```go
package main
import "fmt"

func main() {

    fmt.Println("Effective Go")

}
```

Effective Go

build simple, reliable & efficient software

Table of Contents

《Effective Go》中英双语版

Effective Go - 《实效 GO 编程》

Introduction

Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make effective Go programs different in character from programs written in its relatives. A straightforward translation of a C++ or Java program into Go is unlikely to produce a satisfactory result—Java programs are written in Java, not Go. On the other hand, thinking about the problem from a Go perspective could produce a successful but quite different program. In other words, to write Go well, it's important to understand its properties and idioms. It's also important to know the established conventions for programming in Go, such as naming, formatting, program construction, and so on, so that programs you write will be easy for other Go programmers to understand.

This document gives tips for writing clear, idiomatic Go code. It augments the language specification, the Tour of Go, and How to Write Go Code, all of which you should read first.

引言

Go 是一门全新的语言。尽管它从既有的语言中借鉴了许多理念，但其与众不同的特性，使得用 Go 编程在本质上就不同于其它语言。将现有的 C++ 或 Java 程序直译为 Go 程序并不能令人满意——毕竟 Java 程序是用 Java 编写的，而不是 Go。 另一方面，若从 Go 的角度去分析问题，你就能编写出同样可行但大不相同的程序。 换句话说，要想将 Go 程序写得好，就必须理解其特性和风格。了解命名、格式化、程序结构等既定规则也同样重要，这样你编写的程序才能更容易被其他程序员所理解。

本文档就如何编写清晰、地道的 Go 代码提供了一些技巧。它是对 语言规范、 Go 语言之旅以及 如何使用 Go 编程 的补充说明，因此我们建议您先阅读这些文档。

`Effective Go` 作为 `GO` 语言的入门必读教程，值得每位初学者好好阅读一遍，编辑成书，方便阅读交流。

当前完成章节（全部完成）：

改版说明：@2016.8.6 by bingoHuang, revision to Chinese & English version.

李笑来在他的新书 《人人都是工程师》 中说过一句话：在中国，对绝大多数人来说， `English + Computer Skills = Freedom` （英语 + 计算机技能 = 自由）

我非常的赞同。英语和计算机技能是相辅相成，学习好一门编程语言（如 `Go` ）的同时，还能加强英语学习，何乐而不为。所以我决定将本书改版成中英双语版，方便更多的人来学习阅读。

特别感谢 Golang 官网提供的英文版教程。

感谢 hellogcc 提供的 中文翻译版一，这是我之前制作中文版电子书所参考的资料，翻译的很用心。

要更感谢 Go 语言中文社区 提供的 中文翻译版二，此翻译更贴切有味道，不可多得。本人已和 Go-zh 项目组 沟通过，获取了该社区的授权，故将此作为双语版的中文版本。

参考

参考官方英文版：*Effective Go* 英文版

参考中文翻译版一：*Effective Go* 中文版

参考中文翻译版二：*Effective Go* 中文版

Read, Fork and Star

- **Read on Gitbook**

- **Fork on GitHub**

- 请顺手点一下 STAT ，或者留言讨论，这是对我最大的鼓励！

License

除特别注明外，本页内容均采用知识共享 - 署名（CC-BY）3.0 协议授权，代码采用 BSD 协议 授权。

Introduction

引言

Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make effective Go programs different in character from programs written in its relatives. A straightforward translation of a C++ or Java program into Go is unlikely to produce a satisfactory result—Java programs are written in Java, not Go. On the other hand, thinking about the problem from a Go perspective could produce a successful but quite different program. In other words, to write Go well, it's important to understand its properties and idioms. It's also important to know the established conventions for programming in Go, such as naming, formatting, program construction, and so on, so that programs you write will be easy for other Go programmers to understand.

Go 是一门全新的语言。尽管它从既有的语言中借鉴了许多理念，但其与众不同的特性，使得使用 Go 编程在本质上就不同于其它语言。将现有的 C++ 或 Java 程序直译为 Go 程序并不能令人满意——毕竟 Java 程序是用 Java 编写的，而不是 Go。 另一方面，若从 Go 的角度去分析问题，你就能编写出同样可行但大不相同的程序。 换句话说，要想将 Go 程序写得好，就必须理解其特性和风格。了解命名、格式化、 程序结构等既定规则也同样重要，这样你编写的程序才能更容易被其他程序员所理解。

This document gives tips for writing clear, idiomatic Go code. It augments the language specification, the Tour of Go, and How to Write Go Code, all of which you should read first.

本文档就如何编写清晰、地道的 Go 代码提供了一些技巧。它是对 语言规范、 Go 语言之旅以及 如何使用 Go 编程 的补充说明，因此我们建议您先阅读这些文档。

Examples

示例

The Go package sources are intended to serve not only as the core library but also as examples of how to use the language. Moreover, many of the packages contain working, self-contained executable examples you can run directly from the golang.org web site, such as this one (if necessary, click on the word"Example"to open it up). If you have a question about how to approach a problem or how something might be implemented, the documentation, code and examples in the library can provide answers, ideas and background.

Go 包的源码 不仅是核心库,同时也是学习如何使用 Go 语言的示例源码。 此外,其中的一些包还包含了可工作的,独立的可执行示例,你可以直接在 golang.org 网站上运行它们,比如 这个例子 (单击文字"示例"来展开它)。如果你有任何关于某些问题如何解决,或某些东西如何实现的疑问, 也可以从中获取相关的答案、思路以及后台实现。

Formatting

格式化

Formatting issues are the most contentious but the least consequential. People can adapt to different formatting styles but it's better if they don't have to, and less time is devoted to the topic if everyone adheres to the same style. The problem is how to approach this Utopia without a long prescriptive style guide.

格式化问题总是充满了争议，但却始终没有形成统一的定论。虽说人们可以适应不同的编码风格，但抛弃这种适应过程岂不更好？若所有人都遵循相同的编码风格，在这类问题上浪费的时间将会更少。 问题就在于如何实现这种设想，而无需冗长的语言风格规范。

With Go we take an unusual approach and let the machine take care of most formatting issues. The gofmt program (also available as go fmt, which operates at the package level rather than source file level) reads a Go program and emits the source in a standard style of indentation and vertical alignment, retaining and if necessary reformatting comments. If you want to know how to handle some new layout situation, run gofmt; if the answer doesn't seem right, rearrange your program (or file a bug about gofmt), don't work around it.

在 Go 中我们另辟蹊径，让机器来处理大部分的格式化问题。gofmt 程序（也可用 go fmt，它以包为处理对象而非源文件）将 Go 程序按照标准风格缩进、 对齐，保留注释并在需要时重新格式化。若你想知道如何处理一些新的代码布局，请尝试运行 gofmt；若结果仍不尽人意，请重新组织你的程序（或提交有关 gofmt 的 Bug），而不必为此纠结。

As an example, there's no need to spend time lining up the comments on the fields of a structure. Gofmt will do that for you. Given the declaration

举例来说，你无需花时间将结构体中的字段注释对齐，gofmt 将为你代劳。 假如有以下声明：

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

```
type T struct {
    name string // 对象名
    value int // 对象值
}
```

gofmt will line up the columns:

gofmt 会将它按列对齐为：

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

```go
type T struct {
    name    string // 对象名
    value   int    // 对象值
}
```

All Go code in the standard packages has been formatted with gofmt.

标准包中所有的 Go 代码都已经用 gofmt 格式化过了。

Some formatting details remain. Very briefly:

还有一些关于格式化的细节，它们非常简短：

```
Indentation
  We use tabs for indentation and gofmt emits them by default. Use spaces only if you
must.
Line length
  Go has no line length limit. Don't worry about overflowing a punched card. If a line
 feels too long, wrap it and indent with an extra tab.
Parentheses
  Go needs fewer parentheses than C and Java: control structures (if, for, switch) do
not have parentheses in their syntax. Also, the operator precedence hierarchy is short
er and clearer, so
    x<<8 + y<<16
  means what the spacing implies, unlike in the other languages.
```

```
缩进
  我们使用制表符（tab）缩进，gofmt 默认也使用它。在你认为确实有必要时再使用空格。
行的长度
  Go 对行的长度没有限制，别担心打孔纸不够长。如果一行实在太长，也可进行折行并插入适当的 tab 缩进。
括号
  比起 C 和 Java，Go 所需的括号更少：控制结构（if、for 和 switch）在语法上并不需要圆括号。此外，操
作符优先级处理变得更加简洁，因此
    x<<8 + y<<16
  正表述了空格符所传达的含义。
```

Commentary

注释

Go provides C-style `/* */` block comments and C++-style `//` line comments. Line comments are the norm; block comments appear mostly as package comments, but are useful within an expression or to disable large swaths of code.

Go 语言支持 C 风格的块注释 `/* */` 和 C++ 风格的行注释 `//`。行注释更为常用，而块注释则主要用作包的注释，当然也可在禁用一大段代码时使用。

The program—and web server—godoc processes Go source files to extract documentation about the contents of the package. Comments that appear before top-level declarations, with no intervening newlines, are extracted along with the declaration to serve as explanatory text for the item. The nature and style of these comments determines the quality of the documentation godoc produces.

godoc 既是一个程序，又是一个 Web 服务器，它对 Go 的源码进行处理，并提取包中的文档内容。 出现在顶级声明之前，且与该声明之间没有空行的注释，将与该声明一起被提取出来，作为该条目的说明文档。 这些注释的类型和风格决定了 godoc 生成的文档质量。

Every package should have a package comment, a block comment preceding the package clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the godoc page and should set up the detailed documentation that follows.

每个包都应包含一段包注释，即放置在包子句前的一个块注释。对于包含多个文件的包， 包注释只需出现在其中的任一文件中即可。包注释应在整体上对该包进行介绍，并提供包的相关信息。 它将出现在 godoc 页面中的最上面，并为紧随其后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
```

```
/*
    regexp 包为正则表达式实现了一个简单的库。

    该库接受的正则表达式语法为：

    正则表达式:
        串联 { '|' 串联 }
    串联:
        { 闭包 }
    闭包:
        条目 [ '*' | '+' | '?' ]
    条目:
        '^'
        '$'
        '.'
        字符
        '[' [ '^' ] 字符遍历 ']'
        '(' 正则表达式 ')'
*/
package regexp
```

If the package is simple, the package comment can be brief.

若某个包比较简单，包注释同样可以简洁些。

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

```
// path 包实现了一些常用的工具，以便于操作用反斜杠分隔的路径.
```

Comments do not need extra formatting such as banners of stars. The generated output may not even be presented in a fixed-width font, so don't depend on spacing for alignment— godoc, like gofmt, takes care of that. The comments are uninterpreted plain text, so HTML and other annotations such as `_this_` will reproduce *verbatim* and should not be used. One adjustment godoc does do is to display indented text in a fixed-width font, suitable for program snippets. The package comment for the fmt package uses this to good effect.

注释无需进行额外的格式化，如用星号来突出等。生成的输出甚至可能无法以等宽字体显示， 因此不要依赖于空格对齐，godoc 会像 gofmt 那样处理好这一切。 注释是不会被解析的纯文本，因此像 HTML 或其它类似于 _这样_ 的东西将按照 原样 输出，因此不应使用它们。godoc 所做的调整， 就是将已缩进的文本以等宽字体显示，来适应对应的程序片段。 fmt 包的注释就用了这种不错的效果。

Depending on the context, godoc might not even reformat comments, so make sure they look good straight up: use correct spelling, punctuation, and sentence structure, fold long lines, and so on.

godoc 是否会重新格式化注释取决于上下文，因此必须确保它们看起来清晰易辨： 使用正确的拼写、标点和语句结构以及折叠长行等。

Inside a package, any comment immediately preceding a top-level declaration serves as a doc comment for that declaration. Every exported (capitalized) name in a program should have a doc comment.

在包中，任何顶级声明前面的注释都将作为该声明的文档注释。 在程序中，每个可导出（首字母大写）的名称都应该有文档注释。

Doc comments work best as complete sentences, which allow a wide variety of automated presentations. The first sentence should be a one-sentence summary that starts with the name being declared.

文档注释最好是完整的句子，这样它才能适应各种自动化的展示。 第一句应当以被声明的东西开头，并且是单句的摘要。

```
// 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) {
```

```
// Compile 用于解析正则表达式并返回，如果成功，则 Regexp 对象就可用于匹配所针对的文本。
func Compile(str string) (regexp *Regexp, err error) {
```

If the name always begins the comment, the output of godoc can usefully be run through grep. Imagine you couldn't remember the name"Compile" but were looking for the parsing function for regular expressions, so you ran the command,

若注释总是以名称开头，godoc 的输出就能通过 grep 变得更加有用。假如你记不住
"Compile" 这个名称，而又在找正则表达式的解析函数，那就可以运行

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

If all the doc comments in the package began, "This function...", grep wouldn't help you remember the name. But because the package starts each doc comment with the name, you'd see something like this, which recalls the word you're looking for.

若包中的所有文档注释都以 "此函数…" 开头，grep 就无法帮你记住此名称。 但由于每个包的
文档注释都以其名称开头，你就能看到这样的内容，它能显示你正在寻找的词语。

```
$ 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's declaration syntax allows grouping of declarations. A single doc comment can introduce a group of related constants or variables. Since the whole declaration is presented, such a comment can often be perfunctory.

Go的声明语法允许成组声明。单个文档注释应介绍一组相关的常量或变量。 由于是整体声
明，这种注释往往较为笼统。

```
// 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 ')'")
    ...
)
```

```
// 表达式解析失败后返回错误代码。
var (
    ErrInternal      = errors.New("regexp: internal error")
    ErrUnmatchedLpar = errors.New("regexp: unmatched '('")
    ErrUnmatchedRpar = errors.New("regexp: unmatched ')'")
    ...
)
```

Grouping can also indicate relationships between items, such as the fact that a set of variables is protected by a mutex.

即便是对于私有名称，也可通过成组声明来表明各项间的关系，例如某一组由互斥体保护的变量。

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

即便是对于私有名称，也可通过成组声明来表明各项间的关系，例如某一组由互斥体保护的变量。

Names

命名

Names are as important in Go as in any other language. They even have semantic effect: the visibility of a name outside a package is determined by whether its first character is upper case. It's therefore worth spending a little time talking about naming conventions in Go programs.

正如命名在其它语言中的地位，它在 Go 中同样重要。有时它们甚至会影响语义： 例如，某个名称在包外是否可见，就取决于其首个字符是否为大写字母。 因此有必要花点时间来讨论 Go 程序中的命名约定。

Package names

包名

When a package is imported, the package name becomes an accessor for the contents. After

当一个包被导入后，包名就会成了内容的访问器。在

```
import "bytes"
```

the importing package can talk about bytes.Buffer. It's helpful if everyone using the package can use the same name to refer to its contents, which implies that the package name should be good: short, concise, evocative. By convention, packages are given lower case, single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity, since everyone using your package will be typing that name. And don't worry about collisions a priori. The package name is only the default name for imports; it need not be unique across all source code, and in the rare case of a collision the importing package can choose a different name to use locally. In any case, confusion is rare because the file name in the import determines just which package is being used.

之后，被导入的包就能通过 bytes.Buffer 来引用了。 若所有人都以相同的名称来引用其内容将大有裨益， 这也就意味着包应当有个恰当的名称：其名称应该简洁明了而易于理解。按照惯例， 包应当以小写的单个单词来命名，且不应使用下划线或驼峰记法。err 的命名就是出于简短考虑的，因为任何使用该包的人都会键入该名称。 不必担心引用次序的冲突。包名就是

导入时所需的唯一默认名称， 它并不需要在所有源码中保持唯一，即便在少数发生冲突的情况下， 也可为导入的包选择一个别名来局部使用。 无论如何，通过文件名来判定使用的包，都是不会产生混淆的。

Another convention is that the package name is the base name of its source directory; the package in src/encoding/base64 is imported as "encoding/base64" but has name base64, not encoding_base64 and not encodingBase64.

另一个约定就是包名应为其源码目录的基本名称。在 src/pkg/encoding/base64 中的包应作为 "encoding/base64" 导入，其包名应为 base64， 而非 encoding_base64 或 encodingBase64。

The importer of a package will use the name to refer to its contents, so exported names in the package can use that fact to avoid stutter. (Don't use the import . notation, which can simplify tests that must run outside the package they are testing, but should otherwise be avoided.) For instance, the buffered reader type in the bufio package is called Reader, not BufReader, because users see it as bufio.Reader, which is a clear, concise name. Moreover, because imported entities are always addressed with their package name, bufio.Reader does not conflict with io.Reader. Similarly, the function to make new instances of ring.Ring— which is the definition of a constructor in Go—would normally be called NewRing, but since Ring is the only type exported by the package, and since the package is called ring, it's called just New, which clients of the package see as ring.New. Use the package structure to help you choose good names.

包的导入者可通过包名来引用其内容，因此包中的可导出名称可以此来避免冲突。 （请勿使用 import . 记法，它可以简化必须在被测试包外运行的测试， 除此之外应尽量避免使用。） 例如，bufio 包中的缓存读取器类型叫做 Reader 而非 BufReader，因为用户将它看做 bufio.Reader，这是个清楚而简洁的名称。 此外，由于被导入的项总是通过它们的包名来确定，因此 bufio.Reader 不会与 io.Reader 发生冲突。同样，用于创建 ring.Ring 的新实例的函数（这就是 Go 中的构造函数）一般会称之为 NewRing，但由于 Ring 是该包所导出的唯一类型，且该包也叫 ring，因此它可以只叫做 New，它跟在包的后面，就像 ring.New。使用包结构可以帮助你选择好的名称。

Another short example is once.Do; once.Do(setup) reads well and would not be improved by writing once.DoOrWaitUntilDone(setup). Long names don't automatically make things more readable. A helpful doc comment can often be more valuable than an extra long name.

另一个简短的例子是 once.Do，once.Do(setup) 表述足够清晰， 使用 once.DoOrWaitUntilDone(setup) 完全就是画蛇添足。 长命名并不会使其更具可读性。一份有用的说明文档通常比额外的长名更有价值。

Getters

获取器

Go doesn't provide automatic support for getters and setters. There's nothing wrong with providing getters and setters yourself, and it's often appropriate to do so, but it's neither idiomatic nor necessary to put Get into the getter's name. If you have a field called owner (lower case, unexported), the getter method should be called Owner (upper case, exported), not GetOwner. The use of upper-case names for export provides the hook to discriminate the field from the method. A setter function, if needed, will likely be called SetOwner. Both names read well in practice:

Go 并不对获取器（getter）和设置器（setter）提供自动支持。 你应当自己提供获取器和设置器，通常很值得这样做，但若要将 Get 放到获取器的名字中，既不符合习惯，也没有必要。若你有个名为 owner （小写，未导出）的字段，其获取器应当名为 Owner（大写，可导出）而非 GetOwner。大写字母即为可导出的这种规定为区分方法和字段提供了便利。 若要提供设置器方法，SetOwner 是个不错的选择。两个命名看起来都很合理：

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

Interface names

接口名

By convention, one-method interfaces are named by the method name plus an -er suffix or similar modification to construct an agent noun: Reader, Writer, Formatter, CloseNotifier etc.

按照约定，只包含一个方法的接口应当以该方法的名称加上 - er 后缀来命名，如 Reader、Writer、 Formatter、CloseNotifier 等。

There are a number of such names and it's productive to honor them and the function names they capture. Read, Write, Close, Flush, String and so on have canonical signatures and meanings. To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method String not ToString.

诸如此类的命名有很多，遵循它们及其代表的函数名会让事情变得简单。 Read、Write、Close、Flush、 String 等都具有典型的签名和意义。为避免冲突，请不要用这些名称为你的方法命名， 除非你明确知道它们的签名和意义相同。反之，若你的类型实现了的方法， 与一个众所周知的类型的方法拥有相同的含义，那就使用相同的命名。 请将字符串转换方法命名为 String 而非 ToString。

MixedCaps

驼峰记法

Finally, the convention in Go is to use MixedCaps or mixedCaps rather than underscores to write multiword names.

最后，Go 中约定使用驼峰记法 MixedCaps 或 mixedCaps。

MixedCaps

驼峰记法

Semicolons

分号

Like C, Go's formal grammar uses semicolons to terminate statements, but unlike in C, those semicolons do not appear in the source. Instead the lexer uses a simple rule to insert semicolons automatically as it scans, so the input text is mostly free of them.

和 C 一样，Go 的正式语法使用分号来结束语句；和 C 不同的是，这些分号并不在源码中出现。 取而代之，词法分析器会使用一条简单的规则来自动插入分号，因此因此源码中基本就不用分号了。

The rule is this. If the last token before a newline is an identifier (which includes words like int and float64), a basic literal such as a number or string constant, or one of the tokens

规则是这样的：若在新行前的最后一个标记为标识符（包括 int 和 float64 这类的单词）、数值或字符串常量之类的基本字面或以下标记之一

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

the lexer always inserts a semicolon after the token. This could be summarized as, "if the newline comes after a token that could end a statement, insert a semicolon".

则词法分析将始终在该标记后面插入分号。这点可以概括为："如果新行前的标记为语句的末尾，则插入分号"。

A semicolon can also be omitted immediately before a closing brace, so a statement such as

分号也可在闭括号之前直接省略，因此像

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

needs no semicolons. Idiomatic Go programs have semicolons only in places such as for loop clauses, to separate the initializer, condition, and continuation elements. They are also necessary to separate multiple statements on a line, should you write code that way.

这样的语句无需分号。通常Go程序只在诸如 for 循环子句这样的地方使用分号， 以此来将初始化器、条件及增量元素分开。如果你在一行中写多个语句，也需要用分号隔开。

One consequence of the semicolon insertion rules is that you cannot put the opening brace of a control structure (if, for, switch, or select) on the next line. If you do, a semicolon will be inserted before the brace, which could cause unwanted effects. Write them like this

警告：无论如何，你都不应将一个控制结构（if、for、switch 或 select）的左大括号放在下一行。如果这样做，就会在大括号前面插入一个分号，这可能引起不需要的效果。 你应该这样写

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

not like this

而不是这样

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

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

Control structures

控制结构

The control structures of Go are related to those of C but differ in important ways. There is no do or while loop, only a slightly generalized for; switch is more flexible; if and switch accept an optional initialization statement like that of for; break and continue statements take an optional label to identify what to break or continue; and there are new control structures including a type switch and a multiway communications multiplexer, select. The syntax is also slightly different: there are no parentheses and the bodies must always be brace-delimited.

Go 中的结构控制与 C 有许多相似之处，但其不同之处才是独到之处。 Go 不再使用 do 或 while 循环，只有一个更通用的 for；switch 要更灵活一点；if 和 switch 像 for 一样可接受可选 的初始化语句； 此外，还有一个包含类型选择和多路通信复用器的新控制结构：select。 其 语法也有些许不同：没有圆括号，而其主体必须始终使用大括号括住。

If

In Go a simple if looks like this:

在 Go 中，一个简单的 if 语句看起来像这样：

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

Mandatory braces encourage writing simple if statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a return or break.

强制的大括号促使你将简单的 if 语句分成多行。特别是在主体中包含 return 或 break 等控制 语句时，这种编码风格的好处一比便知。

Since if and switch accept an initialization statement, it's common to see one used to set up a local variable.

由于 if 和 switch 可接受初始化语句， 因此用它们来设置局部变量十分常见。

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

In the Go libraries, you'll find that when an if statement doesn't flow into the next statement—that is, the body ends in break, continue, goto, or return—the unnecessary else is omitted.

在 Go 的库中，你会发现若 if 语句不会执行到下一条语句时，亦即其执行体 以 break、continue、goto 或 return 结束时，不必要的 else 会被省略。

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

This is an example of a common situation where code must guard against a sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in return statements, the resulting code needs no else statements.

下例是一种常见的情况，代码必须防范一系列的错误条件。若控制流成功继续， 则说明程序已排除错误。由于出错时将以 return 结束， 之后的代码也就无需 else 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)
```

Redeclaration and reassignment

重新声明与再次赋值

An aside: The last example in the previous section demonstrates a detail of how the := short declaration form works. The declaration that calls os.Open reads,

题外话：上一节中最后一个示例展示了短声明 := 如何使用。 调用了 os.Open 的声明为

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

This statement declares two variables, f and err. A few lines later, the call to f.Stat reads,

该语句声明了两个变量 f 和 err。在几行之后，又通过

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

which looks as if it declares d and err. Notice, though, that err appears in both statements. This duplication is legal: err is declared by the first statement, but only re-assigned in the second. This means that the call to f.Stat uses the existing err variable declared above, and just gives it a new value.

调用了 f.Stat。它看起来似乎是声明了 d 和 err。注意，尽管两个语句中都出现了 err，但这种重复仍然是合法的：err 在第一条语句中被声明，但在第二条语句中只是被再次赋值罢了。也就是说，调用 f.Stat 使用的是前面已经声明的 err，它只是被重新赋值了而已。

In a := declaration a variable v may appear even if it has already been declared, provided:

在满足下列条件时，已被声明的变量 v 可出现在:= 声明中：

- this declaration is in the same scope as the existing declaration of v (if v is already declared in an outer scope, the declaration will create a new variable §),
- the corresponding value in the initialization is assignable to v, and
- there is at least one other variable in the declaration that is being declared anew.

- 本次声明与已声明的 v 处于同一作用域中（若 v 已在外层作用域中声明过，则此次声明会创建一个新的变量 §），
- 在初始化中与其类型相应的值才能赋予 v，且
- 在此次声明中至少另有一个变量是新声明的。

This unusual property is pure pragmatism, making it easy to use a single err value, for example, in a long if-else chain. You'll see it used often.

这个特性简直就是纯粹的实用主义体现，它使得我们可以很方面地只使用一个 err 值，例如，在一个相当长的 if-else 语句链中，你会发现它用得很频繁。

§ It's worth noting here that in Go the scope of function parameters and return values is the same as the function body, even though they appear lexically outside the braces that enclose the body.

§ 值得一提的是，即便 Go 中的函数形参和返回值在词法上处于大括号之外，但它们的作用域和该函数体仍然相同。

For

The Go for loop is similar to—but not the same as—C's. It unifies for and while and there is no do-while. There are three forms, only one of which has semicolons.

Go 的 for 循环类似于 C，但却不尽相同。它统一了 for 和 while，不再有 do-while 了。它有三种形式，但只有一种需要分号。

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

// Like a C while
for condition { }

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

```
// 如同 C 的 for 循环
for init; condition; post { }

// 如同 C 的 while 循环
for condition { }

// 如同 C 的 for(;;) 循环
for { }
```

Short declarations make it easy to declare the index variable right in the loop.

简短声明能让我们更容易在循环中声明下标变量：

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

If you're looping over an array, slice, string, or map, or reading from a channel, a range clause can manage the loop.

若你想遍历数组、切片、字符串或者映射，或从信道中读取消息，range 子句能够帮你轻松实现循环。

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

If you only need the first item in the range (the key or index), drop the second:

若你只需要该遍历中的第一个项（键或下标），去掉第二个就行了：

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

If you only need the second item in the range (the value), use the blank identifier, an underscore, to discard the first:

若你只需要该遍历中的第二个项（值），请使用空白标识符，即下划线来丢弃第一个值：

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

The blank identifier has many uses, as described in a later section.

空白标识符还有多种用法，它会在后面的小节中描述。

For strings, the range does more work for you, breaking out individual Unicode code points by parsing the UTF-8. Erroneous encodings consume one byte and produce the replacement rune U+FFFD. (The name (with associated builtin type) rune is Go terminology for a single Unicode code point. See the language specification for details.) The loop

对于字符串，range 能够提供更多便利。它能通过解析 UTF-8，将每个独立的 Unicode 码点分离出来。错误的编码将占用一个字节，并以符文 U+FFFD 来代替。（名称"符文"和内建类型 rune 是 Go 对单个 Unicode 码点的成称谓。详情见语言规范）。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)
}
```

prints

```
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
```

```go
for pos, char := range "日本\x80語" { // \x80 是个非法的UTF-8编码
    fmt.Printf("字符 %#U 始于字节位置 %d\n", char, pos)
}
```

将打印

```
字符 U+65E5 '日' 始于字节位置 0
字符 U+672C '本' 始于字节位置 3
字符 U+FFFD '�' 始于字节位置 6
字符 U+8A9E '語' 始于字节位置 7
```

Finally, Go has no comma operator and ++ and -- are statements not expressions. Thus if you want to run multiple variables in a for you should use parallel assignment (although that precludes ++ and --).

最后，Go 没有逗号操作符，而 ++ 和 -- 为语句而非表达式。 因此，若你想要在 for 中使用多个变量，应采用平行赋值的方式 （因为它会拒绝 ++ 和 --）.

```go
// 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]
}
```

```go
// 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's switch is more general than C's. The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the switch has no expression it switches on true. It's therefore possible—and idiomatic—to write an if-else-if-else chain as a switch.

Go 的 switch 比 C 的更通用。其表达式无需为常量或整数，case 语句会自上而下逐一进行求值直到匹配为止。若 switch 后面没有表达式，它将匹配 true，因此，我们可以将 if-else-if-else 链写成一个 switch，这也更符合 Go 的风格。

```go
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
}
```

There is no automatic fall through, but cases can be presented in comma-separated lists.

switch 并不会自动下溯，但 case 可通过逗号分隔来列举相同的处理条件。

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

Although they are not nearly as common in Go as some other C-like languages, break statements can be used to terminate a switch early. Sometimes, though, it's necessary to break out of a surrounding loop, not the switch, and in Go that can be accomplished by putting a label on the loop and"breaking" to that label. This example shows both uses.

尽管它们在 Go 中的用法和其它类 C 语言差不多，但 break 语句可以使 switch 提前终止。不仅是 switch，有时候也必须打破层层的循环。在 Go 中，我们只需将标签放置到循环外，然后 "蹦" 到那里即可。下面的例子展示了二者的用法。

```
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)
        }
    }
```

Of course, the continue statement also accepts an optional label but it applies only to loops.

当然，continue 语句也能接受一个可选的标签，不过它只能在循环中使用。

To close this section, here's a comparison routine for byte slices that uses two switch statements:

```
// 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
}
```

作为这一节的结束，此程序通过使用两个 switch 语句对字节数组进行比较：

```go
// Compare 按字典顺序比较两个字节切片并返回一个整数。
// 若 a == b，则结果为零；若 a < b；则结果为 -1；若 a > b，则结果为 +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
}
```

Type switch

类型选择

A switch can also be used to discover the dynamic type of an interface variable. Such a type switch uses the syntax of a type assertion with the keyword type inside the parentheses. If the switch declares a variable in the expression, the variable will have the corresponding type in each clause. It's also idiomatic to reuse the name in such cases, in effect declaring a new variable with the same name but a different type in each case.

```go
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
}
```

switch 也可用于判断接口变量的动态类型。如 类型选择 通过圆括号中的关键字 type 使用类型断言语法。若 switch 在表达式中声明了一个变量，那么该变量的每个子句中都将有该变量对应的类型。在这些 case 中重用一个名字也是符合语义的，实际上是在每个 case 里声明了一个不同类型但同名的新变量。

```go
var t interface{}
t = functionOfSomeType()
switch t := t.(type) {
default:
    fmt.Printf("unexpected type %T", t)       // %T 输出 t 是什么类型
case bool:
    fmt.Printf("boolean %t\n", t)             // t 是 bool 类型
case int:
    fmt.Printf("integer %d\n", t)             // t 是 int 类型
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t) // t 是 *bool 类型
case *int:
    fmt.Printf("pointer to integer %d\n", *t) // t 是 *int 类型
}
```

Functions

函数

Multiple return values

多值返回

One of Go's unusual features is that functions and methods can return multiple values. This form can be used to improve on a couple of clumsy idioms in C programs: in-band error returns such as -1 for EOF and modifying an argument passed by address.

Go 与众不同的特性之一就是函数和方法可返回多个值。这种形式可以改善 C 中一些笨拙的习惯：将错误值返回（例如用 -1 表示 EOF）和修改通过地址传入的实参。

In C, a write error is signaled by a negative count with the error code secreted away in a volatile location. In Go, Write can return a count and an error: "Yes, you wrote some bytes but not all of them because you filled the device". The signature of the Write method on files from package os is:

在 C 中，写入操作发生的错误会用一个负数标记，而错误码会隐藏在某个不确定的位置。 而在 Go 中，Write 会返回写入的字节数以及一个错误："是的，您写入了一些字节，但并未全部写入，因为设备已满"。 在 os 包中，File.Write 的签名为：

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

and as the documentation says, it returns the number of bytes written and a non-nil error when n != len(b). This is a common style; see the section on error handling for more examples.

正如文档所述，它返回写入的字节数，并在 n != len(b) 时返回一个非 nil 的 error 错误值。 这是一种常见的编码风格，更多示例见错误处理一节。

A similar approach obviates the need to pass a pointer to a return value to simulate a reference parameter. Here's a simple-minded function to grab a number from a position in a byte slice, returning the number and the next position.

我们可以采用一种简单的方法。来避免为模拟引用参数而传入指针。 以下简单的函数可从字节数组中的特定位置获取其值，并返回该数值和下一个位置。

```go
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
}
```

You could use it to scan the numbers in an input slice b like this:

你可以像下面这样，通过它扫描输入的切片 b 来获取数字。

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

Named result parameters

可命名结果形参

The return or result "parameters" of a Go function can be given names and used as regular variables, just like the incoming parameters. When named, they are initialized to the zero values for their types when the function begins; if the function executes a return statement with no arguments, the current values of the result parameters are used as the returned values.

Go 函数的返回值或结果"形参"可被命名，并作为常规变量使用，就像传入的形参一样。命名后，一旦该函数开始执行，它们就会被初始化为与其类型相应的零值；若该函数执行了一条不带实参的 return 语句，则结果形参的当前值将被返回。

The names are not mandatory but they can make code shorter and clearer: they're documentation. If we name the results of nextInt it becomes obvious which returned int is which.

此名称不是强制性的，但它们能使代码更加简短清晰：它们就是文档。若我们命名了 nextInt 的结果，那么它返回的 int 就值如其意了。

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

Because named results are initialized and tied to an unadorned return, they can simplify as well as clarify. Here's a version of io.ReadFull that uses them well:

由于被命名的结果已经初始化，且已经关联至无参数的返回，它们就能让代码简单而清晰。下面的 io.ReadFull 就是个很好的例子：

```go
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's defer statement schedules a function call (the deferred function) to be run immediately before the function executing the defer returns. It's an unusual but effective way to deal with situations such as resources that must be released regardless of which path a function takes to return. The canonical examples are unlocking a mutex or closing a file.

Go 的 defer 语句用于预设一个函数调用（即推迟执行函数），该函数会在执行 defer 的函数返回之前立即执行。它显得非比寻常，但却是处理一些事情的有效方式，例如无论以何种路径返回，都必须释放资源的函数。典型的例子就是解锁互斥和关闭文件。

```go
// 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.
}
```

```go
// Contents 将文件的内容作为字符串返回。
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close 会在我们结束后运行。

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append 将在后面讨论。
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // 我们在这里返回后，f 就会被关闭。
        }
    }
    return string(result), nil // 我们在这里返回后，f 就会被关闭。
}
```

Deferring a call to a function such as Close has two advantages. First, it guarantees that you will never forget to close the file, a mistake that's easy to make if you later edit the function to add a new return path. Second, it means that the close sits near the open, which is much clearer than placing it at the end of the function.

推迟诸如 Close 之类的函数调用有两点好处：第一，它能确保你不会忘记关闭文件。如果你以后又为该函数添加了新的返回路径时，这种情况往往就会发生。第二，它意味着“关闭”离“打开”很近，这总比将它放在函数结尾处要清晰明了。

The arguments to the deferred function (which include the receiver if the function is a method) are evaluated when the defer executes, not when the call executes. Besides avoiding worries about variables changing values as the function executes, this means that a single deferred call site can defer multiple function executions. Here's a silly example.

被推迟函数的实参（如果该函数为方法则还包括接收者）在推迟执行时就会求值，而不是在调用执行时才求值。这样不仅无需担心变量值在函数执行时被改变，同时还意味着单个已推迟的调用可推迟多个函数的执行。下面是个简单的例子。

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

Deferred functions are executed in LIFO order, so this code will cause 4 3 2 1 0 to be printed when the function returns. A more plausible example is a simple way to trace function execution through the program. We could write a couple of simple tracing routines like this:

被推迟的函数按照后进先出（LIFO）的顺序执行，因此以上代码在函数返回时会打印４３２１０。一个更具实际意义的例子是通过一种简单的方法，用程序来跟踪函数的执行。我们可以编写一对简单的跟踪例程：

```go
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....
}
```

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

// 像这样使用它们：
func a() {
    trace("a")
    defer untrace("a")
    // 做一些事情....
}
```

We can do better by exploiting the fact that arguments to deferred functions are evaluated when the defer executes. The tracing routine can set up the argument to the untracing routine. This example:

我们可以充分利用这个特点，即被推迟函数的实参在 defer 执行时才会被求值。 跟踪例程可针对反跟踪例程设置实参。以下例子：

```go
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()
}
```

prints

会打印

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

For programmers accustomed to block-level resource management from other languages, defer may seem peculiar, but its most interesting and powerful applications come precisely from the fact that it's not block-based but function-based. In the section on panic and recover we'll see another example of its possibilities.

对于习惯其它语言中块级资源管理的程序员，defer 似乎有点怪异，但它最有趣而强大的应用恰恰来自于其基于函数而非块的特点。在 panic 和 recover 这两节中，我们将看到关于它可能性的其它例子。

Data

数据

Allocation with new

new 分配

Go has two allocation primitives, the built-in functions new and make. They do different things and apply to different types, which can be confusing, but the rules are simple. Let's talk about new first. It's a built-in function that allocates memory, but unlike its namesakes in some other languages it does not initialize the memory, it only zeros it. That is, new(T) allocates zeroed storage for a new item of type T and returns its address, a value of type `*T` . In Go terminology, it returns a pointer to a newly allocated zero value of type T.

Go 提供了两种分配原语，即内建函数 new 和 make。 它们所做的事情不同，所应用的类型也不同。它们可能会引起混淆，但规则却很简单。 让我们先来看看 new。这是个用来分配内存的内建函数， 但与其它语言中的同名函数不同，它不会初始化内存，只会将内存置零。 也就是说，new(T) 会为类型为 T 的新项分配已置零的内存空间， 并返回它的地址，也就是一个类型为 `*T` 的值。用 Go 的术语来说，它返回一个指针， 该指针指向新分配的，类型为 T 的零值。

Since the memory returned by new is zeroed, it's helpful to arrange when designing your data structures that the zero value of each type can be used without further initialization. This means a user of the data structure can create one with new and get right to work. For example, the documentation for bytes.Buffer states that "the zero value for Buffer is an empty buffer ready to use." Similarly, sync.Mutex does not have an explicit constructor or Init method. Instead, the zero value for a sync.Mutex is defined to be an unlocked mutex.

既然 new 返回的内存已置零，那么当你设计数据结构时， 每种类型的零值就不必进一步初始化了，这意味着该数据结构的使用者只需用 new 创建一个新的对象就能正常工作。例如，bytes.Buffer 的文档中提到"零值的 Buffer 就是已准备就绪的缓冲区。" 同样，sync.Mutex 并没有显式的构造函数或 Init 方法， 而是零值的 sync.Mutex 就已经被定义为已解锁的互斥锁了。

The zero-value-is-useful property works transitively. Consider this type declaration.

"零值属性"可以带来各种好处。考虑以下类型声明。

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

Values of type SyncedBuffer are also ready to use immediately upon allocation or just declaration. In the next snippet, both p and v will work correctly without further arrangement.

SyncedBuffer 类型的值也是在声明时就分配好内存就绪了。后续代码中，p 和 v 无需进一步处理即可正确工作。

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

Constructors and composite literals

构造函数与复合字面

Sometimes the zero value isn't good enough and an initializing constructor is necessary, as in this example derived from package os.

有时零值还不够好，这时就需要一个初始化构造函数，如来自 os 包中的这段代码所示。

```go
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
}
```

There's a lot of boiler plate in there. We can simplify it using a composite literal, which is an expression that creates a new instance each time it is evaluated.

这里显得代码过于冗长。我们可通过复合字面来简化它，该表达式在每次求值时都会创建新的实例。

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

Note that, unlike in C, it's perfectly OK to return the address of a local variable; the storage associated with the variable survives after the function returns. In fact, taking the address of a composite literal allocates a fresh instance each time it is evaluated, so we can combine these last two lines.

请注意，返回一个局部变量的地址完全没有问题，这点与 C 不同。该局部变量对应的数据 在函数返回后依然有效。实际上，每当获取一个复合字面的地址时，都将为一个新的实例分配内存， 因此我们可以将上面的最后两行代码合并：

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

The fields of a composite literal are laid out in order and must all be present. However, by labeling the elements explicitly as field:value pairs, the initializers can appear in any order, with the missing ones left as their respective zero values. Thus we could say

复合字面的字段必须按顺序全部列出。但如果以 字段: 值 对的形式明确地标出元素，初始化字段时就可以按任何顺序出现，未给出的字段值将赋予零值。 因此，我们可以用如下形式：

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

As a limiting case, if a composite literal contains no fields at all, it creates a zero value for the type. The expressions new(File) and &File{} are equivalent.

少数情况下，若复合字面不包括任何字段，它将创建该类型的零值。表达式 new(File) 和 &File{} 是等价的。

Composite literals can also be created for arrays, slices, and maps, with the field labels being indices or map keys as appropriate. In these examples, the initializations work regardless of the values of Enone, Eio, and Einval, as long as they are distinct.

复合字面同样可用于创建数组、切片以及映射，字段标签是索引还是映射键则视情况而定。在下例初始化过程中，无论 Enone、Eio 和 Einval 的值是什么，只要它们的标签不同就行。

```
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"}
```

Allocation with make

make 分配

Back to allocation. The built-in function make(T, args) serves a purpose different from new(T). It creates slices, maps, and channels only, and it returns an initialized (not zeroed) value of type T (not `*T`). The reason for the distinction is that these three types represent, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity, and until those items are initialized, the slice is nil. For slices, maps, and channels, make initializes the internal data structure and prepares the value for use. For instance,

再回到内存分配上来。内建函数 make(T, args) 的目的不同于 new(T)。它只用于创建切片、映射和信道，并返回类型为 T（而非 *T ）的一个已初始化 （而非置零）的值。 出现这种用差异的原因在于，这三种类型本质上为引用数据类型，它们在使用前必须初始化。 例如，切片是一个具有三项内容的描述符，包含一个指向（数组内部）数据的指针、长度以及容量，在这三项被初始化之前，该切片为 nil。对于切片、映射和信道，make 用于初始化其内部的数据结构并准备好将要使用的值。例如，

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

allocates an array of 100 ints and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. (When making a slice, the capacity can be omitted; see the section on slices for more information.) In contrast, new([]int) returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a nil slice value.

会分配一个具有 100 个 int 的数组空间，接着创建一个长度为 10， 容量为 100 并指向该数组中前 10 个元素的切片结构。（生成切片时，其容量可以省略，更多信息见切片一节。） 与此相反，new([]int) 会返回一个指向新分配的，已置零的切片结构， 即一个指向 nil 切片值的指针。

These examples illustrate the difference between new and make.

下面的例子阐明了 new 和 make 之间的区别：

```
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)
```

```
var p *[]int = new([]int)       // 分配切片结构；*p == nil；基本没用
var v  []int = make([]int, 100) // 切片 v 现在引用了一个具有 100 个 int 元素的新数组

// 没必要的复杂：
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// 习惯用法：
v := make([]int, 100)
```

Remember that make applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with new or take the address of a variable explicitly.

请记住，make 只适用于映射、切片和信道且不返回指针。若要获得明确的指针，请使用 new 分配内存。

Arrays

数组

Arrays are useful when planning the detailed layout of memory and sometimes can help avoid allocation, but primarily they are a building block for slices, the subject of the next section. To lay the foundation for that topic, here are a few words about arrays.

在详细规划内存布局时，数组是非常有用的，有时还能避免过多的内存分配，但它们主要用作切片的构件。这是下一节的主题了，不过要先说上几句来为它做铺垫。

There are major differences between the ways arrays work in Go and C. In Go,

- Arrays are values. Assigning one array to another copies all the elements.
- In particular, if you pass an array to a function, it will receive a copy of the array, not a pointer to it.
- The size of an array is part of its type. The types [10]int and [20]int are distinct.

以下为数组在 Go 和 C 中的主要区别。在 Go 中，

- 数组是值。将一个数组赋予另一个数组会复制其所有元素。
- 特别地，若将某个数组传入某个函数，它将接收到该数组的一份副本而非指针。
- 数组的大小是其类型的一部分。类型 [10]int 和 [20]int 是不同的。

The value property can be useful but also expensive; if you want C-like behavior and efficiency, you can pass a pointer to the array.

数组为值的属性很有用，但代价高昂；若你想要 C 那样的行为和效率，你可以传递一个指向该数组的指针。

```go
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
```

```go
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)  // 注意显式的取址操作
```

But even this style isn't idiomatic Go. Use slices instead.

但这并不是 Go 的习惯用法，切片才是。

Slices

切片

Slices wrap arrays to give a more general, powerful, and convenient interface to sequences of data. Except for items with explicit dimension such as transformation matrices, most array programming in Go is done with slices rather than simple arrays.

切片通过对数组进行封装，为数据序列提供了更通用、强大而方便的接口。 除了矩阵变换这类需要明确维度的情况外，Go 中的大部分数组编程都是通过切片来完成的。

Slices hold references to an underlying array, and if you assign one slice to another, both refer to the same array. If a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array. A Read function can therefore accept a slice argument rather than a pointer and a count; the length within the slice sets an upper limit of how much data to read. Here is the signature of the Read method of the File type in package os:

切片保存了对底层数组的引用，若你将某个切片赋予另一个切片，它们会引用同一个数组。若某个函数将一个切片作为参数传入，则它对该切片元素的修改对调用者而言同样可见， 这可以理解为传递了底层数组的指针。因此，Read 函数可接受一个切片实参 而非一个指针和一个计数；切片的长度决定了可读取数据的上限。以下为 os 包中 File 类型的 Read 方法签名：

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

The method returns the number of bytes read and an error value, if any. To read into the first 32 bytes of a larger buffer buf, slice (here used as a verb) the buffer.

该方法返回读取的字节数和一个错误值（若有的话）。若要从更大的缓冲区 b 中读取前 32 个字节，只需对其进行切片即可。

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

Such slicing is common and efficient. In fact, leaving efficiency aside for the moment, the following snippet would also read the first 32 bytes of the buffer.

这种切片的方法常用且高效。若不谈效率，以下片段同样能读取该缓冲区的前 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
}
```

```
    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
    }
```

The length of a slice may be changed as long as it still fits within the limits of the underlying array; just assign it to a slice of itself. The capacity of a slice, accessible by the built-in function cap, reports the maximum length the slice may assume. Here is a function to append data to a slice. If the data exceeds the capacity, the slice is reallocated. The resulting slice is returned. The function uses the fact that len and cap are legal when applied to the nil slice, and return 0.

只要切片不超出底层数组的限制，它的长度就是可变的，只需将它赋予其自身的切片即可。切片的容量可通过内建函数 cap 获得，它将给出该切片可取得的最大长度。 以下是将数据追加到切片的函数。若数据超出其容量，则会重新分配该切片。返回值即为所得的切片。 该函数中所使用的 len 和 cap 在应用于 nil 切片时是合法的，它会返回 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
}
```

```go
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // 重新分配
        // 为了后面的增长，需分配两份。
        newSlice := make([]byte, (l+len(data))*2)
        // copy 函数是预声明的，且可用于任何切片类型。
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
    return slice
}
```

We must return the slice afterwards because, although Append can modify the elements of slice, the slice itself (the run-time data structure holding the pointer, length, and capacity) is passed by value.

最终我们必须返回切片，因为尽管 Append 可修改 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。

The idea of appending to a slice is so useful it's captured by the append built-in function. To understand that function's design, though, we need a little more information, so we'll return to it later.

向切片追加东西的想法非常有用，因此有专门的内建函数 append。 要理解该函数的设计，我们还需要一些额外的信息，我们将稍后再介绍它。

Two-dimensional slices

二维切片

Go's arrays and slices are one-dimensional. To create the equivalent of a 2D array or slice, it is necessary to define an array-of-arrays or slice-of-slices, like this:

Go 的数组和切片都是一维的。要创建等价的二维数组或切片，就必须定义一个数组的数组，或切片的切片，就像这样：

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

```
type Transform [3][3]float64  // 一个 3x3 的数组，其实是包含多个数组的一个数组。
type LinesOfText [][]byte     // 包含多个字节切片的一个切片。
```

Because slices are variable-length, it is possible to have each inner slice be a different length. That can be a common situation, as in our LinesOfText example: each line has an independent length.

由于切片长度是可变的，因此其内部可能拥有多个不同长度的切片。在我们的 LinesOfText 例子中，这是种常见的情况：每行都有其自己的长度。

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

Sometimes it's necessary to allocate a 2D slice, a situation that can arise when processing scan lines of pixels, for instance. There are two ways to achieve this. One is to allocate each slice independently; the other is to allocate a single array and point the individual slices into it. Which to use depends on your application. If the slices might grow or shrink, they should be allocated independently to avoid overwriting the next line; if not, it can be more efficient to construct the object with a single allocation. For reference, here are sketches of the two methods. First, a line at a time:

有时必须分配一个二维数组，例如在处理像素的扫描行时，这种情况就会发生。 我们有两种方式来达到这个目的。一种就是独立地分配每一个切片；而另一种就是只分配一个数组，将各个切片都指向它。采用哪种方式取决于你的应用。若切片会增长或收缩， 就应该通过独立分配来避免覆盖下一行；若不会，用单次分配来构造对象会更加高效。 以下是这两种方法的大概代码，仅供参考。首先是一次一行的：

```
// 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)
}
```

```
// 分配顶层切片。
picture := make([][]uint8, YSize) // 每 y 个单元一行。
// 遍历行，为每一行都分配切片
for i := range picture {
    picture[i] = make([]uint8, XSize)
}
```

And now as one allocation, sliced into lines:

现在是一次分配，对行进行切片：

```go
// 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:]
}
```

```go
// 分配顶层切片，和前面一样。
picture := make([][]uint8, YSize) // 每 y 个单元一行。
// 分配一个大的切片来保存所有像素
pixels := make([]uint8, XSize*YSize) // 拥有类型 []uint8，尽管图片是 [][]uint8.
// 遍历行，从剩余像素切片的前面切出每行来。
for i := range picture {
    picture[i], pixels = pixels[:XSize], pixels[XSize:]
}
```

Maps

映射

Maps are a convenient and powerful built-in data structure that associate values of one type (the key) with values of another type (the element or value) The key can be of any type for which the equality operator is defined, such as integers, floating point and complex numbers, strings, pointers, interfaces (as long as the dynamic type supports equality), structs and arrays. Slices cannot be used as map keys, because equality is not defined on them. Like slices, maps hold references to an underlying data structure. If you pass a map to a function that changes the contents of the map, the changes will be visible in the caller.

映射是方便而强大的内建数据结构，它可以关联不同类型的值。其键可以是任何相等性操作符支持的类型，如整数、浮点数、复数、字符串、指针、接口（只要其动态类型支持相等性判断）、结构以及数组。 切片不能用作映射键，因为它们的相等性还未定义。与切片一样，映射也是引用类型。 若将映射传入函数中，并更改了该映射的内容，则此修改对调用者同样可见。

Maps can be constructed using the usual composite literal syntax with colon-separated key-value pairs, so it's easy to build them during initialization.

映射可使用一般的复合字面语法进行构建，其键 - 值对使用逗号分隔，因此可在初始化时很容易地构建它们。

```go
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,
}
```

Assigning and fetching map values looks syntactically just like doing the same for arrays and slices except that the index doesn't need to be an integer.

赋值和获取映射值的语法类似于数组，不同的是映射的索引不必为整数。

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

An attempt to fetch a map value with a key that is not present in the map will return the zero value for the type of the entries in the map. For instance, if the map contains integers, looking up a non-existent key will return 0. A set can be implemented as a map with value type bool. Set the map entry to true to put the value in the set, and then test it by simple indexing.

若试图通过映射中不存在的键来取值，就会返回与该映射中项的类型对应的零值。 例如，若某个映射包含整数，当查找一个不存在的键时会返回 0。 集合可实现成一个值类型为 bool 的映射。将该映射中的项置为 true 可将该值放入集合中，此后通过简单的索引操作即可判断是否存在。

```go
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")
}
```

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

if attended[person] { // 若某人不在此映射中，则为 false
    fmt.Println(person, "正在开会")
}
```

Sometimes you need to distinguish a missing entry from a zero value. Is there an entry for "UTC" or is that the empty string because it's not in the map at all? You can discriminate with a form of multiple assignment.

有时你需要区分某项是不存在还是其值为零值。如对于一个值本应为零的 "UTC" 条目，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。

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

For obvious reasons this is called the "comma ok" idiom. In this example, if tz is present, seconds will be set appropriately and ok will be true; if not, seconds will be set to zero and ok will be false. Here's a function that puts it together with a nice error report:

显然，我们可称之为“逗号 ok”惯用法。在下面的例子中，若 tz 存在，seconds 就会被赋予适当的值，且 ok 会被置为 true；若不存在，seconds 则会被置为零，而 ok 会被置为 false。

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

To test for presence in the map without worrying about the actual value, you can use the blank identifier (_) in place of the usual variable for the value.

若仅需判断映射中是否存在某项而不关心实际的值，可使用 空白标识符 （ _ ）来代替该值的一般变量。

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

To delete a map entry, use the delete built-in function, whose arguments are the map and the key to be deleted. It's safe to do this even if the key is already absent from the map.

要删除映射中的某项，可使用内建函数 delete，它以映射及要被删除的键为实参。 即便对应的键不在该映射中，此操作也是安全的。

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

```
delete(timeZone, "PDT")  // 现在用标准时间
```

Printing

打印

Formatted printing in Go uses a style similar to C's printf family but is richer and more general. The functions live in the fmt package and have capitalized names: fmt.Printf, fmt.Fprintf, fmt.Sprintf and so on. The string functions (Sprintf etc.) return a string rather than filling in a provided buffer.

Go 采用的格式化打印风格和 C 的 printf 族类似，但却更加丰富而通用。 这些函数位于 fmt 包中，且函数名首字母均为大写：如 fmt.Printf、fmt.Fprintf，fmt.Sprintf 等。 字符串函数（Sprintf 等）会返回一个字符串，而非填充给定的缓冲区。

You don't need to provide a format string. For each of Printf, Fprintf and Sprintf there is another pair of functions, for instance Print and Println. These functions do not take a format string but instead generate a default format for each argument. The Println versions also insert a blank between arguments and append a newline to the output while the Print versions add blanks only if the operand on neither side is a string. In this example each line produces the same output.

你无需提供一个格式字符串。每个 Printf、Fprintf 和 Sprintf 都分别对应另外的函数，如 Print 与 Println。 这些函数并不接受格式字符串，而是为每个实参生成一种默认格式。Println 系列的函数还会在实参中插入空格，并在输出时追加一个换行符，而 Print 版本仅在操作数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都是一样的。

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

The formatted print functions fmt.Fprint and friends take as a first argument any object that implements the io.Writer interface; the variables os.Stdout and os.Stderr are familiar instances.

fmt.Fprint 一类的格式化打印函数可接受任何实现了 io.Writer 接口的对象作为第一个实参；变量 os.Stdout 与 os.Stderr 都是人们熟知的例子。

Here things start to diverge from C. First, the numeric formats such as %d do not take flags for signedness or size; instead, the printing routines use the type of the argument to decide these properties.

从这里开始，就与 C 有些不同了。首先，像 %d 这样的数值格式并不接受表示符号或大小的标记，打印例程会根据实参的类型来决定这些属性。

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

prints

将打印

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

If you just want the default conversion, such as decimal for integers, you can use the catchall format %v (for "value"); the result is exactly what Print and Println would produce. Moreover, that format can print any value, even arrays, slices, structs, and maps. Here is a print statement for the time zone map defined in the previous section.

若你只想要默认的转换，如使用十进制的整数，你可以使用通用的格式 %v（对应"值"）；其结果与 Print 和 Println 的输出完全相同。此外，这种格式还能打印任意值，甚至包括数组、结构体和映射。 以下是打印上一节中定义的时区映射的语句。

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

which gives output

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

```go
fmt.Printf("%v\n", timeZone)  // 或只用 fmt.Println(timeZone)
```

这会输出

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

For maps the keys may be output in any order, of course. When printing a struct, the modified format %+v annotates the fields of the structure with their names, and for any value the alternate format %#v prints the value in full Go syntax.

当然，映射中的键可能按任意顺序输出。当打印结构体时，改进的格式 %+v 会为结构体的每个字段添上字段名，而另一种格式 %#v 将完全按照 Go 的语法打印值。

```
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)
```

prints

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":-25200}
```

(Note the ampersands.) That quoted string format is also available through %q when applied to a value of type string or []byte. The alternate format %#q will use backquotes instead if possible. (The %q format also applies to integers and runes, producing a single-quoted rune constant.) Also, %x works on strings, byte arrays and byte slices as well as on integers, generating a long hexadecimal string, and with a space in the format (% x) it puts spaces between the bytes.

（请注意其中的 & 符号）当遇到 string 或 []byte 值时，可使用 %q 产生带引号的字符串；而格式 %#q 会尽可能使用反引号。（%q 格式也可用于整数和符文，它会产生一个带单引号的符文常量。）此外，%x 还可用于字符串、字节数组以及整数，并生成一个很长的十六进制字符串，而带空格的格式（% x）还会在字节之间插入空格。

Another handy format is %T, which prints the type of a value.

另一种实用的格式是 %T，它会打印某个值的类型.

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

会打印

```
map[string] int
```

If you want to control the default format for a custom type, all that's required is to define a method with the signature String() string on the type. For our simple type T, that might look like this.

若你想控制自定义类型的默认格式，只需为该类型定义一个具有 String() string 签名的方法。对于我们简单的类型 T，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)
```

to print in the format

会打印出如下格式：

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

(If you need to print values of type T as well as pointers to T, the receiver for String must be of value type; this example used a pointer because that's more efficient and idiomatic for struct types. See the section below on pointers vs. value receivers for more information.)

（如果你需要像指向 T 的指针那样打印类型 T 的值， String 的接收者就必须是值类型的；上面的例子中接收者是一个指针， 因为这对结构来说更高效而通用。更多详情见 指针 vs. 值接收者 一节.）

Our String method is able to call Sprintf because the print routines are fully reentrant and can be wrapped this way. There is one important detail to understand about this approach, however: don't construct a String method by calling Sprintf in a way that will recur into your String method indefinitely. This can happen if the Sprintf call attempts to print the receiver directly as a string, which in turn will invoke the method again. It's a common and easy mistake to make, as this example shows.

我们的 String 方法也可调用 Sprintf， 因为打印例程可以完全重入并按这种方式封装。不过要理解这种方式，还有一个重要的细节：请勿通过调用 Sprintf 来构造 String 方法，因为它会无限递归你的的 String 方法。

```
type MyString string

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

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m) // 错误：会无限递归
}
```

It's also easy to fix: convert the argument to the basic string type, which does not have the method.

要解决这个问题也很简单：将该实参转换为基本的字符串类型，它没有这个方法。

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

```
type MyString string
func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", string(m)) // 可以：注意转换
}
```

In the initialization section we'll see another technique that avoids this recursion.

在 初始化 一节中，我们将看到避免这种递归的另一种技术。

Another printing technique is to pass a print routine's arguments directly to another such routine. The signature of Printf uses the type ...interface{} for its final argument to specify that an arbitrary number of parameters (of arbitrary type) can appear after the format.

另一种打印技术就是将打印例程的实参直接传入另一个这样的例程。Printf 的签名为其最后的实参使用了 ...interface{} 类型，这样格式的后面就能出现任意数量，任意类型的形参了。

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

Within the function Printf, v acts like a variable of type []interface{} but if it is passed to another variadic function, it acts like a regular list of arguments. Here is the implementation of the function log.Println we used above. It passes its arguments directly to fmt.Sprintln for

the actual formatting.

在 Printf 函数中，v 看起来更像是 []interface{} 类型的变量，但如果将它传递到另一个变参函数中，它就像是常规实参列表了。 以下是我们之前用过的 log.Println 的实现。它直接将其实参传递给 fmt.Sprintln 进行实际的格式化。

```go
// 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)
}
```

```go
// Println 通过 fmt.Println 的方式将日志打印到标准记录器。
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))  // Output 接受形参 (int, string)
}
```

We write ... after v in the nested call to Sprintln to tell the compiler to treat v as a list of arguments; otherwise it would just pass v as a single slice argument.

在该 Sprintln 嵌套调用中，我们将 ... 写在 v 之后来告诉编译器将 v 视作一个实参列表，否则它会将 v 当做单一的切片实参来传递。

There's even more to printing than we've covered here. See the godoc documentation for package fmt for the details.

还有很多关于打印知识点没有提及。详情请参阅 godoc 对 fmt 包的说明文档。

By the way, a ... parameter can be of a specific type, for instance ...int for a min function that chooses the least of a list of integers:

顺便一提，... 形参可指定具体的类型，例如从整数列表中选出最小值的函数 min，其形参可为 ...int 类型。

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

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

Append

追加

Now we have the missing piece we needed to explain the design of the append built-in function. The signature of append is different from our custom Append function above. Schematically, it's like this:

现在我们要对内建函数 append 的设计进行补充说明。append 函数的签名不同于前面我们自定义的 Append 函数。大致来说，它就像这样：

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

where T is a placeholder for any given type. You can't actually write a function in Go where the type T is determined by the caller. That's why append is built in: it needs support from the compiler.

其中的 T 为任意给定类型的占位符。实际上，你无法在 Go 中编写一个类型 T 由调用者决定的函数。这也就是为何 append 为内建函数的原因：它需要编译器的支持。 What append does is append the elements to the end of the slice and return the result. The result needs to be returned because, as with our hand-written Append, the underlying array may change. This simple example

append 会在切片末尾追加元素并返回结果。我们必须返回结果，原因与我们手写的 Append 一样，即底层数组可能会被改变。以下简单的例子

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

prints [1 2 3 4 5 6]. So append works a little like Printf, collecting an arbitrary number of arguments.

将打印 [1 2 3 4 5 6]。因此 append 有点像 Printf 那样，可接受任意数量的实参。

But what if we wanted to do what our Append does and append a slice to a slice? Easy: use ... at the call site, just as we did in the call to Output above. This snippet produces identical output to the one above.

但如果我们要像 Append 那样将一个切片追加到另一个切片中呢？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相同。

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

Without that ..., it wouldn't compile because the types would be wrong; y is not of type int.

如果没有 ...，它就会由于类型错误而无法编译，因为 y 不是 int 类型的。

Initialization

初始化

Although it doesn't look superficially very different from initialization in C or C++, initialization in Go is more powerful. Complex structures can be built during initialization and the ordering issues among initialized objects, even among different packages, are handled correctly.

尽管从表面上看，Go 的初始化过程与 C 或 C++ 并不算太大，但它确实更为强大。 在初始化过程中，不仅可以构建复杂的结构，还能正确处理不同包对象间的初始化顺序。

Constants

常量

Constants in Go are just that—constant. They are created at compile time, even when defined as locals in functions, and can only be numbers, characters (runes), strings or booleans. Because of the compile-time restriction, the expressions that define them must be constant expressions, evaluatable by the compiler. For instance, 1<<3 is a constant expression, while math.Sin(math.Pi/4) is not because the function call to math.Sin needs to happen at run time.

Go 中的常量就是不变量。它们在编译时创建，即便它们可能是函数中定义的局部变量。 常量只能是数字、字符（符文）、字符串或布尔值。由于编译时的限制， 定义它们的表达式必须也是可被编译器求值的常量表达式。例如 1<<3 就是一个常量表达式，而 math.Sin(math.Pi/4) 则不是，因为对 math.Sin 的函数调用在运行时才会发生。

In Go, enumerated constants are created using the iota enumerator. Since iota can be part of an expression and expressions can be implicitly repeated, it is easy to build intricate sets of values.

在 Go 中，枚举常量使用枚举器 iota 创建。由于 iota 可为表达式的一部分，而表达式可以被隐式地重复，这样也就更容易构建复杂的值的集合了。

```
type ByteSize float64

const (
    // 通过赋予空白标识符来忽略第一个值
    _               = iota // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

The ability to attach a method such as String to any user-defined type makes it possible for arbitrary values to format themselves automatically for printing. Although you'll see it most often applied to structs, this technique is also useful for scalar types such as floating-point types like ByteSize.

由于可将 String 之类的方法附加在用户定义的类型上，因此它就为打印时自动格式化任意值提供了可能性，即便是作为一个通用类型的一部分。尽管你常常会看到这种技术应用于结构体，但它对于像 ByteSize 之类的浮点数标量等类型也是有用的。

```
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)
}
```

The expression YB prints as 1.00YB, while ByteSize(1e13) prints as 9.09TB.

表达式 YB 会打印出 1.00YB，而 ByteSize(1e13) 则会打印出 9.09。

The use here of Sprintf to implement ByteSize's String method is safe (avoids recurring indefinitely) not because of a conversion but because it calls Sprintf with %f, which is not a string format: Sprintf will only call the String method when it wants a string, and %f wants a floating-point value.

在这里用 Sprintf 实现 ByteSize 的 String 方法很安全（不会无限递归），这倒不是因为类型转换，而是它以 %f 调用了 Sprintf，它并不是一种字符串格式：Sprintf 只会在它需要字符串时才调用 String 方法，而 %f 需要一个浮点数值。

Variables

变量

Variables can be initialized just like constants but the initializer can be a general expression computed at run time.

变量的初始化与常量类似，但其初始值也可以是在运行时才被计算的一般表达式。

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

The init function

init 函数

Finally, each source file can define its own niladic init function to set up whatever state is required. (Actually each file can have multiple init functions.) And finally means finally: init is called after all the variable declarations in the package have evaluated their initializers, and those are evaluated only after all the imported packages have been initialized.

最后，每个源文件都可以通过定义自己的无参数 init 函数来设置一些必要的状态。（其实每个文件都可以拥有多个 init 函数。）而它的结束就意味着初始化结束：只有该包中的所有变量声明都通过它们的初始化器求值后 init 才会被调用，而那些 init 只有在所有已导入的包都被初始化后才会被求值。

Besides initializations that cannot be expressed as declarations, a common use of init functions is to verify or repair correctness of the program state before real execution begins.

除了那些不能被表示成声明的初始化外，init 函数还常被用在程序真正开始执行前，检验或校正程序的状态。

```
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")
}
```

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

Methods

方法

Pointers vs. Values

指针 **vs.** 值

As we saw with ByteSize, methods can be defined for any named type (except a pointer or an interface); the receiver does not have to be a struct.

正如 ByteSize 那样，我们可以为任何已命名的类型（除了指针或接口）定义方法；接收者可不必为结构体。

In the discussion of slices above, we wrote an Append function. We can define it as a method on slices instead. To do this, we first declare a named type to which we can bind the method, and then make the receiver for the method a value of that type.

在之前讨论切片时，我们编写了一个 Append 函数。 我们也可将其定义为切片的方法。为此，我们首先要声明一个已命名的类型来绑定该方法，然后使该方法的接收者成为该类型的值。

```
type ByteSlice []byte

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

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // 主体和前面相同。
}
```

This still requires the method to return the updated slice. We can eliminate that clumsiness by redefining the method to take a pointer to a ByteSlice as its receiver, so the method can overwrite the caller's slice.

我们仍然需要该方法返回更新后的切片。为了消除这种不便，我们可通过重新定义该方法，将一个指向 ByteSlice 的指针作为该方法的接收者， 这样该方法就能重写调用者提供的切片了。

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

```go
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // 主体和前面相同，但没有 return。
    *p = slice
}
```

In fact, we can do even better. If we modify our function so it looks like a standard Write method, like this,

其实我们做得更好。若我们将函数修改为与标准 Write 类似的方法，就像这样，

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

```go
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // 依旧和前面相同。
    *p = slice
    return len(data), nil
}
```

then the type `*ByteSlice` satisfies the standard interface io.Writer, which is handy. For instance, we can print into one.

那么类型 `*ByteSlice` 就满足了标准的 io.Writer 接口，这将非常实用。 例如，我们可以通过打印将内容写入。

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

We pass the address of a ByteSlice because only `*ByteSlice` satisfies io.Writer. The rule about pointers vs. values for receivers is that value methods can be invoked on pointers and values, but pointer methods can only be invoked on pointers.

我们将 ByteSlice 的地址传入，因为只有 `*ByteSlice` 才满足 io.Writer。以指针或值为接收者的区别在于：值方法可通过指针和值调用， 而指针方法只能通过指针来调用。

This rule arises because pointer methods can modify the receiver; invoking them on a value would cause the method to receive a copy of the value, so any modifications would be discarded. The language therefore disallows this mistake. There is a handy exception, though. When the value is addressable, the language takes care of the common case of invoking a pointer method on a value by inserting the address operator automatically. In our example, the variable b is addressable, so we can call its Write method with just b.Write. The compiler will rewrite that to (&b).Write for us.

之所以会有这条规则是因为指针方法可以修改接收者；通过值调用它们会导致方法接收到该值的副本， 因此任何修改都将被丢弃，因此该语言不允许这种错误。不过有个方便的例外：若该值是可寻址的， 那么该语言就会自动插入取址操作符来对付一般的通过值调用的指针方法。在我们的例子中，变量 b 是可寻址的，因此我们只需通过 b.Write 来调用它的 Write 方法，编译器会将它重写为 (&b).Write。

By the way, the idea of using Write on a slice of bytes is central to the implementation of bytes.Buffer.

顺便一提，在字节切片上使用 Write 的想法已被 bytes.Buffer 所实现。

Interfaces and other types

接口与其它类型

接口

Interfaces in Go provide a way to specify the behavior of an object: if something can do this, then it can be used here. We've seen a couple of simple examples already; custom printers can be implemented by a String method while Fprintf can generate output to anything with a Write method. Interfaces with only one or two methods are common in Go code, and are usually given a name derived from the method, such as io.Writer for something that implements Write.

Go 中的接口为指定对象的行为提供了一种方法：如果某样东西可以完成这个，那么它就可以用在这里。我们已经见过许多简单的示例了；通过实现 String 方法，我们可以自定义打印函数，而通过 Write 方法，Fprintf 则能对任何对象产生输出。在 Go 代码中，仅包含一两种方法的接口很常见，且其名称通常来自于实现它的方法，如 io.Writer 就是实现了 Write 的一类对象。

A type can implement multiple interfaces. For instance, a collection can be sorted by the routines in package sort if it implements sort.Interface, which contains Len(), Less(i, j int) bool, and Swap(i, j int), and it could also have a custom formatter. In this contrived example Sequence satisfies both.

每种类型都能实现多个接口。例如一个实现了 sort.Interface 接口的集合就可通过 sort 包中的例程进行排序。该接口包括 Len()、Less(i, j int) bool 以及 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。

```go
type Sequence []int

// Methods required by sort.Interface.
// sort.Interface 所需的方法。
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]
}

// Method for printing - sorts the elements before printing.
// 用于打印的方法 - 在打印前对元素进行排序。
func (s Sequence) String() string {
    sort.Sort(s)
    str := "["
    for i, elem := range s {
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

Conversions

类型转换

The String method of Sequence is recreating the work that Sprint already does for slices. We can share the effort if we convert the Sequence to a plain []int before calling Sprint.

Sequence 的 String 方法重新实现了 Sprint 为切片实现的功能。若我们在调用 Sprint 之前将 Sequence 转换为纯粹的 []int，就能共享已实现的功能。

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

This method is another example of the conversion technique for calling Sprintf safely from a String method. Because the two types (Sequence and []int) are the same if we ignore the type name, it's legal to convert between them. The conversion doesn't create a new value, it

just temporarily acts as though the existing value has a new type. (There are other legal conversions, such as from integer to floating point, that do create a new value.)

该方法是通过类型转换技术，在 String 方法中安全调用 Sprintf 的另个一例子。若我们忽略类型名的话，这两种类型（Sequence 和 []int）其实是相同的，因此在二者之间进行转换是合法的。 转换过程并不会创建新值，它只是值暂让现有的时看起来有个新类型而已。 （还有些合法转换则会创建新值，如从整数转换为浮点数等。）

It's an idiom in Go programs to convert the type of an expression to access a different set of methods. As an example, we could use the existing type sort.IntSlice to reduce the entire example to this:

在 Go 程序中，为访问不同的方法集而进行类型转换的情况非常常见。 例如，我们可使用现有的 sort.IntSlice 类型来简化整个示例：

```go
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))
}
```

```go
type Sequence []int

// // 用于打印的方法 - 在打印前对元素进行排序。
func (s Sequence) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

Now, instead of having Sequence implement multiple interfaces (sorting and printing), we're using the ability of a data item to be converted to multiple types (Sequence, sort.IntSlice and []int), each of which does some part of the job. That's more unusual in practice but can be effective.

现在，不必让 Sequence 实现多个接口（排序和打印）， 我们可通过将数据条目转换为多种类型（Sequence、sort.IntSlice 和 []int）来使用相应的功能，每次转换都完成一部分工作。 这在实践中虽然有些不同寻常，但往往却很有效。

Interface conversions and type assertions

接口转换与类型断言

Type switches are a form of conversion: they take an interface and, for each case in the switch, in a sense convert it to the type of that case. Here's a simplified version of how the code under fmt.Printf turns a value into a string using a type switch. If it's already a string, we want the actual string value held by the interface, while if it has a String method we want the result of calling the method.

类型选择 是类型转换的一种形式：它接受一个接口，在选择（switch）中根据其判断选择对应的情况（case），并在某种意义上将其转换为该种类型。以下代码为 fmt.Printf 通过类型选择将值转换为字符串的简化版。若它已经为字符串，我们需要该接口中实际的字符串值；若它有 String 方法，我们则需要调用该方法所得的结果。

```go
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()
}
```

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

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

The first case finds a concrete value; the second converts the interface into another interface. It's perfectly fine to mix types this way.

第一种情况获取具体的值，第二种将该接口转换为另一个接口。这种方式对于混合类型来说非常完美。

What if there's only one type we care about? If we know the value holds a string and we just want to extract it? A one-case type switch would do, but so would a type assertion. A type assertion takes an interface value and extracts from it a value of the specified explicit type. The syntax borrows from the clause opening a type switch, but with an explicit type rather than the type keyword:

若我们只关心一种类型呢？若我们知道该值拥有一个 string 而想要提取它呢？只需一种情况的类型选择就行，但它需要类型断言。类型断言接受一个接口值，并从中提取指定的明确类型的值。其语法借鉴自类型选择开头的子句，但它需要一个明确的类型，而非 type 关键字：

```
value.(typeName)
```

and the result is a new value with the static type typeName. That type must either be the concrete type held by the interface, or a second interface type that the value can be converted to. To extract the string we know is in the value, we could write:

而其结果则是拥有静态类型 typeName 的新值。该类型必须为该接口所拥有的具体类型，或者该值可转换成的第二种接口类型。要提取我们知道在该值中的字符串，可以这样：

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

But if it turns out that the value does not contain a string, the program will crash with a run-time error. To guard against that, use the "comma, ok" idiom to test, safely, whether the value is a string:

但若它所转换的值中不包含字符串，该程序就会以运行时错误崩溃。为避免这种情况，需使用“逗号, ok”惯用测试它能安全地判断该值是否为字符串：

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

```
str, ok := value.(string)
if ok {
    fmt.Printf("字符串值为 %q\n", str)
} else {
    fmt.Printf("该值非字符串\n")
}
```

If the type assertion fails, str will still exist and be of type string, but it will have the zero value, an empty string.

若类型断言失败，str 将继续存在且为字符串类型，但它将拥有零值，即空字符串。

As an illustration of the capability, here's an if-else statement that's equivalent to the type switch that opened this section.

作为对能量的说明，这里有个 if-else 语句，它等价于本节开头的类型选择。

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

通用性

If a type exists only to implement an interface and has no exported methods beyond that interface, there is no need to export the type itself. Exporting just the interface makes it clear that it's the behavior that matters, not the implementation, and that other implementations with different properties can mirror the behavior of the original type. It also avoids the need to repeat the documentation on every instance of a common method.

若某种现有的类型仅实现了一个接口，且除此之外并无可导出的方法，则该类型本身就无需导出。 仅导出该接口能让我们更专注于其行为而非实现，其它属性不同的实现则能镜像该原始类型的行为。 这也能够避免为每个通用接口的实例重复编写文档。

In such cases, the constructor should return an interface value rather than the implementing type. As an example, in the hash libraries both crc32.NewIEEE and adler32.New return the interface type hash.Hash32. Substituting the CRC-32 algorithm for Adler-32 in a Go program requires only changing the constructor call; the rest of the code is unaffected by the change of algorithm.

在这种情况下，构造函数应当返回一个接口值而非实现的类型。例如在 hash 库中，crc32.NewIEEE 和 adler32.New 都返回接口类型 hash.Hash32。要在 Go 程序中用 Adler-32 算法替代 CRC-32， 只需修改构造函数调用即可，其余代码则不受算法改变的影响。

A similar approach allows the streaming cipher algorithms in the various crypto packages to be separated from the block ciphers they chain together. The Block interface in the crypto/cipher package specifies the behavior of a block cipher, which provides encryption of a single block of data. Then, by analogy with the bufio package, cipher packages that implement this interface can be used to construct streaming ciphers, represented by the Stream interface, without knowing the details of the block encryption.

同样的方式能将 crypto 包中多种联系在一起的流密码算法与块密码算法分开。 crypto/cipher 包中的 Block 接口指定了块密码算法的行为， 它为单独的数据块提供加密。接着，和 bufio 包类似，任何实现了该接口的密码包都能被用于构造以 Stream 为接口表示的流密码，而无需知道块加密的细节。

The crypto/cipher interfaces look like this:

crypto/cipher 接口看其来就像这样：

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

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

Here's the definition of the counter mode (CTR) stream, which turns a block cipher into a streaming cipher; notice that the block cipher's details are abstracted away:

这是计数器模式 CTR 流的定义，它将块加密改为流加密，注意块加密的细节已被抽象化了。

```
// 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 返回一个 Stream，其加密 / 解密使用计数器模式中给定的 Block 进行。
// iv 的长度必须与 Block 的块大小相同。
func NewCTR(block Block, iv []byte) Stream
```

NewCTR applies not just to one specific encryption algorithm and data source but to any implementation of the Block interface and any Stream. Because they return interface values, replacing CTR encryption with other encryption modes is a localized change. The constructor calls must be edited, but because the surrounding code must treat the result only as a Stream, it won't notice the difference.

NewCTR 的应用并不仅限于特定的加密算法和数据源，它适用于任何对 Block 接口和 Stream 的实现。因为它们返回接口值，所以用其它加密模式来代替 CTR 只需做局部的更改。构造函数的调用过程必须被修改，但由于其周围的代码只能将它看做 Stream，因此它们不会注意到其中的区别。

Interfaces and methods

接口和方法

Since almost anything can have methods attached, almost anything can satisfy an interface. One illustrative example is in the http package, which defines the Handler interface. Any object that implements Handler can serve HTTP requests.

由于几乎任何类型都能添加方法，因此几乎任何类型都能满足一个接口。一个很直观的例子就是 http 包中定义的 Handler 接口。任何实现了 Handler 的对象都能够处理 HTTP 请求。

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

ResponseWriter is itself an interface that provides access to the methods needed to return the response to the client. Those methods include the standard Write method, so an http.ResponseWriter can be used wherever an io.Writer can be used. Request is a struct containing a parsed representation of the request from the client.

ResponseWriter 接口提供了对方法的访问，这些方法需要响应客户端的请求。 由于这些方法包含了标准的 Write 方法，因此 http.ResponseWriter 可用于任何 io.Writer 适用的场景。 Request 结构体包含已解析的客户端请求。

For brevity, let's ignore POSTs and assume HTTP requests are always GETs; that simplification does not affect the way the handlers are set up. Here's a trivial but complete implementation of a handler to count the number of times the page is visited.

为简单起见，我们假设所有的 HTTP 请求都是 GET 方法，而忽略 POST 方法， 这种简化不会影响处理程序的建立方式。这里有个短小却完整的处理程序实现， 它用于记录某个页面被访问的次数。

```
// 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)
}
```

```
// 简单的计数器服务。
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)
}
```

(Keeping with our theme, note how Fprintf can print to an http.ResponseWriter.) For reference, here's how to attach such a server to a node on the URL tree.

（紧跟我们的主题，注意 Fprintf 如何能输出到 http.ResponseWriter。） 作为参考，这里演示了如何将这样一个服务器添加到 URL 树的一个节点上。

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

But why make Counter a struct? An integer is all that's needed. (The receiver needs to be a pointer so the increment is visible to the caller.)

但为什么 Counter 要是结构体呢？一个整数就够了。 An integer is all that's needed. （接收者必须为指针，增量操作对于调用者才可见。）

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

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

```
// 简单的计数器服务。
type Counter int

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

What if your program has some internal state that needs to be notified that a page has been visited? Tie a channel to the web page.

当页面被访问时，怎样通知你的程序去更新一些内部状态呢？为 Web 页面绑定个信道吧。

```
// 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")
}
```

```
// 每次浏览该信道都会发送一个提醒。
// （可能需要带缓冲的信道。）
type Chan chan *http.Request

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

Finally, let's say we wanted to present on /args the arguments used when invoking the server binary. It's easy to write a function to print the arguments.

最后，假设我们需要输出调用服务器二进制程序时使用的实参 /args。很简单，写个打印实参的函数就行了。

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

How do we turn that into an HTTP server? We could make ArgServer a method of some type whose value we ignore, but there's a cleaner way. Since we can define a method for any type except pointers and interfaces, we can write a method for a function. The http package contains this code:

我们如何将它转换为 HTTP 服务器呢？我们可以将 ArgServer 实现为某种可忽略值的方法，不过还有种更简单的方法。 既然我们可以为除指针和接口以外的任何类型定义方法，同样也能为一个函数写一个方法。 http 包包含以下代码：

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

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

```
// HandlerFunc 类型是一个适配器，它允许将普通函数用做 HTTP 处理程序。
// 若 f 是个具有适当签名的函数，HandlerFunc(f) 就是个调用 f 的处理程序对象。
type HandlerFunc func(ResponseWriter, *Request)

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

HandlerFunc is a type with a method, ServeHTTP, so values of that type can serve HTTP requests. Look at the implementation of the method: the receiver is a function, f, and the method calls f. That may seem odd but it's not that different from, say, the receiver being a channel and the method sending on the channel.

HandlerFunc 是个具有 ServeHTTP 方法的类型， 因此该类型的值就能处理 HTTP 请求。我们来看看该方法的实现：接收者是一个函数 f，而该方法调用 f。这看起来很奇怪，但不必大惊小怪， 区别在于接收者变成了一个信道，而方法通过该信道发送消息。

To make ArgServer into an HTTP server, we first modify it to have the right signature.

为了将 ArgServer 实现成 HTTP 服务器，首先我们得让它拥有合适的签名。

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

```
// 实参服务器。
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}
```

ArgServer now has same signature as HandlerFunc, so it can be converted to that type to access its methods, just as we converted Sequence to IntSlice to access IntSlice.Sort. The code to set it up is concise:

ArgServer 和 HandlerFunc 现在拥有了相同的签名， 因此我们可将其转换为这种类型以访问它的方法，就像我们将 Sequence 转换为 IntSlice 以访问 IntSlice.Sort 那样。 建立代码非常简单：

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

When someone visits the page /args, the handler installed at that page has value ArgServer and type HandlerFunc. The HTTP server will invoke the method ServeHTTP of that type, with ArgServer as the receiver, which will in turn call ArgServer (via the invocation f(c, req) inside HandlerFunc.ServeHTTP). The arguments will then be displayed.

当有人访问 /args 页面时，安装到该页面的处理程序就有了值 ArgServer 和类型 HandlerFunc。 HTTP 服务器会以 ArgServer 为接收者，调用该类型的 ServeHTTP 方法，它会反过来调用 ArgServer（通过 f(c, req)），接着实参就会被显示出来。

In this section we have made an HTTP server from a struct, an integer, a channel, and a function, all because interfaces are just sets of methods, which can be defined for (almost) any type.

在本节中，我们通过一个结构体，一个整数，一个信道和一个函数，建立了一个 HTTP 服务器， 这一切都是因为接口只是方法的集和，而几乎任何类型都能定义方法。

The blank identifier

空白标识符

We've mentioned the blank identifier a couple of times now, in the context of for range loops and maps. The blank identifier can be assigned or declared with any value of any type, with the value discarded harmlessly. It's a bit like writing to the Unix /dev/null file: it represents a write-only value to be used as a place-holder where a variable is needed but the actual value is irrelevant. It has uses beyond those we've seen already.

我们在 for-range 循环和 映射 中提过几次空白标识符。 空白标识符可被赋予或声明为任何类型的任何值，而其值会被无害地丢弃。它有点像 Unix 中的 /dev/null 文件：它表示只写的值，在需要变量但不需要实际值的地方用作占位符。 我们在前面已经见过它的用法了。

The blank identifier in multiple assignment

多重赋值中的空白标识符

The use of a blank identifier in a for range loop is a special case of a general situation: multiple assignment.

for range 循环中对空表标识符的用法是一种具体情况，更一般的情况即为多重赋值。

If an assignment requires multiple values on the left side, but one of the values will not be used by the program, a blank identifier on the left-hand-side of the assignment avoids the need to create a dummy variable and makes it clear that the value is to be discarded. For instance, when calling a function that returns a value and an error, but only the error is important, use the blank identifier to discard the irrelevant value.

若某次赋值需要匹配多个左值，但其中某个变量不会被程序使用，那么用空白标识符来代替该变量可避免创建无用的变量，并能清楚地表明该值将被丢弃。 例如，当调用某个函数时，它会返回一个值和一个错误，但只有错误很重要， 那么可使用空白标识符来丢弃无关的值。

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

Occasionally you'll see code that discards the error value in order to ignore the error; this is terrible practice. Always check error returns; they're provided for a reason.

你偶尔会看见为忽略错误而丢弃错误值的代码，这是种糟糕的实践。请务必检查错误返回，它们会提供错误的理由。

```
// 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)
}
```

```
// 烂代码！若路径不存在，它就会崩溃。
fi, _ := os.Stat(path)
if fi.IsDir() {
    fmt.Printf("%s is a directory\n", path)
}
```

Unused imports and variables

未使用的导入和变量

It is an error to import a package or to declare a variable without using it. Unused imports bloat the program and slow compilation, while a variable that is initialized but not used is at least a wasted computation and perhaps indicative of a larger bug. When a program is under active development, however, unused imports and variables often arise and it can be annoying to delete them just to have the compilation proceed, only to have them be needed again later. The blank identifier provides a workaround.

若导入某个包或声明某个变量而不使用它就会产生错误。未使用的包会让程序膨胀并拖慢编译速度，而已初始化但未使用的变量不仅会浪费计算能力，还有可能暗藏着更大的 Bug。 然而在程序开发过程中，经常会产生未使用的导入和变量。虽然以后会用到它们，但为了完成编译又不得不删除它们才行，这很让人烦恼。空白标识符就能提供一个工作空间。

This half-written program has two unused imports (fmt and io) and an unused variable (fd), so it will not compile, but it would be nice to see if the code so far is correct.

这个写了一半的程序有两个未使用的导入（fmt 和 io）以及一个未使用的变量（fd），因此它不能编译， 但若到目前为止代码还是正确的，我们还是很乐意看到它们的。

```go
package main

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

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

To silence complaints about the unused imports, use a blank identifier to refer to a symbol from the imported package. Similarly, assigning the unused variable fd to the blank identifier will silence the unused variable error. This version of the program does compile.

要让编译器停止关于未使用导入的抱怨，需要空白标识符来引用已导入包中的符号。 同样，将未使用的变量 fd 赋予空白标识符也能关闭未使用变量错误。 该程序的以下版本可以编译。

```go
package main

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

var _ = fmt.Printf // For debugging; delete when done. // 用于调试，结束时删除。
var _ io.Reader    // For debugging; delete when done. // 用于调试，结束时删除。

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

By convention, the global declarations to silence import errors should come right after the imports and be commented, both to make them easy to find and as a reminder to clean things up later.

按照惯例，我们应在导入并加以注释后，再使全局声明导入错误静默，这样可以让它们更易找到， 并作为以后清理它的提醒。

Import for side effect

为副作用而导入

An unused import like fmt or io in the previous example should eventually be used or removed: blank assignments identify code as a work in progress. But sometimes it is useful to import a package only for its side effects, without any explicit use. For example, during its init function, the net/http/pprof package registers HTTP handlers that provide debugging information. It has an exported API, but most clients need only the handler registration and access the data through a web page. To import the package only for its side effects, rename the package to the blank identifier:

像前例中 fmt 或 io 这种未使用的导入总应在最后被使用或移除： 空白赋值会将代码标识为工作正在进行中。但有时导入某个包只是为了其副作用， 而没有任何明确的使用。例如，在 net/http/pprof 包的 init 函数中记录了 HTTP 处理程序的调试信息。它有个可导出的 API， 但大部分客户端只需要该处理程序的记录和通过 Web 叶访问数据。只为了其副作用来哦导入该包， 只需将包重命名为空白标识符：

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

This form of import makes clear that the package is being imported for its side effects, because there is no other possible use of the package: in this file, it doesn't have a name. (If it did, and we didn't use that name, the compiler would reject the program.)

这种导入格式能明确表示该包是为其副作用而导入的，因为没有其它使用该包的可能：在此文件中，它没有名字。（若它有名字而我们没有使用，编译器就会拒绝该程序。）

Interface checks

接口检查

As we saw in the discussion of interfaces above, a type need not declare explicitly that it implements an interface. Instead, a type implements the interface just by implementing the interface's methods. In practice, most interface conversions are static and therefore checked at compile time. For example, passing an `*os.File` to a function expecting an io.Reader will not compile unless `*os.File` implements the io.Reader interface.

就像我们在前面 接口 中讨论的那样，一个类型无需显式地声明它实现了某个接口。取而代之，该类型只要实现了某个接口的方法，其实就实现了该接口。在实践中，大部分接口转换都是静态的，因此会在编译时检测。例如，将一个 `*os.File` 传入一个预期的 io.Reader 函数将不会被编译，除非 `*os.File` 实现了 io.Reader 接口。

Some interface checks do happen at run-time, though. One instance is in the encoding/json package, which defines a Marshaler interface. When the JSON encoder receives a value that implements that interface, the encoder invokes the value's marshaling method to convert it to JSON instead of doing the standard conversion. The encoder checks this property at run time with a type assertion like:

尽管有些接口检查会在运行时进行。encoding/json 包中就有个实例它定义了一个 Marshaler 接口。当 JSON 编码器接收到一个实现了该接口的值，那么该编码器就会调用该值的编组方法，将其转换为 JSON，而非进行标准的类型转换。编码器在运行时通过 类型断言 检查其属性，就像这样：

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

If it's necessary only to ask whether a type implements an interface, without actually using the interface itself, perhaps as part of an error check, use the blank identifier to ignore the type-asserted value:

若只需要判断某个类型是否是实现了某个接口，而不需要实际使用接口本身（可能是错误检查部分），就使用空白标识符来忽略类型断言的值：

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

One place this situation arises is when it is necessary to guarantee within the package implementing the type that it actually satisfies the interface. If a type—for example, json.RawMessage—needs a custom JSON representation, it should implement json.Marshaler, but there are no static conversions that would cause the compiler to verify this automatically. If the type inadvertently fails to satisfy the interface, the JSON encoder will still work, but will not use the custom implementation. To guarantee that the implementation is correct, a global declaration using the blank identifier can be used in the package:

当需要确保某个包中实现的类型一定满足该接口时，就会遇到这种情况。若某个类型（例如 json.RawMessage）需要一种定制的 JSON 表现时，它应当实现 json.Marshaler，不过现在没有静态转换可以让编译器去自动验证它。若该类型通过忽略转换失败来满足该接口，那么

JSON 编码器仍可工作，但它却不会使用定制的实现。为确保其实现正确，可在该包中用空白标识符声明一个全局变量：

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

In this declaration, the assignment involving a conversion of a `*RawMessage` to a Marshaler requires that `*RawMessage` implements Marshaler, and that property will be checked at compile time. Should the json.Marshaler interface change, this package will no longer compile and we will be on notice that it needs to be updated.

在此声明中，我们调用了一个 `*RawMessage` 转换并将其赋予了 Marshaler，以此来要求 `*RawMessage` 实现 Marshaler，这时其属性就会在编译时被检测。 若 json.Marshaler 接口被更改，此包将无法通过编译， 而我们则会注意到它需要更新。

The appearance of the blank identifier in this construct indicates that the declaration exists only for the type checking, not to create a variable. Don't do this for every type that satisfies an interface, though. By convention, such declarations are only used when there are no static conversions already present in the code, which is a rare event.

在这种结构中出现空白标识符，即表示该声明的存在只是为了类型检查。 不过请不要为满足接口就将它用于任何类型。作为约定， 仅当代码中不存在静态类型转换时才能这种声明，毕竟这是种罕见的情况。

Embedding

内嵌

Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to "borrow" pieces of an implementation by embedding types within a struct or interface.

Go 并不提供典型的，类型驱动的子类化概念，但通过将类型 <内嵌到结构体或接口中， 它就能 "借鉴" 部分实现。

Interface embedding is very simple. We've mentioned the io.Reader and io.Writer interfaces before; here are their definitions.

接口内嵌非常简单。我们之前提到过 io.Reader 和 io.Writer 接口，这里是它们的定义。

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

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

The io package also exports several other interfaces that specify objects that can implement several such methods. For instance, there is io.ReadWriter, an interface containing both Read and Write. We could specify io.ReadWriter by listing the two methods explicitly, but it's easier and more evocative to embed the two interfaces to form the new one, like this:

io 包也导出了一些其它接口，以此来阐明对象所需实现的方法。 例如 io.ReadWriter 就是个包含 Read 和 Write 的接口。我们可以通过显示地列出这两个方法来指明 io.ReadWriter， 但通过将这两个接口内嵌到新的接口中显然更容易且更具启发性，就像这样：

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

```
// ReadWriter 接口结合了 Reader 和 Writer 接口。
type ReadWriter interface {
    Reader
    Writer
}
```

This says just what it looks like: A ReadWriter can do what a Reader does and what a Writer does; it is a union of the embedded interfaces (which must be disjoint sets of methods). Only interfaces can be embedded within interfaces.

正如它看起来那样：ReadWriter 能够做任何 Reader 和 Writer 可以做到的事情，它是内嵌接口的联合体 （它们必须是不相交的方法集）。只有接口能被嵌入到接口中。

The same basic idea applies to structs, but with more far-reaching implications. The bufio package has two struct types, bufio.Reader and bufio.Writer, each of which of course implements the analogous interfaces from package io. And bufio also implements a buffered reader/writer, which it does by combining a reader and a writer into one struct using embedding: it lists the types within the struct but does not give them field names.

同样的基本想法可以应用在结构体中，但其意义更加深远。bufio 包中有 bufio.Reader 和 bufio.Writer 这两个结构体类型， 它们每一个都实现了与 io 包中相同意义的接口。此外， bufio 还通过结合 reader/writer 并将其内嵌到结构体中，实现了带缓冲的 reader/writer ：它列出了结构体中的类型，但并未给予它们字段名。

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

```
// ReadWriter 存储了指向 Reader 和 Writer 的指针。
// 它实现了 io.ReadWriter。
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

The embedded elements are pointers to structs and of course must be initialized to point to valid structs before they can be used. The ReadWriter struct could be written as

内嵌的元素为指向结构体的指针，当然它们在使用前必须被初始化为指向有效结构体的指针。 ReadWriter 结构体和通过如下方式定义：

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

but then to promote the methods of the fields and to satisfy the io interfaces, we would also need to provide forwarding methods, like this:

但为了提升该字段的方法并满足 io 接口，我们同样需要提供转发的方法，就像这样：

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

By embedding the structs directly, we avoid this bookkeeping. The methods of embedded types come along for free, which means that bufio.ReadWriter not only has the methods of bufio.Reader and bufio.Writer, it also satisfies all three interfaces: io.Reader, io.Writer, and io.ReadWriter.

而通过直接内嵌结构体，我们就能避免如此繁琐。 内嵌类型的方法可以直接引用，这意味着 bufio.ReadWriter 不仅包括 bufio.Reader 和 bufio.Writer 的方法，它还同时满足下列三个接口： io.Reader、io.Writer 以及 io.ReadWriter。

There's an important way in which embedding differs from subclassing. When we embed a type, the methods of that type become methods of the outer type, but when they are invoked the receiver of the method is the inner type, not the outer one. In our example, when the Read method of a bufio.ReadWriter is invoked, it has exactly the same effect as the forwarding method written out above; the receiver is the reader field of the ReadWriter, not the ReadWriter itself.

还有种区分内嵌与子类的重要手段。当内嵌一个类型时，该类型的方法会成为外部类型的方法， 但当它们被调用时，该方法的接收者是内部类型，而非外部的。在我们的例子中，当 bufio.ReadWriter 的 Read 方法被调用时， 它与之前写的转发方法具有同样的效果；接收者 是 ReadWriter 的 reader 字段，而非 ReadWriter 本身。

Embedding can also be a simple convenience. This example shows an embedded field alongside a regular, named field.

内嵌同样可以提供便利。这个例子展示了一个内嵌字段和一个常规的命名字段。

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

The Job type now has the Log, Logf and other methods of `*log.Logger` . We could have given the Logger a field name, of course, but it's not necessary to do so. And now, once initialized, we can log to the Job:

Job 类型现在有了 Log、Logf 和 `*log.Logger` 的其它方法。我们当然可以为 Logger 提供一个字段名，但完全不必这么做。现在，一旦初始化后，我们就能记录 Job 了：

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

The Logger is a regular field of the Job struct, so we can initialize it in the usual way inside the constructor for Job, like this,

Logger 是 Job 结构体的常规字段，因此我们可在 Job 的构造函数中，通过一般的方式来初始化它，就像这样：

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

or with a composite literal,

或通过复合字面：

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

If we need to refer to an embedded field directly, the type name of the field, ignoring the package qualifier, serves as a field name, as it did in the Read method of our ReaderWriter struct. Here, if we needed to access the `*log.Logger` of a Job variable job, we would write job.Logger, which would be useful if we wanted to refine the methods of Logger.

若我们需要直接引用内嵌字段，可以忽略包限定名，直接将该字段的类型名作为字段名，就像我们在 ReaderWriter 结构体的 Read 方法中做的那样。若我们需要访问 Job 类型的变量 job 的 `*log.Logger` ，可以直接写作 job.Logger。若我们想精炼 Logger 的方法时，这会非常有用。

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

Embedding types introduces the problem of name conflicts but the rules to resolve them are simple. First, a field or method X hides any other item X in a more deeply nested part of the type. If log.Logger contained a field or method called Command, the Command field of Job would dominate it.

内嵌类型会引入命名冲突的问题，但解决规则却很简单。首先，字段或方法 X 会隐藏该类型中更深层嵌套的其它项 X。若 log.Logger 包含一个名为 Command 的字段或方法，Job 的 Command 字段会覆盖它。

Second, if the same name appears at the same nesting level, it is usually an error; it would be erroneous to embed log.Logger if the Job struct contained another field or method called Logger. However, if the duplicate name is never mentioned in the program outside the type definition, it is OK. This qualification provides some protection against changes made to types embedded from outside; there is no problem if a field is added that conflicts with another field in another subtype if neither field is ever used.

其次，若相同的嵌套层级上出现同名冲突，通常会产生一个错误。若 Job 结构体中包含名为 Logger 的字段或方法，再将 log.Logger 内嵌到其中的话就会产生错误。然而，若重名永远不会在该类型定义之外的程序中使用，那就不会出错。 这种限定能够在外部嵌套类型发生修改时提供某种保护。 因此，就算添加的字段与另一个子类型中的字段相冲突，只要这两个相同的字段永远不会被使用就没问题。

Concurrency

并发

Share by communicating

通过通信共享内存

Concurrent programming is a large topic and there is space only for some Go-specific highlights here.

并发编程是个很大的论题。但限于篇幅，这里仅讨论一些 Go 特有的东西。

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

在并发编程中，为实现对共享变量的正确访问需要精确的控制，这在多数环境下都很困难。Go 语言另辟蹊径，它将共享的值通过信道传递，实际上，多个独立执行的线程从不会主动共享。 在任意给定的时间点，只有一个 Go 程能够访问该值。数据竞争从设计上就被杜绝了。 为了提倡这种思考方式，我们将它简化为一句口号：

> Do not communicate by sharing memory; instead, share memory by communicating.
>
> 不要通过共享内存来通信，而应通过通信来共享内存。

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

这种方法意义深远。例如，引用计数通过为整数变量添加互斥锁来很好地实现。 但作为一种高级方法，通过信道来控制访问能够让你写出更简洁，正确的程序。

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit

this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

我们可以从典型的单线程运行在单 CPU 之上的情形来审视这种模型。它无需提供同步原语。现在考虑另一种情况，它也无需同步。现在让它们俩进行通信。若将通信过程看做同步着，那就完全不需要其它同步了。例如，Unix 管道就与这种模型完美契合。 尽管 Go 的并发处理方式来源于 Hoare 的通信顺序处理（CSP）， 它依然可以看做是类型安全的 Unix 管道的实现。

Goroutines

Go 程

They're called goroutines because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

我们称之为 **Go** 程 是因为现有的术语—线程、协程、进程等等—无法准确传达它的含义。 Go 程具有简单的模型：它是与其它 Go 程并发运行在同一地址空间的函数。它是轻量级的， 所有小号几乎就只有栈空间的分配。而且栈最开始是非常小的，所以它们很廉价， 仅在需要时才会随着堆空间的分配（和释放）而变化。

Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management.

Go 程在多线程操作系统上可实现多路复用，因此若一个线程阻塞，比如说等待 I/O，那么其它的线程就会运行。Go 程的设计隐藏了线程创建和管理的诸多复杂性。

Prefix a function or method call with the go keyword to run the call in a new goroutine. When the call completes, the goroutine exits, silently. (The effect is similar to the Unix shell's & notation for running a command in the background.)

在函数或方法前添加 go 关键字能够在新的 Go 程中调用它。当调用完成后， 该 Go 程也会安静地退出。（效果有点像 Unix Shell 中的 & 符号，它能让命令在后台运行。）

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

```
go list.Sort()  // 并发运行 list.Sort，无需等它结束。
```

A function literal can be handy in a goroutine invocation.

函数字面在 Go 程调用中非常有用。

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

```go
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // 注意括号 - 必须调用该函数。
}
```

In Go, function literals are closures: the implementation makes sure the variables referred to by the function survive as long as they are active.

在 Go 中，函数字面都是闭包：其实现在保证了函数内引用变量的生命周期与函数的活动时间相同。

These examples aren't too practical because the functions have no way of signaling completion. For that, we need channels.

这些函数没什么实用性，因为它们没有实现完成时的信号处理。因此，我们需要信道。

Channels

信道

Like maps, channels are allocated with make, and the resulting value acts as a reference to an underlying data structure. If an optional integer parameter is provided, it sets the buffer size for the channel. The default is zero, for an unbuffered or synchronous channel.

信道与映射一样，也需要通过 make 来分配内存。其结果值充当了对底层数据结构的引用。若提供了一个可选的整数形参，它就会为该信道设置缓冲区大小。默认值是零，表示不带缓冲的或同步的信道。

```go
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
```

```go
ci := make(chan int)            // 整数类型的无缓冲信道
cj := make(chan int, 0)         // 整数类型的无缓冲信道
cs := make(chan *os.File, 100)  // 指向文件指针的带缓冲信道
```

Unbuffered channels combine communication—the exchange of a value—with synchronization—guaranteeing that two calculations (goroutines) are in a known state.

无缓冲信道在通信时会同步交换数据，它能确保（两个 Go 程的）计算处于确定状态。

There are lots of nice idioms using channels. Here's one to get us started. In the previous section we launched a sort in the background. A channel can allow the launching goroutine to wait for the sort to complete.

信道有很多惯用法，我们从这里开始了解。在上一节中，我们在后台启动了排序操作。 信道使得启动的 Go 程等待排序完成。

```go
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.
```

```go
c := make(chan int)  // 分配一个信道
// 在 Go 程中启动排序。当它完成后，在信道上发送信号。
go func() {
    list.Sort()
    c <- 1  // 发送信号，什么值无所谓。
}()
doSomethingForAWhile()
<-c  // 等待排序结束，丢弃发来的值。
```

Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

接收者在收到数据前会一直阻塞。若信道是不带缓冲的，那么在接收者收到值前， 发送者会一直阻塞；若信道是带缓冲的，则发送者仅在值被复制到缓冲区前阻塞；若缓冲区已满，发送者会一直等待直到某个接收者取出一个值为止。

A buffered channel can be used like a semaphore, for instance to limit throughput. In this example, incoming requests are passed to handle, which sends a value into the channel, processes the request, and then receives a value from the channel to ready the "semaphore" for the next consumer. The capacity of the channel buffer limits the number of simultaneous calls to process.

带缓冲的信道可被用作信号量，例如限制吞吐量。在此例中，进入的请求会被传递给 handle，它从信道中接收值，处理请求后将值发回该信道中，以便让该"信号量"准备迎接下一次请求。信道缓冲区的容量决定了同时调用 process 的数量上限，因此我们在初始化时首先要填充至它的容量上限。

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

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

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

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

func handle(r *Request) {
    sem <- 1 // 等待活动队列清空。
    process(r)  // 可能需要很长时间。
    <-sem     // 完成；使下一个请求可以运行。
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // 无需等待 handle 结束。
    }
}
```

Once MaxOutstanding handlers are executing process, any more will block trying to send into the filled channel buffer, until one of the existing handlers finishes and receives from the buffer.

由于数据同步发生在信道的接收端（也就是说发送发生在 > 接受之前，参见 Go 内存模型），因此信号必须在信道的接收端获取，而非发送端。

This design has a problem, though: Serve creates a new goroutine for every incoming request, even though only MaxOutstanding of them can run at any moment. As a result, the program can consume unlimited resources if the requests come in too fast. We can address that deficiency by changing Serve to gate the creation of the goroutines. Here's an obvious solution, but beware it has a bug we'll fix subsequently:

然而，它却有个设计问题：尽管只有 MaxOutstanding 个 Go 程能同时运行，但 Serve 还是为每个进入的请求都创建了新的 Go 程。其结果就是，若请求来得很快，该程序就会无限地消耗资源。为了弥补这种不足，我们可以通过修改 Serve 来限制创建 Go 程，这是个明显的解决方案，但要当心我们修复后出现的 Bug。

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

```go
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func() {
            process(req) // 这儿有 Bug，解释见下。
            <-sem
        }()
    }
}
```

The bug is that in a Go for loop, the loop variable is reused for each iteration, so the req variable is shared across all goroutines. That's not what we want. We need to make sure that req is unique for each goroutine. Here's one way to do that, passing the value of req as an argument to the closure in the goroutine:

Bug 出现在 Go 的 for 循环中，该循环变量在每次迭代时会被重用，因此 req 变量会在所有的 Go 程间共享，这不是我们想要的。我们需要确保 req 对于每个 Go 程来说都是唯一的。有一种方法能够做到，就是将 req 的值作为实参传入到该 Go 程的闭包中：

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

Compare this version with the previous to see the difference in how the closure is declared and run. Another solution is just to create a new variable with the same name, as in this example:

比较前后两个版本，观察该闭包声明和运行中的差别。 另一种解决方案就是以相同的名字创建新的变量，如例中所示：

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

```go
func Serve(queue chan *Request) {
    for req := range queue {
        req := req // 为该 Go 程创建 req 的新实例。
        sem <- 1
        go func() {
            process(req)
            <-sem
        }()
    }
}
```

It may seem odd to write

它的写法看起来有点奇怪

```go
req := req
```

but it's a legal and idiomatic in Go to do this. You get a fresh version of the variable with the same name, deliberately shadowing the loop variable locally but unique to each goroutine.

但在 Go 中这样做是合法且惯用的。你用相同的名字获得了该变量的一个新的版本，以此来局部地刻意屏蔽循环变量，使它对每个 Go 程保持唯一。

Going back to the general problem of writing the server, another approach that manages resources well is to start a fixed number of handle goroutines all reading from the request channel. The number of goroutines limits the number of simultaneous calls to process. This Serve function also accepts a channel on which it will be told to exit; after launching the goroutines it blocks receiving from that channel.

回到编写服务器的一般问题上来。另一种管理资源的好方法就是启动固定数量的 handle Go 程，一起从请求信道中读取数据。Go 程的数量限制了同时调用 process 的数量。Serve 同样会接收一个通知退出的信道，在启动所有 Go 程后，它将阻塞并暂停从信道中接收消息。

```go
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.
}
```

```go
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  // 等待通知退出。
}
```

Channels of channels

信道中的信道

One of the most important properties of Go is that a channel is a first-class value that can be allocated and passed around like any other. A common use of this property is to implement safe, parallel demultiplexing.

Go 最重要的特性就是信道是一等值，它可以被分配并像其它值到处传递。 这种特性通常被用来实现安全、并行的多路分解。

In the example in the previous section, handle was an idealized handler for a request but we didn't define the type it was handling. If that type includes a channel on which to reply, each client can provide its own path for the answer. Here's a schematic definition of type Request.

在上一节的例子中，handle 是个非常理想化的请求处理程序， 但我们并未定义它所处理的请求类型。若该类型包含一个可用于回复的信道， 那么每一个客户端都能为其回应提供自己的路径。以下为 Request 类型的大概定义。

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

The client provides a function and its arguments, as well as a channel inside the request object on which to receive the answer.

客户端提供了一个函数及其实参，此外在请求对象中还有个接收应答的信道。

```go
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)
```

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

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

On the server side, the handler function is the only thing that changes.

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

There's clearly a lot more to do to make it realistic, but this code is a framework for a rate-limited, parallel, non-blocking RPC system, and there's not a mutex in sight.

要使其实际可用还有很多工作要做，这些代码仅能实现一个速率有限、并行、非阻塞 RPC 系统的 框架，而且它并不包含互斥锁。

Parallelization

并行化

Another application of these ideas is to parallelize a calculation across multiple CPU cores. If the calculation can be broken into separate pieces that can execute independently, it can be parallelized, with a channel to signal when each piece completes.

这些设计的另一个应用是在多 CPU 核心上实现并行计算。如果计算过程能够被分为几块 可独立执行的过程，它就可以在每块计算结束时向信道发送信号，从而实现并行处理。

Let's say we have an expensive operation to perform on a vector of items, and that the value of the operation on each item is independent, as in this idealized example.

让我们看看这个理想化的例子。我们在对一系列向量项进行极耗资源的操作，而每个项的值 计算是完全独立的。

```go
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
}
```

```go
type Vector []float64

// 将此操应用至 v[i], v[i+1] ... 直到 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    // 发信号表示这一块计算完成。
}
```

We launch the pieces independently in a loop, one per CPU. They can complete in any order but it doesn't matter; we just count the completion signals by draining the channel after launching all the goroutines.

我们在循环中启动了独立的处理块，每个 CPU 将执行一个处理。 它们有可能以乱序的形式完成并结束，但这没有关系；我们只需在所有 Go 程开始后接收，并统计信道中的完成信号即可。

```go
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.
}
```

```
const NCPU = 4  // CPU 核心数

func (v Vector) DoAll(u Vector) {
    c := make(chan int, NCPU)  // 缓冲区是可选的，但明显用上更好
    for i := 0; i < NCPU; i++ {
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // 排空信道。
    for i := 0; i < NCPU; i++ {
        <-c    // 等待任务完成
    }
    // 一切完成。
}
```

The current implementation of the Go runtime will not parallelize this code by default. It dedicates only a single core to user-level processing. An arbitrary number of goroutines can be blocked in system calls, but by default only one can be executing user-level code at any time. It should be smarter and one day it will be smarter, but until it is if you want CPU parallelism you must tell the run-time how many goroutines you want executing code simultaneously. There are two related ways to do this. Either run your job with environment variable GOMAXPROCS set to the number of cores to use or import the runtime package and call runtime.GOMAXPROCS(NCPU). A helpful value might be runtime.NumCPU(), which reports the number of logical CPUs on the local machine. Again, this requirement is expected to be retired as the scheduling and run-time improve.

目前 Go 运行时的实现默认并不会并行执行代码，它只为用户层代码提供单一的处理核心。任意数量的 Go 程都可能在系统调用中被阻塞，而在任意时刻默认只有一个会执行用户层代码。它应当变得更智能，而且它将来肯定会变得更智能。但现在，若你希望 CPU 并行执行，就必须告诉运行时你希望同时有多少 Go 程能执行代码。有两种途径可意识形态，要么在运行你的工作时将 GOMAXPROCS 环境变量设为你要使用的核心数，要么导入 runtime 包并调用 runtime.GOMAXPROCS(NCPU)。runtime.NumCPU() 的值可能很有用，它会返回当前机器的逻辑 CPU 核心数。当然，随着调度算法和运行时的改进，将来会不再需要这种方法。

Be sure not to confuse the ideas of concurrency—structuring a program as independently executing components—and parallelism—executing calculations in parallel for efficiency on multiple CPUs. Although the concurrency features of Go can make some problems easy to structure as parallel computations, Go is a concurrent language, not a parallel one, and not all parallelization problems fit Go's model. For a discussion of the distinction, see the talk cited in this blog post.

注意不要混淆并发和并行的概念：并发是用可独立执行的组件构造程序的方法，而并行则是为了效率在多 CPU 上平行地进行计算。尽管 Go 的并发特性能够让某些问题更易构造成并行计算，但 Go 仍然是种并发而非并行的语言，且 Go 的模型并不适合所有的并行问题。关于

其中区别的讨论，见 此博文。

A leaky buffer

可能泄露的缓冲区

The tools of concurrent programming can even make non-concurrent ideas easier to express. Here's an example abstracted from an RPC package. The client goroutine loops receiving data from some source, perhaps a network. To avoid allocating and freeing buffers, it keeps a free list, and uses a buffered channel to represent it. If the channel is empty, a new buffer gets allocated. Once the message buffer is ready, it's sent to the server on serverChan.

并发编程的工具甚至能很容易地表达非并发的思想。这里有个提取自 RPC 包的例子。 客户端 Go 程从某些来源，可能是网络中循环接收数据。为避免分配和释放缓冲区， 它保存了一个空闲链表，使用一个带缓冲信道表示。若信道为空，就会分配新的缓冲区。 一旦消息缓冲区就绪，它将通过 serverChan 被发送到服务器。 serverChan.

```go
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.
    }
}
```

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

func client() {
    for {
        var b *Buffer
        // 若缓冲区可用就用它，不可用就分配个新的。
        select {
        case b = <-freeList:
            // 获取一个，不做别的。
        default:
            // 非空闲，因此分配一个新的。
            b = new(Buffer)
        }
        load(b)                 // 从网络中读取下一条消息。
        serverChan <- b     // 发送至服务器。
    }
}
```

The server loop receives each message from the client, processes it, and returns the buffer to the free list.

服务器从客户端循环接收每个消息，处理它们，并将缓冲区返回给空闲列表。

```
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.
        }
    }
}
```

```go
func server() {
    for {
        b := <-serverChan    // 等待工作。
        process(b)
        // 若缓冲区有空间就重用它。
        select {
        case freeList <- b:
            // 将缓冲区放大空闲列表中，不做别的。
        default:
            // 空闲列表已满，保持就好。
        }
    }
}
```

The client attempts to retrieve a buffer from freeList; if none is available, it allocates a fresh one. The server's send to freeList puts b back on the free list unless the list is full, in which case the buffer is dropped on the floor to be reclaimed by the garbage collector. (The default clauses in the select statements execute when no other case is ready, meaning that the selects never block.) This implementation builds a leaky bucket free list in just a few lines, relying on the buffered channel and the garbage collector for bookkeeping.

客户端试图从 freeList 中获取缓冲区；若没有缓冲区可用，它就将分配一个新的。服务器将 b 放回空闲列表 freeList 中直到列表已满，此时缓冲区将被丢弃，并被垃圾回收器回收。（select 语句中的 default 子句在没有条件符合时执行，这也就意味着 selects 永远不会被阻塞。）依靠带缓冲的信道和垃圾回收器的记录，我们仅用短短几行代码就构建了一个可能导致缓冲区槽位泄露的空闲列表。

Errors

错误

Library routines must often return some sort of error indication to the caller. As mentioned earlier, Go's multivalue return makes it easy to return a detailed error description alongside the normal return value. It is good style to use this feature to provide detailed error information. For example, as we'll see, os.Open doesn't just return a nil pointer on failure, it also returns an error value that describes what went wrong.

By convention, errors have type error, a simple built-in interface.

库例程通常需要向调用者返回某种类型的错误提示。之前提到过，Go 语言的多值返回特性，使得它在返回常规的值时，还能轻松地返回详细的错误描述。按照约定，错误的类型通常为 error，这是一个内建的简单接口。

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

A library writer is free to implement this interface with a richer model under the covers, making it possible not only to see the error but also to provide some context. As mentioned, alongside the usual `*os.File` return value, os.Open also returns an error value. If the file is opened successfully, the error will be nil, but when there is a problem, it will hold an os.PathError:

库的编写者通过更丰富的底层模型可以轻松实现这个接口，这样不仅能看见错误，还能提供一些上下文。例如，os.Open 可返回一个 os.PathError。

```go
// 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 记录一个错误以及产生该错误的路径和操作。
type PathError struct {
    Op string    // "open"、"unlink" 等等。
    Path string  // 相关联的文件。
    Err error    // 由系统调用返回。
}

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

PathError's Error generates a string like this:

PathError 的 Error 会生成如下错误信息：

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

Such an error, which includes the problematic file name, the operation, and the operating system error it triggered, is useful even if printed far from the call that caused it; it is much more informative than the plain "no such file or directory".

这种错误包含了出错的文件名、操作和触发的操作系统错误，即便在产生该错误的调用 和输出的错误信息相距甚远时，它也会非常有用，这比苍白的 "不存在该文件或目录" 更具说明性。

When feasible, error strings should identify their origin, such as by having a prefix naming the operation or package that generated the error. For example, in package image, the string representation for a decoding error due to an unknown format is "image: unknown format".

错误字符串应尽可能地指明它们的来源，例如产生该错误的包名前缀。例如在 image 包中，由于未知格式导致解码错误的字符串为 "image: unknown format"。

Callers that care about the precise error details can use a type switch or a type assertion to look for specific errors and extract details. For PathErrors this might include examining the internal Err field for recoverable failures.

若调用者关心错误的完整细节，可使用类型选择或者类型断言来查看特定错误，并抽取其细节。 对于 PathErrors，它应该还包含检查内部的 Err 字段以进行可能的错误恢复。

```go
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
}
```

```go
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
}
```

The second if statement here is another type assertion. If it fails, ok will be false, and e will be nil. If it succeeds, ok will be true, which means the error was of type `*os.PathError`, and then so is e, which we can examine for more information about the error.

这里的第二条 if 是另一种 类型断言。若它失败，ok 将为 false，而 e 则为 nil. 若它成功，ok 将为 true，这意味着该错误属于 `*os.PathError` 类型，而 e 能够检测关于该错误的更多信息。

Panic

Panic

The usual way to report an error to a caller is to return an error as an extra return value. The canonical Read method is a well-known instance; it returns a byte count and an error. But what if the error is unrecoverable? Sometimes the program simply cannot continue.

向调用者报告错误的一般方式就是将 error 作为额外的值返回。 标准的 Read 方法就是个众所周知的实例，它返回一个字节计数和一个 error。但如果错误时不可恢复的呢？有时程序就是不能继续运行。

For this purpose, there is a built-in function panic that in effect creates a run-time error that will stop the program (but see the next section). The function takes a single argument of arbitrary type—often a string—to be printed as the program dies. It's also a way to indicate that something impossible has happened, such as exiting an infinite loop.

为此，我们提供了内建的 panic 函数，它会产生一个运行时错误并终止程序（但请继续看下一节）。该函数接受一个任意类型的实参（一般为字符串），并在程序终止时打印。 它还能表明发生了意料之外的事情，比如从无限循环中退出了。

```go
// 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))
}
```

```go
// 用牛顿法计算立方根的一个玩具实现。
func CubeRoot(x float64) float64 {
    z := x/3   // 任意初始值
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // 一百万次迭代并未收敛，事情出错了。
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

This is only an example but real library functions should avoid panic. If the problem can be masked or worked around, it's always better to let things continue to run rather than taking down the whole program. One possible counterexample is during initialization: if the library truly cannot set itself up, it might be reasonable to panic, so to speak.

这仅仅是个示例，实际的库函数应避免 panic。若问题可以被屏蔽或解决， 最好就是让程序继续运行而不是终止整个程序。一个可能的反例就是初始化：若某个库真的不能让自己工作，且有足够理由产生 Panic，那就由它去吧。

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

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

Recover

恢复

When panic is called, including implicitly for run-time errors such as indexing a slice out of bounds or failing a type assertion, it immediately stops execution of the current function and begins unwinding the stack of the goroutine, running any deferred functions along the way. If that unwinding reaches the top of the goroutine's stack, the program dies. However, it is possible to use the built-in function recover to regain control of the goroutine and resume normal execution.

当 panic 被调用后（包括不明确的运行时错误，例如切片检索越界或类型断言失败），程序将立刻终止当前函数的执行，并开始回溯 Go 程的栈，运行任何被推迟的函数。 若回溯到达 Go 程栈的顶端，程序就会终止。不过我们可以用内建的 recover 函数来重新或来取回 Go 程的控制权限并使其恢复正常执行。

A call to recover stops the unwinding and returns the argument passed to panic. Because the only code that runs while unwinding is inside deferred functions, recover is only useful inside deferred functions.

调用 recover 将停止回溯过程，并返回传入 panic 的实参。 由于在回溯时只有被推迟函数中的代码在运行，因此 recover 只能在被推迟的函数中才有效。

One application of recover is to shut down a failing goroutine inside a server without killing the other executing goroutines.

recover 的一个应用就是在服务器中终止失败的 Go 程而无需杀死其它正在执行的 Go 程。

```go
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)
}
```

In this example, if do(work) panics, the result will be logged and the goroutine will exit cleanly without disturbing the others. There's no need to do anything else in the deferred closure; calling recover handles the condition completely.

在此例中，若 do(work) 触发了 Panic，其结果就会被记录，而该 Go 程会被干净利落地结束，不会干扰到其它 Go 程。我们无需在推迟的闭包中做任何事情，recover 会处理好这一切。

Because recover always returns nil unless called directly from a deferred function, deferred code can call library routines that themselves use panic and recover without failing. As an example, the deferred function in safelyDo might call a logging function before calling recover, and that logging code would run unaffected by the panicking state.

由于直接从被推迟函数中调用 recover 时不会返回 nil，因此被推迟的代码能够调用本身使用了 panic 和 recover 的库函数而不会失败。例如在 safelyDo 中，被推迟的函数可能在调用 recover 前先调用记录函数，而该记录函数应当不受 Panic 状态的代码的影响。

With our recovery pattern in place, the do function (and anything it calls) can get out of any bad situation cleanly by calling panic. We can use that idea to simplify error handling in complex software. Let's look at an idealized version of a regexp package, which reports parsing errors by calling panic with a local error type. Here's the definition of Error, an error method, and the Compile function.

通过恰当地使用恢复模式，do 函数（及其调用的任何代码）可通过调用 panic 来避免更坏的结果。我们可以利用这种思想来简化复杂软件中的错误处理。让我们看看 regexp 包的理想化版本，它会以局部的错误类型调用 panic 来报告解析错误。以下是一个 error 类型的 Error 方法和一个 Compile 函数的定义：

```go
// 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
}
```

```go
// Error 是解析错误的类型，它满足 error 接口。
type Error string
func (e Error) Error() string {
    return string(e)
}

// error 是 *Regexp 的方法，它通过用一个 Error 触发 Panic 来报告解析错误。
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile 返回该正则表达式解析后的表示。
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    // 清理返回值。
            err = e.(Error) // 若它不是解析错误，将重新触发 Panic。
        }
    }()
    return regexp.doParse(str), nil
}
```

If doParse panics, the recovery block will set the return value to nil—deferred functions can modify named return values. It will then check, in the assignment to err, that the problem was a parse error by asserting that it has the local type Error. If it does not, the type assertion will fail, causing a run-time error that continues the stack unwinding as though nothing had interrupted it. This check means that if something unexpected happens, such as an index out of bounds, the code will fail even though we are using panic and recover to handle parse errors.

若 doParse 触发了 Panic，恢复块会将返回值设为 nil —被推迟的函数能够修改已命名的返回值。在 err 的赋值过程中，我们将通过断言它是否拥有局部类型 Error 来检查它。若它没有，类型断言将会失败，此时会产生运行时错误，并继续栈的回溯，仿佛一切从未中断过一样。该检查意味着若发生了一些像索引越界之类的意外，那么即便我们使用了 panic 和 recover 来处理解析错误，代码仍然会失败。

With error handling in place, the error method (because it's a method bound to a type, it's fine, even natural, for it to have the same name as the builtin error type) makes it easy to report parse errors without worrying about unwinding the parse stack by hand:

通过适当的错误处理，error 方法（由于它是个绑定到具体类型的方法，因此即便它与内建的 error 类型名字相同也没有关系）能让报告解析错误变得更容易，而无需手动处理回溯的解析栈：

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

Useful though this pattern is, it should be used only within a package. Parse turns its internal panic calls into error values; it does not expose panics to its client. That is a good rule to follow.

尽管这种模式很有用，但它应当仅在包内使用。Parse 会将其内部的 panic 调用转为 error 值，它并不会向调用者暴露出 panic。这是个值得遵守的良好规则。

By the way, this re-panic idiom changes the panic value if an actual error occurs. However, both the original and new failures will be presented in the crash report, so the root cause of the problem will still be visible. Thus this simple re-panic approach is usually sufficient—it's a crash after all—but if you want to display only the original value, you can write a little more code to filter unexpected problems and re-panic with the original error. That's left as an exercise for the reader.

顺便一提，这种重新触发Panic的惯用法会在产生实际错误时改变Panic的值。然而，不管是原始的还是新的错误都会在崩溃报告中显示，因此问题的根源仍然是可见的。这种简单的重新触发Panic的模型已经够用了，毕竟他只是一次崩溃。但若你只想显示原始的值，也可以多

写一点代码来过滤掉不需要的问题，然后用原始值再次触发Panic。 这里就将这个练习留给读者了。

A web server

一个 **Web** 服务器

Let's finish with a complete Go program, a web server. This one is actually a kind of web re-server. Google provides a service at http://chart.apis.google.com that does automatic formatting of data into charts and graphs. It's hard to use interactively, though, because you need to put the data into the URL as a query. The program here provides a nicer interface to one form of data: given a short piece of text, it calls on the chart server to produce a QR code, a matrix of boxes that encode the text. That image can be grabbed with your cell phone's camera and interpreted as, for instance, a URL, saving you typing the URL into the phone's tiny keyboard.

让我们以一个完整的 Go 程序作为结束吧，一个 Web 服务器。该程序其实只是个 Web 服务器的重用。 Google 在 http://chart.apis.google.com 上提供了一个将表单数据自动转换为图表的服务。不过，该服务很难交互， 因为你需要将数据作为查询放到 URL 中。此程序为一种数据格式提供了更好的的接口： 给定一小段文本，它将调用图表服务器来生成二维码（QR码），这是一种编码文本的点格矩阵。 该图像可被你的手机摄像头捕获，并解释为一个字符串，比如 URL， 这样就免去了你在狭小的手机键盘上键入 URL 的麻烦。

Here's the complete program. An explanation follows.

以下为完整的程序，随后有一段解释。

```go
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 .}}
<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={{.}}"
/>
<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>
`
```

The pieces up to main should be easy to follow. The one flag sets a default HTTP port for our server. The template variable templ is where the fun happens. It builds an HTML template that will be executed by the server to display the page; more about that in a

moment.

main 之前的代码应该比较容易理解。我们通过一个标志为服务器设置了默认端口。 模板变量 templ 正式有趣的地方。它构建的 HTML 模版将会被服务器执行并显示在页面中。 稍后我们将详细讨论。

The main function parses the flags and, using the mechanism we talked about above, binds the function QR to the root path for the server. Then http.ListenAndServe is called to start the server; it blocks while the server runs.

main 函数解析了参数标志并使用我们讨论过的机制将 QR 函数绑定到服务器的根路径。然后调用 http.ListenAndServe 启动服务器；它将在服务器运行时处于阻塞状态。

QR just receives the request, which contains form data, and executes the template on the data in the form value named s.

QR 仅接受包含表单数据的请求，并为表单值 s 中的数据执行模板。

The template package html/template is powerful; this program just touches on its capabilities. In essence, it rewrites a piece of HTML text on the fly by substituting elements derived from data items passed to templ.Execute, in this case the form value. Within the template text (templateStr), double-brace-delimited pieces denote template actions. The piece from `{{if .}}` to `{{end}}` executes only if the value of the current data item, called . (dot), is non-empty. That is, when the string is empty, this piece of the template is suppressed.

模板包 html/template 非常强大；该程序只是浅尝辄止。 本质上，它通过在运行时将数据项中提取的元素（在这里是表单值）传给 templ.Execute 执行因而重写了 HTML 文本。 在模板文本（templateStr）中，双大括号界定的文本表示模板的动作。 从 `{{if .}}` 到 `{{end}}` 的代码段仅在当前数据项（这里是点 .）的值非空时才会执行。 也就是说，当字符串为空时，此部分模板段会被忽略。

The two snippets `{{.}}` say to show the data presented to the template—the query string—on the web page. The HTML template package automatically provides appropriate escaping so the text is safe to display.

其中两段 `{{.}}` 表示要将数据显示在模板中 （即将查询字符串显示在 Web 页面上）。 HTML 模板包将自动对文本进行转义， 因此文本的显示是安全的。

The rest of the template string is just the HTML to show when the page loads. If this is too quick an explanation, see the documentation for the template package for a more thorough discussion.

余下的模板字符串只是页面加载时将要显示的 HTML。如果这段解释你无法理解，请参考 文档 获得更多有关模板包的解释。

And there you have it: a useful web server in a few lines of code plus some data-driven HTML text. Go is powerful enough to make a lot happen in a few lines.

你终于如愿以偿了：以几行代码实现的，包含一些数据驱动的HTML文本的Web服务器。 Go 语言强大到能让很多事情以短小精悍的方式解决。