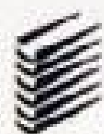
TURING 图灵原创

云计算时代的新型编程语言 编程极客不可不学

示例丰富，引人入胜，一本就Go!

Go语言·云动力

[新加坡]樊虹剑◎著



人民邮电出版社
POSTS & TELECOM PRESS

Go语言·云动力

云计算时代，对编程语言的要求也越来越高，而现有编程语言都无法满足大规模网络应用的需求，更无法同时满足程序员高效编译、高效执行和轻松编程的要求。2007年，Go语言应运而生。它面向Web和多核计算，强调速度，并引入了高效、低延迟的垃圾回收算法，同时精简了类型，摒弃了危险的指针运算。所有这些特点，都使得Go既容易学习，也便于使用，无论是要解决手边的小问题，还是要集体完成大项目，Go都是合适的通用语言。

本书作者是将Go语言在国内传播的第一人，对Go的编译器、运行器和各种包有深入研究，目前已正式成为Go语言的Contributor。本书是他这几年研究心血之结晶，旨在为读者了解Go语言、掌握Go语言提供专业的入门指导。

书中内容共分为9章，全面介绍了Go语言的基础知识，以及Go语言在云计算中的应用。作者还提供了70多个完整的示例程序，方便读者边学边练，加深记忆。通过轻松简洁的介绍，读者定能顺利进入Go语言的世界。

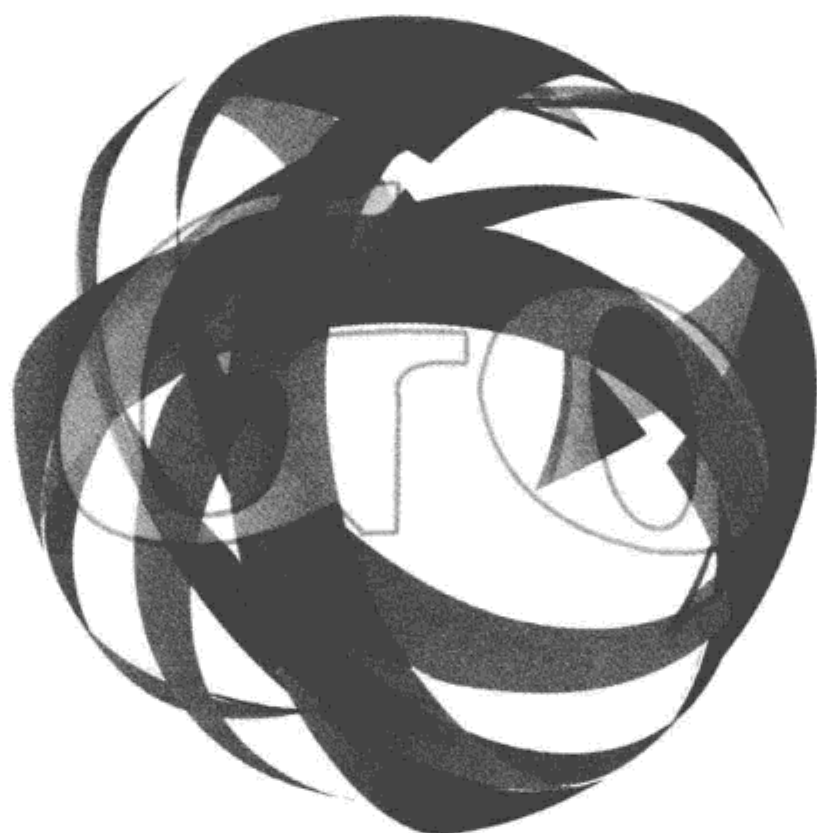
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/Go语言

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-28307-8
定价：39.00元

TURING 图灵原创



Go语言·云动力

[新加坡]樊虹剑○著

人民邮电出版社
北京

图书在版编目（CIP）数据

Go语言·云动力 /（新加坡）樊虹剑著. -- 北京：
人民邮电出版社，2012.6
（图灵原创）
ISBN 978-7-115-28307-8

Ⅰ. ①G… Ⅱ. ①樊… Ⅲ. ①程序语言—程序设计
Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第104152号

内 容 提 要

Go 语言是由谷歌的 Rob Pike、Ken Thompson 和 Robert Griesemer 共同设计开发的一种新型程序设计语言。2012 年 Go 1 的推出，代表着 Go 语言的稳定成熟，也正式宣告 Go 走入了主流语言的行列。本书是 Go 语言程序设计入门书，介绍了 Go 语言的基础知识，包括静态类型、流程控制、函数、动态类型、面向对象、并发编程等内容，以及同其他 C 类语言相比，Go 所具备的全新特性。同时，本书还介绍了 Go 语言在云计算中的应用。

本书适合 Go 语言初学者学习。

图灵原创
Go语言·云动力

◆ 著 [新加坡] 樊虹剑
责任编辑 王军花
执行编辑 李 静
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷
◆ 开本：880×1230 1/32
印张：9.625
字数：250千字 2012年6月第1版
印数：1－4 000册 2012年6月河北第1次印刷
著作权合同登记号 图字：01-2012-2781号
ISBN 978-7-115-28307-8

定价：39.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

前　　言

Less is more.

（舍既是得。）

——Robert Browning

从 1978 年《C 程序设计语言》问世到现在，计算机世界已经发生了翻天覆地的变化。计算机器本身的性能已经得到几千倍的提高，计算机的互联互通更使很多不可思议的事情成为常态。身为计算机灵魂的工程师，程序员一直处在这场变革的风口浪尖。他们既有弄潮儿的自信与洒脱，又时刻面临被后浪吞噬的危险。更微妙的是，尽管计算机科学的先驱们很早就发现，编程是门文字和思维的艺术，而且顶尖程序员的生产力要远大于集体化大规模生产作业中的员工。但是，由于整个计算机产业的迅速发展，以及具备艺术细胞的程序员的稀缺，从业者只能重点投资工具和流程。自从 20 世纪 60 年代提出“软件危机”这一问题以来，各种理论和实践不断涌现，而通过变革编程语言来解决问题的努力，也一直在继续。但到目前为止，软件还在危机，同志仍需努力。

C.A.R Hoare 在 1980 年图灵奖获奖致辞中阐述得很精辟：

“设计软件有两种方式：第一种是尽量简化，使之明显地看不到弱点；另一种是尽量复杂，从而看不到明显的弱点。第一种方式非常困难。就像发现复杂的自然现象之中蕴含着简单的

物理定律一样，它需要投入、明察、甚至灵感。”

今天，我非常高兴地向大家介绍 Go 语言。40 年前，C 编程语言极大地解放了程序员的生产力。今天，新兴的 Go 语言也一定能够再次提升程序员的创造力和协作能力，从而让有才能的程序员得以在创作中充分发挥，让程序员重新成为令人尊敬的计算机灵魂的工程师。

目前，Go 语言才发布了短短几年时间，就已经赢得了广泛赞誉，并得到了广泛应用。它不但用于支持谷歌和 Heroku 的云计算平台，还在一些大公司的内部系统中担当关键角色，甚至普通程序员也用 Go 代替其他常见语言来处理手边的问题。这充分说明 Go 是一种成熟、实用、好用的语言。它的未来非常光明。

本书所介绍的内容正是使用 Go 语言时必不可少的基础知识。作者不满足于平铺直叙的方式，而希望能通过轻松的表达，让 Go 自然而然地接近读者，并以作者自身的经验和观点，启发读者独立地分析和试验，从而让读者能顺利地从入门到精通，让 Go 成为最好的编程语言。

简　介

Go是种通用的编程语言。它类似C,也参考了其他语言的一些设计。但从各方面来看，Go都是种新语言。它的诞生源于Go的发明者对现有编程语言的不满，认为它们使编程变得太过困难，编译速度太慢，或者运行效率太低。所以,2007年,谷歌的Rob Pike、Ken Thompson和Robert Griesemer决定设计一种全新的编程语言。两年后的2009年11月10日,他们发布了源代码全部开放的Go项目。而2012年Go1的推出，既代表着Go语言的稳定成熟，也正式宣告Go走入了主流语言的行列。

Go的作者在介绍Go项目的目的时指出：

已经有十几年没有出现新的系统语言了，但此时计算环境发生了巨大变化，出现如下趋势。

(1) 计算机的运行速度变得极快，而软件开发速度基本没变。

(2) 依赖管理是今天软件开发的一大部分,但传统C语言的头文件是干净的依赖分析和快速编译的对立面。

(3) 反抗Java和C++等语言笨重的类型系统的起义已在壮大，推动着人们使用Python和JavaScript等动态类型系统语言。

(4) 一些基本概念，例如垃圾回收和并行计算，并未在主流语言中得到充分支持。

(5) 多核计算机的兴起带来了担忧和困扰。

所以，我们认为值得尝试开发一种新语言，一种并发的、支持垃圾回收的、快速编译型语言。

Go的产生源于对现有系统编程语言和环境的不满。编程已经变得很难，部分原因要归咎于选用的语言。人们要在高效编译、高效执行和轻松编程之间取舍；没有一种主流语言同时满足这三个方面的要求。相比安全性和效率，更注重轻松编程的程序员选择使用动态类型的语言，如Python和JavaScript，而不是C++，或者相对舒缓些的Java。

Go试图结合能轻松编程的解释型动态类型语言，以及能提高效率和安全性的静态类型编译语言。它面向的是现代的、支持网络功能的多核计算。最后，它还能保证速度：在一台计算机上编译一个大型执行文件最多就几秒钟。而要实现这些目标则需要解决几个语法问题：一个丰富但轻盈的类型系统；并发和支持垃圾回收；稳定的依赖规范等。这些靠库和工具都不能很好解决，新的语言在召唤。

这就是说，Go语言在设计时，就糅合了静态编译型语言的高效和安全，以及动态解释型语言容易编程的特点，又面向当代的网络和多核计算，强调速度，精简了类型，避免冗长的声明，保持每个概念的独立。这些，使Go既容易学习，也便于使用。无论是解决手边的小问题，还是依靠集体完成的大项目，Go都是合适的通用语言。

Go是对C语言最彻底的一次改进，不只是声明语法与C完全不同，而且舍弃了危险的指针运算，还调整了运算符的优先级，并在各个细微的地方做出必要的改变。这使Go语言更规整、更安全。所以尽管具备C类语言的基础可以帮助学习Go，但舍弃C的使用习惯，熟悉Go的语言风格，也是对有经验的程序员的一种挑战。

第1章通过具体的案例，解释Go程序的基本结构，让有一定功力的读者迅速上手，也让初学者对Go有个初步印象，便于展开后续章节。

Go提供了丰富的数据类型，包括不同字宽的整型、浮点型、复数型、数组、字符串、切片、结构、映射、程道和界面。它们声明的变量和常

量，像细胞一样成为一个程序最基本的单元。数据类型、变量和表达式构成了第2章的内容。

Go的流程控制语句能搭建结构良好的程序：块用于语句分组和作用域控制，if-else 用于判断，switch 用于选择，for 用于循环。第3章将通过具体的例子讲述每个基本的控制语句。

Go的函数声明不可以嵌套，但本身可以作为变量，可以返回多个值，可以递归调用，也可以使用并保存外部的变量。变量通常在离开作用域时失效。但Go自动跟踪变量的分配，确保仍被使用的变量不会回收，而不再使用的变量，也能及时地被GC（Garbage Collection，垃圾收集器）收集，释放其所占内存。第4章专门讲解函数。

这样，前四章就全面介绍了Go这一静态编译语言的基础知识。

Go的界面类型是动态编程的基础，在程序运行时才根据具体值的类型来确定调用的方法。第5章介绍映射、界面和动态编程。

第6章介绍Go面向对象的设计。Go的各种数据类型，都可以作为函数的接受者。这样函数成为此类型的方法。这使Go具备面向对象的语法和数据封装特性，但Go不使用继承。Go的内置结构和界面类型可以满足类似功能，也更加灵活。

Go的招牌是并发机制。Go的函数可以用 go 语句并发执行，称为"去程"（goroutine）。并发去程的同步与数据传递靠的是程道类型的变量。select 语句可以从多个程道中随机挑选一个不再阻塞的去程执行。第7章将介绍Go的并发原语的使用。

第8章介绍GAP的Go运行器的使用，分析云计算的特点，以及Go如何在语言层级为云计算提供动力。

第9章介绍Go语言的一些标准包，包括 fmt、bytes、http、time、template、regexp、gob 和 json。

附录提供Go工具的用法、完整的EBNF语法规范，以及中英术语对照表，方便大家参考。

写作风格

语言是我们分析和交流的工具。尽管计算机编程需要使用简单的英语，但我们对于问题的思考和解决方法的推导，仍然要依靠母语。所以，本书尽量不使用英文术语，而是试图和其他科学一样，使用对应的中文翻译。这样，程序就像数学公式一样，成了抽象推理的工具。例如，对于变量名，本书会尽量使用类似数学公式中简单的 xyz 那样的字母，而不使用有意义的英文。尽量不使用注释，而是依靠正文的讲解，帮助理解程序。

Go 有一些新的概念，例如 goroutine 和 channel，并没有现成的译法，作者就自作聪明地把它们译为“去程”和“程道”，希望能音意皆通。而在 Go 中，还有一些受 Java 影响但用法不同的词，例如 interface，为了不受旧有概念的影响，也使用了新的译名“界面”。作者相信这些都能对读者使用母语学习 Go 语言有所帮助。

作者深信，学习和阅读是个需要集中精力的过程，任何花边注脚都是对正文学习的干扰。所以，本书版面非常简单，只有文字介绍和 Go 语言程序。作者也深信，学习编程的最好方法是亲自动手编写程序。只有这样才能理解程序的整体结构和细枝末节，只有自己动手发现和排除错误，才能从中学到东西。所以，尽管本书提供了 70 多个完整的示例程序，但不提供源代码下载，给大家一个边学边练的机会。

目　录

第 1 章　快速入门 …… 1

1.1 编辑和编译 …… 2
1.2 世界，你好！ …… 9
1.3 自我复制 …… 11
1.4 猜数游戏 …… 14
1.5 图灵机 …… 19
1.6 排版工具 …… 23
1.7 游乐场 …… 30
1.8 位钱 …… 36
1.9 小结 …… 43

第 2 章　静态类型 …… 44

2.1 数制 …… 45
2.1.1 整数 …… 45
2.1.2 二进制补码 …… 46
2.1.3 整数比较 …… 47
2.1.4 按位运算 …… 48
2.1.5 浮点数 …… 50
2.1.6 复数 …… 51
2.1.7 优先级 …… 52
2.1.8 表达式 …… 53
2.2 字符和字符串 …… 54

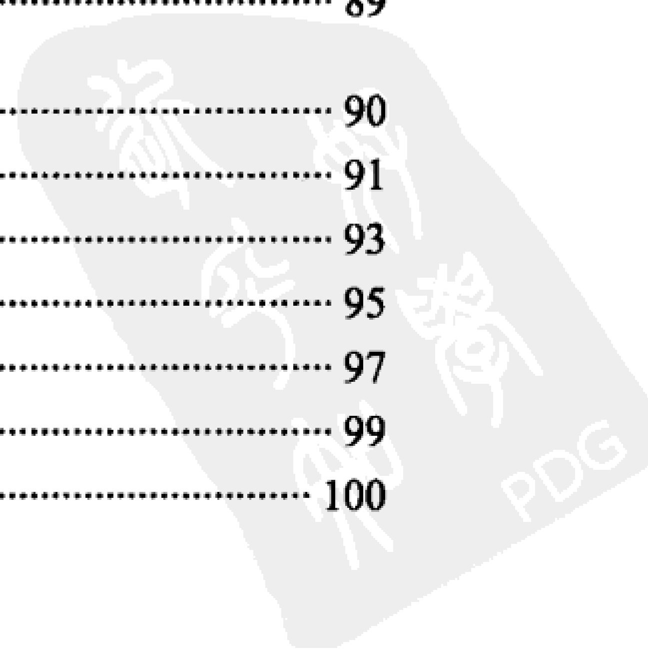
2.2.1 UTF-8 编码 ··· 54
2.2.2 Unicode 字符 ··· 55
2.2.3 转义字符 ··· 56
2.2.4 字符串 ··· 56
2.2.5 字符串转换 ··· 57
2.3 数组 ··· 60
2.3.1 声明 ··· 60
2.3.2 下标 ··· 61
2.3.3 赋值 ··· 61
2.4 切片 ··· 63
2.5 结构体 ··· 67
2.5.1 项 ··· 68
2.5.2 内置 ··· 69
2.6 指针 ··· 71
2.7 小结 ··· 73

第 3 章　流程控制 ··· 74

3.1 简单语句 ··· 75
3.2 判断语句 if ··· 77
3.3 多分支语句 switch ··· 79
3.4 循环语句 for ··· 82
3.5 遍历 ··· 84
3.6 标号和跳转 ··· 85
3.7 作用域 ··· 86
3.8 小结 ··· 88

第 4 章　函数 ··· 89

4.1 签名 ··· 90
4.2 参数 ··· 91
4.3 返回语句 ··· 93
4.4 函数调用 ··· 95
4.5 闭包 ··· 97
4.6 压后 ··· 99
4.7 派错和恢复 ··· 100



4.8 方法 ······102
4.9 包 ······105
4.10 导入 ······107
4.11 程序初始化 ······108
4.12 小结 ······110

第5章 动态类型 ······111

5.1 映射 ······112
5.2 界面类型 ······116
5.3 界面值 ······119
5.4 error 界面 ······122
5.5 有界无类 ······124
5.6 排序 ······126
5.7 类型断言 ······130
5.8 类型分支 ······133
5.9 反射 ······135
5.10 小结 ······141

第6章 面向对象 ······142

6.1 背景 ······143
6.2 术语 ······146
6.3 与 C++对比 ······149
6.3.1 继承 ······149
6.3.2 抽象类 ······151
6.3.3 泛型 ······153
6.4 小结 ······154

第7章 并发编程 ······155

7.1 背景 ······156
7.2 同步通信 ······158
7.3 去程 ······162
7.4 程道 ······164
7.5 遍历与关闭 ······166
7.6 MapReduce ······168

7.7 select 语句 …… 170
7.8 程道值 …… 173
7.9 互斥 …… 175
7.10 小结 …… 177

第8章 云计算 …… 178

8.1 背景 …… 179
8.2 GAE …… 181
8.3 Hello 世界！ …… 183
8.4 画胡子 …… 185
8.5 留言录 …… 191
8.6 用户 API …… 195
8.7 数据库 API …… 196
8.7.1 术语 …… 197
8.7.2 Go 数据库 API …… 197
8.7.3 实体键 …… 199
8.7.4 查询和索引 …… 200
8.7.5 实体组 …… 201
8.7.6 限制 …… 201
8.8 交易 …… 203
8.9 散段 …… 205
8.10 内存缓冲 …… 208
8.11 大件库 …… 210
8.12 URL 抓取 …… 213
8.13 任务队列 …… 214
8.13.1 任务 …… 216
8.13.2 任务执行 …… 217
8.13.3 队列 …… 218
8.14 后端 …… 220
8.15 能力 API …… 221
8.16 电子邮件 API …… 222
8.16.1 发送 …… 222
8.16.2 接收 …… 223
8.17 信道 API …… 224
8.18 小结 …… 226

第9章 标准包 227

9.1 格式包 228
9.1.1 格式输出 Printf 229
9.1.2 动词表 230
9.1.3 宽度和精度 231
9.1.4 报错 232
9.1.5 额外标记 232
9.1.6 格式输入 233
9.1.7 字符串格式 234
9.2 字节包 236
9.3 模板包 239
9.4 正则表达式包 245
9.5 时间包 253
9.6 超链接包 258
9.6.1 http 服务器和客户机 258
9.6.2 https 加密通信 260
9.6.3 Get 263
9.6.4 Post 265
9.6.5 Cookie 268
9.7 编码包 271
9.7.1 gob 271
9.7.2 json 275

附录 A Go 的安装和使用 277

附录 B EBNF 284

附录 C 中英术语对照表 291

第 1 章
快速入门

Well begun is half done.
（万事开头难。）
——谚语

本章通过几个具体的案例，解释 Go 语言程序的基本结构，让有一定功力的读者迅速上手，也让初学者对 Go 有个初步印象，便于后续章节的展开。

1.1 编辑和编译

大家都知道，学习编程的最佳方式就是动手编程。但这里有一个巨大的障碍。动手之前你要熟悉相关的工具，知道怎样使用编辑器，写出源代码。怎样操作编译器、链接器，得到可以执行的程序。另外还要考虑怎样运行这个程序，去哪里查看运行输出。并且还要知道，每一个步骤如果出了错，该怎样应对。

本书不打算手把手地教你使用这些必要工具。最好的办法是请教身边的专家。也可以参考其他资料，尤其是 Go 的正式发布网站 www.golang.org，自己慢慢摸索，并在使用中不断尝试。

为了帮助初学者克服这些与语言无关的技术障碍，我们特意整合了 Windows 上的 Acme 程序编辑器和 Go 编译器，并针对 Go 的编写、编译、执行和除错做了一个简单演示。Go 也可以运行在 Linux、FreeBSD 和 Mac OS X 上。读者可以根据自己的使用环境和习惯相应地调整。这些平台上 Go 语言环境的安装和使用，请参考附录。

读者可以从本书的支持网站 www.goplace.org 下载 acme.zip。它完全不需安装，直接解压到 Windows 的 c:\即可。Go 工具和运行环境要求安装在 c:\Go 下面。而 Acme 是在 c:\acme.app 下面，执行它里面的 acme.bat，就可以启动编辑器了。

我们看一下 acme.bat：

```
set GOROOT=c:\\Go
set GOPATH=c:\\acme-home\\
PATH=%PATH%;%GOROOT%\bin
c:
cd \acme.app\Contents\Resources
acme.exe
```

这里有几个重要的路径：

- GOROOT 是 Go 的正式软件所在路径；
- GOPATH 是我们自己的软件和第三方的 Go 软件所在路径；
- PATH 中必须包括 GOROOT 的 bin 才可以使用 Go 工具；
- acme-home 是 Acme 存放文件的根目录。

如果读者希望安装到其他路径下，则需要对 acme.bat 做相应的修改。

Acme 很神秘？其实编辑器就像钢笔铅笔，纯属个人喜好。一开始用哪个都写不出好字。慢慢地习惯了一种，就觉得它用着顺手了。所以，还没有习惯使用具体某一款编辑器的读者，不妨多试试，或许你会发现 Acme 是最出色的程序编辑器。顺便一提，Acme 和 Go 一样，是 Rob Pike 的作品。C 语言和 Unix 之父 Dennis Ritchie 也使用 Acme。

启动 Acme 后的第一感觉就是“它很不一般”，而且几乎也可以说是“不知所措”。菜单在哪里？帮助在哪里？怎么打开文件？在直接给出这些答案之前，我们有必要了解它的设计哲学。只有明白了我们真正需要的是什么，才晓得为何这样以及如何才能这样。

Acme 和 Unix 的设计哲学是一脉相承的，都是只提供少量基本工具以及组合它们的方法，而不是针对每一项需求都准备互不相干的几十个上百个选项。

其实，Acme 就是一个运行在类似 Unix 的虚拟机上的编辑器。这个虚拟机使用 Inferno 操作系统。这和 Emacs 编辑器运行在 Lisp 虚拟机上类似。只不过，Emacs 只能使用 Lisp 编程来配置编辑器的各项功能，而它的 Lisp 程序只为 Emacs 服务。Acme 则只是 Inferno 的一个程序。我们可以使用类似 Unix 的 sh 脚本，以及使用类似 Go 的 Limbo 编程语言，来编写 Inferno 的其他命令和程序，与 Acme 一起配合，完成所需的编辑工作。

这样的 Acme，也可以认为是 Inferno 操作系统的图形用户界面

(GUI)。只不过，相对于传统 WIMP（窗口、图标、菜单、指针）的 GUI 风格，Acme 简化为 WP（窗口和指针）——刚好也可以代表 Word Processing（文字处理）。这是因为，与其在几千年后再来发明一套类似象形文字的图标，为何不直接使用人们已经使用了很久的字词呢？与其干等着别人为你做好菜，让你伸手去点单，为何不去自己享受做菜的乐趣呢？难道程序员的本行不是编程吗？为何程序员所用的编辑器就不能自己编程定制功能呢？所以，学会了使用 Acme 的程序员，得到了这些问题的答案以后，就再也不能容忍传统的编辑器和 IDE 了。

到底如何才能打开文件？从哪里能得到帮助？

欢迎 Windows 世界的同学打开窗口，感受 Unix 世界的清新和煦。

Unix 的世界除了文件，就是文字。例如，在 Acme 的随便什么地方打入个斜杠字符“/”——这是 Unix 的根目录和目录分隔符，再用鼠标右键单击这个字符，就可以看到根目录下的文件和子目录。Acme 的根目录在 Windows 中是 c:\acme.app；而在 Acme 窗口的最上面一行输入一个点“.”，并同样使用鼠标右键单击这个点，就可以看到用户目录，在 Windows 中该目录为 c:\acme-home。而 Windows 的根目录是 c:\，这里把它定义为/me。这个定义可以在根配置文件/lib/sh/profile 中看到。要打开这个文件，当然可以在 Acme 的任意位置输入文件名再单击右键，也可以从/开始一直右键选择每个目录名，直到所需文件出现在一个窗口中。文件名所在的浅蓝行是窗口栏，在此处输入 me 再单击右键，就在 profile 文件里面找到第一次出现的 me。同样连续单击右键，就一直查找下去。由此可见，在 Acme 里查找文件和查找文字都只需用无名指单击即可。

说到窗口，Acme 的多窗口是并行排列的。默认是两列，每列内可以打开多个窗口，每个窗口都有一行天蓝色的标题栏。滚动条在左侧。滚动条和标题栏交汇的小方格，在窗口内容有变化时变成深蓝。按住鼠标左键移动此方格，可以重新排列窗口的位置。单击可以扩大窗口。而用

中键或右键单击，则可以整列显示此窗口，或再次显示其他窗口的标题栏。在滚动条内单击鼠标左键和右键可以上下翻页，或用中键单击和移动光标到所需位置。

Acme 最独特的是鼠标语言。它的三个鼠标键经单独或组合使用可以完成大量的操作。具体操作如表 1-1 所示。

表1-1 Acme鼠标键组合表

左	中	右	代表
1+	0	0	移动光标
1++	0	0	选词（整行或括号、引号括起来的段落）
1--	0	0	选择范围（选中部分高亮显示）
0	1+	0	执行命令
0	1—	0	红色高亮显示选中部分，执行命令
0	0	1+	打开文件或查找
0	0	1—	绿色高亮显示选中部分，打开或查找
1--	2+	0	删除选择内容
1--	2+	3+	粘贴回删除的内容，即复制
1+	0	2+	粘贴回删除的内容
2+	1+	0	将之前选中的内容传给命令执行

这里，1、2、3 是按键顺序，0 代表不按键。+是单击，++是双击，--是按键拖动鼠标。可以简单地总结为：左键选取文字，中键执行，右键查看选中内容，左中删，左右贴。试着练习几分钟之后，这些也就成了小脑指令了——无需大脑去想了。

如果没有三键鼠标，建议你去买一个，因为你可能以后会经常用到 Acme。当然，偶尔也可以用普通鼠标的滚轮代替中键，甚至在笔记本电脑上用 Ctrl 配合右键代替中键。或者使用表 1-2 所示的 Ctrl 键组合执行常用命令。

PDG

表 1-2 Acme 键盘命令表

Ctrl键组合	执行操作
a	转至行首
b	返回上页
c	复制
d	补全
e	转至行尾
f	转至下页
h	退格
i	Tab
k	左移
l	右移
m	换行
n	下移
o	上移
p	转至文件尾
q	转至文件头
s	保存
u	删整行
v	粘贴
w	删除
x	剪切
y	重复
z	撤销

习惯 Acme 的关键就是鼠标命令，而使用鼠标命令的方便之处在于，Acme 的所有文字，都是可以用鼠标命令来操作的。也就是说，不管命令位于何处，是在标题行或者一个窗口里，只要在其上按中键或按住中键扫过，都是执行保存文件的操作。而在其上按右键，都是执行查找的操

作。这样，我们把一些最常用的命令放在每个窗口的标题行，只要中指轻轻一点就可以了。这些命令和一些不太常用的命令如表 1-3 所示。

表1-3 Acme鼠标命令表

中文	英文	代表
剪	Cut	剪切选中的内容
拷	Copy	复制选中的内容
贴	Paste	粘贴上次剪切的内容
新	New	在新窗口中打开选中的文件
关	Del	关闭此窗口
悔	Undo	撤销上次修改
不悔	Redo	重复上次撤销
存	Put	保存文件
读	Get	重新读取文件
查	Look	查找选中的词语或文件
令	Edit	执行选中的编辑命令
编	Compile	编译Go文件
	Zerox	在新窗口打开同一文件
	Putall	保存所有窗口内容
	Dump	保存窗口状态，下次自动打开
	Newcol	增添一列
	Delcol	删除一列
	Exit	退出Acme（不保存）
	win	命令控制窗口
	man	帮助手册
	g	查找C和Go文件的内容
	mkdir、touch、rm	新建目录，新建文件，删除目录或文件

一些常用的编辑命令可以保存在/guide 文件里。每次选中一条命令，用鼠标中键加左键单击在某个窗口的“令”字符上，就可以对此窗口文件的内容执行编辑操作了。例如，选择几行程序，执行`,s/^/	/g` 命令就可以在每个行首加一个 tab 字符，也就是整体右移。而执行`,s/^//g`就可以去掉行首的 tab，整体左移。这些编辑命令，可能是初学者的最大障碍，却也是 Acme 最强大、最灵活的地方。所有的帮助文件，都在/man 目录下面。如果要学习 Acme 的编辑命令，读者可以在 Acme 里执行 `man acme`；如果要看 shell 的编程，执行 `man sh` 即可；而右键单击类似`sh-std(1)`这样的格式，也可以帮助我们看到帮助文件。这是因为，我们在 lib/plumbing 里，定义了右键的规则，让这一点也能具备可以编程的一点智能。

不多讲了。再次重申，编辑器的选择是个人喜好，选择什么都和 Go 语言无关。

PDG

1.2 世界，你好！

按照惯例，介绍所有语言时使用的第一个程序都是“Hello，World!”。不过，Go语言的这个程序输出的是中文，而且，已经作为Acme的第一个窗口，等待我们编译。

```
package main

import "fmt"

func main() {
    fmt.Println( “你好” )
}
```

读者可以用鼠标中键单击“编”开始编译程序。为了演示，我们故意把“你好”的引号设为中文引号。编译会出错。用鼠标右键单击+Errors窗口中的main.go:6部分，会直接跳到出错行。修改，保存，编译。没错了，“你好”会显示在另一窗口。Acme编辑编译练习完毕。Go也和我们打了招呼。

我们大概了解一下“编译”命令的基本原理。

(1) Acme会使用窗口标题栏的文件名作为参数，调用/dis/goc。

(2) /dis/goc是sh脚本，它先用gofmt格式化此窗户的文件。

(3) 如果没错，再用go run编译执行此文件。

(4) 如果有错，整理出错行号输出。

这样，一个Acme的鼠标动作，就同时解决了Go程序的标准格式的编辑、编译、运行和出错管理。而一切都是简单透明的，完全不像别的编程语言那样，必须利用一个庞大的IDE。

现在解释程序自身。所有 Go 程序，都由函数和变量构成。一个函

数包括一系列的语句，指明要执行的操作，以及执行操作时存放数值的变量。我们这个程序的函数名称是 main。尽管名称没有限制，但 main 包的 main 函数是每一个可执行程序的入口。而“包”则包装了相关的函数、变量和常量，要用 import 导入，才可以使用。例如，我们导入 fmt 包，才可以使用它的 Println 函数。

双引号里的"你好"是 Go 的字符串常量。和 C 的字符串不同，Go 程序不可以改变字符串的内容。但作为参数传递给 Println 函数时，字符串的内容没有复制，而仅仅是将其地址和长度作为字符串的值，复制给参数。也可以说，Go 的参数传递，都是值的复制，而没有其他语言的那种间接的引用参数。

1.3　自我复制

newgo 是一个能生成新程序的程序。有趣的是，生成的这个新程序几乎就是 newgo 自己：

```
/* */
package main
import (
    "fmt"
)

var ()
const ()
func f() {}
func main() {
    fmt.Println(s)
}
consts = `/*   */
package main
import (
    "fmt"
)

var ()
const ()
func f(){}

func main() {
    fmt.Println(s)
}
`
```

包 package，函数 func，变量 var，常量 const，形成了一个 Go 程序最外围的结构。

package 后面是包名，用于声明这个文件的函数、变量和常量都属

于这个包。如果函数名、变量常量名的第一个字母是大写，在用 import 导入这个包之后，就可以直接使用这些函数、变量和常量。例如，我们导入 fmt 包可以使用它的 Println 函数，而 fmt 包里还有很多函数、变量和常量，它们就像 main 包里的 s 一样，不是以大写字母开始的，那么它们就是为这个包所私有，不可以被其他包使用。这么一个简单的命名规则，便实现了以包为单位的数据封装。

func 的后面是函数名。它与小括号括起的参数和大括号括起的一系列的执行语句，构成一个函数。我们的 main 函数没有参数，而且只有一条执行语句，就是调用 fmt 包的 Println 函数。“调用”就是向被调函数的参数赋值，执行被调函数的语句，并使用其返回值。这样，函数就封装了可以被反复调用的一些语句，并能根据不同参数的值，返回不同的结果。程序语言中的函数概念接近数学的函数。例如使用 sin(x) 可以得到对应不同 x 值的正弦值，而且 Go 的 math 包的 Sin 函数完成的就是这个功能。但有些函数，例如 fmt 包的 Println 函数，其功能不仅是返回输出了多少字节和有没有出错，而是要实实在在地把参数的值输出在某个地方。这就不再是纯粹的数学函数了。这个函数有副作用，可以使用参数以外的信息，可以改变返回值以外的信息。而要了解这些额外的信息，除了仔细分析函数的源代码之外，更重要的是要靠文档告诉使用它的程序员。

Go 语言非常重视高质量的文档。在 Acme 中用中键单击 godoc，或者在命令行运行 godoc -http=:6060，并用浏览器访问 http://localhost:6060，都看到类似 Go 语言主页的内容。如果只需查看某个包的某个函数说明，例如 math 包的 Sin 函数，可以直接运行 godoc math Sin。这些函数说明都是从 Go 的源代码和注释中直接抽取的。

Go 的注释有两种。第一种就是用/**/括起的大段文字。另一种是从//开始到行尾的文字。注释文字被编译器忽略。

const 和 var 分别声明常量和变量。常量不仅仅是指不可改变的值，它还告诉编译器这个名字可以直接被它所代表的值替换。而变量则代表在程序执行时，由编译器预先分配的一个内存地址。这个地址处的内存可以被反复读写。

由于编译器可以帮助分配变量地址、替换常量、调用函数、翻译机器语言和分析错误，程序员才得以从直接操作机器语言的任务中解放出来，使用更类似自己思维用语的高级语言。Go 语言就比较接近简单的英语，而为包、函数、变量和常量等取一个好的名字，能使 Go 程序更加清晰易懂。

我们看到字符串 s，除了像"你好"那样使用双引号括起，还可以像本程序一样，用两个反引号括起。它们的区别是后者可以跨行，也就是可以包括换行符，而前者需要用\n 这样的转义字符表示换行。

newgo 的执行结果，几乎但不完全是自身的源代码。作为练习，读者可以修改程序，使它的输出和自身代码完全一样。

提示：反引号 `的编码是 0x60，fmt.Print("%s%c%s%c",s,c,s,c) 可以按顺序逐个输出给定的字符串和字符。

1.4　猜数游戏

下一个程序能猜到你想的数，例如：

请想一个 0～100 的整数。

该数小于或者等于 50 吗?　(y/n) n

该数小于或者等于 75 吗?　(y/n) y

该数小于或者等于 63 吗?　(y/n) n

该数小于或者等于 69 吗?　(y/n) y

该数小于或者等于 66 吗?　(y/n) n

该数小于或者等于 68 吗?　(y/n) y

该数小于或者等于 67 吗?　(y/n) n

该数是 68

这个程序仍旧只有一个函数——main。它让我们有机会介绍变量声明、赋值、算数表达式、循环、判断和输入/输出。

```
package main

import "fmt"

func main() {
    min, max := 0, 100
    fmt.Printf("请想一个%d~%d的整数. \n", min, max)
    for min < max {
        i := (min + max) / 2
        fmt.Printf("该数小于或者等于%d 吗?(y/n)", i)
        var s string
        fmt.Scanf("%s", &s)
        if s != "" && s[0] == 'y' {
            max = i
        } else {
```

```
                min = i + 1
            }
        }
        fmt.Printf("该数是%d\n", max)
    }
```

Go的变量使用前必须声明，通常是在一个函数的最开始处声明，但也可以在使用的地方声明，而且通常不需要使用var，仅仅通过:=赋值，Go就可以引申声明变量的类型。例如：

```
min, max := 0, 100
var s string
```

min和max通过赋值引申声明为int类型，s用var明确声明为string类型。

类型表示变量的取值范围和精度。int是整型，该类型的值可以是0，也可以是一个不大的正负整数，只要它能装入至少32位的机器字。如果需要明确整数的字宽，可以使用int8、int16、int32或int64类型。同样，Go还有float32和float64类型，可以用来表示一个32位和64位符合IEEE-754规定的正负小数。而uint、uint32和uint64等的u，表示无符号unsigned，也就是只能是0和正数，但因为代表负数的位也用来表示正数了，所以这些变量的正数取值范围就扩大了一倍。例如，int8可以表示的整数范围是-128~+127，而uint8的范围是0~255。Go同时规定byte字节类型就是uint8，而rune类型表示Unicode字符，是int32。

string是字符串类型。似乎它应该是一串字符，但实际上，它的内部表示是字节数组，可以用s[i]这样的方式得到其第*i*个字节。但字符串的内容是不可以改变的，也就是说，用s[i]=0这种方式为它的第*i*个字节赋值，编译器会报错。Go使用字节而不是字符作为字符串的单元，是因为Go的字符采用的是UTF-8编码，是不等长的。英文字母数

字可以是 1 个字节，希腊字母要用 2 个字节，而汉字等大字符集，必须使用 3 个以上的字节才能表示。所以字符串取其最小单位，而在用到对应字符时，才去检查需要几个对应的字节。

Go 的:=和=都用来赋值。而:=也同时引申声明了变量的类型。Go 可以在同一行对多个变量同时赋值。例如为此处的 min 和 max，分别赋值 0 和 100。变量的有效范围，也就是它的作用域，从它的声明开始，到包含它的最内层的块。块是由大括号括起的语句。这样，min 和 max 在整个 main 函数内有效，而变量 i 和 s，只在 for 语句块里有效。执行离开块后不再有效的变量，或者随运行栈一起消失，或者留到以后被 GC 回收。

for 语句用于循环执行一个块，直到其条件不再满足。if 语句也是根据条件，判断是执行后面的块，还是执行 else 的块。else 块可以没有。

“条件”是用于比较的逻辑表达式。表达式由变量、常量和操作符组成。此处的<代表“小于”，还有>、>=、<=分别代表“大于”、“大于或等于”、“小于或等于”。“等于”操作符是==，“不等于”是!=。

比较的结果是布尔 bool 类型的值：真 true，假 false。布尔值可以用在逻辑表达式中。&&是只有两者都是真时操作结果才是真，否则结果是假。||是只有两者都是假时结果才能假，否则是真。! 反转真假。例如此处的 if 条件：

```
s != "" && s[0] == 'y'
```

先是比较 s 不等于""，如果字符串是空的，也就是假，&&的操作结果一定也是假，也就没有必要去比较后面的条件了。这和 C 类语言是一样的，也是必要的。因为此时如果执行后面的比较操作，s[0]代表的第一个字节，在空字符串中是不存在的，会出现错误。所以，通过逻辑表达式&&，前面的比较可以保护后面的操作不会出错。

Go 的数值计算使用通常的+和-代表两个数的加减法。而*和/代表乘除法。计算结果的类型与操作数类型相同。所以奇数除 2 的小数部分被舍弃。3/2==1。

fmt 包中 Printf 的 f 代表 format（格式）。Go 的变量函数名倾向使用简单的缩写，类似数学公式，避免使用冗长的文字。

格式是指用一个符号代表一个值的类型和输出位置。此处的“%d ~ %d”表示分别用后面的 min 和 max 两个整数值替换%d 所在的位置。

同样，fmt 的 Scanf 是按格式扫描输入的字符，%s 代表等待输入一个字符串，也就是以空格或者换行键结束的一行字符，赋值给&s。& 是取址运算符。表示 s 的地址。注意，s 不是字符串的内容，Go 的字符串是不可以修改的。它仅仅是把 s 变量的地址传递给 Scanf 函数的参数。需要传递地址，是因为 Go 的参数是值的副本，Scanf 要间接通过这个地址值，使 s 指向输入的那个字符串。

当然，到目前为止，我们对程序的讲解，仅是 Go 语法的介绍，而没有分析算法。数据结构和算法是计算机科学的基础和核心，本书不会去深入探讨。当需要了解时，也仅仅是一笔带过。

此处猜数使用的是二分查找（binary search）算法。也就是每次的判断，都会使搜索的范围减小一半。所以，100 个数要猜 7 次，因为 $2^7=128$。而 100 万个数，猜猜看要猜多少次？

20 次就可以了。二分查找很快，但一次就写出正确的二分查找程序却是出了名的难。建议读者自己动手试着重写一遍。这也是 C 和其他语言都提供二分查找库函数的一个原因。Go 可以使用 sort 包的 Search 函数：

```
package main

import (
    "fmt"
```

```
    "sort"
)

func main() {
    fmt.Println("Pick a number from 0 to 100.")
    fmt.Printf("Your number is %d\n",
        sort.Search(100, func(i int) bool {
            fmt.Printf("Is your number <= %d? ", i)
            var s string
            fmt.Scanf("%s\n", &s)
            return s != "" && s[0] == 'y'
        }))
}
```

现在不需要理解这个程序。

1.5 图灵机

Go 作为高级语言，当然是图灵完备的。我们用 Go 来写一个最简单的图灵机（Turing Machine），使用脑操编程语言，下面的程序可以输出 hi：

```
++++++++++[>++++++++++<-]>++++.+.
```

此编程语言仅有 7 条指令，理论上和任何图灵完备的语言等价。但程序员使用不同语言的表达能力和效率，是有云泥之分的。这也是人们总是在探索新的语言、提高表达效率的原因。

这 7 条指令是：

+——使当前数据单元的值增 1；

-——使当前数据单元的值减 1；

>——下一个单元作为当前数据单元；

<——上一个单元作为当前数据单元；

[——如果当前数据单元的值为 0，下一条指令在对应的]后；

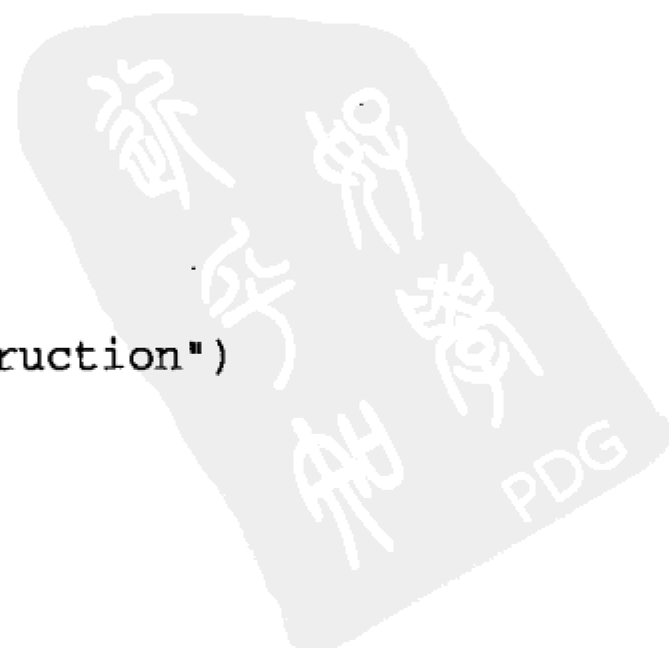
]——如果当前数据单元的值不为 0，下一条指令在对应的[后；

.——把当前数据单元的值作为字符输出。

这样，当前单元的值加 10，作为[和]的循环变量，>到下一个单元，也加 10，<-使循环变量减 1，循环 10 遍，再加 4，得到 104，是字符 h 的 UTF-8 编码，输出，再加 1，输出 i。

```
package main

import "fmt"

var (
    a      [30000]byte
```

```
        prog  = "++++++++++[>++++++++++<-]>++++.+."
        p, pc int
    )
    func loop(inc int) {
        for i := inc; i != 0; pc += inc {
            switch prog[pc+inc] {
            case '[':
                i++
            case ']':
                i--
            }
        }
    }
    func main() {
        for {
            switch prog[pc] {
            case '>':
                p++
            case '<':
                p--
            case '+':
                a[p]++
            case '-':
                a[p]--
            case '.':
                fmt.Print(string(a[p]))
            case '[':
                if a[p] == 0 {
                    loop(1)
                }
            case ']':
                if a[p] != 0 {
                    loop(-1)
                }
            default:
                fmt.Println("Illegal instruction")
            }
            pc++
```



```
            if pc == len(prog) {
                return
            }
        }
    }
```

程序一开始的变量 a 是我们图灵机的数据内存，prog 是指令内存，p 和 pc 分别是这两个内存的指针，代表当前数据单元和当前指令。

函数 loop 执行[和]指令，移动指令指针 pc。这里用到了三段式的 for 语句，也就是：

```
for 初始化;判断;增值 {
```

初始化部分在循环开始前执行一次，通常是给循环控制变量一个初始值，然后每次判断如果为真，就执行大括号的语句块一次，再执行增值部分，再判断、执行、增值，直到条件为假，才跳过大括号里的代码块，继续执行它后面的语句。

switch 是 Go 的单项选择语句。它根据后面的值，选择执行大括号里的某一个 case 分支。同样的 for 和 switch 语句，也出现在 main 函数里。但那里的 for 没有三段式，所以会一直循环，直到 return 结束。而那里的 switch 有一个 default 分支，当其他的 case 都不是 switch 的值时，执行 default 分支。

此程序还用到了++和--运算符，给变量加 1 和减 1。例如 p++就是 p = p + 1，也可以写为 p += 1。这里，函数 loop 的 for 语句增值部分的 pc += inc，就是 pc = pc + inc 的缩写。

注意++和--是单独的语句，不是表达式，不可以用在其他语句里。像*p++=*q++这种高明的 C 语句，在 Go 里是不能用的。目的是避免语义误导，给程序员少一点犯错的机会。

而这种加减 1 的操作，主要用来移动数组的下标。例如，p 和 pc 分别是数组 a 和 prog 的下标，这样 a[p]和 prog[pc]就分别是这两个数

组对应下标所表示的单元中的值。

至此，我们应该可以理解此程序了。作为练习，请读者拿出纸笔，画一个包含 33 个格子的带子，逐一填上 prog 的每个字符，这就是 prog 数组，也是图灵机的指令内存。再画一个至少包含两个格子的带子，作为数组 a，也就是图灵机的数据内存，然后，执行每一条指令，看数据格子是怎样更新的。我们明白了图灵机，也就明白了计算机的理论基础。

在某些人看来，物理宇宙也是如此运行的：一切皆是宿命、神旨、历史规律。但还有人认为，在这个无限的宇宙里，没有不可能，再小的几率也一定会发生，一切都不是确定的，所以机械的图灵机理论不适用。更何况，作为物理宇宙的采样观察者，人类太渺小，采样太少，不可能正确解读物理宇宙的信息的。人还是应该先观察明白自己身边的小事情，看怎样写指令才能体现自己和人类整体的价值，而自己又是不是改变数据的那条指令呢？

好了，哲学人生观的讨论对 Go 编程毫无帮助，我们还是回顾一下都学会了哪些 Go 语言基础知识：包、函数、变量、常量、赋值、类型、字符串、数组、表达式、比较运算符和逻辑运算符、if、else、for、switch、case、++和--。够用了。下面我们直接来看几个完整的工具程序。

1.6 排版工具

本书的写作一开始就面临一个实际的问题：什么样的文件格式最适合中文写作？在用过多种工具后，我决定还是自己动手编写一种最合适的工具。因为输入中文时，标点也是中文的，必须切换到英文输入法，这太麻烦。而这些标点作为标记，是排版工具必不可少的。会写程序的好处就是，可以自己编写适合的工具，减少这种麻烦。

我们的目的是要把一个简单的标志文件，转换为 HTML 格式。首先不要给自己制造麻烦。一种非常简单的格式，就能基本满足本书的版式要求。

我们作出如下规定。

(1) 段落之间用空行分隔，每段的前四个字符如果是

　　以 01 开始，则后续数字对应 HTML 的 h1 到 h6；

　　以 020 开始，则插入文件内容到 HTML 的 pre。

(2) 如果一段以空白字符开头，则作为 HTML 的 pre。

(3) 如果都不是，就是简单的段落，对应到 HTML 的 p。

简化了问题，程序也就非常简单：

```
package main

import (
    "flag"
    "fmt"
    "html"
    "io/ioutil"
    "os"
    "strings"
)
```

```
var (
    esc = html.EscapeString
    tflag *bool = flag.Bool("html", true, "html output")
)

func main() {
    flag.Parse()

    in, _ := ioutil.ReadAll(os.Stdin)
    out := parse(string(in))
    for i := range out {
        fmt.Println(out[i])
    }
}

func parse(in string) []string {
    s := strings.Split(in, "\n\n")
    for i := 0; i < len(s); i++ {
        t := s[i]
        if t == "" { // skip empty lines
            continue
        }
        if t[0] == '\n' { // skip extra newline
            t = t[1:]
        }
        if len(t) < 4 {
            s[i] = para(t)
            continue
        }
        switch t[:2] {
        default:
            s[i] = para(t)
        case "01":
            s[i] = header(t)
        case "02":
            s[i] = importFile(t)
        }
    }
```

```
    return s
}

func para(s string) string {
    if !*tflag {
        return s
    }
    s = esc(s)
    if s[0] == ' ' || s[0] == '\t' {
        // replace a tab with 4 spaces
        s = strings.Replace(s, "\t", "    ", -1)
        return "<pre>" + s + "</pre>"
    }
    return "<p>" + s + "</p>"
}

func header(s string) string {
    if !*tflag {
        return "\t" + s[4:]
    }
    t := string(s[2])
    s = esc(s[4:])

    s = "<h" + t + ">" + s + "</h" + t + ">"
    return s
}

func importFile(s string) string {
    b, err := ioutil.ReadFile(s[4:])

    var t string
    if err != nil {
        t = fmt.Sprintf("Error: %v", err)
    } else {
        t = string(b)
    }
    return para(t)
}
```

如果将文件保存为 ma.go，在命令行使用 go build ma.go 可以得到可执行文件 ma。我们自己遵照规则写个测试文件 test.ma，试着运行 ma < test.ma > test.htm。如果在浏览器看到的是一堆乱码，记得把编码格式改为 UTF-8。

main 函数的第一条语句使用 flag 包的 Parse 函数得到命令参数。这在 tflag 中给出定义。如果使用 ma -html=false，则输出的是文本格式，而不是 html 格式。

ioutil 包的 ReadAll 函数，读取标准输入 os.Stdin 的内容到变量 in。下划线是个特殊的变量，称为“空变量”，它的值不被使用。但因为 ioutil.ReadAll 函数要同时返回出错信息，我们用空变量代表不去理会这个可能的错误。可以比照下面的 importFile 函数使用的 ioutil.ReadFile，看怎样处理错误。

变量 in 的值是字节数组[]byte。在 Go 里通常不需事先声明变量的类型，因为使用:=，编译器可以推断变量的类型。在下一行我们使用 string(in)把这个字节数组转换为字符串类型。虽然 Go 字符串的底层结构是字节数组，但编译器规定字符串类型的值是不可以修改的，所以这里要明确告诉编译器进行类型转换，转换过程可能伴随着内存的复制。这是因为字节数组的内容是可以修改的，Go 要确保程序无法通过变量 in 来修改字符串，就必须复制。当数据量大时，这种类型转换附带的复制操作会影响执行效率。

另一方面，转换后的字符串，传递给函数 parse 的参数时，就不需要再重新复制。在 Go 里，所有的参数都是赋值传递的。这样我们明白了此处实际赋值的是字符串内存的地址和长度，而不是那块内存地址的内容。

这同时也解释了赋值的概念。使用变量名字是为了让编译器自动在内存中分配一个地址，用来存放这个变量的值。而变量的类型告诉编译

器要分配的地址占用多少字节的连续空间。例如 var e float64，程序运行时占用 8 字节的内存，来存放一个 float64 类型的值。然后 e = 2.71828 把值 2.71828 写入 e 所代表的这 8 字节的地址，这就是赋值。

回到我们的程序。parse 函数会返回一个字符串数组[]string，并将其赋值给变量 out。随后，我们用 for 循环，逐行输出 out 的内容。此时循环控制靠的是 range 这个关键字。它在每次循环时，读出 out 数组下一个单元的下标，赋值给此处的变量 i。

数组是 Go 的组合类型之一，代表着内存中同一类型的数据单元的连续分配。例如[6]byte 代表 6 个连续的字节。[4]string 是 4 个连续的字符串的“头”，每个头包括对应字符串“内容”的地址和它的长度。

数组在 Go 中很少直接使用，因为数组在函数传递和赋值时，复制的是全部内容，而不是像字符串那样，仅仅复制“头”。程序中出现的[]byte 和[]string 都不是 Go 的数组，而是 “切片”。切片是数组的“头”，通过函数传递切片和为切片赋值时，只是复制其对应数组的地址、长度和容量。尽管在用法上相似，但切片和数组是不同的类型。

接下来，parse 函数把输入的字符串分段转换，返回一个字符串切片。分段靠的是 strings 包的 Split 函数。字符串中的\n 代表换行。连续两个换行，就是有一个空行的意思。将分段的结果赋值给 s，s 是字符串切片类型。下面又是一个 for 循环，逐一处理 s 切片的每个字符串。

这里使用里 for 循环的第二种格式，它允许把变量初始化、判断、增量放在一起，从而可以明显地看出循环怎样开始，怎样结束，怎样选择下一个数据单元。

len 是 Go 的内置函数，返回数组、切片和字符串等的单元个数，也就是“长度”。单元从 0 开始连续编号，赋值给变量 i，这样 s[i]得到的就是从 0 开始第 i 个单元的内容，也就是每段的字符串，赋值给 t。如果 t 的长度小于 4，表示此段的开头不够我们标记所要求的 4 个数字，

可以直接用 para 函数进行段落转换，然后 continue 继续下一个 for 循环，也就是执行 for 语句的增量部分 i++后继续。

switch 是 Go 的分支语句。它根据后面的条件，也就是 t[:2]的内容，选择执行大括号里的某个 case 分支。t 是字符串。[:2]就是截取第 0 个单元直到第 2 个单元之前的内容。如果是 01，使用 header 函数转换为标题行。如果是 02，使用 importFile 函数，读文件并插入。否则，switch 选择 default 分支，直接使用 para 函数。

在 para 函数中，如果命令行标志 tflag 的值为假，就不再转换为 html，而是直接输出 s。注意，tflag 是指针类型的变量，它的值需要用*间接得到。

变量 esc 是 html 包的 EscapeString 函数。在 Go 里，函数作为值可以直接赋值给同一类型的变量。

如果一行是以空格或者 tab 字符开始，我们用 strings 包的 Replace 函数把每个 tab 字符转换为 4 个空格，再放入 pre 标签。否则，s 字符串前后加上 HTML 的 p 标签，返回新字符串。

再看 header 函数。s[2]拿到的是字符串 s 从 0 开始的第 2 个字节。注意我说的是字节而不是字符。因为 Go 的字符串是使用 UTF-8 编码的 Unicode 字符序列。这也是必须告诉浏览器我们输出的 HTML 文件是 UTF-8 编码的原因。否则，浏览时很可能看到的中文都是乱码。这同样也是为什么如果我们的 Go 程序源代码有中文字符串，存文件时必须使用 UTF-8 编码格式，否则编译可能出错。

最后是 importFile 函数。第一行 ReadFile 读取文件的内容，文件名是从 s 的第 4 个字节开始到 s 的结尾。var 声明 t 是字符串变量，同时赋值为空。放在此处声明，是为了说明变量在使用前必须声明。可以像这里一样使用 var 声明，也可以在第一次赋值时使用:=声明。

这样我们完成了简单的自定义格式到标准 html 文件格式转换的工具

程序，同时熟悉了 Go 中常用的控制语句。因为 Go 可以在 Windows、Linux 和 Mac OS X 等多种操作系统上编译运行，并且得到的是一个完整的可执行文件，用 Go 编写通用软件工具，就显得很方便，很有吸引力。

下一个示例将展示怎样用 Go 把浏览器、服务器和命令行工具有效地组合在一起。

1.7 游乐场

这个程序改编自 Go 语言安装包的 misc/goplay。目的有两个：一是提醒大家，Go 语言本身开放全部源代码，从中我们可以参考大量高质量的 Go 程序；二是通过此程序可以一窥 Go 语言对网络并发编程的支持。

游乐场是一个 Web 服务器。运行此程序，然后通过浏览器访问 http://localhost:1234，就可以连接到此服务器，在文本栏输入 Go 的 main 包的 main 函数，单击 Run，就可以自动编译、链接、执行输入的源代码，并同时显示结果：

```
package main

import (
    "io"
    "log"
    "net/http"
    "os"
    "os/exec"
    "strconv"
)

var uniq = make(chan int)

func init() {
    go func() {
        for i := 0; ; i++ {
            uniq <- i
        }
    }()
}

func main() {
    if err := os.Chdir(os.TempDir()); err != nil {
```

```
        log.Fatal(err)
    }

    http.HandleFunc("/", FrontPage)
    http.HandleFunc("/compile", Compile)
    log.Fatal(http.ListenAndServe("127.0.0.1:1234", nil))
}

func FrontPage(w http.ResponseWriter, _ *http.Request) {
    w.Write([]byte(frontPage))
}

func err(w http.ResponseWriter, e error) bool {
    if e != nil {
        w.Write([]byte(e.Error()))
        return true
    }
    return false
}

func Compile(w http.ResponseWriter, req *http.Request) {
    x := "play_" + strconv.Itoa(<-uniq) + ".go"

    f, e := os.Create(x)
    if err(w, e) {
        return
    }

    defer os.Remove(x)
    defer f.Close()

    _, e = io.Copy(f, req.Body)
    if err(w, e) {
        return
    }
    f.Close()

    cmd := exec.Command("go", "run", x)
```

```
        o, e := cmd.CombinedOutput()
        if err(w, e) {
            return
        }
        w.Write(o)
    }

    const frontPage = `<!doctype html>
    <html><head>
    <script>
    var req;
    function compile(){
        var prog = document.getElementById("edit").value;
        var req = new XMLHttpRequest();
        req.onreadystatechange = function() {
            if(!req || req.readyState != 4)
                return
            document.getElementById("output").innerHTML
= req. responseText;
        }
        req.open("POST", "/compile", true);
        req.setRequestHeader("Content-Type", "text/plain;
charset =utf-8");
        req.send(prog);
    }

    </script>
    </head>
    <body>
    <textarea rows="25" cols="80" id="edit" spellcheck=
"false">
    package main
    import "fmt"
    func main() {
        fmt.Println("hello, world")
    }
    </textarea>
    <button onclick="compile();">run</button>
```

```
<div id="output"></div>
</body>
</html>
`
```

Web 服务器的功能是通过 net/http 包里提供的函数实现的。而 os/exec 包的函数则用来执行 Go 的命令行。strconv 包的函数用于字符串转换。这些随 Go 语言一起提供的标准包，给我们提供了丰富的、可以信赖的大量函数，用于完成一些常用功能。很多时候，我们自己写的程序，就是把这些标准包里的函数有机的结合在一起，达到我们的需求。

var 可以声明包变量。此包的函数都可以读写这个变量。uniq 的类型是 chan int，并由 Go 内置的 make 函数分配内存。chan 是程道（channel）类型，用来在去程（goroutine）之间传递值。chan int 可以存放 int 型数据，Go 会保证不同的去程顺序地读写此程道，而不会产生数据冲突。去程是 Go 的运行器可以独立调度的函数，当计算机有多核 CPU 时，去程是并行运行的。

init 函数在 main 函数之前自动调用，完成程序的初始化。此处 go 语句后的无名函数 func()启动一个新的去程执行。这个函数执行 for 循环，把从 0 开始的一系列整数逐次用<-运算符发送给程道变量 uniq。由于 uniq 只能存放一个整数，在另一个去程取走这个整数之前，发送去程会阻塞，也就是暂时停止执行。

由于 go 语句把 func()交给另一个去程执行，我们的 main 程序，作为一个独立的去程，得以继续执行。main 函数先使用 os 包的 Chdir 把工作目录换到一个临时目录，用来存放编译的结果。然后，用 http 包的 HandleFunc，分别针对访问 URL 的路径，注册 FrontPage 和 Compile 函数。

最后 http 包的 ListenAndServe 函数，运行一个 Web 服务器。由于上面的 HandleFunc 函数已经注册好了 URL 的处理函数，当浏览器访

问 http://localhost:1234 时，FrontPage 函数被 http 包的 ListenAndServe 调用。当然，作为 Web 服务器，ListenAndServe 能处理大量连续的访问，而不必等待每个 FrontPage 或其他 HandleFunc 注册的处理函数执行完毕。这也要求我们的处理函数，必须考虑并发执行的问题。

FrontPage 函数只是显示静态的字符串，并发执行没有任何问题。w 变量是 http.ResponseWriter 界面（interface）类型。我们可以直接使用这个变量的 Write 方法，把转换为字节切片的 frontPage 字符串常量，通过 HTTP 协议发回给浏览器显示。

我们稍微介绍一下 frontPage 的内容。script 标签里的是一段 JavaScript 程序定义的 compile 函数，使用 XMLHttpRequest 把 edit 的内容发送给服务器，也就是我们的 ListenAndServe 函数。由于其发送的 URL 是"/compile"，在 HandleFunc 中注册的 Compile 函数会被调用，来处理发送来的 edit 的内容。Compile 处理的结果，发回给 XMLHttpRequest 的 onreadystagechange 回调函数，用来在 output 中显示。而在下面 body 中，就有 edit 和 output 的说明，分别是 HTML 的 textarea 和 div 标签。同时，还有一个 button 标签，单击时，其 onclick 会回调上面提到的 JavaScript 的 compile 函数。另外，edit 的内容是一个 Go 的 hello world 函数。

这样，我们回过头来看 Go 的 Compile 函数。它的第一行就使用了 uniq 程道。因为程道的互斥可以实现有大量 Web 请求时，并发执行 Compile，实际上程道使并发的去程能够排队按顺序调度。这样，每次从 uniq 拿到的都是不重复的整数，我们把它用 Itoa 函数转换成字符串，拼装成此次 Compile 使用的独特临时文件名。

os.Create 创建文件，如果出错，使用 err 输出错误信息到浏览器，然后立即返回。尽管 Go 语言也有派错（panic）和恢复（recover）异常处理机制，但不鼓励使用它们来处理可以预知的出错情况，而是希

望程序能明确地立即处理并返回。这就是接下来使用了大量类似的出错检查和处理语句的原因。

压后（defer）语句是 Go 语言的一个特色，它注册的一个函数，在它所在函数返回时会被自动执行。所以，不管 `Compile` 函数如何返回，`os.Remove` 都会执行，删除 `Create` 创建的文件。`f.Close()` 也会执行，关闭并释放 `Create` 占用的资源。

`io.Copy` 能把 `Body`，也就是把从 JavaScript 的 `compile` 函数以 POST 方式发送过来的 `edit` 的内容，写入文件 f。再次使用了界面，但我们此处不多解释。

`exec` 包的 `Command` 函数，执行命令行程序，并得到其输出结果。我们用这种方式执行 Go 的 `go run` 命令，并把它的执行结果发送回浏览器。

1.8 位钱

本节的这个例子展示一点点高精度数学包 math/big、一点点散列包 hash、一点点加密包 crypto，还有一点点测试包 testing 的知识。这里不介绍 bitcoin 协议和算法——尽管它们很有趣，而是试图指出，Go 对多种操作系统的支持，是实现这种跨平台应用的理想语言。

位钱（bitcoin）是一种使用加密手段制作的分布式电子货币。它最初于 1998 年由 Wei Dai 提出，并由中本聪（Satoshi Nakamoto）及其伙伴，于 2009 年在 Windows、Linux 和 Mac OS X 上实现。这些客户端软件帮助用户管理电子钱包，钱包里面包括一系列的公钥加密密钥对（public-key cryptographic keypair）。每个密钥对的公钥（public key）转化为一个位钱地址，作为交易的接收地址。这个地址是可以供人使用的，大约 33 个字符，使用的是 Base58 的编码方式。而每个私钥（private key）用来签发发自此钱包的交易。

我们看看如何使用 Go 来完成位钱地址所需的 Base58 编码：

```
package bitcoin

import (
    "math/big"
    "strings"
)

const base58 = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabc
defghijk lmnopqrstuvwxyz"

func EncodeBase58(ba []byte) []byte {
    if len(ba) == 0 {
        return nil
    }
```

```
        //Expected size increase from base58 conversion 25
approximately 137%,use 138% to be safe
        ri := len(ba) * 138 / 100
        ra := make([]byte, ri+1)

        x := new(big.Int).SetBytes(ba) // ba is big-endian
        x.Abs(x)
        y := big.NewInt(58)
        m := new(big.Int)

        for x.Sign() > 0 {
            x, m = x.DivMod(x, y, m)
            ra[ri] = base58[int32(m.Int64())]
            ri--
        }

        //Leading zeros encoded as base58 zeros
        for i := 0; i < len(ba); i++ {
            if ba[i] != 0 {
                break
            }
            ra[ri] = '1'
            ri--
        }
        return ra[ri+1:]
    }

    func DecodeBase58(ba []byte) []byte {
        if len(ba) == 0 {
            return nil
        }

        x := new(big.Int)
        y := big.NewInt(58)
        z := new(big.Int)
        for _, b := range ba {
            v := strings.IndexRune(base58, rune(b))
            z.SetInt64(int64(v))
```

```
        x.Mul(x, y)
        x.Add(x, z)
    }
    xa := x.Bytes()

    // Restore leading zeros
    i := 0
    for i < len(ba) && ba[i] == '1' {
        i++
    }
    ra := make([]byte, i+len(xa))
    copy(ra[i:], xa)
    return ra
}

func EncodeBase58Check(ba []byte) []byte {
    //add 4-byte hash check to the end
    hash := Hash(ba)
    ba = append(ba, hash[:4]...)
    ba = EncodeBase58(ba)
    return ba
}

func DecodeBase58Check(ba []byte) bool {
    ba = DecodeBase58(ba)
    if len(ba) < 4 || ba == nil {
        return false
    }

    k := len(ba) - 4
    hash := Hash(ba[:k])
    for i := 0; i < 4; i++ {
        if hash[i] != ba[k+i] {
            return false
        }
    }
    return true
}
```

big 包实现的是任意精度的整数和分数运算，包括四则运算、位运算、取余数、幂、求最大公约数和随机数等。在计算超长位密码时，通常会用到这些运算，例如 256 位的 SHA 算法。此处，我们直接把任意长度的字节切片作为一个整数，除以 58 取余数，就方便地得到了这个字节切片的 Base58 编码。

big 包运算通常使用 func (z *Int) Op(x, y *Int) *Int 格式。计算是在 z 上进行的，并且返回 z。所以多个运算可以连续地执行。例如，x.Mul(x,y).Add(x.z)和下面分开写的形式是等价的：

```
x.Mul(x, y)
x.Add(x, z)
```

位钱地址编码使用 EncodeBase58Check 函数，它把一个切片散列两次得到的 4 字节加在后面，再使用 Base58 编码，把它转换为人可以读的、由 58 个字符组成的字符串。而 DecodeBase58Check 则用来检查这 4 字节，确保地址没有传输错误。

作为电子支付手段，位钱是未雨绸缪、宁枉勿纵的。它在散列时不仅使用了很可靠的 SHA256 算法，而且还要散列两次：

```
package bitcoin

import (
    "crypto/sha256"
    "hash"
)

var sha, sha2 hash.Hash

func init() {
    sha = sha256.New()
    sha2 = sha256.New() // hash twice
}
```

```
func Hash(ba []byte) []byte {
    sha.Reset()
    sha2.Reset()
    ba = sha.Sum(ba)
    return sha2.Sum(ba)
}
```

hash.Hash 是一个界面，而具体实现依靠的是 SHA256 算法。这里可以看到 Go 的加密包使用起来是多么简单。无论使用怎样的散列算法，只要一个 New 和一个 Sum 就可以了。Reset 用于将值重新置 0。Size 用于返回 Sum 所需的字节数。而它还内置了另一个界面 io.Writer，可以使用 Writer 提供的方法追加数值。

crypto 包的子目录提供了一些常用的散列算法和加密解密算法，例如 MD5、SHA1、SHA256 等散列算法；AES、DES、Elliptic 等加密算法，以及 RSA、DSA、TLS 等协议。这些都用来实现对 Go 的 http 包所使用的 HTTPS 因特网加密通信协议的支持。我们此处只是使用了最简单的 SHA256 算法。说它简单，不是因为算法简单，也不是因为计算机代码实现简单，而是编程界面 API 简单。对于普通程序员来说，能够正确实施复杂精密的密码操作才是最关键的。Go 在简化 API 方面可以说是不遗余力。只要访问 http://code.google.com/p/go/，看看 crypto 和 hash 这两个包的 API 的演变过程就很清楚了。在密码学里，这通常总结为：链条断在最弱的一环。而写程序的人，总是最不可靠、最易出差错的。

为了确保程序少出差错，最直接的做法是随程序源代码一起编写测试用例。每次修订程序时，就自动测试，保证没有不同结果。Go 的测试包可以使用 go test 工具。它会自动执行包目录中所有以_test.go 结尾的文件里所有以 Test 开头的使用测试签名的函数。例如：

```
package bitcoin

import (
```

```
        "testing"
    )

    type test struct {
        en, de string
    }

    var golden = []test{
        {"", ""},
        {"\x61", "2g"},
        {"\x62\x62\x62", "a3gV"},
        {"\x63\x63\x63", "aPEr"},
        {"\x73\x69\x6d\x70\x6c\x79\x20\x61\x20\x6c\x6f\
x6e\x67\x20\x73\x74\x72\x69\x6e\x67",
"2cFupjhnEsSn59qHXstmK2ffpLv2"},
        {"\x00\xeb\x15\x23\x1d\xfc\xeb\x60\x92\x58\x86\
xb6\x7d\x06\x52\x99\x92\x59\x15\xae\xb1\x72\xc0\x66\x47",
"1NS17iag9jJgTHD 1VXjvLCEnZuQ3rJDE9L"},
        {"\x51\x6b\x6f\xcd\x0f", "ABnLTmg"},
        {"\xbf\x4f\x89\x00\x1e\x67\x02\x74\xdd",
"3SEo3LWLoPntC"},
        {"\x57\x2e\x47\x94", "3EFU7m"},
        {"\xec\xac\x89\xca\xd9\x39\x23\xc0\x23\x21",
"EJDM8drfXA 6uyA"},
        {"\x10\xc8\x51\x1e", "Rt5zm"},
        {"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00",
"1111111111"},
    }

    func TestEncodeBase58(t *testing.T) {
        for _, g := range golden {
            s := string(EncodeBase58([]byte(g.en)))
            if s != g.de {
                t.Errorf("EncodeBase58. Need=%v, Got=%v",
g.de, s)
            }
        }
    }
```

```
func TestDecodeBase58(t *testing.T) {
    for _, g := range golden {
        s := string(DecodeBase58([]byte(g.de)))
        if s != g.en {
           t.Errorf("DecodeBase58. Need=%v, Got=%v",
g.en, s)
        }
    }
}

func TestBase58Check(t *testing.T) {
    ba := []byte("Bitcoin")
    ba = EncodeBase58Check(ba)
    if !DecodeBase58Check(ba) {
        t.Errorf("TestBase58Check. Got=%v", ba)
    }
}
```

对于编写支持所有桌面操作系统的位钱程序，这只是个开始。Go 提供了 RIPEMD160 散列算法，也提供了 ECDSA 公钥算法。而 Go 的网络包 net，可以用来实现点对点联网（peer-to-peer networking）。这些已经可以支持位钱的实现了。

1.9 小结

本章通过几个不大的Go语言程序，基本概括了Go的语法和常见用法。本章的内容，在以后的章节中会深入介绍。但如果通过阅读和练习，读者获得了一些惊喜并且也产生疑问，并有继续阅读的动力，那本章的目的也就达到了。

第 2 章

静态类型

Too much typing and too much typing.

（数据太多类，输入也太累。）

—— Russ Cox

计算机的静态编译类型系统主要完成三项工作。

- 类型错误检查
- 变量内存分配
- 确定适当运算

本章将逐一介绍 Go 语言的几种静态类型：整型、浮点型、复数型、布尔型、数组、字符串、切片、结构体和指针类型。

2.1 数制

人类算术采用十进制可能与人有十根手指有关。而晶体管只有开和关两个状态，所以计算机只能使用二进制。只要是整数，不管使用几进制，也就是无论采用什么算法，数量都不会变。例如分 33 块糖，如果是 10 个 10 个算，那就是 10,20,30 还剩 3 块，所以一共 33 块糖。如果两个两个算，那就是 2,4,6,…,32 还剩 1 块，共 33 块糖，结果都一样，只是表示方法不同。我们从小就学会了以 10 为单位的加减乘除各种运算，所以编程语言必须支持我们继续使用十进制，但它会自动转换为二进制才能让计算机处理。而我们程序员，也必须知道一些二进制知识，才能正确地控制计算机。

2.1.1 整数

Go 语言的整数常量，如果不加特殊前缀，都是十进制表示的，而且可以使用指数形式。例如 100 就是一百，1E9 是 1 乘以 10 的 9 次方。把常量赋值给一个变量，这个变量也就有了相同的值。例如 `var i = 100`，则变量 `i` 的值就是 100。常量的运算结果也是常量，例如 100 - 100 得到常量 0。而变量运算的结果——不要惊讶——会被丢弃。例如 `i - i` 不会有任何结果，只有赋值才可以改变一个变量的值。在 Go 里，赋值使用等号。例如 `k = i - i` 后，变量 `k` 的值为 0，而变量 `i` 的值不变。

整数的加减法使用+和–号，乘法使用*号。除法是/号，得到的商是整数，例如 5/2 = 2。而%号得到除法的余数，5%2 = 1。

运算结果几乎不出所料。例如：

```
package main
```

```
import "fmt"

func main() {
    var i = 10000
    fmt.Println(i*i)
}
```

不出所料结果是10 000 000。但如果 i = 100 000，结果就会变成1 410 065 408。如果 i = 1 000 000，结果成了-727 379 968。为什么？

这是因为Go的变量都有类型。上例的变量 i 是 int 类型，最多只能表示 32 位二进制数，而且正数和负数各占一半，差不多就是正负 21 亿。所以两个一万相乘是1亿，还可以装得下，而两个10万相乘是100亿，就“溢出”了。Go不会报错，但运算结果肯定是错了。

如果换成 var i int64 = 100 000，变量 i 就可以表示64位的整数了，可以装下差不多正负 900 万兆大的整数。就暂时不会出错了。但如果程序员不了解或者不注意计算机数值的表示和限制，出错是早晚的事。

Go的整数最大是64位的int64,还有32位的int32,16位的int16和8位的 int8。而 int 没有规定具体的位数，目前为32位。

Go里最小的整数类型是 int8，表示范围为-128 ~ +127。我们不妨来看看二进制数是如何表示的，顺便搞懂二进制补码是什么意思。

2.1.2 二进制补码

int8 是指8位的整数。位，就像十进制有个位、百位、千位等，二进制的位叫第 0 位，第 1 位等。int8 的最高位是第 7 位。0仍是 0，1仍是1，但2要占两位，写为10。所以有一个笑话：

“这个世界只有10种人，懂二进制的和不懂的。”

如此类推3写为11，4是100，直到127是1111111，注意共7位，

而第7位是0。

8位二进制数可以表示256个数。如果也要表示负数，就需要拿出一半的位数给负数。按照规定，int8 的第7位如果是0，代表正数；如果是1，代表负数。但0占了一个正数，这样正数只能从1到127，而负数可以从-1到-128。

那127加1等于几？int8 无法正确表示，因为 int8 这个类型不能表示正数128。01111111 + 1 = 10000000，成了一个负数。我们规定它是-128，这样，-128 + 1 = -127 = 10000001，而 -128+127 = -1 = 11111111，从而使得 -1 + 1 = 11111111 + 1 = 100000000 = 0。第8位被丢弃，所以后面的8位刚好是0。这就是二进制补码。称为“补”，是因为负数刚好是对应正数的反码再补1，而反码是指反转每一位的01。例如，00000001 的反码是11111110，补1成为11111111，刚好就是-1。

为什么要采用补码的格式？我们先看另一种类型的8位整数，uint8，它可以表示0～255的整数。此时，0还是0，127还是01111111，而128是10000000，255是11111111，也就是从0～255是连续表示的。而补码直接从 int8 所能表示的最大正数127加1跳到最小负数-128。

两种类型整数的加法是一样的。-1 + 1 就是 255 + 1 = 0。加减乘法都是这样，对计算机来讲，就是不需额外的电路处理负数。所以几乎所有编程语言都使用二进制补码表示整数。

2.1.3 整数比较

但比较大小就比较麻烦。127是01111111，可10000000如果是-128，当然比127小很多，但如果是128，就大于127。因为8位只能代表256个数，不可能顺序比较-128～255这384个数。所以，Go程序员必须给出明确的转换。例如：

```
package main

import "fmt"

func main() {
    var i int8 = -128
    fmt.Println(uint8(i) > 127)
}
```

大小比较使用>和<，相等和不相等比较用 == 和 !=。大于或等于是>=，小于或等于是<=。比较的结果是布尔类型的值，真为 true，假为 false。

布尔类型的值可以做逻辑运算。只有三种运算，结果也是布尔类型的值：真 true 和假 false：

p && q 表示如 p、q 同时为真则结果为真，否则为假。
p || q 表示如 p、q 同时为假则结果为假，否则为真。
!是真假取反。

例如 (!false == true) && (true && false == false) && (true || false == true) 的结果是 true。注意 q 有可能被短路而不执行。例如下面判断奇偶数的操作，不会出现除以零的错误：

```
i != 0 && 1 /  i == 1
```

布尔值不是特殊的整数值，true 和 false 不可以和整数值做运算和转换。它们作为比较和逻辑操作的结果，通常用在 if 和 for 语句中。

布尔类型的逻辑运算，也称为逻辑“与、或、非”运算，对应着 &&、|| 和 !。它们和二进制整数的按位“与”和按位“或”运算是完全不同的概念。

2.1.4 按位运算

按位运算是指计算机对一个整数的每一位同时分别进行运算。

与&是指只有两位都是 1 时，对应的结果位才是 1，否则是 0。例如

5 & 3 == 1，因为其二进制是 101 & 011 == 001。

或 | 是指只有两位都是 0 时，对应的结果位才是 0，否则是 1。所以 4 | 2 == 6，因为其二进制是 100 | 010 == 110。

异或^是指只有两位相同时，对应的结果位才是 0，否则是 1。所以 4 ^ 2 == 6，因为其二进制是 100 ^ 010 == 110。

异或自身，相当于取反码，也就是逐位取反，Go 会自动用一个全部位都是 1 的数和它异或 例如 ^1 == -2，因为如果是 8 位二进制的情况，相当于是 000000001^11111111 = 11111110。

与非&^用于清位，i&^j 就是把 i 的最低 j 位清零，就是 i&(^j)。例如 7 &^ 3 ==4，因为其二进制是 111 &^ 011 == 100。

除 2 取余数可以用&操作。例如 5 % 2 == 5 & 1，因为我们只看第 0 位。而除以 2 也可以用右移操作>>来实现。例如 5 / 2 == 5 >> 1，表示右移一位。而左移操作<<则等同于乘以 2。

总结如表 2-1 所示。

表2-1 整数算术操作符表

+	加法
-	减法
*	乘法
/	除法取整数商
%	除法取余数
&	按位与
|	按位或
^	按位异或
&^	按位与非
<<	按位左移
>>	按位右移

2.1.5　浮点数

+、-、*、/还可以用在小数和复数上。Go 的小数称为浮点数，有 32 位和 64 位两种，分别是 float32 和 float64 类型，符合 IEEE 754 的规定。它们也是大多数计算机直接支持的浮点数类型。在 Go 里表示浮点数常量要加小数点。例如：

```
var i = 0.0
```

变量 i 的类型是浮点型，而不是整型；值不是整数 0，而是浮点数 0.0。刚好它们在计算机内部的表示是一样的，都是全部位为 0。而整数 1 和浮点数 1.0，则是完全不一样的。我们可以用下面的程序演示：

```
package main

import (
    "fmt"
    "math"
    "strconv"
)

func main() {
    fmt.Println(strconv.FormatUint(1, 2))
    fmt.Println(strconv.FormatUint(math.Float64bits
(1.0), 2))
}
```

同样的常量，例如 1 和 1e4，可以根据使用的场合，作为整数或者浮点数，Go 会自动转换。例如

```
fmt.Println(strconv.FormatUint(1e4, 2))
fmt.Println(strconv.FormatUint(math.float64bits
(1e4), 2))
```

FormatUint 需要一个整数，所以 1e4 是 10000，而 float64bits 需要一个浮点数，1e4 就成了 1.0×10^4。e 或者 E 表示 10 的指数部分。

两个浮点数比较大小可能会吓到你。例如 0.1 + 0.2 != 0.3。这是因为计算机内部不能精确地表示浮点数，一定会做舍入，所以这两个数舍入后再加减，就不会得到预期的浮点数的结果了。浮点数的比较，千万不要用等不等于，要用大于等于，或者小于等于。例如上例要写为：0.1 + 0.2 >= 0.3。

2.1.6 复数

Go 还支持复数类型。complex64 使用两个 float32 类型的数分别作为实部和虚部。complex128 则使用两个 float64 类型的数分别作为实部和虚部。它们可以用 $0 + 1i$ 这样的格式表示，还可以做四则运算。例如：

```
package main

import "fmt"

func main() {
    var i = 0 + 1i
    fmt.Println(i*i)
}
```

我们还可以用 complex、real 和 imag 函数来组装和分装复数类型。例如：

```
var a = complex(3.14, -1)
var b = imag(a)
var c = real(a)
```

不同的数值类型可以相互转换。只有相同的类型才可以做运算，Go 只自动转换常量，对变量的转换必须明确指出，使用的格式类似函数调用，即相当于用类型名作为函数。例如：

PDG

```
package main
import "fmt"
func main() {
    var i int32
    var j int
    fmt.Println(i + int32(j))
}
```

尽管 i 和 j 的内部表示都是 32 位的补码，它们仍是不同类型，直接 i + j 会报错。

2.1.7　优先级

当一起使用多个操作符时，它们的先后次序应该符合我们从小学会的四则运算规则，例如从左到右、先乘除后加减、小括号优先等。但 Go 有 19 个二元操作符，要复杂一些，需要 5 层，从高至低的具体规定如表 2-2 所示。

表2-2　优先级表

级　别	操 作 符
5	* / % << >> & &^
4	+ - | ^
3	== != < <= > >=
2	&&
1	||

一元操作要优先于二元操作。Go 有 7 个一元操作符，如表 2-3 所示。

表2-3　一元操作符表

操 作 符	代　表
+	忽略，+(-1) == -1
-	取负数，-(-1) == 1

（续）

操 作 符	代　　表
!	逻辑非，!true == false
^	逐位取反，^0 == 1
*	取指针变量的值，*p是变量p指向的值
&	取变量的指针，&p是变量p的指针
<-	通信操作。发送或者接收

2.1.8 表达式

这些数值的运算，在编程语言中称为表达式。表达式运算的结果是数值。这样，表达式就是计算的基本单位，可以单独作为一条语句，可以作为其他语句的一部分，与控制语句一起，构成完整的程序。

2.2 字符和字符串

顾名思义，计算机就是计算机器，数值是它们唯一能处理的东西。字符，完全是出于和人交流的需要才引入的。因为计算机的输入和输出，很多时候还是需要人的。

字符是人类语言，在计算机内部也要表示为一个数值才可以进行处理。这就必须规定从字符到数值的编码规则。

2.2.1 UTF-8 编码

ASCII 曾是计算机通用的编码，但它只有 7 位，只能表示 128 个字符。这对使用英文编写程序的程序员是足够了。所以早期的编程语言只支持 ASCII 码。

后来全世界的人都需要计算机了，能处理本地的语言成了必需。但 ASCII 只是美国标准，不是每国都买账。这就有了各种各样的扩展，比如我们国家标准的 GB2312 编码，就是 14 位的、有 6763 个汉字的 ASCII 扩展。它的扩展方式，是看一个 8 位字节的最高位。如果是 0，则刚好是一个 ASCII 字符。如果是 1，则后面的 7 位，以及下一个字节的后 7 位，一起表示一个 14 位的 GB2312 编码的字符。

但这只是中国自己的规定。日本也有自己的规定。各国的不一致就造成了交流的混乱。在全世界计算机和软件生产商的要求下，在国际标准化组织的努力下，字符编码终于统一为 Unicode 字符方案，几乎全球现存的历史文献曾有的文字，都有了唯一的码值。通常是一个 16 位的整数，范围是 0 ~ 65 535。

但难题只解决了一半。因为计算机以及还在使用的一些编程语言，仍使用 8 位的字节处理 7 位的 ASCII，短期内不可能改变。而 16 位的

Unicode，仍需要对应到两个字节。哪个字节在前大家又有了分歧，产生了两种 UTF-16 编码的 Unicode 字符。新的编程语言和计算机系统，试图直接使用 16 位甚至 32 位的 Unicode 编码，作为字符的基本处理单位，但这也面临着兼容现有程序和文件的巨大挑战。

UTF-8 是一种 ASCII 扩展。它把 32 位的 Unicode 字符，编码到一个到多个字节。Unicode 的前 128 个字符，刚好是 ASCII。这样，能处理 ASCII 的 C 语言和其他语言的程序不需修改，也可以处理 UTF-8 编码的 Unicode 字符。这也许是发明 UTF-8 的原因。UTF-8 是 Go 语言的两位作者 Rob Pike 和 Ken Thompson，在十几年前一次聊天时发现的。然后他们把工作用的 Plan9 操作系统的所有程序，在极短的时间内修改成全部支持 UTF-8。现在，UTF-8 广泛用在因特网上，成为 Unicode 最通用的编码方案。

2.2.2 Unicode字符

Go 使用 UTF-8 编码的 Unicode 字符。也就是说，Go 的字符，是 32 位的整数类型，称为 `rune` 类型。`rune` 是一种古日耳曼的文字。在 Go 里，`rune` 类型的每一个数值，都对应着 Unicode 字符表中到一个字符的编号，称为码值。我们可以用 U+四个数字来表示一个 16 位的整数，注意这 4 个数是十六进制的数。例如 U+0061 对应着 a，它是 Unicode 字符表中的第 97 个字符，而 U+00E4 是 ä，等等。

这样把字符写为十六进制数太不直观了。Go 有更好的表示方法。我们可以直接把字符用两个单引号括起来。例如上面的两个字符，可以写为`'a'`和`'ä'`。

在单引号表示字符的格式里，如果要用十六进制数表示，可以用\x 加两个数、\u 加 4 个数或者\U 加 8 个数的格式。例如'\x61'或者'\u0061'或'\U00000061'都是字符'a'。

2.2.3 转义字符

这种使用反斜线的格式称为“转义”。Go 还规定了其他一些转义字符，用来代表那些常用又不能直接用键盘输入的字符，如表 2-4 所示。

表2-4 转义字符表

字 符	表 示
\a	铃音
\b	退格
\f	进表
\n	换行
\r	回车
\t	横向制表
\v	纵向制表
\\	反斜线
\'	单引号

还有一个是\"，代表双引号，它不能出现在单引号括起来的字符中，但可以出现在双引号括起来的字符串中。

Unicode的字符几乎能出现在Go语言的所有地方。只有一个Unicode字符例外，就是它的第0个字符，它一般作为C字符串结束标记的NUL字符，不能出现在Go的源代码中。

2.2.4 字符串

字符串简单来讲就是一串字符。复杂一点讲，就是类似使用 UTF-8 编码的字符的字节数组，但内容不能改变。

在 Go 源代码中，可以使用两种方式表示字符串。一种是双引号""括起的，可以包含转义字符的一串字符。另一种是使用反引号``括起的，

不可以包含转义字符的一串字符。例如"\\n\n\\n"和下面的写法是相同的：

```
`\n
\n`
```

而下例的每行都代表同样的字符串：

```
"日语"
`日语`
"\u65e5\u672c\u8a9e"
"\U000065e5\U0000672c\U00008a9e"
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e"
```

双引号括起的字符串中不可使用\'这个转义字符，直接使用单引号即可，而双引号自身，需要使用\"来表示。

反引号括起的字符串中没有转义字符，\"就是反斜线和双引号这两个字符。

字符串非常有用。例如，Go的报错信息都可以输出为一个字符串。如果我们要打开一个不存在的文件：

```
_, err := os.Open("noSuchFile")
fmt.Println(err)
```

输出：

```
open noSuchFile: The system cannot find the file
specified.
```

2.2.5 字符串转换

因为 string 是字符串，而字符是整数，我们当然可以把一个字符或者整数转换为一个字符串。而不可转换的整数，则是"\uFFFD"。例如：

```
string('a')      // "a"
string(-1)       // "\ufffd" == "\xef\xbf\xbd "
```

```
string(0xf8)     // "\u00f8" == "ø" == "\xc3\xb8"
string(0x65e5)   // "\u65e5" == "日" == "\xe6\x97\xa5"
```

而要得到一个字符串的某个字符，则需要明白 Go 的字符串类似一个 UTF-8 编码的字节数组。也就是说，我们可以像操作数组的下标一样，用[i]得到第 i 个字节，而不是第 i 个字符。i 从 0 开始。例如：

```
package main
import "fmt"
func main() {
    var s = "Go 程序"
    fmt.Printf("%c %c", s[0], s[2])
}
```

得到的是 G c，而不是我们以为的 G 程。这是因为按 UTF-8 的规定，英文字母、数字和标点，一个字符是 1 字节，而一个汉字是 3 字节。s[2] 拿到的只是这 3 个字节的第一个，对应的是 U+00E7 这个字符。注意，字符是 Unicode 码值，不是 UTF-8 编码。如果一下子不能理解，我们输出这个字符串每个字节的十六进制数来看一看。fmt.Printf("% x", s) 得到：

```
47 6f e7 a8 8b
```

如果可以保证字符串的每个字符不超过 1 字节，我们可以直接操作下标。但通常会用更安全、更通用的方法。例如，把字符串转换为 rune 切片，再做下标操作：

```
package main

import "fmt"

func main() {
    var s = "Go 程"
    var r = []rune(s)
    fmt.Printf("%c %c", r[0], r[2])
    fmt.Printf("% x", r)
}
```

输出:

```
G 程[ 47  6f  7a0b]
```

这里的`[]rune`是`rune`切片。`r`的值是3个整数`47`、`6f`、`7a0b`,代表着`s`中每个字符的Unicode码值。

同样,我们可以用`s(r)`把一个`rune`切片转换为一个字符串。字节数组的转换也是一样。

如果理解了字符串类似字节数组,可以用下标取值,下面的数组和切片就很好理解了。它们的主要不同是,字符串不可以使用下标赋值,并且数组的单元可以是任意类型,而不仅仅是字节。

2.3 数组

前面讲的布尔类型、数值类型（包括字符）和字符串类型，以及后面要讲到的各种类型，都可以作为一个单元，重复而且连续地排列在一起，称为一个数组类型。也就说，数组不是特指一组数值，而可以用各种类型（包括数组类型）表示一个单元，这个单元的类型称为此数组类型的基础类型。而每个单元，都可以用下标操作，作为一个单独的变量来读写。

例如，一个点 Point 的二维坐标，可以用两个浮点数来表示。而一条线 Line，可以用两个点表示。例如：

```
package main

import "fmt"

func main() {
    type Point [2]float32
    type Line [2]Point

    a := Point{1,2}
    b := Point{3,4}
    l := Line{a, b}

    fmt.Printf("%g, %g, %v, %v", a[0], l[1][1], b, l)
}
```

输出：

```
1, 4, [3 4], [[1 2] [3 4]]
```

2.3.1 声明

[2]float32 声明一个包括两个 float32 类型的数组类型。我们可

以用关键字 type 声明一个新的类型。这样，Point 是一个包括两个 float32 类型的数组类型，而 Line 就是包括两个 Point 类型的数组类型，它的值间接地包括了 4 个浮点数。

变量声明可以用 var，也可以用:=。数组类型变量的初始值可以放在大括号里，变量声明时逐一赋值给它的每个单元。所以 a := Point{1,2}在声明变量 a 是 Point 类型的同时，给第一个 float32 单元赋值 1.0，给第二个单元赋值 2.0。

2.3.2 下标

严格来讲，Go 的数组、切片和字符串的单元，是从 0 开始取下标的，所以第一个单元，指的是[0]。为了不发生误解，本书以后所说的第 *n* 个单元，都是从 1 开始数的，而下标一定会放在方括号里，称为[n]单元，是从 0 开始的。

数组的长度，也就是其单元的个数，是固定的，而且必须在声明时给定。它是个常量。函数 len 可以得到一个数组的长度。例如 len(l) == 2。

如果我们的下标小于 0 或者大于等于数组长度，编译时会报错：index out of bounds。可如果下标也是个变量，编译时无法得知它的值是否越界，但运行时会检查数组的下标操作，如果下标小于 0 或者大于等于数组的长度，运行会出错。错误信息是：panic: runtime error: index out of range

2.3.3 赋值

Go 的数组有一个重要的特性，它作为一个整体，在赋值时逐位复制到新的变量。这和数的赋值是一样的，而和字符串的赋值不同。例如：

```
package main
```

PDG

```
import "fmt"

func main() {
    type Point [2]float32
    type Line [2]Point

    a := Point{1,2}
    b := a
    c := Line{a, b}
    b[0] = 42
    s := "%v, %v, %v"
    fmt.Printf(s, a, b, c)
}
```

输出：

```
[1 2], [42 2], [[1 2] [1 2]]
```

执行 b := a 后 b 的两个单元值也是 1 和 2，但其后 b 的第一个单元值改为 42，并不影响 a，也不会改变 c。因为 a、b 和 c 占用地址完全不同的内存。

同样的赋值也发生在调用函数 Printf 时。也就是给 Printf 使用的是 a、b 和 c 的副本，占用新的内存，在 Printf 函数内改变复制后数组单元的值，也不会对 a、b 和 c 产生任何影响。所以把数组交给函数处理是安全的，没有副作用。但代价是，复制要占用新内存，也要花时间。对于有很多单元的数组，必须要平衡安全性和效率。通常，我们会使用切片来代替数组。

字符串因为不可以改变其内容，交给函数使用时不会有副作用，因此不需要复制其内容，仅仅复制一个“头”就可以了。这个“头”包括字符串所在内存的地址还有长度。所以上例 Printf 用到的 s 的副本，和 s 一样指向同样地址和长度的内存。

切片也有类似的“头”的概念。

2.4 切片

切片（slice）的得名，大概是联想到了切好片的面包。很少有人可以一次吞下整个面包，我们每次只是拿出其中连续的几片，这在 Go 里表示为[i:j]。例如：

```
package main

import "fmt"

func main() {
    s := [4]int{0,1,2,3}
    t := s[1:3]
    fmt.Println(s[0],t,s[:3],t[1:])
    fmt.Println(len(s),cap(s))
    fmt.Println(len(t),cap(t))
}
```

输出：

```
0 [1 2] [0 1 2] [2]
4 4
2 3
```

s 声明一个单元类型为 int 切片类型，同时赋值。注意，虽然它很像数组的声明，但没有指定长度。每个切片变量只是一个“头”，包括三个部分：指向一个底层数组的指针、长度、还有容量。这里，s 的底层数组是 4 个 int 值表示的一片内存，分别存放着 0、1、2、3。s 的长度为 4，容量也为 4，可以分别用函数 len 和 cap 得到。

切片的下标操作和数组的下标操作一样。s[0]得到第一个单元。而 s[i:j]得到一个新的切片变量，它也指向 s 的底层数组，但是从第[i]个单元开始，长度是 j-i，而容量是 cap(s)-i。如果 i 是 0，可以不写，

如果 j 是 len(s)，也可以不写。

因为切片是数组的头，赋值时仅仅是指针、长度、容量这三个值，并且通过指针，赋值后的切片也可以直接改变底层数组的单元，所以，函数的参数多用切片而不是数组。

make、copy、append函数

切片可以通过声明或赋值得到，还可以通过内置函数 make 创建。例如 make([]string, 0, 10)得到一个长度为 0、容量是 10 的字符串切片。而 make([]byte, 10)创建的字节切片的长度和容量都是 10。

copy(t,o)把切片 o 的单元逐个复制到切片 t。注意不要搞错从谁到谁。可以记为 t 是 to 目的，o 是 origin 起点。Go 的函数参数几乎都遵守这样的次序。

t 和 o 必须是相同单元类型的切片。只有一个例外，o 可以是字符串，复制到 t 的字节切片。len(t)和 len(o)的最小值决定复制的单元数量。t 和 o 可以部分重叠。例如：

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:]) // s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:]) // s == []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "Hi")  // b == []byte("Hi")
```

append(t,o)可以把 o 添加在切片 t 的最后，并返回新的切片。o 可以是 t 的单元类型的变量或值，可以是一个或多个值，也可以是 t 类型的切片，写为 o...这样带省略符号的变量。例如：

```
s0 := []int{0, 0}
s1 := append(s0, 2) // s1 == []int{0, 0, 2}
s2 := append(s1, 3, 5)  // s2 == []int{0, 0, 2, 3, 5}
```

```
s3 := append(s2, s0...) // s3 == []int{0, 0, 2, 3, 5,
0, 0}
```

带省略符号的变量还可以用作字符串，添加到字节切片上。

例如：

```
var b []byte
b = append(b, "hi"...)  //b == []byte{'h', 'i'}
```

注意，append 可能重新分配底层数组来容纳添加的单元，所以我们总是把 append 的返回值重新赋值给切片变量。例如：

```
t = append(t, o, o)
```

使用 append，切片就没有容量的限制，可以灵活地执行添加、插入和删除操作。记住表 2-5 的各种操作，或者记得把本书放在手边。这些切片 append 操作，可以灵活地用在很多场合。希望你能在遇到这些场合时，第一时间想到切片的 append 操作。

表2-5 切片操作

添加切片 b

```
a = append(a, b...)
```

复制

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

删除 [i:j]

```
a = append(a[:i], a[j:]...)
```

删除第i个元素

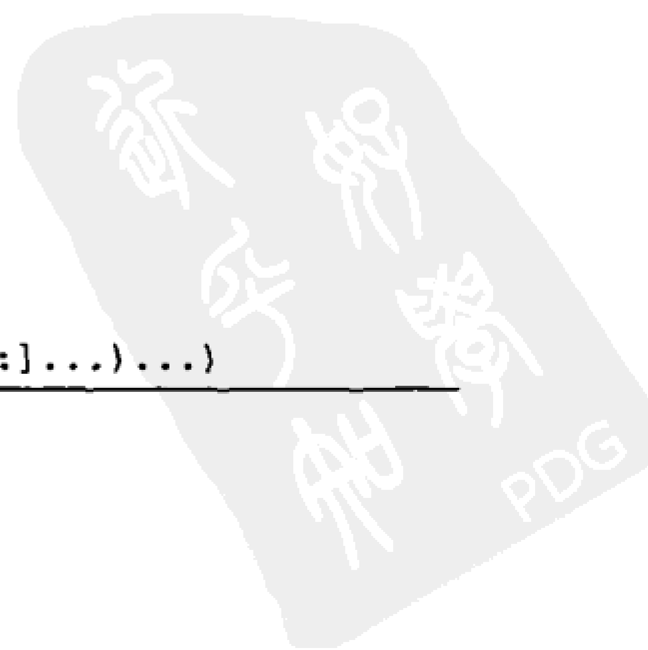
```
a = append(a[:i], a[i+1:]...)
```

扩展j个空元素

```
a = append(a, make([]T, j)...)
```

插入j个空元素

```
a = append(a[:i], append(make([]T, j), a[i:]...)...)
```



（续）

插入元素x `a = append(a[:i], append([]T{x}, a[i:]...)...)`
插入切片b `a = append(a[:i], append(b, a[i:]...)...)`
弹出最后一个元素 `x, a = a[len(a)-1], a[:len(a)-1]`
压入x `a = append(a, x)`

2.5 结构体

如果说数组是相同类型单元的连续排列，并使用下标读写。那结构体可以说是不同类型单元的连续排列，使用项名读写。例如平面上一个点的坐标，可以声明为 [2]float64 数组，也可以声明为 struct{x,y float64}结构体，还可以声明为 struct{r,θ float64}，代表半径和角度的极坐标。它们所占的内存是一样的，都是两个 64 位的浮点数。但读写这两个浮点数的方式是不同的。数组或切片类型的变量使用 a[0] 读取第一个单元。而结构体类型的变量用项名 x、r 等，例如：s.x、s.r。

使用项名而不是下标数字，使得单元的意义更明确。所以即便是坐标这样简单的组合类型，我们也推荐使用 struct。而数组和切片，只有在需要自动计算下标来读写其单元时，例如需要使用 for 循环时，才需要用到。

我们通常使用 type 给结构体类型一个名字，例如 net/url 包的 URL 类型是一个结构体类型：

```
type URL struct {
    Raw           string
    Scheme        string
    RawAuthority  string
    RawUserinfo   string
    Host          string
    RawPath       string
    Path          string
    OpaquePath    bool
    RawQuery      string
    Fragment      string
}
```

我们可以这样读取其 Host 的值：

```
package main

import (
    "fmt"
    "net/url"
)

func main() {
    gurl, er := url.Parse("http://golang.org/pkg")
    fmt.Println(gurl.Host, er)
    //Output:
    //golang.org <nil>

}
```

2.5.1 项

如果一个结构体有多个项（field）时，项名必须不同，唯一的例外是使用下划线_，来做补丁。例如：

```
struct {
    byte0   byte    // 第 1 字节
    _       byte    // 填充
    byte2   byte    // 第 3 字节
}
```

似乎可以通过补丁精确地分配每个项内存的相对位置。但实际上是做不到的，所以补丁在结构体类型里似乎没什么用途。

Go 语言的一个设计原则是避免对内存的直接操作。因为一个结构体的项可以是从 8 位到 128 位不同字宽的数值类型，但计算机可能要求每个项都对齐到 32 位的内存地址。这样，尽管项在内存中是顺序排列的，

但一个 8 位的字节到下一个项之间，可能会有 3 字节的空档。这是程序员无法用 Go 控制的。如果需要，可以调用 C 或其他语言写的函数来完成。

2.5.2 内置

下划线补丁做项名没什么用，但直接使用另一结构体类型做项名就非常有用。例如：

```
package main
import "fmt"
type D2 struct {x, y float64}
type D3 struct {
    D2
    z float64
}
func main() {
    var d2 = D2{1,2}
    d3 := D3{d2, 3}
    fmt.Println(d2.x, d3.D2, d3.x, d3.z)
    fmt.Println(d3.D2 == d2)
}
```

二维点类型 D2 只比三维点类型 D3 少了一个 z，所以很自然地想到可以在 D3 类型里内置 D2 的类型。此时，D2 所占的项名就叫 D2，而 D2 声明的 x 和 y 也可以直接在 D3 的变量中使用。

声明结构体类型的变量时，可以直接在类型名后的大括号里逐个项的赋值。例如 d3 也可以写成 d3 := D3{D2{0,1}, 3}。即便是下划线补丁也需要一个值。

结构体类型的值不可以比较大小，但可以比较是否相等。比较时逐个项进行比较。如果全部项的值都相等，这两个结构体类型的值才相等，否则不相等。

注意，只有两个结构体类型是相同类型时才可以比较，相同结构体类型是说它们项的顺序、名称、类型和标签都相同。

结构体类型在使用时，我们把它的变量作为一个整体，赋值或者作为函数的参数传递。和切片类型不同，但和数组类型一样，结构体类型的变量直接赋值和作为函数的参数传递时，往往涉及大量内存复制。数组还可以通过切片来间接传递，但结构的间接传递，就要使用指针了。

2.6 指针

虽然 Go 语言的设计原则是不直接操作内存，但还是提供了指针类型，可以直接得到其对应底层类型的内存变量地址，并能间接地读写那个类型的变量。这就赋予了程序员一种能力，可以让不同的变量共享同一片内存。但也要求程序员自律，尤其是在使用去程并发编程时，程序员要管理好共享的内存，避免冲突。因为切片也含有一个指向底层数组的指针，这种自律对切片也是同样重要。不使用指针的类型是安全的，因为它们的变量在赋值时都会复制，不存在内存共享的问题。

声明指针类型很简单，只要在某个类型前加乘号或称为星即可。例如*int 是 int 的指针类型；**int 是 int 指针的指针类型等。取变量地址使用&操作符；取指针变量对应的变量的值使用*操作符。例如：

```
package main

import "fmt"

func main() {
    var i int
    var p *int
    var pp **int
    i = 0
    p = &i
    pp = &p
    *p++
    fmt.Println(i, p, *p, pp, *pp)
}
```

输出：

```
1 0xf8400001c0 1 0xf8400001f8 0xf8400001c0
```

你看到的是它们的值。i 的值为 0，p 和 pp 的值是 64 位的内存地址，分别是变量 i 的地址和变量 p 的地址。

Go 虽然有指针类型，但我们在上面很小心地说明它只有两个操作：只能取得一个现有变量的地址，也只能取得那个变量地址的值。这不像其他 C 类语言，Go 语言的指针类型变量不可以通过地址的加减法去读写任意的内存地址。那样做非常非常不安全，也完全没有必要。记得有人比喻说，那就像给黑猩猩一把上了膛的枪。而其他 C 类语言，不仅没有拿走这把枪，还试图教猩猩怎么用。等铸成了大错，才说软件出现危机啦，都是猩猩惹的祸。其实，**没有过错，错的是++。

Go 语言也有++和--操作，但只能用于整数类型。它是单独的语句，不是可以混用的表达式。例如*c++是(*c)++，而不是 C 类语言的*(c++)。

指针是把双刃剑。用好了是柳叶刀，可以救死扶伤，解决复杂问题时游刃有余。用不好，就是猩猩的枪，吓跑了大家，还可能伤到自己。

2.7 小结

计算机科学的基础知识是数据结构和算法。本章只是笼统地介绍了能够被 Go 编译器静态检查的一些基本数据类型和复合类型。Go 的类型系统还有很具体的技术细节，可以参考《Go 语言规范》。而数据的算法，也就是程序流程的控制操作，是下一章需要学习的内容。

第 3 章

流程控制

IBM stole the term "Structured Programming"
and trivialized it to the abolishment of the goto statement.
（IBM 窃取了结构式编程的概念，却贬低它只是弃用了 goto 语句。）
—— Edsger Dijkstra

最初的编程语言只有判断和跳转，也就是 if 和 goto 语句。这是因为最基本的 CPU 只需三种指令流程：顺序、判断和跳转。但 20 世纪 60 年代开始提倡的结构式编程，引入了“块”的概念，以及循环和函数。这使得之前跳来跳去的乱麻程序变得像积木一样，大大提高了程序的清晰度和可靠性。这不仅仅是舍弃了一条 goto。本章我们叙述结构式编程的基础——流程控制语句，也包括被人误解的 goto。

3.1 简单语句

之所以说“简单”，因为这些语句只占一行。如果把它们用大括号括在一起，可以构成一个“块”，成为控制语句的一部分。如果将它们放在布尔表达式前，可以一起用在 if 和 switch 语句中。它们也可以用在 for 语句中，作为初始赋值和增量语句。

最简单的空语句，什么都不做，只是以分号结束的一个空行。

赋值语句是由等号连接的左右两个表达式，例如 x = y。左侧表达式必须可以取地址、映射的下标表达式，或是_空标识符 。例如：

```
x = 1
*p = f()
a[i] = 23
```

算数运算符可以和等号一起用，同时完成运算和赋值。例如：x += y 就是 x = x + y，但表达式 x 只求值一次。例如：

```
a[i] <<= 2
i &^= 1<<n
```

多项赋值把多个值依次赋值给每个变量。例如 x,y = y,x 可以交换 x 和 y 的值。变量是从左至右赋值的，但注意，赋值之前等号两侧的表达式都已经求值了。例如：

```
x, x = 1, 2
```

x 会先赋值 1 ，再赋值 2，所以 x 等于 2。而

```
i, x[i] = 1, 2
```

如果 i=0，则求值得到 x[0]，再赋值 i 为 1，才赋值 x[0]为 2。

自增语句是 ++，自减语句是--。它们用在一个表达式的后面，可以分别对表达式的值加 1 和减 1。例如 x++ 和 x--，分别对应的就是 x +=

1 和 x -= 1 语句。

而表达式语句包括函数调用、方法调用、接收操作，可以单独作为一个语句。例如：

```
h(x+y)
f.Close()
<-ch
```

3.2 判断语句 if

计算机不只会计算，它还会判断。它使用 if 语句把一个计算的结果作为条件，从两个可能中选择一个继续。例如求绝对值：

```
if a < 0 {
    a = -a
}
```

if 后的条件必须是个布尔类型的值。如果是 true，则执行大括号里的语句。如果是 false，则不执行。但如果紧接着的是个 else 语句，则在 if 条件是 false 的时候，执行 else 后大括号中的语句。例如判断是否是闰年：

```
if year % 4 == 0 {
    if year % 100 == 0 {
        if year % 400 == 0 {
            leap = true
        } else {
            leap = false
        }
    } else {
        leap = true
    }
} else {
    leap = false
}
```

这样写程序的目的似乎是为了把一个简单的逻辑变复杂。Go 推荐的写法是把一个看似复杂的逻辑写得很简单。闰年，就是每 4 年一次，但不包括每 100 年却又包括每 400 年。这样就有：

```
if leap = false; year%4==0 && (year%100!=0 || year%400
==0) {
```

```
    leap = true
}
```

我们还看到，if 之后、条件之前，可以有一个简单语句。

if 和 else 之后的语句一定要用大括号括起作为一个块。只有一个例外，就是 else if 连在一起用。例如计算 7 级税率：

```
income := 12345
base := 3500
excess := income - base
tax := 0
if excess < 1500 {
    tax = excess*3/100
} else if excess < 4500 {
    tax = excess*10/100 - 105
} else if excess < 9000 {
    tax = excess*20/100 - 555
} else if excess < 35000 {
    tax = excess*25/100 - 1005
} else if excess < 55000 {
    tax = excess*30/100 - 2755
} else if excess < 80000 {
    tax = excess*35/100 - 5505
} else {
    tax = excess*45/100 - 13505
}
```

使用 switch 来代替多个 else if 看起来会更简单些。

3.3 多分支语句 switch

上面的 7 级税率计算程序也可以用 switch 语句：

```
switch {
case excess < 1500:
    tax = excess*3/100
case excess < 4500:
    tax = excess*10/100 - 105
case excess < 9000:
    tax = excess*20/100 - 555
case excess < 35000:
    tax = excess*25/100 - 1005
case excess < 55000:
    tax = excess*30/100 - 2755
case excess < 80000:
    tax = excess*35/100 - 5505
default:
    tax = excess*45/100 - 13505
}
```

对比 switch 和 else if，可以看到每个 switch 会从上至下逐个 case 地判断条件，如果是 true，则执行冒号后到下一个 case 之前的语句，称为一个分支。如果所有 case 分支都是 false，而且有一个 default 分支，则执行该分支。

switch 后也可以跟一个表达式，而 case 和 default 则是此表达式的可能值。例如：

```
switch day {
case 1,3,5: fmt.Println("Odd day")
case 2,4: fmt.Println("Even day")
case 0,6: fmt.Println("Weekend")
default: fmt.Println("Not a day")
}
```

类似 if，switch 在表达式之前也可以有一个简单语句。例如：

```
switch x := f(); {
case x < 0:
    x =-x
    fallthrough
default:
    g(x)
}
```

fallthrough 使程序继续执行下一分支的语句，可以用来合并公共的语句部分。

switch 表达式也可以求得 true 或者 false 的值，此时就和 if 没有区别了。更多的时候，与 if 相比，switch 可以明确地表示不同状态之间的转换。例如三种生命状态接收到消息后的相互转换：

```
package main

const (
    LIMBO = iota
    LIVE
    DEAD

    KICK
    KILL
)

func recv() int {
    return KILL
}

func main() {
    state := LIMBO
    for {
        switch note := recv(); state {
        case LIMBO:
            switch note {
```

```
            case KICK:
                state = LIVE
            case KILL:
                state = DEAD
            }
        case LIVE:
            switch note {
            case KICK:
                state = LIMBO
            case KILL:
                state = DEAD
            }
        case DEAD:
            switch note {
            case KICK:
                state = LIMBO
            case KILL:
                panic("double kill")
            }
        }
    }

}
```

注意使用 for，才能在不同状态一直循环下去。

3.4 循环语句 for

循环也可以说是一种特殊的判断：它在大括号结束的地方又回到 for 再次判断，直到条件为 false，才跳过这块。

如果没有写判断条件，条件就是 true，就会一直循环下去。for{} 也就是 for true {}。而 for ok {} 在 bool 类型的变量 ok 的值是 true 时循环，是 false 时不再循环。例如：

```
package main

import "fmt"

func main() {
    var i uint8
    for i < 8 {
        fmt.Printf("%08b\n", 1<<i)
        i++
    }
    //Output:
    //00000001
    //00000010
    //00000100
    //00001000
    //00010000
    //00100000
    //01000000
    //10000000
}
```

这个程序刚好可以让我们直观地看到<<左移操作。

其实 Go 里对这种 for 循环的推荐写法是：

```
package main

import "fmt"
```

```
func main() {
    for i := 0; i < 8; i++ {
        fmt.Printf("%08b\n", 1<<uint(i))
    }
}
```

此处的变量 i，一般称为循环变量，从循环开始前设置初始值，到每次循环前的判断，再到再次循环判断前的增值操作，全部放在一起，用分号隔开，一目了然。

此处的 Printf，使用了"%08b\n"这样的格式字符串，把一个整数用二进制输出。

初始 i 使用的是:=变量短赋值的格式。i 是 int 类型，而不能像之前那样用 var 明确声明是 uint8 类型。因为<<左移需要一个 uint 类型，所以才有这里的明确转换。

3.5 遍历

使用 range 的 for 语句可以逐一遍历一个数组、切片、字符串、映射的每一项，或者从程道接收每一个值。每次遍历，range 会把下一项的遍历值赋值给遍历变量，然后执行大括号中的块。例如：

```
for i,v := range r {
    f(i,v)
}
```

range 右侧的表达式称为 range 表达式，可以是数组、数组指针、切片、字符串、映射或程道。因为使用赋值，左侧的遍历变量必须可取地址，或是映射下标表达式。遍历变量可以是一个或两个，如果第二个遍历变量是个_空标识符，则可以忽略，等同于只有一个遍历变量。如果 range 表达式是个程道，则只可以用一个遍历变量。

range 表达式在循环执行前已经完成求值，但对于数组则有可能不需求值。例如只有一个遍历变量取数组下标时，在 range 左侧的函数每次循环都调用一次。对于每次循环，如果是数组、数组指针、切片，则从 0 到其长度 len-1 遍历，得到下标值和对应元素值。如果只有一个遍历变量，则不会取数组或切片对应元素的值。nil 切片不做遍历。而如果是字符串，则从字节 0 开始取得字符串中每个 Unicode 码值。第一个遍历值是其 UTF-8 字节的下标。如果有第二个遍历变量，则为 rune 类型的 Unicode 码值。如果遍历时遇到非法的 UTF-8 顺序，则第二个遍历值为 0xFFFD，并从下一个字节再次遍历。

3.6 标号和跳转

标号由一个标识符和冒号构成，可以作为goto、break和continue语句的目标。例如：

```
allDone: return
```

break 终止最内层的 for、switch 和 select 语句。如果后跟一个标号，标号只能是针对 for、switch 和 select 语句。break 会终止那一层的语句执行。例如：

```
L:for i < n {
    switch i {
    case 5:
        break L
    }
}
```

continue 跳到最内层 for 循环的增值语句部分执行。如果后跟一个标号，标号只能是针对 for。continue 会跳到那一层 for 循环的增值语句部分执行。

goto 直接跳到标号给定的语句执行。它不能跳过变量声明，也不能跳进一个块的内部。例如下面两个例子都会编译报错：

```
goto L  //错误
v := 3
L:
if n % 2 == 1 {
    goto L1
}
for n > 0 {
L1: f()
}
```

3.7 作用域

每个变量的作用域，从它声明之处开始，到所在的块结束为止。与块外的同名变量无关。例如我们修改上面的 for 循环，看看变量 i 的值最终是什么：

```
package main

import "fmt"

var i = "i"

func main() {
    i := i
    for i := 0; i < 2; i++ {
        {
            i := i + 100
            fmt.Println(i)
        }
    }
    fmt.Println("i = ", i)
    //Output:
    //100
    //101
    //i =  i
}
```

这不奇怪。每个大括号都代表一个块；每个块里的变量都是独立的，尽管它们可能有相同的名字。一方面，这种块作用域规则可以保护块内的变量不受它的使用环境的影响，也就是说，块外可以有同名变量，而且块内声明的变量也不可以在块外使用。但另一方面，同名变量和作用域规则增加了程序的复杂度，程序员必须清楚每个变量的有效范围。

每个包内函数外部声明的变量，称为包变量，它的作用域是整个包。

函数的参数和返回的变量以及 if、switch、for 所在行声明的变量，其作用域是紧随的大括号。case 分支所声明的变量的作用域仅限其所在分支。

同一个作用域内不可出现同名变量，也不可以声明变量后不使用。这些都是常见的编译时错误。

3.8 小结

if、switch、for 是 Go 编程最常用的控制方式。灵活地使用它们来操作各种数值表达式，就可以完成所有计算机可以完成的功能。当然，Go 语言除了提供这些最基本的控制语句，还把函数提升到更高的层级，可以帮助程序员写出更复杂、更清晰、更可靠的程序。下一章将介绍 Go 语言的函数。

第4章 函　数

The chief function of the body is to carry the brain around.

（身体的首要功能是驮着脑子。）

——Thomas Edison

函数把大块的语句分成一些小块，使程序员能利用已经做好的一些小块，而不需要每次都从头创建。适当的分块并提供合适的界面，使函数内部的语句与其运行的环境尽量隔离，程序员只需明白怎样操作这些界面，而不必研究函数的实现，这样整体上程序员可以更容易的组合这些预制件，而不用自己动手创建最基础的部分，从而能更可靠、更快速地完成更大型的项目。这就是结构式编程最重要的概念。

4.1 签名

每个 Go 程序最先执行的函数不是 main，而是 init 函数。但 init 是可以没有的，而 main 必须要有。所以我们说，Go 程序是从 main 包的 main 函数开始的。

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

函数的声明用 func，后面是函数名，小括号里的是参数，再后面可以有返回值类型或变量，最后可以用大括号里的语句作为函数体。如果声明时没有函数体，通常它声明的是 Go 要使用的其他语言的函数，例如一个汇编语句例程。Go 函数的声明顺序没有限制，只要调用的 Go 函数在同一个程序中就可以了。

作为程序入口的 main 函数没有参数也没有返回值——这是 main 函数的签名（signature）。一个函数的签名就是它的参数类型和返回值类型，是一个函数与使用环境的界面，与它们的名字无关。相同签名的函数是同一类型的。相同类型的变量才可以赋值，Go 支持函数类型的变量和赋值。例如：

```
package main
import "fmt"
var pr = fmt.Println
var f = func(){
    pr("Hello")
}
var g func()
func main() {
    g = f
    g()
}
```

4.2 参数

参数是函数的输入，返回是输出。对函数来讲，参数就是局部变量，也就是在函数体内声明的变量。只是，局部变量的初始赋值只能在函数内部进行，而参数的初始赋值，由此函数的调用者来完成。例如：

```
package main
import "fmt"
func say(s string){
    fmt.Println(s)
}
func main() {
    say("Hi")
}
```

say 函数的参数是字符串类型的变量 s,但初始赋值是在 main 调用 say 的时候，把字符串值"Hi"赋值给字符串类型的变量 s 的。同样，s 的值赋给 Println 函数的参数。如果我们使用 godoc fmt Println，可以看到它的签名是：

```
func Println(a ...interface{}) (n int, err error)
```

这个...表示 Println 的参数 a 可以接受多个值,这个 interface{} 表示 a 是空界面类型，也就是可以接受几乎所有类型的值。

通常我们会具体规定参数的个数、名称、类型，作为函数的基本文档。例如 godoc time Date:

```
func Date(year int, month Month, day, hour, min, sec,
nsec int, loc *Location) Time
```

这个函数声明按顺序给出 8 个参数的含义和类型，这是一个清楚的函数声明，尽管 Date 完成的功能并不像签名看起来那么容易。

有了清楚的声明，就很容易使用：

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

func main() {
 t := time.Date(2009, time.November, 10, 23, 0, 0, 0,
time.UTC)
 fmt.Printf("Go 发布于%s\n", t.Local())
}
```

作为比较，我们看看 Objective-C 的 date 函数的声明和使用：

```
NSString *dateStr = @"20091110";
NSDateFormatter *dateFormat=[[NSDateFormatter alloc]
init];
[dateFormat setDateFormat:@"yyyyMMdd"];
NSDate *date = [dateFormat dateFromString:dateStr];
[dateFormat setDateFormat:@"EEEE MMMM d, YYYY"];
dateStr = [dateFormat stringFromDate:date];
[dateFormat release];
```

我们不需要知道它在做什么，这里只是说明，同样的功能，用不同语言来描述可以有非常明显的差别。

4.3 返回语句

纯粹数学概念上的函数，功能是把输入转换成输出，而不会受输入之外环境因素的影响，也不会改变输出之外的环境因素。如果一个 Go 函数可以这样纯粹，那它和其他函数组合使用就比较容易，不用担心签名界面之外的因素会妨碍它们的结合。

但计算机是和物理世界紧密联系的。例如人机界面，就不可能是纯函数，Println 输出的除了我们通常不关心的处理的项数和错误，它必须输出字符串给我们看。这个必须的输出不在 Println 的签名里，不是它的返回输出，而是副作用。

其实对于每个 Go 程序，我们都是在利用它的副作用，因为 main 函数的签名也没有返回值。这和 C 程序不同。C 程序的 main 函数会返回一个整数值指出程序的错误。在 Go 里，这需要用到 os 包的 Exit 函数：

```
func Exit(code int)
```

和函数的输入参数一样，多个返回变量也需要放在小括号里。

例如 ioutil 包的 TempFile 函数：

```
func TempFile(dir, prefix string) (f *os.File, err
error)
```

它使用参数 dir 和 prefix 两个字符串，在 dir 目录里创建一个名称以 prefix 开始的临时文件，并打开以供读写。它返回一个*os.File 类型的值，以及一个 error 类型的值。注意返回的是值，而不是变量 f 和 err。这两个变量是函数内部使用的局部变量，它们的值被返回。我们直接看 Go 提供的 TempFile 的实现，并略作简化：

```
    func TempFile(dir, prefix string) (f *os.File, err
error) {
      name:= filepath.Join(dir, prefix+nextSuffix())
      f, err = os.OpenFile(name, os.O_RDWR|os.O_CREATE|
                                os.O_EXCL, 0600)
      return
    }
```

变量 f 和 err 直接被赋 os.OpenFile 的返回值，这样直接 return 就可以了。

因为函数返回的是值，Go 不要求在函数声明时声明返回变量，可以只是类型，但此时，return 后就必须给出每一个需要返回的值。例如 TempFile 函数还可以写成：

```
    func TempFile(dir, prefix string) (*os.File, error) {
            name := filepath.Join(dir, prefix+ nextSuffix())
            return os.OpenFile(name, os.O_RDWR| os.O_CREATE|
os.O_EXCL, 0600)
    }
```

如果函数只需返回一个值，则可以不加小括号。我们已经多次见到过这种用法了。

4.4 函数调用

函数调用（function call）就是利用函数 f 为函数 g 的参数赋值，然后 f 停下，执行 g，直到 g 返回，f 恢复执行，处理 g 的返回值。所以程序的执行是顺序的，并没有同时执行 f 和 g。例如：

```
package main

import "fmt"

func g(oo int) int {
    oo--
    return oo
}

func main() {
    o := 42
    c := g(o)
    c++
    fmt.Println(c,o)
}
```

我们可以利用图示来了解 main 调用 g 时参数和变量的使用情况。如果能够理解 Go 是怎样处理函数调用的，就能对程序的执行效率和瓶颈有个清楚的认识。需要说明的是，由于编译器优化等因素，此处的图示并非一定是程序执行时变量的实际分配。

首先，一个顺序执行的程序的所有函数，都共享同一片内存，用来存放函数的局部变量，以及在函数间传递参数和返回值，甚至还包括当前执行的函数返回时要执行的指令地址（称为返回地址）。这片内存称为栈。注意平常所说的“堆栈”是堆（heap）和栈（stack）的总称，是两片不同的内存，是 Go 程序管理内存的两种方式。

先看 main 开始时栈的情况。main 有两个 int 类型的局部变量，它们的初始值都是 0，所以有：

```
| 0 | o :main
| 0 | c
```

main 的变量 o 的值复制给 g 的参数 oo，然后调用执行函数 g，此时，栈里同时有函数 main 和函数 g 的局部变量——称为框架（frame）。函数 g 的框架正在使用，而 main 的框架在调用 g 时，被 g 的框架压住，不能被 g 使用：

```
| 42 | o    :main
| 0  | c
| 42 | oo   :g
```

g 执行 oo--，返回此时 oo 的值给 main，赋值给 c。此时的栈回到了调用 g 之前的状态，g 的框架不见了，main 也不能使用 g 的变量：

```
| 42 | o    :main
| 41 | c
```

这种通过栈和框架来完成函数参数传递的方式，从 40 年前的 C 语言首先采用，至今仍是 C 类语言和 Go 语言最基本的函数调用方式。而 Go 除此之外，也支持函数的闭包和函数类型的变量。

4.5 闭包

和变量的声明不同，Go 语言不能在函数里声明另外一个函数。所以在 Go 的源文件里，函数声明都是出现在最外层的。“声明”就是把一种类型的变量和一个名字联系起来。Go 里有函数类型的变量。这样，我们虽然不可以在一个函数里直接声明另外一个函数，但可以在一个函数中声明一个函数类型的变量。此时的函数称为闭包（closure）。例如：

```
package main

import "fmt"

func main() {
    add := func(base int) func(int) int {
        return func(n int) int {
                    return base + n
            }
        }
    add5 := add(5)
    fmt.Println(add5(10))
}
```

这个例子唯一的实用价值大概就是用来展示闭包的构建和使用。

add 是一个闭包，因为它是无名的函数类型的变量。可以认为它是一个闭包作坊。类似面包作坊，它会根据参数返回一个闭包。这样 add5 就是使用 5 作为 add 的参数得到的一个闭包。

闭包的声明是在另一个函数的内部，形成嵌套。和块的嵌套一样，内层的变量可以遮盖同名的外层变量，而且外层的变量可以直接在内层使用。例如，add 的 base 参数处在 return 返回的闭包的外层，所以它的值 5 在 add 返回并赋值给 add5 后依旧存在。当 add5 执行时，参数 n 可以和这个从外层得到的 base 相加，得到结果 15。

PDG

如果联想到上一节介绍的函数调用，会奇怪 add 函数 return 之后的框架已经不存在了，为什么 base 还可以用呢？答案是 Go 的编译器会把闭包使用的外围变量分配到堆上，而堆上的变量，是不会随函数返回自动消失的，它们在用完之后，才会被垃圾回收。

4.6 压后

压后 defer 语句注册一个函数。当 defer 所在函数返回时，调用 defer 注册的函数。例如：

```
func f(){
    err := lock(l)
    defer unlock(l)
    if err { return }
    go()
}
```

defer 确保 unlock 一定会在 f()执行结束时执行。

defer 的表达式必须是函数或方法调用。每次 defer 语句执行时，对函数的参数求值但不会立即调用函数。压后语句注册的函数要等到 defer 所在的函数返回时，才按照 LIFO（后进先出）的次序依次执行。例如：

```
// 返回之前输出 3 2 1 0
for i := 0; i <= 3; i++ {
    defer fmt.Print(i)
}
```

注意，此时 defer 所在的函数的返回变量已经求值，因为 defer 调用是 return 的最后一步。例如：

```
// f 返回 2，因为返回变量 result=1 发生在 defer 的 result++之前。
func f() (result int) {
        defer func() {
            result++
        }()
        return 1
}
```

PDG

4.7 派错和恢复

Go语言提供两个内置函数来处理运行时的程序错误：panic 结束函数的执行并报告一个错误信息；recover 恢复被 panic 的函数。例如：

```
package main

import "log"

func protect(g func()) {
    defer func() {
        log.Println("done")
        if x := recover(); x != nil {
            log.Printf("run time panic: %v", x)
        }
    }()
    log.Println("start")
    g()
}

func main() {
    var s []byte
    protect(func() { s[0] = 0 })
    protect(func() { panic(42) })
    s[0] = 42
    //Output:
    //2012/02/29 22:52:04 start
    //2012/02/29 22:52:04 done
    //2012/02/29 22:52:04 run time panic: 42
    //panic: runtime error: index out of range
}
```

运行出错可以是数组或者下标访问越界，也可以是我们人为在程序里直接调用 panic 函数。效果都是一样的。不仅运行出错的那个函数立即结束，连调用它的函数也跟着立即结束，这样一直结束到 main 函数，

程序退出并显示错误信息。

注意，函数结束时一定会执行 defer 注册的函数。利用这个设计，recover 可以在 defer 执行时，得到出错的值，或者 panic 函数的参数。重要的是，recover 终止上述函数的结束过程，而继续执行 defer 所在函数的下一条语句。

综上所述，上例的 protect 函数，是把参数 g 包装起来。g 是一个函数类型的变量，在 protect 的最后被调用。如果 g 函数运行出错，它会引发 protect 函数跟着 panic。但因为 protect 有 defer 函数，并且 defer 有 recover 函数保护，protect 的 panic 会执行到 recover，得到 panic 的值，用 log 打印出来，然后正常返回到运行出错 panic 的下一条语句——也就是 g() 之后的（隐含的）return 返回语句。作为演示，我们 main 函数的第一个 protect 调用的无名函数会产生切片下标越界的运行错误，第二个 protect 调用，直接在无名函数中执行 panic 让它出错。可是这两种运行错误因为有 defer 的 recover 保护，都可以回到 main 函数继续执行。但下一行的切片赋值也同样会出错，因为没有 defer 的 recover 保护，导致 main 函数非正常退出，输出一堆出错信息，包括出错原因和执行栈的每层的函数，供我们分析。

4.8 方法

Go语言是这样规定方法的：

“方法就是一个有接受者的函数”。

实际上，尽管写法不同，编译器是把方法作为函数来编译的，而方法的接受者，被安排成了这个函数的第一个参数。

例如：

```
package main

import "fmt"

type Point struct {
    x, y float64
}
type Rect struct {
    min, max Point
}

func (r *Rect) Area() float64 {
    w := r.max.x - r.min.x
    h := r.max.y - r.min.y
    return w * h
}
func main() {
    var r Rect
    r.max = Point{2, 2}
    fmt.Println(r.Area())
}
```

下面这个方法和对应的普通函数其实是按相同的方式执行的：

```
func (r *Rect) Area() float64
func Area(r *Rect) float64
```

但不同的写法使编译器知道 Area 不是普通的函数,而是属于*Rect 类型的方法。其他的类型也可以有名为 Area 的方法，它们是完全无关的。例如我们可以补充声明：

```
func (r Point) Area() float64 { return 0 }
```

*Rect 和 Point 类型的方法 Area 的接受者类型的基础类型分别是 Rect 和 Point。对于指针类型的接受者，编译器会自动对一个变量取地址，再调用其方法。所以上例的 r.Area()，实际是 Area(&r)，反之亦然。例如：

```
package main
import "fmt"

type Point struct {
    x, y float64
}
type Rect struct {
    min, max Point
}

func (r *Rect) Area() float64 {
    w := r.max.x - r.min.x
    h := r.max.y - r.min.y
    return w * h
}
func (r Point) Area() float64 {
    return 0
}
func main() {
    var r Rect
    r.max = Point{2, 2}
    fmt.Println(r.Area(), (&r).Area(), (*Rect).Area(&r))
    p := &r.min
    fmt.Println(p.Area(), (*p).Area(), Point.Area(r.min))
}
```

从这个例子中我们可以清楚地看出，在方法调用上，Go编译器提供了指针类型和基础类型接受者自动转换的帮助。但我们自己也要很清楚这两种类型接受者的不同：两者都是对应函数的第一个参数，都是通过复制方式赋值。但复制指针的结果变量，可以共享指针指向的内存，可以直接在方法内改变那个内存变量的值。而复制基础变量的值，则没有共享，可以安全使用。但注意不要复制太大的数据类型。

另外，接受者的基础类型不能是指针类型，也就是说，**T 是不可以的。它也不可以是界面类型，或者是在另一个包里声明的类型。

“一个类型可以有相关的方法集合。界面类型的方法集合是其界面。而其他各种具体类型 T 的方法集合,是以 T 作为接受者类型的所有方法。而对应指针类型*T 的方法集合，是所有以*T 和 T 作为接受者类型的方法。也就是说*T 的方法集合包括 T 的方法集合。任何类型都有一个空方法集合。在一个方法集合里，每个方法不能重名。”

这是《Go语言规范》对类型和方法的关系的规定，也是“界面”这个概念的基础。这是Go语言动态编程的基础。下一章会详加说明。

4.9 包

Go 的源文件在最开始要声明自己所属的包，也就是说它必须要属于某一个包。到目前为止，我们所写的都是 package main 的程序，并用 import 导入其他包的函数和变量、常量加以使用。main 包的特殊性在于，它一定有一个 main 函数，作为一个程序执行的起点。没有 main 函数的包，是不能被编译成可执行文件的，只能导入到其他的包里使用。

Go 语言规范里规定：Go 语言的程序是由一系列的包链接构成的。每一个包又由一系列的源程序文件构成。这些文件使用同样的 package 声明，并要求在同一目录下。它们共同声明了一个包的常量、变量和函数。在同一个包的不同文件里可以随意使用这些变量、常量和函数。但需要导出，才可以在其他包里使用。

能够导出的常量、变量和函数，其名称的首字母必须是使用 Unicode 编码的大写字母。这包括大写的英文字母、希腊字母、斯拉夫字母，但不包括汉字。这是因为汉字不区分大小写。所以很不幸，Unicode 规定汉字全部不是大写字母，Go 的包也就不能导出以汉字开头的名字了。有人在 Go 官方的问答录 FAQ 中建议使用“X 日语”这样不雅的名字——我们反对 X 任何东西。还是避免使用汉字变量函数名为妙。毕竟，Go 程序应该使用简单的英文，应该是世界通用的，而没必要无形中竖起另一道墙，让外人看不透。

英文单词普遍应用在所有的包里。例如 package math。尽管可以并且真的有人建议使用像 π 这样的符号表示圆周率 3.141 59，但 Go 的包使用 math.Pi 表示。

math 包是个很好的例子。在 src/pkg/math 目录下有 44 个 Go 源文件，全部都声明为 package math。它们和 67 个汇编语言的源文件，

共同构成了高效的数学包。通常，Go 的文件声明导出函数名，以及一个 Go 语言实现的通用内部函数。而针对每一种 CPU 体系，则通过汇编语句直接使用机器指令得到执行速度最快的函数。例如取绝对值的函数 Abs 的 abs.go：

```
package math

func Abs(x float64) float64

func abs(x float64) float64 {
    switch {
    case x < 0:
        return -x
    case x == 0:
        return 0
    }
    return x
}
```

而对应 Intel 386 的实现 abs_386.s 的汇编代码是：

```
// func Abs(x float64) float64
TEXT ·Abs(SB),7,$0
FMOVD   x+0(FP), F0   // F0=x
FABS                  // F0=|x|
FMOVDP  F0, r+8(FP)
RET
```

对应 ARM 的汇编源文件 abs_arm.s 则只使用 abs.go 里面的通用 Abs 函数：

```
TEXT ·Abs(SB),7,$0
    B ·abs(SB)
```

4.10 导入

import 声明这个 Go 的源文件会使用被导入包导出的变量、常量和函数。同时，它也指出了在这个源文件中如何使用这些导出的内容，以及通过哪个路径可以找到需要导入的包文件。例如：

```
import "math"

var pideg = math.Pi / 360
```

当编译器看到 import 时，它会到$GOROOT/pkg 下面找相应的包文件，例如此处的 math.a。包文件包装好已经编译过的函数和类型信息，还有这个包导入并且使用的其他包的函数，等等。这样，就不存在一连串导入其他包的问题，也就是说，包的依赖关系已经在编译包的时候整合到了这个包里，这是 Go 的编译器能够快速编译的关键。

如果使用语句 import "io/ioutil"，则需要到 io 子目录下查找 ioutil.a。但使用时，只需文件名而不需路径，也就是只需 ioutil.ReadFile。这样，Go 有时使用很长的路径名。例如：

```
import "github.com/bmizerany/pat.go"
var m = pat.New()
```

Go 语言的 go 命令，就自动使用这样的规则，自动下载并编译分散在因特网上的包。读者可以参考附录，或者 Go 的官方网站，学习如何编写自定义 Go 语言包的源程序，使之可以自动被 go 命令使用。

4.11 程序初始化

无论是声明，还是使用make或者new得到的变量值的内存，当没有明确给出初始值时，内存自动清零，称为这个类型变量的“零值”。每个类型的零值规定如表4-1所示。

表4-1 零值表

零　值	类　型
false	布尔型
0	整型
0.0	浮点型
""	字符串
nil	指针、函数、界面、切片、程道和映射

每个类型零值的初始化是递归的。所以复合类型，例如结构体数组的每个单元，如果没有指定初始值，则都会赋予零值。

例如，下面的两个简单声明是等价的：

```
var i int
var i int = 0
```

执行下面的语句，也会产生零值：

```
type T struct {
    i int;
    f float64;
    next *T
}
t := new(T) // 或者 var t T
// t.i == 0
// t.f == 0.0
// t.next == nil
```

程序开始执行时，如果一个包没有导入其他的包，则先初始化包一级的变量，然后调用每一个本包内的 init() 函数。多个 init 函数甚至可以出现在同一个源文件中，它们的执行顺序没有规定。

其他变量和常量的初始化顺序要看它们的依赖关系：如果 A 的初始值要依靠 B 的值，则 A 会在 B 之后初始化。如果它们互相依赖成为一个循环，编译器会报错。依赖分析是通过词法分析完成的：如果 A 里提到 B，或者 A 中一个值在初始化时提到 B，或者 A 提到的一个函数提到 B，等等；都会使 A 依赖 B。如果两个值不相关，则按在源代码中出现的先后次序初始化。注意，因为依赖分析是在一个包内进行的，如果 A 需要靠另一个包里的 B 来完成初始化，则可能不会出现预期结果。

init 函数不可以直接调用，也不可以在程序中引用。例如，不可以把 init 函数的指针赋值给一个函数变量。一个包导入的其他的包会先做初始化。如果一些包都导入同一个包，则此包只初始化一次。而导入包的这种安排是不可能会在初始化时产生循环依赖的。

这样一个完整的程序，就是从一个 main 包开始，然后把它导入的包链接在一起得到的。这样的 main 包必须有一个 main 函数，此函数没有参数也不返回值。而这个程序执行时，会先初始化 main 包，再调用 main 函数。当 main 函数返回时，程序结束。程序的初始化，也就是变量的初始化加上调用那些 init 函数，是在一个去程内完成的，每次一个包。init 函数可以启动其他的去程，它们和初始化去程是并行的。但这些 init 函数自身是顺序执行的；此 init 执行完成之前，不会开始执行下一个 init 函数。

4.12 小结

本章接触了结构式编程的核心——函数。也介绍了函数式编程的核心——闭包。还有动态类型编程和面向对象编程的核心——方法。这些使得函数成为Go语言的核心。我们必须逐步学会如何把实际问题抽象为一个个函数，如何不断地调整每个函数，使之能更好地与其他的函数相配合。还要掌握如何包装函数，让其他程序员导入使用。这些需要大量实践，也需要大量阅读其他优秀程序员的作品。Go的包是开源的，仔细研究每一个Go包的函数的设计和实现，不仅可以提高我们使用这些包的能力，更能提高我们的分析能力并培养思维习惯。后者，对提升自身的修养是至关重要的。借用一句俗语共勉：

“师父领进门，修行在个人。”

第 5 章
动态类型

Languages that try to disallow idiocy
become themselves idiotic.
（避免白痴行为的语言本身变成了白痴语言。）
—— Rob Pike

严格的类型系统使得编译器可以自动检查程序员有没有做出不明智的举措。但过于严格的类型系统却使编程变得太过复杂，程序员不得不一直和编译器讨价还价，才能把自己的想法硬塞进那种语言规定的框架中。

Go 语言试图让程序员能在安全类型检查和灵活类型使用之间取得平衡。它提供了映射和界面类型，以及运行反射系统和类型切换，使安全动态编程变得更容易。本章将仔细分析 Go 的这些特点。

5.1 映射

映射（map）在各种计算机语言中有广泛的使用，与数学中的映射概念相对应。但大多是作为外部库函数，而且叫法和用法不一：某些语言称之为字典（Dictionary）；某些称之为散列表（Hash Table）；有些称之为表。Go和Java、C++一样，都称之为映射。

映射指的是从一个类型的值到另一个类型的值的对应关系。例如，我们关注一下各国的GDP。可以用切片：

```
package main

import "fmt"

const (
    China int = iota
    USA
    Japan
    Total
)

type GDP struct {
    k string
    v float64
}

func main() {
    GDPof := make([]GDP, Total)
    GDPof[China] = GDP{"China", 5.92}
    GDPof[USA] = GDP{"USA", 14.58}
    GDPof[Japan] = GDP{"Japan", 5.45}

    for k, v := range GDPof {
        fmt.Printf("%d. %s:%g\n", k, v.k, v.v)
    }
```

```
    //Output:
    //0. China:5.92
    //1. USA:14.58
    //2. Japan:5.45
}
```

也可以使用映射：

```
package main

import "fmt"

func main() {
    GDPof := map[string]float64{
        "USA":   14.58,
        "Japan": 5.45,
    }
    GDPof["China"] = 5.92
    for k, v := range GDPof {
        fmt.Printf("%s:%g\n", k, v)
    }
    //Output:
    //USA:14.58
    //China:5.92
    //Japan:5.45
}
```

很容易发现，映射单元的读写类似于切片的下标操作。只是切片的下标必须是数字，而映射的下标称为键（key），几乎可以为任何类型。

一个新声明的映射变量的值是 nil，还不可以读写。这一点也类似于切片，映射需要明确地分配底层内存。而我们可以用两种方法，完成这个分配：

```
var m0 = make(map[string]bool)
var m1 = map[string]bool{}
```

第二种方式是直接指定映射变量单元的初始键值。因为映射变量没有底层内存的限制，可以随时添加，也可以使用 delete 函数删除。例

如删除映射 m 的 k 键单元可以用:

```
delete(m, k)
```

映射的单元个数也和切片一样,称为长度,可以使用 len 函数得到。映射可以使用 for 和 range 遍历,但遍历是无序的。如果一个单元还未被遍历就被删除,则其对应的遍历值不会出现。而如果在遍历映射时插入一个新单元,这个单元的遍历值最多出现一次。

和切片一样,range 如果赋值给两个变量,则每次循环分别得到单元的键和值;而如果是赋值给一个变量,则只是得到键。例如:

```
for k, v := range m {
    f(k, v)
}
for k := range m {
    f(k, m[k])
}
```

类似的赋值还被用来检查一个键是否存在。例如:

```
if v, ok := m[k]; ok {
    fmt.Println("Got", v)
} else {
    fmt.Println(k, "not exists")
}
```

因为映射属于内存的共享,为了减少多个并发程序同时读写映射造成冲突,有时可以用到看似复杂的映射结构切片的组合类型。例如,如果只使用英文字符串做键,可以按首字母分为具有 26 个单元的切片,每个单元是单独加锁的映射结构:

```
package main
import (
    "fmt"
    "math/rand"
    "sync"
```

```
    "time"
)
type Lockmap struct {
    sync.Mutex
    m map[string]int
}
var lockmap = make([]Lockmap, 10)

func init() {
    for i := range lockmap {
        lockmap[i].m = make(map[string]int)
    }
}
func counter(s string) {
    i := int(s[0] - '0')
    time.Sleep(time.Duration(i) * time.Millisecond)

    lockmap[i].Lock()
    defer lockmap[i].Unlock()
    lockmap[i].m[s]++
}
func main() {
    for i := 0; i < 20; i++ {
        r := fmt.Sprintf("%d", rand.Uint32())
        go counter(r)
    }
    time.Sleep(time.Second)
    for i := range lockmap {
        for k, v := range lockmap[i].m {
            fmt.Printf("[%d] %s = %d\n", i, k, v)
        }
    }
}
```

5.2 界面类型

Russ Cox 曾经说过，如果让他挑选 Go 的一种语言特色移植到其他语言中，他会选择界面（interface）。

这是因为，虽然界面的概念和 Java 的接口与 Objective-C 的协议（protocol）有些类似，但 Go 语句的界面概念更清晰，表示它可以和各种类型直接互动、扩展。

界面是类型。界面类型的变量可以赋值。任何实现了此界面的具体类型的变量，都可以赋值给界面类型的变量。

这三句话需要大段的文字解释。

界面类型声明的不是数值，也不是数值的组合，而是函数的组合。例如：

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

这个界面类型是两个函数的组合，声明里只需给出函数的签名。习惯上我们用这些函数名+er 来命名一个界面类型，例如上面的界面类型可以叫 ReadWriter。

实际上，界面类型可以只有一个函数，而且类似于结构内置体（embedded struct），界面也可以内置。例如在 io 包里定义了 Reader、Writer 和 ReadWriter 等多个基本的 io 界面类型：

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

type Writer interface {
```

```
    Write(p []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

界面类型的变量可以赋值，这主要用在函数参数中。例如，fmt 包的 Fprint 的签名为：

```
(w io.Writer, a ...interface{})(n int, err error)
```

它的两个参数都是界面类型变量。第一个 w 是 io.Writer 类型，第二个 a 是空界面类型的变长参数。Fprint 的功能是把 a 的每一项的值用默认的格式写入 w。

那 w 和 a，也就是界面类型的参数，可以赋什么样的值呢？Go 语言的一个原则就是必须类型相同才可以赋值。所以 w 和 a 被赋的值也必须是界面类型的值。具体地讲，就是这个值的类型的方法集合里有界面声明的那些函数。我们称此具体类型“实现”了此界面。例如，os 包的 File 类型有下列方法：

```
func (f *File) Write(b []byte) (n int, err error)
```

如果不包括接受者，它的签名和 io.Writer 声明的一样，所以 *File 类型实现了 io.Writer 界面。它作为 io.Writer 这个界面类型的具体类型，可以赋值给 fmt.Fprint 的第一个参数 w，这样 Fprint 就可以使用这个 File 实现的 Write 方法，对文件执行写操作了。

而 interface{}空界面类型声明了 0 个函数，是任何具体类型都实现了的界面，所以 Fprint 的第二个参数，可以被赋任意类型的值，而且因为它是变长参数，所以可以被赋任意多个这样的值。

例如：

```
fmt.Fprint(os.Stderr, "Write ", 42)
```

os.Stderr 是*File 类型，对应着标准错误输出。标准错误输出、标准输入和标准输出，是每个 Go 程序运行时都自动打开的 3 个文件，默认地连接到键盘输入和命令行输出。

从上面的介绍可以看出，界面类型和具体类型是多对多的对应关系。我们就从它们各自的角度，看它如何与对方实现一对多的对应。

5.3 界面值

Go的界面类型不是抽象的，这不只是一个规定，而且它有具体的值。一个界面类型的值是两个指针的组合，分别指向具体类型的值，以及那个具体类型实现的、此界面规定的方法：

```
| *itab | 方法表指针
| *data | 数据指针
```

我们用下面的鸭和电鸭的例子来说明：

```
package main

import "fmt"

type Duck float32

func (Duck) Quack() string {
    return "嘎"
}

type Educk complex128

func (Educk) Quack() string {
    return "叮咚"
}

type Quacker interface {
    Quack() string
}

func main() {
    var d Duck = 0.
    var e Educk = 0i
    var q Quacker
```

```
    fmt.Println(d, e, q)
    q = d
    fmt.Println(q.Quack())
    q = e
    fmt.Println(q.Quack())
    //Output:
    //0 (0+0i) <nil>
    //嘎
    //叮咚
}
```

变量 q 是 Quacker 界面类型，此时还没有具体类型，所以值是 nil。变量 d 和 e 是具体类型，它们都只有一个方法 Quack。

```
| 0.0 | d
| 0.0 | e
|     |
| 0.i |
|     |
| nil | q
| nil |
```

变量 d 赋值给 q，q 的*itab 指向 d 的 Quack 方法，*data 存放着 d 的值。

```
|  0.0      | d
|  *        | q --.
|  0.0      |     |
|  d type   | <---'
| *d.Quack  |
```

因为 d 的值是 32 位的，刚好可以放入*data 所占的内存中，所以会被优化，不存放指针，而是直接存放 d 的值。还因为通过 q 的*itab 可以看到 d 的类型描述，所以可以动态地得知 q 的*data 何时是值，何时是指针。

变量 e 赋值给 q，q 的*itab 指向 e 的 Quack 方法，*data 存放着

e 的值的指针。

```
|  0.0      | e <----.
|           |        |
|  0.i      |        |
|           |        |
| *itab     | q --.  |
| *data     |   --|--'
|  e type   | <---'
| *e.Quack  |
```

e 的 complex128 类型的值无法放入 q 的*data，所以必须使用指针间接得到。而 q 的*itab 已经指向一个新分配的内存，存放着 e 的类型描述和一个指向 e 的 Quack 方法的指针。而之前*itab 指向的 d 的那块描述内存，如果没有其他的指针指向它，最终会被垃圾回收。

注意，方法实际上就是函数。函数的指针，就指向函数编译后得到的、已经加载入内存的那段机器指令。所以从*itab 执行的方法和直接在编程时调用函数，结果是一样的，只是通过界面变量调用是“迟后绑定”，也就是界面变量在运行时可以赋不同的值，调用不同的方法，是“动态”的；而直接的方法调用被编译为“静态”的，不可以在运行时改变。

Go 语言通过界面变量实现的迟后绑定，有人称之为“鸭子类型”(duck type)。说的是，只要走起路来像鸭，叫起来像鸭，就把它当成鸭子。听着很搞笑，可面向对象就是如此。同一个界面变量可以当做不同类型的变量——只要它们都实现了那个界面类型规定的方法 Quack。

5.4 error 界面

我们再来看一个具体的界面类型的例子。Go 语言中的 error 类型是 Go 预声明中唯一的界面类型：

```
type error interface {
    Error() string
}
```

它广泛使用在函数和方法的返回值中。nil 表示没有错误，否则，可以通过它的 Error()得到出错信息，或者让 fmt 包的函数自动正确处理。例如：

```
file, err := os.Open("Nepo")
if err != nil {
    fmt.Println(err)
}
```

如果仅仅是这样，字符串类型就够用了，完全没有必要使用界面类型。但上面的 Open 如果出错，返回值实际上是个 os.PathError 结构：

```
type PathError struct {
        Op   string
        Path string
        Err  error
    }
```

而因为这个结构实现了 Error 方法，就可以作为 error 界面类型的具体类型，用在所有 error 的赋值中：

```
func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

需要指出的是，nil 可以是界面类型的值，也可以是指针、映射、

切片、程道类型的值。也就是说，必须分清是 nil 的界面值，还是 nil 的具体值。你可能已经听不懂了，还是举例说明吧：

```
package main
import "fmt"

type Err struct{}

func (_ *Err) Error() string {
    return "To err is human"
}
func ToErr(ok bool) error {
    var e *Err = nil
    if ok {
        e = &Err{}
    }
    return e
}
func NoErr(ok bool) error {
    if !ok {
        return &Err{}
    }
    return nil
}
func main() {
    fmt.Println(ToErr(true))
    fmt.Println(ToErr(false))
    fmt.Println(NoErr(true))
    fmt.Println(NoErr(false))
    // Output:
    // To err is human
    // To err is human
    // <nil>
    // To err is human
}
```

5.5 有界无类

听起来很像孔夫子的有教无类。孔夫子这简简单单的四个字，就让人捉摸了几千年，比面向对象的继承等还难理解。这个有和无，不是有名无实的布尔对立，也不是有过之而无不及的循环语义，好像与有意无意的分支语句也没关系，看起来更像有备无患这样的判断语句。高级语言的解释就是：

```
if 大家都能受教育 == true {
    原本不同类型的人 := 同样类型
}
```

这也可以理解为所有界面无需声明类型。不像某些面向对象的语言，必须在创建类时声明自己实现了什么接口，或者@interface 时在尖括号里使用了什么@protocol。Go 的界面类型没有对应具体类型，而只是规定一个子集。只要具体类型的方法集合包括这个子集，就可以赋值给界面类型的变量。而具体类型的方法集合，可以包括任意多个子集，也就是同时实现任意多个界面。试想如果 Go 像其他语言一样，需要在类声明的地方也声明所有实现的界面，那代码该有多么冗长。

例如 os 包的*File 类型，同时实现了 io.Reader、io.Writer、io.ReadWriter、io.ReadWriteCloser 和 interface{}等界面，只因为它实现了这些界面规定的方法。

因为每个不同名字的类型都是不同的，所以不可以像下面这样试图让 Goose 像 Duck 一样 Quack()：

```
type Duck float32
func (Duck) Quack() string {
    return "嘎"
}
```

```
type Goose Duck
type Quacker interface {
    Quack() string
}
func main() {
    var d Duck = 0.
    var q Quacker
    g := Goose(d)
    g.Quack() //g.Quack undefined
    q = g // Goose does not implement Quacker
    fmt.Println(q.Quack())
}
```

通过内置结构，就可以实现方法的继承。只需修改两处：

```
(1) type Goose struct {Duck}
(2) g := Goose{d}
```

5.6 排序

我们再用一个简单的例子看看界面的使用。在 sort 包里，界面是这样声明的：

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

然后声明了这些具体类型：

```
type IntSlice []int
type Float64Slice []float64
type StringSlice []string
```

每个类型都有 Len()、Less(int, int)bool 和 Swap(int,int)这 3 个方法。例如类型 IntSlice 的 3 个方法是：

```
func (p IntSlice) Len() int {
    return len(p)
}

func (p IntSlice) Less(i, j int) bool {
    return p[i] < p[j]
}

func (p IntSlice) Swap(i, j int)  {
    p[i], p[j] = p[j], p[i]
}
```

而 Sort 函数的签名是：

```
func Sort(Interface)
```

这样，我们可以用同样的 Sort 函数给不同类型的切片排序：

```
package main

import (
    "fmt"
    "math"
    "sort"
)

func main() {
    is := sort.IntSlice{3, 1, 4, 1, 5, 9, 2, 6}
    ss := sort.StringSlice{"士", "农", "工", "商"}
    fs := sort.Float64Slice{math.Inf(-1),
math.Inf(+1), math. NaN()}
    sort.Sort(is)
    sort.Sort(ss)
    sort.Sort(fs)
    fmt.Println(is, ss, fs)

    //Output:
    //[1 1 2 3 4 5 6 9] [农 商 士 工] [NaN -Inf +Inf]
}
```

我们自己练习一下，用 sort.Sort 给二维点排序，按照它的坐标到原点的距离，从小到大排列。这样的点实际上对应数学中的向量(vector)。参考程序如下：

```
package main

import (
    "fmt"
    "sort"
)

type Vec struct{ x, y float64 }

func (v Vec) lenlen() float64 {
    return v.x*v.x + v.y*v.y
}
```

```
type VecSlice []Vec

func (v VecSlice) Len() int {
    return len(v)
}
func (v VecSlice) Less(i, j int) bool {
    return v[i].lenlen() < v[j].lenlen()
}
func (v VecSlice) Swap(i, j int) {
    v[i], v[j] = v[j], v[i]
}

func main() {
    v0 := Vec{0, 0}
    v1 := Vec{1, 0}
    v2 := Vec{1, 1}
    vs := VecSlice{v1, v2, v0}
    fmt.Println(vs)
    sort.Sort(vs)
    fmt.Println(vs)

    //Output:
    //[{1 0} {1 1} {0 0}]
    //[{0 0} {1 0} {1 1}]
}
```

由于执行 `Less` 比较时不需知道实际的向量长度，而 `math.Sqrt` 取平方根会比较慢，所以此处我们的 `lenlen` 方法返回的是向量长度的平方，对于排序这已经够用（其实 `len` 已经够用），需要知道真正的长度时，才需要取平方根。结论是，如果我们对最底层的机器多一些了解，编程时有意识地加以利用，我们的程序就能快一点，内存就能少用一些。这不是“不成熟的优化”，而是成熟的程序员必备的常识。

您可能已经注意到，`Len` 和 `Swap` 方法的实现几乎完全相同，差别只在于接受者类型。这似乎很像其他面向对象语言的泛型（generic）。但

Go语言的作者认为泛型的适用性不强，无谓地加入只会增加语言的复杂程度，并可能减慢编译的速度。我们可以试着用切片和 append，以及像 Sort 这样用界面来增强程序的通用性，而不是试图一劳永逸地靠一个很难控制的泛型或者 STL 来解决每次都不一样的具体问题。

sort 包还提供了一些方便的函数给 int、float64 和 string 切片。但不方便的是，它们的 Less 只能按从小到大的顺序排序。如果要按从大到小的顺序排序过来，可以参考界面内置结构 Rev 的用法：

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

type Rev struct {
    sort.Interface
}
func (r Rev) Less(i, j int) bool {
    return r.Interface.Less(j, i)
}
func main() {
    a := []int{6, 7, 4, 2}
    sort.Ints(a)
    fmt.Println(a)
    sort.Sort(Rev{sort.IntSlice(a)})
    fmt.Println(a)
    // Output:
    //[2 4 6 7]
    //[7 6 4 2]

}
```

5.7 类型断言

如果 x 是界面类型的变量，T 是某个具体类型，则表达式 x.(T)判断 x 不是 nil，并且 x 的值的类型和 T 的类型是相同的。如果 T 也是界面类型，则这个表达式判断 x 实现了 T 规定的界面。

使用 v, ok = x.(T)赋值，当条件成立时，ok = true 而且 v = T。否则 ok=false 而且 v 是 T 类型的零值。如果不用上述赋值语句，而只是 x.(T)，条件不成立时，会产生运行错误，从而发现编程或者设计问题。也就是说，尽管 x 值的类型是运行时动态得到的，但我们预先可以在编程时指定并且在运行时检查它的正确类型。这就是“断言”的含义。

关于类型相同，Go 语言有具体的规定：给出类型名称的类型是彼此不同的，无名类型也是不同的。两个无名类型只有具备如下条件时，才认为是相同的。

(1) 相同单元类型和相同长度的数组类型是相同的。

(2) 具备相同单元类型的切片类型是相同的。

(3) 具备相同基础类型的指针类型是相同的。

(4) 具备相同键值类型的映射类型是相同的。

(5) 具备相同值类型和方向的程道类型是相同的。

(6) 具备同样顺序、名称、类型和标记项的结构体类型是相同的。无名项视为名称相同。

(7) 具备同样的参数和返回值个数，并且它们的每个参数和返回值的类型也相同，这样的函数类型是相同的。参数和返回值的名称可以不同。两个相同的函数必须都使用变长参数，或者都不使用。

(8) 具备相同的方法集合而且每个方法的名称和类型相同的界面类型是相同的。方法的次序无需相同。

PDG

例如：

```
package main

import "fmt"

type (
    T0  []string
    T00 []string
    T1  struct{ a, b int }
    T11 struct{ c, d int }
    T2  func(int, float64) *T0
    T22 func(int, float64) *[]string
)

var (
    t  interface{}
    t0 T0
    t1 T1
    t2 T2
)

func main() {
    t = t0
    {
        v, ok := t.(T0)
        fmt.Println(v, ok)
    }
    {
        v, ok := t.([]string)
        fmt.Println(v, ok)
    }
    {
        v, ok := t.(T00)
        fmt.Println(v, ok)
    }
    t = t1
    {
```

```
        v, ok := t.(T1)
        fmt.Println(v, ok)
    }
    {
        v, ok := t.(T11)
        fmt.Println(v, ok)
    }
    t = t2
    {
        v, ok := t.(T2)
        fmt.Println(v, ok)
    }
    {
        v, ok := t.(T22)
        fmt.Println(v, ok)
    }
    {
        v, ok := t.(T22)
        fmt.Println(v, ok)
    }
    // Output:
    // [] true
    // [] false
    // [] false
    // {0 0} true
    // {0 0} false
    // <nil> true
    // <nil> false
}
```

5.8 类型分支

类型分支比较的是类型而不是值。它的 switch 表达式类似类型断言(type assertion),但使用关键字 type 而不是实际的类型,所以 switch 表达式是界面类型的值,此界面类型的值所对应的具体类型,使 switch 能选择一个 case 分支或 default 分支。注意,fallthrough 不可以用在类型 switch 中。

除了可以在 switch 表达式前面执行一条简单语句,还可以让此表达式中包括一个变量短声明。此时,变量是声明在每个分支里。如果一个分支只有一种类型,则为此变量的类型;否则,此变量为 switch 表达式的类型。

举例说明会比较清楚。例如下面的 switch 可以改写为 if-else,从而清楚地看到 i 和 v 的类型声明:

```
switch i := x.(type) {
case nil:
    printString("x is nil")
case int:
    printInt(i)  // i is an int
case float64:
    printFloat64(i)  // i is a float64
case func(int) float64:
    printFunction(i)  // i is a function
case bool, string:
printString("type is bool or string")
    // i is an interface{}
default:
    printString("don't know the type")
}
```

此处 case 为 nil,表示 switch 表达式的界面值是 nil 的分支。

此 switch 语句可以使用类型断言，改写为 if-else 语句：

```
v := x  //只计算一次 x 的值
if v == nil {
    printString("x is nil")
} else if i, is_int := v.(int); is_int {
    printInt(i)  // i is an int
} else if i, is_float64 := v.(float64); is_float64 {
    printFloat64(i)  // i is a float64
} else if i, is_func := v.(func(int) float64); is_func {
    printFunction(i)  // i is a function
} else {
    i1, is_bool := v.(bool)
    i2, is_string := v.(string)
    if is_bool || is_string {
        i := v
        printString("type is bool or string")
    // i is an interface{}
    } else {
        i := v
        printString("don't know the type")
    // i is an interface{}
    }
}
```

5.9 反射

Go 的反射包 reflect 让程序可以在运行时，通过类型系统，检查自己的数据结构。这个包我们很少会直接用到，但它提供的函数和类型，是 Go 语言动态类型系统的基础结构。很多 Go 标准包，例如 gob 包，就是在反射包的基础上建立的。此处我们进行简单介绍，一方面是加深对 Go 的动态机制的理解；另一方面，也是对前面几章的静态类型的回顾。

反射依靠的是界面变量，尤其是空界面 interface{}。例如：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.14
    fmt.Println("type:", reflect.TypeOf(x))
    fmt.Println("value:", reflect.ValueOf(x))
    // Output:
    // type: float64
    // value: <float64 Value>
}
```

界面变量在哪里？godoc reflect TypeOf 可以看一下签名：

```
func TypeOf(i interface{}) Type
```

我们可以很明显地看到它使用 interface{}做参数，也可以进一步得知返回值 Type 也是界面类型。我们可以将任意类型的变量赋值给这个界面参数。而界面变量的值，可以简单地理解为两个指针，分别指向其具体变量的值和类型。反射包的函数就提供了标准方法，让这些值和

类型为我们所用。

TypeOf 可以给出任意一个变量的类型，进而得知与这个类型相关的信息。因为每个类型的信息是不尽相同的，此处也没有必要一一审视。这里只是给出一个例子：

```
package main

import (
    "fmt"
    "reflect"
)

type X struct {
    y byte
    z complex128
}

func (x *X) String() string {
    return fmt.Sprintf("%v", x)
}

func main() {
    var x X
    fmt.Println(x)

    v := reflect.TypeOf(x)
    fmt.Println("type:", v)
    fmt.Println("Align:", v.Align())
    fmt.Println("FieldAlign:", v.FieldAlign())
    for i := 0; i < v.NumMethod(); i++ {
        fmt.Println("Method", i, v.Method(i).Name)
    }
    fmt.Println("Name:", v.Name())
    fmt.Println("PkgPath:", v.PkgPath())
    fmt.Println("Kind:", v.Kind())
    for i := 0; i < v.NumField(); i++ {
        fmt.Println("Field", i, v.Field(i).Name)
```

```
    }

    fmt.Println("Size:", v.Size())
    // Output:
    // {0 (0+0i)}
    // type: main.X
    // Align: 4
    // FieldAlign: 4
    // Name: X
    // PkgPath: main
    // Kind: struct
    // Field 0 y
    // Field 1 z
    // Size: 20
}
```

ValueOf 得到一个结构体 Value。而通过这个 Value 的值，我们可以得到与值相关的信息，以及具体的值。同样，我们举例说明：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.14
    v := reflect.ValueOf(x)
    fmt.Println("type:", v.Type())
    fmt.Println("kind is float64:",
    v.Kind() == reflect.Float64)
    fmt.Println("value:", v.Float())
    // Output:
    // type: float64
    // kind is float64: true
    // value: 3.14
}
```

实际上，通过 Value，还可以直接为这个变量赋值。但在赋值之前，需要确认这个具体变量是可以被赋值的。如果我们直接用 v.SetFloat(3.1415)去改变上例中 v 的具体值，就会产生运行错误。因为 v 是通过 reflect.ValueOf(x)得到的，是 x 的一个副本。即便是可以改变 v，也是不能够改变 x。而 x 才是我们要赋值的具体变量。

解决方案很简单——使用指针。可以这样：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.14
    fmt.Println("x:", x)
    v := reflect.ValueOf(&x)
    fmt.Println("v.CanSet:", v.CanSet())
    i := v.Elem()
    fmt.Println("i.CanSet:", i.CanSet())
    i.SetFloat(3.1415)
    fmt.Println("i:", i.Float())
    fmt.Println("x:", x)

    // 输出:
    // x: 3.14
    // v.CanSet: false
    // i.CanSet: true
    // i: 3.1415
    // x: 3.1415
}
```

这样，如果我们有了一个结构体变量的地址，我们就可以直接读写它的每一项。这就是 gob 包操作最底层的机制。它能够直接把一个变量

的值和类型编码打包成可以固化和传输的文字形式；也可以把这种文字形式拆包解码，赋值给一个变量。

下面展示如何通过反射包为一个结构体变量赋值：

```
package main

import (
    "fmt"
    "math"
    "reflect"
)

type X struct {
    Y byte
    Z complex128
}

func main() {
    var x X
    fmt.Println(x)

    v := reflect.ValueOf(&x)
    e := v.Elem()
    e.Field(0).SetUint('e')

    π := math.Pi
    c := complex(math.Cos(π), math.Sin(π))
    e.Field(1).SetComplex(c)
    fmt.Println("最美的公式:  ")
    fmt.Printf("%c^iπ + 1 = %g\n", x.Y, 1+real(x.Z))
    fmt.Println("可惜虚部有点精度误差:", imag(x.Z))
    // Output:
    // {0 (0+0i)}
    // 最美的公式:
    // e^iπ + 1 = 0
    // 可惜虚部有点精度误差: 1.2246063538223773e-16
}
```

这是用一个 Go 程序曲折地验证最美的数学公式——欧拉恒等式（Euler's Identity）: $e^{i\pi} + 1 = 0$。

公认它最美，原因有两个。

(1) 它包括三种基本运算各一次: 加、乘、幂。

(2) 它包括五种基本常量各一次: 数字 0, 1, π, e 和虚数单位 i。

而计算时, 我们又用欧拉公式展开: $e^{ix} = \cos(x) + i\sin(x)$。并且，变量、常量、函数这些数学名词都是欧拉的创造。但尽管计算机和数学有很深的渊源, Go 和 C 类语言的赋值表示却也让正统的数学家摇头不止。例如 `x = x + 1`，根本不可能成立！因为要加这个常量 1，便没有一个变量 x 的值能满足这个等式！

5.10 小结

作为 C 语言家族的一员，Go 使用架构在界面类型上的动态类型系统。这很新颖但也需要很多努力才能逐步掌握。只要想想 Go 的诞生正是出于对过于刻板的静态类型编译系统的不满，我们有理由相信，用好 Go 动态的鸭子类型，才能体会到 Go 在静态和动态之间的平衡。

第 6 章

面向对象

"You know what went wrong?" he shouted,
"You let your programmers do things
which you yourself do not understand."
（"你知道毛病在哪吗？"他嚷嚷着，
"你让程序员做的事你自己都不懂。"）
——C.A.R Hoare《皇帝的旧衣》

Go 语言的作者 Robert Griesemer 是 Smalltalk 和 Strongtalk 的大师，也是 Java Hotspot 编译器的作者。他在这些语言上的实践经验，塑造了 Go 语言面向对象和动态编程的独特一面。本章试图推开堆叠的旧衣，看看我们要面向的真实对象。

6.1 背景

我们都很熟悉积木和拼图。它们都是用很多相似的小块，搭建目标结构的方法。那为什么一个是幼儿的玩具，另一个是考验智力的工具？

这就涉及了面向对象（Object Oriented）的核心。怎样的组件才适合大量使用？从积木和拼图的类比，我们很容易看出，具备规则的平面界面，是轻松使用积木块的关键。而不规则的凹凸不平的曲面界面，立刻让如何拼装两个组件成为智力测验。

面向对象是一种编程方法，而不是语言特性。面向对象是 Alan Kay 在 1967 年提出的。它在 20 世纪 70 年代对 Smalltalk 语言进行了大量的实验，并对后来在计算机领域得到广泛应用的 C++、C#、Java、Objective-C 都产生了重大影响。以至于一整代的相关专业的毕业生，都自然而然地把所学的某种语言对面向对象的某种支持，固化为唯一可行的设计方法，套用在所有实际应用上。

实际上，面向对象的设计方法，就像更早出现、影响更深远的结构式编程一样，是和语言无关的。和 Smalltalk 同期出现的 C 语言，只是结构式编程语言。而它自身的灵活性，不仅可以应用面向对象的设计方法，更是 C++和 Objective-C 的核心。C++靠编译器静态地把面向对象的类型系统加在 C 语言的核心上，而 Objective-C 则是在运行时动态地判断目标类型及其支持的方法，其核心仍是单纯的结构化编程的 C 语言。

Go 同时支持静态类型编译和动态类型判断与方法调度。可是，Go 不想给自己贴上面向对象的标签。因为经过 30 多年的实践，务实的工程师和科学家发现，主流面向对象的语言在处理类型之间的关系方面搞砸了。这些语言的类和继承，涉及不同类型之间的各种复杂关系，这些关系在 Go 里常常可以自动得出。而对于数据的封装和保护，又顾此失彼。

一方面刻意强调安全第一，另一方面又后门大开。

例如 C++的 friend 和 private 是成员存取权限限制符。C++明确规定，private 只能由类成员自己使用，而 friend 可以供子类继承后使用，public 则是完全开放的。也就是说，C++的 private 对 friend 是门户洞开的。

但实际使用时，架构师要耗费大量时间，研究讨论如何摆平三者的关系。虽然几乎不可能应对未来的变化，也几乎对现实的程序没有任何帮助，但很多人却乐此不彼。因为务虚无需对现状负责。大可以用理论批判现实，再画些框框许愿，把自己搞不懂的东西变成一种没人能真正理解的指示，让别人去落实。这也许不是人的问题，而是人局限在某种语境中不得不为。

一些真正有洞察力的先辈们，例如 Hoare，早已经指出并不断告诫我们。但令人失望并且无奈的是，那些真正有决策力的先生们，害怕有无知顽童嚷嚷说他没有穿衣，所以就一直紧抱着所有的衣服，试图遮盖自己光着的身体。

Hoare 在《皇帝的旧衣》一文中指出：

> “程序员一直被复杂性包围；我们无从逃避。应用的复杂性源于我们的野心，我们总是希望计算机能执行更精密的计算。程序的复杂性是因为每个编程项目都有大量矛盾的目的。如果基本的工具——我们自己设计的、编写我们自己代码的语言——也被复杂化，那语言自身就成了问题、而不是解决方案的一部分。”

Go 的作者这样回答 Go 是不是面向对象语言这个问题：

> “是也不是。尽管 Go 有类型和方法，也许可以用一种面向对象风格来编程。但 Go 没有类型阶级。Go 界面的概念提供了一种不同的方式，我们相信它更容易使用并且在某些方面更通

用。并且Go通过在类型里内置类型，提供与子类类似但不尽相同的东西。还有，Go的方法比C++和Java的方法更通用：它们可以定义于任意类型的数据，甚至是像普通未装盒的内建整数类型。它们不仅仅局限于结构体类型。”

6.2　术语

历史上面向对象有自己的一套术语（terminology），例如 model、message、method。模特（model）通常又分为类（class）、对象（object）、演员（actor）等，可以理解为是 Go 的类型。而消息（message）是在 model 之间传递的信息，类似于参数和返回值。方法就是一个 model 接收到 message 后的处理方式，就是函数调用。

不同面向对象的语言使用不同的术语：方法在 C++里称为成员函数（member function），实例（instance）在 C++里称为数据成员（data member）。

本来，面向对象指的是每个对象模特都是独立的个体，而整个程序，甚至整个计算机软件系统和互联系统，都是由这些独立且活跃的对象互相发消息，来一起维系的一种运行状态。不同的模特对同样的消息可能有不同的解读方法。例如，收到“今晚有活动”，每个模特根据发信者的地位以及自身的日程与想象，可以有完全不同的回应。这种迟后绑定属于动态类型编程的范畴，我们已经讲过。现在需要进一步了解的是面向对象编程的本质，可能和你所了解的类、继承和多态等“面向对象”的术语属于不同层次的东西，某些语言的这些规定，不是面向对象的设计方法所必须采用的。

回到起点，听听 Alan Kay 自己对面向对象编程的理解：

> “OOP 对我只代表消息、局部持有和保护、状态处理的隐藏，以及对所有事物尽可能的迟后绑定。”

看看 Go 是怎样实现 Alan Kay 最初的理想的。注意 Alan Kay 并没有说类、实例、封装、继承、多态、重载等术语，尽管他发表上述看法时的 2003 年正是 OO 大行其道之时，是那些术语被推崇得天花乱坠的年

代。我们可以相信，OO 的发明人和实践者对其概念有着深刻的理解，而后人在通过某种语言实现 OO 时所采用的方式与术语，并不是其发明人的本意。

例如在 smalltalk 和 Objective-C 中的语句 `Object Reset:0`。可以理解为：把数值 0 作为消息发送给对象 `Object`，触发对象 `Object` 的 `Reset` 方法，接收消息 0，并进行相应处理。这在 Go 语言里，可以对应到 `Object` 类型的 `Reset` 方法、使用 0 作为参数的函数调用。

如果是 Go 的静态编程，这和 `Object.Reset(0)` 的方法调用没有任何区别。如果 `Object` 是界面变量，那么 `Reset` 方法是在执行时根据当时 `Object` 具体数值的类型，动态选择的。这也是 OO 消息的发送方式。而如果 `Object` 是程道类型，OO 的消息发送和接收，就直接对应到了程道的发送和接收操作。

静态编译的方法调用是最简单最高效的消息传递方式，而动态的消息调度则慢很多。例如，Objective-C 依靠 `objc_msgSend` 函数完成对象 `id` 和方法 `SEL` 的选择。尽管已经尽可能地优化，但绕过 `objc_msgSend` 而直接调用相应的 C 函数仍能成倍地提高速度。所以在遇到速度瓶颈时，还是要想办法绕开 Objective-C 的消息调度机制，直接使用核心的 C 函数调用。而 Go 的方法就是函数，不需要再去分析消息调度的机制和绕道的捷径。

但消息调度有其方便的一面。对象名作为主语可以使用相同的谓语而不会产生歧义。在 GUI 图形界面中，OO 得到了最广泛和有效的利用。同一个消息，例如 `Update`，发送给所有受影响的对象，来更新它们自己的显示状态，`menu.Update()` 和 `window.Update()` 是不同的函数操作，但都叫 `Update`。如果使用 C 的语义，那就必须写为 `menu_update()` 和 `window_update()` 两个完全不同的名字，也无法自动通过指向某个对象的指针，调用其对应对象的 `Update` 方法。当然，这可以通过 C 的

函数指针解决。而 Go 的类型有自己的解决方法。不同的类型可以使用相同的方法名，通过界面变量，也可以自动调用不同类型对应的同名方法。后者已经不再是静态编译的函数调用，而是和消息调度等同的动态调用了。

其实，消息调度，可以看做是函数调用的主谓分离。不只是说主语对象的指针自动成为谓语函数的第一个隐含的参数，更是因为每个对象都有自己的虚拟函数表等机制，实现函数的动态选择。在 Go 里，这正是界面变量的工作。

6.3 与 C++对比

对于熟悉 C++的读者，我们先列出它和 Go 的语法在概念上的不同，再看面向对象设计方法的不同。

(1) Go 没有构造函数和析构函数。Go 必须明确地使用 new 或者某个类型的 New 方法来完成变量的初始化。使用 new 只能得到零值，而 New 方法可以完成赋值。

(2) Go 使用垃圾回收，不需要主动释放内存。

(3) Go 有指针类型但没有指针运算。

(4) Go 的数组是实体而不是引用变量。复制是逐位进行的。可以使用切片来避免不必要的复制。

(5) Go 的赋值和参数传递只有复制，没有引用传递。但 Go 的切片、映射和程道变量与 C++的引用类型用法相同。

(6) Go 的字符串是基本类型，其内容不可以改变。

(7) Go 的映射是基本类型，对应 C++的散列表。

(8) Go 没有头文件。

(9) Go 不允许隐含的类型转换。

(10) Go 不支持函数重载和自定义操作符。

(11) Go 没有 const 和 volatile 修饰符。

(12) Go 使用 nil 作为无效指针值，C++使用 NULL 或者 0。

6.3.1 继承

Go 可以使用无名内置结构实现某种类似 C++子类的继承（inheritance）。例如：

```
package main

import "fmt"

type O struct {
    h int
}

func (a *O) Get() int {
    return a.h
}
func (a *O) Set(h int) {
    a.h = h
}

type OO struct {
    O
    i int
}

func (a *OO) Get() int {
    a.i++
    return a.O.Get()
}

func main() {
    oo := new(OO)
    oo.Set(42)
    fmt.Println(oo, oo.Get())
}
```

这里 OO 的 Set 方法实际上是从 O 继承来的，因为无名类型项的方法可以提升成为包括它的类型的方法。而这里 O 的 Get 方法被子类 OO 覆盖，但仍可以直接从 OO 的 Get 中使用项名得到。

这和 C++的类继承有些不同。无名类型项的方法不是虚拟函数。如果我们需要虚拟函数的动态特性，就要使用 Go 的界面类型。

6.3.2 抽象类

Go 的界面类型类似于 C++的纯抽象类（abstract class），也就是没有数据成员，而所有方法都是纯虚方法的类。甚至 C++的类、子类和模版类都可以对应到 Go 的界面类型。在 Go 里，具体类型提供界面类型声明的方法，可以当成是实现了一个界面，而不需要特意声明它们的继承关系。界面的实现和界面本身是完全分离的。

例如，如果我们不想直接操作一个整数变量，而希望通过取值函数和赋值函数来操作。我们可以：

```
type Int int

func (i *Int) get()int {
    return int(*i)
}
func (i *Int) set(t int) {
    *i = Int(t)
}
```

这样，如果使用 `var v Int`，我们就可以用 `v.set(10)`和 `v.get()`间接存取 v 的值。

我们可以声明一个界面：

```
type GetSetter intreface {
    get() int
    set(int)
}
```

这个界面可以在完成上述 `Int` 类型声明后很多年才给出，但并不影响 `Int` 实现这个界面，也不需要我们去改动 `Int` 所在的源程序。界面声明和实现是独立的。

在 C++里我们需要预先声明两个类型的关系。而 Go 里的类型自动满足一个界面——只要该界面声明了这个类型方法集合的一个子集。Go 采

用的这个方案相比较 C++的多重继承，有明显的优势：一个类型可以自动满足很多界面。界面可以只有一个方法，甚至支持没有方法的空界面。这使 Go 的界面比 C++的继承轻盈很多。界面设计更便利之处是，如果改动继承关系或者添加新关系，我们不需像 C++那样回头改动类的声明，而只需加个界面就解决问题了。换句话说，就是 Go 不需明确指出类型和界面的关系，所以也就不需要像 C++那样去浪费时间管理类型等级了。

可以使用 GetSetter 界面作为参数，来接受 Int 类型的变量传递。例如：

```
func getAndSet(x GetSetter){}
func f(){
    p := new(Int)
    getAndSet(p)
}
```

在 C++里，可以把 GetSetter 作为纯抽象类，然后改动 Int 的类声明让它继承这个纯抽象类，这样 getAndSet 函数的参数，才可以接受 Int。Go 里这种继承关系是自动推导得到的。

而界面类型也支持动态的转换，类似 C++的 dynamic_cast。例如：

```
type Printer interface() {
    print()
}

func g(i GetSetter){
    i.(Printer).print()
}
```

只要传递给 g 函数参数的具体类型确实实现了 print 方法，就可以通过类型断言来动态地转换参数 i 的类型——从 GetSetter 界面转换为 Printer 界面类型，而不像 C++的 dynamic_cast 那样，必须去声明这两个类型之间的关系。这并不是说在 Go 里面它们没有关系，而是这种关系是自动推导得到的，不需要靠某种形式的声明来指出这种关系。

Go 也不支持 C++的方法重载和运算符重载。这样使 Go 的方法调度不需要去匹配类型以便从同名方法中做出选择。Go 只针对方法名做调度匹配，并且编译检查参数和返回值类型必须一致。这样不仅提高了系统的安全性，也减少了程序员可能的困扰。并且，Go 作者的实践经验说明，同名方法重载和运算符重载这种看似实用的东西，其实极少能真正发挥作用，它们的存在只是使语言看起来很完善。但这种花边设计，正是 Go 语言所尽力避免的。

6.3.3 泛型

Go 的空界面 interface{}可以用来为任意类型的变量赋值。这样就可以实现 C++的泛型模版。而 Go 的切片和 append 函数，则可以用来做容器。如果是界面类型，则会在运行时完成类型检查，当然，比起 C++的泛型，这需要多一层的函数调用。例如：

```
type Any interface{}

type Iter interface {
    get() Any
    set(Any)
    next()
    equal(arg *Iter) bool
}
```

6.4 小结

本章反思和对比多于对新概念的介绍，主要针对的是受主流面向对象语言影响很深的读者，希望能够澄清一些误解，并对Go语言的简单实用多一分欣赏和理解，进而能逐渐摆脱固有观念的束缚，做出更接近Go风格的程序。

第 7 章

并发编程

Mathematicians stand on each others' shoulders and computer scientists stand on each others' toes.
（数学家站在彼此肩上，计算机科学家踩在彼此脚尖上。）
——Richard Hamming

Go 语言是 Rob、Robert 和 Ken 一起构思的。Ken 是 C 语言之父，因此 Go 理所当然延用了 C 扎实的静态编译系统；Robert 是 Smalltalk 和 Java 大师，Go 就顺理成章地具备了轻盈灵活的动态身段；Rob 是这三人中唯一经常出头露面的 Go 的代言人，自封队长（Commander），他贡献的是 Go 语言存在的意义——Go 的灵魂——并发。

7.1 背景

Go 语言内建的并发支持，可能是其最引人入胜之处。毕竟，只是比别的编程语言好用，并不能立即吸引已经掌握一种语言的程序员。就像只强调某个牌子的洗衣机省水省电，并不会让人们马上换掉家里还能用的那台一样。但如果称这个新发明有 4 个滚筒能同时将内衣、外衣、深浅颜色的衣服分开洗，保证卫生也不会染色。如果价钱合适，很多人都有更换的冲动甚至需要。

但并发（concurrent）并不容易。即便是 Go 语言给自己贴的标签上有并发这个词，我们仍然要很清楚，如何在执行速度、安全性和程序的清晰性之间做到平衡。Go 语言提供的去程和程道类型，以及 sync 包提供的更基本的各种互斥（mutex），是我们架构并发程序的组件。本章会从不同角度介绍和比较它们。最终的目的是使你熟悉每一个并发组件的功能、用法和优缺点，从而在自己的应用中能妥善做到速度、安全性和清晰性三者均衡。

并发不是并行。并行（parallel）是简单的大量重复，例如谷歌使用 MapReduce 算法，把同一个查询发到几千台服务器并行检索；再如 GPU 图形处理器把一个图像的浮点计算交给十几个 FPU 并行处理，这些并行和并发没有任何关系。

并发的程序，可以是顺序执行的，而且只有在 SMP 对称多核处理器上，才是真正的同时运行。SMP 已经是不可回避的现实了。但现有的编程语言仍对 SMP 缺乏必要的支持。因此，Go 语言从构思开始，就围绕着如何能在编程语言的层级，对并发的程序设计提供一个安全高效的引擎，让程序员能专注于分解问题与组合方案，而不需拘泥于线程管理与信号互斥这些麻烦又危险的操作。

Go 语言并发体系的理论是 C.A.R Hoare 在 1978 年提出的 CSP（Communicating Sequential Process，通信顺序进程）。尽管 CSP 有精确的数学模型，并实际应用在 Hoare 参与设计的 T9000 通算机（Transputer）上。但似乎 Rob Pike 更加注意通信顺序进程在通用计算机语言设计上的潜力，Newsqueak、Alef、Limbo 到现在的 Go 语言，Rob 已经对 CSP 研究了 20 多年。

CSP，也就是 Go 的并发概念，其核心只有简单的 4 个字：同步通信。去程和程道的概念，也就是 Go 语言的关键字 go 和 chan，都是建立在同步通信的基础之上的。

7.2 同步通信

可能没有几个人了解钟表的工作原理了。钟表工作时，它的所有齿轮转动时是环环相扣的，这都由同一个结构——擒纵器控制。我们听到的每一声滴答，是擒纵器的一次摆动。同一个摆动，同步着钟表的所有运动。

计算机也是同样的原理。同一个晶振（Crystal Oscillator），每秒振荡几百万次。它产生的每一个脉冲，同步着 CPU 的每一个 CMOS，也同步着内存的每一个 DRAM，也就是同步着每一条指令的读出、执行和内存数据的存取。

一个程序的指令，一条条地顺序读出，顺序地执行，顺序地更新内存，一切都同步到同一个时钟，按部就班。

那并行在哪里？

并行并不在计算机硬件里。它是软件设计上的东西，属于软件灵魂的范畴。直接和计算机硬件打交道的操作系统，大多是分时的设计。也就是把连续的指令流，分成一片片指令。硬件提供一个定时器，时间一到，就告诉操作系统该换一片指令流了。因为每秒可以这样切换很多次，不同的程序感觉是在同时运行，这就是多任务并行。

多任务切换的单位是程序。不同的程序占用完全不同的内存，而且彼此隔离。因为没有共享内存，所以程序员不需担心共享带来的问题。每个程序仍是从头到尾地顺序执行，顺序设计的程序，完全不必知道自己会被分片切换。

尽管每个程序的内存不被其他程序共享，其他的硬件，比如键盘、鼠标、硬盘、显示器，这些设备一定是被不同程序共享的。所以它们必须由操作系统管理，而不能被某个程序独占。通常，操作系统会使用自

己的内存，叫做缓冲区，存放这些设备需要或者产生的数据。如果一个正在自己的时间片运行的程序需要用到这些设备，就会调用操作系统的一个函数，称为系统调用，并使用缓冲区，交换复制数据。

系统调用时，是一个程序主动要求操作系统切换自己，而时间片结束时，是程序被操作系统强行切换。无论怎样，操作系统切换程序时，必须把正在被一个程序使用的所有CPU寄存器、FPU寄存器等保存起来。再从所有等待运行的程序中，公平地挑选一个，把它上次被切换保存的寄存器重新装入，这样就可以恢复这个程序的执行了。这一过程称为调度。这些来回切换的程序叫进程（process）。

而等待设备操作完成的进程，是暂时不会被调度的。因为通常设备完成一次操作的时间，足够完成几十次百次的多任务切换了。因此使用系统调用的进程，通常处于阻塞（block）状态，要等到设备通知操作系统说自己的操作完成了，阻塞的进程才回到可以被切换的状态。

这种硬件设备能阻塞进程的概念，最早被Unix发现可以用来同步两个进程。Unix发明的管道（pipe）。在shell里用竖线“|”代表。例如，下面这行代码使用了两个进程：

```
find src/pkg | grep _test.go
```

程序find在Unix操作系统中用于查找一个目录下的所有文件，grep用来查找给定的字符串。这样，我们希望找到文件名包含“_test.go”的文件。因为我们使用管道“|”把这两个程序的标准输出和标准输入连接在一起，当find找到并准备输出一些文件名时，如果grep还没有准备好接收，则find被阻塞；同样，如果grep准备接收而没有数据输入时，grep也被阻塞。注意，这两个进程不可能同时阻塞，因为时间片是顺序调度的，它们不可能同时执行。这样，总是一个阻塞在等待另一个阻塞来激活它。而激活发生在通信时，也就是输入或者输出的时刻。这就是靠通信来同步两个进程。管道出现在Unix中的时间，早过CSP理论的提出。

Unix 的设计可分为：内核、工具、壳。内核（kernel）负责进程调度、内存管理和设备共享。工具（utility）指的是各种小程序，每个小程序可以单独使用，功能只有一个，那就是接收标准输入，并将结果发送到标准输出，出错信息写入标准错误输出。这样，所有工具使用同样的标准输入输出和 ASCII 数据格式，就可以在壳（shell）里，让 Unix 用户把不同的工具组合在一起，用一种更粗线条的脚本语言（shell 脚本），完成更复杂的工作，而不需再用 C 这种强大但烦琐危险的编程语言。

也就是说，对内核来讲，进程是多任务的调度单元。一个程序，例如一个服务器，可能需要像服务台的电话总机一样，尽快地把接收到的请求转到相应的处理程序，以便能尽快回来等候和处理下一个请求。

这个“尽快”使人们发现，进程请求独立内存是在浪费时间。也就是说，每次分配一块新内存给处理请求的进程，有些时候是不必要的。如果能够让新进程共享现有的内存，则并发处理还能再快一些。这就产生了线程（thread）的概念。例如 Linux 的线程是作为进程调度的。区别只是进程有独立内存，并有内存保护；而线程则是共享内存的，并且如果不去特意保护，多个线程访问同一块内存，很容易发生冲突。

例如，经典的存取款问题：变量 a 和 b 记录着两个用户的存款。如果把 1 元钱从 a 转到 b，只要 a--和 b++这么简单。但只有 a 有钱才能这样做，所以需要先判断：

```
if a > 0 {
    a--
    b++
}
```

问题就出在这个必须的判断上。如果转账服务器几乎同时接到两次转账请求，而 a 真的就只有 1 块钱，两个请求分别由两个线程处理，它们共享 a 和 b 这两个变量。一号线程执行完判断，a 有钱。刚好这时，时间片结束，内核切换到二号线程，同样判断 a 有钱——因为一号线程

还没来得及减掉那一块钱就被切换了。这样二号线程就把钱从 a 转到 b。等到一号被切换回来继续执行时，尽管 a 没钱了，但一号服务线程已经判断完毕，不会再次检查，它还是一样要执行 a--和 b++，这样就造成系统混乱。

这个例子只是用来说明内存共享会产生问题，而这个问题又不能简单地解决。多线程编程令人头痛，根源就是内存共享，而公平的分享，不是每次编程都能处理好的。

进程不共享内存，又如何解决上面的问题？一个简单的方案，就是自己不动手、只动口，让第三个进程执行上面的判断和减加操作。由于只有一个进程在使用变量 a 和 b，没有共享就没有冲突。但这个进程要懂得接收其他进程发送给它的转账请求并回复。而其他进程可以并发执行，只有当需要转账时，才从并发变成顺序执行。这就要靠发送接收通信来同步并发。

由于操作系统的进程不够快，而线程不直接支持同步通信，Go 语言认识到自己必须发明一个新的并发单元，称为去程。通常，多个去程运行在一个线程上，不靠操作系统切换，而是执行到一个通信操作阻塞时，才转移到下一个可执行的去程。但如果去程使用了操作系统调用了阻塞，它会分配到一个新的线程，以便让之前线程上的其他去程不被操作系统阻塞，而继续按照自己的方式并发执行。

7.3 去程

对于初学者，上面的背景知识不是必须知道的。在 Go 语言里，只要正确使用去程和程道，就能得到正确并发的程序，而完全不必理会底层的进程调度和线程互斥。

Go 语言的去程就是一个函数。go 一个函数，它就和其他函数并发执行了。也就是说，go 不是函数调用，尽管可以给函数的参数赋值，go 函数不会停下正在执行的函数，不会等待那个函数执行完毕返回后再继续执行。它们对 Go 的运行系统来讲，已经是两个去程，可以分别调度，并发执行了。

例如，我们看看电脑 1/1000 秒可以数出几只羊：

```
package main

import (
    "fmt"
    "time"
)

func sheep(i int) {
    for ; ; i += 2 {
        fmt.Println(i, "只羊")
    }
}
func main() {
    go sheep(1)
    go sheep(2)
    time.Sleep(time.Millisecond)
}
```

函数 sheep 是普通的没有返回值的函数。如果不用 go，而是用普通的函数调用，sheep(1)应该会数出所有 int 可以表示的奇数只羊，

PDG

而 sheep(2)和后面的语句根本不会执行。但 go 了之后，sheep(2)也能去数偶数只羊了。而这个也 go 了之后，下面的 time.Sleep 可以让 main 函数去睡上 1 毫秒，然后 main 函数执行结束。

至于输出刚好是顺序的，也就是说两个数羊去程是交替执行的，这并不重要，也不是可以保障的。我们只需要知道，去程是可以交替执行的，而我们必须编程让交替能够发生。例如，如果 sheep 不使用 fmt.Println，而什么都不做，只是默默地数，而且您的电脑只有一个核，那可以肯定，这个核会疯狂地只想着让这个去程一直数下去，而不管其他去程是否会被饿死。用术语来说，就是去程是非抢占式的，必须自己让出（yield）。

去程调度发生在操作系统调用时。例如 fmt.Println 最终会使用一个系统调用，使正在执行的去程让出 CPU 给其他去程。而更多时候，是发生在同步通信时。并且，通信是去程能够输出的最可靠的方式，也是去程能够让出的最可靠的方式。

7.4 程道

去程互相通信靠的是程道（channel）。Go 语言的程道是类型，程道类型的变量可以赋值，可以作为函数参数和函数返回值。但程道类型的运算只有三种：发送 send、接收 receive 和被选择 select。

程道类型的变量使用 chan 声明。例如：

```
var chin chan int
```

这条语句声明了变量 chin 是可以发送接收 int 类型的程道类型变量。发送和接收的操作符都是<-，就像等号是赋值操作符，可以读写变量一样。例如 chin<-0 把 0 发送给程道变量 chin，而<-chin 则从程道接收一个整数；i = <-chin 把从程道 chin 接收的整数赋值给变量 i；chin<-i 把变量 i 的值发送给程道 chin。

例如我们从外太空发送那里现在的时间，穿过时空隧道，到达我们这里，输出：

```
package main

import (
	"fmt"
	"time"
)

var wormhole chan time.Time

func deepspace() {
	wormhole <- time.Now()
}


func main() {
```

```
    wormhole = make(chan time.Time)
    go deepspace()
    fmt.Println(<-wormhole)

    //Output:
    //Wed Feb 29 22:58:42 +0800 SGT 2012
}
```

wormhole 程道类型的变量初始值是 nil，必须 make，才可以操作。因为程道的发送或者接收操作会阻塞去程，并导致去程交替，所以这两个操作必须是在不同的去程里执行，一个发送，另一个接收。否则运行时会出现死锁（deadlock）。

如果不希望去程立即阻塞在程道上，可以在 make 时定义一个程道的容量，例如 make(chan int, 42)。这样一来，程道连续发送或者接收 42 个整数才阻塞。更重要的是，如果去程因为其他原因发生切换，此程道可以像信箱一样操作：可以从多个去程不断地发送数据给这个程道，而不需等待其他去程接收取走这些数据，直到程道满才阻塞；而接收去程也可以不断地从这个程道按发送来的顺序，逐个的取走数据，直到程道空才阻塞。这种有缓冲的程道，可以实现异步（async）通信，类似事件队列。

当然，如果 make 时没有给定容量，也就是容量为 0，程道不能存放数据，发送去程立即阻塞，直到另一去程从这个程道接收数据，才把发送来的值复制到接收变量。而如果一个去程从程道接收数据，但发送数据还未到，则阻塞发送去程，直到有其他去程从程道发送，完成数据复制。此时，程道只是一个约会地点，先到的去程要等待，直到对方出现，才实现赋值——等于把赋值操作分到两个去程中完成。程道赋值实现了两个去程的同步和数据交换。

7.5　遍历与关闭

假设我们有两个去程：生产者和消费者。消费者是盲目的，只要还有得消费，就会一直进行下去；但生产所需的资源终有耗尽的一天。我们希望消费去程能从一个程道，即能得到消费品，也可以被告知别等了。这可以使用 range 和 close 来实现。

range 遍历从程道接收每一个值，直到 close 关闭程道。例如：

```
package main

import (
    "fmt"
    "time"
)

type hole chan time.Time

func deepspace(w hole, h int) {
    defer close(w)
    for ; h > 0; h-- {
        w <- time.Now()
        time.Sleep(time.Second)
    }
}

func consumer(w hole) {
    for msg := range w {
        fmt.Println("consumer", msg)
    }
}



func main() {
```

```
    w := make(hole)
    go deepspace(w, 8)
    go consumer(w)
    for {
        msg, ok := <-w
        if !ok {
            break
        }
        fmt.Println("main", msg)
    }
    fmt.Println("Done")
}
```

消费者 consumer 和外太空生产者 deepspace，还有观赏者 main 都共用一个程道 w。消费者用 range 得到生产者从洞里发送来的每个消息并显示，直到 range 结束。而这个结束，也是通过程道变量 w，从 deepspace 的 close 函数发出的。这样，close 提供了一个标准的边界条件，这样就不需要为程道变量赋予一个特殊值，来代表结束了。

main 则使用接收操作的另一种赋值方式，同时得知 close 消息。这里，如果 w 接受到一个零值，或者被 close，导致 w 返回零值，ok 变量为 false,从而可以正确结束程序的运行。注意,main 和 consumer 两个去程是并发的，所以是交替地从洞 w 得到 deepspace 发送的值。

7.6 MapReduce

熟悉 MapReduce（映射化简）的读者可能想知道，如何使用程道执行相似的操作。答案是：每个去程对应一个 Map 并发执行，结果写入缓冲程道，最后再用一个 Reduce 函数综合这个程道得到的每一个值。例如：

```
package main

import (
    "fmt"
    "math/rand"
)

type hole chan int

func deepspace(w hole) {
        w <- rand.Int()
}

func main() {
    n := 8
    w := make(hole, n)

    //Map
    for i := 0; i < n; i++ {
        go deepspace(w)
    }

    //Reduce
    t := 0
    for i := 0; i < n; i++ {
        t += <-w
    }
    fmt.Println("Total: ", t)
}
```

w 是有 n 个洞的缓冲程道。启动 n 个去程调用 deepspace。这就是 MapReduce 的 n 个 Map 操作。假设每个 deepspace 可以通过 w 发送回自己的人口数量，而且因为距离不同，这些发送的数据在 w 里排队的顺序也是不同的。但在 main 里，range 可以等待并保证接收到 n 个值，并把它们逐一加起来，完成 MapReduce 的 Reduce 动作。

7.7 select 语句

select 和 switch 语句类似,但 select 的每个分支代表的是通信操作。select 随机地选择一个没有阻塞的程道。

select 语句的每个 case 分支，只能是发送和接收语句。程道表达式和出现在发送语句右侧的表达式一起，从上至下求值。如果有多个分支的通信操作可以继续，select 会随机选择其中的一个，执行其后的语句。否则，执行 default 分支；如果没有 default 分支，select 语句会阻塞，直到其中的一个发送和接收操作可以完成。

程道变量的值可以是 nil，这相当于此分支没有出现在 select 语句中；但如果是发送到 nil 程道，依旧要对对应的表达式求值。

尽管 select 语句会阻塞，在进入 select 时，会对程道和发送表达式求值一次，此求值所产生的副作用，会影响整个 select 语句的通信操作。

接收语句可以使用变量短声明，声明 1～2 个新的变量。

例如,我们要在指定的时间内从外太空接收最多 n 个数值,可以这样:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type hole chan int

func deepspace(w hole, h int) {
    defer close(w)
    d := time.Duration(rand.Intn(h)) * time.Second
```

PDG

```
    for ; h > 0; h-- {
        w <- rand.Int()
        time.Sleep(d)
    }
}

func main() {
    n := 8
    w := make(hole)
    t := 0
    maxTime := time.Second

    go deepspace(w, n)
Out:
    for i := 0; i < n; i++ {
        select {
        case n := <-w:
            t += n
        case <-time.After(maxTime):
            fmt.Println("Time out")
            break Out
        }
    }
    fmt.Println("Total: ", t)
}
```

再例如 Go 语言规范里的例子：

```
var c, c1, c2, c3 chan int
var i1, i2 int
select {
case i1 = <-c1:
    print("received ", i1, " from c1\n")
case c2 <- i2:
    print("sent ", i2, " to c2\n")
case i3, ok := (<-c3):  // 等价于: i3, ok := <-c3
    if ok {
        print("received ", i3, " from c3\n")
    } else {
```

```
            print("c3 is closed\n")
        }
    default:
        print("no communication\n")
    }
    for {  // 发送随机序列01到c
        select {
        case c <- 0:
        // 注意，没有语句、没有fallthrough和重叠分支
        case c <- 1:
        }
    }
    select {}  // 永远阻塞
```

7.8 程道值

程道类型的变量可以被传递。这样，可以理解为此程道是局部的私有程道，只有发送和接收的两个去程才知道它的存在。例如，我们可以通过一个程道 a 发送另一个程道变量 b，让接收去程使用 b 而不再使用 a 来和发送去程通信。

我们仍旧使用外太空通信的例子：

```
package main
import (
    "fmt"
)

type Read struct {
    key   string
    reply chan<- string
}
type Write struct {
    key string
    val string
}

var hole = make(chan interface{})

func deepspace() {
    m := map[string]string{}
    for {
        switch r := (<-hole).(type) {
        case Read:
            r.reply <- m[r.key] + " from Mars."
        case Write:
            m[r.key] = r.val
        }
    }
```

```
}

func main() {
    go deepspace()

    hole <- Write{"Name", "Martin"}
    home := make(chan string)
    hole <- Read{"Name", home}
    fmt.Println(<-home)
}
```

这个例子稍微复杂一些，说明了程道类型和其他类型一样，也可以作为界面变量的具体变量，并做类型转换。也就是说，并发和动态编程是独立的两个概念，可以交织在一起使用。

7.9 互斥

去程是并发的。和所有并发执行的进程、线程一样，共享数据会产生冲突。通过程道发送接收避免共享，并不是万灵药。例如程道带来的数据复制，以及去程切换产生的效率损耗，使得传统的互斥，仍然适用于简单或者需要高性能的场合。

尽管 Go 的运行系统通过 CPU 的 `tas` 指令，提供一切互斥操作所必需的最基本的 `TestAndSet` 测试和设置，但 Go 语言的同步包 `sync` 另外提供了各种 `Mutex`、`Cond` 和 `WaitGroup`，能帮助我们更方便地使用这个底层的内存共享机制。具体请参考 `godoc sync`。此时我们只是通过一个简单的例子，比较程道和基本的测试和设置的用法。

例如，我们第 1 章的游乐场示例这样产生序列号：

```
var uniq = make(chan int)

func init() {
    go func() {
        for i := 0; ; i++ {
            uniq <- i
        }
    }()
}
```

它通过唯一的一个无名去程不停地向程道 `uniq` 发送下一个序列号，避免了内存的共享。

可是操作这样小小的 8 字节内存，却动用一个去程和一个程道，导致操作系统进程阻塞和切换，代价未免太高。而且这个唯一的去程程道，在大量并发请求面前，会立刻成为瓶颈，需要动用更加复杂和更加不可靠的方式，例如负载均衡（load balance）。

其实最简单的方式应该是直接使用底层的测试和设置操作。例如：

```
var uniq = uint64(0)
uid := atomic.AddUint64(&uniq, 1)
```

也就是说，回避内存共享并不总是最有效的解决方案。该共享时就共享。只要程序员明智地使用共享工具，清楚地知道必需的保护手段，就不会导致共享冲突。

同样，我们可以用同步包 Mutex 提供的加锁 Lock 和开锁 Unlock 方法，共享任意大小的内存，而不只是测试和设置操作要求的基础内存。例如，结合 defer，我们可以安全地加锁，独占一片需要共享的内存，并在函数离开时自动开锁，释放对此内存的独占：

```
func g(){
    m.Lock()
    defer m.Unlock()
    //...
}
```

本节的内容，受到谷歌并发编程专家 Dmitry Vyukov 的启发，在此表示感谢。他对读者的建议是：

“90%的高层 CSP + 10%底层内存共享”。

也就是说，CSP 方式的程道通信所带来的安全性保证，仍是我们并发编程时的首要考量。而同步包提供的互斥机制，只是用于优化。Go 的并发编程的宣传口号在绝大部分情况下，依旧是并发设计的指导原则：

“只靠通信共享内存；别靠共享内存通信。”

7.10 小结

行百里者半九十。并发编程是很难掌握的。虽然 Go 语言已经使用 CSP 设计简化了所需的步骤，并同时降低了风险，但我们再次提醒读者，世上没有救世主，没有万灵药。我们所需的只能是不断的学习、探索、实践。

第 8 章

云 计 算

Was I deceived, or did a sable cloud
Turn forth her silver lining on the night?
（是我看花眼，还是那朵乌黑的云，
真的在夜里，露出她衬里的银边？）
——John Milton

本章我们介绍云计算的基础知识。重点分析 Go 语言在 App Engine 上开发应用。Go 语言的普及过程中，App Engine 提供了强大的推动力；而 App Engine 也靠着 Go 语言的支撑，获得最佳的性能。

8.1 背景

如果您无暇顾及计算机技术的最新发展，看到云计算，会不会觉得把云和计算扯在一起很奇怪？确实如此。云计算的出现，有偶然也是必然。2006年，亚马逊发现自己在因特网泡沫时大量投资的服务器，竟然只有一成的利用率，就推出产品 AWS，让它的客户有办法通过标准的 HTTP 协议，租用共享这些设备提供的服务。谷歌也不甘落后，在 2008 年发布 App Engine，并引发了业界的狂热。

云的由来，传统的说法是其因特网模型在纸上画不出来，就随便圈几笔代表抽象的远端共享的意思。据说这也继承了 20 世纪初老式电话系统 POTS 的传统。的确如此。电话发明以后，人类首次实现了不受时间和空间约束的信息的实时传播。它彻底改变了人们的通信方法和生活方式。也造就了美国电报电话以及各国电信这些企业的垄断局面。从这里我们也可以引申出云计算的特征。

(1) 信息高度集中在运营商的数据中心。

(2) 用户无法也无需关心自己信息的保存。

(3) 使用唯一的通信标准保障互联互通。

(4) 终端廉价，随手可得，容易使用。

(5) 按信息，以及软硬件的使用收费。

对普通使用者而言，云计算就是浏览网页。只是除了被动地获取影音文字，也可以主动地提供信息和搜索信息。尽管背后的软件一如既往地处理这些信息，但使用方式的改变，让普通用户跳出了传统的购买、安装、培训、使用软件的模式。也就是说软件自身改变不大，而是对用户变得透明了。让他们使用习惯的浏览方式，去避免和目标无关的麻烦。

听起来非常棒。但对软件开发者的挑战在于，它是虚的东西，关乎

的是用户体验，而不像传统的窗口应用那样千人一面。这又回到了本书一贯的主题：程序之所以也叫“设计”，是因为它是大脑构思的产物，是设计师自身修为的体现。程序设计和用户体验的设计是一脉相通的东西。希望读者不要满足于掌握了某种语言的相关技术，而应该不懈地去追求程序设计本身的这种美。

回到技术主题。现有的云计算可分为三种，从低到高分别称为：IaaS、Paas 和 SaaS。也就是基础服务（Infrastructure as a Service）、平台服务（Platform as a Service）和软件服务（Software as a Service）。基础服务是亚马逊 AWS 的模式，只出租服务器；软件服务是 salesforce.com 和 Google Apps 这样的互联网软件寻租模式；平台服务是我们要详谈的谷歌 App Engine 的模式，也就是提供一个开发和运行环境，使程序员不必像基础服务那样事必躬亲，也不会像软件服务那样没有机会动手。

8.2 GAE

谷歌的 App Engine 简称 GAE，可以直接使用谷歌在全球的数据中心，用 Go 语言编写因特网应用程序。因为这些 App 使用的软硬件系统和谷歌自己网站的系统是一样的，所以无论从安全性、性能、可靠性和可扩展性等方面，都比自行架设的网站服务器要胜一筹。无论是个人主页这种极小规格的应用，还是几十万人的跨国公司的内部作业系统，GAE 提供的云计算平台服务都可以胜认。特别是对从一粒种子能够迅速成长为参天大树的因特网起步公司，GAE 云平台极强的扩展能力，更是不容忽视。

GAE 的 Go 处理器是 Go 语言在谷歌内部运行一年后，对外发布的第一个大规模的应用项目。它也是目前 Go 语言最吸引程序员的项目之一。App 的创建、使用和维护都很简单：GAE 的开发者把 Go 写的代码传上去，申请一个 URL 指向它，就可以用浏览器运行那些代码提供的服务了。而同样使用浏览器指向 App 的管理网页，就可以监督此 App 的数据流量、存储空间和 CPU 使用情况。这些都是 App 的收费项目。对于不大的 App，GAE 是免费的；而对于大规模应用，GAE 便可为谷歌带来大笔收入。尽管收费了，但因为所有软硬件和服务都是全球共享的。除非一家公司能达到谷歌的规模，或者有这个梦想以及财力，不然，比起独立运作自己的环球因特网应用系统平台，共享还是会节省很多钱。这种分工合作的规模经济，是有别于小农经济的现代商业的核心，也是云计算一待成熟就一飞冲天的原因。

由于在 GAE 上，用 Go 编译的程序运行在谷歌的服务器上，所以此时 Go 的使用方式，与在自己管理的计算机上使用的 Go 标准包，有一些区别。具体情况如下。

(1) 需要使用 App Engine 提供的 Go SDK，而不是标准的 Go 下载包。

(2) App 使用的是类似 Go 的 `http` 包的界面；而 Go 的 App 类似于标准的 Go 网站服务器。

(3) App 运行时自动编译，不需直接使用 Go 编译器。

(4) App 运行在沙盒里，不可以使用某些需要特权的 Go 包。例如，文件和套接字（socket）都不可以直接使用。

(5) App 可以使用的资源受配额 `quota` 的限制。

总之，在 GAE 服务平台上使用 Go，很具吸引力也很容易。本章先简要分析 Go SDK 自带的几个演示程序，让读者尽快熟悉、尽快入门。接着详细介绍怎样用 Go 来操作 GAE API。

8.3 Hello 世界！

开始写云代码的第一步，当然是下载一个软件开发包。没有使用 Linux 或 Mac OS X 操作系统的程序员们，还要再等等。因为还没有 Windows 的下载可以用。

展开压缩包，进入目录，遵照传统方式执行第一个打招呼程序：

./dev_appserver.py demos/helloworld

如果一切 ok，它会说：到http://localhost:8080 看结果。那我们就要用浏览器了。接下来逐行审视一下这个打招呼程序：

```
package helloworld

import (
    "fmt"
    "net/http"
)

func init() {
    http.HandleFunc("/", handle)
}

func handle(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "<html><body>Hello, World! 세상아
안녕!</body></html>")
}
```

我们程序的入口是 `init`，这是因为 Go 的 `main` 函数是 GAE 提供的。`handle` 中的浏览器的访问请求由 `r` 指向，回复写入 `w` 缓冲区，在 `handle` 返回时，交由 GAE 传送给浏览器。

为了让 GAE 能正确找到我们的程序，在 helloworld 目录下要准备一个叫 app.yaml 的配置文件，其主要内容为：

```
runtime: go
```

它指定 GAE 运行的是 Go 处理器。

```
handlers:
- url: /.*
  script: _go_app
```

告诉 GAE 把/下所有路径都交由_go_app 处理。小心不要拼错，否则会出现一堆 Python 错误。

这个_go_app 位于 google/appengine/ext/go/目录下的__init__.py 里，它是架通_dev_appserver.py 和我们的 Go 程序的桥梁。当浏览器指向我们的 URL 时，_dev_appserver 会编译我们的 Go 程序，并存放在/tmp 下新创建的一个目录里，文件名就叫_go_app。这是个货真价实的可执行文件，所有用到的库和运行环境全部打包，一应俱全，不存在外部依赖。

/tmp 下还有两个文件用于通信，读者请自己看源代码。

8.4 画胡子

我们温习一下谷歌的 Andrew 和 Rob 在 Google IO 大会上演示的 moustach-io 的 http.go，以此了解一个没有用到 GAE 的标准 Go 网站程序，如何运行在 GAE 上。当然，我们还是用 dev_appserver 来冒充真正的云计算。

```
import (
	"bytes"
	"encoding/json"
	"errors"
	"fmt"
	"image"
	"image/jpeg"
	_ "image/png"
	"io"
	"net/http"
	"strconv"
	"text/template"

	"code.google.com/p/goauth2/oauth"
)
```

此例中的图像压缩使用 JPEG 的函数，所以导入 image/jpeg 包。image.Decode 图像解压也可以使用 PNG 格式。但 Go 不允许导入没用到其函数的包。由于此程序没有 png 的函数，所以用 _ 空白标识符来导入 image/png 包，目的是为了使用其 init 函数注册 decode 用的 PNG 格式。

goauth2 和 freetype（在 draw.go 中）是独立于 GAE 的第三方库，读者可以参考 README 用 hg 安装，以便能编译到_go_app 的可执行代码中。

```
    func init() {
        http.HandleFunc("/", errorHandler(upload))
        http.HandleFunc("/edit", errorHandler(edit))
        http.HandleFunc("/img", errorHandler(img))
        http.HandleFunc("/share", errorHandler(share))
        http.HandleFunc("/post", errorHandler(post))
        editTemplate = template.New(nil)
        editTemplate.SetDelims("{{{", "}}}")
        if err := editTemplate.ParseFiles("edit.html");
err != nil {
            panic("can't parse edit.html: " +
err.Error())
        }
    }
```

程序的入口注册 URL 路径的处理函数，editTemplate 的分隔符设定为 3 个{}。errorHandler 很有趣，我们一起看一下。

```
    func errorHandler(fn http.HandlerFunc)
http.HandlerFunc {
        return func(w http.ResponseWriter, r
*http.Request) {
            defer func() {
                if err, ok := recover().(error); ok {

        w.WriteHeader(http.StatusInternalServer Error)
                    errorTemplate.Execute(w, err)
                }
            }()
            fn(w, r)
        }
    }

    // check aborts the current execution if err is non-nil.
    func check(err error) {
        if err != nil {
            panic(err)
        }
    }
```

这里巧妙地使用了 Go 语言提供的 panic，defer 和 recover 异常处理方式。可以这样看：当 check 的 err 非空时，panic 导致 check 及其调用栈的所有函数（例如 upload 和 errorHandler）立即退出。但 errorHandler 定义了 defer，会在退出前执行。而其使用的 recover，会取得 panic 的变量，即 err。这里使用类型断言确定它是 error 类型，并在此处恢复 panic 引起的退出，然后用 errorTemplate 在浏览器中显示出错信息。如果没有 panic，或 panic 传入的不是 error 类型的值，则 defer 不执行任何操作。

注意到没有？errorHandler 接受一个函数，包装 defer 后，返回一个无名函数。Go 的函数可以像普通变量一样使用，等一下会看到。

```
type Image struct {
    Data []byte
}
```

定义 Image 结构体，其 Data 项是字节切片，用来方便安全地完成类似 C 的指针操作。

```
func upload(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
        // No upload; show the upload form.
        uploadTemplate.Execute(w, nil)
        return
    }
    f,_,err := r.FormFile("image")
    check(err)
    defer f.Close()
```

defer 是在函数退出时执行的。它紧跟在打开文件之后，以防忘记关闭。并且每个 err 都用 check 进行异常处理。

```
    var buf bytes.Buffer
    io.Copy(&buf, f)
    i,_,err := image.Decode(&buf)
    check(err)
```

bytes.Buffer 自动管理字节内存，io.Copy 可以高效的从文件 f 复制到 buf。image.Decode 能分析图像格式，并自动用 jpeg 或 png 包的 Decode 函数解压。

```
        const max = 1200
        if b := i.Bounds(); b.Dx() > max || b.Dy() > max {
```

if 语句可以有初始赋值子句。此处的变量 b 只在 if 后的块里有效，即其作用域由{}限定。

```
        if b.Dx() > 2*max || b.Dy() > 2*max {
            w, h := max, max
            if b.Dx() > b.Dy() {
                h = b.Dy() * h / b.Dx()
            } else {
                w = b.Dx() * w / b.Dy()
            }
            i = resize.Resample(i,i.Bounds(), w, h)
            b = i.Bounds()
        }
        w, h := max/2, max/2
        if b.Dx() > b.Dy() {
            h = b.Dy() * h / b.Dx()
        } else {
            w = b.Dx() * w / b.Dy()
        }
        i = resize.Resize(i, i.Bounds(), w, h)
    }
```

由于变量的局部作用域，变量名可以很短，而不用担心会和作用域外的同名变量起冲突。

```
        // Encode as a new JPEG image
        buf.Reset()
        err = jpeg.Encode(&buf, I, nil)
        check(err)
        // Create an App Engine Context for the client's request.
        c := appengine.NewContext(r)
```

```
        // Save the image under a unique key,a hash of the
image.
        key := datastore.NewKey("Image",
keyOf(buf.Bytes()), 0, nil)
        _,err = datastore.Put(c,key,&Image{buf.Bytes()})
        check(err)
        //Redirect to/edit using the key.
        http.Redirect(w,r,"/edit?id="+key.StringID(),
http.Status Found)
    }
```

用 r 的 FromFile 通过浏览器下载上传的图像，解压检查，重新采样改变大小，再压缩回 JPEG 格式，放入 App Engine 的 datastore，再让浏览器重定向到 edit 页面，添加大胡子。

```
    func edit(w http.ResponseWriter,r *http.Request) {
        editTemplate.Execute(w,r.FormValue("id"))
    }
```

在 init()中指定处理 URL /edit 的请求，简单地执行对应的模版。

```
    func keyOf(data []byte) string {
        sha := sha1.New()
        sha.Write(data)
        return fmt.Sprintf("%x",string(sha.Sum())[0:8])
    }
```

由 sha1 得到对应数据的唯一指纹，注意看其 API 是多么地简单，一写一读即告完成。

```
    func img(w http.ResponseWriter, r *http.Request) {
        c := appengine.NewContext(r)
        key := datastore.NewKey("Image", r.FormValue("id"),
0,nil)
        im := new(Image)
        err := datastore.Get(c, key, im)
        check(err)
        m,_,err := image.Decode(bytes.NewBuffer(im.Data))
        check(err)
```

```
    get := func(n string) int { // helper closure
        i, _ := strconv.Atoi(r.FormValue(n))
        return i
    }
    x,y,s,d := get("x"),get("y"),get("s"),get("d")
```

这里用到了闭包。get 是只限此处使用的函数，其内部的 r 从外部包的函数传入（闭包），n 和 i 是正常的参数和返回值。

```
    if x > 0 { // only draw if coordinates provided
        m = moustache(m, x, y, s, d)
    }
```

moustache 在 draw.go 中定义，也使用 package moustachio，编译器会把它们一起编译。

```
    w.Header().Set("Content-type", "image/jpeg")
    jpeg.Encode(w, m, nil)
}
```

然后简单地把 jpeg 图像传回浏览器。

```
func share(w http.ResponseWriter, r *http.Request) {
    url := config(r.Host).AuthCodeURL(r.URL.RawQuery)
    http.Redirect(w, r, url, http.StatusFound)
}
```

8.5 留言录

这个 guestbook（留言录）App 真正用到了 GAE 的 API。在 Go 里，这些 API 是包装在 appengine 包的公共函数和常量、变量。我们从头看起：

```
package guestbook

import (
    "io"
    "net/http"
    "text/template"
    "time"

    "appengine"
    "appengine/datastore"
    "appengine/user"
)
```

GAE 的 appengine 包、datastore 包和 user 包是比较常用的 3 个包，分别负责提供 Go 运行的环境（context），管理数据库存取，管理访问用户的登录。

```
type Greeting struct {
    Author  string
    Content string
    Date    time.Time
}
```

这个在内存中的数据结构会用来读写数据库。

```
func serve404(w http.ResponseWriter) {
    w.WriteHeader(http.StatusNotFound)
    w.Header().Set("Content-Type", "text/plain;
charset =utf-8")
    io.WriteString(w, "Not Found")
```

```
    }
    func serveError(c appengine.Context, w http.Response-
Writer, err error) {
        w.WriteHeader(http.StatusInternalServerError)
        w.Header().Set("Content-Type", "text/plain;
charset=utf-8")
        io.WriteString(w, "Internal Server Error")
        c.Errorf("%v", err)
    }
```

serve404 和 serveError 这两个函数是出错时的信息处理函数。

```
    var mainPage = template.Must(template.New
("guestbook").Parse(
        `<html><body>
    {{range .}}
    {{with .Author}}<b>{{.|html}}</b>{{else}}An anonymous
person {{end}}
    on <em>{{.Date.Format "3:04pm, Mon 2 Jan"}}</em>
    wrote <blockquote>{{.Content|html}}</blockquote>
    {{end}}
    <form action="/sign" method="post">
    <div>
    <textarea name="content" rows="3" cols="60">
</textarea></div>
    <div><input type="submit" value="Sign Guestbook">
</div>
    </form></body></html>
    `))
```

主页除了逐一显示已有的每一条留言，也提供一个 HTML 表让用户提交新留言。

```
    func handleMainPage(w http.ResponseWriter, r
*http.Request) {
        if r.Method != "GET" || r.URL.Path != "/" {
            serve404(w)
            return
        }
        c := appengine.NewContext(r)
```

此函数处理主页请求。我们为每个新的 http 请求都分配一个新的上下文环境 c。之后我们用这个 c 来把 GAE 的服务对应到这个请求。

```
    q := datastore.NewQuery("Greeting").Order("-Date").
Limit(10)
```

可以理解为类似普通数据库的 SQL 查询，此处是谷歌分布式数据库所用的查询方式：查找 Greeting 对象，降序排列，限制最多返回 10 个结果。

```
        var gg []*Greeting
        _, err := q.GetAll(c, &gg)
        if err != nil {
            serveError(c, w, err)
            return
        }
```

执行查询，结果放入 gg 切片。注意，我们无需自己分配 gg。

```
        w.Header().Set("Content-Type", "text/html;
charset=utf-8")
        if err := mainPage.Execute(w, gg); err != nil {
            c.Errorf("%v", err)
        }
    }
```

显示主页，也就是在浏览器中显示查询结果，以及新留言的提交表。

```
    func handleSign(w http.ResponseWriter, r
*http.Request) {
        if r.Method != "POST" {
            serve404(w)
            return
        }
        c := appengine.NewContext(r)
        if err := r.ParseForm(); err != nil {
            serveError(c, w, err)
            return
        }
```

此处处理留言表提交请求。同样每个请求都使用一个新的 GAE 环境 c。

```
g := &Greeting{
    Content: r.FormValue("content"),
    Date:    time.Now(),
}
```

自动分配一个新的 Greating 结构，填入提交的内容。

```
if u := user.Current(c); u != nil {
    g.Author = u.String()
}
```

如果用户还未登录，Current 返回 nil。主页会要求用户登录。否则它返回一个用户结构。我们把它填入 Greeting 的作者一栏。

```
if _, err := datastore.Put(c, datastore.NewIncom-
pleteKey(c, "Greeting", nil), g); err != nil {
        serveError(c, w, err)
        return
}
```

无论用户有没有登录，我们都把这个新的 Greeting 结构写入数据库。NewIncompleteKey 是让数据库自动分配一个新的键字 key 给这条新留言。

```
http.Redirect(w, r, "/", http.StatusFound)
}
```

返回浏览器，让它重新访问 Redirect 主页。

```
func init() {
    http.HandleFunc("/", handleMainPage)
    http.HandleFunc("/sign", handleSign)
}
```

GAE 里 Go 使用 init 来定义 http 的 URL 路径处理函数。

8.6 用户 API

GoApp 可以使用三种方式鉴别用户：谷歌账户、OpenID 识别符，以及 Google Apps 的账户。App 可以检查现在的用户是否已经登录，如果还没有，则可以自动把用户重定向到所需的登录网页，甚至可以创建一个新的谷歌账户。

当用户登录进 App 后，App 就可以得到用户的电子邮件地址或者 OpenID 的识别符。App 也可以检查当前用户是否是管理者，可以据此很方便地实现 App 的管理功能。

对于你的网页，如果需要用户登录才可以访问，就需要在 app.yaml 中做好配置。此处也可以同时配置一个用户是否具备此 App 的管理权限。例如：

```
handlers:

- url: /profile/.*
  login: required

- url: /admin/.*
  login: admin
```

SDK 的开发服务器使用一个假定的登录页面模拟谷歌账户系统。当 App 调用用户 API 时，会提示输入电子邮件地址而不是密码。用户可以输入任意地址，而 App 会当做是用那个电子邮件地址登录的。同时，它也有一个选项来选择此次登录是否使用管理员权限。用户 API 可以返回一个注销 URL，取消此次登录。

PDG

8.7 数据库 API

App Engine 数据库是一种无模式（schemaless）的对象数据库。它具备如下特点。

(1) 不需要安排离线时间（no planned downtime）。

(2) 提供不可分割的交易（atomic transaction）。

(3) 高可读写性（high availability of reads and writes）。

(4) 所有非祖先查询最终一致（eventual consistency for all queries except ancestor queries）。

(5) 所有读和祖先查询强一致(strong consistency for reads and ancestor queries)。

这些，使 App 使用的数据库健壮而且可以扩展，并具备很好的读写性能。使用时，App 可以创建实体，而数据就是作为实体的属性存放的。App 可以查询实体，所有的查询都已经预先做好索引。这样，即便是非常大的数据量，查询速度依然很快。

云计算的数据库和通常意义上的关系数据库是不同的。它们面对的是不同级别的数据处理问题。所需解决的技术问题也有极大的差别。

App Engine 的数据库，幕后使用的是谷歌赫赫有名的 Bigtable。这要求读者从新的角度思考数据的分割整合，SQL 已不再适用。

SQL 的关系数据库，几乎都是主从服务器的结构：一个数据中心的服务器为主服务器，负责写入，并由此复制到其他数据中心的从服务器。因为在任何时刻都只有一个服务器执行写入，所有的查询肯定都是一致的。但如果主服务器离线或者网络出现问题，将造成数据读写服务的中断。这对于大部分的因特网应用来说，是不可接受的。

App Engine 中 Go 使用的数据库，会将数据会复制到很多数据中心，

复制时依靠 Paxos 算法协调。它提供了最可依靠的读写操作，无需担心服务器宕机或者网络的中断，但代价是比较长的写入时延，因为数据需要扩展到不同地理位置的数据中心。这样，绝大部分的查询，需要一段时间，结果最终才会一致。

因特网应用的数据库，也要求能同时处理几万甚至更多的读写请求，这单靠一台服务器是无法实现的。数据必须分散到许多服务器中。

8.7.1 术语

App Engine 数据库有自己的一套术语：

数据对象是实体（entity）。一个实体有一个到多个已命名的属性（property）。而每个属性都有一个或多个值。属性值可以属于不同类型，包括整型、浮点型、字符串型、时间型、二进制数据类型，等等。

和传统的 SQL 数据库不同，App Engine 数据库不需预先定义数据模式。也就是说，数据库存储的实体不需要相同的属性；每个属性也不需要相同的数据类型。这样，如果一个应用要求定义一个正式的数据模式，那么是程序而不是数据库，要维护并检查数据和模型的正确兼容。

App Engine 数据库可以把多个操作放入一个交易，这样，只有它包含的所有操作都成功了，交易才算完成；否则，随便哪个操作失败，都会导致交易取消，并且它执行的操作都会取消，也就是回到交易前的状态。这对于需要支持大量用户同时存取同一个数据的分布式因特网应用，非常关键。

8.7.2 Go数据库API

数据库实体的创建可以使用 Go 的结构体和映射。此时，结构体的成员或者映射的键，就成为实体的属性。我们只要准备好这样的结构体

或者映射值，再生成一个键，交给 datastore.Put()，就创建了一个实体。使用同一个键再次 Put，就可以更新此实体；使用 Get，给出此键，以及一个指向存放对应类型值的指针，就可以取回这个键对应的实体。例如：

```
package main

import (
    "appengine"
    "appengine/datastore"
    "appengine/user"
    "fmt"
    "io"
    "net/http"
    "time"
)

type Employee struct {
    Name     string
    Role     string
    HireDate time.Time
    Account  string
}

func serveError(c appengine.Context, w http.Response-
Writer, err error) {
    w.WriteHeader(http.StatusInternalServerError)
    w.Header().Set("Content-Type", "text/plain;
charset=utf-8")
    io.WriteString(w, "Internal Server Error")
    c.Errorf("%v", err)
}

func handle(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)

    e1 := Employee{
```

```
            Name:      "Arthur Dent",
            Role:      "Hero",
            HireDate: time.Now(),
            Account:  user.Current(c).String(),
        }

        key, err := datastore.Put(c,
datastore.NewIncompleteKey(c, "employee", nil), &e1)
        if err != nil {
            serveError(c, w, err)
            return
        }

        var e2 Employee
        if err = datastore.Get(c, key, &e2); err != nil {
            serveError(c, w, err)
            return
        }

       fmt.Fprintf(w, "Stored and retrieved the Employee:
%v", e2)
    }

    func init() {
        http.HandleFunc("/", handle)
    }
```

8.7.3 实体键

数据库的每个实体，都有唯一的标识符，称为键。键由三部分构成。

(1) 种类（kind），用来给数据库查询分类。

(2) 标识符（identifier），可以是一个字符串，或者是一个整型 ID。

(3) 一个可选的祖先路径（ancestor path），在数据库等级中定位实体。

例如，可以使用种类 `Employee` 来代表属于雇员的实体，而标识符可以是由整数表示的雇员员工编号。因为此标识符是实体键的一部分，

给定后就不可以改变了。

祖先的概念，仍需要多着墨儿笔。

App Engine数据库的实体，类似文件系统的文件和目录，是树状结构的。创建实体时，可以声明另一个实体是它的父（parent）实体，则此实体成为父实体的子（child）实体。这种父子关系在子实体创建时就固定下来，没有办法改变了。没有父实体的实体，不叫孤儿，而叫根（root）。数据库不可以给同一父实体的两个子实体或者两个根实体，同一个数字ID。

这样，一个实体的祖先就可以确定下来了。祖先路径就是它的父实体、它父实体的父实体，等等，直到根。同样，一个实体的子实体，它的子实体的子实体，等等，称为实体的后代。这样，实体在数据库中的完整路径，就是祖先路径加上自己，是一系列的种类-标识符对。例如：

```
Person:Grandpa / Person:Dad / Person:Me
```

而根实体的祖先路径是空的，例如 `Person:Grandpa`。

8.7.4 查询和索引

除了直接使用键来获取一个实体，App 还可以使用属性值来查询。查询操作针对的是某个种类的实体，可以指定实体属性的值和键的过滤器（filter），并限定返回0个或者多个实体。限定实体的返回个数可以节省内存并且提高性能。查询还可以对返回的结果，按照指定的顺序对属性值排序。

查询过滤和排序顺序中的每个属性名，都至少返回一个实体，此属性的值会同时满足所有的过滤条件。

而且，每个查询都会使用索引。索引是一张包括所有查询结果的有序表。一些类型的查询索引是自动生成的，并且 App 可以在 index.yaml 中定义额外的索引。因为数据库会在 App 更新实体时自动更新索引，所

以可以直接通过索引得到查询结果，而不需进一步计算。

SDK 的开发服务器会在查询用到 index.yaml 中未定义的索引时自动给出提示。我们可以在上传 App 到 App Engine 之前，据此优化索引。

8.7.5 实体组

单一的交易可以包括任意多个实体的创建、更新和删除操作。交易可以保证数据的一致性：只有交易中的全部操作成功，数据库才会更新这些数据；否则，数据库不会更新，所有交易中的操作都无效。

例如，如果给某个实体的计数器中的值加 1，需要先读出计数器的值，加 1，再写入数据库更新计数器。如果不用交易，两个进程就可能试图同时更新计数器导致冲突。如果这两个进程都读到同一个值，都加 1 后写回，两次写入实际上只在那个实体的计数器上加 1 而不是加 2。使用交易，就可以确保它们按顺序完整执行。

同一个祖先的多个实体可以放在同一个交易里，称这些实体属于同一个实体组（entity group）。Go 的数据库允许一个交易的实体属于一个实体组，或者属于多个实体组。我们在定义数据模式的时候，这些都要安排好。

App Engine 数据库的交易管理系统使用的是“乐观并发”（optimistic concurrency）。当多个 App 试图同时改动同一个实体组时，第一个提交的 App 会成功，其他 APP 的提交都会失败。App 会重新尝试提交。

8.7.6 限制

App Engine 数据库相对于 SQL 关系数据库，具有如下重要局限。

(1) App Engine 数据库可以扩展，App 流量增大时仍可保持高性能：写入时自动分散数据，读出时查询性能只和结果相关，而和数据量无关。也就是返回 100 个结果的查询，无论是从 100 个实体还是从 100 万个实

体得到，所需时间都是一样的。这也是一些 SQL 的查询无法在 App Engine 数据库中使用的原因。

(2) App Engine 数据库所有的查询都是通过预建的索引进行的，所以能够执行的查询种类是有限制的。SQL 的连接操作、多个属性的不相等过滤和过滤查询结果等都不受支持。

(3) App Engine 数据库的同一种类的实体不要求具备相同的属性，所以无法通过查询返回一个结果实体的属性子集。

8.8 交易

因为 App 的并发特性，数据的交易（transaction）对保证数据的一致就特别重要。交易代表的是一系列不可分割的整体操作：交易中的每一项都完整地执行，或者整个交易取消，所有操作都不对最终数据造成影响。Go 中，交易的执行靠的是下面的函数，它使用 App Engine 数据库交易机制处理参数中指定的函数：

```
datastore.RunInTransaction(c, f) os.Error
```

交易可能出现如下错误。

(1) 太多用户试图同时修改同一个实体组。

(2) App 用完限定资源。

(3) 数据库内部错误。

发生这些错误时，`RunInTransaction` 都会返回一个错误。

我们看看下面这个使用交易的计数器的例子：

```
package counter

import (
    "appengine"
    "appengine/datastore"
    "fmt"
    "net/http"
)

func init() {
    http.HandleFunc("/", handler)
}

type Counter struct {
    Count int
}
```

```
    func handler(w http.ResponseWriter, r *http.Request) {
        c := appengine.NewContext(r)

       key := datastore.NewKey(c, "Counter", "mycounter",
0, nil)
        count := new(Counter)
        err := datastore.RunInTransaction(c,
            func(c appengine.Context) error {
                err := datastore.Get(c, key, count)
                if err != nil && err != datastore.ErrNoSuch-
Entity {
                    return err
                }
                count.Count++
                _, err = datastore.Put(c, key, count)
                return err
            }, nil)
        if err != nil {
            c.Errorf("Transaction failed: %v", err)
            http.Error(w, "Internal Server Error", 500)
            return
        }

        fmt.Fprintf(w, "Current count: %d", count.Count)
    }
```

RunInTransaction 会使交易执行其参数的无名函数，如果此函数返回 nil，代表交易完成。RunInTransaction 会向数据库提交交易，并在成功后返回 nil。如果那个函数因为某个原因返回非 nil 的值，数据库没有改动，RunInTransaction 也返回同样的非 nil 值。

RunInTransaction 可以自动重试 3 次，直到交易成功，或者放弃。注意，RunInTransaction 使用的函数必须是自乘不变的（idempotent）。也就是必须能够多次执行仍得到同样的结果。

当然，此处只是示例。这样安排的交易很容易成为阻塞 App 的瓶颈。在必须支持大规模并发的时候，就需要散段（shard）。

8.9 散段

数据库的散段，是一种横向分割数据的技术。也就是除了纵向地把数据分为列（column）之外，还把多个行（row）分别存放到相互独立的数据库服务器上。这样做不仅减少了每个数据表中的行数，也减小了索引，从而对单个数据库服务器的性能有所提高；更重要的是，这些不同机器上的数据库可以真正地并行使用，这样一来，整体的查询性能可以成倍提高。而且，数据库可以存放于不同地点，安全性也是单个服务器无法比拟的。

散段可以很复杂。此处我们只用一个简单的例子，看看 Go 的 App Engine 应用是如何利用散段的。

```
package main

import (
    "appengine"
    "appengine/datastore"
    "fmt"
    "math/rand"
    "net/http"
)

type Shard struct {
    Count int
}

const (
    numShards = 20
    shardKind = "shard"
)

// Count retrieves the value of the counter.
```

```
func Count(c appengine.Context) (int, error) {
    total := 0
    t := datastore.NewQuery(shardKind).Run(c)
    for {
        s := new(Shard)
        _, err := t.Next(s)
        if err == datastore.Done {
            break
        }
        if err != nil {
            return total, err
        }
        total += s.Count
    }
    return total, nil
}

func Inc(c appengine.Context) error {
    return datastore.RunInTransaction(c,
        func(c appengine.Context) error {
            name := fmt.Sprintf("shard%d", rand.Intn
(numShards))
            key := datastore.NewKey(c, shardKind,
name, 0, nil)
            s := new(Shard)
            err := datastore.Get(c, key, s)
            // A missing entity and a present entity
will both work.
            if err != nil && err != datastore.ErrNoSuch-
Entity {
                return err
            }
            s.Count++
            _, err = datastore.Put(c, key, s)
            return err
        }, nil)
}
```

```
    func inchandler(w http.ResponseWriter, r
*http.Request) {
        Inc(appengine.NewContext(r))
        fmt.Fprintf(w, "<html>Counter updated. <a href=
http:// localhost:8080/count>Check now.</a></html>")
    }

    func counthandler(w http.ResponseWriter, r
*http.Request) {
        c, _ := Count(appengine.NewContext(r))
        fmt.Fprintf(w, "count: %d", c)
    }

    func init() {
        http.HandleFunc("/", inchandler)
        http.HandleFunc("/count", counthandler)
    }
```

此处的散段技术充分利用了 App Engine 的特性——查询极快。我们把一个计数器分成 N 段，然后随机地选择一个散段增 1。当需要知道计数器的值时，就把所有散段的值加在一起。并且散段越多，App 的流通量越大。这种散段技术在消除数据库性能瓶颈方面非常有用。并且，如果配合 memcache 提供的内存缓冲，则更为显著。

8.10 内存缓冲

高性能可扩展的网络应用通常会对数据库使用分布式的内存数据缓冲机制。同样，App Engine 也提供了一个内存缓冲。

内存缓冲（memcache）主要用来给数据库查询提速。如果同一个查询使用同样的参数，而且返回结果如果有更新也不需立即反映在网页上时，可以考虑使用内存缓冲。例如，用户偏好、会话数据，等等。此处，我们再次看看 SDK 中计数器演示的例子怎样使用内存缓冲：

```
    package counter

    import (
        "fmt"
        "io"
        "net/http"
        "strconv"
        "appengine"
        "appengine/memcache"
    )

    func serveError(c appengine.Context, w
http.ResponseWriter, err error) {
        w.WriteHeader(http.StatusInternalServerError)
        w.Header().Set("Content-Type", "text/plain;
charset=utf-8")
        io.WriteString(w, "Internal Server Error")
        c.Errorf("%v", err)
    }

    func handle(w http.ResponseWriter, r *http.Request) {
        c := appengine.NewContext(r)

        item, err := memcache.Get(c, r.URL.Path)
```

```
        if err != nil && err != memcache.ErrCacheMiss {
            serveError(c, w, err)
            return
        }
        n := 0
        if err == nil {
            n, err = strconv.Atoi(string(item.Value))
            if err != nil {
                serveError(c, w, err)
                return
            }
        }
        n++
        item = &memcache.Item{
            Key:   r.URL.Path,
            Value: []byte(strconv.Itoa(n)),
        }
        err = memcache.Set(c, item)
        if err != nil {
            serveError(c, w, err)
            return
        }

        w.Header().Set("Content-Type", "text/plain;
charset=utf-8")
        fmt.Fprintf(w, "%q has been visited %d times", r.
URL.Path, n)
    }

    func init() {
        http.HandleFunc("/", handle)
    }
```

内存缓冲最基本的操作是使用一个键来读写。如果读取时发生的错误是 ErrCacheMiss，表示这个键在缓存中不存在。其他的错误值，是真正的 App Engine 内部错误，需要单独处理。

8.11 大件库

Blob（Binary large object）表示二进制大件。App Engine 的数据库一个实体只能是大约 1 M 字节。可 App 又无法直接读写文件。此时可以使用 Blobstore 大件库 API，它能读写的数据，只受一个 HTTP 连接能够上传和下载的数据量的限制。例如音乐和影像，都可以使用大件存储提供服务。

一个大件是通过 POST 间接创建的。这可以是网页提交的表单（form）里的文件上传一栏。App 使用大件库 API 生成这个表的 action URL，这样，当浏览器提交表时，会自动把那个文件通过此 URL 发送到大件库。然后，App 会用得到的大件键（blob key），重定向到另一个 URL，继续处理表的内容。我们举例说明：

```
package main

import (
    "io"
    "net/http"
    "text/template"

    "appengine"
    "appengine/blobstore"
)

func serveError(c appengine.Context, w http.Response-
Writer, err error) {
    w.WriteHeader(http.StatusInternalServerError)
    w.Header().Set("Content-Type", "text/plain")
    io.WriteString(w, "Internal Server Error")
    c.Errorf("%v", err)
}
```

```
    var rootTemplate = template.Must(template.New("").
Parse(`
    <html><body>
    <form action="{@}" method="POST" enctype="multipart/
form-data">
    Upload File: <input type="file" name="file"><br>
    <input type="submit" name="submit" value="Submit">
    </form></body></html>
    `))

    func handleRoot(w http.ResponseWriter, r *http.
Request) {
        c := appengine.NewContext(r)
        uploadURL, err := blobstore.UploadURL(c, "/upload",
nil)
        if err != nil {
            serveError(c, w, err)
            return
        }
        w.Header().Set("Content-Type", "text/html")
        err = rootTemplate.Execute(w, uploadURL)
        if err != nil {
            c.Errorf("%v", err)
        }
    }

    func handleServe(w http.ResponseWriter, r *http.
Request) {
        blobstore.Send(w, appengine.BlobKey(r.FormValue
("blobKey")))
    }

    func handleUpload(w http.ResponseWriter, r *http.
Request) {
        c := appengine.NewContext(r)
        blobs, _, err := blobstore.ParseUpload(r)
        if err != nil {
            serveError(c, w, err)
            return
        }
```

```
        file := blobs["file"]
        if len(file) == 0 {
            c.Errorf("no file uploaded")
            http.Redirect(w, r, "/", http.StatusFound)
            return
        }
        http.Redirect(w, r, "/serve/?blobKey="+string
(file[0]. BlobKey), http.StatusFound)
    }

    func init() {
        http.HandleFunc("/", handleRoot)
        http.HandleFunc("/serve/", handleServe)
        http.HandleFunc("/upload", handleUpload)
    }
```

上传大件所需的步骤如下所示。

(1) `UploadURL` 得到一个 URL，放入 HTML 的表中作为 action。

(2) HTML 的表必须正确设置项，当用户提交表时，POST 请求由大件库 API 处理，从而创建一个大件。此 API 同时也会在数据库里生成此大件的信息记录，并重写请求使用给定路径作为大件键。

(3) 我们可以在重定向后的表处理函数中，把大件键和 App 的其他数据一起存入数据库。此时，大件已然存入。如果不想保留，App 应该立即删除此大件。

当大件库 API 重写用户请求时，上传文件的 MIME 体被清空，而头会添加大件键。其他表项和 MIME 部件都会保留，并一并提交给上传处理函数。

而下载大件就简单许多。只需在下载处理函数中使用 `Send`，发送大件键即可。例如：

```
    blobstore.Send(w, appengine.BlobKey(r.FormValue
("blobKey")))
```

8.12 URL 抓取

App 除了自己提供因特网服务之外，还可以依靠其他因特网网络应用或者 App 提供的服务。这只需使用 URL 抓取来发送一个 HTTP 或者 HTTPS 的请求，并接收回复。这个抓取服务也是利用了谷歌网络来保证效率和可扩展性。

最简单的方式是使用 urlfetch 包的 Client。它和 http 包的 Client 用法一样，只不过底层使用的是 urlfetch 包的 Transport 来使用 App Engine 发送请求。例如：

```
package main

import (
    "appengine"
    "appengine/urlfetch"
    "fmt"
    "net/http"
)
func handler(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    client := urlfetch.Client(c)
    resp, err := client.Get("http://www.google.com/")
    if err != nil {
        http.Error(w, err.Error(), http.Status-
InternalServerError)
        return
    }
    fmt.Fprintf(w, "HTTP GET returned status %v",
resp.Status)
}
func init() {
    http.HandleFunc("/", handler)
}
```

默认的抓取请求必需在 5 秒之内得到回复，但可以设置这个参数，HTTP 最长为 60 秒；任务队列最长可以是 10 分钟。

8.13 任务队列

如果一个 App 需要长时间处理一件工作，可以把它分成一些较小的独立单元——称为任务，放入后台执行。这依靠的是任务队列 API。通过把这些任务插入一个或者多个队列，App Engine 可以发现并且在系统资源允许的时候，执行这些任务。

此时，需要一个新的配置文件：queue.yaml。如果 App 没有此文件，则它只有一个队列，称为 default。

通过设置 Task 的值，我们可以创建一个任务，然后用 Add 函数加入一个队列就可以了。例如：

```
package counter

import (
    "appengine"
    "appengine/datastore"
    "appengine/taskqueue"
    "net/http"
    "text/template"
)

func init() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/worker", worker)
}

type Counter struct {
    Name  string
    Count int
}

func handler(w http.ResponseWriter, r *http.Request) {
```

```
        c := appengine.NewContext(r)
        if name := r.FormValue("name"); name != "" {
            t := taskqueue.NewPOSTTask("/worker", map
[string][] string{"name": {name}})
            if _, err := taskqueue.Add(c, t, ""); err !=
nil {
                http.Error(w, err.Error(), http.Status-
Internal ServerError)
                return
            }
        }
        q := datastore.NewQuery("Counter")
        var counters []Counter
        if _, err := q.GetAll(c, &counters); err != nil {
            http.Error(w, err.Error(), http.StatusInter-
nalServer Error)
            return
        }
        if err := handlerTemplate.Execute(w, counters);
err != nil {
            http.Error(w, err.Error(), http.Status-
InternalServer Error)
            return
        }
    }

    func worker(w http.ResponseWriter, r *http.Request) {
        c := appengine.NewContext(r)
        name := r.FormValue("name")
        key := datastore.NewKey(c, "Counter", name, 0, nil)
        var counter Counter
        if err := datastore.Get(c, key, &counter); err ==
datastore. ErrNoSuchEntity {
            counter.Name = name
        } else if err != nil {
            c.Errorf("%v", err)
            return
        }
```

```
        counter.Count++
        if _, err := datastore.Put(c, key, &counter);
err != nil {
            c.Errorf("%v", err)
        }
    }

    var handlerTemplate = template.Must(template.New("")
.Parse(
    `
    {{range .}}
    <p>{{.Name}}: {{.Count}}</p>
    {{end}}
    <p>Start a new counter:</p>
    <form action="/" method="POST">
    <input type="text" name="name">
    <input type="submit" value="Add">
    </form>
    `))
```

此 App 定义的用户请求处理函数 handler，对应 Get 请求，显示计数器当前值，而对应 POST 请求，则把一个从 NewPOSTTask 返回的 Task 加入队列。此任务映射到 URL /worker 上。而 App 定义的 worker 处理函数，则在后台用来给数据库的计数器增 1。

8.13.1 任务

任务（task）包括两部分。

(1) 一段数据作为任务的参数。

(2) 一段代码实现这个任务。

这在 App 中，刚好可以使用 HTTP 的请求和回复。数据可以放入 HTTP 请求中，例如作为网表变量、JSON 或者编码之后的二进制数据。而回复的 URL，间接地指向一个在服务器上的服务函数。这样，App Engine 的任务队列 API 让我们把一个任务打包为一个 HTTP 请求数据和

目的 URL，有时也称其为网钩（web hook）。

因为目标 URL 就是一个会被执行的处理函数，而数据也已经打包在一起放入队列里。App 可以预先创建很多这样的网钩。App Engine 服务器就可以在系统资源可用时，一个个地发送这些数据请求到目标 URL，执行任务。而且更重要的是，这些任务是不相关的，可以大规模并发执行。这样，App 的整体性能和数据流通量就可以成倍地提高。

在一个 App 里，任务 URL 是相对于我们 App 自已的 URL 的路径。例如抓取 RSS 的任务的 URL 可以是/worker/fetch_feed。而我们上例的任务 URL 就简单地叫 /worker。

通常我们需要限制用户直接访问任务 URL。做法是配置 app.yaml，把指定的任务 URL 加上管理员权限。例如：

```
handlers:
- url: /worker
  script: _go_app
  login: admin
```

8.13.2 任务执行

任务加入队列后，App Engine 会在第一时间执行它。一个任务最多可以执行 10 分钟。时间结束时，App Engine 会发送一个异常中断，我们的 App 可以接收处理，例如保存结果和记录进度等。

在队列里的任务执行（task execution）的顺序，取决于任务的内容和队列的属性。但也可以指定任务的某些属性，例如 ETA 预估抵达时间，用来特地安排执行顺序。这些可以参考 queue.yaml 的配置。

任务处理函数执行失败，会返回 200-299 之外的 HTTP 状态码。此时，App Engine 会自动重试直到任务执行成功。当然，App Engine 不会太过频繁地重试导致我们 App 接收太多处理请求，目前，最多是每小时重试一次失败的任务。

需要注意的是，任务处理函数也必须保证使用同样的数据多次请求，需要得出同样结果，也就是不会有副作用。这是因为有时 App Engine 会多次执行同一个任务，尽管任务已经成功。

8.13.3 队列

大量使用任务可以得到比较高的 App 性能。我们也需要管理这些大量任务的执行，避免某些任务消耗太多资源。

队列就是任务的容器。通过改变队列的属性，我们可以间接控制任务的执行。例如，要限制每秒发送不超过 10 封电子邮件，我们可以创建一个队列，比如 emailq，配置它每秒执行 10 个任务，然后我们把发送电子邮件的任务都插入此队列。这样，即便一次插入几千个电子邮件发送任务，App 也只会每个最多发送 10 封电子邮件。

队列，顾名思义，是新任务排在队尾，而队头的任务会被执行。但 App Engine 有一个优化——那些低延迟的任务可以插队被优先执行。同时，App Engine 也可能同时执行一个队列的多个任务。这些，在编写 App 任务处理函数时需要考虑。

每个 App 都有个默认的队列使用默认的配置。它会每秒 5 次地执行任务。此队列叫 default，也可以像其他队列一样在 queue.yaml 中配置。

如果我们没有指定任务 URL，则自动从队列名得到默认的 URL。例如队列 emailq 里的任务，如果没有指定一个 URL，则都会发送到 /_ah/queue/emailq。这样，我们可以把纯数据的任务插入队列，然后统一使用默认的任务 URL 处理函数。

问题是如果我们的 App 没有准备这个默认任务 URL 函数，App Engine 仍会发送任务请求到那个 URL，并从系统得出 URL 不存在的错误回复。因为回复的 HTTP 状态码是 404，App Engine 会一直重试。如果我们在 App 的记录（log）里发现大量的 404，原因可能就是 App 忘记

提供这个默认的任务 URL 函数。

在 SDK 开发环境中，可以在管理员界面看到和操作这些任务：

http://localhost:8080/_ah/admin/taskqueue

如果使用任务仍不能满足对后台执行时间和性能的要求，App Engine 还提供了真正的后端处理服务。不过，此时不再对 App 按 CPU 使用收费，而是要按照执行时间计费了。

8.14 后端

App Engine 的后端（backend）比普通的 App 可以使用更多的内存和 CPU，并不再有请求时限的限制。它们适合需要更高性能、连续执行的程序。后端可以理解为一台独立的计算机，尽管它们是分时复用在更强大的实际计算机上。

为了理解普通 App 和后端的区别，我们参考表 8-1，同时学习它们的特性。

表8-1 App和后端比较表

特 性	App	后 端
时限	HTTP请求30秒 任务10分钟	无限制
CPU	按使用计费	可以配置600 M到4.8 GHz
内存	128 MB	从128 M到1 GB
驻留	按需	可以驻留
启动	处理请求时启动	请求/_ah/start启动
关闭	空闲时自动关闭	从管理器关闭
实例	实例无名	实例有自己固定的URL
扩展	随请求自动增加实例	实例个数预先定义
公开	公开或私有请求	默认接收私有请求

具体的后端使用和管理，请读者参考:

code.google.com/appengine/docs/go/backends

8.15 能力 API

由于 App 和后端程序是以分时共享方式使用 GAE 提供的软硬件服务，因此不可避免而且是不可预期地会出现 App Engine 系统内部错误，导致某些服务暂时不能使用。能力（capability）API 就提供给我们一个可靠的手段，去检测这些不幸地服务中断，让 App 进行相应处理。

例如，使用数据库时，可以检测数据库离线，并提示用户：

```
func handler(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    if !capability.Enabled(c, "datastore", "*") {
        http.Error(w, "This service is currently
unavailable.",
            503)
        return
    }
    // ok
}
```

这只需一个函数：Enabled。可以检测的项目如表 8-2 所示。

表8-2 能力检测表

能　力	参　数
大件库	blobstore
数据库读	datastore_v3
数据库写	datastore_v3,write
电子邮件服务	mail
内存缓冲	memcache
任务队列	taskqueue
URL抓取	urlfetch

8.16 电子邮件API

GAE可以以App管理员的名义发送电子邮件；也可以使用用户的谷歌账户发送。App可以用不同的地址接收电子邮件。使用电子邮件 API 可以发送邮件，而邮件的接收是靠从App Engine发来的HTTP请求。

8.16.1 发送

通过简单的mail.Message赋值，我们就可以指定电子邮件的内容Boby、发信人Sender、接收人To和主题Subject。发信人可以是App的管理者，或者使用谷歌账户登录的当前用户，也可以是此App合法的接收地址。

例如发送一个注册确认电子邮件：

```
import (
        "appengine"
        "appengine/mail"
        "fmt"
        "net/http"
)

func confirm(w http.ResponseWriter, r *http.Request) {
        c := appengine.NewContext(r)
        addr := r.FormValue("email")
        url := createConfirmationURL(r)
        msg := &mail.Message{
                Sender:  "Example.com Support<
support@example.
                  com>",
                To:       []string{addr},
                Subject: "注册确认",
                Body:    fmt.Sprintf(confirmMessage,
url),
        }
```

```
            if err := mail.Send(c, msg); err != nil {
                    c.Errorf("Couldn't send email: %v", err)
            }
    }

    const confirmMessage = `
    感谢您的注册!请单击确认您的电子邮件地址: %s
    `
```

8.16.2 接收

要使用 App Engine 帮助 App 接收（receive）电子邮件，就需要在 App 的 app.yaml 中加入：

```
inbound_services:
-mail
```

此后，所有发送到string@appid.appsportmail.com的电子邮件，都会由 App Engine 转发到 Appappid 的路径/_ah/mail。此处的 appid 指得是我们 App 注册的 id，而 string 则是任意的字符串，需要在 /_ah/mail 中注册的函数处理。例如：

```
func init() {
        http.HandleFunc("/_ah/mail/", incomingMail)
}

func incomingMail(w http.ResponseWriter, r *http.
Request) {
        c := appengine.NewContext(r)
        defer r.Body.Close()
        var b bytes.Buffer
        if _, err := b.ReadFrom(r.Body); err != nil {
                c.Errorf("Error reading body: %v", err)
                return
        }
        c.Infof("Received mail: %v", b)
}
```

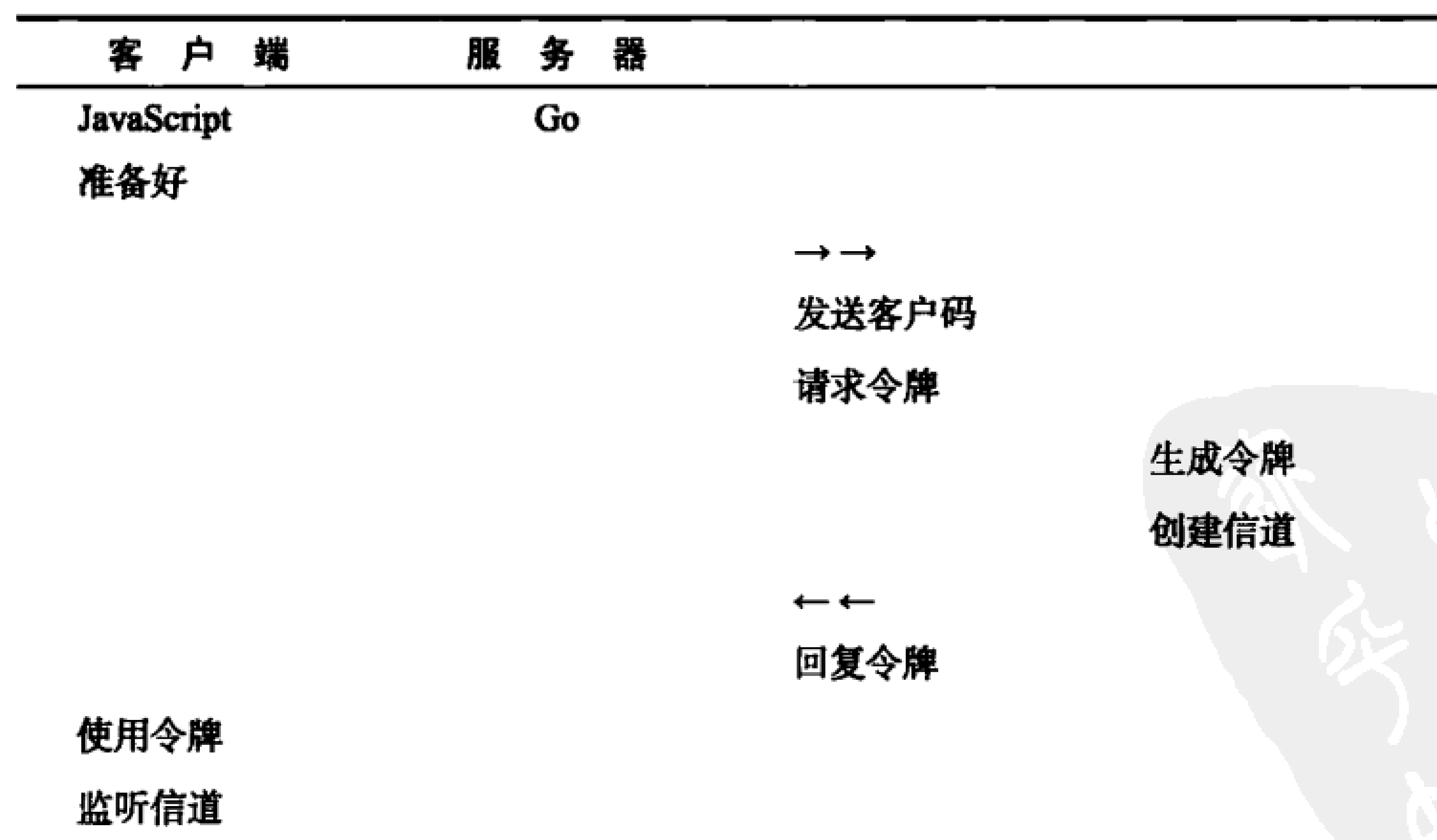
8.17 信道 API

信道的英文名虽然也叫 channel，它和 Go 的去程的程道完全无关。App Engine 的信道 API，提供的是 App 和浏览器的 JavaScript 客户程序的通信手段。

通常情况下，浏览器主动发送请求，服务器被动地响应请求并回复给浏览器显示。如果浏览器没有请求，服务器不可以主动地发送信息（比如一条通知）给浏览器的。要保持一种双向平等的通信，可以使用 HTTP 的轮询（polling），可以使用新颖的网络套接字（web socket）技术，此处也可以使用 App Engine 提供的信道 API。

App 信道是单向的，数据从服务器发送到浏览器。而浏览器客户端只需发送普通的 HTTP 请求给 App 服务器。

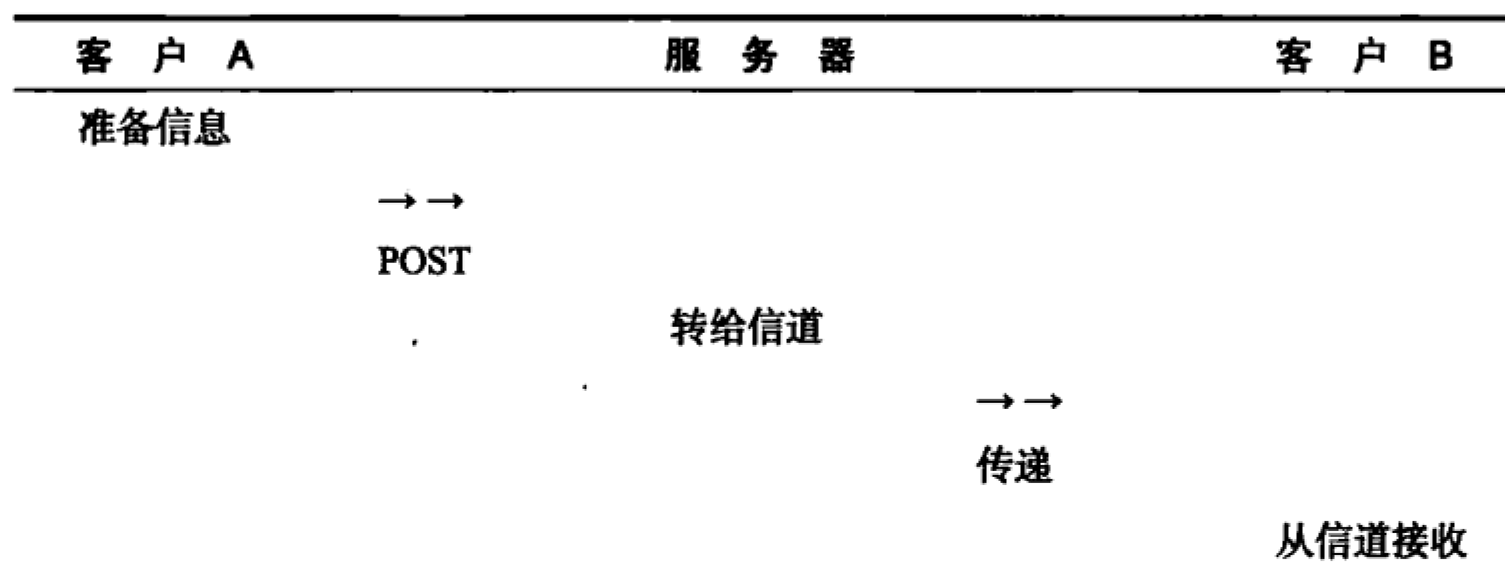
例如，运行在浏览器上的 JavaScript 客户端程序可以主动请求，让服务器上的 Go 程序生成一个信道：



客　户　端	服　务　器		
JavaScript	Go		
准备好			
		→→	
		发送客户码	
		请求令牌	
			生成令牌
			创建信道
		←←	
		回复令牌	
使用令牌			
监听信道			

客户码（Client ID）用来代表唯一的客户端，服务器使用此码找到对应的信道，发送信息给这个客户端。而令牌（token）则是此信道的秘密，同时也包含信道有效期等内容。客户端使用此令牌打开并监听信道。

客户端可以发送一个 HTTP 请求附带一段信息，服务器会使用其客户码把信息通过信道发送给指定的客户端，从而实现两个客户的间接通信。例如：



客　户　A		服　务　器		客　户　B
准备信息				
	→→ POST			
		转给信道		
			→→ 传递	
				从信道接收

需要使用信道 API 的读者，可以参考相关文档。它不仅要用到 Go 语言，还需要 JavaScript，这已经超出本书的范围，我们就不再深入介绍了。

8.18 小结

云计算是目前计算机技术最新的发展动态。Go语言生逢其时，必将成为支撑云计算的重要动力。读者学习完本书介绍的Go语言的方方面面之后，如果能够顺势应用在云计算上，会发现Go在这个新的领域有着极大的潜力，也必会成为你事业腾飞的助力。

第9章
标 准 包

A designer knows that he has achieved perfection
not when there is nothing left to add,
but when there is nothing left to take away.
（设计师知道事情完美的时候，
不是没得可加了，而是没剩什
么可以再减了。）
——Antoine de St-Expure

按照C语言的传统，语言本身应该是一个核，应该尽量紧凑和坚固。外围和操作系统以及其他系统的接口，加上大量的实用工具，可以看成肉。后者也非常重要——尤其在以貌取人的年代。丰满、匀称、光洁、高挑、灵活，这些都可以形容Go语言提供的标准包。此章，我们简单总结一些最常用和实用的包，具体和全面的内容，还需要读者自行参考godoc。

9.1 格式包

fmt 包提供各种 Print 函数打印输出各种格式的字符串，以及各种 Scan 函数，扫描输入格式的字符串。

最简单的就是 Print。它直接输出给定的字符串，例如 fmt.Print("Hello")。如果给的是个数字或者字符，则输出对应的十进制表示。如果是个变量，而且变量的值是数值或者字符串，则按上述相同的方式输出。组合类型的变量，也都有合适的字符串表示。而且，我们可以自己定义一个类型的 String()方法,使各种 Print 函数可以执行此方法，输出这个类型变量值的字符串表示。

Print 可以使用多个值或变量作为参数，中间用逗号隔开，Print 会自动在两个数值之间输出一个空格。例如：

```
package main
import "fmt"
func main() {
    var i = 42
    var s = "Answer"
    fmt.Print(s,"is",i,3.14,'\n',"\n")
}
```

结果：

```
Answeris42 3.14 10
```

我们一直在用的 Println，它不仅能自动输出结尾的"\n"，在两个数值之间自动加空格，而是每项直接都加空格输出。例如把上例换为：

```
fmt.Println(s,"is",i,"\n",3.14,'\n')
```

注意结果第二行 3.14 前的空格：

```
Answer is 42
 3.14 10
```

Println 的 ln 就是 line（行）的缩写，还有一个是 Printf 函数，f 是 format（格式）的缩写。Printf 比较复杂，它有自己一套小小的语言，我们用专门的一节介绍。

9.1.1 格式输出Printf

Printf 函数的第一项必须是个字符串，用来规定后面每项变量和数值的输出格式。例如：

```
package main
import "fmt"
func main() {
    var i = 42
    var s = "Answer"
   fmt.Printf("%s is %d\n %f %c %x\n", s, i, 3.14,'\n',
'\n')
}
```

结果：

```
Answer is 42
 3.140000
 a
```

Printf 第一项的格式字符串中，出现了一些以%开始的字符。类似转义字符，它们也有特殊的含义，在 Go 的文档里，称它们为“动词”，它们代表的是“替换”。例如%s 会被替换为一个字符串，此处是变量 s 的值，而%d 被替换为变量 i 的十进制整数值，%f 是浮点数 3.14，%c 是字符'\n'，也就是换行，而%x 是十六进制的整数，所以输出的 a 不是字符 a，而是字符'\n'的 Unicode 编码的十六进制表示 U+000a。

这些“动词”一般都是其代表的英文单词的缩写：

s　string 字符串

d　decimal 十进制整数

c character 字符

f float 十进制浮点数

x hex 小写 16 进制整数

9.1.2 动词表

全部动词（verb）可以这样按表 9-1 分类。

表9-1 各种数据类型对应的动词

整数：
b binary 2进制整数
o octal 8进制整数
d decimal 十进制整数
x hex 小写十六进制整数
X heX 大写十六进制整数
浮点数和复数部分：
b binary浮点数二进制
e exponent小写e浮点计数法，例如-1234.56e+78
E exponent小写E浮点计数法，例如-1234.56E+78
f float 十进制浮点数，例如-1234.456
g general，选择f或者e输出较短的格式
G General，选择f或者E输出较短的格式
字符：
c character 字符
q quote 单引号括起的字符
U Unicode格式的字符，就是 U+%04X
字符串或者字节切片：
s string或slice 不加转义的字符串或者字节切片
q quote 双引号括起的字符串
x hex 小写十六进制，每个字节两个字符
X heX 大写十六进制，每个字节两个字符

（续）

指针：
p　pointer 由0x开始的16进制整数，代表内存地址
布尔：
t　true或者false
通用：
v　value 值的通用格式
#v　#value，Go语法表示的值
T　Type Go语法表示的类型
%　表示%，所以%%是一个%

9.1.3 宽度和精度

我们还可以规定输出时的宽度和精度。例如

```
fmt.Printf("Answer is %6d and %0.4f", 42, 3.14159)
```

输出：

```
Answer is     42 and 3.1416
```

宽度和精度必须是两个整数，出现在%和动词之间。精度的前面要加小数点，因为它指的是浮点数小数点后的位数。而宽度则是整个数值（包括小数部分）至少要占的字符数，左侧加足够的空格。如果实际需要输出的位数大于宽度，不会截断，而是遵照原值输出。

对于字符串，宽度的概念是一样的，而精度是指输出的最多字符个数，多出的部分被截断丢弃。例如：

```
fmt.Printf("Say %6.4s!", "hello")
```

输出：

```
Say   hell!
```

其实我们还可以不直接指定宽度和精度，而用*代替，让 Printf 自己获取后面对应项的宽度或者精度的值。例如：

```
fmt.Printf("Say %*.4s!", 6, "hello")
```

对应项指的是，格式字符串后从左至右的每项，替换格式字符串中出现的每一个动词、宽度或者精度。

9.1.4 报错

如果次序出错导致类型不对，或者参数个数不够或过多，Printf 都会用一个错误信息替换，帮助我们发现错误。例如：

```
fmt.Printf("Say %6.4s!", 6, "hello")
```

输出：

```
Say %!s(int=  0006)!%!(EXTRA string=  hell)
```

具体的报错格式为：

类型不对：%!verb(type=value)

项数不够：%!verb(MISSING)

项数过多：%!(EXTRA type=value)

非整数宽度：%!(BADWIDTH)

非整数精度：%!(BADPREC)

9.1.5 额外标记

如果我们希望在右侧添空格，可以在宽度前面加负号。类似的标记还有：

+ 正数值加正号；

- 宽度内靠左对齐，右侧添空格；

0 宽度内添 0 而不是空格；

另类格式：

#o：八进制数 0 开头；

#x：十六进制数 0x 开头；

#X：十六进制数 0X 开头；

#p：不输出 0x；

#q：反引号的字符串；

#U：输出可显示的字符而不是 U+的格式。

另外，如果在动词前有一个空格，则% d 会留个空位给正负号，而% x 和% X 则会在每两个字符间留空格。例如：

```
fmt.Printf("% d\n% d\n% x",-42,42,"你好")
```

输出：

```
-42
42
e4 bd a0 e5 a5 bd
```

9.1.6 格式输入

类似 Printf 使用的动词还可以用在 Scanf 中，替换对应格式的字符，成为后面每项变量的值。例如：

```
package main
import "fmt"
func main() {
    var i = 0
    var s = ""
    fmt.Scanf("%s is %d", &s, &i)
    fmt.Printf("%s is %d", s, i)
}
```

在命令行执行此程序，会要求我们输入，如果输入 Answer is 42，则变量 s 的值是字符串"Answer"，而变量 i 的值是整数 42。如果我们输

入 Answer unknown，则变量 i 不会被扫描替换，仍是 0，但我们看不到 Scanf 报错。实际上，fmt 包的这些 Print 和 Scan 函数，都会返回处理的项数和错误。如果错误不是 nil，我们就要查看报错信息并相应处理。例如此处，可以用 fmt.Println 输出 fmt.Scanf 返回的处理项数和错误：

```
fmt.Println(fmt.Scanf("%s is %d", &s, &i))
```

如果输入是 Answer unknown，得到：1 input does not match format。

注意，Scanf 中每项都是基本类型变量的指针（即用&取得的地址），而不是变量自身。这是因为只有利用指针，才能通过调用函数改变外围变量的值。

下面只列出 Scanf 的动词和 Printf 动词的不同之处：

p　　未实现

T　　未实现

e E f F g G同等对待不加区别

s v　　扫描读到的是一个空格分隔的字符串

我们也可以指定宽度，但不可以指定精度。宽度指的是最多扫描的字符个数。

使用 s 和 v 扫描读取字符串要特别注意。因为空格用做字符串扫描的结束标志，我们无法一次读出包括空格的一行字符。而且，连续的空格作为一个空格对待。这一规则同样也适用于格式字符串内的连续空格。

除动词之外的所有字符必须有对应输入，否则 Scanf 报错并停止扫描。

9.1.7 字符串格式

我们一直说输入输出，但并未提及输出到哪里，从哪里输入。其实，

在运行时，程序都会有一个标准输出和一个标准输入，通常是命令行、显示器和键盘。如果我们要输出到一个字符串而不是标准输出，可以使用 Sprint、Sprintln 和 Sprintf。它们的用法和 Print、Println、Printf 几乎一样，只是返回的不是处理的项数和错误，而是结果字符串。这样，我们可以很容易地按规定格式把变量和常量的值转换为一个字符串。例如要将一个整数转换为二进制格式的字符串：

```
s = fmt.Sprintf("%b",2)
```

Sscanf 也是一样，只是待扫描的输入不是来自标准输入，而是来自第一项的字符串，例如：

```
package main
import "fmt"
func main() {
    var i = 0
    var s = ""
    var t = "Answer is 42"
    fmt.Sscanf(t, "%2s is %d", &s, &i)
    fmt.Println(s, i)
}
```

9.2 字节包

因为字符串的不可以改变的，而且字符串和字节切片的底层都是字节数组，所以，在 Go 里，字节切片 []byte 类型和字符串结合使用的情况非常多见。甚至有两个包：bytes 和 strings，提供了一些相似的函数，方便程序员处理这两种类型的值。例如，Count 用来统计不重叠的字段的个数，bytes.Count 使用 (s, sep []byte) int，而 strings.Count 是(s, sep string) int。

```
package main
import (
    "bytes"
    "strings"
     "fmt"
)
func main() {
    s := "你 Go 了吗?"
    fmt.Println(bytes.Count([]byte(s), nil))
    fmt.Println(strings.Count(s, ""))
}
```

结果都是 7。这是因为 Count 认为 nil（空字节切片）或者""（空字符串）的字段，指的是 Unicode 字符之间的部分，也就是字符个数加 1。

注意，我们用[]byte(s)把字符串 s 转换为字节切片，同样也可以用 string(b)把字节切片转换为字符串。因为要确保字符串的内容不会通过字节切片被更改，编译器会在转换时进行复制。所以我们不要做无谓的转换，避免因大量复制降低执行效率。

Split 把一个字符串或者字节切片 s 按字段 sep 分隔为一个字符串切片或者字节切片的切片。听起来很难，直接看函数签名就一目了然了。

```
func Split(s, sep string) []string
func Split(s, sep []byte) [][]byte
```

同样，如果 sep 是 nil 或者""，则按字符分隔。但注意结果不会变成[]rune。例如上表的代码写成如下形式：

```
fmt.Println(bytes.Split([]byte(s), nil))
fmt.Println(strings.Split(s, ""))
```

输出：

```
[[228 189 160] [71] [111] [228 186 134] [229 144 151]
[63]]
[你 G o 了 吗 ?]
```

可以清楚地看到仍是 UTF-8 编码形成，而没有变成 Unicode 码值。

Buffer

bytes 包还有一个重要的类型：Buffer。它提供了一个自动管理的字节切片，不必担心内存分配问题，可以直接读写。它类似队列：写入的字节加在队尾，读从队头开始。如果队列是空的，代表所有写入的字节都读完了。再读就返回错误 io.EOF。

如果我们使用 var b byte.Buffer 或者 b := new(byte.Buffer)，其自动管理的字节切片 b.buf 最初是空的，len 为 0，cap 不是 0。假设 cap 是 1，此时 b.buf 的情况是：

```
| 0 | off=0; len=0; cap=1
```

如果使用 b.WriteByte(1)写入第一个字节，这个字节还可以装入 b.buf：

```
| 1 | off=0; len=1; cap=1
```

如果使用 b.WriteByte(2)再写入一个字节，b.buf 无法装入新的字节，只好重新分配一块大一倍的新字节切片作为 b.buf。

```
| 1 | off=0;
| 2 | len=2; cap=2
```

b.ReadByte()读到1，此时队头（off）移动到第二个单元。

```
| 1 |
| 2 | off=1;len=1; cap=2
```

再次使用b.ReadByte()读到2，此时b.buf空，自动归0，以便重新使用这个字节数组。

```
| 1 | off=0;len=0; cap=2
| 2 |
```

我们来仔细看看Buffer是如何自动管理其底层的数组的。

```
func (b *Buffer) grow(n int) int {
    m := b.Len()
    // If buffer is empty, reset to recover space.
    if m == 0 && b.off != 0 {
        b.Truncate(0)
    }
    if len(b.buf)+n > cap(b.buf) {
        var buf []byte
        if b.buf == nil && n <= len(b.bootstrap) {
            buf = b.bootstrap[0:]
        } else {
            // not enough space anywhere
            buf = make([]byte, 2*cap(b.buf)+n)
            copy(buf, b.buf[b.off:])
        }
        b.buf = buf
        b.off = 0
    }
    b.buf = b.buf[0 : b.off+m+n]
    return b.off + m
}
```

b.Truncate(0)只是b.off = 0，而b.bootstrap是最初的底层数组。

9.3 模板包

模板包不仅可以做模板替换，而且它提供了一整套的小语言，包括变量、选择、重复和函数。我们来看一下。

```
package main

import (
    "os"
    "text/template"
)

const tpl = `
How many roads must {{.}} walk down
Before they call him {{.}}
`

func main() {
    tmpl := new(template.Template)
    tmpl.Parse(tpl)
    tmpl.Execute(os.Stdout, "a man")

    //Output:
    //How many roads must a man walk down
//Before they call him a man
}
```

一个最基本的模板，也要 new、Parse 和 Execute，就好比拿出一张新纸，观摩原件，然后下笔临摹。Parse 的原件里面，有些圈在两对大括号中间的预留空位，在 Execute 时，会被逐一填写。而用大括号括住的那个点儿（.），就是圈点的点，代表当前执行的值，是 Execute 的 "a man"，它是一直不变的当前值，替换 Parse 的 tpl 中的每个点。

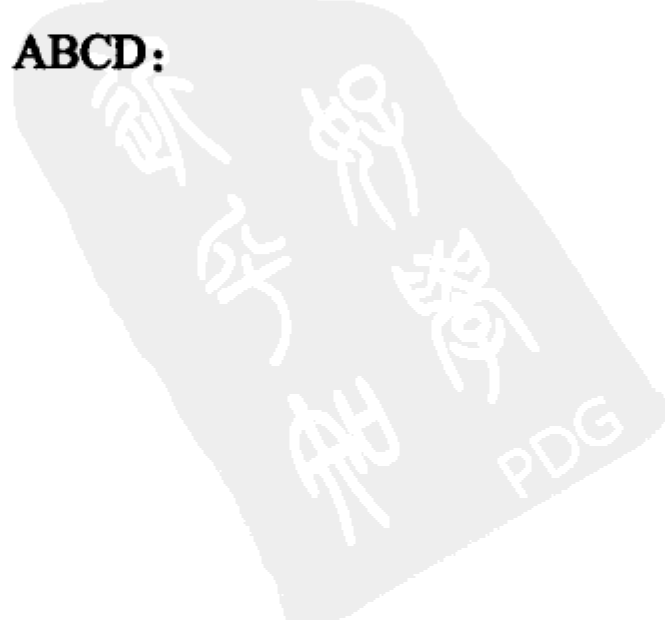
当然，点是可以动态改变的。如果不是一个值，而是一组值。如果

是数组、切片或者映射，我们可以用 range 来遍历每一个点。例如：

```
package main

import (
    "os"
    "text/template"
)

const tpl = `
知止{{range .}}而后能{{.}}，{{.}}{{end}}而后能得。
`

func main() {
    var 大学 = []string{"定", "静", "安", "虑"}
    tmpl := template.New("")
    tmpl.Parse(tpl)
    tmpl.Execute(os.Stdout, 大学)

    //Output:
    //知止而后能定，定而后能静，静而后能安，安而后能虑，虑而
后能得。
}
```

range 里的点，是指字符串切片“大学”，而它和对应{{end}}之间的内容，range 会一遍遍地重复，每遍都会把点替换为当前单元的值，直到最后一个单元。如果 range 遍历的点是空的数组切片或者映射，则它和{{end}}间的内容不会输出，而不仅是不会替换。

如果这组值是组合，则需要圈点项名。注意只有以大写字母开头的项名才是公开的，才可以被圈点，所以，才有下例里的 ABCD：



```
package main

import (
    "os"
    "text/template"
)
```

```
const tpl = `<tr>
<td>{{.A 姓名}}</td>
<td>{{.B 级别}}</td>
<td>{{.C 性别}}</td>
<td>{{.D 爱好}}</td>
</tr>
`

type 剑客 struct {
    A 姓名, B 级别, C 性别, D 爱好 string
}

func main() {
    var 剑客录 = 剑客{"骨惊飞", "北极", "男", "女"}
    tmpl := template.New("")
    tmpl.Parse(tpl)
    tmpl.Execute(os.Stdout, 剑客录)

    //Output:
    //<tr>
    //<td>骨惊飞</td>
    //<td>北极</td>
    //<td>男</td>
    //<td>女</td>
    //</tr>
}
```

当然，切片组合等也可以结合在一起使用。我们就自己挥发想象力吧。

就像range用来重复，if用来判断。如果点是空的，就执行{{else}}之后的部分，直到{{end}}。如果没有 else，就什么都不输出。我们以 if 的使用作为例子，并看看如何用 Must 来检查 Parse 的 tpl 有没有错误：

```
package main

import (
```

```
        "os"
        "text/template"
    )

    const tpl = `{{range .}}
    {{if .B 级别}}{{.A 姓名}}大爷请上座。
    {{else}}{{.A 姓名}}小弟，欢迎再来。
    {{end}}
    {{end}}
    `

    type 剑客 struct {
        A 姓名, B 级别, C 性别, D 爱好 string
    }

    func main() {
        var 剑客录 = []剑客{
            {"骨惊飞", "北极", "男", "女"},
            {"骨灰飞", "", "有", "无"},
        }
        tmpl := template.New("")
        template.Must(tmpl.Parse(tpl))
        tmpl.Execute(os.Stdout, 剑客录)

        //Output:
        //骨惊飞大爷请上座。
        //骨灰飞小弟，欢迎再来。
    }
```

也可以使用变量。例如我们可以得出range时每一单元的下标和值，赋值给变量$i 和$v，并直接在其后使用。注意变量只能是$开头，只有一个$也行。没有人民币符号，也没有其他符号。

```
    package main

    import (
        "os"
        "text/template"
```

```
)

const tpl = `{{range $i,$v := .}}{{$i}}: {{$v.A姓名}}
{{end}}
`

type 剑客 struct {
    A姓名, B级别, C性别, D爱好 string
}

func main() {
    var 剑客录 = []剑客{
        {"骨惊飞", "北极", "男", "女"},
        {"骨灰飞", "", "有", "无"},
    }
    tmpl := template.New("")
    template.Must(tmpl.Parse(tpl))
    tmpl.Execute(os.Stdout, 剑客录)

    //Output:
    //0: 骨惊飞
    //1: 骨灰飞
}
```

如果阿拉伯数字看着不舒服，我们多下些功夫，用大写数字表示：

```
package main

import (
    "os"
    "text/template"
)

var (
    zniota []rune
    fmap   = template.FuncMap{
        "znmap": znmap,
    }
)
```

```
    func init() {
        zniota = []rune("壹贰叁肆伍陆柒捌玖拾")
    }

    func znmap(i int) rune {
        return zniota[i]
    }

    const tpl = `{{range $i,$v := .}}{{$i | znmap | printf
"%c"}}: {{$v.A 姓名}}
    {{end}}
    `

    type 剑客 struct {
        A 姓名, B 级别, C 性别, D 爱好 string
    }

    func main() {
        var 剑客录 = []剑客{
            {"骨惊飞", "北极", "男", "女"},
            {"骨灰飞", "", "有", "无"},
        }
        tmpl := template.New("")
        tmpl.Funcs(fmap)
        template.Must(tmpl.Parse(tpl))
        tmpl.Execute(os.Stdout, 剑客录)

        //Output:
        //壹: 骨惊飞
        //贰: 骨灰飞
    }
```

这里，变量$i 对应的下标用管道线|传给 znmap 这个注册的函数，用来从 zniota 得到对应的汉字，再用|传给 printf 这个内置的函数，用%c 输出对应的字符。znmap 的注册是在 Funcs 中完成的，它需要一个 FuncMap 的映射，此处是 fmap。

这些基础知识可以用一阵子了。要精通模板包的话，还是专心研读 godoc text/template 吧。

9.4 正则表达式包

我不想夸大正则表达式（regular expressions）有多么强大又是多么令人生畏。它是由 Stephen Kleene 在 20 世纪 50 年代提出的概念，但是是由 Ken Thompson 于 1968 年首次实现。他把这种表达式直接编译为机器码，从而能非常快速地在大量文字中搜索到所需内容。最早的几个软件专利就包括 Ken 的这项技术。而靠着 Unix 的普及，以及在 Perl 和大量编程语言中的应用，这种简单的文字描述被固定下来，被广泛地应用于文字的检索和替换，使得正则表达式成为每个程序员必须掌握的小语言。

学习正则表达式的最好方式，是从 Brian Kernighan 在普林斯顿的讲义开始。读者们要知道，K&R 的 K 和 AWK 的 K 就是指这个人。他最著名的事迹就是和 Ritchie 合写了《C 程序设计语言》那部经典。

那篇讲义提到 Rob Pike 写正则表达式的故事，Rob 跟他讲：

"我脑海中的代码就这样活生生地出来了。唯一的挑战是在合适的边界终止递归。也就是说，递归不仅仅是实现方法，它就是写程序时思绪的影子，这也是代码简单的部分原因。可能最重要的是，我开始时并没有设计，我就直接开始写，看着它发展，然后突然完成了。"

K 说他从未见过这么短一段代码能做这么多事情，这么思想丰富，这么有洞察力。如果程序员的祖师爷都这么说，我们后辈就仔细地读读吧。这里是 Go 的版本：

```
package pregexp

func Match(s string, t []byte) (bool, error) {
    r := []byte(s)
    if r[0] == '^' {
        return matchhere(r[1:], t), nil
    }
```

```
    for i := 0; i < len(t); i++ {
        if matchhere(r, t[i:]) {
            return true, nil
        }
    }
    return false, nil
}

func matchhere(r, t []byte) bool {
    switch {
    case len(r) == 0:
        return true
    case len(r) > 1 && r[1] == '*':
        return matchstar(r[0], r[2:], t)
    case r[0] == '$' && len(r) == 1:
        return len(t) == 0
    case len(t) > 0 && (r[0] == '.' || r[0] == t[0]):
        return matchhere(r[1:], t[1:])
    }
    return false
}

func matchstar(c byte, r, t []byte) bool {
    for i := 0; i < len(t); i++ {
        if matchhere(r, t[i:]) {
            return true
        }
        if t[i] != c && c != '.' {
            break
        }
    }
    return false
}
```

为了和Go的regexp包区别，此处Pike的代码称为pregexp包。它们的Match函数签名是相同的，都是判断一个字符串的样式是否出现在另一个字节数组中。这个样式除了要求所有字节都要对应外，还使用

四个特殊符号：

```
.    对应任一字节
*    前一字节没有或者多次出现
^    开始字节
$    结束字节
```

这样，如果从左至右在“apple”里面查找如下样式，Match 应该是 true：

```
"^",
"^a",
"^ap*",
"^app*le$",
"e$",
"ap.le",
"ppl",
```

而下面的样式没有出现，Match 应返回 false：

```
"$",
"^w",
"s$",
"pps",
"apples",
```

尽管只有短短的几十行，只有 4 个特殊字符，但它已经能够完成基本类似 Unix 里 grep 的查找功能。我们把每个找到的内容，也就是样式对应的部分，称为一个“匹配”（match），把*字符称为“元字符”（metacharater）。

Unix 的通用字符查找程序 grep 里，有下列元字符：

```
*    没有或多个
+    一个或多个
?    没有或一个
|    两者选一个
```

*、+、?是重复，|是选择，还有就是顺序排列。在样式里面，重复

优先于排列，排列优先于选择，不然就需要使用小括号。例如：12*3|45是指(1(2*)3)|(45)。这和四则运算的12*3+45是指((12)*3)+(45)略有不同。

好了，我们把Pike的代码读一遍，从而对RegExp的操作有个初步的认识。尽管Go的regexp包的实现比它复杂几百倍，但道理是相同的。

Match是逐个字节进行比较的。我们把样式称为r，文档称为t。如果r以^作为开始，则t的开始字节必须和r之后的字节相同，我们用matchhere去比较。否则，从左至右缩短t，用matchhere去比较r，直到到达t的结尾还没有找到r，就返回false。

matchhere主要处理两种情况，也就是switch的最后一个case，即如果r和t的首字节相同，或者r的首字节是'.'，则r和t同时缩短，去掉相同的首字节，递归调用matchhere函数处理下一个字节，直到r的最后，比较过的字节都相同，表示在t中找到了r，返回true。如果r的最后是个$，并且t到了最后，也返回true表示找到。另一个case是如果r的下一个字节是*，我们需要matchstar辅助。

matchstar在t里连续地找r的*之前的字节，但每次找之前，都再次用matchhere确认r的*之后的部分如果在t里出现了，就返回true结束查找。这些因为*的定义是其之前的字节可以不出现在t里，也就是可以忽略。这样一直重复下去，直到找到，或者t中出现不对的字节，返回false。

注意此处的*是只要找到最短的情况就返回true。在Go的regexp中，*是要找到最长的情况。这对Match来说是无所谓的，只要找到就是true。但Go的regexp还有一些更有用的函数，是类型RegExp的方法，例如Find，能返回找到的那些字节，此时长短才重要，而Go还定义了类似的其他选最长的情况：

```
*          没有或者多次
```

\+ 一次或者多次

? 没有或者一次

{*n*, *m*} *n*~*m* 次

{*n*,} 至少 *n* 次

{*n*} 刚好 *n* 次

对于选最短的情况，Go 的定义是：

*? 没有或者多次

+? 一次或者多次

?? 没有或者一次

{*n*, *m*}? *n*~*m* 次

{*n*,}? 至少 *n* 次

n}? 刚好 *n* 次

如果抽象一下，可以说这些都是“重复操作”，而^%与字符自身，是“接续操作”，还有一种是“选择操作”，例如：

x|y x 没有 y 也行

[xy] x 或 y 都可以

[^xy] 除了 x 和 y 都可以

. 什么都可以

[A-Z] A~Z 的任意字母

[:alnum:] [0-9A-Za-z]

[:alpha:] [A-Za-z]

[:ascii:] [\x00-\x7F]

[:space:] [\t\n\v\f\r]

[:word:] [_0-9A-Za-z]

[:xdigit:] [0-9A-Fa-f]

Go 的 `regexp` 也支持转义字符，例如：

\w [:alnum:]

\W [^\w]

\d [0-9]

\D [^\d]

\s [:space:]

\S [^\s]

\n \012

\r \015

\t \011

\123 八进制数

\x7F 十六进制

\x{10FFFF} 十六进制字符

对于样式中出现的标点，例如*，如果不想作为特殊符号，而希望直接查找，可以用*代表，或者放在\Q与\E之间。

一下子讲了太多，我们可以留到以后慢慢消化，或者参考 godoc regexp。现在是故事时间。

著名的理科线条人网站 xkcd[①]有个著名的 regexp 故事：

“每当我学会点儿新技能，总想编个感人的故事，让我用它逢凶化吉。”

“Oh，那个杀手一定在她度假时跟上了。”

“要找他们就得在 200 MB 的电子邮件里找那些像地址的东西”

“没指望了。”

“你们闪开！”

“我懂正则表达式！”

“哒哒，PERL！”

“！！”

---

① http://xkcd.com/208/

比如我们的女主角说过要去颗小行星度假，那我们就从电子邮件里挖出那些符合小行星命名规范的部分吧。小行星都有临时编号和正式名称。例如 7934 Sinatra 星的临时编号是 1989 SG1，而 18610 Authurdent 星的编号是 1998 CC2。临时编号比较规整，就是发现的年份、可有可无的空格、一个大写字母（不包括 I 和 Z）代表 24 个半月，一个大写字母（不包括 I）代表那个半月里发现星星的序号%25，而后面的数字则是/25，如果是 0 可以不写。例如 2005 年 10 月的下半月发现了 13268 颗星，第一颗临时命名为 2005UA，最后的一颗是 2005UP530。

这样，小行星临时编号的样式可以表示为：

```
\d{4}\s*[A-HJ-Y][A-HJ-Z]\d*
```

下面的程序能找到所有符合这个样式的小行星。

```
package main

import (
    "fmt"
    "io/ioutil"
    "regexp"
)

var pat = `\d{4}\s*[A-HJ-Y][A-HJ-Z]\d*`

func main() {
    text, _ := ioutil.ReadFile("regexp.t")
    reg, _ := regexp.Compile(pat)
    for i, v := range reg.FindAll(text, -1) {
        fmt.Printf("%d : %s\n", i, v)
    }
    //Output:
    //0 : 1989SG1
    //1 : 1998CC2
}
```

我们的例子是：

```
(7934) Sinatra = 1989SG1 ist ein Asteroid des Hauptgürtels
2006年6月25日 - 12/06 - 小行星7934 Sinatra最接近地球

(18610) Arthurdent = 1998CC2
Discovered 1998 Feb. 7 at Starkenburg Observatory.
The earthling Arthur Dent is confronted with the
adversities of life, the universe and everything in a
highly amusing and entertaining way in Douglas Adams' famous
five-volume trilogy "The Hitch Hiker's Guide to the Galaxy".
```

使用 regexp，首先是编译（compile）样式字符串，目的是把它变为能快速执行的内部表示。例如一个状态机或者虚拟机，能机械地自动匹配目标切片或者字符串，返回找到的匹配。

类似 FindAll 的 RegExp 的方法共用 16 个：

```
Find
FindAll      FindString   FindSubmatch
FindIndex
FindAllString
FindAllSubmatch
FindAllIndex
FindAllStringIndex
FindAllStringSubmatch
FindAllSubmatchIndex
FindAllStringSubmatchIndex
        FindStringIndex
        FindStringSubmatch
        FindStringSubmatchIndex
                FindSubmatchIndex
```

这样我们一眼就看清了规律：Find 的方法只是找到第一个匹配；FindAll 的方法返回所有匹配，包含 String 的方法不用[]byte，而是 string；包含 Index 的方法不直接返回[]byte 或者 string，而是间接返回一对[]int，代表匹配部分的开始和结束下标。

9.5 时间包

目前最精确的估计是我们还在生活的宇宙已经存在了 130 亿年。因此 Go 的 time 包把时间的起点定在 2900 亿年前，而不是通常的 1970 年，应该可以说明一个问题，就是 Go 的时间是经得起时间的考验，不会造成类似 C 的千年虫那样的恐慌。了解一点内部细节，树立一些信心之后，我们开始介绍时间包的基本用法。

首先是得到现在时间，非常简单：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()
    fmt.Println(now)
    fmt.Println(now.Format(time.RFC3339))
    fmt.Println(now.Format("01 月 02 日 03 时 04 分 05 秒
06 年 07 区"))

    //Output:
    //Wed Feb 29 23:08:09 +0800 SGT 2012
    //2012-02-29T23:08:09+08:00
    //02 月 29 日 11 时 08 分 09 秒 12 年 07 区

}
```

Time 是一个可以精确到纳秒并带有时区的值。注意是数值而不是指针。它的 Format 方法可以用来按指定格式显示。默认的是传统的美国标准。比较紧凑的格式是 time.RFC3339。而我们也可以自己规定显示

的顺序。

自定义显示顺序很有意思，因为它是靠 01 到 07 来代表日期时间的，也就是例子里的 01 月 02 日 03 时 04 分 05 秒 06 年 07 区。还有几个英文单词代表英文显示的对应项：Mon 是星期一的缩写，Jan 是一月的缩写，Mondy 是星期一的全称，January 是一月的全称，PM 大写代表下午，pm 小写代表下午，MST 是时区代码。另外，2006 代表四位数字的年份，而 06 是两位数字；01 代表单个数字的月份前面补 0——02、03、04、05 也是一样；反之是 1、2、3、4、5；时区可以写为 Z0700 或者 Z07:00——此时如果时区是 UTC 则显示 Z，也可以写为-07、-0700、-07:00。最后，如果有小数点和 0，则显示小数格式的几分之一秒，0 的个数是小数点后的位数。例如.000 可以得到三位。

如果我们要打印本月的月历，可以这样：

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()
    yr, mo, _ := now.Date()
    d1 := time.Date(yr, mo, 1, 0, 0, 0, 0, time.UTC)
    d2 := time.Date(yr, mo+1, 1, 0, 0, 0, 0, time.UTC)
    d2 = d2.Add(-24 * time.Hour)

    w := d1.Weekday()
    _, ww := d1.ISOWeek()
    fmt.Println("\n周\t日    一    二    三    四    五    六    ")
    fmt.Printf("%d\t%*d", ww, int((w+1)*3), 1)
    for i := 2; i <= d2.Day(); i++ {
        if w++; w%7 == 0 {
```

```
            ww++
            fmt.Printf("\n%d\t", ww)
        }
        fmt.Printf("%3d", i)
    }
}
```

2012年的2月份的月历输出是：

```
//周  日  一  二  三  四  五  六
//5                1   2   3   4
//6   5   6   7   8   9   10  11
//7   12  13  14  15  16  17  18
//8   19  20  21  22  23  24  25
//9   26  27  28  29
```

Date 函数返回对应时间变量的年、月、日，Clock 函数返回时、分、秒；Date 函数则使用年、月、日、时、分、秒和时区生成一个时间变量。值得注意的是 Date 函数可以自动规整，我们利用此特性从下个月的第一天减掉24小时，得到这个月的最后一天，也就是时间变量 d2。可我们用的是 Add 加-24小时，因为 Add 方法是加一段时间得到新时间，AddDate 是加上给定的年月日；而 Sub 方法是减去另一个时间，得到一段时间。

这一段时间，是time.Duration类型，其实就是int64。函数ParseDuration 可以把类似"2h45m"这种字符串转换为数值。而函数 Since，就是 time.Now().Sub，得到从那个时间到现在经过的时间。另外 time 包的 Hour、Minute、Second、MilliSecond、Microsecond 和 Nanosecond 常量都是 Duration 类型，分别代表小时、分钟、秒、毫秒、微秒和纳秒。Duration 等于1，就是1纳秒。

同样，从时间变量的方法 Hour()、Minute()、Second()、

Nanosecond()可以得到此时间的小时、分钟、秒和纳秒，这些都不是 Duration 类型，而是 int。而 Year()、Month()、Day()可以得到年、月、日；Weekday()可以得到星期数； ISOWeek 得到是一年的第几个星期。

注意从 Weekday()得到的是 Weekday 类型的值。因为需要定义它的 String()方法，用来显示英文的 Sunday 到 Saturday，尽管它的实际类型是 int，值也只可能 0~6，所以这里可以用 int((w+1) * 3)来根据它的值，计算所需的空格数，提供给前面 Printf 的%*。

如果我们希望显示中文的星期几，可以参考此例重新定义它的 String()方法：

```
package main

import (
    "fmt"
    "time"
)

type Wday int

var days = []rune("日一二三四五六")

func (d Wday) String() string {
    return string(days[d])
}

func main() {
    now := time.Now()
    w := now.Weekday()
    fmt.Print("星期", Wday(w), "\n")
}
```

时间的方法 Equal、After、Before 可以和另一个时间变量比较；IsZero 可以检查是否还未赋值。Local()返回新的时间变量，代表本

地时区的时间；UTC()返回第 0 时区的时间；Zone()返回时区代码和数字；Location()则返回指向时区信息的指针；In 把此时间变量的时区设为给定值。

还有 Unix()返回从 Unix 的原点——也就是 1970 年 1 月 1 日 0 时 UTC 开始的秒数。UnixNano()返回纳秒数。

marshalJSON 和 UnmarshalJSON 用于 JSON 格式的编码和解码；而 GobDecode 与 GobEncode 则是 Go 自己的 gob 格式。

而时间函数除了 Now 和 Date，还有 Unix 用给定的秒数和纳秒数创建一个 Time 变量返回；Parse 用给定的格式字符串拆分另一个代表时间的字符串，创建一个 Time 变量返回。

这些就是全部的 Go 的时间处理函数和方法。就这 35 个。

9.6 超链接包

考虑到谷歌是家因特网公司，以及 Go 语言第一个大规模应用是谷歌的 App Engine 平台，就可以想象 Go 的 http 包会有多么精彩。这也是 Go 语言的集大成之处。本章仔细地琢磨一下这块可以成碧的璞玉。

9.6.1 http服务器和客户机

http 包可以支持服务器和客户机两端的操作。按照惯例我们首先搭建一个“打招呼”的服务器：

```
package main

import (
    "io"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter,
                    _ *http.Request) {
        io.WriteString(w, "Hi there!")
    })
    http.ListenAndServe(":1234", nil)
}
```

http 服务器是 ListenAndServe 函数。此处我们让它在 1234 端口接听。nil 使用默认的转接服务。而这个服务需要用 HandleFunc 注册。此处，注册的服务路径以/开始，一个无名服务函数紧随其后，其 w 是回复界面，而呼叫请求 Request 此时未用，是个_。

这样，当我们通过浏览器呼叫 http://localhost:1234 时，ListenAndServe 响应，并转接给 HandleFunc 注册的服务函数。它把"Hi

there!"写入回复 w，完成一次 http 请求服务，可以看到浏览器的页面显示了"Hi there!"。

稍加改进，当我们访问 http://localhost:1234/图灵，可以看到“图灵你好”。

```
package main

import (
    "fmt"
    "net/http"
)

func hi(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s 你好", r.URL.Path[1:])
}
func main() {
    http.HandleFunc("/", hi)
    http.ListenAndServe(":1234", nil)
}
```

这里唯一真正的改进是使用 URL 包里定义的 URL 类型的 Path，得到访问地址的内容。不用担心字符编码错误导致的乱码等，因为默认的是正确的“Content-Type:text/plain; charset=utf-8”，这一点浏览器的用户通常是看不到的，我们自己写个客户端来显示所有 http 通信的内容。

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    r, err := http.Get("http://localhost:1234/图灵")
    if err != nil {
```

```
            fmt.Println(err)
            return
        }
        defer r.Body.Close()
        fmt.Println("Header:")
        for k, v := range r.Header {
            fmt.Printf("%s :%s\n", k, v)
        }
        body, err := ioutil.ReadAll(r.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
        fmt.Printf("Body: \n%s\n", body)
    }
```

http.Get 访问一个 URL，试图呼叫连接服务器。如果成功，得到回复的 Header 和 Body，我们分别显示。注意，不再需要 http 连接时，应该用 Body.Close()断开连接，释放资源。

9.6.2 https加密通信

众所周知 http 是公开的、不加密的。而因特网上要保障用户的隐私，最简单可靠的方法是 https。也就是用 TLS 协议加密 http 的通信连接。只有服务器和请求 https 的浏览器页面，也就是通信的两端，可以看到解密的内容。而通信过程中经过的所有路由器、防火墙等网络设备，是无法解密通信的内容的。

不明白这一点的读者，可能感觉浏览像打电话，拨号接通互相问候。但实际上更像写信。每个从浏览器发出的请求，都要装入带服务器地址的信封，放入自己计算机的邮筒，转发给最近的邮局，再经过不同邮局间的多次转发，最终到达服务器的邮筒，并由服务器注册的对应函数处理。回复也是一样，只是这一来一去只需几分之一秒，对我们人类来说，

比真正的邮件快万倍，我们相当满意了。

但很多人可能没有意识到，就这一瞬间，所有的信件都可以被拆封阅读。不合时宜的信就此消失了，发信人久久等不到回复。而只有无是无非的才重新放回，假装无事发生一样。

https 就像服务器发给客户端一个配着锁头的信封，只有服务器才有钥匙打开。但发送给客户端时，并未锁上，客户端会当场制作一把一次性的钥匙和锁，放入此信封后，用服务器的锁头一锁，就安全地把这把临时锁匙递交给服务器，从而在以后的通信中，一直使用这把临时锁，而只有服务器和客户端才有钥匙。邮局再也无法拆看信件了。

服务器的锁可以简单理解为证书。实际理论比这个要复杂和缜密得多。因为培养读者对加解密的兴趣不是本书的目的，我们只需了解。

尽管 https 的证书是电子文件，但其价值不菲。我们练习用的就自己创建，自己承认吧。编译运行 http://golang.org/src/pkg/crypto/tls/generate_cert.go，得到的 cert.pem 就是证书，而 key.pem 就是服务器的秘密钥匙。

这样，我们就可以使用这个证书和密钥启动 https 服务：

```
package main

import (
    "log"
    "net/http"
)

func hi(w http.ResponseWriter, _ *http.Request) {
    w.Write([]byte("你好秘匿"))
}
func main() {
    http.HandleFunc("/", hi)
    log.Println("URL:")
    log.Fatal(http.ListenAndServeTLS(":10860",
"cert.pem", "key.pem", nil))
}
```

看起来几乎和 http 的服务器是一样的。这就是 Go 的 http 包的过人之处——API 非常规整。

使用浏览器访问 https://localhost:10860，一个合格的浏览器应该注意到服务器的证书是假的。假证书不是指看着不像真的，而是它完全不被认可。被浏览器认可的 TLS 证书只能从 200 多家发证机构及其认证的下属机构签发。大家也可能不知道，在因特网世界中，你不能相信任何一个网站。尽管它名气很大，它的 URL 也可能只是看着像真的。你也不能相信任何的插件或者扩展，尽管它能带来些许方便。在这个因特网的世界，你只能信任你的浏览器。如果它被黑，你的密秘就没有任何的保障，你所有的信息记录、交易、言论都等着或者已经被人利用了。慎重选择浏览器，要像慎重选择伴侣一样。

修改之前的 http 客户机 `Get("https://localhost:10860")`，也会得到同样的错误提示，而不会继续：

```
Get https://localhost:10860: x509: certificate signed
by unknown authority
```

实际上它使用的是默认客户机的 `Get`。我们可以使用自己的客户机，让 TLS 忽略这个安全检查：

```
package main

import (
    "crypto/tls"
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    url := "https://localhost:10860"
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{
```

```
                InsecureSkipVerify: true,
            },
        }
        c := &http.Client{Transport: tr}
        r, err := c.Get(url)
        if err != nil {
            fmt.Println(err)
            return
        }
        defer r.Body.Close()
        fmt.Println("Header:")
        for k, v := range r.Header {
            fmt.Printf("%s :%s\n", k, v)
        }
        body, err := ioutil.ReadAll(r.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
        fmt.Printf("Body: \n%s\n", body)
    }
```

这里的 Transport，就是 https 客户机和服务器通信连接，它的 TLSClientConfig 只在 https 时使用，如果是 nil，或者它指向的 Config 的 InsecureSkipVerify 是 false，则会进行必须的证书认证。此处我们设置为 true，忽略这个步骤，这样就可以继续保持与服务器的通信了。

9.6.3 Get

https 的 s 代表的安全（secure），只是指通信通道的安全，http 和 https 的通信过程是一样的，都简单的支持八项基本操作：POST、GET、PUT、DELETE、HEAD、CONNECT、TRACE 和 OPTIONS，但浏览器通常只支持 GET、HEAD、POST、PUT，甚至也很少支持 DELETE。

使用默认客户机的 http.Get 或者我们自己的 c.Get 的例子，上面已经给出。并且可以看到，Get 不只可以读回服务器提供的内容，还可以间接地通过指定 Get 的 URL，传递简单的内容给服务器。更进一步，我们使用 url 包来包装复杂些的内容。例如：

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "net/url"
)

func main() {
    host := "http://localhost:1234/"
    user := url.Values{
        "Name": {"Megan Fox"},
        "Sex":  {"female"},
    }

    q := host + "?" + user.Encode()
    fmt.Println(q)
    r, err := http.Get(q)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer r.Body.Close()
    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("%s", body)
}
```

url.Values 用来组织一个映射，从字符串的键对应到字符串切片的值。q 是 Query 查询，即 URL 中问号后面的部分，需要用 Encode() 方法转换里面的特殊字符。这样组织的 URL 提交给 http.Get，如果使用本章第二个例子的服务器,可以看到 URL 的 Path 不包括 Query 部分。我们自己再写一个服务器，回复读到的 Query：

```
package main

import (
	"fmt"
	"net/http"
)

func hi(w http.ResponseWriter, r *http.Request) {
	fmt.Fprintf(w, "RawQuery: %s\n", r.URL.RawQuery)
	fmt.Fprintf(w, "Name: %s\n", r.URL.Query()["Name"])
}
func main() {
	http.HandleFunc("/", hi)
	http.ListenAndServe(":1234", nil)
}
```

输出：

```
http://localhost:1234/?Sex=female&Name=Megan+Fox
RawQuery: Sex=female&Name=Megan+Fox
Name: [Megan Fox]
```

注意，指向 url 的 Values 是个映射，所以每次 Encode 时，得到的 Query 的顺序是不同的。

9.6.4 Post

Get 可以完成双向通信，为什么还要 Post？

按 HTML 规范的说法，Get 是用来“读”服务器的，放在 URL 中

提交的 Query 只是参数。因此 HTML 规范建议服务器不要用它们来“写”操作。例如，不要用 Get 的参数更新数据库，“写”内容时，最好用 Post。

对于 HTML 的表单，使用 Get 和 Post 都可以。例如下面的服务器可以让读者填好表单发送回来。因为服务器用它记录和改变 users 切片，我们用 Post：

```
package main

import (
    "log"
    "net/http"
    "text/template"
    "time"
)

type User struct {
    Name string
    Date time.Time
}

var users []User

var homePage = template.Must(template.New("home").Parse(
    `<html><body>{{range .}}
{{.Name}} dates on {{.Date.Format "3:04pm, 2 Jan"}}<br />
{{end}}

<form action="/post" method="post">
姓名:  <input type='text' name='Name' /><br />
<input type='submit' value='提交' />
</form></body></html>
`))

func home(w http.ResponseWriter, _ *http.Request) {
    if err := homePage.Execute(w, users); err != nil {
        log.Printf("%v", err)
```

```
        }
    }

    func post(w http.ResponseWriter, r *http.Request) {
        if err := r.ParseForm(); err != nil {

        w.WriteHeader(http.StatusInternalServerError)
            log.Printf("%v", err)
            return
        }
        users = append(users, User{
            Name: r.FormValue("Name"),
            Date: time.Now(),
        })

        http.Redirect(w, r, "/", http.StatusFound)
    }

    func main() {
        http.HandleFunc("/", home)
        http.HandleFunc("/post", post)
        log.Println("http://localhost:1234")
        log.Fatalln(http.ListenAndServe(":1234", nil))
    }
```

多打开几个浏览器窗口访问http://localhost:1234，提交，可以看到服务器集中记录了所有提交的内容。

HTML 的 `form` 表单的 `action` 是`"/post"`，刚好是服务器注册的 `post` 处理函数。它用请求 `r` 的 `ParseForm` 方法，把表单通过 `Post` 或者 `Get` 提交的内容，拆分成 `url.Values` 类型的映射。之后，就可以用 `r.FormValue` 直接得到表单某个项的值了。最后，`http.Redirect` 回复浏览器，让它自动重新访问`"/"`主页，这样就可以看到由 `template` 整理的提交历史记录了。

9.6.5 Cookie

如果我们希望分开记录，使不同的客户端用户只看到自己的力量，就需要一种方法来跟踪某个用户的多次请求。Cookie 小甜饼是合适的选择。

当用户访问一个服务器的 URL 时，服务器可以把一个字符串放在 http 头的"Set-Cookie"项里。用户再次访问同样的 URL 时，浏览器会自动把"Set-Cookie"的字符串放在请求里，发送给服务器。这样，只要保证不同的用户有不同的 Cookie，就可以分开每个用户的记录了：

```
package main

import (
    "log"
    "net/http"
    "text/template"
    "time"
)

type User struct {
    Name string
    Date time.Time
}

var users = make(map[string][]User)

var homePage =
template.Must(template.New("home").Parse(
    `<html><body>{{range .}}
{{.Name}} dates on {{.Date.Format "3:04pm, 2 Jan"}}<br />
{{end}}

<form action="/post" method="post">
姓名： <input type='text' name='Name' /><br />
<input type='submit' value='提交' />
```

```
</form></body></html>
`))

func home(w http.ResponseWriter, r *http.Request) {
    var us []User
    cookie, err := r.Cookie("user")
    if err == http.ErrNoCookie {
        v := time.Now().Format(time.RFC3339)
        http.SetCookie(w, &http.Cookie{
            Name:  "user",
            Value: v,
        })
        log.Printf("New user: %s", v)
    } else {
        us = users[cookie.Value]
        log.Printf("Return user: %s", cookie.Value)
    }
    if err := homePage.Execute(w, us); err != nil {
        log.Printf("%v", err)
    }
}

func post(w http.ResponseWriter, r *http.Request) {
    cookie, err := r.Cookie("user")
    if err == http.ErrNoCookie {

    w.WriteHeader(http.StatusInternalServerError)
        log.Printf("%v", err)
        return
    }
    if err := r.ParseForm(); err != nil {

    w.WriteHeader(http.StatusInternalServerError)
        log.Printf("%v", err)
        return
    }
   users[cookie.Value] = append(users[cookie.Value],
User{
```

```
            Name: r.FormValue("Name"),
            Date: time.Now(),
        })

        http.Redirect(w, r, "/", http.StatusFound)
    }

    func main() {
        http.HandleFunc("/", home)
        http.HandleFunc("/post", post)
        log.Println("http://localhost:1234")
        log.Fatalln(http.ListenAndServe(":1234", nil))
    }
```

在 home 函数里，用户第一次访问时，Cookie 的 user 一项应该是不存在的。因此，我们可以从服务器当前时间得到一个唯一的字符串，作为用户跟踪，用 http.SetCookie 放入 w，随表单一起回复给浏览器。

提交表单时，Cookie 从浏览器发送至服务器的 post 函数。此时不可能没有 user 这项，所以我们直接从这项的值，也就是 cookie.Value，得到此用户的历史记录。我们已经把 users 从单纯的切片改为从 Cookie 字符串到切片的映射了。

测试时我们要使用两个浏览器，而不能是一个浏览器的两个窗口。由此也可得知，Cookie 保存在浏览器中是不安全的，可以被其他窗口的服务器服务偷偷取走分析，从而得知你的一系列私人信息。和 https 同理，加密的 Cookie 要保险许多，但这也要取决于服务器对 Cookie 的正确使用。我们就不多讲了。

WWW（World Wide Web，万维网）也被称为 Wild Wild West（荒蛮西域），是机遇遍地又处处陷阱的地方。大胆而谨慎者靠着些许好运气，可以从雀鸟变为凤凰；冒失无知者，不需太多的厄运，都可能顷刻间树倒鸟兽散。Go 语言和 http 包，只是提供了一个好用的工具，读者能做多大走多远，还是要自己努力。Go.od Luck!

9.7 编码包

RAM 只说自己是可以随处读写的记忆体，但没提自己很危险的一面——如果电没了，就什么都没了。要把易失内存的数据固化下来，例如保存到文件里，或者发送到其他的机器里，就需要一种方式把数据在内存中的内部结构，转换成一种外部结构，这就需要编码和解码。就好比头脑中的想法，要写下来，要能传播出去，才是知识。而文字，就是一种思维的编解码方式。Go 最常用的是 gob 和 json 格式。我们逐一审视。

9.7.1 gob

gob 的意思是碎末。Go 的 gob 包就管理这些二进制碎末的编码发送和解码接收。通常可以用在 RPC 远端系统调用的时候，例如 Go 的 rpc 包。此处，我们只通过简单的例子，介绍 gob 最简单的用法。我们先搭个架子：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    name := "test.gob"
    file, err := os.OpenFile(name, os.O_RDWR|os.
O_CREATE, 0666)
    defer file.Close()
    if err != nil {
        fmt.Println(err)
    }
}
```

os.OpenFile 使用 os 包的 OpenFile 函数，os.O_RDWR 和 os.O_CREATE 是 os 包的两个常量，共同指定打开 name 文件用来读写，并且在没有此文件时自动创建。0666 是个八进制的常量，在 Unix 里代表此文件只能由创建者和同组的人读写。os.OpenFile 返回两个值，第一个值是一个地址，存放着此文件的工作状态，我们用 file 变量代表。第二个是可能的出错值，赋值给 err。不管有没有错，main 函数结束，程序结束，结束前 defer 登记的函数 file.Close 执行，然后关闭文件。

要固化保存些什么呢？像π近乎 3.14 那种常识，大概不需特意保存。同样像“不患人之不己知，患其不能也”，也不能因为是孔子说的就具有特别的指导意义，要记下来强制每个人一字不差地记诵。这种话也是常识，犯不着固化。我们就跟踪一下各国的 GDP 吧，这东西太重要了，时刻记在脑袋里不安全，还是得写下来。怎么把映射这东西写到文件里呢？先别想 XML 那种通用格式，那太麻烦，不到万不得已要和其他语言沟通时不用。Go 的 gob 包是最直接最有力的工具：

```
package main

import (
    "encoding/gob"
    "fmt"
    "os"
)

func main() {
    GDP := map[string]float64{
        "USA":   14.58,
        "China": 5.92,
        "Japan": 5.45,
    }

    name := "test.gob"
```

```
        file, err := os.OpenFile(name, os.O_RDWR|os.
O_CREATE, 0666)
        defer file.Close()
        if err != nil {
            fmt.Println(err)
        }
        enc := gob.NewEncoder(file)
        if err := enc.Encode(GDP); err != nil {
            fmt.Println("Cannot encode:", err)
            return
        }

    }
```

固化就这么简单！检查我们的 test.gob 已经不再是 0 字节，而是存放了 GDP 的类型和值。我们写另一个程序读回 GDP：

```
    package main
    import (
        "encoding/gob"
        "fmt"
        "os"
    )
    var GDP map[string]float64
    func main() {
        name := "test.gob"
        file, err := os.Open(name)
        defer file.Close()
        if err != nil {
            fmt.Println("Cannot open:", err)
            return
        }
        dec := gob.NewDecoder(file)
        if err := dec.Decode(&GDP); err != nil {
            fmt.Println("Cannot decode:", err)
            return
        }
        fmt.Println(GDP)
    }
```

也是这么简单！gob 甚至能帮我们自动创建一个 GDP。注意&GDP 是取 GDP 变量值的地址，尽管在创建之前，GDP 的值是 nil，但它也是有地址的。不信，自己加行 fmt.Printf("%x\n", &GDP)，看看它的地址是什么值。

Gob 包不但能精炼固化各种 Go 的数据结构，还记录下类型信息供日后扩展。虽然指针类型的变量不能编码，但它们指向的变量是可以的。也可以理解为，间接的指针树被压扁为一个紧凑的结构。因为类型信息写在值信息的前面，解码时就可以自动根据类型，恢复后面的值到正确的内存变量里。

也因为类型和值一起编码，编解码的类型不必相同，只要兼容就可以了。例如可以把 int32 的变量编码，再解码到一个 int64 的变量中。一个编码的结构，其项可以不出现在解码的变量里，其值自动忽略；而出现在解码变量中的项，如果未曾编码，其项也被自动忽略，也就是不去赋值。如果双方的项名相同，其类型必须兼容。编解码会自动处理指针类型。

例如 struct{A, B int }可以自动解码到下列类型的结构体变量中：

```
struct { A, B int }
*struct { A, B int }
struct { *A, **B int }
struct { A, B int64 }
struct { B, A int }
struct { A, B, C int }      // C 被忽略
struct { B int }            // A 被忽略
struct { B, C int }         // A 和 C 被忽略
```

但下列解码会出错：

```
struct { A int; B uint }  // A、B 类型不兼容
struct { A int; B float } // A、B 类型不兼容
struct { }                // 没有相同成员
struct { C, D int }       // 没有相同成员
```

和 Go 的常量一样，gob 的整数类型只分为带符号和无符号的任意长度的类型，而不分 int8、int16 等。浮点数只有 float64。解码时的变量必须能装下编码的数值，而且整数符号必须相同。否则会出错。

结构、数组和切片可以编解码。并且字符串和字节数组使用特殊高效的表示。解码的切片变量如果容量不够，会自动重新分配。从结果切片的长度，可以得到解码单元的个数。

只有函数和程道不可以编解码。具体对每个类型的支持，以及内部表示，可以参考 godoc encoding/gob

9.7.2 json

JSON（JavaScript Object Notation，JavaScript 对象表示法），是一种简单的文字编码和数据交换格式，非常广泛地用于因特网服务器和浏览器的通信。因为，它是浏览器内部支持的编解码格式之一，也远比 XML 容易。

json 包的基本用法，几乎和 gob 包的用法一致。我们甚至只需对上述 gob 的编解码示例程序做个全局替换，把 gob 全部换成 json 就可以了。

但具体到每个类型，json 和 gob 的处理是不同的。

(1) Go 的布尔类型对应 JSON 的 boolean。

(2) 整数和浮点数都对应 JSON 的 number。

(3) 字符串对应 JSON 的 string，其中非法的 UTF-8 序列都对应 U+FFFD。两个尖括号<>对应到"\u003c"和"\u003e"，以避免某些浏览器把 JSON 当成 HTML。

(4) 数组和切片对应 JSON 的 array。

(5) 字节数组对应 base64 的字符串。

(6) 结构对应 JSON 的 object。除了项标签 tag 是"-"，或者项值为空并且 tag 有"omitempty"之外，所有项对应 object 的一个成员。

(7) 映射对应 JSON 的 object。映射键必须是字符串，object 的 key 直接作为映射的键。

(8) 指针值是指针指向的值，nil 对应 JSON 的 null。

(9) 界面值是界面变量的具体值。nil 对应 JSON 的 null。

(10) 程道、复数、函数值不能使用 JSON。

第 6 项的结构项需要特别说明一下：空值是指 false、0、nil 和所有零长度的数组、切片、映射和字符串。JSON 的 object 的 key 是项名，但也可以使用项的标签——"json"后跟一些选项。例如：

```
// 忽略此项
Field int `json:"-"`
// 项在 JSON 中的 key 是"myName"
Field int `json:"myName"`
// 项在 JSON 中的 key 是"myName"。值为零则忽略
Field int `json:"myName,omitempty"`
// 项在 JSON 中的 key 是"Field"。值为零则忽略
Field int `json:",omitempty"`
```

附录A

Go 的安装和使用

Go 语言的工具和包都使用 BSD 授权来共享源代码。读者可以从 http://code.google.com/p/go选择所需的安装包下载。

Windows 下可以运行 msi 的文件并按照提示安装。默认的安装路径是 c:\Go。而 c:\Go\bin 应该会自动加入 PATH 环境变量。

在 Mac OS X 下可以执行 pkg 文件并按照提示安装。默认的安装路径是/usr/local/go。而/usr/local/go/bin 会自动加入 PATH 环境变量。

Linux 和 FreeBSD 比较复杂，需要在命令行执行下列步骤。

(1) `sudo rm -r /usr/local/go`。

(2) `sudo tar -C /usr/local -xzf go.*.tar.gz`。

(3) 把 `export PATH=$PATH:/usr/local/go/bin` 加入`$HOME/.profile`。

命令行工具

Rob Pike 曾提到 Go 的最初想法产生于等待谷歌的大型服务器进行编译的时候，因而希望设计一种语言，能让谷歌自己写的大型软件迅速编译完毕。从一开始，Go 就需要一种方式来清楚地表示各种代码库的相互关系，所以就有了 package 包和明确的 import 导入语句，而且人们

可以使用任意的格式描述需要导入的代码，因此导入的路径是个字符串。

从一开始，Go 就确定只需利用自己的源代码，而不需 makefile 来编译程序。也就是说，程序员除了明确导入之外，不需在 makefile 或者其他的工具中指明代码和包的相互依赖关系，也就是不需任何配置，就可以编译一个完整的 Go 程序或包。

这种简单不需刻意配置关系的环境，是靠约定完成的。但约定只在得到遵循时才有效。在其他的语言里，每一个包都可以规定自己安装的目录、名称、使用的工具等。这可以理解为每家都可以有自己的家规，但当需要构成更大规模的组织时，就必须遵守同样的约定，也就是习俗。自觉遵守习俗，是使 Go 易读易用的重要环节。

Go 的 `go` 命令提供的工具需要每个包都遵照如下约定。

(1) `GOPATH` 环境变量提供顶层的目录，使 `go` 工具能在 GOPATH/src 下找到源代码，能把编译的包放入 GOPATH/pkg，能把编译链接的可执行文件放入 GOPATH/bin。例如，如果 `export GOPATH=$HOME`，则 Go 官方包之外的源代码都在 ~/src 下，而得到的可执行文件存放在 ~/bin 中，如果我们的`$PATH` 刚好包括 ~/bin，则可以立即执行得到的程序。如果没有设置 `GOPATH`，则使用 `GOROOT`。

(2) 对于从 bitbucket、github、googlecode 和 launchpad 得到的导入包，导入路径是包所在的 URL，但不包括 http://。例如 websocket 包可以从 http://code.google.com/p/go.net 得到，所以使用 `import "code.google.com/p/go.net/websocket"`，可以让 `go` 工具自动下载所需的源代码到 $GOPATH/src/code.google.com/p/go.net/websocket。也可以说，通过一个包所在的 URL，我们可以立即得知所需的 `import`；而且通过$GOPATH/src 里一个包所在的目录名，我们也可以立即确定如何 `import`，而无需任何的配置。

(3) 每个包要使用单独的目录，每个目录只能对应一个包。这样，使

用普通的文件目录管理，就可以组织一个包和它包括的文件，而不需要特意用配置文件指定它们的关系。

(4) 编译一个包所需的信息只能从源代码得到。go 工具不提供命令行选项用来指定文件和包所在的位置。

我们可以使用命令行程序 go，来完成所需的下载、编译、安装等工作。go 可以使用命令参数如下。

表A-1　go tool表

build	包与依赖包的编译
clean	删除目标文件
doc	godoc包文件
fix	go tool fix包
fmt	gofmt包文件
get	包与依赖包的下载和安装
install	包与依赖包的编译与安装
list	包的列表
run	Go程序的编译与运行
test	包的测试
tool	运行指定的go tool
version	显示Go的版本
vet	go tool vet包

使用 go help 可以看到命令简介。另外还可以对以下参数执行 go help，如表 A-2 所示。

表A-2　可执行go help命令的参数

Gopath	GO PATH环境变量
importpath	导入路径描述
remote	远端导入路径句法
Testflag	测试标志描述
testfunc	测试函数描述

具体每个命令的参数总结如表 A-3 所示。

表A-3　命令的参数

go run 文件 [参数]
编译并运行 gofiles 构成的 main 包
-a 强制已经更新的包也重新编译
-n 显示但不执行命令
-x 执行并显示命令
go get 导入路径
下载并安装导入路径的包及其依赖的包
-a 强制已经更新的包也重新编译
-n 显示但不执行命令
-x 执行并显示命令
-v 在编译时同时显示包名
-p n 指定并行编译数，默认是 CPU 的核数
-d 下载但不安装
-fix 下载并执行安装
-u 通过网络更新包和它们依赖的包。默认是不更新已经下载的包
go install 导入路径
编译并安装导入路径的包及其依赖
-a 强制已经更新的包也重新编译
-n 显示但不执行命令
-x 执行并显示命令
-v 在编译时同时显示包名
-p n 指定并行编译数，默认是 CPU 的核数
go build 导入路径或者文件
编译导入路径或者文件指定的包，以及它们依赖的包，但不安装
如果给出的是一系列的.go 文件，则把它们作为同一个包的源文件编译

（续）

如果给出的是一个 main 包，则生成可执行文件（默认为 a.out） 否则编译但不生成文件，只用来确认包可以编译 -a 强制已经更新的包也重新编译 -n 显示但不执行命令 -x 执行并显示命令 -v 在编译时同时显示包名 -p n 指定并行编译数，默认是 CPU 的核数 -o file 指定生成文件名，但不可用于多个包
go clean 导入路径
从包源文件目录删除目标文件。go 命令会在临时目录下生成编译的目标文件，所以 go clean 只是用来删除其他工具或者手动编译产生的目标文件 下列 Makefile 遗留的旧文件会被删除： _obj/ 目标目录 _test/ 测试目录 _testmain.go 测试记录 test.out 测试记录 build.out 测试记录 *.[568ao] 目标文件 下面的 DIR 是目录的最后一项，MAINFILE 是编译未包括的所有 Go 源文件基名： DIR(.exe) DIR.test(.ext) MAINFILE(.exe) -i 删除 go install 生成安装的对应库和代码文件 -r 递归执行 -n 显示但不执行命令 -x 执行并显示命令
go list 导入路径
逐行显示导入路径的包。默认的格式例如： code.google.com/p/google-api-go-client/books/v1

（续）

code.google.com/p/goauth2/oauth

code.google.com/p/sqlite

-f 使用 template 模板包显示下列结构，而上述默认格式就是-f:
'{{.ImportPath}}'

```
type Package struct {
    Name          string // 包名
    Doc           string // 包注释
    ImportPath string // 导入路径
    Dir           string // 包源文件目录
    Version       string // 安装版本
    Stale         bool   // 供 go install
    GoFiles          []string // .go 文件
    TestGoFiles      []string // _test.go 内部文件
    XTestGoFiles []string // _test.go 外部文件
    CFiles           []string // .c 文件
    HFiles           []string // .h 文件
    SFiles           []string // .s 文件
    CgoFiles         []string // .go 文件有 import "C"
    Imports  []string // 此包导入路径
    Deps     []string // 所有依赖
    Incomplete bool               // 错误
    Error *PackageError           // 无法加载包
    DepsErrors []*PackageError // 无法加载依赖
}
```

-json 把上述结构使用 JSON 格式显示

-e 改变错误显示的方式

---

go test 导入路径

自动测试导入路径的包，并显示下例的格式：

```
ok   archive/tar   0.011s
FAIL archive/zip   0.022s
ok   compress/gzip 0.033s
```

后跟测试失败包的详细信息。

（续）

go test 会把符合 "*_test.go"样式的文件重新编译到包里。这些额外的文件包括测试函数、基准函数和示例函数
默认会编译和测试当前目录下的源文件 -file a.go 可以逐个指定只测试这些文件 -c 编译但不执行测试 -i 安装但不执行测试 -x 执行并显示命令 -p n 指定并行编译数，默认是 CPU 的核数
go doc 导入路径
对导入路径的包执行 godoc
go fix 导入路径
对导入路径的包执行 go tool fix
go fmt 导入路径
对导入路径的包执行 gofmt -l -w
go vet 导入路径
对导入路径的包执行 go tool vet
go version
显示 Go 的版本

附录B

EBNF

自 20 世纪 60 年代 Backus 和 Naur 使用 BNF 描述 ALGOL60 的语法以来，所有编程语言的语法都要用某种 BNF 的扩展来精确描述。Go 也不例外。Go 语言的英文规范使用一种简单的 EBNF 来表示语法，但由于 EBNF 同时使用 Go 语言的括号和|符号来表示 EBNF 自身的语法，使得 Go 语言的词法符号要用引号括起，造成识别上的困扰。因此，本书试图引入 Go 语言不会用到的符号来作为 EBNF 的符号，从而使 Go 语言的符号可以不加引号直接使用。EBNF 的规定如下：

•　　表示或者

≤≥　　表示优先

‹›　　表示选项 0 或 1 次

《》　　表示重复 0 或多次

默认的优先级是从左至右，但使用≤≥可以提高优先级。例如:

导入条款 = `import` ≤导入规范 • （《导入规范;》）≥

读作：导入条款定义为，`import` 后跟一个导入规范或小括号括起的以分号结尾的导入规范。具体规范见表 B-1。

表 B-1　Go 语言 EBNF 表

源文件 = 包条款；《导入条款；》《顶层声明；》
包条款 = `package` 包名

PDG

（续）

包名 = 标识符

导入条款 = import ≤导入规范 • (《导入规范;》)≥

导入规范 = ‹. • 包名 › 导入路径

导入路径 = 字符串

顶层声明 = 声明 • 函数声明 • 方法声明

声明 = 常量声明 • 变量声明 • 类型声明

常量声明 = const ≤常量规范 • (《常量规范;》)≥

常量规范 = 标识符列 ‹ ‹类型› = 表达式列 ›

变量声明 = var ≤变量规范 • (《变量规范;》)≥

变量规范 = 标识符列 类型 ‹=表达式列›

类型声明 = type ≤类型规范 • (《类型规范;》)≥

类型规范 = 标识符 类型

标识符列 = 标识符 《, 标识符》

表达式列 = 表达式 《, 表达式》

表达式 = 一元表达式 • 表达式 二元符 一元表达式

一元表达式 = 基本表达式 • 一元符 一元表达式

一元符 = + • - • ! • ^ • * • & • <-

二元符 = ‖ • && • 关系符 • 加符 • 乘符

关系符 = == • != • < • <= • > • >=

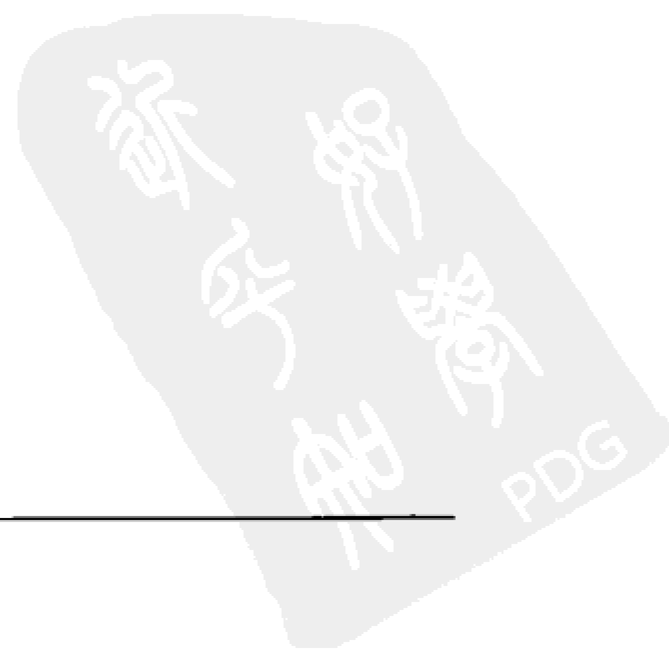
加符 = + • - • | • ^

乘符 = * • / • % • << • >> • & • &^

基本表达式 = 操作数 • 转换 • 内置调用 •
基本表达式 选择符 •
基本表达式 下标 •
基本表达式 切片 •
基本表达式 类型断言 •
基本表达式 调用 •

内置调用 = 标识符(‹内置参数‹,››)

内置参数 = 类型 ‹, 表达式列› • 表达式列



（续）

转换 = 类型(表达式)

选择符 = .标识符

下标 = [表达式]

切片 = [‹表达式› : ‹表达式›]

类型断言 = .(类型)

调用 = (‹参数列‹,››)

参数列 = 表达式列‹...›

操作数 = 字面 • 入围表述 • 方法表述 • (表达式)

字面 = 基本字面 • 组合字面 • 函数字面

基本字面 = 整数 • 浮点数 • 虚数 • 字符 • 字符串

入围表述 = 包名.标识符

组合字面 = 字面类型 字面值

字面类型 = 结构类型 • 数组类型 • [...]基本类型 •
切片类型 • 映射类型 • 类型名

字面值 = { ‹单元列‹,›› }

单元列 = 单元 《,单元》

单元 = ‹键:›值

键 = 域名 • 单元下标

域名 = 标识符

单元下标 = 表达式

值 = 表达式 • 字面值

方法表述 = 接受者类型.方法名

接受者类型 = 类型名 • (*类型名)

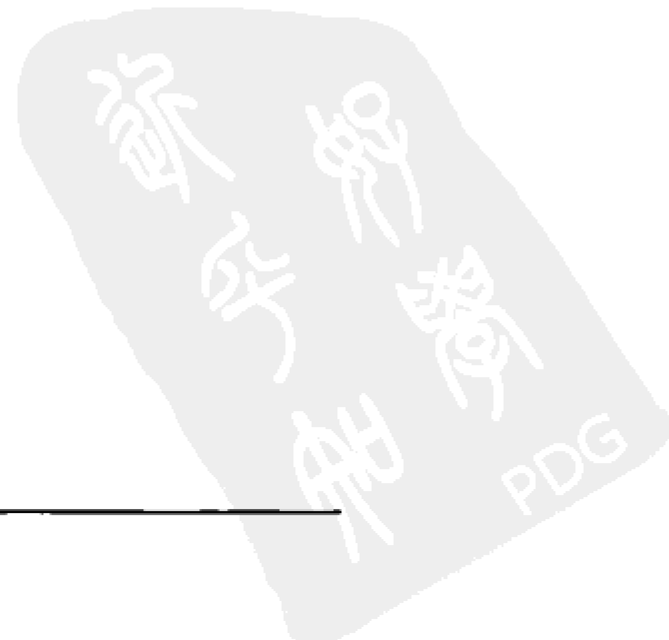
方法名 = 标识符

类型 = 类型名 • 类型字面 • (类型)

类型名 = 入围表述

类型字面 = 数组类型 • 切片类型 • 结构类型 • 指针类型 •
函数类型 • 界面类型 • 映射类型 • 程道类型

数组类型 = [数组长度]单元类型



（续）

数组长度 = 表达式

单元类型 = 类型

切片类型 = []单元类型

结构类型 = struct { 《域声明;》 }

域声明 = ≤标识符列 类型 • 无名域≥ ‹标签›

无名域 = ‹*›类型名

标签 = 字符串

指针类型 = *基本类型

基本类型 = 类型

函数类型 = func 签名

签名 = 参数 ‹结果›

结果 = 参数 • 类型

参数 = (‹参数列‹,››)

参数列 = 参数声明 《,参数声明》

参数声明 = ‹标识符列› ‹...› 类型

界面类型 = interface { 《方法规范;》 }

方法规范 = 方法名 签名 • 界面类型名

界面类型名 = 类型名

映射类型 = map[键类型] 基本类型

键类型 = 类型

程道类型 = ≤chan ‹<-› • <-chan≥ 基本类型

八进制数 = 0 ~ 7

十六进制数 = 0 ~ 9 • a ~ f • A ~ F

十进制数 = 0 ~ 9

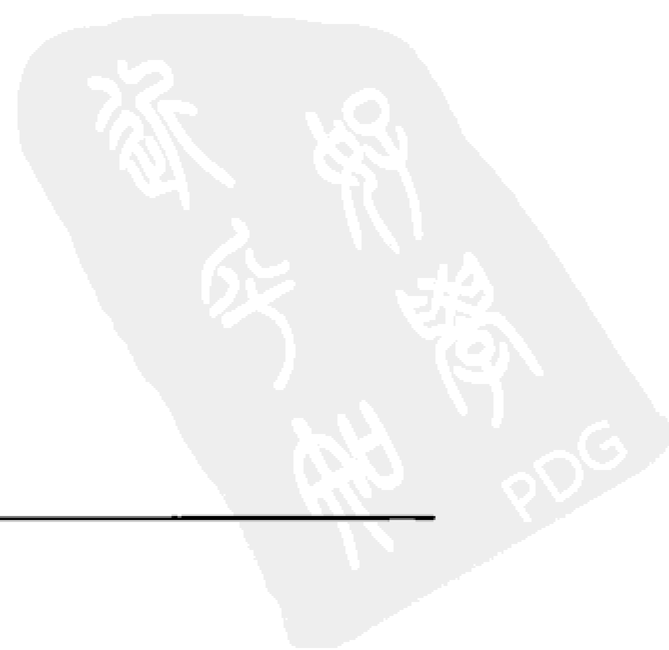
字母 = Unicode 字母 • _

Unicode 字母 = 所有 Unicode 标准规定的字母

Unicode 数字 = 所有 Unicode 标准规定的十进制数

Unicode 字符 = 除换行符之外到所有 Unicode 码值

换行符 = Unicode 码值 U+000A



（续）

标识符 = 字母 《字母 • Unicode 数字》

整数 = 十进制字面 • 八进制字面 • 十六进制字面

十进制字面 = ≤1~9≥ 《十进制数》

八进制字面 = 0 《八进制数》

十六进制字面 = 0 ≤x•X≥ 十六进制数 《十六进制数》

浮点数 = 数字.‹数字› ‹指数› •

数字 指数 •

.数字 ‹指数›

数字 = 十进制数 《十进制数》

指数 = ≤ e•E ≥ ‹ + • - › 数字

虚数 = ≤数字 • 浮点数≥ i

字符 = ' ≤Unicode 值 • 字节值≥ '

Unicode 值 = unicode 字符 • 小 u 值 • 大 u 值 • 转义字符

字节值 = 八进制值 • 十六进制值

八进制值 = \ 八进制数 八进制数 八进制数

十六进制值 = \x hex 数 hex 数

小 u 值 = \u hex 数 hex 数 hex 数 hex 数

大 u 值 = \U hex 数 hex 数 hex 数 hex 数

hex 数 hex 数 hex 数 hex 数

转义字符 = \≤a•b•f•n•r•t•v•\•'•"≥

字符串 = 生字符串 • 熟字符串

生字符串 = ` 《unicode 字符 • 换行符》 `

熟字符串 = " 《Unicode 值 • 字节值》 "

语句 = 声明 • 标号语句 • 简单语句 • go 语句 •

return 语句 • break 语句 • continue 语句 •

goto 语句 • fallthrough 语句 • 块 • if 语句 •

switch 语句 • select 语句 • for 语句 • defer 语句

简单语句 = 空语句 • 表达式语句 • 发送语句 •

增值语句 • 赋值语句

（续）

空语句 =

表达式语句 = 表达式

增值语句 = 表达式 ≤ ++ • -- ≥

赋值语句 = 表达式列 赋值符 表达式列

赋值符 = ‹加符 • 乘符› =

标号语句 = 标号: 语句

标号 = 标识符

发送语句 = 程道 <– 表达式

程道 = 表达式

if 语句 = if ‹简单语句;› 表达式 块

　　　　‹else› ≤if 语句 • 块≥

switch 语句 = 表达式 switch • 类型 switch

表达式 switch = switch ‹简单语句;› ‹表达式› {

　　《表达式分支条》 }

表达式分支条 = 表达式分支: 《语句;》

表达式分支 = case 表达式列 • default

类型 switch=switch‹简单语句;› 类型 switch 卫{类型 switch 条}

类型 switch 卫 = ‹标识符 := › 基本表达式.(type)

类型 switch 条 = 类型 switch: 《语句;》

类型 switch = case 类型列 • default

类型列 = 类型 《, 类型》

for 语句 = for ‹条件 • for 条 • range 条› 块

条件 = 表达式

for 条 = ‹初始语句› ; ‹条件› ; ‹增量语句›

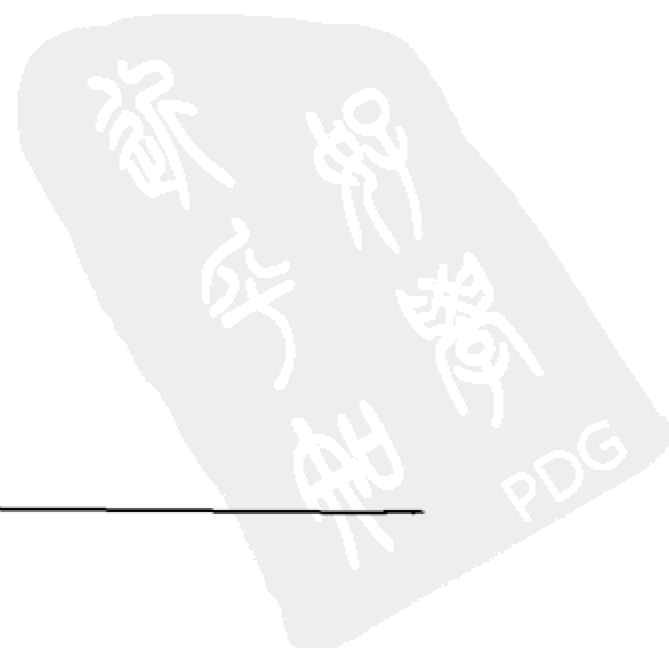
初始语句 = 简单语句

增量语句 = 简单语句

range 条 = 表达式 ‹, 表达式› ≤ = • := ≥ range 表达式

go 语句 = go 表达式

select 语句 = select { 通信条 }



（续）

通信条 = 通信分支: 《语句;》
通信分支 = case ⩽ 发送语句 • 接收语句 ⩾ • default
接收语句 = ‹表达式 ‹, 表达式› ⩽ = • := ⩾› 接收表达式
接收表达式 = 表达式
return 语句 = return 表达式列
break 语句 = break ‹标号›
continue 语句 = continue ‹标号›
goto 语句 = goto 标号
fallthrough 语句 = fallthrough
defer 语句 = defer 表达式

附录C

中英术语对照表

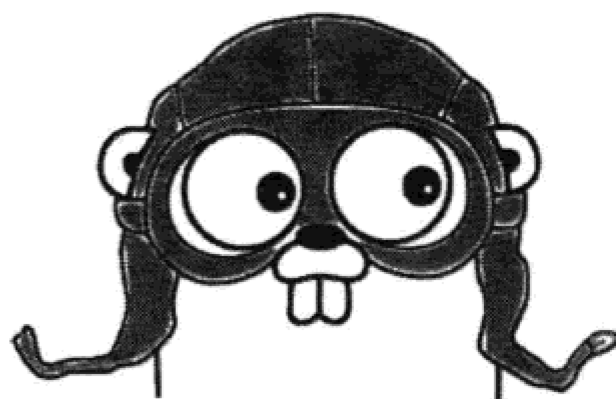
英　文	中　文	注　释
array	数组	相同类型单元连续排列的数据结构
argument	实参	调用函数时的实际参数值
assignment	赋值	替换变量的值
bit	位	计算机处理的最小单位
bool	布尔	代表真假的类型
break	跳出	跳出 `for` 循环
byte	字节	由连续 8 位构成，Go 数据结构的最小单位
case	分支	`switch` 语句不同条件下的执行体
call	调用	传递参数，执行函数，得到返回值
channel	程道	不同去程之间传递信息的数据结构
character	字符	对应 Unicode 的码值
codepoint	码值	Unicode 字符的值
comment	注释	源代码中只供程序员参考的文字
complex	复数	由浮点数实部和虚部的构成的类型
conversion	转换	不同类型值的转换
const	常量	程序编译时确定的值
default	默认	`switch` 语句所有条件都不满足时的执行体
defer	压后	登记函数结束时才会自动执行的函数
dispatch	调度	动态的方法函数调用

（续）

英　文	中　文	注　释
error	错误	预知的错误
fallthrough	接续	switch 接续执行下一个分支
field	项	struct 结构中的一个变量
float	浮点数	IEEE 754 规定的小数类型
for	循环	只要条件满足就持续执行
func	函数	Go 执行调用的单位
goroutine	去程	可并发执行的函数
identifier	标识符	编译器识别的字母数字组合
import	导入	使导入包中的公开值和函数可用
int	整型	整数类型
interface	界面	不同类型具备同样的方法
iota	埃塔	编译时每行自动加 1
key	键	用于映射 查找的值
keyword	关键字	编译器保留使用的标识符
literal	字面量	值在源代码中的文字表示
map	映射	从键直接得到值的数据结构
method	方法	具备接受类型值的函数
panic	派错	运行时出错
parameter	参数	定义函数时的变量
package	包	Go 代码的包装单位
pointer	指针	地址类型，指向给定类型的值
range	遍历	用在 for 循环中逐一得到每个值
recover	恢复	处理 panic 的函数
reflect	反射	运行时得到变量类型值
return	返回	函数结束
rune	日耳	字符的内部表示类型，等同于 int32
scope	作用域	变量名称起作用的范围
select	选择	选择一个没有阻塞的程道

（续）

英　　文	中　　文	注　　释
slice	切片	用于操作数组的类型
statement	语句	Go的句法单位
string	字符串	UTF-8编码的Unicode顺序字符常量
struct	结构	各种类型组合的数据结构
switch	切换	切换到满足条件的分支执行
type	类型	值的类型供编译时检查和分配内存
var	变量	变量声明，运行时可以改变的值



Go语言·云动力

Unix之父 Ken Thompson 接受 *Dr.Dobb* 专访，谈到 Go 语言时的一段对话。

K：刚开始时纯粹是研究。我们仨凑到一块儿决定一起恨 C++。

D：我想好多人站在您这边儿。

K：C++太复杂了。如果可以回头，如果我们有想到过，那时我们应该能做出个面向对象版本的 C。

D：您是说您会去做？

K：但我们不是面向对象的布道者。一开始我们就想到，必须一起讨论 Go 语言的每个特征，这样才不会毫无理由地把不相干的垃圾带进来。

D：的确，Go 真的是很苗条的语言。

摘自：http://drdobbs.com/open-source/229502480