第 1 章

快速入门

Well begun is half done.

（万事开头难。）

——谚语

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

1.1 编辑和编译

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

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

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

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

我们看一下 acme.bat：

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

这里有几个重要的路径：

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

如果读者希望安装到其他路径下，则需要对 acme.bat 做相应的修改。

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

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

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

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

这样的 Acme，也可以认为是 Inferno 操作系统的图形用户界面

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

到底如何才能打开文件？从哪里能得到帮助？

欢迎 Windows 世界的同学打开窗口，感受 Unix 世界的清新和煦。

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

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

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

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

表1-1 Acme鼠标键组合表

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

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

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

表 1-2 Acme 键盘命令表

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

习惯 Acme 的关键就是鼠标命令，而使用鼠标命令的方便之处在于，Acme 的所有文字，都是可以用鼠标命令来操作的。也就是说，不管命令

位于何处，是在标题行或者一个窗口里，只要在其上按中键或按住中键扫过，都是执行保存文件的操作。而在其上按右键，都是执行查找的操作。这样，我们把一些最常用的命令放在每个窗口的标题行，只要中指轻轻一点就可以了。这些命令和一些不太常用的命令如表 1-3 所示。

表1-3 Acme鼠标命令表

中 文	英 文	代 表
剪	Cut	剪切选中的内容
拷	Copy	复制选中的内容
贴	Paste	粘贴上次剪切的内容
新	New	在新窗口中打开选中的文件
关	Del	关闭此窗口
悔	Undo	撤销上次修改
不悔	Redo	重复上次撤销
存	Put	保存文件
读	Get	重新读取文件
查	Look	查找选中的词语或文件
令	Edit	执行选中的编辑命令
编	Compile	编译Go文件
	Zerox	在新窗口打开同一文件
	Putall	保存所有窗口内容
	Dump	保存窗口状态，下次自动打开
	Newcol	增添一列
	Delcol	删除一列
	Exit	退出Acme（不保存）

win	命令控制窗口
man	帮助手册
g	查找C和Go文件的内容
mkdir、touch、rm	新建目录，新建文件，删除目录或文件

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

不多讲了。再次重申，编辑器的选择是个人喜好，选择什么都和 Go 语言无关。

1.2 世界，你好！

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

```
package main

import "fmt"

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

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

我们大概了解一下“编译”命令的基本原理。

(1) Acme 会使用窗口标题栏的文件名作为参数，调用/dis/goc。

(2) /dis/goc 是 sh 脚本，它先用 gofmt 格式化此窗户的文件。

(3) 如果没错，再用 go run 编译执行此文件。

(4) 如果有错，整理出错行号输出。

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

现在解释程序自身。所有 Go 程序，都由函数和变量构成。一个函

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

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

1.3 自我复制

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

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

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

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

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

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

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

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

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

Go 的注释有两种。第一种就是用/**/括起的大段文字。另一种是

从//开始到行尾的文字。注释文字被编译器忽略。

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

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

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

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

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

1.4 猜数游戏

下一个程序能猜到你想的数，例如：

```
请想一个 0～100 的整数。
该数小于或者等于 50 吗?  (y/n) n
该数小于或者等于 75 吗?  (y/n) y
该数小于或者等于 63 吗?  (y/n) n
该数小于或者等于 69 吗?  (y/n) y
该数小于或者等于 66 吗?  (y/n) n
该数小于或者等于 68 吗?  (y/n) y
该数小于或者等于 67 吗?  (y/n) n
该数是 68
```

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

```
package main

import "fmt"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
package main
```

```
import (
    "fmt"
    "sort"
)

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

现在不需要理解这个程序。

1.5 图灵机

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

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

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

这 7 条指令是：

+ ——使当前数据单元的值增 1；

- ——使当前数据单元的值减 1；

> ——下一个单元作为当前数据单元；

< ——上一个单元作为当前数据单元；

[——如果当前数据单元的值为 0，下一条指令在对应的]后；

] ——如果当前数据单元的值不为 0，下一条指令在对应的[后；

. ——把当前数据单元的值作为字符输出。

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

```
package main

import "fmt"

var (
    a     [30000]byte
```

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

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

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

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

```
for 初始化;判断;增值 {
```

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

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

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

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

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

组对应下标所表示的单元中的值。

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

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

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

1.6 排版工具

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

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

我们作出如下规定。

(1) 段落之间用空行分隔，每段的前四个字符如果是

以 01 开始，则后续数字对应 HTML 的 h1 到 h6；

以 020 开始，则插入文件内容到 HTML 的 `pre`。

(2) 如果一段以空白字符开头，则作为 HTML 的 `pre`。

(3) 如果都不是，就是简单的段落，对应到 HTML 的 `p`。

简化了问题，程序也就非常简单：

```
package main

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

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

func main() {
    flag.Parse()

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

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

```
    }
    return s
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

这样我们完成了简单的自定义格式到标准 html 文件格式转换的工具

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

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

1.7 游乐场

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

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

```
package main

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

var uniq = make(chan int)

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

func main() {
```

```
    if err := os.Chdir(os.TempDir()); err != nil {
        log.Fatal(err)
    }

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

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

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

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

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

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

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

```
        cmd := exec.Command("go", "run", x)
        o, e := cmd.CombinedOutput()
        if err(w, e) {
            return
        }
        w.Write(o)
    }

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

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

```
<button onclick="compile();">run</button>
<div id="output"></div>
</body>
</html>
`
```

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

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

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

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

最后 http 包的 ListenAndServe 函数，运行一个 Web 服务器。由

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

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

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

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

os.Create 创建文件，如果出错，使用 err 输出错误信息到浏览器，然后立即返回。尽管 Go 语言也有派错（panic）和恢复（recover）

异常处理机制，但不鼓励使用它们来处理可以预知的出错情况，而是希望程序能明确地立即处理并返回。这就是接下来使用了大量类似的出错检查和处理语句的原因。

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

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

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

1.8 位钱

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

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

我们看看如何使用 Go 来完成位钱地址所需的 Base58 编码：

```
package bitcoin

import (
    "math/big"
    "strings"
)

const base58 = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabc
defghijk lmnopqrstuvwxyz"

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

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

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

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

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

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

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

```
            z.SetInt64(int64(v))
            x.Mul(x, y)
            x.Add(x, z)
        }
        xa := x.Bytes()

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

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

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

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

```
}
```

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

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

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

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

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

```
package bitcoin

import (
    "crypto/sha256"
    "hash"
)

var sha, sha2 hash.Hash

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

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

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

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

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

```
package bitcoin
```

```
    import (
        "testing"
    )

    type test struct {
        en, de string
    }

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

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

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

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

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

1.9 小结

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