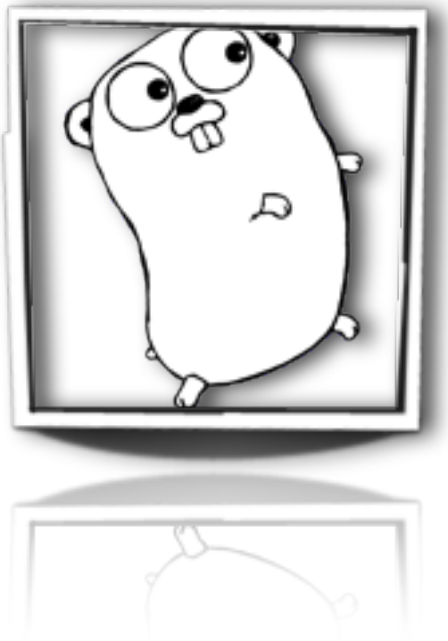
Fan Go



够粉丝

译者: Fango (fan.howard@gmail.com)

更新：http://code.google.com/p/ac-me/

日期：11/4/10

目录

够辅导 A Tutorial

原文：http://golang.org/doc/go_tutorial.html

介绍

本辅导介绍 Go 编程语言的基本知识，面向熟悉 C 或 C++ 的程序员。它不是语言的全面指南；现阶段最接近的是《语言规范》。读过此辅导后，你可以看看《够有效》，它会深挖语言的使用。另外，3 天 Go 课程的幻灯片可以在 doc/ Day 1, Day 2, Day 3 找到。

此处展示一系列适度的程序来显现语言的关键特色。所有程序都可以用（写作时），并收藏在目录 /doc/progs/ 下。

程序片断用原文件的行号标记；清楚起见，空行仍为空。

世界你好

我们依惯例开始：

```
05    package main

07    import fmt "fmt"  // Package implementing formatted I/
O.

09    func main() {
```

```
10          fmt.Printf("Hello, world; or Καλημέρα κόσμε; or こんにちは 世界\n")
11      }
```

每个 Go 的源文件使用 package 语句来声明它所属的包。它也可能导入其它的包来使用其提供的工具。此程序导入 fmt 包来取得我们的老朋友、但现在大写和包限定的，fmt.Printf 。

func 键字引入函数。 main 包的 main 函数是程序开始运行之处（初始化之后）。

字串常量可包含 Unicode 字符，UTF-8 编码。（实际上，Go 源文件规定使用 UTF-8 编码）

注解方式和 C++ 相同：

```
/* ... */
// ...
```

稍后我们会细说打印。

分号

你可能注意到我们的程序没有分号。Go 代码中，出现分号的典型地方是在 for 循环的分隔语句或类似之处。没有必要在每个语句后加分号。

实际上，正式语言使用分号，和 C 或 Java 类似，但它们自动添加在任何看起来像语句结尾的地方。你不需要自己敲入。

具体怎么做的可以看语言规范，但实际上你所要知道的只是，你从不需要在行尾放个分号。（你可以放，如果你在一行放多条语句。）再多帮你一把，你也可以不在结束大括号前放分号。

此方式产生干净的、无分号的代码。吃惊一下，记住你必须把起始大括号和语句放在同一行，比如它在 if 语句里要和 if 在同一行。如果你没做，有些情况下可能不可编译或可能产生错误结果。某种程度上语言强制了大括号的风格。

编译

Go 是编译语言。现存有两种编译器。Gccgo 是使用 GCC 后台的 Go 编译器。 还有一套编译器在每个体系上使用不同（也奇怪）的名称：6g 对64位的x86，8g 对32位的x86，及其它。这套编译器的速度明显的比较快，但生成的代码效率不如 gccgo 。 在此文写作时（2009后期），它们的运行系统也更强健，尽管 gccgo 在追上。

下面是怎样编译运行我们的程序，以 6g 为例：

```
    $ 6g helloworld.go  # compile; object goes into
helloworld.6
    $ 6l helloworld.6   # link; output goes into 6.out
    $ 6.out
    Hello, world; or Καλημέρα κόσμε; or こんにちは 世界
    $
```

使用 gccgo 看起来更传统一些。

```
    $ gccgo helloworld.go
    $ a.out
    Hello, world; or Καλημέρα κόσμε; or こんにちは 世界
    $
```

Echo

接下来，是一个 Unix 工具 echo(1):

```
05    package main

07    import (
08        "os"
09        "flag"  // command line option parser
10    )

12    var omitNewline = flag.Bool("n", false, "don't print
final newline")

14    const (
15        Space = " "
16        Newline = "\n"
17    )

19    func main() {
20        flag.Parse()   // Scans the arg list and sets up
flags
21        var s string = ""
22        for i := 0; i < flag.NArg(); i++ {
23            if i > 0 {
24                s += Space
25            }
26            s += flag.Arg(i)
27        }
28        if !*omitNewline {
29            s += Newline
30        }
31        os.Stdout.WriteString(s)
32    }
```

此程序虽小但做了一些新的事情。上例中，我们看到 func 引入一个函数。 键字 var，const，和 type （还未使用）也引入声明，还有 import 。注意我们可以把同类的声明编组为小括号括起的列，每项一行，正如行 7-10 和 14-17 。但不是必须这样做；我们可以写：

```
const Space = " "
const Newline = "\n"
```

此程序导入 os 包以使用 Stdout 变量，其类型为 *os.File 。import 语句实际上是个声明：其通常的格式，如我们的‘世界你好’所用到的，它命名标识（fmt），用来从导入文件（“fmt”）的包里取得成员，其文件在当前目录或某个标准位置。此程序中，我们舍弃了导入的明确名称；默认的，包以其导入包的名称命名，通常就是文件名本身 。我们的“世界你好”程序可以仅仅 import “fmt” .

你可以指定你自己的导入名，但只有在解决撞名时才必要。

给出 os.Stdout 我们就可以用它的 WriteString 方法打印字串。

导入 flag 包后，行12生成一个全局变量来持有 echo 的 -n 旗标的值。变量 omitNewLing 类型 *bool，是 bool 的指针。

main.main 里，我们分析参量（行20）并生成一个局部字串变量用来制作输出。

声明语句有如下格式：

```
var s string = ""
```

即，var 键字，后跟变量名，后跟其类型，后跟等号及其初始值。

Go 试图扼要，所以此声明也可缩短。因为字串常量的类型就是字串，我们不需知会编译器。我们可以写：

```
var s = ""
```

我们也可以更短些，写成习语：

```
s := ""
```

:= 运算符在 Go 里大量使用，表示初始化的声明。下行的 for 分句中有一例：

```
22          for i := 0; i < flag.NArg(); i++ {
```

flag 包已经分析了参量，剩下未被标识的参量放在一列，可以用显而易见的方式遍历。

Go 的 for 语句和 C 的有一些不同。首先，它是唯一的循环结构；这里没有 while 和 do 。其次，分句没有小括号，但循环体的大括号是必须的。同理适用于 if 和 switch 语句。稍后的例子给出 for 的一些其它写法。

循环体通过添加（用 +=）参量和区隔空格来积累字串 s 。循环后，如果 n 旗标未置，程序添加一新行。最后打印结果。

注意 main.main 是个零维的、无返回值的函数。它被规定成这样。从 main.main 尾部掉出代表“成功”；如果你要表示一个错误返回，调用：

```
os.Exit(1)
```

os 包里有其它启动时的必需品。例如，os.Args 是 flag 包用来取得命令行参量的切片。

类型插曲

Go 有些熟悉的类型如 int 和 float，代表机器‘适当’尺寸的值。它还定义了明确尺寸的类型，如 int8 和 float64 等等，另加无符号整型，如 uint 和 uint32 等。它们是完全不同的类型，即便是 int 和 int32 的尺寸都是 32位，它们也不是同一类型。byte 是 uint8 的同义词，为字串的单元类型。

提到字串，它也是个内部类型。字串是不可变值——它们不仅仅是 byte 数组类型。只要你生成了一个字串值，你就不可以改动它， 尽管你可以改动字串变量——只需重新赋值。下例的 strings.go 片段是合法的：

```
11          s := "hello"
12          if s[1] != 'e' { os.Exit(1) }
13          s = "good bye"
14          var p *string = &s
15          *p = "ciao"
```

但下面的语句是非法的，因为它们更动了 string 值：

```
s[0] = 'x'
(*p)[1] = 'y'
```

用 C++ 的术语，Go 的字串有点像 const strings，字串指针像 const string 引用。是的，它们是指针。但是，Go 做了些简化；接着读。

数组的声明如下：

```
var arrayOfInt [10]int
```

数组，类似字串，为值，但可变。这和 C 不同，那里 arrayOfInt 可以用来作 int 的指针。Go 里，因为数组是值，讲讲数组的指针是有意义的（也有用）。

数组的尺寸是类型的一部分；只是，我们可以声明切片变量，这样便可以赋值指针给任意同样单元类型的数组，或者更常见的，切片表达式格式 a[low:high] 代表一个下标从 low 到 high-1 的子数组。切片看着像极了数组但没有明确的尺寸（[] 对 [10]），它们引用一个底层的、通常无名的正常数组。多个切片可以共享数据，如果它们代表着同一数组；多个数组从来不能共享数据。

切片在 Go 程序里比正常数组更常见；它们更灵活，使用引用语义，也很高效。它们欠缺的是正常数组的对内存布局的精确控制；如果你要在你的结构中放 100 个项的数组，你应使用 正常数组。

传递数组给函数时，你几乎每次都会声明正规参量为切片。调用函数时，取数组的地址，Go 会高效的生成一个切片引用传过去。

使用切片我们可以写下面的函数（来自 sum.go）：

```
09    func sum(a []int) int {   // returns an int
10        s := 0
11        for i := 0; i < len(a); i++ {
12            s += a[i]
13        }
14        return s
15    }
```

然后这样调用：

```
19        s := sum(&[3]int{1,2,3})  // 一个数组的切片传递给 sum
```

注意返回类型（int）怎样在 sum() 的参量后定义。表达式 [3]int{1,2,3} —— 一个由大括号括起的表达式类型 —— 是值的创建者，此处为 3 个 int 的数组。放个 & 在前面使我们拿到此值的一个独特实例的地址。我们（隐式的）提升它为切片传递给 sum() 。

如果你要生成一个正常数组但要编译器替你数出项数，使用 ... 作为数组的尺寸：

```
s := sum(&[...]int{1,2,3})
```

实际上，除非你对某个数据结构的内存布局很在意，你只需使用空方括号且没有 & 的切片：

```
s := sum([]int{1,2,3})
```

还有映射，你可以这样初始化：

```
m := map[string]int{"one":1 , "two":2}
```

内部函数 len() 返回项数，首先出现在 sum 里，可用于字串、数组、切片、映射、和信道。

顺带一提，另一个可用在字串、数组、切片、映射、和信道的是 for 循环的 range 分句。除了写成：

```
for i := 0; i < len(a); i++ { ... }
```

遍历切片（或映射或...) 的项，我们可以写：

```
for i, v := range a { ... }
```

它赋值 i 给下标，v 是目标范围的每个项的值。 参见《够有效》里的使用例程。

分配插曲

Go 的大部分类型为值。如果你有一个 int 或 struct 或数组，赋值拷贝物件的内容。要分配一个新变量，使用 new()，它会返回分配的内存的指针。

```
type T struct { a, b int }
var t *T = new(T)
```

或更地道的：

```
t := new(T)
```

一些类型 —— 映射、切片、和信道（见下）—— 使用引用语义。如果你持有一个切片或映射并改动其内容，其它引用同样底层数据的变量会看到改动。对此三种类型你会使用内部函数 make()：

```
m := make(map[string]int)
```

此语句初始化一个新映射，已可以存放条目里。如果你只是声明映射，例如：

```
var m map[string]int
```

它只生成了 nil 引用，不能存放任何东西。要使用映射，你必须先要用 make() 初始化其引用，或从现有的映射赋值。

注意 new(T) 返回类型 *T，而 make(T) 返回类型 T 。如果你（错误地）用 new() 分配一个引用物件，你收到的是一个 nil 引用指针，等同于声明一个未初始化的变量并取其地址。

常量插曲

尽管整数在 Go 里有很多尺寸，整型常量不是。没有类似 0LL 或 0x0UL 的常量。相反，整型常量做为一个很高精度的值，只有在赋值给一个精度太低不能代表其值的整型变量时才溢出。

```
const hardEight = (1 << 100) >> 97  // 合法
```

一些细微差别值得参考《语言规范》的法律措辞，但下面是一些说明性的例子：

```
    var a uint64 = 0  // a has type uint64, value 0
    a := uint64(0)    // equivalent; uses a "conversion"
    i := 0x1234       // i gets default type: int
    var j int = 1e6   // legal - 1000000 is representable in
an int
    x := 1.5          // a float
    i3div2 := 3/2     // integer division - result is 1
    f3div2 := 3./2.   // floating point division - result is
1.5
```

转换只可用在简单的情况，例如某种符号或尺寸的 int 之间，或从 int 到 float，外加一些其它的简单情况。Go 里没有任何的自动数值转换，除了在变量赋值时使常量具有具体的尺寸。

I/O 包

接下来我们看一个简单的包，使用通常的 open/close/read/write 界面执行文件 I/O 。下面是 file.go 的开始部分：

```
05    package file

07    import (
08        "os"
09        "syscall"
10    )

12    type File struct {
13        fd      int    // file descriptor number
14        name    string // file name at Open time
```

```
15      }
```

前几行声明了包名 file，并导入两个包。os 包隐藏了不同操作系统的差别，提供一致的视角给文件等；此处我们使用它的错误处理工具并重新生成它的文件 I/O 的基本器官。

另一项是低层的、外部的 syscall 包，提供底层系统调用的原始界面。

接下来是个类型定义：type 键字引入一个类型的声明，此处是 File 的数据结构。为了更有趣一点，我们的 File 包括了文件描述符对应的文件名。

因为 File 以大写字母开头， 此类型在包外可用，亦即，可被此包的用户使用。Go 的信息能见规则很简单：如果顶层类型、函数、方法、常量或变量、结构域或方法的名称大写开头，则对包的用户可见。否则，名字及其命名的东西只在声明它们的包里可见。这不仅仅是个惯例；编译器强制此规则。Go 里，公共可见名的术语是“导出的”。

File 例中，每个域都是小写的因此不可被用户看到，但我们很快会导出一些大写的方法。

首先，一个生产 File 的工厂：

```
17      func newFile(fd int, name string) *File {
18          if fd < 0 {
19              return nil
20          }
21          return &File{fd, name}
22      }
```

它返回一个带文件描述符和文件名的 File 结构。此代码使用了 Go 里类似生成映射和数组时所用的 “组合字面” 的概念来架构一个堆分配的新物件。我们可以写：

```
    n := new(File)
    n.fd = fd
    n.name = name
    return n
```

但对于类似 File 的简单结构返回新组合字面的地址更容易些，比如在第21行。我们可以用工厂建设一些熟悉的 *File 导出类型变量。

```
24    var (
25        Stdin  = newFile(0, "/dev/stdin")
26        Stdout = newFile(1, "/dev/stdout")
27        Stderr = newFile(2, "/dev/stderr")
28    )
```

newFile 函数未被导出，因为它是内部的。合适的导出工厂是 Open：

```
30    func Open(name string, mode int, perm int) (file
*File, err os.Error) {
31        r, e := syscall.Open(name, mode, perm)
32        if e != 0 {
33            err = os.Errno(e)
34        }
35        return newFile(r, name), err
36    }
```

这几行有些新东西。首先，Open 返回多个值，一个 File 和一个错误（稍后细谈错误）。我们声明多值返回为一个小括号的声明列；语法

上看起来和第二个参数列一样。函数 syscall.Open 也有多值返回，在31行我们拿到声明的多个值；它声明 r 和 e 持有两个值，都是整型（尽管你要看 syscall 包的里面才知道）。最后，行 35 返回两个值：新 File 的指针和错误。如果 syscall.Open 失败，文件描述符 r 会是负数，newFile 返回 nil 。

至于这些错误：os 库包括一个广义的错误。 正如我们在做的 ，用它来帮助你的界面和整个 Go 代码的错误处理保持一致是个好主意。在 Open 里我们把 Unix 的整数 errno 转换为整型的 os.Errno， 它实现了 os.Error 。

现在我们可以制造 Files 了，我们可以为它们写些方法。要声明某类型的方法，我们定义一个函数，并把此类型的明确的接受者放在函数名前的小括号里。下面是 *File 的一些方法，每个都声明了接受者变量 file 。

```
38    func (file *File) Close() os.Error {
39        if file == nil {
40            return os.EINVAL
41        }
42        e := syscall.Close(file.fd)
43        file.fd = -1  // so it can't be closed again
44        if e != 0 {
45            return os.Errno(e)
46        }
47        return nil
48    }

50    func (file *File) Read(b []byte) (ret int, err
os.Error) {
51        if file == nil {
52            return -1, os.EINVAL
53        }
```

```
54          r, e := syscall.Read(file.fd, b)
55          if e != 0 {
56              err = os.Errno(e)
57          }
58          return int(r), err
59      }

61      func (file *File) Write(b []byte) (ret int, err
os.Error) {
62          if file == nil {
63              return -1, os.EINVAL
64          }
65          r, e := syscall.Write(file.fd, b)
66          if e != 0 {
67              err = os.Errno(e)
68          }
69          return int(r), err
70      }

72      func (file *File) String() string {
73          return file.name
74      }
```

这里没有隐含的 this ，必须使用接受者变量来访问结构的成员。方法不是声明在 struct 声明本身里。struct 声明只定义数据成员。实际上，不仅是 struct，几乎任何你命名的类型都可以生成方法，例如一个整型或数组。稍后我们会看到一个数组的例子。

String 方法的名称来自我们将要介绍的打印协议。

方法们使用了公共变量 os.EINVAL 来返回 （os.Error版本的）Unix 错误代码 EINVAL 。os 库定义了一套标准的错误值。

我们现在可以使用这个新的包了：

```
05    package main

07    import (
08        "./file"
09        "fmt"
10        "os"
11    )

13    func main() {
14        hello := []byte("hello, world\n")
15        file.Stdout.Write(hello)
16        file, err := file.Open("/does/not/exist",  0,  0)
17        if file == nil {
18            fmt.Printf("can't open file; err=%s\n",
err.String())
19            os.Exit(1)
20        }
21    }
```

“./file” 导入里的 “./”告诉编译器使用我们自己的包，而不是安装目录下的某个包。（另外，“file.go”必须先行编译我们才可以导入。）

我们现在编译执行此程序：

```
    $ 6g file.go                        # compile file pack-
age
    $ 6g helloworld3.go                 # compile main pack-
age
    $ 6l -o helloworld3 helloworld3.6   # link - no need to
mention "file"
    $ helloworld3
    hello, world
    can't open file; err=No such file or directory
    $
```

烂猫 Rotting cats

建筑在 file 包上，此处是 Unix 工具 cat(1) 的一个简单版本，progs/cat.go：

```
05    package main

07    import (
08        "./file"
09        "flag"
10        "fmt"
11        "os"
12    )

14    func cat(f *file.File) {
15        const NBUF = 512
16        var buf [NBUF]byte
17        for {
18            switch nr, er := f.Read(&buf); true {
19            case nr < 0:
20                fmt.Fprintf(os.Stderr, "cat: error reading
from %s: %s\n", f.String(), er.String())
21                os.Exit(1)
22            case nr == 0:  // EOF
23                return
24            case nr > 0:
25                if nw, ew := file.Stdout.Write(buf[0:nr]);
nw != nr {
26                    fmt.Fprintf(os.Stderr, "cat: error
writing from %s: %s\n", f.String(), ew.String())
27                }
28            }
29        }
30    }

32    func main() {
33        flag.Parse()   // Scans the arg list and sets up
flags
```

```
34          if flag.NArg() == 0 {
35              cat(file.Stdin)
36          }
37          for i := 0; i < flag.NArg(); i++ {
38              f, err := file.Open(flag.Arg(i), 0, 0)
39              if f == nil {
40                  fmt.Fprintf(os.Stderr, "cat: can't open
%s: error %s\n", flag.Arg(i), err)
41                  os.Exit(1)
42              }
43              cat(f)
44              f.Close()
45          }
46      }
```

到现在这应当容易读了，但 switch 语句引入了一些新特色。类似 for 循环，if 和 switch 可以包括一个初始化语句。18行的 switch 使用它生成了变量 nr 和 er，来持有 f.Read() 的返回值。（25行的 if 使用了同一概念）。switch 语句是通用的：它从上至下评估每个条件寻找第一个匹配值；条件表达式不必是常量或整数，只要它们都是同样类型即可。

因为 switch 的值仅是个 true，我们可以不放它，和 for 语句的情况一致，省略的值代表 true 。实际上，这样的 switch 是 if-else 链的一种形式。讲到这里，要提一下 switch 语句的每个 case 都隐含一个 break 。

行 25 调用 Write() 来切片传入的缓冲，其本身也是个切片。切片提供了 Go 的标准 I/O 缓冲处理方式。

我们现在做一个 cat 的变种，可选的 rot13 输入。当然可以很简单的直接处理字节，但我们利用一下 Go 界面的概念。

cat() 子程序只使用了两个方法：f.Read() 和 String()，因此我们从定义只有这两个方法的一个界面开始。下面是 progs/cat_rot13.go 里的代码：

```
26    type reader interface {
27        Read(b []byte) (ret int, err os.Error)
28        String() string
29    }
```

任何有 reader 这两个方法的类型 —— 不论此类型还包括多少其它的方法 —— 都称为实现了此界面。file.File 实现了这两个方法，所以它实现了 reader 界面。我们可以微调 cat 子程序接受 reader 而不是 *file.File，它一样会正常工作，但首先我们写第二个实现 reader 的类型来润色一下，它包装现存的 reader，用 rot13 处理数据。我们只需定义类型，实现方法，而无需其他的账目管理，我们有了第二个 reader 界面的实现：

```
31    type rotate13 struct {
32        source    reader
33    }

35    func newRotate13(source reader) *rotate13 {
36        return &rotate13{source}
37    }

39    func (r13 *rotate13) Read(b []byte) (ret int, err
os.Error) {
40        r, e := r13.source.Read(b)
41        for i := 0; i < r; i++ {
```

```
42              b[i] = rot13(b[i])
43          }
44          return r, e
45      }

47      func (r13 *rotate13) String() string {
48          return r13.source.String()
49      }
50      // end of rotate13 implementation
```

（42行的 rot13 函数不重要，不值得写在这里。）

为使用新的功能，我们定义一个旗标:

```
14      var rot13Flag = flag.Bool("rot13", false, "rot13 the
input")
```

并在几乎未变的 cat() 函数中使用它:

```
52      func cat(r reader) {
53          const NBUF = 512
54          var buf [NBUF]byte

56          if *rot13Flag {
57              r = newRotate13(r)
58          }
59          for {
60              switch nr, er := r.Read(&buf); {
61              case nr < 0:
62                  fmt.Fprintf(os.Stderr, "cat: error reading
from %s: %s\n", r.String(), er.String())
63                  os.Exit(1)
64              case nr == 0:  // EOF
65                  return
66              case nr > 0:
67                  nw, ew := file.Stdout.Write(buf[0:nr])
68                  if nw != nr {
```

```
69                      fmt.Fprintf(os.Stderr, "cat: error
writing from %s: %s\n", r.String(), ew.String())
70                  }
71              }
72          }
73      }
```

（我们也可以包装 main 而几乎把 cat() 放在一边，只是换它参量的类型；考虑以此为练习。）行 56 到 58 安排好它们：如果 rot13 旗标为真，把我们收到的 reader 包装为 rotate13 并继续。注意界面变量是值，不是指针：参量是 reader 类型，不是 *reader，尽管盖子下面它持有一个 struct 的指针。

下面是它的结果：

```
$ echo abcdefghijklmnopqrstuvwxyz | ./cat
abcdefghijklmnopqrstuvwxyz
$ echo abcdefghijklmnopqrstuvwxyz | ./cat --rot13
nopqrstuvwxyzabcdefghijklm
$
```

依赖注射迷们（Fans of depenency injection）可以欢呼界面使我们替换一个文件描述符的实现变得多么容易。

界面是 Go 最出色的功能之一。如果一个类型实现了一个界面声明的所有方法，此类型即实现了此界面。这意味着一个类型可以实现任意数量的不同界面。这里没有类型层次，事情可以更加随意，正如我们在 rot13 里看到的。类型 file.File 实现了 reader；它也可以实现 writer，或任何从其适合当前情况的方法构造的界面。考虑空界面：

```
type Empty interface {}
```

每个类型都实现空界面，这对容器等应用很有用。

排序

界面提供了一种多态性的简单形式。它们使某物件的定义和实现彻底分离，使得同一界面变量可以在不同时间代表完全不同的实现。

作为一个例子，考虑 progs/sort.go 里的一个简单的排序算法：

```
13    func Sort(data Interface) {
14        for i := 1; i < data.Len(); i++ {
15            for j := i; j > 0 && data.Less(j, j-1); j-- {
16                data.Swap(j, j-1)
17            }
18        }
19    }
```

此代码只需三个方法，我们用 sort 的 Interface 包装它们：

```
07    type Interface interface {
08        Len() int
09        Less(i, j int) bool
10        Swap(i, j int)
11    }
```

我们可以应用 Sort 到任何实现了 Len，Less 和 Swap 的类型。 sort 包里有一些必要的方法来给数组、字串等排序。这里是给 int 数组的代码：

```
33    type IntArray []int

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

这里我们看到对非 struct 类型的方法。 你可以给你的包里定义和命名的任意类型定义方法。

现在是测试例程，来自 progs/sortmain.go 。它使用 sort 包里的一个函数，此外省略，来测试结果是排序好的。

```
12    func ints() {
13        data := []int{74, 59, 238, -784, 9845, 959, 905,
0, 0, 42, 7586, -5467984, 7586}
14        a := sort.IntArray(data)
15        sort.Sort(a)
16        if !sort.IsSorted(a) {
17            panic("fail")
18        }
19    }
```

如果我们有个新的类型要排序，所要做的只是实现这三个方法在那个类型上，例如:

```
30    type day struct {
31        num        int
32        shortName  string
33        longName   string
34    }
```

```
36    type dayArray struct {
37        data []*day
38    }

40    func (p *dayArray) Len() int            { return
len(p.data) }
41    func (p *dayArray) Less(i, j int) bool  { return
p.data[i].num < p.data[j].num }
42    func (p *dayArray) Swap(i, j int)       { p.data[i],
p.data[j] = p.data[j], p.data[i] }
```

打印

目前为止的排版打印都点到为止。此节我们谈谈如何把 Go 的排版 I/O 做到够好。

我们已看到了 fmt 包的简单使用，Printf，Fprintf 实现等等。在 fmt 包里， Printf 使用如下签名声明：

```
Printf(format string, v ...interface{}) (n int, errno
os.Error)
```

符号 ... 引入一个变长参量列，在 C 里会用 stdarg.h 的宏处理。Go 里，多维函数传入一个指定类型的参量切片。 Printf 的例子，声明是 ...interface {} 所以实际类型是一个空界面值的切片 []interafce{} 。Printf 会检查参量，遍历切片，并对每一项使用类型切换，或反思包来解释每个值。有些跑题但这样的运行态分析可以解释 Go Printf 的一些特色，归功于 Printf 能动态的发现其参量的类型。

例如，C 的每个格式必须对应其参量类型。Go 里大部分情况都容易些。对于 %llud 你只需 %d；Printf 知道整数的正负和尺寸并相应处理。下面的片段：

```
10          var u64 uint64 = 1<<64-1
11          fmt.Printf("%d %d\n", u64, int64(u64))
```

打印

```
18446744073709551615 -1
```

实际上，你可偷点懒，%v 会用适当的格式打印任意的值、数组或结构：

```
14          type T struct {
15              a int
16              b string
17          }
18          t := T{77, "Sunset Strip"}
19          a := []int{1, 2, 3, 4}
20          fmt.Printf("%v %v %v\n", u64, t, a)
```

输出是：

```
18446744073709551615 {77 Sunset Strip} [1 2 3 4]
```

如果你用 Print 或 Println 而不是 Printf，你可以完全省略排版。这些例程自动排版。Print 函数只是使用对应的 %v 打印每一项，而 Println

在参量间加空格并加上新行。 下面两行的输出和上面 Printf 的完全一样。

```
21          fmt.Print(u64, " ", t, " ", a, "\n")
22          fmt.Println(u64, t, a)
```

如果你自己的类型要像 Printf 或 Print 一样排版，只需让 String() 方法返回一个字串。 print 例程会检查值看它是否实现了此方法，如是，则用它而不用其它的排版。例如：

```
09      type testType struct {
10          a int
11          b string
12      }

14      func (t *testType) String() string {
15          return fmt.Sprint(t.a) + " " + t.b
16      }

18      func main() {
19          t := &testType{77, "Sunset Strip"}
20          fmt.Println(t)
21      }
```

因为 *testType 有 String() 方法，此类型默认的排版器会用来产生如下输出：

```
77 Sunset Strip
```

留意 String() 方法调用 Sprint（Go 的变种，返回字串）完成排版；特殊的排版器能递归的使用 fmt 库。

Printf 的另一特色是格式 %T 可以打印代表某值类型的字串，在多态代码排漏时会有用。

写个带旗标和精度的充分定制的打印格式完全可能，但那有点离题，所以留作练习吧。

你可能会问，Printf 怎样知道某个类型实现了 String() 方法？它所做的实际上是看一个值是否可以转换为实现此方法的界面变量。大略上是说，给定一个值 v，它的操作是：

```
    type Stringer interface {
        String() string
    }

    s, ok := v.(Stringer)  // Test whether v implements
"String()"
    if ok {
        result = s.String()
    } else {
        result = defaultOutput(v)
    }
```

此代码使用了“类型断言”（v.(Stringer)）来测试 v 所存的值是否满足 Stringer 界面。如是，则 s 成为实现此方法的界面变量，而 ok 为真。然后我们用此界面变量调用方法。（“逗号，ok”的模式是 Go 的惯用语，来测试类型转换、映射更新、通信等操作是否成功，尽管此处是本辅导的唯一的出现。）如果此值不满足界面，ok 为假。

此片段中名称 Stringer 延续了我们在描述简单方法的名后加 ‘[e]r’的传统。

最后一眼，为了此套的完整，除了 Print 等、Sprintf 等，还有 Fprintf 等。与 C 不同，Fprintf 的首参量不是 file，而是类型为 io.Writer 的变量，它实现了 io 库中定义的一个界面类型：

```
type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

（此界面是另一个习惯名称，这里是对 Writer；还可以是 io.Reader，io.ReadWriter等等）。

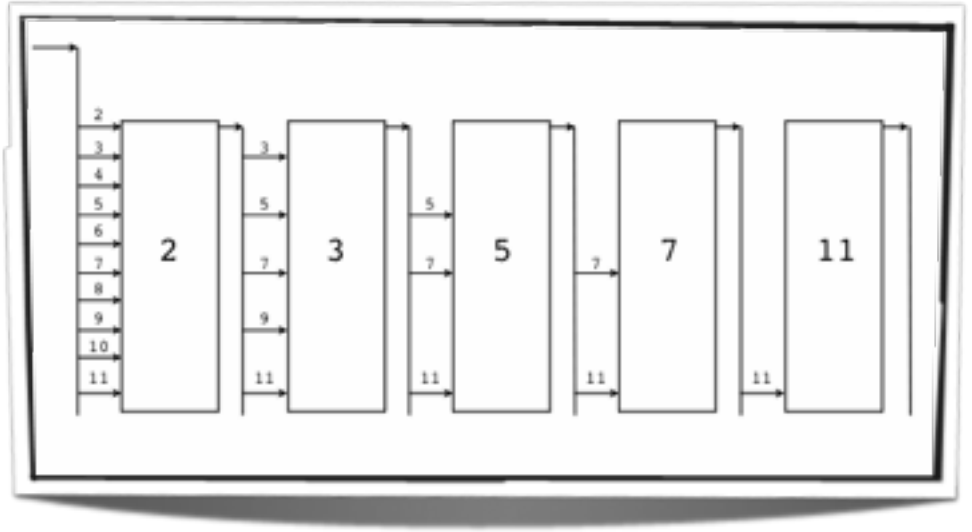
因此你可以调用 Fprintf 在任何实现了标准 Write() 方法的类型上，不仅仅是文件，也可是网络信道，缓冲，和你想要的任何东东。

素数

现在我们到了进程和通信 —— 并发编程。这是个很大的主题，简单起见我们假设你对此有所了解。

此风格的经典问题是个素数筛子。（Eratosthenes 埃氏筛子比此处的算法高效，但此时我们更在意并发而非算法。）它拿一串的自然数，加入一系列的筛选器，每个素数一个，来挑出此素数的倍数。在每一步我们都有目前为止的所有素数的筛选器，下一个跳出的是个素数，它会触发生成筛选器链中的下一个筛子。

流程如下；每个方框代表从前面方框流出的数所触发的筛选器。



为了产生一串整数，我们使用 Go 信道。此概念从 CSP 衍生，代表连接两个并发计算的通信信道。Go 里，信道变量是一个协调通信的运行态物件的引用；类似映射和切片，要使用 make 生成新信道。

下面是 progs/sieve.go 的第一个函数：

```
09    // Send the sequence 2, 3, 4, ... to channel 'ch'.
10    func generate(ch chan int) {
11        for i := 2; ; i++ {
12            ch <- i  // Send 'i' to channel 'ch'.
13        }
14    }
```

generate 函数发送系列 2，3，4，5 ... 到参量信道 ch，使用二元通信操作符 <- 。信道操作会阻塞，因此如果没有人接收 ch 的值，发送操作会等待，直到有人接收。

filter 函数有三个参量：输入信道，输出信道，和一个素数。它把输入的值拷贝到输出，丢弃所有可被此素数除的值。一元通信操作符 <- （接收）从信道上取得下一个值。

```
16    // Copy the values from channel 'in' to channel 'out',
17    // removing those divisible by 'prime'.
18    func filter(in, out chan int, prime int) {
19        for {
20            i := <-in  // Receive value of new variable
'i' from 'in'.
21            if i % prime != 0 {
22                out <- i  // Send 'i' to channel 'out'.
23            }
24        }
25    }
```

generator 和 filter 并发执行。 Go 有自身的进程/线程/轻量进程/协程的模型，为避免语义上的混淆我们将 Go 中并发运行的计算称为*够程*。要启动一个够程，在函数调用前加个键字 go；这样启动的函数和当前的计算并行，但处于同一地址空间：

```
go sum(hugeArray) // calculate sum in the background
```

如果你要获知计算的结束，传入一个信道以便让它报告回来：

```
    ch := make(chan int)
    go sum(hugeArray, ch)
    // ... do something else for a while
    result := <-ch  // wait for, and retrieve, result
```

回到素数筛子。下面是筛子流水线如何缝在一起：

```
28    func main() {
29        ch := make(chan int)  // Create a new channel.
30        go generate(ch)  // Start generate() as a gorou-
tine.
31        for {
32            prime := <-ch
33            fmt.Println(prime)
34            ch1 := make(chan int)
35            go filter(ch, ch1, prime)
36            ch = ch1
37        }
38    }
```

行 29 生成初始信道传递给 generate，然后启动。随着每个素数从信道里跳出来，一个新的 filter 加在流水线后，并且它的输出成为 ch 的新值。

该筛选程序可以微调为这种编程风格的常见模式。下面是 progs/sieve1.go 里的 generate 的变种：

```
10    func generate() chan int {
11        ch := make(chan int)
12        go func(){
13            for i := 2; ; i++ {
14                ch <- i
15            }
16        }()
17        return ch
18    }
```

此版本在内部完成全部的安排，生成输出信道，启动够程运行一个函数字面，并返回调用者此信道。它是个并发执行的工厂，启动够程，返回连接。

函数字面的写法（行 12-16）使我们能架构一个函数并当场执行。注意局部变量 ch 可用于函数字面，并在 generate 返回后依然存活。

同样的修改可用与 filter：

```
21    func filter(in chan int, prime int) chan int {
22        out := make(chan int)
23        go func() {
24            for {
25                if i := <-in; i % prime != 0 {
26                    out <- i
27                }
28            }
29        }()
30        return out
31    }
```

sieve 的主循环因此变得简单清楚，我们也把它变成了一个工厂：

```
33    func sieve() chan int {
34        out := make(chan int)
35        go func() {
36            ch := generate()
37            for {
38                prime := <-ch
39                out <- prime
40                ch = filter(ch, prime)
41            }
42        }()
43        return out
44    }
```

现在 main 对素数的界面为一个素数信道：

```
46    func main() {
```

```
47          primes := sieve()
48          for {
49              fmt.Println(<-primes)
50          }
51      }
```

多路复用

利用信道可以无需明确的复用器来服务多个独立的客户够程。诀窍是在消息中发送一个信道给服务器，用它来回复发送者。现实中的客户服务程序有大量代码，因此这里只用个小小的替代品来阐述概念。从定义 request 类型开始吧，它内置一个信道用来回复。

```
09      type request struct {
10          a, b    int
11          replyc  chan int
12      }
```

服务器微不足道：只是在整数上简单的二元操作。下面代码执行操作回应请求：

```
14      type binOp func(a, b int) int

16      func run(op binOp, req *request) {
17          reply := op(req.a, req.b)
18          req.replyc <- reply
19      }
```

行 14 定义的 binOp 是取两个整数并返回第三个数的函数。

server 例程永久循环，接收请求，并且为避免耗时操作造成的阻塞，启动一个够程完成实际工作。

```
21    func server(op binOp, service chan *request) {
22        for {
23            req := <-service
24            go run(op, req)  // don't wait for it
25        }
26    }
```

我们同样架构一个服务器，启动并返回一个连接的信道：

```
28    func startServer(op binOp) chan *request {
29        req := make(chan *request)
30        go server(op, req)
31        return req
32    }
```

下面是个简单测试。使用加法操作启动服务器，发送 N 个请求而不需等待回复。只当全部请求发送完毕才检查结果。

```
34    func main() {
35        adder := startServer(func(a, b int) int { return a
+ b })
36        const N = 100
37        var reqs [N]request
38        for i := 0; i < N; i++ {
39            req := &reqs[i]
40            req.a = i
41            req.b = i + N
42            req.replyc = make(chan int)
43            adder <- req
44        }
45        for i := N-1; i >= 0; i-- {   // doesn't matter
what order
```

```
46              if <-reqs[i].replyc != N + 2*i {
47                  fmt.Println("fail at", i)
48              }
49          }
50          fmt.Println("done")
51      }
```

此程序的一个烦人之处是它不能干净的关掉服务器；main 返回时仍有很多够程流连阻塞在信道上。为解决此问题，我们可以提供第二个叫 quit 的信道给服务器：

```
32      func startServer(op binOp) (service chan *request,
quit chan bool) {
33          service = make(chan *request)
34          quit = make(chan bool)
35          go server(op, service, quit)
36          return service, quit
37      }
```

它把 quit 信道传给 server 函数，用法如下：

```
21      func server(op binOp, service chan *request, quit chan
bool) {
22          for {
23              select {
24              case req := <-service:
25                  go run(op, req)  // don't wait for it
26              case <-quit:
27                  return
28              }
29          }
30      }
```

server 内部，select 语句选择其选项中列出的多个通信哪个可以继续。如果全部阻塞，它会等待直到某个可以继续，如有多个，它会任选其一。此例中，select 使服务器执行请求直到收到 quit 信息，然后退出，终止运行。

所剩的只是在 main 的最后点一下 quit 信道：

```
40          adder, quit := startServer(func(a, b int) int {
return a + b })
...

55          quit <- true
```

Go 编程和并发编程还有大量的知识，这趟快速旅游应该可以给你一些基本介绍了。

够有效 Effective Go

❖ 原文：http://golang.org/doc/effective_go.html

介绍

Go 是新语言。尽管一些观念是从现存的语言里汲取的，它有些不寻常的特性使得有效编写Go 程序不同于用类似的语言编写程序。从 C++ 或 Java 程序直接翻译到 Go 不大可能得到满意的结果 - Java程序是用Java编写的，不是 Go 的。另一方面，从 Go 的角度考虑问题可以得到满意的、但很不一样的程序。换句话说，要写得 Go 好，重要的是理解它的特性和习语。同样重要的是知道Go的编程惯例，例如命名，排版，程序结构等等。这样其他的 Go 程序员能更容易的明白你写的 Go 程序。

本文列出一些编写明晰地道 Go 代码的小建议，是《语言规范》和《辅导》的补充。你应先读完它们。

范例

Go 包的源码除了作为核心库外，还特意用于展示如何使用语言。如果你困惑如何动手处理某个问题，或某事要怎样实现，它们可以提供答案、主意和背景。

排版

排版问题最有争议也最无实质影响。人们能适应不同的排版风格，但无此必要更好，如每人遵循同样风格，则此主题可花费少些时间。问题在于如何到达此理想境界，又不必使用冗长的规范式的指南。

Go 另辟蹊径，让机器负责大部分排版问题。程序 gofmt 读取 Go 源码并用一套标准的缩进和纵向对齐的风格发表。注解得到保留并必要时重排。如果你想知道怎样处理新布局，运行gofmt。如果结果看起来不对，修正程序（或提交漏洞），不要绕过。

例如，没必要花费时间去对齐某结构中域的注解，gofmt 会给你做好。如下的声明：

```
type T struct {
    name string // name of the object
    value int // its value
}
```

gofmt 会对齐每列：

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

所有库里的代码都已经 gofmt 排版过。

剩下些排版细节，简单讲：

❖ 缩进：我们使用 tab，也是 gofmt 的默认值。只当必须时你才用空格。

❖ 行的长度：Go 没有行的长度限制。不用担心溢出打孔卡片。如果某行感觉太长，折下来并加一 tab 缩进。

❖小括号： Go 很少需要小括号：控制结构（ if, for, switch ）的句法不需小括号。还有，运算优先层次更短更清楚，所以 x<<8 + y<<16 文如其意。

注解

Go 提供 C 式 /* */ 块注解和 C++ 式 // 行注解。通常用行注解；块注解多用于包的注释，也可用于使大片代码失去作用。

程序 - 也是网页服务器 - godoc 处理 Go 的源代码，从中提取包的文档。顶层声明前的注解，如无空行相隔，和声明一起提取作为条目的解释文字。这些注解的性质和风格决定着 godoc 产生的文档的质量。

每个包都应有一个包注解，即 package 前的块注解。对多个文件的包，包注解只需出现在一个文件中，随便哪个。包注解应该介绍此包，并作为一个整体提供此包的对应信息。它首先出现在 godoc 页面，来安排好后续的详细文档。

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

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
```

```
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

包如果简单，包注解可以简短:

```
// The path package implements utility routines for
// manipulating slash-separated filename paths.
```

注解不需多余排版如星星横幅等。生成的结果呈现时可能不是等宽字体，所以不要靠空格对齐， godoc，类似 gofmt 照管这些。最后，注解是不加解释的文本，HTML和其他例如 _this_ 会原样照搬，所以应避免使用。

在包里，紧跟顶层声明前的注解作为此声明的文注解，程序中每个导出（大写）的名字都应该有文注解。

文注解最好是完整的句子。首句应该以声明的名字开始的一句话的总结:

```
// Compile parses a regular expression and returns, if suc-
cessful, a Regexp
// object that can be used to match against text.
func Compile(str string) (regexp *Regexp, error os.Error) {
```

Go 的声明句法允许编组。单一的文注解可以引出一组相联的常量或变量。因为整组声明一起展现，注解可以很粗略：

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal      = os.NewError("internal error")
    ErrUnmatchedLpar = os.NewError("unmatched '('")
    ErrUnmatchedRpar = os.NewError("unmatched ')'")
    ...
)
```

对于私有名称，编组也可以指出它们之间的联系，例如一系列的变量由一个互斥保护。

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

名称

名称在 Go 里和在其它语言里一样重要。某种情况下它们甚至有语义效果：例如，一个名称能否在包外可见取决于它的第一个字母是否大写。所以值得花点时间探讨下 Go 程序的命名约定：

包名

当包引入时，包名成为其内容的引导符。 import “bytes” 后， 导入者可以讲 bytes.Buffer。更有用的是每个包的用户都能使用相同的名称指出它的内容，亦即包应有个好名称：短，精，好记。习惯上包名是小写的单字的名称；应无必要用下划线或大小混写。简错不纠，因为你的包的每个用户都要敲这个名字。还有不要无谓烦扰撞名。包名只是引入时的默认名；它不需在所有源码中都唯一，如出现少见的撞名，导入者可以给出不同的名字局部使用。无论如何，撞名很少见，因为 import 用的文件名只决定使用那个包。

另一个习惯是包名是源目录的基名；src/pkg/container/vector 里的包引入为 “container/vector” 但包名是 vector，不是 container_vector 也不是 containerVector。

导入者使用包名引导其内容 （import . 的记法主要特意用在测试或其它不寻常的场合），所以包的导出的名称可据此避免结结巴巴。例如，bufio 包的 buffered reader 叫 Reader，不叫 BufReader，因为用户看到的是 bufio.Reader 这个清楚简短的名称。再有，因为导入项总是给出其包名，bufio.Reader 不会和 io.Reader 撞名。类似的，用来生成 ring.Ring 的函数 — 即 Go 的架构函数 — 通常会被称为 NewRing，但因为 Ring 是此包唯一的导出类型，并且既然包名叫 ring，它就叫 New。此包的客户看到的是 ring.New。使用包结构帮你来选个好名。

界面名

习惯上，单一成员的界面的名称是其成员名加 -er：Reader, Writer, Formatter 等。

存在这样的一些名称，尊重它们和它们所指的函数会工作的更好。Read, Write, Close, Flush, String 等保有正统的签名和意义。为了避免混淆， 除非有同样的签名和意义，不要给你的方法这些名字。同理，如果你的方法实现了和这些著名方法同样的意图，给它同样的名称和签名；叫你的字符转换器 String 而不是 ToString。

大小混写

最后，Go 习惯使用 MixedCaps 和 mixedCaps，而不是下划线来写多字的名称。

分号

类似 C，Go 的正式语法使用分号结束语句；不同于 C，这些分号不会出现在源码中。词法器在扫描时使用一套简单的规则自动添加分号，所以输入文本几乎不需要它们。

规则是：如果新行前的最后一个符号是个标识符（包括 int 和 float64 等词），或者是基本文字像数字或字串常量，或者如下符号：

```
break continue fallthrough return ++ -- ) }
```

则词法器总是在其后添加分号。这可总结为：“如果新行可结束某语句，加入分号。”

分号也可在结束大括号前省略，如下句：

```
go func() { for { dst <- <-src } }()
```

不需分号。地道的 Go 程序分号只出现在如 for 循环这样的地方，用来隔开初始值，条件和延续元素。它们也可用来分隔同一行的多条语句，如果你要那样写的话。

警示：永远不要把控制语句（ if, for, switch, select)的起始大括号另起一行。如果做了，分号会加在大括号前，进而产生不希望的结果。要这样写：

```
if i < f() {
    g()
}
```

不要这样写：

```
if i < f()  // 错!
{           // 错!
    g()
}
```

控制结构

Go 的控制结构类似 C 但有重要不同。这里没有 do 和 while 循环，只有略微泛化的 for，且 switch 更灵活，if 和 switch 接受类似 for 的可选初始语句，还有新的控制结构：类型切换和多路通信的分路器 select。句法也略有不同：不需小括号，但控制体必须用大括号。

if

Go 的简单 if 如此：

```
if x > 0 {
    return y
}
```

强制的大括号鼓励把简单 if 写成多行。这样的风格无论怎样都是好的，特别是体内含有 return 或 break 等控制语句时。

if 和 switch 接受初始语句，常用来安排局部变量：

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

在 Go 库里，你会发现当 if 语句不能走到下条语句，即 break，continue，goto 或 return 结束时，不必要的 else 被省略：

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

下例是代码必须分析一系列出错可能的典型情况。控制流成功下行，遇错排错，代码很容易读。因为错误情况常用 return 结束，使得代码不需 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
}
codeUsing(f, d)
```

for

Go 的 for 循环类似 C 的但有区别。它统一了 for 和 while 去掉了 do-while 。一共三种格式，只有一种带分号。

```
// 类似 C 的 for
for init; condition; post { }

// 类似 C 的 while
for condition { }

// 类似 C 的 for(;;)
for { }
```

短声明使得在循环里声明下标变量很容易。

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

如果你要遍历数组，切片，字串或映射，或读自信道，range 可帮你管理循环。

```
var m map[string]int
sum := 0
for _, value := range m {  // key is unused
    sum += value
}
```

对字串，range 做工更多，它会分析 UTF-8 拆解字串成独立的 Unicode 字符（错误编码会吃掉一个字节，并产生替换的 rune 字母 U+FFFD）。循环：

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

打印出：

```
character 大 starts at byte position 0
character 中 starts at byte position 3
character 国 starts at byte position 6
```

最后，因为 Go 没有逗号操作符且 ++ 和 -- 是语句而非表达式，如果你要 for 有多个变量，可用并行赋值：

```
// 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]
}
```

switch

Go 的 switch 比 C 的通用。它的表达式不必是常量或整数，每个分支从上至下求值，直到匹配，如果 switch 没有表达式则用 true 来 switch。因此可能，也很地道的，将 if-else-if-else 写为 switch：

```
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
    }
    return 0
}
```

没有自动掉到下一分支，但分支可以是逗号分隔的列表：

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

这里是字节数组的比较例程，使用了两个 switch：

```
// Compare returns an integer comparing the two byte arrays
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a
> b
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
        }
    }
    switch {
    case len(a) < len(b):
```

```
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0
}
```

switch 也用来发现界面变量的动态类型。这种 type switch 使用类型断言的句法，把 type 括起。如果 switch 在表达式里声明了变量，此变量在每个分支里会有对应的类型。

```
switch t := interfaceValue.(type) {
default:
    fmt.Printf("unexpected type %T", t)  // %T 打印类型
case bool:
    fmt.Printf("boolean %t\n", t)
case int:
    fmt.Printf("integer %d\n", t)
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t)
case *int:
    fmt.Printf("pointer to integer %d\n", *t)
}
```

函数

多值返回

Go 的一种不寻常的特性是函数和方法可以返回多值。这可用来改进 C 程序的某些笨拙的习语：带内错误返回（如 -1 作为 EOF）和变更参量。

C 里，write 错误由一个负值通知，其错误码藏在一个易变的位置。Go 里，Write 可以同时返回计数和错误：“ 是，你写了些字节但不是全部因为你填满了设备。” os 包的 *File.Write 的签名是：

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

正如文档所述，它返回写入的字节数，及当 n != len(b)时一个非空的 Error。这是常见的风格；更多例子参见错误处理一节。

类似的方法免除了用返回值指针模拟引用参数的必要。这里是个单纯的函数，从字节数组的某位置拿数，返回其值和下一位置。

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i])-'0'
    }
    return x, i
}
```

你可用其扫描一个输入数组，如下：

```
 for i := 0; i < len(a); {
     x, i = nextInt(a, i)
     fmt.Println(x)
 }
```

带名结果参量

Go 函数的返回或结果“参量”可以起名并用作普通变量，正如传入的参量。带名后，它们在函数开始时初始为对应类型的零值；如果函数执行的 return 不带参量，则结果参量的现有值作为返回值。

这些名称不是必须的，但它们使代码更短更清楚：它们是文档。如果我们命名 nextInt 的结果，哪个 int 对应哪个值会一清二楚。

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

因为带名结果被初始并绑为一个不加修饰的返回值，它们得到简化和净化。io.ReadFull 很好的使用了它们：

```
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
}
```

defer

Go 的 defer 语句安排一个函数调用（被 defer 的函数）延迟发生在执行 defer 的函数刚要返回之前。当函数无论怎样返回，某资源必须释放时，可用这种与众不同、但有效的处理方式。传统的例子包括解锁互斥或关闭文件。

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, os.Error) {
    f, err := os.Open(filename, os.O_RDONLY, 0)
```

```
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're fin-
ished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = bytes.Add(result, buf[0:n])
        if err != nil {
            if err == os.EOF {
                break
            }
            return "", err  // f will be closed if we return
here.
        }
    }
    return string(result), nil // f will be closed if we re-
turn here.
}
```

这样延迟一个函数有双重优势：一是你永远不会忘记关闭文件，此错误在你事后编辑函数添加一个返回路径时常常发生。二是关闭和打开靠在一起，比放在函数尾要清晰很多。

延迟函数的参量（包括接受者，如果函数是一个方法）的求值发生在 defer 语句执行时，而不是延迟函数调用时。除了不必担心函数执行时变量值的改变外，也意味着同一延迟调用可以延迟多个函数的执行。如下傻例：

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

延迟函数执行顺序为 LIFO，所以上面代码在函数返回时打印 4 3 2 1 0。更可信的例子是跟踪程序中函数执行的一个简单方式。我们可以写些简单的跟踪例程:

```
func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

// 如下使用:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

利用被延迟函数的参量在 defer 执行时得值的特点，跟踪函数可以安排未跟踪函数的参量。

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}
func un(s string) {
    fmt.Println("leaving:", s)
}
func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}
func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}
func main() {
    b()
}
```

打印：

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

对于习惯其它语言的块层次资源管理的程序员，defer 可能比较怪，但它最有趣最强力的应用恰恰来自它不是基于块、而是基于函数。在 panic 和 recover 一节我们会看到一个例子。

数据

new()分配

Go 有两个分配原语，new() 和 make() 。它们做法不同，也用作不同类型上。有点乱但规则简单。我们先谈谈 new() 。它是个内部函数，本质上和其它语言的同类一样：new(T)分配一块清零的存储空间给类型 T 的新项并返回其地址，一个类型 *T 的值。 用 Go 的术语，它返回一个类型 T 的新分配的零值。

因为 new() 返回的内存清零， 可以用来安排使用零值的物件而不需再初始化。亦即数据结构的用户可以直接用 new() 生成一个并马上使用。例如， bytes.Buffer 的文档指出“零值的 Buffer 为空并可用”。同

理，sync.Mutex 没有明确的架构函数或 init 方法。 而是，一个 sync.Mutex 的零值定义为开锁的互斥。

零值有用，这个特性可以顺延。考虑下面的声明。

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

类型 SyncBuffer 的值在分配或者声明后立即可用。下例，p 和 v 无需多余的安排已可以正确使用了。

```
p := new(SyncedBuffer)  // type *SyncedBuffer
var v SyncedBuffer      // type  SyncedBuffer
```

架构函数和组合字面

有时零值不够好，有必要使用一个初始化架构函数，如下面从 os 包引出的例子。

```
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
}
```

这里有很多注模。我们可用组合字面简化之，它是个每次求值即生成新实例的表达式。

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

注意返回局部变量的地址是完全 OK 的；变量对应的存储空间在函数返回后仍然存在。实际上，取一个组合字面的地址使每次它求值时都生成一个新实例，因此我们可以把最后两行合起来。

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

组合字面的域必须按顺序给出并全部出现。可是，明确的用域:值对儿标记元素，初始化可用任意顺序，未出现的对应着零值。所以我们可以讲

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

特别的，如果一个组合字面一个域也没有，它生成此类型的零值。表达式 new(File) 和 &File{} 是等价的。

组合字面也可以生成数组、切片和映射，其域为合适的下标或映射键。下例中，无论 Enone Eio 和 Einval 是什么值都可以，只要它们是不同的。

```
a := [...]string{Enone: "no error", Eio: "Eio", Einval: "in-
valid argument"}
s := []string   {Enone: "no error", Eio: "Eio", Einval: "in-
valid argument"}
m := map[int]string{Enone: "no error",Eio:"Eio",Einval: "in-
valid argument"}
```

make()分配

回到分配。内部函数 make(T, args) 的服务目的和 new(T) 不同。它只生成切片，映射和信道，并返回一个初始化的（不是零）的，type T 的，不是 *T 的值。这种区分的原因是，这三种类型，揭开盖子，底下引用的数据结构必须在用前初始化。比如切片是一个三项的描述符，包含数据指针（数组内），长度，和容量；在这些项初始化前，切片为 nil 。对于切片、映射和信道，make 初始化内部数据结构，并准备要用的值。例如，

```
make([]int, 10, 100)
```

分配一个 100 个整数的数组，然后生成一个切片结构，长度为10，容量是100的指向此数组的首10项。（生成切片时，容量可以不写；详见切片一节。）对应的，new([]int) 返回一个新分配的，清零的切片结构，亦即，一个 nil 切片值的指针。

下面的例子展示了 new() 和 make() 的不同。

```
var p *[]int = new([]int)       // 分配切片结构; *p == nil; 非
常少见
var v  []int = make([]int, 100) // 切片 v 代表100个整数的新数组

// 无必要的复杂:
var p *[]int = new([]int)
```

```
*p = make([]int, 100, 100)

// 惯用法:
v := make([]int, 100)
```

记住 make() 只用于映射、切片和信道，不返回指针。要明确的得到指针用 new() 分配。

数组

数组用于安排详细的内存布局，还有助于避免分配，但其主要作为切片的构件，即下节的主题。这里先讲几句打个底儿。

Go 和 C 的数组的主要不同在于：

- 数组为值。数组赋值给另一数组拷贝其全部元素。
- 特别是，如果你传递数组给一个函数，它受到此数组的拷贝，不是指针。
- 数组的尺寸是其类型的一部分。[10]int 和 [20]int 是完全不同的类型。

值的属性可用但昂贵；如你所需的是类似 C 的行为和效率，你可以传递一个指针给数组。

```
func Sum(a *[3]float) (sum float) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float{7.0, 8.5, 9.1}
x := Sum(&array)  // 注意此处明确的地址运算符
```

即便如此也不是地道的 Go 风格。切片才是。

切片

切片包装数组，给数据系列一个通用、强力、方便的界面。除了像变换矩阵那种要求明确尺寸的情况，绝大部分的数组编程在 Go 里使用切片、而不是简单的数组。

切片是引用类型，即如果赋值切片给另一个切片，它们都指向同一底层数组。例如，如果某函数取切片参量，对其元素的改动会显现在调用者中，类似于传递一个底层数组的指针。因此 Read 函数可以接受切片参量，而不需指针和计数；切片的长度决定了可读数据的上限。这里是 os 包的 File 型的 Read 方法的签名：

```
func (file *File) Read(buf []byte) (n int, err os.Error)
```

此方法返回读入字节数和可能的错误值。要读入一个大的缓冲 b 的首32字节， 切片（动词）缓冲。

```
n, err := f.Read(buf[0:32])
```

这种切片常用且高效。实际上，先不管效率，此片段也可读缓冲的首32字节。

```
    var n int
    var err os.Error
    for i := 0; i < 32; i++ {
        nbytes, e := f.Read(buf[i:i+1])  // Read one byte.
        if nbytes == 0 || e != nil {
            err = e
            break
        }
        n += nbytes
    }
```

只要还在底层数组的限制内，切片的长度可以改变，只需赋值自己。切片的容量，可用内部函数 cap 取得，给出此切片可用的最大长度。下面的函数给切片添值。如果数据超过容量，切片重新分配，返回结果切片。此函数利用了 len 和 cap 对 nil 切片合法、返回0的事实。

```
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // reallocate
        // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
        // The copy function is predeclared and works for
any slice type.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
    return slice
}
```

我们必须返回切片，因为尽管 Append 可以改变 slice 的元素， 切片自身（持有指针、长度和容量的运行态数据结构）是值传递的。

添加切片的主意很有用，因此由内置函数 append 实现。要理解此函数的设计，我们需要多一些信息，所以稍后再讲。

映射

映射提供了一个方便强力的内部数据结构，用来联合不同的类型。键可以是任何定义了相等操作符的类型，如整型，浮点型，字串，指针，界面（只要其动态类型支持相等）。结构，数组和切片不可用作映射键，因为其类型未定义相等。类似切片，映射是引用类型。如果你传递映射给某函数，对映射的内容的改动显现给调用者。

映射的生成使用平常的冒号隔开的键值伴组合字面句法，所以很容易初始化时建好它们。

```
var timeZone = map[string] int {
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

赋值和获取映射值语法上就像数组，只是下标不需是整型。

```
offset := timeZone["EST"]
```

试图获取不存在的键的映射值返回对应条目类型的零值。例如，如果映射包含整型数，查找不存在的键返回0。

有时你需区分不在键和零值。 是没有“UTC”的条目，还是因为其值为零？你可以用多值赋值的形式加以区分。

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

道理很明显，此习语称为“逗号ok”。此例中，如果 tz 存在，seconds 相应赋值，ok为真；否则，seconds 为0，ok为假。下面的函数加上了中意的出错报告：

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone", tz)
    return 0
```

```
}
```

要检查映射的存在，又不想管实际值，你可以用空白标识，即下划线（ _ ）。空白标识可以赋值或声明为任意类型的任意值，会被无害的丢弃。如只要测试映射是否存在， 在平常变量的地方使用空白标识即可。

```
_, present := timeZone[tz]
```

要删除映射条目，翻转多值赋值，在右边多放个布尔；如果布尔为假，条目被删。即便键已经不再了，这样做也是安全的。

```
timeZone["PDT"] = 0, false  // Now on Standard Time
```

打印

Go 的排版打印风格类似 C 的 printf 族但更丰富更通用。这些函数活在 fmt 包里，叫大写的名字：fmt.Printf，fmt.Fprintf， fmt.Sprintf 等等。字串函数（Sprintf 等）返回字串，而不是填充给定的缓冲。

你不需给出排版字串。对应每个 Printf，Fprintf 和 Sprintf 都有另一对函数。例如 Print 和 Println。 它们不需排版字串，而是用每个参量默认的格式。Println 版本还会在参量间加入空格和输出新行，而 Print 版本只当操作数的两边都不是字串时才添加空格。下例每行的输出都是一样的：

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

如《辅导》里所讲，fmt.Fprint 和伙伴们的第一个参量可以是任何实现 io.Writer 界面的物件。变量 os.Stdout 和 os.Stderr 是熟悉的实例。

从此事情开始偏离 C 了。首先，数字格式如 %d 没有正负和尺寸的标记；打印例程使用参量的类型决定这些属性。

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

打印出

```
18446744073709551615 ffffffffffffffff; -1 -1
```

如果你只需默认的转换，例如整数用十进制，你可以用全拿格式 %v（代表 value）；结果和Print 与 Println 打印的完全一样。再有，此格式可打印任意值，包括数组，结构和映射。这里是上节定义的时区映射的打印语句。

```
fmt.Printf("%v\n", timeZone)  // or just
fmt.Println(timeZone)
```

打印出：

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

当然，映射的键会以任意顺序输出。打印结构时，改进的格式 %+v 用结构的域名注释，对任意值格式 %#v 打印出完整的 Go 句法。

```
type T struct {
    a int
    b float
    c string
}

t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

打印出

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000,
"UTC":0, "MST":-25200}
```

（注意和号&）。引号括起的字串也可以 %q 用在 string 或 []byte 类型的值上，对应的格式 %#q 如果可能则使用反引号。还有，%x 可用于字串、字节数组和整型，得到长的十六进制串，有空格的格式（% x）会在字节间加空格。

另一好用的格式是 %T，打印某值的类型。

```
fmt.Printf("%T\n", timeZone)
```

打印

```
map[string] int
```

如果你要控制某定制类型的默认格式， 只需在其类型上定义方法 String() string。对我们简单的类型 T，可以是：

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

来打印

```
7/-2.35/"abc\tdef"
```

我们的String() 方法可以调用 Sprint，因为打印例程是完全可以重入可以递归的。我们可以更进一步，把一个打印例程的参量直接传递给另一打印例程。 Printf 的签名的首参量使用类型 ...interface{}，来指定任意数量任意类型的参量可以出现在格式字串的后面。

```
func Printf(format string, v ...interface{}) (n int, errno
os.Error) {
```

Printf 函数中，v 像是一个 []interface{} 类的变量。但如果把它传递给另一个多维函数，它就像一列普通的参量。这里是我们上面用过的 log.Println 的实现。它把自己的参量直接传递给 fmt.Println 来实际打印。

```
// Println prints to the standard logger in the manner of
fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...)) //Output takes parame-
ters(int,string)
}
```

我们在 Sprintln 的调用的 v 后写 ... 告诉编译器把 v 作为一列参量；否则它只是传递一个单一的切片参量。

还有很多打印的内容我们还没讲，细节可参考 godoc 的 fmt 包的文档。

顺便提一句， ... 参量可以是任意给定的类型，例如，...int 在 min 函数里可以选一列整数的最小值。

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1)  // largest int
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Append

现在我们解释 append 的设计。append 的签名和上面我们定制的 Append 函数不同。大体上是：

```
func append(slice[]T, elements...T)[]T
```

T 替代的是任意类型。 实际中你不能写 Go 的函数由调用者决定 T 的类型，所以 append 内置：它需要编译器的支持。

append 所做的是在切片尾添加元素并返回结果。结果需要返回因为，正如我们手写的 Append，底层的数组可能更改。下面简单的例子：

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

打印 [1 2 3 4 5 6]。所以 append 有点像 Printf 收集任意数量的参量。

但如何像我们 Append 一样给切片添加切片呢？容易：使用 ... 在调用的地方，正如我们上面我们调用 Output。 下例产生如上同样的输出：

```
x := []int{1, 2, 3}
y := []int{4, 5, 6}
x = append(x, y...)
fmt.Println(x)
```

没有 ... 将不能编译，因为类型错误； y 不是 int 类型。

初始化

尽管表面看来和 C 或 C++ 的初始化没什么不同，Go 的更够强。复杂结构可在初始化时架设，并且不同包的物件的初始化顺序问题也得到正确处理。

常量

常量在 Go 里是 —— 不变的。它们在编译时生成，即便是局部定义在函数里。它只能是数，字串或布尔。因为编译态的限制，定义它们的表达式必须是常量表达式，可以被编译器求值。例如，1<<3 是常量表达式， math.Sin(math.Pi/4) 不是，因为 math.Sin 的函数调用发生在运行态。

Go 的列举常量可用 iota 生成。 因为 iota 可以是表达式的一部分，并且表达式可以隐含重复，打造一套精致的值可以变得很容易。

```
type ByteSize float64
const (
    _ = iota  // 忽略第一个值，赋值给空白标识
    KB ByteSize = 1<<(10*iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

给类型添加比如 String 等方法的本领，使值自动排版打印自己变得可能，即使只作为通用类型的一部分。

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
```

```
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}
```

表达式 YB 打印 1.00YB，而 ByteSize(1e13) 打印 9.09TB

变量

变量和常量的初始化一样，但可以用运行态计算的普通表达式。

```
var (
    HOME = os.Getenv("HOME")
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)
```

init 函数

最后，每个源文件可以定义自身的 init() 函数来安排所需的状态。唯一的限制是，尽管够程可在初始化时启动，它们只在初始完成后执行；初始化永远是单一的执行序列。最后之后，init() 发生在包里所有变量初始化之后，而其又发生在所有的包全部导入之后。

除了初始化不能表示为声明外，init() 函数常用来在程序运行前验证或修补其状态。

```
func init() {
    if USER == "" {
        log.Exit("$USER not set")
    }
    if HOME == "" {
        HOME = "/usr/" + USER
    }
    if GOROOT == "" {
        GOROOT = HOME + "/go"
    }
    // GOROOT may be overridden by --goroot flag on command
line.
    flag.StringVar(&GOROOT, "goroot", GOROOT, "Go root di-
rectory")
}
```

方法

指针和值

方法可用于任意带名的非指针和界面的类型；接受者没必要是结构。

在上面讨论切片时，我们写了个 Append 函数。其实我们可以把它定义为切片的方法。首先我们声明一个带名的类型，以便我们在其上施加方法，并使此方法的接受者的值是此类型。

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // Body exactly the same as above
}
```

这仍需方法返回更新的切片，更灵活的方式是定义方法接受 ByteSlice 的指针，以便重写调用着的切片。

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Body as above, without the return.
    *p = slice
}
```

实际上，我们能做的更好。如果我们把函数改的像标准的 Write 方法，例如：

```
func (p *ByteSlice) Write(data []byte) (n int, err os.Error)
{
    slice := *p
    // Again as above.
    *p = slice
    return len(data), nil
}
```

此时类型 *ByteSlice 可以满足标准界面 io.Writer，很方便的，例如我们打印到里面：

```
var b ByteSlice
fmt.Fprintf(&b, "This hour has %d days\n", 7) // ”这一小时有7天“是时事节目
```

我们传递 ByteSlice 的地址是因为只有 *ByteSlice 满足 io.Writer。接受者的指针和值规则是，值的方法可用于指针和值，而指针的方法只用

于指针。这是因为指针方法可以改变接受者；使用拷贝的值会导致这些改变的丢失。

顺便一提，字节切片的 Write 已在 bytes.Buffer 实现。

界面和其它类型

界面

Go 的界面提供了指定物件行为的方式：如某物可以做这个，则它可以用在这里。我们已经看到一些简单的例子，定制的打印可以用 String 方法实现，Fprintf 可以输出到任何有 Write 方法的东西。只有一两件方法的界面在 Go 里很常见，并常用其方法命名，如 io.Writer 指实现了 Write 的东西。

一个类型可以实现多个界面。例如，一个收集可以使用 sort 包的例程排序，只要它实现了 sort.Interface，这包括 Len(), Less(i, j int) bool 和 SWap(i, j int)，它还可以有定制的排版器。下面编造的例子满足所有这些：

```
type Sequence []int

// Methods required by sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
```

```
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Method for printing - sorts the elements before printing.
func (s Sequence) String() string {
    sort.Sort(s)
    str := "["
    for i, elem := range s {
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

转换

Sequence 的 String 方法重做了 Sprint 在切片的工作。我们可以在调用 Sprint 前把 Sequence 转为普通的 []int。

```
func (s Sequence) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

转换使得 s 被当作普通的切片，因此得到默认的排版。不加转换，Sprint 会发现 Sequence 的 String 方法，进而无穷递归。如果忽略类型名称，Sequence 和 []int 的类型相同， 它们之间的转换是合法的。转换不会得到新值，它只是暂时假装现有值是新类型。（其它合法的转换，如从整型到浮点型，会生成新值。）

地道的 Go 程序会转换表达式的类型来使用一组不同的方法。例如，我们可以用现有类型 sort.IntArray 把整个例子缩减为：

```
type Sequence []int

// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.IntArray(s).Sort()
    return fmt.Sprint([]int(s))
}
```

现在，无需让 Sequence 实现多个界面（排序和打印），我们使用了把数据转换为多种类型（Sequence，sort.IntArray 和 []int）的能力，每个来完成一部分的工作。这实际上不常见但很有效。

泛化

如果某类型的存在只为了实现某界面，而除此之外没有其它导出的方法，则此类型也不需导出。只导出界面明确了只有行为有价值，而不是实现，其它的不同特性的实现可以镜像其原来的类型。这样也避免了为同一方法的不同实现做文档。

此时，架构函数应返回界面而不是实现类型。例如，哈希库的 crc32.NewIEEE() 和 adler32.New() 都返回界面类型 hash.Hash32。替换一个 Go 出现的 CRC-32 算法为 Adler-32 只需改变架构函数的调用；其余的代码不受算法改变的影响。

同样的方式使得 crypto/block 包的流密（streaming cipher）算法与链接在一起的块密（block cipher）相区隔。比对 bufio 包，它们包装了

Cipher 界面，返回 hash.Hash，io.Reader 和 io.Writer 界面值，而不是特定的实现。

crypto/block 界面包括：

```
type Cipher interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}

// NewECBDecrypter returns a reader that reads data
// from r and decrypts it using c in electronic codebook
(ECB) mode.
func NewECBDecrypter(c Cipher, r io.Reader) io.Reader

// NewCBCDecrypter returns a reader that reads data
// from r and decrypts it using c in cipher block chaining
(CBC) mode
// with the initialization vector iv.
func NewCBCDecrypter(c Cipher, iv []byte, r io.Reader)
io.Reader
```

NewECBDecrypter 和 NewCBCReader 不只用于某特定的加密算法和数据源，而是任意的 Cipher 界面的实现和任意的 io.Reader。因为它们返回 io.Reader 界面值，替换 ECB 加密为 CBC 加密只是局部修改。 架构函数必须编辑，但因为周围代码必须只把结果作为 io.Reader，它不会注意到有什么不同。

界面和方法

因为几乎任何东西都可加以方法，几乎任何东西都可满足某界面。一个展示的例子是 http 包定义的 Handler 界面。任何物件实现了 Handler 都可服务 HTTP 请求。

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

ResponseWriter 本身是个界面，它提供一些可访问的方法来返回客户的请求。这些方法包括标准的 Write 方法。因此 http.ResponseWriter 可用在 io.Writer 可以使用的地方。Request 是个结构，包含客户请求的一个解析过的表示。

为求简短，我们忽略 POST 并假定所有 HTTP 请求都是 GET；此简化不会影响经手者的设置。下面一个小而全的经手者实现了网页访问次数的计数。

```
// Simple counter server.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req
*http.Request) {
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

（注意 Fprintf 怎样打印到 http.ResponseWriter）。作为参考，这里是怎样把服务者加在一个 URL 树的节点上。

```
import "http"
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

可是为何把 Counter 作为结构呢？一个整数能够了。（接受者需是指针，使增量带回调用者）。

```
// Simpler counter server.
type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req
*http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

当某页被访问时怎样通知你的程序更新某些内部状态呢？给网页贴个信道。

```
// A channel that sends a notification on each visit.
// (Probably want the channel to be buffered.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req
*http.Request) {
    ch <- req
    fmt.Fprint(w, "notification sent")
}
```

最后，让我们在 /args 显示启动服务器时的参量。写个打印参量的函数很容易：

```
func ArgServer() {
    for i, s := range os.Args {
        fmt.Println(s)
    }
}
```

怎样把它变成 HTTP 服务器呢？我们可以把 ArgServer 作为某个类型的方法再忽略其值，也有更干净的做法。既然我们可以给任意非指针和界面的类型定义方法，我们可以给函数写个方法。http 包里有如下代码：

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers.  If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler object that calls f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(c, req).
func (f HandlerFunc) ServeHTTP(c http.ResponseWriter, req
*Request) {
    f(w, req)
}
```

HandlerFunc 是个带 ServeHTTP 方法的类型， 所以此类的值都可以服务 HTTP 请求。我们来看看此方法的实现：接受者是个函数，f，方法调用 f 。看起来很怪，但和，比如，接受者是信道，而方法发送到此信道，没什么不同。

要把 ArgServer 变为 HTTP 服务器， 我们首先改成正确的签名：

```
// Argument server.
```

```
func ArgServer(c http.ResponseWriter, req *http.Request) {
    for i, s := range os.Args {
        fmt.Fprintln(w, s)
    }
}
```

ArgServer 现在和 HandlerFunc 有同样的签名，就可以转成此类使用其方法，就像我们把 Sequence 转为 IntArray 来使用 IntArray.Sort 一样。设置代码很简短：

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

当有人访问 /args 页时，此页的经手者有值 ArgServer 和类型 HandlerFunc。HTTP 服务器启动此类型的 ServeHTTP 方法，用 ArgServer 作为接受者，反过来调用 ArgServer （通过启动 handlerFunc.ServeHTTP 的 f(w, req) 。）参量被显示出来。

此节中我们从一个结构，整数，信道和一个函数制造出一个 HTTP 服务器，全赖于界面就是一套方法，可定义在（几乎）任何类型上。

内置

Go 不提供通行的、类型驱动的子类划分的概念，但它通过在结构或界面内置，确实有能力从某实现“借”些片段。

界面内置非常简单。我们提到过 io.Reader 和 io.Writer 的界面；这里是其定义：

```
type Reader interface {
```

```
    Read(p []byte) (n int, err os.Error)
}

type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

io 包也导出一些其它的界面来规范实现其多个方法的物件。例如，io.ReadWriter 界面同时包括 Read 和 Write 。我们可以明确的列出这两个方法来规定 io.ReadWriter ，但更简单更有启发的是内置这两个界面形成新界面，如下：

```
// ReadWriter is the interface that combines the Reader and
Writer interfaces.
type ReadWriter interface {
    Reader
    Writer
}
```

顾名思意，ReadWriter 可做 Reader 和 Writer 两个的事；它是内置界面的集合（必须是没有交集的方法）。只有界面可以内置在界面里。

同样的基本概念适用于结构，但牵连更广。bufio 包有两个结构类型，bufio.Reader 和 bufio.Writer ，每个都当然对应实现着 io 包的界面。并且，bufio 也实现了缓冲的读写，这是通过把 reader 和 writer 内置到一个结构做到的；它列出类型但不给名称：

```
// ReadWriter stores pointers to a Reader and a Writer.
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

内置元素是指针所以使用前必须初始化指向有效的结构。 ReadWriter 结构可以写为：

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

但要提升域的方法又要满足 io 界面，我们还需提供转发方法，如：

```
func (rw *ReadWriter) Read(p []byte) (n int, err os.Error) {
    return rw.reader.Read(p)
}
```

通过直接内置结构，我们避免了这些账目管理。内置类型的方法是附送的，亦即 bufio.ReadWriter 除了有 bufio.Reader 和bufio.Writer 的方法，还同时满足三个界面：io.Reader，io.Writer 和 io.ReadWriter 。

内置和子类划分有着重要不同。我们内置类型时，此类的方法成为外层类型的方法，但调用时其接受者是内层类型，而不是外层。我们的例子里，当 bufio.ReadWriter 的 Read 方法被调用时，它的效果和上面所写的转发方法完全一样；接受者是 ReadWriter 的 reader，而不是 ReadWriter 自身。

内置也用来提供某种便利。下例是一个内置域和普通的，带名的域在一起：

```
type Job struct {
    Command string
```

```
    *log.Logger
}
```

此时 Job 类型有 Log，Logf 和其它 *log.Logger 的方法。 我们当然可以给 Logger 个名字，但无此必要。这里，初始化后，我们可以 log Job：

```
job.Log("starting now...")
```

Logger 是结构的普通域所以我们能用通常的架构函数初始化它：

```
func NewJob(command string, logger *log.Logger) *Job {
    return &Job{command, logger}
}
```

或使用组合字面：

```
job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
```

如果我们需要直接引用内置域，域的类型名，忽略包标识，可作为域名。如果我们要得到 Job 变量 job 的 *log.Logger，我们用 job.Logger 。这可用在细化 Logger 的方法上：

```
func (job *Job) Logf(format string, args ...) {
    job.Logger.Logf("%q: %s", job.Command,
fmt.Sprintf(format, args))
}
```

内置类型导致撞名的问题，但解决方案很简单。首先，域或方法 X 隐藏此类型更深层部分的 X 项。如果 log.Logger 包括叫 Command 的域或方法，则 Job 的 Command 域占主导权。

其次，如果同层出现同名，通常是个错误；如果 Job 结构有另一域或方法叫 Logger，要内置 log.Logger 会出错。但只要重名在类型定义外的程序中不被提及，就不成问题。此条件提供了改动外部的内置类型的某种保护；只要两个域都没被用到，新增的域名和另一子类的域重名也没有问题。

并发

交流来分享（share by communicating)

并发编程是很大的主题，此处只够讲 Go 方面的要点。

很多环境的并发编程变得困难出自于实现正确读写共享变量的微妙性。Go 鼓励一种不一样的方式，这里，共享变量在信道是传递，并且事实上，从来未被独立的执行序列所共享。每一特定时间只有一个够程在存取该值。从设计上数据竞争就不会发生。为鼓励这种思考方式我们把它缩减成一句口号：

> 别靠共享内存来通信，要靠通信来分享内存

此方式可扯的太远。例如，引用计数最好靠整型变量外加一个互斥。但最为高层方式，使用信道控制存取可以更容易写成清楚正确的程序。

思考这种模型一种方式是考虑在单一 CPU 上运行的典型的单线程程序，不需用到同步原语。现在多运行一份；也同样不需同步。现在让两者通信；如果通信是同步者，还是不需其它的同步。例如，Unix 的管道完美的适合这个模型。尽管 Go 的并行方式源自 Hoare 的通信顺序进程（CSP），它也可视为泛化的类型安全的 Unix 管道。

够程

它们叫做够程， 是因为现有的术语 -- 线程，协程和进程等 -- 传达的含义不够精确。够程的模式很简单：它是在同一地址空间和其它够程并列执行的函数。它轻盈，只比分配堆栈空间多费一点儿。因为堆栈开始时很小，所以它们很便宜，只在需要时分配（和释放）堆库存。

够程在多个 OS 线程间复用，所以如果某个需要阻塞，例如在等待 IO，其它的可继续执行。它们的设计隐藏了许多线程生成和管理的复杂性。

在某个函数或方法前加上 go 键字则在新够程中执行此调用。当调用完成，够程安静的退出。（效果类似 Unix shell 的 & -- 在后台执行命令。）

```
go list.Sort()  // run list.Sort in parallel; don't wait for
it.
```

函数字面在实施够程上很顺手：

```
func Announce(message string, delay int64) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
```

```
    }()  // 注意小括号，必须调用函数。
}
```

在 Go 里，函数字面是闭包：实现保证函数引用的变量可以活到它们不在用了。

这些例子没什么用处，因为函数无法通知其结束。那需用到信道。

信道

类型映射，信道是引用类型，使用 make 分配。如果提供了可选的整型参量，它会设置信道的缓冲大小。默认是0，即无缓冲的或同步的信道。

```
ci := make(chan int)            // unbuffered channel of in-
tegers
cj := make(chan int, 0)         // unbuffered channel of in-
tegers
cs := make(chan *os.File, 100)  // buffered channel of
pointers to Files
```

信道结合了通信 -- 即值的交换 -- 和同步 -- 确保两个计算（够程）处于某个已知状态。

信道有很多惯用语。先从一个开始。上节我们启动了个后台的排序。信道使启动够程能等待排序完成。

```
c := make(chan int)  // 分配信道
// 在一个够程开始排序，完成时发回一个信号
go func() {
    list.Sort()
    c <- 1  // 发回一个信号，何值不重要
```

```
}()
doSomethingForAWhile()
<-c    // 等待排序结束，丢弃发来的值
```

接收者阻塞到有数据可以接收。如果信道是非缓冲的，发送者阻塞到接收者收到其值。如果信道有缓冲，发送者只需阻塞到值拷贝到缓冲里；如果缓冲满，则等待直到某个接收者取走一值。

一个缓冲信道可以用作信号灯，比如用来限速。下例中，到来请求传递给 handle，来发送一值到信道，处理请求，以及才信道接收值。信道的容量决定了可同时调用 process 的数量。

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // 等待活跃队列清空
    process(r)  // 可能需很久
    <-sem       // 完。使队列的下一位可以运行
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // 不必等到 handle 结束
    }
}
```

同样的概念，我们可以启动一定数量的 handle 够程，全都读取请求信道。够程的数量限制同时调用 process 的数量。此 Serve 函数也接受一个信道来告知它退出；够程启动后会接收阻塞在此信道。

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
```

```
    }
}

func Serve(clientRequests chan *clientRequests, quit chan
bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Wait to be told to exit.
}
```

信道的信道

Go 的一个最重要的特色是信道作为一等值可以被分配被传递，正如其它的值。此特色常用来实现安全并行的分路器。

上节的例子里，handle 是个理想化的请求经手者，但我们并未定义其经手的类型。如果此类型包括一个可回发的信道，每个客户都可提供自身的回答途径。下面是类型 Request 的语义定义。

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

客户提供一个函数及其参量，以及在请求物件里的一个信道，用来接收答案。

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
```

```
    return
}
request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// 发送请求
clientRequests <- request
// 等待回答
fmt.Printf("answer: %d\n", <-request.resultChan)
```

服务器端，经手函数是唯一需要改变的。

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

当然还要很多工作使其实际，但此代码是个速率限制、并行、非阻塞的 RPC 系统的框架，而且看不到一个互斥。

并发

这些概念的另一应用是在多 CPU 核上并发计算。如果一个运算可以分解为独立片段，则可并发，用一信道通知每个片段的结束。

比如我们有个很花时间的运算执行在一列项上，每个项的运算值都是独立的，如下例：

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
```

```
    c <- 1    // 通知此片段完成
}
```

我们在循环里单独启动片段，每个 CPU 一个。 它们谁先完成都没关系；我们只是启动全部够程前清空信道，再数数结束通知即可。

```
const NCPU = 4  // CPU 核数

func (v Vector) DoAll(u Vector) {
    c := make(chan int, NCPU)  // 缓冲可选但可感觉到
    for i := 0; i < NCPU; i++ {
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // 清空信道
    for i := 0; i < NCPU; i++ {
        <-c    // 等待某个任务结束
    }
    // 全好了
}
```

现在的 gc 实现（6g 等）不会默认的并发此代码。它只投入一个核给用户层的运算。任意多的够程可以阻塞在系统调用上，但默认的每个时刻只有一个可以执行用户层的代码。它本该更聪明些，某天它会变聪明，但那之前如果你要并发 CPU 就必须告知运行态你要同时执行代码的够程的数量。有两种相关的办法，或者你把环境变量 GOMAXPROCS 设为你要用到的核数（默认1）；或者导入 runtime 包调用 runtime.GOMAXPROCS(NCPU)。再提一遍，此要求会随着调度及运行态的进步而退休。

漏水缓冲

并发编程的工具也可用来使非并发的概念更容易表达。下例是从某个 RPC 包里提取的。客户够程循环接收数据自某源，可能是网络。为免分配释放缓冲，它保有一个自由列，并由一个缓冲的信道代表。如果信道空，则新缓冲被分配。当消息缓冲好时，它在 serverChan 上发给服务器。

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        b, ok := <-freeList  // 获取一个缓冲，如有
        if !ok {              // 否则，分配个新缓冲
            b = new(Buffer)
        }
        load(b)              // 从网上读消息
        serverChan <- b      // 发给服务器
    }
}
```

服务器循环读消息自客户，处理，返回缓冲到自由列。

```
func server() {
    for {
        b := <-serverChan    // 等待工作
        process(b)
        _ = freeList <- b    // 重用缓冲，如有空位
    }
}
```

客户无阻的从 freeList 得到一个缓冲，如还没有则客户分配个新的缓冲。服务器无阻的发送给 freeList，放 b 回自由列，除非列满，此时

缓冲掉到地板上被垃圾收集器回收。（发送操作赋值给空白标识使其无阻但会忽略操作是否成功。）此实现仅用几行就打造了个漏水缓冲，靠缓冲信道和垃圾收集器记账。

错误

库例程经常必须返回某种错误指示给调用者。如前所述，Go 的多值返回使得随返回值同时返回详细错误描述变得很容易。习惯上，错误带类型 os.Error，一个简单的界面。

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

库的作者可以自由的用更丰富的模式实现此界面，使其不仅报错也提供一些相关内容。例如， os.Open 返回一个 os.PathError 。

```
// PathError 记录错误和导致它的运算及文件路径
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Error Error  // Returned by the system call.
}

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

PathError 的 String 生成如下字串：

```
open /etc/passwx: no such file or directory
```

此错误，含带其出错文件名、运算和其引发的操作系统错误，甚至打印在远离导致其出错的调用时也很有用；它带的信息远多于单单的“无此文件或目录”。

需要精确出错细节的调用者可以使用类型切换或类型断言来查找特定错误并提取细节。对 PathErrors 这可能包括检查内部的 Error 域用来从失败处恢复。

```
for try := 0; try < 2; try++ {
    file, err = os.Open(filename, os.O_RDONLY, 0)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Error ==
os.ENOSPC {
        deleteTempFiles()  // Recover some space.
        continue
    }
    return
}
```

怕死

通常报错的方式是给调用者一个多于的 os.Error 的返回值。经典的 Read 方法是个出名的实例；它返回字节数和 os.Error 。但错误不可恢复则如何？有时程序就是不可再继续了。

基于此目的，内部函数 panic 实际上会生成一个运行态错误来终止程序（但参见下节）。此函数取一个任意类型的参量 —— 通常是字串——在程序死掉时打印。它也用来指出某种不可能的事情发生了，例

如从永久循环中退出了。实际上，编辑器看到函数尾的 panic 会压制通常的 return 语句检查。

```
// 牛顿立方根求法的玩具实现
func CubeRoot(x float64) float64 {
    z := x/3   // 随意的初始值
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // 百万次后仍未收敛，不对劲
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

这只是个例子，实际的库函数要避免使用 panic 。如果某问题可被屏蔽或绕过，最好让事情继续而不是打死整个程序。一个可能的反例是在初始化时，如果某库函数怎么都不能安排好自己，有理由 panic，可以这么说。

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

回生

当 panic 被叫，包括运行态错误例如数组下标越界或类型断言失败时，它会立即停止当前函数的执行，并开始退绕够程的堆栈，随之运行所有的延迟函数。如果退绕到够程堆栈顶，程序死掉。但是，我们可以用内部函数 recover 重新控制够程，恢复正常运行。

recover 的调用终止退绕并返回传给 panic 的参量。因为退绕时只有延迟函数的代码在运行，recover 只在延迟函数有用。

recover 的一个用途是在服务器内关闭失败的够程而不会杀死其它正在运行的够程。

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

此例子中，如果 do(work) panic了，结果会记录下，够程会不扰人的干净地退出。没必要在延迟函数做其它的事；recover 的调用完全可以处理。

注意有了这种复原的模式，do 函数（及其所有的调用）可以用 panic 从任何糟糕的情况里脱身。我们可用此概念简化复杂软件的出错处理。我们看看 regexp 包里一个理想化的节选，它用局部的 Error 类型调用 panic 来报错。 下面是 Error，error 方法，和 Compile 函数的定义：

```
// Error is the type of a parse error; it satisfies
os.Error.
type Error string
func (e Error) String() string {
    return string(e)
}

// error is a method of *Regexp that reports parsing errors
by
// panicking with an Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile returns a parsed representation of the regular
expression.
func Compile(str string) (regexp *Regexp, err os.Error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Clear return value.
            err = e.(Error) // Will re-panic if not a parse
error.
        }
    }()
    return regexp.doParse(str), nil
}
```

如果 doParse panic了，复原块会设置返回值为 nil ——延迟函数可以修改带名的返回值。它然后通过断定 err 的赋值是类型 Error 来检查问题出自语法分析。如果不是，类型断言会失败，导致一个运行态错误，继续堆栈退绕，就好像无事发生一样。这个检查意味着如果未曾预料的事情发生了，例如数组下标越界，代码会失败，尽管我们用了 panic 和 recover 出来用户触发的错误。

有了这种出错处理，error 方法能轻易的报错，而不需担心自己动手退绕堆栈。

这种有用的模式只应在一个包的内部使用。 Parse 将其内部的 panic 调用转为 os.Error 值；不把 panic 暴露给客户。这个好规则值得效法。

网舍

作为结束我们写一个完整的 Go 程序，一个网页服务器。这个实际上是个网页再服务器。Google 在 http://chart.apis.google.com 提供的服务能自动把数据排版为图表。交互使用它不容易，因为你要把数据作为查询放在 URL 里。下面的程序给一种格式的数据提供了个好用些的界面：给出一小片文本，它会拨给图表服务器生成一个 QR 码，即一个编码文本的方块矩阵。此图像可以用你手机的镜头读取，解释为，比如，一个 URL，省得你用手机幼小的键盘打入 URL 。

下面是完整的程序，跟着的是解释。

```
package main
```

```
import (
    "flag"
    "http"
    "io"
    "log"
    "template"
)

var addr = flag.String("addr",":1718","http service ad-
dress") //Q=17, R=18
var fmap = template.FormatterMap{
    "html": template.HTMLFormatter,
    "url+html": UrlHtmlFormatter,
}
var templ = template.MustParse(templateStr, fmap)

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Exit("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    templ.Execute(req.FormValue("s"), w)
}

func UrlHtmlFormatter(w io.Writer, v interface{}, fmt
string) {
    template.HTMLEscape(w,
[]byte(http.URLEscape(v.(string))))
}


const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
```

```
</head>
<body>
{.section @}
<img
src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&c
hoe=UTF-8&chl={@|url+html}"
/>
<br>
{@|html}
<br>
<br>
{.end}
<form action="/" name=f method="GET"><input maxLength=1024
size=70
name=s value="" title="Text to QR Encode"><input type=submit
value="Show QR" name=qr>
</form>
</body>
</html>
`
```

main 之前的片段应很容易看懂。一个旗标设置我们服务器的默认 HTTP 端口。 template 变量 templ 开始有趣。它打造一个 HTML 模版供服务器显示页面；稍后在细讲。

main 函数分析旗标，并用我们上面谈到的机制，绑定函数 QR 和服务器的根路径。然后调用 http.ListenAndServe 启动服务器；它阻塞到服务器开始运行。

QR 仅是接收请求，里边包含着数据，在表格值 s 上执行模版。

template 包，启发自 json-template，功能强大；此程序仅仅触及了它的能力。本质上，它动态改写一段文本，用传递给 templ.Execute 的数据项、此处是表格值，来替换元素。在模版文本（templateStr）

里，大括号括起的片段代表模版行为。从{.section @} 到 {.end} 的片段使用数据项 @ 的值执行，它是“当前项”的简写，亦即表格值。（当字串空时，这片模版被禁止。）

片段 {@|url+html} 说的是用安装在排版映射（fmap）名字 “url+html” 下是排版器过一遍数据。此处是 UrlHtmlFormatter，会给字串消毒使其安全显示在网页上。

剩余的模版字串只是加载网页时该显示的 HTML 。如果解释得太快，参见 template 包文档的更详细的讨论。

这样你得到了：一个有用的网页服务器，只用了几行代码加一些数据驱动的 HTML 文本。Go 够力，寥寥数语便成就很多。

够规范 Go Spec

原文：http://golang.org/doc/go_spec.html

版本：7/29/2010

介绍

此文是 Go 编程语言的参考手册。其它的信息和文档参见 http://golang.org 。

Go 是面向系统编程的通用语言。它具有强类型、垃圾回收和对并发编程的明确支持。程序是由包架构的，其属性确保了高效的依互管理。现有实现使用传统的编译/链接模型生成可执行二进制码。

语法紧凑规整，使像集成开发环境这样的自动工具分析起来更容易。

标识法

句法规范使用扩展巴科斯范式（EBNF）：

```
产品    = 品名 "=" 表达 "." .
表达    = 替换 { "|" 替换 } .
替换    = 条款 { 条款 } .
条款    = 品名 | 符号 [ "..." 符号 ] | 分组 | 选项 | 重复 .
分组    = "(" 表达 ")" .
选项    = "[" 表达 "]" .
```

```
重复    = "{" 表达 "}" .
```

产品是由条款和操作符构成的表达式。操作符如下，优先级从低到高：

```
|       替换
()      分组
[]      选项（0 或 1 次）
{}      重复（0 或 n 次）
```

小写的品名代表词法符号。非终结符为大小混写。词法标识由双引号“ ”或反引号` `括起。

a...b的形式代表一组字符从 a 到 b 作为替换。

源码表示

源码是 UTF-8 编码的 Unicode 文本。文本未经规范化，因此单个带调码点和从一个字母与声调组合得到的同一字符是完全不同的；后者被当作两个码点。简单起见，本文使用术语字符代表 Unicode 的码点。

每个码点都是不同的，例如，大写和小写的字母是不同的字符。

实现限制：为和其它工具兼容，某编译器可能不允许源代码出现零字符（U+0000）。

字符

下面的条款用来表示特定的 Unicode 字符类：

```
unicode_char   = /* 任何 Unicode 码点 */
unicode_letter = /* 一个分类为 “字母“ 的 Unicode 码点 */
unicode_digit  = /* 一个分类为 “数字“ 的 Unicode 码点 */
```

Unicode 标准5.2 第4.5节 General Category-Normative 定义了一套字符分组。Go 把分组 Lu、Ll、Lt、Lm、和Lo 作为 Unicode 字母，分组 Nd 作为 Unicode 数字。

字母和数字

下划线 _ (U+005F) 是字母。

```
字母   = unicode_字母 | "_"
十进制数 = "0" ... "9"
八进制数      = "0" ... "7"
十六进制数    = "0" ... "9" | "A" ... "F" | "a" ... "f"
```

词法单位

注解

两种注解格式：

行注解由字符序列 // 开始直到新行。行注解和新行作用一样。

通用注解由字符序列 /* 开始直到字符序列 */ 。跨行的通用注解作为新行，否则是空格。

注解不可嵌套。

令符

令符构成了 Go 的词汇。共有四类： 标识，键字，操作及分隔符，和字面。空白格，包括空格（U+0020)，水平表符（U+0009），回车

（U+000D)，和换行（U+000 A），被忽略除非用于隔开令符而不使其连成一个令符。另外，一个换行可能触发加入一个分号。当把输入断成令符时，下一个令符是可以构成一个合法令符的最长字符序列。

分号

正式语法使用分号 “；” 作为某些产品的终结符。Go 程序可以省略绝大部分的分号，两个规则如下：

1． 输入断成令符时，一个分号自动添加在非空的行尾，如果最后的令符是：

- 一个标识
- 一个整型，浮点型，虚数，字符，或字串字面
- 键字 break, continue, fallthrough, return 之一
- 操作及分隔符 + +，－－，），］，或｝之一

2． 为使复杂语句占据一行，分号可以在结束 “)” 或 “}” 前忽略。

为呼应此地道用法，本文的代码例子使用上面的规则忽略分号。

标识

标识命名程序的实体，例如变量和类型。标识是一个或多个字母数字序列。标识的首字符必须是字母。

```
identifier = letter { letter | unicode_digit } .
```

```
a
_x9
ThisVariableIsExported
αβ
```

某些标识是预先声明的。

键字

下面的键字被保留，不能用作标识：

```
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
```

操作及分隔符

下面字符序列代表　操作符，分隔符，和其它特殊令符：

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=
```

整型字面

整型字面是一个数字序列代表一个整数常量。一个可选的前缀设置非十进制基数：0 是八进制，0x 或 0X 是十六进制。十六进制字面中，字母 a – f 和 A – F 代表值 10 到 15 。

```
int_lit     = decimal_lit | octal_lit | hex_lit .
decimal_lit = ( "1" ... "9" ) { decimal_digit } .
octal_lit   = "0" { octal_digit } .
hex_lit     = "0" ( "x" | "X" ) hex_digit { hex_digit } .

42
0600
0xBadFace
170141183460469231731687303715884105727
```

浮点字面

浮点字面是浮点常量的小数表示。它有一个整数部分，一个小数点，一个分数部分，和一个指数部分。整数和分数部分组成小数数字；指数部分是一个 e 和 E 后跟一个可选的带符号十进制指数。整数和分数之一可以省略；小数点或指数之一也可省略。

```
float_lit = decimals "." [ decimals ] [ exponent ] |
            decimals exponent |
            "." decimals [ exponent ] .
decimals  = decimal_digit { decimal_digit } .
exponent  = ( "e" | "E" ) [ "+" | "-" ] decimals .

0.
72.40
072.40  // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
```

虚数字面

虚数字面是复数常量的虚数部分的小数表示。它包括一个浮点字面或十进制整数，后跟小写的字母 i

```
imaginary_lit = (decimals | float_lit) "i" .

0i
011i  // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
```

字符字面

字符字面代表一个整型常量，通常是一个 Unicode 码点，作为括在单引号的一个或多个字符。引号内可以是任何字符，除了单引号和换行符。单个括起的字符代表其自身，由反斜线开始的多个字符序列以不同的格式编码。

最简单的格式是括起的单个字符；因为 Go 的源码是 UTF-8 编码的 Unicode 字符，多个 UTF-8 编码字节能代表单个整型值。例如，字面 'a' 持有一个字节代表 a，Unicode U+0061，值 0x61，而 'ä' 持有两个字节 (0xc3 0xa4)，代表一个 a 的分音，U+00E4，值0xE4 。

几个反斜线换码序列使任意值都可以用ASCII文字表示。有四种方式可以表示整型值为一个数字常量：\x 后跟两个十六进制数字；\u 后跟四个十六进制数字；\U 后跟八个十六进制数字；和单独 \ 后跟三个八进制数字。每种情况的字面值由其对应基数的数字代表。

尽管这些表示都为整数结果，它们的合法范围不同。八进制换码序列表示一个包含 0 到 255 的值。十六进制换码序列可以架构满足这个条件。换码序列 \u 和 \U 代表 Unicode 码点因此其中的某些值是非法

的，特别是那些高过0x10FFFF 的，以及代理伴（surrogate halves）。

反斜线后，某单个字符换码序列代表特定的值:

```
\a   U+0007 警告或闹铃
\b   U+0008 退格
\f   U+000C 进纸
\n   U+000A 进行或换行
\r   U+000D 回车
\t   U+0009 水平制表
\v   U+000b 纵向制表
\\   U+005c 反斜线
\'   U+0027 单引号  (只在字符字面合法)
\"   U+0022双引号（只在字串字面合法）
```

字符字面其它以反斜线开始时序列都是非法的。

```
char_lit         = "'" ( unicode_value | byte_value ) "'" .
unicode_value    = unicode_char | little_u_value |
big_u_value | escaped_char .
byte_value       = octal_byte_value | hex_byte_value .
octal_byte_value = `\` octal_digit octal_digit octal_digit .
hex_byte_value   = `\` "x" hex_digit hex_digit .
little_u_value   = `\` "u" hex_digit hex_digit hex_digit
hex_digit .
big_u_value      = `\` "U" hex_digit hex_digit hex_digit
hex_digit
                         hex_digit hex_digit hex_digit
hex_digit .
escaped_char     = `\` ( "a" | "b" | "f" | "n" | "r" | "t" |
"v" | `\` | "'" | `"` ) .

'a'
'ä'
'本'
'\t'
```

```
'\000'
'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'
```

字串字面

字串字面代表一个字串常量，由一系列的字符连接得到。有两种形式：原始字串字面和解释字串字面。

原始字串字面是反引号 `` 之间的字符序列。其括起的所有字符都是合法的，除了反引号。此原始字串字面的值是括起的未加解释的字符构成的字串；特别是，反斜线没有特殊意义，并且字串可以跨行。

解释字串字面是双引号 “ ” 之间的字符序列。其括起的文本构成其字面的值，不可以跨行，其反斜线换码序列的解释与在字符字面中相同（除了 \' 非法而 \" 合法）。三字 8 进制（\nnn）和双字 16 进制（\xnn）换码代表结果字串中的独立字节；其它的换码代表（可能多字节的）UTF-8 编码的独立字符。因此字串中字面 \377 和 \xFF 代表单个字节值0xFF=255, 但 ÿ, \u00FF, \U000000FF 和 \xc3\xbf 代表双字节0xc3 0xbf UTF-8编码的字符U+00FF。

```
string_lit             = raw_string_lit | interpreted_s-
tring_lit .
raw_string_lit         = "`" { unicode_char } "`" .
interpreted_string_lit = `"` { unicode_value | byte_value }
`"` .
`abc`  // same as "abc"
`\n
\n`    // same as "\\n\n\\n"
"\n"
```

```
""
"Hello, world!\n"
"日本語"
"\u65e5本\U00008a9e"
"\xff\u00FF"
```

下面的例子代表同样的字串

```
"日本語"                                 // UTF-8 input text
`日本語`                                 // UTF-8 input text as a raw
literal
"\u65e5\u672c\u8a9e"                              // The explicit Uni-
code code points
"\U000065e5\U0000672c\U00008a9e"                  // The explicit Uni-
code code points
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e" // The explicit UTF-8
bytes
```

如果源代码用两个码点表示一个字符，例如一个声调和字符的组合，其结果会是，在字符字面中出错（非单一码点），在字串字面中显示为两个码点。

常量

常量有布尔常量、整型常量、浮点常量、复数常量、和字串常量。整数、浮点和复数常量统称为数字常量。

常量值由一个整数、浮点、虚数、字符、和字串字面代表。一个标识象征常量，一个常量表达式，或由一些内部函数、例如 unsafe.Size 加于任意值，cap 或 len 加于某些表达，real 和img 加于复数常量，及 cmplx 加于数字常量，而得到的结果。布尔真值由预先定义的 true 和 false 常量代表。预先定义的标识 iota 象征一个整型常量。

通常，复杂常量是常量表达式的一种形式，会在那一节讨论。

数字常量代表任意精度的值而且不会溢出。

常量可以有或没有类型。字面常量、true、false、iota、和某些常量表达，如只包含无类型的常量操作则没有类型。

常量可以明确的用常量声明或转换来赋予类型，或隐含在变量声明及赋值、或作为某表达式的操作数里。如果某常量值不能被其对应类型的值精确代表则出错。例如，3.0 可以给任意整型和浮点类型，但 2147483648.0 (等于1<<31)可以给 float32, float64, 或 uint32 但不能是 int32 或 string。

没有常量代表 IEEE-754 的无穷和非数值，但 math 包的 Inf, NaN, IsInf, 和 IsNaN 函数在运行态可以返回及测试这些值。

类型

一个类型决定此类型的值特定的值集及其操作。类型名或类型字面可以指明或限定某类型。类型字面从之前声明的类型组合成新类型。

```
Type      = TypeName | TypeLit | "(" Type ")" .
TypeName  = QualifiedIdent.
TypeLit   = ArrayType | StructType | PointerType | Function-
Type | InterfaceType | SliceType | MapType | ChannelType .
```

布尔，数值和字串类型命名实例是预先定义的。组合类型 – 数组，结构，指针，函数，界面，切片，映射和信道类型 – 可以用类型字面构建。

每个类型 T 都有个底层类型： 如果 T 是预定义类型或类型字面，对应的底层类型是 T 自身。否则，T 的底层类型是 T 在其类型声明时的底层类型。

```
type T1 string
type T2 T1
type T3 []T1
type T4 T3
```

string，T1 和 T2 的底层类型是 string。[]T1,T3 和 T4 的底层类型是 []T1。

类型可以有其关联的方法集合。界面类型的方法集合是它的界面。其它命名类型 T 的方法集合包括所有接受者类型 T 的方法。对应指针类型 *T 的方法集合是接受者为 *T 或 T 的方法的集合（既，它也包括 T 的方法集合）。其它类型的方法集合为空。在方法集合中，每个方法的名称必须唯一。

变量的静态类型（既类型）是其声明定义的类型。界面变量同时还有独特的动态类型，是运行态时变量存储的值的实际类型。动态类型在执行时可以变化，但总是可以赋值给界面变量的静态类型。对于非界面类型，其动态类型总是其静态类型。

布尔型

布尔型代表预定义常量 true 和 false 的布尔真值集合。预定义的布尔型是 bool。

数值型

数值型代表整数或浮点数值的集合。预定义的体系结构无关的数值型是:

```
uint8       8位无符号整数集合(0 to 255)
uint16      16位无符号整数集合 (0 to 65535)
uint32      32位无符号整数集合 (0 to 4294967295)
uint64      64位无符号整数集合 (0 to 18446744073709551615)

int8        8位有符号整数集合 (-128 to 127)
int16       16位有符号整数集合 (-32768 to 32767)
int32       32位有符号整数集合 (-2147483648 to 2147483647)
int64       64位有符号整数集合(-9223372036854775808 to
9223372036854775807)

float32     32位 IEEE-754 浮点数集合
float64     64位 IEEE-754 浮点数集合

complex64  float32 实部和虚部的复数集合
complex128  float64 实部和虚部的复数集合
byte        uint8 的常用名
```

n 位整数值有 n 个字元宽，使用二的补码表示。

还有另一套预定义数值类型是实现指明的大小:

```
uint  32或64位
int   32或64位
float 32或64位
complex 实部和虚部是 float 型
uintptr 足够存放指针类型非解释位的一个无符号整数型
```

为避免移植问题，除了 byte 的所有数值型都是独特的。byte 是 uint8 的代名。不同数值类型在表达式或赋值时要转换。例如，int32 和 int 是不同的类型，尽管在某特定体系结构里它们的尺寸一样。

字串类型

字串类型代表字串值的集合。字串表现的像字符数组但不可变：一旦生成，字串的内容则不能更动。预定义的字串类型是 string。

字串的元素为 byte 类型，可以用普通的下标操作存取。对其元素取地址是非法的。如果 s[i] 是字串的第 i 个字节，&s[i] 非法。使用内置函数 len 可以得到字串 s 的长度。如果 s 是字串字面，其长度是编译态常量。

数组类型

数组是某一类型元素 （称为元素类型）的编号序列。元素的个数为长度，并永不为负。

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

长度是数组类型的一部分，必须是常量表达式求值为一个非负的整数值。使用内置函数 len(a) 可以取得数组 a 的长度。其元素可以用整数下标 0 到 len(a)-1 检索。数组类型是单维的，但可以组合为多维类型。

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

切片类型

切片是对数组某连续片段的引用，并包含此数组元素的编号序列。切片类型代表一元素类型的数组的所有切片的集合。未初始化的切片的值为 nil。

```
SliceType = "[" "]" ElementType .
```

类似数组，切片可检索并有长度。内置函数 len(s) 可以取得切片 s 的长度；不同于数组，它在执行时可能改变。 其元素可以用整数下标 0 到 len(s)-1 检索。切片的某元素的下标可以小于其底层数组同一元素的下标。

切片在初始化后，总是关联于一个持有其元素的底层数组。因此切片与其数组，及其它同样数组的切片共享存储空间；相反的，不同数组代表不同的空间。

切片的底层数组可以延展超过切片尾。容量（capacity）测量其扩展：它是切片长度与超出切片的数组的长度之和。可以从某切片切下一个新的切片使其长度最多达到其容量。使用内置函数 cap(a) 可以取得切片 a 的容量。

使用内置函数 make 可以生成一个元素类型为 T 的新初始化的切片值。make 需要切片类型，其长度，和可选的容量：

```
make([]T, length)
```

```
make([]T, length, capacity)
```

make() 调用分配一个新的隐藏的数组并返回其指向的切片值。既，执行：

```
make([]T, length, capacity)
```

得到与分配数组并切片同样的切片，因此下两个例子得到同样的切片：

```
make([]int, 50, 100)
new([100]int)[0:50]
```

类似数组，切片是单维的，但可组合成高维的物件。对于数组的数组，其内层的数组，通过构建，总是同样长度；但对于切片的切片（或切片的数组），长度可动态改变。另外，内层切片必须单独分配（使用make）。

结构类型

结构类型是一系列的带名元素，称为域，每个都有名称和类型。域名可以明确指出（IdentifierList）或隐含（AnonymousField）。结构内，非空的域名必须唯一。

```
StructType     = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl      = (IdentifierList Type | AnonymousField) [
Tag ] .
AnonymousField = [ "*" ] TypeName .
Tag            = string_lit .

// An empty struct.
struct {}

// A struct with 6 fields.
```

```
struct {
      x, y int
      u float
      _ float  // padding
      A *[]int
      F func()
}
```

声明类型但没有明确域名的域是匿名域。此域类型必须指明为类型名 T 或类型名的指针 *T，并且 T 自身不可以是指针类型。此未限定的类型名被作为域名。

```
// A struct with four anonymous fields of type T1, *T2, P.T3
and *P.T4
struct {
      T1        // field name is T1
      *T2       // field name is T2
      P.T3      // field name is T3
      *P.T4     // field name is T4
      x, y int  // field names are x and y
}
```

下面的声明非法，因为域名在结构类型内必须唯一：

```
struct {
      T         // conflicts with anonymous field *T and
*P.T
      *T        // conflicts with anonymous field T and *P.T
      *P.T      // conflicts with anonymous field T and *T
}
```

匿名域的域及其方法被提升为结构的普通域及方法。下面的规则适用与结构类型 S 和一个类型 T：

* 如果 S 包括一匿名域 T，则 S 的方法集合包括 T 的方法集合

* 如果 S 包括一匿名域 *T，则 S 的方法集合包括 *T 的方法集合（也包括 T 的方法集合）

* 如果 S 包括一匿名域 T 或 *T，则 *S 的方法集合包括 *T 的方法集合（也包括 T 的方法集合）

域的声明可以后跟一个可选的字串字面标签，其成为对应域声明的所有域的属性。标签可以从反思界面（reflect interface）看到，否则被忽略。

```
// A struct corresponding to the TimeStamp protocol buffer.
// The tag strings define the protocol buffer field numbers.
struct {
      microsec  uint64 "field 1"
      serverIP6 uint64 "field 2"
      process   string "field 3"
}
```

指针类型

指针类型代表某类型（称为指针的基本类型）的变量的所有指针的集合。未初始化的指针值为 nil。

```
PointerType = "*" BaseType .
BaseType = Type .
*int
*map[string] *chan int
Function types
```

函数类型

函数类型代表同样参量和结果类型的所有函数的集合。 未初始化的函数类型的变量为 nil。

```
FunctionType   = "func" Signature .
Signature      = Parameters [ Result ] .
Result         = Parameters | Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = [ IdentifierList ] [ "..." ] Type .
```

在参量或结果列内，名称（IdentifierList）必须全部存在或全不存在。如果存在，每个名字代表一个指定类型的项（参量或结果）；如不存在，每个类型代表此类型的一项。参量和结果列必须加小括号，除非只是一个未命名的结果可写为不带括号的类型。

函数签名的最后一个参量可以在类型前加 ... 。带此参量的函数称为可变型，此参量可以用零或多个参数调用。

```
func()
func(x int)
func() int
func(prefix string, values ...int)
func(a, b int, z float) bool
func(a, b int, z float) (bool)
func(a, b int, z float, opt ...interface{}) (success bool)
func(int, int, float) (float, *[]int)
func(n int) func(p *T)
```

界面类型

界面类型指明一个方法集合，称为其界面。界面类型的变量可以存储任意是此界面方法集合的超集的类型的值。此类型称为实现了此界面。未初始化的界面类型的变量为 nil。

```
InterfaceType     = "interface" "{" { MethodSpec ";" } "}" .
MethodSpec      = MethodName Signature | InterfaceTypeName .
```

```
MethodName          = identifier .
InterfaceTypeName   = TypeName .
```

正如所有的方法集合，界面类型内每个方法名必须唯一。

```
// A simple File interface
interface {
      Read(b Buffer) bool
      Write(b Buffer) bool
      Close()
}
```

多个类型可以实现一个界面。例如，如果类型 S1 和 S2 有方法集合

```
func (p T) Read(b Buffer) bool { return ... }
func (p T) Write(b Buffer) bool { return ... }
func (p T) Close() { ... }
```

（T 代表 S1 或 S2）则 File 界面被 S1 和 S2 实现，而无论 S1 和 S2 是否有或共享其它的方法。

一个类型实现了其任意方法子集所组成的界面，因此可以实现几个不同的界面。例如，所有类型都实现空界面： interface{}

类似的，考虑下面的界面规范，在一个类型声明内定义称为 Lock 的界面：

```
type Lock interface {
      Lock()
      Unlock()
}
```

如果 S1 和 S2 也实现

```
func (p T) Lock() { ... }
func (p T) Unlock() { ... }
```

它们同时实现了 Lock 界面和 File 界面。

界面可以在方法规范的位置包含一个界面类型名 T。效果等价于在界面内明确的列举 T 的方法：

```
type ReadWrite interface {
      Read(b Buffer) bool
      Write(b Buffer) bool
}

type File interface {
      ReadWrite  // 等同列举 ReadWrite 中的方法
      Lock       // 等同列举 Lock 中的方法
      Close()
}
```

映射类型

映射是一组无序的某类元素，称作元素类型，索引自另一类型的键，称为键类型。未初始化的映射值为 nil。

```
MapType     = "map" "[" KeyType "]" ElementType .
KeyType     = Type .
```

比较操作符 == 和 != 对于键类型的操作数必须完全定义。因此键类型不可以是结构，数组或切片。如果键类型是界面类型，这两个比较操作符对动态的键值必须定义；否则会产生运行态 panic。

```
map [string] int
map [*T] struct { x, y float }
map [string] interface {}
```

映射元素的个数称作其长度。对映射 m，内置函数 len(m) 可用来发现其长度，其在执行时可能改变。使用特殊形式的赋值，可在运行时添加或删除值。

内置函数 make 可以生成新的空映射值，它使用映射的类型和可选的容量作为参数：

```
make(map[string] int)
make(map[string] int, 100)
```

初始容量不会限制其大小：映射会增大来容纳其存储的项。

信道类型

信道提供了一种机制，使两个同时执行的函数可以同步执行并通过传递一个指定元素类型的值来通讯。未初始化的信道的值为 nil。

```
ChannelType = ("chan" [ "<-" ] | "<-" "chan") ElementType .
```

操作符 <- 指明信道的方向，发送或接收。如果未给定方向，信道是双向的。信道可通过转换或赋值限定只能发送或接收。

```
chan T          // 可发送和接收 T 类型的值
chan<- float    // 只可发送浮点数
<-chan int      // 只可接收整数
```

操作符 <- 和最左侧的 chan 关联

```
chan<- chan int     // same as chan<- (chan int)
chan<- <-chan int   // same as chan<- (<-chan int)
<-chan <-chan int   // same as <-chan (<-chan int)
chan (<-chan int)
```

内置函数 make 可以生成新的初始过的信道值，它使用信道类型和可选的容量作为参数：

```
make(chan int, 100)
```

容量，既元素的个数，设定了信道缓冲的大小。如果容量大于零，信道是异步的：只要缓存未满，发送成功而不会阻塞。如果容量为零或省略，通讯只在发送者和接收者都准备好时才成功。

信道可以使用内置函数 close 关闭和用 closed 测试关闭。

类型和值的属性

类型同质

两个类型只能相同或不同。

两个命名类型相同只当它们的类型名源自同一个类型声明。一个命名的和无名的类型是不同的。两个无名类型相同只当它们对应的类型字面相同，既，只当它们有同样的字面结构及对应组件是相同类型。具体是：

* 两数组类型相同只当它们有同样的元素类型且数组长度相等

* 两切片类型相同只当它们有同样的元素类型

* 两结构类型相同只当它们有同样顺序的域，且对应的域的名称，类型和标签相同。两匿名域认为名字相同。不同包的小写的域名永远不同。

* 两指针类型相同只当它们的基类型相同。

* 两函数类型相同只当它们有相同数量的参量和返回值，且对应的参数和返回类型相同，且两函数皆属变参量或皆不是。参量和返回的名称不要求相同。

* 两界面类型相同只当它们有同名方法和相同函数类型的集合。 不同包的小写的方法名永远不同。方法的顺序不重要。

* 两映射类型相同只当它们有相同的键和值类型。

* 两信道类型相同只当它们有相同的值类型且方向相同。

给出下面的声明：

```
type (
      T0 []string
      T1 []string
      T2 struct { a, b int }
      T3 struct { a, c int }
      T4 func(int, float) *T0
      T5 func(x int, y float) *[]string
)
```

下面的类型相同：

```
T0 和 T0
[]int 和 []int
struct { a, b *T5 } 和 struct { a, b *T5 }
func(x int, y float) *[]string 和 func(int, float) (result
*[]string)
```

T0 和 T1 不同是因为它们用不同声明命名；func(int, float) *T0 和 func (x int, y float)*[]string 不同是因为 T0 不同于 []string。

可赋值性

值 x 可赋值给 T 类型的变量（x 可赋值 T）发生在以下情况：

* x 的类型和 T 相同

* x 类型 V 和 T 有相同的底层类型且 V 或 T 是无名类型

* T 是界面类型且 x 实现 T

* x 是双向信道值，T 是信道类型，x 的类型 V 和 T 有相同的元素类型，且 V 或 T 是无名类型

* x 是预定义描述符 nil ，T 是指针，函数，切片，映射，信道或界面类型

* x 是无类型的常量，可由类型 T 的值表示

如果 T 是结构类型，则 T 的全部域必须导出，或在 T 声明的同一个包内赋值。即是，一个结构值能赋值给一个结构变量只当结构的每一个域都能单独被程序合法赋值。

任意值都可赋值给 “空描述符”

块

块是匹配大括号内的一系列声明和语句。

```
Block = "{" { Statement ";" } "}" .
```

除了源代码里的直接块，还有隐含块：

1，宇宙块包括所有 Go 源代码

2，每个包都有“包块”包括此包的所有 Go 源代码

3， 每个文件都有“文件块”包括此文件的所有 Go 源代码

4，每个 if, for 和 switch 语句被认为有自己的隐含块

5，switch 和 select 语句的每个条款作为一个隐形块

块嵌套和影响作用域。

声明和作用域

声明连接一个非空描述符和一个常量，类型，变量，函数或包。程序的每个描述符必须声明。同个块内没有描述符可定义两次，且没有描述符可同时定义在文件和包块内。

```
Declaration   = ConstDecl | TypeDecl | VarDecl .
TopLevelDecl  = Declaration | FunctionDecl | MethodDecl .
```

声明的描述符的作用域是源代码里描述符指定的常量，类型，变量，函数或包的范围。

Go 词法上使用块作用域:

1，预定义描述符的作用域是宇宙块

2，顶层声明（所有函数外）描述常量，类型，变量或函数的描述符的作用域是包块

3，引入包的描述符的作用域是包括引入声明的文件的文件块

4，函数参量和结果变量的描述符的作用域是函数体

5，函数内声明的常量或变量描述符的作用域开始于 ConstSpec 或 VarSpec 后，结束于最内层的包含块

6，函数内声明的类型描述符的作用域开始于 TypeSpec，结束于最内层的包含块

块内声明的描述符可以在内层块内重新声明。当内层声明的描述符出现在作用域时，它代表的是内层声明的实体。

package 条目不是声明；包名不出现在任何作用域。它的功能是指明文件属于同一个包，并指定引入声明的默认名。

标签（label）作用域

标签由标签语句声明，用于 break，continue 和 goto 语句。和其它描述符不同，标签不是块作用域，也不和非标签的描述符冲突。标签作用域是它声明的函数体，且不包括嵌套的函数体。

预定义描述符

下面的描述符隐含的声明在宇宙块：

基本类型：

```
bool byte complex64 complex128 float32 float64
int8 int16 int32 int64 string uint8 uint16 uint32 uint64
```

体系结构相关方便类型：

```
      complex float int uint uintptr
```

常量：

```
      true false iota
```

零值：

```
nil
```

函数：

```
cap close closed cmplx copy imag len make
new panic print println real recover
```

导出描述符

一个描述符可以导出供其它包用限定描述符存取。描述符可被导出如果：

1，描述符名的首字符是 Unicode 大写字母 （Unicode “Lu” 类），且

2，描述符声明在包块，或代表声明在包块的一个类型的域或方法

所有其它的描述符都不能导出。

空白描述符

空白描述符，由下划线 _ 代表，可像其它描述符一样用在声明里，但此声明不会引入新连接（binding）。

常量声明

常量声明连结一列描述符

词汇表

B

★包（package）

★布尔（boolean）

★并发（concurrent）

★标识（identifier）：

C

★参量（parameter 或 argument）：函数引入或返回的变量

D

★导入（import）：

★导出（export）：

★多维函数（variadic function）：可带任意数量的参量的函数。

★大小混写（CamelCase）：

F

★方法（method）：成员函数

★分配（allocation）：

★服务者（server）：

★分句 （clause）：

★反思（reflection）：

G

★够程（goroutine）：Go 的并行单元。每个 Go 程序都是一个独立的够程，其执行时可用 go 语句启动任意函数作为新的独立够程并发执行。

H

★互斥（mutex）：

★函数（function）：

★函数调用（function call）

★组合字面（composite literal）

★缓冲（buffer）：

★回生（recover）：

J

★结构（struct）：

★界面（interface）：

★接受者（receiver）：

★键（key）：

★键值伴（key-value pair）：

★架构函数（contructor）：

★经手者（handler)：

★记账 （bookkeeping）：

★键字（keyword）：

K

★库（library）：

★空白标识（blank identifier）

L

★漏洞（bug）：软件隐藏的缺陷。查漏（debug）可以指出程序有错，不能证明其正确。

★类型（type）：

★类型切换（type switch）：

★列举（enumerate）

★漏水缓冲（leaky buffer）：用缓冲来调节进出的速率，常用在流媒体上。

★零维（niladic）：不带参量

★令符（token）

M

★码点（accented code point）：

N

★内置（embedding）：

P

★排版（formatting，format）

★怕死（panic）：

Q

★签名（signature）：

★切片（slice）：

★旗标（flag）：

S

★声明（declaration）：

★实现（implement）：

★实例（instance）：

★收集 （collection）：

T

★退绕（unwinding）：

W

★物件（object）：也叫对象

★网舍（web server）：

X

★习语（idiom）：习惯用法

★信号灯（semaphore）：

★新行（newline）：

★信道（channel）：够程之间的收发数据结构。协调并发和传递值。

Y

★域（field）：结构（struct）的成员

★引用（reference）：相对于值（value）类型

★映射（map）：

★延迟函数（deferred function）：

Z

★字串（string）

★字面 （literal）：

★阻塞（block）：