# 《Go语言圣经》

《The Go Programming Language》中文版

**Go语言圣经（中文版）**

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

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

在线预览：

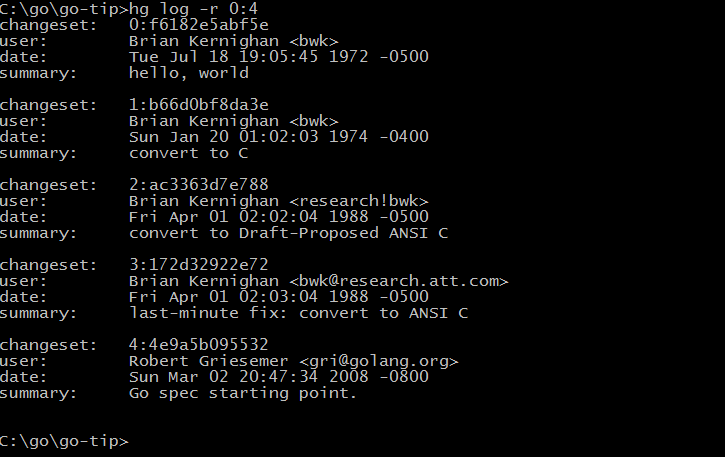
* <http://docs.ruanjiadeng.com/gopl-zh>
* <http://shifei.me/gopl-zh>
* <http://2goo.info/media/html/gopl-zh-gh-pages>
* <http://docs.plhwin.com/gopl-zh>

**版本信息**

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

**译者序**

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



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

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

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

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

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

最后，希望这本书能够帮助大家用Go语言快乐地编程。

2016年1月 于 武汉

## 前言

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

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

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

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

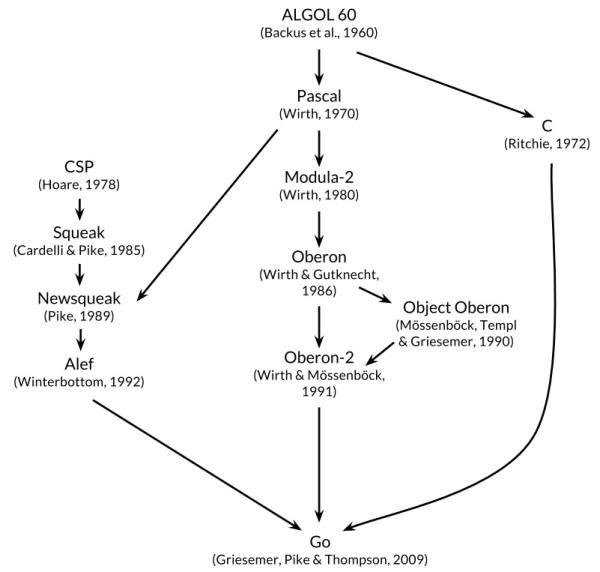
Go语言还是一个开源的项目，可以免费获取编译器、库、配套工具的源代码。Go语言都贡献者来自一个活跃的全球社区。Go语言可以运行在类[UNIX](http://doc.cat-v.org/unix/)系统——比如[Linux](http://www.linux.org/)、[FreeBSD](https://www.freebsd.org/)、[OpenBSD](http://www.openbsd.org/)、[Mac OSX](http://www.apple.com/cn/osx/)、[Plan9](http://plan9.bell-labs.com/plan9/)系统和[Microsoft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

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

## Go语言起源

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

下图展示了有哪些早期的编程语言对Go语言的设计产生了重要影响。



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

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

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

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

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

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

## Go语言项目

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

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

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

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

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

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

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

## 本书的组织

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

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

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

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

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

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

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

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

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

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

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

必要时，Go语言工具会创建目录。例如：

$ export GOPATH=$HOME/gobook # 选择工作目录

$ **go** get gopl.io/ch1/helloworld # 获取/编译/安装  
$ $GOPATH/bin/helloworld # 运行程序  
hello, 世界 # 这是中文

运行这些例子需要安装Go1.5以上的版本。

$ **go** version

**go** version go1.5 linux/amd64

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

## 更多的信息

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

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

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

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

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

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

## 致谢

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

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

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

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

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

2015年10月 于 纽约

## 入门

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

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

### Hello, World

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

gopl.io/ch1/helloworld

package main

import "fmt"

func main() {

fmt.Println("hello, 世界")

}

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

go run helloworld.go

毫无意外，这个命令行输出：

hello, 世界

Go语言原生支持Unicode，你可以处理全世界任何语言的文本。

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

go build helloworld.go

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

./helloworld

hello, 世界

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

gopl.io/ch1/helloworld

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

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

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

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

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

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

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

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

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

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

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

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

### 命令行参数

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

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

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

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

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

gopl.io/ch1/echo1

// Echo1 prints its command-line arguments

package main

import (

"fmt"

"os"

)

func main() {

var s, sep string

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

s += sep + os.Args[i]

sep = " "

}

fmt.Println(s)

}

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

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

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

sep + os.Args[i]

表示连接字符串sep和os.Args。程序中使用的语句：

s += sep + os.Args[i]

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

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

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

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

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

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

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

for initialization; condition; post {

// zero or more statements

}

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

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

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

// a traditional “white” loop

for condition {

// …

}

如果连condition也省略了，像下面这样：

// a traditional infinite loop

for {

// …

}

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

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

gopl.io/ch1/echo2

// Echo2 prints its command-line arguments

package main

import (

"fmt"

"os"

)

func main() {

s, sep := "", ""

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

s += sep + arg

sep = " "

}

fmt.Println(s)

}

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

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

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

s := “”

var s string

var s = “”

var s string = “”

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

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

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

gopl.io/ch1/echo3

// Echo3 prints its command-line arguments

package main

import (

"fmt"

"os"

"strings"

)

func main() {

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

}

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

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

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

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

练习1.2：修改echo程序，使其打印每个参数的索引和值，每个一行。

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

### 查找重复的行

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

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

gopl.io/ch1/dup1

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

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

package main

import (

"bufio"

"fmt"

"os"

)

func main() {

counts := make(map[string]int)

input := bufio.NewScanner(os.Stdin)

for input.Scan() {

counts[input.Text()]++

}

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

for line, n := range counts {

if n > 1 {

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

}

}

}

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

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

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

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

line := input.Text()

counts[line] = counts[line] + 1

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

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

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

程序使用短变量声明创建bufio.Scanner类型的变量input。

input := bufio.NewScanner(os.Stdin)

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

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

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

%d 十进制整数

%x, %o, %b 十六进制，八进制，二进制整数。

%f, %g, %e 浮点数：3.141593 3.141592653589793 3.141593e+∞

%t 布尔：true或false

%c 字符（rune）（Unicode码点）

%s 字符串

%q 带双引号的字符串“abc”或带单引号的字符‘c’

%v 变量的自然形式（natural format）

%T 变量的类型

%%字面上的百分号标志（无操作数）

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

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

gopl.io/ch1/dup2

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

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

package main

import (

"bufio"

"fmt"

"os"

)

func main() {

counts := make(map[string]int)

files := os.Args[1:]

if len(files) == 0 {

countLines(os.Stdin, counts)

} else {

for \_, arg := range files {

f, err := os.Open(arg)

if err != nil {

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

continue

}

countLines(f, counts)

f.Close()

}

}

for line, n := range counts {

if n > 1 {

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

}

}

}

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

input := bufio.NewScanner(f)

for input.Scan() {

counts[input.Text()]++

}

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

}

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

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

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

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

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

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

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

gopl.io/ch1/dup3

package main

import (

"fmt"

"io/ioutil"

"os"

"strings"

)

func main() {

counts := make(map[string]int)

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

data, err := ioutil.ReadFile(filename)

if err != nil {

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

continue

}

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

counts[line]++

}

}

for line, n := range counts {

if n > 1 {

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

}

}

}

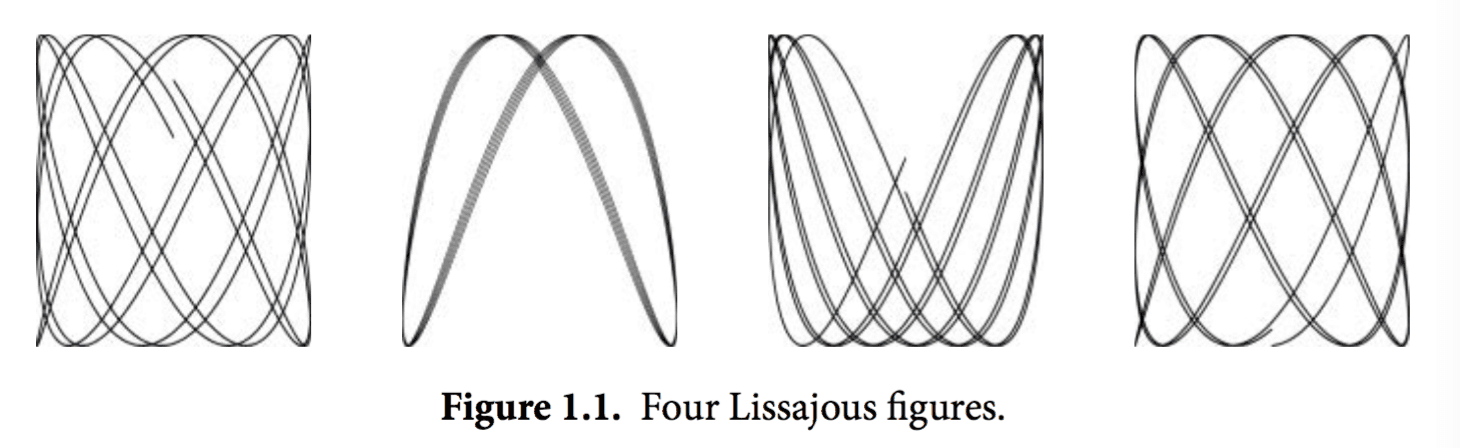
ReadFile函数返回一个字节切片（byte slice），必须把它转换为string，才能用strings.Split分割。我们会在3.5.4节详细讲解字符串和字节切片。

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

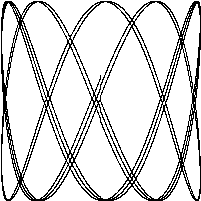
练习1.4：修改dup2，出现重复的行时打印文件名称。

### GIF动画

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



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



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

gopl.io/ch1/Lissajous

// Lissajous generates GIF animations of random Lissajous figures.

package main

import (

"image"

"image/color"

"image/gif"

"io"

"math"

"math/rand"

"os"

)

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

const (

whiteIndex = 0 // first color in palette

blackIndex = 1 // next color in palette

)

func main() {

lissajous(os.Stdout)

}

func lissajous(out io.Writer) {

const (

cycles = 5 //number of complete x oscillator revolutions

res = 0.001 // angular resolution

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

nframes = 64 // number of animation frames

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

)

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

anim := gif.GIF{LoopCount: nframes}

phase := 0.0 // phase difference

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

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

img := image.NewPaletted(rect, palette)

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

x := math.Sin(t)

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

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

}

phase += 0.1

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

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

}

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

}

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

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

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

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

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

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

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

go build gopl.io/ch1/Lissajous

./Lissajous > out.gif

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

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

### 获取URL

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

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

gopl.io/ch1/fetch

// Fetch prints the content found at a URL.

package main

import (

"fmt"

"io/ioutil"

"net/http"

"os"

)

func main() {

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

resp, err := http.Get(url)

if err != nil {

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

os.Exit(1)

}

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

resp.Body.Close()

if err != nil {

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

os.Exit(1)

}

fmt.Printf("%s", b)

}

}

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

go build gopl.io/ch1/fetch

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

<html>

<head>

<title>The Go Programing Language</tital>

…

HTTP请求如果失败了的话，会得到下面这样的结果：

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

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

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

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

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

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

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

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

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

### 并发获取多个URL

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

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

gopl.io/ch1/fetchall

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

package main

import (

"fmt"

"io"

"io/ioutil"

"net/http"

"os"

"time"

)

func main() {

start := time.Now()

ch := make(chan string)

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

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

}

for range os.Args[1:] {

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

}

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

}

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

start := time.Now()

resp, err := http.Get(url)

if err != nil {

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

return

}

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

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

if err != nil {

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

return

}

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

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

}

下面使用fetchall来请求几个地址：

go build gopl.io/ch1/fetchall

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

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

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

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

0.48s elapsed

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

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

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

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

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

### Web服务

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

gopl.io/ch1/server1

// Server1 is a minimal "echo" server.

package main

import (

"fmt"

"log"

"net/http"

)

func main() {

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

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

}

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

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

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

}

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

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

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

现在可以通过命令行来发送客户端请求了：

go build gopl.io/ch1/fetch

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

URL.Path = “/”

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

URL.Path = “/help”

还可以直接在浏览器里访问这个URL，然后得到返回结果，如图1.2：



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

gopl.io/ch1/server2

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

package main

import (

"fmt"

"log"

"net/http"

"sync"

)

var mu sync.Mutex

var count int

func main() {

http.HandleFunc("/", handler)

http.HandleFunc("/count", counter)

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

}

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

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

mu.Lock()

count++

mu.Unlock()

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

}

// counter echoes the number of calls so far.

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

mu.Lock()

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

mu.Unlock()

}

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

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

gopl.io/ch1/server3

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

package main

import (

"fmt"

"log"

"net/http"

)

func main() {

http.HandleFunc("/", handler)

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

}

// handler echoes the HTTP request.

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

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

for k, v := range r.Header {

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

}

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

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

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

log.Print(err)

}

for k, v := range r.Form {

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

}

}

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

GET /?q=query HTTP/1.1

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

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

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

RemoteAddr = "127.0.0.1:59911"

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

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

err := r.ParseForm()

if err != nil {

logs.Print(err)

}

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

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

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

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

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

lissajous(w)

}

http.HandlerFunc(“/”,handler)

或者另一种等价形式：

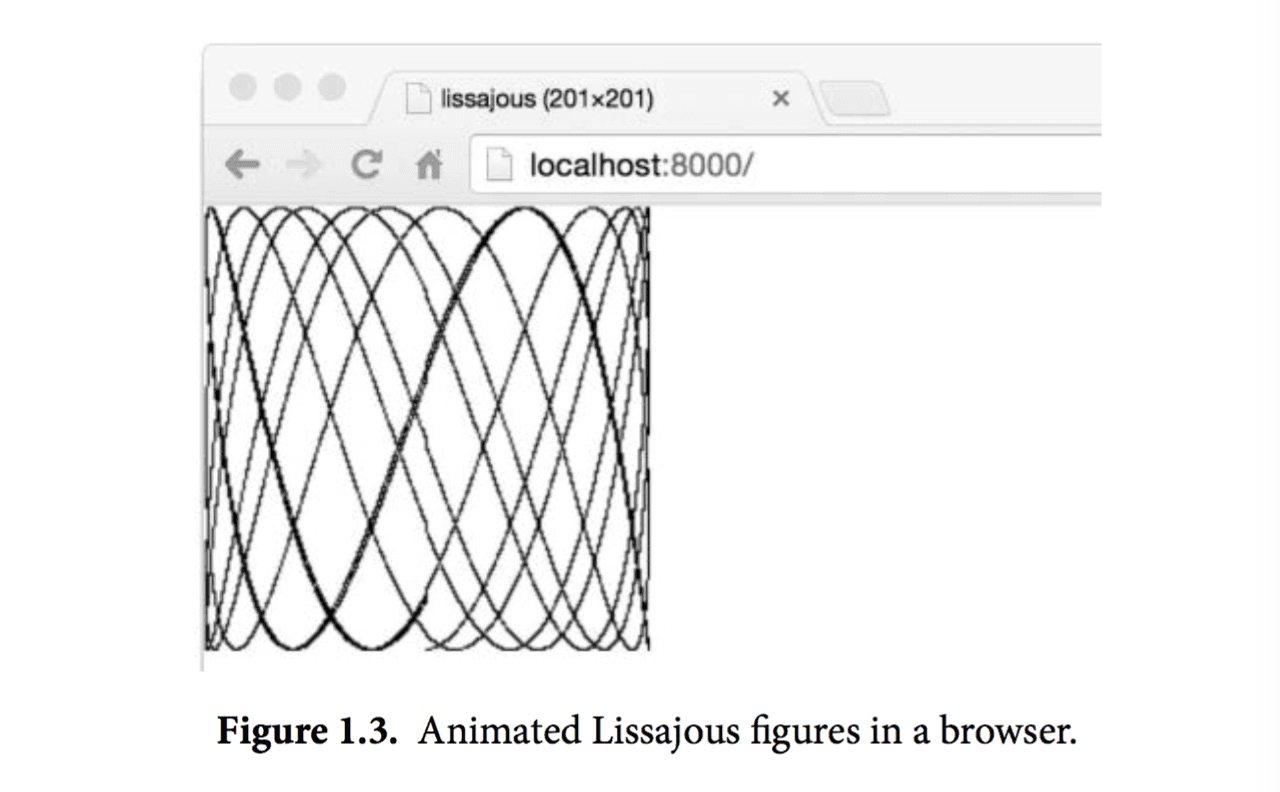
http.HandleFunc(“/”, func(w http.ResponseWriter, r \*http.Request) {

lissajous(w)

})

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

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



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

### 本章要点

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

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

switch coinflip() {

case “heads”:

heads++

case “tails”:

tails++

default:

fmt.Println(“landed on edge!”)

}

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

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

func Signum(x int) int {

switch {

case x > 0:

return +1

default:

return 0

case x < 0:

return -1

}

}

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

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

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

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

type Point struct {

X, Y int

}

var p Point

类型声明和命名类型会在第二章中介绍。

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

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

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

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

go doc http.ListenAndServe

package http // import “net/http”

func ListenAndServe(addr string, handler Handler) error

ListenAndServe listens on the TCP network address addr and then

calls Serve with handler to handle requests on incoming connections.

...

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

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

## 第二章 程序结构

## 第三章 基础数据类型

## 第四章 复合数据类型

## 第五章 函数

## 第六章 方法

## 第七章 接口

## 第八章 Goroutines和Channels

## 第九章 基于共享变量的开发

## 第十章 包和工具

## 第十一章 测试

## 第十二章 反射

## 第十三章 底层编程

## 第十四章 附录