Go 语言入门教程

- 译者
- 简介
- Hello，世界
- 分号（Semicolons）
- 编译
- Echo
- 类型简介
- 申请内存
- 常量
- I/O 包
- Rotting cats
- Sorting
- 打印输出
- 生成素数
- Multiplexing

译者

原文：http://golang.org/install.html
翻译：柴树杉 (chaishushan@gmail.com)
翻译：Bian Jiang (borderj@gmail.com)

简介

本文是关于 Go 编程语言的基础教程，主要面向有 C/C++基础的读者。它并不是一个语言的完整指南，关于 Go 的具体细节请参考 语言规范 一文。在读完这个入门教程后， 深入的华可以继续看 Effective Go ，这个文档 将涉及到 Go 语言的更多特性。此外，还有一个《Go 语言三日教程》系列讲座： 第一日, 第二日, 第三日。

下面将通过一些小程序来演示 go 语言的一些关键特性。所有的演示程序都是可以运行的，程序的代码在安装目录的 "/doc/progs/"子目录中。

文中的代码都会标出在源代码文件中对应的行号。同时为了清晰起见，我们忽略了源代码文件空白行的行号。

Hello，世界

让我们从经典的"Hello, World"程序开始：

```
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 源文件开头都有一个"package"声明语句，指明源文件所在的包。同时，我们也可以根据具体的需要 来选择导入（import 语句）特定功能的包。在这个例子中，我们通过导入“fmt”包来使用我们熟悉的 printf 函数。 不过在 Go 语言中，Printf 函数的是大写字母开头，并且以 fmt 包名作为前缀：fmt.Printf。

关键字“func“用于定义函数。在所有初始化完成后，程序从 main 包中的 main 函数开始执行。

常量字符串可以包含 Unicode 字符，采用 UTF-8 编码。实际上，所有的 Go 语言源文件都采用 UTF-8 编码。

代码注释的方式和 C++类似：

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

```
        // ...
```

稍后，我们还有很多的关于打印的话题。

分号（Semicolons）

比较细心的读者可能发现前面的代码中基本没有出现分号“;”。其实在 go 语言中，只有在分隔 for 循环的初始化语句时才经常用到；但是代码段末尾的分号一般都是省略的。

当然，你也可以像 C 或 JAVA 中那样使用分号。不过在大多数情况下，一个完整语句末尾的分号 都是有 go 编译器自动添加的——用户不需要输入每个分号。

关于分号的详细描述，可以查看 Go 语言说明文档。不过在实际写代码时，只需要记得一行末尾的分号 可以省略就可以了（对于一行写多个语句的，可以用分号隔开）。还有一个额外的好处是：在退出 大括号包围的子区域时，分号也是可以省略的。

在一些特殊情况下，甚至可以写出没有任何分号的代码。不过有一个重要的地方：对于"if"等 后面有大括弧的语句，需要将左大括弧放在"if"语句的同一行，如果不这样的话可能出现编译错误。 Go 语言强制使用将开始大括弧放在同一行末尾的编码风格。

编译

Go 是一个编译型的语言。目前有两种编译器，其中"Gccgo"采用 GCC 作为编译后端。另外还有 根据处理器架构命名的编译器：针对 64 位 x86 结构为"6g"，针对 32 位 x86 结构的为"8g"等等。 这些 go 专用的编译器编译很快，但是产生的目标代码效率比 gccgo 稍差一点。目前（2009 年底）， go 专用的编译器的运行时系统比"gccgo"要相对健壮一点。

下面看看如何编译并运行程序。先是用针对 64 位 x86 结构处理器的“6g”：

```
        $ 6g helloworld.go  # 编译; 输出 helloworld.6
        $ 6l helloworld.6   # 链接; 输出 6.out
        $ 6.out
        Hello, world; or Καλημέρα κόσμε; or こんにちは 世界
        $
```

如果是用 gccgo 编译，方法和传统的 gcc 编译方法类似：

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

Echo

下面的例子是 Unix 系统中"echo"命令的简单实现：

```
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  }
```

程序虽然很小，但是包含了 go 语言的更多特性。在上一个的例子中，我们演示了如何用"func"关键字定义函数。类似的关键字还有："var"、"const"和"type"等，它们可以用于定义变量、常量和类型等，用法和"import"一致。我们可以小括弧声明一组类型相同的变量（如 7－10 和 14－17 行所示）。当然，也可以分开独立定义：

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

程序首先导入 os 包，因为后面要用到包中的一个 *os.File 类型的 Stdout 变量。 这里的 import 语句实际上是一个声明，和我们在 hello world 程序中所使用方法一样，包的名字标识符（fmt）为前缀用于定位包中定位包中的成员，包可以是在当前目录或标准包目录。在导入包的时候一般会默认选用包本身的 名字（在必要的时候可以将导入的包重新命名）。在“hello world”程序中，我们只是简单的 import "fmt" 。

如果需要，你可以自己重新命名被 import 的包。但那不是必须的，只在处理包名字冲突的时候会用到。

通过"os.Stdout"，我们可以用包中的"WriteString"方法来输出字符串。

现在已经导入"flag"包，并且在 12 行创建了一个全局变量，用于保存 echo 的"-n"命令行选项。变量"omitNewline"为一个只想 bool 变量的 bool 型指针。

在"main.main"中，我们首先解析命令行参数（20 行），然后创建了一个局部字符串变量用于保存要输出的内容。

变量声明语法如下：

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

这里有一个"var"关键字，后面跟着变量名字和变量的数据类型，再后面可以用“ = ”符号来进行赋初值。

简洁是 go 的一个目标，变量的定义也有更简略的语法。go 可以根据初始值来判断变量的类型， 没有必要显式写出数据类型。也可以这样定义变量：

```
        var s = "";
```

还有更短的写法：

```
        s := "";
```

操作符":="将在 Go 中声明同时进行初始化一个变量时会经常使用。下面的代码是在"for"中声明并 初始化变量：

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

"flag"包会解析命令行参数，并将不是 flag 选项的参数保存到一个列表中。可以通过 flag 的参数列表 访问普通的命令行参数。

Go 语言的"for"语句和 C 语言中有几个不同的地方：第一，for 是 Go 中唯一的循环语句，Go 中没有 while 或 do 语句；第二，for 的条件语句并不需要用小括号包起来，但是循环体却必须要花括弧，这个规则同样适用于 if 和 switch。后面我们会看到 for 的一些例子。

在循环体中，通过"+="操作符向字符串"s"添加要命令行参数和空白。在循环结束后，根据命令行是否有"-n"选项， 判断末尾是否要添加换行符。最后输出结果。

值得注意的地方是"main.main"函数并没有返回值（函数被定义为没有返回值的类型）。如果"main.main" 运行到了末尾，就表示“成功”。如果想返回一个出错信息，可用系统调用强制退出：

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

"os"包还包含了其它的许多启动相关的功能，例如"os.Args"是"flag"包的一部分（用来获取命令行输入）。

类型简介

Go 语言中有一些通用的类型，例如"int"和"float"，它们对应的内存大小和处理器类型相关。同时， 也包含了许多固定大小的类型，例如"int8"和"float64"，还有无符号类型"uint"和"uint32"等。 需要注意的是，即使"int"和"int32"占有同样的内存大小，但并不是同一种数据类型。不过 "byte"和"uint8"对应是相同的数据类型，它们是字符串中字符类型。

go 中的字符串是一个内建数据类型。字符串虽然是字符序列，但并不是一个字符数组。可以创建新的 字符串，但是不能改变字符串。不过我们可以通过新的字符串来达到想改变字符串的目的。 下面列举"strings.go"例子说明字符串的常见用法：

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

不管如何，试图修改字符串的做法都是被禁止的：

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

Go 中的字符串和 C++中的"const strings"概念类似，字符串指针则相当于 C++中的"const strings" 引用。

是的，它们都是指针，但是 Go 中用法更简单一些。

数组的声明如下：

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

数组和字符串一样也是一个值对象，不过数组的元素是可以修改的。不同于 C 语言的是："int"类型数组 "arrayOfInt"并不能转化为"int"指针。因为，在 Go 语言中数组是一个值对象，它在内部保存"int"指针。

数组的大小是数组类型的一部分。我们还可以通过 *slice*（切片）类型的变量来访问数组。 首先，数据元素的类型要和 *slice*（切片）类型相同，然后通过"a: high"类似的 语法来关联数组的 low 到 heigh-1 的子区间元素。 Slices 和数组的声明语法类似，但是不像数组那样 要指定元素的个数（""和"10"的区别）；它在内部引用特定的空间，或者其它数组的空间。 如果多个 Slices 引用同一个数组，则可以共享数组的空间。但是不同数组之间是无法共享内存空间的。

在 Go 语言中 Slices 比数组使用的更为普遍，因为它更有弹性，引用的语法也使得它效率很高。 但是，Slices 缺少对内存的绝对控制比数组要差一些。例如你只是想要一个可以存放 100 个元素 的空间，那么你就可以选择数组了。创建数组：

```
 [3]int{1,2,3}
```

上面的语句创建一个含有 3 个元素的 int 数组。

当需要传递一个数组给函数时，你应该将函数的参数定义为一个 Slice。这样，在调用函数的时候， 数组将被自动转换为 slice 传入。

比如以下函数以 slices 类型为参数（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  }
```

函数的返回值类型(int)在 sum()函数的参数列表后面定义。

为了调用 sum 函数，我们需要一个 slice 作为参数。我们先创建一个数组，然后将数组转为 slice 类型：

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

如果你创建一个初始化的数组，你可以让编译器自动计算数组的元素数目，只要在数组大小中填写"..."就 可以了：

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

是实际编码中，如果不关心内存的具体细节，可以用 slice 类型（省略数组的大小）来代替数组地址为函数参数：

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

还有 map 类型，可以用以下代码初始化：

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

用内建的"len()"函数，可以获取 map 中元素的数目，该函数在前面的"sum"中用到过。"len()"函数 还可以用在 strings, arrays, slices, maps, 和 channels 中。

还有另外的"range"语法可以用到 strings, arrays, slices, maps, 和 channels 中， 它可以用于"for"循环的迭代。例如以下代码

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

用"range"语法可以写成：

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

这里的"i"对应元素的索引，"v"对应元素的值。关于更多的细节可以参考 Effective Go。

申请内存

在 Go 语言中，大部分的类型都是值变量。例如 int 或 struct(结构体)或 array(数组)类型变量， 赋值的时候都是复制整个元素。如果需要为一个值类型的变量分配空间，可以用 new()：

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

或者更简洁的写法：

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

还有另外一些类型，如：maps, slices 和 channels(见下面)是引用语意（reference semantics）。 如果你一个 slice 或 map 内的元素，那么其他引用了相同 slice 或 map 的变量也能看到这个改变。 对于这三类引用类型的变量，需要用另一个内建的 make()分配并初始化空间：

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

上目的代码定义一个新的 map 并分配了存储空间。如果只是定一个 map 而不想分配空间的话，可以这样：

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

它创建了一个 nil(空的)引用并且没有分配存储空间。如果你想用这个 map, 你必须使用 make 来 分配并初始化内存空间或者指向一个已经有存储空间的 map。

注意: new(T) 返回的类型是 *T , 而 make(T) 返回的是引用语意的 T 。如果你(错误的)使用 new()` 分配了一个引用对象，你将会得到一个指向 nil 引用的指针。这个相当于声明了一个未初始化引用变量并取得 它的地址。

常量

虽然在 Go 中整数(integer)占用了大量的空间，但是常量类型的整数并没有占用很多空间。 这里没有像 0LL 或 0x0UL 的常量，取而代之的是使用整数常量作为大型高精度的值。常量只有 在最终被赋值给一个变量的时候才可以会出现溢出的情况：

```
const hardEight = (1 &lt;&lt; 100) &gt;&gt; 97  // legal，合法
```

具体的语法细节比较琐屑，下面是一些简单的例子：

```
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
```

（强制？）转换只适用于几种简单的情况：转换整数(int)到去其他的精度和大小，整数(int)与 浮点数(float)的转换, 还有其他一些简单情形。在 Go 语言中，系统不会对两种不同类型变量作 任何隐式的类型转换。此外，由常数初始化的变量需要指定确定的类型和大小。

I/O 包

接下来我们使用 open/close/read/write 等基本的系统调用实现一个用于文件 IO 的包。 让我们从文件 file.go 开始：

```
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  }
```

文件的第一行声明当前代码对应—"file"—包，然后导入 os 和 syscall 两个包。 包 os 封装了不同操作系统底层的实现，例如将文件抽象成相同的类型。我们将在系统接口基础 上封装一个基本的文件 IO 接口。

另外还有其他一些比较底层的 syscall 包，它提供一些底层的系统调用(system's calls)。

接下来是一个类型(type)定义：用"type"这个关键字来声明一个类。在这个例子里数据结构(data structure) 名为"File"。为了让这事变的有趣些，我们的 File 包含了一个这个文件的名字(name)用来描述这个文件。

因为结构体名字"File"的首字母是大写，所以这个类型包(package)可以被外部访问。在 GO 中访问规则的处理 是非常简单的：如果顶极类型名字首字母(包括：function, method, constant or variable, or of a structure field or method)是大写，那么引用了这个包(package)的使用者就可以访问到它。不然 名称和被命名的东西将只能有 package 内部看到。这是一个要严格遵循的规则，因为这个访问规则是由 编译器(compiler)强制规范的。在 GO 中，一组公开可见的名称是"exported"。

在这个 File 例子中，所有的字段(fields)都是小写所以从包外部是不能访问的，不过我们在下面将会一个 一个对外访问的出口(exported) —— 一个以大写字母开头的方法。

首先是一个创建 File 结构体的函数：

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

这将返回一个指向新 File 结构体的指针，结构体存有文件描述符和文件名。这段代码使用了 GO 的"复合变量"(composite literal)的概念，和创建内建的 maps 和 arrays 类型变量一样。要创建在堆（heap-allocated）中创建一个新的 对象，我们可以这样写：

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

如果结构比较简单的话，我们可以直接在返回结构体变量地址的时候初始化成员字段，如前面例子的 21 行代码所示。

我们可以用前面的函数（newFile）构造一些 File 类型的变量，返回 File：

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

这里的 newFile 是内部函数，真正包外部可以访问的函数是 Open：

```
30  func Open(name string, mode int, perm uint32) (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  }
```

在这几行里出现了一些新的东西。首先，函数 Open 返回多个值(multi-value)：一个 File 指针和一个 error(等一下会介绍 errors)》我们用括号来表来声明返回多个变量值(multi-value)，语法上它看 起来像第二个参数列表。syscall.Open 系统调用同样也是返回多个值 multi-value。接着我们能在 31 行 创建了 r 和 e 两个变量用于保存 syscall.Open 的返回值。函数最终也是返回 2 个值，分别为 File 指针和一个 error。 如果 syscall.Open 打开失败，文件描述 r 将会是个负值，newFile 将会返回 nil。

关于错误：os 包包含了一些常见的错误类型。在用户自己的代码中也尽量使用这些通用的错误。 在 Open 函数中，我们用 os.Error 函数将 Unix 的整数错误代码转换为 go 语言的错误类型。

现在我们可以创建 Files,我们为它定义了一些常用的方法(methods)。要给一个类型定义一个方法(method)， 需要在函数名前增加一个用于访问当前类型的变量。这些是为*File 类型创建的一些方法：

```
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  }
```

这些并没有隐含的 this 指针（参考 C + + 类），而且类型的方法(methods)也不是定义在 struct 内部——struct 结构 只声明数据成员(data members)。事实上，我们可以给任意数据类型定义方法，例如：整数(integer)，数组(array) 等。后面我们会有一个给数组定义方法的例子。

String 这个方法之所以会被调用是为了更好的打印信息，我们稍后会详细说明。

方法(methods)使用 os.EINVAL 来表示(os.Error 的版本)Unix 错误代码 EINVAL。 在 os 包中针对标准的 error 变量定义各种错误常量。

现在我们可以使用我们自己创建的包(package)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  }
```

--PROG progs/helloworld3.go /package/ END

这个""./""在导入(import)""./file""时告诉编译器(compiler)使用我们自己的 package，而不是在 默认的 package 路径中找。

最后，我们来执行这个程序：

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

Rotting cats

在我们上面创建的 file 包(package)基础之上，实现一个简单的 Unix 工具 "cat(1)", "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  }
```

现在应该很容易被理解，但是还有些新的语法"switch". 比如: 包括了"for"循环， "if"和 "switch"初始化的语句。在"switch"语句的 18 行用了"f.Read()"函数的返回值"nr"和"er"做为 变量(25 行中的"if"也采用同样的方法)。这里的"switch"语法和其他语言语法基本相同，每个分支(cases) 从上到下查找是否与相关的表达式相同，分支(case)的表达式不仅仅是常量(constants)或整数(integers), 它可以是你想到的任意类型。

这个"switch"的值永远是"真(true)", 我们会一直执行它, 就像"for"语句，不写值默认是"真"(true). 事实上，"switch"是从"if-else"由来的。在这里我们要说明, "switch"语句中的每个"分支"(case)都 默认隐藏了"break".

在 25 行中调用"Write()"采用了 slicing 来取得 buffer 数据. 在标准的 GO 中提供了 Slices 对 I/O buffers 的操作。

现在让我们做一个"cat"的升级版让"rot13"来处理输入, 就是个简单的字符处理，但是要 采用 GO 的新特性"接口 *(interface)*"来实现。

这个"cat()"使用了 2 个子程序"f":"Read()"和"String", 让我们定义这 2 个接口， 源码参考 "progs/cat_rot13.go"

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

任何类型的方法都有 reader 这两个方法 —— 也就是说实现了这两个方法， 任何类型的方法都能使用。由于 file.File 实现了 reader 接口，我们就可以让 cat 的子程序访问 reader 从而取代了 *file.File 并且能正常工作，让我们来些第二个类型实现 reader , 一个关注现有的 reader ，另一个 rot13 只关注数据。我们只是定义了这个类型和 实现了这个方法并没有做其他的内部处理, 我们实现了第二个 reader 接口.

```
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"函数非常简单，没有必要在这里进行讨论)

为了使用新的特性，我们定义了一个标记(flag):

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

用它基本上不需要修改"cat()"这个函数：

```
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  }
```

(我们应该对 main 和 cat 单独做些封装，不仅仅是对类型参数的修改，就当是练习)从 56 行到 58 行: 如果 rot13 标记是真，封装的 reader 就会接受数据并传给 rotate13 并处理. 注意: 这个接口的值是变量，不是指针，这个参数是 reader 类型，不是 *reader , 尽管后面转换为 指向结构体的指针。

这里是执行结果:

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

也许你会说使用注入依赖(dependency injection)能轻松的让接口以一个文件描述符执行。

接口(interfaces)是 Go 的一个特性，一个接口是由类型实现的，接口就是声明该类型的所有方法。 也就是说一个类型可以实现多个不同的接口, 没有任何类型的限制，就像我们的例子"rot13". "file.File"这个类型实现了"reader", 它也能实现"writer", 或通过其他的方法来实现这个接口。 参考空接口*(empty interface)*

```
type Empty interface {}
```

任何类型都默认实现了空接口，我们可以用空接口来保存任意类型。

Sorting

接口(interfaces)提供了一个简单形式的多态(polymorphism). 他们把对象的定义和 如何实现的分开处理，允许相同的接口可以有不能的实现方法。

参考这个简单的排序算法(sort algorithm)"progs/sort.go"

```
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  }
```

--PROG progs/sort.go /func.Sort/ /^}/

我们要封装这个排序(sort)的接口(interface)仅需要三个方法。

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

我们可以用任何类型的"Sort"去实现"Len", "Less" 和 "Swap". 这个"sort"包里面 包含一些方法(methods). 下面是整型数组的代码:

```
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] }
```

你看到的是一个没有任何类型的"结构体"(non-struct type). 在你的包里面你可以定义 任何你想定义的类型.

现在用"progs/sortmain.go"程序进行测试，用"sort"包里面的排序函数进行排序。

```
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  }
```

如果我们为 sort 提供一个新类型，我们就需要为这个类型实现三个方法，如下:

```
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] }
```

打印输出

前面例子中涉及到的打印都比较简单。在这一节中，我们将要讨论 Go 语言格式化输出的功能。

我们已经用过"fmt"包中的"Printf"和"Fprintf"等输出函数。"fmt"包中的"Printf"函数的 完整说明如下：

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

其中"..."表示数目可变参数，和 C 语言中"stdarg.h"中的宏类似。不过 Go 中，可变参数是通道 一个空接口（"interface {}"）和反射（reflection）库实现的。反射特性可以帮助"Printf" 函数很好的获取参数的详细特征。

在 C 语言中，printf 函数的要格式化的参数类型必须和格式化字符串中的标志一致。不过在 Go 语言中， 这些细节都被简化了。我们不再需要"%llud"之类的标志，只用"%d"表示要输出一个整数。至于对应 参数的实际类型，"Printf"可以通过反射获取。例如：

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

输出

```
18446744073709551615 -1
```

最简单的方法是用"%v"标志，它可以以适当的格式输出任意的类型（包括数组和结构）。下面的程序，

```
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)
```

将输出：

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

如果是使用"Print"或"Println"函数的话，甚至不需要格式化字符串。这些函数会针对数据类型 自动作转换。"Print"函数默认将每个参数以"%v"格式输出，"Println"函数则是在"Print"函数 的输出基础上增加一个换行。一下两种输出方式和前面的输出结果是一致的。

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

如果要用"Printf"或"Print"函数输出似有的结构类型，之需要为该结构实现一个"String()"方法， 返回相应的字符串就可以了。打印函数会先检测该类型是否实现了"String()"方法，如果实现了则以 该方法返回字符串作为输出。下面是一个简单的例子。

```
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 类型有 String() 方法，因此格式化函数用它作为输出结果：

```
        77 Sunset Strip
```

前面的例子中，"String()"方法用到了"Sprint"（从字面意思可以猜测函数将返回一个字符串）作为格式化的基础函数。在 Go 中，我们可以递归使用"fmt"库中的函数来为格式化服务。

"Printf"函数的另一种输出是"%T"格式，它输出的内容更加详细，可以作为调试信息用。

自己实现一个功能完备，可以输出各种格式和精度的函数是可能的。不过这不是该教程的重点，大家 可以把它当作一个课后练习。

读者可能有疑问，"Printf"函数是如何知道变量是否有"String()"函数实现的。实际上，我们 需要先将变量转换为 Stringer 接口类型，如果转换成功则表示有"String()"方法。下面是一个 演示的例子：

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

这里用到了类型断言("v.(Stringer)")，用来判断变量"v"是否可以满足"Stringer"接口。 如果满足，"s"将对应转换后的 Stringer 接口类型并且"ok"被设置为"true"。然后我们通过"s"， 以 Stringer 接口的方式调用 String()函数。如果不满足该接口特征，"ok"将被设置为 false。

"Stringer"接口的命名通常是在接口方法的名字后面加"er"后缀，这里是"String+er"。

Go 中的打印函数，除了"Printf"和"Sprintf"等之外，还有一个"Fprintf"函数。不过"Fprintf"函数和 的第一个参数并不是一个文件，而是一个在"io"库中定义的接口类型：

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

这里的接口也是采用类似的命名习惯，类型的接口还有"io.Reader"和"io.ReadWriter"等。 在调用"Fprintf"函数时，可以用实现了"Write"方法的任意类型变量作为参数，例如文件、网络、管道等等。

生成素数

这里我们要给出一个并行处理程序及之间的通信。这是一个非常大的课题，我们这里只是给出一些要点。

素数筛选是一个比较经典的问题（这里侧重于 Eratosthenes 素数筛选算法的并行特征）。它以全部的 自然后为筛选对象。首选从第一个素数 2 开始，后续数列中是已经素数倍数的数去掉。每次筛选可以得到 一个新的素数，然后将新的素数加入筛选器，继续筛选后面的自然数列（这里要参考算法的描述调整）。

这里是算法工作的原理图。每个框对应一个素数筛选器，并且将剩下的数列传给下一个素数筛进行筛选。

为了产生整数序列，我们使用管道。管道可以用于连接两个并行的处理单。在 Go 语言中， 管道由运行时库管理，可以用"make"来创建新的管道。

这是"progs/sieve.go"程序的第一个函数：

```
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  }
```

函数"generate"用于生成 2, 3, 4, 5, ...自然数序列，然后依次发送到管道。 这里用到了二元操作符"<-"， 它用于向管道发送数据。当管道没有接受者的时候 会阻塞，直到有接收者从管道接受数据为止。

过滤器函数有三个参数：输入输出管道和用于过滤的素数。当输入管道读出来的数不能被 过滤素数整除时，则将当前整数发送到输出管道。这里用到了"<-"操作符，它用于从 管道读取数据。

```
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  }
```

整数生成器 generator 函数和过滤器 filters 是并行执行的。Go 语言有自己的并发 程序设计模型，这个和传统的进程/线程/轻量线程类似。为了区别，我们把 Go 语言 中的并行程序称为 *goroutines*。如果一个函数要以 goroutines 方式并行执行， 只要用"go"关键字作为函数调用的前缀即可。goroutines 和它的启动线程并行执行， 但是共享一个地址空间。例如，以 goroutines 方式执行前面的 sum 函数：

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

如果想知道计算什么时候结束，可以让 sum 用管道把结果返回：

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

再回到我们的素数筛选程序。下面程序演示如何将不同的素数筛链接在一起：

```
28  func main() {
29      ch := make(chan int)  // Create a new channel.
30      go generate(ch)  // Start generate() as a goroutine.
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  }
```

29 行先调用"generate"函数，用于产生最原始的自然数序列（从 2 开始）。然后 从输出管道读取的第一个数为新的素数，并以这个新的素数生成一个新的过滤器。 然后将新创建的过滤器添加到前一个过滤器后面，新过滤器的输出作为新的输出 管道。

sieve 程序还可以写的更简洁一点。这里是"generate"的改进，代码在 "progs/sieve1.go"中：

```
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  }
```

新完善的 generate 函数在内部进行必须的初始化操作。它创建输出管道，然后 启动 goroutine 用于产生整数序列，最后返回输出管道。它类似于一个并发程序 的工厂函数，完成后返回一个用于链接的管道。

第 12-16 行用 go 关键字启动一个匿名函数。需要注意的是，generate 函数的"ch" 变量对于匿名函数是可见，并且"ch"变量在 generate 函数返回后依然存在（因为 匿名的 goroutine 还在运行）。

这里我们采用过滤器"filter"来筛选后面的素数：

```
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  }
```

函数"sieve"对应处理的一个主循环，它只是依次将数列交给后面的素数筛选器进行筛选。 如果遇到新的素数，再输出素数后以该素数创建信的筛选器。

```
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  }
```

主函数入口启动素数生成服务器，然后打印从管道输出的素数：

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

Multiplexing

基于管道，我们可以很容易实现一个支持多路客户端的服务器程序。采用的技巧是将每个客户端私有的通信管道作为消息的一部分发送给服务器，然后服务器通过这些管道和客户端独立通信。现实中的服务器实现都很复杂，我们这里只给出一个服务器的简单实现来展现前面描述的技巧。首先定义一个"request"类型，里面包含一个 客户端的通信管道。

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

```
12  }
```

服务器对客户端发送过来的两个整数进行运算。下面是具体的函数，函数在运算完之后将结构通过结构中的 管道返回给客户端。

```
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  }
```

第 14 行现定义一个"binOp"函数类型，用于对两个整数进行运算。

服务器 routine 线程是一个无限循环，它接受客户端请求。然后为每个客户端启动一个独立的 routine 线程， 用于处理客户数据（不会被某个客户端阻塞）。

```
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  }
```

启动服务器的方法也是一个类似的 routine 线程，然后返回服务器的请求管道。

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

这里是一个简单的测试。首先启动服务器，处理函数为计算两个整数的和。接着向服务器发送"N"个请求（无阻塞）。 当所有请求都发送完了之后，再进行验证返回结果。

```
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  }
```

前面的服务器程序有个小问题：当 main 函数退出之后，服务器没有关闭，而且可能有一些客户端被阻塞在 管道通信中。为了处理这个问题，我们可给服务器增加一个控制管道，用于退出服务器。

```
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  }
```

首先给"server"函数增加一个控制管道参数，然后这样使用：

```
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  }
```

在服务器函数中，"select"操作服用于从多个通讯管道中选择一个就绪的管道。如果所有的管道都没有数据， 那么将等待知道有任意一个管道有数据。如果有多个管道就绪，则随即选择一个。服务器处理客户端请求，如果 有退出消息则退出。

最后是在 main 函数中保存"quit"管道，然后在退出的时候向服务线程发送停止命令。

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

...

```
55      quit <- true
```

当然，Go 语言及并行编程要讨论的问题很多。这个入门只是给出一些简单的例子。