# 《Go语言圣经》

《The Go Programming Language》中文版

**Go语言圣经（中文版）**

Go语言圣经《The Go Programming Language》中文版本，仅供学习交流之用。

* 项目主页：<http://github.com/golang-china/gopl-zh>
* 项目主页：<http://bitbucket.org/golang-china/gopl-zh>
* 原版官网：<http://gopl.io>

在线预览：

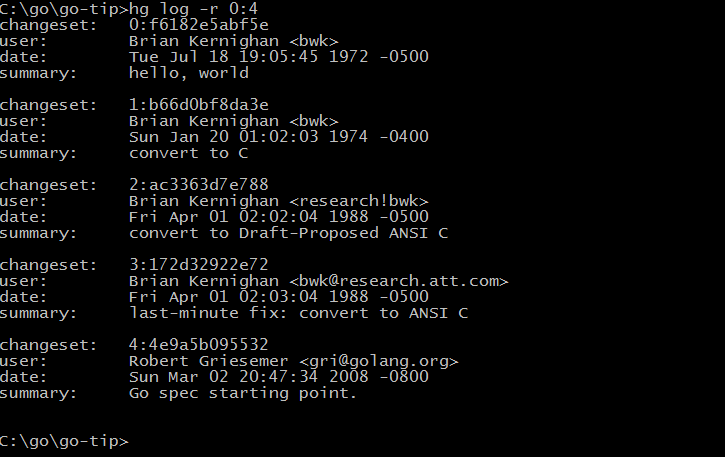
* <http://docs.ruanjiadeng.com/gopl-zh>
* <http://shifei.me/gopl-zh>
* <http://2goo.info/media/html/gopl-zh-gh-pages>
* <http://docs.plhwin.com/gopl-zh>

**版本信息**

* 仓库版本：[master](https://github.com/yar999/gopl-zh/tree/7ff8130293a654b9ac04e153970adbc6aa9f8192/gopl-zh-master.zip)
* 构建时间：2016-03-30

**译者序**

在上个世纪70年代，贝尔实验室的[Ken Thompson](http://genius.cat-v.org/ken-thompson/)和[Dennis M. Ritchie](http://genius.cat-v.org/dennis-ritchie/) 合作发明了[UNIX](http://doc.cat-v.org/unix/)系统，同时Dennis M. Ritchie为了解决UNIX系统的移植性问题而发明了C语言，贝尔实验室的UNIX和C语言两大发明奠定了整个现代IT行业最重要的软件基础（目前的三大桌面操作系统中的[Linux](http://www.linux.org/)和[Mac OS X](http://www.apple.com/cn/osx/) 都是源于UNIX系统，两大移动平台的操作系统IOS和Android也都是源于UNIX系统。C系家族的编程语言占据统治地位达几十年之久）。在UNIX和C语言发明40年之后，目前已经在Google工作的Ken Thompson和[Rob Pike](http://genius.cat-v.org/rob-pike/)（他们在贝尔实验室时就是同事）、还有[Robert Griesemer](http://research.google.com/pubs/author96.html)（设计了V8引擎和HotSpot虚拟机）一起合作，为了解决在21世纪多核和网络化环境下越来越复杂的编程问题而发明了Go语言。从Go语言库早期代码库日志可以看出它的演化历程（Git用git log –before={2008-03-03} –reverse命令查看）：



从早期提交日志中可以看出，Go语言是从Ken Thompson发明的B语言、Dennis M. Ritchie发明的C语言逐步演化过来的，是C语言家族的成员，因此很多人将Go语言称为21世纪的C语言。纵观这几年来的发展趋势，Go语言已经成为云计算、云存储时代最重要的基础编程语言。

在C语言发明之后约5年的时间之后（1978年），[Brian W. Kernighan](http://www.cs.princeton.edu/~bwk/)和Dennis M. Ritchie合作编写出版了C语言方面的经典教材[《The C Programming Language》](http://s3-us-west-2.amazonaws.com/belllabs-microsite-dritchie/cbook/index.html)，该书被誉为C语言程序员的圣经，作者也被大家亲切地称为[K&R](https://en.wikipedia.org/wiki/K%26R)。同样在Go语言正式发布（2009年）约5年之后（2014年开始写作，2015年出版），由Go语言核心团队成员[Alan A. A. Donovan](https://github.com/adonovan)和K&R中的Brian W. Kernighan合作编写了Go语言方面的经典教材[《The Go Programming Language》](http://gopl.io/)。Go语言被誉为21世纪的C语言，如果说K&R所著的是圣经的旧约，那么D&K所著的必将称为圣经的新约。该书介绍了Go语言几乎全部特性，并且随着语言的深入层层递进，对每个细节都解读得非常细致，每一节内容都精彩不容错过，是广大Gopher和必读书目。大部分Go语言核心团队的成员都参与了该书校对工作，因此该书的质量是可以完全放心的。

同时，单凭阅读和学习其语法结构并不能真正地掌握一门编程语言，必须进行足够多的编程实践——亲自编写一些程序并研究学习别人写的程序。要从利用Go语言良好的特性使得程序模块化，充分利用Go的标准函数库以Go语言自己的风格来编写程序。书中包含了上百个精心挑选的习题，希望大家能先用自己的方式尝试完成习题，然后再参考官方给出的解决方案。

该书英文版约从2015年10月开始公开发售，其中日文版本最早参与翻译和审校（参考致谢部分）。在2015年10月，我们并不知道中文版是否会及时引进、将由哪家出版社引进、引进将由何人来翻译、何时能出版，这些信息都成了一个秘密。中国的Go语言社区是全球最大的Go语言社区，我们从一开始就始终紧跟着Go语言的发展脚步。我们应该也完全有能力以中国Go语言社区的力量同步完成Go语言圣经中文版的翻译工作。与此同时，国内有很多Go语言爱好者也在积极关注该书（本人也在第一时间购买了纸质版本，亚马逊价格314人命币。补充：国内也即将出版英文版，价格79元）。为了Go语言的学习与交流，大家决定合作免费翻译该书。

翻译工作从2015年11月20日前后开始，到2016年1月底初步完成，前后历时约2个月时间（在其他语言版本中，全球第一个完成翻译的，基本做到和原版同步）。其中，[chai2010](https://github.com/chai2010)翻译了前言、第2~4章、第10~13章，[Xargin](https://github.com/cch123)翻译了第1章、第6章、第8~9张，[GrazySssst](https://github.com/CrazySssst)翻译了第5章，[foreversmart](https://github.com/foreversmart)翻译了第7章，大家共同参与了基本的校验工作，还有其他一些朋友提供了积极的反馈建议。如果大家还有任何问题或建议，可以直接到[中文版](https://github.com/golang-china/gopl-zh/issues)项目页面提交Issue，如果发现英文版原文在勘误中未提到的任何错误，可以直接去[英文版](https://github.com/adonovan/gopl.io/)项目提交。

最后，希望这本书能够帮助大家用Go语言快乐地编程。

2016年1月 于 武汉

## 前言

“Go是一个开源的编程语言，它很容易用于构建简单、可靠和高效的软件。”（摘自Go语言官方网站：<http://golang.org>）

Go语言由来自Google公司的Robert Griesemer，Rob Pike和Ken Thompson三位大牛于2007年9月开始设计和实现，然后于2009年的11月对外正式发布（译注：关于Go语言的创世纪过程请参考<http://talks.golang.org/2015/how-go-was-made.slide>）。语言及其配套工具的设计目标是具有表达力，高效的编译和执行效率，有效地编写高效和健壮的程序。

Go语言有着和C语言类似的语法外表，和C语言一样是专业程序员的必备工具，可以用最小的代价获得最大的战果。但是它不仅仅是一个更新的C语言。它还从其他语言借鉴了很多好的想法，同时避免引入过渡的复杂性。Go语言中和并发编程相关的特性是全新的也是有效的，同时对数据抽象和面向对象编程的支持也很灵活。Go语言同时还集成了自动垃圾收集技术用于更好地管理内存。

Go语言尤其适合编写网络服务相关基础设施，同时也适合开发一些工具软件和系统软件。但是Go语言确实是一个通用的编程语言，它也可以用在图形图像驱动编程、移动应用程序开发和机器学习等诸多领域。目前Go语言已经成为受欢迎的作为无类型的脚本语言的替代者：因为Go编写的程序通常比脚本语言运行的更快也更安全，而且很少会发生意外的类型错误。

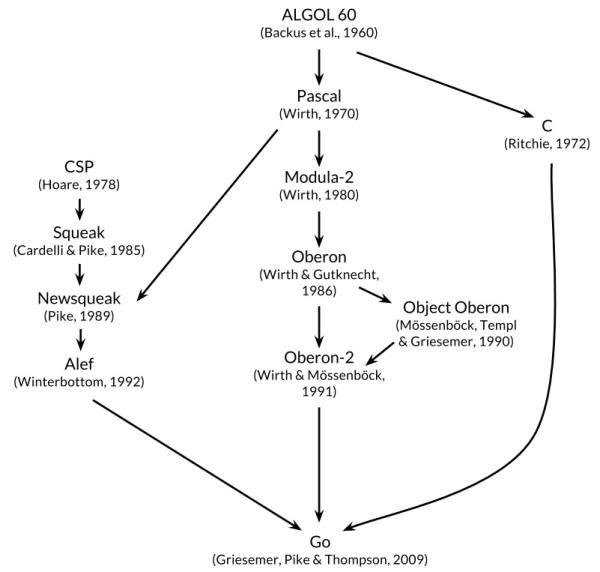
Go语言还是一个开源的项目，可以免费获取编译器、库、配套工具的源代码。Go语言都贡献者来自一个活跃的全球社区。Go语言可以运行在类[UNIX](http://doc.cat-v.org/unix/)系统——比如[Linux](http://www.linux.org/)、[FreeBSD](https://www.freebsd.org/)、[OpenBSD](http://www.openbsd.org/)、[Mac OSX](http://www.apple.com/cn/osx/)、[Plan9](http://plan9.bell-labs.com/plan9/)系统和[Microsoft Windows](https://www.microsoft.com/zh-cn/windows/)操作系统之上。Go语言编写的程序无需修改就可以运行在上面这些环境。

本书是为了帮助你开始以有效的方式使用Go语言，充分利用语言本身的特性和自带的标准库去编写清晰地道的Go程序。

## Go语言起源

编程语言的演化就像生物物种的演化类似，一个成功的编程语言的后代一般都会继承它们祖先的有点；当然有时多种语言杂合也可能会产生令人惊讶的特性；还有一些激进的新特性可能并没有先例。我们可以通过观察编程语言和软硬件环境是如何相互促进、相互影响的演化过程而学到很多。

下图展示了有哪些早期的编程语言对Go语言的设计产生了重要影响。



Go语言有时候被描述为“C类似语言”，或者是“21世纪的C语言”。Go从C语言继承了相似的表达式语法、控制流结构、基础数据类型、调用参数传值、指针等很多思想，还有C语言一直所看中的编译后机器码的运行效率以及和现有操作系统的无缝适配。

但是在Go语言的家族树中还有其它的祖先。其中一个有影响力的分支来自[Niklaus Wirth](https://en.wikipedia.org/wiki/Niklaus_Wirth)所涉及的Pascal语言。然后Modula-2语言激发了包的概念。然后Oberon语言摒弃了模块接口文件和模块实现文件之间的区别。第二代的Oberon-2语言直接影响了包的导入和声明的语法，还有Oberon语言的面向对象特性所提供的方法的声明语法等。

Go语言的另一支祖先，带来了Go语言区别其他语言的重要特性，灵感来自于贝尔实验室的[Tony Hoare](https://en.wikipedia.org/wiki/Tony_Hoare)与1978年发表的鲜为外界所知的关于并发研究的基础文献——顺序通信进程（communicating sequential processes），缩写为CSP。在CSP中，程序是一组中间没有共享状态的平行运行的处理过程，它们之间使用管道进行通信和控制同步。不过Tony Hoare的CSP只是一个用于描述并发性基本概念的描述语言，并不是一个可以编写可执行程序的通用编程语言。

接下来，Rob Pike和其他人开始不断尝试将[CSP](https://en.wikipedia.org/wiki/Communicating_sequential_processes)引入实际的编程语言中。他们第一次尝试引入CSP特性的编程语言叫[Squeak](http://doc.cat-v.org/bell_labs/squeak/)（老鼠间交流的语言），是一个提供鼠标和键盘事件处理的编程语言，它的管道是静态创建的。然后是改进版的[Newsqueak](http://doc.cat-v.org/bell_labs/squeak/)语言，提供了类似C语言语句和表达式的语法和类似Pascal语言的推导语法。Newsqueak是一个带垃圾回收的纯函数式语言，它再次针对键盘、鼠标和窗口事件管理。但是在Newsqueak语言中管道是动态创建的，属于第一类值，可以保存到变量中。

在Plan9操作系统中，这些优秀的想法被吸收到了一个叫Alef的编程语言中。Alef试图将Newsqueak语言改造为系统编程语言，但是因为缺少垃圾回收机制而导致并发编程很痛苦。（译注：在Alef之后还有一个叫Limbo的编程语言，Go语言从其中借鉴了很多特性。具体请参考Pike的讲稿：[http://talks.golang.org/2012/concurrency.slide#9](http://talks.golang.org/2012/concurrency.slide" \l "9)）

Go语言的其他的一些特性零散地来自于其他一些编程语言，比如iota语法是从APL语言借鉴，词法作用域与嵌套函数来自于Scheme语言（和其他很多语言）。当然，我们也可以从Go中发现很多创新的设计。比如Go语言的切片为动态数组提供了有效的随机存取的性能，这可能会让人联想到链表的底层的共享机制。还有Go语言新发明的defer语句。

## Go语言项目

所有的编程语言都反映了语言设计者对编程哲学的反思，通常包括之前的语言所暴露的一些不足地方的改进。Go项目是在Google公司维护超级复杂的几个软件系统遇到的一些问题的反思（但是这类问题绝不是Google公司所特有的）。

正如Rob Pike所说，“软件的复杂性是乘法级相关的”，通过增加一个部分的复杂性来修复问题通常将慢慢地增加其他部分的复杂性。通过增加功能和选项和配置是修复问题的最快途径，但是这很容易让人忘记简洁的内涵，即使从长远来看，简洁依然是好软件的关键因素。

简洁的设计需要在工作开始的时候舍弃不必要的想法，并且在软件的生命周期内严格区别好的改变或坏的改变。通过足够的努力，一个好的改变可以在不破坏原有完整概念的前提下保持自适应，正如Fred Brooks所说的“概念完整性”而一个坏的改变则不能达到这个效果，它们仅仅是通过肤浅的和简单的妥协来破坏原有设计的一致性。只有通过简洁的设计，才能让一个系统保持稳定、安全和持续的进化。

Go项目包括编程语言本身，附带了相关的工具和标准库，最后但并非代表不重要的，关于简介编程哲学的宣言。就事后诸葛的角度来看，Go语言的这些地方都做的还不错：拥有自动垃圾回收、一个包系统、函数作为一等公民、词法作用域、系统调用接口、只读的UTF8字符串等。但是Go语言本身只有很少的特性，也不太可能添加太多的特性。例如，它没有隐式的数值转换，没有构造函数和析构函数，没有运算符重载，没有默认参数，也没有继承，没有泛型，没有异常，没有宏，没有函数修饰，更没有线程局部存储。但是语言本身是成熟和稳定的，而且承诺保证向后兼容：用之前的Go语言编写程序可以用新版本的Go语言编译器和标准库直接构建而不需要修改代码

Go语言有足够的类型系统以避免动态语言中那些粗心的类型错误，但是Go语言的类型系统相比传统的强类型语言又要简洁很多。虽然有时候这会导致一个“无类型”的抽象类型概念，但是Go语言程序员并不需要像C++或Haskell程序员那样纠结于具体类型的安全属性。在实践中Go语言简介的类型系统给程序员带来了更多的安全性和更好的运行时性能。

Go语言鼓励当代计算机系统设计的原则，特别是局部的重要性。它的内置数据类型和大多数的标准库数据结构都经过精心设计而避免显式的初始化或隐式的构造函数，因为很少的内存分配和内存初始化代码被隐藏在库代码中了。Go语言的聚合类型（结构体和数组）可以直接操作它们的元素，只需要更少的存储空间、更少的内存分配，而且指针操作比其他间接操作的语言也更有效率。由于现代计算机是一个并行的机器，Go语言提供了基于CSP的并发特性支持。GO语言的动态栈使得轻量级线程goroutine的初始栈可以很小，因此创建一个goroutine的代价很小，创建百万级的goroutine完全是可行的。

Go语言的标准库（通常被称为语言自带的电池），提供了清晰的构建模块和公共接口，包含I/O操作、文本处理、图像、密码学、网络和分布式应用程序等，并支持许多标准化的文件格式和编解码协议。库和工具使用了大量的约定来减少额外的配置和解释，从而最终简化程序的逻辑，而且每个Go程序结构都是如此的相似，因此Go程序也很容易学习。使用Go语言自带工具构建Go语言项目只需要使用文件名和标识符，一个偶尔的特殊注释来确定所有的库、可执行文件、测试、基准测试、例子、以及特定于平台的变量、项目的文档等；Go语言源代码本身就包含了构建规范。

## 本书的组织

我们假设你已经有一种或多种其他编程语言的使用经历，不管是类似C、C++或Java和编译型语言，还是类似Python、Ruby、JavaScript的脚本语言，因此我们不会像对完全的编程语言初学者那样解释所有的细节。因为Go语言的变量、常量、表达式、控制流和函数等基本语法也是类似的。

第一章包含了本教程的基本结构，通过十几个程序介绍了用Go语言如何实现类似读写文件、文本格式化、创建图像、网络客户端和服务器通讯等日常工作。

第二章描述了Go语言程序的基本元素结构、变量、新类型定义、包和文件、一级作用域的概念。第三章讨论了数字、布尔值、字符串和常量，并演示了如何显示和处理Unicode字符。第四章描述了复合类型，从简单的数组、字典、切片到动态列表。第五章涵盖了函数，并讨论了错误处理、panic和recover，还有defer语句。

第一章到第五章是基础部分，主流命令式编程语言这部分都类似。个别之处，Go语言有自己特色的语法和风格，但是大多数程序员能很快适应。其余章节是Go语言特有的：方法、接口、并发、包、测试和反射等语言特性。

Go语言的面向对象机制与一般语言不同。它没有类层次结构，甚至可以说没有类；仅仅通过组合（而不是继承）简单的对象来构建复杂的对象。方法不仅可以定义在结构体上，而且可以定义在任何用户自定义的类型上；并且具体类型和抽象类型（接口）之间的关系是隐式的。所以很多类型的设计者可能并不知道该类型到底实现了哪些接口。方法在第六章讨论，接口在第七章讨论。

第八章讨论了基于顺序通信进程（CSP）概念的并发编程，使用goroutine和channels处理并发编程。第九章则讨论了传统的基于共享变量的并发编程。

第十章描述了包机制和包的组织结构。这一章还展示了如何有效地利用Go自带的工具，使用单个命令完成编译、测试、基准测试、代码格式化、文档以及其他诸多任务。

第十一章讨论了单元测试，Go语言的工具和标准库中集成了轻量级的测试功能，避免了强大但复杂的测试框架。测试库提供了一些基本构件，必要时可以用来构建复杂的测试构件。

第十二章讨论了反射，一种程序在运行期间审视自己的能力。反射是一个强大的编程工具，不过要谨慎的使用；这一章利用反射机制实现一些重要的Go语言库函数，展示了反射的强大用法。第十三章解释了底层编程的细节，在必要时，可以使用unsafe包绕过Go语言安全的类型系统。

部分章节的后面有练习题，根据对Go语言的理解修改书中的例子来探索Go语言的用法。

书中所有的代码都可以从<http://gopl.io上的Git>仓库下载。go get命令根据每个例子的导入路径智能地获取、构建并安装。只需要选择一个目录作为工作空间，然后将GOPATH环境变量设置为该路径。

必要时，Go语言工具会创建目录。例如：

$ export GOPATH=$HOME/gobook # 选择工作目录

$ go get gopl.io/ch1/helloworld # 获取/编译/安装

$ $GOPATH/bin/helloworld # 运行程序

hello, 世界 # 这是中文

运行这些例子需要安装Go1.5以上的版本。

$ go version

go version go1.5 linux/amd64

如果使用其他的操作系统，请参考<https://golang.org/doc/install>提供的说明安装

## 更多的信息

最佳的帮助信息来自Go语言的官方网站，<https://golang.org>，它提供了完善的参考文档，包括编程语言规范和标准库等诸多权威的帮助信息。同时也包含了如何编写更地道的Go程序的基本教程，还有各种各样的在线文本资源和视频资源，它们是本书最有价值的补充。Go语言的官方博客<https://blog.golang.org会不定期发布一些Go>语言最好的实践文章，包括当前语言的发展状态、未来的计划、会议报告和Go语言相关的各种会议的主题等信息（译注：<http://talks.golang.org>包含了官方收录的各种报告的讲稿）。

在线访问的一个有价值的地方是可以从web页面运行Go语言的程序（而纸质书则没有这么便利了）。这个功能由来自<https://play.golang.org的Go> Playground提供，并且可以方便地嵌入到其他页面中，例如<https://golang.org>的主页，或godoc提供的文档页面中。

Playground可以简单地通过执行一个小程序来测试对语法、语义和对程序库的理解，类似其他很多语言提供的REPL即时运行的工具。同时它可以生成对应的url，非常适合共享Go语言代码片段，汇报bug或提供反馈意见等。

基于Playground构建的Go Tour， <https://tour.golang.org>，是一个系列的Go语言入门教程，它包含了诸多基本概念和结构相关的并可在线运行的互动小程序。

当然，Playground和Tour也有一些限制，它们只能导入标准库，而且因为安全的原因对一些网络库做了限制。如果要在编译和运行时需要访问互联网，对于一些更复杂的实验，你可能需要在自己的电脑上构建并运行程序。幸运的是下载Go语言的过程很简单，从<https://golang.org>下载安装包应该不超过几分钟（译注：感谢伟大的长城，让大陆的Gopher们都学会了自己打洞的基本生活技能，下载时间可能会因为洞的大小等因素从几分钟到几天或更久），然后就可以在自己电脑上编写和运行Go程序了。

Go语言是一个开源项目，你可以在<https://golang.org/pkg>阅读标准库中任意函数和类型的实现代码，和下载安装包的代码完全一致。这样你可以知道很多函数是如何工作的，通过挖掘找出一些答案的细节，或者仅仅是出于欣赏专业级Go代码。

## 致谢

[Rob Pike](http://genius.cat-v.org/rob-pike/)和[Russ Cox](http://research.swtch.com/)，以及很多其他Go团队的核心成员多次仔细阅读了本书的手稿，他们对本书的组织结构和表述用词等给出了很多宝贵的建议。在准备日文版翻译的时候，Yoshike Shibata更是仔细地审阅了本书的每个部分，及时发现了诸多英文和代码的错误。我们非常感谢本书的每一位审阅者，并感谢对本书给出了重要的建议的Brian Goetz、Corey Kosak、Arnold Robbins、Josh Bleecher Snyder和Peter Weinberger等人。

我们还感谢Sameer Ajmani、Ittai Balaban、David Crawshaw、Billy Donohue、Jonathan、Feinberg、AndrewGerrand、Robert Griesemer、John Linderman、Minux Ma（译注：中国人，Go团队成员。）、Bryan Mills、Bala Natarajan、Cosmos Nicolaou、Paul Staniforth、Nigel Tao（译注：好像是陶哲轩的兄弟）以及Howard Trickey给出的许多有价值的建议。我们还要感谢David Brailsford和Raph Levien关于类型设置的建议。

我们的来自Addison-Wesley的编辑Greg Doench收到了很多帮助，从最开始就得到了越来越多的帮助。来自AW生产团队的John Fuller、Dayna lsley、Julie Nahil、Chuti Prasertsith到Barbara Wood，感谢你们的热心帮助。

[Alan Donovan](https://github.com/adonovan/gopl.io/)特别感谢：Sameer Ajmani、Chris Demetriou、Walt Drummond和Google公司的Reid Tatge允许他有充裕的时间去写本书；感谢Stephen Donovan的建议和始终如一的鼓励，以及他的妻子Leila Kazemi并没有让他为了家庭琐事而分心，并热情坚定地支持这个项目。

[Brian Kernighan](http://www.cs.princeton.edu/~bwk/)特别感谢：朋友和同事对他的耐心和宽容，让他慢慢地梳理本书的写作思路。同时感谢他的妻子Meg和其他很多朋友对他写作事业的支持。

2015年10月 于 纽约

## 入门

本章介绍Go语言的基础组件。本章提供了足够的信息和示例程序，希望可以帮你尽快入门，写出有用的程序。本章和之后的章节的示例程序都针对你可能遇到的现实案例。先了解几个Go程序，设计的主题从简单的文本处理、图像处理到互联网客户端和服务端并发。当然，第一章不会解释细枝末节，但用这些程序来学习一门新语言还是很有效的。

学习一门新语言时，会有一种自然的倾向，按照自己熟悉的语言的套路写新语言程序。学习Go语言的过程中，请警惕这种想法，尽量别这么做。我们会演示怎么写好Go语言程序，所以请使用本书的代码作为你自己写程序时的指南。

### Hello, World

我们以现已成为传统的“hello world”案例来开始吧，这个例子首次出现于1978年出版的C语言圣经《The C Programming Language》。C语言是直接影响Go语言设计的语言之一。这个例子体现了Go语言的一些核心理念。

gopl.io/ch1/helloworld

package main

import "fmt"

func main() {

fmt.Println("hello, 世界")

}

Go是一门编译型语言，Go语言的工具链将源代码及其依赖转换成计算机的机器指令。Go语言提供的工具都通过一个单独的命令go调用，go命令有一系列子命令。最简单的一个子命令就是run。这个命令编译一个或多个以.go结尾的源文件，链接库文件，并运行最终生成都可执行文件。（本书使用$表示命令行提示符。）

go run helloworld.go

毫无意外，这个命令行输出：

hello, 世界

Go语言原生支持Unicode，你可以处理全世界任何语言的文本。

如果不只是一次性实验，你肯定希望能够编译这个程序，保存编译结果以备将来之用。可以用build子命令：

go build helloworld.go

这个命令生成一个名为helloworld的可执行的二进制文件，之后你可以随时运行它，不需任何处理。

./helloworld

hello, 世界

本书中，所有的示例代码上都有一行标记，利用这些标记，可以从gopl.io网站上本书源码仓库里获取代码：

gopl.io/ch1/helloworld

执行go get gopl.io/ch1/helloworld命令，就会从网上获取代码，并放到对应目录中。2.6和10.7节有这方面更详细的介绍。

来讨论下程序本身。Go语言的代码通过包（package）组织，包类似于其他语言里的库（libraries）或者模块（modules）。一个包由位于单个目录下的一个或多个.go源代码文件组成，目录定义包的作用。每个源文件都以一条package声明语句开始，这个例子里就是package main，表示该文件属于哪个包，紧跟着一系列导入（import）的包，之后是存储在这个文件里的程序语句。

Go的标准库提供了100多个包，以支持常见功能，如输入、输出、排序以及文本处理。比如fmt包，就含有格式化输出、接收输入的函数。Println是其中一个基础函数，可以打印以空格间隔的一个或多个值，并在最后添加一个换行符，从而输出一整行。

main包比较特殊。它定义了一个独立可执行的程序，而不是一个库。在main里的main函数也很特殊，它是整个程序执行时的入口。main函数所做的事情就是程序做的。当然了，main函数一般调用其他包里的函数完成很多工作，比如fmt.Println。

必须告诉编译器源文件需要哪些包，缺少了必要的包或者导入了不需要的包，程序都无法编译通过。这项严格要求避免了程序开发过程中引入未使用的包。

import声明必须跟在文件的package声明之后。随后，则是组成程序的函数、变量、常量、类型的声明语句（分别由关键字func，var，const，type定义）。这些内容的声明顺序并不重要。这个例子的程序已经尽可能短了，只声明了一个函数，其中只调用了一个其他函数。为了节省篇幅，有些时候，示例程序会省略package和import声明，但是，这些声明在源代码里有，并且必须得有才能编译。

一个函数的声明由func关键字、函数名、参数列表、返回值列表（这个例子里的main函数参数列表和返回值都是空的）以及包含在大括号里的函数体组成。第五章进一步考察函数。

Go语言不需要在语句或者声明的末尾添加分号，除非一行上有多条语句。实际上，编译器会主动把特定符号后的换行符转换为分号，因此换行符添加的位置会影响Go代码的正确解析。举个例子，函数的左括号“{”必须和func函数声明在同一行上，且位于末尾，不能独占一行，而在表达式x+y中，可在+后换行，不能在+前换行。

Go语言在代码格式上采取了很强硬的态度。gofmt工具把代码格式化为标准格式，并且go工具中的fmt子命令会对指定包，否则默认为当前目录中的所有.go源文件应用gofmt命令。本书中的所有代码都被gofmt过。你也应该养成格式化自己的代码的习惯。以法令方式规定标准的代码格式可以避免无尽的无意义的琐碎争执。更重要的是，这样可以做多种自动源码转换，如果放任Go语言代码格式，这些转换就不大可能了。

很多文本编辑器都可以配置为保存文件时自动执行gofmt，这样你的源代码总会被恰当的格式化。还有个相关的工具goimports，可以根据代码需要，自动地添加或删除import声明。这个工具并没有包含在标准的分发包中，可以用下面的命令安装：

go get golang.org/x/tools/cmd/goimports

对于大多数用户来说，下载、编译包、运行测试用例、查看Go语言的文档等等常用功能都可以用go的工具完成。10.7节详细介绍这些知识。

### 命令行参数

大多数的程序都是处理输入，产生输出；这也正是“计算”的定义。但是，程序如何获取要处理的出入数据呢？一些程序生成自己的数据，但通常情况下，输入来自于程序内部：文件、网络连接、其他程序的输出、敲键盘的用户、命令行参数或其他类似输入源。下面几个例子会讨论其中几个输入源，首先是命令行参数。

os包以跨平台的方式，提供了一些与操作系统交互的函数和变量。程序的命令行参数可从os包的Args变量获取；os包外部使用os.Args访问该变量。

os.Args变量是一个字符串（string）的切片（slice）（译注：slice和Python语言中的切片类似，是一个简版的动态数组），切片是Go语言的基础概念，稍后详细介绍。现在先把切片s当作数组元素序列，序列的成长度动态变化，用s[i]访问单个元素，用s[m:n]获取子序列（译注：和Python里的语法差不多）。序列的元素数目为len(s)。和大多数编程语言类似，区间索引时，Go语言里也采用左闭右开形式，即区间包括第一个索引元素，不包括最后一个，因为这样可以简化逻辑。（译注：比如a=[1,2,3,4,5]，a[0:3]=[1,2,3]，不包含最后一个元素）。比如s[m:n]这个切片，0≦m≦n≦len(s)，包含n-m个元素。

os.Args的第一个元素，os.Args[0]是命令本身的名字；其他的元素则是程序启动时传给它的参数。s[m:n]形式的切片表达式，产生从第m个元素到第n-1个元素的切片，下个例子用到的元素包含在os.Args[1:len(os.Args)切片中。如果省略切片表达式的m或n，会默认传入0或len(s)，因此前面的切片可以简写成os.Args[1:]。

下面是Unix里echo命令的一份实现，echo把它的命令行参数打印成一行。程序导入了两个包，用括号把它们括起来写成列表形式，而没有分开写成独立的import声明。两种形式都合法，列表形式习惯上用的多。包导入顺序并不重要；gofmt工具格式化时按照字母顺序对报名排序。（示例有多个版本时，我们会对示例编号，这样可以明确当前正在讨论的是哪个。）

gopl.io/ch1/echo1

// Echo1 prints its command-line arguments

package main

import (

"fmt"

"os"

)

func main() {

var s, sep string

for i:=1; i < len(os.Args); i++ {

s += sep + os.Args[i]

sep = " "

}

fmt.Println(s)

}

注释语句以//开头。对于程序员来说，//之后到行末之间所有的内容都是注释，被编译器忽略。按照惯例，我们在每个包的包声明前添加注释；对于main package，注释包含一句或几句话，从整体角度对程序做个描述。

var声明定义了两个string类型的变量s和sep。变量会在声明时直接初始化。如果变量没有显式初始化，则被隐式地赋予其类型的零值（zero value），数值类型是0，字符串类型是空字符串“”。这个例子里，声明把s和sep隐式地初始化成空字符串。第2章再来详细地讲解变量和声明。

对数值而言，Go语言提供了常规的数值和逻辑运算符。而对string类型，+运算符连接字符串（译注：和C++或者js是一样的）。所以表达式：

sep + os.Args[i]

表示连接字符串sep和os.Args。程序中使用的语句：

s += sep + os.Args[i]

是一条赋值语句，将s的旧值跟sep与os.Args[i]连接后赋值回s，等价于：

s = s + sep + os.Args[i]

运算符 += 是赋值运算符（assignment operator），每种数值运算符或逻辑运算符，如+或\*，都有对应的赋值运算符。

echo程序可以每循环一次输出一个参数，这个版本却是不断的把新文本追加到末尾来构造字符串。字符串s开始为空，即值为“”，每次循环会添加一些文本；第一次迭代之后，还会再插入一个空格，因此循环结束时每个参数中间都有一个空格。这是一种二次加工（quadratic process），当参数数量庞大时，开销很大，但是对于echo，这种情形不大可能出现。本章会介绍echo的若干改进版，下一章解决低效问题。

循环索引变量i在for循环的第一部分中定义。符号:=是短变量声明（short variable declaration）的一部分，这是定义一个或多个变量并根据它们的初始值为这些变量赋予适当类型的语句。下一章有这方面更多说明。

自增语句i++给i加1；这和i+=1以及i=i+1都是等价的。对应的还有i--给i减1。它们是语句，而不像C系的其他语言那样是表达式。所以j = i++非法，而且++和--都只能放在变量名后面，因--i也非法。

Go语言只有for循环这一种循环语句。for循环有多种形式，其中一种如下所示：

for initialization; condition; post {

// zero or more statements

}

for循环三个部分不需括号包围。大括号强制要求，左大括号必须和post语句在同一行。

initialization语句是可选的，在循环开始前执行。initialization如果存在，必须是一条简单语句（simple statement），即短变量声明、自增语句、赋值语句或函数调用。condition是一个布尔表达式（boolean expression），其值在每次循环迭代开始时计算。如果为true则执行循环体语句。post语句在循环体执行结束后执行，之后再次对condition求值。condition值为false时，循环结束。

for循环的这三个部分每个都可以省略，如果省略initialization和post，分号也可以省略：

// a traditional “white” loop

for condition {

// …

}

如果连condition也省略了，像下面这样：

// a traditional infinite loop

for {

// …

}

这就变成一个无限循环，尽管如此，还可以用其他方式终止循环，如一条break或return语句。

for循环的另一种形式，在某种数据类型的区间（range）上遍历，如字符串或切片。echo的第二版本展示了这种形式：

gopl.io/ch1/echo2

// Echo2 prints its command-line arguments

package main

import (

"fmt"

"os"

)

func main() {

s, sep := "", ""

for \_, arg := range os.Args[1:] {

s += sep + arg

sep = " "

}

fmt.Println(s)

}

每次循环迭代，range产生一对值；索引以及在该索引处的元素值。这个例子不需要索引，但range的语法要求要处理元素，必须处理索引。一种思路是把索引赋值给一个临时变量，如temp，然后忽略它的值，当Go语言不允许使用无用的局部变量（local variables），因为这会导致编译错误。

Go语言中这样情况的解决方法是用空标志符（blank identifier），即\_（也就是下划线）。空标识符可用于任何语法需要变量名但程序逻辑不需要的时候，例如，在循环里，丢弃不需要的循环索引，保留元素值。大多数的Go程序员都会像上面这样使用range和\_写echo程序，因为隐式地而非显示地索引os.Args，容易写对。

echo的这个版本使用一条短变量声明来声明并初始化s和seps，也可以将这两个变量分开声明，声明一个变量有好几种方式，下面这些都等价：

s := “”

var s string

var s = “”

var s string = “”

用哪种不用哪种，为什么呢？第一种形式，是一条短变量声明，最简洁，但只能用在函数内部，而不能用于包变量。第二种形式依赖于字符串的默认初始化零值机制，被初始化为“”。第三种形式用的很少，除非同时声明多个变量。第四种形式显式地标明变量的类型，当变量类型与初始类型相同时，类型冗余，但如果两者类型不同，变量类型就必须了。实践中一般使用前两种形式中的某个，初始值重要的话就显式地指定变量的类型，否则使用隐式初始化。

如前文所述，每次循环迭代字符串s的内容都会更新。+=连接原字符串、空格和下个参数，产生新字符串，并把它赋值给s。s原来的内容已经不再使用，将在适当时机对它进行垃圾回收。

如果连接涉及的数据量很大，这种方式代价高昂。一种简单且高效的解决方案是使用strings包的Join函数：

gopl.io/ch1/echo3

// Echo3 prints its command-line arguments

package main

import (

"fmt"

"os"

"strings"

)

func main() {

fmt.Println(strings.Join(os.Args[1:], " "))

}

最后，如果不关心输出格式，只想看看输出值，或许只是为了调试，可以用Println为我们格式化输出。

fmt.Println(os.Args[1:])

这条语句的输出结果跟strings.Join得到的结果很像，只是被放到了一对方括号里。切片都会被打印成这种格式。

**练习1.1：**修改echo程序，使其能够打印os.Args[0]，即被执行命令本身的名字。

**练习1.2：**修改echo程序，使其打印每个参数的索引和值，每个一行。

**练习1.3：**做实验测量潜在低效的版本和使用了strings.Join的版本的运行时间差异。（1.6节讲解了部分time包，11.4节展示了如何写标准测试程序，以得到系统性的性能评测。）

### 查找重复的行

对文件做拷贝、打印、搜索、排序、统计或类似事情的程序都有一个差不多的程序结构：一个处理输入的循环，在每个元素上执行计算处理，在处理的同时或最后产生输出。我们会展示一个名为dup的程序的三个版本；灵感来自于Unix的uniq命令，其寻找相邻的重复行。该程序使用的结构和包是个参考范例，可以方便地修改。

dup的第一个版本打印标准输入中多次出现的行，以重复次数开头。该程序将引入if语句，map数据类型以及bufio包。

gopl.io/ch1/dup1

// Dup1 prints the text of each line that appears more than

// once in the standard input,preceded by its count.

package main

import (

"bufio"

"fmt"

"os"

)

func main() {

counts := make(map[string]int)

input := bufio.NewScanner(os.Stdin)

for input.Scan() {

counts[input.Text()]++

}

// NOTE: ignoring potential errors from input.Err()

for line, n := range counts {

if n > 1 {

fmt.Printf("%d\t%s\n", n, line)

}

}

}

正如for循环一样，if语句条件两边也不加括号，但是主体部分需要加。if语句的else部分是可选的，在if的条件为false时执行。

map存储了键/值（key/value）的集合，对集合元素，提供常数时间的存、取或测试操作。键可以是任何类型，只要其值能用==运算符比较，最常见的例子是字符串；值则可以是任意类型。这个例子中的键是字符串，值是整数。内置函数make创建空map，此外，它还有别的作用。4.3节讨论map。

（译注：从功能和实现上说，Go的map类似于Java语言中的HashMap，Python语言中的dict，Lua语言中的table，通常使用hash实现。遗憾的是，对于该词的翻译并不统一，数学界术语为映射，而计算机界众说纷纭莫衷一是。为了防止对读者造成误解，保留不译。）

每次dup读取一行输入，该行被当做map，其对应的值递增。counts[input.Text()]++语句等价下面两句：

line := input.Text()

counts[line] = counts[line] + 1

map中不含某个键时不用担心，首次读到新行时，等号右边的表达式counts[line]的值将被计算为其类型的零值，对于int即0.

为了打印结果，我们使用了基于range的循环，并在counts这个map上迭代。跟之前类似，每次迭代得到两个结果，键和其在map中对应的值。map的迭代顺序并不确定，从实践来看，该顺序随机，每次运行都会变化。这种设计是有意为之的，因为能防止程序依赖特定遍历顺序，而这是无法保证的。

继续来看bufio包，它使处理输入和输出方便又高效。Scanner类型是该包最有用的特性之一，它读取输入并将其拆成行或单词；通常是处理行形式的输入最简单的方法。

程序使用短变量声明创建bufio.Scanner类型的变量input。

input := bufio.NewScanner(os.Stdin)

该变量从程序的标准输入中读取内容。每次调用input.Scanner，即读入下一行，并移除行末的换行符；读取的内容可以调用input.Text()得到。Scan函数在读到一行时返回true，在无输入时返回false。

类似于C或其他语言里的printf函数，fmt.Printf函数对一些表达式产生格式化输出。该函数的首个参数是个格式字符串，指定后续参数被如何格式化。各个参数的格式取决于“转换字符”（conversion character），形式为百分号后跟一个字母。举个例子，%d表示以十进制形式打印一个整形操作数，而%s则表示把字符串操作数的值展开。

Printf有一大堆这种转换，Go程序员称之为动词（verb）。下面的表格虽然远不是完整的规范，但展示了可用的很多特性：

|  |  |
| --- | --- |
| %d | 十进制整数 |
| %x, %o, %b | 十六进制，八进制，二进制整数。 |
| %f, %g, %e | 浮点数：3.141593，3.141592653589793，3.141593e+∞ |
| %t | 布尔：true或false |
| %c | 字符（rune）（Unicode码点） |
| %s | 字符串 |
| %q | 带双引号的字符串“abc”或带单引号的字符‘c’ |
| %v | 变量的自然形式（natural format） |
| %T | 变量的类型 |
| %% | 字面上的百分号标志（无操作数） |

dup1的格式字符串中还含有制表符\t和换行符\n。字符串字面上可能含有这些代表不可见字符的转义字符（escap sequences）。默认情况下，Printf不会换行。按照惯例，以字母f结尾的格式化函数，如log.Printf和fmt.Errorf，都采用fmt.Printf的格式化准则。而以ln结尾的格式化函数，则遵循Println的方式，以跟%v差不多的方式格式化参数，并在最后添加一个换行符。（译注：后缀f指format，ln指line）

很多程序要么从标准输入中读取数据，如上面的例子所示，要么从一系列具名文件中读取数据。dup程序的下个版本读取标准输入是使用os.Open打开各个具名文件，并操作它们。

gopl.io/ch1/dup2

// Dup2 prints the count and text of lines that appear more than once

// in the input. It reads from stdin or from a list of named files.

package main

import (

"bufio"

"fmt"

"os"

)

func main() {

counts := make(map[string]int)

files := os.Args[1:]

if len(files) == 0 {

countLines(os.Stdin, counts)

} else {

for \_, arg := range files {

f, err := os.Open(arg)

if err != nil {

fmt.Fprintf(os.Stderr, "dup2: %v\n", err)

continue

}

countLines(f, counts)

f.Close()

}

}

for line, n := range counts {

if n > 1 {

fmt.Printf("%d\t%s\n", n, line)

}

}

}

func countLines(f \*os.File, counts map[string]int) {

input := bufio.NewScanner(f)

for input.Scan() {

counts[input.Text()]++

}

// NOTE: ignoring potential errors from input.Err()

}

os.Open函数返回两个值。第一个值是被打开的文件（\*os.File），其后被Scanner读取。

os.Open返回的第二个值是内置error类型的值。如果err等于内置值nil（译注：相当于其他语言里的NULL），那么文件被成功打开。读取文件，直到文件结束，然后调用Close关闭该文件，并释放占用的所有资源。相反的话，如果err的值不是nil，说明打开文件时出错了。这种情况下，错误值描述了所遇到的问题。我们的错误处理非常简单，只是使用Fprintf与表示任意类型默认格式值的动词%v，向标准错误流打印一条信息，然后dup继续处理下一个文件；continue语句直接跳到for循环的下个迭代开始执行。

为了使示例代码保持合理的大小，本书开始的一些示例有意简化了错误处理，显而易见的是，应该检查os.Open返回的错误值，然而，使用input.Scan读取文件过程中，不大可能出现错误，因此我们忽略了错误处理。我们会在跳过错误检查的地方做说明。5.4节中深入介绍错误处理。

注意contLines函数在其声明前被调用。函数和包级别的变量（package-level entities）可以任意顺序声明，并不影响其被调用。（译注：最好还是遵循一定的规范）

map是一个由make函数创建的数据结构的引用。map作为参数传递给某函数时，该函数接收这个引用的一份拷贝（copy，或译为副本），被调用函数对map底层数据结构的任何修改，调用者函数都可以通过持有的map引用看到。在我们的例子中，coutLines函数向counts插入的值，也会被main函数看到。（译注：类似于C++里的引用传递，实际上指针是另一个指针了，但内部存的值指向同一块内存）

dup的前两个版本以“流”模式读取输入，并根据需要拆分成多个行。理论上，这些程序可以处理任意数量的输入数据。还有另一个方法，就是一口气把全部输入数据读到内存中，一次分割为多行，然后处理它们。下面这个版本，dup3，就是这么操作的。这个例子引入了ReadFile函数（来自于io/ioutil包），其读取指定文件的全部内容，strings.Split函数把字符串分割成子串的切片。（Split的作用与前文提到的strings.Join相反。）

我们略微简化了dup3。首先，由于ReadFile函数需要文件名作为参数，因此只读指定文件，不读标准输入。其次，由于行计数代码只在一处用到，故将其移回main函数。

gopl.io/ch1/dup3

package main

import (

"fmt"

"io/ioutil"

"os"

"strings"

)

func main() {

counts := make(map[string]int)

for \_, filename := range os.Args[1:] {

data, err := ioutil.ReadFile(filename)

if err != nil {

fmt.Fprintf(os.Stderr, "dup3: %v\n", err)

continue

}

for \_, line := range strings.Split(string(data), "\n") {

counts[line]++

}

}

for line, n := range counts {

if n > 1 {

fmt.Printf("%d\t%s\n", n, line)

}

}

}

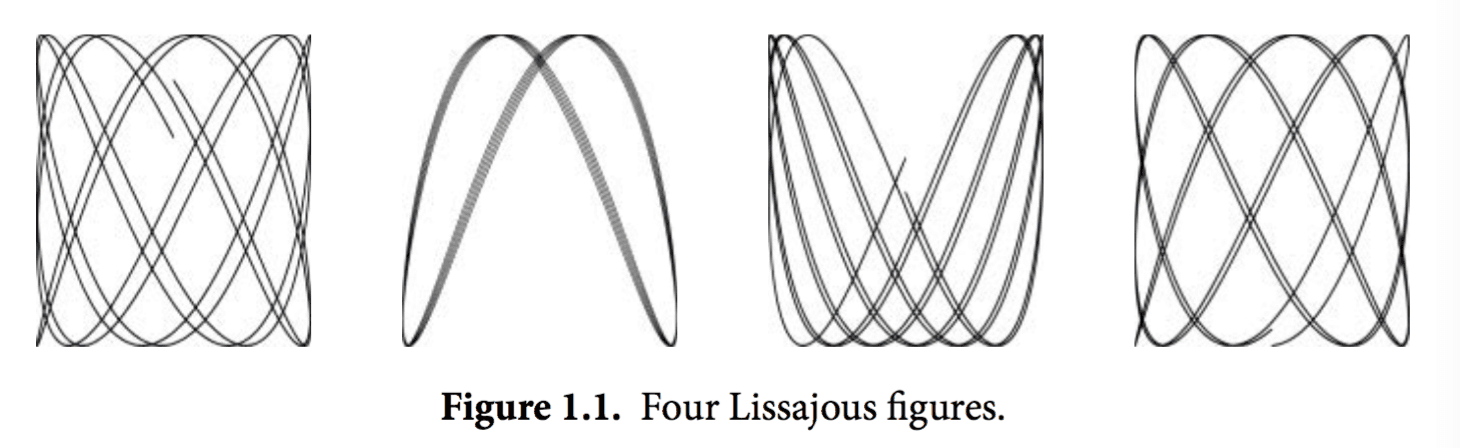
ReadFile函数返回一个字节切片（byte slice），必须把它转换为string，才能用strings.Split分割。我们会在3.5.4节详细讲解字符串和字节切片。

实现上，bufio.Scanner、ioutil.ReadFile和ioutil.WriteFile都使用\*os.File的Read和Write方法，但是，大多数程序员还少需要直接调用那些低级（lower-level）函数。高级（higher-level）函数，像bufio和io/ioutil包中所提供的那些，用起来要容易点。

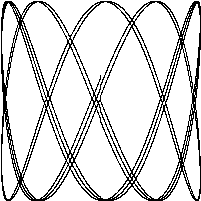
**练习1.4：**修改dup2，出现重复的行时打印文件名称。

### GIF动画

下面的程序会演示Go语言标准库里的image这个package的用法，我们会用这个包来生成一系列的bit-mapped图，然后将这些图片编码为一个GIF动画。我们生成的图形名字叫利萨如图形（Lissajous figures），这种效果是在1960年代的老电影里出现的一种视觉特效。它们是协振子在两个纬度上振动所产生的曲线，比如两个sin正弦波分别在x轴和y轴输入会产生的曲线。图1.1是这样的一个例子：



译注：要看这个程序的结果，需要将标准输出重定向到一个GIF图像文件（使用./Lissajous > output.gif命令）。下面是GIF图像动画效果：



这段代码里我们用了一些新的结构，包括const声明，struct结构体类型，复合声明。和我们举的其它的例子不太一样，这一个例子包含了浮点数运算。这些概念我们只在这里简单地说明一下，之后的章节会更详细地讲解。

gopl.io/ch1/Lissajous

// Lissajous generates GIF animations of random Lissajous figures.

package main

import (

"image"

"image/color"

"image/gif"

"io"

"math"

"math/rand"

"os"

)

var palette = []color.Color{color.White, color.Black}

const (

whiteIndex = 0 // first color in palette

blackIndex = 1 // next color in palette

)

func main() {

lissajous(os.Stdout)

}

func lissajous(out io.Writer) {

const (

cycles = 5 //number of complete x oscillator revolutions

res = 0.001 // angular resolution

size = 100 // image canvas covers [-size..+size]

nframes = 64 // number of animation frames

delay = 8 //delay between frames in 10ms units

)

freq := rand.Float64() \* 3.0 // relative frequency of y oscillator

anim := gif.GIF{LoopCount: nframes}

phase := 0.0 // phase difference

for i := 0; i < nframes; i++ {

rect := image.Rect(0, 0, 2\*size+1, 2\*size+1)

img := image.NewPaletted(rect, palette)

for t := 0.0; t < cycles\*2\*math.Pi; t += res {

x := math.Sin(t)

y := math.Sin(t\*freq + phase)

img.SetColorIndex(size+int(x\*size+0.5), size+int(y\*size+0.5), blackIndex)

}

phase += 0.1

anim.Delay = append(anim.Delay, delay)

anim.Image = append(anim.Image, img)

}

gif.EncodeAll(out, &anim) // NOTE: ignoring encoding errors

}

当我们import了一个包路径包含有多个单词的package时，比如image/color（image和color两个单词），通常我们只需要用最后那个单词表示这个包就可以。所以当我们写color.White时，这个变量指向的是image/color包里的变量，同理gif.GIF是属于image/gif包里的变量。

这个程序里的常量声明给出了一系列的常量值，常量是值在程序编译后运行时始终都不会变化的值，比如圈数、帧数、延迟值。常量声明和变量声明一般都会出现在包级别，所以这些常量在整个包中都是可以共享的，或者你也可以把常量声明定义在函数体内部，那么这种常量就只能在函数体内用。目前常量声明的值必须是一个数字值、字符串或者一个固定的boolean值。

[]color.Color{…}和gif.GIF{…}这两个表达式就是我们说的复合声明（4.2和4.4.1节有说明）。这是示例化Go语言里的复合类型的一种写法。这里的前者生成的是一个slice切片，后者生成的是一个struct结构体。

gif.GIF是一个struct类型（参考4.4节）。struct是一组值或者叫字段的集合，不同的类型集合在一个struct可以让我们以一个统一的单元进行处理。anim是一个gif.GIF类型的struct变量。这种写法会生成一个struct变量，并且其内部变量LoopCount字段会被设置为nframes；而其他的字段会被设置为各自类型默认的零值。struct内部的变量可以以一个点（.）来进行访问，就像在最后两个赋值语句中显式地更新了anim这个struct的Delay和Image字段。

lissajous函数内部有两层嵌套的for循环。外层循环会循环64次，每一次都会生成一个单独的动画帧。它生成了一个包含两种颜色的201&201大小的图片，白色和黑色。所有像素点都会被默认设置为其零值（也就是调色板palette里的第0个值），这里我们设置的是白色。每次外层循环都会生成一张新图片，并将一些像素设置为黑色。其结果会append到anim中的帧列表末尾，并设置一个默认的80ms的延迟值。循环结束后所有的延迟值被编码进了GIF图片中，并将结果写入到输出流。out这个变量是io.Writer类型，这个类型支持把输出结果写到很多目标，很快我们就可以看到例子。

内层循环设置两个偏振值。x轴偏振使用sin函数。y轴偏振也是正弦波，但其相对x轴的偏振是一个0-3的随机值，初始偏振值是一个零值，随着动画的每一帧逐渐增加。循环会一直跑到x轴完成五次完整的循环。每一步它都会调用SetColorIndex来为（x，y）点来染黑色。

main函数调用lissajous函数，用它来想标准输出流打印信息，所以下面这个命令会像图1.1中产生一个GIF动画。

go build gopl.io/ch1/Lissajous

./Lissajous > out.gif

**练习1.5：**修改前面的Lissajous程序里的调色板，由黑色改为绿色。我们可以用color.RGBA{0xRR, 0xGG, 0xBB, 0xff}来得到#RRGGBB这个色值，三个十六进制的字符串分别代表红、绿、蓝像素。

**练习1.6：**修改Lissajous程序，修改其调色板来生成更丰富的颜色，然后修改SetColorIndex的第三个参数，看看显示结果吧。

### 获取URL

对于很多现代应用来说，访问互联网上的信息和访问本地文件系统一样重要。Go语言在net这个强大package的帮助下提供了一系列的package来做这件事件，使用这些包可以更简单地用网络收发信息，还可以建立更底层的网络连接，编写服务器程序。在这些情景下，Go语言原生的并发特性（在第八章中会介绍）显得尤其好用。

为了最简单地展示基于HTTP获取信息的方式，下面给出一个实例程序fetch，这个程序将获取对应的url，并将其源文件打印出来；这个例子的灵感来源于curl工具（译注：unix下的一个用来发http请求的工具，具体可以man curl）。当然，curl提供的功能更为复杂丰富，这里只编写最简单的样例。这个样例之后还会多次被用到。

gopl.io/ch1/fetch

// Fetch prints the content found at a URL.

package main

import (

"fmt"

"io/ioutil"

"net/http"

"os"

)

func main() {

for \_, url := range os.Args[1:] {

resp, err := http.Get(url)

if err != nil {

fmt.Fprintf(os.Stderr, "fetch: %v\n", err)

os.Exit(1)

}

b, err := ioutil.ReadAll(resp.Body)

resp.Body.Close()

if err != nil {

fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)

os.Exit(1)

}

fmt.Printf("%s", b)

}

}

这个程序从两个package中导入了函数，net/http和io/ioutil包，http.Get函数是创建HTTP请求的函数，如果获取过程没有出错，那么会在resp这个结构体中得到访问的请求结果。resp的Body字段包括一个可读的服务器响应流。ioutil.ReadAll函数从response中读取到全部内容；将其结果保存在变量b中。resp.Body.Close关闭resp的Body流，防止资源泄露，Printf函数会将结果b写出到标准输出流中。

go build gopl.io/ch1/fetch

./fetch <http://gopl.io>

<html>

<head>

<title>The Go Programing Language</tital>

…

HTTP请求如果失败了的话，会得到下面这样的结果：

.fetch <http://bad.gopl.io>

fetch: Get <http://bad.gopl.io>: dial tcp: lookup bad.gopl.io: no such host

译注：在大天朝的网络环境下很容易重现这种错误，下面是windows下运行得到的错误信息：

go run main.go <http://gopl.io>

fetch: Get http://gopl.io : dial tcp: lookup gopl.io: getaddrinfow: no such host

无论哪种失败原因，我们的程序都用了os.Exit函数来终止进程，并且返回一个status错误码，其值为1.

**练习1.7：**函数调用io.Copy(dst, src)会从src中读取内容，并将读取的结果写入到dst中，使用这个函数代替掉例子中的ioutil.ReadAll来拷贝响应结构体到os.Stdout，避免申请一个缓冲区（例子中的b）来存储。记得处理io.Copy返回结果中的错误。

**练习1.8：**修改fetch这个范例，如果输入的url函数没有http://前缀的话，为这个url加上该前缀。你可能会用到strings.HasPrefix这个函数。

**练习1.9：**修改fetch打印出HTTP协议的状态码，可以从resp.Status变量得到该状态码。

### 并发获取多个URL

Go语言最有意思并且最新奇的特性就是对并发编程的支持。并发编程是一个大话题，在第八章和第九章中会专门讲到。这里我们只浅尝辄止地来体验一下Go语言里的goroutine和channel。

下面的例子fetchall，和前面小节的fetch程序所要做的工作基本一致，fetchall的特别之处在于它会同时去获取所有的URL，所以这个程序的总执行时间不会超过执行时间最长的那一个任务，前面的fetch程序执行时间则是所有任务执行时间之和。fetchall程序只会打印获取的内容大小和经过的时间，不会像之前那样打印获取的内容。

gopl.io/ch1/fetchall

// Fetchall fetches URLs in parallel and reports their times and sizes

package main

import (

"fmt"

"io"

"io/ioutil"

"net/http"

"os"

"time"

)

func main() {

start := time.Now()

ch := make(chan string)

for \_, url := range os.Args[1:] {

go fetch(url, ch) // start a goroutine

}

for range os.Args[1:] {

fmt.Println(<-ch) // receive from channel ch

}

fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())

}

func fetch(url string, ch chan<- string) {

start := time.Now()

resp, err := http.Get(url)

if err != nil {

ch <- fmt.Sprint(err) // send to channel ch

return

}

nbytes, err := io.Copy(ioutil.Discard, resp.Body)

resp.Body.Close() // don't leak resources

if err != nil {

ch <- fmt.Sprintf("while reading %s: %v", url, err)

return

}

secs := time.Since(start).Seconds()

ch <- fmt.Sprintf("%.2fs %7d %s", secs, nbytes, url)

}

下面使用fetchall来请求几个地址：

go build gopl.io/ch1/fetchall

./fetchall <https://golang.org> <http://gopl.io> <https://godoc.org>

0.14s 6852 <https://godoc.org>

0.16s 7261 <https://golang.org>

0.48s 2475 <http://gopl.io>

0.48s elapsed

goroutine是一种函数的并发执行方式，而channel是用来在goroutine之间进行参数传递。main函数本身也运行在一个goroutine中，而go function则表示创建一个新的goroutine，并在这个新的goroutine中执行这个函数。

main函数中用make函数创建了一个传递string类型参数的channel，对每一个命令行参数，我们都用go这个关键字来创建一个goroutine，并且让函数在这个goroutine异步执行http.Get方法。这个程序里的io.Copy会把相应的Body内容拷贝到ioutil.Discard输出流中（译注：可以把这个变量看作一个垃圾桶，可以向里面写一些不需要的数据），因为我们需要这个方法返回的字节数，但是又不想要其内容。每当请求返回内容时，fetch函数都会往这个channel里写入一个字符串，由main函数里的第二个循环来处理并打印channel里的这个字符串。

当一个goroutine尝试在一个channel上做send或者receive操作时，这个goroutine会阻塞在调用处，直到另一个goroutine往这个channel里写入、或者接收值，这样两个goroutine才会继续执行channel操作之后的逻辑。在这个例子中，每一个fetch函数在执行时都会往channel里发送一个值（ch<-expression），主函数负责接收这些值（<-ch）。这个程序中我们用main函数来接收所有fetch函数传回的字符串，可以避免在goroutine异步执行还没有完成时main函数提前退出。

**练习1.10：**找一个数据量比较大的网站，用本小节中的程序调研网站的缓存策略，对每个URL执行两遍请求，查看两次时间是否有较大的差别，并且每次获取到的响应内容是否一致，修改本节中的程序，将响应结果输出，以便于进行对比。

**练习1.11：**在fetchall中尝试使用长一些的参数列表，比如使用在alexa.com的上百万网站里排名靠前的。如果一个网站没有响应，程序将采取怎样的行为？（Section8.9描述了在这种情况下的应对机制）。

### Web服务

Go语言的内置库使得写一个类似fetch的web服务器变得异常的简单。在本节中，我们会展示一个微型服务器，这个服务器的功能是返回当前用户正在访问的URL。比如用户访问的是<http://localhost:8000/hello>，那么响应是URL.Path=“hello”。

gopl.io/ch1/server1

// Server1 is a minimal "echo" server.

package main

import (

"fmt"

"log"

"net/http"

)

func main() {

http.HandleFunc("/", handle) // each request calls handler

log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

// handler echoes the Path component of the request URL r.

func handle(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)

}

我们只用了八九行代码就实现了一个Web服务程序，这都是多亏了标准库里的方法已经帮我们完成了大量工作。main函数将所有发送到/路径下的请求和handler函数关联起来，/开头的请求其实就是所有发送到当前站点上的请求，服务监听8000端口。发送到这个服务的“请求”是一个http.Request类型的对象，这个对象中包含了请求中的一系列相关字段，其中就包括我们需要的URL。当请求到达服务器时，这个请求会被传给handler函数来处理，这个行数会将/hello这个路径从请求的URL中解析出来，然后把其发送到响应中，这里我们用的是标准输出流的fmt.Fprintf。Web服务会在第7.7节中做更详细的阐述。

让我们在后台运行这个服务程序。如果你的操作系统是Mac OS X或者Linux，那么在运行命令的末尾加上一个&符合，即可让程序简单地跑在后台，windows下可以在另外一个命令行窗口去运行这个程序。

go run src/gopl.io/ch1/server1/main.go &

现在可以通过命令行来发送客户端请求了：

go build gopl.io/ch1/fetch

./fetch <http://localhost:8000>

URL.Path = “/”

./fetch <http://localhost:8000/help>

URL.Path = “/help”

还可以直接在浏览器里访问这个URL，然后得到返回结果，如图1.2：



在这个服务的基础上叠加特性是很容易的。一种比较实用的修改是为访问的url添加某种状态。比如，下面这个版本输出流同样的内容，但是会对请求的次数进行计算；对URL的请求结果会包含各种URL被访问的总次数，直接对/count这个URL的访问要除外。

gopl.io/ch1/server2

// Server2 is a minimal "echo" and counter server.

package main

import (

"fmt"

"log"

"net/http"

"sync"

)

var mu sync.Mutex

var count int

func main() {

http.HandleFunc("/", handler)

http.HandleFunc("/count", counter)

log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

// handler echoes the Path component of the requested URL.

func handler(w http.ResponseWriter, r \*http.Request) {

mu.Lock()

count++

mu.Unlock()

fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)

}

// counter echoes the number of calls so far.

func counter(w http.ResponseWriter, r \*http.Request) {

mu.Lock()

fmt.Fprintf(w, "Count %d\n", count)

mu.Unlock()

}

这个服务器有两个请求处理函数，根据请求的url不同会调用不同的函数：对/count这个url的请求会调用到count这个函数，其它的url都会调用默认的处理函数。如果你的请求pattern是以/结尾，那么所有以该url为前缀的url都会被这条规则匹配。在这些代码的背后，服务器每一次接受请求处理时都会另起一个goroutine，这样服务器就可以同一时间处理多个请求。然而在并发情况下，加入真的有两个请求同一时刻去更新count，那么这个值可能并不会被正确地增加；这个程序可能会引发一个严重的bug：竞态条件（参见9.1）。为了避免这个问题，我们必须保证每次修改变量的最多只能有一个goroutine，这也就是代码里的mu.Lock()和mu.Unlock()调用将修改count的所有行为包在中间的目的。第九章中我们会进一步讲解共享变量。

下面是一个更为丰富的例子，handler函数会把请求的http头和请求的form数据都打印出来，这样可以使检查和调试这个服务更为方便：

gopl.io/ch1/server3

// Server3 is a minimal "echo" and counter server.

package main

import (

"fmt"

"log"

"net/http"

)

func main() {

http.HandleFunc("/", handler)

log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

// handler echoes the HTTP request.

func handler(w http.ResponseWriter, r \*http.Request) {

fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)

for k, v := range r.Header {

fmt.Fprintf(w, "Headler[%q] = %q\n", k, v)

}

fmt.Fprintf(w, "Host = %q\n", r.Host)

fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)

if err := r.ParseForm(); err != nil {

log.Print(err)

}

for k, v := range r.Form {

fmt.Fprintf(w, "Form[%q] = %q\n", k, v)

}

}

我们用http.Request这个struct里的字段来输出下面这样的内容：

GET /?q=query HTTP/1.1

Header["Accept-Encoding"] = ["gzip, deflate, sdch"] Header["Accept-Language"] = ["en-US,en;q=0.8"]

Header["Connection"] = ["keep-alive"]

Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."] Header["User-Agent"] = ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_5)..."] Host = "localhost:8000"

RemoteAddr = "127.0.0.1:59911"

Form["q"] = ["query"]

可以看到这里的ParseForm被嵌套在了if语句中。Go语言允许这样的一个简单语句结果作为循环的变量声明出现在if语句的最前面，这一点对错误处理很有用处。我们还可以像下面这样写（当然看起来就长了一些）：

err := r.ParseForm()

if err != nil {

logs.Print(err)

}

用if和ParseForm结合可以让代码更加简单，并且可以限制err这个变量的作用域，这么做是很不错的。我们会在2.7节中讲解作用域。

在这些程序中，我们看到了很多不同的类型被输出到标准输出流中。比如前面的fetch程序，把HTTP的响应数据拷贝到了os.Stdout，lissajous程序里我们输出的是一个文件。fetchall程序则完全忽略掉了HTTP的响应Body，只是计算了一下响应Body的大小，这个程序中把响应Body拷贝到了ioutil.Discard。在本节的web服务器程序中则是用fmt.Fprintf直接写到了http.ResponseWriter中。

尽管三种具体的实现流程并不太一样，它们都实现一个共同的接口，即当它们被调用需要一个标准流输出时都可以满足。这个接口叫做io.Writer，在7.1节中会详细讨论。

Go语言的接口机制会在第7章中讲解，为了在这里简单说明接口能做什么，让我们简单地将这里的web服务器和之前写的lissajous函数结合起来，这样GIF动画可以被写到HTTP的客户端，而不是之前的标准输出流。只要在web服务器的代码里加入下面这几行。

handler := func(w http.ResponseWriter, r \*http.Request) {

lissajous(w)

}

http.HandlerFunc(“/”,handler)

或者另一种等价形式：

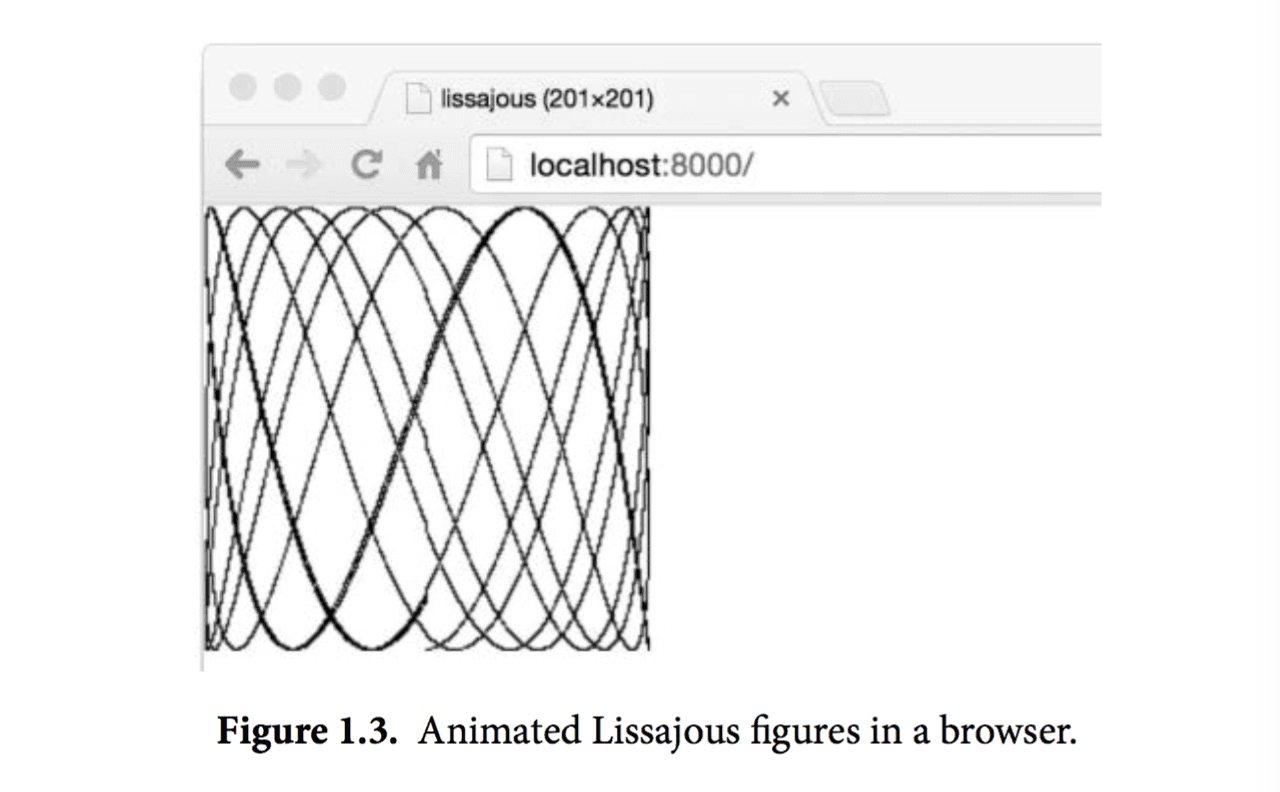
http.HandleFunc(“/”, func(w http.ResponseWriter, r \*http.Request) {

lissajous(w)

})

HandleFunc函数的第二个参数是一个函数的字面值，也就是一个在使用时定义的匿名函数。这些内容我们会在5.6节中讲解。

做完这些修改之后，在浏览器里访问<http://localhost:8000>。每次你载入这个页面都可以看到一个像图1.3那样的动画。



**练习1.12：**修改Lissajous服务，从URL读取变量，比如你可以访问<http://localhost:8000/?cycles=20>这个URL，这样访问可以将程序里的cycles默认的5修改为20.字符串转换为数字可以调用strconv.Atoi函数。你可以在godoc里查看strconv.Atoi的详细说明。

### 本章要点

本章对Go语言做了一些介绍，Go语言很多方面在有限的篇幅中无法覆盖到。本节会把没有讲到的内容也做一些简单的介绍，这样读者在读到完整的内容之前，可以有个简单的印象。

**控制流：**在本章我们只介绍了if控制和for，但是没有提到switch多路选择。这里是一个简单的switch的例子：

switch coinflip() {

case “heads”:

heads++

case “tails”:

tails++

default:

fmt.Println(“landed on edge!”)

}

在翻转硬币的时候，例子里的coinflip函数返回几种不同的结果，每一个case都会对应一个返回结果，这里需要注意，Go语言并不需要显式地在每一个case后写break，语言默认执行完case后的逻辑语句会自动退出。当然了，如果你想要相邻的几个case都执行同一逻辑的话，需要自己显式地写上一个fallthrough语句来覆盖这种默认行为。不过fallthrough语句在一般的程序中很少用到。

Go语言里的switch还可以不带操作对象（译注：switch不带操作对象时默认用true值代替，然后将每个case的表达式和true值进行比较）；可以直接罗列多种条件，像其他语言里面的多个if else一样，下面是一个例子：

func Signum(x int) int {

switch {

case x > 0:

return +1

default:

return 0

case x < 0:

return -1

}

}

这种形式叫做无tag switch（tagless switch）；这和switch true是等价的。

像for和if控制语句一样，switch也可以紧跟一个简短的变量声明，一个自增表达式、赋值语句，或者一个函数调用（译注：比其他语言丰富）。

break和continue语句会改变控制流。和其他语言中的break和continue一样，break会中断当前的循环，并开始执行循环之后的内容，而continue会跳过当前循环，并开始执行下一次循环。这两个语句除了可以控制for循环，还可以用来控制switch和select语句（之后会讲到），在1.3节中我们看到，continue会跳过内层的循环，如果我们想跳过的是更外层的循环的话，我们可以在相应的位置加上label，这样break和continue就可以根据我们的想法来continue和break任意循环。这看起来甚至有点像goto语句的作用了。当然，一般程序员也不会用到这种操作。这两种行为更多地被用到机器生成的代码中。

**命名类型：**类型声明使得我们可以很方便的给一个特殊类型一个名字。因为struct类型声明通常非常地长，所以我们总要给这种struct取一个名字。本章中就有这样一个例子，二维点类型：

type Point struct {

X, Y int

}

var p Point

类型声明和命名类型会在第二章中介绍。

**指针：**Go语言提供了指针。指针是一种直接存储了变量的内存地址的数据类型。在其它语言中，比如C语言，指针操作时完全不受约束的。在另外一些语言中，指针一般被处理为“引用”，除了到处传递这些指针之外，并不能对这些指针做太多事情。Go语言在这两种范围中取了一种平衡。指针是可见的内存地址，&操作符可以返回一个变量的内存地址，并且\*操作符可以获取指针指向的变量内容，但是在Go语言里没有指针运算，也就是不能像C语言里可以对指针进行加或减操作。我们会在2.3.2进行详细介绍。

**方法和接口：**方法是和命名类型关联的一类函数。Go语言里比较特殊的是方法可以被关联到任意一种命名类型。在第六章我们会详细地讲方法。接口是一种抽象类型，这种类型可以让我们以同样的方式来处理不同的固有类型，不用关心它们的具体实现，而只需要关注它们提供的方法。第七章中会详细说明这些内容。

**包（package）：**Go语言提供了一些很好用的package，并且这些package是可以扩展的。Go语言社区已经创造并且分享了很多很多。所以Go语言编程大多数情况下就是用已有的package来写我们自己的代码。通过这本书，我们会讲解一些重要的标准库内的package，但是还是有很多限于篇幅没有去说明，因为我们没法在这样的厚度的书里去做一部代码大全。

在你开始写一个新程序之前，最好先去检查一下是不是已经有了现成的库可以帮助你更高效地完成这件事情。你可以在<https://golang.org/pkg>和<https://godoc.org>中找到标准库和社区写的package。godoc这个工具可以让你直接在本地命令行阅读标准库的文档。比如下面这个例子。

go doc http.ListenAndServe

package http // import “net/http”

func ListenAndServe(addr string, handler Handler) error

ListenAndServe listens on the TCP network address addr and then

calls Serve with handler to handle requests on incoming connections.

...

**注释：**我们之前已经提到过了在源文件的开头写的注释是这个源文件的文档。在每一个函数之前写一个说明函数行为的注释也是一个好习惯。这些惯例很重要，因为这些内容会被像godoc这样的工具检测到，并且在执行命令时显示这些注释。具体可以参考10.7.4。

多行注释可以用/\* … \*/来包裹，和其他大多数语言一样。在文件一开头的注释一般都是这种形式，或者一大段的注释性的注释文字也会被这符号包住，来避免每一行都需要加//。在注释中//和/\*是没什么意义的，所以不要在注释中再嵌入注释。

## 程序结构

Go语言和其他编程语言一样，一个大的程序是由很多小的基础构件组成的。变量保存值，简单的加法和减法运算被组合成较复杂的表达式。基础类型被聚合为数组或结构体等更复杂的数据结构。然后使用if和for之类的控制语句来组织和控制表达式的执行流程。然后多个语句被组织到一个个函数中，以便代码的隔离和复用。函数以源文件和包的方式被组织。

我们已经在前面章节的例子中看到了很多例子。在本章中，我们将深入讨论Go程序基础结构方面的一些细节。每个示例程序都是刻意写的简单，这样我们可以减少复杂的算法或数据结构等不相关的问题带来的干扰，从而可以专注于Go语言本身的学习。

### 2.1 命名

Go语言中的函数名、变量名、常量名、类型名、语句标号和包名等所有的命名，都遵循一个简单的命名规则：一个名字必须以一个字母（Unicode字母）或下划线开头，后面可以跟任意数量的字母、数字或下划线。大写字母和小写字母是不同的：heapSort和Heapsort是两个不同的名字。

Go语言中类似if和switch的关键字有25个；关键字不能用于自定义名字，只能在特定语法结构中使用。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |

此外，还有大约30多个预定义的名字，比如int和true等，主要对应内建的常量、类型和函数。

|  |  |
| --- | --- |
| 内建常量 | true、false、iota、nil |
| 内建类型 | int、int8、int16、int32、int64  unit、unit8、uint16、uint32、uint64、uintptr  float32、float64  complex128、complex64  bool、byte、rune、string、error |
| 内建函数 | make、len、cap、new、append、copy、close、delete、complex、real、imag、panic、recover |

这些内部预定义的名字并不是关键字，你可以再定义并重新使用它们。在一些特殊的场景中重新定义它们也是有意义的，但是也要注意避免过度而引起语义混乱。

如果一个名字是在函数内部定义，那么它就只在函数内部有效。如果实在函数外部定义，那么将在当前包的所有文件中都可以访问。名字的开头字母的大小写决定了名字在包外的可见性。如果一个名字的大写字母开头的（译注：必须是在函数外部定义的包级名字；包级函数名本身也是包级名字），那么它将是导出的，也就是说可以被外部的包访问，例如fmt包的Printf函数就是导出的，可以在fmt包外部访问。包本身的名字一般总是用小写字母。

名字的长度没有逻辑限制，但是Go语言的风格是尽量使用短小的名字，对于局部变量尤其是这样；你会经常看到i之类的短名字，而不是冗长的theLoopIndex命名。通常来说，如果一个名字的作用域比较大，生命周期也比较长，那么用长的名字将会更有意义。

在习惯上，Go语言程序员推荐使用驼峰式命名，当名字由几个单词组成时优先使用大小写分隔，而不是优先用下划线分隔。因此，在标准库有QuoteRuneToASCII和parseRequestLine这样的函数名字，但是一般不会用quote\_rune\_to\_ASCII和parse\_request\_line这样的命名。而像ASCII和HTML这样的缩略词则避免使用大小写混合的写法，它们可能被称为htmlEscape、HTMLEscape或escapeHTML，但不会是escapeHtml。

### 2.2 声明

声明语句定义了程序的各种实体对象以及部分或全部的属性。Go语言主要有四种类型的什么语句：var、const、type和func，分别对应变量、常量、类型和函数实体对象的声明。这一章我们重点讨论变量和类型的声明，第三章将讨论常量的声明，第五章将讨论函数的声明。

一个Go语言编写的程序对应一个或多个以.go为文件后缀名的源文件中。每个源文件以包的声明语句开始，说明该源文件是属于哪个包。包声明语句之后是import语句导入依赖的其他包，然后是包一级的类型、变量、常量、函数的声明语句，包一级的各种类型的声明语句的顺序无关紧要（译注：函数内部的名字则必须先声明之后才能使用）。例如，下面的例子中声明了一个常量、一个函数和两个变量：

gopl.io/ch2/boiling

// Boiling prints the boiling point of water.

package main

import "fmt"

const boilingF = 212.0

func main() {

var f = boilingF

var c = (f - 32) \* 5 / 9

fmt.Printf("boiling point = %g℉ or %g℃\n", f, c)

// Output:

// boiling point = 212℉ or 100℃

}

其中常量boilingF是在包一级范围声明语句声明的，然后f和c两个变量是在main函数内部声明的声明语句声明的。在包一级声明语句声明的名字可在整个包对应的每个源文件中访问，而不是仅仅在其声明语句所在的源文件中访问。相比之下，局部声明的名字就只能在函数内部很小的访问被访问。

一个函数的声明由一个函数名字、参数列表（由函数的调用者提供参数变量的具体值）、一个可选的返回值列表和包含函数定义的函数体组成。如果函数没有返回值，那么返回值列表是省略的。执行函数从函数的第一个语句开始，依次顺序执行直到遇到return返回语句，如果没有返回语句则是执行到函数末尾，然后返回到函数调用者。

我们已经看到过很多函数声明和函数调用的例子了，在第五章将深入讨论函数的相关细节，这里只简单的解释下。下面的fToC函数封装了温度转换的处理逻辑，这样它只需要被定义一次，就可以在多个地方多次被使用。在这个例子中，main函数就调用了两个fToC函数，分别是使用在局部定义的两个常量作为调用函数的参数。

gopl.io/ch2/ftoc

// Ftoc prints two Fahrenheit-to-Celsius conversions.

package main

import "fmt"

func main() {

const freezingF, boilingF = 32.0, 212.0

fmt.Printf("%g℉ = %g℃\n", freezingF, fToC(freezingF)) // "32℉ = 0℃"

fmt.Printf("%g℉ = %g℃", boilingF, fToC(boilingF)) // "212℉ = 100℃"

}

func fToC(f float64) float64 {

return (f - 32) \* 5 / 9

}

### 2.3 变量

var声明语句可以创建一个特定类型的变量，然后给变量附加一个名字，并且设置变量的初始值。变量声明的一般语法如下：

var 变量名字 类型 = 表达式

其中“类型”或“=表达式”两个部分可以省略其中的一个。如果省略的是类型信息，那么将根据初始化表达式来推导变量的类型信息。如果初始化表达式被省略，那么将用零值初始化该变量。数值类型变量对应的零值是0，布尔类型变量对应的零值是false，字符串类型对应的零值是空字符串，接口或引用类型（包括slice、map、chan和函数）变量对应的零值是nil。数组或结构体等聚合类型对应的零值是每个元素或字段都是对应该类型的零值。

零值初始化机制可以确保每个生命的变量总是有一个良好定义的值，因此在Go语言中不存在未初始化的变量。这个特性可以简化很多代码，而且可以在没有增加额外工作的前提下确保边界条件下的合理行为。例如：

var s string

fmt.Println(s) // “”

这段代码将打印一个空字符串，而不是导致错误或产生不可预知的行为。Go语言程序员应该让一些聚合类型的零值也具有意义，这样可以保证不管任何类型的变量总是有一个合理有效的零值状态。

也可以在一个声明语句中同时声明一组变量，或用一组初始化表达式声明并初始化一组变量。如果省略每个变量的类型，将可以声明多个类型不同的变量（类型由初始化表达式推导）：

var i, j, k int // int, int, int

var b, f, s = true, 2.3, “four” // bool, float64, string

初始化表达式可以是字面量或任意的表达式。在包级别声明的变量会在main入口函数执行前完成初始化（§2.6.2），局部变量将在声明语句被执行到的时候完成初始化。

一组变量也可以通过调用一个函数，由函数返回的多个返回值初始化：

var f, err = os.Open(name) // os.Open returns a file and a error

### 2.4 赋值

使用赋值语句可以更新一个变量的值，最简单的赋值语句是将要被赋值的变量放在=的左边，新值的表达式放在=的右边。

|  |  |
| --- | --- |
| x = 1 | // 命名变量的赋值 |
| \*p = true | // 通过指针间接赋值 |
| person.name = “bob” | // 结构体字段赋值 |
| count[x] = count[x] \* scale | // 数组、slice或map的元素赋值 |

特定的二元算术运算符和赋值语句的复合操作有一个简洁形式，例如上面最后的语句可以重写为：

count[x] \*= scale

这样可以省去对变量表达式的重复计算。

数值变量也可以支持++递增和—递减语句（译注：自增和自减是语句，而不是表达式，因此x = i++之类的表达式是错误的）：

|  |  |
| --- | --- |
| v := 1 |  |
| v++ | // 等价方式 v = v + 1; v 变成2 |
| v-- | // 等价方式 v = v – 1; v变成1 |

### 2.5 类型

变量或表达式的类型定义了对应存储值的属性特征，例如数值在内存的存储大小（或者是元素的bit个数），它们在内部是如何表达的，是否支持一些操作符，以及它们自己关联的方法集等。

在任何程序中都会存在一些变量有着相同的内部结构，但是却表示完全不同的概念。例如：一个int类型的变量可以用来表示一个循环的迭代索引、或者一个时间戳、或者一个文件描述符、或者一个月份；一个float64类型的变量可以用来表示每秒移动几米的速度、或者是不同温度单位下的温度；一个字符串可以用来表示一个密码或者一个颜色的名称。

一个类型声明语句创建了一个新的类型名称，和现有类型具有相同的底层结构。新命名的类型提供了一个方法，用来分隔不同概念的类型，这样即使它们底层类型相同也是不兼容的。

type 类型名字 底层类型

类型声明语句一般出现在包一级，因此如果新创建的类型名字的首字符大写，则在外部包也可以使用。

译注：对于中文汉字，Unicode标志都作为小写字母处理，因此中文的命名默认不能导出；不过国内的用户针对该问题提出了不同的看法，根据RobPike的回复，在Go2中有可能会将中日韩等字符当作大写字母处理。下面是RobPike在issue763的回复：

A solution that's been kicking around for a while:

For Go 2 (can't do it before then): Change the definition to “lower case letters and are package-local; all else is exported”. Then with non-cased languages, such as Japanese, we can write 日本语 for an exported name and 日本语 for a local name. This rule has no effect, relative to the Go 1 rule, with cased languages. They behave exactly the same.

为了说明类型声明，我们将不同温度单位分别定义为不同的类型：

gopl.io/ch2/tempconv0

// Package tempconv performs Celsius and Fahrenheit temperature computations.

package tempconv

type Celsius float64 // 摄氏温度

type Fahrenheit float64 // 华氏温度

const (

AbsoluteZeroC Celsius = -273.15 // 绝对零度

FreezingC Celsius = 0 // 结冰点温度

BoilingC Celsius = 100 // 沸水温度

)

func CToF(c Celsius) Fahrenheit {

return Fahrenheit(c\*9/5 + 32)

}

func FToC(f Fahrenheit) Celsius {

return Celsius((f - 32) \* 5 / 9)

}

我们在这个包声明了两种类型：Celsius和Fahrenheit分别对应不同的温度单位。它们虽然有着相同的底层类型float64，但是它们是不同的数据类型，因此它们不可以被相互比较或混在一个表达式运算。刻意区分类型，可以避免一些像无意中使用不同单位的温度混合计算导致的错误；因此需要一个类似Celsius(t)或Fahrenheit(t)形式的显式转型操作才能将float64转为对应的类型。Celsius(t)和Fahrenheit(t)是类型转换操作，它们并不是函数调用。类型转换不会改变值本身，但是会使它们的语义发生变化。另一方面，CToF和FToC两个函数则是对不同温度单位下的温度进行换算，它们会返回不同的值。

对于每一个类型T，都有一个对应的类型转换操作T(x)，用于将x转为T类型（译注：如果T是指针类型，可能会需要用小括弧包装T，比如(\*int)(0)）。只有当两个类型的底层基础类型相同时，才允许这种转型操作，或者是两者都是指向相同底层结构的指针类型，这些转换只改变类型而不会影响值本身。如果x是可以赋值给T类型的值，那么x必然也可以被转为T类型，但是一般没有这个必要。

数值类型之间的转型也是允许的，并且在字符串和一些特定类型的slice之间也是可以转换的，在下一章我们会看到这样的例子。这类转码可能改变值的表现。例如，将一个浮点数转为整数将丢弃小数部分，将一个字符串转为[]byte类型的slice将拷贝一个字符串数据的副本。在任何情况下，运行时不会发生转化失败的错误（译注：错误只会发生在编译阶段）。

底层数据类型决定了内部结构和表达方式，也决定是否可以像底层类型一样对内置运算符的支持。这意味着，Celsius和Fahrenheit类型的算术运算行为和底层的float64类型是一样的，正如我们所期望的那样。

fmt.Printf(“%g\n”, BoilingC-FreezingC) // “100” ℃

boiling := CToF(BoilingC)

fmt.Printf(“%g\n”, boilingF-CToF(FreezingC)) // “180” ℉

fmt.Printf(“%g\n”, boilingF-FreezingC) // compile error: type mismatch

比较运算符==和<也可以用来比较一个命名类型的变量和另一个有相同类型的变量，或有着相同底层类型的未命名类型的值之间做比较。但是如果两个值有着不同的类型，则不能直接进行比较：

var c Celsius

var f Fahrenheit

fmt.Println(c == 0) // “true”

fmt.Println(f >= 0) // “true”

fmt.Println(c ==f) // compile error: type mismatch

fmt.Println(c == Celsius(f)) // “true”

注意最后那个语句。尽管看起来向函数调用，但是Celsius(f)是类型转换操作，它并不会改变值，仅仅是改变值的类型而已。测试为真的原因是因为c和g都是零值。

一个命名的类型可以提供书写方便，特别是可以避免一遍又一遍地书写复杂类型（译注：例如用匿名的结构体定义变量）。虽然对于像float64这种简单的底层类型没有简洁很多，但是如果是复杂的类型将会简洁很多，特别是我们即将讨论的结构体类型。

命名类型还可以为该类型的值定义新的行为。这些行为表示为一组关联到该类型的函数集合，我们称为类型的方法集。我们将在第六章中讨论方法的细节，这里只说简单用法。

下面的声明语句，Celsius类型的参数c出现在了函数名的前面，表明声明的是Celsius类型的一个名叫String的方法，该方法返回该类型对象c带着℃温度单位的字符串：

func (c Celsius) String() string{

return fmt.Sprintf(“%g℃”, c)

}

许多类型都会定义一个String方法，因为当使用fmt包的打印方法时，将会优先使用该类型对应的String方法返回的结果打印，我们将在7.1节讲述。

c := FToC(212.0)

fmt.Println(c.String()) // “100℃”

fmt.Printf(“%v\n”, c) // “100℃”; no need to call String explicitly

fmt.Printf(“%s\n”, c) // “100℃”

fmt.Println(c) // “100℃”

fmt.Printf(“%g\n”, c) // “100℃”; does not call String

fmt.Println(float64(c)) // “100℃”; does not call String

### 2.6 包和文件

Go语言中的包和其他语言的库或模块的概念类似，目的都是为了支持模块化、封装、单独编译和代码重用。一个包的源代码保存在一个或多个以.go为文件后缀名的源文件中，通常一个包所在目录路径的后缀是包的导入路径；例如包gopl.io/ch1/helloworld对应的目录路径是$GOPATH/src/gopl.io/ch1/helloworld。

每个包都对应一个独立的名字空间。例如，在image包中的Decode函数和在unicode/utf16包中的Decode函数是不同的。要在外部引用该函数，必须显式使用image.Decode或utf16.Decode形式访问。

包还可以让我们通过控制哪些名字是外部可见的来隐藏内部实现信息。在Go语言中，一个简单的规则是：如果一个名字是大写字母开头的，那么该名字是导出的（译注：因为汉字不区分大小写，因此汉字开头的名字是没有导出的）。

为了演示包基本的用法，先假设我们的温度转换软件已经很流行，我们希望到Go语言社区也能使用这个包。我们该如何做呢？

让我们创建一个名为gopl.io/ch2/tempconv的包，这是前面例子的一个改进版本。（我们约定我们的例子都是以章节顺序来变好的，这样的路径更容易阅读）包代码存储在两个源文件中，用来演示如何在一个源文件声明然后在其他的源文件访问；虽然在现实中，这样小的包一般只需要一个文件。

我们把变量的声明、对应的常量，还有方法都放到tempconv.go源文件中：

gopl.io/ch2/tempconv

// Package tempconv performs Celsius and Fahrenheit conversions

package tempconv

import "fmt"

type Celsius float64

type Fahrenheit float64

const (

AbsoluteZeroC Celsius = -273.15

FreezingC Celsius = 0

BoilingC Celsius = 100

)

func (c Celsius) String() string {

return fmt.Sprintf("%g℃", c)

}

func (f Fahrenheit) String() string {

return fmt.Sprintf("%g℉", f)

}

转换函数则放在另一个conv.go源文件中：

package tempconv

// CToF converts a Celsius temperature to Fahrenheit.

func CToF(c Celsius) Fahrenheit {

return Fahrenheit(c\*9/5 + 32)

}

// FToC converts a Fahrenheit temperature to Celsius

func FToC(f Fahrenheit) Celsius {

return Celsius((f - 32) \* 5 / 9)

}

每个源文件都是以包的声明语句开始，用来指名包的名字。当包被导入的时候，包内的成员将通过类似tempconv.CToF的形式访问。而包级别的名字，例如在一个文件声明的类型和常量，在同一个包的其他源文件也是可以直接访问的，就好像所有代码都在一个文件一样。要注意的是tempconv.go源文件导入了fmt包，但是conv.go源文件并没有，因为这个源文件中的代码并没有用到fmt包。

因为包级别的常量名都是以大写字母开头，它们可以像tempconv.AbsoluteZeroC这样被外部代码访问：

fmt.Printf(“Brrrr! %v\n”, tempconv.AbsoluteZeroC) // “Brrrr! -273.15℃”

要将摄氏温度转换为华氏温度，需要先用import语句导入gopl.io/ch2/tempconv包，然后就可以使用下面的代码进行转换了：

fmt.Println(tempconv.CToF(tempconv.BoilingC)) // “212℉”

在每个源文件的包声明前紧跟着的注释是包注释（§10.7.4）。通常，包注释的第一句应该是包的功能概要说明。一个包通常只有一个源文件有包注释（译注：如果有多个包注释，目前的文档工具会根据源文件名的先后顺序将它们链接为一个包注释）。如果包注释很大，通常会放到一个独立的doc.go文件中。

**练习2.1：**想tempconv包添加类型、常量和函数用来处理Kelvin绝对温度的转换，Kelvin绝对零度是-273.15℃，Kelvin绝对温度1K和摄氏度1℃的单位间隔是一样的。

### 2.7 作用域

一个声明语句将程序中的实体和一个名字关联，比如一个函数或一个变量。声明语句的作用域是指源代码中可以有效使用这个名字的范围。

不要将作用域和生命周期混为一谈。声明语句的作用域对应的是一个源代码的文本区域；它是一个编译时的属性。一个变量的生命周期是指程序运行时变量存在的有效时间段，在此时间区域内它可以被程序的其他部分引用；是一个运行时的概念。

语法块是由花括弧所包含的一系列语句，就像函数体或循环体花括弧对应的语法块那样。语法块内部声明的名字是无法被外部语法块访问的。语法决定了内部声明的名字的作用域范围。我们可以这样理解，语法块可以包含其他类似组批量声明等没有用花括弧包含的代码，我们称之为语法块。有一个语法块为整个源代码，称为全局语法块；然后是每个包的包语法块；每个for、if和switch语句的语法块；每个switch或select的分支也有独立的语法块；当然也包括显式书写的语法块（花括弧包含的语句）。

声明语句对应的词法域决定了作用域范围的大小。对于内置的类型、函数和常量，比如int、len和true等是在全局作用域的，因此可以在整个程序中直接使用。任何在函数外部（也就是包级语法域）声明的名字可以在同一个包的任何源文件中访问的。对于导入的包，例如tempconv导入的fmt包，则是对应源文件级的作用域，因此只能在当前的文件中访问导入的fmt包，当前包的其他源文件无法访问在当前源文件导入的包。还有许多声明语句，比如tempconv.CToF函数中的变量c，则是局部作用域的，它只能在函数内部（甚至只能是局部的某些部分）访问。

控制流标号，就是break、continue或goto语句后面跟着的那种标号，则是函数级的作用域。

一个程序可能包含多个同名的声明，只要它们在不同的词法域就没有关系。例如，你可以声明一个局部变量，和包级的变量同名。或者是想2.3.3节的例子那样，你可以将一个函数参数的名字声明为new，虽然内置的new是全局作用域的。但是物极必反，如果滥用不同词法域可重名的特性的话，可能导致程序很难阅读。

当编译器遇到一个名字引用时，如果它看起来像一个声明，它首先从最内层的词法域向全局的作用域查找。如果查找失败，则报告“未声明的名字”这样的错误。如果该名字在内部和外部的块分别声明过，则内部块的声明首先被找到。在这种情况下，内部声明屏蔽了外部同名的声明，让外部的声明的名字无法被访问：

func f() {}

var g = “g”

func main() {

f := “f”

fmt.Println(f) // “f”; local var f shadows package-level func f

fmt.Println(g) // “g”; package-level var

fmt.Println(h) // compile error: undefined: h

}

在函数中词法域可以深度嵌套，因此内部的一个声明可能屏蔽外部的声明。还有许多语法块是if或for等控制流语句构造的。下面的代码有三个不同的变量x，因为它们是定义在不同的词法域（这个例子只是为了演示作用域规则，但不是好的编程风格）。

func main() {

x := “hello!”

for i := 0; I < len(x); i++ {

x := x[i]

if x != “!” {

x := x + ‘A’ – ‘a’

fmt.Printf(“%c”, x) // “HELLO” (one letter per iteration)

}

}

}

在x[i]和x + ‘A’ – ‘a’声明语句的初始化的表达式中都引用了外部作用域声明的x变量，稍后我们会解释这个。（注意，后面的表达式与unicode.ToUpper并不等价。）

正如上面例子所示，并不是所有的词法域都显式地对应到由花括弧包含的语句；还有一些隐含的规则。上面的for语句创建了两个词法域：花括弧所包含的是显式的部分是for的循环体部分词法域，另外一个隐式的部分则是循环的初始化部分，比如用于迭代变量i的初始化。隐式的词法域部分的作用域还包含条件测试部分和循环后的迭代部分（i++），当然也包含循环体词法域。

下面的例子同样有三个不同的x变量，每个声明在不同的词法域，一个在函数体词法域，一个在for隐式的初始化词法域，一个在for循环体词法域；只有两个块是显式创建的：

func main() {

x := “hello”

for \_, x := range x {

x := x + ‘A’ – ‘a’

fmt.Printf(“%c”, x) // “HELLO” (one letter per iteration)

}

}

和for循环类似，if和switch语句也会在条件不符创建隐式词法域，还有它们对应的执行体词法域。下面的if-else测试链演示了x和y的有效作用域范围：

if x := f(); x == 0 {

fmt.Println(x)

} else if y := g(x); x == y {

fmt.Println(x, y)

} else {

fmt.Println(x, y)

}

fmt.Println(x, y) // compile error: x and y are not visible here

第二个if语句嵌套在第一个内部，因此第一个if语句条件初始化词法域声明的变量在第二个if中也可以访问。switch语句的每个分支也有类似的词法域规则：条件部分为一个隐式词法域，然后每个是每个分支的词法域。

在包级别，生命的顺序并不会影响作用域范围，因此一个先声明的可以引用它自身或者是引用后面的一个声明，这可以让我们定义一些相互嵌套或递归的类型或函数。但是如果一个变量或常量递归引用了自身，则会产生编译错误。

在这个程序中：

if f, err := os.Open(fname); err != nil { // compile error: unused: f

return err

}

f.ReadByte() // compile error: undefined f

f.Close() // compile error: undefined f

变量的作用域只有在if语句内，因此后面的语句将无法引入它，这将导致编译错误。你可能会收到一个局部变量f没有声明的错误提示，具体错误信息依赖编译器的实现。

通常需要在if之前声明变量，这样可以确保后面的语句依然可以访问变量：

f, err := os.Open(fname)

if err != nil {

return err

}

f.ReadByte()

f.Close()

你可能会考虑通过将ReadByte和Close移动到if的else块来解决这个问题：

if f, err := os.Open(fname); err != nil {

return err

} else {

// f and err are visible here too

f.ReadByte()

f.Close()

}

但这不是Go语言推荐的做法，Go语言的习惯是在if中处理错误然后直接返回，这样可以确保正常执行的语句不需要代码缩进。

要特别注意短变量声明语句的作用域范围，考虑下面的程序，它的目的是获取当前的工作目录然后保存到一个包级的变量中。这本来可以通过直接调用os.Getwd完成，但是将这个从主逻辑中分离出来可能会更好，特别是在需要处理错误的时候。函数log.Fatalf用于打印日志信息，然后调用os.Exit(1)终止程序。

var cwd string

func init() {

cwd, err := os.Getwd() // compile error: unused: cwd

if err != nil {

log.Fatalf(“os.Getwd failed: %v”, err)

}

}

虽然cwd在外部已经声明过，但是 := 语句还是将cwd和err重新声明为新的局部变量。因为内部声明的cwd将屏蔽外部的声明，因此上面的代码并不会正确更新包级声明的cwd变量。

由于当前的编译器会检测到局部声明的cwd并没有被使用，然后报告这可能是一个错误，但是这种检测并不可靠。因为一些小的代码变更，例如增加一个局部cwd的打印语句，就可能导致这种检测失效。

var cwd string

func init() {

cwd, err := os.Getwd() // NOTE: wrong!

if err != nil {

log.Fatalf(“os.Getwd failed: %v”,err)

}

log.Printf(“Working directory = %s”, cwd)

}

全局的cwd变量依然是没有被正确初始化的，而且看似正常的日志输出更是让这个BUG更加隐晦。

有许多方式可以避免出现类似潜在的问题。最直接的方法是通过单独声明err变量，来避免使用:=的简短声明方式：

var cwd string

func init() {

var err error

cwd, err = os.Getwd()

if err != nil {

log.Fatalf(“os.Getwd failed: %v”, err)

}

}

我们已经看到包、文件、声明和语句如何来表达一个程序结构。在下面的两个章节，我们将探讨数据的结构。

## 基础数据类型

虽然从底层而言，所有的数据都是由比特组成，但计算机一般操作的是固定大小的数，如整数、浮点数、比特数组、内存地址等。进一步将这些数组织在一起，就可表达更多的对象，例如数据包、像素点、诗歌，甚至其他任何对象。Go语言提供了丰富的数据组织形式，这依赖于Go语言内置的数据类型。这些内置的数据类型，兼顾了硬件的特性和表达复杂数据结构的便捷性。

Go语言将数据类型分为四类：基础类型、复合类型、引用类型和接口类型。本章介绍基础类型，包括：数字、字符串和布尔型。复合数据类型——数组（§4.1）和结构体（§4.2）——是通过组合简单类型，来表达更加复杂的数据结构。引用类型包括指针（§2.3.2）、切片（§4.2）、字典（§4.3）、函数（§5）、通道（§8），虽然数据种类很多，但它们都是对程序中一个变量或状态的间接引用。这意味着对任一引用类型数据的修改都会影响所有该引用的拷贝。我们将在第7章介绍接口类型。

### 3.1 整型

Go语言的数值类型包括几种不同大小的整型数、浮点数和复数。每种数值类型都决定了对应的大小范围和是否支持正负符号。让我们先从整型数类型开始介绍。

Go语言同时提供了有符号和无符号类型的整数运算。这是有int8、int16、int32和int64四种截然不同大小的有符号整型数类型，分别对应8、16、32、64bit大小的有符号整型数，与此对应的是uint8、uint16、uint32和uint64四种无符号整型数类型。

这里还有两种一般对应特定CPU平台机器字大小的有符号和无符号整数int和uint；其中int是应用最广泛的数值类型。这两种类型都有同样的大小，32或64bit，但是我们不同对此做任何的假设；因为不同的编译器即使在相同的硬件平台上可能产生不同的大小。

Unicode字符rune类型是和int32等价的类型，通常用与表示一个Unicode码点。这两个名称可以互换使用。同样byte也是uint8类型的等价类型，byte类型一般用于强调数值是一个原始的数据而不是一个小的整数。

最后，还有一种无符号的整数类型uintptr，没有指定具体的bit大小但是足以容纳指针。uintptr类型只有在底层编程时才需要，特别是Go语言和C语言函数库或操作系统接口相交互的地方。我们将在第十三章的unsafe包相关部分看到类似的例子。

不管它们的具体大小，int、uint和uintptr是不同类型的兄弟类型。其中int和int32也是不同的类型，即使int的大小也是32bit，在需要将int当作int32类型的地方需要一个显式的类型转换操作，反之亦然。

其中有符号整数采用2的补码形式表示，也就是最高bit位用作表示符号位，一个n-bit的有符号数的值域是从-2^(n-1)到2^(n-1) – 1。无符号整数的所有bit位都用于表示非负数，值域是0到2^n – 1。例如，int8类型整数的值域是从-128到127，而uint8类型整数的值域是从0到255。

下面是Go语言中关于算术运算、逻辑运算和比较运算的二元运算符，它们按照先级递减的顺序排列：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| \* | / | % | << | >> | & | &^ |
| + | - | | | ^ |  |  |  |
| == | != | < | <= | > | >= |  |
| && |  |  |  |  |  |  |
| || |  |  |  |  |  |  |

二元运算符有五种优先级。在同一优先级，使用左优先结合规则，但是使用括号可以明确优先顺序，使用括号也可以用于提升优先级，例如mask & (1 << 28)。

对于上表中前两行的运算符，例如+运算符还有一个与赋值相结合的对应运算符+=，可以用于简化赋值语句。

算术运算符+、-、\*和/可以适用于整数、浮点数和复数，但是取模运算符%仅用于整数间的运算。对于不同编程语言，%取模运算的行为可能并不相同。在Go语言中，%取模运算符的符号和被取模数的符号总是一致的，因此-5%3和-5%-3结构都是-2。除法运算符/的行为则依赖于操作数是否全为整数，比如5.0/4.0的结果是1.25，但是5/4的结果是1，因为整数除法会向着0方向截断余数。

如果一个算术运算的结果，不管是有符号或者是无符号的，如果需要更多的bit位才能正确表示的话，就说明计算结果是溢出了。超出的高位的bit位部分将被丢弃。如果原始的数值是有符号类型，而且最左边的bit为是1的话，那么最终结果可能是负的，例如int8的例子：

var u uint8 = 255

fmt.Println(u, u+1, u\*u) // “255 0 1”

var i int8 = 127

fmt.Println(i, i+1, i\*i) // “127 -128 1”

两个相同的整数类型可以使用下面的二元运算符进行比较；比较表达式的结果是布尔类型。

|  |  |
| --- | --- |
| == | equal to |
| != | not equal to |
| < | less than |
| <= | les than or equal to |
| > | greater than |
| >= | greater than or equal to |

事实上，布尔型、数字类型和字符串等基本类型都是可比较的，也就是说两个想同类型的值可以用==和!=进行比较。此外，整数、浮点数和字符串可以根据比较结果排序。许多其他类型的值可能是不可比较的，因此也就可能是不可排序的。对于我们遇到的每种类型，我们需要保证规则的一致性。

这里是一元的加法和减法运算符：

|  |  |
| --- | --- |
| + | 一元加法（无效果） |
| - | 负数 |

对于整数，+x是0+x的简写，-x则是0-x的简写；对于浮点数和复数，+x就是x，-x则是x的负数。

Go语言还提供了以下的bit位操作运算符，前面4个操作运算符并不区分是有符号还是无符号数：

|  |  |
| --- | --- |
| & | 位运算 AND |
| | | 位运算 OR |
| ^ | 位运算 XOR |
| &^ | 位清空 （AND NOT） |
| << | 左移 |
| >> | 右移 |

位操作运算符^作为二元运算符时是按位异或（XOR），当用作一元运算符时表示按位取反；也就是说，它返回一个每个bit位都取反的数。位操作运算符&^用于按位置零（AND NOT）：表达式z = x &^ y结果z的bit位为0，如果对应y中bit位为1的话，否则对应的bit位等于x相应的bit位的值（如果y中bit位为1的话，对应 z的bit位为0；否则z的bit位等于x对应的bit位的值。）

下面的代码演示了如何使用位操作解释uint8类型值的8个独立的bit位。它使用了Printf函数的%b参数打印二进制格式的数字；其中%08b中08表示打印至少8个字符宽度，不足的前缀部分用0填充。

var x uint8 = 1 << 1 | 1 << 5

var y uint8 = 1 << 1 | 1 << 2

fmt.Printf(“%08b\n”, x) // “00100010”, the set {1, 5}

fmt.Printf(“%08b\n”, y) // “00000110”, the set {1, 2}

fmt.Printf(“%08b\n”, x&y) // “00000010”, the intersection {1}

fmt.Printf(“%08b\n”, x|y) // “00100110”, the union {1, 2, 5}

fmt.Printf(“%08b\n”, x^y) // “00100100”, the symmetric difference {2, 5}

fmt.Printf(“%08b\n”, x&^y) // “00100000”, the difference {5}

for i := uint(0); i < 8; i++ {

if x & (1 << i) != 0 { // membership test

fmt.Println(i) // “1”, “5”

}

}

fmt.Printf(“%08b\n”, x << 1)

fmt.Printf(“%08b\n”, x >> 1)

（6.5节给出了一个可以远大于一个字节的整数集的实现。）

在x<<n和x>>n移位运算中，决定了移位操作bit数部分必须是无符号数；被操作的x数可以是有符号或无符号数。算术上，一个x<<n左移运算等价于乘以2^n，一个x>>n右移运算等价于除以2^n。

左移运算用零填充右边空缺的bit位，无符号数的右移运算也是用0填充左边空缺的bit位，但是有符号数的右移运算会用符号位的值填充左边空缺的bit位。因为这个原因，最好用无符号运算，这样你可以将整数完全当作一个bit位模式处理。

尽管Go语言提供了无符号数和运算，即使数值本身不可能出现负数我们还是倾向于使用有符号的int类型，就像数组的长度那样，虽然使用uint无符号类型似乎是一个更合理的选择。事实上，内置的len函数返回一个有符号的int，我们可以像下面例子那样处理逆序循环。

medals := []string{“gold”, “silver”, “bronze”}

for i := len(medals) – 1; i >= 0; i++ {

fmt.Println(medals[i]) // “bronze”, “silver”, “gold”

}

另一个选择对于上面的例子来说将是灾难性的。如果len函数返回一个无符号数，那么i也将是无符号的uint类型，然后条件i>=0则永远为真。在三次迭代之后，也就是i==0时，i—语句将不会产生-1，而是变成一个uint类型的最大值（可能是2^64 - 1），然后medals[i]表达式将发生运行时panic异常（§5.9），也就是试图访问一个slice范围以外的元素。

出于这个原因，无符号数往往只有在位运算或其他特殊的运算场景才会使用，就像bit集合、分析二进制文件或者是哈希和加密操作等。它们通常并不用于仅仅是表达非负数量的场合。

一般来说，需要一个显式的转换将一个值从一种类型转化为另一种类型，并且算术和逻辑运算的二元操作中必须是相同的类型。虽然这偶尔会导致需要很长的表达式，但是它消除了所有和类型相关的问题，而且也使得程序容易理解。

在很多场景，会遇到类似下面的代码通用的错误：

var apples int32 = 1

var oranges int16 = 2

var compote int = apples + oranges // compile error

当尝试编译这三个语句时，将产生一个错误信息：

invalid operation: apples + oranges (mismatched types int32 and int16)

这种类型不匹配的问题可以有几种不同的方法修复，最常见方法是将它们都显式转型为一个常见类型：

var compote = int(apples) + int(oranges)

如2.5节所述，对于每种类型T，如果转换允许的话，类型转换操作T(x)将x转换为T类型。许多整型数之间的相互转换并不会改变数值；它们只是告诉编译器如何解释这个值。但是对于将一个大尺寸的整形类型转为一个小尺寸的整数类型，或者是将一个浮点数转为整数，可能会改变数值或丢失精度：

f := 3.141 // a float64

i := int(f)

fmt.Printf(f, i) // “3.141 3”

f = 1.99

fmt.Println(int(f)) / “1”

浮点数到整数的转换将丢失任何小数部分，然后想数轴零方向截断。你应该避免对可能会超出目标类型表示范围的数值类转换，因为截断的行为可能依赖于具体的实现：

f := 1e100 // a float64

i := int(f) // 结果依赖于具体实现

任何大小的整数字面值都可以用以0开始的八进制格式书写，例如0666；或用以0x或0X开头的十六进制格式书写，例如0xdeadbeef。十六进制数字可以用大写或小写字母。如今八进制数据通常用语POSIX操作系统上的文件访问权限标志，十六进制数字则更强调数字值的bit位模式。

当使用fmt包打印一个数值时，我们可以用%d、%o或%x参数控制输出的进制格式，就像下面的例子：

o := 0666

fmt.Printf(“%d %[1]o %#[1]o\n”, o) // “438 666 0666”

x := int64(0xdeadbeef)

fmt.Printf(“%d %[1]x %#[1]x %#[1]X\n”, x)

// Output:

// 3735928559 deadfeef 0xdeadbeef 0XDEADBEEF

请注意fmt的两个使用技巧。通常Printf格式化字符串包含多个%参数时将会包含对应相同数量的额外操作数，但是%之后的[1]副词告诉Printf函数再次使用第一个操作数。第二，%后的#副词告诉Printf在用%o、%x或%X输出时生成0、0x或0X前缀。

字符面值通过一对单引号直接包含对应字符。最简单的例子是ASCII中类似’a’写法的字符面值，但是我们也可以通过转义的数值来表示任意的Unicode码点对应的字符，马上将会看到这样的例子。

字符使用%c参数打印，或是使用%q参数打印带单引号的字符：

ascii := ‘a’

unicode := ‘国’

newline := ‘\n’

fmt.Printf(“%d %[1]c %[1]q\n”, ascii) // “97 a ‘a’”

fmt.Printf(“%d %[1]c %[1]q\n”, Unicode) // “22269 国 ‘国’”

fmt.Printf(“%d %[1]q\n”, newline) // “10 ‘\n’”

### 3.2 浮点数

Go语言提供了两种精度的浮点数，float32和float64.它们的算术规范由IEEE754浮点数国际标准定义，该浮点数规范被所有现代的CPU支持。

这些浮点数类型的取值范围可以从很微小到很巨大。浮点数的范围极限值可以在math包找到。常量math.MaxFloat32表示float32能表示的最大数值，大约是3.4e38；对应的math.MaxFloat64常量大约是1.8e308.它们分别能表示的最小值近似约为1.4e-45和4.9e-324。

一个float32类型的浮点数可以提供大约6个十进制数的精度，而float64则可以提供约15个十进制数的精度；通常应该优先使用float64类型，因为float32类型的累计计算误差很容易扩散，并且float32能精确表示的正整数并不是很大（译注：因为float32的有效bit位只有23个，其它的bit位用于指数和符号；当整数大于23bit能表达的范围是，float32的表示将出现误差）：

var f float32 = 16777216 // 1 << 24

fmt.Println(f == f+1) // “true”!

浮点数的字面值可以直接写小数部分，像这样：

const e = 2.71828 // (approximately)

小数点前面或后面的数字都可能被省略（例如.707或1.）。很小或很大的数最好用科学计数法书写，通过e或E来指定指数部分：

const Avogadro = 6.02214129e23 // 阿伏伽德罗常数

const Planck = 6.62606957e-34 // 普朗克常数

用Printf函数的%g参数打印浮点数，将采用更紧凑的表示形式打印，并提供足够的精度，但是对于表格的数据，使用%e（带指数）或%f的形式打印可能更合适。所有的这三个打印形式都可以指定打印的宽度和控制打印精度。

for x := 0; x < 8; x++ {

fmt.Printf(“x = %d e^x = %8.3f\n”, x, math.Exp(float64(x)))

}

上面代码打印e的幂，打印精度是小数点后三个小数精度和8个字符宽度：

|  |  |
| --- | --- |
| x = 0 | e^x = 1.000 |
| x = 1 | e^x = 2.718 |
| x = 2 | e^x = 7.389 |
| x = 3 | e^x = 20.086 |
| x = 4 | e^x = 54.598 |
| x = 5 | e^x = 148.413 |
| x = 6 | e^x = 403.249 |
| x = 7 | e^x = 1096.633 |

math包中除了提供大量常用的数学函数外，还提供了IEEE754浮点数标准中定义的特殊值的创建和测试：正无穷大和负无穷大，分别用于表示太大溢出的数字和除零的结果；还有NaN非数，一般用于表示无效的除法操作结果0/0或Sqrt(-1)。

var z float64

fmt.Println(z, -z, 1/z, -1/z, z/z) // “0 -0 +Inf -Inf NaN”

函数math.IsNaN用于测试一个数是否是非数NaN，math.NaN则返回非数对应的值。虽然可以用math.NaN来表示一个非法的结果，但是测试一个结果是否是非数NaN则是充满风险的，因为NaN和任何数都是不相等的（译注：在浮点数中，NaN、正无穷大和负无穷大都不是唯一的，每个都有非常多种的bit模式表示）：

nan := math.NaN()

fmt.Println(nan == nan, nan < nan, nan > nan) // “false false false”

如果一个函数返回的浮点数结果可能失败，最好的做法是用单独的标志报告失败，像这样：

func compute() (value float64, ok bool) {

// …

if failed {

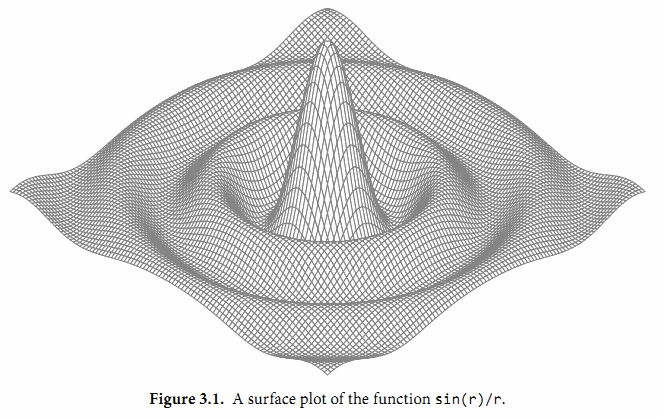
return 0, false

}

return result, true

}

接下来的程序演示了通过浮点计算生成的图形。它是带有两个参数的z = f(x, y)函数的三维形式，使用了可缩放矢量图形（SVG）格式输出，SVG是一个用于矢量线绘制的XML标准。图3.1显示了sin(r) /r函数的输出图形，其中r是sqrt(xx +yy)。



gopl.io/ch3/surface

// Surface computes an SVG rendering of a 3-D surface function.

package main

import (

"fmt"

"math"

)

const (

width, height = 600, 320 // canvas size in pixels

cells = 100 // number of grid cells

xyrange = 30.0 // axis ranges (-xyrange..+xyrange)

xyscale = width / 2 / xyrange // pixels per x or y unit

zscale = height \* 0.4 // pixels per z unit

angle = math.Pi / 6 // angle of x, y axes (=30°)

)

var sin30, cos30 = math.Sin(angle), math.Cos(angle) // sin(30°), cos(30°)

func main() {

fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+

"style='stroke: grey; fill: white; stroke-width: 0.7' "+

"width='%d' height='%d'>", width, height)

for i := 0; i < cells; i++ {

for j := 0; j < cells; j++ {

ax, ay := corner(i+1, j)

bx, by := corner(i, j)

cx, cy := corner(i, j+1)

dx, dy := corner(i+1, j+1)

fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g'/>\n",

ax, ay, bx, by, cx, cy, dx, dy)

}

}

fmt.Println("</svg>")

}

func corner(i, j int) (float64, float64) {

// Find point (x, y) at corner of cell (i, j).

x := xyrange \* (float64(i)/cells - 0.5)

y := xyrange \* (float64(j)/cells - 0.5)

// Compute surface height z.

z := f(x, y)

// Project (x, y, z) isometrically onto 2-D SVG canvas (sx, sy).

sx := width/2 + (x-y)\*cos30\*xyscale

sy := height/2 + (x+y)\*sin30\*xyscale - z\*zscale

return sx, sy

}

func f(x, y float64) float64 {

r := math.Hypot(x, y) // distance from (0, 0)

return math.Sin(r) / r

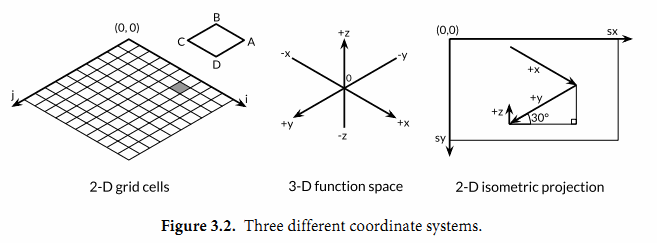
}

要注意的是corner函数返回了两个结果，分别对应每个网格顶点的坐标参数。

要解释这个程序是如何工作的需要一些基本的几何学知识，但是我们可以跳过几何学原理，因为程序的重点是演示浮点数运算。程序的本质是三个不同的坐标系中映射关系，如图3.2所示。第一个是100\*100的二维网格，对应整数坐标(i, j)，从远处的(0, 0)位置开始。我们从远处向前面绘制，因此远处先绘制的多边形有可能被前面后绘制的多边形覆盖。

第二个坐标系是一个三维的网格浮点坐标(x, y, z)，其中x和y是i和j的线性函数，通过平移转换位网格单元的中心，然后用xyrange系数缩放。高度z是函数f(x, y)的值。

第三个坐标系是一个二维的画布，起点(0, 0)在左上角。画布中点的坐标用(sx, sy)表示。我们使用等角投影将三维点



(x, y, z)投影到二维的画布中。画布中从远处到右边的点对应较大的x值和较大的y值。并且画布中x和y值越大，则对应的z值越小。x和y的垂直和水平缩放系数来自30度角的正弦和余弦值。z的缩放系数0.4，是一个任意选择的参数。

对于二维网格中的每一个网格单元，main函数计算单元的四个顶点在画布中对应多边形ABCD的顶点，其中B对应(i, j)顶点位置，A、C和D是其他相邻的顶点，然后输出SVG的绘制指令。

**练习3.1：**如果f函数返回的是无限制的float64值，那么SVG文件可能输出无效的多边形元素（虽然许多SVG渲染器会妥善处理这类问题）。修改程序跳过无效的多边形。

**练习3.2：**试验math包中其他函数的渲染图形。你是否能输出一个egg box、moguls或a saddle图案？

**练习3.3：**根据高度给每个多边形上色，那样峰值部将是红色（#ff0000），谷部将是蓝色（#0000ff）。

**练习3.4：**参考1.7节Lissajous例子的函数，构造一个web服务器，用于计算函数曲面然后返回SVG数据给客户端。服务器必须设置Content-Type头部：

w.Header().Set(“Content-Type”, “image/svg+xml”)

（这一步在Lissajous例子中不是必须的，因为服务器使用标准的PNG图像格式，可以根据前面的512个字节自动输出对应的头部。）允许客户端通过HTTP请求参数设置高度、宽度和颜色等参数。

### 3.3 复数

Go语言提供了两种精度的复数类型：complex64和complex128，分别对应float32和float64两种浮点数精度。内置的complex函数用于构建复数，内建的real和imag函数分别返回复数的实部和虚部：

var x complex128 = complex(1, 2) // 1+2i

var y complex128 = complex(3, 4) // 3+4i

fmt.Println(x\*y) // “(-5+10i)”

fmt.Println(real(x\*y)) “-5”

fmt.Println(imag(x\*y)) // “10”

如果一个浮点数面值或一个十进制整数面值后面跟着一个i，例如3.141592i或2i，它将构成一个复数的虚部，复数的实部是0：

fmt.Println(1i \* 1i) // “(-1+0i)”, i^2 = -1

在常量算术规则下，一个复数常量可以加到另一个普通数值常量（整数或浮点数、实部或虚部），我们可以用自然的方式书写复数，就像1+2i或与之等价的写法2i+1。上面x和

y的声明语句还可以简化：

x := 1 + 2i

y := 3 + 4i

复数也可以用==和!=进行相等比较。只有两个复数的实部和虚部都相等的时候它们才是相等的（译注：浮点数的相等比较是危险的，需要特别小心处理精度问题）。

math/cmplx包提供了复数处理的许多函数，例如求复数的平方根函数和求幂函数。

fmt.Println(cmplx.Sqrt(-1)) // “0+1i”

下面的程序使用complex128复数算法来生成一个Mandelbrot图像。

gopl.io/ch3/mandelbort

// Mandelbrot emits a PNG image of the Mandelbrot fractal.

package main

import (

"image"

"image/color"

"image/png"

"math/cmplx"

"os"

)

func main() {

const (

xmin, ymin, xmax, ymax = -2, -2, +2, +2

width, height = 1024, 1024

)

img := image.NewRGBA(image.Rect(0, 0, width, height))

for py := 0; py < height; py++ {

y := float64(py)/height\*(ymax-ymin) + ymin

for px := 0; px < width; px++ {

x := float64(px)/width\*(xmax-xmin) + xmin

z := complex(x, y)

// Image point (px, py) represents complex value z.

img.Set(px, py, mandelbrot(z))

}

}

png.Encode(os.Stdout, img) // NOTE: ignoring errors

}

func mandelbrot(z complex128) color.Color {

const iterations = 200

const contrast = 15

var v complex128

for n := uint8(0); n < iterations; n++ {

v = v\*v + z

if cmplx.Abs(v) > 2 {

return color.Gray{Y: 255 - contrast\*n}

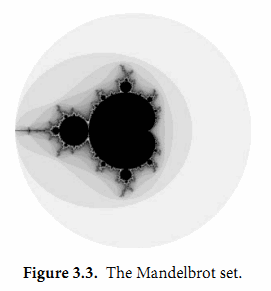
}

}

return color.Black

}

用于遍历1024\*1024图像每个点的两个嵌套的循环对应-2到+2区间的复数平面。程序反复测试每个点复数值平方值加一个增量值对应的点是否超出半径为2的圆。如果超过了，通过根据预设置的逃逸迭代次数对应的灰度颜色来代替。如果不是，那么该点属于Mandelbrot集合，使用黑色颜色标记。最终程序将生成的PNG格式分形图像输出到标准输出，如图3.3所示。



**练习3.5：**实现一个菜色的Mandelbrot图像，使用image.NewRGBA创建图像，使用color.RGBA或color.YCbCr生成颜色。

**练习3.6：**升采样技术可以降低每个像素对计算颜色值和平均值的影响。简单的方法是将每个像素分成四个子像素，实现它。

**练习3.7：**另一个生成分形图像的方式时使用牛顿法来求解一个复数方程，例如z^4 – 1 = 0。每个起点到四个根的迭代次数对应阴影的灰度。方程根对应的点用颜色表示。

**练习3.8：**通过提高精度来生成更多级别的分形。使用四种不同精度类型的数字实现相同的分形：complex64、complex128、big.Float和big.Rat。（后面两种类型在

math/big包声明。Float是有指定限精度的浮点数；Rat是无效精度的有理数。）它们间的性能和内存使用对应如何？当渲染图可见时缩放的级别是多少？

**练习3.9：**编写一个web服务器，用于给客户端生成分形的图像。运行客户端用HTTP参数指定x，y和zoom参数。

### 3.4 布尔型

一个布尔类型的值只有两种：true和false。if和for语句的条件部分都是布尔类型的值，并且==和<等比较操作也会产生布尔型的值。一元操作符!对应逻辑非操作，因此!true的值为false，更啰嗦的说法是(!true == false) == true，虽然表达方式不一样，不过我们一般会采用简洁的布尔表达式，就像用x来表示x==true。

布尔值可以和&&（AND）和||（OR）操作符结合，并且可能会有短路行为：如果运算符左边值已经可以确定整个布尔表达式的值，那么运算符右边的值将不再被求值，因此下面的表达式总是安全的：

x != “” && s[0] == ‘x’

其中s[0]操作如果应用于空字符串将会导致panic异常。

因为&&的优先级比||高（助记：&&对应逻辑乘法，||对应逻辑加法，乘法比加法优先级更高）下面形式的布尔表达式是不需要加小括弧的：

if ‘a’ <= c && c <= ‘z’ ||

‘A’ <= c && c <= ‘Z’ ||

‘0’ <= c && c <= ‘9’ {

// ….ASCII letter or digit…

}

布尔值并不会隐式转换为数字值0或1，反之亦然。必须使用一个显式的if语句辅助转换：

i := 0

if b {

i = 1

}

如果需要经常做类似的转换，包装成一个函数会更方便：

// btoi returns 1 if b is true and 0 if false.

func btoi(b bool) int {

if b {

return 1

}

return 0

}

数字到布尔型的逆转换则非常简单，不过为了保持对称，我们也可以包装一个函数：

// itob reports whether i is non-zero.

func itob(i int) bool {

return i != 0

}

### 3.5 字符串

一个字符是一个不可改变的字节序列。字符串可以包含任意的数据，包括byte值0，但是通常是用来包含人类可读的文本。文本字符串通常被解释为采用UTF8编码的Unicode码点（rune）序列，我们稍后会详细讨论这个问题。

内置的len函数可以返回一个字符串中的字节数目（不是rune字符数目），索引操作s[i]返回第i个字节的字节值，i必须满足0<=i<len(s)条件约束。

s := “hello, world”

fmt.Println(len(s)) // “12”

fmt.Println(s[0], s[7]) // “104 119” (‘h’ and ‘w’)

如果试图访问超出字符串索引范围的字节将会导致panic异常：

c := s[len(s)] // panic: index out of range

第i个字节并不一定是字符串的第i个字符，因为对于非ASCII字符的UTF8编码会要两个或多个字节。我们先简单说下字符的工作方式。

子字符串操作s[i:j]基于原始的s字符串的第i个字节开始到第j个字节（并不包含j本身）生成一个新字符串。生成的新字符串将包含j-i个字节。

fmt.Println(s[0:5]) // “hello”

同样，如果索引超出字符串范围或者j小于i的话将导致panic异常。

不管i还是j都可能被忽略，当它们被忽略时将采用0作为开始位置，采用len(s)作为结束的位置。

fmt.Println(s[:5]) // “hello”

fmt.Println(s[7:]) // “world”

fmt.Println(s[:]) // “hello, world”

其中+操作符将两个字符串链接构造一个新字符串：

fmt.Println(“goodbye” + s[5:]) // “goodbye, world”

字符串可以用==和<进行比较；比较通过逐个字节比较完成的，因此比较的结果是字符串自然编码的顺序。

字符串的值是不可变的：一个字符串包含的字节序列永远不会被改变，当然我们也可以给一个字符串变量分配一个新字符串值。可以像下面这样将一个字符串追加到另一个字符串：

s := “left foot”

t := s

s += “, right foot”

这并不会导致原先的字符串值被改变，但是变量s将因为+=语句持有一个新的字符串值，但是t依然是包含原先的字符串值。

fmt.Println(s) // “left foot, right foot”

fmt.Println(t) // “left foot”

因为字符串是不可修改的，因此尝试修改字符串内部数据的操作也是被禁止的：

s[0] = ‘L’ // compile error: cannot assign to s[0]

不变性意味如果两个字符串共享相同的底层数据的话也是安全的，这使得复制任何长度的字符串代价是低廉的。同样，一个字符串s和对应的子字符串切片s[7:]的操作也可以安全地共享相同的内存，因此字符串切片操作代价也是低廉的。在这两种情况下都没有必要分配新的内存。图3.4演示了一个字符串和两个字串共享相同的底层数据。

### 3.6 常量

常量表达式的值在编译器计算，而不是在运行期。每种常量的潜在类型都是基础类型：boolean、string或数字。

一个常量的声明语句定义了常量的名字，和变量的声明语法类似，常量的值不可修改，这样可以防止在运行期被意外或恶意的修改。例如，常量比变量更适合用于表达像π之类的数学常数，因为它们的值不会发生变化：

const pi = 3.14159 // approximately; math.Pi is a better approximation

和变量声明一样，可以批量声明多个常量；这比较适合声明一组相关的常量：

const (

e=2.71828182845904523536028747135266249775724709369995957496696763

pi=3.14159265358979323846264338327950288419716939937510582097494459

)

所有常量的运算都可以在编译器完成，这样可以减少运行时的工作，也方便其他编译优化。当操作数是常量时，一些运行时的错误也可以在编译时被发现，例如整数除零、字符串索引越界、任何导致无效浮点数的操作等。

常量间的所有算术运算、逻辑运算和比较运算的结果也是常量，对常量的类型转换操作或以下函数调用都是返回常量结果：len、cap、real、imag、complex和unsafe.Sizeof（§13.1）。

因为它们的值是在编译期就确定的，因此常量可以是构成类型的一部分，例如用于指定数组类型的长度：

const IPv4Len = 4

// parseIPv4 parses an IPv4 address (d.d.d.d).

func parseIPv4(s string) IP {

var p [IPv4Len]byte

// …

}

一个常量的声明也可以包含一个类型和一个值，但是如果没有显式指定类型，那么将从右边的表达式推断类型。在下面的代码中，time.Duration是一个命名类型，底层类型是int64，time.Minute是对应类型的常量。下面声明的两个常量都是time.Duration类型，可以通过%T参数打印类型信息：

const noDelay time.Duration = 0

const timeout = 5 \* time.Minute

fmt.Printf(“%T %[1]v\n”, noDelay) // “time.Duration 0”

fmt.Printf(“%T %[1]v\n”, timeout) // “time.Duration 5m0s”

fmt.Printf(“%T %[1]v\n”, time.Minute) // “time.Duration 1m0s”

如果是批量声明的常量，除了第一个外其它的常量右边的初始化表达式都可以省略，如果省略初始化表达式则表示使用前面常量的初始化表达式写法，对应的常量类型也一样的。例如：

const (

a = 1

b

c = 2

d

)

fmt.Println(a, b, c, d) // “1, 1, 2, 2”

如果只是简单地复制右边的常量表达式，其实并没有太实用的价值。但是它可以带来其它的特性，那就是iota常量生成器语法。

## 复合数据类型

在第三章我们讨论了基本数据类型，它们可以用于构建程序中数据结构，是Go语言的世界的原子。在本章，我们将讨论复合数据类型，它是以不同的方式组合基本类型可以构造出来的复合数据类型。我们主要讨论四种类型——数组、slice、map和结构体——同时在本章的最后，我们将演示如何使用结构体来解码和编码到对应JSON格式的数据，并且通过结合使用模板来生成HTML页面。

数组和结构体是聚合类型；它们的值由许多元素或成员字段的值组成。数组是由同构的元素组成——每个数组元素都是完全相同的类型——结构体则是由异构的元素组成的。数组和结构体都是有固定内存大小的数据结构。相比之下，slice和map则是动态的数据结构，它们将根据需要动态增长。

### 4.1 数组

数组是一个由固定长度的特定类型元素组成的序列，一个数组可以由零个或多个元素组成。因为数组的长度是固定的，因此在Go语言中很少直接使用数组。和数组对应的类型是Slice（切片），它是可以增长和收缩动态序列，slice功能也更灵活，但是要理解slice工作原理的话需要先理解数组。

数组的每个元素可以通过索引下标来访问，索引下标的范围是从0开始到数组长度减1的位置。内置的len函数将返回数组中元素的个数。

var a [3]int // array of 3 integers

fmt.Println(a[0]) // print the first element

fmt.Println(a[len(a)-1]) // print the last element, a[2]

// Print the indices and elements.

for i, v := range a {

fmt.Printf("%d %d\n", i, v)

}

// Print the elements only.

for \_, v := range a {

fmt.Printf("%d\n", v)

}

默认情况下，数组的每个元素都被初始化为元素类型对应的零值，对于数字类型来说就是0,。我们也可以用数组字面值语法用一组值来初始化数组：

var q [3]int = [3]int{1, 2, 3}

var r [3]int = [3]int{1, 2}

fmt.Println(r[2]) // "0"

在数组字面值中，如果在数组的长度位置出现的是“…”省略号，则表示数组的长度是根据初始化值的个数来计算。因此，上面q数组的定义可以简化为

q := [...]int{1, 2, 3}

fmt.Printf("%T\n", q) // "[3]int"

数组的长度是数组类型的一个组成部分，因此[3]int和[4]int是两种不同的数组类型。数组的长度必须是常量表达式，因为数组的长度需要在编译阶段确定。

q := [3]int{1, 2, 3}

q = [4]int{1, 2, 3, 4} // compile error: cannot assign [4]int to [3]int

我们将会发现，数组、slice、map和结构体字面值的写法都很相似。上面的形式是直接提供顺序初始化序列，但是也可以指定一个索引和对应值列表的方式初始化，就像下面这样：

type Currency int

const (

USD Currency = iota // 美元

EUR // 欧元

GBP // 英镑

RMB // 人民币

)

symbol := [...]string{USD: "$", EUR: "€", GBP: "￡", RMB: "￥"}

fmt.Println(RMB, symbol[RMB]) // "3 ￥"

在这种形式的数组字面值形式中，初始化索引的顺序是无关紧要的，而且没用到的索引可以省略，和前面提到的规则一样，未指定初始值的元素将用零值初始化。例如：

r := [...]int{99: -1}

定义了一个含有100个元素的数组r，最后一个元素被初始化为-1，其它元素都是用0初始化。

如果一个数组的元素类型是可以相互比较的，那么数组类型也是可以相互比较的，这时候我们可以直接通过==比较运算符来比较两个数组，只有当两个数组的所有元素都是相等的时候数组才是相等的。不相等比较运算符!=遵循同样的规则。

a := [2]int{1, 2}

b := [...]int{1, 2}

c := [2]int{1, 3}

fmt.Println(a == b, a == c, b == c) // "true false false"

d := [3]int{1, 2}

fmt.Println(a == d) // "compile error: cannot compare [2]int == [3]int

作为一个真实的例子，crypto/sha256包的Sum256函数对一个任意的字节slice类型的数据生成一个对应的消息摘要。消息摘要有256bit大小，因此对应[32]byte数组类型。如果两个消息摘要是相同的，那么可以认为两个消息本身也是相同（译注：理论上有HASH码碰撞的情况，但是实际应用可以基本忽略）；如果消息摘要不同，那么消息本身必然也是不同的。下面的例子用SHA256算法分别生成“x”和“X”两个消息的摘要：

gopl.io/ch4/sha256

// The sha256 command computes the SHA256 hash (an array) of a string.

package main

import (

"crypto/sha256"

"fmt"

)

func main() {

c1 := sha256.Sum256([]byte("x"))

c2 := sha256.Sum256([]byte("X"))

fmt.Printf("%x\n%x\n%t\n%T\n", c1, c2, c1 == c2, c1)

// Output:

// 2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881

// 4b68ab3847feda7d6c62c1fbcbeebfa35eab7351ed5e78f4ddadea5df64b8015

// false

// [32]uint8

}

上面例子中，两个消息虽然只有一个字符的差异，但是生成的消息摘要则几乎有一半的bit位是不相同的。需要注意Printf函数的%x副词参数，它用于指定以十六进制的格式打印数组或slice全部的元素，%t副词参数是用于打印布尔型数据，%T副词参数是用于显示一个值对应的数据类型。

当调用一个函数的时候，函数的每个调用参数将会被赋值给函数内部的参数变量，所以函数参数变量接收的是一个复制的副本，并不是原始调用的变量。因为函数参数传递的机制导致传递大的数组类型将是低效的，并且对数组参数的任何的修改都是发生在复制的数组上，并不能直接修改调用时原始的数组变量。在这个方面，Go语言对待数组的方式和其他很多编程语言不同，其他编程语言可能会隐式地将数组作为引用或指针对象传入被调用的函数。

当然，我们可以显式地传入一个数组指针，那样的话函数通过指针对数组的任何修改都可以直接反馈到调用者。下面的函数用于给[32]byte类型的数组清零：

func zero(ptr \*[32]byte) {

for i := range ptr {

ptr[i] = 0

}

}

其实数组字面值[32]byte{}就可以生成一个32字节的数组。而且每个数组的元素都是零值初始化，也就是0,。因此，我们可以将上面的zero函数写的更简洁一点：

func zero(ptr \*[32]byte) {

\*ptr = [32]byte{}

}

虽然通过指针来传递数组参数是高效的，而且也允许在函数内部修改数组的值，但是数组依然是僵化的类型，因为数组的类型包含了僵化的长度信息。上面的zero函数并不能接收指向[16]byte类型数组的指针，而且也没有任何添加和删除数组元素的方法。由于这些原因，除了像SHA256这类需要处理特定大数组的特例外，数组依然很少用作函数参数；相反，我们一般使用slice来替代数组。

**练习4.1：**编写一个函数，计算两个SHA256哈希码中不同bit的数目。（参考2.6.2节的PopCount函数。）

**练习4.2：**编写一个函数，默认打印标准输入的以SHA256哈希码，也可以通过命令行标准参数选择SHA384或SHA512哈希算法。

### 4.2 Slice

Slice（切片）代表变长的序列，序列中每个元素都有相同的类型。一个slice类型一般写作[]T，其中T代表slice中元素的类型；slice的语法和数组很像，只是没有固定长度而已。

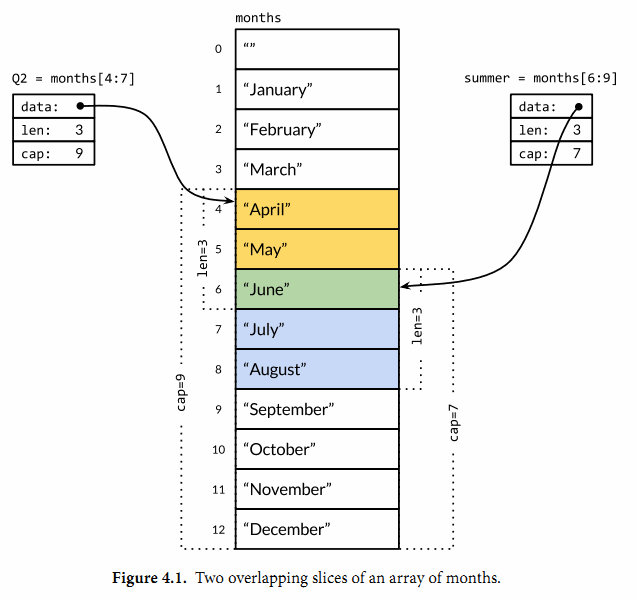
数组和slice之间有着紧密的联系。一个slice是一个轻量级的数据结构，提供了访问数组子序列（或者全部）元素的功能，而且slice的底层确实引用一个数组对象。一个slice由三个部分构成：指针、长度和容量。指针指向第一个slice元素对应的底层数组元素的地址，要注意的是slice的第一个元素并不一定就是数组的第一个元素。长度对应slice中元素的数目；长度不能超过容量，容量一般是从slice的开始位置到底层数据的结尾位置。内置的len和cap函数分别返回slice的长度和容量。

多个slice之间可以共享底层的数据，并且引用的数组部分区间可能重叠。图4.1显示了表示一年中每个月份名字的字符串数组，还有重叠引用了该数组的两个slice。数组这样定义

months := […]string{1: “January”, /\*…\*/, 12: “December”}

因此一月份是months[1]，十二月份是months[12]。通常，数组的第一个元素从索引0开始，但是月份一般是从1开始的，因此我们声明数组时直接跳过第0个元素，第0个元素会被自动初始化为空字符串。

slice的切片操作s[i:j]，其中0 <= i <= j <= cap(s)，用于创建一个新的slice，引用s的从第i个元素开始到第j-1个元素的子序列。新的slice将只有j-i个元素。如果i位置的索引被省略的话将使用0代替，如果j位置的索引被省略的话将使用len(s)代替。因此，months[1:13]切片操作将引用全部有效的月份，和months[1:]操作等价；months[:]切片操作则是引用整个数组。让我们分别定义表示第二季度和北方夏天月份的slice，它们有重叠部分：



Q2 := months[4:7]

summer := months[6:9]

fmt.Println(Q2) // [April May June]

fmt.Println(summer) // [June July August]

两个slice都包含了六月份，下面的代码是一个包含相同月份的测试（性能较低）：

for \_, s := range summer {

for \_, q := range Q2 {

if s == q {

fmt.Printf("%s appears in both\n", s)

}

}

}

如果切片操作超出cap(s)的上限将导致一个panic异常，但是超出len(s)则是意味着扩展了slice，因为新slice的长度会变大：

fmt.Println(summer[:20]) // panic: out of range

endlessSummer := summer[:5] // extend a slice (within capacity)

fmt.Println(endlessSummer) // [June July August September October]

另外，字符串的切片操作和[]byte字节类型切片的切片操作是类似的。都写作x[m:n]，并且都是返回一个原始字节序列的子序列，底层都是共享之前的底层数组，因此这种操作都是常量时间复杂度。x[m:n]切片操作对于字符串则生成一个新字符串，如果x是[]byte的话则生产一个新的[]byte。

因为slice值包含指向第一个slice元素的指针，因此向函数传递slice将允许在函数内部修改底层数组的元素。换句话说，复制一个slice只是对底层的数组创建了一个新的slice别名（§2.3.2）。下面的reverse函数在原内存空间将[]int类型的slice反转，而且它可以用于任意长度的slice。

gopl.io/ch4/rev

// reverse reverses a slice of ints in place.

func reverse(s []int) {

for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {

s[i], s[j] = s[j], s[i]

}

}

这里我们反转数组的应用：

a := [...]int{0, 1, 2, 3, 4, 5}

reverse(a[:])

fmt.Println(a) // "[5 4 3 2 1 0]"

一种将slice元素循环向左旋转n个元素的方法是三次调用reverse反转函数，第一次是反转开头的n个元素，然后是反转剩下的元素，最后是反转整个slice的元素。（如果是向右循环旋转，则将第三个函数调用到第一个调用位置就可以了。）

s := []int{0, 1, 2, 3, 4, 5}

// Rotate s left by two positions

reverse(s[:2])

reverse(s[2:])

reverse(s)

fmt.Println(s) // "[2 3 4 5 0 1]"

要注意的是slice类型的变量s和数组类型的变量a的初始胡语法的差异。slice和数组的字面值语法很类似，它们都是用花括弧包含一系列的初始化元素，但是对于slice并没有指明序列的长度。这会隐式地创建一个合适大小的数组，然后slice的指针指向底层的数组。就像数组字面值一样，slice的字面值也可以按顺序指定初始化值序列，或者是通过索引和元素值指定，或者用两种风格的混合语法初始化。

和数组不同的是，slice之间不能比较，因此我们不能使用==操作符来判断两个slice是否含有全部相等元素。不过标准库提供了高度优化的bytes.Equal函数来判断两个字节型slice是否相等（[]byte），但是对于其他类型的slice，我们必须自己展开每个元素进行比较：

func equal(x, y []string) bool {

if len(x) != len(y) {

return false

}

for i := range x {

if x[i] != y[i] {

return false

}

}

return true

}

上面关于两个slice的深度相等测试，运行的时间并不比支持==操作的数组或字符串更多，但是为何slice不直接支持比较运算符呢？这方面有两个原因。第一个原因，一个slice的元素是间接引用的，一个slice甚至可以包含自身。虽然有很多办法处理这种情形，但是没有一个是简单有效的。

第二个原因，因为slice的元素是间接引用的，一个固定的slice值（译注：指slice本身的值，而不是元素的值）在不同的时刻可能包含不同的元素，因为底层数组的元素可能会被修改。而例如Go语言中map的key只做简单的浅拷贝，它要求key在整个生命周期内保持不变性（译注：例如slice扩容，就会导致其本身的值/地址变化）。而用深度相等判断的话，显然在map的key这种场合不合适。对于像指针或chan之类的引用类型，==相等测试可以判断两个是否引用相同的对象。一个针对slice的浅相等测试的==操作符可能是有一定用处的，也能临时解决map类型的key问题，但是slice和数组不同的相等测试行为会让人困惑。因此，安全的做法是直接禁止slice之间的比较操作。

slice唯一合法的比较操作是和nil比较，例如：

if summer == nil { /\*…\*/ }

一个零值的slice等于nil。一个nil值的slice并没有底层数组。一个nil值的slice的长度和容量都是0，但是也有非nil值的slice的长度和容量也是0的，例如[]int{}或make([]int,3)[3:]。与任意类型的nil值一样，我们可以用[]int(nil)类型转换表达式来生成一个对应类型slice的nil值。

var s []int // len(s) == 0, s == nil

s = nil // len(s) == 0, s == nil

s = []int(nil) // len(s) == 0, s == nil

s = []int{} // len(s) == 0, s != nil

如果你需要测试一个slice是否是空的，使用len(s) == 0 来判断，而不应该用s == nil 来判断。除了和nil相等比较外，一个nil值的slice的行为和其它任意0长度的slice一样；例如reverse(nil)也是安全的。除了文档已经明确说明的地方，所有的Go语言函数应该以相同的方式对待nil值的slice和0长度的slice。

内置的make函数创建一个指定元素类型、长度和容量的slice。容量部分可以省略，在这种情况下，容量将等于长度。

make([]T, len)

make([]T, len, cap) // same as make([]T, cap)[:len]

在底层，make创建了一个匿名的数组变量，然后返回一个slice；只有通过返回的slice才能引用底层匿名的数组变量。在第一种语句中，slice是整个数组的view。在第二个语句中，slice只引用了底层数组的前len个元素，但是容量将包含整个数组。额外的元素是留给未来的增长用的。

#### 4.2.1 append函数

内置的append函数用于向slice追加元素：

var runes []rune

for \_, r := range "Hello, 世界" {

runes = append(runes, r)

}

fmt.Printf("%q\n", runes) // "['H' 'e' 'l' 'l' 'o' ',' ' ' '世' '界']"

在循环中使用append函数构建一个由九个rune字符构成的slice，当然对应这个特殊的问题我们可以通过Go语言内置的[]rune(“hello, 世界”)转换操作完成。

append函数对于理解slice底层是如何工作的非常重要，所以让我们仔细查看究竟是发生了什么。下面是第一个版本的appendInt函数，专门用于处理[]int类型的slice：

gopl.io/ch4/append

func appendInt(x []int, y int) []int {

var z []int

zlen := len(x) + 1

if zlen <= cap(x) {

// There is room to grow. Extend the slice.

z = x[:zlen]

} else {

// There is insufficient space. Allocate a new array.

// Grow by doubling, for amortized linear complexity.

zcap := zlen

if zcap < 2\*len(x) {

zcap = 2 \* len(x)

}

z = make([]int, zlen, zcap)

copy(z, x) // a built-in function; see text

}

z[len(x)] = y

return z

}

每次调用appendInt函数，必须先检测slice底层数组是否有足够的容量来保存新添加的元素。如果有足够空间的话，直接扩展slice（依然在原有的底层数组之上），将新添加的y元素复制到新扩展的空间，并返回slice。因此，输入的x和输出的z共享相同的底层数组。

如果没有足够的增长空间的话，appendInt函数则会先分配一个足够大的slice用于保存新的结果，先将输入的x复制到新的空间，然后添加y元素。结果z和输入的x引用的将是不同的底层数组。

虽然通过循环复制元素更直接，不过内置的copy函数可以方便地将一个slice复制到另一个想同类型的slice。copy函数的第一个参数是要复制的目标slice，第二个参数是源slice，目标和源的位置顺序和dst = src赋值语句是一致的。两个slice可以共享同一个底层数组，甚至有重叠也没有问题。copy函数将返回成功复制的元素的个数（我们这里没有用到），等于两个slice中较小的长度，所以我们不用担心覆盖会超出目标slice的范围。

为了提高内存使用效率，新分配的数组一般略大于保存x和y所需要的最低大小。通过在每次扩展数组时直接将长度翻倍从而避免了多次内存分配，也确保了添加单个元素操作时的平均时间是一个常数。这个程序演示了效果：

func main() {

var x, y []int

for i := 0; i < 10; i++ {

y = appendInt(x, i)

fmt.Printf("%d cap=%d\t%v\n", i, cap(y), y)

x = y

}

}

每一次容量的变化都会导致重新分配内存和copy操作：

0 cap=1 [0]

1 cap=2 [0 1]

2 cap=4 [0 1 2]

3 cap=4 [0 1 2 3]

4 cap=8 [0 1 2 3 4]

5 cap=8 [0 1 2 3 4 5]

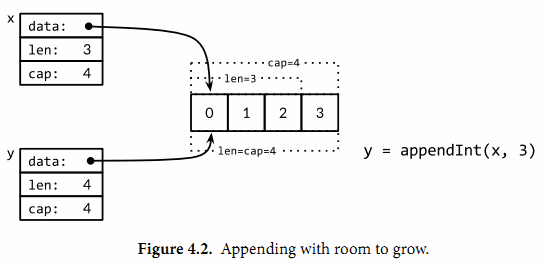
6 cap=8 [0 1 2 3 4 5 6]

7 cap=8 [0 1 2 3 4 5 6 7]

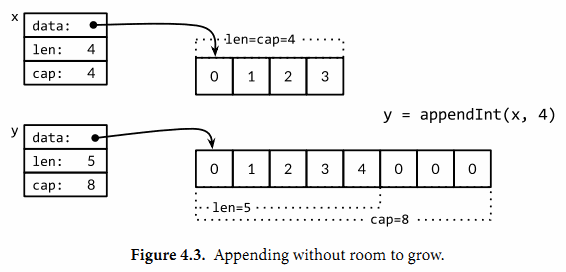
8 cap=16 [0 1 2 3 4 5 6 7 8]

9 cap=16 [0 1 2 3 4 5 6 7 8 9]

让我们仔细看i=3次的迭代。当时x包含了[0 1 2]三个元素，当时容量是4，因此可以简单将新的元素添加到末尾，不需要新的内存分配。然后新的y的长度和容量都是4，并且和x引用着相同的底层数组，如图4.2所示。



在下一次迭代时i=4，现在没有新的空余的空间了，因此appendInt函数分配一个容量为8的底层数组，将x的4个元素[0 1 2 3]复制到新空间的开头，然后添加新的元素i，新元素的值是4。新的y的长度是5，容量是8；后面有3个空闲的位置，三次迭代都不需要分配新的空间。当前迭代中，y和x是对应不同底层数组的view。这次操作如图4.3所示。



内置的append函数可能使用比appendInt更复杂的内存扩展策略。因此，通常我们并不知道append调用是否导致了内存的重新分配，因此我们也不能确认新的slice和原始的slice是否引用的是相同的底层数组空间。同样，我们不能确认在原先的slice上的操作是否会影响到新的slice。因此，通常是将append返回的结果直接赋值给输入的slice变量：

runes = append(runes, r)

更新slice变量不仅对调用append函数是必要的，实际上对应任何可能导致长度、容量或底层数组变化的操作都是必要的。要正确地使用slice，需要记住尽管底层数组的元素是间接访问的，但是slice对应结构体本身的指针、长度和容量部分是直接访问的。要更新这些信息需要像上面例子那样一个显式的赋值操作。从这个角度看，slice并不是一个纯粹的引用类型，它实际上是一个类似下面结构体的聚合类型：

type IntSlice struct {

ptr \*int

len, cap int

}

我们的appendInt函数每次只能向slice追加一个元素，但是内置的append函数则可以追加多个元素，甚至追加一个slice。

var x []int

x = append(x, 1)

x = append(x, 2, 3)

x = append(x, 4, 5, 6)

x = append(x, x...) // append the slice x

fmt.Println(x) // "[1 2 3 4 5 6 1 2 3 4 5 6]"

通过下面的小修改，我们可以达到append函数类似的功能。其中在appendInt函数参数中的最后的“…”省略号表示接收变长的参数为slice。我们将在5.7节详细解释这种特性。

func appendInt(x []int, y ...int) []int {

var z []int

zlen := len(x) + len(y)

// ... expand z to at least zlen...

copy(z[len(x):], y)

return z

}

为了避免重复，和前面相同的代码并没有显示。

#### 4.2.2 slice内存技巧

让我们看看更多的例子，比如旋转slice、反转slice或在slice原有内存空间修改元素。给定一个字符串列表，下面的nonempty函数将在原有slice内存空间之上返回不包含空字符串的列表：

gopl.io/ch4/nonempty

// Nonempty is an example of an in-place slice algorithm

package main

import "fmt"

// nonempty returns a slice holding only the non-empty strings.

// The underlying array is modified during the call.

func nonempty(strings []string) []string {

i := 0

for \_, s := range strings {

if s != "" {

strings[i] = s

i++

}

}

return strings[:i]

}

比较微妙的地方是，输入的slice和输出的slice共享一个底层数组。这可以避免分配另一个数组，不过原来的数据将可能会被覆盖，正如下面两个打印语句看到的那样：

data := []string{"one", "", "three"}

fmt.Printf("%q\n", nonempty(data)) // "["one" "three"]"

fmt.Printf("%q\n", data) // "["one" "three" "three"]"

因此我们通常会这样使用nonempty函数：data = nonempty(data)。

nonempty函数也可以使用append函数实现：

func nonempty2(strings []string) []string {

out := strings[:0] // zero-length slice of original

for \_, s := range strings {

if s != "" {

out = append(out, s)

}

}

return out

}

无论如何实现，以这种方式重用一个slice一般都要求最多为每个输入值产生一个输出值，事实上很多这类算法都是用来过滤或合并序列中相邻的元素。这种slice用法是比较复杂的技巧，虽然使用到了slice的一些技巧，但是对于某些场合是比较清晰和有效的。

一个slice可以用来模拟一个stack。最初给定的空slice对应一个空的stack，然后可以使用append函数将新的值压入stack：

stack = append(stack, v) // push v

stack的顶部位置对应slice的最后一个元素：

top := stack[len(stack)-1] // top of stack

通过收缩stack可以弹出栈顶的元素

stack = stack[:len(stack)-1] // pop

要删除slice中间的某个元素并保存原有的元素顺序，可以通过内置的copy函数将后面的字slice向前依次移动一位完成：

func remove(slice []int, i int) []int {

copy(slice[i:], slice[i+1:])

return slice[:len(slice)-1]

}

func main() {

s := []int{5, 6, 7, 8, 9}

fmt.Println(remove(s, 2)) // "[5 6 8 9]"

}

如果删除元素后不用保持原来顺序的话，我们可以简单的用最后一个元素覆盖被删除的元素：

func remove(slice []int, i int) []int {

slice[i] = slice[len(slice)-1]

return slice[:len(slice)-1]

}

func main() {

s := []int{5, 6, 7, 8, 9}

fmt.Println(remove(s, 2)) // " [5 6 9 8]"

}

**练习4.3：**重写reverse函数，使用数组指针代替slice。

**练习4.4：**编写一个rotate函数，通过一次循环完成旋转。

**练习4.5：**写一个函数在原地完成消除[]string中相邻重复的字符串的操作。

**练习4.6：**编写一个函数，原地将一个UTF-8编码的[]byte类型的slice中相邻的空格（参考unicode.IsSpace）替换成一个空格返回

**练习4.7：**修改reverse函数用于原地反转UTF-8编码的[]byte。是否可以不用分配额外的内存？

### 4.3 Map

哈希表是一种巧妙并且实用的数据结构。它是一个无序的key/value对的集合，其中所有的key都是不同的，然后通过给定的key可以在常数时间复杂度内检索、更新或删除对应的value。

在Go语言中，一个map就是一个哈希表的引用，map类型可以写为map[K]V，其中K和V分别对应key和value。map中所有的key都有相同的类型，所有的value也有着相同的类型，但是key和value之间可以是不同的数据类型。其中K对应的key必须是支持==比较运算符的数据类型，所以map可以通过测试key是否相等来判断是否已经存在。虽然浮点数类型也是支持相等比较符比较的，但是将浮点数用做key类型则是一个怀的想法，正如第三章提到的，最坏的情况是可能出现的NaN和任何浮点数都不相等。对于V对应的value数据类型则没有任何的限制。

内置的make函数可以创建一个map：

ages := make(map[string]int) // mapping from strings to ints

我们也可以用map字面值的语法创建map，同时还可以指定一些最初的key/value：

ages := map[string]int{

"alice": 31,

"charlie": 34,

}

这相当于：

ages := make(map[string]int)

ages ["alice"] = 31

ages ["charlie"] = 34

因此，另一种创建空的map的表达式是map[string]int{}。

map中的元素通过key对应的下标语法访问：

ages["alice"] = 32

fmt.Println(ages["alice"]) // "32

使用内置的delete函数可以删除元素：

delete(ages, "alice") // remove element ages["alice"]

所有这些操作是安全的，即使这些元素不在map中也没有关系；如果一个查找失败将返回value类型对应的零值，例如，即使map中不存在“bob”下面的代码也可以正常工作，因为ages[“bob”]失败时将返回0.

ages["bob"] = ages["bob"] + 1 // happy birthday!

而且x += y和x++等简短赋值语法也可以用在map上，所以上面的代码可以改写成

ages["bob"] += 1

更简单的写法

ages["bob"]++

但是map中的元素并不是一个变量，因此我们不能对map的元素进行取址操作：

\_ = &ages["bob"] // compile error: cannot take address of map element

禁止对map元素去址的原因是map可能随着元素数量的增长而重新分配更大的内存空间，从而可能导致之前的地址无效。

要想遍历map中全部的key/value对的话，可以使用range风格的for循环实现，和之前的slice遍历语法类似。下面的迭代语句将在每次迭代时设置name和age变量，它们对应下一个键值对：

for name, age := range ages {

fmt.Printf("%s\t%d\n", name, age)

}

map的迭代顺序是不确定的，并且不同的哈希函数实现可能导致不同的遍历顺序。在实践中，遍历的顺序是随机的，每一次遍历的顺序都不相同。这是故意的，每次都使用随机的遍历顺序可以强制要求程序不会依赖具体的哈希函数实现。如果要按顺序遍历key/value对，我们必须显式地对key进行排序，可以使用sort包的Strings函数对字符串slice进行排序。下面是常见的处理方式：

var names []string

for name := range ages {

names = append(names, name)

}

sort.Strings(names)

for \_, name := range names {

fmt.Printf("%s\t%d\n", name, ages[name])

}

因为我们一开始就知道names的最终大小，因此给slice分配一个合适的大小将会更有效。下面的代码创建了一个空的slice，但是slice的容量刚好可以放下map中全部的key：

names := make([]string, 0, len(ages))

在上面的第一个range循环中，我们只关心map中的key，所以我们忽略了第二个循环变量。在第二个循环中，我们只关心names中的名字，所以我们使用“\_”空白标识符来忽略第一个循环变量，也就是迭代slice时的索引。

map类型的零值是nil，也就是没有引用任何哈希表。

var ages map[string]int

fmt.Println(ages == nil) // "true"

fmt.Println(len(ages) == 0) // "true"

map上的大部分操作，包括查找、删除、len和range循环都可以安全工作在nil值的map上，它们的行为和一个空的map类似。但是向一个nil值的map存入元素将导致一个panic异常：

ages["carol"] = 21 // panic: assignment to entry in nil map

在向map存数据前必须先创建map。

通过key作为索引下标来访问map将产生一个value。如果key在map中是存在的，那么将得到与key对应的value；如果key不存在，那么将得到value对应类型的零值，正如我们前面看到的ages[“bob”]那样。这个规则很实用，但是有时候可能需要知道对应的元素是否真的是在map之中。例如，如果元素类型是一个数字，你可能需要区分一个已经存在的0，和不存在而返回零值的0，可以像下面这样测试：

age, ok := ages["bob"]

if !ok {

/\* "bob" is not a key in this map; age == 0. \*/

}

你会经常看到将这两个结合起来使用，像这样：

if age, ok := ages["bob"]; !ok {

/\* ... \*/

}

在这种场景下，map的下标语法将产生两个值；第二个是一个布尔值，用于报告元素是否真的存在。布尔变量一般命名为ok，特别适合马上用于if条件判断部分。

和slice一样，map之间也不能进行相等比较；唯一的例外是和nil进行比较。要判断两个map是否包含相同的key和value，我们必须通过一个循环实现：

func equal(x, y map[string]int) bool {

if len(x) != len(y) {

return false

}

for k, xv := range x {

if yv, ok := y[k]; !ok || yv != xv {

return false

}

}

return true

}

从例子中可以看到如何用!ok来区分元素不存在，与元素存在但为0的。我们不能简单地用xv != y[k]判断，那样会导致在判定下面两个map时产生错误的结果：

// True if equal is written incorrectly

mapEqual(map[string]int{"A": 0}, map[string]int{"B": 42})

Go语言中并没有提供一个set类型，但是map中的key也是不相同的，可以用map是实现类似set的功能。为了说明这一点，下面的dedup程序读取多行输入，但是只打印第一次出现的行。（它是1.3节中出现的dup程序的变体。）dedup程序通过map来表示所有的输入行所对应的set集合，以确保已经在集合存在的行不会被重复打印。

gopl.io/ch4/dedup

func main() {

seen := make(map[string]bool) // a set of strings

input := bufio.NewScanner(os.Stdin)

for input.Scan() {

line := input.Text()

if !seen[line] {

seen[line] = true

fmt.Println(line)

}

}

if err := input.Err(); err != nil {

fmt.Fprintf(os.Stderr, "dedup: %v\n", err)

os.Exit(1)

}

}

Go程序员将这种忽略value的map当作一个字符串集合，并非所有map[string]bool类型value都是无关紧要的；有一些则可能会同时包含true和false的值。

有时候我们需要一个map或set的key是slice类型，但是map的key必须是可比较的类型，但是slice并不满足这个条件。不过，我们可以通过两个步骤绕过这个限制。第一步，定义一个辅助函数k，将slice转为map对应的string类型的key，确保只有x和y相等时k(x) == k(y)才成立。然后创建一个key为string类型的map，在每次对map操作时先用k辅助函数将slice转换为string类型。

下面的例子演示了如何使用map来记录提交相同的字符串列表的次数。它使用了fmt.Sprintf函数将字符串列表转换为一个字符串以用于map的key，通过%q参数忠实的记录每个字符串元素的信息：

var m = make(map[string]int)

func k(list []string) string {

return fmt.Sprintf("%q", list)

}

func Add(list []string) {

m[k(list)]++

}

func Count(list []string) int {

return m[k(list)]

}

使用同样的技术可以处理任何不可比较的key类型，而不仅仅是slice类型。这种技术对于想使用自定义key比较函数的时候也很有用，例如在比较字符串的时候忽略大小写。同时，辅助函数k(x)也不一定是字符串类型，它可以返回任何可比较的类型，例如整数、数组或结构体等。

这是map的另一个例子，下面的程序用于统计输入中每个Unicode码点出现的次数。虽然Unicode全部码点的数量巨大，但是出现在特定文档中的字符种类并没有多少，使用map可以用比较自然的方式来跟踪那些出现过的字符的次数。

gopl.io/ch4/charcount

// Charcount computes counts of Unicode characters.

package main

import (

"bufio"

"fmt"

"io"

"os"

"unicode"

"unicode/utf8"

)

func main() {

counts := make(map[rune]int) // counts of Unicode characters

var utflen [utf8.UTFMax + 1]int // count of lengths of UTF-8 encodings

invalid := 0 // count of invalid UTF-8 characters

in := bufio.NewReader(os.Stdin)

for {

r, n, err := in.ReadRune() // returns rune, nbytes, error

if err == io.EOF {

break

}

if err != nil {

fmt.Fprintf(os.Stderr, "charcount: %v\n", err)

os.Exit(1)

}

if r == unicode.ReplacementChar && n == 1 {

invalid++

continue

}

counts[r]++

utflen[n]++

}

fmt.Printf("rune\tcount\n")

for c, n := range counts {

fmt.Printf("%q\t%d\n", c, n)

}

fmt.Print("\nlen\tcount\n")

for i, n := range utflen {

if i > 0 {

fmt.Printf("%d\t%d\n", i, n)

}

}

if invalid > 0 {

fmt.Printf("\n%d invalid UTF-8 characters\n", invalid)

}

}

ReadRune方法执行UTF-8编码并返回三个值：解码的rune字符的值，字符UTF-8编码后的长度，和一个错误值。我们可预期的错误值只有对应文件结尾的io.EOF。如果输入的是无效的UTF-8编码的字符，返回的将是unicode.ReplacementChar表示无效字符，并且编码长度是1。

charcount程序同时打印不同UTF-8编码长度的字符数目。对此，map并不是一个合适的数据结构；因为UTF-8编码的长度总是从1-utf8.UTFMax（最大是4个字节），使用数组将更有效。

作为一个实验，我们用charcount程序对英文版原稿的字符进行了统计。虽然大部分是英语，但是也有一些非ASCII字符。下面是排名前10的非ASCII字符：



下面是不同UTF-8编码长度的字符的数目：

len count

1 765391

2 60

3 70

4 0

Map的value类型也可以是一个聚合类型，比如是一个map或slice。在下面的代码中，图graph的key类型是一个字符串，value类型map[string]bool代表一个字符串集合。从概念上讲，graph将一个字符串类型的key映射到一组相关的字符串集合，它们指向新的graph的key。

gopl.io/ch4/graph

var graph = make(map[string]map[string]bool)

func addEdge(from, to string) {

edges := graph[from]

if edges == nil {

edges = make(map[string]bool)

graph[from] = edges

}

edges[to] = true

}

func hasEdge(from, to string) bool {

return graph[from][to]

}

其中addEdge函数惰性初始化map是一个惯用方式，也就是说在每个值首次作为key时才初始化。addEdge函数显示了如何让map的零值也能正常工作；即使from到to的边不存在，graph[from][to]依然可以返回一个有意义的结果。

**练习4.8：**修改charcount程序，使用unicode.IsLetter等相关的函数，统计字母、数字等Unicode中不同的字符类别。

**练习4.9：**编写一个程序wordfreq程序，报告输入文本中每个单词出现的频率。在第一次调用Scan前先调用input.Split(bufio.ScanWords)函数，这样可以按单词而不是按行输入。

### 4.4 结构体

结构体是一种聚合的数据类型，是由零个或多个任意类型的值聚合成的实体。每个值称为结构体的成员。用结构体的经典案例是处理公司的员工信息，每个员工信息包含一个唯一的员工编号、员工的名字、家庭住址、出生日期、工作岗位、薪资、上级领导等等。所有的这些信息都需要绑定到一个实体中，可以作为一个整体单元被复制，作为函数的参数或返回值，或者是被存储到数组中，等等。

下面两个语句声明了一个叫Employee的命名的结构体类型，并且声明了一个Employee类型的变量dilbert：

type Employee struct {

ID int

Name string

Address string

Dob time.Time

Position string

Salary int

ManagerID int

}

var dilbert Employee

dilbert结构体变量的成员可以通过点操作符访问，比如dilbert.Name和dilbert.DoB。因为dilbert是一个变量，它所有的成员也同样是变量，我们可以直接对每个成员赋值：

dilbert.Salary -= 5000 // demoted, for writing too few lines of code

或者是对成员取地址，然后通过指针访问：

position := &dilbert.Position

\*position = "Senior " + \*position // promoted, for outsourcing to Elbonia

点操作符也可以和指向结构体的指针一起工作：

var employeeOfTheMonth \*Employee = &dilbert

employeeOfTheMonth.Position += " (proactive team player)"

相当于下面语句

(\*employeeOfTheMonth).Position += " (proactive team player)"

下面的EmployeeByID函数将根据给定的员工ID返回对应的员工信息结构体的指针。我们可以使用点操作符来访问它里面的成员：

func EmployeeByID(id int) \*Employee {

/\* ... \*/

}

fmt.Println(EmployeeByID(dilbert.ManagerID).Position) // "Pointy-haired boss"

id := dilbert.ID

EmployeeByID(id).Salary = 0 // fired for ... no real reason

后面的语句通过EmployeeByID返回的结构体指针更新了Employee结构体的成员。如果将EmployeByID函数的返回值从\*Employee指针类型改为Employee值类型，那么更新语句将不能编译通过，因为在赋值语句的左边并不确定是一个变量（译注：调用函数返回的是值，并不是一个可取地址的变量）。

通常一行对应一个结构体成员，成员的名字在前类型在后，不过如果相邻的成员类型相同的话可以被合并到一行，就像下面的Name和Address成员那样：

type Employee struct {

ID int

Name, Address string

Dob time.Time

Position string

Salary int

ManagerID int

}

结构体成员的输入顺序也有重要的意义。我们也可以将Position成员合并（因为也是字符串类型），或者是交换Name和Address出现的先后顺序，那样的话就是定义了不同的结构体类型。通常，我们只是将相关的成员写到一起。

如果结构体成员名字是以大写字母开头的，那么该成员就是导出的；这是Go语言导出规则决定的。一个结构体可能同时包含导出和未导出的成员。

结构体类型往往是冗长的，因为它的每个成员可能都会占一行。虽然我们每次都可以重写整个结构体成员，但是重复会令人厌烦。因此，完整的结构体写法通常只在类型声明语句的地方出现，就像Employee类型声明语句那样。

一个命名为S的结构体类型将不能再包含S类型的成员：因为一个聚合的值不能包含它自身。（该限制同样适用于数组。）但是S类型的结构体可以包含\*S指针类型的成员，这可以让我们创建递归的数据结构，比如链表和树结构等。在下面的代码中，我们使用一个二叉树来实现一个插入排序：

gopl.io/ch4/treesort

type tree struct {

value int

left, right \*tree

}

// Sort sorts values in place.

func Sort(values []int) {

var root \*tree

for \_, v := range values {

root = add(root, v)

}

appendValues(values[:0], root)

}

// appendValues appends the elements of t to values in order

// and returns the resulting slice.

func appendValues(values []int, t \*tree) []int {

if t != nil {

values = appendValues(values, t.left)

values = append(values, t.value)

values = appendValues(values, t.right)

}

return values

}

func add(t \*tree, value int) \*tree {

if t == nil {

// Equivalent to return &tree{value: value}.

t = new(tree)

t.value = value

return t

}

if value < t.value {

t.left = add(t.left, value)

} else {

t.right = add(t.right, value)

}

return t

}

结构体类型的零值是每个成员都是零值。通常会将零值作为最合理的默认值。例如，对于bytes.Buffer类型，结构体初始值就是一个随时可用的空缓存，还有在第9章将会讲到的sync.Mutex的零值也是有效的未锁定状态。有时候这种零值可用的特性是自然获得的，但是也有些类型需要一些额外的工作。

如果结构体没有任何成员的话就是空结构体，写作struct{}。它的大小为0，也不包含任何信息，但是有时候依然是有价值的。有些Go语言程序员用map来模拟set数据结构时，用它来代替map中布尔类型的value，只是强调key的重要性，但是因为节约的空间有限，而且语法比较复杂，所以我们通常会避免这样的用法。

seen := make(map[string]struct{}) // set of strings

// ...

if \_, ok := seen[s]; !ok {

seen[s] = struct{}{}

// ...first time seeing s...

}

#### 4.4.1 结构体字面值

结构体值也可以用结构体字面值表示，结构体字面值可以指定每个成员的值。

type Point struct {X, Y int}

p := Point{1, 2}

这里有两种形式的结构体字面值语法，上面的是第一种语法，要求以结构体成员定义的顺序为每个结构体成员指定一个字面值。它要求写代码和读代码的人要记住结构体的每个成员的类型和顺序，不过结构体成员有细微的调整就可能导致上述代码不能编译。因此，上述的语法一般只在定义结构体的包内部使用，或者是在较小的结构体中使用，这些结构体的成员排列比较规则，比如image.Point{x, y}或color.RGBA{red, green, blue, alpha}。

anim := gif.GIF{LoopCount: nframes}

其实更常用的是第二种写法，以成员名字和相应的值来初始化，可以包含部分或全部的成员，如1.4节的Lissajous程序的写法：

在这种形式的结构体字面值写法中，如果成员被忽略的话将默认用零值。因为提供了成员的名字，所以成员出现的顺序并不重要。

两种不同形式的写法不能混合使用。而且，你不能企图在外部包中用第一种顺序赋值的技巧来偷偷地初始化结构体中未导出的成员。

package p

type T struct {a, b int } // a and b are not exported

package q

import p

var \_ = p.T{a: 1, b: 2} // compile error: can't reference a, b

var \_ = p.T{1, 2} // compile error: can't reference a, b

虽然上面最后一行代码的编译错误信息中并没有显式提到未导出的成员，但是这样企图隐式使用未导出成员的行为也是不允许的。

结构体可以作为函数的参数和返回值。例如，这个Scale函数将Point类型的值缩放后返回：

func Scale(p Point, factor int) Point {

return Point{p.X \* factor, p.Y \* factor}

}

fmt.Println(Scale(Point{1, 2}, 5))

如果考虑效率的话，较大的结构体通常会用指针的方式传入和返回，

func Bonus(e \*Employee, percent int) int {

return e.Salary \* percent / 100

}

如果要在函数内部修改结构体成员的话，用指针传入是必须的；因为在Go语言中，所有的函数参数都是值拷贝传入的，函数参数将不再是函数调用时的原始变量。

func AwardAnnualRaise(e \*Employee) {

e.Salary = e.Salary \* 105 / 100

}

因为结构体通常通过指针处理，可以用下面的写法来创建并初始化一个结构体变量，并返回结构体的地址：

pp := &Point{1, 2}

它和下面的语句是等价的

pp := new(Point)

\*pp = Point{1, 2}

不过&Point{1, 2}写法可以直接在表达式中使用，比如一个函数调用。

#### 4.4.2 结构体比较

如果结构体的全部成员都是可以比较的，那么结构体也是可以比较的，那样的话两个结构体将可以使用==或!=运算符进行比较。相等比较运算符==将比较两个结构体的每个成员，因此下面两个比较的表达式是等价的：

type Point struct {X, Y int}

p := Point{1, 2}

q := Point{2, 1}

fmt.Println(p.X == q.X && p.Y == q.Y) // "false"

fmt.Println(p == q) // "false"

可比较的结构体类型和其他可比较的类型一样，可以用于map的key类型。

type address struct {

hostname string

port int

}

hits := make(map[address]int)

hits[address{"golang.org", 443}]++

#### 4.4.3 结构体嵌入和匿名成员

在本节中，我们将看到如何使用Go语言提供的不同寻常的结构体嵌入机制让一个命名的结构体包含另一个结构体类型的匿名成员，这样就可以通过简单的点运算符x.f来访问匿名成员链中嵌套的x.d.e.f成员。

考虑一个二维的绘图程序，提供了一个各种图形的库，例如矩形、椭圆形、星形和轮形等几何形状。这里是其中两个的定义：

type Circle struct {

X, Y, Radius int

}

type Wheel struct {

X, Y, Radius, Spokes int

}

一个Circle代表的圆形类型包含了标准圆心的X和Y坐标信息，和一个Radius表示的半径信息。一个Wheel轮形除了包含Circle类型所有的全部成员外，还增加了Spokes表示径向辐条的数量。我们可以这样创建一个wheel变量：

var w Wheel

w.X = 8

w.Y = 8

w.Radius = 5

w.Spokes = 20

随着库中几何形状数量的增多，我们一定会注意到它们之间的相似和重复之处，所以我们可能为了便于维护而将相同的属性独立出来：

type Point struct {

X, Y int

}

type Circle struct {

Center Point

Radius int

}

type Wheel struct {

Circle Circle

Spokes int

}

这样改动之后结构体类型变的清晰了，但是这种修改同时也导致了访问每个成员变得繁琐：

var w Wheel

w.Circle.Center.X = 8

w.Circle.Center.Y = 8

w.Circle.Radius = 5

w.Spokes = 20

Go语言有一个特性让我们只声明一个成员对应的数据类型而不指定成员的名字；这类成员就叫匿名成员。匿名成员的数据类型必须是命名的类型或指向一个命名的类型的指针。下面的代码中，Circle和Wheel各自都有一个匿名成员。我们可以说Point类型被嵌入到了Circle结构体，同时Circle类型被嵌入到了Wheel结构体。

type Circle struct {

Point

Radius int

}

type Wheel struct {

Circle

Spokes int

}

得益于匿名嵌入的特性，我们可以直接访问叶子属性而不需要给出完整的路径：

var w Wheel

w.X = 8 // equivalent to w.Circle.Point.X = 8

w.Y = 8 // equivalent to w.Circle.Point.Y = 8

w.Radius = 5 // equivalent to w.Circle.Radius = 5

w.Spokes = 20

在右边的注释中给出的显式形式访问这些叶子成员的语法依然有效，因此匿名成员并不是真的无法访问了。其中匿名成员Circle和Point都有自己的名字——就是命名的类型名字——但是这些名字在点操作符中是可选的。我们在访问子成员的时候可以忽略任何匿名成员部分。

不幸的是，结构体字面值并没有简短表示匿名成员的语法，因此下面的语句都不能编译通过：

w = Wheel{8, 8, 5, 20} // compile error: unknown fields

w = Wheel{X: 8, Y:8, Radius: 5, Spokes: 20} // compile error: unknown fields

结构体字面值必须遵循形状类型声明时的结构，所以我们只能用下面的两种语法，它们彼此是等价的：

gopl.io/ch4/embed

w = Wheel{Circle{Point{8, 8}, 5}, 20}

w = Wheel{

Circle: Circle{

Point: Point{X: 8, Y: 8},

Radius: 5,

},

Spokes: 20, // NOTE: trailing comma necessary here (and at Radius)

}

fmt.Printf("%#v\n", w)

// Output:

// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}

w.X = 42

fmt.Printf("%#v\n", w)

// Output:

// Wheel{Circle:Circle{Point:Point{X:52, Y:8}, Radius:5}, Spokes:20}

需要注意的是Printf函数中%v参数所包含的#副词，它表示用和Go语言类似的语法打印值。对于结构体类型来说，将包含每个成员的名字。

因为匿名成员也有一个隐式的名字，因此不能同时包含两个类型相同的匿名成员，这会导致名字冲突。同时，因为成员的名字是由其类型隐式地决定的，所以匿名成员也有可见性的规则约束。在上面的例子中，Point和Circle匿名成员都是导出的。即使它们不导出（比如改成小写字母开头的point和circle），我们依然可以用简短形式访问匿名成员嵌套的成员

w.X = 8 // equivalent to w.circle.point.X = 8

但是在包外部，因为circle和point没有导出，不能访问它们的成员，因此简短的匿名成员访问语法也是禁止的。

到目前为止，我们看到匿名成员特性只是对访问嵌套成员的点运算符提供了简短的语法糖。稍后，我们将会看到匿名成员并不要求是结构体成员；其实任何命名的类型都可以作为结构体的匿名成员。但是为什么要嵌入一个没有任何子成员类型的匿名成员类型呢？

答案是匿名类型的方法集。简短的点运算符语法可以用于选择匿名成员嵌套的成员，也可以用于访问它们的方法。实际上，外层的结构体不仅仅是获得了匿名成员类型的所有成员，而且也获得了该类型导出的全部的方法。这个机制可以用于将一些有简单行为的对象组合成有复杂行为的对象。组合是Go语言中面向对象编程的核心，我们将在6.3节中专门讨论。

### 4.5 JSON

JavaScript对象表示法（JSON）是一种用于发送和接收结构化信息的标准协议。在类似的协议中，JSON并不是唯一的一个标准协议。XML（§7.14）、ASN.1和Google的Protocol Buffers都是类似的协议，并且有各自的特色，但是由于简洁性、可读性和流行程度等原因，JSON是应用最广泛的一个。

Go语言对于这些标准格式的编码和解码都有良好的支持，由标准库中的encoding/json、encoding/xml、encoding/asn1等包提供支持（译注：Protocol Buffers的支持由github.com/golang/protobuf包提供），并且这类包都有着相似的API接口。本节，我们将对重要的encoding/json包的用法做个概述。

JSON是对JavaScript中各种类型的值——字符串、数字、布尔值和对象——Unicode文本编码。它可以用有效可读的方式表示第三章的基础数据类型和本站的数组、slice、结构体和map等聚合数据类型。

基本的JSON类型有数字（十进制或科学计数法）、布尔值（true和false）、字符串、其中字符串是以双引号包含的Unicode字符序列，支持和GO语言类似的反斜杠转义特性，不过JSON使用的是\Uhhhh转义数字来表示一个UTF-16编码（译注：UTF-16和UTF-8一样是一种变长的编码，有些Unicode码点较大的字符需要用4个字节表示；而且UTF-16还有大端和小端的问题），而不是Go语言的rune类型。

这些基础类型可以通过JSON的数组和对象类型进行递归组合。一个JSON数组是一个有序的值序列，写在一个方括号中并以逗号分隔；一个JSON数组可以用于编码Go语言的数组和slice。一个JSON对象是一个字符串到值的映射，写成一系列的name:value对形式，用花括号包含并以逗号分隔；JSON的对象类型可以用于编码Go语言的map类型（key类型是字符串）和结构体。例如：

|  |  |
| --- | --- |
| Boolean | true |
| number | -273.15 |
| string | “She said \”Hello, BF\”” |
| array | [“gold”, “silver”, “bronze” |
| object | {“year”:1980, “event”: “archery”, “medals”: [“gold”, “silver”, “bronze”]} |

考虑一个应用程序，该程序负责收集各种电影评论并提供反馈功能。它的Movie数据类型和一个典型的表示电影的值列表如下所示。（在结构体声明中，Year和Color成员后面的字符串面值是结构体成员Tag；我们稍后回解释它的作用。）

gopl.io/ch4/movie

type Movie struct {

Title string

Year int `json:"released"`

Color bool `json:"color,omitempty"`

Actors []string

}

var movies = []Movie{

{Title: "Casablanca", Year: 1942, Color: false, Actors: []string{"Humphrey Bogart", "Ingrid Bergman"}},

{Title: "Cool Hand Luke", Year: 1967, Color: true, Actors: []string{"Paul Newman"}},

{Title: "Bullitt", Year: 1968, Color: true, Actors: []string{"Steve McQueen", "Jacqueline Bisset"}},

}

这样的数据结构特别适合JSON格式，并且在两者之间相互转换也很容易。将一个Go语言中类似movies的结构体slice转为JSON的过程叫做编组（marshaling）。编组通过调用json.Marshal函数完成：

data, err := json.Marshal(movies)

if err != nil {

log.Fatalf("JSON marshaling failed: %s", err)

}

fmt.Printf("%s\n", data)

Marshal函数返回一个编码后的字节slice，包含很长的字符串，并且没有空白缩进；我们将它折行以便于显示：

[{"Title":"Casablanca","released":1942,"Actors":["Humphrey Bogart","Ingrid

Bergman"]},{"Title":"Cool Hand Luke","released":1967,"color":true,"Actors":["Paul Newman"]},{"Title":"Bullitt","released":1968,"color":true,"Actors":["Steve McQueen","Jacqueline Bisset"]}]

这种紧凑的表示形式虽然包含了全部的信息，但是很难阅读。为了生成便于阅读的格式，另一个json.MarshalIndent函数将产生整齐缩进的输出。该函数有两个额外的字符串参数用于表示每一行输出的前缀和每一个层级的缩进：

data, err := json.MarshalIndent(movies, "", " ")

if err != nil {

log.Fatalf("JSON marshaling failed: %s", err)

}

fmt.Printf("%s\n", data)

上面的代码将产生这样的输出（译注：在最后一个成员或元素后面并没有逗号分隔符）：

[

{

"Title": "Casablanca",

"released": 1942,

"Actors": [

"Humphrey Bogart",

"Ingrid Bergman"

]

},

{

"Title": "Cool Hand Luke",

"released": 1967,

"color": true,

"Actors": [

"Paul Newman"

]

},

{

"Title": "Bullitt",

"released": 1968,

"color": true,

"Actors": [

"Steve McQueen",

"Jacqueline Bisset"

]

}

]

在编码时，默认使用Go语言结构体的成员名字作为JSON的对象（通过reflect反射技术，我们将在12.6节讨论）。只有导出的结构体成员才会被编码，这也就是我们为什么选择用大写字母开头的成员名称。

细心的读者可能已经注意到，其中Year名字的成员在编码后变成了released，还有Color成员编码后变成了小写字母开头的color。这是因为结构体成员Tag所导致的。一个结构体成员Tag是和在编译阶段关联到该成员的元信息字符串：

Year int `json:"released"`

Color bool `json:"color,omitempty"`

结构体的成员Tag可以是任意的字符串面值，但是通常是一系列用空格分隔的key:”value”键值对序列；因为值中含有双引号字符，因此成员Tag一般用原生字符串面值的形式书写。json开头键名对应的值用于控制encoding/json包的编码和解码的行为，并且encoding/…下面其他的包也遵循这个约定。成员Tag中json对应值的第一部分用于指定JSON对象的名字，比如将Go语言中的TotalCount成员对应到JSON中的total\_count对象。Color成员的Tag还带了一个额外的omitempty选项，表示当Go语言结构体成员为空或零值时不生成该JSON对象（这里false为零值）。果然，Casablanca是一个黑白电影，并没有输出Color成员。

编码的逆操作是解码，对应将JSON数据解码为Go语言的数据结构，Go语言中一般叫unmarshaling，通过json.Unmarshal函数完成。下面的代码将JSON格式的电影数据解码为一个结构体slice，结构体中只有Title成员。通过定义合适的Go语言数据结构，我们可以选择性地解码JSON中感兴趣的成员。当Unmarshal函数调用返回，slice将被只含有Title信息的值填充，其它JSON成员将被忽略。

var titles []struct{ Title string }

if err := json.Unmarshal(data, &titles); err != nil {

log.Fatalf("JSON unmarshaling failed: %s", err)

}

fmt.Println(titles) // "[{Casablanca}{Cool Hand Luke}{Bullitt}]"

许多web服务都提供JSON接口，通过HTTP接口发送JSON格式请求并返回JSON格式的信息。为了说明这一点，我们通过Github的issue查询服务来演示类似的用法。首先，我们要定义合适的类型和常量：

gopl.io/ch4/github

// See https://developer.github.com/v3/search/#search-issues.

package github

import "time"

const IssuesURL = "https;//api.github.com/search/issues"

type IssuesSearchResult struct {

TotalCount int `json:"total\_count"`

Items []\*Issue

}

type Issue struct {

Number int

HTMLURL string `json:"html\_url"`

Title string

State string

User \*User

CreatedAt time.Time `json:"created\_at"`

Body string // in Markdown format

}

type User struct {

Login string

HTMLURL string `json:"html\_url"`

}

和前面一样，即使对应的JSON对象名是小写字母，每个结构体的成员名也是声明为大写字母开头的。因为有些JSON成员名字和Go结构体成员名字并不相同，因此需要Go语言结构体成员Tag来指定对应的JSON名字。同样，在解码的时候也需要做同样地处理，Github服务返回的信息比我们定义的要多很多。

SearchIssues函数发出一个HTTP请求，然后解码返回的JSON格式的结果。因为用户提供的查询条件可能包含类似？和&之类的特殊字符，为了避免对URL造成冲突，我们用url.QueryEscape来对查询中的特殊字符进行转义操作。

gopl.io/ch4/github

// SearchIssues queries the Github issue tracker.

func SearchIssues(terms []string) (\*IssuesSearchResult, error) {

q := url.QueryEscape(strings.Join(terms, " "))

resp, err := http.Get(IssuesURL + "?q=" + q)

if err != nil {

return nil, err

}

// We must close resp.Body on all execution paths.

// (Chapter 5 presents 'defer', which makes this simpler.)

if resp.StatusCode != http.StatusOK {

resp.Body.Close()

return nil, fmt.Errorf("search query failed: %s", resp.Status)

}

var result IssuesSearchResult

if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {

resp.Body.Close()

return nil, err

}

resp.Body.Close()

return &result, nil

}

在早些的例子中，我们使用了json.Unmarshal函数来将JSON格式的字符串解码为字节slice。但是这个例子中，我们使用了基于流式的解码器json.Decoder，它可以从一个输入流解码JSON数据，尽管这不是必须的。如您所料，还有一个针对输出流的json.Encoder编码对象。

我们调用Decode方法来填充变量。这里有很多方法可以格式化结构。下面是最简单的一种，以一个固定宽打印每个issue，但是在下一节我们将看到如何利用模板来输出复杂的格式。

gopl.io/ch4/issues

// Issues prints a table of GitHub issues matching the search terms.

package main

import (

"fmt"

"log"

"mygopl/ch04/github"

"os"

)

func main() {

result, err := github.SearchIssues(os.Args[1:])

if err != nil {

log.Fatal(err)

}

fmt.Printf("%d issues:\n", result.TotalCount)

for \_, item := range result.Items {

fmt.Printf("#%-5d %9.9s %.55s\n", item.Number, item.User.Login, item.Title)

}

}

通过命令行参数指定检索条件。下面的命令是查询Go语言项目中和JSON解码相关的问题，还有查询返回的结果：

E:\GoProjects\src\mygopl\ch04\issues>go run main.go repo:golang/go is:open

json decoder

50 issues:

#33416 bserdar encoding/json: This CL adds Decoder.InternKeys

#40128 rogpeppe proposal: encoding/json: garbage-free reading of tokens

#40127 rogpeppe proposal: encoding/json: add Encoder.EncodeToken method

#34647 babolivie encoding/json: fix byte counter increments when using d

#12001 lukescott encoding/json: Marshaler/Unmarshaler not stream friendl

#29035 jaswdr proposal: encoding/json: add error var to compare the

#11046 kurin encoding/json: Decoder internally buffers full input

#36225 dsnet encoding/json: the Decoder.Decode API lends itself to m

#5901 rsc encoding/json: allow per-Encoder/per-Decoder registrati

#32779 rsc encoding/json: memoize strings during decode

#34543 maxatome encoding/json: Unmarshal & json.(\*Decoder).Token report

#14750 cyberphon encoding/json: parser ignores the case of member names

#28923 mvdan encoding/json: speed up the decoding scanner

#31701 lr1980 encoding/json: second decode after error impossible

#28143 arp242 proposal: encoding/json: add "readonly" tag

#16212 josharian encoding/json: do all reflect work before decoding

#6647 btracey x/tools/cmd/godoc: display type kind of each named type

#34564 mdempsky go/internal/gcimporter: single source of truth for deco

#30301 zelch encoding/xml: option to treat unknown fields as an erro

#33854 Qhesz encoding/json: unmarshal option to treat omitted fields

#26946 deuill encoding/json: clarify what happens when unmarshaling i

#22752 buyology proposal: encoding/json: add access to the underlying d

#33835 Qhesz encoding/json: unmarshalling null into non-nullable gol

#22480 blixt proposal: encoding/json: add omitnil option

#20754 rsc encoding/xml: unmarshal only processes first XML elemen

#27179 lavalamp encoding/json: no way to preserve the order of map keys

#28189 adnsv encoding/json: confusing errors when unmarshaling custo

#7872 extempora encoding/json: Encoder internally buffers full output

#33714 flimzy proposal: encoding/json: Opt-in for true streaming supp

#21823 243083df encoding/xml: very low performance in xml parser

Github的Web服务接口https://developer.github.com/v3/ 包含了更多的特性。

**练习4.10：**修改issues程序，根据问题的时间进行分类，比如不到一个月的、不到一年的、超过一年。

**练习4.11：**编写一个工具，允许用户在命令行创建、读取、更新和关闭Github上的issue，当必要的时候自动打开用户默认的编辑器用于输入文本信息。

**练习4.12：**流行的web漫画服务xkcd也提供了JSON接口。例如，一个<https://xkcd.com/571/info.0.json请求将返回一个很多人喜爱的571>编号的详细描述。下载每个链接（只下载一次）然后创建一个离线索引。编写一个xkcd工具，使用这些离线索引，打印和命令行输入的检索词相匹配的漫画的URL。

**练习4.13：**使用开发电影数据库的JSON服务接口，允许你检索和下载<https://omdbapi.com/>上电影的名字和对应的海报图像。编写一个poster工具，通过命令行输入的电影名字，下载对应的海报。

### 4.6 文本和HTML模板

前面的例子，只是最简单的格式化，使用Printf是完全足够的。但是有时候会需要复杂的打印格式，这时候一般需要将格式化代码分离出来以便更安全地修改。这些功能是由text/template和html/template等模板包提供的，它们提供了一个将变量值填充到一个文本或HTML格式的模板的机制。

一个模板是一个字符串或一个文件，里面包含了一个或多个由双花括号包含的{{action}}对象。大部分的字符串只是按字面值打印，但是对于actions部分将触发其他的行为。每个actions都包含了一个用模板语言书写的表达式，一个action虽然简短但是可以输出复杂的打印值，模板语言包含通过选择结构体的成员、调用函数或方法、表达式控制流if-else语句和range循环语句，还有其他实例化模板等诸多特性。下面是一个简单的模板字符串：

gopl.io/ch4/issuesreport

const templ = `{{.TotalCount}} issues:

{{range .Items}}------------------------------------

Number: {{.Number}}

User: {{.User.Login}}

Title: {{.Title | printf "%.64s"}}

Age: {{.CreatedAt | daysAgo}} days

{{end}}`

这个模板先打印匹配到的issue总数，然后打印每个issue的编号、创建用户、标题还有存在的时间。对于每个action，都有一个当前值的概念，对应点操作符，写作”.”。当前值”.”最初被初始化为调用模板时的参数，在当前例子中对应github.IssuesSearchResult类型的变量。模板中{{.TotalCount}}对应action将展开为结构体中TotalCount成员以默认的方式打印的值。模板中{{range .Items}}和{{end}}对应一个循环action，因此它们直接的内容可能会被展开多次，循环每次迭代的当前值对应当前的Items元素的值。

在一个action中，|操作符表示将前一个表达式的结果作为后一个函数的输入，类似于UNIX中管道的概念。在Title这一行的action中，第二个操作是一个printf函数，是一个基于fmt.Sprintf实现的内置函数，所有模板都可以直接使用。对于Age部分，第二个动作是一个叫daysAgo的函数，通过time.Since函数将CreatedAt成员转换为过去的时间长度：

func daysAgo(t time.Time) int {

return int(time.Since(t).Hours() / 24)

}

需要注意的是CreatedAt的参数类型是time.Time，并不是字符串。以同样的方式，我们可以通过定义一些方法来控制字符串的格式化（§2.5），一个类型同样可以定制自己的JSON编码和解码行为。time.Time类型对应的JSON值是一个标准时间格式的字符串。

生成模板的输出需要两个处理步骤。第一步是要分析模板并转为内部表示，然后基于指定的输入执行模板。分析模板部分一般只需要执行一次。下面的代码创建并分析上面定义的模板templ。注意方法调用链的顺序：template.New先创建并返回一个模板；Funcs方法将daysAgo等自定义函数注册到模板中，并返回模板；最后调用Parse函数分析模板。

report, err := template.New("report").

Funcs(template.FuncMap{"daysAgo": daysAgo}).

Parse(templ)

if err != nil {

log.Fatal(err)

}

因为模板通常在编译时就测试好了，如果模板解析失败将是一个致命的错误。template.Must辅助函数可以简化这个致命错误的处理：它接收一个模板和一个error类型的参数，检测error是否为nil（如果不是nil则发出panic异常），然后返回传入的模板。我们将在5.9节再讨论这个话题。

一旦模板已经创建、注册了daysAgo函数、并通过分析和检测，我们就可以使用github.IssuesSearchResult作为输入源、os.Stdout作为输出源来执行模板：

var report = template.Must(template.New("issuelist").

Funcs(template.FuncMap{"daysAgo": daysAgo}).

Parse(templ))

func main() {

result, err := github.SearchIssues(os.Args[1:])

if err != nil {

log.Fatal(err)

}

if err := report.Execute(os.Stdout, result); err != nil {

log.Fatal(err)

}

}

程序输出一个纯文本报告：

E:\GoProjects\src\mygopl\ch04\issuesreport>go run main.go repo:golang/go

is:open json decoder

50 issues:

------------------------------------

Number: 33416

User: bserdar

Title: encoding/json: This CL adds Decoder.InternKeys

Age: 354 days

------------------------------------

Number: 40128

User: rogpeppe

Title: proposal: encoding/json: garbage-free reading of tokens

Age: 12 days

------------------------------------

Number: 40127

User: rogpeppe

Title: proposal: encoding/json: add Encoder.EncodeToken method

Age: 12 days

------------------------------------

…

现在让我们转到html/template模板包。它使用和text/template包相同的API和模板语言，但是增加了一个将字符串自动转义特性，这可以避免输入字符串和HTML、JavaScript、CSS或URL语法产生冲突的问题。这个特性还可以避免一些长期存在的安全隐患，比如通过生成HTML注入攻击，通过构造一个含有恶意代码的问题标题，这些都可能让模板输出错误的输出，从而让他们控制页面。

下面的模板以HTML格式输出issue列表。注意import语句的不同：

gopl.io/ch4/issueshtml

var issueList = template.Must(template.New("issuelist").Parse(`

<h1>{{.TotalCount}} issues</h1>

<table>

<tr style='text-align: left'>

<th>#</th>

<th>State</th>

<th>User</th>

<th>Title</th>

</tr>

{{range .Items}}

<tr>

<td><a href='{{.HTMLURL}}'>{{.Number}}</a></td>

<td>{{.State}}</td>

<td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>

<td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>

</tr>

{{end}}

</table>

`))

下面的命令将在新的模板上执行一个稍微不同的查询：

go run main.go repo:golang/go commenter:gopherbot json encoder > issues.html

图4.4显示了在web浏览器中的效果图。每个issue包含到Github对应页面的链接。

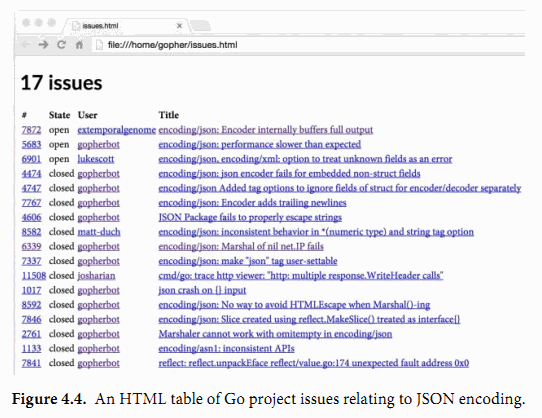
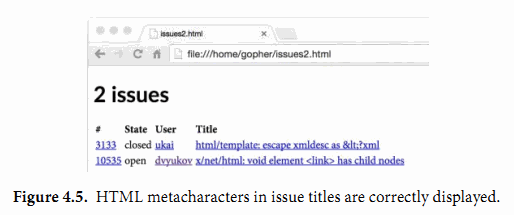


图4.4中issue没有包含会对HTML格式产生冲突的特殊字符，但是我们马上将看到标题中含有&和<字符的issue。下面的命令选择了两个这样的issue：

go run main.go repo:golang/go 3133 10535 > issues2.html

图4.5显示了该查询的结果。注意，html/template包已经自动将特殊字符转义，因此我们依然可以看到正确的字面值。如果我们使用text/template包的话，这2个issue将会产生错误，其中“&lt;”四个字符将会被当作小于字符“<”处理，同时“<link>”字符串将会被当作一个链接元素处理，它们都会导致HTML文档结构的改变，从而导致有未知的风险。

我们也可以通过对信任的HTML字符串使用template.HTML类型来抑制这种自动转义的行为。还有很多采用类型命名的字符串类型分别对应信任的JavaScript、CSS、和URL。下面的程序演示了两个使用不同类型的相同字符串产生的不同结果：A是一个普通字符串，B是一个信任的template.HTML字符串类型。



gopl.io/ch4/autoescape

func main() {

const templ = `<p>A: {{.A}}</p><p>B: {{.B}}</p>`

t := template.Must(template.New("escape").Parse(templ))

var data struct {

A string // untrusted plain text

B template.HTML // trusted HTML

}

data.A = "<b>Hello!</b>"

data.B = "<b>Hello!</b>"

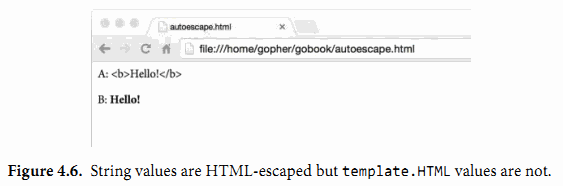
if err := t.Execute(os.Stdout, data); err != nil {

log.Fatal(err)

}

}

图4.6显示了出现在浏览器中的模板输出。我们看到A的黑色标记被转义失效了，但是B没有。



我们这里只讲述了模板系统中最基本的特性。一如既往，如果想了解更多的信息，请自己查看包文档：

go doc text/template

go doc html/template

**练习4.14：**创建一个web服务器，查询一次Github，然后生成BUG报告、里程碑和对应的用户信息。

## 函数

函数可以让我们将一个语句序列打包为一个单元，然后可以从程序中其他地方多次调用。函数的机制可以让我们将一个大的工作分解为小的任务，这样的小任务可以让不同程序员在不同时间、不同地方独立完成。一个函数同时对用户隐藏了其实现细节。由于这些因素，对于任何编程语言来说，函数都是一个至关重要的部分。

我们已经见过许多函数了。现在，让我们多花一点时间来彻底地讨论函数特性。本章的运行示例是一个网络蜘蛛，也就是web搜索引擎中负责抓取网页部分的组件，它们根据抓取网页中的链接继续抓取链接指向的页面。一个网络蜘蛛的例子给我们足够的机会去探索递归函数、匿名函数、错误处理和函数其它的很多特性。

### 5.1 函数声明

函数声明包括函数名、形式参数列表、返回值列表（可省略）以及函数体。

func name(parameter-list) (result-list) {

body

}

形式参数列表描述了函数的参数名以及参数类型。这些参数作为局部变量，其值由参数调用者提供。返回值列表描述了函数返回值的变量名以及类型。如果函数返回一个无名变量或者没有返回值，返回值列表的括号是可以省略的。如果一个函数声明不包括返回值列表，那么函数体执行完毕后，不会产生返回值。在hypot函数中，

func hypot(x, y float64) float64{

return math.Sqrt(x\*x + y\*y)

}

fmt.Println(hypot(3, 4)) // “5”

x和y是形参名，3和4是调用时的传入的实参，函数返回了一个float64类型的值。返回值也可以像形式参数一样被命名。在这种情况下，每个返回值被声明成一个局部变量，并根据该返回值的类型，将其初始化为0.如果一个函数在声明时，包含返回值列表，该函数必须以return语句结尾，除非函数明显无法运行到结尾处。例如函数在结尾时调用了panic异常或函数中存在无限循环。

正如hypot一样，如果一组形参或返回值有相同的类型，我们不必为每个形参都写出参数类型。下面2个声明时等价的：

func f(i, j, k int, s, t string) {/\*…\*/}

func f(i int, j int, k int, s string, t strng) {/\*…\*/}

下面，我们给出4种方法声明拥有2个int型参数和1个int型返回值的函数blank identifier（译注：即下文的\_符合）可以强调某个参数未被使用。

func add(x int, y int) int {

return x + y

}

func sub(x, y int) (z int) {

z = x - y

return

}

func first(x int, \_ int) int {

return x

}

func zero(int, int) int {

return 0

}

fmt.Printf("%T\n", add) // "func(int, int) int"

fmt.Printf("%T\n", sub) // "func(int, int) int"

fmt.Printf("%T\n", first) // "func(int, int) int"

fmt.Printf("%T\n", zero) // "func(int, int) int"

函数的类型被称为函数的标识符，如果两个函数形式参数列表和返回值列表中的变量类型一一对应，那么这两个函数被认为有相同的类型或标识符。形参和返回值的变量名不影响函数标识符，也不影响它们是否可以以省略参数类型的形式表示。

每一次函数调用都必须按照声明顺序为所有参数提供实参（参数值）。在函数调用中，Go语言没有默认参数值，也没有任何方法可以通过参数名指定形参，因此形参和返回值的变量名对于函数调用者而言没有意义。

在函数体中，函数的形参作为局部变量，被初始化为调用者提供的值。函数的形参和有名返回值作为函数最外层的局部变量，被存储在相同的词法块中。

实参通过值的方式传递，因此函数的形参是实参的拷贝。对形参进行修改不会影响实参。但是，如果实参包括引用类型，如指针，slice（切片）、map、function、channel等类型，实参可能会由于函数的间接引用被修改。

你可能会偶尔遇到没有函数体的函数声明，这表示该函数不是以Go实现的。这样的声明定义了函数标识符。

package math

func Sin(x float64) float // implemented in assembly language

### 5.2 递归

函数可以是递归的，这意味着函数可以直接或间接地调用自身。对许多问题而言，递归是一种强有力的技术，例如处理递归的数据结构。在4.4节，我们通过遍历二叉树来实现简单的插入排序，在本章节，我们再次使用它来处理HTML文件。

下文的示例代码使用了非标准包golang.org/x/net/html，解析HTML。golang.org/x/…目录下存储了一些由Go团队设计、维护，怼网络编程、国际化文件处理、移动平台、图像处理、加密解密、开发者工具提供支持的扩展包。未将这些扩展包加入到标准库原因有二，一是部分包仍在开发中，二是对大多数Go语言的开发者而言，扩展包提供的功能很少被使用。

例子中使用golang.org/x/net/html的部分api如下所示。html.Parse函数读入一组bytes解析后，返回html.Node类型的HTML页面树状结构根节点。HTML拥有很多类型的节点如text（文本），comments（注释）类型，在下面的例子中，我们只关注<name key=’value’>形式的节点。

golang/org/x/net/html

package html

type Node struct {

Type NodeType

Data string

Attr []Attribute

FirstChild, NextSibling \*Node

}

type NodeType int32

const (

ErrorNode NodeType = iota

TextNode

DocumentNode

ElementNode

CommentNode

DoctypeNode

)

type Attribute struct {

Key, Val string

}

func Parse(r io.Reader) (\*Node, error)

main函数解析HTML标准输入，通过递归函数visit获得links（链接），并打印出这些links：

gopl.io/ch5/findlinks1

// Findlinks1 prints the links in an HTML document read from standard input.

package main

import (

"fmt"

"golang.org/x/net/html"

"os"

)

func main() {

doc, err := html.Parse(os.Stdin)

if err != nil {

fmt.Fprintf(os.Stderr, "findlinks1: %v\n", err)

os.Exit(1)

}

for \_, link := range visit(nil, doc) {

fmt.Println(link)

}

}

visit函数遍历HTML的节点树，从每一个anchor元素的href属性获得link，将这些links存入字符串数组中，并返回这个字符串数组。

// visit appends to links each link found in n and returns the result.

func visit(links []string, n \*html.Node) []string {

if n.Type == html.ElementNode && n.Data == "a" {

for \_, a := range n.Attr {

if a.Key == "href" {

links = append(links, a.Val)

}

}

}

for c := n.FirstChild; c != nil; c = c.NextSibling {

links = visit(links, c)

}

return links

}

为了遍历结点n的所有后台结点，每次遇到n的孩子结点时，visit递归的调用自身。这些孩子结点存放在FirstChild链表中。

让我们以Go的主页（golang.org）作为目标，运行findlinks。我们以fetch（1.5章）的输出作为findlinks的输入。下面的输出做了简化处理。

$ go build gopl.io/ch1/fetch

$ go build gopl.io/ch5/findlinks1

$ ./fetch https://golang.org | ./findlinks1

#

/doc/

/pkg/

/help/

/blog/

http://play.golang.org/

//tour.golang.org/

https://golang.org/dl/

//blog.golang.org/

/LICENSE

/doc/tos.html

<http://www.google.com/intl/en/policies/privacy/>

注意在页面中出现的链接格式，在之后我们会介绍如何将这些链接，根据跟路径（https://golang.org）生成可以直接访问的url。

在函数outline中，我们通过递归的方式遍历整个HTML节点树，并输出树的结构。在outline内部，每遇到一个HTML元素标签，就将其入栈，并输出。

gopl.io/ch5/outline

func main() {

doc, err := html.Parse(os.Stdin)

if err != nil {

fmt.Fprintf(os.Stderr, "outline: %v\n", err)

os.Exit(1)

}

outline(nil, doc)

}

func outline(stack []string, n \*html.Node) {

if n.Type == html.ElementNode {

stack = append(stack, n.Data) // push tag

fmt.Println(stack)

}

for c := n.FirstChild; c != nil; c = c.NextSibling {

outline(stack, c)

}

}

有一点值得注意：outline有入栈操作，但没有相对应的出栈操作。当outline调用自身时，被调用者接收的是stack的拷贝。被调用者对stack的元素追加操作，修改的是stack的拷贝，其可能会修改slice底层的数组甚至是申请一块新的内存空间进行扩容；但这个过程并不会修改调用方的stack。因此当函数返回时，调用方的stack与其调用自身之前完全一致。

下面是https//golang.org页面的简要结构：

$ go build gopl.io/ch5/outline

$ ./fetch https://golang.org | ./outline

[html]

[html head]

[html head meta]

[html head title]

[html head link]

[html body]

[html body div]

[html body div]

[html body div div]

[html body div div form]

[html body div div form div]

[html body div div form div a]

...

正如你在上面实验中所见，大部分HTML页面只需几层递归就能被处理，但仍然有些页面需要深层次的递归。

大部分编程语言使用固定大小的函数调用栈，常见的大小从64KB到2MB不等。固定大小栈会限制递归的深度，当你用递归处理大量数据时，需要避免栈溢出；除此之外，还会导致安全性问题。与此相反，Go语言使用可变栈，栈的大小按需增加（初始时很小）。这使得我们使用递归时不必考虑溢出和安全问题。

**练习5.1：**修改findlinks代码中比那里n.FirstChild链表的部分，将循环调用visit，改成递归调用。

**练习5.2：**编写函数，记录在HTML树中出现的同名元素的次数。

**练习5.3：**编写函数输出所有的text结点的内容。注意不要访问<script>和<style>元素，因为这些元素对浏览者是不可见的。

**练习5.4：**扩展visit函数，使其能够处理其他类型的结点，如images、scripts和style sheets。

### 5.3 多返回值

在Go中，一个函数可以返回多个值。我们已经在之前例子中看到，许多标准库中的函数返回2个值，一个是期望得到的返回值，另一个是函数出错时的错误信息。下面的例子会展示如何编写多返回值的函数。

下面的程序是findlinks的改进版本。修改后的findlinks可以自己发起HTTP请求，这样我们就不必再运行fetch。因为HTTP请求和解析操作可能会失败，因此findlinks声明了2个返回值：链接列表和错误信息。一般而言，HTML的解析器可以处理HTML页面的错误节点，构造出HTML页面结构，所以解析HTML很少失败。这意味着如果findlinks函数失败了，很可能是由于I/O的错误导致的。

gopl.io/ch5/findlinks2

func main() {

for \_, url := range os.Args[1:] {

links, err := findLinks(url)

if err != nil {

fmt.Fprintf(os.Stderr, "findlinks2: %v\n", err)

continue

}

for \_, link := range links {

fmt.Println(link)

}

}

}

// findLinks performs an HTTP GET request for url, parses the

// response as HTML, and extracts and returns the links

func findLinks(url string) ([]string, error) {

resp, err := http.Get(url)

if err != nil {

return nil, err

}

if resp.StatusCode != http.StatusOK {

resp.Body.Close()

return nil, fmt.Errorf("geeeing %s: %s", url, resp.Status)

}

doc, err := html.Parse(resp.Body)

resp.Body.Close()

if err != nil {

return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)

}

return visit(nil, doc), nil

}

在findlinks中，有4处return语句，每一处return都返回了一组值。前三出return，将http和html包中的错误信息传递给findlinks的调用者。第一处return直接返回错误信息，其他两处通过fmt.Errorf（§7.8）输出详细的错误信息。如果findlinks成功结束，最后的return语句将一组解析获得的连接返回给用户。

在findlinks中，我们必须确保resp.Body被关闭，释放网络资源。虽然Go的垃圾回收机制会回收不被使用的内存，但是这不包括操作系统层面的资源，比如打开的文件、网络连接。因此我们必须显式的释放这些资源。

调用者返回值函数时，返回给调用者的是一组值，调用者必须显式的将这些值分配给变量：

links, err := findLinks(url)

如果某个值不被使用，可以将其分配给blank identifier：

links, \_ := findLinks(url) // errors ignored

一个函数内部可以将另一个有多返回值的函数调用作为返回值，下面的例子展示了与findLinks有相同功能的函数，两者的区别在于下面的例子先输出参数：

func findLinksLog(url string) ([]string, error) {

log.Printf("findLinks %s", url)

return findLinks(url)

}

当你嗲用接受多参数的函数时，可以讲一个返回多参数的函数调用作为该函数的参数。虽然这很少出现在实际生产代码中，但这个特性在debug时很方便，我们只需要一条语句就可以输出所有的返回值。下面的代码是等价的：

log.Println(findLinks(url))

links, err := findLinks(url)

log.Println(links, err)

准确的变量名可以传达函数返回值的含义。尤其在返回值的类型都相同时，就像下面这样：

func Size(rect image.Rectangle) (width, height int)

func Split(path string) (dir, file string)

func HourMinSec(t time.Time) (hour, minute, second int)

虽然良好的命名很重要，但你也不必为每一个返回值都取一个适当的名字。比如，按照惯例，函数的最后一个bool类型的返回值表示函数是否运行成功，error类型的返回值代表函数的错误信息，对于这些类似的惯例，我们不必思考合适的命名，它们都无需解释。

如果一个函数所有的返回值都有显式的变量名，那么该函数的return语句可以省略操作数。这称之为bare return。

// CountWordsAndImages does an HTTP GET request for the HTML

// document url and returns the number of words and images in it.

func CountWordsAndImages(url string) (words, images int, err error) {

resp, err := http.Get(url)

if err != nil {

return

}

doc, err := html.Parse(resp.Body)

resp.Body.Close()

if err != nil {

err = fmt.Errorf("parsing HTML: %s", err)

return

}

words, images = countWordsAndImages(doc)

return

}

func countWordsAndImages(n \*html.Node) (words, images int) {

/\* ... \*/

}

按照返回值列表的次序，返回所有的返回值，在上面的例子中，每一个return语句等价于：

return words, images, err

当一个函数有多处return语句以及许多返回值时，bare return可以减少代码的重复，但是使得代码难以被理解。举个例子，如果你没有仔细地审查代码，很难发现前2处return等价于return 0,0,err（Go会将返回值words和images在函数体的开始处，根据它们的类型，将其初始化为0），最后一处return等价于return words, image, nil。基于以上原因，不宜过度使用bare return。

**练习5.5：**实现countWordsAndImages。（参考练习4.9如何分词）

**练习5.6：**修改gopl.io/ch3/surface（§3.2）中的corner函数，将返回值命名，并使用bare return。

### 5.4 错误

在Go中有一部分函数总是能成功的运行。比如strings.Contains和strconv.FormatBool函数，对各种可能的输入都做了良好的处理，使得运行时几乎不会失败，除非遇到灾难性的、不可预料的情况，比如运行时的内存溢出。导致这种错误的原因很复杂，难以处理，从错误中恢复的可能性也很低。

还有一部分函数只要输入的参数满足一定条件，也能保证运行成功。比如time.Date函数，该函数将年月日等参数构造成time.Time对象，除非最后一个参数（时区）是nil。这种情况下会引发panic异常。panic是来自被调用函数的信号，表示发生了某个已知的bug。一个良好的程序永远不应该发生panic异常。

对于大部分函数而言，永远无法确保能否成功运行。这是因为错误的原因超出了程序员的控制。举个例子，任何进行I/O操作的函数都会面临出现错误的可能，只有没有经验的程序员才会相信读写操作不会失败，即使是简单的读写。因此，当本该可信的操作出乎意料的失败后，我们必须弄清楚导致失败的原因。

在Go的错误处理中，错误是软件包API和应用程序用户界面的一个重要组成部分，程序运行失败仅被认为是几个预期的结果之一。

对于那些将运行失败看作是预期结果的函数，它们会返回一个额外的返回值，通常是最后一个，来传递错误信息。如果导致失败的原因只有一个，额外的返回值可以是一个布尔值，通常被命名为ok。比如，cache.Loopup失败的唯一原因是key不存在，那么代码可以按照下面的方式组织：

value, ok := cache.Loopup(key)

if !ok {

// …cache[key] dose not exist…

}

通常，导致失败的原因不止一种，尤其是对I/O操作而言，用户需要了解更多的错误信息。因此，额外的返回值不再是简单的布尔类型，而是error类型。

内置的error是接口类型。我们将在第七章了解接口类型的含义，以及它对错误处理的影响。现在我们只需要明白error类型可能是nil或者non-nil。nil意味着函数运行成功，non-nil表示失败。对于non-nil的error类型，我们可以通过调用error的Error函数或者输出函数获得字符串类型的错误信息。

fmt.Println(err)

fmt.Printf(“%v”, err)

通常，当函数返回non-nil的error时，其他的返回值是未定义的（undefined），这些未定义的返回值应该被忽略。然后，有少部分函数在发生错误时，仍然会返回一些有用的返回值。比如，当读取文件发生错误时，Read函数会返回可以读取的字节数以及错误信息。对于这种情况，正确的处理方式应该是先处理这些不完整的数据，再处理错误。因此对函数的返回值要有清晰的说明，以便于其他人使用。

在Go中，函数运行失败时会返回错误信息，这些错误信息被认为是一种预期的值而非异常（exception），这使得Go有别于那些将函数运行失败看作是异常的语言。虽然Go有各种异常机制，但这些机制仅被使用在处理那些未被预料到的错误，即bug，而不是那些崽健壮程序中应该被避免的程序错误。对于Go的异常机制我们将在5.9介绍。

Go这样设计的原因是由于对于某个应该在控制流程中处理的错误而言，将这个错误以异常的形式抛出会混乱对错误的描述，这通常会导致一些糟糕的后果。当某个程序错误被当做异常处理后，这个错误会将堆栈跟踪信息返回给终端用户，这些信息复杂且无用，无法帮助定位错误。

正因此，Go会用控制流机制（如if和return）处理错误，这使得编码人员能更多的关注错误处理。

#### 5.4.1 错误处理策略

当一次函数调用返回错误时，调用者应该选择合适的方式处理错误。根据情况的不同，有很多处理方式，让我们来看看常用的五种方式。

首先，也是最常用的方式是传播错误。这意味着函数中某个子程序的失败，会变成该函数的失败。下面，我们以5.3节的findLinks函数作为例子。如果findLinks对http.Get的调用失败，findLinks会直接将这个HTTP错误返回给调用者：

resp, err := http.Get(url)

if err != nil {

return nil, err

}

当对html.Parse的调用失败时，findLinks不会直接返回html.Parse的错误，因为缺少两条重要信息：1、发生错误时的解析器（html parser）；2、发生错误的url。因此，findLinks构造了一个新的错误信息，既包含了这两项，也包括了底层的解析出错的信息。

doc, err := html.Parse(resp.Body)

resp.Body.Close()

if err != nil {

return nil, fmt.Errorf(“parsing %s as HTML: %v”, url, err)

}

fmt.Errorf函数使用fmt.Sprintf格式化错误信息并返回。我们使用该函数添加额外的前缀上下文信息到原始错误信息。当错误最终由main函数处理时，错误信息应提供清晰的从原因到后果的因果链，就像美国宇航局事故调查时做的那样：

genesis: crashed: no parachute: G-switch failed: bad relay orientation

由于错误信息经常是以链式组合在一起的，所以错误信息中应避免大写和换行符。最终的错误信息可能很长，我们可以通过类似grep的工具处理错误信息（译注：grep是一种文本搜索工具）。

编写错误信息时，我们要确保信息对问题细节的描述是详尽的。尤其是要注意错误信息表达的一致性，即相同的函数或同包内的同一组函数返回的错误在构成和处理方式上是相似的。

以os包为例，os包确保文件操作（如os.Open、Read、Write、Close）返回的每个错误的描述不仅仅包含错误的原因（如无权限，文件目录不存在）也包含文件名，这样调用者在构造新的错误信息时无需再添加这些信息。

一般而言，被调用函数f(x)会将调用信息和参数信息作为发生错误时的上下文放在错误信息中并返回给调用者，调用者需要添加一些错误信息中不包含的信息，比如添加url到html.Parse返回的错误中。

让我们来看看处理策略的第二种策略。如果错误的发生是偶然的，或由不可预知的问题导致的。一个明智的选择是重新尝试失败的操作。再重试时，我们需要限制重试的时间间隔或重试的次数，防止无限制的重试。

gopl.io/ch5/wait

// WaitForServer attempts to contact the server of a URL.

// It tries for one minute using eponential back-off.

// It reports an error if all attempts fail.

func WaitForServer(url string) error {

const timeout = 1 \* time.Minute

deadline := time.Now().Add(timeout)

for tries := 0; time.Now().Before(deadline); tries++ {

\_, err := http.Head(url)

if err == nil {

return nil // success

}

log.Printf("server not responding (%s); retrying...", err)

time.Sleep(time.Second << uint(tries)) // exponential back-off

}

return fmt.Errorf("server %s failed to respond after %s", url, timeout)

}

如果错误发生后，程序无法继续运行，我们就可以采用第三种策略：输出错误信息并结束程序。需要注意的是，这种策略只应在main中执行。对库函数而言，应仅向上传播错误

// WaitForServer attempts to contact the server of a URL.

// It tries for one minute using eponential back-off.

// It reports an error if all attempts fail.

func WaitForServer(url string) error {

const timeout = 1 \* time.Minute

deadline := time.Now().Add(timeout)

for tries := 0; time.Now().Before(deadline); tries++ {

\_, err := http.Head(url)

if err == nil {

return nil // success

}

log.Printf("server not responding (%s); retrying...", err)

time.Sleep(time.Second << uint(tries)) // exponential back-off

}

return fmt.Errorf("server %s failed to respond after %s", url, timeout)

}

如果错误发生后，程序无法继续运行，我们就可以采用第三种策略：输出错误信息并结束程序。需要注意的是，这种策略只应在main中执行。对库函数而言，应仅向上传播错误，除非该错误意味着程序内部包含不一致性，即遇到了bug，才能在库函数中结束程序。

// (In function main.)

if err := WaitForServer(url); err != nil {

fmt.Fprintf(os.Stderr, "Site is down: %v\n",err)

os.Exit(1)

}

调用log.Fatalf可以更简洁的代码达到与上文相同的效果。log中的所有函数，都默认会在错误信息之前输出时间信息。

if err := WaitForServer(url); err != nil {

log.Fatalf(“Site is down: %v\n”, err)

}

长时间运行的服务器常采用默认的时间格式，而交互式工具很少采用包含如此多的信息的格式。

2006/01/02 15:04:05 Site is down: no such domain:

bad.gopl.o

我们可以设置log的前缀信息屏蔽时间信息，一般而言，前缀信息会被设置成命令名。

log.SetPrefix(“wait: “)

log.SetFlags(0)

第四种策略：有时，我们只需要输出错误信息就足够了，不需要中断程序的运行。我们可以通过log包提供函数

if err := Ping(); err != nil {

log.Printf(“ping failed: %v; networking disabled”, err)

}

或者标准错误流输出错误信息。

if err := Ping(); err != nil {

fmt.Fprintf(os.Stderr, “ping failed: %v; networking disabled\n”, err)

}

log包中的所有函数会为没有换行符的字符串增加换行符。

第五种，也是最后一种策略：我们可以直接忽略掉错误。

dir, err := ioutil.TempDir(“”, “scratch”)

if err != nil {

return fmt.Errorf(“failed to create temp dir: %v”, err)

}

// …use temp dir…

os.RemoveAll(dir) // ignore errors; $TMPDIR is cleaned periodically

尽管os.RemoveAll会失败，但上面的例子并没有做错误处理。这是因为操作系统会定期的清理临时目录。正因如此，虽然程序没有处理错误，但程序的逻辑不会因此受到影响。我们应该在每次函数调用后，都养成考虑错误处理的习惯，当你决定忽略某个错误时，你应该清晰地写下你的意图。

在Go中，错误处理有一套独特的编码风格。检查某个子函数是否失败后，我们通常将处理失败的逻辑代码放在处理成功的代码之前。如果某个错误会导致函数返回，那么成功时的逻辑代码不应放在else语句块中，而应直接放在函数体中。Go中大部分函数的代码结构几乎相同，首先是一系列的初始检查，防止错误发生，之后是函数的实际逻辑。

#### 5.4.2 文件结尾错误（EOF）

函数经常会返回多种错误，这对终端用户来说可能会很有趣，但对程序而言，这使得情况变得复杂。很多时候，程序必须根据错误类型，作出不同的响应。让我们考虑这样一个例子：从文件中读取n个字节。如果n等于文件的长度，读取过程中的任何错误都表示失败。如果n小于文件的长度，调用者会重复的读取固定大小的数据直到文件结束。这会导致调用者必须分别处理由文件结束引起的各种错误。基于这样的原因，io包保证任何由文件结束引起的读取失败都返回同一个错误——io.EOF，该错误在io包中定义：

package io

import “errors”

// EOF is the error returned by Read when no more input s available.

var EOF = errors.New(“EOF”)

调用者只需通过简单地比较，就可以检测出这个错误。下面的例子展示了如何从标准输入中读取字符，以及判断文件结束。（4.3的chartcount程序展示了更加复杂的代码）

in := bufio.NewReader(os.Stdin)

for {

r, \_, err := in.ReadRune()

if err == io.EOF {

break // finished reading

}

if err != nil {

return fmt.Errorf(“read failed:%v”, err)

}

// … use r…

}

因为文件结束这种错误不需要更多的描述，所以io.EOF有固定的错误信息——“EOF”。对于其他错误，我们可能需要在错误信息中描述错误的类型和数量，这使得我们不能像io.EOF一样采用固定的错误信息。在7.11节中，我们会提出更系统的方法区分某些固定的错误值。

### 5.5 函数值

在Go中，函数被看作第一类值（first-class values）：函数像其他值一样，拥有类型，可以被赋值给其他变量，传递给函数，从函数返回。对函数值（function value）的调用类似函数调用。例子如下：

func square(n int) int { return n \* n }

func negative(n int) int { return -n }

func product(m, n int) int { return m \* n }

f := square

fmt.Println(f(3)) // "9"

f = negative

fmt.Println(f(3)) // "-3"

fmt.Printf("%T\n", f) // "func(int) int"

f = product // compile error: can't assign func(int, int) int to func(int) int

函数类型的零值是nil。调用值为nil的函数值为引起panic错误：

var f func(int) int

f(3) // 此处f的值为nil，会引起panic错误

函数值可以与nil比较：

var f func(int) int

if f != nil {

f(3)

}

但是函数值之间是不可比较的，也不能用函数值作为map的key。

函数值使得我们不仅仅可以通过数据来修改参数化函数，亦可通过行为。标准库中包含许多这样的例子。下面的代码展示了如何使用这个技巧。strings.Map对字符串中的每个字符调用add1函数的返回值组成一个新的字符串返回给调用者。

func add1(r rune) rune {return r + 1}

fmt.Println(strings.Map(add1, "HAL-9000")) // "IBM.:111"

fmt.Println(strings.Map(add1, "VMS")) // "WNT"

fmt.Println(strings.Map(add1, "Admix")) // "Benjy"

5.2节的findLinks函数使用了辅助函数visit，遍历和操作了HTML页面的所有结点。使用函数值，我们可以将遍历结点的逻辑和操作结点的逻辑分离，使得我们可以复用遍历的逻辑，从而对结点进行不同的操作。

gopl.io/ch5/outline2

// forEachNode针对每个结点x，都会调用pre(x)和post(x)。

// pre和post都是可选的。

// 遍历孩子结点之前，pre被调用

// 遍历孩子结点之后，post被调用

func forEachNode(n \*html.Node, pre, post func(n \*html.Node)) {

if pre != nil {

pre(n)

}

for c := n.FirstChild; c != nil; c = c.NextSibling {

forEachNode(c, pre, post)

}

if post != nil {

post(n)

}

}

该函数接收2个函数作为参数，分别在结点的孩子被访问前和访问后调用。这样的设计给调用者更大的灵活性。举个例子，现在我们有startElement和endElement两个函数用于输出HTML元素的开始标签和结束标签<b>…</b>：

var depth int

func startElement(n \*html.Node) {

if n.Type == html.ElementNode {

fmt.Printf("%\*s<%s>\n", depth\*2, "", n.Data)

depth++

}

}

func endElement(n \*html.Node) {

if n.Type == html.ElementNode {

depth--

fmt.Printf("%\*s</%s>\n", depth\*2, "", n.Data)

}

}

上面的代码利用fmt.Printf的一个小技巧控制输出的缩进。%\*s中的\*会在字符串之前填充一些空格。在例子中，每次输出会先填充depth\*2数量的空格，再输出””， 最后再输出HTML标签。

如果我们像下面这样调用forEachNode：

forEachNode(doc, startElement, endElement)

与之前的outline程序相比，我们得到了更加详细的页面结构：

$ go build gopl.io/ch5/outline2

$ ./outline2 http://gopl.io

<html>

<head>

<meta>

</meta>

<title>

</title>

<style>

</style>

</head>

<body>

<table>

<tbody>

<tr>

<td>

<a>

<img>

</img>

...

**练习5.7：**完善startElement和endElement函数，使其成为通用的HTML输出器。要求：输出注释结点，文本节点以及每个元素的属性（<a href=’…’>）。使用简略格式输出没有孩子结点的元素（即用<img/>代替<img></img>）。编写测试，验证程序输出的格式正确。（详见11章）

**练习5.8：**修改pre和post函数，使其返回布尔类型的返回值。返回false时，中止forEachNode的遍历。使用修改后的代码编写ElementByID函数，根据用户输入的id查找第一个拥有该id元素的HTML元素，查找成功后，停止遍历。

func ElementByID(doc \*html.Node, id string) \*html.Node

**练习5.9：**编写函数expand，将s中的“foo”替换为f(“foo”)的返回值。

func expand(s string, f func(string) string) string

### 5.6 匿名函数

拥有函数名的函数只能在包级语法块中被声明，通过函数字面量（function literal），我们可绕过这一限制，在任何表达式中表示一个函数值。函数字面量的语法和函数声明相似，区别在于func关键字后没有函数名。函数值字面量是一种表达式，它的值被称为匿名函数（annoymous function）。

函数字面量允许我们在使用函数时，再定义它。通过这种技巧，我们可以改写之前对strings.Map的调用：

strings.Map(func(r rune) rune { return r + 1}, “HAL-9000”)

更为重要的是，通过这种方式定义的函数可以访问完整的词法环境（lexical environment），这意味着在函数中定义的内部函数可以引用该函数的变量，如下例所思：

gopl.io/ch5/squares

// squares返回一个匿名函数

// 该匿名函数每次被调用时都会返回下一个数的平方。

func squares() func() int {

var x int

return func() int {

x++

return x \* x

}

}

func main() {

f := squares()

fmt.Println(f()) // "1"

fmt.Println(f()) // "4"

fmt.Println(f()) // "9"

fmt.Println(f()) // "16"

}

函数squares返回另一个类型为func() int 的函数。对squares的一次调用会生成一个局部变量x并发挥一个匿名函数。每次调用匿名函数时，该函数都会先使x的值加1，再返回x的平方。第二次调用squares时，会生成第二个x变量，并返回一个新的匿名函数。新匿名函数操作的是第二个x变量。

squares的例子证明，函数值不仅仅是一串代码，还记录了状态。在squares中定义的匿名内部函数可以访问和更新squares中的局部变量，这意味着匿名函数和squares中，存在变量引用。这就是函数值属于引用类型和函数值不可比较的原因。Go使用闭包（closures）技术实现函数值，Go程序员也把函数值叫做闭包。

通过这个例子，我们看到变量的生命周期不由它的作用域决定：squares返回后，变量x仍然隐式的存在于f中。

接下来，我们讨论一下有点学术性的例子，考虑这样一个问题：给定一些计算机课程，每个课程都有前置课程，只有完成了前置课程才可以开始当前课程的学习；我们的目标是选择出一组课程，这组课程必须确保按顺序学习时，能全部被完成。每个课程的前置课程如下：

gopl.io/ch5/toposort

// prereqs记录了每个课程的前置课程

var prereqs = map[string][]string{

"algorithms": {"data structures"},

"calculus": {"linear algebra"},

"compilers": {

"data structures",

"formal languages",

"computer organization",

},

"data structures": {"discrete math"},

"databases": {"data structures"},

"discrete math": {"intro to programming"},

"formal languages": {"discrete math"},

"networks": {"operating systems"},

"operating systems": {"data structures", "computer organization"},

"programming languages": {"data structures", "computer organization"},

}

这类问题被称作拓扑排序。从概念上说，前置条件可以构成有向图。图中的顶点表示课程，边表示课程间的依赖关系。显然，图中应该无环，这也就是说从某点出发的边，最终不会回到该点。下面的代码用深度优先搜索了整张图，获得了符合要求的课程序列。

func main() {

for i, course := range topoSort(prereqs) {

fmt.Printf("%d:\t%s\n", i+1, course)

}

}

func topoSort(m map[string][]string) []string {

var order []string

seen := make(map[string]bool)

var visitAll func(items []string)

visitAll = func(items []string) {

for \_, item := range items {

if !seen[item] {

seen[item] = true

visitAll(m[item])

order = append(order, item)

}

}

}

var keys []string

for key := range m {

keys = append(keys, key)

}

sort.Strings(keys)

visitAll(keys)

return order

}

当匿名函数需要被递归调用时，我们必须首先声明一个变量（在上面的例子中，我们首先声明了visitAll），再将匿名函数赋值给这个变量。如果不分成两部，函数字面量无法与visitAll绑定，我们也无法递归调用该匿名函数。

visitAll := func(items []string) {

//…

visitAll(m[item] // compile error: undefined: visitAll

//…

}

在topsort中，首先对prereqs中的key排序，再调用visitAll。因为prereqs映射的是切片而不是更复杂的map，所以数据的遍历次序是固定的，这意味着你每次运行topsort得到的输出都是一样的。topsort的输出结果如下：

1: intro to programming

2: discrete math

3: data structures

4: algorithms

5: linear algebra

6: calculus

7: formal languages

8: computer organization

9: compilers

10: databases

11: operating systems

12: networks

13: programming languages

让我们回到findLinks这个例子。我们将代码移动到了links包下，将函数重命名为Extract，在第八章我们会再次用到这个函数。新的匿名函数被引入，用于替换原来的visit函数。该匿名函数负责将新连接添加到切片中。在Extract中，使用forEachNode遍历HTML页面，由于Extract只需要在遍历结点前操作结点，所以forEachNode的post参数被传入nil。

gopl.io/ch5/links

// Package links provides a link-extraction function.

package links

import (

"fmt"

"golang.org/x/net/html"

"net/http"

)

// Extract makes an HTTP GET request to the specified URL, parses

// the response as HTML, and returns the links in the HTML document.

func Extract(url string) ([]string, error) {

resp, err := http.Get(url)

if err != nil {

return nil, err

}

if resp.StatusCode != http.StatusOK {

resp.Body.Close()

return nil, fmt.Errorf("getting %s: %s", url, resp.Status)

}

doc, err := html.Parse(resp.Body)

resp.Body.Close()

if err != nil {

return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)

}

var links []string

visitNode := func(n \*html.Node) {

if n.Type == html.ElementNode && n.Data == "a" {

for \_, a := range n.Attr {

if a.Key != "href" {

continue

}

link, err := resp.Request.URL.Parse(a.Val)

if err != nil {

continue // ignore bad URLs

}

links = append(links, link.String())

}

}

}

forEachNode(doc, visitNode, nil)

return links, nil

}

上面的代码对之前的版本做了改进，现在links中存储的不是href属性的原始值，而是通过resp.Request.URL解析后的值。解析后，这些连接以绝对路径的形式存在，可以直接被http.Get访问。

网页抓取的核心问题就是如何遍历图。在topoSort的例子中，已经展示了深度优先遍历，在网页抓取中，我们会展示如何用广度优先遍历图。在第8章，我们会介绍如何将深度优先和广度优先结合使用。

下面的函数实现了广度优先算法。调用者需要输入一个初始的待访问列表和一个函数f。待访问列表中的每个元素被定义为string类型。广度优先算法会为每个元素调用一次f。每次f执行完毕后，会返回一组待访问元素。这些元素会被加入到待访问列表中。当待访问列表中的所有元素都被访问后，breadthFirst函数运行结束。为了避免同一个元素被访问两次，代码中维护了一个map。

gopl.io/ch5/findlinks3

// breadthFirst calls f for each item in the worklist.

// And items returned by f are added to the worklist.

// f is called at most once for each item.

func breadthFirst(f func(item string) []string, worklist []string) {

seen := make(map[string]bool)

for len(worklist) > 0 {

items := worklist

worklist = nil

for \_, item := range items {

if !seen[item] {

seen[item] = true

worklist = append(worklist, f(item)...)

}

}

}

}

就像我们在章节3解释的那样，append的参数“f(item)…”，会将f返回的一组元素一个个添加到worklist中。

在我们网页抓取器中，元素的类型是url。crawl函数会将URL输出，提取其中的新链接，并将这些新链接返回。我们会将crawl作为参数传递给breadthFirest。

func crawl(url string) []string {

fmt.Println(url)

list, err := links.Extract(url)

if err != nil {

log.Print(err)

}

return list

}

为了使抓取器开始运行，我们用命令行输入的参数作为初始的待访问url。

func main() {

// Crawl the web breadth-first.

// starting from the command-line arguments.

breadthFirst(crawl, os.Args[1:])

}

让我们从<https://golang.org>开始，下面是程序的输出结果：

$ go build gopl.io/ch5/findlinks3

$ ./findlinks3 https://golang.org

https://golang.org/

https://golang.org/doc/

https://golang.org/pkg/

https://golang.org/project/

https://code.google.com/p/go-tour/

https://golang.org/doc/code.html

https://www.youtube.com/watch?v=XCsL89YtqCs

http://research.swtch.com/gotour

当所有发现的链接都已经被访问或电脑的内存耗尽时，程序运行结束。

**练习5.10：**重写topoSort函数，用map代替切片并移除对key的排序代码。验证结果的正确性（结果不唯一）。

**练习5.11：**现在线性代数的老师把微积分设为了前置课程。完善topSort，使其能检测有向图中的环。

**练习5.12：**gopl.io/ch5/outline2（5.5节）的startElement和endElement共用了全局变量depth，将它们修改为匿名函数，使其共享outline中的局部变量。

**练习5.13：**修改crawl，使其能保存发现的页面，必要时，可以创建目录来保存这些页面。只保存来自原始域名下的页面。假设初始页面在golang.org下，就不要保存vimeo.com下的页面。

**练习5.14：**使用breadthFirst遍历其他数据结构。比如，topoSort例子中的课程依赖关系（有向图），个人计算机的文件层次结构（树），你所在城市的公交或地铁线路（无向图）。

#### 5.6.1 警告：捕获迭代变量

本节，将介绍Go词法作用域的一个陷阱。请务必仔细的阅读，弄清楚发生问题的原因。即使是经验丰富的程序员也会在这个问题上犯错误。

考虑这样一个问题：你被要求首先创建一些目录，再将目录删除。在下面的例子中我们用函数值来完成删除操作。下面的示例代码需要引入os包。为了使代码简单，我们忽略了所有的异常处理。

var rmdirs []func()

for \_, d := range tempDirs() {

dir := d // NOTE: necessary!

os.MkdirAll(dir, 0755) // creates parent directories too

rmdirs = append(rmdirs, func() {

os.RemoveAll(dir)

})

}

// ...do some work…

for \_, rmdir := range rmdirs {

rmdir() // clean up

}

你可能会感到困惑，为什么要在循环体中用循环遍历d赋值一个新的局部变量，而不是像下面的代码一样直接使用循环变量dir。需要注意，下面的代码是错误的。

var rmdirs []func()

for \_, dir := range tempDirs() {

os.MkdirAll(dir, 0755)

rmdirs = append(rmdirs, func() {

os.RemoveAll(dir) // NOTE: incorrect!

})

}

问题的原因在于循环变量的作用域。在上面的程序中，for循环语句引入了新的词法块，循环变量dir在这个词法块中被声明。在该循环中生成的所有函数值都共享相同的循环变量。需要注意，函数值中记录的是循环变量的内存地址，而不是循环变量某一时刻的值。以dir为例，后续的迭代会不断更新dir的值，当删除操作执行时，for循环已完成，dir中存储的值等于最后一次迭代的值。这意味着，每次对os.RemoveAll的调用删除的都是相同的目录。

通常，为了解决这个问题，我们会引入一个与循环变量同名的局部变量，作为循环变量的副本。比如下面的变量dir，虽然这看起来很奇怪，但却很有用。

for \_, dir := range tempDirs() {

dir := dir // declares inner dir, initialized to outer dir

// …

}

这个问题不仅存在基于range的循环，在下面的循环中，对循环变量i的使用也存在同样的问题：

var rmdirs []func()

dirs := tempDirs()

for i := 0; i < len(dirs); i++ {

os.MkdirAll(dirs[i], 0755) // ok

rmdirs = append(rmdirs, func() {

os.RemoveAll(dirs[i]) // NOTE: incorrect!

})

}

如果你使用go语句（第八章）或者defer语句（5.8节）会经常遇到此类问题。这不是go或defer本身导致的，而是因为它们都会等待循环结束后，再执行函数值。

### 5.7 可变参数

参数数量可变的函数称为可变参数函数。典型的例子就是fmt.Printf和类似函数。Printf首先接收一个必备的参数，之后接收任意个数的后续参数。

在声明可变参数函数时，需要在参数列表的最后一个参数类型之前加上省略符号“…”，这表示该函数会接收任意数量的该类型参数。

gopl.io/ch5/sum

func sum(vals ...int) int {

total := 0

for \_, val := range vals {

total += val

}

return total

}

sum函数返回任意个int型参数的和。在主函数体中，vals被看作是类型为[]int的切片。sum可以接收任意数量的int型参数：

fmt.Println(sum()) // "0"

fmt.Println(sum(3)) // "3"

fmt.Println(sum(1, 2, 3, 4)) // "10"

在上面的代码中，调用者隐式的创建了一个数组，并将原始参数复制到数组中，再把数组的一个切片作为参数传给被调用函数。如果原始参数已经是切片类型，我们该如何传递给sum？只需在最后一个参数后加上省略符。下面的代码功能与上个例子中最后一条语句相同。

values := []int{1,2,3,4}

fmt.Println(sum(values...))

虽然在可变参数函数内部，…int型参数的行为看起来很像切片类型，但实际上，可变参数函数和以切片作为参数的函数是不同的。

func f(...int) {}

func g([]int) {}

fmt.Printf("%T\n", f) // "func(...int)"

fmt.Printf("%T\n", g) // "func([]int)"

可变参数函数经常被用于格式化字符串。下面的errorf函数构造了一个以行号开头的，经过格式化的错误信息。函数名的后缀f是一种通用的命名规范，代表该可变参数函数可以接收Printf风格的格式化字符串。

func errorf(linenum int, format string, args ...interface{}) {

fmt.Fprintf(os.Stderr, "Line %d: ", linenum)

fmt.Fprintf(os.Stderr, format, args...)

fmt.Fprintln(os.Stderr)

}

linenum, name := 12, "count"

errorf(linenum, "undefined: %s", name) // "Line 12: undefined: count"

interface{}表示函数的最后一个参数可以接收任意类型，我们会在第7章详细介绍。

**练习5.15：**编写类似sum的可变参数函数max和min。考虑不传参是，max和min该如何处理，再编写至少接收1个参数的版本。

**练习5.16：**编写多参数版本的strings.Join。

**练习5.17：**编写多参数版本的ElementsByTagName，函数接收一个HTML结点树以及任意数量的标签名，返回与这些标签名匹配的所有元素。下面给出了2个例子：

func ElementsByTagName(doc \*html.Node, name …string) []\*html.Node

images := ElementsByTagName(doc, “img”)

headings := ElementsByTagName(doc, “h1”, “h2”, “h3”, “h4”)

### 5.8 Deferred函数

在findLinks的例子中，我们用http.Get的输出作为html.Parse的输入。只有url的内容的确是HTML格式的，html.Parse才可以正常工作，但实际上，url指向的内容很丰富，可能是图片，纯文本或是其他。将这些格式的内容传递给html.parse，会产生不良后果。

下面的例子获取HTML页面并输出页面的标题。title函数会检查服务器返回的Content-Type字段，如果发现页面不是HTML，将终止函数运行，返回错误。

gopl.io/ch5/title1

func title(url string) error {

resp, err := http.Get(url)

if err != nil {

return err

}

// Check Content-Type is HTML (e.g., "text/html;charset=utf-8").

ct := resp.Header.Get("Content-Type")

if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {

resp.Body.Close()

return fmt.Errorf("%s has type %s, not text/html", url, ct)

}

doc, err := html.Parse(resp.Body)

resp.Body.Close()

if err != nil {

return fmt.Errorf("parsing %s as HTML: %v", url, err)

}

visitNode := func(n \*html.Node) {

if n.Type == html.ElementNode && n.Data == "title" && n.FirstChild != nil {

fmt.Println(n.FirstChild.Data)

}

}

forEachNode(doc, visitNode, nil)

return nil

}

下面展示了运行效果：

$ go build gopl.io/ch5/title1

$ ./title1 http://gopl.io

The Go Programming Language

$ ./title1 https://golang.org/doc/effective\_go.html

Effective Go - The Go Programming Language

$ ./title1 https://golang.org/doc/gopher/frontpage.png

title: https://golang.org/doc/gopher/frontpage.png has type image/png, not text/html

resp.Body.close调用了多次，这是为了确保title在所有执行路径下（即使函数运行失败）都关闭了网络连接。随着函数变得复杂，需要处理的错误也变多，维护清理逻辑变得越来越困难。而Go语言独有的defer机制可以让事情变得简单。

你只需要在调用普通函数或方法前加上关键字defer，就完成了defer所需要的语法。当执行到该条语句时，函数和参数表达式得到计算，但直到包含该defer语句的函数执行完毕时，defer后的函数才会被执行，不论包含defer语句的函数是通过return正常结束，还是由于panic导致的异常结束。你可以在一个函数中执行多条defer语句，它们的执行顺序与声明顺序相反。

defer语句经常被用于处理成对的操作，如打开、关闭、连接、断开连接、加锁、释放锁。通过defer机制，不论函数逻辑多复杂，都能保证在任何执行路径下，资源被释放。释放资源的defer应该直接跟在请求资源的语句后。在下面的代码中，一条defer语句替代了之前的所有resp.Body.Close。

gopl.io/ch5/title2

func title(url string) error {

resp, err := http.Get(url)

if err != nil {

return err

}

defer resp.Body.Close()

ct := resp.Header.Get("Content-Type")

if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {

return fmt.Errorf("%s has type %s, not text/html", url, ct)

}

doc, err := html.Parse(resp.Body)

if err != nil {

return fmt.Errorf("parsing %s as HTML: %v", url, err)

}

// ...print doc's title element...

return nil

}

在处理其他资源时，也可以采用defer机制，比如对文件的操作：

io/ioutil

package ioutil

func ReadFile(filename string) ([]byte, error) {

f, err := os.Open(filename)

if err != nil {

return nil, err

}

defer f.Close()

return ReadAll(f)

}

或是处理互斥锁（9.2章）

var mu sync.Mutex

var m = make(map[string]int)

func lookup(key string) int {

mu.Lock()

defer mu.Unlock()

return m[key]

}

调试复杂程序时，defer机制也常被用于记录何时进入和退出函数。下例中的bigSlowOperation函数，直接调用trace记录函数的被调情况。bigSlowOperation被调时，trace会返回一个函数值，该函数值会在bigSlowOperation退出时被调用。通过这种方式，我们可以只通过一条语句控制函数的入口和所有的出口，甚至可以记录函数的运行时间，如例子中的start。需要注意一点：不要忘记defer语句后的圆括号，否则本该进入时执行的操作会在退出时执行，而本该在退出时执行的，永远不会被执行。

gopl.io/ch5/trace

func bigSlowOperation() {

defer trace("bigSlowOperation")() // don't forget the extra parentheses

// ...lots of work...

time.Sleep(10 \* time.Second) // simulate slow operation by sleeping

}

func trace(msg string) func() {

start := time.Now()

log.Printf("enter %s", msg)

return func() {

log.Printf("exit %s (%s)", msg, time.Since(start))

}

}

每一次bigSlowOperation被调用，程序都会记录函数的进入，退出，持续时间。（我们用time.Sleep模拟一个耗时的操作）

$ go build gopl.io/ch5/trace

$ ./trace

2015/11/18 09:53:26 enter bigSlowOperation

2015/11/18 09:53:36 exit bigSlowOperation (10.000589217s)

我们知道，defer语句中的函数会在return语句更新返回值变量后再执行，又因为在函数中定义的匿名函数可以访问该函数包括返回值变量在内的所有变量，所以，对匿名函数采用defer机制，可以使其观察函数的返回值。

以double函数为例：

func double(x int) int {

return x + x

}

我们只需要首先命名double的返回值，再增加defer语句，我们就可以在double每次被调用时，输出参数以及返回值。

func double(x int) (result int) {

defer func() { fmt.Printf(“double(%d) = %d\n”, x, result) }()

return x + x

}

\_ = double(4)

// Output:

// “double(4) = 8”

可能double函数过于简单，看不出这个小技巧的作用，但对于有许多return语句的函数而言，这个技巧很有用。

被延迟执行的匿名函数甚至可以修改函数返回给调用者的返回值：

func triple(x int) (result int) {

defer func() { result += x}()

return double(x)

}

fmt.Println(triple(4)) // “12”

在循环体中的defer语句需要特别注意，因为只有在函数执行完毕后，这些被延迟的函数才会执行。下面的代码会导致系统的文件描述符耗尽，因为在所有文件都被处理之前，没有文件会被关闭。

for \_, filename := range filenames {

f, err := os.Open(filename)

if err != nil {

return err

}

defer f.Close() // NOTE: risky; could run out of file descriptors

// …process f…

}

一种解决方法是将循环体中的defer语句移至另外一个函数。在每次循环前，调用这个函数。

for \_, filename := range filenames {

if err := doFile(filename); err != nil {

return err

}

}

func doFile(filename string) error {

f, err := os.Open(filename)

if err != nil {

return err

}

defer f.Close()

// …process f…

}

下面的代码是fetch（1.5节）的改进版，我们将http响应信息写入本地文件而不是从标准输出流输出。我们通过path.Base提出url路径的最后一段作为文件名。

gopl.io/ch5/fetch

// Fetch downloads the URL and returns the

// name and length of the local file.

func fetch(url string) (filename string, n int64, err error) {

resp, err := http.Get(url)

if err != nil {

return "", 0, err

}

defer resp.Body.Close()

local := path.Base(resp.Request.URL.Path)

if local == "/" {

local = "index.html"

}

f, err := os.Create(local)

if err != nil {

return "", 0, err

}

n, err = io.Copy(f, resp.Body)

// Close file, but prefer error from Copy, if any.

if closeErr := f.Close(); err != nil {

err = closeErr

}

return local, n, err

}

对resp.Body.Close延迟调用我们已经见过了，在此不做解释。上例中，通过os.Create打开文件进行写入，在关闭文件时，我们没有对f.close采用defer机制，因为这会产生一些微妙的错误。许多文件系统，尤其是NFS，写入文件时发生的错误会被延迟到文件关闭时反馈。如果没有检查文件关闭时的反馈信息，可能会导致数据丢失，而我们还误以为写入操作成功。如果io.Copy和f.close都失败了，我们倾向于将io.Copy的错误信息反馈给调用者，因为它先于f.close发生，更有可能接近问题的本质。

练习5.18：不修改fetch的行为，重写fetch函数，要求使用defer机制关闭文件。

### 5.9 Panic异常

Go的类型系统会在编译时捕获很多错误，但有些错误只能在运行时检查，如数组访问越界、空指针引用等。这些运行时错误会引起panic异常。

一般而言，当panic异常发生时，程序会中断运行，并立即执行在该goroutine（可以先理解成线程，在第8章会详细介绍）中被延迟的函数（defer机制）。随后，程序崩溃并输出日志信息。日志信息包括panic value和函数调用的堆栈跟踪信息。panic value通常是某种错误信息。对于每个goroutine，日志信息中都会有与之相对的，发生panic时的函数调用堆栈跟踪信息。通常，我们不需要再次运行程序去定位问题，日志信息已经提供了足够的诊断依据。因此，在我们填写问题报告时，一般会将panic异常和日志信息一并记录。

不是所有的panic异常都来自运行时，直接调用内置的panic函数也会引发panic异常；panic函数接受任何值作为参数。当某些不应该发生的场景发生时，我们就应该调用panic。比如，当程序到达了某条逻辑上不可能到达的路径：

switch s := suit(drawCard()); s {

case “Spades”: // …

case “Hearts”: // …

case “Diamonds”: // …

case “Clubs”: // …

default:

panic(fmt.Sprintf(“invalid suit %q”, s)) // Joker?

}

断言函数必须满足的前置条件是明智的做法，但这很容易被滥用。除非你能提供更多的错误信息，或者能更快速的发生错误，否则不需要使用断言，编译器在运行时会帮你检查代码。

func Reset(x \*Buffer) {

if x == nil {

panic(“x is nil”) // unnecessary!

}

x.elements = nil

}

虽然Go的panic机制类似于其他语言的异常，但panic的适用场景有一些不同。由于panic会引起程序的崩溃，因此panic一般用于严重错误，如程序内部的逻辑不一致。勤奋的程序员认为任何崩溃都表明代码中存在漏洞，所以对于大部分漏洞，我们应该使用Go提供的错误机制，而不是panic，尽量避免程序的崩溃。在健壮的程序中，任何可以预料到的错误，如不正确的输入、错误的配置或是失败的I/O操作都应该被优雅的处理，最好的处理方式，就是使用Go的错误机制。

考虑regexp.Compile函数，该函数将正则表达式编译成有效的可匹配格式。当输入的正则表达式不合法时，该函数会返回一个错误。当调用者明确的知道正确的输入不会引起函数错误时，要求调用者检查这个错误时不必要和累赘的。我们应该假设函数的输入一直合法，就如前面的断言一样：当调用者输入了不应该出现的输入时，触发panic异常。

在程序源码中，大多数正则表达式是字符串字面值（string literals），因此regexp包提供了包装函数regexp.MustCompile检查输入的合法性。

package regexp

func Compile(expr string) (\*Regexp, error) {/\*…\*/}

func MustCompile(expr string) \*Regexp {

re, err := Compile(expr)

if err != nil {

panic(err)

}

return re

}

包装函数使得调用者可以便捷的用一个编译后的正则表达式为包级别的变量赋值：

var httpSchemeRe = regexp.MustCompile(`^https?:`) // “http:” or “https:”

显然，MustCompile不能接受不合法的输入。函数名中的Must前缀是一种针对此类函数的命名约定，比如template.Must（4.6节）

func main() {

f(3)

}

func f(x int) {

fmt.Printf(“f(%d)\n”, x+0/x) // panics if x == 0

defer fmt.Printf(“defer %d\n”, x)

f(x - 1)

}

上例中的运行输出如下：

f(3)

f(2)

f(1)

defer 1

defer 2

defer 3

当f(0)被调用时，发生panic异常，之前被延迟执行的3个fmt.Printf被调用。程序中断执行后，panic信息和堆栈信息会被输出（下面是简化的输出）：

panic: runtime error: integer divide by zero

main.f(0)

src/gopl.io/ch5/defer1/defer.go:14

main.f(1)

src/gopl.io/ch5/defer1/defer.go:16

main.f(2)

src/gopl.io/ch5/defer1/defer.go:16

main.f(3)

src/gopl.io/ch5/defer1/defer.go:16

main.main()

src/gopl.io/ch5/defer1/defer.go:10

我们在下一节将看到，如何使程序从panic异常中恢复，阻止程序的崩溃。

为了方便诊断问题，runtime包允许程序员输出堆栈信息。在下面的例子中，我们通过在main函数中延迟调用printStack输出堆栈信息。

gopl.io/ch5/defer2

func main() {

defer printStack()

f(3)

}

func printStack() {

var buf [4096]byte

n := runtime.Stack(buf[:], false)

os.Stdout.Write(buf[:n])

}

printStack的简化输出如下（下面只是printStack的输出，不包括panic的日志信息）：

goroutine 1 [running]:

main.printStack()

src/gopl.io/ch5/defer2/defer.go:20

main.f(0)

src/gopl.io/ch5/defer2/defer.go:27

main.f(1)

src/gopl.io/ch5/defer2/defer.go:29

main.f(2)

src/gopl.io/ch5/defer2/defer.go:29

main.f(3)

src/gopl.io/ch5/defer2/defer.go:29

main.main()

src/gopl.io/ch5/defer2/defer.go:15

将panic机制类比其他语言异常机制的读者可能会惊讶，runtime.Stack为何能输出已经被释放函数的信息？在Go的panic机制中，延迟函数的调用在释放堆栈信息之前。

### 5.10 Recover捕获异常

通常来说，不应该对panic异常做任何处理，但有时，也许我们可以从异常中恢复，至少我们可以在程序崩溃前，做一些操作。举个例子，当web服务器遇到不可预料的严重问题时，在崩溃前应该将所有的连接关闭；如果不做任何处理，会使得客户端一直处于等待状态。如果web服务器还在开发阶段，服务器甚至可以将异常信息反馈到客户端，帮助调试。

如果在deferred函数中调用了内置函数recover，并且定义了defer语句的函数发生了panic异常，recover会使程序从panic中恢复，并返回panic value。导致panic异常的函数不会继续运行，但能正常返回。在未发生panic时调用recover，recover会返回nil。

让我们以语言解析器为例，说明recover的使用场景。考虑到语言解析器的复杂性，即使某个语言解析器目前工作正常，也无法肯定它没有漏洞。因此，当某个异常出现时，我们不会选择让解析器崩溃，而是会将panic异常当做普通的解析错误，并附加额外信息提醒用户报告此错误。

func Parse(input string) (s \*Syntax, err error) {

defer func() {

if p := recover(); p != nil {

err = fmt.Errorf(“internal error: %v”, p)

}

}()

// …parser…

}

deferred函数帮助Parse从panic中恢复。在deferred函数内部，panic value被附加到错误信息中；并用err变量接收错误信息，返回给调用者。我们也可以通过调用runtime.Stack往错误信息中添加完整的堆栈调用信息。

不加区分的恢复所有的panic异常，不是可取的做法；因为在panic之后，无法保证包级变量的状态仍然和我们预期一致。比如，对数据结构的一次重要更新没有被完整完成、文件或者网络连接没有被关闭、获得的锁没有被释放。此外，如果写日志时产生的panic被不加区分的恢复，可能会导致漏洞被忽略。

虽然把对panic的处理都集中在一个包下，有助于简化对复杂和不可以预料问题的处理，但作为被广泛遵守的规范，你不应该试图去恢复其他包引起的panic。公有的API应该将函数的运行失败作为error返回，而不是panic。同样的，你也不应该恢复一个由他人开发的函数引起的panic，比如说调用者传入的回调函数，因为你无法确保这样做事安全的。

有时我们很难完全遵循规范，举个例子，net/http包中提供了一个web服务器，将收到的请求分发给用户提供的处理函数。很显然，我们不能因为某个处理函数引发的panic异常，杀掉整个进程；web服务器遇到处理函数导致的panic时回调用panic，输出堆栈信息，继续运行。这样的做法在实践中很便捷，但也会引起资源泄露，或是因为recover操作，导致其他问题。

基于以上原因，安全的做法是有选择性的recover。换句话说，只恢复应该被恢复的panic异常，此外，这些异常所占的比例应该尽可能的低。为了标识某个panic是否应该被恢复，我们可以将panic value设置成特殊类型。在recover时对panic value进行检查，如果发现panic value是特殊类型，就将这个panic作为error处理，如果不是，则按照正常的panic进行处理（在下面的例子中，我们会看到这种方式）。

下面的例子是title函数的变形，如果HTML页面包含多个<title>，该函数会给调用者返回一个错误（error）。在soleTitle内部处理时，如果检测到右多个<title>，会调用panic，阻止函数继续递归，并将特殊类型bailout作为panic的参数。

// soleTitle returns the text of the first non-empty title element

// in doc, and an error if there was not exactly one.

func soleTitle(doc \*html.Node) (title string, err error) {

type bailout struct{}

defer func() {

switch p := recover(); p {

case nil: // no panic

case bailout{}: // "expected" panic

err = fmt.Errorf("multiple title elements")

default:

panic(p) // unexpected panic; carry on panicking

}

}()

// Bail out of recursion if we find more than ont nonempty title.

forEachNode(doc, func(n \*html.Node) {

if n.Type == html.ElementNode && n.Data == "title" && n.FirstChild != nil {

if title != "" {

panic(bailout{}) // multiple title elements

}

title = n.FirstChild.Data

}

}, nil)

if title == "" {

return "", fmt.Errorf("no title element")

}

return title, nil

}

在上例中，deferred函数调用recover，并检查panic value。当panic value是bailout{}类型时，deferred函数生成一个error返回给调用者。当panic value是其他non-nil值是，表示发生了未知的panic异常，deferred函数将调用panic函数并将当前的panic value作为参数传入；此时，等同于recover没有做任何操作。（请注意：在例子中，对可预期的错误采用了panic，这违反了之前的建议，我们在此只是想向读者演示这种机制。）有些情况下，我们无法恢复。某些致命错误会导致Go在运行时终止程序，如内存不足。

**练习5.19：**使用panic和recover编写一个不包含return语句但能返回一个非零值的函数。

## 方法

从90年代早期开始，面向对象编程（OOP）就成为了称霸工程界和教育界的编程范式，所以之后几乎所有大规模被应用的语言都包含了对OOP的支持，go语言也不例外。

尽管没有被大众所接受的明确的OOP的定义，从我们的理解来讲，一个对象其实也就是一个简单的值或者一个变量，在这个对象中会包含一些方法，而一个方法则是一个一个和特殊类型关联的函数。一个面向对象的程序会用方法来表达其属性和对应的操作，这样使用这个对象的用户就不需要直接去操作对象，而是借助方法来做这些事情。

在早些的章节中，我们已经使用了标准库提供的一些方法，比如time.Duration这个类型的Seconds方法：

const day = 24 \* time.Hour

fmt.Println(day.Seconds()) // “86400”

并且在2.5节中，我们定义了一个自己的方法，Celsius类型的String方法：

func (c Celsius) String() string {

return fmt.Sprintf(“%g℃”, c)

}

在本章中，OOP编程的第一方面，我们会向你展示如何有效地定义和使用方法。我们会覆盖到OOP变成的两个关键点，封装和组合。

### 6.1 方法声明

在函数声明前，在其名字之前放上一个变量，即是一个方法。这个附加的参数会将该函数附加到这种类型上，即相当于为这种类型定义了一个独占的方法。

下面来写我们第一个方法的例子，这个例子在package geoemtry下：

gopl.io/ch6/geometry

package geometry

import "math"

type Point struct {

X, Y float64

}

// traditional function

func Distance(p, q Point) float64 {

return math.Hypot(q.X-p.X, q.Y-p.Y)

}

// same thing, but as a method of the Point type

func (p Point) Distance(q Point) float64 {

return math.Hypot(q.X-p.X, q.Y-p.Y)

}

上面的代码里那个附加的参数p，叫做方法的接收器（receiver），早期的面向对象语言留下的遗产将调用一个方法称为“向一个对象发送消息”。

在Go语言中，我们并不会像其他语言那样用this或者self作为接收器；我们可以任意的选择接收器的名字。由于接收器的名字经常会被使用到，所以保持其在方法间传递时的一致性和简短性是不错的主意。这里的建议是可以使用其类型的第一个字母，比如这里使用了Point的首字母p。

在方法调用过程中，接收器参数一般会在方法名之前出现。这和方法声明是一样的，都是接收器参数在方法名字之前。下面是例子：

p :=Point{1, 2}

q :=Point{4, 6}

fmt.Println(Distance(p, q)) // "5", function call

fmt.Println(p.Distance(q)) // "5", method call

可以看到，上面的两个函数调用都是Distance，但是却没有发生冲突。第一个Distance的调用实际上用的是包级别的函数geometry.Distance，而第二个则是使用刚刚声明的Point，调用的是Point类下声明的Point.Distance方法。

这种p.Distance的表达式叫做选择器，因为他会选择合适的对应p这个对象的Distance方法来执行。选择器也会被用来选择一个struct类型的字段，比如p.X。由于方法和字段都是在同一命名空间，所以如果我们在这里声明一个X方法的话，编译器会报错，因为在调用p.X时会有歧义（译注：这里确实挺奇怪的）。

因为每种类型都有其方法的命名空间，我们在用Distance这个名字的时候，不同的Distance调用指向了不同类型里的Distance方法。让我们来定义一个Path类型，这个Path代表一个线段的集合，并且也给这个Path定义一个叫Distance的方法。

// A Path is a journey connecting the points with straight lines

type Path []Point

// Distance returns the distance traveled along the path.

func (path Path) Distance() float64 {

sum := 0.0

for i := range path {

if i > 0 {

sum += path[i-1].Distance(path[i])

}

}

return sum

}

Path是一个命名的slice类型，而不是Point那样的struct类型，然而我们依然可以为它定义方法。在能够给任意类型定义方法这一点上，Go和很多其他的面向对象的语言不太一样。因此在Go语言里，我们为一些简单的数值、字符串、slice、map来定义一些附加行为很方便。我们可以给同一个包内的任意命名类型定义方法，只要这个命名类型的底层类型（译注：这个例子里，底层类型是指[]Point这个slice，Path就是命名类型）不是指针或者interface。

两个Distance方法有不同的类型。他们两个方法之间没有任何关系，尽管Path的Distance方法会在内部调用Point.Distance方法来计算每个连接邻接点的线段的长度。

让我们来调用一个新方法，计算三角形的周长：

perim :=Path{

{1, 1},

{5, 1},

{5, 4},

{1, 1},

}

fmt.Println(perim.Distance()) // "12"

在上面两个对Distance名字的方法的调用中，编译器会根据方法的名字以及接收器来决定具体调用的是哪一个函数。第一个例子中path[i-1]数组中的类型是Point，因此Point.Distance这个方法被调用；在第二个例子中perim的类型是Path，因此Distance调用的是Path.Distance。

对于一个给定的类型，其内部的方法都必须有唯一的方法名，但是不同的类型却可以有同样的方法名，比如我们这里Point和Path就都有Distance这个名字的方法；所以我们没有必要非在方法名之前加类型名来消除歧义，比如PathDistance。这里我们已经看到了方法比之函数的一些好处：方法名可以简短。当我们在包外调用的时候这种好处就会被放大，因为我们可以使用这个短名字，而可以省略掉包的名字，下面是例子：

import “gopl.io/ch6/geometry”

perim := geometry.Path{

{1, 1},

{5, 1},

{5, 4},

{1, 1},

}

fmt.Println(geometry.PathDistance(perim)) // "12", standalone function

fmt.Println(perim.Distance()) // "12", method of geometry.Path

译注：如果我们要用方法去计算perim的distance，还需要去写全geometry的包名，和其函数名，但是因为Path这个类型定义了一个可以直接用的Distance方法，所以我们可以直接写perim.Distance()。相当于可以少打很多字，作者应该是这个意思。因为在Go里包外调用函数需要带上包名，还是挺麻烦的。

### 6.2 基于指针对象的方法

当调用一个函数时，会对其每一个参数值进行拷贝，如果一个函数需要更新一个变量，或者函数的其中一个参数实在太大我们希望能够避免进行这种默认的拷贝，这种情况下我们就需要用到指针了。对应到我们这里用来更新接收器的对象的方法，当这个接受者变量本身比较大时，我们就可以用其指针而不是对象来声明方法，如下：

func (p \*Point) ScaleBy(factor float64) {

p.X \*= factor

p.Y \*= factor

}

这个方法的名字是(\*Point).ScaleBy。这里的括号是必须的；没有括号的话这个表达式可能会被理解为\*(Point.ScaleBy)。

在现实的程序里，一般会约定如果Point这个类有一个指针作为接收器的方法，那么所有Point的方法都必须有一个指针接收器，即使是那些并不需要这个指针接收器的函数。我们在这里打破了这个约定只是为了展示一下两种方法的异同而已。

只有类型(Point)和指向他们的指针(\*Point)，才可能是出现在接收器声明里的两种接收器。此外，为了避免歧义，在声明方法时，如果一个类型名本身是一个指针的话，是不允许其出现在接收器中的，比如下面这个例子：

type P \*int

func (P) f() { /\* … \*/ } // compile error: invalid receiver type

想要调用指针类型方法(\*Point).ScaleBy，只要提供一个Point类型的指针即可，像下面这样。

r := &Point{1, 2}

r.ScaleBy(2)

fmt.Println(\*r) // "{2, 4}"

或者这样：

p :=Point{1, 2}

pptr := &p

pptr.ScaleBy(2)

fmt.Println(p) // "{2, 4}"

或者这样：

p :=Point{1, 2}

(&p).ScaleBy(2)

fmt.Println(p) // "{2, 4}"

不过后面两种方法有些笨拙。幸运的是，go语言本身在这种地方会帮到我们。如果接收器p是一个Point类型的变量，并且其方法需要一个Point指针作为接收器，我们可以用下面这种简短的写法：

p.ScaleBy(2)

编译器会隐式地帮我们用&p去调用ScaleBy这个方法。这种简写方法只适用于“变量”，包括struct里的字段比如p.X，以及array和slice内的元素比如perim[0]。我们不能通过一个无法取到地址的接收器来调用指针方法，比如临时变量的内存地址就无法获取得到：

Point{1, 2}.ScaleBy(2) // compile error: can’t take address of Point literal

但是我们可以用一个\*Point这样的接收器来调用Point的方法，因为我们可以通过地址来找到这个变量，只要用解引用符号\*来取到该变量即可。编译器在这里也会给我们隐式地插入\*这个操作符，所以下面这两种写法等价的：

pptr.Distance(q)

(\*pptr).Distance(q)

这里的几个例子可能让你有些困惑，所以我们总结一下：在每一个合法的方法调用表达式中，也就是下面三种情况里的任意一种情况都是可以的：

不论接收器的实际参数和其形式参数是相同，比如两者都是类型T或者都是类型\*T：

Point{1, 2}.Distance(q) // Point

pptr.ScaleBy(2) // \*Point

或者接收器实参是类型T，但接收器形参是类型\*T，这种情况下编译器会隐式地为我们取变量的地址：

p.ScaleBy(2) // implicit(&p)

或者接收器实参是类型\*T，形参是类型T。编译器会隐式地为我们解引用，取到指针指向的实际变量：

pptr.Distance(q) // implicit (\*pptr)

如果命名类型T（译注：用type xxx定义的类型）的所有方法都是用T类型自己来做接收器（而不是\*T），那么拷贝这种类型的实例就是安全的；调用他的任何一个方法也就会产生一个值的拷贝。比如time.Duration的这个类型，在调用其方法时就会被全部拷贝一份，包括在作为参数传入函数的时候。但是如果一个方法使用指针作为接收器，你需要避免对其进行拷贝，因为这样可能会破坏掉该类型内部的不变性。比如你对bytes.Buffer对象进行了拷贝，那么可能会引起原始对象和拷贝对象只是别名而已，实际上它们指向的对象是一样的。紧接着对拷贝后的变量进行修改可能会让你有意外的结果。

译注：作者这里说的比较绕，其实有两点：

1. 不管你的method的receiver是指针类型还是非指针类型，都是可以通过指针/非指针类型进行调用的，编译器会帮你做类型转换。
2. 在声明一个method的receiver该是指针还是非指针时，你需要考虑两方面的因素，第一方面是这个对象本身是不是特别大，如果声明为非指针变量时，调用会产生一次拷贝；第二方面是如果你用指针类型作为receiver，那么你一定要注意，这种指针类型指向的始终是一块内存地址，就算你对其进行了拷贝。熟悉C或者C++的人这里应该很快能明白。

#### 6.2.1 Nil也是一个合法的接收器类型

就像一些函数允许nil指针作为参数一样，方法理论上也可以用nil指针作为其接收器，尤其当nil对于对象来说是合法的零值时，比如map或者slice。在下面的简单int链表的例子里，nil代表的是空链表：

// An IntList is a linked list of integers.

// A nil \*IntList represents the empty list.

type IntList struct {

Value int

Tail \*IntList

}

// Sum returns the sum of the list elements.

func (list \*IntList) Sum() int {

if list == nil {

return 0

}

return list.Value + list.Tail.Sum()

}

当你定义一个允许nil作为接收器值的方法的类型时，在类型前面的注释中指出nil变量代表的意义是很有必要的，就像我们上面例子里做的这样。

下面是net/url包里Values类型定义的一部分。

net/url

package url

// Values maps a string key to a list of values

type Values map[string][]string

// Get returns the first value associated with the given key,

// or “” if there are none.

func (v Values) Get(key string) string {

if vs := v[key]; len(vs) > 0 {

return vs[0]

}

return “”

}

// Add adds the value to key

// It appends to any existing values associated with key.

func (v Values) Add(key, value string) {

v[key] = append(v[key], value)

}

这个定义向外部暴露了一个map的命名类型，并且提供了一些能够简单操作这个map的方法。这个map的value字段是一个string的slice，所以这个Values是一个多维map。客户端使用这个变量的时候可以使用map固有的一些操作（make，切片，m[key]等等），也可以使用这里提供的操作方法，或者两者并用，都是可以的：

gopl.io/ch6/urlvalues

m := url.Values{"lang": {"en"}} // direct construction

m.Add("item", "1")

m.Add("item", "2")

fmt.Println(m.Get("lang")) // "en"

fmt.Println(m.Get("q")) // ""

fmt.Println(m.Get("item")) // "1" (first value)

fmt.Println(m["item"]) // "[1 2]" (direct map access)

m = nil

fmt.Println(m.Get("item"))

m.Add("item", "3") // panic: assignment to entry in nil map

对Get的最后一次调用中，nil接收器的行为即是一个空map的行为。我们可以等价地将这个操作写成Value(nil).Get(“item”)，但是如果你直接写nil.Get(“item”)的话是无法通过编译的，因为nil的字面量编译器无法判断其准确类型。所以相比之下，最后的那行m.Add的调用就会产生一个panic，因为他尝试更新一个空map。

由于url.Values是一个map类型，并且间接引用了其key/value对，因此url.Values.Add对这个map里的元素做任何的更新、删除操作对调用方都是可见的。实际上，就像在普通函数中一样，虽然可以通过引用来操作内部值，但在方法想要修改引用本身时是不会影响原始值的，比如把他置换为nil，或者让这个引用指向了其他的对象，调用方都不会受影响。（译注：因为传入的是存储了内存地址的变量，你改变这个变量本身是影响不了原始的变量的，想想C语言，是差不多的。）

### 6.3 通过嵌入结构体来扩展类型

来看看ColoredPoint这个类型：

gopl.io/ch6/coloredpoint

import "image/color"

type Point struct {

X, Y float64

}

type ColoredPoint struct {

Point

Color color.RGBA

}

我们完全可以将ColoredPoint定义为一个有三个字段的struct，但是我们却将Point这个类型嵌入到ColoredPoint来提供X和Y这两个字段。像我们在4.4节中看到的那样，内嵌可以使我们在定义ColoredPoint时得到一种语法上的简写形式，并使其包含Point类型所具有的一切字段，然后再定义一些自己的。如果我们想要的话，我们可以直接认为通过嵌入的字段就是ColoredPoint自身的字段，而完全不需要在调用时指出Point，比如下面这样。

var cp ColoredPoint

cp.X = 1

fmt.Println(cp.Point.X) // "1"

cp.Point.Y = 2

fmt.Println(cp.Y) // "2"

对于Point中的方法我们也有类似的用法，我们可以把ColoredPoint类型当作接收器来调用Point里的方法，即使ColoredPoint里没有声明这些方法：

red := color.RGBA{255, 0, 0, 255}

blue := color.RGBA{0, 0, 255, 255}

var p = ColoredPoint{Point{1, 1}, red}

var q = ColoredPoint{Point{5, 4}, blue}

fmt.Println(p.Distance(q.Point)) // "5"

p.ScaleBy(2)

q.ScaleBy(2)

fmt.Println(p.Distance(q.Point)) // "10"

Point类的方法也被引入了ColoredPoint。用这种方式，内嵌可以使我们定义字段特别多的复杂类型，我们可以将字段先按小类型分组，然后定义小类型的方法，之后再把它们组合起来。

读者如果对基于类来实现面向对象的语言比较熟悉的话，可能会倾向于将Point看作一个基类，而ColoredPoint看作其子类或者继承类，或者将ColoredPoint看作“is a”Point类型。但这是错误的理解。请注意上面例子中对Distance方法的调用。Distance有一个参数是Point类型，但q并不是一个Point类，所以尽管q有着Point这个内嵌类型，我们也必须要显式地选择它。尝试直接传q的话你会看到下面这样的错误：

p.Distance(q) // compile error: cannot use q(ColoredPoint) as Point

一个ColoredPoint并不是一个Point，但他“has a”Point，并且它有从Point类里引入的Distance和ScaleBy方法。如果你喜欢从实现的角度来考虑问题，内嵌字段会指导编译器去生成额外的包装方法来委托已经声明好的方法，和下面的形式是等价的：

func (p ColoredPoint) Distance(q Point) float64 {

return p.Point.Distance(q)

}

func (p \*ColoredPoint) ScaleBy(factor float64) {

p.Point.ScaleBy(factor)

}

当Point.Distance被第一个包装方法调用时，它的接收器值是p.Point，而不是p，当然了，在Point类的方法里，你是访问不到ColoredPoint的任何字段的。

在类型中内嵌的匿名字段也可能是一个命名类型的指针，这种情况下字段和方法会被间接地引入到当前的类型中（译注：访问需要通过该指针指向的对象去取）。添加这一层间接关系让我们可以共享通用的结构并动态地改变对象之间的关系。下面这个ColoredPoint的声明内嵌了一个\*Point的指针。

type ColoredPoint struct {

\*Point

Color color.RGBA

}

p := ColoredPoint{&Point{1, 1}, red}

q := ColoredPoint{&Point{5, 4}, blue}

fmt.Println(p.Distance(\*q.Point) // “5”

q.Point = p.Point // p and q now share the same Point

p.ScaleBy(2)

fmt.Println(\*p.Point, \*q.Point) // “{2 2} {2 2”

一个struct类型也可能会有多个匿名字段。我们将ColoredPoint定义为下面这样：

type ColoredPoint struct {

Point

color.RGBA

}

然后这种类型的值便会拥有Point和RGBA类型的所有方法，以及直接定义在ColoredPoint中的方法。当编译器解析一个选择器到方法时，比如p.ScaleBy，它会首先去找直接定义在这个类型里的ScaleBy方法，然后找被ColoredPoint的内嵌字段们引入的方法，然后去找Point和RGBA的内嵌字段引入的方法，然后一直递归向下找。如果选择器有二义性的话编译器会报错，比如你在同一级里有两个同名的方法。

方法只能在命名类型（像Point）或者指向类型的指针上定义，但是多亏了内嵌，有些时候我们给匿名struct类型来定义方法也有了字段。

下面是一个小trick。这个例子展示了简单的cache，其使用两个包级别的变量来实现，一个mutex互斥量（§9.2）和它所操作的cache：

var (

mu sync.Mutex // guards mapping

mapping = make(map[string]string)

)

func Lookup(key string) string {

mu.Lock()

v := mapping[key]

mu.Unlock()

return v

}

下面这个版本在功能上是一致的，但将两个包级别的变量放在了cache这个struct一组内：

var cache = struct {

sync.Mutex

mapping map[string]string

}{

mapping: make(map[string]string),

}

func Lookup(key string) string {

cache.Lock()

v := cache.mapping[key]

cache.Unlock()

return v

}

我们给新的变量起了一个更具表达性的名字：cache。因为sync.Mutex字段也被嵌入到了这个struct里，其Lock和Unlock方法也就都被引入到了这个匿名结构中了，这让我们能够以一个简单明了的语法来对其进行加锁解锁操作。

### 6.4 方法值和方法表达式

### 6.5 示例：Bit数组

### 6.6 封装

## 第七章 接口

## 第八章 Goroutines和Channels

## 第九章 基于共享变量的开发

## 第十章 包和工具

## 第十一章 测试

## 第十二章 反射

## 第十三章 底层编程

## 第十四章 附录