MANNING

Go 语言实战

Go IN ACTION

William Kennedy
〔美〕 Brian Ketelsen 著
Erik St. Martin

李兆海 译
谢孟军 审校

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

需要整本电子书，联系我QQ：2667271557

MANNING

Go 语言实战

Go IN ACTION

William Kennedy
〔美〕 Brian Ketelsen 著
Erik St. Martin

李兆海 译

谢孟军 审校

人 民 邮 电 出 版 社

需要整本电子书，联系我QQ：2667271557

图书在版编目（CIP）数据

Go语言实战 /（美）威廉·肯尼迪（William Kennedy），（美）布赖恩·克特森（Brian Ketelsen），（美）埃里克·圣马丁（Erik St.Martin）著；李兆海译. -- 北京：人民邮电出版社，2017.3
ISBN 978-7-115-44535-3

Ⅰ. ①G… Ⅱ. ①威… ②布… ③埃… ④李… 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026934号

◆ 著　[美] William Kennedy　Brian Ketelsen　Erik St. Martin
译　李兆海
审　校　谢孟军
责任编辑　杨海玲
责任印制　焦志炜

◆ 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
北京昌平百善印刷厂印刷

◆ 开本：800×1000　1/16
印张：15.25
字数：326 千字　2017 年 3 月第 1 版
印数：1－4 000 册　2017 年 3 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-8787 号

定价：59.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

需要整本电子书，联系我QQ：2667271557

译者序

Go 语言是由谷歌公司在 2007 年开始开发的一门语言，目的是能在多核心时代高效编写网络应用程序。Go 语言的创始人 Robert Griesemer、Rob Pike 和 Ken Thompson 都是在计算机发展过程中作出过重要贡献的人。自从 2009 年 11 月正式公开发布后，Go 语言迅速席卷了整个互联网后端开发领域，其社区里不断涌现出类似 vitess、Docker、etcd、Consul 等重量级的开源项目。

在 Go 语言发布后，我就被其简洁、强大的特性所吸引，并于 2010 年开始在技术聚会上宣传 Go 语言，当时所讲的题目是《Go 语言：互联网时代的 C》。现在看来，Go 语言确实很好地解决了互联网时代开发的痛点，而且入门门槛不高，是一种上手容易、威力强大的工具。试想一下，不需要学习复杂的异步逻辑，使用习惯的顺序方法，就能实现高性能的网络服务，并充分利用系统的多个核心，这是多么美好的一件事情。

本书是国外 Go 社区多年经验积累的成果。本书默认读者已经具有一定的编程基础，希望更好地使用 Go 语言。全书以示例为基础，详细介绍了 Go 语言中的一些比较深入的话题。对于有经验的程序员来说，很容易通过学习书中的例子来解决自己实际工作中遇到的问题。辅以文字介绍，读者会对相关问题有更系统的了解和认识。翻译过程中我尽量保持了原书的叙述方法，并加强了叙述逻辑，希望读者会觉得清晰易读。

在翻译本书的过程中，感谢人民邮电出版社编辑杨海玲老师的指导和进度安排，让本书能按时与读者见面。感谢谢孟军对译稿的审校，你的润色使译文读起来流畅了很多。尤其要感谢我老婆对我的支持，感谢你能理解我出于热爱才会“匍匐”在计算机前码字。

最后，感谢读者购买此书。希望读者在探索 Go 语言的道路上，能够享受到和我一样的乐趣。

需要整本电子书，联系我QQ：2667271557

序

在计算机科学领域，提到不同寻常的人，总会有一些名字会闪现在你的脑海中。Rob Pike、Robert Griesmier 和 Ken Thompson 就是其中几个。他们 3 个人负责构建过 UNIX、Plan 9、B、Java 的 JVM HotSpot、V8、Strongtalk[①]、Sawzall、Ed、Acme 和 UTF8，此外还有很多其他的创造。在 2007 年，这 3 个人凑在一起，尝试一个伟大的想法：综合他们多年的经验，借鉴已有的语言，来创建一门与众不同的、全新的系统语言。他们随后以开源的形式发布了自己的实验成果，并将这种语言命名为“Go”。如果按照现在的路线发展下去，这门语言将是这 3 个人最有影响的一项创造。

当人们聚在一起，纯粹是为了让世界变得更好的时候，往往也是他们处于最佳状态的时候。在 2013 年，为了围绕 Go 语言构建一个更好的社区，Brian 和 Erik 联合成立了 Gopher Academy，没过多久，Bill 和其他一些有类似想法的人也加入进来。他们首先注意到，社区需要有一个地方可以在线聚集和分享素材，所以他们在 slack 创立了 Go 讨论版和 Gopher Academy 博客。随着时间的推移，社区越来越大，他们创建了世界上第一个全球 Go 语言大会——GopherCon。随着与社区更深入地交流，他们意识到还需要为广大想学习这门新语言的人提供一些资源，所以他们开始着手写一本书，就是现在你手里拿的这本书。

为 Go 社区贡献了大量的时间和精力的 3 位作者，出于对 Go 语言社区的热爱写就了这本书。我曾在 Bill、Brian 和 Erik 身边，见证了他们在不同的环境和角色（作为 Gopher Academy 博客的编辑，作为大会组织者，甚至是在他们的日常工作中，作为父亲和丈夫）下，都会认真负责地撰写和修订本书。对他们来说，这不仅仅是一本书，也是对他们心爱的语言的献礼。他们并不满足于写就一本“好”书。他们编写、审校，再写、再修改，再三推敲每页文字、每个例子、每一章，直到认为本书的内容配得上他们珍视的这门语言。

离开一门使用舒服、掌握熟练的语言，去学习一门不仅对自己来说，对整个世界来说都是全新的语言，是需要勇气的。这是一条人迹罕至，沿途充满 bug，只有少数先行者熟悉的路。这里

① 一个高性能强类型的 Smalltalk 实现。——译者注

充满了意外的错误，文档不明确或者缺失，而且缺少可以拿来即用的代码库。这是拓荒者、先锋才会选择的道路。如果你正在读这本书，那么你可能正在踏上这段旅途。

本书自始至终是为你——本书的读者精心制作的一本探索、学习和使用 Go 语言的简洁而全面的指导手册。在全世界，你也不会找到比 Bill、Brian 和 Erik 更好的导师了。我非常高兴你能开始探索 Go 语言的优点，期望能在线上和线下大会上遇到你。

Steve Francia

Go 语言开发者，Hugo、Cobra、Viper 和 SPF13-VIM 的创建人

前言

那是 2013 年 10 月，我刚刚花几个月的时间写完 GoingGo.net 博客，就接到了 Brian Ketelsen 和 Erik St. Martin 的电话。他们正在写这本书，问我是否有兴趣参与进来。我立刻抓住机会，参与到写作中。当时，作为一个 Go 语言的新手，这是我进一步了解这门语言的好机会。毕竟，与 Brian 和 Erik 一起工作、一起分享获得的知识，比我从构建博客中学到的要多得多。

完成前 4 章后，我们在 Manning 早期访问项目（MEAP）中发布了这本书。很快，我们收到了来自语言团队成员的邮件。这位成员对很多细节提供了评审意见，还附加了大量有用的知识、意见、鼓励和支持。根据这些评审意见，我们决定从头开始重写第 2 章，并对第 4 章进行了全面修订。据我们所知，对整章进行重写的情况并不少见。通过这段重写的经历，我们学会要依靠社区的帮助来完成写作，因为我们希望能立刻得到社区的支持。

自那以后，这本书就成了社区努力的成果。我们投入了大量的时间研究每一章，开发样例代码，并和社区一起评审、讨论并编辑书中的材料和代码。我们尽了最大的努力来保证本书在技术上没有错误，让代码符合通用习惯，并且使用社区认为应该有的方式来教 Go 语言。同时，我们也融入了自己的思考、自己的实践和自己的指导方式。

我们希望本书能帮你学习 Go 语言，不仅是当下，就是多年以后，你也能从本书中找到有用的东西。Brian、Erik 和我总会在线上帮助那些希望得到我们帮助的人。如果你购买了本书，谢谢你，来和我们打个招呼吧。

William Kennedy

致谢

我们花了 18 个月的时间来写本书。但是，离开下面这些人的支持，我们不可能完成这本书：我们的家人、朋友、同学、同事以及导师，整个 Go 社区，还有我们的出版商 Manning。

当你开始撰写类似的书时，你需要一位编辑。编辑不仅要分享喜悦与成就，而且要不惜一切代价，帮你渡过难关。Jennifer Stout，你才华横溢，善于指导，是很棒的朋友。感谢你这段时间的付出，尤其是在我们最需要你的时候。感谢你让这本书变成现实。还要感谢为本书的开发和出版作出贡献的 Manning 的其他人。

每个人都不可能知晓一切，所以需要社区里的人付出时间和学识。感谢 Go 社区以及所有参与本书不同阶段书稿评审并提供反馈的人。特别感谢 Adam McKay、Alex Basile、Alex Jacinto、Alex Vidal、Anjan Bacchu、Beno î t Benedetti、Bill Katz、Brian Hetro、Colin Kennedy、Doug Sparling,、Jeffrey Lim、Jesse Evans、Kevin Jackson、Mark Fisher、Matt Zulak、Paulo Pires、Peter Krey、Philipp K. Janert、Sam Zaydel 以及 Thomas O’Rourke。还要感谢 Jimmy Frasch é，他在出版前对本书书稿做了快速、准确的技术审校。

这里还需要特别感谢一些人。

Kim Shrier，从最开始就在提供评审意见，并花时间来指导我们。我们从你那里学到了很多，非常感谢。因为你，本书在技术上达到了更好的境界。

Bill Hathaway 在写书的最后一年，深入参与，并矫正了每一章。你的想法和意见非常宝贵。我们必须给予 Bill“第 9 章合著者”的头衔。没有 Bill 的参与、天赋以及努力，就没有这一章的存在。

我们还要特别感谢 Cory Jacobson、Jeffery Lim、Chetan Conikee 和 Nan Xiao 为本书持续提供了评审意见和指导，感谢 Gabriel Aszalos、Fatih Arslan、Kevin Gillette 和 Jason Waldrip 帮助评审样例代码，还要特别感谢 Steve Francia 帮我们作序，认可我们的工作。

最后，我们真诚地感谢我们的家人和朋友。为本书付出的时间和代价，总会影响到你所爱的人。

需要整本电子书，联系我QQ：2667271557

William Kennedy

我首先要感谢 Lisa，我美丽的妻子，以及我的 5 个孩子：Brianna、Melissa、Amanda、Jarrod 和 Thomas。Lisa，我知道你和孩子们有太多的日夜和周末，缺少丈夫和父亲的陪伴。感谢你让我这段时间全力投入本书的工作：我爱你们，爱你们每一个人。

我也要感谢我生意上的伙伴 Ed Gonzalez、创意经理 Erick Zelaya，以及整个 Ardan 工作室的团队。Ed，感谢你从一开始就支持我。没有你，我就无法完成本书。你不仅是生意伙伴，还是朋友和兄长：谢谢你。Erick，感谢你为我、为公司做的一切。我不确定没有你，我们还能不能做到这一切。

Brian Ketelsen

首先要感谢我的家人在我写书的这 4 年间付出的耐心。Christine、Nathan、Lauren 和 Evelyn，感谢你们在游泳时放过在旁边椅子上写作的我，感谢你们相信这本书一定会出版。

Erik St. Martin

我要感谢我的未婚妻 Abby 以及我的 3 个孩子 Halie、Wyatt 和 Allie。感谢你们对我花大量时间写书和组织会议如此耐心和理解。我非常爱你们，有你们我非常幸运。

还要感谢 Bill Kennedy 为本书付出的巨大努力，以及当我们需要他的帮助的时候，他总是立刻想办法组织 GopherCon 来满足我们的要求。还要感谢整个社区出力评审并给出一些鼓励的话。

关于本书

Go 是一门开源的编程语言，目的在于降低构建简单、可靠、高效软件的门槛。尽管这门语言借鉴了很多其他语言的思想，但是凭借自身统一和自然的表达，Go 程序在本质上完全不同于用其他语言编写的程序。Go 平衡了底层系统语言的能力，以及在现代语言中所见到的高级特性。你可以依靠 Go 语言来构建一个非常快捷、高性能且有足够控制力的编程环境。使用 Go 语言，可以写得更少，做得更多。

谁应该读这本书

本书是写给已经有一定其他语言编程经验，并且想学习 Go 语言的中级开发者的。我们写这本书的目的是，为读者提供一个专注、全面且符合语言习惯的视角。我们同时关注语言的规范和实现，涉及的内容包括语法、类型系统，并发、通道、测试以及其他一些主题。我们相信，对于刚开始学 Go 语言的人，以及想要深入了解这门语言内部实现的人来说，本书都是极佳的选择。

章节速览

本书由 9 章组成，每章内容简要描述如下。

- 第 1 章快速介绍这门语言是什么，为什么要创造这门语言，以及这门语言要解决什么问题。这一章还会简要介绍一些 Go 语言的核心概念，如并发。
- 第 2 章引导你完成一个完整的 Go 程序，并教你 Go 作为一门编程语言必须提供的特性。
- 第 3 章介绍打包的概念，以及搭建 Go 工作空间和开发环境的最佳实践。这一章还会展示如何使用 Go 语言的工具链，包括获取和构建代码。
- 第 4 章展示 Go 语言内置的类型，即数组、切片和映射。还会解释这些数据结构背后的实现和机制。
- 第 5 章详细介绍 Go 语言的类型系统，从结构体类型到具名类型，再到接口和类型嵌套。这

一章还会展示如何综合利用这些数据结构，用简单的方法来设计结构并编写复杂的软件。

- 第 6 章深入展示 Go 调度器、并发和通道是如何工作的。这一章还将介绍这个方面背后的机制。
- 第 7 章基于第 6 章的内容，展示一些实际开发中用到的并发模式。你会学到为了控制任务如何实现一个 goroutine 池，以及如何利用池来复用资源。
- 第 8 章对标准库进行探索，深入介绍 3 个包，即 `log`、`json` 和 `io`。这一章专门介绍这 3 个包之间的某些复杂关系。
- 第 9 章以如何利用测试和基准测试框架来结束全书。读者会学到如何写单元测试、表组测试以及基准测试，如何在文档中增加示例，以及如何把这些示例当作测试使用。

关于代码

本书中的所有代码都使用等宽字体表示，以便和周围的文字区分开。在很多代码清单中，代码被注释是为了说明关键概念，并且有时在正文中会用数字编号来给出对应代码的其他信息。

本书的源代码既可以在 Manning 网站（www.manning.com/books/go-in-action）上下载[①]，也可以在 GitHub（https://github.com/goinaction/code）上找到这些源代码。

读者在线

购买本书后，可以在线访问由 Manning 出版社提供的私有论坛。在这个论坛上可以对本书做评论，咨询技术问题，并得到作者或其他读者的帮助。通过浏览器访问 www.manning.com/books/go-in-action 可以访问并订阅这个论坛。这个网页还提供了注册后如何访问论坛，论坛提供什么样的帮助，以及论坛的行为准则等信息。

Manning 向读者承诺提供一个读者之间以及读者和作者之间交流的场所。Manning 并不承诺作者一定会参与，作者参与论坛的行为完全出于作者自愿（没有报酬）。我们建议你向作者提一些有挑战性的问题，否则可能提不起作者的兴趣。

只要本书未绝版，作者在线论坛以及早期讨论的存档就可以在出版商的网站上获取到。

关于作者

William Kennedy（@goinggodotnet）是 Ardan 工作室的管理合伙人。这家工作室位于佛罗里达州迈阿密，是一家专注移动、Web 和系统开发的公司。他也是博客 GoingGo.net 的作者，迈阿密 Go 聚会的组织者。从在培训公司 Ardan Labs 开始，他就专注于 Go 语言教学。无论是在当地，

① 本书源代码也可以从 www.epubit.com.cn 本书网页免费下载。

还是在线上，经常可以在大会或者工作坊中看到他的身影。他总是找时间来帮助那些想获取 Go 语言知识、撰写博客和编码技能提升到更高水平的公司或个人。

Brian Ketelsen（@bketelsen）是 XOR Data Exchange 的 CIO 和联合创始人。Brian 也是每年 Go 语言大会（GohperCon）的合办者，同时也是 Gopher Academy 的创立者。作为专注于社区的组织，Gopher Academy 一直在促进 Go 语言的发展和对 Go 语言开发者的培训。Brian 从 2010 年就开始使用 Go 语言。

Erik St. Martin（@erikstmartin）是 XOR Data Exchange 的软件开发总监。他所在的公司专注于大数据分析，最早在得克萨斯州奥斯汀，后来搬到了佛罗里达州坦帕湾。Erik 长时间为开源软件及其社区做贡献。他是每年 GopherCon 的组织者，也是坦帕湾 Go 聚会的组织者。他非常热爱 Go 语言及 Go 语言社区，积极寻求促进社区成长的新方法。

关于封面插图

本书封面插图的标题为“来自东印度的人”。这幅图选自伦敦的 Thomas Jefferys 的《A Collection of the Dresses of Different Nations, Ancient and Modern》（4 卷），出版于 1757 年到 1772 年之间。书籍首页说明了这幅画的制作工艺是铜版雕刻，手工上色，外层用阿拉伯胶做保护。Thomas Jefferys（1719—1771）被称作“地理界的乔治三世国王”。作为制图者，他在当时英国地图商中处于领先地位。他为政府和其他官员雕刻和印刷地图，同时也制作大量的商业地图和地图册，尤其是北美地图。他作为地图制作者的经历，点燃了他收集各地风俗服饰的兴趣，最终成就了这部衣着集。

对遥远大陆的着迷以及对旅行的乐趣，是 18 世纪晚期才兴起的现象。这类收集品也风行一时，向实地旅行家和空想旅行家们介绍各地的风俗。Jefferys 的画集如此多样，生动地向我们描述了 200 年前世界上不同民族的独立特征。从那之后，衣着的特征发生了改变，那个时代不同地区和国家的多样性，也逐渐消失。现在，很难再通过本地居民的服饰来区分他们所在的大陆。也许，从乐观的方面看，也许，从乐观的角度来看，我们用文化的多样性换取了更加多样化的个人生活——当然也是更加多样化和快节奏的科技生活。

在很难将一本计算机书与另一本区分开的时代，Manning 创造性地将两个世纪以前不同地区的多样性，附着在计算机行业的图书封面上，借以来赞美计算机行业的创造力和进取精神也为 Jeffreys 的画带来了新的生命。

目录

1 第 1 章　关于 Go 语言的介绍　1
- 1.1　用Go解决现代编程难题　2
  - 1.1.1　开发速度　2
  - 1.1.2　并发　3
  - 1.1.3　Go 语言的类型系统　5
  - 1.1.4　内存管理　7
- 1.2　你好，Go　7
- 1.3　小结　8

2 第 2 章　快速开始一个Go程序　9
- 2.1　程序架构　9
- 2.2　main 包　11
- 2.3　search 包　13
  - 2.3.1　search.go　13
  - 2.3.2　feed.go　21
  - 2.3.3　match.go/default.go　24
- 2.4　RSS 匹配器　30
- 2.5　小结　36

3 第 3 章　打包和工具链　37
- 3.1　包　37
  - 3.1.1　包名惯例　38
  - 3.1.2　main 包　38
- 3.2　导入　39
  - 3.2.1　远程导入　40
  - 3.2.2　命名导入　40
- 3.3　函数 init　41
- 3.4　使用 Go 的工具　42
- 3.5　进一步介绍 Go 开发工具　44
  - 3.5.1　go vet　44
  - 3.5.2　Go 代码格式化　45
  - 3.5.3　Go 语言的文档　45
- 3.6　与其他 Go 开发者合作　48
- 3.7　依赖管理　48
  - 3.7.1　第三方依赖　49
  - 3.7.2　对 gb 的介绍　50
- 3.8　小结　52

4 第 4 章　数组、切片和映射　53
- 4.1　数组的内部实现和基础功能　53
  - 4.1.1　内部实现　53
  - 4.1.2　声明和初始化　54
  - 4.1.3　使用数组　55
  - 4.1.4　多维数组　58
  - 4.1.5　在函数间传递数组　59
- 4.2　切片的内部实现和基础功能　60
  - 4.2.1　内部实现　60
  - 4.2.2　创建和初始化　61
  - 4.2.3　使用切片　63
  - 4.2.4　多维切片　74
  - 4.2.5　在函数间传递切片　75
- 4.3　映射的内部实现和基础功能　76
  - 4.3.1　内部实现　76
  - 4.3.2　创建和初始化　78
  - 4.3.3　使用映射　79
  - 4.3.4　在函数间传递映射　81
- 4.4　小结　82

需要整本电子书，联系我QQ：2667271557

5 第 5 章 Go 语言的类型系统 83
5.1 用户定义的类型 83
5.2 方法 87
5.3 类型的本质 90
5.3.1 内置类型 91
5.3.2 引用类型 91
5.3.3 结构类型 93
5.4 接口 95
5.4.1 标准库 96
5.4.2 实现 98
5.4.3 方法集 99
5.4.4 多态 103
5.5 嵌入类型 105
5.6 公开或未公开的标识符 113
5.7 小结 121

6 第 6 章 并发 122
6.1 并发与并行 122
6.2 goroutine 125
6.3 竞争状态 132
6.4 锁住共享资源 135
6.4.1 原子函数 135
6.4.2 互斥锁 138
6.5 通道 140
6.5.1 无缓冲的通道 141
6.5.2 有缓冲的通道 146
6.6 小结 149

7 第 7 章 并发模式 150
7.1 runner 150
7.2 pool 158
7.3 work 168
7.4 小结 174

8 第 8 章 标准库 176
8.1 文档与源代码 177
8.2 记录日志 178
8.2.1 log 包 179
8.2.2 定制的日志记录器 182
8.2.3 结论 186
8.3 编码/解码 187
8.3.1 解码 JSON 187
8.3.2 编码 JSON 192
8.3.3 结论 193
8.4 输入和输出 193
8.4.1 Writer 和 Reader 接口 194
8.4.2 整合并完成工作 195
8.4.3 简单的 curl 199
8.4.4 结论 200
8.5 小结 200

9 第 9 章 测试和性能 201
9.1 单元测试 201
9.1.1 基础单元测试 202
9.1.2 表组测试 205
9.1.3 模仿调用 208
9.1.4 测试服务端点 212
9.2 示例 217
9.3 基准测试 220
9.4 小结 224

需要整本电子书，联系我QQ：2667271557

第 1 章　关于 Go 语言的介绍

本章主要内容

- 用 Go 语言解决现代计算难题
- 使用 Go 语言工具

计算机一直在演化，但是编程语言并没有以同样的速度演化。现在的手机，内置的 CPU 核数可能都多于我们使用的第一台电脑。高性能服务器拥有 64 核、128 核，甚至更多核。但是我们依旧在使用为单核设计的技术在编程。

编程的技术同样在演化。大部分程序不再由单个开发者来完成，而是由处于不同时区、不同时间段工作的一组人来完成。大项目被分解为小项目，指派给不同的程序员，程序员开发完成后，再以可以在各个应用程序中交叉使用的库或者包的形式，提交给整个团队。

如今的程序员和公司比以往更加信任开源软件的力量。Go 语言是一种让代码分享更容易的编程语言。Go 语言自带一些工具，让使用别人写的包更容易，并且 Go 语言也让分享自己写的包更容易。

在本章中读者会看到 Go 语言区别于其他编程语言的地方。Go 语言对传统的面向对象开发进行了重新思考，并且提供了更高效的复用代码的手段。Go 语言还让用户能更高效地利用昂贵服务器上的所有核心，而且它编译大型项目的速度也很快。

在阅读本章时，读者会对影响 Go 语言形态的很多决定有一些认识，从它的并发模型到快如闪电的编译器。我们在前言中提到过，这里再强调一次：这本书是写给已经有一定其他编程语言经验、想学习 Go 语言的中级开发者的。本书会提供一个专注、全面且符合习惯的视角。我们同时专注语言的规范和实现，涉及的内容包括语法、Go 语言的类型系统、并发、通道、测试以及其他一些非常广泛的主题。我们相信，对刚开始要学习 Go 语言和想要深入了解语言内部实现的人来说，本书都是最佳选择。

本书示例中的源代码可以在 https://github.com/goinaction/code 下载。

我们希望读者能认识到，Go 语言附带的工具可以让开发人员的生活变得更简单。最后，读者会意识到为什么那么多开发人员用 Go 语言来构建自己的新项目。

1.1 用 Go 解决现代编程难题

Go 语言开发团队花了很长时间来解决当今软件开发人员面对的问题。开发人员在为项目选择语言时，不得不在快速开发和性能之间做出选择。C 和 C++这类语言提供了很快的执行速度，而 Ruby 和 Python 这类语言则擅长快速开发。Go 语言在这两者间架起了桥梁，不仅提供了高性能的语言，同时也让开发更快速。

在探索 Go 语言的过程中，读者会看到精心设计的特性以及简洁的语法。作为一门语言，Go 不仅定义了能做什么，还定义了不能做什么。Go 语言的语法简洁到只有几个关键字，便于记忆。Go 语言的编译器速度非常快，有时甚至会让人感觉不到在编译。所以，Go 开发者能显著减少等待项目构建的时间。因为 Go 语言内置并发机制，所以不用被迫使用特定的线程库，就能让软件扩展，使用更多的资源。Go 语言的类型系统简单且高效，不需要为面向对象开发付出额外的心智，让开发者能专注于代码复用。Go 语言还自带垃圾回收器，不需要用户自己管理内存。让我们快速浏览一下这些关键特性。

1.1.1 开发速度

编译一个大型的 C 或者 C++项目所花费的时间甚至比去喝杯咖啡的时间还长。图 1-1 是 XKCD 中的一幅漫画，描述了在办公室里开小差的经典借口。

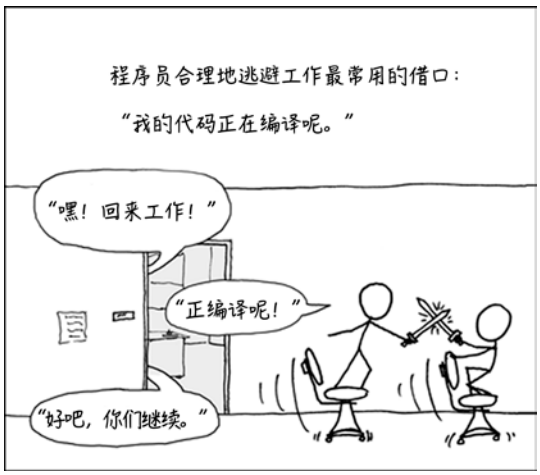


图 1-1 努力工作？（来自 XKCD）

Go 语言使用了更加智能的编译器，并简化了解决依赖的算法，最终提供了更快的编译速度。编译 Go 程序时，编译器只会关注那些直接被引用的库，而不是像 Java、C 和 C++那样，要遍历依赖链中所有依赖的库。因此，很多 Go 程序可以在 1 秒内编译完。在现代硬件上，编译整个 Go

语言的源码树只需要 20 秒。

因为没有从编译代码到执行代码的中间过程，用动态语言编写应用程序可以快速看到输出。代价是，动态语言不提供静态语言提供的类型安全特性，不得不经常用大量的测试套件来避免在运行的时候出现类型错误这类 bug。

想象一下，使用类似 JavaScript 这种动态语言开发一个大型应用程序，有一个函数期望接收一个叫作 `ID` 的字段。这个参数应该是整数，是字符串，还是一个 UUID？要想知道答案，只能去看源代码。可以尝试使用一个数字或者字符串来执行这个函数，看看会发生什么。在 Go 语言里，完全不用为这件事情操心，因为编译器就能帮用户捕获这种类型错误。

1.1.2 并发

作为程序员，要开发出能充分利用硬件资源的应用程序是一件很难的事情。现代计算机都拥有多个核，但是大部分编程语言都没有有效的工具让程序可以轻易利用这些资源。这些语言需要写大量的线程同步代码来利用多个核，很容易导致错误。

Go 语言对并发的支持是这门语言最重要的特性之一。goroutine 很像线程，但是它占用的内存远少于线程，使用它需要的代码更少。通道（channel）是一种内置的数据结构，可以让用户在不同的 goroutine 之间同步发送具有类型的消息。这让编程模型更倾向于在 goroutine 之间发送消息，而不是让多个 goroutine 争夺同一个数据的使用权。让我们看看这些特性的细节。

1．goroutine

goroutine 是可以与其他 goroutine 并行执行的函数，同时也会与主程序（程序的入口）并行执行。在其他编程语言中，你需要用线程来完成同样的事情，而在 Go 语言中会使用同一个线程来执行多个 goroutine。例如，用户在写一个 Web 服务器，希望同时处理不同的 Web 请求，如果使用 C 或者 Java，不得不写大量的额外代码来使用线程。在 Go 语言中，net/http 库直接使用了内置的 goroutine。每个接收到的请求都自动在其自己的 goroutine 里处理。goroutine 使用的内存比线程更少，Go 语言运行时会自动在配置的一组逻辑处理器上调度执行 goroutine。每个逻辑处理器绑定到一个操作系统线程上（见图 1-2）。这让用户的应用程序执行效率更高，而开发工作量显著减少。

如果想在执行一段代码的同时，并行去做另外一些事情，goroutine 是很好的选择。下面是一个简单的例子：

```
func log(msg string) {
    ... 这里是一些记录日志的代码
}

// 代码里有些地方检测到了错误
go log("发生了可怕的事情")
```

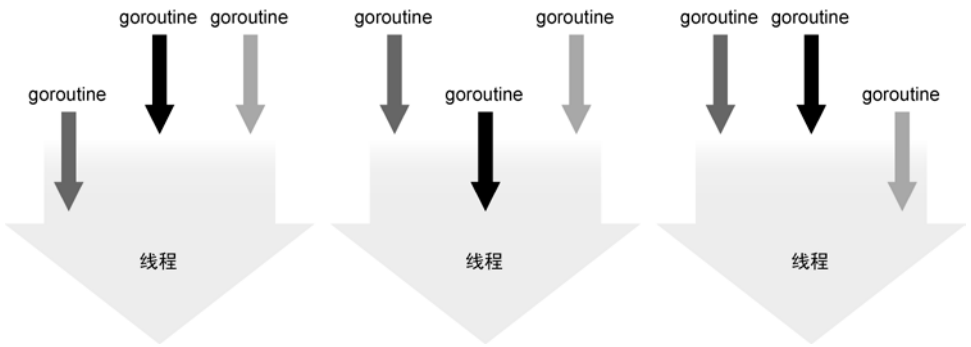


图 1-2　在单一系统线程上执行多个 goroutine

关键字 go 是唯一需要去编写的代码，调度 log 函数作为独立的 goroutine 去运行，以便与其他 goroutine 并行执行。这意味着应用程序的其余部分会与记录日志并行执行，通常这种并行能让最终用户觉得性能更好。就像之前说的，goroutine 占用的资源更少，所以常常能启动成千上万个 goroutine。我们会在第 6 章更加深入地探讨 goroutine 和并发。

2．通道

通道是一种数据结构，可以让 goroutine 之间进行安全的数据通信。通道可以帮用户避免其他语言里常见的共享内存访问的问题。

并发的最难的部分就是要确保其他并发运行的进程、线程或 goroutine 不会意外修改用户的数据。当不同的线程在没有同步保护的情况下修改同一个数据时，总会发生灾难。在其他语言中，如果使用全局变量或者共享内存，必须使用复杂的锁规则来防止对同一个变量的不同步修改。

为了解决这个问题，通道提供了一种新模式，从而保证并发修改时的数据安全。通道这一模式保证同一时刻只会有一个 goroutine 修改数据。通道用于在几个运行的 goroutine 之间发送数据。在图 1-3 中可以看到数据是如何流动的示例。想象一个应用程序，有多个进程需要顺序读取或者修改某个数据，使用 goroutine 和通道，可以为这个过程建立安全的模型。

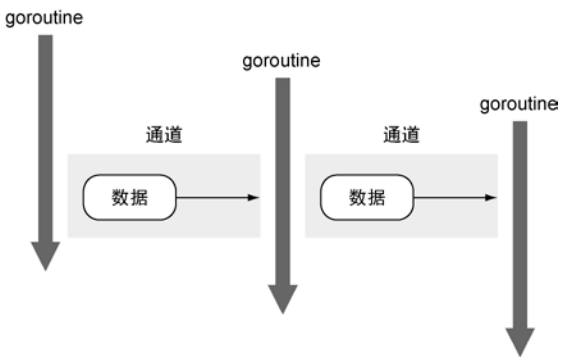


图 1-3　使用通道在 goroutine 之间安全地发送数据

图 1-3 中有 3 个 goroutine，还有 2 个不带缓存的通道。第一个 goroutine 通过通道把数据传给已经在等待的第二个 goroutine。在两个 goroutine 间传输数据是同步的，一旦传输完成，两个 goroutine 都会知道数据已经完成传输。当第二个 goroutine 利用这个数据完成其任务后，将这个数据传给第三个正在等待的 goroutine。这次传输依旧是同步的，两个 goroutine 都会确认数据传输完成。这种在 goroutine 之间安全传输数据的方法不需要任何锁或者同步机制。

需要强调的是，通道并不提供跨 goroutine 的数据访问保护机制。如果通过通道传输数据的一份副本，那么每个 goroutine 都持有一份副本，各自对自己的副本做修改是安全的。当传输的是指向数据的指针时，如果读和写是由不同的 goroutine 完成的，每个 goroutine 依旧需要额外的同步动作。

1.1.3 Go 语言的类型系统

Go 语言提供了灵活的、无继承的类型系统，无需降低运行性能就能最大程度上复用代码。这个类型系统依然支持面向对象开发，但避免了传统面向对象的问题。如果你曾经在复杂的 Java 和 C++程序上花数周时间考虑如何抽象类和接口，你就能意识到 Go 语言的类型系统有多么简单。Go 开发者使用组合（composition）设计模式，只需简单地将一个类型嵌入到另一个类型，就能复用所有的功能。其他语言也能使用组合，但是不得不和继承绑在一起使用，结果使整个用法非常复杂，很难使用。在 Go 语言中，一个类型由其他更微小的类型组合而成，避免了传统的基于继承的模型。

另外，Go 语言还具有独特的接口实现机制，允许用户对行为进行建模，而不是对类型进行建模。在 Go 语言中，不需要声明某个类型实现了某个接口，编译器会判断一个类型的实例是否符合正在使用的接口。Go 标准库里的很多接口都非常简单，只开放几个函数。从实践上讲，尤其对那些使用类似 Java 的面向对象语言的人来说，需要一些时间才能习惯这个特性。

1．类型简单

Go 语言不仅有类似 `int` 和 `string` 这样的内置类型，还支持用户定义的类型。在 Go 语言中，用户定义的类型通常包含一组带类型的字段，用于存储数据。Go 语言的用户定义的类型看起来和 C 语言的结构很像，用起来也很相似。不过 Go 语言的类型可以声明操作该类型数据的方法。传统语言使用继承来扩展结构——Client 继承自 User，User 继承自 Entity，Go 语言与此不同，Go 开发者构建更小的类型——Customer 和 Admin，然后把这些小类型组合成更大的类型。图 1-4 展示了继承和组合之间的不同。

2．Go 接口对一组行为建模

接口用于描述类型的行为。如果一个类型的实例实现了一个接口，意味着这个实例可以执行

一组特定的行为。你甚至不需要去声明这个实例实现某个接口，只需要实现这组行为就好。其他的语言把这个特性叫作*鸭子类型*——如果它叫起来像鸭子，那它就可能是只鸭子。Go 语言的接口也是这么做的。在 Go 语言中，如果一个类型实现了一个接口的所有方法，那么这个类型的实例就可以存储在这个接口类型的实例中，不需要额外声明。

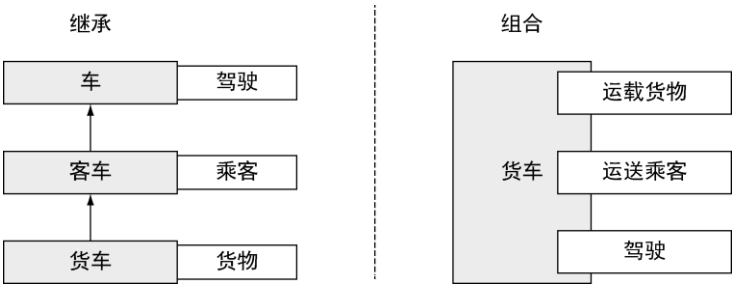


图 1-4 继承和组合的对比

在类似 Java 这种严格的面向对象语言中，所有的设计都围绕接口展开。在编码前，用户经常不得不思考一个庞大的继承链。下面是一个 Java 接口的例子：

```
interface User {
    public void login();
    public void logout();
}
```

在 Java 中要实现这个接口，要求用户的类必须满足 `User` 接口里的所有约束，并且显式声明这个类实现了这个接口。而 Go 语言的接口一般只会描述一个单一的动作。在 Go 语言中，最常使用的接口之一是 `io.Reader`。这个接口提供了一个简单的方法，用来声明一个类型有数据可以读取。标准库内的其他函数都能理解这个接口。这个接口的定义如下：

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

为了实现 `io.Reader` 这个接口，你只需要实现一个 `Read` 方法，这个方法接受一个 `byte` 切片，返回一个整数和可能出现的错误。

这和传统的面向对象编程语言的接口系统有本质的区别。Go 语言的接口更小，只倾向于定义一个单一的动作。实际使用中，这更有利于使用组合来复用代码。用户几乎可以给所有包含数据的类型实现 `io.Reader` 接口，然后把这个类型的实例传给任意一个知道如何读取 `io.Reader` 的 Go 函数。

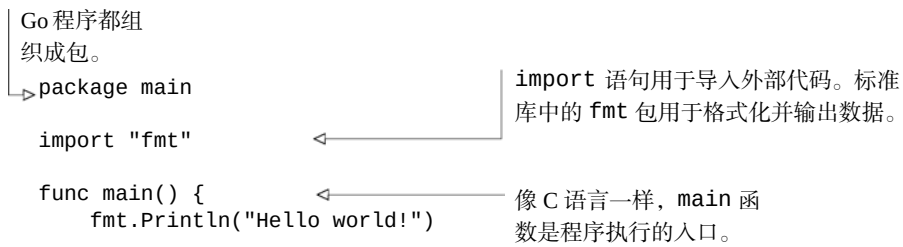
Go 语言的整个网络库都使用了 `io.Reader` 接口，这样可以将程序的功能和不同网络的实现分离。这样的接口用起来有趣、优雅且自由。文件、缓冲区、套接字以及其他的数据源都实现了 `io.Reader` 接口。使用同一个接口，可以高效地操作数据，而不用考虑到底数据来自哪里。

1.1.4 内存管理

不当的内存管理会导致程序崩溃或者内存泄漏，甚至让整个操作系统崩溃。Go 语言拥有现代化的垃圾回收机制，能帮你解决这个难题。在其他系统语言（如 C 或者 C++）中，使用内存前要先分配这段内存，而且使用完毕后要将其释放掉。哪怕只做错了一件事，都可能导致程序崩溃或者内存泄漏。可惜，追踪内存是否还被使用本身就是十分艰难的事情，而要想支持多线程和高并发，更是让这件事难上加难。虽然 Go 语言的垃圾回收会有一些额外的开销，但是编程时，能显著降低开发难度。Go 语言把无趣的内存管理交给专业的编译器去做，而让程序员专注于更有趣的事情。

1.2 你好，Go

感受一门语言最简单的方法就是实践。让我们看看用 Go 语言如何编写经典的 `Hello World!`应用程序：



运行这个示例程序后会在屏幕上输出我们熟悉的一句话。但是怎么运行呢？无须在机器上安装 Go 语言，在浏览器中就可以使用几乎所有 Go 语言的功能。

介绍 Go Playground

Go Playground 允许在浏览器里编辑并运行 Go 语言代码。在浏览器中打开 http://play.golang.org。浏览器里展示的代码是可编辑的（见图 1-5）。点击 Run，看看会发生什么。

可以把输出的问候文字改成别的语言。试着改动 `fmt.Println()`里面的文字，然后再次点击 Run。

分享 Go 代码 Go 开发者使用 Playground 分享他们的想法，测试理论，或者调试代码。你也可以这么做。每次使用 Playground 创建一个新程序之后，可以点击 Share 得到一个用于分享的网址。任何人都能打开这个链接。试试 http://play.golang.org/p/EWIXicJdmz。

图 1-5　Go Playground

要给想要学习写东西或者寻求帮助的同事或者朋友演示某个想法时，Go Playground 是非常好的方式。在 Go 语言的 IRC 频道、Slack 群组、邮件列表和 Go 开发者发送的无数邮件里，用户都能看到创建、修改和分享 Go Playground 上的程序。

1.3　小结

- Go 语言是现代的、快速的，带有一个强大的标准库。
- Go 语言内置对并发的支持。
- Go 语言使用接口作为代码复用的基础模块。

第 2 章　快速开始一个 Go 程序

本章主要内容

- 学习如何写一个复杂的 Go 程序
- 声明类型、变量、函数和方法
- 启动并同步操作 goroutine
- 使用接口写通用的代码
- 处理程序逻辑和错误

为了能更高效地使用语言进行编码，Go 语言有自己的哲学和编程习惯。Go 语言的设计者们从编程效率出发设计了这门语言，但又不会丢掉访问底层程序结构的能力。设计者们通过一组最少的关键字、内置的方法和语法，最终平衡了这两方面。Go 语言也提供了完善的标准库。标准库提供了构建实际的基于 Web 和基于网络的程序所需的所有核心库。

让我们通过一个完整的 Go 语言程序，来看看 Go 语言是如何实现这些功能的。这个程序实现的功能很常见，能在很多现在开发的 Go 程序里发现类似的功能。这个程序从不同的数据源拉取数据，将数据内容与一组搜索项做对比，然后将匹配的内容显示在终端窗口。这个程序会读取文本文件，进行网络调用，解码 XML 和 JSON 成为结构化类型数据，并且利用 Go 语言的并发机制保证这些操作的速度。

读者可以下载本章的代码，用自己喜欢的编辑器阅读。代码存放在这个代码库：

```
https://github.com/goinaction/code/tree/master/chapter2/sample
```

没必要第一次就读懂本章的所有内容，可以多读两遍。在学习时，虽然很多现代语言的概念可以对应到 Go 语言中，Go 语言还是有一些独特的特性和风格。如果放下已经熟悉的编程语言，用一种全新的眼光来审视 Go 语言，你会更容易理解并接受 Go 语言的特性，发现 Go 语言的优雅。

2.1　程序架构

在深入代码之前，让我们看一下程序的架构（如图 2-1 所示），看看如何在所有不同的数据

源中搜索数据。

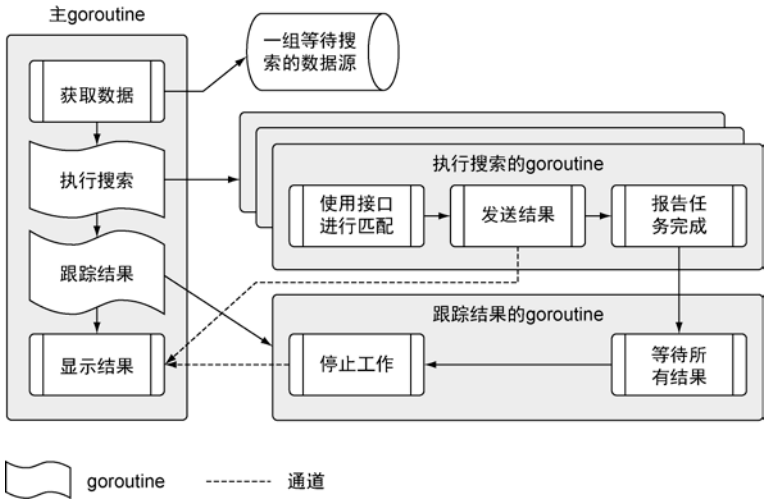


图 2-1　程序架构流程图

这个程序分成多个不同步骤，在多个不同的 goroutine 里运行。我们会根据流程展示代码，从主 goroutine 开始，一直到执行搜索的 goroutine 和跟踪结果的 goroutine，最后回到主 goroutine。首先来看一下整个项目的结构，如代码清单 2-1 所示。

代码清单 2-1　应用程序的项目结构

```
cd $GOPATH/src/github.com/goinaction/code/chapter2

- sample
    - data
        data.json    -- 包含一组数据源
    - matchers
        rss.go       -- 搜索 rss 源的匹配器
    - search
        default.go   -- 搜索数据用的默认匹配器
        feed.go      -- 用于读取 json 数据文件
        match.go     -- 用于支持不同匹配器的接口
        search.go    -- 执行搜索的主控制逻辑
    main.go          -- 程序的入口
```

这个应用的代码使用了 4 个文件夹，按字母顺序列出。文件夹 data 中有一个 JSON 文档，其内容是程序要拉取和处理的数据源。文件夹 matchers 中包含程序里用于支持搜索不同数据源的代码。目前程序只完成了支持处理 RSS 类型的数据源的匹配器。文件夹 search 中包含使用不同匹配器进行搜索的业务逻辑。最后，父级文件夹 sample 中有个 main.go 文件，这是整个程序的入口。

现在了解了如何组织程序的代码，可以继续探索并了解程序是如何工作的。让我们从程序的入口开始。

2.2 main 包

程序的主入口可以在 main.go 文件里找到，如代码清单 2-2 所示。虽然这个文件只有 21 行代码，依然有几点需要注意。

代码清单 2-2　main.go

```
01 package main
02
03 import (
04     "log"
05     "os"
06
07     _ "github.com/goinaction/code/chapter2/sample/matchers"
08      "github.com/goinaction/code/chapter2/sample/search"
09 )
10
11 // init 在 main 之前调用
12 func init() {
13     // 将日志输出到标准输出
14     log.SetOutput(os.Stdout)
15 }
16
17 // main 是整个程序的入口
18 func main() {
19     // 使用特定的项做搜索
20     search.Run("president")
21 }
```

每个可执行的 Go 程序都有两个明显的特征。一个特征是第 18 行声明的名为 `main` 的函数。构建程序在构建可执行文件时，需要找到这个已经声明的 `main` 函数，把它作为程序的入口。第二个特征是程序的第 01 行的包名 `main`，如代码清单 2-3 所示。

代码清单 2-3　main.go：第 01 行

```
01 package main
```

可以看到，`main` 函数保存在名为 `main` 的包里。如果 `main` 函数不在 `main` 包里，构建工具就不会生成可执行的文件。

Go 语言的每个代码文件都属于一个包，main.go 也不例外。包这个特性对于 Go 语言来说很重要，我们会在第 3 章中接触到更多细节。现在，只要简单了解以下内容：一个包定义一组编译过的代码，包的名字类似命名空间，可以用来间接访问包内声明的标识符。这个特性可以把不同包中定义的同名标识符区别开。

现在，把注意力转到 main.go 的第 03 行到第 09 行，如代码清单 2-4 所示，这里声明了所有的导入项。

代码清单 2-4　main.go：第 03 行到第 09 行

```
03 import (
04     "log"
05     "os"
06
07     _ "github.com/goinaction/code/chapter2/sample/matchers"
08      "github.com/goinaction/code/chapter2/sample/search"
09 )
```

顾名思义，关键字 `import` 就是导入一段代码，让用户可以访问其中的标识符，如类型、函数、常量和接口。在这个例子中，由于第 08 行的导入，main.go 里的代码就可以引用 `search` 包里的 `Run` 函数。程序的第 04 行和第 05 行导入标准库里的 `log` 和 `os` 包。

所有处于同一个文件夹里的代码文件，必须使用同一个包名。按照惯例，包和文件夹同名。就像之前说的，一个包定义一组编译后的代码，每段代码都描述包的一部分。如果回头去看看代码清单 2-1，可以看看第 08 行的导入是如何指定那个项目里名叫 `search` 的文件夹的。

读者可能注意到第 07 行导入 `matchers` 包的时候，导入的路径前面有一个下划线，如代码清单 2-5 所示。

代码清单 2-5　main.go：第 07 行

```
07     _ "github.com/goinaction/code/chapter2/sample/matchers"
```

这个技术是为了让 Go 语言对包做初始化操作，但是并不使用包里的标识符。为了让程序的可读性更强，Go 编译器不允许声明导入某个包却不使用。下划线让编译器接受这类导入，并且调用对应包内的所有代码文件里定义的 `init` 函数。对这个程序来说，这样做的目的是调用 `matchers` 包中的 rss.go 代码文件里的 `init` 函数，注册 RSS 匹配器，以便后用。我们后面会展示具体的工作方式。

代码文件 main.go 里也有一个 `init` 函数，在第 12 行到第 15 行中声明，如代码清单 2-6 所示。

代码清单 2-6　main.go：第 11 行到第 15 行

```
11 // init 在 main 之前调用
12 func init() {
13     // 将日志输出到标准输出
14     log.SetOutput(os.Stdout)
15 }
```

程序中每个代码文件里的 `init` 函数都会在 `main` 函数执行前调用。这个 `init` 函数将标准库里日志类的输出，从默认的标准错误（`stderr`），设置为标准输出（`stdout`）设备。在第 7 章，我们会进一步讨论 `log` 包和标准库里其他重要的包。

最后，让我们看看 `main` 函数第 20 行那条语句的作用，如代码清单 2-7 所示。

代码清单 2-7　main.go：第 19 行到第 20 行

```
19      // 使用特定的项做搜索
20      search.Run("president")
```

可以看到，这一行调用了 search 包里的 Run 函数。这个函数包含程序的核心业务逻辑，需要传入一个字符串作为搜索项。一旦 Run 函数退出，程序就会终止。

现在，让我们看看 search 包里的代码。

2.3 search 包

这个程序使用的框架和业务逻辑都在 search 包里。这个包由 4 个不同的代码文件组成，每个文件对应一个独立的职责。我们会逐步分析这个程序的逻辑，到时再说明各个代码文件的作用。

由于整个程序都围绕匹配器来运作，我们先简单介绍一下什么是匹配器。这个程序里的匹配器，是指包含特定信息、用于处理某类数据源的实例。在这个示例程序中有两个匹配器。框架本身实现了一个无法获取任何信息的默认匹配器，而在 matchers 包里实现了 RSS 匹配器。RSS 匹配器知道如何获取、读入并查找 RSS 数据源。随后我们会扩展这个程序，加入能读取 JSON 文档或 CSV 文件的匹配器。我们后面会再讨论如何实现匹配器。

2.3.1 search.go

代码清单 2-8 中展示的是 search.go 代码文件的前 9 行代码。之前提到的 Run 函数就在这个文件里。

代码清单 2-8　search/search.go：第 01 行到第 09 行

```
01 package search
02
03 import (
04      "log"
05      "sync"
06 )
07
08 // 注册用于搜索的匹配器的映射
09 var matchers = make(map[string]Matcher)
```

可以看到，每个代码文件都以 package 关键字开头，随后跟着包的名字。文件夹 search 下的每个代码文件都使用 search 作为包名。第 03 行到第 06 行代码导入标准库的 log 和 sync 包。

与第三方包不同，从标准库中导入代码时，只需要给出要导入的包名。编译器查找包的时候，总是会到 GOROOT 和 GOPATH 环境变量（如代码清单 2-9 所示）引用的位置去查找。

代码清单 2-9　GOROOT 和 GOPATH 环境变量

```
GOROOT="/Users/me/go"
GOPATH="/Users/me/spaces/go/projects"
```

log 包提供打印日志信息到标准输出（stdout）、标准错误（stderr）或者自定义设备的功能。sync 包提供同步 goroutine 的功能。这个示例程序需要用到同步功能。第 09 行是全书第一次声明一个变量，如代码清单 2-10 所示。

代码清单 2-10　search/search.go：第 08 行到第 09 行

```
08 // 注册用于搜索的匹配器的映射
09 var matchers = make(map[string]Matcher)
```

这个变量没有定义在任何函数作用域内，所以会被当成包级变量。这个变量使用关键字 var 声明，而且声明为 Matcher 类型的映射（map），这个映射以 string 类型值作为键，Matcher 类型值作为映射后的值。Matcher 类型在代码文件 matcher.go 中声明，后面再讲这个类型的用途。这个变量声明还有一个地方要强调一下：变量名 matchers 是以小写字母开头的。

在 Go 语言里，标识符要么从包里公开，要么不从包里公开。当代码导入了一个包时，程序可以直接访问这个包中任意一个公开的标识符。这些标识符以大写字母开头。以小写字母开头的标识符是不公开的，不能被其他包中的代码直接访问。但是，其他包可以间接访问不公开的标识符。例如，一个函数可以返回一个未公开类型的值，那么这个函数的任何调用者，哪怕调用者不是在这个包里声明的，都可以访问这个值。

这行变量声明还使用赋值运算符和特殊的内置函数 make 初始化了变量，如代码清单 2-11 所示。

代码清单 2-11　构建一个映射

```
make(map[string]Matcher)
```

map 是 Go 语言里的一个引用类型，需要使用 make 来构造。如果不先构造 map 并将构造后的值赋值给变量，会在试图使用这个 map 变量时收到出错信息。这是因为 map 变量默认的零值是 nil。在第 4 章我们会进一步了解关于映射的细节。

在 Go 语言中，所有变量都被初始化为其零值。对于数值类型，零值是 0；对于字符串类型，零值是空字符串；对于布尔类型，零值是 false；对于指针，零值是 nil。对于引用类型来说，所引用的底层数据结构会被初始化为对应的零值。但是被声明为其零值的引用类型的变量，会返回 nil 作为其值。

现在，让我们看看之前在 main 函数中调用的 Run 函数的内容，如代码清单 2-12 所示。

代码清单 2-12　search/search.go：第 11 行到第 57 行

```
11 // Run 执行搜索逻辑
12 func Run(searchTerm string) {
13     // 获取需要搜索的数据源列表
14     feeds, err := RetrieveFeeds()
```

```
    if err != nil {
        log.Fatal(err)
    }

    // 创建一个无缓冲的通道，接收匹配后的结果
    results := make(chan *Result)

    // 构造一个 waitGroup，以便处理所有的数据源
    var waitGroup sync.WaitGroup

    // 设置需要等待处理
    // 每个数据源的 goroutine 的数量
    waitGroup.Add(len(feeds))

    // 为每个数据源启动一个 goroutine 来查找结果
    for _, feed := range feeds {
        // 获取一个匹配器用于查找
        matcher, exists := matchers[feed.Type]
        if !exists {
            matcher = matchers["default"]
        }

        // 启动一个 goroutine 来执行搜索
        go func(matcher Matcher, feed *Feed) {
            Match(matcher, feed, searchTerm, results)
            waitGroup.Done()
        }(matcher, feed)
    }

    // 启动一个 goroutine 来监控是否所有的工作都做完了
    go func() {
        // 等候所有任务完成
        waitGroup.Wait()

        // 用关闭通道的方式，通知 Display 函数
        // 可以退出程序了
        close(results)
    }()

    // 启动函数，显示返回的结果，并且
    // 在最后一个结果显示完后返回
    Display(results)
}
```

Run 函数包括了这个程序最主要的控制逻辑。这段代码很好地展示了如何组织 Go 程序的代码，以便正确地并发启动和同步 goroutine。先来一步一步考察整个逻辑，再考察每步实现代码的细节。

先来看看 Run 函数是怎么定义的，如代码清单 2-13 所示。

代码清单 2-13　search/search.go：第 11 行到第 12 行

```
// Run 执行搜索逻辑
func Run(searchTerm string) {
```

Go 语言使用关键字 func 声明函数，关键字后面紧跟着函数名、参数以及返回值。对于 Run 这个函数来说，只有一个参数，是 string 类型的，名叫 searchTerm。这个参数是 Run 函数要搜索的搜索项，如果回头看看 main 函数（如代码清单 2-14 所示），可以看到如何传递这个搜索项。

代码清单 2-14　main.go：第 17 行到第 21 行

```
17 // main 是整个程序的入口
18 func main() {
19     // 使用特定的项做搜索
20     search.Run("president")
21 }
```

Run 函数做的第一件事情就是获取数据源 feeds 列表。这些数据源从互联网上抓取数据，之后对数据使用特定的搜索项进行匹配，如代码清单 2-15 所示。

代码清单 2-15　search/search.go：第 13 行到第 17 行

```
13     // 获取需要搜索的数据源列表
14     feeds, err := RetrieveFeeds()
15     if err != nil {
16         log.Fatal(err)
17     }
```

这里有几个值得注意的重要概念。第 14 行调用了 search 包的 RetrieveFeeds 函数。这个函数返回两个值。第一个返回值是一组 Feed 类型的切片。切片是一种实现了一个动态数组的引用类型。在 Go 语言里可以用切片来操作一组数据。第 4 章会进一步深入了解有关切片的细节。

第二个返回值是一个错误值。在第 15 行，检查返回的值是不是真的是一个错误。如果真的发生错误了，就会调用 log 包里的 Fatal 函数。Fatal 函数接受这个错误的值，并将这个错误在终端窗口里输出，随后终止程序。

不仅仅是Go语言，很多语言都允许一个函数返回多个值。一般会像RetrieveFeeds函数这样声明一个函数返回一个值和一个错误值。如果发生了错误，永远不要使用该函数返回的另一个值[①]。这时必须忽略另一个值，否则程序会产生更多的错误，甚至崩溃。

让我们仔细看看从函数返回的值是如何赋值给变量的，如代码清单 2-16 所示。

代码清单 2-16　search/search.go：第 13 行到第 14 行

```
13     // 获取需要搜索的数据源列表
14     feeds, err := RetrieveFeeds()
```

这里可以看到简化变量声明运算符（:=）。这个运算符用于声明一个变量，同时给这个变量

① 这个说法并不严格成立，Go 标准库中的 io.Reader.Read 方法就允许同时返回数据和错误。但是，如果是自己实现的函数，要尽量遵守这个原则，保持含义足够明确。——译者注