

学习Go 语言

针对即

将发布的Go-1 进行了更新

==GO



<http://golang.org>

作者：
Miek Gieben
译者：
邢兴

感谢：
Go 作者
Google
Go Nuts 邮件列表



This work is licensed under the *Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License*.

Miek Gieben – ©2010 - 2012
邢兴 – ©2011 , 2012

本作品依照署名-非商业性使用-相同方式共享3.0 Unported许可证发布。访问<http://creativecommons.org/licenses/by-nc-sa/3.0/> 查看该许可证副本，或写信到Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA。

本书所有实例代码依此方式放入公共领域。

让我们开始学习Go吧。(0.5)

更新至即将发布的Go-1版本

Contents

读者	vi
1 简介	1
官方文档	1
获得 Go	2
在 Windows 下获得 Go	2
前身	4
练习	4
答案	5
2 基础	6
Hello World	7
编译和运行代码	7
变量、类型和保留字	8
运算符和内建函数	12
Go 保留字	12
控制结构	13
内建函数	18
array、slices 和 map	19
练习	22
答案	25
3 函数	30
作用域	31
多个返回值	32
命名返回参数	33
延迟的代码	34
变参	35
函数作为值	36
回调	36
恐慌 (Panic) 和恢复 (Recover)	37
练习	38
答案	41
4 包	48
构建一个包	49
标识符	50
包的文档	51
测试包	51
常用的包	53
练习	54
答案	57
5 进阶	60
内存分配	60
定义自己的类型	63
转换	65

练习	66
答案	69
6 接口	72
方法	74
接口名字	75
简短的例子	75
练习	80
答案	81
7 并发	84
更多关于 channel	86
练习	87
答案	89
8 通讯	92
文件和目录	92
命令行参数	94
执行命令	94
网络	95
练习	96
答案	99
A 版权	108
贡献者	108
许可证和版权	109
B 索引	110
C Bibliography	112

List of Figures

1.1 Go 编年史	4
2.1 array 与 slice 对比	20
3.1 一个简单的 LIFO 栈	38
6.1 使用反射去除层次关系	79

List of Code Examples

2.1 Hello world	7
2.2 程序的 Makefile	8
2.3 Declaration with =	8
2.4 Declaration with :=	8
2.5 Familiar types are still distinct	9
2.6 array 和 slice	20

2.7	Simple for loop	25
2.8	For loop with an array	25
2.9	Fizz-Buzz	26
2.10	Strings	27
2.11	Runes in strings	27
2.12	Reverse a string	28
3.1	函数定义	30
3.2	递归函数	31
3.3	局部作用域	31
3.4	全局作用域	31
3.5	当函数调用函数时的作用域	31
3.6	没有 defer	34
3.7	With defer	34
3.8	函数符号	35
3.9	带参数的函数符号	35
3.10	在 defer 中访问返回值	35
3.11	Anonymous function	36
3.12	使用 map 的函数作为值	36
3.13	Go 中的平均值函数	41
3.14	stack.String()	43
3.15	有变参的函数	43
3.16	Fibonacci function in Go	44
3.17	Map 函数	44
3.18	Bubble sort	45
4.1	A small package	48
4.2	Use of the even package	48
4.3	用于包的 Makefile	49
4.4	even 包的测试	52
4.5	包里的 Stack	57
4.6	Push/Pop 测试	57
4.7	包的 Makefile	57
4.8	逆波兰计算器	58
5.1	Use of a pointer	60
5.2	获取指针指向的值	60
5.3	Structures	63
5.4	Go 中更加通用的 map 函数	69
5.5	cat 程序	70
6.1	定义结构和结构的方法	72
6.2	用空接口作为参数的函数	74
6.3	实现接口失败	74
6.4	扩展内建类型错误	75
6.5	扩展非本地类型错误	75
6.6	使用反射自省	78
6.7	反射类型和值	78
6.8	私有成员的反射	79
6.9	公有成员的反射	79
6.10	通用的计算最大值	81
7.1	Go routine 实践	84
7.2	Go routines 和 channel	85
7.3	使用 select	86

7.4	Go 的 channel	89
7.5	添加额外的退出 channel	89
7.6	Go 的斐波那契函数	90
8.1	从文件读取 (无缓冲)	92
8.2	从文件读取 (缓冲)	93
8.3	在 shell 中创建目录	93
8.4	在 Go 中创建目录	93
8.5	Processes in Perl	96
8.8	uniq(1) 的 Perl 实现	97
8.6	Go 中的进程	99
8.7	wc(1) 的 Go 实现	100
8.9	uniq(1) 的 Go 实现	101
8.10	简易 echo 服务器	101
8.11	数字游戏	102

List of Exercises

1	(1) Documentation	4
2	(1) For-loop	22
3	(1) FizzBuzz	22
4	(1) Strings	22
5	(4) Average	23
6	(4) 平均值	38
7	(3) 整数顺序	38
8	(4) 作用域	38
9	(5) 栈	38
10	(5) 变参	38
11	(5) 斐波那契	38
12	(4) Map function	39
13	(3) 最小值和最大值	39
14	(5) 冒泡排序	39
15	(6) 函数返回一个函数	39
16	(2) stack 包	54
17	(7) 计算器	54
18	(4) 指针运算	66
19	(6) 使用 interface 的 map 函数	66
20	(6) 指针	66
21	(6) 链表	67
22	(6) Cat	67
23	(8) 方法调用	67
24	(6) 接口和编译	80
25	(5) 指针和反射	80
26	(7) 接口和 max()	80
27	(4) Channel	87
28	(7) 斐波那契 II	87
29	(8) 进程	96
30	(5) 单词和字母统计	96
31	(4) Uniq	96
32	(9) Quine	97

33	(8) Echo 服务	97
34	(9) 数字游戏	97
35	(8) *Finger 守护进程	97

前言

”Go 是面向对象的语言吗？是也不是。”

FAQ
GO AUTHORS

读者

这是关于来自Google 的Go 语言的简介。目标是为这个新的、革命性的语言提供一个指南。

这本书的目标读者是那些熟悉编程，并且了解多种编程语言，例如C[23]，C++[34]，Perl [25]，Java [24]，Erlang[21]，Scala[4]，Haskell[13]。这不是教你如何编程的书，只是教你如何使用Go。

学习某样新东西，最佳的方式可能是通过建立自己的程序来探索它。因此每章都包含了若干练习（和答案）让你熟悉这个语言。练习标有编号**Q n** ，而 n 是一个数字。在练习编号后面的圆括号中指定了该题的难度。难度范围从0 到9，0 是最简单，而9 最难。其后为了容易索引，提供了一个简短的名字。例如：

Q1. (1) map 函数...

展示了难度等级1、编号**Q1** 的关于map() 函数的问题。相关答案在练习的下一页。答案的顺序和练习一致，而对于以**A n** 开头的问题，对应编号 n 的练习。一些练习没有答案，它们被用星号标识出来。

内容布局

第1 章：简介

提供了关于Go 的简介和历史。同时讨论了如何获得Go 本身的代码。虽然Go 完全可能在Windows 平台上使用，但这里还是假设使用类Unix 环境。

第2 章：基础

讨论了语言中可用的基本类型，变量和控制结构。

第3 章：函数

在第三章中了解了函数，这是Go 程序中的基本部件。

第4 章：包

在第4 章中，会了解在包中整合函数和数据。同时也将了解如何对包编写文档和进行测试。

第5 章：进阶

然后，在第5 章中会看到如何创建自定义的类型。同时也将了解Go 中的内存分配。

第6 章：接口

Go 不支持传统意义上的面向对象。在Go 中接口是核心概念。

第7 章：并发

通过go 关键字，函数可以在不同的例程（叫做goroutines）中执行。通过channel 来完成这些goroutines 之间的通讯。

第8章：通讯

最后一章展示了如何用接口来完成Go 程序的其他部分。如何创建、读取和写入文件。同时也简要了解一下网络。

希望你喜欢这本书，同时也喜欢Go 语言。

Miek Gieben, 2011 – miek@miek.nl 邢兴, 2011 – mikespook@gmail.com

1

简介

“对此感兴趣，并且希望做点什么。”

在为Go 添加复数支持时
KEN THOMPSON

什么是Go？来自于网站：[12]：

Go 编程语言是一个使得程序员更加有效率的开源项目。Go 是有表达力、简洁、清晰和有效率的。它的并行机制使其很容易编写多核和网络应用，而新奇的类型系统允许构建有弹性的模块化程序。Go 编译到机器码非常快速，同时具有便利的垃圾回收和强大的运行时反射。它是快速的、静态类型编译语言，但是感觉上是动态类型的，解释型语言。

Go 是一个年轻的语言，特性仍然在不断增加或删除中。所以当你阅读的时候，可能部分内容已经过时。一些练习的答案可能会随着Go 不断的演化而变成错误的。我们将尽可能让这个文档与最新的Go 发布版本保持一致。通过努力已经建立了“特性检验”代码示例。

本书使用了下面的约定：

- 代码用DejaVu Mono 显示；
- 关键词用DejaVu Mono Bold 显示；
- 注释用DejaVu Mono Italic 显示；
- 代码中额外的标记，`←` 用这种形式展现；
- 使用数字❶对长内容标记——解释会跟随其后；
- 行号在右边展示；
- Shell 示例用% 作为标记；
- 强调的段落会缩进，在左边有竖线。

官方文档

Go 已经有大量的文档。例如Go Tutorial [11] 和Effective Go [6]。网站<http://golang.org/doc/> 是绝佳的起点^a。虽然并不一定要阅读这些文档，但是强烈建议这么做。

Go 用叫做godoc 的Go 程序格式化其文档。你可以用它查阅在线文档。例如，假设我们需要了解关于hash 包的更多信息。可以用命令godoc hash 查阅它。如何创建你自己的包的文档在第4 章中介绍。

在互联网上搜索时，应当使用“golang”这个词来代替原始的“go”。

^a<http://golang.org/doc/> 本身是由Go 程序godoc 提供服务的。

获得Go

Ubuntu 和Debian 都有Go 包在其仓库中，查找“golang”包。但是工作的时候仍然有一些小问题。现在仍然使用源码进行安装。

因此将会从mercurial 中获取代码并且编译。对于其他类Unix 系统，过程类似。

- 首先安装Mercurial （获取hg 命令）。在Ubuntu/Debian/Fedora 需要安装mercurial 包；
- 为了编译Go 需要包：bison，gcc，libc6-dev，ed，gawk 和make；
- 设置环境变量GOROOT 为Go 安装目录：
% `export GOROOT=~/go`
- 然后获取Go 源代码：
% `hg clone -r release https://go.googlecode.com/hg/ $GOROOT`
- 设置PATH 到Go 的二进制文件所在目录，这样Shell 可以找到它们：
% `export PATH=$GOROOT/bin:$PATH`
- 编译Go
% `cd $GOROOT/src`
% `./all.bash`

如果全部都没问题，你应当看到下面的内容：

```
Installed Go for linux/amd64 in /home/go.  
Installed commands in /home/go/bin.  
The compiler is 6g.
```

现在，Go 已经被安装到了系统中，可以开始游戏了。

在Windows 下获得Go

在Windows 下使用Go 的最佳方式是MinGW [33] 环境。它为Windows 提供了一个类UNIX 的环境。这和Cygwin 比较类似，不过并没有尝试模拟POSIX，而是使用微软C 运行库和Windows API 代替。

- 从[33] 下载最新的版本；
- 将其解压缩到C:\ 盘；
- 确认C:\MinGW 中的内容存在。注意：当你解压缩MinGW 的时候目录应当已经被创建了；
- 确认存在C:\MinGW；
- 使用mintty 的快捷方式开启一个终端窗口；
- 将环境变量GOROOT 设为Go 的安装目录的根目录：

```
% export GOROOT=/c/go
```

- 用下面的命令下载最新的Go 稳定发布版本

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

- 进入src 目录

```
% cd $GOROOT/src
```

- 然后编译Go

```
% ./all.bash
```

现在就获得了一个可以工作的Go 环境。接下来需要设置GOROOT、GOBIN 和PATH 变量。

- 创建叫做setup.sh 的文件，并放置到MinGW 目录，文件内容如下：

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

- 现在运行mintty 的时候，可以执行这个文件来设置Go 环境：

```
% . ./setup.sh
```

保持更新

新的发布会公布在Go Nuts 邮件列表[20] 中。将已经存在的代码树更新到最新，需要执行：

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

看看你现在用的版本：

```
% cd $GOROOT
% hg identify
79997f0e5823 release/release.2010-10-20
```

这是发布版本2010-10-20。发布描述为“稳定”发布，相比之下，“weekly”发布就要变化更多一些。如果希望跟踪weekly 发布，而不是稳定的版本，可以使用：

```
% hg update weekly
```

代替

```
% hg update release
```

前身

Go 的前身来自于Inferno [14]（基于Plan 9 [16] 的改造）。Inferno 包含了一个叫做Limbo [17] 的语言。来自于Limbo 论文中的引用：

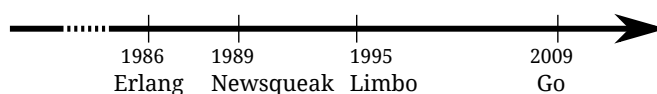
Limbo 是用于开发运行在小型计算机上的分布式应用的编程语言。它支持模块化编程，编译期和运行时的强类型检查，进程内基于具有类型的channel 通讯，原子性垃圾收集，和简单的抽象数据类型。它被设计用于即便是没有硬件内存保护的小型设备上，也能安全的运行。

Go 从Limbo 继承的另一个特性是channel（参阅第7 章）。从Limbo 文档来的另一段：

[channel] 是用于向系统中其他代理发送和接收特定类型对象的通讯机制。*channel* 可以用于本地进程间通讯；用于连接到命名的目的地的库方法。两种情况都是直接发送和接收操作的。

在Go 中，channel 比在Limbo 中更加好用。如果我们对Go 的历史深入探索，会发现一个到“Newsqueak” [30] 的引用，这是在类C 语言中使用channel 通讯的拓荒者。channel 对于这些语言并不是独一无二的，另一个非类C 语言，Erlang [21]，也在使用它。

Figure 1.1. Go 编年史



使用channel 与其他进程进行通讯的办法，叫做通讯序列化过程（Communicating Sequential Processes - CSP），由C. A. R. Hoare [28] 设计构想，而他正是那个发明快速排序[27] 算法的人。

| Go 是第一个实现了简单的（或更加简单的）并行开发的、跨平台类C 语言。

练习

Q1. (1) Documentation

- Go 的文档可以通过godoc 程序阅读，它包含在Go 的发布包中。
godoc hash 给出了hash 包的信息。阅读compress 包的帮助，给出了下面的结果：

```
% godoc compress
SUBDIRECTORIES
    bzip2
    flate
    gzip
    lzw
    testdata
    zlib
```

哪个godoc 的命令可以显示compress 包中的gzip 文档？

答案

A1. (1) Documentation

1. *gzip* 包在*compress* 的子目录中，所以只需要 `godoc compress/gzip` 即可。
也可以指定“Go 手册”中某个函数的的文档。例如，函数*Printf* 在*fmt* 包中，仅阅读这个函数的文档，使用：`godoc fmt Printf`。
甚至可以显示源代码：`godoc -src fmt Printf`。
所有的内建函数同样可以通过`godoc builtin`的方式在godoc 中访问。

2

基础

“在Go中，代码说到做到。”

Go Nuts 邮件列表
ANDREW GERRAND

有一些东西使得Go不同于其他语言。

清晰并且简洁

Go 努力保持小并且优美，你可以在短短几行代码里做许多事情。

并行

Go 让函数很容易成为非常轻量的线程。这些线程在Go中被叫做goroutines^a；

Channel

这些goroutines之间的通讯由channel [38, 28] 完成；

快速

编译很快，执行也很快。目标是跟C一样快。编译时间用秒计算；

安全

当转换一个类型到另一个类型的时候需要显式的转换并遵循严格的规则。Go有垃圾收集，在Go中无须free()，语言会处理这一切；

标准的格式化

Go程序可以被格式化为程序员希望的（几乎）任何形式，但是官方格式是存在的。标准也非常简单：gofmt的输出就是官方认可的格式。

类型后置

类型在变量名的后面，像这样var a int，来代替C中的int a；

UTF-8

任何地方都是UTF-8的，包括字符串以及程序代码。你可以在代码中使用 $\Phi = \Phi + 1$ ；

开源

Go的许可证是完全开源的，查阅Go发布的源码中的LICENSE文件。

开心

用Go写程序会非常开心！

Erlang [21] 与Go在部分功能上类似。Erlang和Go之间主要的区别是Erlang是函数式语言，而Go是命令式的。Erlang运行在虚拟机上，而Go是编译的。Go用起来感觉更接近Unix。

^a是的，它的发音很接近coroutines，但是goroutines确实有一些不同，我们将在第7章讨论。

Hello World

在Go 指南中，用一个传统的方式展现了Go：让它打印”Hello World”（Ken Thompson 和Dennis Ritchie 在20 世纪70 年代，发布C 语言的时候开创了这个先河）。我们不认为可以做得更好，所以就是这个，Go 的”Hello World”。

Listing 2.1. Hello world

```
package main ❶ 1

import "fmt" // 实现格式化的I/O。❶ 3

/* Print something */ ❷ 5
func main() { ❸ 6
    ❹ 7
    fmt.Printf("Hello, world; or καλημέρα κόσμε; or こんにちは世界\n") 8
} 9
```

逐行阅读这个程序。

- ❶ 首行这个是必须的。所有的Go 文件以**package** <something> 开头，对于独立运行的执行文件必须是**package main**；
- ❶ 这是说需要将”*fmt*”包加入**main**。不是**main** 的其他包都被称为库，其他许多编程语言有着类似的概念（参阅第4 章）。末尾以// 开头的内容是注释；
- ❷ 这同样是注释，不过这是被包裹于/* 和*/ 之间的；
- ❸ **package main** 必须首先出现，紧跟着是**import**。在Go 中，**package** 总是首先出现，然后是**import**，然后是其他所有内容。当Go 程序在执行的时候，首先调用的函数是**main.main()**，这是从C 中继承而来。这里定义了这个函数；
- ❹ 第8 行调用了来自于*fmt* 包的函数打印字符串到屏幕。字符串由" 包裹，并且可以包含非ASCII 的字符。这里使用了希腊文和日文。

编译和运行代码

Go 编译器叫做<数字>g，数字是6 表示用于64 位Intel 而8 表示32 位Intel。连接器用相同的命名方式：<数字>l。在本书中，我们将使用6g 和6l 完成所有的编译。为了编译上面的代码，执行：

```
% 6g helloworld.go
```

并且用6l 链接它：

```
% 6l helloworld.6
```

然后执行：

% ./6.out ← 默认的 (64位) Go 可执行文件名 (32 位上是 8.out)

Hello, world; or καλημέρα κόσμε; or こんにちは世界

使用Makefile

另一种可以少折腾的（在建立好后）编译Go 程序的办法，是使用Makefile。下面是用于构建helloworld的：

Listing 2.2. 程序的Makefile

```
include $(GOROOT)/src/Make.inc 1

TARG=helloworld 3
GOFILES=\ 4
    helloworld.go\ 5

include $(GOROOT)/src/Make.cmd 7
```

行3 指定了编译的程序的名字，而行5 列举了源代码文件。现在执行make 就可以完成你的程序编译。留意Go 使用了不同于make 的，叫做gomake 的命令。它是（当前来说）一个GNU make 的小封装。Go 程序的构建系统在将来可能会发生变化，从而make 会被移除。所以我们使用gomake 进行构建。留意Makefile 创建的可执行文件叫做helloworld，而不是6.out 或者8.out。

变量、类型和保留字

在接下来的章节中，我们将会了解这个新语言的变量、基本类型、保留字和控制流。Go 在语法上有着类C 的感觉。如果你希望将两个（或更多）语句放在一行书写，它们必须用分号(;)分隔。一般情况下，你不需要分号。

Go 同其他语言不同的地方在于变量的类型在变量名的后面。不是：`int a`，而是[a int](#)。当定义了一个变量，它默认赋值为其类型的null 值。这意味着，在[var a int](#)后，a 的值为0。而[var s string](#)，意味着s 被赋值为零长度字符串，也就是""。

在Go 中，声明和赋值是两步的一个过程，但是可以连在一起。比较下面作用相同的代码片段。

Listing 2.3. Declaration with =

```
var a int
var b bool
a = 15
b = false
```

Listing 2.4. Declaration with :=

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

在左边使用了保留字[var](#) 声明变量，然后赋值给它。右边的代码使用了[:=](#) 使得在一步内完成了声明和赋值（这一形式只可用在函数内）。在这种情况下，变量的类型是由值推演出来的。值15 表示是[int](#) 类型，值false 告诉Go 它的类型应当是[bool](#)。多个[var](#) 声明可以成组，[const](#) 和[import](#) 同样允许这么做。留意圆括号的使用：

```
var (
    x int
    b bool
)
```

有相同类型的多个变量同样可以在一行内完成声明：`var x, y int` 让 `x` 和 `y` 都是 `int` 类型变量。同样可以使用平行赋值：

```
a, b := 20, 16
```

让 `a` 和 `b` 都是整数变量，并且赋值 20 给 `a`，16 给 `b`。

一个特殊的变量名是 `_`（下划线）。任何赋给它的值都被丢弃。在这个例子中，将 35 赋值给 `b`，同时丢弃 34。

```
_, b := 34, 35
```

Go 的编译器对声明却未使用的变量在报错，下面的代码会产生这个错误：声明了 `i` 却未使用

```
package main
func main() {
    var i int
}
```

布尔类型

布尔类型表示由预定义的常量 `true` 和 `false` 代表的布尔判定值。布尔类型是 `bool`。

数字类型

Go 有众所周知的类型如 `int`，这个类型根据你的硬件决定适当的长度。意味着在 32 位硬件上，是 32 位的；在 64 位硬件上是 64 位的。注意：`int` 是 32 或 64 位之一，不会定义成其他值。`uint` 情况相同。

如果你希望明确其长度，你可以使用 `int32` 或者 `uint32`。完整的整数类型列表（符号和无符号）是 `int8`，`int16`，`int32`，`int64` 和 `byte`，`uint8`，`uint16`，`uint32`，`uint64`。`byte` 是 `uint8` 的别名。浮点类型的值有 `float32` 和 `float64`（没有 `float` 类型）。64 位的整数和浮点数总是 64 位的，即便是在 32 位的架构上。

需要留意的是这些类型全部都是独立的，并且混合用这些类型向变量赋值会引起编译器错误，例如下面的代码：

Listing 2.5. Familiar types are still distinct

```
package main                                     1

func main() {                                     3
    var a int                                     4
    var b int32                                   5
    a = 15                                         6
    b = a + a                                     7
    b = b + 5                                     8
}                                                  9
```

\leftarrow Generic integer type
 \leftarrow 32 bits integer type
 \leftarrow Illegal mixing of these types
 \leftarrow 5 is a (typeless) constant, so this is OK

在行7 触发一个赋值错误：

```
types.go:7: cannot use a + a (type int) as type int32 in assignment
```

赋值可以用八进制、十六进制或科学计数法：077, 0xFF, 1e3 or 6.022e23 这些都是合法的。

常量

常量在Go 中，也就是constant。它们在编译时被创建，只能是数字、字符串或布尔值；`const x = 42` 生成x 这个常量。可以使用*iota*^b 生成枚举值。

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

第一个*iota* 表示为0，因此a 等于0，当*iota* 再次在新的一行使用时，它的值增加了1，因此b 的值是1。

也可以像下面这样，省略Go 重复的*iota*：

```
const (
    a = iota
    b           ← Implicitly b = iota
)
```

如果需要，可以明确指定常量的类型：

```
const (
    a = 0           ← Is an int now
    b string = "0"
```

字符串

另一个重要的内建类型是string。赋值字符串的例子：

```
s := "Hello World!"
```

字符串在Go 中是UTF-8 的由双引号(") 包裹的字符序列。如果你使用单引号(') 则表示一个字符 (UTF-8编码) ——这种在Go 中不是 string。

一旦给变量赋值，字符串就不能修改了：在Go 中字符串是不可变的。从C 来的用户，下面的情况在Go 中是非法的。

```
var s string = "hello"
s[0] = 'c'      ← 修改第一个字符为'c'，这会报错
```

在Go 中实现这个，需要下面的方法：

```
s := "hello"
c := []byte(s)    ❶
c[0] = 'c'        ❷
```

^b 单词[iota] 在日常英语短语‘not one iota’，意思是‘不是最小的差异’，是来自新约中的短语：“until heaven and earth pass away, not an iota, not a dot, will pass from the Law.” [40]

```
s2 := string(c) ❷  
fmt.Printf("%s\n", s2) ❸
```

❶ 转换s 为字节数组，查阅在第5 章“转换”节、65 页的内容；

❷ 修改数组的第一个元素；

❸ 创建新的字符串s2 保存修改；

❹ 用fmt.Printf 函数输出字符串。

多行字符串

基于分号的置入（查阅[6] 章节“Semicolons”），你需要小心使用多行字符串。如果这样写：

```
s := "Starting part"  
    + "Ending part"
```

会被转换为：

```
s := "Starting part";  
    + "Ending part";
```

这是错误的语法，应当这样写：

```
s := "Starting part" +  
    "Ending part"
```

Go 就不会在错误的地方插入分号。另一种方式是使用反引号` 作为原始字符串符号：

```
s := `Starting part  
    Ending part`
```

留意最后一个例子s 现在也包含换行。不像转义字符串标识，原始字符串标识的值在引号内的字符是不转义的。

复数

Go 原生支持复数。它的变量类型是**complex128**（64 位虚数部分）。如果需要小一些的，还有**complex64**——32 位的虚数部分。复数写为 $re + imi$ ，re 是实数部分，im 是虚数部分，而i 是标记 i （ $\sqrt{-1}$ ）。使用复数的一个例子：

```
var c complex64 = 5+5i; fmt.Printf("Value is: %v", c)  
将会打印：(5+5i)
```

错误

任何足够大的程序或多或少都会需要使用到错误。因此Go 有为了错误而存在的内建类型。

运算符和内建函数

Go 支持普通的数字运算符，表格2.1 列出了当前支持的运算符，以及其优先级。它们全部是从左到右结合的。

Table 2.1. 运算优先级

Precedence	Operator(s)
Highest	* / % << >> & &^
	+ - ^
	== != < <= > >=
	<-
	&&
Lowest	

+ - * / 和% 会像你期望的那样工作，& | ^ 和&^ 分别表示位运算符按位与，按位或，按位异或和位清除。&& 和|| 运算符是逻辑与和逻辑或。表格中没有列出的是逻辑非：!。

虽然Go 不支持运算符重载（或者方法重载），而一些内建运算符却支持重载。例如+ 可以用于整数、浮点数、复数和字符串（字符串相加表示串联它们）。

Go 保留字

Table 2.2. Go 中的保留字

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

表格2.2 列出了Go 中所有的保留字。在下面的段落和章节中会介绍它们。其中有一些已经遇到过了。

- **var** 和**const** 参阅”变量、类型和保留字” 在8 页；
- **package** 和**import** 已经有过短暂的接触，在”Hello World” 部分。在第4 章对其有详细的描述。

其他都有对应的介绍和章节：

- **func** 用于定义函数和方法；
- **return** 用于从函数返回，**func** 和**return** 参阅第3 章了解详细信息；
- **go** 用于并行（第7 章）；

- **select** 用于选择不同类型的通讯，参阅第7章；
- **interface** 参阅第6章；
- **struct** 用于抽象数据类型，参阅第5章；
- **type** 同样参阅第5章。

控制结构

在Go中只有很少的几个控制结构^c。例如这里没有do或者while循环，只有**for**。有（灵活的）**switch**语句和**if**，而**switch**接受像**for**那样可选的初始化语句。还有叫做类型选择和多路通讯转接器的**select**（参阅第7章）。语法有所不同（同C相比）：无需圆括号，而语句体必须总是包含在大括号内。

if

在Go中**if**看起来是这样的：

```
if x > 0 {           ← { is mandatory
    return y
} else {
    return x
}
```

强制大括号鼓励将简单的**if**语句写在多行上。无论如何，这都是一个很好的形式，尤其是语句体中含有控制语句，例如**return**或者**break**。

if和**switch**接受初始化语句，通常用于设置一个（局部）变量。

```
if err := file.Chmod(0664); err != nil {    ← nil is like C's NULL
    log.Stderr(err)    ← Scope of err is limited to if's body
    return err
}
```

可以像通常那样使用逻辑运算符（参考2.1表格）：

```
if true && true {
    println("true")
}
if ! false {
    println("true")
}
```

在Go库中，你会发现当一个**if**语句不会进入下一个语句流程——也就是说，语句体结束于**break**，**continue**，**goto**或者**return**，不必要的**else**会被省略。

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
doSomething(f)
```

^c这个章节复制于[6]。

这个例子通常用于检测可能的错误序列。成功的流程一直执行到底部使代码很好读，当遇到错误的时候就排除它。这样错误的情况结束于**return**语句，这样就无须**else**语句。

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    return err
}
doSomething(f, d)
```

下面的语法在Go中是非法的：

```
if err != nil
{
    return err
}
```

← 必须同if 在同一行

参阅[6] "Semicolons" 章节了解其后更深入的原因。

if-then-else 的结尾

注意如果在函数中这样结束：

```
if err != nil {
    return err
} else {
    return nil
}
```

它不会编译。这是Go编译器的一个bug。参阅[19]了解更多关于此问题的描述，以及可能的修复。

goto

Go有**goto**语句——明智地使用它。用**goto**跳转到一定是当前函数内定义的标签。例如假设这样一个循环：

```
func myfunc() {
    i := 0
Here:      ← 这行的第一个词，以分号结束作为标签
    println(i)
    i++
    goto Here    ← 跳转
}
```

标签名是大小写敏感的。

for

Go的**for**循环有三种形式，只有其中的一种使用分号。

```

for init; condition; post { }    ← 和C的for一样

for condition { }                ← 和while一样

for { }                          ← 和C的for(;;)一样 (死循环)

```

短声明使得在循环中声明一个序号变量更加容易。

```

sum := 0
for i := 0; i < 10; i++ {
    sum += i    ← sum = sum + i 的简化写法
}              ← i 实例在循环结束会消失

```

最后，由于Go没有逗号表达式，而++和-是语句而不是表达式，如果你想在**for**中执行多个变量，应当使用平行赋值。

```

// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {    ← 平行赋值
    a[i], a[j] = a[j], a[i]    ← 这里也是
}

```

break 和continue

利用**break**可以提前退出循环，**break**终止当前的循环。

```

for i := 0; i < 10; i++ {
    if i > 5 {
        break    ← 终止这个循环，只打印0到5
    }
    println(i)
}

```

循环嵌套循环时，可以在**break**后指定标签。用标签决定哪个循环被终止：

```

J: for j := 0; j < 5; j++ {
    for i := 0; i < 10; i++ {
        if i > 5 {
            break J    ← 现在终止的是j循环，而不是i的那个
        }
        println(i)
    }
}

```

利用**continue**让循环进入下一个迭代，而略过剩下的所有代码。下面打印了0到5。

```

for i := 0; i < 10; i++ {
    if i > 5 {
        continue    ← 跳过下面所有的代码println(i)
    }
    println(i)
}

```


range

保留字 **range** 可用于循环。它可以在 slice、array、string、map 和 channel（参阅第7章）。**range** 是个迭代器，当被调用的时候，从它循环的内容中返回一个键值对。基于不同的内容，**range** 返回不同的东西。

当对 slice 或者 array 做循环时，**range** 返回序号作为键，这个序号对应的内容作为值。考虑这个代码：

```
list := []string{"a", "b", "c", "d", "e", "f"} ❶
for k, v := range list { ❷
    // 对k 和v 做想做的事情❸
}
```

❶ 创建一个字符串的 slice（参阅“array、slices 和 map”在19页）。

❷ 用 **range** 对其进行循环。每一个迭代，**range** 将返回 **int** 类型的序号，**string** 类型的值，以0 和“a”开始。

❸ k 的值为0...5，而v 在循环从“a”...“f”。

也可以在字符串上直接使用 **range**。这样字符串被打散成独立的Unicode 字符^d 并且起始位按照UTF-8 解析。循环：

```
for pos, char := range "aΦx" {
    fmt.Printf("character '%c' starts at byte position %d\n", char, pos)
}
```

打印

```
character 'a' starts at byte position 0
character 'Φ' starts at byte position 1
character 'x' starts at byte position 3    ← Φ took 2 bytes
```

switch

Go 的 **switch** 非常灵活。表达式不必是常量或整数，执行的过程从上至下，直到找到匹配项，而如果 **switch** 没有表达式，它会匹配 **true**。这产生一种可能——使用 **switch** 编写 **if-else-if-else** 判断序列。

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
```

^d在UTF-8 世界的字符有时被称作runes。通常，当人们讨论字符时，多数是指8 位字符。UTF-8 字符可能会有32 位，称作rune。

```
    return 0
}
```

它不会匹配失败后自动向下尝试，但是可以使用 `fallthrough` 使其这样做。没有 `fallthrough`：

```
switch i {
    case 0: // 空的case 体
    case 1:
        f() // 当 i == 0 时，f 不会被调用！
}
```

而这样：

```
switch i {
    case 0: fallthrough
    case 1:
        f() // 当 i == 0 时，f 会被调用！
}
```

用 `default` 可以指定当其他所有分支都不匹配的时候的行为。

```
switch i {
    case 0:
    case 1:
        f()
    default:
        g() // 当 i 不等于 0 或 1 时调用
}
```

分支可以使用逗号分隔的列表。

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+': ← , as "or"
        return true
    }
    return false
}
```

这里有一个使用两个 `switch` 对字节数组进行比较的例子：

```
// 比较返回两个字节数组字典数序先后的整数。
// 如果 a == b 返回 0，如果 a < b 返回 -1，而如果 a > b 返回 +1
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    // 长度不同，则不相等
    switch {
```

```

    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0    // 字符串相等
}

```

内建函数

预定义了少数函数，这意味着无需引用任何包就可以使用它们。表格2.3 列出了所有的内建函数。

Table 2.3. Go 中的预定义函数

close	new	panic	complex
delete	make	recover	real
len	append	print	imag
cap	copy	println	

close 用于channel 通讯。使用它来关闭channel，参阅第7 章了解更多。

delete 用于在map 中删除实例。

len 和 *cap* 可用于不同的类型，*len* 用于返回字符串、slice 和数组的长度。参阅“array、slices 和 map” 小节了解更多关于slice、数组和函数*cap* 的详细信息。

new 用于各种类型的内存分配。参阅“用 new 分配内存” 在61 页。

make 用于内建类型（map、slice 和channel）的内存分配。参阅“用 make 分配内存” 在61 页。

copy 用于复制slice。*append* 用于追加slice。参阅本章的“slice”。

panic 和 *recover* 用于异常处理机制。参阅“恐慌（Panic）和恢复（Recover）” 在37 页了解更多信息。

print 和 *println* 是底层打印函数，可以在不引入*fmt* 包的情况下使用。它们主要用于调试。

complex、*real* 和 *imag* 全部用于处理复数。有了之前给的简单的例子，不用再进一步讨论复数了。

array、slices 和 map

可以利用array 在列表中进行多个值的排序，或者使用更加灵活的：slice。字典或哈希类型同样可以使用，在Go 中叫做map。

array

array 由[n]<type> 定义，*n* 标示array 的长度，而<type> 标示希望存储的内容的类型。对array 的元素赋值或索引是由方括号完成的：

```
var arr [10]int
arr[0] = 42
arr[1] = 13
fmt.Printf("The first element is %d\n", arr[0])
```

像var arr = [10]int 这样的数组类型有固定的大小。大小是类型的一部分。由于不同的类型，因此不能改变大小。数组同样是值类型的：将一个数组赋值给另一个数组，会复制所有的元素。尤其是当向函数内传递一个数组的时候，它会获得一个数组的副本，而不是数组的指针。

可以像这样声明一个数组：var a [3]int，如果不使用零来初始化它，则用复合声明：a := [3]int{1, 2, 3} 也可以简写为a := [...]int{1, 2, 3}，Go 会自动统计元素的个数。注意，所有项目必须都指定。因此，如果你使用多维数组，有一些内容你必须录入：

```
a := [2][2]int{ [2]int{1,2}, [2]int{3,4} }
```

类似于：

```
a := [2][2]int{ [...]int{1,2}, [...]int{3,4} }
```

当声明一个array 时，你必须在方括号内输入些内容，数字或者三个点(...)。从发布版2010-10-27[10] 开始这个语法就变得更加简单了。来自于发布记录：

array、slice 和map 的复合声明变得更加简单。使用复合声明的array、slice 和map，元素复合声明的类型与外部一直，则可以省略。

这表示上面的例子可以修改为：

```
a := [2][2]int{ {1,2}, {3,4} }
```

slice

slice 与array 接近，但是在新的元素加入的时候可以增加长度。slice 总是指向底层的一个array。slice 是一个指向 array 的指针，这是其与array 不同的地方；slice 是引用类型，这意味着当赋值某个slice 到另外一个变量，两个引用会指向同一个array。例如，如果一个函数需要一个slice 参数，在其内对slice 元素的修改也会体现在函数调用者中，这和传递底层的array 指针类似。通过：

```
sl := make([]int, 10)
```

创建了一个保存有10 个元素的slice。需要注意的是底层的array 并无不同。slice 总是与一个固定长度的array 成对出现。其影响slice 的容量和长度。图2.1 描述了下面的Go 代码。首先创建了*m* 个元素长度的array，元素类型int：var array[m]int

复合声明允许你直接将值赋值给array、slice 或者map。参阅第62 页的“构造函数与复合声明”章节了解更多信息。

Go 发布版2010-10-27 [10]。

TODO

Add push/pop to this section as container/vector will be deprecated.

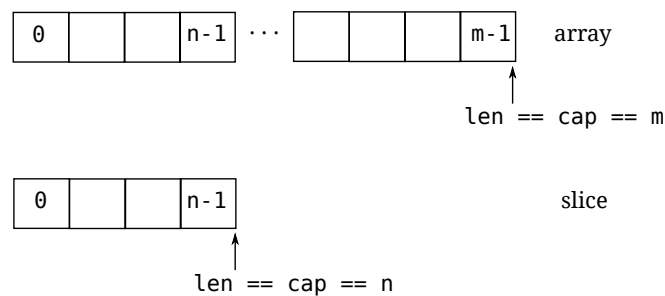
引用类型使用make 创建。

然后对这个array 创建slice : `slice := array[0:n]`

然后现在有 :

- `len(slice) == n == cap(slice) == n;`
- `len(array) == cap(array) == m.`

Figure 2.1. array 与slice 对比



给定一个array 或者其他slice , 一个新slice 通过`a[I:J]` 的方式创建。这会创建一个新的slice, 指向a, 从序号I 开始, 结束在序号J之前。长度为J - I。

// `array[n:m]` 从array 创建了一个slice , 具有元素n to m-1

`a := [...]int{1, 2, 3, 4, 5}` ❶

`s1 := a[2:4]` ❷

`s2 := a[1:5]` ❸

`s3 := a[:]` ❹

`s4 := a[:4]` ❺

`s5 := s2[:]` ❻

- ❶ 定义一个5 个元素的array, 序号从0 到4 ;
- ❷ 从序号2 至3 创建slice, 它包含元素3, 4 ;
- ❸ 从序号1 至4 创建, 它包含元素2, 3, 4, 5 ;
- ❹ 用array 中的所有元素创建slice, 这是`a[0:len(a)]` 的简化写法 ;
- ❺ 从序号0 至3 创建, 这是`a[0:4]` 的简化写法, 得到1, 2, 3, 4 ;
- ❻ 从slice s2 创建slice, 注意s5 仍然指向array a。

在2.6 列出的代码中, 我们在第八行尝试做一些错误的事情, 让一些东西超出范围 (底层array 的最大长度), 然后得到了一个运行时错误。

Listing 2.6. array 和slice

`package main`

1

```

func main() {
    var array [100]int    // Create array, index from 0 to 99
    slice := array[0:99]  // Create slice, index from 0 to 98

    slice[98] = 'a'       // OK
    slice[99] = 'a'       // Error: "throw: index out of range"
}

```

如果你想要扩展slice，有一堆内建函数让你的日子更加好过一些：**append** 和**copy**。来自于[8]：

函数**append** 向slice *s* 追加零值或其他*x* 值，并且返回追加后的新的、与*s* 有相同类型的slice。如果*s* 没有足够的容量存储追加的值，**append** 分配一个足够大的、新的slice 来存放原有slice 的元素和追加的值。因此，返回的slice 可能指向不同的底层array。

```

s0 := []int{0, 0}
s1 := append(s0, 2)
s2 := append(s1, 3, 5, 7)
s3 := append(s2, s0...)

```

- ① 追加一个元素，*s1* == []int{0, 0, 2}；
- ① 追加多个元素，*s2* == []int{0, 0, 2, 3, 5, 7}；
- ② 追加一个slice，*s3* == []int{0, 0, 2, 3, 5, 7, 0, 0}。注意这三个点！

还有

函数**copy** 从源slice *src* 复制元素到目标*dst*，并且返回复制的元素的个数。源和目标可能重叠。复制的数量是**len(src)** 和**len(dst)** 中的最小值。

```

var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
n1 := copy(s, a[0:])    ← n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])    ← n2 == 4, s == []int{2, 3, 4, 5, 4, 5}

```

map

许多语言都内建了类似的类型，例如Perl 有哈希，Python 有字典，而C++ 同样也有map（作为它们的库）。在Go 中有**map** 类型。**map** 可以认为是一个用字符串做索引的数组（在其最简单的形式下）。下面定义了**map** 类型，用于将**string**（月的缩写）转换为**int**——那个月的天数。一般定义map 的方法是：**map[<from type>]<to type>**

```

monthdays := map[string]int{
    "Jan": 31, "Feb": 28, "Mar": 31,
    "Apr": 30, "May": 31, "Jun": 30,
    "Jul": 31, "Aug": 31, "Sep": 30,
    "Oct": 31, "Nov": 30, "Dec": 31,    ← 逗号是必须的
}

```

留意，当只需要声明一个map的时候，使用make的形式：`monthdays := make(map[string] int)`

当在map中索引（搜索）时，使用方括号，例如打印出12月的天数：`fmt.Printf("%d\n", monthdays["Dec"])`

当对array、slice、string或者map循环遍历的时候，**range**会帮助你，每次调用，它都会返回一个键和对应的值。

```
year := 0
for _, days := range monthdays {      ← 键没有使用，因此用_, days
    year += days
}
fmt.Printf("Numbers of days in a year: %d\n", year)
```

向map增加元素，可以这样做：

```
monthdays["Undecim"] = 30      ← 添加一个月
monthdays["Feb"]      = 29     ← 闰年时重写这个元素
```

检查元素是否存在，可以使用下面的方式[31]：

```
var value int
var present bool

value, present = monthdays["Jan"]    ← 如果存在，present 则有值true
                                      ← 或者更接近Go的方式
v, ok := monthdays["Jan"]           ← “逗号ok”形式
```

也可以从map中移除元素：

```
delete monthdays["Mar"]            ← 删除"Mar"吧，总是下雨的月份
```

通常来说语句`delete(m, x)`会删除map中由`m[x]`建立的实例。

练习

Q2. (1) For-loop

1. 创建一个基于**for**的简单的循环。使其循环10次，并且使用**fmt**包打印出计数器的值。
2. 用**goto**改写1的循环。保留字**for**不可使用。
3. 再次改写这个循环，使其遍历一个array，并将这个array打印到屏幕上。

Q3. (1) FizzBuzz

1. 解决这个叫做Fizz-Buzz [36]的问题：

编写一个程序，打印从1到100的数字。当是三个倍数就打印“Fizz”代替数字，当是五的倍数就打印“Buzz”。当数字同时是三和五的倍数时，打印“FizzBuzz”。

Q4. (1) Strings

1. 建立一个Go程序打印下面的内容（到100个字符）：

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
...
```

2. 建立一个程序统计字符串里的字符数量：

```
asSASA ddd dsjksjs dk
```

同时输出这个字符串的字节数。提示：看看 *utf8* 包。

3. 扩展上一个问题的程序，替换位置4 开始的三个字符为'abc'。
4. 编写一个Go 程序可以逆转字符串，例如"foobar" 被打印成"rafoof"。提示：不幸的是你需要知道一些关于转换的内容，去"转换" 在65" 页的内容。

Q5. (4) Average

1. 编写计算一个类型是 `float64` 的slice 的平均值的代码。在稍候的练习中(Q6 将会改写为函数)。

答案

A2. (1) For-loop

1. 有许多种解法，其中一种可能是：

Listing 2.7. Simple for loop

```
package main

import "fmt"

func main() {
    for i := 0; i < 10; i++ {    ← See section for on page 14
        fmt.Printf("%d\n", i)
    }
}
```

编译并观察输出。

```
% 6g for.go && 6l -o for for.6
% ./for
0
1
.
.
.
9
```

2. 改写的循环最终看起来像这样（仅显示了main函数）：

```
func main() {
    i := 0                ← 定义循环变量
I:                      ← 定义标签
    fmt.Printf("%d\n", i)
    i++
    if i < 10 {
        goto I          ← 跳转回标签
    }
}
```

3. 下面是可能的解法之一：

Listing 2.8. For loop with an array

```
func main() {
    var arr [10]int      ← Create an array with 10 elements
    for i := 0; i < 10; i++ {
        arr[i] = i       ← Fill it one by one
    }
    fmt.Printf("%v", arr)    ← With %v Go prints the type
}
```

也可以用复合声明的硬编码来实现这个：

```
a := [...]int{0,1,2,3,4,5,6,7,8,9}    ← 通过[...] 让Go 来计数
fmt.Printf("%v\n", a)
```

A3. (1) FizzBuzz

1. 下面简单的程序，是一种解决办法。

Listing 2.9. Fizz-Buzz

```
package main

import "fmt"

func main() {
    const (
        FIZZ = 3 ❶
        BUZZ = 5
    )
    var p bool ❶
    for i := 1; i < 100; i++ { ❷;
        p = false
        if i%FIZZ == 0 { ❸
            fmt.Printf("Fizz")
            p = true
        }
        if i%BUZZ == 0 { ❹
            fmt.Printf("Buzz")
            p = true
        }
        if !p { ❺
            fmt.Printf("%v", i)
        }
        fmt.Println() ❻
    }
}
```

- ❶ Define two constants to make the code more readable. See section “常量”;
- ❶ Holds if we already printed something;
- ❷ for-loop, see section “for”
- ❸ If divisible by FIZZ, print “Fizz”;
- ❹ And if divisible by BUZZ, print “Buzz”. Note that we have also taken care of the FizzBuzz case;
- ❺ If neither FIZZ nor BUZZ printed, print the value;
- ❻ Format each output on a new line.

A4. (1) Strings

1. 这是一个解法：

Listing 2.10. Strings

```
package main

import "fmt"

func main() {
    str := "A"
    for i := 0; i < 100; i++ {
        fmt.Printf("%s\n", str)
        str = str + "A"    ← String concatenation
    }
}
```

2. 为了解决这个问题，需要`utf8`包的帮助。首先，阅读一下文档[godoc utf8 | less](#)。在阅读文档的时候，会注意到`func RuneCount(p []byte)int`。然后，将`string`转换为`byte slice`：

```
str := "hello"
b := []byte(str)    ← 转换，参阅65 页
```

将这些整合到一起，得到下面的程序。

Listing 2.11. Runes in strings

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    str := "dsjkdshdjsdh...js"
    fmt.Printf("String %s\nLenght: %d, Runes: %d\n", str,
        len([]byte(str)), utf8.RuneCount([]byte(str)))
}
```

3. 可以用下面的方法逆转字符串。我们从左边（i）至右（j）的交换字符，就像这样：

Listing 2.12. Reverse a string

```

import "fmt"

func main() {
    s := "foobar"
    a := []byte(s)    ← Again a conversion
    // Reverse a
    for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
        a[i], a[j] = a[j], a[i]    ← Parallel assignment
    }
    fmt.Printf("%s\n", string(a))    ← Convert it back
}

```

A5. (4) Average

1. 下面的代码计算了平均值。

```

sum := 0.0
switch len(xs) {
case 0:    ❶
    ave = 0
default:    ❷
    for _, v := range xs {
        sum += v
    }
    ave = sum / float64(len(xs))    ❸
}

```

❶ 如果长度是零，返回0；

❷ 否则计算平均值；

❸ 为了能够进行除法，必须将值转换为float64。

3

函数

“我总是兴奋于阳光的轻抚和沉寂在早期编程语言中。无需太多文字；许多已经完成了。旧的程序阅读起来就像是同表达良好的研究工作者或受到良好训练的机器同事沟通一样，而不是与编译器争论。谁愿意让其成熟到发出这样的声音呢？”

RICHARD P. GABRIEL

函数是构建Go程序的基础部件；所遇有趣的事情都是在它其中发生的。函数的定义看起来像这样：

Listing 3.1. 函数定义

`type mytype int` ← 新的类型，参阅5 章节

`func (p mytype) funcname(q int) (r,s int) { return 0,0 }`

① ② ③ ④ ⑤

- ① 保留字 `func` 用于定义一个函数；
- ② 函数可以定义用于特定的类型，这类函数更加通俗的称呼是 `method`。这部分称作 *receiver* 而它是可选的。它将在6 章使用；
- ③ `funcname` 是你函数的名字；
- ④ `int` 类型的变量 `q` 是输入参数。参数用 *pass-by-value* 方式传递，意味着它们会被复制；
- ⑤ 变量 `r` 和 `s` 是这个函数的 *named return parameters*。在Go 的函数中可以返回多个值。参阅“多个返回值”在32。如果想要返回无命名的参数，只需要提供类型：`(int, int)`。如果只有一个返回值，可以省略圆括号。如果函数是一个子过程，并且没有任何返回值，也可以省略这些内容；
- ⑥ 这是函数体，注意 `return` 是一个语句，所以包裹参数的括号是可选的。

这里有两个例子，左边的函数没有返回值，右边的只是简单的将输入返回。

```
func subroutine(in int) {  
    return  
}  
  
func identity(in int) int {  
    return in  
}
```

可以随意安排函数定义的顺序，编译器会在执行前扫描每个文件。所以函数原型在Go 中都是过期的旧物。Go 不允许函数嵌套。然而你可以利用匿名函数实现它，参阅本章第36 页的“函数作为值”。

递归函数跟其他语言是一样的：

Listing 3.2. 递归函数

```
func rec(i int) {  
    if i == 10 {  
        return  
    }  
    rec(i+1)  
    fmt.Printf("%d ", i)  
}
```

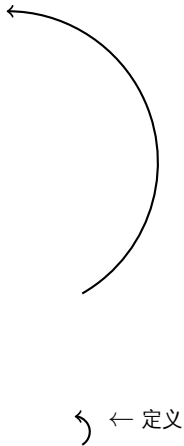
这会打印：9 8 7 6 5 4 3 2 1 0。

作用域

在Go 中，定义在函数外的变量是全局的，那些定义在函数内部的变量，对于函数来说是局部的。如果命名覆盖——一个局部变量与一个全局变量有相同的名字——在函数执行的时候，局部变量将覆盖全局变量。

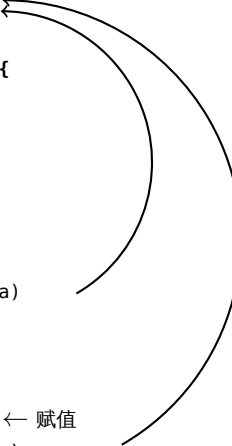
Listing 3.3. 局部作用域

```
package main  
  
var a = 6  
  
func main() {  
    p()  
    q()  
    p()  
}  
  
func p() {  
    println(a)  
}  
  
func q() {  
    a := 5  
    println(a)  
}
```



Listing 3.4. 全局作用域

```
package main  
  
var a = 6  
  
func main() {  
    p()  
    q()  
    p()  
}  
  
func p() {  
    println(a)  
}  
  
func q() {  
    a = 5  
    println(a)  
}
```



在3.3 中定义了函数q() 的局部变量a。局部变量a 仅在q() 中可见。这也就是为什么代码会打印：656。在3.4 中没有定义局部变量，只有全局变量a。这将使得赋值全局可见。这段代码将会打印：655。

在下面的例子中，我们在f() 中调用g()：

Listing 3.5. 当函数调用函数时的作用域

```
package main  
  
var a int
```



```

func main() {
    a = 5
    println(a)
    f()
}

func f() {
    a := 6
    println(a)
    g()
}

func g() {
    println(a)
}

```

输出内容将是：565。局部变量仅仅在执行定义它的函数时有效。

多个返回值

Go 一个非常特别的特性是函数和方法可以返回多个值（Python 同样也可以）。这可以用于改进一大堆在C 程序中糟糕的惯例用法：修改参数的方式，返回一个错误（例如遇到EOF 则返回-1）。在Go 中，Write 返回一个计数值和一个错误：“是的，你写入了一些字节，但是由于设备异常，并不是全部都写入了。”。os 包中的*File.Write 是这样声明的：

```
func (file *File) Write(b []byte) (n int, err Error)
```

如同文档所述，它返回写入的字节数和一个非nil 的Error 当n != len(b)。这是Go 中常见的样式。

类似的方法避免了传递指针模拟引用参数来返回值。这里有个样例函数，从字节数组的指定位上取得数值，返回这个值和下一个位置。

```

func nextInt(b []byte, i int) (int, int) {
    x := 0
    // 假设所有的都是数字
    for ; i < len(b); i++ {
        x = x*10 + int(b[i])-'0'
    }
    return x, i
}

```

你可以在输入的数组中扫描数字，像这样：

```

a := []byte{'1', '2', '3', '4'}
var x int
for i := 0; i < len(a); {           ← 没有i++
    x, i = nextInt(a, i)
    println(x)
}

```

没有元组作为原生类型，多返回值可能是最佳的选择。你可以精确的返回希望的值，而无须重载域空间到特定的错误信号上。

命名返回参数

Go 函数的返回值或者结果参数可以指定一个名字，并且像原始的变量那样使用，就像输入参数那样。如果对其命名，在函数开始时，它们会用其类型的零值初始化；如果函数在不加参数的情况下执行了 **return** 语句，结果参数的当前值会作为返回值返回。用这个特性，允许（再一次的）用较少的代码做更多的事^a。

名字不是强制的，但是它们可以使得代码更加健壮和清晰：这是文档。如果命名 `nextInt` 的 `int` 返回值哪个代表哪个。

```
func nextInt(b []byte, pos int) (value, nextPos int) { /* ... */ }
```

由于命名结果会被初始化并关联于无修饰的 **return**，它们可以非常简单并且清晰。这里有一个 `ioutil.ReadFull` 的版本，很好的运用了它：

```
func ReadFull(r Reader, buf []byte) (n int, err os.Error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:len(buf)]
    }
    return
}
```

在下面的例子中，定义了一个简单的函数，用于计算 值 `x` 的阶乘。

这一节的部分内容来自[18]。

```
func Factorial(x int) int {    ← func Factorial(x int) (int) 同样也行
    if x == 0 {
        return 1
    } else {
        return x * Factorial(x - 1)
    }
}
```

所以，也可以将函数编写为：

```
func Factorial(x int) (result int) {
    if x == 0 {
        result = 1
    } else {
        result = x * Factorial(x - 1)
    }
    return
}
```

当命名了返回值，代码变得健壮并且易读。同样也可以编写一个多返回值的函数：

```
func fib(n) (val, pos int) {    ← 都是int
    if n == 0 {
        val = 1
        pos = 0
    }
```

^a这是Go 的格言：“用更少的代码做更多的事”。

```

    } else if n == 1 {
        val = 1
        pos = 1
    } else {
        v1, _ := fib(n-1)
        v2, _ := fib(n-2)
        val = v1 + v2
        pos = n
    }
    return
}

```

延迟的代码

假设有一个函数，打开文件并且对其进行若干读写。在这样的函数中，经常有提前返回的地方。如果你这样做，就需要关闭正在工作的文件描述符。这经常导致产生下面的代码：

Listing 3.6. 没有defer

```

func ReadWrite() bool {
    file.Open("file")
    // 做一些工作
    if failureX {
        file.Close() ←
        return false
    }

    if failureY {
        file.Close() ←
        return false
    }
    file.Close() ←
    return true
}

```

在这里有许多重复的代码。为了解决这些，Go 有了 **defer** 语句。在 **defer** 后指定的函数会在函数退出前调用。

上面的代码可以被改写为下面这样。这使得函数更加可读、健壮，将 **Close** 对应的放置于 **Open** 后。

Listing 3.7. With defer

```

func ReadWrite() bool {
    file.Open("file")
    defer file.Close() ← file.Close() 被添加到了defer 列表
    // 做一些工作
    if failureX {
        return false ← Close() 现在自动调用
    }
    if failureY {

```

```

        return false    ← 这里也是
    }
    return true
}

```

可以将多个函数放入“延迟列表”中，这个例子来自[6]：

```

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

```

延迟的函数是按照后进先出（LIFO）的顺序执行，所以上面的代码打印：4 3 2 1 0。

利用defer 甚至可以修改返回值，假设正在使用命名结果参数和函数符号^b，例如：

Listing 3.8. 函数符号

```

defer func() {
    /* ... */
}()    ← () 在这里是必须的

```

或者这个例子，更加容易了解为什么，以及在哪里需要括号：

Listing 3.9. 带参数的函数符号

```

defer func(x int) {
    /* ... */
}(5)    ← 为输入参数x 赋值5

```

在这个（匿名）函数中，可以访问任何命名返回参数：

Listing 3.10. 在defer 中访问返回值

```

func f() (ret int) {    ← ret 初始化为零
    defer func() {
        ret++    ← ret 增加为1
    }()
    return 0    ← 返回的是1 而不是 0！
}

```

变参

接受变参的函数是有着不定数量的参数的。为了做到这点，首先需要定义函数使其接受变参：

```

func myfunc(arg ...int) {}

```

arg ... int 告诉Go 这个函数接受不定数量的参数。注意，这些参数的类型全部是int。在函数体中，变量arg 是一个int 的slice：

```

for _, n := range arg {
    fmt.Printf("And the number is: %d\n", n)
}

```

^b函数符号也就是被叫做闭包的东西。

如果不指定变参的类型，默认是空的接口`interface{}`（参阅第6章）。假设有另一个变参函数叫做`myfunc2`，下面的例子演示了如何向其传递变参：

```
func myfunc(arg ...int) {
    myfunc2(arg...)      ← 按原样传递
    myfunc2(arg[:2]...)  ← 传递片段
}
```

函数作为值

就像其他在Go中的几乎所有东西，函数也同样是值而已。它们可以像下面这样赋值给变量：

Listing 3.11. Anonymous function

```
func main() {
    a := func() {          ← 定义一个匿名函数，并且赋值给a
        println("Hello")
    }                      ← 这里没有()
    a()                    ← 调用函数
}
```

如果使用`fmt.Printf("%T\n", a)`打印`a`的类型，输出结果是`func()`。

函数作为值，也会被用在其他一些地方，例如`map`。这里将整数转换为函数：

Listing 3.12. 使用map的函数作为值

```
var xs = map[int]func() int{
    1: func() int { return 10 },
    2: func() int { return 20 },
    3: func() int { return 30 }, ← 必须有逗号
    /* ... */
}
```

也可以编写一个接受函数作为参数的函数，例如工作在`int`的`slice`上的`Map`函数。这是一个留给读者的练习，参考在第39页的练习Q12。

回调

当函数作为值时，就可以很容易的传递到其他函数里，然后可以作为回调。首先定义一个函数，对整数做一些“事情”：

```
func printit(x int) {      ← 函数无返回值
    fmt.Print("%v\n", x)   ← 仅仅打印
}
```

这个函数的标识是`func printit(int)`，或者没有函数名：`func(int)`。创建新的函数使用这个作为回调，需要用到这个标识：

```
func callback(y int, f func(int)) { ← f 将会保存函数
    f(y)                             ← 调用回调函数f 输入变量y
}
```

恐慌 (Panic) 和恢复 (Recover)

^c Go 没有例如像Java 那样的异常机制：不能抛出一个异常。作为代替，它使用了恐慌和恢复 (panic-and-recover) 机制。一定要记得，这应当作为最后的手段被使用，你的代码中应当没有，或者很少的令人恐慌的东西。这是个强大的工具，明智地使用它。那么，应该如何使用它。

下面的描述来自于[5]：

Panic

是一个内建函数，可以中断原有的控制流程，进入一个令人恐慌的流程中。当函数F 调用panic，函数F 的执行被中断，并且F 中的延迟函数会正常执行，然后F 返回到调用它的地方。在调用的地方，F 的行为就像调用了panic。这一过程继续向上，直到程序崩溃时的所有goroutine 返回。

恐慌可以直接调用panic 产生。也可以由运行时错误产生，例如访问越界的数组。

Recover

是一个内建的函数，可以让进入令人恐慌的流程中的goroutine 恢复过来。Recover 仅在延迟函数中有效。

在正常的执行过程中，调用recover 会返回nil 并且没有其他任何效果。如果当前的goroutine 陷入恐慌，调用recover 可以捕获到panic 的输入值，并且恢复正常的执行。

这个函数检查作为其参数的函数在执行时是否会产生panic^d：

```
func throwsPanic(f func()) (b bool) { ❶
defer func() { ❷
    if x := recover(); x != nil {
        b = true
    }
}()
f() ❸
return ❹
}
```

❶ 定义一个新函数throwsPanic 接受一个函数作为参数，参看“函数作为值”。当这个函数会产生panic，就返回true，否则返回false；

❷ 定义了一个利用recover 的defer 函数，如果当前的goroutine 产生了panic，这个defer 函数能够发现。如果recover() 返回非nil 值，设置b 为true；

❸ 调用作为参数接收的函数。

❹ 返回b 的值。由于b 是命名返回值（第33 页），无须指定b。

^c对应异常机制，Go 的这种错误机制或许可以叫做恐慌机制：当你遇到它时应该感到恐慌，然后应该恢复 (recover) 它。

^d复制于Eleanor McHugh 的演讲稿

练习

Q6. (4) 平均值

1. 编写一个函数用于计算一个 `float64` 类型的 slice 的平均值。

Q7. (3) 整数顺序

1. 编写函数，返回其（两个）参数正确的（自然）数字顺序：

$f(7, 2) \rightarrow 2, 7$

$f(2, 7) \rightarrow 2, 7$

Q8. (4) 作用域

1. 下面的程序有什么错误？

```
package main                                1

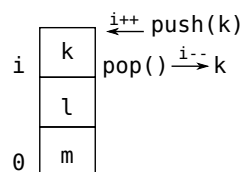
import "fmt"                                3

func main() {                                5
    for i := 0; i < 10; i++ {                6
        fmt.Printf("%v\n", i)                7
    }                                         8
    fmt.Printf("%v\n", i)                    9
}
```

Q9. (5) 栈

1. 创建一个固定大小保存整数的栈。它无须超出限制的增长。定义 `push` 函数——将数据放入栈，和 `pop` 函数从栈中取得内容。栈应当是后进先出（LIFO）的。

Figure 3.1. 一个简单的 LIFO 栈



2. 更进一步。编写一个 `String` 方法将栈转化为字符串形式的表达。可以这样的方式打印整个栈：`fmt.Printf("My stack %v\n", stack)`

栈可以被输出成这样的形式：`[0:m] [1:l] [2:k]`

Q10. (5) 变参

1. 编写函数接受整数类型变参，并且每行打印一个数字。

Q11. (5) 斐波那契

1. 斐波那契数列以：`1, 1, 2, 3, 5, 8, 13, ...` 开始。或者用数学形式表达： $x_1 = 1; x_2 = 1; x_n = x_{n-1} + x_{n-2} \quad \forall n > 2$ 。

编写一个函数，接受 `int` 值，并给出这个值得到的斐波那契数列。

Q12. (4) Map function `map()` 函数是一个接受一个函数和一个列表作为参数的函数。函数应用于列表中的每个元素，而一个新的包含有计算结果的列表被返回。因此：

$$\text{map}(f(), (a_1, a_2, \dots, a_{n-1}, a_n)) = (f(a_1), f(a_2), \dots, f(a_{n-1}), f(a_n))$$

1. 编写Go中的简单的`map()`函数。它能工作于操作整数的函数就可以了。
2. 扩展代码使其工作于字符串列表。

Q13. (3) 最小值和最大值

1. 编写一个函数，计算`int slice ([]int)`中的最大值。
2. 编写一个函数，计算`int slice ([]int)`中的最小值。

Q14. (5) 冒泡排序

1. 编写一个针对`int`类型的`slice`冒泡排序的函数。这里[37]：

它在一个列表上重复步骤来排序，比较每个相邻的元素，并且顺序错误的时候，交换它们。一遍一遍扫描列表，直到没有交换为止，这意味着列表排序完成。算法得名于更小的元素就像“泡泡”一样冒到列表的顶端。

[37] 这里有一个过程代码作为示例：

```
procedure bubbleSort( A : list of sortable items )
do
  swapped = false
  for each i in 1 to length(A) - 1 inclusive do:
    if A[i-1] > A[i] then
      swap( A[i-1], A[i] )
      swapped = true
    end if
  end for
while swapped
end procedure
```

Q15. (6) 函数返回一个函数

1. 编写一个函数返回另一个函数，返回的函数的作用是对一个整数+2。函数的名称叫做`plusTwo`。然后可以像下面这样使用：

```
p := plusTwo()
fmt.Printf("%v\n", p(2))
```

应该打印4。参阅第36页的回调了解更多关于这个的内容。

2. 使1中的函数更加通用化，创建一个`plusX(x)`函数，返回一个函数用于对整数加上`x`。

答案

A6. (4) 平均值

1. 下面的函数计算平均值。

Listing 3.13. Go 中的平均值函数

```
func average(xs []float64) (ave float64) { ❶
    sum := 0.0
    switch len(xs) {
    case 0: ❷
        ave = 0
    default: ❸
        for _, v := range xs {
            sum += v
        }
        ave = sum / float64(len(xs)) ❹
    }
    return ❺
}
```

- ❶ 可以使用命名返回值；
- ❷ 如果长度是零，返回0；
- ❸ 否则，计算平均值；
- ❹ 为了使除法能正常计算，必须将值转换为float64；
- ❺ 得到平均值，返回它

A7. (3) 整数顺序

1. 这里可以利用Go中的多返回值（参阅第“多个返回值”节）：

```
func order(a, b int) (int, int) {
    if a > b {
        return b, a
    }
    return a, b
}
```

A8. (4) 作用域

1. 这个程序不能被编译，由于第9行的变量i，未定义：i仅在for循环中有效。为了修正这个，main()应修改为：

```
func main() {
    var i int
    for i = 0; i < 10; i++ {
        fmt.Printf("%v\n", i)
    }
}
```

```

    }
    fmt.Printf("%v\n", i)
}

```

现在*i* 在**for** 循环外定义，并且在其后仍然可访问。这会打印数字从0 到10。

A9. (5) 栈

1. 首先定义一个新的类型来表达栈；需要一个数组（来保存键）和一个指向最后一个元素的索引。这个小栈只能保存10 个元素。

```

type stack struct {    ← 栈不应该被导出
    i    int
    data [10]int
}

```

然后需要push 和pop 函数来使用这个。首先展示一下错误的解法！在Go 的数据传递中，是值传递，意味着一个副本被创建并传递给函数。push 函数的第一个版本大约是这样：

```

func (s stack) push(k int) {    ← 工作于参数的副本
    if s.i+1 > 9 {
        return
    }
    s.data[s.i] = k
    s.i++
}

```

函数对**stack** 类型的变量*s* 进行处理。调用这个，只需要*s.push(50)*，将整数50 放入栈中。但是push 函数得到的是*s* 的副本，所以它不会有真正的结果。用这个方法，不会有内容放入栈中，例如下面的代码：

```

var s stack    ← 让s 是一个stack 变量
s.push(25)
fmt.Printf("stack %v\n", s);
s.push(14)
fmt.Printf("stack %v\n", s);

```

打印：

```

stack [0:0]
stack [0:0]

```

为了解决这个，需要向函数push 提供一个指向栈的指针。这意味着需要修改push

```

func (s stack)push(k int) → func (s *stack)push(k int)

```

应当使用new()（参阅第5 章第“用 new 分配内存”节）创建指针指向的**stack** 的空间，因此例子中的第1 行需要是 *s := new(stack)*

而两个函数变为：

```

func (s *stack) push(k int) {
    s.data[s.i] = k
    s.i++
}

```

```
func (s *stack) pop() int {
    s.i--
    return s.data[s.i]
}
```

像下面这样使用

```
func main() {
    var s stack
    s.push(25)
    s.push(14)
    fmt.Printf("stack %v\n", s)
}
```

2. 这里有一个额外的问题，对于这个练习中编写打印栈的代码的时候非常有价值。根据Go文档fmt.Printf("%v")可以打印实现了Stringer接口的任何值(%v)。为了使其工作，需要为类型定义一个String()函数：

Listing 3.14. stack.String()

```
func (s stack) String() string {
    var str string
    for i := 0; i <= s.i; i++ {
        str = str + "[" +
            strconv.Itoa(i) + ":" + strconv.Itoa(s.data[i]) + "]"
    }
    return str
}
```

A10. (5) 变参

1. 需要使用...语法来实现函数接受若干个数字作为变参。

Listing 3.15. 有变参的函数

```
package main

import "fmt"

func main() {
    printthem(1, 4, 5, 7, 4)
    printthem(1, 2, 4)
}

func printthem(numbers ...int) {    ← numbers 现在是整数类型的slice
    for _, d := range numbers {
        fmt.Printf("%d\n", d)
    }
}
```

A11. (5) 斐波那契

1. 下面的程序会计算出斐波那契数列。

Listing 3.16. Fibonacci function in Go

```
package main

import "fmt"

func fibonacci(value int) []int {
    x := make([]int, value) ❶
    x[0], x[1] = 1, 1 ❷
    for n := 2; n < value; n++ {
        x[n] = x[n-1] + x[n-2] ❸
    }
    return x ❹
}

func main() {
    for _, term := range fibonacci(10) { ❺
        fmt.Printf("%v ", term)
    }
}
```

- ❶ 创建一个用于保存函数执行结果的array；
- ❷ 开始计算斐波那契数列；
- ❸ $x_n = x_{n-1} + x_{n-2}$ ；
- ❹ 返回整个 array；
- ❺ 使用保留字range可以逐个得到斐波那契函数返回的序列。这里有10个。并且打印出来了。

A12. (4) Map function

Listing 3.17. Map 函数

```
1. func Map(f func(int) int, l []int) []int {
    j := make([]int, len(l))
    for k, v := range l {
        j[k] = f(v)
    }
    return j
}

func main() {
    m := []int{1, 3, 4}
    f := func(i int) int {
        return i * i
    }
}
```

```

    }
    fmt.Printf("%v", (Map(f, m)))
}

```

2. 字符串问题的答案

A13. (3) 最小值和最大值

1. 这个函数用于计算最大值：

```

func max(l []int) (max int) { ❶
    max = l[0]
    for _, v := range l { ❷
        if v > max { ❸
            max = v
        }
    }
    return ❹
}

```

- ❶ 使用了命名返回参数；
- ❷ 对`l`循环。元素的序号不重要；
- ❸ 如果找到了新的最大值，记住它；
- ❹ 一个遥远的返回，当前的`max` 值被返回。

2. 这个函数用于计算最小值，这几乎与`max` 完全一致。

```

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

```

有心的读者可能已经将`max` 和`min` 合成一个函数，用一个选择来判断是取最小值还是最大值，或者两个值都返回。

A14. (5) 冒泡排序

1. 冒泡排序并不是最有效率的，对于 n 个元素它的算法复杂度是 $O(n^2)$ 。快速排序[27] 是更好的排序算法。

但是冒泡排序容易实现，下面是一个例子。

Listing 3.18. Bubble sort

```

func main() {
    n := []int{5, -1, 0, 12, 3, 5}
}

```

```

        fmt.Printf("unsorted %v\n", n)
        bubblesort(n)
        fmt.Printf("sorted %v\n", n)
    }

    func bubblesort(n []int) {
        for i := 0; i < len(n) - 1; i++ {
            for j := i + 1; j < len(n); j++ {
                if n[j] < n[i] {
                    n[i], n[j] = n[j], n[i]
                }
            }
        }
    }
}

```

由于slice 是一个引用类型，bubblesort 函数可以工作，并且无须返回排序后的slice。

A15. (6) 函数返回一个函数

```

1. func main() {
    p2 := plusTwo()
    fmt.Printf("%v\n", p2(2))
}

func plusTwo() func(int) int { ❶
    return func(x int) int { return x + 2 } ❷
}

```

❶ 定义新的函数返回一个函数。看看你写的跟要表达的意思是如何的；

❷ 函数符号，在返回语句中定义了一个+2 的函数。

2. 这里我们使用闭包：

```

func plusX(x int) func(int) int { ❶
    return func(y int) int { return x + y } ❷
}

```

❶ 再次定义一个函数返回一个函数；

❷ 在函数符号中使用局部变量x。

4 包

“^

对是否有按位非的运算符的回答。”

KEN THOMPSON

包是函数和数据的集合，跟Perl的包[15]差不多。用**package**保留字定义一个包。文件名不需要与包名一致。包名的约定是使用小写字母。Go包可以由多个文件组成，但是使用相同的**package** <name> 这一行。让我们在文件even.go中定义一个叫做even的包。

Listing 4.1. A small package

```
package even          ← 开始自定义的包

func Even(i int) bool { ← 可导出函数
    return i % 2 == 0
}

func odd(i int) bool {  ← 私有函数
    return i % 2 == 1
}
```

名称以大写字母起始的是可导出的，可以在包的外部调用，稍后会讨论这个。现在可以在下面的程序myeven.go中使用这个包了：

Listing 4.2. Use of the even package

```
package main

import (                ❶
    "./even"            ❷
    "fmt"                ❸
)

func main() {
    i := 5
    fmt.Printf("Is %d even? %v\n", i, even.Even(i)) ❹
}
```

- ❶ 导入下面的包；
- ❷ 本地包even 在这里导入；
- ❸ 官方fmt 包导入；
- ❹ 调用even 包中的函数。访问一个包中的函数的语法是<package>.Function()。

现在只需要编译和链接，首先是那个包，然后是`myeven.go`，并且链接它们：

```
% 6g even.go      ← 包
% 6g myeven.go    ← 程序
% 6l -o myeven myeven.6      ← 链接步骤
```

然后测试一下：

```
% ./myeven
Is 5 even? false
```

在Go中，当函数的首字母大写的时候，函数会被从包中导出（在包外部可见，或者说公有的），因此函数名是`Even`。如果修改`myeven.go`的第7行，使用未导出的函数`even.odd`：

```
fmt.Printf("Is %d even? %v\n", i, even.odd(i))
```

由于使用了私有的函数，会得到一个编译错误：

```
myeven.go:7: cannot refer to unexported name even.odd
myeven.go:7: undefined: even.odd
```

概括来说：

- 公有函数的名字以大写字母开头；
- 私有函数的名字以小写字母开头。

这个规则同样适用于定义在包中的其他名字（新类型、全局变量）。注意，“大写”的含义并不仅限于US ASCII，它被扩展到了整个Unicode范围。所以大写的希腊语、古埃及语都是可以的。

构建一个包

为了创建一个别人可以用的包（只需要`import "even"`即可使用），首先需要创建一个目录放入包文件。

```
% mkdir even
% cp even.go even/
```

然后可以使用根据`even`包改写的Makefile文件。

Listing 4.3. 用于包的Makefile

```
include $(GOROOT)/src/Make.inc      1

TARG=even                          3
GOFILES=\                          4
    even.go\                        5

include $(GOROOT)/src/Make.pkg      7
```

注意第3行定义了`even`包，而第4和第5行录入了构成包的文件。同样注意，这不是在第2章，第“使用 Makefile”节中使用的那个 Makefile。最后一行的`include`语句是不同的。^a

如果现在执行`gomake`，一个叫做“`_go_.6`”的文件、一个叫做“`_obj/`”的目录，和“`_obj/`”目录中，叫做“`even.a`”的文件被创建。文件`even.a`是有编译后的Go代码的静态库。通过`gomake install`包（好吧，只有`even.a`）被安装在官方包目录中：

```
% make install
cp _obj/even.a $GOROOT/pkg/linux_amd64/even.a
```

安装完后，就可以修改`import`从“`./even`”到“`even`”。

但是，如果不希望将包安装到官方的Go目录，或者没有权限这么做的话。在使用6/8g的时候，可以指定`I`标志替换包目录。有了这个标志，你可以让你的导入语句变成（`import "even"`）却继续部署在自定义的目录中。所以，下面的命令可以联合包构建（并且链接）`myeven`程序。

```
% 6g -I even/_obj myeven.go    ← Building
% 6l -L even/_obj myeven.6     ← Linking
```

导入但是没有使用的包会引起一个错误。

标识符

像在其他语言中一样，Go的命名是很重要的。在某些情况下，它们甚至有语义上的作用：例如，在包外是否可见决定于首字母是不是大写。因此有必要花点时间讨论一下Go程序的命名规则。

使用的规则是让众所周知的缩写保持原样，而不是去尝试到底哪里应该大写。`Atoi`，`Getwd`，`Chmod`。

驼峰式对那些有完整单词的会很好：`ReadFileNewWriterMakeSlice`。

包名

当包导入（通过`import`）时，包名成为了内容的入口。在

```
import "bytes"
```

之后，导入包的可以调用函数`bytes.Buffer`。任何使用这个包的人，可以使用同样的名字访问到它的内容，因此这样的包名是好的：短的、简洁的、好记的。根据规则，包名是小写的一个单词；不应当有下划线或混合大小写。由于每个人都可能需要录入这个名字，所以尽可能的简短。不要提前考虑冲突。包名是导入的默认名称。就上面的`bytes.Buffer`来说。通过

```
import bar "bytes"
```

它变成了`bar.Buffer`。因此，它无须在整个代码中唯一，在少有的冲突中，可以给导入的包选择另一个名字在局部使用。在任何时候，冲突都是很少见的，因为导入的文件名会用来做判断，到底是哪个包使用了。

^a在Makefile文件中可以访问下面的预定义变量：`$(GC)`标示Go编译器（6g/8g），`$(LD)`当前链接器，以及`$0`作为预链接二进制文件的后缀（6/8）。

另一个规则是包名就是代码的根目录名；在`src/pkg/compress/gzip`的包，作为`compress/gzip`导入，但名字是`gzip`，不是`compress_gzip`也不是`compressGzip`。

导入包将使用其名字引用到内容上，所以导入的包可以利用这个避免罗嗦。例如，缓冲类型`bufio`包的读取方法，叫做`Reader`，而不是`BufReader`，因为用户看到的是`bufio.Reader`这个清晰、简洁的名字。更进一步说，由于导入的实例总是它们包名指向的地址，`bufio.Reader`不会与`io.Reader`冲突。类似的，`ring.Ring`（包`container/ring`）创建新实例的函数——在Go中定义的构造函数——通常叫做`NewRing`，但是由于`Ring`是这个包唯一的一个导出的类型，同时，这个包也叫做`ring`，所以它可以只称作`New`。包的客户看到的是`ring.New`。用包的结构帮助你选择更好的名字。

另外一个简短的例子是`once.Do`（参看`sync`）；`once.Do(setup)`读起来很不错，并且命名为`once.DoOrWaitUntilDone(setup)`不会有任何帮助。长的名字不会让其变得容易阅读。如果名字表达了一些复杂并且微妙的内容，更好的办法是编写一些有帮助的注释，而不是将所有信息都放入名字里。

最后，在Go中使用混合大小写 `MixedCaps` 或者 `mixedCaps`，而不是下划线区分含有多个单词的名字。

包的文档

这段复制于[6]。

每个包都应该有包注释，在`package`前的一个注释块。对于多文件包，包注释只需要出现在一个文件前，任意一个文件都可以。包注释应当对包进行介绍，并提供关于包的整体信息。这会出现于`godoc`生成的关于包的页面上，并且相关的细节会跟随气候。来自官方`regexp`包的例子：

```
/*
    The regexp package implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation '|' concatenation
*/
package regexp
```

每个定义（并且导出）的函数应当有一小段文字描述该函数的行为，来自于`fmt`包：

```
// Printf formats according to a format specifier and writes to standard output.
func Printf(format string, a ...interface{}) (n int, errno os.Error) {
```

测试包

在Go中为包编写单元测试应当是一种习惯。编写测试需要包含`testing`包和程序`gotest`。两者都有良好的文档。当对某个包进行测试时，务必要记得必须使用`Makefile`进行编译（参阅第“构建一个包”节）。

测试本身是由`gotest`完成的。`gotest`程序调用了所有的测试函数。`even`包没有定义任何测试函数，执行`gotest`，这样：

```
% gotest
gotest: no test files found in current directory
```

在测试文件中定义一个测试来修复这个。测试文件也在包目录中，被命名为*_test.go。这些测试文件同Go程序中的其他文件一样，但是gotest只会执行测试函数。每个测试函数都有相同的标识，它的名字以Test开头：

```
func TestXxx(t *testing.T)    ← Test<Capital>restOftheNe
```

编写测试时，需要告诉gotest测试是失败还是成功。测试成功则直接返回。当测试失败可以用下面的函数标记[9]。这是非常重要的（参阅godoc testing了解更多）：

```
func (t *T) Fail()
```

Fail 标记测试函数失败，但仍然继续执行。

```
func (t *T) FailNow()
```

FailNow 标记测试函数失败，并且中断其执行。这将会执行下一个测试。因此，当前文件的其他所有测试都被跳过。

```
func (t *T) Log(args ...interface{})
```

Log 用默认格式对其参数进行格式化，与Print()类似，并且记录文本到错误日志。

```
func (t *T) Fatal(args ...interface{})
```

Fatal 等价于Log()后跟随FailNow()。

将这些凑到一起，就可以编写测试了。首先，选择名字even_test.go。然后添加下面的内容：

Listing 4.4. even 包的测试

```
package even                                                    1

import "testing"                                                3

func TestEven(t *testing.T) {                                   5
    if ! Even(2) {                                              6
        t.Log("2 should be even!")                             7
        t.Fail()                                               8
    }                                                            9
}                                                                10
```

注意在第一行使用了package even，测试使用与被测试的包使用相同的名字空间。这不仅仅是为了方便，也允许了测试未导出的函数和结构。然后导入testing包，并且在第5行定义了这个文件中唯一的测试函数。展示的Go代码应当没有任何惊异的地方：检查了Even函数是否工作正常。现在等待了好久的时刻到了，执行测试：

```
% gotest
6g -o _gotest_.6 even.go even_test.go
rm -f _test/even.a
gopack grc _test/even.a _gotest_.6
PASS
```

测试执行并且报告PASS。成功了！

为了展示失败的测试，修改测试函数：

```
// Entering the twilight zone
func TestEven(t *testing.T) {
    if Even(2) {
        t.Log("2 should be odd!")
        t.Fail()
    }
}
```

然后得到：

```
--- FAIL: even.TestEven
    2 should be odd!
FAIL
gotest: "./6.out" failed: exit status 1
```

然后你可以以此行事（修复测试的实例）

在编写包的时候应当一边写代码，一边写（一些）文档和测试函数。这可以让你的程序更好，并且它展示了你的努力。

常用的包

标准的Go 代码库中包含了大量的包，并且在安装Go 的时候多数会伴随一起安装。浏览\$GOROOT/src/pkg 目录并且查看那些包会非常有启发。无法对每个包就加以解说，不过下面的这些值得讨论：^b

fmt

包*fmt* 实现了格式化的I/O 函数，这与C 的printf 和scanf 类似。格式化短语派生于C 。一些短语（%-序列）这样使用：

```
%v          默认格式的值。当打印结构时，加号（%+v）会增加字段名；
%#v         Go 样式的值表达；
%T          带有类型的Go 样式的值表达；
```

io

这个包提供了原始的I/O 操作界面。它主要的任务是对os 包这样的原始的I/O 进行封装，增加一些其他相关，使其具有抽象功能用在公共的接口上。

bufio

这个包实现了缓冲的I/O。它封装于io.Reader 和io.Writer 对象，创建了另一个对象（Reader 和Writer）在提供缓冲的同时实现了一些文本I/O 的功能。

^b描述来自包的godoc。额外的解释用斜体。

sort

sort 包提供了对数组和用户定义集合的原始的排序功能。

strconv

strconv 包提供了将字符串转换成基本数据类型，或者从基本数据类型转换为字符串的功能。

os

os 包提供了与平台无关的操作系统功能接口。其设计是Unix形式的。

sync

sync 包提供了基本的同步原语，例如互斥锁。

flag

flag 包实现了命令行解析。参阅“命令行参数”在第94页。

json

json 包实现了编码与解码RFC 4627 [22] 定义的JSON对象。

template

数据驱动模板，用于生成文本输出，例如HTML。

将模板关联到某个数据结构上进行解析。模板内容指向数据结构的元素（通常结构的字段或者map的键）控制解析并且决定某个值会被显示。模板扫描结构以便解析，而“游标”@ 决定了当前位置在结构中的值。

http

http 实现了HTTP请求、响应和URL的解析，并且提供了可扩展的HTTP服务和基本的HTTP客户端。

unsafe

unsafe 包包含了Go程序中数据类型上所有不安全的操作。通常无须使用这个。

reflect

reflect 包实现了运行时反射，允许程序通过抽象类型操作对象。通常用于处理静态类型 `interface{}` 的值，并且通过 `Typeof` 解析出其动态类型信息，通常会返回一个有接口类型 `Type` 的对象。包含一个指向类型的指针，`*StructType`、`*IntType` 等等，描述了底层类型的详细信息。可以用于类型转换或者类型赋值。参阅6，第“自省和反射”节。

exec

exec 包执行外部命令。

练习

Q16. (2) stack 包

1. 参考Q9练习。在这个练习中将从那个代码中建立一个独立的包。为stack的实现创建一个合适的包，`Push`、`Pop` 和 `Stack` 类型需要被导出。
2. 为这个包编写一个单元测试，至少测试 `Push` 后 `Pop` 的工作情况。
3. 还有哪个官方提供的包可以做为（FIFO）stack使用？

Q17. (7) 计算器

1. 使用stack 包创建逆波兰计算器。
2. 扩展一下，用你在问题3 中发现的包重写计算器。

答案

A16. (2) stack 包

1. 在创建stack包时，仅有一些小细节需要修改。首先，导出的函数应当大写首字母，因此应该是**Stack**。完整的包变为：

Listing 4.5. 包里的Stack

```
package stack

type Stack struct {
    i    int
    data [10]int
}

func (s *Stack) Push(k int) {
    s.data[s.i] = k
    s.i++
}

func (s *Stack) Pop() (ret int) {
    s.i--
    ret = s.data[s.i]
    return
}
```

2. 为了让单元测试正常工作，需要做一些准备。下面用一分钟的时间来做这些。首先是单元测试本身。创建文件“pushpop_test.go”，有如下内容：

Listing 4.6. Push/Pop 测试

```
package stack
import "testing"
func TestPushPop(t *testing.T) {
    c := new(Stack)
    c.Push(5)
    if c.Pop() != 5 {
        t.Log("Pop doesn't give 5")
        t.Fail()
    }
}
```

因为gotest调用gomake编译包，所以需要有一个makefile:

Listing 4.7. 包的Makefile

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

```

stack-as-package.go\

include $(GOROOT)/src/Make.pkg

完成了这些工作，可以调用gotest来测试包：
%gotest
rm -f _test/stack.a
6g -o _gotest_.6 stack-as-package.go pushpop_test.go
rm -f _test/stack.a
gopack grc _test/stack.a _gotest_.6
PASS

```

3. 应当是包`container/vector`。

A17. (7) 计算器

1. 这是第一个答案：

Listing 4.8. 逆波兰计算器

```

package main

import ( "bufio"; "os"; "strconv"; "fmt")

var reader *bufio.Reader = bufio.NewReader(os.Stdin)
var st = new(Stack)

type Stack struct {
    i    int
    data [10]int
}

func (s *Stack) push(k int) {
    if s.i+1 > 9 { return }
    s.data[s.i] = k
    s.i++
}

func (s *Stack) pop() (ret int) {
    s.i--
    if s.i < 0 { s.i = 0; return }
    ret = s.data[s.i]
    return
}

func main() {
    for {
        s, err := reader.ReadString('\n')
        var token string
        if err != nil { return }
        for _, c := range s {
            switch {

```

```

        case c >= '0' && c <= '9':
            token = token + string(c)
        case c == ' ':
            r, _ := strconv.Atoi(token)
            st.push(r)
            token = ""
        case c == '+':
            fmt.Printf("%d\n", st.pop()+st.pop())
        case c == '*':
            fmt.Printf("%d\n", st.pop()*st.pop())
        case c == '-':
            p := st.pop()
            q := st.pop()
            fmt.Printf("%d\n", q-p)
        case c == 'q':
            return
        default:
            //error
    }
}
}
}

```

2. *container/vector* 包应当是不错的选择。它同样预定义了Push和Pop函数。对于我们的程序来说修改是非常小的，下面的差异文件显示了不同的地方：

```

--- calc.go      2010-05-16 10:19:13.886855818 +0200
+++ calcvec.go   2010-05-16 10:13:35.000000000 +0200
@@ -5,11 +5,11 @@
     "os"
     "strconv"
     "fmt"
-    "./stack"
+    "container/vector"
 )

 var reader *bufio.Reader = bufio.NewReader(os.Stdin)
-var st = new(Stack)
+var st = new(vector.IntVector)

 func main() {
     for {

```

只有两行需要修改。太棒了。

5

进阶

“Go 有指针，但是没有指针运算。你不能用指针变量遍历字符串的各个字节。”

Go For C++ Programmers
GO AUTHORS

Go 有指针。然而却没有指针运算，因此它们更象是引用而不是你所知道的来自于C 的指针。指针非常有用。在Go 中调用函数的时候，得记得变量是值传递的。因此，为了修改一个传递入函数的值的效率和可能性，有了指针。

通过类型作为前缀来定义一个指针*：**var p *int**。现在p 是一个指向整数值的指针。所有新定义的变量都被赋值为其类型的零值，而指针也一样。一个新定义的或者没有任何指向的指针，有值nil。在其他语言中，这经常被叫做空（NULL）指针，在Go 中就是nil。让指针指向某些内容，可以使用取址操作符（&），像这样：

Listing 5.1. Use of a pointer

```
var p *int
fmt.Printf("%v", p)    ← 打印nil

var i int              ← 定义一个整型变量i
p = &i                 ← 使得p 指向i

fmt.Printf("%v", p)    ← 打印出来的内容类似0x7ff96b81c000a
```

从指针获取值是通过在指针变量前置*实现的：

Listing 5.2. 获取指针指向的值

```
p = &i                ← 获取i 的地址
*p = 8                ← 修改i 的值
fmt.Printf("%v\n", *p) ← 打印8
fmt.Printf("%v\n", i)  ← 同上
```

前面已经说了，没有指针运算，所以如果这样写：***p++**，它表示**(*p)++**：首先获取指针指向的值，然后对这个值加一。^a

内存分配

Go 同样也垃圾收集，也就是说无须担心内存分配和回收。

Go 有两个内存分配原语，**new** 和**make**。它们应用于不同的类型，做不同的工作，可能有些迷惑人，但是规则很简单。下面的章节展示了在Go 中如何处理内存分配，并且希望能够让**new** 和**make** 之间的区别更加清晰。

^a参看练习18。

用new 分配内存

内建函数`new`本质上说跟其他语言中的同名函数功能一样：`new(T)` 分配了零值填充的`T` 类型的内存空间，并且返回其地址，一个`*T` 类型的值。用Go 的术语说，它返回了一个指针，指向新分配的类型`T` 的零值。有一点非常重要：

| `new` 返回指针。

这意味着使用者可以用`new` 创建一个数据结构的实例并且可以直接工作。如`bytes.Buffer` 的文档所述“Buffer 的零值是一个准备好了的空缓冲。”类似的，`sync.Mutex` 也没有明确的构造函数或`Init` 方法。取而代之，`sync.Mutex` 的零值被定义为非锁定的互斥量。

零值是非常有用的。例如这样的类型定义，63 页的“定义自己的类型” 内容。

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

`SyncedBuffer` 的值在分配内存或定义之后立刻就可以使用。在这个片段中，`p` 和`v` 都可以在没有任何更进一步处理的情况下工作。

```
p := new(SyncedBuffer)    ← Type *SyncedBuffer, 已经可以使用
var v SyncedBuffer        ← Type SyncedBuffer, 同上
```

用make 分配内存

回到内存分配。内建函数`make(T, args)` 与`new(T)` 有着不同的功能。它只能创建`slice`，`map` 和`channel`，并且返回一个有初始值（非零）的`T` 类型，而不是`*T`。本质来讲，导致这三个类型有所不同的原因是指向数据结构的引用在使用前必须被初始化。例如，一个`slice`，是一个包含指向数据（内部`array`）的指针，长度和容量的三项描述符；在这些项目被初始化之前，`slice` 为`nil`。对于`slice`，`map` 和`channel`，`make` 初始化了内部的数据结构，填充适当的值。

| `make` 返回初始化后的（非零）值。

例如，`make([]int, 10, 100)` 分配了100 个整数的数组，然后用长度10 和容量100 创建了`slice` 结构指向数组的前10 个元素。区别是，`new([]int)` 返回指向新分配的内存的指针，而零值填充的`slice` 结构是指向`nil` 的`slice` 值。

这个例子展示了`new` 和`make` 的不同。

```
var p *[]int = new([]int)    ← 分配slice 结构内存；*p == nil
                             ← 已经可用
var v []int = make([]int, 100) ← v 指向一个新分配的有100 个整数的数组。

var p *[]int = new([]int)    ← 不必要的复杂例子
*p = make([]int, 100, 100)

v := make([]int, 100)        ← 更常见
```

务必记得`make` 仅适用于`map`，`slice` 和`channel`，并且返回的不是指针。应当用`new` 获得特定的指针。

`new` 分配；`make` 初始化

上面的两段可以简单总结为：

- `new(T)` 返回 `*T` 指向一个零值 `T`
- `make(T)` 返回初始化后的 `T`

当然 `make` 仅适用于 `slice`，`map` 和 `channel`。

构造函数与复合声明

有时零值不能满足需求，必须要有一个用于初始化的构造函数，例如这个来自 `os` 包的例子。

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

有许多冗长的内容。可以使用复合声明使其更加简洁，每次只用一个表达式创建一个新的实例。

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}    ← Create a new File
    return &f                    ← Return the address of f
}
```

返回本地变量的地址没有问题；在函数返回后，相关的存储区域仍然存在。

事实上，从复合声明获取分配的实例的地址更好，因此可以最终将两行缩短到一行。^b

```
return &File{fd, name, nil, 0}
```

The items (called of a composite +literal are laid out in order and must all be 所有的项目（称作字段）都必须按顺序全部写上。然而，通过对元素用字段:值成对的标识，初始化内容可以按任意顺序出现，并且可以省略初始化为零值的字段。因此可以这样

```
return &File{fd: fd, name: name}
```

在特定的情况下，如果复合声明不包含任何字段，它创建特定类型的零值。表达式 `new(File)` 和 `&File{}` 是等价的。

^b从复合声明中获取地址，意味着告诉编译器在堆中分配空间，而不是栈中。

复合声明同样可以用于创建array，slice 和map，通过指定适当的索引和map 键来标识字段。在这个例子中，无论是Enone，Eio 还是Eintval 初始化都能很好的工作，只要确保它们不同就好了。

```
ar := [...]string    {Enone: "no error", Eintval: "invalid argument"}
sl := []string        {Enone: "no error", Eintval: "invalid argument"}
ma := map[int]string{Enone: "no error", Eintval: "invalid argument"}
```

定义自己的类型

自然，Go 允许定义新的类型，通过保留字**type** 实现：

```
type foo int
```

创建了一个新的类型foo 作用跟**int** 一样。创建更加复杂的类型需要用到**struct** 保留字。这有个在一个数据结构中记录某人的姓名（**string**）和年龄（**int**），并且使其成为一个新的类型的例子：

Listing 5.3. Structures

```
package main

import "fmt"

type NameAge struct {
    name string    ← 不导出
    age  int        ← 不导出
}

func main() {
    a := new(NameAge)
    a.name = "Pete"; a.age = 42

    fmt.Printf("%v\n", a)
}
```

通常，`fmt.Printf("%v\n", a)` 的输出是

```
&{Pete 42}
```

这很棒！Go 知道如何打印结构。如果仅想打印某一个，或者某几个结构中的字段，需要使用`<field name>`。例如，仅仅打印名字：

```
fmt.Printf("%s", a.name)    ← %s 格式化字符串
```

结构字段

之前已经提到结构中的项目被称为field。没有字段的结构：**struct {}**

或者有四个^c字段的：

^c是的，四 (4) 个。


```
struct {
    x, y int
    A *[]int
    F func()
}
```

如果省略字段的名字，可以创建匿名字段，例如：

```
struct {
    T1          ← 字段名字是T1
    *T2         ← 字段名字是T2
    P.T3        ← 字段名字是T3
    x, y int     ← 字段名字是x 和y
}
```

注意首字母大写的字段可以被导出，也就是说，在其他包中可以进行读写。字段名以小写字幕开头是当前包的私有的。包的函数定义是类似的，参阅第4章了解更多细节。

方法

可以对新定义的类型创先函数以便操作，可以通过两种途径：

1. 创建一个函数接受这个类型的参数。

```
func doSomething(in1 *NameAge, in2 int) { /* ... */ }
```

（你可能已经猜到了）这是函数调用。

2. 创建一个工作在这个类型上的函数（参阅在3.1中定义的接收方）：

```
func (in1 *NameAge) doSomething(in2 int) { /* ... */ }
```

这是方法调用，可以类似这样使用：

```
var n *NameAge
n.doSomething(2)
```

使用函数还是方法完全是由程序员说了算，但是若需要满足接口（参看下一章）就必须使用方法。如果没有这样的需求，那就完全由习惯来决定是使用函数还是方法了。

但是下面的内容一定要留意，引用自[8]：

如果 x 可获取地址，并且 $\&x$ 的方法中包含了 m ， $x.m()$ 是 $(\&x).m()$ 更短的写法。

根据上面所述，这意味着下面的情况不是错误：

```
var n NameAge          ← 不是指针
n.doSomething(2)
```

这里Go会查找**NameAge**类型的变量 n 的方法列表，没有找到就会再查找***NameAge**类型的方法列表，并且将其转化为 $(\&n).doSomething(2)$ 。

下面的类型定义中有一些微小但是很重要的不同之处。同时可以参阅[8, section "Type Declarations"]。假设有：

```
// Mutex 数据类型有两个方法，Lock 和Unlock。  
type Mutex struct           { /* Mutex 字段 */ }  
func (m *Mutex) Lock()      { /* Lock 实现 */ }  
func (m *Mutex) Unlock()    { /* Unlock 实现 */ }
```

现在用两种不同的风格创建了两个数据类型。

- `type NewMutex Mutex;`
- `type PrintableMutex struct {Mutex }.`

现在NewMutex 等同于Mutex，但是它没有任何Mutex 的方法。换句话说，它的方法是空的。
但是PrintableMutex 已经从Mutex 继承了方法集合。如同[8]所说：

**PrintableMutex 的方法集合包含了Lock 和Unlock 方法，被绑定到其匿名字段Mutex。*

转换

有时需要将一个类型转换为另一个类型。在Go 中可以做到，不过有一些规则。首先，将一个值转换为另一个是由操作符（看起来像函数：byte()）完成的，并且不是所有的转换都是允许的。

Table 5.1. 合法的转换，float64 同float32 类似

From	xb []byte	xi []int	s string	f float32	i int
To					
[]byte	×		[]byte(s)		
[]int		×	[]int(s)		
string	string(xb)	string(xi)	×		
float32				×	float32(i)
int				int(f)	×

- 从string 到字节或者整形的slice。

```
mystring := "hello this is string"
```

```
byteslice := []byte(mystring)
```

转换到byte slice，每个byte 保存字符串对应字节的整数值。注意Go 的字符串是UTF-8 编码的，一些字符可能是1、2、3 或者4 个字节结尾。

```
intslice := []int(mystring)
```

转换到int slice，每个int 保存Unicode 编码的指针。字符串中的每个字符对应一个整数。

- 从字节或者整形的slice 到string。

```

b := []byte{'h','e','l','l','o'}    ← 复合声明
s := string(b)
i := []int{257,1024,65}
r := string(i)

```

对于数值，定义了下面的转换：

- 将整数转换到指定的 (bit) 长度：`uint8(int)`；
- 从浮点数到整数：`int(float32)`。这会截断浮点数的小数部分；
- 其他的类似：`float32(int)`。

用户定义类型的转换

如何在自定义类型之间进行转换？这里创建了两个类型**Foo**和**Bar**，而**Bar**是**Foo**的一个别名：

```

type foo struct { int }    ← 匿名字段
type bar foo               ← bar 是foo 的别名

```

然后：

```

var b bar = bar{1}        ← 声明b 为bar 类型
var f foo = b              ← 赋值b 到f

```

最后一行会引起错误：

cannot use b (type bar) as type foo in assignment(不能使用b (类型bar) 作为类型foo 赋值)

这可以通过转换来修复：

```
var f foo = foo(b)
```

注意转换那些字段不一致的结构是相当困难的。同时注意，转换**b**到**int**同样会出错；整数与有整数字段的结构并不一样。

练习

Q18. (4) 指针运算

1. 在正文的第60 页有这样的文字：

...这里没有指针运算，因此如果这样写：`*p++`，它被解释为`(*p)++`：首先解析引用然后增加值。

当像这样增加一个值的时候，什么类型可以工作？

2. 为什么它不能工作在所有类型上？

Q19. (6) 使用interface 的map 函数

1. 使用练习Q12 的答案，利用interface 使其更加通用。

Q20. (6) 指针

1. 假设定义了下面的结构：

```
type Person struct {
    name string
    age  int
}
```

下面两行之间的区别是什么？

```
var p1 Person
p2 := new(Person)
```

2. 下面两个内存分配的区别是什么？

```
func Set(t *T) {
    x = t
}
```

和

```
func Set(t T) {
    x = &t
}
```

Q21. (6) 链表

1. 使用 *container/list* 包创建（双向）链表。将值1，2 和4 存入并打印。
2. 自行实现链表。然后做与问题1 相同的实现。

Q22. (6) Cat

1. 编写一个程序，模仿Unix 的cat 程序。对于不知道这个程序的人来说，下面的调用显示了文件blah 的内容：
- ```
% cat blah
```
2. 使其支持n 开关，用于输出每行的行号。

#### Q23. (8) 方法调用

1. 假设有下面的程序：

```
package main

import "container/vector"

func main() {
 k1 := vector.IntVector{}
 k2 := &vector.IntVector{}
 k3 := new(vector.IntVector)
 k1.Push(2)
 k2.Push(3)
 k3.Push(4)
}
```

k1，k2 和k3 的类型是什么？

2. 当前，这个程序可以编译并且运行良好。在不同类型的变量上Push 都可以工作。Push 的文档这样描述：

*func (p \*IntVector) Push(x int) Push 增加x 到向量的末尾。*

那么接受者应当是**\*IntVector** 类型，为什么上面的代码可以工作？

## 答案

### A18. (4) 指针运算

1. 这仅能工作于指向数字 (`int`, `uint` 等等) 的指针值。
2. `++` 仅仅定义在数字类型上, 同时由于在Go中没有运算符重载, 所以会在其他类型上失败 (编译错误)。

### A19. (6) 使用interface 的map 函数

*Listing 5.4. Go 中更加通用的map 函数*

```
1. package main
 import "fmt"

 /* 定义一个空的 interface 类型 */
 type e interface{}

 func mult2(f e) e {
 switch f.(type) {
 case int:
 return f.(int) * 2
 case string:
 return f.(string) + f.(string) + f.(string) + f.(string)
 }
 return f
 }

 func Map(n []e, f func(e) e) []e {
 m := make([]e, len(n))
 for k, v := range n {
 m[k] = f(v)
 }
 return m
 }

 func main() {
 m := []e{1, 2, 3, 4}
 s := []e{"a", "b", "c", "d"}
 mf := Map(m, mult2)
 sf := Map(s, mult2)
 fmt.Printf("%v\n", mf)
 fmt.Printf("%v\n", sf)
 }
```

### A20. (6) 指针

1. 第一行: `var p1 Person` 分配了Person-值 给p1. p1 的类型是Person。

第二行：`p2 := new(Person)` 分配了内存并且将指针赋值给 `p2`。`p2` 的类型是 `*Person`。

2. 在第二个函数中，`x` 指向一个新的（堆上分配的）变量 `t`，其包含了实际参数值的副本。

在第一个函数中，`x` 指向了 `t` 指向的内容，也就是实际上的参数指向的内容。

因此在第二个函数，我们有了“额外”的变量存储了相关值的副本。

#### A21. (6) 链表

- 1.
- 2.

#### A22. (6) Cat

1. 下面是 `cat` 的实现，同样支持 `n` 输出每行的行号。

Listing 5.5. `cat` 程序

```
package main

❶
import (
 "os"
 "fmt"
 "bufio"
 "flag"
)

var numberFlag = flag.Bool("n", false, "number each line") ❷

❸
func cat(r *bufio.Reader) {
 i := 1
 for {
 buf, e := r.ReadBytes('\n') ❹
 if e == os.EOF {
 break
 }
 if *numberFlag { ❺
 fmt.Fprintf(os.Stdout, "%5d %s", i, buf)
 i++
 } else { ❻
 fmt.Fprintf(os.Stdout, "%s", buf)
 }
 }
 return
}

func main() {
```

```

 flag.Parse()
 if flag.NArg() == 0 {
 cat(bufio.NewReader(os.Stdin))
 }
 for i := 0; i < flag.NArg(); i++ {
 f, e := os.Open(flag.Arg(i), os.O_RDONLY, 0)
 if e != nil {
 fmt.Fprintf(os.Stderr, "%s: error reading from\n",
 "%s: %s\n",
 os.Args[0], flag.Arg(i), e.String())
 continue
 }
 cat(bufio.NewReader(f))
 }
}

```

- ❶ 包含所有需要用到的包；
- ❷ 定义新的开关“n”，默认是关闭的。注意很容易写的帮助文本；
- ❸ 实际上读取并且显示文件内容的函数；
- ❹ 每次读一行；
- ❺ 如果到达文件结尾；
- ❻ 如果设定了行号，打印行号然后是内容本身；
- ❼ 否则，仅仅打印该行内容。

#### A23. (8) 方法调用

1. k1 的类型是 **vector.IntVector**。为什么？这里使用了符号{}，因此获得了类型的值。变量k2 是 **\*vector.IntVector**，因为获得了复合语句的地址 (&)。而最后的k3 同样是 **\*vector.IntVector** 类型，因为new 返回该类型的指针。
2. 在[8] 的“调用”章节，有这样的描述：

当x 的方法集合包含m，并且参数列表可以赋值给m 的参数，方法调用x.m() 是合法的。如果x 可以被地址化，而&x 的方法集合包含m，x.m() 可以作为(&x).m() 的省略写法。

换句话说，由于k1 可以被地址化，而**\*vector.IntVector** 具有 Push 方法，调用k1.Push(2) 被Go 转换为(&k1).Push(2) 来使型系统愉悦（也使你愉悦——现在你已经了解到这一点）。<sup>d</sup>

<sup>d</sup>参阅本章的第“方法”节。



# 6

## 接口

我对外科手术般进入我的身体总是有恐惧。你知道我说的是什么。

*eXistenZ*  
TED PIKUL

下面的内容来自[35]。是Ian Lance Taylor 编写的，他是Go的作者之一。

在Go中，保留字*interface* 被赋予了多种不同的含义。每个类型都有接口，意味着对那个类型定义了方法集合。这段代码定义了一个具有一个字段和两个方法的结构类型*s*。

Listing 6.1. 定义结构和结构的方法

```
type S struct { i int }
func (p *S) Get() int { return p.i }
func (p *S) Put(v int) { p.i = v }
```

也可以定义接口类型，仅仅是方法的集合。这里定义了一个有两个方法的接口*I*：

```
type I interface {
 Get() int
 Put(int)
}
```

对于接口*I*，*s* 是合法的实现，因为它定义了*I* 所需的两个方法。注意，即便是没有明确定义*s* 实现了*I*，这也是正确的。

Go 程序可以利用这个特点来实现接口的另一个含义，就是接口值：

```
func f(p I) { ❶
 fmt.Println(p.Get()) ❶
 p.Put(1) ❷
}
```

❶ 定义一个函数接受一个接口类型作为参数；

❶ *p* 实现了接口*I*，必须有*Get()* 方法；

❷ *Put()* 方法是类似的。

这里的变量*p* 保存了接口类型的值。因为*s* 实现了*I*，可以调用*f* 向其传递*s* 类型的值的指针：

```
var s S; f(&s)
```

获取*s* 的地址，而不是*s* 的值的原因，是因为在*s* 的指针上定义了方法，参阅上面的代码6.1。这并不是必须的——可以定义让方法接受值——但是这样的话*Put* 方法就不会像期望的那样工作了。

实际上，无须明确一个类型是否实现了一个接口意味着Go 实现了叫做duck typing[39] 的模式。这不是纯粹的duck typing，因为如果可能的话Go 编译器将对类型是否实现了接口进行

实现静态检查。然而，Go 确实有纯粹动态的方面，如可将一个接口类型转换到另一个。通常情况下，转换的检查是在运行时进行的。如果是非法转换——当在已有接口值中存储的类型值不匹配将要转换到的接口——程序会抛出运行时错误。

在Go 中的接口有着与许多其他编程语言类似的思路：C++ 中的纯抽象虚基类，Haskell 中的typeclasses 或者Python 中的duck typing。然而没有其他任何一个语言联合了接口值、静态类型检查、运行时动态转换，以及无须明确定义类型适配一个接口。这些给Go 带来的结果是，强大、灵活、高效和容易编写的。

到底是什么？

来定义另外一个类型同样实现了接口I：

```
type R struct { i int }
func (p *R) Get() int { return p.i }
func (p *R) Put(v int) { p.i = v }
```

函数f 现在可以接受R 个s 类型的变量。假设需要在函数f 中知道实际的类型。在Go 中可以使用type switch 得到。

```
func f(p I) {
 switch t := p.(type) { ❶
 case *S: ❷
 case *R: ❸
 case S: ❹
 case R: ❺
 default: ❻
 }
}
```

❶ 类型判断。在switch 语句中使用(type)。保存类型到变量t；

❷ p 的实际类型是S 的指针；

❸ p 的实际类型是R 的指针；

❹ p 的实际类型是S；

❺ p 的实际类型是R；

❻ 实现了I 的其他类型。

在switch 之外使用(type) 是非法的。类型判断不是唯一的运行时得到类型的方法。为了在运行时得到类型，同样可以使用“comma, ok”来判断一个接口类型是否实现了某个特定接口：

```
if t, ok := something.(I); ok {
 // 对于某些实现了接口I 的
 // t 是其所拥有的类型
}
+
```

确定一个变量实现了某个接口，可以使用：

```
t := something.(I)
```

## 空接口

由于每个类型都能匹配到空接口：`interface{}`。我们可以创建一个接受空接口作为参数的普通函数：

Listing 6.2. 用空接口作为参数的函数

```
func g(something interface{}) int {
 return something.(I).Get()
}
```

在这个函数中的`return something.(I).Get()`是有一点窍门的。值`something`具有类型`interface{}`，这意味着方法没有任何约束：它能包含任何类型。`.(I)`是类型断言，用于转换`something`到`I`类型的接口。如果有这个类型，则可以调用`Get()`函数。因此，如果创建一个`*S`类型的新变量，也可以调用`g()`，因为`*S`同样实现了空接口。

```
s = new(S)
fmt.Println(g(s));
```

调用`g`的运行不会出问题，并且将打印0。如果调用`g()`的参数没有实现`I`会带来一个麻烦：

Listing 6.3. 实现接口失败

```
i := 5 ← 声明i 是一个"该死的" int
fmt.Println(g(i))
```

这能编译，但是当运行的时候会得到：

```
panic: interface conversion: int is not main.I: missing method Get
```

这是绝对没问题，内建类型`int`没有`Get()`方法。

## 方法

方法就是有接收者的函数（参阅第3章）。

可以在任意类型上定义方法（除了非本地类型，包括内建类型：`int`类型不能有方法）。然而可以新建一个拥有方法的整数类型。例如：

```
type Foo int

func (self Foo) Emit() {
 fmt.Printf("%v", self)
}

type Emitter interface {
 Emit()
}
```

对那些非本地（定义在其他包的）类型也一样：

Listing 6.4. 扩展内建类型错误

```
func (i int) Emit() {
 fmt.Printf("%d", i)
}
```

不能定义新的方法  
在非本地类型int 上

Listing 6.5. 扩展非本地类型错误

```
func (a *net.AddrError) Emit() {
 fmt.Printf("%v", a)
}
```

不能定义新的方法  
在非本地类型net.AddrError 上

接口类型的方法

接口定义为一个方法的集合。方法包含实际的代码。换句话说，一个接口就是定义，而方法就是实现。因此，接收者不能定义为接口类型，这样做的话会引起invalid receiver type ... 的编译器错误。来自语言说明书[8] 的权威内容：

接收者类型必须是T 或\*T，这里的T 是类型名。T 叫做接收者基础类型或简称基础类型。基础类型一定不能使指针或接口类型，并且定义在与方法相同的包中。

接口指针

Pointers to interfaces

在Go 中创建指向接口的指针是无意义的。实际上创建接口值的指针也是非法的。在2010-10-13 的发布日志中进行的描述，使得没有任何余地怀疑这一事实：

语言的改变是使用指针指向接口值不再自动反引用指针。指向接口值的指针通常是低级的错误，而不是正确的代码。

这来自[7]。如果不是这个限制，这个代码：

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

就会复制标准输入到buf 的副本，而不是buf 本身。这看起来永远不会是一个期望的结果。

接口名字

根据规则，单方法接口命名为方法名加上-er 后缀：Reader，Writer，Formatter 等。有一堆这样的命名，高效的反映了它们职责和包含的函数名。Read，Write，Close，Flush，String 等等有着规范的声明和含义。为了避免混淆，除非有类似的声明和含义，否则不要让方法与这些重名。相反的，如果类型实现了与众所周知的类型相同的方法，那么就用相同的名字和声明；将字符串转换方法命名为String 而不是ToString。

文本复制于[6]。

简短的例子

回顾那个冒泡排序的练习（Q14），对整型数组排序：

```
func bubblesort(n []int) {
 for i := 0; i < len(n)-1; i++ {
 for j := i + 1; j < len(n); j++ {
 if n[j] < n[i] {
 n[i], n[j] = n[j], n[i]
 }
 }
 }
}
```

排序字符串的版本是类似的，除了函数的声明：

```
func bubblesortString(n []string) { /* ... */ }
```

基于此，可能会需要两个函数，每个类型一个。而通过使用接口可以让这个变得更加通用。来创建一个可以对字符串和整数进行排序的函数，这个例子的某些行是无法运行的：

```
func sort(i []interface{}) { ❶
 switch i.(type) { ❶
 case string: ❷
 // ...
 case int:
 // ...
 }
 return /* ... */ ❸
}
```

- ❶ 函数将接收一个空接口的slice；
- ❶ 使用type switch 找到输入参数实际的类型；
- ❷ 然后排序；
- ❸ 返回排序的slice。

但是如果用`sort([]int{1, 4, 5})`调用这个函数，会失败：`cannot use i (type []int) as type []interface in function argument`

这是因为Go 不能简单的将其转换为接口的slice。转换到接口是容易的，但是转换到slice 的开销就高了。

简单来说：Go 不能（隐式）转换为slice。

那么如何创建Go 形式的这些“通用”函数呢？用Go 隐式的处理来代替type switch 方式的类型推断吧。下面的步骤是必须的：

1. 定义一个有着若干排序相关的方法的接口类型（这里叫做**Sorter**）。至少需要获取slice 长度的函数，比较两个值的函数和交换函数；

```
type Sorter interface {
 Len() int ← len() 作为方法
 Less(i, j int) bool ← p[j] < p[i] 作为方法
 Swap(i, j int) ← p[i], p[j] = p[j], p[i] 作为方法
}
```

关于这个话题完整的邮件列表讨论可以在[29] 这里找到。

2. 定义用于排序slice的新类型。注意定义的是slice类型；

```
type Xi []int
type Xs []string
```

3. 实现Sorter接口的方法。整数的：

```
func (p Xi) Len() int { return len(p) }
func (p Xi) Less(i int, j int) bool { return p[j] < p[i] }
func (p Xi) Swap(i int, j int) { p[i], p[j] = p[j], p[i] }
```

和字符串的：

```
func (p Xs) Len() int { return len(p) }
func (p Xs) Less(i int, j int) bool { return p[j] < p[i] }
func (p Xs) Swap(i int, j int) { p[i], p[j] = p[j], p[i] }
```

4. 编写作用于Sorter接口的通用排序函数。

```
func Sort(x Sorter) { ❶
 for i := 0; i < x.Len() - 1; i++ { ❷
 for j := i + 1; j < x.Len(); j++ {
 if x.Less(i, j) {
 x.Swap(i, j)
 }
 }
 }
}
```

❶ x现在是Sorter类型；

❷ 使用定义的函数，实现了冒泡排序。

现在可以像下面这样使用通用的Sort函数：

```
ints := Xi{44, 67, 3, 17, 89, 10, 73, 9, 14, 8}
strings := Xs{"nut", "ape", "elephant", "zoo", "go"}
```

```
Sort(ints)
fmt.Printf("%v\n", ints)
Sort(strings)
fmt.Printf("%v\n", strings)
```

在接口中列出接口

看一下下面的接口定义，这个是来自包container/heap的：

```
type Interface interface {
 sort.Interface
 Push(x interface{})
 Pop() interface{}
}
```

这里有另外一个接口在`heap.Interface`的定义中被列出，这看起来有些古怪，但是这的确是正确的，要记得接口只是一些方法的列表。`sort.Interface`同样是这样一个列表，因此将其包含在接口内是毫无错误的。

### 自省和反射

在下面的例子中，了解一下定义在`Person`的定义中的“标签”（这里命名为“namestr”）。为了做到这个，需要`reflect`包（在Go中没有其他方法）。要记得，查看标签意味着返回类型的定义。因此使用`reflect`包来指出变量的类型，然后访问标签。

Listing 6.6. 使用反射自省

```
type Person struct {
 name string "namestr" ← "namestr" 是标签
 age int
}

p1 := new(Person) ← new 返回Person 的指针
ShowTag(p1) ← 调用ShowTag() 并传递指针

func ShowTag(i interface{}) {

 switch t := reflect.TypeOf(i); t.Kind() { ← Get type, switch on Kind()
 case reflect.Ptr: ← Its a pointer, hence a reflect.Ptr
 tag := t.Elem().Field(0).Tag ← 0 1 2
 }
```

`Elem` 返回`v`指向的值。如果`v`是空指针，`Elem`返回空值。

- ❶ We are dealing with a **Type** and according to the documentation<sup>a</sup>:

```
// Elem returns a type's element type.
// It panics if the type's Kind is not Array, Chan, Map, Ptr, or Slice.
Elem() Type
```

同样的在`t`使用`Elem()`得到了指针指向的值。

- ❷ We have now dereferenced the pointer and are "inside" our structure. We now use `Field(0)` to access the zeroth field;
- ❸ 结构`StructField`有成员`Tag`，返回字符串类型的标签名。因此，在第 $0^{th}$ 个字段上可以用`.Tag`访问这个名字：`Field(0).Tag`。这样得到了`namestr`。

为了让类型和值之间的区别更加清晰，看下面的代码：

Listing 6.7. 反射类型和值

```
func show(i interface{}) {
 switch t := i.(type) {
 case *Person:
```

<sup>a</sup>godoc reflect

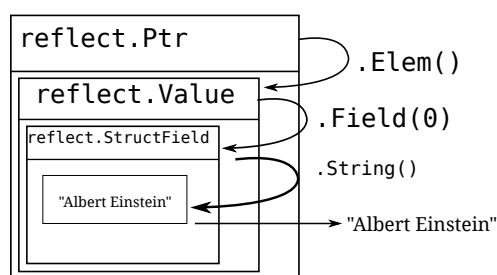
```

t := reflect.TypeOf(i) ← 得到类型的元数据
v := reflect.ValueOf(i) ← 得到实际的值
tag := t.Elem().Field(0).Tag ❶
name := v.Elem().Field(0).String() ❷
}

```

- ❶ 这里希望获得“标签”。因此需要Elem() 重定向至其上，访问第一个字段来获取标签。  
注意将t 作为一个reflect.Type 来操作；
- ❷ 现在需要访问其中一个成员的值，并让  
v 上的Elem() 进行重定向。这样就访问到了结构。然后访问第一个字段Field(0) 并且  
调用其上的String() 方法。

Figure 6.1. 用反射去除层次关系。通过Elem() 访问\*Person，使用godoc reflect 中描述的方法获得string 内部包含的内容。



设置值与获得值类似，但是仅仅工作在可导出的成员上。这些代码：

Listing 6.8. 私有成员的反射

```

type Person struct {
 name string "namestr" ← 名称
 age int
}

func Set(i interface{}) {
 switch i.(type) {
 case *Person:
 r := reflect.ValueOf(i)
 r.Elem(0).Field(0).
 SetString("Albert
 Einstein")
 }
}

```

Listing 6.9. 公有成员的反射

```

type Person struct {
 Name string "namestr" ← Name
 age int
}

func Set(i interface{}) {
 switch i.(type) {
 case *Person:
 r := reflect.ValueOf(i)
 r.Elem().Field(0).
 SetString("Albert
 Einstein")
 }
}

```

左边的代码可以编译并运行，但是当运行的时候，将得到打印了栈的运行错误：

```
panic: reflect.Value.SetString using value obtained using unexported field
```

右边的代码没有问题，并且设置了成员变量Name 为“Albert Einstein”。当然，这仅仅工作于调用Set() 时传递一个指针参数。



## 练习

### Q24. (6) 接口和编译

1. 在第74 页的代码6.3 编译正常——就像文中开始描述的那样。但是当运行的时候，会得到运行时错误，因此有些东西有错误。为什么代码编译没有问题呢？

### Q25. (5) 指针和反射

1. 在第自省和反射 节，第78 页的最后一段中，有这样的描述：

右边的代码没有问题，并且设置了成员变量`Name` 为“*Albert Einstein*”。  
当然，这仅仅工作于调用`Set()` 时传递一个指针参数。

为什么是这样的情况？

### Q26. (7) 接口和`max()`

1. 在练习Q13 中创建了工作于一个整形slice 上的最大函数。现在的问题是创建一个显示最大数字的程序，同时工作于整数和浮点数。虽然在这里会相当困难，不过还是让程序尽可能的通用吧。

## 答案

### A24. (6) 接口和编译

1. 代码能够编译是因为整数类型实现了空接口，这是在编译时检查的。

修复这个正确的途径是测试这个空接口可以被转换，如果可以，调用对应的方法。6.2 列出的Go 代码中定义了函数g——这里重复一下：

```
func g(any interface{}) int { return any.(I).Get() }
```

应当修改为：

```
func g(any interface{}) int {
 if v, ok := any.(I); ok { // 检查是否可以转换
 return v.Get() // 如果可以，调用Get()
 }
 return -1 // 随便返回个什么
}
```

如果现在调用g()，就不会有运行时错误了。在Go 中这种用法被称作“comma ok”。

### A25. (5) 指针和反射

1. 当调用一个非指针参数，变量是复制（call-by-value）的。因此，进行魔法般的反射是在副本上。这样就不能改变原来的值，仅仅改变副本。

### A26. (7) 接口和max()

1. 下面的程序计算了最大值。它是Go 能做到的最通用的形式了。

Listing 6.10. 通用的计算最大值

```
package main

func Less(l, r interface{}) bool { ❶
 switch l.(type) {
 case int:
 if _, ok := r.(int); ok {
 return l.(int) < r.(int) ❷
 }
 case float32:
 if _, ok := r.(float32); ok {
 return l.(float32) < r.(float32) ❷
 }
 }
 return false
}

func main() {
 var a, b, c int = 5, 15, 0
}
```

```
var x, y, z float32 = 5.4, 29.3, 0.0

if c = a; Less(a, b) { ❸
 c = b
}
if z = x; Less(x, y) { ❹
 z = y
}
println(c, z)
}
```

- ❶ 也可以选择让这个函数的返回值为`interface{}`，但是这也就意味着调用者不得不总是使用类型断言来从接口中解析出实际的类型；
- ❷ 所有类型定义为整数。然后进行比较；
- ❸ 参数是`float32`；
- ❹ 获得`a` 和`b` 中的最大值；
- ❺ 浮点类型也一样。



# 7 并发

- “并行是关于性能的；
- 并发是关于程序设计的。”

Google IO 2010  
ROBE PIKE

在这章中将展示Go使用channel和goroutine开发并行程序的能力。goroutine是Go并发能力的核心要素。但是，goroutine到底是什么？来自[6]：

叫做goroutine是因为已有的短语——线程、协程、进程等等——传递了不准确的含义。goroutine有简单的模型：它是与其他goroutine并行执行的，有着相同地址空间的函数。。它是轻量的，仅比分配栈空间多一点点消耗。而初始时栈是很小的，所以它们也是廉价的，并且随着需要在堆空间上分配（和释放）。

goroutine是一个普通的函数，只是需要使用保留字go作为开头。

```
ready("Tee", 2) ← 普通函数调用
go ready("Tee", 2) ← ready() 作为goroutine 运行
```

下面程序的思路来自[32]。让一个函数作为两个goroutine执行，goroutine等待一段时间，然后打印一些内容到屏幕。在第14和15行，启动了goroutine。main函数等待足够的长的时间，这样每个goroutine会打印各自的文本到屏幕。现在是在第17行等待5秒钟（time.Sleep()按ns计算），但实际上没有任何办法知道，当所有goroutine都已经退出应当等待多久。

Listing 7.1. Go routine 实践

```
func ready(w string, sec int64) { 8
 time.Sleep(sec * 1e9) 9
 fmt.Println(w, "is ready!") 10
} 11

func main() { 13
 go ready("Tee", 2) 14
 go ready("Coffee", 1) 15
 fmt.Println("I'm waiting") 16
 time.Sleep(5 * 1e9) 17
} 18
```

表7.1 输出：

```
I'm waiting ← 立刻
Coffee is ready! ← 1 秒后
Tee is ready! ← 2 秒后
```

如果不等待goroutine 的执行（例如，移除第17 行），程序立刻终止，而任何正在执行的goroutine 都会停止。为了修复这个，需要一些能够同goroutine 通讯的机制。这一机制通过channels 的形式使用。channel 可以与Unix shell 中的双向管道做类比：可以通过它发送或者接收值。这些值只能是特定的类型：channel 类型。定义一个channel 时，也需要定义发送到channel 的值的类型。注意，必须使用**make** 创建channel：

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

创建channel ci 用于发送和接收整数，创建channel cs 用于字符串，以及channel cf 使用了空接口来满足各种类型。向channel 发送或接收数据，是通过类似的操作符完成的：<-。具体作用则依赖于操作符的位置：

```
ci <- 1 ← 发送整数1 到channel ci
<-ci ← 从channel ci 接收整数
i := <-ci ← 从channel ci 接收整数，并保存到i 中
```

将这些放到实例中去。

Listing 7.2. Go routines 和channel

```
var c chan int ❶

func ready(w string, sec int) {
 time.Sleep(int64(sec) * 1e9)
 fmt.Println(w, "is ready!")
 c <- 1 ❷
}

func main() {
 c = make(chan int) ❸
 go ready("Tee", 2) ❹
 go ready("Coffee", 1)
 fmt.Println("I'm waiting, but not too long")
 <-c ❺
 <-c ❻
}
```

❶ 定义c 作为int 型的channel。就是说：这个channel 传输整数。注意这个变量是全局的，这样goroutine 可以访问它；

❷ 发送整数1 到channel c；

❸ 初始化c；

❹ 用保留字go 开始一个goroutine；

❺ 等待，直到从channel 上接收一个值。注意，收到的值被丢弃了；

❻ 两个goroutines，接收两个值。

这里仍然有一些丑陋的东西；不得不从channel中读取两次（第14和15行）。在这个例子中没问题，但是如果不知道有启动了多少个goroutine怎么办呢？这里有另一个Go内建的保留字：**select**。通过**select**（和其他东西）可以监听channel上输入的数据。

在这个程序中使用**select**，并不会让它变得更短，因为运行的goroutine太少了。移除第14和15行，并用下面的内容替换它们：

Listing 7.3. 使用select

```
L: for { 14
 select { 15
 case <-c: 16
 i++ 17
 if i > 1 { 18
 break L 19
 } 20
 } 21
} 22
```

使其并行运行

虽然goroutine是并发执行的，但是它们并不是并行运行的。如果不告诉Go额外的东西，同一时刻只会会有一个goroutine执行。利用runtime.GOMAXPROCS(n)可以设置goroutine并行执行的数量。来自文档：

*GOMAXPROCS* 设置了同时运行的CPU的最大数量，并返回之前的设置。如果  $n < 1$ ，不会改变当前设置。当调度得到改进后，这将被移除。

如果不希望修改任何源代码，同样可以通过设置环境变量GOMAXPROCS为目标值。

## 更多关于channel

当在Go中用 `ch := make(chan bool)` 创建channel时，bool型的无缓冲channel会被创建。这对于程序来说意味着什么呢？首先，如果读取（`value := <-ch`）它将会被阻塞，直到有数据接收。其次，任何发送（`ch<-5`）将会被阻塞，直到数据被读出。无缓冲channel是在多个goroutine之间同步很棒的工具。

不过Go也允许指定channel的缓冲大小，很简单，就是channel可以存储多少元素。`ch := make(chan bool, 4)`，创建了可以存储4个元素的bool型channel。在这个channel中，前4个元素可以无阻塞的写入。当写入第5<sup>1</sup>元素时，代码将会阻塞，直到其他goroutine从channel中读取一些元素，腾出空间。

虽然从channel读取是阻塞的，但是仍然可以用下面的方式实现非阻塞的读取：

```
x, ok = <-ch
```

当读取到了内容时ok为true，否则为false。同时，x从channel获取到值。一句话来说，在Go中下面的为true：

$$\text{ch} := \text{make}(\text{chan type}, \text{value}) \begin{cases} \text{value} == 0 & \rightarrow \text{无缓冲 (阻塞)} \\ \text{value} > 0 & \rightarrow \text{缓冲 (非阻塞, 直到value个元素)} \end{cases}$$

TODO  
需要对这个做一些测试。

关闭channel

在goroutines 中包装函数

参阅godns/resolver.go 如何在goroutine 中用错误处理等包装Query() 函数。

## 练习

### Q27. (4) Channel

1. 修改在练习Q2 中创建的程序，换句话说，主体中调用的函数现在是一个goroutine 并且使用channel 通讯。不用担心goroutine 是如何停止的。
2. 在完成了问题1 后，仍有一些待解决的问题。其中一个麻烦是goroutine 在main.main() 结束的时候，没有进行清理。更糟的是，由于main.main() 和main.shower() 的竞争关系，不是所有数字都被打印了。本应该打印到9，但是有时只打印到8。添加第二个退出channel，可以解决这两个问题。试试吧。<sup>a</sup>

### Q28. (7) 斐波那契II

1. 这是类似的练习，第一个在第38 页的练习11。完整的问题描述：

斐波那契数列以：1, 1, 2, 3, 5, 8, 13, ... 开头。或用数学形式： $x_1 = 1; x_2 = 1; x_n = x_{n-1} + x_{n-2} \quad \forall n > 2$ 。

编写一个函数接收`int` 值，并给出同样数量的斐波那契数列。

但是现在有额外条件：必须使用channel。

---

<sup>a</sup>需要用到select 语句。





## 答案

### A27. (4) Channel

1. 程序可能的形式是：

Listing 7.4. Go 的channel

```

package main 1

import "fmt" 3

func main() { 5
 ch := make(chan int) 6
 go shower(ch) 7
 for i := 0; i < 10; i++ { 8
 ch <- i 9
 } 10
} 11

func shower(c chan int) { 13
 for { 14
 j := <-c 15
 fmt.Printf("%d\n", j) 16
 } 17
} 18

```

以通常的方式开始，在第6行创建了一个新的int类型的channel。下一行调用了shower函数，用ch变量作为参数，这样就可以与其通讯。然后进入for循环（第8-10行），在循环中发送（通过<-）数字到函数（现在是goroutine）shower。在函数shower中等待（阻塞方式），直到接收到了数字（第15行）。每个收到的数字都被打印（第16行）出来，然后继续第14行开始的死循环。

2. 答案是

Listing 7.5. 添加额外的退出channel

```

package main 1

import "fmt" 3

func main() { 5
 ch := make(chan int) 6
 quit := make(chan bool) 7
 go shower(ch, quit) 8
 for i := 0; i < 10; i++ { 9
 ch <- i 10
 } 11
 quit <- false // 或者是true，这没啥关系 12
} 13

```

```

func shower(c chan int, quit chan bool) { 15
 for { 16
 select { 17
 case j := <-c: 18
 fmt.Printf("%d\n", j) 19
 case <-quit: 20
 break 21
 } 22
 } 23
} 24

```

在第20行从退出channel读取并丢弃该值。可以使用`q := <-quit`，但是可能只需要用这个变量一次——在Go中这是非法的。另一种办法，你可能已经想到了：`_ = <-quit`。在Go中这是合法的，但是第20行的形式在Go中更好。

#### A28. (7) 斐波那契II

1. 下面的程序使用channel计算了斐波那契数列。

Listing 7.6. Go 的斐波那契函数

```

package main
import "fmt"

func dup3(in <-chan int) (<-chan int, <-chan int, <-chan int) {
 a, b, c := make(chan int, 2), make(chan int, 2), make(chan int, 2)
 go func() {
 for {
 x := <-in
 a <- x
 b <- x
 c <- x
 }
 }()
 return a, b, c
}

func fib() <-chan int {
 x := make(chan int, 2)
 a, b, out := dup3(x)
 go func() {
 x <- 0
 x <- 1
 <-a
 for {
 x <- <-a+<-b
 }
 }()
}

```

```
 return out
 }

 func main() {
 x := fib()
 for i := 0; i < 10; i++ {
 fmt.Println(<-x)
 }
 }

 // See sdh33b.blogspot.com/2009/12/fibonacci-in-go.html
```

# 8

## 通讯

“好的沟通就像是一杯刺激的浓咖啡，然后就难以入睡。”

ANNE MORROW LINDBERGH

在这章中将介绍Go 中与外部通讯的通讯模块。

### 文件和目录

在Go 中，从文件读取（或写入）是很容易的。程序只需要使用`os` 包就可以从文件`/etc/passwd` 中读取数据。

*Listing 8.1. 从文件读取（无缓冲）*

```
package main

import "os"

func main() {
 buf := make([]byte, 1024)
 f, _ := os.Open("/etc/passwd", os.O_RDONLY, 0666)
 defer f.Close()

 f, _ := os.Open("/etc/passwd") ❶
 defer f.Close() ❶

 for {
 n, _ := f.Read(buf) ❷
 if n == 0 { break } ❸
 os.Stdout.Write(buf[:n]) ❹
 }
}
```

- ❶ 打开文件；
- ❶ 确保关闭（close）它；
- ❷ 一次读取1024 字节；
- ❸ 到达文件末尾；
- ❹ 将内容写入Stdout；

如果想要使用缓冲IO，则有`bufio` 包：

Listing 8.2. 从文件读取 (缓冲)

```

package main

import ("os"; "bufio")

func main() {
 buf := make([]byte, 1024)
 f, _ := os.Open("/etc/passwd") ❶
 defer f.Close()
 r := bufio.NewReader(f) ❶

 w := bufio.NewWriter(os.Stdout)
 defer w.Flush()
 for {
 n, _ := r.Read(buf) ❷
 if n == 0 { break }
 w.Write(buf[0:n])
 }
}

```

- ❶ 打开文件，`os.Open` 返回一个实现了 `io.Reader` 接口的 `*os.File`；
- ❶ 转换 `f` 为有缓冲的 `Reader`。`NewReader` 需要一个 `io.Reader`，因此或许你认为这会出错。但其实不会。任何有 `Read()` 函数就实现了这个接口。同时，从列表8.1可以看到，`*os.File` 已经这样做了；
- ❷ 从 `Reader` 读取，而向 `Writer` 写入，然后向屏幕输出文件。

更加通用的方法（但是也略微复杂一点）是 `ReadLine`，参阅 `bufio` 包的文档。

Listing 8.3. 在 *shell* 中创建目录

```

if [! -e name]; then
 mkdir name
else
 # error
fi

```

在 *shell* 脚本中更常见的情景是检查某个目录是否存在，不存在就创建一个。

Listing 8.4. 在 *Go* 中创建目录

```

if f, e := os.Stat("name"); e != nil {
 os.Mkdir("name", 0755)
} else {
 // error
}

```

前面的程序将整个文件读出，但是通常情况下会希望一行一行的读取。下面的片段展示了如何实现：

```
f, _ := os.Open("/etc/passwd")
defer f.Close()
r := bufio.NewReader(f)
s, ok := r.ReadString('\n') { ← 从输入中读取一行
// ... | ← s 保存了字符串，通过string 包就可以解析它|
```

## 命令行参数

来自命令行的参数在程序中通过字符串slice `os.Args` 获取，导入包`os` 即可。`flag` 包有着精巧的接口，同样提供了解析标识的方法。这个例子是一个DNS 查询工具：

```
dnssec := flag.Bool("dnssec", false, "Request DNSSEC records") ❶
port := flag.String("port", "53", "Set the query port") ❶
flag.Usage = func() { ❷
 fmt.Fprintf(os.Stderr, "Usage: %s [OPTIONS] [name ...]\n", os.Args[0])
 flag.PrintDefaults() ❸
}
flag.Parse() ❹
```

- ❶ 定义bool 标识，`-dnssec`。变量必须是指针，否则package 无法设置其值；
- ❶ 类似的，`port` 选项；
- ❷ 简单的重定义Usage 函数，有点罗嗦；
- ❸ 指定的每个标识，`PrintDefaults` 将输出帮助信息；
- ❹ 解析标识，并填充变量。

## 执行命令

`exec` 包有函数可以执行外部命令，这也是在Go 中主要的执行命令的方法。通过定义一个有着数个方法的`*exec.Cmd` 结构来使用。

执行`ls -l`：

```
import "exec"

cmd := exec.Command("/bin/ls", "-l")
err := cmd.Run()
```

从命令行的标准输出中获得信息同样简单：

```
import "exec"

cmd := exec.Command("/bin/ls", "-l")
buf, err := cmd.Output() ← buf 是一个[]byte
```

## 网络

所有网络相关的类型和函数可以在`net`包中找到。这其中最重要的函数是`Dial`。当`Dial`到远程系统，这个函数返回`Conn`接口类型，可以用于发送或接收信息。函数`Dial`简洁的抽象了网络层和传输层。因此IPv4或者IPv6，TCP或者UDP可以共用一个接口。

通过TCP连接到远程系统（端口80），然后是UDP，最后是TCP通过IPv6，大致是这样：<sup>a</sup>

```
conn, e := Dial("tcp", "192.0.32.10:80")
conn, e := Dial("udp", "192.0.32.10:80")
conn, e := Dial("tcp", "[2620:0:2d0:200::10]:80") ← 方括号是强制的
```

如果没有错误（由`e`返回），就可以使用`conn`从套接字中读写。在包`net`中的原始定义是：

```
// Read reads data from the connection.
// Read can be made to time out and return a net.Error with Timeout() == true
// after a fixed time limit; see SetTimeout and SetReadTimeout.
Read(b []byte)(n int, err os.Error)

// Write writes data to the connection.
// Write can be made to time out and return a net.Error with Timeout() == true
// after a fixed time limit; see SetTimeout and SetWriteTimeout.
Write(b []byte)(n int, err os.Error)
```

但是这些都是隐含的低层<sup>b</sup>，通常总是应该使用更高层次的包。例如`http`包。一个简单的`http Get`作为例子：

```
package main
import ("io/ioutil"; "http"; "fmt") ❶

func main() {
 r, err := http.Get("http://www.google.com/robots.txt") ❶
 if err != nil { fmt.Printf("%s\n", err.String()); return } ❷
 b, err := ioutil.ReadAll(r.Body) ❸
 r.Body.Close()
 if err == nil { fmt.Printf("%s", string(b)) } ❹
}
```

❶ 需要的导入；

❶ 使用`http`的`Get`获取`html`；

❷ 错误处理；

❸ 将整个内容读入`b`；

❹ 如果一切OK的话，打印内容。

<sup>a</sup>在这个例子中，可以认为192.0.32.10和2620:0:2d0:200::10是`www.example.org`。

<sup>b</sup>练习Q33是关于使用这些的。



## 练习

### Q29. (8) 进程

1. 编写一个程序，列出所有正在运行的进程，并打印每个进程执行的子进程个数。输出应当类似：

```
Pid 0 has 2 children: [1 2]
Pid 490 has 2 children: [1199 26524]
Pid 1824 has 1 child: [7293]
```

- 为了获取进程列表，需要得到`ps -e -opid,ppid,comm`的输出。输出类似：

```
PID PPID COMMAND
9024 9023 zsh
19560 9024 ps
```

- 如果父进程有一个子进程，就打印`child`，如果多于一个，就打印`children`；
- 进程列表要按照数字排序，这样就以`pid 0`开始，依次展示。

这里有一个Perl 版本的程序来帮助上手（或者造成绝对的混乱）。

*Listing 8.5. Processes in Perl*

```
#!/usr/bin/perl -l
my (%child, $pid, $parent);
my @ps=`ps -e -opid,ppid,comm`; # Capture the output from `ps`
foreach (@ps[1..$#ps]) { # Discard the header line
 ($pid, $parent, undef) = split; # Split the line, discard 'comm'
 push @{$child{$parent}}, $pid; # Save the child PIDs on a list
}
Walk through the sorted PPIDs
foreach (sort { $a <=> $b } keys %child) {
 print "Pid ", $_, " has ", @{$child{$_}}+0, " child", # Print
 them
 @{$child{$_}} == 1 ? " : " : "ren: ", "[@{$child{$_}}]";
}
}
```

### Q30. (5) 单词和字母统计

1. 编写一个从标准输入中读取文本的小程序，并进行下面的操作：

1. 计算字符数量（包括空格）；
2. 计算单词数量；
3. 计算行数。

换句话说，实现一个`wc(1)`（参阅本地的手册页面），然而只需要从标准输入读取。

### Q31. (4) Uniq

1. 编写一个Go 程序模仿Unix 命令`uniq` 的功能。程序应当像下面这样运行，提供一个下面这样的列表：

```
'a' 'b' 'a' 'a' 'a' 'c' 'd' 'e' 'f' 'g'
```

它将打印出没有后续重复的项目：

```
'a' 'b' 'a' 'c' 'd' 'e' 'f'
```

下面列出的8.8 是Perl 实现的算法。

Listing 8.8. *uniq(1)* 的Perl 实现

```
#!/usr/bin/perl
my @a = qw/a b a a a c d e f g/;
print my $first = shift @a;
foreach (@a) {
 if ($first ne $_) { print; $first = $_; }
}
```

**Q32. (9) Quine** A *Quine* 是一个打印自己的程序。

1. 用Go 编写一个Quine 程序。

**Q33. (8) Echo 服务**

1. 编写一个简单的echo 服务。使其监听于本地的TCP 端口8053 上。它应当可以读取一行（以换行符结尾），将这行原样返回然后关闭连接。
2. 让这个服务可以并发，这样每个请求都可以在独立的goroutine 中进行处理。

**Q34. (9) 数字游戏**

- 从列表中随机选择六个数字：

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100

数字可以多次被选中；

- 从1...1000 中选择一个随机数*i*；
- 尝试用先前的六个数字（或者其中的几个）配合运算符+，-，\*和/，计算出*i*；

例如，选择了数字：1，6，7，8，8 和75。并且*i* 为977。可以用许多方法来实现，其中一种：

$$((((1 * 6) * 8) + 75) * 8) - 7 = 977$$

或者

$$(8 * (75 + (8 * 6))) - (7/1) = 977$$

1. 实现像这样的数字游戏。使其打印像上面那样格式的结果（也就是说，输出应当是带有括号的中序表达式）
2. 计算全部可能解，并且全部显示出来（或者仅显示有多少个）。在上面的例子中，有544 种方法。

**Q35. (8) \*Finger 守护进程**

1. 编写一个finger 守护进程，可以工作于finger(1) 命令。

来自Debian 的包描述：

*Fingerd* 是一个基于RFC 1196 [41] 的简单的守护进程，它为许多站点提供了“finger”程序的接口。这个程序支持返回一个友好的、面向用户的系统或用户当前状况的详细报告。



## 答案

### A29. (8) 进程

1. 有许多工作需要做。可以将程序分为以下几个部分：

1. 运行ps 获得输出；
2. 解析输出并保存每个PPID 的子PID；
3. 排序PPID 列表；
4. 打印排序后的列表到屏幕。

在下面的解法中，选择container/vector 保存PID。”列表”自动增长。

程序清单：

Listing 8.6. Go 中的进程

```
package main

import ("fmt"; "sort"; "exec"; "strings"; "strconv"; "container/
vector")

func main() {
 ps := exec.Command("ps", "-e", "-opid,ppid,comm")
 output, _ := ps.Output()
 child := make(map[int]*vector.IntVector)
 for i, s := range strings.Split(string(output), "\n") {
 if i == 0 || len(s) == 0 { // Kill first and last line
 continue
 }
 f := strings.Fields(s)
 fpp, _ := strconv.Atoi(f[1]) // the parent's pid
 fp, _ := strconv.Atoi(f[0]) // the child's pid
 if _, present := child[fpp]; !present {
 v := new(vector.IntVector)
 child[fpp] = v
 }
 // Save the child PIDs on a vector
 child[fpp].Push(fp) // Save the child PIDs on a vector
 }
 schild := make([]int, len(child))
 i := 0
 for k, _ := range child {
 schild[i] = k
 i++
 }
 sort.Ints(schild)
 for _, ppid := range schild { // Walk through the sorted list
 fmt.Printf("Pid %d has %d child", ppid, child[ppid].
 Len())
 }
}
```

```

 if child[ppid].Len() == 1 {
 fmt.Printf(": %v\n", []int{*child[ppid]})
 } else {
 fmt.Printf("ren: %v\n", []int{*child[ppid]})
 }
 }
}

```

### A30. (5) 单词和字母统计

1. 下面是wc(1) 的一种实现。

Listing 8.7. wc(1) 的Go 实现

```

package main

import (
 "os"
 "fmt"
 "bufio"
 "strings"
)

func main() {
 var chars, words, lines int
 r := bufio.NewReader(os.Stdin) ❶
 for {
 switch s, ok := r.ReadString('\n'); true { ❶
 case ok != nil: ❷
 fmt.Printf("%d %d %d\n", chars, words, lines);
 return
 default: ❸
 chars += len(s)
 words += len(strings.Fields(s))
 lines++
 }
 }
}

```

- ❶ Start a new reader that reads from standard input;
- ❶ Read a line from the input;
- ❷ If we received an error, we assume it was because of a EOF. So we print the current values;
- ❸ Otherwise we count the charaters, words and increment the lines.

### A31. (4) Uniq

1. 下面是uniq 的Go 实现。

Listing 8.9. *uniq(1)* 的Go 实现

```

package main

import "fmt"

func main() {
 list := []string{"a", "b", "a", "a", "c", "d", "e", "f"}
 first := list[0]

 fmt.Printf("%s ", first)
 for _, v := range list[1:] {
 if first != v {
 fmt.Printf("%s ", v)
 first = v
 }
 }
}

```

## A32. (9) Quine

1. 下面的Quine 来自Russ Cox :

```

/* Go quine */
package main
import "fmt"
func main() {
 fmt.Printf("%s%c%s%c\n", q, 0x60, q, 0x60)
}
var q = `/* Go quine */
package main
import "fmt"
func main() {
 fmt.Printf("%s%c%s%c\n", q, 0x60, q, 0x60)
}
var q = `

```

## A33. (8) Echo 服务

1. 一个简单的echo 服务器是这样 :

Listing 8.10. 简易echo 服务器

```

package main
import ("net"; "fmt"; "bufio")

func main() {
 l, err := net.Listen("tcp", "127.0.0.1:8053")

```

```

 if err != nil {
 fmt.Printf("Failure to listen: %s\n", err.String())
 }
 for {
 if c, err := l.Accept(); err == nil { Echo(c) }
 }
 }

 func Echo(c net.Conn) {
 defer c.Close()
 line, err := bufio.NewReader(c).ReadString('\n')
 if err != nil {
 fmt.Printf("Failure to read: %s\n", err.String())
 return
 }
 _, err = c.Write([]byte(line))
 if err != nil {
 fmt.Printf("Failure to write: %s\n", err.String())
 return
 }
 }
}

```

当运行起来的时候可以看到如下内容：

```

% nc 127.0.0.1 8053
Go is *awesome*
Go is *awesome*

```

2. 为了使其能够并发处理链接，只需要修改一行代码，就是：

```
if c, err := l.Accept(); err == nil { Echo(c) }
```

改为：

```
if c, err := l.Accept(); err == nil { go Echo(c) }
```

#### A34. (9) 数字游戏

1. 下面的是一种可能的解法。它使用了递归和回溯来得到答案。由于程序的尺寸问题，用了较小的字体来展示。

Listing 8.11. 数字游戏

```

package main

// 为了增加而需要重写
import (
 "fmt"
 "strconv"
 "flag"
)

const (

```

```

 _ = 1000 * iota
 ADD
 SUB
 MUL
 DIV
 MAXPOS = 11
)

 var mop = map[int]string{ADD: "+", SUB: "-", MUL: "*", DIV: "/"}

 var (
 ok bool
 value int
)

 type Stack struct {
 i int
 data [MAXPOS]int
 }

 func (s *Stack) Reset() { s.i = 0 }

 func (s *Stack) Len() int { return s.i }

 func (s *Stack) Push(k int) { s.data[s.i] = k; s.i++ }

 func (s *Stack) Pop() int { s.i--; return s.data[s.i] }

 var found int
 var stack = new(Stack)

 func main() {
 flag.Parse()
 list := []int{1, 6, 7, 8, 8, 75, ADD, SUB, MUL, DIV}
 // Arg0 包含了需要处理的数字
 magic, ok := strconv.Atoi(flag.Arg(0))
 if ok != nil {
 return
 }
 f := make([]int, MAXPOS)
 solve(f, list, 0, magic)
 }

 func solve(form, numberop []int, index, magic int) {
 var tmp int
 for i, v := range numberop {
 if v == 0 {
 goto NEXT
 }
 if v < ADD {

```



```

 // 是一个数字，保存起来
 tmp = numberop[i]
 numberop[i] = 0
 }
 form[index] = v
 value, ok = rpncalc(form[0 : index+1])

 if ok && value == magic {
 if v < ADD {
 numberop[i] = tmp // 重置并继续
 }
 found++
 fmt.Printf("%s = %d #%d\n", rpnstr(form[0:
 index+1]), value, found)
 //goto NEXT
 }

 if index == MAXPOS-1 {
 if v < ADD {
 numberop[i] = tmp // 重置并继续
 }
 goto NEXT
 }
 solve(form, numberop, index+1, magic)
 if v < ADD {
 numberop[i] = tmp // 重置并继续
 }
NEXT:
}

// convert rpn to nice infix notation and string
// the r must be valid rpn form
func rpnstr(r []int) (ret string) {
 s := make([]string, 0) // Still memory intensive
 for k, t := range r {
 switch t {
 case ADD, SUB, MUL, DIV:
 a, s := s[len(s)-1], s[:len(s)-1]
 b, s := s[len(s)-1], s[:len(s)-1]
 if k == len(r)-1 {
 s = append(s, b+mop[t]+a)
 } else {
 s = append(s, "("+b+mop[t]+a+")")
 }
 default:
 s = append(s, strconv.Itoa(t))
 }
 }
 for _, v := range s {

```

```

 ret += v
 }
 return
}

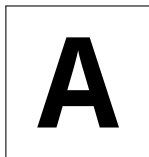
// return result from the rpn form.
// if the expression is not valid, ok is false
func rpnCalc(r []int) (int, bool) {
 stack.Reset()
 for _, t := range r {
 switch t {
 case ADD, SUB, MUL, DIV:
 if stack.Len() < 2 {
 return 0, false
 }
 a := stack.Pop()
 b := stack.Pop()
 if t == ADD {
 stack.Push(b + a)
 }
 if t == SUB {
 // disallow negative subresults
 if b-a < 0 {
 return 0, false
 }
 stack.Push(b - a)
 }
 if t == MUL {
 stack.Push(b * a)
 }
 if t == DIV {
 if a == 0 {
 return 0, false
 }
 // disallow fractions
 if b%a != 0 {
 return 0, false
 }
 stack.Push(b / a)
 }
 default:
 stack.Push(t)
 }
 }
 if stack.Len() == 1 { // there is only one!
 return stack.Pop(), true
 }
 return 0, false
}

```

2. 开始运行permrec时，输入977作为第一个参数：

```
% ./permrec 977
1+(((6+7)*75)+(8/8)) = 977 #1
...
((75+(8*6))*8)-7 = 977 #542
(((75+(8*6))*8)-7)*1 = 977 #543
(((75+(8*6))*8)-7)/1 = 977 #544
```





# 版权

本作品由 $\text{\LaTeX}$ 创作。主文本设置为Google Droid 字体。所有打印文本设置为DejaVu Mono。  
中文字体设置为WenQuanYi Zen Hei。

## 贡献者

下面的成员编写了这本书。

- Miek Gieben     <miek@miek.nl> ;
- JC van Winkel。

下面的成员翻译了这本书。

- 邢兴     <mikespook@gmail.com>。

帮助试读、检查练习和改进文案（不论真实姓名还是别名都不分排名先后）*Filip Zuludek* , *Jonathan Kans* , *Jaap Akkerhuis* , *Mayuresh Kathe* , *Makoto Inoue* , *Ben Bullock* , *Bob Cunningham* , *Dan Kortschak* , *Sonia Keys* , *Babu Sreekanth* , *Haiping Fan* , *Cecil New* , *Andrey Mirtchovski* , *Thomas Kapplet* , *Uriel* , *Brian Fallik* , *Paulo Pinto* , *Jinpu Hu* , *Russel Winder* , *Michael Stapelberg*。

Miek Gieben



Miek Gieben 从荷兰内梅亨大学取得计算机科学硕士学位。他参与并开发了DNSSEC 协议——下一代DNS——DNSSEC protocol [1, 3, 2]以及如核心认证[26]。在玩过了Erlang 后，Go 是他当前迷恋的主要语言。他将所有业余时间用来对Go 的探索和编码。他是Go DNS 库的维护者：<https://github.com/miekg/godns>。他的个人博客是<http://www.miek.nl> 以及Twitter 帐号@miekg。博文和推多数情况下都是关于Go 的。

邢兴



在你看到本书的中文版的时候，邢兴已经从中山大学获得了软件工程硕士学位。他的博客是<http://mikespook.com> 并且拥有Twitter 帐号@mikespook。博文和推大部分情况下都是用来扯蛋的，而且会时不时的扯疼……

## 许可证和版权

本作品依照署名-非商业性使用-相同方式共享3.0 Unported许可证发布。访问<http://creativecommons.org/licenses/by-nc-sa/3.0/> 查看该许可证副本，或写信到Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA。

本书所有实例代码依此方式放入公共领域。

©Miek Gieben – 2010; , 2011。

©邢兴– 2011。

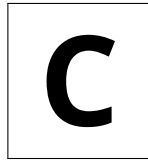
# B

## 索引

- array
  - capacity, 19
  - length, 19
  - multidimensional, 19
- buffered, 92
- built-in
  - append, 18, 21
  - cap, 18
  - close, 18
  - complex, 18
  - copy, 18, 21
  - delete, 18
  - imag, 18
  - len, 18
  - make, 18, 60
  - new, 18, 60
  - panic, 18
  - print, 18
  - println, 18
  - real, 18
  - recover, 18
- channel, 6, 85
  - blocking read, 86
  - blocking write, 86
  - non-blocking read, 86
  - non-blocking write, 86
  - unbuffered, 86
- channels, 85
- closure, 35
- complex numbers, 18
- deferred list, 35
- duck typing, 72
- field
  - anonymous, 64
- fields, 62
- function
  - as values, 36
  - call, 64
  - literal, 35
  - literals, 36
- generic, 76
- goroutine, 84
- goroutines, 6
- gotest, 51
- interface, 72
  - set of methods, 72
  - type, 72
  - value, 72
- keyword
  - break, 13, 15
  - continue, 15
  - default, 17
  - defer, 34
  - else, 13
  - fallthrough, 17
  - for, 14
  - go, 84
  - goto, 14
  - if, 13
  - import, 50
  - iota, 10
  - map, 21
    - add elements, 22
    - existence, 22
    - remove elements, 22
  - package, 48
  - range, 16, 22
    - on maps, 16, 22
    - on slices, 16
  - return, 13
  - select, 86
  - struct, 63
  - switch, 16
  - type, 63
- label, 14
- literal
  - composite, 19, 62
- method, 30
- method call, 64
- methods
  - inherited, 65
- MixedCaps, 51
- named return parameters, 30
- networking

- Dial, 95
- nil, 60
- operator
  - address-of, 60
  - and, 12
  - bit wise xor, 12
  - bitwise
    - and, 12
    - clear, 12
    - or, 12
  - channel, 85
  - increment, 60
  - not, 12
  - or, 12
- package
  - bufio, 51, 53, 92
  - bytes, 50
  - compress/gzip, 51
  - even, 48
  - exec, 54, 94
  - flag, 54
  - fmt, 18, 53
  - http, 54
  - io, 53
  - json, 54
  - os, 54
  - reflect, 54, 78
  - ring, 51
  - sort, 54
  - strconv, 54
  - sync, 54
  - template, 54
  - unsafe, 54
- parallel assignment, 9, 15
- pass-by-value, 30
- private, 49
- public, 49
- receiver, 30
- reference types, 19
- runes, 16
- scope
  - local, 31
- slice
  - capacity, 19
  - length, 19
- string literal
  - interpreted, 11
  - raw, 11
- type assertion, 74
- type switch, 73
- variables
  - $\_$ , 9
  - assigning, 8
  - declaring, 8
  - underscore, 9





## Bibliography

- [1] et al. Arends. Dns security introduction and requirements. <http://www.ietf.org/rfc/rfc4033.txt>, 2005.
- [2] et al. Arends. Protocol modifications for the dns security extensions. <http://www.ietf.org/rfc/rfc4035.txt>, 2005.
- [3] et al. Arends. Resource records for the dns security extensions. <http://www.ietf.org/rfc/rfc4034.txt>, 2005.
- [4] LAMP Group at EPFL. Scala. <http://www.scala-lang.org/>, 2003.
- [5] Go Authors. Defer, panic, and recover. <http://blog.golang.org/2010/08/defer-panic-and-recover.html>, 2010.
- [6] Go Authors. Effective go. [http://golang.org/doc/effective\\_go.html](http://golang.org/doc/effective_go.html), 2010.
- [7] Go Authors. Go faq. [http://golang.org/doc/go\\_faq.html](http://golang.org/doc/go_faq.html), 2010.
- [8] Go Authors. Go language specification. [http://golang.org/doc/go\\_spec.html](http://golang.org/doc/go_spec.html), 2010.
- [9] Go Authors. Go package documentation. <http://golang.org/doc/pkg/>, 2010.
- [10] Go Authors. Go release history. <http://golang.org/doc/devel/release.html>, 2010.
- [11] Go Authors. Go tutorial. [http://golang.org/doc/go\\_tutorial.html](http://golang.org/doc/go_tutorial.html), 2010.
- [12] Go Authors. Go website. <http://golang.org/>, 2010.
- [13] Haskell Authors. Haskell. <http://www.haskell.org/>, 1990.
- [14] Inferno Authors. Inferno. <http://www.vitanuova.com/inferno/>, 1995.
- [15] Perl Package Authors. Comprehensive perl archive network. <http://cpan.org/>, 2010.
- [16] Plan 9 Authors. Plan 9. <http://plan9.bell-labs.com/plan9/index.html>, 1992.
- [17] Plan 9 Authors. Limbo. <http://www.vitanuova.com/inferno/papers/limbo.html>, 2010.
- [18] Mark C. Chu-Carroll. Google's new language: Go. [http://scienceblogs.com/goodmath/2009/11/googles\\_new\\_language\\_go.php](http://scienceblogs.com/goodmath/2009/11/googles_new_language_go.php), 2010.
- [19] Go Community. Go issue 65: Compiler can't spot guaranteed return in if statement. <http://code.google.com/p/go/issues/detail?id=65>, 2010.
- [20] Go Community. Go nuts mailing list. <http://groups.google.com/group/golang-nuts>, 2010.
- [21] Ericsson Cooperation. Erlang. <http://www.erlang.se/>, 1986.
- [22] D. Crockford. The application/json media type for javascript object notation (json). <http://www.ietf.org/rfc/rfc4627.txt>, 2006.

- [23] Brian Kernighan Dennis Ritchie. The c programming language, 1975.
- [24] James Gosling et al. Java. <http://oracle.com/java/>, 1995.
- [25] Larray Wall et al. Perl. <http://perl.org/>, 1987.
- [26] Kolkman & Gieben. Dnssec operational practices. <http://www.ietf.org/rfc/rfc4641.txt>, 2006.
- [27] C. A. R. Hoare. Quicksort. <http://en.wikipedia.org/wiki/Quicksort>, 1960.
- [28] C. A. R. Hoare. Communicating sequential processes (csp). <http://www.usingcsp.com/cspbook.pdf>, 1985.
- [29] Go Community (SnakE in particular). Function accepting a slice of interface types. [http://groups.google.com/group/golang-nuts/browse\\_thread/thread/225fad3b5c6d0321](http://groups.google.com/group/golang-nuts/browse_thread/thread/225fad3b5c6d0321), 2010.
- [30] Rob Pike. Newsqueak: A language for communicating with mice. <http://swtch.com/~rsc/thread/newsqueak.pdf>, 1989.
- [31] Rob Pike. The go programming language, day 2. <http://golang.org/doc/GoCourseDay2.pdf>, 2010.
- [32] Rob Pike. The go programming language, day 3. <http://golang.org/doc/GoCourseDay3.pdf>, 2010.
- [33] Joseph Poirier. Go mingw. [https://bitbucket.org/jpoirier/go\\_mingw/downloads/](https://bitbucket.org/jpoirier/go_mingw/downloads/), 2011.
- [34] Bjarne Stroustrup. The c++ programming language, 1983.
- [35] Ian Lance Taylor. Go interfaces. <http://www.airs.com/blog/archives/277>, 2010.
- [36] Imran On Tech. Using fizzbuzz to find developers who grok coding. <http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/>, 2010.
- [37] Wikipedia. Bubble sort. [http://en.wikipedia.org/wiki/Bubble\\_sort](http://en.wikipedia.org/wiki/Bubble_sort), 2010.
- [38] Wikipedia. Communicating sequential processes. [http://en.wikipedia.org/wiki/Communicating\\_sequential\\_processes](http://en.wikipedia.org/wiki/Communicating_sequential_processes), 2010.
- [39] Wikipedia. Duck typing. [http://en.wikipedia.org/wiki/Duck\\_typing](http://en.wikipedia.org/wiki/Duck_typing), 2010.
- [40] Wikipedia. Iota. <http://en.wikipedia.org/wiki/Iota>, 2010.
- [41] D. Zimmerman. The finger user information protocol. <http://www.ietf.org/rfc/rfc1196.txt>, 1990.

当前页留空。