Go入门指南

极客学院出版

前言

本书介绍

在接触 Go 语言之后，对这门编程语言非常着迷，期间也陆陆续续开始一些帮助国内编程爱好者了解和发展 Go 语言的工作，比如开始录制视频教程《Go编程基础》。但由于目前国内并没有比较好的 Go 语言书籍，而国外的优秀书籍因为英文的缘故在一定程度上也为不少 Go 语言爱好者带来了一些学习上的困扰，不仅为了加快扩散 Go 爱好者的国内群体，本人在完成阅读这本名叫《The Way to Go》之后，决定每天抽出一点时间来进行翻译的工作，并且以开源的形式免费分享给有需要的 Go 语言爱好者。

关于译者

本书的主要译者是 @无闻Unknwon，是一名 Go 语言爱好者和传播者，目前是 Go Walker、Gopm、Gogs 和 Macaron 创始人，极客学院签约讲师。

适用人群

适合有一定编程基础，初学 Go 语言的爱好者。

> Martini&Macaron 交流群：371440803 > Golang 编程：245386165

更新日期	更新内容
2015-08-25	10.8 垃圾回收和 SetFinalizer

目录

1 前言

用更少的代码，更短的编译时间，创建运行更快的程序，享受更多的乐趣

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

随着软件规模的不断扩大，诸多的学者和谷歌的开发者们在公司内部的软件开发过程中开始经历大量的挫折，在诸多问题上都不能给出令人满意的解决方案，尤其是在使用 C++ 来开发大型的服务端软件时，情况更是不容乐观。由于二进制文件一般都是非常巨大的，因此需要耗费大量的时间在编译这些文件上，同时编程语言的设计思想也已经非常陈旧，这些情况都充分证明了现有的编程语言已不符合时下的生产环境。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）极大了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！

链接

- 目录
- 下一部分: Go 语言的起源，发展与普及

2

第一部分：学习 Go 语言

第1章：Go 语言的起源，发展与普及

第2章：安装与运行环境

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。

链接

- 目录
- 上一章：Go 解释器
- 下一节：Go 开发环境的基本要求

3.1 Go 开发环境的基本要求

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

1. 语法高亮是必不可少的功能，这也是为什么每个开发工具都提供配置文件来实现自定义配置的原因。
2. 可以自动保存代码，至少在每次编译前都会保存。
3. 可以显示代码所在的行数。
4. 拥有较好的项目文件纵览和导航能力，可以同时编辑多个源文件并设置书签，能够匹配括号，能够跳转到某个函数或类型的定义部分。
5. 完美的查找和替换功能，替换之前最好还能预览结果。
6. 可以注释或取消注释选中的一行或多行代码。
7. 当有编译错误时，双击错误提示可以跳转到发生错误的位置。
8. 跨平台，能够在 Linux、Mac OS X 和 Windwos 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。

链接

- 目录
- 上一节：编辑器、集成开发环境与其它工具
- 下一节：编辑器和集成开发环境

3.2 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、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，详见页面：http://gohelp.wordpress.com/。

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

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

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

链接

- 目录
- 上一节：Go 开发环境的基本要求
- 下一节：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）。

链接

- 目录

- 上一节：编辑器和集成开发环境
- 下一节：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` 编译并安装自身包和依赖包
- `go install` 安装自身包和依赖包

链接

- 目录
- 上一节：调试器
- 下一节：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 map1` 会格式化并重写 map1 目录及其子目录下的所有 Go 源文件。

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

实例：

```
gofmt -r “(a) -> a” -w *.go
```

上面的代码会将源文件中没有意义的括号去掉。

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

上面的代码会将源文件中多余的 `len(a)` 去掉。（**译者注：了解切片（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/。

链接

- 目录
- 上一节：构建并运行 Go 程序
- 下一节：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）。

链接

- 目录
- 上一节：格式化代码
- 下一节：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` 之所以能够正常功能是因为 Go 在标准库就提供生成抽象语法树和通过抽象语法树对代码进行还原的功能。该工具会尝试更新当前目录下的所有 Go 源文件，并在完成代码更新后在控制台输出相关的文件名称。
- `go test` 是一个轻量级的单元测试框架（第 13 章）。

链接

- 目录
- 上一节：生成代码文档

- 下一节：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，并比这两门语言使用少占用 70% 的内存。在很多情况下这种比较是没有意义的，因为像谷歌这样拥有成千上万台服务器的公司都抛弃 C++ 而开始将 Go 用于生产环境已经足够说明它本身所具有的优势。

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

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

其实比较多门语言之间的性能是一种非常猥琐的行为，因为任何一种语言都有其所擅长和薄弱的方面。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，比 Java 少 4 倍左右。

链接

- 目录
- 上一节：其它工具
- 下一节：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.CStrin g(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。

链接

- 目录
- 上一节：Go 性能说明
- 下一部分：语言的核心结构与技术

3

第二部分：语言的核心结构与技术

第4章：基本结构和基本数据类型

5.0 控制结构

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

- if-else 结构
- switch 结构
- elect 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。

链接

- 目录
- 上一章：指针
- 下一节：if-else 结构

6.0 函数

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。

链接

- 目录
- 上一章：标签与 goto
- 下一节：介绍

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。

链接

- 目录
- 上一章：通过内存缓存来提升性能
- 下一节：声明和初始化

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。

链接

- 目录
- 上一章：字符串、数组和切片的应用
- 下一节：声明、初始化和 make

9.0 包（package）

本章主要针对 Go 语言的包展开讲解。

链接

- 目录
- 上一章：将 map 的键值对调
- 下一节：标准库概述

10 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。

链接

- 目录
- 上一章：在 Go 程序中使用外部库
- 下一节：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，可以使用逗号符给字段赋值：`structname.fieldname = value`。

同样的，使用逗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 mt 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)
```

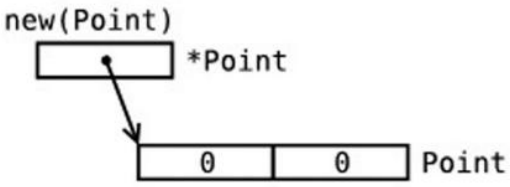
在（A）中，值必须以字段在结构体定义时的顺序给出，& 不是必须的。（B）显示了另一种方式，字段名加一个冒号放在值的前面，这种情况下值的顺序不必一致，并且某些字段还可以被忽略掉，就像（C）中那样。

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。

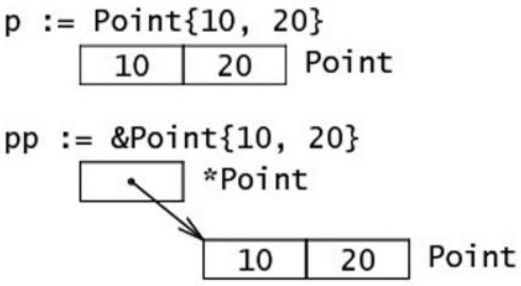
下图说明了结构体类型实例和一个指向它的指针的内存布局：

```
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 strcut1 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
```

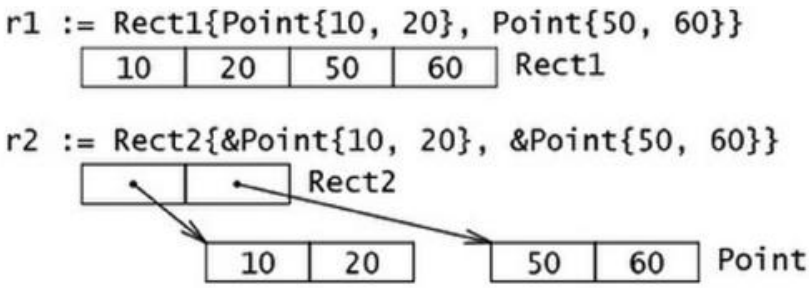
在上面例子的第二种情况中，可以直接通过指针，像 `pers2.lastName="Woodward"` 这样给结构体字段赋值，没有像 C++ 中那样需要使用 `->` 操作符，Go 会自动做这样的转换。

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：

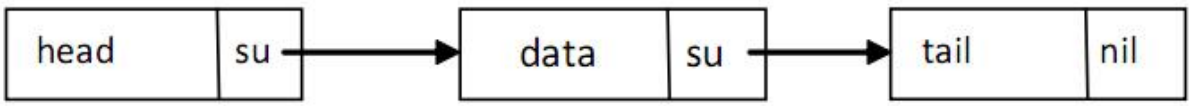
```
type Rect1 struct {Min, Max Point }
type Rect2 struct {Min, Max *Point }
```



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：

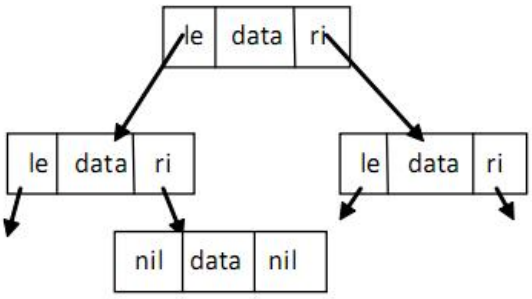
```
type Node struct {
    data    float64
    su      *Node
}
```

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      *su
}
```

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 strcut {
    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 persionext1.go：

修改 persionext1.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 10.3 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 `Primeter()`，然后进行测试。

链接

- 目录
- 上一节：结构（struct）与方法（method）
- 下一节：使用工厂方法创建结构体

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，俺惯例，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(id, 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子。

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（参考第 4.2.1、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）
```

下面的例子来说明了在映射上使用 new 和 make 的区别，以及可能的发生的错误：

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
    (*y).thingOne = "hello"
    (*y).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() 一个映射并试图使用数据填充它，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。

链接

- 目录
- 上一节：结构体定义 ）
- 下一节：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
```

链接

- 目录
- 上一节：使用工厂方法创建结构体实例
- 下一节：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
```

链接

- 目录
- 上一节：使用自定义包中的结构体
- 下一节：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，此时类型也就是字段的名字。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.Printf("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，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。

链接

- 目录
- 上一节：带标签的结构体
- 下一节：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。

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
```

别名类型不能有它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` 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 list.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。

但是有一个绕点的方式：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() 接受通过拷贝接受 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。

我们知道方法不需要指针作为接收者，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 float }
// A method on Point3
func (p Point3) Abs float {
    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 10.11（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.firsetname 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 方法只适用成员名。

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 节的联系）：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

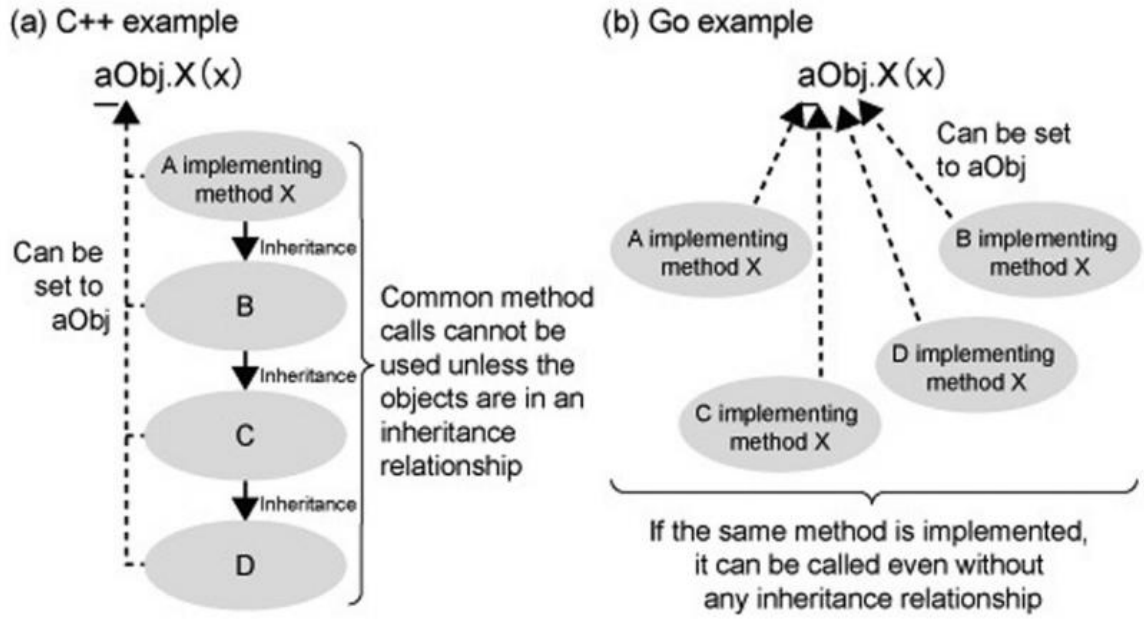
在编程中一些基本操作会一遍又一遍的出现，比如打开（Open）、关闭（Close）、读（Read）、写（Write）、排序（Sort）等等，并且它们都有一个大致的意思：打开（Open）可以作用于一个文件、一个网络连接、一个数据库连接等等。具体的实现可能千差万别，但是基本的概念是一致的。在 Go 语言中，通过使用接口（参考 第 11 章），标准库广泛的应用了这些规则，在标准库中这些通用方法都有一致的名字，比如 `Open()`、`Read()`、`Write()` 等。想写规范的 Go 程序，就应该遵守这些约定，给方法合适的名字和签名，就像那些通用方法那样。这样做会使 Go 开发的软件更加具有一致性和可读性。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(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**。

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。

链接

- 目录

- 上一节：匿名字段和内嵌结构体
- 下一节：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

当定义一个了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，它导致了一个无限迭代调用（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", s)
}
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 := &{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

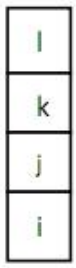
为 int 定义一个别名类型 Day，定义一个字符串数组它包含一周七天的名字，为类型 Day 定义 String() 方法，它输出星期几的名字。使用 iota 定义一个枚举常量用于表示一周的中每天（MO、TU...）。

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）之顶部（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。

push 将一个新值放到栈的最顶部一个非空（非零）的格子中。

pop 获取栈的最顶部一个非空（非零）的格子的值。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 `Stack` 类型，并为它定义一个 `Push` 和 `Pop` 方法，再为它定义 `String()` 方法（用于调试）它输出栈的内容，比如：`[0:i] [1:j] [2:k] [3:l]`。

1）stack_arr.go：使用长度为 4 的 int 数据作为底层数据结构。 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，所以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。

链接

- 目录
- 上一节：方法
- 下一节：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)
```

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。

链接

- 目录
- 上一节：类型的 String() 方法和格式化描述符
- 下一章：什么是接口?