

Effective Go中 文版

看云文档小组



目 录

- 简介
- 格式
- 注释
- 名字
- 分号
- 控制结构
- 函数
- 数据
- 初始化
- 方法
- 接口和其它类型
- 空白标识符
- 内嵌 (Embedding)
- 并发
- 错误
- 一个web服务示例
- 中文译者

简介

原文出处：http://www.hellogcc.org/effective_go.html

简介

Go是一个新的语言。虽然是借鉴了现有的语言，但是它独有的特性可以使得高效的Go程序，与其它语言编写的程序相比，大不相同。直接将C++或者Java 程序转换为Go程序，是不可能产生令人满意的结果——Java程序是使用Java编写的，而不是Go。另一方面，从Go的角度考虑问题则会产生成功的，而且 大不相同的程序。换句话说，想要编写好的Go程序，理解它的特性和风格是非常重要的。了解Go语言编程中已有的约定也非常重要，例如命名，格式，程序结构，等等。这会使得其他Go程序员容易理解你编写的程序。

该文档对如何编写清晰，符合语言规范的Go代码，给出了一些建议。你应该先阅读[language specification](#)，[Tour of Go](#)和[How to Write Go Code](#)，然后将该文档作为扩展阅读。

例子

[Go package sources](#)旨在不仅作为核心库来使用，而且还可以作为如何使用语言的例子。此外，许多程序包都包含了可以在[golang.org](#)网站上独立执行的例子，例如[这一个](#)（如果需要，点击单词"Example"来打开）。如果你对如何处理一个问题，或者如何进行实现有疑问，那么库中的文档，代码和例子可以提供答案，概念和背景。

格式

格式

格式化是一个最具争议，但又无关紧要的问题。人们可以习惯于不同的格式风格。但是，最好不必这样，这就不用每个人是否遵守相同风格的话题上花费时间了。问题是在没有一个长效的风格指导下，如何达到这样美好的乌托邦。

对于Go，我们采取了不同寻常的方式，让机器来处理大多数的格式问题。程序 `gofmt`（也可以用 `go fmt`，其操作于程序包的级别，而不是源文件级别），读入一个Go程序，然后输出按照标准风格缩进和垂直对齐的源码，并且保留了根据需要进行重新格式化的注释。如果你想知道如何处理某种新的布局情况，可以运行 `gofmt`；如果答案看起来不正确，则需要重新组织你的程序（或者提交一个关于 `gofmt` 的bug），不要把问题绕过去。

举个例子，不需要花费时间对结构体中每个域的注释进行排列。`Gofmt` 将会替你完成这些。给定一个声明

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

`gofmt` 将会按列进行排列：

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

标准程序包中的所有Go代码，都已经使用 `gofmt` 进行了格式化。

还是有一些格式化的细节的。非常简短：

缩进

我们使用tab进行缩进，这是 `gofmt` 的缺省输出。只有在你必须的时候才使用空格。

行长度

Go没有行长度限制。不必担心会有打孔卡片溢出。如果感觉一行太长，可以折成几行，并额外使用一个tab进行缩进。

括号

Go相比C和Java，很少需要括号：控制结构（ `if` ， `for` ， `switch` ）的语法不需要括号。而且，操作符优先级更短，更清晰。这样，

```
x<<8 + y<<16
```

的含义就已经由空格表明了。这不像其它语言。

注释

注释

Go提供了C风格的块注释 `/* */` 和C++风格的行注释 `//`。通常为行注释；块注释大多数作为程序包的注释，但也可以用于一个表达式中，或者用来注释掉一大片代码。

程序—同时又是网络服务器— `godoc`，用来处理Go源文件，抽取有关程序包内容的文档。在顶层声明之前出现，并且中间没有换行的注释，会随着声明一起被抽取，作为该项的解释性文本。这些注释的本质和风格决定了 `godoc` 所产生文档的质量。

每个程序包都应该有一个包注释，一个位于`package`子句之前的块注释。对于有多个文件的程序包，包注释只需要出现在一个文件中，任何一个文件都可以。包注释应该用来介绍该程序包，并且提供与整个程序包相关的信息。它将会首先出现在 `godoc` 页面上，并会建立后续的详细文档。

```
/*
Package regexp 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
```

如果程序包很简单，则包注释可以非常简短。

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

注释不需要额外的格式，例如星号横幅。生成的输出甚至可能会不按照固定宽度的字体进行展现，所以不要依靠用空格进行对齐— `godoc`，就像 `gofmt`，会处理这些事情。注释是不作解析的普通文本，所以HTML和其它注解，例如 `_this_`，将会逐字的被复制。对于缩进的文本，`godoc` 确实会进行调整，来按照固定宽度的字体进行显示，这适合于程序片段。 `fmt package`的包注释使用了这种方式来获得良好的效果。

根据上下文，`godoc` 甚至可能不会重新格式化注释，所以要确保它们看起来非常直接：使用正确的拼写，标点，以及语句结构，将较长的行进行折叠，等等。

在程序包里面，任何直接位于顶层声明之前的注释，都会作为该声明的文档注释。程序中每一个被导出的（大写的）名字，都应该有一个文档注释。

文档注释作为完整的语句可以工作的最好，可以允许各种自动化的展现。第一条语句应该为一条概括语句，并且使用被声明的名字作为开头。

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

如果都是使用名字来起始一个注释，那么就可以通过 `grep` 来处理 `godoc` 的输出。设想你正在查找正规表达式的解析函数，但是不记得名字“`Compile`”了，那么，你运行了命令

```
$ godoc regexp | grep parse
```

如果程序包中所有的文档注释都起始于“`This function...`”，那么 `grep` 将无法帮助你想起这个名字。但是，因为程序包是使用名字来起始每个文档注释，所以你将会看到类似这样的信息，这将使你想起你要查找的单词。

```
$ godoc regexp | grep parse
    Compile parses a regular expression and returns, if successful, a Regexp
    parsed. It simplifies safe initialization of global variables holding
    cannot be parsed. It simplifies safe initialization of global variables
$
```

Go的声明语法允许对声明进行组合。单个的文档注释可以用来介绍一组相关的常量或者变量。由于展现的是整个声明，这样的注释通常非常肤浅。

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

分组还可以用来指示各项之间的关系，例如一组实际上由一个互斥进行保护的变量。

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

```
        errorCount uint32  
    )
```


名字

- 程序包名
- Get方法
- 接口名
- 混合大小写

和其它语言一样，名字在Go中是非常重要的。它们甚至还具有语义的效果：一个名字在程序包之外的可见性是由它的首字符是否为大写来确定的。因此，值得花费一些时间来讨论Go程序中的命名约定。

程序包名

当一个程序包被导入，程序包名便可以用来访问它的内容。在

```
import "bytes"
```

之后，导入的程序包便可以谈到 `bytes.Buffer`。如果每个使用程序包的人都可以使用相同的名字来引用它的内容，这会是很有帮助的。这意味着程序包名要很好：简短，简明，能引起共鸣的。按照惯例，程序包使用小写，一个单词的名字；不需要使用下划线或者混合大小写。要力求简短，因为每个使用你的程序包的人都将敲入那个名字。不用担心会与先前的有冲突。程序包名只是导入的缺省名字；其不需要在所有源代码中是唯一的。对于很少出现的冲突情况下，导入的程序包可以选择一个不同的名字在本地使用。不管怎样，冲突是很少的，因为导入的文件名确定了所要使用的程序包。

另一种约定是，程序包名为其源目录的基础名；在 `src/pkg/encoding/base64` 中的程序包，是作为 `"encoding/base64"` 来导入的，但是名字为 `base64`，而不是 `encoding_base64` 或 `encodingBase64`。

程序包的导入者将使用名字来引用其内容，因此在程序包中被导出的名字可以利用这个事实来避免口吃现象。（不要使用 `import .` 标记，这将会简化那些必须在程序包之外运行，本不应该避免的测试）例如，在 `bufio` 程序包中的带缓冲的读入类型叫做 `Reader`，而不是 `BufReader`，因为用户看到的是 `bufio.Reader`，一个清晰，简明的名字。而且，因为被导入的实体总是通过它们的程序包名来寻址，所以 `bufio.Reader` 和 `io.Reader` 并不冲突。类似的，为 `ring.Ring` 创建一个新实例的函数——在Go中是定义一个构造器——通常会被叫做 `NewRing`，但是由于 `Ring` 是程序包导出的唯一类型，由于程序包叫做 `ring`，所以它只叫做 `New`。这样，程序包的客户将会看到 `ring.New`。使用程序包结构可以帮助你选择好的名字。

另一个小例子是 `once.Do`；`once.Do(setup)` 很好读，写成 `once.DoOrWaitUntilDone(setup)` 并不会有所改善。长名字并不会自动使得事物更易读。具有帮助性的文档注释往往会比格外长的名字更有用。

Get方法

Go不提供对Get方法和Set方法的自动支持。你自己提供Get方法和Set方法是没有错的，通常这么做是合适的。但是，在Get方法的名字中加上 `Get`，是不符合语言习惯的，并且也没有必要。如果你有一个域叫做 `owner`（小写，不被导出），则Get方法应该叫做 `Owner`（大写，被导出），而不是 `GetOwner`。对于要导出的，使用大写名字，提供了区别域和方法的钩子。Set方法，如果需要，则可以叫做 `SetOwner`。这些名字在实际中都很好读：

```
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

接口名

按照约定，单个方法的接口使用方法名加上“er”后缀来命名，或者类似的修改来构造一个施动者名词：`Reader`，`Writer`，`Formatter`，`CloseNotifier`等。

有许多这样的名字，最有效的方式就是尊重它们，以及它们所体现的函数名字。`Read`，`Write`，`Close`，`Flush`，`String`等，都具有规范的签名和含义。为了避免混淆，不要为你的方法使用这些名字，除非其具有相同的签名和含义。反过来讲，如果你的类型实现了一个和众所周知的类型具有相同含义的方法，那么就使用相同的名字和签名；例如，为你的字符串转换方法起名为 `String`，而不是 `ToString`。

混合大小写

最后，Go约定使用 `MixedCaps` 或者 `mixedCaps` 的形式，而不是下划线来书写多个单词的名字。

分号

类似于C，Go的规范语法是使用分号来终结语句的。但是于C不同的是，这些分号并不在源码中出现。词法分析器会在扫描时，使用简单的规则自动插入分号，因此输入文本中大部分是没有分号的。

规则是这样的，如果在换行之前的最后一个符号为一个标识符（包括像 `int` 和 `float64` 这样的单词），一个基本的文字，例如数字或者字符串常量，或者如下的一个符号

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

则词法分析器总是会在符号之后插入一个分号。这可以总结为“如果换行出现在可以结束一条语句的符号之后，则插入一个分号”。

紧挨着右大括号之前的分号也可以省略掉，这样，语句

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

就不需要分号。地道的Go程序只在 `for` 循环子句中使用分号，来分开初始化，条件和继续执行，这些元素。分号也用于在一行中分开多条语句，这也是你编写代码应该采用的方式。

分号插入规则所导致的一个结果是，你不能将控制结构（`if`，`for`，`switch` 或 `select`）的左大括号放在下一行。如果这样做，则会在大括号之前插入一个分号，这将会带来不是想要的效果。应该这样编写

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

而不是这样

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

分号

控制结构

- [If](#)
- [重新声明和重新赋值](#)
- [For](#)
- [Switch](#)
- [类型switch](#)

Go的控制结构与C的相关，但是有重要的区别。没有 `do` 或者 `while` 循环，只有一个稍微广义的 `for`；`switch` 更加灵活；`if` 和 `switch` 接受一个像 `for` 那样可选的初始化语句；`break` 和 `continue` 语句接受一个可选的标号来指定中断或继续什么；还有一些新的控制结构，包括类型switch和多路通信复用器（multiway communications multiplexer），`select`。语句也稍微有些不同：没有圆括号，并且控制结构体必须总是由大括号包裹。

If

Go中，简单的 `if` 看起来是这样的：

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

强制的大括号可以鼓励大家在多行中编写简单的 `if` 语句。不管怎样，这是一个好的风格，特别是当控制结构体包含了一条控制语句，例如 `return` 或者 `break`。

既然 `if` 和 `switch` 接受一个初始化语句，那么常见的方式是用来建立一个局部变量。

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

在Go的库中，你会发现当 `if` 语句不会流向下一条语句时——也就是说，控制结构体结束于 `break`，`continue`，`goto` 或者 `return`——则不必要的 `else` 会被省略掉。

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

这个例子是一种常见的情况，代码必须防范一系列的错误条件。如果成功的控制流是沿着页面往下走，来

消除它们引起的错误情况，那么代码会非常易读。由于错误情况往往会结束于 `return` 语句，因此代码不需要有 `else` 语句。

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

重新声明和重新赋值

另外：上一章节的最后一个例子，展示了 `:=` 短声明形式的工作细节。该声明调用了 `os.Open` 进行读取，

```
f, err := os.Open(name)
```

该语句声明了两个变量，`f` 和 `err`。几行之后，又调用了 `f.Stat` 进行读取，

```
d, err := f.Stat()
```

这看起来像是又声明了 `d` 和 `err`。但是，注意 `err` 在两条语句中都出现了。这种重复是合法的：`err` 是在第一条语句中被声明，而在第二条语句中只是被重新赋值。这意味着使用之前已经声明过的 `err` 变量调用 `f.Stat`，只会是赋给其一个新的值。

在 `:=` 声明中，变量 `v` 即使已经被声明过，也可以出现，前提是：

- 该声明和 `v` 已有的声明在相同的作用域中（如果 `v` 已经在外面的作用域里被声明了，则该声明将会创建一个新的变量 §）
- 初始化中相应的值是可以被赋给 `v` 的
- 并且，声明中至少有其它一个变量将被声明为一个新的变量

这种不寻常的属性纯粹是从实用主义方面来考虑的。例如，这会使得在一个长的 `if-else` 链中，很容易地使用单个 `err` 值。你会经常看到这种用法。

§ 值得一提的是，在Go中，函数参数和返回值的作用域与函数体的作用域是相同的，虽然它们在词法上是出现在包裹函数体的大括号外面。

For

Go的 `for` 循环类似于—但又不等同于—C的。它统一了 `for` 和 `while`，并且没有 `do-while`。有三

种形式，其中只有一个具有分号。

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

短声明使得在循环中很容易正确的声明索引变量。

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

如果你是在数组，切片，字符串或者map上进行循环，或者从channel中进行读取，则可以使用 `range` 子句来管理循环。

```
for key, value := range oldMap {
    newMap[key] = value
}
```

如果你只需要range中的第一项（`key`或者`index`），则可以丢弃第二个：

```
for key := range m {
    if key.expired() {
        delete(m, key)
    }
}
```

如果你只需要range中的第二项（`value`），则可以使用空白标识符，一个下划线，来丢弃第一个：

```
sum := 0
for _, value := range array {
    sum += value
}
```

空白标识符有许多用途，这在[后面的章节](#)中会有介绍。

对于字符串，`range` 会做更多的事情，通过解析UTF-8来拆分成单个的Unicode编码点。错误的编码会消耗一个字节，产生一个替代的符文（`rune`）`U+FFFD`。（名字（与内建类型相关联的）`rune` 是Go的术语，用于指定一个单独的Unicode编码点。详情参见[the language specification](#)）循环

```
for pos, char := range "日本\x80語" { // \x80 is an illegal UTF-8 encoding
    fmt.Printf("character %#U starts at byte position %d\n", char, pos)
}
```

会打印出

```
character U+65E5 '日' starts at byte position 0
character U+672C '本' starts at byte position 3
character U+FFFD '?' starts at byte position 6
character U+8A9E '語' starts at byte position 7
```

最后，Go没有逗号操作符，并且 `++` 和 `--` 是语句而不是表达式。因此，如果你想在 `for` 中运行多个变量，你需要使用并行赋值（尽管这样会阻碍使用 `++` 和 `--`）。

```
// 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` 要比C的更加通用。表达式不需要为常量，甚至不需要为整数，`case`是按照从上到下的顺序进行求值，直到找到匹配的。如果 `switch` 没有表达式，则对 `true` 进行匹配。因此，可以—按照语言习惯—将 `if - else - if - else` 链写成一个 `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
}
```

`switch`不会自动从一个`case`子句跌落到下一个`case`子句。但是`case`可以使用逗号分隔的列表。

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

虽然和其它类C的语言一样，使用 `break` 语句来提前中止 `switch` 在Go中几乎不怎么常见。不过，有时本文档使用 [看云](#) 构建

候是需要中断包含它的循环，而不是switch。在Go中，可以通过在循环上加一个标号，然后“breaking”到那个标号来达到目的。该例子展示了这些用法。

```
Loop:
    for n := 0; n < len(src); n += size {
        switch {
        case src[n] < sizeOne:
            if validateOnly {
                break
            }
            size = 1
            update(src[n])

        case src[n] < sizeTwo:
            if n+1 >= len(src) {
                err = errShortInput
                break Loop
            }
            if validateOnly {
                break
            }
            size = 2
            update(src[n] + src[n+1]<<shift)
        }
    }
```

当然，continue 语句也可以接受一个可选的标号，但是只能用于循环。

作为这个章节的结束，这里有一个对字节切片进行比较的程序，使用了两个 switch 语句：

```
// Compare returns an integer comparing the two byte slices,
// 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
}
```

类型switch

switch还可用于获得一个接口变量的动态类型。这种类型switch使用类型断言的语法，在括号中使用关键字 type 。如果switch 在表达式中声明了一个变量，则变量会在每个子句中具有对应的类型。比较符合

语言习惯的方式是在这些case里重用一個名字，实际上是在每个case里声名一个新的变量，其具有相同的名字，但是不同的类型。

```
var t interface{}
t = functionOfSomeType()
switch t := t.(type) {
default:
    fmt.Printf("unexpected type %T", t)          // %T prints whatever type t has
case bool:
    fmt.Printf("boolean %t\n", t)                // t has type bool
case int:
    fmt.Printf("integer %d\n", t)                // t has type int
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t)    // t has type *bool
case *int:
    fmt.Printf("pointer to integer %d\n", *t)    // t has type *int
}
```

函数

- [多个返回值](#)
- [命名的结果参数](#)
- [延期执行](#)

多个返回值

Go的其中一个不同寻常的特点是，函数和方法可以返回多个值。这种形式可以用来改进C程序中几个笨拙的语言风格：返回一个错误，例如 `-1` 对应于 `EOF`，同时修改一个由地址传递的参数。

在C中，一个写错误是由一个负的计数和一个隐藏在易变位置（a volatile location）的错误代码来表示的。在Go中，`Write` 可以返回一个计数和一个错误：“是的，你写了一些字节，但并没有全部写完，由于设备已经被填满了”。在程序包 `os` 的文件中，`Write` 方法的签名是：

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

正如文档所言，其返回写入的字节数和一个非零的 `error`，当 `n != len(b)` 的时候。这是一种常见的风格；更多的例子可以参见错误处理章节。

类似的方法使得不再需要传递一个返回值指针来模拟一个引用参数。这里有一个非常简单的函数，用来从字节切片中的一个位置抓取一个数，返回该数和下一个位置。

```
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
}
```

你可以使用它来扫描输入切片 `b` 中的数字，如：

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

命名的结果参数

Go函数的返回或者结果“参数”可以给定一个名字，并作为一个普通变量来使用，就像是输入参数一样。当被命名时，它们在函数起始处被初始化为对应类型的零值；如果函数执行了没有参数的 `return` 语句，

则结果参数的当前值便被作为要返回的值。

名字并不是强制的，但是可以使代码更加简短清晰：它们也是文档。如果我们将 `nextInt` 的结果进行命名，则其要返回的 `int` 是对应的哪一个就很显然了。

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

因为命名结果是被初始化的，并且与没有参数的 `return` 绑定在一起，所以它们即简单又清晰。这里是一个 `io.ReadFull` 的版本，很好地使用了这些特性：

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

延期执行

Go 的 `defer` 语句用来调度一个函数调用（被延期的函数），使其在执行 `defer` 的函数即将返回之前才被运行。这是一种不寻常但又很有效的方法，用于处理类似于不管函数通过哪个执行路径返回，资源都必须要被释放的情况。典型的例子是对一个互斥解锁，或者关闭一个文件。

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close() // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

对像 `Close` 这样的函数调用进行延期，有两个好处。首先，其确保了你不会忘记关闭文件，如果你之后修改了函数增加一个新的返回路径，会很容易犯这样的错。其次，这意味着关闭操作紧挨着打开操作，这本文档使用 [看云](#) 构建

比将其放在函数结尾更加清晰。

被延期执行的函数，它的参数（包括接收者，如果函数是一个方法）是在defer执行的时候被求值的，而不是在调用执行的时候。这样除了不用担心变量随着函数的执行值会改变，这还意味着单个被延期执行的调用点可以延期多个函数执行。这里有一个简单的例子。

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

被延期的函数按照LIFO的顺序执行，所以这段代码会导致在函数返回时打印出 4 3 2 1 0。一个更加真实的例子，这是一个跟踪程序中函数执行的简单方法。我们可以编写几个类似这样的，简单的跟踪程序：

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

// Use them like this:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

利用被延期的函数的参数是在 defer 执行的时候被求值这个事实，我们可以做的更好些。trace程序可以为untrace程序建立参数。这个例子：

```
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()
}
```

会打印出

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

对于习惯于其它语言中的块级别资源管理的程序员，`defer` 可能看起来很奇怪，但是它最有趣和强大的应用正是来自于这样的事实，这是基于函数的而不是基于块的。我们将会在 `panic` 和 `recover` 章节中看到它另一个可能的例子。

数据

- [使用`new`进行分配](#)
- [构造器和复合文字](#)
- [使用`make`进行分配](#)
- [数组](#)
- [切片](#)
- [二维切片](#)
- [Maps](#)
- [打印输出](#)
- [append内建函数](#)

使用 new 进行分配

Go有两个分配原语，内建函数 `new` 和 `make`。它们所做的事情有所不同，并且用于不同的类型。这会有些令人混淆，但规则其实很简单。我们先讲下 `new`。这是一个用来分配内存的内建函数，但是不像在其它语言中，它并不初始化内存，只是将其置零。也就是说，`new(T)` 会为 `T` 类型的新项目，分配被置零的存储，并且返回它的地址，一个类型为 `*T` 的值。在Go的术语中，其返回一个指向新分配的类型为 `T`，值为零的指针。

由于 `new` 返回的内存是被置零的，这会有助于你将数据结构设计成，每个类型的零值都可以使用，而不需要进一步初始化。这意味着，数据结构的用户可以使用 `new` 来创建数据，并正确使用。例如，`bytes.Buffer` 的文档说道，“`Buffer` 的零值是一个可以使用的空缓冲”。类似的，`sync.Mutex` 没有显式的构造器和 `Init` 方法。相反的，`sync.Mutex` 的零值被定义为一个未加锁的互斥。

“零值可用”的属性是可以传递的。考虑这个类型声明。

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

`SyncedBuffer` 类型的值也可以在分配或者声明之后直接使用。在下一个片段中，`p` 和 `v` 都不需要进一步的处理便可以正确地工作。

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

构造器和复合文字

有时候零值并不够好，需要一个初始化构造器（`constructor`），正如这个源自程序包 `os` 的例子。

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

有许多这样的模版。我们可以使用复合文字 (composite literal) 进行简化，其为一个表达式，在每次求值的时候会创建一个新实例。

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

注意，不像C，返回一个局部变量的地址是绝对没有问题的；变量关联的存储在函数返回之后依然存在。实际上，使用复合文字的地址也会在每次求值时分配一个新的实例，所以，我们可以将最后两行合并起来。

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

复合文字的域按顺序排列，并且必须都存在。然而，通过 `field : value` 显式地为元素添加标号，则初始化可以按任何顺序出现，没有出现的则对应为零值。因此，我们可以写成

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

作为一种极端情况，如果复合文字根本不包含域，则会为该类型创建一个零值。表达式 `new(File)` 和 `&File{}` 是等价的。

复合文字还可用于arrays，slices和maps，域标号使用适当的索引或者map key。下面的例子中，不管 `Enone`，`Eio` 和 `Einval` 的值是什么，只要它们不同，初始化就可以工作。

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


使用 make 进行分配

回到分配的话题。内建函数 `make(T, args)` 与 `new(T)` 的用途不一样。它只用来创建 slice, map 和 channel, 并且返回一个初始化的(而不是置零), 类型为 `T` 的值(而不是 `*T`)。之所以有所不同, 是因为这三个类型的背后是象征着, 对使用前必须初始化的数据结构的引用。例如, slice 是一个三项描述符, 包含一个指向数据(在数组中)的指针, 长度, 以及容量, 在这些项被初始化之前, slice 都是 `nil` 的。对于 slice, map 和 channel, `make` 初始化内部数据结构, 并准备好可用的值。例如,

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

分配一个有100个int的数组, 然后创建一个长度为10, 容量为100的slice结构, 并指向数组前10个元素上。(当创建slice时, 容量可以省略掉, 更多信息参见slice章节。)对应的, `new([]int)` 返回一个指向新分配的, 被置零的slice结构体的指针, 即指向 `nil` slice值的指针。

这些例子阐释了 `new` 和 `make` 之间的差别。

```
var p *[]int = new([]int) // allocates slice structure; *p == nil; rarely useful
var v []int = make([]int, 100) // the slice v now refers to a new array of 100 ints

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

记住 `make` 只用于map, slice和channel, 并且不返回指针。要获得一个显式的指针, 使用 `new` 进行分配, 或者显式地使用一个变量的地址。

数组

数组可以用于规划内存的精细布局, 有时利于避免分配, 不过从根本上讲, 它们是切片的基本构件, 这是下一章节的话题。作为铺垫, 这里介绍一下数组。

在Go和C中, 数组的工作方式有几个重要的差别。在Go中,

- 数组是值。将一个数组赋值给另一个, 会拷贝所有的元素。
- 特别是, 如果你给函数传递一个数组, 其将收到一个数组的拷贝, 而不是它的指针。
- 数组的大小是其类型的一部分。类型 `[10]int` 和 `[20]int` 是不同的。

数组为值这样的属性, 可以很有用处, 不过也会有代价; 如果你希望类C的行为和效率, 可以传递一个数组的指针。

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
}
```

```

    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array) // Note the explicit address-of operator

```

不过，这种风格并不符合Go的语言习惯。相反的，应该使用切片。

切片

切片（slice）对数组进行封装，提供了一个针对串行数据，更加通用，强大和方便的接口。除了像转换矩阵这样具有显式维度的项，Go中大多数的数组编程都是通过切片完成，而不是简单数组。

切片持有对底层数组的引用，如果你将一个切片赋值给另一个，二者都将引用同一个数组。如果函数接受一个切片参数，那么其对切片的元素所做的改动，对于调用者是可见的，好比是传递了一个底层数组的指针。因此，`Read` 函数可以接受一个切片参数，而不是一个指针和一个计数；切片中的长度已经设定了要读取的数据的上限。这是程序包 `os` 中，`File` 类型的 `Read` 方法的签名：

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

该方法返回读取的字节数和一个错误值，如果存在的话。要读取一个大缓冲 `b` 中的前32个字节，可以将缓冲进行切片（这里是动词）。

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

这种切片很常见，而且有效。实际上，如果先不考虑效率，下面的片段也可以读取缓冲的前32个字节。

```

var n int
var err 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
}

```

只要还符合底层数组的限制，切片的长度就可以进行改变；直接将其赋值给它自己的切片。切片的容量，可以通过内建函数 `cap` 访问，告知切片可以获得的最大长度。这里有一个函数可以为切片增加数据。如果数据超出了容量，则切片会被重新分配，然后返回新产生的切片。该函数利用了一个事实，即当用于 `nil` 切片时，`len` 和 `cap` 是合法的，并且返回0。

```

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
}

```

我们必须在后面返回切片，尽管 `Append` 可以修改 `slice` 的元素，切片本身（持有指针，长度和容量的运行时数据结构）是按照值传递的。

为切片增加元素的想法非常有用，以至于实现了一个内建的 `append` 函数。不过，要理解该函数的设计，我们还需要一些更多的信息，所以我们放到后面再说。

二维切片

Go的数组和切片都是一维的。要创建等价的二维数组或者切片，需要定义一个数组的数组或者切片的切片，类似这样：

```

type Transform [3][3]float64 // A 3x3 array, really an array of arrays.
type LinesOfText [][]byte    // A slice of byte slices.

```

因为切片是可变长度的，所以可以将每个内部的切片具有不同的长度。这种情况很常见，正如我们的 `LinesOfText` 例子中：每一行都有一个独立的长度。

```

text := LinesOfText{
    []byte("Now is the time"),
    []byte("for all good gophers"),
    []byte("to bring some fun to the party."),
}

```

有时候是需要分配一个二维切片的，例如这种情况可见于当扫描像素行的时候。有两种方式可以实现。一种是独立的分配每一个切片；另一种是分配单个数组，为其指定单独的切片们。使用哪一种方式取决于你的应用。如果切片们可能会增大或者缩小，则它们应该被单独的分配以避免覆写了下一行；如果不会，则构建单个分配会更加有效。作为参考，这里有两种方式的框架。首先是一次一行：

```

// Allocate the top-level slice.
picture := make([][]uint8, YSize) // One row per unit of y.
// Loop over the rows, allocating the slice for each row.
for i := range picture {
    picture[i] = make([]uint8, XSize)
}

```

然后是分配一次，被切片成多行：

```
// Allocate the top-level slice, the same as before.
picture := make([][]uint8, YSize) // One row per unit of y.
// Allocate one large slice to hold all the pixels.
pixels := make([]uint8, XSize*YSize) // Has type []uint8 even though picture is [][]uint8.
// Loop over the rows, slicing each row from the front of the remaining pixels slice.
for i := range picture {
    picture[i], pixels = pixels[:XSize], pixels[XSize:]
}
```

Maps

Map是一种方便，强大的内建数据结构，其将一个类型的值（key）与另一个类型的值（element或value）关联一起。key可以为任何定义了等于操作符的类型，例如整数，浮点和复数，字符串，指针，接口（只要其动态类型支持等于操作），结构体和数组。切片不能作为map的key，因为它们没有定义等于操作。和切片类似，map持有对底层数据结构的引用。如果将map传递给函数，其对map的内容做了改变，则这些改变对于调用者是可见的。

Map可以使用通常的复合文字语法来构建，使用分号分隔key和value，这样很容易在初始化的时候构建它们。

```
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map的值，在语法上看起来跟数组和切片类似，只不过索引不需要为一个整数。

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

尝试使用一个不在map中的key来获取map值，将会返回map中元素相应类型的零值。例如，如果map包含的是整数，则查找一个不存在的key将会返回0。可以通过值类型为bool的map来实现一个集合。将map项设置为true，来将值放在集合中，然后通过简单的索引来进行测试。

```
attended := map[string]bool{
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // will be false if person is not in the map
    fmt.Println(person, "was at the meeting")
}
```

有时你需要区分没有的项和值为零的项。是否有一个项为 "UTC"，或者由于其根本不在map中，所以为空字符串？你可以通过多赋值的形式来进行辨别。

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

这被形象的称为“comma ok”用法。在这个例子中，如果 tz 存在，seconds 将被设置为适当的值，ok 将为真；如果不存在，seconds 将被设置为零，ok 将为假。这有个例子，并增加了一个友好的错误报告：

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

如果只测试是否在map中存在，而不关心实际的值，你可以将通常使用变量的地方换成空白标识符（_）

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

要删除一个map项，使用 delete 内建函数，其参数为map和要删除的key。即使key已经不在map中，这样做也是安全的。

```
delete(timeZone, "PDT") // Now on Standard Time
```

打印输出

Go中的格式化打印使用了与C中 printf 家族类似的风格，不过更加丰富和通用。这些函数位于 fmt 程序包中，并具有大写的名字：fmt.Printf，fmt.Fprintf，fmt.Sprintf 等等。字符串函数（Sprintf 等）返回一个字符串，而不是填充到提供的缓冲里。

你不需要提供一个格式串。对每个 Printf，Fprintf 和 Sprintf，都有另外一对相应的函数，例如 Print 和 Println。这些函数不接受格式串，而是为每个参数生成一个缺省的格式。Println 版本还会在参数之间插入一个空格，并添加一个换行，而 Print 版本只有当两边的操作数都不是字符串的时候才增加一个空格。在这个例子中，每一行都会产生相同的输出。

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

格式化打印函数 `fmt.Fprint` 等，接受的第一个参数为任何一个实现了 `io.Writer` 接口的对象；变量 `os.Stdout` 和 `os.Stderr` 是常见的实例。

接下来这些就和C不同了。首先，数字格式，像 `%d`，并不接受正负号和大小的标记；相反的，打印程序使用参数的类型来决定这些属性。

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

会打印出

```
18446744073709551615 ffffffffffffffffff; -1 -1
```

如果只是想要缺省的转换，像十进制整数，你可以使用通用格式 `%v`（代表“value”）；这正是 `Print` 和 `Println` 所产生的结果。而且，这个格式可以打印任意的值，甚至是数组，切片，结构体和map。这是一个针对前面章节中定义的时区map的打印语句

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

其会输出

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

当然，map的key可能会按照任意顺序被输出。当打印一个结构体时，带修饰的格式 `%+v` 会将结构体的域使用它们的名字进行注解，对于任意的值，格式 `%#v` 会按照完整的Go语法打印出该值。

```
type T struct {
    a int
    b float64
    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)
```

会打印出

```
&{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}
```

(注意符号&) 还可以通过 %q 来实现带引号的字符串格式，用于类型为 string 或 []byte 的值。格式 %#q 将尽可能的使用反引号。(格式 %q 还用于整数和符文，产生一个带单引号的符文常量。) 还有， %x 用于字符串，字节数组和字节切片，以及整数，生成一个长的十六进制字符串，并且如果在格式中有一个空格 (% x)，其将会在字节中插入空格。

另一个方便的格式是 %T，其可以打印出值的类型。

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

会打印出

```
map[string] int
```

如果你想控制自定义类型的缺省格式，只需要对该类型定义一个签名为 String() string 的方法。对于我们的简单类型 T，看起来可能是这样的。

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

会按照如下格式打印

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

(如果你需要打印类型为 T 的值，同时需要指向 T 的指针，那么 String 的接收者必须为值类型的；这个例子使用了指针，是因为这对于结构体类型更加有效和符合语言习惯。更多信息参见下面的章节 [pointers vs. value receivers](#))

我们的 String 方法可以调用 Sprintf，是因为打印程序是完全可重入的，并且可以按这种方式进行包装。然而，对于这种方式，有一个重要的细节需要明白：不要将调用 Sprintf 的 String 方法构造成无穷递归。如果 Sprintf 调用尝试将接收者直接作为字符串进行打印，就会导致再次调用该方法，发生这样的情况。这是一个很常见的错误，正如这个例子所示。

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m) // Error: will recur forever.
}
```

这也容易修改：将参数转换为没有方法函数的，基本的字符串类型。

```
type MyString string
func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", string(m)) // OK: note conversion.
}
```

在[初始化章节](#)，我们将会看到另一种避免该递归的技术。

另一种打印技术，是将一个打印程序的参数直接传递给另一个这样的程序。Printf 的签名使用了类型 `...interface{}` 作为最后一个参数，来指定在格式之后可以出现任意数目的（任意类型的）参数。

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

在函数 Printf 内部，v 就像是一个类型为 `[]interface{}` 的变量，但是如果其被传递给另一个可变参数的函数，其就像是一个正常的参数列表。这里有一个对我们上面用到的函数 `log.Println` 的实现。其将参数直接传递给 `fmt.Sprintln` 来做实际的格式化。

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

我们在嵌套调用 `Sprintln` 中 v 的后面使用了 `...` 来告诉编译器将 v 作为一个参数列表；否则，其会只将 v 作为单个切片参数进行传递。

除了我们这里讲到的之外，还有很多有关打印的技术。详情参见 `godoc` 文档中对 `fmt` 的介绍。

顺便说下，`...` 参数可以为一个特定的类型，例如 `...int`，可以用于最小值函数，来选择整数列表中的最小值：

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

append内建函数

现在，我们需要解释下 `append` 内建函数的设计了。`append` 的签名与我们上面定制的 `Append` 函数不

同。简略地讲，类似于这样：

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

其中T为任意给定类型的占位符。你在Go中是无法写出一个类型 T 由调用者来确定的函数。这就是为什么 `append` 是内建的：它需要编译器的支持。

`append` 所做的事情是将元素添加到切片的结尾，并返回结果。需要返回结果，是因为和我们手写的 `Append` 一样，底层的数组可能会改变。这个简单的例子

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

会打印出 `[1 2 3 4 5 6]`。所以 `append` 的工作方式有点像 `Printf`，搜集任意数目的参数。

但是，如果我们想按照我们的 `Append` 那样做，给切片增加一个切片，那么该怎么办？简单：在调用点使用 `...`，就像我们在上面调用 `Output` 时一样。这个片段会产生和上面相同的输出。

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

如果没有 `...`，则会因为类型错误而无法编译；`y` 不是 `int` 型的。

初始化

- [常量](#)
- [变量](#)
- [init函数](#)

Go中的初始化，虽然表面上看和C或者C++差别不大，但功能更加强大。在初始化过程中可以构建复杂的结构体，并且能够正确处理初始化对象之间，甚至不同程序包之间的顺序问题。

常量

Go中的常量仅仅就是一常量。它们是在编译时被创建，即使被定义为函数局部的也如此，并且只能是数字，字符（符文），字符串或者布尔类型。由于编译时的限制，定义它们的表达式必须为能被编译器求值的常量表达式。例如，`1<<3` 是一个常量表达式，而 `math.Sin(math.Pi/4)` 不是，因为函数调用 `math.Sin` 需要在运行时才发生。

在Go中，枚举常量使用 `iota` 枚举器来创建。由于 `iota` 可以为表达式的一部分，并且表达式可以被隐式的重复，所以很容易创建复杂的值集。

```
type ByteSize float64

const (
    _           = iota // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

可以将一个方法，比如 `String`，附加到任何用户定义的类型上，这种能力使得任何值都可以自动格式化打印。虽然你会看到它经常用于结构体，但这种技术还可用于标量类型，比如 `ByteSize` 这样的浮点类型。

```
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` 会打印出 `1.00YB`，而 `ByteSize(1e13)` 会打印出 `9.09TB`。

这里使用 `Sprintf` 来实现 `ByteSize` 的 `String` 方法是安全的（避免了无穷递归），这并不是因为做了转换，而是因为它使用 `%f` 来调用 `Sprintf` 的，其不是一个字符串格式：`Sprintf` 只有当想要一个字符串的时候，才调用 `String` 方法，而 `%f` 是想要一个浮点值。

变量

变量可以像常量那样进行初始化，不过初始值可以为运行时计算的通用表达式。

```

var (
    home   = os.Getenv("HOME")
    user   = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)

```

init函数

最后，每个源文件可以定义自己的不带参数的（`niladic`）`init` 函数，来设置它所需的状态。（实际上每个文件可以有多个 `init` 函数。）`init` 是在程序包中所有变量声明都被初始化，以及所有被导入的程序包中的变量初始化之后才被调用。

除了用于无法通过声明来表示的初始化以外，`init` 函数的一个常用法是在真正执行之前进行验证或者修复程序状态的正确性。

```

func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath may be overridden by --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath, "override default GOPATH")
}

```


方法

指针 vs. 值

正如我们从 `ByteSlice` 上看到的，任何命名类型（指针和接口除外）都可以定义方法（method）；接收者（receiver）不必为一个结构体。

在上面有关切片的讨论中，我们编写了一个 `Append` 函数。我们还可以将其定义成切片的方法。为此，我们首先声明一个用于绑定该方法的命名类型，然后将方法的接收者作为该类型的值。

```
type ByteSlice []byte

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

这样还是需要方法返回更新后的切片。我们可以通过重新定义方法，接受一个 `ByteSlice` 的指针作为它的接收者，来消除这样笨拙的方式。这样，方法就可以改写调用者的切片。

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

实际上，我们可以做的更好。如果我们将函数修改成标准 `Write` 方法的样子，像这样，

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

那么类型 `*ByteSlice` 就会满足标准接口 `io.Writer`，这样就很方便。例如，我们可以打印到该类型的变量中。

```
var b ByteSlice
fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

我们传递 `ByteSlice` 的地址，是因为只有 `*ByteSlice` 才满足 `io.Writer`。关于接收者对指针和值的规则是这样的，值方法可以在指针和值上进行调用，而指针方法只能在指针上调用。这是因为指针方法可以修改接收者；使用拷贝的值来调用它们，将会导致那些修改会被丢弃。

顺便说一下，在字节切片上使用 `write` 的思想，是实现 `bytes.Buffer` 的核心。

接口和其它类型

- [接口](#)
- [转换](#)
- [接口转换和类型断言](#)
- [概述](#)
- [接口和方法](#)

接口

Go中的接口为指定对象的行为提供了一种方式：如果事情可以这样做，那么它就可以在这里使用。我们已经看到一些简单的例子；自定义的打印可以通过 `String` 方法来实现，而 `Fprintf` 可以通过 `Write` 方法输出到任意的地方。只有一个或两个方法的接口在Go代码中很常见，并且它的名字通常来自这个方法，例如实现 `Write` 的 `io.Writer`。

类型可以实现多个接口。例如，如果一个集合实现了 `sort.Interface`，其包含 `Len()`，`Less(i, j int) bool` 和 `Swap(i, j int)`，那么它就可以通过程序包 `sort` 中的程序来进行排序，同时它还可以有一个自定义的格式器。在这个人造的例子中，`Sequence` 同时符合这些条件。

```
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` 方法重复了 `Sprint` 对切片所做的工作。如果我们在调用 `Sprint` 之前，将 `Sequence` 转换为普通的 `[]int`，则可以共享所做的工作。

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

这个对象方法算是转换技术的另一个例子，从 `String` 方法中安全地调用 `Sprintf`。因为如果我们忽略类型名字，这两个类型（`Sequence` 和 `[]int`）是相同的，在它们之间进行转换是合法的。该转换并不创建新的值，只不过是暂时使现有的值具有一个新的类型。（有其它的合法转换，像整数到浮点，是会创建新值的。）

将表达式的类型进行转换，来访问不同的方法集合，这在Go程序中是一种常见用法。例如，我们可以使用已有类型 `sort.IntSlice` 来将整个例子简化成这样：

```
type Sequence []int

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

现在，`Sequence` 没有实现多个接口（排序和打印），相反的，我们利用了能够将数据项转换为多个类型（`Sequence`，`sort.IntSlice` 和 `[]int`）的能力，每个类型完成工作的一部分。这在实际中不常见，但是却可以很有效。

接口转换和类型断言

类型switch为一种转换形式：它们接受一个接口，在switch的每个case中，从某种意义上将其转换为那种case的类型。这里有一个简化版本，展示了 `fmt.Printf` 中的代码如何使用类型switch将一个值转换为字符串。如果其已经是字符串，那么我们想要接口持有的实际字符串值，如果其有一个 `String` 方法，则我们想要调用该方法的结果。

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

var value interface{} // Value provided by caller.
switch str := value.(type) {
case string:
    return str
case Stringer:
    return str.String()
}
```

第一种情况找到一个具体的值；第二种将接口转换为另一个。使用这种方式进行混合类型完全没有问题。

如果我们只关心一种类型该如何做？如果我们知道值为一个 `string`，只是想将它抽取出来该如何做？只

有一个case的类型switch是可以的，不过也可以用类型断言。类型断言接受一个接口值，从中抽取显式指定类型的值。其语法借鉴了类型switch子句，不过是使用了显式的类型，而不是 `type` 关键字：

```
value.(typeName)
```

结果是一个为静态类型 `typeName` 的新值。该类型或者是一个接口所持有的具体类型，或者是可以被转换的另一个接口类型。要抽取我们已知值中的字符串，可以写成：

```
str := value.(string)
```

不过，如果该值不包含一个字符串，则程序会产生一个运行时错误。为了避免这样，可以使用“comma, ok”的习惯用法来安全地测试值是否为一个字符串：

```
str, ok := value.(string)
if ok {
    fmt.Printf("string value is: %q\n", str)
} else {
    fmt.Printf("value is not a string\n")
}
```

如果类型断言失败，则 `str` 将依然存在，并且类型为字符串，不过其为零值，一个空字符串。

这里有一个 `if - else` 语句的实例，其效果等价于这章开始的类型switch例子。

```
if str, ok := value.(string); ok {
    return str
} else if str, ok := value.(Stringer); ok {
    return str.String()
}
```

概述

如果一个类型只是用来实现接口，并且除了该接口以外没有其它被导出的方法，那就不需要导出这个类型。只导出接口，清楚地表明了其重要的是行为，而不是实现，并且其它具有不同属性的实现可以反映原始类型的行为。这也避免了对每个公共方法实例进行重复的文档介绍。

这种情况下，构造器应该返回一个接口值，而不是所实现的类型。作为例子，在hash库里，`crc32.NewIEEE` 和 `adler32.New` 都是返回了接口类型 `hash.Hash32`。在Go程序中，用CRC-32算法来替换Adler-32，只需要修改构造器调用；其余代码都不受影响。

类似的方式可以使得在不同 `crypto` 程序包中的流密码算法，可以与链在一起的块密码分离开。

`crypto/cipher` 程序包中的 `Block` 接口，指定了块密码的行为，即提供对单个数据块的加密。然后，根据 `bufio` 程序包类推，实现该接口的加密包可以用于构建由 `Stream` 接口表示的流密码，而无需知道

块加密的细节。

`crypto/cipher` 接口看起来是这样的：

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

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

这里有一个计数器模式（CTR）流的定义，其将块密码转换为流密码；注意块密码的细节被抽象掉了：

```
// NewCTR returns a Stream that encrypts/decrypts using the given Block in
// counter mode. The length of iv must be the same as the Block's block size.
func NewCTR(block Block, iv []byte) Stream
```

`NewCTR` 并不只是用于一个特定的加密算法和数据源，而是用于任何对 `Block` 接口的实现和任何 `Stream`。因为它们返回接口值，所以将CTR加密替换为其它加密模式只是一个局部的改变。构造器调用必须被修改，不过因为上下文代码必须将结果只作为 `Stream` 来处理，所以其不会注意到差别。

接口和方法

由于几乎任何事物都可以附加上方法，所以几乎任何事物都能够满足接口的要求。一个示例是在 `http` 程序包中，其定义了 `Handler` 接口。任何实现了 `Handler` 的对象都可以为HTTP请求提供服务。

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

`ResponseWriter` 本身是一个接口，提供了对用于向客户端返回响应的方法的访问。这些方法包括了标准的 `Write` 方法，所以任何可以使用 `io.Writer` 的地方，都可以使用 `http.ResponseWriter`。

简单起见，让我们忽略POST，假设HTTP请求总是GET；这种简化不影响建立处理的方式。这里有一个简单而完整的handler实现，用于计算页面的访问次数。

```
// 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)
}
```

(题外话，注意 `Fprintf` 是如何能够打印到 `http.ResponseWriter` 的。) 作为参考，下面给出了如何将该服务附加到URL树上的节点。

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

但是为什么 `Counter` 为一个结构体？只需要一个整数就可以了。（接收者需要为一个指针，这样增量才能对调用者可见。）

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

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

如果你的程序具有某个内部状态，当页面被访问时需要被告知，那么该如何？可以将一个channel绑定到网页上。

```
// 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，比方说我们想在 `/args` 上展现我们唤起服务二进制时所使用的参数。这很容易编写一个函数来打印参数。

```
func ArgServer() {
    fmt.Println(os.Args)
}
```

我们怎么将它转换成HTTP服务？我们可以将 `ArgServer` 创建为某个类型的方法，忽略该类型的值，不过有一种更干净的方式。既然我们可以为除了指针和接口以外的任何类型来定义方法，那么我们可以为函数编写一个方法。`http` 程序包包含了这样的代码：

```
// 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(w ResponseWriter, req *Request) {
    f(w, req)
}
```

`HandlerFunc` 为一个类型，其具有一个方法，`ServeHTTP`，所以该类型值可以为HTTP请求提供服务。看下该方法的实现：接收者为一个函数，`f`，并且该方法调用了 `f`。这看起来可能有些怪异，但是这与接收者为channel，方法在channel上进行发送数据并无差别。

要将 `ArgServer` 放到HTTP服务中，我们首先将其签名修改正确。

```
// Argument server.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}
```

`ArgServer` 现在具有和 `HandlerFunc` 相同的签名，所以其可以被转换为那个类型，然后访问它的方法，就像我们将 `Sequence` 转换为 `IntSlice`，来访问 `IntSlice.Sort` 一样。代码实现很简洁：

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

当有人访问页面 `/args` 时，在该页上安装的处理者就具有值 `ArgServer` 和类型 `HandlerFunc`。HTTP服务将会调用该方法 `ServeHTTP`，将 `ArgServer` 作为接收者，其将转而调用 `ArgServer`（通过在 `HandlerFunc.ServeHTTP` 内部调用 `f(c, req)`）。然后，参数就被显示出来了。

在这章节，我们分别通过结构体，整数，channel，以及函数创建了HTTP服务，这都是因为接口就是一个方法的集合，其可以针对（几乎）任何类型来定义。

空白标识符

- 空白标识符在多赋值语句中的使用
- 未使用的导入和变量
- 副作用式导入
- 接口检查

截至目前，我们已经两次提及“空白标识符”这个概念了，一次是在讲 `for range loops` 形式的循环时，另一次是在讲 `maps` 结构时。空白标识符可以赋值给任意变量或者声明为任意类型，只要忽略这些值不会带来问题就可以。这有点像在 Unix 系统中向 `/dev/null` 文件写入数据：它为那些需要出现但值其实可以忽略的变量提供了一个“只写”的占位符。但正如我们之前看到的那样，它实际的用途其实不止于此。

空白标识符在多赋值语句中的使用

空白标识符在 `for range` 循环中使用的其实是其应用在多语句赋值情况下的一个特例。

一个多赋值语句需要多个左值，但假如其中某个左值在程序中并没有被使用到，那么就可以用空白标识符来占位，以避免引入一个新的无用变量。例如，当调用的函数同时返回一个值和一个 `error`，但我们只关心 `error` 时，那么就可以用空白标识符来对另一个返回值进行占位，从而将其忽略。

```
if _, err := os.Stat(path); os.IsNotExist(err) {
    fmt.Printf("%s does not exist\n", path)
}
```

有时，你也会发现一些代码用空白标识符对 `error` 占位，以忽略错误信息；这不是一种好的做法。好的实现应该总是检查返回的 `error` 值，因为它会告诉我们错误发生的原因。

```
// Bad! This code will crash if path does not exist.
fi, _ := os.Stat(path)
if fi.IsDir() {
    fmt.Printf("%s is a directory\n", path)
}
```

未使用的导入和变量

如果你在程序中导入了一个包或声明了一个变量却没有使用的话，会引起编译错误。因为，导入未使用的包不仅会使程序变得臃肿，同时也降低了编译效率；初始化一个变量却不使用，轻则造成对计算的浪费，重则可能会引起更加严重 BUG。当一个程序处于开发阶段时，会存在一些暂时没有被使用的导入包和变量，如果为了使程序编译通过而将它们删除，那么后续开发需要使用时，又得重新添加，这非常麻烦。空白标识符为上述场景提供了解决方案。

以下一段代码包含了两个未使用的导入包（`fmt` 和 `io`）以及一个未使用的变量（`fd`），因此无法编译

本文档使用 [看云](#) 构建

译通过。我们可能希望这个程序现在就可以正确编译。

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
}
```

为了禁止编译器对未使用导入包的错误报告，我们可以用空白标识符来引用一个被导入包中的符号。同样的，将未使用的变量 `fd` 赋值给一个空白标识符也可以禁止编译错误。这个版本的程序就可以编译通过了。

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

var _ = fmt.Printf // For debugging; delete when done.
var _ io.Reader    // For debugging; delete when done.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
    _ = fd
}
```

按照约定，用来临时禁止未使用导入错误的全局声明语句必须紧随导入语句块之后，并且需要提供相应的注释信息——这些规定使得将来很容易找并删除这些语句。

副作用式导入

像上面例子中的导入的包，`fmt` 或 `io`，最终要么被使用，要么被删除：使用空白标识符只是一种临时性的举措。但有时，导入一个包仅仅是为了引入一些副作用，而不是为了真正使用它们。例如，`net/http/pprof` 包会在其导入阶段调用 `init` 函数，该函数注册HTTP处理程序以提供调试信息。这个包中确实也包含一些导出的API，但大多数客户端只会通过注册处理函数的方式访问web页面的数据，

而不需要使用这些API。为了实现仅为副作用而导入包的操作，可以在导入语句中，将包用空白标识符进行重命名：

```
import _ "net/http/pprof"
```

这种非常干净的导入包的方式，由于在当前文件中，被导入的包是匿名的，因此你无法访问包内的任何符号。（如果导入的包不是匿名的，而在程序中又没有使用到其内部的符号，那么编译器将报错。）

接口检查

正如我们在前面[接口](#)那章所讨论的，一个类型不需要明确的声明它实现了某个接口。一个类型要实现某个接口，只需要实现该接口对应的方法就可以了。在实际中，多数接口的类型转换和检查都是在编译阶段静态完成的。例如，将一个 `*os.File` 类型传入一个接受 `io.Reader` 类型参数的函数时，只有在 `*os.File` 实现了 `io.Reader` 接口时，才能编译通过。

但是，也有一些接口检查是发生在运行时的。其中一个例子来自 `encoding/json` 包内定义的 `Marshaler` 接口。当JSON编码器接收到一个实现了`Marshaler`接口的参数时，就调用该参数的 `marshaling`方法来代替标准方法处理JSON编码。编码器利用[类型断言](#)机制在运行时进行类型检查：

```
m, ok := val.(json.Marshaler)
```

假设我们只是想知道某个类型是否实现了某个接口，而实际上并不需要使用这个接口本身——例如在一段错误检查代码中——那么可以使用空白标识符来忽略类型断言的返回值：

```
if _, ok := val.(json.Marshaler); ok {
    fmt.Printf("value %v of type %T implements json.Marshaler\n", val, val)
}
```

在某些情况下，我们必须在包的内部确保某个类型确实满足某个接口的定义。例如类型

`json.RawMessage`，如果它要提供一种定制的JSON格式，就必须实现 `json.Marshaler` 接口，但是编译器不会自动对其进行静态类型验证。如果该类型在实现上没有充分满足接口定义，JSON编码器仍然会工作，只不过不是用定制的方式。为了确保接口实现的正确性，可以在包内部，利用空白标识符进行一个全局声明：

```
var _ json.Marshaler = (*RawMessage)(nil)
```

在该声明中，赋值语句导致了从 `*RawMessage` 到 `Marshaler` 的类型转换，这要求 `*RawMessage` 必须正确实现了 `Marshaler` 接口，该属性将在编译期间被检查。当 `json.Marshaler` 接口被修改后，上面的代码将无法正确编译，因而很容易发现错误并及时修改代码。

在这个结构中出现的空白标识符，表示了该声明语句仅仅是为了触发编译器进行类型检查，而非创建任何

新的变量。但是，也不需要对所有满足某接口的类型都进行这样的处理。按照约定，这类声明仅当代码中没有其他静态转换时才需要使用，这类情况通常很少出现。

内嵌 (Embedding)

Go没有提供经典的类型驱动式的派生类概念，但却可以通过内嵌其他类型或接口代码的方式来实现类似的功能。

接口的“内嵌”比较简单。我们之前曾提到过 `io.Reader` 和 `io.Writer` 这两个接口，以下是它们的实现：

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

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

在 `io` 包中，还提供了许多其它的接口，它们定义一类可以同时实现几个不同接口的类型。例如 `io.ReadWriter` 接口，它同时包含了 `Read` 和 `Write` 两个接口。尽管可以通过列出 `Read` 和 `Write` 两个方法的详细声明的方式来定义 `io.ReadWriter` 接口，但是以内嵌两个已有接口进行定义的方式会使代码显得更加简洁、直观：

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

这段代码的意义很容易理解：一个 `ReadWriter` 类型可以同时完成 `Reader` 和 `Writer` 的功能，它是这些内嵌接口的联合（这些内嵌接口必须是一组不相干的方法）。接口只能“内嵌”接口类型。

类似的想法也可以应用于结构体的定义，其实现稍稍复杂一些。在 `bufio` 包中，有两个结构体类型：`bufio.Reader` 和 `bufio.Writer`，它们分别实现了 `io` 包中的类似接口。`bufio` 包还实现了一个带缓冲的reader/writer类型，实现的方法是将reader和writer组合起来内嵌到一个结构体中：在结构体中，只列出了两种类型，但没有给出对应的字段名。

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

内嵌的元素是指向结构体的指针，因此在使用前，必须将其初始化并指向有效的结构体数据。结构体

`ReadWriter` 可以被写作如下形式：

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

为了使各字段对应的方法能满足 `io` 的接口规范，我们还需要提供如下的方法：

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

通过对结构体直接进行“内嵌”，我们避免了一些复杂的记录。所有内嵌类型的方法可以不受约束的使用，换句话说，`bufio.ReadWriter` 类型不仅具有 `bufio.Reader` 和 `bufio.Writer` 两个方法，同时也满足 `io.Reader`，`io.Writer` 和 `io.ReadWriter` 这三个接口。

在“内嵌”和“子类型”两种方法间存在一个重要的区别。当我们内嵌一个类型时，该类型的所有方法会变成外部类型的方法，但是当这些方法被调用时，其接收的参数仍然是内部类型，而非外部类型。在本例中，一个 `bufio.ReadWriter` 类型的 `Read` 方法被调用时，其效果和调用我们刚刚实现的那个 `Read` 方法是一样的，只不过前者接收的参数是 `ReadWriter` 的 `reader` 字段，而不是 `ReadWriter` 本身。

“内嵌”还可以用一种更简单的方式表达。下面的例子展示了如何将内嵌字段和一个普通的命名字段同时放在一个结构体定义中。

```
type Job struct {
    Command string
    *log.Logger
}
```

现在，`Job` 类型拥有了 `Log`，`Logf` 以及 `*log.Logger` 的其他所有方法。当然，我们可以给 `Logger` 提供一个命名字段，但完全没有必要这样做。现在，当初始化结束后，就可以在 `Job` 类型上调用日志记录功能了。

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

`Logger` 是结构体 `Job` 的一个常规字段，因此我们可以在 `Job` 的构造方法中按通用方式对其进行初始化：

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

或者写成下面的形式：

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

如果我们需要直接引用一个内嵌的字段，那么将该字段的类型名称省略了包名后，就可以作为字段名使用，正如之前在 `ReaderWriter` 结构体的 `Read` 方法中实现的那样。可以用 `job.Logger` 访问 `Job` 类型变量 `job` 的 `*log.Logger` 字段。当需要重新定义 `Logger` 的方法时，这种引用方式就变得非常有用了。

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

内嵌类型会引入命名冲突，但是解决冲突的方法也很简单。首先，一个名为 `x` 的字段或方法可以将其它同名的类型隐藏在更深层的嵌套之中。假设 `log.Logger` 中也包含一个名为 `Command` 字段或方法，那么可以用 `Job` 的 `Command` 字段对其访问进行封装。

其次，同名冲突出现在同一嵌套层里通常是错误的；如果结构体 `Job` 本来已经包含了一个名为 `log.Logger` 的字段或方法，再继续内嵌 `log.Logger` 就是不对的。但假设这个重复的名字并没有在定义之外的地方被使用到，就不会造成什么问题。这个限定为在外部进行类型嵌入修改提供了保护；如果新加入的字段和某个内部类型的字段有命名冲突，但该字段名没有被访问过，那么就不会引起任何问题。

并发

- [以通信实现共享](#)
- [Goroutines](#)
- [Channels](#)
- [Channel类型的Channel](#)
- [并行](#)
- [一个“Leaky Buffer”的示例](#)

以通信实现共享

并发程序设计是一个比较大的主题，这里我们只讨论一些Go语言特有的亮点。

由于需要考虑很多繁琐的细节以保证对共享变量访问的正确型，使得并发编程在很多情况下都会变得异常复杂。Go语言鼓励开发者采用一种不同的方法，即将共享变量通过Channel相互传递——事实上并没有真正在不同的执行线程间共享数据——的方式解决上述问题。在任意时刻，仅有一个Goroutine可以访问某个变量。数据竞争问题在设计上就被规避了。为了鼓励采用这种思维模式，我们将其总结为一句口号：

勿以共享方式通信，以通信实现共享。

这种方法还可以走得更远。举例而言，“引用计数”最好的实现途径可能就是通过在一个共享的整数周围加一个锁进行保护。但是在更高的层次，通过使用Channel控制共享整数访问可以更容易的写出整洁、正确的程序。

试着用下面的方法来分析上述模型：想象我们只是在处理传统的单线程程序，该程序仅运行在一个物理CPU上。基于这个前提进行开发，是无需提供任何同步原语的。现在，启动另一个类似的实例；它同样也不需要任何同步原语。然后让这两个实例进行通信；如果将通信本身算作一种同步原语，那么它是系统中仅有的同步原语。Unix操作系统的管道（Pipeline）就是上述模型的一个很好实例。尽管Go语言的并发模型源自Hoare的CSP模型（Communicating Sequential Processes, 国内译为“通信顺序进程”，台湾译为“交谈循序程序”），但它也可以被看成是一种类型安全的、一般化的Unix管道。

Goroutines

之所以称之为Goroutine，主要是由于现有的一些概念——“线程”、“协程”以及“进程”等——都不足以准确描述其内涵。每个Goroutine都对应一个非常简单的模型：它是一个并发的函数执行线索，并且在多个并发的Goroutine间，资源是共享的。Goroutine非常轻量，创建的开销不会比栈空间分配的开销大多少。并且其初始栈空间很小——这也就是它轻量的原因——在后续执行中，会根据需要在堆空间分配（或释放）额外的栈空间。

Goroutine与操作系统线程间采用“多到多”的映射方式，因此假设一个Goroutine因某种原因阻塞——

比如等待一个尚未到达的IO —— 其他Goroutine可以继续执行。我们在实现中屏蔽了许多底层关于线程创建、管理的复杂细节。

在一个函数或是方法前加上 `go` 关键字就可以创建一个Goroutine并调用该函数或方法。当该函数执行结束，Goroutine也随之隐式退出。（这种效果很像在Unix Shell里用 `&` 符号在后台启动一个命令。）

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

还可以将“函数文本”（function literals）嵌入到一个Goroutine创建之际，方法如下：

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }() // Note the parentheses - must call the function.
}
```

在Go中，这种“函数文本”的形式就是闭包（closure）：实现保证了在这类函数中被引用的变量在函数结束之前不会被释放。

以上的例子并不是很实用，因为执行函数无法发布其完成的信号。因此，我们还需要channel这一结构。

Channels

与map结构类似，channel也是通过 `make` 进行分配的，其返回值实际上是一个指向底层相关数据结构的引用。如果在创建channel时提供一个可选的整型参数，会设置该channel的缓冲区大小。该值缺省为0，用来构建默认的“无缓冲channel”，也称为“同步channel”。

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

无缓冲的channel使得通信——值的交换——和同步机制组合——共同保证了两个执行线索（Goroutines）运行于可控的状态。

对于channel，有很多巧妙的用法。我们通过以下示例开始介绍。上一节中，我们曾在后台发起过一个排序操作。通过使用channel，可以让发起操作的Goroutine等待排序操作的完成。

```
c := make(chan int) // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on the channel.
go func() {
    list.Sort()
    c <- 1 // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c // Wait for sort to finish; discard sent value.
```

接收方会一直阻塞直到有数据到来。如果channel是无缓冲的，发送方会一直阻塞直到接收方将数据取出。如果channel带有缓冲区，发送方会一直阻塞直到数据被拷贝到缓冲区；如果缓冲区已满，则发送方只能在接收方取走数据后才能从阻塞状态恢复。

带缓冲区的channel可以像信号量一样使用，用来完成诸如吞吐率限制等功能。在以下示例中，到来的请求以参数形式传入 `handle` 函数，该函数从channel中读出一个值，然后处理请求，最后再向channel写入以使“信号量”可用，以便响应下一次处理。该channel的缓冲区容量决定了并发调用 `process` 函数的上限，因此在channel初始化时，需要传入相应的容量参数。

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

func handle(r *Request) {
    <-sem          // Wait for active queue to drain.
    process(r)     // May take a long time.
    sem <- 1       // Done; enable next request to run.
}

func init() {
    for i := 0; i < MaxOutstanding; i++ {
        sem <- 1
    }
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req) // Don't wait for handle to finish.
    }
}
```

由于在Go中，数据同步发生在从channel接收数据阶段（也就是说，发送操作发生在接收操作之前，参见[Go内存模型](#)），因此获取信号量的操作必须实现在channel的接收阶段，而不是发送阶段。

这样的设计会引入一个问题：`Serve` 会为每个请求创建一个新的Goroutine，尽管在任意时刻只有最多 `MaxOutstanding` 个可以执行。如果请求到来的速度过快，将迅速导致系统资源完全消耗。我们可以通过修改 `Serve` 的实现来对Goroutine的创建进行限制。以下给出一个简单的实现，请注意其中包含一个BUG，我们会在后续进行修正：

```
func Serve(queue chan *Request) {
    for req := range queue {
        <-sem
        go func() {
            process(req) // Buggy; see explanation below.
            sem <- 1
        }()
    }
}
```

刚才说的BUG源自Go中 `for` 循环的实现，循环的迭代变量会在循环中被重用，因此 `req` 变量会在所有本文档使用 [看云](#) 构建

Goroutine间共享。这不是我们所乐见的，我们需要保证 req 变量是每个Goroutine私有的。这里提供一个方法，将 req 的值以参数形式提供给goroutine对应的闭包：

```
func Serve(queue chan *Request) {
    for req := range queue {
        <-sem
        go func(req *Request) {
            process(req)
            sem <- 1
        }(req)
    }
}
```

请与之前有BUG的实现进行对比，看看闭包在声明和运行上的不同之处。另一个解决方案是，干脆创建一个新的同名变量，示例如下：

```
func Serve(queue chan *Request) {
    for req := range queue {
        <-sem
        req := req // Create new instance of req for the goroutine.
        go func() {
            process(req)
            sem <- 1
        }()
    }
}
```

这样写可能看起来怪怪的

```
req := req
```

但它确实是合法的并且在Go中是一种惯用的方法。你可以如法炮制一个新的同名变量，用来为每个Goroutine创建循环变量的私有拷贝。

回到实现通用服务器的问题上来，另一个有效的资源管理途径是启动固定数量的 handle Goroutine，每个Goroutine都直接从channel中读取请求。这个固定的数值就是同时执行 process 的最大并发数。

Serve 函数还需要一个额外的channel参数，用来等待退出通知；当创建完所有的Goroutine之后，Server 自身阻塞在该channel上等待结束信号。

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

func Serve(clientRequests chan *Request, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
```

```

    }
    <-quit // Wait to be told to exit.
}

```

Channel类型的Channel

Channel在Go语言中是一个 first-class 类型，这意味着channel可以像其他 first-class 类型变量一样进行分配、传递。该属性的一个常用方法是用来实现安全、并行的解复用（demultiplexing）处理。

在上节的那个例子中，`handle` 是一个理想化的请求处理，但我们并没有定义处理的类型。如果处理的类型中包括一个用来响应的channel，则每个客户端可以提供其独特的响应方式。这里提供一个简单的

`Request` 类型定义：

```

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

```

客户端提供了一个函数及其参数，以及一个内部的channel变量用来接收回答消息。

```

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

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)

```

在服务器端，只有处理函数`handle`需要改变：

```

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

```

显然，上述例子还有很大的优化空间以提高其可用性，但是这套代码已经可以作为一类对速度要求不高、并行、非阻塞式RPC系统的实现框架了，而且实现中没有使用任何显式的互斥语法。

并行

上述这些想法的另一个应用场景是将计算在不同的CPU核心之间并行化，如果计算可以被划分为不同的可独立执行的部分，那么它就是可并行化的，任务可以通过一个channel发送结束信号。

假设我们需要在数组上进行一个比较耗时的操作，并且操作的值在每个数据上是独立的，正如下面这个理想化的例子一样：

```
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    // signal that this piece is done
}
```

我们在每个CPU上加载一个循环无关的迭代计算。这些计算可能以任意次序完成，但这是无关紧要的；我们仅需要在创建完所有Goroutine后，从channel中读取结束信号进行计数即可。

```
const NCPU = 4 // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, NCPU) // Buffering optional but sensible.
    for i := 0; i < NCPU; i++ {
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < NCPU; i++ {
        <-c    // wait for one task to complete
    }
    // All done.
}
```

在目前的Go runtime 实现中，这段代码在默认情况下是不会被并行化的。对于用户态任务，我们默认仅提供一个物理CPU进行处理。任意数目的Goroutine可以阻塞在系统调用上，但默认情况下，在任意时刻，只有一个Goroutine可以被调度执行。我们未来可能会将其设计的更加智能，但是目前，你必须通过设置 GOMAXPROCS 环境变量或者导入 runtime 包并调用 runtime.GOMAXPROCS(NCPU)，来告诉Go的运行系统最大并行执行的Goroutine数目。你可以通过 runtime.NumCPU() 获得当前运行系统的逻辑核数，作为一个有用的参考。需要重申：上述方法可能会随我们对实现的完善而最终被淘汰。

注意不要把“并发”和“并行”这两个概念搞混：“并发”是指用一些彼此独立的执行模块构建程序；而“并行”则是指通过将计算任务在多个处理器上同时执行以提高效率。尽管对于一些问题，我们可以利用“并发”特性方便的构建一些并行的程序部件，但是Go终究是一门“并发”语言而非“并行”语言，并非所有的并行编程模式都适用于Go语言模型。要进一步区分两者的概念，请参考[这篇博客](#)的相关讨论。

一个“Leaky Buffer”的示例

并发编程的工具甚至可以更简单的表达一些非并发的想法。下面提供一个示例，它是从RPC的一个包里抽象而来的。客户端从某些源——比如网络——循环接收数据。为了避免频繁的分配、释放内存缓冲，程序在内部实现了一个空闲链表，并用一个Buffer指针型channel将其封装。当该channel为空时，程序为其分配一个新的Buffer对象。一旦消息缓冲就绪，它就会被经由 serverChan 发送到服务器端。

```

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

func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not.
        select {
        case b = <-freeList:
            // Got one; nothing more to do.
        default:
            // None free, so allocate a new one.
            b = new(Buffer)
        }
        load(b)           // Read next message from the net.
        serverChan <- b   // Send to server.
    }
}

```

服务器端循环从客户端接收并处理每个消息，然后将Buffer对象返回到空闲链表中。

```

func server() {
    for {
        b := <-serverChan // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
        case freeList <- b:
            // Buffer on free list; nothing more to do.
        default:
            // Free list full, just carry on.
        }
    }
}

```

客户端会尝试从空闲链表 `freeList` 中获取Buffer对象；如果没有可用对象，则分配一个新的。服务器端会将用完的Buffer对象 `b` 加入到空闲链表 `freeList` 中，如果链表已满，则将 `b` 丢弃，垃圾收集器会在未来某个时刻自动回收对应的内存单元。（`select` 语句中的 `default` 分支会在没有其他case分支满足条件时执行，这意味着 `select` 语句块不会阻塞。）以上就是一个 Leaky Bucket Free List 的简单实现，借助Go语言中带缓冲的channel以及“垃圾收集”机制，我们仅用几行代码就将其搞定了。

错误

- [严重故障 \(Panic\)](#)
- [恢复 \(Recover\)](#)

向调用者返回某种形式的错误信息是库历程必须提供的一项功能。通过前面介绍的函数多返回值的特性，Go中的错误信息可以很容易同正常情况下的返回值一起返回给调用者。方便起见，错误通常都用内置接口 `error` 类型表示。

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

库开发人员可以通过实现该接口来丰富其内部功能，使其不仅能够呈现错误本身，还能提供更多的上下文信息。举例来说，`os.Open` 函数会返回 `os.PathError` 错误。

```
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error      // Returned by the system call.
}

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

`PathError` 的 `Error` 方法会生成类似下面给出的错误信息：

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

这条错误信息包括了足够的信息：出现异常的文件名，操作类型，以及操作系统返回的错误信息等，因此即使它冒出来的时候距离真正错误发生时刻已经间隔了很久，也不会给调试分析带来很大困难，比直接输出一句 “no such file or directory” 要友好的多。

如果可能，描述错误的字符串应该能指明错误发生的原始位置，比如在前面加上一些诸如操作名称或包名称的前缀信息。例如在 `image` 包中，用来输出未知图片类型的错误信息的格式是这样的：“image: unknown format”。

对于需要精确分析错误信息的调用者，可以通过类型开关或类型断言的方式查看具体的错误并深入错误的细节。就 `PathErrors` 类型而言，这些细节信息包含在一个内部的 `Err` 字段中，可以被用来进行错误恢

复。

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

在上面例子中，第二个 `if` 语句是另一种形式的[类型断言](#)。如该断言失败，`ok` 的值将为`false`且 `e` 的值为 `nil`。如果断言成功，则 `ok` 值为`true`，说明当前的错误，也就是 `e`，属于 `*os.PathError` 类型，因而可以进一步获取更多的细节信息。

严重故障 (Panic)

通常来说，向调用者报告错误的方式就是返回一个额外的 `error` 变量：`Read` 方法就是一个很好的例子；该方法返回一个字节计数值和一个 `error` 变量。但是对于那些不可恢复的错误，比如错误发生后程序将不能继续执行的情况，该如何处理呢？

为了解决上述问题，Go语言提供了一个内置的 `panic` 方法，用来创建一个运行时错误并结束当前程序（关于退出机制，下一节还有进一步介绍）。该函数接受一个任意类型的参数，并在程序挂掉之前打印该参数内容，通常我们会选择一个字符串作为参数。方法 `panic` 还适用于指示一些程序中的不可达状态，比如从一个无限循环中退出。

```
// A toy implementation of cube root using Newton's method.
func CubeRoot(x float64) float64 {
    z := x/3 // Arbitrary initial value
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // A million iterations has not converged; something is wrong.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

以上仅仅提供一个应用的示例，在实际的库设计中，应尽量避免使用 `panic`。如果程序错误可以以某种方式掩盖或是绕过，那么最好还是继续执行而不是让整个程序终止。不过还是有一些反例的，比方说，如果库历程确实没有办法正确完成其初始化过程，那么触发 `panic` 退出可能就是一种更加合理的方式。

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

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

恢复 (Recover)

对于一些隐式的运行时错误，如切片索引越界、类型断言错误等情形下，`panic` 方法就会被调用，它将立刻中断当前函数的执行，并展开当前Goroutine的调用栈，依次执行之前注册的defer函数。当栈展开操作达到该Goroutine栈顶端时，程序将终止。但这时仍然可以使用Go的内建 `recover` 方法重新获得Goroutine的控制权，并将程序恢复到正常执行的状态。

调用 `recover` 方法会终止栈展开操作并返回之前传递给 `panic` 方法的那个参数。由于在栈展开过程中，只有defer型函数会被执行，因此 `recover` 的调用必须置于defer函数内才有效。

在下面的示例应用中，调用 `recover` 方法会终止server中失败的那个Goroutine，但server中其它的Goroutine将继续执行，不受影响。

```
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)
}
```

在这里例子中，如果 `do(work)` 调用发生了panic，则其结果将被记录且发生错误的那个Goroutine将干净的退出，不会干扰其他Goroutine。你不需要在defer指示的闭包中做别的操作，仅需调用 `recover` 方法，它将帮你搞定一切。

只有直接在defer函数中调用 `recover` 方法，才会返回非 `nil` 的值，因此defer函数的代码可以调用那些本身使用了 `panic` 和 `recover` 的库函数而不会引发错误。还用上面的那个例子说明：`safelyDo` 里的defer函数在调用 `recover` 之前可能调用了日志记录函数，而日志记录程序的执行将不受panic状态的影响。

有了错误恢复的模式，`do` 函数及其调用的代码可以通过调用 `panic` 方法，以一种很干净的方式从错误状态中恢复。我们可以使用该特性为那些复杂的软件实现更加简洁的错误处理代码。让我们来看下面这个例子，它是 `regexp` 包的一个简化版本，它通过调用 `panic` 并传递一个局部错误类型来报告“解析错误”（Parse Error）。下面的代码包括了 `Error` 类型定义，`error` 处理方法以及 `Compile` 函数：

```

// Error is the type of a parse error; it satisfies the error interface.
type Error string
func (e Error) Error() 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 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` 方法触发 `panic`，错误恢复代码会将返回值置为 `nil`——因为 `defer` 函数可以修改命名的返回值变量；然后，错误恢复代码会对返回的错误类型进行类型断言，判断其是否属于 `Error` 类型。如果类型断言失败，则会引发运行时错误，并继续进行栈展开，最后终止程序——这个过程将不再会被中断。类型检查失败可能意味着程序中还有其他部分触发了 `panic`，如果某处存在索引越界访问等，因此，即使我们已经使用了 `panic` 和 `recover` 机制来处理解析错误，程序依然会异常终止。

有了上面的错误处理过程，调用 `error` 方法（由于它是一个类型的绑定的方法，因而即使与内建类型 `error` 同名，也不会带来什么问题，甚至是一直更加自然的用法）使得“解析错误”的报告更加方便，无需费心去考虑手工处理栈展开过程的复杂问题。

```

if pos == 0 {
    re.error("' ' illegal at start of expression")
}

```

上面这种模式的妙处在于，它完全被封装在模块的内部，`Parse` 方法将其内部对 `panic` 的调用隐藏在 `error` 之中；而不会将 `panics` 信息暴露给外部使用者。这是一个设计良好且值得学习的编程技巧。

顺便说一下，当确实有错误发生时，我们习惯采取的“重新触发 `panic`”（`re-panic`）的方法会改变 `panic` 的值。但新旧错误信息都会出现在崩溃报告中，引发错误的原始点仍然可以找到。所以，通常这种简单的重新触发 `panic` 的机制就足够了——所有这些错误最终导致了程序的崩溃——但是如果只想显示最初的错误信息的话，你就需要稍微多写一些代码来过滤掉那些由重新触发引入的多余信息。这个功能就留给读者自己去实现吧！

错误

一个web服务示例

让我们以一个完整的Go程序示例——一个web服务——来作为这篇文档的结尾。事实上，这个例子其实是一类“Web re-server”，也就是说它其实是对另一个Web服务的封装。谷歌公司提供了一个用来自动将数据格式化为图表或图形的在线服务，其网址是：<http://chart.apis.google.com>。这个服务使用起来其实有点麻烦——你需要把数据添加到URL中作为请求参数，因此不易于进行交互操作。我们现在的这个程序会为用户提供一个更加友好的界面来处理某种形式的数据：对于给定的 一小段文本数据，该服务将调用图标在线服务来产生一个QR码，它用一系列二维方框来编码文本信息。可以用手机摄像头扫描该QR码并进行交互操作，比如将 URL地址编码成一个QR码，你就省去了往手机里输入这个URL地址的时间。

下面是完整的程序代码，后面会给出详细的解释。

```
package main

import (
    "flag"
    "html/template"
    "log"
    "net/http"
)

var addr = flag.String("addr", ":1718", "http service address") // Q=17, R=18

var templ = template.Must(template.New("qr").Parse(templateStr))

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

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

const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{{if .}}

<br>
{{.}}
<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` 函数之前的部分很容易理解。包`flag`用来构建我们这个服务默认的HTTP端口。从模板变量 `templ` 开始进入了比较好玩的部分，它的功能是用来构建一个HTML模板，该模板被我们的服务器处理并用来显示页面信息；我们后面还会看到更多细节。

`main` 函数使用我们之前介绍的机制来解析`flag`，并将函数 `QR` 绑定到我们服务的根路径。然后调用 `http.ListenAndServe` 方法启动服务；该方法将在服务器运行过程中一直处于阻塞状态。

`QR` 函数用来接收包含格式化数据的请求信息，并以该数据 `s` 为参数对模板进行实例化操作。

模板包 `html/template` 的功能非常强大；上述程序仅仅触及其冰山一角。本质上说，它会根据传入 `templ.Execute` 方法的参数，在本例中是格式化数据，在后台替换相应的元素并重新生成HTML文本。在模板文本（`templateStr`）中，双大括号包裹的区域意味着需要进行模板替换动作。在 `{{if .}}` 和 `{{end}}` 之间的部分只有在当前数据项，也就是 `.`，不为空时才被执行。也就是说，如果对应字符串为空，内部的模板信息将被忽略。

代码片段 `{{.}}` 表示在页面中显示传入模板的数据 —— 也就是查询字符串本身。HTML模板包会自动提供合适的处理方式，使得文本可以安全的显示。

模板串的剩余部分就是将被加载显示的普通HTML文本。如果你觉得这个解释太笼统了，可以进一步参考 [Go文档](#)中，关于模板包的深入讨论。

看，仅仅用了很少量的代码加上一些数据驱动的HTML文本，你就搞定了—一个很有用的web服务。这就是Go语言的牛X之处：用很少的一点代码就能实现很强大的功能。

中文译者

按拼音排序：

Amal Cao <amalcaowei AT gmail.com>

邢明杰 <mingjie.xing AT gmail.com>