Effective Go几乎是学习Go语言所必须阅读的重要的文档，以下是本人对该文档的翻译。由于涉及内容较多，翻译过程中不可避免地会产生一些错误，希望读到的朋友在评论中指出。随着Go新版本的发布，我将继续保持此文档的更新。

最后更新时间：2014/07/13 10:19

介绍

Go是一个新语言，尽管它从已有的语言中借用了一些概念，但是Go语言独有的特征使实际的Go程序与其他语言编写的程序不尽相同。将C++或Java程序直译为Go程序将无法得到满意的结果——Java程序是用Java所写，而不是Go。另一方面，以Go的视角考虑问题能产生一个成功的但也相当不同的程序。换句话说，要想写好Go程序，必须要理解其特征和习惯用法。同样重要的是了解Go既定的规范，如命名、格式化、程序结构等等，这样你所编写的程序将能很容易地被其他Go程序员所理解。

此文档将就如何编写清晰的、符合语言习惯的Go程序给出一些提示。它是对Go语言规范、Go语言之旅以及如何编写Go程序的补充，在阅读本文之前，你应该先阅读这些文档。

示例

Go语言包源代码不仅是核心库，同时也是关于演示如何使用此语言的示例代码。并且许多包还包含可独立运行的示例，你可以直接从golang.org网站运行他们，例如这个例子（点击“Example”打开它）。 如果你有关于如何解决问题和某些东西如何实现方面的疑问，他们可以给出答案、思路和背景。

格式化

格式化总是最易引起争论但很难争论出结果。人们总是需要适应多种不同的格式化样式，但如果所有人都遵循同一种样式，那么在该议题上将花费更少的时间，这样或许更好一点。问题在于如何实现这个理想并且不需要一个冗长的样式说明手册。

在Go中我们可以使用一个特别的方法，即让机器来处理大部分的格式化问题。 `gofmt` 程序（也可以使用 `go fmt` ，它在包的级别上而非源文件的级别上进行操作）读取一个Go程序并生成标准样式的源代码，这些样式调整包括缩进、垂直对齐、保留注释并在需要时重新格式化注释。如果你想要知道如何处理一些新的布局情况，请运行 `gofmt` ，如果结果看起来不太对，请重新调整你的程序（或提交一个关于 `gofmt` 的bug），不要自个去调整 `gofmt` 代码。

举例来说，对于结构体中的注释，并不需要花费时间将他们对齐。 `gofmt` 将为你做这些事。对于如下的声明

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

`gofmt` 将对齐各列：

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

在标准包中的所有Go代码已经使用 `gofmt` 格式化过了。

还有一些格式化细节。这些都非常简明

- 缩进
  我们使用制表符(tab)进行缩进， `gofmt` 生成的代码默认也使用它。你也可以特意地使用空格缩进。
- 行长
  Go没有行长限制。不要担心代码会在穿孔卡片上溢出。如果一行实在有点太长，可以进行换行并使用一个额外的制表符将其缩进。
- 圆括号
  与C和Java相比，Go很少使用圆括号：控制结构（ `if` 、 `for` 和 `switch` ）在语法上不需要圆括号。另外，操作符优先级别变得更短更清楚，因此

  x<<8 + y<<16

所表达的意思和其中空格暗示的一样，这一点与其他语言也不相同。

注释

Go提供了C样式的 `/* */` 块注释和C++样式的 `//` 行注释。行注释是标准的；块注释主要用于包注释，但也用于表达式内部或禁用一大段代码。

`godoc` 是一个程序，也是一个网络服务器，它对Go的源代码文件进行处理，提取其中的包文档内容。出现在顶层声明前方，中间不包括空行的注释与声明一起提取出来作为一个所声明项的解释文本。这些注释的内容和样式决定了 `godoc` 生成的文档的质量。

每个包都应该有一个*包注释*，它是一个放在包语句（package）前方的块注释。对于包含多个文件的包，包注释只需要出现在一个文件中，任何一个文件都可以。包注释应该在整体上对此包进行介绍，并提供包的相关信息。它将在 `godoc` 页面中率先出现并为其下的内容建立详细的文档。

```
/*
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
```

如果包很简单，包的注释可以很简短。

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

注释不需要额外的格式化，如星号横幅。生成的输出甚至可能无法以固定宽度的字体显示，因此不要依赖空格进行对齐，就像 `gofmt` 一样， `godoc` 会自动对这些进行处理。注释是不被解析的纯文本，HTML或其他类似的东西如 `_this_` 将*照原样*输出，因此不应该使用他们。 `godoc` 所做的另一项工作是以固定宽度字体显示缩进的文本，因此可以用缩进文本来显示代码片段。fmt包的注释就利用了此项功能达到了好的显示效果。

根据上下文， `godoc` 甚至可能不会重新格式化注释，因此必须保证他们看上去是直白的：使用正确的拼写、标点和句子结构，折叠长行，等等。

在一个包内，任何直接放在顶层声明前方的的注释都被认为是该声明的*文档注释*。在一个程序中每个导出的（首字母大写）名称都应该有一个文档注释。

文档注释最好是一个完整的句子，这样它就适用于各种自动化的展示。第一个句子应该是一个概括性的单句，并以被声明事物的名称起头。

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

如果注释总是以被声明事物的名称起头，就可以在 `godoc` 的输出中使用 `grep` 命令。假设你不记得 `Compile` 这个名称，但知道自己在找正则表达式的parsing功能，因此可以用以下方法运行命令，

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

要是包中的所有文档注释都以“This function...”开头， `grep` 就不会帮你找到想要的函数名称。但若每个文档注释都以名称起头，你将会看到如下结果，它使你回想起你正要找的东西。

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

Go的声明语法允许进行成组声明。一个单一的文档注释可以介绍一组相关的常量或变量。由于是进行整体声明，这种注释往往是概括性的。

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

同时也可以使用成组声明来表明项目之间的关系，例如，一组被一个互斥对象保护着的变量。

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

命名

命名在Go中与在其他语言中一样重要。在一些情况下，他们甚至对语义有影响：例如，一个名称在包外的可见性是由其首个字符是否为大写字母决定的。因此有必要花点时间来讨论Go语言中的命名约定。

包的名称

当一个包被导入时，包的名称变成其内容的访问器。在

```
import "bytes"
```

之后，被导入的包可以使用 bytes.Buffer 。所有使用此包的人都以一个相同的名称引用其内容将是非常有好处的，这意味着应该给包起一个好名称：简短、简明且容易理解。按照常规，包的名称是小写的、单个单词的名称；并不需要下划线或大小写混写。 err 的命名就是出于简洁考虑的，由于任何使用你的包的人都将键入其名称。不必担心其与*已有的东西*冲突。包的名称是仅有的需要导入的默认名称；它并不要求在整个源代码中都是独一无二的，即便在少数发生冲突的情况，也可以将包以一个不同的名称导入以便局部使用。在所有情况下，由于可根据文件名判定所使用的是哪个包，因此不会造成混淆。

另一个命名常规就是包名应该是其源代码所在目录的基本名称（译注：去掉路径中最后一个 / 之前所有东西之后所剩的名称）；在 src/pkg/encoding/base64 中的包以 encoding/base64 导入，但其名称应是 base64 ，而不是 encoding_base64 或 encodingBase64 。

包导入器将使用包的名称引用其内容，因此包的导出名称可以借此避免冲突。（不要使用 import . ，因为该方法主要用于以简化的方法运行所要测试的外部包。）例如， bufio 包中的缓冲区读取器叫做 Reader 而不是 BufReader ，因为其用户看到的是 bufio.Reader ，这已经是一个清晰、简明的名称了。另外，由于导入的项目总是用他们的包名限定， bufio.Reader 就不会与 io.Reader 发生冲突。同样地，生成了 ring.Ring 的新实例的函数——这就是Go中的*构造器*——通常可以命名为 NewRing ，但由于 Ring 是此包所导出的唯一类型，并且包的名称就叫做 ring ，该构造器可以被命名为 New ，它跟在包的后面，如 ring.New 。使用包结构可以帮助你选择好的名字。

另一个短的示例是 once.Do ； once.Do(setup) 读起来很顺，将其写成 once.DoOrWaitUntilDone(setup) 也并不会更好。长名称不会自动使事物变得更有可读性。如果名称代表的事物比较复杂且难以琢磨，更好的方法是写一个有用的文档注释而不是使用一个特别加长的名称。

Getter

Go不对读取器（getter）和写入器（setter）提供自动支持。若你要自己提供getter和setter不仅没有什么不对，并且往往是恰当的，但要将 Get 放入getter名称中既不合常规也不必要。如果你有一个叫做 owner （小写，不可导出的）的字段，其getter方法应该是 Owner （首字母大写，可导出的），而不是 GetOwner 。使用大写字母名称导出提供了辨别字段和方法的钩子。如果需要，一个setter函数应该类似 SetOwner 。在实际中，两种名称读起来都很好：

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

接口的名称

按照约定，仅一个方法的接口名称以方法名加 -er 后缀命名，或通过相似的修改来构建一个代理名词，如 Reader 、 Writer 、 Formatter 、 CloseNotifier 等等。

这类名称有很多，用这种方法来表示他们自身以及他们所代表的函数名都是非常高效的。 `Read` 、 `Write` 、 `Close` 、 `Flush` 、 `String` 等都具有规范的签名和意义。为避免混淆，你的方法的名称不应该与这些名称一样，除非他们具有同样的签名和意义。相反地，如果你的类型实现了一个与这些熟知类型同样意义的方法，请保持他们的名称和签名相同；如将你的字符串转换器方法命名为 `String` 而不是 `ToString` 。

大小写混写

最后，Go的一项约定是在写多个单词的名称时，使用 `MixedCaps` 或 `mixedCaps` 而不是用下划线分割。

分号

和C一样，Go的正式语法使用分号来终止语句；和C不同的是，这些分号不在源代码中出现。取而代之的是，词法分析器在扫描过程中使用简单的规则自动插入分号，因此输入文本多数时候就不需要分号了。

规则是这样的：如果在一个换行符前方的最后一个标记是一个标识符（包括像 `int` 和 `float64` 这样的单词）、一个基本的如数值这样的文字、或以下标记中的一个

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

词法分析器将始终在此标记后面插入分号。这一点可概括为，“如果换行符前方的标记可能是语句的末尾，则插入分号”。

在右大括号前方的分号也可以省略，因此一个如下形式的语句

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

是不需要分号的。通常Go程序仅需在 `for` 循环语句中使用分号，以此来分开初始化器、条件和增量单元。如果你在一行中写多个语句，也需要用分号分开。

这样的分号插入规则导致一种后果，即你不能将一个控制结构（ `if` 、 `for` 、 `switch` 或 `select` ）的左大括号放在下一行。如果这样做，将会在大括号的前方插入一个分号，这可能导致出现不想要的结果。你应该这样写

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

而不是这样写

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

控制结构

Go的控制结构与C有关，但在一些重要的方面又有所不同。其中没有 `do` 或 `while` 循环，而仅有一个更广义的 `for` ； `switch` 要更灵活一点； `if` 和 `switch` 像 `for` 一样接受可选的初始化语句； `break` 和 `continue` 能可选地接收一个标签以辨别要终止或继续什么；另外还有一个包含一个类型切换和一个多路通信复用器的新控制结构， `select` 。其语法也稍微有点不同：没有圆括号，而其主体必须始终用大括号包括着。

If

以下是Go的一个简单的If语句：

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

强制大括号更容易写出简单的多行 `if` 语句。这是一种好的风格，尤其是当主体包含如 `return` 或 `break` 这样的控制结构时。

由于 `if` 和 `switch` 接受初始化语句，经常会看到将其用于建立局部变量。

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

在Go的库中，你将发现当 `if` 语句没有执行到其下面的语句时——也就是说，其主体以 `break` 、 `continue` 、 `goto` 或 `return` 结束——不必要的 `else` 是被省略的。

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

代码必须防止一系列的错误条件，以下就是一个常见情况的示例。若非出错，控制流能成功地向下执行，这些代码都读起来也很顺畅。由于出错时会在 `return` 语句中终止，最终代码就不需要 `else` 语句了。

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

重复声明和重复赋值

顺便说一下：上一节的最后一个例子展示了简短形式的声明 `:=` 的详细工作方式。调用了 `os.Open` 的声明的语句为

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

该语句声明了两个变量 `f` 和 `err` 。几行之后，又通过以下语句调用了 `f.Stat`

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

该语句表面上是声明了 `d` 和 `err` 。注意，在两个语句中都出现了 `err` 。这种重复是合法的： `err` 在第一个语句中被声明，而在第二个语句中仅仅被*重新赋值*。这意味着对 `f.Stat` 的调用使用的是前面已经声明的 `err` 变量，这里只是给它一个新的值而已。

在满足以下条件时，变量 `v` 可出现在 `:=` 声明中，即便是该变量已经被声明过了：

- 该声明的作用域与已有的 `v` （如果 `v` 已经在更靠外一级的作用域内被声明，则此声明会创建一个新的变量 §）的声明的作用域相同，
- 对 `v` 所赋的值是类型匹配的，并且
- 在声明中至少有另外一个变量是新声明的。

这种不寻常的特性纯粹是出于实用主义，这使我们可以很方便地只使用一个 `err` 值，比如用于一个长 `if-else` 语句链中。这种用法会经常被看到。

§ 值得注意的是，在Go中，尽管函数参数、返回值在词法上出现在大括号的外面，而函数体则包含在大括号内部，但他们的作用域一样的。

For

Go的 for 循环与C的相似但却不一样。它统一了 for 和 while ，因此也就不再有 do-while 了。 for 语句有三种形式，只有一种具有分号。

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

// 和 C 的 while 语句类似
for condition { }

// 和 C 的 for(;;) 语句类似
for { }
```

使用缩短的声明语句能更轻易地在循环中声明索引变量。

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

如果你是在一个数组、切片、字符串或映射内进行循环遍历，或读取一个信道，可使用一个 range 从句来管理循环。

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

如果你只需要 range 返回的第一个项目（键或索引），去掉第二个就行了：

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

如果你只需要 range 返回的第二个项目（值），可使用空白标识符（一个下划线）来抛弃第一个：

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

空白标志符很有用，在稍后的一节中将对其讲解。

对于字符串， `range` 将为你做更多的工作，它可解析UTF-8并分解成单个Unicode代码点。错误的编码占用一个字节并用一个rune(有时被译作“符文”)字符U+FFFD替代。（ `rune` （它对于一个内建类型）名称是一个Go的术语，它代表单个Unicode代码点。参见语言规范。）以下循环

```
for pos, char := range "日本\x80語" { // \x80 是一个非法的 UTF-8 编码
    fmt.Printf("character %#U starts at byte position %d\n", char, pos)
}
```

将打印

```
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没有逗号运算符， `++` 和 `--` 是语句而非表达式。因此如果你想要在 `for` 中使用多变量，你应该使用并列赋值（这种情况无法使用 `++` 和 `--` ）。

```
// 反转 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更常见。其表达式不需要是常量甚至是整数，从上到下对每个分支的值进行比较，直到发现一个匹配的值，如果 `switch` 中没有表达式，它将匹配 `true` 。因此可能——也是常常——将一个 `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
}
```

这里没有自动的向下贯穿（译注：fall through，即找到一个分支入口后不再进行判断而执行其下面的分支），但多个分支可以通过以逗号分割的列表来呈现。

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

仍然可以使用 `break` 语句来提早结束一个 `switch` ，但Go中这样用不如其他类C语言那么普遍。有时只是需要break其所包围的循环，而不是switch语句，在Go中可以通过给循环一个标签，并且break到这个标签。下例同时显示了这两方面的用法。

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

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

当然， `continue` 语句同样可接受这样一个可选的标签，但它只能用于循环。

以下程序通过使用两个 `switch` 语句对字节数组进行对比：

```
// Compare 按照字典顺序比较两个字节切片.
// 若 a == b，返回 0；若 a < b，返回 -1；如果 a > b，返回 +1
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
}
```

类型切换

可以使用 switch 去发现一个接口变量的动态类型。这种*类型切换*使用放在圆括号内的关键字 type 实现类型断言语法。如果在开关的表达式中声明了一个变量，在每个从句中将有该变量对应的类型。在这种情况下通常会重用变量的名称，即声明一个具有同样名称但却有不同类型的变量。

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

函数

多个返回值

Go的一个特有性质就是函数和方法具有多个返回值。这种特性使C程序中各种笨拙习惯用法得以改善：带内（译注：同一个返回变量内）返回错误（例如 -1 代表 EOF ）和通过传递地址修改一个参量。

在C中，一个写错误是使用一个负数来标志，该错误代码隐藏在另外的不确定的位置。在Go中， Write 可以返回一个数值*和*一个错误：“是的，您写入了一些字节，但并没有全部写入，因为设备已满”。在 os 包中 Write 方法的签名是：

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

正如文档所述，当 n != len(b) 时，它返回被写入的字节的数目以及一个非 nil 的 error ；这是一个常用的方式；参见错误处理一节获得更多示例。

以往一般通过传递一个指针到一个返回值以模拟引用参数，现在一个相似的方法使这样不再必须。以下简单的函数从字节切片的特定位置获取一个数字，它返回该数字和下一个位置。

```
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，例如：

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

带名称的结果参数

Go函数的返回值或结果“参数”可以给定名称并像常规的变量那样使用，就像接收的参数那样。命名后，一旦函数开始，他们就被初始化为其类型对应的零值；如果函数中的 return 语句不带参量，结果参数的当前值将作为返回值返回。

此名称并不是强制要求的，但它能使代码变得更加简短和清晰：他们就是文档。如果我们命名了 nextInt 的返回值，就能很容易地知道各个返回的 int 所代表的意思。

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

由于被命名的返回结果被初始化并可与一个不带参数的 return 绑定，他们不仅可使代码变得清晰，也可使代码简化。这里的 io.ReadFull 是使用他们的一个很好的范例：

```
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
}
```

Defer

Go的 defer 语句预设一个函数调用（*延期的*函数），该调用在函数执行 defer 返回时立刻运行。该方法显得不同常规，但却是处理一些情况的有效方式，如无论函数怎样返回，都必须进行资源释放。典型的例子是解开一个互斥锁并关闭文件。

```
// Contents 以字符串形式返回文件的内容.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // 当结束时将运行 f.Close.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append 将在随后讨论.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // 如果在此处返回， f 将被关闭.
        }
    }
    return string(result), nil // 如果在此处返回， f 将被关闭.
}
```

对像 Close 这样的函数的延期调用有两个优点。第一，它确保你不会忘记关闭文件，在一段时间之后编辑函数以便向其中添加新的返回路径时，往往会发生此种错误。第二，它意味着关闭与打开靠得

很近，这要比将关闭放在函数结尾处更为清楚明了。

被延期函数的参量（如果函数是一个方法，将还包括接收者）是在进行*延期*时被估值，而不是在*调用*时被估值。这样不仅可不必担心变量值被改变，同时也意味着单个延期调用可以延期多个函数执行。以下是一个不太聪明的例子：

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

被延期的函数以后进先出（LIFO）的顺行执行，因此以上代码在返回时将打印 `4 3 2 1 0` 。一个更合理的例子是用一种简单的方法通过程序追踪函数调用。我们能以如下方式写一些简单的追踪例程：

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

// 以如下方法使用他们:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

我们可以通过利用被延期函数的参量在 `defer` 执行时被估值的特点更好地完成工作。追踪例程可以针对非追踪例程建立参量。如下例所示：

```
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()
}
```

此程序将打印

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

对于习惯于其他语言的块级资源管理的程序员， `defer` 看起来有点怪异。但它最有趣和强大的应用恰恰来自于它是基于函数而不是基于块的特点。在 `panic` 和 `recover` 节中我们将看到它的另一种应用的例子。

数据

使用 new 分配内存

Go具有两种分配内存的机制，分别是内建的函数 `new` 和 `make` 。他们所做的事不同，所应用到的类型也不同，这可能引起混淆，但规则却很简单。让我们先讨论 `new` 。它是一个分配内存的内建函数，但不同于其他语言中同名的 `new` 所作的工作，这里它只是将内存*清零*，而不是*初始化*内存。 `new(T)` 为一个类型为 `T` 的新项目分配了调到零值的存储空间并返回其地址，也就是一个类型为 `*T` 的值。用Go的术语来说，就是它返回了一个指向新分配的类型为 `T` 的零值的指针。

由于由 `new` 返回的内存中的值是零，这样就更便于设计数据结构，因为每个类型的零值不必进一步进行初始化就已可以使用。这意味着数据结构的用户在使用 `new` 创建数据后就立刻可使用它。例如， `bytes.Buffer` 的文档这样表述，“零值的 `Buffer` 是一个已准备就绪的空缓冲器。”同样地， `sync.Mutex` 不具有一个显式的构造器或 `Init` 方法，但零值的 `sync.Mutex` 已经是一个解开锁定的互斥锁了。

零值属性是可以传递的，这一点很有用。考虑以下的类型声明。

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

类型 SyncedBuffer 的值同样也是在声明时就分配好内存并准备就绪的。在下一个程序片段中， p 和 v 不需要处理就可以正确地工作。

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

构造器和复合文字

有时零值并不足够好，这就需要一个初始化构造器，如来自 os 包的这段代码所示。

```
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
}
```

这里有很多的类似的语句。我们可以使用*复合文字*（composite literal，或译作“复合字面”）来对其进行简化，以下是一个在每次求值时创建一个新实例的表达式。

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

注意，返回一个局部变量的地址是完全没问题的，这一点与C不同；与此变量对应的存储在函数返回后仍然存在。事实上，每当进行获取一个复合文字的地址运算时，都将为一个新的实例分配内存，因此以上代码的最后两行可以被合并起来。

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

复合文字的字段必须按顺序全部给出。但如果显式地用*字段*:*值*来标记元素，他们在初始化器中出现的顺序可以是任意的，没有给出的字段则为零值。因此我们可以用

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

少数情况下，如果复合文字不包括任何字段，它将创建该类型的零值。表达式 `new(File)` 和 `&File{}` 是一样的。

复合文字同样可以用于创建数组、切片和映射，其字段标签是相称的索引或映射键。下例的初始化工作不管 `Enone` 、 `Eio` 和 `Einval` 是什么，只要他们不同就行。

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

使用 make 分配内存

再回到内存分配上来。内建 `make(T, args )` 函数的目的与 `new(T)` 不同。它仅用于创建切片、映射和信道，并返回类型 `T` （不是 `*T` ）的一个*被初始化了的*（不是*零*）值。这种差别的出现是由于这三种类型实质上是对在使用前必须进行初始化的数据结构的引用。例如，切片是一个具有三项内容的描述符，包括指向数据（在一个数组内部）的指针、长度以及容量，在这三项内容被初始化之前，切片值为 `nil` 。对于切片、映射和信道， `make` 初始化了其内部的数据结构并准备了将要使用的值。例如

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

为一个具有100个整数的数组分配内存并创建一个长度为10、容量为100并指向此数组前10个元素的切片构造。（生成切片时，其容量可以省略；详见切片一节。）相反， `new([]int)` 返回一个指向新分配内存的零切片结构的指针，也即一个指向 `nil` 切片值的指针。

以下示例说明了 `new` 和 `make` 的不同。

```
var p *[]int = new([]int)       // 为切片结构分配内存；*p == nil；很少使用
var v  []int = make([]int, 100) // 切片v现在是对一个新的有100个整数的数组的引用

// 毫无必要地使问题复杂化：
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// 习惯用法:
v := make([]int, 100)
```

请记住 `make` 只适用于映射、切片和信道，并且其返回值不是指针。要显式地获得一个指针，请用 `new` 分配内存，或显式地取得一个变量的地址。

数组

在详细地规划内存布局时，数组是很有用的，有时使用数组避免进行内存分配，但数组主要用作切片的构建块，这将是下一节讨论的主题。作为对该主题的一个铺垫，这里先对数组说上几句。

Go和C中的数组的主要区别在于，在Go中，

- 数组是值。将一个数组赋值给另一个将复制其所有的元素。
- 特别地，如果你传递一个数组给函数，它将收到此数组的一个*副本*，而不是一个指向它的指针。
- 数组的尺寸是其类型的一部分。`[10]int` 和 `[20]int` 是不同的类型。

数组是值的属性很有用，但代价昂贵；如果你想要类似C的行为和效率，你可以传递一个指向数组的指针。

```
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)  // 注意显式给定的取地址操作符
```

但这种风格并不是Go的习惯用法。切片才是。

切片

切片（slice）通过包装数组而给出了对数据序列的通用、强大和方便的接口。除了如矩阵变换这样显式要求尺寸的情况，多数情况下，Go中的数组编程是通过切片而非简单数组来完成的。

切片保存了对其底层数组的引用，如果你将一个切片赋值给另外一个，这两个切片将引用同一个底层数组。例如，如果一个函数获取了一个切片参量，其对切片元素的改变对调用者来说是可见的，这与传递一个指向底层数组的指针相类似。因此，一个 `Read` 函数可接受一个切片参量而不是一个指针和一个计数；切片的长度设定了可被读取数据的上限。以下是 `os` 中 `File` 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（如果有的话）。要读入一个大缓冲器 `b` 的前32字节，*切分*（slice，这里的slice是一个动词）该缓冲器就可以了。

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

这种切分方法常用且高效。事实上，暂不考虑效率问题，以下片段同样读取缓冲器的前32字节。

```
var n int
var err error
for i := 0; i < 32; i++ {
    nbytes, e := f.Read(buf[i:i+1])  // 读取一个字节.
    if nbytes 0 || e != nil {
        err = e
        break
    }
    n += nbytes
}
```

切片的长度是可以改变的，只要它不超出底层数组的长度极限；只需将其自身的一个切片赋值给它就可以了。切片的*容量*可通过内建的* `cap` *函数来访问，它将给出此切片可被赋予的最大长度。以下是一个为切片追加数据的函数。如果数据超出容量，将重新为切片分配内存。返回值为所得的切片。该函数中所使用的 `len` 、 `cap` 在当应用于 `nil` 切片时，将返回0。

```
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // 重新分配内存
        // 分配所需的两倍的内存, 以便适应将来的增长.
        newSlice := make([]byte, (l+len(data))*2)
        // copy 函数已被预先声明了, 它对任何切片类型都适用.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
    return slice
}
```

我们必须在最后返回切片，这是因为尽管 `Append` 可以修改 `slice` 的元素，但切片自身（其运行时数据结构包含指针、长度和容量）是通过值传递的。

为切片追加东西的想法相当不错，因此有一个专门的内建函数 `append` 实现了此功能。要理解此函数的设计，还需要更多一点信息，我们将稍候再介绍它。

二维切片

Go的数组和切片是一维的。要创建二维数组或切片，需要定义一个数组的数组或切片的切片，如下

```
type Transform [3][3]float64  // 一个 3x3 的数组, 其实是数组的数组.
type LinesOfText [][]byte     // 一个字节切片的切片.
```

由于切片的长度时可变的，这样每个切片元素的长度可以各不相同。以下给出的 LinesOfText 示例就是一种常见的情况：每行的长度都各不相同。

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

有些时候，确实需要使用二维的切片，例如，用其处理像素的扫描线。有两种方法可实现这些。其一是独立地对每个切片分配内存；其二是先分配单个数组的内存，然后将各个独立的切片指向其内部。具体使用何种方法取决于你的程序。如果切片可能增长或减小，应独立地分配内存以避免覆盖写入下一行；反之，使用单个数组构建对象将更加高效。以下给出了两种方法的草案以供参考。首先是一次一行的方法：

```
// 为顶层的切片分配内存.
picture := make([][]uint8, YSize) // y 方向的每个单位分配一行.
// 对行进行扫描, 为每行的切片分配内存.
for i := range picture {
    picture[i] = make([]uint8, XSize)
}
```

然后是一次分配内存，再将其切分为多行的方法：

```
// 为顶层的切片分配内存, 和前面的代码一样.
picture := make([][]uint8, YSize) // y 方向的每个单位分配一行.
// 分哦在涂个大的切片来保存所有像素.
pixels := make([]uint8, XSize*YSize) // 尽管 picture 的类型是[][]uint8, 这
// 通过循环从剩下的 pixels 切片的前部逐次切分出各行.
for i := range picture {
    picture[i], pixels = pixels[:XSize], pixels[XSize:]
}
```

映射

映射是内建的一个方便且强大的数据结构，它可以将一种类型（*键*）与另外一种类型（*元素*或*值*）的值。其键值可以是任意的已定义了相等操作符的类型，如整数、浮点数和复数、字符串、指针以、接口（只要该动态类型支持实现了相等接口）、结构体和数组。切片不能被用于映射键值，因为这些类型上没有定义相等。像切片一样，映射是引用类型。如果你将一个映射传递给函数，并改变了映射的内容，则该更改对调用者来说是可见的。

可以使用常用的复合文字语法进行构建，其各个键-值对之间用逗号分割，因此可以在初始化时来构建他们。

```
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,
}
```

赋值和获取映射值的语法与数组类似，不同的是映射的索引不必是一个整数。

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

如果试图使用一个不存在的键值来获取映射值，就会返回一个映射项目对应类型的零值。例如，如果映射的值类型为整数，查找一个不存在的键值将返回 `0` 。可以使用一个值类型为 `bool` 的映射来实现一个集合。将映射中的项目设置为 `true` ，以此将值放入集合中，然后通过简单的索引对其进行测试。

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

if attended[person] { // 如果 person 不在映射中, 将为 false
    fmt.Println(person, "was at the meeting")
}
```

有时你需要区分不存在的项目和零值。如对一个值本应为零的 `"UTC"` 项，也可能是由于不存在该项而得到零值。你可以使用多重赋值的形式来分辨这种情况。

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

为了便于记忆，可将这种方法称作“逗号ok”。在此例中，如果 `tz` 存在，将会对 `seconds` 进行设置并且 `ok` 将为true；否则 `seconds` 将被设为零值并且 `ok` 将为false。以下函数将这种方法以及适当的错误报告结合在了一起。

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

```
    return 0
}
```

若只需测试映射中是否存在某项而不关心实际的值，可以在通常放接收值的变量的地方使用空白标识符（ _ ）替代。

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

要删除映射中的一项，可使用内建的 delete 函数，其参量为此映射变量和要被删除的键。即便对应的键值在映射中并不存在，此操作也是安全的。

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

打印

Go中格式化打印的风格与C的 printf 一族类似，但却更丰富和通用。这些函数位于 fmt 包中，且函数名以大写字母开头，如 fmt.Printf 、 fmt.Fprintf 、 fmt.Sprintf 等等。字符串函数（ Sprintf 等）并不是填充一个给定的缓冲器，而是返回一个字符串。

你可以不用提供一个格式字符串。每个 Printf 、 Fprintf 和 Sprintf ，都分别对应另外一对函数，如 Print 和 Println 。这些函数并不接受格式字符串，但却为每个参量产生一个默认的格式。 Println 同时在参量之间插入空格并向输出追加一个换行符，而 Print 仅当在操作数的两侧都没有字符串时才添加空白。以下示例中各行产生的输出是一样的。

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

进行格式化打印的 fmt.Fprint 一类函数的第一个参量接受任何实现了 io.Writer 接口的对象；变量 os.Stdout 和 os.Stderr 都是大家熟悉的实例。

下面的事情开始与C有些不同。首先，像 %d 这样的数值格式并不接受表示正负符号或尺寸的标记；打印程序依据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))
```

将打印

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

如果你只想要默认的转换，例如使用十进制的整数，你可以使用通用的格式 %v （ v 指“值”）；其结果与 Print 和 Println 产生的输出是完全一样的。另外，此格式可打印*任何*值，甚至包括数组、结构体和映射。以下是针对上一节定义的时区映射的打印语句。

```
fmt.Printf("%v\n", timeZone)  // 或者只是使用 fmt.Println(timeZone)
```

这将输出

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

当然映射中的键可能以任意的顺序输出。当打印结构体时，改进的格式 %+v 将使用字段的名称标明结构体的字段，而另外一个格式 %#v 将完全以Go语法打印任意值。

```
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)
```

将打印

```
&{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'
```

（请留意其中的&符号。）当遇到类型为 string 或 []byte 的值时，可以使用 %q 产生引号包括的字符串；而格式 %#q 将尽可能使用反引号。（ %q 格式也可以应用于整数和rune类型，产生一个单引号包括的rune常量。）另外， %x 可被用于字符串、字节数组、字节切片以及整数，并产生一个长十六进制字符串，而带空格的格式（ % x ）还会在字节之间插入空格。

另一个趁手的格式是 %T ，它会打印值的*类型*。

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

将打印

```
map[string] int
```

如果你需要控制一个自定义类型的默认格式，你只需要为此类型定义一个具有 String() string 签名的方法。对前面定义的简单 T 类型，可进行如下操作。

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

它将以格式化的形式打印

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

（如果你要像指向 T 的指针那样打印类型 T 的*值*， String 的接收者必须是值类型；上例中的接收者是一个指针，因为这对结构体类型来说更高效和通用。详见下节的指针接收者和值接收者。

String 方法可以调用 Sprintf ，这是因为打印例程是完全可重入并以这种方式封装。但是，理解这种方式的细节时，有一条重要的细节需要遵守：在通过调用 Sprintf 构建 String 方法时，不能无限递归地调用你的 String 方法。这种情况可能出现在当 Sprintf 调用试图直接以字符串打印接收者时，进而再次调用该方法时。这是一个常见且易犯的错误，如下例所示。

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m) // 错误: 将永远递归下去.
}
```

同时这种错误也容易修复：将参量转变为基础字符串类型，这样就不再调用此方法

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", string(m)) // OK: 注意其中的转换.
}
```

在初始化一节中，我们将学到另外一种避免这种递归调用的技术。

另一种打印技术是将一个打印例程的参量直接传递给另一个这样的例程。 Printf 的签名是 ...interface{} ，这样出现在其格式化字符串后面的参量就可以是任意类型任意个数的参数。

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

在函数 Printf 中， v 就像一个类型为 []interface{} 的变量，但如果它被传递给另外一个变参函数，它就变得与一个常规的参量列表一样了。以下是我们在上面用过的 log.Println 函数的实现。它直接将其参量传递给 fmt.Sprintln 进行实际的格式化。

```
// 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,
}
```

在对 Sprintln 的嵌套调用中， v 后面跟着 ... ，这告诉编译器将 v 作为一个参量列表对待，否则，它就只是将 v 作为单个切片参量。

关于打印，还有更多内容。详见 fmt 包的 godoc 文档。

顺便说一下， ... 可以指定类型，如 ...int 可以使一个求最小值的函数选定一个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
}
```

追加

现在我们需要对内建的 append 函数设计进行补充解释。 append 的签名与前面自定义 Append 函数并不相同。从原理上来将，其签名就如：

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

其中*T*是一个针对任何给定类型的占位符。实际上在Go中无法写一个其类型 T 由调用者决定的函数。这就是为何 append 是内建函数的原因：它需要编译器的支持。

append 所做的是在切片的末尾追加元素并返回结果。必须返回结果的原因与前面我们自己写的 Append 一样，即其底层的数组可能已发生改变。以下是一个简单的例子

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

它将打印 `[1 2 3 4 5 6]` 。因此 `append` 就像 `Printf` ，可以接受任意个数的参量。

但如果我们想要做在 `Append` 中所做的工作，或者将一个切片追加到另一个切片该怎么办？很简单：在调用的地方使用 `...` ，就像我们在上面调用 `Output` 那样。以下代码片段的输出与上一个一样。

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

若没有 `...` ，上面的代码会由于类型错误而无法工作；这是因为 `y` 不是 `int` 类型。

初始化

尽管在表面上，Go的初始化与C或C++相比区别并不是很大，但Go却更强大。在初始化过程中可以构建复杂的结构，并且在不同包中的不同被初始化对象间的排序问题能够很好地处理。

常量

Go中的常量就是不变常数。他们在编译时被创建，即便在函数中定义的局部常量也是如此，常量只能是数字、字符（rune）、字符串或布尔值。由于编译时的限制，定义他们的表达式必须是可以被编译器求值的常量表达式。例如 `1<<3` 是常量表达式，而 `math.Sin(math.Pi/4)` 则不是，这是由于函数调用 `math.Sin` 是在运行时发生的。

在Go中，枚举常量使用枚举符 `iota` 创建。由于 `iota` 可以是一个表达式的一部分，并且表达式可以被隐含地重复，这样就更容易构建复杂的值集。

```
type ByteSize float64

const (
    _           = iota // 通过将其赋值给空标识符而忽略第一个值
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

由于可以为任何用户自定义的类型附加一个如 String 这样的方法，从而可以使这些值在打印时自动地格式化他们自己。尽管这种做法大多数是应用于结构体，但其实对标量类型同样有用，如浮点类型的 ByteSize 。

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", float64(b/YB))
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", float64(b/ZB))
    case b >= EB:
        return fmt.Sprintf("%.2fEB", float64(b/EB))
    case b >= PB:
        return fmt.Sprintf("%.2fPB", float64(b/PB))
    case b >= TB:
        return fmt.Sprintf("%.2fTB", float64(b/TB))
    case b >= GB:
        return fmt.Sprintf("%.2fGB", float64(b/GB))
    case b >= MB:
        return fmt.Sprintf("%.2fMB", float64(b/MB))
    case b >= KB:
        return fmt.Sprintf("%.2fKB", float64(b/KB))
    }
    return fmt.Sprintf("%.2fB", float64(b))
}
```

表达式 YB 的打印形式为 1.00YB ，而 ByteSize(1e13) 则打印 9.09TB 。

注意在 ByteSize 的 String 方法中调用 Sprintf 函数是安全的（要避免无限递归调用），这不仅是因为使用了转换，同时因为它通过 %f 调用 Sprintf ， %f 不是一个字符串格式，它需要匹配一个浮点数： Sprintf 仅在需要匹配一个字符串时才调用 String 方法。

变量

变量的初始化与常量类似，但初始化器可以是在运行时被计算的普通表达式。

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

init 函数

最后，每个源文件都可以定义其自己的无参的 `init` 函数来建立各种需要的状态。（实际上每个文件可以具有多个 `init` 函数。）并且它的结束就意味着初始化的结束： `init` 是在包中声明的所有变量求得其初值后被调用，并且求初值工作是在所有被导入的包被初始化之后进行的。

另外初始化中不能进行声明， `init` 的一个常见应用是在真正开始执行前对程序状态的正确性进行验证或修复。

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath 可能被命令行中的 --gopath 标记覆盖.
    flag.StringVar(&gopath, "gopath", gopath, "override default GOPATH")
}
```

方法

指针与值的对比

如同我们在前面看到的 `ByteSize` 那样，可以针对不是指针或接口的其他任何具有名称的类型定义方法；其接收者可以不是结构体。

在前面讨论切片时，我们曾写了一个 `Append` 函数。我们也可以将其定义为切片的方法。要这样做，首先需要声明一个具有名称的类型来绑定该方法，然后使此方法的接收者为该类型的一个值。

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // 方法主体部分与前面的函数完全相同.
}
```

这里仍然要求此方法返回更新过的切片。为了消除这种不便，我们可以重定义此方法，使其接受一个指向 `ByteSlice` 的*指针*作为其接收者，这样此方法就可以覆盖调用者的切片了。

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // 方法主体部分与前面一样, 但没有 return 语句.
```

```
    *p = slice
}
```

事实上，我们可以做得更好。我们可以修改此函数使其看起来更像一个标准的 Write 方法，如下所示：

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // 仍是和前面一样.
    *p = slice
    return len(data), nil
}
```

这样类型 *ByteSlice 就满足标准的接口 io.Writer 了，这样做有其便利之处。例如，我们可以将其用于打印。

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

我们传递了 ByteSlice 的地址，由于只有 *ByteSlice 满足 io.Writer 。关于接收者为指针还是值的规则是：值的方法可以被通过指针和值进行调用，但指针的方法只能被通过指针调用。

之所以这样做是因为指针方法可能修改接收者；通过值调用指针的方法将使此方法收到一个值的副本，而对此副本的任何改动将会被抛弃。Go语言这样规定可以避免发生这种错误。为了方便，这里有个特列。当值是可寻址时，该语言会自动插入取地址操作符，从能能根据常见情况通过值调用指针方法。在上例种，变量 b 是可寻址的，因此我们可仅使用 b.Write 方法调用其 Write 方法。编译器将将其重写为 (&b).Write 。

顺便说一下，在一个字节切片上使用 Write 的想法已由 bytes.Buffer 实现了。

接口和其他类型

接口

Go中的接口提供了一个指定对象行为的方法：如果某样东西可以完成*这个*，则它可被用于*此处*。我们已经见过许多简单的示例了；自定义的打印函数可以通过 String 实现，而 Fprintf 能对任何实现了 Write 的东西产生输出。只有一两个方法的接口在Go代码中很常见，并且其名称常常从方法得来，如 io.Writer 就是实现了 Write 的接口。

一个类型可实现多个接口。例如，一个实现了 sort.Interface 接口的集合就可以使用 sort 对其排序，该接口包括 Len() 、 Less(i, j int) bool 和 Swap(i, j int) ，另外，该集合仍然可以有一个自定义的格式化器。以下特意构建的例子 Sequence 就同时满足这两种情况。

```
type Sequence []int

// sort.Interface 要求的方法.
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]
}

// 打印时用到的方法 - 在打印之前先要对其元素进行排序.
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 方法重复做了 Sprint 针对切片已经实现了的工作。如果将 Sequence 转换为一个普通的 []int ，就能够使用 Sprint 的这种功能。

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

此方法是在 String 方法中安全地调用 Sprintf 所使用的转换技术的另一个示例。由于在忽略类型名称的情况下， Sequence 和 []int 这两种类型是相同的，因此在这两者之间进行转换是合法的。该转换并不会创建一个新值，它只是暂时地认为已有的值具有一个新类型。（还有另外一些合法的转换也不创建新值，如从整数转换为浮点数。）

Go程序常常转换一个表达式的类型以使用不同的方法。例如，我们可以使用已有的 sort.IntSlice 将以上整个示例缩减成这样：

```
type Sequence []int

// 打印时用到的方法 - 在打印之前先要对其元素进行排序.
func (s Sequence) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

现在，不必让 Sequence 实现多个接口，我们可以转而通过将数据项转换为多种类型（ Sequence 、 sort.IntSlice 和 []int ）而使用相应的功能，每次转换都完成一部分工作。在实际使用中，这种做法显得怪怪的，但却很有效。

接口转换和类型断言

类型切换是转换的一种：他们接受一个接口，然后对 switch 语句的每个 case ，在某种意义上将其转变为此种 case 下的类型。以下是 fmt.Printf 函数代码如何使用类型切换将一个值转变为字符串的简化版本。如果接口已经是一个字符串，我们就取得该接口的实际字符串值；如果该接口有一个 String 方法，我们就取得调用此方法的结果。

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

var value interface{} // 由调用者提供的值.
switch str := value.(type) {
case string:
    return str
case Stringer:
    return str.String()
}
```

第一个 case 试图找到一个具体的值；而第二个 case 则将此接口转换为另外一个接口。这种方式对混合类型是非常完美的。

如果我们只关心一种类型，情况又会怎么样呢？如果我们知道某个值保存了一个字符串，我们只是想得到它又该怎么办呢？这时可以使用只有一个 case 的 switch ，但最好使用*类型断言*。一个类型断言接受一个接口值，并从中取得一个明确指定类型的值。其语法借用自类型切换语句的开头部分，但具有一个明确的类型而不是 type 关键字：

```
value.(typeName)
```

其结果将得到一个新的具有静态类型 typeName 的值。此类型必须要么是接口所具有的实体类型，或者是接口值所能转换的一个接口类型。要从一个值中得到一个字符串，我们可以这样写：

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

但如果最终此值不包含一个字符串，此程序将会崩溃，并发出一个运行时错误。为了避免出现这种情况，可使用“逗号，ok”方式来安全地进行测试：

```
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， `str` 将依旧作为一个字符串类型存在，但其值将为零值，即一个空字符串。

为了进一步把问题解释清楚，以下提供一个与本小节开头的类型切换等同的 `if-else` 语句示例：

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

通用性

如果一个类型只实现了一个接口，并且该类型没有除该接口外其他的导出方法，则就不需要导出该类型。仅导出接口的方式明确说明了事情的行为，而不必强调其实现，而具有不同属性的其他实现则可以参照原始类型的行为。这样同样可以避免对一个常用方法的每个实例写重复的文档。

在这种情况下，构造器应该返回一个接口值而不是实现的类型。例如，在哈希库中， `crc32.NewIEEE` 和 `adler32.New` 都返回接口类型 `hash.Hash32` 。在Go程序中，将CRC-32算法替换为Adler-32只需要更改构造器调用，而其余的代码则不受算法更改的影响。

同样的方式使各个 `crypto` 包中的流加密算法与他们链接起来形成的块加密区分开来。 `crypto/cipher` 包中的 `Block` 接口指定了一个块加密行为，它提供对单个数据块的加密。然后，与 `bufio` 包类似，实现此接口的加密包可以被用于构建流加密，这由 `Stream` 表示，并且不必知道块加密的细节。

`crypto/cipher` 接口是这样的：

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

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

这里是计数模式（CTR）流的定义，它将一个块加密转变为流加密；注意块加密的细节是抽象的：

```
// NewCTR 返回一个流，该流使用在计数器模式中给定的Block加密/解密.
// iv 的长度必须等于 Block 中的块尺寸.
func NewCTR(block Block, iv []byte) Stream
```

NewCTR 使用的加密算法和数据源并没有被特别限定，可以是任何 Block 接口的实现和任意的 Stream 。由于他们返回了接口值，将CTR加密替换为其他的加密模式将只是一个局部更改。必须要修改其构造器调用，但由于外围的代码仅将结果看作一个 Stream ，它将不会在意已完成的改动。

接口和方法

由于几乎所有的东西都可以附加方法，因此几乎所有的东西都能满足一个接口。 http 包中就有一个示例，它定义了 Handler 接口。任何实现了 Handler 的对象都能服务HTTP请求。

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

ResponseWriter 本身是一个接口，它提供了用于返回客户端响应的访问方法。这些包括标准的 Write 方法，因此一个 http.ResponseWriter 可被用于所有可使用 io.Writer 的地方。 Request 是一个结构体，它包含了对来自客户端请求解析后的表示。

为了简明起见，让我们忽略POST而假设HTTP请求始终是GET；这种简化并不影响处理程序（handler）的构建方式。以下是一个很小但却完整的处理程序实现，它可以对页面的访问次数进行计数。

```
// 简单的计数器服务器。
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 能打印到一个 http.ResponseWriter 中。）作为参考，这里演示了如何将这样一个服务器程序加到URL树的一个节点上。

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

但为什么要使 Counter 成为一个结构体呢？其实只需要一个整数就够了。（接收者必须是一个指针，这样该增加值对调用者是可见的。）

```
// 简单的计数器服务器。
type Counter int

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

当页面被访问后怎样通知程序去更新一些内部状态呢？请为web页面连上一个信道。

```
// 信道可以在每次访问时发送一个通知.
// (将来可能需要缓存此信道.)
type Chan chan *http.Request

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

最后，假如我们想在调用服务器二进制文件时显示 /args 中的参量。可以很容易地写一个函数来打印这些参量。

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

如何将这些放入HTTP服务器中呢？我们可以将 ArgServer 变成某些类型的一个方法，而这些类型的值可以忽略，但也有一个更整洁的做法。由于我们可以为除了指针和接口的其他任何类型定义方法，也即我们可以为一个函数写一个方法。 http 包中包含以下代码：

```
// HandlerFunc 类型是一个适配器, 它允许将普通的函数作为 HTTP 处理程序
// 使用. 如果 f 是一个具有合适签名的函数, HandleFunc(f) 就是一个调用
```

```
// f 的处理程序对象.
type HandlerFunc func(ResponseWriter, *Request)

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

HandlerFunc 类型有一个 ServeHTTP 方法，因此该类型的值可以为HTTP请求提供服务。查看该方法的实现：其接收者是一个函数 f ，并且该方法调用了 f 。这看起来有点古怪，但却与接收者为信道而方法发送到此信道并没有什么不同。

要使 ArgServer 变为一个HTTP服务器，需要对其进行修改，使其拥有正确的签名。

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

现在 ArgServer 的签名与 HandlerFunc 一样了，这样就能其转化为此类型以访问其方法，就像将 Sequence 转换为 IntSlice 并访问 IntSlice.Sort 一样。设置代码很简洁：

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

当某人访问 /args 页面时，对应此页面的处理程序的值为 ArgServer ，类型为 HandlerFunc 。HTTP服务器将调用该类型的 ServeHTTP 方法，并以 ArgServer 作为接收者，这将相应地调用 ArgServer （通过调用 HandlerFunc.ServeHTTP 中的 f(c, req) 显示）。进而将显示各参量。

在该小节中，我们从一个结构体、整数、信道及函数构建了一个HTTP服务器，这全都因为接口就是方法的集合，它可以针对（几乎是）任何类型定义。

空白标识符

空白标识符通常使用在多重赋值时，而在一个 for range 循环中使用只是其应用情景之一。

如果赋值的左侧需要多个值，但其中一个值在程序中并没有被用到，在赋值语句的左侧可以使用一个空白标识符来避免创建一个无用的变量，同时明确说明此值被丢弃。例如，如果要调用的函数将返回一个值和一个错误，但只有错误是重要的，就可以使用空白标识符来舍弃不相关的值。

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

```
}
```

有的代码会舍弃错误值，这是为了忽略错误；不过这种做法通常很糟糕。请始终对错误返回值进行检查；之所以返回错误，都是有原因的。

```
// 糟糕的方法! 如果路径不存在, 此代码将崩溃.
fi, _ := os.Stat(path)
if fi.IsDir() {
    fmt.Printf("%s is a directory\n", path)
}
```

未使用的导入和变量

Go把导入一个包或声明一个变量后而不使用它的行为看作是一个错误。未使用的导入使程序变大并降低编译速度，而初始化一个变量却不使用它首先会浪费计算，并有可能导致一个更大的bug。当程序的开发并不活跃时，常常会出现未使用的导入或变量，并且有些变量之所以难以删除只是因为他们要被用于编译过程而不是编译之后。可以使用空白标识符来解决此问题。

以下写就一般的程序具有两个未使用的导入（ `fmt` 和 `io` ）以及一个未使用的变量（ `fd` ），因此它无法通过编译，但只要是正确的，该程序看起来还不错。

```
package main

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

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

为了避免编译器对未使用导入的抱怨，可以使用一个空白标识符来关联来自导入包的一个符号。同样地，将未使用 `fd` 变量赋值给一个空白标识符也将阻止出现未使用变量的错误。以下的程序就能通过编译。

```
package main
```

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

var _ = fmt.Printf // 仅用作调试; 在调试结束后就应该删除.
var _ io.Reader    // 仅用作调试; 在调试结束后就应该删除.

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

按照惯例，对于为了防止导入错误而加入的全局空白标识符声明，应被紧放在导入之后并加注声明，这样使他们便于被找到，并提醒我们需要在今后对他们进行清理。

为了次要作用而导入

在前面的例子中，如 `fmt` 或 `io` 这些未使用的导入最终应被使用或删除：空白赋值语句只是为了标明代码的工作进度。但有时并不需要显式地使用包，而只是为了使用包的一些次要作用而导入包。例如，net/http/pprof包的 `init` 在执行过程中将对提供调试信息的HTTP处理程序进行注册。虽然该包具有一个导出的API，但多数客户端仅需要处理程序注册并通过一个网页访问数据。如果只为了其次要作用而导入包，可将其名称更改为空白标识符：

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

这种形式的导入明确说明了只是为了使用包的次要作用才导入包，由于该包已不可能有其他用途，因此在此文件中，该包不需要具备名称。（如果该包有名称，但我们却没有使用此名称，编译器将拒绝编译此程序。）

接口检查

正如前面讨论接口时所讲得那样，一个类型并不需要明确声明其所实现的接口。要实现某个接口，该类型只需要实现此接口的方法即可。实际上多数接口转换是静态的，并需要在编译时进行检查。例如，如果一个函数接收的东西需要实现 `io.Reader` 接口，并将一个 `*os.File` 传递给此函数，则只有在 `*os.File` 实现了 `io.Reader` 接口时，编译才能通过。

然而，仍有一些接口检查工作是在运行时完成的。encoding/json包就是一个例子，该包定义了一个Marshaler接口。当JSON编码器收到一个实现了此接口的值时，该编码器将调用接收到值本身的编码方法将其转换为JSON，而不是进行标准的转换。编码器在运行时使用一种和类型断言类似的方法来检查这方面的属性：

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

如果只需要询问是否一个类型实现了一个接口，而没有实际使用接口自身，可以在错误检查中，使用空白标识符去忽略类型断言的值：

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

当需要确保包中的类型确实满足某接口时，就会使用这种方式。如果像json.RawMessage这样的类型需要一个自定义的JSON表示，它应该实现 `json.Marshaler` ，但这里不存在可导致编译器自动进行验证的静态转换。如果类型非故意地不能满足此接口，JSON编码器将仍能工作，但却不使用自定义的实现。要确保实现是正确的，在此包中可以使用一个具有空白标识符的全局声明：

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

在此声明中的赋值涉及将一个 `*RawMessage` 转换为 `Marshaler` ，这就要求 `*RawMessage` 实现了 `Marshaler` ，在编译时将对这方面属性进行验证。假若 `json.Marshaler` 发生了改变，此包浆无法再通过编译，这就使我们知道此包需要更新了。

在这种构造方式中空白标识符的出现仅是为了表明该声明的存在仅用于进行检查，而不是为了创建一个变量。请不要用这种方法进行所有的接口满足情况的验证。通常情况下，这种声明仅用于当代码中不存在静态转换时，这种情况比较少见。

嵌入

Go不提供典型的、类型驱动的子类化概念，但它通过在一个结构体或接口中*嵌入*类型而能够从前者的实现中“借用”一些东西。

接口的嵌入非常简单。前面我们已经提到了 `io.Reader` 和 `io.Writer` 接口。这里是他们的定义。

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

type Writer interface {
```

```
    Write(p []byte) (n int, err error)
}
```

io 包同时也导出了几个其他的接口来指定对象能够实现的几个类似的方法。例如， io.ReadWriter 就是一个包含 Read 和 Write 的接口。我们可以通过显式地列出这两个方法来定义 io.ReadWriter ，但更简便且更易被理解的是嵌入这两个接口而形成新的一个，如下所示：

```
// ReadWriter 接口组合了 Reader 和 Writer 接口.
type ReadWriter interface {
    Reader
    Writer
}
```

正如该代码看起来那样： ReadWriter 能够完成 Reader *和* Writer 所完成的工作；它是被嵌入接口（他们的方法不能有交集）的一个并集。只有接口才能被嵌入到接口中。

同样的理念也可被应用于结构体中，但却有更多的涵义。 bufio 包有两个结构体类型， bufio.Reader 和 bufio.Writer ，两者各自实现了来自 io 的对应接口。另外 bufio 还实现了一个缓冲的reader/writer，这是使用嵌入将一个reader和一个writer组合成一个结构体：它在结构体内列出了这些类型但并没有给出这些类型的字段名称。

```
// ReadWriter 包含了指向一个 Reader 和一个 Writer 的指针.
// 它实现了 io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

被嵌入的元素是指向结构体的指针，在可以使用这些类型之前必须将他们初始化，使他们指向合法的结构体。 ReadWriter 结构体也可以如下方式定义

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

但若要使字段的方法提升为结构体的方法，以使结构体满足 io 接口，我们还需要提供转发的方法，如下所示：

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

而通过直接嵌入结构体，就可以不必这么繁琐。嵌入类型的方法被自动继承得来，这意味着 bufio.ReadWriter 不仅具有 bufio.Reader 和 bufio.Writer 方法，同时还满足三个接口： io.Reader ， io.Writer 和 io.ReadWriter 。

嵌入与子类化有一个重大不同。当嵌入一个类型时，此类型的方法变为外部一级类型的方法，而当这些方法被调用时，他们的接收者是内部一级的类型，而非外部一级。在上例中，当调用 bufio.ReadWriter 的 Read 方法时，将出现与以上转发的方法相同的结果；其接收者为 ReadWriter 的 reader 字段，而非 ReadWriter 本身。

嵌入还有另外一个小便利，如下例子展示了一个嵌入字段以及一个正常的命名字段。

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

现在 Job 类型具有 *log.Logger 的 Log 、 Logf 等方法。我们当然也可以给 Logger 一个字段名称，但却没有必要这么做。现在，一旦完成初始化，我们就可以对 Job 进行日志记录：

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

Logger 是 Job 结构体的一个常规的字段，我们可以通过 Job 的构造器使用常规的方法初始化它，

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

或者使用复合文字，

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

如果我们需要直接引用嵌入的字段，就使用字段的类型名称，省略包限定词，其作用和字段名称一样，如同我们在 ReaderWriter 结构体的 Read 方法。如果我们需要访问 Job 类型的 job 变量的 *log.Logger ，写成 job.Logger 就行了。当我们想要精确控制 Logger 的方法时，这种方式将很有用。

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

嵌入类型会引入类型冲突的问题，但解决的规则也很简单。首先，一个字段或方法 X 将隐藏更深层嵌入类型的 X 项。如果 log.Logger 包含一个名称为 Command 的字段或方法，将只使用 Job

的 `Command` 字段。

其次，如果相同的嵌套级别上出现相同的名称，这将出现一个错误；如果 `Job` 结构体包含另外一个名称为 `Logger` 的字段或方法，则再嵌入 `log.Logger` 将产生错误。但是，如果重复的名字从未在类型定义以外的程序中用到过，就不会出现问题。这种限定对外部嵌入类型修改的一些保护；如果所添加的一个字段与另一个次级类型中的另一个字段产生冲突，但两个字段都没有被用到过，则就不会发生问题。

并发

通过通信共享

并发编程是一个大论题，由于篇幅限制，这里只讨论一些Go特有的东西。

要实现对共享变量的正确访问非常复杂，这使得多数环境中的并发编程都很困难。Go尝试一种不同的方法，其中共享的值通过信道（channel）进行传递，事实上，从来没有什么东西会被多个执行的线程一直共享。在任何给定的时间，只能有一个goroutine访问该值。这样就从设计上杜绝了数据竞争。为了鼓励这种思考方式，我们将其简化为一个口号：

> 不要通过共享内存来进行通信，而应通过通信来共享内存。

这种方式有很多好处。例如，虽然通过将一个整型变量设置为互斥量来实现引用计数是一种非常好的方法。但在高级的方法中，使用信道来控制访问可以更容易地编写整洁且正确的程序。

来说明此模型的一个方法是考虑一个运行在单CPU上的典型的单线程程序。它不需要什么同步机制。现在运行另外一个同样的程序；它同样也不需要同步。然后让这两个程序通信；如果通信正好合拍，同样也不需要其他的同步。例如，Unix的管道就完美地符合此种模型。尽管Go的并发概念始自Hoare的通信序列处理（CSP），它同样可被看作是实现了类型安全的Unix管道。

goroutine

之所以叫*goroutine*（有些地方翻译为Go程）是因为已有的一些术语——线程、协程、进程等——可能会传达不准确的涵义。goroutine具有简单的模型：它是一个与其他goroutine在同一地址空间中并发执行的函数。相对于在栈空间分配内存，它更轻量级且消耗少。这样起始时栈就可以更小，因此也更轻省，随着程序的增长，可以根据需要在堆上分配（和释放）存储空间。

goroutine可复用多个操作系统线程，因此如果其中的一个被阻塞，比如等待I/O，其他的会继续运行。这种设计隐藏了许多线程创建和管理的复杂性。

在函数或方法调用的前面加上 `go` 关键字可在一个新的goroutine中运行调用。当调用完成后，此goroutine将会静默地退出。（这种效果与Unix shell的 `&` 符号可在后台运行命令的概念相似。）

```
go list.Sort()  // 并发地运行 list.Sort concurrently; 不必等待其运行结束.
```

在goroutine调用中使用函数文字（function literal）会非常方便。

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // 注意这里的圆括号 – 必须要调用此函数.
}
```

在Go中，函数文字就是闭包：其实现能确保被此函数引用的变量只要是活动的（能再次被使用）就一直存在。

由于函数在完成后没办法发出信号，这些例子并没有什么实用性。要做到更有用，就需要信道。

信道

与映射一样，信道（channel）是引用类型，需要使用 make 分配内存，结果将得到一个对底层数据结构的引用。如果同时提供了一个可选的整型参数，它将为信道设置缓存的大小。其默认值是零，相当于一个无缓存的或同步的信道。

```
ci := make(chan int)            // 无缓存的整数类型信道
cj := make(chan int, 0)         // 无缓存的整数类型信道
cs := make(chan *os.File, 100)  // 缓存的信道, 指向 File 的指针
```

无缓存的信道将以下东西组合在一起：通信——值的交换——同步——确保两个计算（goroutine）的状态可知。

使用信道有许多章法。这里先从其中的一个出发。在上一节中我们在后台启动了一个排序 goroutine。可以通过使用一个信道来使得启动该排序例程的goroutine等待着排序的完成。

```
c := make(chan int)  // 为一个信道分配内存
// 在一个 goroutine 中启动排序; 当其完成后, 向信道发消息.
go func() {
    list.Sort()
    c <- 1  // 发送一个消息; 值是什么不重要
}()
doSomethingForAWhile()
<-c   // 等待排序结束; 丢弃所发送的值
```

接收者将一直被阻塞，直到收到数据。如果此信道无缓存，发送者将一直被阻塞直到接收者收到了此值。如果信道有一个缓存，发送者仅在值被复制到缓存之前被阻塞；如果缓存满了，这就意味着需要等到一些接收者取回了一个值。

一个缓存的信道可以被像“臂板信号机”那样使用，如进行限制通过。在下例中，进入的请求被传递到 handle ，该 handle 向信道发送一个值，然后处理请求，再然后从信道接收一个值使此“臂板

信号机”对下一个消费者可用。信道缓存的容量限制了同时调用 process 的数量，因此在初始化时我们需要将信道填充到特定的容量来准备信道。

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

func handle(r *Request) {
    sem <- 1    // 等待活动队列(queue)耗尽.
    process(r)  // 可能要花费很长时间.
    <-sem       // 完成; 使下一个请求可以运行.
}

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

当正在执行 process 的处理程序数目达到 MaxOutstanding 时，任何其他的发向此填满的信道缓存的东西都将被阻塞，直到其中一个已有的处理程序结束执行并再次从信道缓存接收东西。

这种设计有一个问题： Serve 为每个进入的请求创建一个新的goroutine，即便是这些信道共同使用的 MaxOutstanding 随时都能运行时也是如此（TODO: 此处翻译需要再斟酌）。这样如果请求发生太快，可能导致程序无限制地消耗资源。我们可以通过更改 Serve 来控制goroutine的创建，从而解决这个问题。以下是一个简单的解决方法，但要注意此段代码存在一个bug，我们在稍后对其进行修正：

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func() {
            process(req) // Buggy; 请看下面的解释
            <-sem
        }()
    }
}
```

其bug就是在Go的 for 循环中，每次迭代都要重用循环变量，因此 req 变量会在所有goroutine之间共享。这可不是我们想要的结果。我们需要确保 req 对每个goroutine都是唯一的。以下是解决此问题的一条途径，即将 req 的值作为一个参量到goroutine的闭包中：

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

```
            process(req)
            <-sem
        }(req)
    }
}
```

可将此版本与上个版本进行差异性对比来看闭包是如何声明和运行的。另一个解决途径是创建一个同名的新变量，如下例所示：

```
func Serve(queue chan *Request) {
    for req := range queue {
        req := req // 为 goroutine 创建一个新的 req 实例
        sem <- 1
        go func() {
            process(req)
            <-sem
        }()
    }
}
```

这样写看起来很奇怪

```
req := req
```

但它是合法的，也是Go的惯用法。通过这样你将获得相同名字的全新的变量，可以特意地在局部位置隐藏循环变量，确保循环变量对每个goroutine都是唯一的。

再次回到写服务器的常规问题，另一个很好地管理资源的方法是启动固定数目的 `handle` goroutine，这些goroutine都从请求的信道中读取。goroutine的数目限制了同时调用的 `process` 的数目。这里的 `Serve` 函数同样接受一个信道，该信道将告知此函数何时退出；在启动goroutine后，它将不再从该信道上接收东西。

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

func Serve(clientRequests chan *Request, quit chan bool) {
    // 启动处理程序
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // 等待被告知可以退出.
}
```

信道的信道

Go最重要的特性之一就是信道根本就是值，它可以像其他值一样被分配内存并进行传递。常使用这种特性来实现安全、并行的多路复用。

在上一节的例子中， `handle` 是针对请求的理想化的处理程序，但我们并没有定义其所处理请求的类型。如果该类型包括它要回复的一个信道，每个客户端都能提供自己的回答路径。以下是对类型定义的一个示意。

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

客户端提供了一个函数和它的参量，以及一个包含在用来进行接收回答的请求对象中的信道。

```
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
// 等待响应
fmt.Printf("answer: %d\n", <-request.resultChan)
```

在服务器一侧，仅仅需要对处理程序函数进行更改。

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

很显然实际应用中还需要做很多的工作，但这里的代码构建了一个针对限制速率的、并行的、非阻塞的RPC（远程过程调用）系统的框架，并且这里看不到一个使用互斥的情况。

并行化

此种概念的另一种应用就是在多个CPU内核上实现并行计算。如果计算可以被拆分为多个可独立执行的块，它就可以进行并行处理，当每块计算完成后，就使用一个信道来标记。

假设我们需要对一个向量的多个子项行大量操作，并且对每个子项的操作值是相互独立的，以下是一个理想化的示例。

```
type Vector []float64

// 将操作应用到 v[i], v[i+1] ... 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    // 发信号表明此操作片段已完成
}
```

我们在一个循环中独立地启动各个计算块，每个CPU对应对应一块。他们的完成顺序可以是任意的；我们只需要在启动所有goroutine后通过从信道得到的信号计算出已完成计算块的数量就行了。

```
const NCPU = 4  // CPU核心的个数

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    // 等待一项任务完成
    }
    // 所有工作已完成.
}
```

Go运行时的当前实现将默认不并行运行此代码。它对用户级处理只投入一个核心。任意数量的goroutine在系统调用时会被阻塞，但默认在任何时候只能有一个内核能执行用户级代码。它本该更聪明一点，并且有一天这一点会实现，但目前如果你想使多个CPU并行运行，就必须明确告诉运行时你想要同时执行的goroutine的数目。有两种相关方法来实现这一目的。要么是使用环境变量 `GOMAXPROCS` 设定将要使用的核心数目来运行你的工作，要么是导入 `runtime` 包并调用 `runtime.GOMAXPROCS(NCPU)` 。 `runtime.NumCPU()` 的值很有用，它会报告在本地机子上的CPU数目。另外，随着开发进度的完成以及运行时的改善，将来会不再要求这样做。

请主要不要混淆并发——将一个程序构造成多个可独立执行的组件——和并行——为了提高效率在多个CPU上进行并行计算。尽管Go的并发特征可以像并行计算一样使一些问题更容易构造，但Go是一个并发语言，不是一个并行语言，并非所有的并行问题都适合Go的模型。要了解两者的区别，请参见这篇博客的讨论。

一个漏桶缓存

并发编程工具甚至能使非并发的思想更容易被表达。以下示例的概念是从RPC包抽象而来。客户端goroutine循环从一些源（可能为一个网络）接收数据。为了避免对缓存分配和释放内存，它维持了一个自由列表，并使用一个缓存的信道来代表它。如果该信道是空的，将分配一个新的缓存。一旦此消息缓存就绪，它将通过 serverChan 送给服务器。

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

func client() {
    for {
        var b *Buffer
        // 如果已有缓存，则获取它，否则对其分配内存。
        select {
        case b = <-freeList:
            // 获的了一个，其他什么也不做。
        default:
            // 不存在，分配一个新的。
            b = new(Buffer)
        }
        load(b)              // 从网络读取下一条消息。
        serverChan <- b      // 发送到服务器。
    }
}
```

服务器在循环中从客户端接收每条消息，并对其进行处理，然后向自由列表返回此缓存。

```
func server() {
    for {
        b := <-serverChan    // 等待工作任务。
        process(b)
        // 在有空间时重用缓存。
        select {
        case freeList <- b:
            // 有缓存在自由列表；其他什么也不做。
        default:
            // 自由列表满了，继续下去。
        }
    }
}
```

客户端试图从 freeList 检出一个缓存；如果没有可用的，它将分配一个新的。如果自由列表没有满，服务器向 freeList 的发送将 b 放回自由列表；如果自由列表满了，则自由列表下方位置的缓存将被删除进而被垃圾回收器回收。（当其他的case都不可用时， select 语句中的 default 从

句将被执行，这样 selects 将永远不会被阻塞。）这种实现基于缓存的信道和垃圾回收器记账，仅通过区区几行代码，构建了一个漏桶算法的自由列表。

错误

计算机库程序必须经常向调用者返回一些错误指示。前面已经提到过，Go的多值返回使其在返回一个常规的返回值之外，还能轻易地返回一个详细的错误描述。通常情况下，错误的类型是 error ，这是一个内建的接口。

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

库的作者可以在此封装之下自由地用更丰富的模型实现此接口，从而不仅看到错误，并且提供一些上下文。例如， os.Open 不仅返回了一个常规的 *os.File ，它同时还返回了一个错误值 os.PathError 。如果文件被成功打开，该错误值为 nil ，但如果出现问题，该问题将会被保存在 os.PathError 中：

```
// PathError 记录了一个错误以及引发此错误的操作和文件路径.
type PathError struct {
    Op string    // "open", "unlink" 等等.
    Path string  // 对应的文件.
    Err error    // 由系统调用返回.
}

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

PathError 的 error 产生的字符串样式为：

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

这种错误包含了有问题的文件名称、操作以及其所触发的操作系统错误，打印出这些信息非常有用，即便距离引发此错误的调用很远时也是如此；相对于平白的“no such file or directory”，它更有说明性。

错误字符串应尽可能地标示出他们的来源，如放入一个包的名称作为产生的错误信息的前缀。比如，在 image 包中，由于未知格式引发的解码错误字符串表示为：”image: unknown format”。

当调用者想准确地得到错误细节时，可以使用一个类型切换或类型断言来查找特定的错误并抽取其细节。如对于 PathErrors ，就可以检查内部的 Err 字段以进行错误恢复。

```
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()  // 恢复一些空间.
        continue
    }
    return
}
```

上面第二个 `if` 语句在Go中很常见。通过“逗号ok”的习惯用法（先前已经在检查映射的上下文中提到过）。如果类型断言失败， `ok` 将为假（false），并且 `e` 将为 `nil` 。如果成功， `ok` 将为真（true），这意味着此错误正属于 `*os.PathError` 类型，即为 `e` ，我们可以用它对此错误的更多信息进行检查。

严重错误（Panic）

通常通过返回一个额外的 `error` 值来向调用者报告一个错误。标准的 `Read` 方法就是一个大家熟知的例子；它返回一个字节统计数和一个 `error` 。但当错误不可恢复时会怎么样呢？有时只是简单地让程序停止运行。

对此，有一个内建的 `panic` 函数，它将创建一个运行时错误并使程序停止（请继续看下一节）。该函数接收一个任意类型——往往是字符串——的参量作为程序死亡时要打印的东西。它同样也是标明已经发生了一些不可能完成事件的一种方法，例如，退出无限循环。事实上，当编译器在函数的结尾处检查到一个 `panic` 时，就会停止进行常规的 `return` 语句检查。

```
// 随意写的用牛顿方法求解立方根的代码.
func CubeRoot(x float64) float64 {
    z := x/3   // 任意的初值
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // 出错了, 没有完成百万次的迭代.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

这仅仅是一个示例，但实际的库函数应避免 `panic` 。如果问题可以被掩盖或解决，最好是让事情继续下去而不是终止整个程序。一个反例可能发生在初始化期间：如果库不能设定自己，这时就应该发

出严重错误。

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

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

恢复

当 panic 被调用时，包括隐式的运行时错误，如对数组的引用越界或类型断言失败，它将立即停止当前函数的执行并开始解开goroutine的堆栈，同时运行所有被延期的函数。如果这种解开达到goroutine堆栈的顶端，程序就死亡了。但是，也可以使用内建的 recover 函数来重新获得goroutine的控制权并恢复正常的执行。

对 recover 的调用会通知解开堆栈并返回传递到 panic 的参量。由于仅在解开期间运行的代码处在被延期的函数之内， recover 仅在被延期的函数内部才是有用的。

recover 的应用之一就是关闭一个服务器内运行失败的goroutine，同时不用杀死其他正在执行的goroutine。

```
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)
}
```

在此例中，如果 do(work) 发生严重错误（panic），其结果将被记录下来，goroutine会干净地退出，并不会打断其他的goroutine。在被延期的闭包中并不需要做其他事情；对 recover 的调用彻底地处理了这种情况。

由于除非 recover 直接被一个延期的函数调用，它将总是返回 nil ，因此被延期的代码可以调用使用了 panic 和 recover 的库程序而不发生错误。例如，在 safelyDo 中的被延期函数可能在

调用 recover 之前先调用一个日志记录函数，此记录代码的运行并不受错误处理（panicking）状态的影响。

通过合理地使用恢复模式， do 函数（或其他任何名称）可以通过调用 panic 从任何糟糕的情况中干净利落地脱身。我们可以使用这种概念在复杂的软件中简化错误处理。让我们来看看来自 regexp 包中的一个理想化的节选，它以一个局部错误类型通过调用 panic 报告解析错误。以下是 Error 、 error 方法以及 Compile 函数的定义。

```
// 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 遭遇严重错误，恢复代码将把返回值设为 nil ——被延期的函数可以修改已被命名的返回值。然后，它会对 err 再进行检查，这种检查是通过断言 err 具有局部的 Error 类型来断定出现的问题是一个解析错误。如果不是解析错误，此类型断言将会失败，这将引起一个运行时错误，从而使堆栈的解开继续进行下去，就如同不曾有什么打断过此项解开工作一样。这种检查意味着如果发生了一些未遇到到的事情，例如数组索引超限，则即便我们已经使用了 panic 和 recover 来处理用户触发的错误，代码仍将失败。

通过合理地使用错误处理， error 方法使其能很容易地报告解析错误而不必操心需要手动解开解析堆栈。

尽管这种模式很有用，它应该只被用于包内。 Parse 将其内部的 panic 调用转变为 error 值；它没有向客户端暴露 panics 。这是一种需要遵循的好原则。

顺被提一下，如果一个实际的错误发生了，这种重新触发严重错误（re-panic）的习惯用法改变了严重错误（panic）的值。但是，原来的和新的错误都将会出现在崩溃报告中，因此引发问题的根仍然是可见的。因此这种简单的重新触发严重错误的方法通常已经足够了——它毕竟是一个意外事故——但如果你只想显示原始值，你可以稍微多写点代码来筛选出未遇到的问题并重新触发此原始的严重错误。这就在为练习留给读者了。

一个web服务器

让我们以一个完整的Go程序作为结束：一个web服务器。该程序实际上是web服务的重用。Google在http://chart.apis.google.com提供了一个自动将数据格式化为图表和图形的服务。该服务很难交互地使用，这是因为你需要将数据放入URL构成一个查询。此处的程序为其中一种数据形式提供了一个更好的接口：给出一小段文本，它将调用图表服务器产生一个二维码（QR code），即一个对此文本进行编码的矩阵框。此图像可以通过你的手机摄像头获取，并被解释为一个URL，免除了你从手机的小键盘上输入URL的麻烦。

以下是完整的程序。其后的文字是对它的解释。

```
package main

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

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


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 .}}
<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF

{{.}}


{{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` 程序之前的部分应该很容易理解。一个标记（flag）设置了我们服务器的默认端口。模板变量 `templ` 正是有趣之处。它构建了一个将被解析并显示的网页的HTML模板；过会儿再细讲。

`main` 解析了标记并使用我们在前面已讲过的机制将 `QR` 函数绑定到服务器的根路径。然后调用 `http.ListenAndServe` 启动服务器；当服务器运行时它将保持阻塞。

`QR` 只是接收包含表单数据的请求，并在名称为 `s` 的表单值所包含的数据上执行模板。

`template` 包非常强大；该程序将使用了它功能的一点皮毛。本质上，它通过将文本中的元素替换为传递到 `templ.Execute` 的数据项（上例为表单值）元素来重写一段文本。在模板文本中（`templateStr`），双达括号包括起来的文本标明了模板的行为。而从 `{{if .}}` 到 `{{end}}` 间的文本片段仅在当前数据项（称作点 `.`）的值为非空时才被执行。也就是说，当此字符串为空时，此部分模板就会被忽略。

`{{urlquery .}}` 片段告知 `urlquery` 函数去处理数据，它使查询字符串可以安全地在web页面上显示。

余下的模板字符串只是在页面加载时将要显示的HTML。如果你无理解这种快速入门解释，请参看模板包的 `template` 文档以得到一个更彻底的讨论。

这就是你现在得到的：一个仅通过少数几行代码实现的有用的web服务器以及一些数据驱动的HTML文本。Go很强大，使其能仅用少数几行代码完成大量的工作。