的一个重要事实。

值包裹

重申一遍：目前（Go 1.23），接口值的类型必须为一个基本接口类型。 在本文余下的内容里，当一个值类型被提及，此值类型可能是一个非接口类型，也可能是一个基本接口类型，但它肯定不是一个非基本接口类型。

每个接口值都可以看作是一个用来包裹一个非接口值的盒子。 欲将一个非接口值包裹在一个接口值中，此非接口值的类型必须实现了此接口值的类型。

在Go中，如果类型T实现了一个（基本）接口类型I，则类型T的值都可以隐式转换到类型I。 换句话说，类型T的值可以赋给类型I的可修改值。 当一个T值被转换到类型I（或者赋给一个I值）的时候，

- 如果类型T是一个非接口类型，则此T值的一个复制将被包裹在结果（或者目标）I值中。 此操作的时间复杂度为$O(n)$，其中n为T值的尺寸。
- 如果类型T也为一个接口类型，则此T值中当前包裹的（非接口）值将被复制一份到结果（或者目标）I值中。 官方标准编译器为此操作做了优化，使得此操作的时间复杂度为$O(1)$，而不是$O(n)$。

包裹在一个接口值中的非接口值的类型信息也和此非接口值一起被包裹在此接口值中（见下面详解）。

当一个非接口值被包裹在一个接口值中，此非接口值称为此接口值的***动态值***，此非接口值的类型称为此接口值的***动态类型***。

接口值的动态值的直接部分是不可修改的，除非它的动态值被整体替换为另一个动态值。

接口类型的零值也用预声明的`nil`标识符来表示。 一个nil接口值中什么也没包裹。将一个接口值修改为nil将清空包裹在此接口值中的非接口值。

（注意，在Go中，很多其它非接口类型的零值也使用`nil`标识符来表示。 非接口类型的nil零值也可以被包裹在接口值中。 一个包裹了一个nil非接口值的接口值不是一个nil接口值，因为它并非什么都没包裹。）

因为任何类型都实现了空接口类型，所以任何非接口值都可以被包裹在任何一个空接口类型的接口值中。 （以后，一个空接口类型的接口值将被称为一个空接口值。注意空接口值和nil接口值是两个不同的概念。） 因为这个原因，空接口值可以被认为是很多其它语言中的`any`类型。

当一个类型不确定值（除了类型不确定的`nil`）被转换为一个空接口类型（或者赋给一个空接口值），此类型不确定值将首先转换为它的默认类型。 （或者说，此类型不确定值将被推断为一个它的默认类型的类型确定值。）

下面这个例子展示了一些目标值为接口类型的赋值。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Aboutable interface {
 6|    About() string
 7| }
 8|
 9| // 类型*Book实现了接口类型Aboutable。
10| type Book struct {
11|    name string
12| }
13| func (book *Book) About() string {
14|    return "Book: " + book.name
15| }
16|
17| func main() {
18|    // 一个*Book值被包裹在了一个Aboutable值中。
19|    var a Aboutable = &Book{"Go语言101"}
20|    fmt.Println(a) // &{Go语言101}
21|
22|    // i是一个空接口值。类型*Book实现了任何空接口类型。
23|    var i interface{} = &Book{"Rust 101"}
24|    fmt.Println(i) // &{Rust 101}
```

```
25|
26|    // Aboutable实现了空接口类型interface{}。
27|    i = a
28|    fmt.Println(i) // &{Go语言101}
29| }
```

请注意，在之前的文章中多次使用过的`fmt.Println`函数的原型为：

```
func Println(a ...interface{}) (n int, err error)
```

这解释了为什么任何类型的实参都可以使用在`fmt.Println`函数调用中。

下面是另一个展示了一个空接口类型的值包裹着各种非接口值的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var i interface{}
 7|    i = []int{1, 2, 3}
 8|    fmt.Println(i) // [1 2 3]
 9|    i = map[string]int{"Go": 2012}
10|    fmt.Println(i) // map[Go:2012]
11|    i = true
12|    fmt.Println(i) // true
13|    i = 1
14|    fmt.Println(i) // 1
15|    i = "abc"
16|    fmt.Println(i) // abc
17|
18|    // 将接口值i中包裹的值清除掉。
19|    i = nil
20|    fmt.Println(i) // <nil>
21| }
```

在编译时刻，Go编译器将构建一个全局表用来存储代码中要用到的各个类型的信息。 对于一个类型来说，这些信息包括：此类型的种类（kind）（第14章）、此类型的所有方法和字段信息、此类型的尺寸，等等。 这个全局表将在程序启动的时候被加载到内存中。

在运行时刻，当一个非接口值被包裹到一个接口值，Go运行时（至少对于官方标准运行时来说）将分析和构建这两个值的类型的实现关系信息，并将此实现关系信息存入到此接口值内。 对每一对这样的类型，它们的实现关系信息将仅被最多构建一次。并且为了程序效率考虑，此实现关系信息将被缓存在内存中的一个全局映射

中，以备后用。 所以此全局映射中的条目数永不减少。 事实上，一个非零接口值在内部只是使用一个指针字段来引用着此全局映射中的一个实现关系信息条目（第17章）。

对于一个非接口类型和接口类型对，它们的实现关系信息包括两部分的内容：

1. 动态类型（即此非接口类型）的信息。
2. 一个方法表（切片类型），其中存储了所有此接口类型指定的并且为此非接口类型（动态类型）声明的方法。

这两部分的内容对于实现Go中的两个特性起着至关重要的作用。

1. 动态类型信息是实现反射（#reflection）的关键。
2. 方法表是实现多态（见下一节）的关键。

多态（polymorphism）

多态是接口的一个关键功能和Go语言的一个重要特性。

当非接口类型`T`的一个值`t`被包裹在接口类型`I`的一个接口值`i`中，通过`i`调用接口类型`I`指定的一个方法时，事实上为非接口类型`T`声明的对应方法将通过非接口值`t`被调用。 换句话说，**调用一个接口值的方法实际上将调用此接口值的动态值的对应方法**。 比如，当方法`i.m`被调用时，其实被调用的是方法`t.m`。 一个接口值可以通过包裹不同动态类型的动态值来表现出各种不同的行为，这称为多态。

当方法`i.m`被调用时，`i`存储的实现关系信息的方法表中的方法`t.m`将被找到并被调用。 此方法表是一个切片，所以此寻找过程只不过是一个切片元素访问操作，不会消耗很多时间。

注意，在nil接口值上调用方法将产生一个恐慌，因为没有具体的方法可被调用。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Filter interface {
 6|    About() string
 7|    Process([]int) []int
 8| }
 9|
10| // UniqueFilter用来删除重复的数字。
11| type UniqueFilter struct{}
12| func (UniqueFilter) About() string {
```

```
13|    return "删除重复的数字"
14| }
15| func (UniqueFilter) Process(inputs []int) []int {
16|    outs := make([]int, 0, len(inputs))
17|    pusheds := make(map[int]bool)
18|    for _, n := range inputs {
19|       if !pusheds[n] {
20|          pusheds[n] = true
21|          outs = append(outs, n)
22|       }
23|    }
24|    return outs
25| }
26|
27| // MultipleFilter用来只保留某个整数的倍数数字。
28| type MultipleFilter int
29| func (mf MultipleFilter) About() string {
30|    return fmt.Sprintf("保留%v的倍数", mf)
31| }
32| func (mf MultipleFilter) Process(inputs []int) []int {
33|    var outs = make([]int, 0, len(inputs))
34|    for _, n := range inputs {
35|       if n % int(mf) == 0 {
36|          outs = append(outs, n)
37|       }
38|    }
39|    return outs
40| }
41|
42| // 在多态特性的帮助下，只需要一个filteAndPrint函数。
43| func filteAndPrint(fltr Filter, unfiltered []int) []int {
44|    // 在fltr参数上调用方法其实是调用fltr的动态值的方法。
45|    filtered := fltr.Process(unfiltered)
46|    fmt.Println(fltr.About() + ":\n\t", filtered)
47|    return filtered
48| }
49|
50| func main() {
51|    numbers := []int{12, 7, 21, 12, 12, 26, 25, 21, 30}
52|    fmt.Println("过滤之前：\n\t", numbers)
53|
54|    // 三个非接口值被包裹在一个Filter切片
55|    // 的三个接口元素中。
56|    filters := []Filter{
57|       UniqueFilter{},
```

```
58|        MultipleFilter(2),
59|        MultipleFilter(3),
60|    }
61|
62|    // 每个切片元素将被赋值给类型为Filter的
63|    // 循环变量fltr。每个元素中的动态值也将
64|    // 被同时复制并被包裹在循环变量fltr中。
65|    for _, fltr := range filters {
66|        numbers = filteAndPrint(fltr, numbers)
67|    }
68| }
```

输出结果：

```
过滤之前：
    [12 7 21 12 12 26 25 21 30]
删除重复的数字：
    [12 7 21 26 25 30]
保留2的倍数：
    [12 26 30]
保留3的倍数：
    [12 30]
```

在上面这个例子中，多态使得我们不必为每个过滤器类型写一个单独的`filteAndPrint`函数。

除了上述这个好处，多态也使得一个代码包的开发者可以在此代码包中声明一个接口类型并声明一个拥有此接口类型参数的函数（或者方法），从而此代码包的一个用户可以在用户包中声明一个实现了此接口类型的用户类型，并且将此用户类型的值做为实参传递给此代码包中声明的函数（或者方法）的调用。 此代码包的开发者并不用关心一个用户类型具体是如何声明的，只要此用户类型满足此代码包中声明的接口类型规定的行为即可。

事实上，多态对于一个语言来说并非一个不可或缺的特性。我们可以通过其它途径来实现多态的作用。 但是，多态可以使得我们的代码更加简洁和优雅。

反射（reflection）

一个接口值中存储的动态类型信息可以被用来检视此接口值的动态值和操纵此动态值所引用的值。 这称为反射。

本篇文章将不介绍`reflect`标准包中提供的各种反射功能。 请阅读后面的[Go中的反射](https://golang.google.cn/pkg/reflect/)（https://golang.google.cn/pkg/reflect/）一文来了解如何使用此包。 本文

下面将只介绍Go中的内置反射机制。在Go中，内置反射机制包括类型断言（type assertion）和`type-switch`流程控制代码块。

类型断言

Go中有四种接口相关的类型转换情形：

1. 将一个非接口值转换为一个接口类型。在这样的转换中，此非接口值的类型必须实现了此接口类型。
2. 将一个接口值转换为另一个接口类型（前者接口值的类型实现了后者目标接口类型）。
3. 将一个接口值转换为一个非接口类型（此非接口类型必须实现了此接口值的接口类型）。
4. 将一个接口值转换为另一个接口类型（前者接口值的类型未实现后者目标接口类型，但是前者的动态类型有可能实现了目标接口类型）。

前两种情形已经在前面几节中介绍过了。这两种情形都要求源值的类型必须实现了目标接口类型。 它们都是通过普通类型转换（无论是隐式的还是显式的）来达成的。这两种情形的合法性是在编译时刻验证的。

本节将介绍后两种情形。这两种情形的合法性是在运行时刻通过类型断言来验证的。 事实上，类型断言也适用于上面列出的第二种情形。

一个类型断言表达式的语法为`i.(T)`，其中`i`为一个接口值，`T`为一个类型名或者类型字面表示。 类型`T`可以为

- 任意一个非接口类型。
- 或者一个任意接口类型。

在一个类型断言表达式`i.(T)`中，`i`称为断言值，`T`称为断言类型。 一个断言可能成功或者失败。

- 对于`T`是一个非接口类型的情况，如果断言值`i`的动态类型存在并且此动态类型和`T`为同一类型，则此断言将成功；否则，此断言失败。 当此断言成功时，此类型断言表达式的估值结果为断言值`i`的动态值的一个复制。 我们可以把此种情况看作是一次拆封动态值的尝试。
- 对于`T`是一个接口类型的情况，当断言值`i`的动态类型存在并且此动态类型实现了接口类型`T`，则此断言将成功；否则，此断言失败。 当此断言成功时，此类型断言表达式的估值结果为一个包裹了断言值`i`的动态值的一个复制的`T`值。

一个失败的类型断言的估值结果为断言类型的零值。

按照上述规则，如果一个类型断言中的断言值是一个零值nil接口值，则此断言必定失败。

对于大多数场合，一个类型断言被用做一个单值表达式。 但是，当一个类型断言被用做一个赋值语句中的唯一源值时，此断言可以返回一个可选的第二个结果并被视作为一个多值表达式。 此可选的第二个结果为一个类型不确定的布尔值，用来表示此断言是否成功了。

注意：如果一个断言失败并且它的可选的第二个结果未呈现，则此断言将造成一个恐慌。

一个展示了如何使用类型断言的例子（断言类型为非接口类型）：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    // 编译器将把123的类型推断为内置类型int。
 7|    var x interface{} = 123
 8|
 9|    // 情形一：
10|    n, ok := x.(int)
11|    fmt.Println(n, ok) // 123 true
12|    n = x.(int)
13|    fmt.Println(n) // 123
14|
15|    // 情形二：
16|    a, ok := x.(float64)
17|    fmt.Println(a, ok) // 0 false
18|
19|    // 情形三：
20|    a = x.(float64) // 将产生一个恐慌
21| }
```

另一个展示了如何使用类型断言的例子（断言类型为接口类型）：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type Writer interface {
 6|    Write(buf []byte) (int, error)
 7| }
 8|
 9| type DummyWriter struct{}
```

```
10| func (DummyWriter) Write(buf []byte) (int, error) {
11|    return len(buf), nil
12| }
13|
14| func main() {
15|    var x interface{} = DummyWriter{}
16|    // y的动态类型为内置类型string。
17|    var y interface{} = "abc"
18|    var w Writer
19|    var ok bool
20|
21|    // DummyWriter既实现了Writer，也实现了interface{}。
22|    w, ok = x.(Writer)
23|    fmt.Println(w, ok) // {} true
24|    x, ok = w.(interface{})
25|    fmt.Println(x, ok) // {} true
26|
27|    // y的动态类型为string。string类型并没有实现Writer。
28|    w, ok = y.(Writer)
29|    fmt.Println(w, ok) // <nil> false
30|    w = y.(Writer)     // 将产生一个恐慌
31| }
```

事实上，对于一个动态类型为T的接口值i，方法调用i.m(...)等价于i.(T).m(...)。

type-switch流程控制代码块

type-switch流程控制的语法或许是Go中最古怪的语法。 它可以被看作是类型断言的增强版。它和switch-case流程控制代码块有些相似。 一个type-switch流程控制代码块的语法如下所示：

```
 1| switch aSimpleStatement; v := x.(type) {
 2| case TypeA:
 3|    ...
 4| case TypeB, TypeC:
 5|    ...
 6| case nil:
 7|    ...
 8| default:
 9|    ...
10| }
```

其中aSimpleStatement;部分是可选的。 aSimpleStatement必须是一个简单语句（第11章）。 x必须为一个估值结果为接口值的表达式，它称为此代码块中的断言值。 v称为此代码块中的断言结果，它必须出现在一个短变量声明形式中。

在一个type-switch代码块中，每个case关键字后可以跟随预声明的nil标识符或者一个由逗号分割的若干个类型名和类型字面表示形式组成的类型列表。 在同一个type-switch代码块中，这些跟随在所有case关键字后的条目不可重复。

如果跟随在某个case关键字后的条目为一个非接口类型（用一个类型名或类型字面表示），则此非接口类型必须实现了断言值x的（接口）类型。

下面是一个使用了type-switch代码块的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    values := []interface{}{
 7|       456, "abc", true, 0.33, int32(789),
 8|       []int{1, 2, 3}, map[int]bool{}, nil,
 9|    }
10|    for _, x := range values {
11|       // 这里，虽然变量v只被声明了一次，但是它在
12|       // 不同分支中可以表现为多个类型的变量值。
13|       switch v := x.(type) {
14|       case []int: // 一个类型字面表示
15|          // 在此分支中，v的类型为[]int。
16|          fmt.Println("int slice:", v)
17|       case string: // 一个类型名
18|          // 在此分支中，v的类型为string。
19|          fmt.Println("string:", v)
20|       case int, float64, int32: // 多个类型名
21|          // 在此分支中，v的类型为x的类型interface{}。
22|          fmt.Println("number:", v)
23|       case nil:
24|          // 在此分支中，v的类型为x的类型interface{}。
25|          fmt.Println(v)
26|       default:
27|          // 在此分支中，v的类型为x的类型interface{}。
28|          fmt.Println("others:", v)
29|       }
30|       // 注意：在最后的三个分支中，v均为接口值x的一个复制。
```

```
31|    }
32| }
```

输出结果：

```
number: 456
string: abc
others: true
number: 0.33
number: 789
int slice: [1 2 3]
others: map[]
<nil>
```

上面这个例子程序在逻辑上等价于下面这个：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    values := []interface{}{
 7|       456, "abc", true, 0.33, int32(789),
 8|       []int{1, 2, 3}, map[int]bool{}, nil,
 9|    }
10|    for _, x := range values {
11|       if v, ok := x.([]int); ok {
12|          fmt.Println("int slice:", v)
13|       } else if v, ok := x.(string); ok {
14|          fmt.Println("string:", v)
15|       } else if x == nil {
16|          v := x
17|          fmt.Println(v)
18|       } else {
19|          _, isInt := x.(int)
20|          _, isFloat64 := x.(float64)
21|          _, isInt32 := x.(int32)
22|          if isInt || isFloat64 || isInt32 {
23|             v := x
24|             fmt.Println("number:", v)
25|          } else {
26|             v := x
27|             fmt.Println("others:", v)
28|          }
29|       }
```

```
30|     }
31| }
```

如果我们不关心一个type-switch代码块中的断言结果，则此type-switch代码块可简写为switch x.(type) {...}。

type-switch代码块和switch-case代码块有很多共同点：

- 在一个type-switch代码块中，最多只能有一个default分支存在。
- 在一个type-switch代码块中，如果default分支存在，它可以为最后一个分支、第一个分支或者中间的某个分支。
- 一个type-switch代码块可以不包含任何分支，它可以被视为一个空操作。

但是，和switch-case代码块不一样， fallthrough语句不能使用在type-switch代码块中。

更多接口相关的内容

接口值相关的比较

接口值相关的比较有两种情形：

1. 比较一个非接口值和接口值；
2. 比较两个接口值。

对于第一种情形，非接口值的类型必须实现了接口值的类型（假设此接口类型为I），所以此非接口值可以被隐式转化为（包裹到）一个I值中。 这意味着非接口值和接口值的比较可以转化为两个接口值的比较。所以下面我们只探讨两个接口值比较的情形。

比较两个接口值其实是比较这两个接口值的动态类型和和动态值。

下面是（使用==比较运算符）比较两个接口值的步骤：

1. 如果其中一个接口值是一个nil接口值，则比较结果为另一个接口值是否也为一个nil接口值。
2. 如果这两个接口值的动态类型不一样，则比较结果为false。
3. 对于这两个接口值的动态类型一样的情形，
    - 如果它们的动态类型为一个不可比较类型（第48章），则将产生一个恐慌。
    - 否则，比较结果为它们的动态值的比较结果。

简而言之，两个接口值的比较结果只有在下面两种任一情况下才为true：

1. 这两个接口值都为nil接口值。
2. 这两个接口值的动态类型相同、动态类型为可比较类型、并且动态值相等。

根据此规则，两个包裹了不同非接口类型的nil零值的接口值是不相等的。 一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var a, b, c interface{} = "abc", 123, "a"+"b"+"c"
 7|    fmt.Println(a == b) // 第二步的情形。输出"false"。
 8|    fmt.Println(a == c) // 第三步的情形。输出"true"。
 9|
10|    var x *int = nil
11|    var y *bool = nil
12|    var ix, iy interface{} = x, y
13|    var i interface{} = nil
14|    fmt.Println(ix == iy) // 第二步的情形。输出"false"。
15|    fmt.Println(ix == i)  // 第一步的情形。输出"false"。
16|    fmt.Println(iy == i)  // 第一步的情形。输出"false"。
17|
18|    var s []int = nil // []int为一个不可比较类型。
19|    i = s
20|    fmt.Println(i == nil) // 第一步的情形。输出"false"。
21|    fmt.Println(i == i)   // 第三步的情形。将产生一个恐慌。
22| }
```

接口值的内部结构

对于标准编译器/运行时，空接口值和非空接口值是使用两种内部结构来表示的。详情请阅读值部（第17章）一文。

指针动态值和非指针动态值

标准编译器/运行时对接口值的动态值为指针类型的情况做了特别的优化。 此优化使得接口值包裹指针动态值比包裹非指针动态值的效率更高。 对于小尺寸值（第34章），此优化的作用不大； 但是对于大尺寸值，包裹它的指针比包裹此值本身的效率高得多。 对于类型断言，此结论亦成立。

所以尽量避免在接口值中包裹大尺寸值。对于大尺寸值，应该尽量包裹它的指针。

一个[]T类型的值不能直接被转换为类型[]I，即使类型T实现了接口类型I

比如，我们不能直接将一个[]string值转换为类型[]interface{}。 我们必须使用一个循环来实现此转换：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    words := []string{
 7|       "Go", "is", "a", "high",
 8|       "efficient", "language.",
 9|    }
10|
11|    // fmt.Println函数的原型为：
12|    // func Println(a ...interface{}) (n int, err error)
13|    // 所以words...不能传递给此函数的调用。
14|
15|    // fmt.Println(words...) // 编译不通过
16|
17|    // 将[]string值转换为类型[]interface{}。
18|    iw := make([]interface{}, 0, len(words))
19|    for _, w := range words {
20|       iw = append(iw, w)
21|    }
22|    fmt.Println(iw...) // 编译没问题
23| }
```

一个接口类型每个指定的每一个方法都对应着一个隐式声明的函数

如果接口类型I指定了一个名为m的方法描述，则编译器将隐式声明一个与之对应的函数名为I.m的函数。 此函数比m的方法描述中的参数多一个。此多出来的参数为函数I.m的第一个参数，它的类型为I。 对于一个类型为I的值i，方法调用i.m(...)和函数调用I.m(i, ...)是等价的。

一个例子：

```
 1| package main
 2|
```

```
 3| import "fmt"
 4|
 5| type I interface {
 6|    m(int)bool
 7| }
 8|
 9| type T string
10| func (t T) m(n int) bool {
11|    return len(t) > n
12| }
13|
14| func main() {
15|    var i I = T("gopher")
16|    fmt.Println(i.m(5))                          // true
17|    fmt.Println(I.m(i, 5))                       // true
18|    fmt.Println(interface {m(int) bool}.m(i, 5)) // true
19|
20|    // 下面这几行被执行的时候都将会产生一个恐慌。
21|    I(nil).m(5)
22|    I.m(nil, 5)
23|    interface {m(int) bool}.m(nil, 5)
24| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

类型内嵌

从结构体（第16章）一文中，我们得知一个结构体类型可以拥有若干字段。 每个字段由一个字段名和一个字段类型组成。事实上，有时，一个字段可以仅由一个字段类型组成。 这样的字段声明方式称为类型内嵌（type embedding）。

此篇文章将解释类型内嵌的目的和各种和类型内嵌相关的细节。

类型内嵌语法

下面是一个使用了类型内嵌的例子：

```
 1| package main
 2|
 3| import "net/http"
 4|
 5| func main() {
 6|     type P = *bool
 7|     type M = map[int]int
 8|     var x struct {
 9|         string // 一个具名非指针类型
10|         error  // 一个具名接口类型
11|         *int   // 一个无名指针类型
12|         P      // 一个无名指针类型的别名
13|         M      // 一个无名类型的别名
14|
15|         http.Header // 一个具名映射类型
16|     }
17|     x.string = "Go"
18|     x.error = nil
19|     x.int = new(int)
20|     x.P = new(bool)
21|     x.M = make(M)
22|     x.Header = http.Header{}
23| }
```

在上面这个例子中，有六个类型被内嵌在了一个结构体类型中。每个类型内嵌形成了一个内嵌字段（embedded field）。

因为历史原因，内嵌字段有时也称为匿名字段。但是，事实上，每个内嵌字段有一个（隐式的）名字。 此字段的非限定（unqualified）（https://golang.google.cn/ref/spec#Qualified_identifiers）类型名即为此字

段的名称。 比如，上例中的六个内嵌字段的名称分别为`string`、`error`、`int`、`P`、`M`和`Header`。

哪些类型可以被内嵌?

当前的Go白皮书（1.23）[规定](https://golang.google.cn/ref/spec#Struct_types)（https://golang.google.cn/ref/spec#Struct_types）：

> An embedded field must be specified as a type name `T` or as a pointer to a non-interface type name `*T`, and `T` itself may not be a pointer type.

翻译过来：

> 一个内嵌字段必须被声明为形式`T`或者一个基类型为非接口类型的指针类型`*T`，其中`T`为一个类型名但是`T`不能表示一个指针类型。

此规则描述在Go 1.9之前是精确的。但是随着从Go 1.9引入的自定义类型别名概念，此描述[有些过时和不太准确了](https://github.com/golang/go/issues/22005)（https://github.com/golang/go/issues/22005）。 比如，此描述没有包括上一节的例子中的`P`内嵌字段的情形。

这里，本文试图使用一个更精确的描述：

- 一个类型名`T`只有在它既不表示一个具名指针类型也不表示一个基类型为指针类型或者接口类型的指针类型的情况下才可以被用做内嵌字段。
- 一个指针类型`*T`只有在`T`为一个类型名并且`T`既不表示一个指针类型也不表示一个接口类型的时候才能被用做内嵌字段。

下面列出了一些可以被或不可以被内嵌的类型或别名：

```
 1| type Encoder interface {Encode([]byte) []byte}
 2| type Person struct {name string; age int}
 3| type Alias = struct {name string; age int}
 4| type AliasPtr = *struct {name string; age int}
 5| type IntPtr *int
 6| type AliasPP = *IntPtr
 7|
 8| // 这些类型或别名都可以被内嵌。
 9| Encoder
10| Person
11| *Person
12| Alias
13| *Alias
14| AliasPtr
15| int
```

```
16| *int
17|
18| // 这些类型或别名都不能被内嵌。
19| AliasPP           // 基类型为一个指针类型
20| *Encoder          // 基类型为一个接口类型
21| *AliasPtr         // 基类型为一个指针类型
22| IntPtr            // 具名指针类型
23| *IntPtr           // 基类型为一个指针类型
24| *chan int         // 基类型为一个无名类型
25| struct {age int} // 无名非指针类型
26| map[string]int   // 无名非指针类型
27| []int64          // 无名非指针类型
28| func()           // 无名非指针类型
```

一个结构体类型中不允许有两个同名字段，此规则对匿名字段同样适用。 根据上述内嵌字段的隐含名称规则，一个无名指针类型不能和它的基类型同时内嵌在同一个结构体类型中。 比如，`int`和`*int`类型不能同时内嵌在同一个结构体类型中。

一个结构体类型不能内嵌（无论间接还是直接）它自己。

一般说来，只有内嵌含有字段或者拥有方法的类型才有意义（后续几节将阐述原因），尽管很多既没有字段也没有方法的类型也可以被内嵌。

类型内嵌的意义是什么?

类型内嵌的主要目的是为了将被内嵌类型的功能扩展到内嵌它的结构体类型中，从而我们不必再为此结构体类型重复实现被内嵌类型的功能。

很多其它流行面向对象的编程语言都是用继承来实现上述目的。两种实现方式有[它们各自的利弊](https://en.wikipedia.org/wiki/Composition_over_inheritance)（https://en.wikipedia.org/wiki/Composition_over_inheritance）。 这里，此篇文章将不讨论哪种方式更好一些，我们只需知道Go选择了类型内嵌这种方式。 这两种方式有一个很大的不同点：

- 如果类型`T`继承了另外一个类型，则类型`T`获取了另外一个类型的能力。 同时，一个`T`类型的值也可以被当作另外一个类型的值来使用。
- 如果一个类型`T`内嵌了另外一个类型，则另外一个类型变成了类型`T`的一部分。 类型`T`获取了另外一个类型的能力，但是`T`类型的任何值都不能被当作另外一个类型的值来使用。

下面是一个展示了如何通过类型内嵌来扩展类型功能的例子：

```
1| package main
2|
```

```
 3| import "fmt"
 4|
 5| type Person struct {
 6|    Name string
 7|    Age  int
 8| }
 9| func (p Person) PrintName() {
10|    fmt.Println("Name:", p.Name)
11| }
12| func (p *Person) SetAge(age int) {
13|    p.Age = age
14| }
15|
16| type Singer struct {
17|    Person // 通过内嵌Person类型来扩展之
18|    works  []string
19| }
20|
21| func main() {
22|    var gaga = Singer{Person: Person{"Gaga", 30}}
23|    gaga.PrintName() // Name: Gaga
24|    gaga.Name = "Lady Gaga"
25|    (&gaga).SetAge(31)
26|    (&gaga).PrintName()   // Name: Lady Gaga
27|    fmt.Println(gaga.Age) // 31
28| }
```

从上例中，当类型`Singer`内嵌了类型`Person`之后，看上去类型`Singer`获取了类型`Person`所有的字段和方法， 并且类型`*Singer`获取了类型`*Person`所有的方法。此结论是否正确？随后几节将给出答案。

注意，类型`Singer`的一个值不能被当作`Person`类型的值用。下面的代码编译不通过：

```
 1| var gaga = Singer{}
 2| var _ Person = gaga
```

当一个结构体类型内嵌了另一个类型，此结构体类型是否获取了被内嵌类型的字段和方法？

下面这个程序使用反射（第27章）列出了上一节的例子中的`Singer`类型的字段和方法，以及`*Singer`类型的方法。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "reflect"
 6| )
 7|
 8| ... // 为节省篇幅，上一个例子中声明的类型在这里省略了。
 9|
10| func main() {
11|    t := reflect.TypeOf(Singer{}) // the Singer type
12|    fmt.Println(t, "has", t.NumField(), "fields:")
13|    for i := 0; i < t.NumField(); i++ {
14|       fmt.Print(" field#", i, ": ", t.Field(i).Name, "\n")
15|    }
16|    fmt.Println(t, "has", t.NumMethod(), "methods:")
17|    for i := 0; i < t.NumMethod(); i++ {
18|       fmt.Print(" method#", i, ": ", t.Method(i).Name,
"\n")
19|    }
20|
21|    pt := reflect.TypeOf(&Singer{}) // the *Singer type
22|    fmt.Println(pt, "has", pt.NumMethod(), "methods:")
23|    for i := 0; i < pt.NumMethod(); i++ {
24|       fmt.Print(" method#", i, ": ", pt.Method(i).Name,
"\n")
25|    }
26| }
```

输出结果：

```
main.Singer has 2 fields:
 field#0: Person
 field#1: works
main.Singer has 1 methods:
 method#0: PrintName
*main.Singer has 2 methods:
 method#0: PrintName
 method#1: SetAge
```

从此输出结果中，我们可以看出类型`Singer`确实拥有一个`PrintName`方法，以及类型`*Singer`确实拥有两个方法：`PrintName`和`SetAge`。 但是类型`Singer`并不拥有一个`Name`字段。那么为什么选择器表达式`gaga.Name`是合法的呢？ 毕竟`gaga`是`Singer`类型的一个值。 请阅读下一节以获取原因。

选择器的缩写形式

从前面的结构体（第16章）和方法（第22章）两篇文章中，我们得知，对于一个值x，x.y称为一个选择器，其中y可以是一个字段名或者方法名。 如果y是一个字段名，那么x必须为一个结构体值或者结构体指针值。 一个选择器是一个表达式，它表示着一个值。 如果选择器x.y表示一个字段，此字段也可能拥有自己的字段（如果此字段的类型为另一个结构体类型）和方法，比如x.y.z，其中z可以是一个字段名，也可是一个方法名。

在Go中，（不考虑下面将要介绍的选择器碰撞和遮挡），***如果一个选择器中的中部某项对应着一个内嵌字段，则此项可被省略掉***。 因此内嵌字段又被称为匿名字段。

一个例子：

```
 1| package main
 2|
 3| type A struct {
 4|     x int
 5| }
 6|
 7| func (a A) MethodA() {}
 8|
 9| type B struct {
10|     *A
11| }
12|
13| type C struct {
14|     B
15| }
16|
17| func main() {
18|     var c = &C{B: B{A: &A{FieldX: 5}}}
19|
20|     // 这几行是等价的。
21|     _ = c.B.A.FieldX
22|     _ = c.B.FieldX
23|     _ = c.A.FieldX // A是类型C的一个提升字段
24|     _ = c.FieldX   // FieldX也是一个提升字段
25|
26|     // 这几行是等价的。
27|     c.B.A.MethodA()
28|     c.B.MethodA()
29|     c.A.MethodA()
```

```
30|     c.MethodA() // MethodA是类型C的一个提升方法
31| }
```

这就是为什么在上一节的例子中选择器表达式`gaga.Name`是合法的， 因为它只不过是`gaga.Person.Name`的一个缩写形式。

类似的，选择器`gaga.PrintName`可以被看作是`gaga.Person.PrintName`的缩写形式。 但是，我们也可以不把它看作是一个缩写。毕竟，类型`Singer`确实拥有一个`PrintName`方法， 尽管此方法是被隐式声明的（请阅读下下节以获得详情）。 同样的原因，选择器`(&gaga).PrintName`和`(&gaga).SetAge`可以看作（也可以不看作）是`(&gaga.Person).PrintName`和`(&gaga.Person).SetAge`的缩写。

`Name`被称为类型`Singer`的一个提升字段（promoted field）。 `PrintName`被称为类型`Singer`的一个提升方法（promoted method）。

注意：我们也可以使用选择器`gaga.SetAge`，但是只有在`gaga`是一个可寻址的类型为`Singer`的值的情况下。 它只不过是`(&gaga).SetAge`的一个语法糖（第22章）。

在上面的例子中，`c.B.A.FieldX`称为选择器表达式`c.FieldX`、`c.B.FieldX`和`c.A.FieldX`的完整形式。 类似的，`c.B.A.MethodA`可以称为`c.MethodA`、`c.B.MethodA`和`c.A.MethodA`的完整形式。

如果一个选择器的完整形式中的所有中部项均对应着一个内嵌字段，则中部项的数量称为此选择器的深度。 比如，上面的例子中的选择器`c.MethodA`的深度为*2*，因为此选择器的完整形式为`c.B.A.MethodA`，并且`B`和`A`都对应着一个内嵌字段。

选择器遮挡和碰撞

一个值`x`（这里我们总认为它是可寻址的）可能同时拥有多个最后一项相同的选择器，并且这些选择器的中间项均对应着一个内嵌字段。 对于这种情形（假设最后一项为`y`）：

- 只有深度最浅的一个完整形式的选择器（并且最浅者只有一个）可以被缩写为`x.y`。 换句话说，`x.y`表示深度最浅的一个选择器。其它完整形式的选择器被此最浅者所遮挡（压制）。
- 如果有多个完整形式的选择器同时拥有最浅深度，则任何完整形式的选择器都不能被缩写为`x.y`。 我们称这些同时拥有最浅深度的完整形式的选择器发生了碰撞。

如果一个方法选择器被另一个方法选择器所遮挡，并且它们对应的方法描述是一致的，那么我们可以说第一个方法被第二个覆盖（overridden）了。

举个例子，假设A、B和C为三个定义类型（第14章）：

```
 1| type A struct {
 2|    x string
 3| }
 4| func (A) y(int) bool {
 5|    return false
 6| }
 7|
 8| type B struct {
 9|    y bool
10| }
11| func (B) x(string) {}
12|
13| type C struct {
14|    B
15| }
```

下面这段代码编译不通过，原因是选择器v1.A.x和v1.B.x的深度一样，所以它们发生了碰撞，结果导致它们都不能被缩写为v1.x。 同样的情况发生在选择器v1.A.y和v1.B.y身上。

```
 1| var v1 struct {
 2|    A
 3|    B
 4| }
 5|
 6| func f1() {
 7|    _ = v1.x // error: 模棱两可的v1.x
 8|    _ = v1.y // error: 模棱两可的v1.y
 9| }
```

下面的代码编译没问题。选择器v2.C.B.x被另一个选择器v2.A.x遮挡了，所以v2.x实际上是选择器v2.A.x的缩写形式。 因为同样的原因，v2.y是选择器v2.A.y（而不是选择器v2.C.B.y）的缩写形式。

```
 1| var v2 struct {
 2|    A
 3|    C
 4| }
 5|
 6| func f2() {
 7|    fmt.Printf("%T \n", v2.x) // string
```

```
 8|    fmt.Printf("%T \n", v2.y) // func(int) bool
 9| }
```

一个被遮挡或者碰撞的选择器并不妨碍更深层的选择器被提升，如下例所示中的.M和.z：

```
 1| package main
 2|
 3| type x string
 4| func (x) M() {}
 5|
 6| type y struct {
 7|    z byte
 8| }
 9|
10| type A struct {
11|    x
12| }
13| func (A) y(int) bool {
14|    return false
15| }
16|
17| type B struct {
18|    y
19| }
20| func (B) x(string) {}
21|
22| func main() {
23|    var v struct {
24|       A
25|       B
26|    }
27|    //_ = v.x // error: 模棱两可的v.x
28|    //_ = v.y // error: 模棱两可的v.y
29|    _ = v.M // ok. <=> v.A.x.M
30|    _ = v.z // ok. <=> v.B.y.z
31| }
```

一个不寻常的但需要注意的细节是：来自不同库包的两个非导出方法（或者字段）将总是被认为是两个不同的标识符，即使它们的名字完全一致。 因此，当它们的属主类型被同时内嵌在同一个结构体类型中的时候，它们绝对不会相互碰撞或者遮挡。 举个例子，下面这个含有两个库包的Go程序编译和运行都没问题。 但是，如果将其中所有出现的m()改为M()，则此程序将编译不过。 原因是A.M和B.M碰撞了，导致c.M为一个非法的选择器。

```
 1| package foo // import "x.y/foo"
 2|
 3| import "fmt"
 4|
 5| type A struct {
 6|    n int
 7| }
 8|
 9| func (a A) m() {
10|    fmt.Println("A", a.n)
11| }
12|
13| type I interface {
14|    m()
15| }
16|
17| func Bar(i I) {
18|    i.m()
19| }
```

```
 1| package main
 2|
 3| import "fmt"
 4| import "x.y/foo"
 5|
 6| type B struct {
 7|    n bool
 8| }
 9|
10| func (b B) m() {
11|    fmt.Println("B", b.n)
12| }
13|
14| type C struct{
15|    foo.A
16|    B
17| }
18|
19| func main() {
20|    var c C
21|    c.m()      // B false
22|    foo.Bar(c) // A 0
23| }
```

为内嵌了其它类型的结构体类型声明的隐式方法

上面已经提到过，类型`Singer`和`*Singer`都有一个`PrintName`方法，并且类型`*Singer`还有一个`SetAge`方法。 但是，我们从没有为这两个类型声明过这几个方法。这几个方法从哪来的呢?

事实上，假设结构体类型`S`内嵌了一个类型（或者类型别名）`T`，并且此内嵌是合法的，

- 对内嵌类型`T`的每一个方法，如果此方法对应的选择器既不和其它选择器碰撞也未被其它选择器遮挡，则编译器将会隐式地为结构体类型`S`声明一个同样描述的方法。 继而，编译器也将为指针类型`*S`隐式声明（第22章）一个相应的方法。
- 对类型`*T`的每一个方法，如果此方法对应的选择器既不和其它选择器碰撞也未被其它选择器遮挡，则编译器将会隐式地为类型`*S`声明一个同样描述的方法。

简单说来，

- 类型`struct{T}`和`*struct{T}`均将获取类型`T`的所有方法。
- 类型`*struct{T}`、`struct{*T}`和`*struct{*T}`都将获取类型`*T`的所有方法。

下面展示了编译器为类型`Singer`和`*Singer`隐式声明的三个（提升）方法：

```
1| // 注意：这些声明不是合法的Go语法。这里这样表示只是为了
2| // 解释目的。它们有助于解释提升方法值是如何被估值的。
3| func (s Singer) PrintName = s.Person.PrintName
4| func (s *Singer) PrintName = s.Person.PrintName
5| func (s *Singer) SetAge = s.Person.SetAge
```

右边的部分为各个提升方法相应的完整形式选择器形式。

从方法（第22章）一文中，我们得知我们不能为无名的结构体类型（和基类型为无名结构体类型的指针类型）声明方法。 但是，通过类型内嵌，这样的类型也可以拥有方法。

如果一个结构体类型内嵌了一个实现了一个接口类型的类型（此内嵌类型可以是此接口类型自己），则一般说来，此结构体类型也实现了此接口类型，除非发生了选择器碰撞和遮挡。 比如，上例中的结构体类型和以它为基类型的指针类型均实现了接口类型`I`。

请注意：一个类型将只会获取它（直接或者间接）内嵌了的类型的方法。 换句话说，一个类型的方法集由为类型直接（显式或者隐式）声明的方法和此类型的底层类型的方法集组成。 比如，在下面的例子中，

- 类型`Age`没有方法，因为代码中既没有为它声明任何方法，它也没有内嵌任何类型，。
- 类型`X`有两个方法：`IsOdd`和`Double`。 其中`IsOdd`方法是通过内嵌类型`MyInt`而得来的。
- 类型`Y`没有方法，因为它所内嵌的类型`Age`没有方法，另外代码中也没有为它声明任何方法。
- 类型`Z`只有一个方法：`IsOdd`。 此方法是通过内嵌类型`MyInt`而得来的。 它没有获取到类型`X`的`Double`方法，因为它并没有内嵌类型`X`。

```
 1| type MyInt int
 2| func (mi MyInt) IsOdd() bool {
 3|    return mi%2 == 1
 4| }
 5|
 6| type Age MyInt
 7|
 8| type X struct {
 9|    MyInt
10| }
11| func (x X) Double() MyInt {
12|    return x.MyInt + x.MyInt
13| }
14|
15| type Y struct {
16|    Age
17| }
18|
19| type Z X
```

提升方法值的正规化和估值

假设`v.m`是一个合法的提升方法表达式，在编译时刻，编译器将把此提升方法表达式正规化。 正规化过程分为两步：首先找出此提升方法表达式的完整形式；然后将此完整形式中的隐式取地址和解引用操作均转换为显式操作。

和其它方法值估值（第22章）的规则一样，对于一个已经正规化的方法值表达式`v.m`，在运行时刻，当`v.m`被估值的时候，属主实参`v`的估值结果的一个副本将被存储下来以供后面调用此方法值的时候使用。

以下面的代码为例：

- 提升方法表达式`s.M1`的完整形式为`s.T.X.M1`。 将此完整形式中的隐式取地址和解引用操作转换为显式操作之后的结果为`(*s.T).X.M1`。 在运行时刻，属主实参`(*s.T).X`被估值并且估值结果的一个副本被存储下来以供后用。 此估值结果为`1`，这就是为什么调用`f()`总是打印出`1`。
- 提升方法表达式`s.M2`的完整形式为`s.T.X.M2`。 将此完整形式中的隐式取地址和解引用操作转换为显式操作之后的结果为`(&(*s.T).X).M2`。 在运行时刻，属主实参`&(*s.T).X`被估值并且估值结果的一个副本被存储下来以供后用。 此估值结果为提升字段`s.X`（也就是`(*s.T).X`）的地址。 任何对`s.X`的修改都可以通过解引用此地址而反映出来，但是对`s.T`的修改是不会通过此地址反映出来的。 这就是为什么两个`g()`调用都打印出了`2`。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type X int
 6|
 7| func (x X) M1() {
 8|    fmt.Println(x)
 9| }
10|
11| func (x *X) M2() {
12|    fmt.Println(*x)
13| }
14|
15| type T struct { X }
16|
17| type S struct { *T }
18|
19| func main() {
20|    var t = &T{X: 1}
21|    var s = S{T: t}
22|    var f = s.M1 // <=> (*s.T).X.M1
23|    var g = s.M2 // <=> (&(*s.T).X).M2
24|    s.X = 2
25|    f() // 1
26|    g() // 2
27|    s.T = &T{X: 3}
28|    f() // 1
29|    g() // 2
30| }
```

接口类型内嵌各种类型

接口类型可以内嵌各种类型。详情请阅读接口（第23章）一文。

一个有趣的类型内嵌的例子

在本文的最后，让我们来看一个有趣的例子。 此例子程序将陷入死循环并会因堆栈溢出而崩溃退出。 如果你已经理解了多态（第23章）和类型内嵌，那么就不难理解为什么此程序将死循环。

```
 1| package main
 2|
 3| type I interface {
 4|     m()
 5| }
 6|
 7| type T struct {
 8|     I
 9| }
10|
11| func main() {
12|     var t T
13|     var i = &t
14|     t.I = i
15|     i.m() // 将调用t.m()，然后再次调用i.m()，......
16| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

非类型安全指针

我们已经从Go中的指针（第15章）一文中学习到关于指针的各种概念和规则。 从那篇文章中，我们得知，相对于C指针，Go指针有很多限制。 比如，Go指针不支持算术运算，并且对于任意两个指针值，很可能它们不能转换到对方的类型。

事实上，在那篇文章中解释的指针的完整称呼应该为类型安全指针。 虽然类型安全指针有助于我们轻松写出安全的代码，但是有时候施加在类型安全指针上的限制也确实导致我们不能写出最高效的代码。

实际上，Go也支持限制较少的非类型安全指针。 非类型安全指针和C指针类似，它们都很强大，但同时也都很危险。 在某些情形下，通过非类型安全指针的帮助，我们可以写出效率更高的代码； 但另一方面，使用非类型安全指针也导致我们可能轻易地写出潜在的不安全的代码，这些潜在的不安全点很难在它们产生危害之前被及时发现。

使用非类型安全指针的另外一个较大的风险是Go中目前提供的非类型安全指针机制并不受到Go 1 兼容性保证（https://golang.google.cn/doc/golcompat）的保护。使用了非类型安全指针的代码可能从今后的某个Go版本开始将不再能编译通过，或者运行行为发生了变化。

如果出于种种原因，你确实希望在你的代码中使用非类型安全指针，你不仅需要提防上述风险，你还需遵守Go官方文档中列出的非类型安全指针使用模式，并清楚地知晓使用非类型安全指针带来的效果。否则，你很难使用非类型安全指针写出安全的代码。

关于unsafe标准库包

非类型安全指针在Go中为一种（第14章）特别的类型。 我们必须引入unsafe标准库包（https://golang.google.cn/pkg/unsafe/）来使用非类型安全指针。 非类型安全指针unsafe.Pointer被声明定义为：

```
type Pointer *ArbitraryType
```

当然，这不是一个普通的类型定义。这里的ArbitraryType仅仅是暗示unsafe.Pointer类型值可以被转换为任意类型安全指针（反之亦然）。换句话说，unsafe.Pointer类似于C语言中的void*。

非类型安全指针是指底层类型为unsafe.Pointer的类型。

非类型安全指针的零值也使用预声明的nil标识符来表示。

在Go 1.17之前，unsafe标准库包只提供了三个函数：

- func Alignof(variable ArbitraryType) uintptr。 此函数用来取得一个值在内存中的地址对齐保证（address alignment guarantee）。 注意，同一个类型的值做为结构体字段和非结构体字段时地址对齐保证可能是不同的。 当然，这和具体编译器的实现有关。对于目前的标准编译器，同一个类型的值做为结构体字段和非结构体字段时的地址对齐保证总是相同的。 gccgo编译器对这两种情形是区别对待的。
- func Offsetof(selector ArbitraryType) uintptr。 此函数用来取得一个结构体值的某个字段的地址相对于此结构体值的地址的偏移。 在一个程序中，对于同一个结构体类型的不同值的对应相同字段，此函数的返回值总是相同的。
- func Sizeof(variable ArbitraryType) uintptr。 此函数用来取得一个值的尺寸（亦即此值的类型的尺寸）。 在一个程序中，对于同一个类型的不同值，此函数的返回值总是相同的。

注意：

- 这三个函数的返回值的类型均为内置类型uintptr。下面我们将了解到uintptr类型的值可以转换为非类型安全指针（反之亦然）。
- 尽管这三个函数之一的任何调用的返回结果在同一个编译好的程序中总是一致的，但是这样的一个调用在不同架构的操作系统中（或者使用不同的编译器编译时）的返回值可能是不一样的。
- 这三个函数的调用总是在编译时刻被估值，估值结果为类型为uintptr的常量。
- 传递给Offsetof函数的实参必须为一个字段选择器形式value.field。 此选择器可以表示一个内嵌字段，但此选择器的路径中不能包含指针类型的隐式字段（第24章）。

一个使用了这三个函数的例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "unsafe"
 5|
 6| func main() {
 7|    var x struct {
 8|       a int64
 9|       b bool
10|       c string
11|    }
12|    const M, N = unsafe.Sizeof(x.c), unsafe.Sizeof(x)
13|    fmt.Println(M, N) // 16 32
```

```
14|
15|    fmt.Println(unsafe.Alignof(x.a)) // 8
16|    fmt.Println(unsafe.Alignof(x.b)) // 1
17|    fmt.Println(unsafe.Alignof(x.c)) // 8
18|
19|    fmt.Println(unsafe.Offsetof(x.a)) // 0
20|    fmt.Println(unsafe.Offsetof(x.b)) // 8
21|    fmt.Println(unsafe.Offsetof(x.c)) // 16
22| }
```

下面是一个展示了上面提到的最后一个注意点的例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "unsafe"
 5|
 6| func main() {
 7|    type T struct {
 8|       c string
 9|    }
10|    type S struct {
11|       b bool
12|    }
13|    var x struct {
14|       a int64
15|       *S
16|       T
17|    }
18|
19|    fmt.Println(unsafe.Offsetof(x.a)) // 0
20|
21|    fmt.Println(unsafe.Offsetof(x.S)) // 8
22|    fmt.Println(unsafe.Offsetof(x.T)) // 16
23|
24|    // 此行可以编译过，因为选择器x.c中的隐含字段T为非指针。
25|    fmt.Println(unsafe.Offsetof(x.c)) // 16
26|
27|    // 此行编译不过，因为选择器x.b中的隐含字段S为指针。
28|    //fmt.Println(unsafe.Offsetof(x.b)) // error
29|
30|    // 此行可以编译过，但是它将打印出字段b在x.S中的偏移量.
31|    fmt.Println(unsafe.Offsetof(x.S.b)) // 0
32| }
```

注意，上面程序中的注释所暗示的输出结果是此程序在AMD64架构上使用标准编译器1.23版本编译时的结果。

unsafe包提供的这三个函数看上去并不怎么危险。 它们的原型在以后的Go 1版本中几乎不可能会发生改变（https://groups.google.com/g/golang-nuts/c/Pi7VveWDUPc）。 Rob Pike甚至曾经将这几个函数挪到其它包中（https://github.com/golang/go/issues/5602）。 unsafe包的危险性基本上来自于非类型安全指针。它们和C指针一样危险，这是Go安全指针千方百计设法去避免的。

Go 1.17引入了一个新类型和两个新函数。 此新类型为IntegerType。它的定义如下。 此类型不代表着一个具体类型，它只是表示任意整数类型（有点泛型的意思）。

```
type IntegerType int
```

Go 1.17引入的两个函数为：

- func Add(ptr Pointer, len IntegerType) Pointer。 此函数在一个（非安全）指针表示的地址上添加一个偏移量，然后返回表示新地址的一个指针。 此函数以一种更正规的形式部分地覆盖了下面将要介绍的使用模式3中展示的合法用法。
- func Slice(ptr *ArbitraryType, len IntegerType) []ArbitraryType。 此函数用来从一个任意（安全）指针派生出一个指定长度的切片。

Go 1.20进一步引入了三个函数：

- func String(ptr *byte, len IntegerType) string。 此函数用来从一个任意（安全）byte指针派生出一个指定长度的字符串。
- func StringData(str string) *byte。 此函数用来获取一个字符串底层字节序列中的第一个byte的指针。
- func SliceData(slice []ArbitraryType) *ArbitraryType。 此函数用来获取一个切片底层元素序列中的第一个元素的指针。

Go 1.17之后引入的这些函数具有一定的危险性，需谨慎使用。 下面是使用了Go 1.17引入的两个函数的一个例子。

```
1| package main
2|
3| import (
4|     "fmt"
5|     "unsafe"
6| )
```

```
 7|
 8| func main() {
 9|    a := [16]int{3: 3, 9: 9, 11: 11}
10|    fmt.Println(a)
11|    eleSize := int(unsafe.Sizeof(a[0]))
12|    p9 := &a[9]
13|    up9 := unsafe.Pointer(p9)
14|    p3 := (*int)(unsafe.Add(up9, -6 * eleSize))
15|    fmt.Println(*p3) // 3
16|    s := unsafe.Slice(p9, 5)[:3]
17|    fmt.Println(s) // [9 0 11]
18|    fmt.Println(len(s), cap(s)) // 3 5
19|
20|    t := unsafe.Slice((*int)(nil), 0)
21|    fmt.Println(t == nil) // true
22|
23|    // 下面是两个不正确的调用。因为它们
24|    // 的返回结果引用了未知的内存块。
25|    _ = unsafe.Add(up9, 7 * eleSize)
26|    _ = unsafe.Slice(p9, 8)
27| }
```

下面这两个函数使用了非类型安全的方式实现了字符串和字节切片之间的类型转换。 和类型安全方式相比，它们不用复制字符串和字节切片的底层字节序列，因此效率更高。

```
 1| import "unsafe"
 2|
 3| func String2ByteSlice(str string) []byte {
 4|    if str == "" {
 5|       return nil
 6|    }
 7|    return unsafe.Slice(unsafe.StringData(str), len(str))
 8| }
 9|
10| func ByteSlice2String(bs []byte) string {
11|    if len(bs) == 0 {
12|       return ""
13|    }
14|    return unsafe.String(unsafe.SliceData(bs), len(bs))
15| }
```

非类型安全指针相关的类型转换

目前（Go 1.23），Go支持下列和非类型安全指针相关的类型转换：

- 一个类型安全指针值可以被显式转换为一个非类型安全指针类型，反之亦然。
- 一个uintptr值可以被显式转换为一个非类型安全指针类型，反之亦然。 但是，注意，一个nil非类型安全指针类型不应该被转换为uintptr并进行算术运算后再转换回来。

通过使用这些转换规则，我们可以将任意两个类型安全指针转换为对方的类型，我们也可以将一个安全指针值和一个uintptr值转换为对方的类型。

然而，尽管这些转换在编译时刻是合法的，但是它们中一些在运行时刻并非是合法和安全的。 这些转换摧毁了Go的类型系统（不包括非类型安全指针部分）精心设立的内存安全屏障。 我们必须遵循本文后面要介绍的一些用法指示来使用非类型安全指针才能写出合法并安全的代码。

我们需要知道的一些事实

在开始介绍合法的非类型安全指针使用模式之前，我们需要知道一些事实。

事实一：非类型安全指针值是指针但uintptr值是整数

每一个非零安全或者不安全指针值均引用着另一个值。但是一个uintptr值并不引用任何值，它被看作是一个整数，尽管常常它存储的是一个地址的数字表示。

Go是一门支持垃圾回收的语言。 当一个Go程序在运行中，Go运行时（runtime）将不时地检查哪些内存块将不再被程序中的任何仍在使用中的值所引用并且回收这些内存块（第43章）。 指针在这一过程中扮演着重要的角色。值与值之间和内存块与值之间的引用关系是通过指针来表征的。

既然一个uintptr值是一个整数，那么它可以参与算术运算。

下一节中的例子将展示指针和uintptr值的不同。

事实二：不再被使用的内存块的回收时间点是不确定的

在运行时刻，一次新的垃圾回收过程可能在一个不确定的时间启动，并且此过程可能需要一段不确定的时长才能完成。 所以一个不再被使用的内存块的回收时间点（第43章）是不确定的。

一个例子：

```
1| import "unsafe"
2|
```

```
 3| // 假设此函数不会被内联（inline）。
 4| //go:noinline
 5| func createInt() *int {
 6|     return new(int)
 7| }
 8|
 9| func foo() {
10|     p0, y, z := createInt(), createInt(), createInt()
11|     var p1 = unsafe.Pointer(y) // 和y一样引用着同一个值
12|     var p2 = uintptr(unsafe.Pointer(z))
13|
14|     // 此时，即使z指针值所引用的int值的地址仍旧存储
15|     // 在p2值中，但是此int值已经不再被使用了，所以垃圾
16|     // 回收器认为可以回收它所占据的内存块了。另一方面，
17|     // p0和p1各自所引用的int值仍旧将在下面被使用。
18|
19|     // uintptr值可以参与算术运算。
20|     p2 += 2; p2--; p2--
21|
22|     *p0 = 1                             // okay
23|     *(*int)(p1) = 2                     // okay
24|     *(*int)(unsafe.Pointer(p2)) = 3 // 危险操作！
25| }
```

在上面这个例子中，值`p2`仍旧在使用这个事实并不能保证曾经被`z`指针值所引用的`int`值所占的内存块一定还没有被回收。 换句话说，当`*(*int)(unsafe.Pointer(p2)) = 3`被执行的时候，此内存块有可能已经被回收了。 所以，继续通过解引用值`p2`中存储的地址是非常危险的，因为此内存块可能已经被重新分配给其它值使用了。

事实三：一个值的地址在程序运行中可能改变

详情请阅读内存块（第43章）一文（见链接所指一节的尾部）。 这里我们只需要知道当一个协程的栈的大小改变时，开辟在此栈上的内存块需要移动，从而相应的值的地址将改变。

事实四：一个值的生命范围可能并没有代码中看上去的大

比如中下面这个例子，值`t`仍旧在使用中并不能保证被值`t.y`所引用的值仍在被使用。

```
 1| type T struct {
 2|     x int
 3|     y *[1<<23]byte
 4| }
 5|
 6| func bar() {
 7|     t := T{y: new([1<<23]byte)}
 8|     p := uintptr(unsafe.Pointer(&t.y[0]))
 9|
10|     ... // 使用t.x和t.y
11|
12|     // 一个聪明的编译器能够觉察到值t.y将不会再被用到，
13|     // 所以认为t.y值所占的内存块可以被回收了。
14|
15|     *(*byte)(unsafe.Pointer(p)) = 1 // 危险操作！
16|
17|     println(t.x) // ok。继续使用值t，但只使用t.x字段。
18| }
```

事实五：*unsafe.Pointer是一个类型安全指针类型

是的，类型*unsafe.Pointer是一个类型安全指针类型。 它的基类型为unsafe.Pointer。 既然它是一个类型安全指针类型，根据上面列出的类型转换规则，它的值可以转换为类型unsafe.Pointer，反之亦然。

一个例子：

```
 1| package main
 2|
 3| import "unsafe"
 4|
 5| func main() {
 6|     x := 123                // 类型为int
 7|     p := unsafe.Pointer(&x) // 类型为unsafe.Pointer
 8|     pp := &p                // 类型为*unsafe.Pointer
 9|     p = unsafe.Pointer(pp)
10|     pp = (*unsafe.Pointer)(p)
11| }
```

如何正确地使用非类型安全指针?

unsafe标准库包的文档中列出了六种非类型安全指针的使用模式（https://golang.google.cn/pkg/unsafe/#Pointer）。 下面将对它们逐一进行讲解。

使用模式一：将类型*T1的一个值转换为非类型安全指针值，然后将此非类型安全指针值转换为类型*T2。

利用前面列出的非类型安全指针相关的转换规则，我们可以将一个*T1值转换为类型*T2，其中T1和T2为两个任意类型。 然而，我们只有在T1的尺寸不小于T2并且此转换具有实际意义的时候才应该实施这样的转换。

通过将一个*T1值转换为类型*T2，我们也可以将一个T1值转换为类型T2。

一个这样的例子是math标准库包中的Float64bits函数。 此函数将一个float64值转换为一个uint64值。 在此转换过程中，此float64值在内存中的每个位（bit）都保持不变。 函数math.Float64frombits为此转换的逆转换。

```
 1| func Float64bits(f float64) uint64 {
 2|     return *(*uint64)(unsafe.Pointer(&f))
 3| }
 4|
 5| func Float64frombits(b uint64) float64 {
 6|     return *(*float64)(unsafe.Pointer(&b))
 7| }
```

请注意，函数调用math.Float64bits(aFloat64)的结果和显式转换uint64(aFloat64)的结果不同。

在下面这个例子中，我们使用此模式将一个[]MyString值和一个[]string值转换为对方的类型。 结果切片和被转换的切片将共享底层元素。（这样的转换是不可能通过安全的方式来实现的。）

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "unsafe"
 6| )
 7|
 8| func main() {
 9|     type MyString string
10|     ms := []MyString{"C", "C++", "Go"}
11|     fmt.Printf("%s\n", ms)  // [C C++ Go]
```

```
    // ss := ([]string)(ms) // 编译错误
    ss := *(*[]string)(unsafe.Pointer(&ms))
    ss[1] = "Zig"
    fmt.Printf("%s\n", ms) // [C Zig Go]
    // ms = []MyString(ss) // 编译错误
    ms = *(*[]MyString)(unsafe.Pointer(&ss))
}
```

顺便说一句，从Go 1.17开始，我们也可以使用`unsafe.Slice`函数来实现这样的转换：

```
func main() {
    ...

    ss = unsafe.Slice((*string)(&ms[0]), len(ms))
    ms = unsafe.Slice((*MyString)(&ss[0]), len(ss))
}
```

此模式在实践中的另一个应用是将一个不再使用的字节切片转换为一个字符串（从而避免对底层字节序列的一次开辟和复制）。如下例所示：

```
func ByteSlice2String(bs []byte) string {
    return *(*string)(unsafe.Pointer(&bs))
}
```

此实现借鉴于`strings`标准库包中的`Builder`类型的`String`方法的实现。 字节切片的尺寸比字符串的尺寸要大，并且它们的底层结构（第17章）类似，所以此转换（对于当前的主流Go编译器来说）是安全的。 即使这样，此实现也只推荐在标准库中使用，而不推荐在用户代码中使用。 从Go 1.20开始，在用户代码中，最好尽量使用本文前面介绍的使用`unsafe.String`函数的实现。

反过来，下面这个例子中的转换是非法的，因为字符串的尺寸比字节切片的尺寸小。

```
func String2ByteSlice(s string) []byte {
    return *(*[]byte)(unsafe.Pointer(&s)) // 危险
}
```

在后面的模式六中展示了一种合法的（无需复制底层字节序列即可）将一个字符串转换为字节切片的实现。

注意：当运用上面展示的使用非类型安全指针将一个字节切片转换为字符串的技巧时，请确保结果字符串在使用过程中绝对不修改此字节切片中的字节值。

使用模式二：将一个非类型安全指针值转换为一个uintptr值，然后使用此uintptr值。

此模式不是很有用。一般我们将最终的转换结果uintptr值输出到日志中用来调试，但是有很多其它安全并且简洁的途径也可以实现此目的。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "unsafe"
 5|
 6| func main() {
 7|    type T struct{a int}
 8|    var t T
 9|    fmt.Printf("%p\n", &t)                          //
0xc6233120a8
10|    println(&t)                                     //
0xc6233120a8
11|    fmt.Printf("%x\n", uintptr(unsafe.Pointer(&t))) //
c6233120a8
12| }
```

输出地址在每次运行中可能都会不同。

使用模式三：将一个非类型安全指针转换为一个uintptr值，然后此uintptr值参与各种算术运算，再将算术运算的结果uintptr值转回非类型安全指针。

转换前后的非类型安全指针必须指向同一个内存块。一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "unsafe"
 5|
 6| type T struct {
 7|    x bool
 8|    y [3]int16
 9| }
10|
11| const N = unsafe.Offsetof(T{}.y)
```

```
12| const M = unsafe.Sizeof(T{}.y[0])
13|
14| func main() {
15|     t := T{y: [3]int16{123, 456, 789}}
16|     p := unsafe.Pointer(&t)
17|     // "uintptr(p) + N + M + M"为t.y[2]的内存地址。
18|     ty2 := (*int16)(unsafe.Pointer(uintptr(p)+N+M+M))
19|     fmt.Println(*ty2) // 789
20| }
```

其实，对于这样地址加减运算，更推荐使用上面介绍的Go 1.17中引入的`unsafe.Add`函数来完成。

注意：在上面这个例子中，转换`unsafe.Pointer(uintptr(p) + N + M + M)`不应该像下面这样被拆成两行。 请阅读下面的代码中的注释以获取原因。

```
 1| func main() {
 2|     t := T{y: [3]int16{123, 456, 789}}
 3|     p := unsafe.Pointer(&t)
 4|     // ty2 := (*int16)(unsafe.Pointer(uintptr(p)+N+M+M))
 5|     addr := uintptr(p) + N + M + M
 6|
 7|     // ...（一些其它操作）
 8|
 9|     // 从这里到下一行代码执行之前，t值将不再被任何值
10|     // 引用，所以垃圾回收器认为它可以被回收了。一旦
11|     // 它真地被回收了，下面继续使用t.y[2]值的曾经
12|     // 的地址是非法和危险的！另一个危险的原因是
13|     // t的地址在执行下一行之前可能改变（见事实三）。
14|     // 另一个潜在的危险是：如果在此期间发生了一些
15|     // 操作导致协程堆栈大小改变的情况，则记录在addr
16|     // 中的地址将失效。
17|     ty2 := (*int16)(unsafe.Pointer(addr))
18|     fmt.Println(*ty2)
19| }
```

这样的bug是非常微妙和很难被觉察到的，并且爆发出来的几率是相当得低。 一旦这样的bug爆发出来，将很让人摸不到头脑。这也是使用非类型安全指针被认为是危险操作的原因之一。

中间uintptr值可以参与`&^`清位运算来进行内存对齐计算，只要保证转换前后的非类型安全指针同时指向同一个内存块，整个转换就是合法安全的。

另一个需要注意的细节是最好不要将一个内存块的结尾边界地址存储在一个（安全或非安全）指针中。 这样做将导致紧随着此内存块的另一个内存块因为被引用而不

会被垃圾回收掉，或者因为形成非法指针而导致程序崩溃（取决于具体编译器实现）。 请阅读这个问答（第51章）以获取更多解释。

使用模式四：将非类型安全指针值转换为uintptr值并传递给syscall.Syscall函数调用。

通过对上一个使用模式的解释，我们知道像下面这样含有uintptr类型的参数的函数定义是危险的。

```
1| // 假设此函数不会被内联。
2| func DoSomething(addr uintptr) {
3|     // 对处于传递进来的地址处的值进行读写...
4| }
```

上面这个函数是危险的原因在于此函数本身不能保证传递进来的地址处的内存块一定没有被回收。 如果此内存块已经被回收了或者被重新分配给了其它值，那么此函数内部的操作将是非法和危险的。

然而，syscall标准库包中的Syscall函数的原型为：

```
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err
Errno)
```

那么此函数是如何保证处于传递给它的地址参数值a1、a2和a3处的内存块在此函数执行过程中一定没有被回收和被移动呢？ 此函数无法做出这样的保证。事实上，是编译器做出了这样的保证。 这是syscall.Syscall这样的函数的特权。其它自定义函数无法享受到这样的待遇。

我们可以认为编译器针对每个syscall.Syscall函数调用中的每个被转换为uintptr类型的非类型安全指针实参添加了一些指令，从而保证此非类型安全指针所引用着的内存块在此调用返回之前不会被垃圾回收和移动。

注意：在Go 1.15之前，类型转换表达式uintptr(anUnsafePointer)可以呈现为相关实参的子表达式。 但是，从Go 1.15开始，使用此模式的要求变得略加严格：相关实参必须呈现为uintptr(anUnsafePointer)这种形式。

下面这个调用是安全的：

```
1| syscall.Syscall(syscall.SYS_READ, uintptr(fd),
2|          uintptr(unsafe.Pointer(p)), uintptr(n))
```

但下面这个调用则是危险的：

```
 1| u := uintptr(unsafe.Pointer(p))
 2| // 被p所引用着的值在此时有可能会被回收掉，
 3| // 或者它的地址已经发生了改变。
 4| syscall.Syscall(SYS_READ, uintptr(fd), u, uintptr(n))
 5|
 6| // 相关实参必须呈现为"uintptr(anUnsafePointer)"
 7| // 这种形式。事实上，Go 1.15之前，此调用是合法的；
 8| // 但是Go 1.15略改了一点规则。
 9| syscall.Syscall(SYS_XXX, uintptr(uintptr(fd)),
10|          uint(uintptr(unsafe.Pointer(p))), uintptr(n))
```

注意：此使用模式也适用于Windows系统中的syscall.Proc.Call（https://golang.org/pkg/syscall/?GOOS=windows#Proc.Call）和syscall.LazyProc.Call（https://golang.org/pkg/syscall/?GOOS=windows#LazyProc.Call）系统调用。

再提醒一次，此使用模式不适用于其它自定义函数。

使用模式五：将reflect.Value.Pointer或者reflect.Value.UnsafeAddr方法的uintptr返回值立即转换为非类型安全指针。

reflect标准库包中的Value类型的Pointer和UnsafeAddr方法都返回一个uintptr值，而不是一个unsafe.Pointer值。 这样设计的目的是避免用户不引用unsafe标准库包就可以将这两个方法的返回值（如果是unsafe.Pointer类型）转换为任何类型安全指针类型。

这样的设计需要我们将这两个方法的调用的uintptr结果立即转换为非类型安全指针。 否则，将出现一个短暂的可能导致处于返回的地址处的内存块被回收掉的时间窗。 此时间窗是如此短暂以至于此内存块被回收掉的几率非常之低，因而这样的编程错误造成的bug的重现几率亦十分得低。

比如，下面这个调用是安全的：

```
p := (*int)
(unsafe.Pointer(reflect.ValueOf(new(int)).Pointer()))
```

而下面这个调用是危险的：

```
 1| u := reflect.ValueOf(new(int)).Pointer()
 2| // 在这个时刻，处于存储在u中的地址处的内存块
 3| // 可能会被回收掉。
 4| p := (*int)(unsafe.Pointer(u))
```

注意：Go 1.19引入了一个新的方法：`reflect.Value.UnsafePointer()`。 官方推荐以后使用此方法来替换上述两个方法。也就说，承认了原来的设计思路并不太对路。 `UnsafePointer()`返回一个`unsafe.Pointer`值。

使用模式六：将一个`reflect.SliceHeader`或者`reflect.StringHeader`值的`Data`字段转换为非类型安全指针，以及其逆转换。

和上一小节中提到的同样的原因，`reflect`标准库包中的`SliceHeader`和`StringHeader`类型的`Data`字段的类型被指定为`uintptr`，而不是`unsafe.Pointer`。

我们可以将一个字符串的指针值转换为一个`*reflect.StringHeader`指针值，从而可以对此字符串的内部进行修改。 类似地，我们可以将一个切片的指针值转换为一个`*reflect.SliceHeader`指针值，从而可以对此切片的内部进行修改。

一个使用`reflect.StringHeader`的例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "unsafe"
 5| import "reflect"
 6|
 7| func main() {
 8|    a := [...]byte{'G', 'o', 'l', 'a', 'n', 'g'}
 9|    s := "Java"
10|    hdr := (*reflect.StringHeader)(unsafe.Pointer(&s))
11|    hdr.Data = uintptr(unsafe.Pointer(&a))
12|    hdr.Len = len(a)
13|    fmt.Println(s) // Golang
14|    // 现在，字符串s和切片a共享着底层的byte字节序列，
15|    // 从而使得此字符串中的字节变得可以修改。
16|    a[2], a[3], a[4], a[5] = 'o', 'g', 'l', 'e'
17|    fmt.Println(s) // Google
18| }
```

一个使用了`reflect.SliceHeader`的例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
```

```
 5|    "unsafe"
 6|    "reflect"
 7| )
 8|
 9| func main() {
10|    a := [6]byte{'G', 'o', '1', '0', '1'}
11|    bs := []byte("Golang")
12|    hdr := (*reflect.SliceHeader)(unsafe.Pointer(&bs))
13|    hdr.Data = uintptr(unsafe.Pointer(&a))
14|
15|    hdr.Len = 2
16|    hdr.Cap = len(a)
17|    fmt.Printf("%s\n", bs) // Go
18|    bs = bs[:cap(bs)]
19|    fmt.Printf("%s\n", bs) // Go101
20| }
```

一般说来，我们只应该从一个已经存在的字符串值得到一个`*reflect.StringHeader`指针， 或者从一个已经存在的切片值得到一个`*reflect.SliceHeader`指针， 而不应该从一个全新的`StringHeader`值生成一个字符串，或者从一个全新的`SliceHeader`值生成一个切片。 比如，下面的代码是不安全的：

```
1| var hdr reflect.StringHeader
2| hdr.Data = uintptr(unsafe.Pointer(new([5]byte)))
3| // 在此时刻，上一行代码中刚开辟的数组内存块已经不再被任何值
4| // 所引用，所以它可以被回收了。
5| hdr.Len = 5
6| s := *(*string)(unsafe.Pointer(&hdr)) // 危险！
```

下面是一个展示了如何通过使用非类型安全途径将一个字符串转换为字节切片的例子。 和使用类型安全途径进行转换不同，使用非类型安全途径避免了复制一份底层字节序列。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "reflect"
 6|    "strings"
 7|    "unsafe"
 8| )
 9|
10| func String2ByteSlice(str string) (bs []byte) {
11|    strHdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
```

```
12|    sliceHdr := (*reflect.SliceHeader)(unsafe.Pointer(&bs))
13|    sliceHdr.Data = strHdr.Data
14|    sliceHdr.Cap = strHdr.Len
15|    sliceHdr.Len = strHdr.Len
16|    return
17| }
18|
19| func main() {
20|    // str := "Golang"
21|    // 对于官方标准编译器来说，上面这行将使str中的字节
22|    // 开辟在不可修改内存区。所以这里我们使用下面这行。
23|    str := strings.Join([]string{"Go", "land"}, "")
24|    s := String2ByteSlice(str)
25|    fmt.Printf("%s\n", s) // Goland
26|    s[5] = 'g'
27|    fmt.Println(str) // Golang
28| }
```

注意：当使用上面展示的使用非类型安全指针将一个字符串转换为字节切片时，请确保结果此源字符串的生命期内务必不要修改结果字节切片中的字节值（上面的例子违背了此原则）。 事实上，更为推荐的是最好永远不要修改结果字节切片中的字节值。此非类型安全方式的目的主要是为了在局部感知范围内避免一次内存开辟，而不是一种通用的方式。

我们可以使用类似的实现（如下所示）来将一个字节切片转换为字符串。 此实现被模式一中展示的方法略为安全一些（但是也更慢一些）。

```
1| func ByteSlice2String(bs []byte) (str string) {
2|    sliceHdr := (*reflect.SliceHeader)(unsafe.Pointer(&bs))
3|    strHdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
4|    strHdr.Data = sliceHdr.Data
5|    strHdr.Len = sliceHdr.Len
6|    return
7| }
```

同样地，请确保结果此结果字符串的生命期内务必不要修改实参字节切片中的字节值。

最后，顺便举一个违背了模式三的使用原则的例子：

```
1| package main
2|
3| import (
4|    "fmt"
5|    "reflect"
6|    "unsafe"
```

```
 7| )
 8|
 9| func Example_Bad() *byte {
10|    var str = "godoc"
11|    hdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
12|    pbyte := (*byte)(unsafe.Pointer(hdr.Data + 2))
13|    return pbyte // *pbyte == 'd'
14| }
15|
16| func main() {
17|    fmt.Println(string(*Example_Bad()))
18| }
```

下面是两个正确的实现：

```
 1| func Example_Good1() *byte {
 2|    var str = "godoc"
 3|    hdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
 4|    pbyte := (*byte)(unsafe.Pointer(
 5|       uintptr(unsafe.Pointer(hdr.Data)) + 2))
 6|    return pbyte
 7| }
 8|
 9| // 从Go 1.17开始也可以使用此实现。
10| func Example_Good2() *byte {
11|    var str = "godoc"
12|    hdr := (*reflect.StringHeader)(unsafe.Pointer(&str))
13|    pbyte := (*byte)(unsafe.Add(unsafe.Pointer(hdr.Data),
2))
14|    return pbyte
15| }
```

上面这几个例子借鉴自Bryan C. Mills在slack中发表的一个留言。

`reflect`标准库包中`SliceHeader`和`StringHeader`类型的[文档](https://golang.google.cn/pkg/reflect/#SliceHeader)（https://golang.google.cn/pkg/reflect/#SliceHeader）提到这两个结构体类型的定义不保证在以后的版本中不发生改变。 这也可以看作是使用非类型安全指针的另一个（较低的）潜在风险。 好在目前（Go 1.23）的两个主流Go编译器（标准编译器和gccgo编译器）都认可当前版本中的定义。

Go核心开发团队也意识到了这两个类型的使用不方便并且容易出错，因此，这两个类型从Go 1.20开始已经被不再被推荐使用了并且它们已经在Go 1.21中被宣布为废弃了。 取而代之，我们应该尽量使用本文前面介绍的`unsafe.String`、`unsafe.StringData`、`unsafe.Slice`和`unsafe.SliceData`这几个函数。

总结一下

从上面解释中，我们得知，对于某些情形，非类型安全机制可以帮助我们写出运行效率更高的代码。 但是，使用非类型安全指针也使得我们可能轻易地写出一些重现几率非常低的微妙的bug。 一个含有这样的bug的程序很可能在很长一段时间内都运行正常，但是突然变得不正常甚至崩溃。 这样的bug很难发现和调试。

我们只应该在不得不使用非类型安全机制的时候才使用它们。 特别地，当我们使用非类型安全机制时，请务必遵循上面列出的使用模式。

重申一次，我们应该知晓当前的非类型安全机制规则和使用模式可能在以后的Go版本中完全失效。 当然，目前没有任何迹象表明这种变化将很快会来到。 但是，一旦发生这种变化，本文中列出的当前是正确的代码将变得不再安全甚至编译不通过。 所以，在实践中，请尽量保证能够将使用了非类型安全机制的代码轻松改为使用安全途径实现。

最后值得提一下的是，Go官方工具链1.14中加入了一个`-gcflags=all=-d=checkptr`编译器动态分析选项（在Windows平台上推荐使用工具链1.15+）。 当此选项被使用的时候，编译出的程序在运行时会监测到很多（但并非所有）非类型安全指针的错误使用。一旦错误的使用被监测到，恐慌将产生。 感谢Matthew Dempsky实现了此特性（https://github.com/golang/go/issues/22218）。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

泛型

在Go 1.18以前，Go只支持内置泛型。 从Go 1.18开始，Go也支持自定义泛型。 本文只介绍内置泛型。

Go通过各种一等公民组合类型来实现内置泛型。 我们可以用各种一等公民组合类型来组合出无穷个类型。 本文将展示一些自定义组合类型的例子并解释如何解读这些自定义类型。

一些复杂组合类型的例子

Go中的组合类型字面表示设计得非常直观和易于解读。 即使对于一些非常复杂的类型，我们也几乎不可能在解读它们的字面形式中迷失。 下面将从简单到复杂列出一些自定义组合类型的例子并进行解读。

先看一个简单的例子：

```
1| [3][4]int
```

当解读一个类型的字面形式时，我们应该从左到右进行解读。 左边开头的`[3]`表示着这个类型为一个数组类型，它右边的整个部分为它的元素类型。 对于这个例子，它的元素类型为另外一个数组类型`[4]int`。 此另外一个数组类型的元素类型为内置类型`int`。 第一个数组类型可以被看作是一个二维数组类型。

一个使用此数组类型的例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5| )
 6|
 7| func main() {
 8|     matrix := [3][4]int{
 9|         {1, 0, 0, 1},
10|         {0, 1, 0, 1},
11|         {0, 0, 1, 1},
12|     }
13|
14|     matrix[1][1] = 3
15|     a := matrix[1] // 变量a的类型为[4]int
```

```
16|    fmt.Println(a) // [0 3 0 1]
17| }
```

类似的，

- `[][]string`是一个元素类型为另一个切片类型`[]string`的切片类型。
- `**bool`是一个基类型为另一个指针类型`*bool`的指针类型。
- `chan chan int`是一个元素类型为另一个通道类型的`chan int`的通道类型。
- `map[int]map[int]string`是一个元素类型为另一个映射类型`map[int]string`的映射类型。 这两个映射类型的键值类型均为内置类型`int`。
- `func(int32) func(int32)`是一个只有一个输入参数和一个返回值的函数类型，此返回值的类型为一个只有一个输入参数的函数类型。 这两个函数类型的输入参数的类型均为内置类型`int32`。

下面是另一个自定义组合类型：

```
1| chan *[16]byte
```

最左边的`chan`关键字表明此类型是一个通道类型。 `chan`关键字右边的整个部分`*[16]byte`表示此通道类型的元素类型，此元素类型是一个指针类型。 此指针类型的基类型为`*`右边的整个部分：`[16]byte`，此基类型为一个数组类型。 此数组类型的元素类型为内置类型`byte`。

一个使用此通道类型的例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6|    "crypto/rand"
 7| )
 8|
 9| func main() {
10|    c := make(chan *[16]byte)
11|
12|    go func() {
13|       // 使用两个数组以避免数据竞争。
14|       var dataA, dataB = new([16]byte), new([16]byte)
15|       for {
16|          _, err := rand.Read(dataA[:])
17|          if err != nil {
```

```
18|             close(c)
19|          } else {
20|             c <- dataA
21|             dataA, dataB = dataB, dataA
22|          }
23|       }
24|    }()
25|
26|    for data := range c {
27|       fmt.Println((*data)[:])
28|       time.Sleep(time.Second / 2)
29|    }
30| }
```

类似的，类型`map[string][]func(int) int`为一个映射类型。 此映射类型的键值类型为内置类型`string`，右边剩余的部分为此映射类型的元素类型。 `[]`表明此映射的元素类型为一个切片类型，此切片类型的元素类型为一个函数类型`func(int) int`。

一个使用了此映射类型的例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    addone := func(x int) int {return x + 1}
 7|    square := func(x int) int {return x * x}
 8|    double := func(x int) int {return x + x}
 9|
10|    transforms := map[string][]func(int) int {
11|       "inc,inc,inc": {addone, addone, addone},
12|       "sqr,inc,dbl": {square, addone, double},
13|       "dbl,sqr,sqr": {double, double, square},
14|    }
15|
16|    for _, n := range []int{2, 3, 5, 7} {
17|       fmt.Println(">>>", n)
18|       for name, transfers := range transforms {
19|          result := n
20|          for _, xfer := range transfers {
21|             result = xfer(result)
22|          }
23|          fmt.Printf(" %v: %v \n", name, result)
24|       }
```

```
25|    }
26| }
```

下面是一个看上去有些复杂的类型：

```
 1| []map[struct {
 2|    a int
 3|    b struct {
 4|       x string
 5|       y bool
 6|    }
 7| }]interface {
 8|    Build([]byte, struct {x string; y bool}) error
 9|    Update(dt float64)
10|    Destroy()
11| }
```

让我们从左到右解读此类型。 最左边开始的`[]`表明这是一个切片类型，紧跟着的`map`关键字表明此切片类型的元素为一个映射类型。 `map`关键字后紧跟的一对方括号`[]`中的结构体类型字面形式表明此映射的键值类型为一个结构体类型。 此中括号右边的整个部分表明此映射的元素类型为一个接口类型。此接口类型指定了三个方法。 此映射的键值结构体类型有两个字段，第一个字段的名称和类型为`a`和内置类型`int`； 第二个字段的名称为`b`，它的类型为另外一个结构体类型`struct {x string; y bool}`。 此另外一个结构体类型也有两个字段：内置`string`类型的字段`x`和内置`bool`类型的字段`y`。

请注意第二个结构体类型也被用做刚提及的接口类型所指定的其中一个方法中的其中一个参数类型。

我们经常将复杂类型的各个组成部分单独提前声明为一个类型名，从而获得更高的可读性。 下面的代码中的类型别名`T`和上面刚解读的类型表示同一个类型。

```
 1| type B = struct {
 2|    x string
 3|    y bool
 4| }
 5|
 6| type K = struct {
 7|    a int
 8|    b B
 9| }
10|
11| type E = interface {
12|    Build([]byte, B) error
```

```
13|     Update(dt float64)
14|     Destroy()
15| }
16|
17| type T = []map[K]E
```

内置泛型函数

Go中当前的内置泛型除了上述类型组合，还有一些支持泛型的内置函数。 比如，内置函数 `len` 可以用来获取各种容器值的长度。 `unsafe` 标准库包中的函数也可以被看作是支持泛型的内置函数。 这些函数在前面的各篇文章中已经一一介绍过了。

自定义泛型

从1.18版本开始，Go已经支持自定义泛型。 请阅读[《Go自定义泛型101》](https://gfw.go101.org/generics/101.html)（https://gfw.go101.org/generics/101.html）一书来了解如何使用自定义泛型。

本书由[老貘](#)历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问[github.com/go1ang101/go1ang101](github.com/go1ang101/go1ang101)获取本书最新版）

反射

Go是一门具有良好反射支持的静态语言。 本文将解释`reflect`标准库包中提供的反射功能。

在阅读本剩下的部分之前，最好先阅读Go类型系统概述（第14章）和接口（第23章）两篇文章。

反射概述

Go中提供的反射功能带来了很多动态特性。 很多标准库，比如`fmt`和很多`encoding`包，均十分依赖于反射机制。

我们可以通过`reflect`库包中`Type`和`Value`两个类型提供的功能来观察不同的Go值。 本文下面的内容将介绍如何使用这两个类型。

Go反射机制设计的目标之一是任何非反射操作都可以通过反射机制来完成。 由于各种各样的原因，此目标并没有得到100%的实现。 但是，目前大部分的非反射操作都可以通过反射机制来完成。 另一方面，通过反射，我们也可以完成一些使用非反射操作不可能完成的操作。 不能或者只能通过反射完成的操作将在下面的讲解中提及。

`reflect.Type`类型和值

通过调用`reflect.TypeOf`函数，我们可以从一个任何非接口类型的值创建一个`reflect.Type`值，此`reflect.Type`值表示着此非接口值的类型。通过此值，我们可以得到很多此非接口类型的信息。 当然，我们也可以将一个接口值传递给一个`reflect.TypeOf`函数调用，但是此调用将返回一个表示着此接口值的动态类型的`reflect.Type`值。 实际上，`reflect.TypeOf`函数的唯一参数的类型为`interface{}`，`reflect.TypeOf`函数将总是返回一个表示着此唯一接口参数值的动态类型的`reflect.Type`值。 那如何得到一个表示着某个接口类型的`reflect.Type`值呢？ 我们必须通过下面将要介绍的一些间接途径来达到这一目的。

从Go 1.22开始，我们也可以调用`reflect.TypeFor`函数（https://docs.go101.org/std/pkg/reflect.html#name-TypeFor）来得到一个表示着一个编译时刻已知的类型的`reflect.Type`值。 此编译时刻已知的类型可以是一个非接口类型，也可以是一个接口类型。

类型reflect.Type为一个接口类型，它指定了若干方法（https://golang.google.cn/pkg/reflect/#Type）。 通过这些方法，我们能够观察到一个reflect.Type值所表示的Go类型的各种信息。 这些方法中的有些适用于所有种类（https://golang.google.cn/pkg/reflect/#Kind）的类型，有些只适用于一种或几种类型。 通过不合适的reflect.Type属主值调用某个方法将在运行时产生一个恐慌。 请阅读reflect代码库中各个方法的文档来获取如何正确地使用这些方法。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| func main() {
 7|     type A = [16]int16
 8|     var c <-chan map[A][]byte
 9|     tc := reflect.TypeOf(c)
10|     fmt.Println(tc.Kind())    // chan
11|     fmt.Println(tc.ChanDir()) // <-chan
12|     tm := tc.Elem()
13|     ta, tb := tm.Key(), tm.Elem()
14|     fmt.Println(tm.Kind(), ta.Kind(), tb.Kind()) // map
array slice
15|     tx, ty := ta.Elem(), tb.Elem()
16|
17|     // byte是uint8类型的别名。
18|     fmt.Println(tx.Kind(), ty.Kind()) // int16 uint8
19|     fmt.Println(tx.Bits(), ty.Bits()) // 16 8
20|     fmt.Println(tx.ConvertibleTo(ty)) // true
21|     fmt.Println(tb.ConvertibleTo(ta)) // false
22|
23|     // 切片类型和映射类型都是不可比较类型。
24|     fmt.Println(tb.Comparable()) // false
25|     fmt.Println(tm.Comparable()) // false
26|     fmt.Println(ta.Comparable()) // true
27|     fmt.Println(tc.Comparable()) // true
28| }
```

目前，Go支持26种种类的类型（https://golang.google.cn/pkg/reflect/#Kind）。

在上面这个例子中，我们使用方法`Elem`来得到某些类型的元素类型。 实际上，此方法也可以用来得到一个指针类型的基类型。一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| type T []interface{m()}
 7| func (T) m() {}
 8|
 9| func main() {
10|    tp := reflect.TypeOf(new(interface{}))
11|    tt := reflect.TypeOf(T{})
12|    fmt.Println(tp.Kind(), tt.Kind()) // ptr slice
13|
14|    // 使用间接的方法得到表示两个接口类型的reflect.Type值。
15|    ti, tim := tp.Elem(), tt.Elem()
16|    fmt.Println(ti.Kind(), tim.Kind()) // interface
interface
17|
18|    fmt.Println(tt.Implements(tim))  // true
19|    fmt.Println(tp.Implements(tim))  // false
20|    fmt.Println(tim.Implements(tim)) // true
21|
22|    // 所有的类型都实现了任何空接口类型。
23|    fmt.Println(tp.Implements(ti))  // true
24|    fmt.Println(tt.Implements(ti))  // true
25|    fmt.Println(tim.Implements(ti)) // true
26|    fmt.Println(ti.Implements(ti))  // true
27| }
```

上面这个例子同时也展示了如何通过间接的途径得到一个表示一个接口类型的`reflect.Type`值。

我们可以通过反射列出一个类型的所有方法和一个结构体类型的所有（导出和非导出）字段的类型。 我们也可以通过反射列出一个函数类型的各个输入参数和返回结果类型。

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| type F func(string, int) bool
```

```
 7| func (f F) m(s string) bool {
 8|    return f(s, 32)
 9| }
10| func (f F) M() {}
11|
12| type I interface{m(s string) bool; M()}
13|
14| func main() {
15|    var x struct {
16|       F F
17|       i I
18|    }
19|    tx := reflect.TypeOf(x)
20|    fmt.Println(tx.Kind())        // struct
21|    fmt.Println(tx.NumField())    // 2
22|    fmt.Println(tx.Field(1).Name) // i
23|    // 包路径（PkgPath）是非导出字段（或者方法）的内在属性。
24|    fmt.Println(tx.Field(0).PkgPath) //
25|    fmt.Println(tx.Field(1).PkgPath) // main
26|
27|    tf, ti := tx.Field(0).Type, tx.Field(1).Type
28|    fmt.Println(tf.Kind())               // func
29|    fmt.Println(tf.IsVariadic())         // false
30|    fmt.Println(tf.NumIn(), tf.NumOut()) // 2 1
31|    t0, t1, t2 := tf.In(0), tf.In(1), tf.Out(0)
32|    // 下一行打印出：string int bool
33|    fmt.Println(t0.Kind(), t1.Kind(), t2.Kind())
34|
35|    fmt.Println(tf.NumMethod(), ti.NumMethod()) // 1 2
36|    fmt.Println(tf.Method(0).Name)              // M
37|    fmt.Println(ti.Method(1).Name)              // m
38|    _, ok1 := tf.MethodByName("m")
39|    _, ok2 := ti.MethodByName("m")
40|    fmt.Println(ok1, ok2) // false true
41| }
```

从上面这个例子我们可以看出：

1. 对于非接口类型，`reflect.Type.NumMethod`方法只返回一个类型的所有导出的方法（包括通过内嵌得来的隐式方法）的个数，并且 方法`reflect.Type.MethodByName`不能用来获取一个类型的非导出方法； 而对于接口类型，则并无这些限制（Go 1.16之前的文档对这两个方法的描述不准确，并没有体现出这个差异）。 此情形同样存在于下一节将要介绍的`reflect.Value`类型上的相应方法。

2. 虽然`reflect.Type.NumField`方法返回一个结构体类型的所有字段（包括非导出字段）的数目，但是不推荐（https://golang.org/pkg/reflect/#pkg-note-BUG）使用方法`reflect.Type.FieldByName`来获取非导出字段。

我们可以通过反射来检视结构体字段的标签信息（https://golang.org/pkg/reflect/#StructTag）。结构体字段标签的类型为`reflect.StructTag`，它的方法`Get`和`Lookup`用来检视字段标签中的键值对。一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| type T struct {
 7|    X    int  `max:"99" min:"0" default:"0"`
 8|    Y, Z bool `optional:"yes"`
 9| }
10|
11| func main() {
12|    t := reflect.TypeOf(T{})
13|    x := t.Field(0).Tag
14|    y := t.Field(1).Tag
15|    z := t.Field(2).Tag
16|    fmt.Println(reflect.TypeOf(x)) // reflect.StructTag
17|    // v的类型为string
18|    v, present := x.Lookup("max")
19|    fmt.Println(len(v), present)      // 2 true
20|    fmt.Println(x.Get("max"))         // 99
21|    fmt.Println(x.Lookup("optional")) //  false
22|    fmt.Println(y.Lookup("optional")) // yes true
23|    fmt.Println(z.Lookup("optional")) // yes true
24| }
```

注意：

- 键值对中的键不能包含空格（Unicode值为32）、双引号（Unicode值为34）和冒号（Unicode值为58）。
- 为了形成键值对，所设想的键值对形式中的冒号的后面不能紧跟着空格字符。所以
  `` `optional: "yes"` ``不形成键值对。
- 键值对中的值中的空格不会被忽略。所以
  `` `json:"author, omitempty“` ``、

`json:" author,omitempty“`以及
`json:"author,omitempty“`各不相同。

- 每个字段标签应该呈现为单行才能使它的整个部分都对键值对的形成有贡献。

reflect代码包也提供了一些其它函数来动态地创建出来一些无名组合类型。

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| func main() {
 7|    ta := reflect.ArrayOf(5, reflect.TypeOf(123))
 8|    fmt.Println(ta) // [5]int
 9|    tc := reflect.ChanOf(reflect.SendDir, ta)
10|    fmt.Println(tc) // chan<- [5]int
11|    tp := reflect.PtrTo(ta)
12|    fmt.Println(tp) // *[5]int
13|    ts := reflect.SliceOf(tp)
14|    fmt.Println(ts) // []*[5]int
15|    tm := reflect.MapOf(ta, tc)
16|    fmt.Println(tm) // map[[5]int]chan<- [5]int
17|    tf := reflect.FuncOf([]reflect.Type{ta},
18|              []reflect.Type{tp, tc}, false)
19|    fmt.Println(tf) // func([5]int) (*[5]int, chan<- [5]int)
20|    tt := reflect.StructOf([]reflect.StructField{
21|       {Name: "Age", Type: reflect.TypeOf("abc")},
22|    })
23|    fmt.Println(tt)            // struct { Age string }
24|    fmt.Println(tt.NumField()) // 1
25| }
```

上面的例子并未展示和reflect.Type相关的所有函数和方法。 请阅读reflect标准库代码包的文档以获取如何使用这些函数和方法。

注意，到目前为止（Go 1.23），我们无法通过反射动态创建一个接口类型。这是Go反射目前的一个限制。

另一个限制是使用反射动态创建结构体类型的时候可能会有各种不完美的情况出现。

第三个限制是我们无法通过反射来声明一个新的类型。

reflect.Value类型和值

类似的，我们可以通过调用reflect.ValueOf函数，从一个非接口类型的值创建一个reflect.Value值。 此reflect.Value值代表着此非接口值。 和reflect.TypeOf函数类似， reflect.ValueOf函数也只有一个interface{}类型的参数。 当我们将一个接口值传递给一个reflect.ValueOf函数调用时，此调用返回的是代表着此接口值的动态值的一个reflect.Value值。 我们必须通过间接的途径获得一个代表一个接口值的reflect.Value值。

被一个reflect.Value值代表着的值常称为此reflect.Value值的底层值（underlying value）。

reflect.Value类型有很多方法（https://golang.google.cn/pkg/reflect/）。我们可以调用这些方法来观察和操纵一个reflect.Value属主值表示的Go值。 这些方法中的有些适用于所有种类类型的值，有些只适用于一种或几种类型的值。 通过不合适的reflect.Value属主值调用某个方法将在运行时产生一个恐慌。 请阅读reflect代码库中各个方法的文档来获取如何正确地使用这些方法。

一个reflect.Value值的CanSet方法将返回此reflect.Value值代表的Go值是否可以被修改（可以被赋值）。 如果一个Go值可以被修改，则我们可以调用对应的reflect.Value值的Set方法来修改此Go值。 注意： reflect.ValueOf函数直接返回的reflect.Value值都是不可修改的。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| func main() {
 7|     n := 123
 8|     p := &n
 9|     vp := reflect.ValueOf(p)
10|     fmt.Println(vp.CanSet(), vp.CanAddr()) // false false
11|     vn := vp.Elem() // 取得vp的底层指针值引用的值的代表值
12|     fmt.Println(vn.CanSet(), vn.CanAddr()) // true true
13|     vn.Set(reflect.ValueOf(789)) // <=> vn.SetInt(789)
14|     fmt.Println(n)               // 789
15| }
```

一个结构体值的非导出字段不能通过反射来修改。

```
 1| package main
 2|
 3| import "fmt"
```

```
 4| import "reflect"
 5|
 6| func main() {
 7|    var s struct {
 8|       X interface{} // 一个导出字段
 9|       y interface{} // 一个非导出字段
10|    }
11|    vp := reflect.ValueOf(&s)
12|    // 如果vp代表着一个指针，下一行等价于"vs := vp.Elem()"。
13|    vs := reflect.Indirect(vp)
14|    // vx和vy都各自代表着一个接口值。
15|    vx, vy := vs.Field(0), vs.Field(1)
16|    fmt.Println(vx.CanSet(), vx.CanAddr()) // true true
17|    // vy is addressable but not modifiable.
18|    fmt.Println(vy.CanSet(), vy.CanAddr()) // false true
19|    vb := reflect.ValueOf(123)
20|    vx.Set(vb)     // okay, 因为vx代表的值是可修改的。
21|    // vy.Set(vb)  // 会造成恐慌，因为vy代表的值是不可修改的。
22|    fmt.Println(s) // {123 <nil>
23|    fmt.Println(vx.IsNil(), vy.IsNil()) // false true
24| }
```

上例中同时也展示了如何间接地获取底层值为接口值的`reflect.Value`值。

从上两例中，我们可以得知有两种方法获取一个代表着一个指针所引用着的值的`reflect.Value`值：

1. 通过调用代表着此指针值的`reflect.Value`值的`Elem`方法。
2. 将代表着此指针值的`reflect.Value`值的传递给一个`reflect.Indirect`函数调用。（如果传递给一个`reflect.Indirect`函数调用的实参不代表着一个指针值，则此调用返回此实参的一个复制。）

注意：`reflect.Value.Elem`方法也可以用来获取一个代表着一个接口值的动态值的`reflect.Value`值，比如下例中所示。

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| func main() {
 7|    var z = 123
 8|    var y = &z
 9|    var x interface{} = y
10|    v := reflect.ValueOf(&x)
```

```
    vx := v.Elem()
    vy := vx.Elem()
    vz := vy.Elem()
    vz.Set(reflect.ValueOf(789))
    fmt.Println(z) // 789
}
```

reflect标准库包中也提供了一些对应着内置函数或者各种非反射功能的函数。 下面这个例子展示了如何利用这些函数将一个（效率不高的）自定义泛型函数绑定到不同的类型的函数值上。

```
package main

import "fmt"
import "reflect"

func InvertSlice(args []reflect.Value) []reflect.Value {
    inSlice, n := args[0], args[0].Len()
    outSlice := reflect.MakeSlice(inSlice.Type(), 0, n)
    for i := n-1; i >= 0; i-- {
       element := inSlice.Index(i)
       outSlice = reflect.Append(outSlice, element)
    }
    return []reflect.Value{outSlice}
}

func Bind(p interface{},
       f func ([]reflect.Value) []reflect.Value) {
    // invert代表着一个函数值。
    invert := reflect.ValueOf(p).Elem()
    invert.Set(reflect.MakeFunc(invert.Type(), f))
}

func main() {
    var invertInts func([]int) []int
    Bind(&invertInts, InvertSlice)
    fmt.Println(invertInts([]int{2, 3, 5})) // [5 3 2]

    var invertStrs func([]string) []string
    Bind(&invertStrs, InvertSlice)
    fmt.Println(invertStrs([]string{"Go", "C"})) // [C Go]
}
```

如果一个reflect.Value值的底层值为一个函数值，则我们可以调用此reflect.Value值的Call方法来调用此函数。 每个Call方法调用接受一个

`[]reflect.Value`类型的参数（表示传递给相应函数调用的各个实参）并返回一个同类型结果（表示相应函数调用返回的各个结果）。

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| type T struct {
 7|    A, b int
 8| }
 9|
10| func (t T) AddSubThenScale(n int) (int, int) {
11|    return n * (t.A + t.b), n * (t.A - t.b)
12| }
13|
14| func main() {
15|    t := T{5, 2}
16|    vt := reflect.ValueOf(t)
17|    vm := vt.MethodByName("AddSubThenScale")
18|    results := vm.Call([]reflect.Value{reflect.ValueOf(3)})
19|    fmt.Println(results[0].Int(), results[1].Int()) // 21 9
20|
21|    neg := func(x int) int {
22|       return -x
23|    }
24|    vf := reflect.ValueOf(neg)
25|    fmt.Println(vf.Call(results[:1])[0].Int()) // -21
26|    fmt.Println(vf.Call([]reflect.Value{
27|       vt.FieldByName("A"), // 如果是字段b，则造成恐慌
28|    })[0].Int()) // -5
29| }
```

请注意：非导出结构体字段值不能用做反射函数调用中的实参。 如果上例中的`vt.FieldByName("A")`被替换为`vt.FieldByName("b")`，则将产生一个恐慌。

下面是一个使用映射反射值的例子。

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| func main() {
 7|    valueOf := reflect.ValueOf
```

```
 8|    m := map[string]int{"Unix": 1973, "Windows": 1985}
 9|    v := valueOf(m)
10|    // 第二个实参为Value零值时，表示删除一个映射条目。
11|    v.SetMapIndex(valueOf("Windows"), reflect.Value{})
12|    v.SetMapIndex(valueOf("Linux"), valueOf(1991))
13|    for i := v.MapRange(); i.Next(); {
14|       fmt.Println(i.Key(), "\t:", i.Value())
15|    }
16| }
```

注意：方法reflect.Value.MapRange方法是从Go 1.12开始才支持的。

下面是一个使用通道反射值的例子。

```
 1| package main
 2|
 3| import "fmt"
 4| import "reflect"
 5|
 6| func main() {
 7|    c := make(chan string, 2)
 8|    vc := reflect.ValueOf(c)
 9|    vc.Send(reflect.ValueOf("C"))
10|    succeeded := vc.TrySend(reflect.ValueOf("Go"))
11|    fmt.Println(succeeded) // true
12|    succeeded = vc.TrySend(reflect.ValueOf("C++"))
13|    fmt.Println(succeeded) // false
14|    fmt.Println(vc.Len(), vc.Cap()) // 2 2
15|    vs, succeeded := vc.TryRecv()
16|    fmt.Println(vs.String(), succeeded) // C true
17|    vs, sentBeforeClosed := vc.Recv()
18|    fmt.Println(vs.String(), sentBeforeClosed) // Go true
19|    vs, succeeded = vc.TryRecv()
20|    fmt.Println(vs.String()) // <$1>
21|    fmt.Println(succeeded)   // false
22| }
```

reflect.Value类型的TrySend和TryRecv方法对应着只有一个case分支和一个default分支的select流程控制代码块（第21章）。

我们可以使用reflect.Select函数在运行时刻来模拟具有不定case分支数量的select流程控制代码块。

```
 1| package main
 2|
```

```
 3| import "fmt"
 4| import "reflect"
 5|
 6| func main() {
 7|    c := make(chan int, 1)
 8|    vc := reflect.ValueOf(c)
 9|    succeeded := vc.TrySend(reflect.ValueOf(123))
10|    fmt.Println(succeeded, vc.Len(), vc.Cap()) // true 1 1
11|
12|    vSend, vZero := reflect.ValueOf(789), reflect.Value{}
13|    branches := []reflect.SelectCase{
14|       {Dir: reflect.SelectDefault, Chan: vZero, Send:
vZero},
15|       {Dir: reflect.SelectRecv, Chan: vc, Send: vZero},
16|       {Dir: reflect.SelectSend, Chan: vc, Send: vSend},
17|    }
18|    selIndex, vRecv, sentBeforeClosed :=
reflect.Select(branches)
19|    fmt.Println(selIndex)         // 1
20|    fmt.Println(sentBeforeClosed) // true
21|    fmt.Println(vRecv.Int())      // 123
22|    vc.Close()
23|    // 再模拟一次select流程控制代码块。因为vc已经关闭了，
24|    // 所以需将最后一个case分支去除，否则它可能会造成一个恐慌。
25|    selIndex, _, sentBeforeClosed =
reflect.Select(branches[:2])
26|    fmt.Println(selIndex, sentBeforeClosed) // 1 false
27| }
```

一些reflect.Value值可能表示着不合法的Go值。 这样的值为reflect.Value类型的零值（即没有底层值的reflect.Value值）。

```
 1| package main
 2|
 3| import "reflect"
 4| import "fmt"
 5|
 6| func main() {
 7|    var z reflect.Value // 一个reflect.Value零值
 8|    fmt.Println(z)      // <$1>
 9|    v := reflect.ValueOf((*int)(nil)).Elem()
10|    fmt.Println(v)      // <$1>
11|    fmt.Println(v == z) // true
12|    var i = reflect.ValueOf([]interface{}{nil}).Index(0)
13|    fmt.Println(i)             // <nil>
```

```
14|    fmt.Println(i.Elem())      // <$1>
15|    fmt.Println(i.Elem() == z) // true
16| }
```

从上面的例子中，我们知道，使用空接口`interface{}`值做为中介，一个Go值可以转换为一个`reflect.Value`值。 逆过程类似，通过调用一个`reflect.Value`值的`Interface`方法得到一个`interface{}`值，然后将此`interface{}`断言为原来的Go值。 但是，请注意，调用一个代表着非导出字段的`reflect.Value`值的`Interface`方法将导致一个恐慌。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "reflect"
 6|    "time"
 7| )
 8|
 9| func main() {
10|    vx := reflect.ValueOf(123)
11|    vy := reflect.ValueOf("abc")
12|    vz := reflect.ValueOf([]bool{false, true})
13|    vt := reflect.ValueOf(time.Time{})
14|
15|    x := vx.Interface().(int)
16|    y := vy.Interface().(string)
17|    z := vz.Interface().([]bool)
18|    m := vt.MethodByName("IsZero").Interface().(func() bool)
19|    fmt.Println(x, y, z, m()) // 123 abc [false true] true
20|
21|    type T struct {x int}
22|    t := &T{3}
23|    v := reflect.ValueOf(t).Elem().Field(0)
24|    fmt.Println(v)              // 3
25|    fmt.Println(v.Interface()) // panic
26| }
```

`Value.IsZero`方法是Go 1.13中引进的，此方法用来查看一个值是否为零值。

从Go 1.17开始，一个切片可以被转化为一个相同元素类型的数组的指针类型（第18章）。 但是如果在这样的一个转换中数组类型的长度过长，将导致恐慌产生。 因此Go 1.17同时引入了一个`Value.CanConvert(T Type)`方法，用来检查一个转换是否会成功（即不会产生恐慌）。

一个使用了CanConvert方法的例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "reflect"
 6| )
 7|
 8| func main() {
 9|     s := reflect.ValueOf([]int{1, 2, 3, 4, 5})
10|     ts := s.Type()
11|     t1 := reflect.TypeOf(&[5]int{})
12|     t2 := reflect.TypeOf(&[6]int{})
13|     fmt.Println(ts.ConvertibleTo(t1)) // true
14|     fmt.Println(ts.ConvertibleTo(t2)) // true
15|     fmt.Println(s.CanConvert(t1))     // true
16|     fmt.Println(s.CanConvert(t2))     // false
17| }
```

上面这些例子并未触及到所有的和reflect.Value相关的函数和方法，请阅读reflect标准库包的文档以获取如何使用这些函数和方法。 另外请注意Go细节101（第50章）中提到的一些（第50章）关于反射的细节（第50章）。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

Go代码断行规则

如果你已经写了一些Go代码，你应该知道，Go代码风格不能太随意。 具体说来，我们不能随意在某个空格或者符号字符处断行。 本文余下的部分将列出Go代码中的详细断行规则。

分号插入规则

我们在Go编程中常遵循的一个规则是：一个显式代码块的起始左大括号`{`不放在下一行。 比如，下面这个`for`循环代码块编译将失败。

```
1|    for i := 5; i > 0; i--
2|    { // error: 未预料到的新行
3|    }
```

为了让上面这个`for`循环代码块编译成功，我们不能在起始左大括号`{`前断行，而应该像下面这样进行修改：

```
1|    for i := 5; i > 0; i-- {
2|    }
```

然而，有时候起始左大括号`{`却可以放在一个新行上，比如下面这个`for`循环代编译时没有问题的。

```
1|    for
2|    {
3|       // do something ...
4|    }
```

那么，Go代码中的根本性换行规则究竟是如何定义的呢？ 在回答这个问题之前，我们应该知道一个事实：正式的Go语法是使用（英文）分号`;`做为结尾标识符的。 但是，我们很少在Go代码中使用和看到分号。为什么呢？原因是大多数分号都是可选的，因此它们常常被省略。 在编译时刻，Go编译器会自动插入这些省略的分号。

比如，下面这个程序中的十个分号都是可以被省略掉的。

```
1| package main;
2|
3| import "fmt";
4|
5| func main() {
```

```
 6|    var (
 7|       i   int;
 8|       sum int;
 9|    );
10|    for i < 6 {
11|       sum += i;
12|       i++;
13|    };
14|    fmt.Println(sum);
15| };
```

假设上面这个程序存储在一个`semicolons.go`文件中，我们可以运行`go fmt semicolons.go`将此程序中的不必要的分号去除掉。 在编译时刻，编译器会自动此插入这些去除掉的分号（至此文件的内存中的版本）。

自动插入分号的规则是什么呢？Go白皮书[这样描述](https://golang.google.cn/ref/spec#Semicolons)（https://golang.google.cn/ref/spec#Semicolons）：

1. 在Go代码中，注释除外，如果一个代码行的最后一个语法词段（token）为下列所示之一，则一个分号将自动插入在此字段后（即行尾）：
   - 一个[标识符](#)（第5章）；
   - 一个整数、浮点数、虚部、码点或者字符串[字面量](#)（第6章）；
   - 这几个跳转关键字之一：`break`、`continue`、`fallthrough`和`return`；
   - 自增运算符`++`或者自减运算符`--`；
   - 一个右括号：`)`、`]`或`}`。
2. 为了允许一条复杂语句完全显示在一个代码行中，分号可能被插入在一个右小括号`)`或者右大括号`}`之前。

对于上述第一条规则描述的情形，我们当然也可以手动插入这些分号，就像此前的例子中所示。换句话说，这些分号在编程时是可选的。

上述第二条规则允许我们写出如下的代码：

```
1| import (_ "math"; "fmt")
2| var (a int; b string)
3| const (M = iota; N)
4| type (MyInt int; T struct{x bool; y int32})
5| type I interface{m1(int) int; m2() string}
6| func f() {print("a"); panic(nil)}
```

编译器在编译时刻将自动插入所需的分号，如下所示：

```
1| var (a int; b string;);
2| const (M = iota; N;);
3| type (MyInt int; T struct{x bool; y int32;};);
4| type I interface{m1(int) int; m2() string;};
5| func f() {print("a"); panic(nil);};
```

编译器不会为其它任何情形插入分号。如果其它任何情形需要一个分号，我们必须手动插入此分号。 比如，上例中的每行中的第一个分号必须手动插入。下例中的分号也都需要手动插入。

```
1| var a = 1; var b = true
2| a++; b = !b
3| print(a); print(b)
```

从以上两条规则可以看出，一个分号永远不会插入在`for`关键字后，这就是为什么上面的裸`for`循环例子是合法的。

分号自动插入规则导致的一个结果是：自增和自减运算必须呈现为单独的语句，它们不能被当作表达式使用。 比如，下面的代码是编译不通过的：

```
1| func f() {
2|     a := 0
3|     println(a++)
4|     println(a--)
5| }
```

上面代码编译不通过的原因是它等价于下面的代码：

```
1| func f() {
2|     a := 0
3|     println(a++;)
4|     println(a--;)
5| }
```

分号自动插入规则导致的另一个结果是：我们不能在选择器中的句点`.`之前断行。在选择器中的句点之后断行是允许的，比如：

```
1|     anObject.
2|        MethodA().
3|        MethodB().
4|        MethodC()
```

而下面这样是非法的：

```
1|     anObject
2|        .MethodA()
```

```
 3|       .MethodB()
 4|       .MethodC()
```

此代码片段是非法的原因是编译器将自动在每个右小括号`)`后插入一个分号，如下面所示：

```
 1|    anObject;
 2|       .MethodA();
 3|       .MethodB();
 4|       .MethodC();
```

上述分号自动插入规则可以让我们写出更简洁的代码，同时也允许我们写出一些合法的但看上去有些怪异的代码，比如：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func alwaysFalse() bool {return false}
 6|
 7| func main() {
 8|    for
 9|    i := 0
10|    i < 6
11|    i++ {
12|       // 使用i ...
13|    }
14|
15|    if x := alwaysFalse()
16|    !x {
17|       // ...
18|    }
19|
20|    switch alwaysFalse()
21|    {
22|    case true: fmt.Println("true")
23|    case false: fmt.Println("false")
24|    }
25| }
```

上例中所有的流程控制代码块都是合法的。编译器将在这些行的行尾自动插入一个分号：第*9*行、第*10*行、第*15*行和第*20*行。

注意，上例中的`switch-case`代码块将输出`true`，而不是`false`。 此代码块和下面这个是不同的：

```
 1|    switch alwaysFalse() {
 2|    case true: fmt.Println("true")
 3|    case false: fmt.Println("false")
 4|    }
```

如果我们使用`go fmt`命令格式化前者，一个分号将自动添加到`alwaysFalse()`函数调用之后，如下所示：

```
 1|    switch alwaysFalse();
 2|    {
 3|    case true: fmt.Println("true")
 4|    case false: fmt.Println("false")
 5|    }
```

插入此分号后，此代码块将和下者等价：

```
 1|    switch alwaysFalse(); true {
 2|    case true: fmt.Println("true")
 3|    case false: fmt.Println("false")
 4|    }
```

这就是它输出`true`的原因。

常使用`go fmt`和`go vet`命令来格式化和发现可能的逻辑错误是一个好习惯。

下面是一个很少见的情形，此情形中所示的代码看上去是合法的，但是实际上是编译不通过的。

```
 1| func f() {
 2|    switch x {
 3|    case 1:
 4|    {
 5|       goto A
 6|       A: // 这里编译没问题
 7|    }
 8|    case 2:
 9|       goto B
10|       B: // syntax error: 跳转标签后缺少语句
11|    case 0:
12|       goto C
13|       C: // 这里编译没问题
14|    }
15| }
```

编译错误信息表明跳转标签的声明之后必须跟一条语句。 但是，看上去，上例中的三个标签声明没什么不同，它们都没有跟随一条语句。 那为什么只有`B:`标签声明是不合法的呢？ 原因是，根据上述第二条分号自动插入规则，编译器将在`A:`和`C:`标签声明之后的右大括号`}`字符之前插入一个分号，如下所示：

```
 1| func f(x int) {
 2|    switch x {
 3|    case 1:
 4|    {
 5|       goto A
 6|       A:
 7|    ;} // 一个分号插入到了这里
 8|    case 2:
 9|       goto B
10|       B: // syntax error: 跳转标签后缺少语句
11|    case 0:
12|       goto C
13|       C:
14|    ;} // 一个分号插入到了这里
15| }
```

一个单独的分号实际上表示一条空语句（第11章）。 这就意味着`A:`和`C:`标签声明之后确实跟随了一条语句，所以它们是合法的。 而`B:`标签声明跟随的`case 0:`不是一条语句，所以它是不合法的。

我们可以在`B:`标签声明之后手动插入一个分号使之变得合法。

逗号，从不会被自动插入

一些包含多个类似项目的语法形式多用逗号`,`来做为这些项目之间的分割符，比如组合字面量和函数参数列表等。 在这样的一个语法形式中，最后一个项目后总可以跟一个可选的逗号。 如果此逗号为它所在代码行的最后一个有效字符，则此逗号是必需的；否则，此逗号可以省略。 编译器在任何情况下都不会自动插入逗号。

比如，下面的代码是合法的：

```
 1| func f1(a int, b string,) (x bool, y int,) {
 2|    return true, 789
 3| }
 4| var f2 func (a int, b string) (x bool, y int)
 5| var f3 func (a int, b string, // 最后一个逗号是必需的
 6| ) (x bool, y int,             // 最后一个逗号是必需的
 7| )
```

```
 8| var _ = []int{2, 3, 5, 7, 9,} // 最后一个逗号是可选的
 9| var _ = []int{2, 3, 5, 7, 9,  // 最后一个逗号是必需的
10| }
11| var _ = []int{2, 3, 5, 7, 9}
12| var _, _ = f1(123, "Go",) // 最后一个逗号是可选的
13| var _, _ = f1(123, "Go",  // 最后一个逗号是必需的
14| )
15| var _, _ = f1(123, "Go")
16| // 对于显式转换也是一样的:
17| var _ = string(65,) // 最后一个逗号是可选的
18| var _ = string(65,  // 最后一个逗号是必需的
19| )
```

而下面这段代码是不合法的，因为编译器将自动在每一行的行尾插入一个分号（除了第二行）。 其中三行在插入分号后将导致编译错误。

```
 1| func f1(a int, b string,) (x bool, y int // error
 2| ) {
 3|    return true, 789
 4| }
 5| var _ = []int{2, 3, 5, 7, 9 // error: unexpected newline
 6| }
 7| var _, _ = f1(123, "Go" // error: unexpected newline
 8| )
```

结束语

最后，根据上面的解释，在这里描述一下Go代码中的断行规则。

在Go代码中，以下断行是没问题的（不影响程序行为的）：

- 在除了`break`、`continue`和`return`这几个跳转关键字之外的任何关键字之后断行，或者在不跟随标签的`break`和`continue`关键字以及不跟随返回值的`return`关键字之后断行；
- 在（显式输入的或者隐式被编译器插入的）分号`;`之后断行；
- 在不会导致新的隐式分号被编译器插入的情况下断行。

和很多Go中的其它设计细节一样，Go代码断行规则设计的评价也是褒贬不一。 有些程序员不太喜欢这样的断行规则，因为这样的规则限制了代码风格的自由度。 但是这些规则不但使得代码编译速度大大提高，另一方面也使得不同Go程序员写出的代码风格大体一致，从而相互可以比较轻松地读懂对方的代码。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

更多关于延迟函数调用的知识点

延迟调用函数已经在前面介绍过了（第13章）。 限于当时对Go的了解程度，很多延迟调用函数相关的细节和用例并没有在之前的文章中提及。 这些细节和用例将在本文中列出。

很多有返回值的内置函数是不能被延迟调用的

在Go中，自定义函数的调用的返回结果都可以被舍弃。 但是，大多数内置函数（除了copy和recover）的调用的返回结果都不可以舍弃（第49章）（至少对于Go 1.23来说是如此）。 另一方面，我们已经了解到延迟函数调用的所有返回结果必须都舍弃掉。 所以，很多内置函数是不能被延迟调用的。

幸运的是，在实践中，延迟调用内置函数的需求很少见。 根据我的经验，只有append函数有时可能会需要被延迟调用。 对于这种情形，我们可以延迟调用一个调用了append函数的匿名函数来满足这个需求。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     s := []string{"a", "b", "c", "d"}
 7|     defer fmt.Println(s) // [a x y d]
 8|     // defer append(s[:1], "x", "y") // 编译错误
 9|     defer func() {
10|         _ = append(s[:1], "x", "y")
11|     }()
12| }
```

延迟调用的函数值的估值时刻

一个被延迟调用的函数值是在其调用被推入延迟调用队列之前被估值的。 例如，下面这个例子将输出false。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
```

```
 6|    var f = func () {
 7|       fmt.Println(false)
 8|    }
 9|    defer f()
10|    f = func () {
11|       fmt.Println(true)
12|    }
13| }
```

一个被延迟调用的函数值可能是一个nil函数值。这种情形将导致一个恐慌。 对于这种情形，恐慌产生在此延迟调用被执行而不是被推入延迟调用队列的时候。 一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    defer fmt.Println("此行可以被执行到")
 7|    var f func() // f == nil
 8|    defer f()    // 将产生一个恐慌
 9|    fmt.Println("此行可以被执行到")
10|    f = func() {} // 此行不会阻止恐慌产生
11| }
```

延迟方法调用的属主实参的估值时刻

前面的文章曾经解释过：一个延迟调用的实参也是在此调用被推入延迟调用队列时估值的（第13章）。 方法的属主实参也不例外。比如，下面这个程序将打印出 1342。

```
 1| package main
 2|
 3| type T int
 4|
 5| func (t T) M(n int) T {
 6|   print(n)
 7|   return t
 8| }
 9|
10| func main() {
11|    var t T
12|    // t.M(1)是方法调用M(2)的属主实参，因此它
13|    // 将在M(2)调用被推入延迟调用队列时被估值。
```

```
14|    defer t.M(1).M(2)
15|    t.M(3).M(4)
16| }
```

延迟调用使得代码更简洁和鲁棒

一个例子：

```
 1| import "os"
 2|
 3| func withoutDefers(filepath string, head, body []byte)
error {
 4|    f, err := os.Open(filepath)
 5|    if err != nil {
 6|       return err
 7|    }
 8|
 9|    _, err = f.Seek(16, 0)
10|    if err != nil {
11|       f.Close()
12|       return err
13|    }
14|
15|    _, err = f.Write(head)
16|    if err != nil {
17|       f.Close()
18|       return err
19|    }
20|
21|    _, err = f.Write(body)
22|    if err != nil {
23|       f.Close()
24|       return err
25|    }
26|
27|    err = f.Sync()
28|    f.Close()
29|    return err
30| }
31|
32| func withDefers(filepath string, head, body []byte) error {
33|    f, err := os.Open(filepath)
34|    if err != nil {
35|       return err
```

```
36|    }
37|    defer f.Close()
38|
39|    _, err = f.Seek(16, 0)
40|    if err != nil {
41|       return err
42|    }
43|
44|    _, err = f.Write(head)
45|    if err != nil {
46|       return err
47|    }
48|
49|    _, err = f.Write(body)
50|    if err != nil {
51|       return err
52|    }
53|
54|    return f.Sync()
55| }
```

上面哪个函数看上去更简洁？显然，第二个使用了延迟调用的函数，虽然只是简洁了些许。 另外第二个函数将导致更少的bug，因为第一个函数中含有太多的`f.Close()`调用，从而有较高的几率漏掉其中一个。

下面是另外一个延迟调用使得代码更鲁棒的例子。 如果`doSomething`函数产生一个恐慌，则函数`f2`在退出时将导致互斥锁未解锁。 所以函数`f1`更鲁棒。

```
 1| var m sync.Mutex
 2|
 3| func f1() {
 4|    m.Lock()
 5|    defer m.Unlock()
 6|    doSomething()
 7| }
 8|
 9| func f2() {
10|    m.Lock()
11|    doSomething()
12|    m.Unlock()
13| }
```

延迟调用可能会导致性能损失

延迟调用并非没有缺点。对于早于1.13版本的官方标准编译器来说，延迟调用将导致一些性能损失。 从Go官方工具链1.13版本开始，官方标准编译器对一些常见的延迟调用场景做了很大的优化。 因此，一般我们不必太在意延迟调用导致的性能损失。感谢Dan Scales实现了此优化。

延迟调用导致的暂时性内存泄露

一个较大的延迟调用队列可能会消耗很多内存。 另外，某些资源可能因为某些调用被延迟的太久而未能被及时释放。

比如，如果下面的例子中的函数需要处理大量的文件，则在此函数退出之前，将有大量的文件句柄得不到释放。

```
 1| func writeManyFiles(files []File) error {
 2|    for _, file := range files {
 3|       f, err := os.Open(file.path)
 4|       if err != nil {
 5|          return err
 6|       }
 7|       defer f.Close()
 8|
 9|       _, err = f.WriteString(file.content)
10|       if err != nil {
11|          return err
12|       }
13|
14|       err = f.Sync()
15|       if err != nil {
16|          return err
17|       }
18|    }
19|
20|    return nil
21| }
```

对于这种情形，我们应该使用一个匿名函数将需要及时执行延迟的调用包裹起来。比如，上面的函数可以改进为如下：

```
 1| func writeManyFiles(files []File) error {
 2|    for _, file := range files {
 3|       if err := func() error {
 4|          f, err := os.Open(file.path)
 5|          if err != nil {
 6|             return err
```

```
 7|          }
 8|          defer f.Close() // 将在此循环步内执行
 9|
10|          _, err = f.WriteString(file.content)
11|          if err != nil {
12|             return err
13|          }
14|
15|          return f.Sync()
16|       }(); err != nil {
17|          return err
18|       }
19|    }
20|
21|    return nil
22| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

一些恐慌/恢复用例

恐慌和恢复（panic/recover）已经在之前的文章中介绍过了（第13章）。 下面将展示一些恐慌/恢复用例。

用例1：避免恐慌导致程序崩溃

这可能是最常见的panic/recover用例了。 此用例广泛地使用于并发程序中，尤其是响应大量用户请求的应用。

一个例子：

```
 1| package main
 2|
 3| import "errors"
 4| import "log"
 5| import "net"
 6|
 7| func main() {
 8|    listener, err := net.Listen("tcp", ":12345")
 9|    if err != nil {
10|       log.Fatalln(err)
11|    }
12|    for {
13|       conn, err := listener.Accept()
14|       if err != nil {
15|          log.Println(err)
16|       }
17|       // 在一个新协程中处理客户端连接。
18|       go ClientHandler(conn)
19|    }
20| }
21|
22| func ClientHandler(c net.Conn) {
23|    defer func() {
24|       if v := recover(); v != nil {
25|          log.Println("捕获了一个恐慌：", v)
26|          log.Println("防止了程序崩溃")
27|       }
28|       c.Close()
29|    }()
30|    panic("未知错误") // 演示目的产生的一个恐慌
31| }
```

运行此服务器程序，并在另一个终端窗口运行`telnet localhost 12345`，我们可以观察到服务器程序不会因为客户连接处理协程中的产生的恐慌而导致崩溃。

如果我们在上例中不捕获客户连接处理协程中的潜在恐慌，则这样的恐慌将使整个程序崩溃。

用例2：自动重启因为恐慌而退出的协程

当在一个协程将要退出时，程序侦测到此协程是因为一个恐慌而导致此次退出时，我们可以立即重新创建一个相同功能的协程。 一个例子：

```
 1| package main
 2|
 3| import "log"
 4| import "time"
 5|
 6| func shouldNotExit() {
 7|    for {
 8|       time.Sleep(time.Second) // 模拟一个工作负载
 9|       // 模拟一个未预料到的恐慌。
10|       if time.Now().UnixNano() & 0x3 == 0 {
11|          panic("unexpected situation")
12|       }
13|    }
14| }
15|
16| func NeverExit(name string, f func()) {
17|    defer func() {
18|       if v := recover(); v != nil { // 侦测到一个恐慌
19|          log.Printf("协程%s崩溃了，准备重启一个", name)
20|          go NeverExit(name, f) // 重启一个同功能协程
21|       }
22|    }()
23|    f()
24| }
25|
26| func main() {
27|    log.SetFlags(0)
28|    go NeverExit("job#A", shouldNotExit)
29|    go NeverExit("job#B", shouldNotExit)
30|    select{} // 永久阻塞主线程
31| }
```

用例3：使用panic/recover函数调用模拟长程跳转

有时，我们可以使用panic/recover函数调用来模拟跨函数跳转，尽管一般这种方式并不推荐使用。 这种跳转方式的可读性不高，代码效率也不是很高，唯一的好处是它有时可以使代码看上去不是很啰嗦。

在下面这个例子中，一旦一个恐慌在一个内嵌函数中产生，当前协程中的执行将会跳转到延迟调用处。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    n := func () (result int)  {
 7|       defer func() {
 8|          if v := recover(); v != nil {
 9|             if n, ok := v.(int); ok {
10|                result = n
11|             }
12|          }
13|       }()
14|
15|       func () {
16|          func () {
17|             func () {
18|                // ...
19|                panic(123) // 用恐慌来表示成功返回
20|             }()
21|             // ...
22|          }()
23|       }()
24|       // ...
25|       return 0
26|    }()
27|    fmt.Println(n) // 123
28| }
```

用例4：使用panic/recover函数调用来减少错误检查代码

一个例子：

```
 1| func doSomething() (err error) {
 2|    defer func() {
 3|       switch e := recover().(type) {
 4|       case nil:
 5|       case error:
 6|          err = e
 7|       default:
 8|          panic(e) // 重新抛出此恐慌
 9|       }
10|    }()
11|
12|    doStep1()
13|    doStep2()
14|    doStep3()
15|    doStep4()
16|    doStep5()
17|
18|    return
19| }
20|
21| // 在现实中，各个doStepN函数的原型可能不同。
22| // 这里，每个doStepN函数的行为如下：
23| // * 如果已经成功，则调用panic(nil)来制造一个恐慌
24| //   以示不需继续；
25| // * 如果本步失败，则调用panic(err)来制造一个恐慌
26| //   以示不需继续；
27| // * 不制造任何恐慌表示继续下一步。
28| func doStepN() {
29|    ...
30|    if err != nil {
31|       panic(err)
32|    }
33|    ...
34|    if done {
35|       panic(nil)
36|    }
37| }
```

下面这段同功能的代码比上面这段代码看上去要啰嗦一些。

```
 1| func doSomething() (err error) {
 2|    shouldContinue, err := doStep1()
 3|    if !shouldContinue {
```

```
 4|       return err
 5|    }
 6|    shouldContinue, err = doStep2()
 7|    if !shouldContinue {
 8|       return err
 9|    }
10|    shouldContinue, err = doStep3()
11|    if !shouldContinue {
12|       return err
13|    }
14|    shouldContinue, err = doStep4()
15|    if !shouldContinue {
16|       return err
17|    }
18|    shouldContinue, err = doStep5()
19|    if !shouldContinue {
20|       return err
21|    }
22|
23|    return
24| }
25|
26| // 如果返回值err不为nil，则shouldContinue一定为true。
27| // 如果shouldContinue为true，返回值err可能为nil或者非nil。
28| func doStepN() (shouldContinue bool, err error) {
29|    ...
30|    if err != nil {
31|       return false, err
32|    }
33|    ...
34|    if done {
35|       return false, nil
36|    }
37|    return true, nil
38| }
```

但是，这种panic/recover函数调用的使用方式一般并不推荐使用，因为它的效率略低一些，并且这种用法不太符合Go编程习俗。

另外需要注意的是：从Go 1.21开始，一个panic(nil)调用将变得和panic(new(runtime.PanicNilError))等价（https://github.com/golang/go/issues/25448）。所以，从Go 1.21开始，上面代码中的延迟函数调用应该被重写为：

```
 1| import "runtime"
 2|
 3| ...
 4|
 5| func doSomething() (err error) {
 6|    defer func() {
 7|       switch e := recover().(type) {
 8|       case nil, *runtime.PanicNilError:
 9|       case error:
10|          err = e
11|       default:
12|          panic(e) // 重新抛出此恐慌
13|       }
14|    }()
15|
16|    doStep1()
17|    ...
18| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

详解恐慌和恢复原理

恐慌和恢复原理已经在前面的文章中介绍过了（第13章）。 一些恐慌和恢复用例也在上一篇文章中（第30章）得到了展示。 本文将详细解释一下恐慌和恢复原理。函数调用的退出阶段也将被一并详细解释。

函数调用的退出阶段

在Go中，一个函数调用在其退出完毕之前可能将经历一个退出阶段。 在此退出阶段，所有在执行此函数调用期间被推入延迟调用队列的延迟函数调用将按照它们的推入顺序的逆序被执行。 当这些延迟函数调用都退出完毕之后，此函数调用的退出阶段也就结束了，或者说此函数调用也退出完毕了，

退出阶段有时候也被称为返回阶段。

一个函数调用可能通过三种途径进入它的退出阶段：

1. 此调用正常返回；
2. 当此调用中产生了一个恐慌；
3. 当`runtime.Goexit`函数在此调用中被调用并且退出完毕。

比如，在下面这段代码中，

- 函数`f0`或者`f1`的一个调用将在它正常返回后进入它的退出阶段；
- 函数`f2`的一个调用将在“被零除”恐慌产生之后进入它的退出阶段；
- 函数`f3`的一个调用将在其中的`runtime.Goexit`函数调用退出完毕之后进入它的退出阶段。

```
 1| import (
 2|     "fmt"
 3|     "runtime"
 4| )
 5|
 6| func f0() int {
 7|     var x = 1
 8|     defer fmt.Println("正常退出：", x)
 9|     x++
10|     return x
11| }
12|
13| func f1() {
14|     var x = 1
15|     defer fmt.Println("正常退出：", x)
```

```
16|    x++
17| }
18|
19| func f2() {
20|    var x, y = 1, 0
21|    defer fmt.Println("因恐慌而退出：", x)
22|    x = x / y // 将产生一个恐慌
23|    x++       // 执行不到
24| }
25|
26| func f3() int {
27|    x := 1
28|    defer fmt.Println("因Goexit调用而退出：", x)
29|    x++
30|    runtime.Goexit()
31|    return x+x // 执行不到
32| }
```

顺便说一下，一般`runtime.Goexit()`函数不希望在主协程中调用。

函数调用关联恐慌和Goexit信号

当一个函数调用中直接产生了一个恐慌的时候，我们可以认为此（尚未被恢复的）恐慌将和此函数调用相关联起来。 类似地，当一个函数调用直接调用了`runtime.Goexit`函数，则`runtime.Goexit`函数返回完毕之后，我们可以认为一个Goexit信号将和此函数调用相关联起来。 按照上一节中的解释，当一个恐慌或者一个Goexit信号和一个函数调用相关联之后，此函数调用将立即进入它的退出阶段。

我们已经了解到恐慌是可以被恢复的（第13章）。 但是，Goexit信号是不能被取消的。

在任何一个给定时刻，一个函数调用最多只能和一个未恢复的恐慌相关联。 如果一个调用正和一个未恢复的恐慌相关联，则

- 在此恐慌被恢复之后，此调用将不再和任何恐慌相关联。
- 当在此函数调用中产生了一个新的恐慌，此新恐慌将替换原来的未被恢复的恐慌做为和此函数调用相关联的恐慌。

比如，在下面这个例子中，最终被恢复的恐慌是恐慌3。它是最后一个和`main`函数调用相关联的恐慌。

```
1| package main
2|
```

```
 3| import "fmt"
 4|
 5| func main() {
 6|    defer func() {
 7|       fmt.Println(recover()) // 3
 8|    }()
 9|
10|    defer panic(3) // 将替换恐慌2
11|    defer panic(2) // 将替换恐慌1
12|    defer panic(1) // 将替换恐慌0
13|    panic(0)
14| }
```

因为Goexit信号不可被取消，争论一个函数调用是否最多只能和一个Goexit信号相关联是没有意义和没有必要的。

在某个时刻，一个协程中可能共存多个未被恢复的恐慌，尽管这在实际编程中并不常见。 每个未被恢复的恐慌和此协程的调用堆栈中的一个尚未退出的函数调用相关联。 当仍和一个未被恢复的恐慌相关联的一个内层函数调用退出完毕之后，此未被恢复的恐慌将传播到调用此内层函数调用的外层函数调用中。 这和在此外层函数调用中直接产生一个新的恐慌的效果是一样的。也就是说，

- 如果此外层函数已经和一个未被恢复的旧恐慌相关联，则传播出来的新恐慌将替换此旧恐慌并和此外层函数调用相关联起来。 对于这种情形，此外层函数调用肯定已经进入了它的退出阶段（刚提及的内层函数肯定就是被延迟调用的），这时延迟调用队列中的下一个延迟调用将被执行。
- 如果此外层函数尚未和一个未被恢复的旧恐慌相关联，则传播出来的恐慌将和此外层函数调用相关联起来。 对于这种情形，如果此外层函数调用尚未进入它的退出阶段，则它将立即进入。

所以，当一个协程完成完毕后，此协程中最多只有一个尚未被恢复的恐慌。 如果一个协程带着一个尚未被恢复的恐慌退出完毕，则这将使整个程序崩溃，此恐慌信息将在程序崩溃的时候被打印出来。

在一个函数调用被执行的起始时刻，此调用将没有任何恐慌和Goexit信号和它相关联，这个事实和此函数调用的外层调用是否已经进入退出阶段无关。 当然，在此函数调用的执行过程中，恐慌可能产生，`runtime.Goexit`函数也可能被调用，因此恐慌和Goexit信号以后可能和此调用相关联起来。

下面这个例子程序在运行时将崩溃，因为新开辟的协程在退出完毕时仍带有一个未被恢复的恐慌。

```
 1| package main
 2|
 3| func main() {
```

```
 4|    // 新开辟一个协程。
 5|    go func() {
 6|       // 一个匿名函数调用。
 7|       // 当它退出完毕时，恐慌2将传播到此新协程的入口
 8|       // 调用中，并且替换掉恐慌0。恐慌2永不会被恢复。
 9|       defer func() {
10|          // 上一个例子中已经解释过了：恐慌2将替换恐慌1.
11|          defer panic(2)
12|
13|          // 当此匿名函数调用退出完毕后，恐慌1将传播到刚
14|          // 提到的外层匿名函数调用中并与之关联起来。
15|          func () {
16|             panic(1)
17|             // 在恐慌1产生后，此新开辟的协程中将共存
18|             // 两个未被恢复的恐慌。其中一个（恐慌0）
19|             // 和此协程的入口函数调用相关联；另一个
20|             // （恐慌1）和当前这个匿名调用相关联。
21|          }()
22|       }()
23|       panic(0)
24|    }()
25|
26|    select{}
27| }
```

此程序的输出（当使用标准编译器1.23版本编译）：

```
panic: 0
   panic: 1
   panic: 2

...
```

此输出的格式并非很完美，它容易让一些程序员误认为恐慌0是最终未被恢复的恐慌。而事实上，恐慌2才是最终未被恢复的恐慌。

类似地，当一个和Goexit信号相关联的内层函数调用退出完毕后，此Goexit信号也将传播到外层函数调用中，并和外层函数调用相关联起来。 如果外层函数调用尚未进入退出阶段，则其将立即进入。

当一个Goexit信号和一个函数调用相关联起来的时候，如果此函数调用正在和一个未被恢复的恐慌相关联着，则此恐慌将被恢复。 比如下面这个程序将正常退出并打印出`<nil>`，因为恐慌`bye`被Goexit信号恢复了。

```
 1| package main
 2|
```

```
 3| import (
 4|    "fmt"
 5|    "runtime"
 6| )
 7|
 8| func f() {
 9|    defer func() {
10|       fmt.Println(recover())
11|    }()
12|
13|    // 此调用产生的Goexit信号恢复之前产生的恐慌。
14|    defer runtime.Goexit()
15|    panic("bye")
16| }
17|
18| func main() {
19|    go f()
20|
21|    for runtime.NumGoroutine() > 1 {
22|       runtime.Gosched()
23|    }
24| }
```

一些recover调用相当于空操作（No-Op）

内置recover函数必须在合适的位置调用才能发挥作用；否则，它的调用相当于空操作。 比如，在下面这个程序中，没有一个recover函数调用恢复了恐慌bye。

```
 1| package main
 2|
 3| func main() {
 4|    defer func() {
 5|       defer func() {
 6|          recover() // 空操作
 7|       }()
 8|    }()
 9|    defer func() {
10|       func() {
11|          recover() // 空操作
12|       }()
13|    }()
14|    func() {
15|       defer func() {
16|          recover() // 空操作
```

```
17|       }()
18|    }()
19|    func() {
20|       defer recover() // 空操作
21|    }()
22|    func() {
23|       recover() // 空操作
24|    }()
25|    recover()       // 空操作
26|    defer recover() // 空操作
27|    panic("bye")
28| }
```

我们已经知道下面这个recover调用是有作用的。

```
 1| package main
 2|
 3| func main() {
 4|    defer func() {
 5|       recover() // 将恢复恐慌"byte"
 6|    }()
 7|
 8|    panic("bye")
 9| }
```

那么为什么本节中的第一个例子中的所有recover调用都不起作用呢？ 让我们先看看当前版本的Go白皮书（https://golang.org/ref/spec#Handling_panics）是怎么说的：

> 在下面的情况下，recover函数调用的返回值为nil：
>
> - 传递给相应panic函数调用的实参为nil；
> - 当前协程并没有处于恐慌状态；
> - recover函数并未直接在一个延迟函数调用中调用。

上一篇文章中提供了一个第一种情况的例子（第30章）。

本节中的第一个例子中的大多recover调用要么符合Go白皮书中描述的第二种情况，要么符合第三种情况，除了第一个recover调用。 是的，当前版本的白皮书中的描述并不准确。第三种情况应该更精确地描述为：

- recover函数并未直接被一个延迟函数调用所直接调用， 或者它直接被一个延迟函数调用所直接调用但是此延迟调用没有被和期望被恢复的恐慌相关联的函数调用所直接调用。

在本节中的第一个例子中，期望被恢复的恐慌和main函数调用相关联。 第一个recover调用确实被一个延迟函数调用所直接调用，但是此延迟函数调用并没有被main函数直接调用。 这就是为什么此recover调用是一个空操作的原因。

事实上，当前版本的白皮书也没有解释清楚为什么下面这个例子中的第二个recover调用（按照代码行顺序）没有起作用。 此调用本期待用来恢复恐慌1。

```
 1| // 此程序将带着未被恢复的恐慌1而崩溃退出。
 2| package main
 3|
 4| func demo() {
 5|    defer func() {
 6|       defer func() {
 7|          recover() // 此调用将恢复恐慌2
 8|       }()
 9|
10|       defer recover() // 空操作
11|
12|       panic(2)
13|    }()
14|    panic(1)
15| }
16|
17| func main() {
18|    demo()
19| }
```

当前版本的白皮书没提到的一点是：每个recover调用都试图恢复当前协程中最新产生的且尚未恢复的恐慌。 当然，如果这个假设中的最新产生的且尚未恢复的恐慌不存在，则此recover调用是一个空操作。

Go运行时认为上例中的第二个recover调用试图恢复最新产生的尚未恢复的恐慌，即恐慌2。 而此时和恐慌2相关联的函数调用为此第二个recover调用的直接调用者，即外层的延迟函数调用。 此第二个recover调用并没有被外层的延迟函数调用所直接调用的某个延迟函数调用所调用； 相反，它直接被外层的延迟函数调用所调用。这就是为什么此第二个recover调用是一个空操作的原因。

总结

好了，到此我们可以对哪些recover调用会起作用做一个简短的描述：

> 一个recover调用只有在它的直接外层调用（即recover调用的父调用）是一个延迟调用，并且此延迟调用（即父调用）的直接外层调用（即

> `recover`调用的爷调用）和当前协程中最新产生并且尚未恢复的恐慌相关联时才起作用。 一个有效的`recover`调用将最新产生并且尚未恢复的恐慌和与此恐慌相关联的函数调用（即爷调用）剥离开来，并且返回当初传递给产生此恐慌的`panic`函数调用的参数。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

代码块和标识符作用域

本文将解释代码块和标识符的作用域。

（注意：本文中描述的代码块的层级关系和Go白皮书中有所不同。）

代码块

Go代码中有四种代码块。

- 万物代码块（the universe code block）。一个程序只有一个万物代码块，它包含着一个程序中所有的代码；
- 包代码块（package code block）。一个包代码块含着一个代码包中的所有代码，但不包括此代码包中的源代码文件中的所有引入声明。
- 文件代码块（file code block）。一个文件代码块包含着一个源文件中的所有代码，包括此文件中的包引入声明。
- 局部代码块（local code block）。一般说来，一对大括号`{}`中的代码形成了一个局部代码块。 但是也有一些局部代码块并不包含在一对大括号中，这样的代码块称为隐式代码块，而包含在一对大括号中的局部代码块称为显式代码块。 组合字面量中的大括号和代码块无关。

各种控制流程中的一些关键字跟随着一些隐式局部代码块：

- 一个`if`、`switch`或者`for`关键字跟随着两个内嵌在一起的局部代码块。 其中一个代码块是隐式的，另一个是显式的，此显式的代码块内嵌在此隐式代码块之中。 如果这样的一个关键字跟随着一个变量短声明形式，则被声明的变量声明在此隐式代码块中。
- 一个`else`关键字可以跟随着一个显式或者隐式代码块。此显式或者隐式代码块内嵌在跟随在对应`if`关键字后的隐式代码块中。 如果此`else`关键字立即跟随着另一个`if`关键字，则跟随在此`else`关键字后的代码块可以为隐式的，否则，此代码块必须为显式的。
- 一个`select`关键字跟随着一个显式局部代码块。
- 每个`case`和`default`关键字后跟随着一个隐式代码块，此隐式代码块内嵌在对应的`switch`或者`select`关键字后跟随的显式代码块中。

不内嵌在任何其它局部代码块中的局部代码块称为顶层（或者包级）局部代码块。顶层局部代码块肯定都是函数体。

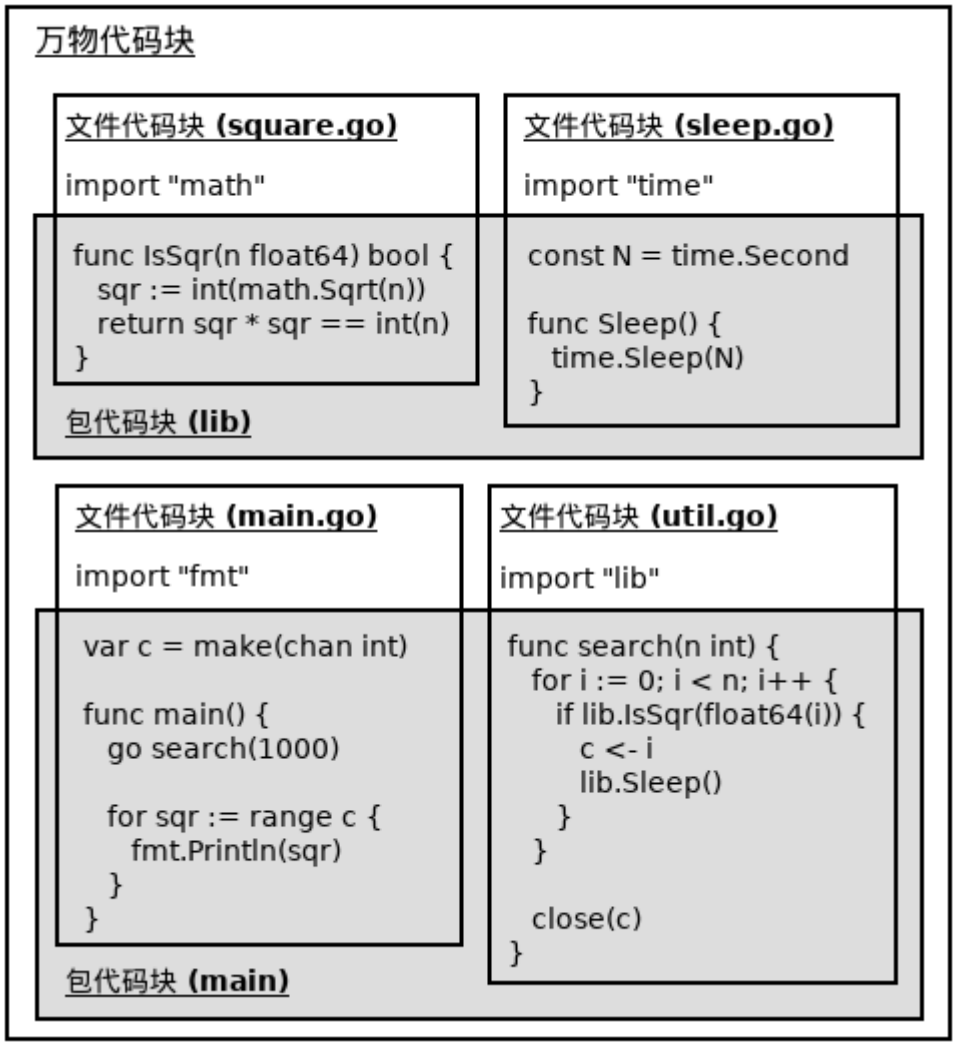
注意，一个函数声明中的输入参数和输出结果变量都被看作是声明在此函数体代码块内，虽然看上去它们好像声明在函数体代码块之外。

各种代码块的层级关系：

- 所有的包代码块均直接内嵌在万物代码块中；
- 所有的文件代码块也均直接内嵌在万物代码块中；（注意：`go/*` 标准库认为文件代码块内嵌在包代码块中。）
- 每个顶层局部代码块同时直接内嵌在一个包代码块和一个文件代码块中；（注意：`go/*` 标准库认为顶层局部代码块内嵌在文件代码中。）
- 一个非顶层局部代码块肯定直接内嵌在另一个局部代码块中。

（本文和`go/*` 标准库的解释有所不同的原因是为了让下面对标识符遮挡的解释更加简单和清楚。）

下面是一张展示上述代码块层级关系的图片：



代码块主要用来解释各种代码元素声明中的标识符的可声明位置和作用域。

各种代码元素的可声明位置

我们可以声明六种代码元素：

- 包引入；

- 定义类型和类型别名；
- 具名常量；
- 变量；
- 函数；
- 跳转标签。

在一个代码元素的声明中，一个标识符和一个代码元素绑定在了一起。 或者说，在此声明中，被声明的代码元素将被赋予此标识符做为它的名称。 此后，我们就可以用此标识符来代表此代码元素。

下标展示了各种代码元素可以被直接声明在何种代码块中：

	万物代码块	包代码块	文件代码块	局部代码块
预声明的（即内置的）代码元素[1]	可以			
包引入			可以	
定义类型和类型别名（不含内置的）		可以	可以	可以
具名常量（不含内置常量）		可以	可以	可以
变量（不含内置变量）[2]		可以	可以	可以
函数（不含内置函数）		可以	可以	
跳转标签				可以

[1] 预声明代码元素展示在builtin标准库（https://golang.google.cn/pkg/builtin/）中。
[2] 不包括结构体字段变量声明。

所以，

- 包引入不能声明在包代码块和局部代码块中；
- 函数不能被声明在局部代码块中；（匿名函数可以定义在局部代码块中，但它们不属于元素声明。）
- 跳转标签只能被声明在局部代码块中。

请注意：

- 如果包含两个代码元素声明的最内层代码块为同一个，则这两个代码元素不能同名。
- 声明在一个包中的一个包级代码元素的名称不能和此包中任何源文件中的包引入名同名。 或者反过来说更容易理解：一个包中的任何包引入名不能和此包中的任何包级代码元素的名称重名。 此规则可能会在以后被放宽（https://github.com/golang/go/issues/7429）。
- 如果包含两个跳转标签的最内层函数体为同一个，则这两个标签不能同名；
- 一个跳转标签的所有引用必须处于包含此跳转标签声明的最内层函数体代码块内；

- 各种控制流程中的隐式代码块对元素声明有特殊的要求。 一般说来，声明语句不允许出现在这样的隐式代码块中，除了一些变量短声明：
  - 每个`if`、`switch`或者`for`关键字后可以紧跟着一条变量短声明语句；
  - 一个`select`控制流程中的每个`case`关键字后可以紧跟着一条变量短声明语句。

（顺便说一下，`go/*`标准库代码包认为文件代码块中只能包含包引入声明。）

声明在包代码块中并且在所有局部代码块之外的代码元素称为包级（package-level）元素。 包级元素可以是具名常量、变量、函数、定义类型或类型别名。

代码元素标识符的作用域

一个代码元素标识符的作用域是指此标识符可被识别的代码范围（或可见范围）。

不考虑本文最后一节将要解释的标识符遮挡，Go白皮书[这样描述](https://golang.google.cn/ref/spec#Declarations_and_scope)（https://golang.google.cn/ref/spec#Declarations_and_scope）各种代码元素的标识符的作用域：

- 内置代码元素标识符的作用域为整个万物代码块；
- 一个包引入声明标识符的作用域为包含它的声明的文件代码块；
- 直接声明在一个包代码块中的一个常量、类型、变量或者函数（不包括方法）的标识符的作用域为此包代码块；
- 在函数体中声明的一个常量或者变量的标识符的作用域起始于此常量或者变量的描述（specification）的结尾（对于变量短声明，为此变量声明的结尾），并终止于包含此常量或者变量的声明的最内层代码块的结尾；
- 一个函数参数（包括方法属主参数）和结果标识符的作用域为其对应函数体局部代码块；
- 在函数体中声明的一个类型的标识符的作用域起始于此类型的描述中它的标识符的结尾，并终止于包含此类型的声明的最内层代码块的结尾；
- 一个跳转标签的标识符的作用域为包含此标签的声明的最内层函数体代码块，但要排除掉此内嵌在此最内层函数体代码块中的各个匿名函数体代码块。
- 关于类型参数的作用域，请阅读[《Go自定义泛型101》](https://gfw.go101.org/generics/555-type-constraints-and-parameters.html#type-parameter-scope)（https://gfw.go101.org/generics/555-type-constraints-and-parameters.html#type-parameter-scope）一书。

空标识符没有作用域。

（注意，预声明的`iota`标识符只能使用在常量声明中。）

你可能已经注意到了局部定义类型的作用域和其它局部元素（变量、常量和类型别名）的作用域的定义有微小的差别。 此差别体现在一个定义类型的声明中可以立即使用此定义类型的标识符。 下面这个例子展示了这一差异：

```
 1| package main
 2|
 3| func main() {
 4|    // var v int = v   // error: v未定义
 5|    // const C int = C // error: C未定义
 6|    /*
 7|    type T = struct {
 8|       *T    // error: 不可循环引用
 9|       x []T // error: 不可循环引用
10|    }
11|    */
12|
13|    // 下面所有的类型定义声明都是合法的。
14|    type T struct {
15|       *T
16|       x []T
17|    }
18|    type A [5]*A
19|    type S []S
20|    type M map[int]M
21|    type F func(F) F
22|    type Ch chan Ch
23|    type P *P
24|
25|    // ...
26|    var p P
27|    p = &p
28|    p = **********************p
29|    **********************p = p
30|
31|    var s = make(S, 3)
32|    s[0] = s
33|    s = s[0][0][0][0][0][0][0][0]
34|
35|    var m = M{}
36|    m[1] = m
37|    m = m[1][1][1][1][1][1][1][1]
38| }
```

注意：`fmt.Print(s)`和`fmt.Print(m)`调用都将导致恐慌（因为堆栈溢出）。

下面是一个展示了包级声明和局部声明的标识符的作用域差异的例子：

```
 1| package main
 2|
```

```
 3| // 下面这两行中各自等号左边和右边的标识符表示同一个代码元素。
 4| // 右边的标识符不是预声明的标识符。
 5| /*
 6| const iota = iota // error: 循环引用
 7| var true = true   // error: 循环引用
 8| */
 9|
10| var a = b // 可以使用其后声明的变量的标识符
11| var b = 123
12|
13| func main() {
14|     // 下面两行中右边的标识符为预声明的标识符。
15|     const iota = iota // ok
16|     var true = true   // ok
17|     _ = true
18|
19|     // 下面几行编译不通过。
20|     /*
21|     var c = d // 不能使用其后声明变量标识符
22|     var d = 123
23|     _ = c
24|     */
25| }
```

标识符遮挡

不考虑跳转标签，一个在外层代码块直接声明的标识符将被在内层代码块直接声明的相同标识符所遮挡。

跳转标签标识符不会被遮挡。

如果一个标识符被遮挡了，它的作用域将不包括遮挡它的标识符的作用域。

下面是一个有趣的例子。在此例子中，有6个变量均被声明为`x`。 一个在更深层代码块中声明的`x`遮挡了所有在外层声明的`x`。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| var p0, p1, p2, p3, p4, p5 *int
 6| var x = 9999 // x#0
 7|
 8| func main() {
```

```
 9|    p0 = &x
10|    var x = 888  // x#1
11|    p1 = &x
12|    for x := 70; x < 77; x++ {  // x#2
13|       p2 = &x
14|       x := x - 70 //  // x#3
15|       p3 = &x
16|       if x := x - 3; x > 0 { // x#4
17|          p4 = &x
18|          x := -x // x#5
19|          p5 = &x
20|       }
21|    }
22|
23|    // 9999 888 77 6 3 -3
24|    fmt.Println(*p0, *p1, *p2, *p3, *p4, *p5)
25| }
```

下面是另一个关于标识符遮挡和作用域的例子。此例子程序运行将输出`Sheep Goat`而不是`Sheep Sheep`。 请阅读其中的注释获取原因。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| var f = func(b bool) {
 6|    fmt.Print("Goat")
 7| }
 8|
 9| func main() {
10|    var f = func(b bool) {
11|       fmt.Print("Sheep")
12|       if b {
13|          fmt.Print(" ")
14|          f(!b) // 此f乃包级变量f也。
15|       }
16|    }
17|    f(true) // 此f为刚声明的局部变量f。
18| }
```

如果我们将上例更改为如下所示，则此程序将运行输出`Sheep Sheep`。

```
 1| func main() {
 2|    var f func(b bool)
 3|    f = func(b bool) {
```

```
 4|       fmt.Print("Sheep")
 5|       if b {
 6|          fmt.Print(" ")
 7|          f(!b) // 现在，此f变为局部变量f了。
 8|       }
 9|    }
10|    f(true)
11| }
```

在某些情况下，当一些标识符被内层的一个变量短声明中声明的变量所遮挡时，一些新手Go程序员会搞不清楚此变量短声明中声明的哪些变量是新声明的变量。 下面这个例子（含有bug）展示了Go编程中一个比较有名的陷阱。 几乎每个Go程序员在刚开始使用Go的时候都曾经掉入过此陷阱。

```
 1| package main
 2|
 3| import "fmt"
 4| import "strconv"
 5|
 6| func parseInt(s string) (int, error) {
 7|    n, err := strconv.Atoi(s)
 8|    if err != nil {
 9|       // 一些新手Go程序员会认为下一行中声明
10|       // 的err变量已经在外层声明过了。然而其
11|       // 实下一行中的b和err都是新声明的变量。
12|       // 此新声明的err遮挡了外层声明的err。
13|       b, err := strconv.ParseBool(s)
14|       if err != nil {
15|          return 0, err
16|       }
17|
18|       // 如果代码运行到这里，一些新手Go程序员
19|       // 期望着内层的nil err将被返回。但是其实
20|       // 返回是外层的非nil err。因为内层的err
21|       // 的作用域到外层if代码块结尾就结束了。
22|       if b {
23|          n = 1
24|       }
25|    }
26|    return n, err
27| }
28|
29| func main() {
30|    fmt.Println(parseInt("TRUE"))
31| }
```

程序输出：

```
1 strconv.Atoi: parsing "TRUE": invalid syntax
```

Go语言目前只有25个关键字（第5章）。 关键字不能被用做标识符。Go中很多常见的名称，比如int、bool、string、len、cap、nil等，并不是关键字，它们是预声明标识符。 这些预声明的标识符声明在万物代码块中，所以它们可以被声明在内层的相同标识符所遮挡。 下面是一个展示了预声明标识符被遮挡的古怪的例子。它编译和运行都没有问题。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5| )
 6|
 7| const len = 3      // 遮挡了内置函数len
 8| var true = 0       // 遮挡了内置常量true
 9| type nil struct {} // 遮挡了内置变量nil
10| func int(){}       // 遮挡了内置类型int
11|
12| func main() {
13|    fmt.Println("a weird program")
14|    var output = fmt.Println
15|
16|    var fmt = [len]nil{{}, {}, {}} // 遮挡了包引入fmt
17|    // var n = len(fmt) // error: len是一个常量
18|    var n = cap(fmt)    // 我们只好使用内置cap函数
19|
20|    // for关键字跟随着一个隐式代码块和一个显式代码块。
21|    // 变量短声明中的true遮挡了全局变量true。
22|    for true := 0; true < n; true++ {
23|       // 下面声明的false遮挡了内置常量false。
24|       var false = fmt[true]
25|       // 下面声明的true遮挡了循环变量true。
26|       var true = true+1
27|       // 下一行编译不通过，因为fmt是一个数组。
28|       // fmt.Println(true, false)
29|       output(true, false)
30|    }
31| }
```

输出结果：

```
a weird program
1 {}
2 {}
3 {}
```

是的，此例子是一个极端的例子。标识符遮挡是一个有用的特性，但是千万不要滥用之。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

表达式估值顺序规则

本文将解释各种情形下表达式（第11章）的估值顺序。

一个表达式将在其所依赖的其它表达式估值之后进行估值

这属于常识，没什么好解释的。一个显然的例子是一个表达式将在组成它的子表达式都估值之后才能进行估值。 比如，在一个函数调用`f(x, y[n])`中，

- `f()`将在它所依赖的子表达式`f`、`x`和`y[n]`估值之后进行估值；
- 表达式`y[n]`将在它所依赖的子表达式`n`和`y`估值之后进行估值。

另外，程序代码要素初始化顺序（第10章）章节中提供了一个关于包级变量初始化顺序的例子。

包级变量初始化顺序

在运行时刻，当一个包被加载的时候，不依赖于任何其它未初始化包级变量的未初始化包级变量将按照它们在代码中的声明顺序被初始化，直到此过程中不再有任何包级变量被初始化。 对于一个成功编译了的Go程序，当所有这样的过程结束之后，所有的包级变量都应该被初始化了。

在包级变量初始化过程中，呈现为空标识符的包级变量和其它包级变被同等对待。

举个例子，在下面的这个程序中，变量`a`依赖于变量`b`，变量`c`和`_`依赖于变量`a`。所以

1. 第一个被初始化的变量是`b`，因为它是第一个不依赖于其它变量的包级变量。
2. 第二个被初始化的变量是`a`，因为在变量`b`初始化之后，变量`a`是第一个不再依赖于未初始化包级变量的包级变量。.
3. 第三和第四个被初始化的变量是`_`和`c`。 在变量`a`和`b`初始化之后，标记变量`_`和`c`均不再依赖于未初始化包级变量。

```
1| package main
2|
3| import "fmt"
4|
5| var (
```

```
 6|    _ = f()
 7|    a = b / 2
 8|    b = 6
 9|    c = f()
10| )
11|
12| func f() int {
13|    a++
14|    return a
15| }
16|
17| func main() {
18|    fmt.Println(a, b, c) // 5 6 5
19| }
```

上面这个程序打印出`5 6 5`。

通过一个多值表达式源值来初始化的多个包级变量将被一起初始化。 比如，在包级变量声明`var x, y = f()`中，变量`x`和`y`将被一起初始化。 或者说，在它们的初始化之间不会有其它包级变量被初始化。

在初始化所有包级变量之前，含有多个源值表达式的包级变量声明将被拆解为多个单源值表达式变量声明。 比如

```
1| var m, n = expr1, expr2
```

和

```
1| var m = expr1
2| var n = expr2
```

是等价的。

如果一些包级变量之间存在着编译器难以觉察的依赖关系，则这些包级变量的初始化顺序是未定义的，依赖于具体编译器实现。 在下面这个Go白皮书中的例子中，

- 变量`a`肯定在变量`b`之后初始化；
- 但是变量`x`有可能在变量`b`之前、或者在变量`b`和变量`a`之间、或者在变量`a`之后初始化，取决于具体的编译器实现；
- 函数`sideEffect()`有可能在变量`x`初始化之前或者之后被调用，取决于具体的编译器实现。

```
1| // x是否依赖于a和b，不同的编译器有不同的见解。
2| var x = I(T{}).ab()
3| // 假设函数sideEffect和x、a以及b均无关系。
```

```
 4| var _ = sideEffect()
 5| var a = b
 6| var b = 42
 7|
 8| type I interface    { ab() []int }
 9| type T struct{}
10| func (T) ab() []int { return []int{a, b} }
```

注意：因为一个代码包中常常包含若干个代码源文件，而Go白皮书并没有强制规定编译器按照如何顺序来编译一个代码包中的源文件， 所以尽量不要让一个代码包中不同源文件中的包级变量存在复杂的依赖关系；否则不同的编译器可能会将一些包级变量初始化为不同结果。

另请注意：Go官方工具链的某些版本未正确实现在本节中描述的规则。

- Go官方工具链1.22.0-1.22.4版本中的一个bug（https://github.com/golang/go/issues/66575）。
- 1.23版本之前Go官方工具链中的一个bug（https://github.com/golang/go/issues/66585）。

布尔（逻辑）运算表达式中的操作数子表达式的估值顺序

在一个布尔运算`a && b`中，右操作数表达式`b`只在左操作数表达式`a`被估值为`true`的时候才会被估值。 所以，操作数表达式`b`如果需要被估值的话，它肯定在操作数表达式`a`之后估值。

在一个布尔运算`a || b`中，右操作数表达式`b`只在左操作数表达式`a`被估值为`false`的时候才会被估值。 所以，操作数表达式`b`如果需要被估值的话，它肯定在操作数表达式`a`之后估值。

通常估值顺序（The Usual Order）

Go白皮书这样描述**通常估值顺序**：

> ...，当估值一个表达式、赋值语句或者函数返回语句中的操作数时，所有的函数调用、方法调用和通道操作将按照它们在代码中的出现顺序进行估值。

注意：一个显式类型转换`T(v)`不属于函数调用。

举个例子，在表达式`[]int{x, fa(), fb(), y}`中，假设`x`和`y`是两个`int`类型的变量，`fa`和`fb`是两个返回值为`int`类型的函数，则调用`fa()`保证在调用`fb()`之前执行。 但是，下面这两个估值顺序没有在Go白皮书中指定：

- 变量`x`（或者`y`）和调用`fa()`（或者`fb()`）的相对估值顺序；
- 变量`x`、变量`y`、函数值`fa`和函数值`fb`的相对估值顺序。

下面是Go白皮书中提到的另一个例子：

```
y[z.f()], ok = g(h(a, b), i()+x[j()], <-c), k()
```

在此赋值语句中，

- `c`是一个通道表达式，它将被估值为一个通道值；
- `g`、`h`、`i`、`j`和`k`是一些函数表达式，它们将被估值为一些函数值；
- `f`是表达式`z`值的一个方法。

综合考虑上一节和本节上面已经提到的规则，编译器应该保证下列在运行时刻的估值顺序：

- 此赋值中涉及到的函数调用、方法调用和通道操作必须按照这样的顺序执行：`z.f()`→`h()`→`i()`→`j()`→`<-c`→`g()`→`k()`；
- 调用`h()`在表达式`h`、`a`和`b`估值之后调用；
- `y[]`在方法调用`z.f()`执行之后被估值；
- 方法调用`z.f()`在表达式`z`估值之后执行；
- `x[]`在调用`j()`执行之后被估值。

然而，下列次序在Go白皮书中未指定，它们依赖于具体编译器实现：

- 表达式`y`、`z`、`g`、`h`、`a`、`b`、`x`、`i`、`j`、`c`和`k`之间的相对估值顺序；
- 表达式`y[]`、`x[]`和`<-c`之间的相对估值顺序。

根据上述通常估值顺序规则，我们知道下面声明的变量`x`、`m`和`n`的初始值可能将出现歧义。

```
1|    a := 1
2|    f := func() int { a++; return a }
3|
4|    // x可能初始化为[1, 2]或者[2, 2]，
5|    // 因为a和f()的相对估值顺序未指定。
6|    x := []int{a, f()}
7|
8|    // m可能初始化为{2: 1}或者{2: 2}，
9|    // 因为两个映射元素的赋值顺序未指定。
```

```
10|    m := map[int]int{a: 1, a: 2}
11|
12|    // n可能初始化为{2: 3}或者{3: 3}，
13|    // 因为a和f()的相对估值顺序未指定。
14|    n := map[int]int{a: f()}
```

赋值语句中的表达式估值和赋值执行顺序

除了上面介绍的规则，Go白皮书对赋值语句中的表达式估值和各个单值赋值执行顺序进行了更多描述（原英文描述不是十分精确，在翻译过程中对之略加改进）：

> 一条赋值语句的执行分为两个阶段。 首先，做为目标值的元素索引表达式中的容器值表达式和索引值表达式、做为目标值的指针解引用表达式中的指针值表达式、以及此赋值语句中的其它非目标值表达式将按照上述通常估值顺序估值。 然后，各个单值赋值将按照从左到右的顺序执行。

以后，我们可以称第一个阶段为估值阶段，称第二个阶段为实施阶段。

Go白皮书并没有清楚地说明在第二个阶段中发生的赋值操作是否会对在第一个阶段结尾确定下来的各个子表达式的估值结果造成影响，此举曾造成了一些（https://github.com/golang/go/issues/23188#issuecomment-403951267）争议（https://github.com/golang/go/issues/15620）。 所以，这里下面将对赋值语句中的表达式估值顺序做出一些补充解释。

首先，先明确一下：第二个阶段中发生的赋值操作绝不会对在第一个阶段结尾确定下来的各个子表达式的估值结果造成影响.

为了方便下面的解释，对于一个赋值语句，我们假设一个做为目标值的容器（切片或者映射）元素索引表达式中的映射值总是可寻址的。 如果它是不可寻址的，我们可以认为在实施第二个阶段之前，此容器值已经被赋给了一个临时变量（可寻址的）并且在此赋值语句中此容器值已经被此临时变量取代。

在估值阶段结束之后、实施阶段开始之前的时刻，赋值语句中的每个目标值表达式都已经被估值为它的最基本形式。 不同风格的目标值表达式有着不同的最基本形式：

- 如果一个目标值表达式是一个空标识符，则它的最基本形式依旧是一个空标识符；
- 如果一个目标值表达式是一个容器（数组或者切片或者映射）元素索引表达式`c[k]`，则它的最基本形式为`(*cAddr)[k]`，其中一个`cAddr`为指向`c`的指针；
- 对于其它情形的任何一个目标值表达式，它必然是可寻址的，则它的最基本形式为它的地址的解引用形式。

假设a和b为两个可寻址的同类型变量，则下面的赋值语句

```
 1|    a, b = b, a
```

将按照如下步骤执行：

```
 1| // 估值阶段
 2| P0 := &a; P1 := &b
 3| R0 := b; R1 := a
 4|
 5| // 最基本形式：*P0, *P1 = R0, R1
 6|
 7| // 实施阶段
 8| *P0 = R0
 9| *P1 = R1
```

下面是另外一个例子，其中x[0]而不是x[1]被修改了。

```
 1|    x := []int{0, 0}
 2|    i := 0
 3|    i, x[i] = 1, 2
 4|    fmt.Println(x) // [2 0]
```

上例中的第3行的分解执行步骤如下：

```
 1| // 估值阶段
 2| P0 := &i; P1 := &x; T2 := i
 3| R0 := 1; R1 := 2
 4| // 到这里，T2 == 0
 5|
 6| // 最基本形式：*P0, (*P1)[T2] = R0, R1
 7|
 8| // 实施阶段
 9| *P0 = R0
10| (*P1)[T2] = R1
```

下面是一个略为复杂一点的例子。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    m := map[string]int{"Go": 0}
 7|    s := []int{1, 1, 1}; olds := s
 8|    n := 2
```

```
 9|    p := &n
10|    s, m["Go"], *p, s[n] = []int{2, 2, 2}, s[1], m["Go"], 5
11|    fmt.Println(m, s, n) // map[Go:1] [2 2 2] 0
12|    fmt.Println(olds)    // [1 1 5]
13| }
```

上例中的第*10*行的分解执行步骤如下：

```
 1| // 估值阶段
 2| P0 := &s; PM1 := &m; K1 := "Go"; P2 := p; PS3 := &s; T3 :=
2
 3| R0 := []int{2, 2, 2}; R1 := s[1]; R2 := m["Go"]; R3 := 5
 4| // 到这里，R1 == 1, R2 == 0
 5|
 6| // 最基本形式：*P0, (*PM1)[K1], *P2, (*PS3)[T3] = R0, R1, R2,
R3
 7|
 8| // 实施阶段
 9| *P0 = R0
10| (*PM1)[K1] = R1
11| *P2 = R2
12| (*PS3)[T3] = R3
```

下面这个例子将一个切片中的所有元素循环顺移了一位。

```
 1|    x := []int{2, 3, 5, 7, 11}
 2|    t := x[0]
 3|    var i int
 4|    for i, x[i] = range x {}
 5|    x[i] = t
 6|    fmt.Println(x) // [3 5 7 11 2]
```

另一个例子：

```
 1|    x := []int{123}
 2|    x, x[0] = nil, 456        // 此句不会发生恐慌
 3|    x, x[0] = []int{123}, 789 // 此句将产生恐慌
```

尽管使用复杂的多值赋值语句是合法的，但是在实践中并不推荐使用，因为复杂多值赋值语句的可读性不高，编译速度较慢，并且执行效率相对略低。

上面已经提到了，并非所有的估值顺序都在Go白皮书中指定清楚了，所以不同的Go编译器对这些未指定的估值顺序有着自己的理解。 一些跨编译器兼容性不好的代码将导致使用不同的编译器编译的程序的运行结果不同。 在下面这个例子中，表达式`x+1`和`f(&x)`的估值顺序是编译器相关的，所以此程序输出`100 99`或者`1 99`都是合理的。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    f := func (p *int) int {
 7|       *p = 99
 8|       return *p
 9|    }
10|
11|    x := 0
12|    y, z := x+1, f(&x)
13|    fmt.Println(y, z)
14| }
```

下面是另一个可能输出不同结果的程序。它可能输出`1 7 2`、`1 8 2`或者`1 9 2`，取决于不同的编译器实现。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    x, y := 0, 7
 7|    f := func() int {
 8|       x++
 9|       y++
10|       return x
11|    }
12|    fmt.Println(f(), y, f())
13| }
```

switch-case流程控制代码块中的表达式估值顺序

`switch-case`流程控制代码块中的表达式估值顺序已经在前面的文章中大致描述过了（第12章）。 这里仅仅展示一个例子。简单来说，在进入一个分支代码块之前，各个`case`关键字后跟随的表达式将按照从上到下和从左到右的顺序进行估值，直到某个比较结果为`true`为止。

```
 1| package main
 2|
 3| import "fmt"
```

```
 4|
 5| func main() {
 6|    f := func(n int) int {
 7|       fmt.Printf("f(%v) is called.\n", n)
 8|       return n
 9|    }
10|
11|    switch x := f(3); x + f(4) {
12|    default:
13|    case f(5):
14|    case f(6), f(7), f(8):
15|    case f(9), f(10):
16|    }
17| }
```

在运行时刻，各个`f()`调用将按照传给它们参数的大小顺序进行估值，直到和调用`f(7)`的比较结果为`true`为止。 所以调用`f(8)`、`f(9)`和`f(10)`将不会被估值。

输出结果：

```
f(3) is called.
f(4) is called.
f(5) is called.
f(6) is called.
f(7) is called.
```

`select-case`流程控制代码块中的表达式估值顺序

当执行一个`select-case`流程控制代码块时，各个`case`关键值后跟随的所有通道操作中的通道表达式和所有通道发送操作中的发送值表达式都将被按照它们在代码中的出现次序（从上到下从左到右）估值一次。

注意：以通道接收操作做为源值的赋值语句中的目标值表达式只有在此通道接收操作被选中之后才会被估值。

比如，在下面这个例子中，表达式`*fptr("aaa")`将永不会得到估值，因为它对应的通道接收操作`<-fchan("bbb", nil)`是个不可能被选中的阻塞操作。

```
 1| package main
 2|
 3| import "fmt"
 4|
```

```
 5| func main() {
 6|    c := make(chan int, 1)
 7|    c <- 0
 8|    fchan := func(info string, c chan int) chan int {
 9|       fmt.Println(info)
10|       return c
11|    }
12|    fptr := func(info string) *int {
13|       fmt.Println(info)
14|       return new(int)
15|    }
16|
17|    select {
18|    case *fptr("aaa") = <-fchan("bbb", nil): // blocking
19|    case *fptr("ccc") = <-fchan("ddd", c):   // non-blocking
20|    case fchan("eee", nil) <- *fptr("fff"):  // blocking
21|    case fchan("ggg", nil) <- *fptr("hhh"):  // blocking
22|    }
23| }
```

上例的输出结果：

```
bbb
ddd
eee
fff
ggg
hhh
ccc
```

注意，表达式`*fptr("ccc")`是上例中最后一个被估值的表达式。 它在对应的数据接收操作`<-fchan("ddd", c)`被选中之后才会进行估值。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

值复制成本

在Go编程中，值复制是很常见的操作。赋值、传参和通道发送操作均涉及到值复制。 本篇文章将谈谈各种不同种类的类型的Go值的复制成本。

值尺寸（value size）

一个值的尺寸表示此值的直接部分（第17章）在内存中占用多少个字节，它的间接部分（如果存在的话）对它的尺寸没有贡献。

在Go中，如果两个值的类型为同一种类（第14章）的类型，并且它们的类型的种类不为字符串、接口、数组和结构体，则这两个值的尺寸总是相等的。

事实上，对于官方标准编译器来说，任意两个字符串值的尺寸总是相等的，即使它们的字符串类型并不是同一个类型。 同样地，任意两个接口值的尺寸也都是相等的。

目前（Go 1.23），至少对于官方标准编译器来说，任何一个特定类型的所有值的尺寸都是相同的。所以我们也常说一个值的尺寸为此值的类型的尺寸。

一个数组类型的尺寸取决于它的元素类型的尺寸和它的长度。它的尺寸为它的元素类型的尺寸和它的长度的乘积。

一个结构体类型的尺寸取决于它的各个字段的类型尺寸和这些字段的排列顺序。 为了程序执行性能，编译器需要保证某些类型的值在内存中存放时必须满足特定的内存地址对齐（第44章）要求。 地址对齐可能会造成相邻的两个字段之间在内存中被插入填充一些多余的字节。 所以，一个结构体类型的尺寸必定不小于（常常会大于）此结构体类型的各个字段的类型尺寸之和。

下表列出了各种种类的类型的尺寸（对标准编译器1.23版本来说）。 在此表中，一个word表示一个原生字。在32位系统架构中，一个word为4个字节；而在64位系统架构中，一个word为8个字节。

类型种类	值尺寸	Go白皮书（https://golang.google.cn/ref/spec#Numeric_types）中的要求（https://golang.google.cn/ref/spec#Size_and_alignment_guarantees）
布尔	1 byte	未做特别要求
int8, uint8 (byte)	1 byte	1 byte
int16, uint16	2 bytes	2 bytes
int32 (rune), uint32, float32	4 bytes	4 bytes
int64, uint64, float64, complex64	8 bytes	8 bytes
complex128	16 bytes	16 bytes

类型种类	值尺寸	Go白皮书（https://golang.google.cn/ref/spec#Numeric_types）中的要求（https://golang.google.cn/ref/spec#Size_and_alignment_guarantees）
int, uint	1 word	架构相关，在32位系统架构中为4个字节，而在64位系统架构中为8个字节
uintptr	1 word	必须足够存下任一个内存地址
字符串	2 words	未做特别要求
指针和非类型安全指针	1 word	未做特别要求
切片	3 words	未做特别要求
映射	1 word	未做特别要求
通道	1 word	未做特别要求
函数	1 word	未做特别要求
接口	2 words	未做特别要求
结构体	所有字段尺寸之和 + 所有填充的字节数（第44章）	一个不含任何尺寸大于零的字段的结构体类型的尺寸为零
数组	元素类型的尺寸 * 长度	一个元素类型的尺寸为零的数组类型的尺寸为零

值复制成本

一般说来，复制一个值的成本正比于此值的尺寸。 但是，值尺寸并非是值复制成本的唯一决定因素。 不同的CPU型号和编译器版本可能会对某些特定尺寸的值的复制做了优化。

在实践中，我们可以将尺寸不大于4个原生字并且字段数不超过4个的结构体值看作是小尺寸值。复制小尺寸值的代价是比较小的。

对于标准编译器，除了大尺寸的结构体和数组类型，其它类型均为小尺寸类型。

为了防止在函数传参和通道操作中因为值复制代价太高而造成的性能损失，我们应该避免使用大尺寸的结构体和数组类型做为参数类型和通道的元素类型，应该在这些场合下使用基类型为这样的大尺寸类型的指针类型。 另一方面，我们也要考虑到太多的指针将会增加垃圾回收的压力。所以到底应该使用大尺寸类型还是以大尺寸类型为基类型的指针类型做为参数类型或通道的元素类型取决于具体的应用场景。

一般来说，在实践中，我们很少使用基类型为切片类型、映射类型、通道类型、函数类型、字符串类型和接口类型的指针类型，因为复制这些类型的值的代价很小。

如果一个数组或者切片的元素类型是一个大尺寸类型，我们应该避免在for-range循环中使用双循环变量来遍历这样的数组或者切片类型的值中的元素。 因为，在遍历过程中，每个元素将被复制给第二个循环变量一次。

下面这个例子展示了三种遍历一个切片的方法的性能差异。

```
 1| package main
 2|
 3| import "testing"
 4|
 5| type S [12]int64
 6| var sX = make([]S, 1000)
 7| var sY = make([]S, 1000)
 8| var sZ = make([]S, 1000)
 9| var sumX, sumY, sumZ int64
10|
11| func Benchmark_Loop(b *testing.B) {
12|    for i := 0; i < b.N; i++ {
13|       sumX = 0
14|       for j := 0; j < len(sX); j++ {
15|          sumX += sX[j][0]
16|       }
17|    }
18| }
19|
20| func Benchmark_Range_OneIterVar(b *testing.B) {
21|    for i := 0; i < b.N; i++ {
22|       sumY = 0
23|       for j := range sY {
24|          sumY += sY[j][0]
25|       }
26|    }
27| }
28|
29| func Benchmark_Range_TwoIterVar(b *testing.B) {
30|    for i := 0; i < b.N; i++ {
31|       sumZ = 0
32|       for _, v := range sZ {
33|          sumZ += v[0]
34|       }
35|    }
36| }
```

运行此基准测试，我们将得到下面的结果：

```
Benchmark_Loop-4             424342 2708 ns/op
Benchmark_Range_OneIterVar-4 407905 2808 ns/op
Benchmark_Range_TwoIterVar-4 214860 3915 ns/op
```

可以看出，使用双循环变量的方法的效率比另外两种方法的效率低不少。 但是请注意，某些编译器版本可能会做出一些特别的优化从而消除上面几种遍历方法的效率差异。 上面的基准测试结果基于Go标准编译器1.23版本。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

边界检查消除

Go是一个内存安全的语言。在数组和切片的索引和子切片操作中，Go运行时将检查操作中使用的下标是否越界。 如果下标越界，一个恐慌将产生，以防止这样的操作破坏内存安全。这样的检查称为边界检查。 边界检查使得我们的代码能够安全地运行；但是另一方面，也使得我们的代码运行效率略微降低。

从Go官方工具链1.7开始，官方标准编译器使用了一个新的基于SSA（single-assignment form，静态单赋值形式）的后端。 SSA使得Go编译器可以有效利用诸如BCE（https://en.wikipedia.org/wiki/Bounds-checking_elimination）（bounds check elimination，边界检查消除）和CSE（https://en.wikipedia.org/wiki/Common_subexpression_elimination）（common subexpression elimination，公共子表达式消除）等优化。 BCE可以避免很多不必要的边界检查，CSE可以避免很多重复表达式的计算，从而使得编译器编译出的程序执行效率更高。有时候这些优化的效果非常明显。

本文将展示一些例子来解释边界检查消除在官方标准编译器1.7+中的表现。

对于Go官方工具链1.7+，我们可以使用编译器选项`-gcflags="-d=ssa/check_bce"`来列出哪些代码行仍然需要边界检查。

例子1

```
 1| // example1.go
 2| package main
 3|
 4| func f1(s []int) {
 5|     _ = s[0] // 第5行：  需要边界检查
 6|     _ = s[1] // 第6行：  需要边界检查
 7|     _ = s[2] // 第7行：  需要边界检查
 8| }
 9|
10| func f2(s []int) {
11|     _ = s[2] // 第11行：  需要边界检查
12|     _ = s[1] // 第12行：  边界检查消除了！
13|     _ = s[0] // 第13行：  边界检查消除了！
14| }
15|
16| func f3(s []int, index int) {
17|     _ = s[index] // 第17行：  需要边界检查
18|     _ = s[index] // 第18行：  边界检查消除了！
```

```
19| }
20|
21| func f4(a [5]int) {
22|     _ = a[4] // 第22行：  边界检查消除了！
23| }
24|
25| func main() {}
```

```
$ go run -gcflags="-d=ssa/check_bce" example1.go
./example1.go:5: Found IsInBounds
./example1.go:6: Found IsInBounds
./example1.go:7: Found IsInBounds
./example1.go:11: Found IsInBounds
./example1.go:17: Found IsInBounds
```

我们可以看到函数`f2`中的第*12*行和第*13*行不再需要边界检查了，因为第*11*行的检查确保了第*12*行和第*13*行中使用的下标肯定不会越界。

但是，函数`f1`中的三行仍然都需要边界检查，因为第*5*行中的边界检查不能保证第*6*行和第*7*行中的下标没有越界，第*6*行中的边界检查也不能保证第第*7*行中的下标没有越界。

在函数`f3`中，编译器知道如果第一个`s[index]`是安全的，则第二个`s[index]`是无需边界检查的。

编译器也正确地认为函数`f4`中的唯一一行（第*22*行）是无需边界检查的。

注意：目前（Go官方工具链1.23版本），如果一个泛型函数中的操作涉及到类型参数，则标准编译器不会检查此操作是否需要边界检查。 下面是一个示例来证明这一点：

```
 1| // example1b.go
 2| package main
 3|
 4| func foo[E any](s []E) {
 5|     _ = s[0] // 第5行
 6|     _ = s[1] // 第6行
 7|     _ = s[2] // 第7行
 8| }
 9|
10| // var _ = foo[bool]
11|
12| func main() {
13| }
```

如果其中的变量声明行被注释掉，则标准编译器什么也不输出：

```
$ go run -gcflags="-d=ssa/check_bce" example1b.go
```

如果其中的变量声明行被启用，则标准编译器输出：

```
./aaa.go:5:7: Found IsInBounds
./example1b.go:6:7: Found IsInBounds
./example1b.go:7:7: Found IsInBounds
./example1b.go:4:6: Found IsInBounds
```

例子2

```
 1| // example2.go
 2| package main
 3|
 4| func f5(s []int) {
 5|    for i := range s {
 6|       _ = s[i]
 7|       _ = s[i:len(s)]
 8|       _ = s[:i+1]
 9|    }
10| }
11|
12| func f6(s []int) {
13|    for i := 0; i < len(s); i++ {
14|       _ = s[i]
15|       _ = s[i:len(s)]
16|       _ = s[:i+1]
17|    }
18| }
19|
20| func f7(s []int) {
21|    for i := len(s) - 1; i >= 0; i-- {
22|       _ = s[i]
23|       _ = s[i:len(s)]
24|    }
25| }
26|
27| func f8(s []int, index int) {
28|    if index >= 0 && index < len(s) {
29|       _ = s[index]
30|       _ = s[index:len(s)]
31|    }
```

```
32| }
33|
34| func f9(s []int) {
35|    if len(s) > 2 {
36|        _, _, _ = s[0], s[1], s[2]
37|    }
38| }
39|
40| func main() {}
```

```
$ go run -gcflags="-d=ssa/check_bce" example2.go
```

酷！官方标准编译器消除了上例程序中的所有边界检查。

注意：在Go官方工具链1.11之前，官方标准编译器没有足够聪明到认为第22行是不需要边界检查的。

例子3

```
 1| // example3.go
 2| package main
 3|
 4| import "math/rand"
 5|
 6| func fa() {
 7|    s := []int{0, 1, 2, 3, 4, 5, 6}
 8|    index := rand.Intn(7)
 9|    _ = s[:index] // 第9行：  需要边界检查
10|    _ = s[index:] // 第10行：  边界检查消除了！
11| }
12|
13| func fb(s []int, index int) {
14|    _ = s[:index] // 第14行：  需要边界检查
15|    _ = s[index:] // 第15行：  需要边界检查（不够智能？）
16| }
17|
18| func fc() {
19|    s := []int{0, 1, 2, 3, 4, 5, 6}
20|    s = s[:4]
21|    index := rand.Intn(7)
22|    _ = s[:index] // 第22行：  需要边界检查
23|    _ = s[index:] // 第23行：  需要边界检查（不够智能？）
24| }
```

```
25|
26| func main() {}
```

```
$ go run -gcflags="-d=ssa/check_bce" example3.go
./example3.go:9: Found IsSliceInBounds
./example3.go:14: Found IsSliceInBounds
./example3.go:15: Found IsSliceInBounds
./example3.go:22: Found IsSliceInBounds
./example3.go:23: Found IsSliceInBounds
```

噢，仍然有这么多的边界检查！

但是等等，为什么官方标准编译器认为第*10*行不需要边界检查，却认为第*15*和第*23*行仍旧需要边界检查呢？ 是标准编译器不够智能吗？

事实上，这里标准编译器做得对！为什么呢？ 原因是一个子切片表达式中的起始下标可能会大于基础切片的长度。 让我们先看一个简单的使用了子切片的例子：

```
 1| package main
 2|
 3| func main() {
 4|     s0 := make([]int, 5, 10)
 5|     // len(s0) == 5, cap(s0) == 10
 6|
 7|     index := 8
 8|
 9|     // 在Go中，对于一个子切片表达式s[a:b]，a和b须满足
10|     // 0 <= a <= b <= cap(s);否则，将产生一个恐慌。
11|
12|     _ = s0[:index]
13|     // 上一行是安全的不能保证下一行是无需边界检查的。
14|     // 事实上，下一行将产生一个恐慌，因为起始下标
15|     // index大于终止下标（即切片s0的长度）。
16|     _ = s0[index:] // panic
17| }
```

所以**如果`s[:index]`是安全的则`s[index:]`是无需边界检查的**这条论述只有在`len(s)`和`cap(s)`相等的情况下才正确。这就是函数`fb`和`fc`中的代码仍旧需要边界检查的原因。

标准编译器成功地检测到在函数`fa`中`len(s)`和`cap(s)`是相等的。干得漂亮！Go语言开发团队！

例子4

```
 1| // example4.go
 2| package main
 3|
 4| import "math/rand"
 5|
 6| func fb2(s []int, index int) {
 7|    _ = s[index:] // 第7行：  需要边界检查
 8|    _ = s[:index] // 第8行：  边界检查消除了！
 9| }
10|
11| func fc2() {
12|    s := []int{0, 1, 2, 3, 4, 5, 6}
13|    s = s[:4]
14|    index := rand.Intn(7)
15|    _ = s[index:] // 第15行：  需要边界检查
16|    _ = s[:index] // 第16行：  边界检查消除了！
17| }
18|
19| func main() {}
```

```
$ go run -gcflags="-d=ssa/check_bce" example4.go
./example4.go:7:7: Found IsSliceInBounds
./example4.go:15:7: Found IsSliceInBounds
```

在此例子中，标准编译器成功推断出：

- 在函数`fb2`中，如果第*7*行是安全的，则第*8*行是无需边界检查的；
- 在函数`fc2`中，如果第*15*行是安全的，则第*16*行是无需边界检查的。

注意：Go官方工具链1.9之前中的标准编译器没有出推断出第*8*行不需要边界检查。

例子5

当前版本的标准编译器并非足够智能到可以消除到一切应该消除的边界检查。 有时候，我们需要给标准编译器一些暗示来帮助标准编译器将这些不必要的边界检查消除掉。

```
 1| // example5.go
 2| package main
 3|
 4| func fd(is []int, bs []byte) {
 5|    if len(is) >= 256 {
 6|       for _, n := range bs {
 7|          _ = is[n] // 第7行：  需要边界检查
```

```
 8|       }
 9|    }
10| }
11|
12| func fd2(is []int, bs []byte) {
13|    if len(is) >= 256 {
14|       is = is[:256] // 第14行：  一个暗示
15|       for _, n := range bs {
16|          _ = is[n] // 第16行：  边界检查消除了！
17|       }
18|    }
19| }
20|
21| func main() {}
```

```
$ go run -gcflags="-d=ssa/check_bce" example5.go
./example5.go:7: Found IsInBounds
```

总结

本文上面列出的例子并没有涵盖标准编译器支持的所有边界检查消除的情形。本文列出的仅仅是一些常见的情形。

尽管标准编译器中的边界检查消除特性依然不是100%完美，但是对很多常见的情形，它确实很有效。 自从标准编译器支持此特性以来，在每个版本更新中，此特性都在不断地改进增强。 无需质疑，在以后的版本中，标准编译器会更加得智能，以至于上面第5个例子中提供给编译器的暗示有可能将变得不再必要。 谢谢Go语言开发团队出色的工作！

参考：

1. Bounds Check Elimination (https://docs.google.com/document/d/1vdAEAjYdzjnPA9WDOQ1e4e05cYVMpqSxJYZT33Cqw2g)
2. Utilizing the Go 1.7 SSA Compiler (https://talks.godoc.org/github.com/klauspost/talks/2016/go17-compiler.slide) (第二部分 (https://talks.godoc.org/github.com/klauspost/talks/2016/go17-compiler-2.slide))

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

并发同步概述

本文将解释什么是并发同步，并列出Go支持的几种并发同步技术。

什么是并发同步?

并发同步是指如何控制若干并发计算（在Go中，即协程），从而

- 避免在它们之间产生数据竞争的现象;
- 避免在它们无所事事的时候消耗CPU资源。

并发同步有时候也称为数据同步。

Go支持哪些并发同步技术?

通道（第21章）一文已经解释了如何使用通道来做并发同步。 除了使用通道，Go还支持几种通用并发同步技术，比如互斥锁和原子操作。 请阅读下面的文章来了解各种并发同步技术的使用。

- 通道用例大全（第37章）
- 如何优雅地关闭通道（第38章）
- `sync`标准库包中提供的并发同步技术（第39章）
- `sync/atomic`标准库包中提供的原子操作技术（第40章）

我们也可以利用网络和文件读写来做并发同步，但是这样的并发同步方法使用在一个程序进程内部时效率相对比较低。 一般来说，这样的方法多适用于多个进程之间或多个主机之间的并发同步。《Go语言101》中不介绍这样的并发同步方法。

为了更好地理解各种并发同步技术，推荐在适当的时候（具有一定的Go并发编程经验时）阅读Go中的内存顺序保证（第41章）一文。

Go中提供的各种并发同步技术并不能阻止Go程序员写出不正确的并发代码（第42章）。 但是，这些技术使得Go程序员可以轻松写出正确的并发代码。特别地，Go中提供的独特的通道技术使得并发编程变得很轻松和惬意。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

通道用例大全

在阅读本文之前，请先阅读通道（第21章）一文。 那篇文章详细地解释了通道类型和通道值，以及各种通道操作的规则细节。 一个Go新手程序员可能需要反复多次阅读那篇文章和当前这篇文章来精通Go通道编程。

本文余下的内容将展示很多通道用例。 希望这篇文章能够说服你接收下面的观点：

- 使用通道进行异步和并发编程是简单和惬意的；
- 通道同步技术比被很多其它语言采用的其它同步方案（比如角色模型（https://en.wikipedia.org/wiki/Actor_model）和async/await模式（https://en.wikipedia.org/wiki/Async/await））有着更多的应用场景和更多的使用变种。

请注意，本文的目的是展示尽量多的通道用例。但是，我们应该知道通道并不是Go支持的唯一同步技术，并且通道并不是在任何情况下都是最佳的同步技术。 请阅读原子操作（第40章）和其它并发同步技术（第39章）来了解更多的Go支持的同步技术。

将通道用做future/promise

很多其它流行语言支持future/promise来实现异步（并发）编程。 Future/promise常常用在请求/回应场合。

返回单向接收通道做为函数返回结果

在下面这个例子中，`sumSquares`函数调用的两个实参请求并发进行。 每个通道读取操作将阻塞到请求返回结果为止。 两个实参总共需要大约3秒钟（而不是6秒钟）准备完毕（以较慢的一个为准）。

```
 1| package main
 2|
 3| import (
 4|     "time"
 5|     "math/rand"
 6|     "fmt"
 7| )
 8|
 9| func longTimeRequest() <-chan int32 {
10|     r := make(chan int32)
```

```
11|
12|    go func() {
13|       time.Sleep(time.Second * 3) // 模拟一个工作负载
14|       r <- rand.Int31n(100)
15|    }()
16|
17|    return r
18| }
19|
20| func sumSquares(a, b int32) int32 {
21|    return a*a + b*b
22| }
23|
24| func main() {
25|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
26|
27|    a, b := longTimeRequest(), longTimeRequest()
28|    fmt.Println(sumSquares(<-a, <-b))
29| }
```

将单向发送通道类型用做函数实参

和上例一样，在下面这个例子中，`sumSquares`函数调用的两个实参的请求也是并发进行的。 和上例不同的是`longTimeRequest`函数接收一个单向发送通道类型参数而不是返回一个单向接收通道结果。

```
 1| package main
 2|
 3| import (
 4|    "time"
 5|    "math/rand"
 6|    "fmt"
 7| )
 8|
 9| func longTimeRequest(r chan<- int32)  {
10|    time.Sleep(time.Second * 3) // 模拟一个工作负载
11|    r <- rand.Int31n(100)
12| }
13|
14| func sumSquares(a, b int32) int32 {
15|    return a*a + b*b
16| }
17|
18| func main() {
```

```
19|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
20|
21|    ra, rb := make(chan int32), make(chan int32)
22|    go longTimeRequest(ra)
23|    go longTimeRequest(rb)
24|
25|    fmt.Println(sumSquares(<-ra, <-rb))
26| }
```

对于上面这个特定的例子，我们可以只使用一个通道来接收回应结果，因为两个参数的作用是对等的。

```
 1| ...
 2|
 3|    results := make(chan int32, 2) // 缓冲与否不重要
 4|    go longTimeRequest(results)
 5|    go longTimeRequest(results)
 6|
 7|    fmt.Println(sumSquares(<-results, <-results))
 8| }
```

这可以看作是后面将要提到的数据聚合的一个应用。

采用最快回应

本用例可以看作是上例中只使用一个通道变种的增强。

有时候，一份数据可能同时从多个数据源获取。这些数据源将返回相同的数据。 因为各种因素，这些数据源的回应速度参差不一，甚至某个特定数据源的多次回应速度之间也可能相差很大。 同时从多个数据源获取一份相同的数据可以有效保障低延迟。我们只需采用最快的回应并舍弃其它较慢回应。

注意：如果有N个数据源，为了防止被舍弃的回应对应的协程永久阻塞，则传输数据用的通道必须为一个容量至少为$N-1$的缓冲通道。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6|    "math/rand"
 7| )
 8|
 9| func source(c chan<- int32) {
10|    ra, rb := rand.Int31(), rand.Intn(3) + 1
```

```
11|    // 睡眠1秒/2秒/3秒
12|    time.Sleep(time.Duration(rb) * time.Second)
13|    c <- ra
14| }
15|
16| func main() {
17|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
18|
19|    startTime := time.Now()
20|    c := make(chan int32, 5) // 必须用一个缓冲通道
21|    for i := 0; i < cap(c); i++ {
22|       go source(c)
23|    }
24|    rnd := <- c // 只有第一个回应被使用了
25|    fmt.Println(time.Since(startTime))
26|    fmt.Println(rnd)
27| }
```

“采用最快回应”用例还有一些其它实现方式，本文后面将会谈及。

更多“请求/回应”用例变种

做为函数参数和返回结果使用的通道可以是缓冲的，从而使得请求协程不需阻塞到它所发送的数据被接收为止。

有时，一个请求可能并不保证返回一份有效的数据。对于这种情形，我们可以使用一个形如`struct{v T; err error}`的结构体类型或者一个空接口类型做为通道的元素类型以用来区分回应的值是否有效。

有时，一个请求可能需要比预期更长的用时才能回应，甚至永远都得不到回应。 我们可以使用本文后面将要介绍的超时机制来应对这样的情况。

有时，回应方可能会不断地返回一系列值，这也同时属于后面将要介绍的数据流的一个用例。

使用通道实现通知

通知可以被看作是特殊的请求/回应用例。在一个通知用例中，我们并不关心回应的值，我们只关心回应是否已发生。 所以我们常常使用空结构体类型`struct{}`来做为通道的元素类型，因为空结构体类型的尺寸为零，能够节省一些内存（虽然常常很少量）。

向一个通道发送一个值来实现单对单通知

我们已知道，如果一个通道中无值可接收，则此通道上的下一个接收操作将阻塞到另一个协程发送一个值到此通道为止。 所以一个协程可以向此通道发送一个值来通知另一个等待着从此通道接收数据的协程。

在下面这个例子中，通道 done 被用来做为一个信号通道来实现单对单通知。

```
 1| package main
 2|
 3| import (
 4|    "crypto/rand"
 5|    "fmt"
 6|    "os"
 7|    "sort"
 8| )
 9|
10| func main() {
11|    values := make([]byte, 32 * 1024 * 1024)
12|    if _, err := rand.Read(values); err != nil {
13|       fmt.Println(err)
14|       os.Exit(1)
15|    }
16|
17|    done := make(chan struct{}) // 也可以是缓冲的
18|
19|    // 排序协程
20|    go func() {
21|       sort.Slice(values, func(i, j int) bool {
22|          return values[i] < values[j]
23|       })
24|       done <- struct{}{} // 通知排序已完成
25|    }()
26|
27|    // 并发地做一些其它事情...
28|
29|    <- done // 等待通知
30|    fmt.Println(values[0], values[len(values)-1])
31| }
```

从一个通道接收一个值来实现单对单通知

如果一个通道的数据缓冲队列已满（非缓冲的通道的数据缓冲队列总是满的）但它的发送协程队列为空，则向此通道发送一个值将阻塞，直到另外一个协程从此通道接收一个值为止。 所以我们可以通过从一个通道接收数据来实现单对单通知。一般我们使用非缓冲通道来实现这样的通知。

这种通知方式不如上例中介绍的方式使用得广泛。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6| )
 7|
 8| func main() {
 9|    done := make(chan struct{})
10|       // 此信号通道也可以缓冲为1。如果这样，则在下面
11|       // 这个协程创建之前，我们必须向其中写入一个值。
12|
13|    go func() {
14|       fmt.Print("Hello")
15|       // 模拟一个工作负载。
16|       time.Sleep(time.Second * 2)
17|
18|       // 使用一个接收操作来通知主协程。
19|       <- done
20|    }()
21|
22|    done <- struct{}{} // 阻塞在此，等待通知
23|    fmt.Println(" world!")
24| }
```

另一个事实是，上面的两种单对单通知方式其实并没有本质的区别。 它们都可以被概括为较快者等待较慢者发出通知。

多对单和单对多通知

略微扩展一下上面两个用例，我们可以很轻松地实现多对单和单对多通知。

```
 1| package main
 2|
 3| import "log"
 4| import "time"
 5|
```

```
type T = struct{}

func worker(id int, ready <-chan T, done chan<- T) {
    <-ready // 阻塞在此，等待通知
    log.Print("Worker#", id, "开始工作")
    // 模拟一个工作负载。
    time.Sleep(time.Second * time.Duration(id+1))
    log.Print("Worker#", id, "工作完成")
    done <- T{} // 通知主协程（N-to-1）
}

func main() {
    log.SetFlags(0)

    ready, done := make(chan T), make(chan T)
    go worker(0, ready, done)
    go worker(1, ready, done)
    go worker(2, ready, done)

    // 模拟一个初始化过程
    time.Sleep(time.Second * 3 / 2)
    // 单对多通知
    ready <- T{}; ready <- T{}; ready <- T{}
    // 等待被多对单通知
    <-done; <-done; <-done
}
```

事实上，上例中展示的多对单和单对多通知实现方式在实践中用的并不多。 在实践中，我们多使用`sync.WaitGroup`来实现多对单通知，使用关闭一个通道的方式来实现单对多通知（详见下一个用例）。

通过关闭一个通道来实现群发通知

上一个用例中的单对多通知实现在实践中很少用，因为通过关闭一个通道的方式在来实现单对多通知的方式更简单。 我们已经知道，从一个已关闭的通道可以接收到无穷个值，我们可以利用这一特性来实现群发通知。

我们可以把上一个例子中的三个数据发送操作`ready <- struct{}{}`替换为一个通道关闭操作`close(ready)`来达到同样的单对多通知效果。

```
...
    close(ready) // 群发通知
...
```

当然，我们也可以通过关闭一个通道来实现单对单通知。事实上，关闭通道是实践中用得最多通知实现方式。

从一个已关闭的通道可以接收到无穷个值这一特性也将被用在很多其它在后面将要介绍的用例中。 实际上，这一特性被广泛地使用于标准库包中。比如，`context`标准库包使用了此特性来传达操作取消消息。

定时通知（timer）

用通道实现一个一次性的定时通知器是很简单的。 下面是一个自定义实现：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6| )
 7|
 8| func AfterDuration(d time.Duration) <- chan struct{} {
 9|    c := make(chan struct{}, 1)
10|    go func() {
11|       time.Sleep(d)
12|       c <- struct{}{}
13|    }()
14|    return c
15| }
16|
17| func main() {
18|    fmt.Println("Hi!")
19|    <- AfterDuration(time.Second)
20|    fmt.Println("Hello!")
21|    <- AfterDuration(time.Second)
22|    fmt.Println("Bye!")
23| }
```

事实上，`time`标准库包中的`After`函数提供了和上例中`AfterDuration`同样的功能。 在实践中，我们应该尽量使用`time.After`函数以使代码看上去更干净。

注意，操作`<-time.After(aDuration)`将使当前协程进入阻塞状态，而一个`time.Sleep(aDuration)`函数调用不会如此。

`<-time.After(aDuration)`经常被使用在后面将要介绍的超时机制实现中。

将通道用做互斥锁（mutex）

上面的某个例子提到了容量为1的缓冲通道可以用做一次性二元信号量（https://en.wikipedia.org/wiki/Semaphore_(programming)）。事实上，容量为1的缓冲通道也可以用做多次性二元信号量（即互斥锁）尽管这样的互斥锁效率不如sync标准库包中提供的互斥锁高效。

有两种方式将一个容量为1的缓冲通道用做互斥锁：

1. 通过发送操作来加锁，通过接收操作来解锁；
2. 通过接收操作来加锁，通过发送操作来解锁。

下面是一个通过发送操作来加锁的例子。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    mutex := make(chan struct{}, 1) // 容量必须为1
 7|
 8|    counter := 0
 9|    increase := func() {
10|       mutex <- struct{}{} // 加锁
11|       counter++
12|       <-mutex // 解锁
13|    }
14|
15|    increase1000 := func(done chan<- struct{}) {
16|       for i := 0; i < 1000; i++ {
17|          increase()
18|       }
19|       done <- struct{}{}
20|    }
21|
22|    done := make(chan struct{})
23|    go increase1000(done)
24|    go increase1000(done)
25|    <-done; <-done
26|    fmt.Println(counter) // 2000
27| }
```

下面是一个通过接收操作来加锁的例子，其中只显示了相对于上例而修改了的部分。

```
 1| ...
 2| func main() {
 3|    mutex := make(chan struct{}, 1)
 4|    mutex <- struct{}{} // 此行是必需的
 5|
 6|    counter := 0
 7|    increase := func() {
 8|       <-mutex // 加锁
 9|       counter++
10|       mutex <- struct{}{} // 解锁
11|    }
12| ...
```

将通道用做计数信号量（counting semaphore）

缓冲通道可以被用做[计数信号量](https://en.wikipedia.org/wiki/Semaphore_(programming))（https://en.wikipedia.org/wiki/Semaphore_(programming)）。 计数信号量可以被视为多主锁。如果一个缓冲通道的容量为N，那么它可以被看作是一个在任何时刻最多可有N个主人的锁。 上面提到的二元信号量是特殊的计数信号量，每个二元信号量在任一时刻最多只能有一个主人。

计数信号量经常被使用于限制最大并发数。

和将通道用做互斥锁一样，也有两种方式用来获取一个用做计数信号量的通道的一份所有权。

1. 通过发送操作来获取所有权，通过接收操作来释放所有权；
2. 通过接收操作来获取所有权，通过发送操作来释放所有权。

下面是一个通过接收操作来获取所有权的例子：

```
 1| package main
 2|
 3| import (
 4|    "log"
 5|    "time"
 6|    "math/rand"
 7| )
 8|
 9| type Seat int
10| type Bar chan Seat
11|
12| func (bar Bar) ServeCustomer(c int) {
13|    log.Print("顾客#", c, "进入酒吧")
```

```
 14|    seat := <- bar // 需要一个位子来喝酒
 15|    log.Print("++ customer#", c, " drinks at seat#", seat)
 16|    log.Print("++ 顾客#", c, "在第", seat, "个座位开始饮酒")
 17|    time.Sleep(time.Second * time.Duration(2 +
rand.Intn(6)))
 18|    log.Print("-- 顾客#", c, "离开了第", seat, "个座位")
 19|    bar <- seat // 释放座位，离开酒吧
 20| }
 21|
 22| func main() {
 23|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
 24|
 25|    bar24x7 := make(Bar, 10) // 此酒吧有10个座位
 26|    // 摆放10个座位。
 27|    for seatId := 0; seatId < cap(bar24x7); seatId++ {
 28|       bar24x7 <- Seat(seatId) // 均不会阻塞
 29|    }
 30|
 31|    for customerId := 0; ; customerId++ {
 32|       time.Sleep(time.Second)
 33|       go bar24x7.ServeCustomer(customerId)
 34|    }
 35|    for {time.Sleep(time.Second)} // 睡眠不属于阻塞状态
 36| }
```

在上例中，只有获得一个座位的顾客才能开始饮酒。 所以在任一时刻同时在喝酒的顾客数不会超过座位数10。

上例`main`函数中的最后一行`for`循环是为了防止程序退出。 后面将介绍一种更好的实现此目的的方法。

在上例中，尽管在任一时刻同时在喝酒的顾客数不会超过座位数10，但是在某一时刻可能有多于10个顾客进入了酒吧，因为某些顾客在排队等位子。 在上例中，每个顾客对应着一个协程。虽然协程的开销比系统线程小得多，但是如果协程的数量很多，则它们的总体开销还是不能忽略不计的。 所以，最好当有空位的时候才创建顾客协程。

```
  1| ... // 省略了和上例相同的代码
  2|
  3| func (bar Bar) ServeCustomerAtSeat(c int, seat Seat) {
  4|    log.Print("++ 顾客#", c, "在第", seat, "个座位开始饮酒")
  5|    time.Sleep(time.Second * time.Duration(2 +
rand.Intn(6)))
  6|    log.Print("-- 顾客#", c, "离开了第", seat, "个座位")
  7|    bar <- seat // 释放座位，离开酒吧
```

```
 8| }
 9|
10| func main() {
11|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
12|
13|    bar24x7 := make(Bar, 10)
14|    for seatId := 0; seatId < cap(bar24x7); seatId++ {
15|       bar24x7 <- Seat(seatId)
16|    }
17|
18|    // 这个for循环和上例不一样。
19|    for customerId := 0; ; customerId++ {
20|       time.Sleep(time.Second)
21|       seat := <- bar24x7 // 需要一个空位招待顾客
22|       go bar24x7.ServeCustomerAtSeat(customerId, seat)
23|    }
24|    for {time.Sleep(time.Second)}
25| }
```

在上面这个修改后的例子中，在任一时刻最多只有10个顾客协程在运行（但是在程序的生命期内，仍旧会有大量的顾客协程不断被创建和销毁）。

在下面这个更加高效的实现中，在程序的生命期内最多只会有10个顾客协程被创建出来。

```
 1| ... // 省略了和上例相同的代码
 2|
 3| func (bar Bar) ServeCustomerAtSeat(consumers chan int) {
 4|    for c := range consumers {
 5|       seatId := <- bar
 6|       log.Print("++ 顾客#", c, "在第", seatId, "个座位开始饮
酒")
 7|       time.Sleep(time.Second * time.Duration(2 +
rand.Intn(6)))
 8|       log.Print("-- 顾客#", c, "离开了第", seatId, "个座位")
 9|       bar <- seatId // 释放座位，离开酒吧
10|    }
11| }
12|
13| func main() {
14|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
15|
16|    bar24x7 := make(Bar, 10)
17|    for seatId := 0; seatId < cap(bar24x7); seatId++ {
18|       bar24x7 <- Seat(seatId)
```

```
19|    }
20|
21|    consumers := make(chan int)
22|    for i := 0; i < cap(bar24x7); i++ {
23|       go bar24x7.ServeCustomerAtSeat(consumers)
24|    }
25|
26|    for customerId := 0; ; customerId++ {
27|       time.Sleep(time.Second)
28|       consumers <- customerId
29|    }
30| }
```

题外话：当然，如果我们并不关心座位号（这种情况在编程实践中很常见），则实际上`bar24x7`计数信号量是完全不需要的：

```
 1| ... // 省略了和上例相同的代码
 2|
 3| func ServeCustomer(consumers chan int) {
 4|    for c := range consumers {
 5|       log.Print("++ 顾客#", c, "开始在酒吧饮酒")
 6|       time.Sleep(time.Second * time.Duration(2 +
rand.Intn(6)))
 7|       log.Print("-- 顾客#", c, "离开了酒吧")
 8|    }
 9| }
10|
11| func main() {
12|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
13|
14|    const BarSeatCount = 10
15|    consumers := make(chan int)
16|    for i := 0; i < BarSeatCount; i++ {
17|       go ServeCustomer(consumers)
18|    }
19|
20|    for customerId := 0; ; customerId++ {
21|       time.Sleep(time.Second)
22|       consumers <- customerId
23|    }
24| }
```

通过发送操作来获取所有权的实现相对简单一些，省去了摆放座位的步骤。

```
 1| package main
 2|
 3| import (
 4|    "log"
 5|    "time"
 6|    "math/rand"
 7| )
 8|
 9| type Customer struct{id int}
10| type Bar chan Customer
11|
12| func (bar Bar) ServeCustomer(c Customer) {
13|    log.Print("++ 顾客#", c.id, "开始饮酒")
14|    time.Sleep(time.Second * time.Duration(3 +
rand.Intn(16)))
15|    log.Print("-- 顾客#", c.id, "离开酒吧")
16|    <- bar // 离开酒吧，腾出位子
17| }
18|
19| func main() {
20|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
21|
22|    bar24x7 := make(Bar, 10) // 最多同时服务10位顾客
23|    for customerId := 0; ; customerId++ {
24|       time.Sleep(time.Second * 2)
25|       customer := Customer{customerId}
26|       bar24x7 <- customer // 等待进入酒吧
27|       go bar24x7.ServeCustomer(customer)
28|    }
29|    for {time.Sleep(time.Second)}
30| }
```

对话（或称乒乓）

两个协程可以通过一个通道进行对话，整个过程宛如打乒乓球一样。 下面是一个这样的例子，它将打印出一系列斐波那契（Fibonacci）数。

```
 1| package main
 2|
 3| import "fmt"
 4| import "time"
 5| import "os"
 6|
 7| type Ball uint64
```

```
 8|
 9| func Play(playerName string, table chan Ball) {
10|    var lastValue Ball = 1
11|    for {
12|       ball := <- table // 接球
13|       fmt.Println(playerName, ball)
14|       ball += lastValue
15|       if ball < lastValue { // 溢出结束
16|          os.Exit(0)
17|       }
18|       lastValue = ball
19|       table <- ball // 回球
20|       time.Sleep(time.Second)
21|    }
22| }
23|
24| func main() {
25|    table := make(chan Ball)
26|    go func() {
27|       table <- 1 // （裁判）发球
28|    }()
29|    go Play("A:", table)
30|    Play("B:", table)
31| }
```

使用通道传送传输通道

一个通道类型的元素类型可以是另一个通道类型。 在下面这个例子中， 单向发送通道类型`chan<- int`是另一个通道类型`chan chan<- int`的元素类型。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| var counter = func (n int) chan<- chan<- int {
 6|    requests := make(chan chan<- int)
 7|    go func() {
 8|       for request := range requests {
 9|          if request == nil {
10|             n++ // 递增计数
11|          } else {
12|             request <- n // 返回当前计数
13|          }
14|       }
```

```
15|    }()
16|    return requests // 隐式转换到类型chan<- (chan<- int)
17| }(0)
18|
19| func main() {
20|    increase1000 := func(done chan<- struct{}) {
21|       for i := 0; i < 1000; i++ {
22|          counter <- nil
23|       }
24|       done <- struct{}{}
25|    }
26|
27|    done := make(chan struct{})
28|    go increase1000(done)
29|    go increase1000(done)
30|    <-done; <-done
31|
32|    request := make(chan int, 1)
33|    counter <- request
34|    fmt.Println(<-request) // 2000
35| }
```

尽管对于上面这个用例来说，使用通道传送传输通道这种方式并非是最有效的实现方式，但是这种方式肯定有最适合它的用武之地。

检查通道的长度和容量

我们可以使用内置函数`cap`和`len`来查看一个通道的容量和当前长度。 但是在实践中我们很少这样做。我们很少使用内置函数`cap`的原因是一个通道的容量常常是已知的或者不重要的。 我们很少使用内置函数`len`的原因是一个`len`调用的结果并不能总能准确地反映出的一个通道的当前长度。

但有时确实有一些场景需要调用这两个函数。比如，有时一个协程欲将一个未关闭的并且不会再向其中发送数据的缓冲通道中的所有数据接收出来，在确保只有此一个协程从此通道接收数据的情况下，我们可以用下面的代码来实现之：

```
1| for len(c) > 0 {
2|    value := <-c
3|    // 使用value ...
4| }
```

我们也可以用本文后面将要介绍的尝试接收机制来实现这一需求。两者的运行效率差不多，但尝试接收机制的优点是多个协程可以并发地进行读取操作。

有时一个协程欲将一个缓冲通道写满而又不阻塞，在确保只有此一个协程向此通道发送数据的情况下，我们可以用下面的代码实现这一目的：

```
for len(c) < cap(c) {
    c <- aValue
}
```

当然，我们也可以使用后面将要介绍的尝试发送机制来实现这一需求。

使当前协程永久阻塞

Go中的选择机制（select）（第21章）是一个非常独特的特性。它给并发编程带来了很多新的模式和技巧。

我们可以用一个无分支的`select`流程控制代码块使当前协程永久处于阻塞状态。这是`select`流程控制的最简单的应用。 事实上，上面很多例子中的`for {time.Sleep(time.Second)}`都可以换为`select{}`。

一般，`select{}`用在主协程中以防止程序退出。

一个例子：

```
package main

import "runtime"

func DoSomething() {
    for {
        // 做点什么...

        runtime.Gosched() // 防止本协程霸占CPU不放
    }
}

func main() {
    go DoSomething()
    go DoSomething()
    select{}
}
```

顺便说一句，另外还有一些使当前协程永久阻塞的方法（第46章），但是`select{}`是最简单的方法。

尝试发送和尝试接收

含有一个default分支和一个case分支的select代码块可以被用做一个尝试发送或者尝试接收操作，取决于case关键字后跟随的是一个发送操作还是一个接收操作。

- 如果case关键字后跟随的是一个发送操作，则此select代码块为一个尝试发送操作。 如果case分支的发送操作是阻塞的，则default分支将被执行，发送失败；否则发送成功， case分支得到执行。
- 如果case关键字后跟随的是一个接收操作，则此select代码块为一个尝试接收操作。 如果case分支的接收操作是阻塞的，则default分支将被执行，接收失败；否则接收成功， case分支得到执行。

尝试发送和尝试接收代码块永不阻塞。

标准编译器对尝试发送和尝试接收代码块做了特别的优化，使得它们的执行效率比多case分支的普通select代码块执行效率高得多。

下例演示了尝试发送和尝试接收代码块的工作原理。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    type Book struct{id int}
 7|    bookshelf := make(chan Book, 3)
 8|
 9|    for i := 0; i < cap(bookshelf) * 2; i++ {
10|       select {
11|       case bookshelf <- Book{id: i}:
12|          fmt.Println("成功将书放在书架上", i)
13|       default:
14|          fmt.Println("书架已经被占满了")
15|       }
16|    }
17|
18|    for i := 0; i < cap(bookshelf) * 2; i++ {
19|       select {
20|       case book := <-bookshelf:
21|          fmt.Println("成功从书架上取下一本书", book.id)
22|       default:
23|          fmt.Println("书架上已经没有书了")
24|       }
```

```
25|    }
26| }
```

输出结果：

```
成功将书放在书架上 0
成功将书放在书架上 1
成功将书放在书架上 2
书架已经被占满了
书架已经被占满了
书架已经被占满了
成功从书架上取下一本书 0
成功从书架上取下一本书 1
成功从书架上取下一本书 2
书架上已经没有书了
书架上已经没有书了
书架上已经没有书了
```

后面的很多用例还要用到尝试发送和尝试接收代码块。

无阻塞地检查一个通道是否已经关闭

假设我们可以保证没有任何协程会向一个通道发送数据，则我们可以使用下面的代码来（并发安全地）检查此通道是否已经关闭，此检查不会阻塞当前协程。

```
1| func IsClosed(c chan T) bool {
2|    select {
3|    case <-c:
4|       return true
5|    default:
6|    }
7|    return false
8| }
```

此方法常用来查看某个期待中的通知是否已经来临。此通知将由另一个协程通过关闭一个通道来发送。

峰值限制（peak/burst limiting）

将通道用做计数信号量（#semaphore）用例和通道尝试（发送或者接收）操作结合起来可用实现峰值限制。 峰值限制的目的是防止过大的并发请求数。

下面是对将通道用做计数信号量（#semaphore）一节中的最后一个例子的简单修改，从而使得顾客不再等待而是离去或者寻找其它酒吧。

```
 1| ...
 2|    bar24x7 := make(Bar, 10) // 此酒吧只能同时招待10个顾客
 3|    for customerId := 0; ; customerId++ {
 4|       time.Sleep(time.Second)
 5|       consumer := Consumer{customerId}
 6|       select {
 7|       case bar24x7 <- consumer: // 试图进入此酒吧
 8|          go bar24x7.ServeConsumer(consumer)
 9|       default:
10|          log.Print("顾客#", customerId, "不愿等待而离去")
11|       }
12|    }
13| ...
```

另一种“采用最快回应”的实现方式

在上面的“采用最快回应”用例一节已经提到，我们也可以使用选择机制来实现“采用最快回应”用例。 每个数据源协程只需使用一个缓冲为1的通道并向其尝试发送回应数据即可。示例代码如下：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "math/rand"
 6|    "time"
 7| )
 8|
 9| func source(c chan<- int32) {
10|    ra, rb := rand.Int31(), rand.Intn(3)+1
11|    // 休眠1秒/2秒/3秒
12|    time.Sleep(time.Duration(rb) * time.Second)
13|    select {
14|    case c <- ra:
15|    default:
16|    }
17| }
18|
19| func main() {
20|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
21|
22|    c := make(chan int32, 1) // 此通道容量必须至少为1
23|    for i := 0; i < 5; i++ {
24|       go source(c)
```

```
25|    }
26|    rnd := <-c // 只采用第一个成功发送的回应数据
27|    fmt.Println(rnd)
28| }
```

注意，使用选择机制来实现“采用最快回应”的代码中使用的通道的容量必须至少为1，以保证最快回应总能够发送成功。 否则，如果数据请求者因为种种原因未及时准备好接收，则所有回应者的尝试发送都将失败，从而所有回应的数据都将被错过。

第三种“采用最快回应”的实现方式

如果一个“采用最快回应”用例中的数据源的数量很少，比如两个或三个，我们可以让每个数据源使用一个单独的缓冲通道来回应数据，然后使用一个`select`代码块来同时接收这三个通道。 示例代码如下：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "math/rand"
 6|    "time"
 7| )
 8|
 9| func source() <-chan int32 {
10|    c := make(chan int32, 1) // 必须为一个缓冲通道
11|    go func() {
12|       ra, rb := rand.Int31(), rand.Intn(3)+1
13|       time.Sleep(time.Duration(rb) * time.Second)
14|       c <- ra
15|    }()
16|    return c
17| }
18|
19| func main() {
20|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
21|
22|    var rnd int32
23|    // 阻塞在此直到某个数据源率先回应。
24|    select{
25|    case rnd = <-source():
26|    case rnd = <-source():
27|    case rnd = <-source():
28|    }
```

```
29|    fmt.Println(rnd)
30| }
```

注意：如果上例中使用的通道是非缓冲的，未被选中的case分支对应的两个source函数调用中开辟的协程将处于永久阻塞状态，从而造成内存泄露（第45章）。

本小节和上一小节中展示的两种方法也可以用来实现多对单通知。

超时机制（timeout）

在一些请求/回应用例中，一个请求可能因为种种原因导致需要超出预期的时长才能得到回应，有时甚至永远得不到回应。 对于这样的情形，我们可以使用一个超时方案给请求者返回一个错误信息。 使用选择机制可以很轻松地实现这样的一个超时方案。

下面这个例子展示了如何实现一个支持超时设置的请求：

```
 1| func requestWithTimeout(timeout time.Duration) (int, error)
{
 2|    c := make(chan int)
 3|    go doRequest(c) // 可能需要超出预期的时长回应
 4|
 5|    select {
 6|    case data := <-c:
 7|       return data, nil
 8|    case <-time.After(timeout):
 9|       return 0, errors.New("超时了！")
10|    }
11| }
```

脉搏器（ticker）

我们可以使用尝试发送操作来实现一个每隔一定时间发送一个信号的脉搏器。

```
 1| package main
 2|
 3| import "fmt"
 4| import "time"
 5|
 6| func Tick(d time.Duration) <-chan struct{} {
 7|    c := make(chan struct{}, 1) // 容量最好为1
 8|    go func() {
```

```
 9|       for {
10|          time.Sleep(d)
11|          select {
12|          case c <- struct{}{}:
13|          default:
14|          }
15|       }
16|    }()
17|    return c
18| }
19|
20| func main() {
21|    t := time.Now()
22|    for range Tick(time.Second) {
23|       fmt.Println(time.Since(t))
24|    }
25| }
```

事实上，time标准库包中的Tick函数提供了同样的功能，但效率更高。 我们应该尽量使用标准库包中的实现。

速率限制（rate limiting）

上面已经展示了如何使用尝试发送实现峰值限制（#peak-limiting）。 同样地，我们也可以使用使用尝试机制来实现速率限制，但需要前面刚提到的定时器实现的配合。 速率限制常用来限制吞吐和确保在一段时间内的资源使用不会超标。

下面的例子借鉴了官方Go维基中的例子（https://github.com/golang/go/wiki/RateLimiting）。 在此例中，任何一分钟时段内处理的请求数不会超过200。

```
 1| package main
 2|
 3| import "fmt"
 4| import "time"
 5|
 6| type Request interface{}
 7| func handle(r Request) {fmt.Println(r.(int))}
 8|
 9| const RateLimitPeriod = time.Minute
10| const RateLimit = 200 // 任何一分钟内最多处理200个请求
11|
12| func handleRequests(requests <-chan Request) {
13|    quotas := make(chan time.Time, RateLimit)
```

```
14|
15|    go func() {
16|       tick := time.NewTicker(RateLimitPeriod / RateLimit)
17|       defer tick.Stop()
18|       for t := range tick.C {
19|          select {
20|          case quotas <- t:
21|          default:
22|          }
23|       }
24|    }()
25|
26|    for r := range requests {
27|       <-quotas
28|       go handle(r)
29|    }
30| }
31|
32| func main() {
33|    requests := make(chan Request)
34|    go handleRequests(requests)
35|    // time.Sleep(time.Minute)
36|    for i := 0; ; i++ {requests <- i}
37| }
```

上例的代码虽然可以保证任何一分钟时段内处理的请求数不会超过200，但是如果在开始的一分钟内没有任何请求，则接下来的某个瞬时时间点可能会同时处理最多200个请求（试着将`time.Sleep`行的注释去掉看看）。 这可能会造成卡顿情况。我们可以将速率限制和峰值限制一并使用来避免出现这样的情况。

开关

通道（第21章）一文提到了向一个nil通道发送数据或者从中接收数据都属于阻塞操作。 利用这一事实，我们可以将一个`select`流程控制中的`case`操作中涉及的通道设置为不同的值，以使此`select`流程控制选择执行不同的分支。

下面是另一个乒乓模拟游戏的实现。此实现使用了选择机制。在此例子中，两个`case`操作中的通道有且只有一个为nil，所以只能是不为nil的通道对应的分支被选中。 每个循环步将对调这两个`case`操作中的通道，从而改变两个分支的可被选中状态。

```
 1| package main
 2|
```

```
 3| import "fmt"
 4| import "time"
 5| import "os"
 6|
 7| type Ball uint8
 8| func Play(playerName string, table chan Ball, serve bool) {
 9|    var receive, send chan Ball
10|    if serve {
11|       receive, send = nil, table
12|    } else {
13|       receive, send = table, nil
14|    }
15|    var lastValue Ball = 1
16|    for {
17|       select {
18|       case send <- lastValue:
19|       case value := <- receive:
20|          fmt.Println(playerName, value)
21|          value += lastValue
22|          if value < lastValue { // 溢出了
23|             os.Exit(0)
24|          }
25|          lastValue = value
26|       }
27|       receive, send = send, receive // 开关切换
28|       time.Sleep(time.Second)
29|    }
30| }
31|
32| func main() {
33|    table := make(chan Ball)
34|    go Play("A:", table, false)
35|    Play("B:", table, true)
36| }
```

下面是另一个也展示了开关效果的但简单得多的（非并发的）小例子。 此程序将不断打印出`1212...`。 它在实践中没有太多实用价值，这里只是为了学习的目的才展示之。

```
 1| package main
 2|
 3| import "fmt"
 4| import "time"
 5|
 6| func main() {
```

```
 7|    for c := make(chan struct{}, 1); true; {
 8|       select {
 9|       case c <- struct{}{}:
10|          fmt.Print("1")
11|       case <-c:
12|          fmt.Print("2")
13|       }
14|       time.Sleep(time.Second)
15|    }
16| }
```

控制代码被执行的几率

我们可以通过在一个`select`流程控制中使用重复的`case`操作来增加对应分支中的代码的执行几率。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    foo, bar := make(chan struct{}), make(chan struct{})
 7|    close(foo); close(bar) // 仅为演示目的
 8|    x, y := 0.0, 0.0
 9|    f := func(){x++}
10|    g := func(){y++}
11|    for i := 0; i < 100000; i++ {
12|       select {
13|       case <-foo: f()
14|       case <-foo: f()
15|       case <-bar: g()
16|       }
17|    }
18|    fmt.Println(x/y) // 大致为2
19| }
```

在上面这个例子中，函数`f`的调用执行几率大致为函数`g`的两倍。

从动态数量的分支中选择

每个select控制流程中的分支数量在运行中是固定的，但是我们可以使用reflect标准库包中提供的功能在运行时刻来构建动态分支数量的select控制流程。 但是请注意：一个select控制流程中的分支越多，此select控制流程的执行效率就越低（这是我们常常只使用不多于三个分支的select控制流程的原因）。

reflect标准库包中也提供了模拟尝试发送和尝试接收代码块的TrySend和TryRecv函数。

数据流操纵

本节将介绍一些使用通道进行数据流处理的用例。

一般来说，一个数据流处理程序由多个模块组成。不同的模块执行分配给它们的不同的任务。 每个模块由一个或者数个并行工作的协程组成。实践中常见的工作任务包括：

- 数据生成/搜集/加载；
- 数据服务/存盘；
- 数据计算/处理；
- 数据验证/过滤；
- 数据聚合/分流；
- 数据组合/拆分；
- 数据复制/增殖；
- 等等。

一个模块中的工作协程从一些其它模块接收数据做为输入，并向另一些模块发送输出数据。 换句话数，一个模块可能同时兼任数据消费者和数据产生者的角色。

多个模块一起组成了一个数据流处理系统。

下面将展示一些模块工作协程的实现。这些实现仅仅是为了解释目的，所以它们都很简单，并且它们可能并不高效。

数据生成/搜集/加载

一个数据产生者可能通过以下途径生成数据：

- 加载一个文件、或者读取一个数据库、或者用爬虫抓取网页数据；
- 从一个软件或者硬件系统搜集各种数据；
- 产生一系列随机数；
- 等等。

这里，我们使用一个随机数产生器做为一个数据产生者的例子。 此数据产生者函数没有输入，只有输出。

```
 1| import (
 2|    "crypto/rand"
 3|    "encoding/binary"
 4| )
 5|
 6| func RandomGenerator() <-chan uint64 {
 7|    c := make(chan uint64)
 8|    go func() {
 9|       rnds := make([]byte, 8)
10|       for {
11|          _, err := rand.Read(rnds)
12|          if err != nil {
13|             close(c)
14|             break
15|          }
16|          c <- binary.BigEndian.Uint64(rnds)
17|       }
18|    }()
19|    return c
20| }
```

事实上，此随机数产生器是一个多返回值的future/promise。

一个数据产生者可以在任何时刻关闭返回的通道以结束数据生成。

数据聚合

一个数据聚合模块的工作协程将多个数据流合为一个数据流。 假设数据类型为`int64`，下面这个函数将任意数量的数据流合为一个。

```
 1| func Aggregator(inputs ...<-chan uint64) <-chan uint64 {
 2|    out := make(chan uint64)
 3|    for _, in := range inputs {
 4|       go func(in <-chan uint64) {
 5|          for {
 6|             out <- <-in // <=> out <- (<-in)
 7|          }
 8|       }(in)
 9|    }
10|    return out
11| }
```

一个更完美的实现需要考虑一个输入数据流是否已经关闭。（下面要介绍的其它工作协程同理。）

```
 1| import "sync"
 2|
 3| func Aggregator(inputs ...<-chan uint64) <-chan uint64 {
 4|    output := make(chan uint64)
 5|    var wg sync.WaitGroup
 6|    for _, in := range inputs {
 7|       wg.Add(1)
 8|       go func(int <-chan uint64) {
 9|          defer wg.Done()
10|          // 如果通道in被关闭，此循环将最终结束。
11|          for x := range in {
12|             output <- x
13|          }
14|       }(in)
15|    }
16|    go func() {
17|       wg.Wait()
18|       close(output)
19|    }()
20|    return output
21| }
```

如果被聚合的数据流的数量很小，我们也可以使用一个`select`控制流程代码块来聚合这些数据流。

```
 1| // 假设数据流的数量为2。
 2| ...
 3|    output := make(chan uint64)
 4|    go func() {
 5|       inA, inB := inputs[0], inputs[1]
 6|       for {
 7|          select {
 8|          case v := <- inA: output <- v
 9|          case v := <- inB: output <- v
10|          }
11|       }
12|    }
13| ...
```

数据分流

数据分流是数据聚合的逆过程。数据分流的实现很简单，但在实践中用的并不多。

```
 1| func Divisor(input <-chan uint64, outputs ...chan<- uint64)
{
 2|    for _, out := range outputs {
 3|       go func(o chan<- uint64) {
 4|          for {
 5|             o <- <-input // <=> o <- (<-input)
 6|          }
 7|       }(out)
 8|    }
 9| }
```

数据合成

数据合成将多个数据流中读取的数据合成一个。

下面是一个数据合成工作函数的实现中，从两个不同数据流读取的两个`uint64`值组成了一个新的`uint64`值。 当然，在实践中，数据的组合比这复杂得多。

```
 1| func Composor(inA, inB <-chan uint64) <-chan uint64 {
 2|    output := make(chan uint64)
 3|    go func() {
 4|       for {
 5|          a1, b, a2 := <-inA, <-inB, <-inA
 6|          output <- a1 ^ b & a2
 7|       }
 8|    }()
 9|    return output
10| }
```

数据分解

数据分解是数据合成的逆过程。一个数据分解者从一个通道读取一份数据，并将此数据分解为多份数据。 这里就不举例了。

数据复制/增殖

数据复制（增殖）可以看作是特殊的数据分解。一份输入数据将被复制多份并输出给多个数据流。

一个例子：

```
 1| func Duplicator(in <-chan uint64) (<-chan uint64, <-chan
uint64) {
 2|    outA, outB := make(chan uint64), make(chan uint64)
 3|    go func() {
 4|       for x := range in {
 5|          outA <- x
 6|          outB <- x
 7|       }
 8|    }()
 9|    return outA, outB
10| }
```

数据计算/分析

数据计算和数据分析模块的功能因具体程序不同而有很大的差异。 一般来说，数据分析者接收一份数据并对之加工处理后转换为另一份数据。

下面的简单示例中，每个输入的`uint64`值将被进行位反转后输出。

```
 1| func Calculator(in <-chan uint64, out chan uint64) (<-chan
uint64) {
 2|    if out == nil {
 3|       out = make(chan uint64)
 4|    }
 5|    go func() {
 6|       for x := range in {
 7|          out <- ^x
 8|       }
 9|    }()
10|    return out
11| }
```

数据验证/过滤

一个数据验证或过滤者的任务是检查输入数据的合理性并抛弃不合理的数据。 比如，下面的工作者协程将抛弃所有的非素数。

```
 1| import "math/big"
 2|
 3| func Filter0(input <-chan uint64, output chan uint64) <-
chan uint64 {
 4|    if output == nil {
 5|       output = make(chan uint64)
```

```
 6|    }
 7|    go func() {
 8|       bigInt := big.NewInt(0)
 9|       for x := range input {
10|          bigInt.SetUint64(x)
11|          if bigInt.ProbablyPrime(1) {
12|             output <- x
13|          }
14|       }
15|    }()
16|    return output
17| }
18|
19| func Filter(input <-chan uint64) <-chan uint64 {
20|    return Filter0(input, nil)
21| }
```

请注意这两个函数版本分别被本文下面最后展示的两个例子所使用。

数据服务/存盘

一般，一个数据服务或者存盘模块为一个数据流系统中的最后一个模块。 这里的实现值是简单地将数据输出到终端。

```
 1| import "fmt"
 2|
 3| func Printer(input <-chan uint64) {
 4|    for x := range input {
 5|       fmt.Println(x)
 6|    }
 7| }
```

组装数据流系统

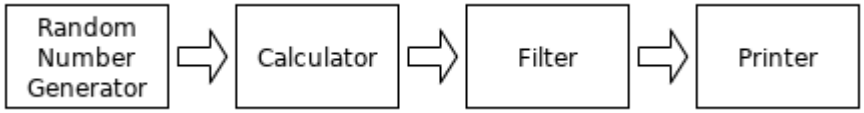
现在，让我们使用上面的模块工作者函数实现来组装一些数据流系统。 组装数据流仅仅是创建一些工作者协程函数调用，并为这些调用指定输入数据流和输出数据流。

数据流系统例子1（一个流线型系统）：

```
 1| package main
 2|
 3| ... // 上面的模块工作者函数实现
```

```
 4|
 5| func main() {
 6|    Printer(
 7|       Filter(
 8|          Calculator(
 9|             RandomGenerator(), nil,
10|          ),
11|       ),
12|    )
13| }
```

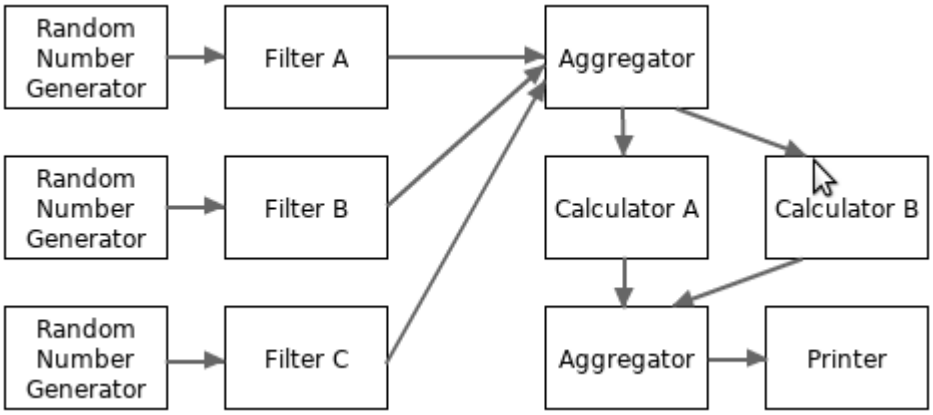
上面这个流线型系统描绘在下图中：



数据流系统例子2（一个单向无环图系统）：

```
 1| package main
 2|
 3| ... // 上面的模块工作者函数实现
 4|
 5| func main() {
 6|    filterA := Filter(RandomGenerator())
 7|    filterB := Filter(RandomGenerator())
 8|    filterC := Filter(RandomGenerator())
 9|    filter := Aggregator(filterA, filterB, filterC)
10|    calculatorA := Calculator(filter, nil)
11|    calculatorB := Calculator(filter, nil)
12|    calculator := Aggregator(calculatorA, calculatorB)
13|    Printer(calculator)
14| }
```

上面这个单向无环图系统描绘在下图中：



更复杂的数据流系统可以表示为任何拓扑结构的图。比如一个复杂的数据流系统可能有多个输出模块。 但是有环拓扑结构的数据流系统在实践中很少用。

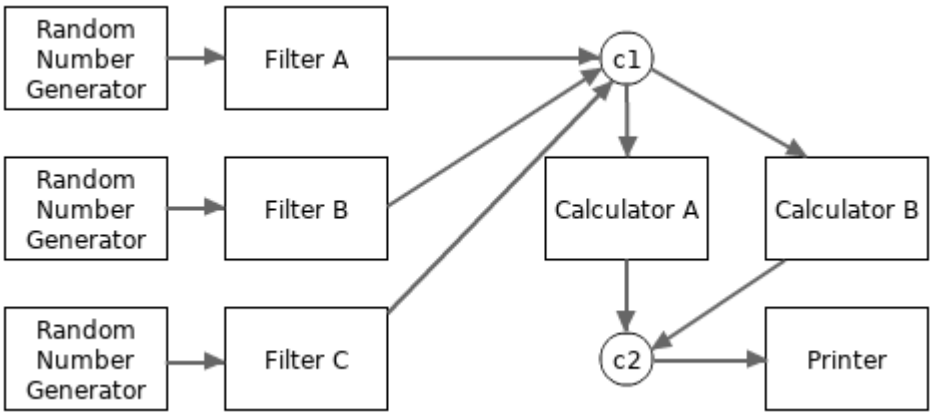
从上面两个例子可以看出，使用通道来构建数据流系统是很简单和直观的。

从上例可以看出，通过使用数据聚合模块，我们可以很轻松地实现各个模块的工作协程数量的扇入（fan-in）和扇出（fan-out）。

事实上，我们也可以使用一个简单的通道来代替数据聚合模块的角色。比如，下面的代码使用两个通道代替了上例中的两个数据聚合器。

```
 1| package main
 2|
 3| ... // 上面的模块工作者函数实现
 4|
 5| func main() {
 6|     c1 := make(chan uint64, 100)
 7|     Filter0(RandomGenerator(), c1) // filterA
 8|     Filter0(RandomGenerator(), c1) // filterB
 9|     Filter0(RandomGenerator(), c1) // filterC
10|     c2 := make(chan uint64, 100)
11|     Calculator(c1, c2) // calculatorA
12|     Calculator(c1, c2) // calculatorB
13|     Printer(c2)
14| }
```

修改后的数据流的拓扑结构如下图所示：



上面的代码示例并没有太多考虑如何关闭一个数据流。请阅读此篇文章（第38章）来了解如何优雅地关闭通道。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

如何优雅地关闭通道

在本文发表数日前，我曾写了一篇文章来解释通道的规则（第21章）。 那篇文章在reddit（https://www.reddit.com/r/golang/comments/5k489v/the_full_list_of_channel_rules_in_golang/）和HN（https://news.ycombinator.com/item?id=13252416）上获得了很多点赞，但也有很多人对Go通道的细节设计提出了一些批评意见。

这些批评主要针对于通道设计中的下列细节：

1. 没有一个简单和通用的方法用来在不改变一个通道的状态的情况下检查这个通道是否已经关闭。
2. 关闭一个已经关闭的通道将产生一个恐慌，所以在不知道一个通道是否已经关闭的时候关闭此通道是很危险的。
3. 向一个已关闭的通道发送数据将产生一个恐慌，所以在不知道一个通道是否已经关闭的时候向此通道发送数据是很危险的。

这些批评看上去有几分道理（实际上属于对通道的不正确使用导致的偏见）。 是的，Go语言中并没有提供一个内置函数来检查一个通道是否已经关闭。

在Go中，如果我们能够保证从不会向一个通道发送数据，那么有一个简单的方法来判断此通道是否已经关闭。 此方法已经在上一篇文章通道用例大全（第37章）中展示过了。 这里为了本文的连贯性，在下面的例子中重新列出了此方法。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type T int
 6|
 7| func IsClosed(ch <-chan T) bool {
 8|    select {
 9|    case <-ch:
10|       return true
11|    default:
12|    }
13|
14|    return false
15| }
16|
17| func main() {
18|    c := make(chan T)
19|    fmt.Println(IsClosed(c)) // false
```

```
20|    close(c)
21|    fmt.Println(IsClosed(c)) // true
22| }
```

如前所述，此方法并不是一个通用的检查通道是否已经关闭的方法。

事实上，即使有一个内置`closed`函数用来检查一个通道是否已经关闭，它的有用性也是十分有限的。 原因是当此函数的一个调用的结果返回时，被查询的通道的状态可能已经又改变了，导致此调用结果并不能反映出被查询的通道的最新状态。 虽然我们可以根据一个调用`closed(ch)`的返回结果为`true`而得出我们不应该再向通道`ch`发送数据的结论， 但是我们不能根据一个调用`closed(ch)`的返回结果为`false`而得出我们可以继续向通道`ch`发送数据的结论。

通道关闭原则

一个常用的使用Go通道的原则是**不要在数据接收方或者在有多个发送者的情况下关闭通道**。 换句话说，我们只应该让一个通道唯一的发送者关闭此通道。

下面我们将称此原则为**通道关闭原则**。

当然，这并不是一个通用的关闭通道的原则。通用的原则是**不要关闭已关闭的通道**。 如果我们能够保证从某个时刻之后，再没有协程将向一个未关闭的非nil通道发送数据，则一个协程可以安全地关闭此通道。 然而，做出这样的保证常常需要很大的努力，从而导致代码过度复杂。 另一方面，遵循**通道关闭原则**是一件相对简单的事儿。

粗鲁地关闭通道的方法

如果由于某种原因，你一定非要从数据接收方或者让众多发送者中的一个关闭一个通道，你可以使用恢复机制（第13章）来防止可能产生的恐慌而导致程序崩溃。 下面就是这样的一个实现（假设通道的元素类型为`T`）。

```
 1| func SafeClose(ch chan T) (justClosed bool) {
 2|    defer func() {
 3|       if recover() != nil {
 4|          // 一个函数的返回结果可以在defer调用中修改。
 5|          justClosed = false
 6|       }
 7|    }()
 8|
 9|    // 假设ch != nil。
10|    close(ch)   // 如果ch已关闭，则产生一个恐慌。
```

```
11|    return true // <=> justClosed = true; return
12| }
```

此方法违反了**通道关闭原则**。

同样的方法可以用来粗鲁地向一个关闭状态未知的通道发送数据。

```
 1| func SafeSend(ch chan T, value T) (closed bool) {
 2|    defer func() {
 3|       if recover() != nil {
 4|          closed = true
 5|       }
 6|    }()
 7|
 8|    ch <- value  // 如果ch已关闭，则产生一个恐慌。
 9|    return false // <=> closed = false; return
10| }
```

这样的粗鲁方法不仅违反了**通道关闭原则**，而且Go白皮书和标准编译器[不保证](https://github.com/golang/go/issues/30372)（https://github.com/golang/go/issues/30372）它的实现中不[存在数据竞争](https://github.com/golang/go/issues/27769)（https://github.com/golang/go/issues/27769）。

礼貌地关闭通道的方法

很多Go程序员喜欢使用`sync.Once`来关闭通道。

```
 1| type MyChannel struct {
 2|    C    chan T
 3|    once sync.Once
 4| }
 5|
 6| func NewMyChannel() *MyChannel {
 7|    return &MyChannel{C: make(chan T)}
 8| }
 9|
10| func (mc *MyChannel) SafeClose() {
11|    mc.once.Do(func() {
12|       close(mc.C)
13|    })
14| }
```

当然，我们也可以使用`sync.Mutex`来防止多次关闭一个通道。

```
 1| type MyChannel struct {
 2|    C      chan T
 3|    closed bool
 4|    mutex  sync.Mutex
 5| }
 6|
 7| func NewMyChannel() *MyChannel {
 8|    return &MyChannel{C: make(chan T)}
 9| }
10|
11| func (mc *MyChannel) SafeClose() {
12|    mc.mutex.Lock()
13|    defer mc.mutex.Unlock()
14|    if !mc.closed {
15|       close(mc.C)
16|       mc.closed = true
17|    }
18| }
19|
20| func (mc *MyChannel) IsClosed() bool {
21|    mc.mutex.Lock()
22|    defer mc.mutex.Unlock()
23|    return mc.closed
24| }
```

这些实现确实比上一节中的方法礼貌一些，但是它们不能完全有效地避免数据竞争。 目前的Go白皮书并不保证发生在一个通道上的并发关闭操作和发送操作不会产生数据竞争。 如果一个`SafeClose`函数和同一个通道上的发送操作同时运行，则数据竞争可能发生（虽然这样的数据竞争一般并不会带来什么危害）。

优雅地关闭通道的方法

上一节中介绍的`SafeSend`函数有一个弊端，它的调用不能做为`case`操作而被使用在`select`代码块中。 另外，很多Go程序员（包括我）认为上面两节展示的关闭通道的方法不是很优雅。 本节下面将介绍一些在各种情形下使用纯通道操作来关闭通道的方法。

（为了演示程序的完整性，下面这些例子中使用到了`sync.WaitGroup`。在实践中，`sync.WaitGroup`并不是必需的。）

情形一：M个接收者和一个发送者。发送者通过关闭用来传输数据的通道来传递发送结束信号

这是最简单的一种情形。当发送者欲结束发送，让它关闭用来传输数据的通道即可。

```
 1| package main
 2|
 3| import (
 4|    "time"
 5|    "math/rand"
 6|    "sync"
 7|    "log"
 8| )
 9|
10| func main() {
11|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
12|    log.SetFlags(0)
13|
14|    // ...
15|    const Max = 100000
16|    const NumReceivers = 100
17|
18|    wgReceivers := sync.WaitGroup{}
19|    wgReceivers.Add(NumReceivers)
20|
21|    // ...
22|    dataCh := make(chan int)
23|
24|    // 发送者
25|    go func() {
26|       for {
27|          if value := rand.Intn(Max); value == 0 {
28|             // 此唯一的发送者可以安全地关闭此数据通道。
29|             close(dataCh)
30|             return
31|          } else {
32|             dataCh <- value
33|          }
34|       }
35|    }()
36|
37|    // 接收者
38|    for i := 0; i < NumReceivers; i++ {
39|       go func() {
40|          defer wgReceivers.Done()
41|
42|          // 接收数据直到通道dataCh已关闭
```

```
43|          // 并且dataCh的缓冲队列已空。
44|          for value := range dataCh {
45|             log.Println(value)
46|          }
47|       }()
48|    }
49|
50|    wgReceivers.Wait()
51| }
```

情形二：一个接收者和N个发送者，此唯一接收者通过关闭一个额外的信号通道来通知发送者不要再发送数据了

此情形比上一种情形复杂一些。我们不能让接收者关闭用来传输数据的通道来停止数据传输，因为这样做违反了**通道关闭原则**。 但是我们可以让接收者关闭一个额外的信号通道来通知发送者不要再发送数据了。

```
 1| package main
 2|
 3| import (
 4|    "time"
 5|    "math/rand"
 6|    "sync"
 7|    "log"
 8| )
 9|
10| func main() {
11|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
12|    log.SetFlags(0)
13|
14|    // ...
15|    const Max = 100000
16|    const NumSenders = 1000
17|
18|    wgReceivers := sync.WaitGroup{}
19|    wgReceivers.Add(1)
20|
21|    // ...
22|    dataCh := make(chan int)
23|    stopCh := make(chan struct{})
24|       // stopCh是一个额外的信号通道。它的
25|       // 发送者为dataCh数据通道的接收者。
26|       // 它的接收者为dataCh数据通道的发送者。
27|
```

```
28|    // 发送者
29|    for i := 0; i < NumSenders; i++ {
30|       go func() {
31|          for {
32|             // 这里的第一个尝试接收用来让此发送者
33|             // 协程尽早地退出。对于这个特定的例子，
34|             // 此select代码块并非必需。
35|             select {
36|             case <- stopCh:
37|                return
38|             default:
39|             }
40|
41|             // 即使stopCh已经关闭，此第二个select
42|             // 代码块中的第一个分支仍很有可能在若干个
43|             // 循环步内依然不会被选中。如果这是不可接受
44|             // 的，则上面的第一个select代码块是必需的。
45|             select {
46|             case <- stopCh:
47|                return
48|             case dataCh <- rand.Intn(Max):
49|             }
50|          }
51|       }()
52|    }
53|
54|    // 接收者
55|    go func() {
56|       defer wgReceivers.Done()
57|
58|       for value := range dataCh {
59|          if value == Max-1 {
60|             // 此唯一的接收者同时也是stopCh通道的
61|             // 唯一发送者。尽管它不能安全地关闭dataCh数
62|             // 据通道，但它可以安全地关闭stopCh通道。
63|             close(stopCh)
64|             return
65|          }
66|
67|          log.Println(value)
68|       }
69|    }()
70|
71|    // ...
```

```
72|    wgReceivers.Wait()
73| }
```

如此例中的注释所述，对于此额外的信号通道`stopCh`，它只有一个发送者，即`dataCh`数据通道的唯一接收者。 `dataCh`数据通道的接收者关闭了信号通道`stopCh`，这是不违反**通道关闭原则**的。

在此例中，数据通道`dataCh`并没有被关闭。是的，我们不必关闭它。 当一个通道不再被任何协程所使用后，它将逐渐被垃圾回收掉，无论它是否已经被关闭。 所以这里的优雅性体现在通过不关闭一个通道来停止使用此通道。

情形三：M个接收者和N个发送者。它们中的任何协程都可以让一个中间调解协程帮忙发出停止数据传送的信号

这是最复杂的一种情形。我们不能让接收者和发送者中的任何一个关闭用来传输数据的通道，我们也不能让多个接收者之一关闭一个额外的信号通道。 这两种做法都违反了**通道关闭原则**。 然而，我们可以引入一个中间调解者角色并让其关闭额外的信号通道来通知所有的接收者和发送者结束工作。 具体实现见下例。注意其中使用了一个尝试发送操作来向中间调解者发送信号。

```
 1| package main
 2|
 3| import (
 4|    "time"
 5|    "math/rand"
 6|    "sync"
 7|    "log"
 8|    "strconv"
 9| )
10|
11| func main() {
12|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
13|    log.SetFlags(0)
14|
15|    // ...
16|    const Max = 100000
17|    const NumReceivers = 10
18|    const NumSenders = 1000
19|
20|    wgReceivers := sync.WaitGroup{}
21|    wgReceivers.Add(NumReceivers)
22|
23|    // ...
```

```
24|    dataCh := make(chan int)
25|    stopCh := make(chan struct{})
26|       // stopCh是一个额外的信号通道。它的发送
27|       // 者为中间调解者。它的接收者为dataCh
28|       // 数据通道的所有的发送者和接收者。
29|    toStop := make(chan string, 1)
30|       // toStop是一个用来通知中间调解者让其
31|       // 关闭信号通道stopCh的第二个信号通道。
32|       // 此第二个信号通道的发送者为dataCh数据
33|       // 通道的所有的发送者和接收者，它的接收者
34|       // 为中间调解者。它必须为一个缓冲通道。
35|
36|    var stoppedBy string
37|
38|    // 中间调解者
39|    go func() {
40|       stoppedBy = <-toStop
41|       close(stopCh)
42|    }()
43|
44|    // 发送者
45|    for i := 0; i < NumSenders; i++ {
46|       go func(id string) {
47|          for {
48|             value := rand.Intn(Max)
49|             if value == 0 {
50|                // 为了防止阻塞，这里使用了一个尝试
51|                // 发送操作来向中间调解者发送信号。
52|                select {
53|                case toStop <- "发送者#" + id:
54|                default:
55|                }
56|                return
57|             }
58|
59|             // 此处的尝试接收操作是为了让此发送协程尽早
60|             // 退出。标准编译器对尝试接收和尝试发送做了
61|             // 特殊的优化，因而它们的速度很快。
62|             select {
63|             case <- stopCh:
64|                return
65|             default:
66|             }
67|
68|             // 即使stopCh已关闭，如果这个select代码块
```

```
69|             // 中第二个分支的发送操作是非阻塞的，则第一个
70|             // 分支仍很有可能在若干个循环步内依然不会被选
71|             // 中。如果这是不可接受的，则上面的第一个尝试
72|             // 接收操作代码块是必需的。
73|             select {
74|             case <- stopCh:
75|                return
76|             case dataCh <- value:
77|             }
78|          }
79|       }(strconv.Itoa(i))
80|    }
81|
82|    // 接收者
83|    for i := 0; i < NumReceivers; i++ {
84|       go func(id string) {
85|          defer wgReceivers.Done()
86|
87|          for {
88|             // 和发送者协程一样，此处的尝试接收操作是为了
89|             // 让此接收协程尽早退出。
90|             select {
91|             case <- stopCh:
92|                return
93|             default:
94|             }
95|
96|             // 即使stopCh已关闭，如果这个select代码块
97|             // 中第二个分支的接收操作是非阻塞的，则第一个
98|             // 分支仍很有可能在若干个循环步内依然不会被选
99|             // 中。如果这是不可接受的，则上面尝试接收操作
100
  |             // 代码块是必需的。
101
  |             select {
102
  |             case <- stopCh:
103
  |                return
104
  |             case value := <-dataCh:
105
  |                if value == Max-1 {
106
  |                   // 为了防止阻塞，这里使用了一个尝试
```

```
107
  |                    // 发送操作来向中间调解者发送信号。
108
  |                    select {
109
  |                    case toStop <- "接收者#" + id:
110
  |                    default:
111
  |                    }
112
  |                    return
113
  |                }
114
  |
115
  |                log.Println(value)
116
  |            }
117
  |          }
118
  |       }(strconv.Itoa(i))
119
  |    }
120
  |
121
  |    // ...
122
  |    wgReceivers.Wait()
123
  |    log.Println("被" + stoppedBy + "终止了")
124
  | }
```

在此例中，**通道关闭原则**依旧得到了遵守。

请注意，信号通道`toStop`的容量必须至少为1。 如果它的容量为0，则在中间调解者还未准备好的情况下就已经有某个协程向`toStop`发送信号时，此信号将被抛弃。

我们也可以不使用尝试发送操作向中间调解者发送信号，但信号通道`toStop`的容量必须至少为数据发送者和数据接收者的数量之和，以防止向其发送数据时（有一个极其微小的可能）导致某些发送者和接收者协程永久阻塞。

```
 1| ...
 2| toStop := make(chan string, NumReceivers + NumSenders)
 3| ...
 4|          value := rand.Intn(Max)
 5|          if value == 0 {
 6|             toStop <- "sender#" + id
 7|             return
 8|          }
 9| ...
10|             if value == Max-1 {
11|                toStop <- "receiver#" + id
12|                return
13|             }
14| ...
```

情形四：“M个接收者和一个发送者”情形的一个变种：用来传输数据的通道的关闭请求由第三方发出

有时，数据通道（dataCh）的关闭请求需要由某个第三方协程发出。对于这种情形，我们可以使用一个额外的信号通道来通知唯一的发送者关闭数据通道（dataCh）。

```
 1| package main
 2|
 3| import (
 4|    "time"
 5|    "math/rand"
 6|    "sync"
 7|    "log"
 8| )
 9|
10| func main() {
11|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
12|    log.SetFlags(0)
13|
14|    // ...
15|    const Max = 100000
16|    const NumReceivers = 100
17|    const NumThirdParties = 15
18|
19|    wgReceivers := sync.WaitGroup{}
20|    wgReceivers.Add(NumReceivers)
21|
```

```
22|    // ...
23|    dataCh := make(chan int)
24|    closing := make(chan struct{}) // 信号通道
25|    closed := make(chan struct{})
26|
27|    // 此stop函数可以被安全地多次调用。
28|    stop := func() {
29|       select {
30|       case closing<-struct{}{}:
31|          <-closed
32|       case <-closed:
33|       }
34|    }
35|
36|    // 一些第三方协程
37|    for i := 0; i < NumThirdParties; i++ {
38|       go func() {
39|          r := 1 + rand.Intn(3)
40|          time.Sleep(time.Duration(r) * time.Second)
41|          stop()
42|       }()
43|    }
44|
45|    // 发送者
46|    go func() {
47|       defer func() {
48|          close(closed)
49|          close(dataCh)
50|       }()
51|
52|       for {
53|          select{
54|          case <-closing: return
55|          default:
56|          }
57|
58|          select{
59|          case <-closing: return
60|          case dataCh <- rand.Intn(Max):
61|          }
62|       }
63|    }()
64|
65|    // 接收者
66|    for i := 0; i < NumReceivers; i++ {
```

```
67|       go func() {
68|          defer wgReceivers.Done()
69|
70|          for value := range dataCh {
71|             log.Println(value)
72|          }
73|       }()
74|    }
75|
76|    wgReceivers.Wait()
77| }
```

上述代码中的stop函数中使用的技巧偷自Roger Peppe在此贴（https://groups.google.com/forum/#!msg/golang-nuts/1EKehHH7kZY/SRmCtXDZAAAJ）中的一个留言。

情形五：“N个发送者”的一个变种：用来传输数据的通道必须被关闭以通知各个接收者数据发送已经结束了

在上面的提到的“N个发送者”情形中，为了遵守**通道关闭原则**，我们避免了关闭数据通道（dataCh）。但是有时候，数据通道（dataCh）必须被关闭以通知各个接收者数据发送已经结束。对于这种“N个发送者”情形，我们可以使用一个中间通道将它们转化为“一个发送者”情形，然后继续使用上一节介绍的技巧来关闭此中间通道，从而避免了关闭原始的dataCh数据通道。

```
 1| package main
 2|
 3| import (
 4|    "time"
 5|    "math/rand"
 6|    "sync"
 7|    "log"
 8|    "strconv"
 9| )
10|
11| func main() {
12|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
13|    log.SetFlags(0)
14|
15|    // ...
16|    const Max = 1000000
17|    const NumReceivers = 10
18|    const NumSenders = 1000
```

```
19|    const NumThirdParties = 15
20|
21|    wgReceivers := sync.WaitGroup{}
22|    wgReceivers.Add(NumReceivers)
23|
24|    // ...
25|    dataCh := make(chan int)   // 将被关闭
26|    middleCh := make(chan int) // 不会被关闭
27|    closing := make(chan string)
28|    closed := make(chan struct{})
29|
30|    var stoppedBy string
31|
32|    stop := func(by string) {
33|       select {
34|       case closing <- by:
35|          <-closed
36|       case <-closed:
37|       }
38|    }
39|
40|    // 中间层
41|    go func() {
42|       exit := func(v int, needSend bool) {
43|          close(closed)
44|          if needSend {
45|             dataCh <- v
46|          }
47|          close(dataCh)
48|       }
49|
50|       for {
51|          select {
52|          case stoppedBy = <-closing:
53|             exit(0, false)
54|             return
55|          case v := <- middleCh:
56|             select {
57|             case stoppedBy = <-closing:
58|                exit(v, true)
59|                return
60|             case dataCh <- v:
61|             }
62|          }
63|       }
```

```
64|    }()
65|
66|    // 一些第三方协程
67|    for i := 0; i < NumThirdParties; i++ {
68|       go func(id string) {
69|          r := 1 + rand.Intn(3)
70|          time.Sleep(time.Duration(r) * time.Second)
71|          stop("3rd-party#" + id)
72|       }(strconv.Itoa(i))
73|    }
74|
75|    // 发送者
76|    for i := 0; i < NumSenders; i++ {
77|       go func(id string) {
78|          for {
79|             value := rand.Intn(Max)
80|             if value == 0 {
81|                stop("sender#" + id)
82|                return
83|             }
84|
85|             select {
86|             case <- closed:
87|                return
88|             default:
89|             }
90|
91|             select {
92|             case <- closed:
93|                return
94|             case middleCh <- value:
95|             }
96|          }
97|       }(strconv.Itoa(i))
98|    }
99|
100
  |    // 接收者
101
  |    for range [NumReceivers]struct{}{} {
102
  |       go func() {
103
  |          defer wgReceivers.Done()
```

```
104
  |
105
  |           for value := range dataCh {
106
  |               log.Println(value)
107
  |           }
108
  |       }()
109
  |    }
110
  |
111
  |    // ...
112
  |    wgReceivers.Wait()
113
  |    log.Println("stopped by", stoppedBy)
114
  | }
```

更多情形?

在日常编程中可能会遇到更多的变种情形，但是上面介绍的情形是最常见和最基本的。 通过聪明地使用通道（和其它并发同步技术），我相信，对于各种变种，我们总会找到相应的遵守**通道关闭原则**的解决方法。

结论

并没有什么情况非得逼得我们违反**通道关闭原则**。 如果你遇到了此情形，请考虑修改你的代码流程和结构设计。

使用通道编程宛如在艺术创作一般！

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

sync标准库包中提供的并发同步技术

通道用例大全（第37章）一文中介绍了很多通过使用通道来实现并发同步的用例。事实上，通道并不是Go支持的唯一的一种并发同步技术。而且对于一些特定的情形，通道并不是最有效和可读性最高的同步技术。 本文下面将介绍sync标准库包中提供的各种并发同步技术。相对于通道，这些技术对于某些情形更加适用。

sync标准库包提供了一些用于实现并发同步的类型。这些类型适用于各种不同的内存顺序需求。 对于这些特定的需求，这些类型使用起来比通道效率更高，代码实现更简洁。

（请注意：为了避免各种异常行为，最好不要复制sync标准库包中提供的类型的值。）

sync.WaitGroup（等待组）类型

每个sync.WaitGroup值在内部维护着一个计数，此计数的初始默认值为零。

*sync.WaitGroup类型有三个方法（https://golang.google.cn/pkg/sync/#WaitGroup）：Add(delta int)、Done()和Wait()。

对于一个可寻址的sync.WaitGroup值wg，

- 我们可以使用方法调用wg.Add(delta)来改变值wg维护的计数。
- 方法调用wg.Done()和wg.Add(-1)是完全等价的。
- 如果一个wg.Add(delta)或者wg.Done()调用将wg维护的计数更改成一个负数，一个恐慌将产生。
- 当一个协程调用了wg.Wait()时，
    - 如果此时wg维护的计数为零，则此wg.Wait()此操作为一个空操作（no-op）；
    - 否则（计数为一个正整数），此协程将进入阻塞状态。 当以后其它某个协程将此计数更改至0时（一般通过调用wg.Done()），此协程将重新进入运行状态（即wg.Wait()将返回）。

请注意wg.Add(delta)、wg.Done()和wg.Wait()分别是(&wg).Add(delta)、(&wg).Done()和(&wg).Wait()的简写形式。

一般，一个sync.WaitGroup值用来让某个协程等待其它若干协程都先完成它们各自的任务。 一个例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "math/rand"
 6|    "sync"
 7|    "time"
 8| )
 9|
10| func main() {
11|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
12|
13|    const N = 5
14|    var values [N]int32
15|
16|    var wg sync.WaitGroup
17|    wg.Add(N)
18|    for i := 0; i < N; i++ {
19|       i := i
20|       go func() {
21|          values[i] = 50 + rand.Int31n(50)
22|          fmt.Println("Done:", i)
23|          wg.Done() // <=> wg.Add(-1)
24|       }()
25|    }
26|
27|    wg.Wait()
28|    // 所有的元素都保证被初始化了。
29|    fmt.Println("values:", values)
30| }
```

在此例中，主协程等待着直到其它5个协程已经将各自负责的元素初始化完毕此会打印出各个元素值。 这里是一个可能的程序执行输出结果：

```
Done: 4
Done: 1
Done: 3
Done: 0
Done: 2
values: [71 89 50 62 60]
```

我们可以将上例中的Add方法调用拆分成多次调用：

```
 1| ...
 2|    var wg sync.WaitGroup
 3|    for i := 0; i < N; i++ {
 4|       wg.Add(1) // 将被执行5次
 5|       i := i
 6|       go func() {
 7|          values[i] = 50 + rand.Int31n(50)
 8|          wg.Done()
 9|       }()
10|    }
11| ...
```

一个*sync.WaitGroup值的Wait方法可以在多个协程中调用。 当对应的sync.WaitGroup值维护的计数降为0，这些协程都将得到一个（广播）通知而结束阻塞状态。

```
 1| func main() {
 2|    rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
 3|
 4|    const N = 5
 5|    var values [N]int32
 6|
 7|    var wgA, wgB sync.WaitGroup
 8|    wgA.Add(N)
 9|    wgB.Add(1)
10|
11|    for i := 0; i < N; i++ {
12|       i := i
13|       go func() {
14|          wgB.Wait() // 等待广播通知
15|          log.Printf("values[%v]=%v \n", i, values[i])
16|          wgA.Done()
17|       }()
18|    }
19|
20|    // 下面这个循环保证将在上面的任何一个
21|    // wg.Wait调用结束之前执行。
22|    for i := 0; i < N; i++ {
23|       values[i] = 50 + rand.Int31n(50)
24|    }
25|    wgB.Done() // 发出一个广播通知
26|    wgA.Wait()
27| }
```

一个WaitGroup可以在它的一个Wait方法返回之后被重用。 但是请注意，当一个WaitGroup值维护的基数为零时，它的带有正整数实参的Add方法调用不能和它的Wait方法调用并发运行，否则将可能出现数据竞争。

sync.Once类型

每个*sync.Once值有一个Do(f func())方法。 此方法只有一个类型为func()的参数。

对一个可寻址的sync.Once值o， o.Do()（即(&o).Do()的简写形式）方法调用可以在多个协程中被多次并发地执行， 这些方法调用的实参应该（但并不强制）为同一个函数值。 在这些方法调用中，有且只有一个调用的实参函数（值）将得到调用。 此被调用的实参函数保证在任何o.Do()方法调用返回之前退出。 换句话说，被调用的实参函数内的代码将在任何o.Do()方法返回调用之前被执行。

一般来说，一个sync.Once值被用来确保一段代码在一个并发程序中被执行且仅被执行一次。

一个例子：

```
 1| package main
 2|
 3| import (
 4|    "log"
 5|    "sync"
 6| )
 7|
 8| func main() {
 9|    log.SetFlags(0)
10|
11|    x := 0
12|    doSomething := func() {
13|       x++
14|       log.Println("Hello")
15|    }
16|
17|    var wg sync.WaitGroup
18|    var once sync.Once
19|    for i := 0; i < 5; i++ {
20|       wg.Add(1)
21|       go func() {
22|          defer wg.Done()
23|          once.Do(doSomething)
```

```
24|          log.Println("world!")
25|       }()
26|    }
27|
28|    wg.Wait()
29|    log.Println("x =", x) // x = 1
30| }
```

在此例中，Hello将仅被输出一次，而world!将被输出5次，并且Hello肯定在所有的5个world!之前输出。

sync.Mutex（互斥锁）和sync.RWMutex（读写锁）类型

*sync.Mutex和*sync.RWMutex类型都实现了[sync.Locker接口类型](https://golang.google.cn/pkg/sync/#Locker)（https://golang.google.cn/pkg/sync/#Locker）。 所以这两个类型都有两个方法：Lock()和Unlock()，用来保护一份数据不会被多个使用者同时读取和修改。

除了Lock()和Unlock()这两个方法，*sync.RWMutex类型还有两个另外的方法：RLock()和RUnlock()，用来支持多个读取者并发读取一份数据但防止此份数据被某个数据写入者和其它数据访问者（包括读取者和写入者）同时使用。

（注意：这里的***数据读取者***和***数据写入者***不应该从字面上理解。有时候某些数据读取者可能修改数据，而有些数据写入者可能只读取数据。）

一个Mutex值常称为一个互斥锁。 一个Mutex零值为一个尚未加锁的互斥锁。 一个（可寻址的）Mutex值m只有在未加锁状态时才能通过m.Lock()方法调用被成功加锁。 换句话说，一旦m值被加了锁（亦即某个m.Lock()方法调用成功返回），一个新的加锁试图将导致当前协程进入阻塞状态，直到此Mutex值被解锁为止（通过m.Unlock()方法调用）。

注意：m.Lock()和m.Unlock()分别是(&m).Lock()和(&m).Unlock()的简写形式。

一个使用sync.Mutex的例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "runtime"
 6|    "sync"
```

```
 7| )
 8|
 9| type Counter struct {
10|    m sync.Mutex
11|    n uint64
12| }
13|
14| func (c *Counter) Value() uint64 {
15|    c.m.Lock()
16|    defer c.m.Unlock()
17|    return c.n
18| }
19|
20| func (c *Counter) Increase(delta uint64) {
21|    c.m.Lock()
22|    c.n += delta
23|    c.m.Unlock()
24| }
25|
26| func main() {
27|    var c Counter
28|    for i := 0; i < 100; i++ {
29|       go func() {
30|          for k := 0; k < 100; k++ {
31|             c.Increase(1)
32|          }
33|       }()
34|    }
35|
36|    // 此循环仅为演示目的。
37|    for c.Value() < 10000 {
38|       runtime.Gosched()
39|    }
40|    fmt.Println(c.Value()) // 10000
41| }
```

在上面这个例子中，一个`Counter`值使用了一个`Mutex`字段来确保它的字段`n`永远不会被多个协程同时使用。

一个`RWMutex`值常称为一个读写互斥锁，它的内部包含两个锁：一个写锁和一个读锁。 对于一个可寻址的`RWMutex`值`rwm`，数据写入者可以通过方法调用`rwm.Lock()`对`rwm`加写锁，或者通过`rwm.RLock()`方法调用对`rwm`加读锁。 方法调用`rwm.Unlock()`和`rwm.RUnlock()`用来解开`rwm`的写锁和读锁。 `rwm`的读锁维护着一个计数。当`rwm.RLock()`调用成功时，此计数增1；当`rwm.Unlock()`调

用成功时，此计数减1； 一个零计数表示rwm的读锁处于未加锁状态；反之，一个非零计数（肯定大于零）表示rwm的读锁处于加锁状态。

注意rwm.Lock()、rwm.Unlock()、rwm.RLock()和rwm.RUnlock()分别是(&rwm).Lock()、(&rwm).Unlock()、(&rwm).RLock()和(&rwm).RUnlock()的简写形式。

对于一个可寻址的RWMutex值rwm，下列规则存在：

- rwm的写锁只有在它的写锁和读锁都处于未加锁状态时才能被成功加锁。 换句话说， rwm的写锁在任何时刻最多只能被一个数据写入者成功加锁，并且rwm的写锁和读锁不能同时处于加锁状态。
- 当rwm的写锁正处于加锁状态的时候，任何新的对之加写锁或者加读锁的操作试图都将导致当前协程进入阻塞状态，直到此写锁被解锁，这样的操作试图才有机会成功。
- 当rwm的读锁正处于加锁状态的时候，新的加写锁的操作试图将导致当前协程进入阻塞状态。 但是，一个新的加读锁的操作试图将成功，只要此操作试图发生在任何被阻塞的加写锁的操作试图之前（见下一条规则）。 换句话说，一个读写互斥锁的读锁可以同时被多个数据读取者同时加锁而持有。 当rwm的读锁维护的计数清零时，读锁将返回未加锁状态。
- 假设rwm的读锁正处于加锁状态的时候，为了防止后续数据写入者没有机会成功加写锁，后续发生在某个被阻塞的加写锁操作试图之后的所有加读锁的试图都将被阻塞。
- 假设rwm的写锁正处于加锁状态的时候，（至少对于标准编译器来说，）为了防止后续数据读取者没有机会成功加读锁，发生在此写锁下一次被解锁之前的所有加读锁的试图都将在此写锁下一次被解锁之后肯定取得成功，即使所有这些加读锁的试图发生在一些仍被阻塞的加写锁的试图之后。

后两条规则是为了确保数据读取者和写入者都有机会执行它们的操作。

请注意：一个锁并不会绑定到一个协程上，即一个锁并不记录哪个协程成功地加锁了它。 换句话说，一个锁的加锁者和此锁的解锁者可以不是同一个协程，尽管在实践中这种情况并不多见。

在上一个例子中，如果Value方法被十分频繁调用而Increase方法并不频繁被调用，则Counter类型的m字段的类型可以更改为sync.RWMutex，从而使得执行效率更高，如下面的代码所示。

```
1| ...
2| type Counter struct {
3|    //m sync.Mutex
4|    m sync.RWMutex
5|    n uint64
```

```
 6| }
 7|
 8| func (c *Counter) Value() uint64 {
 9|    //c.m.Lock()
10|    //defer c.m.Unlock()
11|    c.m.RLock()
12|    defer c.m.RUnlock()
13|    return c.n
14| }
15| ...
```

sync.RWMutex值的另一个应用场景是将一个写任务分隔成若干小的写任务。下一节中展示了一个这样的例子。

根据上面列出的后两条规则，下面这个程序最有可能输出abdc。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6|    "sync"
 7| )
 8|
 9| func main() {
10|    var m sync.RWMutex
11|    go func() {
12|       m.RLock()
13|       fmt.Print("a")
14|       time.Sleep(time.Second)
15|       m.RUnlock()
16|    }()
17|    go func() {
18|       time.Sleep(time.Second * 1 / 4)
19|       m.Lock()
20|       fmt.Print("b")
21|       time.Sleep(time.Second)
22|       m.Unlock()
23|    }()
24|    go func() {
25|       time.Sleep(time.Second * 2 / 4)
26|       m.Lock()
27|       fmt.Print("c")
28|       m.Unlock()
29|    }()
```

```
30|    go func () {
31|       time.Sleep(time.Second * 3 / 4)
32|       m.RLock()
33|       fmt.Print("d")
34|       m.RUnlock()
35|    }()
36|    time.Sleep(time.Second * 3)
37|    fmt.Println()
38| }
```

请注意，上例这个程序仅仅是为了解释和验证上面列出的读写锁的后两条加锁规则。 此程序使用了`time.Sleep`调用来做协程间的同步。这种所谓的同步方法不应该被使用在生产代码中（第42章）。

`sync.Mutex`和`sync.RWMutex`值也可以用来实现通知，尽管这不是Go中最优雅的方法来实现通知。 下面是一个使用了`Mutex`值来实现通知的例子。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync"
 6|    "time"
 7| )
 8|
 9| func main() {
10|    var m sync.Mutex
11|    m.Lock()
12|    go func() {
13|       time.Sleep(time.Second)
14|       fmt.Println("Hi")
15|       m.Unlock() // 发出一个通知
16|    }()
17|    m.Lock() // 等待通知
18|    fmt.Println("Bye")
19| }
```

在此例中，`Hi`将确保在`Bye`之前打印出来。 关于`sync.Mutex`和`sync.RWMutex`值相关的内存顺序保证，请阅读Go中的内存顺序保证（第41章）一文。

sync.Cond类型

`sync.Cond`类型提供了一种有效的方式来实现多个协程间的通知。

每个sync.Cond值拥有一个sync.Locker类型的名为L的字段。 此字段的具体值常常为一个*sync.Mutex值或者*sync.RWMutex值。

*sync.Cond类型有三个方法（https://golang.google.cn/pkg/sync/#Cond）：Wait()、Signal()和Broadcast()。

每个Cond值维护着一个先进先出等待协程队列。 对于一个可寻址的Cond值c，

- c.Wait()必须在c.L字段值的锁处于加锁状态的时候调用；否则，c.Wait()调用将造成一个恐慌。 一个c.Wait()调用将
  1. 首先将当前协程推入到c所维护的等待协程队列；
  2. 然后调用c.L.Unlock()对c.L的锁解锁；
  3. 然后使当前协程进入阻塞状态；

  （*当前协程将被另一个协程通过c.Signal()或c.Broadcast()调用唤醒而重新进入运行状态。*）

  一旦当前协程重新进入运行状态，c.L.Lock()将被调用以试图重新对c.L字段值的锁加锁。 此c.Wait()调用将在此试图成功之后退出。
- 一个c.Signal()调用将唤醒并移除c所维护的等待协程队列中的第一个协程（如果此队列不为空的话）。
- 一个c.Broadcast()调用将唤醒并移除c所维护的等待协程队列中的所有协程（如果此队列不为空的话）。

请注意：c.Wait()、c.Signal()和c.Broadcast()分别为(&c).Wait()、(&c).Signal()和(&c).Broadcast()的简写形式。

c.Signal()和c.Broadcast()调用常用来通知某个条件的状态发生了变化。 一般说来，c.Wait()应该在一个检查某个条件是否已经得到满足的循环中调用。

下面是一个典型的sync.Cond用例。

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "math/rand"
 6|     "sync"
 7|     "time"
 8| )
 9|
10| func main() {
11|     rand.Seed(time.Now().UnixNano()) // Go 1.20之前需要
```

```
12|
13|    const N = 10
14|    var values [N]string
15|
16|    cond := sync.NewCond(&sync.Mutex{})
17|
18|    for i := 0; i < N; i++ {
19|       d := time.Second * time.Duration(rand.Intn(10)) / 10
20|       go func(i int) {
21|          time.Sleep(d) // 模拟一个工作负载
22|          cond.L.Lock()
23|          // 下面的修改必须在cond.L被锁定的时候执行
24|          values[i] = string('a' + i)
25|          cond.Broadcast() // 可以在cond.L被解锁后发出通知
26|          cond.L.Unlock()
27|          // 上面的通知也可以在cond.L未锁定的时候发出。
28|          //cond.Broadcast() // 上面的调用也可以放在这里
29|       }(i)
30|    }
31|
32|    // 此函数必须在cond.L被锁定的时候调用。
33|    checkCondition := func() bool {
34|       fmt.Println(values)
35|       for i := 0; i < N; i++ {
36|          if values[i] == "" {
37|             return false
38|          }
39|       }
40|       return true
41|    }
42|
43|    cond.L.Lock()
44|    defer cond.L.Unlock()
45|    for !checkCondition() {
46|       cond.Wait() // 必须在cond.L被锁定的时候调用
47|    }
48| }
```

一个可能的输出：

```
[         ]
[     f    ]
[  c   f    ]
[  c   f  h  ]
[ b c   f  h  ]
```

```
[a b c   f  h  j]
[a b c   f g h i j]
[a b c  e f g h i j]
[a b c d e f g h i j]
```

因为上例中只有一个协程（主协程）在等待通知，所以其中的cond.Broadcast()调用也可以换为cond.Signal()。 如上例中的注释所示，cond.Broadcast()和cond.Signal()不必在cond.L的锁处于加锁状态时调用。

为了防止数据竞争，对自定义条件的修改必须在cond.L的锁处于加锁状态时才能执行。 另外，checkCondition函数和cond.Wait方法也必须在cond.L的锁处于加锁状态时才可被调用。

事实上，对于上面这个特定的例子，cond.L字段的也可以为一个*sync.RWMutex值。 对自定义条件的十个部分的修改可以在RWMutex值的读锁处于加锁状态时执行。这十个修改可以并发进行，因为它们是互不干扰的。 如下面的代码所示：

```
 1| ...
 2|    cond := sync.NewCond(&sync.RWMutex{})
 3|    cond.L.Lock()
 4|
 5|    for i := 0; i < N; i++ {
 6|       d := time.Second * time.Duration(rand.Intn(10)) / 10
 7|       go func(i int) {
 8|          time.Sleep(d)
 9|          cond.L.(*sync.RWMutex).RLock()
10|          values[i] = string('a' + i)
11|          cond.L.(*sync.RWMutex).RUnlock()
12|          cond.Signal()
13|       }(i)
14|    }
15| ...
```

在上面的代码中，此sync.RWMutex值的用法有些不符常规。 它的读锁被一些修改数组元素的协程所加锁并持有，而它的写锁被主协程加锁持有用来读取并检查各个数组元素的值。

Cond值所表示的自定义条件可以是一个虚无。对于这种情况，此Cond值纯粹被用来实现通知。 比如，下面这个程序将打印出abc或者bac。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
```

```
 5|    "sync"
 6| )
 7|
 8| func main() {
 9|    wg := sync.WaitGroup{}
10|    wg.Add(1)
11|    cond := sync.NewCond(&sync.Mutex{})
12|    cond.L.Lock()
13|    go func() {
14|       cond.L.Lock()
15|       go func() {
16|          cond.L.Lock()
17|          cond.Broadcast()
18|          cond.L.Unlock()
19|       }()
20|       cond.Wait()
21|       fmt.Print("a")
22|       cond.L.Unlock()
23|       wg.Done()
24|    }()
25|    cond.Wait()
26|    fmt.Print("b")
27|    cond.L.Unlock()
28|    wg.Wait()
29|    fmt.Println("c")
30| }
```

如果需要，多个`sync.Cond`值可以共享一个`sync.Locker`值。但是这种情形在实践中并不多见。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

sync/atomic标准库包中提供的原子操作

原子操作是比其它同步技术更基础的操作。原子操作是无锁的，常常直接通过CPU指令直接实现。 事实上，其它同步技术的实现常常依赖于原子操作。

注意，本文中的很多例子并非并发程序。它们只是用来演示如何使用sync/atomic标准库包中提供的原子操作。

Go 1.19之前的版本中支持的原子操作概述

对于一个整数类型T，sync/atomic标准库包提供了下列原子操作函数。 其中T可以是内置int32、int64、uint32、uint64和uintptr类型。

```
1| func AddT(addr *T, delta T)(new T)
2| func LoadT(addr *T) (val T)
3| func StoreT(addr *T, val T)
4| func SwapT(addr *T, new T) (old T)
5| func CompareAndSwapT(addr *T, old, new T) (swapped bool)
```

比如，下列五个原子操作函数提供给了内置int32类型。

```
1| func AddInt32(addr *int32, delta int32)(new int32)
2| func LoadInt32(addr *int32) (val int32)
3| func StoreInt32(addr *int32, val int32)
4| func SwapInt32(addr *int32, new int32) (old int32)
5| func CompareAndSwapInt32(addr *int32,
6|             old, new int32) (swapped bool)
```

下列四个原子操作函数提供给了（安全）指针类型。 因为这几个函数被引入标准库的时候，Go还不支持自定义泛型，所以这些函数是通过非类型安全指针（第25章）unsafe.Pointer来实现的。

```
1| func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
2| func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
3| func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer,
4|             ) (old unsafe.Pointer)
5| func CompareAndSwapPointer(addr *unsafe.Pointer,
6|             old, new unsafe.Pointer) (swapped bool)
```

因为Go（安全）指针不支持算术运算，所以相对于整数类型，指针类型的原子操作少了一个AddPointer函数。

sync/atomic标准库包也提供了一个Value类型，以它为基的指针类型*Value拥有四个方法（见下，其中后两个是从Go 1.17开始才支持的）。 Value值用来原子读取和修改任何类型的Go值。

```
 1| func (*Value) Load() (x interface{})
 2| func (*Value) Store(x interface{})
 3| func (*Value) Swap(new interface{}) (old interface{})
 4| func (*Value) CompareAndSwap(old, new interface{}) (swapped
bool)
```

Go 1.19+ 版本中新增的原子操作概述

Go 1.19引入了几个各自拥有若干方法的类型用来实现上一节中列出的函数提供的同样的功能。

在这些类型中，Int32、Int64、Uint32、Uint64和Uintptr用来实现整数原子操作。 下面列出的是atomic.Int32类型的方法。其它四个类型的方法是类似的。

```
 1| func (*Int32) Add(delta int32) (new int32)
 2| func (*Int32) Load() int32
 3| func (*Int32) Store(val int32)
 4| func (*Int32) Swap(new int32) (old int32)
 5| func (*Int32) CompareAndSwap(old, new int32) (swapped bool)
```

从Go 1.18开始，Go已经开始支持自定义泛型。 从Go 1.19开始，一些标准库包开始使用自定义泛型，这其中包括sync/atomic标准库包。 此包在Go 1.19中引入的atomic.Pointer[T any]类型就是一个泛型类型。 下面列出了它的方法：

```
 1| (*Pointer[T]) Load() *T
 2| (*Pointer[T]) Store(val *T)
 3| (*Pointer[T]) Swap(new *T) (old *T)
 4| (*Pointer[T]) CompareAndSwap(old, new *T) (swapped bool)
```

Go 1.19也引入了一个Bool类型来进行布尔原子操作。

整数原子操作

本文的余下部分将通过一些示例来展示如何使用这些原子操作函数。

下面这个例子展示了如何使用Add原子操作来并发地递增一个int32值。 在此例子中，主协程中创建了1000个新协程。每个新协程将整数n的值增加1。 原子操作保证这1000个新协程之间不会发生数据竞争。此程序肯定打印出1000。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync"
 6|    "sync/atomic"
 7| )
 8|
 9| func main() {
10|    var n int32
11|    var wg sync.WaitGroup
12|    for i := 0; i < 1000; i++ {
13|       wg.Add(1)
14|       go func() {
15|          atomic.AddInt32(&n, 1)
16|          wg.Done()
17|       }()
18|    }
19|    wg.Wait()
20|
21|    fmt.Println(atomic.LoadInt32(&n)) // 1000
22| }
```

如果我们将新协程中的语句`atomic.AddInt32(&n, 1)`替换为`n++`，则最后的输出结果很可能不是`1000`。

下面的代码使用Go 1.19引入的`atomic.Int32`类型和它的方法重新实现了上面的程序。 此实现略显整洁。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync"
 6|    "sync/atomic"
 7| )
 8|
 9| func main() {
10|    var n atomic.Int32
11|    var wg sync.WaitGroup
12|    for i := 0; i < 1000; i++ {
13|       wg.Add(1)
14|       go func() {
15|          n.Add(1)
16|          wg.Done()
```

```
17|       }()
18|    }
19|    wg.Wait()
20|
21|    fmt.Println(n.Load()) // 1000
22| }
```

StoreT和LoadT原子操作函数或者方法经常被用来需要并发运行的实现setter和getter方法。 下面的例子使用了原子操作函数：

```
 1| type Page struct {
 2|    views uint32
 3| }
 4|
 5| func (page *Page) SetViews(n uint32) {
 6|    atomic.StoreUint32(&page.views, n)
 7| }
 8|
 9| func (page *Page) Views() uint32 {
10|    return atomic.LoadUint32(&page.views)
11| }
```

下面这个例子使用了Go 1.19引入的类型和方法：

```
 1| type Page struct {
 2|    views atomic.Uint32
 3| }
 4|
 5| func (page *Page) SetViews(n uint32) {
 6|    page.views.Store(n)
 7| }
 8|
 9| func (page *Page) Views() uint32 {
10|    return page.views.Load()
11| }
```

如果T是一个有符号整数类型，比如int32或int64，则AddT函数调用的第二个实参可以是一个负数，用来实现原子减法操作。 但是如果T是一个无符号整数类型，比如uint32、uint64或者uintptr，则AddT函数调用的第二个实参需要为一个非负数，那么如何实现无符号整数类型T值的原子减法操作呢？ 毕竟sync/atomic标准库包没有提供SubstractT函数。 根据欲传递的第二个实参的特点，我们可以把T为一个无符号整数类型的情况细分为两类：

1. 第二个实参为类型为T的一个变量值v。 因为-v在Go中是合法的，所以-v可以直接被用做AddT调用的第二个实参。
2. 第二个实参为一个正整数常量c，这时-c在Go中是编译不通过的，所以它不能被用做AddT调用的第二个实参。 这时我们可以使用^T(c-1)（仍为一个正数）做为AddT调用的第二个实参。

此^T(v-1)小技巧对于无符号类型的变量v也是适用的，但是^T(v-1)比T(-v)的效率要低。

对于这个^T(c-1)小技巧，如果c是一个类型确定值并且它的类型确实就是T，则它的表示形式可以简化为^(c-1)。

一个例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync/atomic"
 6| )
 7|
 8| func main() {
 9|    var (
10|       n uint64 = 97
11|       m uint64 = 1
12|       k int    = 2
13|    )
14|    const (
15|       a        = 3
16|       b uint64 = 4
17|       c uint32 = 5
18|       d int    = 6
19|    )
20|
21|    show := fmt.Println
22|    atomic.AddUint64(&n, -m)
23|    show(n) // 96 (97 - 1)
24|    atomic.AddUint64(&n, -uint64(k))
25|    show(n) // 94 (96 - 2)
26|    atomic.AddUint64(&n, ^uint64(a - 1))
27|    show(n) // 91 (94 - 3)
28|    atomic.AddUint64(&n, ^(b - 1))
29|    show(n) // 87 (91 - 4)
30|    atomic.AddUint64(&n, ^uint64(c - 1))
```

```
31|    show(n) // 82 (87 - 5)
32|    atomic.AddUint64(&n, ^uint64(d - 1))
33|    show(n) // 76 (82 - 6)
34|    x := b; atomic.AddUint64(&n, -x)
35|    show(n) // 72 (76 - 4)
36|    atomic.AddUint64(&n, ^(m - 1))
37|    show(n) // 71 (72 - 1)
38|    atomic.AddUint64(&n, ^uint64(k - 1))
39|    show(n) // 69 (71 - 2)
40| }
```

SwapT函数调用和StoreT函数调用类似，但是返回修改之前的旧值（因此称为置换操作）。

一个CompareAndSwapT函数调用传递的旧值和目标值的当前值匹配的情况下才会将目标值改为新值，并返回true；否则立即返回false。

一个例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync/atomic"
 6| )
 7|
 8| func main() {
 9|    var n int64 = 123
10|    var old = atomic.SwapInt64(&n, 789)
11|    fmt.Println(n, old) // 789 123
12|    swapped := atomic.CompareAndSwapInt64(&n, 123, 456)
13|    fmt.Println(swapped) // false
14|    fmt.Println(n)       // 789
15|    swapped = atomic.CompareAndSwapInt64(&n, 789, 456)
16|    fmt.Println(swapped) // true
17|    fmt.Println(n)       // 456
18| }
```

下面是与之对应的使用Go 1.19引入的类型和方法的实现：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync/atomic"
```

```
 6| )
 7|
 8| func main() {
 9|    var n atomic.Int64
10|    n.Store(123)
11|    var old = n.Swap(789)
12|    fmt.Println(n.Load(), old) // 789 123
13|    swapped := n.CompareAndSwap(123, 456)
14|    fmt.Println(swapped)  // false
15|    fmt.Println(n.Load()) // 789
16|    swapped = n.CompareAndSwap(789, 456)
17|    fmt.Println(swapped)  // true
18|    fmt.Println(n.Load()) // 456
19| }
```

请注意，到目前为止（Go 1.23），一个64位字（int64或uint64值）的原子操作要求此64位字的内存地址必须是8字节对齐的。 对于Go 1.19引入的原子方法操作，此要求无论在32-bit还是64-bit架构上总是会得到满足，但是对于32-bit架构上的原子函数操作，此要求并非总会得到满足。 请阅读关于Go值的内存布局（第44章）一文获取详情。

指针值的原子操作

上面已经提到了sync/atomic标准库包为指针值的原子操作提供了四个函数，并且指针值的原子操作是通过非类型安全指针来实现的。

从非类型安全指针（第25章）一文，我们得知，在Go中， 任何指针类型的值可以被显式转换为非类型安全指针类型unsafe.Pointer，反之亦然。 所以指针类型*unsafe.Pointer的值也可以被显式转换为类型unsafe.Pointer，反之亦然。

下面这个程序不是一个并发程序。它仅仅展示了如何使用指针原子操作。在这个例子中，类型T可以为任何类型。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync/atomic"
 6|    "unsafe"
 7| )
 8|
 9| type T struct {x int}
10|
11| func main() {
```

```
12|    var pT *T
13|    var unsafePPT = (*unsafe.Pointer)(unsafe.Pointer(&pT))
14|    var ta, tb = T{1}, T{2}
15|    // 修改
16|    atomic.StorePointer(
17|       unsafePPT, unsafe.Pointer(&ta))
18|    fmt.Println(pT) // &{1}
19|    // 读取
20|    pa1 := (*T)(atomic.LoadPointer(unsafePPT))
21|    fmt.Println(pa1 == &ta) // true
22|    // 置换
23|    pa2 := atomic.SwapPointer(
24|       unsafePPT, unsafe.Pointer(&tb))
25|    fmt.Println((*T)(pa2) == &ta) // true
26|    fmt.Println(pT) // &{2}
27|    // 比较置换
28|    b := atomic.CompareAndSwapPointer(
29|       unsafePPT, pa2, unsafe.Pointer(&tb))
30|    fmt.Println(b) // false
31|    b = atomic.CompareAndSwapPointer(
32|       unsafePPT, unsafe.Pointer(&tb), pa2)
33|    fmt.Println(b) // true
34| }
```

是的，目前指针的原子操作使用起来是相当的啰嗦。 事实上，啰嗦还是次要的，更主要的是，因为指针的原子操作需要引入unsafe标准库包，所以这些操作函数不在Go 1兼容性保证（https://golang.google.cn/doc/golcompat）之列。

与之相对，如果我们使用Go 1.19引入的Pointer泛型类型和它的方法来做指针原子操作，代码将变得简洁的多。 下面的代码证明了这一点。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync/atomic"
 6| )
 7|
 8| type T struct {x int}
 9|
10| func main() {
11|    var pT atomic.Pointer[T]
12|    var ta, tb = T{1}, T{2}
13|    // store
14|    pT.Store(&ta)
```

```
15|    fmt.Println(pT.Load()) // &{1}
16|    // load
17|    pa1 := pT.Load()
18|    fmt.Println(pa1 == &ta) // true
19|    // swap
20|    pa2 := pT.Swap(&tb)
21|    fmt.Println(pa2 == &ta) // true
22|    fmt.Println(pT.Load())  // &{2}
23|    // compare and swap
24|    b := pT.CompareAndSwap(&ta, &tb)
25|    fmt.Println(b) // false
26|    b = pT.CompareAndSwap(&tb, &ta)
27|    fmt.Println(b) // true
28| }
```

更为重要的是，上面这段代码没有引入unsafe标准库包，所以Go 1会保证它的向后兼容性。

任何类型值的原子操作

sync/atomic标准库包中提供的Value类型可以用来读取和修改任何类型的值。

类型*Value有几个方法：Load、Store、Swap和CompareAndSwap（其中后两个方法实在Go 1.17中引入的）。 这些方法均以interface{}做为参数类型，所以传递给它们的实参可以是任何类型的值。 但是对于一个可寻址的Value类型的值v，一旦v.Store方法（(&v).Store的简写形式）被曾经调用一次，则传递给值v的后续方法调用的实参的具体类型必须和传递给它的第一次调用的实参的具体类型一致； 否则，将产生一个恐慌。nil接口类型实参也将导致v.Store()方法调用产生恐慌。

一个例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync/atomic"
 6| )
 7|
 8| func main() {
 9|    type T struct {a, b, c int}
10|    var ta = T{1, 2, 3}
11|    var v atomic.Value
```

```
12|    v.Store(ta)
13|    var tb = v.Load().(T)
14|    fmt.Println(tb)       // {1 2 3}
15|    fmt.Println(ta == tb) // true
16|
17|    v.Store("hello") // 将导致一个恐慌
18| }
```

另一个例子（针对Go 1.17+）：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "sync/atomic"
 6| )
 7|
 8| func main() {
 9|    type T struct {a, b, c int}
10|    var x = T{1, 2, 3}
11|    var y = T{4, 5, 6}
12|    var z = T{7, 8, 9}
13|    var v atomic.Value
14|    v.Store(x)
15|    fmt.Println(v) // {{1 2 3}}
16|    old := v.Swap(y)
17|    fmt.Println(v)       // {{4 5 6}}
18|    fmt.Println(old.(T)) // {1 2 3}
19|    swapped := v.CompareAndSwap(x, z)
20|    fmt.Println(swapped, v) // false {{4 5 6}}
21|    swapped = v.CompareAndSwap(y, z)
22|    fmt.Println(swapped, v) // true {{7 8 9}}
23| }
```

事实上，我们也可以使用上一节介绍的指针原子操作来对任何类型的值进行原子读取和修改，不过需要多一级指针的间接引用。 两种方法有各自的好处和缺点。在实践中需要根据具体需要选择合适的方法。

原子操作相关的内存顺序保证

详见Go中的内存顺序保证（第41章）一文对此情况的说明。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

Go中的内存顺序保证

关于内存顺序

很多编译器优化（在编译时刻）和CPU处理器优化（在运行时刻）会常常调整指令执行顺序，从而使得指令执行顺序和代码中指定的顺序不太一致。 指令顺序也称为内存顺序（https://en.wikipedia.org/wiki/Memory_ordering）。

当然，指令执行顺序的调整规则不是任意的。 最基本的要求是发生在一个不与其它协程共享数据的协程内部的指令执行顺序调整在此协程内部必须不能被觉察到。 换句话说，从这样的一个协程的角度看，其中的指令执行顺序和代码中指定的顺序总是一致的，即使确实有一些指令的执行顺序发生了调整。

然而，如果一些协程之间共享数据，那么在其中一个协程中发生的指令执行顺序调整将有可能被剩余的其它协程觉察到，从而影响到所有这些协程的行为。 在并发编程中，多个协程之间共享数据是家常便饭。 如果我们忽视了指令执行顺序调整带来的影响，我们编写的并发程序的行为将依赖于特定编译器和CPU。这样的程序常常表现出异常行为。

下面是一个编写得非常不职业的Go程序。此程序的编写没有考虑指令执行顺序调整带来的影响。 此程序扩展于官方文档Go 1 内存模型（https://golang.google.cn/ref/mem）一文中的一个例子.

```
 1| package main
 2|
 3| import "log"
 4| import "runtime"
 5|
 6| var a string
 7| var done bool
 8|
 9| func setup() {
10|    a = "hello, world"
11|    done = true
12|    if done {
13|       log.Println(len(a)) // 如果被打印出来，它总是12
14|    }
15| }
16|
17| func main() {
18|    go setup()
19|
```

```
20|    for !done {
21|       runtime.Gosched()
22|    }
23|    log.Println(a) // 期待的打印结果：hello, world
24| }
```

此程序的行为很可能正如我们所料，`hello, world`将被打印输出。 然而，此程序的行为并非跨编译器和跨平台架构兼容的。 如果此程序使用一个不同的（但符合Go规范的）编译器或者不同的编译器版本编译，它的运行结果可能是不同的。 即使此程序使用同一个编译器编译，在不同的平台架构上的运行结果也可能是不同的。

编译器和CPU可能调整`setup`函数中的前两条语句的执行顺序，使得`setup`协程中的指令的执行顺序和下面的代码指定的顺序一致。

```
1| func setup() {
2|    done = true
3|    a = "hello, world"
4|    if done {
5|       log.Println(len(a))
6|    }
7| }
```

`setup`协程并不会觉察到此执行顺序调整，所以此协程中的`log.Println(len(a))`语句将总是打印出`12`（如果此打印语句在程序退出之前得到了执行的话）。 但是，此执行顺序调整将被主协程觉察到，所以最终的打印结果有可能是空，而不是`hello, world`。

除了没有考虑指令执行顺序调整带来的影响，此程序还存在数据竞争的问题。 变量`a`和`done`的使用没有进行同步。 所以此程序是一个充满了各种并发编程错误的不良例子。 一个职业的Go程序员不应该写出这样的使用于生产环境中的代码。

我们可以使用Go官方工具链中的`go build -race`命令来编译并运行一个程序，以检查此程序中是否存在着数据竞争。

Go内存模型（Memory Model）

有时，为了程序的逻辑正确性，我们需要确保一个协程中的一些语句一定要在另一个协程的某些语句之后（或者之前）执行（从这两个协程的角度观察都是如此）。 指令执行顺序调整可能会给此需求带来一些麻烦。 我们应如何防止某些可能发生的指令执行顺序调整呢?

不同的CPU架构提供了不同的栅栏（fence）指令来防止各种指令执行顺序调整。 一些编程语言提供对应的函数来在代码中的合适位置插入各种栅栏指令。 但是，理解

和正确地使用这些栅栏指令极大地提高了并发编程的门槛。

Go语言的设计哲学是用尽量少的易于理解的特性来支持尽量多的使用场景，同时还要尽量保证代码的高执行效率。 所以Go内置和标准库并没有提供直接插入各种栅栏指令的途径。 事实上，这些栅栏指令被使用在Go中提供的各种高级数据同步技术的实现中。 所以，我们应该使用Go中提供的高级数据同步技术来保证我们所期待的代码执行顺序。

本文的余下部分将列出Go中保证的各种代码执行顺序。 这些顺序保证可能在[Go 1 内存模型](https://golang.google.cn/ref/mem)（https://golang.google.cn/ref/mem）一文提到了，也可能没有提到。

在下面的叙述中，如果我们说事件A保证发生在事件B之前，这意味着这两个事件涉及到的任何协程都将观察到在事件A之前的语句肯定将在事件B之后的语句先执行。对于不相关的协程，它们所观察到的顺序可能并非如此所述。

一个协程的创建发生在此协程中的任何代码执行之前

在下面这个函数中，对x和y的赋值保证发生在对它们的打印之前，并且对x的打印肯定发生在对y的打印之前。

```
 1| var x, y int
 2| func f1() {
 3|     x, y = 123, 789
 4|     go func() {
 5|         fmt.Println(x)
 6|         go func() {
 7|             fmt.Println(y)
 8|         }()
 9|     }()
10| }
```

然而这些顺序在下面这个函数中是得不到任何保证的。此函数存在着数据竞争。

```
 1| var x, y int
 2| func f2() {
 3|     go func() {
 4|         fmt.Println(x) // 可能打印出0、123，或其它值
 5|     }()
 6|     go func() {
 7|         fmt.Println(y) // 可能打印出0、789，或其它值
 8|     }()
 9|     x, y = 123, 789
10| }
```

通道操作相关的顺序保证

下面列出的是通道操作做出的基本顺序保证：

1. 一个通道上的第*n*次成功发送操作的开始发生在此通道上的第*n*次成功接收操作完成之前，无论此通道是缓冲的还是非缓冲的。
2. 一个容量为*m*通道上的第*n*次成功接收操作的开始发生在此通道上的第*n*+*m*次发送操作完成之前。 特别地，如果此通道是非缓冲的（`m == 0`），则此通道上的第*n*次成功接收操作的开始发生在此通道上的第*n*次发送操作完成之前。
3. 一个通道的关闭操作发生在任何因为此通道被关闭而从此通道接收到了零值的操作完成之前。

事实上， 对一个非缓冲通道来说，其上的第*n*次成功发送的完成的发送和其上的第*n*次成功接收的完成应被视为同一事件。

下面这段代码展示了一个非缓冲通道上的发送和接收操作是如何保证特定的代码执行顺序的。

```
 1| func f3() {
 2|    var a, b int
 3|    var c = make(chan bool)
 4|
 5|    go func() {
 6|       a = 1
 7|       c <- true
 8|       if b != 1 {
 9|          panic("b != 1") // 绝不可能发生
10|       }
11|    }()
12|
13|    go func() {
14|       b = 1
15|       <-c
16|       if a != 1  {
17|          panic("a != 1") // 绝不可能发生
18|       }
19|    }()
20| }
```

对于函数`f3`中创建的两个协程，下列顺序将得到保证：

- 赋值语句`b = 1`肯定在条件`b != 1`被估值之前执行完毕。
- 赋值语句`a = 1`肯定在条件`a != 1`被估值之前执行完毕。

所以，上例代码中两个协程中的panic调用将永不可能得到执行。 做为对比，下面这段代码中的panic调用有可能会得到执行，因为上述通道操作相关的顺序保证对于不相关的协程是无效的。

```
 1| func f4() {
 2|    var a, b, x, y int
 3|    c := make(chan bool)
 4|
 5|    go func() {
 6|       a = 1
 7|       c <- true
 8|       x = 1
 9|    }()
10|
11|    go func() {
12|       b = 1
13|       <-c
14|       y = 1
15|    }()
16|
17|    // 一个和上面的通道操作不相关的协程。
18|    // 这是一个不良代码的例子，它造成了很多数据竞争。
19|    go func() {
20|       if x == 1 {
21|          if a != 1 {
22|             panic("a != 1") // 有可能发生
23|          }
24|          if b != 1 {
25|             panic("b != 1") // 有可能发生
26|          }
27|       }
28|
29|       if y == 1 {
30|          if a != 1 {
31|             panic("a != 1") // 有可能发生
32|          }
33|          if b != 1 {
34|             panic("b != 1") // 有可能发生
35|          }
36|       }
37|    }()
38| }
```

这里的新创建的第三个协程是一个和通道c上的发送和接收操作不相关的一个协程。 它所观察到的执行顺序和其它两个新创建的协程可能是不同的。 条件a != 1

和b != 1的估值有可能为true，所以四个panic调用有可能会得到执行。

事实上，大多数编译器的实现确实很可能能够保证上面这个不良的例子中的四个panic调用永远不可能被执行，但是，没有任何Go官方文档做出了这样的保证。 此不良例子的执行结果是依赖于不同的编译器和不同的编译器版本的。 我们编写的Go代码应该以Go官方文档中明确记录下来的规则为依据。

下面是一个缓冲通道的例子。

```
 1| func f5() {
 2|    var k, l, m, n, x, y int
 3|    c := make(chan bool, 2)
 4|
 5|    go func() {
 6|       k = 1
 7|       c <- true
 8|       l = 1
 9|       c <- true
10|       m = 1
11|       c <- true
12|       n = 1
13|    }()
14|
15|    go func() {
16|       x = 1
17|       <-c
18|       y = 1
19|    }()
20| }
```

在此例子中，下面的顺序得以保证：

- 赋值语句k = 1的执行保证在赋值语句y = 1的执行之前结束。
- 赋值语句x = 1的执行保证在赋值语句n = 1的执行之前结束。

然而，赋值语句x = 1的执行并不能保证在赋值语句l = 1和m = 1的执行之前结束， 赋值语句l = 1和m = 1的执行也不能保证在赋值语句y = 1的执行之前结束。

下面是一个通道关闭的例子。在这个例子中，赋值语句k = 1的执行保证在赋值语句y = 1执行之前结束，但不能保证在赋值语句x = 1执行之前结束。

```
 1| func f6() {
 2|    var k, x, y int
 3|    c := make(chan bool, 1)
```

```
 4|
 5|    go func() {
 6|       c <- true
 7|       k = 1
 8|       close(c)
 9|    }()
10|
11|    go func() {
12|       <-c
13|       x = 1
14|       <-c
15|       y = 1
16|    }()
17| }
```

互斥锁相关的顺序保证

Go中和互斥锁（第39章）相关的顺序保证：

1. 对于一个可寻址的`sync.Mutex`类型或者`sync.RWMutex`类型的值`m`，第*n*次成功的`m.Unlock()`方法调用保证发生在第*n+1*次`m.Lock()`方法调用返回之前。
2. 对一个可寻址的`RWMutex`类型值`rw`，如果它的第*n*次`rw.Lock()`方法调用已成功返回，并且有一个`rw.RLock()`方法调用保证发生在此第*n*次`rw.Lock()`方法调用返回之后，则第*n*次成功的`rw.Unlock()`方法调用保证发生在此`rw.RLock()`方法调用返回之前。
3. 对一个可寻址的`RWMutex`类型值`rw`，如果它的第*n*次`rw.RLock()`方法调用已成功返回，并且有一个`rw.Lock()`方法调用保证发生在此第*n*次`rw.RLock()`方法调用返回之后，则第*m*次成功的`rw.RUnlock()`方法调用（其中`m <= n`）保证发生在此`rw.Lock()`方法调用返回之前。

在下面这个例子中，下列顺序肯定得到保证。

- 赋值语句`a = 1`的执行保证在赋值语句`b = 1`的执行之前结束。
- 赋值语句`m = 1`的执行保证在赋值语句`n = 1`的执行之前结束。
- 赋值语句`x = 1`的执行保证在赋值语句`y = 1`的执行之前结束。

```
 1| func fab() {
 2|    var a, b int
 3|    var l sync.Mutex // or sync.RWMutex
 4|
 5|    l.Lock()
 6|    go func() {
```

```
 7|       l.Lock()
 8|       b = 1
 9|       l.Unlock()
10|    }()
11|    go func() {
12|       a = 1
13|       l.Unlock()
14|    }()
15| }
16|
17| func fmn() {
18|    var m, n int
19|    var l sync.RWMutex
20|
21|    l.RLock()
22|    go func() {
23|       l.Lock()
24|       n = 1
25|       l.Unlock()
26|    }()
27|    go func() {
28|       m = 1
29|       l.RUnlock()
30|    }()
31| }
32|
33| func fxy() {
34|    var x, y int
35|    var l sync.RWMutex
36|
37|    l.Lock()
38|    go func() {
39|       l.RLock()
40|       y = 1
41|       l.RUnlock()
42|    }()
43|    go func() {
44|       x = 1
45|       l.Unlock()
46|    }()
47| }
```

注意，在下面这段代码中，根据Go官方文档，赋值语句`p = 1`的执行并不能保证在赋值语句`q = 1`的执行之前结束，尽管多数编译器确实能够做出这样的保证。

```
1| var p, q int
2| func fpq() {
3|    var l sync.Mutex
4|    p = 1
5|    l.Lock()
6|    l.Unlock()
7|    q = 1
8| }
```

sync.WaitGroup值做出的顺序保证

假设在某个给定时刻，一个可寻址的sync.WaitGroup值wg维护的计数不为0，并且有一个wg.Wait()方法调用在此给定时刻之后调用。 如果有一组wg.Add(n)方法调用在此给定时刻之后调用，并且我们可以保证这组调用中只有最后一个返回的调用会将wg维护的计数修改为0， 则这组调用中的每个调用保证都发生在此wg.Wait()方法调用返回之前。

注意：调用wg.Done()和wg.Add(-1)是等价的。

请阅读对sync.WaitGroup类型的解释（第39章）来获取如何使用sync.WaitGroup值。

sync.Once值做出的顺序保证

请阅读对sync.Once类型的解释（第39章）来获取sync.Once值做出的顺序保证和如何使用sync.Once值。

sync.Cond值做出的顺序保证

sync.Cond值出的顺序保证有些难以表达清楚。所以这里就只可意会不可言传了。请阅读对sync.Cond类型的解释（第39章）来获取如何使用sync.Cond值。

原子操作相关的顺序保证

从Go 1.19开始，Go 1 内存模型正式地说明Go程序中执行的原子操作按照顺序一致次序（sequentially consistent order）执行。 如果一个原子（写）操作A的效果被一个原子（读）操作B观察到，则A肯定被同步到B之前执行。

按照这个说法，在下面这个程序中，对变量b的原子写操作肯定发生在对其读取结果为1的原子原子读操作之前。 从而使得对变量a的普通写操作也发生于对其的普通读操作之前。 所以此程序保证会打印出2。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "runtime"
 6|    "sync/atomic"
 7| )
 8|
 9| func main() {
10|    var a, b int32 = 0, 0
11|
12|    go func() {
13|       a = 2
14|       atomic.StoreInt32(&b, 1)
15|    }()
16|
17|    for {
18|       if n := atomic.LoadInt32(&b); n == 1 {
19|          fmt.Println(a) // 2
20|          break
21|       }
22|       runtime.Gosched()
23|    }
24| }
```

请阅读原子操作（第40章）一文来获取如何使用原子操作。

和终结器相关的顺序保证

调用`runtime.SetFinalizer(x, f)`发生在终结调用`f(x)`被执行之前。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

一些常见并发编程错误

Go语言是一门天然支持并发的编程语言。 通过使用go关键字，我们可以很轻松地创建协程；通过使用（第37章）通道（第21章）和Go中提供的（第40章）其它各种同步技术（第39章），并发编程变得简单、轻松和有趣。

另一方面，Go并不阻止程序员在并发编程中因为粗心或者经验不足而犯错。 本文的余下部分将展示一些常见的并发错误，来帮助Go程序员在实践中避免这些错误。

当需要同步的时候没有同步

我们已经知道，源文件中的代码行在运行时刻并非总是按照它们的出现次序被执行（第41章）。

下面这个示例程序犯了两个错误：

- 首先，主协程中对变量b的读取和匿名协程中的对变量b的写入可能会产生数据竞争；
- 其次，在主协程中，条件b == true成立并不能确保条件a != nil也成立。编译器和CPU可能会对调整此程序中匿名协程中的某些指令的顺序（第41章）已获取更快的执行速度。 所以，站在主协程的视角看，对变量b的赋值可能会发生在对变量a的赋值之前，这将造成在修改a的元素时a依然为一个nil切片。

```
 1| package main
 2|
 3| import (
 4|     "time"
 5|     "runtime"
 6| )
 7|
 8| func main() {
 9|     var a []int // nil
10|     var b bool  // false
11|
12|     // 一个匿名协程。
13|     go func () {
14|         a = make([]int, 3)
15|         b = true // 写入b
16|     }()
17|
```

```
18|    for !b { // 读取b
19|       time.Sleep(time.Second)
20|       runtime.Gosched()
21|    }
22|    a[0], a[1], a[2] = 0, 1, 2 // 可能会发生恐慌
23| }
```

上面这个程序可能在很多计算机上运行良好，但是可能会在某些计算机上因为恐慌而崩溃退出；或者使用某些编译器编译的时候运行良好，但使用另外的某个编译器编译的时候将造成程序运行时崩溃退出。

我们应该使用通道或者sync标准库包中的同步技术来确保内存顺序。比如：

```
 1| package main
 2|
 3| func main() {
 4|    var a []int = nil
 5|    c := make(chan struct{})
 6|
 7|    go func () {
 8|       a = make([]int, 3)
 9|       c <- struct{}{}
10|    }()
11|
12|    <-c
13|    a[0], a[1], a[2] = 0, 1, 2 // 绝不会造成恐慌
14| }
```

使用time.Sleep调用来做同步

让我们看一个简单的例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6| )
 7|
 8| func main() {
 9|    var x = 123
10|
11|    go func() {
12|       x = 789 // 写入x
```

```
13|    }()
14|
15|    time.Sleep(time.Second)
16|    fmt.Println(x) // 读取x
17| }
```

我们期望着此程序打印出789。 事实上，则其运行结果常常正如我们所期待的。 但是，此程序中的同步处理实现的正确吗？否！原因很简单，Go运行时并不能保证对x的写入一定发生在对x的读取之前。 在某些特定的情形下，比如CPU资源被很一些其它计算密集的程序所占用，则对x的写入有可能发生在对x的读取之后。 因此，我们不应该在正式的项目中使用time.Sleep调用来做同步。

让我们看另一个简单的例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "time"
 6| )
 7|
 8| func main() {
 9|    var n = 123
10|
11|    c := make(chan int)
12|
13|    go func() {
14|       c <- n + 0
15|    }()
16|
17|    time.Sleep(time.Second)
18|    n = 789
19|    fmt.Println(<-c)
20| }
```

你觉得此程序会输出什么？123还是789？ 事实上，它的输出是和具体使用的编译器相关的。 对于标准编译器1.23版本来说，它很可能输出123。 但是从理论上说，它也可能输出789。

让我们将此例中的c <- n + 0一行换成c <- n，然后重新运行它，你将会发现它的输出变成了789（如果它使用标准编译器1.23版本编译的话）。 重申一次，此结果是和具体使用的编译器和编译器的版本相关的。

是的，此程序中存在数据竞争。表达式n的估值可能发生在赋值语句n = 789执行之前、之后、或者期间。 time.Sleep调用并不能保证n的估值发生在此赋值之后。

对于这个特定的例子，我们应该将欲发送的值在开启新协程之前存储在一个临时变量中来避免数据竞争。

```
1| ...
2|    tmp := *p
3|    go func() {
4|       c <- tmp
5|    }()
6| ...
```

使一些协程永久处于阻塞状态

有很多原因导致某个协程永久阻塞，比如：

- 从一个永远不会有其它协程向其发送数据的通道接收数据；
- 向一个永远不会有其它协程从中读取数据的通道发送数据；
- 被自己死锁了；
- 和其它协程相互死锁了；
- 等等。

除了有时我们故意地将主协程永久阻塞以防止程序退出外，其它大多数造成协程永久阻塞的情况都不是我们所期待的。 Go运行时很难分辨出一个处于阻塞状态的协程是否将永久阻塞下去，所以Go运行时不会释放永久处于阻塞状态的协程占用的资源。

在采用最快回应（第37章）通道用例中，如果被当作future/promise来用的通道的容量不足够大，则较慢回应的协程在准备发送回应结果时将永久阻塞。 比如，下面的例子中，每个请求将导致4个协程永久阻塞。

```
 1| func request() int {
 2|    c := make(chan int)
 3|    for i := 0; i < 5; i++ {
 4|       i := i
 5|       go func() {
 6|          c <- i // 4个协程将永久阻塞在这里
 7|       }()
 8|    }
 9|    return <-c
10| }
```

为了防止有4个协程永久阻塞，被当作future/promise使用的通道的容量必须至少为4。

在第二种“采用最快回应”实现方法（第37章）中，如果被当作future/promise使用的通道是一个非缓冲通道（如下面的代码所示），则有可能导致其通道的接收者可能会错过所有的回应而导致处于永久阻塞状态。

```
 1| func request() int {
 2|    c := make(chan int)
 3|    for i := 0; i < 5; i++ {
 4|       i := i
 5|       go func() {
 6|          select {
 7|          case c <- i:
 8|          default:
 9|          }
10|       }()
11|    }
12|    return <-c // 有可能永久阻塞在此
13| }
```

接收者协程可能会永久阻塞的原因是如果5个尝试发送操作都发生在接收操作<-c准备好之前，亦即5个个尝试发送操作都失败了，则接收者协程将永远无值可接收（从而将处于永久阻塞状态）。

将通道c改为一个缓冲通道，则至少会有一个尝试发送将成功，从而接收者协程肯定不会永久阻塞。

复制sync标准库包中的类型的值

在实践中，sync标准库包中的类型（除了Locker接口类型）的值不应该被复制。我们只应该复制它们的指针值。

下面是一个有问题的并发编程的例子。 在此例子中，当Counter.Value方法被调用时，一个Counter属主值将被复制，此属主值的字段Mutex也将被一同复制。此复制并没有被同步保护，因此复制结果可能是不完整的，并非被复制的属主值的一个快照。 即使此Mutex字段得以侥幸完整复制，它的副本所保护的是对字段n的一个副本的访问，因此一般是没有意义的。

```
 1| import "sync"
 2|
 3| type Counter struct {
 4|    sync.Mutex
```

```
 5|    n int64
 6| }
 7|
 8| // 此方法实现是没问题的。
 9| func (c *Counter) Increase(d int64) (r int64) {
10|    c.Lock()
11|    c.n += d
12|    r = c.n
13|    c.Unlock()
14|    return
15| }
16|
17| // 此方法的实现是有问题的。当它被调用时，
18| // 一个Counter属主值将被复制。
19| func (c Counter) Value() (r int64) {
20|    c.Lock()
21|    r = c.n
22|    c.Unlock()
23|    return
24| }
```

我们应该将`Value`方法的属主参数类型更改为指针类型`*Counter`来避免复制`sync.Mutex`值。

Go官方工具链中提供的`go vet`命令将提示此例中的`Value`方法的声明可能是一个潜在的逻辑错误。

在错误的地方调用sync.WaitGroup.Add方法

每个`sync.WaitGroup`值内部维护着一个计数。此计数的初始值为0。 如果一个`sync.WaitGroup`值的`Wait`方法在此计数为0的时候被调用，则此调用不会阻塞，否则此调用将一直阻塞到此计数变为0为止。

为了让一个`WaitGroup`值的使用有意义，在此值的计数为0的情况下，对它的下一次`Add`方法的调用必须出现在对它的下一次`Wait`方法的调用之前。

比如，在下面的例子中，`Add`方法的调用位置是不合适的。 此例子程序的打印结果并不总是`100`，而可能是`0`到`100`间的任何一个值。 原因是没有任何一个`Add`方法调用可以确保发生在唯一的`Wait`方法调用之前，结果导致没有任何一个`Done`方法调用可以确保发生在唯一的`Wait`方法调用返回之前。

```
 1| package main
 2|
```

```
 3| import (
 4|    "fmt"
 5|    "sync"
 6|    "sync/atomic"
 7| )
 8|
 9| func main() {
10|    var wg sync.WaitGroup
11|    var x int32 = 0
12|    for i := 0; i < 100; i++ {
13|       go func() {
14|          wg.Add(1)
15|          atomic.AddInt32(&x, 1)
16|          wg.Done()
17|       }()
18|    }
19|
20|    fmt.Println("等待片刻...")
21|    wg.Wait()
22|    fmt.Println(atomic.LoadInt32(&x))
23| }
```

我们应该将对Add方法的调用移出匿名协程之外，像下面这样，使得任何一个Done方法调用都确保发生在唯一的Wait方法调用返回之前。

```
 1| ...
 2|    for i := 0; i < 100; i++ {
 3|       wg.Add(1)
 4|       go func() {
 5|          atomic.AddInt32(&x, 1)
 6|          wg.Done()
 7|       }()
 8|    }
 9| ...
```

不当地使用用做Future/Promise的通道

从通道用例大全（第37章）一文中，我们了解到一些函数可以返回用做future/promise的通道结果。 假设fa和fb是这样的两个函数，则下面的调用方式并没有体现出这两个函数的真正价值。

```
 1| doSomethingWithFutureArguments(<-fa(), <-fb())
```

在上面这行调用中，两个实参值（promise回应结果）的生成实际上是串行进行的，future/promise的价值没有体现出来。

我们应该像下面这样调用这两个函数来并发生成两个回应结果：

```
1| ca, cb := fa(), fb()
2| doSomethingWithFutureArguments(<-ca, <-cb)
```

没有让最后一个活跃的发送者关闭通道

Go程序员常犯的一个错误是关闭一个后续可能还会有协程向其发送数据的通道。 当向一个已关闭的通道发送数据的时候，一个恐慌将产生。

这样的错误曾经发生在一些很有名的项目中，比如Kubernetes项目中的这个bug（https://github.com/kubernetes/kubernetes/pull/45291/files?diff=split）和这个bug（https://github.com/kubernetes/kubernetes/pull/39479/files?diff=split）。

请阅读此篇文章（第38章）来了解如何安全和优雅地关闭通道。

对地址不保证为8字节对齐的值执行64位原子操作

64位非方法原子操作中涉及到的实参地址必须为8字节对齐的。不满足此条件的64位原子操作将造成一个恐慌。 对于标准编译器，这样的情形只可能发生在32位的架构中（https://golang.google.cn/pkg/sync/atomic/#pkg-note-BUG）。 从Go 1.19版本开始，我们可以使用64位方法原子操作来避免这一缺陷。 请阅读内存布局一文（第44章）来获知如何确保让64位的整数值的地址在32位的架构中8字节对齐。

没留意过多的`time.After`函数调用消耗了大量资源

`time`标准库包中的`After`函数返回一个用做延迟通知的通道（第37章）。 此函数给并发编程带来了很多便利，但是它的每个调用都需要创建一个`time.Timer`值，在Go 1.23之前，此新创建的`Timer`值在传递给`After`函数调用的时长（实参）内肯定不会被垃圾回收。 如果此函数在某个时段内被多次频繁调用，则可能导致积累很多尚未过期的`Timer`值从而造成大量的内存和计算消耗。

比如在下面这个例子中，（在Go 1.23之前），如果`longRunning`函数被调用并且在一分钟内有一百万条消息到达， 那么在某个特定的很小时间段（大概若干秒）内将存在一百万个活跃的`Timer`值，即使其中只有一个是真正有用的。

```
 1| import (
 2|    "fmt"
 3|    "time"
 4| )
 5|
 6| // 如果某两个连续的消息的间隔大于一分钟，此函数将返回。
 7| func longRunning(messages <-chan string) {
 8|    for {
 9|       select {
10|       case <-time.After(time.Minute):
11|          return
12|       case msg := <-messages:
13|          fmt.Println(msg)
14|       }
15|    }
16| }
```

注意：从Go 1.23开始，这个问题不再存在。从Go 1.23开始的Go运行时做了改进，从而使得当一个`Timer`对象不再被使用时将立即变得可以垃圾回收。

在Go 1.23之前，为了避免太多的`Timer`值被创建，我们应该只使用（并复用）一个`Timer`值，像下面这样：

```
 1| func longRunning(messages <-chan string) {
 2|    timer := time.NewTimer(time.Minute)
 3|    defer timer.Stop()
 4|
 5|    for {
 6|       select {
 7|       case <-timer.C: // 过期了
 8|          return
 9|       case msg := <-messages:
10|          fmt.Println(msg)
11|
12|          // 此if代码块很重要。
13|          if !timer.Stop() {
14|             <-timer.C
15|          }
16|       }
17|
18|       // 必须重置以复用。
19|       timer.Reset(time.Minute)
20|    }
21| }
```

注意，此示例中的if代码块用来舍弃一个潜在的在执行第二个分支代码块的时候发送过来的超时通知（从Go 1.23开始，此举已经不再必要）。

注意：从Go 1.23开始，Timer.Reset方法将会自动舍弃潜在的超时通知。 因此，上面的代码可以被简化为：

```
 1| func longRunning(messages <-chan string) {
 2|    timer := time.NewTimer(time.Minute)
 3|    // defer timer.Stop() // 从Go 1.23开始，可要可不要
 4|
 5|    for {
 6|       select {
 7|       case <-timer.C:
 8|          return
 9|       case msg := <-messages:
10|          fmt.Println(msg)
11|       }
12|
13|       timer.Reset(time.Minute)
14|    }
15| }
```

不正确地使用time.Timer值

（注意：本节提到的问题从Go 1.23开始不再存在。）

一个典型的time.Timer的使用已经在上一节中展示了。一些解释：

- 如果一个Timer值已经过期或者已经被终止（stopped），则相应的Stop方法调用返回false。 在此Timer值尚未终止的时候，Stop方法调用返回false只能意味着此Timer值已经过期。
- 在Go 1.23之前，一个Timer值被终止之后，它的通道字段C最多只能含有一个过期的通知。
- 在Go 1.23之前，在一个Timer终止（stopped）之后并且在重置和重用此Timer值之前，我们应该确保此Timer值中肯定不存在过期的通知。 这就是上一节中的例子中的if代码块的意义所在。

一个*Timer值的Reset方法必须在对应Timer值过期或者终止之后才能被调用；否则，此Reset方法调用和一个可能的向此Timer值的C通道字段的发送通知操作产生数据竞争。

如果上一节中的例子中的select流程控制代码块中的第一个分支被选中，则这表示相应的Timer值已经过期，所以我们不必终止它。 但是我们必须在第二个分支中通过终止此Timer以检查此Timer中是否存在一个过期的通知。 如果确实有一个过期的通知，我们必须在重用这个Timer之前将此过期的通知取出；否则，此过期的通知将下一个循环步导致在第一个分支立即被选中（Go 1.23之前）。

比如，下面这个程序将在运行后大概一秒钟（而不是十秒钟）后退出。 而且此程序存在着潜在的数据竞争。

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "time"
 6| )
 7|
 8| func main() {
 9|     start := time.Now()
10|     timer := time.NewTimer(time.Second/2)
11|     select {
12|     case <-timer.C:
13|     default:
14|         time.Sleep(time.Second) // 此分支被选中的可能性较大
15|     }
16|     timer.Reset(time.Second * 10) // 可能数据竞争
17|     <-timer.C
18|     fmt.Println(time.Since(start)) // 大约1s
19| }
```

当一个time.Timer值不再被使用后，我们不必（但是推荐）终止之。

在多个协程中使用同一个time.Timer值比较容易写出不当的并发代码，所以尽量不要跨协程使用一个Timer值。

我们不应该依赖于time.Timer的Reset方法的返回值。此返回值只要是为了历史兼容性而存在的。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

内存块

Go是一门支持自动内存管理的语言，比如自动内存开辟和自动垃圾回收。 所以Go程序员在编程时无须进行各种纷繁的内存管理操作。 这不仅给Go程序员提供了很多便利和节省了很多开发时间，而且也帮助Go程序员避免了很多因为疏忽大意而造成的bug。

在Go编程中，尽管我们无须知道底层的自动内存管理是如何实现的，但是知道自动内存管理实现中的一些概念和事实对我们写出高质量的Go代码是非常有帮助的。

本文将解释和列出标准编译器和运行时中内存块开辟和垃圾回收实现相关的一些概念和事实。 内存管理的其它方面，比如内存申请和内存释放，将不会在本文中探讨。

内存块（memory block）

一个内存块是一段在运行时刻承载着若干值部（第17章）的连续内存片段。 不同的内存块的大小可能不同，因它们所承载的值部的尺寸而定。 一个内存块同时可能承载着不同Go值的若干值部，但是一个值部在内存中绝不会跨内存块存储，无论此值部的尺寸有多大。

一个内存块可能承载若干值部的原因有很多，这里仅列出一部分：

- 一个结构体值很可能有若干字段，所以当为此结构体值开辟了一个内存块时，此内存块同时也将承载此结构体值的各个字段值（的直接部分）。
- 一个数组值常常包含很多元素，所以当为此数组值开辟了一个内存块时，此内存块同时也将承载此数组值的各个元素值（的直接部分）。
- 两个切片的底层间接部分的元素序列可能承载在同一个内存块上，这两个间接值部甚至可能有部分重叠。

一个值引用着承载着它的值部的内存块

我们已经知道，一个值部可能引用着另一个值部。这里，我们将引用的定义扩展一下。 我们可以说一个内存块被它承载着的各个值部所引用着。 所以，当一个值部v被另一个值部引用着时，此另一个值部也（间接地）引用着承载着值部v的内存块。

什么时候需要开辟内存块?

在Go中，在下列场合（不限于）将发生开辟内存块的操作：

- 显式地调用new和make内置函数。 一个new函数调用总是只开辟一个内存块。 一个make函数调用有可能会开辟多个内存块来承载创建的切片/映射/通道值的直接和底层间接值部。
- 使用字面量创建映射、切片或函数值。在此创建过程中，一个或多个内存块将被开辟出来。
- 声明变量。
- 将一个非接口值赋给一个接口值。（对于标准编译器来说，不包括将一个指针值赋给一个接口值的情况。）
- 衔接非常量字符串。
- 将字符串转换为字节切片或者码点切片，或者反之，除了一些编译器优化情形（第19章）。
- 将一个整数转换为字符串。
- 调用内置append函数并且基础切片的容量不足够大。
- 向一个映射添加一个键值条目并且此映射底层内部的哈希表需要改变容量。

内存块将被开辟在何处?

对每一个使用标准编译器编译的Go程序，在运行时刻，每一个协程将维护一个栈（stack）。 一个栈是一个预申请的内存段，它做为一个内存池供某些内存块从中开辟。 在Go官方工具链1.19版本之前，一个栈的初始尺寸总是2KiB。 从1.19版本开始，栈的初始尺寸是自适应的（https://docs.google.com/document/d/1YD1GIdVTPnmUiTAav1ZxBI1d9pwGQgZT7IKFK1IXohQ）。 每个栈的尺寸在协程运行的时候将按照需要增长和收缩。 栈的最小尺寸为2KiB。

（注意：Go运行时维护着一个协程栈的最大尺寸限制，此限制为全局的。 如果一个协程在增长它的栈的时候超过了此限制，整个程序将崩溃。 对于目前的官方标准Go官方工具链1.23版本，此最大限制的默认值在64位系统上为1GB，在32位系统上为250MB。 我们可以在运行时刻调用`runtime/debug`标准库包中的`SetMaxStack`来修改此值。 另外请注意，当前的官方标准编译器实现中，实际上允许的协程栈的最大尺寸为不超过最大尺寸限制的2的幂。 所以对于默认设置，实际上允许的协程栈的最大尺寸在64位系统上为512MiB，在32位系统上为128MiB。）

内存块可以被开辟在栈上。开辟在一个协程维护的栈上的内存块只能在此协程内部被使用（引用）。 其它协程是无法访问到这些内存块的。 一个协程可以无需使用任何数据同步技术而使用开辟在它的栈上的内存块上的值部。

堆（heap）是一个虚拟的概念。每个程序只有一个堆。 一般地，如果一个内存块没有开辟在任何一个栈上，则我们说它开辟在了堆上。 开辟在堆上的内存块可以被多个协程并发地访问。 在需要的时候，对承载在它们之上的值部的访问需要做同步。

如果编译器觉察到一个内存块在运行时将会被多个协程访问，或者不能轻松地断定此内存块是否只会被一个协程访问，则此内存块将会被开辟在堆上。 也就是说，编译器将采取保守但安全的策略，使得某些可以安全地被开辟在栈上的内存块也有可能会被开辟在堆上。

事实上，栈对于Go程序来说并非必要。Go程序中所有的内存块都可以开辟在堆上。支持栈只是为了让Go程序的运行效率更高。

- 从栈上开辟内存块比在堆上快得多；
- 开辟在栈上的内存块不需要被垃圾回收；
- 开辟在栈上的内存块对CPU缓存更加友好。

如果一个内存块被开辟在某处（堆上或某个栈上），则我们也可以说承载在此内存块上的各个值部也开辟在此处。

如果一个局部声明的变量的某些值部被开辟在堆上，则我们说这些值部以及此局部变量逃逸到了堆上。 我们可以运行Go官方工具链中提供的`go build -gcflags -m`命令来查看代码中哪些局部值的值部在运行时刻会逃逸到堆上。 如上所述，目前官方Go标准编译器中的逃逸分析器并不十分完美，因此某些可以安全地开辟在栈上的值也可能会逃逸到了堆上。

在运行时刻，每一个仍在被使用中的逃逸到堆上的值部肯定被至少一个开辟在栈上的值部所引用着。 如果一个逃逸到堆上的值是一个被声明为`T`类型的局部变量，则在运行时，一个`*T`类型的隐式指针将被创建在栈上。 此指针存储着此`T`类型的局部变量的在堆上的地址，从而形成了一个从栈到堆的引用关系。 另外，编译器还将所有对此局部变量的使用替换为对此指针的解引用。 此`*T`值可能从今后的某一时刻不再被使用从而使得此引用关系不再存在。 此引用关系在下面介绍的垃圾回收过程中发挥着重要的作用。

类似地，我们可以认为每个包级变量（常称全局变量）都被开辟在了堆上，并且它被一个开辟在一个全局内存区上的隐式指针所引用着。 事实上，此指针引用着此包级变量的直接部分，此直接部分又引用着其它的值（部）。

一个开辟在堆上的内存块可能同时被开辟在若干不同栈上的值部所引用着。

一些事实：

- 如果一个结构体值的一个字段逃逸到了堆上，则此整个结构体值也逃逸到了堆上。
- 如果一个数组的某个元素逃逸到了堆上，则此整个数组也逃逸到了堆上。
- 如果一个切片的某个元素逃逸到了堆上，则此切片中的所有元素都将逃逸到堆上，但此切片值的直接部分可能开辟在栈上。
- 如果一个值部`v`被一个逃逸到了堆上的值部所引用，则此值部`v`也将逃逸到堆上。

使用内置new函数开辟的内存可能开辟在堆上，也可能开辟在栈上。这是与C++不同的一点。

当一个协程的栈的大小（因为栈增长或者收缩而）改变时，一个新的内存段将申请给此栈使用。 原先已经开辟在老的内存段上的内存块将很有可能被转移到新的内存段上，或者说这些内存块的地址将改变。 相应地，引用着这些开辟在此栈上的内存块的指针（它们同样开辟在此栈上）中存储的地址也将得到刷新。 下面是一个展示开辟在栈上的值的地址改变的例子。

```
 1| package main
 2|
 3| // 下面这行是为了防止f函数的调用被内联。
 4| //go:noinline
 5| func f(i int) byte {
 6|    var a [1<<20]byte // 使栈增长
 7|    return a[i]
 8| }
 9|
10| func main(){
11|    var x int
12|    println(&x)
13|    f(100)
14|    println(&x)
15| }
```

我们可以看到，上例打引出的两个地址不一样（如果使用官方标准编译器1.23版本编译之）。

一个内存块在什么条件下可以被回收?

为包级变量的直接部分开辟的内存块永远不会被回收。

每个协程的栈将在此协程退出之时被整体回收，此栈上开辟的各个内存块没必要被一个一个单独回收。 栈内存池并不由垃圾回收器回收。

对一个开在堆上的内存块，当它不再被任何开辟在协程栈的仍被使用中的，以及全局内存区上的，值部所（直接或者间接）地引用着，则此内存块可以被安全地垃圾回收了。 我们称这样的内存块为不再被使用的内存块。开辟在堆上的不再被使用的内存块将在以后某个时刻被垃圾回收器回收掉。

下面是一个展示了一些内存块在何时可以被垃圾回收的例子。

```
 1| package main
 2|
```

```
 3| var p *int
 4|
 5| func main() {
 6|    done := make(chan bool)
 7|    // done通道将被使用在主协程和下面将要
 8|    // 创建的新协程中，所以它将被开辟在堆上。
 9|
10|    go func() {
11|       x, y, z := 123, 456, 789
12|       _ = z  // z可以被安全地开辟在栈上。
13|       p = &x // 因为x和y都会将曾经被包级指针p所引用过，
14|       p = &y // 因此，它们都将开辟在堆上。
15|
16|       // 到这里，x已经不再被任何其它值所引用。或者说承载
17|       // 它的内存块已经不再被使用。此内存块可以被回收了。
18|
19|       p = nil
20|       // 到这里，y已经不再被任何其它值所引用。
21|       // 承载它的内存块可以被回收了。
22|
23|       done <- true
24|    }()
25|
26|    <-done
27|    // 到这里，done已经不再被任何其它值所引用。一个
28|    // 聪明的编译器将认为承载它的内存块可以被回收了。
29|
30|    // ...
31| }
```

有时，聪明的编译器可能会做出一些出人意料的（但正确的）的优化。 比如在下面这个例子中，切片bs的底层间接值部在bs仍在使用之前就已经被标准编译器发觉已经不再被使用了。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    // 假设此切片的长度很大，以至于它的元素
 7|    // 将被开辟在堆上。
 8|    bs := make([]byte, 1 << 31)
 9|
10|    // 一个聪明的编译器将觉察到bs的底层元素
11|    // 部分已经不会再被使用，而正确地认为bs的
```

```
12|     // 底层元素部分在此刻可以被安全地回收了。
13|
14|     fmt.Println(len(bs))
15| }
```

关于切片值的内部实现结构，请参考值部（第17章）一文。

顺便说一下，有时候出于种种原因，我们希望确保上例中的bs切片的底层间接值部不要在fmt.Println调用之前被垃圾回收。 这时，我们可以使用一个runtime.KeepAlive函数调用以便让垃圾回收器知晓在此调用之前切片bs和它所引用着的值部仍在被使用中。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "runtime"
 5|
 6| func main() {
 7|     bs := make([]int, 1000000)
 8|
 9|     fmt.Println(len(bs))
10|     runtime.KeepAlive(&bs)
11|     // 对于这个特定的例子，也可以调用
12|     // runtime.KeepAlive(bs)。
13| }
```

如何判断一个堆内存块是否仍在被使用?

目前的官方Go标准运行时（1.23版本）使用一个并发三色（tri-color）标记清扫（mark-sweep）算法（https://blog.golang.org/go15gc）来实现垃圾回收。 这里仅会对此算法的原理做一个大致的描述。一个具体实现可能和此大致描述会有很多细节上的差别。

一个垃圾回收过程分为两个阶段：标记阶段和清扫阶段。

在标记阶段，垃圾回收器（实际上是一组协程）使用三色算法来分析哪些（开辟在堆上的）内存块已经不再使用了。

- 在每一轮（见下一节的解释）垃圾回收过程的开始，所有的堆内存块将被标记为白色。
- 然后垃圾回收器将所有开辟在栈和全局内存区上的内存块标记为灰色，并把它们加入一个灰色内存块列表。

- 循环下面两步直到灰色内存块列表为空：
  1. 从个灰色内存块列表中取出一个内存块，并把它标记为黑色。
  2. 然后扫描承载在此内存块上的指针值，并通过这些指针找到它们引用着的内存块。 如果一个引用着的内存块为白色的，则将其标记为灰色并加入灰色内存块列表；否则，忽略之。

（注意这里在算法中使用三色而不是两色的原因是此标记过程是并发的。在标记的过程中，很多其它普通用户协程也正在运行中。 在此标记过程中对指针的写入需要一些额外的开销，欲更深入了解此点，请以“write barrier golang”为关键字自行搜索以深入了解。 简而言之，当在某个用户协程中，一个已经标记为黑色的内存块在标记过程中被修改而使其新引用着的一个仍标记为白色的内存块时，此白色内存块需要被标记为灰色，否则此白色内存块有可能将被认为是垃圾而回收掉；除此之外的情况不做特殊处理。）

在清扫阶段，仍被标记为白色的内存块将被认为不再使用而被回收掉。

一个不再被使用的内存块被回收后可能并不会立即释放给操作系统，这样Go运行时可以将其重新分配给其它值部使用。 不用担心，官方Go运行时的实现比大多数主流的Java运行时要消耗少得多的内存。

此垃圾回收算法不会移动内存块来整理内存碎片。

不再被使用的内存块将在什么时候被回收?

垃圾回收过程将消耗相当的CPU资源和一些内存资源。所以垃圾回收过程并非总在运行。 一个新的垃圾回收过程将在程序运行中的某些实时指标达到某些条件时才会被触发。 这些条件怎么定义的是一个垃圾回收调度问题。

官方标准运行时（runtime）的垃圾回收调度实现仍在随着版本递增而在不断地改善中。 所以很难精确地描述此实现并同时保证描述的长期有效性。这里，我仅仅列出一些相关的参考文献：

- 关于GOGC和GOMEMLIMIT环境变量（https://golang.org/pkg/runtime/#hdr-Environment_Variables）（注意GOMEMLIMIT环境变量从Go 1.19开始才支持）；
- A Guide to the Go Garbage Collector（https://go.dev/doc/gc-guide）；
- GC Pacer Redesign（https://github.com/golang/proposal/blob/master/design/44167-gc-pacer-redesign.md）；
- 我的Go程序优化101（https://gfw.go101.org/optimizations/101.html）一书中的“垃圾回收”一章（此书中文版目前尚未发布，英文版（https://go101.org/optimizations/101.html）可付费购买）。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

内存布局

本文将介绍Go中的各种类型的尺寸和对齐保证。 知晓这些保证对于估计结构体值的尺寸和正确使用64位整数原子操作函数是必要的。

Go是一门属于C语言家族的编程语言，所以本文谈及的很多概念和C语言是相通的。

Go中的类型对齐保证（alignment guarantee）

为了充分利用CPU指令来达到最佳程序性能，为一个特定类型的值开辟的内存块的起始地址必须为某个整数N的倍数。 N被称为此类型的值地址对齐保证，或者简单地称为此类型的对齐保证。 我们也可以说此类型的值的地址保证为N字节对齐的。

事实上，每个类型有两个对齐保证。当它被用做结构体类型的字段类型时的对齐保证称为此类型的字段对齐保证，其它情形的对齐保证称为此类型的一般对齐保证。

对于一个类型`T`，我们可以调用`unsafe.Alignof(t)`来获得它的一般对齐保证，其中`t`为一个`T`类型的非字段值， 也可以调用`unsafe.Alignof(x.t)`来获得`T`的字段对齐保证，其中`x`为一个结构体值并且`t`为一个类型为`T`的结构体字段值。

`unsafe`标准库包中的函数的调用都是在编译时刻估值的。

在运行时刻，对于类型为`T`的一个值`t`，我们可以调用`reflect.TypeOf(t).Align()`来获得类型`T`的一般对齐保证， 也可以调用`reflect.TypeOf(t).FieldAlign()`来获得`T`的字段对齐保证。

对于当前的官方Go标准编译器（1.23版本），一个类型的一般对齐保证和字段对齐保证总是相等的。对于gccgo编译器，这两者可能不相等。

Go白皮书仅列出了[些许类型对齐保证要求](https://golang.google.cn/ref/spec#Size_and_alignment_guarantees)（https://golang.google.cn/ref/spec#Size_and_alignment_guarantees）。

> 一个合格的Go编译器必须保证：
>
> 1. 对于任何类型的变量`x`， `unsafe.Alignof(x)`的结果最小为`1`。
> 2. 对于一个结构体类型的变量`x`， `unsafe.Alignof(x)`的结果为`x`的所有字段的对齐保证`unsafe.Alignof(x.f)`中的最大值（但是最小为`1`）。
> 3. 对于一个数组类型的变量`x`， `unsafe.Alignof(x)`的结果和此数组的元素类型的一个变量的对齐保证相等。

从这些要求可以看出，Go白皮书并未为任何类型指定了确定的对齐保证要求，它只是指定了一些最基本的要求。

即使对于同一个编译器，具体类型的对齐保证在不同的架构上也是不相同的。 同一个编译器的不同版本做出的具体类型的对齐保证也有可能是不相同的。 当前版本（1.23）的标准编译器做出的对齐保证列在了下面：

类型种类	对齐保证（字节数）
bool, uint8, int8	1
uint16, int16	2
uint32, int32	4
float32, complex64	4
数组	取决于元素类型
结构体类型	取决于各个字段类型
其它类型	一个自然字的尺寸

这里，一个自然字（native word）的尺寸在32位的架构上为4字节，在64位的架构上为8字节。

这意味着，对于当前版本的标准编译器，其它类型的对齐保证为4或者8，具体取决于程序编译时选择的目标架构。 此结论对另一个流行Go编译器gccgo也成立。

一般情况下，在Go编程中，我们不必关心值地址的对齐保证。 除非有时候我们打算优化一下内存消耗，或者编写跨平台移植性良好的Go代码。 请阅读下两节以获得详情。

类型的尺寸和结构体字节填充（structure padding）

Go白皮书只对以下种类的类型的尺寸进行了明确规定（https://golang.google.cn/ref/spec#Size_and_alignment_guarantees）。

类型种类	尺寸（字节数）
uint8, int8	1
uint16, int16	2
uint32, int32, float32	4
uint64, int64	8
float64, complex64	8
complex128	16
uint, int	取决于编译器实现。通常在32位架构上为4，在64位架构上为8。

`uintptr`	取决于编译器实现。但必须 能够存下任一个内存地址。

Go白皮书没有对其它种类的类型的尺寸做出明确规定。 请阅读值复制成本（第34章）一文来获取标准编译器使用的各种其它类型的尺寸。

标准编译器（和gccgo编译器）将确保一个类型的尺寸为此类型的对齐保证的倍数。

为了满足前面提到的各条地址对齐保证要求规则，Go编译器可能会在结构体的相邻字段之间填充一些字节。 这使得一个结构体类型的尺寸并非等于它的各个字段类型尺寸的简单相加之和。

下面是一个展示了一些字节是如何填充到一个结构体中的例子。 首先，从上面的描述中，我们已得知（对于标准编译器来说）：

- 内置类型`int8`的对齐保证和尺寸均为1个字节； 内置类型`int16`的对齐保证和尺寸均为2个字节； 内置类型`int64`的尺寸为8个字节，但它的对齐保证在32位架构上为4个字节，在64位架构上为8个字节。
- 下例中的类型`T1`和`T2`的对齐保证均为它们的各个字段的最大对齐保证。 所以它们的对齐保证和内置类型`int64`相同，即在32位架构上为4个字节，在64位架构上为8个字节。
- 类型`T1`和`T2`尺寸需为它们的对齐保证的倍数，即在32位架构上为4n个字节，在64位架构上为8n个字节。

```
 1| type T1 struct {
 2|     a int8
 3|
 4|     // 在64位架构上，为了让字段b的地址为8字节对齐，
 5|     // 需在这里填充7个字节。在32位架构上，为了让
 6|     // 字段b的地址为4字节对齐，需在这里填充3个字节。
 7|
 8|     b int64
 9|     c int16
10|
11|     // 为了让类型T1的尺寸为T1的对齐保证的倍数，
12|     // 在64位架构上需在这里填充6个字节，在32架构
13|     // 上需在这里填充2个字节。
14| }
15| // 类型T1的尺寸在64位架构上为24个字节（1+7+8+2+6），
16| // 在32位架构上为16个字节（1+3+8+2+2）。
17|
18| type T2 struct {
19|     a int8
20|
21|     // 为了让字段c的地址为2字节对齐，
```

```
22|     // 需在这里填充1个字节。
23|
24|     c int16
25|
26|     // 在64位架构上，为了让字段b的地址为8字节对齐，
27|     // 需在这里填充4个字节。在32位架构上，不需填充
28|     // 字节即可保证字段b的地址为4字节对齐的。
29|
30|     b int64
31| }
32| // 类型T2的尺寸在64位架构上位16个字节（1+1+2+4+8），
33| // 在32位架构上为12个字节（1+1+2+8）。
```

从这个例子可以看出，尽管类型`T1`和`T2`拥有相同的字段集，但是它们的尺寸并不相等。

一个有趣的事实是有时候一个结构体类型中零尺寸类型的字段可能会影响到此结构体类型的尺寸。 请阅读此问答（第51章）获取详情。

64位字原子操作的地址对齐保证要求

在此文中，64位字是指类型为内置类型`int64`或`uint64`的值。

原子操作（第40章）一文提到了一个事实：一个64位字的原子操作要求此64位字的地址必须是8字节对齐的。 这对于标准编译器目前支持的64位架构来说并不是一个问题，因为标准编译器保证任何一个64位字的地址在64位架构上都是8字节对齐的。

然而，在32位架构上，标准编译器为64位字做出的地址对齐保证仅为4个字节。 对一个不是8字节对齐的64位字进行64位原子操作将在运行时刻产生一个恐慌。 更糟的是，一些非常老旧的架构并不支持64位原子操作需要的基本指令。

`sync/atomic`标准库包文档的末尾（https://golang.google.cn/pkg/sync/atomic/#pkg-note-BUG）提到：

> On x86-32, the 64-bit functions use instructions unavailable before the Pentium MMX.
>
> On non-Linux ARM, the 64-bit functions use instructions unavailable before the ARMv6k core.
>
> On both ARM and x86-32, it is the caller's responsibility to arrange for 64-bit alignment of 64-bit words accessed atomically. The first word in a variable or in an allocated struct, array, or slice can be relied upon to be 64-bit aligned.

所以，情况并非无可挽救。

1. 这些非常老旧的架构在今日已经相当的不主流了。 如果一个程序需要在这些架构上对64位字进行原子操作，还有很多其它同步技术（第39章）可用。
2. 对其它不是很老旧的32位架构，有一些途径可以保证在这些架构上对一些64位字的原子操作是安全的。

这些途径被描述为**开辟的结构体、数组和切片值中的第一个（64位）字可以被认为是8字节对齐的**。 这里的***开辟的***应该如何解读？ 我们可以认为一个***开辟的值***为一个声明的变量、内置函数`make`的调用返回值，或者内置函数`new`的调用返回值所引用的值。 如果一个切片是从一个开辟的数组派生出来的并且此切片和此数组共享第一个元素，则我们也可以将此切片看作是一个开辟的值。

此对哪些64位字可以在32位架构上被安全地原子访问的描述是有些保守的。 有很多此描述并未包括的64位字在32位架构上也是可以被安全地原子访问的。 比如，如果一个元素类型为64位字的数组或者切片的第一个元素可以被安全地进行64位原子访问，则此数组或切片中的所有元素都可以被安全地进行64位原子访问。 只是因为很难用三言两语将所有在32位架构上可以被安全地原子访问的64位字都罗列出来，所以官方文档采取了一种保守的描述。

下面是一个展示了哪些64位字在32位架构上可以和哪些不可以被安全地原子访问的例子。

```
 1| type (
 2|    T1 struct {
 3|       v uint64
 4|    }
 5|
 6|    T2 struct {
 7|       _ int16
 8|       x T1
 9|       y *T1
10|    }
11|
12|    T3 struct {
13|       _ int16
14|       x [6]int64
15|       y *[6]int64
16|    }
17| )
18|
19| var a int64    // a可以安全地被原子访问
20| var b T1       // b.v可以安全地被原子访问
21| var c [6]int64 // c[0]可以安全地被原子访问
22|
```

```
var d T2 // d.x.v不能被安全地被原子访问
var e T3 // e.x[0]不能被安全地被原子访问

func f() {
    var f int64           // f可以安全地被原子访问
    var g = []int64{5: 0} // g[0]可以安全地被原子访问

    var h = e.x[:] // h[0]可以安全地被原子访问

    // 这里，d.y.v和e.y[0]都可以安全地被原子访问，
    // 因为*d.y和*e.y都是开辟出来的。
    d.y = new(T1)
    e.y = &[6]int64{}

    _, _, _ = f, g, h
}

// 事实上，c、g和e.y.v的所有以元素都可以被安全地原子访问。
// 只不过官方文档没有明确地做出保证。
```

如果一个结构体类型的某个64位字的字段（通常为第一个字段）在代码中需要被原子访问，为了保证此字段值在各种架构上都可以被原子访问，我们应该总是使用此结构体的开辟值。 当此结构体类型被用做另一个结构体类型的一个字段的类型时，此字段应该（尽量）被安排为另一个结构体类型的第一个字段，并且总是使用另一个结构体类型的开辟值。

如果一个结构体含有需要一个被原子访问的字段，并且我们希望此结构体可以自由地用做其它结构体的任何字段（可能非第一个字段）的类型，则我们可以用一个`[15]byte`值来模拟此64位值，并在运行时刻动态地决定此64位值的地址。 比如：

```
package mylib

import (
    "unsafe"
    "sync/atomic"
)

type Counter struct {
    x [15]byte // 模拟：x uint64
}

func (c *Counter) xAddr() *uint64 {
    // 此返回结果总是8字节对齐的。
    return (*uint64)(unsafe.Pointer(
       (uintptr(unsafe.Pointer(&c.x)) + 7)/8*8))
```

```
16| }
17|
18| func (c *Counter) Add(delta uint64) {
19|     p := c.xAddr()
20|     atomic.AddUint64(p, delta)
21| }
22|
23| func (c *Counter) Value() uint64 {
24|     return atomic.LoadUint64(c.xAddr())
25| }
```

通过采用此方法，`Counter`类型可以自由地用做其它结构体的任何字段的类型，而无需担心此类型中维护的64位字段值可能不是8字节对齐的。 此方法的缺点是，对于每个`Counter`类型的值，都有7个字节浪费了。而且此方法使用了非类型安全指针。

Go 1.19引入了一种更为优雅的方法来保证一些值的地址对齐保证为8字节。 Go 1.19在`sync/atomic`标准库包中加入了[几个原子类型](#)（第40章）。 这些类型包括`atomic.Int64`和`atomic.Uint64`。 这两个类型的值在内存中总是8字节对齐的，即使在32位架构上也是如此。 我们可以利用这个事实来确保一些64位字在32位架构上总是8字节对齐的。 比如，无论在32位架构还是64位架构上，下面的代码所示的`T`类型的`x`字段在任何情形下总是8字节对齐的。

```
1| type T struct {
2|     _ [0]atomic.Int64
3|     x int64
4| }
```

本书由[老貘](#)历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

一些可能的内存泄漏场景

当使用一门支持自动垃圾回收的语言编程时，一般来说我们不需要关心内存泄露问题，因为程序的运行时会负责回收不再使用的内存。 但是，我们确实也需要知道一些特殊的可能会造成暂时性或永久性内存泄露的情形。 本文的余下部分将列出一些这样的情形。

子字符串造成的暂时性内存泄露

Go白皮书并没有说明一个子字符串表达式的结果（子）字符串和基础字符串是否应该共享一个承载底层字节序列（第19章）的内存块（第43章）。 但标准编译器确实让它们共享一个内存块，而且很多标准库包的函数原型设计也默认了这一点。 这是一个好的设计，它不仅节省内存，而且还减少了CPU消耗。 但是有时候它会造成暂时性的内存泄露。

比如，当下面这段代码中的`demo`函数被调用之后，将会造成大约1M字节的暂时性内存泄露，直到包级变量`s0`的值在其它某处被重新修改为止。

```
 1| var s0 string // 一个包级变量
 2|
 3| // 一个演示目的函数。
 4| func f(s1 string) {
 5|     s0 = s1[:50]
 6|     // 目前，s0和s1共享着承载它们的字节序列的同一个内存块。
 7|     // 虽然s1到这里已经不再被使用了，但是s0仍然在使用中，
 8|     // 所以它们共享的内存块将不会被回收。虽然此内存块中
 9|     // 只有50字节被真正使用，而其它字节却无法再被使用。
10| }
11|
12| func demo() {
13|     s := createStringWithLengthOnHeap(1 << 20) // 1M bytes
14|     f(s)
15| }
```

为防止上面的`f`函数产生临时性内存泄露，我们可以将子字符串表达式的结果转换为一个字节切片，然后再转换回来。

```
1| func f(s1 string) {
2|     s0 = string([]byte(s1[:50]))
3| }
```

此种防止临时性内存泄露的方法不是很高效，因为在此过程中底层的字节序列被复制了两次，其中一次是不必要的。

我们可以利用官方Go标准编译器对字符串衔接所做的优化（第19章）来防止一次不必要的复制，代价是有一个字节的浪费。

```
1| func f(s1 string) {
2|     s0 = (" " + s1[:50])[1:]
3| }
```

此第二种防止临时性内存泄露的方法有可能在将来会失效，并且它对于其它编译器来说很可能是无效的。

第三种防止临时性内存泄露的方法是使用在Go 1.10种引入的`strings.Builder`类型来防止一次不必要的复制。

```
1| import "strings"
2|
3| func f(s1 string) {
4|     var b strings.Builder
5|     b.Grow(50)
6|     b.WriteString(s1[:50])
7|     s0 = b.String()
8| }
```

此第三种方法的缺点是它的实现有些啰嗦（和前两种方法相比）。 一个好消息是，从Go 1.12开始，我们可以调用`strings`标准库包中的`Repeat`函数来克隆一个字符串。 从Go 1.12开始，此函数将利用`strings.Builder`来防止一次不必要的复制。

从Go 1.18开始，`strings`标准库包中引入了一个`Clone`函数。 调用此函数为克隆一个字符串的最佳实现方式。

子切片造成的暂时性内存泄露

和子字符串情形类似，子切片也可能会造成暂时性的内存泄露。 在下面这段代码中，当函数`g`被调用之后，承载着切片`s1`的元素的内存块的开头大段内存将不再可用（假设没有其它值引用着此内存块）。 同时因为`s0`仍在引用着此内存块，所以此内存块得不到释放。

```
1| var s0 []int
2|
3| func g(s1 []int) {
4|     // 假设s1的长度远大于30。
```

```
5|     s0 = s1[len(s1)-30:]
6| }
```

如果我们想防止这样的临时性内存泄露，我们必须在函数g中将30个元素均复制一份，使得切片s0和s1不共享承载底层元素的内存块。

```
1| func g(s1 []int) {
2|     s0 = make([]int, 30)
3|     copy(s0, s1[len(s1)-30:])
4|     // 现在，如果再没有其它值引用着承载着s1元素的内存块，
5|     // 则此内存块可以被回收了。
6| }
```

因为未重置丢失的切片元素中的指针而造成的临时性内存泄露

在下面这段代码中，h函数调用之后，s的首尾两个元素将不再可用。

```
1| func h() []*int {
2|     s := []*int{new(int), new(int), new(int), new(int)}
3|     // 使用此s切片 ...
4|
5|     return s[1:3:3]
6| }
```

只要h函数调用返回的切片仍在被使用中，它的各个元素就不会回收，包括首尾两个已经丢失的元素。 因此这两个已经丢失的元素引用着的两个int值也不会被回收，即使我们再也无法使用这两个int值。

为了防止这样的暂时性内存泄露，我们必须重置丢失的元素中的指针。

```
1| func h() []*int {
2|     s := []*int{new(int), new(int), new(int), new(int)}
3|     // 使用此s切片 ...
4|
5|     s[0], s[len(s)-1] = nil, nil // 重置首尾元素指针
6|     return s[1:3:3]
7| }
```

我们经常需要在删除切片元素操作（第18章）中重置一些切片元素中的指针值。

因为协程被永久阻塞而造成的永久性内存泄露

有时，一个程序中的某些协程会永久处于阻塞状态。 Go运行时并不会将处于永久阻塞状态的协程杀掉，因此永久处于阻塞状态的协程所占用的资源将永得不到释放。

Go运行时出于两个原因并不杀掉处于永久阻塞状态的协程。 一是有时候Go运行时很难分辨出一个处于阻塞状态的协程是永久阻塞还是暂时性阻塞；二是有时我们可能故意永久阻塞某些协程。

我们应该避免因为代码设计中的一些错误而导致一些协程处于永久阻塞状态。

因为没有停止不再使用的time.Ticker值而造成的永久性内存泄露

当一个time.Timer值不再被使用，一段时间后它将被自动垃圾回收掉。 但对于一个不再使用的time.Ticker值，我们必须调用它的Stop方法结束它，否则它将永远不会得到回收。

因为不正确地使用终结器（finalizer）而造成的永久性内存泄露

将一个终结器设置到一个循环引用值组中的一个值上可能导致被此值组中的值所引用的内存块永远得不到回收（https://golang.google.cn/pkg/runtime/#SetFinalizer）。

比如，当下面这个函数被调用后，承载着x和y的两个内存块将不保证会被逐渐回收。

```
 1| func memoryLeaking() {
 2|    type T struct {
 3|       v [1<<20]int
 4|       t *T
 5|    }
 6|
 7|    var finalizer = func(t *T) {
 8|        fmt.Println("finalizer called")
 9|    }
10|
11|    var x, y T
12|
13|    // 此SetFinalizer函数调用将使x逃逸到堆上。
14|    runtime.SetFinalizer(&x, finalizer)
15|
16|    // 下面这行将形成一个包含x和y的循环引用值组。
```

```
17|     // 这有可能造成x和y不可回收。
18|     x.t, y.t = &y, &x // y也逃逸到了堆上。
19| }
```

所以，不要为一个循环引用值组中的值设置终结器。

顺便说一下，我们不应该把终结器用做析构函数（第51章）。

延迟调用函数导致的临时性内存泄露

请阅读此文（第29章）以获得详情。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

一些简单的总结

索引：

- 哪些种类型的值可以有间接底层部分？ (#type-with-underlyings)
- 哪些种类型的值可以用做内置len（以及cap、close、delete和make）函数调用的实参？ (#have-length-types)
- 各种容器类型比较 (#container-types)
- 哪些种类型的值可以用组合字面量（T{...}）表示？ (#composite-literals-types)
- 各种类型的尺寸 (#type-sizes)
- 哪些种类型的零值使用预声明的nil标识符表示？ (#nil-zero-value-types)
- 我们可以为什么样的类型声明方法？ (#types-can-have-methods)
- 什么样的类型可以被内嵌在结构体类型中？ (#types-can-be-embedded)
- 哪些函数调用将在编译时刻被估值？ (#compile-time-evaluation)
- 哪些值是可寻址的？ (#not-addressable)
- 哪些类型不支持比较？ (#not-comparable)
- 哪些代码元素允许被声明却不使用？ (#declared-but-unused)
- 哪些具名代码元素可多个被一起声明在一对小括号()中？ (#groupable-elements)
- 哪些具名代码元素的声明可以被声明在函数内也可以被声明在函数外？ (#declaration-places)
- 哪些表达式的估值结果可以包含一个额外的可选的值？ (#additional-return)
- 几种导致当前协程永久阻塞的方法 (#block-forever)
- 几种衔接字符串的方法 (#concatenate-strings)
- 官方标准编译器中实现的一些优化 (#compiler-optimizations)
- 在Go程序运行中将会产生恐慌或者崩溃的情形 (#panic-cases)

哪些种类型的值可以有间接底层部分？

在Go中，下列种类的类型的值可以有间接底层部分：

- 字符串类型
- 函数类型
- 切片类型
- 映射类型
- 通道类型
- 接口类型

注意：此答案基于标准编译器的实现。事实上，函数类型的值是否有间接底层部分是难以证明的。另外，字符串和接口类型的值在逻辑上应该被认为是不含间接底层部分。 请阅读值部一文（第17章）获取更多信息。

哪些种类型的值可以用做内置len（以及cap、close、delete和make）函数调用的实参？

	len	cap	close	delete	make
字符串值	可以				

	len	cap	close	delete	make
数组或者数组指针值	可以	可以			
切片值	可以	可以			可以
映射值	可以			可以	可以
通道值	可以	可以	可以		可以

可以被用做内置函数`len`调用的参数的值的类型都可以被称为（广义上的）容器类型。 这些容器类型的值都可以跟在`for-range`循环的`range`关键字后。

各种容器类型比较

类型	容器值是否支持添加新的元素?	容器值中的元素是否可以被替换?	容器值中的元素是否可寻址?	访问容器值元素是否会更改容器长度?	容器值是否可以有间接底层部分?
字符串	否	否	否	否	是[1]
数组	否	是[2]	是[2]	否	否
切片	否[3]	是	是	否	是
映射	是	是	否	否	是
通道	是[4]	否	否	是	是

[1] 对于标准编译器和运行时来说。
[2] 对于可寻址的数组值来说。
[3] 一般说来，一个切片的长度只能通过将另外一个切片赋值给它来被整体替换修改，这里我们不视这种情况为“添加新的元素”。 其实，切片的长度也可以通过调用`reflect.SetLen`来单独修改。增加切片的长度可以看作是一种变相的向切片添加元素。 但`reflect.SetLen`函数的效率很低，因此很少使用。
[4] 对于带缓冲并且缓冲未满的通道来说。

哪些种类型的值可以用组合字面量（`T{...}`）表示?

下面在四种类型的值（除了切片和映射类型的零值）可以用组合字面量表示。

类型（`T`）	`T{}`是类型`T`的零值?
结构体类型	是
数组类型	是
切片类型	否 （零值用`nil`表示）
映射类型	否 （零值用`nil`表示）

各种类型的尺寸

详见值复制成本（第34章）一文。

哪些种类型的零值使用预声明的`nil`标识符表示?

下面这些类型的零值可以用预声明的nil标识符表示。

类型（T）	T(nil)的尺寸
指针	1 word
切片	3 words
映射	1 word
通道	1 word
函数	1 word
接口	2 words

上表列出的尺寸为标准编译器的结果。 一个word（原生字）在32位的架构中为4个字节，在64位的架构中为8个字节。 一个Go值的间接底层部分（第17章）未统计在尺寸中。

一个类型的零值的尺寸和其它非零值的尺寸是一致的。

我们可以为什么样的类型声明方法?

详见方法（第22章）一文。

什么样的类型可以被内嵌在结构体类型中?

详见类型内嵌（第24章）一文。

哪些函数调用将在编译时刻被估值?

如果一个函数调用在编译时刻被估值，则估值结果为一个常量。

函数	返回类型	其调用是否总是在编译时刻估值?
unsafe.Sizeof	uintptr	总是如此。 *但是请注意，如果这样的一个函数调用的实参类型为一个类型参数（https://go101.org/generics/555-type-constraints-and-parameters.html），则此函数调用的结果将不被视为一个常量。*
unsafe.Alignof		
unsafe.Offsetof		
len	int	否 Go语言白皮书（https://golang.google.cn/ref/spec#Length_and_capacity）中提到： • 如果表达式s表示一个字符串常量，则表达式len(s)将在编译时刻估值； • 如果表达式s表示一个数组或者数组的指针，并且s中不含有数据接收操作和估值结果为非常量的函数调用，则表达式len(s)和cap(s)将在编译时刻估值。 *请注意，即使这样的一个函数调用在编译时刻被估值，如果函数调*

函数	返回类型	其调用是否总是在编译时刻估值?
cap		*用的实参类型为一个类型参数（https://go101.org/generics/555-type-constraints-and-parameters.html），则此函数调用的结果将不被视为一个常量。*
real	默认类型为float64（结果为类型不确定值）	否 Go语言白皮书（https://golang.google.cn/ref/spec#Constants）提到： 表达式real(s)和imag(s)在s为一个复数常量表达式时才在编译时刻估值。
imag		
complex	默认类型为complex128（结果为类型不确定值）	否 Go语言白皮书（https://golang.google.cn/ref/spec#Constants）提到： 表达式complex(sr, si)只有在sr和si都为常量表达式的时候才在编译时刻估值。

哪些值是可寻址的?

请阅读此条问答（第51章）获取详情。

哪些类型不支持比较?

请阅读此条问答（第51章）获取详情。

哪些代码元素允许被声明却不使用?

	允许被声明却不使用?
包引入	不允许
类型	允许
变量	包级全局变量允许，但局部变量不允许（对于官方标准编译器）。
常量	允许
函数	允许
跳转标签	不允许

哪些具名代码元素可多个被一起声明在一对小括号 ( ) 中?

下面这些同种类的代码元素可多个被一起声明在一对小括号 ( ) 中：

- 包引入
- 类型
- 变量
- 常量

函数是不能多个被一起声明在一对小括号 () 中的。跳转标签也不能。

哪些具名代码元素的声明可以被声明在函数内也可以被声明在函数外?

下面这些代码元素的声明既可以被声明在函数内也可以被声明在函数外：

- 类型
- 变量
- 常量

包引入必须被声明在其它种类的代码元素的声明之前。

函数必须声明在任何函数体之外。匿名函数可以定义在函数体内，但那不属于声明。

跳转标签必须声明在函数体内。

哪些表达式的估值结果可以包含一个额外的可选的值?

下列表达式的估值结果可以包含一个额外的可选的值：

	语法	额外的可选的值（语法示例中的ok）的含义	舍弃额外的可选的值会对估值行为发生影响吗?
映射元素访问	`e, ok = aMap[key]`	键值key对应的条目是否存储在映射值中	否
数据接收	`e, ok = <- aChannel`	被接收到的值e是否是在通道关闭之前发送的	否
类型断言	`v, ok = anInterface.(T)`	接口值的动态类型是否为类型T	是 （当可选的值被舍弃并且断言失败的时候，将产生一个恐慌。）

几种导致当前协程永久阻塞的方法

无需引入任何包，我们可以使用下面几种方法使当前协程永久阻塞：

1. 向一个永不会被接收数据的通道发送数据。

```
make(chan struct{}) <- struct{}{}
// 或者
make(chan<- struct{}) <- struct{}{}
```

2. 从一个未被并且将来也不会被发送数据的（并且保证永不会被关闭的）通道读取数据。

```
<-make(chan struct{})
// 或者
<-make(<-chan struct{})
// 或者
for range make(<-chan struct{}) {}
```

3. 从一个nil通道读取或者发送数据。

```
chan struct{}(nil) <- struct{}{}
// 或者
<-chan struct{}(nil)
// 或者
for range chan struct{}(nil) {}
```

4. 使用一个不含任何分支的select流程控制代码块。

```
select{}
```

几种衔接字符串的方法

详见字符串（第19章）一文。

官方标准编译器中实现的一些优化

详见Go语言101维基中的一文（https://github.com/go101/go101/wiki/The-perceivable-optimizations-made-by-the-standard-Go-compiler-%28gc%29）。

在Go程序运行中将会产生恐慌或者崩溃的情形

详见Go语言101维基中的一文（https://github.com/go101/go101/wiki/Panic-and-crash-cases）。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

Go中的nil

nil是Go中的一个使用频率很高的预声明标识符。 很多种类的类型的零值都用nil表示。 很多有其它语言编程经验的程序员在初学Go语言的时候常将nil看成是其它语言中的null或者NULL。 这种看法只是部分上正确的，但是Go中的nil和其它语言中的null或者NULL也是有很大的区别的。

本文的剩余部分将列出和nil相关的各种事实。

nil是一个预声明的标识符

我们可以直接使用它。

预声明的nil标识符可以表示很多种类型的零值

在Go中，预声明的nil可以表示下列种类（kind）的类型的零值：

- 指针类型（包括类型安全和非类型安全指针）
- 映射类型
- 切片类型
- 函数类型
- 通道类型
- 接口类型

预声明标识符nil没有默认类型

Go中其它的预声明标识符都有各自的默认类型，比如

- 预声明标识符true和false的默认类型均为内置类型bool。
- 预声明标识符iota的默认类型为内置类型int。

但是，预声明标识符nil没有一个默认类型，尽管它有很多潜在的可能类型。 事实上，预声明标识符nil是Go中唯一一个没有默认类型的类型不确定值。 我们必须在代码中提供足够的信息以便让编译器能够推断出一个类型不确定的nil值的期望类型。

一个例子：

```
 1| package main
 2|
 3| func main() {
 4|    // 代码中必须提供充足的信息来让编译器推断出某个nil的类型。
 5|    _ = (*struct{})(nil)
 6|    _ = []int(nil)
 7|    _ = map[int]bool(nil)
 8|    _ = chan string(nil)
 9|    _ = (func())(nil)
10|    _ = interface{}(nil)
11|
12|    // 下面这一组和上面这一组等价。
13|    var _ *struct{} = nil
14|    var _ []int = nil
15|    var _ map[int]bool = nil
16|    var _ chan string = nil
17|    var _ func() = nil
18|    var _ interface{} = nil
19|
20|    // 下面这行编译不通过。
21|    var _ = nil
22| }
```

nil不是一个关键字

预声明标识符nil可以被更内层的同名标识符所遮挡。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    nil := 123
 7|    fmt.Println(nil) // 123
 8|
 9|    // 下面这行编译报错，因为此行中的nil是一个int值。
10|    var _ map[string]int = nil
11| }
```

（顺便说一下，其它语言中的null和NULL也不是关键字。）

不同种类的类型的nil值的尺寸很可能不相同

一个类型的所有值的内存布局都是一样的，此类型nil值也不例外（假设此类型的零值使用nil表示）。 所以同一个类型的nil值和非nil值的尺寸是一样的。但是不同类型的nil值的尺寸可能是不一样的。

一个例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "unsafe"
 6| )
 7|
 8| func main() {
 9|    var p *struct{} = nil
10|    fmt.Println( unsafe.Sizeof( p ) ) // 8
11|
12|    var s []int = nil
13|    fmt.Println( unsafe.Sizeof( s ) ) // 24
14|
15|    var m map[int]bool = nil
16|    fmt.Println( unsafe.Sizeof( m ) ) // 8
17|
18|    var c chan string = nil
19|    fmt.Println( unsafe.Sizeof( c ) ) // 8
20|
21|    var f func() = nil
22|    fmt.Println( unsafe.Sizeof( f ) ) // 8
23|
24|    var i interface{} = nil
25|    fmt.Println( unsafe.Sizeof( i ) ) // 16
26| }
```

上例打印出来的尺寸值取决于系统架构和具体编译器实现。 上例中的输出是使用官方标准编译器编译并在64位的系统架构上运行的结果。 在32位的系统架构上，这些输出值将减半。

对于官方标准编译器，如果两个类型属于同一种（kind）类型，并且它们的零值用nil表示，则这两个类型的尺寸肯定相等。

两个不同类型的nil值可能不能相互比较

比如，下例中的两行中的比较均编译不通过。

```
1| // error: 类型不匹配
2| var _ = (*int)(nil) == (*bool)(nil)
3| // error: 类型不匹配
4| var _ = (chan int)(nil) == (chan bool)(nil)
```

请阅读Go中的值比较规则（第48章）来了解哪些值可以相互比较。 类型确定的nil值也要遵循这些规则。

下面这些比较是合法的：

```
 1| type IntPtr *int
 2| // 类型IntPtr的底层类型为*int。
 3| var _ = IntPtr(nil) == (*int)(nil)
 4|
 5| // 任何类型都实现了interface{}类型。
 6| var _ = (interface{})(nil) == (*int)(nil)
 7|
 8| // 一个双向通道可以隐式转换为和它的
 9| // 元素类型一样的单项通道类型。
10| var _ = (chan int)(nil) == (chan<- int)(nil)
11| var _ = (chan int)(nil) == (<-chan int)(nil)
```

同一个类型的两个nil值可能不能相互比较

在Go中，映射类型、切片类型和函数类型是不支持比较类型。 比较同一个不支持比较的类型的两个值（包括nil值）是非法的。 比如，下面的几个比较都编译不通过。

```
1| var _ = ([]int)(nil) == ([]int)(nil)
2| var _ = (map[string]int)(nil) == (map[string]int)(nil)
3| var _ = (func())(nil) == (func())(nil)
```

但是，映射类型、切片类型和函数类型的任何值都可以和类型不确定的裸nil标识符比较。

```
1| // 这几行编译都没问题。
2| var _ = ([]int)(nil) == nil
3| var _ = (map[string]int)(nil) == nil
4| var _ = (func())(nil) == nil
```

两个nil值可能并不相等

如果可被比较的两个nil值中的一个的类型为接口类型，而另一个不是，则比较结果总是false。 原因是，在进行此比较之前，此非接口nil值将被转换为另一个nil值的接口类型，从而将此比较转化为两个接口值的比较。 从接口（第23章）一文中，我们得知每个接口值可以看作是一个包裹非接口值的盒子。 一个非接口值被转换为一个接口类型的过程可以看作是用一个接口值将此非接口值包裹起来的过程。 一个nil接口值中什么也没包裹，但是一个包裹了nil非接口值的接口值并非什么都没包裹。 一个什么都没包裹的接口值和一个包裹了一个非接口值（即使它是nil）的接口值是不相等的。

一个例子：

```
1| fmt.Println( (interface{})(nil) == (*int)(nil) ) // false
```

访问nil映射值的条目不会产生恐慌

访问一个nil映射将得到此映射的类型的元素类型的零值。

比如：

```
1| fmt.Println( (map[string]int)(nil)["key"] ) // 0
2| fmt.Println( (map[int]bool)(nil)[123] )     // false
3| fmt.Println( (map[int]*int64)(nil)[123] )   // <nil>
```

range关键字后可以跟随nil通道、nil映射、nil切片和nil数组指针

遍历nil映射和nil切片的循环步数均为零。

遍历一个nil数组指针的循环步数为对应数组类型的长度。 （但是，如果此数组类型的长度不为零并且第二个循环变量未被舍弃或者忽略，则对应for-range循环将导致一个恐慌。）

遍历一个nil通道将使当前协程永久阻塞。

比如，下面的代码将输出0、1、2、3和4后进入阻塞状态。 Hello、world和Bye不会被输出。

```
1| for range []int(nil) {
2|     fmt.Println("Hello")
3| }
4|
5| for range map[string]string(nil) {
```

```
 6|    fmt.Println("world")
 7| }
 8|
 9| for i := range (*[5]int)(nil) {
10|    fmt.Println(i)
11| }
12|
13| for range chan bool(nil) { // 阻塞在此
14|    fmt.Println("Bye")
15| }
```

通过nil非接口属主实参调用方法不会造成恐慌

一个例子：

```
 1| package main
 2|
 3| type Slice []bool
 4|
 5| func (s Slice) Length() int {
 6|    return len(s)
 7| }
 8|
 9| func (s Slice) Modify(i int, x bool) {
10|    s[i] = x // panic if s is nil
11| }
12|
13| func (p *Slice) DoNothing() {
14| }
15|
16| func (p *Slice) Append(x bool) {
17|    *p = append(*p, x) // 如果p为空指针，则产生一个恐慌。
18| }
19|
20| func main() {
21|    // 下面这几行中的选择器不会造成恐慌。
22|    _ = ((Slice)(nil)).Length
23|    _ = ((Slice)(nil)).Modify
24|    _ = ((*Slice)(nil)).DoNothing
25|    _ = ((*Slice)(nil)).Append
26|
27|    // 这两行也不会造成恐慌。
28|    _ = ((Slice)(nil)).Length()
29|    ((*Slice)(nil)).DoNothing()
```

```
30|
31|    // 下面这两行都会造成恐慌。但是恐慌不是因为nil
32|    // 属主实参造成的。恐慌都来自于这两个方法内部的
33|    // 对空指针的解引用操作。
34|    /*
35|    ((Slice)(nil)).Modify(0, true)
36|    ((*Slice)(nil)).Append(true)
37|    */
38| }
```

事实上，上面的Append方法实现不完美。我们应该像下面这样实现之：

```
 1| func (p *Slice) Append(x bool) {
 2|    if p == nil {
 3|       *p = []bool{x}
 4|       return
 5|    }
 6|    *p = append(*p, x)
 7| }
```

如果类型T的零值可以用预声明的nil标识符表示，则*new(T)的估值结果为一个T类型的nil值

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    fmt.Println(*new(*int) == nil)         // true
 7|    fmt.Println(*new([]int) == nil)        // true
 8|    fmt.Println(*new(map[int]bool) == nil) // true
 9|    fmt.Println(*new(chan string) == nil)  // true
10|    fmt.Println(*new(func()) == nil)       // true
11|    fmt.Println(*new(interface{}) == nil)  // true
12| }
```

总结一下

在Go中，为了简单和方便，nil被设计成一个可以表示成很多种类型的零值的预声明标识符。 换句话说，它可以表示很多内存布局不同的值，而不仅仅是一个值。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

类型转换、赋值和值比较规则大全

此篇文章将列出Go中所有的类型转换、赋值和值比较规则。 请注意：在阐述这些规则的时候，自定义泛型中频繁使用的类型参数类型被特意忽略掉了。 也就是说，本文不考虑涉及到自定义泛型（https://gfw.go101.org/generics/101.html）的情形。

类型转换规则大全

在Go中，如果一个值v可以被显式地转换为类型T，则此转换可以使用语法形式(T)(v)来表示。 在大多数情况下，特别是T为一个类型名（即一个标识符）时，此形式可简化为T(v)。

当我们说一个值x可以被隐式转换为一个类型T，这同时也意味着x可以被显式转换为类型T。

1. 显然的类型转换规则

如果两个类型表示着同一个类型，则它们的值可以相互**隐式**转换为这两个类型中的任意一个。
比如，

- 类型byte和uint8的任何值可以转换为这两个类型中的任意一个。
- 类型rune和int32的任何值可以转换为这两个类型中的任意一个。
- 类型[]byte和[]uint8的任何值可以转换为这两个类型中的任意一个。

此条规则没什么可解释的，无论你是否认为此种情况中发生了转换。

2. 底层类型相关的类型转换规则

给定一个非接口值x和一个非接口类型T，并假设x的类型为Tx，

- 如果类型Tx和T的底层类型（第14章）相同（忽略掉结构体字段标签），则x可以被显式转换为类型T。
- 如果类型Tx和T中至少有一个是无名类型（第14章）并且它们的底层类型相同（考虑结构体字段标签），则x可以被**隐式**转换为类型T。

- 如果类型Tx和T的底层类型不同，但是两者都是无名的指针类型并且它们的基类型的底层类型相同（忽略掉结构体字段标签），则x可以（而且只能）被显式转换为类型T。

（注意：两处“忽略掉结构体字段标签”从Go 1.8开始生效。）

一个例子：

```
 1| package main
 2|
 3| func main() {
 4|     // 类型[]int、IntSlice和MySlice共享底层类型：[]int。
 5|     type IntSlice []int
 6|     type MySlice  []int
 7|     type Foo = struct{n int `foo`}
 8|     type Bar = struct{n int `bar`}
 9|
10|     var s  = []int{}
11|     var is = IntSlice{}
12|     var ms = MySlice{}
13|     var x map[Bar]Foo
14|     var y map[Foo]Bar
15|
16|     // 这两行隐式转换编译不通过。
17|     /*
18|     is = ms
19|     ms = is
20|     */
21|
22|     // 必须使用显式转换。
23|     is = IntSlice(ms)
24|     ms = MySlice(is)
25|     x = map[Bar]Foo(y)
26|     y = map[Foo]Bar(x)
27|
28|     // 这些隐式转换是没问题的。
29|     s = is
30|     is = s
31|     s = ms
32|     ms = s
33| }
```

指针相关的转换例子：

```
 1| package main
 2|
 3| func main() {
 4|    type MyInt int
 5|    type IntPtr *int
 6|    type MyIntPtr *MyInt
 7|
 8|    var pi = new(int)  // pi的类型为*int
 9|    var ip IntPtr = pi // 没问题，因为底层类型相同
10|                       // 并且pi的类型为无名类型。
11|
12|    // var _ *MyInt = pi // 不能隐式转换
13|    var _ = (*MyInt)(pi) // 显式转换是没问题的
14|
15|    // 类型*int的值不能被直接转换为类型MyIntPtr，
16|    // 但是可以间接地转换过去。
17|    /*
18|    var _ MyIntPtr = pi  // 不能隐式转换
19|    var _ = MyIntPtr(pi) // 也不能显式转换
20|    */
21|    var _ MyIntPtr = (*MyInt)(pi)  // 间接隐式转换没问题
22|    var _ = MyIntPtr((*MyInt)(pi)) // 间接显式转换没问题
23|
24|    // 类型IntPtr的值不能被直接转换为类型MyIntPtr，
25|    // 但是可以间接地转换过去。
26|    /*
27|    var _ MyIntPtr = ip  // 不能隐式转换
28|    var _ = MyIntPtr(ip) // 也不能显式转换
29|    */
30|    // 间接隐式或者显式转换都是没问题的。
31|    var _ MyIntPtr = (*MyInt)((*int)(ip))  // ok
32|    var _ = MyIntPtr((*MyInt)((*int)(ip))) // ok
33| }
```

3．通道相关的类型转换规则

给定一个通道值x，假设它的类型Tx是一个双向通道类型，T也是一个通道类型（无论是双向的还是单向的）。如果Tx和T的元素类型相同并且它们中至少有一个为无名类型，则x可以被**隐式**转换为类型T。

一个例子：

```
 1| package main
 2|
```

```
 3| func main() {
 4|    type C chan string
 5|    type C1 chan<- string
 6|    type C2 <-chan string
 7|
 8|    var ca C
 9|    var cb chan string
10|
11|    cb = ca // ok，因为底层类型相同
12|    ca = cb // ok，因为底层类型相同
13|
14|    // 这4行都满足此第3条转换规则的条件。
15|    var _, _ chan<- string = ca, cb // ok
16|    var _, _ <-chan string = ca, cb // ok
17|    var _ C1 = cb                   // ok
18|    var _ C2 = cb                   // ok
19|
20|    // 类型C的值不能直接转换为类型C1或C2。
21|    /*
22|    var _ = C1(ca) // compile error
23|    var _ = C2(ca) // compile error
24|    */
25|
26|    // 但是类型C的值可以间接转换为类型C1或C2。
27|    var _ = C1((chan<- string)(ca)) // ok
28|    var _ = C2((<-chan string)(ca)) // ok
29|    var _ C1 = (chan<- string)(ca)  // ok
30|    var _ C2 = (<-chan string)(ca)  // ok
31| }
```

4．和接口实现相关的类型转换规则

给定一个值x和一个接口类型I，如果x的类型（或者默认类型）为Tx并且类型Tx实现了接口类型I，则x可以被**隐式**转换为类型I。 此转换的结果为一个类型为I的接口值。此接口值包裹了

- x的一个副本（如果Tx是一个非接口值）；
- x的动态值的一个副本（如果Tx是一个接口值）。

请阅读接口（第23章）一文获取更多详情和示例。

5．类型不确定值相关的类型转换规则

如果一个类型不确定值可以表示为类型T的值，则它可以被**隐式**转换为类型T。
一个例子：

```
 1| package main
 2|
 3| func main() {
 4|    var _ []int = nil
 5|    var _ map[string]int = nil
 6|    var _ chan string = nil
 7|    var _ func()() = nil
 8|    var _ *bool = nil
 9|    var _ interface{} = nil
10|
11|    var _ int = 123.0
12|    var _ float64 = 123
13|    var _ int32 = 1.23e2
14|    var _ int8 = 1 + 0i
15| }
```

6．常量相关的类型转换规则

（此规则和上一条规则有些重叠。）

常量的类型转换结果一般仍然是一个常量（除非目标类型不是基本类型）。

给定一个常量值x和一个基本类型T，如果x可以表示成类型T的一个值，则x可以被显式地转换为类型T；特别地，如果x是一个类型不确定值，则它可以被**隐式**转换为类型T。
一个例子：

```
 1| package main
 2|
 3| func main() {
 4|    // 这些隐式转换都是合法的。
 5|    const I = 123
 6|    const I1, I2 int8 = 0x7F, -0x80
 7|    const I3, I4 int8 = I, 0.0
 8|    const F = 0.123456789
 9|    const F32 float32 = F
10|    const F32b float32 = I
11|    const F64 float64 = F
12|    const C1, C2 complex64 = F, I
13|
14|    // const F64b float64 = I3 // 这个赋值编译将失败
```

```
15|    const F64b = float64(I3)   // 这个编译没问题
16|
17|    // const I5 int = C2 // 这个赋值编译将失败
18|    const I5 = int(C2)   // 这个编译没问题
19| }
```

7．非常量数值转换规则

非常量浮点数和整数值可以被显式转换为任何浮点数和整数类型。
非常量复数值可以被显式转换为任何复数类型。
注意，

- 非常量复数值不能被转换为浮点数或整数类型。
- 非常量浮点数和整数值不能被转换为复数类型。
- 在非常量数值的转换过程中，溢出和舍入是允许的。当一个浮点数被转换为整数时，小数部分将被舍弃（向零靠拢）。

一个例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var a, b = 1.6, -1.6 // 类型均为float64
 7|    fmt.Println(int(a), int(b)) // 1 -1
 8|
 9|    var i, j int16 = 0x7FFF, -0x8000
10|    fmt.Println(int8(i), uint16(j)) // -1 32768
11|
12|    var c1 complex64 = 1 + 2i
13|    var _ = complex128(c1)
14| }
```

8．字符串相关的转换规则

如果一个值的类型（或者默认类型）为一个整数类型，则此值可以被当作一个码点值（rune值）显式转换为任何字符串类型。
一个字符串可以被显式转换为一个字节切片类型，反之亦然。 字节切片类型是指底层类型为`[]byte`的类型。
一个字符串可以被显式转换为一个码点切片类型，反之亦然。 码点切片类型是指底层类型为`[]rune`的类型。

请阅读字符串（第19章）一文获取更多详情和示例。

9．切片相关的类型转换规则

从Go 1.17开始，一个切片可以被转化为一个相同元素类型的数组的指针类型。 但是如果数组的长度大于被转化切片的长度，则将导致恐慌产生。

这里有一个例子（第18章）。

从Go 1.20开始，一个切片可以被转化为一个相同元素类型的数组。 但是如果数组的长度大于被转化切片的长度，则将导致恐慌产生。

这里有一个例子（第18章）。

10．非类型安全指针相关的类型转换规则

非类型安全指针类型是指底层类型为`unsafe.Pointer`的类型。

任何类型安全指针类型的值可以被显式转化为一个非类型安全指针类型，反之亦然。
任何uintptr值可以被显式转化为一个非类型安全指针类型，反之亦然。

请阅读非类型安全指针（第25章）一文获取详情和示例。

赋值规则

赋值可以看作是隐式类型转换。 各种隐式转换规则在上一节中已经列出。

除了这些规则，赋值语句中的目标值必须为一个可寻址的值、一个映射元素表达式或者一个空标识符。

在一个赋值中，源值被复制给了目标值。精确地说，源值的直接部分（第17章）被复制给了目标值。

注意：函数传参和结果返回其实都是赋值。

值比较规则

Go白皮书提到（https://golang.google.cn/ref/spec#Comparison_operators）：

> 在任何比较中，第一个比较值必须能被赋值给第二个比较值的类型，或者反之。

所以，值比较规则和赋值规则非常相似。 换句话说，两个值是否可以比较取决于其中一个值是否可以隐式转换为另一个值的类型。 很简单？此规则描述基本正确，但是存在另外一条优先级更高的规则：

> 如果一个比较表达式中的两个比较值均为类型确定值，则它们的类型必须都属于可比较类型（第14章）。

按照上面这条规则，如果一个不可比较类型（肯定是一个非接口类型）实现了一个接口类型，则比较这两个类型的值是非法的，即使前者的值可以隐式转化为后者。

注意，尽管切片/映射/函数类型为不可比较类型，但是它们的值可以和类型不确定的预声明`nil`标识符比较。

上述规则并未覆盖所有的情况。如果两个值均为类型不确定值，它们可以比较吗？这种情况的规则比较简单：

- 两个类型不确定的布尔值可以相互比较。
- 两个类型不确定的数字值可以相互比较。
- 两个类型不确定的字符串值可以相互比较。

两个类型不确定的数字值的比较结果服从直觉。

注意，两个类型不确定的`nil`值不能相互比较。

任何比较的结果均为一个类型不确定的布尔值。

一些值比较的例子：

```
 1| package main
 2|
 3| // 一些类型为不可比较类型的变量。
 4| var s []int
 5| var m map[int]int
 6| var f func()()
 7| var t struct {x []int}
 8| var a [5]map[int]int
 9|
10| func main() {
11|     // 这些比较编译不通过。
12|     /*
13|     _ = s == s
14|     _ = m == m
15|     _ = f == f
16|     _ = t == t
17|     _ = a == a
18|     _ = nil == nil
```

```
19|     _ = s == interface{}(nil)
20|     _ = m == interface{}(nil)
21|     _ = f == interface{}(nil)
22|     */
23|
24|     // 这些比较编译都没问题。
25|     _ = s == nil
26|     _ = m == nil
27|     _ = f == nil
28|     _ = 123 == interface{}(nil)
29|     _ = true == interface{}(nil)
30|     _ = "abc" == interface{}(nil)
31| }
```

两个值是如何进行比较的?

假设两个值可以相互比较，并且它们的类型同为T。（如果它们的类型不同，则其中一个可以转换为另一个的类型。这里我们不考虑两者均为类型不确定值的情形。）

1. 如果T是一个布尔类型，则这两个值只有在它们同为true或者false的时候比较结果才为true。
2. 如果T是一个整数类型，则这两个值只有在它们在内存中的表示完全一致的情况下比较结果才为true。
3. 如果T是一个浮点数类型，则这两个值只要满足下面任何一种情况，它们的比较结果就为true：
    - 它们都为+Inf；
    - 它们都为-Inf；
    - 它们都为-0.0或者都为+0.0。
    - 它们都不是NaN并且它们在内存中的表示完全一致。
4. 如果T是一个复数类型，则这两个值只有在它们的实部和虚部均做为浮点数进行进行比较的结果都为true的情况下比较结果才为true。
5. 如果T是一个指针类型（类型安全或者非类型安全），则这两个值只有在它们所表示的地址值相等或者它们都为nil的情况下比较结果才为true。
6. 如果T是一个通道类型，则这两个值只有在它们引用着相同的底层内部通道或者它们都为nil时比较结果才为true。
7. 如果T是一个结构体类型，则它们的相应字段将逐对进行比较（第16章）。只要有一对字段不相等，这两个结构体值就不相等。
8. 如果T是一个数组类型，则它们的相应元素将逐对进行比较（第18章）。只要有一对元素不相等，这两个结构体值就不相等。

9. 如果T是一个接口类型，请参阅两个接口值是如何进行比较的（第23章）。
10. 如果T是一个字符串类型，请参阅两个字符串值是如何进行比较的（第19章）。

请注意，动态类型均为同一个不可比较类型的两个接口值的比较将产生一个恐慌。比如下面的例子：

```
 1| package main
 2|
 3| func main() {
 4|     type T struct {
 5|         a interface{}
 6|         b int
 7|     }
 8|     var x interface{} = []int{}
 9|     var y = T{a: x}
10|     var z = [3]T{{a: y}}
11|
12|     // 这三个比较均会产生一个恐慌。
13|     _ = x == x
14|     _ = y == y
15|     _ = z == z
16| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/golang101/golang101获取本书最新版）

Go中的一些语法/语义例外

本篇文章将列出Go中的各种语法和语义例外。 这些例外中的一些属于方便编程的语法糖，一些属于内置泛型特权，一些源于历史原因或者其它各种逻辑原因。

嵌套函数调用

基本规则：

> 如果一个函数（包括方法）调用的返回值个数不为零，并且它的返回值列表可以用做另一个函数调用的实参列表，则此前者调用可以被内嵌在后者调用之中，但是此前者调用不可和其它的实参混合出现在后者调用的实参列表中。

语法糖：

> 如果一个函数调用刚好返回一个结果，则此函数调用总是可以被当作一个单值实参用在其它函数调用中，并且此函数调用可以和其它实参混合出现在其它函数调用的实参列表中。

例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5| )
 6|
 7| func f0() float64 {return 1}
 8| func f1() (float64, float64) {return 1, 2}
 9| func f2(float64, float64) {}
10| func f3(float64, float64, float64) {}
11| func f4()(x, y []int) {return}
12| func f5()(x map[int]int, y int) {return}
13|
14| type I interface {m()(float64, float64)}
15| type T struct{}
16| func (T) m()(float64, float64) {return 1, 2}
17|
18| func main() {
19|    // 这些行编译没问题。
20|    f2(f0(), 123)
21|    f2(f1())
22|    fmt.Println(f1())
```

```
23|    _ = complex(f1())
24|    _ = complex(T{}.m())
25|    f2(I(T{}).m())
26|
27|    // 这些行编译不通过。
28|    /*
29|    f3(123, f1())
30|    f3(f1(), 123)
31|    */
32|
33|    // 此行从Go官方工具链1.15开始才能够编译通过。
34|    println(f1())
35|
36|    // 下面这三行从Go官方工具链1.13开始才能够编译通过。
37|    copy(f4())
38|    delete(f5())
39|    _ = complex(I(T{}).m())
40| }
```

选择结构体字段值

基本规则：

> 指针类型和值没有字段。

语法糖：

> 我们可以通过一个结构体值的指针来选择此结构体的字段。

例子：

```
 1| package main
 2|
 3| type T struct {
 4|    x int
 5| }
 6|
 7| func main() {
 8|    var t T
 9|    var p = &t
10|
11|    p.x *= 2
12|    // 上一行是下一行的语法糖。
13|    (*p).x *= 2
14| }
```

方法调用的属主实参

基本规则：

> 为类型*T显式声明的方法肯定不是类型T的方法。

语法糖：

> 尽管为类型*T显式声明的方法肯定不是类型T的方法，但是可寻址的T值可以用做这些方法的调用的属主实参。

例子：

```
 1| package main
 2|
 3| type T struct {
 4|     x int
 5| }
 6|
 7| func (pt *T) Double() {
 8|     pt.x *= 2
 9| }
10|
11| func main() {
12|     // T{3}.Double() // error: T值没有Double方法。
13|
14|     var t = T{3}
15|
16|     t.Double() // 现在：t.x == 6
17|     // 上一行是下一行的语法糖。
18|     (&t).Double() // 现在：t.x == 12
19| }
```

取组合字面量的地址

基本规则：

> 组合字面量是不可寻址的，并且是不可寻址的值是不能被取地址的。

语法糖：

> 尽管组合字面量是不可寻址的，它们仍可以被显式地取地址。

请阅读结构体（第16章）和内置容器类型（第18章）两篇文章获取详情。

指针值和选择器

基本规则：

> 一般说来，具名（第14章）的指针类型的值不能使用在选择器语法形式中。

语法糖：

> 如果值x的类型为一个一级具名指针类型，并且(*x).f是一个合法的选择器，则x.f也是合法的。

任何多级指针均不能出现在选择器语法形式中。

上述语法糖的例外：

> 上述语法糖只对f为字段的情况才有效，对于f为方法的情况无效。

例子：

```
 1| package main
 2|
 3| type T struct {
 4|    x int
 5| }
 6|
 7| func (T) y() {
 8| }
 9|
10| type P *T
11| type PP **T // 一个多级指针类型
12|
13| func main() {
14|    var t T
15|    var p P = &t
16|    var pt = &t   // pt的类型为*T
17|    var ppt = &pt // ppt的类型为**T
18|    var pp PP = ppt
19|    _ = pp
20|
21|    _ = (*p).x // 合法
22|    _ = p.x    // 合法（因为x为一个字段）
23|
24|    _ = (*p).y // 合法
25|    // _ = p.y // 不合法（因为y为一个方法）
26|
27|    // 下面的选择器均不合法。
28|    /*
29|    _ = ppt.x
30|    _ = ppt.y
```

```
31|     _ = pp.x
32|     _ = pp.y
33|     */
34| }
```

容器和容器元素的可寻址性

基本规则：

> 如果一个容器值是可寻址的，则它的元素也是可寻址的。

例外：

> 一个映射值的元素总是不可寻址的，即使此映射本身是可寻址的。

语法糖：

> 一个切片值的元素总是可寻址的，即使此切片值本身是不可寻址的。

例子：

```
 1| package main
 2|
 3| func main() {
 4|     var m = map[string]int{"abc": 123}
 5|     _ = &m // okay
 6|
 7|     // 例外。
 8|     // p = &m["abc"] // error: 映射元素不可寻址
 9|
10|     // 语法糖。
11|     f := func() []int {
12|         return []int{0, 1, 2}
13|     }
14|     // _ = &f() // error: 函数调用是不可寻址的
15|     _ = &f()[2] // okay
16| }
```

修改值

基本规则：

> 不可寻址的值不可修改。

例外：

> 尽管映射元素是不可寻址的，但是它们可以被修改（但是它们必须被整个覆盖修改）。

例子：

```
 1| package main
 2|
 3| func main() {
 4|    type T struct {
 5|       x int
 6|    }
 7|
 8|    var mt = map[string]T{"abc": {123}}
 9|    // _ = &mt["abc"]     // 映射元素是不可寻址的
10|    // mt["abc"].x = 456  // 部分修改是不允许的
11|    mt["abc"] = T{x: 789} // 整体覆盖修改是可以的
12| }
```

函数参数

基本规则：

> 函数的每个参数一般为某个类型一个值。

例外：

> 内置`make`和`new`函数的第一个参数为一个类型。

同一个代码包中的函数命名

基本规则：

> 同一个代码包中声明的函数的名称不能重复。

例外：

> 同一个代码包中可以声明若干个名为`init`类型为`func()`的函数。

函数调用

基本规则：

> 名称为非空标识符的函数可以被调用。

例外：

> `init`函数不可被调用。

函数值

基本规则：

> 声明的函数可以被用做函数值。

例外：

> `init`函数不可被用做函数值。

例子：

```
 1| package main
 2|
 3| import "fmt"
 4| import "unsafe"
 5|
 6| func init() {}
 7|
 8| func main() {
 9|     // 这两行编译没问题。
10|     var _ = main
11|     var _ = fmt.Println
12|
13|     // 下面这行编译不通过。
14|     var _ = init
15| }
```

泛型类型实参的传递方式

基本规则：

> 在泛型类型实参列表中，所有实参均包裹在同一对方括号中，各个实参之间使用逗号分开

例外：

> 内置泛型类型的类型实参传递形态各异。映射类型的键值类型实参单独包裹在一对方括号中，其它实参并没有被包裹。 内置`new`泛型函数的类型实参是包裹在一对圆括号中。 内置`make`泛型函数的类型实参是和值实参混杂在一起并包裹在同一对圆括号中。

舍弃函数调用返回值

基本规则：

> 一个函数调用的所有返回值可以被一并忽略舍弃。

例外：

> 内置函数（展示在`builtin`和`unsafe`标准库包中的函数）调用的返回值不能被舍弃。

例外中的例外：

> 内置函数copy和recover的调用的返回值可以被舍弃。

声明的变量

基本规则：

> 声明的变量总是可寻址的。

例外：

> 预声明的nil（https://golang.google.cn/pkg/builtin/#pkg-variables）变量是不可寻址的。

所以，预声明的nil是一个不可更改的变量。

传参

基本规则：

> 当一个实参被传递给对应的形参时，此实参必须能够赋值给此形参类型。

语法糖：

> 如果内置函数copy和append的一个调用的第一个形参为一个字节切片（这时，一般来说，第二形参也应该是一个字节切片），则第二个形参可以是一个字符串，即使字符串不能被赋给一个字节切片。（假设append函数调用的第二个形参使用arg...形式传递。）

例子：

```
 1| package main
 2|
 3| func main() {
 4|     var bs = []byte{1, 2, 3}
 5|     var s = "xyz"
 6|
 7|     copy(bs, s)
 8|     // 上一行是下一行的语法糖和优化。
 9|     copy(bs, []byte(s))
10|
11|     bs = append(bs, s...)
12|     // 上一行是下一行的语法糖和优化。
13|     bs = append(bs, []byte(s)...)
14| }
```

比较

基本规则：

> 映射、切片和函数类型是不支持比较的。

例外：

> 映射、切片和函数值可以和预声明标识符`nil`比较。

例子：

```
 1| package main
 2|
 3| func main() {
 4|    var s1 = []int{1, 2, 3}
 5|    var s2 = []int{7, 8, 9}
 6|    //_ = s1 == s2 // error: 切片值不可比较。
 7|    _ = s1 == nil  // ok
 8|    _ = s2 == nil  // ok
 9|
10|    var m1 = map[string]int{}
11|    var m2 = m1
12|    // _ = m1 == m2 // error: 映射值不可比较。
13|    _ = m1 == nil   // ok
14|    _ = m2 == nil   // ok
15|
16|    var f1 = func(){}
17|    var f2 = f1
18|    // _ = f1 == f2 // error: 函数值不可比较。
19|    _ = f1 == nil   // ok
20|    _ = f2 == nil   // ok
21| }
```

比较二

基本规则：

> 如果一个值可以隐式转换为一个可比较类型，则这此值和此可比较类型的值可以用==和!=比较符来做比较。

例外：

> 一个不可比较类型（一定是一个非接口类型）的值不能和一个接口类型的值比较，即使此不可比较类型实现了此接口类型（从而此不可比较类型的值可以被隐式转换为此接口类型）。

请阅读值比较规则（第48章）获取详情。

空组合字面量

基本规则：

> 如果一个类型T的值可以用组合字面量表示，则T{}表示此类型的零值。

例外：

> 对于一个映射或者切片类型T，T{}不是它的零值，它的零值使用预声明的nil表示。

例子：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    // new(T)返回类型T的一个零值的地址。
 7|
 8|    type T0 struct {
 9|       x int
10|    }
11|    fmt.Println( T0{} == *new(T0) ) // true
12|    type T1 [5]int
13|    fmt.Println( T1{} == *new(T1) ) // true
14|
15|    type T2 []int
16|    fmt.Println( T2{} == nil ) // false
17|    type T3 map[int]int
18|    fmt.Println( T3{} == nil ) // false
19| }
```

容器元素遍历

基本规则：

> 只有容器值可以跟在range关键字后，for-range循环遍历出来的是容器值的各个元素。 每个容器元素对应的键值（或者索引）也将被一并遍历出来。

例外1：

> 当range关键字后跟的是字符串时，遍历出来的是码点值，而不是字符串的各个元素byte字节值。

例外2：

> 当range关键字后跟的是通道时，通道的元素的键值（次序）并未被一同遍历出来。

语法糖：

> 尽管数组指针不属于容器，但是range关键字后可以跟数组指针来遍历数组元素。

内置类型的方法

基本规则：

> 内置类型都没有方法。

例外：

> 内置类型error有一个Error() string方法。

值的类型

基本规则：

> 每个值要么有一个确定的类型要么有一个默认类型。

例外：

> 类型不确定的nil值既没有确定的类型也没有默认类型。

常量值

基本规则：

> 常量值的值固定不变。常量值可以被赋给变量值。

例外1：

> 预声明的iota是一个绑定了0的常量，但是它的值并不固定。 在一个包含多个常量描述的常量声明中，如果一个iota的值出现在一个常量描述中，则它的值将被自动调整为此常量描述在此常量声明中的次序值，尽管此调整发生在编译时刻。

例外2：

> iota只能被用于常量声明中，它不能被赋给变量。

舍弃表达式中可选的结果值对程序行为的影响

基本规则：

> 表达式中可选的结果值是否被舍弃不会对程序行为产生影响。

例外：

> 当一个失败的类型断言表达式的可选的第二个结果值被舍弃时，当前协程将产生一个恐慌。

例子：

```
 1| package main
 2|
 3| func main() {
 4|     var ok bool
 5|
 6|     var m = map[int]int{}
 7|     _, ok = m[123]
 8|     _ = m[123] // 不会产生恐慌
 9|
10|     var c = make(chan int, 2)
11|     c <- 123
12|     close(c)
13|     _, ok = <-c
14|     _ = <-c // 不会产生恐慌
15|
16|     var v interface{} = "abc"
17|     _, ok = v.(int)
18|     _ = v.(int) // 将产生一个恐慌！
19| }
```

else关键字后跟随另一个代码块

基本规则：

> else关键字后必须跟随一个显式代码块{...}。

语法糖：

> else关键字后可以跟随一个（隐式）if代码块，

比如，在下面这个例子中，函数foo编译没问题，但是函数bar编译不过。

```
 1| func f() []int {
 2|     return nil
 3| }
 4|
 5| func foo() {
 6|     if vs := f(); len(vs) == 0 {
 7|     } else if len(vs) == 1 {
```

```
 8|    }
 9|
10|    if vs := f(); len(vs) == 0 {
11|    } else {
12|       switch {
13|       case len(vs) == 1:
14|       default:
15|       }
16|    }
17|
18|    if vs := f(); len(vs) == 0 {
19|    } else {
20|       for _, _ = range vs {
21|       }
22|    }
23| }
24|
25| func bar() {
26|    if vs := f(); len(vs) == 0 {
27|    } else switch { // error
28|    case len(vs) == 1:
29|    default:
30|    }
31|
32|    if vs := f(); len(vs) == 0 {
33|    } else for _, _ = range vs { // error
34|    }
35| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

Go细节101

索引：

- 代码包相关的细节：
  - 一个包可以在一个源文件里被引入多次。 (#import-package-multiple-times)
- 控制流相关的细节：
  - 在switch和select流程控制代码块中，default分支可以放在所有的case分支之前或者所有的case分支之后，也可以放在case分支之间。 (#default-branch-position)
  - switch流程控制代码块中的数字常量case表达式不能重复，但是布尔常量case表达式可以重复。 (#constant-case-expressions)
  - switch流程控制代码块里的switch表达式总是被估值为类型确定值。 (#switch-expressions-are-typed)
  - switch流程控制代码块中的switch表达式的缺省默认值为类型确定值true（其类型为预声明类型bool）。 (#default-switch-expression-value)
  - 有时候，显式代码块的开括号{可以放在下一行。 (#open-brace-on-the-next-line)
  - 有些case分支代码块必须是显式的。 (#some-case-branch-blocks-must-be-explicit)
  - 嵌套的延迟函数调用可以修改外层函数的返回结果。 (#defer-modify-results)
  - 某些recover函数调用是空操作。 (#no-op-recover-calls)
  - 我们可以使用os.Exit函数调用退出一个程序和使用runtime.Goexit函数调用退出一个协程。 (#os-exit-runtime-goexit)
- 操作符相关的细节：
  - 递增运算符++和递减运算符--的优先级低于解引用运算符*和取地址运算符&，解引用运算符和取地址运算符的优先级低于选择器.中的属性选择操作符。 (#some-operator-precedence)
  - 移位运算中的左类型不确定操作数的类型推断规则取决于右操作数是否是常量。 (#type-deduction-in-bit-shift)
- 指针相关的细节：
  - 如果两个指针的类型具有不同的底层类型但是它们的基类型却共享相同的底层类型，则这两个指针值可以间接相互转换为对方的类型。 (#conversion-of-pointers-with-different-underlying-types)
  - 两个零尺寸值的地址可能相等，也可能不相等。 (#addresses-of-zeor-sized-values)

    - 一个指针类型的基类型可以为此指针类型自身。(#self-based-pointer-type)
    - 有关选择器缩写形式的细节。(#pointer-selector-shorthands)
- 容器相关的细节：
    - 有时候，嵌套组合字面量可以被简化。(#composite-literal-simplification)
    - 在某些情形下，我们可以将数组指针当作数组来用。(#use-pointer-as-array)
    - 从nil映射中读取元素不会导致崩溃，读取结果是一个零元素值。(#nil-map-element-retrieving)
    - 从一个nil映射中删除一个条目不会导致崩溃，这是一个空操作。(#nil-map-element--deletion)
    - `append`函数调用的结果可能会与原始切片共享一些元素，也可能不共享任何元素。(#append-slice-elements)
    - 从一个基础切片派生出的子切片的长度可能大于基础切片的长度。(#subslice)
    - 从一个nil切片中派生子切片是允许的，只要子切片表达式中使用的所有索引都为零，则不会有恐慌产生，结果子切片同样是一个nil切片。(#derive-from-nil-slice)
    - 用`range`遍历nil映射或者nil切片是没问题的，这属于空操作。(#range-over-nil-map-and-slice)
    - 用`range`遍历nil数组指针时，如果忽略或省略第二个迭代变量，则此遍历是没问题的。遍历中的循环步数为相应数组类型的长度。(#range-over-nil-array-pointer)
    - 切片的长度和容量可以被单独修改。(#modify-slice-length-and-capacity-separately)
    - 切片和数组组合字面量中的索引必须是非负常量。(#indexes-in-composite-literals)
    - 切片/数组/映射组合字面量的常量索引和键值不能重复。(#constant-keys-in-composite-literals)
    - 不可寻址的数组的元素依旧是不可寻址的，但是不可寻址的切片的元素总是可寻址的。(#array-slice-element-addressability)
    - 可以从不可寻址的切片派生子切片，但是不能从不可寻址的数组派生子切片。(#array-slice-derivable)
    - 把以`NaN`做为键值的条目放如映射就宛如把条目放入黑洞一样。(#nan-as-map-keys)
    - 字符串转换为byte切片或rune切片后的结果切片的容量可能会大于长度。(#capacity-of-slice-from-string)
    - 对于切片`s`，循环`for i = range s {...}`并不等价于循环`for i = 0; i < len(s); i++ {...}`。(#different-loops)
    - 一个映射中的条目的遍历次序在两次遍历中可能并不相同。我们可以认为映射中的条目的遍历次序是随机的。(#map-entry-iteration)

    - 在对一个映射进行条目遍历期间，在此映射中创建的新条目可能会在当前遍历中被遍历出来，也可能不会。(#create-entry-in-map-entry-iteration)
- 函数和方法相关的细节：
    - 一个多返回值函数调用表达式不能和其它表达式混用在一个赋值语句的右侧或者另一个函数调用的实参列表中。(#call-as-expression)
    - 某些函数调用是在编译时刻被估值的。(#compile-time-evaluated-calls)
    - 每一个方法都对应着一个隐式声明的函数。(#method-as-function)
- 接口相关的细节：
    - 如果两个接口值具有相同的动态类型并且此动态类型不支持比较，则比较这两个接口值将导致一个恐慌。(#comparison-interfaces-may-panic)
    - 类型断言可以用于将一个接口值转换为另一个接口类型，即使此接口值的类型并未实现另一个接口类型。(#type-assertion-to-interface-type)
    - 一个失败的类型断言的可选的第二个结果是否被舍弃将影响此类型断言的行为。(#failed-type-assertion-may-panic)
    - 关于在编译时刻即可确定总是失败的目标类型为接口类型的断言。(#impossible-to-interface-assertion)
    - 以相同实参调用两次`errors.New`函数返回的两个`error`值是不相等的。(#errors-new-different-returns)
- 管道相关的细节：
    - 单向接收通道无法被关闭。(#receive-only-channel-canot-be-closed)
    - 发送一个值到一个已关闭的通道被视为一个非阻塞操作，该操作会导致恐慌。(#send-to-closed-channel-is-non-blocking)
- 更多关于类型和值的细节：
    - 类型可以在声明函数体内。(#local-type-declaration)
    - 对于标准编译器，结构体中的某些零尺寸字段的尺寸有可能会被视为一个字节。(#final-zero-size-field)
    - NaN != NaN，Inf == Inf。(#nan-inf)
    - 不同代码包中的两个非导出方法名和结构体字段名总是被视为不同的名称。(#non-exported-names-from-different-packages)
    - 在结构体值的比较中，名为空标识符的字段将被忽略。(#blank-fields-are-ignored-in-comparisons)
- 杂项：
    - 在某些很少见的场景中，圆括号是必需的。(#sometimes-parentheses-are-required)
    - 栈溢出不可被挽救，它将使程序崩溃。(#stack-overflow-is-unrecoverable)

    - 某些表达式的估值顺序取决于具体编译器实现。 (#some-evaluation-order-are-compiler-dependent)
- 标准包相关的细节：
    - reflect.DeepEqual(x, y)和x == y的结果可能会不同。 (#reflect-deep-equal)
    - reflect.Value.Bytes()方法返回一个[]byte值，它的元素类型byte可能并非属主参数代表的Go切片值的元素类型。 (#reflect-value-bytes)
    - 我们应该使用os.IsNotExist(err)而不是err == os.ErrNotExist来检查文件是否存在。 (#check-file-existent)
    - flag标准库包对待布尔选项不同于整数和字符串选项。 (#boolean-flag)
    - [Sp|Fp|P]rintf支持位置参数。 (#printf-positional-arguments)

一个包可以在一个源文件里被引入多次。

一个Go源文件可以多次引入同一个包。但是每次的引入名称必须不同。这些相同的包引入引用着同一个包实例。

示例：

```
1| package main
2|
3| import "fmt"
4| import "io"
5| import inout "io"
6|
7| func main() {
8|     fmt.Println(&inout.EOF == &io.EOF) // true
9| }
```

在switch和select流程控制代码块中，default分支可以放在所有的case分支之前或者所有的case分支之后，也可以放在case分支之间。

示例：

```
1|     switch n := rand.Intn(3); n {
2|     case 0: fmt.Println("n == 0")
```

```
 3|    case 1: fmt.Println("n == 1")
 4|    default: fmt.Println("n == 2")
 5|    }
 6|
 7|    switch n := rand.Intn(3); n {
 8|    default: fmt.Println("n == 2")
 9|    case 0: fmt.Println("n == 0")
10|    case 1: fmt.Println("n == 1")
11|    }
12|
13|    switch n := rand.Intn(3); n {
14|    case 0: fmt.Println("n == 0")
15|    default: fmt.Println("n == 2")
16|    case 1: fmt.Println("n == 1")
17|    }
18|
19|    var x, y chan int
20|
21|    select {
22|    case <-x:
23|    case y <- 1:
24|    default:
25|    }
26|
27|    select {
28|    case <-x:
29|    default:
30|    case y <- 1:
31|    }
32|
33|    select {
34|    default:
35|    case <-x:
36|    case y <- 1:
37|    }
```

switch流程控制代码块中的数字常量case表达式不能重复，但是布尔常量case表达式可以重复。

例如，下面的代码在编译时会失败。

```
 1| package main
 2|
 3| func main() {
```

```
4|    switch 123 {
5|    case 123:
6|    case 123: // error: duplicate case
7|    }
8| }
```

但是下面的代码在编译时是没问题的。

```
1| package main
2|
3| func main() {
4|    switch false {
5|    case false:
6|    case false:
7|    }
8| }
```

关于原因，请阅读这个issue（https://github.com/golang/go/issues/28357）。此行为依赖于编译器。事实上，标准编译器同样不允许重复的字符串case表达式，但是gccgo编译器却允许。

switch流程控制代码块里的switch表达式总是被估值为类型确定值。

例如，在下列switch代码块中的switch表达式123被视为一个int值，而不是一个类型不确定的整数。

```
1| package main
2|
3| func main() {
4|    switch 123 {
5|    case int64(123):  // error: 类型不匹配
6|    case uint32(789): // error: 类型不匹配
7|    }
8| }
```

switch流程控制代码块中的switch表达式的缺省默认值为类型确定值true（其类型为预声明类型bool）。

例如，下列程序会打印出true。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    switch { // <=> switch true {
 7|    case true:  fmt.Println("true")
 8|    case false: fmt.Println("false")
 9|    }
10| }
```

有时候，显式代码块的开括号{可以放在下一行。

例如：

```
 1| package main
 2|
 3| func main() {
 4|    var i = 0
 5| Outer:
 6|    for
 7|    { // 在这里断行是没问题的
 8|       switch
 9|       { // 在这里断行是没问题的
10|       case i == 5:
11|          break Outer
12|       default:
13|          i++
14|       }
15|    }
16| }
```

下面程序的结果会打印什么？`true`还是`false`？ 答案是`true`。 关于原因请阅读Go中的代码断行规则（第28章）一文。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func False() bool {
 6|    return false
 7| }
```

```
 8|
 9| func main() {
10|    switch False()
11|    {
12|    case true:  fmt.Println("true")
13|    case false: fmt.Println("false")
14|    }
15| }
```

有些case分支代码块必须是显式的。

例如，下面的程序会在编译时将失败。

```
 1| func demo(n, m int) (r int) {
 2|    switch n {
 3|    case 123:
 4|       if m > 0 {
 5|          goto End
 6|       }
 7|       r++
 8|
 9|       End: // syntax error: 标签后缺少语句
10|    default:
11|       r = 1
12|    }
13|    return
14| }
```

为了编译通过，case分支代码块必须改成显式的：

```
 1| func demo(n, m int) (r int) {
 2|    switch n {
 3|    case 123: {
 4|       if m > 0 {
 5|          goto End
 6|       }
 7|       r++
 8|
 9|       End:
10|    }
11|    default:
12|       r = 1
13|    }
```

```
14|    return
15| }
```

另外，我们可以在标签`End:`之后加一个分号，如下所示：

```
 1| func demo(n, m int) (r int) {
 2|    switch n {
 3|    case 123:
 4|       if m > 0 {
 5|          goto End
 6|       }
 7|       r++
 8|
 9|       End:;
10|    default:
11|       r = 1
12|    }
13|    return
14| }
```

关于原因，请阅读Go的代码断行规则（第28章）一文。

嵌套的延迟函数调用可以修改外层函数的返回结果。

例如：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func F() (r int) {
 6|    defer func() {
 7|       r = 789
 8|    }()
 9|
10|    return 123 // <=> r = 123; return
11| }
12|
13| func main() {
14|    fmt.Println(F()) // 789
15| }
```

某些recover函数调用是空操作。

我们需要在正确的地方调用recover函数。 关于细节，请阅读 在正确的位置调用内置函数recover （第31章）一文。

我们可以使用os.Exit函数调用退出一个程序和使用runtime.Goexit函数调用退出一个协程。

我们可以通过调用os.Exit函数从任何函数里退出一个程序。 os.Exit函数调用接受一个int代码值做为参数并将此代码返回给操作系统。

示例：

```
 1| // exit-example.go
 2| package main
 3|
 4| import "os"
 5| import "time"
 6|
 7| func main() {
 8|    go func() {
 9|       time.Sleep(time.Second)
10|       os.Exit(1)
11|    }()
12|    select{}
13| }
```

运行：

```
$ go run a.go
exit status 1
$ echo $?
1
```

我们可以通过调用runtime.Goexit函数退出一个goroutine。 runtime.Goexit函数没有参数。

在下面的示例中，文字Java将不会被打印出来。

```
 1| package main
 2|
 3| import "fmt"
 4| import "runtime"
```

```
 5|
 6| func main() {
 7|    c := make(chan int)
 8|    go func() {
 9|       defer func() {c <- 1}()
10|       defer fmt.Println("Go")
11|       func() {
12|          defer fmt.Println("C")
13|          runtime.Goexit()
14|       }()
15|       fmt.Println("Java")
16|    }()
17|    <-c
18| }
```

递增运算符++和递减运算符--的优先级低于解引用运算符*和取地址运算符&，解引用运算符和取地址运算符的优先级低于选择器.中的属性选择操作符。

例如：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type T struct {
 6|    x int
 7|    y *int
 8| }
 9|
10| func main() {
11|    var t T
12|    p := &t.x // <=> p := &(t.x)
13|    fmt.Printf("%T\n", p) // *int
14|
15|    *p++ // <=> (*p)++
16|    *p-- // <=> (*p)--
17|
18|    t.y = p
19|    a := *t.y // <=> *(t.y)
20|    fmt.Printf("%T\n", a) // int
21| }
```

移位运算中的左类型不确定操作数的类型推断规则取决于右操作数是否是常量。

```
 1| package main
 2|
 3| func main() {
 4| }
 5|
 6| const M  = 2
 7| var _ = 1.0 << M // 编译没问题。1.0将被推断为一个int值。
 8|
 9| var N = 2
10| var _ = 1.0 << N // 编译失败。1.0将被推断为一个float64值。
```

关于原因请阅读运算操作符（第8章）一文。

如果两个指针的类型具有不同的底层类型但是它们的基类型却共享相同的底层类型，则这两个指针值可以间接相互转换为对方的类型。

例如：

```
 1| package main
 2|
 3| type MyInt int64
 4| type Ta    *int64
 5| type Tb    *MyInt
 6|
 7| func main() {
 8|    var a Ta
 9|    var b Tb
10|
11|    //a = Ta(b) // error: 直接转换是不允许的。
12|
13|    // 但是间接转换是允许的。
14|    y := (*MyInt)(b)
15|    x := (*int64)(y)
16|    a = x           // 等价于下一行
17|    a = (*int64)(y) // 等价于下一行
18|    a = (*int64)((*MyInt)(b))
```

```
19|     _ = a
20| }
```

两个零尺寸值的地址可能相等，也可能不相等。

两个零尺寸值的地址是否相等时依赖于具体编译器实现以及具体编译器版本。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     a := struct{}{}
 7|     b := struct{}{}
 8|     x := struct{}{}
 9|     y := struct{}{}
10|     m := [10]struct{}{}
11|     n := [10]struct{}{}
12|     o := [10]struct{}{}
13|     p := [10]struct{}{}
14|
15|     fmt.Println(&x, &y, &o, &p)
16|
17|     // 对于标准编译器1.23版本，x、y、o和p将
18|     // 逃逸到堆上，但是a、b、m和n则开辟在栈上。
19|
20|     fmt.Println(&a == &b) // false
21|     fmt.Println(&x == &y) // true
22|     fmt.Println(&a == &x) // false
23|
24|     fmt.Println(&m == &n) // false
25|     fmt.Println(&o == &p) // true
26|     fmt.Println(&n == &p) // false
27| }
```

上面代码中所示的输出是针对标准编译器1.23版本的。

一个指针类型的基类型可以为此指针类型自身。

例如：

```
 1| package main
 2|
```

```
 3| func main() {
 4|    type P *P
 5|    var p P
 6|    p = &p
 7|    p = **************p
 8| }
```

类似的，

- 一个切片类型的元素类型可以是此切片类型自身，
- 一个映射类型的元素类型可以是此映射类型自身，
- 一个通道类型的元素类型可以是此通道类型自身，
- 一个函数类型的输入参数和返回结果值类型可以是此函数类型自身。

```
 1| package main
 2|
 3| func main() {
 4|    type S []S
 5|    type M map[string]M
 6|    type C chan C
 7|    type F func(F) F
 8|
 9|    s := S{0:nil}
10|    s[0] = s
11|    m := M{"Go": nil}
12|    m["Go"] = m
13|    c := make(C, 3)
14|    c <- c; c <- c; c <- c
15|    var f F
16|    f = func(F)F {return f}
17|
18|    _ = s[0][0][0][0][0][0][0][0]
19|    _ = m["Go"]["Go"]["Go"]["Go"]
20|    <-<-<-c
21|    f(f(f(f(f))))
22| }
```

有关选择器缩写形式的细节。

无论一个指针值的类型是具名的还是无名的，如果它的（指针）类型的基类型为一个结构体类型，则我们可以使用此指针值来选择它所引用着的结构体中的字段。 但是，如果此指针的类型为一个具名类型，则我们不能使用此指针值来选择它所引用着的结构体中的方法。

我们总是不能使用二级以上指针来选择结构体字段和方法。

```
 1| package main
 2|
 3| type T struct {
 4|     x int
 5| }
 6| func (T) m(){} // T有一个方法m。
 7|
 8| type P *T  // P为一个具名一级指针类型。
 9| type PP *P // PP为一个具名二级指针类型。
10|
11| func main() {
12|     var t T
13|     var tp = &t
14|     var tpp = &tp
15|     var p P = tp
16|     var pp PP = &p
17|     tp.x = 12  // 没问题
18|     p.x = 34   // 没问题
19|     pp.x = 56  // error: 类型PP没有名为x的字段或者方法。
20|     tpp.x = 78 // error: 类型**T没有名为x的字段或者方法。
21|
22|     tp.m()  // 没问题，因为类型*T也有一个m方法。
23|     p.m()   // error: 类型P没有名为m的字段或者方法。
24|     pp.m()  // error: 类型PP没有名为m的字段或者方法。
25|     tpp.m() // error: 类型**T没有名为m的字段或者方法。
26| }
```

有时候，嵌套组合字面量可以被简化。

关于细节，请阅读内嵌组合字面量可以被简化（第18章）这一章节。

在某些情形下，我们可以将数组指针当作数组来用。

关于细节，请阅读把数组指针当做数组来使用（第18章）这一章节。

从nil映射中读取元素不会导致崩溃，读取结果是一个零元素值。

例如，函数Foo1和Foo2是等价的，但是函数Foo2比函数Foo1简洁得多。

```
 1| func Foo1(m map[string]int) int {
 2|    if m != nil {
 3|       return m["foo"]
 4|    }
 5|    return 0
 6| }
 7|
 8| func Foo2(m map[string]int) int {
 9|    return m["foo"]
10| }
```

从一个nil映射中删除一个条目不会导致崩溃，这是一个空操作。

例如，下面这个程序不会因为恐慌而崩溃。

```
 1| package main
 2|
 3| func main() {
 4|    var m map[string]int // nil
 5|    delete(m, "foo")
 6| }
```

append函数调用的结果可能会与原始切片共享一些元素，也可能不共享任何元素。

关于细节，请阅读添加和删除容器元素（第18章）这一章节。

从一个基础切片派生出的子切片的长度可能大于基础切片的长度。

例如：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    s := make([]int, 3, 9)
```

```
 7|    fmt.Println(len(s)) // 3
 8|    s2 := s[2:7]
 9|    fmt.Println(len(s2)) // 5
10| }
```

关于细节，请阅读从数组或者切片派生切片（第18章）这一章节。

从一个nil切片中派生子切片是允许的，只要子切片表达式中使用的所有索引都为零，则不会有恐慌产生，结果子切片同样是一个nil切片。

例如，下面的程序在运行时刻不会产生恐慌。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var x []int // nil
 7|    a := x[:]
 8|    b := x[0:0]
 9|    c := x[:0:0]
10|    // 下一行将打印出三个true。
11|    fmt.Println(a == nil, b == nil, c == nil)
12| }
```

关于细节，请阅读从数组或者切片派生切片（第18章）这一章节。

用range遍历nil映射或者nil切片是没问题的，这属于空操作。

例如，下面的程序可以编译是没问题的。

```
 1| package main
 2|
 3| func main() {
 4|    var s []int // nil
 5|    for range s {
 6|    }
 7|
 8|    var m map[string]int // nil
 9|    for range m {
```

```
10|    }
11| }
```

用range遍历nil数组指针时，如果忽略或省略第二个迭代变量，则此遍历是没问题的。遍历中的循环步数为相应数组类型的长度。

例如，下面的程序会输出`01234`。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    var a *[5]int // nil
 7|    for i, _ := range a {
 8|       fmt.Print(i)
 9|    }
10| }
```

切片的长度和容量可以被单独修改。

我们可以通过反射途径单独修改一个切片的长度或者容量。 关于细节，请阅读单独修改一个切片的长度或者容量（第18章）这一章节。

切片和数组组合字面量中的索引必须是非负常量。

例如，下面的程序将编译失败。

```
1| var k = 1
2| var x = [2]int{k: 1} // error: 索引必须为一个常量
3| var y = []int{k: 1}  // error: 索引必须为一个常量
```

注意，映射组合字面量中的键值不必为常量。

切片/数组/映射组合字面量的常量索引和键值不能重复。

例如，下面的程序将编译失败。

```
// error: 重复的索引：1
var a = []bool{0: false, 1: true, 1: true}
// error: 重复的索引：0
var b = [...]string{0: "foo", 1: "bar", 0: "foo"}
// error: 重复的键值："foo"
var c = map[string]int{"foo": 1, "foo": 2}
```

这个特性可以用于在编译时刻断言某些条件（第52章）。

不可寻址的数组的元素依旧是不可寻址的，但是不可寻址的切片的元素总是可寻址的。

原因是一个数组值的元素和此数组存储在同一个内存块中。 但是切片的情况大不相同（第51章）。

一个例子：

```
package main

func main() {
    // 组合字面量是不可寻址的。

    /* 取容器元素的地址。 */

    // 取不可寻址的切片的元素的地址是没问题的
    _ = &[]int{1}[0]
    // error: 不能取不可寻址的数组的元素的地址
    _ = &[5]int{}[0]

    /* 修改元素值。 */

    // 修改不可寻址的切片的元素是没问题的
    []int{1,2,3}[1] = 9
    // error: 不能修改不可寻址的数组的元素
    [3]int{1,2,3}[1] = 9
}
```

可以从不可寻址的切片派生子切片，但是不能从不可寻址的数组派生子切片。

原因和上一个细节是一样的。

例如：

```
 1| package main
 2|
 3| func main() {
 4|     // 映射元素是不可寻址的。
 5|
 6|     // 下面几行编译没问题。
 7|     _ = []int{6, 7, 8, 9}[1:3]
 8|     var ms = map[string][]int{"abc": {0, 1, 2, 3}}
 9|     _ = ms["abc"][1:3]
10|
11|     // 下面几行将编译失败，因为不可从不可寻址的数组派生切片。
12|     /*
13|     _ = [...]int{6, 7, 8, 9}[1:3] // error
14|     var ma = map[string][4]int{"abc": {0, 1, 2, 3}}
15|     _ = ma["abc"][1:3]  // error
16|     */
17| }
```

把以NaN做为键值的条目放如映射就宛如把条目放入黑洞一样。

原因是下面的另一个细节（#nan-inf）中提到的`NaN != NaN`。 但是，在Go1.12之前，以`NaN`作为键值的元素只能在`for-range`循环中被找到； 从Go1.12开始，以`NaN`作为键值的元素也可以通过类似`fmt.Print`的函数打印出来。

```
 1| package main
 2|
 3| import "fmt"
 4| import "math"
 5|
 6| func main() {
 7|     var a = math.NaN()
 8|     fmt.Println(a) // NaN
 9|
10|     var m = map[float64]int{}
11|     m[a] = 123
12|     v, present := m[a]
13|     fmt.Println(v, present) // 0 false
14|     m[a] = 789
```

```
15|    v, present = m[a]
16|    fmt.Println(v, present) // 0 false
17|
18|    fmt.Println(m) // map[NaN:789 NaN:123]
19|    delete(m, a)   // no-op
20|    fmt.Println(m) // map[NaN:789 NaN:123]
21|
22|    for k, v := range m {
23|       fmt.Println(k, v)
24|    }
25|    // the above loop outputs:
26|    // NaN 123
27|    // NaN 789
28| }
```

注意：在Go1.12之前，两个`fmt.Println(m)`调用均打印出`map[NaN:<nil> NaN:<nil>]`。

字符串转换为byte切片或rune切片后的结果切片的容量可能会大于长度。

我们不应该假设结果切片的长度和容量总是相等的。

在下面的例子中，如果最后一个`fmt.Println`行被删除，在其前面的两行会打印相同的值`5`；否则，一个打印`5`，一个打印`8`（对于标准编译器1.23版本来说）。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    s := "abcde"
 7|    x := []byte(s)              // len(s) == 1
 8|    fmt.Println(cap([]byte(s))) // 32
 9|    fmt.Println(cap(x))         // 8
10|    fmt.Println(x)
11| }
```

如果我们假设结果切片的长度和容量总是相等，就可能写出一些有bug的代码（https://github.com/go101/go101/wiki/The-capacity-of-the-result-byte-%28or-rune%29-slice-of-a-conversion-from-a-string-is-undefined）。

对于切片s，循环for i = range s {...}并不等价于循环for i = 0; i < len(s); i++ {...}。

对于这两个循环，迭代变量i的最终值可能是不同的。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| var i int
 6|
 7| func fa(s []int, n int) int {
 8|    i = n
 9|    for i = 0; i < len(s); i++ {}
10|    return i
11| }
12|
13| func fb(s []int, n int) int {
14|    i = n
15|    for i = range s {}
16|    return i
17| }
18|
19| func main() {
20|    s := []int{2, 3, 5, 7, 11, 13}
21|    fmt.Println(fa(s, -1), fb(s, -1)) // 6 5
22|    s = nil
23|    fmt.Println(fa(s, -1), fb(s, -1)) // 0 -1
24| }
```

一个映射中的条目的遍历次序在两次遍历中可能并不相同。我们可以认为映射中的条目的遍历次序是随机的。

比如下面这个例子不会无穷尽地循环下去（注意每次退出前的循环次数可能不同）：

```
 1| package main
 2|
 3| import "fmt"
```

```
 4|
 5| func f(m map[byte]byte) string {
 6|    bs := make([]byte, 0, 2*len(m))
 7|    for k, v := range m {
 8|       bs = append(bs, k, v)
 9|    }
10|    return string(bs)
11| }
12|
13| func main() {
14|    m := map[byte]byte{'a':'A', 'b':'B', 'c':'C'}
15|    s0 := f(m)
16|    for i := 1; ; i++{
17|       if s := f(m); s != s0 {
18|          fmt.Println(s0)
19|          fmt.Println(s)
20|          fmt.Println(i)
21|          return
22|       }
23|    }
24| }
```

注意：对映射进行JSON格式化输出中的映射条目是按照它们的键值排序的。 另外，从Go 1.12开始，使用`fmt`标准库包中的打印函数打印映射时，输出的映射条目也是按照它们的键值排序的； 而在Go 1.12之前，这些打印输出时乱序的。

在对一个映射进行条目遍历期间，在此映射中创建的新条目可能会在当前遍历中被遍历出来，也可能不会。

有例为证：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    m := map[int]int{0: 0, 1: 100, 2: 200}
 7|    r, n, i:= len(m), len(m), 0
 8|    for range m {
 9|       m[n] = n*100
10|       n++
11|       i++
```

```
12|     }
13|     fmt.Printf("新增了%d个条目，其中%d个被遍历出来，%d个没有。\n",
14|         i, i - r, n - i,
15|     )
16| }
```

感谢Valentin Deleplace提出了上面两条细节建议（https://twitter.com/val_deleplace/status/1358784287399698434）。

一个多返回值函数调用表达式不能和其它表达式混用在一个赋值语句的右侧或者另一个函数调用的实参列表中。

关于细节，请阅读有返回值的函数的调用是一种表达式（第20章）这一章节。

某些函数调用是在编译时刻被估值的。

关于细节，请阅读哪些函数调用将在编译时刻被估值？（第46章）这一总结。

每一个方法都对应着一个隐式声明的函数。

关于细节，请阅读每个方法对应着一个隐式声明的函数（第22章）这一章节。

如果两个接口值具有相同的动态类型并且此动态类型不支持比较，则比较这两个接口值将导致一个恐慌。

例如：

```
1| package main
2|
3| func main() {
4|     var x interface{} = []int{}
5|     _ = x == x // panic
6| }
```

类型断言可以用于将一个接口值转换为另一个接口类型，即使此接口值的类型并未实现另一个接

口类型。

例如：

```
 1| package main
 2|
 3| type Foo interface {
 4|    foo()
 5| }
 6|
 7| type T int
 8| func (T) foo() {}
 9|
10| func main() {
11|    var x interface{} = T(123)
12|    // 下面这两行将编译失败。
13|    /*
14|    var _ Foo = x   // error: interface{}类型没有实现Foo类型
15|    var _ = Foo(x)  // error: interface{}类型没有实现Foo类型
16|    */
17|    // 但是下面这行可以编译通过。
18|    var _ = x.(Foo) // okay
19| }
```

一个失败的类型断言的可选的第二个结果是否被舍弃将影响此类型断言的行为。

如果第二个可选结果出现在失败的类型断言中，那么此类型断言不会导致恐慌。否则，恐慌将产生。 例如：

```
 1| package main
 2|
 3| func main() {
 4|    var x interface{} = true
 5|    _, _ = x.(int) // 断言失败，但不会导致恐慌。
 6|    _ = x.(int)    // 断言失败，并导致一个恐慌。
 7| }
```

关于在编译时刻即可确定总是失败的目标类型为接口类型的断言。

在编译时刻，编译可以发现某些目标类型为接口类型的断言是不可能成功的。比如下面这个程序中的断言：

```
 1| package main
 2|
 3| type Ia interface {
 4|    m()
 5| }
 6|
 7| type Ib interface {
 8|    m() int
 9| }
10|
11| type T struct{}
12|
13| func (T) m() {}
14|
15| func main() {
16|    var x Ia = T{}
17|    _ = x.(Ib) // panic: main.T is not main.Ib
18| }
```

这样的断言并不会导致编译失败（但编译后的程序将在运行时刻产生恐慌）。 从Go官方工具链1.15开始，`go vet`会对对这样的断言做出警告。

以相同实参调用两次errors.New函数返回的两个error值是不相等的。

原因是`errors.New`函数会复制输入的字符串实参至一个局部变量并取此局部变量的指针作为返回`error`值的动态值。 两次调用会产生两个不同的指针。

```
 1| package main
 2|
 3| import "fmt"
 4| import "errors"
 5|
 6| func main() {
 7|    notfound := "not found"
 8|    a, b := errors.New(notfound), errors.New(notfound)
 9|    fmt.Println(a == b) // false
10| }
```

单向接收通道无法被关闭。

例如，下面的代码会在编译时候失败。

```
 1| package main
 2|
 3| func main() {
 4| }
 5|
 6| func foo(c <-chan int) {
 7|    close(c) // error: 不能关闭单向接收通道
 8| }
```

发送一个值到一个已关闭的通道被视为一个非阻塞操作，该操作会导致恐慌。

例如，在下面的程序里，如果第二个case分支会被选中，则在运行时刻将产生一个恐慌。

```
 1| package main
 2|
 3| func main() {
 4|    var c = make(chan bool)
 5|    close(c)
 6|    select {
 7|    case <-c:
 8|    case c <- true: // panic: 向已关闭的通道发送数据
 9|    default:
10|    }
11| }
```

类型可以在声明函数体内。

类型可以声明在函数体内。例如，

```
 1| package main
 2|
 3| func main() {
 4|    type T struct{}
 5|    type S = []int
 6| }
```

对于标准编译器，结构体中的某些零尺寸字段的尺寸有可能会被视为一个字节。

关于细节，请阅读这个FAQ条目（第51章）。

NaN != NaN, Inf == Inf。

此规则遵循IEEE-754标准，并与大多数其它语言是一致的。

```
 1| package main
 2|
 3| import "fmt"
 4| import "math"
 5|
 6| func main() {
 7|    var a = math.Sqrt(-1.0)
 8|    fmt.Println(a)      // NaN
 9|    fmt.Println(a == a) // false
10|
11|    var x = 0.0
12|    var y = 1.0 / x
13|    var z = 2.0 * y
14|    fmt.Println(y, z, y == z) // +Inf +Inf true
15| }
```

不同代码包中的两个非导出方法名和结构体字段名总是被视为不同的名称。

例如，在包foo中声明了如下的类型：

```
 1| package foo
 2|
 3| type I = interface {
 4|    about() string
 5| }
 6|
 7| type S struct {
 8|    a string
 9| }
10|
11| func (s S) about() string {
```

```
12|    return s.a
13| }
```

在包bar中声明了如下的类型：

```
 1| package bar
 2|
 3| type I = interface {
 4|    about() string
 5| }
 6|
 7| type S struct {
 8|    a string
 9| }
10|
11| func (s S) about() string {
12|    return s.a
13| }
```

那么，

- 两个包中的两个类型S的值不能相互转换。
- 两个包中的两个接口类型指定了两个不同的方法集。
- 类型foo.S没有实现接口类型 bar.I。
- 类型bar.S没有实现接口类型foo.I。

```
 1| package main
 2|
 3| import "包2/foo"
 4| import "包2/bar"
 5|
 6| func main() {
 7|    var x foo.S
 8|    var y bar.S
 9|    var _ foo.I = x
10|    var _ bar.I = y
11|
12|    // 下面这些行将编译失败。
13|    x = foo.S(y)
14|    y = bar.S(x)
15|    var _ foo.I = y
16|    var _ bar.I = x
17| }
```

在结构体值的比较中，名为空标识符的字段将被忽略。

比如，下面这个程序将打印出true。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type T struct {
 6|     _ int
 7|     _ bool
 8| }
 9|
10| func main() {
11|     var t1 = T{123, true}
12|     var t2 = T{789, false}
13|     fmt.Println(t1 == t2) // true
14| }
```

在某些很少见的场景中，圆括号是必需的。

例如：

```
 1| package main
 2|
 3| type T struct{x, y int}
 4|
 5| func main() {
 6|     // 因为{}的烦扰，下面这三行均编译失败。
 7|     /*
 8|     if T{} == T{123, 789} {}
 9|     if T{} == (T{123, 789}) {}
10|     if (T{}) == T{123, 789} {}
11|     var _ = func()(nil) // nil被认为是一个类型
12|     */
13|
14|     // 必须加上一对小括号()才能编译通过。
15|     if (T{} == T{123, 789}) {}
16|     if (T{}) == (T{123, 789}) {}
17|     var _ = (func())(nil) // nil被认为是一个值
18| }
```

栈溢出不可被挽救，它将使程序崩溃。

在目前的主流Go编译器实现中，栈溢出是致命错误。一旦栈溢出发生，程序将不可恢复地崩溃。

```
 1| package main
 2|
 3| func f() {
 4|     f()
 5| }
 6|
 7| func main() {
 8|     defer func() {
 9|         recover() // 无法防止程序崩溃
10|     }()
11|     f()
12| }
```

运行结果：

```
runtime: goroutine stack exceeds 1000000000-byte limit
fatal error: stack overflow

runtime stack:
...
```

关于更多不可恢复的致命错误，请参考[此篇维基文章](https://github.com/go101/go101/wiki/Panic-and-crash-cases)（https://github.com/go101/go101/wiki/Panic-and-crash-cases）。

某些表达式的估值顺序取决于具体编译器实现。

关于细节，请阅读[表达式估值顺序规则](#)（第33章）一文。

`reflect.DeepEqual(x, y)`和`x == y`的结果可能会不同。

如果表达式`x`和`y`的类型不相同，则函数调用`DeepEqual(x, y)`的结果总为`false`，但`x == y`的估值结果有可能为`true`。

如果`x`和`y`为（同类型的）两个引用着不同其它值的指针值，则`x == y`的估值结果总为`false`，但函数调用`DeepEqual(x, y)`的结果可能为`true`，因为函数

reflect.DeepEqual将比较x和y所引用的两个值。

第三个区别是当x和y均处于某个循环引用链中时，为了防止死循环，DeepEqual调用的结果可能为true。

第四个区别是一个DeepEqual(x, y)调用无论如何不应该产生一个恐慌，但是如果x和y是两个动态类型相同的接口值并且它们的动态类型是不可比较类型的时候，x == y将产生一个恐慌。

一个展示了这些不同的例子：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "reflect"
 6| )
 7|
 8| func main() {
 9|    type Book struct {page int}
10|    x := struct {page int}{123}
11|    y := Book{123}
12|    fmt.Println(reflect.DeepEqual(x, y)) // false
13|    fmt.Println(x == y)                  // true
14|
15|    z := Book{123}
16|    fmt.Println(reflect.DeepEqual(&z, &y)) // true
17|    fmt.Println(&z == &y)                  // false
18|
19|    type Node struct{peer *Node}
20|    var q, r, s Node
21|    q.peer = &q // 形成一个循环引用链
22|    r.peer = &s // 形成一个循环引用链
23|    s.peer = &r
24|    println(reflect.DeepEqual(&q, &r)) // true
25|    fmt.Println(q == r)                // false
26|
27|    var f1, f2 func() = nil, func(){}
28|    fmt.Println(reflect.DeepEqual(f1, f1)) // true
29|    fmt.Println(reflect.DeepEqual(f2, f2)) // false
30|
31|    var a, b interface{} = []int{1, 2}, []int{1, 2}
32|    fmt.Println(reflect.DeepEqual(a, b)) // true
33|    fmt.Println(a == b)                  // 产生恐慌
34| }
```

注意：如果传递给一个DeepEqual调用的两个实参均为函数类型值，则此调用只有在这两个实参都为nil并且它们的类型相同的情况下才返回true。 比较元素中含有函数值的容器值或者比较字段中含有函数值的结构体值也是类似的。 另外要注意：如果两个同类型切片共享相同的元素序列（即它们的长度相同并且它们的各对相应元素的地址也相同），则使用DeepEqual比较它们时返回的结果总是为true，即使它们的元素中含有函数值。 一个例子：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "reflect"
 6| )
 7|
 8| func main() {
 9|     a := [1]func(){func(){}}
10|     b := a
11|     fmt.Println(reflect.DeepEqual(a, a))       // false
12|     fmt.Println(reflect.DeepEqual(a[:], a[:])) // true
13|     fmt.Println(reflect.DeepEqual(a[:], b[:])) // false
14|     a[0], b[0] = nil, nil
15|     fmt.Println(reflect.DeepEqual(a[:], b[:])) // true
16| }
```

reflect.Value.Bytes()方法返回一个[]byte值，它的元素类型byte可能并非属主参数代表的Go切片值的元素类型。

假设一个自定义类型MyByte的底层类型为内置类型byte，我们知道Go类型系统禁止切片类型[]MyByte的值转换为类型[]byte。 但是，当前的reflect.Value类型的Bytes方法的实现可以帮我们绕过这个限制。 此实现应该是违反了Go类型系统的规则。

例子：

```
 1| package main
 2|
 3| import "bytes"
 4| import "fmt"
 5| import "reflect"
 6|
 7| type MyByte byte
```

```
 8|
 9| func main() {
10|    var mybs = []MyByte{'a', 'b', 'c'}
11|    var bs []byte
12|
13|    // bs = []byte(mybs) // this line fails to compile
14|
15|    v := reflect.ValueOf(mybs)
16|    bs = v.Bytes() // okay. Violating Go type system.
17|    fmt.Println(bytes.HasPrefix(bs, []byte{'a', 'b'})) //
true
18|
19|    bs[1], bs[2] = 'r', 't'
20|    fmt.Printf("%s \n", mybs) // art
21| }
```

虽然这违反了Go类型系统的规则，但是貌似此违反并没有什么害处，相反，它带来了一些好处。 比如，我们可以将`bytes`标准库包中提供的函数（间接）应用到`[]MyByte`值上，如上例所示。

注意：`reflect.Value.Bytes()`方法以后可能会被移除（https://github.com/golang/go/issues/27727）。

我们应该使用`os.IsNotExist(err)`而不是`err == os.ErrNotExist`来检查文件是否存在。

使用`err == os.ErrNotExist`可能漏掉一些错误。

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "os"
 6| )
 7|
 8| func main() {
 9|    _, err := os.Stat("a-nonexistent-file.abcxyz")
10|    fmt.Println(os.IsNotExist(err))    // true
11|    fmt.Println(err == os.ErrNotExist) // false
12| }
```

如果你的项目只支持Go 1.13+，则更推荐（https://github.com/golang/go/issues/38198）使用`errors.Is(err,`

os.ErrNotExist)来检查文件是否存在。

```
 1| package main
 2|
 3| import (
 4|    "errors"
 5|    "fmt"
 6|    "os"
 7| )
 8|
 9| func main() {
10|    _, err := os.Stat("a-nonexistent-file.abcxyz")
11|    fmt.Println(errors.Is(err, os.ErrNotExist)) // true
12| }
```

flag标准库包对待布尔命令选项不同于数值和字符串选项。

传递程序选项有三种形式。

1. -flag：仅适用于布尔选项。
2. -flag=x：用于任何类型的选项。.
3. -flag x：仅用于非布尔选项。

请注意，使用第一种形式的布尔选项将被视为最后一个选项，其后面的所有项都被视为参数。

```
 1| package main
 2|
 3| import "fmt"
 4| import "flag"
 5|
 6| var b = flag.Bool("b", true, "一个布尔选项")
 7| var i = flag.Int("i", 123, "一个整数选项")
 8| var s = flag.String("s", "hi", "一个字符串选项")
 9|
10| func main() {
11|    flag.Parse()
12|    fmt.Print("b=", *b, ", i=", *i, ", s=", *s, "\n")
13|    fmt.Println("arguments:", flag.Args())
14| }
```

如果我们用下面显示的标志和参数运行此程序

```
./exampleProgram -b false -i 789 -s bye arg0 arg1
```

输出结果会是：

```
b=true, i=123, s=hi
arguments: [false -i 789 -s bye arg0 arg1]
```

这个输出显然不是我们所期望的。

我们应该像这样传递选项和参数：

```
./exampleProgram -b=false -i 789 -s bye arg0 arg1
```

或者

```
./exampleProgram -i 789 -s bye -b arg0 arg1
```

以获取我们期望的输出：

```
b=true, i=789, s=bye
arguments: [arg0 arg1]
```

[Sp|Fp|P]rintf函数支持位置参数。

下面的程序会打印coco。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     // The next line prints: coco
 7|     fmt.Printf("%[2]v%[1]v%[2]v%[1]v", "o", "c")
 8| }
```

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

Go问答101

（这是一份非官方Go问答列表。官方版问答列表在这里（https://golang.google.cn/doc/faq）。）

索引：

- 编译器与运行时
  - 编译器错误信息non-name *** on left side of :=意味着什么？（#error-non-name）
  - 编译器错误信息unexpected newline, expecting { after if clause意味着什么？（#error-missing-left-brace）
  - 编译器错误信息declared and not used意味着什么？（#error-declared-not-used）
  - Go运行时是否维护映射条目的遍历顺序？（#map-iteration-order）
  - Go编译器是否会进行字节填充以确保结构体字段的地址对齐？（#compiler-field-padding）
  - 为什么一个结构体类型的最后一个字段类型的尺寸为零时会影响此结构体类型的尺寸？（#final-zero-size-field）
  - new(T)是var t T; (&t)的语法糖吗？（#new-function）
  - 运行时错误信息all goroutines are asleep - deadlock意味着什么？（#fatal-error-deadlock）
  - 64位整数值的地址是否能保证总是64位对齐的，以便可以被安全地原子访问？（#64-bit-alignment）
  - 赋值是原子操作吗？（#atomical-assignment）
  - 是否每一个零值在内存中占据的字节都是零？（#zero-values-zero-bytes）
  - 标准的Go编译器是否支持函数内联？（#function-inline）
  - 终结器（finalizer）可以用做对象的析构函数吗？（#finalizers）
- 标准库
  - 如何使用尽可能短的代码行数来获取任意月份的天数？（#get-days-of-month）
  - 函数调用time.Sleep(d)和通道接收<-time.After(d)操作之间有何区别？（#time-sleep-after）
  - 调用strings和bytes标准库包里TrimLeft和TrimRight函数经常会返回不符预期的结果，这些函数的实现存在bugs吗？（#trim）
  - 函数fmt.Print和fmt.Println的区别是什么？（#fmt-print-println）

    - 函数log.Print和函数log.Println有什么区别吗？（#log-print-println）
    - 函数fmt.Print、fmt.Println和fmt.Printf的实现进行同步了吗？（#fmt-print-synced）
    - 内置的print和println函数与fmt和log标准库包中相应的打印函数有什么区别？（#print-builtin-fmt-log）
    - 通过标准库包math/rand和crypto/rand生成的随机数之间有什么区别？（#math-crypto-rand）
    - 标准库中为什么没有math.Round函数？（#math-round）
- 类型系统
    - 哪些类型不支持比较？（#uncomparable-types）
    - 为什么两个nil值有时候会不相等？（#nil-equality）
    - 为什么类型[]T1和[]T2没有共享相同底层类型，即使不同的类型T1和T2共享相同的底层类型？（#slice-conversion）
    - 哪些值可以被取地址，哪些值不可以被取地址？（#unaddressable-values）
    - 为什么映射元素不可被取地址？（#map-elements-are-unaddressable）
    - 为什么非空切片的元素总是可被取地址，即便对于不可取地址的切片也是如此？（#slice-elements-always-addressable）
    - 对任意的非指针、非接口类型T，为什么类型*T的方法集总是类型T的方法集的超集，但是反之却不然？（#method-set-relation）
    - 我们可以为哪些类型声明方法？（#types-can-have-methods）
    - 在Go里如何声明不可变量？（#declare-immutables）
    - 为什么没有内置的set容器类型？（#set-container-type）
    - 什么是byte？什么是rune？如何将[]byte和[]rune的值转换为字符串？（#byte-rune-slice-string）
    - 如何原子地操作指针值？（#pointer-atomic）
- 其它
    - iota是什么意思？（#about-iota）
    - 为什么没有一个内置的closed函数用来检查通道是否已经关闭？（#check-if-channels-closed）
    - 函数返回局部变量的指针是否安全？（#return-local-pointers）
    - 单词*gopher*在Go社区中表示什么？（#gopher）

编译器错误信息non-name *** on left side of :=意味着什么？

直到目前（Go 1.23），Go中对短变量声明有一个强制性约束（https://golang.google.cn/ref/spec#Short_variable_declarations）：

> 所有位于:=符号左侧的条目都必须是纯标识符（https://golang.google.cn/ref/spec#Identifiers），并且其中至少有一个为新变量名称。

这意味着容器元素索引表达式（x[i]）、结构体的字段选择器（x.f）、指针解引用（*p）和限定标识符（aPackage.Value）都不能出现在:=符号的左侧。

目前，这还是一个未解决问题（https://github.com/golang/go/issues/377）（已经和一个相关问题（https://github.com/golang/go/issues/6842）合并）。而且感觉Go核心开发团队目前并未有立即解决此问题的打算（https://github.com/golang/go/issues/30318）。

编译器错误信息unexpected newline, expecting { ...意味着什么?

在编写Go代码时，我们不能随意断行。 请阅读代码断行规则（第28章）一文以了解Go代码断行规则。 一般来说，根据这些规则，在左括号之前断行是不合法的。

例如，下列代码片段

```
 1| if true
 2| {
 3| }
 4|
 5| for i := 0; i < 10; i++
 6| {
 7| }
 8|
 9| var _ = []int
10| {
11|    1, 2, 3
12| }
```

将会被编译器解释成

```
 1| if true;
 2| {
 3| }
 4|
 5| for i := 0; i < 10; i++;
 6| {
 7| }
 8|
```

```
 9| var _ = []int;
10| {
11|     1, 2, 3;
12| }
```

Go编译器将为每个左大括号{起始的代码行报告一个语法错误。 为避免这些报错，我们需要将上述代码重写为下面这样：

```
 1| if true {
 2| }
 3|
 4| for i := 0; i < 10; i++ {
 5| }
 6|
 7| var _ = []int {
 8|     1, 2, 3,
 9| }
```

编译器错误信息declared and not used意味着什么？

对于标准编译器，在局部代码块中声明的每一个变量必须被至少一次用做r-value（right-hand-side value，右值）。

因此，下列代码将编译失败，因为y只被用做目标值（目标值都为左值）。

```
1| func f(x bool) {
2|     var y = 1 // y被声明了但没有被用做右值
3|     if x {
4|         y = 2 // 这里，y被用做左值
5|     }
6| }
```

Go运行时是否维护映射条目的遍历顺序？

不。Go白皮书（https://golang.google.cn/ref/spec#For_range）明确提到映射元素的迭代顺序是未定义的。 所以对于同一个映射值，它的一个遍历过程和下一个遍历过程中的元素呈现次序不保证是相同的。 对于标准编译器，映射元素的遍历顺序是随机的。 如果你需要固定的映射元素遍历顺序，那么你就需要自己来维护这个顺序。 更多信息请阅读Go官方博客文章Go maps in action（https://blog.golang.com/go-maps-in-action#TOC_7.）。

但是请注意：从Go 1.12开始，标准库包中的各个打印函数的结果中，映射条目总是排了序的。

Go编译器是否会进行字节填充以确保结构体字段的地址对齐?

至少对于标准的Go编译器和gccgo，答案是肯定的。 具体需要填充多少个字节取决于操作系统和编译器实现。 请阅读关于Go值的内存布局（第44章）一文获取详情。

Go编译器将不会重新排列结构体的字段来最小化结构体值的尺寸。 因为这样做会导致意想不到的结果。 但是，根据需要，程序员可以手工重新排序字段来实现填充最小化。

为什么一个结构体类型的最后一个字段类型的尺寸为零时会影响此结构体的尺寸?

一个可寻址的结构值的所有字段都可以被取地址。 如果非零尺寸的结构体值的最后一个字段的尺寸是零，那么取此最后一个字段的地址将会返回一个越出了为此结构体值分配的内存块的地址。 这个返回的地址可能指向另一个被分配的内存块。 在目前的官方Go标准运行时的实现中，如果一个内存块被至少一个依然活跃的指针引用，那么这个内存块将不会被视作垃圾因而肯定不会被回收。 所以只要有一个活跃的指针存储着此非零尺寸的结构体值的最后一个字段的越界地址，它将阻止垃圾收集器回收另一个内存块，从而可能导致内存泄漏。

为避免上述问题，标准的Go编译器会确保取一个非零尺寸的结构体值的最后一个字段的地址时，绝对不会返回越出分配给此结构体值的内存块的地址。 Go标准编译器通过在需要时在结构体最后的零尺寸字段之后填充一些字节来实现这一点。

如果一个结构体的全部字段的类型都是零尺寸的(因此整个结构体也是零尺寸的)，那么就不需要再填充字节，因为标准编译器会专门处理零尺寸的内存块。

一个例子：

```
1| package main
2|
3| import (
4|     "unsafe"
5|     "fmt"
6| )
7|
8| func main() {
9|     type T1 struct {
```

```
10|       a struct{}
11|       x int64
12|    }
13|    fmt.Println(unsafe.Sizeof(T1{})) // 8
14|
15|    type T2 struct {
16|       x int64
17|       a struct{}
18|    }
19|    fmt.Println(unsafe.Sizeof(T2{})) // 16
20| }
```

另请见官方FAQ中的此问答（https://golang.google.cn/doc/faq#zero_size_types）。

new(T)是var t T; (&t)的语法糖吗?

虽然这两者在实现上会有一些微妙的差别，取决于编译器的具体实现，但是我们基本上可以认为这两者是等价的。 即，通过new函数分配的内存块可以在栈上，也可以在堆上。

运行时错误信息all goroutines are asleep - deadlock意味着什么?

用词*asleep*在这里其实并不准确，实际上它的意思是***处于阻塞状态***。

因为一个处于阻塞状态的协程只能被另一个协程解除阻塞，如果程序中所有的协程都进入了阻塞状态，则它们将永远都处于阻塞状态。 这意味着程序死锁了。一个正常运行的程序永远不应该死锁，一个死锁的程序肯定是由于逻辑实现上的bug造成的。 因此官方Go标准运行时将在一个程序死锁时令其崩溃退出。

64位整数值的地址是否能保证总是64位对齐的，以便可以被安全地原子访问?

传递给sync/atomic标准库包中的64位函数的地址必须是64位对齐的，否则调用这些函数将在运行时导致恐慌产生。

对于标准编译器和gccgo编译器，在64位架构下，64位整数的地址将保证总是64位对齐的。 所以它们总是可以被安全地原子访问。 但在32位架构下，64位整数的地址仅保证是32位对齐的。 所以原子访问某些64位整数可能会导致恐慌。 但是，有一

些方法可以保证一些64位整数总是可以被安全地原子访问。 请阅读关于Go值的内存布局（第44章）一文以获得详情。

赋值是原子操作吗?

对于标准编译器来说，赋值不是原子操作。

请阅读官方FAQ中的此问答（https://golang.google.cn/doc/faq#What_operations_are_atomic_What_about_mutexes）以了解更多。

是否每一个零值在内存中占据的字节都是零?

对于大部分类型，答案是肯定的。不过事实上，这依赖于编译器。 例如，对于标准编译器，对于字符串或者浮点数类型的某些零值，此结论并不十分正确。

比如：

```
 1| package main
 2|
 3| import (
 4|     "unsafe"
 5|     "fmt"
 6| )
 7|
 8| func main() {
 9|     // case 1:
10|     var s = "abc"[0:0]
11|     fmt.Println(s == "") // true
12|     var addr = *(*uintptr)(unsafe.Pointer(&s))
13|     fmt.Println(addr) // <a-non-zero-value>
14|
15|     // case 2:
16|     var x = 0.0
17|     var y = -x
18|     fmt.Println(y == 0) // true
19|     var n = *(*uintptr)(unsafe.Pointer(&y))
20|     fmt.Println(n) // 9223372036854775808
21| }
```

反过来，对于标准编译器已经支持的所有架构，如果一个值的所有字节都是零，那么这个值肯定是它的类型的零值。 然而，Go规范并没有保证这一点。我曾听说在某些比较老的处理器上，空指针表示的内存地址并不为零。

标准的Go编译器是否支持函数内联?

是的，标准编译器支持函数内联。编译器会自动内联一些满足某些条件的短小函数。这些内联条件可能会在不同编译器版本之间发生变化。

目前（Go 1.23），对于标准编译器，

- 没有显式的方式来在用户代码中指定哪些函数应该被内联。
- 尽管编译参数`-gcflags "-l"`可以阻止任何函数被内联， 但是并没有一个正式的方式来避免某个特定的用户函数被内联。 目前我们可以在函数声明前增加一行`//go:noinline`指令来避免这个函数被内联。 但是此方式不保证永久有效。

终结器（finalizer）可以用做对象的析构函数吗?

在Go程序里，我们可以通过调用`runtime.SetFinalizer`函数来给一个对象设置一个终结器函数。 一般说来，此终结器函数将在此对象被垃圾回收之前调用。 但是终结器并非被设计为对象的析构函数。 通过`runtime.SetFinalizer`函数设置的终结器函数并不保证总会被运行。 因此我们不应该依赖于终结器来保证程序的正确性。

终结器的主要用途是为了库包的维护者能够尽可能地避免因为库包使用者不正确地使用库包而带来的危害。 例如，我们知道，当在程序中使用完某个文件后，我们应该将其关闭。 但是有时候因为种种原因，比如经验不足或者粗心大意，导致一些文件在使用完成后并未被关闭，那么和这些文件相关的很多资源只有在此程序退出之后才能得到释放。这属于资源泄漏。 为了尽可能地避免防止资源泄露，`os`库包的维护者将会在一个`os.File`对象被被创建的时候为之设置一个终结器。 此终结器函数将关闭此`os.File`对象。当此`os.File`对象因为不再被使用而被垃圾回收的时候，此终结器函数将被调用。

请记住，有一些终结器函数永远不会被调用，并且有时候不当的设置终结器函数将会阻止对象被垃圾回收。 关于更多细节，请阅读runtime.SetFinalizer函数的文档（https://golang.google.cn/pkg/runtime/#SetFinalizer）。

如何使用尽可能短的代码行数来获取任意月份的天数?

假设输入的年份是一个自然年，并且输入的月份也是一个自然月（1代表1月）。

```
 1| days := time.Date(year, month+1, 0, 0, 0, 0, 0,
time.UTC).Day()
```

对于Go中的`time`标准库包，正常月份的去值范围为`[1, 12]`，并且每个月的起始日是`1`。 所以，`y`年的`m`月的起始时间就是`time.Date(y, m, 1, 0, 0, 0, 0, time.UTC)`。

传递给`time.Date`函数的实参可以超出它们的正常范围，此函数将这些实参进行规范化。 例如，1月32日会被转换成2月1日。

以下是一些Go语言里的日期使用示例：

```
 1| package main
 2|
 3| import (
 4|    "time"
 5|    "fmt"
 6| )
 7|
 8| func main() {
 9|    // 2017-02-01 00:00:00 +0000 UTC
10|    fmt.Println(time.Date(2017, 1, 32, 0, 0, 0, 0,
time.UTC))
11|
12|    // 2017-01-31 23:59:59.999999999 +0000 UTC
13|    fmt.Println(time.Date(2017, 1, 32, 0, 0, 0, -1,
time.UTC))
14|
15|    // 2017-01-31 00:00:00 +0000 UTC
16|    fmt.Println(time.Date(2017, 2, 0, 0, 0, 0, 0, time.UTC))
17|
18|    // 2016-12-31 00:00:00 +0000 UTC
19|    fmt.Println(time.Date(2016, 13, 0, 0, 0, 0, 0,
time.UTC))
20|
21|    // 2017-02-01 00:00:00 +0000 UTC
22|    fmt.Println(time.Date(2016, 13, 32, 0, 0, 0, 0,
time.UTC))
23| }
```

函数调用`time.Sleep(d)`和通道接收`<-time.After(d)`操作之间有何区别?

两者都会将当前的goroutine执行暂停一段时间。 区别在于`time.Sleep(d)`函数调用将使当前的协程进入睡眠子状态，但是当前协程的（主）状态（第13章）依然为运行状态； 而通道接收`<-time.After(d)`操作将使当前协程进入阻塞状态。

调用strings和bytes标准库包里TrimLeft和TrimRight函数经常会返回不符预期的结果，这些函数的实现存在bugs吗?

哈，我们不能保证这些函数的实现绝对没有bug，但是如果这些函数返回的结果是不符你的预期，更有可能的是你的期望是不正确的。

标准包`strings`和`bytes`里有多个修剪（trim）函数。 这些函数可以被分类为两组：

1. `Trim`、`TrimLeft`、`TrimRight`、`TrimSpace`、`TrimFunc`、`TrimLeftFunc`和`TrimRightFunc`。 这些函数将修剪首尾所有满足指定（或隐含）条件的utf-8编码的Unicode码点(即rune)。（`TrimSpace`隐含了修剪各种空格符。） 这些函数将检查每个开头或结尾的rune值，直到遇到一个不满足条件的rune值为止。
2. `TrimPrefix`和`TrimSuffix`。 这两个函数会把指定前缀或后缀的子字符串（或子切片）作为一个整体进行修剪。

部分（https://github.com/golang/go/issues/19172）程序员（https://github.com/golang/go/issues/18160）会（https://github.com/golang/go/issues/14657）把`TrimLeft`和`TrimRight`函数当作`TrimPrefix`和`TrimSuffix`函数而误用（https://github.com/golang/go/issues/19371）。 自然地，函数返回的结果很可能不是预期的那样。

例如：

```
 1| package main
 2|
 3| import (
 4|     "fmt"
 5|     "strings"
 6| )
 7|
 8| func main() {
 9|     var s = "abaay森z众xbbab"
10|     o := fmt.Println
11|     o(strings.TrimPrefix(s, "ab")) // aay森z众xbbab
```

```
12|    o(strings.TrimSuffix(s, "ab")) // abaay森z众xbb
13|    o(strings.TrimLeft(s, "ab"))   // y森z众xbbab
14|    o(strings.TrimRight(s, "ab"))  // abaay森z众x
15|    o(strings.Trim(s, "ab"))       // y森z众x
16|    o(strings.TrimFunc(s, func(r rune) bool {
17|       return r < 128 // trim all ascii chars
18|    })) // 森z众
19| }
```

函数fmt.Print和fmt.Println 的区别是什么?

fmt.Println函数总会在两个相邻的参数之间输出一个空格，然而fmt.Print函数仅当两个相邻的参数（的具体值）都不是字符串类型时才会在它们之间输出一个空格。

另外一个区别是fmt.Println函数会在结尾写入一个换行符，但是fmt.Print函数不会。

函数log.Print 和函数 log.Println 有什么区别吗?

函数log.Print与log.Println的区别与上一个问题里描述的关于函数fmt.Print和fmt.Println的第一个区别点类似。

这两个函数都会在结尾输出一个换行符。

函数fmt.Print、fmt.Println和fmt.Printf的实现进行同步了吗?

没有。 如果有同步的需求，请使用log标准库包里的相应函数。 你可以调用log.SetFlags(0)来避免每一个日志行的前缀输出。

内置的print和println函数与fmt和log标准库包中相应的打印函数有什么区别?

除了上一个问题里提到的区别之外，这三组函数之间还有一些其他区别。

1. 内置的print/println函数总是写入标准错误。 fmt标准包里的打印函数总是写入标准输出。 log标准包里的打印函数会默认写入标准错误，然而也可以通过log.SetOutput函数来配置。
2. 内置print/println函数的调用不能接受数组和结构体参数。
3. 对于组合类型的参数，内置的print/println函数将输出参数的底层值部的地址，而fmt和log标准库包中的打印函数将输出接口参数的动态值的字面形式。
4. 调用内置的print/println函数不会使调用参数引用的值逃逸到堆上，而fmt和log标准库包中的打印函数将使调用参数引用的值逃逸到堆上。
5. 如果一个实参有String() string或Error() string方法，那么fmt和log标准库包里的打印函数在打印参数时会调用这两个方法，而内置的print/println函数则会忽略参数的这些方法。
6. 内置的print/println函数不保证在未来的Go版本中继续存在。

标准库包math/rand和crypto/rand生成的随机数之间有什么区别?

通过math/rand标准库包生成的伪随机数序列对于给定的种子是确定的。 这样生成的随机数不适用于安全敏感的环境中。 如果处于加密安全目的，我们应该使用crypto/rand标准库包生成的伪随机数序列。

标准库中为什么没有math.Round函数?

math.Round函数是有的，但是只是从Go 1.10开始才有这个函数。 从Go 1.10开始，标准库添加了两个新函数math.Round和math.RoundToEven。

在Go 1.10之前，关于 math.Round函数是否应该被添加进标准包，经历了很长时候的讨论（https://github.com/golang/go/issues/4594）。

哪些类型不支持比较?

下列类型不支持比较：

- 映射（map）
- 切片
- 函数
- 包含不可比较字段的结构体类型
- 元素类型为不可比较类型的数组类型

不支持比较的类型不能用做映射类型的键值类型。

请注意：

- 尽管映射，切片和函数值不支持比较，但是它们的值可以与类型不确定的`nil`标识符比较。
- 如果两个接口值的动态类型相同且不可比较，那么在运行时比较这两个接口的值（第23章）会产生一个恐慌。

关于为什么映射，切片和函数不支持比较，请阅读Go的官方FAQ中关于这个问答（https://golang.google.cn/doc/faq#map_keys）。

为什么两个nil值有时候会不相等?

（Go官方FAQ中的这个答案（https://golang.google.cn/doc/faq#nil_error）也回答了这个问题。）

一个接口值可以看作是一个包裹非接口值的盒子。被包裹在一个接口值中的非接口值的类型必须实现了此接口值的类型。 在Go中，很多种类型的类型的零值都是用`nil`来表示的。 一个什么都没包裹的接口值为一个零值接口值，即nil接口值。 一个包裹着其它非接口类型的nil值的接口值并非什么都没包裹，所以它不是（或者说它不等于）一个nil接口值。

当对一个nil接口值和一个nil非接口值进行比较时（假设它们可以比较），此nil非接口值将先被转换为nil接口值的类型，然后再进行比较； 此转换的结果为一个包裹了此nil非接口值的一个副本的接口值，此接口值不是（或者说它不等于）一个nil接口值，所以此比较不相等。

关于更详细的解释请阅读接口（第23章）和关于Go中的`nil`（第47章）两篇文章。

一个示例：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     var pi *int = nil
 7|     var pb *bool = nil
 8|     var x interface{} = pi
 9|     var y interface{} = pb
10|     var z interface{} = nil
11|
12|     fmt.Println(x == y)   // false
```

```
13|    fmt.Println(x == nil) // false
14|    fmt.Println(y == nil) // false
15|    fmt.Println(x == z)   // false
16|    fmt.Println(y == z)   // false
17| }
```

为什么类型[]T1和[]T2没有共享相同底层类型，即使不同的类型T1和T2共享相同的底层类型?

（不久前，Go官方FAQ也增加了一个相似的问题（https://golang.google.cn/doc/faq#convert_slice_with_same_underlying_type）。）

在Go语言中，仅当两个切片类型共享相同的底层类型（第14章）时，其中一个切片类型才可以转换成另一个切片的类型而不需要使用unsafe机制（第25章）。

一个无名组合类型的底层类型是此组合类型本身。 所以即便两个不同的类型T1和T2共享相同的底层类型，类型[]T1和[]T2也依然是不同的类型，因此它们的底层类型也是不同的。这意味着其中一个的值不能转换为另一个。

底层类型[]T1和[]T2不同的原因是：

- 把[]T1和[]T2的值相互转换的需求在实践中并不常见。
- 使得底层类型的溯源规则（第14章）更加简单。

同样的原因也适用于其它组合类型。 例如：类型map[T]T1 和 map[T]T2同样不共享相同的底层类型，即便T1 和 T2共享相同的底层类型。

类型[]T1的值时候有可能通过使用unsafe机制转换成[]T2的，但是一般不建议这么做：

```
 1| package main
 2|
 3| import (
 4|    "fmt"
 5|    "unsafe"
 6| )
 7|
 8| func main() {
 9|    type MyInt int
10|
11|    var a = []int{7, 8, 9}
```

```
12|    var b = *(*[]MyInt)(unsafe.Pointer(&a))
13|    b[0]= 123
14|    fmt.Println(a) // [123 8 9]
15|    fmt.Println(b) // [123 8 9]
16|    fmt.Printf("%T \n", a) // []int
17|    fmt.Printf("%T \n", b) // []main.MyInt
18| }
```

哪些值可以被取地址，哪些值不可以被取地址?

以下的值是不可以寻址的：

- 字符串的字节元素
- 映射元素
- 接口值的动态值（类型断言的结果）
- 常量（包括具名常量和字面量）
- 声明的包级别函数
- 方法（用做函数值）
- 中间结果值
    - 函数调用
    - 显式值转换
    - 各种操作，不包含指针解引用（dereference）操作，但是包含：
        - 通道接收操作
        - 子字符串操作
        - 子切片操作
        - 加法、减法、乘法、以及除法等等。

> 请注意：`&T{}`在Go里是一个语法糖，它是`tmp := T{}; (&tmp)`的简写形式。 所以`&T{}`是合法的并不代表字面量`T{}`是可寻址的。

以下的值是可寻址的，因此可以被取地址：

- 变量
- 可寻址的结构体的字段
- 可寻址的数组的元素
- 任意切片的元素（无论是可寻址切片或不可寻址切片）
- 指针解引用（dereference）操作

为什么映射元素不可被取地址?

在Go中，映射的设计保证一个映射值在内存允许的情况下可以加入任意个条目。 另外为了防止一个映射中为其条目开辟的内存段支离破碎，官方标准编译器使用了哈

希表来实现映射。 并且为了保证元素索引的效率，一个映射值的底层哈希表只为其中的所有条目维护一段连续的内存段。 因此，一个映射值随着其中的条目数量逐渐增加时，其维护的连续的内存段需要不断重新开辟来增容，并把原来内存段上的条目全部复制到新开辟的内存段上。 另外，即使一个映射值维护的内存段没有增容，某些哈希表实现也可能在当前内存段中移动其中的条目。 总之，映射中的元素的地址会因为各种原因而改变。 如果映射元素可以被取地址，则Go运行时（runtime）必须在元素地址改变的时候修改所有存储了元素地址的指针值。 这极大得增加了Go编译器和运行时的实现难度，并且严重影响了程序运行效率。 因此，目前，Go中禁止取映射元素的地址。

映射元素不可被取地址的另一个原因是表达式`aMap[key]`可能返回一个存储于`aMap`中的元素，也可能返回一个不存储于其中的元素零值。 这意味着表达式`aMap[key]`在`(&aMap[key]).Modify()`调用执行之后可能仍然被估值为元素零值。 这将使很多人感到困惑，因此在Go中禁止取映射元素的地址。

为什么非空切片的元素总是可被取地址，即便对于不可寻址的切片也是如此?

切片的内部类型是一个结构体，类似于

```
1| struct {
2|    elements unsafe.Pointer // 引用着一个元素序列
3|    length   int
4|    capacity int
5| }
```

每一个切片间接引用一个元素序列。 尽管一个非空切片是不可取地址的，它的内部元素序列需要开辟在内存中的某处因而必须是可取地址的。 取一个切片的元素地址事实上是取内部元素序列上的元素地址。 因此，不可寻址的非空切片的元素也是可以被取地址的。

对任意的非指针和非接口类型T，为什么类型*T的方法集总是类型T的方法集的超集，但是反之却不然?

在Go语言中，为了方便，对于一个非指针和非接口类型`T`，

- 一个`T`类型的值可以调用为`*T`类型的方法，但是仅当此`T`的值是可寻址的情况下。 编译器在调用指针属主方法前，会自动取此`T`值的地址。 因为不是任何

T值都是可寻址的，所以并非任何T值都能够调用为类型*T的方法。 这种便利只是一个语法糖，而不是一种固有的规则。

- 一个*T类型的值可以调用为类型T的方法。 这是因为解引用指针总是合法的。 这种便利不仅仅是一个语法糖，它也是一种固有的规则。

所以很合理的， *T的方法集总是T方法集的超集，但反之不然。

事实上，你可以认为对于每一个为类型T声明的方法，编译器都会为类型*T自动隐式声明一个同名和同签名的方法。 详见方法（第22章）一文。

```
 1| func (t T) MethodX(v0 ParamType0, ...) (ResultType0, ...) {
 2|     ...
 3| }
 4|
 5| // 编译器将会为*T隐式声明一个如下的方法。
 6| func (pt *T) MethodX(v0 ParamType0, ...) (ResultType0, ...)
{
 7|     return (*pt).MethodX(v0, ...)
 8| }
```

更多解释请阅读Go官方FAQ中的这个问答（https://golang.google.cn/doc/faq#different_method_sets）。

我们可以为哪些类型声明方法?

请阅读方法（第22章）一文获取答案。

在Go里如何声明不可变量?

如下是三种***不可变值***的定义：

1. 没有地址的值（所以它们不可以寻址）。
2. 有地址但是因为种种原因在语法上不可以寻址的值。
3. 可寻址但不允许在语法上被修改的值。

在Go语言中，直到现在（Go 1.23），没有值满足第三种定义。 但是零尺寸类型的变量值可以被视为事实上的（可被取地址的）不变量。

具名常量值满足第一种定义。

方法和包级函数可以被视为声明的不可变值。它们满足第二种定义。 字符串的字节元素和映射条目中的元素值同样满足第二种定义。

在Go中没有办法声明其它不可变值。

为什么没有内置的set容器类型?

集合（set）可以看作是不关心元素值的映射。 在Go语言里，`map[Tkey]struct{}`经常被用做一个集合类型。

什么是byte? 什么是rune? 如何将[]byte和[]rune类型的值转换为字符串?

在Go语言里，`byte`是`uint8`类型的一个别名。 换言之，`byte` 和 `uint8`是相同的类型。 `rune`和`int32`属于同样类似的关系。

一个`rune`值通常被用来存储一个Unicode码点。

`[]byte`和`[]rune`类型的值可以被显式地直接转换成字符串，反之亦然。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     var s0 = "Go"
 7|
 8|     var bs = []byte(s0)
 9|     var s1 = string(bs)
10|
11|     var rs = []rune(s0)
12|     var s2 = string(rs)
13|
14|     fmt.Println(s0 == s1) // true
15|     fmt.Println(s0 == s2) // true
16| }
```

更多关于字符串的信息，请阅读Go中的字符串（第19章）一文。

如何原子地操作指针值?

参见指针值的原子操作（第40章）。

iota是什么意思?

Iota是希腊字母表中的第九个字母。 在Go语言中，`iota`用在常量声明中。 在每一个常量声明组中，其值在该常量声明组的第*N*个常量规范中的值为`N`。

为什么没有一个内置的`closed`函数用来检查通道是否已经关闭?

原因是此函数的实用性非常有限。 此类函数调用的返回结果不能总是反映输入通道实参的最新状态。 所以依靠此函数的返回结果来做决定不是一个好主意。

如果你确实需要这种函数，你可以不怎么费功夫地自己写一个。 请阅读如何优雅地关闭通道（第38章）一文来了解如何编写一个`closed`函数以及如何避免使用这样的函数。

函数返回局部变量的指针是否安全?

是的，在Go中这是绝对安全的。

支持栈的Go编译器将会对每个局部变量进行逃逸分析。 对于官方标准编译器来说，如果一个值可以在编译时刻被断定它在运行时刻仅会在一个协程中被使用，则此值将被开辟在（此协程的）栈上；否则此值将被开辟在堆上。 请阅读内存块（第43章）一文了解更多。

单词*gopher*在Go社区中表示什么?

在Go社区中，***gopher***表示Go程序员。 这个昵称可能是源自于Go语言采用了一个卡通小地鼠（gopher）（https://blog.golang.com/gopher）做为吉祥物。 顺便说一下，这个卡通小地鼠是由*Renee French*设计的。 *Renee French*是Go项目首任负责人*Rob Pike*的妻子。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

Go技巧101

索引：

- 如何强制一个代码包的使用者总是使用带字段名称的组合字面量来表示此代码包中的结构体类型的值？ (#force-to-use-keyed-struct-literals)
- 如何使一个结构体类型不可比较？ (#make-struct-type-uncomparable)
- 不要使用其中涉及到的表达式之间会相互干涉的赋值语句。 (#dont-use-assignments-with-expression-dependencies)
- 如何模拟一些其它语言中支持的`for i in 0..N`循环代码块？ (#simulate-loop-from-0-to-n)
- 当我们废弃一个仍在使用的切片中的一些元素时，我们应该重置这些元素中的指针来避免暂时性的内存泄漏。 (#reset-pointers-for-dead-elements)
- 一些标准包中的某些类型的值不期望被复制。 (#avoid-copying-values-of-some-types)
- 我们可以利用`memclr`优化来重置数组或者切片中一段连续的元素。 (#memclr)
- 如何在不导入`reflect`标准库包的情况下检查一个值是否拥有某个方法。 (#check-method-existance)
- 如何高效且完美的克隆一个切片？ (#clone-slice-efficiently-and-perfectly)
- 在部分场景下我们应该使用三下标子切片形式。 (#use-three-index-subslice-form)
- 使用匿名函数来使部分延迟函数调用尽早执行。 (#execute-deferred-calls-earlier)
- 确保并表明一个自定义类型实现了指定的接口类型。 (#assert-implementation)
- 一些编译时刻断言技巧。 (#assert-at-compile-time)
- 如何声明一个最大的int和uint常量？ (#declare-max-int-uint)
- 如何在编译时刻决定系统原生字的尺寸？ (#detect-word-size-at-compile-time)
- 如何保证64位原子函数调用中操作的64位整数的地址在32位架构上总是64位对齐的？ (#64bit-alignment-guarantee)
- 尽量避免将大尺寸的值包裹在接口值中。 (#avoid-boxing-large-size-values)
- 利用BCE（边界检查消除）进行性能优化。 (#make-use-of-bce)

如何强制一个代码包的使用者总是使用带字段名称的组合字面量来表示此代码包中的结构体类型

的值?

代码包的开发者可以在一个结构体类型定义里放置一个非导出的零尺寸的字段，这样编译器将会禁止代码包的使用者使用含有一些字段但却不含有字段名字的组合字面量来创建此结构体类型的值。

例如：

```
1| // foo.go
2| package foo
3|
4| type Config struct {
5|     _    [0]int
6|     Name string
7|     Size int
8| }
```

```
1| // main.go
2| package main
3|
4| import "foo"
5|
6| func main() {
7|     //_ = foo.Config{[0]int{}, "bar", 123} // 编译不通过
8|     _ = foo.Config{Name: "bar", Size: 123} // 编译没问题
9| }
```

请尽量不要把零尺寸的非导出字段用做结构体的最后一个字段，因为这样做会有可能会增大结构体类型的尺寸（第51章）而导致一些内存浪费。

如何使一个结构体类型不可比较?

有时候，我们想要避免一个自定义的结构体类型被用做一个映射的键值类型，那么我们可以放置一个非导出的零尺寸的不可比较类型的字段在结构体类型中以使此结构体类型不可比较。 例如：

```
1| package main
2|
3| type T struct {
4|     dummy        [0]func()
5|     AnotherField int
6| }
7|
```

```
 8| var x map[T]int // 编译错误：非法的键值类型
 9|
10| func main() {
11|     var a, b T
12|     _ = a == b // 编译错误：非法的比较
13| }
```

不要使用其中涉及到的表达式之间会相互干涉的赋值语句。

目前(Go 1.23)，在一些多值赋值中有一些表达式估值顺序是未指定的（https://github.com/go101/go101/wiki/Some-evaluation-orders-in-multi-value-assignments-are-unspecified）。 因此，如果一个多值赋值语句中涉及的表达式会相互干涉，或者不太容易确定是否会相互干涉，我们应该将此多值赋值语句分拆成多个单值赋值语句。

事实上，在一些写得很糟糕的代码中，单值赋值中的表达式求值顺序也有可能是有歧义的。 例如，下面的程序可能会打印`[7 0 9]`、`[0 8 9]`或者`[7 8 9]`，依赖于具体编译器实现。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| var a = &[]int{1, 2, 3}
 6| var i int
 7| func f() int {
 8|     i = 1
 9|     a = &[]int{7, 8, 9}
10|     return 0
11| }
12|
13| func main() {
14|     // 表达式"a"、"i"和"f()"的估值顺序未定义。
15|     (*a)[i] = f()
16|     fmt.Println(*a)
17| }
```

换言之，一条赋值语句中的某个函数调用表达式的估值有可能会影响到其它非函数调用表达式的估值结果。 请阅读表达式估值顺序规则（第33章）以获取更多细节。

如何模拟一些其它语言中支持的`for i in 0..N`循环代码块?

我们可以通过遍历一个元素尺寸为零的数组或者一个空数组指针来模拟这样的循环。 例如：

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|    const N = 5
 7|
 8|    for i := range [N]struct{}{} {
 9|       fmt.Println(i)
10|    }
11|    for i := range [N][0]int{} {
12|       fmt.Println(i)
13|    }
14|    for i := range (*[N]int)(nil) {
15|       fmt.Println(i)
16|    }
17| }
```

当我们废弃一个仍在使用的切片中的一些元素时，我们应该重置这些元素中的指针来避免暂时性的内存泄漏。

关于细节，请阅读如何删除切片元素（第18章）和因为未重置丢失的切片元素中的指针而造成的临时性内存泄露（第45章）。

一些标准包中的某些类型的值不期望被复制。

`bytes.Buffer`类型、`strings.Builder`类型以及在`sync`标准库包里的类型的值不推荐被复制。 （它们确实不应该被复制，尽管在某些特定情形下复制它们或许是没有问题的。）

`strings.Builder`的实现会在运行时刻探测到非法的`strings.Builder`值复制。一旦这样的复制被发现，就会产生恐慌。例如：

```
 1| package main
 2|
 3| import "strings"
 4|
 5| func main() {
 6|    var b strings.Builder
 7|    b.WriteString("hello ")
 8|    var b2 = b
 9|    b2.WriteString("world!") // 一个恐慌将在这里产生
10| }
```

复制标准库包sync中类型的值会被Go官方工具链提供的go vet命令检测到并被警告。

```
 1| // demo.go
 2| package demo
 3|
 4| import "sync"
 5|
 6| func f(m sync.Mutex) { // warning: f passes lock by value:
sync.Mutex
 7|    m.Lock()
 8|    defer m.Unlock()
 9|    // do something ...
10| }
```

```
$ go vet demo.go
./demo.go:5: f passes lock by value: sync.Mutex
```

复制bytes.Buffer的值不会在运行时被检查到，也不会被go vet命令所检测到。千万要小心不要随意这样做。

我们可以利用memclr优化来重置数组或者切片中一段连续的元素。

关于细节，请阅读memclr优化（第18章）。

如何在不导入reflect标准库包的情况下检查一个值是否拥有某个方法。

可以使用下面的例子中的方法。（假设需要被检查的方法的描述是`M(int) string`。）

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| type A int
 6| type B int
 7| func (b B) M(x int) string {
 8|    return fmt.Sprint(b, ": ", x)
 9| }
10|
11| func check(v interface{}) bool {
12|    _, has := v.(interface{M(int) string})
13|    return has
14| }
15|
16| func main() {
17|    var a A = 123
18|    var b B = 789
19|    fmt.Println(check(a)) // false
20|    fmt.Println(check(b)) // true
21| }
```

如何高效且完美地克隆一个切片?

关于细节请阅读[这篇wiki文章](https://github.com/go101/go101/wiki/How-to-perfectly-clone-a-slice%3F)（https://github.com/go101/go101/wiki/How-to-perfectly-clone-a-slice%3F）和[这篇wiki文章](https://github.com/go101/go101/wiki/How-to-efficiently-clone-a-slice%3F)（https://github.com/go101/go101/wiki/How-to-efficiently-clone-a-slice%3F）。

在部分场景下我们应该使用三下标子切片形式。

假设一个包提供了一个`func NewX(...Option) *X`函数，并且这个函数的实现将输入选项与一些内部默认选项合并，那么下面的实现是不推荐的。

```
 1| func NewX(opts ...Option) *X {
 2|    options := append(opts, defaultOpts...)
 3|    // 使用合并后选项来创建一个X值并返回其指针。
 4|    // ...
 5| }
```

上述实现不被推荐的原因是append函数调用可能会修改输入实参opts的底层潜在Option元素序列。 对大多数场景，这可能是没问题的。但是对某些特殊场景，这有可能会导致后续代码执行产生不期望的结果。

为了避免输入实参的底层Option元素序列被修改，我们应该使用下面的实现方法：

```
 1| func NewX(opts ...Option) *X {
 2|     // 改用三下标子切片格式。
 3|     opts = append(opts[:len(opts):len(opts)],
defaultOpts...)
 4|     // 使用合并后选项来创建一个X值并返回其指针。
 5|     // ...
 6| }
```

另一方面，对于NewX函数的调用者来说，不应该依赖于此函数的具体实现，所以最好使用三下标子切片形式options[:len(options):cap(options)]来传递实参。

另外一个需要使用三下标子切片格式的场景在这篇wiki文章（https://github.com/go101/go101/wiki/The-capacity-of-the-result-byte-%28or-rune%29-slice-of-a-conversion-from-a-string-is-undefined）中被提及。

三下标子切片格式的一个缺点是它们有些冗长。 事实上，我曾经提了一个建议（https://github.com/golang/go/issues/25638）来让三下标格式看上起简洁得多。 但是此建议被否决了。

使用匿名函数来使部分延迟函数调用尽早执行。

关于细节，请阅读这篇文章（第29章）。

确保并表明一个自定义类型实现了指定的接口类型。

我们可以将一个自定义类型的一个值赋给指定接口类型的一个变量来确保此自定义类型实现了指定接口类型。 更重要的是，这样可以表明此自定义类型实现了指定接口类型。 使用自解释的代码编写文档比使用注释来编写文档要自然得多。

```
 1| package myreader
 2|
 3| import "io"
 4|
```

```
 5| type MyReader uint16
 6|
 7| func NewMyReader() *MyReader {
 8|    var mr MyReader
 9|    return &mr
10| }
11|
12| func (mr *MyReader) Read(data []byte) (int, error) {
13|    switch len(data) {
14|    default:
15|       *mr = MyReader(data[0]) << 8 | MyReader(data[1])
16|       return 2, nil
17|    case 2:
18|       *mr = MyReader(data[0]) << 8 | MyReader(data[1])
19|    case 1:
20|       *mr = MyReader(data[0])
21|    case 0:
22|    }
23|    return len(data), io.EOF
24| }
25|
26| // 下面三行中的任一行都可以保证类型*MyReader实现
27| // 了接口io.Reader。
28| var _ io.Reader = NewMyReader()
29| var _ io.Reader = (*MyReader)(nil)
30| func _() {_ = io.Reader(nil).(*MyReader)}
```

一些编译时刻断言技巧。

除了上一个技巧中提到过的编译时刻断言技巧，下面将要介绍更多编译时刻断言技巧。

下面是一些方法用来在编译时刻保证常量N不小于另一个常量M：

```
 1| // 下面任一行均可保证N >= M
 2| func _(x []int) {_ = x[N-M]}
 3| func _(){_ = []int{N-M: 0}}
 4| func _([N-M]int){}
 5| var _ [N-M]int
 6| const _ uint = N-M
 7| type _ [N-M]int
 8|
 9| // 如果M和N都是正整数常量，则我们也可以使用下一行所示的方法。
10| var _ uint = N/M - 1
```

另一个方法是借鉴@lukechampine（https://twitter.com/lukechampine/status/1026695476811390976）的一个点子。 此点子利用了容器组合字面量中不能出现重复的常量键值（第18章）这一规则。

```
 1| var _ = map[bool]struct{}{false: struct{}{}, N>=M: struct{}
{}}
```

此方法看上去有些冗长，但是它更加通用。它可以用来断言任何条件。 其实，它也可以不必很冗长，但需要多消耗一点（完全可以忽略的）内存，如下面所示：

```
 1| var _ = map[bool]int{false: 0, N>=M: 1}
```

类似地，下面是断言两个整数常量相等的方法：

```
 1| var _ [N-M]int; var _ [M-N]int
 2| type _ [N-M]int; type _ [M-N]int
 3| const _, _ uint = N-M, M-N
 4| func _([N-M]int, [M-N]int) {}
 5|
 6| var _ = map[bool]int{false: 0, M==N: 1}
 7|
 8| var _ = [1]int{M-N: 0} // 唯一被允许的元素索引下标为0
 9| var _ = [1]int{}[M-N]  // 唯一被允许的元素索引下标为0
10|
11| var _ [N-M]int = [M-N]int{}
```

最后一行的灵感同样来自于Luke Champine的一条tweet。

下面是一些用来断言一个常量字符串是不是一个空串的方法。

```
 1| type _ [len(aStringConstant)-1]int
 2| var _ = map[bool]int{false: 0, aStringConstant != "": 1}
 3| var _ = aStringConstant[:1]
 4| var _ = aStringConstant[0]
 5| const _ = 1/len(aStringConstant)
```

最后一行借鉴自Jan Mercl的一个点子（https://groups.google.com/d/msg/golang-nuts/w1-JQMaH7c4/qzBFSPImBgAJ）。

有时候，为了避免包级变量消耗太多的内存，我们可以把断言代码放在一个名为空标识符的函数体中。 例如：

```
 1| func _() {
 2|     var _ = map[bool]int{false: 0, N>=M: 1}
```

```
 3|     var _ [N-M]int
 4| }
```

如何声明一个最大的int和uint常量?

```
 1| const MaxUint = ^uint(0)
 2| const MaxInt = int(^uint(0) >> 1)
```

如何在编译时刻决定系统原生字的尺寸?

这个技巧和Go无关。

```
 1| const Is64bitArch = ^uint(0) >> 63 == 1
 2| const Is32bitArch = ^uint(0) >> 63 == 0
 3| const WordBits = 32 << (^uint(0) >> 63) // 64或32
```

如何保证64位原子函数调用中操作的64位整数的地址在32位架构上总是64位对齐的?

关于细节，请阅读关于Go值的内存布局（第44章）一文。

尽量避免将大尺寸的值包裹在接口值中。

当一个非接口值被赋值给一个接口值时，此非接口值的一个副本将被包裹到此接口值中。 副本复制的开销和非接口值的尺寸成正比。尺寸越大，复制开销越大。 所以请尽量避免将大尺寸的值包裹到接口值中。

在下面的例子中，后两个打印调用的成本要比前两个低得多。

```
 1| package main
 2|
 3| import "fmt"
 4|
 5| func main() {
 6|     var a [1000]int
 7|
 8|     // 这两行的开销相对较大，因为数组a中的元素都将被复制。
 9|     fmt.Println(a)
10|     fmt.Printf("Type of a: %T\n", a)
11|
```

```
12|    // 这两行的开销较小，数组a中的元素没有被复制。
13|    fmt.Printf("%v\n", a[:])
14|    fmt.Println("Type of a:", fmt.Sprintf("%T", &a)[1:])
15| }
```

关于不同种类的类型的尺寸，请阅读值复制成本（第34章）一文。

利用BCE（边界检查消除）进行性能优化。

请阅读此文（第35章）来获知什么是边界检查消除（BCE）以及目前的标准编译器对BCE的支持程度。

下面是一个利用了BCE进行性能优化的例子：

```
 1| package main
 2|
 3| import (
 4|    "strings"
 5|    "testing"
 6| )
 7|
 8| func NumSameBytes_1(x, y string) int {
 9|    if len(x) > len(y) {
10|       x, y = y, x
11|    }
12|    for i := 0; i < len(x); i++ {
13|       if x[i] != y[i] {
14|          return i
15|       }
16|    }
17|    return len(x)
18| }
19|
20| func NumSameBytes_2(x, y string) int {
21|    if len(x) > len(y) {
22|       x, y = y, x
23|    }
24|    if len(x) <= len(y) { // 虽然代码多了，但是效率提高了
25|       for i := 0; i < len(x); i++ {
26|          if x[i] != y[i] { // 边界检查被消除了
27|             return i
28|          }
29|       }
30|    }
```

```
31|    return len(x)
32| }
33|
34| var x = strings.Repeat("hello", 100) + " world!"
35| var y = strings.Repeat("hello", 99) + " world!"
36|
37| func BenchmarkNumSameBytes_1(b *testing.B) {
38|    for i := 0; i < b.N; i++ {
39|       _ = NumSameBytes_1(x, y)
40|    }
41| }
42|
43| func BenchmarkNumSameBytes_2(b *testing.B) {
44|    for i := 0; i < b.N; i++ {
45|       _ = NumSameBytes_2(x, y)
46|    }
47| }
```

从下面所示的基准测试结果来看，函数`NumSameBytes_2`比函数`NumSameBytes_1`效率更高。

```
BenchmarkNumSameBytes_1-4       10000000          669 ns/op
BenchmarkNumSameBytes_2-4       20000000          450 ns/op
```

请注意：标准编译器（gc）的每个新的主版本都会有很多小的改进。 上例中所示的优化从gc 1.11开始才有效。 未来的gc版本可能会变得更加智能，以使函数`NumSameBytes_2`中使用技巧变得不再必要。 事实上，从gc 1.11开始，如果`x`和`y`是两个切片，即使上例中使用小技巧没有被使用，`y[i]`中的边界检查也已经被消除了。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

更多关于Go的知识

《Go语言101》中的系列文章主要着墨于Go语法和语义。 更多和Go相关的话题并没有在《Go语言101》中详尽解释。 本文将对这些话题做一个简单的介绍，并提供一些链接供读者自行探索。

程序性能分析、基准测试、单元测试和模糊测试

我们可以使用Go官方工具链中的`go test`命令来运行单元测试和基准测试。 测试源文件名必须以`_test.go`结尾。 Go官方工具链也支持程序性能分析。 请阅读下列文章获取详情：

- Go程序性能分析（https://github.com/golang/go/wiki/Performance）
- `testing`标准库包（https://golang.google.cn/pkg/testing/）
- 使用例子程序做测试（https://blog.golang.org/examples）。
- 使用子单元测试和子基准测试（https://blog.golang.org/subtests）
- `go test`命令选项（https://golang.google.cn/cmd/go/#hdr-Test_packages）
- Go Fuzzing（https://go.dev/doc/fuzz/）。

gccgo

gccgo（https://github.com/golang/gofrontend）是Go核心团队维护开发的另一款Go编译器。 它的主要目的是为了验证标准编译器（gc）的正确性。 我们可以在`go run`、`go build`和`go install`等命令中使用选项`-compiler=gccgo`来指定使用gccgo编译器。 此选项需要安装`gccgo`软件包才能起作用。 gccgo命令也可单独运行（https://golang.google.cn/doc/install/gccgo#Using_gccgo）。

go/*标准库包

`go/*`标准库包提供Go源文件解析相关的功能。这些库包对于开发各种Go源代码分析工具很有帮助。 请阅读Go代码分析（https://github.com/golang/example/tree/master/gotypes）和这些库包的文档（https://golang.google.cn/pkg/go/）来获取如何使用这些库包中提供的功能。

系统调用

我们可以使用syscall标准库包中的函数来进行系统调用。 注意此标准库包和其它标准库包不同，它其中的函数是和具体操作系统相关的。

Go汇编

Go函数可以使用Go汇编语言来编写。Go汇编是一种跨平台（尽管并非100%）的汇编语言。 Go汇编汇编常用来实现一些对性能要求很高的函数。

更多详情，请阅读下列文章：

- Go汇编快速导读（https://golang.google.cn/doc/asm）
- Go汇编的设计（https://talks.golang.org/2016/asm.slide）

cgo

通过cgo机制，我们可以在Go代码中调用C代码，或者反之。 请阅读下列链接获取详情：

- cgo官方文档（https://blog.golang.org/cmd/cgo/）
- C? Go? Cgo!（https://blog.golang.org/c-go-cgo）
- Go维基中的cgo页面（https://github.com/golang/go/wiki/cgo）

通过C代码做为桥梁，我们也可以使C++和Go代码能够相互调用。

注意，cgo的使用将给跨平台Go项目的维护开发带来一些麻烦。另外Go和C之间互调不如Go-Go和C-C调用高效。

跨平台编译

标准Go编译器支持跨平台编译。通过设置GOOS和GOARCH两个环境变量，我们可以使用go build命令在Linux系统中编译出Windows和Mac程序，反之亦然。 请阅读下面的代码获取详情：

- 在Linux上编译Windows程序
  （https://github.com/golang/go/wiki/WindowsCrossCompiling）
- 目前支持的操作系统和编译架构
  （https://golang.google.cn/doc/install/source#environment）。

特别地，自从GO 1.11，标准Go编译器开始支持WebAssembly做为一种新的架构。请阅读此篇维基文章（https://github.com/golang/go/wiki/WebAssembly）获取详情。

编译器指示（compiler directive）

标准Go编译器支持若干编译器指示（https://golang.google.cn/cmd/compile/#hdr-Compiler_Directives）。一个编译器指示以注释的形式`//go:DirectiveName args`出现。比如我们可以使用go:generate（https://blog.golang.org/generate）编译器指示来生成代码；或者使用Go 1.16版本引入go:embed（https://golang.google.cn/pkg/embed/）编译器指示来内嵌数据文件。

构建编译约束/标签（build constraint/tag）

我们可以使用构建编译约束（https://golang.google.cn/pkg/go/build/）（或称标签）来让编译器选择性地忽略某些源文件。选择标签可以出现在源代码文件中的最顶部，也可以出现在源代码文件名（不包含`.go`后缀）的结尾（并用下划线分隔开来）。注意：Go官方工具链1.17引入的（https://github.com/golang/go/issues/41184）新的`//go:build`指示（https://go.googlesource.com/proposal/+/master/design/draft-gobuild.md）将逐渐替代`// +build`构建编译约束。

更多的编译模式（build mode）

Go官方工具链中的`go build`命令支持更多的编译模式。运行`go help buildmode`可以列出所有支持的模式，或者查看官方文档（https://golang.google.cn/cmd/go/#hdr-Build_modes）来查看这些模式的解释说明。除了最常用的default模式，plugin（插件）模式可能是另一种用的较多的模式。我们可以使用`plugin`标准库包（https://golang.google.cn/pkg/plugin/）中的函数来加载和使用Go插件。

本书由老貘历时三年写成。目前本书仍在不断改进和增容中。你的赞赏是本书和Go101.org网站不断增容和维护的动力。



（请搜索关注微信公众号“Go 101”或者访问github.com/go1ang101/go1ang101获取本书最新版）

目录

目录 1
第0章：关于《Go语言101》 4
第1章：致谢 8
第2章：Go语言简介 10
第3章：Go官方工具链 13
第4章：程序源代码基本元素介绍 17
第5章：关键字和标识符 22
第6章：基本类型和它们的字面量表示 25
第7章：常量和变量 36
第8章：运算操作符 53
第9章：函数声明和调用 66
第10章：代码包和包引入 73
第11章：表达式、语句和简单语句 83
第12章：基本流程控制语法 87
第13章：协程、延迟函数调用、以及恐慌和恢复 103
第14章：Go类型系统概述 117
第15章：指针 128
第16章：结构体 137
第17章：值部 146
第18章：数组、切片和映射 152
第19章：字符串 194
第20章：函数 206
第21章：通道 219
第22章：方法 234
第23章：接口 247
第24章：类型内嵌 266
第25章：非类型安全指针 281
第26章：泛型 300
第27章：反射 305
第28章：Go代码断行规则 319
第29章：更多关于延迟函数调用的知识点 327
第30章：一些恐慌/恢复用例 333
第31章：详解恐慌和恢复原理 339
第32章：代码块和标识符作用域 347
第33章：表达式估值顺序规则 357
第34章：值复制成本 368
第35章：边界检查消除 372
第36章：并发同步概述 380
第37章：通道用例大全 382
第38章：如何优雅地关闭通道 417
第39章：sync标准库包中提供的并发同步技术 435

第40章：sync/atomic标准库包中提供的原子操作 449
第41章：Go中的内存顺序保证 460
第42章：一些常见并发编程错误 471
第43章：内存块 483
第44章：内存布局 491
第45章：一些可能的内存泄漏场景 499
第46章：一些简单的总结 504
第47章：Go中的nil 510
第48章：类型转换、赋值和值比较规则大全 518
第49章：Go中的一些语法/语义例外 528
第50章：Go细节101 542
第51章：Go问答101 579
第52章：Go技巧101 599
第53章：更多关于Go的知识 612