Go语言圣经（中文版）

The Go Programming Language

Alan A. A. Donovan，Brian W. Kernighan（著）
chai2010，Xargin，CrazySssst，foreversmart（译）

http://github.com/golang-china/gopl-zh

前言

译者序

在上个世纪 70 年代，贝尔实验室的 Ken Thompson 和 Dennis M. Ritchie 合作发明了 UNIX 操作系统，同 时 Dennis M. Ritchie 爲了解决 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的中 Linux 和 Mac OS X 都是源 于 UINX 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 UINX 和 C 语言发明 40 年之后，目前已经在 Google 工作的 Ken Thompson 和 Rob Pike（他们在贝尔实验室时就是同事）、还有 Robert Griesemer（设计了 V8 引擎和 HotSpot 虚拟机）一起合作，爲了解决在 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）：

```
C:\go\go-tip>hg log -r 0:4
changeset:   0:f6182e5abf5e
user:        Brian Kernighan <bwk>
date:        Tue Jul 18 19:05:45 1972 -0500
summary:     hello, world

changeset:   1:b66d0bf8da3e
user:        Brian Kernighan <bwk>
date:        Sun Jan 20 01:02:03 1974 -0400
summary:     convert to C

changeset:   2:ac3363d7e788
user:        Brian Kernighan <research!bwk>
date:        Fri Apr 01 02:02:04 1988 -0500
summary:     convert to Draft-Proposed ANSI C

changeset:   3:172d32922e72
user:        Brian Kernighan <bwk@research.att.com>
date:        Fri Apr 01 02:03:04 1988 -0500
summary:     last-minute fix: convert to ANSI C

changeset:   4:4e9a5b095532
user:        Robert Griesemer <gri@golang.org>
date:        Sun Mar 02 20:47:34 2008 -0800
summary:     Go spec starting point.


C:\go\go-tip>
```

从早期提交日志中也可以看出，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 语言称爲 21 世纪的 C 语言。纵观这几年来的发 展趋势，Go 语言已经成爲云计算、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 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》，该书被誉爲 C 语言程序员的圣经，作者也 被大家亲切地称爲 K&R。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 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》。Go 语言被誉爲 21 世纪的 C 语言，如果说 K&R 所着 的是圣经的旧约，那麼 D&K 所着的必将成爲圣经的新约。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 的必读书 目。同时，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，单凭閱读和学习其语法结构并不能眞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，同时同步发售的还有日文版本。不过比较可惜的是，中文版并 没有在同步发售之列，甚至连中文版是否会引进、是由哪个出版社引进、即使引进将由何人来翻译、何时 能出版都成了一个祕密。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 人民币）。爲了 Go 语言的学习和交流，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。其 中，chai2010 翻译了前言、第 2~4 章、第 10~13 章，Xargin 翻译了第 1 章、第 6 章、第 8~9 章，CrazySssst 翻译了第 5 章，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，同时避免引入过度的复杂性。 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 语言已经成爲受欢迎的作爲无类型的脚本语言的替代者： 因爲 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，可以免费获编译器、库、配套工具的源代码。 Go 语言的贡献者来自一个活跃的全球社区。Go 语言可以运行在类 UNIX 系统—— 比如 Linux、FreeBSD、OpenBSD、Mac OSX——和 Plan9 系统和 Microsoft Windows 操作系统之上。 Go 语言编写的程序无需脩改就可以运行在上面这些环境。

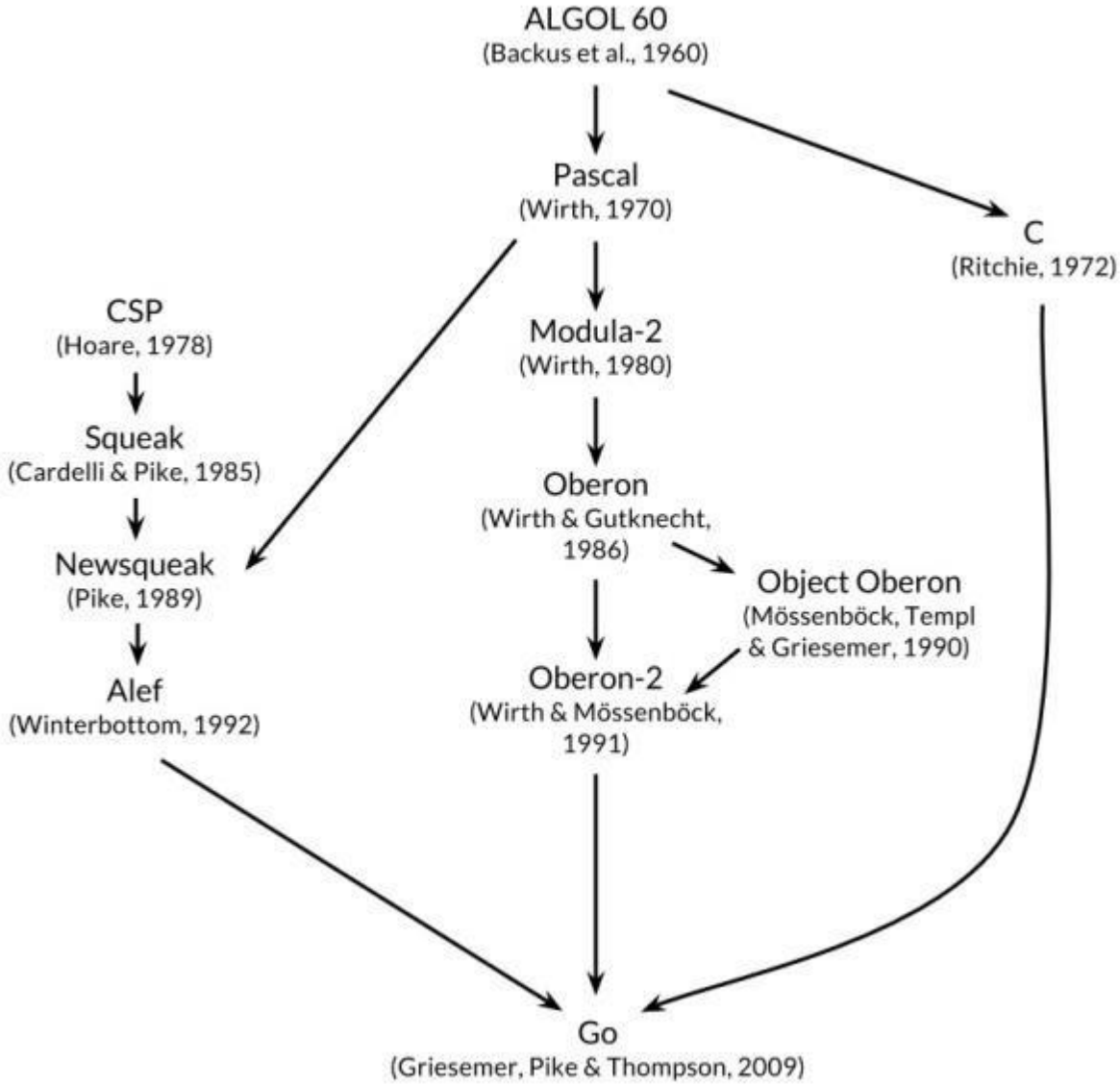
本书是爲了帮助你开始以有效的方式使用 Go 语言，充分利用语言本身的特性和自带的标准库去编写清晰地 道的 Go 程序。

Go 语言起源

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优点；当然有时多种语言杂合也可能会産生令人惊讶的特性；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 语言的设计産生了重要影响。



Go 语言有时候被描述爲“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 所设计 的 https://en.wikipedia.org/wiki/Pascal_(programming_language)">Pascal 语言。然后 Modula-2 语言 激发了包的概念。然后 https://en.wikipedia.org/wiki/Oberon_(programming_language)">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 https://en.wikipedia.org/wiki/Oberon-2_(programming_language)">Oberon-2 语言直接影响了包的导入和声明的语法，还 有 https://en.wikipedia.org/wiki/Oberon_(programming_language)">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 于 1978 年发表的鲜爲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*，缩写爲 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 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（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 语言，提供了类似 C 语言语句和表达式的语法和类 似 https://en.wikipedia.org/wiki/Pascal_(programming_language)">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 改造爲系统编程语言，但是因爲缺少垃圾回收机制而导致并发编程很痛苦。（译注：在 Aelf 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 ）

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 https://en.wikipedia.org/wiki/APL_(programming_language)">APL 语言借鉴，词法作用域与嵌套函 数来自于 https://en.wikipedia.org/wiki/Scheme_(programming_language)">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 语言的切片爲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

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（但是这类问题絶不是 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脩复问题通常将慢 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脩复问题的最快的途径，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，关于简洁编程哲学的 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 作爲一等公民、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，没有函数脩饰，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 语言简洁的类型系统给了程序员 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，因爲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（通常被称爲语言自带的电池），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译码协议。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名 称，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

本书的组织

我们假设你已经有一个或多个其他编程语言的使用经历，不管是类似 C、c++或 Java 的编译型语言，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。因爲 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。

第二章描述了一个 Go 语言程序的基本元素结构、变量、定义新的类型、包和文件、以及作用域的概念。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，对于任何主流命令式编程语言这个部分都是类似的。虽然有时候 Go 语言的语 法和风格会有自己的特色，但是大多数程序员将能很快地适应。剩下的章节是 Go 语言中特有地方：方法、

接口、并发、包、测试和反射等语言特性。

Go 语言的面向对象是不同寻常的。它没有类层次结构，甚至可以说没有类；仅仅是通过组合（而不是继承）简单的对象来构建复杂的对象。方法不仅仅可以定义在结构体上, 而且可以定义在任何用户自己定义的类型上；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满足了哪些接口。方法将在第六章讨论，接口将在第七章将讨论。

第八章讨论了基于顺序通信进程(CSP)概念的并发编程，通过使用 goroutines 和 channels 处理并发编程。 第九章则讨论了更爲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的利用 Go 自带的工具，通过一个命令提供 了编译、测试、基准测试、代码格式化、文档和许多其他任务。

第十一章讨论了单元测试，Go 语言的工具和标准库中集成的轻量级的测试功能，从而避免了采用强大但复 杂的测试框架。测试库提供一些基本的构件，如果有必要可以用来构建更复杂的测试构件。

第十二章讨论了反射，一个程序在运行期间来审视自己的能力。反射是一个强大的编程工具，不过要谨慎地使用；这一章通过用利用反射机制实现一些重要的 Go 语言库函数来展示了反射的强大用法。第十三章解 释了底层编程的细节，通过使用 unsafe 包来绕过 Go 语言安全的类型系统，当然有时这是必要的。

有些章节的后面可能会有一些练习，你可以根据你对 Go 语言的理解，然后脩改书中的例子来探索 Go 语言 的其他用法。

书中所有的代码都可以从 http://gopl.io 上的 Git 仓库下载。go get 命令可以根据每个例子的其导入路径智 能地获取、构建并安装。你只需要选择一个目録作爲工作空间，然后将 GOPATH 环境指向这个工作目録。

Go 语言工具将在必要时创建的相应的目录。例如：

```
$ export GOPATH=$HOME/gobook    # 選擇工作目録
$ go get gopl.io/ch1/helloworld # 獲取/編譯/安装
$ $GOPATH/bin/helloworld        # 運行程序
Hello, 世界                     # 這是中文
```

要运行这些例子, 你需要安装 Go1.5 以上的版本.

```
$ go version
go version go1.5 linux/amd64
```

如果你用的是其他的操作系统, 请参考 https://golang.org/doc/install 提供的帮助安装。

更多的信息

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/ 包含了官方收録的各种报告的讲稿）。

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（而纸质书则没有这麼便利了）。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 可以简单的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，滙报 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，而且因爲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，对于一些更复制的实验，你可能需要在自己的计算机上构建 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，下载时间可能会因爲洞 的大小等因素从几分钟到几天或更久），然后就可以在自己计算机上编写和运行 Go 程序了。

Go 语言是一个开源项目，你可以在 https://golang.org/pkg 閱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

致谢

Rob Pike 和 Russ Cox，以及很多其他 Go 团队的核心成员多次仔细閱读了本书的手稿，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，Yoshiki Shibata 更是仔细地审閱了 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閱者，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 、 Andrew Gerrand、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 生產团队的 John Fuller、Dayna Isley、Julie Nahil、Chuti Prasertsith 到 Barbara Wood，感谢 你们的热心帮助。

Alan Donovan 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 并没有让他爲了家庭琐事而分心，并热情坚定地支持这个项目。

Brian Kernighan 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

第 1 章 入门

本章会介绍 Go 语言里的一些基本组件。我们希望用信息和例子尽快带你入门。本章和之后章节的例子都是 针对眞实的开发案例给出。本章我们只是简单地爲你介绍一些 Go 语言的入门例子，从简单的文件处理、图 像处理到互联网并发客户端和服务端程序。当然，在第一章我们不会详尽地一一去帮助细枝末节，不过用 这些程序来学习一门新语言肯定是很有效的。

当你学习一门新语言时，你会用这门新语言去重写自己以前熟悉语言例子的倾向。在学习 Go 语言的过程中，尽量避免这麽做。我们会向你演示如何才能写出好的 Go 语言程序，所以请使用这里的代码作爲你写自 己的 Go 程序时的指南。

Hello, World

1.1. Hello, World

我们以 1978 年出版的 C 语言圣经《The C Programming Language》中经典的 “hello world” 案例来开 始吧（译注：本书作者之一 Brian W. Kernighan 也是 C 语言圣经一书的作者）。C 语言对 Go 语言的设计産 生了很多影响。用这个例子，我们来讲解一些 Go 语言的核心特性：

```
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和其依赖一起打包，生成机器的本地指令(译注：静态编译)。Go 语言提供的工具可以通过 go 命令下的一系列子命令来调用。最简单的一个子命令就是 run。这个命 令会将一个或多个文件名以.go 结尾的源文件，和关联库链接到一起，然后运行最终的可执行文件。（本书 将用$表示命令行的提示符。）

```
$ go run helloworld.go
```

毫无意外，这个命令会输出：

```
Hello, 世界
```

Go 语言原生支持 Unicode 标准，所以你可以用 Go 语言处理世界上的任何自然语言。

如果你希望自己的程序不只是简单的一次性实验，那麼你一定会希望能够编译这个程序，并且能够将编译结果保存下来以备将来之用。这个可以用 build 子命令来实现：

```
$ go build helloworld.go
```

这会创建一个名爲 helloworld 的可执行的二进制文件（译注：在 Windows 系统下生成的可执行文件是 helloworld.exe，增加了.exe 后缀名），之后你可以在任何时间去运行这个二进制文件，不需要其它的任 何处理(译注：因爲是静态编译，所以也不用担心在系统库更新的时候冲突，幸福感满满)。

下面是运行我们的编译结果样例（译注：在 Windows 系统下在命令行直接输入 helloworld.exe 命令运 行）：

```
$ ./helloworld
Hello, 世界
```

本书中我们所有的例子都做了一个特殊标记，你可以通过这些标记在 http://gopl.io 在线网站上找到这些样例代码，比如这个

```
gopl.io/ch1/helloworld
```

如果你执行 `go get gopl.io/ch1/helloworld` 命令，go 命令能够自己从网上获取到这些代码（译注：需要先安装 Git 或 Hg 之类的版本管理工具，并将对应的命令添加到 PATH 环境变量中），并且将这些代码放到 对应的目録中（译注：序言已经提及，需要先设置好 GOPATH 环境变量，下载的代码会放在 $GOPATH/src/gopl.io/ch1/helloworld 目録）。更详细的介绍在 2.6 和 10.7 章节中。

我们来讨论一下程序本身。Go 语言的代码是通过 package 来组织的，package 的概念和你知道的其它语言 里的 libraries 或者 modules 概念比较类似。一个 package 会包含一个或多个.go 结束的源代码文件。每一个 源文件都是以一个 package xxx 的声明语句开头的，比如我们的例子里就是 package main。这行声明语句 表示该文件是属于哪一个 package，紧跟着是一系列 import 的 package 名，表示这个文件中引入的 package。再之后是本文件本身的代码。

Go 的标准库已经提供了 100 多个 package，用来完成一门程序语言的一些常见的基本任务，比如输入、输 出、排序或者字符串/文本处理。比如 fmt 这个 package，就包括接收输入、格式化输出的各种函数。Println 是其中的一个常用的函数，可以用这个函数来打印一个或多个值，该函数会将这些参数用空格隔开 进行输出，并在输出完毕之后在行末加上一个换行符。

package main 是一个比较特殊的 package。这个 package 里会定义一个独立的程序，这个程序是可以运行 的，而不是像其它 package 一样对应一个 library。在 main 这个 package 里，main 函数也是一个特殊的函

数，这是我们整个程序的入口（译注：其实 C 系语言差不多都是这样）。main 函数所做的事情就是我们程 序做的事情。当然了，main 函数一般是通过是调用其它 packge 里的函数来完成自己的工作，比如 fmt.Println。

我们必须告诉编译器如何要正确地执行这个源文件，需要用到哪些 package，这就是 import 在这个文件里 扮演的角色。上述的 hello world 例子只用到了一个其它的 package，就是 fmt。一般情况下，需要 import 的 package 可能不只一个。

这也正是因爲 go 语言必须引入所有要用到的 package 的原则，假如你没有在代码里 import 需要用到的 package，程序将无法编译通过，同时当你 import 了没有用到的 package，也会无法编译通过（译注：Go 语言编译过程没有警告信息，争议特性之一）。

import 声明必须跟在文件的 package 声明之后。在 import 语句之后，则是各种方法、变量、常量、类型的 声明语句(分别用关键字 func, var, const, type 来进行定义)。这些内容的声明顺序并没有什麽规定，可以随 便调整顺序(译注：最好还是定一下规范)。我们例子里的程序比较简单，只包含了一个函数。并且在该函数 里也只调用了一个其它函数。爲了节省空间，有些时候的例子我们会省略 package 和 import 声明，但是读 者需要注意这些声明是一定要包含在源文件里的。

一个函数的声明包含 func 这个关键字、函数名、参数列表、返回结果列表（我们例子里的 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。关于函数的更详细描述在第五章。

Go 语言是一门不需要分号作爲语句或者声明结束的语言，除非要在一行中将多个语句、声明隔开。然而在 编译时，编译器会主动在一些特定的符号（译注：比如行末是，一个标识符、一个整数、浮点数、虚数、 字符或字符串文字、关键字 break、continue、fallthrough 或 return 中的一个、运算符和分隔符++、--、
)、]或}中的一个） 后添加分号，所以在哪里加分号合适是取决于 Go 语言代码的。例如：在 Go 语言中的函 数声明和 { 大括号必须在同一行，而在 x + y 这样的表达式中，在+号后换行可以，但是在+号前换行则会 有问题（译注：以+结尾的话不会被插入分号分隔符，但是以 x 结尾的话则会被分号分隔符，从而导致编译 错误）。

Go 语言在代码格式上采取了很强硬的态度。gofmt 工具会将你的代码格式化爲标准格式（译注：这个格式 化工具没有任何可以调整代码格式的参数，Go 语言就是这麽任性），并且 go 工具中的 fmt 子命令会自动对 特定 package 下的所有.go 源文件应用 gofmt 工具格式化。如果不指定 package，则默认对当前目録下的源 文件进行格式化。本书中的所有代码已经是执行过 gofmt 后的标准格式代码。你应该在自己的代码上也执 行这种格式化。规定一种标准的代码格式可以规避掉无尽的无意义的撕逼（译注：也导致了 Go 语言的 TIOBE 排名较低，因爲缺少撕逼的话题）。当然了，这可以避免由于代码格式导致的逻辑上的歧义。

很多文本编辑器都可以设置爲保存文件时自动执行 gofmt，所以你的源代码应该总是会被格式化。这里还有一个相关的工具，goimports，会自动地添加你代码里需要用到的 import 声明以及需要移除的 import 声 明。这个工具并没有包含在标准的分发包中，然而你可以自行安装：

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

对于大多数用户来说，下载、build package、运行测试用例、显示 Go 语言的文档等等常用功能都是可以

用 go 的工具来实现的。这些工具的详细介绍我们会在 10.7 节中提到。

命令行参数

1.2. 命令行参数

大多数的程序都是处理输入，産生输出；这也正是“计算”的定义。但是一个程序要如何获取输入呢？一些程序会生成自己的数据，但通常情况下，输入都来自于程序外部：比如文件、网络连接、其它程序的输出、用户的键盘、命令行的参数或其它类似输入源。下面几个例子会讨论其中的一些输入类型，首先是命令行参数。

os 这个 package 提供了操作系统无关（跨平台）的，与系统交互的一些函数和相关的变量，运行时程序的 命令行参数可以通过 os 包中一个叫 Args 的这个变量来获取；当在 os 包外部使用该变量时，需要用 os.Args 来访问。

os.Args 这个变量是一个字符串（string）的 slice（译注：slice 和 Python 语言中的切片类似，是一个简版 的动态数组），slice 在 Go 语言里是一个基础的数据结构，之后我们很快会提到。现在可以先把 slice 当一个 简单的元素序列，可以用类似 s[i]的下标访问形式获取其内容，并且可以用形如 s[m:n]的形式来获取到一个 slice 的子集(译注：和 python 里的语法差不多)。其长度可以用 len(s)函数来获取。和其它大多数编程语言 类似，Go 语言里的这种索引形式也采用了左闭右开区间，包括 m~n 的第一个元素，但不包括最后那个元 素（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）。这样可以简化我们的处理逻 辑。比如 s[m:n]这个 slice，0 ≤ m ≤ n ≤ len(s)，包含 n-m 个元素。

os.Args 的第一个元素，即 os.Args[0]是命令行执行时的命令本身；其它的元素则是执行该命令时传给这个 程序的参数。前面提到的切片表达式，s[m:n]会返回第 m 到第 n-1 个元素，所以下一个例子里需要用到的 os.Args[1:len(os.Args)]即是除了命令本身外的所有传入参数。如果我们省略 s[m:n]里的 m 和 n，那麽默认 这个表达式会填入 0:len(s)，所以这里我们还可以省略掉 n，写成 os.Args[1:]。

下面是一个 Unix 里 echo 命令的实现，这个命令会在单行内打印出命令行参数。这个程序 import 了两个 package，并且用括号把这两个 package 包了起来，这是分别 import 各个 package 声明的简化写法。当然 了你分开来写 import 也没有什麽问题，只是一般爲了方便我们都会像下面这样来导入多个 package 。我们 自己写的导入顺序并不重要，因爲 gofmt 工具会帮助我们按照字母顺序来排列好这些导入包名。（本书中 如果一个例子有多种版本时，我们会用编号标记出来）

```
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)
}
```

Go 语言里的注释是以//来表示。//之后的内容一直到行末都是这条注释的一部分，并且这些注释会被编译器忽略。

按照惯例，我们会在每一个 package 前面放上这个 package 的详尽的注释对其进行帮助；对于一个 main package 来说，一般这段评论会包含几句话来帮助这个项目/程序整体是做什麽用的。

var 关键字用来做变量声明。这个程序声明了 s 和 sep 两个 string 变量。变量可以在声明期间直接进行初始 化。如果没有显式地初始化的话，Go 语言会隐式地给这些未初始化的变量赋予对应其类型的零值，比如数 值类型就是 0，字符串类型就是空字符串 “” 。在这个例子里的 s 和 sep 被隐式地赋值爲了空字符串。在第 2 章中我们会更详细地讲解变量和声明。

对于数字类型，Go 语言提供了常规的数值计算和逻辑运算符。而对于 string 类型，+号表示字符串的连接（译注：和 C++或者 js 是一样的）。所以下面这个表达式：

```
sep + os.Args[i]
```

表示将 sep 字符串和 os.Args[i]字符串进行连接。我们在程序里用的另外一个表达式：

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

会将 sep 与 os.Args[i]连接，然后再将得到的结果与 s 进行连接并赋值运给 s，这种方式和下面的表达是等价
的：

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

运算符+=是一个赋值运算符(assignment operator)，每一种数值和逻辑运算符，例如*或者+都有其对应的赋值运算符。

echo 程序可以每循环一次输出一个参数，不过我们这里的版本是不断地将其结果连接到一个字符串的末尾。s 这个字符串在声明的时候是一个空字符串，而之后循环每次都会被在末尾添加一段字符串；第一次迭代之后，一个空格会被插入到字符串末尾，所以每插入一个新值，都会和前一个中间有一个空格隔开。这是一种非线性的操作，当我们的参数数量变得庞大的时候（当然不是说这里的 echo，一般 echo 也不会有太 多参数）其运行开销也会变得庞大。下面我们会介绍一系列的 echo 改进版，来应对这里说到的运行效率低 下。

在 for 循环中，我们用到了 i 来做下标索引，可以看到我们用了:=符号来给 i 进行初始化和赋值，这是 var xxx=yyy 的一种简写形式，Go 语言会根据等号右边的值的类型自动判断左边的值类型，下一章会对这一点 进行详细帮助。

自增表达式 i++会爲 i 加上 1；这和 i += 1 以及 i = i + 1 都是等价的。对应的还有 i--是给 i 减去 1。这些在 Go 语 言里是语句，而不像 C 系的其它语言里是表达式。所以在 Go 语言里 j = i++是非法的，而且++和--都只能 放在变量名后面，因此--i 也是非法的。

在 Go 语言里只有 for 循环一种循环。当然了爲了满足需求，Go 的 for 循环有很多种形式，下面是其中的一 种：

```
for initialization; condition; post {
    // zero or more statements
}
```

这里需要注意，for 循环的两边是不需要像其它语言一样写括号的。并且左大括号需要和 for 语句在同一行。

initialization 部分是可选的，如果你写了这部分的话，在 for 循环之前这部分的逻辑会被执行。需要注意的 是这部分必须是一个简单的语句，也就是说是一个简短的变量声明，一个赋值语句，或是一个函数调用。 condition 部分必须是一个结果爲 boolean 值的表达式，在每次循环之前，语言都会检查当前是否满足这个 条件，如果不满足的话便会结束循环；post 部分的语句则是在每次循环迭代结束之后被执行，之后 conditon 部分会在下一次执行前再被执行，依此往复。当 condition 条件里的判断结果变爲 false 之后，循 环即结束。

上面提到是 for 循环里的三个部分都是可以被省略的，如果你把 initialization 和 post 部分都省略的话，那麼 连中间隔离他们的分号也是可以被省略的，比如下面这种 for 循环，就和传统的 while 循环是一样的：

```
// a traditional "while" loop
for condition {
    // ...
}
```

当然了，如果你连唯一的条件都省了，那麼 for 循环就会变成一个无限循环，像下面这样：

```
// a traditional infinite loop
for {
    // ...
}
```

在无限循环中，你还是可以靠 break 或者 return 语句来终止掉循环。

如果你的遍历对象是 string 或者 slice 类型值的话，还有另外一种循环的写法，我们来看看另一个版本的 echo：

```
gopl.io/ch1/echo2
// Echo2 prints its command-line arguments.
package main

import (
    "fmt"
)

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

每一次循环迭代，range 都会返回一对结果；当前迭代的下标以及在该下标处的元素的值。在这个例子里，我们不需要这个下标，但是因爲 range 的处理要求我们必须要同时处理下标和值。我们可以在这里声 明一个接收 index 的临时变量来解决这个问题，但是 Go 语言又不允许只声明而在后续代码里不使用这个变 量，如果你这样做了编译器会返回一个编译错误。

在 Go 语言中，应对这种情况的解决方法是用空白标识符，对，就是上面那个下划线_。空白标识符可以在 任何你接收自己不需要处理的值时使用。在这里，我们用它来忽略掉 range 返回的那个没用的下标值。大 多数的 Go 程序员都会像上面这样来写类似的 os.Args 遍历，由于遍历 os.Args 的下标索引是隐式自动生成 的，可以避免因显式更新索引导致的错误。

上面这个版本将 s 和 sep 的声明和初始化都放到了一起，但是我们可以等价地将声明和赋值分开来写，下面 这些写法都是等价的

```
s := ""
var s string
var s = ""
var s string = ""
```

那麼这些等价的形式应该怎麼做选择呢？这里提供一些建议：第一种形式，只能用在一个函数内部，而

package 级别的变量，禁止用这样的声明方式。第二种形式依赖于 string 类型的内部初始化机制，被初始 化爲空字符串。第三种形式使用得很少，除非同时声明多个变量。第四种形式会显式地标明变量的类型， 在多变量同时声明时可以用到。实践中你应该只使用上面的前两种形式，显式地指定变量的类型，让编译 器自己去初始化其值，或者直接用隐式初始化，表明初始值怎麽样并不重要。

像上面提到的，每次循环迭代中字符串 s 都会得到一个新内容。+=语句会分配一个新的字符串，并将老字符串连接起来的值赋予给它。而目标字符串的老字面值在得到新值以后就失去了用处，这些临时值会被 Go 语言的垃圾收集器躲掉。

如果不断连接的数据量很大，那麽上面这种操作就是成本非常高的操作。更简单并且有效的一种方式是使用 strings 包提供的 Join 函数，像下面这样：

```
gopl.io/ch1/echo3
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

最后，如果我们对输出的格式也不是很关心，只是想简单地输出值得的话，还可以像下面这麽写，Println 函数会爲我们自动格式化输出。

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

这个输出结果和前面的 string.Join 得到的结果很相似，只是被自动地放到了一个方括号里，对 slice 调用 Println 函数都会被打印成这样形式的结果。

练习 1.1： 修改 echo 程序，使其能够打印 os.Args[0]。

练习 1.2： 修改 echo 程序，使其打印 value 和 index，每个 value 和 index 显示一行

。 练习 1.3： 上手实践前面提到的 strings.Join 和直接 Println，并观察输出结果的区

别。

查找重复的行

1.3. 查找重复的行

文件拷贝、文件打印、文件搜索、文件排序、文件统计类的程序一般都会有比较相似的程序结构：一个处理输入的循环，在每一个输入元素上执行计算处理，在处理的同时或者处理完成之后进行结果输出。我们会展示一个叫 dup 程序的三个版本；这个程序的灵感来自于 linux 的 uniq 命令，我们的程序将会找到相邻的 重复的行。这个程序提供的模式可以很方便地被修改来完成不同的需求。

第一个版本的 dup 会输出标准输入流中的出现多次的行，在行内容前会有其出现次数的计数。这个程序将 引入 if 表达式，map 内置数据结构和 bufio 的 package。

```
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)
        }
    }
}
```

和我们前面提到的 for 循环一样，在 if 条件的两边，我们也不需要加括号，但是 if 表达式后的逻辑体的花括号
是不能省略的。如果需要的话，像其它编程语言一样，这个 if 表达式也可以有 else 部分，这部分逻辑会在 if 中的条件结果爲 false 时被执行。

map 是 Go 语言内置的 key/value 型数据结构，这个数据结构能够提供常数时间的存储、获取、测试操作。 key 可以是任意数据类型，只要该类型能够用==运算符来进行比较，string 是最常用的 key 类型。而 value 类型的范围就更大了，基本上什麽类型都是可以的。这个例子中的 key 都是 string 类型，value 用的是 int 类 型。我们用内置 make 函数来创建一个空的 map，当然了，make 方法还可以有别的用处。在 4.3 章中我们 还会对 map 进行更深入的讨论。

dup 程序每次读取输入的一行，这一行的内容会被当做一个 map 的 key，而其 value 值会被+1。 counts[input.Text()]++这个语句和下面的两句是等价的：

```
line := input.Text()
counts[line] = counts[line] + 1
```

当然了，在这个例子里我们并不用担心 map 在没有当前的 key 时就对其进行++操作会有什麽问题，因爲 Go 语言在碰到这种情况时，会自动将其初始化爲 0，然后再进行操作。

在这里我们又用了一个 range 的循环来打印结果，这次 range 是被用在 map 这个数据结构之上。这一次的情 况和上次比较类似，range 会返回两个值，一个 key 和在 map 对应这个 key 的 value。对 map 进行 range 循环 时，其迭代顺序是不确定的，从实践来看，很可能每次运行都会有不一样的结果（译注：这是

计者有意爲之的，因爲其底层实现不保证插入顺序和遍历顺序一致，也希望程序员不要依赖遍历时的顺 序，所以鞅脆直接在遍历的时候做了随机化处理，醉了。补充：好像说随机序可以防止某种类型的攻击， 虽然不太明白，但是感觉还蛮厉害的），来避免程序员在业务中依赖遍历时的顺序。

然后轮到我们例子中的 bufio 这个 package 了，这个 package 主要的目的是帮助我们更方便有效地处理程序 的输入和输出。而这个包最有用的一个特性就是其中的一个 Scanner 类型，用它可以简单地接收输入，或 者把输入打散成行或者单词；这个类型通常是处理行形式的输入最简单的方法了。

本程序中用了一个短变量声明，来创建一个 buffio.Scanner 对象：

```
input := bufio.NewScanner(os.Stdin)
```

scanner 对象可以从程序的标准输入中读取内容。对 input.Scanner 的每一次调用都会调入一个新行，并且
会自动将其行末的换行符去掉；其结果可以用 input.Text()得到。Scan 方法在读到了新行的时候会返回 true，而在没有新行被读入时，会返回 false。

例子中还有一个 fmt.Printf，这个函数和 C 系的其它语言里的那个 printf 函数差不多，都是格式化输出的方 法。fmt.Printf 的第一个参数卽是输出内容的格式规约，每一个参数如何格式化是取决于在格式化字符串里 出现的“转换字符”，这个字符串是跟着%号后的一个字母。比如%d 表示以一个整数的形式来打印一个变 量，而%s，则表示以 string 形式来打印一个变量。

Printf 有一大堆这种转换，Go 语言程序员把这些叫做 verb（动词）。下面的表格列出了常用的动词，当然 了不是全部，但基本也够用了。

```
%d          int 變
%x, %o, %b  分別爲 16 進製，8 進製，2 進製形式的 int
%f, %g, %e  浮點數： 3.141593 3.141592653589793 3.141593e+00
%t          布爾變量：true 或 false
%c          rune (Unicode 碼點)，Go 語言里特有的 Unicode 字符類型
%s          string
%q          帶雙引號的字符串 "abc" 或 帶單引號的 rune 'c'
%v          會將任意變量以易讀的形式打印出來
%T          打印變量的類型
%%          字符型百分比標誌（%符號本身，沒有其他操作）
```

dup1 中的程序还包含了一个\t 和\n 的格式化字符串。在字符串中会以这些特殊的转义字符来表示不可见字
符。Printf 默认不会在输出内容后加上换行符。按照惯例，用来格式化的函数都会在末尾以 f 字母结尾（译 注：f 后缀对应 format 或 fmt 缩写），比如 log.Printf，fmt.Errorf，同时还有一系列对应以 ln 结尾的函数
（译注：ln 后缀对应 line 缩写），这些函数默认以%v 来格式化他们的参数，并且会在输出结束后在最后自 动加上一个换行符。

许多程序从标准输入中读取数据，像上面的例子那样。除此之外，还可能从一系列的文件中读取。下一个 dup 程序就是从标准输入中读到一些文件名，用 os.Open 函数来打开每一个文件获取内容的。

```
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 函数会返回两个值。第一个值是一个打开的文件类型(*os.File)，这个对象在下面的程序中被 Scanner 读取。

os.Open 返回的第二个值是一个 Go 语言内置的 error 类型。如果这个 error 和内置值的 nil（译注：相当于其 它语言里的 NULL）相等的话，帮助文件被成功的打开了。之后文件被读取，一直到文件的最后，文件的 Close 方法关闭该文件，并释放相应的占用一切资源。另一方面，如果 err 的值不是 nil 的话，那帮助在打开 文件的时候出了某种错误。这种情况下，error 类型的值会描述具体的问题。我们例子里的简单错误处理会 在标准错误流中用 Fprintf 和%v 来格式化该错误字符串。然后继续处理下一个文件；continue 语句会直接

跳过之后的语句，直接开始执行下一个循环迭代。

我们在本书中早期的例子中做了比较详尽的错误处理，当然了，在实际编码过程中，像 os.Open 这类的函数是一定要检查其返回的 error 值的；爲了减少例子程序的代码量，我们姑且简化掉这些不太可能返回错误 的处理逻辑。后面的例子里我们会跳过错误检查。在 5.4 节中我们会对错误处理做更详细的阐述。

读者可以再观察一下上面的例子，我们的 countLines 函数是在其声明之前就被调用了。在 Go 语言里，函数 和包级别的变量可以以任意的顺序被声明，并不影响其被调用。（译注：最好还是遵循一定的规范）

再来讲讲 map 这个数据结构，map 是用 make 函数创建的数据结构的一个引用。当一个 map 被作爲参数传 递给一个函数时，函数接收到的是一份引用的拷贝，虽然本身并不是一个东西，但因爲他们指向的是同一 块数据对象（译注：类似于 C++里的引用传递），所以你在函数里对 map 里的值进行脩改时，原始的 map 内的值也会改变。在我们的例子中，我们在 countLines 函数中插入到 counts 这个 map 里的值，在主函数中 也是看得到的。

上面这个版本的 dup 是以流的形式来处理输入，并将其打散爲行。理论上这些程序也是可以以二进制形式来处理输入的。我们也可以一次性的把整个输入内容全部读到内存中，然后再把其分割爲多行，然后再去处理这些行内的数据。下面的 dup3 这个例子就是以这种形式来进行操作的。这个例子引入了一个新函数 ReadFile（从 io/ioutil 包提供），这个函数会把一个指定名字的文件内容一次性调入，之后我们用 strings.Split 函数把文件分割爲多个子字符串，并存储到 slice 结构中。（Split 函数是 strings.Join 的逆函数，Join 函数之前提到过）

我们简化了 dup3 这个程序。首先，它只读取命名的文件，而不去读标准输入，因爲 ReadFile 函数需要一个 文件名参数。其次，我们将行计数逻辑移回到了 main 函数，因爲现在这个逻辑只有一个地方需要用到。

```
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 byte 的 slice，这个 slice 必须被转换爲 string，之后才能够用 string.Split 方法来进行
处理。我们在 3.5.4 节中会更详细地讲解 string 和 byte slice（字节数组）。

在更底层一些的地方，bufio.Scanner，ioutil.ReadFile 和 ioutil.WriteFile 使用的是*os.File 的 Read 和 Write 方法，不过一般程序员并不需要去直接了解到其底层实现细节，在 bufio 和 io/ioutil 包中提供的方法已经足 够好用。

练习 1.4： 脩改 dup2，使其可以打印重复的行分别出现在哪些文件。

GIF 动畫

1.4. GIF 动畫

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爲一个 GIF 动畫。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産 生的曲线，比如两个 sin 正弦波分别在 x 轴和 y 轴输入会産生的曲线。图 1.1 是这样的一个例子：

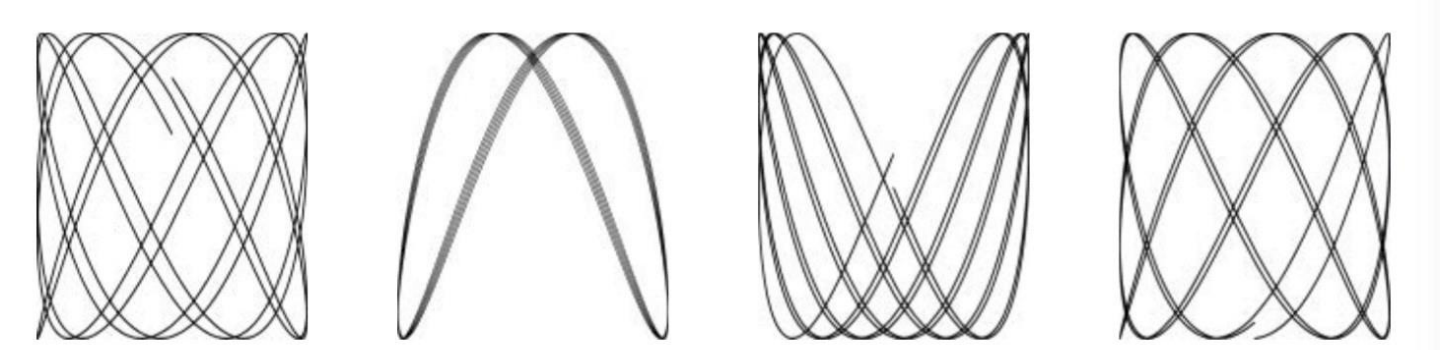
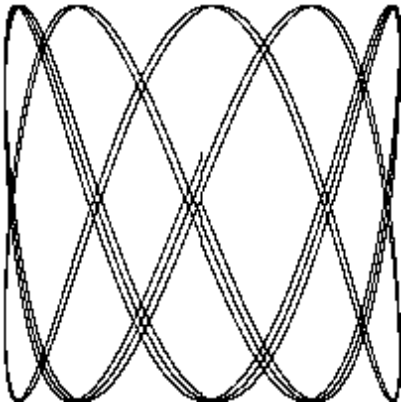


Figure 1.1. Four Lissajous figures.

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 图像动畫效果：



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帮助一下，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),
bla    kIndex)
        }
        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，常量是指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，那麼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 节有帮助）。这是实例化 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 字段会被设置爲 nframes；而其它的字段会被设置爲各自类型默 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畫帧。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爲其零值（也就 是 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 置爲黑色。其结果会 append 到之前结果之后。这里我们用到了 append(参考 4.2.1)这个内置函数，将结果 appen 到 anim 中的帧列表末尾，并会设置一个默认的 80ms 的延迟值。最终循环结束，所有的延迟值也被 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，这个类型让我们可以可以让 我们把输出结果写到很多目标，很快我们就可以看到了。

内存循环设置了两个偏振。x 轴偏振使用的是一个 sin 函数。y 轴偏振也是一个正弦波，但是其其相对 x 轴的 偏振是一个 0-3 的随机值，并且初始偏振值是一个零值，并随着动畫的每一帧逐渐增加。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 来爲(x, y)点来染黑色。

main 函数调用了 lissajous 函数，并且用它来向标准输出中打印信息，所以下面这个命令会像图 1.1 中産生 一个 GIF 动畫。

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

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

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

获取 URL

1.5. 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 来做这件事情，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）就显得尤其好用了。

爲了最简单地展示基于 HTTP 获取信息的方式，下面给出一个示例程序 fetch，这个程序将获取对应的 url， 并将其源文本打印出来；这个例子的灵感来源于 curl 工具（译注：unix 下的一个网络相关的工具）。当然 了，curl 提供的功能更爲复杂丰富，这里我们只编写最简单的样例。之后我们还会在本书中经常用到这个 例子。

```
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，那麽会在 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。这之后 ioutil.ReadAll 函数从 response 中读取到全部内容；其结果保存在变量 b 中。 resp.Body.Close 这一句会关闭 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 Programming Language</title>title>
...
```

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 is known.
```

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，其值爲 1。

练习 1.7： 函数调用 io.Copy(dst, src)会从 src 中读取内容，并将读到的结果写入到 dst 中，使用这个函数替 代掉例子中的 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： 脩改 fetch 这个范例，如果输入的 url 参数没 `http://` 前缀的话，爲这个 url 加上该前缀。你可有
能会用到 strings.HasPrefix 这个函数。

练习 1.9： 脩改 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

1.6. 并发获取多个 URL

Go 语言最有意思并且最新奇的特性就是其对并发编程的支持了。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，和上面的 fetch 程序所要做的工作是一致的，但是这个 fetchall 的特别之处在于它会同 时去获取所有的 URL，所以这个程序的获取时间不会超过执行时间最长的那一个任务，而不会像前面的 fetch 程序一样，执行时间是所有任务执行时间之和。这次的 fetchall 程序只会打印获取的内容大小和经过 的时间，不会像上面那样打印出获取的内容。

```
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)
}
```

下面是一个使用的例子

```
$ 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 函数也是 运行在一个 goroutine 中，而 go function 则表示创建一个新的 goroutine，并在这个这个新的 goroutine 里执行这个函数。

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 会把响应的 Body 内容拷贝到 ioutil.Discard 输出流中（译注：这是一个垃圾桶，可以向里面写一些 不需要的数据），因爲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 ch 这个 channel 里写入一个字符串，由 main 函数里的第二个 for 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、或者接收了值，这样两个 goroutine 才会继续执行操作 channel 完成之后的逻辑。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)，主函数接收这些值(<-ch)。这个程序中我们用 main 函数来所有 fetch 函数传回的字符串，可 以避免在 goroutine 异步执行时同时结束。

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。

Web 服务

1.7. Web 服务

Go 语言的内置库让我们写一个像 fetch 这样例子的 web 服务器变得异常地简单。在本节中，我们会展示一 个微型服务器，这个服务的功能是返回当前用户正在访问的 URL。也就是说比如用户访问的是 http://localhost:8000/hello ，那麼响应是 URL.Path = "hello"。

```
gopl.io/ch1/server1
// Server1 is a minimal "echo" server.
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // each request calls handler
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// handler echoes the Path component of the request URL r.
func handler(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个 Web 服务程序，这都是多亏了标准库里的方法已经帮我们处理了大量的工作。main 函数会将所有发送到/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，我们的服务跑在了 8000 端口上。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，这个函数会将/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 详细阐述。

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，那麼在运行命令的末尾加上 一个&符号，即可让程序简单地跑在后台，而在 windows 下，你需要在另外一个命令行窗口去运行这个程 序了。

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

现在我们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"
```

另外我们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：

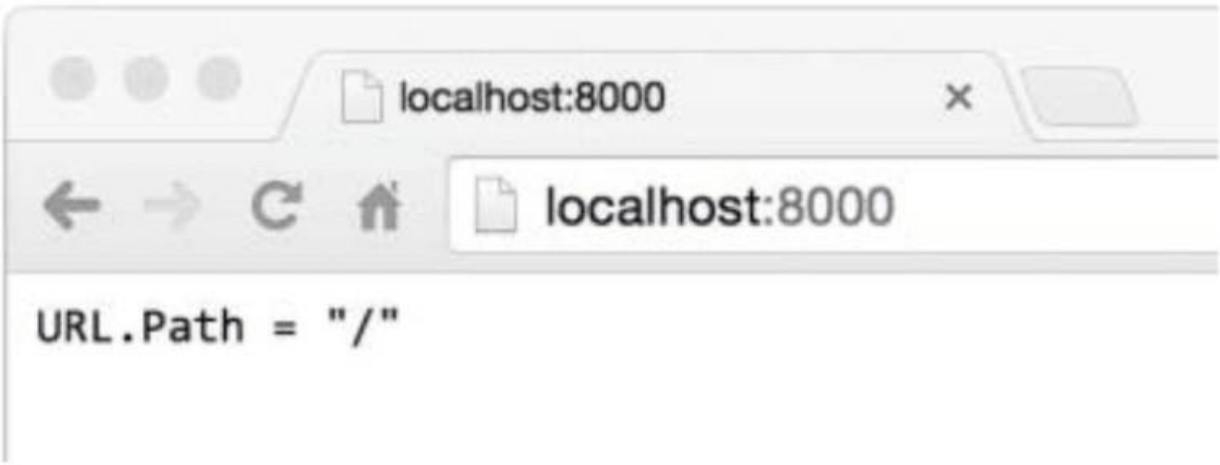


Figure 1.2. A response from the echo server.

在这个服务的基础上叠加特性是很容易的。一种比较实用的脩改是爲访问的 url 添加某种状态。比如，下面
这个版本输出了同样的内容，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，请求的 url 会决定具体调用哪一个：对/count 这个 url 的请求会调用到 count 这个函数，其它所有的 url 都会调用默认的处理函数。如果你的请求 pattern 是以/结尾，那麽所有以 该 url 爲前缀的 url 都会被这条规则匹配。在这些代码的背后，服务器每一次接收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数请求。然而在并发情况下，假如眞的有两个请求同一时 刻去更新 count，那麽这个值可能并不会被正确地增加；这个程序可能会被引发一个严重的 bug：竞态条件
（参见 9.1）。爲了避免这个问题，我们必须保证每次脩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()调用将脩改 count 的所有行爲包在中间的目的。第九章中我们会进一步讲 解共享变量。

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

```
gopl.io/ch1/server3
// 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, "Header[%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的语句结果作爲循环的变量
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 {
    log.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，这麼做是很不错的。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，就把 HTTP 的 响应数据拷贝到了 os.Stdout，或者在 lissajous 程序里我们输出的是一个文件。fetchall 程序则完全忽略到 了 HTTP 的响应体，只是计算了一下响应体的大小，这个程序中把响应体拷贝到了 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。

尽管这三种具体的实现流程并不太一样，他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。这个接口叫作 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，爲了在这里简单帮助接口能做什麽，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 动畫可以被写到 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.HandleFunc("/", 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。

做完这些脩改之后，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 那样的动畫。

练习 1.12： 脩改 Lissajour 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 脩改爲 20。字符串转换爲数字可以调用 strconv.Atoi 函 数。你可以在 dodoc 里查看 strconv.Atoi 的详细帮助。

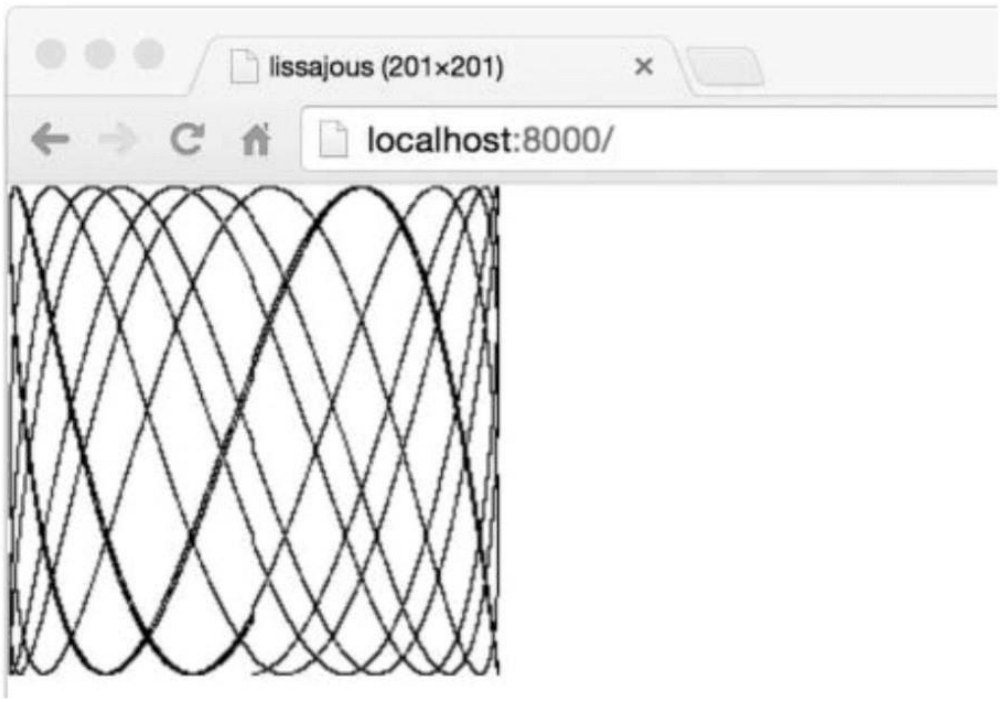


Figure 1.3. Animated Lissajous figures in a browser.

本章要点

1.8. 本章要点

本章中对 Go 语言做了一些介绍，实际上 Go 语言还有很多方面在这有限的篇幅中还没有覆盖到。这里我们 会把没有讲到的内容也做一些简单的介绍，这样读者在之后看到完整的内容之前，也可以有个简单印象。

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 都会对应个返回结果，这里需
要注意，Go 语言并不需要显式地去在每一个 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 来覆盖这种默认行爲。不过 fallthrough 语句在一般的编程中用到得很少。

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，像其它语言里面的多个 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。

break 和 continue 语句会改变控制流。和其它语言中的 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 会中跳过当前循环，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 是内层的循环，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。这两种行爲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地给一个特殊类型一个名字。因爲 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，指针操作是完全不受约束的。在另外一些语言中，指针一般被处理爲“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 中进行详细介绍。

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。第七章中会详细帮助这些内容。

包（packages）： 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，但是还是有很多我们没有篇幅去帮助，因 爲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閱读标准库的文档。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爲的注释也是一个好习惯。这些惯例很重要，因爲这些内容会被像 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，和其它大多数语言一样。在文件一开头的注释一般都是这种形式，或者一大段的解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/*是没什麽意义的，所以不要在注释中再嵌入注释。

程序结构

第 2 章 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。基础类型被聚合爲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榦 扰，从而可以专注于 Go 语言本身的学习。

命名

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 内建類

型: int int8 int16 int32 int64
        uint uint8 uint16 uint32 uint64 uintptr
        float32 float64 complex128 complex64
        bool byte rune string error

内建函數: make len cap new append copy close delete
        complex real imag
        panic recover
```

这些内部预先定义的名字并不是关键字，你可以再定义中重新使用它们。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，那麼它的就只在函数内部有效。如果是在函数外部定义，那麼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。如果一个名字是大写字母开头的（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），那麼它将是导出的，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，那麼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 驼峯式 命名，当名字有几个单词组成的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 这样的函数命名，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，它们可能被称爲 htmlEscape、HTMLEscape 或 escapeHTML，但不会是 escapeHtml。

声明

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声明语句：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 爲文件后缀名的源文件中。每个源文件以包的声明语句开始，帮助该源文件是属于哪个包。包声明语句之后是 import 语句导入依赖的其它包，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°F or %g°C\n", f, c)
    // Output:
    // boiling point = 212°F or 100°C
}
```

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范围被访问。

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，那麽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 renturn 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 函数就调用了两次 fToC 函数，分别是使用在局部定义的两个常量作爲调用函数 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°F = %g°C\n", freezingF, fToC(freezingF)) // "32°F = 0°C"
    fmt.Printf("%g°F = %g°C\n", boilingF, fToC(boilingF))   // "212°F = 100°C"
}

func fToC(f float64) float64 {
    return (f - 32) * 5 / 9
}
```

变量

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 變量名字 類型 = 表達式
```

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，那麽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，那麽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声明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爲。例 如：

```
var s string
fmt.Println(s) // ""
```

这段代码将打印一个空字符串，而不是导致错误或産生不可预知的行爲。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 an error
```

{% include "./ch2-03-1.md" %}

{% include "./ch2-03-2.md" %}

{% include "./ch2-03-3.md" %}

{% include "./ch2-03-4.md" %}

赋值

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                       // 命令變量的賦值
*p = true                   // 通過指針間接賦值
person.name = "bob"         // 結構體字段賦值
count[x] = count[x] * scale // 數組、slice 或 map 的元素賦
```

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爲：

```
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++    // 等價方式 v = v + 1 ; v 變成 2
v--    // 等價方式 v = v - 1 ; v 變成 1
```

{% include "./ch2-04-1.md" %}

{% include "./ch2-04-2.md" %}

类型

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 類型名字 底層類型
```

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 标志都作爲小写字母处理，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 RobPik 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.

爲了帮助类型声明，我们将不同温度单位分别定义爲不同的类型：

```
gopl.io/ch2/tempconv0
// Package tempconv performs Celsius and Fahrenheit temperature computations.
package tempconv

import "fmt"

type Celsius float64    // 攝氏溫度
type Fahrenheit float64 // 華氏溫度

const (
    AbsoluteZeroC Celsius = -273.15 // 絕對零度
    FreezingC     Celsius = 0       // 結冰點溫度
    BoilingC      Celsius = 100     // 沸水溫度
)

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 转爲对应的类型。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 转爲 T 类型（译注：如果 T 是指针类型，可
能会需要用小括号包装 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，那麼 x 必然也可以被转爲 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。这类转换可能改变值的表现。例如，将一个浮点数转爲整数将丢弃小数部分，将一
个字符串转爲`[]byte`类型的 slice 将拷贝一个字符串数据的副本。在任何情况下，运行时不会发生转换失败
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 类型的算术运算行爲和底层的 float64 类型是一样的，正如我们所期望的那样。

```
fmt.Printf("%g\n", BoilingC-FreezingC) // "100" °C
boilingF := CToF(BoilingC)
fmt.Printf("%g\n", boilingF-CToF(FreezingC)) // "180" °F
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。尽管看起来想函数调用，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。测试爲眞的原因是因爲 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。

命名类型还可以爲该类型的值定义新的行爲。这些行爲表示爲一组关联到该类型的函数集合，我们称爲类型的方法集。我们将在第六章中讨论方法的细节，这里值说写简单用法。

下面的声明语句，Celsius 类型的参数 c 出现在了函数名的前面，表示声明的是 Celsius 类型的一个叫名叫

String 的方法，该方法返回该类型对象 c 带着°C 温度单位的字符串：

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

许多类型都会定义一个 String 方法，因爲当使用 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°C"
fmt.Printf("%v\n", c) // "100°C"; no need to call String explicitly
fmt.Printf("%s\n", c)   // "100°C"
fmt.Println(c)          // "100°C"
fmt.Printf("%g\n", c) // "100"; does not call String
fmt.Println(float64(c)) // "100"; does not call String
```

包和文件

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，目的都是爲了支持模块化、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 爲文件后缀名的源文件中，通常一个包所在目録路径的后缀是 包的导入路径；例如包 gopl.io/ch1/helloworld 对应的目録路径是 $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，那麽该名字是导出的（译注：因爲汉字不区分大小写，因此汉字开头的 名字是没有导出的）。

爲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？

让我们创建一个名爲 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 都是以章节顺序来编号的，这样的路径更容易閱读）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", c) }
func (f Fahrenheit) String() string { return fmt.Sprintf("%g°F", 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，因爲这个源文件中的代码并没有用到 fmt 包。

因爲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°C"
```

要将摄氏温度转换爲华氏温度，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°F"
```

在每个源文件的包声明前仅跟着的注释是包注释（§10.7.4）。通常，包注释的第一句应该先是包的功能概要帮助。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爲一个包注释）。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： 向 tempconv 包添加类型、常量和函数用来处理 Kelvin 絶对温度的转换，Kelvin 絶对零度是 −273.15°C，Kelvin 絶对温度 1K 和摄氏度 1°C 的单位间隔是一样的。

{% include "./ch2-06-1.md" %}

{% include "./ch2-06-2.md" %}

作用域

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爲一谈。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。

语法块是由花括号所包含的一系列语句，就像函数体或循环体花括号对应的语法块那样。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号包含的代码，我们称之爲语法块。有一个语法块爲整个源代码，称爲全局语法块；然后是每个包的包语法决；每个 for、if 和 switch 语句的语法决；每个 switch 或 select 的分支也有独立的语法决；当然也包括显式书写的语法块（花括号包含的语句）。

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，当前包的其它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。或者是像 2.3.3 节的例子那样，你可以将一个函数参数的名字声明爲 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，可能导致程序很难閱 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，因爲它们是定义在不同的词法域（这个例子只是爲了演 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号包含的语句；还有一些隐含的规则。上面的 for 语句创建了两个词法域：花括号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 语句也会在条件部分创建隐式词法域，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：条件部分爲一个隐式词法域，然后每个是每个分支的 词法域。

在包级别，声明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，则会産生编译错误。

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 f 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録然后保存到一个包级的变量中。这可以本来通过直接调用 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 重新声明爲新的局部变量。因爲内部声明的 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 并没有本使用，然后报告这可能是一个错误，但是这种检测并 不可靠。因爲一些小的代码变更，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 `:=` 的简短用
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

第 3 章 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 语言将数据类型分爲四类：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脩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。

整型

3.1. 整型

Go 语言的数值类型包括几种不同大小的整形数、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。让我们先从整形数类型开始介绍。

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

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，但是我们不能对此做任何的假设；因爲不同的编译器 即使在相同的硬件平台上可能産生不同的大小。

Unicode 字符 rune 类型是和 int32 等价的类型，通常用于表示一个 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 在底层编程是才需要，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的排列：

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

二元运算符有五种优先级。在同一个优先级，使用左优先结合规则，但是使用括号可以明确优先级，使用括号也可以用于提陞优先级，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`/`可以适用与于整数、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，%取模运算的行爲可能并不相同。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`结果都是-2。除法运算符`/`的行爲则依赖于操作数是否爲全爲整数，比如`5.0/4.0`的结果是 1.25，但是 5/4 的结果是 1，因爲整数除法会向着 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 爲是 1 的话，那麼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比较运算符进行比较；比较表达式的结果是布尔类型。

```
==    equal to
!=    not equal to
<     less than
<=    less 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，也就是说两个相同类型的值可以用==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。许多其它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 (無效果)
-    負數
```

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：

```
&    位運算 AND
|    位運算 OR
^    位運算 XOR
&^   位清空 (AND NOT)
<<   左移
>>   右移
```

位操作运算符 `^` 作爲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 位爲 0，如果对应 y 中 bit 位爲 1 的话，否则对应的 bit 位等于 x 相应的 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) // "01000100", the set {2, 6}
fmt.Printf("%08b\n", x>>1) // "00010001", the set {0, 4}
```

（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。因爲这个原因，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，那麼 i 也将是无符号的 uint
类型，然后条件 i >= 0 则永远爲眞。在三次迭代之后，也就是 i == 0 时，i--语句将不会産生-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它特殊的运算场景才会使用，就像 bit 集合、分析二进制文 件格式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位另一种类型，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，将産生一个错误信息：

```
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脩复，最常见方法是将它们都显式转型爲一个常见类型：

```
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 转换爲 T 类型。许多整形数之间

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数类型转爲一个小尺寸的整数类型，或者是将一个浮点数转爲整数，可能会改变数值或丢失精度：

```
f := 3.141 // a float64
i := int(f)
fmt.Println(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，然后向数轴零方向截断。你应该避免对可能会超出目标类型表示范围的数值类型转换，因爲截断的行爲可能依赖于具体的实现：

```
f := 1e100  // a float64
i := int(f) // 結果依賴於具體實現
```

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于 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\n", x)
// Output:
// 3735928559 deadbeef 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 `#` 副词告诉 Printf 在的
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，或者是用 `%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'"
```

浮点数

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爲 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，因爲 float32 类型的纍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：因爲 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 能表达的范围时，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 // 阿伏伽德羅常數
const Planck   = 6.62606957e-34 // 普朗尅常數
```

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，但是对应表格的
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 =  403.429
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，因爲 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(x*x*+*y*y)。

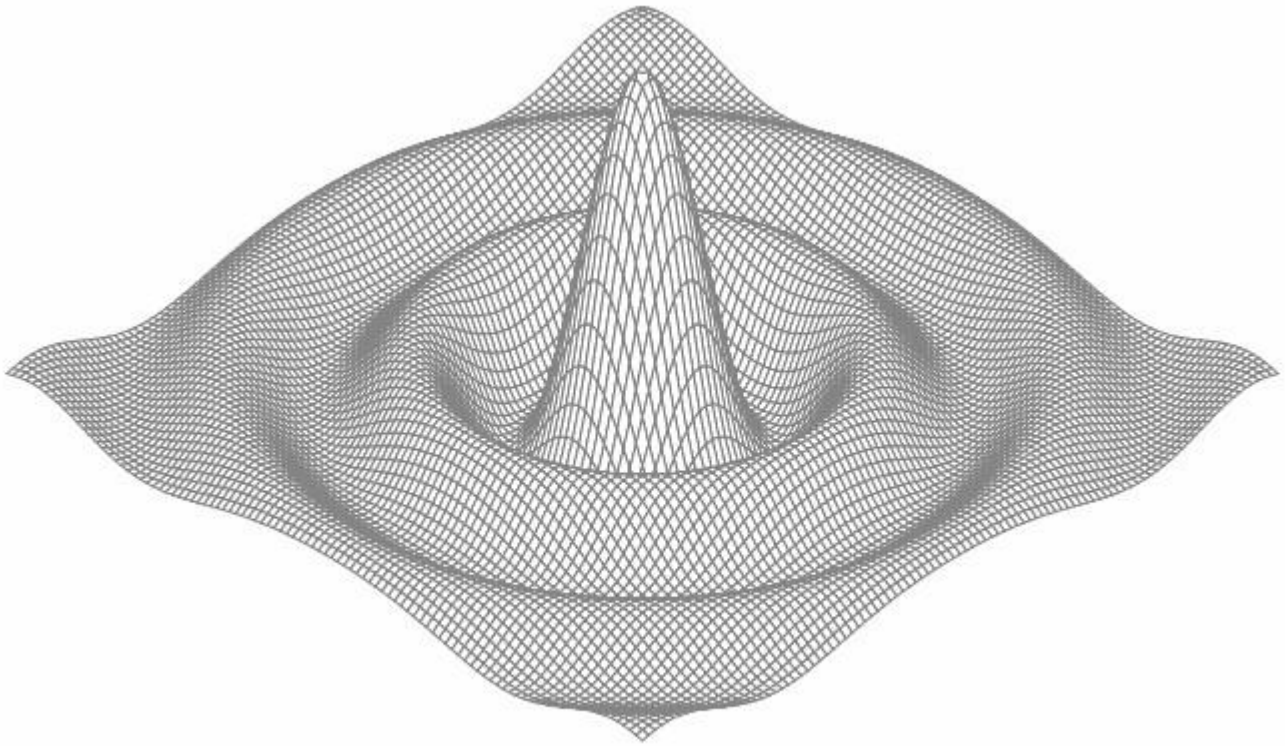


Figure 3.1. A surface plot of the function `sin(r)/r`.

```
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，因爲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 100x100 的二 维网格，对应整数整数坐标(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畫布，起点(0,0)在左上角。畫布中点的坐标用(sx, sy)表示。我们使用等角投影将三维点

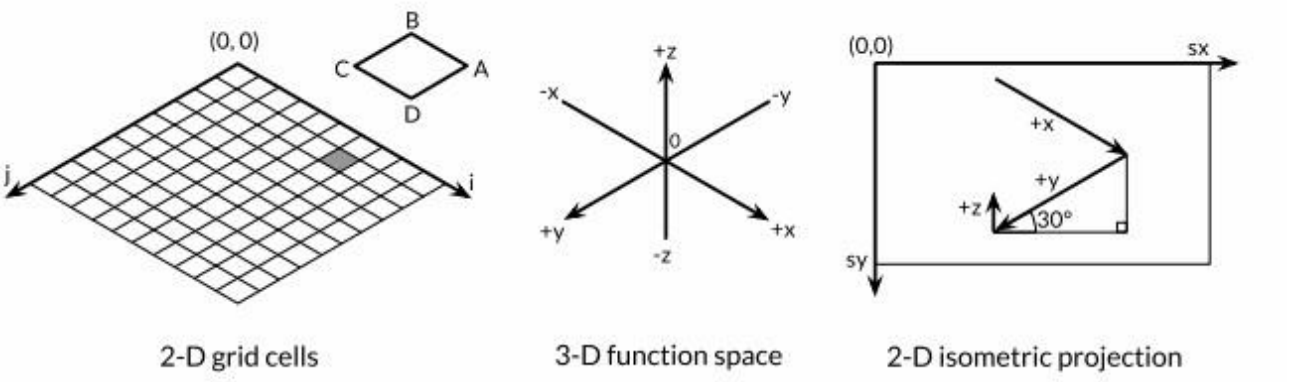


Figure 3.2. Three different coordinate systems.

(x,y,z)投影到二维的畫布中。畫布中从远处到右边的点对应较大的 x 值和较大的 y 值。并且畫布中 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畫布中对应多边形 ABCD 的顶点，其 中 B 对应(i,j)顶点位置，A、C 和 D 是其它相邻的顶点，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，那麼 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）。脩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，那样峯值部将是红色(#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，因爲服务器使用标准的 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。

复数

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/mandelbrot

// 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{255 - contrast*n}
        }
    }
    return color.Black
}
```

用于遍历 1024x1024 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对应

复数值平方值加一个增量值对应的点是否超出半径爲 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，那麽该点属于 Mandelbrot 集合，使用黑色颜色标记。最终程序将生成 的 PNG 格式分形图像图像输出到标准输出，如图 3.3 所示。

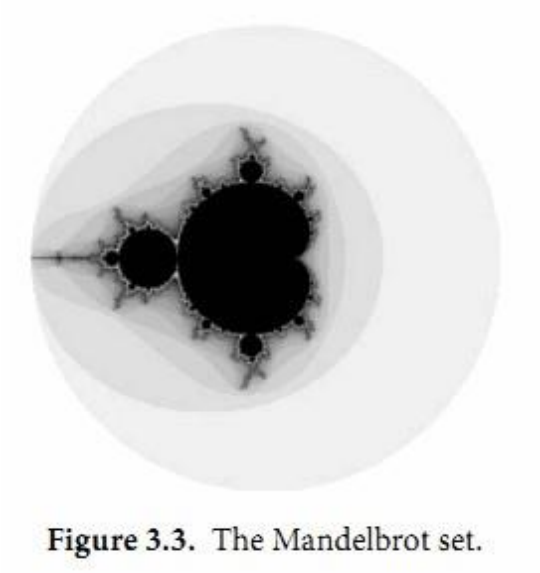


Figure 3.3. The Mandelbrot set.

练习 3.5： 实现一个彩色的 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： 陞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层四个子像素，实现它。

练习 3.7： 另一个生成分形图像的方式是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比如何？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。运行客户端用过 HTTP 参数参数指定 x,y 和 zoom 参数。

布尔型

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
操作也会産生布尔型的值。一元操作符！对应逻辑非操作，因此 `!true` 的值爲 `false`，更罗嗦的说法是 `(!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爲：如果运算符左边值已经可以确定整个布尔表达式的值，那麼运算符右边的值将不在被求值，因此下面的表达式总是安全的：

```
s != "" && 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。

因爲 && 的优先级比 || 高（助记： && 对应逻辑乘法， || 对应逻辑加法，乘法比加法优先级要高），下面形式的布尔表达式是不需要加小括号的：

```
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爲数字值 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, 不过爲了保持对称, 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 }
```

字符串

3.5. 字符串

一个字符串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爲采用 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，因爲对于非 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 作爲开始位置，采用 len(s)作爲结束的位置。

```
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"
```

这并不会导致原始的字符串值被改变，但是变量 s 将因爲+=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"
```

因爲字符串是不可脩改的，因此尝试脩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符串 共享相同的底层数据。

{% include "./ch3-05-1.md" %}

{% include "./ch3-05-2.md" %}

{% include "./ch3-05-3.md" %}

{% include "./ch3-05-4.md" %}

{% include "./ch3-05-5.md" %}

常量

3.6. 常量

常量表达式的值在编译期计算，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，常量的值不可脩改，这样可以防止在运行期被意外或恶意的脩改。例如，常量比变量更适合用于表达像 π 之类的数学常数，因爲它们的值不会发生 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期完成，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）。

因爲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明类型，那麼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。

{% include "./ch3-06-1.md" %}

{% include "./ch3-06-2.md" %}

复合数据类型

第四章 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译码 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。

数组

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。因爲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爲元素类型对应的零值，对于数字类型来说就是 0。我们也可以使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爲

```
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，因爲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值序列，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                 // 歐元
    GBP                 // 英鎊
    RMB                 // 人民幣
)

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爲-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，那麼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
```

作爲一个眞实的例子，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，那麼可 以认爲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，那麼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 信息的摘要：

```
gopl.io/ch4/sha256

import "crypto/sha256"

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。因爲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脩改都是发生在复制的数组上，并不能直接脩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它很多编程语言不同，其它编程语言可能会隐式地将数组作爲引用 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脩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脩改数组的值，但是数组依然是殭化的类型，因爲数组的类型包含了殭化的长度信息。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或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小数组 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： 编写一个程序，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。

Slice

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第 0 个元素，第 0 个元素会被自动初始化爲空字符串。

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：

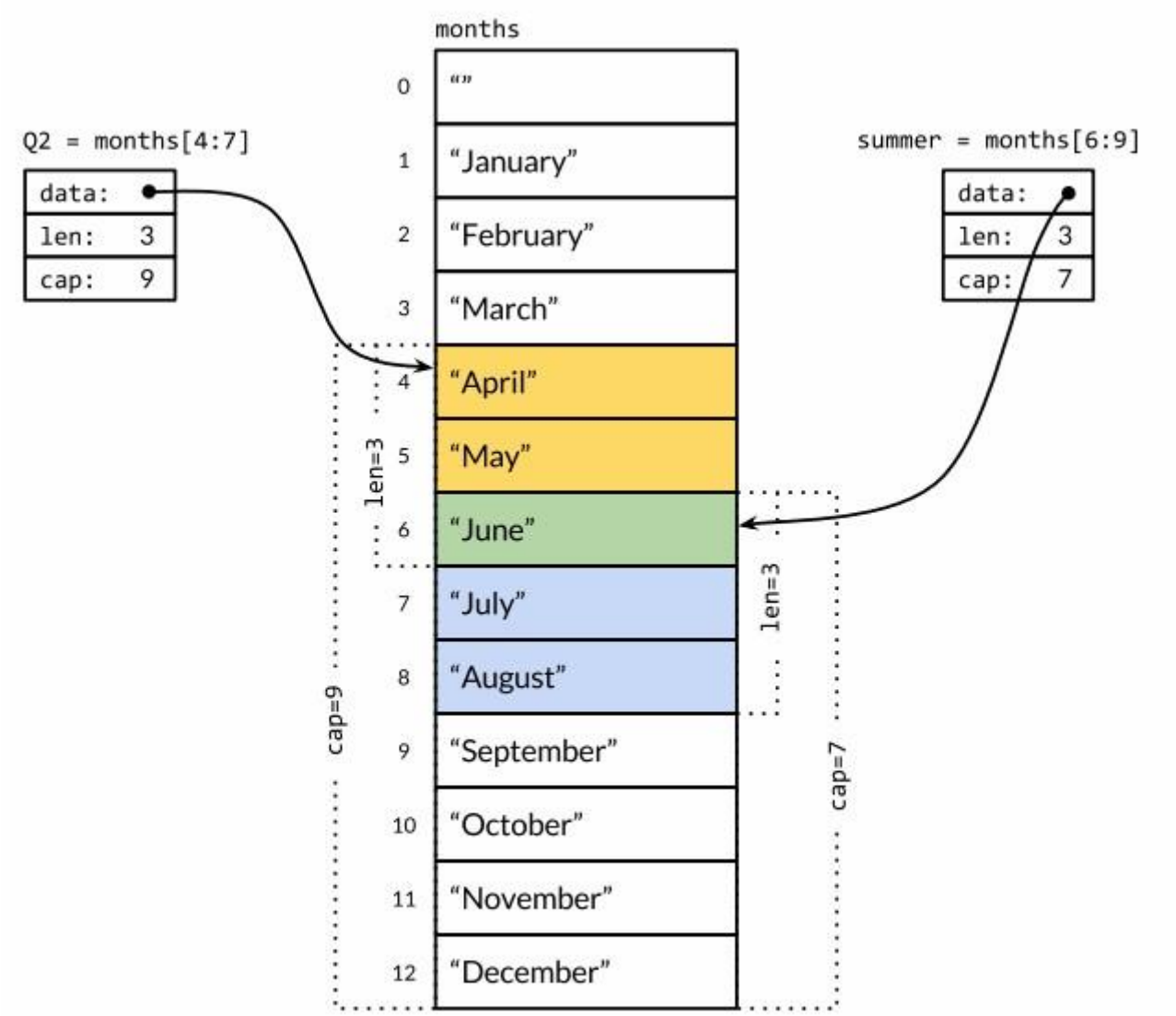


Figure 4.1. Two overlapping slices of an array of months.

```
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，因爲新 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。它们都写作 x[m:n]，并且都是返回一个原始字节系列的子序列，底层都是共享之前的底层数组，因此切片操作对应常量时间复杂度。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 的话则生成一个新的[]byte。

因爲 slice 值包含指向第一个 slice 元素的指针，因此向函数传递 slice 将允许在函数内部脩改底层数组的元 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 元素循环向左镟转 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。（如果是向右循环镟转，则将第三个函数调用移到 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 的初始化语法的差异。slice 和数组的字面值语法很类似，
它们都是用花括号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，或者的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，但是爲何 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，因爲 slice 的元素是间接引用的，一个固定值的 slice 在不同的时间可能包含不同的元素，因爲 底层数组的元素可能会被脩改。并且 Go 语言中 map 等哈希表之类的数据结构的 key 只做简单的浅拷贝，它 要求在整个声明周期中相等的 key 必须对相同的元素。对于像指针或 chan 之类的引用类型，== 相等测试可 以判断两个是否是引用相同的对象。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爲会让人睏惑。因此，安全的做饭是直接 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 的行爲和其它任意 0 産长度的 slice 一样；例如 reverse(nil)也是安全的。除了文 档已经明确帮助的地方，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的数组。额外的元素是留给未来的增长用的。

{% include "./ch4-02-1.md" %}

{% include "./ch4-02-2.md" %}

Map

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 类型可以写爲 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，所以的 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运算符比较的，但是将浮点数用做 key 类型则是一 个坏的想法，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，因爲 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 元素取址的原因是 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：

```
import "sort"

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])
}
```

因爲我们一开始就知道 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，它们的行爲和

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 作爲索引下标来访问 map 将産生一个 value。如果 key 在 map 中是存在的，那麽将得到与 key 对应 的 value；如果 key 不存在，那麽将得到 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眞的是在 map 之中。例如，如果元素类型是一个 数字，你可以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 的下标语法将産生两个值；第二个是一个布尔值，用于报告元素是否眞的存在。布尔变量一般命名爲 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
}
```

要注意我们是如何用!ok 来区分元素缺失和元素不同的。我们不能简单地用 xv != y[k]判断，那样会导致在判断下面两个 map 时産生错误的结果：

```
// True if equal is written incorrectly.
equal(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 实现类似 set 的功能。爲
了帮助这一点，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 `map[string]bool` 类型 value 都是无关所有
紧要的；有一些则可能会同时包含 tue 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 转爲 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 爲 string 类型的 map，在每次对 map 操作时先用 k 辅助函数将 slice 转化爲 string 类型。

下面的例子演示了如何使用 map 来记録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 列表转换爲一个字符串以用于 map 的 key，通过%q 参数忠实地记録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。

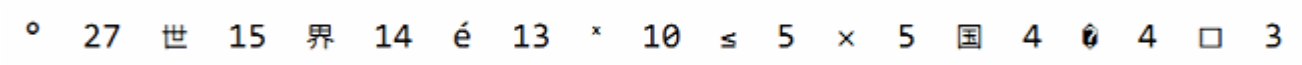
```
gopl.io/ch4/charcount

// 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 译码并返回三个值：译码的 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；因爲 UTF-8 编码的长度总是从 1 到 utf8.UTFMax（最大是 4 个字节），使用数组将更有效。

作爲一个实验，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。从概念上将，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爲 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。

结构体

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。每个值称爲结构体的成员。用结构体的经典案例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，可以作爲一个整体单元被复制，作爲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。因爲 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 EmployeeByID
函数的返回值从 `*Employee` 指针类型改爲 Employee 值类型，那麼更新语句将不能编译通过，因爲在赋值
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如果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（因爲也是字符串类型），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，那麼该成员就是导出的；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，因爲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。

一个命名爲 S 的结构体类型将不能再包含 S 类型的成员：因爲一个聚合的值不能包含它自身。（该限制同样
适应于数组。）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对是零值。通常会将零值作爲最合理的默认值。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{}。它的大小爲 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 带模拟 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，但是因爲节约的空间有限，而且语法比较复杂，所有我们通常避免避免这 样的用法。

```
seen := make(map[string]struct{}) // set of strings
// ...
if _, ok := seen[s]; !ok {
    seen[s] = struct{}{}
    // ...first time seeing s...
}
```

{% include "./ch4-04-1.md" %}

{% include "./ch4-04-2.md" %}

{% include "./ch4-04-3.md" %}

JSON

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 本文编码。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章的数组、slice、结构体和 map 等聚合数据类型。

基本的 JSON 类型有数字（十进制或科学记数法）、布尔值（true 或 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，写成以系列的 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 们稍后会解释它的作用。）

```
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，并且在两种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 转爲 JSON 的过程叫编组（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 函数返还一个编码后的字节 slice，包含很长的字符串，并且没有空白缩进；我们将它摺行以便于显示：

```
[{"Title":"Casablanca","released":1942,"Actors":["Humphrey Bogart","Ingr
id Bergman"]},{"Title":"Cool Hand Luke","released":1967,"color":true,"Ac
tors":["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，但是很难閱读。爲了生成便于閱读的格式，另一个 json.MarshalIndent 函数将産生整齐缩进的输出。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)
```

上面的代码将産生这样的输出（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爲 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，这也就是我们爲什麽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。这是因爲构体成员 Tag 所导致的。一个构体成员 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
爲值中含义双引号字符，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爲，并且 encoding/...下面其它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 语言结构体成员爲空或 零值时不生成 JSON 对象（这里 false 爲零值）。果然，Casablanca 是一个黑白电影，并没有输出 Color 成 员。

编码的逆操作是译码，对应将 JSON 数据译码爲 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 格式的电影数据译码爲一个结构体 slice，结构体中只有 Title 成员。通过定义合适的 Go 语言数据结构，我们可以选择性地译码 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。爲了帮助这
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
// Package github provides a Go API for the GitHub issue tracker.
// 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爲大小字母开头的。因爲有些 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，在译码的时候也需要做同样的处理，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，然后译码返回的 JSON 格式的结果。因爲用户提供的查询条件可能
包含类似`?`和`&`之类的特殊字符，爲了避免对 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
package github

import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "strings"
)

// 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 格式的字符串译码爲字节 slice。但是这个例
子中，我们使用了基于流式的译码器 json.Decoder，它可以从一个输入流译码 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。这里有多种方法可以格式化结构。下面是最简单的一种，以一个固定宽 度打印每个 issue，但是在下一节我们将看到如果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"
    "os"

    "gopl.io/ch4/github"
)

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 译码相关的问题，还有查询返回
的结果：

```
$ go build gopl.io/ch4/issues
$ ./issues repo:golang/go is:open json decoder
13 issues:
#5680    eaigner encoding/json: set key converter on en/decoder
#6050  gopherbot encoding/json: provide tokenizer
#8658  gopherbot encoding/json: use bufio
#8462  kortschak encoding/json: UnmarshalText confuses json.Unmarshal
#5901        rsc encoding/json: allow override type marshaling
#9812  klauspost encoding/json: string tag not symmetric
#7872  extempora encoding/json: Encoder internally buffers full output
#9650    cespare encoding/json: Decoding gives errPhase when unmarshalin
#6716  gopherbot encoding/json: include field name in unmarshal error me
#6901  lukescott encoding/json, encoding/xml: option to treat unknown fi
#6384    joeshaw encoding/json: encode precise floating point integers u
#6647    btracey x/tools/cmd/godoc: display type kind of each named type
#4237  gjemiller encoding/base64: URLEncoding padding is optional
```

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、更新和删除 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 漫畵服务 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 相匹配的漫畵的 URL。

练习 4.13： 使用开放电影数据库的 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。

文本和 HTML 模板

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脩改。这写功能是由 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面值打印，但是对于 actions 部分将触发其它的行爲。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，还有其它实例化模板等诸多特性。下面是一 个简单的模板字符串：

{% raw %}

```
gopl.io/ch4/issuesreport

const templ = `{{.TotalCount}} issues:
{{range .Items}}----------------------------------------
Number: {{.Number}}
User:   {{.User.Login}}
Title:  {{.Title | printf "%.64s"}}
Age:    {{.CreatedAt | daysAgo}} days
{{end}}`
```

{% endraw %}

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 一个 action，都有一个当前值的概念，对应点操作符，写作 “.” 。当前值 “.” 最初被初始化爲调用模板是
的参数，在当前例子中对应 github.IssuesSearchResult 类型的变量。模板中 将 {{.TotalCount}} 对应 action 展开爲结构体中 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爲后一个函数的输入，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 成员转换爲过去的时间长度：

```
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 编码和解码行爲。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爲内部表示，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)
}
```

因爲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：它接受一个模板和一个 error 类型的参数，检测 error 是否爲 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 作爲输入源、os.Stdout 作爲输出源来执行模板
：

```
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：

```
$ go build gopl.io/ch4/issuesreport
$ ./issuesreport repo:golang/go is:open json decoder
13 issues:
----------------------------------------
Number: 5680
User:      eaigner
Title:     encoding/json: set key converter on en/decoder
Age:       750 days
----------------------------------------
Number: 6050
User:      gopherbot
Title:     encoding/json: provide tokenizer
Age:       695 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 语法産生冲突的问 题。这个特性还可以避免一些长期存在的安全问题，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：

{% raw %}

```
gopl.io/ch4/issueshtml

import "html/template"

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}}</td>
  <td>{{.State}}</td>
  <td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>
  <td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>
</tr>
{{end}}
</table>
`))
```

{% endraw %}

下面的命令将在新的模板上执行一个稍微不同的查询：

```
$ go build gopl.io/ch4/issueshtml
$ ./issueshtml 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。

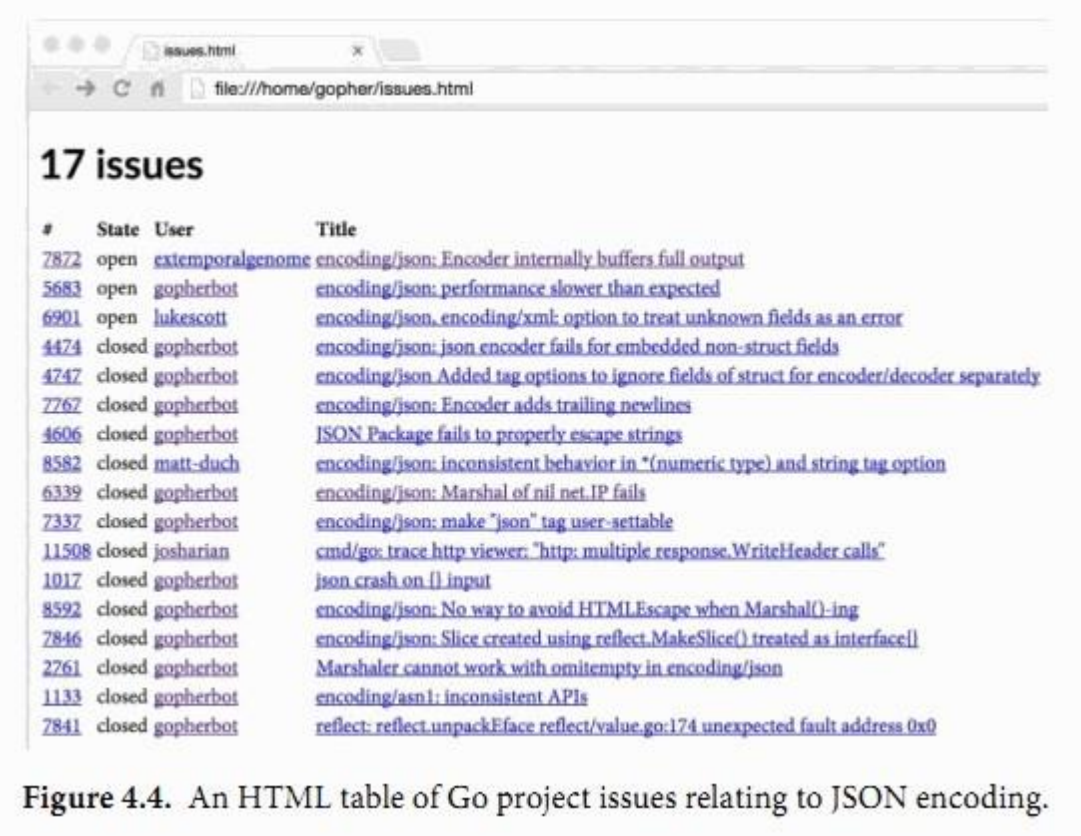


Figure 4.4. An HTML table of Go project issues relating to JSON encoding.

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 `&` 和 `<` 字符含有
的 issue。下面的命令选择了两个这样的 issue：

```
$ ./issueshtml 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“<”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爲。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 相同字符串産生的不同结果：A 是一个普通字符串，B 是一个信任的 template.HTML 字符串类型。

Figure 4.5. HTML metacharacters in issue titles are correctly displayed.

{% raw %}

```
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)
    }
}
```

{% endraw %}

图 4.6 显示了出现在浏览器中的模板输出。我们看到 A 的黑体标记被转义失效了，但是 B 没有。

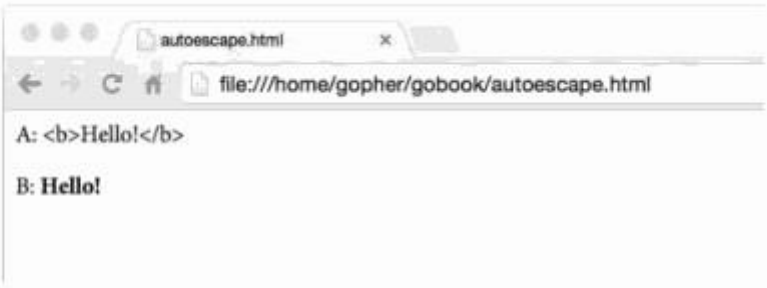


Figure 4.6. String values are HTML-escaped but `template.HTML` values are not.

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

第五章 函数

函数可以让我们将一个语句序列打包爲一个单元，然后可以从程序中其它地方多次调用。函数的机制可以让我们将一个大的工作分解爲小的任务，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。

函数声明

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。这些参数作爲局部变量，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，那麼函数体执行完毕后，不会返回任何值。
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数，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，将其初始化爲 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，我们不必爲每个形参都写出参数类型。下面 2 个声 明是等价的：

```
func f(i, j, k int, s, t string)                 { /* ... */ }
func f(i int, j int, k int,  s string, t string) { /* ... */ }
```

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(译者注：即下文
的_符号)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"
```

函数的类型被称爲函数的标识符。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，那麼这两个函数被认爲有相同的类型和标识符。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爲所有参数提供实参（参数值）。在函数调用时，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，函数的形参作爲局部变量，被初始化爲调用者提供的值。函数的形参和有名返回值作爲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。对形参进行脩改不会影响实参。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简介引用 被脩改。

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
```

递归

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的调用自身。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/... 目録下存储了一些由 Go 团队设计、维护，对网络编程、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。

例子中调用 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 拥有很多类型的结点如 text（文本）,commnets（注 释）类型，在下面的例子中，我们 只关注< name key='value' >形式的结点。

```
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"
    "os"

    "golang.org/x/net/html"
)

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
}
```

爲了遍历结点 n 的所有后代结点，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）作爲目标，运行 findlinks。我们以 fetch（1.5 章）的输出作爲 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，根据根路径（ 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 结点树，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。被调用者的入栈操作，修改的是 stack 的拷贝，而不是调用者的 stack,因对当函数返回时, 调用者的 stack 并未被修改。

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 代码中遍历 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 text 结点的内容。注意不要访 `<script>` 和 `<style>` 元素,因爲这些元素对浏问 览者是不可见的。

练习 5.4 : 扩展 vist 函数，使其能够处理其他类型的结点，如 images、scripts 和 style sheets。

多返回值

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。因爲 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 页面的错误结点，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("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)
    }
    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。前三处 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 finallinks 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。

调用多返回值函数时，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作爲返回值，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)
}
```

当你调用接受多参数的函数时，可以将一个返回多参数的函数作爲该函数的参数。虽然这很少出现在实际生産代码中，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，但你也不必爲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。

如果一个函数将所有的返回值都显示的变量名，那麽该函数的 return 语句可以省略操作数。这称之爲 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，如果你没有仔细的审查代码，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，将其初始化爲 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： 脩改 gopl.io/ch3/surface (§3.2) 中的 corner 函数，将返回值命名，并使用 bare return。

错误

5.4. 错误

在 Go 中有一部分函数总是能成功的运行。比如 string.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。这是因爲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，即时是简单的读写。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爲 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，通常被命名爲 ok。比 如，cache.Lookup 失败的唯一原因是 key 不存在，那麽代码可以按照下面的方式组织：

```
value, ok := cache.Lookup(key)
if !ok {
    // ...cache[key] does 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。然而，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帮助，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，这些错误信息被认爲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，而不是那些在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作异常处理后，这个错误会将堆栈 根据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 使用控制流机制（如 if 和 return）处理异常，这使得编码人员能更多的关注错误处理。

{% include "./ch5-04-1.md" %}

{% include "./ch5-04-2.md" %}

函数值

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。调用值爲 nil 的函数值会引起 panic 错误：

```
var f func(int) int
f(3) // 此處 f 的值爲 nil,會引起 panic 錯誤
```

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，也不能用函数值作爲 map 的 key。

函数值使得我们不仅仅可以通过数据来参数化函数，亦可通过行爲。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。string.Map 对字符串中的每个字符调用 add1 函数，并将每个 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 針對每個結點 x,都會調用 pre(x)和 post(x)。
// pre 和 post 都是可選的。
// 遍歷孩子結點之前,pre 被調用
// 遍歷孩子結點之後，post 被調用
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 个函数作爲参数，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 startElemen 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，使其成爲通用的 HTML 输出器。要求：输出注释结点，
文本结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： 脩改 pre 和 post 函数，使其返回布尔类型的返回值。返回 false 时，中止 forEachNoded 的遍历。
使用脩改后的代码编写 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"替换爲 f("foo")的返回值。

```
func expand(s string, f func(string) string) string
```

匿名函数

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，它的值被成爲匿名函数（anonymous function）。

函数字面量允许我们在使用时函数时，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")
```

更爲重要的是，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，如下例所示：

```
gopl.io/ch5/squares
// squares 返迴一個匿名函數。
// 該匿名函數每次被調用時都會返迴下一個數的平方。
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 返回另一个类型爲 func() int 的函数。对 squares 的一次调用会生成一个局部变量 x 并返回一

个匿名函数。每次调用时匿名函数时，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，还记録了状态。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，我们讨论一个有点学术性的例子，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。因爲 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，将函数重命名爲 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"
    "net/http"
    "golang.org/x/net/html"
)
// 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，这些连接以絶对路径的形式存在，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 个元素被定义爲 string 类型。广度优先算法会爲每个元素调用一次 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。爲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.
// Any 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 作爲参数传递给 breadthFirst。

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

爲了使抓取器开始运行，我们用命令行输入的参数作爲初始的待访问 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计算机的内存耗尽时，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爲了前置课程。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 共享了全局变量 depth，将它 们脩改爲匿名函数，使其共享 outline 中的局部变量。

练习 5.13： 脩改 crawl，使其能保存发现的页面，必要时，可以创建目録来保存这些页面。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）。

{% include "./ch5-06-1.md" %}

可变参数

5.7. 可变参数

参数数量可变的函数称爲爲可变参数函数。典型的例子就是 fmt.Printf 和类似函数。Printf 首先接收一个的 必备参数，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。在函数体中,vals 被看作是类型爲[] 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爲参数传给被调函数。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...)) // "10"
```

虽然在可变参数函数内部，...int 型参数的行爲看起来很像切片类型，但实际上，可变参数函数和以切片作爲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"
```

interfac{}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。考虑不传参时，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")
```

Deferred 函数

Panic 异常

5.9. Panic 异常

TODO

Recover 捕获异常

5.10. Recover 捕获异常

TODO

方法

第六章 方法

从 90 年代早期开始，面向对象编程(OOP)就成爲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。

尽管没有被大衆所接受的明确的 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", 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 编程的
两个关键点，封装和组合。

方法声明

6.1. 方法声明

在函数声明时，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，即相当于爲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 geometry 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爲“向一个对象发送消息”。

在 Go 语言中，我们并不会像其它语言那样用 this 或者 self 作爲接收器；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，因爲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，因爲在调用 p.X 时会有歧义(译注：这里确实挺奇怪的)。

因爲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，然而我们依然可以爲它定义方法。在能够给
任意类型定义方法这一点上，Go 和很多其它的面向对象的语言不太一样。因此在 Go 语言里，我们爲一些 简单的数值、字符串、slice、map 来定义一些附加行爲很方便。方法可以被声明到任意类型，只要不是一 个指针或者一个 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，因爲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 爲 Path 这个变量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。因爲在 Go 里包外调用函数需要带上包名，还是挺麻烦的。

基于指针对象的方法

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爲 `*(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爲接收器的方法，那麽所有 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爲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)，才是可能会出现在接收器声明里的两种接收器。此外，爲了避免歧义，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 指针作爲接收器，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，因爲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晕，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：

不论是接收器的实际参数和其接收器的形式参数相同，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
```

或者接收器形参是类型 T，但接收器实参是类型*T，这种情况下编译器会隐式地爲我们取变量的地址：

```
p.ScaleBy(2) // implicit (&p)
```

或者接收器形参是类型*T，实参是类型 T。编译器会隐式地爲我们解引用，取到指针指向的实际变量：

```
pptr.Distance(q) // implicit (*pptr)
```

如果类型 T 的所有方法都是用 T 类型自己来做接收器(而不是*T)，那麽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產生一个值的拷贝。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，包括在作爲参数传入函数的时候。但是如果一个方法使用指针作爲接收器，你需要避免对其 进行拷贝，因爲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，那麽可能 会引起原始对象和拷贝对象只是别名而已，但实际上其指向的对象是一致的。紧接着对拷贝后的变量进行 脩改可能会有让你意外的结果。

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类型时，你需要考虑两方面的内部，第一方面是这个 对象本身是不是特别大，如果声明爲非指针变量时，调用会產生一次拷贝；第二方面是如果你用指针类型 作爲 receiver，那麽你一定要注意，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 艹的人这里应该很快能明白。

{% include "./ch6-02-1.md" %}

通过嵌入结构体来扩展类型

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 定义爲一个有三个字段的 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 时得到一种句法上的简写形式，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，我们可以直接认爲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 定义爲下面这样：

```
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手段。

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，但将两个包级吧的变量放在了 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。因爲 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。

方法值和方法表达式

6.4. 方法值和方法表达式

我们经常选择一个方法，并且在同一个表达式里执行，比如常见的 p.Distance()形式，实际上将其分成两步 来执行也是可能的。p.Distance 叫作 “选择器” ，选择器会返回一个方法"值"->一个将方法(Point.Distance)绑定到特定接收器变量的函数。这个函数可以不通过指定其接收器即可被调用；即调用时不需要指定接收器(译注：因爲已经在前文中指定过了)，只要传入函数的参数即可：

```
p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance        // method value
fmt.Println(distanceFromP(q))      // "5"
var origin Point                   // {0, 0}
fmt.Println(distanceFromP(origin)) // "2.23606797749979", sqrt(5)

scaleP := p.ScaleBy // method value
scaleP(2)          // p becomes (2, 4)
scaleP(3)          //      then (6, 12)
scaleP(10)         //      then (60, 120)
```

在一个包的 API 需要一个函数值、且调用方希望操作的是某一个绑定了对象的方法的话，方法"值"会非常实
用(=_=眞是绕)。举例来说，下面例子中的 time.AfterFunc 这个函数的功能是在指定的延迟时间之后来执行 一个(译注：另外的)函数。且这个函数操作的是一个 Rocket 对象 r

```
type Rocket struct { /* ... */ }
func (r *Rocket) Launch() { /* ... */ }
r := new(Rocket)
time.AfterFunc(10 * time.Second, func() { r.Launch() })
```

直接用方法"值"传入 AfterFunc 的话可以更爲简短：

```
time.AfterFunc(10 * time.Second, r.Launch)
```

译注：省掉了上面那个例子里的匿名函数。

和方法"值"相关的还有方法表达式。当调用一个方法时，与调用一个普通的函数相比，我们必须要用选择器(p.Distance)语法来指定方法的接收器。

当 T 是一个类型时，方法表达式可能会写作 T.f 或者(*T).f，会返回一个函数"值"，这种函数会将其第一个参 数用作接收器，所以可以用通常(译注：不写选择器)的方式来对其进行调用：

```
p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance // method expression
fmt.Println(distance(p, q))  // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"

scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p)            // "{2 4}"
fmt.Printf("%T\n", scale) // "func(*Point, float64)"

// 譯註：這個 Distance 實際上是指定了 Point 對象爲接收器的一個方法 func (p Point) Distance()，
// 但通過 Point.Distance 得到的函數需要比實際的 Distance 方法多一個參數，
// 即其需要用第一個額外參數指定接收器，後面排列 Distance 方法的參數。
// 看起來本書中函數和方法的區別是指有沒有接收器，而不像其他語言那樣是指有沒有返迴值。
```

当你根据一个变量来决定调用同一个类型的哪个函数时，方法表达式就显得很有用了。你可以根据选择来调用接收器各不相同的方法。下面的例子，变量 op 代表 Point 类型的 addition 或者 subtraction 方 法，Path.TranslateBy 方法会爲其 Path 数组中的每一个 Point 来调用对应的方法：

```
type Point struct{ X, Y float64 }

func (p Point) Add(q Point) Point { return Point{p.X + q.X, p.Y + q.Y} }
func (p Point) Sub(q Point) Point { return Point{p.X - q.X, p.Y - q.Y} }

type Path []Point

func (path Path) TranslateBy(offset Point, add bool) {
    var op func(p, q Point) Point
    if add {
        op = Point.Add
    } else {
        op = Point.Sub
    }
    for i := range path {
        // Call either path[i].Add(offset) or path[i].Sub(offset).
        path[i] = op(path[i], offset)
    }
}
```

示例: Bit 数组

6.5. 示例: Bit 数组

Go 语言里的集合一般会用 map[T]bool 这种形式来表示，T 代表元素类型。集合用 map 类型来表示虽然非常 灵活，但我们可以以一种更好的形式来表示它。例如在数据流分析领域，集合元素通常是一个非负整数， 集合会包含很多元素，并且集合会经常进行并集、交集操作，这种情况下，bit 数组会比 map 表现更加理 想。(译注：这里再补充一个例子，比如我们执行一个 http 下载任务，把文件按照 16kb 一块划分爲很多块， 需要有一个全局变量来标识哪些块下载完成了，这种时候也需要用到 bit 数组)

一个 bit 数组通常会用一个无符号数或者称之爲“字”的 slice 或者来表示，每一个元素的每一位都表示集合 里的一个值。当集合的第 i 位被设置时，我们才说这个集合包含元素 i。下面的这个程序展示了一个简单的 bit 数组类型，并且实现了三个函数来对这个 bit 数组来进行操作：

```
gopl.io/ch6/intset
// An IntSet is a set of small non-negative integers.
// Its zero value represents the empty set.
type IntSet struct {
    words []uint64
}

// Has reports whether the set contains the non-negative value x.
func (s *IntSet) Has(x int) bool {
    word, bit := x/64, uint(x%64)
    return word < len(s.words) && s.words[word]&(1<<bit) != 0
}

// Add adds the non-negative value x to the set.
func (s *IntSet) Add(x int) {
    word, bit := x/64, uint(x%64)
    for word >= len(s.words) {
        s.words = append(s.words, 0)
    }
    s.words[word] |= 1 << bit
}

// UnionWith sets s to the union of s and t.
func (s *IntSet) UnionWith(t *IntSet) {
    for i, tword := range t.words {
        if i < len(s.words) {
            s.words[i] |= tword
        } else {
            s.words = append(s.words, tword)
        }
    }
}
```

因爲每一个字都有 64 个二进制位，所以爲了定位 x 的 bit 位，我们用了 x/64 的商作爲字的下标，并且用 x%64
得到的值作爲这个字内的 bit 的所在位置。UnionWith 这个方法里用到了 bit 位的“或”逻辑操作符号| 来一 次完成 64 个元素的或计算。(在练习 6.5 中我们还会程序用到这个 64 位字的例子。)

当前这个实现还缺少了很多必要的特性，我们把其中一些作爲练习题列在本小节之后。但是有一个方法如果缺失的话我们的 bit 数组可能会比较难混：将 IntSet 作爲一个字符串来打印。这里我们来实现它，让我们 来给上面的例子添加一个 String 方法，类似 2.5 节中做的那样：

```
// String returns the set as a string of the form "{1 2 3}".
func (s *IntSet) String() string {
    var buf bytes.Buffer
    buf.WriteByte('{')
    for i, word := range s.words {
        if word == 0 {
            continue
        }
        for j := 0; j < 64; j++ {
            if word&(1<<uint(j)) != 0 {
                if buf.Len() > len("{") {
                    buf.WriteByte('}')
                }
                fmt.Fprintf(&buf, "%d", 64*i+j)"}")}}
            }
        }
    }
    buf.WriteByte('}')
    return buf.String()
}
```

这里留意一下 String 方法，是不是和 3.5.4 节中的 intsToString 方法很相似；bytes.Buffer 在 String 方法里经
常这麽用。当你爲一个复杂的类型定义了一个 String 方法时，fmt 包就会特殊对待这种类型的值，这样可以 让这些类型在打印的时候看起来更加友好，而不是直接打印其原始的值。fmt 会直接调用用户定义的 String 方法。这种机制依赖于接口和类型断言，在第 7 章中我们会详细介绍。

现在我们就可以在实战中直接用上面定义好的 IntSet 了：

```
var x, y IntSet
x.Add(1)
x.Add(144)
x.Add(9)
fmt.Println(x.String()) // "{1 9 144}"

y.Add(9)
y.Add(42)
fmt.Println(y.String()) // "{9 42}"

x.UnionWith(&y)
fmt.Println(x.String()) // "{1 9 42 144}"
fmt.Println(x.Has(9), x.Has(123)) // "true false"
```

这里要注意：我们声明的 String 和 Has 两个方法都是以指针类型*IntSet 来作爲接收器的，但实际上对于这
两个类型来说，把接收器声明爲指针类型也没什麽必要。不过另外两个函数就不是这样了，因爲另外两个函数操作的是 s.words 对象，如果你不把接收器声明爲指针对象，那麽实际操作的是拷贝对象，而不是原来 的那个对象。因此，因爲我们的 String 方法定义在 IntSet 指针上，所以当我们的变量是 IntSet 类型而不是 IntSet 指针时，可能会有下面这样让人意外的情况：

```
fmt.Println(&x)         // "{1 9 42 144}"
fmt.Println(x.String()) // "{1 9 42 144}"
fmt.Println(x)          // "{[4398046511618 0 65536]}"
```

在第一个 Println 中，我们打印一个*IntSet 的指针，这个类型的指针确实有自定义的 String 方法。第二 Println，我们直接调用了 x 变量的 String()方法；这种情况下编译器会隐式地在 x 前插入&操作符，这样相当 远我们还是调用的 IntSet 指针的 String 方法。在第三个 Println 中，因爲 IntSet 类型没有 String 方法，所以 Println 方法会直接以原始的方式理解并打印。所以在这种情况下&符号是不能忘的。在我们这种场景下， 你把 String 方法绑定到 IntSet 对象上，而不是 IntSet 指针上可能会更合适一些，不过这也需要具体问题具体 分析。

练习 6.1: 爲 bit 数组实现下面这些方法

```
func (*IntSet) Len() int      // return the number of elements
func (*IntSet) Remove(x int)  // remove x from the set
func (*IntSet) Clear()        // remove all elements from the set
func (*IntSet) Copy() *IntSet // return a copy of the set
```

练习 6.2: 定义一个变参方法(*IntSet).AddAll(...int)，这个方法可以爲一组 IntSet 值求和，比如 s.AddAll(1,2,3)。

练习 6.3: (*IntSet).UnionWith 会用|操作符计算两个集合的交集，我们再爲 IntSet 实现另外的几个函数 IntersectWith(交集：元素在 A 集合 B 集合均出现),DifferenceWith(差集：元素出现在 A 集合，未出现在 B 集 合),SymmetricDifference(并差集：元素出现在 A 但没有出现在 B，或者出现在 B 没有出现在 A)。

练习 6.4: 实现一个 Elems 方法，返回集合中的所有元素，用于做一些 range 之类的遍历操作。

练习 6.5: 我们这章定义的 IntSet 里的每个字都是用的 uint64 类型，但是 64 位的数值可能在 32 位的平台上不 高效。脩改程序，使其使用 uint 类型，这种类型对于 32 位平台来说更合适。当然了，这里我们可以不用简 单粗暴地除 64，可以定义一个常量来决定是用 32 还是 64，这里你可能会用到平台的自动判断的一个智能表 达式：32 << (^uint(0) >> 63)

封装

6.6. 封装

一个对象的变量或者方法如果对调用方是不可见的话，一般就被定义爲“封装”。封装有时候也被叫做信息隐藏，同时也是面向对象编程最关键的一个方面。

Go 语言只有一种控制可见性的手段：大写首字母的标识符会从定义它们的包中被导出，小写字母的则不会。这种限制包内成员的方式同样适用于 struct 或者一个类型的方法。因而如果我们想要封装一个对象，我 们必须将其定义爲一个 struct。

这也就是前面的小节中 IntSet 被定义爲 struct 类型的原因，尽管它只有一个字段：

```
type IntSet struct {
    words []uint64
}
```

当然，我们也可以把 IntSet 定义爲一个 slice 类型，尽管这样我们就需要把代码中所有方法里用到的 s.words
用*s 替换掉了：

```
type IntSet []uint64
```

尽管这个版本的 IntSet 在本质上是一样的，他也可以允许其它包中可以直接读取并编辑这个 slice。换句话说，相对*s 这个表达式会出现在所有的包中，s.words 只需要在定义 IntSet 的包中出现(译注：所以还是推荐 后者吧的意思)。

这种基于名字的手段使得在语言中最小的封装单元是 package，而不是像其它语言一样的类型。一个 struct 类型的字段对同一个包的所有代码都有可见性，无论你的代码是写在一个函数还是一个方法里。

封装提供了三方面的优点。首先，因爲调用方不能直接脩改对象的变量值，其只需要关注少量的语句并且只要弄懂少量变量的可能的值即可。

第二，隐藏实现的细节，可以防止调用方依赖那些可能变化的具体实现，这样使设计包的程序员在不破坏对外的 api 情况下能得到更大的自由。

把 bytes.Buffer 这个类型作爲例子来考虑。这个类型在做短字符串叠加的时候很常用，所以在设计的时候可 以做一些预先的优化，比如提前预留一部分空间，来避免反复的内存分配。又因爲 Buffer 是一个 struct 类 型，这些额外的空间可以用附加的字节数组来保存，且放在一个小写字母开头的字段中。这样在外部的调 用方只能看到性能的提陞，但并不会得到这个附加变量。Buffer 和其增长算法我们列在这里，爲了简洁性 稍微做了一些精简：

```
type Buffer struct {
    buf     []byte
    initial [64]byte
    /* ... */
}

// Grow expands the buffer's capacity, if necessary,
// to guarantee space for another n bytes. [...]
func (b *Buffer) Grow(n int) {
    if b.buf == nil {
        b.buf = b.initial[:0] // use preallocated space initially
    }
    if len(b.buf)+n > cap(b.buf) {
        buf := make([]byte, b.Len(), 2*cap(b.buf) + n)
        copy(buf, b.buf)
        b.buf = buf
    }
}
```

封装的第三个优点也是最重要的优点，是阻止了外部调用方对对象内部的值任意地进行修改。因爲对象内部变量只可以被同一个包内的函数修改，所以包的作者可以让这些函数确保对象内部的一些值的不变性。比如下面的 Counter 类型允许调用方来增加 counter 变量的值，并且允许将这个值 reset 爲 0，但是不允许随 便设置这个值(译注：因爲压根就访问不到)：

```
type Counter struct { n int }
func (c *Counter) N() int     { return c.n }
func (c *Counter) Increment() { c.n++ }
func (c *Counter) Reset()     { c.n = 0 }
```

只用来访问或修改内部变量的函数被称爲 setter 或者 getter，例子如下，比如 log 包里的 Logger 类型对应的
一些函数。在命名一个 getter 方法时，我们通常会省略掉前面的 Get 前缀。这种简洁上的偏好也可以推广到 各种类型的前缀比如 Fetch，Find 或者 Lookup。

```
package log
type Logger struct {
    flags  int
    prefix string
    // ...
}
func (l *Logger) Flags() int
func (l *Logger) SetFlags(flag int)
func (l *Logger) Prefix() string
func (l *Logger) SetPrefix(prefix string)
```

Go 的编码风格不禁止直接导出字段。当然，一旦进行了导出，就没有办法在保证 API 兼容的情况下去除对其的导出，所以在一开始的选择一定要经过深思熟虑并且要考虑到包内部的一些不变量的保证，未来可能的变化，以及调用方的代码质量是否会因爲包的一点修改而变差。

封装并不总是理想的。 虽然封装在有些情况是必要的，但有时候我们也需要暴露一些内部内容，比如：time.Duration 将其表现暴 露爲一个 int64 数字的纳秒，使得我们可以用一般的数值操作来对时间进行对比，甚至可以定义这种类型的 常量：

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

另一个例子，将 IntSet 和本章开头的 geometry.Path 进行对比。Path 被定义爲一个 slice 类型，这允许其调
用 slice 的字面方法来对其内部的 points 用 range 进行迭代遍历；在这一点上，IntSet 是没有办法让你这麼做 的。

这两种类型决定性的不同：geometry.Path 的本质是一个坐标点的序列，不多也不少，我们可以预见到之后也并不会给他增加额外的字段，所以在 geometry 包中将 Path 暴露爲一个 slice。相比之下，IntSet 仅仅 是在这里用了一个[]uint64 的 slice。这个类型还可以用[]uint 类型来表示，或者我们甚至可以用其它完全不 同的占用更小内存空间的东西来表示这个集合，所以我们可能还会需要额外的字段来在这个类型中记録元 素的个数。也正是因爲这些原因，我们让 IntSet 对调用方透明。

在这章中，我们学到了如何将方法与命名类型进行组合，并且知道了如何调用这些方法。尽管方法对于 OOP 编程来说至关重要，但他们只是 OOP 编程里的半边天。爲了完成 OOP，我们还需要接口。Go 里的接 口会在下一章中介绍。

接口

第七章 接口

接口类型是对其它类型行爲的抽象和概括；因爲接口类型不会和特定的实现细节绑定在一起，通过这种抽象的方式我们可以让我们的函数更加灵活和更具有适应能力。

很多面向对象的语言都有相似的接口概念，但 Go 语言中接口类型的独特之处在于它是满足隐式实现的。也 就是说，我们没有必要对于给定的具体类型定义所有满足的接口类型；简单地拥有一些必需的方法就足够 了。这种设计可以让你创建一个新的接口类型满足已经存在的具体类型却不会去改变这些类型的定义；当 我们使用的类型来自于不受我们控制的包时这种设计尤其有用。

在本章，我们会开始看到接口类型和值的一些基本技巧。顺着这种方式我们将学习几个来自标准库的重要接口。很多 Go 程序中都尽可能多的去使用标准库中的接口。最后,我们会在(§7.10)看到类型断言的知识， 在(§7.13)看到类型开关的使用并且学到他们是怎样让不同的类型的概括成爲可能。

接口是合约

7.1. 接口约定

目前爲止，我们看到的类型都是具体的类型。一个具体的类型可以准确的描述它所代表的值并且展示出对类型本身的一些操作方式就像数字类型的算术操作，切片类型的索引、附加和取范围操作。具体的类型还可以通过它的方法提供额外的行爲操作。总的来说，当你拿到一个具体的类型时你就知道它的本身是什麽和你可以用它来做什麽。

在 Go 语言中还存在着另外一种类型：接口类型。接口类型是一种抽象的类型。它不会暴露出它所代表的对 象的内部值的结构和这个对象支持的基础操作的集合；它们只会展示出它们自己的方法。也就是说当你有 看到一个接口类型的值时，你不知道它是什麽，唯一知道的就是可以通过它的方法来做什麽。

在本书中，我们一直使用两个相似的函数来进行字符串的格式化：fmt.Printf 它会把结果写到标准输出和 fmt.Sprintf 它会把结果以字符串的形式返回。得益于使用接口，我们不必可悲的因爲返回结果在使用方式 上的一些浅显不同就必需把格式化这个最晒难的过程复制一份。实际上，这两个函数都使用了另一个函数 fmt.Fprintf 来进行封装。fmt.Fprintf 这个函数对它的计算结果会被怎麽使用是完全不知道的。

```
package fmt
func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)
func Printf(format string, args ...interface{}) (int, error) {
    return Fprintf(os.Stdout, format, args...)
}
func Sprintf(format string, args ...interface{}) string {
    var buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return buf.String()
}
```

Fprintf 的前缀 F 表示文件(File)也表明格式化输出结果应该被写入第一个参数提供的文件中。在 Printf 函数中
的第一个参数 os.Stdout 是*os.File 类型；在 Sprintf 函数中的第一个参数&buf 是一个指向可以写入字节的内 存缓冲区，然而它
并不是一个文件类型尽管它在某种意义上和文件类型相似。 即使 Fprintf 函数中的第一个参数也不是一

个文件类型。它是 io.Writer 类型这是一个接口类型定义如下：

```
package io
// Writer is the interface that wraps the basic Write method.
type Writer interface {
    // Write writes len(p) bytes from p to the underlying data stream.
    // It returns the number of bytes written from p (0 <= n <= len(p))
    // and any error encountered that caused the write to stop early.
    // Write must return a non-nil error if it returns n < len(p).
    // Write must not modify the slice data, even temporarily.
    //
    // Implementations must not retain p.
    Write(p []byte) (n int, err error)
}
```

io.Writer 类型定义了函数 Fprintf 和这个函数调用者之间的约定。一方面这个约定需要调用者提供具体类型的值就像*os.File 和*bytes.Buffer，这些类型都有一个特定签名和行爲的 Write 的函数。另一方面这个约定 保证了 Fprintf 接受任何满足 io.Writer 接口的值都可以工作。Fprintf 函数可能没有假定写入的是一个文件或 是一段内存，而是写入一个可以调用 Write 函数的值。

因爲 fmt.Fprintf 函数没有对具体操作的值做任何假设而是仅仅通过 io.Writer 接口的约定来保证行爲，所以 第一个参数可以安全地传入一个任何具体类型的值只需要满足 io.Writer 接口。一个类型可以自由的使用另 一个满足相同接口的类型来进行替换被称作可替换性(LSP 里氏替换)。这是一个面向对象的特征。

让我们通过一个新的类型来进行校验，下面*ByteCounter 类型里的 Write 方法，仅仅在丢失写向它的字节 前统计它们的长度。(在这个+=赋值语句中，让 len(p)的类型和*c 的类型匹配的转换是必须的。)

```
// gopl.io/ch7/bytecounter
type ByteCounter int
func (c *ByteCounter) Write(p []byte) (int, error) {
    *c += ByteCounter(len(p)) // convert int to ByteCounter
    return len(p), nil
}
```

因爲*ByteCounter 满足 io.Writer 的约定，我们可以把它传入 Fprintf 函数中；Fprintf 函数执行字符串格式
化的过程不会去关注 ByteCounter 正确的纍加结果的长度。

```
var c ByteCounter
c.Write([]byte("hello"))
fmt.Println(c) // "5", = len("hello")
c = 0 // reset the counter
var name = "Dolly"
fmt.Fprintf(&c, "hello, %s", name)
fmt.Println(c) // "12", = len("hello, Dolly")
```

除了 io.Writer 这个接口类型，还有另一个对 fmt 包很重要的接口类型。Fprintf 和 Fprintln 函数向类型提供了
一种控制它们值输出的途径。在 2.5 节中，我们爲 Celsius 类型提供了一个 String 方法以便于可以打印成这 样"100°C" ，在 6.5 节中我们给*IntSet 添加一个 String 方法，这样集合可以用传统的符号来进行表示就像"
{1 2 3}"。给一个类型定义 String 方法，可以让它满足最广泛使用之一的接口类型 fmt.Stringer：

```
package fmt
// The String method is used to print values passed
// as an operand to any format that accepts a string
// or to an unformatted printer such as Print.
type Stringer interface {
    String() string
}
```

我们会在 7.10 节解释 fmt 包怎麼发现哪些值是满足这个接口类型的。

练习 7.1:使用来自 ByteCounter 的思路，实现一个针对对单词和行数的计数器。你会发现 bufio.ScanWords 非常的有用。

练习 7.2:写一个带有如下函数签名的函数 CountingWriter，传入一个 io.Writer 接口类型，返回一个新的 Writer 类型把原来的 Writer 封装在里面和一个表示写入新的 Writer 字节数的 int64 类型指针

```
func CountingWriter(w io.Writer) (io.Writer, *int64)
```

练习 7.3:爲在 gopl.io/ch4/treesort (§4.4)的*tree 类型实现一个 String 方法去展示 tree 类型的值序列。

接口类型

7.2. 接口类型

接口类型具体描述了一系列方法的集合，一个实现了这些方法的具体类型是这个接口类型的实例。

io.Writer 类型是用的最广泛的接口之一，因爲它提供了所有的类型写入 bytes 的抽象，包括文件类型，内 存缓冲区，网络链接，HTTP 客户端，压缩工具，哈希等等。io 包中定义了很多其它有用的接口类型 。 Reader 可以代表任意可以读取 bytes 的类型，Closer 可以是任意可以关闭的值，例如一个文件或是网络链 接。（到现在你可能注意到了很多 Go 语言中单方法接口的命名习惯）

```
package io
type Reader interface {
    Read(p []byte) (n int, err error)
}
type Closer interface {
    Close() error
}
```

在往下看，我们发现有些新的接口类型通过组合已经有的接口来定义。下面是两个例子：

```
type ReadWriter interface {
    Reader
    Writer
}
type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

上面用到的语法和结构内嵌相似，我们可以用这种方式以一个简写命名另一个接口，而不用声明它所有的方法。这种方式本称爲接口内嵌。尽管略失简洁，我们可以像下面这样，不使用内嵌来声明 io.Writer 接 口。

```
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Write(p []byte) (n int, err error)
}
```

或者甚至使用种混合的风格：

```
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Writer
}
```

上面 3 种定义方式都是一样的效果。方法的顺序变化也没有影响，唯一重要的就是这个集合里面的方法。

练习 7.4:strings.NewReader 函数通过读取一个 string 参数返回一个满足 io.Reader 接口类型的值（和其它 值）。实现一个简单版本的 NewReader，并用它来构造一个接收字符串输入的 HTML 解析器（§5.2）

练习 7.5:io 包里面的 LimitReader 函数接收一个 io.Reader 接口类型的 r 和字节数 n，并且返回另一个从 r 中读 取字节但是当读完 n 个字节后就表示读到文件结束的 Reader。实现这个 LimitReader 函数：

```
func LimitReader(r io.Reader, n int64) io.Reader
```

实现接口的条件

7.3. 实现接口的条件

一个类型如果拥有一个接口需要的所有方法，那麼这个类型就实现了这个接口。例如，*os.File 类型实现了 io.Reader，Writer，Closer，和 ReadWriter 接口*。bytes.Buffer 实现了 Reader，Writer，和 ReadWriter 这些接口，但是它没有实现 Closer 接口因爲它不具有 Close 方法。Go 的程序员经常会简要的把一个具体的 类型描述成一个特定的接口类型。举个例子，*bytes.Buffer 是 io.Writer；*os.Files 是 io.ReadWriter 。

接口指定的规则非常简单：表达一个类型属于某个接口只要这个类型实现这个接口。所以：

```
var w io.Writer
w = os.Stdout          // OK: *os.File has Write method
w = new(bytes.Buffer)  // OK: *bytes.Buffer has Write method
w = time.Second        // compile error: time.Duration lacks Write method

var rwc io.ReadWriteCloser
rwc = os.Stdout        // OK: *os.File has Read, Write, Close methods
rwc = new(bytes.Buffer) // compile error: *bytes.Buffer lacks Close method
```

这个规则甚至适用于等式右边本身也是一个接口类型

```
w = rwc             // OK: io.ReadWriteCloser has Write method
rwc = w             // compile error: io.Writer lacks Close method
```

因爲 ReadWriter 和 ReadWriteCloser 包含所有 Writer 的方法，所以任何实现了 ReadWriter 和 ReadWriteCloser 的类型必定也实现了 Writer 接口

在进一步学习前，必须先解释表示一个类型持有一个方法当中的细节。回想在 6.2 章中，对于每一个命名过 的具体类型 T；它一些方法的接收者是类型 T 本身然而另一些则是一个 *T 的指针。还记得在 T 类型的参数上调 用一个* T 的方法是合法的，只要这个参数是一个变量；编译器隐式的获取了它的地址。但这仅仅是一个语法糖：T 类型的值不拥有所有*T 指针的方法，那这样它就可能只实现更少的接口。

举个例子可能会更清晰一点。在第 6.5 章中，IntSet 类型的 String 方法的接收者是一个指针类型，所以我们 不能在一个不能寻址的 IntSet 值上调用这个方法：

```
type IntSet struct { /* ... */ }
func (*IntSet) String() string
var _ = IntSet{}.String() // compile error: String requires *IntSet receiver
```

但是我们可以在一个 IntSet 值上调用这个方法：

```
var s IntSet
var _ = s.String() // OK: s is a variable and &s has a String method
```

然而，由于只有 *IntSet 类型有 String 方法，所有也只有* IntSet 类型实现了 fmt.Stringer 接口：

```
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s  // compile error: IntSet lacks String method
```

12.8 章包含了一个打印出任意值的所有方法的程序，然后可以使用 godoc -analysis=type tool(§10.7.4)展
示每个类型的方法和具体类型和接口之间的关系

就像信封封装和隐藏信件起来一样，接口类型封装和隐藏具体类型和它的值。即使具体类型有其它的方法也只有接口类型暴露出来的方法会被调用到：

```
os.Stdout.Write([]byte("hello")) // OK: *os.File has Write method
os.Stdout.Close()                // OK: *os.File has Close method

var w io.Writer
w = os.Stdout
w.Write([]byte("hello")) // OK: io.Writer has Write method
w.Close()                // compile error: io.Writer lacks Close method
```

一个有更多方法的接口类型，比如 io.ReadWriter，和少一些方法的接口类型,例如 io.Reader，进行对比;
更多方法的接口类型会告诉我们更多关于它的值持有的信息，并且对实现它的类型要求更加严格。那麼关于 interface{}类型，它没有任何方法，请讲出哪些具体的类型实现了它？

这看上去好像没有用，但实际上 interface{}被称爲空接口类型是不可或缺的。因爲空接口类型对实现它的 类型没有要求，所以我们可以将任意一个值赋给空接口类型。

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer)
```

尽管不是很明显，从本书最早的的例子中我们就已经在使用空接口类型。它允许像 fmt.Println 或者 5.7 章中 的 errorf 函数接受任何类型的参数。

对于创建的一个 interface{}值持有一个 boolean，float，string，map，pointer，或者任意其它的类型； 我们当然不能直接对它持有的值做操作，因爲 interface{}没有任何方法。我们会在 7.10 章中学到一种用类 型断言来获取 interface{}中值的方法。

因爲接口实现只依赖于判断的两个类型的方法，所以没有必要定义一个具体类型和它实现的接口之间的关系。也就是说，尝试文档化和断言这种关系几乎没有用，所以并没有通过程序强制定义。下面的定义在编译期断言一个*bytes.Buffer 的值实现了 io.Writer 接口类型:

```
// *bytes.Buffer must satisfy io.Writer
var w io.Writer = new(bytes.Buffer)
```

因爲任意 *bytes.Buffer 的值，甚至包括 nil 通过*(bytes.Buffer)(nil)进行显示的转换都实现了这个接口，所以
我们不必分配一个新的变量。并且因爲我们絶不会引用变量 w，我们可以使用空标识符来来进行代替。
总 的看，这些变化可以让我们得到一个更朴素的版本：

```
// *bytes.Buffer must satisfy io.Writer
var _ io.Writer = (*bytes.Buffer)(nil)
```

非空的接口类型比如 io.Writer 经常被指针类型实现，尤其当一个或多个接口方法像 Write 方法那样隐式的给接收者带来变化的时候。一个结构体的指针是非常常见的承载方法的类型。

但是并不意味着只有指针类型满足接口类型，甚至连一些有设置方法的接口类型也可能会被 Go 语言中其它 的引用类型实现。我们已经看过 slice 类型的方法(geometry.Path, §6.1)和 map 类型的方法(url.Values,
§6.2.1)，后面还会看到函数类型的方法的例子(http.HandlerFunc, §7.7)。甚至基本的类型也可能会实现一些接口；就如我们在 7.4 章中看到的 time.Duration 类型实现了 fmt.Stringer 接口。

一个具体的类型可能实现了很多不相关的接口。考虑在一个组织出售数字文化產品比如音乐，电影和书籍的程序中可能定义了下列的具体类型：

```
Album
Book
Movie
Magazine
Podcast
TVEpisode
Track
```

我们可以把每个抽象的特点用接口来表示。一些特性对于所有的这些文化產品都是共通的，例如标题，创作日期和作者列表。

```
type Artifact interface {
    Title() string
    Creators() []string
    Created() time.Time
}
```

其它的一些特性只对特定类型的文化產品才有。和文字排版特性相关的只有 books 和 magazines，还有只
有 movies 和 TV 剧集和屏幕分辨率相关。

```
type Text interface {
    Pages() int
    Words() int
    PageSize() int
}
type Audio interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // e.g., "MP3", "WAV"
}
type Video interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // e.g., "MP4", "WMV"
    Resolution() (x, y int)
}
```

这些接口不止是一种有用的方式来分组相关的具体类型和表示他们之间的共同特定。我们后面可能会发现其它的分组。举例，如果我们发现我们需要以同样的方式处理 Audio 和 Video，我们可以定义一个 Streamer 接口来代表它们之间相同的部分而不必对已经存在的类型做改变。

```
type Streamer interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string
}
```

每一个具体类型的组基于它们相同的行爲可以表示成一个接口类型。不像基于类的语言，他们一个类实现的接口集合需要进行显式的定义，在 Go 语言中我们可以在需要的时候定义一个新的抽象或者特定特点的组，而不需要修改具体类型的定义。当具体的类型来自不同的作者时这种方式会特别有用。当然也确实没有必要在具体的类型中指出这些共性。

flag.Value 接口

7.4. flag.Value 接口

在本章，我们会学到另一个标准的接口类型 flag.Value 是怎麼帮助命令行标记定义新的符号的。思考下面 这个会休眠特定时间的程序：

```
// gopl.io/ch7/sleep
var period = flag.Duration("period", 1*time.Second, "sleep period")

func main() {
    flag.Parse()
    fmt.Printf("Sleeping for %v...", *period)
    time.Sleep(*period)
    fmt.Println()
}
```

在它休眠前它会打印出休眠的时间周期。fmt 包调用 time.Duration 的 String 方法打印这个时间周期是以用
户友好的注解方式，而不是一个纳秒数字：

```
$ go build gopl.io/ch7/sleep
$ ./sleep
Sleeping for 1s...
```

默认情况下，休眠周期是一秒，但是可以通过 -period 这个命令行标记来控制。flag.Duration 函数创建一个 time.Duration 类型的标记变量并且允许用户通过多种用户友好的方式来设置这个变量的大小，这种方式 还包括和 String 方法相同的符号排版形式。这种对称设计使得用户交互良好。

```
$ ./sleep -period 50ms
Sleeping for 50ms...
$ ./sleep -period 2m30s
Sleeping for 2m30s...
$ ./sleep -period 1.5h
Sleeping for 1h30m0s...
$ ./sleep -period "1 day"
invalid value "1 day" for flag -period: time: invalid duration 1 day
```

因爲时间周期标记值非常的有用，所以这个特性被构建到了 flag 包中；但是我们爲我们自己的数据类型定义新的标记符号是简单容易的。我们只需要定义一个实现 flag.Value 接口的类型，如下：

```
package flag

// Value is the interface to the value stored in a flag.
type Value interface {
    String() string
    Set(string) error
}
```

String 方法格式化标记的值用在命令行帮组消息中；这样每一个 flag.Value 也是一个 fmt.Stringer。Set 方 法解析它的字符串参数并且更新标记变量的值。实际上，Set 方法和 String 是两个相反的操作，所以最好的 办法就是对他们使用相同的注解方式。

让我们定义一个允许通过摄氏度或者华氏温度变换的形式指定温度的 celsiusFlag 类型。注意 celsiusFlag 内 嵌了一个 Celsius 类型(§2.5)，因此不用实现本身就已经有 String 方法了。爲了实现 flag.Value，我们只需要 定义 Set 方法：

```
// gopl.io/ch7/tempconv
// *celsiusFlag satisfies the flag.Value interface.
type celsiusFlag struct{ Celsius }

func (f *celsiusFlag) Set(s string) error {
    var unit string
    var value float64
    fmt.Sscanf(s, "%f%s", &value, &unit) // no error check needed
    switch unit {
    case "C", "°C":
        f.Celsius = Celsius(value)
        return nil
    case "F", "°F":
        f.Celsius = FToC(Fahrenheit(value))
        return nil
    }
    return fmt.Errorf("invalid temperature %q", s)
}
```

调用 fmt.Sscanf 函数从输入 s 中解析一个浮点数（value）和一个字符串（unit）。虽然通常必须检查 Sscanf 的错误返回，但是在这个例子中我们不需要因爲如果有错误发生，就没有 switch case 会匹配到。

下面的 CelsiusFlag 函数将所有逻辑都封装在一起。它返回一个内嵌在 celsiusFlag 变量 f 中的 Celsius 指针给 调用者。Celsius 字段是一个会通过 Set 方法在标记处理的过程中更新的变量。调用 Var 方法将标记加入应用 的命令行标记集合中，有异常复杂命令行接口的全局变量 flag.CommandLine.Programs 可能有几个这个 类型的变量。调用 Var 方法将一个 *celsiusFlag 参数赋值给一个 flag.Value 参数,导致编译器去检 查* celsiusFlag 是否有必须的方法。

```
// CelsiusFlag defines a Celsius flag with the specified name,
// default value, and usage, and returns the address of the flag variable.
// The flag argument must have a quantity and a unit, e.g., "100C".
func CelsiusFlag(name string, value Celsius, usage string) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return &f.Celsius
}
```

现在我们可以开始在我们的程序中使用新的标记：

```
// gopl.io/ch7/tempflag
var temp = tempconv.CelsiusFlag("temp", 20.0, "the temperature")

func main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

下面是典型的场景：

```
$ go build gopl.io/ch7/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
invalid value "273.15K" for flag -temp: invalid temperature "273.15K"
Usage of ./tempflag:
  -temp value
        the temperature (default 20°C)
$ ./tempflag -help
Usage of ./tempflag:
  -temp value
        the temperature (default 20°C)
```

练习 7.6： 对 tempFlag 加入支持开尔文温度。

练习 7.7： 解释爲什麽帮助信息在它的默认值是 20.0 没有包含°C 的情况下输出了°C。

接口值

7.5. 接口值

概念上讲一个接口的值，接口值，由两个部分组成，一个具体的类型和那个类型的值。它们被称爲接口的动态类型和动态值。对于像 Go 语言这种静态类型的语言，类型是编译期的概念；因此一个类型不是一个值。在我们的概念模型中，一些提供每个类型信息的值被称爲类型描述符，比如类型的名称和方法。在一个接口值中，类型部分代表与之相关类型的描述符。

下面 4 个语句中，变量 w 得到了 3 个不同的值。（开始和最后的值是相同的）

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

让我们进一步观察在每一个语句后的 w 变量的值和动态行爲。第一个语句定义了变量 w:

```
var w io.Writer
```

在 Go 语言中，变量总是被一个定义明确的值初始化，即使接口类型也不例外。对于一个接口的零值就是它
的类型和值的部分都是 nil（图 7.1）。

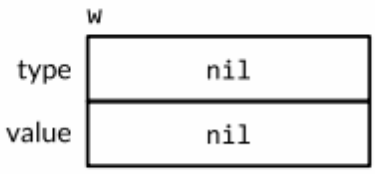


Figure 7.1. A nil interface value.

一个接口值基于它的动态类型被描述爲空或非空，所以这是一个空的接口值。你可以通过使用 w==nil 或者 w!=nil 来判读接口值是否爲空。调用一个空接口值上的任意方法都会産生 panic:

```
w.Write([]byte("hello")) // panic: nil pointer dereference
```

第二个语句将一个*os.File 类型的值赋给变量 w:

```
w = os.Stdout
```

这个赋值过程调用了一个具体类型到接口类型的隐式转换，这和显式的使用 io.Writer(os.Stdout)是等价的。这类转换不管是显式的还是隐式的，都会刻畫出操作到的类型和值。这个接口值的动态类型被设爲 *os.Stdout 指针的类型描述符，它的动态值持有 os.Stdout 的拷贝；这是一个代表处理标准输出的 os.File 类 型变量的指针（图 7.2）。

Figure 7.2. An interface value containing an `*os.File` pointer.

调用一个包含*os.File 类型指针的接口值的 Write 方法，使得(*os.File).Write 方法被调用。这个调用输 出 “hello” 。

```
w.Write([]byte("hello")) // "hello"
```

通常在编译期，我们不知道接口值的动态类型是什麽，所以一个接口上的调用必须使用动态分配。因爲不是直接进行调用，所以编译器必须把代码生成在类型描述符的方法 Write 上，然后间接调用那个地址。这个 调用的接收者是一个接口动态值的拷贝，os.Stdout。效果和下面这个直接调用一样：

```
os.Stdout.Write([]byte("hello")) // "hello"
```

第三个语句给接口值赋了一个*bytes.Buffer 类型的值

```
w = new(bytes.Buffer)
```

现在动态类型是*bytes.Buffer 并且动态值是一个指向新分配的缓冲区的指针（图 7.3）。

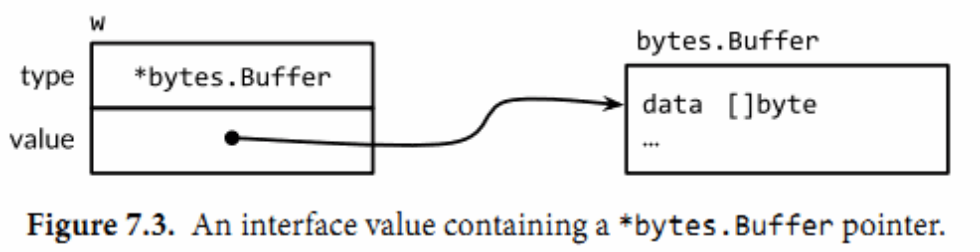


Figure 7.3. An interface value containing a *bytes.Buffer pointer.

Write 方法的调用也使用了和之前一样的机制：

```
w.Write([]byte("hello")) // writes "hello" to the bytes.Buffers
```

这次类型描述符是*bytes.Buffer，所以调用了(*bytes.Buffer).Write 方法，并且接收者是该缓冲区的地址。这个调用把字符串 “hello” 添加到缓冲区中。

最后，第四个语句将 nil 赋给了接口值：

```
w = nil
```

这个重置将它所有的部分都设爲 nil 值，把变量 w 恢复到和它之前定义时相同的状态图，在图 7.1 中可以看
到。

一个接口值可以持有任意大的动态值。例如，表示时间实例的 time.Time 类型，这个类型有几个对外不公 开的字段。我们从它上面创建一个接口值,

```
var x interface{} = time.Now()
```

结果可能和图 7.4 相似。从概念上讲，不论接口值多大，动态值总是可以容下它。（这只是一个概念上的模
型；具体的实现可能会非常不同）

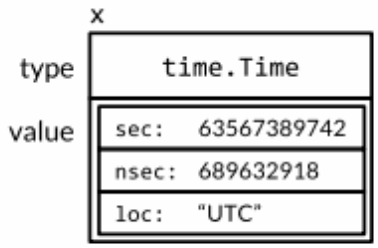


Figure 7.4. An interface value holding a `time.Time` struct.

接口值可以使用==和!=来进行比较。两个接口值相等仅当它们都是 nil 值或者它们的动态类型相同并且动态值也根据这个动态类型的==操作相等。因爲接口值是可比较的，所以它们可以用在 map 的键或者作 爲 switch 语句的操作数。

然而，如果两个接口值的动态类型相同，但是这个动态类型是不可比较的（比如切片），将它们进行比较就会失败并且 panic:

```
var x interface{} = []int{1, 2, 3}
fmt.Println(x == x) // panic: comparing uncomparable type []int
```

考虑到这点，接口类型是非常与衆不同的。其它类型要麽是安全的可比较类型（如基本类型和指针）要麽是完全不可比较的类型（如切片，映射类型，和函数），但是在比较接口值或者包含了接口值的聚合类型时，我们必须要意识到潜在的 panic。同样的风险也存在于使用接口作爲 map 的键或者 switch 的操作数。 只能比较你非常确定它们的动态值是可比较类型的接口值。

当我们处理错误或者调试的过程中，得知接口值的动态类型是非常有帮助的。所以我们使用 fmt 包的%T 动 作:

```
var w io.Writer
fmt.Printf("%T\n", w) // "<nil>"
w = os.Stdout
fmt.Printf("%T\n", w) // "*os.File"
w = new(bytes.Buffer)
fmt.Printf("%T\n", w) // "*bytes.Buffer"
```

在 fmt 包内部，使用反射来获取接口动态类型的名称。我们会在第 12 章中学到反射相关的知识。

{% include "./ch7-05-1.md" %}

sort.Interface 接口

7.6. sort.Interface 接口

TODO

http.Handler 接口

7.7. http.Handler 接口

TODO

error 接口

7.8. error 接口

TODO

示例: 表达式求值

7.9. 示例: 表达式求值

TODO

类型断言

7.10. 类型断言

TODO

基于类型断言识别错误类型

7.11. 基于类型断言识别错误类型

TODO

通过类型断言查询接口

7.12. 通过类型断言查询接口

TODO

类型分支

7.13. 类型分支

TODO

示例: 基于标记的 XML 译码

7.14. 示例: 基于标记的 XML 译码

TODO

补充几点

7.15. 补充几点

TODO

Goroutines 和 Channels

第八章 Goroutines 和 Channels

并发程序指的是同时做好几件事情的程序，随着硬件的发展，并发程序显得越来越重要。Web 服务器会一 次处理成韆上万的请求。平板计算机和手机 app 在渲染用户动畫的同时，还会后台执行各种计算任务和网络 请求。即使是传统的批处理问题--读取数据，计算，写输出--现在也会用并发来隐藏掉 I/O 的操作延迟充分 利用现代计算机设备的多核，尽管计算机的性能每年都在增长，但并不是线性。

Go 语言中的并发程序可以用两种手段来实现。这一章会讲解 goroutine 和 channel，其支持“顺序进程通 信”(communicating sequential processes)或被简称爲 CSP。CSP 是一个现代的并发编程模型，在这种 编程模型中值会在不同的运行实例(goroutine)中传递，尽管大多数情况下被限制在单一实例中。第 9 章会 覆盖到更爲传统的并发模型：多线程共享内存，如果你在其它的主流语言中写过并发程序的话可能会更熟 悉一些。第 9 章同时会讲一些本章不会深入的并发程序带来的重要风险和陷阱。

尽管 Go 对并发的支持是衆多强力特性之一，但大多数情况下跟踪并发程序还是很睏难，并且在线性程序中 我们的直觉往往还会让我们误入歧途。如果这是你第一次接触并发，那麽我推荐你稍微多花一些时间来思 考这两个章节中的样例。

Goroutines

8.1. Goroutines

在 Go 语言中，每一个并发的执行单元叫作一个 goroutine。设想这里有一个程序有两个函数，一个函数做 一些计算，另一个输出一些结果，假设两个函数没有相互之间的调用关系。一个线性的程序会先调用其中 的一个函数，然后再调用来一个，但如果是在有两个甚至更多个 goroutine 的程序中，对两个函数的调用 就可以在同一时间。我们马上就会看到这样的一个程序。

如果你使用过操作系统或者其它语言提供的线程，那麽你可以简单地把 goroutine 类比作一个线程，这样 你就可以写出一些正确的程序了。goroutine 和线程的本质区别会在 9.8 节中讲。

当一个程序启动时，其主函数即在一个单独的 goroutine 中运行，我们叫它 main goroutine。新的 goroutine 会用 go 语句来创建。在语法上，go 语句是一个普通的函数或方法调用前加上关键字 go。go 语 句会使其语句中的函数在一个新创建的 goroutine 中运行。而 go 语句本身会迅速地完成。

```
f()    // call f(); wait for it to return
go f() // create a new goroutine that calls f(); don't wait
```

在下面的例子中，main goroutine 会计算第 45 个菲波那契数。由于计算函数使用了效率非常低的递归，所

以会运行相当可观的一段时间，在这期间我们想要让用户看到一个可见的标识来表明程序依然在正常运行，所以显示一个动畫的小图标：

```
gopl.io/ch8/spinner
func main() {
    go spinner(100 * time.Millisecond)
    const n = 45
    fibN := fib(n) // slow
    fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}

func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/` {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}
```

动畫显示了几秒之后，fib(45)的调用成功地返回，并且打印结果：

```
Fibonacci(45) = 1134903170
```

然后主函数返回。当主函数返回时，所有的 goroutine 都会直接打断，程序退出。除了从主函数退出或者直接退出程序之外，没有其它的编程方法能够让一个 goroutine 来打断另一个的执行，但是我们之后可以看到，可以通过 goroutine 之间的通信来让一个 goroutine 请求请求其它的 goroutine，并让其自己结束执 行。

注意这里的两个独立的单元是如何进行组合的，spinning 和菲波那契的计算。每一个都是写在独立的函数 中，但是每一个函数都会并发地执行。

示例: 并发的 Clock 服务

8.2. 示例: 并发的 Clock 服务

网络编程是并发大显身手的一个领域，由于服务器是最典型的需要同时处理很多连接的程序，这些连接一

般来自远彼此独立的客户端。在本小节中，我们会讲解 go 语言的 net 包，这个包提供编写一个网络客户端 或者服务器程序的基本组件，无论两者间通信是使用 TCP，UDP 或者 Unix domain sockets。在第一章中 我们已经使用过的 net/http 包里的方法，也算是 net 包的一部分。

我们的第一个例子是一个顺序执行的时钟服务器，它会每隔一秒钟将当前时间写到客户端：

```
gopl.io/ch8/clock1
// Clock1 is a TCP server that periodically writes the time.
package main

import (
    "io"
    "log"
    "net"
    "time"
)

func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }

    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err) // e.g., connection aborted
            continue
        }
        handleConn(conn) // handle one connection at a time
    }
}

func handleConn(c net.Conn) {
    defer c.Close()
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return // e.g., client disconnected
        }
        time.Sleep(1 * time.Second)
    }
}
```

Listen 函数创建了一个 net.Listener 的对象，这个对象会监听一个网络端口上到来的连接，在这个例子里我
们用的是 TCP 的 localhost:8000 端口。listener 对象的 Accept 方法会直接阻塞，直到一个新的连接被创建， 然后会返回一个 net.Conn 对象来表示这个连接。

handleConn 函数会处理一个完整的客户端连接。在一个 for 死循环中，将当前的时候用 time.Now()函数得 到，然后写到客户端。由于 net.Conn 实现了 io.Writer 接口，我们可以直接向其写入内容。这个死循

直执行，直到写入失败。最可能的原因是客户端主动断开连接。这种情况下 handleConn 函数会用 defer 调 用关闭服务器侧的连接，然后返回到主函数，继续等待下一个连接请求。

time.Time.Format 方法提供了一种格式化日期和时间信息的方式。它的参数是一个格式化模板标识如何来 格式化时间，而这个格式化模板限定爲 Mon Jan 2 03:04:05PM 2006 UTC-0700。有 8 个部分(周几，月 份，一个月的第几天，等等)。可以以任意的形式来组合前面这个模板；出现在模板中的部分会作爲参考来 对时间格式进行输出。在上面的例子中我们只用到了小时、分钟和秒。time 包里定义了很多标准时间格 式，比如 time.RFC1123。在进行格式化的逆向操作 time.Parse 时，也会用到同样的策略。(译注：这是 go 语言和其它语言相比比较奇葩的一个地方。。你需要记住格式化字符串是 1 月 2 日下午 3 点 4 分 5 秒零六年 UTC-0700，而不像其它语言那样 Y-m-d H:i:s 一样，当然了这里可以用 1234567 的方式来记忆，倒是也不 麻烦)

爲了连接例子里的服务器，我们需要一个客户端程序，比如 netcat 这个工具(nc 命令)，这个工具可以用来 执行网络连接操作。

```
$ go build gopl.io/ch8/clock1
$ ./clock1 &
$ nc localhost 8000
13:58:54
13:58:55
13:58:56
13:58:57
^C
```

客户端将服务器发来的时间显示了出来，我们用 Control+C 来中断客户端的执行，在 Unix 系统上，你会看
到^C 这样的响应。如果你的系统没有装 nc 这个工具，你可以用 telnet 来实现同样的效果，或者也可以用我 们下面的这个用 go 写的简单的 telnet 程序，用 net.Dial 就可以简单地创建一个 TCP 连接：

```
gopl.io/ch8/netcat1
// Netcat1 is a read-only TCP client.
package main

import (
    "io"
    "log"
    "net"
    "os"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    mustCopy(os.Stdout, conn)
}

func mustCopy(dst io.Writer, src io.Reader) {
    if _, err := io.Copy(dst, src); err != nil {
        log.Fatal(err)
    }
}
```

这个程序会从连接中读取数据，并将读到的内容写到标准输出中，直到遇到 end of file 的条件或者发生错误。mustCopy 这个函数我们在本节的几个例子中都会用到。让我们同时运行两个客户端来进行一个测 试，这里可以开两个终端窗口，下面左边的是其中的一个的输出，右边的是另一个的输出：

```
$ go build gopl.io/ch8/netcat1
$ ./netcat1
13:58:54                        $ ./netcat1
13:58:55
13:58:56
^C
                                13:58:57
                                13:58:58
                                13:58:59
                                ^C
$ killall clock1
```

killall 命令是一个 Unix 命令行工具，可以用给定的进程名来杀掉所有名字匹配的进程。

第二个客户端必须等待第一个客户端完成工作，这样服务端才能继续向后执行；因爲我们这里的服务器程序同一时间只能处理一个客户端连接。我们这里对服务端程序做一点小改动，使其支持并发：在 handleConn 函数调用的地方增加 go 关键字，让每一次 handleConn 的调用都进入一个独立的 goroutine。

```
gopl.io/ch8/clock2
for {
    conn, err := listener.Accept()
    if err != nil {
        log.Print(err) // e.g., connection aborted
        continue
    }
    go handleConn(conn) // handle connections concurrently
}
```

现在多个客户端可以同时接收到时间了：

```
$ go build gopl.io/ch8/clock2
$ ./clock2 &
$ go build gopl.io/ch8/netcat1
$ ./netcat1
14:02:54                    $ ./netcat1
14:02:55                    14:02:55
14:02:56                    14:02:56
14:02:57                    ^C
14:02:58
14:02:59                    $ ./netcat1
14:03:00                    14:03:00
14:03:01                    14:03:01
^C                        14:03:02
                        ^C
$ killall clock2
```

练习 8.1: 修改 clock2 来支持传入参数作爲端口号，然后写一个 clockwall 的程序，这个程序可以同时与多个
clock 服务器通信，从多服务器中读取时间，并且在一个表格中一次显示所有服务传回的结果，类似于你在 某些办公室里看到的时钟墙。如果你有地理学上分布式的服务器可以用的话，让这些服务器跑在不同的机 器上面；或者在同一台机器上跑多个不同的实例，这些实例监听不同的端口，假装自己在不同的时区。像 下面这样：

```
$ TZ=US/Eastern    ./clock2 -port 8010 &
$ TZ=Asia/Tokyo    ./clock2 -port 8020 &
$ TZ=Europe/London ./clock2 -port 8030 &
$ clockwall NewYork=localhost:8010 Tokyo=localhost:8020 London=localhost:8030
```

练习 8.2: 实现一个并发 FTP 服务器。服务器应该解析客户端来的一些命令，比如 cd 命令来切换目録，ls 来列
出目録内文件，get 和 send 来传输文件，close 来关闭连接。你可以用标准的 ftp 命令来作爲客户端，或者也 可以自己实现一个。

示例: 并发的 Echo 服务

8.3. 示例: 并发的 Echo 服务

clock 服务器每一个连接都会起一个 goroutine。在本节中我们会创建一个 echo 服务器，这个服务在每个连 接中会有多个 goroutine。大多数 echo 服务仅仅会返回他们读取到的内容，就像下面这个简单的 handleConn 函数所做的一样：

```
func handleConn(c net.Conn) {
    io.Copy(c, c) // NOTE: ignoring errors
    c.Close()
}
```

一个更有意思的 echo 服务应该模拟一个实际的 echo 的“回响”，并且一开始要用大写 HELLO 来表示“声
音很大”，之后经过一小段延迟返回一个有所缓和的 Hello，然后一个全小写字母的 hello 表示声音渐渐变 小直至消失，像下面这个版本的 handleConn(译注：笑看作者脑洞大开)：

```
gopl.io/ch8/reverb1
func echo(c net.Conn, shout string, delay time.Duration) {
    fmt.Fprintln(c, "\t", strings.ToUpper(shout))
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", shout)
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", strings.ToLower(shout))
}

func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        echo(c, input.Text(), 1*time.Second)
    }
    // NOTE: ignoring potential errors from input.Err()
    c.Close()
}
```

我们需要陞级我们的客户端程序，这样它就可以发送终端的输入到服务器，并把服务端的返回输出到终端上，这使我们有了使用并发的另一个好机会：

```
gopl.io/ch8/netcat2
func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    go mustCopy(os.Stdout, conn)
    mustCopy(conn, os.Stdin)
}
```

当 main goroutine 从标准输入流中读取内容并将其发送给服务器时，另一个 goroutine 会读取并打印服务 端的响应。当 main goroutine 碰到输入终止时，例如，用户在终端中按了 Control-D(^D)，在 windows 上 是 Control-Z，这时程序就会被终止，尽管其它 goroutine 中还有进行中的任务。(在 8.4.1 中引入了 channels 后我们会明白如何让程序等待两边都结束)。

下面这个会话中，客户端的输入是左对齐的，服务端的响应会用缩进来区别显示。
客户端会向服务器“喊三次话”：

```
$ go build gopl.io/ch8/reverb1
$ ./reverb1 &
$ go build gopl.io/ch8/netcat2
$ ./netcat2
Hello?
   HELLO?
   Hello?
   hello?
Is there anybody there?
   IS THERE ANYBODY THERE?
Yooo-hooo!
   Is there anybody there?
   is there anybody there?
   YOOO-HOOO!
   Yooo-hooo!
yooo-hooo!
^D
$ killall reverb1
```

注意客户端的第三次 shout 在前一个 shout 处理完成之前一直没有被处理，这貌似看起来不是特别“现实”。眞实世界里的回响应该是会由三次 shout 的回声组合而成的。爲了模拟眞实世界的回响，我们需要更 多的 goroutine 来做这件事情。这样我们就再一次地需要 go 这个关键词了，这次我们用它来调用 echo：

```
gopl.io/ch8/reverb2
func handleConn(c net.Conn) {
   input := bufio.NewScanner(c)
   for input.Scan() {
      go echo(c, input.Text(), 1*time.Second)
   }
   // NOTE: ignoring potential errors from input.Err()
   c.Close()
}
```

go 后跟的函数的参数会在 go 语句自身执行时被求值；因此 input.Text()会在 main goroutine 中被求值。
现在回响是并发并且会按时间来覆盖掉其它响应了：

```
$ go build gopl.io/ch8/reverb2
$ ./reverb2 &
$ ./netcat2
Is there anybody there?
    IS THERE ANYBODY THERE?
Yooo-hooo!
    Is there anybody there?
    YOOO-HOOO!
    is there anybody there?
    Yooo-hooo!
    yooo-hooo!
^D
$ killall reverb2
```

让服务使用并发不只是处理多个客户端的请求，甚至在处理单个连接时也可能会用到，就像我们上面的两个 go 关键词的用法。然而在我们使用 go 关键词的同时，需要慎重地考虑 net.Conn 中的方法在并发地调用 时是否安全，事实上对于大多数类型来说也确实不安全。我们会在下一章中详细地探讨并发安全性。

Channels

8.4. Channels

如果说 goroutine 是 Go 语音程序的并发体的话，那麼 channels 它们之间的通信机制。一个 channels 是一个 通信机制，它可以让一个 goroutine 通过它给另一个 goroutine 发送值信息。每个 channel 都有一个特殊的 类型，也就是 channels 可发送数据的类型。一个可以发送 int 类型数据的 channel 一般写爲 chan int。

使用内置的 make 函数，我们可以创建一个 channel：

```
ch := make(chan int) // ch has type 'chan int'
```

和 map 类似，channel 也一个对应 make 创建的底层数据结构的引用。当我吗复制一个 channel 或用于函数
参数传递时，我吗只是拷贝了一个 channel 引用，因此调用者何被调用者将引用同一个 channel 对象。和其 它的引用类型一样，channel 的零值也是 nil。

两个相同类型的 channel 可以使用==运算符比较。如果两个 channel 引用的是相通的对象，那麼比较的结 果爲眞。一个 channel 也可以和 nil 进行比较。

一个 channel 有发送和接受两个主要操作，都是通信行爲。一个发送语句将一个值从一个 goroutine 通过 channel 发送到另一个执行接收操作的 goroutine。发送和接收两个操作都是用`<-`运算符。在发送语句中，`<-`运算符分割 channel 和要发送的值。在接收语句中，`<-`运算符写在 channel 对象之前。一个不使用接收结果的接收操作也是合法的。

```
ch <- x  // a send statement
x = <-ch // a receive expression in an assignment statement
<-ch     // a receive statement; result is discarded
```

Channel 还支持 close 操作，用于关闭 channel，随后对基于该 channel 的任何发送操作都将导致 panic 异
常。对一个已经被 close 过的 channel 之行接收操作依然可以接受到之前已经成功发送的数据；如
果 channel 中已经没有数据的话讲产生一个零值的数据。

使用内置的 close 函数就可以关闭一个 channel：

```
close(ch)
```

以最简单方式调用 make 函数创建的时一个无缓存的 channel，但是我们也可以指定第二个整形参数，对应
channel 的容量。如果 channel 的容量大于零，那麽该 channel 就是带缓存的 channel。

```
ch = make(chan int)    // unbuffered channel
ch = make(chan int, 0) // unbuffered channel
ch = make(chan int, 3) // buffered channel with capacity 3
```

我们将先讨论无缓存的 channel，然后在 8.4.4 节讨论带缓存的 channel。

{% include "./ch8-04-1.md" %}

{% include "./ch8-04-2.md" %}

{% include "./ch8-04-3.md" %}

{% include "./ch8-04-4.md" %}

并发的循环

8.5. 并发的循环

本节中，我们会探索一些用来在并行时循环迭代的常见并发模型。我们会探究从全尺寸图片生成一些缩略图的问题。gopl.io/ch8/thumbnail 包提供了 ImageFile 函数来帮我们拉伸图片。我们不会帮助这个函数的 实现，只需要从 gopl.io 下载它。

```
gopl.io/ch8/thumbnail
package thumbnail
// ImageFile reads an image from infile and writes
// a thumbnail-size version of it in the same directory.
// It returns the generated file name, e.g., "foo.thumb.jpg".
func ImageFile(infile string) (string, error)
```

下面的程序会循环迭代一些图片文件名，并爲每一张图片生成一个缩略图：

```
gopl.io/ch8/thumbnail
// makeThumbnails makes thumbnails of the specified files.
func makeThumbnails(filenames []string) {
    for _, f := range filenames {
        if _, err := thumbnail.ImageFile(f); err != nil {
            log.Println(err)
        }
    }
}
```

显然我们处理文件的顺序无关紧要，因爲每一个图片的拉伸操作和其它图片的处理操作都是彼此独立的。像这种子问题都是完全彼此独立的问题被叫做易并行问题(译注：embarrassingly parallel，直译的话更像是尴尬并行)。易并行问题是最容易被实现成并行的一类问题(废话)，并且是最能够享受并发带来的好处，能够随着并行的规模线性地扩展。

下面让我们并行地执行这些操作，从而将文件 IO 的延迟隐藏掉，并用上多核 cpu 的计算能力来拉伸图像。 我们的第一个并发程序只是使用了一个 go 关键字。这里我们先忽略掉错误，之后再进行处理。

```
// NOTE: incorrect!
func makeThumbnails2(filenames []string) {
    for _, f := range filenames {
        go thumbnail.ImageFile(f) // NOTE: ignoring errors
    }
}
```

这个版本运行的实在有点太快，实际上，由于它比最早的版本使用的时间要短得多，即使当文件名的 slice 中只包含有一个元素。这就有点奇怪了，如果程序没有并发执行的话，那爲什麽一个并发的版本还是要快呢？答案其实是 makeThumbnails 在它还没有完成工作之前就已经返回了。它启动了所有的 goroutine， 没一个文件名对应一个，但没有等待它们一直到执行完毕。

没有什麽直接的办法能够等待 goroutine 完成，但是我们可以改变 goroutine 里的代码让其能够将完成情况 报告给外部的 goroutine 知晓，使用的方式是向一个共享的 channel 中发送事件。因爲我们已经知道内部的 goroutine 只有 len(filenames)，所以外部的 goroutine 只需要在返回之前对这些事件计数。

```
// makeThumbnails3 makes thumbnails of the specified files in parallel.
func makeThumbnails3(filenames []string) {
    ch := make(chan struct{})
    for _, f := range filenames {
        go func(f string) {
            thumbnail.ImageFile(f) // NOTE: ignoring errors
            ch <- struct{}{}
        }(f)
    }
    // Wait for goroutines to complete.
    for range filenames {
        <-ch
    }
}
```

注意我们将 f 的值作爲一个显式的变量传给了函数，而不是在循环的闭包中声明：

```
for _, f := range filenames {
    go func() {
        thumbnail.ImageFile(f) // NOTE: incorrect!
        // ...
    }()
}
```

回忆一下之前在 5.6.1 节中，匿名函数中的循环变量快照问题。上面这个单独的变量 f 是被所有的匿名函数值
所共享，且会被连续的循环迭代所更新的。当新的 goroutine 开始执行字面函数时，for 循环可能已经更新 了 f 并且开始了另一轮的迭代或者(更有可能的)已经结束了整个循环，所以当这些 goroutine 开始读取 f 的值 时，它们所看到的值已经是 slice 的最后一个元素了。显式地添加这个参数，我们能够确保使用的 f 是当 go 语 句执行时的“当前”那个 f。

如果我们想要从每一个 worker goroutine 往主 goroutine 中返回值时该怎麽办呢？当我们调用 thumbnail.ImageFile 创建文件失败的时候，它会返回一个错误。下一个版本的 makeThumbnails 会返回 其在做拉伸操作时接收到的第一个错误：

```
// makeThumbnails4 makes thumbnails for the specified files in parallel.
// It returns an error if any step failed.
func makeThumbnails4(filenames []string) error {
    errors := make(chan error)

    for _, f := range filenames {
        go func(f string) {
            _, err := thumbnail.ImageFile(f)
            errors <- err
        }(f)}
    }

    for range filenames {
        if err := <-errors; err != nil {
            return err // NOTE: incorrect: goroutine leak!
        }
    }

    return nil
}
```

这个程序有一个微秒的 bug。当它遇到第一个非 nil 的 error 时会直接将 error 返回到调用方，使得没有一个
goroutine 去排空 errors channel。这样剩下的 worker goroutine 在向这个 channel 中发送值时，都会永远 地阻塞下去，并且永远都不会退出。这种情况叫做 goroutine 泄露(§8.4.4)，可能会导致整个程序卡住或者 跑出 out of memory 的错误。

最简单的解决办法就是用一个具有合适大小的 buffered channel，这样这些 worker goroutine 向 channel 中发送测向时就不会被阻塞。(一个可选的解决办法是创建一个另外的 goroutine，当 main goroutine 返回 第一个错误的同时去排空 channel)

下一个版本的 makeThumbnails 使用了一个 buffered channel 来返回生成的图片文件的名字，附带生成时 的错误。

```
// makeThumbnails5 makes thumbnails for the specified files in parallel.
// It returns the generated file names in an arbitrary order,
// or an error if any step failed.

func makeThumbnails5(filenames []string) (thumbfiles []string, err error) {
    type item struct {
        thumbfile string
        err       error
    }

    ch := make(chan item, len(filenames))
    for _, f := range filenames {
        go func(f string) {
            var it item
            it.thumbfile, it.err = thumbnail.ImageFile(f)
            ch <- it
        }(f)
    }

    for range filenames {
        it := <-ch
        if it.err != nil {
            return nil, it.err
        }
        thumbfiles = append(thumbfiles, it.thumbfile)
    }

    return thumbfiles, nil
}
```

我们最后一个版本的 makeThumbnails 返回了新文件们的大小总计数(bytes)。和前面的版本都不一样的一

点是我们在这个版本里没有把文件名放在 slice 里，而是通过一个 string 的 channel 传过来，所以我们无法对 循环的次数进行预测。

爲了知道最后一个 goroutine 什麽时候结束(最后一个结束并不一定是最后一个开始)，我们需要一个递增的 计数器，在每一个 goroutine 启动时加一，在 goroutine 退出时减一。这需要一种特殊的计数器，这个计数 器需要在多个 goroutine 操作时做到安全并且提供提供在其减爲零之前一直等待的一种方法。这种计数类 型被称爲 sync.WaitGroup，下面的代码就用到了这种方法：

```
// makeThumbnails6 makes thumbnails for each file received from the channel.
// It returns the number of bytes occupied by the files it creates.
func makeThumbnails6(filenames <-chan string) int64 {
    sizes := make(chan int64)
    var wg sync.WaitGroup // number of working goroutines
    for f := range filenames {
        wg.Add(1)
        // worker
        go func(f string) {
            defer wg.Done()
            thumb, err := thumbnail.ImageFile(f)
            if err != nil {
                log.Println(err)
                return
            }
            info, _ := os.Stat(thumb) // OK to ignore error
            sizes <- info.Size()
        }(f)
    }

    // closer
    go func() {
        wg.Wait()
        close(sizes)
    }()


    var total int64
    for size := range sizes {
        total += size
    }
    return total
}
```

注意 Add 和 Done 方法的不对策。Add 是爲计数器加一，必须在 worker goroutine 开始之前调用，而不是
在 goroutine 中；否则的话我们没办法确定 Add 是在"closer" goroutine 调用 Wait 之前被调用。并且 Add 还 有一个参数，但 Done 却没有任何参数；其实它和 Add(-1)是等价的。我们使用 defer 来确保计数器即使是 在出错的情况下依然能够正确地被减掉。上面的程序代码结构是当我们使用并发循环，但又不知道迭代次 数时很通常而且很地道的写法。

sizes channel 携带了每一个文件的大小到 main goroutine，在 main goroutine 中使用了 range loop 来计 算总和。观察一下我们是怎样创建一个 closer goroutine，并让其等待 worker 们在关闭掉 sizes channel 之 前退出的。两步操作：wait 和 close，必须是基于 sizes 的循环的并发。考虑一下另一种方案：如果等待操作 被放在了 main goroutine 中，在循环之前，这样的话就永远都不会结束了，如果在循环之后，那麽又变成 了不可达的部分，因爲没有任何东西去关闭这个 channel，这个循环就永远都不会终止。

图 8.5 表明了 makethumbnails6 函数中事件的序列。纵列表示 goroutine。窄线段代表 sleep，粗线段代表 活动。斜线箭头代表用来同步两个 goroutine 的事件。时间向下流动。注意 main goroutine 是如何

的时间被唤醒执行其 range 循环，等待 worker 发送值或者 closer 来关闭 channel 的。

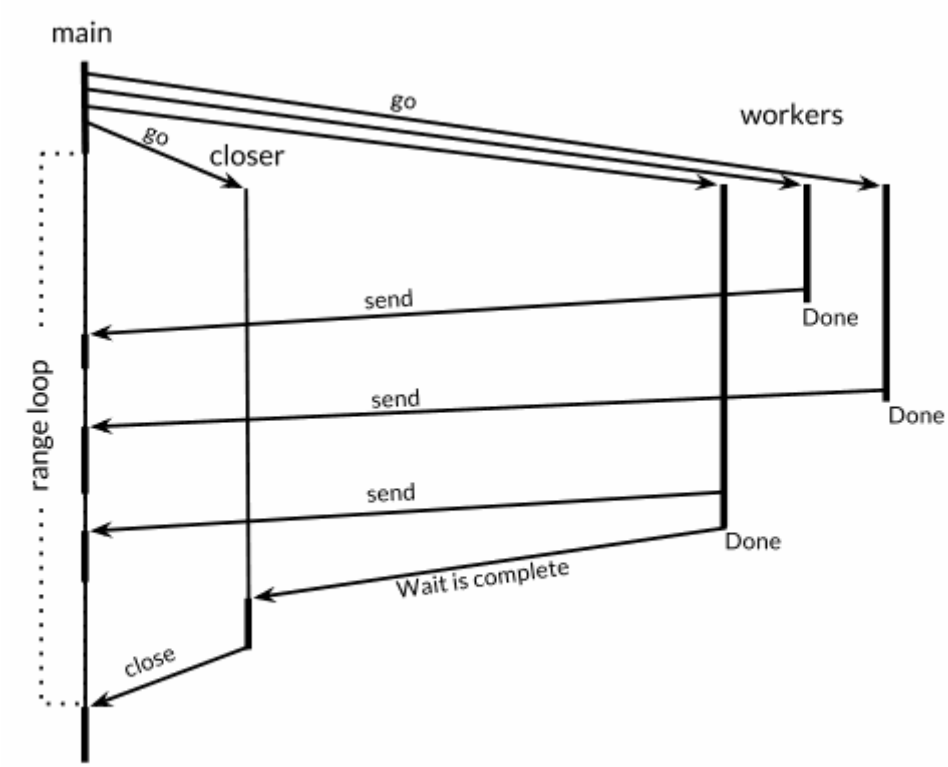


Figure 8.5. The sequence of events in `makeThumbnails6`.

练习 8.4: 修改 reverb2 服务器，在每一个连接中使用 sync.WaitGroup 来计数活跃的 echo goroutine。当计 数减爲零时，关闭 TCP 连接的写入，像练习 8.3 中一样。验证一下你的修改版 netcat3 客户端会一直等待所 有的并发“喊叫”完成，即使是在标准输入流已经关闭的情况下。

练习 8.5: 使用一个已有的 CPU 绑定的顺序程序，比如在 3.3 节中我们写的 Mandelbrot 程序或者 3.2 节中的 3- D surface 计算程序，并将他们的主循环改爲并发形式，使用 channel 来进行通信。在多核计算机上这个程 序得到了多少速度上的改进？使用多少个 goroutine 是最合适的呢？

示例: 并发的 Web 爬虫

8.6. 示例: 并发的 Web 爬虫

在 5.6 节中，我们做了一个简单的 web 爬虫，用 bfs(广度优先)算法来抓取整个网站。在本节中，我们会让这 个这个爬虫并行化，这样每一个彼此独立的抓取命令可以并行进行 IO，最大化利用网络资源。crawl 函数和 gopl.io/ch5/findlinks3 中的是一样的。

```
gopl.io/ch8/crawl1
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

主函数和 5.6 节中的 breadthFirst(深度优先)类似。像之前一样，一个 worklist 是一个记録了需要处理的元素
的队列，每一个元素都是一个需要抓取的 URL 列表，不过这一次我们用 channel 代替 slice 来做这个队列。 每一个对 crawl 的调用都会在他们自己的 goroutine 中进行并且会把他们抓到的链接发送回 worklist。

```
func main() {
    worklist := make(chan []string)

    // Start with the command-line arguments.
    go func() { worklist <- os.Args[1:] }()

    // Crawl the web concurrently.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

注意这里的 crawl 所在的 goroutine 会将 link 作爲一个显式的参数传入，来避免“循环变量快照”的问题(在
5.6.1 中有讲解)。另外注意这里将命令行参数传入 worklist 也是在一个另外的 goroutine 中进行的，这是爲 了避免在 main goroutine 和 crawler goroutine 中同时向另一个 goroutine 通过 channel 发送内容时发生死 锁(因爲另一边的接收操作还没有准备好)。当然，这里我们也可以用 buffered channel 来解决问题，这里 不再赘述。

现在爬虫可以高并发地运行起来，并且可以産生一大坨的 URL 了，不过还是会有俩问题。一个问题是在运 行一段时间后可能会出现在 log 的错误信息里的：

```
$ go build gopl.io/ch8/crawl1
$ ./crawl1 http://gopl.io/
http://gopl.io/
https://golang.org/help/
https://golang.org/doc/
https://golang.org/blog/
...
2015/07/15 18:22:12 Get ...: dial tcp: lookup blog.golang.org: no such host
2015/07/15 18:22:12 Get ...: dial tcp 23.21.222.120:443: socket:
                                    too many open files
...
```

最初的错误信息是一个让人莫名的 DNS 查找失败，即使这个域名是完全可靠的。而随后的错误信息揭示了原因：这个程序一次性创建了太多网络连接，超过了每一个进程的打开文件数限制，既而导致了在调用 net.Dial 像 DNS 查找失败这样的问题。

这个程序实在是太他妈并行了。无穷无尽地并行化并不是什麽好事情，因爲不管怎麽说，你的系统总是会有一个些限制因素，比如 CPU 核心数会限制你的计算负载，比如你的硬盘转轴和磁头数限制了你的本地磁 盘 IO 操作频率，比如你的网络带宽限制了你的下载速度上限，或者是你的一个 web 服务的服务容量上限等 等。爲了解决这个问题，我们可以限制并发程序所使用的资源来使之适应自己的运行环境。对于我们的例 子来说，最简单的方法就是限制对 links.Extract 在同一时间最多不会有超过 n 次调用，这里的 n 是 fd 的 limit- 20，一般情况下。这个一个夜店里限制客人数目是一个道理，只有当有客人离开时，才会允许新的客人进 入店内(译注：作者你个老流氓)。

我们可以用一个有容量限制的 buffered channel 来控制并发，这类似于操作系统里的计数信号量概念。从概念上讲，channel 里的 n 个空槽代表 n 个可以处理内容的 token(通行证)，从 channel 里接收一个值会释放 其中的一个 token，并且生成一个新的空槽位。这样保证了在没有接收介入时最多有 n 个发送操作。(这里可 能我们拿 channel 里填充的槽来做 token 更直观一些，不过还是这样吧~)。由于 channel 里的元素类型并不 重要，我们用一个零值的 struct{}来作爲其元素。

让我们重写 crawl 函数，将对 links.Extract 的调用操作用获取、释放 token 的操作包裹起来，来确保同一时 间对其只有 20 个调用。信号量数量和其能操作的 IO 资源数量应保持接近。

```
gopl.io/ch8/crawl2
// tokens is a counting semaphore used to
// enforce a limit of 20 concurrent requests.
var tokens = make(chan struct{}, 20)

func crawl(url string) []string {
    fmt.Println(url)
    tokens <- struct{}{} // acquire a token
    list, err := links.Extract(url)
    <-tokens // release the token
    if err != nil {
        log.Print(err)
    }
    return list
}
```

第二个问题是这个程序永远都不会终止，即使它已经爬到了所有初始链接衍生出的链接。(当然，除非你慎重地选择了合适的初始化 URL 或者已经实现了练习 8.6 中的深度限制，你应该还没有意识到这个问题)。爲了 使这个程序能够终止，我们需要在 worklist 爲空或者没有 crawl 的 goroutine 在运行时退出主循环。

```
func main() {
    worklist := make(chan []string)
    var n int // number of pending sends to worklist

    // Start with the command-line arguments.
    n++
    go func() { worklist <- os.Args[1:] }()


    // Crawl the web concurrently.
    seen := make(map[string]bool)

    for ; n > 0; n-- {
        list := <-worklist
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                n++
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

这个版本中，计算器 n 对 worklist 的发送操作数量进行了限制。每一次我们发现有元素需要被发送到 worklist 时，我们都会对 n 进行++操作，在向 worklist 中发送初始的命令行参数之前，我们也进行过一次
++操作。这里的操作++是在每启动一个 crawler 的 goroutine 之前。主循环会在 n 减爲 0 时终止，这时

候说

明没活可躲了。

现在这个并发爬虫会比 5.6 节中的深度优先搜索版快上 20 倍，而且不会出什麼错，并且在其完成任务时也会 正确地终止。

下面的程序是避免过度并发的另一种思路。这个版本使用了原来的 crawl 函数，但没有使用计数信号量，取 而代之用了 20 个长活的 crawler goroutine，这样来保证最多 20 个 HTTP 请求在并发。

```
func main() {
    worklist := make(chan []string) // lists of URLs, may have duplicates
    unseenLinks := make(chan string) // de-duplicated URLs

    // Add command-line arguments to worklist.
    go func() { worklist <- os.Args[1:] }()

    // Create 20 crawler goroutines to fetch each unseen link.
    for i := 0; i < 20; i++ {
        go func() {
            for link := range unseenLinks {
                foundLinks := crawl(link)
                go func() { worklist <- foundLinks }()
            }
        }()
    }

    // The main goroutine de-duplicates worklist items
    // and sends the unseen ones to the crawlers.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                unseenLinks <- link
            }
        }
    }
}
```

所有的爬虫 goroutine 现在都是被同一个 channel-unseenLinks 喂饱的了。主 goroutine 负责拆分它从 worklist 里拿到的元素，然后把没有抓过的经由 unseenLinks channel 发送给一个爬虫的 goroutine。

seen 这个 map 被限定在 main goroutine 中；也就是说这个 map 只能在 main goroutine 中进行访问 。类似 于其它的信息隐藏方式，这样的约束可以让我们从一定程度上保证程序的正确性。例如，内部变量不能够 在函数外部被访问到；变量(§2.3.4)在没有被转义的情况下是无法在函数外部访问的；一个对象的封装字段 无法被该对象的方法以外的方法访问到。在所有的情况下，信息隐藏都可以帮助我们约束我们的程序，使 其不发生意料之外的情况。

crawl 函数爬到的链接在一个专有的 goroutine 中被发送到 worklist 中来避免死锁。爲了节省空间，这个例 子的终止问题我们先不进行详细阐述了。

练习 8.6: 爲并发爬虫增加深度限制。也就是说，如果用户设置了 depth=3，那麽只有从首页跳转三次以内 能够跳到的页面才能被抓取到。

练习 8.7: 完成一个并发程序来创建一个在线网站的本地镜像，把该站点的所有可达的页面都抓取到本地硬盘。爲了省事，我们这里可以只取出现在该域下的所有页面(比如 golang.org 结尾，译注：外链的应该就不 算了。)当然了，出现在页面里的链接你也需要进行一些处理，使其能够在你的镜像站点上进行跳转，而不 是指向原始的链接。

译注： 拓
展閱读：
http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/

基于 select 的多路复用

8.7. 基于 select 的多路复用

下面的程序会进行火箭发射的倒计时。time.Tick 函数返回一个 channel，程序会周期性地像一个节拍器一 样向这个 channel 发送事件。每一个事件的值是一个时间戳，不过更有意思的是其传送方式。

```
gopl.io/ch8/countdown1
func main() {
    fmt.Println("Commencing countdown.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        j<-tick
    }
    launch()
}
```

现在我们让这个程序支持在倒计时中，用户按下 return 键时直接中断发射流程。首先，我们启动一个 goroutine，这个 goroutine 会尝试从标准输入中调入一个单独的 byte 并且，如果成功了，会向名爲 abort 的 channel 发送一个值。

```
gopl.io/ch8/countdown2
abort := make(chan struct{})
go func() {
    os.Stdin.Read(make([]byte, 1)) // read a single byte
    abort <- struct{}{}
}()
```

现在每一次计数循环的迭代都需要等待两个 channel 中的其中一个返回事件了：ticker channel 当一切正常
时(就像 NASA jorgon 的"nominal"，译注：这梗估计我们是不懂了)或者异常时返回的 abort 事件。我们无

法做到从每一个 channel 中接收信息，如果我们这麽做的话，如果第一个 channel 中没有事件发过来那麽程 序就会立刻被阻塞，这样我们就无法收到第二个 channel 中发过来的事件。这时候我们需要多路复用(multiplex)这些操作了，爲了能够多路复用，我们使用了 select 语句。

```
select {
case <-ch1:
    // ...
case x := <-ch2:
    // ...use x...
case ch3 <- y:
    // ...
default:
    // ...
}
```

上面是 select 语句的一般形式。和 switch 语句稍微有点相似，也会有几个 case 和最后的 default 选择支。每
一个 case 代表一个通信操作(在某个 channel 上进行发送或者接收)并且会包含一些语句组成的一个语句块。 一个接收表达式可能只包含接收表达式自身(译注：不把接收到的值赋值给变量什麽的)，就像上面的第一个 case，或者包含在一个简短的变量声明中，像第二个 case 里一样；第二种形式让你能够引用接收到的值。

select 会等待 case 中有能够执行的 case 时去执行。当条件满足时，select 才会去通信并执行 case 之后的语 句；这时候其它通信是不会执行的。一个没有任何 case 的 select 语句写作 select{}，会永远地等待下去。

让我们回到我们的火箭发射程序。time.After 函数会立即返回一个 channel，并起一个新的 goroutine 在经 过特定的时间后向该 channel 发送一个独立的值。下面的 select 语句会会一直等待到两个事件中的一个到 达，无论是 abort 事件或者一个 10 秒经过的事件。如果 10 秒经过了还没有 abort 事件进入，那麽火箭就会发 射。

```
func main() {
    // ...create abort channel...

    fmt.Println("Commencing countdown. Press return to abort.")
    select {
    case <-time.After(10 * time.Second):
        // Do nothing.
    case <-abort:
        fmt.Println("Launch aborted!")
        return
    }
    launch()
}
```

下面这个例子更微秒。ch 这个 channel 的 buffer 大小是 1，所以会交替的爲空或爲满，所以只有一个 case 可
以进行下去，无论 i 是奇数或者偶数，它都会打印 0 2 4 6 8。

```
ch := make(chan int, 1)
for i := 0; i < 10; i++ {
    select {
    case x := <-ch:
        fmt.Println(x) // "0" "2" "4" "6" "8"
    case ch <- i:
    }
}
```

如果多个 case 同时就绪时，select 会随机地选择一个执行，这样来保证每一个 channel 都有平等的被 select
的机会。增加前一个例子的 buffer 大小会使其输出变得不确定，因爲当 buffer 既不爲满也不爲空时，select 语句的执行情况就象是抛硬币的行爲一样是随机的。

下面让我们的发射程序打印倒计时。这里的 select 语句会使每次循环迭代等待一秒来执行退出操作。

```
gopl.io/ch8/countdown3
func main() {
    // ...create abort channel...

    fmt.Println("Commencing countdown. Press return to abort.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        select {
        case <-tick:
            // Do nothing.
        case <-abort:
            fmt.Println("Launch aborted!")
            return
        }
    }
    launch()
}
```

time.Tick 函数表现得好像它创建了一个在循环中调用 time.Sleep 的 goroutine，每次被唤醒时发送一个事件。当 countdown 函数返回时，它会停止从 tick 中接收事件，但是 ticker 这个 goroutine 还依然存活，继续 徒劳地尝试从 channel 中发送值，然而这时候已经没有其它的 goroutine 会从该 channel 中接收值了--这被 称爲 goroutine 泄露(§8.4.4)。

Tick 函数挺方便，但是只有当程序整个生命周期都需要这个时间时我们使用它才比较合适。否则的话，我 们应该使用下面的这种模式：

```
ticker := time.NewTicker(1 * time.Second)
<-ticker.C    // receive from the ticker's channel
ticker.Stop() // cause the ticker's goroutine to terminate
```

有时候我们希望能够从 channel 中发送或者接收值，并避免因爲发送或者接收导致的阻塞，尤其是当

channel 没有准备好写或者读时。select 语句就可以实现这样的功能。select 会有一个 default 来设置当其它 的操作都不能够马上被处理时程序需要执行哪些逻辑。

下面的 select 语句会在 abort channel 中有值时，从其中接收值；无值时什麼都不做。这是一个非阻塞的接 收操作；反复地做这样的操作叫做 “轮询 channel” 。

```
select {
case <-abort:
    fmt.Printf("Launch aborted!\n")
    return
default:
    // do nothing
}
```

channel 的零值是 nil。也许会让你觉得比较奇怪，nil 的 channel 有时候也是有一些用处的。因爲对一个 nil
的 channel 发送和接收操作会永远阻塞，在 select 语句中操作 nil 的 channel 永远都不会被 select 到。

这使得我们可以用 nil 来激活或者禁用 case，来达成处理其它输入或输出事件时超时和取消的逻辑。我们会 在下一节中看到一个例子。

练习 8.8: 使用 select 来改造 8.3 节中的 echo 服务器，爲其增加超时，这样服务器可以在客户端 10 秒中没有任 何喊话时自动断开连接。

示例: 并发的字典遍历

8.8. 示例: 并发的字典遍历

在本小节中，我们会创建一个程序来生成指定目録的硬盘使用情况报告，这个程序和 Unix 里的 du 工具比较 相似。大多数工作用下面这个 walkDir 函数来完成，这个函数使用 dirents 函数来枚举一个目録下的所有入 口。

```
gopl.io/ch8/du1
// walkDir recursively walks the file tree rooted at dir
// and sends the size of each found file on fileSizes.
func walkDir(dir string, fileSizes chan<- int64) {
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            subdir := filepath.Join(dir, entry.Name())
            walkDir(subdir, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}

// dirents returns the entries of directory dir.
func dirents(dir string) []os.FileInfo {
    entries, err := ioutil.ReadDir(dir)
    if err != nil {
        fmt.Fprintf(os.Stderr, "du1: %v\n", err)
        return nil
    }
    return entries
}
```

ioutil.ReadDir 函数会返回一个 os.FileInfo 类型的 slice，os.FileInfo 类型也是 os.Stat 这个函数的返回值。对
每一个子目録而言，walkDir 会递归地调用其自身，并且会对每一个文件也递归调用。walkDir 函数会向 fileSizes 这个 channel 发送一条消息。这条消息包含了文件的字节大小。

下面的主函数，用了两个 goroutine。后台的 goroutine 调用 walkDir 来遍历命令行给出的每一个路径并最 终关闭 fileSizes 这个 channel。主 goroutine 会对其从 channel 中接收到的文件大小进行纍加，并输出其 和。

```
package main

import (
    "flag"
    "fmt"
    "io/ioutil"
    "os"
    "path/filepath"
)

func main() {
    // Determine the initial directories.
    flag.Parse()
    roots := flag.Args()
    if len(roots) == 0 {
        roots = []string{"."}
    }

    // Traverse the file tree.
    fileSizes := make(chan int64)
    go func() {
        for _, root := range roots {
            walkDir(root, fileSizes)
        }
        close(fileSizes)
    }()

    // Print the results.
    var nfiles, nbytes int64
    for size := range fileSizes {
        nfiles++
        nbytes += size
    }
    printDiskUsage(nfiles, nbytes)
}

func printDiskUsage(nfiles, nbytes int64) {
   fmt.Printf("%d files  %.1f GB\n", nfiles, float64(nbytes)/1e9)
}
```

这个程序会在打印其结果之前卡住很长时间。

```
$ go build gopl.io/ch8/du1
$ ./du1 $HOME /usr /bin /etc
213201 files  62.7 GB
```

如果在运行的时候能够让我们知道处理进度的话想必更好。但是，如果简单地把 printDiskUsage 函数调用
移动到循环里会导致其打印出成百上韆的输出。 下面这个 du 的变种会间歇打印内容，不过只有在调用时

提供了-v 的 flag 才会显示程序进度信息。在 roots 目

録上循环的后台 goroutine 在这里保持不变。主 goroutine 现在使用了计时器来每 500ms 生成事件，然后用 select 语句来等待文件大小的消息来更新总大小数据，或者一个计时器的事件来打印当前的总大小数据。 如果-v 的 flag 在运行时没有传入的话，tick 这个 channel 会保持爲 nil，这样在 select 里的 case 也就相当于被 禁用了。

```
gopl.io/ch8/du2
var verbose = flag.Bool("v", false, "show verbose progress messages")

func main() {
    // ...start background goroutine...

    // Print the results periodically.
    var tick <-chan time.Time
    if *verbose {
        tick = time.Tick(500 * time.Millisecond)
    }
    var nfiles, nbytes int64
loop:
    for {
        select {
        case size, ok := <-fileSizes:
            if !ok {
                break loop // fileSizes was closed
            }
            nfiles++
            nbytes += size
        case <-tick:
            printDiskUsage(nfiles, nbytes)
        }
    }
    printDiskUsage(nfiles, nbytes) // final totals
}
```

由于我们的程序不再使用 range 循环，第一个 select 的 case 必须显式地判断 fileSizes 的 channel 是不是已经
被关闭了，这里可以用到 channel 接收的二值形式。如果 channel 已经被关闭了的话，程序会直接退出循 环。这里的 break 语句用到了标签 break，这样可以同时终结 select 和 for 两个循环；如果没有用标签就 break 的话只会退出内层的 select 循环，而外层的 for 循环会使之进入下一轮 select 循环。

现在程序会悠闲地爲我们打印更新流：

```
$ go build gopl.io/ch8/du2
$ ./du2 -v $HOME /usr /bin /etc
28608 files  8.3 GB
54147 files  10.3 GB
93591 files  15.1 GB
127169 files  52.9 GB
175931 files  62.2 GB
213201 files  62.7 GB
```

然而这个程序还是会花上很长时间才会结束。无法对 walkDir 做并行化处理没什麽别的原因，无非是因爲磁 盘系统并行限制。下面这个第三个版本的 du，会对每一个 walkDir 的调用创建一个新的 goroutine。它使用 sync.WaitGroup (§8.5)来对仍旧活跃的 walkDir 调用进行计数，另一个 goroutine 会在计数器减爲零的时 候将 fileSizes 这个 channel 关闭。

```
gopl.io/ch8/du3
func main() {
    // ...determine roots...
    // Traverse each root of the file tree in parallel.
    fileSizes := make(chan int64)
    var n sync.WaitGroup
    for _, root := range roots {
        n.Add(1)
        go walkDir(root, &n, fileSizes)
    }
    go func() {
        n.Wait()
        close(fileSizes)
    }()
    // ...select loop...
}

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            n.Add(1)
            subdir := filepath.Join(dir, entry.Name())
            go walkDir(subdir, n, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}
```

由于这个程序在高峯期会创建成百上韆的 goroutine，我们需要脩改 dirents 函数，用计数信号量来阻止他
同时打开太多的文件，就像我们在 8.7 节中的并发爬虫一样：

```
// sema is a counting semaphore for limiting concurrency in dirents.
var sema = make(chan struct{}, 20)

// dirents returns the entries of directory dir.
func dirents(dir string) []os.FileInfo {
    sema <- struct{}{}        // acquire token
    defer func() { <-sema }() // release token
    // ...
```

这个版本比之前那个快了好几倍，尽管其具体效率还是和你的运行环境，机器配置相关。

练习 8.9: 编写一个 du 工具，每隔一段时间将 root 目録下的目録大小计算并显示出来。

并发的退出

8.9. 并发的退出

有时候我们需要通知 goroutine 停止它正在辚的事情，比如一个正在执行计算的 web 服务，然而它的客户端已经断开了和服务端的连接。

Go 语言并没有提供在一个 goroutine 中终止另一个 goroutine 的方法，由于这样会导致 goroutine 之间的共 享变量落在未定义的状态上。在 8.7 节中的 rocket launch 程序中，我们往名字叫 abort 的 channel 里发送了 一个简单的值，在 countdown 的 goroutine 中会把这个值理解爲自己的退出信号。但是如果我们想要退出 两个或者任意多个 goroutine 怎麽办呢？

一种可能的手段是向 abort 的 channel 里发送和 goroutine 数目一样多的事件来退出它们。如果这些 goroutine 中已经有一些自己退出了，那麽会导致我们的 channel 里的事件数比 goroutine 还多，这样导致 我们的发送直接被阻塞。另一方面，如果这些 goroutine 又生成了其它的 goroutine，我们的 channel 里的 数目又太少了，所以有些 goroutine 可能会无法接收到退出消息。一般情况下我们是很难知道在某一个时 刻具体有多少个 goroutine 在运行着的。另外，当一个 goroutine 从 abort channel 中接收到一个值的时 候，他会消费掉这个值，这样其它的 goroutine 就没法看到这条信息。爲了能够达到我们退出 goroutine 的 目的，我们需要更靠谱的策略，来通过一个 channel 把消息广播出去，这样 goroutine 们能够看到这条事件 消息，并且在事件完成之后，可以知道这件事已经发生过了。

回忆一下我们关闭了一个 channel 并且被消费掉了所有已发送的值，操作 channel 之后的代码可以立卽被执 行，并且会産生零值。我们可以将这个机制扩展一下，来作爲我们的广播机制：不要向 channel 发送值， 而是用关闭一个 channel 来进行广播。

只要一些小脩改，我们就可以把退出逻辑加入到前一节的 du 程序。首先，我们创建一个退出的 channel， 这个 channel 不会向其中发送任何值，但其所在的闭包内要写明程序需要退出。我们同时还定义了一个工 具函数，cancelled，这个函数在被调用的时候会轮询退出状态。

```
gopl.io/ch8/du4
var done = make(chan struct{})

func cancelled() bool {
    select {
    case <-done:
        return true
    default:
        return false
    }
}
```

下面我们创建一个从标准输入流中读取内容的 goroutine，这是一个比较典型的连接到终端的程序。每当有输入被读到(比如用户按了回车键)，这个 goroutine 就会把取消消息通过关闭 done 的 channel 广播出去。

```
// Cancel traversal when input is detected.
go func() {
    os.Stdin.Read(make([]byte, 1)) // read a single byte
    close(done)
}()
```

现在我们需要使我们的 goroutine 来对取消进行响应。在 main goroutine 中，我们添加了 select 的第三个
case 语句，尝试从 done channel 中接收内容。如果这个 case 被满足的话，在 select 到的时候即会返回，但 在结束之前我们需要把 fileSizes channel 中的内容“排”空，在 channel 被关闭之前，舍弃掉所有值。这样 可以保证对 walkDir 的调用不要被向 fileSizes 发送信息阻塞住，可以正确地完成。

```
for {
    select {
    case <-done:
        // Drain fileSizes to allow existing goroutines to finish.
        for range fileSizes {
            // Do nothing.
        }
        return
    case size, ok := <-fileSizes:
        // ...
    }
}
```

walkDir 这个 goroutine 一启动就会轮询取消状态，如果取消状态被设置的话会直接返回，并且不做额外的
事情。这样我们将所有在取消事件之后创建的 goroutine 改变爲无操作。

```
func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    if cancelled() {
        return
    }
    for _, entry := range dirents(dir) {
        // ...
    }
}
```

在 walkDir 函数的循环中我们对取消状态进行轮询可以带来明显的益处，可以避免在取消事件发生时还去创建 goroutine。取消本身是有一些代价的；想要快速的响应需要对程序逻辑进行侵入式的脩改。确保在取消发生之后不要有代价太大的操作可能会需要脩改你代码里的很多地方，但是在一些重要的地方去检查取消事件也确实能带来很大的好处。

对这个程序的一个简单的性能分析可以揭示瓶颈在 dirents 函数中获取一个信号量。下面的 select 可以让这

种操作可以被取消，并且可以将取消时的延迟从几百毫秒降低到几十毫秒。

```
func dirents(dir string) []os.FileInfo {
    select {
    case sema <- struct{}{}: // acquire token
    case <-done:
        return nil // cancelled
    }
    defer func() { <-sema }() // release token
    // ...read directory...
}
```

现在当取消发生时，所有后台的 goroutine 都会迅速停止并且主函数会返回。当然，当主函数返回时，一个程序会退出，而我们又无法在主函数退出的时候确认其已经释放了所有的资源(译注：因爲程序都退出了，你的代码都没法执行了)。这里有一个方便的窍门我们可以一用：取代掉直接从主函数返回，我们调用一个 panic，然后 runtime 会把每一个 goroutine 的栈 dump 下来。如果 main goroutine 是唯一一个剩下的 goroutine 的话，他会清理掉自己的一切资源。但是如果还有其它的 goroutine 没有退出，他们可能没办法 被正确地取消掉，也有可能被取消但是取消操作会很花时间；所以这里的一个调研还是很有必要的。我们 用 panic 来获取到足够的信息来验证我们上面的判断，看看最终到底是什麽样的情况。

练习 8.10: HTTP 请求可能会因 http.Request 结构体中 Cancel channel 的关闭而取消。修改 8.6 节中的 web crawler 来支持取消 http 请求。

提示: http.Get 并没有提供方便地定制一个请求的方法。你可以用 http.NewRequest 来取而代之，设置它的 Cancel 字段，然后用 http.DefaultClient.Do(req)来进行这个 http 请求。

练习 8.11:紧接着 8.4.4 中的 mirroredQuery 流程，实现一个并发请求 url 的 fetch 的变种。当第一个请求返回 时，直接取消其它的请求。

示例: 聊天服务

8.10. 示例: 聊天服务

我们用一个聊天服务器来终结本章节的内容，这个程序可以让一些用户通过服务器向其它所有用户广播文本消息。这个程序中有四种 goroutine。main 和 broadcaster 各自是一个 goroutine 实例，每一个客户端的 连接都会有一个 handleConn 和 clientWriter 的 goroutine。broadcaster 是 select 用法的不错的样例，因爲 它需要处理三种不同类型的消息。
下面演示的 main goroutine 的工作，是 listen 和 accept(译注：网络编程里的概念)从客户端过来的连接。 对每一个连接，程序都会建立一个新的 handleConn 的 goroutine，就像我们在本章开头的并发的 echo 服 务器里所做的那样。

```
gopl.io/ch8/chat
func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    go broadcaster()
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err)
            continue
        }
        go handleConn(conn)
    }
}
```

然后是 broadcaster 的 goroutine。他的内部变量 clients 会记録当前建立连接的客户端集合。其记録的内容
是每一个客户端的消息发出 channel 的"资格"信息。

```
type client chan<- string // an outgoing message channel

var (
    entering = make(chan client)
    leaving  = make(chan client)
    messages = make(chan string) // all incoming client messages
)

func broadcaster() {
    clients := make(map[client]bool) // all connected clients
    for {
        select {
        case msg := <-messages:
            // Broadcast incoming message to all
            // clients' outgoing message channels.
            for cli := range clients {
                cli <- msg
            }
        case cli := <-entering:
            clients[cli] = true

        case cli := <-leaving:
            delete(clients,  cli)
            close(cli)
        }
    }
}
```

broadcaster 监听来自全局的 entering 和 leaving 的 channel 来获知客户端的到来和离开事件。当其接收到

其中的一个事件时，会更新 clients 集合，当该事件是离开行爲时，它会关闭客户端的消息发出 channel。

broadcaster 也会监听全局的消息 channel，所有的客户端都会向这个 channel 中发送消息。当 broadcaster 接收到什麽消息时，就会将其广播至所有连接到服务端的客户端。

现在让我们看看每一个客户端的 goroutine。handleConn 函数会爲它的客户端创建一个消息发出 channel 并通过 entering channel 来通知客户端的到来。然后它会读取客户端发来的每一行文本，并通过全局的消 息 channel 来将这些文本发送出去，并爲每条消息带上发送者的前缀来标明消息身份。当客户端发送完毕 后，handleConn 会通过 leaving 这个 channel 来通知客户端的离开并关闭连接。

```
func handleConn(conn net.Conn) {
    ch := make(chan string) // outgoing client messages
    go clientWriter(conn, ch)

    who := conn.RemoteAddr().String()
    ch <- "You are " + who
    messages <- who + " has arrived"
    entering <- ch

    input := bufio.NewScanner(conn)
    for input.Scan() {
        messages <- who + ": " + input.Text()
    }
    // NOTE: ignoring potential errors from input.Err()

    leaving <- ch
    messages <- who + " has left"
    conn.Close()
}

func clientWriter(conn net.Conn, ch <-chan string) {
    for msg := range ch {
    fmt.Fprintln(conn, msg) // NOTE: ignoring network errors
    }
}
```

另外，handleConn 爲每一个客户端创建了一个 clientWriter 的 goroutine 来接收向客户端发出消息 channel 中发送的广播消息，并将它们写入到客户端的网络连接。客户端的读取方循环会在 broadcaster 接 收到 leaving 通知并关闭了 channel 后终止。

下面演示的是当服务器有两个活动的客户端连接，并且在两个窗口中运行的情况，使用 netcat 来聊天：

```
$ go build gopl.io/ch8/chat
$ go build gopl.io/ch8/netcat3
$ ./chat &
$ ./netcat3
You are 127.0.0.1:64208              $ ./netcat3
127.0.0.1:64211 has arrived           You are 127.0.0.1:64211
Hi!
127.0.0.1:64208: Hi!
127.0.0.1:64208: Hi!
                                Hi yourself.
127.0.0.1:64211: Hi yourself.         127.0.0.1:64211: Hi yourself.
^C
                                127.0.0.1:64208 has left
$ ./netcat3
You are 127.0.0.1:64216              127.0.0.1:64216 has arrived
                                Welcome.
127.0.0.1:64211: Welcome.             127.0.0.1:64211: Welcome.
                                ^C
127.0.0.1:64211 has left"
```

当与 n 个客户端保持聊天 session 时，这个程序会有 2n+2 个并发的 goroutine，然而这个程序却并不需要显
式的锁(§9.2)。clients 这个 map 被限制在了一个独立的 goroutine 中，broadcaster，所以它不能被并发地 访问。多个 goroutine 共享的变量只有这些 channel 和 net.Conn 的实例，两个东西都是并发安全的。我们 会在下一章中更多地解决约束，并发安全以及 goroutine 中共享变量的含义。

练习 8.12: 使 broadcaster 能够将 arrival 事件通知当前所有的客户端。爲了达成这个目的，你需要有一个客 户端的集合，并且在 entering 和 leaving 的 channel 中记録客户端的名字。

练习 8.13: 使聊天服务器能够断开空闲的客户端连接，比如最近五分钟之后没有发送任何消息的那些客户 端。提示：可以在其它 goroutine 中调用 conn.Close()来解除 Read 调用，就像 input.Scanner()所做的那 样。

练习 8.14: 脩改聊天服务器的网络协议这样每一个客户端就可以在 entering 时可以提供它们的名字。将消息 前缀由之前的网络地址改爲这个名字。

练习 8.15: 如果一个客户端没有及时地读取数据可能会导致所有的客户端被阻塞。脩改 broadcaster 来跳过 一条消息，而不是等待这个客户端一直到其准备好写。或者爲每一个客户端的消息发出 channel 建立缓冲 区，这样大部分的消息便不会被丢掉；broadcaster 应该用一个非阻塞的 send 向这个 channel 中发消息。

基于共享变量的并发

第九章 基于共享变量的并发

前一章我们介绍了一些使用 goroutine 和 channel 这样直接而自然的方式来实现并发的方法。然而这样做我 们实际上屏蔽掉了在写并发代码时必须处理的一些重要而且细微的问题。

在本章中，我们会细致地了解并发机制。尤其是在多 goroutine 之间的共享变量，并发问题的分析手段， 以及解决这些问题的基本模式。最后我们会解释 goroutine 和操作系统线程之间的技术上的一些区别。

竞争条件

9.1. 竞争条件

TODO

sync.Mutex 互斥锁

9.2. sync.Mutex 互斥锁

TODO

sync.RWMutex 读写锁

9.3. sync.RWMutex 读写锁

在 100 刀的存款消失时不做记録多少还是会让我们有一些恐慌，Bob 写了一个程序，每秒运行几百次来检查 他的银行余额。他会在家，在工作中，甚至会在他的手机上来运行这个程序。银行注意到这些陡增的流量 使得存款和取款有了延时，因爲所有的余额查询请求是顺序执行的，这样会互斥地获得锁，并且会暂时阻 止其它的 goroutine 运行。

由于 Balance 函数只需要读取变量的状态，所以我们同时让多个 Balance 调用并发运行事实上是安全的，只 要在运行的时候没有存款或者取款操作就行。在这种场景下我们需要一种特殊类型的锁，其允许多个只读 操作并行执行，但写操作会完全互斥。这种锁叫作“多读单写”锁(multiple readers, single writer

lock)，Go 语言提供的这样的锁是 sync.RWMutex：

```
var mu sync.RWMutex
var balance int
func Balance() int {
    mu.RLock() // readers lock
    defer mu.RUnlock()
    return balance
}
```

Balance 函数现在调用了 RLock 和 RUnlock 方法来获取和释放一个读取或者共享锁。Deposit 函数没有变化，会调用 mu.Lock 和 mu.Unlock 方法来获取和释放一个写或互斥锁。

在这次修改后，Bob 的余额查询请求就可以彼此并行地执行并且会很快地完成了。锁在更多的时间范围可用，并且存款请求也能够及时地被响应了。

RLock 只能在临界区共享变量没有任何写入操作时可用。一般来说，我们不应该假设逻辑上的只读函数/方 法也不会去更新某一些变量。比如一个方法功能是访问一个变量，但它也有可能会同时去给一个内部的计 数器+1(译注：可能是记録这个方法的访问次数啥的)，或者去更新缓存--使即时的调用能够更快。如果有 疑惑的话，请使用互斥锁。

RWMutex 只有当获得锁的大部分 goroutine 都是读操作，而锁在竞争条件下，也就是说，goroutine 们必 须等待才能获取到锁的时候，RWMutex 才是最能带来好处的。RWMutex 需要更复杂的内部记録，所以会 让它比一般的无竞争锁的 mutex 慢一些。

内存同步

9.4. 内存同步

你可能比较纠结爲什麼 Balance 方法需要用到互斥条件，无论是基于 channel 还是基于互斥量。毕竟和存款 不一样，它只由一个简单的操作组成，所以不会碰到其它 goroutine 在其执行"中"执行其它的逻辑的风险。 这里使用 mutex 有两方面考虑。第一 Balance 不会在其它操作比如 Withdraw “中间” 执行。第二(更重要) 的是"同步"不仅仅是一堆 goroutine 执行顺序的问题；同样也会涉及到内存的问题。

在现代计算机中可能会有一堆处理器，每一个都会有其本地缓存(local cache)。爲了效率，对内存的写入一般会在每一个处理器中缓冲，并在必要时一起 flush 到主存。这种情况下这些数据可能会以与当初 goroutine 写入顺序不同的顺序被提交到主存。像 channel 通信或者互斥量操作这样的原语会使处理器将其 聚集的写入 flush 并 commit，这样 goroutine 在某个时间点上的执行结果才能被其它处理器上运行的 goroutine 得到。

考虑一下下面代码片段的可能输出：

```
var x, y int
go func() {
    x = 1 // A1
    fmt.Print("y:", y, " ") // A2
}()
go func() {
    y = 1                   // B1
    fmt.Print("x:", x, " ") // B2
}()
```

因爲两个 goroutine 是并发执行，并且访问共享变量时也没有互斥，会有数据竞争，所以程序的运行结果没法预测的话也请不要惊讶。我们可能希望它能够打印出下面这四种结果中的一种，相当于几种不同的交错执行时的情况：

```
y:0 x:1
x:0 y:1
x:1 y:1
y:1 x:1
```

第四行可以被解释爲执行顺序 A1,B1,A2,B2 或者 B1,A1,A2,B2 的执行结果。
然而实际的运行时还是有些情况让我们有点惊讶：

```
x:0 y:0
y:0 x:0
```

但是根据所使用的编译器，CPU，或者其它很多影响因子，这两种情况也是有可能发生的。那麽这两种情况要怎麽解释呢？

在一个独立的 goroutine 中，每一个语句的执行顺序是可以被保证的；也就是说 goroutine 是顺序连贯的。 但是在不使用 channel 且不使用 mutex 这样的显式同步操作时，我们就没法保证事件在不同的 goroutine 中 看到的执行顺序是一致的了。尽管 goroutine A 中一定需要观察到 x=1 执行成功之后才会去读取 y，但它没 法确保自己观察得到 goroutine B 中对 y 的写入，所以 A 还可能会打印出 y 的一个旧版的值。

尽管去理解并发的一种尝试是去将其运行理解爲不同 goroutine 语句的交错执行，但看看上面的例子，这 已经不是现代的编译器和 cpu 的工作方式了。因爲赋值和打印指向不同的变量，编译器可能会断定两条语 句的顺序不会影响执行结果，并且会交换两个语句的执行顺序。如果两个 goroutine 在不同的 CPU 上执 行，每一个核心有自己的缓存，这样一个 goroutine 的写入对于其它 goroutine 的 Print，在主存同步之前 就是不可见的了。

所有并发的问题都可以用一致的、简单的既定的模式来规避。所以可能的话，将变量限定在 goroutine 内 部；如果是多个 goroutine 都需要访问的变量，使用互斥条件来访问。

sync.Once 初始化

9.5. sync.Once 初始化

TODO

竞争条件检测

9.6. 竞争条件检测

即使我们小心到不能再小心，但在并发程序中犯错还是太容易了。幸运的是，Go 的 runtime 和工具链爲我 们装备了一个复杂但好用的动态分析工具，竞争检查器(the race detector)。

只要在 go build，go run 或者 go test 命令后面加上-race 的 flag，就会使编译器创建一个你的应用的“脩 改”版或者一个附带了能够记録所有运行期对共享变量访问工具的 test，并且会记録下每一个读或者写共 享变量的 goroutine 的身份信息。另外，脩改版的程序会记録下所有的同步事件，比如 go 语句，channel 操作，以及对(*sync.Mutex).Lock，(*sync.WaitGroup).Wait 等等的调用。(完整的同步事件集合是在 The Go Memory Model 文档中有帮助，该文档是和语言文档放在一起的。译 注：https://golang.org/ref/mem)

竞争检查器会检查这些事件，会寻找在哪一个 goroutine 中出现了这样的 case，例如其读或者写了一个共 享变量，这个共享变量是被另一个 goroutine 在没有进行榦预同步操作便直接写入的。这种情况也就表明 了是对一个共享变量的并发访问，即数据竞争。这个工具会打印一份报告，内容包含变量身份，读取和写 入的 goroutine 中活跃的函数的调用栈。这些信息在定位问题时通常很有用。9.7 节中会有一个竞争检查器 的实战样例。

竞争检查器会报告所有的已经发生的数据竞争。然而，它只能检测到运行时的竞争条件；并不能证明之后不会发生数据竞争。所以爲了使结果尽量正确，请保证你的测试并发地覆盖到了你到包。

由于需要额外的记録，因此构建时加了竞争检测的程序跑起来会慢一些，且需要更大的内存，即时是这样，这些代价对于很多生産环境的工作来说还是可以接受的。对于一些偶发的竞争条件来说，让竞争检查器来榦活可以节省无数日夜的 debugging。(译注：多少服务端 C 和 C 艹程序员爲此尽摺腰)

示例: 并发的非阻塞缓存

9.7. 示例: 并发的非阻塞缓存

TODO

Goroutines 和线程

9.8. Goroutines 和线程

在上一章中我们说 goroutine 和操作系统的线程区别可以先忽略。尽管两者的区别实际上只是一个量的区 别，但量变会引起质变的道理同样适用于 goroutine 和线程。现在正是我们来区分开两者的最佳时机。

{% include "./ch9-08-1.md" %}

{% include "./ch9-08-2.md" %}

{% include "./ch9-08-3.md" %}

{% include "./ch9-08-4.md" %}

包和工具

第十章 包和工具

现在随便一个小程序的实现都可能包含超过 10000 个函数。然而作者一般只需要考虑其中很小的一部分和 做很少的设计，因爲絶大部分代码都是由他人编写的，它们通过类似包或模块的方式被重用。

Go 语言有超过 100 个的标准包（译注：可以用`go list std | wc -l`命令查看标准包的具体数目），标准库爲大多数的程序提供了必要的基础构件。在 Go 的社区，有很多成熟的包被设计、共享、重用和改进，目前 互联网上已经发布了非常多的 Go 语音开源包，它们可以通过 http://godoc.org 检索。在本章，我们将演 示如果使用已有的包和创建新的包。

Go 还自带了工具箱，里面有很多用来简化工作区和包管理的小工具。在本书开始的时候，我们已经见识过 如何使用工具箱自带的工具来下载、构件和运行我们的演示程序了。在本章，我们将看看这些工具的基本 设计理论和尝试更多的功能，例如打印工作区中包的文档和查询相关的元数据等。在下一章，我们将探讨 探索包的单元测试用法。

包简介

10.1. 包简介

任何包系统设计的目的都是爲了简化大型程序的设计和维护工作，通过将一组相关的特性放进一个独立的单元以便于理解和更新，在每个单元更新的同时保持和程序中其它单元的相对独立性。这种模块化的特性允许每个包可以被其它的不同项目共享和重用，在项目范围内、甚至全球范围统一的分发和复用。

每个包一般都定义了一个不同的名字空间用于它内部的每个标识符的访问。每个名字空间关联到一个特定的包，让我们给类型、函数等选择简短明了的名字，这样可以避免在我们使用它们的时候减少和其它部分名字的冲突。

每个包还通过控制包内名字的可见性和是否导出来实现封装特性。通过限制包成员的可见性并隐藏包 API 的 具体实现，将允许包的维护者在不影响外部包用户的前提下调整包的内部实现。通过限制包内变量的可见 性，还可以强制用户通过某些特定函数来访问和更新内部变量，这样可以保证内部变量的一致性和并发时 的互斥约束。

当我们修改了一个源文件，我们必须重新编译该源文件对应的包和所有依赖该包的其他包。即使是从头构建，Go 语言编译器的编译速度也明显快于其它编译语言。Go 语言的闪电般的编译速度主要得益于三个语 言特性。第一点，所有导入的包必须在每个文件的开头显式声明，这样的话编译器就没有必要读取和分析 整个源文件来判断包的依赖关系。第二点，禁止包的环状依赖，因爲没有循环依赖，包的依赖关系形成一 个有向无环图，每个包可以被独立编译，而且很可能是被并发编译。第三点，编译后包的目标文件不仅仅

记录包本身的导出信息，目标文件同时还记录了包的依赖关系。因此，在编译一个包的时候，编译器只需要读取每个直接导入包的目标文件，而不需要遍历所有依赖的的文件（译注：很多都是重复的间接依 赖）。

导入路径

10.2. 导入路径

每个包是由一个全局唯一的字符串所标识的导入路径定位。出现在 import 语句中的导入路径也是字符串。

```
import (
    "fmt"
    "math/rand"
    "encoding/json"

    "golang.org/x/net/html"

    "github.com/go-sql-driver/mysql"
)
```

就像我们在 2.6.1 节提到过的，Go 语言的规范并没有指明包的导入路径字符串的具体含义，导入路径的具体
含义是由构建工具来解释的。在本章，我们将深入讨论 Go 语言工具箱的功能，包括大家经常使用的构建测 试等功能。当然，也有第三方扩展的工具箱存在。例如，Google 公司内部的 Go 语言码农，他们就使用内 部的多语言构建系统（译注：Google 公司使用的是类似 Bazel 的构建系统，支持多种编程语言，目前该构 件系统还不能完整支持 Windows 环境），用不同的规则来处理包名字和定位包，用不同的规则来处理单元 测试等等，因爲这样可以更紧密适配他们内部环境。

如果你计划分享或发布包，那麽导入路径最好是全球唯一的。爲了避免冲突，所有非标准库包的导入路径建议以所在组织的互联网域名爲前缀；而且这样也有利于包的检索。例如，上面的 import 语句导入了 Go 团 队维护的 HTML 解析器和一个流行的第三方维护的 MySQL 驱动。

包声明

10.3. 包声明

在每个 Go 语音源文件的开头都必须有包声明语句。包声明语句的主要目的是确定当前包被其它包导入时默 认的标识符（也称爲包名）。

例如，math/rand 包的每个源文件的开头都包含 `package rand` 包声明语句，所以当你导入这个包，你就

可以用 rand.Int、rand.Float64 类似的方式访问包的成员。

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println(rand.Int())
}
```

通常来说，默认的包名就是包导入路径名的最后一段，因此即使两个包的导入路径不同，它们依然可能有一个相同的包名。例如，math/rand 包和 crypto/rand 包的包名都是 rand。稍后我们将看到如何同时导入 两个有相同包名的包。

关于默认包名一般采用导入路径名的最后一段的约定也有三种例外情况。第一个例外，包对应一个可执行程序，也就是 main 包，这时候 main 包本身的导入路径是无关紧要的。名字爲 main 的包是给 go build（§10.7.3）构建命令一个信息，这个包编译完之后必须调用连接器生成一个可执行程序。

第二个例外，包所在的目録中可能有一些文件名是以_test.go 爲后缀的 Go 源文件（译注：前面必须有其它
的字符，因爲以 _ 前缀的源文件是被忽略的），并且这些源文件声明的包名也是以_test 爲后缀名的。这种目録可以包含两种包：一种普通包，加一种则是测试的外部扩展包。所有以_test 爲后缀包名的测试外部扩 展包都由 go test 命令独立编译，普通包和测试的外部扩展包是相互独立的。测试的外部扩展包一般用来避 免测试代码中的循环导入依赖，具体细节我们将在 11.2.4 节中介绍。

第三个例外，一些依赖版本号的管理工具会在导入路径后追加版本号信息，例如"gopkg.in/yaml.v2"。这种情况下包的名字并不包含版本号后缀，而是 yaml。

导入声明

10.4. 导入声明

可以在一个 Go 语言源文件包声明语句之后，其它非导入声明语句之前，包含零到多个导入包声明语句。每 个导入声明可以单独指定一个导入路径，也可以通过圆括号同时导入多个导入路径。下面两个导入形式是 等价的，但是第二种形式更爲常见。

```
import "fmt"
import "os"

import (
    "fmt"
    "os"
)
```

导入的包之间可以通过添加空行来分组；通常将来自不同组织的包独自分组。包的导入顺序无关紧要，但是在每个分组中一般会根据字符串顺序排列。（gofmt 和 goimports 工具都可以将不同分组导入的包独立 排序。）

```
import (
    "fmt"
    "html/template"
    "os"

    "golang.org/x/net/html"
    "golang.org/x/net/ipv4"
)
```

如果我们想同时导入两个有着名字相同的包，例如 math/rand 包和 crypto/rand 包，那麼导入声明必须至
少爲一个同名包指定一个新的包名以避免冲突。这叫做导入包的重命名。

```
import (
    "crypto/rand"
    mrand "math/rand" // alternative name mrand avoids conflict
)
```

导入包的重命名只影响当前的源文件。其它的源文件如果导入了相同的包，可以用导入包原本默认的名字或重命名爲另一个完全不同的名字。

导入包重命名是一个有用的特性，它不仅仅只是爲了解决名字冲突。如果导入的一个包名很笨重，特别是在一些自动生成的代码中，这时候用一个简短名称会更方便。选择用简短名称重命名导入包时候最好统 一，以避免包名混乱。选择另一个包名称还可以帮助避免和本地普通变量名産生冲突。例如，如果文件中 已经有了一个名爲 path 的变量，那麼我们可以将"path"标准包重命名爲 pathpkg。

每个导入声明语句都明确指定了当前包和被导入包之间的依赖关系。如果遇到包循环导入的情况，Go 语言 的构建工具将报告错误。

包的匿名导入

10.5. 包的匿名导入

如果只是导入一个包而并不使用导入的包将会导致一个编译错误。但是有时候我们只是想利用导入包而产生的副作用：它会计算包级变量的初始化表达式和执行导入包的 init 初始化函数（§2.6.2）。这时候我们需
要抑制“unused import”编译错误，我们可以用下划线 _ 来重命名导入的包。像往常一样，下划线 _ 爲空白标识符，并不能被访问。

```
import _ "image/png" // register PNG decoder
```

这个被称爲包的匿名导入。它通常是用来实现一个编译时机制，然后通过在 main 主程序入口选择性地导入
附加的包。首先，让我们看看如何使用该特性，然后再看看它是如何工作的。

标准库的 image 图像包包含了一 函数，用于从 接口读取数据并译码图像，它调用底个 Decode io.Reader
层注册的图像译码器来完成任务，然后返回 image.Image 类型的图像。使用image.Decode 很容易编写一个图像格式的转换工具，读取一种格式的图像，然后编码爲另一种图像格式：

```
gopl.io/ch10/jpeg
// The jpeg command reads a PNG image from the standard input
// and writes it as a JPEG image to the standard output.
package main

import (
    "fmt"
    "image"
    "image/jpeg"
    _ "image/png" // register PNG decoder
    "io"
    "os"
)

func main() {
    if err := toJPEG(os.Stdin, os.Stdout); err != nil {
        fmt.Fprintf(os.Stderr, "jpeg: %v\n", err)
        os.Exit(1)
    }
}

func toJPEG(in io.Reader, out io.Writer) error {
    img, kind, err := image.Decode(in)
    if err != nil {
        return err
    }
    fmt.Fprintln(os.Stderr, "Input format =", kind)
    return jpeg.Encode(out, img, &jpeg.Options{Quality: 95})
}
```

如果我们将 gopl.io/ch3/mandelbrot （§3.3）的输出导入到这个程序的标准输入，它将译码输入的 PNG 格式图像，然后转换爲 JPEG 格式的图像输出（图 3.3）。

```
$ go build gopl.io/ch3/mandelbrot
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
Input format = png
```

要注意 image/png 包的匿名导入语句。如果没有这一行语句，程序依然可以编译和运行，但是它将不能正
确识别和解码 PNG 格式的图像：

```
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
jpeg: image: unknown format
```

下面的代码演示了它的工作机制。标准库还提供了 GIF、PNG 和 JPEG 等格式图像的译码器，用户也可以提
供自己的译码器，但是爲了保持程序体积较小，很多译码器并没有被全部包含，除非是明确需要支持的格式。image.Decode 函数在译码时会依次查询支持的格式列表。每个格式驱动列表的每个入口指定了四件事情：格式的名称；一个用于描述这种图像数据开头部分模式的字符串，用于译码器检测识别；一个 Decode 函数用于完成译码图像工作；一个 DecodeConfig 函数用于译码图像的大小和颜色空间的信息。每 个驱动入口是通过调用 image.RegisterFormat 函数注册，一般是在每个格式包的 init 初始化函数中调用， 例如 image/png 包是这样注册的：

```
package png // image/png

func Decode(r io.Reader) (image.Image, error)
func DecodeConfig(r io.Reader) (image.Config, error)

func init() {
    const pngHeader = "\x89PNG\r\n\x1a\n"
    image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}
```

最终的效果是，主程序只需要匿名导入特定图像驱动包就可以用 image.Decode 译码对应格式的图像了。

数据库包 database/sql 也是采用了类似的技术，让用户可以根据自己需要选择导入必要的数据库驱动。例 如：

```
import (
    "database/mysql"
    _ "github.com/lib/pq"              // enable support for Postgres
    _ "github.com/go-sql-driver/mysql" // enable support for MySQL
)

db, err = sql.Open("postgres", dbname) // OK
db, err = sql.Open("mysql", dbname)    // OK
db, err = sql.Open("sqlite3", dbname)  // returns error: unknown driver "sqlite3"
```

练习 10.1： 扩展 jpeg 程序，以支持任意图像格式之间的相互转换，使用 image.Decode 检测支持的格式类 型，然后通过 flag 命令行标志参数选择输出的格式。

练习 10.2： 设计一个通用的压缩文件读取框架，用来读取 ZIP（archive/zip）和 POSIX tar（archive/tar）格式压缩的文档。使用类似上面的注册技术来扩展支持不同的压缩格式，然后根据需要 通过匿名导入选择导入要支持的压缩格式的驱动包。

包和命名

10.6. 包和命名

在本节中，我们将提供一些关于 Go 语言独特的包和成员命名的约定。

当创建一个包，一般要用短小的包名，但也不能太短导致难以理解。标准库中最常用的包有 bufio、 bytes、flag、fmt、http、io、json、os、sort、sync 和 time 等包。

它们的名字都简洁明了。例如，不要将一个类似 imageutil 或 ioutilis 的通用包命名爲 util，虽然它看起来很 短小。要尽量避免包名使用可能被经常用于局部变量的名字，这样可能导致用户重命名导入包，例如前面 看到的 path 包。

包名一般采用单数的形式。标准库的 bytes、errors 和 strings 使用了复数形式，这是爲了避免和预定义的类 型冲突，同样还有 go/types 是爲了避免和 type 关键字冲突。

要避免包名有其它的含义。例如，2.5 节中我们的温度转换包最初使用了 temp 包名，虽然并没有持续多久。但这是一个糟糕的尝试，因爲 temp 几乎是临时变量的同义词。然后我们有一段时间使用了 temperature 作爲包名，虽然名字并没有表达包的眞实用途。最后我们改成了和 strconv 标准包类似的 tempconv 包名，这个名字比之前的就好多了。

现在让我们看看如何命名包的成员。由于是通过包的导入名字引入包里面的成员，例如 fmt.Println，同时 包含了包名和成员名信息。因此，我们一般并不需要关注 Println 的具体内容，因爲 fmt 包名已经包含了这 个信息。当设计一个包的时候，需要考虑包名和成员名两个部分如何很好地配合。下面有一些例子：

```
bytes.Equal    flag.Int    http.Get    json.Marshal
```

我们可以看到一些常用的命名模式。strings 包提供了和字符串相关的诸多操作：

```
package strings

func Index(needle, haystack string) int

type Replacer struct{ /* ... */ }
func NewReplacer(oldnew ...string) *Replacer

type Reader struct{ /* ... */ }
func NewReader(s string) *Reader
```

字符串 string 本身并没有出现在每个成员名字中。因爲用户会这样引用这些成员 strings.Index、strings.Replacer 等。

其它一些包，可能只描述了单一的数据类型，例如 html/template 和 math/rand 等，只暴露一个主要的数 据结构和与它相关的方法，还有一个以 New 命名的函数用于创建实例。

```
package rand // "math/rand"

type Rand struct{ /* ... */ }
func New(source Source) *Rand
```

这可能导致一些名字重复，例如 template.Template 或 rand.Rand，这就是爲什麽这些种类的包名往往特别短的原因之一。

在另一个极端，还有像 net/http 包那样含有非常多的名字和种类不多的数据类型，因爲它们都是要执行一 个复杂的复合任务。尽管有将近二十种类型和更多的函数，但是包中最重要的成员名字却是简单明了的： Get、Post、Handle、Error、Client、Server 等。

工具

10.7. 工具

本章剩下的部分将讨论 Go 语言工具箱的具体功能，包括如何下载、格式化、构建、测试和安装 Go 语言编 写的程序。

Go 语言的工具箱集合了一系列的功能的命令集。它可以看作是一个包管理器（类似于 Linux 中的 apt 和 rpm 工具），用于包的查询、计算的包依赖关系、从远程版本控制系统和下载它们等任务。它也是一个构建系 统，计算文件的依赖关系，然后调用编译器、滙编器和连接器构建程序，虽然它故意被设计成没有标准的 make 命令那麽复杂。它也是一个单元测试和基准测试的驱动程序，我们将在第 11 章讨论测试话题。

Go 语言工具箱的命令有着类似“瑞士军刀”的风格，带着一打子的子命令，有一些我们经常用到，例如 get、run、build 和 fmt 等。你可以运行 go 或 go help 命令查看内置的帮助文档，爲了查询方便，我们列出

了最常用的命令：

```
$ go
...
    build          compile packages and dependencies
    clean          remove object files
    doc            show documentation for package or symbol
    env            print Go environment information
    fmt            run gofmt on package sources
    get            download and install packages and dependencies
    install        compile and install packages and dependencies
    list           list packages
    run            compile and run Go program
    test           test packages
    version          print Go version
    vet            run go tool vet on packages

Use "go help [command]" for more information about a command.
...
```

爲了达到零配置的设计目标，Go 语言的工具箱很多地方都依赖各种约定。例如，根据给定的源文件的名称，Go 语言的工具可以找到源文件对应的包，因爲每个目録只包含了单一的包，并且到的导入路径和工作 区的目録结构是对应的。给定一个包的导入路径，Go 语言的工具可以找到对应的目録中没个实体对应的源 文件。它还可以根据导入路径找到存储代码仓库的远程服务器的 URL。

{% include "./ch10-07-1.md" %}

{% include "./ch10-07-2.md" %}

{% include "./ch10-07-3.md" %}

{% include "./ch10-07-4.md" %}

{% include "./ch10-07-5.md" %}

{% include "./ch10-07-6.md" %}

测试

第十一章 测试

Maurice Wilkes, 第一个存储程序计算机 EDSAC 的设计者, 1949 年在他的实验室爬楼梯时有一个顿悟. 在《计算机先驱回忆録》(Memoirs of a Computer Pioneer)里, 他回忆到: "忽然间有一种醍醐灌顶的感觉, 我整个后半生的美好时光都将在寻找程序 BUG 中度过了.". 肯定从那之后的每一个存储程序的码农都可以同 情 Wilkes 的想法, 虽然也许不是没有人睏惑于他对软件开发的难度的天眞看法.

现在的程序已经远比 Wilkes 时代的更大也更复杂, 也有许多技术可以让软件的复杂性可得到控制. 其中有两种技术在实践中证明是比较有效的. 第一种是代码在被正式部署前需要进行代码评审. 第二种是测试, 是本章的讨论主题.

我们说测试的时候一般是指自动化测试, 也就是写一些小的程序用来检测被测试代码(産品代码)的行爲和预期的一样, 这些通常都是精心挑选的执行某些特定的功能或者是通过随机性的输入要验证边界的处理.

软件测试是一个巨大的领域. 测试的任务一般占据了一些程序员的部分时间和另一些程序员的全部时间. 和软件测试技术相关的图书或博客文章有成韆上万之多. 每一种主流的编程语言, 都有一打的用于测试的软件包, 也有大量的测试相关的理论, 每种都吸引了大量技术先驱和追随者. 这些都足以说服那些想要编写有效测试的程序员重新学习一套全新的技能.

Go 语言的测试技术是相对低级的. 它依赖一个 'go test' 测试命令, 和一组按照约定方式编写的测试函数, 测 试命令可以运行测试函数. 编写相对轻量级的纯测试代码是有效的, 而且它很容易延伸到基准测试和示例文 档.

在实践中, 编写测试代码和编写程序本身并没有多大区别. 我们编写的每一个函数也是针对每个具体的任务. 我们必须小心处理边界条件, 思考合适的数据结构, 推断合适的输入应该産生什麽样的结果输出. 编程测试代码和编写普通的 Go 代码过程是类似的; 它并不需要学习新的符号, 规则和工具.

go test

11.1. go test

`go test` 是一个按照一定的约定和组织的测试代码的驱动程序. 在包目録内, 以 `_test.go` 爲后缀名的源文件并不是 `go build` 构建包的以部分, 它们是 `go test` 测试的一部分.

早 `*_test.go` 文件中, 有三种类型的函数: 测试函数, 基准测试函数, 例子函数. 一个测试函数是以 Test 爲函数名前缀的函数, 用于测试程序的一些逻辑行爲是否正确; `go test` 会调用这些测试函数并报告测试结果是 PASS 或 FAIL. 基准测试函数是以 Benchmark 爲函数名前缀的函数, 用于衡量一些函数的性能; `go test`

会多次运行基准函数以计算一个平均的执行时间. 例子函数是以 Example 爲函数名前缀的函数, 提供一个由 机器检测正确性的例子文档. 我们将在 11.2 节 讨论测试函数的细节, 在 11.4 节讨论基准测试函数的细节, 在 11.6 讨论例子函数的细节.

go test 命令会遍历所有的 *_test.go 文件中上述函数, 然后生成一个临时的 main 包调用相应的测试函数, 然后构建并运行, 报告测试结果, 最后清理临时文件.

测试函数

11.2. 测试函数

每个测试函数必须导入 testing 包. 测试函数有如下的签名:

```
func TestName(t *testing.T) {
    // ...
}
```

测试函数的名字必须以 Test 开头, 可选的后缀名必须以大写字母开头:

```
func TestSin(t *testing.T) { /* ... */ }
func TestCos(t *testing.T) { /* ... */ }
func TestLog(t *testing.T) { /* ... */ }
```

其中 t 参数用于报告测试失败和附件的日志信息. 让我们顶一个一个实例包 gopl.io/ch11/word1, 只有一个函数 IsPalindrome 用于检查一个字符串是否从前向后和从后向前读都一样. (这个实现对于一个字符串是否是回文字符串前后重复测试了两次; 我们稍后会再讨论这个问题.)

```
gopl.io/ch11/word1
// Package word provides utilities for word games.
package word

// IsPalindrome reports whether s reads the same forward and backward.
// (Our first attempt.)
func IsPalindrome(s string) bool {
    for i := range s {
        if s[i] != s[len(s)-1-i] {
            return false
        }
    }
    return true
}
```

在相同的目録下, word_test.go 文件包含了 TestPalindrome 和 TestNonPalindrome 两个测试函数. 每

一个都是测试 IsPalindrome 是否给出正确的结果, 并使用 t.Error 报告失败:

```
package word

import "testing"

func TestPalindrome(t *testing.T) {
    if !IsPalindrome("detartrated") {
        t.Error(`IsPalindrome("detartrated") = false`)
    }
    if !IsPalindrome("kayak") {
        t.Error(`IsPalindrome("kayak") = false`)
    }
}

func TestNonPalindrome(t *testing.T) {
    if IsPalindrome("palindrome") {
        t.Error(`IsPalindrome("palindrome") = true`)
    }
}
```

`go test` (或 `go build`) 命令 如果没有参数指定包那麽将默认采用当前目録对应的包. 我们可以用下面的命令构建和运行测试.

```
$ cd $GOPATH/src/gopl.io/ch11/word1
$ go test
ok   gopl.io/ch11/word1  0.008s
```

还比较满意, 我们运行了这个程序, 不过没有提前退出是因爲还没有遇到 BUG 报告. 一个法国名爲 Noelle Eve Elleon 的用户抱怨 IsPalindrome 函数不能识别 ‘‘été.’’. 另外一个来自美国中部用户的抱怨是不能识别 ‘‘A man, a plan, a canal: Panama.’’. 执行特殊和小的 BUG 报告爲我们提供了新的更自然的测试用例.

```
func TestFrenchPalindrome(t *testing.T) {
    if !IsPalindrome("été") {
        t.Error(`IsPalindrome("été") = false`)
    }
}

func TestCanalPalindrome(t *testing.T) {
    input := "A man, a plan, a canal: Panama"
    if !IsPalindrome(input) {
        t.Errorf(`IsPalindrome(%q) = false`, input)
    }
}
```

爲了避免两次输入较长的字符串, 我们使用了提供了有类似 Printf 格式化功能的 Errorf 函数来滙报错误结果.

当添加了这两个测试用例之后, `go test` 返回了测试失败的信息.

```
$ go test
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
FAIL    gopl.io/ch11/word1  0.014s
```

先编写测试用例并观察到测试用例触发了和用户报告的错误相同的描述是一个好的测试习惯. 只有这样, 我们才能定位我们要眞正解决的问题.

先写测试用例的另好处是, 运行测试通常会比手工描述报告的处理更快, 这让我们可以进行快速地迭代. 如果测试集有很多运行缓慢的测试, 我们可以通过只选择运行某些特定的测试来加快测试速度.

参数 `-v` 用于打印每个测试函数的名字和运行时间:

```
$ go test -v
=== RUN TestPalindrome
--- PASS: TestPalindrome (0.00s)
=== RUN TestNonPalindrome
--- PASS: TestNonPalindrome (0.00s)
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1  0.017s
```

参数 `-run` 是一个正则表达式, 只有测试函数名被它正确匹配的测试函数才会被 `go test` 运行:

```
$ go test -v -run="French|Canal"
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1  0.014s
```

当然, 一旦我们已经脩复了失败的测试用例, 在我们提交代码更新之前, 我们应该以不带参数的 `go test` 命令运行全部的测试用例, 以确保更新没有引入新的问题.

我们现在的任务就是脩复这些错误. 简要分析后发现第一个 BUG 的原因是我们采用了 byte 而不是 rune 序 列, 所以像 "été" 中的 é 等非 ASCII 字符不能正确处理. 第二个 BUG 是因爲没有忽略空格和字母的大小写导 致的.

针对上述两个 BUG, 我们仔细重写了函数:

```
gopl.io/ch11/word2
// Package word provides utilities for word games.
package word

import "unicode"

// IsPalindrome reports whether s reads the same forward and backward.
// Letter case is ignored, as are non-letters.
func IsPalindrome(s string) bool {
    var letters []rune
    for _, r := range s {
        if unicode.IsLetter(r) {
            letters = append(letters, unicode.ToLower(r))
        }
    }
    for i := range letters {
        if letters[i] != letters[len(letters)-1-i] {
            return false
        }
    }
    return true
}
```

同时我们也将之前的所有测试数据合并到了一个测试中的表格中.

```
func TestIsPalindrome(t *testing.T) {
    var tests = []struct {
        input string
        want  bool
    }{
        {"", true},
        {"a", true},
        {"aa", true},
        {"ab", false},
        {"kayak", true},
        {"detartrated", true},
        {"A man, a plan, a canal: Panama", true},
        {"Evil I did dwell; lewd did I live.", true},
        {"Able was I ere I saw Elba", true},
        {"été", true},
        {"Et se resservir, ivresse reste.", true},
        {"palindrome", false}, // non-palindrome
        {"desserts", false},   // semi-palindrome
    }
    for _, test := range tests {
        if got := IsPalindrome(test.input); got != test.want {
            t.Errorf("IsPalindrome(%q) = %v", test.input, got)
        }
    }
}
```

我们的新测试阿都通过了:

```
$ go test gopl.io/ch11/word2
ok      gopl.io/ch11/word2      0.015s
```

这种表格驱动的测试在 Go 中很常见的. 我们很容易想表格添加新的测试数据, 并且后面的测试逻辑也没有冗
余, 这样我们可以更好地完善错误信息.

失败的测试的输出并不包括调用 t.Errorf 时刻的堆栈调用信息. 不像其他语言或测试框架的 assert 断言, t.Errorf 调用也没有引起 panic 或停止测试的执行. 即使表格中前面的数据导致了测试的失败, 表格后面的测试数据依然会运行测试, 因此在一个测试中我们可能了解多个失败的信息.

如果我们眞的需要停止测试, 或许是因爲初始化失败或可能是早先的错误导致了后续错误等原因, 我们可以使用 t.Fatal 或 t.Fatalf 停止测试. 它们必须在和测试函数同一个 goroutine 内调用.

测试失败的信息一般的形式是 "f(x) = y, want z", f(x) 解释了失败的操作和对应的输出, y 是实际的运行结果, z 是期望的正确的结果. 就像前面检查回文字符串的例子, 实际的函数用于 f(x) 部分. 如果显示 x 是表格驱动型测试中比较重要的部分, 因爲同一个断言可能对应不同的表格项执行多次. 要避免无用和冗余的信息. 在测试类似 IsPalindrome 返回布尔类型的函数时, 可以忽略并没有额外信息的 z 部分. 如果 x, y 或 z 是 y 的长度, 输出一个相关部分的简明总结即可. 测试的作者应该要努力帮助程序员诊断失败的测试.

练习 11.1: 爲 4.3 节 中的 charcount 程序编写测试.

练习 11.2: 爲 (§6.5)的 IntSet 编写一组测试, 用于检查每个操作后的行爲和基于内置 map 的集合等价 , 后面 练习 11.7 将会用到.

{% include "./ch11-02-1.md" %}

{% include "./ch11-02-2.md" %}

{% include "./ch11-02-3.md" %}

{% include "./ch11-02-4.md" %}

{% include "./ch11-02-5.md" %}

{% include "./ch11-02-6.md" %}

测试覆盖率

11.3. 测试覆盖率

就其性质而言, 测试不可能是完整的. 计算机科学家 Edsger Dijkstra 曾说过: "测试可以显示存在缺陷, 但是并不是说没有 BUG." 再多的测试也不能证明一个包没有 BUG. 在最好的情况下, 测试可以增强我们的信息, 包在我们测试的环境是可以正常工作的.

由测试驱动触发运行到的被测试函数的代码数目称爲测试的覆盖率. 测试覆盖率并不能量化 — 甚至连最简单的动态程序也难以精确测量 — 但是可以启发并帮助我们编写的有效的测试代码.

这些帮助信息中语句的覆盖率是最简单和最广泛使用的. 语句的覆盖率是指在测试中至少被运行一次的代码占总代码数的比例. 在本节中, 我们使用 `go test` 中集成的测试覆盖率工具, 来度量下面代码的测试覆盖率, 帮助我们识别测试和我们期望间的差距.

The code below is a table-driven test for the expression evaluator we built back in Chapter 7:

下面的代码是一个表格驱动的测试, 用于测试第七章的表达式求值程序:

```
gopl.io/ch7/eval

func TestCoverage(t *testing.T) {
    var tests = []struct {
        input string
        env   Env
        want  string // expected error from Parse/Check or result from Eval
    }{
        {"x % 2", nil, "unexpected '%'"},
        {"!true", nil, "unexpected '!'"},
        {"log(10)", nil, `unknown function "log"`},
        {"sqrt(1, 2)", nil, "call to sqrt has 2 args, want 1"},
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
    }

    for _, test := range tests {
        expr, err := Parse(test.input)
        if err == nil {
            err = expr.Check(map[Var]bool{})
        }
        if err != nil {
            if err.Error() != test.want {
                t.Errorf("%s: got %q, want %q", test.input, err, test.want)
            }
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        if got != test.want {
            t.Errorf("%s: %v => %s, want %s",
                test.input, test.env, got, test.want)
        }
    }
}
```

首先, 我们要确保所有的测试都正常通过:

```
$ go test -v -run=Coverage gopl.io/ch7/eval
=== RUN TestCoverage
--- PASS: TestCoverage (0.00s)
PASS
ok      gopl.io/ch7/eval        0.011s
```

下面这个命令可以显示测试覆盖率工具的用法信息:

```
$ go tool cover
Usage of 'go tool cover':
Given a coverage profile produced by 'go test':
    go test -coverprofile=c.out

Open a web browser displaying annotated source code:
    go tool cover -html=c.out
...
```

`go tool` 命令运行 Go 工具链的底层可执行程序. 这些底层可执行程序放在 $GOROOT/pkg/tool/${GOOS}_${GOARCH} 目録. 因爲 `go build` 的原因, 我们很小直接调用这些底层工具.

现在我们可以用 `-coverprofile` 标志参数重新运行:

```
$ go test -run=Coverage -coverprofile=c.out gopl.io/ch7/eval
ok      gopl.io/ch7/eval        0.032s      coverage: 68.5% of statements
```

这个标志参数通过插入生成钩子代码来统计覆盖率数据. 也就是说, 在运行每个测试前, 它会脩改要测试代码的副本, 在每个块都会设置一个布尔标志变量. 当被脩改后的被测试代码运行退出时, 将统计日志数据写入 c.out 文件, 并打印一部分执行的语句的一个总结. (如果你需要的是摘要,使用 `go test -cover` .)

如果使用了 `-covermode=count` 标志参数, 那麼将在每个代码块插入一个计数器而不是布尔标志量. 在统计结果中记録了每个块的执行次数, 这可以用于衡量哪些是被频繁执行的热点代码.

爲了收集数据, 我们运行了测试覆盖率工具, 打印了测试日志, 生成一个 HTML 报告, 然后在浏览器中打开(图 11.3).

```
$ go tool cover -html=c.out
```

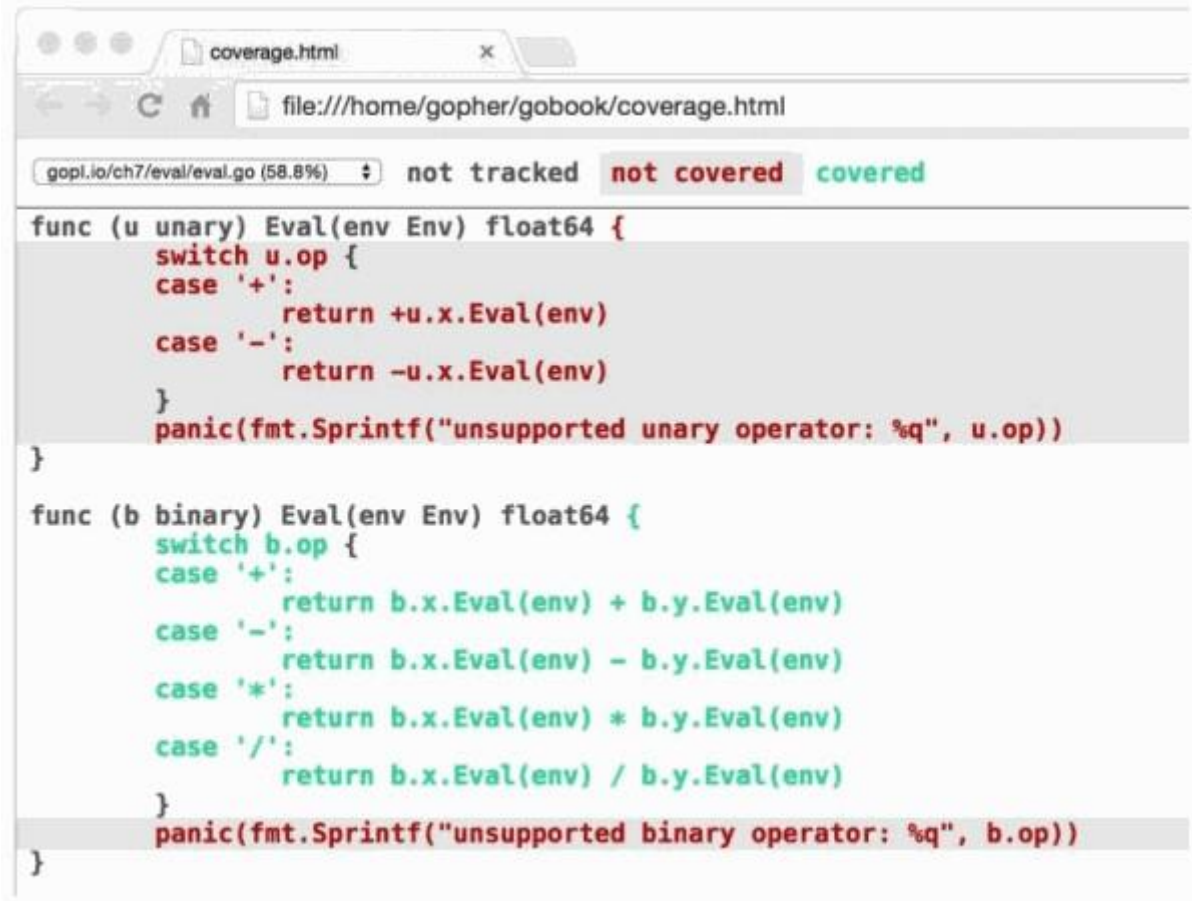


Figure 11.3. A coverage report.

绿色的代码块被测试覆盖到了, 红色的则表示没有被覆盖到. 爲了清晰起见, 我们将的背景红色文本的背景设置成了阴影效果. 我们可以马上发现 unary 操作的 Eval 方法并没有被执行到. 如果我们针对这部分未被覆盖的代码添加下面的测试, 然后重新运行上面的命令, 那麼我们将会看到那个红色部分的代码也变成绿色了:

```
{"-x * -x", eval.Env{"x": 2}, "4"}
```

不过两个 panic 语句依然是红色的. 这是没有问题的, 因爲这两个语句并不会被执行到.

实现 100% 的测试覆盖率听起来很好, 但是在具体实践中通常是不可行的, 也不是值得推荐的做法. 因爲那只能帮助代码被执行过而已, 并不意味着代码是没有 BUG 的; 因爲对于逻辑复杂的语句需要针对不同的输入 执行多次. 有一些语句, 例如上面的 panic 语句则永远都不会被执行到. 另外, 还有一些隐晦的错误在现实中 很少遇到也很难编写对应的测试代码. 测试从本质上来说是一个比较务实的工作, 编写测试代码和编写应用 代码的成本对比是需要考虑的. 测试覆盖率工具可以帮助我们快速识别测试薄弱的地方, 但是设计好的测试 用例和编写应用代码一样需要严密的思考.

基准测试

11.4. 基准测试

基准测试是测量一个程序在固定工作负载下的性能. 在 Go 语言中, 基准测试函数和普通测试函数类似, 但是以 Benchmark 爲前缀名, 并且带有一 `*testing.B` 类型的参数; `*testing.B` 除了提供和 `*testing.T` 类个
似的方法, 还有额外一些和性能测量相关的方法. 它还提供了一个整数 N, 用于指定操作执行的循环次数

. 下面是 IsPalindrome 函数的基准测试, 其中循环将执行 N 次.

```
import "testing"

func BenchmarkIsPalindrome(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IsPalindrome("A man, a plan, a canal: Panama")
    }
}
```

我们用下面的命令运行基准测试. 和普通测试不同的是, 默认情况下不运行任何基准测试. 我们需要通过 `-bench` 命令行标志参数手工指定要运行的基准测试函数. 该参数是一个正则表达式, 用于匹配要执行的基准测试函数的名字, 默认值是空的. 其中 ‘.’ 模式将可以匹配所有基准测试函数, 但是这里总共只有一个基准测试函数, 因此 和 `-bench=IsPalindrome` 参数是等价的效果.

```
$ cd $GOPATH/src/gopl.io/ch11/word2
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000              1035 ns/op
ok      gopl.io/ch11/word2      2.179s
```

基准测试名的数字后缀部分, 这里是 8, 表示运行时对应的 GOMAXPROCS 的值, 这对于一些和并发相关的基准测试是重要的信息.

报告显示每次调用 IsPalindrome 函数花费 1.035 微秒, 是执行 1,000,000 次的平均时间. 因爲基准测试驱 动器并不知道每个基准测试函数运行所花的时候, 它会尝试在眞正运行基准测试前先尝试用较小的 N 运行 测试来估算基准测试函数所需要的时间, 然后推断一个较大的时间保证稳定的测量结果.

循环在基准测试函数内实现, 而不是放在基准测试框架内实现, 这样可以让每个基准测试函数有机会在循环启动前执行初始化代码, 这样并不会显著影响每次迭代的平均运行时间. 如果还是担心初始化代码部分对测量时间带来榦扰, 那麽可以通过 testing.B 参数的方法来临时关闭或重置计时器, 不过这些一般很少会用到.

现在我们有了一个基准测试和普通测试, 我们可以很容易测试新的让程序运行更快的想法. 也许最明显的优化是在 IsPalindrome 函数中第二个循环的停止检查, 这样可以避免每个比较都做两次:

```
n := len(letters)/2
for i := 0; i < n; i++ {
    if letters[i] != letters[len(letters)-1-i] {
        return false
    }
}
return true
```

不过很多情况下, 一个明显的优化并不一定就能代码预期的效果. 这个改进在基准测试中值带来了 4% 的性能提陞.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000             992 ns/op
ok      gopl.io/ch11/word2      2.093s
```

另一个改进想法是在开始爲每个字符预先分配一个足够大的数组, 这样就可以避免在 append 调用时可能会导致内存的多次重新分配. 声明一个 letters 数组变量, 并指定合适的大小, 像这样,

```
letters := make([]rune, 0, len(s))
for _, r := range s {
    if unicode.IsLetter(r) {
        letters = append(letters, unicode.ToLower(r))
    }
}
```

这个改进提陞性能约 35%, 报告结果是基于 2,000,000 次迭代的平均运行时间统计.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 2000000                   697 ns/op
ok      gopl.io/ch11/word2      1.468s
```

如这个例子所示, 快的程序往往是有很少的内存分配. `-benchmem` 命令行标志参数将在报告中包含内存的分配数据统计. 我们可以比较优化前后内存的分配情况:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    1000000   1026 ns/op    304 B/op  4 allocs/op
```

这是优化之后的结果:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    2000000    807 ns/op    128 B/op  1 allocs/op
```

一次内存分配代替多次的内存分配节省了 75%的分配调用次数和减少近一半的内存需求.

这个基准测试告诉我们所需的絶对时间依赖给定的具体操作, 两个不同的操作所需时间的差异也是和不同环境相关的. 例如, 如果一个函数需要 1ms 处理 1,000 个元素, 那麽处理 10000 或 1 百万 将需要多少时间 呢? 这样的比较揭示了渐近增长函数的运行时间. 另一个例子: I/O 缓存该设置爲多大呢? 基准测试可以帮助我们选择较小的缓存但能带来满意的性能. 第三个例子: 对于一个确定的工作那种算法更好? 基准测试可以评估两种不同算法对于相同的输入在不同的场景和负载下的优缺点.

比较基准测试都是结构类似的代码. 它们通常是采用一个参数的函数, 从几个标志的基准测试函数入口调用, 就像这样:

```
func benchmark(b *testing.B, size int) { /* ... */ }
func Benchmark10(b *testing.B)         { benchmark(b, 10) }
func Benchmark100(b *testing.B)        { benchmark(b, 100) }
func Benchmark1000(b *testing.B)       { benchmark(b, 1000) }
```

通过函数参数来指定输入的大小, 但是参数变量对于每个具体的基准测试都是固定的. 要避免直接修改 b.N 来控制输入的大小. 除非你将它作爲一个固定大小的迭代计算输入, 否则基准测试的结果将毫无意义.

基准测试对于编写代码是很有帮助的, 但是即使工作完成了应应当保存基准测试代码. 因爲随着项目的发展, 或者是输入的增加, 或者是部署到新的操作系统或不同的处理器, 我们可以再次用基准测试来帮助我们改进设计.

练习 11.6: 爲 2.6.2 节 的 练习 2.4 和 练习 2.5 的 PopCount 函数编写基准测试. 看看基于表格算法在不同情况下的性能.

练习 11.7: 爲 *IntSet (§6.5) 的 Add, UnionWith 和 其他方法编写基准测试, 使用大量随机出入. 你可以让这些方法跑多快? 选择字的大小对于性能的影响如何? IntSet 和基于内建 map 的实现相比有多快?

剖析

11.5. 剖析

测量基准对于衡量特定操作的性能是有帮助的, 但是, 当我们视图让程序跑的更快的时候, 我们通常并不知道从哪里开始优化. 每个码农都应该知道 Donald Knuth 在 1974 年的 ‘‘Structured Programming with go to Statements’’ 上所说的格言. 虽然经常被解读爲不重视性能的意思, 但是从原文我们可以看到不同的含义:

> 毫无疑问, 效率会导致各种滥用. 程序员需要浪费大量的时间思考, 或者担心, 被部分程序的速度所躲扰, 实际上这些尝试提陞效率的行爲可能産生强烈的负面影响, 特别是当调试和维护的时候. 我们不应该过度纠结于细节的优化, 应该说约 97%的场景: 过早的优化是万恶之源.

> 我们当然不应该放弃那关键的 3%的机会. 一个好的程序员不会因爲这个理由而满足, 他们会明智地观察和识别哪些是关键的代码; 但是只有在关键代码已经被确认的前提下才会进行优化. 对于判断哪些部分是关键代码是经常容易犯经验性错误的地方, 因此程序员普通使用的测量工具, 使得他们的直觉很不靠谱.

当我们想仔细观察我们程序的运行速度的时候, 最好的技术是如何识别关键代码. 自动化的剖析技术是基于程序执行期间一些抽样数据, 然后推断后面的执行状态; 最终産生一个运行时间的统计数据文件.

Go 语言支持多种类型的剖析性能分析, 每一种关注不同的方面, 但它们都涉及到每个采样记録的感兴趣的一系列事件消息, 每个事件都包含函数调用时函数调用堆栈的信息. 内建的 `go test` 工具对几种分析方式都提供了支持.

CPU 分析文件标识了函数执行时所需要的 CPU 时间. 当前运行的系统线程在每隔几毫秒都会遇到操作系统的
中断事件, 每次中断时都会记録一个分析文件然后恢复正常的运行.

堆分析则记録了程序的内存使用情况. 每个内存分配操作都会触发内部平均内存分配例程, 每个 512KB 的内存申请都会触发一个事件.

阻塞分析则记録了 goroutine 最大的阻塞操作, 例如系统调用, 管道发送和接收, 还有获取锁等. 分析库会记 録每个 goroutine 被阻塞时的相关操作.

在测试环境下只需要一个标志参数就可以生成各种分析文件. 当一次使用多个标志参数时需要当心, 因爲分析操作本身也可能会影像程序的运行.

```
$ go test -cpuprofile=cpu.out
$ go test -blockprofile=block.out
$ go test -memprofile=mem.out
```

对于一些非测试程序也很容易支持分析的特性, 具体的实现方式和程序是短时间运行的小工具还是长时间运行的服务会有很大不同, 因此 Go 的 runtim 运行时包提供了程序运行时控制分析特性的接口.

一旦我们已经收集到了用于分析的采样数据, 我们就可以使用 pprof 据来分析这些数据. 这是 Go 工具箱自带的一个工具, 但并不是一个日常工具, 它对应 `go tool pprof` 命令. 该命令有许多特性和选项, 但是最重要的有两个, 就是生成这个概要文件的可执行程序和对于的分析日志文件.

爲了提高分析效率和减少空间, 分析日志本身并不包含函数的名字; 它只包含函数对应的地址. 也就是说 pprof 需要和分析日志对于的可执行程序. 虽然 `go test` 命令通常会丢弃临时用的测试程序, 但是在启用分析的时候会将测试程序保存爲 foo.test 文件, 其中 foo 部分对于测试包的名字.

下面的命令演示了如何生成一个 CPU 分析文件. 我们选择 `net/http` 包的一个基准测试. 通常是基于一个已经确定了是关键代码的部分进行基准测试. 基准测试会默认包含单元测试, 这里我们用 -run=NONE 禁止单元测试.

```
$ go test -run=NONE -bench=ClientServerParallelTLS64 \
    -cpuprofile=cpu.log net/http
 PASS
 BenchmarkClientServerParallelTLS64-8  1000
    3141325 ns/op 143010 B/op 1747 allocs/op
ok       net/http       3.395s

$ go tool pprof -text -nodecount=10 ./http.test cpu.log
2570ms of 3590ms total (71.59%)
Dropped 129 nodes (cum <= 17.95ms)
Showing top 10 nodes out of 166 (cum >= 60ms)
    flat  flat%   sum%     cum   cum%
  1730ms 48.19% 48.19%  1750ms 48.75%  crypto/elliptic.p256ReduceDegree
   230ms  6.41% 54.60%   250ms  6.96%  crypto/elliptic.p256Diff
   120ms  3.34% 57.94%   120ms  3.34%  math/big.addMulVVW
   110ms  3.06% 61.00%   110ms  3.06%  syscall.Syscall
    90ms  2.51% 63.51%  1130ms 31.48%  crypto/elliptic.p256Square
    70ms  1.95% 65.46%   120ms  3.34%  runtime.scanobject
    60ms  1.67% 67.13%   830ms 23.12%  crypto/elliptic.p256Mul
    60ms  1.67% 68.80%   190ms  5.29%  math/big.nat.montgomery
    50ms  1.39% 70.19%    50ms  1.39%  crypto/elliptic.p256ReduceCarry
    50ms  1.39% 71.59%    60ms  1.67%  crypto/elliptic.p256Sum
```

参数 `-text` 标志参数用于指定输出格式, 在这里每行是一个函数, 根据使用 CPU 的时间来排序. 其中 `-nodecount=10` 标志参数限制了只输出前 10 行的结果. 对于严重的性能问题, 这个文本格式基本可以帮助查明原因了.

这个概要文件告诉我们, HTTPS 基准测试中 `crypto/elliptic.p256ReduceDegree` 函数占用了将近一般的 CPU 资源. 相比之下, 如果一个概要文件中主要是 runtime 包的内存分配的函数, 那麼减少内存消耗可能是一 个值得尝试的优化策略.

对于一些更微妙的问题, 你可能需要使用 pprof 的图形显示功能. 这个需要安装 GraphViz 工具, 可以从 www.graphviz.org 下载. 参数 `-web` 用于生成一个有向图文件, 包含 CPU 的使用和最特点的函数等信息.

这一节我们只是简单看了下 Go 语言的分析据工具. 如果想了解更多, 可以閱读 Go 官方博客的 ‘‘Profiling Go Programs’’ 一文.

示例函数

11.6. 示例函数

第三种 `go test` 特别处理的函数是示例函数, 以 Example 爲函数名开头. 示例函数没有函数参数和返回值. 下面是 IsPalindrome 函数对应的示例函数:

```
func ExampleIsPalindrome() {
    fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"))
    fmt.Println(IsPalindrome("palindrome"))
    // Output:
    // true
    // false
}
```

示例函数有三个用处. 最主要的一个是用于文档: 一个包的例子可以更简洁直观的方式来演示函数的用法, 会文字描述会更直接易懂, 特别是作爲一个提醒或快速参考时. 一个例子函数也可以方便展示属于同一个接口的几种类型或函数直接的关系, 所有的文档都必须关联到一个地方, 就像一个类型或函数声明都统一到包一样. 同时, 示例函数和注释并不一样, 示例函数是完整眞是的 Go 代码, 需要介绍编译器的编译时检查, 这样 可以保证示例代码不会腐烂成不能使用的旧代码.

根据示例函数的后缀名部分, godoc 的 web 文档会将一个示例函数关联到某个具体函数或包本身, 因此 ExampleIsPalindrome 示例函数将是 IsPalindrome 函数文档的一部分, Example 示例函数将是包文档的一部分.

示例文档的第二个用处是在 `go test` 执行测试的时候也运行示例函数测试. 如果示例函数内含有类似上面例子中的 `// Output:` 和注释, 这样的注释, 那麽测试工具会执行这个示例函数, 然后检测这个示例函数的标准输出是否匹配.

示例函数的第三个目的提供一个眞实的演练场. golang.org 是由 dogoc 提供的服务, 它使用了 Go Playground 技术让用户可以在浏览器中在线编辑和运行每个示例函数, 就像 图 11.4 所示的那样. 这通常是学习函数使用或 Go 语言特性的最快方式.

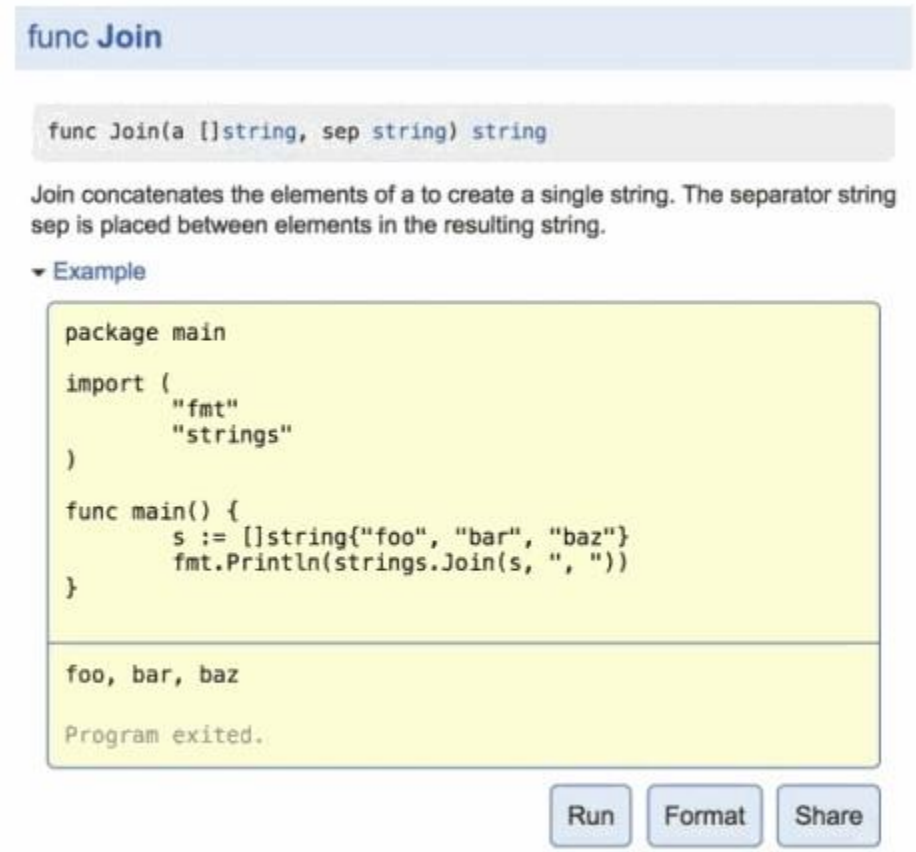


Figure 11.4. An interactive example of `strings.Join` in godoc.

本书最后的两掌是讨论 reflect 和 unsafe 包, 一般的 Go 用于很少需要使用它们. 因此, 如果你还没有写过任
何眞是的 Go 程序的话, 现在可以忽略剩余部分而直接编码了.

反射

第十二章 反射

Go 提供了一种机制在运行时更新变量和检查它们的值，调用它们的方法，和它们支持的内在操作，但是在编 译时并不知道这些变量的类型．这种机制被称爲反射．反射也可以让我们将类型本身作爲第一类的值类型处 理．

在本章，我们将探讨 Go 语言的反射特性，看看它可以给语言增加哪些表达力，以及在两个至关重要的 API 是 如何用反射机制的：一个是 fmt 包提供的字符串格式功能，另一个是类似 encoding/json 和 encoding/xml 提供的针对特定协议的编译码功能．对于我们在 4.6 节中看到过的 text/template 和 html/template 包，它们的实现也是依赖反射技术的．然后，反射是一个复杂的内省技术，而应该随意使用，因此，尽管上面这些包都是用反射技术实现的，但是它们自己的 API 都没有公开反射相关的接口．

爲何需要反射?

12.1. 爲何需要反射?

有时候我们需要编写一个函数能够处理一类并不满足普通公共接口的类型的值，也可能它们并没有确定的表示方式，或者在我们设计该函数的时候还这些类型可能还不存在，各种情况都有可能．

一个大家熟悉的例子是 fmt.Fprintf 函数提供的字符串格式化处理逻辑，它可以用例对任意类型的值格式化打印，甚至是用户自定义的类型．让我们来尝试实现一个类似功能的函数．简单起见，我们的函数只接收一个参数，然后返回和 fmt.Sprint 类似的格式化后的字符串，我们的函数名也叫 Sprint．

我们使用了 switch 分支首先来测试输入参数是否实现了 String 方法，如果是的话就使用该方法．然后继续增加测试分支，检查是否是每个基于 string，int，bool 等基础类型的动态类型，并在每种情况下执行适当的格式化操作．

```
func Sprint(x interface{}) string {
    type stringer interface {
        String() string
    }
    switch x := x.(type) {
    case stringer:
        return x.String()
    case string:
        return x
    case int:
        return strconv.Itoa(x)
    // ...similar cases for int16, uint32, and so on...
    case bool:
        if x {
            return "true"
        }
        return "false"
    default:
        // array, chan, func, map, pointer, slice, struct
        return "???"
    }
}
```

但是我们如何处理其它类似 []float64, map[string][]string 等类型呢? 我们当然可以添加更多的测试分支, 但是这些组合类型的数目基本是无穷的. 还有如何处理 url.Values 等命令的类型呢? 虽然类型分支可以识别出底层的基础类型是 map[string][]string, 但是它并不匹配 url.Values 类型, 因爲这是两种不同的类型, 而且 switch 分支也不可能包含每个类似 url.Values 的类型, 这会导致对这些库的依赖.

没有一种方法来检查未知类型的表示方式, 我们被卡住了. 这就是我们爲何需要反射的原因.

reflect.Type 和 reflect.Value

12.2. reflect.Type 和 reflect.Value

反射是由 reflect 包提供支持. 它定义了两个重要的类型, Type 和 Value. 一个 Type 表示一个 Go 类型. 它 是一个接口, 有许多方法来区分类型和检查它们的组件, 例如一个结构体的成员或一个函数的参数等. 唯一 能反映 reflect.Type 实现的是接口的类型描述信息(§7.5), 同样的实体标识了动态类型的接口值.

函数 reflect.TypeOf 接受任意的 interface{} 类型, 并返回对应动态类型的 reflect.Type:

```
t := reflect.TypeOf(3) // a reflect.Type
fmt.Println(t.String()) // "int"
fmt.Println(t)          // "int"
```

其中 TypeOf(3) 调用将值 3 作爲 interface{} 类型参数传入. 回到 7.5 节 的将一个具体的值转爲接口类型会

有一个隐式的接口转换操作, 它会创建一个包含两个信息的接口值: 操作数的动态类型(这里是 int)和它的动 态的值(这里是 3).

因爲 reflect.TypeOf 返回的是一个动态类型的接口值, 它总是返回具体的类型. 因此, 下面的代码将打印 "*os.File" 而不是 "io.Writer". 稍后, 我们将看到 reflect.Type 是具有识别接口类型的表达方式功能的.

```
var w io.Writer = os.Stdout
fmt.Println(reflect.TypeOf(w)) // "*os.File"
```

要注意的是 reflect.Type 接口是满足 fmt.Stringer 接口的. 因爲打印动态类型值对于调试和日志是有帮助的, fmt.Printf 提供了一个简短的 %T 标志参数, 内部使用 reflect.TypeOf 的结果输出:

```
fmt.Printf("%T\n", 3) // "int"
```

reflect 包中另一个重要的类型是 Value. 一个 reflect.Value 可以持有一个任意类型的值. 函数 reflect.ValueOf 接受任意的 interface{} 类型, 并返回对应动态类型的 reflect.Value. 和 reflect.TypeOf 类 似, reflect.ValueOf 返回的结果也是对于具体的类型, 但是 reflect.Value 也可以持有一个接口值.

```
v := reflect.ValueOf(3) // a reflect.Value
fmt.Println(v)          // "3"
fmt.Printf("%v\n", v)   // "3"
fmt.Println(v.String()) // NOTE: "<int Value>"
```

和 reflect.Type 类似, reflect.Value 也满足 fmt.Stringer 接口, 但是除非 Value 持有的是字符串, 否则 String 只是返回具体的类型. 相同, 使用 fmt 包的 %v 标志参数, 将使用 reflect.Values 的结果格式化.

调用 Value 的 Type 方法将返回具体类型所对应的 reflect.Type:

```
t := v.Type()           // a reflect.Type
fmt.Println(t.String()) // "int"
```

逆操作是调用 reflect.ValueOf 对应的 reflect.Value.Interface 方法. 它返回一个 interface{} 类型表示 reflect.Value 对应类型的具体值:

```
v := reflect.ValueOf(3) // a reflect.Value
x := v.Interface()      // an interface{}
i := x.(int)            // an int
fmt.Printf("%d\n", i)   // "3"
```

一个 reflect.Value 和 interface{} 都能保存任意的值. 所不同的是, 一个空的接口隐藏了值对应的表示方式和所有的公开的方法, 因此只有我们知道具体的动态类型才能使用类型断言来访问内部的值(就像上面那样), 对于内部值并没有特别可做的事情. 相比之下, 一个 Value 则有很多方法来检查其内容, 无论它的具体类型是什麽. 让我们再次尝试实现我们的格式化函数 format.Any.

我们使用 reflect.Value 的 Kind 方法来替代之前的类型 switch. 虽然还是有无穷多的类型, 但是它们的 kinds 类型却是有限的: Bool, String 和 所有数字类型的基础类型; Array 和 Struct 对应的聚合类型; Chan, Func, Ptr, Slice, 和 Map 对应的引用类似; 接口类型; 还有表示空值的无效类型. (空的 reflect.Value 对应 Invalid 无效类型.)

```
gopl.io/ch12/format
package format

import (
    "reflect"
    "strconv"
)

// Any formats any value as a string.
func Any(value interface{}) string {
    return formatAtom(reflect.ValueOf(value))
}

// formatAtom formats a value without inspecting its internal structure.
func formatAtom(v reflect.Value) string {
    switch v.Kind() {
    case reflect.Invalid:
        return "invalid"
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return strconv.FormatUint(v.Uint(), 10)
    // ...floating-point and complex cases omitted for brevity...
    case reflect.Bool:
        return strconv.FormatBool(v.Bool())
    case reflect.String:
        return strconv.Quote(v.String())
    case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
        return v.Type().String() + " 0x" +
            strconv.FormatUint(uint64(v.Pointer()), 16)
    default: // reflect.Array, reflect.Struct, reflect.Interface
        return v.Type().String() + " value"
    }
}
```

到目前未知, 我们的函数将每个值视作一个不可分割没有内部结构的, 因此它叫 formatAtom. 对于聚合类型(结构体和数组)个接口只是打印类型的值, 对于引用类型(channels, functions, pointers, slices, 和 maps), 它十六进制打印类型的引用地址. 虽然还不够理想, 但是依然是一个重大的进步, 并且 Kind 只关心底层表示, format.Any 也支持新命名的类型. 例如:

```
var x int64 = 1
var d time.Duration = 1 * time.Nanosecond
fmt.Println(format.Any(x))                  // "1"
fmt.Println(format.Any(d))                  // "1"
fmt.Println(format.Any([]int64{x}))         // "[]int64 0x8202b87b0"
fmt.Println(format.Any([]time.Duration{d})) // "[]time.Duration 0x8202b87e0"
```

Display 递归打印

12.3. Display 递归打印

接下来，让我们看看如何改善聚合数据类型的显示。我们并不想完全克隆一个 fmt.Sprint 函数，我们只是 像构建一个用于调式用的 Display 函数，给定一个聚合类型 x，打印这个值对应的完整的结构，同时记录每 个发现的每个元素的路径。让我们从一个例子开始。

```
e, _ := eval.Parse("sqrt(A / pi)")
Display("e", e)
```

在上面的调用中，传入 Display 函数的参数是在 7.9 节一个表达式求值函数返回的语法树。Display 函数的输
出如下：

```
Display e (eval.call):
e.fn = "sqrt"
e.args[0].type = eval.binary
e.args[0].value.op = 47
e.args[0].value.x.type = eval.Var
e.args[0].value.x.value = "A"
e.args[0].value.y.type = eval.Var
e.args[0].value.y.value = "pi"
```

在可能的情况下，你应该避免在一个包中暴露和反射相关的接口。我们将定义一个未导出的 display 函数用
于递归处理工作，导出的是 Display 函数，它只是 display 函数简单的包装以接受 interface{}类型的参数：

```
gopl.io/ch12/display

func Display(name string, x interface{}) {
    fmt.Printf("Display %s (%T):\n", name, x)
    display(name, reflect.ValueOf(x))
}
```

在 display 函数中，我们使用了前面定义的打印基础类型——基本类型、函数和 chan 等——元素值的

formatAtom 函数，但是我们会使用 reflect.Value 的方法来递归显示聚合类型的每一个成员或元素。在递

归下降过程中，path 字符串，从最开始传入的起始值（这里是“e”），将逐步增长以表示如何达到当前 值（例如“e.args[0].value”）。

因爲我们不再模拟 fmt.Sprint 函数，我们将直接使用 fmt 包来简化我们的例子实现。

```
func display(path string, v reflect.Value) {
    switch v.Kind() {
    case reflect.Invalid:
        fmt.Printf("%s = invalid\n", path)
    case reflect.Slice, reflect.Array:
        for i := 0; i < v.Len(); i++ {
            display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))
        }
    case reflect.Struct:
        for i := 0; i < v.NumField(); i++ {
            fieldPath := fmt.Sprintf("%s.%s", path, v.Type().Field(i).Name)
            display(fieldPath, v.Field(i))
        }
    case reflect.Map:
        for _, key := range v.MapKeys() {
            display(fmt.Sprintf("%s[%s]", path,
                formatAtom(key)), v.MapIndex(key))
        }
    case reflect.Ptr:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            display(fmt.Sprintf("(*%s)", path), v.Elem())
        }
    case reflect.Interface:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            fmt.Printf("%s.type = %s\n", path, v.Elem().Type())
            display(path+".value", v.Elem())
        }
    default: // basic types, channels, funcs
        fmt.Printf("%s = %s\n", path, formatAtom(v))
    }
}
```

让我们针对不同类型分别讨论。

Slice 和数组： 两种的处理逻辑是一样的。Len 方法返回 slice 或数组值中的元素个数，Index(i)活动索引 i 对 应的元素，返回的也是一个 reflect.Value 类型的值；如果索引 i 超出范围的话将导致 panic 异常，这些行爲 和数组或 slice 类型内建的 len(a)和 a[i]等操作类似。display 针对序列中的每个元素递归调用自身处理，我们 通过在递归处理时向 path 附加“[i]”来表示访问路径。

虽然 reflect.Value 类型带有很多方法，但是只有少数的方法对任意值都是可以安全调用的。例如，Index 方 法只能对 Slice、数组或字符串类型的值调用，其它类型如果调用将导致 panic 异常。

结构体： NumField 方法报告结构体中成员的数量，Field(i)以 reflect.Value 类型返回第 i 个成员的值。成员 列表包含了匿名成员在内的全部成员。通过在 path 添加 ".f" 来表示成员路径，我们必须获得结构体对应 的 reflect.Type 类型信息，包含结构体类型和第 i 个成员的名字。

Maps: MapKeys 方法返回一个 reflect.Value 类型的 slice，每一个都对应 map 的可以。和往常一样，遍历 map 时顺序是随机的。MapIndex(key)返回 map 中 key 对应的 value。我们向 path 添加 "[key]" 来表示访 问路径。（我们这里有一个未完成的工作。其实 map 的 key 的类型并不局限于 formatAtom 能完美处理的类 型；数组、结构体和接口都可以作爲 map 的 key。针对这种类型，完善 key 的显示信息是练习 12.1 的任务。
）

指针： Elem 方法返回指针指向的变量，还是 reflect.Value 类型。技术指针是 nil，这个操作也是安全的，在 这种情况下指针是 Invalid 无效类型，但是我们可以用 IsNil 方法来显式地测试一个空指针，这样我们可以打 印更合适的信息。我们在 path 前面添加 "*" ，并用括号包含以避免歧义。

接口： 再一次，我们使用 IsNil 方法来测试接口是否是 nil，如果不是，我们可以调用 v.Elem()来获取接口对 应的动态值，并且打印对应的类型和值。

现在我们的 Display 函数总算完工了，让我们看看它的表现吧。下面的 Movie 类型是在 4.5 节的电影类型上 演变来的：

```
type Movie struct {
    Title, Subtitle string
    Year           int
    Color          bool
    Actor          map[string]string
    Oscars         []string
    Sequel         *string
}
```

让我们声明一个该类型的变量，然后看看 Display 函数如何显示它：

```
strangelove := Movie{
    Title:    "Dr. Strangelove",
    Subtitle: "How I Learned to Stop Worrying and Love the Bomb",
    Year:     1964,
    Color:    false,
    Actor: map[string]string{
        "Dr. Strangelove":           "Peter Sellers",
        "Grp. Capt. Lionel Mandrake": "Peter Sellers",
        "Pres. Merkin Muffley":       "Peter Sellers",
        "Gen. Buck Turgidson":        "George C. Scott",
        "Brig. Gen. Jack D. Ripper":  "Sterling Hayden",
        `Maj. T.J. "King" Kong`:      "Slim Pickens",
    },

    Oscars: []string{
        "Best Actor (Nomin.)",
        "Best Adapted Screenplay (Nomin.)",
        "Best Director (Nomin.)",
        "Best Picture (Nomin.)",
    },
}
```

Display("strangelove", strangelove)调用将显示（strangelove 电影对应的中文名是《奇爱博士》）：

```
Display strangelove (display.Movie):
strangelove.Title = "Dr. Strangelove"
strangelove.Subtitle = "How I Learned to Stop Worrying and Love the Bomb"
strangelove.Year = 1964
strangelove.Color = false
strangelove.Actor["Gen. Buck Turgidson"] = "George C. Scott"
strangelove.Actor["Brig. Gen. Jack D. Ripper"] = "Sterling Hayden"
strangelove.Actor["Maj. T.J. \"King\" Kong"] = "Slim Pickens"
strangelove.Actor["Dr. Strangelove"] = "Peter Sellers"
strangelove.Actor["Grp. Capt. Lionel Mandrake"] = "Peter Sellers"
strangelove.Actor["Pres. Merkin Muffley"] = "Peter Sellers"
strangelove.Oscars[0] = "Best Actor (Nomin.)"
strangelove.Oscars[1] = "Best Adapted Screenplay (Nomin.)"
strangelove.Oscars[2] = "Best Director (Nomin.)"
strangelove.Oscars[3] = "Best Picture (Nomin.)"
strangelove.Sequel = nil
```

我们也可以使用 Display 函数来显示标准库中类型的内部结构，例如`*os.File` 类型：

```
Display("os.Stderr", os.Stderr)
// Output:
// Display os.Stderr (*os.File):
// (*(*os.Stderr).file).fd = 2
// (*(*os.Stderr).file).name = "/dev/stderr"
// (*(*os.Stderr).file).nepipe = 0
```

要注意的是，结构体中未导出的成员对反射也是可见的。需要当心的是这个例子的输出在不同操作系统上可能是不同的，并且随着标准库的发展也可能导致结果不同。（这也是将这些成员定义爲私有成员的原因之一！）我们深圳可以用 Display 函数来显示 reflect.Value，来查看`*os.File`类型的内部表示方式。`Display("rV", reflect.ValueOf(os.Stderr))`调用的输出如下，当然不同环境得到的结果可能有差异：

```
Display rV (reflect.Value):
(*rV.typ).size = 8
(*rV.typ).hash = 871609668
(*rV.typ).align = 8
(*rV.typ).fieldAlign = 8
(*rV.typ).kind = 22
(*(*rV.typ).string) = "*os.File"

(*(*(*rV.typ).uncommonType).methods[0].name) = "Chdir"
(*(*(*(*rV.typ).uncommonType).methods[0].mtyp).string) = "func() error"
(*(*(*(*rV.typ).uncommonType).methods[0].typ).string) = "func(*os.File) error"
...
```

观察下面两个例子的区别：

```
var i interface{} = 3

Display("i", i)
// Output:
// Display i (int):
// i = 3

Display("&i", &i)
// Output:
// Display &i (*interface {}):
// (*&i).type = int
// (*&i).value = 3
```

在第一个例子中，Display 函数将调用 reflect.ValueOf(i)，它返回一个 Int 类型的值。正如我们在 12.2 节中
提到的，reflect.ValueOf 总是返回一个值的具体类型，因爲它是从一个接口值提取的内容。

在第二个例子中，Display 函数调用的是 reflect.ValueOf(&i)，它返回一个指向 i 的指针，对应 Ptr 类型。在 switch 的 Ptr 分支中，通过调用 Elem 来返回这个值，返回一个 Value 来表示 i，对应 Interface 类型。一个间 接获得的 Value，就像这一个，可能代表任意类型的值，包括接口类型。内部的 display 函数递归调用自 身，这次它将打印接口的动态类型和值。

目前的实现，Display 如果显示一个带环的数据结构将会陷入死循环，例如首位项链的链表：

```
// a struct that points to itself
type Cycle struct{ Value int; Tail *Cycle }
var c Cycle
c = Cycle{42, &c}
Display("c", c)
```

Display 会永远不停地进行深度递归打印：

```
Display c (display.Cycle):
c.Value = 42
(*c.Tail).Value = 42
(*(*c.Tail).Tail).Value = 42
(*(*(*c.Tail).Tail).Tail).Value = 42
...ad infinitum...
```

许多 Go 语言程序都包含了一些循环的数据结果。Display 支持这类带环的数据结构是比较棘手的，需要增加一个额外的记録访问的路径；代价是昂贵的。一般的解决方案是采用不安全的语言特性，我们将在 13.3 节看到具体的解决方案。

带环的数据结构很少会对 fmt.Sprint 函数造成问题，因爲它很少尝试打印完整的数据结构。例如，当它遇 到一个指针的时候，它只是简单第打印指针的数值。虽然，在打印包含自身的 slice 或 map 时可能遇到睏 难，但是不保证处理这种是罕见情况却可以避免额外的麻烦。

练习 12.1： 扩展 Displayhans，以便它可以显示包含以结构体或数组作爲 map 的 key 类型的值。

练习 12.2： 增强 display 函数的稳健性，通过记録边界的步数来确保在超出一定限制前放弃递归。（在 13.3 节，我们会看到另一种探测数据结构是否存在环的技术。）

示例: 编码 S 表达式

12.4. 示例: 编码 S 表达式

Display 是一个用于显示结构化数据的调试工具，但是它并不能将任意的 Go 语言对象编码爲通用消息然后用于进程间通信。

正如我们在 4.5 节中中看到的，Go 语言的标准库支持了包括 JSON、XML 和 ASN.1 等多种编码格式。还有另 一种依然被广泛使用的格式是 S 表达式格式，采用类似 Lisp 语言的语法。但是和其他编码格式不同的 是，Go 语言自带的标准库并不支持 S 表达式，主要是因爲它没有一个公认的标准规范。

在本节中，我们将定义一个包用于将 Go 语言的对象编码爲 S 表达式格式，它支持以下结构：

```
42          integer
"hello"     string (with Go-style quotation)
foo         symbol (an unquoted name)
(1 2 3)     list   (zero or more items enclosed in parentheses)
```

布尔型习惯上使用 t 符号表示 true，空列表或 nil 符号表示 false，但是爲了简单起见，我们暂时忽略布尔类型。同时忽略的还有 chan 管道和函数，因爲通过反射并无法知道它们的确切状态。我们忽略的还浮点数、 复数和 interface。支持它们是练习 12.3 的任务。

我们将 Go 语言的类型编码爲 S 表达式的方法如下。整数和字符串以自然的方式编码。Nil 值编码爲 nil 符号。 数组和 slice 被编码爲一个列表。

结构体被编码爲成员对象的列表，每个成员对象对应一个个仅有两个元素的子列表，其中子列表的第一个元素是成员的名字，子列表的第二个元素是成员的值。Map 被编码爲键值对的列表。传统上，S 表达式使用点状符号列表(key . value)结构来表示 key/value 对，而不是用一个含双元素的列表，不过爲了简单我们 忽略了点状符号列表。

编码是由一个 encode 递归函数完成，如下所示。它的结构本质上和前面的 Display 函数类似：

```
gopl.io/ch12/sexpr

func encode(buf *bytes.Buffer, v reflect.Value) error {
    switch v.Kind() {
    case reflect.Invalid:
        buf.WriteString("nil")

    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        fmt.Fprintf(buf, "%d", v.Int())

    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        fmt.Fprintf(buf, "%d", v.Uint())

    case reflect.String:
        fmt.Fprintf(buf, "%q", v.String())

    case reflect.Ptr:
        return encode(buf, v.Elem())

    case reflect.Array, reflect.Slice: // (value ...)
        buf.WriteByte('(')
        for i := 0; i < v.Len(); i++ {
            if i > 0 {
                buf.WriteByte(' ')
            }
            if err := encode(buf, v.Index(i)); err != nil {
                return err
            }
        }
```

```
            buf.WriteByte(')')

        case reflect.Struct: // ((name value) ...)
            buf.WriteByte('(')
            for i := 0; i < v.NumField(); i++ {
                if i > 0 {
                    buf.WriteByte(' ')
                }
                fmt.Fprintf(buf, "(%s ", v.Type().Field(i).Name)
                if err := encode(buf, v.Field(i)); err != nil {
                    return err
                }
                buf.WriteByte(')')
            }
            buf.WriteByte(')')

        case reflect.Map: // ((key value) ...)
            buf.WriteByte('(')
            for i, key := range v.MapKeys() {
                if i > 0 {
                    buf.WriteByte(' ')
                }
                buf.WriteByte('(')
                if err := encode(buf, key); err != nil {
                    return err
                }
                buf.WriteByte(' ')
                if err := encode(buf, v.MapIndex(key)); err != nil {
                    return err
                }
                buf.WriteByte(')')
            }
            buf.WriteByte(')')

        default: // float, complex, bool, chan, func, interface
            return fmt.Errorf("unsupported type: %s", v.Type())
        }
        return nil
    }
```

Marshal 函数是对 encode 的保证，以保持和 encoding/...下其它包有着相似的 API：

```
    // Marshal encodes a Go value in S-expression form.
    func Marshal(v interface{}) ([]byte, error) {
        var buf bytes.Buffer
        if err := encode(&buf, reflect.ValueOf(v)); err != nil {
            return nil, err
        }
        return buf.Bytes(), nil
    }
```

下面是 Marshal 对 12.3 节的 strangelove 变量编码后的结果：

```
((Title "Dr. Strangelove") (Subtitle "How I Learned to Stop Worrying and Lo
ve the Bomb") (Year 1964) (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sell
ers") ("Pres. Merkin Muffley" "Peter Sellers") ("Gen. Buck Turgidson" "Geor
ge C. Scott") ("Brig. Gen. Jack D. Ripper" "Sterling Hayden") ("Maj. T.J. \
"King\" Kong" "Slim Pickens") ("Dr. Strangelove" "Peter Sellers"))) (Oscars
("Best Actor (Nomin.)" "Best Adapted Screenplay (Nomin.)" "Best Director (N
omin.)" "Best Picture (Nomin.)")) (Sequel nil))
```

整个输出编码爲一行中以减少输出的大小，但是也很难閱读。这里有一个对 S 表达式格式化的约定。编写一个 S 表达式的格式化函数将作爲一个具有挑战性的练习任务；不过 http://gopl.io 也提供了一个简单的版 本。

```
((Title "Dr. Strangelove")
 (Subtitle "How I Learned to Stop Worrying and Love the Bomb")
 (Year 1964)
 (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sellers")
         ("Pres. Merkin Muffley" "Peter Sellers")
         ("Gen. Buck Turgidson" "George C. Scott")
         ("Brig. Gen. Jack D. Ripper" "Sterling Hayden")
         ("Maj. T.J. \"King\" Kong" "Slim Pickens")
         ("Dr. Strangelove" "Peter Sellers")))
 (Oscars ("Best Actor (Nomin.)"
          "Best Adapted Screenplay (Nomin.)"
          "Best Director (Nomin.)"
          "Best Picture (Nomin.)"))
 (Sequel nil))
```

和 fmt.Print、json.Marshal、Display 函数类似，sexpr.Marshal 函数处理带环的数据结构也会陷入死循环。

在 12.6 节中，我们将给出 S 表达式译码器的实现步骤，但是在那之前，我们还需要先了解如果通过反射技术 来更新程序的变量。

练习 12.3： 实现 encode 函数缺少的分支。将布尔类型编码爲 t 和 nil，浮点数编码爲 Go 语言的格式，复数 1+2i 编码爲#C(1.0 2.0)格式。接口编码爲类型名和值对，例如("[]int" (1 2 3))，但是这个形式可能会造成 歧义：reflect.Type.String 方法对于不同的类型可能返回相同的结果。

练习 12.4： 修改 encode 函数，以上面的格式化形式输出 S 表达式。

练习 12.5： 修改 encode 函数，用 JSON 格式代替 S 表达式格式。然后使用标准库提供的 json.Unmarshal 解 码器来验证函数是正确的。

练习 12.6： 修改 encode，作爲一个优化，忽略对是零值对象的编码。

练习 12.7： 创建一个基于流式的 API，用于 S 表达式的译码，和 json.Decoder(§4.5)函数功能类似。

通过 reflect.Value 修改值

12.5. 通过 reflect.Value 修改值

到目前爲止，反射还只是程序中变量的另一种访问方式。然而，在本节中我们将重点讨论如果通过反射机制来修改变量。

回想一下，Go 语言中类似 x、x.f[1]和*p 形式的表达式都可以表示变量，但是其它如 x + 1 和 f(2)则不是变 量。一个变量就是一个可寻址的内存空间，里面存储了一个值，并且存储的值可以通过内存地址来更新。

对于 reflect.Values 也有类似的区别。有一些 reflect.Values 是可取地址的；其它一些则不可以。考虑以下 的声明语句：

```
x := 2                   // value   type    variable?
a := reflect.ValueOf(2)  // 2       int     no
b := reflect.ValueOf(x)  // 2       int     no
c := reflect.ValueOf(&x) // &x      *int    no
d := c.Elem()            // 2       int     yes (x)
```

其中 a 对应的变量则不可取地址。因爲 a 中的值仅仅是整数 2 的拷贝副本。b 中的值也同样不可取地址。c 中
的值还是不可取地址，它只是一个指针 `&x` 的拷贝。实际上，所有通过 reflect.ValueOf(x)返回的 reflect.Value 都是不可取地址的。但是对于 d，它是 c 的解引用方式生成的，指向另一个变量，因此是可取 地址的。我们可以通过调用 reflect.ValueOf(&x).Elem()，来获取任意变量 x 对应的可取地址的 Value 。

我们可以通过调用 reflect.Value 的 CanAddr 方法来判断其是否可以被取地址：

```
fmt.Println(a.CanAddr()) // "false"
fmt.Println(b.CanAddr()) // "false"
fmt.Println(c.CanAddr()) // "false"
fmt.Println(d.CanAddr()) // "true"
```

每当我们通过指针间接地获取的 reflect.Value 都是可取地址的，即使开始的是一个不可取地址的 Value。在
反射机制中，所有关于是否支持取地址的规则都是类似的。例如，slice 的索引表达式 e[i]将隐式地包含一个 指针，它就是可取地址的，即使开始的 e 表达式不支持也没有关系。以此类推，reflect.ValueOf(e).Index(i) 对于的值也是可取地址的，即使原始的 reflect.ValueOf(e)不支持也没有关系 。

要从变量对应的可取地址的 reflect.Value 来访问变量需要三个步骤。第一步是调用 Addr()方法，它返回一 个 Value，里面保存了指向变量的指针。然后是在 Value 上调用 Interface()方法，也就是返回一个 interface{}，里面通用包含指向变量的指针。最后，如果我们知道变量的类型，我们可以使用类型的断言

机制将得到的 interface{}类型的接口强制环爲普通的类型指针。这样我们就可以通过这个普通指针来更新 变量了：

```
x := 2
d := reflect.ValueOf(&x).Elem() // d refers to the variable x
px := d.Addr().Interface().(*int) // px := &x
*px = 3                          // x = 3
fmt.Println(x)                   // "3"
```

或者，不使用指针，而是通过调用可取地址的 reflect.Value 的 reflect.Value.Set 方法来更新对于的值：

```
d.Set(reflect.ValueOf(4))
fmt.Println(x) // "4"
```

Set 方法将在运行时执行和编译时类似的可赋值性约束的检查。以上代码，变量和值都是 int 类型，但是如果变量是 int64 类型，那麼程序将抛出一个 panic 异常，所以关键问题是要确保改类型的变量可以接受对应 的值：

```
d.Set(reflect.ValueOf(int64(5))) // panic: int64 is not assignable to int
```

通用对一个不可取地址的 reflect.Value 调用 Set 方法也会导致 panic 异常：

```
x := 2
b := reflect.ValueOf(x)
b.Set(reflect.ValueOf(3)) // panic: Set using unaddressable value
```

这里有很多用于基本数据类型的 Set 方法：SetInt、SetUint、SetString 和 SetFloat 等。

```
d := reflect.ValueOf(&x).Elem()
d.SetInt(3)
fmt.Println(x) // "3"
```

从某种程度上说，这些 Set 方法总是尽可能地完成任务。以 SetInt 爲例，只要变量是某种类型的有符号整数
就可以工作，即使是一些命名的类型，只要底层数据类型是有符号整数就可以，而且如果对于变量类型值太大的话会被自动截断。但需要谨慎的是：对于一个引用 interface{}类型的 reflect.Value 调用 SetInt 会导致 panic 异常，即使那个 interface{}变量对于整数类型也不行。

```
x := 1
rx := reflect.ValueOf(&x).Elem()
rx.SetInt(2)                    // OK, x = 2
rx.Set(reflect.ValueOf(3))      // OK, x = 3
rx.SetString("hello")           // panic: string is not assignable to int
rx.Set(reflect.ValueOf("hello")) // panic: string is not assignable to int

var y interface{}
ry := reflect.ValueOf(&y).Elem()
ry.SetInt(2)                    // panic: SetInt called on interface Value
ry.Set(reflect.ValueOf(3))      // OK, y = int(3)
ry.SetString("hello")           // panic: SetString called on interface Value
ry.Set(reflect.ValueOf("hello")) // OK, y = "hello"
```

当我们用 Display 显示 os.Stdout 结构时，我们发现反射可以越过 Go 语言的导出规则的限制读取结构体中未
导出的成员，比如在类 Unix 系统上 os.File 结构体中的 fd int 成员。然而，利用反射机制并不能脩改这些未 导出的成员：

```
stdout := reflect.ValueOf(os.Stdout).Elem() // *os.Stdout, an os.File var
fmt.Println(stdout.Type())                  // "os.File"
fd := stdout.FieldByName("fd")
fmt.Println(fd.Int()) // "1"
fd.SetInt(2)          // panic: unexported field
```

一个可取地址的 reflect.Value 会记録一个结构体成员是否是未导出成员，如果是的话则拒絶脩改操作。因此，CanAddr 方法并不能正确反映一个变量是否是可以被脩改的。另一个相关的方法 CanSet 是用于检查对 应的 reflect.Value 是否是可取地址并可被脩改的：

```
fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"
```

示例: 译码 S 表达式

12.6. 示例: 译码 S 表达式

标准库中 encoding/...下每个包中提供的 Marshal 编码函数都有一个对应的 Unmarshal 函数用于译码。例 如，我们在 4.5 节中看到的，要将包含 JSON 编码格式的字节 slice 数据译码爲我们自己的 Movie 类型
（§12.3），我们可以这样做：

```
data := []byte{/* ... */}
var movie Movie
err := json.Unmarshal(data, &movie)
```

Unmarshal 函数使用了反射机制类脩改 movie 变量的每个成员，根据输入的内容爲 Movie 成员创建对应的 map、结构体和 slice。

现在让我们爲 S 表达式编码实现一个简易的 Unmarshal，类似于前面的 json.Unmarshal 标准库函数，对应 我们之前实现的 sexpr.Marshal 函数的逆操作。我们必须提醒一下，一个健壮的和通用的实现通常需要比例 子更多的代码，爲了便于演示我们采用了精简的实现。我们只支持 S 表达式有限的子集，同时处理错误的方 式也比较粗暴，代码的目的是爲了演示反射的用法，而不是构造一个实用的 S 表达式的译码器。

词法分析器 lexer 使用了标准库中的 text/scanner 包将输入流的字节数据解析爲一个个类似注释、标识符、 字符串面值和数字面值之类的标记。输入扫描器 scanner 的 Scan 方法将提前扫描和返回下一个记号，对于 rune 类型。大多数记号，比如 “(” ，对应一个单一 rune 可表示的 Unicode 字符，但是 text/scanner 也可以 用小的负数表示记号标识符、字符串等由多个字符组成的记号。调用 Scan 方法将返回这些记号的类型，接 着调用 TokenText 方法将返回记号对应的文本内容。

因爲每个解析器可能需要多次使用当前的记号，但是 Scan 会一直向前扫描，所有我们包装了一个 lexer 扫描 器辅助类型，用于跟踪最近由 Scan 方法返回的记号。

```
gopl.io/ch12/sexpr

type lexer struct {
    scan  scanner.Scanner
    token rune // the current token
}

func (lex *lexer) next()        { lex.token = lex.scan.Scan() }
func (lex *lexer) text() string { return lex.scan.TokenText() }

func (lex *lexer) consume(want rune) {
    if lex.token != want { // NOTE: Not an example of good error handling.
        panic(fmt.Sprintf("got %q, want %q", lex.text(), want))
    }
    lex.next()
}
```

现在让我们转到语法解析器。它主要包含两个功能。第一个是 read 函数，用于读取 S 表达式的当前标记，然后根据 S 表达式的当前标记更新可取地址的 reflect.Value 对应的变量 v。

```
func read(lex *lexer, v reflect.Value) {
    switch lex.token {
    case scanner.Ident:
        // The only valid identifiers are
        // "nil" and struct field names.
        if lex.text() == "nil" {
            v.Set(reflect.Zero(v.Type()))
            lex.next()
            return
        }
    case scanner.String:
        s, _ := strconv.Unquote(lex.text()) // NOTE: ignoring errors
        v.SetString(s)
        lex.next()
        return
    case scanner.Int:
        i, _ := strconv.Atoi(lex.text()) // NOTE: ignoring errors
        v.SetInt(int64(i))
        lex.next()
        return
    case '(':
        lex.next()
        readList(lex, v)
        lex.next() // consume ')'
        return
    }
    panic(fmt.Sprintf("unexpected token %q", lex.text()))
}
```

我们的 S 表达式使用标识符区分两个不同类型，结构体成员名和 nil 值的指针。read 函数值处理 nil 类型的标
识符。当遇到 scanner.Ident 爲“nil”是，使用 reflect.Zero 函数将变量 v 设置爲零值。而其它任何类型的标 识符，我们都作爲错误处理。后面的 readList 函数将处理结构体的成员名。

一个“(”标记对应一个列表的开始。第二个函数 readList，将一个列表译码到一个聚合类型中（map、结 构体、slice 或数组），具体类型依然于传入待填充变量的类型。每次遇到这种情况，循环继续解析每个元 素直到遇到于开始标记匹配的结束标记“)”，endList 函数用于检测结束标记。

最有趣的部分是递归。最简单的是对数组类型的处理。直到遇到“)”结束标记，我们使用 Index 函数来获取数组每个元素的地址，然后递归调用 read 函数处理。和其它错误类似，如果输入数据导致译码器的引用超出了数组的范围，译码器将抛出 panic 异常。slice 也采用类似方法解析，不同的是我们将爲每个元素创建 新的变量，然后将元素添加到 slice 的末尾。

在循环处理结构体和 map 每个元素时必须译码一个(key value)格式的对应子列表。对于结构体，key 部分 对于成员的名字。和数组类似，我们使用 FieldByName 找到结构体对应成员的变量，然后递归调用 read 函 数处理。对于 map，key 可能是任意类型，对元素的处理方式和 slice 类似，我们创建一个新的变量，然后 递归填充它，最后将新解析到的 key/value 对添加到 map。

```
func readList(lex *lexer, v reflect.Value) {
    switch v.Kind() {
    case reflect.Array: // (item ...)
        for i := 0; !endList(lex); i++ {
            read(lex, v.Index(i))
        }

    case reflect.Slice: // (item ...)
        for !endList(lex) {
            item := reflect.New(v.Type().Elem()).Elem()
            read(lex, item)
            v.Set(reflect.Append(v, item))
        }

    case reflect.Struct: // ((name value) ...)
        for !endList(lex) {
            lex.consume('(')
            if lex.token != scanner.Ident {
                panic(fmt.Sprintf("got token %q, want field name", lex.text()))
            }
            name := lex.text()
            lex.next()
            read(lex, v.FieldByName(name))
            lex.consume(')')
        }

    case reflect.Map: // ((key value) ...)
        v.Set(reflect.MakeMap(v.Type()))
        for !endList(lex) {
            lex.consume('(')
            key := reflect.New(v.Type().Key()).Elem()
            read(lex, key)
            value := reflect.New(v.Type().Elem()).Elem()
            read(lex, value)
            v.SetMapIndex(key, value)
            lex.consume(')')
        }

    default:
        panic(fmt.Sprintf("cannot decode list into %v", v.Type()))
    }
}

func endList(lex *lexer) bool {
    switch lex.token {
    case scanner.EOF:
        panic("end of file")
    case ')':
        return true
    }
    return false
}
```

最后，我们将解析器包装爲导出的 Unmarshal 译码函数，隐藏了一些初始化和清理等边缘处理。内部解析 器以 panic 的方式抛出错误，但是 Unmarshal 函数通过在 defer 语句调用 recover 函数来捕获内部 panic（§5.10），然后返回一个对 panic 对应的错误信息。

```
// Unmarshal parses S-expression data and populates the variable
// whose address is in the non-nil pointer out.
func Unmarshal(data []byte, out interface{}) (err error) {
    lex := &lexer{scan: scanner.Scanner{Mode: scanner.GoTokens}}
    lex.scan.Init(bytes.NewReader(data))
    lex.next() // get the first token
    defer func() {
        // NOTE: this is not an example of ideal error handling.
        if x := recover(); x != nil {
            err = fmt.Errorf("error at %s: %v", lex.scan.Position, x)
        }
    }()
    read(lex, reflect.ValueOf(out).Elem())
    return nil
}
```

生產实现不应该对任何输入问题都用 panic 形式报告，而且应该报告一些错误相关的信息，例如出现错误输
入的行号和位置等。尽管如此，我们希望通过这个例子来展示类似 encoding/json 等包底层代码的实现思 路，以及如何使用反射机制来填充数据结构。

练习 12.8： sexpr.Unmarshal 函数和 json.Unmarshal 一样，都要求在译码前输入完整的字节 slice。定义 一个和 json.Decoder 类似的 sexpr.Decoder 类型，支持从一个 io.Reader 流译码。脩改 sexpr.Unmarshal 函 数，使用这个新的类型实现。

练习 12.9： 编写一个基于标记的 API 用于译码 S 表达式，参考 xml.Decoder（7.14）的风格。你将需要五 种类型的标记：Symbol、String、Int、StartList 和 EndList。

练习 12.10： 扩展 sexpr.Unmarshal 函数，支持布尔型、浮点数和 interface 类型的译码，使用 练习 12.3： 的方案。（提示：要译码接口，你需要将 name 映射到每个支持类型的 reflect.Type。）

获取结构体字段标识

12.7. 获取结构体字段标识

在 4.5 节我们使用构体成员标签用于设置对应 JSON 对应的名字。其中 json 成员标签让我们可以选择成员的 名字和抑制零值成员的输出。在本节，我们将看到如果通过反射机制类获取成员标签。

对于一个 web 服务，大部分 HTTP 处理函数要做的第一件事情就是展开请求中的参数到本地变量中。我们定 义了一个工具函数，叫 params.Unpack，通过使用结构体成员标签机制来让 HTTP 处理函数解析请求参数 更方便。

首先，我们看看如何使用它。下面的 search 函数是一个 HTTP 请求处理函数。它定义了一个匿名结构体类型
的变量，用结构体的每个成员表示 HTTP 请求的参数。其中结构体成员标签指明了对于请求参数的名字，爲 了减少 UTRL 的长度这些参数名通常都是神祕的缩略词。Unpack 将请求参数填充到合适的结构体成员中， 这样我们可以方便地通过合适的类型类来访问这些参数。

```
gopl.io/ch12/search

import "gopl.io/ch12/params"

// search implements the /search URL endpoint.
func search(resp http.ResponseWriter, req *http.Request) {
    var data struct {
        Labels     []string `http:"l"`
        MaxResults int      `http:"max"`
        Exact      bool     `http:"x"`
    }
    data.MaxResults = 10 // set default
    if err := params.Unpack(req, &data); err != nil {
        http.Error(resp, err.Error(), http.StatusBadRequest) // 400
        return
    }

    // ...rest of handler...
    fmt.Fprintf(resp, "Search: %+v\n", data)
}
```

下面的 Unpack 函数主要完成三件事情。第一，它调用 req.ParseForm()来解析 HTTP 请求。然后，req.Form 将包含所有的请求参数，不管 HTTP 客户端使用的是 GET 还是 POST 请求方法。

下一步，Unpack 函数将构建每个结构体成员有效参数名字到成员变量的映射。如果结构体成员有成员标签 的话，有效参数名字可能和实际的成员名字不相同。reflect.Type 的 Field 方法将返回一个 reflect.StructField，里面含有每个成员的名字、类型和可选的成员标签等信息。其中成员标签信息对应 reflect.StructTag 类型的字符串，并且提供了 Get 方法用于解析和根据特定 key 提取的子串，例如这里的 http:"..."形式的子串。

```
gopl.io/ch12/params

// Unpack populates the fields of the struct pointed to by ptr
// from the HTTP request parameters in req.
func Unpack(req *http.Request, ptr interface{}) error {
    if err := req.ParseForm(); err != nil {
        return err
    }

    // Build map of fields keyed by effective name.
    fields := make(map[string]reflect.Value)
    v := reflect.ValueOf(ptr).Elem() // the struct variable
    for i := 0; i < v.NumField(); i++ {
        fieldInfo := v.Type().Field(i) // a reflect.StructField
        tag := fieldInfo.Tag           // a reflect.StructTag
        name := tag.Get("http")
        if name == "" {
            name = strings.ToLower(fieldInfo.Name)
        }
        fields[name] = v.Field(i)
    }

    // Update struct field for each parameter in the request.
    for name, values := range req.Form {
        f := fields[name]
        if !f.IsValid() {
            continue // ignore unrecognized HTTP parameters
        }
        for _, value := range values {
            if f.Kind() == reflect.Slice {
                elem := reflect.New(f.Type().Elem()).Elem()
                 if err := populate(elem, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
                f.Set(reflect.Append(f, elem))
            } else {
                if err := populate(f, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
            }
        }
    }
    return nil
}
```

最后，Unpack 遍历 HTTP 请求的 name/valu 参数键值对，并且根据更新相应的结构体成员。回想一下，同
一个名字的参数可能出现多次。如果发生这种情况，并且对应的结构体成员是一个 slice，那麼就将所有的 参数添加到 slice 中。其它情况，对应的成员值将被覆盖，只有最后一次出现的参数值才是起作用的。

populate 函数小心用请求的字符串类型参数值来填充单一的成员 v（或者是 slice 类型成员中的单一的元 素）。目前，它仅支持字符串、有符号整数和布尔型。其中其它的类型将留做练习任务。

```
func populate(v reflect.Value, value string) error {
    switch v.Kind() {
    case reflect.String:
        v.SetString(value)

    case reflect.Int:
        i, err := strconv.ParseInt(value, 10, 64)
        if err != nil {
            return err
        }
        v.SetInt(i)

    case reflect.Bool:
        b, err := strconv.ParseBool(value)
        if err != nil {
            return err
        }
        v.SetBool(b)

    default:
        return fmt.Errorf("unsupported kind %s", v.Type())
    }
    return nil
}
```

如果我们上上面的处理程序添加到一个 web 服务器，则可以産生以下的会话：

```
$ go build gopl.io/ch12/search
$ ./search &
$ ./fetch 'http://localhost:12345/search'
Search: {Labels:[] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming&max=100'
Search: {Labels:[golang programming] MaxResults:100 Exact:false}
$ ./fetch 'http://localhost:12345/search?x=true&l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:true}
$ ./fetch 'http://localhost:12345/search?q=hello&x=123'
x: strconv.ParseBool: parsing "123": invalid syntax
$ ./fetch 'http://localhost:12345/search?q=hello&max=lots'
max: strconv.ParseInt: parsing "lots": invalid syntax
```

练习 12.11： 编写相应的 Pack 函数，给定一个结构体值，Pack 函数将返回合并了所有结构体成员和值的 URL。

练习 12.12： 扩展成员标签以表示一个请求参数的有效值规则。例如，一个字符串可以是有效的 email 地址或一个信用卡号码，还有一个整数可能需要是有效的邮政编码。修改 Unpack 函数以检查这些规则。

练习 12.13： 修改 S 表达式的编码器（§12.4）和解码器（§12.6），采用和 encoding/json 包（§4.5）类似 的方式使用成员标签中的 sexpr:"..."字符串。

显示一个类型的方法集

12.8. 显示一个类型的方法集

我们的最后一个例子是使用 reflect.Type 来打印任意值的类型和枚举它的方法：

```
gopl.io/ch12/methods

// Print prints the method set of the value x.
func Print(x interface{}) {
    v := reflect.ValueOf(x)
    t := v.Type()
    fmt.Printf("type %s\n", t)

    for i := 0; i < v.NumMethod(); i++ {
        methType := v.Method(i).Type()
        fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,
            strings.TrimPrefix(methType.String(), "func"))
    }
}
```

reflect.Type 和 reflect.Value 都提供了一个 Method 方法。每次 t.Method(i)调用将一个 reflect.Method 的
实例，对应一个用于描述一个方法的名称和类型的结构体。每次 v.Method(i)方法调用都返回一个 reflect.Value 以表示对应的值（§6.4），也就是一个方法是帮到它的接收者的。使用 reflect.Value.Call 方 法（我们之类没有演示），将可以调用一个 Func 类型的 Value，但是这个例子中只用到了它的类型。

这是属于 time.Duration 和`*strings.Replacer`两个类型的方法：

```
methods.Print(time.Hour)
// Output:
// type time.Duration
// func (time.Duration) Hours() float64
// func (time.Duration) Minutes() float64
// func (time.Duration) Nanoseconds() int64
// func (time.Duration) Seconds() float64
// func (time.Duration) String() string

methods.Print(new(strings.Replacer))
// Output:
// type *strings.Replacer
// func (*strings.Replacer) Replace(string) string
// func (*strings.Replacer) WriteString(io.Writer, string) (int, error)
```

`

几点忠告

12.9. 几点忠告

虽然反射提供的 API 远多于我们讲到的，我们前面的例子主要是给出了一个方向，通过反射可以实现哪些功 能。反射是一个强大并富有表达力的工具，但是它应该被小心地使用，原因有三。

第一个原因是，基于反射的代码是比较脆弱的。对于每一个会导致编译器报告类型错误的问题，在反射中都有与之相对应的问题，不同的是编译器会在构建时马上报告错误，而反射则是在眞正运行到的时候才会抛出 panic 异常，可能是写完代码很久之后的时候了，而且程序也可能运行了很长的时间。

以前面的 readList 函数（§12.6）爲例，爲了从输入读取字符串并填充 int 类型的变量而调用的 reflect.Value.SetString 方法可能导致 panic 异常。絶大多数使用反射的程序都有类似的风险，需要非常小 心地检查每个 reflect.Value 的对于值的类型、是否可取地址，还有是否可以被脩改等。

避免这种因反射而导致的脆弱性的问题的最好方法是将所有的反射相关的使用控制在包的内部，如果可能的话避免在包的 API 中直接暴露 reflect.Value 类型，这样可以限制一些非法输入。如果无法做到这一点，在 每个有风险的操作前指向额外的类型检查。以标准库中的代码爲例，当 fmt.Printf 收到一个非法的操作数 是，它并不会抛出 panic 异常，而是打印相关的错误信息。程序虽然还有 BUG，但是会更加容易诊断。

```
fmt.Printf("%d %s\n", "hello", 42) // "%!d(string=hello) %!s(int=42)"
```

反射同样降低了程序的安全性，还影响了自动化重构和分析工具的准确性，因爲它们无法识别运行时才能确认的类型信息。

避免使用反射的第二个原因是，即使对应类型提供了相同文档，但是反射的操作不能做静态类型检查，而且大量反射的代码通常难以理解。总是需要小心翼翼地爲每个导出的类型和其它接受 interface{}或 reflect.Value 类型参数的函数维护帮助文档。

第三个原因，基于反射的代码通常比正常的代码运行速度慢一到两个数量级。对于一个典型的项目，大部分函数的性能和程序的整体性能关系不大，所以使用反射可能会使程序更加清晰。测试是一个特别适合使用反射的场景，因爲每个测试的数据集都很小。但是对于性能关键路径的函数，最好避免使用反射。

底层编程

第 13 章 底层编程

Go 语言的设计包含了诸多安全策略，限制了可能导致程序运行出现错误的用法。编译时类型检查检查可以 发现大多数类型不匹配的操作，例如两个字符串做减法的错误。字符串、map、slice 和 chan 等所有的内置 类型，都有严格的类型转换规则。

对于无法静态检测到的错误，例如数组访问越界或使用空指针，运行时动态检测可以保证程序在遇到问题的时候立即终止并打印相关的错误信息。自动内存管理（垃圾内存自动回收）可以消除大部分野指针和内存泄漏相关的问题。

Go 语言的实现刻意隐藏了很多底层细节。我们无法知道一个结构体真实的内存布局，也无法获取一个运行 时函数对应的机器码，也无法知道当前的 goroutine 是运行在哪个操作系统线程之上。事实上，Go 语言的 调度器会自己决定是否需要将某个 goroutine 从一个操作系统线程转移到另一个操作系统线程。一个指向 变量的指针也并没有展示变量真实的地址。因为垃圾回收器可能会根据需要移动变量的内存位置，当然变 量对应的地址也会被自动更新。

总的来说，Go 语言的这些特性使得 Go 程序相比较低级的 C 语言来说更容易预测和理解，程序也不容易崩 溃。通过隐藏底层的实现细节，也使得 Go 语言编写的程序具有高度的可移植性，因为语言的语义在很大程 度上是独立于任何编译器实现、操作系统和 CPU 系统结构的（当然也不是完全绝对独立：例如 int 等类型就 依赖于 CPU 机器字的大小，某些表达式求值的具体顺序，还有编译器实现的一些额外的限制等）。

有时候我们可能会放弃使用部分语言特性而优先选择更好具有更好性能的方法，例如需要与其他语言编写的库互操作，或者用纯 Go 语言无法实现的某些函数。

在本章，我们将展示如何使用 unsafe 包来襬脱 Go 语言规则带来的限制，讲述如何创建 C 语言函数库的绑定，以及如何进行系统调用。

本章提供的方法不应该轻易使用（译注：属于黑魔法，虽然可能功能很强大，但是也容易误伤到自己）。如果没有处理好细节，它们可能导致各种不可预测的并且隐晦的错误，甚至连有经验的的 C 语言程序员也无 法理解这些错误。使用 unsafe 包的同时也放弃了 Go 语言保证与未来版本的兼容性的承诺，因为它必然会在 有意无意中会使用很多实现的细节，而这些实现的细节在未来的 Go 语言中很可能会被改变。

要注意的是，unsafe 包是一个采用特殊方式实现的包。虽然它可以和普通包一样的导入和使用，但它实际 上是由编译器实现的。它提供了一些访问语言内部特性的方法，特别是内存布局相关的细节。将这些特性 封装到一个独立的包中，是为在极少数情况下需要使用的时候，同时引起人们的注意（译注：因为看包的 名字就知道使用 unsafe 包是不安全的）。此外，有一些环境因为安全的因素可能限制这个包的使用。

不过 unsafe 包被广泛地用于比较低级的包, 例如 runtime、os、syscall 还有 net 包等，因为它们需要和操作 系统密切配合，但是对于普通的程序一般是不需要使用 unsafe 包的。

unsafe.Sizeof, Alignof 和 Offsetof

13.1. unsafe.Sizeof, Alignof 和 Offsetof

unsafe.Sizeof 函数返回操作数在内存中的字节大小，参数可以是任意类型的表达式，但是它并不会对表达 式进行求值。一个 Sizeof 函数调用是一个对应 uintptr 类型的常量表达式，因此返回的结果可以用作数组类 型的长度大小，或者用作计算其他的常量。

```
import "unsafe"
fmt.Println(unsafe.Sizeof(float64(0))) // "8"
```

Sizeof 函数返回的大小只包括数据结构中固定的部分，例如字符串对应结构体中的指针和字符串长度部分，但是并不包含指针指向的字符串的内容。Go 语言中非聚合类型通常有一个固定的大小，尽管在不同工 具链下生成的实际大小可能会有所不同。考虑到可移植性，引用类型或包含引用类型的大小在 32 位平台上 是 4 个字节，在 64 位平台上是 8 个字节。

计算机在加载和保存数据时，如果内存地址合理地对齐的将会更有效率。例如 2 字节大小的 int16 类型的变 量地址应该是偶数，一个 4 字节大小的 rune 类型变量的地址应该是 4 的倍数，一个 8 字节大小的 float64、 uint64 或 64-bit 指针类型变量的地址应该是 8 字节对齐的。但是对于再大的地址对齐倍数则是不需要的，即 使是 complex128 等较大的数据类型最多也只是 8 字节对齐。

由于地址对齐这个因素，一个聚合类型（结构体或数组）的大小至少是所有字段或元素大小的总和，或者更大因爲可能存在内存空洞。内存空洞是编译器自动添加的没有被使用的内存空间，用于保证后面每个字段或元素的地址相对于结构或数组的开始地址能够合理地对齐（译注：内存空洞可能会存在一些随机数据，可能会对用 unsafe 包直接操作内存的处理産生影响）。

类型	大小
bool	1 个字节
intN, uintN, floatN, complexN	N/8 个字节(例如 float64 是 8 个字节) int, uint, uintptr	1 个机器字
*T	1 个机器字
string	2 个机器字(data,len)
[]T	3 个机器字(data,len,cap) map	1 个机器字
func	1 个机器字
chan	1 个机器字
interface	2 个机器字(type,value) Go 语言的规范并没有要求一个字段的声明顺序和内存中的顺序是一致的，所以理论上一个编译器可以随意

地重新排列每个字段的内存位置，随然在写作本书的时候编译器还没有这麼做。下面的三个结构体虽然有着相同的字段，但是第一种写法比另外的两个需要多 50%的内存。

```
                        // 64-bit  32-bit
struct{ bool; float64; int16 } // 3 words 4words
struct{ float64; int16; bool } // 2 words 3words
struct{ bool; int16; float64 } // 2 words 3words
```

关于内存地址对齐算法的细节超出了本书的范围，也不是每一个结构体都需要担心这个问题，不过有效的包装可以使数据结构更加紧凑（译注：未来的 Go 语言编译器应该会默认优化结构体的顺序，当然用于应该 也能够指定具体的内存布局，相同讨论请参考 Issue10014 ），内存使用率和性能都可能会受益。

`unsafe.Alignof` 函数返回对应参数的类型需要对齐的倍数. 和 Sizeof 类似, Alignof 也是返回一个常量表达式, 对应一个常量. 通常情况下布尔和数字类型需要对齐到它们本身的大小(最多 8 个字节), 其它的类型对 齐到机器字大小.

`unsafe.Offsetof` 函数的参数必须是一个字段 `x.f`, 然后返回 `f` 字段相对于 `x` 起始地址的偏移量, 包括可能的空洞.

图 13.1 显示了一个结构体变量 x 以及其在 32 位和 64 位机器上的典型的内存. 灰色区域是空洞.

```
var x struct {
    a bool
    b int16
    c []int
}
```

下面显示了对 x 和它的三个字段调用 unsafe 包相关函数的计算结果：

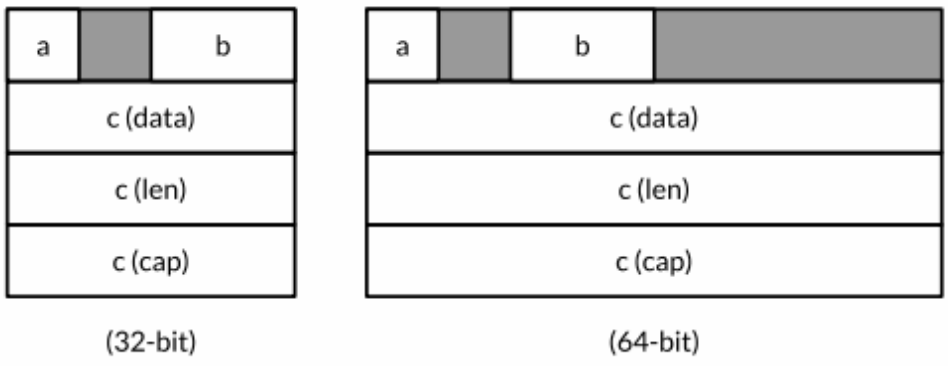


Figure 13.1. Holes in a struct.

32 位系统：

```
Sizeof(x)   = 16  Alignof(x)   = 4
Sizeof(x.a) = 1   Alignof(x.a) = 1 Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2 Offsetof(x.b) = 2
Sizeof(x.c) = 12  Alignof(x.c) = 4 Offsetof(x.c) = 4
```

64 位系统：

```
Sizeof(x)   = 32  Alignof(x)   = 8
Sizeof(x.a) = 1   Alignof(x.a) = 1 Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2 Offsetof(x.b) = 2
Sizeof(x.c) = 24  Alignof(x.c) = 8 Offsetof(x.c) = 8
```

虽然这几个函数在不安全的 unsafe 包，但是这几个函数调用并不是眞的不安全，特别在需要优化内存空间时它们返回的结果对于理解原生的内存布局很有帮助。

unsafe.Pointer

13.2. unsafe.Pointer

大多数指针类型会写成 `*T`，表示是“一个指向 T 类型变量的指针”。unsafe.Pointer 是特别定义的一种指
针类型（译注：类似 C 语言中的`void*`类型的指针），它可以包含任意类型变量的地址。当然，我们不可以直接通过`*p`来获取 unsafe.Pointer 指针指向的眞实变量的值，因爲我们并不知道变量的具体类型。和普
通指针一样，unsafe.Pointer 指针也是可以比较的，并且支持和 nil 常量比较判断是否爲空指针。

一个普通的`*T`类型指针可以被转化爲 unsafe.Pointer 类型指针，并且一个 unsafe.Pointer 类型指针也可以
被转回普通的指针，被转回普通的指针类型并不需要和原始的`*T`类型相同。通过将`*float64`类型指针转化爲`*uint64`类型指针，我们可以查看一个浮点数变量的位模式。

```
package math

func Float64bits(f float64) uint64 { return *(*uint64)(unsafe.Pointer(&f)) }

fmt.Printf("%#016x\n", Float64bits(1.0)) // "0x3ff0000000000000"
```

通过转爲新类型指针，我们可以更新浮点数的位模式。通过位模式操作浮点数是可以的，但是更重要的意义是指针转换语法让我们可以在不破坏类型系统的前提下向内存写入任意的值。

一个 unsafe.Pointer 指针也可以被转化爲 uintptr 类型，然后保存到指针型数值变量中（译注：这只是和当 前指针相同的一个数字值，并不是一个指针），然后用以做必要的指针数值运算。（第三章内容，uintptr 是一个无符号的整型数，足以保存一个地址）这种转换虽然也是可逆的，但是将 uintptr 转爲 unsafe.Pointer 指针可能会破坏类型系统，因爲并不是所有的数字都是有效的内存地址。

许多将 unsafe.Pointer 指针转爲原生数字，然后再转回爲 unsafe.Pointer 类型指针的操作也是不安全的。比
如下面的例子需要将变量 x 的地址加上 b 字段地址偏移量转化`*int16`类型指针，然后通过该指针更新爲 x.b：

```
//gopl.io/ch13/unsafeptr

var x struct {
    a bool
    b int16
    c []int
}

// 和 pb := &x.b 等價
pb := (*int16)(unsafe.Pointer(
    uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
*pb = 42
fmt.Println(x.b) // "42"
```

上面的写法尽管很繁琐，但在这里并不是一件坏事，因爲这些功能应该很谨慎地使用。不要试图引入一个 uintptr 类型的临时变量，因爲它可能会破坏代码的安全性（译注：这是眞正可以体会 unsafe 包爲何不安全 的例子）。下面段代码是错误的：

```
// NOTE: subtly incorrect!
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
pb := (*int16)(unsafe.Pointer(tmp))
*pb = 42
```

產生错误的原因很微妙。有时候垃圾回收器会移动一些变量以降低内存碎片等问题。这类垃圾回收器被称爲移动 GC。当一个变量被移动，所有的保存改变量旧地址的指针必须同时被更新爲变量移动后的新地址。 从垃圾收集器的视角来看，一个 unsafe.Pointer 是一个指向变量的指针，因此当变量被移动是对应的指针 也必须被更新；但是 uintptr 类型的临时变量只是一个普通的数字，所以其值不应该被改变。上面错误的代 码因爲引入一个非指针的临时变量 tmp，导致垃圾收集器无法正确识别这个是一个指向变量 x 的指针。当第
二个语句执行时，变量 x 可能已经被转移，这时候临时变量 tmp 也就不再是现在 `&x.b` 地址。第三个向之的 前无效地址空间的赋值语句将彻底摧毁整个程序！

还有很多类似原因导致的错误。例如这条语句：

```
pT := uintptr(unsafe.Pointer(new(T))) // 提示: 錯誤!
```

这里并没有指针引用 `new` 新创建的变量，因此该语句执行完成之后，垃圾收集器有权马上回收其内存空间，所以返回的 pT 将是无效的地址。

虽然目前的 Go 语言实现还没有使用移动 GC（译注：未来可能实现），但这不该是编写错误代码侥幸的理由：当前的 Go 语言实现已经有移动变量的场景。在 5.2 节我们提到 goroutine 的栈是根据需要动态增长的。 当发送栈动态增长的时候，原来栈中的所以变量可能需要被移动到新的更大的栈中，所以我们并不能确保 变量的地址在整个使用周期内是不变的。

在编写本文时，还没有清晰的原则来指引 Go 程序员，什麼样的 unsafe.Pointer 和 uintptr 的转换是不安全的

（参考 Issue7192 ）. 译注: 该问题已经关闭），因此我们强烈建议按照最坏的方式处理。将所有包含变量地址的 uintptr 类型变量当作 BUG 处理，同时减少不必要的 unsafe.Pointer 类型到 uintptr 类型的转换。在第 一个例子中，有三个转换——字段偏移量到 uintptr 的转换和转回 unsafe.Pointer 类型的操作——所有的转 换全在一个表达式完成。

当调用一个库函数，并且返回的是 uintptr 类型地址时（译注：普通方法实现的函数不尽量不要返回该类型。下面例子是 reflect 包的函数，reflect 包和 unsafe 包一样都是采用特殊技术实现的，编译器可能给它们 开了后门），比如下面反射包中的相关函数，返回的结果应该立即转换爲 unsafe.Pointer 以确保指针指向 的是相同的变量。

```
package reflect

func (Value) Pointer() uintptr
func (Value) UnsafeAddr() uintptr
func (Value) InterfaceData() [2]uintptr // (index 1)
```

示例: 深度相等判断

13.3. 示例: 深度相等判断

来自 reflect 包的 DeepEqual 函数可以对两个值进行深度相等判断。DeepEqual 函数使用内建的==比较操 作符对基础类型进行相等判断，对于复合类型则递归该变量的每个基础类型然后做类似的比较判断。因爲 它可以工作在任意的类型上，甚至对于一些不支持==操作运算符的类型也可以工作，因此在一些测试代码 中广泛地使用该函数。比如下面的代码是用 DeepEqual 函数比较两个字符串数组是否相等。

```
func TestSplit(t *testing.T) {
    got := strings.Split("a:b:c", ":")
    want := []string{"a", "b", "c"};
    if !reflect.DeepEqual(got, want) { /* ... */ }
}
```

尽管 DeepEqual 函数很方便，而且可以支持任意的数据类型，但是它也有不足之处。例如，它将一个 nil 值
的 map 和非 nil 值但是空的 map 视作不相等，同样 nil 值的 slice 和非 nil 但是空的 slice 也视作不相等。

```
var a, b []string = nil, []string{}
fmt.Println(reflect.DeepEqual(a, b)) // "false"

var c, d map[string]int = nil, make(map[string]int)
fmt.Println(reflect.DeepEqual(c, d)) // "false"
```

我们希望在这里实现一个自己的 Equal 函数，用于比较类型的值。和 DeepEqual 函数类似的地方是它也是

基于 slice 和 map 的每个元素进行递归比较，不同之处是它将 nil 值的 slice（map 类似）和非 nil 值但是空的 slice 视作相等的值。基础部分的比较可以基于 reflect 包完成，和 12.3 章的 Display 函数的实现方法类似。同 样，我们也定义了一个内部函数 equal，用于内部的递归比较。读者目前不用关心 seen 参数的具体含义。 对于每一对需要比较的 x 和 y，equal 函数首先检测它们是否都有效（或都无效），然后检测它们是否是相同 的类型。剩下的部分是一个巨大的 switch 分支，用于相同基础类型的元素比较。因爲页面空间的限制，我 们省略了一些相似的分支。

```
gopl.io/ch13/equal
func equal(x, y reflect.Value, seen map[comparison]bool) bool {
    if !x.IsValid() || !y.IsValid() {
        return x.IsValid() == y.IsValid()
    }
    if x.Type() != y.Type() {
        return false
    }

    // ...cycle check omitted (shown later)...

    switch x.Kind() {
    case reflect.Bool:
        return x.Bool() == y.Bool()
    case reflect.String:
        return x.String() == y.String()

    // ...numeric cases omitted for brevity...

    case reflect.Chan, reflect.UnsafePointer, reflect.Func:
        return x.Pointer() == y.Pointer()
    case reflect.Ptr, reflect.Interface:
        return equal(x.Elem(), y.Elem(), seen)
    case reflect.Array, reflect.Slice:
        if x.Len() != y.Len() {
            return false
        }
        for i := 0; i < x.Len(); i++ {
            if !equal(x.Index(i), y.Index(i), seen) {
                return false
            }
        }
        return true

    // ...struct and map cases omitted for brevity...
    }
    panic("unreachable")
}
```

和前面的建议一样，我们并不公开 reflect 包相关的接口，所以导出的函数需要在内部自己将变量转爲 reflect.Value 类型。

```
// Equal reports whether x and y are deeply equal.
func Equal(x, y interface{}) bool {
    seen := make(map[comparison]bool)
    return equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)
}

type comparison struct {
    x, y unsafe.Pointer
    treflect.Type
}
```

爲了确保算法对于有环的数据结构也能正常退出，我们必须记録每次已经比较的变量，从而避免进入第二次的比较。Equal 函数分配了一组用于比较的结构体，包含每对比较对象的地址（unsafe.Pointer 形式保存）和类型。我们要记録类型的原因是，有些不同的变量可能对应相同的地址。例如，如果 x 和 y 都是数组 类型，那麼 x 和 x[0]将对应相同的地址，y 和 y[0]也是对应相同的地址，这可以用于区分 x 与 y 之间的比较或 x[0]与 y[0]之间的比较是否进行过了。

```
// cycle check
if x.CanAddr() && y.CanAddr() {
    xptr := unsafe.Pointer(x.UnsafeAddr())
    yptr := unsafe.Pointer(y.UnsafeAddr())
    if xptr == yptr {
        return true // identical references
    }
    c := comparison{xptr, yptr, x.Type()}
    if seen[c] {
        return true // already seen
    }
    seen[c] = true
}
```

这是 Equal 函数用法的例子:

```
fmt.Println(Equal([]int{1, 2, 3}, []int{1, 2, 3}))        // "true"
fmt.Println(Equal([]string{"foo"}, []string{"bar"}))      // "false"
fmt.Println(Equal([]string(nil), []string{}))             // "true"
fmt.Println(Equal(map[string]int(nil), map[string]int{})) // "true"
```

Equal 函数甚至可以处理类似 12.3 章中导致 Display 陷入陷入死循环的带有环的数据。

```
// Circular linked lists a -> b -> a and c -> c.
type link struct {
    value string
    tail *link
}
a, b, c := &link{value: "a"}, &link{value: "b"}, &link{value: "c"}
a.tail, b.tail, c.tail = b, a, c
fmt.Println(Equal(a, a)) // "true"
fmt.Println(Equal(b, b)) // "true"
fmt.Println(Equal(c, c)) // "true"
fmt.Println(Equal(a, b)) // "false"
fmt.Println(Equal(a, c)) // "false"
```

练习 13.1： 定义一个深比较函数，对于十亿以内的数字比较，忽略类型差异。

练习 13.2： 编写一个函数，报告其参数是否循环数据结构。

通过 cgo 调用 C 代码

13.4. 通过 cgo 调用 C 代码

Go 程序可能会遇到要访问 C 语言的某些硬件驱动函数的场景，或者是从一个 C++语言实现的嵌入式数据库 查询记録的场景，或者是使用 Fortran 语言实现的一些线性代数库的场景。C 语言作爲一个通用语言，很多 库会选择提供一个 C 兼容的 API，然后用其他不同的编程语言实现（译者：Go 语言需要也应该拥抱这些巨大 的代码遗産）。

在本节中，我们将构建一个简易的数据压缩程序，使用了一个 Go 语言自带的叫 cgo 的用于支援 C 语言函数 调用的工具。这类工具一般被称爲 *foreign-function interfaces* （简称 ffi），并且在类似工具中 cgo 也不 是唯一的。SWIG（ http://swig.org ）是另一个类似的且被广泛使用的工具，SWIG 提供了很多复杂特性 以支援 C++的特性，但 SWIG 并不是我们要讨论的主题。

在标准库的 compress/... 子包有很多流行的压缩算法的编码和解码实现，包括流行的 LZW 压缩算法（Unix 的 compress 命令用的算法）和 DEFLATE 压缩算法（GNU gzip 命令用的算法）。这些包的 API 的细 节虽然有些差异，但是它们都提供了针对 io.Writer 类型输出的压缩接口和提供了针对 io.Reader 类型输入 的解压缩接口。例如：

```
package gzip // compress/gzip
func NewWriter(w io.Writer) io.WriteCloser
func NewReader(r io.Reader) (io.ReadCloser, error)
```

bzip2 压缩算法，是基于优雅的 Burrows-Wheeler 变换算法，运行速度比 gzip 要慢，但是可以提供更高的
压缩比。标准库的 compress/bzip2 包目前还没有提供 bzip2 压缩算法的实现。完全从头开始实现是一个压

缩算法是一件繁琐的工作，而且 http://bzip.org 已经有现成的 libbzip2 的开源实现，不仅文档齐全而且性 能又好。

如果是比较小的 C 语言库，我们完全可以用纯 Go 语言重新实现一遍。如果我们对性能也没有特殊要求的话，我们还可以用 os/exec 包的方法将 C 编写的应用程序作爲一个子进程运行。只有当你需要使用复杂而且 性能更高的底层 C 接口时，就是使用 cgo 的场景了（译注：用 os/exec 包调用子进程的方法会导致程序运行 时依赖那个应用程序）。下面我们将通过一个例子讲述 cgo 的具体用法。

译注：本章采用的代码都是最新的。因爲之前已经出版的书中包含的代码只能在 Go1.5 之前使用。从 Go1.6 开始，Go 语言已经明确规定了哪些 Go 语言指针可以之间传入 C 语言函数。新代码重点是增加了 bz2alloc 和 bz2free 的两个函数，用于 bz_stream 对象空间的申请和释放操作。下面是新代码中增加的注释，帮助这个 问题：

```
// The version of this program that appeared in the first and second
// printings did not comply with the proposed rules for passing
// pointers between Go and C, described here:
// https://github.com/golang/proposal/blob/master/design/12416-cgo-pointers.md
//
// The rules forbid a C function like bz2compress from storing 'in'
// and 'out' (pointers to variables allocated by Go) into the Go
// variable 's', even temporarily.
//
// The version below, which appears in the third printing, has been
// corrected.  To comply with the rules, the bz_stream variable must
// be allocated by C code.  We have introduced two C functions,
// bz2alloc and bz2free, to allocate and free instances of the
// bz_stream type.  Also, we have changed bz2compress so that before
// it returns, it clears the fields of the bz_stream that contain
// pointers to Go variables.
```

要使用 libbzip2，我们需要先构建一个 bz_stream 结构体，用于保持输入和输出缓存。然后有三个函数：BZ2_bzCompressInit 用于初始化缓存，BZ2_bzCompress 用于将输入缓存的数据压缩到输出缓 存，BZ2_bzCompressEnd 用于释放不需要的缓存。（目前不要担心包的具体结构, 这个例子的目的就是演 示各个部分如何组合在一起的。）

我们可以在 Go 代码中直接调用 BZ2_bzCompressInit 和 BZ2_bzCompressEnd，但是对于 BZ2_bzCompress，我们将定义一个 C 语言的包装函数，用它完成眞正的工作。下面是 C 代码，对应一个独 立的文件。

```
gopl.io/ch13/bzip

/* This file is gopl.io/ch13/bzip/bzip2.c,          */
/* a simple wrapper for libbzip2 suitable for cgo. */
#include <bzlib.h>

int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen) {
    s->next_in = in;
    s->avail_in = *inlen;
    s->next_out = out;
    s->avail_out = *outlen;
    int r = BZ2_bzCompress(s, action);
    *inlen -= s->avail_in;
    *outlen -= s->avail_out;
    s->next_in = s->next_out = NULL;
    return r;
}
```

现在让我们转到 Go 语言部分，第一部分如下所示。其中`import "C"`的语句是比较特别的。其实并没有一个叫 C 的包，但是这行语句会让 Go 编译程序在编译之前先运行 cgo 工具。

```
// Package bzip provides a writer that uses bzip2 compression (bzip.org).
package bzip

/*
#cgo CFLAGS: -I/usr/include
#cgo LDFLAGS: -L/usr/lib -lbz2
#include <bzlib.h>
#include <stdlib.h>
bz_stream* bz2alloc() { return calloc(1, sizeof(bz_stream)); }
int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen);
void bz2free(bz_stream* s) { free(s); }
*/
import "C"

import (
    "io"
    "unsafe"
)

type writer struct {
    w      io.Writer // underlying output stream
    stream *C.bz_stream
    outbuf [64 * 1024]byte
}

// NewWriter returns a writer for bzip2-compressed streams.
func NewWriter(out io.Writer) io.WriteCloser {
    const blockSize = 9
    const verbosity = 0
    const workFactor = 30
    w := &writer{w: out, stream: C.bz2alloc()}
    C.BZ2_bzCompressInit(w.stream, blockSize, verbosity, workFactor)
    return w
}
```

在预处理过程中，cgo 工具爲生成一个临时包用于包含所有在 Go 语言中访问的 C 语言的函数或类型。例如 C.bz_stream 和 C.BZ2_bzCompressInit。cgo 工具通过以某种特殊的方式调用本地的 C 编译器来发现在 Go 源文件导入声明前的注释中包含的 C 头文件中的内容（译注 `import "C"` 语句前仅捱着的注释是对应 cgo ： 的特殊语法，对应必要的构建参数选项和 C 语言代码）。

在 cgo 注释中还可以包含#cgo 指令，用于给 C 语言工具链指定特殊的参数。例如 CFLAGS 和 LDFLAGS 分别
对应传给 C 语言编译器的编译参数和链接器参数，使它们可以特定目録找到 bzlib.h 头文件和 libbz2.a 库文 件。这个例子假设你已经在/usr 目録成功安装了 bzip2 库。如果 bzip2 库是安装在不同的位置，你需要更新 这些参数（译注：这里有一个从纯 C 代码生成的 cgo 绑定，不依赖 bzip2 静态库和操作系统的具体环境，具 体请访问 https://github.com/chai2010/bzip2 ）。

NewWriter 函数通过调用 C 语言的 BZ2_bzCompressInit 函数来初始化 stream 中的缓存。在 writer 结构中还包括了另一个 buffer，用于输出缓存。

下面是 Write 方法的实现，返回成功压缩数据的大小，主体是一个循环中调用 C 语言的 bz2compress 函数实 现的。从代码可以看到，Go 程序可以访问 C 语言的 bz_stream、char 和 uint 类型，还可以访问 bz2compress 等函数，甚至可以访问 C 语言中像 BZ_RUN 那样的宏定义，全部都是以 C.x 语法访问。其中 C.uint 类型和 Go 语言的 uint 类型并不相同，即使它们具有相同的大小也是不同的类型。

```
func (w *writer) Write(data []byte) (int, error) {
    if w.stream == nil {
        panic("closed")
    }
    var total int // uncompressed bytes written

    for len(data) > 0 {
        inlen, outlen := C.uint(len(data)), C.uint(cap(w.outbuf))
        C.bz2compress(w.stream, C.BZ_RUN,
            (*C.char)(unsafe.Pointer(&data[0])), &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        total += int(inlen)
        data = data[inlen:]
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return total, err
        }
    }
    return total, nil
}
```

在循环的每次迭代中，向 bz2compress 传入数据的地址和剩余部分的长度，还有输出缓存 w.outbuf 的地址
和容量。这两个长度信息通过它们的地址传入而不是值传入，因爲 bz2compress 函数可能会根据已经压缩 的数据和压缩后数据的大小来更新这两个值。每个块压缩后的数据被写入到底层的 io.Writer。

Close 方法和 Write 方法有着类似的结构，通过一个循环将剩余的压缩数据刷新到输出缓存。

```
// Close flushes the compressed data and closes the stream.
// It does not close the underlying io.Writer.
func (w *writer) Close() error {
    if w.stream == nil {
        panic("closed")
    }
    defer func() {
        C.BZ2_bzCompressEnd(w.stream)
        C.bz2free(w.stream)
        w.stream = nil
    }()
    for {
        inlen, outlen := C.uint(0), C.uint(cap(w.outbuf))
        r := C.bz2compress(w.stream, C.BZ_FINISH, nil, &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return err
        }
        if r == C.BZ_STREAM_END {
            return nil
        }
    }
}
```

压缩完成后，Close 方法用了 defer 函数确保函数退出前调用 C.BZ2_bzCompressEnd 和 C.bz2free 释放相关
的 C 语言运行时资源。此刻 w.stream 指针将不再有效，我们将它设置爲 nil 以保证安全，然后在每个方法中 增加了 nil 检测，以防止用户在关闭后依然错误使用相关方法。

上面的实现中，不仅仅写是非并发安全的，甚至并发调用 Close 和 Write 方法也可能导致程序的的崩溃。修 复这个问题是练习 13.3 的内容。

下面的 bzipper 程序，使用我们自己包实现的 bzip2 压缩命令。它的行爲和许多 Unix 系统的 bzip2 命令类 似。

```
gopl.io/ch13/bzipper

// Bzipper reads input, bzip2-compresses it, and writes it out.
package main

import (
    "io"
    "log"
    "os"
    "gopl.io/ch13/bzip"
)

func main() {
    w := bzip.NewWriter(os.Stdout)
    if _, err := io.Copy(w, os.Stdin); err != nil {
        log.Fatalf("bzipper: %v\n", err)
    }
    if err := w.Close(); err != nil {
        log.Fatalf("bzipper: close: %v\n", err)
    }
}
```

在上面的场景中，我们使用 bzipper 压缩了/usr/share/dict/words 系统自带的词典，从 938,848 字节压缩到 335,405 字节。大约是原始数据大小的三分之一。然后使用系统自带的 bunzip2 命令进行解压。压缩前后 文件的 SHA256 哈希码是相同了，这也帮助了我们的压缩工具是正确的。（如果你的系统没有 sha256sum 命令，那麼请先按照练习 4.2 实现一个类似的工具）

```
$ go build gopl.io/ch13/bzipper
$ wc -c < /usr/share/dict/words
938848
$ sha256sum < /usr/share/dict/words
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
$ ./bzipper < /usr/share/dict/words | wc -c
335405
$ ./bzipper < /usr/share/dict/words | bunzip2 | sha256sum
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
```

我们演示了如何将一个 C 语言库链接到 Go 语言程序。相反, 将 Go 编译爲静态库然后链接到 C 程序，或者将
Go 程序编译爲动态库然后在 C 程序中动态加载也都是可行的（译注：在 Go1.5 中，Windows 系统的 Go 语言 实现并不支持生成 C 语言动态库或静态库的特性。不过好消息是，目前已经有人在尝试解决这个问题，具体 请访问 Issue11058 ）。这里我们只展示的 cgo 很小的一些方面，更多的关于内存管理、指针、回调函 数、中断信号处理、字符串、errno 处理、终结器，以及 goroutines 和系统线程的关系等，有很多细节可 以讨论。特别是如何将 Go 语言的指针传入 C 函数的规则也是异常复杂的（译注：简单来说，要传入 C 函数 的 Go 指针指向的数据本身不能包含指针或其他引用类型；并且 C 函数在返回后不能继续持有 Go 指针；并且 在 C 函数返回之前，Go 指针是被锁定的，不能导致对应指针数据被移动或栈的调整），部分的原因在 13.2 节有讨论到，但是在 Go1.5 中还没有被明确（译注：Go1.6 将会明确 cgo 中的指针使用

规则）。如果要进一

步閱读，可以从 https://golang.org/cmd/cgo 开始。

练习 13.3： 使用 sync.Mutex 以保证 bzip2.writer 在多个 goroutines 中被并发调用是安全的。

练习 13.4： 因爲 C 库依赖的限制。 使用 os/exec 包启动/bin/bzip2 命令作爲一个子进程，提供一个纯 Go 的 bzip.NewWriter 的替代实现（译注：虽然是纯 Go 实现，但是运行时将依赖/bin/bzip2 命令，其他操作 系统可能无法运行）。

几点忠告

13.5. 几点忠告

我们在前一章结尾的时候，我们警告要谨慎使用 reflect 包。那些警告同样适用于本章的 unsafe 包。

高级语言使得程序员不用在关心眞正运行程序的指令细节，同时也不再需要关注许多如内存布局之类的实现细节。因爲高级语言这个絶缘的抽象层，我们可以编写安全健壮的，并且可以运行在不同操作系统上的具有高度可移植性的程序。

但是 unsafe 包，它让程序员可以透过这个絶缘的抽象层直接使用一些必要的功能，虽然可能是爲了获得更 好的性能。但是代价就是牺牲了可移植性和程序安全，因此使用 unsafe 包是一个危险的行爲。我们对何时 以及如何使用 unsafe 包的建议和我们在 11.5 节提到的 Knuth 对过早优化的建议类似。大多数 Go 程序员可能 永远不会需要直接使用 unsafe 包。当然，也永远都会有一些需要使用 unsafe 包实现会更简单的场景。如果 确实认爲使用 unsafe 包是最理想的方式，那麽应该尽可能将它限制在较小的范围，那样其它代码就忽略 unsafe 的影响。

现在，赶紧将最后两章抛入脑后吧。编写一些实实在在的应用是眞理。请远离 reflect 的 unsafe 包，除非你 确实需要它们。

最后，用 Go 快乐地编程。我们希望你能像我们一样喜欢 Go 语言。

附録

附録：作者/译者

英文作者

- Alan A. A. Donovan is a member of Google’s Go team in New York. He holds computer science degrees from Cambridge and MIT and has been programming in industry since 1996. Since 2005, he has worked at Google on infrastructure projects and was the co-designer of its proprietary build system, Blaze. He has built many libraries and tools for static analysis of Go programs, including oracle, `godoc -analysis`, eg, and gorename.
- Brian W. Kernighan is a professor in the Computer Science Department at Princeton University. He was a member of technical staff in the Computing Science Research Center at Bell Labs from 1969 until 2000, where he worked on languages and tools for Unix. He is the co-author of several books, including The C Programming Language, Second Edition (Prentice Hall, 1988), and The Practice of Programming (Addison-Wesley, 1999) .

中文译者

中文译者	章节
chai2010 <chaishushan@gmail.com>	前言/第 2~4 章/第 10~13 章
CrazySssst	第 5 章
foreversmart <njutree@gmail.com>	第 7 章
Xargin <cao1988228@163.com>	第 1 章/第 6 章/第 8~9 章

译文授权