

Go

The Way to Go

Go入门指南

Ivo Balbaert 著

陈佳桦 译

版权信息

书名：Go入门指南——The Way to Go（中文版）

作者：Ivo Balbaert

译者：陈佳桦

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

目录

[译者序](#)

[前言](#)

[1 Go 语言的起源，发展与普及](#)

[1.1 起源与发展](#)

[1.2 语言的主要特性与发展的环境和影响因素](#)

[2 安装与运行环境](#)

[2.1 平台与架构](#)

[2.2 Go 环境变量](#)

[2.3 在 Linux 上安装 Go](#)

[2.4 在 Mac OS X 上安装 Go](#)

[2.5 在 Windows 上安装 Go](#)

[2.6 安装目录清单](#)

[2.7 Go 类虚拟机（runtime）](#)

[2.8 Go 解释器](#)

[3 编辑器、集成开发环境与其它工具](#)

[3.1 Go 开发环境的基本要求](#)

[3.2 编辑器和集成开发环境](#)

[3.3 调试器](#)

[3.4 构建并运行 Go 程序](#)

[3.5 格式化代码](#)

[3.6 生成代码文档](#)

[3.7 其它工具](#)

[3.8 Go 性能说明](#)

[3.9 与其它语言进行交互](#)

[4 基本结构和基本数据类型](#)

[4.1 文件名、关键字与标识符](#)

[4.2 Go 程序的基本结构和要素](#)

[4.3 常量](#)

[4.4 变量](#)

[4.5 基本类型和运算符](#)

[4.6 字符串](#)

[4.7 strings 和 strconv 包](#)

[4.8 时间和日期](#)

[4.9 指针](#)

[5 控制结构](#)

[5.1 if-else 结构](#)

译者序

在接触 Go 语言之后，对这门编程语言非常着迷，期间也陆陆续续开始一些帮助国内编程爱好者了解和发展 Go 语言的工作，比如开始录制视频教程《Go编程基础》。但由于目前国内并没有比较好的 Go 语言书籍，而国外的优秀书籍因为英文的缘故在

一定程度上也为不少 Go 语言爱好者带来了一些学习上的困扰，不仅为了加快扩散 Go 爱好者的国内群体，同时充分贯彻 Asta 谢的为己为人精神，本人在完成阅读这本名叫《The Way to Go》之后，决定每天抽出一段时间来进行翻译的工作，并且以开源的形式免费分享给有需要的 Go 语言爱好者。

尽管该书对目前 Go 语言版本来说有小部分内容相对过时，但是为当下不可多得的好书，部分内容已获得作者同意根据当前 Go 语言版本进行修改。

该翻译版本已获得原作者（Ivo Balbaert）本人授权，并表示支持开源事业的发展！

本书全面地讲解了 Go 学习者需要了解的知识，不仅涵盖了基础概念，同时也包含一些项目开发中所涉及到的高级技巧。每个知识点的讲解和示例都非常完备，部分章节在最后还会提供练习题以便学习者巩固及加深印象，是 Go 学习者不可错过的一本经典书籍。

同时感谢以下几位开源爱好者的共同奉献：

- [zhanning](#)
- [themorecolor](#)
- [everyx](#)

前言

用更少的代码，更短的编译时间，创建运行更快的程序，享受更多的乐趣

对于学习 Go 编程语言的爱好者来说，这本书无疑是最适合你的一本书籍，这里包含了当前最全面的学习资源。本书通过对官方的在线文档、名人博客、书籍、相关文章以及演讲的资料收集和整理，并结合我自身在软件工程、编程语言和数据库开发的授课经验，将这些零碎的知识点组织成系统化的概念和技术分类来进行讲解。

随着软件规模的不断扩大，诸多的学者和谷歌的开发者们在公司内部的软件开发过程中开始经历大量的挫折，在诸多问题上都不能给出令人满意的解决方案，尤其是在使用 C++ 来开发大型的服务端软件时，情况更是不容乐观。由于二进制文件一般都是非常巨大的，因此需要耗费大量的时间在编译这些文件上，同时编程语言的设计思想也已经非常陈旧，这些情况都充分证明了现有的编程语言已不符合时的生产环境。尽管硬件在过去的几十年中有了飞速的发展，但人们依旧没有找到机会去改变 C++ 在软件开发的重要地位，并在实际开发过程中忍受着它所带来的令人头疼的一些问题。因此学者们坐下来总结出了现在生产环境与软件开发之间的主要矛盾，并尝试设计一门全新的编程语言来解决这些问题。

以下就是他们讨论得出的对编程语言的设计要求：

- 能够以更快的速度开发软件
- 开发出的软件能够很好地在现代的多核计算机上工作
- 开发出的软件能够很好地在网络环境下工作
- 使人们能够享受软件开发的过程

Go 语言就在这样的环境下诞生了，它让人感觉像是 Python 或 Ruby 这样的动态语言，但却又拥有像 C 或者 Java 这类语言的高性能和安全性。

Go 语言出现的目的是希望在编程领域创造最实用的方式来进行软件开发。它并不是要用奇怪的语法和晦涩难懂的概念来从根本上推翻已有的编程语言，而是建立并改善了 C、Java、C# 中的许多语法风格。它提倡通过接口来针对面向对象编程，通过 goroutine 和 channel 来支持并发和并行编程。

这本书是为那些想要学习 Go 这门全新的，迷人的和充满希望的编程语言的开发者量身定做的。当然，你在学习 Go 语言之前需要具备一些关于编程的基础知识和经验，并且拥有合适的学习环境，但你并不需要对 C 或者 Java 或其他类似的语言有非常深入的了解。

对于那些熟悉 C 或者面向对象编程语言的开发者，我们将会在本书中用 Go 和一些编程语言的相关概念进行比较（书中会使用大家所熟知的缩写“OO”来表示面向对象）。

本书将会从最基础的概念讲起，同时也会讨论一些类似在应用 goroutine 和 channel 时有多少种不同的模式，如何在 Go 语言中使用谷歌 API，如何操作内存，如何在 Go 语言中进行程序测试和如何使用模板来开发 Web 应用这些高级概念和技巧。

在本书的第一部分，我们将会讨论 Go 语言的起源（第1章），以及如何安装 Go 语言（第2章）和开发环境（第3章）。

在本书的第二部分，我们将会带领你贯穿 Go 语言的核心思想，譬如简单与复杂类型（第4，7，8章），控制结构（第5章），函数（第6章），结构与方法（第10章）和接口（第11章）。我们会对 Go 语言的函数式和面向对象编程进行透彻的讲解，包括如何使用 Go 语言来构造大型项目（第9章）。

在本书的第三部分，你将会学习到如何处理不同格式的文件（第12章）和如何在 Go 语言中巧妙地使用错误处理机制（第13

章)。然后我们会对 Go 语言中最值得称赞的设计 goroutine 和 channel 进行并发和多核应用的基本技巧的讲解(第14章)。最后,我们会讨论如何将 Go 语言应用到分布式和Web应用中的相关网络技巧(第15章)。

我们会在本书的第四部分向你展示许多 Go 语言的开发模式和一些编码规范,以及一些非常有用的代码片段(第18章)。在前面章节完成对所有的 Go 语言技巧的学习之后,你将会学习如何构造一个完整 Go 语言项目(第19章),然后我们会介绍一些关于 Go 语言在云(Google App Engine)方面的应用(第20章)。在本书的最后一章(第21章),我们会讨论一些在全世界范围内已经将 Go 语言投入实际开发的公司和组织。本书将会在最后给出一些对 Go 语言爱好者的引用,Go 相关包和工具的参考,以及章节练习的答案和所有参考资源和文献的清单。

Go 语言有一个被称之为“没有废物”的宗旨,就是将一切没有必要的东西都去掉,不能去掉的就无底线地简化,同时追求最大程度的自动化。他完美地诠释了敏捷编程的KISS秘诀:短小精悍!

Go 语言通过改善或去除在 C、C++ 或 Java 中的一些所谓“开放”特性来让开发者们的工作更加便利。这里只举例其中的几个,比如对于变量的默认初始化,内存分配与自动回收,以及更简洁却不失健壮的控制结构。同时我们也会发现 Go 语言旨在减少不必要的编码工作,这使得 Go 语言的代码更加简洁,从而比传统的面向对象语言更容易阅读和理解。

与 C++ 或 Java 这些有着庞大体系的语言相比,Go 语言简洁到可以将它整个的装入你的大脑中,而且比学习 Scala (Java 的并发语言)有更低的门槛,真可谓是 21 世纪的 C 语言!

作为一门系统编程语言,你不应该为 Go 语言的大多数代码示例和练习都和控制台有着密不可分的关系而感到惊奇,因为提供平台依赖性的 GUI (用户界面) 框架并不是一个简单的任务。有许多由第三方发起的 GUI 框架项目正在如火如荼地进行中,或许我们会在不久的将来看到一些可用的 Go 语言 GUI 框架。不过现阶段的 Go 语言已经提供了大量有关 Web 方面的功能,我们可以通过它强大的 http 和 template 包来达到 Web 应用的 GUI 实现。

我们会经常涉及到一些关于 Go 语言的编码规范,了解和使用这些已经被广泛认同的规范应该是你学习阶段最重要的实践。我会在书中尽量使用已经讲解的概念或者技巧来解释相关的代码示例,以避免你在不了解某些高级概念的情况下而感到迷茫。

我们通过 227 个完整的代码示例和书中的解释说明来对所有涉及到的概念和技巧进行彻底的讲解,你可以下载这些代码到你的电脑上运行,从而加深对概念的理解。

本书会尽可能地将前后章节的内容联系起来,当然这也同时要求你通过学习不同的知识来对一个问题提出尽可能多的解决方案。记住,学习一门新语言的最佳方式就是实践,运行它的代码,修改并尝试更多的方案。因此,你绝对不可以忽略书中的 130 个代码练习,这将对你的学习好 Go 语言有很大的帮助。比如,我们就为斐波那契算法提供了 13 个不同的版本,而这些版本都使用了不同的概念和技巧。

你可以通过访问本书的[官方网站](<https://sites.google.com/site/thewaytogo2012/>)下载书中的代码,并获得有关本书的勘误情况和内容更新。

为了让你在成为 Go 语言大师的道路上更加顺利,我们会专注于一些特别的章节以提供 Go 语言开发模式的最佳实践,同时也会帮助初学者逃离一些语言的陷阱。第 18 章可以作为你在开发时的一个参考手册,因为当中包含了众多的有价值的代码片段以及相关的解释说明。

最后要说明的是,你可以通过完整的索引来快速定位你需要阅读的章节。书中所有的代码都在 Go1 版本下测试通过。(译者注:所有代码经作者同意将会根据需要进行相关修改以在 Go1.1 版本下运行)

这里有一段来自在 C++、Java 和 Python 领域众所周知的专家 bruce Eckel 的评论:

“作为一个有着 C/C++ 背景的开发者的,我在使用 Go 语言时仿佛呼吸到了新鲜空气一般,令人心旷神怡。我认为使用 Go 语言进行系统编程开发比使用 C++ 有着更显著的优势,因为它在解决一些很难用 C++ 解决的问题的同时,让我的工作变得更加高效。我并不是说 C++ 的存在是一个错误,相反地,我认为这是历史发展的必然结果。当我深陷在 C 语言这门略微比汇编语言好一点的泥潭时,我坚信任何语言的构造都不可能支持大型项目的开发。像垃圾回收或并发语言支持这类东西,在当时都是极其荒谬的主意,根本没有人会在乎。C++ 向大型项目开发迈出了重要的第一步,带领我们走进这个广袤无垠的世界。很庆幸 Stroustrup 做了让 C++ 兼容 C 语言以能够让其编译 C 程序这个正确的决定。我们当时需要 C++ 的出现。”

“之后我们学到了更多。我们毫无疑问地接受了垃圾回收,异常处理和虚拟机这些当年人们认为只有疯子才会想的东西。C++ 的复杂程度(新版的 C++ 甚至更加复杂)极大地影响了软件开发的高效性,这使得它也不再适合这个时代。人们不再像过往那样认同在 C++ 中兼容使用 C 语言的方法,认为这些工作只是在浪费时间,牺牲人们的努力。就在此时,Go 语言已经成功地解决了 C++ 中那些本打算解决却未能解决的关键问题。”

我非常想要向发明这门精湛的语言的 Go 开发团队表示真挚的感谢,尤其是团队的领导者 Rob Pike、Russ Cox 和 Andrew Gerrand,他们阐述的例子和说明都非常的完美。同时,我还要感谢 Miek Gieben、Frank Muller、Ryann Dolan 和 Satish V.J. 给予我巨大的帮助,还有那些 Golang-nuts 邮件列表里的所有的成员。

欢迎来到 Go 语言开发的奇妙世界!

1 Go 语言的起源,发展与普及

本章主要介绍 Go 语言的开发缘由，过程以及现状。

1.1 起源与发展

Go 语言起源 2007 年，并于 2009 年正式对外发布。它从 2009 年 9 月 21 日开始作为谷歌公司 20% 兼职项目，即相关员工利用 20% 的空余时间来参与 Go 语言的研发工作。该项目的三位领导者均是著名的 IT 工程师：Robert Griesemer，参与开发 Java HotSpot 虚拟机；Rob Pike，Go 语言项目总负责人，贝尔实验室 Unix 团队成员，参与的项目包括 Plan 9，Inferno 操作系统和 Limbo 编程语言；Ken Thompson，贝尔实验室 Unix 团队成员，C 语言、Unix 和 Plan 9 的创始人之一，与 Rob Pike 共同开发了 UTF-8 字符集规范。自 2008 年 1 月起，Ken Thompson 就开始研发一款以 C 语言为目标结果的编译器来拓展 Go 语言的设计思想。

这是一个由计算机领域“发明之父”所组成的黄金团队，他们对系统编程语言，操作系统和并行都有着非常深刻的见解



图1.1 Go语言设计者：Griesemer、Thompson 和 Pike

在 2008 年年中，Go 语言的设计工作接近尾声，一些员工开始以全职工作状态投入到这个项目的编译器和运行实现上。Ian Lance Taylor 也加入到了开发团队中，并于 2008 年 5 月创建了一个 gcc 前端。

Russ Cox 加入开发团队后着手语言和类库方面的开发，也就是 Go 语言的标准包。在 2009 年 10 月 30 日，Rob Pike 以 Google Techtalk 的形式第一次向人们宣告了 Go 语言的存在。

直到 2009 年 11 月 10 日，开发团队将 Go 语言项目以 BSD-style 授权（完全开源）正式公布在 Linux 和 Mac OS X 平台上的版本。Hector Chu 于同年 11 月 22 日公布了 Windows 版本。

作为一个开源项目，Go 语言借助开源社区的有生力量达到快速地发展，并吸引更多的开发者来使用并改善它。自该开源项目发布以来，超过 200 名非谷歌员工的贡献者对 Go 语言核心部分提交了超过 1000 个修改建议。在过去的 18 个月里，又有 150 开发者贡献了新的核心代码。这俨然形成了世界上最大的开源团队，并使该项目跻身 Ohloh 前 2% 的行列。大约在 2011 年 4 月 10 日，谷歌开始抽调员工进入全职开发 Go 语言项目。开源化的语言显然能够让更多的开发者参与其中并加速它的发展速度。Andrew Gerrard 在 2010 年加入到开发团队中成为共同开发者与支持者。

在 Go 语言在 2010 年 1 月 8 日被 Tiobe（闻名于它的编程语言流行程度排名）宣布为“2009 年年度语言”后，引起各界很大的反响。目前 Go 语言在这项排名中的最高记录是在 2010 年 2 月创下的第 13 名，流行程度 1778%。

时间轴：

- 2007 年 9 月 21 日：雏形设计
- 2009 年 11 月 10 日：首次公开发布
- 2010 年 1 月 8 日：当选 2009 年年度语言
- 2010 年 5 月：谷歌投入使用
- 2011 年 5 月 5 日：Google App Engine 支持 Go 语言

从 2010 年 5 月起，谷歌开始将 Go 语言投入到后端基础设施的实际开发中，例如开发用于管理后端复杂环境的项目。有句话叫“吃你自己的狗食”，这也体现了谷歌确实想要投资这门语言，并认为它是有生产价值的。

Go 语言的官方网站是 <https://golang.org>，这个站点采用 Python 作为前端，并且使用 Go 语言自带的工具 godoc 运行在 Google App Engine 上来作为 Web 服务器提供文本内容。在官网的首页有一个功能叫做 Go-playground，是一个 Go 代码的简单编辑器的沙盒，它可以在没有安装 Go 语言的情况下在你的浏览器中编译并运行 Go，它提供了一些示例，其中包括国际惯例“Hello, World!”。

更多的信息详见 <http://code.google.com/p/go/>，Go 项目 Bug 追踪和功能预期详见 <http://code.google.com/p/go/issues/list>。

Go 通过以下的 Logo 来展示它的速度，并以囊地鼠（Gopher）作为它的吉祥物。

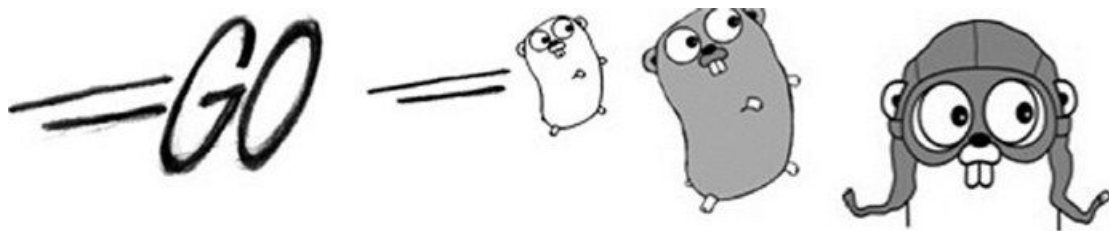


图1.2 Go 语言 Logo

谷歌邮件列表 [golang-nuts](#) 非常活跃，每天的讨论和问题解答数以百计。

关于 Go 语言在 Google App Engine 的应用，这里有一个单独的邮件列表 [google-appengine-go](#)，不过 2 个邮件列表的讨论内容并不是分得很清楚，都会涉及到相关的话题。[go-lang.cat-v.org/](#) 是 Go 语言开发社区的资源站，[irc.freenode.net](#) 的 #go-nuts 是官方的 Go IRC 频道。

http://twitter.com/#!/go_nuts 是 Go 语言在 Twitter 的官方帐号，大家一般使用 #golang 作为话题标签。

这里还有一个在 Linked-in 的小组：http://www.linkedin.com/groups?gid=2524765&trk=myg_ugrp_ovr。

Go 编程语言的维基百科：[http://en.wikipedia.org/wiki/Go_\(programming_language\)](http://en.wikipedia.org/wiki/Go_(programming_language))

Go 语言相关资源的搜索引擎页面：<http://go-lang.cat-v.org/go-search>

Go 语言还有一个运行在 Google App Engine 上的 [Go Tour](#)，你也可以通过执行命令 `go install go-tour.googlecode.com/hg/gotour` 安装到你的本地机器上。对于中文读者，可以访问该指南的[中文版本](#)，或通过命令 `go install https://bitbucket.org/mikespook/go-tour-zh/gotour` 进行安装。

1.2 语言的主要特性与发展的环境和影响因素

正如“21 世纪的 C 语言”这句话所说，Go 语言并不是凭空而造的，而是和 C++、Java 和 C# 一样属于 C 系。不仅如此，设计者们还汲取了其它编程语言的精粹部分融入到 Go 语言当中。

在声明和包的设计方面，Go 语言受到 Pascal、Modula 和 Oberon 系语言的影响；在并发原理的设计上，Go 语言从同样受到 Tony Hoare 的 CSP（通信序列进程 *Communicating Sequential Processes*）理论影响的 Limbo 和 Newsqueak 的实践中借鉴了一些经验，并使用了和 Erlang 类似的机制。

这是一门完全开源的编程语言，因为它使用 BSD 授权许可，所以任何人都可以进行商业软件的开发而不需要支付任何费用。

尽管为了能够让目前主流的开发者们能够对 Go 语言中的类 C 语言的语法感到非常亲切而易于转型，但是它在极大程度上简化了这些语法，使得它们比 C/C++ 的语法更加简洁和干净。同时，Go 语言也拥有一些动态语言的特性，这使得使用 Python 和 Ruby 的开发者们在使用 Go 语言的时候感觉非常容易上手。

下图展示了一些其它编程语言对 Go 语言的影响：

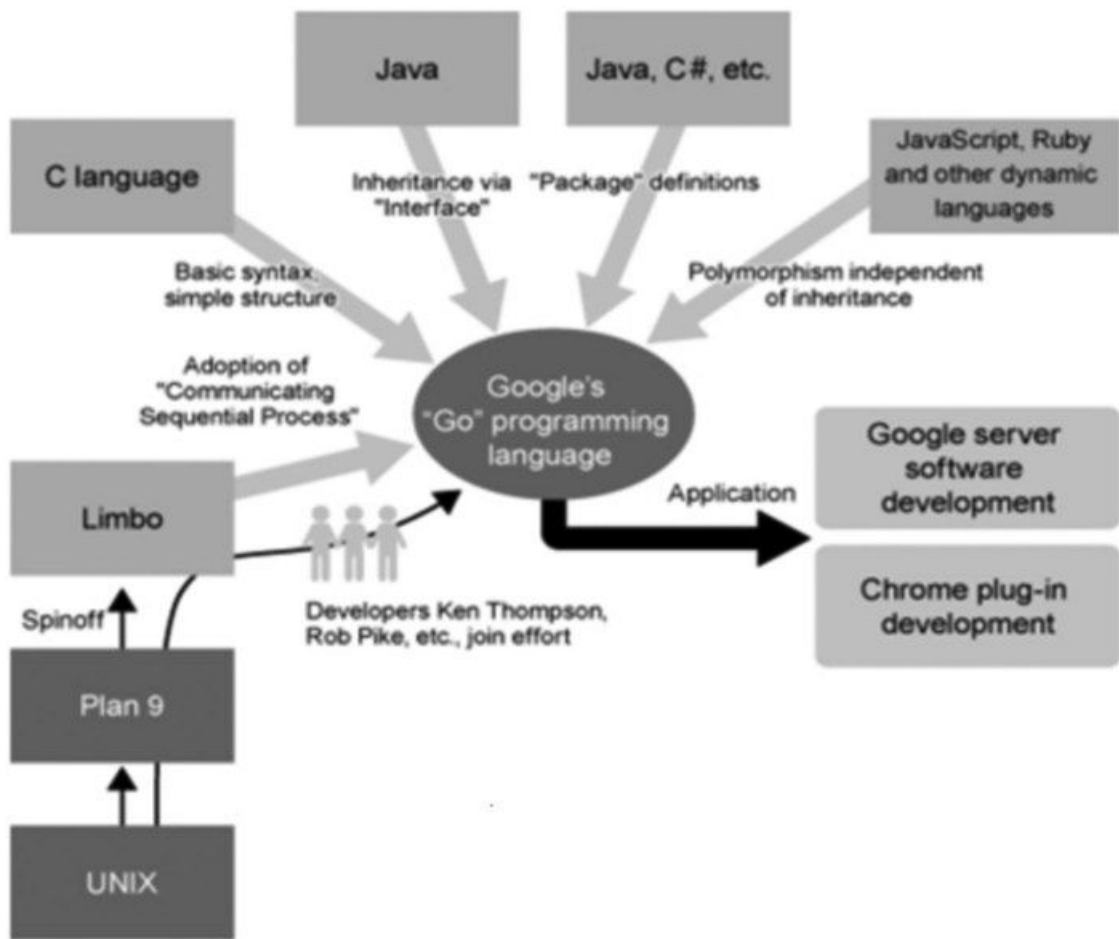


图1.3 其它编程语言对 Go 语言的影响

1.2.2 为什么要创造一门编程语言

- C/C++的发展速度无法跟上计算机发展的脚步，十多年来也没有出现一门与时代相符的主流系统编程语言，因此人们需要一门新的系统编程语言来弥补这个空缺，尤其是在计算机信息时代。
- 对比计算机性能的提升，软件开发领域不被认为发展地足够快或者比硬件发展更加成功（有许多项目均以失败告终），同时应用程序的体积始终在不断地扩大，这就迫切地需要一门具备更高层次概念的低级语言来突破现状。
- 在 Go 语言出现之前，开发者们总是面临非常艰难的抉择，究竟是使用执行速度快但是编译速度并不理想的语言（如：C++），还是使用编译速度较快但执行效率不佳的语言（如：.NET、Java），或者说开发难度较低但执行速度一般的动态语言呢？显然，Go 语言在这 3 个条件之间做到了最佳的平衡：快速编译，高效执行，易于开发。

1.2.3 Go 语言的发展目标

Go 语言的主要目标是将静态语言的安全性和高效性与动态语言的易开发性进行有机结合，达到完美平衡，从而使编程变得更加有趣，而不是在艰难抉择中痛苦前行。

因此，Go 语言是一门类型安全和内存安全的编程语言。虽然 Go 语言中仍有指针的存在，但并不允许进行指针运算。

Go 语言的另一个目标对于网络通信、并发和并行编程的极佳支持，从而更好地利用大量的分布式和多核的计算机，这一点对于谷歌内部的使用来说就非常重要了。设计者通过 goroutine 这种轻量级线程的概念来实现这个目标，然后通过 channel 来实现各个 goroutine 之间的通信。他们实现了分段栈增长和 goroutine 在线程基础上多路复用技术的自动化。

这个特性显然是 Go 语言最强有力的部分，不仅支持了日益重要的多核与多处理器计算机，也弥补了现存编程语言在这方面所存在的不足。

Go 语言中另一个非常重要的特性就是它的构建速度（编译和链接到机器代码的速度），一般情况下构建一个程序的时间只需要数百毫秒到几秒。作为大量使用 C++ 来构建基础设施的谷歌来说，无疑从根本上摆脱了 C++ 在构建速度上非常不理想的噩梦。这不仅极大地提升了开发者的生产力，同时也使得软件开发过程中的代码测试环节更加紧凑，而不必浪费大量的时间在等待程序的构建上。

依赖管理是现今软件开发的一个重要组成部分，但是 C 语言中“头文件”的概念却导致越来越多因为依赖关系而使得构建一个大型的项目需要长达几个小时的时间。人们越来越需要一门具有严格的、简洁的依赖关系分析系统从而能够快速编译的编程语

言。这正是 Go 语言采用包模型的根本原因，这个模型通过严格的依赖关系检查机制来加快程序构建的速度，提供了非常好的可量测性。

整个 Go 语言标准库的编译时间一般都在 20 秒以内，其它的常规项目也只需要半秒钟的时间来完成编译工作。这种闪电般的编译速度甚至比编译 C 语言或者 Fortran 更加快，使得编译这一环节不再成为在软件开发中困扰开发人员的问题。在这之前，动态语言将快速编译作为自身的一大亮点，像 C++ 那样的静态语言一般都有非常漫长的编译和链接工作。而同样作为静态语言的 Go 语言，通过自身优良的构建机制，成功地去除了这个弊端，使得程序的构建过程变得微不足道，拥有了像脚本语言和动态语言那样的高效开发的能力。

另外，Go 语言在执行速度方面也可以与 C/C++ 相提并论。

由于内存问题（通常称为内存泄漏）长期以来一直伴随着 C++ 的开发者们，Go 语言的设计者们认为内存管理不应该是开发人员所需要考虑的问题。因此尽管 Go 语言像其它静态语言一样执行本地代码，但它依旧运行在某种意义上的虚拟机，以此来实现高效快速的垃圾回收（使用了一个简单的标记-清除算法）。

尽管垃圾回收并不容易实现，但考虑这将是未来并发应用程序发展的一个重要组成部分，Go 语言的设计者们还是完成了这项艰难的任务。

Go 语言还能够运行时进行反射相关的操作。

使用 `go install` 能够很轻松地对外部包进行部署。

此外，Go 语言还支持调用由 C 语言编写的海量库文件（第 3.9 节），从而能够将过去开发的软件进行快速迁移。

1.2.4 指导设计原则

Go 语言通过减少关键字的数量（25 个）来简化编码过程中的混乱和复杂度。干净、整齐和简洁的语法也能够提高程序的编译速度，因为这些关键字在编译过程中少到甚至不需要符号表来协助解析。

这些方面的工作都是为了减少编码的工作量，甚至可以与 Java 的简化程度相比较。

Go 语言有一种极简抽象艺术家的感觉，因为它只提供了一到两种方法来解决某个问题，这使得开发者们的代码都非常容易阅读和理解。众所周知，代码的可读性是软件工程里最重要的一部分（译者注：代码是写给人看的，不是写给机器看的）。

这些设计理念没有建立其它概念之上，所以并不会因为牵扯到一些概念而将某个概念复杂化，他们之间是相互独立的。

Go 语言有一套完整的编码规范，你可以在 [Go 语言编码规范](#) 页面进行查看。

它不像 Ruby 那样通过实现过程来定义编码规范。作为一门具有明确编码规范的语言，它要求可以采用不同的编译器如 `gc` 和 `gccgo`（第 2.1 节）进行编译工作，这对语言本身拥有更好的编码规范起到很大帮助。

[LALR](http://en.wikipedia.org/wiki/LALR_parser) 是 Go 语言的语法标准，你也可以在 `src/cmd/gc/go.y` 中查看到，这种语法标准在编译时不需要符号表来协助解析。

1.2.5 语言的特性

Go 语言从本质上（程序和结构方面）来实现并发编程。

因为 Go 语言没有类和继承的概念，所以它和 Java 或 C++ 看起来并不相同。但是它通过接口（`interface`）的概念来实现多态性。Go 语言有一个清晰易懂的轻量级类型系统，在类型之间也没有层级之说。因此可以说这是一门混合型的语言。

在传统的面向对象语言中，使用面向对象编程技术显得非常的臃肿，它们总是通过复杂的模式来构建庞大的类型层级，这违背了编程语言应该提升生产力的宗旨。

函数是 Go 语言中的基本构件，它们的使用方法非常灵活。在第六章，我们会看到 Go 语言在函数式编程方面的基本概念。

Go 语言使用静态类型，所以它是类型安全的一门语言，加上通过构建到本地代码，程序的执行速度也非常快。

作为强类型语言，隐式的类型转换是不被允许的，记住一条原则：让所有的东西都是显式的。

Go 语言其实也有一些动态语言的特性（通过关键字 `var`），所以它对那些逃离 Java 和 .Net 世界而使用 Python、Ruby、PHP 和 JavaScript 的开发者们也具有很大的吸引力。

Go 语言支持交叉编译，比如说你可以在运行 Linux 系统的计算机上开发运行下 Windows 下运行的应用程序。这是第一门完全支持 UTF-8 的编程语言（译者注：.NET 好像也支持吧？），这不仅体现在它可以处理使用 UTF-8 编码的字符串，就连它的源码文件格式都是使用的 UTF-8 编码。Go 语言做到了真正的国际化！

1.2.6 语言的用途

Go 语言被设计成一门应用于搭载 Web 服务器，存储集群或类似用途的巨型中央服务器的系统编程语言。对于高性能分布式系统领域而言，Go 语言无疑比大多数其它语言有着更高的开发效率。它提供了海量并行的支持，这对于游戏服务端的开发而言是再好不过了。

Go 语言一个非常好的目标就是实现所谓的复杂事件处理（CEP），这项技术要求海量并行支持，高度的抽象化和高性能。当我们进入到物联网时代，CEP 必然会成为人们关注的焦点。

但是 Go 语言同时也是一门可以用于实现一般目标的语言，例如对于文本的处理，前端展现，甚至像使用脚本一样使用它。

值得注意的是，因为垃圾回收和自动内存分配的原因，Go 语言不适合用来开发对实时性要求很高的软件。

越来越多的谷歌内部的大型分布式应用程序都开始使用 Go 语言来开发，例如谷歌地球的一部分代码就是由 Go 语言完成的。

如果你想知道一些其它组织使用 Go 语言开发的实际应用项目，你可以到这个页面进行查看：<http://go-lang.cat-v.org/organizations-using-go>。出于隐私保护的考虑，许多公司的项目都没有展示在这个页面。我们将会在第 21 章讨论到一个使用 Go 语言开发的大型存储区域网络（SAN）案例。

在 Chrome 浏览器中内置了一款 Go 语言的编译器用于本地客户端（NaCl（译者注：为什么我觉得这是“氯化钠”？）），这很可能会被用于在 Chrome OS 中执行 Go 语言开发的应用程序。

Go 语言可以在 Intel 或 ARM 处理器上运行，因此它也可以在安卓系统下运行，例如 Nexus 系列的产品。

在 Google App Engine 中使用 Go 语言：2011年5月5日，官方发布了用于开发运行在 Google App Engine 上的 Web 应用的 Go SDK，在此之前，开发者们只能选择使用 Python 或者 Java。这主要是 David Symonds 和 Nigel Tao 努力的成果。目前最新的稳定版是基于 r60.3 的 SDK 1.6.1，于 2011 年 12 月 13 日发布。当前 Go 语言的稳定版本是 Go 1（译者注：目前最新的稳定版是 Go1.1）。

1.2.7 关于特性丢失

许多能够在大多数面向对象语言中使用的特性 Go 语言都没有支持，但其中的一部分可能会在未来被支持。

- 为了简化设计，不支持函数重载和操作符重载
- 为了避免在 C/C++ 开发中的一些 Bug 和混乱，不支持隐式转换
- Go 语言通过另一种途径实现面向对象设计（第 10 - 11 章）来放弃类和类型的继承
- 尽管在接口的使用方面（第 11 章）可以实现类似变体类型的功能，但本身不支持变体类型
- 不支持动态加载代码
- 不支持动态链接库
- 不支持泛型
- 通过 recover 和 panic 来替代异常机制（第 13.2 - 3 节）
- 不支持断言
- 不支持静态变量

关于 Go 语言开发团队对于这些方面的讨论，你可以通过这个页面查看：http://golang.org/doc/go_faq.html

1.2.8 使用 Go 语言编程

如果你有其它语言的编程经历（面向对象编程语言，如：Java、C#、Object-C、Python、Ruby），在你进入到 Go 语言的世界之后，你将会像迷恋你的 X 语言一样无法自拔。Go 语言使用了与其它语言不同的设计模式，所以当你尝试将你的 X 语言的代码迁移到 Go 语言时，你将会非常失望，所以你需要从头开始，用 Go 的理念来思考。

如果你在至高点使用 Go 的理念来重新审视和分析一个问题，你通常会找到一个适用于 Go 语言的优雅解决方案。

1.2.9 小结

这里列举一些 Go 语言的必杀技：

- 简化问题，易于学习
- 内存管理，简洁语法，易于使用
- 快速编译，高效开发
- 高效执行
- 并发支持，轻松驾驭
- 静态类型
- 标准类库，规范统一
- 易于部署
- 文档全面
- 免费开源

2 安装与运行环境

本章主要介绍 Go 语言的开发环境的搭建以及开发工具的选择。

2.1 平台与架构

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

Go 语言开发团队开发了适用于以下操作系统的编译器：

- Linux
- FreeBSD
- Mac OS X（也称为 Darwin）

目前有2个版本的编译器：Go 原生编译器 gc 和非原生编译器 gccgo，这两款编译器都是在类 Unix 系统下工作。其中，gc 版本的编译器已经被移植到 Windows 平台上，并集成在主要发行版中，你也可以通过安装 MinGW 从而在 Windows 平台下使用 gcc 编译器。这两个编译器都是以单通道的方式工作。

你可以获取以下平台上的 Go 1.1 源码和二进制文件：

- FreeBSD 7+：amd64 和 386 架构
- Linux 2.6+：amd64、386 和 arm 架构
- Mac OS X（Snow Leopard + Lion）：amd64 和 386 架构
- Windows 2000+：amd64 和 386 架构

对于非常底层的纯 Go 语言代码或者包而言，在各个操作系统平台上的可移植性是非常强的，只需要将源码拷贝到相应平台上进行编译即可，或者可以使用交叉编译来构建目标平台的应用程序（第2.2节）。但如果你打算使用 cgo 或者类似文件监控系统的软件，就需要根据实际情况进行相应地修改了。

1. Go 原生编译器 gc：

主要基于 Ken Thompson 先前在 Plan9 操作系统上使用的 C 工具链。

Go 语言的编译器和链接器都是使用 C 语言编写并产生本地代码，Go 不存在自我引导之类的功能。因此如果使用一个有不同指令集的编译器来构建 Go 程序，就需要针对操作系统和处理器架构（32 位操作系统或 64 位操作系统）进行区别对待。

这款编译器使用非分代、无压缩和并行的方式进行编译，它的编译速度要比 gccgo 更快，产生更好的本地代码，但编译后的程序不能够使用 gcc 进行链接。

编译器目前支持以下基于 Intel 或 AMD 处理器架构的程序构建。

<i>No of bits</i>	<i>Processor name</i>	<i>Compiler</i>	<i>Linker</i>
64 bit implementation	amd64 (also named x86-64)	6g	6l
32 bit implementation	386 (also named x86 or x86-32)	8g	8l
32 bit RISC implementation	arm (ARM)	5g	5l

图2.1 gc 编译器支持的处理器架构

当你第一次看到这套命名系统的时候你会觉得很奇葩，不过这些命名都是来自于 Plan9 项目。

g = 编译器：将源代码编译为项目代码（程序文本）

l = 链接器：将项目代码链接到可执行的二进制文件（机器代码）

（相关的 C 编译器名称为 6c、8c 和 5c，相关的汇编器名称为 6a、8a 和 5a）

标记（Flags） 是指可以通过命令行设置可选参数来影响编译器或链接器的构建过程或得到一个特殊的目标结果。

可用的编译器标记如下：

```
flags:
-I 针对包的目录搜索
-d 打印声明信息
-e 不限制错误打印的个数
-f 打印栈结构
-h 发生错误时进入恐慌（panic）状态
-o 指定输出文件名 // 详见第3.4节
-S 打印产生的汇编代码
-V 打印编译器版本 // 详见第2.3节
-u 禁止使用 unsafe 包中的代码
-w 打印归类后的语法解析树
-x 打印 lex tokens
```

从 Go 1.0.3 版本开始，不再使用 8g, 8l 之类的指令进行程序的构建，取而代之的是统一的 go build 和 go install 等命令，而这些指令会自动调用相关的编译器或链接器。

如果你想获得更深层次的信息，你可以在目录 \$GOROOT/src/cmd 下找到编译器和链接器的源代码。Go 语言本身是由 C 语言开发的，而不是 Go 语言。词法分析程序是 GNU bison，语法分析程序是名为 \$GOROOT/src/cmd/gc/go.y 的 yacc 文件，它会在同一目录输出 y.tab.{c,h} 文件。如果你想知道更多有关构建过程的信息，你可以查看相同目录下的 Makefile 文件，另一个版本的构建过程的概述可以在 \$GOROOT/src/make.bash 中找到。

大部分的目录都包含了名为 doc.go 的文件，这个文件提供了更多详细的信息。

2. gccgo 编译器：

一款相对于 gc 而言更加传统的编译器，使用 GCC 作为后端。GCC 是一款非常流行的 GNU 编译器，它能够构建基于众多处理器架构的应用程序。编译速度相对 gc 较慢，但产生的本地代码运行要稍微快一点。它同时也提供一些与 C 语言之间的互操作性。

从 Go 1 版本开始，gc 和 gccgo 在编译方面都有等价的功能。

3. 文件扩展名与包（package）：

Go 语言源文件的扩展名很显然就是 .go 。

C 文件使用后缀名 .c，汇编文件使用后缀名 .s。所有的源代码文件都是通过包（packages）来组织。包含可执行代码的包文件在被压缩后使用扩展名 .a（AR 文档）。

Go 语言的标准库（第9.1节）包文件在被安装后就是使用这种格式的文件。

注意： 当你在创建目录时，文件夹名称永远不应该包含空格，而应该使用下划线"_"或者其它一般符号代替。

2.2 Go 环境变量

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

Go 开发环境依赖于一些操作系统环境变量，你最好在安装 Go 之间就已经设置好他们。如果你使用的是 Windows 的话，你完全不用进行手动设置，Go 将被默认安装在目录 c:/go 下。这里列举几个最为重要的环境变量：

- **\$GOROOT** 表示 Go 在你的电脑上的安装位置，它的值一般都是 \$HOME/go，当然，你也可以安装在别的地方。
- **\$GOARCH** 表示目标机器的处理器架构，它的值可以是 386, amd64 或 arm。
- **\$GOOS** 表示目标机器的操作系统，它的值可以是 darwin, freebsd, linux 或 windows
- **\$GOBIN** 表示编译器和链接器的安装位置，默认是 \$GOROOT/bin，如果你使用的是 Go 1.0.3 及以后的版本，一般情况下你可以将它的值设置为空，Go 将会使用前面提到的默认值。

目标机器是指你打算运行你的 Go 应用程序的机器。

Go 编译器支持交叉编译，也就是说你可以在一台机器上构建运行在具有不同操作系统和处理器架构上运行的应用程序，也就是说编写源代码的机器可以和目标机器有完全不同的特性（操作系统与处理器架构）。

为了区分本地机器和目标机器，你可以使用 \$GOHOSTOS 和 \$GOHOSTARCH 设置目标机器的参数，这两个变量只有在进行交叉编译的时候才会用到，如果你不进行显示设置，他们的值会和本地机器（\$GOOS 和 \$GOARCH）一样。

- **\$GOPATH** 默认采用和 \$GOROOT 一样的值，但从 Go 1.1 版本开始，你必须修改为其它路径。它可以包含多个包含 Go 语言源码文件、包文件和可执行文件的路径，而这些路径下又必须分别包含三个规定的目录：src，pkg 和 bin，这三个目录分别用于存放源码文件、包文件和可执行文件。

- **\$GOARM** 专门针对基于 arm 架构的处理器，它的值可以是 5 或 6，默认为 6。
- **\$GOMAXPROCS** 用于设置应用程序可使用的处理器个数与核数，详见第14.1.3节

在接下来的章节中，我们将会讨论如何在 Linux，Mac OS X 和 Windows 上安装 Go 语言。在 FreeBSD 上的安装和 Linux 非常类似。开发团队正在尝试将 Go 语言移植到其它例如 OpenBSD，DragonFlyBSD，NetBSD，Plan 9，Haiku 和 Solaris 操作系统上，你可以在这个页面找到最近的动态：<http://go-lang.cat-v.org/os-ports>

2.3 在 Linux 上安装 Go

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

如果你能够自己下载并编译 Go 的源代码来说是非常有教育意义的，你可以根据这个页面找到安装指南和下载地址：<http://golang.org/doc/install.html>。

我们接下来也会带你一步步的完成安装过程。

1. 设置 Go 环境变量

我们在 Linux 系统下一般通过文件 `$HOME/.bashrc` 配置自定义环境变量，根据不同的发行版也可能是文件 `$HOME/.profile`，然后使用 `gedit` 或 `vi` 来编辑文件内容。

```
export GOROOT=$HOME/go
export GOBIN=$GOROOT/bin
export GOARCH=386
export GOOS=linux
```

（译者注：目前的 Go 版本一般情况下已不需要设置 `$GOBIN`）

为了确保相关文件在文件系统的任何地方都能被调用，你还需要添加以下内容：

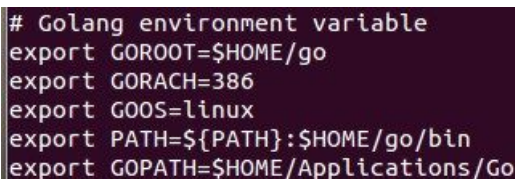
```
export PATH=$GOROOT/bin:$PATH
```

在开发 Go 项目时，你还需要一个环境变量来保存你的工作目录。

```
export GOPATH=$HOME/Applications/Go
```

`$GOPATH` 可以包含多个工作目录，取决于你的个人情况。如果你设置了多个工作目录，那么当你在之后使用 `go get`（远程包安装命令）时远程包将会被安装在第一个目录下。

在完成这些设置后，你需要在终端输入指令 `source .bashrc` 以使这些环境变量生效。然后重启终端，输入 `go env` 和 `env` 来检查环境变量是否设置正确。

A terminal window with a dark background and light-colored text. It displays the output of the 'go env' command, showing various environment variables for Go, including GOROOT, GOARCH, GOOS, PATH, and GOPATH.

```
# Golang environment variable
export GOROOT=$HOME/go
export GOARCH=386
export GOOS=linux
export PATH=${PATH}:$HOME/go/bin
export GOPATH=$HOME/Applications/Go
```

图2.2 在终端输入 `go env` 后打印的信息

2. 安装 C 工具

Go 的工具链是用 C 语言编写的，因此在安装 Go 之前你需要先安装相关的 C 工具。如果你使用的是 Ubuntu 的话，你可以在终端输入以下指令（译者注：由于网络环境的特殊性，你可能需要将每个工具分开安装）。

```
sudo apt-get install bison ed gawk gcc libc6-dev make
```

你可以在其它发行版上使用 RPM 之类的工具。

3. 安装 Mercurial

Go 源代码是通过 Mercurial 来进行版本管理的，因此你必须在你下载 Go 源代码之前安装这款工具。你可以使用指令 `hg` 来检查你的计算机上是否已经安装该工具，如果没有，请使用以下指令来完成安装：


```
sudo apt-get install python-setuptools
sudo apt-get install python-dev
sudo apt-get install build-essential
sudo apt-get install mercurial
```

4. 获取 Go 源代码

通过以下指令来从服务器获取 Go 源代码到你的计算机上，这里我们直接使用 \$GOROOT 的值，在获取源代码之前，你不能手动创建相关目录，否则将导致获取失败。

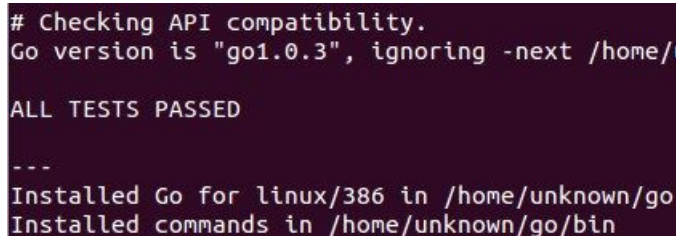
```
hg clone -r release https://go.googlecode.com/hg/ $GOROOT
```

5. 构建 Go

在终端使用以下指令来进行编译工作。

```
cd $GOROOT/src
./all.bash
```

在完成编译之后（通常在 1 分钟以内，如果你在 B 型树莓派上编译，一般需要 1 个小时），你会在终端看到如下信息被打印：

A terminal window with a dark background and light-colored text. The text shows the Go compiler checking API compatibility, reporting the version as 'go1.0.3', and stating 'ALL TESTS PASSED'. It then shows the installation path for Linux/386 as '/home/unknown/go' and the commands as '/home/unknown/go/bin'.

```
# Checking API compatibility.
Go version is "go1.0.3", ignoring -next /home/...

ALL TESTS PASSED

---
Installed Go for linux/386 in /home/unknown/go
Installed commands in /home/unknown/go/bin
```

图2.3 完成编译后在终端打印的信息

注意事项

如果你在编译过程中发生错误，请尝试使用指令 `hg pull -u` 来更新源代码，然后进行第 5 步。

注意事项

在测试 net/http 包时有一个测试会尝试连接 google.com，你可能会看到如下所示的一个无厘头的错误报告：

```
'make[2]: Leaving directory `/localusr/go/src/pkg/net'
```

如果你正在使用一个带有防火墙的机器，我建议你可以在编译过程中暂时关闭防火墙，以避免不必要的错误。

解决这个问题的另一个办法是通过设置环境变量 `$DISABLE_NET_TESTS` 来告诉构建工具忽略 net/http 包的相关测试：

```
export DISABLE_NET_TESTS=1
```

如果这样还是没有办法阻止错误的发生的话，你可以通过添加 net 包到忽略测试列表，这个列表在 \$GOROOT/src/pkg 下的 Makefile 中。

如果你完全不想运行包的测试，你可以直接运行 `./make.bash` 来进行单纯的构建过程。

6. 测试安装

使用你最喜爱的编辑器来输入以下内容，并保存为文件名 test.go。

Example 2.1 [hello_world1.go](#)

```
package main
```

```
func main() {  
    println("Hello", "world")  
}
```

切换相关目录到下，然后执行指令 `go run hello_world1.go`，将会打印信息：Hello, world。

7. 验证安装版本

你可以通过在终端输入指令 `go version` 来打印 Go 的版本信息。

如果你想要通过 Go 代码在运行时检测版本，可以通过以下例子实现。

Example 2.2 [version.go](#)

```
package main  
  
import (  
    "fmt"  
    "runtime"  
)  
  
func main() {  
    fmt.Printf("%s", runtime.Version())  
}
```

这段代码将会输出 `go1.0.3` 或 `go1.1`。

8. 更新源代码版本

在终端输入以下指令来完成源代码的更新：

```
cd $GOROOT  
hg pull  
hg update release  
cd src  
sudo ./all.bash
```

你可以在这个页面查看到最新的稳定版：<http://golang.org/doc/devel/release.html>

当前最新的稳定版 Go 1 系列于 2012 年 3 月 28 日发布。

Go 的源代码有以下三个分支：

- Go release: 最新稳定版，实际开发最佳选择
- Go weekly: 包含最近更新的版本，一般每周更新一次
- Go tip: 永远保持最新的版本，相当于内测版

你可以通过 `gofix` 这个工具来进行 Go 源代码的版本迁移，将旧版本的代码升级到最新版。

当你在使用不同的版本时，注意官方博客发布的信息，因为你所查阅的文档可能和你正在使用的版本不相符。

2.4 在 Mac OS X 上安装 Go

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

如果你想要在你的 Mac 系统上安装 Go，则必须使用 Intel 64 位处理器，Go 不支持 PowerPC 处理器。

你可以通过该页面查看有关在 PowerPC 处理器上的移植进度：<https://codetr-go-ppc.googlecode.com/hg/>。

注意事项

在 Mac 系统下使用到的 C 工具链是 Xcode 的一部分，因此你需要通过安装 Xcode 来完成这些工具的安装。你并不需要安装完整的 Xcode，而只需要安装它的命令行工具部分。

你可以在这个页面下载到 Mac 系统下的一键安装包或源代码自行编译：<https://code.google.com/p/go/downloads/list>

通过源代码编译安装的过程与环境变量的配置与在 Linux 系统非常相似，因此不再赘述。

2.5 在 Windows 上安装 Go

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

你可以在这个页面下载到 Windows 系统下的一键安装包：<http://code.google.com/p/go/downloads/list>

前期的 Windows 移植工作由 Hector Chu 完成，但目前的发行版已经由 Joe Poirier 全职维护。

在完成安装包的安装之后，你只需要配置 `$GOPATH` 这一个环境变量就可以开始使用 Go 语言进行开发了，其它的环境变量安装包均会进行自动设置。在默认情况下，Go 将会被安装在目录 `c:\go` 下，但如果你在安装过程中修改安装目录，则可能需要手动修改所有的环境变量的值。

如果你想要测试安装，则可以使用指令 `go run` 运行 `hello_world1.go`。

如果发生错误 `fatal error: can't find import: fmt` 则说明你的环境变量没有配置正确。

如果你想要在 Windows 下使用 `cgo`（调用 C 语言写的代码），则需要安装 [MinGW](#)，如果你使用的是 64 位操作系统，请务必安装 64 位版本的 MinGW。安装完成进行环境变量等相关配置即可使用。

在 Windows 下运行在虚拟机里的 Linux 系统上安装 Go：

如果你想要在 Windows 下的虚拟机里的 Linux 系统上安装 Go，你可以选择使用虚拟机软件 [VMware](#)，下载 [VMware player](#)，搜索并下载一个你喜欢的 Linux 发行版镜像，然后安装到虚拟机里，安装 Go 的流程参考第 2.3 节中的内容。

2.6 安装目录清单

你的 Go 安装目录（`$GOROOT`）的文件夹结构应该如下所示：

README, AUTHORS, CONTRIBUTORS, LICENSE

- `\bin` 包含可执行文件，如：编译器，Go 工具
- `\doc` 包含示例程序，代码工具，本地文档等
- `\include` 包含 C/C++ 头文件
- `\lib` 包含文档模版
- `\misc` 包含与支持 Go 编辑器有关的配置文件以及 `cgo` 的示例
- `\pkg\os_arch` 包含标准库的包的对象文件（.a）
- `\src` 包含源代码构建脚本
- `\src\cmd` 包含 Go 和 C 的编译器和命令行脚本
- `\src\lib9 \src\libbio \src\libmach` 包含 C 文件
- `\src\pkg` 包含 Go 标准库的包的完整源代码（Go 是一门开源语言）

2.7 Go 类虚拟机（runtime）

尽管 Go 编译器产生的是本地可执行代码，这些代码仍旧运行在 Go 的 `runtime`（这部分的代码可以在 `runtime` 包中找到）当中。这个 `runtime` 类似 Java 和 .NET 语言所用到的虚拟机，它负责管理包括内存分配、垃圾回收（第 10.8 节）、栈处理、goroutine、channel、切片（slice）、map 和反射（reflection）等等。

`runtime` 主要由 C 语言编写，并且是每个 Go 包的最顶级包。你可以在目录 `$GOROOT/src/pkg/runtime/` 中找到相关内容（主要看 `mgc0.c` 和其它以 `m` 开头的文件）。

垃圾回收器 Go 拥有简单却高效的标记-清除回收器。它的主要思想来源于 IBM 的可复用垃圾回收器，旨在打造一个高效、低延迟的并发回收器。目前 `gccgo` 还没有回收器，同时适用 `gc` 和 `gccgo` 的新回收器正在研发中。使用一门具有垃圾回收功能的编程语言不代表你可以避免内存分配所带来的问题，分配和回收内容都是消耗 CPU 资源的一种行为。

Go 的可执行文件都比相对应的源代码文件要大很多，这恰恰说明了 Go 的 `runtime` 嵌入到了每一个可执行文件当中。当然，在部署到数量巨大的集群时，较大的文件体积也是比较头疼的问题。但总得来说，Go 的部署工作还是要比 Java 和 Python 轻松得多。因为 Go 不需要依赖任何其它文件，它只需要一个单独的静态文件，这样你也不会像使用其它语言一样在各种不同版本的依赖文件之间混淆。

2.8 Go 解释器

因为 Go 具有像动态语言那样快速编译的能力，自然而然地就有人会问 Go 语言能否在 REPL（real-eval-print loop）编程环境下实现。Sebastien Binet 已经使用这种环境实现了一个 Go 解释器，你可以在这个页面找到：<https://bitbucket.org/binet/igo>

3 编辑器、集成开发环境与其它工具

因为 Go 语言还是一门相对年轻的编程语言，所以不管是在集成开发环境（IDE）还是相关的插件方面，发展都不是很成熟。不过目前还是有一些 IDE 能够较好地支持 Go 的开发，有些开发工具甚至是跨平台的，你可以在 Linux、Mac OS X 或者 Windows 下工作。

你可以通过查阅该页面来获取 Go 开发工具的最新信息：<http://go-lang.cat-v.org/text-editors/>

3.1 Go 开发环境的基本要求

这里有一个你可以期待你用来开发 Go 的集成开发环境有哪些特性的列表，从而替代你使用文本编辑器写代码和命令行编译与链接程序的方式。

1. 语法高亮是必不可少的功能，这也是为什么每个开发工具都提供配置文件来实现自定义配置的原因。
2. 可以自动保存代码，至少在每次编译前都会保存。
3. 可以显示代码所在的行数。
4. 拥有较好的项目文件纵览和导航能力，可以同时编辑多个源文件并设置书签，能够匹配括号，能够跳转到某个函数或类型的定义部分。
5. 完美的查找和替换功能，替换之前最好还能预览结果。
6. 可以注释或取消注释选中的一行或多行代码。
7. 当有编译错误时，双击错误提示可以跳转到发生错误的位置。
8. 跨平台，能够在 Linux、Mac OS X 和 Windows 下工作，这样就可以专注于一个开发环境。
9. 最好是免费的，不过有些开发者还是希望能够通过支付一定金额以获得更好的开发环境。
10. 最好是开源的。
11. 能够通过插件架构来轻易扩展和替换某个功能。
12. 尽管集成开发环境本身就是非常复杂的，但一定要让人感觉操作方便。
13. 能够通过代码模版来简化编码过程从而提升编码速度。
14. 使用 Go 项目的概念来浏览和管理项目中的文件，同时还要拥有构建系统的概念，这样才能更加方便的构建、清理或运行我们建立的程序或项目。构建出的程序需要能够通过命令行或 IDE 内部的控制台运行。
15. 拥有断点、检查变量值、单步执行、逐过程执行标识库中代码的能力。
16. 能够方便的存取最近使用过的文件或项目。
17. 拥有对包、类型、变量、函数和方法的智能代码补全的功能。
18. 能够对项目或包中的代码建立抽象语法树视图（AST-view）。
19. 内置 Go 的相关工具。
20. 能够方便完整地查阅 Go 文档。
21. 能够方便地在不同的 Go 环境之间切换。
22. 能够导出不同格式的代码文件，如：PDF，HTML 或格式化后的代码。
23. 针对一些特定的项目有项目模板，如：Web 应用，App Engine 项目，从而能够更快地开始开发工作。
24. 具备代码重构的能力。
25. 集成像 hg 或 git 这样的版本控制工具。
26. 集成 Google App Engine 开发及调试的功能。

3.2 编辑器和集成开发环境

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

这些编辑器包含了代码高亮和其它与 Go 有关的一些使用工具：Emacs、Vim、Xcode3、KDE Kate、TextWrangler、BBEdit、McEdit、TextMate、TextPad、JEdit、SciTE、Nano、Notepad++、Geany、SlickEdit 和 Sublime Text 2。

你可以将 Linux 的文本编辑器 GEdit 改造成一个很好的 Go 开发工具，详见页面：<http://gohelp.wordpress.com/>。

Sublime Text 是一个革命性的跨平台（Linux、Mac OS X、Windows）文本编辑器，它支持编写非常多的编程语言代码。对于 Go 而言，它有一个插件叫做 **GoSublime** 来支持代码补全和代码模版。

这里还有一些更加高级的 Go 开发工具，其中一些是以插件的形式利用本身是作为开发 Java 的工具。

NetBeans Go For NetBeans 插件是收费的，它支持语法高亮和代码模版；另外免费的插件正在开发中，你可以通过这个页面获取最新进展：<http://www.turnelvisionlabs.com/downloads/go/>

gogo 可以运行在 Linux 和 Mac 上的一个非常基础的开发工具。

Golde 是一个 IntelliJ IDEA 的插件，具有很好的操作体验和代码补全功能，你可以从这个页面下载：<https://github.com/mtoader/google-go-lang-idea-plugin>

LiteIDE（**golangide**）这是一款专门针对 Go 开发的集成开发环境，在编辑、编译和运行 Go 程序和项目方面都有非常好的支持。同时还包括了对源代码的抽象语法树视图和一些内置工具。（译者注：此开发环境由国人 vfc 大叔开发）

GoClipse 是一款 Eclipse IDE 的插件，拥有非常多的特性以及通过 GoCode 来实现代码补全功能。

如果你对集成开发环境都不是很熟悉，那就使用 LiteIDE 吧，另外使用 GoClipse 或者 Golde 也是不错的选择。

代码补全 一般都是通过内置 GoCode 实现的（如：LietIDE、GoClipse），如果需要手动安装 GoCode，在命令行输入指令 `go get -u github.com/nsf/gocode` 即可。（译者注：务必事先配置好 Go 环境变量）

下表展示了目前三个最好的集成开发环境在功能上的对比，其中 + 表示具备此项功能，++ 表示支持度很好，空白表示不支持此功能。

	Golang LiteIDE	GoClipse	Golde
Syntax highlighting	++	+	+
Automatic saving before building	+		
Line numbering	+	+	
Bookmarks			
Bracket matching		+	
Find / Replace	+	++	
Go to definition			
(Un)Comment		+	
Compiler error double click	++	+	
Cross platform	+	+	
Free	+	+	
Open source	+	+	
Plugin-architecture	+	+	
Easy to use	++	+	
Code snippets			
Project concept	+	+	
Code Folding			
Build system	+	+	
Debugging	+	+	
Recent files and projects	+	+	
Code completion	+	++	
AST-view of code	++	+	
Built-in go tools	+	+	
Browsing of go documentation	+		
Easy switching different Go-environments	++	+	
Exporting code to different formats	++	+	
Project templates			
Integration with version control-systems			
Integration with Google App Engine		++	

接下来会对这三个集成开发环境做更加详细的说明。

3.2.1 Golang LiteIDE

这款 IDE 的当前最新版本号为 X17，你可以从该页面下载到：<http://code.google.com/p/golangide/>

LiteIDE 是一款非常好用的轻量级 Go 集成开发环境（基于 QT、Kate 和 SciTE），包含了跨平台开发及其它必要的特性，对代码编写、自动补全和运行调试都有极佳的支持。它采用了 Go 项目的概念来对项目文件进行浏览和管理，它还支持在各个 Go 开发环境之间随意切换以及交叉编译的功能。

同时，它具备了抽象语法树视图的功能，可以清楚地纵览项目中的常量、变量、函数、不同类型以及他们的属性和方法。

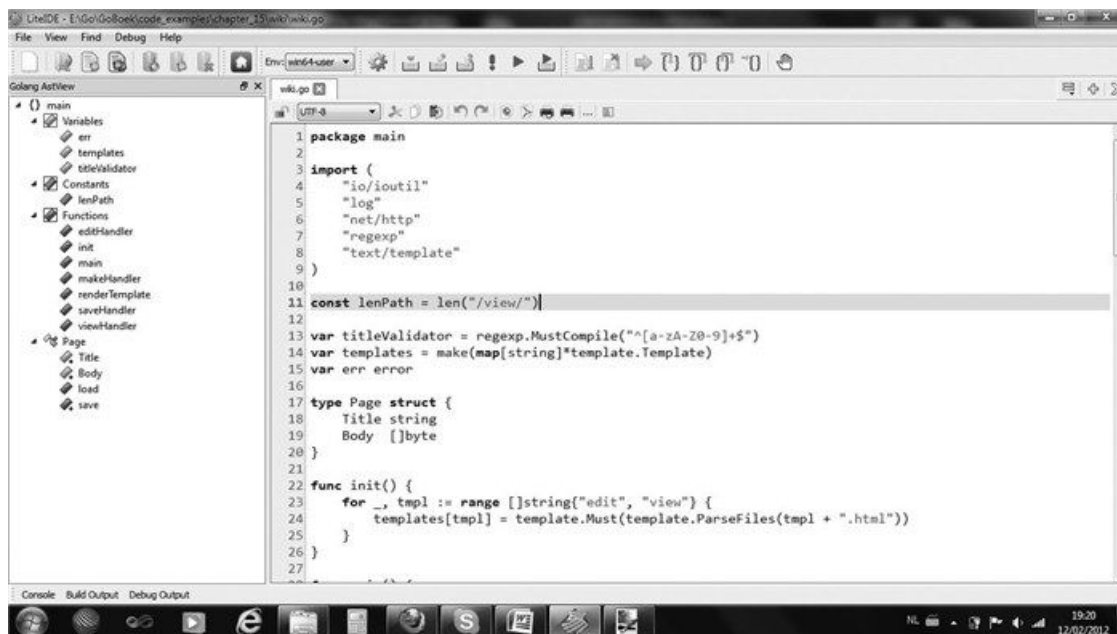


图3.1 LiteIDE 代码编辑界面和抽象语法树视图

3.2.2 GoClipse

该款插件的当前最新版本号为 0.7.6，你可以从该页面下载到：<http://code.google.com/p/goclipse/>

其依附于著名的 Eclipse 这个大型开发环境，虽然需要安装 JVM 运行环境，但却可以很容易地享有 Eclipse 本身所具有的诸多功能。这是一个非常好的编辑器，完善的代码补全、抽象语法树视图、项目管理和程序调试功能。

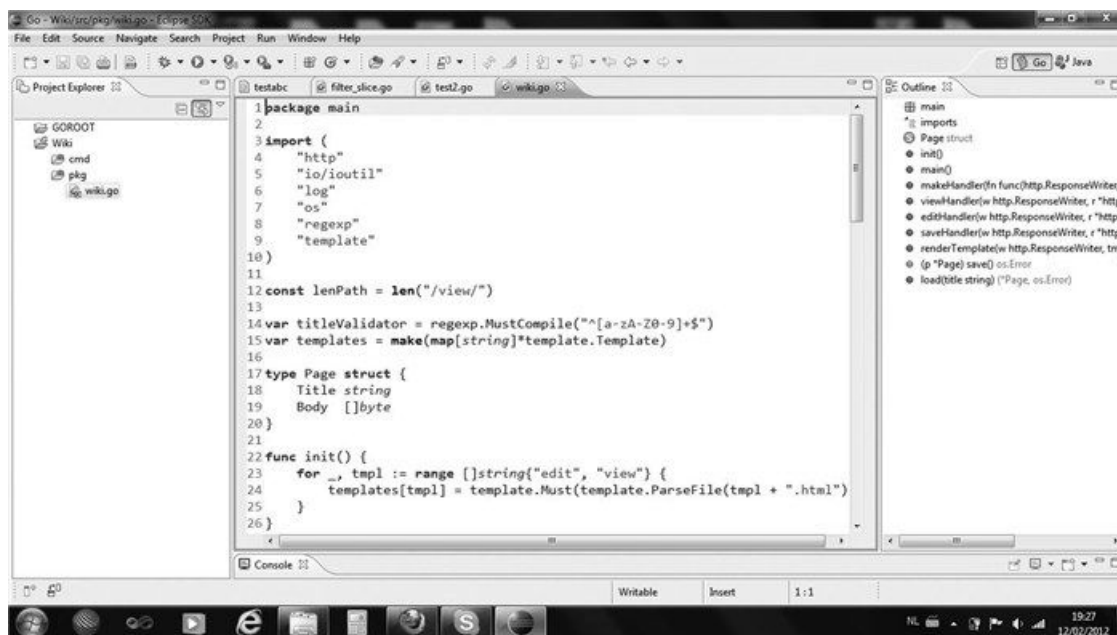


图3.2 GoClipse 代码编辑界面、抽象语法树视图和项目

3.3 调试器

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

应用程序的开发过程中调试是必不可少的一个环节，因此有一个好的调试器是非常重要的，可惜的是，Go 在这方面的的发展还不是很完善。目前可用的调试器是 gdb，最新版均以内置在集成开发环境 LiteIDE 和 GoClipse 中，但是该调试器的调试方式并不灵活且操作难度较大。

如果你不想使用调试器，你可以按照下面的一些有用的方法来达到基本调试的目的：

1. 在合适的位置使用打印语句输出相关变量的值（`print` / `println` 和 `fmt.Print` / `fmt.Println` / `fmt.Printf`）。
2. 在 `fmt.Printf` 中使用下面的说明符来打印有关变量的相关信息：
 - `%+v` 打印包括字段在内的实例的完整信息
 - `%#v` 打印包括字段和限定类型名称在内的实例的完整信息
 - `%T` 打印某个类型的完整说明
3. 使用 `panic` 语句（第 13.2 节）来获取栈跟踪信息（直到 `panic` 时所有被调用函数的列表）。
4. 使用关键字 `defer` 来跟踪代码执行过程（第 6.4 节）。

3.4 构建并运行 Go 程序

（译者注：由于 Go 语言版本更替，本节中的相关内容经原作者同意将被直接替换而不作另外说明）

在大多数 IDE 中，每次构建程序之前都会自动调用源码格式化工具 `gofmt` 并保存格式化后的源文件。如果构建成功则不会输出任何信息，而当发生编译时错误时，则会指明源码中具体第几行出现了什么错误，如：`a declared and not used`。一般情况下，你可以双击 IDE 中的错误信息直接跳转到发生错误的那一行。

如果程序执行一切顺利并成功退出后，将会在控制台输出 `Program exited with code 0`。

从 Go 1 版本开始，使用 Go 自带的更加方便的工具来构建应用程序：

- `go build` 编译并安装自身包和依赖包
- `go install` 安装自身包和依赖包

3.5 格式化代码

Go 开发团队不想要 Go 语言像许多其它语言那样总是在为代码风格而引发无休止的争论，浪费大量宝贵的开发时间，因此他们制作了一个工具：`go fmt`（`gofmt`）。这个工具可以将你的源代码格式化成符合官方统一标准的风格，属于语法风格层面上的小型重构。遵循统一的代码风格是 Go 开发中无可撼动的铁律，因此你必须在编译或提交版本管理系统之前使用 `gofmt` 来格式化你的代码。

尽管这种做法也存在一些争论，但使用 `gofmt` 后你不再需要自成一套代码风格而是和所有人使用相同的规则。这不仅增强了代码的可读性，而且在接手外部 Go 项目时，可以更快地了解其代码的含义。此外，大多数开发工具也都内置了这一功能。

Go 对于代码的缩进层级方面使用 `tab` 还是空格并没有强制规定，一个 `tab` 可以代表 4 个或 8 个空格。在实际开发中，1 个 `tab` 应该代表 4 个空格，而在本身的例子当中，每个 `tab` 代表 8 个空格。至于开发工具方面，一般都是直接使用 `tab` 而不替换成空格。

在命令行输入 `gofmt -w program.go` 会格式化该源文件的代码然后将格式化后的代码覆盖原始内容（如果不加参数 `-w` 则只会打印格式化后的结果而不重写文件）；`gofmt -w *.go` 会格式化并重写所有 Go 源文件；`gofmt map1` 会格式化并重写 `map1` 目录及其子目录下的所有 Go 源文件。

`gofmt` 也可以通过在参数 `-r` 后面加入用双引号括起来的替换规则实现代码的简单重构，规则的格式：`<原始内容> -> <替换内容>`。

实例：

```
gofmt -r "(a) -> a" -w *.go
```

上面的代码会将源文件中没有意义的括号去掉。

```
gofmt -r "a[n:len(a)] -> a[n:]" -w *.go
```

上面的代码会将源文件中多余的 `len(a)` 去掉。（译者注：了解 `slice` 之后就明白这为什么是多余的了）

```
gofmt -r 'A.Func1(a,b) -> A.Func2(b,a)' -w *.go
```

上面的代码会将源文件中符合条件的函数的参数调换位置。

如果想要了解有关 `gofmt` 的更多信息，请访问该页面：<http://golang.org/cmd/gofmt/>

3.6 生成代码文档

`go doc` 工具会从 Go 程序和包文件中提取顶级声明的首行注释以及每个对象的相关注释，并生成相关文档。

它也可以作为一个提供在线文档浏览的 web 服务器，<http://golang.org> 就是通过这种形式实现的。

一般用法

- `go doc package` 获取包的文档注释，例如：`go doc fmt` 会显示使用 `godoc` 生成的 `fmt` 包的文档注释。
- `go doc package/subpackage` 获取子包的文档注释，例如：`go doc container/list`。
- `go doc package function` 获取某个函数在某个包中的文档注释，例如：`go doc fmt Printf` 会显示有关 `fmt.Printf()` 的使用说明。

这个工具只能获取在 Go 安装目录下 `.../go/src/pkg` 中的注释内容。此外，它还可以作为一个本地文档浏览 web 服务器。在命令行输入 `godoc -http=:6060`，然后使用浏览器打开 <http://localhost:6060> 后，你就可以看到本地文档浏览服务器提供的页面。

`godoc` 也可以用于生成非标准库的 Go 源码文件的文档注释（第 9.6 章）。

如果想要获取更多有关 `godoc` 的信息，请访问该页面：<http://golang.org/cmd/godoc/>

3.7 其它工具

Go 自带的工具集主要使用脚本和 Go 语言自身编写的，目前版本的 Go 实现了以下三个工具：

- `go install` 是安装 Go 包的工具，类似 Ruby 中的 `rubygems`。主要用于安装非标准库的包文件，将源代码编译成对象文件。
- `go fix` 用于将你的 Go 代码从旧的发行版迁移到最新的发行版，它主要负责简单的、重复的、枯燥无味的修改工作，如像 API 等复杂的函数修改，工具则会给出文件名和代码行数的提示以便让开发人员快速定位并升级代码。Go 开发团队一般也使用这个工具升级 Go 内置工具以及谷歌内部项目的代码。`go fix` 之所以能够正常功能是因为 Go 在标准库就提供生成抽象语法树和通过抽象语法树对代码进行还原的功能。该工具会尝试更新当前目录下的所有 Go 源文件，并在完成代码更新后在控制台输出相关的文件名称。
- `go test` 是一个轻量级的单元测试框架（第 13 章）。

3.8 Go 性能说明

根据 Go 开发团队和基本的算法测试，Go 的性能大概在 C 语言的 10%~20% 之间（译者注：由于出版时间限制，该数据应为 2013 年 3 月 28 日之前产生）。虽然没有官方的性能标准，但是与其它各个语言相比已经拥有非常出色的表现。

如果说 Go 语言的执行效率大约比 C++ 慢 20% 也许更有实际意义。保守估计在相同的环境和执行目标的情况下，Go 程序比 Java 或 Scala 应用程序要快上 2 倍，并比这两门语言使用少占用 70% 的内存。在很多情况下这种比较是没有意义的，因为像谷歌这样拥有成千上万台服务器的公司都抛弃 C++ 而开始将 Go 用于生产环境已经足够说明它本身所具有的优势。

时下流行的语言大都是运行在虚拟机上，如：Java 和 Scala 使用的 JVM，C# 和 VB.NET 使用的 .NET CLR。尽管虚拟机的性能已经有了很大的提升，但任何使用 JIT 编译器和脚本语言解释器的编程语言（Ruby、Python、Perl 和 JavaScript）在 C 和 C++ 的绝对优势下甚至都无法在性能上望其项背。

如果说 Go 比 C++ 要慢 20%，那么 Go 就要比任何非静态和编译型语言快 2 到 10 倍，并且能够更加高效地使用内存。

其实比较多门语言之间的性能是一种非常猥琐的行为，因为任何一种语言都有其所长和薄弱的方面。例如在处理文本方面，那些只处理纯字节的语言显然要比处理 Unicode 这种更为复杂编码的语言要出色的多。有些人可能认为使用两种不同的语言实现同一个目标能够得出正确的结论，但是很多时候测试者可能对一门语言非常了解而对另一门语言只是大概明白，测试者对程序编写的手法在一定程度也会影响结果的公平性，因此测试程序应该分别由各自语言的擅长者来编写，这样才能得到具有可比性的结果。另外，像在统计学方面，人们很难避免人为因素对结果的影响，所以这在严格意义上并不是科学。还要注意的，测试结果的可比性还要根据测试目标来区别，例如很多发展十多年的语言已经针对各类问题拥有非常成熟的类库，而作为一门新生语言的 Go 语言，并没有足够的时间来推导各类问题的最佳解决方案。如果你想获取更多有关性能的资料，请访问 [Computer Language Benchmark Game](#)（详见引用 27）。

这里有一些评测结果：

- 比较 Go 和 Python 在简单的 web 服务器方面的性能，单位为传输量每秒：

原生的 Go http 包要比 `web.py` 快 7 至 8 倍，如果使用 `web.go` 框架则稍微差点，比 `web.py` 快 6 至 7 倍。在 Python 中被广泛

使用的 tornado 异步服务器和框架在 web 环境下要比 web.py 快很多，Go 大概只比它快 1.2 至 1.5 倍（详见引用 26）。

- Go 和 Python 在一般开发的平均水平测试中，Go 要比 Python 3 快 25 倍左右，少占用三分之二的内存，但比 Python 大概多写一倍的代码（详见引用 27）。
- 根据 Robert Hundt（2011 年 6 月，详见引用 28）的文章对 C++、Java、Go 和 Scala，以及 Go 开发团队的反应（详见引用 29），可以得出以下结论：
 - Go 和 Scala 之间具有更多的可比性（都使用更少的代码），而 C++ 和 Java 都使用非常冗长的代码。
 - Go 的编译速度要比绝大多数语言都要快，比 Java 和 C++ 快 5 至 6 倍，比 Scala 快 10 倍。
 - Go 的二进制文件体积是最大的（每个可执行文件都包含 runtime）。
 - 在最理想的情况下，Go 能够和 C++ 一样快，比 Scala 快 2 至 3 倍，比 Java 快 5 至 10 倍。
 - Go 在内存管理方面也可以和 C++ 相媲美，几乎只需要 Scala 所使用的一半，比 Java 少 4 倍左右。

3.9 与其它语言进行交互

3.9.1 与 C 进行交互

工具 cgo 提供了对 FFI（外部函数接口）的支持，能够使用 Go 代码安全地调用 C 语言库，你可以访问 cgo 文档主页：<http://golang.org/cmd/cgo>。cgo 会替代 Go 编译器来产生可以组合在同一个包中的 Go 和 C 代码。在实际开发中一般使用 cgo 创建单独的 C 代码包。

如果你想要在你的 Go 程序中使用 cgo，则必须在单独的一行使用 `import "C"` 来导入，一般来说你可能还需要 `import "unsafe"`。

然后，你可以在 `import "C"` 之前使用注释（单行或多行注释均可）的形式导入 C 语言库（甚至有效的 C 语言代码），它们之间没有空行，例如：

```
// #include <stdio.h>
// #include <stdlib.h>
import "C"
```

名称 "C" 并不属于标准库的一部分，这只是 cgo 集成的一个特殊名称用于引用 C 的命名空间。在这个命名空间里所包含的 C 类型都可以被使用，例如 `C.uint`、`C.long` 等等，还有 `libc` 中的函数 `C.random()` 等也可以被调用。

当你想要使用某个类型作为 C 中函数的参数时，必须将其转换为 C 中的类型，反之亦然，例如：

```
var i int
C.uint(i) // 从 Go 中的 int 转换为 C 中的无符号 int
int(C.random()) // 从 C 中 random() 函数返回的 long 转换为 Go 中的 int
```

下面的 2 个 Go 函数 `Random()` 和 `Seed()` 分别调用了 C 中的 `C.random()` 和 `C.srandom()`。

Example 3.2 [cl.go](#)

```
package rand
// #include <stdlib.h>
import "C"
func Random() int {
    return int(C.random())
}
func Seed(i int) {
    C.srandom(C.uint(i))
}
```

C 当中并没有明确的字符串类型，如果你想要将一个 `string` 类型的变量从 Go 转换到 C，可以使用 `C.CString(s)`；同样，可以使用 `C.GoString(cs)` 从 C 转换到 Go 中的 `string` 类型。

Go 的内存管理机制无法管理通过 C 代码分配的内存。

开发人员需要通过手动调用 `C.free` 来释放变量的内存：

```
defer C.free(unsafe.Pointer(Cvariable))
```

这一行最好紧跟在使用 C 代码创建某个变量之后，这样就不会忘记释放内存了。下面的代码展示了如何使用 cgo 创建变量、使用并释放其内存：

Example 3.3 [c2.go](#)

```
package print
// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"
func Print(s string) {
    cs := C.CString(s)
    defer C.free(unsafe.Pointer(cs))
    C.fputs(cs, (*C.FILE)(C.stdout))
}
```

构建 cgo 包

你可以在使用将会在第 9.5 节讲到的 Makefile 文件（因为我们使用了一个独立的包），除了使用变量 GOFILES 之外，还需要使用变量 CGOFILES 来列出需要使用 cgo 编译的文件列表。例如，Example 3.2 中的代码就可以使用包含以下内容的 Makefile 文件来编译，你可以使用 gomake 或 make：

```
include $(GOROOT)/src/Make.inc
TARG=rand
CGOFILES=\
c1.go\
include $(GOROOT)/src/Make.pkg
```

3.9.2 与 C++ 进行交互

SWIG（简化封装器和接口生成器）支持在 Linux 系统下使用 Go 代码调用 C 或者 C++ 代码。这里有一些使用 SWIG 的注意事项：

- 编写需要封装的库的 SWIG 接口。
- SWIG 会产生 C 的存根函数。
- 这些库可以使用 cgo 来调用。
- 相关的 Go 文件也可以被自动生成。

这类接口支持方法重载、多重继承以及使用 Go 代码实现 C++ 的抽象类。

目前使用 SWIG 存在的一个问题是它无法支持所有的 C++ 库，比如说它就无法解析 TObject.h。

4 基本结构和基本数据类型

本章主要介绍 Go 语言的基本结构和基本数据类型。

4.1 文件名、关键字与标识符

Go 的源文件以 .go 为后缀名存储在计算机中，这些文件名均由小写字母组成，如 scanner.go。如果文件名由多个部分组成，则使用下划线 _ 对它们进行分隔，如 scanner_test.go。文件名不包含空格或其他特殊字符。

一个源文件可以包含任意多行的代码，Go 本身没有对源文件的大小进行限制。

你会发现在 Go 代码中的几乎所有东西都有一个名称或标识符。另外，Go 语言也是区分大小写的，这与 C 家族中的其它语言相同。有效的标识符必须以字符（可以使用任何 UTF-8 编码的字符或 _）开头，然后紧跟着 0 个或多个字符或 Unicode 数字，如：X56, group1, _x23, i, øε12。

以下是无效的标识符：

1ab（以数字开头），case（Go 语言的关键字），a+b（运算符是不允许的）

`_` 本身就是一个特殊的标识符，被称为空白标识符。它可以像其他标识符那样用于变量的声明或赋值（任何类型都可以赋值给它），但任何赋给这个标识符的值都将被抛弃，因此这些值不能在后续的代码中使用，也不可以使用这个标识符作为变量对其它变量的进行赋值或运算。

在编码过程中，你可能会遇到没有名称的变量、类型或方法，尽管这不是必须的，但有时候这样做可以极大地增强代码的灵活性，这些变量被统称为匿名变量。

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

之所以刻意地将 Go 代码中的关键字保持的这么少，是为了简化在编译过程第一步中的代码解析。和其它语言一样，关键字不能够作标识符使用。

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符，其中包含了基本类型的名称和一些基本的内置函数（第 6.5 节），它们的作用都将在接下来的章节中进行进一步地讲解。

append	bool	byte	cap	close	complex	complex64	complex128	uint16
copy	false	float32	float64	imag	int	int8	int16	uint32
int32	int64	iota	len	make	new	nil	panic	uint64
print	println	real	recover	string	true	uint	uint8	uintptr

程序一般由关键字，常量，变量，运算符，类型和函数组成。

程序中可能会使用到这些分隔符：括号（），中括号[] 和大括号{}。

程序中可能会使用到这些标点符号：.，；和 ...。

程序的代码通过语句来实现结构化。每个语句不需要像 C 家族中的其它语言一样以分号；结尾，因为这些工作都将由 Go 编译器自动完成。

如果你打算将多个语句写在同一行，它们则必须使用；人为区分，但在实际开发中我们并不鼓励这种做法。

4.2 Go 程序的基本结构和要素

Example 4.1 [hello_world.go](#)

```
package main

import "fmt"

func main() {
    fmt.Println("hello, world")
}
```

4.2.1 包的概念、导入与可见性

包是结构化代码的一种方式：每个程序都由包（通常简称为 pkg）的概念组成，可以使用自身的包或者从其它包中导入内容。

如同其它一些编程语言中的类库或命名空间的概念，每个 Go 文件都属于且仅属于一个包。一个包可以由许多以 .go 为扩展名的源文件组成，因此文件名和包名一般来说都是不相同的。

你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main 表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。

一个应用程序可以包含不同的包，而且即使你只使用 main 包也不必把所有的代码都写在一个巨大的文件里：你可以用一些较小的文件，并且在每个文件非注释的第一行都使用 package main 来指明这些文件都属于 main 包。如果你打算编译包名不是为 main 的源文件，如 pack1，编译后产生的对象文件将会是 pack1.a 而不是可执行程序。另外要注意的是，所有的包名都应该使用小写字母。

标准库

在 Go 的安装文件里包含了一些可以直接使用的包，即标准库。在 Windows 下，标准库的位置在 Go 根目录下的子目录 `pkg\windows_386` 中；在 Linux 下，标准库在 Go 根目录下的子目录 `pkg\linux_amd64` 中（如果是安装的是 32 位，则在 `linux_386` 目录中）。一般情况下，标准包会存放在 `$GOROOT/pkg/$GOOS_$GOARCH/` 目录下。

Go 的标准库包含了大量的包（如：`fmt`，`os`），但是你也可以创建自己的包（第 8 章）。

如果想要构建一个程序，则包和包内的文件都必须以正确的顺序进行编译。包的依赖关系决定了其构建顺序。

属于同一个包的源文件必须全部被一起编译，一个包既是编译时的一个单元，因此根据惯例，每个目录都只包含一个包。

如果对一个包进行更改或重新编译，所有引用了这个包的客户端程序都必须全部重新编译。

Go 中的包模型采用了显式依赖关系的机制来达到快速编译的目的，编译器会从后缀名为 `.o` 的对象文件（需要且只需要这个文件）中提取传递依赖类型的信息。

如果 `A.go` 依赖 `B.go`，而 `B.go` 又依赖 `C.go`：

- 编译 `C.go`，`B.go`，然后是 `A.go`。
- 为了编译 `A.go`，编译器读取的是 `B.o` 而不是 `C.o`。

这种机制对于编译大型的项目时可以显著地提升编译速度。

每一段代码只会被编译一次

一个 Go 程序是通过 `import` 关键字将一组包链接在一起。

`import "fmt"` 告诉 Go 编译器这个程序需要使用 `fmt` 包（的函数，或其他元素），`fmt` 包实现了格式化 IO（输入/输出）的函数。包名被封闭在半角双引号 `"` 中。如果你打算从已编译的包中导入并加载公开声明的方法，不需要插入已编译包的源代码。

如果需要多个包，它们可以被分别导入：

```
import "fmt"
import "os"
```

或：

```
import "fmt"; import "os"
```

但是还有更短且更优雅的方法（被称为因式分解关键字，该方法同样适用于 `const`、`var` 和 `type` 的声明或定义）：

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

（它甚至还可以更短：`import ("fmt", "os")` 但使用 `gofmt` 后将会被强制换行）

当你导入多个包时，导入的顺序会按照字母排序。

如果包名不是以 `.` 或 `/` 开头，如 `"fmt"` 或者 `"container/list"`，则 Go 会在全局文件进行查找；如果包名以 `./` 开头，则 Go 会在相对目录中查找；如果包名以 `/` 开头（在 Windows 下也可以这样使用），则会在系统的绝对路径中查找。

导入包即等同于包含了这个包的所有的代码对象。

除了符号 `_`，包中所有代码对象的标识符必须是唯一的，以避免名称冲突。但是相同的标识符可以在不同的包中使用，因为可以使用包名来区分它们。

包通过下面这个被编译器强制执行的规则来决定是否将自身的代码对象暴露给外部文件：

可见性规则

当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：`Group1`，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 `public`）；标识

符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 `private`）。

（大写字母可以使用任何 Unicode 编码的字符，比如希腊文，不仅仅是 ASCII 码中的大写字母）。

因此，在导入一个外部包后，能够且只能访问该包中导出的对象。

假设在包 `pack1` 中我们有一个变量或函数叫做 `Thing`（以 `T` 开头，所以它能够被导出），那么在当前包中导入 `pack1` 包，`Thing` 就可以像面向对象语言那样使用点标记来调用：`pack1.Thing`（`pack1` 在这里是不可以省略的）

因此包也可以作为命名空间使用，帮助避免命名冲突（名称冲突）：两个包中的同名变量的区别在于他们的包名，例如 `pack1.Thing` 和 `pack2.Thing`。

你可以通过使用包的别名来解决包名之间的名称冲突，或者说根据你的个人喜好对包名进行重新设置，如：`import fm "fmt"`。下面的代码展示了如何使用包的别名：

Example 4.2 [alias.go](#)

```
package main
import fm "fmt" // alias3

func main() {
    fm.Println("hello, world")
}
```

注意事项 如果你导入了一个包却没有使用它，则会在构建程序时引发错误，如 `imported and not used: os`，这正是遵循了 Go 的格言：“没有不必要的代码！”

包的分级声明和初始化

你可以在使用 `import` 导入包之后定义或声明 0 个或多个常量（`const`）、变量（`var`）和类型（`type`），这些对象的作用域都是全局的（在本包范围内），所以可以被本包中所有的函数调用（如 `gotemplate.go` 源文件中的 `c` 和 `v`），然后声明一个或多个函数（`func`）。

4.2.2 函数

这是定义一个函数最简单的格式：`func functionName()`

你可以在括号（`()`）中写入 0 个或多个函数的参数（使用逗号，分隔），每个参数的名称后面必须紧跟着该参数的类型。

`main` 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（译者注：如果有 `init()` 函数则会先执行该函数）。如果你的 `main` 包的源代码没有包含 `main` 函数，则会引发构建错误 `undefined: main.main`。`main` 函数即没有参数，也没有返回类型（与 C 家族中的其它语言恰好相反）。如果你不小心为 `main` 函数添加了参数或者返回类型，将会引发构建错误：

```
func main must have no arguments and no return values results.
```

在程序开始执行并完成初始化后，第一个调用（程序的入口点）的函数是 `main.main()`（如：C 语言），该函数一旦返回就表示程序已成功执行并立即退出。

函数里的代码（函数体）使用大括号 { } 括起来。

左大括号 { 必须与方法的声明放在同一行，这是编译器的强制规定，否则你在使用 `gofmt` 时就会出现 `build-error: syntax error: unexpected semicolon or newline before {` 这样的错误提示。

（这是因为编译器会产生 `func main()`；这样的结果，很明显这错误的）（Go 语言虽然看起来不使用分号作为语句的结束，但实际上这一过程是由编译器自动完成，因此才会引发向上面这样的错误）

右大括号 } 需要被放在紧接着函数体的下一行。如果你的函数非常简短，你也可以将它们放在同一行：`func Sum(a, b int) int { return a + b }`。

对于大括号 { } 的使用规则在任何时候都是相同的（如：if 语句等）。

因此符合规范的函数一般写成如下的形式：

```
func functionName(parameter_list) (return_value_list) {
    ...
}
```

```
}
```

其中：

```
parameter_list 的形式为 (param1 type1, param2 type2, ...)  
return_value_list 的形式为 (ret1 type1, ret2 type2, ...)
```

只有当某个函数需要被外部包调用的时候才使用大写字母开头，并遵循 Pascal 命名法；否则就遵循骆驼命名法，即第一个单词的首字母小写，其余单词的首字母大写。

`fmt.Println("hello, world")` 这一行调用了 `fmt` 包中的 `Println` 函数，可以将字符串输出到控制台，并在最后自动增加换行字符 `\n`。

使用 `fmt.Print("hello, world\n")` 可以得到相同的结果。

`Print` 和 `Println` 这两个函数也支持使用变量，如：`fmt.Println(arr)`。如果没有特别指定，它们会以默认的打印格式将变量 `arr` 输出到控制台。

单纯地打印一个字符串或变量甚至可以使用预定义的方法来实现，如：`print`、`println`：`print("ABC")`、`println("ABC")`、`println(i)`（带一个变量 `i`）。

这些函数只可以用于调试阶段，在部署程序的时候务必将它们替换成 `fmt` 中的相关函数。

当被调用函数的代码执行到结束符 `}` 或返回语句时就会返回，然后程序继续执行调用该函数之后的代码。

程序正常退出的代码为 `0` `Program exited with code 0`；如果程序因为异常而被终止，则会返回非零值，如：`1`。这个数值可以用来测试是否成功执行一个程序。

4.2.3 注释

Example 4.2 [hello_world2.go](#)

```
package main  
import "fmt" // Package implementing formatted I/O.  
func main() {  
    fmt.Printf("Καλημέρα κόσμε; or こんにちは 世界\n")  
}
```

上面这个例子通过打印 `Καλημέρα κόσμε; or こんにちは 世界` 展示了如何在 Go 中使用国际化字符，以及如何使用注释。

注释不会被编译，但可以通过 `godoc` 来使用（第 3.6 节）

单行注释是最常见的注释形式，你可以在任何地方使用以 `//` 开头的单行注释。多行注释也叫块注释，均已以 `/*` 开头，并以 `*/` 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。

每一个包应该有相关注释，在 `package` 语句之前的块注释将被默认认为是这个包的文档说明，其中应该提供一些相关信息并对整体功能做简要的介绍。一个包可以分散在多个文件中，但是只需要在其中一个进行注释说明即可。当开发人员需要了解包的一些情况时，自然会用 `godoc` 来显示包的文档说明，在首行的简要注释之后可以用成段的注释来进行更详细的说明，而不必拥挤在一起。另外，在多段注释之间应以空行分隔加以区分。

示例：

```
// Package superman implements methods for saving the world.  
//  
// Experience has shown that a small number of procedures can prove  
// helpful when attempting to save the world.  
package superman
```

几乎所有全局作用域的类型、常量、变量、函数和被导出的对象都应该有一个合理的注释。如果这种注释（称为文档注释）出现在函数前面，例如函数 `Abcd`，则要以 `"Abcd..."` 作为开头。

示例：

```
// enterOrbit causes Superman to fly into low Earth orbit, a position
```

```
// that presents several possibilities for planet salvation.
func enterOrbit() error {
    ...
}
```

godoc 工具（第 3.6 节）会收集这些注释并产生一个技术文档。

4.2.4 类型

可以包含数据的变量（或常量）可以使用不同的数据类型或类型来保存数据。使用 `var` 声明的变量的值会自动初始化为该类型的零值。类型定义了某个变量的值的集合与可对其进行操作的集合。

类型可以是基本类型，如：`int`, `float`, `bool`, `string`；结构化的（复合的），如：`struct`, `array`, `slice`, `map`, `channel`；只描述类型的行为的，如：`interface`。

结构化的类型没有真正的值，它使用 `nil` 作为默认值（在 Objective-C 中是 `nil`，在 Java 中是 `null`，在 C 和 C++ 中是 `NULL` 或 `0`）。值得注意的是，Go 语言中不存在类型继承。

函数也可以是一个确定的类型，就是以函数作为返回类型。这种类型的声明要写在函数名和可选的参数列表之后，例如：

```
func FunctionName (a typea, b typeb) typeFunc
```

你可以在函数体中的某处返回使用类型为 `typeFunc` 的变量 `var`：

```
return var
```

一个函数可以拥有多返回值，返回类型之间需要使用逗号分割，并使用小括号（`）` 将它们括起来，如：`func FunctionName (a typea, b typeb) (t1 type1, t2 type2)`

示例：函数 `Atoi`(第 4.7 节)：`func Atoi(s string) (i int, err error)`

返回的形式：`return var1, var2`

这种多返回值一般用于判断某个函数是否执行成功（`true/false`）或与其它返回值一同返回错误消息（详见之后的并行赋值）。

使用 `type` 关键字可以定义你自己的类型，你可能想要定义一个结构体(第 10 章)，但是也可以定义一个已经存在的类型的别名，如：

```
type IZ int
```

（译者注：这里并不是真正意义上的别名，因为使用这种方法定义之后的类型可以拥有更多的特性，且在类型转换时必须显式转换）

然后我们可以像 `var a IZ = 5` 这样声明变量。

这里我们可以看到 `int` 是变量 `a` 的底层类型，这也使得它们之间存在相互转换的可能（第 4.2.6 节）。

如果你有多个类型需要定义，可以使用因式分解关键字的方式，例如：

```
type (
    IZ int
    FZ float
    STR string
)
```

每个值都必须在经过编译后属于某个类型（编译器必须能够推断出所有值的类型），因为 Go 语言是一种静态类型语言。

4.2.5 Go 程序的一般结构

下面的程序可以被顺利编译但什么都做不了，不过这很好地展示了一个 Go 程序的首选结构。这种结构并没有被强制要求，编译器也不关心 `main` 函数在前还是变量的声明在前，但使用统一的结构能够在从上至下阅读 Go 代码时有更好的体验。

所有的结构将在这一章或接下来的章节中进一步地解释说明，但总体思路如下：

- 在完成包的 `import` 之后，开始对常量、变量和类型的定义或声明。
- 如果存在 `init` 函数的话，则对该函数进行定义（这是一个特殊的函数，每个含有该函数的包都会首先执行这个函数）。
- 如果当前包是 `main` 包，则定义 `main` 函数。
- 然后定义其余的函数，首先是类型的方法，接着是按照 `main` 函数中先后调用的顺序来定义相关函数，如果有很多函数，则可以按照字母顺序来进行排序。

Example 4.4 `gotemplate.go`

```
package main
import (
    "fmt"
)
const c = "C"
var v int = 5
type T struct{}
func init() { // initialization of package
}
func main() {
    var a int
    Func1()
    // ...
    fmt.Println(a)
}
func (t T) Method1() {
    //...
}
func Func1() { // exported function Func1
    //...
}
```

Go 程序的执行（程序启动）顺序如下：

1. 按顺序导入所有被 `main` 包引用的其它包，然后在每个包中执行如下流程：
2. 如果该包又导入了其它的包，则从第一步开始递归执行，但是每个包只会被导入一次。
3. 然后以相反的顺序在每个包中初始化常量和变量，如果该包含有 `init` 函数的话，则调用该函数。
4. 在完成这一切之后，`main` 也执行同样的过程，最后调用 `main` 函数开始执行程序。

4.2.6 类型转换

在必要以及可行的情况下，一个类型的值可以被转换成另一种类型的值。由于 Go 语言不存在隐式类型转换，因此所有的转换都必须显式说明，就像调用一个函数一样（类型在这里的作用可以看作是一种函数）：

```
valueOfTypeB = typeB(valueOfTypeA)
```

（译者注：类型B的值 = 类型B(类型A的值)）

示例：

```
a := 5.0
b := int(a)
```

但这只能在定义正确的情况下转换成功，例如从一个取值范围较小的类型转换到一个取值范围较大的类型（例如将 `int16` 转换为 `int32`）。当从一个取值范围较大的转换到取值范围较小的类型时（例如将 `int32` 转换为 `int16` 或将 `float32` 转换为 `int`），会发生精度丢失（截断）的情况。当编译器捕捉到非法的类型转换时会引发编译时错误，否则将引发运行时错误。

具有相同底层类型的变量之间可以相互转换：

```
var a IZ = 5
c := int(a)
d := IZ(c)
```

4.2.7 Go 命名规范

干净、可读的代码和简洁性是 Go 追求的主要目标。通过 `gofmt` 来强制实现统一的代码风格。Go 语言中对象的命名也应该是简洁且有意义的。像 Java 和 Python 中那样使用混合着大小写和下划线的冗长的名称会严重降低代码的可读性。名称不需要指出自己所属的包，因为在调用的时候会使用包名作为限定符。返回某个对象的函数或方法的名称一般都是使用名词，没有“Get...”之类的字符（译者注：个人觉得这个有点牵强），如果是用于修改某个对象，则使用“SetName”。有必须要的话可以使用大小写混合的方式，如 `MixedCaps` 或 `mixedCaps`，而不是使用下划线来分割多个名称。

4.3 常量

常量使用关键字 `const` 定义，用于存储不会改变的数据。

存储在常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：`const identifier [type] = value`，例如 `const Pi = 3.14159`。

在 Go 语言中，你可以省略类型说明符 `[type]`，因为编译器可以根据变量的值来推断其类型。

- 显式类型定义：`const b string = "abc"`
- 隐式类型定义：`const b = "abc"`

一个没有指定类型的常量被使用时，会根据其使用环境而推断出它所需要具备的类型。换句话说，未定义类型的常量会在必要时刻根据上下文来获得相关类型。

```
var n int
f(n + 5) // 无类型的数字型常量“5”它的类型在这里变成了 int
```

常量的值必须是能够在编译时就能够确定的；你可以在其赋值表达式中涉及计算过程，但是所有用于计算的值必须在编译期间就能获得。

- 正确的做法：`const c1 = 2/3`
- 错误的做法：`const c2 = getNumber()` // 引发构建错误: `getNumber()` used as value

（译者注：因为在编译期间自定义函数均属于未知，因此无法用于常量的赋值，但内置函数可以使用，如：`len()`）

数字型的常量是没有大小和符号的，并且可以使用任何精度而不会导致溢出：

```
const Ln2= 0.693147180559945309417232121458\
          176568075500134360255254120680009
const Log2E= 1/Ln2 // this is a precise reciprocal
const Billion = 1e9 // float constant
const hardEight = (1 << 100) >> 97
```

根据上面的例子我们可以看到，反斜杠 `\` 可以在常量表达式中作为多行的连接符使用。

与各种类型的数字型变量相比，你无需担心常量之间的类型转换问题，因为它们都是非常理想的数字。

不过需要注意的是，当常量赋值给一个精度过小的数字型变量时，可能会因为无法正确表达常量所代表的数值而导致溢出，这会在编译期间就引发错误。另外，常量也允许使用并行赋值的形式：

```
const beef, two, c = "meat", 2, "veg"
const Monday, Tuesday, Wednesday, Thursday, Friday, Saturday = 1, 2, 3, 4, 5, 6
const (
    Monday, Tuesday, Wednesday = 1, 2, 3
    Thursday, Friday, Saturday = 4, 5, 6
)
```

常量还可以用作枚举：

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

现在，数字 0、1 和 2 分别代表未知性别、女性和男性。这些枚举值可以用于测试某个变量或常量的实际值，比如使用

switch/case 结构 (第 5.3 节).

在这个例子中, `iota` 可以被用作枚举值:

```
const (  
    a = iota  
    b = iota  
    c = iota  
)
```

第一个 `iota` 等于 0, 每当 `iota` 在新的一行被使用时, 它的值都会自动加 1; 所以 `a=0`, `b=1`, `c=2` 可以简写为如下形式:

```
const (  
    a = iota  
    b  
    c  
)
```

(译者注: 关于 `iota` 的使用涉及到非常复杂多样的情况, 这里作者解释的并不清晰, 因为很难对 `iota` 的用法进行直观的文字描述。如希望进一步了解, 请观看视频教程 [《Go编程基础》第四课: 常量与运算符](#))

`iota` 也可以用在表达式中, 如: `iota + 50`。在每遇到一个新的常量块或单个常量声明时, `iota` 都会重置为 0 (译者注: 简单地讲, 每遇到一次 `const` 关键字, `iota` 就重置为 0)。

当然, 常量之所以为常量就是恒定不变的量, 因此我们无法在程序运行过程中修改它的值; 如果你在代码中试图修改常量的值则会引发编译错误。

引用 `time` 包中的一段代码作为示例: 一周中每天的名称。

```
const (  
    Sunday = iota  
    Monday  
    Tuesday  
    Wednesday  
    Thursday  
    Friday  
    Saturday  
)
```

你也可以使用某个类型作为枚举常量的类型:

```
type Color int  
const (  
    RED Color = iota // 0  
    ORANGE // 1  
    YELLOW // 2  
    GREEN // ..  
    BLUE  
    INDIGO  
    VIOLET // 6  
)
```

注意事项 作为约定, 常量的标识符主要使用大写字母, 标识符中各个部分的连接字符可用小写字母以便区分, 如: `const INCHTOCM = 2.54`; 这样不仅增强了可读性, 而且不会与第 4.2 节中描述的可见性规则冲突。

4.4 变量

4.4.1 简介

声明变量的一般形式是使用 `var` 关键字: `var identifier type`。

需要注意的是, Go 和许多编程语言不同, 它在声明变量时将变量的类型放在变量的名称之后。Go 为什么要选择这么做呢?

首先, 它是为了避免像 C 语言中那样含糊不清的声明形式, 例如: `int* a, b;`。在这个例子中, 只有 `a` 是指针而 `b` 不是。如果你想要这两个变量都是指针, 则需要将它们分开书写 (你可以在该页面找到有关于这个话题的更多讨论)

论：<http://blog.golang.org/2010/07/gos-declaration-syntax.html>）。

而在 Go 中，则可以和轻松地将它们都声明为指针类型：`var a, b *int`。

其次，这种语法能够按照从左至右的顺序阅读，使得代码更加容易理解。

示例：

```
var a int
var b bool
var str string
```

你也可以改写成这种形式：

```
var (
    a int
    b bool
    str string
)
```

这种因式分解关键字的写法一般用于声明全局变量。

当一个变量被声明之后，系统自动赋予它该类型的零值：`int` 为 0，`float` 为 0.0，`bool` 为 `false`，`string` 为空字符串，指针为 `nil`。记住，所有的内存在 Go 中都是经过初始化的。

变量的命名规则遵循骆驼命名法，即首个单词小写，每个新单词的首字母大写，例如：`numShips`，`startDate`。

但如果你的全局变量希望能够被外部包所使用，则需要将首个单词的首字母也大写（第 4.2 节：可见性规则）。

一个变量（常量、类型或函数）在程序中都有一定的作用范围，称之为作用域。如果一个变量在函数体外声明，则被认为是全局变量，可以在整个包甚至外部包（被导出后）使用，不管你声明在哪个源文件里或在哪个源文件里调用该变量。

在函数体内声明的变量称之为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。在第 5 章，我们将会学习到像 `if` 和 `for` 这些控制结构，而在这些结构中声明的变量的作用域只在相应的代码块内。一般情况下，局部变量的作用域可以通过代码块（用大括号括起来的部分）判断。

尽管变量的标识符必须是唯一的，但你可以在某个代码块的内层代码块中使用相同名称的变量，则此时外部的同名变量将会暂时隐藏（结束内部代码块的执行后隐藏的外部同名变量又会出现，而内部同名变量则被释放），你任何的操作都只会影响内部代码块的局部变量。

变量可以编译期间就被赋值，赋值给变量使用运算符等号 `=`，当然你也可以在运行时对变量进行赋值操作。

示例：

```
a = 15
b = false
```

一般情况下，只有类型相同的变量之间才可以相互赋值，例如：`a = b`。

声明与赋值（初始化）语句也可以组合起来。

示例：

```
var identifier [type] = value
var a int = 15
var i = 5
var b bool = false
var str string = "Go says hello to the world!"
```

但是 Go 编译器的智商已经高到可以根据变量的值来自动推断其类型，这有点像 Ruby 和 Python 这类动态语言，只不过它们是在运行时进行推断，而 Go 是在编译时就已经完成推断过程。因此，你还可以使用下面的这些形式来声明及初始化变量：

```
var a = 15
var b = false
var str = "Go says hello to the world!"
```

或:

```
var (  
    a = 15  
    b = false  
    str = "Go says hello to the world!"  
    numShips = 50  
    city string  
)
```

不过自动推断类型并不是任何时候都适用的，当你想要给变量的类型并不是自动推断出的某种类型时，你还是需要显式指定变量的类型，例如：`var n int64 = 2`。

然而，`var a` 这种语法是不正确的，因为编译器没有任何可以用于自动推断类型的依据。变量的类型也可以在运行时实现自动推断，例如：

```
var (  
    HOME = os.Getenv("HOME")  
    USER = os.Getenv("USER")  
    GOROOT = os.Getenv("GOROOT")  
)
```

这种写法主要用于声明包级别的全局变量，当你在函数体内声明局部变量时，应使用简短声明语法 `:=`，例如：`a := 1`。

下面这个例子展示了如何在运行时获取所在的操作系统类型，它通过 `os` 包中的函数 `os.Getenv()` 来获取环境变量中的值，并保存到 `string` 类型的局部变量 `path` 中。

Example 4.5 goos.go

```
package main  
  
import (  
    "fmt"  
    "os"  
)  
  
func main() {  
    var goos string = os.Getenv("GOOS")  
    fmt.Printf("The operating system is: %s\n", goos)  
    path := os.Getenv("PATH")  
    fmt.Printf("Path is %s\n", path)  
}
```

如果你在 Windows 下运行这段代码，则会输出 `The operating system is: windows` 以及相应的环境变量的值；如果你在 Linux 下运行这段代码，则会输出 `The operating system is: linux` 以及相应的的环境变量的值。

这里用到了 `Printf` 的格式化输出的功能（第 4.4.3 节）。

4.4.2 值类型和引用类型

程序中所用到的内存在计算机中使用一堆箱子来表示（这也是人们在讲解它的时候的画法），这些箱子被称为“字”。根据不同的处理器以及操作系统类型，所有的字都具有 32 位（4 字节）或 64 位（8 字节）的相同长度；所有的字都使用相关的内存地址来进行表示（以十六进制数表示）。

所有像 `int`，`float`，`bool`，`string` 这些基本类型都属于值类型，使用这些类型的变量直接指向存在内存中的值：

另外，像数组（第 7 章）和结构（第 10 章）这些复合类型也是值类型。

当使用等号 `=` 将一个变量的值赋值给另一个变量时，如：`j = i`，实际上是在内存中将 `i` 的值进行了拷贝：

你可以通过 `&i` 来获取变量 `i` 的内存地址（第 4.9 节），例如：`0xf840000040`（每次的地址都可能不一样）。值类型的变量的值存储在栈中。

内存地址会根据机器的不同而有所不同，甚至相同的程序在不同的机器上执行后也会有不同的内存地址。因为每台机器可能有不同的存储器布局，并且位置分配也可能不同。

更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。

一个引用类型的变量 `r1` 存储的是 `r1` 的值所在的内存地址（数字），或内存地址中第一个字所在的位置。

这个内存地址为称之为指针（你可以从上图中很清晰地看到，第 4.9 节将会详细说明），这个指针实际上也被存在另外的某一个字中。

同一个引用类型的指针指向的多个字可以是在连续的内存地址中（内存布局是连续的），这也是计算效率最高的一种存储形式；也可以将这些字分散存放在内存中，每个字都指示了下一个字所在的内存地址。

当使用赋值语句 `r2 = r1` 时，只有引用（地址）被复制。

如果 `r1` 的值被改变了，那么这个值的所有引用都会指向被修改后的内容，在这个例子中，`r2` 也会受到影响。

在 Go 语言中，指针（第 4.9 节）属于引用类型，其它的引用类型还包括 `slices`（第 7 章），`maps`（第 8 章）和 `channel`（第 13 章）。被引用的变量会存储在堆中，以便进行垃圾回收，且比栈拥有更大的内存空间。

4.4.3 打印

函数 `Printf` 可以在 `fmt` 包外部使用，这是因为它以大写字母 `P` 开头，该函数主要用于打印输出到控制台。通常使用的格式化字符串作为第一个参数：

```
func Printf(format string, list of variables to be printed)
```

在 Example 4.5 中，格式化字符串为：`"The operating system is: %s\n"`。

这个格式化字符串可以含有一个或多个的格式化标识符，例如：`%..`，其中 `..` 可以被不同类型所对应的标识符替换，如 `%s` 代表字符串标识符、`%v` 代表使用类型的默认输出格式的标识符。这些标识符所对应的值从格式化字符串后的第一个逗号开始按照相同顺序添加，如果参数超过 1 个则同样需要使用逗号分隔。使用这些占位符可以很好地控制格式化输出的文本。

函数 `fmt.Sprintf` 与 `Printf` 的作用是完全相同的，不过前者将格式化后的字符串以返回值的形式返回给调用者，因此你可以在程序中使用包含变量的字符串，具体例子可以参见 Example 15.4 `simple_tcp_server.go`。

函数 `fmt.Print` 和 `fmt.Println` 会自动使用格式化标识符 `%v` 对字符串进行格式化，两者都会在每个参数之间自动增加空格，而后者还会在字符串的最后加上一个换行符。例如：`fmt.Print("Hello:", 23)` 将输出：`Hello: 23`。

4.4.4 简短形式，使用 `:=` 赋值操作符

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，而这个时候再在 Example 4.4.1 的最后一个声明语句写上 `var` 关键字就显得有些多余了，因此我们可以将它们简写为 `a := 50` 或 `b := false`。

`a` 和 `b` 的类型（`int` 和 `bool`）将由编译器自动推断。

这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 `:=` 可以高效地创建一个新的变量，称之为初始化声明。

注意事项

如果在相同的代码块中，我们不可以再次对于相同名称的变量使用初始化声明，例如：`a := 20` 就是不被允许的，编译器会提示错误 `no new variables on left side of :=`，但是 `a = 20` 是可以的，因为这是给相同的变量赋予一个新的值。

如果你在定义变量 `a` 之前使用它，则会得到编译错误 `undefined: a`。

如果你声明了一个局部变量却没有在相同的代码块中使用它，同样会得到编译错误，例如下面这个例子当中的变量 `a`：

```
func main() {  
    var a string = "abc"  
    fmt.Println("hello, world")  
}
```

尝试编译这段代码将得到错误 `a declared and not used`。

此外，单纯地给 `a` 赋值也是不够的，这个值必须被使用，所以使用 `fmt.Println("hello, world", a)` 会移除错误。

但是全局变量是允许声明但不使用。

其他的简短形式为：

同一类型的多个变量可以声明在同一行，如：`var a, b, c int`。

(这是将类型写在标识符后面的一个重要原因)

多变量可以在同一行进行赋值，如：`a, b, c = 5, 7, "abc"`。

上面这行假设了变量 `a`, `b` 和 `c` 都被已经声明，否则的话应该这样使用：`a, b, c := 5, 7, "abc"`。

右边的这些值以相同的顺序赋值给左边的变量，所以 `a` 的值是 5，`b` 的值是 7，`c` 的值是 "abc"。

这被称为 **并行** 或 **同时** 赋值。

如果你想要交换两个变量的值，则可以简单地使用 `a, b = b, a`。

(在 Go 语言中，这样省去了使用交换函数的必要)

空白标识符 `_` 也被用于抛弃值，如值 5 在：`_, b = 5, 7` 中被抛弃。

`_` 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 `val` 和错误 `err` 是通过调用 `Func1` 函数同时得到：`val, err = Func1(var1)`。

4.4.5 init 函数

变量除了可以在全局声明中初始化，也可以在 `init()` 函数中初始化。这是一类非常特殊的函数，它不能够被人为调用，而是在每个包完成初始化后自动执行，并且执行优先级比 `main()` 函数高。

每一个源文件都可以包含且只包含一个 `init()` 函数。初始化总是以单线程执行，并且按照包的依赖关系顺序执行。

一个可能的用途是在开始执行程序之前对数据进行检验或修复，以保证程序状态的正确性。

Example 4.6 `init.go`:

```
package trans
import "math"
var Pi float64
func init() {
    Pi = 4 * math.Atan(1) // init() function computes Pi
}
```

在它的 `init()` 函数中计算变量 `Pi` 的初始值。

Example 4.7 `user_init.go` 中导入了包 `trans`（在相同的路径中）并且使用到了变量 `Pi`:

```
package main
import (
    "fmt"
    "./trans"
)
var twoPi = 2 * trans.Pi
func main() {
    fmt.Printf("2*Pi = %g\n", twoPi) // 2*Pi = 6.283185307179586
}
```

`init()` 函数也经常被用在当一个程序开始之前调用后台执行的 `goroutine`，如下面这个例子当中的 `backend()`：

```
func init() {
    // setup preparations
    go backend()
}
```

练习 推断以下程序的输出，并解释你的答案，然后编译并执行它们。

练习 4.1 local_scope.go:

```
package main
var a = "G"
func main() {
    n()
    m()
    n()
}
func n() { print(a) }
func m() {
    a := "0"
    print(a)
}
```

练习 4.2 global_scope.go:

```
package main
var a = "G"
func main() {
    n()
    m()
    n()
}
func n() {
    print(a)
}
func m() {
    a = "0"
    print(a)
}
```

练习 4.3 function_calls_function.go

```
package main
var a string
func main() {
    a = "G"
    print(a)
    f1()
}
func f1() {
    a := "0"
    print(a)
    f2()
}
func f2() {
    print(a)
}
```

4.5 基本类型和运算符

我们将在这个部分讲解有关布尔型、数字型和字符型的相关知识。

表达式是一种特定的类型的值，它可以由其它的值以及运算符组合而成。每个类型都定义了可以和自己结合的运算符集合，如果你使用了不在这个集合中的运算符，则会在编译时获得编译错误。

一元运算符只可以用于一个值的操作（作为后缀），而二元运算符则可以和两个值或者操作数结合（作为中缀）。

只有两个类型相同的值才可以和二元运算符结合，另外要注意的是，Go 是强类型语言，因此不会进行隐式转换，任何不同类型之间的转换都必须显式说明（第 4.2 节）。Go 不存在像 C 和 Java 那样的运算符重载，表达式的解析顺序是从左至右。

你可以在第 4.5.3 节找到有关运算符优先级的相关信息，优先级越高的运算符在条件相同的情况下将被优先执行。但是你可以通过使用括号将其中的表达式括起来，以人为地提升某个表达式的运算优先级。

4.5.1 布尔类型 bool

一个简单的例子：`var b bool = true`。

布尔型的值只可以是常量 `true` 或者 `false`。

两个类型相同的值可以使用相等 `==` 或者不等 `!=` 运算符来进行比较并获得一个布尔型的值。

当相等运算符两边的值是完全相同的值的时候会返回 `true`，否则返回 `false`，并且只有在两个的值的类型相同的情况下才可以使用。

示例：

```
var aVar = 10
aVar == 5 -> false
aVar == 10 -> true
```

当不等运算符两边的值是不同的时候会返回 `true`，否则返回 `false`。

示例：

```
var aVar = 10
aVar != 5 -> true
aVar != 10 -> false
```

Go 对于值之间的比较有非常严格的限制，只有两个类型相同的值才可以进行比较，如果值的类型是接口（`interface`，第 11 章），它们也必须都实现了相同的接口。如果其中一个值是常量，那么另外一个值的类型必须和该常量类型相兼容的。如果以上条件都不满足，则其中一个值的类型必须在被转换为和另外一个值的类型相同之后才可以进行比较。

布尔型的常量和变量也可以通过和逻辑运算符（非 `!`、和 `&&`、或 `||`）结合来产生另外一个布尔值，这样的逻辑语句就其本身而言，并不是一个完整的 Go 语句。

逻辑值可以被用于条件结构中的条件语句（第 5 章），以便测试某个条件是否满足。另外，和 `&&`、或 `||` 与相等 `==` 或不等 `!=` 属于二元运算符，而非 `!` 属于一元运算符。在接下来的内容中，我们会使用 `T` 来代表条件符合的语句，用 `F` 来代表条件不符合的语句。

Go 语言中包含以下逻辑运算符：

非运算符：`!`

```
!T -> false
!F -> true
```

非运算符用于取得和布尔值相反的结果。

和运算符：`&&`

```
T && T -> true
T && F -> false
F && T -> false
F && F -> false
```

只有当两边的值都为 `true` 的时候，和运算符的结果才是 `true`。

或运算符：`||`

```
T || T -> true
T || F -> true
F || T -> true
F || F -> false
```

只有当两边的值都为 `false` 的时候，或运算符的结果才是 `false`，其中任意一边的值为 `true` 就能够使得该表达式的结果为 `true`。

在 Go 语言中，`&&` 和 `||` 是具有快捷性质的运算符，当运算符左边表达式的值已经能够决定整个表达式的值的时候（`&&` 左边的值为 `false`，`||` 左边的值为 `true`），运算符右边的表达式将不会被执行。利用这个性质，如果你有多个条件判断，应当将计算过程较为复杂的表达式放在运算符的右侧以减少不必要的运算。

利用括号同样可以升级某个表达式的运算优先级。

在格式化输出时，你可以使用 %t 来表示你要输出的值为布尔型。

布尔值（以及任何结果为布尔值的表达式）最常用在条件结构的条件语句中，例如：if、for 和 switch 结构（第 5 章）。

对于布尔值的好的命名能够很好地提升代码的可读性，例如以 is 或者 Is 开头的 isSorted、isFinished、isVisible，使用这样的命名能够在阅读代码的获得阅读正常语句一样的良好体验，例如标准库中的 unicode.IsDigit(ch)（第 4.5.5 节）。

4.5.2 数字类型

4.5.2.1 整型 int 和浮点型 float

Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码（二的补码，详情参见：http://en.wikipedia.org/wiki/Two's_complement）。

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

这些类型的长度都是根据运行程序所在的操作系统类型所决定的：

- int 和 uint 在 32 位操作系统上，它们均使用 32 位（4 个字节），在 64 位操作系统上，它们均使用 64 位（8 个字节）。
- uintptr 的长度被设定为足够存放一个指针即可。

Go 语言中没有 float 类型。

与操作系统架构无关的类型都有固定的大小，并在类型的名称中就可以看出来：

整数：

- int8 (-128 -> 127)
- int16 (-32768 -> 32767)
- int32 (-2,147,483,648 -> 2,147,483,647)
- int64 (-9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807)

无符号整数：

- uint8 (0 -> 255)
- uint16 (0 -> 65,535)
- uint32 (0 -> 4,294,967,295)
- uint64 (0 -> 18,446,744,073,709,551,615)

浮点型（IEEE-754 标准）：

- float32 (+- 1e-45 -> +- 3.4 * 1e38)
- float64 (+- 5 * 1e-324 -> 107 * 1e308)

int 型是计算最快的一种类型。

整型的零值为 0，浮点型的零值为 0.0。

float32 精确到小数点后 7 位，float64 精确到小数点后 15 位。由于精确度的缘故，你在使用 == 或者 != 来比较浮点数时应当非常小心。你最好在正式使用前测试对于精确度要求较高的运算。

你应该尽可能地使用 float64，因为 math 包中所有有关数学运算的函数都会要求接收这个类型。

你可以通过增加前缀 0 来表示 8 进制数（如：077），增加前缀 0x 来表示 16 进制数（如：0xFF），以及使用 e 来表示 10 的连乘（如：1e3 = 1000，或者 6.022e23 = 6.022 x 1e23）。

你可以使用 a := uint64(0) 来同时完成类型转换和赋值操作，这样 a 的类型就是 uint64。

Go 中不允许不同类型之间的混合使用，但是对于常量的类型限制非常少，因此允许常量之间的混合使用，下面这个程序很好地解释了这个现象（该程序无法通过编译）：

Example 4.8 [type_mixing.go](#)

```
package main
```

```
func main() {
    var a int
    var b int32
    a = 15
    b = a + a    // 编译错误
    b = b + 5    // 因为 5 是常量，所以可以通过编译
}
```

如果你尝试编译该程序，则将得到编译错误 `cannot use a + a (type int) as type int32 in assignment`。

同样地，`int16` 也不能够被隐式转换为 `int32`。

下面这个程序展示了通过显示转换来避免这个问题（第 4.2 节）。

Example 4.9 [casting.go](#)

```
package main

import "fmt"

func main() {
    var n int16 = 34
    var m int32
    // compiler error: cannot use n (type int16) as type int32 in assignment
    //m = n
    m = int32(n)

    fmt.Printf("32 bit int is: %d\n", m)
    fmt.Printf("16 bit int is: %d\n", n)
}

// the output is:
32 bit int is: 34
16 bit int is: 34
```

格式化说明符

在格式化字符串里，`%d` 用于格式化整数（`%x` 和 `%X` 用于格式化 16 进制表示的数字），`%g` 用于格式化浮点型（`%f` 输出浮点数，`%e` 输出科学计数表示法），`%0d` 用于规定输出定长的整数，其中开头的数字 0 是必须的。

`%n.mg` 用于表示数字 `n` 并精确到小数点后 `m` 位，除了使用 `g` 之外，还可以使用 `e` 或者 `f`，例如：使用格式化字符串 `%5.2e` 来输出 3.4 的结果为 `3.40e+00`。

数字值转换

当进行类似 `a32bitInt = int32(a32Float)` 的转换时，小数点后的数字将被丢弃。这种情况一般发生当从取值范围较大的类型转换为取值范围较小的类型时，或者你可以写一个专门用于处理类型转换的函数来确保没有发生精度的丢失。下面这个例子展示如何安全地从 `int` 型转换为 `int8`：

```
func Uint8FromInt(n int) (uint8, error) {
    if 0 <= n && n <= math.MaxUint8 { // conversion is safe
        return uint8(n), nil
    }
    return 0, fmt.Errorf("%d is out of the uint8 range", n)
}
```

或者安全地从 `float64` 转换为 `int`：

```
func IntFromFloat64(x float64) int {
    if math.MinInt32 <= x && x <= math.MaxInt32 { // x lies in the integer range
        whole, fraction := math.Modf(x)
        if fraction >= 0.5 {
            whole++
        }
        return int(whole)
    }
    panic(fmt.Sprintf("%g is out of the int32 range", x))
}
```

不过如果你实际存的数字超出你要转换到的类型的取值范围的话，则会引发 panic（第 13.2 节）。

问题 4.1 int 和 int64 是相同的类型吗？

4.5.2.2 复数

Go 拥有以下复数类型：

```
complex64 (32 位实数和虚数)
complex128 (64 位实数和虚数)
```

复数使用 re+imI 来表示，其中 re 代表实数部分，im 代表虚数部分，I 代表根号负 1。

示例：

```
var c1 complex64 = 5 + 10i
fmt.Printf("The value is: %v", c1)
// 输出: 5 + 10i
```

如果 re 和 im 的类型均为 float32，那么类型为 complex64 的复数 c 可以通过以下方式来获得：

```
c = complex(re, im)
```

函数 real(c) 和 imag(c) 可以分别获得相应的实数和虚数部分。

在使用格式化说明符时，可以使用 %v 来表示复数，但当你希望只表示其中的一个部分的时候需要使用 %f。

复数支持和其它数字类型一样的运算。当你使用等号 == 或者不等号 != 对复数进行比较运算时，注意对精确度的把握。cmath 包中包含了一些操作复数的公共方法。如果你对内存的要求不是特别高，最好使用 complex128 作为计算类型，因为相关函数都使用这个类型的参数。

4.5.2.3 位运算

位运算只能用于整数类型的变量，且需当它们拥有等长位模式时。

%b 是用于表示位的格式化标识符。

二元运算符

- 按位与 &：

对应位置上的值经过和运算结果，具体参见和运算符，第 4.5.1 节，并将 T (true) 替换为 1，将 F (false) 替换为 0

```
1 & 1 -> 1
1 & 0 -> 0
0 & 1 -> 0
0 & 0 -> 0
```

- 按位或 |：

对应位置上的值经过或运算结果，具体参见或运算符，第 4.5.1 节，并将 T (true) 替换为 1，将 F (false) 替换为 0

```
1 | 1 -> 1
1 | 0 -> 1
0 | 1 -> 1
0 | 0 -> 0
```

- 按位异或 ^：

对应位置上的值根据以下规则组合：

```
1 ^ 1 -> 0
```

```
1 ^ 0 -> 1
0 ^ 1 -> 1
0 ^ 0 -> 0
```

- 位清除 &^：将指定位置上的值设置为 0。

一元运算符

- 按位补足 ^：

该运算符与异或运算符一同使用，即 m^x ，对于无符号 x 使用“全部位设置为 1”，对于有符号 x 时使用 $m=-1$ 。例如：

```
^2 = ^10 = -01 ^ 10 = -11
```

- 位左移 <<：
 - 用法：bitP << n。
 - bitP 的位向左移动 n 位，右侧空白部分使用 0 填充；如果 n 等于 2，则结果是 2 的相应倍数，即 2 的 n 次方。例如：

```
1 << 10 // 等于 1 KB
1 << 20 // 等于 1 MB
1 << 30 // 等于 1 GB
```

- 位右移 >>：
 - 用法：bitP >> n。
 - bitP 的位向右移动 n 位，左侧空白部分使用 0 填充；如果 n 等于 2，则结果是当前值除以 2 的 n 次方。

当希望把结果赋值给第一个操作数时，可以简写为 $a \ll= 2$ 或者 $b \wedge= a \& 0xffffffff$ 。

位左移常见实现存储单位的用例

使用位左移与 iota 计数配合可优雅地实现存储单位的常量枚举：

```
type ByteSize float64
const (
    _ = iota // 通过赋值给空白标识符来忽略值
    KB ByteSize = 1<<(10*iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

在通讯中使用位左移表示标识的用例

```
type BitFlag int
const (
    Active BitFlag = 1 << iota // 1 << 0 == 1
    Send // 1 << 1 == 2
    Receive // 1 << 2 == 4
)

flag := Active | Send // == 3
```

4.5.2.4 逻辑运算符

Go 中拥有以下逻辑运算符： $==$ 、 $!=$ （第 4.5.1 节）、 $<$ 、 $<=$ 、 $>$ 、 $>=$ 。

它们之所以被称为逻辑运算符是因为它们的运算结果总是为布尔值 bool。例如：


```
b3:= 10 > 5 // b3 is true
```

4.5.2.5 算术运算符

常见可用于整数和浮点数的二元运算符有 +、-、* 和 /。

（相对于一般规则而言，Go 在进行字符串拼接时允许使用对运算符 + 的重载，但 Go 本身不允许开发者进行自定义的运算符重载）

/ 对与整数运算而言，结果依旧为整数，例如：9 / 4 -> 2。

取余运算符只能作用于整数：9 % 4 -> 1。

整数除以 0 可能导致程序崩溃，将会导致运行时的恐慌状态（如果除以 0 的行为在编译时就能被捕捉到，则会引发编译错误）；第 13 章将会详细讲解如何正确地处理此类情况。

浮点数除以 0.0 会返回一个无穷尽的结果，使用 +Inf 表示。

练习 4.4 尝试编译 [divby0.go](#)。

你可以将语句 `b = b + a` 简写为 `b+=a`，同样的写法也可用于 `--`、`*=`、`/=`、`%=`。

对于整数和浮点数，你可以使用一元运算符 `++`（递增）和 `--`（递减），但只能用于后缀：

```
i++ -> i += 1 -> i = i + 1
i-- -> i -= 1 -> i = i - 1
```

同时，带有 `++` 和 `--` 的只能作为语句，而非表达式，因此 `n = i++` 这种写法是无效的，其它像 `f(i++)` 或者 `a[i]=b[i++]` 这些可以用于 C、C++ 和 Java 中的写法在 Go 中也是不允许的。

在运算时 **溢出** 不会产生错误，Go 会简单地将超出位数抛弃。如果你需要范围无限大的整数或者有理数（意味着只被限制于计算机内存），你可以使用标准库中的 `big` 包，该包提供了类似 `big.Int` 和 `big.Rat` 这样的类型（第 9.4 节）。

4.5.2.6 随机数

一些像游戏或者统计学类的应用需要用到随机数。`rand` 包实现了伪随机数的生成。

Example 4.10 [random.go](#) 演示了如何生成 10 个非负随机数：

```
package main
import (
    "fmt"
    "rand"
    "time"
)

func main() {
    for i := 0; i < 10; i++ {
        a := rand.Int()
        fmt.Printf("%d / ", a)
    }
    for i := 0; i < 5; i++ {
        r := rand.Intn(8)
        fmt.Printf("%d / ", r)
    }
    fmt.Println()
    timens := int64(time.Now().Nanosecond())
    rand.Seed(timens)
    for i := 0; i < 10; i++ {
        fmt.Printf("%.2f / ", 100*rand.Float32())
    }
}
```

可能的输出：

```
816681689 / 1325201247 / 623951027 / 478285186 / 1654146165 /
1951252986 / 2029250107 / 762911244 / 1372544545 / 591415086 / / 3 / 0 / 6 / 4 / 2 / 22.10
```

函数 `rand.Float32` 和 `rand.Float64` 返回介于 `[0.0, 1.0)` 之间的伪随机数，其中包括 `0.0` 但不包括 `1.0`。函数 `rand.Intn` 返回介于 `[0, n)` 之间的伪随机数。

你可以使用 `Seed(value)` 函数来提供伪随机数的生成种子，一般情况下都会使用当前时间的纳秒级数字（第 4.8 节）。

4.5.3 运算符与优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

优先级	运算符
7	<code>^</code> <code>!</code>
6	<code>*</code> <code>/</code> <code>%</code> <code><<</code> <code>>></code> <code>&</code> <code>&^</code>
5	<code>+</code> <code>-</code> <code> </code> <code>^</code>
4	<code>==</code> <code>!=</code> <code><</code> <code><=</code> <code>>=</code> <code>></code>
3	<code><-</code>
2	<code>&&</code>
1	<code> </code>

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

4.5.4 类型别名

当你在使用某个类型时，你可以给它起另一个名字，然后你就可以在你的代码中使用新的名字（用于简化名称或解决名称冲突）。

在 `type TZ int` 中，`TZ` 就是 `int` 类型的新名称（用于表示程序中的时区），然后就可以使用 `TZ` 来操作 `int` 类型的数据。

Example 4.11 [type.go](#)

```
package main
import "fmt"

type TZ int

func main() {
    var a, b TZ = 3, 4
    c := a + b
    fmt.Printf("c has the value: %d", c) // 输出: c has the value: 7
}
```

实际上，类型别名得到的新类型并非和原类型完全相同，新类型不会拥有原类型所附带的方法（第 10 章）；`TZ` 可以自定义一个方法用来输出更加人性化的时区信息。

练习 4.5 定义一个 `string` 的类型别名 `Rope`，并声明一个该类型的变量。

4.5.5 字符类型

严格来说，这并不是 Go 语言的一个类型，字符只是整数的特殊用例。`byte` 类型是 `uint8` 的别名，对于只占用 1 个字节的传统 ASCII 编码的字符来说，完全没有问题。例如：`var ch byte = 'A'`；字符使用单引号括起来。

在 ASCII 码表中，`A` 的值是 65，而使用 16 进制表示则为 41，所以下面的写法是等效的：

```
var ch byte = 65 或 var ch byte = '\x41'
```

（`\x` 总是紧跟着长度为 2 的 16 进制数）

另外一种可能的写法是 `\` 后面紧跟着长度为 3 的十进制数，例如：`\377`。

不过 Go 同样支持 Unicode（UTF-8），因此字符同样称为 Unicode 代码点或者 `runes`，并在内存中使用 `int` 来表示。在文档中，一般使用格式 `U+hhhh` 来表示，其中 `h` 表示一个 16 进制数。其实 `rune` 也是 Go 当中的一个类型，并且是 `int32` 的别名。

在书写 Unicode 字符时，需要在 16 进制数之前加上前缀 `\u` 或者 `\U`。

因为 Unicode 至少占用 2 个字节，所以我们使用 `int16` 或者 `int` 类型来表示。如果需要使用到 4 字节，则会加上 `\U` 前缀；前缀 `\u` 则总是紧跟着长度为 4 的 16 进制数，前缀 `\U` 紧跟着长度为 8 的 16 进制数。

Example 4.12 [char.go](https://play.golang.org/p/0000000000)

```
var ch int = '\u0041'
var ch2 int = '\u03B2'
var ch3 int = '\U00101234'
fmt.Printf("%d - %d - %d\n", ch, ch2, ch3) // integer
fmt.Printf("%c - %c - %c\n", ch, ch2, ch3) // character
fmt.Printf("%X - %X - %X\n", ch, ch2, ch3) // UTF-8 bytes
fmt.Printf("%U - %U - %U", ch, ch2, ch3) // UTF-8 code point
```

输出：

```
65 - 946 - 1053236
A - β - r
41 - 3B2 - 101234
U+0041 - U+03B2 - U+101234
```

格式化说明符 `%c` 用于表示字符；当和字符配合使用时，`%v` 或 `%d` 会输出用于表示该字符的整数；`%U` 输出格式为 `U+hhhh` 的字符串（另一个示例见第 5.4.4 节）。

包 `unicode` 包含了一些针对测试字符的非常有用的函数（其中 `ch` 代表字符）：

```
判断是否为字母：    unicode.IsLetter(ch)
判断是否为数字：    unicode.IsDigit(ch)
判断是否为空白符号： unicode.IsSpace(ch)
```

这些函数返回一个布尔值。包 `utf8` 拥有更多与 `rune` 相关的函数。

（译者注：关于类型的相关讲解，可参考视频教程《Go编程基础》第3课：[类型与变量](#)）

4.6 字符串

字符串是 UTF-8 字符的一个序列（当字符为 ASCII 码时则占用 1 个字节，其它字符根据需要占用 2-4 个字节）。UTF-8 是被广泛使用的编码格式，是文本文件的标准编码，其它包括 XML 和 JSON 在内，也都使用该编码。由于该编码对占用字节长度的不定性，Go 中的字符串也可能根据需要占用 1 至 4 个字节（示例见第 4.6 节），这与其它语言如 C++、Java 或者 Python 不同（Java 始终使用 2 个字节）。Go 这样做的好处是不仅减少了内存和硬盘空间占用，同时也不用像其它语言那样需要对使用 UTF-8 字符集的文本进行编码和解码。

字符串是一种值类型，且值不可变，即创建某个文本后你无法再次修改这个文本的内容；更深入地讲，字符串是字节的定长数组。

Go 支持以下 2 种形式的字面值：

- 解释字符串：

该类字符串使用双引号括起来，其中的相关的转义字符将被替换，这些转义字符包括：

- `\n`：换行符
- `\r`：回车符
- `\t`：tab 键
- `\u` 或 `\U`：Unicode 字符
- `\\`：反斜杠自身

- 非解释字符串：

该类字符串使用反引号括起来，支持换行，例如：

```
`This is a raw string \n` 中的 `\\n` 会被原样输出。
```

和 C/C++ 一样，Go 中的字符串是根据长度限定，而非特殊字符。

string 类型的零值为长度为零的字符串，即空字符串 ""。

一般的比较运算符（==、!=、<、<=、>=、>）通过在内存中按字节比较来实现字符串的对比。你可以通过函数 len() 来获取字符串所占的字节长度，例如：len(str)。

字符串的内容（纯字节）可以通过标准索引法来获取，在中括号 [] 内写入索引，索引从 0 开始计数。

```
字符串 str 的第 1 个字节:    str[0]
第 i 个字节:                str[i]
最后 1 个字节:              str[len(str)-1]
```

需要注意的是，这种转换方案只对纯 ASCII 码的字符串有效。

注意事项 获取字符串中某个字节的地址的行为是非法的，例如：&str[i]。

字符串拼接符 +

两个字符串 s1 和 s2 可以通过 s := s1 + s2 拼接在一起。

s2 追加在 s1 尾部并生成一个新的字符串 s。

你可以通过以下方式对代码中多行的字符串进行拼接：

```
str := "Beginning of the string "+
      "second part of the string"
```

由于编译器行尾自动补全分号的缘故，加号 + 必须放在第一行。

拼接的简写形式 += 也可以用于字符串：

```
s := "hel" + "lo,"
s += "world!"
fmt.Println(s) //输出 "hello, world!"
```

在循环中使用加号 + 拼接字符串并不是最高效的做法，更好的办法是使用函数 strings.Join()（第 4.7.10 节），有没有更好的办法了？有！使用字节缓冲（bytes.Buffer）拼接更加给力（第 7.2.6 节）！

在第 7 章，我们会讲到通过将字符串看作是字节（byte）的切片（slice）来实现对其标准索引法的操作。会在第 5.4.1 节中讲到的 for 循环只会根据索引返回字符串中的纯字节，而在第 5.4.4 节（以及第 7.6.1 节的示例）将会展示如何使用 for-range 循环来实现对 Unicode 字符串的迭代操作。在下一节，我们会学习到许多有关字符串操作的函数和方法，同时 fmt 包中的 fmt.Sprintf(x) 也可以格式化生成并返回你所需要的字符串（第 4.4.3 节）。

练习 4.6 [count_characters.go](#)

创建一个用于统计字节和字符（rune）的程序，并对字符串 asSASA ddd dsjkdsjs dk 进行分析，然后再分析 asSASA ddd dsjkdsjsこん dk，最后解释两者不同的原因（提示：使用 unicode/utf8 包）。

4.7 strings 和 strconv 包

作为一种基本数据结构，每种语言都有一些对于字符串的预定义处理函数。Go 中使用 strings 包来完成对字符串的主要操作。

4.7.1 前缀和后缀

HasPrefix 判断字符串 s 是否以 prefix 开头：

```
strings.HasPrefix(s, prefix string) bool
```

HasSuffix 判断字符串 s 是否以 suffix 结尾：

```
strings.HasSuffix(s, suffix string) bool
```

Example 4.13 [presuffix.go](#)

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var str string = "This is an example of a string"
    fmt.Printf("T/F? Does the string \"%s\" have prefix %s? ", str, "Th")
    fmt.Printf("%t\n", strings.HasPrefix(str, "Th"))
}
```

输出:

```
T/F? Does the string "This is an example of a string" have prefix Th? true
```

这个例子同样演示了转移字符 \ 和格式化字符串的使用。

4.7.2 字符串包含关系

Contains 判断字符串 s 是否包含 substr :

```
strings.Contains(s, substr string) bool
```

4.7.3 判断子字符串或字符在父字符串中出现的位置（索引）

Index 返回字符串 str 在字符串 s 中的索引（str 的第一个字符的索引），-1 表示字符串 s 不包含字符串 str :

```
strings.Index(s, str string) int
```

LastIndex 返回字符串 str 在字符串 s 中最后出现位置的索引（str 的第一个字符的索引），-1 表示字符串 s 不包含字符串 str :

```
strings.LastIndex(s, str string) int
```

如果 ch 是非 ASCII 编码的字符，建议使用以下函数来对字符进行定位：

```
strings.IndexRune(s string, ch int) int
```

Example 4.14 [index_in_string.go](#)

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var str string = "Hi, I'm Marc, Hi."

    fmt.Printf("The position of \"Marc\" is: ")
    fmt.Printf("%d\n", strings.Index(str, "Marc"))

    fmt.Printf("The position of the first instance of \"Hi\" is: ")
    fmt.Printf("%d\n", strings.Index(str, "Hi"))
    fmt.Printf("The position of the last instance of \"Hi\" is: ")
    fmt.Printf("%d\n", strings.LastIndex(str, "Hi"))
}
```



```
    fmt.Printf("The position of \"Burger\" is: ")
    fmt.Printf("%d\n", strings.Index(str, "Burger"))
}
```

输出：

```
The position of "Marc" is: 8
The position of the first instance of "Hi" is: 0
The position of the last instance of "Hi" is: 14
The position of "Burger" is: -1
```

4.7.4 字符串替换

Replace 用于将字符串 str 中的前 n 个字符串 old 替换为字符串 new，并返回一个新的字符串，如果 n = -1 则替换所有字符串 old 为字符串 new：

```
strings.Replace(str, old, new, n) string
```

4.7.5 统计字符串出现次数

Count 用于计算字符串 str 在字符串 s 中出现的非重叠次数：

```
strings.Count(s, str string) int
```

Example 4.15 [count_substring.go](#)

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var str string = "Hello, how is it going, Hugo?"
    var manyG = "gggggggggg"

    fmt.Printf("Number of H's in %s is: ", str)
    fmt.Printf("%d\n", strings.Count(str, "H"))

    fmt.Printf("Number of double g's in %s is: ", manyG)
    fmt.Printf("%d\n", strings.Count(manyG, "gg"))
}
```

输出：

```
Number of H's in Hello, how is it going, Hugo? is: 2
Number of double g's in gggggggggg is: 5
```

4.7.6 重复字符串

Repeat 用于重复 count 次字符串 s 并返回一个新的字符串：

```
strings.Repeat(s, count int) string
```

Example 4.16 [repeat_string.go](#)

```
package main

import (
```

```

    "fmt"
    "strings"
)

func main() {
    var origS string = "Hi there! "
    var newS string

    newS = strings.Repeat(origS, 3)
    fmt.Printf("The new repeated string is: %s\n", newS)
}

```

输出:

```
The new repeated string is: Hi there! Hi there! Hi there!
```

4.7.7 修改字符串大小写

ToLower 将字符串中的 Unicode 字符全部转换为相应的小写字符:

```
strings.ToLower(s) string
```

ToUpper 将字符串中的 Unicode 字符全部转换为相应的大写字符:

```
strings.ToUpper(s) string
```

Example 4.17 [toupper_lower.go](#)

```

package main

import (
    "fmt"
    "strings"
)

func main() {
    var orig string = "Hey, how are you George?"
    var lower string
    var upper string

    fmt.Printf("The original string is: %s\n", orig)
    lower = strings.ToLower(orig)
    fmt.Printf("The lowercase string is: %s\n", lower)
    upper = strings.ToUpper(orig)
    fmt.Printf("The uppercase string is: %s\n", upper)
}

```

输出:

```
The original string is: Hey, how are you George?
The lowercase string is: hey, how are you george?
The uppercase string is: HEY, HOW ARE YOU GEORGE?
```

4.7.8 修剪字符串

你可以使用 `strings.TrimSpace(s)` 来剔除字符串开头和结尾的空白符号; 如果你想要剔除指定字符, 则可以使用 `strings.Trim(s, "cut")` 来将开头和结尾的 `cut` 去除掉。该函数的第二个参数可以包含任何字符, 如果你只想剔除开头或者结尾的字符串, 则可以使用 `TrimLeft` 或者 `TrimRight` 来实现。

4.7.9 分割字符串

`strings.Fields(s)` 将会利用 1 个或多个空白符号来作为动态长度的分隔符将字符串分割成若干小块，并返回一个 slice，如果字符串只包含空白符号，则返回一个长度为 0 的 slice。

`strings.Split(s, sep)` 用于自定义分割符号来对指定字符串进行分割，同样返回 slice。

因为这 2 个函数都会返回 slice，所以习惯使用 for-range 循环来对其进行处理（第 7.3 节）。

4.7.10 拼接 slice 到字符串

`Join` 用于将元素类型为 `string` 的 slice 使用分割符号来拼接组成一个字符串：

```
Strings.Join(s1 []string, sep string)
```

Example 4.18 [strings_splitjoin.go](#)

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str := "The quick brown fox jumps over the lazy dog"
    s1 := strings.Fields(str)
    fmt.Printf("Splitted in slice: %v\n", s1)
    for _, val := range s1 {
        fmt.Printf("%s - ", val)
    }
    fmt.Println()
    str2 := "GO1|The ABC of Go|25"
    s2 := strings.Split(str2, "|")
    fmt.Printf("Splitted in slice: %v\n", s2)
    for _, val := range s2 {
        fmt.Printf("%s - ", val)
    }
    fmt.Println()
    str3 := strings.Join(s2, ";")
    fmt.Printf("s2 joined by ;: %s\n", str3)
}
```

输出：

```
Splitted in slice: [The quick brown fox jumps over the lazy dog]
The - quick - brown - fox - jumps - over - the - lazy - dog -
Splitted in slice: [GO1 The ABC of Go 25]
GO1 - The ABC of Go - 25 -
s2 joined by ;: GO1;The ABC of Go;25
```

其它有关字符串操作的文档请参阅官方文档 <http://golang.org/pkg/strings/>（译者注：国内用户可访问 <http://docs.studygolang.com/pkg/strings/>）。

4.7.11 从字符串中读取内容

函数 `strings.NewReader(str)` 用于生成一个 `Reader` 并读取字符串中的内容，然后返回指向该 `Reader` 的指针，从其它类型读取内容的函数还有：

- `Read()` 从 `[]byte` 中读取内容。
- `ReadByte()` 和 `ReadRune()` 从字符串中读取下一个 `byte` 或者 `rune`。

4.7.12 字符串与其它类型的转换

与字符串相关的类型转换都是通过 `strconv` 包实现的。

该包包含了一些变量用于获取程序运行的操作系统平台下 `int` 类型所占的位数，如：`strconv.IntSize`。

任何类型 **T** 转换为字符串总是成功的。

针对从数字类型转换到字符串，Go 提供了以下函数：

- `strconv.Itoa(i int) string` 返回数字 `i` 所表示的字符串类型的十进制数。
- `strconv.FormatFloat(f float64, fmt byte, prec int, bitSize int) string` 将 64 位浮点型的数字转换为字符串，其中 `fmt` 表示格式（其值可以是 'b'、'e'、'f' 或 'g'），`prec` 表示精度，`bitSize` 则使用 32 表示 `float32`，用 64 表示 `float64`。

将字符串转换为其它类型 **tp** 并不总是可能的，可能会在运行时抛出错误 `parsing "...": invalid argument`。

针对从字符串类型转换为数字类型，Go 提供了以下函数：

- `strconv.Atoi(s string) (i int, err error)` 将字符串转换为 `int` 型。
- `strconv.ParseFloat(s string, bitSize int) (f float64, err error)` 将字符串转换为 `float64` 型。

利用多返回值的特性，这些函数会返回 2 个值，第 1 个是转换后的结果（如果转换成功），第 2 个是可能出现的错误，因此，我们一般使用以下形式来进行从字符串到其它类型的转换：

```
val, err = strconv.Atoi(s)
```

在下面这个示例中，我们忽略可能出现的转换错误：

Example 4.19 [string_conversion.go](#)

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var orig string = "666"
    var an int
    var newS string

    fmt.Printf("The size of ints is: %d\n", strconv.IntSize)

    an, _ = strconv.Atoi(orig)
    fmt.Printf("The integer is: %d\n", an)
    an = an + 5
    newS = strconv.Itoa(an)
    fmt.Printf("The new string is: %s\n", newS)
}
```

输出：

```
The size of ints is: 32
The integer is: 666
The new string is: 671
```

在第 5.1 节，我们将会利用 `if` 语句来对可能出现的错误进行分类处理。

更多有关该包的讨论，请参阅官方文档 <http://golang.org/pkg/strconv/>（译者注：国内用户可访问 <http://docs.studygolang.com/pkg/strconv/>）。

4.8 时间和日期

`time` 包为我们提供了一个数据类型 `time.Time`（作为值使用）以及显示和测量时间和日期的功能函数。

当前时间可以使用 `time.Now()` 获取，或者使用 `t.Day()`、`t.Minute()` 等等来获取时间的一部分；你甚至可以自定义时间格式化字符串，例如：`fmt.Printf("%02d.%02d.%4d\n", t.Day(), t.Month(), t.Year())` 将会输出 `21.07.2011`。

`Duration` 类型表示两个连续时刻所相差的纳秒数，类型为 `int64`。`Location` 类型映射某个时区的时间，`UTC` 表示通用协调世界时间。

包中的一个预定义函数 `func (t Time) Format(layout string) string` 可以根据一个格式化字符串来将一个时间 `t` 转换为相应格式的字符串，你可以使用一些预定义的格式，如：`time.ANSIC` 或 `time.RFC822`。

一般的格式化设计是通过对于一个标准时间的格式化描述来展现的，这听起来很奇怪，但看下面这个例子你就会一目了然：

```
fmt.Println(t.Format("02 Jan 2006 15:04"))
```

输出：

```
21 Jul 2011 10:31
```

其它有关时间操作的文档请参阅官方文档 <http://golang.org/pkg/time/>（译者注：国内用户可访问 <http://docs.studygolang.com/pkg/time/>）。

Example 4.20 [time.go](#)

```
package main
import (
    "fmt"
    "time"
)

var week time.Duration
func main() {
    t := time.Now()
    fmt.Println(t) // e.g. Wed Dec 21 09:52:14 +0100 RST 2011
    fmt.Printf("%02d.%02d.%4d\n", t.Day(), t.Month(), t.Year())
    // 21.12.2011
    t = time.Now().UTC()
    fmt.Println(t) // Wed Dec 21 08:52:14 +0000 UTC 2011
    fmt.Println(time.Now()) // Wed Dec 21 09:52:14 +0100 RST 2011
    // calculating times:
    week = 60 * 60 * 24 * 7 * 1e9 // must be in nanosec
    week_from_now := t.Add(week)
    fmt.Println(week_from_now) // Wed Dec 28 08:52:14 +0000 UTC 2011
    // formatting times:
    fmt.Println(t.Format(time.RFC822)) // 21 Dec 11 0852 UTC
    fmt.Println(t.Format(time.ANSIC)) // Wed Dec 21 08:56:34 2011
    fmt.Println(t.Format("02 Jan 2006 15:04")) // 21 Dec 2011 08:52
    s := t.Format("20060102")
    fmt.Println(t, "=>", s)
    // Wed Dec 21 08:52:14 +0000 UTC 2011 => 20111221
}
```

输出的结果已经写在每行 `//` 的后面。

如果你需要在应用程序在经过一定时间或周期执行某项任务（事件处理的特例），则可以使用 `time.After` 或者 `time.Ticker`：我们将会在第 14.5 节讨论这些有趣的事情。另外，`time.Sleep (Duration d)` 可以实现对某个进程（实质上是 `goroutine`）时长为 `d` 的暂停。

4.9 指针

不像 Java 和 .NET，Go 语言为程序员提供了控制数据结构的指针的能力；但是，你不能进行指针运算。通过给予程序员基本内存布局，Go 语言允许你控制特定集合的数据结构、分配的数量以及内存访问模式，这些对构建运行良好的系统是非常重要的：指针对于性能的影响是不言而喻的，而如果你想要做的是系统编程、操作系统或者网络应用，指针更是不可或缺的一部分。

由于各种原因，指针对于使用面向对象编程的现代程序员来说可能显得有些陌生，不过我们将会在这一小节对此进行解释，并在未来的章节中展开深入讨论。

程序在内存中存储它的值，每个内存块（或字）有一个地址，通常用十六进制数表示，如：`0x6b0820` 或 `0xf84001d7f0`。

Go 语言的取地址符是 `&`，放到一个变量前使用就会返回相应变量的内存地址。

下面的代码片段（Example 4.9 `pointer.go`）可能输出 `An integer: 5, its location in memory: 0x6b0820`（这个值随着你每次运行程序而变化）。


```
var i1 = 5
fmt.Printf("An integer: %d, it's location in memory: %p\n", i1, &i1)
```

这个地址可以存储在一个叫做指针的特殊数据类型中，在本例中这是一个指向 `int` 的指针，即 `i1`：此处使用 `*int` 表示。如果我们想调用指针 `intP`，我们可以这样声明它：

```
var intP *int
```

然后使用 `intP = &i1` 是合法的，此时 `intP` 指向 `i1`。

（指针的格式化标识符为 `%p`）

`intP` 存储了 `i1` 的内存地址；它指向了 `i1` 的位置，它引用了变量 `i1`。

一个指针变量可以指向任何一个值的内存地址 它指向那个值的内存地址，在 32 位机器上占用 4 个字节，在 64 位机器上占用 8 个字节，并且与它所指向的值的大小无关。当然，可以声明指针指向任何类型的值来表明它的原始性或结构性；你可以在指针类型前面加上 `*` 号（前缀）来获取指针所指向的内容，这里的 `*` 号是一个类型更改器。使用一个指针引用一个值被称为间接引用。

当一个指针被定义后没有分配到任何变量时，它的值为 `nil`。

一个指针变量通常缩写为 `ptr`。

注意事项

在书写表达式类似 `var p *type` 时，切记在 `*` 号和指针名称间留有一个空格，因为 `var p*type` 是语法正确的，但是在更复杂的表达式中，它容易被误认为是一个乘法表达式！

符号 `*` 可以放在一个指针前，如 `*intP`，那么它将得到这个指针指向地址上所存储的值；这被称为反引用（或者内容或者间接引用）操作符；另一种说法是指针转移。

对于任何一个变量 `var`，如下表达式都是正确的：`var == *(&var)`。

现在，我们应当能理解 `pointer.go` 中的整个程序和他的输出：

Example 4.21 [pointer.go](#)：

```
package main
import "fmt"
func main() {
    var i1 = 5
    fmt.Printf("An integer: %d, its location in memory: %p\n", i1, &i1)
    var intP *int
    intP = &i1
    fmt.Printf("The value at memory location %p is %d\n", intP, *intP)
}
```

输出：

```
An integer: 5, its location in memory: 0x24f0820
The value at memory location 0x24f0820 is 5
```

我们可以用下图来表示内存使用的情况：

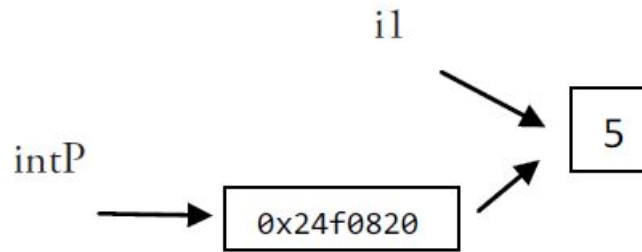


Fig 4.4: Pointers and memory usage

程序 `string_pointer.go` 为我们展示了指针对 `string` 的例子。

它展示了分配一个新的值给 `*p` 并且更改这个变量自己的值（这里是一个字符串）。

Example 4.22 [string_pointer.go](#)

```
package main
import "fmt"
func main() {
    s := "good bye"
    var p *string = &s
    *p = "ciao"
    fmt.Printf("Here is the pointer p: %p\n", p) // prints address
    fmt.Printf("Here is the string *p: %s\n", *p) // prints string
    fmt.Printf("Here is the string s: %s\n", s) // prints same string
}
```

输出：

```
Here is the pointer p: 0x2540820
Here is the string *p: ciao
Here is the string s: ciao
```

通过对 `*p` 赋另一个值来更改“对象”，这样 `s` 也会随之更改。

内存示意图如下：

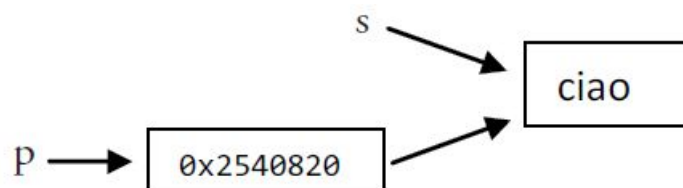


Fig 4.5: Pointers and memory usage, 2

注意事项

你不能得到一个文字或常量的地址，例如：

```
const i = 5
ptr := &i //error: cannot take the address of i
ptr2 := &10 //error: cannot take the address of 10
```

所以说，Go 语言和 C、C++ 以及 D 语言这些低级（系统）语言一样，都有指针的概念。但是对于经常导致 C 语言内存泄漏继而程序崩溃的指针运算（所谓的指针算法，如：`pointer+2`，移动指针指向字符串的字节数或数组的某个位置）是不被允许的。Go 语言中的指针保证了内存安全，更像是 Java、C# 和 VB.NET 中的引用。

因此 `c = *p++` 在 Go 语言的代码中是不合法的。

指针的一个高级应用是你可以传递一个变量的引用（如函数的参数），这样不会传递变量的拷贝。指针传递是很廉价的，只占用 4 个或 8 个字节。当程序在工作中需要占用大量的内存，或很多变量，或者两者都有，使用指针会减少内存占用和提高效率。被指向的变量也保存在内存中，直到没有任何指针指向它们，所以从它们被创建开始就具有相互独立的生命周期。

另一方面（虽然不太可能），由于一个指针导致的间接引用（一个进程执行了另一个地址），指针的过度频繁使用也会导致性能下降。

指针也可以指向另一个指针，并且可以进行任意深度的嵌套，导致你可以有多级的间接引用，但在大多数情况这会使你的代码结构不清晰。

如我们所见，在大多数情况下 Go 语言可以使程序员轻松创建指针，并且隐藏间接引用，如：自动反向引用。

对一个空指针的反向引用是不合法的，并且会使程序崩溃：

Example 4.23 [testcrash.go](https://golang.org/src/testcrash.go) :

```
package main
func main() {
    var p *int = nil
    *p = 0
}
// in Windows: stops only with: <exit code="-1073741819" msg="process crashed"/>
// runtime error: invalid memory address or nil pointer dereference
```

问题 4.2 列举 Go 语言中 * 号的所有用法。

5 控制结构

到目前为止，我们看到的都是 Go 程序都是从 main() 函数开始执行，然后按顺序执行该函数体中的代码。但我们经常会需要只有在满足一些特定情况时才执行某些代码，也就是说在代码里进行条件判断。针对这种需求，Go 提供了下面这些条件结构和分支结构：

```
if-else 结构
switch 结构
select 结构，用于 channel 的选择(第 14.4 节)
```

可以使用迭代或循环结构来重复执行一次或多次某段代码（任务）：

```
for (range) 结构
```

一些如 break 和 continue 这样的关键字可以用于中途改变循环的状态。

此外，你还可以使用 return 来结束某个函数的执行，或使用 goto 和标签来调整程序的执行位置。

Go 完全省略了 if、switch 和 for 结构中条件语句两侧的括号，相比 Java、C++ 和 C# 中减少了很多视觉混乱的因素，同时也使你的代码更加简洁。

5.1 if-else 结构

if 是用于测试某个条件（布尔型或逻辑型）的语句，如果该条件成立，则会执行 if 后由大括号括起来的代码块，否则就忽略该代码块继续执行后续的代码。

```
if condition {
    // do something
}
```

如果存在第二个分支，则可以在上面代码的基础上添加 else 关键字以及另一代码块，这个代码块中的代码只有在条件不满足时才会执行。if 和 else 后的两个代码块是相互独立的分支，只可能执行其中一个。

```
if condition {
    // do something
} else {
```

```
// do something
}
```

如果存在第三个分支，则可以使用下面这种三个独立分支的形式：

```
if condition1 {
    // do something
} else if condition2 {
    // do something else
} else {
    // catch-all or default
}
```

else-if分支的数量是没有限制的，但是为了代码的可读性，还是不要在 if 后面加入太多的 else-if 结构。如果你必须使用这种形式，则把尽可能先满足的条件放在前面。

即使当代码块之间只有一条语句时，大括号也不可被省略(尽管有些人并不赞成，但这还是符合了软件工程原则的主流做法)。

关键字 if 和 else 之后的左大括号 { 必须和关键字在同一行，如果你使用了 else-if 结构，则前段代码块的右大括号必须和 else-if 关键字在同一行。这两条规则都是被编译器强制规定的。

非法的Go代码：

```
if x{
}
else { // 无效的
}
```

要注意的是，在你使用 gofmt 格式化代码之后，每个分支内的代码都会缩进 4 个或 8 个空格，或者是 1 个 tab，并且右大括号与对应的 if 关键字垂直对齐。

在有些情况下，条件语句两侧的括号是可以被省略的；当条件比较复杂时，则可以使用括号让代码更易读。条件允许是符合条件，需使用 &&、|| 或 !，你可以使用括号来提升某个表达式的运算优先级，并提高代码的可读性。

一种可能用到条件语句的场景是测试变量的值，在不同的情况执行不同的语句，不过将在第 5.3 节讲到的 switch 结构会更适合这种情况。

Example 5.1 [boolens.go](#)

```
package main
import "fmt"
func main() {
    bool1 := true
    if bool1 {
        fmt.Printf("The value is true\n")
    } else {
        fmt.Printf("The value is false\n")
    }
}
```

输出：

```
The value is true
```

注意事项 这里不需要使用 `if bool1 == true` 来判断，因为 `bool1` 本身已经是一个布尔类型的值。

这种做法一般都用在测试 `true` 或者有利条件时，但你也可以使用取反 `!` 来判断值的相反结果，如：`if !bool1` 或者 `if !(condition)`。后者的括号大多数情况下是必须的，如这种情况：`if !(var1 == var2)`。

当 if 结构内有 `break`、`continue`、`goto` 或者 `return` 语句时，Go 代码的常见写法是省略 `else` 部分（另见第 5.2 节）。无论满足哪个条件都会返回 `x` 或者 `y` 时，一般使用以下写法：

```
if condition {
    return x
}
```

```
}  
return y
```

注意事项 不要同时在 if-else 结构的两个分支里都使用 return 语句，这将导致编译报错 “function ends without a return statement”（你可以认为这是一个编译器的 Bug 或者特性）。（译者注：该问题已经在 Go 1.1 中被修复或者说改进）

这里举一些有用的例子：

1. 判断一个字符串是否为空：if str == "" { ... } 或 if len(str) == 0 {...}。
2. 判断运行 Go 程序的操作系统类型，这可以通过常量 runtime.GOOS 来判断(第 2.2 节)。

```
if runtime.GOOS == "windows" {  
    ...  
} else { // Unix - like  
    ...  
}
```

这段代码一般被放在 init() 函数中执行。这儿还有一段示例来演示如何根据操作系统来决定输入结束的提示：

```
var prompt = "Enter a digit, e.g. 3 "+ "or %s to quit."  
  
func init() {  
    if runtime.GOOS == "windows" {  
        prompt = fmt.Sprintf(prompt, "Ctrl+Z, Enter")  
    } else { //Unix-like  
        prompt = fmt.Sprintf(prompt, "Ctrl+D")  
    }  
}
```

3. 函数 Abs() 用于返回一个整型数字的绝对值：

```
func Abs(x int) int {  
    if x < 0 {  
        return -x  
    }  
    return x  
}
```

4. isGreater 用于比较两个整型数字的大小：

```
func isGreater(x, y int) bool {  
    if x > y {  
        return true  
    }  
    return false  
}
```

在第四种情况中，if 可以包含一个初始化语句（如：给一个变量赋值）。这种写法具有固定的格式（在初始化语句后方必须加上分号）：

```
if initialization; condition {  
    // do something  
}
```

例如：

```
val := 10  
if val > max {  
    // do something  
}
```

你也可以这样写:

```
if val := 10; val > max {  
    // do something  
}
```

但要注意的是, 使用简短方式 `:=` 声明的变量的作用域只存在于 `if` 结构中 (在 `if` 结构的大括号之间, 如果使用 `if-else` 结构则在 `else` 代码块中变量也会存在)。如果变量在 `if` 结构之前就已经存在, 那么在 `if` 结构中, 该变量原来的值会被隐藏。最简单的解决方案就是不要在初始化语句中声明变量 (见 5.2 节的例 3 了解更多)。

Example 5.2 [ifelse.go](#)

```
package main  
  
import "fmt"  
  
func main() {  
    var first int = 10  
    var cond int  
  
    if first <= 0 {  
        fmt.Printf("first is less than or equal to 0\n")  
    } else if first > 0 && first < 5 {  
        fmt.Printf("first is between 0 and 5\n")  
    } else {  
        fmt.Printf("first is 5 or greater\n")  
    }  
    if cond = 5; cond > 10 {  
        fmt.Printf("cond is greater than 10\n")  
    } else {  
        fmt.Printf("cond is not greater than 10\n")  
    }  
}
```

输出:

```
first is 5 or greater cond is not greater than 10
```

下面的代码片段展示了如何通过在初始化语句中获取函数 `process()` 的返回值, 并在条件语句中作为判定条件来决定是否执行 `if` 结构中的代码:

```
if value := process(data); value > max {  
    ...  
if value := process(data); value > max {  
    ...  
}
```