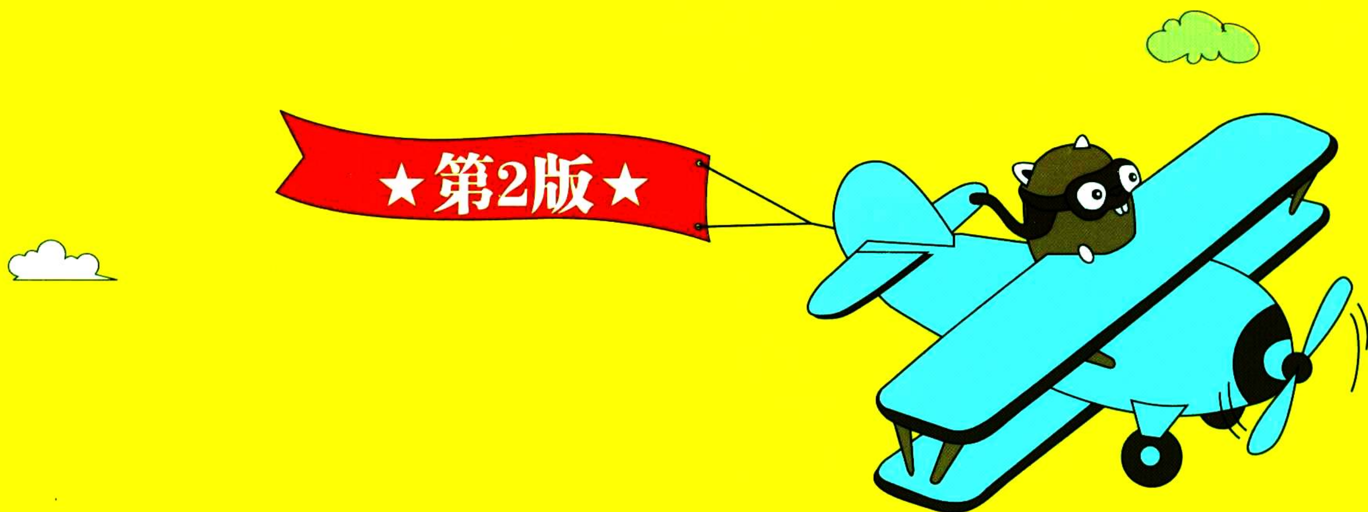
图灵原创



Go 并发编程实战

郝林◎著

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ或微信312082710

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我微信或QQ312082710

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，本人概不负责，我们仅仅只是帮助你寻找到你要的pdf而已。

TURING 图灵原创

Go并发编程实战

郝林◎著

人民邮电出版社
北京

图书在版编目（CIP）数据

Go并发编程实战 / 郝林著. -- 2版. -- 北京 : 人民邮电出版社, 2017.4
（图灵原创）
ISBN 978-7-115-45251-1

Ⅰ. ①G… Ⅱ. ①郝… Ⅲ. ①程序语言—程序设计
Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2017)第052901号

内容提要

本书首先介绍了Go语言的优秀特性、安装设置方法、工程结构、标准命令和工具、语法基础、数据类型以及流程控制方法，接着阐述了与多进程编程和多线程编程有关的知识，然后重点介绍了goroutine、channel以及Go提供的传统同步方法，最后通过一个完整实例——网络爬虫框架进一步阐述Go语言的哲学和理念，同时分享作者在多年编程生涯中的一些见解和感悟。

与上一版相比，本书不仅基于Go 1.8进行了全面更新，而且更深入地描绘了Go运行时系统的内部机理，并且大幅改进了示例代码。

本书适用于有一定计算机编程基础的从业者以及对Go语言编程感兴趣的爱好者，非常适合作为Go语言编程进阶教程。

◆ 著　　　　郝　林
　责任编辑　王军花
　责任印制　彭志环
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　三河市海波印务有限公司印刷
◆ 开本：800×1000　1/16
　印张：23.75
　字数：472千字　　　　2017年4月第2版
　印数：8 501 – 12 500册　　2017年4月河北第1次印刷

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第8052号

序

Go 是年轻而有活力的语言。

它最初于 2007 年由 Robert Griesemer、Rob Pike 和 Ken Thompson 在 Google 开始开发，2009 年正式发布。作者们希望 Go 能使复杂、高效系统的编写工作变得简单、可靠；同时，也希望 Go 能成为一个相对通用的编程环境，适应诸如桌面应用、移动应用、数值计算等。

Go 的设计理念充分体现了这些设计目标。它是极简化语言的代表，推崇少即是多。为了避免复杂、不可读的代码，Go 限制了语言功能与语法特性。Go 的可读性在众多编程语言中是独树一帜的。另外，为了减轻使用者编写高性能应用的负担，它也引入了 runtime，提供了诸如协程、垃圾回收等功能。runtime 虽然使语言本身的实现更复杂，但它让使用者获得了更简单易用的编程环境。

Go 语言是极易掌握的语言，它与 C 语言十分相近。熟悉 C、C++等语言的编程人员可以在短时间内掌握 Go 语言来编写简单、高效的应用。它只有 20 多个语言关键词。作为初学者，它也是相对容易入门的语言。

国内的 Go 语言社区十分活跃，这得益于致力推广 Go 的技术精英们。我认识本书作者郝林，也是源于他组织的 Go 语言北京交流会。利用业余时间，他广泛推广普及 Go 语言，组织、邀请技术专家参与交流会。他坚持不懈两年有余，取得了显著的成绩。郝林对 Go 社区建设的执着与热情令人敬佩。我相信，本书也是凝集了他对技术推广的一腔热情，希望让 Go 语言的初学者、工程师们能更快捷、深入地理解 Go 语言，以促进整个技术领域的发展。

Go 语言方面的图书对培养高素质的业余爱好者、从业人员起到了至关重要的作用。本书在各种 Go 语言图书中也是特点鲜明。本书首先介绍了 Go 语言的基础知识，对初学者有所铺垫。书中大量篇幅覆盖了 Go 语言的并发特性，详细讲解了其中的哲学、原理与实现。我相信很多像我这样，每天都沉浸在 Go 语言的从业人员，也并不完全知道 Go 内部实现的奥妙。每天花上一些时间来读此书，即便对有经验的 Go 从业人员来说，也会有所帮助。

在翻读本书时，我也深深体会到了作者写作的用心之处，每章不光有概念的讲解，还有

实现实例和经典案例。这些细心之处，让这样一本严肃的技术书读起来并不枯燥、乏味。书末更有独立的一章来介绍用 Go 语言实现的一个爬虫系统。相信很多读者都会迫不及待地跟着作者一起动起手来，实践书中的知识与概念。

最后，作为 Go 社区和开源社区的一员，我希望读者们能够在享受 Go 开发带来的乐趣与收获的同时，能够回馈、融入社区。你们的每一个建议与意见，每一个问题反馈与代码补丁，都会促进和推动开源社区，以及整个计算机产业的发展。我想这也是郝林如此用心编写此书的一个初衷。

李响，CoreOS 分布式系统组主管（Head of distributed systems）

2017 年 3 月 5 日，于美国加利福尼亚州

前　　言

很高兴你能选择这本已经过大量改进的书，希望本书能够让你成为真正的 Go 粉。很多 Go 语言爱好者都喜欢称自己是 Gopher，这是一个来自官方的传统，希望你也能这样称呼自己。

Go 编程语言（或称 Golang，以下简称 Go 语言）是云计算时代的 C 语言。7 年过去了，它渐渐向世人证明此言不虚。如果你关注 TIOBE 的编程语言排行榜就会发现，Go 语言从前些年的第 50 多位，经过多次上窜已经跃居第 13 位，跻身绝对主流的编程语言行列！同时，它还被评为 2016 年的年度语言！经过了数年的不断改进，Go 语言在开发效率和程序运行效率方面又上了数个台阶。

下面我简单列举一下它在最近两年比较显而易见的变化。

- **本身的自举**。也就是说，Go 语言几乎完全用 Go 语言程序重写了自己，仅留有一些汇编程序。Go 语言的自举非常彻底，包括了最核心的编译器、链接器、运行时系统等。现在任何学习 Go 语言的人都可以直接读它的源代码了。此变化也使 Go 程序的跨平台编译变得轻而易举。
- **运行时系统的改进**。这主要体现在更高效的调度器、内存管理以及垃圾回收方面。调度器已能让 goroutine 更及时地获得运行时机。运行时系统对内存的利用和控制也更加精细了。因垃圾回收而产生的调度停顿时间已经小于原来的 1‰。另外，最大 P 数量的默认值由原先的 1 变为与当前计算机的 CPU 核心数相同。
- **标准工具的增强**。在 Go 1.4 加入 `go generate` 之后，一个惊艳的程序调试工具 `go tool trace` 也被添加进来了。另外，`go tool compile`、`go tool asm` 和 `go tool link` 等工具也已到位；一旦你安装好 Go，就可以直接使用它们。同时，几乎所有的标准工具和命令都得到了不同程度的改进。
- **访问控制的细化**。这种细化始于 Go 1.4，正式支持始于 Go 1.5，至今已被广泛应用。经过细化，对于 Go 程序中的程序实体，除了原先的两种访问控制级别（公开和包级私有）之外，又多了一种——模块级私有。这是通过把名称首字母大写的程序实体放入 internal 代码包实现的。

- **vendor 机制的支持**。自 Go 1.5 之后，一个特殊的目录——vendor——被逐渐启用。它用于存放其父目录中的代码包所依赖的那些代码包。在程序被编译时，编译器会优先引用存于其中的代码包。这为固化程序的依赖代码迈出了很重要的一步。在 Go 1.7 中，vendor 目录以及背后的机制被正式支持。

当然，上述变化并不是全部。它的标准库也经历了超多的功能和性能改进。如果你是在本书的第 1 版面市时开始学习 Go 语言的，那么一定能感受到这些变化带来的巨大红利。

从本书第 1 版出版至今，Go 语言的版本号已经从 1.4 升至 1.8，本书第 2 版就是基于 Go 1.8 写成的。

本书第 2 版根据 Go 语言本身的变化以及第 1 版读者的大量反馈做了很多改进，也重写了非常多的内容。你的第一感觉肯定是书变薄、变轻了！没错，最直观的改进就是书中几乎每个章节都更为精炼。在讲 Go 编程基础的时候，我只说明重点，并尽量用代码和表格代替文字，同时让它们变得易于速查。因此，本书在这方面的篇幅由 5 章缩减为了 2 章。然而，讲并发编程理念、方法和实战的部分却得到了适当的扩充。你可能已经发现，后面几章的内容非常多，这是因为我更加细致地讲解了那些核心知识。

同时，本书的所有示例程序都重新编排，变得更加有条理，更加容易查找。当然，我对示例程序本身也做了相当大的改造。首先是充分使用 Go 语言的新特性。比如，有的代码包拥有了自己的 internal 包。又比如，示例项目本身也用到了 vendor 目录。其次，更多的优质代码包被引入进来并充分利用，比如 io/ioutil、context 等。另外，还对一些关键示例进行了脱胎换骨的改写。比如，完全重写了 ConcurrentMap，它的性能比原先的版本高出数倍。又比如，我对网络爬虫框架做了大幅更新，既包括较为底层的数据结构，也包括调度器以及一些模块的接口和实现。这使得它更易用、扩展性更好，同时代码中体现的技巧和理念也更多、更突出。因此，本书最后一章也几乎完全重写了。

本书结构

本书共分为 6 章。

第 1 章，快速介绍了 Go 语言的优秀特性、安装设置方法、工程结构以及标准命令和工具。

第 2 章，讲述了 Go 语言的语法基础、数据类型以及流程控制方法。

第 3 章，主要阐述了与多进程编程和多线程编程有关的知识。这些知识作为理解 Go 语言并发编程模型的先导内容。看过以后，你就可以对并发编程有一个比较清晰的理解。之后，本章还简要剖析了多核时代（基于多 CPU 核心的计算时代）的并发编程需求。

第 4 章，在深入展示和说明 Go 语言的并发编程模型之后，本章讲解 Go 特有的编程要素——goroutine（也可称为 Go 例程）的用法以及背后的运作机理。此外，本章还会对 Go 并发编程中另一个不可或缺的部分——channel 进行重点介绍，包括概念、使用规则以及应用技巧。

第 5 章，会对 Go 语言提供的传统同步方法进行介绍。这包括互斥锁、条件变量、原子操作、WaitGroup、临时对象池等，这些同步方法大多是标准库代码包 sync 中的一员。虽然 Go 语言官方并不建议优先使用这些方式来保障程序的并发安全性，但不容忽视的是，它们在一些应用场景中确实简单有效。

第 6 章，包括一个基本囊括了本书所有概念和知识的完整示例——网络爬虫框架。我会带你逐步编写这个示例，并进一步阐述 Go 语言的哲学和理念，同时分享我在多年编程生涯中的一些见解和感悟。你可以通过这个示例来巩固前面学到的 Go 语言知识，并加深对 Go 并发编程的理解。

附录 A，简单介绍目前在国内外比较活跃的一部分 Go 语言开源项目和 Go 语言社区，这会使你学习 Go 语言的道路变得更加顺畅，也有利于你找到志同道合的朋友。

目标读者

原则上来讲，任何对计算机编程和 Go 语言感兴趣的人都可以阅读本书。但是，当你学习一门编程语言的时候，往往还是需要有一些基础的。比如，怎样使用文本编辑器、怎样在相应的操作系统中安装软件，等等。而要想成为一名高级的 Go 软件工程师，你需要了解的知识可能比表面看上去的多很多。这就像摘苹果一样，如果要摘到苹果，就需要徒手爬上果树，或者找到足够高的梯子；如果想摘到果树顶端最甜的那个苹果，就需要花费更多的时间和精力，爬过更多的枝叶。希望本书能成为帮助你摘苹果的梯子。但是，在想有所收获之前，请先潜心学习和积累。

当然，你在阅读本书的过程中边看边学也完全没问题，甚至可以看完本书再去学习相关知识。采用哪种学习方式，这完全取决于你自己。

关于示例代码

我会把本书涉及的示例代码[①]都放到一个名为 example.v2 的项目中，你可以访问

① 本书源代码也可去图灵社区（iTuring.cn）本书主页免费注册下载。

https://github.com/gopcp/example.v2 查看或下载。你存放该项目目录的绝对路径应该包含在环境变量 GOPATH 中。如果你不了解 Git（一款代码版本控制工具），请在网上搜索“git”并详细了解。

关于勘误

由于作者水平和时间有限，书中难免会有一些纰漏和错误，欢迎读者及时指正，本书后续印次或版本中将加以改正。非常希望和大家一起学习和讨论 Go 语言，并共同推动 Go 语言在中国的发展。你可以通过电子邮件（hypermind@outlook.com）联系我，也可以到图灵社区（iTuring.cn）本书主页上发表评论。

致谢

写书是一项需要大量精力和毅力的工作，尤其是编写技术图书，更需要作者对相关知识进行深入的梳理和系统的整合，还需要制作各种图表，编写各种示例，工作量确实不小。但是，这个写作过程也很有趣，我通过写作也收获了很多。当然，很多收获来自他人的传授。其中，图灵公司的编辑王军花和傅志红老师都给予了我很大的帮助，尤其是在写作技巧和图书结构方面。当然，还有目前中国业内公认的 Go 语言专家们，我在编写本书的时候经常向他们讨教。在此，谨对帮助过我的所有人表示由衷的感谢。同时也要感谢我的家人，没有他们的支持和理解，我不可能在有限的业余时间里完成本书。

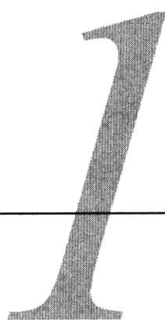
目　　录

第 1 章　初识 Go 语言 ··· 1
1.1　语言特性 ··· 1
1.2　安装和设置 ··· 2
1.3　工程结构 ··· 3
1.3.1　工作区 ··· 3
1.3.2　GOPATH ··· 4
1.3.3　源码文件 ··· 5
1.3.4　代码包 ··· 8
1.4　标准命令简述 ··· 11
1.5　问候程序 ··· 13
1.6　小结 ··· 14
第 2 章　语法概览 ··· 15
2.1　基本构成要素 ··· 15
2.1.1　标识符 ··· 15
2.1.2　关键字 ··· 16
2.1.3　字面量 ··· 17
2.1.4　操作符 ··· 17
2.1.5　表达式 ··· 19
2.2　基本类型 ··· 20
2.3　高级类型 ··· 22
2.3.1　数组 ··· 23
2.3.2　切片 ··· 23
2.3.3　字典 ··· 24
2.3.4　函数和方法 ··· 25
2.3.5　接口 ··· 28
2.3.6　结构体 ··· 29
2.4　流程控制 ··· 30
2.4.1　代码块和作用域 ··· 30
2.4.2　if 语句 ··· 32
2.4.3　switch 语句 ··· 32
2.4.4　for 语句 ··· 34
2.4.5　defer 语句 ··· 36
2.4.6　panic 和 recover ··· 38
2.5　聊天机器人 ··· 40
2.6　小结 ··· 44
第 3 章　并发编程综述 ··· 45
3.1　并发编程基础 ··· 45
3.1.1　串行程序与并发程序 ··· 46
3.1.2　并发程序与并行程序 ··· 46
3.1.3　并发程序与并发系统 ··· 47
3.1.4　并发程序的不确定性 ··· 47
3.1.5　并发程序内部的交互 ··· 47
3.2　多进程编程 ··· 48
3.2.1　进程 ··· 48
3.2.2　关于同步 ··· 55
3.2.3　管道 ··· 60
3.2.4　信号 ··· 65
3.2.5　socket ··· 74
3.3　多线程编程 ··· 97
3.3.1　线程 ··· 98
3.3.2　线程的同步 ··· 107
3.4　多线程与多进程 ··· 125
3.5　多核时代的并发编程 ··· 126
3.6　小结 ··· 130
第 4 章　Go 的并发机制 ··· 131
4.1　原理探究 ··· 131
4.1.1　线程实现模型 ··· 132
4.1.2　调度器 ··· 142
4.1.3　更多细节 ··· 158
4.2　goroutine ··· 160

4.2.1 go 语句与 goroutine ……160
4.2.2 主 goroutine 的运作 ……166
4.2.3 runtime 包与 goroutine……166
4.3 channel……169
4.3.1 channel 的基本概念 ……169
4.3.2 单向 channel ……180
4.3.3 for 语句与 channel……184
4.3.4 select 语句 ……185
4.3.5 非缓冲的 channel……190
4.3.6 time 包与 channel……192
4.4 实战演练：载荷发生器……198
4.4.1 参数和结果 ……199
4.4.2 基本结构……201
4.4.3 初始化……206
4.4.4 启动和停止 ……212
4.4.5 调用器和功能测试……221
4.5 小结 ……231
第 5 章 同步……232
5.1 锁的使用 ……232
5.1.1 互斥锁 ……232
5.1.2 读写锁 ……236
5.1.3 锁的完整示例 ……238
5.2 条件变量 ……244
5.3 原子操作 ……247
5.3.1 增或减……247
5.3.2 比较并交换……249
5.3.3 载入……250
5.3.4 存储……251
5.3.5 交换……251
5.3.6 原子值……252
5.3.7 应用于实际……256
5.4 只会执行一次……257
5.5 WaitGroup……258
5.6 临时对象池……262
5.7 实战演练——Concurrent Map ……265
5.8 小结……280
第 6 章 网络爬虫框架设计和实现……281
6.1 网络爬虫与框架……281
6.2 功能需求和分析……283
6.3 总体设计 ……284
6.4 详细设计 ……286
6.4.1 基本数据结构 ……286
6.4.2 接口的设计……293
6.5 工具的实现 ……309
6.5.1 缓冲器……309
6.5.2 缓冲池……311
6.5.3 多重读取器 ……317
6.6 组件的实现 ……318
6.6.1 内部基础接口 ……319
6.6.2 组件注册器……321
6.6.3 下载器……323
6.6.4 分析器……325
6.6.5 条目处理管道 ……328
6.7 调度器的实现……329
6.7.1 基本结构……329
6.7.2 初始化……331
6.7.3 启动 ……333
6.7.4 停止 ……343
6.7.5 其他方法……344
6.7.6 总结 ……345
6.8 一个简单的图片爬虫 ……346
6.8.1 概述 ……346
6.8.2 命令参数……346
6.8.3 初始化调度器 ……348
6.8.4 监控调度器……354
6.8.5 启动调度器……364
6.9 扩展与思路 ……365
6.10 本章小结 ……368
附录 A Go 语言的学习资源……369

第 1 章

初识 Go 语言



在讲解怎样用 Go 语言之前，我们先介绍 Go 语言的特性、基础概念和标准命令。

1.1 语言特性

我们可以用几个关键词或短语来概括 Go 语言的主要特性。

- **开放源代码**。这显示了Go作者开放的态度以及营造语言生态的决心。顺便说一句，Go本身就是用Go语言编写的。
- **静态类型和编译型**。在Go中，每个变量或常量都必须在声明时指定类型，且不可改变。另外，程序必须通过编译生成归档文件或可执行文件，而后才能被使用或执行。不过，其语法非常简洁，就像一些解释型脚本语言那样，易学易用。
- **跨平台**。这主要是指跨计算架构和操作系统。目前，它已经支持绝大部分主流的计算架构和操作系统，并且这个范围还在不断扩大。只要下载与之对应的Go语言安装包，并且经过简单的安装和设置，就可以使Go就绪了。除此之外，在编写Go语言程序的过程中，我们几乎感觉不到不同平台的差异。
- **自动垃圾回收**。程序在运行过程中的垃圾回收工作一般由Go运行时系统全权负责。不过，Go也允许我们对此项工作进行干预。
- **原生的并发编程**。拥有自己的并发编程模型，其主要组成部分有goroutine（也可称为Go例程）和channel（也可称为通道）。另外，还拥有一个特殊的关键字go。
- **完善的构建工具**。它自带了很多强大的命令和工具，通过它们，可以很轻松地完成Go程序的获取、编译、测试、安装、运行、分析等一系列工作。
- **多编程范式**。Go支持函数式编程。函数类型为第一等类型，可以方便地传递和赋值。此外，它还支持面向对象编程，有接口类型与实现类型的概念，但用嵌入替代了继承。
- **代码风格强制统一**。Go安装包中有自己的代码格式化工具，可以用来统一程序的

编码风格。
- **高效的编程和运行**。Go简洁、直接的语法使我们可以快速编写程序。加之它强大的运行时系统，程序可以充分利用计算环境飞快运行。
- **丰富的标准库**。Go是通用的编程语言，其标准库中有很多开箱即用的API。尤其是在编写诸如系统级程序、Web程序和分布式程序时，我们几乎无需依赖第三方库。

看到Go如此多的先进特性后，你是否已经心动了？反正我已经为此折服并感到非常激动。这正是我迫切需要的语言！这也正是我迫不及待地深入研究它，并通过一本书把我知道的所有细节分享给大家的原因。

1.2 安装和设置

安装Go相当简单，只要你的操作系统被Go语言支持即可。Go语言官方网站放出的每一个版本都会有主流平台的二进制安装包以及源码包，你可以自行挑选对应的文件下载。Go的下载地址为：https://golang.org/dl。

本节，我们以64位的Linux操作系统为计算环境，简要描述安装和设置过程。

假设已经从Go的下载地址中挑选了go1.8.linux-amd64.tar.gz文件并下载。注意，这个文件的名称中有两个关键词，一个是linux，一个是amd64。前者表示该文件对应的操作系统种类，其他的同类词有darwin、freebsd、windows等。后者表示该文件对应的计算架构种类，其中amd64表示64位的计算架构；另一个同类词是386，表示32位的计算架构。实际上，这里的计算架构和操作系统可以统称为Go的计算平台或计算环境。

解压该文件时会得到一个名为go的文件夹，其中包含所有的Go语言相关文件。该文件夹下还有很多文件夹和文件，下面简要说明其中主要文件夹的功用。

- **api文件夹**。用于存放依照Go版本顺序的API增量列表文件。这里所说的API包含公开的变量、常量、函数等。这些API增量列表文件用于Go语言API检查。
- **bin文件夹**。用于存放主要的标准命令文件，包括go、godoc和gofmt。
- **blog文件夹**。用于存放官方博客中的所有文章，这些文章都是Markdown格式的。
- **doc文件夹**。用于存放标准库的HTML格式的程序文档。我们可以通过godoc命令启动一个Web程序展现这些文档。
- **lib文件夹**。用于存放一些特殊的库文件。
- **misc文件夹**。用于存放一些辅助类的说明和工具。

- **pkg文件夹**。用于存放安装Go标准库后的所有归档文件。注意，你会发现其中有名称为linux_amd64的文件夹，我们称为平台相关目录。可以看到，这类文件夹的名称由对应的操作系统和计算架构的名称组合而成。通过`go install`命令，Go程序（这里是标准库中的程序）会被编译成平台相关的归档文件并存放到其中。另外，pkg/tool/linux_amd64文件夹存放了使用Go制作软件时用到的很多强大命令和工具。
- **src文件夹**。用于存放Go自身、Go标准工具以及标准库的所有源码文件。深入探究Go，就靠它了。
- **test文件夹**。存放用来测试和验证Go本身的所有相关文件。

现在，你已经大致了解了 go 文件夹中的目录结构及其用途。下面我们把这个 go 文件夹放到一个合适的目录中。对于 Linux 操作系统，Go 官方推荐的目录是/usr/local。

在这之后，我们需要设置一个环境变量，即：`GOROOT`。`GOROOT` 的值应该是 Go 的根目录。这里是/usr/local/go。另外，环境变量中 `PATH` 中也应该增加一项，即`$GOROOT/bin`。这样就可以在任意目录下使用那几个 Go 命令了。

至此，Go 已经安装好了。下面重点了解一下 Go 中的一些基础概念。

1.3 工程结构

Go 是一门推崇软件工程理念的编程语言，它为开发周期的每个环节都提供了完备的工具和支持。Go 语言高度强调代码和项目的规范和统一，这集中体现在工程结构或者说代码体制的细节之处。Go 也是一门开放的语言，它本身就是开源软件。更重要的是，它让开发人员很容易通过 `go get` 命令从各种公共代码库（比如著名的代码托管网站 GitHub）中下载开源代码并使用。这除了得益于 Go 自带命令的强大之外，还应该归功于 Go 工程结构的严谨和完善。本节中，我们详述 Go 的工程结构。

1.3.1 工作区

一般情况下，Go 源码文件必须放在工作区中。但是对于命令源码文件来说，这不是必需的。工作区其实就是一个对应于特定工程的目录，它应包含 3 个子目录：src 目录、pkg 目录和 bin 目录，下面逐一说明。

- **src目录**。用于以代码包的形式组织并保存Go源码文件，这里的代码包与src下的子目录一一对应。例如，若一个源码文件被声明属于代码包log，那么它就应当保

存在src/log目录中。当然，你也可以把Go源码文件直接放在src目录下，但这样的Go源码文件就只能被声明属于main代码包了。除非用于临时测试或演示，一般还是建议把Go源码文件放入特定的代码包中。

- **pkg目录**。用于存放通过`go install`命令安装后的代码包的归档文件。前提是代码包中必须包含Go库源码文件。归档文件是指那些名称以“.a”结尾的文件。该目录与GOROOT目录下的pkg目录功能类似。区别在于，工作区中的pkg目录专门用来存放用户代码的归档文件。编译和安装用户代码的过程一般会以代码包为单位进行。比如`log`包被编译安装后，将生成一个名为log.a的归档文件，并存放在当前工作区的pkg目录下的平台相关目录中。
- **bin目录**。与pkg目录类似，在通过`go install`命令完成安装后，保存由Go命令源码文件生成的可执行文件。在类Unix操作系统下，这个可执行文件一般来说名称与源码文件的主文件名相同。而在Windows操作系统下，这个可执行文件的名称则是源码文件主文件名加.exe后缀。

注意　这里有必要明确一下Go语言的命令源码文件和库源码文件的区别。所谓命令源码文件，就是声明属于main代码包并且包含无参数声明和结果声明的`main`函数的源码文件。这类源码文件是程序的入口，它们可以独立运行（使用`go run`命令），也可以通过`go build`或`go install`命令得到相应的可执行文件。所谓库源码文件，则是指存在于某个代码包中的普通源码文件。

1.3.2　GOPATH

我们需要将工作区的目录路径添加到环境变量GOPATH中。否则，即使处于同一工作区（事实上，未被加入GOPATH中的目录不应该称为工作区），代码之间也无法通过绝对代码包路径调用。在实际开发环境中，工作区可以只有一个，也可以有多个，这些工作区的目录路径都需要添加到GOPATH中。与GOROOT一样，我们应该确保GOPATH一直有效。

注意

- GOPATH中不要包含Go语言的根目录，以便将Go语言本身的工作区同用户工作区严格分开。
- 通过Go工具中的代码获取命令`go get`，可将指定项目的源码下载到我们在GOPATH中设定的第一个工作区中，并在其中完成编译和安装。

本书约定，示例项目会被存放在$HOME/golang/example.v2 目录下，并且在该目录

的 src/gopcp.v2 子目录中存放示例代码。而示例代码依赖的第三方代码包则会被放入$HOME/ golang/example.v2/src/gopcp.v2/vendor 目录。vendor 目录是一个特殊的目录，一般用于存放其他代码包目录中的源码文件依赖的那些代码包。vendor 目录及其机制并不复杂，它们的说明详见 https://docs.google.com/document/d/1Bz5-UB7g2uPBdOx- rw5t9MxJ-wkfpx90cqG9AFL0JAYo 和 https://golang.org/doc/go1.6#go_command。它在 Go 1.5 版本时被试验性地添加进 Go 命令，并在 Go 1.6 版本中变成默认选项。而到了 Go 1.7 版本，它已经成为 Go 命令的正式成员。

注意，你需要把 example.v2 所在的目录当成一个工作区目录，即需要把这个目录添加到 GOPATH 中。

1.3.3 源码文件

我为本书涉及的示例程序专门建立了项目 example.v2，可以访问 https://github.com/gopcp/example.v2 下载。不过，无论从哪里得到示例项目，你都需要将 example.v2 文件夹放到$HOME/golang 目录下，并设置好环境变量 GOPATH。

example.v2 项目包含了 Go 源码文件的所有 3 个种类，即命令源码文件、库源码文件和测试源码文件，下面详细说明这 3 类源码文件。

1. 命令源码文件

如果一个源码文件被声明属于 main 代码包，且该文件代码中包含无参数声明和结果声明的 main 函数，则它就是命令源码文件。命令源码文件可通过 go run 命令直接启动运行。

同一个代码包中的所有源码文件，其所属代码包的名称必须一致。如果命令源码文件和库源码文件处于同一个代码包中，那么在该包中就无法正确执行 go build 和 go install 命令。换句话说，这些源码文件将无法通过常规方法编译和安装。因此，命令源码文件通常会单独放在一个代码包中。这是合理且必要的，因为通常一个程序模块或软件的启动入口只有一个。

同一个代码包中可以有多个命令源码文件，可通过 go run 命令分别运行，但这会使 go build 和 go install 命令无法编译和安装该代码包。所以，我们也不应该把多个命令源码文件放在同一个代码包中。

当代码包中有且仅有一个命令源码文件时，在文件所在目录中执行 go build 命令，

即可在该目录下生成一个与目录同名的可执行文件；而若使用 go install 命令，则可在当前工作区的 bin 目录下生成相应的可执行文件。例如，代码包 gopcp.v2/helper/ds 中只有一个源码文件 showds.go，且它是命令源码文件，则相关操作和结果如下：

```
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/ds$ ls
showds.go
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/ds$ go build
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/ds$ ls
ds  showds.go
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/ds$ go install
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/ds$ ls ../../../../bin
ds
```

需要特别注意，只有当环境变量 GOPATH 中只包含一个工作区的目录路径时，go install 命令才会把命令源码文件安装到当前工作区的 bin 目录下；否则，像这样执行 go install 命令就会失败。此时必须设置环境变量 GOBIN，该环境变量的值是一个目录的路径，该目录用于存放所有因安装 Go 命令源码文件而生成的可执行文件。

2. 库源码文件

通常，库源码文件声明的包名会与它直接所属的代码包（目录）名一致，且库源码文件中不包含无参数声明和无结果声明的 main 函数。下面来安装（其中包含编译）gopcp.v2/helper/log 包，其中含有若干库源码文件：

```
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/log$ ls
base            logger.go       logger_test.go logrus
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/log$ go install
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/log$ ls ../../../../pkg
linux_amd64
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/log$ ls ../../../../pkg/linux_amd64/gopcp.v2/helper
log   log.a
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/log$ ls ../../../../pkg/linux_amd64/gopcp.v2/helper
/log
base.a   field.a  logrus.a
```

这里，我们通过在 gopcp.v2/helper/log 代码包的目录下执行 go install 命令，成功安装了该包并生成了若干归档文件。这些归档文件的存放目录由以下规则产生。

- 安装库源码文件时所生成的归档文件会被存放到当前工作区的pkg目录中。example.v2项目的gopcp.v2/helper/log包所属工作区的根目录是~/golang/example.v2。因此，上面所说的pkg目录即~/golang/example.v2/pkg。
- 根据被编译时的目标计算环境，归档文件会被放在该pkg目录下的平台相关目录中。例如，我是在64位的Linux计算环境下安装的，对应的平台相关目录就是linux_amd64，那么归档文件一定会被存放到~/golang/example.v2/pkg/linux_amd64

目录中的某个地方。

- 存放归档文件的目录的相对路径与被安装的代码包的上一级代码包的相对路径一致。第一个相对路径是相对于工作区的pkg目录下的平台相关目录而言的，而第二个相对路径是相对于工作区的src目录而言的。如果被安装的代码包没有上一级代码包（也就是说，它的父目录就是工作区的src目录），那么它的归档文件就会被直接存放到当前工作区的pkg目录的平台相关目录下。例如，gopcp.v2/helper/log包的归档文件log.a一定会被存放到~/golang/example.v2/pkg/linux_amd64/gopcp.v2/helper这个目录下。而它的子代码包gopcp.v2/helper/log/base的归档文件base.a，则一定会被存放到~/golang/example.v2/pkg/ linux_amd64/gopcp.v2/helper/log目录下。

3. 测试源码文件

测试源码文件是一种特殊的库文件，可以通过执行 go test 命令运行当前代码包下的所有测试源码文件。成为测试源码文件的充分条件有两个，如下。

- 文件名需要以“_test.go”结尾。
- 文件中需要至少包含一个名称以Test开头或Benchmark开头，且拥有一个类型为*testing.T或*testing.B的参数的函数。testing.T和testing.B是两个结构体类型。而*testing.T和*testing.B则分别为前两者的指针类型。它们分别是功能测试和基准测试所需的。

当在一个代码包中执行 go test 命令时，该代码包中的所有测试源码文件会被找到并运行。我们依然以 gopcp.v2/helper/log 包为例：

```
hc@ubt:~/golang/example.v2/src/gopcp.v2/helper/log$ go test
PASS
ok      gopcp.v2/helper/log             0.008s
```

这里使用 go test 命令在 gopcp.v2/helper/log 包中找到并运行了测试源码文件 logger_test.go，且调用其中所有的测试函数。命令行的回显信息表示我们通过了测试，并且运行测试源码文件中的测试程序共花费了 0.080 s。

最后插一句，Go 代码的文本文件需要以 UTF-8 编码存储。如果源码文件中出现了非 UTF-8 编码的字符，那么在运行、编译或安装的时候，Go 命令会抛出 illegal UTF-8 sequence 错误。

1.3.4　代码包

在 Go 中，代码包是代码编译和安装的基本单元，也是非常直观的代码组织形式。

1. 包声明

细心的读者可能已经发现，在 example.v2 项目的代码包中，多数源码文件名称看似都与包名没什么关系。实际上，在 Go 语言中，代码包中的源码文件可以任意命名。另外，这些任意名称的源码文件都必须以包声明语句作为文件中代码的第一行。比如，gopcp.v2/helper/log/base 包中的所有源码文件都要先声明自己属于某一个代码包：

```
package "base"
```

其中 package 是 Go 中用于包声明语句的关键字。Go 规定包声明中的包名是代码包路径的最后一个元素。比如，gopcp.v2/helper/log/base 包的源码文件包声明中的包名是 base。但是，不论命令源码文件存放在哪个代码包中，它都必须声明属于 main 包。

2. 包导入

代码包 gopcp.v2/helper/log 中的 logger.go 需要依赖 base 子包和 logrus 子包，因此需要在源码文件中使用代码包导入语句，如：

```
import "gopcp.v2/helper/log/base"
import "gopcp.v2/helper/log/logrus"
```

这需要用到代码包导入路径，即代码包在工作区的 src 目录下的相对路径。

当导入多个代码包时，你需要用圆括号括起它们，且每个代码包名独占一行。在使用被导入代码包中公开的程序实体时，需要使用包路径的最后一个元素加“.”的方式指定代码所在的包。

因此，上述语句可以写成：

```
import (
    "gopcp.v2/helper/log/base"
    "gopcp.v2/helper/log/logrus"
)
```

同一个源码文件中导入的多个代码包的最后一个元素不能重复，否则一旦使用其中的程序实体，就会引起编译错误。但是，如果你只导入不使用，同样会引起编译错误。一个解决方法是为其中一个起个别名，比如：

```
import (
    "github.com/Sirupsen/logrus"
```

```
    mylogrus "gopcp.v2/helper/log/logrus"
)
```

如果我们想不加前缀而直接使用某个依赖包中的程序实体，就可以用“.”来代替别名，如下所示：

```
import (
    . "gopcp.v2/helper/log/logrus"
)
```

看到那个“.”了吗？之后，在当前源码文件中，我们就可以这样做了：

```
var logger = NewLogger("gopcp") // NewLogger是gopcp.v2/helper/log/logrus包中的函数
```

这里强调一下，Go 中的变量、常量、函数和类型声明可统称为程序实体，而它们的名称统称为标识符。标识符可以是 Unicode 字符集中任意能表示自然语言文字的字符、数字以及下划线（_）。标识符不能以数字或下划线开头。

实际上，标识符的首字符的大小写控制着对应程序实体的访问权限。如果标识符的首字符是大写形式，那么它所对应的程序实体就可以被本代码包之外的代码访问到，也称为可导出的或公开的；否则，对应的程序实体就只能被本包内的代码访问，也称为不可导出的或包级私有的。要想成为可导出的程序实体，还需要额外满足以下两个条件。

- 程序实体必须是非局部的。局部的程序实体是指：它被定义在了函数或结构体的内部。
- 代码包所属目录必须包含在GOPATH中定义的工作区目录中。

代码包导入还有另外一种情况：如果只想初始化某个代码包，而不需要在当前源码文件中使用那个代码包中的任何程序实体，就可以用“_”来代替别名：

```
import (
    _ "github.com/Sirupsen/logrus"
)
```

这种情况下，我们只是触发了这个代码包中的初始化操作（如果有的话）。符号“_”就像一个垃圾桶，它在代码中使用很广泛，在后续章节中你还可以看到它的身影。

3. 包初始化

在 Go 中，可以有专门的函数负责代码包初始化，称为代码包初始化函数。这个函数需要无任何参数声明和结果声明，且名称必须为 init，如下所示：

```
func init() {
    fmt.Println("Initialize...")
}
```

Go会在程序真正执行前对整个程序的依赖进行分析，并初始化相关的代码包。也就是说，所有的代码包初始化函数都会在main函数（命令源码文件中的入口函数）执行前执行完毕，而且只会执行一次。另外，对于每一个代码包来说，其中的所有全局变量的初始化，都会在代码包的初始化函数执行前完成。这避免了在代码包初始化函数对某个变量进行赋值之后，又被该变量声明中赋予的值覆盖掉的问题。代码清单 1-1 展示了全局赋值语句、代码包初始化函数以及主函数的执行顺序。其中，双斜杠及其右边的内容为代码注释，Go编译器在编译的时候会将其忽略。

代码清单 1-1　pkg_init.go

```
package main // 命令源码文件必须在这里声明自己属于 main 包

import ( // 导入标准库代码包 fmt 和 runtime
    "fmt"
    "runtime"
)

func init() { // 代码包初始化函数
    fmt.Printf("Map: %v\n", m) // 格式化的打印
    // 通过调用 runtime 包的代码获取当前机器的操作系统和计算架构，
    // 而后通过 fmt 包的 Sprintf 方法进行格式化字符串生成并赋值给变量 info
   info = fmt.Sprintf("OS: %s, Arch: %s", runtime.GOOS, runtime.GOARCH)
}

// 非局部变量，map 类型，且已初始化
var m = map[int]string{1: "A", 2: "B", 3: "C"}

// 非局部变量，string 类型，未被初始化
var info string

func main() { // 命令源码文件必须有的入口函数，也称主函数
    fmt.Println(info) // 打印变量 info
}
```

运行这个文件：

```
hc@ubt:~/golang/example.v2/src/gopcp.v2/chapter1/pkginit$ go run pkg_init.go
Map: map[1:A 2:B 3:C]
OS: linux, Arch: amd64
```

关于每行代码的用途，在源码文件中我已经作了基本的解释。这里只解释这个小程序的输出。

第一行是对变量m的值格式化后的结果。可以看到，在函数init的第一条语句执行时，变量m已经被初始化并赋值了。这验证了：当前代码包中所有全局变量的初始化会在代码包初始化函数执行前完成。

第二行是对变量 info 的值格式化后的结果。变量 info 被定义时并没有显式赋值，因此它被赋予类型 string 的零值——""（空字符串）。之后，变量 info 在代码包初始化函数 init 中被赋值，并在入口函数 main 中输出。这验证了：所有的代码包初始化函数都会在 main 函数执行前执行完毕。

同一个代码包中可以存在多个代码包初始化函数，甚至代码包内的每一个源码文件都可以定义多个代码包初始化函数。Go 不会保证同一个代码包中多个代码包初始化函数的执行顺序。此外，被导入的代码包的初始化函数总是会先执行。在上例中，fmt 和 runtime 包中的 init 函数（如果有的话）会先执行，然后当前文件中的 init 函数才会执行。

1.4 标准命令简述

Go 本身包含了大量用于处理 Go 程序的命令和工具。go 命令就是其中最常用的一个，它有许多子命令，下面就来了解一下。

- build。用于编译指定的代码包或Go语言源码文件。命令源码文件会被编译成可执行文件，并存放到命令执行的目录或指定目录下。而库源码文件被编译后，则不会在非临时目录中留下任何文件。
- clean。用于清除因执行其他go命令而遗留下来的临时目录和文件。
- doc。用于显示Go语言代码包以及程序实体的文档。
- env。用于打印Go语言相关的环境信息。
- fix。用于修正指定代码包的源码文件中包含的过时语法和代码调用。这使得我们在升级Go语言版本时，可以非常方便地同步升级程序。
- fmt。用于格式化指定代码包中的Go源码文件。实际上，它是通过执行gofmt命令来实现功能的。
- generate。用于识别指定代码包中源码文件中的go:generate注释，并执行其携带的任意命令。该命令独立于Go语言标准的编译和安装体系。如果你有需要解析的go:generate注释，就单独运行它。这个命令非常有用，我常用它自动生成或改动Go源码文件。
- get。用于下载、编译并安装指定的代码包及其依赖包。从我们自己的代码中转站或第三方代码库上自动拉取代码，就全靠它了。
- install。用于编译并安装指定的代码包及其依赖包。安装命令源码文件后，代码包所在的工作区目录的bin子目录，或者当前环境变量GOBIN指向的目录中会生成相应的可执行文件。而安装库源码文件后，会在代码包所在的工作区目录的pkg子目

录中生成相应的归档文件。

- list。用于显示指定代码包的信息，它可谓是代码包分析的一大便捷工具。利用Go语言标准库代码包text/template中规定的模板语法，你可以非常灵活地控制输出信息。
- run。用于编译并运行指定的命令源码文件。当你想不生成可执行文件而直接运行命令源码文件时，就需要使用它。
- test。用于测试指定的代码包，前提是该代码包目录中必须存在测试源码文件。
- tool。用于运行Go语言的特殊工具。
- vet。用于检查指定代码包中的Go语言源码，并报告发现可疑代码问题。该命令提供了除编译以外的又一个程序检查方法，可用于找到程序中的潜在错误。
- version。用于显示当前安装的Go语言的版本信息以及计算环境。

执行这些命令时，可以通过附加一些额外的标记来定制命令的执行过程。下面是一些比较通用的标记。

- -a。用于强行重新编译所有涉及的Go语言代码包（包括Go语言标准库中的代码包），即使它们已经是最新的了。该标记可以让我们有机会通过改动更底层的代码包来做一些实验。
- -n。使命令仅打印其执行过程中用到的所有命令，而不真正执行它们。如果只想查看或验证命令的执行过程，而不想改变任何东西，使用它正合适。
- -race。用于检测并报告指定Go语言程序中存在的数据竞争问题。当用Go语言编写并发程序时，这是很重要的检测手段之一。
- -v。用于打印命令执行过程中涉及的代码包。这一定包括我们指定的目标代码包，并且有时还会包括该代码包直接或间接依赖的那些代码包。这会让你知道哪些代码包被命令处理过了。
- -work。用于打印命令执行时生成和使用的临时工作目录的名字，且命令执行完成后不删除它。这个目录下的文件可能会对你有用，也可以从侧面了解命令的执行过程。如果不添加此标记，那么临时工作目录会在命令执行完毕前删除。
- -x。使命令打印其执行过程中用到的所有命令，同时执行它们。

我们可以把这些标记看作命令的特殊参数，它们都可以添加到命令名称和命令的真正参数中间。用于编译、安装、运行和测试Go语言代码包或源码文件的命令都支持它们。

上面提到了tool这个子命令，它用来运行一些特殊的Go语言工具。直接执行go tool命令，可以看到这些特殊工具。它们有的是其他Go标准命令的底层支持，有的则是可以独当一面的利器。其中有两个工具值得特别介绍一下。

- pprof。用于以交互的方式访问一些性能概要文件。命令将会分析给定的概要文件，并根据要求提供高可读性的输出信息。这个工具可以分析的概要文件包括CPU概要文件、内存概要文件和程序阻塞概要文件。这些包含Go程序运行信息的概要文件，可以通过标准库代码包runtime和runtime/pprof中的程序来生成。
- trace。用于读取Go程序踪迹文件，并以图形化的方式展示出来。它能够让我们深入了解Go程序在运行过程中的内部情况。比如，当前进程中堆的大小及使用情况。又比如，程序中的多个goroutine是怎样被调度的，以及它们在某个时刻被调度的原因。Go程序踪迹文件可以通过标准库代码包runtime/trace和net/http/pprof中的程序来生成。

上述两个特殊工具对于Go程序调优非常有用。如果想探究程序运行的过程，或者想让程序跑得更快、更稳定，那么这两个工具是必知必会的。另外，这两个工具都受到 `go test` 命令的直接支持，因此你可以很方便地把它们融入到程序测试当中。

至此，我对Go语言标准命令和工具做了一个简要的介绍，你可以通过它们自带的帮助文档和用法进一步熟悉它们。此外，我还在GitHub网站上放置了一个《Go命令教程》，具体网址是 https://github.com/GoHackers/go_command_tutorial，其中有大多数Go标准命令的详细用法和示例。

1.5 问候程序

我相信你现在已经对Go的基本概念和标准命令有所了解了，下面通过一个小程序来切身感受一下Go程序的编写方法。

在讲代码包初始化的时候已经给出过一个小程序，其中展示了代码包声明语句、代码包导入语句、变量赋值语句，以及全局变量和函数的声明方法。代码清单1-2展示了一个问候程序。

代码清单 1-2 hello.go

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
```

```
    // 声明并初始化带缓冲的读取器
    // 准备从标准输入读取内容
    inputReader := bufio.NewReader(os.Stdin)
    fmt.Println("Please input your name:")
    // 以\n为分隔符读取一段内容
    input, err := inputReader.ReadString('\n')
    if err != nil {
        fmt.Printf("Found an error : %s\n", err)
    } else {
        // 对input进行切片操作，去掉内容中最后一个字节\n
        input = input[:len(input)-1]
        fmt.Printf("Hello, %s!\n", input)
    }
}
```

这里导入了两个我们未曾谋面的标准库代码包 bufio 和 os，另外还对字符串值做了切片。操作符:=用于在声明局部变量的同时对其赋值，是一种小语法糖。平行赋值规则允许我们同时对 input 和 err 赋值。能够如此赋值的另一个原因是，Go 中的函数可以返回多个结果值。作为结果值之一的 err 用于表示可能发生的错误，这也是 Go 惯用的方法。请注意程序中以 if 开头的那一行，它表示一个用于条件判断的代码块的开始。该代码块的含义是：若 err 的值不为 nil（空），则会打印错误，否则会打印问候语。

现在，运行这个文件：

```
hc@ubt:~/golang/example.v2/src/gopcp.v2/chapter1/hello $ go run hello.go
Please input your name:
Robert
Hello, Robert!
```

其中斜体表示用户的输入。程序根据用户的输入在标准输出上打印了一句问候。

1.6 小结

本章讨论了 Go 语言的一些必备知识和概念，另外还添加了一些简单的程序。相信读者已经对 Go 语言有了一个宏观的了解。在这里，我强烈建议你去下载本书的示例项目。请读一读前面讲述的那些程序，修改一下，然后再运行一下看看效果，多试验几次。请记住，这是学习一门编程语言最直接、最有效的途径。

第 2 章

语法概览

本章，我们会快速浏览一下 Go 的语法，内容涉及基本构成要素（比如标识符、关键字、字面量、操作符等）和基本类型（比如 bool、byte、rune、int、string 等）、高级类型（比如数组、切片、字典、接口、结构体等）和流程控制语句（if、switch、for、defer 等）。

2.1 基本构成要素

Go 的语言符号又称为词法元素，共包括 5 类内容——标识符（identifier）、关键字（keyword）、字面量（literal）、分隔符（delimiter）和操作符（operator），它们可以组成各种表达式和语句，而后者都无需以分号结尾。

2.1.1 标识符

标识符可以表示程序实体，前者即为后者的名称。在一般情况下，同一个代码块中不允许出现同名的程序实体。标识符的组成规则请见 1.3.4 节中“包导入”部分。使用不同代码包中的程序实体需要用到限定标识符，比如：os.O_RDONLY。

另外，Go 中还存在着一类特殊的标识符，叫作预定义标识符，它们是在 Go 源码中声明的。这类标识符包括以下几种。

- 所有基本数据类型的名称。
- 接口类型error。
- 常量true、false和iota。

所有内建函数的名称，即 append、cap、close、complex、copy、delete、imag、len、make、new、panic、print、println、real 和 recover。

这里强调一下空标识符，它由一个下划线_表示，一般用在变量声明或代码包导入语句中。若在代码中存在一个变量 x，但是却不存在任何对它的使用，则编译器会报错。如果在变量 x 的声明代码后添加这样一行代码：

```
_ = x
```

就可以绕过编译器检查，使它不产生任何编译错误。这是因为这段代码确实用到了变量 x，只不过它没有在变量 x 上进行任何操作，也没有将它赋值给任何其他变量。空标识符就像一个垃圾桶。在相关初始化工作完成之后，操作对象就会被弃之不用。

2.1.2 关键字

关键字是指被编程语言保留的字符序列，编程人员不能把它们用作标识符。因此，关键字也称为保留字。

Go 的关键字可以分为 3 类，包括用于程序声明的关键字、用于程序实体声明和定义的关键字，以及用于程序流程控制的关键字，如表 2-1 所示。

表2-1 关键字及分类

类别	关键字
程序声明	import和package
程序实体声明和定义	chan、const、func、interface、map、struct、type和var
程序流程控制	go、select、break、case、continue、default、defer、else、fallthrough、for、goto、if、range、return和switch

Go 的关键字共有 25 个，其中与并发编程有关的关键字有 go、chan 和 select。

这里特别说明一下关键字 type 的用途——类型声明。我们可以使用它声明一个自定义类型：

```
type myString string
```

这里把名为 myString 的类型声明为 string 类型的一个别名类型。反过来说，string 类型是 myString 类型的潜在类型。再看另一个例子，基本数据类型 rune 是 int32 类型的一个别名类型。int32 类型就是 rune 类型的潜在类型。虽然类型及其潜在类型是不同的两个类型，但是它们的值可以进行类型转换，例如：string(mystring("ABC"))。这样的类型转换不会产生新值，几乎没什么代价。

自定义的类型一般都会基于 Go 中的一个或多个预定义类型，就像上面的 myString 和 string 那样。如果为自定义类型关联若干方法（函数的变体），那么还可以让它成为某

个或某些接口类型的实现类型。另外，还有一个比较特殊的类型，叫空接口。它的类型字面量是 interface{}。在 Go 语言中，任何类型都是空接口类型的实现类型。

2.1.3 字面量

简单来说，字面量就是值的一种标记法。但是，在 Go 中，字面量的含义要更加广泛一些。我们常常用到的字面量有以下 3 类。

- 用于表示基础数据类型值的各种字面量。这是最常用到的一类字面量，例如，表示浮点数类型值的12E-3。
- 用于构造各种自定义的复合数据类型的类型字面量。例如，下面的字面量定义了一个名称为Name的结构体类型：

```
type Name struct {
    Forename    string
    Surname     string
}
```

- 用于表示复合数据类型的值的复合字面量，它可以用来构造struct（结构体）、array（数组）、slice（切片）和map（字典）类型的值。复合字面量一般由字面类型以及被花括号包裹的复合元素的列表组成。字面类型指的就是复合数据类型的名称。例如，下面的复合字面量构造出了一个Name类型的值：

```
Name{Forename: "Robert", Surname: "Hao"}
```

其中 Name 表示这个值的类型，紧随其后的就是由键值对表示的复合元素列表。

2.1.4 操作符

操作符，也称运算符。它是用于执行特定算术或逻辑操作的符号，操作的对象称为操作数。下面我们通过表 2-2 来看看 Go 中的操作符都有哪些。

表2-2 操作符

符 号	说 明	示 例
||	逻辑或操作，它是二元操作符，同时也属于逻辑操作符	true || false // 结果是true
&&	逻辑与操作，它是二元操作符，同时也属于逻辑操作符	true && false // 结果是false
==	相等判断操作，它是二元操作符，同时也属于比较操作符	"abc" == "abc" // 结果是true
!=	不等判断操作，它是二元操作符，同时也属于比较操作符	"abc" != "Abc" // 结果是true
<	小于判断操作，它是二元操作符，同时也属于比较操作符	1 < 2 // 结果是true
<=	小于或等于判断操作，它是二元操作符，同时也属于比较操作符	1 <= 2 // 结果是true

（续）

<table>
<tr><th>符　　号</th><th>说　　明</th><th>示　　例</th></tr>
<tr><td>></td><td>大于判断操作，它是二元操作符，同时也属于比较操作符</td><td>3 > 2 // 结果是true</td></tr>
<tr><td>>=</td><td>大于或等于判断操作，它是二元操作符，同时也属于比较操作符</td><td>3 >= 2 // 结果是true</td></tr>
<tr><td>+</td><td>求和操作，它既是一元操作符又是二元操作符，同时也属于算术操作符。若作为一元操作符，此操作符不会对原值产生任何影响</td><td>+1 // 结果是1
1 + 2 // 结果是3</td></tr>
<tr><td>-</td><td>求差操作，它既是一元操作符又是二元操作符。若作为一元操作符，则表示求反操作。同时，它也属于算术操作符</td><td>-1 // 结果为-1（1的相反数）
1 - 3 // 结果是-2</td></tr>
<tr><td>|</td><td>按位或操作，它是二元操作符，同时也属于算术操作符</td><td>5 | 11 // 结果是15</td></tr>
<tr><td>^</td><td>按位异或操作，它既是一元操作符又是二元操作符。若作为一元操作符，则表示按位补码操作。同时，它也属于算术操作符</td><td>5 ^ 11 // 结果是14
^5 // 结果是-6</td></tr>
<tr><td>*</td><td>求乘积操作，它既是一元操作符又是二元操作符。同时，它也属于算术操作符和地址操作符。若作为地址操作符，则表示取值操作</td><td>*p // 若p为指向整数类型值2的指针类型值，则结果为2
2 * 5 // 结果是10</td></tr>
<tr><td>/</td><td>求商操作，它是二元操作符，同时也属于算术操作符</td><td>10 / 5 // 结果是2</td></tr>
<tr><td>%</td><td>求余数操作，它是二元操作符，同时也属于算术操作符</td><td>12 % 5 // 结果是2</td></tr>
<tr><td><<</td><td>按位左移操作，它是二元操作符，同时也属于算术操作符</td><td>4 << 2 // 结果是16</td></tr>
<tr><td>>></td><td>按位右移操作，它是二元操作符，同时也属于算术操作符</td><td>4 >> 2 // 结果是1</td></tr>
<tr><td>&</td><td>按位与操作，它既是一元操作符又是二元操作符。同时，它也属于算术操作符和地址操作符。若作为地址操作符，则表示取址操作</td><td>&v // 结果为标识符v所代表的值在内存中的地址
5 & 11 // 结果是1</td></tr>
<tr><td>&^</td><td>按位清除操作，它是二元操作符，同时也属于算术操作符</td><td>5 &^ 11 // 结果是4</td></tr>
<tr><td>!</td><td>逻辑非操作，它是一元操作符，同时也属于逻辑操作符</td><td>!b // 若b的值为false，则表达式的结果为true</td></tr>
<tr><td><-</td><td>接收操作，它是一元操作符，同时也属于接收操作符</td><td><- ch // 若ch代表了元素类型为byte的通道类型值，则此表达式就表示从ch中接收一个byte类型值的操作</td></tr>
</table>

如表 2-2 所示，Go 的操作符一共有 21 个，并分为了 5 类：算术操作符、比较操作符、逻辑操作符、地址操作符和接收操作符。

当一个表达式中存在多个操作符时，就涉及操作顺序的问题。在 Go 中，一元操作符拥有最高的优先级，而二元操作符的优先级如表 2-3 所示。

表2-3　二元操作符的优先级

<table>
<tr><th>优 先 级</th><th>操　作　符</th></tr>
<tr><td>5</td><td>*、/、%、<<、>>、&和&^</td></tr>
<tr><td>4</td><td>+、-、|和^</td></tr>
<tr><td>3</td><td>==、!=、<、<=、>和>=</td></tr>
<tr><td>2</td><td>&&</td></tr>
<tr><td>1</td><td>||</td></tr>
</table>

在表 2-3 中，我以数字来表示操作符的优先级，数字越大就意味着优先级越高。如果在一个表达式中出现了处于相同优先级的多个操作符，且这些操作符之间仅存在操作数，那么就会按照从左到右的顺序进行操作。当然，我们可以使用圆括号显式改变原有的操作顺序，例如表达式 `a << (4 * b) & c` 等同于`(a << (4 * b)) & c`，即子表达式 `4 * b` 会先被求值。

最后需要注意的是，`++`和`--`是语句而不是表达式，因而它们不存在于任何操作符优先级层次之内。例如，表达式`*p--`等同于`(*p)--`。

2.1.5 表达式

表达式是把操作符和函数作用于操作数的计算方法。在 Go 中，表达式是构成具有词法意义的代码的最基本元素。Go 的表达式有很多种，具体如表 2-4 所示。

表2-4 表达式的种类

种　类	用　途	示　例
选择表达式	选择一个值中的字段或方法	context.Speaker // context是变量名
索引表达式	选取数组、切片、字符串或字典值中的某个元素	array1[1] // array1表示一个数组值，其长度必须大于1
切片表达式	选取数组、数组指针、切片或字符串值中的某个范围的元素	slice1[0:2] // slice1表示一个切片值，其容量必须大于或等于2
类型断言	判断一个接口值的实际类型否是为某个类型，或一个非接口值的类型是否实现了某个接口类型	v1.(I1) // v1表示一个接口值，I1表示一个接口类型
调用表达式	调用一个函数或一个值的方法	v1.M1() // v1表示一个值，M1表示与该值关联的一个方法

关于类型断言，有两点需要注意，如下。

- 如果v1是一个非接口值，那么必须在做类型断言之前把它转换成接口值。因为Go中的任何类型都是空接口类型的实现类型，所以一般会这样做：`interface{}(v1).(I1)`。
- 如果类型断言的结果为否，就意味着该类型断言是失败的。失败的类型断言会引发一个运行时恐慌（或称运行时异常），解决方法是：

  ```
  var i1, ok = interface{}(v1).(I1)
  ```

  这里声明并赋值了两个变量，其中 ok 是布尔类型的变量，它的值体现了类型断言的成败。如果成功，i1 就会是经过类型转换后的 I1 类型的值，否则它将会是 I1 类型的零值（或称默认值）。如此一来，当类型断言失败时，运行时恐慌就不

会发生。

关键字 var 用于变量的声明。在它和等号=之间可以有多个由逗号隔开的变量名。这种在一条语句中同时为多个变量赋值的方式叫平行赋值。另外，如果在声明变量的同时进行赋值，那么等号左边的变量类型可以省略。如果不使用上述几个技巧的话，上面那条语句可以写成：

```
var i1 I1
var ok bool
i1, ok = interface{}(v1).(I1)
```

另一方面，上面那条语句还可以简写成：

```
i1, ok := interface{}(v1).(I1)
```

这种简写方式只能出现在函数中。有了符号:=，关键字 var 也可以省略了，这叫短变量声明。

2.2 基本类型

Go 有很多预定义类型，这里简单地把它们分为基本类型和高级类型。Go 的基本类型并不多，并且大部分都与整数相关，具体如表 2-5 所示。

表2-5 基本类型

名称	宽度（字节）	零值	说明
bool	1	false	布尔类型，其值不为真即为假。真用常量true表示，假由常量false表示
byte	1	0	字节类型，它也可以看作是一个由8位二进制数表示的无符号整数类型
rune	4	0	rune类型，它是由Go语言定义的特有的数据类型，专用于存储Unicode字符。它也可以看作一个由32位二进制数表示的有符号整数类型
int/uint	-	0	有符号整数类型/无符号整数类型，其宽度与平台有关
int8/uint8	1	0	由8位二进制数表示的有符号整数类型/无符号整数类型
int16/uint16	2	0	由16位二进制数表示的有符号整数类型/无符号整数类型
int32/uint32	4	0	由32位二进制数表示的有符号整数类型/无符号整数类型
int64/uint64	8	0	由64位二进制数表示的有符号整数类型/无符号整数类型
float32	4	0.0	由32位二进制数表示的浮点数类型
float64	8	0.0	由64位二进制数表示的浮点数类型
complex64	8	0.0 + 0.0i	由64位二进制数表示的复数类型，它由float32类型的实部和虚部联合表示

（续）

名 称	宽度（字节）	零 值	说 明
complex128	16	0.0 + 0.0i	由128位二进制数表示的复数类型，它由float64类型的实部和虚部联合表示
string	-	""	字符串类型。一个字符串类型表示了一个字符串值的集合。而一个字符串值实质上是一个字节序列。注意：字符串类型的值是不可变的，即一旦创建，其内容就不可改变

Go 的基本类型共有 18 个，其中 int 和 uint 的实际宽度会根据计算架构的不同而不同。在 386 计算架构下，它的宽度为 32 比特，即 4 字节。在 amd64（有时也称为 x86-64）计算架构下，它们的宽度为 64 比特，即 8 字节。

byte 可以看作类型 uint8 的别名类型，而 rune 可以看作 int32 的别名类型。rune 类型专用于存储 Unicode 编码的单个字符。我们可以用 5 种方式来表示一个 rune 字面量，具体如下。

- 该rune字面量所对应的字符，比如'a'、'ä'或'—'，这个字符必须是Unicode编码规范所支持的。
- 使用“\x”为前导并后跟两位十六进制数，这种方式可以表示宽度为1字节的值，即一个ASCII编码值。
- 使用“\”为前导并后跟3位八进制数，这种表示法也只能表示有限宽度的值，即它只能用于表示在0和255之间的值。它与上一个表示法的表示范围是一致的。
- 使用“\u”为前导并后跟4位十六进制数，它只能用于表示2字节宽度的值。
- 使用“\U”为前导并后跟8位十六进制数，它只能用于表示4字节宽度的值，这种方式即为Unicode编码规范中的UCS-4表示法。

此外，rune 字面量还支持一类特殊的字符序列——转义符。转义符的表示方式是在“\”后面追加一个特定的单字符，参见表 2-6。

表2-6 转义符说明

转义符	Unicode代码点	说 明
\a	U+0007	告警铃声或蜂鸣声
\b	U+0008	退格符
\f	U+000C	换页符
\n	U+000A	换行符
\r	U+000D	回车符
\t	U+0009	水平制表符
\v	U+000b	垂直制表符

（续）

转义符	Unicode代码点	说　明
\\	U+005c	反斜杠
\'	U+0027	单引号，仅在rune字面量中有效
\"	U+0022	双引号，仅在string字面量中有效

除了上述转义符外，rune 字面量中以“\”为前导的字符序列都是不合法的。

在 Go 中，字符串值表示了一个字符值的集合。在底层，一个字符串值即一个包含了若干字节的序列。长度为 0 的序列与一个空字符串相对应。字符串的长度即底层字节序列中字节的个数。一个字符串字面量的长度在编译期间就能够确定。字符串字面量有两种表示形式：原生字符串字面量（由反引号“`”包裹）和解释型字符串字面量（由双引号“"”包裹）。前者所见即所得，而后者则可以解析转义字符。

注意，字符串值是不可变的！我们不可能改变一个字符串的内容。对字符串的操作只会返回一个新字符串，而不会改变原字符串并返回。

只有基本类型及其别名类型才可以作为常量的类型。常量的声明会用到关键字 const。单一常量声明一般由关键字 const、常量名、常量类型、等号=和常量值组成。下面是两个常量的声明：

```
const DEFAULT_IP string = "192.168.0.1"
const DEFAULT_PORT int = 9001
```

像这样多个常量同时声明还可以简写成：

```
const (
    DEFAULT_IP string = "192.168.0.1"
    DEFAULT_PORT int = 9001
)
```

注意，Go 官方的命名规范中指出常量的命名要用驼峰法。但是，我认为常量的命名应该使用大小写一致的单词，且多个单词时用下划线进行分割，这样才能从名称上快速区分常量和变量，本书示例中的常量也都是按照这种方式命名的。

2.3 高级类型

Go 的基本数据类型都完整地确定了类型的方方面面。而其高级数据类型却不同，它们是为用户定义自己的数据类型而服务的。比如，我们可以通过指定元素类型和长度来形成一个确切的数组类型，也可以通过指定键类型和元素类型来形成一个确切的字典类

型，等等。Go 的高级数据类型相当于自定义数据类型的模板或制作工具。

2.3.1 数组

数组（array）就是由若干相同类型的元素组成的序列。先看一个示例：

```
var ipv4 [4]uint8 = [4]uint8{192, 168, 0, 1}
```

在这条赋值语句中，我为刚声明的变量 ipv4 赋了值。在这种情况下，变量名右边的类型字面量可以省略。如果它在函数里面，那么关键字 var 也可以省略，但赋值符号必须由=变为:=。

2

类型字面量[4]uint8 表明这个变量的类型是长度为 4 且元素类型为 uint8 的数组类型。注意，数组的长度是数组类型的一部分。只要类型声明中的数组长度不同，即使两个数组类型的元素类型相同，它们也是不同的类型。更重要的是，一旦在声明中确定了数组类型的长度，就无法改变它了。

数组类型的零值一定是一个不包含任何元素的空数组。一个类型的零值即为该类型变量未被显式赋值时的默认值。以 ipv4 为例，其所属类型的零值就是[4]uint8{}。

在上述示例中，等号右边的字面量表示该类型的一个值。我们可以忽略那个在方括号中表示数组长度的正整数值，示例如下：

```
[...]uint8{192, 168, 0, 1}
```

方括号中的特殊标记...表示需由 Go 编译器计算该值的元素数量并以此获得其长度。

索引表达式和切片表达式都可以应用于数组值，前者会得到该数组值中的一个元素，而后者则会得到一个元素类型与之相同的切片值。此外，Go 的内建函数 len 和 cap 也都可以应用于数组值，并都可以得到其长度。

当需要详细规划程序所用的内存时，数组类型非常有用。使用数组值可以完全避免耗时费力的内存二次分配操作，因为它的长度是不可变的。

2.3.2 切片

切片（slice）可以看作一种对数组的包装形式，它包装的数组称为该切片的底层数组。反过来讲，切片是针对其底层数组中某个连续片段的描述。下面的代码声明了一个切片类型的变量：

```
var ips = []string{"192.168.0.1", "192.168.0.2", "192.168.0.3"}
```

与数组不同，切片的类型字面量（如[]string）并不携带长度信息。切片的长度是可变的，且并不是类型的一部分；只要元素类型相同，两个切片的类型就是相同的。此外，一个切片类型的零值总是 nil，此零值的长度和容量都为 0。

切片值相当于对某个底层数组的引用。其内部结构包含了 3 个元素：指向底层数组中某个元素的指针、切片的长度以及切片的容量。这里所说的容量是指，从指针指向的那个元素到底层数组的最后一个元素的元素个数。

切片值的容量意味着，在不更换底层数组的前提下，它的长度的最大值。我们可以通过 cap 函数和切片表达式，在此前提下最大化一个切片值的长度，就像这样：ips[:cap(ips)]。

除了 len 和 cap，内建函数 append 也可应用于切片值，示例如下：

```
ips = append(ips, "192.168.0.4")
```

在这条语句中，等号右边的代码会依据 ips 的值生成新的切片值，并把"192.168.0.4"追加到该值的最后。然后，赋值操作会把这个新的切片值再赋给 ips。注意，新、旧切片值可能指向不同的底层数组。若新切片值的底层数组的长度不足以完成元素追加操作，它将会被更长的底层数组替换，以容纳更多的元素。

另一个值得一提的内建函数是 make，它用于初始化切片、字典或通道类型的值。对于切片类型来说，用 make 函数的好处就是可以用很短的代码初始化一个长度很大的值，示例如下：

```
ips = make([]string, 100)
```

等号右边的代码会初始化一个元素类型为 string 且长度为 100 的切片值。可以试想一下，如果用复合字面量初始化这样一个切片值将需要多少代码。用 make 函数初始化的切片值中的每一个元素值都会是其元素类型的零值，这里 ips 中的那 100 个元素的值都会是空字符串""。

2.3.3 字典

Go 中字典类型是散列表（hash table）的一个实现，其官方称谓是 map。散列表是一个实现了关联数组的数据结构，关联数组是用于表示键值对的无序集合的一种抽象数据类型。Go 中称键值对为键–元素对，它把字典值中的每个键都看作与其对应的元素的索引，这样的索引在同一个字典值中是唯一的。

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ或微信312082710

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我微信或QQ312082710

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，本人概不负责，我们仅仅只是帮助你寻找到你要的pdf而已。