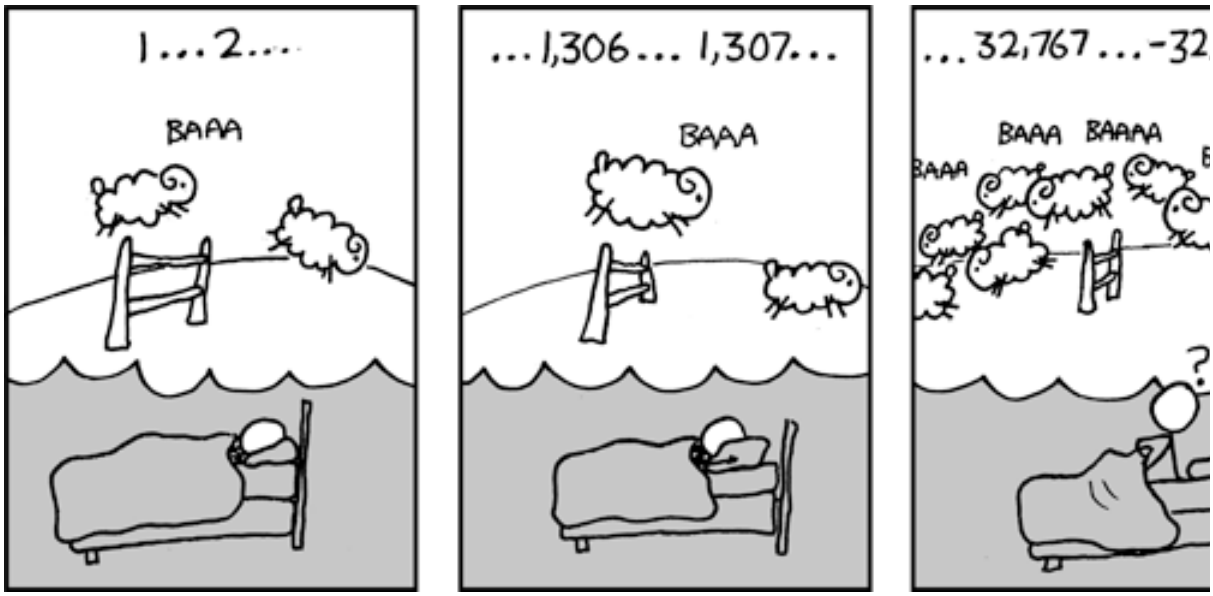
# Go Kernel

**Fango**

# Go Kernel




http://xkcd.com/571/

# 计划

朋友们, 我需要坦白一件事: 写作这活儿, 比编程累. 而且读过"沉默的大多数"的, 还可能把那个作者称为"傻大姐". 就是说作者辛苦学会了一点儿技能, 就觉得有了超级智慧, 非要别人知道, 也不管人家想不想学, 是不是想跟你学, 就死乞白赖的要教人家. 呜呼, 朋友们, 我在此郑重声明, 我绝对没有要教人的计划, 其实到处都有傻瓜书, 教程书, 大全书可以受教. 我的计划不是教书, 是交流. 是有系统的整理出自己学习的感悟, 让自己有所提高, 也让读者有个目标可以切磋.

语言的学习, 众所周知, 最简单的是语法, 然后是词汇, 习语和例句, 最难的是贯通. 编程语言也该如此, 先学会简单语法, 再大量阅读和练习, 增进习语和例程库, 进而能自如地写出像样的作品. 我感觉像样是很不容易的. 除了要合格达到别人的基本要求, 还要过自己审美这一关. 即, 不是交差完事, 还要自己有个满足感, 觉得自己够格了, 语言本身已经不是障碍了.

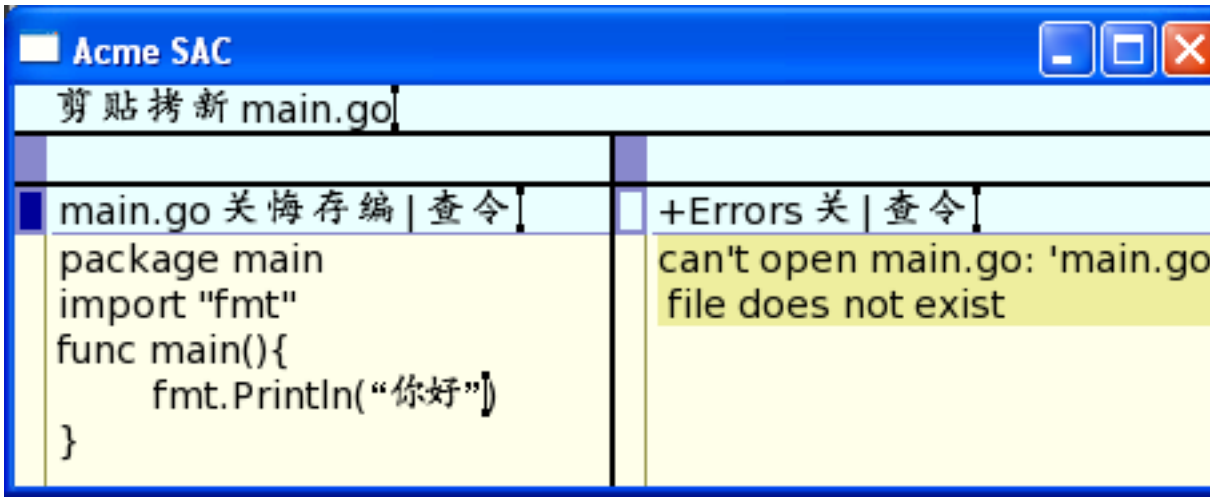
这样, 就有了写三本书的计划, “核”只讲语法和习语. “库”讲最常用的例程库, 而“壳”讲它们的组合. 熟悉 UNIX 的朋友会明白我的用意.

能最快掌握新语言的有两种人: 牙牙学语的新人和熟悉多种语言的通译, 而把它做为第二语言来学的人, 要时刻避免母语的影响, 否则很难自如的转换, 甚至出现邯郸学步的窘况.

所以, 我计划完全不和其它语言比较, 默认朋友们或是博学的才子, 或是好学的新鲜人. 希望前者在笑看我弄斧之余不吝指正, 也欢迎后者在看到罗嗦或不明之处时, 不要大而化之, 尽量告诉我怎样写会比较好些.

另外, 随书奉上完整的编译环境和示例程序, 作为参考基准, 以及新人最快上手的工具. 据个人经验, 第一感觉对理性选择有很大影响. 轻松上路者视困难为挑战, 而负重出行者更容易半途而退.

# Acme



acme

我计划中的基准编译环境, 就是 Acme.

最早的 Acme 是 Go 的主创者 Rob Pike 在90年代写的, 沿用至今. 最知名的用户除了她的作者, 还有 C 的发明人 Dennis, 以及 Go 的另一核心成员, 也是 Plan9port 的作者, Russ. 后来 Acme 被移植到 Inferno OS, 又被整理为直接在 Windows 和 Mac OSX 上运行的 Acme-SAC. 最后, 我把她略微汉化, 加入 Go 的元素, 就有了上图所示的基准编译环境, 即后文中所指的 Acme. 顺便一提, 本书文字, 及其所有程序, 皆完全在此 Acme 完成.

1. 从 `http://code.google.com/p/gomingw/downloads/list` 下载

gowin32, 可以是 installer 或者 zip, 安装或展开到 c:\

2. 从 `http://code.google.com/p/ac-me/downloads/list` 下载 acme.zip, 展开到 Windows 的 c:\
3. 运行 c:\acme.app\acme.bat

如果你的 PC 没有 C 盘, 糟糕, 你需要花些时间调整 acme.bat.

Acme 给我的第一感觉, 就是干净. 当代横行的 WIMP, 即窗口 Window,图标 Icon, 菜单 Menu, 指针 Pointer, 只剩后现代的 WP. 也可代表 Word Processing, 即文字处理 — 文字代表一切. 因为我们是程序员, 不是画图的, 文字是我们赋予的生命. 我们一个个的敲入它们, 删除它们, 拷贝它们, 查找它们, 编译它们, 执行它们. 如果它们可以被形象的快速的使用, 便没有使用 IM 的必要. 而 Pointer, 鼠标, 就可以大大加快文字的处理速度.

最好是三键的鼠标, 而当代鼠标的滚轮, 也可权当中键使用. 笔记本电脑只有两个键的, 只好用 Ctrl 和右键一起当中键的班. 实在不行了, 快捷键组合也能顶顶. 用惯遗产 vi 和 emacs 的朋友可能说我们也纯文本呀也快捷键呀, 我要强烈暗示使用三键鼠标的后现代 Acme 是无以伦比的好用, 快速, 强大! 况且, 还有比三键鼠标更好的 geek 招牌吗?

三个鼠标键的组合是这样的:

鼠标键表

| 左 | 中 | 右 | 代表 |
|---|---|---|---|
| 1+ | 0 | 0 | 移动光标 |
| 1++ | 0 | 0 | 选词, 整行, 或括号引号的段落 |
| 1— | 0 | 0 | 选亮 |
| 0 | 1+ | 0 | 执行命令 |
| 0 | 1— | 0 | 红色选亮, 执行命令 |
| 0 | 0 | 1+ | 打开文件, 或查找词 |
| 0 | 0 | 1— | 绿色选亮打开或查找 |
| 1— | 2+ | 0 | 选亮删除 |
| 1— | 2+ | 3+ | 选亮删除贴回, 即拷贝 |
| 1+ | 0 | 2+ | 删除 |
| 2+ | 1+ | 0 | 之前选亮传给命令执行 |

表中 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. 可以简单的总结为: 左键选亮, 中键执行, 右键查找(文件和词).左中删, 左右贴. 试着练习几分钟就成了小脑指令了 — 也就是不用走大脑了.

Acme 的命令是无穷的, 因为其背后有 Inferno OS 支持, 是一个完整的后现代 Unix 环

境, 所有在 /dis 里的命令 (右击看看), 都可直接中击执行, 而我们也可以用其 sh 甚至 limbo 写自己的命令, 例如 /dis/goc (右击看看).

当然, 最最常用的命令我做了汉化, 放在了提示栏上. 没有汉化的, 可以直接敲入使用其英文命令. 就那么几个:

命令表

| 中文 | 英文 | 代表 |
|---|---|---|
| 剪 | Cut | 剪掉选亮 |
| 拷 | Copy | 拷贝选亮 |
| 贴 | Paste | 贴回上次剪或拷的内容 |
| 新 | New | 新窗口打开后跟选亮的文件 |
| 关 | Del | 关闭此窗口 |
| 悔 | Undo | 反悔上次修改 |
| 不悔 | Redo | 反悔上次反悔 |
| 存 | Put | 保存文件 |
| 读 | Get | 重新读取文件 |
| 查 | Look | 查找选亮词或文件 |
| 令 | Edit | 执行选亮的编辑命令 |
| 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 | 删除此列 |
| | Exit | 退出 Acme (不保存) |

而快捷键有:

快捷键表

| ASCII | Ctrl组合 | 代表 |
|---|---|---|
| 01 | ^a | 行首 |
| 02 | ^b | 上页 |
| 03 | ^c | 拷贝 |

| | | |
|---|---|---|
| 04 | ^d | 补充 Complete |
| 05 | ^e | 行尾 |
| 06 | ^f | 下页 |
| 07 | ^g | |
| 08 | ^h | 退格 Backspace |
| 09 | ^i | Tab |
| 0a | ^j | |
| 0b | ^k | 左移 |
| 0c | ^l | 右移 |
| 0d | ^m | 换行 |
| 0e | ^n | 下移 |
| 0f | ^o | 上移 |
| 10 | ^p | 文件尾 End |
| 11 | ^q | 文件头 Home |
| 12 | ^r | |
| 13 | ^s | 保存 Save |
| 14 | ^t | |
| 15 | ^u | 删整行 |
| 16 | ^v | 贴回 Paste |
| 17 | ^w | 删词 |
| 18 | ^x | 剪除 Cut |
| 19 | ^y | 不悔 Redo |
| 1a | ^z | 悔 Undo |

当然, 光标键, 翻页键, 甚至滚轮, 都可如期使用.

后现代的 Acme 当然是多窗口的, 而且绝没有重叠. 默认是两列, 每列内可以打开任意多个窗口, 每个窗口都有一行天蓝色底的提升栏. 滚动条在左侧, 和提升栏交汇的小方格, 在窗口内容未保存时是深蓝, 用鼠标左键移动此方格, 可以安排窗口新的位置, 单击可以扩大窗口, 而用中键或右键点击, 可以整列显示此窗口, 或显示其它窗口的提示栏. 在滚动条内左右鼠标键可以翻页, 或用中键点击和移动光条到所需位置.

练习时间到.

参照本章开始的图片, 在首行敲入 main.go, 用鼠标中键红底扫过“新 main.go”, 会出现两个窗口. 那个叫 +Errors 的窗口, 提示你此文件并不存在. 用鼠标左键拖动 main.go 窗口提示栏的方格, 移动到合适位置, 照图敲入我们的第一个 Go 源代码. 鼠标中键点击“存”, 深蓝方格消失. 点击“编”开始编译程序. 如果你看的够仔细, 图中的“你好”的引号是中文, 编译会有错. 鼠标右键点击 +Errors 窗口中main.go:4 的部分, 会直接跳到出错行. 修改, 存, 编. 没错, 好似什么都没发生, 用英文一说, Unix 的哲学就是 “No news is good news.” 也就是没事儿别烦我的意思.

在 main.go 的提示行敲入 ./ 鼠标右键扫过, 会出现一个窗口显示 main.go 所在的目录内容, 有个文件叫 main.8, 是编译的结果, 在其前敲入 8l, 再用鼠标中键扫过 ‘8l main.8’ 链接. 又是 good news. 中击‘读’, 会发现多出个 8.out.exe 是 8l 的结果, 在其前敲入 os -t, 扫过‘os -t 8.out.exe’, 会在 +Errors 窗口输出 “你好”. 基准编译环境检测正常. 而我们和 Go 也打了招呼了.

# 编译和包

活动一下. 鼠标中键扫过 “新 empty.go”, 在新窗口敲入 //. 然后中键点击“存”和“编”.

“empty.go:2: package statement must be first” 出现在“+Errors” 窗口, 告诉我们每个 go 程序要以 package 开始. 用鼠标从 +Errors 窗口剪下 package statement, 用鼠标扫过 // 把它贴到 empty.go 窗口. “关”掉 +Errors 窗口. “存 编” empty.go, 这次, 编译静悄悄地成功了.

编译的结果是 empty.8, 鼠标右键打开看看. 头几行用英语明白的告诉我们 go 的版本, package 名, 和 import runtime. 后边是一些显示乱码, 掺杂着一些英文词, 内容是压缩过的, 叫“中间码”.

执行 8l empty.8, 看到 +Errors 窗口出现 “8l: empty.8: not package main (package statement)”, 告诉我们需要 package main 才能继续. 好, 我们把 empty.go 换成 package main, “存 编”, 8l empty.8 这次告诉我们 “runtime.mainstart: undefined: main.main”

总结一下:

1. 编译就是个反复发现, 理解, 改正程序错误的过程
2. 每个 go 程序要以 package 开始
3. 编译的结果是个压缩的中间文件
4. runtime 自动 import 到此中间文件
5. 此中间文件需要 8l 链接才能得到可执行文件
6. 8l 需要一个包含 package main 的中间文件
7. 8l 还需要一个 main.main

解释一下. 计算机只懂机器语言, 而且每种机器体系的语言都不同. 我们可以不懂这么多机器的语言, 编译器懂就行了. 我们只需学一种 go 语言, 不同的 Go 编译器例如 8g, 6g, 5g 会帮我们翻译, 并同时提醒我们 go 语言的用法错误, 还能做些常见的优化. 这就是编译器的基本职责.

我们当然可以只用一个文件包含成千上万的 go 语句. 更方便自己的是把它们分成一些较小的文件, 都交给编译器自动组合在一起. 根据我们自己的需要, 一个或几个文件用来完成同一类功能的, 我们就声明 (也就是告诉编译器) 这些文件属于同一个"包 package". 当然, 编译的单位还是一个个的文件, "包"只是用来有效的组织这些文件的相互关系, 给以后或者别人使用时方便些. 例如, fmt 包, 集中了字符输出的功能, 在第一章 main.go 的例子中, 我们已经看到怎样 import "fmt" 这个包, 又怎样 fmt.Println("你好") 使用这个包里的一个函数.

再看看 main.go. 如果我们把第二行用 // 注释, 即 // import "fmt", 告诉编译器不需处理这行. 编译的结果会是错误 "main.go:4: undefined: fmt", 因为编译器必须知道 fmt 是什么, fmt.Println 又是什么. import 就是要"导入"这些信息. 打开 /n/c/go/pkg/windows_386/fmt.a, 查找 Println. 这下我们知道编译器从哪里得到这些信息了. 编译器 (8g, 6g 等) 用这些信息检查 go 程序类型的正确, 而链接器 (8l, 6l 等) 则从这个文件中提取需要的中间码 (例如 Println), 加上我们的 main.8, 一次过翻译为机器语言. 当然, 还有默认的 runtime.a 中我们需要用到的中间码. 这些 .a 的文件叫"库文件", 是因为它们"库存"预先编译过的中间码, 供新的 go 代码"导入" 使用. 以后的章节, 我们也会写自己的"库" .

还记得那个特殊的 main.main 吗? 链接后的机器语言被计算机执行时, 会从 runtime.a 提供的一段代码开始, 用来预先安排好运行环境, 之后, 会到 package main 中找 func main 来执行我们自己写的 go 语句. 这就是为什么 main.go 可以链接, 而 empty.go 不可以的道理. 确切的说, go 链接后的机器语言, 完整的包括所有所需的指令, 而不需动态的查找和调用其它指令 (DLL 等), 这里必须要有一个main.main 作为运行的入口. 这种静态链接, 和当代横行的动态链接 (DLL 等)形成强烈对比, 主要是为了最大限度的减少对随时可以变化的外部机器代码的依赖(俗称 DLL 地狱), 而代价只是较大的可执行文件而已.

鼠标执行 godoc 会弹出一个空黑的 CMD 窗口. 这是 Go 自带的文档服务器. 在浏览器敲入 http://localhost:6060 可以看到和 Go 的官网很像的主页. 而 http://localhost:6060/pkg/ 可以浏览所有已经安装在我们电脑上的 go 库. 6060 像不像 GOGO 呢?

有没有 package archive? 它下面只有两个目录. 我们不可以 import "archive", 因为根本没有 package archive. 它只提供个目录结构存放相关的包. 我们可以 import "archive/zip", 而在使用时, 只需指明最后的部分, 例如 zip.File,

当然, 目录结构的包也是可以的, 例如 import "hash" 和 import "hash/crc32", 此时, 公共

目录的包提供一个公共的接口, 而子目录的包提供特定的实现.

由于使用包名时不包括目录名, 那多个不同目录下的包重名怎么办? 例如, 我们写了自己的 zip.go, 编译为 zip.8

```
package zip
type File string
```

然后 main.go 改为 (不要敲入行号!):

```
1  package main
2
3  import "archive/zip"
4  import "./zip"
5
6  func main(){
7      var a zip.File
8      var b zip.File
9      _, _ = a, b
10 }
```

此时编译会有错误

```
main.go:4: zip redeclared as imported package name
previous declaration at main.go:3
```

zip 重复声明了. 后面的 a 和 b 不知道用那个 zip 的 File 了. 解决方法是把其中一个命名, 例如第4行我们自己的 ./zip 就叫 myzip, 而第8行的 b 是指 myzip 的 File就可以了.

```
1  package main
2
3  import "archive/zip"
4  import myzip "./zip"
5
6  func main(){
7      var a zip.File
8      var b myzip.File
9      _, _ = a, b
10 }
```

如果 import 时把某个包命名为点儿 . 则可以直接用那个包里的名字. 当然这样查找名字的出处时会麻烦, 也增加了重名的可能. 例如

```
import . "./zip"
var b File
```

另外, 多个 import 可以括号在一起. 例如

```
import (
    "archive/zip"
    myzip "./zip"
```

)
Go 还有一个简单的私有化规则. 只有英文大写开头的名字, 例如 Println 和 File, 才可以被 import. 其它的名字都是只限包内私用. 这样, 包可以提供一个稳定的对外接口, 而内部的实现可以很好的隐藏起来供局部的改变.

试着把 zip.go 和 main.go 的 File 换成 file, 看编译器会告诉我们什么. 编译器是我们最好的助手. 像 Go 这种静态编译强制类型的语言, 可以很早就帮助发现一些无心的或难以察觉的错误. 而那些动态的弱类型的语言 (不点名批评), 可能要等到运行时, 才能在最不恰当的时刻让最不能得罪的用户看到你最愚蠢的小错, 例如, 名字少写了一个字母…

最后, 是 Go 官方规范 里对 package 的定义:

Go 程序由包们链接而成. 每个包又是由一个或几个源文件一起构成, 声明属于包的, 在同一包里的所有文件都可存取的常量, 类型, 变量和函数. 这些元素可能可以'导出'并用于另一个包.

包的研究到此为止. 我们接下来研究的, 就是那些'元素'.

# 常量变量和函数

某人说总是有点不正常的 Geek 都有点数学倾向是正常的. 为了证明这一点, 我们写个 Go 程序来验证 "最美的数学公式" (http://en.wikipedia.org/wiki/Euler's_identity)

在 acme 中, 像 Π 这样的字符, 可以按一下 Alt 键, 再按 *p 的组合输入. 全部的组合在 /lib/keyboard (鼠标右击打开看看).

```
1  package main
2  import (
3      "fmt"
4      "math"
5  )
6
7  func main(){
8      π := math.Pi
9      r := math.Cos(π) + math.Sin(π)
10     fmt.Println("e^iπ^ + 1 = ", r + 1)
11 }
```

输出结果很接近可惜不是零.

$e^{i\pi} + 1 = 1.1102230246251565e{-}16$

这种不是因为 Go 的 Pi 不够精确. 在 Go 里, 像 math.Pi (去 godoc 看看) 这样的常数是不受精度限制的, 它已经高达小数点后 62 位了!

```
const  Pi  = 3.14159 26535 89793 23846 26433 83279 50288 41971
69399 37510 58209 74944 59
```

而变量 π, 以及 math.Cos 和 Sin 的参量, 都只是 64 位的 float64 类型. 不要以为是 Pi 的精度不够. 因为 Pi 是十进制的 $10^{62}$, 因为差不多 $10^3$ 是 $2^{10}$, 所以 math.Pi 是二进制的 200 多位, 而 float64 实际的精度只有二进制 52 位, 也就是说 math.Pi 小数点后 15 位都被截断了, 没有意义了. 不信, 自己可以定义自己的 const Pi 试试.

提到 Euler, 就顺便一提"变量", "常量" 和 "函数" 这些数学名词都是他的创造. 英文分别是 variable, constant 和 function. 在 Go 里速记为 var, const 和 func. 但尽管计算机和数学有很深的渊源, 它借用数学名词却代表不同的意思. 例如每个数学迷看到都皱眉的程序:

```
x = x + 1
```

根本不可能成立嘛! 因为要加这个常量 1, 便没有一个变量 x 的值能满足这个函数.

在 Go 和大部分的计算机语言中, 常量和变量按字面的意思, 就是值不会发生和可以发生变化的名字. 因为常量名字的值不会变化, 所以在编译时其名字都用其值作了适当替换. 而变量, 因为其值可以变化, 就需要编译器在 RAM 内存中给它安排个位置 (称为地址), 用变量的名字代表.

例如上边的 Pi 这个常量, 已经被 3.14 替换, 便没有记录在 main.8 的必要, 而 π 这个变量, 出现在 main.8 中, 其实也只是用来 debug 调试时使用, 因为此名字对程序的运行没有意义, 需要的只是编译器安排给它的内存位置(地址).

地址在哪里? 是相对于函数的. 数学的函数是一系列高度抽象的方程式表达式, 计算机不懂, 必须由程序员一步步的给出求解过程, 再由编译器翻译出机器语言, 才能让这一堆石头和铜做出来的东西过过电, 好歹总能给我们个结果. 所以函数在其它某些计算机语言里也称为 procedure 过程. 但和数学函数一样, 理想的计算机函数, 应该是自闭的. 就是说对每个特定的输入, 会有个确定的输出. 例如 math.Sin 和 Cos. 当然, 除了这种"纯"函数, 计算机还要和环境(包括人)交流, 也会使用函数, 但此时的输入和输出, 就不只是函数声明时的"参数和返回值"了.

地址在哪里? 再等一下. 先说"函数声明时的参数和返回值". 在 Go 里, 包是较大的代码组织结构, 再下一层, 就是"函数". 例如 main.main, 就是 Go 要求我们要把我们程序的入

口代码安排在 main 包的 main 函数里. main 函数的声明是

```
func main() {
```

其实 { 不是函数声明的一部分, 此处是为了清楚表明 { 之前的部分是声明了一个类型是 func, 名字叫 main, 参数是(), 返回值没有类型的东西. 一旦这样声明, Go 的编译器就吧 main 这个名字和其类型, 参数, 返回值类型联系在一起, 不再接受同名的声明了. 而 { 和 其配对的 } 之间的语句, 则是这个函数的定义.

中键扫过 godoc math Sin 看看 math.Sin 的声明:

```
func Sin(x float64) float64
```

math 包的 Sin 函数需要一个 float64 类型的参量 (叫不叫 x 没关系), 返回一个 float64 类型的值.

函数的输入参量和返回值都是可变的, 都在内存中有地址, 都是变量. Go 是这样安排函数变量的地址的: 程序运行时, 从操作系统管理的未用内存中, 申请得到一块连续的空间, 叫作 stack 栈. 执行到某个函数时, 先把此函数的返回地址 (可以认为是行号), 以及参量和此函数内定义的变量, 写入栈. 它们占用的这一段栈里的内存空间, 称为一个 frame 帧. 当此函数执行完毕时, 此帧不再需要. 此函数临时占用的那段栈空间, 会被下一次运行的函数使用.

运行下面程序, 观察一下函数运行时函数对栈的使用情况:

```
1  package main
2  import (
3      "fmt"
4      "runtime/debug"
5  )
6
7  var N = 2
8
9  func f(n int) int {
10     fmt.Println("* * * * 进 f(", n, ")")
11     debug.PrintStack()
12     if n > 1 {
13         n = n * f(n-1)
14     }
15     fmt.Println("* * * * 出  f(", n, ")")
16     debug.PrintStack()
17     return n
18 }
19
20 func main () {
21     f(N)
22 }
```

Go 使用调试器 gdb, 我们会在以后介绍. 尽管 gdb 很有效, 但基本的观察和推理比学会使用调试器要重要. 理解程序的结构, 运行时的状态和转换, 在适当的位置加上一些跟踪用的 print, 就好比中医的"望闻问切", 也是行之有效, 并且这种'实时的', '动态的'调试有时非常必要的. 而动用 gdb 的时候, 就像 X光, 看的很清楚, 但也只是某一瞬间的'快照', 且对活体有较强的副作用, 要慎用.

仔细观察上面程序的输出, 对照程序流程, 我们在第 9 行使用 debug.PrintStack 输出栈的状态时, 栈里有 4 个帧: f 函数, main 函数, mainstart 和 goexit. 正在使用 debug.PrintStack 的 f 函数处在栈的顶端, 下面压着正使用 f(2) 的 main 函数, 再下面压着正使用 main.main 的另一函数等等. 所以, 又有"压栈 Push" 和 "弹出栈 Pop" 的说法.

下面会有一个图说明, 但现在试着自己想象这个函数栈的操作.

语句 n * f(n–1) 中, 有使用了 f 函数自己. 这时 n 是 1, 此时的栈里有 5 个帧. 最上面的是 n = 1 时的 f 函数在执行 PrintStack, 下面压的是 n = 2 时的 f 函数在执行 n * f(n–1) 等等. 这种函数重复使用自己的设计, 称为"递归". 每次递归调用时, 可以看到栈会多出一个帧, 所以, 必须保证有一个结束, 不然, 所有的栈内存都会被占用, 导致"栈溢出". 此处, n 每次递归会减一, 到 n = 1 时结束, 可以观察到栈在一层层退出的情况.

另外, 变量 n, 作为函数 f 的参量, 出现在 f(1) 和 f(2) 的帧中. 这一方面说明, 参量的值, 还有函数内部使用的其它变量(称为局部变量)的值, 只存在于函数运行时动态得到地

址中, 并且独立于其它函数的帧, 包括自己的另一次调用. 另一方面, 也说明参量和局部变量的名字, 所指的不是固定的某一内存地址, 而是相对于帧顶的一个距离(称为偏移 offset). 由于编译器帮助我们管理, 我们 Go 编程时, 只需对这些有个概念上的了解, 就可以辅助我们的调试工作了.

在函数内部的变量, 只供此函数使用, 称为局部变量, 其名字对应的是对应函数帧顶的偏移, 只有在函数运行时, 其值才会在内存出现.

还有一种写在函数外的变量, 例如上面程序的 "N", 属于包内公用, 称为"包变量". 如果此变量是大写英文字母开头, 还可由其它包导入共用. 它们的名字指向一个程序开始运行时便已经固定的内存地址. 我们可以用 6nm 8.out.exe | grep 'main.N' 看到编译器分配给 main 包的这个 N 变量的地址.

讲了这么多编译器如何区别对待常量变量, 无非是告诉大家语言中的硬性的规定, 都是或着为了计算机运行程序时更有效, 更安全, 或者为了程序员思考问题编写程序时更有效, 更安全. 理解规定背后的道理, 比强记这些规定, 更有效, 更安全.

下面一章, 我们就来看看 Go 语言的类型设计, 如何做到"更有效, 更安全".

# 数值类型

上一节我们一直讲常量只是个值的替换, 可以没有类型. 而变量和参量的名字代表内存的地址, 那当我们用 1 和 100000000000000 这样的值写入某个变量和参量的地址时, 它们所占据的内存是大小不同的. 不知道每个变量需要多大的空间, 就无法有效的分配连续的空间, 供不同的变量使用, 而不会在写入一个过大的值时, 踩到另一变量的脚.

当然, 顺便一提, 也有很多语言, 变量没有类型, 但值有类型. 此时, 已知的值会得到合适的空间, 而变量的名字再指向这个值的地址. Go 没有采用这种方案, 因为, 编译器可以使用变量的类型做安全检查, 而这种方案, 牺牲了编译时的宝贵的把关, 容易造成运行时的错误.

Go 里占用空间最小的类型是 byte, 字节. 现代电脑的内存至少有几百亿个字节, 我们大概可以想象程序可以玩的棋盘有多么的大. byte 也写为 uint8. u 是 unsigned, 代表没有符号. 8 是二进制的位数, 每位或 0 或 1, 这样一个字节是可以表示从 00000000 到 11111111, 即十进制的 0 到 255 的 256 个数. 可以理解, 我们需要空间较大的类型, 来表示更多的数. 最好是字节的整倍数, 所以 Go 还使用 16位, 32位, 和 64位 整数类型(整

型).

uint8, uint16, uint32 和 uint64 这四种无符号的(整型) , 还有对应的 int8, int16, int32 和 int64 四种有符号的整型. 所谓“有符号”, 就是用最高的一位作为符号位, 0 代表正数, 1 代表负数. 例如, uint8 可以表示 0 到 255 个正数, 而 int8 表示的是 –128 到 127 的整数. 同样的 8 位有两个类型, 所以同样的 11111111 如果是 uint8 类型的变量, 其值表示的是 255. 而如果变量类型是 int8, 则代表 –1, 在做四则运算和比较两数大小时, 就会有不同的结果. Go 允许我们做数值类型的转换, 但要求我们必须像个函数调用一样明确的写明, 例如,

```
1  package main
2  import (
3      "fmt"
4
5  )
6
7  func main () {
8      v := uint16(0x01F0)
9      fmt.Println(int8(v))
10     fmt.Printf("%x\n", uint32(int8(v)))
11     fmt.Printf("%x\n", uint32(uint8(v)))
12 }
```

解释一下. 0x 后跟的是十六进制是, 也就是每位可以是 0 到 15, 在 Go 里 10 到 15 分别有 a 到 f 代表. 之所以有这种数值表示法, 是因为早期的程序员发现他们有必要一眼看清每个值对应的二进制数的 0 和 1 的变化. 十进制的每位不可以直接对应二进制, 而十六进制的每位, 就是 0000 到 1111 这四位二进制, 直观又简洁.

这样, 常量 0x10F0 被转换为无符号 16 位整数, v 也是这个类型. 其值是二进制的 0001 0000 1111 0000. 当 int8(v) 时, 最后的 8 位, 也就是 0xF0, 被解释位有符号数, 0xFF 是 –1, 所以 0xF0 是 –16. 当 0xF0 作为有符号数扩展时, 最高位, 也就是符号位延展, 所以才有 uint32 后的结果. 而如果是无符号数, 例如 0xF0 的变量是 uint8 类型, 因为没有符号位, 所以使用 0 延展, 那 uint32 后, 是 0x000000F0, 即仍旧是 0xF0.

还有, Printf 的 f 是 format “格式化” 的意思, 其 %x 代表使用十六进制数显示后面参量的值, 代表一个回车.

既然有 64 位的整数类型, 而计算机的内存又是极大的,还有必要提供占用内存较小的类型吗? 答案是大部分时候都不需要. 所以, Go 还有 int 和 uint 类型代表足够大的有符号和无符号整型. 当不需担心数值“溢出”时, 通常就用它们.

什么是“溢出”? 例如 uint8 类型的最大值是 255, 如果这个变量加一, 成了 256, 即九位的

1 0000 0000, 取最后 8 位, 结果这个 255 + 1 = 0, 计算出错. 计算机知道, 电脑里会有个溢出标记, 可惜的是这只针对某些类型运算, 例如两个 32 位数相加. 所以, Go 选择忽略这个标记. 只是这会造成难以察觉的, 运行时才会出现的错误. 编程时要小心小心!

除了 10 种整数类型, Go 还支持 2 种实数类型, 和 2 种复数类型. 分别是 float32, float64, complex64, 和 complex128.

实数类型在计算机里称为"浮点数". 是相对于另一种计算机里的"定点数"而言的. Go 里不直接支持定点数, 因为它其实是程序员自己可以规划的, 例如拿个 uint32 的变量, 规定最后 8 位代表小数部分, 高位的 24 位代表整数部分. 这样加减法的运算结果小数点的位置是固定不变的. 所以, 定点数就是可以用整数类型, 当然其位数, 也就是能表示的数值范围, 也就很有限了.

浮点数是专门用来表示范围极大的数值的, 但牺牲的是精度. 例如, float32 规定 32 位二进制中, 1 位表示正负, 8 位表示幂, 只剩下 23 位表示小数的部分. 我们知道 10 位二进制几乎等于 3 位十进制 ($2^{10}$=1024, $10^3$= 1000). 所以 float32 的精度, 也就是十进制的小数点后七位. 而幂级也是最大到十进制的 38. 具体的说, float32 能表示的数的范围, 是在 $\pm 1.175494351 \times 10^{-38}$ 到 $\pm 3.4028235 \times 10^{38}$ 之间. 而当幂级是 23 时, float32 代表的是介于 8388608 到 16777215 的一个整数, 此时的精度最高. 离开此区间越远, 精度越低, 例如, 幂级是24时, 表示数值范围增大一倍 (从 16777216 到 33554430) , 而精度降低一半 (数值间距为 2, 无法表示奇数).

float64 可以表示范围更大, 精度更高的数, 但同样对其中的绝大部分数而言, 是无法表示其精确数值的. 经典的例子是 0.1 + 0.2 不等于 0.3

```
1 package main
2 import (
3     "fmt"
4 )
5
6 func main(){
7     var a, b = 0.1, 0.2
8     fmt.Println(a + b, "≠", 0.1 + 0.2)
9 }
```

但注意常量是有无限精度的, 所以常量的 0.1 + 0.2 = 0.3, 而变量受精度的限制, 只能是 0.30000000000000004 (因为 0.1 只能在有限二进制位内近似)

浮点数主要用于数值计算, 而复数 complex64 和 complex128 则分别是由两个 float32 和两个 float64 代表其实部和虚部的值. 因为 Go 支持 cmath 包, 和 1i 这样的写法, 所以就

有下面 (0+1i) = (0+1i) 这样的运算和输出:

```
1 package main
2 import (
3     "fmt"
4     "cmath"
5 )
6
7 func main(){
8     fmt.Println(cmath.Sqrt(-1), "=", 1i)
9 }
```

最后, 我们看一下 Go 语言规范对数值类型的规定:

“数值类型”表示一组整数或浮点数的数值. 预先定义的, 与体系结构无关的数值类型有:

```
uint8 八位无符号数 0 到 255
uint16 十六位无符号数 0 到 65535
uint32 三十二位无符号数 0 到 4294967295
uint64 六十四位无符号数 0 到 18446744073709551615

int8 八位符号数 -128 到 127
int16 十六位符号数 -32768 到 32767
int32 三十二位符号数 -2147483648 到 2147483647
int64 六十四位符号数 -9223372036854775808 到 9223372036854775807

float32 IEEE-754 三十二位浮点数
float64 IEEE-754 六十四位浮点数

complex64 由 float32 实部和虚部组成的复数
complex128 由 float64 实部和虚部组成的复数

byte     是 uint8 的常用别名
```

n 位整数的值是 n 个位宽, 并使用二的补码表示.

另外还有一组预定义的, 由具体实现规定的数值类型:

```
uint     三十二位或六十四位
int 和 uint 尺寸一样
uintptr 一个足够大的无符号整数, 足以存放指针类型数值的不加解释的位
```

为了避免移植问题, 所有数值类型都是不同的, 除 byte 之外, 因为它是 uint8 的别名. 当不同的数值类型在表达式或赋值时混用时, 必须转换. 例如, int32 和 int 是不同的类型,

尽管在特定的体系结构里, 它们可能有同样的尺寸.

规范里使用了很多我们还没有介绍的概念. 现在重要的是明白, Go 要求程序员使用不同数值类型的变量, 并明确的转换, 是为了精确的声明每个变量能代表的数值的尺寸和精度, 从而有效的利用内存, 并通过编译器进行最大限度的安全检查. 这也是 Go 除了数值变量之外, 要求其他变量也要有明确类型的用意 — 有效, 安全.

下一节, 我们看看另一种非常常见的变量类型 — 字符串.

# 字符串

Unicode 是国际标准的大字符集, 包含了包括中文在内的几乎所有文字的编码. 所谓"编码", 就是每个字在 Unicode 都有唯一的一个位置, 用数字代表, 例如, "一" 的 Unicode 编码是 0x4E00, 必须至少用 16 位数的变量表示. 在 Go 中, 使用的是 int, 但对英文国家而言 (Go 的发源国), 英文字母数字和标点符号的 Unicode 都在 0 到 127 的范围内, 使用32位表示就太过浪费, 用 8 位的 byte 就足够了. 所以, 在 Go 中, 字符串, 也就是零到多个字符组成的一串词, 有了双重身份: 整体看是一串 byte, 单独看每个字符, 是 int.

具体怎样实现? 像 0x4E00 这样的两个字节, 到底是 0x4E 先, 还是 00 先? Go 的方案是使用Unicode 的 UTF–8 编码把 int 的字符放入 byte 的字串中. 再具体的讲, 就是 Go 的祖国最常用的 128 个字符还是只用一个 byte, 保持其最高位是 0. 其它的 Unicode 字符, 则使用两到三个 byte, 每个 byte 的最高位都是 1, 并且开始 byte 和 后续 byte 的高位模式不同, 从而能唯一界定每个字符, 如下表:

```
0 - 0x7F          0xxxxxxx
0x80 - 0x7FF      110xxxxx 10xxxxxx
0x800 - 0xFFFF    1110xxxx 10xxxxxx 10xxxxxx
```

这样我们祖国的几千个文字, 每个都要用三个字节才能表示. 相对于国标的 GB 2312 的每字两个字节, 是增大了三分之一. 可这也是和国际接轨的需要.

```
1  package main
2  import (
3      "fmt"
4  )
5
6  func main(){
7      s := "utf-8编码"
8      fmt.Println(len(s), s)
9      fmt.Println([]byte(s))
10     fmt.Println([]int(s))
11 }
```

输出是

```
11 utf-8编码
[117 116 102 45 56 231 188 150 231 160 129]
[117 116 102 45 56 32534 30721]
```

len 函数是取得字串的长度 — 不是字符的个数 — 而是 utf–8 编码的字节数. 此处是 1 x 5 + 3 * 2 = 11.

当把字串转换为 byte 数组 (稍后讲) 再显示时, 可以清楚的看到每个 byte 的数值, 前 5 个 byte 小于 128, 而后边每三个 byte 是一个 utf–8 编码的汉字, 每个 byte 都大过 128, 并且只有每个的第一个字节大过 192 (0xC0).

当用 []int 转换为 int 数组后, 每个字符的 Unicode 编码数值就清楚的显示出来了.

上面程序字符串常量使用双引号括起. 对应的字符串变量类型称为 string. 和数值类型不同, string 的变量名字代表的内存地址, 放的不是那些字符, 而是那些字符存放地址, 以及字符串的长度. 这样, 就需要编译器预先把字符串常量在内存中存放好, 才能动态的在函数运行时的栈中, 由 string 类型的变量引用它.

其实, 对于较大的数值, Go 的编译器也不会直接替换, 写在指令流里面, 也是先写在一个固定地址, 让机器指令间接引用. 这样, 每个编译链接好的可执行的程序, 都包括指令段和数据段, 称为 Text section 和 Data section. 还可能会有一个 bss 段, 存放初始数值为零的那些包变量的地址. bss 没有必要放在程序文件里 (反正已经知道都是零了), 而是直接在程序运行前, 分配一块内存并全部清零. 当然, 指令和数据段的区分, 只是概念上的, 某些具体实现 (ARM), 会把数据和指令混在一起. 重点是, 这些常量数据, 包括字符串, 是不可以修改的.

不只是字符串常量, 包括用 string() 转换的字符串, 没有预先分配在数据段中, 也是不可以修改的. 例如:

```
1  package main
2  import (
3      "fmt"
4  )
5
6  func main(){
7      a := "utf-8编码"
8      b := []byte(a)
9      c := string(b)
10     c[0] = 'U'
11     fmt.Println(c)
12 }
```

编译时会出错 “cannot assign to c[0]”. 要在 byte 数组那边改才可以:

```
b := []byte(a)
b[0] = 'U'
c := string(b)
```

现在我们知道, string 类型的变量 a 指向存放在数据段的字符串常量 “utf–8编码”. 那变量 b 和 c 在程序 stack 栈中的地址存放什么呢? 这又要介绍 Go 的运行环境 runtime 管理的另一块内存, “heap 堆”.

我们知道函数运行时会在程序的栈 stack 中有个私有的帧 frame, 存放参量和局部变量. 而程序的堆 heap 则是所有函数公有的. 像上例中需要内存存放转换结果时, 编译器会安排一段机器指令, 告诉 Go 的 runtime 在 heap 中分配合适尺寸的内存, 并拷贝. 也就是从 a 到 b 再到 c, 会在 heap 分配两个同样尺寸的内存 (因为 string 和 []byte 的底层表示是一样的), 拷贝两次. 对于很长的字符串, 这种分配和拷贝是很耗时间的. 但由于 string 不可修改, 到 []byte 的转换就有必要. 只是我们程序员要尽量安排, 减少这种转换.

原则上, 这种看起来很无谓的 string 和 []byte 转换是有必要的. 因为 string 内容不可改写, 如果转换到 []byte 使用同一内存, 则多了一个 []byte 的变量指向它, 而 []byte 是可以改写的, 这样就造成之前 string 的变量指向的“不可改写”的内容, 被背后改动了, 可能造成不可预知的后果. 为了避免这种冲突, 目前 Go 的编译器强制拷贝. 而将来 Go 的编译器能不能分析程序, 发现和回避冲突, 优化某些情况下的重用内存, 则要依靠 Go 的编译器的作者 Ken Thompson ( Unix 的发明人, C 编译器的作者), 及其同事, 再显神通了.

可是, 变量 abc 本身存放在栈上, 它们指向的内容存放在堆上, 当函数程序完毕, 栈中此函数的帧和它的所有变量都不在有效, 那它们指向的堆上的内容怎样清除? Go 的 runtime 有 GC, 废品回收器, 会定期的检查堆上的数据还是否有效, 即是否还有变量指向. 无主的数据所占的内存就会被回收. 同时, 有主的数据, 可能被移动到新的位置, 使得无用的内存能连成一片, 供下次分配较大内存时使用. 这有要求变量的内存被改写,

以便指向其新的数据位置. 所以, GC 是个相对复杂的技术. 目前 Go 的 GC 只保证可靠, 但效率不高, 有时在大量废品产生时, 不能快速处理. 以后 GC 会得到很大提高, 但现在的策略, 是要求我们程序员, 尽量少制造废品. 这就要求我们程序员, 尽量清楚自己的变量对内存的使用情况. 这也是为什么我不厌其烦的讲的这么"底层" 的原因之一.

顺便一提, 上例还出现了单引号括起的字符 'U', 是 int 类型. 代表 Unicode 的编码.

至此, 我们应该大体上对 Go 的源代码结构, 和运行时内存的分配有个整体的把握. 图示而言, 源代码结构是:

目录
    包
        文件
            package
            import
            var
            const
            func
                var
                const
                控制语句

而程序运行时的内存大体是:

| 指令段 | 数据段 | bss | 堆 .... 栈 |

通常, 栈是从程序内存的最高地址向下长, 而堆则对着向上长, 这样它们可以共享中间的空白内存.

这样, 我们知道常量出现在指令段或数据段, 变量出现在 bss, 和堆栈中. 数值变量的内存直接存放数值, 而字符串变量以及数组变量的内存, 存放的是在堆上分配的内容的地址及其长度. 函数内的变量和参量, 存放在栈上, 函数运行结束时丢弃. 堆上存放的数据, 由 runtime 的 GC 定期清理. 类型保障编译器正确的分配和检查变量的内存使用, 类型转换我们必须明确的告知编译器以保证类型检查. 真正有效的使用内存, 编译器可以帮忙, 但最有效的是我们编程的方式.

大体上是这样, 以后还会细化. 我们先放下数据, 下节我们来看算法 — 控制语句.

# 控制语句

Go 编程的本质 — 是数据和算法.

算法的本质, 是赋值, 判断和跳转.

“赋值”, 就是改变变量的值. 例如 x = x + 1, 不是让我们求解数学方程式中变量 x 的值(无解), 而是说, 读变量名字 x 所代表的内存的数值, 加一, 再写回变量名字 x 所代表的内存. “=” 是赋值, “写”的意思.

读写很重要吗? 赋值很重要吗? 是的. 对计算机而言, 哪怕是最复杂的计算机, 也能抽象为最简单的“图灵机”. “图灵机”只有三样东西: #. tape, 即内存. 每个地址存放一个有限的数值 #. head, 即当前内存地址 PC, 可以读写此地址内的数值, 也能每次移动到上一个或下一个地址. #. table, 即指令表, 根据 head 读到的数值, 决定 #. 写回新的数值 #. 移动 head 到上一个, 或下一个地址.

当这些动作在一瞬间完成时, “机械”的电脑就被某些人神话为有了“智能”了.

上面的第三步, 就包括了赋值, 判断, 跳转三种基本指令.

Go 赋值时 “=”, 还有 “:=” 代表的是声明且赋初始值, 即把变量的类型说明和赋值一起完成.

Go 的判断用 “if” 和 “for”. 根据后边的表达式的真假, 决定是继续下条语句, 还是跳过整块的 block.

block 是用对应的大括号 {} 括起的语言. 与 func 函数对大括号的使用是一样的. 例如:

```
1  package main
2  import (
3      "fmt"
4  )
5
6  const N = 10
7  func main(){
8      a := 2
9      for a < N {
10         b := 2
11         for b < a {
12             if a % b == 0 {
13                 break
14             }
15             b = b + 1
16         }
17         if b == a {
18             fmt.Println(a)
19         }
20         a++
21     }
22 }
```

质数是只能被 1 和自己整除的正数. 为了求得 N 以内的质数, 我们多次运用 for 和 if 判断和跳转. 第一个 for a < N {} 在 a 的值比 N 小的时候, 一直执行 {} 里的语句, 称为“循环”. 注意到其 } 前的 a++, 其实和 for b < N {} 循环最后的 b = b + 1 是一样的, 都是表示变量的数值加一, 从而可以结束循环. 而 a % b 是变量 a 的值除以变量 b 的值的余数. “==” 判断是否等于后边的数 0. 如果是, 则 a 能被别的数除, 不是质数, break 跳出 for 的循环, 即 for b < N {}, 否则执行 b++ 然后 } 会跳转到 for b < N 重新判断是否循环.

如果循环忘记写 a++ 等, 就会永远满足 a < N, 造成循环没有办法结束. 所以, 写 for 时, 要特别留神结束条件. 由于这种用一个变量来控制循环结束的方式很常用, Go 提供了一个写在一行的方法: for a:=2; a < N; a++. 这样, 上例可以清晰的改为:

```
1  package main
2  import "fmt"
3  const N = 10
4  func main(){
5      for a := 2; a < N; a++ {
6          for b := 2; b < a; b++ {
7              if a % b == 0 { break }
8          }
9          if b == a { fmt.Println(a) }
10     }
11 }
```

编译出错, “undefined: b”. 为什么?

这些 {} 块, 决定了一个变量的“作用域”, 即有效期, 从它声明之处到这个块的结束终止. 实现上类似函数的帧, 进入作用域时, 编译器生成的指令在 stack 栈上分配此变量的内存, 离开作用域时, 此块内存不在需要, 会被下次重用.

我们讲过 “:=” 是声明同时赋初始值, for 的块从 for 到对应的 } 结束. 所以, 变量 b 在第 9 行已经无效了. 一个改法是

```
for a, b := 2, 2; a < N; a++ {
    for ; b < a; b++ {
```

仍使用三段式的 for 语句, 但外层循环多个循环控制变量同时赋值, 内层循环则没有循环控制变量赋值. 这几种形态的 for 语句都会常常用到.

if 判断语言还可带有 else 的分支, 此时, 其内部声明的变量的作用域延展到 else 的块. 在 Go 中, else 出现的比 if 少, 因为 Go 鼓励一种编程风格, 即 shortcut — 使用 break, continue, return 这样的语句, 在 if 条件满足时提前结束. 条件不满足时, 跳过 if 块继续, 此时, else 就显得多余了.

当然, 如果是多个分支, 可以使用 else, 例如:

```
if a > b {
    return 1
} else if a == b {
    return 0
} else {
    return -1
}
```

但这种多重分支的情况, 使用 switch 更清晰. 例如:

```
switch {
case a > b: return 1
case a==b:  return 0
case a < b: return -1
}
```

if 和 switch 语句也可在判断之前执行一条语句, 通常是个赋值. 例如:

```
if e := f(); e != "" {
```

执行函数和检查返回值在一起完成. 函数 f 出错时返回一个字符串解释错误, 当没有错误时, 就返回个空字符串. 这样, 上面的 if 能在函数 f 执行完后, 立即进行出错处理. 而变量 e 的作用域, 也只限于这条 if 语句的出错处理中使用, 不会影响到外层同名变量的值.

上面这句话, 正是块和作用域的一起完成的使命: 用“块”把变量名字包裹起来, 即不会

受外部同名变量的影响, 也不会影响到外部同名变量. 这样, 我们写程序时, 起变量名会很容易, 但一不留神, 还是会犯一种常见的错误. 此错误最常见的起因是 Go 允许函数给返回参量取名, 并直接 return 返回那些参量的值. 例如:

```
package main
import "fmt"

func f() string {
    return "小心"
}
func g()(e string) {
    if e := f(); e != "" {
        e = "再小心"
    }
    return
}
func main(){
    fmt.Println(g())
}
```

执行结果是 “再小心” 吗? 不是. 是空行. 为什么?

因为我们不小心, 把 if 语言的 e 变量重新声明, 隐藏了同名的外部的函数返回参量 e. 这两个 e 是完全不同的变量, 内存中的地址完全不一样, 函数返回的, 仍是空的那个 e, 而不是赋值为 “再小心” 的那个 e. 改为下句再试试:

```
if e = f(); e != "" {
```

这样, 我们终于有足够的知识, 可以结合 godoc, 试着自己写个的包了. 当然, “写”, 和前面的示例程序一样, 是指我们要自己敲入所有的字符, 而不是整段的剪贴. 更不是拿眼睛扫过就完了. 根据我的经验和观察, 看过的过目就忘, 照抄的有点印象, 而自己动脑写的才会历久恒新. 再者, 敲入就是模仿, 是能动的发现细枝末节, 而模仿出了错误再参考修正, 也是非常好的练习. 好. 我们开始:

```
mkdir base58; touch base58/encode.go
```

每个包最好有自己的目录, 很快我们会写 Makefile, 和自动测试文件 base58_test.go. 现在, 先看 base58.go

```
1
2
3
4
5
6
7
```

```
/*
  Package base58 implements encoding byte array to 58 alphanumerics.
 0OIl are not used as they look the same in some fonts.
 Punctuations are not used so double click selects the whole string.
*/
package base58
import (
    "big"
)

const             table                                          = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz"

// Encode encodes src to dst in base58.
// It returns the number of bytes written.
// It is in backwards so dst[n:] is valid.
// Result is at most 138% (about 2^8 / 2^6) larger.
// No error checking. Panic if dst is too small.
func Encode(dst, src []byte)(n int) {
    x := new(big.Int).SetBytes(src)
    x.Abs(x)
    y := big.NewInt(58)
    m := new(big.Int)
    n = len(dst) - 1
    for x.Sign() > 0 {
        x, m = x.DivMod(x, y, m) // x = x/y, m = x%y
        dst[n] = table[int(m.Int64())]
        n--
    }

    // Leading zeroes encoded as '1'
    for i := 0; i<len(src) && src[i] == 0; i++  {
        dst[n] = '1'
        n--
    }
    return n+1
}
```

这个包的 Encode 函数, 是把参量 src 指向的 byte 数组重新编码, 写入 dst 指向的 byte 数组, 并返回 dst 的有效长度.

由 /* 和 */ 括起的部分, 以及 // 开始到行尾的部分代表注释, 不被编译. Go 的风格不欢迎冗长的注释. 但鼓励在每个包的开始, 和每个函数的开始, 紧接着 package 和 func 语句, 对此包或此函数的用法和局限, 给出简短的说明. 按照 Go 的观点, 源代码本身是最可靠的说明. 如果代码不够清晰易懂, 尽量修改代码, 而不是详加解释. 注释在清晰流畅的代码中, 会像噪音一样刺耳, 应谨慎使用.

就像 int32 占四个字节的内存, int64 占八个字节的内存, big 包把一块多个字节的内存当作一个数, 做四则运算. 这样, 我们把这个数连续的除以 58, 得到的介于 0 到 57 的余数, 查表 table,就可以得到一个对应的字符. 当被除数为 0 时结束, 结果字串就是那块内存内容的 base58 编码. 作为特例 (有注释), 那块内存开始的 0, 都被作为 '1' 填补在结果字串的前端.

像 new(big.Int).SetBytes(src) 这样的语法, 我们下节会讲到. 但意思大概可以猜出来, 是 new 初始化一个整型的 big 内存, 填上 src 的内容.

像 dst[n] 这样的语法, 是读 byte 数组 dst 内容的第 n 个元素. 同样 table[int(m.Int64())] 是把 int64 类型的余数转换为 int 类型的数作为位移, 用来对应读取字符串 table 的字符. 注意, 数组和字符串的位移都是从 0 开始的.

base58 的包是要被别的包使用的, 当然我们可以写个 main 包来测试它. 但 gotest 提供了自动测试的方法. 这需要我们在同一目录下写个 Makefile:

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

TARG=base58
GOFILES=\
    encode.go\

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

然后再写个以 _test.go 结尾的测试文件 base58_test.go, 同样也是 package base58, 但使用 testing 包来自动运行其中所有以 Test 开头的函数.

```
1
2
3
4
5
6
7
8
9
1package base58
0import (
1    "testing"
1)
1
2type test struct {
1    en, de string
3}
1
4var golden = []test {
1    {"", ""},
5    {"\x61", "2g"},
1    {"\x62\x62\x62", "a3gV"},
6    {"\x63\x63\x63", "aPEr"},
1            {"\x73\x69\x6d\x70\x6c\x79\x20\x61\x20\x6c\x6f\x6e
7\x67\x20\x73\x74\x72\x69\x6e\x67",
1"2cFupjhnEsSn59qHXstmK2ffpLv2"},
8        {"\x00\xeb\x15\x23\x1d\xfc\xeb\x60\x92\x58\x86\xb6\x7d
1\x06\x52\x99\x92\x59\x15\xae\xb1\x72\xc0\x66\x47",
9"1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L"},
2    {"\x51\x6b\x6f\xcd\x0f", "ABnLTmg"},
0                    {"\xbf\x4f\x89\x00\x1e\x67\x02\x74\xdd",
2"3SEo3LWLoPntC"},
1    {"\x57\x2e\x47\x94", "3EFU7m"},
2                {"\xec\xac\x89\xca\xd9\x39\x23\xc0\x23\x21",
2"EJDM8drfXA6uyA"},
2    {"\x10\xc8\x51\x1e", "Rt5zm"},
3                {"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00",
2"1111111111"},
4}
2
5func TestEncode(t *testing.T) {
2    dst := make([]byte, 100)
6    for _, g := range golden {
2        n := Encode(dst, []byte(g.en))
7        s := string(dst[n:]);
2        if s != g.de {
8            t.Errorf("Bad Encode. Need=%v, Got=%v", g.de, s);
2        }
9    }
```

```
3}
0
3
1
3
2
3
3
3
4
```

此时, 执行 gotest, 会自动编译链接运行此目录下所有 _test.go 结尾的文件里所有的 Test 开头的函数. 结果或者是显示 t.Errorf 给出的出错信息, 或者是 PASS 自动测试通过.

当然, 可能会有大段大段未接触过的东西. 不用着急. 到此为止, 我们已经介绍完了 Go 的基本类型和基本控制语句. 我们已经走了一半的路了. 休息一下. 下面的章节, 会逐一介绍它们的组合, 和更多的语句及其用法.

# 数组和切片

我们一直讲 byte 数组, 却没有讲什么是数组 array. 因为它太简单了. 照字面意思, 就是一组数. 严格的讲, 是一组类型相同的内存组合成的一块内存. 使用 [] 表示, 后跟类型. 所以 [2]byte 就是由两个 byte 占用的一片内存,. 而 var a [2]byte 则声明变量 a 指向由两个 byte 占用的一片内存, 初始值都为 0. 这两个 byte 分别可以用 a[0] 和 a[1] 来读写. 例如 a[0] = a[1]. 数组的长度在声明时就已经确定, 还可以用 len 函数得到, 例如 len(a) = 2.

数组要求在 [] 中给定元素个数, 和数组所用内存的地址, 一起记录在数组变量中. 所以, 数组变量类似于字符串变量, 都是间接的指向它们的内容. 但和字符串变量不同, 数组内容是可以改变的, 在数组变量用作函数参量时, 会拷贝其内容. 对于较大的数组, 这种拷贝很费时间, 并且如果函数修改了数组的内容, 又要返回此数组, 又多了一次拷贝.

并且, 数组上固定大小的. 很多适合我们分配一个数组的内存, 是为了作为缓冲区, 每次添加新的数据. 这样, 使用数组变量指向这个缓冲区, 我们还需要一个单独的变量, 记录下一次数据添加的位置. 在 Go 中, 专门有一个称为 “slice 切片”的变量类型, 综合了上述的两种变量. 并且, 在作为函数参数传递时, 拷贝的只是切片变量本身, 而不是它指向的数组内容. 这样, 就和数组变量互为补充. 术语上讲, 数组变量是“alue 值”传递, 而切片变量是“reference 引用”传递.

而上一节 base58 例子中的 src, dst []byte 就是切片变量. 再看一个例子:

```
1  package main
2  import "fmt"
3
4  func main(){
5      var a [4]byte
6      var s []byte
7      s = a[1:3]
8      s[0]++
9      s[1] = a[1] + 1
10     fmt.Println("a:", a)
11     fmt.Println("s:", s)
12
13 }
```

a 作为数组变量, 声明时给定元素类型和个数: 4 个 byte. 声明本身就已经分配了内存, 并全部清为 0. 如果要初始为其它的值, 可以类似这样:

```
a := [4]byte{0, 1, 2, 3}
```

而 s 作为切片变量, 只需声明元素类型. 声明时它还没有指向数组, 也没有长度. 所以需要 s = a[1:3] 这样赋值, 指向的数组是从 a[1] 开始的, 到 a[2] 结束, 有两个元素, 在数学里应该写为 a[1:3), 即包括开始位置, 不包括结束位置的元素序列.

而切片变量也使用类型数组变量的语法读取元素, 也是从第 0 个位置开始. 这样 s[0] 实际是对应的是 a[1].

实际应用时, 切片比数组变量更常见. 在 Go 中 make 函数可以"制造"一个给定长度的数组并返回一个切片变量指向它. 就像在 base58_test.go 中:

```
dst := make([]byte, 100)
```

这样, 切片变量 dst 的长度 (也就是元素个数) 和被 make 分配的数组 (没有对应的数组变量) 长度相等. 之后, 我们随时可以重新切片, 例如 dst = dst[len(dst) - n : ] 只拿最后的 n 个元素. 注意如果省略掉冒号前后的下标, 则分别表示第0个元素开始和直到最后一个元素.

切片还可以在它指向的数组内重新切片. 因此, 切片不但记录了自己的长度, 还记录了底层数组的长度(称为容量 capacity), 分别由函数 len 和 cap 得到. 例如:

```
1  package main
2  import "fmt"
3
4  func main(){
5      s := []byte{1, 2, 3, 4}
6      s = s[:1]
7      fmt.Println(s, len(s), cap(s))
8      s = s[1:cap(s)]
9      fmt.Println(s, len(s), cap(s))
10 }
```

类似数组变量, 切片变量也可以在声明时初始化. 此时, s 的长度和容量都是 4. 而 s[:1] 之后长度为 1 而容量未变. 但 s[1:cap(s)] 之后, s 指向的数组的初始位置变了, 它的容量也跟着小了. 而且, 初始位置再也不能回到起点了. 即, 容量不可以扩大.

函数make 可以同时给出切片的长度和容量, 例如, s = make([]byte, 5, 10). 如果你觉得奇怪, 为什么同一个 make 函数可以带 2 个或 3 个参数? 栈如何预留参数的内存? 加分. 会问问题才不会厌倦读书. 我们 godoc fmt Println 看看 fmt.Println 的函数声明:

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

函数的最后一个参量, 可以是 … 代表的不限定个数的参量, 实际上是作为一个切片参量传递给函数的. 再例如下面的 Send 函数可以发送一条信息给多个人:

```
func Send(what string, to ...string)
```

函数 Send 的参量 to 是 []string 类型的切片变量. 当我们 Send(“Hello”, “Alice”, “Bob”, “Carol”) 时, 参量 to 指向的数组是 [3]string {“Alice”, “Bob”, “Carol”}. 如果我们有:

```
s := []string{"Dog", "Elephant", "Frog"}
Send("Bye", s...)
```

则在 Send 执行时, 其帧中参量 to 的位置, 指向的数组, 就是 s 指向的数组.

另外, 注意不定长参量的声明和使用, 分别在前和后加上 … 还有, make, len, cap 等是 Go 的内部函数, 不需要包名 (恭喜你如果你也一直有这个疑问).