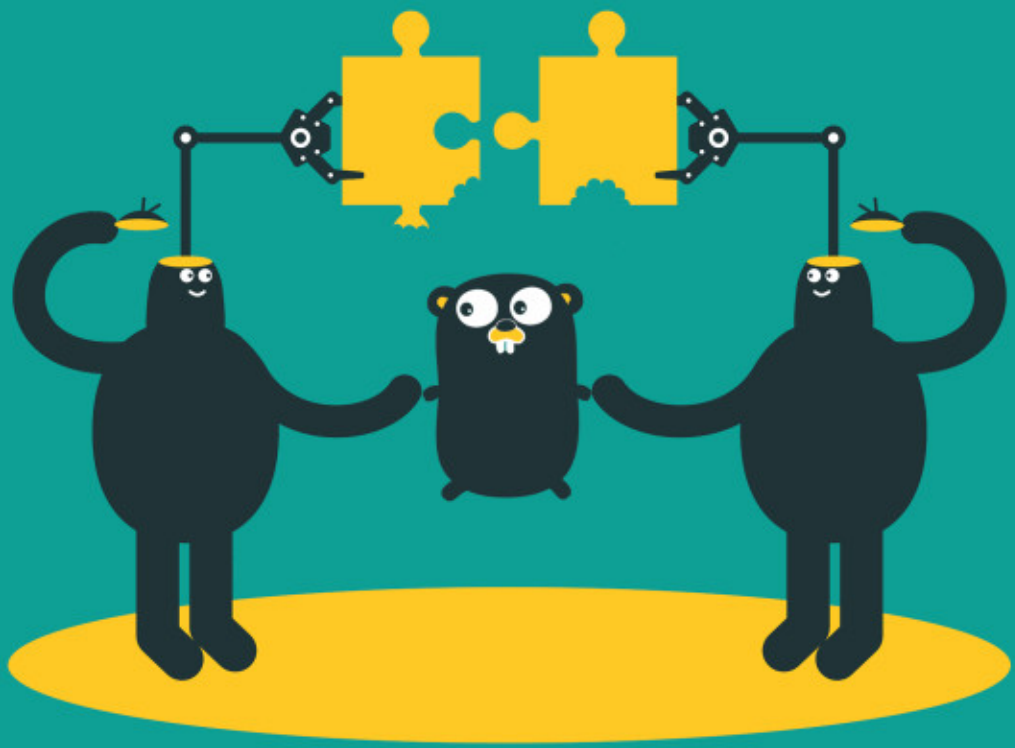
# GET PROGRAMMING WITH GO



Nathan Youngman
Roger Peppé

MANNING

# 《Get Programming with Go》中文版（试读版）

欢迎阅读本文档！ 这是为了让读者能够第一时间了解《Get Programming with Go》中文版而创建的试读版本， 包含了书本的《译者序》、《前言》、《关于本书》和第 1 至第 5 章。

**本文档基于译者原稿生成，未经出版社处理，在内容和排版上与正式出版的图书会有所不同。*

## 购买本书

如果您对本书感兴趣的话， 那么可以考虑在 GPWGCN.com (http://GPWGCN.com) 购买完整版本。

《Get Programming with Go》中文版全书共包含 6 个单元，32 个大章。

## 版权声明

# 目录

# 译者序

跟本书的相遇，纯属偶然。

2018 年 6 月下旬的一个晚上，我从外面散步完回家，坐在阳台上享受着初夏的微风，期间百无聊赖地拿起手机刷起了朋友圈，并在上面发现杨海玲编辑正在为《Get Programming with Go》一书招募译者。

我非常喜欢 Go 语言，并且一直对这门语言保持关注。尽管之前已经翻译过《Go Web 编程》一书，但我还是常常期待有机会可以再次翻译 Go 方面的图书。 因此当我看到《Get Programming with Go》正在寻找译者的消息时，我马上有预感觉得自己的愿望说不定就要实现了。

果不其然，当我快速地在 Manning 出版社的网站上浏览了《Get Programming with Go》的相关信息并且试读了公开的章节之后，我强烈地感受到这是一本非常有趣的 Go 图书，并且我深信凭借本书优质的内容，它将在未来的 Go 入门读物中占有自己的一席之地。

简单来说，如果我想要再次翻译一本 Go 图书的话，那么毫无疑问就应该是这本《Get Programming with Go》！ 在打定主意之后，我就向杨海玲编辑表达了想要翻译本书的意向，并在之后顺利地成为了本书的译者。

---

Go 作为一门广受关注的热门语言，在市场上从来不缺少相关的图书，特别是面向初学者的图书。

跟市面上很多声称自己是入门书但是却只会一股脑地将各种语言细节硬塞给你的“伪入门书”不一样，这是一本真正面向初学者的书。 整本书的学习曲线非常平缓，不会像过山车那样忽高忽低。 书中的内容首先从变量、循环、分支、类型等基础知识开始，逐渐进阶至函数、方法、收集器和结构，最后再深入到指针、错误处理和并发等高级特性。 只要沿着书本一页页读下去，你就会循序渐进地学到越来越多 Go 语言知识，并且逐步掌握 Go 语言的众多特性。

除了上述提到的优点之外，本书还是一本非常有趣的书。 作者在书中列举了大量跟天文以及航天有关的例子，你不仅要计算从地球乘坐宇宙飞船航行至火星所需的天数，还要在火星上放置探测器以便查找生命存在的痕迹，甚至还要想

办法改造火星使得它能够适宜人类居住。特别值得一提的是，书中很多地方都带有可爱的地鼠（gopher）插图，它们就像旅行途中优美的景色一样，将为我们的学习旅途增添大量的乐趣。

总而言之，这是一本既有趣又实用的 Go 语言入门书。如果你只想读一本关于 Go 语言的入门书，那么我强烈推荐你读这一本。

---

回想当初，自己在通过阅读图书学习编程的时候，总是希望能够成为译者，做出属于自己的翻译作品。但是在实际成为译者之后才明白，原来翻译并不是一件简单的事情，它考验的不仅仅是译者的语言能力、文字能力和技术能力，它还要求译者要有足够的耐心和锲而不舍的毅力。

具体到这次的《Get Programming with Go》一书，作者在遣词造句方面虽然没有太多难懂的地方，但是内容中却使用了大量有趣的例子，比如搭乘太空飞船前往火星、计算人类在火星上的重量和年龄、调查火箭发射事故、在火星上放置探测器、组建地鼠工厂等等。如何在译文中准确无误地表达原文的意思并且保持原文的趣味性就成了本次翻译的一大挑战。换句话来说，译文不仅要保持原文的“形”，还要兼顾原文的“神”，做到“形神兼备”才行。为此，我在这方面用了足够多的时间也花了足够多的功夫，我有信心大家在阅读本书的时候，能够感受到跟原作一样的趣味性。

除了书中有趣的例子之外，本书翻译的另一个难点在于例子中涉及的大量外部知识，特别是跟天文、航天有关的知识，比如地球和火星之间的距离、用光速到达火星所需的时间、火星上各个着陆点的名字，诸如此类。为了能够正确地翻译这些知识，我通常都会在维基百科上面查找并且验证相关的信息，然后再进行翻译。曾经有一段时间我的浏览器上面打开的都是天文、航天相关的维基页面，甚至让我产生了一种自己在 NASA 上班的错觉（笑）。

虽然花了不少时间，但这本《Get Programming with Go》终于还是翻译完了。今后如果有机会的话，我大概也会继续进行翻译工作吧。希望在不远的将来，当读者们在译者一栏看到“黄健宏”这个名字的时候，能够像看到食品包装上粘贴的“安全许可”标识一样安心：他们不必担心书本的翻译质量，只需要尽情地享受阅读带来的快乐即可。当然要达成这个目标并不容易，但我会在接下来的工作中继续努力，希望大家可以一如既往地支持我。

最后，我要感谢人民邮电出版社和杨海玲编辑又一次把一本有趣的 Go 书交给了我进行翻译，我也要感谢我的家人和朋友，他们的关怀和帮助让我得以顺利完成本书。

黄健宏
2019 年 10 月于清远

# 前言

*一切都在变化，没有东西是亘古不变的。——* 赫拉克利特

在 2005 年欧洲旅行期间，Nathan 听说了一些关于新框架 Ruby On Rails 的传闻。 于是他在赶回加拿大艾伯塔省庆祝圣诞节期间，在市中心的计算机书店购入了一本《Agile Web Development with Rails》（2005年，Pragmatic Bookshelf 出版社出版），并在接下来的两年里将自己的事业从 ColdFusion 转向至了 Ruby 。

在英国的约克大学，Roger 被引荐给了经过彻底精简之后的贝尔实验室，并在那里跟包括 Go 创造者 Rob Pike 、Ken Thompson 在内的成员一起研究 UNIX 以及由同一批人发明的 Plan 9 操作系统。 Roger 对此产生了极大的兴趣，并在之后开始进行 Inferno 系统的相关工作，该系统使用了独有的 Limbo 语言，它是 Go 的一个关系密切的原型。

当 Go 在 2009 年 11 月作为开源项目对外发布的时候，Roger 立即就发现了它的潜力，他开始使用 Go 并为 Go 的标准库和生态系统做贡献。 Roger 至今仍然对 Go 的成功感到高兴，除了全职使用 Go 进行编程之外，他还运营着一个本地的 Go 见面会。

另一边，Nathan 虽然观看了 Rob Pike 发布 Go 的技术演讲，但他在 2011 年之前都没有认真地审视过这种语言。 直到一位同事高度评价了 Go 之后，Nathan 才最终在圣诞节假期通读了《The Go Programming Language Phrasebook》（2012 年 ，Addison-Wesley Professional 出版社出版）的毛边本。 在之后的数年里，Nathan 从使用 Go 编写业余项目并撰写 Go 相关的博客（nathany.com）开始，逐渐转向至组织本地的 Go 见面会（edmontongo.org）并在工作中使用 Go 。

因为工具和技术都在持续地变化和改进，所以对计算机科学世界的学习总是永无止境的。 无论你已经拥有计算机科学学位还是刚开始接触这一行，自学新技能都是非常重要的。 我们衷心希望本书可以在你学习 Go 编程语言的过程中予以帮助。

# 致谢

能够为您撰写本书并帮助您学习 Go 是一种莫大的荣幸，非常感谢您的阅读！

除了封面上的两位作者之外，本书还包含了许多人的贡献。

首先也是最重要的，我们要感谢本书的编辑 Jennifer Stout 和 Marina Michaels 提供有价值的反馈信息，并且持续地、循序渐进地推动我们向终点进发。 我们要感谢 Joel Kotarski 和 Matt Merkes 提供准确的技术编辑，感谢 Christopher Haupt 提供技术校对，并感谢文字编辑 Corbin Collins 改善了我们的语法和风格。 此外，我们还要感谢 Bert Bates 和系列编辑 Dan Maharry 、 Elesha Hyde ，他们的谈话和指导对本书的形成提供了帮助。

Olga Shalakhina 和 Erick Zelaya 为本书提供了精彩绝伦的插图，Monica Kamsvaag 为本书设计了封面，April Milne 负责为本书美化和修饰图表，而 Renée French 则为 Go 创造了人见人爱的吉祥物，我们要对他们表示感谢。 特别鸣谢 Dan Allen ，他创建了本书创作时使用的 AsciiDoctor ，并为我们提供了持续的支持。

感谢 Marjan Bace、Matko Hrvatin、Mehmed Pasic、Rebecca Rinehart、Nicole Butterfield、Candace Gillhoolley、Ana Romac、Janet Vail、David Novak、Dottie Marsico、Melody Dolab、Elizabeth Martin 以及 Manning 出版社的其他工作人员，是他们让本书变成了现实，使得你能够亲手读到这本《Get Programming with Go》。

感谢 Aleksandar Dragosavljević 将本书送达至一众审稿人，感谢包括 Brendan Ward、 Charles Kevin、 Doug Sparling、 Esther Tsai、 Gianluigi Spagnuolo、 Jeff Smith、 John Guthrie、 Luca Campobasso、 Luis Gutierrez、 Mario Carrion、 Mikaël Dautrey、 Nat Luengnaruemitchai、 Nathan Farr、 Nicholas Boers、 Nicholas Land、 Nitin Gode、 Orlando Sánchez、 Philippe Charrière、 Rob Weber、 Robin Percy、 Steven Parr、 Stuart Woodward、 Tom Goodheard、 Ulises Flynn、 以及 William E. Wheeler 在内的所有审稿人，他们都提供了有价值的反馈信息。 我们还要感谢那些通过论坛提供反馈的早期阅读者。

最后，我们还要对 Michael Stephens 表示感谢，是他提出了撰写一本书的疯狂想法，也感谢 Go 社区创造了我们乐于为其写书的语言及生态。

## Nathan Youngman

理所当然地，我需要感谢我的父母，是他们生育并抚养了我。 我的父母从我小时候开始就鼓励我钻研计算机编程，并为我提供了相应的书本、课程以及接触计算机的机会。

除了出现在封面上的官方评论之外，我还要感谢 Matthias Stone 为早期草稿提供反馈，还有 Terry Youngman 帮助我进行头脑风暴以获得更多想法。 我要感谢埃德蒙顿 Go 社区为我加油打气，还有我的雇主 Mark Madsen 给我以方便，让我得以将写书工作付诸实践。

我要向我的合著者 Roger Peppé 致以最大的感谢，他通过撰写第 7 单元让原本漫长的道路变得触手可及，并为本项目注入了强劲的能量。

## Roger Peppé

我得向我的妻子 Carmen 致以最诚挚的感谢，创作本书占用了我们本该在山间漫步的时间，而她却对此毫无怨言，并且始终如一地支持着我。

非常感谢 Nathan Youngman 和 Manning 出版社对我的信任，是他们让我成为了本书的合著者，并且在本书创作的最后阶段仍然对我保持耐心。

# 关于本书

## 目标读者

Go 适用于各种水平的程序员，这对于任何大型项目来说都是至关重要的。 作为一种相对较为小型的语言，Go 的语法极少，需要掌握的概念也不多，因此它非常适合用作初学者的入门语言。

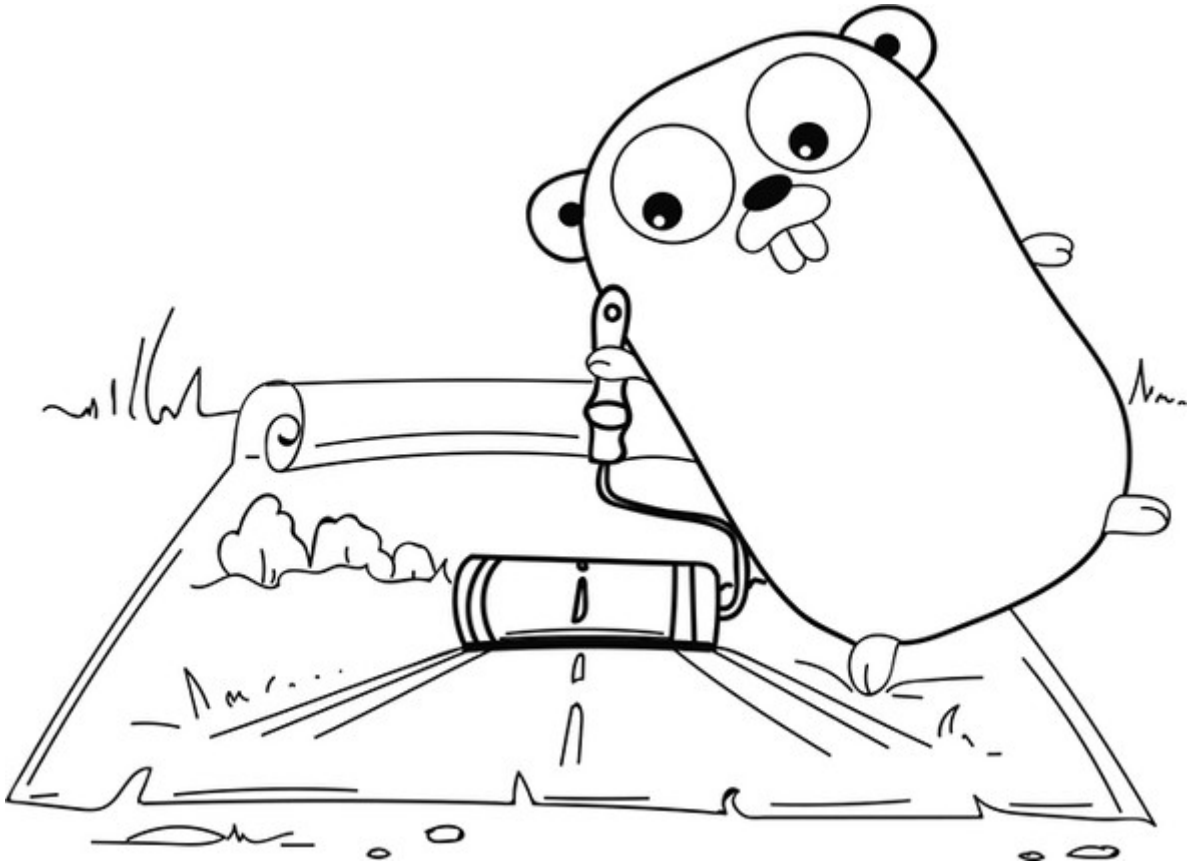
遗憾的是，很多学习 Go 语言的资源都假设读者拥有 C 编程语言的工作经验，而本书的目的则在于弥补这一缺陷，为脚本使用者、业余爱好者和初学者提供一条学习 Go 语言的康庄大道。 为了让起步的过程变得更容易一些，本书的所有代码清单和练习都可以在 Go 游乐场（play.golang.org）里面执行，你在阅读本书的时候甚至不需要安装任何东西。

如果你曾经使用过诸如 JavaScript、Lua、PHP、Perl、Python 或者 Ruby 这样的脚本语言，那么你已经做好了学习 Go 的万全准备；而如果你曾经使用过 Scratch 或者 Excel 的公式，又或者编写过 HTML ，那么你毫无疑问可以像 Audrey Lim 在她的演讲《初学者之心》（youtu.be/fZh8uCInEfw）中所说的一样，选择 Go 作为你的第一门“真正”的编程语言。 虽然掌握 Go 并不是一件容易的事情，这需要相应的耐心和努力，但我们希望本书在这个过程中能够助你一臂之力。

## 组织方式和路线图

本书将以循序渐进的方式讲解高效使用 Go 所必需的概念，并提供大量练习来磨砺你的技能。 这是一本初学者指南，旨在从头到尾地进行阅读，并且每个课程都建立在前面课程的基础之上。 本书虽然没有完整地描述 Go 的所有语言特性（详见golang.org/ref/spec），但是却涵盖了其中的绝大部分特性，并且提及了面向对象设计以及并发等高级主题。

无论你是打算使用 Go 编写大型的*并发*web 服务，还是说只想用 Go 编写小型脚本和简单的工具，本书都会帮助你打下坚实的基础。



- 单元 1 将组合使用*变量*、*循环*和*分支*构建小型程序，其中包括问候程序和火箭发射器。
- 单元 2 将探索文本和数字*类型*。学习如何使用旋转 13 算法解码加密消息，调查 Arianne 5 号火箭解体的原因，并使用大整数计算光抵达仙女座星系所需的时间。
- 单元 3 将使用*函数*和*方法*模拟构建一个火星气象站，并使用温度转换程序处理传感器读数。
- 单元 4 将在展示*数组*和*映射*用法的同时地球化太阳系、统计温度出现的次数并模拟康威生命游戏。
- 单元 5 将引入一系列*面向对象*语言概念，并说明这些概念在 Go 这种独树一帜的非面向对象语言中是如何实现的。期间使用了*结构*和方法以便在火星表面自由穿梭，接着通过满足*接口*来改善输出，并在最后通过将一个结构嵌入至另一个结构来创建更大的结构。
- 单元 6 将深挖本质，研究如何使用*指针*实现修改，想办法战胜说 nil 的骑士并学习如何冷静地处理错误。
- 单元 7 引入了 Go 的*并发*原语，并在组建地鼠工厂装配线的时候，想办法让数以千计正在运行的任务能够互相通讯。

- 附录提供了练习的参考答案，但提出你自己的解答毫无疑问可以让编程变得更加有趣！

# 示例代码

为了区分代码和普通文字，所有代码都将使用 `fixed-width` 这样的等宽字体进行表示，并且很多代码清单都会使用注释以突出重要的概念。

你可以从 Manning 出版社的网页（www.manning.com/books/get-programming-with-go）里面下载所有代码清单的源代码，里面还包含了本书所有练习的答案。 你也可以通过访问以下这个 GitHub 页面来在线阅览这些源代码：github.com/nathany/getprogramming-with-go 。

尽管你可以从 GitHub 上面直接复制粘贴代码，但我们还是建议你亲手键入书中的示例代码。 通过亲手键入代码并修复其中的打字错误，然后试验这些代码，你将能够从书中收获更多知识。

# 书本论坛

Manning 出版社运营着一个私有的网络论坛，而购买《Get Programming with Go》则让你获得了自由访问该论坛的权利。 你可以在论坛上发表关于本书的评论和技术问题、分享你的练习答案，又或者向论坛上的作者和其他用户求助。 请使用你的网络浏览器指向 forums.manning.com/forums/get-programming-with-go 以访问和订阅该论坛，又或者在 forums.manning.com/ forums/about 中获得 Manning 论坛的更多信息以及行为准则。

Manning 承诺为读者提供一个场所，让每个读者都可以与其他读者以及作者产生有意义的对话。 但 Manning 并不保证作者参与讨论的程度，他们对于论坛的一切贡献都是无偿并且自愿的。 我们建议你尝试向作者提出一些有挑战性的问题以便引起他们的兴趣。 Manning 保证这个论坛以及过往讨论的存档将在本书正常销售期间一直对外开放。

# 关于作者

NATHAN YOUNGMAN （内森·扬曼）既是一位自学成才的网络开发者，也是终生学习概念的一位践行者。 他是加拿大埃德蒙顿市 Go 聚会的组织者，Canada Learning Code 机构的导师以及地鼠玩偶的狂热摄影爱好者。

ROGER PEPPÉ （罗杰·乔）是一位 Go 贡献者，他维护着一系列开源 Go 项目，运营着英国纽卡斯尔市的 Go 聚会，并且当前正在担任 Go 云端基础设施软件的相关工作。

# 单元 0：入门

根据传统，学习新编程语言的首要步骤，就是准备好相应的工具和环境，以便运行简单的“Hello World”应用。 但是通过 Go 游乐场，我们只需要点击一下鼠标就可以完成这个古老而繁琐的习俗。

在将配置环境这只拦路虎轻而易举地解决掉之后，我们就可以开始学习一些语法和概念，并使用它们来编写和修改简单的程序了。

# 第 1 章：各就各位，预备，Go！

阅读本课程能够帮助你：

- 了解 Go 与众不同的地方
- 了解如何访问 Go 游乐场
- 学会如何将文本打印到屏幕上
- 对包含任意自然语言的文本进行实验

Go 是一门为*云计算*而生的现代化编程语言。 根据 thenewstack.io/who-is-the-go-developer/ (https://thenewstack.io/who-is-the-go-developer/) 和 golang.org/wiki/GoUsers (https://golang.org/wiki/GoUsers) 的记载， 包括亚马逊（Amazon）、苹果（Apple）、科能软件（Canonical）、雪佛龙（Chevron）、迪士尼（Disney）、脸书（Facebook）、通用电气（GE）、谷歌（Google）、Heroku、微软（Microsoft）、Twitch、威瑞森无线（Verizon） 和沃尔玛（Walmart）在内的公司都使用了 Go 来开发重要的项目，并且由于诸如 CloudFlare、Cockroach Labs、DigitalOcean、Docker、InfluxData、Iron.io、 Let’s Encrypt、Light Code Labs、Red Hat CoreOS、SendGrid 这样的公司以及云原生计算基金会（Cloud Native Computing Foundation）等组织的推动，许多 web 底层基础设施正在陆续迁移至 Go 之上。

尽管 Go 正在数据中心（data center）大放异彩，但它的应用场景并不仅限于工作区域。 比如 Ron Evans 和 Adrian Zankich 就创建了用于控制机器人和硬件的 Gobot 库（gobot.io (https://gobot.io/)），而 Alan Shreve 则创建了以学习

Go 为目的的开发工具 ngrok（ngrok.com (https://ngrok.com/)），并将该项目转变成了自己的全职事业。

为了向图 1-1 所示的那只无忧无虑的 Go 吉祥物表示敬意，社区中的 Go 拥护者通常会把自己称为 gopher （地鼠）。 虽然编程路上充满着各式各样的挑战，但通过使用 Go 并阅读本书，我们希望你能够从中发现编程的乐趣。

---

图 1-1 Renée French 设计的 Go 地鼠吉祥物

---

本课程将展示一个运行在 web 浏览器里面的 Go 程序，并基于该程序进行一些实验。

{请考虑这一点}

像英语这样的自然语言充斥着各式各样模棱两可的话。 比如说，当你向数字助理说出“Call me a cab”的时候，它是应该帮你致电出租车公司呢，还是应该假设你想要把自己的名字改成“*a cab*”呢?

清晰度对于编程语言永远都是最重要的。 假如编程语言的语法或者句法允许歧义存在，那么计算机也许就无法完成人们指定的行为，这样一来编程工作将变得毫无意义。

Go 并不是一门完美的语言，但跟我们之前使用过的其他语言相比，它在清晰度方面所做的努力是无人能及的。 在学习这一课的时候，你将会看到一些名词缩写以及行业术语。 虽然一开始你可能会对这些内容感到陌生，但我们希望你可以多花些时间，字斟句酌，仔细体会 Go 是如何减少语言中的歧义的。

# 1.1 什么是 Go？

Go 是一门*编译*语言。 在运行程序之前，Go 首先需要使用*编译器*将用户编写的代码转化为计算机能够理解的 0 和 1 。 为了便于执行和分发，编译器还会把所有代码整合并编译成一个单独的*可执行文件*。 在编译的过程中，Go 编译器能够捕捉到程序中包括拼写错误在内的一些人为失误。

并非所有编程语言都需要编译才能运行，比如 Python 、 Ruby 和其他一些流行语言就选择了在程序运行的时候，通过*解释器*一条接一条地转化代码中的声明，但这也意味着 bug 可能会隐藏在测试尚未尝触及到的代码当中。

不过换个角度来看，解释器不仅能够让开发过程变得迅速且具有交互性，还能够让语言本身变得灵活、轻松和令人愉快。 相反地，编译语言却常常因为像机器人一样顽固不化、墨守成规而广为人知，并且缓慢的编译速度也常常为人所诟病，然而实际上并非所有编译语言都是如此。

我们想要构造出这样一种语言， 它不仅可以像 C++ 和 Java 这类静态编译语言一样安全、高效， 并且还可以像 Python 这类动态类型解释语言一样身轻如燕并且充满乐趣。 —— Rob Pike，《极客周刊》（详情请见 mng.bz/jr8y (mng.bz/jr8y)）

Go 在考虑软件开发的*体验*方面可谓煞费苦心。 首先，即使是大型程序的编译也可以在极短的时间内完成，并且只需要用到一条命令。 其次，语言排除了那些可能会导致歧义的特性，鼓励可预测和简明易懂的代码。 最后，Go 为 Java 等传统语言死板的数据结构提供了轻量级的替代品。

> Java 避免了 C++ 当中许多不常见、难懂和令人迷惑的特性， 根据我们的经验， 这些特性带来的麻烦要比好处多得多。 —— James Gosling，《Java：概述》

每一种新编程语言都会对以往想法进行改良。 与早期语言相比，在 Go 里面高效地使用内存将变得更为容易，出错的可能性也更低，并且 Go 还能利用多核机器上的每个核心获得额外的性能优势。 很多成功案例都会把性能提升列举为转向 Go 的其中一个原因。 比如根据 mng.bz/Wevx (http://mng.bz/Wevx) 和 mng.bz/8yo2 (http://mng.bz/8yo2) 的记载，Iron.io (http://Iron.io) 只用了 2 台 Go 服务器就替换掉了他们原来使用的 30 台 Ruby 服务器； 而根据 mng.bz/EnYl (http://mng.bz/EnYl) 的记载，Bitly (https://bitly.com/) 在使用 Go 重写原有的 Python 应用之后也获得了持续、可测量的性能提升，这导致他们在之后把自己的 C 应用也“更新换代”成了相应的 Go 版本（详情请见 mng.bz/EnYl (https://mng.bz/EnYl)） 。

Go 不仅像解释语言一样简单和有趣，它还拥有编译语言快如闪电的性能优势以及坚如磐石的可靠性，并且由于 Go 是一门只包含几种简单概念的小型语言，所以它学习起来也相对比较快。 终上所述，我们就得出了以下 Go 箴言：

并且由于 Go 是一门只包含几种简单概念的小型语言，所以它学习起来也相对比较快。

> Go 是一门开源编程语言， 它可以大规模地生产出**简单**、**高效**并且**可信赖**的软件。 —— 《Go 品牌手册》（Go Brand Book）

> {提示}
>
> 当你在互联网上搜索 Go 的相关话题时， 可以使用关键字 golang 来代表 Go 语言。 这种将 -lang 后缀添加到语言名字之后的做法也适用于其他编程语言， 比如 Ruby 、 Rust 等。

{速查 1-1}

Go 编译器的两个优点是什么?

{速查 1-1 答案}

Go 编译器可以在极短的时间内完成对大型程序的编译， 并且它还可以在程序运行之前找出代码中的一些人为失误， 比如拼写错误等。

# 1.2 Go 游乐场

上手学习 Go 语言最快捷的方式就是使用 Go 游乐场 play.golang.org (https://play.golang.org/) ， 这个工具可以让你在无需安装任何软件的情况下直接编辑、运行和试验 Go 程序。 当你点击游乐场的运行按钮（Run）的时候，游乐场就会在 Google 的服务器上编译并运行你输入的代码， 然后在屏幕上打印出代码的执行结果。

图 1-2 Go 游乐场


点击游乐场的分享按钮（share）可以获得一个访问当前代码的链接。 你可以通过这个链接把自己的代码分享给朋友，又或者将其用作浏览器书签以便保存工作进度。

{注意}

虽然本书列出的所有代码和练习都可以通过 Go 游乐场执行， 但如果你更习惯使用自己的编辑器和命令行， 那么你可以访问 golang.org/dl/ (https://golang.org/dl/) ， 并按照页面中的指示将 Go 下载并安装至你的电脑。

{速查 1-2}

Go 游乐场里面的运行按钮是用来干什么的?

{速查 1-2 答案}

运行按钮可以在 Google 的服务器上编译并执行用户输入的代码。

# 1.3 包和函数

当我们访问 Go 游乐场的时候将会看到以下代码，它作为学习 Go 语言的起点真的再合适不过了。

代码清单 1-1 与游乐场的初次见面： `playground.go`

```
package main                                    // 声明本代码所属的包

import (
    "fmt"                                       // 导入 fmt 包，使其可用（fmt是format的缩写）
)

func main() {                                   // 声明一个名为 main 的函数
    fmt.Println("Hello, playground")            // 在屏幕上打印出“Hello, playground”
}
```

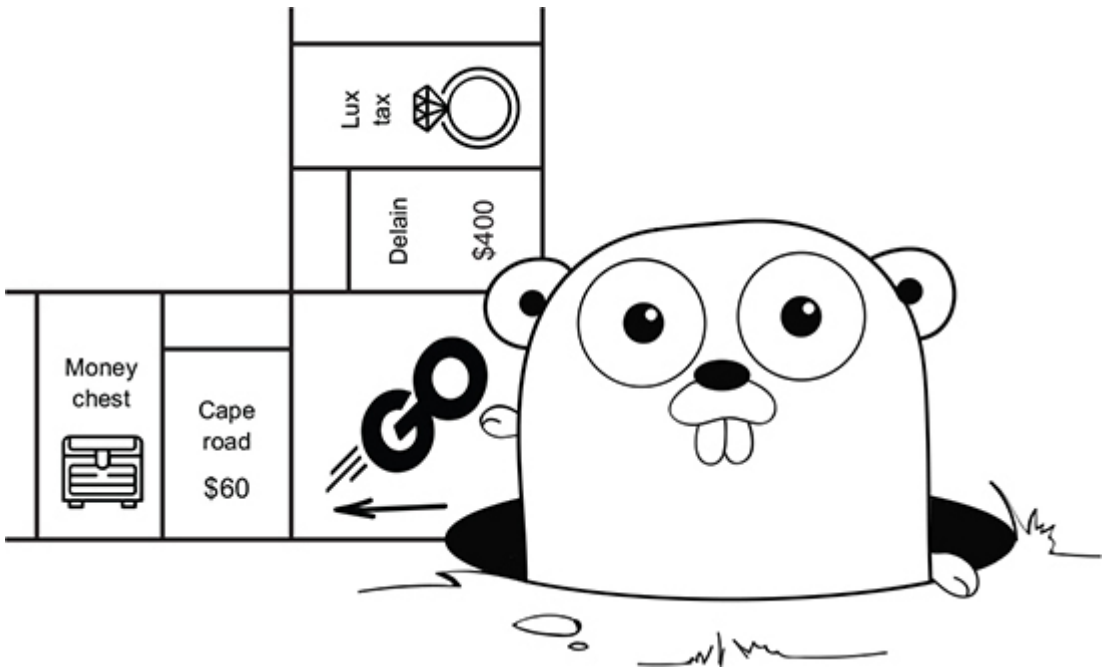
尽管这段代码非常简短，但它却引入了 `package` 、 `import` 和 `func` 这三个非常重要的关键字，这些保留关键字都有它们各自的特殊目的。

`package` 关键字声明了代码所属的包，在本例中这个包的名字就是 `main` 。所有用 Go 编写的代码都会被组织成*各式各样的包*，并且每个包都对应一个单独的构想。比如 Go 语言本身就提供了一个面向数学、压缩、加密、图像处理等领域的标准库。

在 `package` 关键字之后，代码使用了 `import` 关键字来导入自己将要用到的包。一个包可以包含任意数量的*函数*。比如 `math` 包就提供了诸如 `Sin` 、 `Cos` 、 `Tan` 和 `Sqrt` （平方根）等函数，而此处用到的 `fmt` 包则提供了用于格式化输入和输出的函数。因为在屏幕上显示文本是一个非常常用的操作，所以 Go 使用了缩写 `fmt` 作为包名。Gopher 们通常把这个包的名字读作“FŌŌMT!”，给人的感觉仿佛就像这个库是使用漫画书上的大爆炸字体撰写的一样。

`func` 关键字用于声明函数，在本例中这个函数的名字就是 `main` 。每个函数的*体*（body）都需要使用大括号 `{}` 实施包围，这样 Go 才能知道每个函数从何处开始，又在何处结束。

`main` 这一*标识符*（identifier）具有特殊意义。当我们运行一个 Go 程序的时候，它总是从 `main` 包的 `main` 函数开始运行。如果 `main` 不存在，那么 Go 编译器将报告一个错误，因为它无法得知程序应该从何处开始执行。




为了打印出一个由文本组成的*行*，例子中的代码使用了 `Println` 函数（其中 `ln` 为行的英文字母 line 的缩写）。每次用到被导入包中的某个函数时，我们都需要在函数的名字前面加上包的名字以及一个点号作为前缀。比如代码清单

中的 `Println` 函数前面就带有 `fmt` 以及一个点号作为前缀，这是因为 `Println` 函数就是由被导入的 `fmt` 包提供的。 Go 的这一特性可以让用户在阅读代码的时候立即弄清楚各个函数分别来源于哪个包。

当我们按下 Go 游乐场中的运行按钮时，代码中被引号包围的文本将输出至屏幕，最终使得文本“Hello, playground”出现在游乐场的输出区域中。 对于英语来说，即使是缺少一个逗号也有可能会让整个句子的意义变得完全不同。 同样地，标点符号对于编程语言来说也是至关重要的：比如 Go 就需要依靠引号、圆括号和大括号等符号来理解用户输入的代码。

{速查 1-3}

1. Go 程序从何处开始执行?
2. `fmt` 包提供了什么功能?

{速查 1-3 答案}

1. Go 程序从 `main` 包的 `main` 函数开始执行。
2. `fmt` 包提供了用于格式化输入和输出的函数。

# 1.4 唯一无二的大括号风格

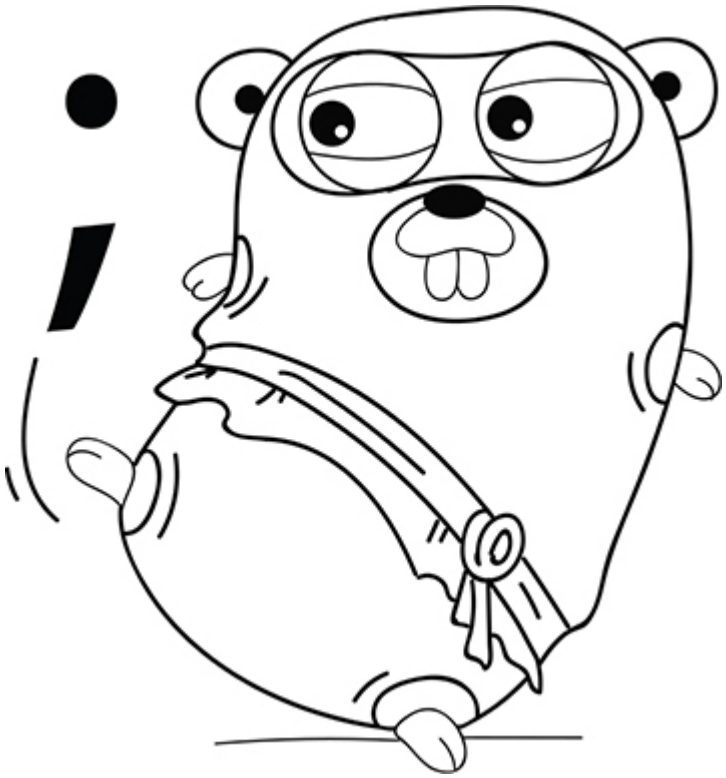
Go 对于大括号 `{}` 的摆放位置非常挑剔。 在代码清单 1-1 中，左大括号 `{` 跟 `func` 关键字位于同一行，而右大括号 `}` 则独占一行。 这是 Go 语言*唯一允许的大括号放置风格*，除此之外的其他风格都是不被允许的。 作为参考，mng.bz/NdE2 (http://mng.bz/NdE2) 展示了各种不同的大括号放置风格。

Go 之所以如此严格地限制大括号的放置风格，跟这门语言刚刚诞生时发生的一些事情有关。 在早期，使用 Go 编写的代码总是无一幸免地带有分号，它们就像迷路的小狗一样跟在每条单独的语句后面，比如这样：

```
fmt.Println("Hello, fire hydrant");
```

到了 2009 年 12 月，一群忍者 gopher 决定把分号从语言中驱逐出去。 好吧，这么说也不太准确。 实际上，Go 编译器将为你代劳，自动地插入那些可爱的分号。 这种机制运行得非常完美，但它的代价就是要求用户必须遵守上面提到

的*唯一无二的大括号放置风格*。



如果用户尝试将左大括号和 `func` 关键字放在不同的行里面，那么 Go 将报告一个语法错误：

```
func main()       // 函数体缺失
{                 // 语法错误：在 { 之前发现了意料之外的分号或新行
}
```

出现这个问题并不是编译器有意刁难，这纯粹是由于分号被插入到了错误的位置，导致编译器犯了点小迷糊，最终才不得不求助于你。

> {提示}
>
> 在阅读本书的时候， 亲力亲为、不厌其烦地键入书中展示的每段代码将是一个不错的主意。 这样一来， 如果你键入了错误的代码， 那么就会看到相应的语法错误， 但这并不是一件坏事。 能够阅读、理解并纠正代码中的错误是一种至关重要的技能， 而且坚持不懈也是一种宝贵的品质。

> {速查 1-4}
>
> 用户必须将左大括号 { 置于何处才能避免引起语法错误?

{速查 1-4 答案}

左大括号必须与 `func` 关键字位于同一行而不是独占一行， 这是 Go 语言唯一允许的大括号放置风格。

# 1.5 课后小结

- 通过 Go 游乐场，我们可以在不必安装任何东西的情况下开始使用 Go 。
- 每个 Go 程序都由包（package）中包含的函数组成。
- 为了将文本输出至屏幕，我们需要用到标准库提供的 `fmt` 包。
- 跟自然语言一样，编程语言中的标点符号也是至关重要的。
- 到目前为止，我们已经看见了 Go 25 个关键字中的 3 个，它们分别是：`package` 、 `import` 和 `func` 。

为了确认你是否已经弄懂了前面介绍的知识，请按照接下来展示的练习修改 Go 游乐场中的代码，并点击运行按钮来查看结果。 如果你在做练习的过程中遇到麻烦无法继续下去，那么可以通过刷新浏览器来让代码回到最初的样子。

## 实验：playground.go

- 修改代码中被引号包围的文本，使得程序在将文本打印至屏幕时，可以通过你的名字向你打招呼。
- 在 `main` 函数的函数体 `{}` 之内添加第二行代码，使得程序可以打印出两个文本行。就像这样：

```
fmt.Println("Hello, world")
fmt.Println("Hello, 世界")
```

- Go 支持所有自然语言字符。 你可以尝试让程序用中文、日文、俄罗斯文甚至是西班牙文打印文本。 如果你不会说上述提到的这些语言，那么可以先通过 Google 翻译工具（translate.google.com (translate.google.com)）进行翻译，然后再把翻译后的文本复制/粘贴到 Go 游乐场。

你可以通过点击 Go 游乐场中的分享按钮来获得访问当前代码的链接，然后将该链接发布至本书的论坛 forums.manning.com/forums/get-programming-with-go (forums.manning.com/forums/get-programming-with-go) 与其他读者进行分享。

最后，你可以将你的解法和附录中展示的参考答案进行对比，从而判断其是否正确。

# 单元 1： 命令式编程

跟母亲菜谱上记载的烹饪方法一样，绝大多数计算机程序都由一系列步骤组成。 程序员需要精确地告诉计算机*如何*完成任务，然后由计算机负责执行程序员的指令。 这种通过编写指令来进行编程的方式被称为*命令式*编程，它就像是在教计算机如何做菜一样！

在这一单元，我们将会了解到一些 Go 的基本知识，并开始学习如何通过 Go 的句法去命令计算机。 单元中的各个课程将陆续介绍一些知识，这些知识将帮助我们解决单元最后提出的挑战：构建一个计算火星旅行所需费用的应用程序。

# 第 2 章： 被美化的计算器

阅读本课程能够帮助你：

- 学会如何让计算机执行算术操作
- 学会如何声明变量和常量
- 了解声明和赋值的区别
- 学会如何使用标准库生成伪随机数

计算机程序能够胜任相当多的工作。 作为例子，本课程将展示如何使用程序去解决数学问题。

> {请考虑这一点}
>
> 我们为什么要编写程序来做那些只需要按一下计算器就能够解决的问题呢?
>
> 首先，人类的记性通常都不太好，可能无法精确地记下光的速度又或者火星沿着轨道环绕太阳一周所需的时间，而程序和计算机就没有这个问题。 其次，代码可以保存起来以供之后阅读，它既是一个计算器也是一份参考说明。 最后，程序是可执行文件，人们可以随时根据自己的需要来共享和修改它。

## 2.1 执行计算

人们总是希望自己能够变得更年轻和更苗条，如果你也有同样的想法，那么火星应该能满足你的愿望。 火星上的一年相当于地球的 687 天，而相对较弱的地心引力则使得同一物体在火星上的重量只有地球上的 38% 。

为了计算本书作者 Nathan 在火星上的年龄和体重，我们写下了代码清单 2-1 所示的小程序。 Go 跟其他编程语言一样，提供了 `+` 、 `-` 、 `*` 、 `/` 和 `%` 等算术*操作符*，并将它们分别用于执行加法、减法、乘法、除法和取模运算。

> {提示}
>
> 取模运算符 `%` 能够计算出两个整数相除所得的余数。 比如说， `42 % 10` 的结果为 `2` 。

---

代码清单 2-1 你好，火星： `mars.go`

```
// My weight loss program.                                  // 为人类读者提供的注释
package main

import "fmt"

// main is the function where it all begins.                // 为人类读者提供的注释
func main() {
    fmt.Print("My weight on the surface of Mars is ")
    fmt.Print(149.0 * 0.3783)                               // 打印出 56.3667
    fmt.Print(" lbs, and I would be ")
    fmt.Print(41 * 365 / 687)                               // 打印出 21
    fmt.Print(" years old.")
}
```

{注意}

虽然代码清单 2-1 会以英镑为单位打印体重，但计量单位的选择对于体重的计算并无影响。无论你使用的是什么计量单位，火星上的重量都只相当于地球上重量的 38% 。

这段代码的第一行为注释。当 Go 在代码里面发现双斜线 `//` 的时候，它会忽略双斜线之后直到行尾为止的所有内容。计算机编程的本质就是交流，好的代码不仅能够把程序员的指令传达给计算机，还能够把程序员的意图传达给其他阅读代码的人。注释的存在纯粹是为了帮助人类理解代码的意图，它不会对程序的行为产生任何影响。

上面的代码清单会调用 `Print` 函数好几次，以便将完整的句子打印在同一行里面。达到这一目的的另一种方法是调用 `Println` 函数，并向它传递一组由逗号分隔的*参数*，这些参数可以是文本、数字又或者算术*表达式*：

```
fmt.Println("My weight on the surface of Mars is", 149.0*0.3783, "lbs, and I wou

 // 这个函数调用将打印出句子“My weight on the surface of Mars is 56.3667 lbs, and I w
```

{速查 2-1}

请在 Go 游乐场 play.golang.org 中输入并运行代码清单 2-1 ，然后将作者的年龄（41）以及重量（149.0）替换成自身的年龄和重量，看看你在火星上的年龄和重量是多少?

{速查 2-1 答案}

这个问题没有标准答案， 程序的具体输出取决于你输入的重量和年龄。

{提示}

在修改代码之后，请点击 Go 游乐场中的“格式化（Format）”按钮。 这样游乐场就会在不改变代码行为的前提下，自动重新格式化代码的缩进和空白。

# 2.2 格式化输出

通过使用代码清单 2-2 中展示的 `Printf` 函数，用户可以在文本中的任何位置插入给定的值。 `Printf` 函数跟 `Print` 函数以及 `Println` 函数同属一门，但前者对输出拥有更大的控制权。

代码清单 2-2 Printf: `fmt.go`

```
fmt.Printf("My weight on the surface of Mars is %v lbs,", 149.0*0.3783) // 打印出“

fmt.Printf(" and I would be %v years old.\n", 41*365/687)              // 打印出“
```

跟 `Print` 和 `Println` 不一样的是， `Printf` 接受的第一个参数总是文本，第二个参数则是表达式，而文本中包含的*格式化变量* `%v` 则会在之后被替换成表达式的*值*。

{注意}

之后的课程将按需介绍更多除 `%v` 之外的其他格式化变量， 你也可以通过查看文档 golang.org/pkg/fmt/ 得到完整的格式化变量参考列表。

虽然 `Println` 会自动将输出的内容推进至下一行，但 `Printf` 和 `Print` 却不会那么做。 对于后面这两个函数，用户可以通过在文本里面放置换行符 `\n` 来将输出内容推进至下一行。

如果用户给定了多个格式化变量，那么 Printf 函数将按顺序把它们替换成相应的值：

```
fmt.Printf("My weight on the surface of %v is %v lbs.\n", "Earth", 149.0)

 // 打印出“My weight on the surface of Earth is 149 lbs.”
```

Printf 除了可以在句子的任意位置将格式化变量替换成指定的值之外，还能够调整文本的位置。比如说，用户可以通过给定带有宽度的格式化变量 %4v，将文本的长度填充至 4 个字符长。当宽度为正数时，空格将被填充至文本左边，而当宽度为负数时，空格将被填充至文本右边：

```
fmt.Printf("%-15v $%4v\n", "SpaceX", 94)
fmt.Printf("%-15v $%4v\n", "Virgin Galactic", 100)
```

上面这两行代码将打印出以下内容：

```
SpaceX          $  94
Virgin Galactic $ 100
```

{速查 2-2}

1. 如何才能打印出一个新行?
2. Printf 函数在遇到格式化变量 %v 的时候会产生何种行为?

{速查 2-2 答案}

1. 你可以在待打印文本的任意位置通过添加 \n 字符来插入新行，又或者直接调用 fmt.Println() 。
2. 格式化变量 %v 将被替换成用户在后续参数中给定的值。

# 2.3 常量和变量

代码清单 2-1 中的计数器在计算时使用了类似 0.3783 这样的*字面*数字，但并没有具体说明这些数字所代表的的含义，程序员有时候会把这种没有说明具体含义的字面数字称之为*魔数*。 通过使用常量和变量并为字面数字赋予描述性的名称，我们可以有效地减少魔数的存在。

在了解过居住在火星对于年龄和体重有何种好处之后，我们接下来要考虑的就是旅行所需消耗的时长。 对于我们的旅程来说，光速旅行是最为理想的。 因为光在太空的真空环境中会以*固定*速度传播，所以相应的计算将会变得较为简单。 与此相反的是，根据行星在太阳轨道上所处的位置不同，地球和火星之间的距离将会产生相当大的*变化*。

代码清单 2-3 引入了两个新的关键字 `const` 和 `var` ，它们分别用于声明常量和变量。

---

代码清单 2-3 实现光速旅行： `lightspeed.go`

```
// How long does it take to get to Mars?
package main

import "fmt"

func main() {
    const lightSpeed = 299792 // km/s
    var distance = 56000000   // km

    fmt.Println(distance/lightSpeed, "seconds") // 打印出“186 seconds”

    distance = 401000000
    fmt.Println(distance/lightSpeed, "seconds") // 打印出“1337 seconds”
 }
```

---

只要将代码清单 2-3 中的代码输入至 Go 游乐场然后点击运行按钮，我们就可以计算出从地球出发到火星所需的时间了。 能够以光速前进是一件非常便捷的事情，不消一会儿工夫你就到达了目的地，你甚至都不会听到有人抱怨说“我们怎么还没到？”。

这段代码的第一次计算通过声明 `distance` 变量并为其赋予初始值 56,000,000 公里来模拟火星与地球相邻时的情形，而在进行第二次计算的时候，则通过为 `distance` 变量赋予新值 401,000,000 公里来模拟火星和地球分列太阳两侧时的情形（其中 401,000,000 公里代表的是火星和地球之间的直线距离）。

{注意}

`lightSpeed` 常量是不能被修改的， 尝试为其赋予新值将导致 Go 编译器报告错误：“无法对 `lightSpeed` 进行赋值”。

{注意}

变量必须先声明后使用。 如果变量尚未使用 `var` 关键字进行声明， 那么尝试向它赋值将导致 Go 报告错误， 比如在前面的代码中执行 `speed = 16` 就会这样。 这一限制有助于发现类似“想要向 `distance` 赋值结果却键入了 `distence` ”这样的问题。

{速查 2-3}

1. 尽管 SpaceX 公司的星际运输系统因为缺少曲速引擎而无法以光速行进，但它仍然能够以每小时 100,800 公里这一可观的速度驶向火星。 如果这个雄心勃勃的公司在 2025 年的一月份， 也即是地球和火星之间相距 96,300,000 公里远的时候发射宇宙飞船， 那么它需要用多少天才能够到达火星？ 请修改代码清单 2-3 来计算并回答这一问题。
2. 在地球上， 一天总共有 24 个小时。 如果要在程序中为数字 24 指定一个描述性的名字， 你会用什么关键字?

{速查 2-3 答案}

1. 虽然宇宙飞船在实际中不可能只沿着直线行进， 但作为一个粗略的估计， 它从地球飞行至火星大约需要用 39 天。 以下是进行计算所需修改的代码：

```
const hoursPerDay = 24
var speed = 100800      // 公里/小时
var distance = 96300000 // 公里
fmt.Println(distance/speed/hoursPerDay, "days")
```

2. 因为地球一天拥有的小时数不太可能在程序运行的过程中发生变化， 所以我们可以使用 `const` 关键字来定义它。

# 2.4 走捷径

虽然访问火星也许没有捷径可走，但 Go 却提供了一些能够让我们少打些字的快捷方式。

## 2.4.1 一次声明多个变量

用户在声明变量或者常量的时候，既可以在每一行中单独声明一个变量：

```
var distance = 56000000
var speed = 100800
```

也可以打包声明多个变量：

```
var (
     distance = 56000000
     speed = 100800
)
```

又或者在同一行中声明多个变量：

```
var distance, speed = 56000000, 100800
```

需要注意的是，为了保证代码的可读性，我们在打包声明多个变量或者将多个变量放在同一行之前，应该先考虑这些变量是否相关。

> {速查 2-4}
>
> 请在只使用一行代码的情况下， 同时定义出一天的小时数以及每小时的分钟数。

> {速查 2-4 答案}
>
> ```
> const hoursPerDay, minutesPerHour = 24, 60
> ```

## 2.4.2 增量并赋值操作符

有几种快捷方式可以让我们在赋值的同时执行一些操作。比如以下代码清单中的最后两行就是等效的：

代码清单 2-4 赋值操作符： `shortcut.go`

```
var weight = 149.0
weight = weight * 0.3783
weight *= 0.3783
```

Go 为加一操作提供了额外的快捷方式，它们的执行方式如下：

代码清单 2-5 加法操作符

```
var age = 41
age = age + 1   // 生日快乐!
age += 1
age++
```

用户可以使用 `count--` 执行减一操作，又或者使用类似 `price /= 2` 这样的方式执行其他常见的算术运算。

> {注意}
>
> 顺带一提， Go 并不支持 `++count` 这种见诸于 C 和 JAVA 等语言中的前置加法操作。

> {速查 2-5}
>
> 请用最简短的代码实现“从 `weight` 变量中减去两磅”这一操作。

> {速查 2-5 答案}
>
> ```
> weight -= 2
> ```

# 2.5 数字游戏

让人类随意想出一个介于 1 至 10 之间的数字是非常容易的，但如果我们想要让 Go 来完成同样的事情，那么就需要用到 `rand` 包来生成伪随机数。 这些数字之所以被称为伪随机数，是因为它们并非真正随机，只是看上去像是随机的而已。

代码清单 2-6 中的代码会打印出两个介于 1 至 10 之间的数字。 这个程序会先向 `Intn` 函数传入数字 `10` 以获得一个介于 0 至 9 之间的伪随机数，然后把这个数字加一并将其结果赋值给变量 `num` 。 因为常量无法使用函数调用的结果作为值，所以 `num` 被声明成了变量而不是常量。

> {注意}
>
> 如果我们在写代码的时候忘记对伪随机数执行加一操作， 那么程序将返回一个介于 0 至 9 的数字而不是我们想要的介于 1 至 10 的数字。 这是典型的计算机编程错误差一错误（off-by-one error）的其中一个例子。
>
> 这种错误是典型的计算机编程错误之一。

代码清单 2-6 随机数字： `rand.go`

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    var num = rand.Intn(10) + 1
    fmt.Println(num)

    num = rand.Intn(10) + 1
    fmt.Println(num)
}
```

虽然 rand 包的*导入路径*为 math/rand ，但是我们在调用 Intn 函数的时候只需要使用包名 rand 作为前缀即可，不需要使用整个导入路径。

{提示}

从原则上来讲， 我们在使用某个包之前必须先通过 import 关键字导入该包， 但是贴心的 Go 游乐场也可以在有需要的时候自动为我们添加所需的导入语句。 为此， 你需要确保游乐场中的“Imports”（导入）点击框已经处于打勾状态， 并点击游乐场中的“Format”（格式化）按钮。 这样一来， Go 游乐场就会找出程序正在使用的包， 然后更新代码以添加相应的导入语句。

{注意}

因为 Go 游乐场会把每个程序的执行结果都缓存起来， 所以即使我们重复执行代码清单 2-6 所示的程序， 最终也只会得到相同的结果， 不过能够做到这一点已经足以验证我们的想法了。

{速查 2-6}

地球和火星相邻时的距离和它们分列太阳两侧时的距离是完全不同的。 请编写一个程序， 它能够随机地生成一个介于 56,000,000 公里至 401,000,000 公里之间的距离。

{速查 2-6 答案}

```
// 随机地产生一个从地球到火星之间的距离（以公里为单位）
var distance = rand.Intn(345000001) + 56000000
fmt.Println(distance)
```

# 2.6 课后小结

- Print 、 Println 和 Printf 函数都可以将文本和数字打印到屏幕上。
- 通过 Printf 函数和格式化变量 %v ，用户可以将值放置到被打印文本的任意位置上。
- const 关键字声明的是常量，它们无法被改变。

- `var` 关键字声明的是变量，它们可以在程序运行的过程中被赋予新值。
- `rand` 包的导入路径为 `math/rand` 。
- `rand` 包中的 `Intn` 函数可以生成伪随机数。
- 到目前为止，我们已经使用了 Go 25 个关键字中的 5 个，它们分别是：`package` 、 `import` 、 `func` 、 `const` 和 `var` 。

为了检验你是否已经掌握了上述知识，请尝试完成以下实验。

## 实验： `malacandra.go`

> Malacandra 并不遥远， 我们大约只需要二十八天就可以到达那里。 —— C.S.Lewis ，《沉寂的星球》

*Malacandra* 是 C.S.Lewis 在《太空三部曲》中为火星取的别名。 请编写一个程序，计算出在距离为 56,000,000 公里的情况下，宇宙飞船需要以每小时多少公里的速度飞行才能够只用 28 天就到达 Malacandra 。

请将你的解答和附录中列出的参考答案进行对比。

# 第 3 章：循环和分支

阅读本课程能够帮助你：

- 学会如何让计算机通过 `if` 和 `switch` 做选择
- 学会使用 `for` 重复执行指定的代码
- 学会基于条件实现循环和分支处理

计算机程序很少能够像小说那样从开头一直读到结尾，它们更像是那种能够自选结局的故事书和交互小说，后者可以基于特定条件选择不同路径，又或者重复相同步骤直到满足指定条件为止。

如果你对 `if` 、 `else` 以及 `for` 这三个见诸于多种编程语言的关键字已经非常熟悉，那么可以把本课程看作是 Go 语法的快速简介。

> {请考虑这一点}
>
> 作者 Nathan 年轻时跟家人一起长途旅行的时候， 会一起玩“二十个问题”（Twenty Questions）游戏来消磨时间： 一个人心里要想着一样东西， 而其他人则通过提问的方式来猜测这个东西是什么， 并且被提问的人只能回答“是”或者“不是”。 类似“它有多大？”这样的问题是无法回答的， 更常见的问法是“它比烤面包机要大吗？”。
>
> 计算机程序同样基于是/否问题执行操作。 对于类似“是否比烤面包机要大”这样的条件， 计算机处理器要么继续执行后续步骤， 要么通过 `JMP` 指令跳转至程序的其他位置， 至于复杂的决策则会被分解成多个更小和更简单的条件。
>
> 以你今天所穿的衣服为例， 你是如何挑选每一件衣服的呢， 其中又取决于哪些因素？ 你是根据天气预报、当天的活动计划、衣服是否完好或者是否潮流来挑选衣服的吗， 又或者你只是随心所欲地挑选了一套， 根本没考虑那么多？ 如果你要写一个程序来决定早上如何穿衣打扮的话， 那么你会提出哪些只能回答“是”或者“否”的问题呢？

## 3.1 真或假

在阅读能够自选结局的故事书时，你会碰到类似以下这样的选择：

> 如果你选择走出洞穴， 那么请翻到第 21 页。 —— Edward Packard，《时间的洞穴》

在 Go 中，诸如“是否走出洞穴”这样的问题可以用 `ture` 和 `false` 这两个预定义常量来回答。 你可以像这样使用这两个常量：

```
var walkOutside = true
var takeTheBluePill = false
```

> {注意}
>
> 某些编程语言对于“真”的定义比较宽松。 比如 Python 和 JavaScript 就把空文本 `""` 和数字零看作是“假”， 但是 Ruby 和 Elixir 却把这两个值看作是“真”。 对于 Go 来说， `true` 是唯一的真值， 而 `flase` 则是唯一的假值。

为了纪念 19 世纪时的数学家乔治·布尔，我们把“真”和“假”称为*布尔*值。 Go 的标准库里面有好些函数都会返回布尔值。 比如在接下来的代码清单 3-1 里面，程序就使用了 `strings` 包中的 `Contains` 函数来检查 `command` 变量是否包含单词“outside”，并且由于这一问题的答案为真，所以函数将返回 `true` 作为结果。

---

代码清单 3-1 返回布尔值的函数： `contains.go`

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println("You find yourself in a dimly lit cavern.")

    var command = "walk outside"
    var exit = strings.Contains(command, "outside")

    fmt.Println("You leave the cave:", exit)    // 打印文本“You leave the cave: tr
 }
```

{速查 3-1}

1. 如果洞穴里面出现了令人目眩的正午阳光， 你该如何声明名为 `wearShades` 的布尔变量呢?
2. 如果洞穴入口附近有一个指示牌，

你该如何判断 `command` 变量是否包含单词 `"read"` 呢?

{速查 3-2 答案}

1. `var wearShades = true`
2. `var read = strings.Contains(command, "read")`

# 3.2 比较

比较两个值是得出 `true` 或者 `false` 的另一种方式。 Go 提供了表 3-1 所示的比较运算符。

表 3-1 比较运算符

| | |
|---|---|
| **== 相等** | **!= 不相等** |
| < 小于 | > 大于 |
| <= 小于等于 | >= 大于等于 |

表 3-1 中的运算符既可以比较文本，又可以比较数字。 比如下面的代码清单 3-2 就展示了一个比较数字的例子。

代码清单 3-2 比较数字： `compare.go`

```
fmt.Println("There is a sign near the entrance that reads 'No Minors'.")

var age = 41
var minor = age < 18

fmt.Printf("At age %v, am I a minor? %v\n", age, minor)
```

运行这个代码清单，我们将得到以下输出：

```
There is a sign near the entrance that reads 'No Minors'.
At age 41, am I a minor? false
```

{注意}

JavaScript 和 PHP 都提供了特殊的三等号（*threequals*）运算符来实现严格的相等性检查。 在这些语言中， 宽松检查 `"1" == 1` 的结果为真， 而严格检查 `"1" === 1` 的结果则为假。 Go 只提供了一个相等运算符， 并且它不允许直接比较文本和数字。 本书将在第 10 课演示如何将数字转换为文本， 以及如何将文本转换为数字。

{速查 3-2}

`"apple"` 和 `"banana"` 这两个单词， 哪个更大一些?

{速查 3-2 答案}

因为语句 `fmt.Println("apple" > "banana")` 的执行结果为 `false` ， 所以单词 `"banana"` 比单词 `"apple"` 要大。

# 3.2 使用 `if` 实现分支判断

正如代码清单 3-3 所示，计算机可以使用布尔值或者比较条件，在 `if` 语句中选择不同的执行路径。

代码清单 3-3 分支： `if.go`

```
package main

import "fmt"

func main() {
    var command = "go east"

    if command == "go east" {           // 检查命令是否为“go east”
         fmt.Println("You head further up the mountain.")
    } else if command == "go inside" {  // 在第一次检查为假之后，检查命令是否为“go insi
         fmt.Println("You enter the cave where you live out the rest of your life
    } else {                            // 如果前两次检查都为假，那么执行第三个分支
         fmt.Println("Didn't quite get that.")
    }
}
```

执行代码清单 3-3，我们将得到以下输出：

```
You head further up the mountain.
```

`else if` 语句和 `else` 语句都是可选的。当有多个分支路径可选时，你可以重复使用 `else if` 直到满足需要为止。

> {注意}
>
> 如果你张冠李戴地误用了赋值操作符 `=` 来代替相等运算符 `==`，那么 Go 将报告一个错误。

> {速查 3-3}
>
> 冒险游戏通常会使用不同的空间作为场景。请编写一个程序，它使用 `if` 和 `else if` 分别打印出洞穴、入口和大山这三个空间的描述。在编写程序的过程中，请记得一定要像代码清单 3-3 那样，根据 Go 唯一合法的括号风格来放置大括号 `{}`。

{速查 3-3 答案}

```
package main
import "fmt"
func main() {
	var room = "cave"
	if room == "cave" {
		fmt.Println("You find yourself in a dimly lit cavern.")
	} else if room == "entrance" {
		fmt.Println("There is a cavern entrance here and a path to th
	} else if room == "mountain" {
		fmt.Println("There is a cliff here. A path leads west down th
	} else {
	    fmt.Println("Everything is white.")
	}
}
```

# 3.4 逻辑运算符

在 Go 中，逻辑运算符 `||` 代表“*逻辑或*”，而逻辑运算符 `&&` 则代表“*逻辑与*”。这些逻辑运算符可以一次检查多个条件，图 3-1 和 3-2 展示了它们的求值方式。

图 3-1 逻辑或：当 `a` 、 `b` 两值中至少有一个为真时， `a || b` 为真

| | false | true |
|---|---|---|
| false | false | true |
| true | true | true |

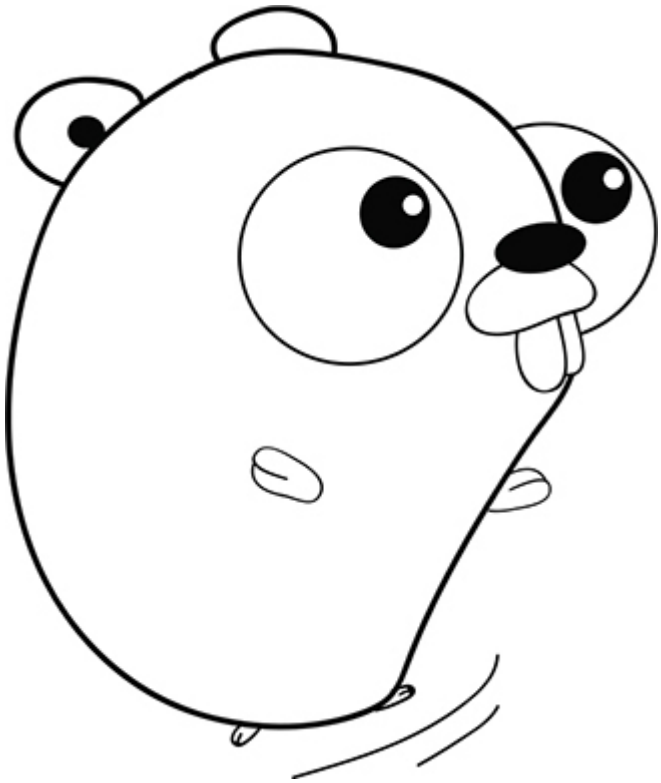
图 3-2 逻辑与：当且仅当 `a` 、 `b` 两值都为真时， `a && b` 为真

| | false | true |
|---|---|---|
| false | false | false |
| true | false | true |

代码清单 3-4 展示的是一段判断 2100 年是否为闰年的程序，其中用到的判断指定年份是否为闰年的规则如下：

- 能够被 4 整除但是不能被 100 整除的年份为闰年
- 可以被 400 整除的年份也是闰年

> {注意}
>
> 正如之前所说， 取模运算符 `%` 可以计算出两个整数相除时所得的余数， 而余数为零则表示一个数被另一个数整除了。



代码清单 3-4 闰年识别器： `leap.go`

```
fmt.Println("The year is 2100, should you leap?")

var year = 2100
var leap = year%400 == 0 || (year%4 == 0 && year%100 != 0)

if leap {
    fmt.Println("Look before you leap!")
} else {
    fmt.Println("Keep your feet on the ground.")
}
```

执行代码清单 3-4 中的程序将得到以下输出：

```
The year is 2100, should you leap?
Keep your feet on the ground.
```

跟大多数编程语言一样，Go 也采用了*短路逻辑*： 如果位于 || 运算符之前的第一个条件为真，那么位于运算符之后的条件就可以被忽略，没有必要再对其进行求值。 具体到代码清单 3-4 中的例子，当给定年份可以被 400 整除时，程序就不必再进行后续的判断了。

&& 运算符的行为跟 || 运算符正好相反： 只有在两个条件都为真的情况下，运算结果才会为真。 对于代码清单 3-4 中的例子，如果给定年份无法被 4 整除，那么程序就不会求值后续条件。

*逻辑非*运算符 ！ 可以将一个布尔值从 false 转变为 true ， 又或者将 true 转变为 false 。 作为例子，代码清单 3-5 将在玩家没有火把或者未点燃火把时打印出一条信息。

代码清单 3-5 逻辑非运算符： torch.go

```
var haveTorch = true
var litTorch = false

if !haveTorch || !litTorch {
    fmt.Println("Nothing to see here.") // 打印出“Nothing to see here.”
}
```

{速查 3-4}

1. 首先请使用纸和笔， 将代码清单 3-4 中闰年表达式的年份替换为 `2000` ； 接着求值所有取模运算， 计算出它们的余数（如果有需要可以使用计算器）； 在此之后， 求值 `==` 和 `!=` 条件以得出 `true` 或者 `false` ； 最后， 求值逻辑运算符 `&&` 和 `||` ， 并最终判断出 2000 年是否为闰年。
2. 如果我们在求值 `2000%400 == 0` 为 `true` 时使用短路逻辑， 是不是就可以节省一些时间了？ —

{速查 3-4 答案}

1. 是的， 2000 年的确是闰年：

```
2000%400 == 0 || (2000%4 == 0 && 2000%100 != 0) 0 == 0 || (0 == 0 && 0
true || (true && false)
true || (false)
true
```

2. 是的， 计算并写下等式的后半部分需要花费额外的时间。 虽然计算机执行相同计算的速度要快得多， 但短路逻辑仍然能够起到节约时间的作用。

# 3.5 使用 switch 实现分支判断

正如代码清单 3-6 所示，Go 提供了 `switch` 语句，它可以将一个值和多个值进行比较。

---

代码清单 3-6 `switch` 语句： `concise-switch.go`

```
fmt.Println("There is a cavern entrance here and a path to the east.")
var command = "go inside"

switch command {    // 将命令和给定的多个分支进行比较
case "go east":
    fmt.Println("You head further up the mountain.")
case "enter cave", "go inside":     // 使用逗号分隔可选值
    fmt.Println("You find yourself in a dimly lit cavern.")
case "read sign":
    fmt.Println("The sign reads 'No Minors'.")
default:
    fmt.Println("Didn't quite get that.")
}
```

执行这个程序将产生以下输出：

```
There is a cavern entrance here and a path to the east.
You find yourself in a dimly lit cavern.
```

> {注意}
>
> 除了文字以外， switch 语句还可以接受数字作为条件。

switch 的另一种用法是像 if...else 那样，在每个分支中单独设置比较条件。 正如代码清单 3-7 所示， switch 还拥有独特的 fallthrough 关键字，它可以用于执行下一分支的代码。

代码清单 3-7 switch 语句： switch.go

```
var room = "lake"

switch {    // 比较表达式将被放置到单独的分支里面。
case room == "cave":
    fmt.Println("You find yourself in a dimly lit cavern.")
case room == "lake":
    fmt.Println("The ice seems solid enough.")
    fallthrough    // 下降（fall through）至下一分支。
case room == "underwater":
    fmt.Println("The water is freezing cold.")
}
```

执行这段代码将产生以下输出：

```
The ice seems solid enough.
The water is freezing cold.
```

{注意}

在 C、Java、JavaScript 等语言中， 下降是 switch 语句各个分支的默认行为。 Go 对此采取了更为谨慎的做法， 用户需要显式地使用 fallthrough 关键字才会引发下降。

{速查 3-5}

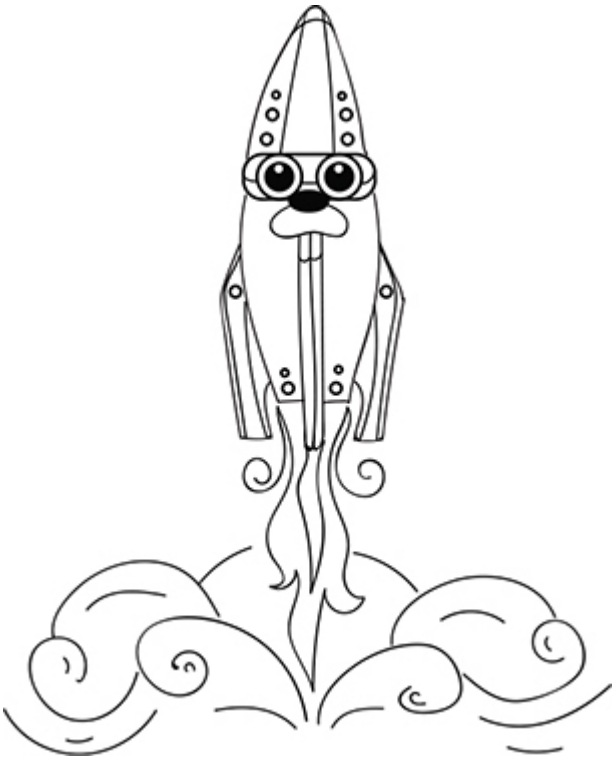
请修改代码清单 3-7 ， 通过将 room 设置为每个分支的比较对象， 让 swtich 语句能够以更为紧凑的形式出现。

{速查 3-5 答案}

```
switch room {
case "cave":
      fmt.Println("You find yourself in a dimly lit cavern.")
case "lake":
      fmt.Println("The ice seems solid enough.")
      fallthrough
case "underwater":
      fmt.Println("The water is freezing cold.")
 }
```

# 3.6 使用循环实现重复执行

当你需要重复执行同一段代码的时候，比起一遍又一遍键入相同的代码，更好的办法是使用 for 关键字。 比如代码清单 3-8 就展示了如何重复执行同一段代码直到 count 变量的值等于 0 。



代码清单 3-8 倒数循环： countdown.go

```
package main

import (
    "fmt"
    "time"
)

func main() {
    var count = 10              // 声明并初始化

    for count > 0 {             // 为循环设置条件
        fmt.Println(count)
        time.Sleep(time.Second)
        count--                 // 每次循环之后将计数器的值减一，以免产生无限循环
    }
    fmt.Println("Liftoff!")
}
```

在每次迭代开始之前，表达式 `count > 0` 都会被求值并产生一个布尔值： 当该值为 `false` 也即是 `count` 变量等于 `0` 的时候，循环就会停止； 反之，如果布尔值为真，那么程序将继续执行循环的体（body），也即是被 `{` 和 `}` 包裹的那部分代码。

此外，我们还可以通过不为 `for` 语句设置任何条件来产生*无限*循环，然后在有需要的时候通过在循环体内使用 `break` 语句来跳出循环。比如接下来的代码清单 3-9 就会持续地进行 360° 旋转，直到随机触发停止条件为止。

---

代码清单 3-9 超越无限： `infinity.go`

```
var degrees = 0

for {
    fmt.Println(degrees)

    degrees++
    if degrees >= 360 {
        degrees = 0
        if rand.Intn(2) == 0 {
            break
        }
    }
}
```

---

> {注意}
>
> 之后的第 4 课和第 9 课将介绍 `for` 循环的更多不同形式。

> {速查 3-6}
>
> 火箭的发射过程并非总是一帆风顺。请实现一个火箭发射倒数程序，它在倒数过程中的每一秒钟都伴随着百分之一的几率会发射失败并停止倒数。

{速查 3-6 答案}

```
var count = 10
for count > 0 {
      fmt.Println(count)
      time.Sleep(time.Second)
      if rand.Intn(100) == 0 {
              break
    }
      count--
}
if count == 0 {
      fmt.Println("Liftoff!")
} else {
      fmt.Println("Launch failed.")
}
```

# 3.7 课后小结

- 布尔值是唯一可以用于条件判断的值。
- Go 通过 `if` 、 `switch` 和 `for` 来实现分支判断和重复执行。
- 直到目前为止，我们已经使用了 Go 25 个关键字中的 12 个，它们分别是： `package` 、 `import` 、 `func` 、 `var` 、 `if` 、 `else` 、 `switch` 、 `case` 、 `default` 、 `fallthrough` 、 `for` 还有 `break` 。

为了检验你是否已经掌握了上述知识，请尝试完成以下实验。

## 实验： `guess.go`

请编写一个“猜数字”程序，让它重复地从 1 至 100 之间随机选择一个数字，直到这个数字跟你在程序开头声明的数字相同为止。 请打印出程序随机选中的每个数字，并说明该数字是大于还是小于你指定的数字。

# 第 4 章：变量作用域

阅读本课程能够帮助你：

- 知悉变量作用域的好处
- 学会用更简洁的方式声明变量
- 了解到 `for` 、 `if` 和 `switch` 是如何与变量作用域相互协作的
- 学会如何控制作用域的范围

在程序运行的过程中，很多变量都会在短暂使用之后被丢弃，这是由编程语言的*作用域规则*决定的。

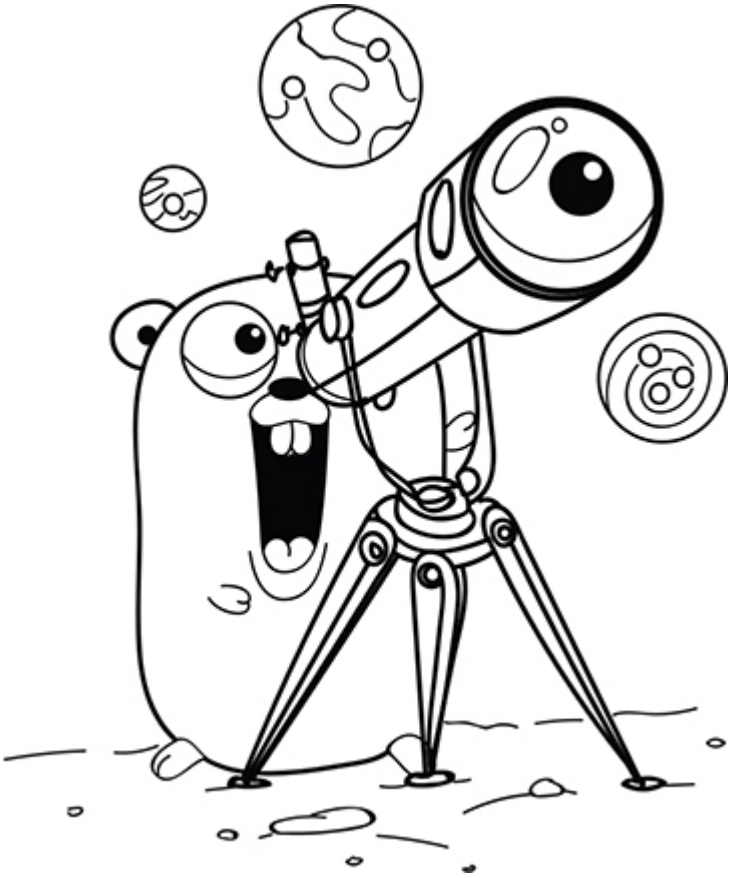
> {请考虑这一点}
>
> 你可以在脑海里面一次记住多少东西?
>
> 据说人类的短期记忆最多只能记住大概 7 样东西， 比如七位数的电话号码就是一个很好的例子。
>
> 虽然计算机的短期记忆存储器或者说随机访问存储器（RAM）可以记住大量值， 但是别忘了， 程序代码除了需要被计算机读取之外， 还需要被人类阅读， 所以它还是应该尽可能地保持简洁。
>
> 与此类似， 如果程序中的变量可以随时修改又或者在任何位置随意访问， 那么光是跟踪大型程序中的变量就足以让人手忙脚乱。 变量作用域的好处是可以让程序员聚焦于特定函数或者部分代码的相关变量， 而不需要考虑除此之外的其他变量。

## 4.1 审视作用域

变量从声明之时开始就处于*作用域*当中，换句话来说变量就是从那时开始变为可见的（visible）。只要变量仍然存在于作用域当中，程序就可以随时访问它，然而变量一旦离开作用域，尝试继续访问它将引发错误。

变量作用域的其中一个好处是让我们可以为不同的变量重用相同的名字。 因为除了极少数小型程序之外，程序的变量几乎不可能不出现重名。

除此之外，变量作用域还能够帮助我们更好地阅读代码，让我们无需在脑海里记住所有变量。 毕竟一旦某个变量离开了作用域，我们就不必再关心它了。



Go 的作用域通常会随着大括号 `{}` 的出现而开启和结束。 在接下来展示的代码清单 4-1 中， `main` 函数开启了一个作用域，而 `for` 循环则开启了一个嵌套作用域。

---

代码清单 4-1 作用域规则： `scope.go`

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    var count = 0

    for count < 10 {    // 开启新的作用域
        var num = rand.Intn(10) + 1
        fmt.Println(num)

        count++
    }   // 作用域结束
}
```

---

因为 count 变量的声明位于 main 函数的*函数作用域*之内，所以它在 main 函数结束之前将一直可见。反观 num 变量，因为它的声明位于 for 循环的作用域之内，所以它在循环结束之后便不再可见。

尝试在循环结束之后访问 num 变量将引发 Go 编译器报错。与之相对的是，因为 count 变量的声明位于 for 循环之外，所以即使在循环结束之后，程序也可以在有需要的时候继续访问 count 变量。另一方面，如果我们想要把 count 变量也限制在循环的作用域之内，那么就需要用到在 Go 中声明变量的另一种方式。

{速查 4-1}

1. 变量作用域对我们有什么好处?
2. 变量在脱离作用域之后会发生什么事情？ 请修改代码清单 4-1 ， 尝试在循环结束之后访问 num 变量， 看看会什么发生什么事情?

{速查 4-1 答案}

1. 作用域可以让我们在多个不同的地方使用相同的变量名而不会引发任何冲突， 并且在编程的时候只需要考虑位于当前作用域之内的变量即可。
2. 脱离作用域的变量将变得不可见并且无法访问。 尝试在 num 变量的作用域之外访问它将导致 Go 编译器报告以下错误： undefined: num 。

# 4.2 简短声明

简短声明为 var 关键字提供了另一种备选语法。 以下两行代码是完全等效的：

```
var count = 10
count := 10
```

初看上去，少键入两三个字符似乎不算什么，但小数怕长计，所以简短声明还是要比 var 关键字流行得多。 更重要的是，简短声明还可以用在一些 var 关键字无法使用的地方。

代码清单 4-2 展示了 for 循环的另一种形式，它包含了初始化语句、比较条件语句以及对 count 变量执行减法运算的后置语句。 在使用这种形式的 for 循环时，我们需要依次向循环提供初始化语句、比较条件语句和后置语句。

---

代码清单 4-2 更简洁的倒数程序： loop.go

```
var count = 0

for count = 10; count > 0; count-- {
    fmt.Println(count)
}

fmt.Println(count)  // count 变量仍然处于作用域之内
```

---

在不使用简短声明的情况下， count 变量的声明必须放置在循环之外，这意味着 count 变量将在循环结束之后继续存在于作用域。

但是正如代码清单 4-3 所示，在使用简短声明的情况下， count 变量的声明和初始化将成为 for 循环的一部分，并且该变量将在循环结束之后脱离作用域，而尝试在循环之外访问 count 变量将导致 Go 编译器报告 undefined: count 错误。

---

代码清单 4-3 在 for 循环中使用简短声明： short-loop.go

```
for count := 10; count > 0; count-- {
    fmt.Println(count)
}    // 随着循环结束，count 变量将不再处于作用域之内。
```

> {提示}
>
> 为了代码的可读性考虑， 声明变量的位置和使用变量的位置应该尽可能地贴近。

除了 for 循环之外，简短声明还可以在 if 语句里面声明新的变量。比如代码清单 4-4 中的 num 变量就可以用在 if 语句的所有分支当中。

代码清单 4-4 在 if 语句中使用简短声明： short-if.go

```
if num := rand.Intn(3); num == 0 {
    fmt.Println("Space Adventures")
} else if num == 1 {
    fmt.Println("SpaceX")
} else {
    fmt.Println("Virgin Galactic")
}    // 随着 if 语句结束，num 变量将不再处于作用域之内。
```

正如代码清单 4-5 所示，跟 if 语言一样，简短声明也可以用在 switch 语句里面。

代码清单 4-5 在 switch 语句中使用简短声明： short-switch.go

```
switch num := rand.Intn(10); num {
case 0:
    fmt.Println("Space Adventures")
case 1:
    fmt.Println("SpaceX")
case 2:
    fmt.Println("Virgin Galactic")
default:
    fmt.Println("Random spaceline #", num)
}
```

{速查 4-5}

如果代码清单 4-4 和 4-5 不使用简短声明， 那么 num 变量的作用域将产生何种变化?

{速查 4-2 答案}

因为 if 语句、 switch 语句和 for 语句只能使用业已声明的变量， 所以在不使用简短声明的情况下， 程序只能在 if 等语句的前面声明 num 变量， 从而导致该变量在 if 等语句结束之后仍然存在于作用域。

# 4.3 作用域的范围

代码清单 4-6 展示的程序能够生成并显示一个随机的日期（这个日期也许就是我们启程去火星的日期）。 除此之外，这个程序还演示了 Go 中的几种不同的作用域，并阐明了在声明变量时考虑作用域的重要性。

---

代码清单 4-6 变量作用域规则： scope-rules.go

```
package main

import (
    "fmt"
    "math/rand"
)

var era = "AD"      // era 变量在整个包都是可用的。

func main() {
    year := 2018    // era 变量和 year 变量都处于作用域之内。

    switch month := rand.Intn(12) + 1; month {  // 变量 era 、year 和 month 都处于作
    case 2:
        day := rand.Intn(28) + 1                // 变量 era 、year 、month 和 day
        fmt.Println(era, year, month, day)
    case 4, 6, 9, 11:
        day := rand.Intn(30) + 1                // 这两个 day 变量是全新声明的变量，跟
        fmt.Println(era, year, month, day)
    default:
        day := rand.Intn(31) + 1                // 这两个 day 变量是全新声明的变量，跟
        fmt.Println(era, year, month, day)
    }   // month 变量和 day 变量不再处于作用域之内。
}       // year 变量不再处于作用域之内。
```

因为 era 变量的声明位于 main 函数之外的*包作用域*中，所以它对于 main 包中的所有函数都是可见的。

> {注意}
>
> 因为包作用域不允许使用简短声明，所以我们无法在这个作用域中使用 era := "AD" 来进行声明。

year 变量只在 main 函数中可见。如果包中还存在着其他函数，那么它们将会看见 era 变量，但是却无法看到 year 变量。*函数作用域*比包作用域要狭窄，它起始于 func 关键字，并终结于函数声明的右大括号。

month 变量在整个 switch 语句的任何位置都可见，不过一旦 switch 语句结束，month 就不再处于作用域之内了。switch 语句的作用域始于 switch 关键字，并终结于 switch 语句的右大括号。

因为 switch 的每个 case 分支都拥有自己独立的作用域，所以三个分支分别拥有三个独立的 day 变量。 在每个分支结束之后，该分支声明的 day 变量将不再处于作用域之内。 switch 分支的作用域是唯一一种无需使用大括号标识的作用域。

代码清单 4-6 中的代码距离完美还有相当远的一段距离。 变量 month 和 day 狭窄的作用域导致 Println 重复出现了三次，这种代码重复可能会引发修改，并因此导致错误。 比如说，我们可能会决定不再在每个分支中都打印 era 变量，但是却忘记了修改某个分支。 在某些情况下，出现代码重复是正常的，但这种情况通常被认为是代码的*坏味道*，需要谨慎地处理。

为了消除重复并简化代码，我们需要将代码清单 4-6 中的某些变量声明移动到范围更宽广的函数作用域中，使得这些变量可以在 switch 语句结束之后继续为程序所用。 为此，我们需要对代码实施*重构*，也即是在不改变代码行为的基础上对代码进行修改。 重构得出的代码清单 4-7 跟之前的代码行为完全相同，它们都可以选取并打印出随机的日期。

---

代码清单 4-7 重构后的随机日期选取程序： random-date.go

```
package main

import (
    "fmt"
    "math/rand"
)

var era = "AD"

func main() {
    year := 2018
    month := rand.Intn(12) + 1
    daysInMonth := 31

    switch month {
    case 2:
        daysInMonth = 28
    case 4, 6, 9, 11:
        daysInMonth = 30
    }

    day := rand.Intn(daysInMonth) + 1
    fmt.Println(era, year, month, day)
}
```

---

尽管狭窄的作用域有助于减少脑力负担，但代码清单 4-6 的例子也表明了过分约束变量将*损害*代码的可读性。 在遇到这种问题的时候，我们应该根据具体情况逐步实施重构，直到代码的可读性能够满足我们的要求为止。

> {速查 4-3}
>
> 请说出一种能够鉴别变量是否被约束得太紧的方法。

> {速查 4-3 答案}
>
> 如果代码重复是由变量声明引起的， 那么变量可能就是被约束得太紧了。

# 4.4 课后小结

- 左大括号 `{` 开启一个新的作用域而右大括号 `}` 则结束该作用域。
- 虽然没有用到大括号，但关键字 `case` 和 `default` 也都引入了新的作用域。
- 声明变量的位置决定了变量所处的作用域。
- 简短声明不仅仅是 `var` 声明的快捷方式，它还可以用在 `var` 声明无法使用的地方。
- 在 `for` 语句、 `if` 语句和 `switch` 语句所在行声明的变量，其作用域将持续至语句结束为止。
- 宽广的作用域有时候会比狭窄的作用域更好，反之亦然。

为了检验你是否已经掌握了上述知识，请尝试完成以下实验。

## 实验： `random-dates.go`

请修改代码清单 4-7 ，让它可以处理闰年：

- 生成一个随机年份而不是一直使用 `2018` 年。
- 如果生成的年份为闰年，那么将一月份的 `daysInMonth` 变量的值设置为 29 ，反之则将其设置为 28 。 提示：你可以在 `case` 代码块的内部放置 `if` 语句。

- 使用 `for` 循环生成并显示 10 个随机日期。

# 第 5 章：单元实验——前往火星的航行票

欢迎来到本书的第一个单元实验，现在是时候使用我们在本单元学习到的知识来编写程序了！ 我们的目标是在 Go 游乐场编写一个太空航行票务生成器，它需要用到变量、常量、`switch` 、 `if` 和 `for` ，并使用 `fmt` 包和 `math/rand` 包来显示文本、对齐文本以及生成随机数。

在计划以火星为目的地的旅行时，能够从一个地方获知多家太空航行公司的票价将是一件非常方便的事情。 虽然现在已经有不少聚合各大航空公司飞机票价格的网站，但目前还没有网站推出过相应的太空航行票务服务。 不过这对于我们来说并非难事，毕竟我们可以通过 Go 来让计算机解决这一问题。

为此，我们需要构建一个原型程序，它可以随机生成 10 张太空航行票，并将它们打印到格式工整、标题美观的表格里面，就像这样：

| 太空航行公司 | 飞行天数 | 飞行类型 | 价格（百万美元） |
| --- | --- | --- | --- |
| Virgin Galactic | 23 | 往返 | 96 |
| Virgin Galactic | 39 | 单程 | 37 |
| SpaceX | 31 | 单程 | 41 |
| Space Adventures | 22 | 往返 | 100 |
| Space Adventures | 22 | 单程 | 50 |
| Virgin Galactic | 30 | 往返 | 84 |
| Virgin Galactic | 24 | 往返 | 94 |
| Space Adventures | 27 | 单程 | 44 |
| Space Adventures | 28 | 往返 | 86 |
| SpaceX | 41 | 往返 | 72 |

程序打印的表格应该包含以下四列：

- 提供服务的太空航行公司
- 以天为单位，到达火星所需的单程飞行时间
- 票价是否包含返程票
- 以百万美元为单位的票价

对于每张太空航行票，从 Space Adventures、SpaceX、Virgin Galactic 这三间太空航行公司中随机选择一间作为服务商。

使用 2020 年 10 月 13 日作为所有太空航行票的出发日期，火星和地球在那一天的距离为 62,100,000 公里。

在每秒 16 公里至每秒 30 公里之间随机选择一种作为宇宙飞船的飞行速度，该速度决定了飞行的时长以及价格。 宇宙飞船每次飞行的价格从 3600 万美元到 5000 万美元不等，速度越快的航线价格也越贵，至于双程票则需要收取双倍费用。

请在完成实验之后，将你的解答发布到本书在 forums.manning.com/forums/get-programming-with-go (https://forums.manning.com/forums/get-programming-with-go) 的论坛里面。如果你在解题的过程中被难住了，那么可以随时到论坛里面请求帮助，又或者去看看本书附录提供的参考答案。

# 试读结束

《Get Programming with Go》中文版试读到此结束，感谢您的阅读!

您可以访问 GPWGCN.com (http://GPWGCN.com) 购买本书或者获取更多本书的相关信息。

---

 Back to top