目　录

总结和增强

前言

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

转自：https://github.com/DL911/the-way-to-go_ZH_CN

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下的生产环境。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 或其它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 下载书中的代码（**译者注：所有代码文件已经包括在 GitHub 仓库中**），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 Go1.4 版本下测试通过。

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，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）极大的影响了软件开发的高效性，这使得它再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、Ryanne 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！

Go 语言的起源，发展与普及

起源与发展

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）正式公布了 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 Gerrand 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 2017 年 1 月创下的第13名，流行程度 2.325%。

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 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 github.com/golang/go，Go 项目 Bug 追踪和功能预期详见 github.com/golang/go/issues。

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。

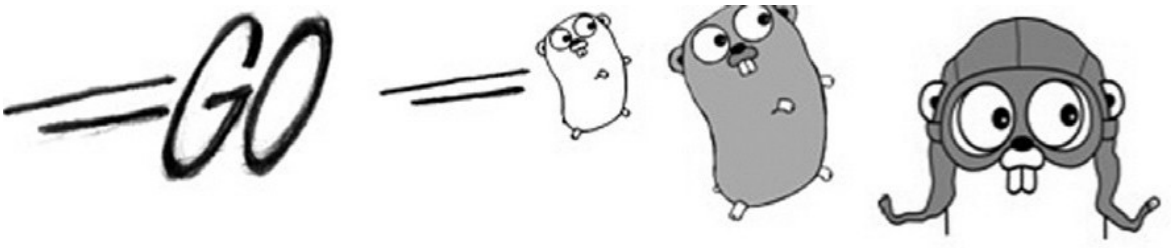


图1.2 Go 语言 Logo

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。

@golang 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：www.linkedin.com/groups?gid=2524765&trk=myg_ugrp_ovr。

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

Go 语言相关资源的搜索引擎页面：gowalker.org

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。

语言的主要特性与发展的环境和影响因素

1.2 语言的主要特性与发展的环境和影响因素

1.2.1 影响 Go 语言发展的早期编程语言

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：

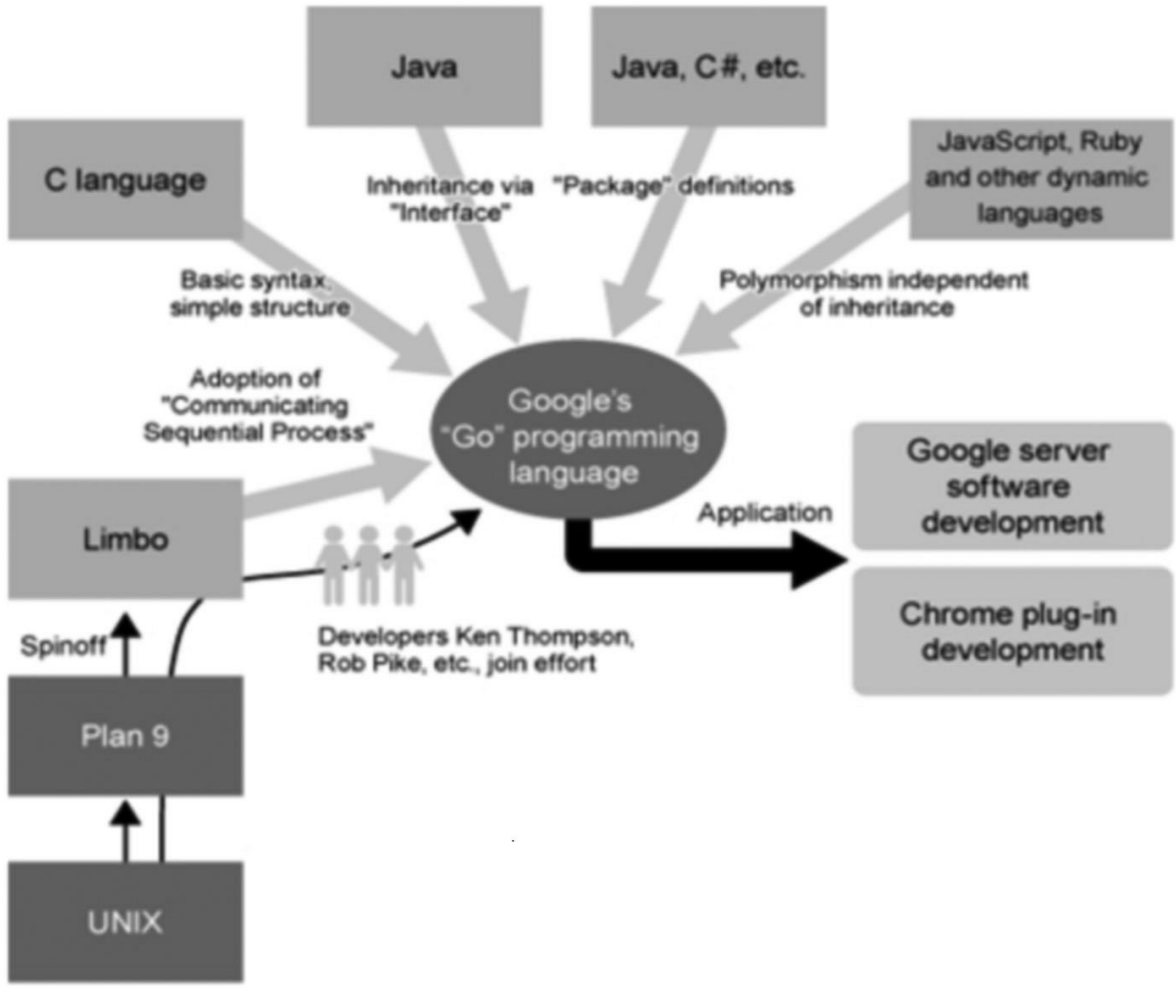


图 1.3 其它编程语言对 Go 语言的影响

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。
- 相比计算机性能的提升，软件开发领域不被认为发展得足够快或者比硬件发展得更加成功（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乐趣，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 语言的另一个目标是对于网络通信、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 语言还能够在运行时进行反射相关的操作。

使用 `go install` 能够很轻松地对第三方包进行部署。

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 是 Go 语言的语法标准，你也可以在 `src/cmd/internal/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 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，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 使用 Go 的组织 页面进行查看。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），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 Go 1.4 的 SDK 1.9.18，于 2015 年 2 月 18 日发布。当前 Go 语言的稳定版本是 Go 1.4.2。

1.2.7 关于特性缺失

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 Go 常见问题 页面查看。

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 语言的优雅的解决方案。

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

安装与运行环境

平台与架构

2.1 平台与架构

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形式工作。

你可以获取以下平台上的 Go 1.4 源码和二进制文件：

- 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 Plan 9 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。（ **译者注：Go从1.5版本开始已经实现自举。Go语言的编译器和链接器都是Go语言编写的**）

   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。

No of bits	*Processor name*	*Compiler*	*Linker*
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 Plan 9 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 [`$GOROOT/src/cmd`](https://github.com/golang/go/tree/master/src/cmd) 下找到编译器和链接器的源代码。Go 语言本身是由 C 语言开发的，而不是 Go 语言（Go 1.5 开始自举）。词法分析程序是 GNU bison，语法分析程序是名为 [`$GOROOT/src/cmd/gc/go.y`](https://github.com/golang/go/blob/master/src%2Fcmd%2Finternal%2Fgc%2Fgo.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，你可以在 [`$GOROOT/src/make.bash`](https://github.com/golang/go/blob/master/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。

Go 环境变量

2.2 Go 环境变量

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 之前就已经设置好他们。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` 设置本地机器的操作系统名称和编译体系结构，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 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：Go Porting Efforts。

在 Linux 上安装 Go

2.3 在 Linux 上安装 Go

如果你能够自己下载并编译 Go 的源代码的话,对你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：Download the Go distribution。

我们接下来也会带你一步步地完成安装过程。

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
   ```

   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=$PATH:$GOROOT/bin
   ```

   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。

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. 获取 Go 源代码

   从 官方页面 或 国内镜像 下载 Go 的源码包到你的计算机上，然后将解压后的目录 `go` 通过命令移动到 `$GOROOT` 所指向的位置。

   ```
   wget https://storage.googleapis.com/golang/go<VERSION>.src.tar.gz
   tar -zxvf go<VERSION>.src.tar.gz
   sudo mv go $GOROOT
   ```

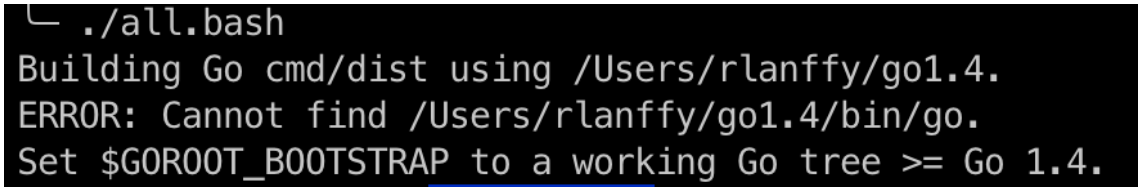
4. 构建 Go

在终端使用以下指令来进行编译工作。

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

编译注意事项

编译时如果出现如下报错：



可能是因为 `$GOROOT_BOOTSTRAP` 变量没有设置。这个目录在安装 Go 1.5 版本及之后的版本时需要设置。

由于在 1.4 版本后，Go 编译器实现了自举，即通过 1.4 版本来编译安装之后版本的编译器。如果不设置该环境变量的话，会产生这样一个错误 `Set $GOROOT_BOOTSTRAP to a working Go tree >= Go 1.4.` 。

设置 `$GOROOT_BOOTSTRAP` 变量：

```
export GOROOT_BOOTSTRAP=$HOME/go1.4
```

设置完成后，下载 1.4 版本的源码到该目录：

```
git clone https://github.com/golang/go.git $HOME/go1.4
git checkout -b release-branch.go1.4 origin/release-branch.go1.4
```

进入 1.4 的文件夹后，进行编译：

```
cd $HOME/go1.4/src
./make.bash
```

1.4 编译安装好之后，进入 `$GOROOT` 文件夹，真正开始编译安装 Go：

```
cd $HOME/go/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：

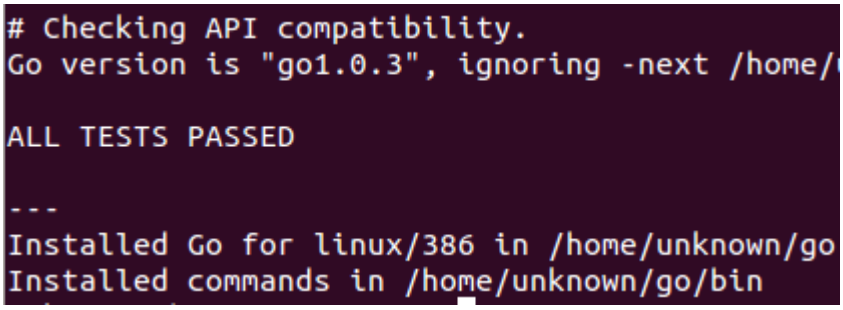


图 2.3 完成编译后在终端打印的信息

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。

5. 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 `hello_world1.go` 。

示例 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` 。

6. 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。

示例 2.2 version.go

```
package main

import (
    "fmt"
    "runtime"
)

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

这段代码将会输出 `go1.4.2` 或类似字符串。

7. 更新版本

你可以在 发布历史 页面查看到最新的稳定版。

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
```

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。

在 Mac OS X 上安装 Go

2.4 在 Mac OS X 上安装 Go

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://codedr-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 下载页面 页面下载到 Mac 系统下的一键安装包或源代码自行编译。

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。

在 Windows 上安装 Go

2.5 在 Windows 上安装 Go

你可以在 下载页面 页面下载到 Windows 系统下的一键安装包。

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，一般推荐安装 TDM-GCC。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。

安装目录清单

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.md, 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
- `/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
- `/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和标准库的包的完整源代码（Go 是一门开源语言）
- `/src/cmd` ：包含 Go 和 C 的编译器和命令行脚本

Go 运行时（runtime）

2.7 Go 运行时（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（Go 1.5 开始自举），并且是每个 Go 包的最顶级包。你可以在目录 `$GOROOT/src/runtime` 中找到相关内容。

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。但总的来说，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。

Go 解释器

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（read-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://github.com/sbinet/igo。

编辑器、集成开发环境与其它工具

3.0 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 编辑器和 IDE 扩展 页面来获取 Go 开发工具的最新信息。

Go 开发环境的基本要求

3.1 Go 开发环境的基本要求

这里有一个可以用来开发 Go 的集成开发环境，你期待有以下哪些特性，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。

编辑器和集成开发环境

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、Xcode 6、KD Kate、TextWrangler、BBEdit、McEdit、TextMate、TextPad、JEdit、SciTE、Nano、Notepad++、Geany、SlickEdit、Visual Studio Code、IntelliJ IDEA 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。

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。

IntelliJ Idea Plugin 是一个 IntelliJ IDEA 的插件，具有很好的操作体验和代码补全功能。

LiteIDE 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（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 IntelliJ Idea Plugin 也是不错的选择。

代码补全 一般都是通过内置 GoCode 实现的（如：LieteIDE、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（务必事先配置好 Go 环境变量）。

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 LiteIDE

这款 IDE 的当前最新版本号为 X27，你可以从 GitHub 页面获取详情。

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。

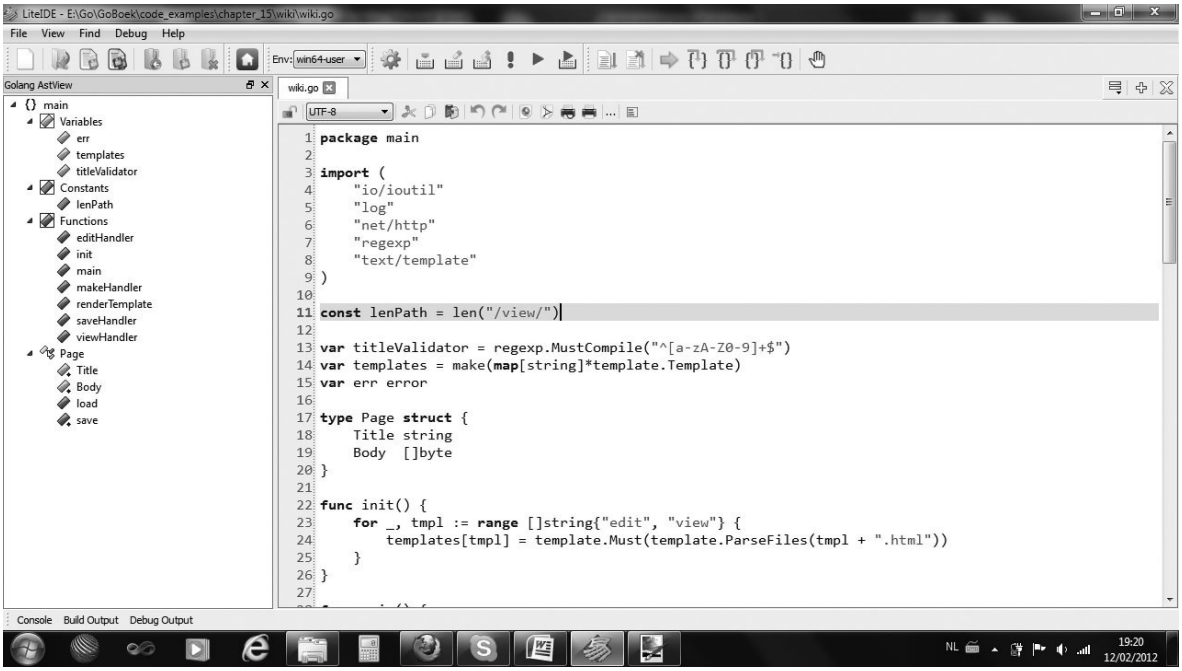


图 3.1 LiteIDE 代码编辑界面和抽象语法树视图

3.2.2 GoClipse

该款插件的当前最新版本号为 0.9.1，你可以从 GitHub 页面获取详情。

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。

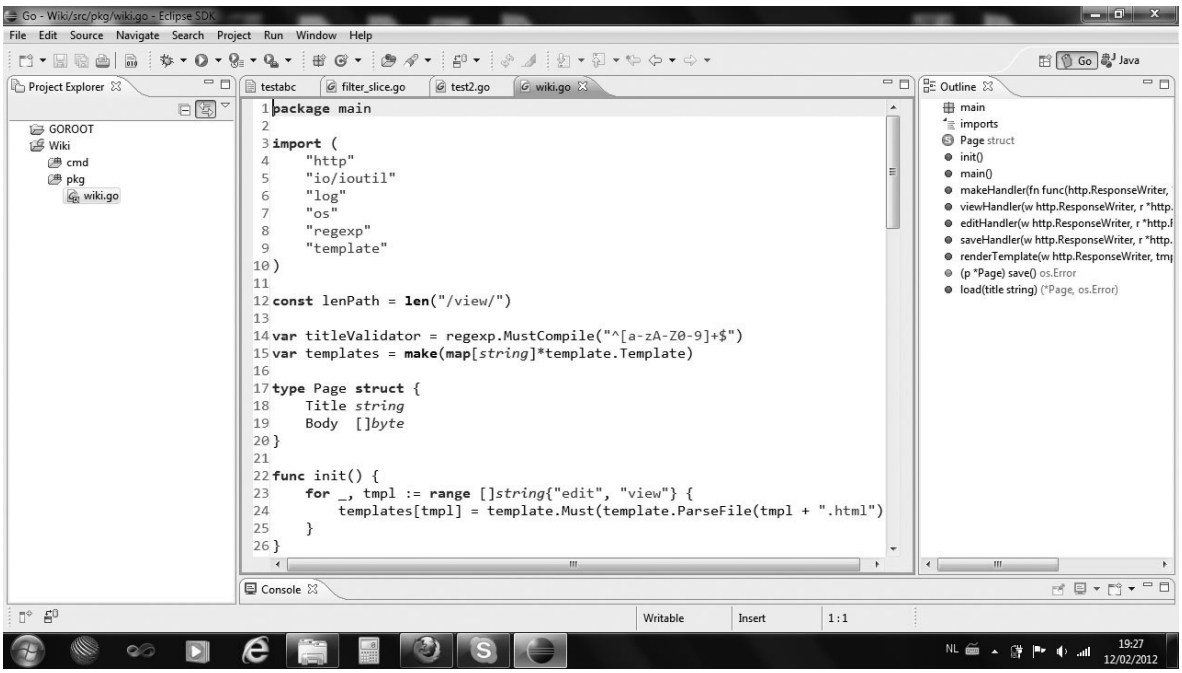


图 3.2 GoClipse 代码编辑界面、抽象语法树视图和项目管理

调试器

3.3 调试器

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）。

构建并运行 Go 程序

3.4 构建并运行 Go 程序

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

格式化代码

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 mapl` 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：了解切片（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/。

生成代码文档

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` 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/（在线版的第三方包 `godoc` 可以使用 Go Walker）。

其它工具

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，如果像 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` 之所以能够正常工作是因为 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）。

Go 性能说明

3.8 Go 性能说明

根据 Go 开发团队和基本的算法测试，Go 语言与 C 语言的性能差距大概在 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，并比这两门语言占用的内存降低了 70% 。在很多情况下这种比较是没有意义的，而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 用于生产环境才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擅长和薄弱的方面。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。还要注意的是，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，是Java的五分之一左右。

与其它语言进行交互

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()` 。

示例 3.2 c1.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 时，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：

示例 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，示例 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。

基本结构和基本数据类型

文件名、关键字与标识符

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母（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、ɵε12。

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。虽然这不是必须的，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	c
copy	false	float32	float64	imag	int	int8	i
int32	int64	iota	len	make	new	nil	p

print	println	real	recover	string	true	uint	u

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。

Go 程序的基本结构和要素

4.2 Go 程序的基本结构和要素

示例 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 `packl` ，编译后产生的对象文件将会是 `packl.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 和 os），但是你也可以创建自己的包（第 9 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，一个包即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"
)
```

它甚至还可以更短的形式，但使用 gofmt 后将会被强制换行：

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

当你导入多个包时，最好按照字母顺序排列包名，这样做更加清晰易读。

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。

译者注：以相对路径在GOPATH下导入包会产生报错信息

报错信息：local import "./XXX" in non-local package

引用：Go programs cannot use relative import paths within a work space.

注解：在GOPATH外可以以相对路径的形式执行go build（go install 不可以）

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，能够且只能够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：

示例 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（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 函数既没有参数，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 时就会出现错误提示：

```
`build-error: syntax error: unexpected semicolon or newline before {`
```

（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，因此才会引发像上面这样的错误

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。

下面这一行调用了 `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` ：

```
fmt.Println("hello, world")
```

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 即 `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

示例 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

变量（或常量）包含数据，这些数据可以有不同的数据类型，简称类型。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
```

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。

然后我们可以使用下面的方式声明变量：

```
var a IZ = 5
```

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 float64
    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。

示例 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)
```

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，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。

常量

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`

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 = "eat", 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，并且没有赋值的常量默认会应用上一行的赋值表达式：

```
// 赋值一个常量时，之后没赋值的常量都会应用上一行的赋值表达式
const (
    a = iota  // a = 0
    b         // b = 1
    c         // c = 2
    d = 5     // d = 5
    e         // e = 5
)

// 赋值两个常量，iota 只会增长一次，而不会因为使用了两次就增长两次
const (
    Apple, Banana = iota + 1, iota + 2 // Apple=1 Banana=2
    Cherimoya, Durian                  // Cherimoya=2 Durian=3
    Elderberry, Fig                    // Elderberry=3, Fig=4

)

// 使用 iota 结合 位运算 表示资源状态的使用案例
const (
    Open = 1 << iota  // 0001
    Close             // 0010
    Pending           // 0100
)

const (
    _           = iota             // 使用 _ 忽略不需要的 iota
    KB = 1 << (10 * iota)          // 1 << (10*1)
    MB                             // 1 << (10*2)
    GB                             // 1 << (10*3)
    TB                             // 1 << (10*4)
    PB                             // 1 << (10*5)
    EB                             // 1 << (10*6)
    ZB                             // 1 << (10*7)
```

```
    YB                              // 1 << (10*8)
)
```

（ **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（ **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。

变量

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 Go 语言的声明语法 页面找到有关于这个话题的更多讨论）。

而在 Go 中，则可以很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` 和 `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，当变量a和变量b之间类型相同时，才能进行如 `a = b` 的赋值。

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通过 `runtime` 包在运行时获取所在的操作系统类型，以及如何通过 `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。

示例 4.5 goos.go

```
package main

import (
    "fmt"
  "runtime"
    "os"
)

func main() {
    var goos string = runtime.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 和 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：

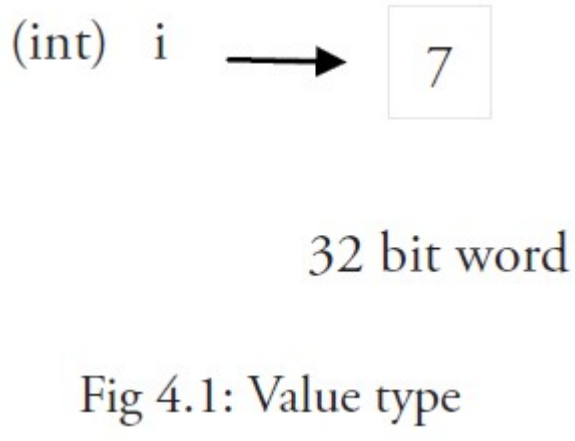


Fig 4.1: Value type

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：

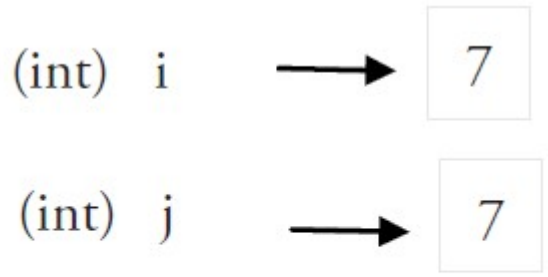


Fig 4.2: Assignment of value types

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。

Fig 4.3: Reference types and assignment

这个内存地址被称之为指针（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)
```

在示例 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，具体例子可以参见示例 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 都已经被声明，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。

每个源文件都只能包含一个 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。

示例 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。

示例 4.7 user_init.go 中导入了包 trans（需要init.go目录为./trans/init.go）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 := "O"
    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 = "O"
    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 := "O"
    print(a)
    f2()
}

func f2() {
    print(a)
}
```

基本类型和运算符

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 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。

与运算符：`&&`

```
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（详情参见 二的补码 页面）。

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。（Go语言中只有 float32 和 float64）没有double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）：

示例 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式转换来避免这个问题（第 4.2 节）。

示例 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)
}
```

输出：

```
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），`%0nd` 用于规定输出长度为n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：

```
^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 <<= 2` 或者 `b ^= 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）

`/` 对于整数运算而言，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。

示例 4.10 random.go 演示了如何生成 10 个非负随机数：

```
package main
import (
    "fmt"
    "math/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("%2.2f / ", 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
/ 65.77 / 65.89 / 16.85 / 75.56 / 46.90 / 55.24 / 55.95 / 25.58 / 70.61 /
```

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 `rand.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        ^ !
```

```
6        * / % << >> & &^
5        + - | ^
4        == != < <= >= >
3        <-
2        &&
1        ||
```

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。

示例 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 8 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。

示例 4.12 char.go

```
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 类型相关的函数。

（ **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** ）

字符串

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 中的字符串里面的字符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++不一样，Go 中的字符串是根据长度限定，而非特殊字符 `\0` 。

`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 - 1]`
- 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来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.Sprint(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）。

strings 和 strconv 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
```

示例 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义字符 `\` 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
```

如果需要查询非 ASCII 编码的字符在父字符串中的位置，建议使用以下函数来对字符进行定位：

```
strings.IndexRune(s string, r rune) int
```

```
注: 原文为 "If ch is a non-ASCII character use strings.IndexRune(s string, ch int) int."
该方法在最新版本的 Go 中定义为 func IndexRune(s string, r rune) int
实际使用中的第二个参数 rune 可以是 rune 或 int, 例如 strings.IndexRune("chicken", 99) 或 strings.IndexRune("chicken", rune('k'))
```

示例 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
```

示例 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
```

示例 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
```

示例 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(sl []string, sep string) string
```

示例 4.18 strings_splitjoin.go

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str := "The quick brown fox jumps over the lazy dog"
    sl := strings.Fields(str)
    fmt.Printf("Splitted in slice: %v\n", sl)
    for _, val := range sl {
        fmt.Printf("%s - ", val)
    }
    fmt.Println()
    str2 := "GO1|The ABC of Go|25"
    sl2 := strings.Split(str2, "|")
    fmt.Printf("Splitted in slice: %v\n", sl2)
    for _, val := range sl2 {
        fmt.Printf("%s - ", val)
    }
    fmt.Println()
    str3 := strings.Join(sl2,";")
    fmt.Printf("sl2 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 -
sl2 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（ **译者注：国内用户可访问 该页面** ）。

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：

示例 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：

```
64 位系统:
The size of ints is: 64
32 位系统:
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（ **译者注：国内用户可访问 该页面** ）。

时间和日期

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（ **译者注：国内用户可访问 该页面** ）。

示例 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(time.Duration(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
    // The time must be 2006-01-02 15:04:05
    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（d Duration）` 可以实现对某个进程（实质上是 goroutine）时长为 d 的暂停。

指针

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（示例 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 的全部内容及其输出：

示例 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：

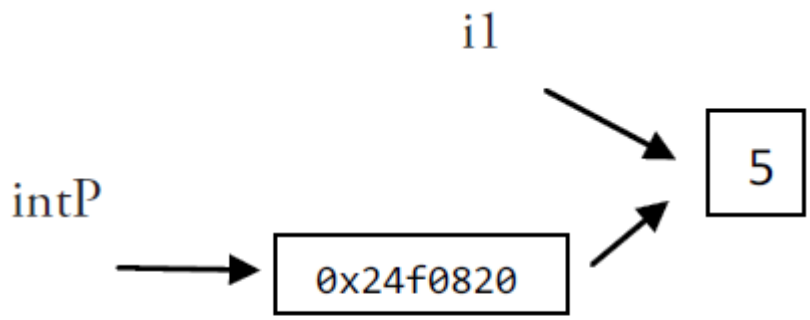


Fig 4.4: Pointers and memory usage

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）。

示例 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：

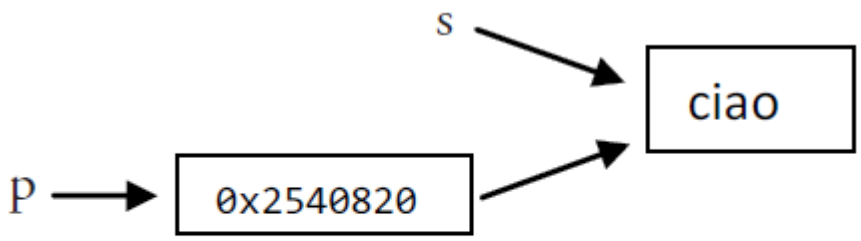


Fig 4.5: Pointers and memory usage, 2

注意事项

你不能获取字面量或常量的地址，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：

示例 4.23 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。

控制结构

5.0 控制结构

到目前为止，我们看到的 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。

if-else 结构

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。

示例 5.1 booleans.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 == "" { ... }`
   - `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)。

示例 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 {
    ...
}
```

测试多返回值函数的错误

5.2 测试多返回值函数的错误

Go 语言的函数经常使用两个返回值来表示执行是否成功：返回某个值以及 true 表示成功；返回零值（或 nil）和 false 表示失败（第 4.4 节）。当不使用 true 或 false 的时候，也可以使用一个 error 类型的变量来代替作为第二个返回值：成功执行的话，error 的值为 nil，否则就会包含相应的错误信息（Go 语言中的错误类型为 error: `var err error`，我们将会在第 13 章进行更多地讨论）。这样一来，就很明显需要用一个 if 语句来测试执行结果；由于其符号的原因，这样的形式又称之为 comma,ok 模式（pattern）。

在第 4.7 节的程序 `string_conversion.go` 中，函数 `strconv.Atoi` 的作用是将一个字符串转换为一个整数。之前我们忽略了相关的错误检查：

```
anInt, _ = strconv.Atoi(origStr)
```

如果 origStr 不能被转换为整数，anInt 的值会变成 0 而 `_` 无视了错误，程序会继续运行。

这样做是非常不好的：程序应该在最接近的位置检查所有相关的错误，至少需要暗示用户有错误发生并对函数进行返回，甚至中断程序。

我们在第二个版本中对代码进行了改进：

示例 1：

示例 5.3 string_conversion2.go

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var orig string = "ABC"
    // var an int
    var newS string
    // var err error

    fmt.Printf("The size of ints is: %d\n", strconv.IntSize)
    // anInt, err = strconv.Atoi(origStr)
    an, err := strconv.Atoi(orig)
    if err != nil {
        fmt.Printf("orig %s is not an integer - exiting with error\n", orig)
        return
    }
    fmt.Printf("The integer is %d\n", an)
    an = an + 5
    newS = strconv.Itoa(an)
    fmt.Printf("The new string is: %s\n", newS)
}
```

这是测试 err 变量是否包含一个真正的错误（`if err != nil`）的习惯用法。如果确实存在错误，则会打印相应的错误信息然后通过 return 提前结束函数的执行。我们还可以使用携带返回值的 return 形式，例如 `return err`。这样一来，函

数的调用者就可以检查函数执行过程中是否存在错误了。

习惯用法

```
value, err := pack1.Function1(param1)
if err != nil {
    fmt.Printf("An error occured in pack1.Function1 with parameter %v", param1)
    return err
}
// 未发生错误，继续执行：
```

由于本例的函数调用者属于 main 函数，所以程序会直接停止运行。

如果我们想要在错误发生的同时终止程序的运行，我们可以使用 `os` 包的 `Exit` 函数：

习惯用法

```
if err != nil {
    fmt.Printf("Program stopping with error %v", err)
    os.Exit(1)
}
```

（此处的退出代码 1 可以使用外部脚本获取到）

有时候，你会发现这种习惯用法被连续重复地使用在某段代码中。

当没有错误发生时，代码继续运行就是唯一要做的事情，所以 if 语句块后面不需要使用 else 分支。

示例 2：我们尝试通过 `os.Open` 方法打开一个名为 `name` 的只读文件：

```
f, err := os.Open(name)
if err != nil {
    return err
}
doSomething(f) // 当没有错误发生时，文件对象被传入到某个函数中
doSomething
```

练习 5.1 尝试改写 string_conversion2.go 中的代码，要求使用 `:=` 方法来对 err 进行赋值，哪些地方可以被修改？

示例 3：可以将错误的获取放置在 if 语句的初始化部分：

习惯用法

```
if err := file.Chmod(0664); err != nil {
    fmt.Println(err)
    return err
}
```

示例 4：或者将 ok-pattern 的获取放置在 if 语句的初始化部分，然后进行判断：

习惯用法

```
if value, ok := readData(); ok {
...
}
```

注意事项

如果您像下面一样，没有为多返回值的函数准备足够的变量来存放结果：

```
func mySqrt(f float64) (v float64, ok bool) {
    if f < 0 { return } // error case
    return math.Sqrt(f),true
}

func main() {
    t := mySqrt(25.0)
    fmt.Println(t)
}
```

您会得到一个编译错误： `multiple-value mySqrt() in single-value context` 。

正确的做法是：

```
t, ok := mySqrt(25.0)
if ok { fmt.Println(t) }
```

注意事项 2

当您将字符串转换为整数时，且确定转换一定能够成功时，可以将 `Atoi` 函数进行一层忽略错误的封装：

```
func atoi (s string) (n int) {
    n, _ = strconv.Atoi(s)
    return
}
```

实际上， `fmt` 包（第 4.4.3 节）最简单的打印函数也有 2 个返回值：

```
count, err := fmt.Println(x) // number of bytes printed, nil or 0, error
```

当打印到控制台时，可以将该函数返回的错误忽略；但当输出到文件流、网络流等具有不确定因素的输出对象时，应该始终检查是否有错误发生（另见练习 6.1b）。

switch 结构

5.3 switch 结构

相比较 C 和 Java 等其它语言而言，Go 语言中的 switch 结构使用上更加灵活。它接受任意形式的表达式：

```
switch var1 {
    case val1:
        ...
    case val2:
        ...
    default:
        ...
}
```

变量 var1 可以是任何类型，而 val1 和 val2 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同的类型；或者最终结果为相同类型的表达式。前花括号 `{` 必须和 switch 关键字在同一行。

您可以同时测试多个可能符合条件的值，使用逗号分割它们，例如： `case val1, val2, val3` 。

每一个 `case` 分支都是唯一的，从上至下逐一测试，直到匹配为止。（ Go 语言使用快速的查找算法来测试 switch 条件与 case 分支的匹配情况，直到算法匹配到某个 case 或者进入 default 条件为止。）

一旦成功地匹配到某个分支，在执行完相应代码后就会退出整个 switch 代码块，也就是说您不需要特别使用 `break` 语句来表示结束。

因此，程序也不会自动地去执行下一个分支的代码。如果在执行完每个分支的代码后，还希望继续执行后续分支的代码，可以使用 `fallthrough` 关键字来达到目的。

因此：

```
switch i {
    case 0: // 空分支，只有当 i == 0 时才会进入分支
    case 1:
        f() // 当 i == 0 时函数不会被调用
}
```

并且：

```
switch i {
    case 0: fallthrough
    case 1:
        f() // 当 i == 0 时函数也会被调用
}
```

在 `case ...:` 语句之后，您不需要使用花括号将多行语句括起来，但您可以在分支中进行任意形式的编码。当代码块只有一行时，可以直接放置在 `case` 语句之后。

您同样可以使用 `return` 语句来提前结束代码块的执行。当您在 switch 语句块中使用 `return` 语句，并且您的函数是有返回值的，您还需要在 switch 之后添加相应的 `return` 语句以确保函数始终会返回。

可选的 `default` 分支可以出现在任何顺序，但最好将它放在最后。它的作用类似与 `if-else` 语句中的 `else` ，表示不符合任何已给出条件时，执行相关语句。

示例 5.4 switch1.go：

```
package main

import "fmt"

func main() {
    var num1 int = 100

    switch num1 {
    case 98, 99:
        fmt.Println("It's equal to 98")
    case 100:
        fmt.Println("It's equal to 100")
    default:
        fmt.Println("It's not equal to 98 or 100")
    }
}
```

输出：

```
It's equal to 100
```

在第 12.1 节，我们会使用 switch 语句判断从键盘输入的字符（详见第 12.2 节的 switch.go）。switch 语句的第二种形式是不提供任何被判断的值（实际上默认为判断是否为 true），然后在每个 case 分支中进行测试不同的条件。当任一分支的测试结果为 true 时，该分支的代码会被执行。这看起来非常像链式的 `if-else` 语句，但是在测试条件非常多的情况下，提供了可读性更好的书写方式。

```
switch {
    case condition1:
        ...
    case condition2:
        ...
    default:
        ...
}
```

例如：

```
switch {
    case i < 0:
        f1()
    case i == 0:
        f2()
    case i > 0:
        f3()
}
```

任何支持进行相等判断的类型都可以作为测试表达式的条件，包括 int、string、指针等。

示例 5.4 switch2.go：

```
package main

import "fmt"
```

```
func main() {
    var num1 int = 7

    switch {
        case num1 < 0:
            fmt.Println("Number is negative")
        case num1 > 0 && num1 < 10:
            fmt.Println("Number is between 0 and 10")
        default:
            fmt.Println("Number is 10 or greater")
    }
}
```

输出：

```
Number is between 0 and 10
```

switch 语句的第三种形式是包含一个初始化语句：

```
switch initialization {
    case val1:
        ...
    case val2:
        ...
    default:
        ...
}
```

这种形式可以非常优雅地进行条件判断：

```
switch result := calculate(); {
    case result < 0:
        ...
    case result > 0:
        ...
    default:
        // 0
}
```

在下面这个代码片段中，变量 a 和 b 被平行初始化，然后作为判断条件：

```
switch a, b := x[i], y[j]; {
    case a < b: t = -1
    case a == b: t = 0
    case a > b: t = 1
}
```

switch 语句还可以被用于 type-switch（详见第 11.4 节）来判断某个 interface 变量中实际存储的变量类型。

问题 5.1：

请说出下面代码片段输出的结果：

```
    k := 6
    switch k {
```

```
    case 4:
        fmt.Println("was <= 4")
        fallthrough
    case 5:
        fmt.Println("was <= 5")
        fallthrough
    case 6:
        fmt.Println("was <= 6")
        fallthrough
    case 7:
        fmt.Println("was <= 7")
        fallthrough
    case 8:
        fmt.Println("was <= 8")
        fallthrough
    default:
        fmt.Println("default case")
    }
```

练习 5.2： season.go：

写一个 Season 函数，要求接受一个代表月份的数字，然后返回所代表月份所在季节的名称（不用考虑月份的日期）。

for 结构

5.4 for 结构

如果想要重复执行某些语句，Go 语言中您只有 for 结构可以使用。不要小看它，这个 for 结构比其它语言中的更为灵活。

注意事项 其它许多语言中也没有发现和 do while 完全对等的 for 结构，可能是因为这种需求并不是那么强烈。

5.4.1 基于计数器的迭代

文件 for1.go 中演示了最简单的基于计数器的迭代，基本形式为：

```
for 初始化语句; 条件语句; 修饰语句 {}
```

示例 5.6 for1.go：

```
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        fmt.Printf("This is the %d iteration\n", i)
    }
}
```

输出：

```
This is the 0 iteration
This is the 1 iteration
This is the 2 iteration
This is the 3 iteration
This is the 4 iteration
```

由花括号括起来的代码块会被重复执行已知次数，该次数是根据计数器（此例为 i）决定的。循环开始前，会执行且仅会执行一次初始化语句 `i := 0;` ；这比在循环之前声明更为简短。紧接着的是条件语句 `i < 5;` ，在每次循环开始前都会进行判断，一旦判断结果为 false，则退出循环体。最后一部分为修饰语句 `i++` ，一般用于增加或减少计数器。

这三部分组成的循环的头部，它们之间使用分号 `;` 相隔，但并不需要括号 `()` 将它们括起来。例如： `for (i = 0; i < 10; i++) { }` ，这是无效的代码！

同样的，左花括号 `{` 必须和 for 语句在同一行，计数器的生命周期在遇到右花括号 `}` 时便终止。一般习惯使用 i、j、z 或 ix 等较短的名称命名计数器。

特别注意，永远不要在循环体内修改计数器，这在任何语言中都是非常差的实践！

您还可以在循环中同时使用多个计数器：

```
for i, j := 0, N; i < j; i, j = i+1, j-1 {}
```

这得益于 Go 语言具有的平行赋值的特性（可以查看第 7 章 string_reverse.go 中反转数组的示例）。

您可以将两个 for 循环嵌套起来：

```
for i:=0; i<5; i++ {
    for j:=0; j<10; j++ {
        println(j)
    }
}
```

如果您使用 for 循环迭代一个 Unicode 编码的字符串，会发生什么？

示例 5.7 for_string.go：

```
package main

import "fmt"

func main() {
    str := "Go is a beautiful language!"
    fmt.Printf("The length of str is: %d\n", len(str))
    for ix :=0; ix < len(str); ix++ {
        fmt.Printf("Character on position %d is: %c \n", ix, str[ix])
    }
    str2 := "日本語"
    fmt.Printf("The length of str2 is: %d\n", len(str2))
    for ix :=0; ix < len(str2); ix++ {
        fmt.Printf("Character on position %d is: %c \n", ix, str2[ix])
    }
}
```

输出：

```
The length of str is: 27
Character on position 0 is: G
Character on position 1 is: o
Character on position 2 is:
Character on position 3 is: i
Character on position 4 is: s
Character on position 5 is:
Character on position 6 is: a
Character on position 7 is:
Character on position 8 is: b
Character on position 9 is: e
Character on position 10 is: a
Character on position 11 is: u
Character on position 12 is: t
Character on position 13 is: i
Character on position 14 is: f
Character on position 15 is: u
Character on position 16 is: l
Character on position 17 is:
Character on position 18 is: l
Character on position 19 is: a
Character on position 20 is: n
Character on position 21 is: g
Character on position 22 is: u
Character on position 23 is: a
Character on position 24 is: g
Character on position 25 is: e
```

```
Character on position 26 is: !
The length of str2 is: 9
Character on position 0 is: æ
Character on position 1 is:
Character on position 2 is: ¥
Character on position 3 is: æ
Character on position 4 is:
Character on position 5 is: ¬
Character on position 6 is: è
Character on position 7 is: ª
Character on position 8 is:
```

如果我们打印 str 和 str2 的长度，会分别得到 27 和 9。

由此我们可以发现，ASCII 编码的字符占用 1 个字节，既每个索引都指向不同的字符，而非 ASCII 编码的字符（占有 2 到 4 个字节）不能单纯地使用索引来判断是否为同一个字符。我们会在第 5.4.4 节解决这个问题。

练习题

练习 5.4 for_loop.go

1. 使用 for 结构创建一个简单的循环。要求循环 15 次然后使用 fmt 包来打印计数器的值。
2. 使用 goto 语句重写循环，要求不能使用 for 关键字。

练习 5.5 for_character.go

创建一个程序，要求能够打印类似下面的结果（尾行达 25 个字符为止）：

```
G
GG
GGG
GGGG
GGGGG
GGGGGG
```

1. 使用 2 层嵌套 for 循环。
2. 仅用 1 层 for 循环以及字符串连接。

练习 5.6 bitwise_complement.go

使用按位补码从 0 到 10，使用位表达式 `%b` 来格式化输出。

练习 5.7 Fizz-Buzz 问题：fizzbuzz.go

写一个从 1 打印到 100 的程序，但是每当遇到 3 的倍数时，不打印相应的数字，但打印一次 “Fizz”。遇到 5 的倍数时，打印 `Buzz` 而不是相应的数字。对于同时为 3 和 5 的倍数的数，打印 `FizzBuzz` （提示：使用 switch 语句）。

练习 5.8 rectangle_stars.go

使用 `*` 符号打印宽为 20，高为 10 的矩形。

5.4.2 基于条件判断的迭代

for 结构的第二种形式是没有头部的条件判断迭代（类似其它语言中的 while 循环），基本形式为：`for 条件语句 {}` 。

您也可以认为这是没有初始化语句和修饰语句的 for 结构，因此 `;;` 便是多余的了。

Listing 5.8 for2.go：

```
package main

import "fmt"

func main() {
    var i int = 5

    for i >= 0 {
        i = i - 1
        fmt.Printf("The variable i is now: %d\n", i)
    }
}
```

输出：

```
The variable i is now: 4
The variable i is now: 3
The variable i is now: 2
The variable i is now: 1
The variable i is now: 0
The variable i is now: -1
```

5.4.3 无限循环

条件语句是可以被省略的，如 `i:=0; ; i++` 或 `for { }` 或 `for ;; { }`（`;;` 会在使用 gofmt 时被移除）：这些循环的本质就是无限循环。最后一个形式也可以被改写为 `for true { }`，但一般情况下都会直接写 `for { }`。

如果 for 循环的头部没有条件语句，那么就会认为条件永远为 true，因此循环体内必须有相关的条件判断以确保会在某个时刻退出循环。

想要直接退出循环体，可以使用 break 语句（第 5.5 节）或 return 语句直接返回（第 6.1 节）。

但这两者之间有所区别，break 只是退出当前的循环体，而 return 语句提前对函数进行返回，不会执行后续的代码。

无限循环的经典应用是服务器，用于不断等待和接受新的请求。

```
for t, err = p.Token(); err == nil; t, err = p.Token() {
    ...
}
```

5.4.4 for-range 结构

这是 Go 特有的一种的迭代结构，您会发现它在许多情况下都非常有用。它可以迭代任何一个集合（包括数组和 map，详见第 7 和 8 章）。语法上很类似其它语言中 foreach 语句，但您依旧可以获得每次迭代所对应的索引。一般形式为：`for ix, val := range coll { }`。

要注意的是，`val` 始终为集合中对应索引的值拷贝，因此它一般只具有只读性质，对它所做的任何修改都不会影响到集合中原有的值（**译者注：如果 `val` 为指针，则会产生指针的拷贝，依旧可以修改集合中的原值**）。一个字符串是 Unicode 编码的字符（或称之为 `rune`）集合，因此您也可以用它迭代字符串：

```
for pos, char := range str {
...
```

```
}
```

每个 rune 字符和索引在 for-range 循环中是一一对应的。它能够自动根据 UTF-8 规则识别 Unicode 编码的字符。

示例 5.9 range_string.go：

```
package main

import "fmt"

func main() {
    str := "Go is a beautiful language!"
    fmt.Printf("The length of str is: %d\n", len(str))
    for pos, char := range str {
        fmt.Printf("Character on position %d is: %c \n", pos, char)
    }
    fmt.Println()
    str2 := "Chinese: 日本語"
    fmt.Printf("The length of str2 is: %d\n", len(str2))
    for pos, char := range str2 {
        fmt.Printf("character %c starts at byte position %d\n", char, pos)
    }
    fmt.Println()
    fmt.Println("index int(rune) rune    char bytes")
    for index, rune := range str2 {
        fmt.Printf("%-2d      %d      %U '%c' % X\n", index, rune, rune, rune, []byte(string(rune)))
    }
}
```

输出：

```
The length of str is: 27
Character on position 0 is: G
Character on position 1 is: o
Character on position 2 is:
Character on position 3 is: i
Character on position 4 is: s
Character on position 5 is:
Character on position 6 is: a
Character on position 7 is:
Character on position 8 is: b
Character on position 9 is: e
Character on position 10 is: a
Character on position 11 is: u
Character on position 12 is: t
Character on position 13 is: i
Character on position 14 is: f
Character on position 15 is: u
Character on position 16 is: l
Character on position 17 is:
Character on position 18 is: l
Character on position 19 is: a
Character on position 20 is: n
Character on position 21 is: g
Character on position 22 is: u
Character on position 23 is: a
Character on position 24 is: g
Character on position 25 is: e
Character on position 26 is: !
```

```
The length of str2 is: 18
character C starts at byte position 0
character h starts at byte position 1
character i starts at byte position 2
character n starts at byte position 3
character e starts at byte position 4
character s starts at byte position 5
character e starts at byte position 6
character : starts at byte position 7
character   starts at byte position 8
character 日 starts at byte position 9
character 本 starts at byte position 12
character 語 starts at byte position 15

index int(rune) rune    char bytes
0       67      U+0043 'C' 43
1       104      U+0068 'h' 68
2       105      U+0069 'i' 69
3       110      U+006E 'n' 6E
4       101      U+0065 'e' 65
5       115      U+0073 's' 73
6       101      U+0065 'e' 65
7       58      U+003A ':' 3A
8       32      U+0020 ' ' 20
9       26085      U+65E5 '日' E6 97 A5
12      26412      U+672C '本' E6 9C AC
15      35486      U+8A9E '語' E8 AA 9E
```

请将输出结果和 Listing 5.7（for_string.go）进行对比。

我们可以看到，常用英文字符使用 1 个字节表示，而汉字（**译者注：严格来说，“Chinese: 日本語”的Chinese应该是Japanese**）使用 3 个字符表示。

练习 5.9 以下程序的输出结果是什么？

```
for i := 0; i < 5; i++ {
    var v int
    fmt.Printf("%d ", v)
    v = 5
}
```

问题 5.2： 请描述以下 for 循环的输出结果：

1.

```
for i := 0; ; i++ {
    fmt.Println("Value of i is now:", i)
}
```

2.

```
for i := 0; i < 3; {
    fmt.Println("Value of i:", i)
}
```

3.

```
s := ""
for ; s != "aaaaa"; {
    fmt.Println("Value of s:", s)
    s = s + "a"
}
```

4.

```
for i, j, s := 0, 5, "a"; i < 3 && j < 100 && s != "aaaaa"; i, j,
    s = i+1, j+1, s + "a" {
    fmt.Println("Value of i, j, s:", i, j, s)
}
```

Break 与 continue

5.5 Break 与 continue

您可以使用 break 语句重写 for2.go 的代码：

示例 5.10 for3.go：

```
for {
    i = i - 1
    fmt.Printf("The variable i is now: %d\n", i)
    if i < 0 {
        break
    }
}
```

因此每次迭代都会对条件进行检查（i < 0），以此判断是否需要停止循环。如果退出条件满足，则使用 break 语句退出循环。

一个 break 的作用范围为该语句出现后的最内部的结构，它可以被用于任何形式的 for 循环（计数器、条件判断等）。但在 switch 或 select 语句中（详见第 13 章），break 语句的作用结果是跳过整个代码块，执行后续的代码。

下面的示例中包含了嵌套的循环体（for4.go），break 只会退出最内层的循环：

示例 5.11 for4.go：

```
package main

func main() {
    for i:=0; i<3; i++ {
        for j:=0; j<10; j++ {
            if j>5 {
                break
            }
            print(j)
        }
        print("  ")
    }
}
```

输出：

```
012345 012345 012345
```

关键字 continue 忽略剩余的循环体而直接进入下一次循环的过程，但不是无条件执行下一次循环，执行之前依旧需要满足循环的判断条件。

示例 5.12 for5.go：

```
package main

func main() {
    for i := 0; i < 10; i++ {
```

```
        if i == 5 {
            continue
        }
        print(i)
        print(" ")
    }
}
```

输出：

```
0 1 2 3 4 6 7 8 9
```

显然，5 被跳过了。

另外，关键字 continue 只能被用于 for 循环中。

标签与 goto

5.6 标签与 goto

for、switch 或 select 语句都可以配合标签（label）形式的标识符使用，即某一行第一个以冒号（ `:` ）结尾的单词（gofmt 会将后续代码自动移至下一行）。

示例 5.13 for6.go：

（标签的名称是大小写敏感的，为了提升可读性，一般建议使用全部大写字母）

```
package main

import "fmt"

func main() {

LABEL1:
    for i := 0; i <= 5; i++ {
        for j := 0; j <= 5; j++ {
            if j == 4 {
                continue LABEL1
            }
            fmt.Printf("i is: %d, and j is: %d\n", i, j)
        }
    }

}
```

本例中，continue 语句指向 LABEL1，当执行到该语句的时候，就会跳转到 LABEL1 标签的位置。

您可以看到当 j==4 和 j==5 的时候，没有任何输出：标签的作用对象为外部循环，因此 i 会直接变成下一个循环的值，而此时 j 的值就被重设为 0，即它的初始值。如果将 continue 改为 break，则不会只退出内层循环，而是直接退出外层循环了。另外，还可以使用 goto 语句和标签配合使用来模拟循环。

示例 5.14 goto.go：

```
package main

func main() {
    i:=0
    HERE:
        print(i)
        i++
        if i==5 {
            return
        }
        goto HERE
}
```

上面的代码会输出 `01234` 。

使用逆向的 goto 会很快导致意大利面条式的代码，所以不应当使用而选择更好的替代方案。

特别注意 使用标签和 goto 语句是不被鼓励的：它们会很快导致非常糟糕的程序设计，而且总有更加可读的替代方案来实现相同的需求。

一个建议使用 goto 语句的示例会在第 15.1 章的 simple_tcp_server.go 中出现：示例中在发生读取错误时，使用 goto 来跳出无限读取循环并关闭相应的客户端链接。

定义但未使用标签会导致编译错误： `label … defined and not used` 。

如果您必须使用 goto，应当只使用正序的标签（标签位于 goto 语句之后），但注意标签和 goto 语句之间不能出现定义新变量的语句，否则会导致编译失败。

示例 5.15 goto2.go：

```
// compile error goto2.go:8: goto TARGET jumps over declaration of b at goto2.go:8
package main

import "fmt"

func main() {
        a := 1
        goto TARGET // compile error
        b := 9
    TARGET:
        b += a
        fmt.Printf("a is %v *** b is %v", a, b)
}
```

问题 5.3 请描述下面 for 循环的输出：

1.

```
i := 0
for { //since there are no checks, this is an infinite loop
    if i >= 3 { break }
    //break out of this for loop when this condition is met
    fmt.Println("Value of i is:", i)
    i++
}
fmt.Println("A statement just after for loop.")
```

2.

```
for i := 0; i<7 ; i++ {
    if i%2 == 0 { continue }
    fmt.Println("Odd:", i)
}
```

函数（function）

6.0 函数（function）

函数是 Go 里面的基本代码块：Go 函数的功能非常强大，以至于被认为拥有函数式编程语言的多种特性。在这一章，我们将对第 4.2.2 节 所简要描述的函数进行详细的讲解。

介绍

6.1 介绍

每一个程序都包含很多的函数：函数是基本的代码块。

Go是编译型语言，所以函数编写的顺序是无关紧要的；鉴于可读性的需求，最好把 `main()` 函数写在文件的前面，其他函数按照一定逻辑顺序进行编写（例如函数被调用的顺序）。

编写多个函数的主要目的是将一个需要很多行代码的复杂问题分解为一系列简单的任务（那就是函数）来解决。而且，同一个任务（函数）可以被调用多次，有助于代码重用。

（事实上，好的程序是非常注意DRY原则的，即不要重复你自己（Don't Repeat Yourself），意思是执行特定任务的代码只能在程序里面出现一次。）

当函数执行到代码块最后一行（ `}` 之前）或者 `return` 语句的时候会退出，其中 `return` 语句可以带有零个或多个参数；这些参数将作为返回值（参考 第 6.2 节）供调用者使用。简单的 `return` 语句也可以用来结束 for 死循环，或者结束一个协程（goroutine）。

Go 里面有三种类型的函数：

- 普通的带有名字的函数
- 匿名函数或者lambda函数（参考 第 6.8 节）
- 方法（Methods，参考 第 10.6 节）

除了main()、init()函数外，其它所有类型的函数都可以有参数与返回值。函数参数、返回值以及它们的类型被统称为函数签名。

作为提醒，提前介绍一个语法：

这样是不正确的 Go 代码：

```
func g()
{
}
```

它必须是这样的：

```
func g() {
}
```

函数被调用的基本格式如下：

```
pack1.Function(arg1, arg2, …, argn)
```

`Function` 是 `pack1` 包里面的一个函数，括号里的是被调用函数的实参（argument）：这些值被传递给被调用函数的*形参*（parameter，参考 第 6.2 节）。函数被调用的时候，这些实参将被复制（简单而言）然后传递给被调用函数。函数一般是在其他函数里面被调用的，这个其他函数被称为调用函数（calling function）。函数能多次调用其他函数，这些被调用函数按顺序（简单而言）执行，理论上，函数调用其他函数的次数是无穷的（直到函数调用栈被耗尽）。

一个简单的函数调用其他函数的例子：

示例 6.1 greeting.go

```
package main

func main() {
    println("In main before calling greeting")
    greeting()
    println("In main after calling greeting")
}

func greeting() {
    println("In greeting: Hi!!!!!")
}
```

代码输出：

```
In main before calling greeting
In greeting: Hi!!!!!
In main after calling greeting
```

函数可以将其他函数调用作为它的参数，只要这个被调用函数的返回值个数、返回值类型和返回值的顺序与调用函数所需求的实参是一致的，例如：

假设 f1 需要 3 个参数 `f1(a, b, c int)` ，同时 f2 返回 3 个参数 `f2(a, b int) (int, int, int)` ，就可以这样调用 f1： `f1(f2(a, b))` 。

函数重载（function overloading）指的是可以编写多个同名函数，只要它们拥有不同的形参与/或者不同的返回值，在 Go 里面函数重载是不被允许的。这将导致一个编译错误：

```
funcName redeclared in this book, previous declaration at lineno
```

Go 语言不支持这项特性的主要原因是函数重载需要进行多余的类型匹配影响性能；没有重载意味着只是一个简单的函数调度。所以你需要给不同的函数使用不同的名字，我们通常会根据函数的特征对函数进行命名（参考 第 11.12.5 节）。

如果需要申明一个在外部定义的函数，你只需要给出函数名与函数签名，不需要给出函数体：

```
func flushICache(begin, end uintptr) // implemented externally
```

函数也可以以申明的方式被使用，作为一个函数类型，就像：

```
type binOp func(int, int) int
```

在这里，不需要函数体 `{}` 。

函数是一等值（first-class value）：它们可以赋值给变量，就像 `add := binOp` 一样。

这个变量知道自己指向的函数的签名，所以给它赋一个具有不同签名的函数值是不可能的。

函数值（functions value）之间可以相互比较：如果它们引用的是相同的函数或者都是 nil 的话，则认为它们是相同的函数。函数不能在其它函数里面声明（不能嵌套），不过我们可以通过使用匿名函数（参考 第 6.8 节）来破除这个限制。

目前 Go 没有泛型（generic）的概念，也就是说它不支持那种支持多种类型的函数。不过在大部分情况下可以通过接口（interface），特别是空接口与类型选择（type switch，参考 第 11.12 节）与/或者通过使用反射（reflection，参考 第 6.8 节）来实现相似的功能。使用这些技术将导致代码更为复杂、性能更为低下，所以在非常注意性能的的场合，最好是为每一个类型单独创建一个函数，而且代码可读性更强。

函数参数与返回值

6.2 函数参数与返回值

函数能够接收参数供自己使用，也可以返回零个或多个值（我们通常把返回多个值称为返回一组值）。相比与 C、C++、Java 和 C#，多值返回是 Go 的一大特性，为我们判断一个函数是否正常执行（参考 第 5.2 节）提供了方便。

我们通过 `return` 关键字返回一组值。事实上，任何一个有返回值（单个或多个）的函数都必须以 `return` 或 `panic`（参考 第 13 章）结尾。

在函数块里面，`return` 之后的语句都不会执行。如果一个函数需要返回值，那么这个函数里面的每一个代码分支（code-path）都要有 `return` 语句。

问题 6.1：下面的函数将不会被编译，为什么呢？大家可以试着纠正过来。

```
func (st *Stack) Pop() int {
    v := 0
    for ix := len(st) - 1; ix >= 0; ix-- {
        if v = st[ix]; v != 0 {
            st[ix] = 0
            return v
        }
    }
}
```

函数定义时，它的形参一般是有名字的，不过我们也可以定义没有形参名的函数，只有相应的形参类型，就像这样：`func f(int, int, float64)`。

没有参数的函数通常被称为 **niladic** 函数（niladic function），就像 `main.main()`。

6.2.1 按值传递（call by value） 按引用传递（call by reference）

Go 默认使用按值传递来传递参数，也就是传递参数的副本。函数接收参数副本之后，在使用变量的过程中可能对副本的值进行更改，但不会影响到原来的变量，比如 `Function(arg1)`。

如果你希望函数可以直接修改参数的值，而不是对参数的副本进行操作，你需要将参数的地址（变量名前面添加&符号，比如&variable）传递给函数，这就是按引用传递，比如 `Function(&arg1)`，此时传递给函数的是一个指针。如果传递给函数的是一个指针，指针的值（一个地址）会被复制，但指针的值所指向的地址上的值不会被复制；我们可以通过这个指针的值来修改这个值所指向的地址上的值。（**译者注：指针也是变量类型，有自己的地址和值，通常指针的值指向一个变量的地址。所以，按引用传递也是按值传递。**）

几乎在任何情况下，传递指针（一个32位或者64位的值）的消耗都比传递副本来得少。

在函数调用时，像切片（slice）、字典（map）、接口（interface）、通道（channel）这样的引用类型都是默认使用引用传递（即使没有显式的指出指针）。

有些函数只是完成一个任务，并没有返回值。我们仅仅是利用了这种函数的副作用，就像输出文本到终端，发送一个邮件或者是记录一个错误等。

但是绝大部分的函数还是带有返回值的。

如下，simple_function.go 里的 `MultiPly3Nums` 函数带有三个形参，分别是 `a`、`b`、`c`，还有一个 `int` 类型的返回值（被注释的代码具有和未注释部分同样的功能，只是多引入了一个本地变量）：

示例 6.2 simple_function.go

```
package main

import "fmt"

func main() {
    fmt.Printf("Multiply 2 * 5 * 6 = %d\n", MultiPly3Nums(2, 5, 6))
    // var i1 int = MultiPly3Nums(2, 5, 6)
    // fmt.Printf("MultiPly 2 * 5 * 6 = %d\n", i1)
}

func MultiPly3Nums(a int, b int, c int) int {
    // var product int = a * b * c
    // return product
    return a * b * c
}
```

输出显示：

```
Multiply 2 * 5 * 6 = 60
```

如果一个函数需要返回四到五个值，我们可以传递一个切片给函数（如果返回值具有相同类型）或者是传递一个结构体（如果返回值具有不同的类型）。因为传递一个指针允许直接修改变量的值，消耗也更少。

问题 6.2：

如下的两个函数调用有什么不同：

```
(A) func DoSomething(a *A) {
        b = a
    }

(B) func DoSomething(a A) {
        b = &a
    }
```

6.2.2 命名的返回值（named return variables）

如下，multiple_return.go 里的函数带有一个 `int` 参数，返回两个 `int` 值；其中一个函数的返回值在函数调用时就已经被赋予了一个初始零值。

`getX2AndX3` 与 `getX2AndX3_2` 两个函数演示了如何使用非命名返回值与命名返回值的特性。当需要返回多个非命名返回值时，需要使用 `()` 把它们括起来，比如 `(int, int)`。

命名返回值作为结果形参（result parameters）被初始化为相应类型的零值，当需要返回的时候，我们只需要一条简单的不带参数的return语句。需要注意的是，即使只有一个命名返回值，也需要使用 `()` 括起来（参考 第 6.6 节的 fibonacci.go 函数）。

示例 6.3 multiple_return.go

```
package main

import "fmt"

var num int = 10
```

```
var numx2, numx3 int

func main() {
    numx2, numx3 = getX2AndX3(num)
    PrintValues()
    numx2, numx3 = getX2AndX3_2(num)
    PrintValues()
}

func PrintValues() {
    fmt.Printf("num = %d, 2x num = %d, 3x num = %d\n", num, numx2, numx3)
}

func getX2AndX3(input int) (int, int) {
    return 2 * input, 3 * input
}

func getX2AndX3_2(input int) (x2 int, x3 int) {
    x2 = 2 * input
    x3 = 3 * input
    // return x2, x3
    return
}
```

输出结果：

```
num = 10, 2x num = 20, 3x num = 30
num = 10, 2x num = 20, 3x num = 30
```

警告：

- return 或 return var 都是可以的。
- 不过 `return var = expression`（表达式） 会引发一个编译错误：`syntax error: unexpected =, expecting semicolon or newline or }`。

即使函数使用了命名返回值，你依旧可以无视它而返回明确的值。

任何一个非命名返回值（使用非命名返回值是很糟的编程习惯）在 `return` 语句里面都要明确指出包含返回值的变量或是一个可计算的值（就像上面警告所指出的那样）。

尽量使用命名返回值：会使代码更清晰、更简短，同时更加容易读懂。

练习 6.1 mult_returnval.go

编写一个函数，接收两个整数，然后返回它们的和、积与差。编写两个版本，一个是非命名返回值，一个是命名返回值。

练习 6.2 error_returnval.go

编写一个名字为 MySqrt 的函数，计算一个 float64 类型浮点数的平方根，如果参数是一个负数的话将返回一个错误。编写两个版本，一个是非命名返回值，一个是命名返回值。

6.2.3 空白符（blank identifier）

空白符用来匹配一些不需要的值，然后丢弃掉，下面的 blank_identifier.go 就是很好的例子。

`ThreeValues` 是拥有三个返回值的不需要任何参数的函数，在下面的例子中，我们将第一个与第三个返回值赋给了 `i1` 与 `f1`。第二个返回值赋给了空白符 `_`，然后自动丢弃掉。

示例 6.4 blank_identifier.go

```
package main

import "fmt"

func main() {
    var i1 int
    var f1 float32
    i1, _, f1 = ThreeValues()
    fmt.Printf("The int: %d, the float: %f \n", i1, f1)
}

func ThreeValues() (int, int, float32) {
    return 5, 6, 7.5
}
```

输出结果：

```
The int: 5, the float: 7.500000
```

另外一个示例，函数接收两个参数，比较它们的大小，然后按小-大的顺序返回这两个数，示例代码为minmax.go。

示例 6.5 minmax.go

```
package main

import "fmt"

func main() {
    var min, max int
    min, max = MinMax(78, 65)
    fmt.Printf("Minmium is: %d, Maximum is: %d\n", min, max)
}

func MinMax(a int, b int) (min int, max int) {
    if a < b {
        min = a
        max = b
    } else { // a = b or a < b
        min = b
        max = a
    }
    return
}
```

输出结果：

```
Minimum is: 65, Maximum is 78
```

6.2.4 改变外部变量（outside variable）

传递指针给函数不但可以节省内存（因为没有复制变量的值），而且赋予了函数直接修改外部变量的能力，所以被修改的变量不再需要使用 `return` 返回。如下的例子，`reply` 是一个指向 `int` 变量的指针，通过这个指针，我们在函数内修改了这个 `int` 变量的数值。

示例 6.6 side_effect.go

```
package main

import (
    "fmt"
)

// this function changes reply:
func Multiply(a, b int, reply *int) {
    *reply = a * b
}

func main() {
    n := 0
    reply := &n
    Multiply(10, 5, reply)
    fmt.Println("Multiply:", *reply) // Multiply: 50
}
```

这仅仅是个指导性的例子，当需要在函数内改变一个占用内存比较大的变量时，性能优势就更加明显了。然而，如果不小心使用的话，传递一个指针很容易引发一些不确定的事，所以，我们要十分小心那些可以改变外部变量的函数，在必要时，需要添加注释以便其他人能够更加清楚的知道函数里面到底发生了什么。

传递变长参数

6.3 传递变长参数

如果函数的最后一个参数是采用 `...type` 的形式，那么这个函数就可以处理一个变长的参数，这个长度可以为 0，这样的函数称为变参函数。

```
func myFunc(a, b, arg ...int) {}
```

这个函数接受一个类似某个类型的 slice 的参数（详见第 7 章），该参数可以通过第 5.4.4 节中提到的 for 循环结构迭代。

示例函数和调用：

```
func Greeting(prefix string, who ...string)
Greeting("hello:", "Joe", "Anna", "Eileen")
```

在 Greeting 函数中，变量 `who` 的值为 `[]string{"Joe", "Anna", "Eileen"}` 。

如果参数被存储在一个 slice 类型的变量 `slice` 中，则可以通过 `slice...` 的形式来传递参数，调用变参函数。

示例 6.7 varnumpar.go

```
package main

import "fmt"

func main() {
    x := min(1, 3, 2, 0)
    fmt.Printf("The minimum is: %d\n", x)
    slice := []int{7,9,3,5,1}
    x = min(slice...)
    fmt.Printf("The minimum in the slice is: %d", x)
}

func min(s ...int) int {
    if len(s)==0 {
        return 0
    }
    min := s[0]
    for _, v := range s {
        if v < min {
            min = v
        }
    }
    return min
}
```

输出：

```
The minimum is: 0
The minimum in the slice is: 1
```

练习 6.3 varargs.go

写一个函数，该函数接受一个变长参数并对每个元素进行换行打印。

一个接受变长参数的函数可以将这个参数作为其它函数的参数进行传递：

```
func F1(s ...string) {
    F2(s...)
    F3(s)
}

func F2(s ...string) { }
func F3(s []string) { }
```

变长参数可以作为对应类型的 slice 进行二次传递。

但是如果变长参数的类型并不是都相同的呢？使用 5 个参数来进行传递并不是很明智的选择，有 2 种方案可以解决这个问题：

1. 使用结构（详见第 10 章）：

   定义一个结构类型，假设它叫 `Options` ，用以存储所有可能的参数：

   ```
   type Options struct {
       par1 type1,
       par2 type2,
       ...
   }
   ```

   函数 F1 可以使用正常的参数 a 和 b，以及一个没有任何初始化的 Options 结构： `F1(a, b, Options {})` 。如果需要对选项进行初始化，则可以使用 `F1(a, b, Options {par1:val1, par2:val2})` 。

2. 使用空接口：

   如果一个变长参数的类型没有被指定，则可以使用默认的空接口 `interface{}` ，这样就可以接受任何类型的参数（详见第 11.9 节）。该方案不仅可以用于长度未知的参数，还可以用于任何不确定类型的参数。一般而言我们会使用一个 for-range 循环以及 switch 结构对每个参数的类型进行判断：

   ```
   func typecheck(..,..,values ··· interface{}) {
       for _, value := range values {
           switch v := value.(type) {
               case int: ···
               case float: ···
               case string: ···
               case bool: ···
               default: ···
           }
       }
   }
   ```

defer 和追踪

6.4 defer 和追踪

关键字 defer 允许我们推迟到函数返回之前（或任意位置执行 `return` 语句之后）一刻才执行某个语句或函数（为什么要在返回之后才执行这些语句？因为 `return` 语句同样可以包含一些操作，而不是单纯地返回某个值）。

关键字 defer 的用法类似于面向对象编程语言 Java 和 C# 的 `finally` 语句块，它一般用于释放某些已分配的资源。

示例 6.8 defer.go：

```
package main
import "fmt"

func main() {
    function1()
}

func function1() {
    fmt.Printf("In function1 at the top\n")
    defer function2()
    fmt.Printf("In function1 at the bottom!\n")
}

func function2() {
    fmt.Printf("Function2: Deferred until the end of the calling function!")
}
```

输出：

```
In Function1 at the top
In Function1 at the bottom!
Function2: Deferred until the end of the calling function!
```

请将 defer 关键字去掉并对比输出结果。

使用 defer 的语句同样可以接受参数，下面这个例子就会在执行 defer 语句时打印 `0`：

```
func a() {
    i := 0
    defer fmt.Println(i)
    i++
    return
}
```

当有多个 defer 行为被注册时，它们会以逆序执行（类似栈，即后进先出）：

```
func f() {
    for i := 0; i < 5; i++ {
        defer fmt.Printf("%d ", i)
    }
}
```

上面的代码将会输出： `4 3 2 1 0` 。

关键字 defer 允许我们进行一些函数执行完成后的收尾工作，例如：

1. 关闭文件流 （详见 第 12.2 节）

```
// open a file
defer file.Close()
```

2. 解锁一个加锁的资源 （详见 第 9.3 节）

```
mu.Lock()
defer mu.Unlock()
```

3. 打印最终报告

```
printHeader()
defer printFooter()
```

4. 关闭数据库链接

```
// open a database connection
defer disconnectFromDB()
```

合理使用 defer 语句能够使得代码更加简洁。

以下代码模拟了上面描述的第 4 种情况：

```
package main

import "fmt"

func main() {
    doDBOperations()
}

func connectToDB() {
    fmt.Println("ok, connected to db")
}

func disconnectFromDB() {
    fmt.Println("ok, disconnected from db")
}

func doDBOperations() {
    connectToDB()
    fmt.Println("Defering the database disconnect.")
    defer disconnectFromDB() //function called here with defer
    fmt.Println("Doing some DB operations ...")
    fmt.Println("Oops! some crash or network error ...")
    fmt.Println("Returning from function here!")
    return //terminate the program
    // deferred function executed here just before actually returning, even if
    // there is a return or abnormal termination before
}
```

输出：

```
ok, connected to db
Defering the database disconnect.
Doing some DB operations ...
Oops! some crash or network error ...
Returning from function here!
ok, disconnected from db
```

使用 defer 语句实现代码追踪

一个基础但十分实用的实现代码执行追踪的方案就是在进入和离开某个函数打印相关的消息，即可以提炼为下面两个函数：

```
func trace(s string) { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }
```

以下代码展示了何时调用这两个函数：

示例 6.10 defer_tracing.go:

```
package main

import "fmt"

func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

func a() {
    trace("a")
    defer untrace("a")
    fmt.Println("in a")
}

func b() {
    trace("b")
    defer untrace("b")
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

输出：

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

上面的代码还可以修改为更加简便的版本（示例 6.11 defer_tracing2.go）：

```
package main

import "fmt"

func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

使用 defer 语句来记录函数的参数与返回值

下面的代码展示了另一种在调试时使用 defer 语句的手法（示例 6.12 defer_logvalues.go）：

```
package main

import (
    "io"
    "log"
)

func func1(s string) (n int, err error) {
    defer func() {
        log.Printf("func1(%q) = %d, %v", s, n, err)
    }()
    return 7, io.EOF
}

func main() {
    func1("Go")
}
```

输出：

```
Output: 2011/10/04 10:46:11 func1("Go") = 7, EOF
```

内置函数

6.5 内置函数

Go 语言拥有一些不需要进行导入操作就可以使用的内置函数。它们有时可以针对不同的类型进行操作，例如：len、cap 和 append，或必须用于系统级的操作，例如：panic。因此，它们需要直接获得编译器的支持。

以下是一个简单的列表，我们会在后面的章节中对它们进行逐个深入的讲解。

名称	说明
close	用于管道通信
len、cap	len 用于返回某个类型的长度或数量（字符串、数组、切片、map 和管道）；cap 是容量的意思，用于返回某个类型的最大容量（只能用于切片和 map）
new、make	new 和 make 均是用于分配内存：new 用于值类型和用户定义的类型，如自定义结构，make 用于内置引用类型（切片、map 和管道）。它们的用法就像是函数，但是将类型作为参数：new(type)、make(type)。new(T) 分配类型 T 的零值并返回其地址，也就是指向类型 T 的指针（详见第 10.1 节）。它也可以被用于基本类型：`v := new(int)`。make(T) 返回类型 T 的初始化之后的值，因此它比 new 进行更多的工作（详见第 7.2.3/4 节、第 8.1.1 节和第 14.2.1 节）**new() 是一个函数，不要忘记它的括号**
copy、append	用于复制和连接切片
panic、recover	两者均用于错误处理机制
print、println	底层打印函数（详见第 4.2 节），在部署环境中建议使用 fmt 包
complex、real imag	用于创建和操作复数（详见第 4.5.2.2 节）

递归函数

6.6 递归函数

当一个函数在其函数体内调用自身，则称之为递归。最经典的例子便是计算斐波那契数列，即前两个数为1，从第三个数开始每个数均为前两个数之和。

数列如下所示：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, …
```

下面的程序可用于生成该数列（示例 6.13 fibonacci.go）：

```
package main

import "fmt"

func main() {
    result := 0
    for i := 0; i <= 10; i++ {
        result = fibonacci(i)
        fmt.Printf("fibonacci(%d) is: %d\n", i, result)
    }
}

func fibonacci(n int) (res int) {
    if n <= 1 {
        res = 1
    } else {
        res = fibonacci(n-1) + fibonacci(n-2)
    }
    return
}
```

输出：

```
fibonacci(0) is: 1
fibonacci(1) is: 1
fibonacci(2) is: 2
fibonacci(3) is: 3
fibonacci(4) is: 5
fibonacci(5) is: 8
fibonacci(6) is: 13
fibonacci(7) is: 21
fibonacci(8) is: 34
fibonacci(9) is: 55
fibonacci(10) is: 89
```

许多问题都可以使用优雅的递归来解决，比如说著名的快速排序算法。

在使用递归函数时经常会遇到的一个重要问题就是栈溢出：一般出现在大量的递归调用导致的程序栈内存分配耗尽。这个问题可以通过一个名为懒惰求值的技术解决，在 Go 语言中，我们可以使用管道（channel）和 goroutine（详见第 14.8 节）来实现。练习 14.12 也会通过这个方案来优化斐波那契数列的生成问题。

Go 语言中也可以使用相互调用的递归函数：多个函数之间相互调用形成闭环。因为 Go 语言编译器的特殊性，这些函数的声明顺序可以是任意的。下面这个简单的例子展示了函数 odd 和 even 之间的相互调用（示例 6.14 mut_recurs.go）：

```
package main

import (
    "fmt"
)

func main() {
    fmt.Printf("%d is even: is %t\n", 16, even(16)) // 16 is even: is true
    fmt.Printf("%d is odd: is %t\n", 17, odd(17))
    // 17 is odd: is true
    fmt.Printf("%d is odd: is %t\n", 18, odd(18))
    // 18 is odd: is false
}

func even(nr int) bool {
    if nr == 0 {
        return true
    }
    return odd(RevSign(nr) - 1)
}

func odd(nr int) bool {
    if nr == 0 {
        return false
    }
    return even(RevSign(nr) - 1)
}

func RevSign(nr int) int {
    if nr < 0 {
        return -nr
    }
    return nr
}
```

练习题

练习 6.4

重写本节中生成斐波那契数列的程序并返回两个命名返回值（详见第 6.2 节），即数列中的位置和对应的值，例如 5 与 4，89 与 10。

练习 6.5

使用递归函数从 10 打印到 1。

练习 6.6

实现一个输出前 30 个整数的阶乘的程序。

n! 的阶乘定义为：`n! = n * (n-1)!, 0! = 1`，因此它非常适合使用递归函数来实现。

然后，使用命名返回值来实现这个程序的第二个版本。

特别注意的是，使用 int 类型最多只能计算到 12 的阶乘，因为一般情况下 int 类型的大小为 32 位，继续计算会导致溢出错误。那么，如何才能解决这个问题呢？

最好的解决方案就是使用 big 包（详见第 9.4 节）。

将函数作为参数

6.7 将函数作为参数

函数可以作为其它函数的参数进行传递，然后在其它函数内调用执行，一般称之为回调。下面是一个将函数作为参数的简单例子（function_parameter.go）：

```
package main

import (
    "fmt"
)

func main() {
    callback(1, Add)
}

func Add(a, b int) {
    fmt.Printf("The sum of %d and %d is: %d\n", a, b, a+b)
}

func callback(y int, f func(int, int)) {
    f(y, 2) // this becomes Add(1, 2)
}
```

输出：

```
The sum of 1 and 2 is: 3
```

将函数作为参数的最好的例子是函数 `strings.IndexFunc()`：

该函数的签名是 `func IndexFunc(s string, f func(c rune) bool) int`，它的返回值是在函数 `f(c)` 返回 true、-1 或从未返回时的索引值。

例如 `strings.IndexFunc(line, unicode.IsSpace)` 就会返回 `line` 中第一个空白字符的索引值。当然，您也可以书写自己的函数：

```
func IsAscii(c int) bool {
    if c > 255 {
        return false
    }
    return true
}
```

在第 14.10.1 节中，我们将会根据一个客户端/服务端程序作为示例对这个用法进行深入讨论。

```
type binOp func(a, b int) int
func run(op binOp, req *Request) { … }
```

练习 6.7

包 `strings` 中的 `Map` 函数和 `strings.IndexFunc()` 一样都是非常好的使用例子。请学习它的源代码并基于该函数书写一个程序，要求将指定文本内的所有非 ASCII 字符替换成 `?` 或空格。您需要怎么做才能删除这些字符呢？

闭包

6.8 闭包

当我们不希望给函数起名字的时候，可以使用匿名函数，例如：`func(x, y int) int { return x + y }`。

这样的一个函数不能够独立存在（编译器会返回错误：`non-declaration statement outside function body`），但可以被赋值于某个变量，即保存函数的地址到变量中：`fplus := func(x, y int) int { return x + y }`，然后通过变量名对函数进行调用：`fplus(3,4)`。

当然，您也可以直接对匿名函数进行调用：`func(x, y int) int { return x + y }(3, 4)`。

下面是一个计算从 1 到 1 百万整数的总和的匿名函数：

```
func() {
    sum := 0
    for i := 1; i <= 1e6; i++ {
        sum += i
    }
}()
```

表示参数列表的第一对括号必须紧挨着关键字 `func`，因为匿名函数没有名称。花括号 `{}` 涵盖着函数体，最后的一对括号表示对该匿名函数的调用。

下面的例子展示了如何将匿名函数赋值给变量并对其进行调用（function_literal.go）：

```
package main

import "fmt"

func main() {
    f()
}
func f() {
    for i := 0; i < 4; i++ {
        g := func(i int) { fmt.Printf("%d ", i) } //此例子中只是为了演示匿名函数可分配不同的内存地址，在现实开发中，不应该把该部分信息放置到循环中。
        g(i)
        fmt.Printf(" - g is of type %T and has value %v\n", g, g)
    }
}
```

输出：

```
0 - g is of type func(int) and has value 0x681a80
1 - g is of type func(int) and has value 0x681b00
2 - g is of type func(int) and has value 0x681ac0
3 - g is of type func(int) and has value 0x681400
```

我们可以看到变量 `g` 代表的是 `func(int)`，变量的值是一个内存地址。

所以我们实际上拥有的是一个函数值：匿名函数可以被赋值给变量并作为值使用。

练习 6.8 在 main 函数中写一个用于打印 `Hello World` 字符串的匿名函数并赋值给变量 `fv`，然后调用该函数并打印变量 `fv` 的类型。

匿名函数像所有函数一样可以接受或不接受参数。下面的例子展示了如何传递参数到匿名函数中：

```
func (u string) {
    fmt.Println(u)
    ...
}(v)
```

请学习以下示例并思考（return_defer.go）：函数 `f` 返回时，变量 `ret` 的值是什么？

```
package main

import "fmt"

func f() (ret int) {
    defer func() {
        ret++
    }()
    return 1
}
func main() {
    fmt.Println(f())
}
```

变量 `ret` 的值为 2，因为 `ret++` 是在执行 `return 1` 语句后发生的。

这可用于在返回语句之后修改返回的 `error` 时使用。

defer 语句和匿名函数

关键字 `defer` （详见第 6.4 节）经常配合匿名函数使用，它可以用于改变函数的命名返回值。

匿名函数还可以配合 `go` 关键字来作为 goroutine 使用（详见第 14 章和第 16.9 节）。

匿名函数同样被称之为闭包（函数式语言的术语）：它们被允许调用定义在其它环境下的变量。闭包可使得某个函数捕捉到一些外部状态，例如：函数被创建时的状态。另一种表示方式为：一个闭包继承了函数所声明时的作用域。这种状态（作用域内的变量）都被共享到闭包的环境中，因此这些变量可以在闭包中被操作，直到被销毁，详见第 6.9 节中的示例。闭包经常被用作包装函数：它们会预先定义好 1 个或多个参数以用于包装，详见下一节中的示例。另一个不错的应用就是使用闭包来完成更加简洁的错误检查（详见第 16.10.2 节）。

应用闭包：将函数作为返回值

6.9 应用闭包：将函数作为返回值

在程序 `function_return.go` 中我们将会看到函数 Add2 和 Adder 均会返回签名为 `func(b int) int` 的函数：

```
func Add2() (func(b int) int)
func Adder(a int) (func(b int) int)
```

函数 Add2 不接受任何参数，但函数 Adder 接受一个 int 类型的整数作为参数。

我们也可以将 Adder 返回的函数存到变量中（function_return.go）。

```
package main

import "fmt"

func main() {
    // make an Add2 function, give it a name p2, and call it:
    p2 := Add2()
    fmt.Printf("Call Add2 for 3 gives: %v\n", p2(3))
    // make a special Adder function, a gets value 2:
    TwoAdder := Adder(2)
    fmt.Printf("The result is: %v\n", TwoAdder(3))
}

func Add2() func(b int) int {
    return func(b int) int {
        return b + 2
    }
}

func Adder(a int) func(b int) int {
    return func(b int) int {
        return a + b
    }
}
```

输出：

```
Call Add2 for 3 gives: 5
The result is: 5
```

下例为一个略微不同的实现（function_closure.go）：

```
package main

import "fmt"

func main() {
    var f = Adder()
    fmt.Print(f(1), " - ")
    fmt.Print(f(20), " - ")
```

```
    fmt.Print(f(300))
}

func Adder() func(int) int {
    var x int
    return func(delta int) int {
        x += delta
        return x
    }
}
```

函数 Adder() 现在被赋值到变量 f 中（类型为 `func(int) int`）。

输出：

```
1 - 21 - 321
```

三次调用函数 f 的过程中函数 Adder() 中变量 delta 的值分别为：1、20 和 300。

我们可以看到，在多次调用中，变量 x 的值是被保留的，即 `0 + 1 = 1`，然后 `1 + 20 = 21`，最后 `21 + 300 = 321`：闭包函数保存并积累其中的变量的值，不管外部函数退出与否，它都能够继续操作外部函数中的局部变量。

这些局部变量同样可以是参数，例如之前例子中的 `Adder(as int)`。

这些例子清楚地展示了如何在 Go 语言中使用闭包。

在闭包中使用到的变量可以是在闭包函数体内声明的，也可以是在外部函数声明的：

```
var g int
go func(i int) {
    s := 0
    for j := 0; j < i; j++ { s += j }
    g = s
}(1000) // Passes argument 1000 to the function literal.
```

这样闭包函数就能够被应用到整个集合的元素上，并修改它们的值。然后这些变量就可以用于表示或计算全局或平均值。

练习 6.9 不使用递归但使用闭包改写第 6.6 节中的斐波那契数列程序。

练习 6.10

学习并理解以下程序的工作原理：

一个返回值为另一个函数的函数可以被称之为工厂函数，这在您需要创建一系列相似的函数的时候非常有用：书写一个工厂函数而不是针对每种情况都书写一个函数。下面的函数演示了如何动态返回追加后缀的函数：

```
func MakeAddSuffix(suffix string) func(string) string {
    return func(name string) string {
        if !strings.HasSuffix(name, suffix) {
            return name + suffix
        }
        return name
    }
}
```

现在，我们可以生成如下函数：

```
addBmp := MakeAddSuffix(".bmp")
addJpeg := MakeAddSuffix(".jpeg")
```

然后调用它们：

```
addBmp("file") // returns: file.bmp
addJpeg("file") // returns: file.jpeg
```

可以返回其它函数的函数和接受其它函数作为参数的函数均被称之为高阶函数，是函数式语言的特点。我们已经在第 6.7 中得知函数也是一种值，因此很显然 Go 语言具有一些函数式语言的特性。闭包在 Go 语言中非常常见，常用于 goroutine 和管道操作（详见第 14.8-14.9 节）。在第 11.14 节的程序中，我们将会看到 Go 语言中的函数在处理混合对象时的强大能力。

使用闭包调试

6.10 使用闭包调试

当您在分析和调试复杂的程序时，无数个函数在不同的代码文件中相互调用，如果这时候能够准确地知道哪个文件中的具体哪个函数正在执行，对于调试是十分有帮助的。您可以使用 `runtime` 或 `log` 包中的特殊函数来实现这样的功能。包 `runtime` 中的函数 `Caller()` 提供了相应的信息，因此可以在需要的时候实现一个 `where()` 闭包函数来打印函数执行的位置：

```
where := func() {
    _, file, line, _ := runtime.Caller(1)
    log.Printf("%s:%d", file, line)
}
where()
// some code
where()
// some more code
where()
```

您也可以设置 `log` 包中的 `flag` 参数来实现：

```
log.SetFlags(log.Llongfile)
log.Print("")
```

或使用一个更加简短版本的 `where` 函数：

```
var where = log.Print
func func1() {
where()
... some code
where()
... some code
where()
}
```

计算函数执行时间

6.11 计算函数执行时间

有时候，能够知道一个计算执行消耗的时间是非常有意义的，尤其是在对比和基准测试中。最简单的一个办法就是在计算开始之前设置一个起始时候，再由计算结束时的结束时间，最后取出它们的差值，就是这个计算所消耗的时间。想要实现这样的做法，可以使用 `time` 包中的 `Now()` 和 `Sub` 函数：

```
start := time.Now()
longCalculation()
end := time.Now()
delta := end.Sub(start)
fmt.Printf("longCalculation took this amount of time: %s\n", delta)
```

您可以查看示例 6.20 fibonacci.go 作为实例学习。

如果您对一段代码进行了所谓的优化，请务必对它们之间的效率进行对比再做出最后的判断。在接下来的章节中，我们会学习如何进行有价值的优化操作。

通过内存缓存来提升性能

6.12 通过内存缓存来提升性能

当在进行大量的计算时，提升性能最直接有效的一种方式就是避免重复计算。通过在内存中缓存和重复利用相同计算的结果，称之为内存缓存。最明显的例子就是生成斐波那契数列的程序（详见第 6.6 和 6.11 节）：

要计算数列中第 n 个数字，需要先得到之前两个数的值，但很明显绝大多数情况下前两个数的值都是已经计算过的。即每个更后面的数都是基于之前计算结果的重复计算，正如示例 6.11 fibonnaci.go 所展示的那样。

而我们要做就是将第 n 个数的值存在数组中索引为 n 的位置（详见第 7 章），然后在数组中查找是否已经计算过，如果没有找到，则再进行计算。

程序 Listing 6.17 - fibonacci_memoization.go 就是依照这个原则实现的，下面是计算到第 40 位数字的性能对比：

- 普通写法：4.730270 秒
- 内存缓存：0.001000 秒

内存缓存的优势显而易见，而且您还可以将它应用到其它类型的计算中，例如使用 map（详见第 7 章）而不是数组或切片（Listing 6.21 - fibonacci_memoization.go）：

```
package main

import (
    "fmt"
    "time"
)

const LIM = 41

var fibs [LIM]uint64

func main() {
    var result uint64 = 0
    start := time.Now()
    for i := 0; i < LIM; i++ {
        result = fibonacci(i)
        fmt.Printf("fibonacci(%d) is: %d\n", i, result)
    }
    end := time.Now()
    delta := end.Sub(start)
    fmt.Printf("longCalculation took this amount of time: %s\n", delta)
}
func fibonacci(n int) (res uint64) {
    // memoization: check if fibonacci(n) is already known in array:
    if fibs[n] != 0 {
        res = fibs[n]
        return
    }
    if n <= 1 {
        res = 1
    } else {
        res = fibonacci(n-1) + fibonacci(n-2)
    }
    fibs[n] = res
```

```
    return
}
```

内存缓存的技术在使用计算成本相对昂贵的函数时非常有用（不仅限于例子中的递归），譬如大量进行相同参数的运算。这种技术还可以应用于纯函数中，即相同输入必定获得相同输出的函数。

数组与切片

7.0 数组与切片

这章我们开始剖析 **容器**, 它是可以包含大量条目（item）的数据结构, 例如数组、切片和 map。从这看到 Go 明显受到 Python 的影响。

以 `[]` 符号标识的数组类型几乎在所有的编程语言中都是一个基本主力。Go 语言中的数组也是类似的，只是有一些特点。Go 没有 C 那么灵活，但是拥有切片（slice）类型。这是一种建立在 Go 语言数组类型之上的抽象，要想理解切片我们必须先理解数组。数组有特定的用处，但是却有一些呆板，所以在 Go 语言的代码里并不是特别常见。相对的，切片确实随处可见的。它们构建在数组之上并且提供更强大的能力和便捷。

声明和初始化

7.1 声明和初始化

7.1.1 概念

数组是具有相同 **唯一类型** 的一组已编号且长度固定的数据项序列（这是一种同构的数据结构）；这种类型可以是任意的原始类型例如整型、字符串或者自定义类型。数组长度必须是一个常量表达式，并且必须是一个非负整数。数组长度也是数组类型的一部分，所以[5]int和[10]int是属于不同类型的。数组的编译时值初始化是按照数组顺序完成的（如下）。

注意事项 如果我们想让数组元素类型为任意类型的话可以使用空接口作为类型（参考 第 11 章）。当使用值时我们必须先做一个类型判断（参考 第 11 章）。

数组元素可以通过 **索引**（位置）来读取（或者修改），索引从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。（数组以 0 开始在所有类 C 语言中是相似的）。元素的数目，也称为长度或者数组大小必须是固定的并且在声明该数组时就给出（编译时需要知道数组长度以便分配内存）；数组长度最大为 2Gb。

声明的格式是：

```
var identifier [len]type
```

例如：

```
var arr1 [5]int
```

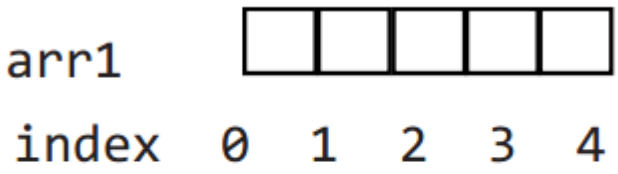


Fig 7.1: Array in memory

每个元素是一个整型值，当声明数组时所有的元素都会被自动初始化为默认值 0。

arr1 的长度是 5，索引范围从 0 到 `len(arr1)-1` 。

第一个元素是 `arr1[0]` ，第三个元素是 `arr1[2]` ；总体来说索引 i 代表的元素是 `arr1[i]` ，最后一个元素是 `arr1[len(arr1)-1]` 。

对索引项为 i 的数组元素赋值可以这么操作： `arr[i] = value` ，所以数组是 **可变的**。

只有有效的索引可以被使用，当使用等于或者大于 `len(arr1)` 的索引时：如果编译器可以检测到，会给出索引超限的提示信息；如果检测不到的话编译会通过而运行时会 panic:（参考 第 13 章）

```
runtime error: index out of range
```

由于索引的存在，遍历数组的方法自然就是使用 for 结构:

- 通过 for 初始化数组项
- 通过 for 打印数组元素
- 通过 for 依次处理元素

示例 7.1 for_arrays.go

```
package main
import "fmt"

func main() {
    var arr1 [5]int

    for i:=0; i < len(arr1); i++ {
        arr1[i] = i * 2
    }

    for i:=0; i < len(arr1); i++ {
        fmt.Printf("Array at index %d is %d\n", i, arr1[i])
    }
}
```

输出结果:

```
Array at index 0 is 0
Array at index 1 is 2
Array at index 2 is 4
Array at index 3 is 6
Array at index 4 is 8
```

for 循环中的条件非常重要： `i < len(arr1)` ，如果写成 `i <= len(arr1)` 的话会产生越界错误。

IDIOM:

```
for i:=0; i < len(arr1); i++ {
    arr1[i] = ...
}
```

也可以使用 for-range 的生成方式:

IDIOM:

```
for i,_:= range arr1 {
...
}
```

在这里i也是数组的索引。当然这两种 for 结构对于切片（slices）（参考 第 7 章）来说也同样适用。

问题 7.1 下面代码段的输出是什么？

```
a := [...]string{"a", "b", "c", "d"}
for i := range a {
    fmt.Println("Array item", i, "is", a[i])
}
```

Go 语言中的数组是一种 **值类型**（不像 C/C++ 中是指向首元素的指针），所以可以通过 `new()` 来创建： `var arr1 = new([5]int)` 。

那么这种方式和 `var arr2 [5]int` 的区别是什么呢？arr1 的类型是 `*[5]int` ，而 arr2的类型是 `[5]int` 。

这样的结果就是当把一个数组赋值给另一个时，需要再做一次数组内存的拷贝操作。例如：

```
arr2 := *arr1
arr2[2] = 100
```

这样两个数组就有了不同的值，在赋值后修改 arr2 不会对 arr1 生效。

所以在函数中数组作为参数传入时，如 `func1(arr2)` ，会产生一次数组拷贝，func1 方法不会修改原始的数组 arr2。

如果你想修改原数组，那么 arr2 必须通过&操作符以引用方式传过来，例如 func1(&arr2），下面是一个例子

示例 7.2 pointer_array.go:

```
package main
import "fmt"
func f(a [3]int) { fmt.Println(a) }
func fp(a *[3]int) { fmt.Println(a) }

func main() {
    var ar [3]int
    f(ar)  // passes a copy of ar
    fp(&ar) // passes a pointer to ar
}
```

输出结果：

```
[0 0 0]
&[0 0 0]
```

另一种方法就是生成数组切片并将其传递给函数（详见第 7.1.4 节）。

练习

练习7.1：array_value.go: 证明当数组赋值时，发生了数组内存拷贝。

练习7.2：for_array.go: 写一个循环并用下标给数组赋值（从 0 到 15）并且将数组打印在屏幕上。

练习7.3：fibonacci_array.go: 在第 6.6 节我们看到了一个递归计算 Fibonacci 数值的方法。但是通过数组我们可以更快的计算出 Fibonacci 数。完成该方法并打印出前 50 个 Fibonacci 数字。

7.1.2 数组常量

如果数组值已经提前知道了，那么可以通过 **数组常量** 的方法来初始化数组，而不用依次使用 `[]=` 方法（所有的组成元素都有相同的常量语法）。

示例 7.3 array_literals.go

```
package main
import "fmt"

func main() {
    // var arrAge = [5]int{18, 20, 15, 22, 16}
    // var arrLazy = [...]int{5, 6, 7, 8, 22}
    // var arrLazy = []int{5, 6, 7, 8, 22}    //注：初始化得到的实际上是切片slice
    var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
    // var arrKeyValue = []string{3: "Chris", 4: "Ron"}    //注：初始化得到的实际上是切片slice

    for i:=0; i < len(arrKeyValue); i++ {
        fmt.Printf("Person at %d is %s\n", i, arrKeyValue[i])
    }
}
```

第一种变化：

```
var arrAge = [5]int{18, 20, 15, 22, 16}
```

注意 `[5]int` 可以从左边起开始忽略：`[10]int {1, 2, 3}` :这是一个有 10 个元素的数组，除了前三个元素外其他元素都为 0。

第二种变化：

```
var arrLazy = [...]int{5, 6, 7, 8, 22}
```

`...` 可同样可以忽略，从技术上说它们其实变化成了切片。

第三种变化： `key: value 语法`

```
var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
```

只有索引 3 和 4 被赋予实际的值，其他元素都被设置为空的字符串，所以输出结果为：

```
Person at 0 is
Person at 1 is
Person at 2 is
Person at 3 is Chris
Person at 4 is Ron
```

在这里数组长度同样可以写成 `...` 。

你可以取任意数组常量的地址来作为指向新实例的指针。

示例 7.4 pointer_array2.go

```
package main
import "fmt"

func fp(a *[3]int) { fmt.Println(a) }

func main() {
    for i := 0; i < 3; i++ {
```

```
        fp(&[3]int{i, i * i, i * i * i})
    }
}
```

输出结果：

```
&[0 0 0]
&[1 1 1]
&[2 4 8]
```

几何点（或者数学向量）是一个使用数组的经典例子。为了简化代码通常使用一个别名：

```
type Vector3D [3]float32
var vec Vector3D
```

7.1.3 多维数组

数组通常是一维的，但是可以用来组装成多维数组，例如： `[3][5]int` ， `[2][2][2]float64` 。

内部数组总是长度相同的。Go 语言的多维数组是矩形式的（唯一的例外是切片的数组，参见第 7.2.5 节）。

示例 7.5 multidim_array.go

```
package main
const (
    WIDTH  = 1920
    HEIGHT = 1080
)

type pixel int
var screen [WIDTH][HEIGHT]pixel

func main() {
    for y := 0; y < HEIGHT; y++ {
        for x := 0; x < WIDTH; x++ {
            screen[x][y] = 0
        }
    }
}
```

7.1.4 将数组传递给函数

把一个大数组传递给函数会消耗很多内存。有两种方法可以避免这种现象：

- 传递数组的指针
- 使用数组的切片

接下来的例子阐明了第一种方法：

示例 7.6 array_sum.go

```
package main
import "fmt"
```

```
func main() {
    array := [3]float64{7.0, 8.5, 9.1}
    x := Sum(&array) // Note the explicit address-of operator
    // to pass a pointer to the array
    fmt.Printf("The sum of the array is: %f", x)
}

func Sum(a *[3]float64) (sum float64) {
    for _, v := range a { // derefencing *a to get back to the array is not necessary!
        sum += v
    }
    return
}
```

输出结果：

```
The sum of the array is: 24.600000
```

但这在 Go 中并不常用，通常使用切片（参考 第 7.2 节）。

切片

7.2 切片

7.2.1 概念

切片（slice）是对数组一个连续片段的引用（该数组我们称之为相关数组，通常是匿名的），所以切片是一个引用类型（因此更类似于 C/C++ 中的数组类型，或者 Python 中的 list 类型）。这个片段可以是整个数组，或者是由起始和终止索引标识的一些项的子集。需要注意的是，终止索引标识的项不包括在切片内。切片提供了一个相关数组的动态窗口。

切片是可索引的，并且可以由 `len()` 函数获取长度。

给定项的切片索引可能比相关数组的相同元素的索引小。和数组不同的是，切片的长度可以在运行时修改，最小为 0 最大为相关数组的长度：切片是一个 **长度可变的数组**。

切片提供了计算容量的函数 `cap()` 可以测量切片最长可以达到多少：它等于切片的长度 + 数组除切片之外的长度。如果 s 是一个切片， `cap(s)` 就是从 `s[0]` 到数组末尾的数组长度。切片的长度永远不会超过它的容量，所以对于 切片 s 来说该不等式永远成立： `0 <= len(s) <= cap(s)` 。

多个切片如果表示同一个数组的片段，它们可以共享数据；因此一个切片和相关数组的其他切片是共享存储的，相反，不同的数组总是代表不同的存储。数组实际上是切片的构建块。

优点 因为切片是引用，所以它们不需要使用额外的内存并且比使用数组更有效率，所以在 Go 代码中 切片比数组更常用。

声明切片的格式是： `var identifier []type` （不需要说明长度）。

一个切片在未初始化之前默认为 nil，长度为 0。

切片的初始化格式是： `var slice1 []type = arr1[start:end]` 。

这表示 slice1 是由数组 arr1 从 start 索引到 `end-1` 索引之间的元素构成的子集（切分数组，start:end 被称为 slice 表达式）。所以 `slice1[0]` 就等于 `arr1[start]` 。这可以在 arr1 被填充前就定义好。

如果某个人写： `var slice1 []type = arr1[:]` 那么 slice1 就等于完整的 arr1 数组（所以这种表示方式是 `arr1[0:len(arr1)]` 的一种缩写）。另外一种表述方式是： `slice1 = &arr1` 。

`arr1[2:]` 和 `arr1[2:len(arr1)]` 相同，都包含了数组从第三个到最后的所有元素。

`arr1[:3]` 和 `arr1[0:3]` 相同，包含了从第一个到第三个元素（不包括第四个）。

如果你想去掉 slice1 的最后一个元素，只要 `slice1 = slice1[:len(slice1)-1]` 。

一个由数字 1、2、3 组成的切片可以这么生成： `s := [3]int{1,2,3}[:]` (注: 应先用 `s := [3]int{1, 2, 3}` 生成数组, 再使用 `s[:]` 转成切片) 甚至更简单的 `s := []int{1,2,3}` 。

`s2 := s[:]` 是用切片组成的切片，拥有相同的元素，但是仍然指向相同的相关数组。

一个切片 s 可以这样扩展到它的大小上限： `s = s[:cap(s)]` ，如果再扩大的话就会导致运行时错误（参见第 7.7 节）。

对于每一个切片（包括 string），以下状态总是成立的：

```
s == s[:i] + s[i:] // i是一个整数且: 0 <= i <= len(s)
len(s) <= cap(s)
```

切片也可以用类似数组的方式初始化：`var x = []int{2, 3, 5, 7, 11}`。这样就创建了一个长度为 5 的数组并且创建了一个相关切片。

切片在内存中的组织方式实际上是一个有 3 个域的结构体：指向相关数组的指针，切片长度以及切片容量。下图给出了一个长度为 2，容量为 4 的切片y。

- `y[0] = 3` 且 `y[1] = 5`。
- 切片 `y[0:4]` 由 元素 3，5，7 和 11 组成。

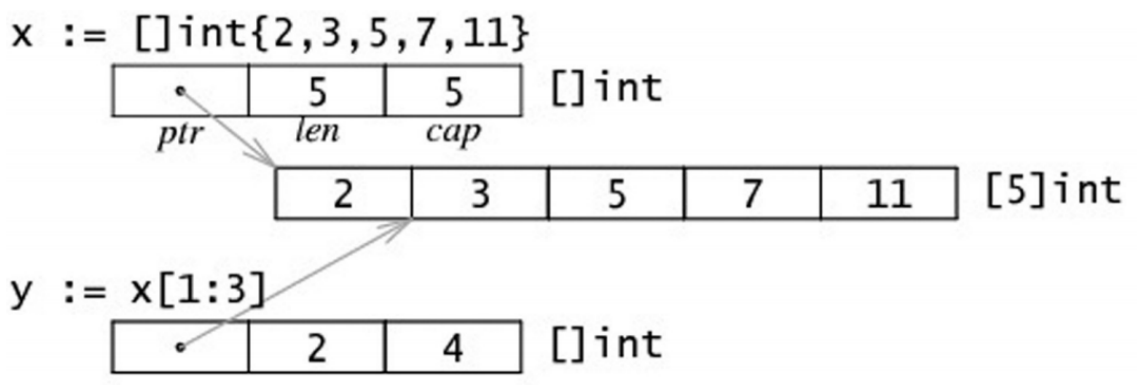


Fig 7.2: Slice in memory

示例 7.7 array_slices.go

```
package main
import "fmt"

func main() {
    var arr1 [6]int
    var slice1 []int = arr1[2:5] // item at index 5 not included!

    // load the array with integers: 0,1,2,3,4,5
    for i := 0; i < len(arr1); i++ {
        arr1[i] = i
    }

    // print the slice
    for i := 0; i < len(slice1); i++ {
        fmt.Printf("Slice at %d is %d\n", i, slice1[i])
    }

    fmt.Printf("The length of arr1 is %d\n", len(arr1))
    fmt.Printf("The length of slice1 is %d\n", len(slice1))
    fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))

    // grow the slice
    slice1 = slice1[0:4]
    for i := 0; i < len(slice1); i++ {
        fmt.Printf("Slice at %d is %d\n", i, slice1[i])
    }
    fmt.Printf("The length of slice1 is %d\n", len(slice1))
    fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))

    // grow the slice beyond capacity
```

```
    //slice1 = slice1[0:7 ] // panic: runtime error: slice bound out of range
}
```

输出：

```
Slice at 0 is 2
Slice at 1 is 3
Slice at 2 is 4
The length of arr1 is 6
The length of slice1 is 3
The capacity of slice1 is 4
Slice at 0 is 2
Slice at 1 is 3
Slice at 2 is 4
Slice at 3 is 5
The length of slice1 is 4
The capacity of slice1 is 4
```

如果 s2 是一个 slice，你可以将 s2 向后移动一位 `s2 = s2[1:]` ，但是末尾没有移动。切片只能向后移动， `s2 = s2[-1:]` 会导致编译错误。切片不能被重新分片以获取数组的前一个元素。

注意 绝对不要用指针指向 slice。切片本身已经是一个引用类型，所以它本身就是一个指针!!

问题 7.2： 给定切片 `b:= []byte{'g', 'o', 'l', 'a', 'n', 'g'}` ，那么 `b[1:4]` 、 `b[:2]` 、 `b[2:]` 和 `b[:]` 分别是什么？

7.2.2 将切片传递给函数

如果你有一个函数需要对数组做操作，你可能总是需要把参数声明为切片。当你调用该函数时，把数组分片，创建为一个 切片引用并传递给该函数。这里有一个计算数组元素和的方法:

```
func sum(a []int) int {
    s := 0
    for i := 0; i < len(a); i++ {
        s += a[i]
    }
    return s
}

func main() {
    var arr = [5]int{0, 1, 2, 3, 4}
    sum(arr[:])
}
```

7.2.3 用 make() 创建一个切片

当相关数组还没有定义时，我们可以使用 make() 函数来创建一个切片 同时创建好相关数组： `var slice1 []type = make([]type, len)` 。

也可以简写为 `slice1 := make([]type, len)` ，这里 `len` 是数组的长度并且也是 `slice` 的初始长度。

所以定义 `s2 := make([]int, 10)` ，那么 `cap(s2) == len(s2) == 10` 。

make 接受 2 个参数：元素的类型以及切片的元素个数。

如果你想创建一个 slice1，它不占用整个数组，而只是占用以 len 为个数个项，那么只要：`slice1 := make([]type, len, cap)`。

make 的使用方式是：`func make([]T, len, cap)`，其中 cap 是可选参数。

所以下面两种方法可以生成相同的切片：

```
make([]int, 50, 100)
new([100]int)[0:50]
```

下图描述了使用 make 方法生成的切片的内存结构：![]

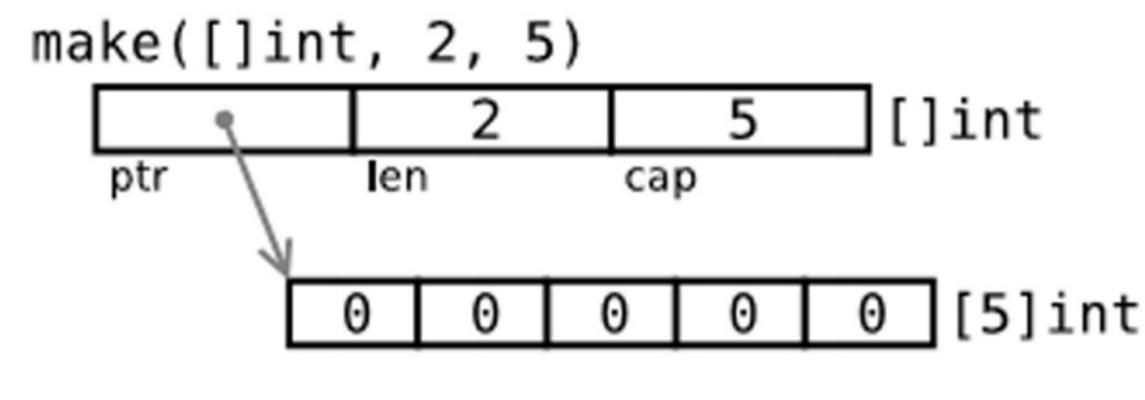


Fig 7.2: Slice in memory

示例 7.8 make_slice.go

```
package main
import "fmt"

func main() {
    var slice1 []int = make([]int, 10)
    // load the array/slice:
    for i := 0; i < len(slice1); i++ {
        slice1[i] = 5 * i
    }

    // print the slice:
    for i := 0; i < len(slice1); i++ {
        fmt.Printf("Slice at %d is %d\n", i, slice1[i])
    }
    fmt.Printf("\nThe length of slice1 is %d\n", len(slice1))
    fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))
}
```

输出：

```
Slice at 0 is 0
Slice at 1 is 5
Slice at 2 is 10
Slice at 3 is 15
```

```
Slice at 4 is 20
Slice at 5 is 25
Slice at 6 is 30
Slice at 7 is 35
Slice at 8 is 40
Slice at 9 is 45

The length of slicel is 10
The capacity of slicel is 10
```

因为字符串是纯粹不可变的字节数组，它们也可以被切分成 切片。

练习 7.4： fobinacci_funcarray.go: 为练习 7.3 写一个新的版本，主函数调用一个使用序列个数作为参数的函数，该函数返回一个大小为序列个数的 Fibonacci 切片。

7.2.4 new() 和 make() 的区别

看起来二者没有什么区别，都在堆上分配内存，但是它们的行为不同，适用于不同的类型。

- new(T) 为每个新的类型T分配一片内存，初始化为 0 并且返回类型为*T的内存地址：这种方法 **返回一个指向类型为 T，值为 0 的地址的指针**，它适用于值类型如数组和结构体（参见第 10 章）；它相当于 `&T{}`。
- make(T) **返回一个类型为 T 的初始值**，它只适用于3种内建的引用类型：切片、map 和 channel（参见第 8 章，第 13 章）。

换言之，new 函数分配内存，make 函数初始化；下图给出了区别：

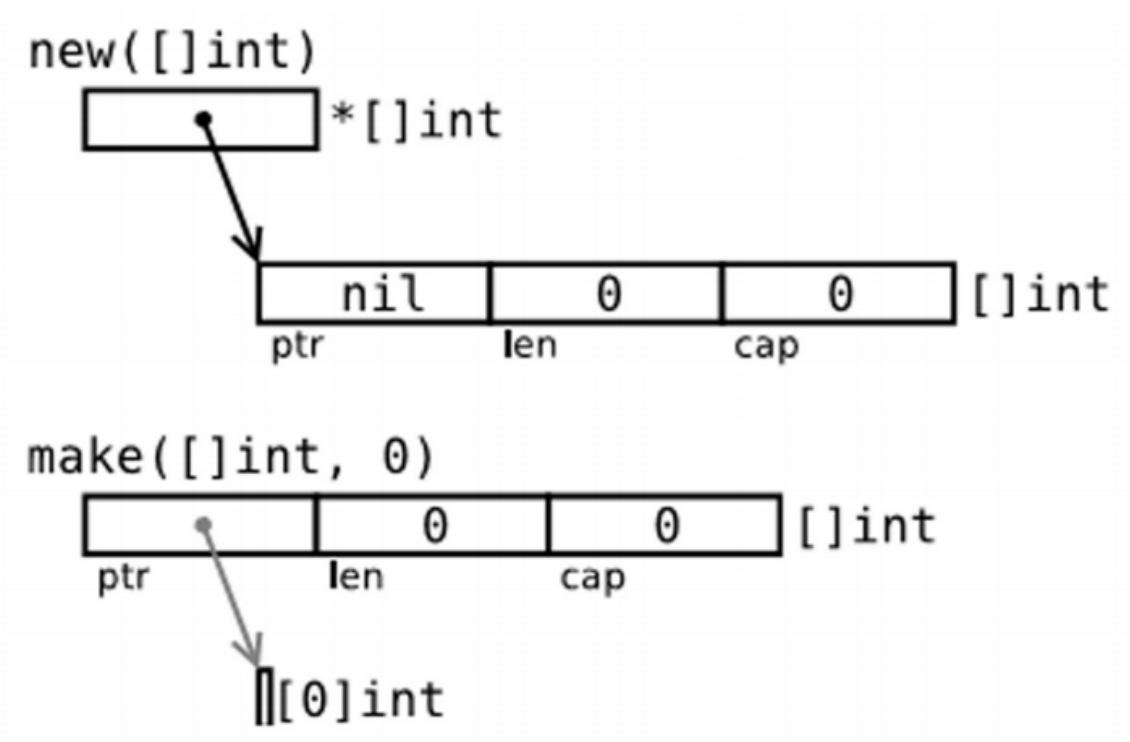


Fig 7.3: Difference between new() and make()

在图 7.3 的第一幅图中：

```
var p *[]int = new([]int) // *p == nil; with len and cap 0
p := new([]int)
```

在第二幅图中，`p := make([]int, 0)`，切片 已经被初始化，但是指向一个空的数组。

以上两种方式实用性都不高。下面的方法：

```
var v []int = make([]int, 10, 50)
```

或者

```
v := make([]int, 10, 50)
```

这样分配一个有 50 个 int 值的数组，并且创建了一个长度为 10，容量为 50 的 切片 v，该 切片 指向数组的前 10 个元素。

问题 7.3 给定 `s := make([]byte, 5)`，len(s) 和 cap(s) 分别是多少？ `s = s[2:4]`，len(s) 和 cap(s) 又分别是多少？
问题 7.4 假设 `s1 := []byte{'p', 'o', 'e', 'm'}` 且 `s2 := s1[2:]`，s2 的值是多少？如果我们执行 `s2[1] = 't'`，s1 和 s2 现在的值又分别是多少？

译者注：如何理解new、make、slice、map、channel的关系

1.slice、map以及channel都是golang内建的一种引用类型，三者在内存中存在多个组成部分，
需要对内存组成部分初始化后才能使用，而make就是对三者进行初始化的一种操作方式

2. new 获取的是存储指定变量内存地址的一个变量，对于变量内部结构并不会执行相应的初始化操作，
所以slice、map、channel需要make进行初始化并获取对应的内存地址，而非new简单的获取内存地址

7.2.5 多维 切片

和数组一样，切片通常也是一维的，但是也可以由一维组合成高维。通过分片的分片（或者切片的数组），长度可以任意动态变化，所以 Go 语言的多维切片可以任意切分。而且，内层的切片必须单独分配（通过 make 函数）。

7.2.6 bytes 包

类型 `[]byte` 的切片十分常见，Go 语言有一个 bytes 包专门用来解决这种类型的操作方法。

bytes 包和字符串包十分类似（参见第 4.7 节）。而且它还包含一个十分有用的类型 Buffer:

```
import "bytes"

type Buffer struct {
    ...
}
```

这是一个长度可变的 bytes 的 buffer，提供 Read 和 Write 方法，因为读写长度未知的 bytes 最好使用 buffer。

Buffer 可以这样定义：`var buffer bytes.Buffer`。

或者使用 new 获得一个指针：`var r *bytes.Buffer = new(bytes.Buffer)`。

或者通过函数：`func NewBuffer(buf []byte) *Buffer`，创建一个 Buffer 对象并且用 buf 初始化好；NewBuffer 最好用在从 buf 读取的时候使用。

通过 buffer 串联字符串

类似于 Java 的 StringBuilder 类。

在下面的代码段中，我们创建一个 buffer，通过 `buffer.WriteString(s)` 方法将字符串 s 追加到后面，最后再通过 `buffer.String()` 方法转换为 string：

```
var buffer bytes.Buffer
for {
    if s, ok := getNextString(); ok { //method getNextString() not shown here
        buffer.WriteString(s)
    } else {
        break
    }
}
fmt.Print(buffer.String(), "\n")
```

这种实现方式比使用 `+=` 要更节省内存和 CPU，尤其是要串联的字符串数目特别多的时候。

练习 7.5 给定切片 sl，将一个 `[]byte` 数组追加到 sl 后面。写一个函数 `Append(slice, data []byte) []byte`，该函数在 sl 不能存储更多数据的时候自动扩容。
练习 7.6 把一个缓存 buf 分片成两个 切片：第一个是前 n 个 bytes，后一个是剩余的，用一行代码实现。

For-range 结构

7.3 For-range 结构

这种构建方法可以应用于数组和切片:

```
for ix, value := range slice1 {
    ...
}
```

第一个返回值 ix 是数组或者切片的索引，第二个是在该索引位置的值；他们都是仅在 for 循环内部可见的局部变量。value 只是 slice1 某个索引位置的值的一个拷贝，不能用来修改 slice1 该索引位置的值。

示例 7.9 slices_forrange.go

```
package main

import "fmt"

func main() {
    var slice1 []int = make([]int, 4)

    slice1[0] = 1
    slice1[1] = 2
    slice1[2] = 3
    slice1[3] = 4

    for ix, value := range slice1 {
        fmt.Printf("Slice at %d is: %d\n", ix, value)
    }
}
```

示例 7.10 slices_forrange2.go

```
package main
import "fmt"

func main() {
    seasons := []string{"Spring", "Summer", "Autumn", "Winter"}
    for ix, season := range seasons {
        fmt.Printf("Season %d is: %s\n", ix, season)
    }

    var season string
    for _, season = range seasons {
        fmt.Printf("%s\n", season)
    }
}
```

slices_forrange2.go 给出了一个关于字符串的例子， `_` 可以用于忽略索引。

如果你只需要索引，你可以忽略第二个变量，例如：

```
for ix := range seasons {
    fmt.Printf("%d", ix)
}
// Output: 0 1 2 3
```

如果你需要修改 `seasons[ix]` 的值可以使用这个版本。

多维切片下的 for-range：

通过计算行数和矩阵值可以很方便的写出如（参考第 7.1.3 节）的 for 循环来，例如（参考第 7.5 节的例子 multidim_array.go）：

```
for row := range screen {
    for column := range screen[row] {
        screen[row][column] = 1
    }
}
```

问题 7.5 假设我们有如下数组： `items := [...]int{10, 20, 30, 40, 50}`

a) 如果我们写了如下的 for 循环，那么执行完 for 循环后的 `items` 的值是多少？如果你不确定的话可以测试一下:)

```
for _, item := range items {
    item *= 2
}
```

b) 如果 a) 无法正常工作，写一个 for 循环让值可以 double。

问题 7.6 通过使用省略号操作符 `...` 来实现累加方法。

练习 7.7 sum_array.go

a) 写一个 Sum 函数，传入参数为一个 32 位 float 数组成的数组 arrF，返回该数组的所有数字和。

如果把数组修改为切片的话代码要做怎样的修改？如果用切片形式方法实现不同长度数组的的和呢？

b) 写一个 SumAndAverage 方法，返回两个 int 和 float32 类型的未命名变量的和与平均值。

练习 7.8 min_max.go

写一个 minSlice 方法，传入一个 int 的切片并且返回最小值，再写一个 maxSlice 方法返回最大值。

切片重组（reslice）

7.4 切片重组（reslice）

我们已经知道切片创建的时候通常比相关数组小，例如：

```
slice1 := make([]type, start_length, capacity)
```

其中 `start_length` 作为切片初始长度而 `capacity` 作为相关数组的长度。

这么做的好处是我们的切片在达到容量上限后可以扩容。改变切片长度的过程称之为切片重组 **reslicing**，做法如下： `slice1 = slice1[0:end]` ，其中 end 是新的末尾索引（即长度）。

将切片扩展 1 位可以这么做：

```
sl = sl[0:len(sl)+1]
```

切片可以反复扩展直到占据整个相关数组。

示例 7.11 reslicing.go

```
package main
import "fmt"

func main() {
    slice1 := make([]int, 0, 10)
    // load the slice, cap(slice1) is 10:
    for i := 0; i < cap(slice1); i++ {
        slice1 = slice1[0:i+1]
        slice1[i] = i
        fmt.Printf("The length of slice is %d\n", len(slice1))
    }

    // print the slice:
    for i := 0; i < len(slice1); i++ {
        fmt.Printf("Slice at %d is %d\n", i, slice1[i])
    }
}
```

输出结果：

```
The length of slice is 1
The length of slice is 2
The length of slice is 3
The length of slice is 4
The length of slice is 5
The length of slice is 6
The length of slice is 7
The length of slice is 8
The length of slice is 9
The length of slice is 10
Slice at 0 is 0
Slice at 1 is 1
```

```
Slice at 2 is 2
Slice at 3 is 3
Slice at 4 is 4
Slice at 5 is 5
Slice at 6 is 6
Slice at 7 is 7
Slice at 8 is 8
Slice at 9 is 9
```

另一个例子：

```
var ar = [10]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
var a = ar[5:7] // reference to subarray {5, 6} - len(a) is 2 and cap(a) is 5
```

将 a 重新分片：

```
a = a[0:4] // ref of subarray {5, 6, 7, 8} - len(a) is now 4 but cap(a) is still 5
```

问题 7.7

1) 如果 a 是一个切片，那么 `a[n:n]` 的长度是多少？

2) `a[n:n+1]` 的长度又是多少？

切片的复制与追加

7.5 切片的复制与追加

如果想增加切片的容量，我们必须创建一个新的更大的切片并把原分片的内容都拷贝过来。下面的代码描述了从拷贝切片的 copy 函数和向切片追加新元素的 append 函数。

示例 7.12 copy_append_slice.go

```
package main
import "fmt"

func main() {
    slFrom := []int{1, 2, 3}
    slTo := make([]int, 10)

    n := copy(slTo, slFrom)
    fmt.Println(slTo)
    fmt.Printf("Copied %d elements\n", n) // n == 3

    sl3 := []int{1, 2, 3}
    sl3 = append(sl3, 4, 5, 6)
    fmt.Println(sl3)
}
```

`func append(s[]T, x ...T) []T` 其中 append 方法将 0 个或多个具有相同类型 s 的元素追加到切片后面并且返回新的切片；追加的元素必须和原切片的元素同类型。如果 s 的容量不足以存储新增元素，append 会分配新的切片来保证已有切片元素和新增元素的存储。因此，返回的切片可能已经指向一个不同的相关数组了。append 方法总是返回成功，除非系统内存耗尽了。

如果你想将切片 y 追加到切片 x 后面，只要将第二个参数扩展成一个列表即可：`x = append(x, y...)` 。

注意： append 在大多数情况下很好用，但是如果你想完全掌控整个追加过程，你可以实现一个这样的 AppendByte 方法：

```
func AppendByte(slice []byte, data ...byte) []byte {
    m := len(slice)
    n := m + len(data)
    if n > cap(slice) { // if necessary, reallocate
        // allocate double what's needed, for future growth.
        newSlice := make([]byte, (n+1)*2)
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:n]
    copy(slice[m:n], data)
    return slice
}
```

`func copy(dst, src []T) int` copy 方法将类型为 T 的切片从源地址 src 拷贝到目标地址 dst，覆盖 dst 的相关元素，并且返回拷贝的元素个数。源地址和目标地址可能会有重叠。拷贝个数是 src 和 dst 的长度最小值。如果 src 是字符串那么元素类型就是 byte。如果你还想继续使用 src，在拷贝结束后执行 `src = dst` 。

练习 7.9

给定一个slice `s []int` 和一个 int 类型的因子factor，扩展 s 使其长度为 `len(s) * factor` 。

练习 7.10

用顺序函数过滤容器：s 是前 10 个整型的切片。构造一个函数 Filter，第一个参数是 s，第二个参数是一个 `fn func(int) bool` ，返回满足函数 fn 的元素切片。通过 fn 测试方法测试当整型值是偶数时的情况。

练习 7.11

写一个函数 InsertStringSlice 将切片插入到另一个切片的指定位置。

练习 7.12

写一个函数 RemoveStringSlice 将从 start 到 end 索引的元素从切片 中移除。

字符串、数组和切片的应用

7.6 字符串、数组和切片的应用

7.6.1 从字符串生成字节切片

假设 s 是一个字符串（本质上是一个字节数组），那么就可以直接通过 `c := []byte(s)` 来获取一个字节的切片 c。另外，您还可以通过 copy 函数来达到相同的目的：`copy(dst []byte, src string)`。

同样的，还可以使用 for-range 来获得每个元素（Listing 7.13—for_string.go）：

```
package main

import "fmt"

func main() {
    s := "\u00ff\u754c"
    for i, c := range s {
        fmt.Printf("%d:%c ", i, c)
    }
}
```

输出：

```
0:ÿ 2:界
```

我们知道，Unicode 字符会占用 2 个字节，有些甚至需要 3 个或者 4 个字节来进行表示。如果发现错误的 UTF8 字符，则该字符会被设置为 U+FFFD 并且索引向前移动一个字节。和字符串转换一样，您同样可以使用 `c := []int32(s)` 语法，这样切片中的每个 int 都会包含对应的 Unicode 代码，因为字符串中的每次字符都会对应一个整数。类似的，您也可以将字符串转换为元素类型为 rune 的切片：`r := []rune(s)`。

可以通过代码 `len([]int32(s))` 来获得字符串中字符的数量，但使用 `utf8.RuneCountInString(s)` 效率会更高一点。(参考count_characters.go)

您还可以将一个字符串追加到某一个字节切片的尾部：

```
var b []byte
var s string
b = append(b, s...)
```

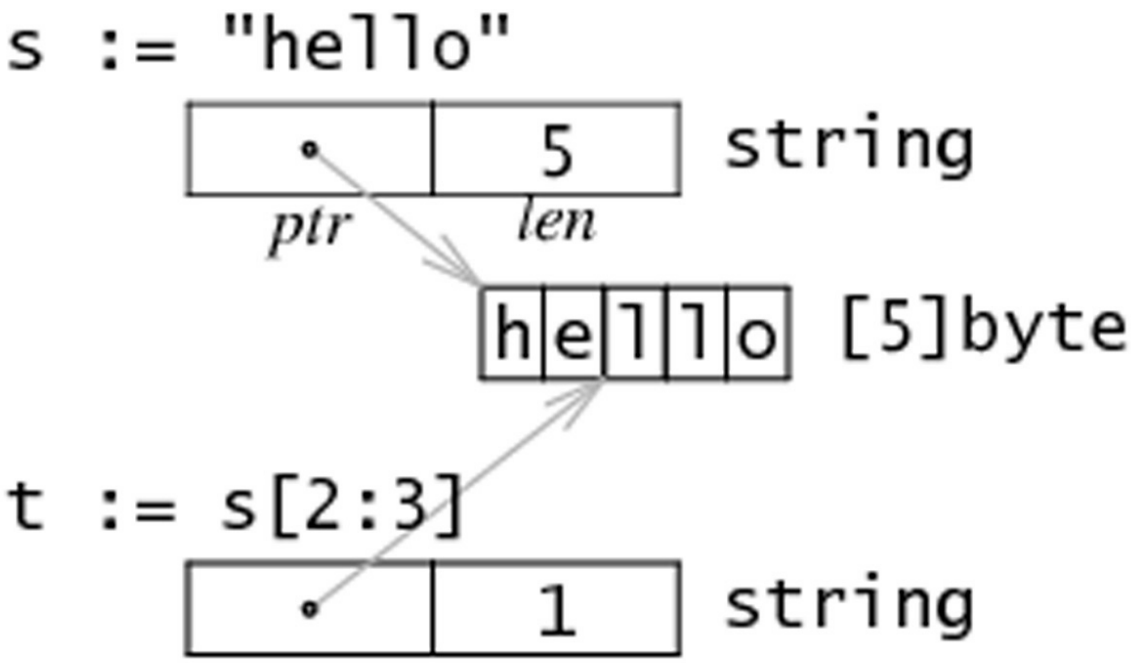
7.6.2 获取字符串的某一部分

使用 `substr := str[start:end]` 可以从字符串 str 获取到从索引 start 开始到 `end-1` 位置的子字符串。同样的，`str[start:]` 则表示获取从 start 开始到 `len(str)-1` 位置的子字符串。而 `str[:end]` 表示获取从 0 开始到 `end-1` 的子字符串。

7.6.3 字符串和切片的内存结构

在内存中，一个字符串实际上是一个双字结构，即一个指向实际数据的指针和记录字符串长度的整数（见图 7.4）。因为指针对用户来说是完全不可见，因此我们可以依旧把字符串看做是一个值类型，也就是一个字符数组。

字符串 `string s = "hello"` 和子字符串 `t = s[2:3]` 在内存中的结构可以用下图表示：



7.6.4 修改字符串中的某个字符

Go 语言中的字符串是不可变的，也就是说 `str[index]` 这样的表达式是不可以被放在等号左侧的。如果尝试运行 `str[i] = 'D'` 会得到错误：`cannot assign to str[i]` 。

因此，您必须先将字符串转换成字节数组，然后再通过修改数组中的元素值来达到修改字符串的目的，最后将字节数组转换回字符串格式。

例如，将字符串 “hello” 转换为 “cello”：

```
s := "hello"
c := []byte(s)
c[0] = 'c'
s2 := string(c) // s2 == "cello"
```

所以，您可以通过操作切片来完成对字符串的操作。

7.6.5 字节数组对比函数

下面的 `Compare` 函数会返回两个字节数组字典顺序的整数对比结果，即 `0 if a == b, -1 if a < b, 1 if a > b` 。

```
func Compare(a, b[]byte) int {
    for i:=0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
```

```
    }
    // 数组的长度可能不同
    switch {
    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0 // 数组相等
}
```

7.6.6 搜索及排序切片和数组

标准库提供了 `sort` 包来实现常见的搜索和排序操作。您可以使用 `sort` 包中的函数 `func Ints(a []int)` 来实现对 int 类型的切片排序。例如 `sort.Ints(arri)` ，其中变量 arri 就是需要被升序排序的数组或切片。为了检查某个数组是否已经被排序，可以通过函数 `IntsAreSorted(a []int) bool` 来检查，如果返回 true 则表示已经被排序。

类似的，可以使用函数 `func Float64s(a []float64)` 来排序 float64 的元素，或使用函数 `func Strings(a []string)` 排序字符串元素。

想要在数组或切片中搜索一个元素，该数组或切片必须先被排序（因为标准库的搜索算法使用的是二分法）。然后，您就可以使用函数 `func SearchInts(a []int, n int) int` 进行搜索，并返回对应结果的索引值。

当然，还可以搜索 float64 和字符串：

```
func SearchFloat64s(a []float64, x float64) int
func SearchStrings(a []string, x string) int
```

您可以通过查看 官方文档 来获取更详细的信息。

这就是如何使用 `sort` 包的方法，我们会在第 11.6 节对它的细节进行深入，并实现一个属于我们自己的版本。

7.6.7 append 函数常见操作

我们在第 7.5 节提到的 append 非常有用，它能够用于各种方面的操作：

1. 将切片 b 的元素追加到切片 a 之后： `a = append(a, b...)`
2. 复制切片 a 的元素到新的切片 b 上：

```
b = make([]T, len(a))
copy(b, a)
```

3. 删除位于索引 i 的元素： `a = append(a[:i], a[i+1:]...)`
4. 切除切片 a 中从索引 i 至 j 位置的元素： `a = append(a[:i], a[j:]...)`
5. 为切片 a 扩展 j 个元素长度： `a = append(a, make([]T, j)...)`
6. 在索引 i 的位置插入元素 x： `a = append(a[:i], append([]T{x}, a[i:]...)...)`
7. 在索引 i 的位置插入长度为 j 的新切片： `a = append(a[:i], append(make([]T, j), a[i:]...)...)`
8. 在索引 i 的位置插入切片 b 的所有元素： `a = append(a[:i], append(b, a[i:]...)...)`
9. 取出位于切片 a 最末尾的元素 x： `x, a = a[len(a)-1], a[:len(a)-1]`

10. 将元素 x 追加到切片 a： `a = append(a, x)`

因此，您可以使用切片和 append 操作来表示任意可变长度的序列。

从数学的角度来看，切片相当于向量，如果需要的话可以定义一个向量作为切片的别名来进行操作。

如果您需要更加完整的方案，可以学习一下 Eleanor McHugh 编写的几个包：slices、chain 和 lists。

7.6.8 切片和垃圾回收

切片的底层指向一个数组，该数组的实际容量可能要大于切片所定义的容量。只有在没有任何切片指向的时候，底层的数组内存才会被释放，这种特性有时会导致程序占用多余的内存。

示例 函数 `FindDigits` 将一个文件加载到内存，然后搜索其中所有的数字并返回一个切片。

```
var digitRegexp = regexp.MustCompile("[0-9]+")

func FindDigits(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return digitRegexp.Find(b)
}
```

这段代码可以顺利运行，但返回的 `[]byte` 指向的底层是整个文件的数据。只要该返回的切片不被释放，垃圾回收器就不能释放整个文件所占用的内存。换句话说，一点点有用的数据却占用了整个文件的内存。

想要避免这个问题，可以通过拷贝我们需要的部分到一个新的切片中：

```
func FindDigits(filename string) []byte {
   b, _ := ioutil.ReadFile(filename)
   b = digitRegexp.Find(b)
   c := make([]byte, len(b))
   copy(c, b)
   return c
}
```

事实上，上面这段代码只能找到第一个匹配正则表达式的数字串。要想找到所有的数字，可以尝试下面这段代码：

```
func FindFileDigits(filename string) []byte {
   fileBytes, _ := ioutil.ReadFile(filename)
   b := digitRegexp.FindAll(fileBytes, len(fileBytes))
   c := make([]byte, 0)
   for _, bytes := range b {
      c = append(c, bytes...)
   }
   return c
}
```

练习 7.12

编写一个函数，要求其接受两个参数，原始字符串 str 和分割索引 i，然后返回两个分割后的字符串。

练习 7.13

假设有字符串 str，那么 `str[len(str)/2:] + str[:len(str)/2]` 的结果是什么？

练习 7.14

编写一个程序，要求能够反转字符串，即将 “Google” 转换成 “elgooG”（提示：使用 `[]byte` 类型的切片）。

如果您使用两个切片来实现反转，请再尝试使用一个切片（提示：使用交换法）。

如果您想要反转 Unicode 编码的字符串，请使用 `[]int32` 类型的切片。

练习 7.15

编写一个程序，要求能够遍历一个字符数组，并将当前字符和前一个字符不相同的字符拷贝至另一个数组。

练习 7.16

编写一个程序，使用冒泡排序的方法排序一个包含整数的切片（算法的定义可参考 维基百科）。

练习 7.17

在函数式编程语言中，一个 map-function 是指能够接受一个函数原型和一个列表，并使用列表中的值依次执行函数原型，公式为：`map ( F(), (e1,e2, . . . ,en) ) = ( F(e1), F(e2), ... F(en) )` 。

编写一个函数 `mapFunc` 要求接受以下 2 个参数：

- 一个将整数乘以 10 的函数
- 一个整数列表

最后返回保存运行结果的整数列表。

Map

8.0 Map

map 是一种特殊的数据结构：一种元素对（pair）的无序集合，pair 的一个元素是 key，对应的另一个元素是 value，所以这个结构也称为关联数组或字典。这是一种快速寻找值的理想结构：给定 key，对应的 value 可以迅速定位。

map 这种数据结构在其他编程语言中也称为字典（Python）、hash 和 HashTable 等。

声明、初始化和 make

8.1 声明、初始化和 make

8.1.1 概念

map 是引用类型，可以使用如下声明：

```
var map1 map[keytype]valuetype
var map1 map[string]int
```

（ `[keytype]` 和 `valuetype` 之间允许有空格，但是 gofmt 移除了空格）

在声明的时候不需要知道 map 的长度，map 是可以动态增长的。

未初始化的 map 的值是 nil。

key 可以是任意可以用 == 或者 != 操作符比较的类型，比如 string、int、float。所以数组、切片和结构体不能作为 key (译者注：含有数组切片的结构体不能作为 key，只包含内建类型的 struct 是可以作为 key 的），但是指针和接口类型可以。如果要用结构体作为 key 可以提供 `Key()` 和 `Hash()` 方法，这样可以通过结构体的域计算出唯一的数字或者字符串的 key。

value 可以是任意类型的；通过使用空接口类型（详见第 11.9 节），我们可以存储任意值，但是使用这种类型作为值时需要先做一次类型断言（详见第 11.3 节）。

map 传递给函数的代价很小：在 32 位机器上占 4 个字节，64 位机器上占 8 个字节，无论实际上存储了多少数据。通过 key 在 map 中寻找值是很快的，比线性查找快得多，但是仍然比从数组和切片的索引中直接读取要慢 100 倍；所以如果你很在乎性能的话还是建议用切片来解决问题。

map 也可以用函数作为自己的值，这样就可以用来做分支结构（详见第 5 章）：key 用来选择要执行的函数。

如果 key1 是 map1 的key，那么 `map1[key1]` 就是对应 key1 的值，就如同数组索引符号一样（数组可以视为一种简单形式的 map，key 是从 0 开始的整数）。

key1 对应的值可以通过赋值符号来设置为 val1： `map1[key1] = val1` 。

令 `v := map1[key1]` 可以将 key1 对应的值赋值给 v；如果 map 中没有 key1 存在，那么 v 将被赋值为 map1 的值类型的空值。

常用的 `len(map1)` 方法可以获得 map 中的 pair 数目，这个数目是可以伸缩的，因为 map-pairs 在运行时可以动态添加和删除。

示例 8.1 make_maps.go

```
package main
import "fmt"

func main() {
    var mapLit map[string]int
    //var mapCreated map[string]float32
    var mapAssigned map[string]int

    mapLit = map[string]int{"one": 1, "two": 2}
```

```
    mapCreated := make(map[string]float32)
    mapAssigned = mapLit

    mapCreated["key1"] = 4.5
    mapCreated["key2"] = 3.14159
    mapAssigned["two"] = 3

    fmt.Printf("Map literal at \"one\" is: %d\n", mapLit["one"])
    fmt.Printf("Map created at \"key2\" is: %f\n", mapCreated["key2"])
    fmt.Printf("Map assigned at \"two\" is: %d\n", mapLit["two"])
    fmt.Printf("Map literal at \"ten\" is: %d\n", mapLit["ten"])
}
```

输出结果：

```
Map literal at "one" is: 1
Map created at "key2" is: 3.141590
Map assigned at "two" is: 3
Mpa literal at "ten" is: 0
```

mapLit 说明了 `map literals` 的使用方法： map 可以用 `{key1: val1, key2: val2}` 的描述方法来初始化，就像数组和结构体一样。

map 是 **引用类型** 的： 内存用 make 方法来分配。

map 的初始化： `var map1 = make(map[keytype]valuetype)` 。

或者简写为： `map1 := make(map[keytype]valuetype)` 。

上面例子中的 mapCreated 就是用这种方式创建的： `mapCreated := make(map[string]float32)` 。

相当于： `mapCreated := map[string]float32{}` 。

mapAssigned 也是 mapList 的引用，对 mapAssigned 的修改也会影响到 mapLit 的值。

不要使用 new，永远用 make 来构造 map

注意 如果你错误的使用 new() 分配了一个引用对象，你会获得一个空引用的指针，相当于声明了一个未初始化的变量并且取了它的地址：

```
mapCreated := new(map[string]float32)
```

接下来当我们调用： `mapCreated["key1"] = 4.5` 的时候，编译器会报错：

```
invalid operation: mapCreated["key1"] (index of type *map[string]float32).
```

为了说明值可以是任意类型的，这里给出了一个使用 `func() int` 作为值的 map：

示例 8.2 map_func.go

```
package main
import "fmt"

func main() {
    mf := map[int]func() int{
        1: func() int { return 10 },
```

```
        2: func() int { return 20 },
        5: func() int { return 50 },
    }
    fmt.Println(mf)
}
```

输出结果为： `map[1:0x10903be0 5:0x10903ba0 2:0x10903bc0]` ：整形都被映射到函数地址。

8.1.2 map 容量

和数组不同，map 可以根据新增的 key-value 对动态的伸缩，因此它不存在固定长度或者最大限制。但是你也可以选择标明 map 的初始容量 `capacity` ，就像这样： `make(map[keytype]valuetype, cap)` 。例如：

```
map2 := make(map[string]float32, 100)
```

当 map 增长到容量上限的时候，如果再增加新的 key-value 对，map 的大小会自动加 1。所以出于性能的考虑，对于大的 map 或者会快速扩张的 map，即使只是大概知道容量，也最好先标明。

这里有一个 map 的具体例子，即将音阶和对应的音频映射起来：

```
noteFrequency := map[string]float32 {
    "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
    "G0": 24.50, "A0": 27.50, "B0": 30.87, "A4": 440}
```

8.1.3 用切片作为 map 的值

既然一个 key 只能对应一个 value，而 value 又是一个原始类型，那么如果一个 key 要对应多个值怎么办？例如，当我们要处理unix机器上的所有进程，以父进程（pid 为整形）作为 key，所有的子进程（以所有子进程的 pid 组成的切片）作为 value。通过将 value 定义为 `[]int` 类型或者其他类型的切片，就可以优雅的解决这个问题。

这里有一些定义这种 map 的例子：

```
mp1 := make(map[int][]int)
mp2 := make(map[int]*[]int)
```

测试键值对是否存在及删除元素

8.2 测试键值对是否存在及删除元素

测试 map1 中是否存在 key1：

在例子 8.1 中，我们已经见过可以使用 `val1 = map1[key1]` 的方法获取 key1 对应的值 val1。如果 map 中不存在 key1，val1 就是一个值类型的空值。

这就会给我们带来困惑了：现在我们没法区分到底是 key1 不存在还是它对应的 value 就是空值。

为了解决这个问题，我们可以这么用： `val1, isPresent = map1[key1]`

isPresent 返回一个 bool 值：如果 key1 存在于 map1，val1 就是 key1 对应的 value 值，并且 isPresent为true；如果 key1 不存在，val1 就是一个空值，并且 isPresent 会返回 false。

如果你只是想判断某个 key 是否存在而不关心它对应的值到底是多少，你可以这么做：

```
_, ok := map1[key1] // 如果key1存在则ok == true，否则ok为false
```

或者和 if 混合使用：

```
if _, ok := map1[key1]; ok {
    // ...
}
```

从 map1 中删除 key1：

直接 `delete(map1, key1)` 就可以。

如果 key1 不存在，该操作不会产生错误。

示例 8.4 map_testelement.go

```
package main
import "fmt"

func main() {
    var value int
    var isPresent bool

    map1 := make(map[string]int)
    map1["New Delhi"] = 55
    map1["Beijing"] = 20
    map1["Washington"] = 25
    value, isPresent = map1["Beijing"]
    if isPresent {
        fmt.Printf("The value of \"Beijing\" in map1 is: %d\n", value)
    } else {
        fmt.Printf("map1 does not contain Beijing")
    }

    value, isPresent = map1["Paris"]
```

```
    fmt.Printf("Is \"Paris\" in mapl ?: %t\n", isPresent)
    fmt.Printf("Value is: %d\n", value)

    // delete an item:
    delete(mapl, "Washington")
    value, isPresent = mapl["Washington"]
    if isPresent {
        fmt.Printf("The value of \"Washington\" in mapl is: %d\n", value)
    } else {
        fmt.Println("mapl does not contain Washington")
    }
}
```

输出结果：

```
The value of "Beijing" in mapl is: 20
Is "Paris" in mapl ?: false
Value is: 0
mapl does not contain Washington
```

for-range 的配套用法

8.3 for-range 的配套用法

可以使用 for 循环构造 map：

```
for key, value := range map1 {
    ...
}
```

第一个返回值 key 是 map 中的 key 值，第二个返回值则是该 key 对应的 value 值；这两个都是仅 for 循环内部可见的局部变量。其中第一个返回值key值是一个可选元素。如果你只关心值，可以这么使用：

```
for _, value := range map1 {
    ...
}
```

如果只想获取 key，你可以这么使用：

```
for key := range map1 {
    fmt.Printf("key is: %d\n", key)
}
```

示例 8.5 maps_forrange.go：

```
package main
import "fmt"

func main() {
    map1 := make(map[int]float32)
    map1[1] = 1.0
    map1[2] = 2.0
    map1[3] = 3.0
    map1[4] = 4.0
    for key, value := range map1 {
        fmt.Printf("key is: %d - value is: %f\n", key, value)
    }
}
```

输出结果：

```
key is: 3 - value is: 3.000000
key is: 1 - value is: 1.000000
key is: 4 - value is: 4.000000
key is: 2 - value is: 2.000000
```

注意 map 不是按照 key 的顺序排列的，也不是按照 value 的序排列的。

问题 8.1： 下面这段代码的输出是什么？

```
capitals := map[string] string {"France":"Paris", "Italy":"Rome", "Japan":"Tokyo" }
for key := range capitals {
    fmt.Println("Map item: Capital of", key, "is", capitals[key])
}
```

练习 8.1

创建一个 map 来保存每周 7 天的名字，将它们打印出来并且测试是否存在 Tuesday 和 Hollyday。

map 类型的切片

8.4 map 类型的切片

假设我们想获取一个 map 类型的切片，我们必须使用两次 `make()` 函数，第一次分配切片，第二次分配 切片中每个 map 元素（参见下面的例子 8.4）。

示例 8.4 maps_forrange2.go：

```
package main
import "fmt"

func main() {
    // Version A:
    items := make([]map[int]int, 5)
    for i:= range items {
        items[i] = make(map[int]int, 1)
        items[i][1] = 2
    }
    fmt.Printf("Version A: Value of items: %v\n", items)

    // Version B: NOT GOOD!
    items2 := make([]map[int]int, 5)
    for _, item := range items2 {
        item = make(map[int]int, 1) // item is only a copy of the slice element.
        item[1] = 2 // This 'item' will be lost on the next iteration.
    }
    fmt.Printf("Version B: Value of items: %v\n", items2)
}
```

输出结果：

```
Version A: Value of items: [map[1:2] map[1:2] map[1:2] map[1:2] map[1:2]]
Version B: Value of items: [map[] map[] map[] map[] map[]]
```

需要注意的是，应当像 A 版本那样通过索引使用切片的 map 元素。在 B 版本中获得的项只是 map 值的一个拷贝而已，所以真正的 map 元素没有得到初始化。

map 的排序

8.5 map 的排序

map 默认是无序的，不管是按照 key 还是按照 value 默认都不排序（详见第 8.3 节）。

如果你想为 map 排序，需要将 key（或者 value）拷贝到一个切片，再对切片排序（使用 sort 包，详见第 7.6.6 节），然后可以使用切片的 for-range 方法打印出所有的 key 和 value。

下面有一个示例：

示例 8.6 sort_map.go：

```
// the telephone alphabet:
package main
import (
    "fmt"
    "sort"
)

var (
    barVal = map[string]int{"alpha": 34, "bravo": 56, "charlie": 23,
                            "delta": 87, "echo": 56, "foxtrot": 12,
                            "golf": 34, "hotel": 16, "indio": 87,
                            "juliet": 65, "kili": 43, "lima": 98}
)

func main() {
    fmt.Println("unsorted:")
    for k, v := range barVal {
        fmt.Printf("Key: %v, Value: %v / ", k, v)
    }
    keys := make([]string, len(barVal))
    i := 0
    for k, _ := range barVal {
        keys[i] = k
        i++
    }
    sort.Strings(keys)
    fmt.Println()
    fmt.Println("sorted:")
    for _, k := range keys {
        fmt.Printf("Key: %v, Value: %v / ", k, barVal[k])
    }
}
```

输出结果：

```
unsorted:
Key: bravo, Value: 56 / Key: echo, Value: 56 / Key: indio, Value: 87 / Key: juliet, Value: 65 / Key: alpha, V
alue: 34 / Key: charlie, Value: 23 / Key: delta, Value: 87 / Key: foxtrot, Value: 12 / Key: golf, Value: 34 /
Key: hotel, Value: 16 / Key: kili, Value: 43 / Key: lima, Value: 98 /
sorted:
Key: alpha, Value: 34 / Key: bravo, Value: 56 / Key: charlie, Value: 23 / Key: delta, Value: 87 / Key: echo,
```

```
 Value: 56 / Key: foxtrot, Value: 12 / Key: golf, Value: 34 / Key: hotel, Value: 16 / Key: indio, Value: 87 /
Key: juliet, Value: 65 / Key: kili, Value: 43 / Key: lima, Value: 98 /
```

但是如果你想要一个排序的列表你最好使用结构体切片，这样会更有效：

```
type name struct {
    key string
    value int
}
```

将 map 的键值对调

8.6 将 map 的键值对调

这里对调是指调换 key 和 value。如果 map 的值类型可以作为 key 且所有的 value 是唯一的，那么通过下面的方法可以简单的做到键值对调。

示例 8.7 invert_map.go：

```
package main
import (
    "fmt"
)

var (
    barVal = map[string]int{"alpha": 34, "bravo": 56, "charlie": 23,
                            "delta": 87, "echo": 56, "foxtrot": 12,
                            "golf": 34, "hotel": 16, "indio": 87,
                            "juliet": 65, "kili": 43, "lima": 98}
)

func main() {
    invMap := make(map[int]string, len(barVal))
    for k, v := range barVal {
        invMap[v] = k
    }
    fmt.Println("inverted:")
    for k, v := range invMap {
        fmt.Printf("Key: %v, Value: %v / ", k, v)
    }
}
```

输出结果：

```
inverted:
Key: 34, Value: golf / Key: 23, Value: charlie / Key: 16, Value: hotel / Key: 87, Value: delta / Key: 98, Val
ue: lima / Key: 12, Value: foxtrot / Key: 43, Value: kili / Key: 56, Value: bravo / Key: 65, Value: juliet /
```

如果原始 value 值不唯一那这么做肯定会出问题；这种情况下不会报错，但是当遇到不唯一的 key 时应当直接停止对调，且此时对调后的 map 很可能没有包含原 map 的所有键值对！一种解决方法就是仔细检查唯一性并且使用多值 map，比如使用 `map[int][]string` 类型。

练习 8.2

构造一个将英文饮料名映射为法语（或者任意你的母语）的集合；先打印所有的饮料，然后打印原名和翻译后的名字。接下来按照英文名排序后再打印出来。

包（package）

9.0 包（package）

本章主要针对 Go 语言的包展开讲解。

标准库概述

9.1 标准库概述

像 `fmt` 、 `os` 等这样具有常用功能的内置包在 Go 语言中有 150 个以上，它们被称为标准库，大部分(一些底层的除外)内置于 Go 本身。完整列表可以在 Go Walker 查看。

在贯穿本书的例子和练习中，我们都是用标准库的包。可以通过查阅第 350 页包中的内容快速找到相关的包的实例。这里我们只是按功能进行分组来介绍这些包的简单用途，我们不会深入讨论他们的内部结构。

- `unsafe`: 包含了一些打破 Go 语言“类型安全”的命令，一般的程序中不会被使用，可用在 C/C++ 程序的调用中。
- `syscall` - `os` - `os/exec`:
  - `os`: 提供给我们一个平台无关性的操作系统功能接口，采用类Unix设计，隐藏了不同操作系统间差异，让不同的文件系统和操作系统对象表现一致。
  - `os/exec`: 提供我们运行外部操作系统命令和程序的方式。
  - `syscall`: 底层的外部包，提供了操作系统底层调用的基本接口。

通过一个 Go 程序让Linux重启来体现它的能力。

示例 9.1 reboot.go：

```
package main
import (
    "syscall"
)

const LINUX_REBOOT_MAGIC1 uintptr = 0xfee1dead
const LINUX_REBOOT_MAGIC2 uintptr = 672274793
const LINUX_REBOOT_CMD_RESTART uintptr = 0x1234567

func main() {
    syscall.Syscall(syscall.SYS_REBOOT,
        LINUX_REBOOT_MAGIC1,
        LINUX_REBOOT_MAGIC2,
        LINUX_REBOOT_CMD_RESTART)
}
```

- `archive/tar` 和 `/zip-compress`: 压缩(解压缩)文件功能。
- `fmt` - `io` - `bufio` - `path/filepath` - `flag`:
  - `fmt`: 提供了格式化输入输出功能。
  - `io`: 提供了基本输入输出功能，大多数是围绕系统功能的封装。
  - `bufio`: 缓冲输入输出功能的封装。
  - `path/filepath`: 用来操作在当前系统中的目标文件名路径。
  - `flag`: 对命令行参数的操作。
- `strings` - `strconv` - `unicode` - `regexp` - `bytes`:
  - `strings`: 提供对字符串的操作。
  - `strconv`: 提供将字符串转换为基础类型的功能。
  - `unicode`: 为 unicode 型的字符串提供特殊的功能。

    - `regexp`：正则表达式功能。
    - `bytes`：提供对字符型分片的操作。
    - `index/suffixarray`：子字符串快速查询。
- `math`、`math/cmath`、`math/big`、`math/rand`、`sort`：
    - `math`：基本的数学函数。
    - `math/cmath`：对复数的操作。
    - `math/rand`：伪随机数生成。
    - `sort`：为数组排序和自定义集合。
    - `math/big`：大数的实现和计算。
- `container`、`/list-ring-heap`：实现对集合的操作。
    - `list`：双链表。
    - `ring`：环形链表。

下面代码演示了如何遍历一个链表(当 l 是 `*List`)：

```
for e := l.Front(); e != nil; e = e.Next() {
    //do something with e.Value
}
```

- `time`、`log`：
    - `time`：日期和时间的基本操作。
    - `log`：记录程序运行时产生的日志,我们将在后面的章节使用它。
- `encoding/json`、`encoding/xml`、`text/template`：
    - `encoding/json`：读取并解码和写入并编码 JSON 数据。
    - `encoding/xml`：简单的 XML1.0 解析器,有关 JSON 和 XML 的实例请查阅第 12.9/10 章节。
    - `text/template`：生成像 HTML 一样的数据与文本混合的数据驱动模板（参见第 15.7 节）。
- `net`、`net/http`、`html`：（参见第 15 章）
    - `net`：网络数据的基本操作。
    - `http`：提供了一个可扩展的 HTTP 服务器和基础客户端，解析 HTTP 请求和回复。
    - `html`：HTML5 解析器。
- `runtime`：Go 程序运行时的交互操作，例如垃圾回收和协程创建。
- `reflect`：实现通过程序运行时反射，让程序操作任意类型的变量。

`exp` 包中有许多将被编译为新包的实验性的包。在下次稳定版本发布的时候，它们将成为独立的包。如果前一个版本已经存在了，它们将被作为过时的包被回收。然而 Go1.0 发布的时候并没有包含过时或者实验性的包。

练习 9.1

使用 `container/list` 包实现一个双向链表，将 101、102 和 103 放入其中并打印出来。

练习 9.2

通过使用 `unsafe` 包中的方法来测试你电脑上一个整型变量占用多少个字节。

regexp 包

9.2 regexp 包

正则表达式语法和使用的详细信息请参考 维基百科。

在下面的程序里，我们将在字符串中对正则表达式模式（pattern）进行匹配。

如果是简单模式，使用 `Match` 方法便可：

```
ok, _ := regexp.Match(pat, []byte(searchIn))
```

变量 ok 将返回 true 或者 false,我们也可以使用 `MatchString` ：

```
ok, _ := regexp.MatchString(pat, searchIn)
```

更多方法中，必须先将正则模式通过 `Compile` 方法返回一个 Regexp 对象。然后我们将掌握一些匹配，查找，替换相关的功能。

示例 9.2 pattern.go：

```
package main
import (
    "fmt"
    "regexp"
    "strconv"
)
func main() {
    //目标字符串
    searchIn := "John: 2578.34 William: 4567.23 Steve: 5632.18"
    pat := "[0-9]+.[0-9]+" //正则

    f := func(s string) string{
        v, _ := strconv.ParseFloat(s, 32)
        return strconv.FormatFloat(v * 2, 'f', 2, 32)
    }

    if ok, _ := regexp.Match(pat, []byte(searchIn)); ok {
    fmt.Println("Match Found!")
    }

    re, _ := regexp.Compile(pat)
    //将匹配到的部分替换为"##.#"
    str := re.ReplaceAllString(searchIn, "##.#")
    fmt.Println(str)
    //参数为函数时
    str2 := re.ReplaceAllStringFunc(searchIn, f)
    fmt.Println(str2)
}
```

输出结果：

```
Match Found!
John: ##.# William: ##.# Steve: ##.#
John: 5156.68 William: 9134.46 Steve: 11264.36
```

`Compile` 函数也可能返回一个错误，我们在使用时忽略对错误的判断是因为我们确信自己正则表达式是有效的。当用户输入或从数据中获取正则表达式的时候，我们有必要去检验它的正确性。另外我们也可以使用 `MustCompile` 方法，它可以像 `Compile` 方法一样检验正则的有效性，但是当正则不合法时程序将 panic（详情查看第 13.2 节)。

锁和 sync 包

9.3 锁和 sync 包

在一些复杂的程序中，通常通过不同线程执行不同应用来实现程序的并发。当不同线程要使用同一个变量时，经常会出现一个问题：无法预知变量被不同线程修改的顺序！(这通常被称为资源竞争,指不同线程对同一变量使用的竞争)显然这无法让人容忍，那我们该如何解决这个问题呢？

经典的做法是一次只能让一个线程对共享变量进行操作。当变量被一个线程改变时(临界区)，我们为它上锁，直到这个线程执行完成并解锁后，其他线程才能访问它。

特别是我们之前章节学习的 map 类型是不存在锁的机制来实现这种效果(出于对性能的考虑)，所以 map 类型是非线程安全的。当并行访问一个共享的 map 类型的数据，map 数据将会出错。

在 Go 语言中这种锁的机制是通过 sync 包中 Mutex 来实现的。sync 来源于 “synchronized” 一词，这意味着线程将有序的对同一变量进行访问。

`sync.Mutex` 是一个互斥锁，它的作用是守护在临界区入口来确保同一时间只能有一个线程进入临界区。

假设 info 是一个需要上锁的放在共享内存中的变量。通过包含 `Mutex` 来实现的一个典型例子如下：

```
import  "sync"

type Info struct {
    mu sync.Mutex
    // ... other fields, e.g.: Str string
}
```

如果一个函数想要改变这个变量可以这样写:

```
func Update(info *Info) {
    info.mu.Lock()
    // critical section:
    info.Str = // new value
    // end critical section
    info.mu.Unlock()
}
```

还有一个很有用的例子是通过 Mutex 来实现一个可以上锁的共享缓冲器:

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

在 sync 包中还有一个 `RWMutex` 锁：他能通过 `RLock()` 来允许同一时间多个线程对变量进行读操作，但是只能一个线程进行写操作。如果使用 `Lock()` 将和普通的 `Mutex` 作用相同。包中还有一个方便的 `Once` 类型变量的方法 `once.Do(call)` ，这个方法确保被调用函数只能被调用一次。

相对简单的情况下，通过使用 sync 包可以解决同一时间只能一个线程访问变量或 map 类型数据的问题。如果这种方式导致程序明显变慢或者引起其他问题，我们要重新思考来通过 goroutines 和 channels 来解决问题，这是在 Go 语言中所提倡用来实现并发的技术。我们将在第 14 章对其深入了解，并在第 14.7 节中对这两种方式进行比较。

精密计算和 big 包

9.4 精密计算和 big 包

我们知道有些时候通过编程的方式去进行计算是不精确的。如果你使用 Go 语言中的 float64 类型进行浮点运算，返回结果将精确到 15 位，足以满足大多数的任务。当对超出 int64 或者 uint64 类型这样的大数进行计算时，如果对精度没有要求，float32 或者 float64 可以胜任，但如果对精度有严格要求的时候，我们不能使用浮点数，在内存中它们只能被近似的表示。

对于整数的高精度计算 Go 语言中提供了 big 包，被包含在 math 包下：有用来表示大整数的 `big.Int` 和表示大有理数的 `big.Rat` 类型（可以表示为 2/5 或 3.1416 这样的分数，而不是无理数或 π）。这些类型可以实现任意位类型的数字，只要内存足够大。缺点是更大的内存和处理开销使它们使用起来要比内置的数字类型慢很多。

大的整型数字是通过 `big.NewInt(n)` 来构造的，其中 n 为 int64 类型整数。而大有理数是通过 `big.NewRat(n, d)` 方法构造。n（分子）和 d（分母）都是 int64 型整数。因为 Go 语言不支持运算符重载，所以所有大数字类型都有像是 `Add()` 和 `Mul()` 这样的方法。它们作用于作为 receiver 的整数和有理数，大多数情况下它们修改 receiver 并以 receiver 作为返回结果。因为没有必要创建 `big.Int` 类型的临时变量来存放中间结果，所以运算可以被链式地调用，并节省内存。

示例 9.2 big.go：

```
// big.go
package main

import (
    "fmt"
    "math"
    "math/big"
)

func main() {
    // Here are some calculations with bigInts:
    im := big.NewInt(math.MaxInt64)
    in := im
    io := big.NewInt(1956)
    ip := big.NewInt(1)
    ip.Mul(im, in).Add(ip, im).Div(ip, io)
    fmt.Printf("Big Int: %v\n", ip)
    // Here are some calculations with bigInts:
    rm := big.NewRat(math.MaxInt64, 1956)
    rn := big.NewRat(-1956, math.MaxInt64)
    ro := big.NewRat(19, 56)
    rp := big.NewRat(1111, 2222)
    rq := big.NewRat(1, 1)
    rq.Mul(rm, rn).Add(rq, ro).Mul(rq, rp)
    fmt.Printf("Big Rat: %v\n", rq)
}

/* Output:
Big Int: 43492122561469640008497075573153004
Big Rat: -37/112
*/
```

输出结果：

```
Big Int: 434921225614696400084970755731530004
Big Rat: -37/112
```

自定义包和可见性

9.5 自定义包和可见性

包是 Go 语言中代码组织和代码编译的主要方式。关于它们的很多基本信息已经在 4.2 章节中给出，最引人注目的便是可见性。现在我们来看看具体如何来使用自己写的包。在下一节，我们将回顾一些标准库中的包，自定义的包和标准库以外的包。

当写自己包的时候，要使用短小的不含有 `_` (下划线)的小写单词来为文件命名。这里有个简单例子来说明包是如何相互调用以及可见性是如何实现的。

当前目录下（examples/chapter_9/book/）有一个名为 package_mytest.go 的程序，它使用了自定义包 pack1 中 pack1.go 的代码。这段程序(连同编译链接生成的 pack1.a)存放在当前目录下一个名为 pack1 的文件夹下。所以链接器将包的对象和主程序对象链接在一起。

示例 9.4 pack1.go：

```
package pack1
var Pack1Int int = 42
var pack1Float = 3.14

func ReturnStr() string {
    return "Hello main!"
}
```

它包含了一个整型变量 `Pack1Int` 和一个返回字符串的函数 `ReturnStr` 。这段程序在运行时不做任何的事情，因为它没有一个 main 函数。

在主程序 package_mytest.go 中这个包通过声明的方式被导入，只到包的目录一层。

```
import "./pack1"
```

import 的一般格式如下:

```
import "包的路径或 URL 地址"
```

例如：

```
import "github.com/org1/pack1"
```

路径是指当前目录的相对路径。

示例 9.5 package_mytest.go：

```
package main

import (
    "fmt"
    "./pack1"
)

func main() {
```

```
    var test1 string
    test1 = pack1.ReturnStr()
    fmt.Printf("ReturnStr from package1: %s\n", test1)
    fmt.Printf("Integer from package1: %d\n", pack1.Pack1Int)
    // fmt.Printf("Float from package1: %f\n", pack1.pack1Float)
}
```

输出结果：

```
ReturnStr from package1: Hello main!
Integer from package1: 42
```

如果包 pack1 和我们的程序在同一路径下，我们可以通过 `"import ./pack1"` 这样的方式来引入，但这不被视为一个好的方法。

下面的代码试图访问一个未引用的变量或者函数，甚至没有编译。将会返回一个错误：

```
fmt.Printf("Float from package1: %f\n", pack1.pack1Float)
```

错误：

```
cannot refer to unexported name pack1.pack1Float
```

主程序利用的包必须在主程序编写之前被编译。主程序中每个 pack1 项目都要通过包名来使用： `pack1.Item` 。具体使用方法请参见示例 4.6 和 4.7。

因此，按照惯例,子目录和包之间有着密切的联系：为了区分,不同包存放在不同的目录下，每个包(所有属于这个包中的 go 文件)都存放在和包名相同的子目录下：

Import with `.` ：

```
import . "./pack1"
```

当使用 `.` 来做为包的别名时，你可以不通过包名来使用其中的项目。例如： `test := ReturnStr()` 。

在当前的命名空间导入 pack1 包，一般是为了具有更好的测试效果。

Import with `_` ：

```
import _ "./pack1/pack1"
```

pack1包只导入其副作用，也就是说，只执行它的init函数并初始化其中的全局变量。

导入外部安装包：

如果你要在你的应用中使用一个或多个外部包，首先你必须使用 `go install` （参见第 9.7 节）在你的本地机器上安装它们。

假设你想使用 `http://codesite.ext/author/goExample/goex` 这种托管在 Google Code、GitHub 和 Launchpad 等代码网站上的包。

你可以通过如下命令安装：

```
go install codesite.ext/author/goExample/goex
```

将一个名为 `codesite.ext/author/goExample/goex` 的 map 安装在 `$GOROOT/src/` 目录下。

通过以下方式，一次性安装，并导入到你的代码中：

```
import goex "codesite.ext/author/goExample/goex"
```

因此该包的 URL 将用作导入路径。

在 `http://golang.org/cmd/goinstall/` 的 `go install` 文档中列出了一些广泛被使用的托管在网络代码仓库的包的导入路径

包的初始化:

程序的执行开始于导入包，初始化 main 包然后调用 main 函数。

一个没有导入的包将通过分配初始值给所有的包级变量和调用源码中定义的包级 init 函数来初始化。一个包可能有多个 init 函数甚至在一个源码文件中。它们的执行是无序的。这是最好的例子来测定包的值是否只依赖于相同包下的其他值或者函数。

init 函数是不能被调用的。

导入的包在包自身初始化前被初始化，而一个包在程序执行中只能初始化一次。

编译并安装一个包(参见第 9.7 节):

在 Linux/OS X 下可以用类似第 3.9 节的 Makefile 脚本做到这一点：

```
include $(GOROOT)/src/Make.inc
TARG=pack1
GOFILES=\
    pack1.go\
    pack1b.go\
include $(GOROOT)/src/Make.pkg
```

通过 `chmod 777 ./Makefile` 确保它的可执行性。

上面脚本内的include语引入了相应的功能，将自动检测机器的架构并调用正确的编译器和链接器。

然后终端执行 make 或 `gomake` 工具：他们都会生成一个包含静态库 pack1.a 的 _obj 目录。

go install(参见第 9.7 节，从 Go1 的首选方式)同样复制 pack1.a 到本地的 $GOROOT/pkg 的目录中一个以操作系统为名的子目录下。像 `import "pack1"` 代替 `import "path to pack1"` ，这样只通过名字就可以将包在程序中导入。

当第 13 章我们遇到使用测试工具进行测试的时候我们将重新回到自己的包的制作和编译这个话题。

问题 9.1

a）一个包能分成多个源文件么？

b）一个源文件是否能包含多个包？

练习 9.3

创建一个程序 main_greetings.go 能够和用户说 “Good Day” 或者 “Good Night”。不同的问候应该放到单独的 greetings 包中。

在同一个包中创建一个 `IsAM` 函数返回一个布尔值用来判断当前时间是 AM 还是 PM，同样创建 `IsAfternoon` 和 `IsEvening` 函数。

使用 main_greetings 作出合适的问候(提示：使用 time 包)。

练习 9.4 创建一个程序 main_oddven.go 判断前 100 个整数是不是偶数，将判断所用的函数编写在 even 包里。

练习 9.5 使用第 6.6 节的斐波那契程序：

1）将斐波那契功能放入自己的 fibo 包中并通过主程序调用它，存储最后输入的值在函数的全局变量。

2）扩展 fibo 包将通过调用斐波那契的时候，操作也作为一个参数。实验“+”和“*”

main_fibo.go / fibonacci.go

为自定义包使用 godoc

9.6 为自定义包使用 godoc

godoc工具（第 3.6 节）在显示自定义包中的注释也有很好的效果：注释必须以 `//` 开始并无空行放在声明（包，类型，函数）前。godoc 会为每个文件生成一系列的网页。

例如：

- 在 doc_examples 目录下我们有第 11.7 节中的用来排序的 go 文件，文件中有一些注释（文件需要未编译）
- 命令行下进入目录下并输入命令：

  godoc -http=:6060 -goroot=".."

（ `.` 是指当前目录，-goroot 参数可以是 `/path/to/my/package1` 这样的形式指出 package1 在你源码中的位置或接受用冒号形式分隔的路径，无根目录的路径为相对于当前目录的相对路径）

- 在浏览器打开地址：http://localhost:6060

然后你会看到本地的 godoc 页面（详见第 3.6 节）从左到右一次显示出目录中的包：

doc_example:

doc_example | Packages | Commands | Specification

下面是链接到源码和所有对象时有序概述（所以是很好的浏览和查找源代码的方式），连同文件/注释：

sort 包

```
func Float64sAreSorted

type IntArray

func IntsAreSortedfunc IsSortedfunc Sort

func (IntArray) Len

func SortFloat64s

func (IntArray) Less

func SortInts

func (IntArray) Swap

func SortStrings type Interface

func StringsAreSorted type StringArray type Float64Array

func (StringArray) Len

func (Float64Array) Len
```

```
func (StringArray) Less

func (Float64Array) Less

func (StringArray) Swap

func (Float64Array) Swap

// Other packages
import "doc_example"
```

使用通用的接口排序:

```
func Float64sAreSorted[Top]
func Float64sAreSorted(a []float64) bool

func IntsAreSorted[Top]
func IntsAreSorted(a []int) bool

func IsSorted[Top]
func IsSorted(data Interface) bool
Test if data is sorted

func Sort[Top]
func Sort(data Interface)
General sort function

func SortInts[Top]
func SortInts(a []int)

Convenience wrappers for common cases: type IntArray[Top]
Convenience types for common cases: IntArray type IntArray []int
```

如果你在一个团队中工作，并且源代码树被存储在网络硬盘上，就可以使用 godoc 给所有团队成员连续文档的支持。通过设置 `sync_minutes=n` ，你甚至可以让它每 n 分钟自动更新您的文档!

使用 go install 安装自定义包

9.7 使用 go install 安装自定义包

go install 是 Go 中自动包安装工具：如需要将包安装到本地它会从远端仓库下载包：检出、编译和安装一气呵成。

在包安装前的先决条件是要自动处理包自身依赖关系的安装。被依赖的包也会安装到子目录下，但是没有文档和示例：可以到网上浏览。

go install 使用了 GOPATH 变量(详见第 2.2 节)。

远端包(详见第 9.5 节)：

假设我们要安装一个有趣的包 tideland（它包含了许多帮助示例，参见 项目主页）。

因为我们需要创建目录在 Go 安装目录下，所以我们需要使用 root 或者 su 的身份执行命令。

确保 Go 环境变量已经设置在 root 用户下的 `./bashrc` 文件中。

使用命令安装： `go install tideland-cgl.googlecode.com/hg` 。

可执行文件 `hg.a` 将被放到 `$GOROOT/pkg/linux_amd64/tideland-cgl.googlecode.com` 目录下，源码文件被放置在 `$GOROOT/src/tideland-cgl.googlecode.com/hg` 目录下，同样有个 `hg.a` 放置在 `_obj` 的子目录下。

现在就可以在 go 代码中使用这个包中的功能了，例如使用包名 cgl 导入：

```
import cgl "tideland-cgl.googlecode.com/hg"
```

从 Go1 起 go install 安装 Google Code 的导入路径形式是： `"code.google.com/p/tideland-cgl"`

升级到新的版本：

更新到新版本的 Go 之后本地安装包的二进制文件将全被删除。如果你想更新，重编译、重安装所有的go安装包可以使用： `go install -a` 。

go 的版本发布的很频繁，所以需要注意发布版本和包的兼容性。go1 之后都是自己编译自己了。

go install 同样可以使用 go install 编译链接并安装本地自己的包（详见第 9.8.2 节）。

更多信息可以在 官方网站 找到。

自定义包的目录结构、go install 和 go test

9.8 自定义包的目录结构、go install 和 go test

为了示范，我们创建了一个名为 uc 的简单包，它含有一个 `UpperCase` 函数将字符串的所有字母转换为大写。当然这并不值得创建一个自定义包，同样的功能已被包含在 `strings` 包里，但是同样的技巧也可以应用在更复杂的包中。

9.8.1 自定义包的目录结构

下面的结构给了你一个好的示范(uc 代表通用包名，名字为粗体的代表目录，斜体代表可执行文件):

```
/home/user/goprograms
    ucmain.go     (uc包主程序)
    Makefile (ucmain的makefile)
    ucmain
    src/uc      (包含uc包的go源码)
        uc.go
         uc_test.go
         Makefile (包的makefile)
         uc.a
         _obj
            uc.a
        _test
            uc.a
    bin         (包含最终的执行文件)
        ucmain
    pkg/linux_amd64
        uc.a     (包的目标文件)
```

将你的项目放在 goprograms 目录下(你可以创建一个环境变量 GOPATH，详见第 2.2/3 章节：在 .profile 和 .bashrc 文件中添加 `export GOPATH=/home/user/goprograms`)，而你的项目将作为 src 的子目录。uc 包中的功能在 uc.go 中实现。

示例 9.6 uc.go：

```
package uc
import "strings"

func UpperCase(str string) string {
    return strings.ToUpper(str)
}
```

包通常附带一个或多个测试文件，在这我们创建了一个 uc_test.go 文件，如第 9.8 节所述。

示例 9.7 test.go

```
package uc
import "testing"

type ucTest struct {
    in, out string
}

var ucTests = []ucTest {
```

```
    ucTest{"abc", "ABC"},
    ucTest{"cvo-az", "CVO-AZ"},
    ucTest{"Antwerp", "ANTWERP"},
}

func TestUC(t *testing.T) {
    for _, ut := range ucTests {
        uc := UpperCase(ut.in)
        if uc != ut.out {
            t.Errorf("UpperCase(%s) = %s, must be %s", ut.in, uc,
            ut.out)
        }
    }
}
```

通过指令编译并安装包到本地： `go install uc` ，这会将 uc.a 复制到 pkg/linux_amd64 下面。

另外，使用 make ， 通过以下内容创建一个包的 Makefile 在 src/uc 目录下:

```
include $(GOROOT)/src/Make.inc

TARG=uc
GOFILES=\
        uc.go\

include $(GOROOT)/src/Make.pkg
```

在该目录下的命令行调用: gomake

这将创建一个 _obj 目录并将包编译生成的存档 uc.a 放在该目录下。

这个包可以通过 go test 测试。

创建一个 uc.a 的测试文件在目录下，输出为 PASS 时测试通过。

在第 13.8 节我们将给出另外一个测试例子并进行深入研究。

备注：有可能你当前的用户不具有足够的资格使用 go install(没有权限)。这种情况下，选择 root 用户 su。确保 Go 环境变量和 Go 源码路径也设置给 su，同样也适用你的普通用户(详见第 2.3 节)。

接下来我们创建主程序 ucmain.go:

示例 9.8 ucmain.go：

```
package main
import (
    "./src/uc"
    "fmt"
)

func main() {
    str1 := "USING package uc!"
    fmt.Println(uc.UpperCase(str1))
}
```

然后在这个目录下输入 `go install` 。

另外复制 uc.a 到 /home/user/goprograms 目录并创建一个 Makefile 并写入文本：

```
include $(GOROOT)/src/Make.inc
TARG=ucmain
GOFILES=\
    ucmain.go\

include $(GOROOT)/src/Make.cmd
```

执行 gomake 编译 `ucmain.go` 生成可执行文件ucmain

运行 `./ucmain` 显示: `USING PACKAGE UC!` 。

9.8.2 本地安装包

本地包在用户目录下，使用给出的目录结构，以下命令用来从源码安装本地包：

```
go install /home/user/goprograms/src/uc # 编译安装uc
cd /home/user/goprograms/uc
go install ./uc     # 编译安装uc（和之前的指令一样）
cd ..
go install .    # 编译安装ucmain
```

安装到 `$GOPATH` 下：

如果我们想安装的包在系统上的其他 Go 程序中被使用，它一定要安装到 `$GOPATH` 下。
这样做，在 .profile 和 .bashrc 中设置 `export GOPATH=/home/user/goprograms` 。

然后执行 go install uc 将会复制包存档到 `$GOPATH/pkg/LINUX_AMD64/uc` 。

现在，uc 包可以通过 `import "uc"` 在任何 Go 程序中被引用。

9.8.3 依赖系统的代码

在不同的操作系统上运行的程序以不同的代码实现是非常少见的：绝大多数情况下语言和标准库解决了大部分的可移植性问题。

你有一个很好的理由去写平台特定的代码，例如汇编语言。这种情况下，按照下面的约定是合理的：

```
prog1.go
prog1_linux.go
prog1_darwin.go
prog1_windows.go
```

prog1.go 定义了不同操作系统通用的接口，并将系统特定的代码写到 prog1_os.go 中。
对于 Go 工具你可以指定 `prog1_$GOOS.go` 或 `prog1_$GOARCH.go`
或在平台 Makefile 中： `prog1_$(GOOS).go\` 或 `prog1_$(GOARCH).go\` 。

通过 Git 打包和安装

9.9 通过 Git 打包和安装

9.9.1 安装到 GitHub

以上的方式对于本地包来说是可以的，但是我们如何打包代码到开发者圈子呢？那么我们需要一个云端的源码的版本控制系统，比如著名的 Git。

在 Linux 和 OS X 的机器上 Git 是默认安装的，在 Windows 上你必须先自行安装，参见 GitHub 帮助页面。

这里将通过为第 9.8 节中的 uc 包创建一个 git 仓库作为演示

进入到 uc 包目录下并创建一个 Git 仓库在里面: `git init` 。

信息提示: `Initialized empty git repository in $PWD/uc` 。

每一个 Git 项目都需要一个对包进行描述的 README.md 文件，所以需要打开你的文本编辑器（gedit、notepad 或 LiteIde）并添加一些说明进去。

- 添加所有文件到仓库: `git add README.md uc.go uc_test.go Makefile` 。
- 标记为第一个版本: `git commit -m "initial rivision"` 。

现在必须登录 GitHub 网站。

如果您还没有账号，可以去注册一个开源项目的免费帐号。输入正确的帐号密码和有效的邮箱地址并进一步创建用户。然后你将获得一个 Git 命令的列表。本地仓库的操作命令已经完成。一个优秀的系统在你遇到任何问题的时候将 引导你。

在云端创建一个新的 uc 仓库;发布的指令为(`NNNN` 替代用户名):

```
git remote add origin git@github.com:NNNN/uc.git
git push -u origin master
```

操作完成后检查 GitHub 上的包页面: `http://github.com/NNNN/uc` 。

9.9.2 从 GitHub 安装

如果有人想安装您的远端项目到本地机器，打开终端并执行（NNNN 是你在 GitHub 上的用户名）: `go get github.com/NNNN/uc` 。

这样现在这台机器上的其他 Go 应用程序也可以通过导入路径: `"github.com/NNNN/uc"` 代替 `"./uc/uc"` 来使用。

也可以将其缩写为: `import uc "github.com/NNNN/uc"` 。

然后修改 Makefile: 将 `TARG=uc` 替换为 `TARG=github.com/NNNN/uc` 。

Gomake（和 go install）将通过 `$GOPATH` 下的本地版本进行工作。

网站和版本控制系统的其他的选择(括号中为网站所使用的版本控制系统):

- BitBucket(hg/Git)

- GitHub(Git)
- Google Code(hg/Git/svn)
- Launchpad(bzr)

版本控制系统可以选择你熟悉的或者本地使用的代码版本控制。Go 核心代码的仓库是使用 Mercurial(hg) 来控制的，所以它是一个最可能保证你可以得到开发者项目中最好的软件。Git 也很出名，同样也适用。如果你从未使用过版本控制，这些网站有一些很好的帮助并且你可以通过在谷歌搜索 “{name} tutorial”，(name为你想要使用的版本控制系统),得到许多很好的教程。

Go 的外部包和项目

9.10 Go 的外部包和项目

现在我们知道如何使用 Go 以及它的标准库了，但是 Go 的生态要比这大的多。当着手自己的 Go 项目时，最好先查找下是否有些存在的第三方的包或者项目能不能使用。大多数可以通过 go install 来进行安装。

Go Walker 支持根据包名在海量数据中查询。

目前已经有许多非常好的外部库，如：

- MySQL(GoMySQL), PostgreSQL(go-pgsql), MongoDB (mgo, gomongo), CouchDB (couch-go), ODBC (godbcl), Redis (redis.go) and SQLite3 (gosqlite) database drivers
- SDL bindings
- Google’s Protocal Buffers(goprotobuf)
- XML-RPC(go-xmlrpc)
- Twitter(twitterstream)
- OAuth libraries(GoAuth)

在 Go 程序中使用外部库

9.11 在 Go 程序中使用外部库

（本节我们将创建一个 Web 应用和它的 Google App Engine 版本,在第 19 和 21 章分别说明，当你阅读到这些章节时可以再回到这个例子。）

当开始一个新项目或增加新的功能到现有的项目，你可以通过在应用程序中使用已经存在的库来节省开发时间。为了做到这一点，你必须理解库的 API（应用编程接口），那就是：库中有哪些方法可以调用，如何调用。你可能没有这个库的源代码，但作者肯定有记载的 API 以及详细介绍了如何使用它。

作为一个例子，我们将使用谷歌的 API 的 urlshortener 编写一个小程序：你可以尝试一下在 http://goo.gl/ 输入一个像 “http://www.destandaard.be" 这样的URL，你会看到一个像 “http://goo.gl/O9SUO" 这样更短的 URL 返回，也就是说，在 Twitter 之类的服务中这是非常容易嵌入的。谷歌 urlshortener 服务的文档可以在 “http://code.google.com/apis/urlshortener/" 找到。(第 19 章，我们将开发自己版本的 urlshortener)。

谷歌将这项技术提供给其他开发者，作为 API 我们可以在我们自己的应用程序中调用（释放到指定的限制）。他们也生成了一个 Go 语言客户端库使其变得更容易。

备注：谷歌让通过使用 Google API Go 客户端服务的开发者生活变得更简单，Go 客户端程序自动生成于 Google 库的 JSON 描述。更多详情在 项目页面 查看。

下载并安装 Go 客户端库:
将通过 go install 实现。但是首先要验证环境变量中是否含有 `GOPATH` 变量，因为外部源码将被下载到 `$GOPATH/src` 目录下并被安装到 `$GOPATH/PKG/"machine_arch"/` 目录下。

我们将通过在终端调用以下命令来安装 API:

```
go install google.golang.org/api/urlshortener/v1
```

go install 将下载源码，编译并安装包

使用 urlshortener 服务的 web 程序:
现在我们可以通过导入并赋予别名来使用已安装的包：

```
import "google.golang.org/api/urlshortener/v1"
```

现在我们写一个 Web 应用(参见第 15 章 4-8 节)通过表单实现短地址和长地址的相互转换。我们将使用 `template` 包并写三个处理函数：root 函数通过执行表单模板来展示表单。short 函数将长地址转换为短地址，long 函数逆向转换。

要调用 urlshortener 接口必须先通过 http 包中的默认客户端创建一个服务实例 urlshortenerSvc：

```
urlshortenerSvc, _ := urlshortener.New(http.DefaultClient)
```

我们通过调用服务中的 `Url.Insert` 中的 `Do` 方法传入包含长地址的 `Url` 数据结构从而获取短地址：

```
url, _ := urlshortenerSvc.Url.Insert(&urlshortener.Url{LongUrl: longUrl}).Do()
```

返回 `url` 的 `Id` 便是我们需要的短地址。

我们通过调用服务中的 `Url.Get` 中的 `Do` 方法传入包含短地址的Url数据结构从而获取长地址：

```
url, error := urlshortenerSvc.Url.Get(shwortUrl).Do()
```

返回的长地址便是转换前的原始地址。

示例 9.9 urlshortener.go

```
package main

import (
    "fmt"
    "net/http"
    "text/template"

    "google.golang.org/api/urlshortener/v1"
)
func main() {
    http.HandleFunc("/", root)
    http.HandleFunc("/short", short)
    http.HandleFunc("/long", long)

    http.ListenAndServe("localhost:8080", nil)
}
// the template used to show the forms and the results web page to the user
var rootHtmlTmpl = template.Must(template.New("rootHtml").Parse(`
<html><body>
<h1>URL SHORTENER</h1>
{{if .}}{{.}}<br /><br />{{end}}
<form action="/short" type="POST">
Shorten this: <input type="text" name="longUrl" />
<input type="submit" value="Give me the short URL" />
</form>
<br />
<form action="/long" type="POST">
Expand this: http://goo.gl/<input type="text" name="shortUrl" />
<input type="submit" value="Give me the long URL" />
</form>
</body></html>
`))
func root(w http.ResponseWriter, r *http.Request) {
   rootHtmlTmpl.Execute(w, nil)
}
func short(w http.ResponseWriter, r *http.Request) {
    longUrl := r.FormValue("longUrl")
    urlshortenerSvc, _ := urlshortener.New(http.DefaultClient)
    url, _ := urlshortenerSvc.Url.Insert(&urlshortener.Url{LongUrl:
    longUrl,}).Do()
    rootHtmlTmpl.Execute(w, fmt.Sprintf("Shortened version of %s is : %s",
    longUrl, url.Id))
}

func long(w http.ResponseWriter, r *http.Request) {
    shortUrl := "http://goo.gl/" + r.FormValue("shortUrl")
    urlshortenerSvc, _ := urlshortener.New(http.DefaultClient)
    url, err := urlshortenerSvc.Url.Get(shortUrl).Do()
    if err != nil {
        fmt.Println("error: %v", err)
        return

    }
    rootHtmlTmpl.Execute(w, fmt.Sprintf("Longer version of %s is : %s",
```

```
    shortUrl, url.LongUrl))
}
```

执行这段代码：

```
go run urlshortener.go
```

通过浏览 `http://localhost:8080/` 的页面来测试。

为了代码的简洁我们并没有检测返回的错误状态，但是在真实的生产环境的应用中一定要做检测。

将应用放入 Google App Engine，我们只需要在之前的代码中作出如下改变：

```
package main -> package urlshort
func main() -> func init()
```

创建一个和包同名的目录 `urlshort` ，并将以下两个安装目录复制到这个目录：

```
google.golang.org/api/urlshortener
google.golang.org/api/googleapi
```

此外还要配置下配置文件 `app.yaml` ，内容如下：

```
application: urlshort
version: 0-1-test
runtime: go
api_version: 3
handlers:
- url: /.*
script: _go_app
```

现在你可以去到你的项目目录并在终端运行： `dev_appserver.py urlshort`

在浏览器打开你的 Web应用：http://localhost:8080。

结构（struct）与方法（method）

10.0 结构（struct）与方法（method）

Go 通过类型别名（alias types）和结构体的形式支持用户自定义类型，或者叫定制类型。一个带属性的结构体试图表示一个现实世界中的实体。结构体是复合类型（composite types），当需要定义一个类型，它由一系列属性组成，每个属性都有自己的类型和值的时候，就应该使用结构体，它把数据聚集在一起。然后可以访问这些数据，就好像它是一个独立实体的一部分。结构体也是值类型，因此可以通过 **new** 函数来创建。

组成结构体类型的那些数据称为 **字段（fields）**。每个字段都有一个类型和一个名字；在一个结构体中，字段名字必须是唯一的。

结构体的概念在软件工程上旧的术语叫 ADT（抽象数据类型：Abstract Data Type），在一些老的编程语言中叫 **记录（Record）**，比如 Cobol，在 C 家族的编程语言中它也存在，并且名字也是 **struct**，在面向对象的编程语言中，跟一个无方法的轻量级类一样。不过因为 Go 语言中没有类的概念，因此在 Go 中结构体有着更为重要的地位。

结构体定义

10.1 结构体定义

结构体定义的一般方式如下：

```
type identifier struct {
    field1 type1
    field2 type2
    ...
}
```

`type T struct {a, b int}` 也是合法的语法，它更适用于简单的结构体。

结构体里的字段都有 **名字**，像 field1、field2 等，如果字段在代码中从来也不会被用到，那么可以命名它为 _。

结构体的字段可以是任何类型，甚至是结构体本身（参考第 10.5 节），也可以是函数或者接口（参考第 11 章）。可以声明结构体类型的一个变量，然后像下面这样给它的字段赋值：

```
var s T
s.a = 5
s.b = 8
```

数组可以看作是一种结构体类型，不过它使用下标而不是具名的字段。

使用 new

使用 **new** 函数给一个新的结构体变量分配内存，它返回指向已分配内存的指针：`var t *T = new(T)`，如果需要可以把这条语句放在不同的行（比如定义是包范围的，但是分配却没有必要在开始就做）。

```
var t *T
t = new(T)
```

写这条语句的惯用方法是：`t := new(T)`，变量 `t` 是一个指向 `T` 的指针，此时结构体字段的值是它们所属类型的零值。

声明 `var t T` 也会给 `t` 分配内存，并零值化内存，但是这个时候 `t` 是类型T。在这两种方式中，`t` 通常被称做类型 T 的一个实例（instance）或对象（object）。

示例 10.1 structs_fields.go 给出了一个非常简单的例子：

```
package main
import "fmt"

type struct1 struct {
    i1  int
    f1  float32
    str string
}

func main() {
    ms := new(struct1)
```

```
    ms.i1 = 10
    ms.f1 = 15.5
    ms.str= "Chris"

    fmt.Printf("The int is: %d\n", ms.i1)
    fmt.Printf("The float is: %f\n", ms.f1)
    fmt.Printf("The string is: %s\n", ms.str)
    fmt.Println(ms)
}
```

输出：

```
The int is: 10
The float is: 15.500000
The string is: Chris
&{10 15.5 Chris}
```

使用 `fmt.Println` 打印一个结构体的默认输出可以很好的显示它的内容，类似使用 **%v** 选项。

就像在面向对象语言所作的那样，可以使用点号符给字段赋值：`structname.fieldname = value` 。

同样的，使用点号符可以获取结构体字段的值：`structname.fieldname` 。

在 Go 语言中这叫 **选择器（selector）**。无论变量是一个结构体类型还是一个结构体类型指针，都使用同样的 **选择器符（selector-notation）** 来引用结构体的字段：

```
type myStruct struct { i int }
var v myStruct    // v是结构体类型变量
var p *myStruct   // p是指向一个结构体类型变量的指针
v.i
p.i
```

初始化一个结构体实例（一个结构体字面量：struct-literal）的更简短和惯用的方式如下：

```
    ms := &struct1{10, 15.5, "Chris"}
    // 此时ms的类型是 *struct1
```

或者：

```
    var ms struct1
    ms = struct1{10, 15.5, "Chris"}
```

混合字面量语法（composite literal syntax） `&struct1{a, b, c}` 是一种简写，底层仍然会调用 `new ()` ，这里值的顺序必须按照字段顺序来写。在下面的例子中能看到可以通过在值的前面放上字段名来初始化字段的方式。表达式 `new(Type)` 和 `&Type{}` 是等价的。

时间间隔（开始和结束时间以秒为单位）是使用结构体的一个典型例子：

```
type Interval struct {
    start int
    end   int
}
```

初始化方式：

```
intr := Interval{0, 3}            (A)
intr := Interval{end:5, start:1}  (B)
intr := Interval{end:5}           (C)
```

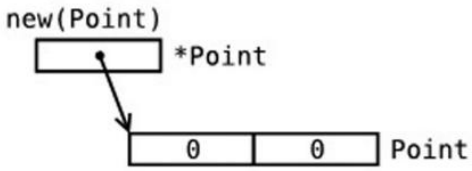
在（A）中，值必须以字段在结构体定义时的顺序给出，**&** 不是必须的。（B）显示了另一种方式，字段名加一个冒号放在值的前面，这种情况下值的顺序不必一致，并且某些字段还可以被忽略掉，就像（C）中那样。

结构体类型和字段的命名遵循可见性规则（第 4.2 节），一个导出的结构体类型中有些字段是导出的，另一些不是，这是可能的。

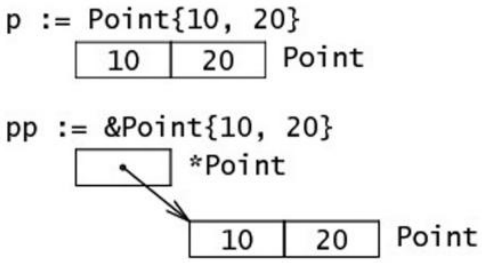
下图说明了结构体类型实例和一个指向它的指针的内存布局：

```
type Point struct { x, y int }
```

使用 new 初始化：



作为结构体字面量初始化：



类型 struct1 在定义它的包 pack1 中必须是唯一的，它的完全类型名是：`pack1.struct1`。

下面的例子 Listing 10.2—person.go 显示了一个结构体 Person，一个方法，方法有一个类型为 `*Person` 的参数（因此对象本身是可以被改变的），以及三种调用这个方法的不同方式：

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

type Person struct {
    firstName    string
    lastName     string
}

func upPerson(p *Person) {
    p.firstName = strings.ToUpper(p.firstName)
    p.lastName = strings.ToUpper(p.lastName)
```

```
}

func main() {
    // 1-struct as a value type:
    var pers1 Person
    pers1.firstName = "Chris"
    pers1.lastName = "Woodward"
    upPerson(&pers1)
    fmt.Printf("The name of the person is %s %s\n", pers1.firstName, pers1.lastName)

    // 2—struct as a pointer:
    pers2 := new(Person)
    pers2.firstName = "Chris"
    pers2.lastName = "Woodward"
    (*pers2).lastName = "Woodward"  // 这是合法的
    upPerson(pers2)
    fmt.Printf("The name of the person is %s %s\n", pers2.firstName, pers2.lastName)

    // 3—struct as a literal:
    pers3 := &Person{"Chris","Woodward"}
    upPerson(pers3)
    fmt.Printf("The name of the person is %s %s\n", pers3.firstName, pers3.lastName)
}
```

输出：

```
The name of the person is CHRIS WOODWARD
The name of the person is CHRIS WOODWARD
The name of the person is CHRIS WOODWARD
```

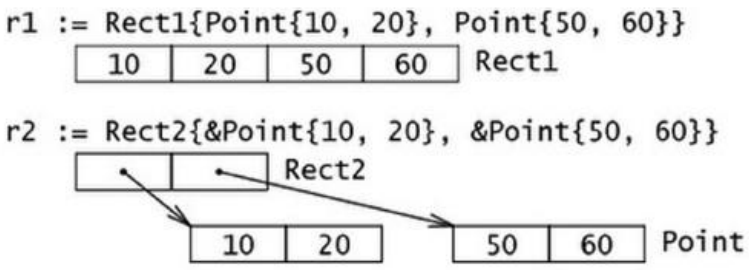
在上面例子的第二种情况中，可以直接通过指针，像 `pers2.lastName="Woodward"` 这样给结构体字段赋值，没有像 C++ 中那样需要使用 `->` 操作符，Go 会自动做这样的转换。

注意也可以通过解指针的方式来设置值： `(*pers2).lastName = "Woodward"`

结构体的内存布局

Go 语言中，结构体和它所包含的数据在内存中是以连续块的形式存在的，即使结构体中嵌套有其他的结构体，这在性能上带来了很大的优势。不像 Java 中的引用类型，一个对象和它里面包含的对象可能会在不同的内存空间中，这点和 Go 语言中的指针很像。下面的例子清晰地说明了这些情况：

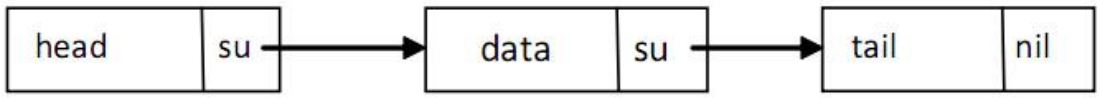
```
type Rect1 struct {Min, Max Point }
type Rect2 struct {Min, Max *Point }
```



递归结构体

结构体类型可以通过引用自身来定义。这在定义链表或二叉树的元素（通常叫节点）时特别有用，此时节点包含指向临近节点的链接（地址）。如下所示，链表中的 `su`，树中的 `ri` 和 `le` 分别是指向别的节点的指针。

链表：



这块的 `data` 字段用于存放有效数据（比如 float64），`su` 指针指向后继节点。

Go 代码：

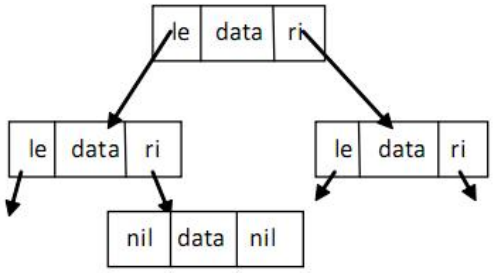
```
type Node struct {
    data    float64
    su      *Node
}
```

链表中的第一个元素叫 `head`，它指向第二个元素；最后一个元素叫 `tail`，它没有后继元素，所以它的 `su` 为 nil 值。当然真实的链接会有很多数据节点，并且链表可以动态增长或收缩。

同样地可以定义一个双向链表，它有一个前趋节点 `pr` 和一个后继节点 `su`：

```
type Node struct {
    pr      *Node
    data    float64
    su      *Node
}
```

二叉树：



二叉树中每个节点最多能链接至两个节点：左节点（le）和右节点（ri），这两个节点本身又可以有左右节点，依次类推。树的顶层节点叫根节点（**root**），底层没有子节点的节点叫叶子节点（**leaves**），叶子节点的 `le` 和 `ri` 指针为 nil 值。在 Go 中可以如下定义二叉树：

```
type Tree struct {
    le      *Tree
    data    float64
    ri      *Tree
}
```

结构体转换

Go 中的类型转换遵循严格的规则。当为结构体定义了一个 alias 类型时，此结构体类型和它的 alias 类型都有相同的底层类型，它们可以如示例 10.3 那样互相转换，同时需要注意其中非法赋值或转换引起的编译错误。

示例 10.3：

```
package main
import "fmt"

type number struct {
    f float32
}

type nr number   // alias type

func main() {
    a := number{5.0}
    b := nr{5.0}
    // var i float32 = b   // compile-error: cannot use b (type nr) as type float32 in assignment
    // var i = float32(b)  // compile-error: cannot convert b (type nr) to type float32
    // var c number = b    // compile-error: cannot use b (type nr) as type number in assignment
    // needs a conversion:
    var c = number(b)
    fmt.Println(a, b, c)
}
```

输出：

```
{5} {5} {5}
```

练习 10.1 vcard.go：

定义结构体 Address 和 VCard，后者包含一个人的名字、地址编号、出生日期和图像，试着选择正确的数据类型。构建一个自己的 vcard 并打印它的内容。

```
提示：
VCard 必须包含住址，它应该以值类型还是以指针类型放在 VCard 中呢？
第二种会好点，因为它占用内存少。包含一个名字和两个指向地址的指针的 Address 结构体可以使用 %v 打印：
{Kersschot 0x126d2b80 0x126d2be0}
```

练习 10.2 personex1.go：

修改 personex1.go，使它的参数 upPerson 不是一个指针，解释下二者的区别。

练习 10.3 point.go：

使用坐标 X、Y 定义一个二维 Point 结构体。同样地，对一个三维点使用它的极坐标定义一个 Polar 结构体。实现一个 `Abs()` 方法来计算一个 Point 表示的向量的长度，实现一个 `Scale` 方法，它将点的坐标乘以一个尺度因子（提示：使用 `math` 包里的 `Sqrt` 函数）（function Scale that multiplies the coordinates of a point with a scale
factor）。

练习 10.4 rectangle.go：

定义一个 Rectangle 结构体，它的长和宽是 int 类型，并定义方法 `Area()` 和 `Perimeter()`，然后进行测试。

使用工厂方法创建结构体实例

10.2 使用工厂方法创建结构体实例

10.2.1 结构体工厂

Go 语言不支持面向对象编程语言中那样的构造子方法，但是可以很容易的在 Go 中实现“构造子工厂”方法。为了方便通常会为类型定义一个工厂，按惯例，工厂的名字以 new 或 New 开头。假设定义了如下的 File 结构体类型：

```
type File struct {
    fd      int     // 文件描述符
    name    string  // 文件名
}
```

下面是这个结构体类型对应的工厂方法，它返回一个指向结构体实例的指针：

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }

    return &File{fd, name}
}
```

然后这样调用它：

```
f := NewFile(10, "./test.txt")
```

在 Go 语言中常常像上面这样在工厂方法里使用初始化来简便的实现构造函数。

如果 `File` 是一个结构体类型，那么表达式 `new(File)` 和 `&File{}` 是等价的。

这可以和大多数面向对象编程语言中笨拙的初始化方式做个比较： `File f = new File(...)` 。

我们可以说是工厂实例化了类型的一个对象，就像在基于类的OO语言中那样。

如果想知道结构体类型T的一个实例占用了多少内存，可以使用： `size := unsafe.Sizeof(T{})` 。

如何强制使用工厂方法

通过应用可见性规则参考4.2.1节、9.5 节就可以禁止使用 new 函数，强制用户使用工厂方法，从而使类型变成私有的，就像在面向对象语言中那样。

```
type matrix struct {
    ...
}

func NewMatrix(params) *matrix {
    m := new(matrix) // 初始化 m
    return m
}
```

在其他包里使用工厂方法：

```
package main
import "matrix"
...
wrong := new(matrix.matrix)     // 编译失败（matrix 是私有的）
right := matrix.NewMatrix(...)  // 实例化 matrix 的唯一方式
```

10.2.2 map 和 struct vs new() 和 make()

new 和 make 这两个内置函数已经在第 7.2.4 节通过切片的例子说明过一次。

现在为止我们已经见到了可以使用 `make()` 的三种类型中的其中两个：

```
slices  /  maps / channels（见第 14 章）
```

下面的例子说明了在映射上使用 new 和 make 的区别以及可能发生的错误：

示例 10.4 new_make.go（不能编译）

```
package main

type Foo map[string]string
type Bar struct {
    thingOne string
    thingTwo int
}

func main() {
    // OK
    y := new(Bar)
    (*y).thingOne = "hello"
    (*y).thingTwo = 1

    // NOT OK
    z := make(Bar) // 编译错误：cannot make type Bar
    (*z).thingOne = "hello"
    (*z).thingTwo = 1

    // OK
    x := make(Foo)
    x["x"] = "goodbye"
    x["y"] = "world"

    // NOT OK
    u := new(Foo)
    (*u)["x"] = "goodbye" // 运行时错误!! panic: assignment to entry in nil map
    (*u)["y"] = "world"
}
```

试图 `make()` 一个结构体变量，会引发一个编译错误，这还不是太糟糕，但是 `new()` 一个 map 并试图向其填充数据，将会引发运行时错误！ 因为 `new(Foo)` 返回的是一个指向 `nil` 的指针，它尚未被分配内存。所以在使用 `map` 时要特别谨慎。

使用自定义包中的结构体

10.3 使用自定义包中的结构体

下面的例子中，main.go 使用了一个结构体，它来自 struct_pack 下的包 structPack。

示例 10.5 structPack.go：

```
package structPack

type ExpStruct struct {
    Mi1 int
    Mf1 float32
}
```

示例 10.6 main.go：

```
package main
import (
    "fmt"
    "./struct_pack/structPack"
)

func main() {
    struct1 := new(structPack.ExpStruct)
    struct1.Mi1 = 10
    struct1.Mf1 = 16.

    fmt.Printf("Mi1 = %d\n", struct1.Mi1)
    fmt.Printf("Mf1 = %f\n", struct1.Mf1)
}
```

输出：

```
Mi1 = 10
Mf1 = 16.000000
```

带标签的结构体

10.4 带标签的结构体

结构体中的字段除了有名字和类型外，还可以有一个可选的标签（tag）：它是一个附属于字段的字符串，可以是文档或其他的重要标记。标签的内容不可以在一般的编程中使用，只有包 `reflect` 能获取它。我们将在下一章（第 11.10 节）中深入的探讨 `reflect` 包，它可以在运行时自省类型、属性和方法，比如：在一个变量上调用 `reflect.TypeOf()` 可以获取变量的正确类型，如果变量是一个结构体类型，就可以通过 Field 来索引结构体的字段，然后就可以使用 Tag 属性。

示例 10.7 struct_tag.go：

```
package main

import (
    "fmt"
    "reflect"
)

type TagType struct { // tags
    field1 bool   "An important answer"
    field2 string "The name of the thing"
    field3 int    "How much there are"
}

func main() {
    tt := TagType{true, "Barak Obama", 1}
    for i := 0; i < 3; i++ {
        refTag(tt, i)
    }
}

func refTag(tt TagType, ix int) {
    ttType := reflect.TypeOf(tt)
    ixField := ttType.Field(ix)
    fmt.Printf("%v\n", ixField.Tag)
}
```

输出：

```
An important answer
The name of the thing
How much there are
```

匿名字段和内嵌结构体

10.5 匿名字段和内嵌结构体

10.5.1 定义

结构体可以包含一个或多个 **匿名（或内嵌）字段**，即这些字段没有显式的名字，只有字段的类型是必须的，此时类型就是字段的名字。匿名字段本身可以是一个结构体类型，即 **结构体可以包含内嵌结构体**。

可以粗略地将这个和面向对象语言中的继承概念相比较，随后将会看到它被用来模拟类似继承的行为。Go 语言中的继承是通过内嵌或组合来实现的，所以可以说，在 Go 语言中，相比较于继承，组合更受青睐。

考虑如下的程序：

示例 10.8 structs_anonymous_fields.go：

```
package main

import "fmt"

type innerS struct {
    in1 int
    in2 int
}

type outerS struct {
    b    int
    c    float32
    int  // anonymous field
    innerS //anonymous field
}

func main() {
    outer := new(outerS)
    outer.b = 6
    outer.c = 7.5
    outer.int = 60
    outer.in1 = 5
    outer.in2 = 10

    fmt.Printf("outer.b is: %d\n", outer.b)
    fmt.Printf("outer.c is: %f\n", outer.c)
    fmt.Printf("outer.int is: %d\n", outer.int)
    fmt.Printf("outer.in1 is: %d\n", outer.in1)
    fmt.Printf("outer.in2 is: %d\n", outer.in2)

    // 使用结构体字面量
    outer2 := outerS{6, 7.5, 60, innerS{5, 10}}
    fmt.Println("outer2 is:", outer2)
}
```

输出：

```
outer.b is: 6
outer.c is: 7.500000
outer.int is: 60
outer.in1 is: 5
outer.in2 is: 10
outer2 is:{6 7.5 60 {5 10}}
```

通过类型 `outer.int` 的名字来获取存储在匿名字段中的数据，于是可以得出一个结论：在一个结构体中对于每一种数据类型只能有一个匿名字段。

10.5.2 内嵌结构体

同样地结构体也是一种数据类型，所以它也可以作为一个匿名字段来使用，如同上面例子中那样。外层结构体通过 `outer.in1` 直接进入内层结构体的字段，内嵌结构体甚至可以来自其他包。内层结构体被简单的插入或者内嵌进外层结构体。这个简单的“继承”机制提供了一种方式，使得可以从另外一个或一些类型继承部分或全部实现。

另外一个例子：

示例 10.9 embedd_struct.go：

```
package main

import "fmt"

type A struct {
    ax, ay int
}

type B struct {
    A
    bx, by float32
}

func main() {
    b := B{A{1, 2}, 3.0, 4.0}
    fmt.Println(b.ax, b.ay, b.bx, b.by)
    fmt.Println(b.A)
}
```

输出：

```
1 2 3 4
{1 2}
```

练习 10.5 anonymous_struct.go：

创建一个结构体，它有一个具名的 float 字段，2 个匿名字段，类型分别是 int 和 string。通过结构体字面量新建一个结构体实例并打印它的内容。

10.5.3 命名冲突

当两个字段拥有相同的名字（可能是继承来的名字）时该怎么办呢？

1. 外层名字会覆盖内层名字（但是两者的内存空间都保留），这提供了一种重载字段或方法的方式；
2. 如果相同的名字在同一级别出现了两次，如果这个名字被程序使用了，将会引发一个错误（不使用没关系）。没有办法来解决这种问题引起的二义性，必须由程序员自己修正。

例子：

```
type A struct {a int}
type B struct {a, b int}

type C struct {A; B}
var c C
```

规则 2：使用 `c.a` 是错误的，到底是 `c.A.a` 还是 `c.B.a` 呢？会导致编译器错误：**ambiguous DOT reference c.a disambiguate with either c.A.a or c.B.a**。

```
type D struct {B; b float32}
var d D
```

规则1：使用 `d.b` 是没问题的：它是 float32，而不是 `B` 的 `b` 。如果想要内层的 `b` 可以通过 `d.B.b` 得到。

方法

10.6 方法

10.6.1 方法是什么

在 Go 语言中，结构体就像是类的一种简化形式，那么面向对象程序员可能会问：类的方法在哪里呢？在 Go 中有一个概念，它和方法有着同样的名字，并且大体上意思相同：Go 方法是作用在接收者（receiver）上的一个函数，接收者是某种类型的变量。因此方法是一种特殊类型的函数。

接收者类型可以是（几乎）任何类型，不仅仅是结构体类型：任何类型都可以有方法，甚至可以是函数类型，可以是 int、bool、string 或数组的别名类型。但是接收者不能是一个接口类型（参考 第 11 章），因为接口是一个抽象定义，但是方法却是具体实现；如果这样做会引发一个编译错误：**invalid receiver type…**。

最后接收者不能是一个指针类型，但是它可以是任何其他允许类型的指针。

一个类型加上它的方法等价于面向对象中的一个类。一个重要的区别是：在 Go 中，类型的代码和绑定在它上面的方法的代码可以不放置在一起，它们可以存在在不同的源文件，唯一的要求是：它们必须是同一个包的。

类型 T（或 *T）上的所有方法的集合叫做类型 T（或 *T）的方法集（method set）。

因为方法是函数，所以同样的，不允许方法重载，即对于一个类型只能有一个给定名称的方法。但是如果基于接收者类型，是有重载的：具有同样名字的方法可以在 2 个或多个不同的接收者类型上存在，比如在同一个包里这么做是允许的：

```
func (a *denseMatrix) Add(b Matrix) Matrix
func (a *sparseMatrix) Add(b Matrix) Matrix
```

别名类型没有原始类型上已经定义过的方法。

定义方法的一般格式如下：

```
func (recv receiver_type) methodName(parameter_list) (return_value_list) { ... }
```

在方法名之前，`func` 关键字之后的括号中指定 receiver。

如果 `recv` 是 receiver 的实例，Method1 是它的方法名，那么方法调用遵循传统的 `object.name` 选择器符号：**recv.Method1()**。

如果 `recv` 是一个指针，Go 会自动解引用。

如果方法不需要使用 `recv` 的值，可以用 _ 替换它，比如：

```
func (_ receiver_type) methodName(parameter_list) (return_value_list) { ... }
```

`recv` 就像是面向对象语言中的 `this` 或 `self`，但是 Go 中并没有这两个关键字。随个人喜好，你可以使用 `this` 或 `self` 作为 receiver 的名字。下面是一个结构体上的简单方法的例子：

示例 10.10 method .go：

```
package main
```

```
import "fmt"

type TwoInts struct {
    a int
    b int
}

func main() {
    two1 := new(TwoInts)
    two1.a = 12
    two1.b = 10

    fmt.Printf("The sum is: %d\n", two1.AddThem())
    fmt.Printf("Add them to the param: %d\n", two1.AddToParam(20))

    two2 := TwoInts{3, 4}
    fmt.Printf("The sum is: %d\n", two2.AddThem())
}

func (tn *TwoInts) AddThem() int {
    return tn.a + tn.b
}

func (tn *TwoInts) AddToParam(param int) int {
    return tn.a + tn.b + param
}
```

输出：

```
The sum is: 22
Add them to the param: 42
The sum is: 7
```

下面是非结构体类型上方法的例子：

示例 10.11 method2.go：

```
package main

import "fmt"

type IntVector []int

func (v IntVector) Sum() (s int) {
    for _, x := range v {
        s += x
    }
    return
}

func main() {
    fmt.Println(IntVector{1, 2, 3}.Sum()) // 输出是6
}
```

练习 10.6 employee_salary.go

定义结构体 `employee` ，它有一个 `salary` 字段，给这个结构体定义一个方法 `giveRaise` 来按照指定的百分比增加薪水。

练习 10.7 iteration_list.go

下面这段代码有什么错？

```
package main

import "container/list"

func (p *list.List) Iter() {
    // ...
}

func main() {
    lst := new(list.List)
    for _= range lst.Iter() {
    }
}
```

类型和作用在它上面定义的方法必须在同一个包里定义，这就是为什么不能在 int、float 或类似这些的类型上定义方法。试图在 int 类型上定义方法会得到一个编译错误：

```
cannot define new methods on non-local type int
```

比如想在 `time.Time` 上定义如下方法：

```
func (t time.Time) first3Chars() string {
    return time.LocalTime().String()[0:3]
}
```

类型在其他的，或是非本地的包里定义，在它上面定义方法都会得到和上面同样的错误。

但是有一个间接的方式：可以先定义该类型（比如：int 或 float）的别名类型，然后再为别名类型定义方法。或者像下面这样将它作为匿名类型嵌入在一个新的结构体中。当然方法只在这个别名类型上有效。

示例 10.12 method_on_time.go：

```
package main

import (
    "fmt"
    "time"
)

type myTime struct {
    time.Time //anonymous field
}

func (t myTime) first3Chars() string {
    return t.Time.String()[0:3]
}
func main() {
    m := myTime{time.Now()}
    // 调用匿名Time上的String方法
    fmt.Println("Full time now:", m.String())
    // 调用myTime.first3Chars
    fmt.Println("First 3 chars:", m.first3Chars())
}
```

```
/* Output:
Full time now: Mon Oct 24 15:34:54 Romance Daylight Time 2011
First 3 chars: Mon
*/
```

10.6.2 函数和方法的区别

函数将变量作为参数：**Function1(recv)**

方法在变量上被调用：**recv.Method1()**

在接收者是指针时，方法可以改变接收者的值（或状态），这点函数也可以做到（当参数作为指针传递，即通过引用调用时，函数也可以改变参数的状态）。

不要忘记 Method1 后边的括号 ()，否则会引发编译器错误： `method recv.Method1 is not an expression, must be called`

接收者必须有一个显式的名字，这个名字必须在方法中被使用。

receiver_type 叫做 **（接收者）基本类型**，这个类型必须在和方法同样的包中被声明。

在 Go 中，（接收者）类型关联的方法不写在类型结构里面，就像类那样；耦合更加宽松；类型和方法之间的关联由接收者来建立。

方法没有和数据定义（结构体）混在一起：它们是正交的类型；表示（数据）和行为（方法）是独立的。

10.6.3 指针或值作为接收者

鉴于性能的原因，`recv` 最常见的是一个指向 receiver_type 的指针（因为我们不想要一个实例的拷贝，如果按值调用的话就会是这样），特别是在 receiver 类型是结构体时，就更是如此了。

如果想要方法改变接收者的数据，就在接收者的指针类型上定义该方法。否则，就在普通的值类型上定义方法。

下面的例子 `pointer_value.go` 作了说明：`change()` 接受一个指向 B 的指针，并改变它内部的成员；`write()` 通过拷贝接受 B 的值并只输出B的内容。注意 Go 为我们做了探测工作，我们自己并没有指出是否在指针上调用方法，Go 替我们做了这些事情。b1 是值而 b2 是指针，方法都支持运行了。

示例 10.13 pointer_value.go：

```
package main

import (
    "fmt"
)

type B struct {
    thing int
}

func (b *B) change() { b.thing = 1 }

func (b B) write() string { return fmt.Sprint(b) }

func main() {
    var b1 B // b1是值
    b1.change()
    fmt.Println(b1.write())
```

```
    b2 := new(B) // b2是指针
    b2.change()
    fmt.Println(b2.write())
}

/* 输出:
{1}
{1}
*/
```

试着在 `write()` 中改变接收者b的值：将会看到它可以正常编译，但是开始的 b 没有被改变。

我们知道方法将指针作为接收者不是必须的，如下面的例子，我们只是需要 `Point3` 的值来做计算：

```
type Point3 struct { x, y, z float64 }
// A method on Point3
func (p Point3) Abs() float64 {
    return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
}
```

这样做稍微有点昂贵，因为 `Point3` 是作为值传递给方法的，因此传递的是它的拷贝，这在 Go 中是合法的。也可以在指向这个类型的指针上调用此方法（会自动解引用）。

假设 `p3` 定义为一个指针： `p3 := &Point{ 3, 4, 5}` 。

可以使用 `p3.Abs()` 来替代 `(*p3).Abs()` 。

像例子 10.10（method1.go）中接收者类型是 `*TwoInts` 的方法 `AddThem()` ，它能在类型 `TwoInts` 的值上被调用，这是自动间接发生的。

因此 `two2.AddThem` 可以替代 `(&two2).AddThem()` 。

在值和指针上调用方法：

可以有连接到类型的方法，也可以有连接到类型指针的方法。

但是这没关系：对于类型 T，如果在 *T 上存在方法 `Meth()` ，并且 `t` 是这个类型的变量，那么 `t.Meth()` 会被自动转换为 `(&t).Meth()` 。

指针方法和值方法都可以在指针或非指针上被调用，如下面程序所示，类型 `List` 在值上有一个方法 `Len()` ，在指针上有一个方法 `Append()` ，但是可以看到两个方法都可以在两种类型的变量上被调用。

示例 10.14 methodset1.go：

```
package main

import (
    "fmt"
)

type List []int

func (l List) Len() int        { return len(l) }
func (l *List) Append(val int) { *l = append(*l, val) }

func main() {
    // 值
```

```
    var lst List
    lst.Append(1)
    fmt.Printf("%v (len: %d)", lst, lst.Len()) // [1] (len: 1)

    // 指针
    plst := new(List)
    plst.Append(2)
    fmt.Printf("%v (len: %d)", plst, plst.Len()) // &[2] (len: 1)
}
```

10.6.4 方法和未导出字段

考虑 `person2.go` 中的 `person` 包：类型 `Person` 被明确的导出了，但是它的字段没有被导出。例如在 `use_person2.go` 中 `p.firstName` 就是错误的。该如何在另一个程序中修改或者只是读取一个 `Person` 的名字呢？

这可以通过面向对象语言一个众所周知的技术来完成：提供 getter 和 setter 方法。对于 setter 方法使用 Set 前缀，对于 getter 方法只使用成员名。

示例 10.15 person2.go：

```
package person

type Person struct {
    firstName string
    lastName  string
}

func (p *Person) FirstName() string {
    return p.firstName
}

func (p *Person) SetFirstName(newName string) {
    p.firstName = newName
}
```

示例 10.16—use_person2.go：

```
package main

import (
    "./person"
    "fmt"
)

func main() {
    p := new(person.Person)
    // p.firstName undefined
    // (cannot refer to unexported field or method firstName)
    // p.firstName = "Eric"
    p.SetFirstName("Eric")
    fmt.Println(p.FirstName()) // Output: Eric
}
```

并发访问对象

对象的字段（属性）不应该由 2 个或 2 个以上的不同线程在同一时间去改变。如果在程序发生这种情况，为了安全并发访问，可以使用包 `sync` （参考第 9.3 节）中的方法。在第 14.17 节中我们会通过 goroutines 和 channels 探索另一种方式。

10.6.5 内嵌类型的方法和继承

当一个匿名类型被内嵌在结构体中时，匿名类型的可见方法也同样被内嵌，这在效果上等同于外层类型 **继承** 了这些方法：**将父类型放在子类型中来实现亚型**。这个机制提供了一种简单的方式来模拟经典面向对象语言中的子类和继承相关的效果，也类似 Ruby 中的混入（mixin）。

下面是一个示例（可以在练习 10.8 中进一步学习）：假定有一个 `Engine` 接口类型，一个 `Car` 结构体类型，它包含一个 `Engine` 类型的匿名字段：

```
type Engine interface {
    Start()
    Stop()
}

type Car struct {
    Engine
}
```

我们可以构建如下的代码：

```
func (c *Car) GoToWorkIn() {
    // get in car
    c.Start()
    // drive to work
    c.Stop()
    // get out of car
}
```

下面是 `method3.go` 的完整例子，它展示了内嵌结构体上的方法可以直接在外层类型的实例上调用：

```
package main

import (
    "fmt"
    "math"
)

type Point struct {
    x, y float64
}

func (p *Point) Abs() float64 {
    return math.Sqrt(p.x*p.x + p.y*p.y)
}

type NamedPoint struct {
    Point
    name string
}

func main() {
    n := &NamedPoint{Point{3, 4}, "Pythagoras"}
```

```
    fmt.Println(n.Abs()) // 打印5
}
```

内嵌将一个已存在类型的字段和方法注入到了另一个类型里：匿名字段上的方法“晋升”成为了外层类型的方法。当然类型可以有只作用于本身实例而不作用于内嵌“父”类型上的方法，

可以覆写方法（像字段一样）：和内嵌类型方法具有同样名字的外层类型的方法会覆写内嵌类型对应的方法。

在示例 10.18 method4.go 中添加：

```
func (n *NamedPoint) Abs() float64 {
    return n.Point.Abs() * 100.
}
```

现在 `fmt.Println(n.Abs())` 会打印 `500` 。

因为一个结构体可以嵌入多个匿名类型，所以实际上我们可以有一个简单版本的多重继承，就像： `type Child struct { Father; Mother}` 。在第 10.6.7 节中会进一步讨论这个问题。

结构体内嵌和自己在同一个包中的结构体时，可以彼此访问对方所有的字段和方法。

练习 10.8 inheritance_car.go

创建一个上面 `Car` 和 `Engine` 可运行的例子，并且给 `Car` 类型一个 `wheelCount` 字段和一个 `numberOfWheels()` 方法。

创建一个 `Mercedes` 类型，它内嵌 `Car` ，并新建 `Mercedes` 的一个实例，然后调用它的方法。

然后仅在 `Mercedes` 类型上创建方法 `sayHiToMerkel()` 并调用它。

10.6.6 如何在类型中嵌入功能

主要有两种方法来实现在类型中嵌入功能：

A：聚合（或组合）：包含一个所需功能类型的具名字段。

B：内嵌：内嵌（匿名地）所需功能类型，像前一节 10.6.5 所演示的那样。

为了使这些概念具体化，假设有一个 `Customer` 类型，我们想让它通过 `Log` 类型来包含日志功能， `Log` 类型只是简单地包含一个累积的消息（当然它可以是复杂的）。如果想让特定类型都具备日志功能，你可以实现一个这样的 `Log` 类型，然后将它作为特定类型的一个字段，并提供 `Log()` ，它返回这个日志的引用。

方式 A 可以通过如下方法实现（使用了第 10.7 节中的 `String()` 功能）：

示例 10.19 embed_func1.go：

```
package main

import (
    "fmt"
)

type Log struct {
    msg string
}

type Customer struct {
```

```
    Name string
    log  *Log
}

func main() {
    c := new(Customer)
    c.Name = "Barak Obama"
    c.log = new(Log)
    c.log.msg = "1 - Yes we can!"
    // shorter
    c = &Customer{"Barak Obama", &Log{"1 - Yes we can!"}}
    // fmt.Println(c) &{Barak Obama 1 - Yes we can!}
    c.Log().Add("2 - After me the world will be a better place!")
    //fmt.Println(c.log)
    fmt.Println(c.Log())

}

func (l *Log) Add(s string) {
    l.msg += "\n" + s
}

func (l *Log) String() string {
    return l.msg
}

func (c *Customer) Log() *Log {
    return c.log
}
```

输出：

```
1 - Yes we can!
2 - After me the world will be a better place!
```

相对的方式 B 可能会像这样：

```
package main

import (
    "fmt"
)

type Log struct {
    msg string
}

type Customer struct {
    Name string
    Log
}

func main() {
    c := &Customer{"Barak Obama", Log{"1 - Yes we can!"}}
    c.Add("2 - After me the world will be a better place!")
    fmt.Println(c)

}
```

```
func (l *Log) Add(s string) {
    l.msg += "\n" + s
}

func (l *Log) String() string {
    return l.msg
}

func (c *Customer) String() string {
    return c.Name + "\nLog:" + fmt.Sprintln(c.Log)
}
```

输出：

```
Barak Obama
Log:{1 - Yes we can!
2 - After me the world will be a better place!}
```

内嵌的类型不需要指针，`Customer` 也不需要 `Add` 方法，它使用 `Log` 的 `Add` 方法，`Customer` 有自己的 `String` 方法，并且在它里面调用了 `Log` 的 `String` 方法。

如果内嵌类型嵌入了其他类型，也是可以的，那些类型的方法可以直接在外层类型中使用。

因此一个好的策略是创建一些小的、可复用的类型作为一个工具箱，用于组成域类型。

10.6.7 多重继承

多重继承指的是类型获得多个父类型行为的能力，它在传统的面向对象语言中通常是不被实现的（C++ 和 Python 例外）。因为在类继承层次中，多重继承会给编译器引入额外的复杂度。但是在 Go 语言中，通过在类型中嵌入所有必要的父类型，可以很简单的实现多重继承。

作为一个例子，假设有一个类型 `CameraPhone`，通过它可以 `Call()`，也可以 `TakeAPicture()`，但是第一个方法属于类型 `Phone`，第二个方法属于类型 `Camera`。

只要嵌入这两个类型就可以解决这个问题，如下所示：

```
package main

import (
    "fmt"
)

type Camera struct{}

func (c *Camera) TakeAPicture() string {
    return "Click"
}

type Phone struct{}

func (p *Phone) Call() string {
    return "Ring Ring"
}

type CameraPhone struct {
    Camera
    Phone
}
```

```
func main() {
    cp := new(CameraPhone)
    fmt.Println("Our new CameraPhone exhibits multiple behaviors...")
    fmt.Println("It exhibits behavior of a Camera: ", cp.TakeAPicture())
    fmt.Println("It works like a Phone too: ", cp.Call())
}
```

输出：

```
Our new CameraPhone exhibits multiple behaviors...
It exhibits behavior of a Camera: Click
It works like a Phone too: Ring Ring
```

练习 10.9 point_methods.go：

从 `point.go` 开始（第 10.1 节的练习）：使用方法来实现 `Abs()` 和 `Scale()` 函数，`Point` 作为方法的接收者类型。也为 `Point3` 和 `Polar` 实现 `Abs()` 方法。完成了 `point.go` 中同样的事情，只是这次通过方法。

练习 10.10 inherit_methods.go：

定义一个结构体类型 `Base`，它包含一个字段 `id`，方法 `Id()` 返回 `id`，方法 `SetId()` 修改 `id`。结构体类型 `Person` 包含 `Base`，及 `FirstName` 和 `LastName` 字段。结构体类型 `Employee` 包含一个 `Person` 和 `salary` 字段。

创建一个 `employee` 实例，然后显示它的 `id`。

练习 10.11 magic.go：

首先预测一下下面程序的结果，然后动手实验下：

```
package main

import (
    "fmt"
)

type Base struct{}

func (Base) Magic() {
    fmt.Println("base magic")
}

func (self Base) MoreMagic() {
    self.Magic()
    self.Magic()
}

type Voodoo struct {
    Base
}

func (Voodoo) Magic() {
    fmt.Println("voodoo magic")
}

func main() {
    v := new(Voodoo)
```

```
    v.Magic()
    v.MoreMagic()
}
```

10.6.8 通用方法和方法命名

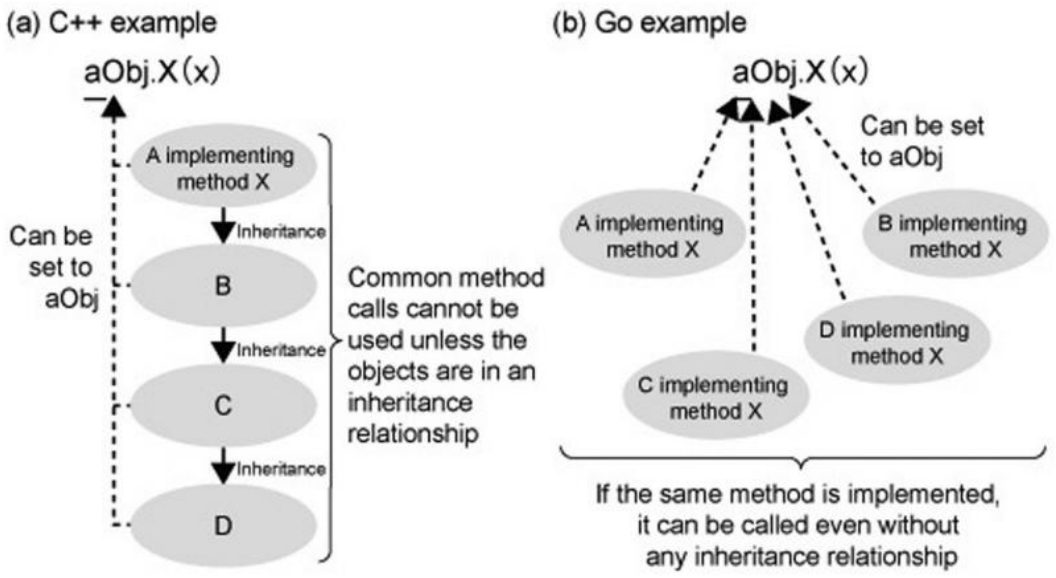
在编程中一些基本操作会一遍又一遍的出现，比如打开（Open）、关闭（Close）、读（Read）、写（Write）、排序（Sort）等等，并且它们都有一个大致的意思：打开（Open）可以作用于一个文件、一个网络连接、一个数据库连接等等。具体的实现可能千差万别，但是基本的概念是一致的。在 Go 语言中，通过使用接口（参考 第 11 章），标准库广泛的应用了这些规则，在标准库中这些通用方法都有一致的名字，比如 `Open()` 、 `Read()` 、 `Write()` 等。想写规范的 Go 程序，就应该遵守这些约定，给方法合适的名字和签名，就像那些通用方法那样。这样做会使 Go 开发的软件更加具有一致性和可读性。比如：如果需要一个 convert-to-string 方法，应该命名为 `String()` ，而不是 `ToString()` （参考第 10.7 节）。

10.6.9 和其他面向对象语言比较 Go 的类型和方法

在如 C++、Java、C# 和 Ruby 这样的面向对象语言中，方法在类的上下文中被定义和继承：在一个对象上调用方法时，运行时会检测类以及它的超类中是否有此方法的定义，如果没有会导致异常发生。

在 Go 语言中，这样的继承层次是完全没必要的：如果方法在此类型定义了，就可以调用它，和其他类型上是否存在这个方法没有关系。在这个意义上，Go 具有更大的灵活性。

下面的模式就很好的说明了这个问题：



Go 不需要一个显式的类定义，如同 Java、C++、C# 等那样，相反地，“类”是通过提供一组作用于一个共同类型的方法集来隐式定义的。类型可以是结构体或者任何用户自定义类型。

比如：我们想定义自己的 `Integer` 类型，并添加一些类似转换成字符串的方法，在 Go 中可以如下定义：

```
type Integer int
func (i *Integer) String() string {
    return strconv.Itoa(int(*i))
}
```

在 Java 或 C# 中，这个方法需要和类 `Integer` 的定义放在一起，在 Ruby 中可以直接在基本类型 int 上定义这个方法。

总结

在 Go 中，类型就是类（数据和关联的方法）。Go 不知道类似面向对象语言的类继承的概念。继承有两个好处：代码复用和多态。

在 Go 中，代码复用通过组合和委托实现，多态通过接口的使用来实现：有时这也叫 **组件编程（Component Programming）**。

许多开发者说相比于类继承，Go 的接口提供了更强大、却更简单的多态行为。

备注

如果真的需要更多面向对象的能力，看一下 `goop` 包（Go Object-Oriented Programming），它由 Scott Pakin 编写: 它给 Go 提供了 JavaScript 风格的对象（基于原型的对象），并且支持多重继承和类型独立分派，通过它可以实现你喜欢的其他编程语言里的一些结构。

问题 10.1

我们在某个类型的变量上使用点号调用一个方法： `variable.method()` ，在使用 Go 以前，在哪儿碰到过面向对象的点号？

问题 10.2

a）假设定义： `type Integer int` ，完成 `get()` 方法的方法体: `func (p Integer) get() int { ... }` 。

b）定义： `func f(i int) {}; var v Integer` ，如何就 v 作为参数调用f?

c）假设 `Integer` 定义为 `type Integer struct {n int}` ，完成 `get()` 方法的方法体： `func (p Integer) get() int { ... }` 。

d）对于新定义的 `Integer` ，和 b）中同样的问题。

类型的 String() 方法和格式化描述符

10.7 类型的 String() 方法和格式化描述符

当定义了一个有很多方法的类型时，十之八九你会使用 `String()` 方法来定制类型的字符串形式的输出，换句话说：一种可阅读性和打印性的输出。如果类型定义了 `String()` 方法，它会被用在 `fmt.Printf()` 中生成默认的输出：等同于使用格式化描述符 `%v` 产生的输出。还有 `fmt.Print()` 和 `fmt.Println()` 也会自动使用 `String()` 方法。

我们使用第 10.4 节中程序的类型来进行测试：

示例 10.22 method_string.go：

```
package main

import (
    "fmt"
    "strconv"
)

type TwoInts struct {
    a int
    b int
}

func main() {
    two1 := new(TwoInts)
    two1.a = 12
    two1.b = 10
    fmt.Printf("two1 is: %v\n", two1)
    fmt.Println("two1 is:", two1)
    fmt.Printf("two1 is: %T\n", two1)
    fmt.Printf("two1 is: %#v\n", two1)
}

func (tn *TwoInts) String() string {
    return "(" + strconv.Itoa(tn.a) + "/" + strconv.Itoa(tn.b) + ")"
}
```

输出：

```
two1 is: (12/10)
two1 is: (12/10)
two1 is: *main.TwoInts
two1 is: &main.TwoInts{a:12, b:10}
```

当你广泛使用一个自定义类型时，最好为它定义 `String()` 方法。从上面的例子也可以看到，格式化描述符 `%T` 会给出类型的完全规格，`%#v` 会给出实例的完整输出，包括它的字段（在程序自动生成 `Go` 代码时也很有用）。

备注

不要在 `String()` 方法里面调用涉及 `String()` 方法的方法，它会导致意料之外的错误，比如下面的例子，它导致了一个无限递归调用（`TT.String()` 调用 `fmt.Sprintf`，而 `fmt.Sprintf` 又会反过来调用 `TT.String()` ...），很快就会导致内存溢出：

```
type TT float64

func (t TT) String() string {
    return fmt.Sprintf("%v", t)
}
t.String()
```

练习 10.12 type_string.go

给定结构体类型 T:

```
type T struct {
    a int
    b float32
    c string
}
```

值 `t` : `t := &T{7, -2.35, "abc\tdef"}` 。给 T 定义 `String()` ，使得 `fmt.Printf("%v\n", t)` 输出： `7 / -2.350000 / "abc\tdef"` 。

练习 10.13 celsius.go

为 float64 定义一个别名类型 `Celsius` ，并给它定义 `String()` ，它输出一个十进制数和 °C 表示的温度值。

练习 10.14 days.go

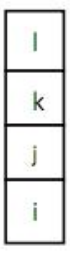
为 int 定义一个别名类型 `Day` ，定义一个字符串数组它包含一周七天的名字，为类型 `Day` 定义 `String()` 方法，它输出星期几的名字。使用 `iota` 定义一个枚举常量用于表示一周的中每天（MO、TU...）。

练习 10.15 timezones.go

为 int 定义别名类型 `TZ` ，定义一些常量表示时区，比如 UTC，定义一个 map，它将时区的缩写映射为它的全称，比如： `UTC -> "Universal Greenwich time"` 。为类型 `TZ` 定义 `String()` 方法，它输出时区的全称。

练习 10.16 stack_arr.go/stack_struct.go

实现栈（stack）数据结构：



它的格子包含数据，比如整数 i、j、k 和 l 等等，格子从底部（索引 0）至顶部（索引 n）来索引。这个例子中假定 `n=3` ，那么一共有 4 个格子。

一个新栈中所有格子的值都是 0。

将一个新值放到栈的最顶部一个空（包括零）的格子中，这叫做 `push` 。

获取栈的最顶部一个非空（非零）的格子的值，这叫做 `pop` 。
现在可以理解为什么栈是一个后进先出（LIFO）的结构了吧。

为栈定义一个 `Stack` 类型，并为它定义 `Push` 和 `Pop` 方法，再为它定义 `String()` 方法（用于调试）输出栈的内容，比如： `[0:i] [1:j] [2:k] [3:l]` 。

1）stack_arr.go：使用长度为 4 的 int 数组作为底层数据结构。

2）stack_struct.go：使用包含一个索引和一个 int 数组的结构体作为底层数据结构，索引表示第一个空闲的位置。

3）使用常量 LIMIT 代替上面表示元素个数的 4 重新实现上面的 1）和 2），使它们更具有一般性。

垃圾回收和 SetFinalizer

10.8 垃圾回收和 SetFinalizer

Go 开发者不需要写代码来释放程序中不再使用的变量和结构占用的内存，在 Go 运行时中有一个独立的进程，即垃圾收集器（GC），会处理这些事情，它搜索不再使用的变量然后释放它们的内存。可以通过 `runtime` 包访问 GC 进程。

通过调用 `runtime.GC()` 函数可以显式的触发 GC，但这只在某些罕见的场景下才有用，比如当内存资源不足时调用 `runtime.GC()`，它会在此函数执行的点上立即释放一大片内存，此时程序可能会有短时的性能下降（因为 `GC` 进程在执行）。

如果想知道当前的内存状态，可以使用：

```
// fmt.Printf("%d\n", runtime.MemStats.Alloc/1024)
// 此处代码在 Go 1.5.1下不再有效，更正为
var m runtime.MemStats
runtime.ReadMemStats(&m)
fmt.Printf("%d Kb\n", m.Alloc / 1024)
```

上面的程序会给出已分配内存的总量，单位是 Kb。进一步的测量参考 文档页面。

如果需要在一个对象 obj 被从内存移除前执行一些特殊操作，比如写到日志文件中，可以通过如下方式调用函数来实现：

```
runtime.SetFinalizer(obj, func(obj *typeObj))
```

`func(obj *typeObj)` 需要一个 `typeObj` 类型的指针参数 `obj`，特殊操作会在它上面执行。`func` 也可以是一个匿名函数。

在对象被 GC 进程选中并从内存中移除以前，`SetFinalizer` 都不会执行，即使程序正常结束或者发生错误。

练习 10.17

从练习 10.16 开始（它基于结构体实现了一个栈结构），为栈的实现（stack_struct.go）创建一个单独的包 `stack`，并从 `main` 包 `main.stack.go` 中调用它。

接口（interface）与反射（reflection）

11.0 接口（interface）与反射（reflection）

本章介绍 Go 语言中接口和反射的相关内容。

接口是什么

11.1 接口是什么

Go 语言不是一种“*传统*”的面向对象编程语言：它里面没有类和继承的概念。

但是 Go 语言里有非常灵活的 **接口** 概念，通过它可以实现很多面向对象的特性。接口提供了一种方式来 **说明** 对象的行为：如果谁能搞定这件事，它就可以用在这儿。

接口定义了一组方法（方法集），但是这些方法不包含（实现）代码：它们没有被实现（它们是抽象的）。接口里也不能包含变量。

通过如下格式定义接口：

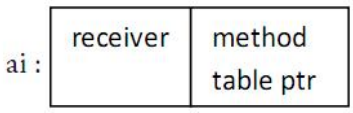
```
type Namer interface {
    Method1(param_list) return_type
    Method2(param_list) return_type
    ...
}
```

上面的 `Namer` 是一个 **接口类型**。

（按照约定，只包含一个方法的）接口的名字由方法名加 `[e]r` 后缀组成，例如 `Printer` 、 `Reader` 、 `Writer` 、 `Logger` 、 `Converter` 等等。还有一些不常用的方式（当后缀 `er` 不合适时），比如 `Recoverable` ，此时接口名以 `able` 结尾，或者以 `I` 开头（像 `.NET` 或 `Java` 中那样）。

Go 语言中的接口都很简短，通常它们会包含 0 个、最多 3 个方法。

不像大多数面向对象编程语言，在 Go 语言中接口可以有值，一个接口类型的变量或一个 **接口值**：`var ai Namer` ， `ai` 是一个多字（multiword）数据结构，它的值是 `nil` 。它本质上是一个指针，虽然不完全是一回事。指向接口值的指针是非法的，它们不仅一点用也没有，还会导致代码错误。



此处的方法指针表是通过运行时反射能力构建的。

类型（比如结构体）实现接口方法集中的方法，每一个方法的实现说明了此方法是如何作用于该类型的：**即实现接口**，同时方法集也构成了该类型的接口。实现了 `Namer` 接口类型的变量可以赋值给 `ai` （接收者值），此时方法表中的指针会指向被实现的接口方法。当然如果另一个类型（也实现了该接口）的变量被赋值给 `ai` ，这二者（译者注：指针和方法实现）也会随之改变。

类型不需要显式声明它实现了某个接口：接口被隐式地实现。多个类型可以实现同一个接口。

实现某个接口的类型（除了实现接口方法外）可以有其他的方法。

一个类型可以实现多个接口。

接口类型可以包含一个实例的引用， 该实例的类型实现了此接口（接口是动态类型）。

即使接口在类型之后才定义，二者处于不同的包中，被单独编译：只要类型实现了接口中的方法，它就实现了此接口。

所有这些特性使得接口具有很大的灵活性。

第一个例子：

示例 11.1 interfaces.go：

```
package main

import "fmt"

type Shaper interface {
    Area() float32
}

type Square struct {
    side float32
}

func (sq *Square) Area() float32 {
    return sq.side * sq.side
}

func main() {
    sq1 := new(Square)
    sq1.side = 5

    var areaIntf Shaper
    areaIntf = sq1
    // shorter,without separate declaration:
    // areaIntf := Shaper(sq1)
    // or even:
    // areaIntf := sq1
    fmt.Printf("The square has area: %f\n", areaIntf.Area())
}
```

输出：

```
The square has area: 25.000000
```

上面的程序定义了一个结构体 `Square` 和一个接口 `Shaper`，接口有一个方法 `Area()`。

在 `main()` 方法中创建了一个 `Square` 的实例。在主程序外边定义了一个接收者类型是 `Square` 方法的 `Area()`，用来计算正方形的面积：结构体 `Square` 实现了接口 `Shaper`。

所以可以将一个 `Square` 类型的变量赋值给一个接口类型的变量：`areaIntf = sq1`。

现在接口变量包含一个指向 `Square` 变量的引用，通过它可以调用 `Square` 上的方法 `Area()`。当然也可以直接在 `Square` 的实例上调用此方法，但是在接口实例上调用此方法更令人兴奋，它使此方法更具有一般性。接口变量里包含了接收者实例的值和指向对应方法表的指针。

这是 **多态** 的 Go 版本，多态是面向对象编程中一个广为人知的概念：根据当前的类型选择正确的方法，或者说：同一种类型在不同的实例上似乎表现出不同的行为。

如果 `Square` 没有实现 `Area()` 方法，编译器将会给出清晰的错误信息：

```
cannot use sq1 (type *Square) as type Shaper in assignment:
*Square does not implement Shaper (missing Area method)
```

如果 `Shaper` 有另外一个方法 `Perimeter()`，但是 `Square` 没有实现它，即使没有人在 `Square` 实例上调用这个方法，编译器也会给出上面同样的错误。

扩展一下上面的例子，类型 `Rectangle` 也实现了 `Shaper` 接口。接着创建一个 `Shaper` 类型的数组，迭代它的每一个元素并在上面调用 `Area()` 方法，以此来展示多态行为：

示例 11.2 interfaces_poly.go：

```
package main

import "fmt"

type Shaper interface {
    Area() float32
}

type Square struct {
    side float32
}

func (sq *Square) Area() float32 {
    return sq.side * sq.side
}

type Rectangle struct {
    length, width float32
}

func (r Rectangle) Area() float32 {
    return r.length * r.width
}

func main() {

    r := Rectangle{5, 3} // Area() of Rectangle needs a value
    q := &Square{5}      // Area() of Square needs a pointer
    // shapes := []Shaper{Shaper(r), Shaper(q)}
    // or shorter
    shapes := []Shaper{r, q}
    fmt.Println("Looping through shapes for area ...")
    for n, _ := range shapes {
        fmt.Println("Shape details: ", shapes[n])
        fmt.Println("Area of this shape is: ", shapes[n].Area())
    }
}
```

输出：

```
Looping through shapes for area ...
Shape details:  {5 3}
Area of this shape is:  15
Shape details:  &{5}
Area of this shape is:  25
```

在调用 `shapes[n].Area()` 这个时，只知道 `shapes[n]` 是一个 `Shaper` 对象，最后它摇身一变成为了一个 `Square` 或 `Rectangle` 对象，并且表现出了相对应的行为。

也许从现在开始你将看到通过接口如何产生 **更干净、更简单** 及 **更具有扩展性** 的代码。在 11.12.3 中将看到在开发中为类型添加新的接口是多么的容易。

下面是一个更具体的例子：有两个类型 `stockPosition` 和 `car`，它们都有一个 `getValue()` 方法，我们可以定义一个具有此方法的接口 `valuable`。接着定义一个使用 `valuable` 类型作为参数的函数 `showValue()`，所有实现了 `valuable` 接口的类型都可以用这个函数。

示例 11.3 valuable.go：

```
package main

import "fmt"

type stockPosition struct {
    ticker     string
    sharePrice float32
    count      float32
}

/* method to determine the value of a stock position */
func (s stockPosition) getValue() float32 {
    return s.sharePrice * s.count
}

type car struct {
    make  string
    model string
    price float32
}

/* method to determine the value of a car */
func (c car) getValue() float32 {
    return c.price
}

/* contract that defines different things that have value */
type valuable interface {
    getValue() float32
}

func showValue(asset valuable) {
    fmt.Printf("Value of the asset is %f\n", asset.getValue())
}

func main() {
    var o valuable = stockPosition{"GOOG", 577.20, 4}
    showValue(o)
    o = car{"BMW", "M3", 66500}
    showValue(o)
}
```

输出：

```
Value of the asset is 2308.800049
Value of the asset is 66500.000000
```

一个标准库的例子

`io` 包里有一个接口类型 `Reader`：

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

定义变量 `r` ： `var r io.Reader`

那么就可以写如下的代码：

```
    var r io.Reader
    r = os.Stdin     // see 12.1
    r = bufio.NewReader(r)
    r = new(bytes.Buffer)
    f,_ := os.Open("test.txt")
    r = bufio.NewReader(f)
```

上面 `r` 右边的类型都实现了 `Read()` 方法，并且有相同的方法签名， `r` 的静态类型是 `io.Reader` 。

备注

有的时候，也会以一种稍微不同的方式来使用接口这个词：从某个类型的角度来看，它的接口指的是：它的所有导出方法，只不过没有显式地为这些导出方法额外定一个接口而已。

练习 11.1 simple_interface.go：

定义一个接口 `Simpler` ，它有一个 `Get()` 方法和一个 `Set()` ， `Get()` 返回一个整型值， `Set()` 有一个整型参数。创建一个结构体类型 `Simple` 实现这个接口。

接着定一个函数，它有一个 `Simpler` 类型的参数，调用参数的 `Get()` 和 `Set()` 方法。在 `main` 函数里调用这个函数，看看它是否可以正确运行。

练习 11.2 interfaces_poly2.go：

a) 扩展 interfaces_poly.go 中的例子，添加一个 `Circle` 类型

b) 使用一个抽象类型 `Shape` （没有字段） 实现同样的功能，它实现接口 `Shaper` ，然后在其他类型里内嵌此类型。扩展 10.6.5 中的例子来说明覆写。

接口嵌套接口

11.2 接口嵌套接口

一个接口可以包含一个或多个其他的接口，这相当于直接将这些内嵌接口的方法列举在外层接口中一样。

比如接口 `File` 包含了 `ReadWrite` 和 `Lock` 的所有方法，它还额外有一个 `Close()` 方法。

```
type ReadWrite interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
}

type Lock interface {
    Lock()
    Unlock()
}

type File interface {
    ReadWrite
    Lock
    Close()
}
```

类型断言：如何检测和转换接口变量的类型

11.3 类型断言：如何检测和转换接口变量的类型

一个接口类型的变量 `varI` 中可以包含任何类型的值，必须有一种方式来检测它的 **动态** 类型，即运行时在变量中存储的值的实际类型。在执行过程中动态类型可能会有所不同，但是它总是可以分配给接口变量本身的类型。通常我们可以使用 **类型断言** 来测试在某个时刻 `varI` 是否包含类型 `T` 的值：

```
v := varI.(T)       // unchecked type assertion
```

varI 必须是一个接口变量，否则编译器会报错： `invalid type assertion: varI.(T) (non-interface type (type of varI) on left)` 。

类型断言可能是无效的，虽然编译器会尽力检查转换是否有效，但是它不可能预见所有的可能性。如果转换在程序运行时失败会导致错误发生。更安全的方式是使用以下形式来进行类型断言：

```
if v, ok := varI.(T); ok {  // checked type assertion
    Process(v)
    return
}
// varI is not of type T
```

如果转换合法， `v` 是 `varI` 转换到类型 `T` 的值， `ok` 会是 `true` ；否则 `v` 是类型 `T` 的零值， `ok` 是 `false` ，也没有运行时错误发生。

应该总是使用上面的方式来进行类型断言。

多数情况下，我们可能只是想在 `if` 中测试一下 `ok` 的值，此时使用以下的方法会是最方便的：

```
if _, ok := varI.(T); ok {
    // ...
}
```

示例 11.4 type_interfaces.go：

```
package main

import (
    "fmt"
    "math"
)

type Square struct {
    side float32
}

type Circle struct {
    radius float32
}

type Shaper interface {
    Area() float32
}
```

```
func main() {
    var areaIntf Shaper
    sq1 := new(Square)
    sq1.side = 5

    areaIntf = sq1
    // Is Square the type of areaIntf?
    if t, ok := areaIntf.(*Square); ok {
        fmt.Printf("The type of areaIntf is: %T\n", t)
    }
    if u, ok := areaIntf.(*Circle); ok {
        fmt.Printf("The type of areaIntf is: %T\n", u)
    } else {
        fmt.Println("areaIntf does not contain a variable of type Circle")
    }
}

func (sq *Square) Area() float32 {
    return sq.side * sq.side
}

func (ci *Circle) Area() float32 {
    return ci.radius * ci.radius * math.Pi
}
```

输出：

```
The type of areaIntf is: *main.Square
areaIntf does not contain a variable of type Circle
```

程序中定义了一个新类型 `Circle`，它也实现了 `Shaper` 接口。 `if t, ok := areaIntf.(*Square); ok` 测试 `areaIntf` 里是否有一个包含 `*Square` 类型的变量，结果是确定的；然后我们测试它是否包含一个 `*Circle` 类型的变量，结果是否定的。

备注

如果忽略 `areaIntf.(*Square)` 中的 `*` 号，会导致编译错误： `impossible type assertion: Square does not implement Shaper (Area method has pointer receiver)` 。

类型判断：type-switch

11.4 类型判断：type-switch

接口变量的类型也可以使用一种特殊形式的 `switch` 来检测：**type-switch**（下面是示例 11.4 的第二部分）：

```
switch t := areaIntf.(type) {
case *Square:
    fmt.Printf("Type Square %T with value %v\n", t, t)
case *Circle:
    fmt.Printf("Type Circle %T with value %v\n", t, t)
case nil:
    fmt.Printf("nil value: nothing to check?\n")
default:
    fmt.Printf("Unexpected type %T\n", t)
}
```

输出：

```
Type Square *main.Square with value &{5}
```

变量 `t` 得到了 `areaIntf` 的值和类型，所有 `case` 语句中列举的类型（`nil` 除外）都必须实现对应的接口（在上例中即 `Shaper`），如果被检测类型没有在 `case` 语句列举的类型中，就会执行 `default` 语句。

可以用 `type-switch` 进行运行时类型分析，但是在 `type-switch` 不允许有 `fallthrough` 。

如果仅仅是测试变量的类型，不用它的值，那么就可以不需要赋值语句，比如：

```
switch areaIntf.(type) {
case *Square:
    // TODO
case *Circle:
    // TODO
...
default:
    // TODO
}
```

下面的代码片段展示了一个类型分类函数，它有一个可变长度参数，可以是任意类型的数组，它会根据数组元素的实际类型执行不同的动作：

```
func classifier(items ...interface{}) {
    for i, x := range items {
        switch x.(type) {
        case bool:
            fmt.Printf("Param #%d is a bool\n", i)
        case float64:
            fmt.Printf("Param #%d is a float64\n", i)
        case int, int64:
            fmt.Printf("Param #%d is a int\n", i)
        case nil:
            fmt.Printf("Param #%d is a nil\n", i)
```

```
        case string:
            fmt.Printf("Param #%d is a string\n", i)
        default:
            fmt.Printf("Param #%d is unknown\n", i)
        }
    }
}
```

可以这样调用此方法： `classifier(13, -14.3, "BELGIUM", complex(1, 2), nil, false)` 。

在处理来自于外部的、类型未知的数据时，比如解析诸如 JSON 或 XML 编码的数据，类型测试和转换会非常有用。

在示例 12.17（xml.go）中解析 XML 文档时，我们就会用到 `type-switch` 。

练习 11.4 simple_interface2.go：

接着练习 11.1 中的内容，创建第二个类型 `RSimple` ，它也实现了接口 `Simpler` ，写一个函数 `fi` ，使它可以区分 `Simple` 和 `RSimple` 类型的变量。

测试一个值是否实现了某个接口

11.5 测试一个值是否实现了某个接口

这是 11.3 类型断言中的一个特例：假定 `v` 是一个值，然后我们想测试它是否实现了 `Stringer` 接口，可以这样做：

```
type Stringer interface {
    String() string
}

if sv, ok := v.(Stringer); ok {
    fmt.Printf("v implements String(): %s\n", sv.String()) // note: sv, not v
}
```

`Print` 函数就是如此检测类型是否可以打印自身的。

接口是一种契约，实现类型必须满足它，它描述了类型的行为，规定类型可以做什么。接口彻底将类型能做什么，以及如何做分离开来，使得相同接口的变量在不同的时刻表现出不同的行为，这就是多态的本质。

编写参数是接口变量的函数，这使得它们更具有一般性。

使用接口使代码更具有普适性。

标准库里到处都使用了这个原则，如果对接口概念没有良好的把握，是不可能理解它是如何构建的。

在接下来的章节中，我们会讨论两个重要的例子，试着去深入理解它们，这样你就可以更好的应用上面的原则。

使用方法集与接口

11.6 使用方法集与接口

在第 10.6.3 节及例子 methodset1.go 中我们看到，作用于变量上的方法实际上是不区分变量到底是指针还是值的。当碰到接口类型值时，这会变得有点复杂，原因是接口变量中存储的具体值是不可寻址的，幸运的是，如果使用不当编译器会给出错误。考虑下面的程序：

示例 11.5 methodset2.go：

```
package main

import (
    "fmt"
)

type List []int

func (l List) Len() int {
    return len(l)
}

func (l *List) Append(val int) {
    *l = append(*l, val)
}

type Appender interface {
    Append(int)
}

func CountInto(a Appender, start, end int) {
    for i := start; i <= end; i++ {
        a.Append(i)
    }
}

type Lener interface {
    Len() int
}

func LongEnough(l Lener) bool {
    return l.Len()*10 > 42
}

func main() {
    // A bare value
    var lst List
    // compiler error:
    // cannot use lst (type List) as type Appender in argument to CountInto:
    //       List does not implement Appender (Append method has pointer receiver)
    // CountInto(lst, 1, 10)
    if LongEnough(lst) { // VALID:Identical receiver type
        fmt.Printf("- lst is long enough\n")
    }

    // A pointer value
```

```
    plst := new(List)
    CountInto(plst, 1, 10) //VALID:Identical receiver type
    if LongEnough(plst) {
        // VALID: a *List can be dereferenced for the receiver
        fmt.Printf("- plst is long enough\n")
    }
}
```

讨论

在 `lst` 上调用 `CountInto` 时会导致一个编译器错误，因为 `CountInto` 需要一个 `Appender`，而它的方法 `Append` 只定义在指针上。 在 `lst` 上调用 `LongEnough` 是可以的，因为 `Len` 定义在值上。

在 `plst` 上调用 `CountInto` 是可以的，因为 `CountInto` 需要一个 `Appender`，并且它的方法 `Append` 定义在指针上。 在 `plst` 上调用 `LongEnough` 也是可以的，因为指针会被自动解引用。

总结

在接口上调用方法时，必须有和方法定义时相同的接收者类型或者是可以从具体类型 `P` 直接可以辨识的：

- 指针方法可以通过指针调用
- 值方法可以通过值调用
- 接收者是值的方法可以通过指针调用，因为指针会首先被解引用
- 接收者是指针的方法不可以通过值调用，因为存储在接口中的值没有地址

将一个值赋值给一个接口时，编译器会确保所有可能的接口方法都可以在此值上被调用，因此不正确的赋值在编译期就会失败。

译注

Go 语言规范定义了接口方法集的调用规则：

- 类型 T 的可调用方法集包含接受者为 *T 或 T 的所有方法集
- 类型 *T 的可调用方法集包含接受者为 *T 的所有方法
- 类型 *T 的可调用方法集不包含接受者为 T 的方法

第一个例子：使用 Sorter 接口排序

11.7 第一个例子：使用 Sorter 接口排序

一个很好的例子是来自标准库的 `sort` 包，要对一组数字或字符串排序，只需要实现三个方法：反映元素个数的 `Len()` 方法、比较第 `i` 和 `j` 个元素的 `Less(i, j)` 方法以及交换第 `i` 和 `j` 个元素的 `Swap(i, j)` 方法。

排序函数的算法只会使用到这三个方法（可以使用任何排序算法来实现，此处我们使用冒泡排序）：

```
func Sort(data Sorter) {
    for pass := 1; pass < data.Len(); pass++ {
        for i := 0;i < data.Len() - pass; i++ {
            if data.Less(i+1, i) {
                data.Swap(i, i + 1)
            }
        }
    }
}
```

`Sort` 函数接收一个接口类型的参数：`Sorter`，它声明了这些方法：

```
type Sorter interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

参数中的 `int` 是待排序序列长度的类型，而不是说要排序的对象一定要是一组 `int`。`i` 和 `j` 表示元素的整型索引，长度也是整型的。

现在如果我们想对一个 `int` 数组进行排序，所有必须做的事情就是：为数组定一个类型并在它上面实现 `Sorter` 接口的方法：

```
type IntArray []int
func (p IntArray) Len() int           { return len(p) }
func (p IntArray) Less(i, j int) bool { return p[i] < p[j] }
func (p IntArray) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
```

下面是调用排序函数的一个具体例子：

```
data := []int{74, 59, 238, -784, 9845, 959, 905, 0, 0, 42, 7586, -5467984, 7586}
a := sort.IntArray(data) //conversion to type IntArray from package sort
sort.Sort(a)
```

完整的、可运行的代码可以在 `sort.go` 和 `sortmain.go` 里找到。

同样的原理，排序函数可以用于一个浮点型数组，一个字符串数组，或者一个表示每周各天的结构体 `dayArray`。

示例 11.6 sort.go：

```
package sort

type Sorter interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}

func Sort(data Sorter) {
    for pass := 1; pass < data.Len(); pass++ {
        for i := 0; i < data.Len()-pass; i++ {
            if data.Less(i+1, i) {
                data.Swap(i, i+1)
            }
        }
    }
}

func IsSorted(data Sorter) bool {
    n := data.Len()
    for i := n - 1; i > 0; i-- {
        if data.Less(i, i-1) {
            return false
        }
    }
    return true
}

// Convenience types for common cases
type IntArray []int

func (p IntArray) Len() int           { return len(p) }
func (p IntArray) Less(i, j int) bool { return p[i] < p[j] }
func (p IntArray) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }

type StringArray []string

func (p StringArray) Len() int           { return len(p) }
func (p StringArray) Less(i, j int) bool { return p[i] < p[j] }
func (p StringArray) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }

// Convenience wrappers for common cases
func SortInts(a []int)       { Sort(IntArray(a)) }
func SortStrings(a []string) { Sort(StringArray(a)) }

func IntsAreSorted(a []int) bool       { return IsSorted(IntArray(a)) }
func StringsAreSorted(a []string) bool { return IsSorted(StringArray(a)) }
```

示例 11.7 sortmain.go：

```
package main

import (
    "./sort"
    "fmt"
)

func ints() {
    data := []int{74, 59, 238, -784, 9845, 959, 905, 0, 0, 42, 7586, -5467984, 7586}
    a := sort.IntArray(data) //conversion to type IntArray
```

```
    sort.Sort(a)
    if !sort.IsSorted(a) {
        panic("fails")
    }
    fmt.Printf("The sorted array is: %v\n", a)
}

func strings() {
    data := []string{"monday", "friday", "tuesday", "wednesday", "sunday", "thursday", "", "saturday"}
    a := sort.StringArray(data)
    sort.Sort(a)
    if !sort.IsSorted(a) {
        panic("fail")
    }
    fmt.Printf("The sorted array is: %v\n", a)
}

type day struct {
    num       int
    shortName string
    longName  string
}

type dayArray struct {
    data []*day
}

func (p *dayArray) Len() int           { return len(p.data) }
func (p *dayArray) Less(i, j int) bool { return p.data[i].num < p.data[j].num }
func (p *dayArray) Swap(i, j int)      { p.data[i], p.data[j] = p.data[j], p.data[i] }

func days() {
    Sunday    := day{0, "SUN", "Sunday"}
    Monday    := day{1, "MON", "Monday"}
    Tuesday   := day{2, "TUE", "Tuesday"}
    Wednesday := day{3, "WED", "Wednesday"}
    Thursday  := day{4, "THU", "Thursday"}
    Friday    := day{5, "FRI", "Friday"}
    Saturday  := day{6, "SAT", "Saturday"}
    data := []*day{&Tuesday, &Thursday, &Wednesday, &Sunday, &Monday, &Friday, &Saturday}
    a := dayArray{data}
    sort.Sort(&a)
    if !sort.IsSorted(&a) {
        panic("fail")
    }
    for _, d := range data {
        fmt.Printf("%s ", d.longName)
    }
    fmt.Printf("\n")
}

func main() {
    ints()
    strings()
    days()
}
```

输出：

```
The sorted array is: [-5467984 -784 0 0 42 59 74 238 905 959 7586 7586 9845]
The sorted array is: [ friday monday saturday sunday thursday tuesday wednesday]
Sunday Monday Tuesday Wednesday Thursday Friday Saturday
```

备注：

`panic("fail")` 用于停止处于在非正常情况下的程序（详细请参考 第13章），当然也可以先打印一条信息，然后调用 `os.Exit(1)` 来停止程序。

上面的例子帮助我们进一步了解了接口的意义和使用方式。对于基本类型的排序，标准库已经提供了相关的排序函数，所以不需要我们再重复造轮子了。对于一般性的排序， `sort` 包定义了一个接口：

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

这个接口总结了需要用于排序的抽象方法，函数 `Sort(data Interface)` 用来对此类对象进行排序，可以用它们来实现对其他类型的数据（非基本类型）进行排序。在上面的例子中，我们也是这么做的，不仅可以对 `int` 和 `string` 序列进行排序，也可以对用户自定义类型 `dayArray` 进行排序。

练习 11.5 interfaces_ext.go：

a). 继续扩展程序，定义类型 `Triangle` ，让它实现 `AreaInterface` 接口。通过计算一个特定三角形的面积来进行测试（三角形面积=0.5 * (底 * 高)）

b). 定义一个新接口 `PeriInterface` ，它有一个 `Perimeter` 方法。让 `Square` 实现这个接口，并通过一个 `Square` 示例来测试它。

练习 11.6 point_interfaces.go：

继续 10.3 中的练习 point_methods.go，定义接口 `Magnitude` ，它有一个方法 `Abs()` 。让 `Point` 、 `Point3` 及 `Polar` 实现此接口。通过接口类型变量使用方法做point.go中同样的事情。

练习 11.7 float_sort.go / float_sortmain.go：

类似11.7和示例11.3/4，定义一个包 `float64` ，并在包里定义类型 `Float64Array` ，然后让它实现 `Sorter` 接口用来对 `float64` 数组进行排序。

另外提供如下方法：

- `NewFloat64Array()` ：创建一个包含25个元素的数组变量（参考10.2）
- `List()` ：返回数组格式化后的字符串，并在 `String()` 方法中调用它，这样就不用显式地调用 `List()` 来打印数组（参考10.7）
- `Fill()` ：创建一个包含10个随机浮点数的数组（参考4.5.2.6）

在主程序中新建一个此类型的变量，然后对它排序并进行测试。

练习 11.8 sort.go/sort_persons.go：

定义一个结构体 `Person` ，它有两个字段： `firstName` 和 `lastName` ，为 `[]Person` 定义类型 `Persons` 。让 `Persons` 实现 `Sorter` 接口并进行测试。

第二个例子：读和写

11.8 第二个例子：读和写

读和写是软件中很普遍的行为，提起它们会立即想到读写文件、缓存（比如字节或字符串切片）、标准输入输出、标准错误以及网络连接、管道等等，或者读写我们的自定义类型。为了让代码尽可能通用，Go 采取了一致的方式来读写数据。

`io` 包提供了用于读和写的接口 `io.Reader` 和 `io.Writer`：

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

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

只要类型实现了读写接口，提供 `Read()` 和 `Write` 方法，就可以从它读取数据，或向它写入数据。一个对象要是可读的，它必须实现 `io.Reader` 接口，这个接口只有一个签名是 `Read(p []byte) (n int, err error)` 的方法，它从调用它的对象上读取数据，并把读到的数据放入参数中的字节切片中，然后返回读取的字节数和一个 `error` 对象，如果没有错误发生返回 `nil`，如果已经到达输入的尾端，会返回 `io.EOF("EOF")`，如果读取的过程中发生了错误，就会返回具体的错误信息。类似地，一个对象要是可写的，它必须实现 `io.Writer` 接口，这个接口也只有一个签名是 `Write(p []byte) (n int, err error)` 的方法，它将指定字节切片中的数据写入调用它的对象里，然后返回实际写入的字节数和一个 `error` 对象（如果没有错误发生就是 `nil`）。

`io` 包里的 `Readers` 和 `Writers` 都是不带缓冲的，`bufio` 包里提供了对应的带缓冲的操作，在读写 `UTF-8` 编码的文本文件时它们尤其有用。在 第12章 我们会看到很多在实战中使用它们的例子。

在实际编程中尽可能的使用这些接口，会使程序变得更通用，可以在任何实现了这些接口的类型上使用读写方法。

例如一个 `JPEG` 图形解码器，通过一个 `Reader` 参数，它可以解码来自磁盘、网络连接或以 `gzip` 压缩的 `HTTP` 流中的 `JPEG` 图形数据，或者其他任何实现了 `Reader` 接口的对象。

空接口

11.9 空接口

11.9.1 概念

空接口或者最小接口 不包含任何方法，它对实现不做任何要求：

```
type Any interface {}
```

任何其他类型都实现了空接口（它不仅仅像 `Java/C#` 中 `Object` 引用类型），`any` 或 `Any` 是空接口一个很好的别名或缩写。

空接口类似 `Java/C#` 中所有类的基类：`Object` 类，二者的目标也很相近。

可以给一个空接口类型的变量 `var val interface {}` 赋任何类型的值。

示例 11.8 empty_interface.go：

```
package main
import "fmt"

var i = 5
var str = "ABC"

type Person struct {
    name string
    age  int
}

type Any interface{}

func main() {
    var val Any
    val = 5
    fmt.Printf("val has the value: %v\n", val)
    val = str
    fmt.Printf("val has the value: %v\n", val)
    pers1 := new(Person)
    pers1.name = "Rob Pike"
    pers1.age = 55
    val = pers1
    fmt.Printf("val has the value: %v\n", val)
    switch t := val.(type) {
    case int:
        fmt.Printf("Type int %T\n", t)
    case string:
        fmt.Printf("Type string %T\n", t)
    case bool:
        fmt.Printf("Type boolean %T\n", t)
    case *Person:
        fmt.Printf("Type pointer to Person %T\n", t)
    default:
        fmt.Printf("Unexpected type %T", t)
```

```
    }
}
```

输出：

```
val has the value: 5
val has the value: ABC
val has the value: &{Rob Pike 55}
Type pointer to Person *main.Person
```

在上面的例子中，接口变量 `val` 被依次赋予一个 `int`，`string` 和 `Person` 实例的值，然后使用 `type-switch` 来测试它的实际类型。每个 `interface {}` 变量在内存中占据两个字长：一个用来存储它包含的类型，另一个用来存储它包含的数据或者指向数据的指针。

示例 emptyint_switch.go 说明了空接口在 `type-switch` 中联合 `lambda` 函数的用法：

```
package main

import "fmt"

type specialString string

var whatIsThis specialString = "hello"

func TypeSwitch() {
    testFunc := func(any interface{}) {
        switch v := any.(type) {
        case bool:
            fmt.Printf("any %v is a bool type", v)
        case int:
            fmt.Printf("any %v is an int type", v)
        case float32:
            fmt.Printf("any %v is a float32 type", v)
        case string:
            fmt.Printf("any %v is a string type", v)
        case specialString:
            fmt.Printf("any %v is a special String!", v)
        default:
            fmt.Println("unknown type!")
        }
    }
    testFunc(whatIsThis)
}

func main() {
    TypeSwitch()
}
```

输出：

```
any hello is a special String!
```

练习 11.9 simple_interface3.go：

继续 练习11.2，在它中添加一个 `gI` 函数，它不再接受 `Simpler` 类型的参数，而是接受一个空接口参数。然后通过类型断言判断参数是否是 `Simpler` 类型。最后在 `main` 使用 `gI` 取代 `fI` 函数并调用它。确保你的代码足够安全。

11.9.2 构建通用类型或包含不同类型变量的数组

在 7.6.6 中我们看到了能被搜索和排序的 `int` 数组、 `float` 数组以及 `string` 数组，那么对于其他类型的数组呢，是不是我们必须得自己编程实现它们？

现在我们知道该怎么做了，就是通过使用空接口。让我们给空接口定一个别名类型 `Element` ： `type Element interface{}`

然后定义一个容器类型的结构体 `Vector` ，它包含一个 `Element` 类型元素的切片：

```
type Vector struct {
    a []Element
}
```

`Vector` 里能放任何类型的变量，因为任何类型都实现了空接口，实际上 `Vector` 里放的每个元素可以是不同类型的变量。我们为它定义一个 `At()` 方法用于返回第 `i` 个元素：

```
func (p *Vector) At(i int) Element {
    return p.a[i]
}
```

再定一个 `Set()` 方法用于设置第 `i` 个元素的值：

```
func (p *Vector) Set(i int, e Element) {
    p.a[i] = e
}
```

`Vector` 中存储的所有元素都是 `Element` 类型，要得到它们的原始类型（unboxing：拆箱）需要用到类型断言。TODO：The compiler rejects assertions guaranteed to fail，类型断言总是在运行时才执行，因此它会产生运行时错误。

练习 11.10 min_interface.go / minmain.go：

仿照11.7中开发的 `Sorter` 接口，创建一个 `Miner` 接口并实现一些必要的操作。函数 `Min` 接受一个 `Miner` 类型变量的集合，然后计算并返回集合中最小的元素。

11.9.3 复制数据切片至空接口切片

假设你有一个 `myType` 类型的数据切片，你想将切片中的数据复制到一个空接口切片中，类似：

```
var dataSlice []myType = FuncReturnSlice()
var interfaceSlice []interface{} = dataSlice
```

可惜不能这么做，编译时会出错： `cannot use dataSlice (type []myType) as type []interface { } in assignment` 。

原因是它们俩在内存中的布局是不一样的（参考 Go wiki）。

必须使用 `for-range` 语句来一个一个显式地赋值：

```
var dataSlice []myType = FuncReturnSlice()
var interfaceSlice []interface{} = make([]interface{}, len(dataSlice))
for i, d := range dataSlice {
    interfaceSlice[i] = d
}
```

11.9.4 通用类型的节点数据结构

在10.1中我们遇到了诸如列表和树这样的数据结构，在它们的定义中使用了一种叫节点的递归结构体类型，节点包含一个某种类型的数据字段。现在可以使用空接口作为数据字段的类型，这样我们就能写出通用的代码。下面是实现一个二叉树的部分代码：通用定义、用于创建空节点的 `NewNode` 方法，及设置数据的 `SetData` 方法。

示例 11.10 node_structures.go：

```
package main

import "fmt"

type Node struct {
    le   *Node
    data interface{}
    ri   *Node
}

func NewNode(left, right *Node) *Node {
    return &Node{left, nil, right}
}

func (n *Node) SetData(data interface{}) {
    n.data = data
}

func main() {
    root := NewNode(nil, nil)
    root.SetData("root node")
    // make child (leaf) nodes:
    a := NewNode(nil, nil)
    a.SetData("left node")
    b := NewNode(nil, nil)
    b.SetData("right node")
    root.le = a
    root.ri = b
    fmt.Printf("%v\n", root) // Output: &{0x125275f0 root node 0x125275e0}
}
```

11.9.5 接口到接口

一个接口的值可以赋值给另一个接口变量，只要底层类型实现了必要的方法。这个转换是在运行时进行检查的，转换失败会导致一个运行时错误：这是 `Go` 语言动态的一面，可以拿它和 `Ruby` 和 `Python` 这些动态语言相比较。

假定：

```
var ai AbsInterface // declares method Abs()
type SqrInterface interface {
    Sqr() float
}
var si SqrInterface
pp := new(Point) // say *Point implements Abs, Sqr
var empty interface{}
```

那么下面的语句和类型断言是合法的：

```
empty = pp                // everything satisfies empty
ai = empty.(AbsInterface) // underlying value pp implements Abs()
// (runtime failure otherwise)
si = ai.(SqrInterface) // *Point has Sqr() even though AbsInterface doesn't
empty = si             // *Point implements empty set
// Note: statically checkable so type assertion not necessary.
```

下面是函数调用的一个例子：

```
type myPrintInterface interface {
    print()
}

func f3(x myInterface) {
    x.(myPrintInterface).print() // type assertion to myPrintInterface
}
```

`x` 转换为 `myPrintInterface` 类型是完全动态的：只要 `x` 的底层类型（动态类型）定义了 `print` 方法这个调用就可以正常运行（译注：若 `x` 的底层类型未定义 `print` 方法，此处类型断言会导致 `panic`，最佳实践应该为 `if mpi, ok := x.(myPrintInterface); ok { mpi.print() }`，参考 11.3 章节）。

反射包

11.10 反射包

11.10.1 方法和类型的反射

在 10.4 节我们看到可以通过反射来分析一个结构体。本节我们进一步探讨强大的反射功能。反射是用程序检查其所拥有的结构，尤其是类型的一种能力；这是元编程的一种形式。反射可以在运行时检查类型和变量，例如它的大小、方法和 `动态` 的调用这些方法。这对于没有源代码的包尤其有用。这是一个强大的工具，除非真得有必要，否则应当避免使用或小心使用。

变量的最基本信息就是类型和值：反射包的 `Type` 用来表示一个 Go 类型，反射包的 `Value` 为 Go 值提供了反射接口。

两个简单的函数，`reflect.TypeOf` 和 `reflect.ValueOf`，返回被检查对象的类型和值。例如，x 被定义为：`var x float64 = 3.4`，那么 `reflect.TypeOf(x)` 返回 `float64`，`reflect.ValueOf(x)` 返回 `<float64 Value>`

实际上，反射是通过检查一个接口的值，变量首先被转换成空接口。这从下面两个函数签名能够很明显的看出来：

```
func TypeOf(i interface{}) Type
func ValueOf(i interface{}) Value
```

接口的值包含一个 type 和 value。

反射可以从接口值反射到对象，也可以从对象反射回接口值。

reflect.Type 和 reflect.Value 都有许多方法用于检查和操作它们。一个重要的例子是 Value 有一个 Type 方法返回 reflect.Value 的 Type。另一个是 Type 和 Value 都有 Kind 方法返回一个常量来表示类型：Uint、Float64、Slice 等等。同样 Value 有叫做 Int 和 Float 的方法可以获取存储在内部的值（跟 int64 和 float64 一样）

```
const (
    Invalid Kind = iota
    Bool
    Int
    Int8
    Int16
    Int32
    Int64
    Uint
    Uint8
    Uint16
    Uint32
    Uint64
    Uintptr
    Float32
    Float64
    Complex64
    Complex128
    Array
    Chan
    Func
    Interface
    Map
    Ptr
```

```
    Slice
    String
    Struct
    UnsafePointer
)
```

对于 float64 类型的变量 x，如果 `v:=reflect.ValueOf(x)` ，那么 `v.Kind()` 返回 `reflect.Float64` ，所以下面的表达式是 `true`
`v.Kind() == reflect.Float64`

Kind 总是返回底层类型：

```
type MyInt int
var m MyInt = 5
v := reflect.ValueOf(m)
```

方法 `v.Kind()` 返回 `reflect.Int` 。

变量 v 的 `Interface()` 方法可以得到还原（接口）值，所以可以这样打印 v 的值： `fmt.Println(v.Interface())`

尝试运行下面的代码：

示例 11.11 reflect1.go：

```
// blog: Laws of Reflection
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4
    fmt.Println("type:", reflect.TypeOf(x))
    v := reflect.ValueOf(x)
    fmt.Println("value:", v)
    fmt.Println("type:", v.Type())
    fmt.Println("kind:", v.Kind())
    fmt.Println("value:", v.Float())
    fmt.Println(v.Interface())
    fmt.Printf("value is %5.2e\n", v.Interface())
    y := v.Interface().(float64)
    fmt.Println(y)
}
```

输出：

```
type: float64
value: 3.4
type: float64
kind: float64
value: 3.4
3.4
value is 3.40e+00
3.4
```

x 是一个 float64 类型的值，`reflect.ValueOf(x).Float()` 返回这个 float64 类型的实际值；同样的适用于 `Int()`, `Bool()`, `Complex()`, `String()`

11.10.2 通过反射修改(设置)值

继续前面的例子（参阅 11.9 reflect2.go），假设我们要把 x 的值改为 3.1415。Value 有一些方法可以完成这个任务，但是必须小心使用：`v.SetFloat(3.1415)` 。

这将产生一个错误：`reflect.Value.SetFloat using unaddressable value` 。

为什么会这样呢？问题的原因是 v 不是可设置的（这里并不是说值不可寻址）。是否可设置是 Value 的一个属性，并且不是所有的反射值都有这个属性：可以使用 `CanSet()` 方法测试是否可设置。

在例子中我们看到 `v.CanSet()` 返回 false：`settability of v: false`

当 `v := reflect.ValueOf(x)` 函数通过传递一个 x 拷贝创建了 v，那么 v 的改变并不能更改原始的 x。要想 v 的更改能作用到 x，那就必须传递 x 的地址 `v = reflect.ValueOf(&x)` 。

通过 Type() 我们看到 v 现在的类型是 `*float64` 并且仍然是不可设置的。

要想让其可设置我们需要使用 `Elem()` 函数，这间接的使用指针：`v = v.Elem()`

现在 `v.CanSet()` 返回 true 并且 `v.SetFloat(3.1415)` 设置成功了！

示例 11.12 reflect2.go：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4
    v := reflect.ValueOf(x)
    // setting a value:
    // v.SetFloat(3.1415) // Error: will panic: reflect.Value.SetFloat using unaddressable value
    fmt.Println("settability of v:", v.CanSet())
    v = reflect.ValueOf(&x) // Note: take the address of x.
    fmt.Println("type of v:", v.Type())
    fmt.Println("settability of v:", v.CanSet())
    v = v.Elem()
    fmt.Println("The Elem of v is: ", v)
    fmt.Println("settability of v:", v.CanSet())
    v.SetFloat(3.1415) // this works!
    fmt.Println(v.Interface())
    fmt.Println(v)
}
```

输出：

```
settability of v: false
type of v: *float64
settability of v: false
The Elem of v is:  <float64 Value>
settability of v: true
```

```
3.1415
<float64 Value>
```

反射中有些内容是需要用地址去改变它的状态的。

11.10.3 反射结构

有些时候需要反射一个结构类型。 `NumField()` 方法返回结构内的字段数量；通过一个 for 循环用索引取得每个字段的值 `Field(i)` 。

我们同样能够调用签名在结构上的方法，例如，使用索引 n 来调用： `Method(n).Call(nil)` 。

示例 11.13 reflect_struct.go：

```
package main

import (
    "fmt"
    "reflect"
)

type NotknownType struct {
    s1, s2, s3 string
}

func (n NotknownType) String() string {
    return n.s1 + " - " + n.s2 + " - " + n.s3
}

// variable to investigate:
var secret interface{} = NotknownType{"Ada", "Go", "Oberon"}

func main() {
    value := reflect.ValueOf(secret) // <main.NotknownType Value>
    typ := reflect.TypeOf(secret)    // main.NotknownType
    // alternative:
    //typ := value.Type()  // main.NotknownType
    fmt.Println(typ)
    knd := value.Kind() // struct
    fmt.Println(knd)

    // iterate through the fields of the struct:
    for i := 0; i < value.NumField(); i++ {
        fmt.Printf("Field %d: %v\n", i, value.Field(i))
        // error: panic: reflect.Value.SetString using value obtained using unexported field
        //value.Field(i).SetString("C#")
    }

    // call the first method, which is String():
    results := value.Method(0).Call(nil)
    fmt.Println(results) // [Ada - Go - Oberon]
}
```

输出：

```
main.NotknownType
struct
Field 0: Ada
```

```
Field 1: Go
Field 2: Oberon
[Ada - Go - Oberon]
```

但是如果尝试更改一个值，会得到一个错误：

```
panic: reflect.Value.SetString using value obtained using unexported field
```

这是因为结构中只有被导出字段（首字母大写）才是可设置的；来看下面的例子：

示例 11.14 reflect_struct2.go：

```
package main

import (
    "fmt"
    "reflect"
)

type T struct {
    A int
    B string
}

func main() {
    t := T{23, "skidoo"}
    s := reflect.ValueOf(&t).Elem()
    typeOfT := s.Type()
    for i := 0; i < s.NumField(); i++ {
        f := s.Field(i)
        fmt.Printf("%d: %s %s = %v\n", i,
            typeOfT.Field(i).Name, f.Type(), f.Interface())
    }
    s.Field(0).SetInt(77)
    s.Field(1).SetString("Sunset Strip")
    fmt.Println("t is now", t)
}
```

输出：

```
0: A int = 23
1: B string = skidoo
t is now {77 Sunset Strip}
```

附录 37 深入阐述了反射概念。

Printf 和反射

11.11 Printf 和反射

在 Go 语言的标准库中，前几节所述的反射的功能被大量地使用。举个例子，fmt 包中的 Printf（以及其他格式化输出函数）都会使用反射来分析它的 `...` 参数。

Printf 的函数声明为：

```
func Printf(format string, args ... interface{}) (n int, err error)
```

Printf 中的 `...` 参数为空接口类型。Printf 使用反射包来解析这个参数列表。所以，Printf 能够知道它每个参数的类型。因此格式化字符串中只有%d而没有 %u 和 %ld，因为它知道这个参数是 unsigned 还是 long。这也是为什么 Print 和 Println 在没有格式字符串的情况下还能如此漂亮地输出。

为了让大家更加具体地了解 Printf 中的反射，我们实现了一个简单的通用输出函数。其中使用了 type-switch 来推导参数类型，并根据类型来输出每个参数的值（这里用了 10.7 节中练习 10.13 的部分代码）

示例 11.15 print.go：

```
package main

import (
    "os"
    "strconv"
)

type Stringer interface {
    String() string
}

type Celsius float64

func (c Celsius) String() string {
    return strconv.FormatFloat(float64(c),'f', 1, 64) + " ° C"
}

type Day int

var dayName = []string{"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}

func (day Day) String() string {
    return dayName[day]
}

func print(args ...interface{}) {
    for i, arg := range args {
        if i > 0 {os.Stdout.WriteString(" ")}
        switch a := arg.(type) { // type switch
            case Stringer:    os.Stdout.WriteString(a.String())
            case int:         os.Stdout.WriteString(strconv.Itoa(a))
            case string:      os.Stdout.WriteString(a)
            // more types
            default:          os.Stdout.WriteString("???")
        }
```

```
    }
}

func main() {
    print(Day(1), "was", Celsius(18.36))  // Tuesday was 18.4 °C
}
```

在 12.8 节中我们将阐释 `fmt.Fprintf()` 是怎么运用同样的反射原则的。

接口与动态类型

11.12 接口与动态类型

11.12.1 Go 的动态类型

在经典的面向对象语言（像 C++，Java 和 C#）中数据和方法被封装为 `类` 的概念：类包含它们两者，并且不能剥离。

Go 没有类：数据（结构体或更一般的类型）和方法是一种松耦合的正交关系。

Go 中的接口跟 Java/C# 类似：都是必须提供一个指定方法集的实现。但是更加灵活通用：任何提供了接口方法实现代码的类型都隐式地实现了该接口，而不用显式地声明。

和其它语言相比，Go 是唯一结合了接口值，静态类型检查（是否该类型实现了某个接口），运行时动态转换的语言，并且不需要显式地声明类型是否满足某个接口。该特性允许我们在不改变已有的代码的情况下定义和使用新接口。

接收一个（或多个）接口类型作为参数的函数，其**实参**可以是任何实现了该接口的类型的变量。 `实现了某个接口的类型可以被传给任何以此接口为参数的函数` 。

类似于 Python 和 Ruby 这类动态语言中的 `动态类型（duck typing）` ：这意味着对象可以根据提供的方法被处理（例如，作为参数传递给函数），而忽略它们的实际类型：它们能做什么比它们是什么更重要。

这在程序 duck_dance.go 中得以阐明，函数 DuckDance 接受一个 IDuck 接口类型变量。仅当 DuckDance 被实现了 IDuck 接口的类型调用时程序才能编译通过。

示例 11.16 duck_dance.go：

```
package main

import "fmt"

type IDuck interface {
    Quack()
    Walk()
}

func DuckDance(duck IDuck) {
    for i := 1; i <= 3; i++ {
        duck.Quack()
        duck.Walk()
    }
}

type Bird struct {
    // ...
}

func (b *Bird) Quack() {
    fmt.Println("I am quacking!")
}

func (b *Bird) Walk()  {
    fmt.Println("I am walking!")
}
```

```
func main() {
    b := new(Bird)
    DuckDance(b)
}
```

输出：

```
I am quacking!
I am walking!
I am quacking!
I am walking!
I am quacking!
I am walking!
```

如果 `Bird` 没有实现 `Walk()` （把它注释掉），会得到一个编译错误：

```
cannot use b (type *Bird) as type IDuck in function argument:
*Bird does not implement IDuck (missing Walk method)
```

如果对 `cat` 调用函数 `DuckDance()` ，Go 会提示编译错误，但是 Python 和 Ruby 会以运行时错误结束。

11.12.2 动态方法调用

像 Python，Ruby 这类语言，动态类型是延迟绑定的（在运行时进行）：方法只是用参数和变量简单地调用，然后在运行时才解析（它们很可能有像 `responds_to` 这样的方法来检查对象是否可以响应某个方法，但是这也意味着更大的编码量和更多的测试工作）

Go 的实现与此相反，通常需要编译器静态检查的支持：当变量被赋值给一个接口类型的变量时，编译器会检查其是否实现了该接口的所有函数。如果方法调用作用于像 `interface{}` 这样的“泛型”上，你可以通过类型断言（参见 11.3 节）来检查变量是否实现了相应接口。

例如，你用不同的类型表示 XML 输出流中的不同实体。然后我们为 XML 定义一个如下的“写”接口（甚至可以把它定义为私有接口）：

```
type xmlWriter interface {
    WriteXML(w io.Writer) error
}
```

现在我们可以实现适用于该流类型的任何变量的 `StreamXML` 函数，并用类型断言检查传入的变量是否实现了该接口；如果没有，我们就调用内建的 `encodeToXML` 来完成相应工作：

```
// Exported XML streaming function.
func StreamXML(v interface{}, w io.Writer) error {
    if xw, ok := v.(xmlWriter); ok {
        // It's an  xmlWriter, use method of asserted type.
        return xw.WriteXML(w)
    }
    // No implementation, so we have to use our own function (with perhaps reflection):
    return encodeToXML(v, w)
}

// Internal XML encoding function.
func encodeToXML(v interface{}, w io.Writer) error {
    // ...
}
```

Go 在这里用了和 `gob` 相同的机制：定义了两个接口 `GobEncoder` 和 `GobDecoder` 。这样就允许类型自己实现从流编解码的具体方式；如果没有实现就使用标准的反射方式。

因此 Go 提供了动态语言的优点，却没有其他动态语言在运行时可能发生错误的缺点。

对于动态语言非常重要的单元测试来说，这样即可以减少单元测试的部分需求，又可以发挥相当大的作用。

Go 的接口提高了代码的分离度，改善了代码的复用性，使得代码开发过程中的设计模式更容易实现。用 Go 接口还能实现 `依赖注入模式` 。

11.12.3 接口的提取

`提取接口` 是非常有用的设计模式，可以减少需要的类型和方法数量，而且不需要像传统的基于类的面向对象语言那样维护整个的类层次结构。

Go 接口可以让开发者找出自己写的程序中的类型。假设有一些拥有共同行为的对象，并且开发者想要抽象出这些行为，这时就可以创建一个接口来使用。
我们来扩展 11.1 节的示例 11.2 interfaces_poly.go，假设我们需要一个新的接口 `TopologicalGenus` ，用来给 shape 排序（这里简单地实现为返回 int）。我们需要做的是给想要满足接口的类型实现 `Rank()` 方法：

示例 11.17 multi_interfaces_poly.go：

```
//multi_interfaces_poly.go
package main

import "fmt"

type Shaper interface {
    Area() float32
}

type TopologicalGenus interface {
    Rank() int
}

type Square struct {
    side float32
}

func (sq *Square) Area() float32 {
    return sq.side * sq.side
}

func (sq *Square) Rank() int {
    return 1
}

type Rectangle struct {
    length, width float32
}

func (r Rectangle) Area() float32 {
    return r.length * r.width
}

func (r Rectangle) Rank() int {
    return 2
}

func main() {
```

```
    r := Rectangle{5, 3} // Area() of Rectangle needs a value
    q := &Square{5}      // Area() of Square needs a pointer
    shapes := []Shaper{r, q}
    fmt.Println("Looping through shapes for area ...")
    for n, _ := range shapes {
        fmt.Println("Shape details: ", shapes[n])
        fmt.Println("Area of this shape is: ", shapes[n].Area())
    }
    topgen := []TopologicalGenus{r, q}
    fmt.Println("Looping through topgen for rank ...")
    for n, _ := range topgen {
        fmt.Println("Shape details: ", topgen[n])
        fmt.Println("Topological Genus of this shape is: ", topgen[n].Rank())
    }
}
```

输出：

```
Looping through shapes for area ...
Shape details:  {5 3}
Area of this shape is:  15
Shape details:  &{5}
Area of this shape is:  25
Looping through topgen for rank ...
Shape details:  {5 3}
Topological Genus of this shape is:  2
Shape details:  &{5}
Topological Genus of this shape is:  1
```

所以你不用提前设计出所有的接口； `整个设计可以持续演进，而不用废弃之前的决定` 。类型要实现某个接口，它本身不用改变，你只需要在这个类型上实现新的方法。

11.12.4 显式地指明类型实现了某个接口

如果你希望满足某个接口的类型显式地声明它们实现了这个接口，你可以向接口的方法集中添加一个具有描述性名字的方法。例如：

```
type Fooer interface {
    Foo()
    ImplementsFooer()
}
```

类型 Bar 必须实现 `ImplementsFooer` 方法来满足 `Fooer` 接口，以清楚地记录这个事实。

```
type Bar struct{}
func (b Bar) ImplementsFooer() {}
func (b Bar) Foo() {}
```

大部分代码并不使用这样的约束，因为它限制了接口的实用性。

但是有些时候，这样的约束在大量相似的接口中被用来解决歧义。

11.12.5 空接口和函数重载

在 6.1 节中，我们看到函数重载是不被允许的。在 Go 语言中函数重载可以用可变参数 `...T` 作为函数最后一个参数来实现（参见 6.3 节）。如果我们把 T 换为空接口，那么可以知道任何类型的变量都是满足 T (空接口）类型的，这样就允许我们传递任何数量任何类型的参数给函数，即重载的实际含义。

函数 `fmt.Printf` 就是这样做的：

```
fmt.Printf(format string, a ...interface{}) (n int, errno error)
```

这个函数通过枚举 `slice` 类型的实参动态确定所有参数的类型。并查看每个类型是否实现了 `String()` 方法，如果是就用于产生输出信息。我们可以回到 11.10 节查看这些细节。

11.12.6 接口的继承

当一个类型包含（内嵌）另一个类型（实现了一个或多个接口）的指针时，这个类型就可以使用（另一个类型）所有的接口方法。

例如：

```
type Task struct {
    Command string
    *log.Logger
}
```

这个类型的工厂方法像这样：

```
func NewTask(command string, logger *log.Logger) *Task {
    return &Task{command, logger}
}
```

当 `log.Logger` 实现了 `Log()` 方法后，Task 的实例 task 就可以调用该方法：

```
task.Log()
```

类型可以通过继承多个接口来提供像 `多重继承` 一样的特性：

```
type ReaderWriter struct {
    *io.Reader
    *io.Writer
}
```

上面概述的原理被应用于整个 Go 包，多态用得越多，代码就相对越少（参见 12.8 节）。这被认为是 Go 编程中的重要的最佳实践。

有用的接口可以在开发的过程中被归纳出来。添加新接口非常容易，因为已有的类型不用变动（仅仅需要实现新接口的方法）。已有的函数可以扩展为使用接口类型的约束性参数：通常只有函数签名需要改变。对比基于类的 OO 类型的语言在这种情况下则需要适应整个类层次结构的变化。

练习 11.11：map_function_interface.go：

在练习 7.13 中我们定义了一个 map 函数来使用 int 切片 （map_function.go）。

通过空接口和类型断言，现在我们可以写一个可以应用于许多类型的 `泛型` 的 map 函数，为 int 和 string 构建一个把 int 值加倍和将字符串值与其自身连接（译者注：即 `"abc"` 变成 `"abcabc"` ）的 map 函数 `mapFunc` 。

提示：为了可读性可以定义一个 interface{} 的别名，比如：type obj interface{}

练习 11.12：map_function_interface_var.go：

稍微改变练习 11.11，允许 `mapFunc` 接收不定数量的 items。

练习 11.13：main_stack.go—stack/stack_general.go：

在练习 10.16 和 10.17 中我们开发了一些栈结构类型。但是它们被限制为某种固定的内建类型。现在用一个元素类型是 interface{}（空接口）的切片开发一个通用的栈类型。

实现下面的栈方法：

```
Len() int
IsEmpty() bool
Push(x interface{})
Pop() (interface{}, error)
```

`Pop()` 改变栈并返回最顶部的元素； `Top()` 只返回最顶部元素。

在主程序中构建一个充满不同类型元素的栈，然后弹出并打印所有元素的值。

总结：Go 中的面向对象

11.13 总结：Go 中的面向对象

我们总结一下前面看到的：Go 没有类，而是松耦合的类型、方法对接口的实现。

OO 语言最重要的三个方面分别是：封装，继承和多态，在 Go 中它们是怎样表现的呢？

- 封装（数据隐藏）：和别的 OO 语言有 4 个或更多的访问层次相比，Go 把它简化为了 2 层（参见 4.2 节的可见性规则）：

  1）包范围内的：通过标识符首字母小写， `对象` 只在它所在的包内可见

  2）可导出的：通过标识符首字母大写， `对象` 对所在包以外也可见

类型只拥有自己所在包中定义的方法。

- 继承：用组合实现：内嵌一个（或多个）包含想要的行为（字段和方法）的类型；多重继承可以通过内嵌多个类型实现
- 多态：用接口实现：某个类型的实例可以赋给它所实现的任意接口类型的变量。类型和接口是松耦合的，并且多重继承可以通过实现多个接口实现。Go 接口不是 Java 和 C# 接口的变体，而且接口间是不相关的，并且是大规模编程和可适应的演进型设计的关键。

结构体、集合和高阶函数

11.14 结构体、集合和高阶函数

通常你在应用中定义了一个结构体，那么你也可能需要这个结构体的（指针）对象集合，比如：

```
type Any interface{}
type Car struct {
    Model        string
    Manufacturer string
    BuildYear    int
    // ...
}

type Cars []*Car
```

然后我们就可以使用高阶函数，实际上也就是把函数作为定义所需方法（其他函数）的参数，例如：

1）定义一个通用的 `Process()` 函数，它接收一个作用于每一辆 car 的 f 函数作参数：

```
// Process all cars with the given function f:
func (cs Cars) Process(f func(car *Car)) {
    for _, c := range cs {
        f(c)
    }
}
```

2）在上面的基础上，实现一个查找函数来获取子集合，并在 `Process()` 中传入一个闭包执行（这样就可以访问局部切片 `cars` ）：

```
// Find all cars matching a given criteria.
func (cs Cars) FindAll(f func(car *Car) bool) Cars {

    cars := make([]*Car, 0)
    cs.Process(func(c *Car) {
        if f(c) {
            cars = append(cars, c)
        }
    })
    return cars
}
```

3）实现 Map 功能，产出除 car 对象以外的东西：

```
// Process cars and create new data.
func (cs Cars) Map(f func(car *Car) Any) []Any {
    result := make([]Any, 0)
    ix := 0
    cs.Process(func(c *Car) {
        result[ix] = f(c)
        ix++
    })
```

```
    return result
}
```

现在我们可以定义下面这样的具体查询：

```
allNewBMWs := allCars.FindAll(func(car *Car) bool {
    return (car.Manufacturer == "BMW") && (car.BuildYear > 2010)
})
```

4）我们也可以根据参数返回不同的函数。也许我们想根据不同的厂商添加汽车到不同的集合，但是这（这种映射关系）可能会是会改变的。所以我们可以定义一个函数来产生特定的添加函数和 map 集：

```
func MakeSortedAppender(manufacturers []string)(func(car *Car),map[string]Cars) {
    // Prepare maps of sorted cars.
    sortedCars := make(map[string]Cars)
    for _, m := range manufacturers {
        sortedCars[m] = make([]*Car, 0)
    }
    sortedCars["Default"] = make([]*Car, 0)
    // Prepare appender function:
    appender := func(c *Car) {
        if _, ok := sortedCars[c.Manufacturer]; ok {
            sortedCars[c.Manufacturer] = append(sortedCars[c.Manufacturer], c)
        } else {
            sortedCars["Default"] = append(sortedCars["Default"], c)
        }

    }
    return appender, sortedCars
}
```

现在我们可以用它把汽车分类为独立的集合，像这样：

```
manufacturers := []string{"Ford", "Aston Martin", "Land Rover", "BMW", "Jaguar"}
sortedAppender, sortedCars := MakeSortedAppender(manufacturers)
allUnsortedCars.Process(sortedAppender)
BMWCount := len(sortedCars["BMW"])
```

我们让这些代码在下面的程序 cars.go 中执行：

示例 11.18 cars.go：

```
// cars.go
package main

import (
    "fmt"
)

type Any interface{}
type Car struct {
    Model        string
    Manufacturer string
    BuildYear    int
    // ...
}
type Cars []*Car
```

```
func main() {
    // make some cars:
    ford := &Car{"Fiesta", "Ford", 2008}
    bmw := &Car{"XL 450", "BMW", 2011}
    merc := &Car{"D600", "Mercedes", 2009}
    bmw2 := &Car{"X 800", "BMW", 2008}
    // query:
    allCars := Cars([]*Car{ford, bmw, merc, bmw2})
    allNewBMWs := allCars.FindAll(func(car *Car) bool {
        return (car.Manufacturer == "BMW") && (car.BuildYear > 2010)
    })
    fmt.Println("AllCars: ", allCars)
    fmt.Println("New BMWs: ", allNewBMWs)
    //
    manufacturers := []string{"Ford", "Aston Martin", "Land Rover", "BMW", "Jaguar"}
    sortedAppender, sortedCars := MakeSortedAppender(manufacturers)
    allCars.Process(sortedAppender)
    fmt.Println("Map sortedCars: ", sortedCars)
    BMWCount := len(sortedCars["BMW"])
    fmt.Println("We have ", BMWCount, " BMWs")
}

// Process all cars with the given function f:
func (cs Cars) Process(f func(car *Car)) {
    for _, c := range cs {
        f(c)
    }
}

// Find all cars matching a given criteria.
func (cs Cars) FindAll(f func(car *Car) bool) Cars {
    cars := make([]*Car, 0)

    cs.Process(func(c *Car) {
        if f(c) {
            cars = append(cars, c)
        }
    })
    return cars
}

// Process cars and create new data.
func (cs Cars) Map(f func(car *Car) Any) []Any {
    result := make([]Any, len(cs))
    ix := 0
    cs.Process(func(c *Car) {
        result[ix] = f(c)
        ix++
    })
    return result
}

func MakeSortedAppender(manufacturers []string) (func(car *Car), map[string]Cars) {
    // Prepare maps of sorted cars.
    sortedCars := make(map[string]Cars)

    for _, m := range manufacturers {
        sortedCars[m] = make([]*Car, 0)
    }
    sortedCars["Default"] = make([]*Car, 0)
```

```
    // Prepare appender function:
    appender := func(c *Car) {
        if _, ok := sortedCars[c.Manufacturer]; ok {
            sortedCars[c.Manufacturer] = append(sortedCars[c.Manufacturer], c)
        } else {
            sortedCars["Default"] = append(sortedCars["Default"], c)
        }
    }
    return appender, sortedCars
}
```

输出：

```
AllCars:  [0xf8400038a0 0xf840003bd0 0xf840003ba0 0xf840003b70]
New BMWs:  [0xf840003bd0]
Map sortedCars:  map[Default:[0xf840003ba0] Jaguar:[] Land Rover:[] BMW:[0xf840003bd0 0xf840003b70] Aston Mar
tin:[] Ford:[0xf8400038a0]]
We have  2  BMWs
```

读写数据

12.0 读写数据

除了 fmt 和 os 包，我们还需要用到 bufio 包来处理缓冲的输入和输出。

读取用户的输入

12.1 读取用户的输入

我们如何读取用户的键盘（控制台）输入呢？从键盘和标准输入 `os.Stdin` 读取输入，最简单的办法是使用 `fmt` 包提供的 Scan 和 Sscan 开头的函数。请看以下程序：

示例 12.1 readinput1.go：

```
// 从控制台读取输入:
package main
import "fmt"

var (
    firstName, lastName, s string
    i int
    f float32
    input = "56.12 / 5212 / Go"
    format = "%f / %d / %s"
)

func main() {
    fmt.Println("Please enter your full name: ")
    fmt.Scanln(&firstName, &lastName)
    // fmt.Scanf("%s %s", &firstName, &lastName)
    fmt.Printf("Hi %s %s!\n", firstName, lastName) // Hi Chris Naegels
    fmt.Sscanf(input, format, &f, &i, &s)
    fmt.Println("From the string we read: ", f, i, s)
     // 输出结果: From the string we read: 56.12 5212 Go
}
```

`Scanln` 扫描来自标准输入的文本，将空格分隔的值依次存放到后续的参数内，直到碰到换行。`Scanf` 与其类似，除了 `Scanf` 的第一个参数用作格式字符串，用来决定如何读取。`Sscan` 和以 `Sscan` 开头的函数则是从字符串读取，除此之外，与 `Scanf` 相同。如果这些函数读取到的结果与您预想的不同，您可以检查成功读入数据的个数和返回的错误。

您也可以使用 `bufio` 包提供的缓冲读取（buffered reader）来读取数据，正如以下例子所示：

示例 12.2 readinput2.go：

```
package main
import (
    "fmt"
    "bufio"
    "os"
)

var inputReader *bufio.Reader
var input string
var err error

func main() {
    inputReader = bufio.NewReader(os.Stdin)
    fmt.Println("Please enter some input: ")
    input, err = inputReader.ReadString('\n')
```

```
    if err == nil {
        fmt.Printf("The input was: %s\n", input)
    }
}
```

`inputReader` 是一个指向 `bufio.Reader` 的指针。`inputReader := bufio.NewReader(os.Stdin)` 这行代码，将会创建一个读取器，并将其与标准输入绑定。

`bufio.NewReader()` 构造函数的签名为：`func NewReader(rd io.Reader) *Reader`

该函数的实参可以是满足 `io.Reader` 接口的任意对象（任意包含有适当的 `Read()` 方法的对象，请参考章节 11.8），函数返回一个新的带缓冲的 `io.Reader` 对象，它将从指定读取器（例如 `os.Stdin`）读取内容。

返回的读取器对象提供一个方法 `ReadString(delim byte)`，该方法从输入中读取内容，直到碰到 `delim` 指定的字符，然后将读取到的内容连同 `delim` 字符一起放到缓冲区。

`ReadString` 返回读取到的字符串，如果碰到错误则返回 `nil`。如果它一直读到文件结束，则返回读取到的字符串和 `io.EOF`。如果读取过程中没有碰到 `delim` 字符，将返回错误 `err != nil`。

在上面的例子中，我们会读取键盘输入，直到回车键（\n）被按下。

屏幕是标准输出 `os.Stdout`；`os.Stderr` 用于显示错误信息，大多数情况下等同于 `os.Stdout`。

一般情况下，我们会省略变量声明，而使用 `:=`，例如：

```
inputReader := bufio.NewReader(os.Stdin)
input, err := inputReader.ReadString('\n')
```

我们将从现在开始使用这种写法。

第二个例子从键盘读取输入，使用了 `switch` 语句：

示例 12.3 switch_input.go：

```
package main
import (
    "fmt"
    "os"
    "bufio"
)

func main() {
    inputReader := bufio.NewReader(os.Stdin)
    fmt.Println("Please enter your name:")
    input, err := inputReader.ReadString('\n')

    if err != nil {
        fmt.Println("There were errors reading, exiting program.")
        return
    }

    fmt.Printf("Your name is %s", input)
    // For Unix: test with delimiter "\n", for Windows: test with "\r\n"
    switch input {
    case "Philip\r\n":  fmt.Println("Welcome Philip!")
    case "Chris\r\n":   fmt.Println("Welcome Chris!")
    case "Ivo\r\n":     fmt.Println("Welcome Ivo!")
    default: fmt.Printf("You are not welcome here! Goodbye!")
    }
```

```
    // version 2:
    switch input {
    case "Philip\r\n":  fallthrough
    case "Ivo\r\n":     fallthrough
    case "Chris\r\n":   fmt.Printf("Welcome %s\n", input)
    default: fmt.Printf("You are not welcome here! Goodbye!\n")
    }

    // version 3:
    switch input {
    case "Philip\r\n", "Ivo\r\n":   fmt.Printf("Welcome %s\n", input)
    default: fmt.Printf("You are not welcome here! Goodbye!\n")
    }
}
```

注意：Unix和Windows的行结束符是不同的！

练习

练习 12.1: word_letter_count.go

编写一个程序，从键盘读取输入。当用户输入 'S' 的时候表示输入结束，这时程序输出 3 个数字：
i) 输入的字符的个数，包括空格，但不包括 '\r' 和 '\n'
ii) 输入的单词的个数
iii) 输入的行数

练习 12.2: calculator.go

编写一个简单的逆波兰式计算器，它接受用户输入的整型数（最大值 999999）和运算符 +、-、*、/。
输入的格式为：number1 ENTER number2 ENTER operator ENTER -> 显示结果
当用户输入字符 'q' 时，程序结束。请使用您在练习11.13中开发的 `stack` 包。

文件读写

12.2 文件读写

12.2.1 读文件

在 Go 语言中，文件使用指向 `os.File` 类型的指针来表示的，也叫做文件句柄。我们在前面章节使用到过标准输入 `os.Stdin` 和标准输出 `os.Stdout`，他们的类型都是 `*os.File`。让我们来看看下面这个程序：

示例 12.4 fileinput.go：

```
package main

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

func main() {
    inputFile, inputError := os.Open("input.dat")
    if inputError != nil {
        fmt.Printf("An error occurred on opening the inputfile\n" +
            "Does the file exist?\n" +
            "Have you got acces to it?\n")
        return // exit the function on error
    }
    defer inputFile.Close()

    inputReader := bufio.NewReader(inputFile)
    for {
        inputString, readerError := inputReader.ReadString('\n')
        fmt.Printf("The input was: %s", inputString)
        if readerError == io.EOF {
            return
        }
    }
}
```

变量 `inputFile` 是 `*os.File` 类型的。该类型是一个结构，表示一个打开文件的描述符（文件句柄）。然后，使用 `os` 包里的 `Open` 函数来打开一个文件。该函数的参数是文件名，类型为 `string`。在上面的程序中，我们以只读模式打开 `input.dat` 文件。

如果文件不存在或者程序没有足够的权限打开这个文件，Open函数会返回一个错误：`inputFile, inputError = os.Open("input.dat")`。如果文件打开正常，我们就使用 `defer inputFile.Close()` 语句确保在程序退出前关闭该文件。然后，我们使用 `bufio.NewReader` 来获得一个读取器变量。

通过使用 `bufio` 包提供的读取器（写入器也类似），如上面程序所示，我们可以很方便的操作相对高层的 string 对象，而避免了去操作比较底层的字节。

接着，我们在一个无限循环中使用 `ReadString('\n')` 或 `ReadBytes('\n')` 将文件的内容逐行（行结束符 '\n'）读取出来。

注意： 在之前的例子中，我们看到，Unix和Linux的行结束符是 \n，而Windows的行结束符是 \r\n。在使用 `ReadString` 和 `ReadBytes` 方法的时候，我们不需要关心操作系统的类型，直接使用 \n 就可以了。另外，我们也可以使用 `ReadLine()` 方法来实现相同的功能。

一旦读取到文件末尾，变量 `readerError` 的值将变成非空（事实上，其值为常量 `io.EOF` ），我们就会执行 `return` 语句从而退出循环。

其他类似函数：

1) 将整个文件的内容读到一个字符串里：

如果您想这么做，可以使用 `io/ioutil` 包里的 `ioutil.ReadFile()` 方法，该方法第一个返回值的类型是 `[]byte` ，里面存放读取到的内容，第二个返回值是错误，如果没有错误发生，第二个返回值为 nil。请看示例 12.5。类似的，函数 `WriteFile()` 可以将 `[]byte` 的值写入文件。

示例 12.5 read_write_file1.go：

```
package main
import (
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
    inputFile := "products.txt"
    outputFile := "products_copy.txt"
    buf, err := ioutil.ReadFile(inputFile)
    if err != nil {
        fmt.Fprintf(os.Stderr, "File Error: %s\n", err)
        // panic(err.Error())
    }
    fmt.Printf("%s\n", string(buf))
    err = ioutil.WriteFile(outputFile, buf, 0644) // oct, not hex
    if err != nil {
        panic(err.Error())
    }
}
```

2) 带缓冲的读取

在很多情况下，文件的内容是不按行划分的，或者干脆就是一个二进制文件。在这种情况下，`ReadString()` 就无法使用了，我们可以使用 `bufio.Reader` 的 `Read()` ，它只接收一个参数：

```
buf := make([]byte, 1024)
...
n, err := inputReader.Read(buf)
if (n == 0) { break}
```

变量 n 的值表示读取到的字节数.

3) 按列读取文件中的数据

如果数据是按列排列并用空格分隔的，你可以使用 `fmt` 包提供的以 FScan 开头的一系列函数来读取他们。请看以下程序，我们将 3 列的数据分别读入变量 v1、v2 和 v3 内，然后分别把他们添加到切片的尾部。

示例 12.6 read_file2.go：

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

func main() {
    file, err := os.Open("products2.txt")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    var col1, col2, col3 []string
    for {
        var v1, v2, v3 string
        _, err := fmt.Fscanln(file, &v1, &v2, &v3)
        // scans until newline
        if err != nil {
            break
        }
        col1 = append(col1, v1)
        col2 = append(col2, v2)
        col3 = append(col3, v3)
    }

    fmt.Println(col1)
    fmt.Println(col2)
    fmt.Println(col3)
}
```

输出结果：

```
[ABC FUNC GO]
[40 56 45]
[150 280 356]
```

注意： `path` 包里包含一个子包叫 `filepath`，这个子包提供了跨平台的函数，用于处理文件名和路径。例如 Base() 函数用于获得路径中的最后一个元素（不包含后面的分隔符）：

```
import "path/filepath"
filename := filepath.Base(path)
```

练习 12.3：read_csv.go

文件 products.txt 的内容如下：

```
"The ABC of Go";25.5;1500
"Functional Programming with Go";56;280
"Go for It";45.9;356
"The Go Way";55;500
```

每行的第一个字段为 title，第二个字段为 price，第三个字段为 quantity。内容的格式基本与 示例 12.3c 的相同，除了分隔符改成了分号。请读取出文件的内容，创建一个结构用于存取一行的数据，然后使用结构的切片，并把数据打印出来。

关于解析 CSV 文件，`encoding/csv` 包提供了相应的功能。具体请参考 http://golang.org/pkg/encoding/csv/

12.2.2 `compress`包：读取压缩文件

`compress` 包提供了读取压缩文件的功能，支持的压缩文件格式为：bzip2、flate、gzip、lzw 和 zlib。

下面的程序展示了如何读取一个 gzip 文件。

示例 12.7 gzipped.go：

```
package main

import (
    "fmt"
    "bufio"
    "os"
    "compress/gzip"
)

func main() {
    fName := "MyFile.gz"
    var r *bufio.Reader
    fi, err := os.Open(fName)
    if err != nil {
        fmt.Fprintf(os.Stderr, "%v, Can't open %s: error: %s\n", os.Args[0], fName,
            err)
        os.Exit(1)
    }
    defer fi.Close()
    fz, err := gzip.NewReader(fi)
    if err != nil {
        r = bufio.NewReader(fi)
    } else {
        r = bufio.NewReader(fz)
    }

    for {
        line, err := r.ReadString('\n')
        if err != nil {
            fmt.Println("Done reading file")
            os.Exit(0)
        }
        fmt.Println(line)
    }
}
```

12.2.3 写文件

请看以下程序：

示例 12.8 fileoutput.go：

```
package main

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

```
func main () {
    // var outputWriter *bufio.Writer
    // var outputFile *os.File
    // var outputError os.Error
    // var outputString string
    outputFile, outputError := os.OpenFile("output.dat", os.O_WRONLY|os.O_CREATE, 0666)
    if outputError != nil {
        fmt.Printf("An error occurred with file opening or creation\n")
        return
    }
    defer outputFile.Close()

    outputWriter := bufio.NewWriter(outputFile)
    outputString := "hello world!\n"

    for i:=0; i<10; i++ {
        outputWriter.WriteString(outputString)
    }
    outputWriter.Flush()
}
```

除了文件句柄，我们还需要 `bufio` 的 `Writer` 。我们以只写模式打开文件 `output.dat` ，如果文件不存在则自动创建：

```
outputFile, outputError := os.OpenFile("output.dat", os.O_WRONLY|os.O_CREATE, 0666)
```

可以看到， `OpenFile` 函数有三个参数：文件名、一个或多个标志（使用逻辑运算符“|”连接），使用的文件权限。

我们通常会用到以下标志：

- `os.O_RDONLY` ：只读
- `os.O_WRONLY` ：只写
- `os.O_CREATE` ：创建：如果指定文件不存在，就创建该文件。
- `os.O_TRUNC` ：截断：如果指定文件已存在，就将该文件的长度截为0。

在读文件的时候，文件的权限是被忽略的，所以在使用 `OpenFile` 时传入的第三个参数可以用0。而在写文件时，不管是 Unix 还是 Windows，都需要使用 0666。

然后，我们创建一个写入器（缓冲区）对象：

```
outputWriter := bufio.NewWriter(outputFile)
```

接着，使用一个 for 循环，将字符串写入缓冲区，写 10 次： `outputWriter.WriteString(outputString)`

缓冲区的内容紧接着被完全写入文件： `outputWriter.Flush()`

如果写入的东西很简单，我们可以使用 `fmt.Fprintf(outputFile, "Some test data.\n")` 直接将内容写入文件。 `fmt` 包里的 F 开头的 Print 函数可以直接写入任何 `io.Writer` ，包括文件（请参考章节12.8）。

程序 `filewrite.go` 展示了不使用 `fmt.FPrintf` 函数，使用其他函数如何写文件：

示例 12.8 filewrite.go：

```
package main

import "os"
```

```
func main() {
    os.Stdout.WriteString("hello, world\n")
    f, _ := os.OpenFile("test", os.O_CREATE|os.O_WRONLY, 0666)
    defer f.Close()
    f.WriteString("hello, world in a file\n")
}
```

使用 `os.Stdout.WriteString("hello, world\n")` ，我们可以输出到屏幕。

我们以只写模式创建或打开文件"test"，并且忽略了可能发生的错误： `f, _ := os.OpenFile("test", os.O_CREATE|os.O_WRONLY, 0666)`

我们不使用缓冲区，直接将内容写入文件： `f.WriteString( )`

练习 12.4：wiki_part1.go

（这是一个独立的练习，但是同时也是为章节15.4做准备）

程序中的数据结构如下，是一个包含以下字段的结构:

```
type Page struct {
    Title string
    Body  []byte
}
```

请给这个结构编写一个 `save` 方法，将 Title 作为文件名、Body作为文件内容，写入到文本文件中。

再编写一个 `load` 函数，接收的参数是字符串 title，该函数读取出与 title 对应的文本文件。请使用 `*Page` 做为参数，因为这个结构可能相当巨大，我们不想在内存中拷贝它。请使用 `ioutil` 包里的函数（参考章节12.2.1）。

文件拷贝

12.3 文件拷贝

如何拷贝一个文件到另一个文件？最简单的方式就是使用 io 包：

示例 12.10 filecopy.go：

```
// filecopy.go
package main

import (
    "fmt"
    "io"
    "os"
)

func main() {
    CopyFile("target.txt", "source.txt")
    fmt.Println("Copy done!")
}

func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}
```

注意 `defer` 的使用：当打开dst文件时发生了错误，那么 `defer` 仍然能够确保 `src.Close()` 执行。如果不这么做，src文件会一直保持打开状态并占用资源。

从命令行读取参数

12.4 从命令行读取参数

12.4.1 os 包

os 包中有一个 string 类型的切片变量 `os.Args`，用来处理一些基本的命令行参数，它在程序启动后读取命令行输入的参数。来看下面的打招呼程序：

示例 12.11 os_args.go：

```
// os_args.go
package main

import (
    "fmt"
    "os"
    "strings"
)

func main() {
    who := "Alice "
    if len(os.Args) > 1 {
        who += strings.Join(os.Args[1:], " ")
    }
    fmt.Println("Good Morning", who)
}
```

我们在 IDE 或编辑器中直接运行这个程序输出：`Good Morning Alice`

我们在命令行运行 `os_args` 或 `./os_args` 会得到同样的结果。

但是我们在命令行加入参数，像这样：`os_args John Bill Marc Luke`，将得到这样的输出：`Good Morning Alice John Bill Marc Luke`

这个命令行参数会放置在切片 `os.Args[]` 中（以空格分隔），从索引1开始（`os.Args[0]` 放的是程序本身的名字，在本例中是 `os_args`）。函数 `strings.Join` 以空格为间隔连接这些参数。

练习 12.5：hello_who.go

写一个"Hello World"的变种程序：把人的名字作为程序命令行执行的一个参数，比如：`hello_who Evan Michael Laura` 那么会输出 `Hello Evan Michael Laura`！

12.4.2 flag 包

flag 包有一个扩展功能用来解析命令行选项。但是通常被用来替换基本常量，例如，在某些情况下我们希望在命令行给常量一些不一样的值。（参看 19 章的项目）

在 flag 包中有一个 Flag 被定义成一个含有如下字段的结构体：

```
type Flag struct {
    Name     string // name as it appears on command line
    Usage    string // help message
```

```
    Value    Value  // value as set
    DefValue string // default value (as text); for usage message
}
```

下面的程序 `echo.go` 模拟了 Unix 的 echo 功能：

```
package main

import (
    "flag" // command line option parser
    "os"
)

var NewLine = flag.Bool("n", false, "print newline") // echo -n flag, of type *bool

const (
    Space   = " "
    Newline = "\n"
)

func main() {
    flag.PrintDefaults()
    flag.Parse() // Scans the arg list and sets up flags
    var s string = ""
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 {
            s += " "
            if *NewLine { // -n is parsed, flag becomes true
                s += Newline
            }
        }
        s += flag.Arg(i)
    }
    os.Stdout.WriteString(s)
}
```

`flag.Parse()` 扫描参数列表（或者常量列表）并设置 flag，`flag.Arg(i)` 表示第i个参数。`Parse()` 之后 `flag.Arg(i)` 全部可用，`flag.Arg(0)` 就是第一个真实的 flag，而不是像 `os.Args(0)` 放置程序的名字。

`flag.Narg()` 返回参数的数量。解析后 flag 或常量就可用了。
`flag.Bool()` 定义了一个默认值是 `false` 的 flag：当在命令行出现了第一个参数（这里是 “n”），flag 被设置成 `true`（NewLine 是 `*bool` 类型）。flag 被解引用到 `*NewLine`，所以当值是 `true` 时将添加一个 Newline（”\n”）。

`flag.PrintDefaults()` 打印 flag 的使用帮助信息，本例中打印的是：

```
-n=false: print newline
```

`flag.VisitAll(fn func(*Flag))` 是另一个有用的功能：按照字典顺序遍历 flag，并且对每个标签调用 fn （参考 15.8 章的例子）

当在命令行（Windows）中执行：`echo.exe A B C`，将输出：`A B C`；执行 `echo.exe -n A B C`，将输出：

```
A
B
C
```

每个字符的输出都新起一行，每次都在输出的数据前面打印使用帮助信息： `-n=false: print newline` 。

对于 `flag.Bool` 你可以设置布尔型 flag 来测试你的代码，例如定义一个 flag `processedFlag` ：

```
var processedFlag = flag.Bool("proc", false, "nothing processed yet")
```

在后面用如下代码来测试：

```
if *processedFlag { // found flag -proc
    r = process()
}
```

要给 flag 定义其它类型，可以使用 `flag.Int()` ， `flag.Float64()` ， `flag.String()` 。

在第 15.8 章你将找到一个具体的例子。

用 buffer 读取文件

12.5 用 buffer 读取文件

在下面的例子中，我们结合使用了缓冲读取文件和命令行 flag 解析这两项技术。如果不加参数，那么你输入什么屏幕就打印什么。

参数被认为是文件名，如果文件存在的话就打印文件内容到屏幕。命令行执行 `cat test` 测试输出。

示例 12.11 cat.go：

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)

func cat(r *bufio.Reader) {
    for {
        buf, err := r.ReadBytes('\n')
        fmt.Fprintf(os.Stdout, "%s", buf)
        if err == io.EOF {
            break
        }
    }
    return
}

func main() {
    flag.Parse()
    if flag.NArg() == 0 {
        cat(bufio.NewReader(os.Stdin))
    }
    for i := 0; i < flag.NArg(); i++ {
        f, err := os.Open(flag.Arg(i))
        if err != nil {
            fmt.Fprintf(os.Stderr, "%s:error reading from %s: %s\n", os.Args[0], flag.Arg(i), err.Error())
            continue
        }
        cat(bufio.NewReader(f))
        f.Close()
    }
}
```

在 12.6 章节，我们将看到如何使用缓冲写入。

练习 12.6：cat_numbered.go

扩展 cat.go 例子，使用 flag 添加一个选项，目的是为每一行头部加入一个行号。使用 `cat -n test` 测试输出。

用切片读写文件

12.6 用切片读写文件

切片提供了 Go 中处理 I/O 缓冲的标准方式，下面 `cat` 函数的第二版中，在一个切片缓冲内使用无限 for 循环（直到文件尾部 EOF）读取文件，并写入到标准输出（ `os.Stdout` ）。

```
func cat(f *os.File) {
    const NBUF = 512
    var buf [NBUF]byte
    for {
        switch nr, err := f.Read(buf[:]); true {
        case nr < 0:
            fmt.Fprintf(os.Stderr, "cat: error reading: %s\n", err.Error())
            os.Exit(1)
        case nr == 0: // EOF
            return
        case nr > 0:
            if nw, ew := os.Stdout.Write(buf[0:nr]); nw != nr {
                fmt.Fprintf(os.Stderr, "cat: error writing: %s\n", ew.Error())
            }
        }
    }
}
```

上面的代码来自于 `cat2.go` ，使用了 os 包中的 `os.File` 和 `Read` 方法； `cat2.go` 与 `cat.go` 具有同样的功能。

示例 12.14 cat2.go：

```
package main

import (
    "flag"
    "fmt"
    "os"
)

func cat(f *os.File) {
    const NBUF = 512
    var buf [NBUF]byte
    for {
        switch nr, err := f.Read(buf[:]); true {
        case nr < 0:
            fmt.Fprintf(os.Stderr, "cat: error reading: %s\n", err.Error())
            os.Exit(1)
        case nr == 0: // EOF
            return
        case nr > 0:
            if nw, ew := os.Stdout.Write(buf[0:nr]); nw != nr {
                fmt.Fprintf(os.Stderr, "cat: error writing: %s\n", ew.Error())
            }
        }
    }
}
```

```
func main() {
    flag.Parse() // Scans the arg list and sets up flags
    if flag.NArg() == 0 {
        cat(os.Stdin)
    }
    for i := 0; i < flag.NArg(); i++ {
        f, err := os.Open(flag.Arg(i))
        if f == nil {
            fmt.Fprintf(os.Stderr, "cat: can't open %s: error %s\n", flag.Arg(i), err)
            os.Exit(1)
        }
        cat(f)
        f.Close()
    }
}
```

用 defer 关闭文件

12.7 用 defer 关闭文件

`defer` 关键字（参看 6.4）对于在函数结束时关闭打开的文件非常有用，例如下面的代码片段：

```
func data(name string) string {
    f, _ := os.OpenFile(name, os.O_RDONLY, 0)
    defer f.Close() // idiomatic Go code!
    contents, _ := ioutil.ReadAll(f)
    return string(contents)
}
```

在函数 return 后执行了 `f.Close()`

使用接口的实际例子：fmt.Fprintf

12.8 使用接口的实际例子：fmt.Fprintf

例子程序 `io_interfaces.go` 很好的阐述了 io 包中的接口概念。

示例 12.15 io_interfaces.go：

```
// interfaces being used in the GO-package fmt
package main

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

func main() {
    // unbuffered
    fmt.Fprintf(os.Stdout, "%s\n", "hello world! - unbuffered")
    // buffered: os.Stdout implements io.Writer
    buf := bufio.NewWriter(os.Stdout)
    // and now so does buf.
    fmt.Fprintf(buf, "%s\n", "hello world! - buffered")
    buf.Flush()
}
```

输出：

```
hello world! - unbuffered
hello world! - buffered
```

下面是 `fmt.Fprintf()` 函数的实际签名

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

其不是写入一个文件，而是写入一个 `io.Writer` 接口类型的变量，下面是 `Writer` 接口在 io 包中的定义：

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

`fmt.Fprintf()` 依据指定的格式向第一个参数内写入字符串，第一个参数必须实现了 `io.Writer` 接口。`Fprintf()` 能够写入任何类型，只要其实现了 `Write` 方法，包括 `os.Stdout`，文件（例如 os.File），管道，网络连接，通道等等，同样的也可以使用 bufio 包中缓冲写入。bufio 包中定义了 `type Writer struct{...}` 。

bufio.Writer 实现了 Write 方法：

```
func (b *Writer) Write(p []byte) (nn int, err error)
```

它还有一个工厂函数：传给它一个 `io.Writer` 类型的参数，它会返回一个带缓冲的 `bufio.Writer` 类型的 `io.Writer`：

```
func NewWriter(wr io.Writer) (b *Writer)
```

其适合任何形式的缓冲写入。

在缓冲写入的最后千万不要忘了使用 `Flush()`，否则最后的输出不会被写入。

在 15.2-15.8 章节，我们将使用 `fmt.Fprint` 函数向 `http.ResponseWriter` 写入，其同样实现了 io.Writer 接口。

练习 12.7：remove_3till5char.go

下面的代码有一个输入文件 `goprogram`，然后以每一行为单位读取，从读取的当前行中截取第 3 到第 5 的字节写入另一个文件。然而当你运行这个程序，输出的文件却是个空文件。找出程序逻辑中的 bug，修正它并测试。

```
package main

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

func main() {
    inputFile, _ := os.Open("goprogram")
    outputFile, _ := os.OpenFile("goprogramT", os.O_WRONLY|os.O_CREATE, 0666)
    defer inputFile.Close()
    defer outputFile.Close()
    inputReader := bufio.NewReader(inputFile)
    outputWriter := bufio.NewWriter(outputFile)
    for {
        inputString, _, readerError := inputReader.ReadLine()
        if readerError == io.EOF {
            fmt.Println("EOF")
            return
        }
        outputString := string(inputString[2:5]) + "\r\n"
        _, err := outputWriter.WriteString(outputString)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
    fmt.Println("Conversion done")
}
```

JSON 数据格式

12.9 JSON 数据格式

数据结构要在网络中传输或保存到文件，就必须对其编码和解码；目前存在很多编码格式：JSON，XML，gob，Google 缓冲协议等等。Go 语言支持所有这些编码格式；在后面的章节，我们将讨论前三种格式。

结构可能包含二进制数据，如果将其作为文本打印，那么可读性是很差的。另外结构内部可能包含匿名字段，而不清楚数据的用意。

通过把数据转换成纯文本，使用命名的字段来标注，让其具有可读性。这样的数据格式可以通过网络传输，而且是与平台无关的，任何类型的应用都能够读取和输出，不与操作系统和编程语言的类型相关。

下面是一些术语说明：

- 数据结构 -> 指定格式 = `序列化` 或 `编码`（传输之前）
- 指定格式 -> 数据格式 = `反序列化` 或 `解码`（传输之后）

序列化是在内存中把数据转换成指定格式（data -> string），反之亦然（string -> data structure）

编码也是一样的，只是输出一个数据流（实现了 io.Writer 接口）；解码是从一个数据流（实现了 io.Reader）输出到一个数据结构。

我们都比较熟悉 XML 格式(参阅 12.10)；但有些时候 JSON（JavaScript Object Notation，参阅 http://json.org）被作为首选，主要是由于其格式上非常简洁。通常 JSON 被用于 web 后端和浏览器之间的通讯，但是在其它场景也同样的有用。

这是一个简短的 JSON 片段：

```
{
    "Person": {
        "FirstName": "Laura",
        "LastName": "Lynn"
    }
}
```

尽管 XML 被广泛的应用，但是 JSON 更加简洁、轻量（占用更少的内存、磁盘及网络带宽）和更好的可读性，这也使它越来越受欢迎。

Go 语言的 json 包可以让你在程序中方便的读取和写入 JSON 数据。

我们将在下面的例子里使用 json 包，并使用练习 10.1 vcard.go 中一个简化版本的 Address 和 VCard 结构（为了简单起见，我们忽略了很多错误处理，不过在实际应用中你必须要合理的处理这些错误，参阅 13 章）

示例 12.16 json.go：

```
// json.go
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "os"
)
```

```
type Address struct {
    Type    string
    City    string
    Country string
}

type VCard struct {
    FirstName string
    LastName  string
    Addresses []*Address
    Remark    string
}

func main() {
    pa := &Address{"private", "Aartselaar", "Belgium"}
    wa := &Address{"work", "Boom", "Belgium"}
    vc := VCard{"Jan", "Kersschot", []*Address{pa, wa}, "none"}
    // fmt.Printf("%v: \n", vc) // {Jan Kersschot [0x126d2b80 0x126d2be0] none}:
    // JSON format:
    js, _ := json.Marshal(vc)
    fmt.Printf("JSON format: %s", js)
    // using an encoder:
    file, _ := os.OpenFile("vcard.json", os.O_CREATE|os.O_WRONLY, 0666)
    defer file.Close()
    enc := json.NewEncoder(file)
    err := enc.Encode(vc)
    if err != nil {
        log.Println("Error in encoding json")
    }
}
```

`json.Marshal()` 的函数签名是 `func Marshal(v interface{}) ([]byte, error)`，下面是数据编码后的 JSON 文本（实际上是一个 []byte）：

```
{
    "FirstName": "Jan",
    "LastName": "Kersschot",
    "Addresses": [{
        "Type": "private",
        "City": "Aartselaar",
        "Country": "Belgium"
    }, {
        "Type": "work",
        "City": "Boom",
        "Country": "Belgium"
    }],
    "Remark": "none"
}
```

出于安全考虑，在 web 应用中最好使用 `json.MarshalforHTML()` 函数，其对数据执行HTML转码，所以文本可以被安全地嵌在 HTML `<script>` 标签中。

`json.NewEncoder()` 的函数签名是 `func NewEncoder(w io.Writer) *Encoder`，返回的Encoder类型的指针可调用方法 `Encode(v interface{})`，将数据对象 v 的json编码写入 `io.Writer` w 中。

JSON 与 Go 类型对应如下：

- bool 对应 JSON 的 boolean

- float64 对应 JSON 的 number
- string 对应 JSON 的 string
- nil 对应 JSON 的 null

不是所有的数据都可以编码为 JSON 类型：只有验证通过的数据结构才能被编码：

- JSON 对象只支持字符串类型的 key；要编码一个 Go map 类型，map 必须是 map[string]T（T是 `json` 包中支持的任何类型）
- Channel，复杂类型和函数类型不能被编码
- 不支持循环数据结构；它将引起序列化进入一个无限循环
- 指针可以被编码，实际上是对指针指向的值进行编码（或者指针是 nil）

反序列化：

`UnMarshal()` 的函数签名是 `func Unmarshal(data []byte, v interface{}) error` 把 JSON 解码为数据结构。

示例12.16中对 vc 编码后的数据为 `js` ，对其解码时，我们首先创建结构 VCard 用来保存解码的数据： `var v VCard` 并调用 `json.Unmarshal(js, &v)` ，解析 []byte 中的 JSON 数据并将结果存入指针 &v 指向的值。

虽然反射能够让 JSON 字段去尝试匹配目标结构字段；但是只有真正匹配上的字段才会填充数据。字段没有匹配不会报错，而是直接忽略掉。

（练习 15.2b twitter_status_json.go 中用到了 UnMarshal）

解码任意的数据：

json 包使用 `map[string]interface{}` 和 `[]interface{}` 储存任意的 JSON 对象和数组；其可以被反序列化为任何的 JSON blob 存储到接口值中。

来看这个 JSON 数据，被存储在变量 b 中：

```
b := []byte(`{"Name": "Wednesday", "Age": 6, "Parents": ["Gomez", "Morticia"]}`)
```

不用理解这个数据的结构，我们可以直接使用 Unmarshal 把这个数据编码并保存在接口值中：

```
var f interface{}
err := json.Unmarshal(b, &f)
```

f 指向的值是一个 map，key 是一个字符串，value 是自身存储作为空接口类型的值：

```
map[string]interface{} {
    "Name": "Wednesday",
    "Age":  6,
    "Parents": []interface{} {
        "Gomez",
        "Morticia",
    },
}
```

要访问这个数据，我们可以使用类型断言

```
m := f.(map[string]interface{})
```

我们可以通过 for range 语法和 type switch 来访问其实际类型：

```
for k, v := range m {
    switch vv := v.(type) {
    case string:
        fmt.Println(k, "is string", vv)
    case int:
        fmt.Println(k, "is int", vv)

    case []interface{}:
        fmt.Println(k, "is an array:")
        for i, u := range vv {
            fmt.Println(i, u)
        }
    default:
        fmt.Println(k, "is of a type I don't know how to handle")
    }
}
```

通过这种方式，你可以处理未知的 JSON 数据，同时可以确保类型安全。

解码数据到结构

如果我们事先知道 JSON 数据，我们可以定义一个适当的结构并对 JSON 数据反序列化。下面的例子中，我们将定义：

```
type FamilyMember struct {
    Name    string
    Age     int
    Parents []string
}
```

并对其反序列化：

```
var m FamilyMember
err := json.Unmarshal(b, &m)
```

程序实际上是分配了一个新的切片。这是一个典型的反序列化引用类型（指针、切片和 map）的例子。

编码和解码流

json 包提供 Decoder 和 Encoder 类型来支持常用 JSON 数据流读写。NewDecoder 和 NewEncoder 函数分别封装了 io.Reader 和 io.Writer 接口。

```
func NewDecoder(r io.Reader) *Decoder
func NewEncoder(w io.Writer) *Encoder
```

要想把 JSON 直接写入文件，可以使用 json.NewEncoder 初始化文件（或者任何实现 io.Writer 的类型），并调用 Encode()；反过来与其对应的是使用 json.NewDecoder 和 Decode() 函数：

```
func NewDecoder(r io.Reader) *Decoder
func (dec *Decoder) Decode(v interface{}) error
```

来看下接口是如何对实现进行抽象的：数据结构可以是任何类型，只要其实现了某种接口，目标或源数据要能够被编码就必须实现 io.Writer 或 io.Reader 接口。由于 Go 语言中到处都实现了 Reader 和 Writer，因此 Encoder 和 Decoder 可被应用的场景非常广泛，例如读取或写入 HTTP 连接、websockets 或文件。

XML 数据格式

12.10 XML 数据格式

下面是与 12.9 节 JSON 例子等价的 XML 版本：

```
<Person>
    <FirstName>Laura</FirstName>
    <LastName>Lynn</LastName>
</Person>
```

如同 json 包一样，也有 `Marshal()` 和 `UnMarshal()` 从 XML 中编码和解码数据；但这个更通用，可以从文件中读取和写入（或者任何实现了 io.Reader 和 io.Writer 接口的类型）

和 JSON 的方式一样，XML 数据可以序列化为结构，或者从结构反序列化为 XML 数据；这些可以在例子 15.8（twitter_status.go）中看到。

encoding/xml 包实现了一个简单的 XML 解析器（SAX），用来解析 XML 数据内容。下面的例子说明如何使用解析器：

示例 12.17 xml.go：

```
// xml.go
package main

import (
    "encoding/xml"
    "fmt"
    "strings"
)

var t, token xml.Token
var err error

func main() {
    input := "<Person><FirstName>Laura</FirstName><LastName>Lynn</LastName></Person>"
    inputReader := strings.NewReader(input)
    p := xml.NewDecoder(inputReader)

    for t, err = p.Token(); err == nil; t, err = p.Token() {
        switch token := t.(type) {
        case xml.StartElement:
            name := token.Name.Local
            fmt.Printf("Token name: %s\n", name)
            for _, attr := range token.Attr {
                attrName := attr.Name.Local
                attrValue := attr.Value
                fmt.Printf("An attribute is: %s %s\n", attrName, attrValue)
                // ...
            }
        case xml.EndElement:
            fmt.Println("End of token")
        case xml.CharData:
            content := string([]byte(token))
            fmt.Printf("This is the content: %v\n", content)
            // ...
```

```
        default:
            // ...
        }
    }
}
```

输出：

```
Token name: Person
Token name: FirstName
This is the content: Laura
End of token
Token name: LastName
This is the content: Lynn
End of token
End of token
```

包中定义了若干 XML 标签类型：StartElement，Chardata（这是从开始标签到结束标签之间的实际文本），EndElement，Comment，Directive 或 ProcInst。

包中同样定义了一个结构解析器： `NewParser` 方法持有一个 io.Reader（这里具体类型是 strings.NewReader）并生成一个解析器类型的对象。还有一个 `Token()` 方法返回输入流里的下一个 XML token。在输入流的结尾处，会返回（nil，io.EOF）

XML 文本被循环处理直到 `Token()` 返回一个错误，因为已经到达文件尾部，再没有内容可供处理了。通过一个 type-switch 可以根据一些 XML 标签进一步处理。Chardata 中的内容只是一个 []byte，通过字符串转换让其变得可读性强一些。

用 Gob 传输数据

12.11 用 Gob 传输数据

Gob 是 Go 自己的以二进制形式序列化和反序列化程序数据的格式；可以在 `encoding` 包中找到。这种格式的数据简称为 Gob （即 Go binary 的缩写）。类似于 Python 的 “pickle” 和 Java 的 “Serialization”。

Gob 通常用于远程方法调用（RPCs，参见 15.9节 的 rpc 包）参数和结果的传输，以及应用程序和机器之间的数据传输。
它和 JSON 或 XML 有什么不同呢？Gob 特定地用于纯 Go 的环境中，例如，两个用 Go 写的服务之间的通信。这样的话服务可以被实现得更加高效和优化。
Gob 不是可外部定义，语言无关的编码方式。因此它的首选格式是二进制，而不是像 JSON 和 XML 那样的文本格式。
Gob 并不是一种不同于 Go 的语言，而是在编码和解码过程中用到了 Go 的反射。

Gob 文件或流是完全自描述的：里面包含的所有类型都有一个对应的描述，并且总是可以用 Go 解码，而不需要了解文件的内容。

只有可导出的字段会被编码，零值会被忽略。在解码结构体的时候，只有同时匹配名称和可兼容类型的字段才会被解码。当源数据类型增加新字段后，Gob 解码客户端仍然可以以这种方式正常工作：解码客户端会继续识别以前存在的字段。并且还提供了很大的灵活性，比如在发送者看来，整数被编码成没有固定长度的可变长度，而忽略具体的 Go 类型。

假如在发送者这边有一个有结构 T：

```
type T struct { X, Y, Z int }
var t = T{X: 7, Y: 0, Z: 8}
```

而在接收者这边可以用一个结构体 U 类型的变量 u 来接收这个值：

```
type U struct { X, Y *int8 }
var u U
```

在接收者中，X 的值是7，Y 的值是0（Y的值并没有从 t 中传递过来，因为它是零值）

和 JSON 的使用方式一样，Gob 使用通用的 `io.Writer` 接口，通过 `NewEncoder()` 函数创建 `Encoder` 对象并调用 `Encode()` ；相反的过程使用通用的 `io.Reader` 接口，通过 `NewDecoder()` 函数创建 `Decoder` 对象并调用 `Decode()` 。

我们把示例 12.12 的信息写进名为 vcard.gob 的文件作为例子。这会产生一个文本可读数据和二进制数据的混合，当你试着在文本编辑中打开的时候会看到。

在示例 12.18 中你会看到一个编解码，并且以字节缓冲模拟网络传输的简单例子：

示例 12.18 gob1.go：

```
// gob1.go
package main

import (
    "bytes"
    "fmt"
    "encoding/gob"
    "log"
)
```

```
type P struct {
    X, Y, Z int
    Name    string
}

type Q struct {
    X, Y *int32
    Name string
}

func main() {
    // Initialize the encoder and decoder.  Normally enc and dec would be
    // bound to network connections and the encoder and decoder would
    // run in different processes.
    var network bytes.Buffer    // Stand-in for a network connection
    enc := gob.NewEncoder(&network) // Will write to network.
    dec := gob.NewDecoder(&network)     // Will read from network.
    // Encode (send) the value.
    err := enc.Encode(P{3, 4, 5, "Pythagoras"})
    if err != nil {
        log.Fatal("encode error:", err)
    }
    // Decode (receive) the value.
    var q Q
    err = dec.Decode(&q)
    if err != nil {
        log.Fatal("decode error:", err)
    }
    fmt.Printf("%q: {%d,%d}\n", q.Name, q.X, q.Y)
}
// Output:   "Pythagoras": {3,4}
```

示例 12.19 gob2.go 编码到文件：

```
// gob2.go
package main

import (
    "encoding/gob"
    "log"
    "os"
)

type Address struct {
    Type            string
    City            string
    Country         string
}

type VCard struct {
    FirstName    string
    LastName    string
    Addresses    []*Address
    Remark        string
}

var content    string

func main() {
    pa := &Address{"private", "Aartselaar","Belgium"}
```

```
    wa := &Address{"work", "Boom", "Belgium"}
    vc := VCard{"Jan", "Kersschot", []*Address{pa,wa}, "none"}
    // fmt.Printf("%v: \n", vc) // {Jan Kersschot [0x126d2b80 0x126d2be0] none}:
    // using an encoder:
    file, _ := os.OpenFile("vcard.gob", os.O_CREATE|os.O_WRONLY, 0666)
    defer file.Close()
    enc := gob.NewEncoder(file)
    err := enc.Encode(vc)
    if err != nil {
        log.Println("Error in encoding gob")
    }
}
```

练习 12.8：degob.go：

写一个程序读取 vcard.gob 文件，解码并打印它的内容。

Go 中的密码学

12.12 Go 中的密码学

通过网络传输的数据必须加密，以防止被 hacker（黑客）读取或篡改，并且保证发出的数据和收到的数据检验和一致。
鉴于 Go 母公司的业务，我们毫不惊讶地看到 Go 的标准库为该领域提供了超过 30 个的包：

- `hash` 包：实现了 `adler32`、`crc32`、`crc64` 和 `fnv` 校验；
- `crypto` 包：实现了其它的 hash 算法，比如 `md4`、`md5`、`sha1` 等。以及完整地实现了 `aes`、`blowfish`、`rc4`、`rsa`、`xtea` 等加密算法。

下面的示例用 `sha1` 和 `md5` 计算并输出了一些校验值。

示例 12.20 hash_sha1.go：

```
// hash_sha1.go
package main

import (
    "fmt"
    "crypto/sha1"
    "io"
    "log"
)

func main() {
    hasher := sha1.New()
    io.WriteString(hasher, "test")
    b := []byte{}
    fmt.Printf("Result: %x\n", hasher.Sum(b))
    fmt.Printf("Result: %d\n", hasher.Sum(b))
    //
    hasher.Reset()
    data := []byte("We shall overcome!")
    n, err := hasher.Write(data)
    if n!=len(data) || err!=nil {
        log.Printf("Hash write error: %v / %v", n, err)
    }
    checksum := hasher.Sum(b)
    fmt.Printf("Result: %x\n", checksum)
}
```

输出：

```
Result: a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
Result: [169 74 143 229 204 177 155 166 28 76 8 115 211 145 233 135 152 47 187 211]
Result: e2222bfc59850bbb00a722e764a555603bb59b2a
```

通过调用 `sha1.New()` 创建了一个新的 `hash.Hash` 对象，用来计算 SHA1 校验值。`Hash` 类型实际上是一个接口，它实现了 `io.Writer` 接口：

```
type Hash interface {
    // Write (via the embedded io.Writer interface) adds more data to the running hash.
```

```
    // It never returns an error.
    io.Writer

    // Sum appends the current hash to b and returns the resulting slice.
    // It does not change the underlying hash state.
    Sum(b []byte) []byte

    // Reset resets the Hash to its initial state.
    Reset()

    // Size returns the number of bytes Sum will return.
    Size() int

    // BlockSize returns the hash's underlying block size.
    // The Write method must be able to accept any amount
    // of data, but it may operate more efficiently if all writes
    // are a multiple of the block size.
    BlockSize() int
}
```

通过 io.WriteString 或 hasher.Write 将给定的 []byte 附加到当前的 `hash.Hash` 对象中。

练习 12.9：hash_md5.go：

在示例 12.20 中检验 md5 算法。

错误处理与测试

13.0 错误处理与测试

Go 没有像 Java 和 .NET 那样的 `try/catch` 异常机制：不能执行抛异常操作。但是有一套 `defer-panic-and-recover` 机制（参见 13.2-13.3 节）。

Go 的设计者觉得 `try/catch` 机制的使用太泛滥了，而且从底层向更高的层级抛异常太耗费资源。他们给 Go 设计的机制也可以“捕捉”异常，但是更轻量，并且只应该作为（处理错误的）最后的手段。

Go 是怎么处理普通错误的呢？通过在函数和方法中返回错误对象作为它们的唯一或最后一个返回值——如果返回 nil，则没有错误发生——并且主调（calling）函数总是应该检查收到的错误。

永远不要忽略错误，否则可能会导致程序崩溃！！

处理错误并且在函数发生错误的地方给用户返回错误信息：照这样处理就算真的出了问题，你的程序也能继续运行并且通知给用户。 `panic and recover` 是用来处理真正的异常（无法预测的错误）而不是普通的错误。

库函数通常必须返回某种错误提示给主调（calling）函数。

在前面的章节中我们了解了 Go 检查和报告错误条件的惯有方式：

- 产生错误的函数会返回两个变量，一个值和一个错误码；如果后者是 nil 就是成功，非 nil 就是发生了错误。
- 为了防止发生错误时正在执行的函数（如果有必要的话甚至会是整个程序）被中止，在调用函数后必须检查错误。

下面这段来自 pack1 包的代码 Func1 测试了它的返回值：

```
if value, err := pack1.Func1(param1); err != nil {
    fmt.Printf("Error %s in pack1.Func1 with parameter %v", err.Error(), param1)
    return    // or: return err
} else {
    // Process(value)
}
```

为了更清晰的代码，应该总是使用包含错误值变量的 if 复合语句

上例除了 `fmt.Printf` 还可以使用 log 中对应的方法（参见 13.3 节 和 15.2 节），如果程序中止也没关系的话甚至可以使用 `panic` （参见后面的章节）。

错误处理

13.1 错误处理

Go 有一个预先定义的 error 接口类型

```
type error interface {
    Error() string
}
```

错误值用来表示异常状态；我们可以在 5.2 节中看到它的标准用法。处理文件操作的例子可以在 12 章找到；我们将在 15 章看到网络操作的例子。errors 包中有一个 errorString 结构体实现了 error 接口。当程序处于错误状态时可以用 `os.Exit(1)` 来中止运行。

13.1.1 定义错误

任何时候当你需要一个新的错误类型，都可以用 `errors` （必须先 import）包的 `errors.New` 函数接收合适的错误信息来创建，像下面这样：

```
err := errors.New("math - square root of negative number")
```

在示例 13.1 中你可以看到一个简单的用例：

示例 13.1 errors.go：

```
// errors.go
package main

import (
    "errors"
    "fmt"
)

var errNotFound error = errors.New("Not found error")

func main() {
    fmt.Printf("error: %v", errNotFound)
}
// error: Not found error
```

可以把它用于计算平方根函数的参数测试：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New ("math - square root of negative number")
    }
   // implementation of Sqrt
}
```

你可以像下面这样调用 Sqrt 函数：

```
if f, err := Sqrt(-1); err != nil {
    fmt.Printf("Error: %s\n", err)
}
```

由于 `fmt.Printf` 会自动调用 `String()` 方法（参见 10.7 节），所以错误信息 “Error: math - square root of negative number” 会打印出来。通常（错误信息）都会有像 “Error:” 这样的前缀，所以你的错误信息不要以大写字母开头。

在大部分情况下自定义错误结构类型很有意义的，可以包含除了（低层级的）错误信息以外的其它有用信息，例如，正在进行的操作（打开文件等），全路径或名字。看下面例子中 os.Open 操作触发的 PathError 错误：

```
// PathError records an error and the operation and file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error  // Returned by the system call.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": "+ e.Err.Error()
}
```

如果有不同错误条件可能发生，那么对实际的错误使用类型断言或类型判断（type-switch）是很有用的，并且可以根据错误场景做一些补救和恢复操作。

```
//  err != nil
if e, ok := err.(*os.PathError); ok {
    // remedy situation
}
```

或：

```
switch err := err.(type) {
    case ParseError:
        PrintParseError(err)
    case PathError:
        PrintPathError(err)
    ...
    default:
        fmt.Printf("Not a special error, just %s\n", err)
}
```

作为第二个例子考虑用 json 包的情况。当 json.Decode 在解析 JSON 文档发生语法错误时，指定返回一个 SyntaxError 类型的错误：

```
type SyntaxError struct {
    msg    string // description of error
// error occurred after reading Offset bytes, from which line and columnnr can be obtained
    Offset int64
}

func (e *SyntaxError) Error() string { return e.msg }
```

在调用代码中你可以像这样用类型断言测试错误是不是上面的类型：

```
if serr, ok := err.(*json.SyntaxError); ok {
    line, col := findLine(f, serr.Offset)
    return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
}
```

包也可以用额外的方法（methods）定义特定的错误，比如 net.Error：

```
package net
type Error interface {
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

在 15.1 节 我们可以看到怎么使用它。

正如你所看到的一样，所有的例子都遵循同一种命名规范：错误类型以“Error”结尾，错误变量以“err”或“Err”开头。

syscall 是低阶外部包，用来提供系统基本调用的原始接口。它们返回封装整数类型错误码的syscall.Errno；类型 syscall.Errno 实现了 Error 接口。

大部分 syscall 函数都返回一个结果和可能的错误，比如：

```
r, err := syscall.Open(name, mode, perm)
if err != nil {
    fmt.Println(err.Error())
}
```

os 包也提供了一套像 os.EINAL 这样的标准错误，它们基于 syscall 错误：

```
var (
    EPERM        Error = Errno(syscall.EPERM)
    ENOENT       Error = Errno(syscall.ENOENT)
    ESRCH        Error = Errno(syscall.ESRCH)
    EINTR        Error = Errno(syscall.EINTR)
    EIO            Error = Errno(syscall.EIO)
    ...
)
```

13.1.2 用 fmt 创建错误对象

通常你想要返回包含错误参数的更有信息量的字符串，例如：可以用 `fmt.Errorf()` 来实现：它和 fmt.Printf() 完全一样，接收一个或多个格式占位符的格式化字符串和相应数量的占位变量。和打印信息不同的是它用信息生成错误对象。

比如在前面的平方根例子中使用：

```
if f < 0 {
    return 0, fmt.Errorf("math: square root of negative number %g", f)
}
```

第二个例子：从命令行读取输入时，如果加了 help 标志，我们可以用有用的信息产生一个错误：

```
if len(os.Args) > 1 && (os.Args[1] == "-h" || os.Args[1] == "--help") {
    err = fmt.Errorf("usage: %s infile.txt outfile.txt", filepath.Base(os.Args[0]))
```

```
    return
}
```

运行时异常和 panic

13.2 运行时异常和 panic

当发生像数组下标越界或类型断言失败这样的运行错误时，Go 运行时会触发*运行时 panic*，伴随着程序的崩溃抛出一个 `runtime.Error` 接口类型的值。这个错误值有个 `RuntimeError()` 方法用于区别普通错误。

`panic` 可以直接从代码初始化：当错误条件（我们所测试的代码）很严苛且不可恢复，程序不能继续运行时，可以使用 `panic` 函数产生一个中止程序的运行时错误。 `panic` 接收一个做任意类型的参数，通常是字符串，在程序死亡时被打印出来。Go 运行时负责中止程序并给出调试信息。在示例 13.2 panic.go 中阐明了它的工作方式：

```
package main

import "fmt"

func main() {
    fmt.Println("Starting the program")
    panic("A severe error occurred: stopping the program!")
    fmt.Println("Ending the program")
}
```

输出如下：

```
Starting the program
panic: A severe error occurred: stopping the program!
panic PC=0x4f3038
runtime.panic+0x99 /go/src/pkg/runtime/proc.c:1032
        runtime.panic(0x442938, 0x4f08e8)
main.main+0xa5 E:/Go/GoBoek/code examples/chapter 13/panic.go:8
        main.main()
runtime.mainstart+0xf 386/asm.s:84
        runtime.mainstart()
runtime.goexit /go/src/pkg/runtime/proc.c:148
        runtime.goexit()
---- Error run E:/Go/GoBoek/code examples/chapter 13/panic.exe with code Crashed
---- Program exited with code -1073741783
```

一个检查程序是否被已知用户启动的具体例子：

```
var user = os.Getenv("USER")

func check() {
    if user == "" {
        panic("Unknown user: no value for $USER")
    }
}
```

可以在导入包的 init() 函数中检查这些。

当发生错误必须中止程序时， `panic` 可以用于错误处理模式：

```
if err != nil {
    panic("ERROR occurred:" + err.Error())
```

```
}
```

Go panicking：

在多层嵌套的函数调用中调用 panic，可以马上中止当前函数的执行，所有的 defer 语句都会保证执行并把控制权交还给接收到 panic 的函数调用者。这样向上冒泡直到最顶层，并执行（每层的） defer，在栈顶处程序崩溃，并在命令行中用传给 panic 的值报告错误情况：这个终止过程就是 *panicking*。

标准库中有许多包含 `Must` 前缀的函数，像 `regexp.MustComplie` 和 `template.Must` ；当正则表达式或模板中转入的转换字符串导致错误时，这些函数会 panic。

不能随意地用 panic 中止程序，必须尽力补救错误让程序能继续执行。

从 panic 中恢复（Recover）

13.3 从 panic 中恢复（Recover）

正如名字一样，这个（recover）内建函数被用于从 panic 或 错误场景中恢复：让程序可以从 panicking 重新获得控制权，停止终止过程进而恢复正常执行。

`recover` 只能在 defer 修饰的函数（参见 6.4 节）中使用：用于取得 panic 调用中传递过来的错误值，如果是正常执行，调用 `recover` 会返回 nil，且没有其它效果。

总结：panic 会导致栈被展开直到 defer 修饰的 recover() 被调用或者程序中止。

下面例子中的 protect 函数调用函数参数 g 来保护调用者防止从 g 中抛出的运行时 panic，并展示 panic 中的信息：

```
func protect(g func()) {
    defer func() {
        log.Println("done")
        // Println executes normally even if there is a panic
        if err := recover(); err != nil {
            log.Printf("run time panic: %v", err)
        }
    }()
    log.Println("start")
    g() //   possible runtime-error
}
```

这跟 Java 和 .NET 这样的语言中的 catch 块类似。

log 包实现了简单的日志功能：默认的 log 对象向标准错误输出中写入并打印每条日志信息的日期和时间。除了 `Println` 和 `Printf` 函数，其它的致命性函数都会在写完日志信息后调用 os.Exit(1)，那些退出函数也是如此。而 Panic 效果的函数会在写完日志信息后调用 panic；可以在程序必须中止或发生了临界错误时使用它们，就像当 web 服务器不能启动时那样（参见 15.4 节中的例子）。

log 包用那些方法（methods）定义了一个 Logger 接口类型，如果你想自定义日志系统的话可以参考（参见 http://golang.org/pkg/log/#Logger）。

这是一个展示 panic，defer 和 recover 怎么结合使用的完整例子：

示例 13.3 panic_recover.go：

```
// panic_recover.go
package main

import (
    "fmt"
)

func badCall() {
    panic("bad end")
}

func test() {
    defer func() {
        if e := recover(); e != nil {
            fmt.Printf("Panicing %s\r\n", e)
```

```
        }
    }()
    badCall()
    fmt.Printf("After bad call\r\n") // <-- wordt niet bereikt
}

func main() {
    fmt.Printf("Calling test\r\n")
    test()
    fmt.Printf("Test completed\r\n")
}
```

输出：

```
Calling test
Panicing bad end
Test completed
```

`defer-panic-recover` 在某种意义上也是一种像 `if`，`for` 这样的控制流机制。

Go 标准库中许多地方都用了这个机制，例如，json 包中的解码和 regexp 包中的 Complie 函数。Go 库的原则是即使在包的内部使用了 panic，在它的对外接口（API）中也必须用 recover 处理成返回显式的错误。

自定义包中的错误处理和 panicking

13.4 自定义包中的错误处理和 panicking

这是所有自定义包实现者应该遵守的最佳实践：

1）*在包内部，总是应该从 panic 中 recover*：不允许显式的超出包范围的 panic()

2）*向包的调用者返回错误值（而不是 panic）*。

在包内部，特别是在非导出函数中有很深层次的嵌套调用时，将 panic 转换成 error 来告诉调用方为何出错，是很实用的（且提高了代码可读性）。

下面的代码则很好地阐述了这一点。我们有一个简单的 parse 包（示例 13.4）用来把输入的字符串解析为整数切片；这个包有自己特殊的 `ParseError` 。

当没有东西需要转换或者转换成整数失败时，这个包会 panic（在函数 fields2numbers 中）。但是可导出的 Parse 函数会从 panic 中 recover 并用所有这些信息返回一个错误给调用者。为了演示这个过程，在 panic_recover.go 中 调用了 parse 包（示例 13.5）；不可解析的字符串会导致错误并被打印出来。

示例 13.4 parse.go：

```
// parse.go
package parse

import (
    "fmt"
    "strings"
    "strconv"
)

// A ParseError indicates an error in converting a word into an integer.
type ParseError struct {
    Index int      // The index into the space-separated list of words.
    Word  string   // The word that generated the parse error.
    Err error // The raw error that precipitated this error, if any.
}

// String returns a human-readable error message.
func (e *ParseError) String() string {
    return fmt.Sprintf("pkg parse: error parsing %q as int", e.Word)
}

// Parse parses the space-separated words in in put as integers.
func Parse(input string) (numbers []int, err error) {
    defer func() {
        if r := recover(); r != nil {
            var ok bool
            err, ok = r.(error)
            if !ok {
                err = fmt.Errorf("pkg: %v", r)
            }
        }
    }()

    fields := strings.Fields(input)
```

```
    numbers = fields2numbers(fields)
    return
}

func fields2numbers(fields []string) (numbers []int) {
    if len(fields) == 0 {
        panic("no words to parse")
    }
    for idx, field := range fields {
        num, err := strconv.Atoi(field)
        if err != nil {
            panic(&ParseError{idx, field, err})
        }
        numbers = append(numbers, num)
    }
    return
}
```

示例 13.5 panic_package.go：

```
// panic_package.go
package main

import (
    "fmt"
    "./parse/parse"
)

func main() {
    var examples = []string{
            "1 2 3 4 5",
            "100 50 25 12.5 6.25",
            "2 + 2 = 4",
            "1st class",
            "",
    }

    for _, ex := range examples {
        fmt.Printf("Parsing %q:\n  ", ex)
        nums, err := parse.Parse(ex)
        if err != nil {
            fmt.Println(err) // here String() method from ParseError is used
            continue
        }
        fmt.Println(nums)
    }
}
```

输出：

```
Parsing "1 2 3 4 5":
  [1 2 3 4 5]
Parsing "100 50 25 12.5 6.25":
  pkg: pkg parse: error parsing "12.5" as int
Parsing "2 + 2 = 4":
  pkg: pkg parse: error parsing "+" as int
Parsing "1st class":
  pkg: pkg parse: error parsing "1st" as int
```

```
Parsing "":
  pkg: no words to parse
```

一种用闭包处理错误的模式

13.5 一种用闭包处理错误的模式

每当函数返回时，我们应该检查是否有错误发生：但是这会导致重复乏味的代码。结合 defer/panic/recover 机制和闭包可以得到一个我们马上要讨论的更加优雅的模式。不过这个模式只有当所有的函数都是同一种签名时可用，这样就有相当大的限制。一个很好的使用它的例子是 web 应用，所有的处理函数都是下面这样：

```
func handler1(w http.ResponseWriter, r *http.Request) { ... }
```

假设所有的函数都有这样的签名：

```
func f(a type1, b type2)
```

参数的数量和类型是不相关的。

我们给这个类型一个名字：

```
fType1 = func f(a type1, b type2)
```

在我们的模式中使用了两个帮助函数：

1）check：这是用来检查是否有错误和 panic 发生的函数：

```
func check(err error) { if err != nil { panic(err) } }
```

2）errorhandler：这是一个包装函数。接收一个 fType1 类型的函数 fn 并返回一个调用 fn 的函数。里面就包含有 defer/recover 机制，这在 13.3 节中有相应描述。

```
func errorHandler(fn fType1) fType1 {
    return func(a type1, b type2) {
        defer func() {
            if err, ok := recover().(error); ok {
                log.Printf("run time panic: %v", err)
            }
        }()
        fn(a, b)
    }
}
```

当错误发生时会 recover 并打印在日志中；除了简单的打印，应用也可以用 template 包（参见 15.7 节）为用户生成自定义的输出。check() 函数会在所有的被调函数中调用，像这样：

```
func f1(a type1, b type2) {
    ...
    f, _, err := // call function/method
    check(err)
    t, err := // call function/method
    check(err)
    _, err2 := // call function/method
```

```
    check(err2)
    ...
}
```

通过这种机制，所有的错误都会被 recover，并且调用函数后的错误检查代码也被简化为调用 check(err) 即可。在这种模式下，不同的错误处理必须对应不同的函数类型；它们（错误处理）可能被隐藏在错误处理包内部。可选的更加通用的方式是用一个空接口类型的切片作为参数和返回值。

我们会在 15.5 节的 web 应用中使用这种模式。

练习

练习 13.1：recover_dividebyzero.go

用示例 13.3 中的编码模式通过整数除以 0 触发一个运行时 panic。

练习 13.2：panic_defer.go

阅读下面的完整程序。不要执行它，写出程序的输出结果。然后编译执行并验证你的预想。

```
// panic_defer.go
package main

import "fmt"

func main() {
    f()
    fmt.Println("Returned normally from f.")
}

func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
```

输出：

```
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
```

```
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.
```

练习 13.3：panic_defer_convint.go

写一个 ConvertInt64ToInt 函数把 int64 值转换为 int 值，如果发生错误（提示：参见 4.5.2.1 节）就 panic。然后在函数 IntFromInt64 中调用这个函数并 recover，返回一个整数和一个错误。请测试这个函数！

启动外部命令和程序

13.6 启动外部命令和程序

os 包有一个 `StartProcess` 函数可以调用或启动外部系统命令和二进制可执行文件；它的第一个参数是要运行的进程，第二个参数用来传递选项或参数，第三个参数是含有系统环境基本信息的结构体。

这个函数返回被启动进程的 id（pid），或者启动失败返回错误。

exec 包中也有同样功能的更简单的结构体和函数；主要是 `exec.Command(name string, arg ...string)` 和 `Run()`。首先需要用系统命令或可执行文件的名字创建一个 `Command` 对象，然后用这个对象作为接收者调用 `Run()`。下面的程序（因为是执行 Linux 命令，只能在 Linux 下面运行）演示了它们的使用：

示例 13.6 exec.go：

```
// exec.go
package main
import (
    "fmt"
    "os/exec"
    "os"
)

func main() {
// 1) os.StartProcess //
/******************/
/* Linux: */
env := os.Environ()
procAttr := &os.ProcAttr{
            Env: env,
            Files: []*os.File{
                os.Stdin,
                os.Stdout,
                os.Stderr,
            },
        }
// lst example: list files
pid, err := os.StartProcess("/bin/ls", []string{"ls", "-l"}, procAttr)
if err != nil {
        fmt.Printf("Error %v starting process!", err)  //
        os.Exit(1)
}
fmt.Printf("The process id is %v", pid)
```

输出：

```
The process id is &{2054 0}total 2056
-rwxr-xr-x 1 ivo ivo 1157555 2011-07-04 16:48 Mieken_exec
-rw-r--r-- 1 ivo ivo    2124 2011-07-04 16:48 Mieken_exec.go
-rw-r--r-- 1 ivo ivo   18528 2011-07-04 16:48 Mieken_exec_go_.6
-rwxr-xr-x 1 ivo ivo  913920 2011-06-03 16:13 panic.exe
-rw-r--r-- 1 ivo ivo     180 2011-04-11 20:39 panic.go
```

```
// 2nd example: show all processes
pid, err = os.StartProcess("/bin/ps", []string{"ps", "-e", "-opid,ppid,comm"}, procAttr)

if err != nil {
        fmt.Printf("Error %v starting process!", err)  //
        os.Exit(1)
}

fmt.Printf("The process id is %v", pid)
```

```
// 2) exec.Run //
/***************/
// Linux:  OK, but not for ls ?
// cmd := exec.Command("ls", "-l")  // no error, but doesn't show anything ?
// cmd := exec.Command("ls")           // no error, but doesn't show anything ?
    cmd := exec.Command("gedit")  // this opens a gedit-window
    err = cmd.Run()
    if err != nil {
        fmt.Printf("Error %v executing command!", err)
        os.Exit(1)
    }
    fmt.Printf("The command is %v", cmd)
// The command is &{/bin/ls [ls -l] []  <nil> <nil> <nil> 0xf840000210 <nil> true [0xf84000ea50 0xf84000e9f0
 0xf84000e9c0] [0xf84000ea50 0xf84000e9f0 0xf84000e9c0] [] [] 0xf8400128c0}
}
// in Windows: uitvoering: Error fork/exec /bin/ls: The system cannot find the path specified.  starting proce
ss!
```

Go 中的单元测试和基准测试

13.7 Go 中的单元测试和基准测试

首先所有的包都应该有一定的必要文档，然后同样重要的是对包的测试。

在第 3 章中提到了 Go 的测试工具 gotest， 我们已经在 9.8 节中使用过了。这里我们会用更多的例子进行详细说明。

名为 testing 的包被专门用来进行自动化测试，日志和错误报告。并且还包含一些基准测试函数的功能。

备注：gotest 是 Unix bash 脚本，所以在 Windows 下你需要配置 MINGW 环境（参见 2.5 节）；在 Windows 环境下把所有的 pkg/linux_amd64 替换成 pkg/windows。

对一个包做（单元）测试，需要写一些可以频繁（每次更新后）执行的小块测试单元来检查代码的正确性。于是我们必须写一些 Go 源文件来测试代码。测试程序必须属于被测试的包，并且文件名满足这种形式 `*_test.go` ，所以测试代码和包中的业务代码是分开的。

`_test` 程序不会被普通的 Go 编译器编译，所以当放应用部署到生产环境时它们不会被部署；只有 gotest 会编译所有的程序：普通程序和测试程序。

测试文件中必须导入 “testing” 包，并写一些名字以 `TestZzz` 打头的全局函数，这里的 `Zzz` 是被测试函数的字母描述，如 TestFmtInterface，TestPayEmployees 等。

测试函数必须有这种形式的头部：

```
func TestAbcde(t *testing.T)
```

T 是传给测试函数的结构类型，用来管理测试状态，支持格式化测试日志，如 t.Log，t.Error，t.ErrorF 等。在函数的结尾把输出跟想要的结果对比，如果不等就打印一个错误。成功的测试则直接返回。

用下面这些函数来通知测试失败：

1） `func (t *T) Fail()`

```
标记测试函数为失败，然后继续执行（剩下的测试）。
```

2） `func (t *T) FailNow()`

```
标记测试函数为失败并中止执行；文件中别的测试也被略过，继续执行下一个文件。
```

3） `func (t *T) Log(args ...interface{})`

```
args 被用默认的格式格式化并打印到错误日志中。
```

4） `func (t *T) Fatal(args ...interface{})`

```
结合 先执行 3），然后执行 2）的效果。
```

运行 go test 来编译测试程序，并执行程序中所有的 TestZZZ 函数。如果所有的测试都通过会打印出 PASS。

gotest 可以接收一个或多个函数程序作为参数，并指定一些选项。

结合 –chatty 或 -v 选项，每个执行的测试函数以及测试状态会被打印。

例如：

```
go test fmt_test.go --chatty
=== RUN fmt.TestFlagParser
--- PASS: fmt.TestFlagParser
=== RUN fmt.TestArrayPrinter
--- PASS: fmt.TestArrayPrinter
...
```

testing 包中有一些类型和函数可以用来做简单的基准测试；测试代码中必须包含以 `BenchmarkZzz` 打头的函数并接收一个 `*testing.B` 类型的参数，比如：

```
func BenchmarkReverse(b *testing.B) {
    ...
}
```

命令 `go test -test.bench=.*` 会运行所有的基准测试函数；代码中的函数会被调用 N 次（N是非常大的数，如 N = 1000000），并展示 N 的值和函数执行的平均时间，单位为 ns（纳秒，ns/op）。如果是用 testing.Benchmark 调用这些函数，直接运行程序即可。

具体可以参见 14.16 节中用 goroutines 运行基准测试的例子以及练习 13.4：string_reverse_test.go

测试的具体例子

13.8 测试的具体例子

在练习 9.4 中你写了一个叫 main_oddeven.go 的程序用来测试前 100 个整数是否是偶数。这个函数属于 even 包。

下面是一种可能的方案：

示例 13.7 even_main.go：

```
package main

import (
    "fmt"
    "even/even"
)

func main() {
    for i:=0; i<=100; i++ {
        fmt.Printf("Is the integer %d even? %v\n", i, even.Even(i))
    }
}
```

上面使用了 even.go 中的 even 包：

示例 13.8 even/even.go：

```
package even

func Even(i int) bool {         // Exported function
    return i%2 == 0
}

func Odd(i int) bool {          // Exported function
    return i%2 != 0
}
```

在 even 包的路径下，我们创建一个名为 oddeven_test.go 的测试程序：

示例 13.9 even/oddeven_test.go：

```
package even

import "testing"

func TestEven(t *testing.T) {
    if !Even(10) {
        t.Log(" 10 must be even!")
        t.Fail()
    }
    if Even(7) {
        t.Log(" 7 is not even!")
        t.Fail()
    }
```

```
}

func TestOdd(t *testing.T) {
    if !Odd(11) {
        t.Log(" 11 must be odd!")
        t.Fail()
    }
    if Odd(10) {
        t.Log(" 10 is not odd!")
        t.Fail()
    }
}
```

由于测试需要具体的输入用例且不可能测试到所有的用例（非常像一个无穷的数），所以我们必须对要使用的测试用例思考再三。

至少应该包括：

- 正常的用例
- 反面的用例（错误的输入，如用负数或字母代替数字，没有输入等）
- 边界检查用例（如果参数的取值范围是 0 到 1000，检查 0 和 1000 的情况）

可以直接执行 go install 安装 even 或者创建一个 以下内容的 Makefile：

```
include $(GOROOT)/src/Make.inc
TARG=even
GOFILES=\
      even.go\
include $(GOROOT)/src/Make.pkg
```

然后执行 make（或 gomake）命令来构建归档文件 even.a

测试代码不能在 GOFILES 参数中引用，因为我们不希望生成的程序中有测试代码。如果包含了测试代码，go test 会给出错误提示！go test 会生成一个单独的包含测试代码的 `_test` 程序。

现在我们可以用命令： `go test` （或 `make test` ）来测试 even 包。

因为示例 13.5 中的测试函数不会调用 t.Log 和 t.Fail，所以会得到一个 PASS 的结果。在这个简单例子中一切都正常执行。

为了看到失败时的输出，把函数 TestEven 改为：

```
func TestEven(t *testing.T) {
    if Even(10) {
        t.Log("Everything OK: 10 is even, just a test to see failed output!")
        t.Fail()
    }
}
```

现在会调用 t.Log 和 t.Fail，得到的结果如下：

```
--- FAIL: even.TestEven (0.00 seconds)
Everything OK: 10 is even, just a test to see failed output!
FAIL
```

练习 13.4：string_reverse_test.go

为练习 7.14 string_reverse.go 写一个单元测试。

把 string_reverse 放到自己的包 strev 中，只包含一个可导出函数 reverse。

实现并测试它。

用（测试数据）表驱动测试

13.9 用（测试数据）表驱动测试

编写测试代码时，一个较好的办法是把测试的输入数据和期望的结果写在一起组成一个数据表：表中的每条记录都是一个含有输入和期望值的完整测试用例，有时还可以结合像测试名字这样的额外信息来让测试输出更多的信息。

实际测试时简单迭代表中的每条记录，并执行必要的测试。这在练习 13.4 中有具体的应用。

可以抽象为下面的代码段：

```
var tests = []struct{     // Test table
    in  string
    out string

}{
    {"in1", "exp1"},
    {"in2", "exp2"},
    {"in3", "exp3"},
...
}

func TestFunction(t *testing.T) {
    for i, tt := range tests {
        s := FuncToBeTested(tt.in)
        if s != tt.out {
            t.Errorf("%d. %q => %q, wanted: %q", i, tt.in, s, tt.out)
        }
    }
}
```

如果大部分函数都可以写成这种形式，那么写一个帮助函数 verify 对实际测试会很有帮助：

```
func verify(t *testing.T, testnum int, testcase, input, output, expected string) {
    if expected != output {
        t.Errorf("%d. %s with input = %s: output %s != %s", testnum, testcase, input, output, expected)
    }
}
```

TestFunction 则变为：

```
func TestFunction(t *testing.T) {
    for i, tt := range tests {
        s := FuncToBeTested(tt.in)
        verify(t, i, “FuncToBeTested: “, tt.in, s, tt.out)
    }
}
```

性能调试：分析并优化 Go 程序

13.10 性能调试：分析并优化 Go 程序

13.10.1 时间和内存消耗

可以用这个便捷脚本 *xtime* 来测量：

```
#!/bin/sh
/usr/bin/time -f '%Uu %Ss %er %MkB %C' "$@"
```

在 Unix 命令行中像这样使用 `xtime goprogexec`，这里的 progexec 是一个 Go 可执行程序，这句命令行输出类似：56.63u 0.26s 56.92r 1642640kB progexec，分别对应用户时间，系统时间，实际时间和最大内存占用。

13.10.2 用 go test 调试

如果代码使用了 Go 中 testing 包的基准测试功能，我们可以用 gotest 标准的 `-cpuprofile` 和 `-memprofile` 标志向指定文件写入 CPU 或 内存使用情况报告。

使用方式： `go test -x -v -cpuprofile=prof.out -file x_test.go`

编译执行 x_test.go 中的测试，并向 prof.out 文件中写入 cpu 性能分析信息。

13.10.3 用 pprof 调试

你可以在单机程序 progexec 中引入 runtime/pprof 包；这个包以 pprof 可视化工具需要的格式写入运行时报告数据。对于 CPU 性能分析来说你需要添加一些代码：

```
var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to file")

func main() {
    flag.Parse()
    if *cpuprofile != "" {
        f, err := os.Create(*cpuprofile)
        if err != nil {
            log.Fatal(err)
        }
        pprof.StartCPUProfile(f)
        defer pprof.StopCPUProfile()
    }
...
```

代码定义了一个名为 cpuprofile 的 flag，调用 Go flag 库来解析命令行 flag，如果命令行设置了 cpuprofile flag，则开始 CPU 性能分析并把结果重定向到那个文件。（os.Create 用拿到的名字创建了用来写入分析数据的文件）。这个分析程序最后需要在程序退出之前调用 StopCPUProfile 来刷新挂起的写操作到文件中；我们用 defer 来保证这一切会在 main 返回时触发。

现在用这个 flag 运行程序： `progexec -cpuprofile=progexec.prof`

然后可以像这样用 gopprof 工具： `gopprof progexec progexec.prof`

gopprof 程序是 Google pprofC++ 分析器的一个轻微变种；关于此工具更多的信息，参见 https://github.com/gperftools/gperftools。

如果开启了 CPU 性能分析，Go 程序会以大约每秒 100 次的频率阻塞，并记录当前执行的 goroutine 栈上的程序计数器样本。

此工具一些有趣的命令：

1） `topN`

用来展示分析结果中最开头的 N 份样本，例如： `top5`
它会展示在程序运行期间调用最频繁的 5 个函数，输出如下：

```
Total: 3099 samples
626 20.2% 20.2% 626 20.2% scanblock
309 10.0% 30.2% 2839 91.6% main.FindLoops
...
```

第 5 列表示函数的调用频度。

2） `web` 或 `web 函数名`

该命令生成一份 SVG 格式的分析数据图表，并在网络浏览器中打开它（还有一个 gv 命令可以生成 PostScript 格式的数据，并在 GhostView 中打开，这个命令需要安装 graphviz）。函数被表示成不同的矩形（被调用越多，矩形越大），箭头指示函数调用链。

3） `list 函数名` 或 `weblist 函数名`

展示对应函数名的代码行列表，第 2 列表示当前行执行消耗的时间，这样就很好地指出了运行过程中消耗最大的代码。

如果发现函数 `runtime.mallocgc` （分配内存并执行周期性的垃圾回收）调用频繁，那么是应该进行内存分析的时候了。找出垃圾回收频繁执行的原因，和内存大量分配的根源。

为了做到这一点必须在合适的地方添加下面的代码：

```
var memprofile = flag.String("memprofile", "", "write memory profile to this file")
...

CallToFunctionWhichAllocatesLotsOfMemory()
if *memprofile != "" {
    f, err := os.Create(*memprofile)
    if err != nil {
        log.Fatal(err)
    }
    pprof.WriteHeapProfile(f)
    f.Close()
    return
}
```

用 -memprofile flag 运行这个程序： `progexec -memprofile=progexec.mprof`

然后你可以像这样再次使用 gopprof 工具： `gopprof progexec progexec.mprof`

`top5` ， `list 函数名` 等命令同样适用，只不过现在是以 Mb 为单位测量内存分配情况，这是 top 命令输出的例子：

```
Total: 118.3 MB
    66.1  55.8%  55.8%  103.7  87.7% main.FindLoops
```

```
    30.5 25.8% 81.6% 30.5 25.8% main.*LSG・NewLoop
    ...
```

从第 1 列可以看出，最上面的函数占用了最多的内存。

同样有一个报告内存分配计数的有趣工具：

```
gopprof --inuse_objects progexec progexec.mprof
```

对于 web 应用来说，有标准的 HTTP 接口可以分析数据。在 HTTP 服务中添加

```
import _ "http/pprof"
```

会为 /debug/pprof/ 下的一些 URL 安装处理器。然后你可以用一个唯一的参数——你服务中的分析数据的 URL 来执行 gopprof 命令——它会下载并执行在线分析。

```
gopprof http://localhost:6060/debug/pprof/profile # 30-second CPU profile
gopprof http://localhost:6060/debug/pprof/heap # heap profile
```

在 Go-blog（引用 15）中有一篇很好的文章用具体的例子进行了分析：分析 Go 程序（2011年6月）。

协程（goroutine）与通道（channel）

14.0 协程（goroutine）与通道（channel）

作为一门 21 世纪的语言，Go 原生支持应用之间的通信（网络，客户端和服务端，分布式计算，参见第 15 章）和程序的并发。程序可以在不同的处理器和计算机上同时执行不同的代码段。Go 语言为构建并发程序的基本代码块是 协程 (goroutine) 与通道 (channel)。他们需要语言，编译器，和runtime的支持。Go 语言提供的垃圾回收器对并发编程至关重要。

不要通过共享内存来通信，而通过通信来共享内存。

通信强制协作。

并发、并行和协程

14.1 并发、并行和协程

14.1.1 什么是协程

一个应用程序是运行在机器上的一个进程；进程是一个运行在自己内存地址空间里的独立执行体。一个进程由一个或多个操作系统线程组成，这些线程其实是共享同一个内存地址空间的一起工作的执行体。几乎所有'正式'的程序都是多线程的，以便让用户或计算机不必等待，或者能够同时服务多个请求（如 Web 服务器），或增加性能和吞吐量（例如，通过对不同的数据集并行执行代码）。一个并发程序可以在一个处理器或者内核上使用多个线程来执行任务，但是只有同一个程序在某个时间点同时运行在多核或者多处理器上才是真正的并行。

并行是一种通过使用多处理器以提高速度的能力。所以并发程序可以是并行的，也可以不是。

公认的，使用多线程的应用难以做到准确，最主要的问题是内存中的数据共享，它们会被多线程以无法预知的方式进行操作，导致一些无法重现或者随机的结果（称作 `竞态` ）。

不要使用全局变量或者共享内存，它们会给你的代码在并发运算的时候带来危险。

解决之道在于同步不同的线程，对数据加锁，这样同时就只有一个线程可以变更数据。在 Go 的标准库 `sync` 中有一些工具用来在低级别的代码中实现加锁；我们在第 9.3 节中讨论过这个问题。不过过去的软件开发经验告诉我们这会带来更高的复杂度，更容易使代码出错以及更低的性能，所以这个经典的方法明显不再适合现代多核/多处理器编程： `thread-per-connection` 模型不够有效。

Go 更倾向于其他的方式，在诸多比较合适的范式中，有个被称作 `Communicating Sequential Processes（顺序通信处理）` （CSP, C. Hoare 发明的）还有一个叫做 `message passing-model（消息传递）` （已经运用在了其他语言中，比如 Erlang）。

在 Go 中，应用程序并发处理的部分被称作 `goroutines（协程）` ，它可以进行更有效的并发运算。在协程和操作系统线程之间并无一对一的关系：协程是根据一个或多个线程的可用性，映射（多路复用，执行于）在他们之上的；协程调度器在 Go 运行时很好的完成了这个工作。

协程工作在相同的地址空间中，所以共享内存的方式一定是同步的；这个可以使用 `sync` 包来实现（参见第 9.3 节），不过我们很不鼓励这样做：Go 使用 `channels` 来同步协程（可以参见第 14.2 节等章节）

当系统调用（比如等待 I/O）阻塞协程时，其他协程会继续在其他线程上工作。协程的设计隐藏了许多线程创建和管理方面的复杂工作。

协程是轻量的，比线程更轻。它们痕迹非常不明显（使用少量的内存和资源）：使用 4K 的栈内存就可以在堆中创建它们。因为创建非常廉价，必要的时候可以轻松创建并运行大量的协程（在同一个地址空间中 100,000 个连续的协程）。并且它们对栈进行了分割，从而动态的增加（或缩减）内存的使用；栈的管理是自动的，但不是由垃圾回收器管理的，而是在协程退出后自动释放。

协程可以运行在多个操作系统线程之间，也可以运行在线程之内，让你可以很小的内存占用就可以处理大量的任务。由于操作系统线程上的协程时间片，你可以使用少量的操作系统线程就能拥有任意多个提供服务的协程，而且 Go 运行时可以聪明的意识到哪些协程被阻塞了，暂时搁置它们并处理其他协程。

存在两种并发方式：确定性的（明确定义排序）和非确定性的（加锁/互斥从而未定义排序）。Go 的协程和通道理所当然的支持确定性的并发方式（例如通道具有一个 sender 和一个 receiver）。我们会在第 14.7 节中使用一个常见的算法问题（工人问题）来对比两种处理方式。

协程是通过使用关键字 `go` 调用（执行）一个函数或者方法来实现的（也可以是匿名或者 lambda 函数）。这样会在当前的计算过程中开始一个同时进行的函数，在相同的地址空间中并且分配了独立的栈，比如： `go sum(bigArray)` ，在后台计算总和。

协程的栈会根据需要进行伸缩，不出现栈溢出；开发者不需要关心栈的大小。当协程结束的时候，它会静默退出：用来启动这个协程的函数不会得到任何的返回值。

任何 Go 程序都必须有的 `main()` 函数也可以看做是一个协程，尽管它并没有通过 `go` 来启动。协程可以在程序初始化的过程中运行（在 `init()` 函数中）。

在一个协程中，比如它需要进行非常密集的运算，你可以在运算循环中周期的使用 `runtime.Gosched()`：这会让出处理器，允许运行其他协程；它并不会使当前协程挂起，所以它会自动恢复执行。使用 `Gosched()` 可以使计算均匀分布，使通信不至于迟迟得不到响应。

14.1.2 并发和并行的差异

Go 的并发原语提供了良好的并发设计基础：表达程序结构以便表示独立地执行的动作；所以Go的重点不在于并行的首要位置：并发程序可能是并行的，也可能不是。并行是一种通过使用多处理器以提高速度的能力。但往往是，一个设计良好的并发程序在并行方面的表现也非常出色。

在当前的运行时（2012 年一月）实现中，Go 默认没有并行指令，只有一个独立的核心或处理器被专门用于 Go 程序，不论它启动了多少个协程；所以这些协程是并发运行的，但他们不是并行运行的：同一时间只有一个协程会处在运行状态。

这个情况在以后可能会发生改变，不过届时，为了使你的程序可以使用多个核心运行，这时协程就真正的是并行运行了，你必须使用 `GOMAXPROCS` 变量。

这会告诉运行时有多少个协程同时执行。

并且只有 gc 编译器真正实现了协程，适当的把协程映射到操作系统线程。使用 `gccgo` 编译器，会为每一个协程创建操作系统线程。

14.1.3 使用 GOMAXPROCS

在 gc 编译器下（6g 或者 8g）你必须设置 GOMAXPROCS 为一个大于默认值 1 的数值来允许运行时支持使用多于 1 个的操作系统线程，所有的协程都会共享同一个线程除非将 GOMAXPROCS 设置为一个大于 1 的数。当 GOMAXPROCS 大于 1 时，会有一个线程池管理许多的线程。通过 `gccgo` 编译器 GOMAXPROCS 有效的与运行中的协程数量相等。假设 n 是机器上处理器或者核心的数量。如果你设置环境变量 GOMAXPROCS>=n，或者执行 `runtime.GOMAXPROCS(n)`，接下来协程会被分割（分散）到 n 个处理器上。更多的处理器并不意味着性能的线性提升。有这样一个经验法则，对于 n 个核心的情况设置 GOMAXPROCS 为 n-1 以获得最佳性能，也同样需要遵守这条规则：协程的数量 > 1 + GOMAXPROCS > 1。

所以如果在某一时间只有一个协程在执行，不要设置 GOMAXPROCS！

还有一些通过实验观察到的现象：在一台 1 颗 CPU 的笔记本电脑上，增加 GOMAXPROCS 到 9 会带来性能提升。在一台 32 核的机器上，设置 GOMAXPROCS=8 会达到最好的性能，在测试环境中，更高的数值无法提升性能。如果设置一个很大的 GOMAXPROCS 只会带来轻微的性能下降；设置 GOMAXPROCS=100，使用 `top` 命令和 `H` 选项查看到只有 7 个活动的线程。

增加 GOMAXPROCS 的数值对程序进行并发计算是有好处的；

请看 goroutine_select2.go

总结：GOMAXPROCS 等同于（并发的）线程数量，在一台核心数多于1个的机器上，会尽可能有等同于核心数的线程在并行运行。

14.1.4 如何用命令行指定使用的核心数量

使用 `flags` 包，如下：

```
var numCores = flag.Int("n", 2, "number of CPU cores to use")
```

在 main() 中：

```
flag.Parse()
runtime.GOMAXPROCS(*numCores)
```

协程可以通过调用 `runtime.Goexit()` 来停止，尽管这样做几乎没有必要。

示例 14.1-goroutine1.go 介绍了概念：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("In main()")
    go longWait()
    go shortWait()
    fmt.Println("About to sleep in main()")
    // sleep works with a Duration in nanoseconds (ns) !
    time.Sleep(10 * 1e9)
    fmt.Println("At the end of main()")
}

func longWait() {
    fmt.Println("Beginning longWait()")
    time.Sleep(5 * 1e9) // sleep for 5 seconds
    fmt.Println("End of longWait()")
}

func shortWait() {
    fmt.Println("Beginning shortWait()")
    time.Sleep(2 * 1e9) // sleep for 2 seconds
    fmt.Println("End of shortWait()")
}
```

输出：

```
In main()
About to sleep in main()
Beginning longWait()
Beginning shortWait()
End of shortWait()
End of longWait()
At the end of main() // after 10s
```

`main()` ， `longWait()` 和 `shortWait()` 三个函数作为独立的处理单元按顺序启动，然后开始并行运行。每一个函数都在运行的开始和结束阶段输出了消息。为了模拟他们运算的时间消耗，我们使用了 `time` 包中的 `Sleep` 函数。 `Sleep()` 可以按照指定的时间来暂停函数或协程的执行，这里使用了纳秒（ns，符号 1e9 表示 1 乘 10 的 9 次方，e=指数）。

他们按照我们期望的顺序打印出了消息，几乎都一样，可是我们明白这是模拟出来的，以并行的方式。我们让 `main()` 函数暂停 10 秒从而确定它会在另外两个协程之后结束。如果不这样（如果我们让 `main()` 函数停止 4 秒）， `main()` 会提前结束， `longWait()` 则无法完成。如果我们不在 `main()` 中等待，协程会随着程序的结束而消亡。

当 `main()` 函数返回的时候，程序退出：它不会等待任何其他非 main 协程的结束。这就是为什么在服务器程序中，每一个请求都会启动一个协程来处理，`server()` 函数必须保持运行状态。通常使用一个无限循环来达到这样的目的。

另外，协程是独立的处理单元，一旦陆续启动一些协程，你无法确定他们是什么时候真正开始执行的。你的代码逻辑必须独立于协程调用的顺序。

为了对比使用一个线程，连续调用的情况，移除 go 关键字，重新运行程序。

现在输出：

```
In main()
Beginning longWait()
End of longWait()
Beginning shortWait()
End of shortWait()
About to sleep in main()
At the end of main() // after 17 s
```

协程更有用的一个例子应该是在一个非常长的数组中查找一个元素。

将数组分割为若干个不重复的切片，然后给每一个切片启动一个协程进行查找计算。这样许多并行的协程可以用来进行查找任务，整体的查找时间会缩短（除以协程的数量）。

14.1.5 Go 协程（goroutines）和协程（coroutines）

（译者注：标题中的“Go协程（goroutines）”即是 14 章讲的协程指的是 Go 语言中的协程。而“协程（coroutines）”指的是其他语言中的协程概念，仅在本节出现。）

在其他语言中，比如 C#，Lua 或者 Python 都有协程的概念。这个名字表明它和 Go协程有些相似，不过有两点不同：

- Go 协程意味着并行（或者可以以并行的方式部署），协程一般来说不是这样的
- Go 协程通过通道来通信；协程通过让出和恢复操作来通信

Go 协程比协程更强大，也很容易从协程的逻辑复用到 Go 协程。

协程间的信道

14.2 协程间的信道

14.2.1 概念

在第一个例子中，协程是独立执行的，他们之间没有通信。他们必须通信才会变得更有用：彼此之间发送和接收信息并且协调/同步他们的工作。协程可以使用共享变量来通信，但是很不提倡这样做，因为这种方式给所有的共享内存的多线程都带来了困难。

而 Go 有一种特殊的类型，*通道（channel）*，就像一个可以用于发送类型化数据的管道，由其负责协程之间的通信，从而避开所有由共享内存导致的陷阱；这种通过通道进行通信的方式保证了同步性。数据在通道中进行传递：*在任何给定时间，一个数据被设计为只有一个协程可以对其访问，所以不会发生数据竞争。* 数据的所有权（可以读写数据的能力）也因此被传递。

工厂的传送带是个很有用的例子。一个机器（生产者协程）在传送带上放置物品，另外一个机器（消费者协程）拿到物品并打包。

通道服务于通信的两个目的：值的交换，同步的，保证了两个计算（协程）任何时候都是可知状态。

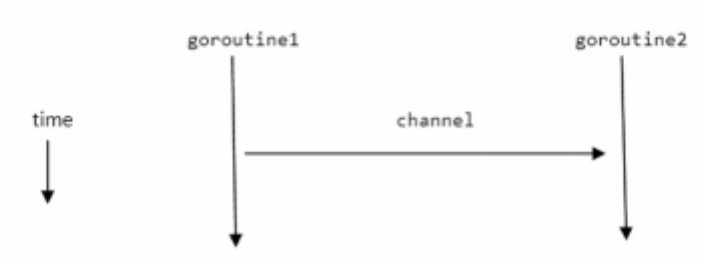


Fig 14.1: Channels and goroutines

通常使用这样的格式来声明通道：`var identifier chan datatype`

未初始化的通道的值是nil。

所以通道只能传输一种类型的数据，比如 `chan int` 或者 `chan string`，所有的类型都可以用于通道，空接口 `interface{}` 也可以。甚至可以（有时非常有用）创建通道的通道。

通道实际上是类型化消息的队列：使数据得以传输。它是先进先出（FIFO）的结构所以可以保证发送给他们的元素的顺序（有些人知道，通道可以比作 Unix shells 中的双向管道（two-way pipe））。通道也是引用类型，所以我们使用 `make()` 函数来给它分配内存。这里先声明了一个字符串通道 ch1，然后创建了它（实例化）：

```
var ch1 chan string
ch1 = make(chan string)
```

当然可以更短：`ch1 := make(chan string)`。

这里我们构建一个int通道的通道：`chanOfChans := make(chan int)`。

或者函数通道：`funcChan := make(chan func())`（相关示例请看第 14.17 节）。

所以通道是第一类对象：可以存储在变量中，作为函数的参数传递，从函数返回以及通过通道发送它们自身。另外它们是类型化的，允许类型检查，比如尝试使用整数通道发送一个指针。

14.2.2 通信操作符 <-

这个操作符直观的标示了数据的传输：信息按照箭头的方向流动。

流向通道（发送）

`ch <- int1` 表示：用通道 ch 发送变量 int1（双目运算符，中缀 = 发送）

从通道流出（接收），三种方式：

`int2 = <- ch` 表示：变量 int2 从通道 ch（一元运算的前缀操作符，前缀 = 接收）接收数据（获取新值）；假设 int2 已经声明过了，如果没有的话可以写成： `int2 := <- ch` 。

`<- ch` 可以单独调用获取通道的（下一个）值，当前值会被丢弃，但是可以用来验证，所以以下代码是合法的：

```
if <- ch != 1000{
    ...
}
```

同一个操作符 `<-` 既用于**发送**也用于**接收**，但Go会根据操作对象弄明白该干什么 。虽非强制要求，但为了可读性通道的命名通常以 `ch` 开头或者包含 `chan` 。通道的发送和接收都是原子操作：它们总是互不干扰的完成的。下面的示例展示了通信操作符的使用。

示例 14.2-goroutine2.go

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan string)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1e9)
}

func sendData(ch chan string) {
    ch <- "Washington"
    ch <- "Tripoli"
    ch <- "London"
    ch <- "Beijing"
    ch <- "Tokyo"
}

func getData(ch chan string) {
    var input string
    // time.Sleep(2e9)
    for {
        input = <-ch
        fmt.Printf("%s ", input)
```

```
    }
}
```

输出：

```
Washington Tripoli London Beijing tokyo
```

`main()` 函数中启动了两个协程：`sendData()` 通过通道 ch 发送了 5 个字符串，`getData()` 按顺序接收它们并打印出来。

如果 2 个协程需要通信，你必须给他们同一个通道作为参数才行。

尝试一下如果注释掉 `time.Sleep(1e9)` 会如何。

我们发现协程之间的同步非常重要：

- main() 等待了 1 秒让两个协程完成，如果不这样，sendData() 就没有机会输出。
- getData() 使用了无限循环：它随着 sendData() 的发送完成和 ch 变空也结束了。
- 如果我们移除一个或所有 `go` 关键字，程序无法运行，Go 运行时会抛出 panic：

```
---- Error run E:/Go/Goboek/code examples/chapter 14/goroutine2.exe with code Crashed ---- Program exited wit
h code -2147483645: panic: all goroutines are asleep-deadlock!
```

为什么会这样？运行时（runtime）会检查所有的协程（像本例中只有一个）是否在等待着什么东西（可从某个通道读取或者写入某个通道），这意味着程序将无法继续执行。这是死锁（deadlock）的一种形式，而运行时（runtime）可以为我们检测到这种情况。

注意：不要使用打印状态来表明通道的发送和接收顺序：由于打印状态和通道实际发生读写的时间延迟会导致和真实发生的顺序不同。

练习 14.4：解释一下为什么如果在函数 `getData()` 的一开始插入 `time.Sleep(2e9)`，不会出现错误但也没有输出呢。

14.2.3 通道阻塞

默认情况下，通信是同步且无缓冲的：在有接受者接收数据之前，发送不会结束。可以想象一个无缓冲的通道在没有空间来保存数据的时候：必须要一个接收者准备好接收通道的数据然后发送者可以直接把数据发送给接收者。所以通道的发送/接收操作在对方准备好之前是阻塞的：

1）对于同一个通道，发送操作（协程或者函数中的），在接收者准备好之前是阻塞的：如果ch中的数据无人接收，就无法再给通道传入其他数据：新的输入无法在通道非空的情况下传入。所以发送操作会等待 ch 再次变为可用状态：就是通道值被接收时（可以传入变量）。

2）对于同一个通道，接收操作是阻塞的（协程或函数中的），直到发送者可用：如果通道中没有数据，接收者就阻塞了。

尽管这看上去是非常严格的约束，实际在大部分情况下工作的很不错。

程序 `channel_block.go` 验证了以上理论，一个协程在无限循环中给通道发送整数数据。不过因为没有接收者，只输出了一个数字 0。

示例 14.3-channel_block.go

```
package main

import "fmt"
```

```
func main() {
    ch1 := make(chan int)
    go pump(ch1)        // pump hangs
    fmt.Println(<-ch1) // prints only 0
}

func pump(ch chan int) {
    for i := 0; ; i++ {
        ch <- i
    }
}
```

输出：

```
0
```

`pump()` 函数为通道提供数值，也被叫做生产者。

为通道解除阻塞定义了 `suck` 函数来在无限循环中读取通道，参见示例 14.4-channel_block2.go：

```
func suck(ch chan int) {
    for {
        fmt.Println(<-ch)
    }
}
```

在 `main()` 中使用协程开始它：

```
go pump(ch1)
go suck(ch1)
time.Sleep(1e9)
```

给程序 1 秒的时间来运行：输出了上万个整数。

练习 14.1：channel_block3.go：写一个通道证明它的阻塞性，开启一个协程接收通道的数据，持续 15 秒，然后给通道放入一个值。在不同的阶段打印消息并观察输出。

14.2.4 通过一个（或多个）通道交换数据进行协程同步。

通信是一种同步形式：通过通道，两个协程在通信（协程会和）中某刻同步交换数据。无缓冲通道成为了多个协程同步的完美工具。

甚至可以在通道两端互相阻塞对方，形成了叫做死锁的状态。Go 运行时会检查并 panic，停止程序。死锁几乎完全是由糟糕的设计导致的。

无缓冲通道会被阻塞。设计无阻塞的程序可以避免这种情况，或者使用带缓冲的通道。

练习 14.2： blocking.go

解释为什么下边这个程序会导致 panic：所有的协程都休眠了 - 死锁！

```
package main

import (
```

```
    "fmt"
)

func f1(in chan int) {
    fmt.Println(<-in)
}

func main() {
    out := make(chan int)
    out <- 2
    go f1(out)
}
```

14.2.5 同步通道-使用带缓冲的通道

一个无缓冲通道只能包含 1 个元素，有时显得很局限。我们给通道提供了一个缓存，可以在扩展的 `make` 命令中设置它的容量，如下：

```
buf := 100
ch1 := make(chan string, buf)
```

buf 是通道可以同时容纳的元素（这里是 string）个数

在缓冲满载（缓冲被全部使用）之前，给一个带缓冲的通道发送数据是不会阻塞的，而从通道读取数据也不会阻塞，直到缓冲空了。

缓冲容量和类型无关，所以可以（尽管可能导致危险）给一些通道设置不同的容量，只要他们拥有同样的元素类型。内置的 `cap` 函数可以返回缓冲区的容量。

如果容量大于 0，通道就是异步的了：缓冲满载（发送）或变空（接收）之前通信不会阻塞，元素会按照发送的顺序被接收。如果容量是0或者未设置，通信仅在收发双方准备好的情况下才可以成功。

同步： `ch :=make(chan type, value)`

- value == 0 -> synchronous, unbuffered (阻塞)
- value > 0 -> asynchronous, buffered（非阻塞）取决于value元素

若使用通道的缓冲，你的程序会在“请求”激增的时候表现更好：更具弹性，专业术语叫：更具有伸缩性（scalable）。在设计算法时首先考虑使用无缓冲通道，只在不确定的情况下使用缓冲。

练习 14.3：channel_buffer.go：给 channel_block3.go 的通道增加缓冲并观察输出有何不同。

14.2.6 协程中用通道输出结果

为了知道计算何时完成，可以通过信道回报。在例子 `go sum(bigArray)` 中，要这样写：

```
ch := make(chan int)
go sum(bigArray, ch) // bigArray puts the calculated sum on ch
// .. do something else for a while
sum := <- ch // wait for, and retrieve the sum
```

也可以使用通道来达到同步的目的，这个很有效的用法在传统计算机中称为信号量（semaphore）。或者换个方式：通过通道发送信号告知处理已经完成（在协程中）。

在其他协程运行时让 main 程序无限阻塞的通常做法是在 `main` 函数的最后放置一个 `select {}` 。

也可以使用通道让 `main` 程序等待协程完成，就是所谓的信号量模式，我们会在接下来的部分讨论。

14.2.7 信号量模式

下边的片段阐明：协程通过在通道 `ch` 中放置一个值来处理结束的信号。 `main` 协程等待 `<-ch` 直到从中获取到值。

我们期望从这个通道中获取返回的结果，像这样：

```
func compute(ch chan int){
    ch <- someComputation() // when it completes, signal on the channel.
}

func main(){
    ch := make(chan int)     // allocate a channel.
    go compute(ch)        // start something in a goroutines
    doSomethingElseForAWhile()
    result := <- ch
}
```

这个信号也可以是其他的，不返回结果，比如下面这个协程中的匿名函数（lambda）协程：

```
ch := make(chan int)
go func(){
    // doSomething
    ch <- 1 // Send a signal; value does not matter
}()
doSomethingElseForAWhile()
<- ch    // Wait for goroutine to finish; discard sent value.
```

或者等待两个协程完成，每一个都会对切片s的一部分进行排序，片段如下：

```
done := make(chan bool)
// doSort is a lambda function, so a closure which knows the channel done:
doSort := func(s []int){
    sort(s)
    done <- true
}
i := pivot(s)
go doSort(s[:i])
go doSort(s[i:])
<-done
<-done
```

下边的代码，用完整的信号量模式对长度为N的 float64 切片进行了 N 个 `doSomething()` 计算并同时完成，通道 sem 分配了相同的长度（且包含空接口类型的元素），待所有的计算都完成后，发送信号（通过放入值）。在循环中从通道 sem 不停的接收数据来等待所有的协程完成。

```
type Empty interface {}
var empty Empty
...
data := make([]float64, N)
res := make([]float64, N)
sem := make(chan Empty, N)
...
for i, xi := range data {
```

```
    go func (i int, xi float64) {
        res[i] = doSomething(i, xi)
        sem <- empty
    } (i, xi)
}
// wait for goroutines to finish
for i := 0; i < N; i++ { <-sem }
```

注意上述代码中闭合函数的用法：`i`、`xi`都是作为参数传入闭合函数的，这一做法使得每个协程（译者注：在其启动时）获得一份`i`和`xi`的单独拷贝，从而向闭合函数内部屏蔽了外层循环中的`i`和`xi`变量；否则，for 循环的下一次迭代会更新所有协程中`i`和`xi`的值。另一方面，切片`res`没有传入闭合函数，因为协程不需要`res`的单独拷贝。切片`res`也在闭合函数中但并不是参数。

14.2.8 实现并行的 for 循环

在上一部分章节 14.2.7 的代码片段中：for 循环的每一个迭代是并行完成的：

```
for i, v := range data {
    go func (i int, v float64) {
        doSomething(i, v)
        ...
    } (i, v)
}
```

在 for 循环中并行计算迭代可能带来很好的性能提升。不过所有的迭代都必须是独立完成的。有些语言比如 Fortress 或者其他并行框架以不同的结构实现了这种方式，在 Go 中用协程实现起来非常容易：

14.2.9 用带缓冲通道实现一个信号量

信号量是实现互斥锁（排外锁）常见的同步机制，限制对资源的访问，解决读写问题，比如没有实现信号量的`sync`的 Go 包，使用带缓冲的通道可以轻松实现：

- 带缓冲通道的容量和要同步的资源容量相同
- 通道的长度（当前存放的元素个数）与当前资源被使用的数量相同
- 容量减去通道的长度就是未处理的资源个数（标准信号量的整数值）

不用管通道中存放的是什么，只关注长度；因此我们创建了一个长度可变但容量为0（字节）的通道：

```
type Empty interface {}
type semaphore chan Empty
```

将可用资源的数量N来初始化信号量`semaphore`：`sem = make(semaphore, N)`

然后直接对信号量进行操作：

```
// acquire n resources
func (s semaphore) P(n int) {
    e := new(Empty)
    for i := 0; i < n; i++ {
        s <- e
    }
}

// release n resources
```

```
func (s semaphore) V(n int) {
    for i:= 0; i < n; i++{
        <- s
    }
}
```

可以用来实现一个互斥的例子：

```
/* mutexes */
func (s semaphore) Lock() {
    s.P(1)
}

func (s semaphore) Unlock(){
    s.V(1)
}

/* signal-wait */
func (s semaphore) Wait(n int) {
    s.P(n)
}

func (s semaphore) Signal() {
    s.V(1)
}
```

练习 14.5：gosum.go：用这种习惯用法写一个程序，开启一个协程来计算2个整数的和并等待计算结果并打印出来。

练习 14.6：producer_consumer.go：用这种习惯用法写一个程序，有两个协程，第一个提供数字 0，10，20，...90 并将他们放入通道，第二个协程从通道中读取并打印。 `main()` 等待两个协程完成后再结束。

习惯用法：通道工厂模式

编程中常见的另外一种模式如下：不将通道作为参数传递给协程，而用函数来生成一个通道并返回（工厂角色）；函数内有个匿名函数被协程调用。

在 channel_block2.go 加入这种模式便有了示例 14.5-channel_idiom.go：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    stream := pump()
    go suck(stream)
    time.Sleep(1e9)
}

func pump() chan int {
    ch := make(chan int)
    go func() {
        for i := 0; ; i++ {
            ch <- i
        }
    }()
    return ch
```

```
}

func suck(ch chan int) {
    for {
        fmt.Println(<-ch)
    }
}
```

14.2.10 给通道使用 for 循环

`for` 循环的 `range` 语句可以用在通道 `ch` 上，便可以从通道中获取值，像这样：

```
for v := range ch {
    fmt.Printf("The value is %v\n", v)
}
```

它从指定通道中读取数据直到通道关闭，才继续执行下边的代码。很明显，另外一个协程必须写入 `ch`（不然代码就阻塞在 for 循环了），而且必须在写入完成后才关闭。 `suck` 函数可以这样写，且在协程中调用这个动作，程序变成了这样：

示例 14.6-channel_idiom2.go：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    suck(pump())
    time.Sleep(1e9)
}

func pump() chan int {
    ch := make(chan int)
    go func() {
        for i := 0; ; i++ {
            ch <- i
        }
    }()
    return ch
}

func suck(ch chan int) {
    go func() {
        for v := range ch {
            fmt.Println(v)
        }
    }()
}
```

习惯用法：通道迭代模式

这个模式用到了后边14.6章示例 producer_consumer.go 的生产者-消费者模式，通常，需要从包含了地址索引字段 items 的容器给通道填入元素。为容器的类型定义一个方法 `Iter()`，返回一个只读的通道（参见第 14.2.11 节）items，如下：

```
func (c *container) Iter () <- chan item {
    ch := make(chan item)
    go func () {
        for i:= 0; i < c.Len(); i++{    // or use a for-range loop
            ch <- c.items[i]
        }
    } ()
    return ch
}
```

在协程里，一个 for 循环迭代容器 c 中的元素（对于树或图的算法，这种简单的 for 循环可以替换为深度优先搜索）。

调用这个方法的代码可以这样迭代容器：

```
for x := range container.Iter() { ... }
```

其运行在自己启动的协程中，所以上边的迭代用到了一个通道和两个协程（可能运行在不同的线程上）。 这样我们就有了一个典型的生产者-消费者模式。如果在程序结束之前，向通道写值的协程未完成工作，则这个协程不会被垃圾回收；这是设计使然。这种看起来并不符合预期的行为正是由通道这种线程安全的通信方式所导致的。如此一来，一个协程为了写入一个永远无人读取的通道而被挂起就成了一个bug，而并非你预想中的那样被悄悄回收掉（garbage-collected）了。

习惯用法：生产者消费者模式

假设你有 `Produce()` 函数来产生 `Consume` 函数需要的值。它们都可以运行在独立的协程中，生产者在通道中放入给消费者读取的值。整个处理过程可以替换为无限循环：

```
for {
    Consume(Produce())
}
```

14.2.11 通道的方向

通道类型可以用注解来表示它只发送或者只接收：

```
var send_only chan<- int        // channel can only receive data
var recv_only <-chan int        // channel can only send data
```

只接收的通道（<-chan T）无法关闭，因为关闭通道是发送者用来表示不再给通道发送值了，所以对只接收通道是没有意义的。通道创建的时候都是双向的，但也可以分配有方向的通道变量，就像以下代码：

```
var c = make(chan int) // bidirectional
go source(c)
go sink(c)

func source(ch chan<- int){
    for { ch <- 1 }
}

func sink(ch <-chan int) {
    for { <-ch }
}
```

习惯用法：管道和选择器模式

更具体的例子还有协程处理它从通道接收的数据并发送给输出通道：

```
sendChan := make(chan int)
receiveChan := make(chan string)
go processChannel(sendChan, receiveChan)

func processChannel(in <-chan int, out chan<- string) {
    for inValue := range in {
        result := ... /// processing inValue
        out <- result
    }
}
```

通过使用方向注解来限制协程对通道的操作。

这里有一个来自 Go 指导的很赞的例子，打印了输出的素数，使用选择器（‘筛’）作为它的算法。每个 prime 都有一个选择器，如下图：

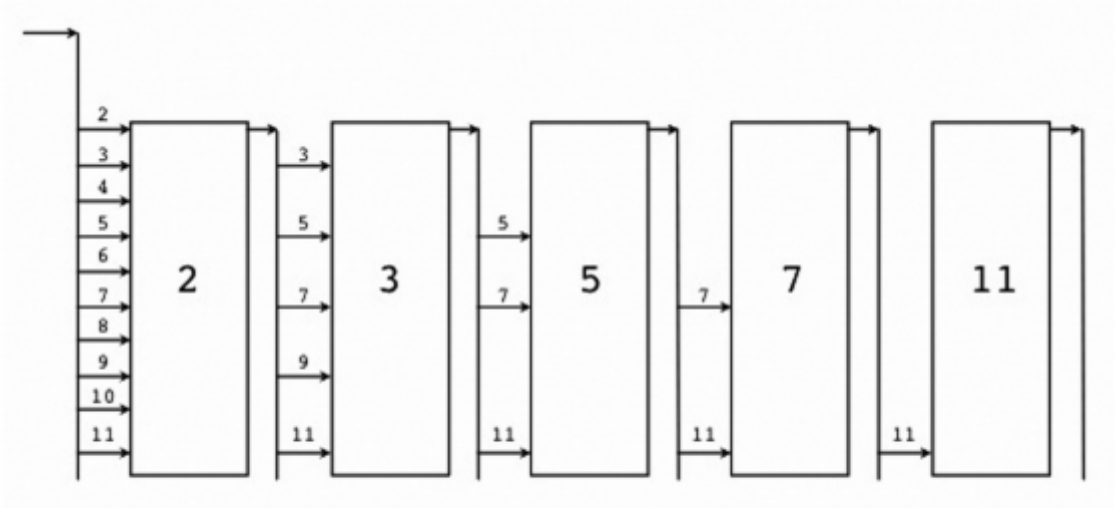


Fig 14.2: The sieve prime-algorithm

版本1：示例 14.7-sieve1.go

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.package main
package main

import "fmt"

// Send the sequence 2, 3, 4, ... to channel 'ch'.
func generate(ch chan int) {
    for i := 2; ; i++ {
        ch <- i // Send 'i' to channel 'ch'.
    }
}

// Copy the values from channel 'in' to channel 'out',
// removing those divisible by 'prime'.
func filter(in, out chan int, prime int) {
    for {
        i := <-in // Receive value of new variable 'i' from 'in'.
```

```
        if i%prime != 0 {
            out <- i // Send 'i' to channel 'out'.
        }
    }
}

// The prime sieve: Daisy-chain filter processes together.
func main() {
    ch := make(chan int) // Create a new channel.
    go generate(ch)      // Start generate() as a goroutine.
    for {
        prime := <-ch
        fmt.Print(prime, " ")
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}
```

协程 `filter(in, out chan int, prime int)` 拷贝整数到输出通道，丢弃掉可以被 prime 整除的数字。然后每个 prime 又开启了一个新的协程，生成器和选择器并发请求。

输出：

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479
487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619
631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769
773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929
937 941 947 953 967 971 977 983 991 997 1009 1013...
```

第二个版本引入了上边的习惯用法：函数 `sieve` 、 `generate` 和 `filter` 都是工厂；它们创建通道并返回，而且使用了协程的 lambda 函数。 `main` 函数现在短小清晰：它调用 `sieve()` 返回了包含素数的通道，然后通过 `fmt.Println(<-primes)` 打印出来。

版本2：示例 14.8-sieve2.go

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

package main

import (
    "fmt"
)

// Send the sequence 2, 3, 4, ... to returned channel
func generate() chan int {
    ch := make(chan int)
    go func() {
        for i := 2; ; i++ {
            ch <- i
        }
    }()
    return ch
}
```

```
// Filter out input values divisible by 'prime', send rest to returned channel
func filter(in chan int, prime int) chan int {
    out := make(chan int)
    go func() {
        for {
            if i := <-in; i%prime != 0 {
                out <- i
            }
        }
    }()
    return out
}

func sieve() chan int {
    out := make(chan int)
    go func() {
        ch := generate()
        for {
            prime := <-ch
            ch = filter(ch, prime)
            out <- prime
        }
    }()
    return out
}

func main() {
    primes := sieve()
    for {
        fmt.Println(<-primes)
    }
}
```

协程的同步：关闭通道-测试阻塞的通道

14.3 协程的同步：关闭通道-测试阻塞的通道

通道可以被显式的关闭；尽管它们和文件不同：不必每次都关闭。只有在当需要告诉接收者不会再提供新的值的时候，才需要关闭通道。只有发送者需要关闭通道，接收者永远不会需要。

继续看示例 goroutine2.go（示例 14.2）：我们如何在通道的 `sendData()` 完成的时候发送一个信号，`getData()` 又如何检测到通道是否关闭或阻塞？

第一个可以通过函数 `close(ch)` 来完成：这个将通道标记为无法通过发送操作 `<-` 接受更多的值；给已经关闭的通道发送或者再次关闭都会导致运行时的 panic。在创建一个通道后使用 defer 语句是个不错的办法（类似这种情况）：

```
ch := make(chan float64)
defer close(ch)
```

第二个问题可以使用逗号，ok 操作符：用来检测通道是否被关闭。

如何来检测可以收到没有被阻塞（或者通道没有被关闭）？

```
v, ok := <-ch   // ok is true if v received value
```

通常和 if 语句一起使用：

```
if v, ok := <-ch; ok {
  process(v)
}
```

或者在 for 循环中接收的时候，当关闭或者阻塞的时候使用 break：

```
v, ok := <-ch
if !ok {
  break
}
process(v)
```

在示例程序 14.2 中使用这些可以改进为版本 goroutine3.go，输出相同。

实现非阻塞通道的读取，需要使用 select（参见第 14.4 节）。

示例 14.9-goroutine3.go：

```
package main

import "fmt"

func main() {
    ch := make(chan string)
    go sendData(ch)
    getData(ch)
}
```

```
func sendData(ch chan string) {
    ch <- "Washington"
    ch <- "Tripoli"
    ch <- "London"
    ch <- "Beijing"
    ch <- "Tokio"
    close(ch)
}

func getData(ch chan string) {
    for {
        input, open := <-ch
        if !open {
            break
        }
        fmt.Printf("%s ", input)
    }
}
```

改变了以下代码：

- 现在只有 `sendData()` 是协程，`getData()` 和 `main()` 在同一个线程中：

```
go sendData(ch)
getData(ch)
```

- 在 `sendData()` 函数的最后，关闭了通道：

```
func sendData(ch chan string) {
    ch <- "Washington"
    ch <- "Tripoli"
    ch <- "London"
    ch <- "Beijing"
    ch <- "Tokio"
    close(ch)
}
```

- 在 for 循环的 `getData()` 中，在每次接收通道的数据之前都使用 `if !open` 来检测：

```
for {
        input, open := <-ch
        if !open {
            break
        }
        fmt.Printf("%s ", input)
    }
```

使用 for-range 语句来读取通道是更好的办法，因为这会自动检测通道是否关闭：

```
for input := range ch {
    process(input)
}
```

阻塞和生产者-消费者模式：

在第 14.2.10 节的通道迭代器中，两个协程经常是一个阻塞另外一个。如果程序工作在多核心的机器上，大部分时间只用到了一个处理器。可以通过使用带缓冲（缓冲空间大于 0）的通道来改善。比如，缓冲大小为 100，迭代器在阻塞之前，至少可以从容器获得 100 个元素。如果消费者协程在独立的内核运行，就有可能让协程不会出现阻塞。

由于容器中元素的数量通常是已知的，需要让通道有足够的容量放置所有的元素。这样，迭代器就不会阻塞（尽管消费者协程仍然可能阻塞）。然而，这实际上加倍了迭代容器所需要的内存使用量，所以通道的容量需要限制一下最大值。记录运行时间和性能测试可以帮助你找到最小的缓存容量带来最好的性能。

使用 select 切换协程

14.4 使用 select 切换协程

从不同的并发执行的协程中获取值可以通过关键字 `select` 来完成，它和 `switch` 控制语句非常相似（章节5.3）也被称作通信开关；它的行为像是“你准备好了吗”的轮询机制； `select` 监听进入通道的数据，也可以是用通道发送值的时候。

```
select {
case u:= <- ch1:
        ...
case v:= <- ch2:
        ...
        ...
default: // no value ready to be received
        ...
}
```

`default` 语句是可选的；fallthrough 行为，和普通的 switch 相似，是不允许的。在任何一个 case 中执行 `break` 或者 `return` ，select 就结束了。

`select` 做的就是：选择处理列出的多个通信情况中的一个。

- 如果都阻塞了，会等待直到其中一个可以处理
- 如果多个可以处理，随机选择一个
- 如果没有通道操作可以处理并且写了 `default` 语句，它就会执行： `default` 永远是可运行的（这就是准备好了，可以执行）。

在 `select` 中使用发送操作并且有 `default` 可以确保发送不被阻塞！如果没有 `default` ，select 就会一直阻塞。

`select` 语句实现了一种监听模式，通常用在（无限）循环中；在某种情况下，通过 `break` 语句使循环退出。

在程序 goroutine_select.go 中有 2 个通道 `ch1` 和 `ch2` ，三个协程 `pump1()` 、 `pump2()` 和 `suck()` 。这是一个典型的生产者消费者模式。在无限循环中， `ch1` 和 `ch2` 通过 `pump1()` 和 `pump2()` 填充整数； `suck()` 也是在无限循环中轮询输入的，通过 `select` 语句获取 `ch1` 和 `ch2` 的整数并输出。选择哪一个 case 取决于哪一个通道收到了信息。程序在 main 执行 1 秒后结束。

示例 14.10-goroutine_select.go：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)

    go pump1(ch1)
    go pump2(ch2)
```

```
    go suck(ch1, ch2)

    time.Sleep(1e9)
}

func pump1(ch chan int) {
    for i := 0; ; i++ {
        ch <- i * 2
    }
}

func pump2(ch chan int) {
    for i := 0; ; i++ {
        ch <- i + 5
    }
}

func suck(ch1, ch2 chan int) {
    for {
        select {
        case v := <-ch1:
            fmt.Printf("Received on channel 1: %d\n", v)
        case v := <-ch2:
            fmt.Printf("Received on channel 2: %d\n", v)
        }
    }
}
```

输出：

```
Received on channel 2: 5
Received on channel 2: 6
Received on channel 1: 0
Received on channel 2: 7
Received on channel 2: 8
Received on channel 2: 9
Received on channel 2: 10
Received on channel 1: 2
Received on channel 2: 11
...
Received on channel 2: 47404
Received on channel 1: 94346
Received on channel 1: 94348
```

一秒内的输出非常惊人，如果我们给它计数（goroutine_select2.go），得到了 90000 个左右的数字。

练习：

练习 14.7：

- a）在练习 5.4 的 for_loop.go 中，有一个常见的 for 循环打印数字。在函数 `tel` 中实现一个 for 循环，用协程开始这个函数并在其中给通道发送数字。`main()` 线程从通道中获取并打印。不要使用 `time.Sleep()` 来同步：goroutine_panic.go
- b）也许你的方案有效，可能会引发运行时的 panic：`throw:all goroutines are asleep-deadlock!` 为什么会这样？你如何解决这个问题？goroutine_close.go

- c）解决 a）的另外一种方式：使用一个额外的通道传递给协程，然后在结束的时候随便放点什么进去。`main()` 线程检查是否有数据发送给了这个通道，如果有就停止：goroutine_select.go

练习 14.8：

从示例 6.13 fibonacci.go 的斐波那契程序开始，制定解决方案，使斐波那契周期计算独立到协程中，并可以把结果发送给通道。

结束的时候关闭通道。`main()` 函数读取通道并打印结果：goFibonacci.go

使用练习 6.9 fibonacci2.go 中的算法写一个更短的 gofibonacci2.go

使用 `select` 语句来写，并让通道退出（gofibonacci_select.go）

注意：当给结果计时并和 6.13 对比时，我们发现使用通道通信的性能开销有轻微削减；这个例子中的算法使用协程并非性能最好的选择；但是 gofibonacci3 方案使用了 2 个协程带来了 3 倍的提速。

练习 14.9：

做一个随机位生成器，程序可以提供无限的随机 0 或者 1 的序列：random_bitgen.go

练习 14.10：polar_to_cartesian.go

（这是一种综合练习，使用到第 4、9、11 章和本章的内容。）写一个可交互的控制台程序，要求用户输入二位平面极坐标上的点（半径和角度（度））。计算对应的笛卡尔坐标系的点的 x 和 y 并输出。使用极坐标和笛卡尔坐标的结构体。

使用通道和协程：

- `channel1` 用来接收极坐标
- `channel2` 用来接收笛卡尔坐标

转换过程需要在协程中进行，从 channel1 中读取然后发送到 channel2。实际上做这种计算不提倡使用协程和通道，但是如果运算量很大很耗时，这种方案设计就非常合适了。

练习 14.11： concurrent_pi.go / concurrent_pi2.go

使用以下序列在协程中计算 pi：开启一个协程来计算公式中的每一项并将结果放入通道，`main()` 函数收集并累加结果，打印出 pi 的近似值。

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots.$$

计算执行时间（参见第 6.11 节）

再次声明这只是为了一边练习协程的概念一边找点乐子。

如果你需要的话可使用 `math.pi` 中的 Pi；而且不使用协程会运算的更快。一个急速版本：使用 `GOMAXPROCS`，开启和 `GOMAXPROCS` 同样多个协程。

习惯用法：后台服务模式

服务通常是是用后台协程中的无限循环实现的，在循环中使用 `select` 获取并处理通道中的数据：

```
// Backend goroutine.
func backend() {
    for {
        select {
        case cmd := <-ch1:
            // Handle ...
        case cmd := <-ch2:
            ...
        case cmd := <-chStop:
            // stop server
        }
    }
}
```

在程序的其他地方给通道 `ch1` ， `ch2` 发送数据，比如：通道 `stop` 用来清理结束服务程序。

另一种方式（但是不太灵活）就是（客户端）在 `chRequest` 上提交请求，后台协程循环这个通道，使用 `switch` 根据请求的行为来分别处理：

```
func backend() {
    for req := range chRequest {
        switch req.Subjext() {
            case A1:  // Handle case ...
            case A2:  // Handle case ...
            default:
              // Handle illegal request ..
              // ...
        }
    }
}
```

通道、超时和计时器（Ticker）

14.5 通道、超时和计时器（Ticker）

`time` 包中有一些有趣的功能可以和通道组合使用。

其中就包含了 `time.Ticker` 结构体，这个对象以指定的时间间隔重复的向通道 C 发送时间值：

```
type Ticker struct {
    C <-chan Time // the channel on which the ticks are delivered.
    // contains filtered or unexported fields
    ...
}
```

时间间隔的单位是 ns（纳秒，int64），在工厂函数 `time.NewTicker` 中以 `Duration` 类型的参数传入：`func NewTicker(dur) *Ticker` 。

在协程周期性的执行一些事情（打印状态日志，输出，计算等等）的时候非常有用。

调用 `Stop()` 使计时器停止，在 `defer` 语句中使用。这些都很好的适应 `select` 语句:

```
ticker := time.NewTicker(updateInterval)
defer ticker.Stop()
...
select {
case u:= <-ch1:
    ...
case v:= <-ch2:
    ...
case <-ticker.C:
    logState(status) // call some logging function logState
default: // no value ready to be received
    ...
}
```

`time.Tick()` 函数声明为 `Tick(d Duration) <-chan Time` ，当你想返回一个通道而不必关闭它的时候这个函数非常有用：它以 d 为周期给返回的通道发送时间，d是纳秒数。如果需要像下边的代码一样，限制处理频率（函数 `client.Call()` 是一个 RPC 调用，这里暂不赘述（参见第 15.9 节）：

```
import "time"

rate_per_sec := 10
var dur Duration = 1e9 / rate_per_sec
chRate := time.Tick(dur) // a tick every 1/10th of a second
for req := range requests {
    <- chRate // rate limit our Service.Method RPC calls
    go client.Call("Service.Method", req, ...)
}
```

这样只会按照指定频率处理请求： `chRate` 阻塞了更高的频率。每秒处理的频率可以根据机器负载（和/或）资源的情况而增加或减少。

问题 14.1：扩展上边的代码，思考如何承载周期请求数的暴增（提示：使用带缓冲通道和计时器对象）。

定时器（Timer）结构体看上去和计时器（Ticker）结构体的确很像（构造为 `NewTimer(d Duration)` ），但是它只发送一次时间，在 `Dration d` 之后。

还有 `time.After(d)` 函数，声明如下：

```
func After(d Duration) <-chan Time
```

在 `Duration d` 之后，当前时间被发到返回的通道；所以它和 `NewTimer(d).C` 是等价的；它类似 `Tick()` ，但是 `After()` 只发送一次时间。下边有个很具体的示例，很好的阐明了 `select` 中 `default` 的作用：

示例 14.11：timer_goroutine.go：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(1e8)
    boom := time.After(5e8)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            fmt.Println("    .")
            time.Sleep(5e7)
        }
    }
}
```

输出：

```
    .
    .
tick.
    .
    .
tick.
    .
    .
tick.
    .
    .
tick.
    .
    .
tick.
BOOM!
```

习惯用法：简单超时模式

要从通道 `ch` 中接收数据，但是最多等待1秒。先创建一个信号通道，然后启动一个 `lambda` 协程，协程在给通道发送数据之前是休眠的：

```
timeout := make(chan bool, 1)
go func() {
        time.Sleep(1e9) // one second
        timeout <- true
}()
```

然后使用 `select` 语句接收 `ch` 或者 `timeout` 的数据：如果 `ch` 在 1 秒内没有收到数据，就选择到了 `time` 分支并放弃了 `ch` 的读取。

```
select {
    case <-ch:
        // a read from ch has occured
    case <-timeout:
        // the read from ch has timed out
        break
}
```

第二种形式：取消耗时很长的同步调用

也可以使用 `time.After()` 函数替换 `timeout-channel` 。可以在 `select` 中通过 `time.After()` 发送的超时信号来停止协程的执行。以下代码，在 `timeoutNs` 纳秒后执行 `select` 的 `timeout` 分支后，执行 `client.Call` 的协程也随之结束，不会给通道 `ch` 返回值：

```
ch := make(chan error, 1)
go func() { ch <- client.Call("Service.Method", args, &reply) } ()
select {
case resp := <-ch
    // use resp and reply
case <-time.After(timeoutNs):
    // call timed out
    break
}
```

注意缓冲大小设置为 1 是必要的，可以避免协程死锁以及确保超时的通道可以被垃圾回收。此外，需要注意在有多个 `case` 符合条件时， `select` 对 `case` 的选择是伪随机的，如果上面的代码稍作修改如下，则 `select` 语句可能不会在定时器超时信号到来时立刻选中 `time.After(timeoutNs)` 对应的 `case` ，因此协程可能不会严格按照定时器设置的时间结束。

```
ch := make(chan int, 1)
go func() { for { ch <- 1 } } ()
L:
for {
    select {
    case <-ch:
        // do something
    case <-time.After(timeoutNs):
        // call timed out
        break L
    }
}
```

第三种形式：假设程序从多个复制的数据库同时读取。只需要一个答案，需要接收首先到达的答案， `Query` 函数获取数据库的连接切片并请求。并行请求每一个数据库并返回收到的第一个响应：

```
func Query(conns []Conn, query string) Result {
    ch := make(chan Result, 1)
    for _, conn := range conns {
        go func(c Conn) {
            select {
            case ch <- c.DoQuery(query):
            default:
            }
        }(conn)
    }
    return <- ch
}
```

再次声明，结果通道 `ch` 必须是带缓冲的：以保证第一个发送进来的数据有地方可以存放，确保放入的首个数据总会成功，所以第一个到达的值会被获取而与执行的顺序无关。正在执行的协程可以总是可以使用 `runtime.Goexit()` 来停止。

在应用中缓存数据：

应用程序中用到了来自数据库（或者常见的数据存储）的数据时，经常会把数据缓存到内存中，因为从数据库中获取数据的操作代价很高；如果数据库中的值不发生变化就没有问题。但是如果值有变化，我们需要一个机制来周期性的从数据库重新读取这些值：缓存的值就不可用（过期）了，而且我们也不希望用户看到陈旧的数据。

协程和恢复（recover）

14.6 协程和恢复（recover）

一个用到 `recover` 的程序（参见第 13.3 节）停掉了服务器内部一个失败的协程而不影响其他协程的工作。

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)   // start the goroutine for that work
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Printf("Work failed with %s in %v", err, work)
        }
    }()
    do(work)
}
```

上边的代码，如果 `do(work)` 发生 panic，错误会被记录且协程会退出并释放，而其他协程不受影响。

因为 `recover` 总是返回 `nil`，除非直接在 `defer` 修饰的函数中调用，`defer` 修饰的代码可以调用那些自身可以使用 `panic` 和 `recover` 避免失败的库例程（库函数）。举例，`safelyDo()` 中 `defer` 修饰的函数可能在调用 `recover` 之前就调用了一个 `logging` 函数，`panicking` 状态不会影响 `logging` 代码的运行。因为加入了恢复模式，函数 `do`（以及它调用的任何东西）可以通过调用 `panic` 来摆脱不好的情况。但是恢复是在 `panicking` 的协程内部的：不能被另外一个协程恢复。

新旧模型对比：任务和worker

14.7 新旧模型对比：任务和worker

假设我们需要处理很多任务：一个worker处理一项任务。任务可以被定义为一个结构体（具体的细节在这里并不重要）：

```
type Task struct {
    // some state
}
```

旧模式：使用共享内存进行同步

由各个任务组成的任务池共享内存；为了同步各个worker以及避免资源竞争，我们需要对任务池进行加锁保护：

```
    type Pool struct {
        Mu      sync.Mutex
        Tasks   []*Task
    }
```

sync.Mutex(参见9.3是互斥锁：它用来在代码中保护临界区资源：同一时间只有一个go协程（goroutine）可以进入该临界区。如果出现了同一时间多个go协程都进入了该临界区，则会产生竞争：Pool结构就不能保证被正确更新。在传统的模式中（经典的面向对象的语言中应用得比较多，比如C++,JAVA,C#），worker代码可能这样写：

```
func Worker(pool *Pool) {
    for {
        pool.Mu.Lock()
        // begin critical section:
        task := pool.Tasks[0]         // take the first task
        pool.Tasks = pool.Tasks[1:]  // update the pool of tasks
        // end critical section
        pool.Mu.Unlock()
        process(task)
    }
}
```

这些worker有许多都可以并发执行；他们可以在go协程中启动。一个worker先将pool锁定，从pool获取第一项任务，再解锁和处理任务。加锁保证了同一时间只有一个go协程可以进入到pool中：一项任务有且只能被赋予一个worker。如果不加锁，则工作协程可能会在 `task:=pool.Tasks[0]` 发生切换，导致 `pool.Tasks=pool.Tasks[1:]` 结果异常：一些worker获取不到任务，而一些任务可能被多个worker得到。加锁实现同步的方式在工作协程比较少时可以工作的很好，但是当工作协程数量很大，任务量也很多时，处理效率将会因为频繁的加锁/解锁开销而降低。当工作协程数增加到一个阈值时，程序效率会急剧下降，这就成为了瓶颈。

新模式：使用通道

使用通道进行同步：使用一个通道接受需要处理的任务，一个通道接受处理完成的任务（及其结果）。worker在协程中启动，其数量N应该根据任务数量进行调整。

主线程扮演着Master节点角色，可能写成如下形式：

```
    func main() {
        pending, done := make(chan *Task), make(chan *Task)
        go sendWork(pending)       // put tasks with work on the channel
```

```
        for i := 0; i < N; i++ {   // start N goroutines to do work
            go Worker(pending, done)
        }
        consumeWork(done)          // continue with the processed tasks
    }
```

worker的逻辑比较简单：从pending通道拿任务，处理后将其放到done通道中：

```
    func Worker(in, out chan *Task) {
        for {
            t := <-in
            process(t)
            out <- t
        }
    }
```

这里并不使用锁：从通道得到新任务的过程没有任何竞争。随着任务数量增加，worker数量也应该相应增加，同时性能并不会像第一种方式那样下降明显。在pending通道中存在一份任务的拷贝，第一个worker从pending通道中获得第一个任务并进行处理，这里并不存在竞争（对一个通道读数据和写数据的整个过程是原子性的：参见14.2.2)。某一个任务会在哪一个worker中被执行是不可知的，反过来也是。worker数量的增多也会增加通信的开销，这会对性能有轻微的影响。

从这个简单的例子中可能很难看出第二种模式的优势，但含有复杂锁运用的程序不仅在编写上显得困难，也不容易编写正确，使用第二种模式的话，就无需考虑这么复杂的东西了。

因此，第二种模式对比第一种模式而言，不仅性能是一个主要优势，而且还有个更大的优势：代码显得更清晰、更优雅。一个更符合go语言习惯的worker写法：

IDIOM: Use an in- and out-channel instead of locking

```
    func Worker(in, out chan *Task) {
        for {
            t := <-in
            process(t)
            out <- t
        }
    }
```

对于任何可以建模为Master-Worker范例的问题，一个类似于worker使用通道进行通信和交互、Master进行整体协调的方案都能完美解决。如果系统部署在多台机器上，各个机器上执行Worker协程，Master和Worker之间使用netchan或者RPC进行通信（参见15章）。

怎么选择是该使用锁还是通道？

通道是一个较新的概念，本节我们着重强调了在go协程里通道的使用，但这并不意味着经典的锁方法就不能使用。go语言让你可以根据实际问题进行选择：创建一个优雅、简单、可读性强、在大多数场景性能表现都能很好的方案。如果你的问题适合使用锁，也不要忌讳使用它。go语言注重实用，什么方式最能解决你的问题就用什么方式，而不是强迫你使用一种编码风格。下面列出一个普遍的经验法则：

- 使用锁的情景：
    - 访问共享数据结构中的缓存信息
    - 保存应用程序上下文和状态信息数据
- 使用通道的情景：
    - 与异步操作的结果进行交互

- 分发任务
- 传递数据所有权

当你发现你的锁使用规则变得很复杂时，可以反省使用通道会不会使问题变得简单些。

惰性生成器的实现

14.8 惰性生成器的实现

生成器是指当被调用时返回一个序列中下一个值的函数，例如：

```
    generateInteger() => 0
    generateInteger() => 1
    generateInteger() => 2
    ....
```

生成器每次返回的是序列中下一个值而非整个序列；这种特性也称之为惰性求值：只在你需要时进行求值，同时保留相关变量资源（内存和cpu）：这是一项在需要时对表达式进行求值的技术。例如，生成一个无限数量的偶数序列：要产生这样一个序列并且在一个一个的使用可能会很困难，而且内存会溢出！但是一个含有通道和go协程的函数能轻易实现这个需求。

在14.12的例子中，我们实现了一个使用 int 型通道来实现的生成器。通道被命名为 `yield` 和 `resume` ，这些词经常在协程代码中使用。

示例 14.12 lazy_evaluation.go：

```
package main

import (
    "fmt"
)

var resume chan int

func integers() chan int {
    yield := make(chan int)
    count := 0
    go func() {
        for {
            yield <- count
            count++
        }
    }()
    return yield
}

func generateInteger() int {
    return <-resume
}

func main() {
    resume = integers()
    fmt.Println(generateInteger())  //=> 0
    fmt.Println(generateInteger())  //=> 1
    fmt.Println(generateInteger())  //=> 2
}
```

有一个细微的区别是从通道读取的值可能会是稍早前产生的，并不是在程序被调用时生成的。如果确实需要这样的行为，就得实现一个请求响应机制。当生成器生成数据的过程是计算密集型且各个结果的顺序并不重要时，那么就可以将生成器放入到go协程实现并行化。但是得小心，使用大量的go协程的开销可能会超过带来的性能增益。

这些原则可以概括为：通过巧妙地使用空接口、闭包和高阶函数，我们能实现一个通用的惰性生产器的工厂函数 `BuildLazyEvaluator` （这个应该放在一个工具包中实现）。工厂函数需要一个函数和一个初始状态作为输入参数，返回一个无参、返回值是生成序列的函数。传入的函数需要计算出下一个返回值以及下一个状态参数。在工厂函数中，创建一个通道和无限循环的go协程。返回值被放到了该通道中，返回函数稍后被调用时从该通道中取得该返回值。每当取得一个值时，下一个值即被计算。在下面的例子中，定义了一个 `evenFunc` 函数，其是一个惰性生成函数：在main函数中，我们创建了前10个偶数，每个都是通过调用 `even()` 函数取得下一个值的。为此，我们需要在 `BuildLazyIntEvaluator` 函数中具体化我们的生成函数，然后我们能够基于此做出定义。

示例 14.13 general_lazy_evalution1.go：

```
package main

import (
    "fmt"
)

type Any interface{}
type EvalFunc func(Any) (Any, Any)

func main() {
    evenFunc := func(state Any) (Any, Any) {
        os := state.(int)
        ns := os + 2
        return os, ns
    }

    even := BuildLazyIntEvaluator(evenFunc, 0)

    for i := 0; i < 10; i++ {
        fmt.Printf("%vth even: %v\n", i, even())
    }
}

func BuildLazyEvaluator(evalFunc EvalFunc, initState Any) func() Any {
    retValChan := make(chan Any)
    loopFunc := func() {
        var actState Any = initState
        var retVal Any
        for {
            retVal, actState = evalFunc(actState)
            retValChan <- retVal
        }
    }
    retFunc := func() Any {
        return <- retValChan
    }
    go loopFunc()
    return retFunc
}

func BuildLazyIntEvaluator(evalFunc EvalFunc, initState Any) func() int {
    ef := BuildLazyEvaluator(evalFunc, initState)
    return func() int {
        return ef().(int)
    }
}
```

输出：

```
0th even: 0
1th even: 2
2th even: 4
3th even: 6
4th even: 8
5th even: 10
6th even: 12
7th even: 14
8th even: 16
9th even: 18
```

练习14.12：general_lazy_evaluation2.go
通过使用14.12中工厂函数生成前10个斐波那契数

提示：因为斐波那契数增长很迅速，使用 `uint64` 类型。
注：这种计算通常被定义为递归函数，但是在没有尾递归的语言中，例如go语言，这可能会导致栈溢出，但随着go语言中堆栈可扩展的优化，这个问题就不那么严重。这里的诀窍是使用了惰性求值。gccgo编译器在某些情况下会实现尾递归。

实现 Futures 模式

14.9 实现 Futures 模式

所谓Futures就是指：有时候在你使用某一个值之前需要先对其进行计算。这种情况下，你就可以在另一个处理器上进行该值的计算，到使用时，该值就已经计算完毕了。

Futures模式通过闭包和通道可以很容易实现，类似于生成器，不同地方在于Futures需要返回一个值。

参考条目文献给出了一个很精彩的例子：假设我们有一个矩阵类型，我们需要计算两个矩阵A和B乘积的逆，首先我们通过函数 `Inverse(M)` 分别对其进行求逆运算，再将结果相乘。如下函数 `InverseProduct()` 实现了如上过程：

```
func InverseProduct(a Matrix, b Matrix) {
    a_inv := Inverse(a)
    b_inv := Inverse(b)
    return Product(a_inv, b_inv)
}
```

在这个例子中，a和b的求逆矩阵需要先被计算。那么为什么在计算b的逆矩阵时，需要等待a的逆计算完成呢？显然不必要，这两个求逆运算其实可以并行执行的。换句话说，调用 `Product` 函数只需要等到 `a_inv` 和 `b_inv` 的计算完成。如下代码实现了并行计算方式：

```
func InverseProduct(a Matrix, b Matrix) {
    a_inv_future := InverseFuture(a)   // start as a goroutine
    b_inv_future := InverseFuture(b)   // start as a goroutine
    a_inv := <-a_inv_future
    b_inv := <-b_inv_future
    return Product(a_inv, b_inv)
}
```

`InverseFuture` 函数以 `goroutine` 的形式起了一个闭包，该闭包会将矩阵求逆结果放入到future通道中：

```
func InverseFuture(a Matrix) chan Matrix {
    future := make(chan Matrix)
    go func() {
        future <- Inverse(a)
    }()
    return future
}
```

当开发一个计算密集型库时，使用Futures模式设计API接口是很有意义的。在你的包使用Futures模式，且能保持友好的API接口。此外，Futures可以通过一个异步的API暴露出来。这样你可以以最小的成本将包中的并行计算移到用户代码中。（参见参考文件18：http://www.golangpatterns.info/concurrency/futures）

复用

14.10 复用

14.10.1 典型的客户端/服务器（C/S）模式

客户端-服务器应用正是 goroutines 和 channels 的亮点所在。

客户端(Client)可以是运行在任意设备上的任意程序，它会按需发送请求(request)至服务器。服务器(Server)接收到这个请求后开始相应的工作，然后再将响应(response)返回给客户端。典型情况下一般是多个客户端（即多个请求）对应一个（或少量）服务器。例如我们日常使用的浏览器客户端，其功能就是向服务器请求网页。而Web服务器则会向浏览器响应网页数据。

使用Go的服务器通常会在协程中执行向客户端的响应，故而会对每一个客户端请求启动一个协程。一个常用的操作方法是客户端请求自身中包含一个通道，而服务器则向这个通道发送响应。

例如下面这个 `Request` 结构，其中内嵌了一个 `replyc` 通道。

```
type Request struct {
    a, b       int
    replyc     chan int // reply channel inside the Request
}
```

或者更通俗的：

```
type Reply struct{...}
type Request struct{
    arg1, arg2, arg3 some_type
    replyc chan *Reply
}
```

接下来先使用简单的形式,服务器会为每一个请求启动一个协程并在其中执行 `run()` 函数，此举会将类型为 `binOp` 的 `op` 操作返回的int值发送到 `replyc` 通道。

```
type binOp func(a, b int) int

func run(op binOp, req *Request) {
    req.replyc <- op(req.a, req.b)
}
```

`server` 协程会无限循环以从 `chan *Request` 接收请求，并且为了避免被长时间操作所堵塞，它将为每一个请求启动一个协程来做具体的工作：

```
func server(op binOp, service chan *Request) {
    for {
        req := <-service; // requests arrive here
        // start goroutine for request:
        go run(op, req);  // don't wait for op to complete
    }
}
```

`server` 本身则是以协程的方式在 `startServer` 函数中启动：

```
func startServer(op binOp) chan *Request {
    reqChan := make(chan *Request);
    go server(op, reqChan);
    return reqChan;
}
```

`startServer` 则会在 `main` 协程中被调用。

在以下测试例子中，100个请求会被发送到服务器，只有它们全部被送达后我们才会按相反的顺序检查响应：

```
func main() {
    adder := startServer(func(a, b int) int { return a + b })
    const N = 100
    var reqs [N]Request
    for i := 0; i < N; i++ {
        req := &reqs[i]
        req.a = i
        req.b = i + N
        req.replyc = make(chan int)
        adder <- req  // adder is a channel of requests
    }
    // checks:
    for i := N - 1; i >= 0; i-- {
        // doesn't matter what order
        if <-reqs[i].replyc != N+2*i {
            fmt.Println("fail at", i)
        } else {
            fmt.Println("Request ", i, "is ok!")
        }
    }
    fmt.Println("done")
}
```

这些代码可以在multiplex_server.go找到

输出：

```
Request 99 is ok!
Request 98 is ok!
...
Request 1 is ok!
Request 0 is ok!
done
```

这个程序仅启动了100个协程。然而即使执行100,000个协程我们也能在数秒内看到它完成。这说明了Go的协程是如何的轻量：如果我们启动相同数量的真实的线程，程序早就崩溃了。

示例： 14.14-multiplex_server.go

```
package main

import "fmt"

type Request struct {
    a, b   int
    replyc chan int // reply channel inside the Request
}
```

```
type binOp func(a, b int) int

func run(op binOp, req *Request) {
    req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *Request) {
    for {
        req := <-service // requests arrive here
        // start goroutine for request:
        go run(op, req) // don't wait for op
    }
}

func startServer(op binOp) chan *Request {
    reqChan := make(chan *Request)
    go server(op, reqChan)
    return reqChan
}

func main() {
    adder := startServer(func(a, b int) int { return a + b })
    const N = 100
    var reqs [N]Request
    for i := 0; i < N; i++ {
        req := &reqs[i]
        req.a = i
        req.b = i + N
        req.replyc = make(chan int)
        adder <- req
    }
    // checks:
    for i := N - 1; i >= 0; i-- { // doesn't matter what order
        if <-reqs[i].replyc != N+2*i {
            fmt.Println("fail at", i)
        } else {
            fmt.Println("Request ", i, " is ok!")
        }
    }
    fmt.Println("done")
}
```

14.10.2 卸载（Teardown）：通过信号通道关闭服务器

在上一个版本中 `server` 在 `main` 函数返回后并没有完全关闭，而被强制结束了。为了改进这一点，我们可以提供一个退出通道给 `server`：

```
func startServer(op binOp) (service chan *Request, quit chan bool) {
    service = make(chan *Request)
    quit = make(chan bool)
    go server(op, service, quit)
    return service, quit
}
```

`server` 函数现在则使用 `select` 在 `service` 通道和 `quit` 通道之间做出选择：

```
func server(op binOp, service chan *request, quit chan bool) {
    for {
        select {
            case req := <-service:
                go run(op, req)
            case <-quit:
                return
        }
    }
}
```

当 `quit` 通道接收到一个 `true` 值时， `server` 就会返回并结束。

在 `main` 函数中我们做出如下更改：

```
adder, quit := startServer(func(a, b int) int { return a + b })
```

在 `main` 函数的结尾处我们放入这一行： `quit <- true`

完整的代码在 multiplex_server2.go,输出和上一个版本是一样的。

示例： 14.15-multiplex_server2.go

```
package main

import "fmt"

type Request struct {
    a, b   int
    replyc chan int // reply channel inside the Request
}

type binOp func(a, b int) int

func run(op binOp, req *Request) {
    req.replyc <- op(req.a, req.b)
}

func server(op binOp, service chan *Request, quit chan bool) {
    for {
        select {
        case req := <-service:
            go run(op, req)
        case <-quit:
            return
        }
    }
}

func startServer(op binOp) (service chan *Request, quit chan bool) {
    service = make(chan *Request)
    quit = make(chan bool)
    go server(op, service, quit)
    return service, quit
}

func main() {
    adder, quit := startServer(func(a, b int) int { return a + b })
```

```
    const N = 100
    var reqs [N]Request
    for i := 0; i < N; i++ {
        req := &reqs[i]
        req.a = i
        req.b = i + N
        req.replyc = make(chan int)
        adder <- req
    }
    // checks:
    for i := N - 1; i >= 0; i-- { // doesn't matter what order
        if <-reqs[i].replyc != N+2*i {
            fmt.Println("fail at", i)
        } else {
            fmt.Println("Request ", i, " is ok!")
        }
    }
    quit <- true
    fmt.Println("done")
}
```

练习 14.13 multiplex_server3.go:使用之前的例子，编写一个在 `Request` 结构上带有 `String()` 方法的版本，它能决定服务器如何输出；并使用以下两个请求来测试这个程序：

```
    req1 := &Request{3, 4, make(chan int)}
    req2 := &Request{150, 250, make(chan int)}
    ...
    // show the output
    fmt.Println(req1,"\n",req2)
```

限制同时处理的请求数

14.11 限制同时处理的请求数

使用带缓冲区的通道很容易实现这一点（参见 14.2.5），其缓冲区容量就是同时处理请求的最大数量。程序max_tasks.go虽然没有做什么有用的事但是却包含了这个技巧：超过 `MAXREQS` 的请求将不会被同时处理，因为当信号通道表示缓冲区已满时 `handle` 函数会阻塞且不再处理其他请求，直到某个请求从 `sem` 中被移除。 `sem` 就像一个信号量，这一专业术语用于在程序中表示特定条件的标志变量。

示例：14.16-max_tasks.go

```
package main

const MAXREQS = 50

var sem = make(chan int, MAXREQS)

type Request struct {
    a, b   int
    replyc chan int
}

func process(r *Request) {
    // do something
}

func handle(r *Request) {
    sem <- 1 // doesn't matter what we put in it
    process(r)
    <-sem // one empty place in the buffer: the next request can start
}

func server(service chan *Request) {
    for {
        request := <-service
        go handle(request)
    }
}

func main() {
    service := make(chan *Request)
    go server(service)
}
```

通过这种方式，应用程序可以通过使用缓冲通道（通道被用作信号量）使协程同步其对该资源的使用，从而充分利用有限的资源（如内存）。

链式协程

14.12 链式协程

下面的演示程序 chaining.go 再次展示了启动巨量的Go协程是多么容易。这些协程已全部在 main 函数中的 for 循环里启动。当循环完成之后，一个0被写入到最右边的通道里，于是100,000个协程开始执行，接着 `1000000` 这个结果会在1.5秒之内被打印出来。

这个程序同时也展示了如何通过 `flag.Int` 来解析命令行中的参数以指定协程数量，例如： `chaining -n=7000` 会生成7000个协程。

示例 14.17 - chaining.go

```
package main

import (
    "flag"
    "fmt"
)

var ngoroutine = flag.Int("n", 100000, "how many goroutines")

func f(left, right chan int) { left <- 1 + <-right }

func main() {
    flag.Parse()
    leftmost := make(chan int)
    var left, right chan int = nil, leftmost
    for i := 0; i < *ngoroutine; i++ {
        left, right = right, make(chan int)
        go f(left, right)
    }
    right <- 0      // bang!
    x := <-leftmost // wait for completion
    fmt.Println(x)  // 100000, ongeveer 1,5 s
}
```

译者注：原本认为leftmost的结果为1，认为只在最初做了一次赋值，实际结果为100000（无缓存信道具有同步阻塞的特性）

1.主线程的right <- 0，right不是最初循环的那个right，而是最终循环的right

2.for循环中最初的go f(left, right)因为没有发送者一直处于等待状态

3.当主线程的right <- 0执行时，类似于递归函数在最内层产生返回值一般

在多核心上并行计算

14.13 在多核心上并行计算

假设我们有 `NCPU` 个CPU核心： `const NCPU = 4 //对应一个四核处理器` 然后我们想把计算量分成 `NCPU` 个部分，每一个部分都和其他部分并行运行。

这可以通过以下代码所示的方式完成（我们且省略具体参数）

```
func DoAll(){
    sem := make(chan int, NCPU) // Buffering optional but sensible
    for i := 0; i < NCPU; i++ {
        go DoPart(sem)
    }
    // Drain the channel sem, waiting for NCPU tasks to complete
    for i := 0; i < NCPU; i++ {
        <-sem // wait for one task to complete
    }
    // All done.
}

func DoPart(sem chan int) {
    // do the part of the computation
    sem <-1 // signal that this piece is done
}

func main() {
    runtime.GOMAXPROCS(NCPU) // runtime.GOMAXPROCS = NCPU
    DoAll()
}
```

- `DoAll()` 函数创建了一个 `sem` 通道，每个并行计算都将在对其发送完成信号；在一个 for 循环中 `NCPU` 个协程被启动了，每个协程会承担 `1/NCPU` 的工作量。每一个 `DoPart()` 协程都会向 `sem` 通道发送完成信号。

- `DoAll()` 会在 for 循环中等待 `NCPU` 个协程完成： `sem` 通道就像一个信号量，这份代码展示了一个经典的信号量模式。（参见 14.2.7）

在以上运行模型中，您还需将 `GOMAXPROCS` 设置为 `NCPU` （参见 14.1.3）。

并行化大量数据的计算

14.14 并行化大量数据的计算

假设我们需要处理一些数量巨大且互不相关的数据项，它们从一个 `in` 通道被传递进来，当我们处理完以后又要将它们放入另一个 `out` 通道，就像一个工厂流水线一样。处理每个数据项也可能包含许多步骤：Preprocess（预处理） / StepA（步骤A） / StepB（步骤B） / ... / PostProcess（后处理）

一个典型的用于解决按顺序执行每个步骤的顺序流水线算法可以写成下面这样：

```
func SerialProcessData(in <-chan *Data, out chan<- *Data) {
    for data := range in {
        tmpA := PreprocessData(data)
        tmpB := ProcessStepA(tmpA)
        tmpC := ProcessStepB(tmpB)
        out <- PostProcessData(tmpC)
    }
}
```

一次只执行一个步骤，并且按顺序处理每个项目：在第1个项目没有被 `PostProcess` 并放入 `out` 通道之前绝不会处理第2个项目。

如果你仔细想想，你很快就会发现这将会造成巨大的时间浪费。

一个更高效的计算方式是让每一个处理步骤作为一个协程独立工作。每一个步骤从上一步的输出通道中获得输入数据。这种方式仅有极少数时间会被浪费，而大部分时间所有的步骤都在一直执行中：

```
func ParallelProcessData (in <-chan *Data, out chan<- *Data) {
    // make channels:
    preOut := make(chan *Data, 100)
    stepAOut := make(chan *Data, 100)
    stepBOut := make(chan *Data, 100)
    stepCOut := make(chan *Data, 100)
    // start parallel computations:
    go PreprocessData(in, preOut)
    go ProcessStepA(preOut,StepAOut)
    go ProcessStepB(StepAOut,StepBOut)
    go ProcessStepC(StepBOut,StepCOut)
    go PostProcessData(StepCOut,out)
}
```

通道的缓冲区大小可以用来进一步优化整个过程。

漏桶算法

14.15 漏桶算法

（译者注：翻译遵照原文，但是对于完全没听过这个算法的人来说比较晦涩，请配合代码片段理解）

考虑以下的客户端-服务器结构：客户端协程执行一个无限循环从某个源头（也许是网络）接收数据；数据读取到 `Buffer` 类型的缓冲区。为了避免分配过多的缓冲区以及释放缓冲区，它保留了一份空闲缓冲区列表，并且使用一个缓冲通道来表示这个列表： `var freeList = make(chan *Buffer,100)`

这个可重用的缓冲区队列（freeList）与服务器是共享的。 当接收数据时，客户端尝试从 `freeList` 获取缓冲区；但如果此时通道为空，则会分配新的缓冲区。 一旦消息被加载后，它将被发送到服务器上的 `serverChan` 通道：

```
var serverChan = make(chan *Buffer)
```

以下是客户端的算法代码：

```
func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not
        select {
            case b = <-freeList:
                // Got one; nothing more to do
            default:
                // None free, so allocate a new one
                b = new(Buffer)
        }
        loadInto(b)         // Read next message from the network
        serverChan <- b     // Send to server

    }
}
```

服务器的循环则接收每一条来自客户端的消息并处理它，之后尝试将缓冲返回给共享的空闲缓冲区：

```
func server() {
    for {
        b := <-serverChan       // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
            case freeList <- b:
                // Reuse buffer if free slot on freeList; nothing more to do
            default:
                // Free list full, just carry on: the buffer is 'dropped'
        }
    }
}
```

但是这种方法在 `freeList` 通道已满的时候是行不通的，因为无法放入空闲 `freeList` 通道的缓冲区会被“丢到地上”由垃圾收集器回收（故名：漏桶算法）

对Go协程进行基准测试

14.16 对Go协程进行基准测试

在 13.7 节 我们提到了在Go语言中对你的函数进行基准测试。在此我们将其应用到一个用协程向通道写入整数再读出的实例中。这个函数将通过 `testing.Benchmark` 调用 `N` 次（例如： `N = 1,000,000` ）， `BenchMarkResult` 有一个 `String()` 方法来输出其结果。 `N` 的值将由 `gotest` 来判断并取得一个足够大的数字，以获得合理的基准测试结果。当然同样的基准测试方法也适用于普通函数。

如果你想排除指定部分的代码或者更具体的指定要测试的部分，可以使用 `testing.B.startTimer()` 和 `testing.B.stopTimer()` 来开始或结束计时器。基准测试只有在所有的测试通过后才能运行！

示例：14.18-benchmark_channels.go

```
package main

import (
    "fmt"
    "testing"
)

func main() {
    fmt.Println(" sync", testing.Benchmark(BenchmarkChannelSync).String())
    fmt.Println("buffered", testing.Benchmark(BenchmarkChannelBuffered).String())
}

func BenchmarkChannelSync(b *testing.B) {
    ch := make(chan int)
    go func() {
        for i := 0; i < b.N; i++ {
            ch <- i
        }
        close(ch)
    }()
    for range ch {
    }
}

func BenchmarkChannelBuffered(b *testing.B) {
    ch := make(chan int, 128)
    go func() {
        for i := 0; i < b.N; i++ {
            ch <- i
        }
        close(ch)
    }()
    for range ch {
    }
}
```

输出：

```
  Output:Windows:  N       Time 1 op   Operations per sec
  sync      1000000  2443 ns/op  -->  409 332 / s
```

```
buffered   1000000   4850 ns/op   -->   810 477 / s
Linux:
```

使用通道并发访问对象

14.17 使用通道并发访问对象

为了保护对象被并发访问修改，我们可以使用协程在后台顺序执行匿名函数来替代使用同步互斥锁。在下面的程序中我们有一个类型 `Person` 其中包含一个字段 `chF` ，这是一个用于存放匿名函数的通道。

这个结构在构造函数 `NewPerson()` 中初始化的同时会启动一个后台协程 `backend()` 。 `backend()` 方法会在一个无限循环中执行 `chF` 中放置的所有函数，有效的将它们序列化从而提供了安全的并发访问。更改和读取 `salary` 的方法会通过将一个匿名函数写入 `chF` 通道中，然后让 `backend()` 按顺序执行以达到其目的。需注意的是 `Salary` 方法创建的闭包函数是如何将 `fChan` 通道包含在其中的。

当然，这是一个简化的例子，它不应该被用在这种案例下。但是它却向我们展示了在更复杂的场景中该如何解决这种问题。

示例：14.19-conc_access.go

```
package main

import (
    "fmt"
    "strconv"
)

type Person struct {
    Name   string
    salary float64
    chF    chan func()
}

func NewPerson(name string, salary float64) *Person {
    p := &Person{name, salary, make(chan func())}
    go p.backend()
    return p
}

func (p *Person) backend() {
    for f := range p.chF {
        f()
    }
}

// Set salary.
func (p *Person) SetSalary(sal float64) {
    p.chF <- func() { p.salary = sal }
}

// Retrieve salary.
func (p *Person) Salary() float64 {
    fChan := make(chan float64)
    p.chF <- func() { fChan <- p.salary }
    return <-fChan
}

func (p *Person) String() string {
    return "Person - name is: " + p.Name + " - salary is: " + strconv.FormatFloat(p.Salary(), 'f', 2, 64)
}
```

```
func main() {
    bs := NewPerson("Smith Bill", 2500.5)
    fmt.Println(bs)
    bs.SetSalary(4000.25)
    fmt.Println("Salary changed:")
    fmt.Println(bs)
}
```

输出：

```
Person - name is: Smith Bill - salary is: 2500.50
Salary changed:
Person - name is: Smith Bill - salary is: 4000.25
```

网络、模板与网页应用

15.0 网络、模板与网页应用

Go 在编写 web 应用方面非常得力。因为目前它还没有GUI（Graphic User Interface 即图形化用户界面）的框架，通过文本或者模板展现的 html 页面是目前 Go 编写界面应用程序的唯一方式。（**译者注：实际上在翻译的时候，已经有了一些不太成熟的GUI库例如：go ui。**）

tcp 服务器

15.1 tcp 服务器

这部分我们将使用 TCP 协议和在 14 章讲到的协程范式编写一个简单的客户端-服务器应用，一个（web）服务器应用需要响应众多客户端的并发请求：Go 会为每一个客户端产生一个协程用来处理请求。我们需要使用 net 包中网络通信的功能。它包含了处理 TCP/IP 以及 UDP 协议、域名解析等方法。

服务器端代码是一个单独的文件：

示例 15.1 server.go

```
package main

import (
    "fmt"
    "net"
)

func main() {
    fmt.Println("Starting the server ...")
    // 创建 listener
    listener, err := net.Listen("tcp", "localhost:50000")
    if err != nil {
        fmt.Println("Error listening", err.Error())
        return //终止程序
    }
    // 监听并接受来自客户端的连接
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Error accepting", err.Error())
            return // 终止程序
        }
        go doServerStuff(conn)
    }
}

func doServerStuff(conn net.Conn) {
    for {
        buf := make([]byte, 512)
        len, err := conn.Read(buf)
        if err != nil {
            fmt.Println("Error reading", err.Error())
            return //终止程序
        }
        fmt.Printf("Received data: %v", string(buf[:len]))
    }
}
```

在 `main()` 中创建了一个 `net.Listener` 类型的变量 `listener`，他实现了服务器的基本功能：用来监听和接收来自客户端的请求（在 localhost 即 IP 地址为 127.0.0.1 端口为 50000 基于TCP协议）。`Listen()` 函数可以返回一个 `error` 类型的错误变量。用一个无限 for 循环的 `listener.Accept()` 来等待客户端的请求。客户端的请求将产生一个 `net.Conn` 类型的连接变量。然后一个独立的协程使用这个连接执行 `doServerStuff()`，开始使用一个 512 字节的缓冲 `data` 来读取客户端发送来的数据，并且把它们打印到服务器的终端，`len` 获取客户端发送的数

据字节数；当客户端发送的所有数据都被读取完成时，协程就结束了。这段程序会为每一个客户端连接创建一个独立的协程。必须先运行服务器代码，再运行客户端代码。

客户端代码写在另一个文件 client.go 中：

示例 15.2 client.go

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)

func main() {
    //打开连接:
    conn, err := net.Dial("tcp", "localhost:50000")
    if err != nil {
        //由于目标计算机积极拒绝而无法创建连接
        fmt.Println("Error dialing", err.Error())
        return // 终止程序
    }

    inputReader := bufio.NewReader(os.Stdin)
    fmt.Println("First, what is your name?")
    clientName, _ := inputReader.ReadString('\n')
    // fmt.Printf("CLIENTNAME %s", clientName)
    trimmedClient := strings.Trim(clientName, "\r\n") // Windows 平台下用 "\r\n", Linux平台下使用 "\n"
    // 给服务器发送信息直到程序退出:
    for {
        fmt.Println("What to send to the server? Type Q to quit.")
        input, _ := inputReader.ReadString('\n')
        trimmedInput := strings.Trim(input, "\r\n")
        // fmt.Printf("input:--%s--", input)
        // fmt.Printf("trimmedInput:--%s--", trimmedInput)
        if trimmedInput == "Q" {
            return
        }
        _, err = conn.Write([]byte(trimmedClient + " says: " + trimmedInput))
    }
}
```

客户端通过 `net.Dial` 创建了一个和服务器之间的连接。

它通过无限循环从 `os.Stdin` 接收来自键盘的输入，直到输入了“Q”。注意裁剪 `\r` 和 `\n` 字符（仅 Windows 平台需要）。裁剪后的输入被 `connection` 的 `Write` 方法发送到服务器。

当然，服务器必须先启动好，如果服务器并未开始监听，客户端是无法成功连接的。

如果在服务器没有开始监听的情况下运行客户端程序，客户端会停止并打印出以下错误信息： `对tcp 127.0.0.1:50000发起连接时产生错误：由于目标计算机的积极拒绝而无法创建连接` 。

打开命令提示符并转到服务器和客户端可执行程序所在的目录，Windows 系统下输入server.exe（或者只输入server），Linux系统下输入./server。

接下来控制台出现以下信息： `Starting the server ...`

在 Windows 系统中，我们可以通过 CTRL/C 停止程序。

然后开启 2 个或者 3 个独立的控制台窗口，分别输入 client 回车启动客户端程序

以下是服务器的输出：

```
Starting the Server ...
Received data: IVO says: Hi Server, what's up ?
Received data: CHRIS says: Are you busy server ?
Received data: MARC says: Don't forget our appointment tomorrow !
```

当客户端输入 Q 并结束程序时，服务器会输出以下信息：

```
Error reading WSARecv tcp 127.0.0.1:50000: The specified network name is no longer available.
```

在网络编程中 `net.Dial` 函数是非常重要的，一旦你连接到远程系统，函数就会返回一个 `Conn` 类型的接口，我们可以用它发送和接收数据。 `Dial` 函数简洁地抽象了网络层和传输层。所以不管是 IPv4 还是 IPv6，TCP 或者 UDP 都可以使用这个公用接口。

以下示例先使用 TCP 协议连接远程 80 端口，然后使用 UDP 协议连接，最后使用 TCP 协议连接 IPv6 地址：

示例 15.3 dial.go

```
// make a connection with www.example.org:
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    conn, err := net.Dial("tcp", "192.0.32.10:80") // tcp ipv4
    checkConnection(conn, err)
    conn, err = net.Dial("udp", "192.0.32.10:80") // udp
    checkConnection(conn, err)
    conn, err = net.Dial("tcp", "[2620:0:2d0:200::10]:80") // tcp ipv6
    checkConnection(conn, err)
}
func checkConnection(conn net.Conn, err error) {
    if err != nil {
        fmt.Printf("error %v connecting!", err)
        os.Exit(1)
    }
    fmt.Printf("Connection is made with %v\n", conn)
}
```

下边也是一个使用 net 包从 socket 中打开，写入，读取数据的例子：

示例 15.4 socket.go

```
package main

import (
    "fmt"
    "io"
```

```
    "net"
)

func main() {
    var (
        host          = "www.apache.org"
        port          = "80"
        remote        = host + ":" + port
        msg    string = "GET / \n"
        data          = make([]uint8, 4096)
        read          = true
        count         = 0
    )
    // 创建一个socket
    con, err := net.Dial("tcp", remote)
    // 发送我们的消息，一个http GET请求
    io.WriteString(con, msg)
    // 读取服务器的响应
    for read {
        count, err = con.Read(data)
        read = (err == nil)
        fmt.Printf(string(data[0:count]))
    }
    con.Close()
}
```

练习 15.1

编写新版本的客户端和服务器（client1.go / server1.go）：

- 增加一个检查错误的函数 `checkError(error)`；讨论如下方案的利弊：为什么这个重构可能并没有那么理想？看看在示例15.14 中它是如何被解决的
- 使客户端可以通过发送一条命令 SH 来关闭服务器
- 让服务器可以保存已经连接的客户端列表（他们的名字）；当客户端发送 WHO 指令的时候，服务器将显示如下列表：

```
This is the client list: 1:active, 0=inactive
User IVO is 1
User MARC is 1
User CHRIS is 1
```

  注意：当服务器运行的时候，你无法编译/连接同一个目录下的源码来产生一个新的版本，因为 `server.exe` 正在被操作系统使用而无法被替换成新的版本。

下边这个版本的 simple_tcp_server.go 从很多方面优化了第一个tcp服务器的示例 server.go 并且拥有更好的结构，它只用了 80 行代码！

示例 15.5 simple_tcp_server.go：

```
// Simple multi-thread/multi-core TCP server.
package main

import (
    "flag"
    "fmt"
    "net"
    "os"
)
```

```
const maxRead = 25

func main() {
    flag.Parse()
    if flag.NArg() != 2 {
        panic("usage: host port")
    }
    hostAndPort := fmt.Sprintf("%s:%s", flag.Arg(0), flag.Arg(1))
    listener := initServer(hostAndPort)
    for {
        conn, err := listener.Accept()
        checkError(err, "Accept: ")
        go connectionHandler(conn)
    }
}

func initServer(hostAndPort string) *net.TCPListener {
    serverAddr, err := net.ResolveTCPAddr("tcp", hostAndPort)
    checkError(err, "Resolving address:port failed: '"+hostAndPort+"'")
    listener, err := net.ListenTCP("tcp", serverAddr)
    checkError(err, "ListenTCP: ")
    println("Listening to: ", listener.Addr().String())
    return listener
}

func connectionHandler(conn net.Conn) {
    connFrom := conn.RemoteAddr().String()
    println("Connection from: ", connFrom)
    sayHello(conn)
    for {
        var ibuf []byte = make([]byte, maxRead+1)
        length, err := conn.Read(ibuf[0:maxRead])
        ibuf[maxRead] = 0 // to prevent overflow
        switch err {
        case nil:
            handleMsg(length, err, ibuf)
        case os.EAGAIN: // try again
            continue
        default:
            goto DISCONNECT
        }
    }
DISCONNECT:
    err := conn.Close()
    println("Closed connection: ", connFrom)
    checkError(err, "Close: ")
}

func sayHello(to net.Conn) {
    obuf := []byte{'L', 'e', 't', '\'', 's', ' ', 'G', 'O', '!', '\n'}
    wrote, err := to.Write(obuf)
    checkError(err, "Write: wrote "+string(wrote)+" bytes.")
}

func handleMsg(length int, err error, msg []byte) {
    if length > 0 {
        print("<", length, ":")
        for i := 0; ; i++ {
            if msg[i] == 0 {
                break
            }
            fmt.Printf("%c", msg[i])
```

```
        }
        print(">")
    }
}

func checkError(error error, info string) {
    if error != nil {
        panic("ERROR: " + info + " " + error.Error()) // terminate program
    }
}
```

（译者注：应该是由于go版本的更新，会提示os.EAGAIN undefined ,修改后的代码：simple_tcp_server_v1.go）

都有哪些改进？

- 服务器地址和端口不再是硬编码，而是通过命令行参数传入，并通过 `flag` 包来读取这些参数。这里使用了 `flag.NArg()` 检查是否按照期望传入了2个参数：

```
if flag.NArg() != 2 {
    panic("usage: host port")
}
```

传入的参数通过 `fmt.Sprintf` 函数格式化成字符串

```
hostAndPort := fmt.Sprintf("%s:%s", flag.Arg(0), flag.Arg(1))
```

- 在 `initServer` 函数中通过 `net.ResolveTCPAddr` 得到了服务器地址和端口，这个函数最终返回了一个 `*net.TCPListener`
- 每一个连接都会以协程的方式运行 `connectionHandler` 函数。函数首先通过 `conn.RemoteAddr()` 获取到客户端的地址并显示出来
- 它使用 `conn.Write` 发送 Go 推广消息给客户端
- 它使用一个 25 字节的缓冲读取客户端发送的数据并一一打印出来。如果读取的过程中出现错误，代码会进入 `switch` 语句 `default` 分支，退出无限循环并关闭连接。如果是操作系统的 `EAGAIN` 错误，它会重试。
- 所有的错误检查都被重构在独立的函数 `checkError` 中，当错误产生时，利用错误上下文来触发 panic。

在命令行中输入 `simple_tcp_server localhost 50000` 来启动服务器程序，然后在独立的命令行窗口启动一些 client.go 的客户端。当有两个客户端连接的情况下服务器的典型输出如下，这里我们可以看到每个客户端都有自己的地址：

```
E:\Go\GoBoek\code examples\chapter 14>simple_tcp_server localhost 50000
Listening to: 127.0.0.1:50000
Connection from: 127.0.0.1:49346
<25:Ivo says: Hi server, do y><12:ou hear me ?>
Connection from: 127.0.0.1:49347
<25:Marc says: Do you remembe><25:r our first meeting serve><2:r?>
```

net.Error：

`net` 包返回的错误类型遵循惯例为 `error` ，但有些错误实现包含额外的方法，他们被定义为 `net.Error` 接口：

```
package net

type Error interface {
    Timeout() bool // 错误是否超时
```

```
    Temporary() bool // 是否是临时错误
}
```

通过类型断言，客户端代码可以测试 `net.Error` ，从而区分是临时发生的还是必然会出现的错误。举例来说，一个网络爬虫程序在遇到临时发生的错误时可能会休眠或者重试，如果是一个必然发生的错误，则他会放弃继续执行。

```
// in a loop - some function returns an error err
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue // try again
}
if err != nil {
    log.Fatal(err)
}
```

一个简单的 web 服务器

15.2 一个简单的 web 服务器

http 是比 tcp 更高层的协议，它描述了网页服务器如何与客户端浏览器进行通信。Go 提供了 `net/http` 包，我们马上就来看下。先从一些简单的示例开始，首先编写一个“Hello world!”网页服务器：查看示例15.6

我们引入了 `http` 包并启动了网页服务器，和 15.1节 的 `net.Listen("tcp", "localhost:50000")` 函数的 tcp 服务器是类似的，使用 `http.ListenAndServe("localhost:8080", nil)` 函数，如果成功会返回空，否则会返回一个错误（地址 localhost 部分可以省略，8080 是指定的端口号）。

`http.URL` 用于表示网页地址，其中字符串属性 `Path` 用于保存 url 的路径；`http.Request` 描述了客户端请求，内含一个 `URL` 字段。

如果 `req` 是来自 html 表单的 POST 类型请求，“var1” 是该表单中一个输入域的名称，那么用户输入的值就可以通过 Go 代码 `req.FormValue("var1")` 获取到（见 15.4节）。还有一种方法是先执行 `request.ParseForm()`，然后再获取 `request.Form["var1"]` 的第一个返回参数，就像这样：

```
var1, found := request.Form["var1"]
```

第二个参数 `found` 为 `true`。如果 `var1` 并未出现在表单中，`found` 就是 `false`。

表单属性实际上是 `map[string][]string` 类型。网页服务器发送一个 `http.Response` 响应，它是通过 `http.ResponseWriter` 对象输出的，后者组装了 HTTP 服务器响应，通过对其写入内容，我们就将数据发送给了 HTTP 客户端。

现在我们仍然要编写程序，以实现服务器必须做的事，即如何处理请求。这是通过 `http.HandleFunc` 函数完成的。在这个例子中，当根路径“/”（url地址是 `http://localhost:8080`）被请求的时候（或者这个服务器上的其他任意地址），`HelloServer` 函数就被执行了。这个函数是 `http.HandlerFunc` 类型的，它们通常被命名为 Prefhandler，和某个路径前缀 Pref 匹配。

`http.HandleFunc` 注册了一个处理函数（这里是 `HelloServer`）来处理对应 `/` 的请求。

`/` 可以被替换为其他更特定的 url，比如 `/create`，`/edit` 等等；你可以为每一个特定的 url 定义一个单独的处理函数。这个函数需要两个参数：第一个是 `ReponseWriter` 类型的 `w`；第二个是请求 `req`。程序向 `w` 写入了 `Hello` 和 `r.URL.Path[1:]` 组成的字符串：末尾的 `[1:]` 表示“创建一个从索引为 1 的字符到结尾的子切片”，用来丢弃路径开头的“/”，`fmt.Fprintf()` 函数完成了本次写入（见 12.8节）；另一种可行的写法是 `io.WriteString(w, "hello, world!\n")`。

总结：第一个参数是请求的路径，第二个参数是当路径被请求时，需要调用的处理函数的引用。

示例 15.6 hello_world_webserver.go：

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func HelloServer(w http.ResponseWriter, req *http.Request) {
    fmt.Println("Inside HelloServer handler")
    fmt.Fprintf(w, "Hello,"+req.URL.Path[1:])
}
```

```
func main() {
    http.HandleFunc("/", HelloServer)
    err := http.ListenAndServe("localhost:8080", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err.Error())
    }
}
```

使用命令行启动程序，会打开一个命令窗口显示如下文字：

```
Starting Process E:/Go/GoBoek/code_examples/chapter_14/hello_world_webserver.exe...
```

然后打开浏览器并输入 url 地址： `http://localhost:8080/world` ，浏览器就会出现文字： `Hello, world` ，网页服务器会响应你在 `:8080/` 后边输入的内容。

`fmt.Println` 在服务器端控制台打印状态；在每个处理函数被调用时，把请求记录下来也许更为有用。

注：
1）前两行（没有错误处理代码）可以替换成以下写法：

```
http.ListenAndServe(":8080", http.HandlerFunc(HelloServer))
```

2） `fmt.Fprint` 和 `fmt.Fprintf` 都是可以用来写入 `http.ResponseWriter` 的函数（他们实现了 `io.Writer` ）。
比如我们可以使用

```
fmt.Fprintf(w, "<h1>%s<h1><div>%s</div>", title, body)
```

来构建一个非常简单的网页并插入 `title` 和 `body` 的值。

如果你需要更多复杂的替换，使用模板包（见 15.7节）

3）如果你需要使用安全的 https 连接，使用 `http.ListenAndServeTLS()` 代替 `http.ListenAndServe()`

4）除了 `http.HandleFunc("/", Hfunc)` ，其中的 `HFunc` 是一个处理函数，签名为：

```
func HFunc(w http.ResponseWriter, req *http.Request) {
    ...
}
```

也可以使用这种方式： `http.Handle("/", http.HandlerFunc(HFunc))`

`HandlerFunc` 只是定义了上述 HFunc 签名的别名：

```
type HandlerFunc func(ResponseWriter, *Request)
```

它是一个可以把普通的函数当做 HTTP 处理器（ `Handler` ）的适配器。如果函数 `f` 声明的合适， `HandlerFunc(f)` 就是一个执行 `f` 函数的 `Handler` 对象。

`http.Handle` 的第二个参数也可以是 `T` 类型的对象 obj： `http.Handle("/", obj)` 。

如果 T 有 `ServeHTTP` 方法，那就实现了http 的 `Handler` 接口：

```
func (obj *Typ) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ...
}
```

这个用法也在 15.8节 `Counter` 和 `Chan` 类型上使用。只要实现了 `http.Handler`，`http` 包就可以处理任何 HTTP 请求。

练习 15.2：webhello2.go

编写一个网页服务器监听端口 9999，有如下处理函数：

- 当请求 `http://localhost:9999/hello/Name` 时，响应：`hello Name`（Name 需是一个合法的姓，比如 Chris 或者 Madeleine）
- 当请求 `http://localhost:9999/shouthello/Name` 时，响应：`hello NAME`

练习 15.3：hello_server.go

创建一个空结构 `hello` 并为它实现 `http.Handler`。运行并测试。

访问并读取页面数据

15.3 访问并读取页面数据

在下边这个程序中，数组中的 url 都将被访问：会发送一个简单的 `http.Head()` 请求查看返回值；它的声明如下：`func Head(url string) (r *Response, err error)`

返回的响应 `Response` 其状态码会被打印出来。

示例 15.7 poll_url.go：

```
package main

import (
    "fmt"
    "net/http"
)

var urls = []string{
    "http://www.google.com/",
    "http://golang.org/",
    "http://blog.golang.org/",
}

func main() {
    // Execute an HTTP HEAD request for all url's
    // and returns the HTTP status string or an error string.
    for _, url := range urls {
        resp, err := http.Head(url)
        if err != nil {
            fmt.Println("Error:", url, err)
        }
        fmt.Println(url, ": ", resp.Status)
    }
}
```

输出为：

```
http://www.google.com/ : 302 Found
http://golang.org/ : 200 OK
http://blog.golang.org/ : 200 OK
```

译者注 由于国内的网络环境现状，很有可能见到如下超时错误提示：
Error: http://www.google.com/ Head http://www.google.com/: dial tcp 216.58.221.100:80: connectex: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.

在下边的程序中我们使用 `http.Get()` 获取并显示网页内容；`Get` 返回值中的 `Body` 属性包含了网页内容，然后我们用 `ioutil.ReadAll` 把它读出来：

示例 15.8 http_fetch.go：

```
package main
```

```
import (
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
)

func main() {
    res, err := http.Get("http://www.google.com")
    checkError(err)
    data, err := ioutil.ReadAll(res.Body)
    checkError(err)
    fmt.Printf("Got: %q", string(data))
}

func checkError(err error) {
    if err != nil {
        log.Fatalf("Get : %v", err)
    }
}
```

当访问不存在的网站时，这里有一个 `CheckError` 输出错误的例子：

```
2011/09/30 11:24:15 Get: Get http://www.google.bex: dial tcp www.google.bex:80:GetHostByName: No such host is known.
```

译者注 和上一个例子相似，你可以把google.com更换为一个国内可以顺畅访问的网址进行测试

在下边的程序中，我们获取一个 twitter 用户的状态，通过 `xml` 包将这个状态解析成为一个结构：

示例 15.9 twitter_status.go

```
package main

import (
    "encoding/xml"
    "fmt"
    "net/http"
)

/*这个结构会保存解析后的返回数据。
他们会形成有层级的XML，可以忽略一些无用的数据*/
type Status struct {
    Text string
}

type User struct {
    XMLName xml.Name
    Status  Status
}

func main() {
    // 发起请求查询推特Goodland用户的状态
    response, _ := http.Get("http://twitter.com/users/Googland.xml")
    // 初始化XML返回值的结构
    user := User{xml.Name{"", "user"}, Status{""}}
    // 将XML解析为我们的结构
    xml.Unmarshal(response.Body, &user)
```

```
    fmt.Printf("status: %s", user.Status.Text)
}
```

输出：

```
status: Robot cars invade California, on orders from Google: Google has been testing self-driving cars ... http://bit.ly/cbtpUN http://retwt.me/97p<exit code="0" msg="process exited normally"/>
```

译者注 和上边的示例相似，你可能无法获取到xml数据，另外由于go版本的更新，`xml.Unmarshal`函数的第一个参数需是[]byte类型，而无法传入`Body`。

我们会在 15.4节 中用到`http`包中的其他重要的函数：

- `http.Redirect(w ResponseWriter, r *Request, url string, code int)`：这个函数会让浏览器重定向到`url`（可以是基于请求 url 的相对路径），同时指定状态码。
- `http.NotFound(w ResponseWriter, r *Request)`：这个函数将返回网页没有找到，HTTP 404错误。
- `http.Error(w ResponseWriter, error string, code int)`：这个函数返回特定的错误信息和 HTTP 代码。
- 另一个`http.Request`对象`req`的重要属性：`req.Method`，这是一个包含`GET`或`POST`字符串，用来描述网页是以何种方式被请求的。

go为所有的HTTP状态码定义了常量，比如：

```
http.StatusContinue           = 100
http.StatusOK                 = 200
http.StatusFound              = 302
http.StatusBadRequest         = 400
http.StatusUnauthorized       = 401
http.StatusForbidden          = 403
http.StatusNotFound           = 404
http.StatusInternalServerError    = 500
```

你可以使用`w.header().Set("Content-Type", "../..")`设置头信息。

比如在网页应用发送 html 字符串的时候，在输出之前执行`w.Header().Set("Content-Type", "text/html")`（通常不是必要的）。

练习 15.4：扩展 http_fetch.go 使之可以从控制台读取url，使用 12.1节 学到的接收控制台输入的方法（http_fetch2.go）

练习 15.5：获取 json 格式的推特状态，就像示例 15.9（twitter_status_json.go）

写一个简单的网页应用

15.4 写一个简单的网页应用

下边的程序在端口 8088 上启动了一个网页服务器；`SimpleServer` 会处理 url `/test1` 使它在浏览器输出 `hello world` 。`FormServer` 会处理 url `/test2` ：如果 url 最初由浏览器请求，那么它是一个 `GET` 请求，返回一个 `form` 常量，包含了简单的 `input` 表单，这个表单里有一个文本框和一个提交按钮。当在文本框输入一些东西并点击提交按钮的时候，会发起一个 POST 请求。`FormServer` 中的代码用到了 `switch` 来区分两种情况。请求为 POST 类型时，`name` 属性 为 `inp` 的文本框的内容可以这样获取：`request.FormValue("inp")` 。然后将其写回浏览器页面中。在控制台启动程序，然后到浏览器中打开 url `http://localhost:8088/test2` 来测试这个程序：

示例 15.10 simple_webserver.go

```
package main

import (
    "io"
    "net/http"
)

const form = `
    <html><body>
        <form action="#" method="post" name="bar">
            <input type="text" name="in" />
            <input type="submit" value="submit"/>
        </form>
    </body></html>
`

/* handle a simple get request */
func SimpleServer(w http.ResponseWriter, request *http.Request) {
    io.WriteString(w, "<h1>hello, world</h1>")
}

func FormServer(w http.ResponseWriter, request *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    switch request.Method {
    case "GET":
        /* display the form to the user */
        io.WriteString(w, form)
    case "POST":
        /* handle the form data, note that ParseForm must
           be called before we can extract form data */
        //request.ParseForm();
        //io.WriteString(w, request.Form["in"][0])
        io.WriteString(w, request.FormValue("in"))
    }
}

func main() {
    http.HandleFunc("/test1", SimpleServer)
    http.HandleFunc("/test2", FormServer)
    if err := http.ListenAndServe(":8088", nil); err != nil {
        panic(err)
```

```
    }
}
```

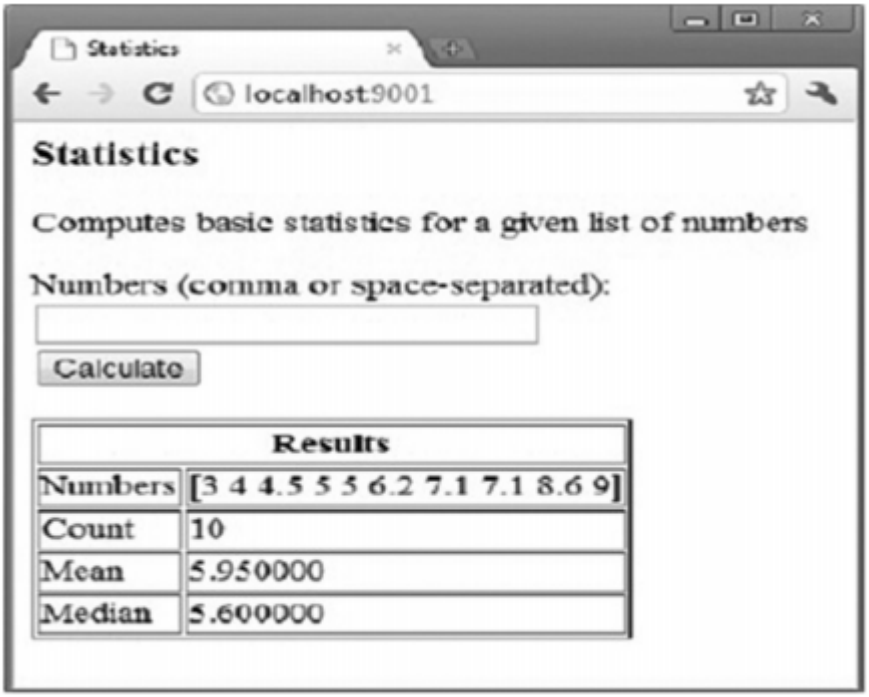
注：当使用字符串常量表示 html 文本的时候，包含 `<html><body>...</body></html>` 对于让浏览器将它识别为 html 文档非常重要。

更安全的做法是在处理函数中，在写入返回内容之前将头部的 `content-type` 设置为 `text/html` ： `w.Header().Set("Content-Type", "text/html")` 。

`content-type` 会让浏览器认为它可以使用函数 `http.DetectContentType([]byte(form))` 来处理收到的数据。

练习 15.6 statistics.go

编写一个网页程序，可以让用户输入一连串的数字，然后将它们打印出来，计算出这些数字的均值和中值，就像下边这张截图一样：



确保网页应用健壮

15.5 确保网页应用健壮

当网页应用的处理函数发生 panic，服务器会简单地终止运行。这可不妙：网页服务器必须是足够健壮的程序，能够承受任何可能的突发问题。

首先能想到的是在每个处理函数中使用 `defer/recover`，不过这样会产生太多的重复代码。13.5节 使用闭包的错误处理模式是更优雅的方案。我们把这种机制应用到前一章的简单网页服务器上。实际上，它可以被简单地应用到任何网页服务器程序中。

为增强代码可读性，我们为页面处理函数创建一个类型：

```
type HandleFnc func(http.ResponseWriter, *http.Request)
```

我们的错误处理函数应用了13.5节 的模式，成为 `logPanics` 函数：

```
func logPanics(function HandleFnc) HandleFnc {
    return func(writer http.ResponseWriter, request *http.Request) {
        defer func() {
            if x := recover(); x != nil {
                log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
            }
        }()
        function(writer, request)
    }
}
```

然后我们用 `logPanics` 来包装对处理函数的调用：

```
http.HandleFunc("/test1", logPanics(SimpleServer))
http.HandleFunc("/test2", logPanics(FormServer))
```

处理函数现在可以恢复 panic 调用，类似13.5节 中的错误检测函数。完整代码如下：

示例 15.11 robust_webserver.go

```
package main

import (
    "io"
    "log"
    "net/http"
)

const form = `<html><body><form action="#" method="post" name="bar">
        <input type="text" name="in"/>
        <input type="submit" value="Submit"/>
    </form></html></body>`

type HandleFnc func(http.ResponseWriter, *http.Request)

/* handle a simple get request */
```

```
func SimpleServer(w http.ResponseWriter, request *http.Request) {
    io.WriteString(w, "<h1>hello, world</h1>")
}

/* handle a form, both the GET which displays the form
   and the POST which processes it.*/
func FormServer(w http.ResponseWriter, request *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    switch request.Method {
    case "GET":
        /* display the form to the user */
        io.WriteString(w, form)
    case "POST":
        /* handle the form data, note that ParseForm must
           be called before we can extract form data*/
        //request.ParseForm();
        //io.WriteString(w, request.Form["in"][0])
        io.WriteString(w, request.FormValue("in"))
    }
}

func main() {
    http.HandleFunc("/test1", logPanics(SimpleServer))
    http.HandleFunc("/test2", logPanics(FormServer))
    if err := http.ListenAndServe(":8088", nil); err != nil {
        panic(err)
    }
}

func logPanics(function HandleFnc) HandleFnc {
    return func(writer http.ResponseWriter, request *http.Request) {
        defer func() {
            if x := recover(); x != nil {
                log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
            }
        }()
        function(writer, request)
    }
}
```

用模板编写网页应用

15.6 用模板编写网页应用

以下程序是用 100 行以内代码实现可行的 wiki 网页应用，它由一组页面组成，用于阅读、编辑和保存。它是来自 Go 网站 codelab 的 wiki 制作教程，我所知的最好的 Go 教程之一，非常值得进行完整的实验，以见证并理解程序是如何被构建起来的（https://golang.org/doc/articles/wiki/）。这里，我们将以自顶向下的视角，从整体上给出程序的补充说明。程序是网页服务器，它必须从命令行启动，监听某个端口，例如 8080。浏览器可以通过请求 URL 阅读 wiki 页面的内容，例如：`http://localhost:8080/view/page1`。

接着，页面的文本内容从一个文件中读取，并显示在网页中。它包含一个超链接，指向编辑页面（`http://localhost:8080/edit/page1`）。编辑页面将内容显示在一个文本域中，用户可以更改文本，点击“保存”按钮保存到对应的文件中。然后回到阅读页面显示更改后的内容。如果某个被请求阅读的页面不存在（例如：`http://localhost:8080/edit/page999`），程序可以作出识别，立即重定向到编辑页面，如此新的 wiki 页面就可以被创建并保存。

wiki 页面需要一个标题和文本内容，它在程序中被建模为如下结构体，Body 字段存放内容，由字节切片组成。

```
type Page struct {
    Title string
    Body  []byte
}
```

为了在可执行程序之外维护 wiki 页面内容，我们简单地使用了文本文件作为持久化存储。程序、必要的模板和文本文件可以在 wiki 中找到。

示例 15.12 wiki.go

```
package main

import (
    "net/http"
    "io/ioutil"
    "log"
    "regexp"
    "text/template"
)

const lenPath = len("/view/")

var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")
var templates = make(map[string]*template.Template)
var err error

type Page struct {
    Title string
    Body  []byte
}

func init() {
    for _, tmpl := range []string{"edit", "view"} {
        templates[tmpl] = template.Must(template.ParseFiles(tmpl + ".html"))
    }
}
```

```
func main() {
    http.HandleFunc("/view/", makeHandler(viewHandler))
    http.HandleFunc("/edit/", makeHandler(editHandler))
    http.HandleFunc("/save/", makeHandler(saveHandler))
    err := http.ListenAndServe("localhost:8080", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err.Error())
    }
}

func makeHandler(fn func(http.ResponseWriter, *http.Request, string)) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        title := r.URL.Path[lenPath:]
        if !titleValidator.MatchString(title) {
            http.NotFound(w, r)
            return
        }
        fn(w, r, title)
    }
}

func viewHandler(w http.ResponseWriter, r *http.Request, title string) {
    p, err := load(title)
    if err != nil { // page not found
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request, title string) {
    p, err := load(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request, title string) {
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err := p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}

func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
    err := templates[tmpl].Execute(w, p)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}

func (p *Page) save() error {
    filename := p.Title + ".txt"
    // file created with read-write permissions for the current user only
    return ioutil.WriteFile(filename, p.Body, 0600)
}
```

```
func load(title string) (*Page, error) {
    filename := title + ".txt"
    body, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil, err
    }
    return &Page{Title: title, Body: body}, nil
}
```

让我们来通读代码：

- 首先导入必要的包。由于我们在构建网页服务器， `http` 当然是必须的。不过还导入了 `io/ioutil` 来方便地读写文件， `regexp` 用于验证输入标题，以及 `template` 来动态创建 html 文档。
- 为避免黑客构造特殊输入攻击服务器，我们用如下正则表达式检查用户在浏览器上输入的 URL（同时也是 wiki 页面标题）：

  ```
  var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")
  ```

  `makeHandler` 会用它对请求管控。
- 必须有一种机制把 `Page` 结构体数据插入到网页的标题和内容中，可以利用 `template` 包通过如下步骤完成：

  i. 先在文本编辑器中创建 html 模板文件，例如 view.html：

  ```
  <h1>{{.Title |html}}</h1>
  <p>[<a href="/edit/{{.Title |html}}">edit</a>]</p>
  <div>{{printf "%s" .Body |html}}</div>
  ```

  把要插入的数据结构字段放在 `{{` 和 `}}` 之间，这里是把 `Page` 结构体数据 `{{.Title |html}}` 和 `{{printf "%s" .Body |html}}` 插入页面（当然可以是非常复杂的 html，但这里尽可能地简化了，以突出模板的原理。）（ `{{.Title |html}}` 和 `{{printf "%s" .Body |html}}` 语法说明详见后续章节）。

  ii. `template.Must(template.ParseFiles(tmpl + ".html"))` 把模板文件转换为 `*template.Template` 类型的对象，为了高效，在程序运行时仅做一次解析，在 `init()` 函数中处理可以方便地达到目的。所有模板对象都被保持在内存中，存放在以 html 文件名作为索引的 map 中：

  ```
  templates = make(map[string]*template.Template)
  ```

  此种技术被称为*模板缓存*，是推荐的最佳实践。

  iii. 为了真正从模板和结构体构建出页面，必须使用：

  ```
  templates[tmpl].Execute(w, p)
  ```

  它基于模板执行，用 `Page` 结构体对象 p 作为参数对模板进行替换，并写入 `ResponseWriter` 对象 w。必须检查该方法的 error 返回值，万一有一个或多个错误，我们可以调用 `http.Error` 来明示。在我们的应用程

序中，这段代码会被多次调用，所以把它提取为单独的函数 `renderTemplate` 。

- 在 `main()` 中网页服务器用 `ListenAndServe` 启动并监听 8080 端口。但正如 15.2节 那样，需要先为紧接在 URL `localhost:8080/` 之后， 以 `view` ， `edit` 或 `save` 开头的 url 路径定义一些处理函数。在大多数网页服务器应用程序中，这形成了一系列 URL 路径到处理函数的映射，类似于 Ruby 和 Rails，Django 或 ASP.NET MVC 这样的 MVC 框架中的路由表。请求的 URL 与这些路径尝试匹配，较长的路径被优先匹配。如不与任何路径匹配，则调用 / 的处理程序。

  在此定义了 3 个处理函数，由于包含重复的启动代码，我们将其提取到单独的 `makeHandler` 函数中。这是一个值得研究的特殊高阶函数：其参数是一个函数，返回一个新的闭包函数：

```
func makeHandler(fn func(http.ResponseWriter, *http.Request, string)) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
      title := r.URL.Path[lenPath:]
      if !titleValidator.MatchString(title) {
          http.NotFound(w, r)
          return
      }
      fn(w, r, title)
  }
}
```

- 闭包封闭了函数变量 `fn` 来构造其返回值。但在此之前，它先用 `titleValidator.MatchString(title)` 验证输入标题 `title` 的有效性。如果标题包含了字母和数字以外的字符，就触发 NotFound 错误（例如：尝试 `localhost:8080/view/page++` ）。 `viewHandler` ， `editHandler` 和 `saveHandler` 都是传入 `main()` 中 `makeHandler` 的参数，类型必须都与 `fn` 相同。

- `viewHandler` 尝试按标题读取文本文件，这是通过调用 `load()` 函数完成的，它会构建文件名并用 `ioutil.ReadFile` 读取内容。如果文件存在，其内容会存入字符串中。一个指向 `Page` 结构体的指针按字面量被创建： `&Page{Title: title, Body: body}` 。

  另外，该值和表示没有 error 的 nil 值一起返回给调用者。然后在 `renderTemplate` 中将该结构体与模板对象整合。

  万一发生错误，也就是说 wiki 页面在磁盘上不存在，错误会被返回给 `viewHandler` ，此时会自动重定向，跳转请求对应标题的编辑页面。

- `editHandler` 基本上也差不多：尝试读取文件，如果存在则用“编辑”模板来渲染；万一发生错误，创建一个新的包含指定标题的 `Page` 对象并渲染。

- 当在编辑页面点击“保存”按钮时，触发保存页面内容的动作。按钮须放在 html 表单中，它开头是这样的：

```
<form action="/save/{{.Title |html}}" method="POST">
```

  这意味着，当提交表单到类似 `http://localhost/save/{Title}` 这样的 URL 格式时，一个 POST 请求被发往网页服务器。针对这样的 URL 我们已经定义好了处理函数： `saveHandler()` 。在 request 对象上调用 `FormValue()` 方法，可以提取名称为 body 的文本域内容，用这些信息构造一个 `Page` 对象，然后尝试通过调用 `save()` 方法保存其内容。万一运行失败，执行 `http.Error` 以将错误显示到浏览器。如果保存成功，重

定向浏览器到该页的阅读页面。 `save()` 函数非常简单，利用 `ioutil.WriteFile()` ，写入 `Page` 结构体的 `Body` 字段到文件 `filename` 中，之后会被用于模板替换占位符 `{{printf "%s" .Body |html}}` 。

探索 template 包

15.7 探索 template 包

（template 包的文档可以在 https://golang.org/pkg/text/template/ 找到。）

在前一章节，我们使用 template 对象把数据结构整合到 HTML 模板中。这项技术确实对网页应用程序非常有用，然而模板是一项更为通用的技术方案：数据驱动的模板被创建出来，以生成文本输出。HTML 仅是其中的一种特定使用案例。

模板通过与数据结构的整合来生成，通常为结构体或其切片。当数据项传递给 `tmpl.Execute()` ，它用其中的元素进行替换， 动态地重写某一小段文本。**只有被导出的数据项**才可以被整合进模板中。可以在 `{{` 和 `}}` 中加入数据求值或控制结构。数据项可以是值或指针，接口隐藏了他们的差异。

15.7.1 字段替换：`{{.FieldName}}`

要在模板中包含某个字段的内容，使用双花括号括起以点（ `.` ）开头的字段名。例如，假设 `Name` 是某个结构体的字段，其值要在被模板整合时替换，则在模板中使用文本 `{{.Name}}` 。当 `Name` 是 map 的键时这么做也是可行的。要创建一个新的 Template 对象，调用 `template.New` ，其字符串参数可以指定模板的名称。正如 15.5节 出现过的， `Parse` 方法通过解析模板定义字符串，生成模板的内部表示。当使用包含模板定义字符串的文件时，将文件路径传递给 `ParseFiles` 来解析。解析过程如产生错误，这两个函数第二个返回值 error != nil。最后通过 `Execute` 方法，数据结构中的内容与模板整合，并将结果写入方法的第一个参数中，其类型为 `io.Writer` 。再一次地，可能会有 error 返回。以下程序演示了这些步骤，输出通过 `os.Stdout` 被写到控制台。

示例 15.13 template_field.go

```
package main

import (
    "fmt"
    "os"
    "text/template"
)

type Person struct {
    Name string
    nonExportedAgeField string
}

func main() {
    t := template.New("hello")
    t, _ = t.Parse("hello {{.Name}}!")
    p := Person{Name: "Mary", nonExportedAgeField: "31"}
    if err := t.Execute(os.Stdout, p); err != nil {
        fmt.Println("There was an error:", err.Error())
    }
}
```

输出： `hello Mary!`

数据结构中包含一个未导出的字段，当我们尝试把它整合到类似这样的定义字符串：

```
t, _ = t.Parse("your age is {{.nonExportedAgeField}}!")
```

会产生错误：

```
There was an error: template: nonexported template hello:1: can’ t evaluate field nonExportedAgeField in type main.Person.
```

如果只是想简单地把 `Execute()` 方法的第二个参数用于替换，使用 `{{.}}` 。

当在浏览器环境中进行这些步骤，应首先使用 `html` 过滤器来过滤内容，例如 `{{html .}}` ， 或者对 `FieldName` 过滤： `{{ .FieldName |html }}` 。

`|html` 这部分代码，是请求模板引擎在输出 `FieldName` 的结果前把值传递给 html 格式化器，它会执行 HTML 字符转义（例如把 `>` 替换为 `>` ）。这可以避免用户输入数据破坏 HTML 文档结构。

15.7.2 验证模板格式

为了确保模板定义语法是正确的，使用 `Must` 函数处理 `Parse` 的返回结果。在下面的例子中 `tOK` 是正确的模板， `tErr` 验证时发生错误，会导致运行时 panic。

示例 15.14 template_validation.go

```
package main

import (
    "text/template"
    "fmt"
)

func main() {
    tOk := template.New("ok")
    //a valid template, so no panic with Must:
    template.Must(tOk.Parse("/* and a comment */ some static text: {{ .Name }}"))
    fmt.Println("The first one parsed OK.")
    fmt.Println("The next one ought to fail.")
    tErr := template.New("error_template")
    template.Must(tErr.Parse(" some static text {{ .Name }"))
}
```

输出：

```
The first one parsed OK.
The next one ought to fail.
panic: template: error_template:1: unexpected "}" in operand
```

模板语法出现错误比较少见，可以使用 13.3节 概括的 `defer/recover` 机制来报告并纠正错误。

在代码中常见到这 3 个基本函数被串联使用：

```
var strTempl = template.Must(template.New("TName").Parse(strTemplateHTML))
```

练习 15.7 template_validation_recover.go

在上述示例代码上实现 defer/recover 机制。

15.7.3 `If-else`

运行 `Execute` 产生的结果来自模板的输出，它包含静态文本，以及被 `{{}}` 包裹的称之为*管道*的文本。例如，运行这段代码（示例 15.15 pipline1.go）：

```
t := template.New("template test")
t = template.Must(t.Parse("This is just static text. \n{{\"This is pipeline data - because it is evaluated within the double braces.\"}} {{`So is this, but within reverse quotes.`}}\n"))
t.Execute(os.Stdout, nil)
```

输出结果为：

```
This is just static text.
This is pipeline data—because it is evaluated within the double braces. So is this, but within reverse quotes.
```

现在我们可以对管道数据的输出结果用 `if-else-end` 设置条件约束：如果管道是空的，类似于：

```
{{if ``}} Will not print. {{end}}
```

那么 `if` 条件的求值结果为 `false`，不会有输出内容。但如果是这样：

```
{{if `anything`}} Print IF part. {{else}} Print ELSE part.{{end}}
```

会输出 `Print IF part.`。以下程序演示了这点：

示例 15.16 template_ifelse.go

```
package main

import (
    "os"
    "text/template"
)

func main() {
    tEmpty := template.New("template test")
    tEmpty = template.Must(tEmpty.Parse("Empty pipeline if demo: {{if ``}} Will not print. {{end}}\n")) //empty pipeline following if
    tEmpty.Execute(os.Stdout, nil)

    tWithValue := template.New("template test")
    tWithValue = template.Must(tWithValue.Parse("Non empty pipeline if demo: {{if `anything`}} Will print. {{end}}\n")) //non empty pipeline following if condition
    tWithValue.Execute(os.Stdout, nil)

    tIfElse := template.New("template test")
    tIfElse = template.Must(tIfElse.Parse("if-else demo: {{if `anything`}} Print IF part. {{else}} Print ELSE part.{{end}}\n")) //non empty pipeline following if condition
    tIfElse.Execute(os.Stdout, nil)
}
```

输出：

```
Empty pipeline if demo:
Non empty pipeline if demo: Will print.
if-else demo: Print IF part.
```

15.7.4 点号和 `with-end`

点号（ `.` ）可以在 Go 模板中使用：其值 `{{.}}` 被设置为当前管道的值。

`with` 语句将点号设为管道的值。如果管道是空的，那么不管 `with-end` 块之间有什么，都会被忽略。在被嵌套时，点号根据最近的作用域取得值。以下程序演示了这点：

示例 15.17 template_with_end.go

```
package main

import (
    "os"
    "text/template"
)

func main() {
    t := template.New("test")
    t, _ = t.Parse("{{with `hello`}}{{.}}{{end}}!\n")
    t.Execute(os.Stdout, nil)

    t, _ = t.Parse("{{with `hello`}}{{.}} {{with `Mary`}}{{.}}{{end}}{{end}}!\n")
    t.Execute(os.Stdout, nil)
}
```

输出：

```
hello!
hello Mary!
```

15.7.5 模板变量 `$`

可以在模板内为管道设置本地变量，变量名以 `$` 符号作为前缀。变量名只能包含字母、数字和下划线。以下示例使用了多种形式的有效变量名。

示例 15.18 template_variables.go

```
package main

import (
    "os"
    "text/template"
)

func main() {
    t := template.New("test")
    t = template.Must(t.Parse("{{with $3 := `hello`}}{{$3}}{{end}}!\n"))
    t.Execute(os.Stdout, nil)

    t = template.Must(t.Parse("{{with $x3 := `hola`}}{{$x3}}{{end}}!\n"))
    t.Execute(os.Stdout, nil)

    t = template.Must(t.Parse("{{with $x_1 := `hey`}}{{$x_1}} {{.}} {{$x_1}}{{end}}!\n"))
    t.Execute(os.Stdout, nil)
}
```

输出：

```
hello!
hola!
hey hey hey!
```

15.7.6 range-end

`range-end` 结构格式为：`{{range pipeline}} T1 {{else}} T0 {{end}}`。

`range` 被用于在集合上迭代：管道的值必须是数组、切片或 map。如果管道的值长度为零，点号的值不受影响，且执行 `T0`；否则，点号被设置为数组、切片或 map 内元素的值，并执行 `T1`。

如果模板为：

```
{{range .}}
{{.}}
{{end}}
```

那么执行代码：

```
s := []int{1,2,3,4}
t.Execute(os.Stdout, s)
```

会输出：

```
1
2
3
4
```

如需更实用的示例，请参考 20.7节，来自 App Engine 数据库的数据通过模板来显示：

```
{{range .}}
    {{with .Author}}
        <p><b>{{html .}}</b> wrote:</p>
    {{else}}
        <p>An anonymous person wrote:</p>
    {{end}}
    <pre>{{html .Content}}</pre>
    <pre>{{html .Date}}</pre>
{{end}}
```

这里 `range .` 在结构体切片上迭代，每次都包含 `Author`、`Content` 和 `Date` 字段。

15.7.7 模板预定义函数

也有一些可以在模板代码中使用的预定义函数，例如 `printf` 函数工作方式类似于 `fmt.Sprintf`：

示例 15.19 predefined_functions.go

```
package main
```

```
import (
    "os"
    "text/template"
)

func main() {
    t := template.New("test")
    t = template.Must(t.Parse("{{with $x := `hello`}}{{printf `%s %s` $x `Mary`}}{{end}}!\n"))
    t.Execute(os.Stdout, nil)
}
```

输出 `hello Mary!` 。

预定义函数也在 15.6节 中使用： `{{ printf "%s" .Body|html}}` ，否则字节切片 `Body` 会作为数字序列打印出来。

精巧的多功能网页服务器

15.8 精巧的多功能网页服务器

为进一步深入理解 `http` 包以及如何构建网页服务器功能，让我们来学习和体会下面的例子：先列出代码，然后给出不同功能的实现方法，程序输出显示在表格中。

示例 15.20 elaborated_webserver.go

```
package main

import (
    "bytes"
    "expvar"
    "flag"
    "fmt"
    "io"
    "log"
    "net/http"
    "os"
    "strconv"
)

// hello world, the web server
var helloRequests = expvar.NewInt("hello-requests")

// flags:
var webroot = flag.String("root", "/home/user", "web root directory")

// simple flag server
var booleanflag = flag.Bool("boolean", true, "another flag for testing")

// Simple counter server. POSTing to it will set the value.
type Counter struct {
    n int
}

// a channel
type Chan chan int

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(Logger))
    http.Handle("/go/hello", http.HandlerFunc(HelloServer))
    // The counter is published as a variable directly.
    ctr := new(Counter)
    expvar.Publish("counter", ctr)
    http.Handle("/counter", ctr)
    // http.Handle("/go/", http.FileServer(http.Dir("/tmp"))) // uses the OS filesystem
    http.Handle("/go/", http.StripPrefix("/go/", http.FileServer(http.Dir(*webroot))))
    http.Handle("/flags", http.HandlerFunc(FlagServer))
    http.Handle("/args", http.HandlerFunc(ArgServer))
    http.Handle("/chan", ChanCreate())
    http.Handle("/date", http.HandlerFunc(DateServer))
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
```

```
        log.Panicln("ListenAndServe:", err)
    }
}

func Logger(w http.ResponseWriter, req *http.Request) {
    log.Print(req.URL.String())
    w.WriteHeader(404)
    w.Write([]byte("oops"))
}

func HelloServer(w http.ResponseWriter, req *http.Request) {
    helloRequests.Add(1)
    io.WriteString(w, "hello, world!\n")
}

// This makes Counter satisfy the expvar.Var interface, so we can export
// it directly.
func (ctr *Counter) String() string { return fmt.Sprintf("%d", ctr.n) }

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    switch req.Method {
    case "GET": // increment n
        ctr.n++
    case "POST": // set n to posted value
        buf := new(bytes.Buffer)
        io.Copy(buf, req.Body)
        body := buf.String()
        if n, err := strconv.Atoi(body); err != nil {
            fmt.Fprintf(w, "bad POST: %v\nbody: [%v]\n", err, body)
        } else {
            ctr.n = n
            fmt.Fprint(w, "counter reset\n")
        }
    }
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}

func FlagServer(w http.ResponseWriter, req *http.Request) {
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    fmt.Fprint(w, "Flags:\n")
    flag.VisitAll(func(f *flag.Flag) {
        if f.Value.String() != f.DefValue {
            fmt.Fprintf(w, "%s = %s [default = %s]\n", f.Name, f.Value.String(), f.DefValue)
        } else {
            fmt.Fprintf(w, "%s = %s\n", f.Name, f.Value.String())
        }
    })
}

// simple argument server
func ArgServer(w http.ResponseWriter, req *http.Request) {
    for _, s := range os.Args {
        fmt.Fprint(w, s, " ")
    }
}

func ChanCreate() Chan {
    c := make(Chan)
    go func(c Chan) {
        for x := 0; ; x++ {
            c <- x
        }
```

```
    }(c)
    return c
}

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    io.WriteString(w, fmt.Sprintf("channel send #%d\n", <-ch))
}

// exec a program, redirecting output
func DateServer(rw http.ResponseWriter, req *http.Request) {
    rw.Header().Set("Content-Type", "text/plain; charset=utf-8")
    r, w, err := os.Pipe()
    if err != nil {
        fmt.Fprintf(rw, "pipe: %s\n", err)
        return
    }

    p, err := os.StartProcess("/bin/date", []string{"date"}, &os.ProcAttr{Files: []*os.File{nil, w, w}})
    defer r.Close()
    w.Close()
    if err != nil {
        fmt.Fprintf(rw, "fork/exec: %s\n", err)
        return
    }
    defer p.Release()
    io.Copy(rw, r)
    wait, err := p.Wait()
    if err != nil {
        fmt.Fprintf(rw, "wait: %s\n", err)
        return
    }
    if !wait.Exited() {
        fmt.Fprintf(rw, "date: %v\n", wait)
        return
    }
}
```

处理函数	调用 URL	浏览器获得响应
Logger	http://localhost:12345/ （根）	oops

`Logger` 处理函数用 `w.WriteHeader(404)` 来输出“404 Not Found”头部。

此项技术通常很有用，无论何时服务器执行代码产生错误，都可以应用类似这样的代码：

```
if err != nil {
    w.WriteHeader(400)
    return
}
```

另外利用 `logger` 包的函数，针对每个请求在服务器端命令行打印日期、时间和 URL。

处理函数	调用 URL	浏览器获得响应
HelloServer	http://localhost:12345/go/hello	hello, world!

包 `expvar` 可以创建（Int，Float 和 String 类型）变量，并将它们发布为公共变量。它会在 HTTP URL `/debug/vars` 上以 JSON 格式公布。通常它被用于服务器操作计数。 `helloRequests` 就是这样一个 `int64` 变量，该处理函数对其加 1，然后写入“hello world!”到浏览器。

处理函数	调用 URL	浏览器获得响应
Counter	http://localhost:12345/counter	counter = 1
Counter	刷新（GET 请求）	counter = 2

计数器对象 `ctr` 有一个 `String()` 方法，所以它实现了 `expvar.Var` 接口。这使其可以被发布，尽管它是一个结构体。 `ServeHTTP` 函数使 `ctr` 成为处理器，因为它的签名正确实现了 `http.Handler` 接口。

处理函数	调用 URL	浏览器获得响应
FileServer	http://localhost:12345/go/ggg.html	404 page not found

`FileServer(root FileSystem) Handler` 返回一个处理器，它以 `root` 作为根，用文件系统的内容响应 HTTP 请求。要获得操作系统的文件系统，用 `http.Dir` ，例如：

```
http.Handle("/go/", http.FileServer(http.Dir("/tmp")))
```

处理函数	调用 URL	浏览器获得响应
FlagServer	http://localhost:12345/flags	Flags: boolean = true root = /home/rsc

该处理函数使用了 `flag` 包。 `VisitAll` 函数迭代所有的标签（flag），打印它们的名称、值和默认值（当不同于“值”时）。

处理函数	调用 URL	浏览器获得响应
ArgServer	http://localhost:12345/args	./elaborated_webserver.exe

该处理函数迭代 `os.Args` 以打印出所有的命令行参数。如果没有指定则只有程序名称（可执行程序的路径）会被打印出来。

处理函数	调用 URL	浏览器获得响应
Channel	http://localhost:12345/chan	channel send #1
Channel	刷新	channel send #2

每当有新请求到达，通道的 `ServeHTTP` 方法从通道获取下一个整数并显示。由此可见，网页服务器可以从通道中获取要发送的响应，它可以由另一个函数产生（甚至是客户端）。下面的代码片段正是一个这样的处理函数，但会在 30 秒后超时：

```
func ChanResponse(w http.ResponseWriter, req *http.Request) {
    timeout := make (chan bool)
    go func () {
        time.Sleep(30e9)
        timeout <- true
    }()
    select {
    case msg := <-messages:
        io.WriteString(w, msg)
    case stop := <-timeout:
        return
    }
}
```

处理函数	调用 URL	浏览器获得响应
DateServer	http://localhost:12345/date	显示当前时间（由于是调用 /bin/date，仅在 Unix 下有效）

可能的输出： `Thu Sep 8 12:41:09 CEST 2011` 。

`os.Pipe()` 返回一对相关联的 `File` ，从 `r` 读取数据，返回已读取的字节数来自于 `w` 的写入。函数返回这两个文件和错误，如果有的话：

```
func Pipe() (r *File, w *File, err error)
```

用 rpc 实现远程过程调用

15.9 用 rpc 实现远程过程调用

Go 程序之间可以使用 `net/rpc` 包实现相互通信，这是另一种客户端-服务器应用场景。它提供了一种方便的途径，通过网络连接调用远程函数。当然，仅当程序运行在不同机器上时，这项技术才实用。 `rpc` 包建立在 `gob` 包之上（见12.11节），实现了自动编码/解码传输的跨网络方法调用。

服务器端需要注册一个对象实例，与其类型名一起，使之成为一项可见的服务：它允许远程客户端跨越网络或其他 I/O 连接访问此对象已导出的方法。总之就是在网络上暴露类型的方法。

`rpc` 包使用了 http 和 tcp 协议，以及用于数据传输的 `gob` 包。服务器端可以注册多个不同类型的对象（服务），但同一类型的多个对象会产生错误。

我们讨论一个简单的例子：定义一个类型 `Args` 及其方法 `Multiply` ，完美地置于单独的包中。方法必须返回可能的错误。

示例15.21 rpc_objects.go

```
package rpc_objects

import "net"

type Args struct {
    N, M int
}

func (t *Args) Multiply(args *Args, reply *int) net.Error {
    *reply = args.N * args.M
    return nil
}
```

（译注：Go 当前版本要求此方法返回类型为 `error` ，以上示例中返回 `net.Error` 已无法通过编译，见更新后的 rpc_objects.go。）

服务器端产生一个 `rpc_objects.Args` 类型的对象 `calc` ，并用 `rpc.Register(object)` 注册。调用 `HandleHTTP()` ，然后用 `net.Listen` 在指定的地址上启动监听。也可以按名称来注册对象，例如： `rpc.RegisterName("Calculator", calc)` 。

以协程启动 `http.Serve(listener, nil)` 后，会为每一个进入 `listener` 的 HTTP 连接创建新的服务线程。我们必须用诸如 `time.Sleep(1000e9)` 来使服务器在一段时间内保持运行状态。

示例 15.22 rpc_server.go

```
package main

import (
    "net/http"
    "log"
    "net"
    "net/rpc"
    "time"
    "./rpc_objects"
)
```

```
func main() {
    calc := new(rpc_objects.Args)
    rpc.Register(calc)
    rpc.HandleHTTP()
    listener, e := net.Listen("tcp", "localhost:1234")
    if e != nil {
        log.Fatal("Starting RPC-server -listen error:", e)
    }
    go http.Serve(listener, nil)
    time.Sleep(1000e9)
}
```

输出：

```
Starting Process E:/Go/GoBoek/code_examples/chapter_14/rpc_server.exe ...
** 5 秒后： **
End Process exit status 0
```

客户端必须知晓对象类型及其方法的定义。执行 `rpc.DialHTTP()` 连接到服务器后，就可以用 `client.Call("Type.Method", args, &reply)` 调用远程对象的方法。 `Type` 是远程对象的类型名， `Method` 是要调用的方法， `args` 是用 Args 类型初始化的对象， `reply` 是一个必须事先声明的变量，方法调用产生的结果将存入其中。

示例 15.23 rpc_client.go

```
package main

import (
    "fmt"
    "log"
    "net/rpc"
    "./rpc_objects"
)

const serverAddress = "localhost"

func main() {
    client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
    if err != nil {
        log.Fatal("Error dialing:", err)
    }
    // Synchronous call
    args := &rpc_objects.Args{7, 8}
    var reply int
    err = client.Call("Args.Multiply", args, &reply)
    if err != nil {
        log.Fatal("Args error:", err)
    }
    fmt.Printf("Args: %d * %d = %d", args.N, args.M, reply)
}
```

先启动服务器，再运行客户端，然后就能得到如下输出结果：

```
Starting Process E:/Go/GoBoek/code_examples/chapter_14/rpc_client.exe ...

Args: 7 * 8 = 56
End Process exit status 0
```

该远程调用以同步方式进行，它会等待服务器返回结果。也可使用如下方式异步地执行调用：

```
call1 := client.Go("Args.Multiply", args, &reply, nil)
replyCall := <- call1.Done
```

如果最后一个参数值为 `nil` ，调用完成后会创建一个新的通道。

如果你有一个以 root 管理员身份运行的 Go 服务器，想要以不同的用户身份运行某部分代码，Brad Fitz 利用 `rpc` 写的 `go-runas` 包可以完成任务：https://github.com/bradfitz/go-runas。我们将会在 19 章看到一个完整的项目，它是一个使用了 `rpc` 的应用程序。

基于网络的通道 netchan

15.10 基于网络的通道 netchan

备注：Go 团队决定改进并重新打造 `netchan` 包的现有版本，它已被移至 `old/netchan` 。 `old/` 目录用于存放过时的包代码，它们不会成为 Go 1.x 的一部分。本节仅出于向后兼容性讨论 `netchan` 包的概念。

一项和 `rpc` 密切相关的技术是基于网络的通道。类似 14 章所使用的通道都是本地的，它们仅存在于被执行的机器内存空间中。 `netchan` 包实现了类型安全的网络化通道：它允许一个通道两端出现由网络连接的不同计算机。其实现原理是，在其中一台机器上将传输数据发送到通道中，那么就可以被另一台计算机上同类型的通道接收。一个导出器（ `exporter` ）会按名称发布（一组）通道。导入器（ `importer` ）连接到导出的机器，并按名称导入这些通道。之后，两台机器就可按通常的方式来使用通道。网络通道不是同步的，它们类似于带缓存的通道。

发送端示例代码如下：

```
exp, err := netchan.NewExporter("tcp", "netchanserver.mydomain.com:1234")
if err != nil {
    log.Fatalf("Error making Exporter: %v", err)
}
ch := make(chan myType)
err := exp.Export("sendmyType", ch, netchan.Send)
if err != nil {
    log.Fatalf("Send Error: %v", err)
}
```

接收端示例代码如下：

```
imp, err := netchan.NewImporter("tcp", "netchanserver.mydomain.com:1234")
if err != nil {
    log.Fatalf("Error making Importer: %v", err)
}
ch := make(chan myType)
err = imp.Import("sendmyType", ch, netchan.Receive)
if err != nil {
    log.Fatalf("Receive Error: %v", err)
}
```

与 websocket 通信

15.11 与 websocket 通信

备注：Go 团队决定从 Go 1 起，将 `websocket` 包移出 Go 标准库，转移到 `code.google.com/p/go` 下的子项目 `websocket`，同时预计近期将做重大更改。

`import "websocket"` 这行要改成：

```
import websocket "code.google.com/p/go/websocket"
```

与 http 协议相反，websocket 是通过客户端与服务器之间的对话，建立的基于单个持久连接的协议。然而在其他方面，其功能几乎与 http 相同。在示例 15.24 中，我们有一个典型的 websocket 服务器，他会自启动并监听 websocket 客户端的连入。示例 15.25 演示了 5 秒后会终止的客户端代码。当连接到来时，服务器先打印 `new connection`，当客户端停止时，服务器打印 `EOF => closing connection`。

示例 15.24 websocket_server.go

```
package main

import (
    "fmt"
    "net/http"
    "websocket"
)

func server(ws *websocket.Conn) {
    fmt.Printf("new connection\n")
    buf := make([]byte, 100)
    for {
        if _, err := ws.Read(buf); err != nil {
            fmt.Printf("%s", err.Error())
            break
        }
    }
    fmt.Printf(" => closing connection\n")
    ws.Close()
}

func main() {
    http.Handle("/websocket", websocket.Handler(server))
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        panic("ListenAndServe: " + err.Error())
    }
}
```

示例 15.25 websocket_client.go

```
package main

import (
    "fmt"
    "time"
```

```
    "websocket"
)

func main() {
    ws, err := websocket.Dial("ws://localhost:12345/websocket", "",
        "http://localhost/")
    if err != nil {
        panic("Dial: " + err.Error())
    }
    go readFromServer(ws)
    time.Sleep(5e9)
    ws.Close()
}

func readFromServer(ws *websocket.Conn) {
    buf := make([]byte, 1000)
    for {
        if _, err := ws.Read(buf); err != nil {
            fmt.Printf("%s\n", err.Error())
            break
        }
    }
}
```

用 smtp 发送邮件

15.12 用 smtp 发送邮件

`smtp` 包实现了用于发送邮件的“简单邮件传输协议”（Simple Mail Transfer Protocol）。它有一个 `Client` 类型，代表一个连接到 SMTP 服务器的客户端：

- `Dial` 方法返回一个已连接到 SMTP 服务器的客户端 `Client`
- 设置 `Mail`（from，即发件人）和 `Rcpt`（to，即收件人）
- `Data` 方法返回一个用于写入数据的 `Writer`，这里利用 `buf.WriteTo(wc)` 写入

示例 15.26 smtp.go

```
package main

import (
    "bytes"
    "log"
    "net/smtp"
)

func main() {
    // Connect to the remote SMTP server.
    client, err := smtp.Dial("mail.example.com:25")
    if err != nil {
        log.Fatal(err)
    }
    // Set the sender and recipient.
    client.Mail("sender@example.org")
    client.Rcpt("recipient@example.net")
    // Send the email body.
    wc, err := client.Data()
    if err != nil {
        log.Fatal(err)
    }
    defer wc.Close()
    buf := bytes.NewBufferString("This is the email body.")
    if _, err = buf.WriteTo(wc); err != nil {
        log.Fatal(err)
    }
}
```

如果需要认证，或有多个收件人时，也可以用 `SendMail` 函数发送。它连接到地址为 `addr` 的服务器；如果可以，切换到 TLS（“传输层安全”加密和认证协议），并用 PLAIN 机制认证；然后以 `from` 作为发件人，`to` 作为收件人列表，`msg` 作为邮件内容，发出一封邮件：

```
func SendMail(addr string, a Auth, from string, to []string, msg []byte) error
```

示例 15.27 smtp_auth.go

```
package main

import (
```

```
    "log"
    "net/smtp"
)

func main() {
    // Set up authentication information.
    auth := smtp.PlainAuth(
        "",
        "user@example.com",
        "password",
        "mail.example.com",
    )
    // Connect to the server, authenticate, set the sender and recipient,
    // and send the email all in one step.
    err := smtp.SendMail(
        "mail.example.com:25",
        auth,
        "sender@example.org",
        []string{"recipient@example.net"},
        []byte("This is the email body."),
    )
    if err != nil {
        log.Fatal(err)
    }
}
```

常见的陷阱与错误

16.0 常见的陷阱与错误

在之前的内容中，有时候使用 `!!...!!` 标记警告go语言中的一些错误使用方式。当你在编程时候遇到的一个困难，可以确定本书特定的章节能找到类似的主题。为了方便起见，这里列出了一些常见陷阱，以便于你能发现更多的解释和例子：

- 永远不要使用形如 `var p*a` 声明变量，这会混淆指针声明和乘法运算（参考4.9小节）
- 永远不要在 `for` 循环自身中改变计数器变量（参考5.4小节）
- 永远不要在 `for-range` 循环中使用一个值去改变自身的值（参考5.4.4小节）
- 永远不要将 `goto` 和前置标签一起使用（参考5.6小节）
- 永远不要忘记在函数名（参考第6章）后加括号()，尤其调用一个对象的方法或者使用匿名函数启动一个协程时
- 永远不要使用 `new()` 一个map，一直使用make（参考第8章）
- 当为一个类型定义一个String()方法时，不要使用 `fmt.Print` 或者类似的代码（参考10.7小节）
- 永远不要忘记当终止缓存写入时，使用 `Flush` 函数（参考12.2.3小节）
- 永远不要忽略错误提示，忽略错误会导致程序崩溃（参考13.1小节）
- 不要使用全局变量或者共享内存，这会使并发执行的代码变得不安全（参考14.1小节）
- `println` 函数仅仅是用于调试的目的

最佳实践：对比以下使用方式：

- 使用正确的方式初始化一个元素是切片的映射，例如 `map[type]slice` （参考8.1.3小节）
- 一直使用逗号，ok或者checked形式作为类型断言（参考11.3小节）
- 使用一个工厂函数创建并初始化自己定义类型（参考10.2小节-18.4小节）
- 仅当一个结构体的方法想改变结构体时，使用结构体指针作为方法的接受者，否则使用一个结构体值类型10.6.3小节

本章主要汇总了go语言使用过程中最常见的错误和注意事项。在之前的章节已经涉及到了完整的示例和解释，你应该做的不仅仅是阅读这段的标题。

误用短声明导致变量覆盖

16.1 误用短声明导致变量覆盖

```
var remember bool = false
if something {
    remember := true //错误
}
// 使用remember
```

在此代码段中，`remember`变量永远不会在`if`语句外面变成`true`，如果`something`为`true`，由于使用了短声明`:=`，`if`语句内部的新变量`remember`将覆盖外面的`remember`变量，并且该变量的值为`true`，但是在`if`语句外面，变量`remember`的值变成了`false`，所以正确的写法应该是：

```
if something {
    remember = true
}
```

此类错误也容易在`for`循环中出现，尤其当函数返回一个具名变量时难于察觉，例如以下的代码段：

```
func shadow() (err error) {
    x, err := check1() // x是新创建变量，err是被赋值
    if err != nil {
        return // 正确返回err
    }
    if y, err := check2(x); err != nil { // y和if语句中err被创建
        return // if语句中的err覆盖外面的err，所以错误的返回nil!
    } else {
        fmt.Println(y)
    }
    return
}
```

误用字符串

16.2 误用字符串

当需要对一个字符串进行频繁的操作时，谨记在go语言中字符串是不可变的（类似java和c#）。使用诸如 `a += b` 形式连接字符串效率低下，尤其在一个循环内部使用这种形式。这会导致大量的内存开销和拷贝。**应该使用一个字符数组代替字符串，将字符串内容写入一个缓存中。** 例如以下的代码示例：

```
var b bytes.Buffer
...
for condition {
    b.WriteString(str) // 将字符串str写入缓存buffer
}
    return b.String()
```

注意：由于编译优化和依赖于使用缓存操作的字符串大小，当循环次数大于15时，效率才会更佳。

发生错误时使用 defer 关闭一个文件

16.3 发生错误时使用 defer 关闭一个文件

如果你在一个for循环内部处理一系列文件，你需要使用defer确保文件在处理完毕后被关闭，例如：

```
for _, file := range files {
    if f, err = os.Open(file); err != nil {
        return
    }
    // 这是错误的方式，当循环结束时文件没有关闭
    defer f.Close()
    // 对文件进行操作
    f.Process(data)
}
```

但是在循环结尾处的defer没有执行，所以文件一直没有关闭！垃圾回收机制可能会自动关闭文件，但是这会产生一个错误，更好的做法是：

```
for _, file := range files {
    if f, err = os.Open(file); err != nil {
        return
    }
    // 对文件进行操作
    f.Process(data)
    // 关闭文件
    f.Close()
}
```

defer仅在函数返回时才会执行，在循环的结尾或其他一些有限范围的代码内不会执行。

何时使用 new() 和 make()

16.4 何时使用 new() 和 make()

在第7.2.1小节和第10.2.2小节，我们已经讨论过此问题，并使用代码进行详细说明，观点如下：

```
- 切片、映射和通道，使用make
- 数组、结构体和所有的值类型，使用new
```

不需要将一个指向切片的指针传递给函数

16.5 不需要将一个指向切片的指针传递给函数

在第4.9小节，我们已经知道，切片实际是一个指向潜在数组的指针。我们常常需要把切片作为一个参数传递给函数是因为：实际就是传递一个指向变量的指针，在函数内可以改变这个变量，而不是传递数据的拷贝。

因此应该这样做：

```
func findBiggest( listOfNumbers []int ) int {}
```

而不是：

```
func findBiggest( listOfNumbers *[]int ) int {}
```

当切片作为参数传递时，切记不要解引用切片。

使用指针指向接口类型

16.6 使用指针指向接口类型

查看如下程序：`nexter` 是一个接口类型，并且定义了一个 `next()` 方法读取下一字节。函数 `nextFew` 将 `nexter` 接口作为参数并读取接下来的 `num` 个字节，并返回一个切片：这是正确做法。但是 `nextFew2` 使用一个指向 `nexter` 接口类型的指针作为参数传递给函数：当使用 `next()` 函数时，系统会给出一个编译错误：**n.next undefined (type *nexter has no field or method next)**（译者注：n.next未定义（*nexter类型没有next成员或next方法））

例 16.1 pointer_interface.go (不能通过编译):

```
package main
import (
    "fmt"
)
type nexter interface {
    next() byte
}
func nextFew1(n nexter, num int) []byte {
    var b []byte
    for i:=0; i < num; i++ {
        b[i] = n.next()
    }
    return b
}
func nextFew2(n *nexter, num int) []byte {
    var b []byte
    for i:=0; i < num; i++ {
        b[i] = n.next() // 编译错误:n.next未定义（*nexter类型没有next成员或next方法）
    }
    return b
}
func main() {
    fmt.Println("Hello World!")
}
```

永远不要使用一个指针指向一个接口类型，因为它已经是一个指针。

使用值类型时误用指针

16.7 使用值类型时误用指针

将一个值类型作为一个参数传递给函数或者作为一个方法的接收者，似乎是对内存的滥用，因为值类型一直是传递拷贝。但是另一方面，值类型的内存是在栈上分配，内存分配快速且开销不大。如果你传递一个指针，而不是一个值类型，go编译器大多数情况下会认为需要创建一个对象，并将对象移动到堆上，所以会导致额外的内存分配：因此当使用指针代替值类型作为参数传递时，我们没有任何收获。

误用协程和通道

16.8 误用协程和通道

由于教学需要和对协程的工作原理有一个直观的了解，在第14章使用了一些简单的算法，举例说明了协程和通道的使用，例如生产者或者迭代器。在实际应用中，你不需要并发执行，或者你不需要关注协程和通道的开销，在大多数情况下，通过栈传递参数会更有效率。

但是，如果你使用 `break` 、 `return` 或者 `panic` 去跳出一个循环，很有可能会导致内存溢出，因为协程正处理某些事情而被阻塞。在实际代码中，通常仅需写一个简单的过程式循环即可。**当且仅当代码中并发执行非常重要，才使用协程和通道。**

闭包和协程的使用

16.9 闭包和协程的使用

请看下面代码：

```
package main

import (
    "fmt"
    "time"
)

var values = [5]int{10, 11, 12, 13, 14}

func main() {
    // 版本A:
    for ix := range values { // ix是索引值
        func() {
            fmt.Print(ix, " ")
        }() // 调用闭包打印每个索引值
    }
    fmt.Println()
    // 版本B: 和A版本类似，但是通过调用闭包作为一个协程
    for ix := range values {
        go func() {
            fmt.Print(ix, " ")
        }()
    }
    fmt.Println()
    time.Sleep(5e9)
    // 版本C: 正确的处理方式
    for ix := range values {
        go func(ix interface{}) {
            fmt.Print(ix, " ")
        }(ix)
    }
    fmt.Println()
    time.Sleep(5e9)
    // 版本D: 输出值:
    for ix := range values {
        val := values[ix]
        go func() {
            fmt.Print(val, " ")
        }()
    }
    time.Sleep(1e9)
}
```

输出：

```
0 1 2 3 4

4 4 4 4 4
```

```
1 0 3 4 2

10 11 12 13 14
```

版本A调用闭包5次打印每个索引值，版本B也做相同的事，但是通过协程调用每个闭包。按理说这将执行得更快，因为闭包是并发执行的。如果我们阻塞足够多的时间，让所有协程执行完毕，版本B的输出是： `4 4 4 4 4` 。为什么会这样？在版本B的循环中， `ix` 变量实际是一个单变量，表示每个数组元素的索引值。因为这些闭包都只绑定到一个变量，这是一个比较好的方式，当你运行这段代码时，你将看见每次循环都打印最后一个索引值 `4` ，而不是每个元素的索引值。因为协程可能在循环结束后还没有开始执行，而此时 `ix` 值是 `4` 。

版本C的循环写法才是正确的：调用每个闭包时将 `ix` 作为参数传递给闭包。 `ix` 在每次循环时都被重新赋值，并将每个协程的 `ix` 放置在栈中，所以当协程最终被执行时，每个索引值对协程都是可用的。注意这里的输出可能是 `0 2 1 3 4` 或者 `0 3 1 2 4` 或者其他类似的序列，这主要取决于每个协程何时开始被执行。

在版本D中，我们输出这个数组的值，为什么版本B不能而版本D可以呢？

因为版本D中的变量声明是在循环体内部，所以在每次循环时，这些变量相互之间是不共享的，所以这些变量可以单独的被每个闭包使用。

糟糕的错误处理

16.10 糟糕的错误处理

译者注：该小结关于错误处理的观点，译者并不完全赞同，关于本小结的部分想法请参考关于16.10.2小节错误处理的一些见解

依附于第13章模式的描述和第17.1小节与第17.2.4小节的总结。

16.10.1 不要使用布尔值：

像下面代码一样，创建一个布尔型变量用于测试错误条件是多余的：

```
var good bool
    // 测试一个错误，`good`被赋为`true`或者`false`
    if !good {
        return errors.New("things aren' t good")
    }
```

立即检测一个错误：

```
... err1 := api.Func1()
if err1 != nil { … }
```

16.10.2 避免错误检测使代码变得混乱：

避免写出这样的代码：

```
... err1 := api.Func1()
if err1 != nil {
    fmt.Println("err: " + err.Error())
    return
}
err2 := api.Func2()
if err2 != nil {
...
    return
}
```

首先，包括在一个初始化的 `if` 语句中对函数的调用。但即使代码中到处都是以 `if` 语句的形式通知错误（通过打印错误信息）。通过这种方式，很难分辨什么是正常的程序逻辑，什么是错误检测或错误通知。还需注意的是，大部分代码都是致力于错误的检测。通常解决此问题的好办法是尽可能以闭包的形式封装你的错误检测，例如下面的代码：

```
func httpRequestHandler(w http.ResponseWriter, req *http.Request) {
    err := func () error {
        if req.Method != "GET" {
            return errors.New("expected GET")
        }
        if input := parseInput(req); input != "command" {
            return errors.New("malformed command")
        }
        // 可以在此进行其他的错误检测
```

```
    } ()

        if err != nil {
            w.WriteHeader(400)
            io.WriteString(w, err)
            return
        }
        doSomething() ...
```

这种方法可以很容易分辨出错误检测、错误通知和正常的程序逻辑（更详细的方式参考第13.5小节）。

在开始阅读第17章前，先回答下列2个问题：

- 问题 16.1：总结你能记住的所有关于 `,ok` 模式的情况。
- 问题 16.2：总结你能记住的所有关于 `defer` 模式的情况。

模式

逗号 ok 模式

17.1 逗号 ok 模式

在学习本书第二部分和第三部分时，我们经常在一个表达式返回2个参数时使用这种模式： `,ok` ，第一个参数是一个值或者 `nil` ，第二个参数是 `true` / `false` 或者一个错误 `error` 。在一个需要赋值的 `if` 条件语句中，使用这种模式去检测第二个参数值会让代码显得优雅简洁。这种模式在go语言编码规范中非常重要。下面总结了所有使用这种模式的例子：

（1）在函数返回时检测错误（参考第5.2小节）：

```
value, err := pack1.Func1(param1)

if err != nil {
    fmt.Printf("Error %s in pack1.Func1 with parameter %v", err.Error(), param1)
    return err
}

// 函数Func1没有错误:
Process(value)

e.g.: os.Open(file) strconv.Atoi(str)
```

这段代码中的函数将错误返回给它的调用者，当函数执行成功时，返回的错误是 `nil` ，所以使用这种写法：

```
func SomeFunc() error {
    ...
    if value, err := pack1.Func1(param1); err != nil {
        ...
        return err
    }
    ...
    return nil
}
```

这种模式也常用于通过 `defer` 使程序从 `panic` 中恢复执行（参考第17.2（4）小节）。

要实现简洁的错误检测代码，更好的方式是使用闭包，参考第16.10.2小节

（2）检测映射中是否存在一个键值（参考第8.2小节）： `key1` 在映射 `map1` 中是否有值？

```
if value, isPresent = map1[key1]; isPresent {
        Process(value)
}
// key1不存在
...
```

（3）检测一个接口类型变量 `varI` 是否包含了类型 `T` ：类型断言（参考第11.3小节）：

```
if value, ok := varI.(T); ok {
    Process(value)
}
// 接口类型varI没有包含类型T
```

（4）检测一个通道 `ch` 是否关闭（参考第14.3小节）：

```
for input := range ch {
    Process(input)
}
```

或者:

```
for {
    if input, open := <-ch; !open {
        break // 通道是关闭的
    }
    Process(input)
}
```

defer 模式

17.2 defer 模式

使用 `defer` 可以确保资源不再需要时，都会被恰当地关闭或归还到“池子”中。更重要的一点是，它可以恢复 panic。

1. 关闭一个文件流：（见 12.7节）

```
// 先打开一个文件 f
defer f.Close()
```

2. 解锁一个被锁定的资源（ `mutex` ）：（见 9.3节）

```
mu.Lock()
defer mu.Unlock()
```

3. 关闭一个通道（如有必要）：

```
ch := make(chan float64)
defer close(ch)
```

也可以是两个通道：

```
answerα, answerβ := make(chan int), make(chan int)
defer func() { close(answerα); close(answerβ) }()
```

4. 从 panic 恢复：（见 13.3节）

```
defer func() {
 if err := recover(); err != nil {
     log.Printf("run time panic: %v", err)
 }
}()
```

5. 停止一个计时器：（见 14.5节）

```
tick1 := time.NewTicker(updateInterval)
defer tick1.Stop()
```

6. 释放一个进程 p：（见 13.6节）

```
p, err := os.StartProcess(…, …, …)
defer p.Release()
```

7. 停止 CPU 性能分析并立即写入：（见 13.10节）

```
pprof.StartCPUProfile(f)
defer pprof.StopCPUProfile()
```

当然 `defer` 也可以在打印报表时避免忘记输出页脚。

可见性模式

17.3 可见性模式

我们在 4.2.1节 见过简单地使用可见性规则控制对类型成员的访问，他们可以是 Go 变量或函数。10.2.1节 展示了如何在单独的包中定义类型时，强制使用工厂函数。

运算符模式和接口

17.4 运算符模式和接口

运算符是一元或二元函数，它返回一个新对象而不修改其参数，类似 C++ 中的 `+` 和 `*`，特殊的中缀运算符（`+`，`-`，`*` 等）可以被重载以支持类似数学运算的语法。但除了一些特殊情况，Go 语言并不支持运算符重载：为了克服该限制，运算符必须由函数来模拟。既然 Go 同时支持面向过程和面向对象编程，我们有两种选择：

17.4.1 函数作为运算符

运算符由包级别的函数实现，以操作一个或两个参数，并返回一个新对象。函数针对要操作的对象，在专门的包中实现。例如，假设要在包 `matrix` 中实现矩阵操作，就会包含 `Add()` 用于矩阵相加，`Mult()` 用于矩阵相乘，他们都会返回一个矩阵。这两个函数通过包名来调用，因此可以创造出如下形式的表达式：

```
m := matrix.Add(m1, matrix.Mult(m2, m3))
```

如果我们想在这些运算中区分不同类型的矩阵（稀疏或稠密），由于没有函数重载，我们不得不给函数起不同的名称，例如：

```
func addSparseToDense (a *sparseMatrix, b *denseMatrix) *denseMatrix
func addDenseToDense (a *denseMatrix, b *denseMatrix) *denseMatrix
func addSparseToSparse (a *sparseMatrix, b *sparseMatrix) *sparseMatrix
```

这可不怎么优雅，我们能选择的最佳方案是将它们隐藏起来，作为包的私有函数，并暴露单一的 `Add()` 函数作为公共 API。可以在嵌套的 `switch` 断言中测试类型，以便在任何支持的参数组合上执行操作：

```
func Add(a Matrix, b Matrix) Matrix {
    switch a.(type) {
    case sparseMatrix:
        switch b.(type) {
        case sparseMatrix:
            return addSparseToSparse(a.(sparseMatrix), b.(sparseMatrix))
        case denseMatrix:
            return addSparseToDense(a.(sparseMatrix), b.(denseMatrix))
        ...
        }
    default:
        // 不支持的参数
        ...
    }
}
```

然而，更优雅和优选的方案是将运算符作为方法实现，标准库中到处都运用了这种做法。有关 Ryanne Dolan 实现的线性代数包的更详细信息，可以在 https://github.com/skelterjohn/go.matrix 找到。

17.4.2 方法作为运算符

根据接收者类型不同，可以区分不同的方法。因此我们可以为每种类型简单地定义 `Add` 方法，来代替使用多个函数名称：

```
func (a *sparseMatrix) Add(b Matrix) Matrix
func (a *denseMatrix) Add(b Matrix) Matrix
```

每个方法都返回一个新对象，成为下一个方法调用的接收者，因此我们可以使用*链式调用*表达式：

```
m := m1.Mult(m2).Add(m3)
```

比上一节面向过程的形式更简洁。

正确的实现同样可以基于类型，通过 `switch` 类型断言在运行时确定：

```
func (a *sparseMatrix) Add(b Matrix) Matrix {
    switch b.(type) {
    case sparseMatrix:
        return addSparseToSparse(a.(sparseMatrix), b.(sparseMatrix))
    case denseMatrix:
        return addSparseToDense(a.(sparseMatrix), b.(denseMatrix))
    ...
    default:
        // 不支持的参数
        ...
    }
}
```

再次地，这比上一节嵌套的 `switch` 更简单。

17.4.3 使用接口

当在不同类型上执行相同的方法时，创建一个通用化的接口以实现多态的想法，就会自然产生。

例如定义一个代数 `Algebraic` 接口：

```
type Algebraic interface {
    Add(b Algebraic) Algebraic
    Min(b Algebraic) Algebraic
    Mult(b Algebraic) Algebraic
    ...
    Elements()
}
```

然后为我们的 `matrix` 类型定义 `Add()`，`Min()`，`Mult()`，.....等方法。

每种实现上述 `Algebraic` 接口类型的方法都可以链式调用。每个方法实现都应基于参数类型，使用 `switch` 类型断言来提供优化过的实现。另外，应该为仅依赖于接口的方法，指定一个默认处理分支：

```
func (a *denseMatrix) Add(b Algebraic) Algebraic {
    switch b.(type) {
    case sparseMatrix:
        return addDenseToSparse(a, b.(sparseMatrix))
    ...
    default:
        for x in range b.Elements() ...
    }
}
```

如果一个通用的功能无法仅使用接口方法来实现，你可能正在处理两个不怎么相似的类型，此时应该放弃这种运算符模式。例如，如果 `a` 是一个集合而 `b` 是一个矩阵，那么编写 `a.Add(b)` 没有意义。就集合和矩阵运算而言，很难实

现一个通用的 `a.Add(b)` 方法。遇到这种情况，把包拆分成两个，然后提供单独的 `AlgebraicSet` 和 `AlgebraicMatrix` 接口。

出于性能考虑的实用代码片段

字符串

18.1 字符串

（1）如何修改字符串中的一个字符：

```
str:="hello"
c:=[]byte(str)
c[0]='c'
s2:= string(c) // s2 == "cello"
```

（2）如何获取字符串的子串：

```
substr := str[n:m]
```

（3）如何使用 `for` 或者 `for-range` 遍历一个字符串：

```
// gives only the bytes:
for i:=0; i < len(str); i++ {
… = str[i]
}
// gives the Unicode characters:
for ix, ch := range str {
…
}
```

（4）如何获取一个字符串的字节数： `len(str)`

如何获取一个字符串的字符数：

最快速： `utf8.RuneCountInString(str)`

`len([]rune(str))`

（5）如何连接字符串：

最快速：
`with a bytes.Buffer` （参考章节7.2）

`Strings.Join()` （参考章节4.7）

使用 `+=` ：

```
str1 := "Hello "
str2 := "World!"
str1 += str2 //str1 == "Hello World!"
```

数组和切片

18.2 数组和切片

创建：

```
arr1 := new([len]type)
```

```
slice1 := make([]type, len)
```

初始化：

```
arr1 := [...]type{i1, i2, i3, i4, i5}
```

```
arrKeyValue := [len]type{i1: val1, i2: val2}
```

```
var slice1 []type = arr1[start:end]
```

（1）如何截断数组或者切片的最后一个元素：

```
line = line[:len(line)-1]
```

（2）如何使用 `for` 或者 `for-range` 遍历一个数组（或者切片）：

```
for i:=0; i < len(arr); i++ {
… = arr[i]
}
for ix, value := range arr {
…
}
```

（3）如何在一个二维数组或者切片 `arr2Dim` 中查找一个指定值 `V` ：

```
found := false
Found: for row := range arr2Dim {
    for column := range arr2Dim[row] {
        if arr2Dim[row][column] == V{
            found = true
            break Found
        }
    }
}
```

映射

18.3 映射

创建： `map1 := make(map[keytype]valuetype)`

初始化： `map1 := map[string]int{"one": 1, "two": 2}`

（1）如何使用 `for` 或者 `for-range` 遍历一个映射：

```
for key, value := range map1 {
...
}
```

（2）如何在一个映射中检测键 `key1` 是否存在：

`val1, isPresent = map1[key1]`

返回值：键 `key1` 对应的值或者 `0` ， `true` 或者 `false`

（3）如何在映射中删除一个键：

`delete(map1, key1)`

结构体

18.4 结构体

创建：

```
type struct1 struct {
    field1 type1
    field2 type2
    …
}
ms := new(struct1)
```

初始化：

```
ms := &struct1{10, 15.5, "Chris"}
```

当结构体的命名以大写字母开头时，该结构体在包外可见。
通常情况下，为每个结构体定义一个构建函数，并推荐使用构建函数初始化结构体（参考例10.2）：

```
ms := Newstruct1{10, 15.5, "Chris"}
func Newstruct1(n int, f float32, name string) *struct1 {
    return &struct1{n, f, name}
}
```

接口

18.5 接口

（1）如何检测一个值 `v` 是否实现了接口 `Stringer` ：

```
if v, ok := v.(Stringer); ok {
    fmt.Printf("implements String(): %s\n", v.String())
}
```

（2）如何使用接口实现一个类型分类函数：

```
func classifier(items ...interface{}) {
    for i, x := range items {
        switch x.(type) {
        case bool:
            fmt.Printf("param #%d is a bool\n", i)
        case float64:
            fmt.Printf("param #%d is a float64\n", i)
        case int, int64:
            fmt.Printf("param #%d is an int\n", i)
        case nil:
            fmt.Printf("param #%d is nil\n", i)
        case string:
            fmt.Printf("param #%d is a string\n", i)
        default:
            fmt.Printf("param #%d' s type is unknown\n", i)
        }
    }
}
```

函数

18.6 函数

如何使用内建函数 `recover` 终止 `panic` 过程（参考章节13.3）：

```
func protect(g func()) {
    defer func() {
        log.Println("done")
        // Println executes normally even if there is a panic
        if x := recover(); x != nil {
            log.Printf("run time panic: %v", x)
        }
    }()
    log.Println("start")
    g()
}
```

文件

18.7 文件

（1）如何打开一个文件并读取：

```
file, err := os.Open("input.dat")
  if err != nil {
    fmt.Printf("An error occurred on opening the inputfile\n" +
      "Does the file exist?\n" +
      "Have you got acces to it?\n")
    return
  }
  defer file.Close()
  iReader := bufio.NewReader(file)
  for {
    str, err := iReader.ReadString('\n')
    if err != nil {
      return // error or EOF
    }
    fmt.Printf("The input was: %s", str)
  }
```

（2）如何通过切片读写文件：

```
func cat(f *file.File) {
  const NBUF = 512
  var buf [NBUF]byte
  for {
    switch nr, er := f.Read(buf[:]); true {
    case nr < 0:
      fmt.Fprintf(os.Stderr, "cat: error reading from %s: %s\n",
        f.String(), er.String())
      os.Exit(1)
    case nr == 0: // EOF
      return
    case nr > 0:
      if nw, ew := file.Stdout.Write(buf[0:nr]); nw != nr {
        fmt.Fprintf(os.Stderr, "cat: error writing from %s: %s\n",
          f.String(), ew.String())
      }
    }
  }
}
```

协程（goroutine）与通道（channel）

18.8 协程（goroutine）与通道（channel）

出于性能考虑的建议：

实践经验表明，为了使并行运算获得高于串行运算的效率，在协程内部完成的工作量，必须远远高于协程的创建和相互来回通信的开销。

1 出于性能考虑建议使用带缓存的通道：

使用带缓存的通道可以很轻易成倍提高它的吞吐量，某些场景其性能可以提高至10倍甚至更多。通过调整通道的容量，甚至可以尝试着更进一步的优化其性能。

2 限制一个通道的数据数量并将它们封装成一个数组：

如果使用通道传递大量单独的数据，那么通道将变成性能瓶颈。然而，将数据块打包封装成数组，在接收端解压数据时，性能可以提高至10倍。

创建： `ch := make(chan type,buf)`

（1）如何使用 `for` 或者 `for-range` 遍历一个通道：

```
for v := range ch {
    // do something with v
}
```

（2）如何检测一个通道 `ch` 是否关闭：

```
//read channel until it closes or error-condition
for {
    if input, open := <-ch; !open {
        break
    }
    fmt.Printf("%s", input)
}
```

或者使用（1）自动检测。

（3）如何通过一个通道让主程序等待直到协程完成：

（信号量模式）：

```
ch := make(chan int) // Allocate a channel.
// Start something in a goroutine; when it completes, signal on the channel.
go func() {
    // doSomething
    ch <- 1 // Send a signal; value does not matter.
}()
doSomethingElseForAWhile()
<-ch // Wait for goroutine to finish; discard sent value.
```

如果希望程序一直阻塞，在匿名函数中省略 `ch <- 1` 即可。

（4）通道的工厂模板：以下函数是一个通道工厂，启动一个匿名函数作为协程以生产通道：

```
func pump() chan int {
    ch := make(chan int)
    go func() {
        for i := 0; ; i++ {
            ch <- i
        }
    }()
    return ch
}
```

（5）通道迭代器模板：

（6）如何限制并发处理请求的数量：参考章节14.11

（7）如何在多核CPU上实现并行计算：参考章节14.13

（8）如何终止一个协程：`runtime.Goexit()`

（9）简单的超时模板：

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1e9) // one second
    timeout <- true
}()
select {
    case <-ch:
    // a read from ch has occurred
    case <-timeout:
    // the read from ch has timed out
}
```

（10）如何使用输入通道和输出通道代替锁：

```
func Worker(in, out chan *Task) {
    for {
        t := <-in
        process(t)
        out <- t
    }
}
```

（11）如何在同步调用运行时间过长时将之丢弃：参考章节14.5 第二个变体

（12）如何在通道中使用计时器和定时器：参考章节14.5

（13）典型的服务器后端模型：参考章节14.4

网络和网页应用

18.9 网络和网页应用

18.9.1 模板：

制作、解析并使模板生效：

```
var strTempl = template.Must(template.New("TName").Parse(strTemplateHTML))
```

在网页应用中使用HTML过滤器过滤HTML特殊字符：

`{{html .}}` 或者通过一个字段 `FieldName {{ .FieldName |html }}`

使用缓存模板（参考章节15.7）

其他

18.10 其他

如何在程序出错时终止程序：

```
if err != nil {
    fmt.Printf("Program stopping with error %v", err)
    os.Exit(1)
}
```

或者：

```
if err != nil {
    panic("ERROR occurred: " + err.Error())
}
```

出于性能考虑的最佳实践和建议

18.11 出于性能考虑的最佳实践和建议

（1）尽可能的使用 `:=` 去初始化声明一个变量（在函数内部）；

（2）尽可能的使用字符代替字符串；

（3）尽可能的使用切片代替数组；

（4）尽可能的使用数组和切片代替映射（详见参考文献15）；

（5）如果只想获取切片中某项值，不需要值的索引，尽可能的使用 `for range` 去遍历切片，这比必须查询切片中的每个元素要快一些；

（6）当数组元素是稀疏的（例如有很多 `0` 值或者空值 `nil` ），使用映射会降低内存消耗；

（7）初始化映射时指定其容量；

（8）当定义一个方法时，使用指针类型作为方法的接受者；

（9）在代码中使用常量或者标志提取常量的值；

（10）尽可能在需要分配大量内存时使用缓存；

（11）使用缓存模板（参考章节15.7）。

构建一个完整的应用程序

简介

19.1 简介

由于 web 无处不在，本章我们将开发一个完整的程序： `goto` ，它是一个 web 缩短网址应用程序。示例来自 Andrew Gerrand 的讲座（见参考资料 22）。我们将把项目分成 3 个阶段，每一个都会比之前阶段包含更多的功能，并逐渐展示更多 Go 语言中的特性。我们会大量使用在 15章 所学的网页应用程序的知识。

版本 1： 利用映射和结构体，与 `sync` 包的 `Mutex` 一起使用，以及一个结构体工厂。

版本 2： 数据以 `gob` 格式写入文件以实现持久化。

版本 3： 利用协程和通道重写应用（见 14章）。

版本 4： 如果我们要使用 json 格式的文件该如何修改？

版本 5： 用 rpc 协议实现的分布式版本。

由于代码变更频繁，不会展示在此处，仅给出访问地址。

短网址项目简介

19.2 短网址项目简介

你肯定知道有些浏览器中的地址（称为 URL）非常长且/或复杂，在网上有一些将他们转换成简短 URL 来使用的服务。我们的项目与此类似：它是具有 2 个功能的 *web 服务*（web service）：

添加 (Add)

给定一个较长的 URL，会将其转换成较短的版本，例如：

```
http://maps.google.com/maps?f=q&source=s_q&hl=en&geocode=&q=tokyo&sll=37.0625,-95.677068&sspn=68.684234,65.566406&ie=UTF8&hq=&hnear=Tokyo,+Japan&t=h&z=9
```

- (A) 转变为：`http://goto/UrcGq`
- (B) 并保存这对数据

重定向 (Redirect)

短网址被请求时，会把用户重定向到原始的长 URL。因此如果你在浏览器输入网址 (B)，会被重定向到页面 (A)。

数据结构

版本 1 - 数据结构和前端界面

第 1 个版本的代码 *goto_v1* 见 goto_v1。

19.3 数据结构

（本节代码见 goto_v1/store.go。）

当程序运行在生产环境时，会收到很多短网址的请求，同时会有一些将长 URL 转换成短 URL 的请求。我们的程序要以什么样的结构存储这些数据呢？19.2 节 中 (A) 和 (B) 两种 URL 都是字符串，此外，它们相互关联：给定键 (B) 能获取到值 (A)，他们互相*映射*（map）。要将数据存储在内存中，我们需要这种结构，它们几乎存在于所有的编程语言中，只是名称有所不同，例如“哈希表”或“字典”等。

Go 语言就有这种内建的映射（map）：`map[string]string`。

键的类型写在 `[` 和 `]` 之间，紧接着是值的类型。有关映射的所有知识详见 8 章。为特定类型指定一个别名在严谨的程序中非常实用。Go 语言中通过关键字 `type` 来定义，因此有定义：

```
type URLStore map[string]string
```

它从短 URL 映射到长 URL，两者都是字符串。

要创建那种类型的变量，并命名为 m，使用：

```
m := make(URLStore)
```

假设 *http://goto/a* 映射到 *http://google.com/*，我们要把它们存储到 m 中，可以用如下语句：

```
m["a"] = "http://google.com/"
```

（键只是 *http://goto/* 的后缀，其前缀总是不变的。）

要获得给定 “a” 对应的长 URL，可以这么写：

```
url := m["a"]
```

此时 `url` 的值等于 `http://google.com/`。

注意，使用了 `:=` 就不需要指明 url 的类型为 `string`，编译器会从右侧的值中推断出来。

使程序线程安全

这里，变量 `URLStore` 是中心化的内存存储。当收到网络流量时，会有很多 `Redirect` 服务的请求。这些请求其实只涉及读操作：以给定的短 URL 作为键，返回对应的长 URL 的值。然而，对 `Add` 服务的请求则大不相同，它们会更改 `URLStore`，添加新的键值对。当在瞬间收到大量更新请求时，可能会产生如下问题：添加操作可能被另一个同类请求打断，写入的长 URL 值可能会丢失；另外，读取和更改同时进行，导致可能读到脏数据。代码中的 map 并不保证当开始更新数据时，会彻底阻止另一个更新操作的启动。也就是说，map 不是线程安全的，goto 会并发地为很多请求提供服务。因此必须

使 `URLStore` 是线程安全的，以便可以从不同的线程访问它。最简单和经典的方法是为其增加一个锁，它是 Go 标准库 `sync` 包中的 `Mutex` 类型，必须导入到我们的代码中（关于锁详见 9.3 节）。

现在，我们把 `URLStore` 类型的定义更改为一个结构体（就是字段的集合，类似 C 或 Java，10 章介绍了结构体），它含有两个字段：`map` 和 `sync` 包的 `RWMutex`：

```
import "sync"
type URLStore struct {
    urls map[string]string        // map from short to long URLs
    mu sync.RWMutex
}
```

`RWMutex` 有两种锁：分别对应读和写。多个客户端可以同时设置读锁，但只有一个客户端可以设置写锁（以排除所有的读锁），有效地串行化变更，使他们按顺序生效。

我们将在 `Get` 函数中实现 `Redirect` 服务的读请求，在 `Set` 函数中实现 `Add` 服务的写请求。`Get` 函数类似下面这样：

```
func (s *URLStore) Get(key string) string {
    s.mu.RLock()
    url := s.urls[key]
    s.mu.RUnlock()
    return url
}
```

函数按照键（短 URL）返回对应映射后的 URL。它所处理的变量是指针类型（见 4.9 节），指向 `URLStore`。但在读取值之前，先用 `s.mu.RLock()` 放置一个读锁，这样就不会有更新操作妨碍读取。数据读取后撤销锁定，以便挂起的更新操作可以开始。如果键不存在于 map 中会怎样？会返回字符串的零值（空字符串）。注意点号（`.`）类似面向对象的语言：在 `s` 的 `mu` 字段上调用方法 `RLock()`。

`Set` 函数同时需要 URL 的键值对，且必须放置写锁 `Lock()` 来排除同一时刻任何其他更新操作。函数返回布尔值 `true` 或 `false` 来表示 `Set` 操作是否成功：

```
func (s *URLStore) Set(key, url string) bool {
    s.mu.Lock()
    _, present := s.urls[key]
    if present {
        s.mu.Unlock()
        return false
    }
    s.urls[key] = url
    s.mu.Unlock()
    return true
}
```

形式 `_, present := s.urls[key]` 可以测试 map 中是否已经包含该键，包含则 `present` 为 `true`，否则为 `false`。这种形式称为“逗号 ok 模式”，在 Go 代码中会频繁出现。如果键已存在，`Set` 函数直接返回布尔值 `false`，map 不会被更新（这样可以保证短 URL 不会重复）。如果键不存在，把它加入 map 中并返回 `true`。左侧 `_` 是一个值的占位符，赋值给 `_` 来表明我们不会使用它。注意在更新后尽早调用 `Unlock()` 来释放对 `URLStore` 的锁定。

使用 defer 简化代码

目前代码还比较简单，容易记得操作完成后调用 `Unlock()` 解锁。然而在代码更复杂时很容易忘记解锁，或者放置在错误的位置，往往导致问题很难追踪。对于这种情况 Go 提供了一个特殊关键字 `defer`（见 6.4 节）。在本例中，可以在 `Lock` 之后立即示意 `Unlock`，不过其效果是 `Unlock()` 只会在函数返回之前被调用。

`Get` 可以简化成以下代码（我们消除了本地变量 `url` ）：

```
func (s *URLStore) Get(key string) string {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return s.urls[key]
}
```

`Set` 的逻辑在某种程度上也变得清晰了（我们不用再考虑解锁的事了）：

```
func (s *URLStore) Set(key, url string) bool {
    s.mu.Lock()
    defer s.mu.Unlock()
    _, present := s.urls[key]
    if present {
        return false
    }
    s.urls[key] = url
    return true
}
```

URLStore 工厂函数

`URLStore` 结构体中包含 map 类型的字段，使用前必须先用 `make` 初始化。在 Go 中创建一个结构体实例，一般是通过定义一个前缀为 `New` ，能返回该类型已初始化实例的函数（通常是指向实例的指针）。

```
func NewURLStore() *URLStore {
    return &URLStore{ urls: make(map[string]string) }
}
```

在 `return` 语句中，创建了 `URLStore` 字面量实例，其中包含初始化了的 map 映射。锁无需特别指明初始化，这是 Go 创建结构体实例的惯例。 `&` 是取址运算符，它将我们要返回的内容变成指针，因为 `NewURLStore` 返回类型是 `*URLStore` 。然后调用该函数来创建 `URLStore` 变量：

```
var store = NewURLStore()
```

使用 URLStore

要新增一对短/长 URL 到 map 中，我们只需调用 s 上的 `Set` 方法，由于返回布尔值，可以把它包裹在 `if` 语句中：

```
if s.Set("a", "http://google.com") {
    // 成功
}
```

要获取给定短 URL 对应的长 URL，调用 s 上的 `Get` 方法，将返回值放入变量 `url` ：

```
if url := s.Get("a"); url != "" {
    // 重定向到 url
} else {
    // 键未找到
}
```

这里我们利用 Go 语言 `if` 语句的特性，可以在起始部分、条件判断前放置初始化语句。另外还需要一个 `Count` 方法以获取 map 中键值对的数量，可以使用内建的 `len` 函数：

```
func (s *URLStore) Count() int {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return len(s.urls)
}
```

如何根据给定的长 URL 计算出短 URL 呢？为此我们创建一个函数 `genKey(n int) string {…}`，将 `s.Count()` 的当前值作为其整型参数传入。（具体算法并不重要，示例代码可以在 key.go 找到。）

现在，我们可以创建一个 `Put` 方法，接收一个长 URL，用 `genKey` 生成其短 URL 键，调用 `Set` 方法在此键下存储长 URL 数据，然后返回这个键：

```
func (s *URLStore) Put(url string) string {
    for {
        key := genKey(s.Count())
        if s.Set(key, url) {
            return key
        }
    }
    // shouldn't get here
    return ""
}
```

`for` 循环会一直尝试调用 `Set` 直到成功为止（意味着生成了一个尚未存在的短网址）。现在我们定义好了数据存储，以及配套的可工作的函数（见代码 store.go）。但这本身并不能完成任务，我们还需要开发 web 服务器以交付 `Add` 和 `Redirect` 服务。

用户界面：web 服务端

19.4 用户界面：web 服务端

（本节代码见 goto_v1/main.go。）

我们尚未编写启动程序的必要函数。它们（总是）类似 C，C++ 或 Java 中的 `main()` 函数，我们的 web 服务器由它启动，例如用如下命令在本地 8080 端口启动 web 服务器：

```
http.ListenAndServe(":8080", nil)
```

（web 服务器的功能来自于 `http` 包，15 章 做了深入介绍）。web 服务器会在一个无限循环中监听到来的请求，但我们必须定义针对这些请求，服务器该如何响应。可以用被称为 HTTP 处理器的 `HandleFunc` 函数来办到，例如代码：

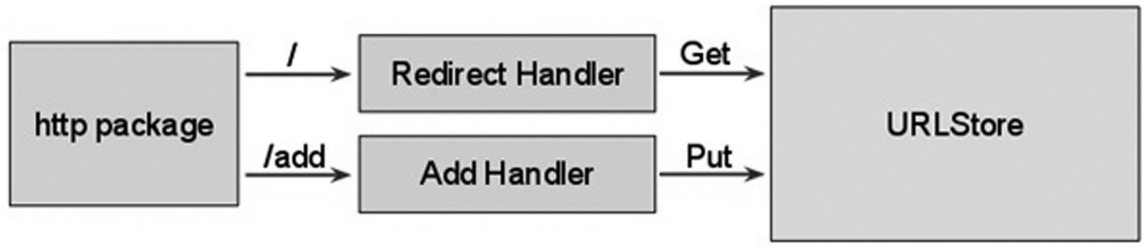
```
http.HandleFunc("/add", Add)
```

如此，每个以 `/add` 结尾的请求都会调用 `Add` 函数（尚未完成）。

程序有两个 HTTP 处理器：

- `Redirect`，用于对短 URL 重定向
- `Add`，用于处理新提交的 URL

示意图：



最简单的 `main()` 函数类似这样：

```
func main() {
    http.HandleFunc("/", Redirect)
    http.HandleFunc("/add", Add)
    http.ListenAndServe(":8080", nil)
}
```

对 `/add` 的请求由 `Add` 处理器处理，所有其他请求会被 `Redirect` 处理器处理。处理函数从到来的请求（一个类型为 `*http.Request` 的变量）中获取信息，然后产生响应并写入 `http.ResponseWriter` 类型变量 `w`。

`Add` 函数必须做的事有：

1. 读取长 URL，即：用 `r.FormValue("url")` 从 HTML 表单提交的 HTTP 请求中读取 URL
2. 使用 store 上的 `Put` 方法存储长 URL
3. 将对应的短 URL 发送给用户

每个需求都转化为一行代码：

```
func Add(w http.ResponseWriter, r *http.Request) {
    url := r.FormValue("url")
    key := store.Put(url)
    fmt.Fprintf(w, "http://localhost:8080/%s", key)
}
```

这里 `fmt` 包的 `Fprintf` 函数用来替换字符串中的关键字 `%s`，然后将结果作为响应发送回客户端。注意 `Fprintf` 把数据写到了 `ResponseWriter` 中，其实 `Fprintf` 可以将数据写到任何实现了 `io.Writer` 的数据结构，即该结构实现了 `Write` 方法。Go 中 `io.Writer` 称为接口，可见 `Fprintf` 利用接口变得十分通用，可以对很多不同的类型写入数据。Go 中接口的使用十分普遍，它使代码更通用（见 11 章）。

还需要一个表单，仍然可以用 `Fprintf` 来输出，这次将常量写入 `w`。让我们来修改 `Add`，当未指定 URL 时显示 HTML 表单：

```
func Add(w http.ResponseWriter, r *http.Request) {
    url := r.FormValue("url")
    if url == "" {
        fmt.Fprint(w, AddForm)
        return
    }
    key := store.Put(url)
    fmt.Fprintf(w, "http://localhost:8080/%s", key)
}

const AddForm = `
<form method="POST" action="/add">
URL: <input type="text" name="url">
<input type="submit" value="Add">
</form>
`
```

在那种情况下，发送字符串常量 `AddForm` 到客户端，它是 html 表单，包含一个 `url` 输入域和一个提交按钮，点击后发送 POST 请求到 `/add`。这样 `Add` 处理函数被再次调用，此时 `url` 的值来自文本域。（ `` ` `` 用来创建原始字符串，否则按惯例 “”` 将成为字符串边界。）

`Redirect` 函数在 HTTP 请求路径中找到键（短 URL 的键是请求路径去除首字符，在 Go 中可以写为 `[1:]`。例如请求 “/abc”，键就是 “abc”），用 `Get` 函数从 `store` 检索到对应的长 URL，对用户发送 HTTP 重定向。如果没找到 URL，发送 404 “Not Found” 错误取而代之：

```
func Redirect(w http.ResponseWriter, r *http.Request) {
    key := r.URL.Path[1:]
    url := store.Get(key)
    if url == "" {
        http.NotFound(w, r)
        return
    }
    http.Redirect(w, r, url, http.StatusFound)
}
```

（ `http.NotFound` 和 `http.Redirect` 是发送通用 HTTP 响应的工具函数。）

我们已经完整地遍历了 goto_v1 的代码。

编译和运行

可执行程序已包含在示例代码下，如果你想立即测试可以跳过本节。其中包含 3 个 go 源文件和一个 Makefile 文件，通过它应用可以被编译和链接，只须如下操作：

- **Linux 和 OSX 平台：** 在终端窗口源码目录下启动 `make` 命令，或在 LiteIDE 中构建项目。
- **Windows 平台：** 启动 MINGW 环境，步骤为：开始菜单，所有程序，MinGW，MinGW Shell（见 2.5.5 节），在命令行窗口输入 `make` 并回车，源代码被编译并链接为原生 exe 可执行程序。

生成内容为可执行程序，Linux/OS X 下为 `goto` ，Windows 下为 `goto.exe` 。

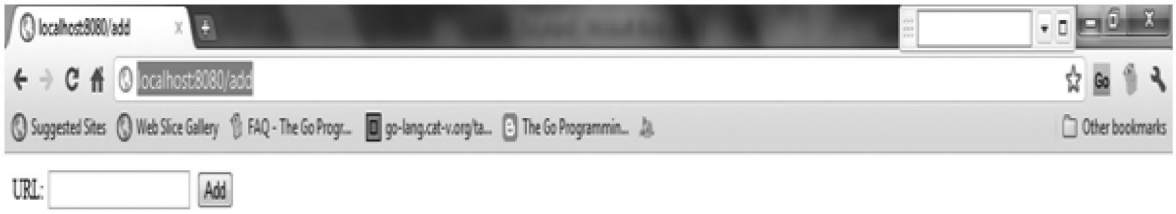
要启动并运行 web 服务器，那么：

- **Linux 和 OSX 平台：** 输入命令 `./goto` 。
- **Windows 平台：** 从 Go IDE 启动程序（如果 Windows 防火墙阻止程序启动，设置允许该程序）

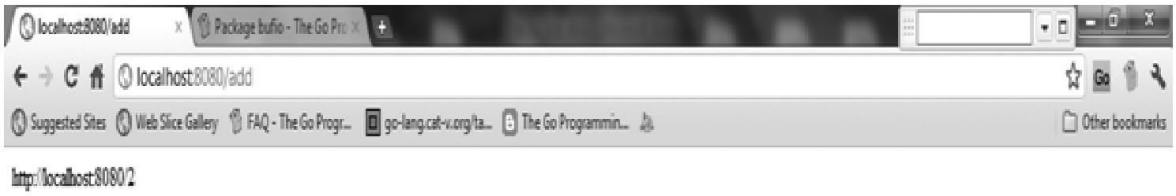
测试该程序

打开浏览器并请求 url： `http://localhost:8080/add`

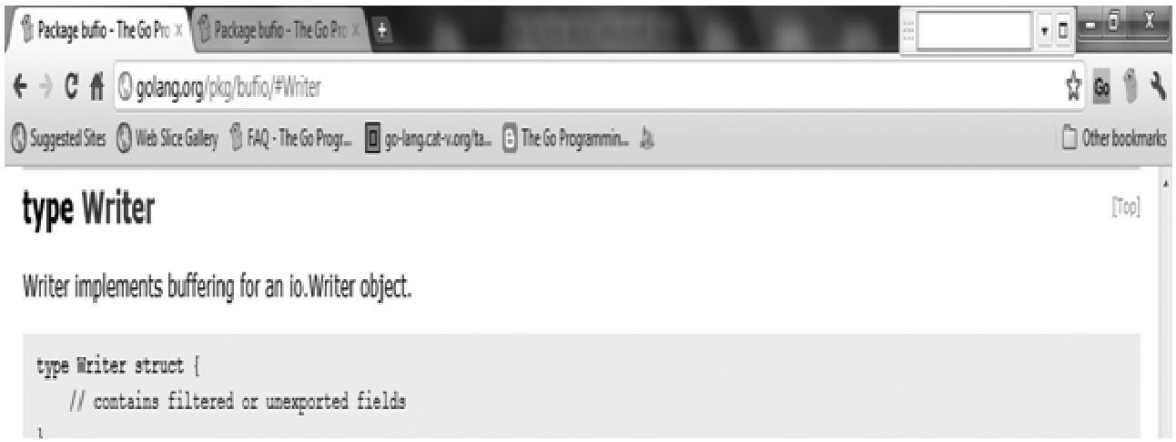
这会激活 `Add` 处理函数。请求还未包含 url 变量，所以响应会输出 html 表单询问输入：



添加一个长 URL 以获取等价的缩短版本，例如 `http://golang.org/pkg/bufio/#Writer` ，然后单击按钮。应用会为你产生一个短 URL 并打印出来，例如 `http:// localhost:8080/2` 。



复制该 URL 并在浏览器地址栏粘贴以发出请求，现在轮到 `Redirect` 处理函数上场了，对应长 URL 的页面被显示了出来。



持久化存储：gob

版本 2 - 添加持久化存储

第 2 个版本的代码 *goto_v2* 见 goto_v2。

19.5 持久化存储：gob

（本节代码见 goto_v2/store.go 和 goto_v2/main.go。）

当 goto 进程（监听在 8080 端口的 web 服务器）终止，这迟早会发生，内存 map 中缩短的 URL 就会丢失。要保留这些数据，就得将其保存到磁盘文件中。我们将修改 `URLStore`，使它可以保存数据到文件，且在 goto 启动时还原这些数据。为此我们使用 Go 标准库的 `encoding/gob` 包：它用于序列化和反序列化，将数据结构转换为字节数组（确切地说是切片），反之亦然（见 12.11 节）。

通过 `gob` 包的 `NewEncoder` 和 `NewDecoder` 函数，可以指定数据要写入或读取的位置。返回的 `Encoder` 和 `Decoder` 对象提供了 `Encode` 和 `Decode` 方法，用于对文件写入和从中读取 Go 数据结构。提示：`Encoder` 实现了 `Writer` 接口，同样 `Decoder` 实现了 `Reader` 接口。我们在 `URLStore` 上增加一个新的 `file` 字段（`*os.File` 类型），它是用于读写已打开文件的句柄。

```
type URLStore struct {
    urls map[string]string
    mu sync.RWMutex
    file *os.File
}
```

我们把这个文件命名为 store.gob，当初始化 `URLStore` 时将其作为参数传入：

```
var store = NewURLStore("store.gob")
```

接着，调整 `NewURLStore` 函数：

```
func NewURLStore(filename string) *URLStore {
    s := &URLStore{urls: make(map[string]string)}
    f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
    if err != nil {
        log.Fatal("URLStore:", err)
    }
    s.file = f
    return s
}
```

现在，更新后的 `NewURLStore` 函数接受一个文件名参数，它会打开该文件（见 12 章），将返回的 `*os.File` 作为 `file` 字段的值存储在 `URLStore` 变量 `store` 中，即这里的本地变量 `s`。

对 `OpenFile` 的调用可能会失败（例如文件可能被删除或改名）。它会返回一个错误 err，注意 Go 是如何处理这种情况的：

```
f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
if err != nil {
```

```
    log.Fatal("URLStore:", err)
}
```

当 err 不为 `nil` ，表示确实发生了错误，那么输出一条消息并停止程序执行。这是处理错误的一种方式，大多数情况下错误应该返回给调用函数，但这种检测错误的模式在 Go 代码中也很普遍。在 `}` 之后可以确定文件被成功打开了。

打开该文件时启用了写入标志，更精确地说是“追加模式”。每当一对新的短/长 URL 在程序中创建后，我们通过 `gob` 把它存储到文件 “store.gob” 中。

为达到目的，定义一个新的结构体类型 `record` ：

```
type record struct {
    Key, URL string
}
```

以及新的 `save` 方法，将给定的键和 URL 组成 `record` ，以 `gob` 编码的形式写入磁盘。

```
func (s *URLStore) save(key, url string) error {
    e := gob.NewEncoder(s.file)
    return e.Encode(record{key, url})
}
```

goto 程序启动时，磁盘上存储的数据必须读取到 `URLStore` 的 map 中。为此，我们编写 `load` 方法：

```
func (s *URLStore) load() error {
    if _, err := s.file.Seek(0, 0); err != nil {
        return err
    }
    d := gob.NewDecoder(s.file)
    var err error
    for err == nil {
        var r record
        if err = d.Decode(&r); err == nil {
            s.Set(r.Key, r.URL)
        }
    }
    if err == io.EOF {
        return nil
    }
    return err
}
```

这个新的 `load` 方法会寻址（ `Seek` ）到文件的起始位置，读取并解码（ `Decode` ）每一条记录（ `record` ），然后用 `Set` 方法将数据存储到 map 中。再次注意无处不在的错误处理模式。文件的解码由一个无限循环完成，只要没有错误就会一直继续：

```
for err == nil {
    ...
}
```

如果得到了一个错误，可能是刚解码了最后一条记录，于是产生了 `io.EOF` （EndOfFile） 错误。若并非此种错误，表示产生了解码错误，用 `return err` 来返回它。对该方法的调用必须加入到 `NewURLStore` 中：

```
func NewURLStore(filename string) *URLStore {
    s := &URLStore{urls: make(map[string]string)}
```

```
    f, err := os.OpenFile(filename, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
    if err != nil {
        log.Fatal("Error opening URLStore:", err)
    }
    s.file = f
    if err := s.load(); err != nil {
        log.Println("Error loading data in URLStore:", err)
    }
    return s
}
```

同时在 `Put` 方法中，当新的 URL 对加入到 map 中，也应该立即将它们保存到数据文件中：

```
func (s *URLStore) Put(url string) string {
    for {
        key := genKey(s.Count())
        if s.Set(key, url) {
            if err := s.save(key, url); err != nil {
                log.Println("Error saving to URLStore:", err)
            }
            return key
        }
    }
    panic("shouldn' t get here")
}
```

编译并测试这第二个版本的程序，或直接使用现有的可执行程序，验证关闭服务器（在终端窗口可以按 CTRL/C）并重启后，短 URL 仍然有效。goto 程序第一次启动时，文件 store.gob 还不存在，因此当载入数据时会得到错误：

```
2011/09/11 11:08:11 Error loading URLStore: open store.gob: The system cannot find the file specified.
```

结束进程并重启后，就能正常工作了。或者，可以在 goto 启动前先创建空的 store.gob 文件。

备注： 当第二次启动 goto 时，可能会产生错误：

```
Error loading URLStore: extra data in buffer
```

这是由于 `gob` 是基于流的协议，它不支持重新开始。在版本 4 中，会用 json 作为存储协议来补救此问题。

用协程优化性能

版本 3 - 添加协程

第 3 个版本的代码 *goto_v3* 见 goto_v3。

19.6 用协程优化性能

如果有太多客户端同时尝试添加 URL，第 2 个版本依旧存在性能问题。得益于锁机制，我们的 map 可以在并发访问环境下安全地更新，但每条新产生的记录都要立即写入磁盘，这种机制成为了瓶颈。写入操作可能同时发生，根据不同操作系统的特性，可能会产生数据损坏。就算不产生写入冲突，每个客户端在 `Put` 函数返回前，必须等待数据写入磁盘。因此，在一个 I/O 负载很高的系统中，客户端为了完成 `Add` 请求，将等待更长的不必要的时间。

为缓解该问题，必须对 `Put` 和存储进程*解耦*：我们将使用 Go 的并发机制。我们不再将记录直接写入磁盘，而是发送到一个*通道*中，它是某种形式的缓冲区，因而发送函数不必等待它完成。

保存进程会从该通道读取数据并写入磁盘。它是以 `saveLoop` 协程启动的独立线程。现在 `main` 和 `saveLoop` 并行地执行，不会再发生阻塞。

将 `URLStore` 的 `file` 字段替换为 `record` 类型的通道：`save chan record` 。

```
type URLStore struct {
    urls map[string]string
    mu sync.RWMutex
    save chan record
}
```

通道和 map 一样，必须用 `make` 创建。我们会以此修改 `NewURLStore` 工厂函数，并给定缓冲区大小为1000，例如：`save := make(chan record, saveQueueLength)` 。为解决性能问题，`Put` 可以发送记录 record 到带缓冲的 `save` 通道：

```
func (s *URLStore) Put(url string) string {
    for {
        key := genKey(s.Count())
        if s.Set(key, url) {
            s.save <- record{key, url}
            return key
        }
    }
    panic("shouldn't get here")
}
```

`save` 通道的另一端必须有一个接收者：新的 `saveLoop` 方法在独立的协程中运行，它接收 record 值并将它们写入到文件。`saveLoop` 是在 `NewURLStore()` 函数中用 `go` 关键字启动的。现在，可以移除不必要的打开文件的代码。以下是修改后的 `NewURLStore()` ：

```
const saveQueueLength = 1000
func NewURLStore(filename string) *URLStore {
    s := &URLStore{
        urls: make(map[string]string),
        save: make(chan record, saveQueueLength),
    }
```

```
    if err := s.load(filename); err != nil {
        log.Println("Error loading URLStore:", err)
    }
    go s.saveLoop(filename)
    return s
}
```

以下是 `saveLoop` 方法的代码：

```
func (s *URLStore) saveLoop(filename string) {
    f, err := os.Open(filename, os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0644)
    if err != nil {
        log.Fatal("URLStore:", err)
    }
    defer f.Close()
    e := gob.NewEncoder(f)
    for {
        // taking a record from the channel and encoding it
        r := <-s.save
        if err := e.Encode(r); err != nil {
            log.Println("URLStore:", err)
        }
    }
}
```

在无限循环中，记录从 `save` 通道读取，然后编码到文件中。

我们在 14 章深入学习了协程和通道，但在这里我们见到了实用的案例，更好地管理程序的不同部分。注意现在 `Encoder` 对象只被创建一次，而不是每次保存时都创建，这也可以节省了一些内存和运算处理。

还有一个改进可以使 goto 更灵活：我们可以将文件名、监听地址和主机名定义为标志（flag），来代替在程序中硬编码或定义常量。这样当程序启动时，可以在命令行中指定它们的新值，如果没有指定，将采用 flag 的默认值。该功能来自另一个包，所以需要 `import "flag"` （这个包的更详细信息见 12.4 节）。

先创建一些全局变量来保存 flag 的值：

```
var (
    listenAddr = flag.String("http", ":8080", "http listen address")
    dataFile = flag.String("file", "store.gob", "data store file name")
    hostname = flag.String("host", "localhost:8080", "host name and port")
)
```

为了处理命令行参数，必须把 `flag.Parse()` 添加到 `main` 函数中，在 flag 解析后才能实例化 `URLStore`，一旦得知了 `dataFile` 的值（在代码中使用了 `*dataFile`，因为 flag 是指针类型必须解除引用来获取值，见 4.9 节）：

```
var store *URLStore
func main() {
    flag.Parse()
    store = NewURLStore(*dataFile)
    http.HandleFunc("/", Redirect)
    http.HandleFunc("/add", Add)
    http.ListenAndServe(*listenAddr, nil)
}
```

现在 `Add` 处理函数中须用 `*hostname` 替换 `localhost:8080`：

```
fmt.Fprintf(w, "http://%s/%s", *hostname, key)
```

编译或直接使用现有的可执行程序测试第 3 个版本。

以 json 格式存储

版本 4 - 用 JSON 持久化存储

第 4 个版本的代码 *goto_v4* 见 goto_v4。

19.7 以 json 格式存储

如果你是个敏锐的测试者也许已经注意到了，当 goto 程序启动 2 次，第 2 次启动后能读取短 URL 且完美地工作。然而从第 3 次开始，会得到错误：

```
Error loading URLStore: extra data in buffer
```

这是由于 gob 是基于流的协议，它不支持重新开始。为补救该问题，这里我们使用 json 作为存储协议（见 12.9 节），它以纯文本形式存储数据，因此也可以被非 Go 语言编写的进程读取。同时也显示了更换一种不同的持久化协议是多么简单，因为与存储打交道的代码被清晰地隔离在 2 个方法中，即 `load` 和 `saveLoop` 。

从创建新的空文件 store.json 开始，更改 main.go 中声明文件名变量的那一行：

```
var dataFile = flag.String("file", "store.json", "data store file name")
```

在 store.go 中导入 `json` 取代 `gob` 。然后在 `saveLoop` 中唯一需要被修改的行：

```
e := gob.NewEncoder(f)
```

更改为：

```
e := json.NewEncoder(f)
```

类似的，在 `load` 方法中：

```
d := gob.NewDecoder(f)
```

修改为：

```
d := json.NewDecoder(f)
```

这就是所有要改动的地方！编译，启动并测试，你会发现之前的错误不会再发生了。

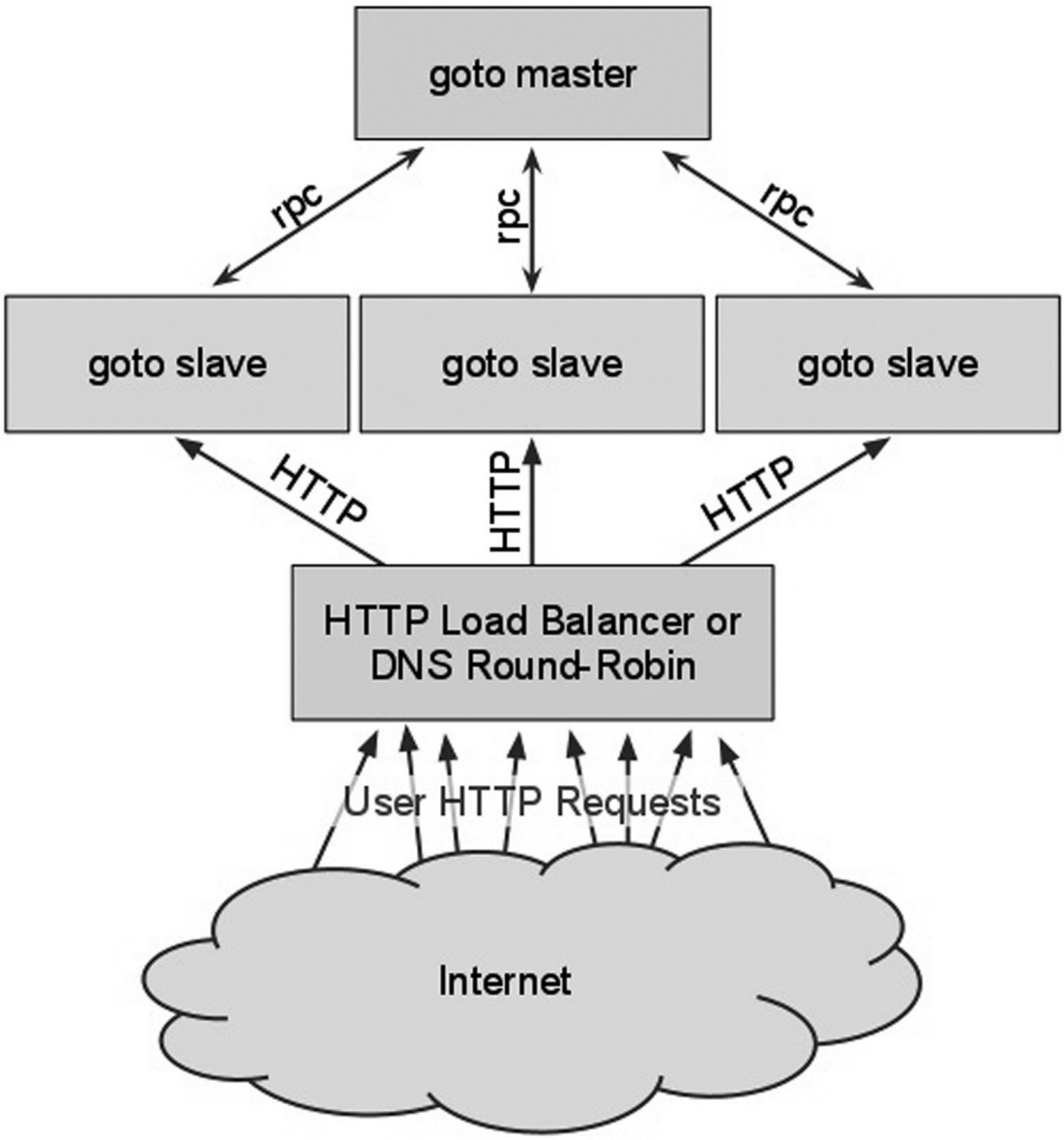
多服务器处理架构

版本 5 - 分布式程序

第 5 个版本的代码 *goto_v5*（19.8 节和 19.9 节讨论）见 goto_v5。该版本仍然基于 `gob` 存储，但很容易调整为使用 json，正如版本 4 演示的那样。

19.8 多服务器处理架构

目前为止 goto 以单线程运行，但即使用协程，在一台机器上运行的单一进程，也只能为一定数量的并发请求提供服务。一个缩短网址服务，相对于 `Add`（用 `Put()` 写入），通常 `Redirect` 服务（用 `Get()` 读取）要多得多。因此我们应该可以创建任意数量的只读的从（slave）服务器，提供服务并缓存 `Get` 方法调用的结果，将 `Put` 请求转发给主（master）服务器，类似如下架构：



对于 slave 进程，要在网络上运行 goto 应用的一个 master 节点实例，它们必须能相互通信。Go 的 `rpc` 包为跨越网络发起函数调用提供了便捷的途径。这里将把 `URLStore` 变为 RPC 服务（15.9 节 详细讨论了 rpc 包）。slave 进程将应对 `Get` 请求以交付长 URL。当一个长 URL 要被转换为缩短版本（使用 `Put` 方法）时，它们通过 rpc 连接把任务委托给 master 进程，因此只有 master 节点会写入数据文件。

截至目前 `URLStore` 上基本的 `Get()` 和 `Put()` 方法具有如下签名：

```
func (s *URLStore) Get(key string) string
func (s *URLStore) Put(url string) string
```

而 RPC 调用仅能使用如下形式的方法（t 是 T 类型的值）：

```
func (t T) Name(args *ArgType, reply *ReplyType) error
```

要使 `URLStore` 成为 RPC 服务，需要修改 `Put` 和 `Get` 方法使它们符合上述函数签名。以下是修改后的签名：

```
func (s *URLStore) Get(key, url *string) error
func (s *URLStore) Put(url, key *string) error
```

`Get()` 代码变更为：

```
func (s *URLStore) Get(key, url *string) error {
    s.mu.RLock()
    defer s.mu.RUnlock()
    if u, ok := s.urls[*key]; ok {
        *url = u
        return nil
    }
    return errors.New("key not found")
}
```

现在，键和长 URL 都变成了指针，必须加上前缀 `*` 来取得它们的值，例如 `*key` 这种形式。`u` 是一个值，可以用 `*url = u` 来将其赋值给指针。

接着对 `Put()` 代码做同样的改动：

```
func (s *URLStore) Put(url, key *string) error {
    for {
        *key = genKey(s.Count())
            if err := s.Set(key, url); err == nil {
            break
        }
    }
    if s.save != nil {
        s.save <- record{*key, *url}
    }
    return nil
}
```

`Put()` 调用 `Set()`，由于后者也要做调整，`key` 和 `url` 参数现在是指针类型，还必须返回 `error` 取代 `boolean`：

```
func (s *URLStore) Set(key, url *string) error {
    s.mu.Lock()
```

```
    defer s.mu.Unlock()
    if _, present := s.urls[*key]; present {
        return errors.New("key already exists")
    }
    s.urls[*key] = *url
    return nil
}
```

同样，当从 `load()` 调用 `Set()` 时，也必须做调整：

```
s.Set(&r.Key, &r.URL)
```

还必须修改 HTTP 处理函数以适应 `URLStore` 上的更改。 `Redirect` 处理函数现在返回 `URLStore` 给出错误的字符串形式：

```
func Redirect(w http.ResponseWriter, r *http.Request) {
    key := r.URL.Path[1:]
    var url string
    if err := store.Get(&key, &url); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, url, http.StatusFound)
}
```

`Add` 处理函数也以基本相同的方式修改：

```
func Add(w http.ResponseWriter, r *http.Request) {
    url := r.FormValue("url")
    if url == "" {
        fmt.Fprint(w, AddForm)
        return
    }
    var key string
    if err := store.Put(&url, &key); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    fmt.Fprintf(w, "http://%s/%s", *hostname, key)
}
```

要使应用程序更灵活，正如之前章节所为，可以添加一个命令行标志（flag）来决定是否在 `main()` 函数中启用 RPC 服务器：

```
var rpcEnabled = flag.Bool("rpc", false, "enable RPC server")
```

要使 RPC 工作，还要用 `rpc` 包来注册 `URLStore` ，并用 `HandleHTTP` 创建基于 HTTP 的 RPC 处理器：

```
func main() {
    flag.Parse()
    store = NewURLStore(*dataFile)
    if *rpcEnabled { // flag has been set
        rpc.RegisterName("Store", store)
        rpc.HandleHTTP()
    }
```

```
    ... (set up http like before)
}
```

使用代理缓存

19.9 使用代理缓存

`URLStore` 已经成为了有效的 RPC 服务，现在可以创建另一种代表 RPC 客户端的类型，它会转发请求到 RPC 服务器，我们称它为 `ProxyStore` 。

```
type ProxyStore struct {
    client *rpc.Client
}
```

一个 RPC 客户端必须使用 `DialHTTP()` 方法连接到服务器，所以我们把这句加入 `NewProxyStore` 函数，它用于创建 `ProxyStore` 对象。

```
func NewProxyStore(addr string) *ProxyStore {
    client, err := rpc.DialHTTP("tcp", addr)
    if err != nil {
        log.Println("Error constructing ProxyStore:", err)
    }
    return &ProxyStore{client: client}
}
```

`ProxyStore` 有 `Get` 和 `Put` 方法，它们利用 RPC 客户端的 `Call` 方法，将请求直接传递给服务器：

```
func (s *ProxyStore) Get(key, url *string) error {
    return s.client.Call("Store.Get", key, url)
}

func (s *ProxyStore) Put(url, key *string) error {
    return s.client.Call("Store.Put", url, key)
}
```

带缓存的 ProxyStore

可是，如果 slave 进程只是简单地代理所有的工作到 master 节点，不会得到任何增益！我们打算用 slave 节点来应对 `Get` 请求。要做到这点，它们必须有 `URLStore` 中 map 的一份副本（缓存）。因此我们对 `ProxyStore` 的定义进行扩展，将 `URLStore` 包含在其中：

```
type ProxyStore struct {
    urls *URLStore
    client *rpc.Client
}
```

`NewProxyStore` 也必须做修改：

```
func NewProxyStore(addr string) *ProxyStore {
    client, err := rpc.DialHTTP("tcp", addr)
    if err != nil {
        log.Println("ProxyStore:", err)
    }
```

```
    return &ProxyStore{urls: NewURLStore(""), client: client}
}
```

还必须修改 `NewURLStore` 以便给出空文件名时，不会尝试从磁盘写入或读取文件：

```
func NewURLStore(filename string) *URLStore {
    s := &URLStore{urls: make(map[string]string)}
    if filename != "" {
        s.save = make(chan record, saveQueueLength)
        if err := s.load(filename); err != nil {
            log.Println("Error loading URLStore: ", err)
        }
        go s.saveLoop(filename)
    }
    return s
}
```

`ProxyStore` 的 `Get` 方法需要扩展：**它应该首先检查缓存中是否有对应的键**。如果有，`Get` 返回已缓存的结果。否则，应该发起 RPC 调用，然后用返回结果更新其本地缓存：

```
func (s *ProxyStore) Get(key, url *string) error {
    if err := s.urls.Get(key, url); err == nil { // url found in local map
        return nil
    }
    // url not found in local map, make rpc-call:
    if err := s.client.Call("Store.Get", key, url); err != nil {
        return err
    }
    s.urls.Set(key, url)
    return nil
}
```

同样地，`Put` 方法仅当成功完成了远程 RPC `Put` 调用，才更新本地缓存：

```
func (s *ProxyStore) Put(url, key *string) error {
    if err := s.client.Call("Store.Put", url, key); err != nil {
        return err
    }
    s.urls.Set(key, url)
    return nil
}
```

汇总

slave 节点使用 `ProxyStore`，只有 master 使用 `URLStore`。有鉴于创造它们的方式，它们看上去十分一致：两者都实现了相同签名的 `Get` 和 `Put` 方法，因此我们可以指定一个 `Store` 接口来概括它们的行为：

```
type Store interface {
    Put(url, key *string) error
    Get(key, url *string) error
}
```

现在全局变量 `store` 可以成为 `Store` 类型：

```
var store Store
```

最后，我们改写 `main()` 函数以便程序只作为 master 或 slave 启动（我们只能这么做，因为现在 store 是 `Store` 接口类型！）。

为此我们添加一个没有默认值的新命令行标志 `masterAddr` 。

```
var masterAddr = flag.String("master", "", "RPC master address")
```

如果给出 master 地址，就启动一个 slave 进程并创建新的 `ProxyStore` ；否则启动 master 进程并创建新的 `URLStore` ：

```
func main() {
    flag.Parse()
    if *masterAddr != "" { // we are a slave
        store = NewProxyStore(*masterAddr)
    } else { // we are the master
        store = NewURLStore(*dataFile)
    }
    ...
}
```

这样，我们已启用了 `ProxyStore` 作为 web 前端，以代替 `URLStore` 。

其余的前端代码继续和之前一样地工作，它们不必在意 `Store` 接口。只有 master 进程会写数据文件。

现在可以加载一个 master 节点和数个 slave 节点，对 slave 进行压力测试。

编译这个版本 4 或直接使用现有的可执行程序。

要进行测试，首先在命令行用以下命令启动 master 节点：

```
./goto -http=:8081 -rpc=true    # （Windows 平台用 goto 代替 ./goto）
```

这里提供了 2 个标志：master 监听 8081 端口，已启用 RPC。

slave 节点用以下命令启动：

```
./goto -master=127.0.0.1:8081
```

它获取到 master 的地址，并在 8080 端口接受客户端请求。

在源码目录下已包含了以下 shell 脚本 demo.sh，用来在类 Unix 系统下自动启动程序：

```
#!/bin/sh
gomake
./goto -http=:8081 -rpc=true &
master_pid=$!
sleep 1
./goto -master=127.0.0.1:8081 &
slave_pid=$!
echo "Running master on :8081, slave on :8080."
echo "Visit: http://localhost:8080/add"
echo "Press enter to shut down"
read
```

```
kill $master_pid
kill $slave_pid
```

要在 Windows 下测试，启动 MINGW shell 并启动 master，然后每个 slave 都要单独启动新的 MINGW shell 并启动 slave 进程。

总结和增强

19.10 总结和增强

通过逐步构建 goto 应用程序，我们遇到了几乎所有的 Go 语言特性。

虽然这个程序按照我们的目标行事，仍然有一些可改进的途径：

- *审美*：用户界面可以（极大地）美化。为此可以使用 Go 的 `template` 包（见 15.7 节）。
- *可靠性*：master/slave 之间的 RPC 连接应该可以更可靠：如果客户端到服务器之间的连接中断，客户端应该尝试重连。用一个 “dialer” 协程可以达成。
- *资源减负*：由于 URL 数据库大小不断增长，内存占用可能会成为一个问题。可以通过多台 master 服务器按照键分片来解决。
- *删除*：要支持删除短 URL，master 和 slave 之间的交互将变得更复杂。