

得到一个指向 `Response` 类型值的指针。之后会监测网络请求是否出错，并在第 127 行安排函数返回时调用 `Close` 方法。

在第 131 行，我们检测了 `Response` 值的 `StatusCode` 字段，确保收到的响应是 200。任何不是 200 的请求都需要作为错误处理。如果响应值不是 200，我们使用 `fmt` 包里的 `Errorf` 函数返回一个自定义的错误。最后 3 行代码很像之前解码 JSON 数据文件的代码。只是这次使用 `xml` 包并调用了同样叫作 `NewDecoder` 的函数。这个函数会返回一个指向 `Decoder` 值的指针。之后调用这个指针的 `Decode` 方法，传入 `rssDocument` 类型的局部变量 `document` 的地址。最后返回这个局部变量的地址和 `Decode` 方法调用返回的错误值。

最后我们来看看实现了 `Matcher` 接口的方法，如代码清单 2-54 所示。

代码清单 2-54 `matchers/rss.go`: 第 69 行到第 112 行

```
69 // Search 在文档中查找特定的搜索项
70 func (m rssMatcher) Search(feed *search.Feed, searchTerm string)
                                ([]*search.Result, error) {
71     var results []*search.Result
72
73     log.Printf("Search Feed Type[%s] Site[%s] For Uri[%s]\n",
                                feed.Type, feed.Name, feed.URI)
74
75     // 获取要搜索的数据
76     document, err := m.retrieve(feed)
77     if err != nil {
78         return nil, err
79     }
80
81     for _, channelItem := range document.Channel.Item {
82         // 检查标题部分是否包含搜索项
83         matched, err := regexp.MatchString(searchTerm, channelItem.Title)
84         if err != nil {
85             return nil, err
86         }
87
88         // 如果找到匹配的项，将其作为结果保存
89         if matched {
90             results = append(results, &search.Result{
91                 Field: "Title",
92                 Content: channelItem.Title,
93             })
94         }
95
96         // 检查描述部分是否包含搜索项
97         matched, err = regexp.MatchString(searchTerm, channelItem.Description)
98         if err != nil {
99             return nil, err
100         }
101
102         // 如果找到匹配的项，将其作为结果保存
```

```

103         if matched {
104             results = append(results, &search.Result{
105                 Field:  "Description",
106                 Content: channelItem.Description,
107             })
108         }
109     }
110
111     return results, nil
112 }

```

我们从第 71 行 `results` 变量的声明开始分析，如代码清单 2-55 所示。这个变量用于保存并返回找到的结果。

代码清单 2-55 matchers/rss.go: 第 71 行

```

71     var results []*search.Result

```

我们使用关键字 `var` 声明了一个值为 `nil` 的切片，切片每一项都是指向 `Result` 类型值的指针。`Result` 类型的声明在之前 `match.go` 代码文件的第 08 行中可以找到。之后在第 76 行，我们使用刚刚看过的 `retrieve` 方法进行网络调用，如代码清单 2-56 所示。

代码清单 2-56 matchers/rss.go: 第 75 行到第 79 行

```

75     // 获取要搜索的数据
76     document, err := m.retrieve(feed)
77     if err != nil {
78         return nil, err
79     }

```

调用 `retrieve` 方法返回了一个指向 `rssDocument` 类型值的指针以及一个错误值。之后，像已经多次看过的代码一样，检查错误值，如果真的是一个错误，直接返回。如果没有错误发生，之后会依次检查得到的 RSS 文档的每一项的标题和描述，如果与搜索项匹配，就将其作为结果保存，如代码清单 2-57 所示。

代码清单 2-57 matchers/rss.go: 第 81 行到第 86 行

```

81     for _, channelItem := range document.Channel.Item {
82         // 检查标题部分是否包含搜索项
83         matched, err := regexp.MatchString(searchTerm, channelItem.Title)
84         if err != nil {
85             return nil, err
86         }

```

既然 `document.Channel.Item` 是一个 `item` 类型值的切片，我们在第 81 行对其使用 `for range` 循环，依次访问其内部的每一项。在第 83 行，我们使用 `regexp` 包里的 `MatchString` 函数，对 `channelItem` 值里的 `Title` 字段进行搜索，查找是否有匹配的搜索项。之后在第 84 行检查错误。如果没有错误，就会在第 89 行到第 94 行检查匹配的结果，如代码清单 2-58 所示。

代码清单 2-58 matchers/rss.go: 第 88 行到第 94 行

```
88         // 如果找到匹配的项，将其作为结果保存
89         if matched {
90             results = append(results, &search.Result{
91                 Field:  "Title",
92                 Content: channelItem.Title,
93             })
94         }
```

如果调用 `MatchString` 方法返回的 `matched` 的值为真，我们使用内置的 `append` 函数，将搜索结果加入到 `results` 切片里。`append` 这个内置函数会根据切片需要，决定是否要增加切片的长度和容量。我们会在第 4 章了解关于内置函数 `append` 的更多知识。这个函数的第一个参数是希望追加到的切片，第二个参数是要追加的值。在这个例子中，追加到切片的值是一个指向 `Result` 类型值的指针。这个值直接使用字面声明的方式，初始化为 `Result` 类型的值。之后使用取地址运算符（&），获得这个新值的地址。最终将这个指针存入了切片。

在检查标题是否匹配后，第 97 行到第 108 行使用同样的逻辑检查 `Description` 字段。最后，在第 111 行，`Search` 方法返回了 `results` 作为函数调用的结果。

2.5 小结

- 每个代码文件都属于一个包，而包名应该与代码文件所在的文件夹同名。
- Go 语言提供了多种声明和初始化变量的方式。如果变量的值没有显式初始化，编译器会将变量初始化为零值。
- 使用指针可以在函数间或者 goroutine 间共享数据。
- 通过启动 goroutine 和使用通道完成并发和同步。
- Go 语言提供了内置函数来支持 Go 语言内部的数据结构。
- 标准库包含很多包，能做很多很有用的事情。
- 使用 Go 接口可以编写通用的代码和框架。

第 3 章 打包和工具链

本章主要内容

- 如何组织 Go 代码
- 使用 Go 语言自带的相关命令
- 使用其他开发者提供的工具
- 与其他开发者合作

我们在第 2 章概览了 Go 语言的语法和语言结构。本章会进一步介绍如何把代码组织成包，以及如何操作这些包。在 Go 语言里，包是个非常重要的概念。其设计理念是使用包来封装不同语义单元的功能。这样做，能够更好地复用代码，并对每个包内的数据的使用有更好的控制。

在进入具体细节之前，假设读者已经熟悉命令行提示符，或者操作系统的 shell，而且应该已经在本书前言的帮助下，安装了 Go。如果上面这些都准备好了，就让我们开始进入细节，了解什么是包，以及包为什么对 Go 语言的生态非常重要。

3.1 包

所有 Go 语言的程序都会组织成若干组文件，每组文件被称为一个包。这样每个包的代码都可以作为很小的复用单元，被其他项目引用。让我们看看标准库中的 http 包是怎么利用包的特性组织功能的：

```
net/http/  
  cgi/  
  cookiejar/  
    testdata/  
  fcgi/  
  httptest/  
  httputil/  
  pprof/  
  testdata/
```

这些目录包括一系列以 .go 为扩展名的相关文件。这些目录将实现 HTTP 服务器、客户端、

测试工具和性能调试工具的相关代码拆分成功能清晰的、小的代码单元。以 `cookiejar` 包为例，这个包里包含与存储和获取网页会话上的 `cookie` 相关的代码。每个包都可以单独导入和使用，以便开发者可以根据自己的需要导入特定功能。例如，如果要想实现 HTTP 客户端，只需要导入 `http` 包就可以。

所有的 `.go` 文件，除了空行和注释，都应该在第一行声明自己所属的包。每个包都在一个单独的目录里。不能把多个包放到同一个目录中，也不能把同一个包的文件分拆到多个不同目录中。这意味着，同一个目录下的所有 `.go` 文件必须声明同一个包名。

3.1.1 包名惯例

给包命名的惯例是使用包所在目录的名字。这让用户在导入包的时候，就能清晰地知道包名。我们继续以 `net/http` 包为例，在 `http` 目录下的所有文件都属于 `http` 包。给包及其目录命名时，应该使用简洁、清晰且全小写的名字，这有利于开发时频繁输入包名。例如，`net/http` 包下面的包，如 `cgi`、`httputil` 和 `pprof`，名字都很简洁。

记住，并不需要所有包的名字都与别的包不同，因为导入包时使用的是完整路径的，所以可以区分同名的不同包。一般情况下，包被导入后会使用你的包名作为前缀，不过这个导入后的名字可以修改。这个特性在需要导入不同目录的同名包时很有用。 3.2 节会展示如何修改导入的包名。



3.1.2 `main` 包

在 Go 语言里，命名为 `main` 的包具有特殊的含义。Go 语言的编译程序会试图把这种名字的包编译为二进制可执行文件。所有用 Go 语言编译的可执行程序都必须有一个名叫 `main` 的包。

当编译器发现某个包的名字为 `main` 时，它一定也会发现名为 `main()` 的函数，否则不会创建可执行文件。`main()` 函数是程序的入口，所以，如果没有这个函数，程序就没有办法开始执行。程序编译时，会使用声明 `main` 包的代码所在的目录的目录名作为二进制可执行文件的文件名。

命令和包 Go 文档里经常使用命令（`command`）这个词来指代可执行程序，如命令行应用程序。这会让新手在阅读文档时产生困惑。记住，在 Go 语言里，命令是指任何可执行程序。作为对比，包更常用来指语义上可导入的功能单元。

让我们来实际体验一下。首先，在 `$GOPATH/src/hello/` 目录里创建一个叫 `hello.go` 的文件，并输入代码清单 3-1 里的内容。这是个经典的“Hello World!”程序，不过，注意一下包的声明以及 `import` 语句。

代码清单 3-1 经典的“Hello World!”程序

```
01 package main
02
03 import "fmt"
```

← `fmt` 包提供了完成
格式化输出的功能。

```

04
05 func main() {
06     fmt.Println("Hello World!")
07 }

```

获取包的文档 别忘了，可以访问 <http://golang.org/pkg/fmt/> 或者在终端输入 `godoc fmt` 来了解更多关于 `fmt` 包的细节。

保存了文件后，可以在 `$GOPATH/src/hello/` 目录里执行命令 `go build`。这条命令执行完后，会生成一个二进制文件。在 UNIX、Linux 和 Mac OS X 系统上，这个文件会命名为 `hello`，而在 Windows 系统上会命名为 `hello.exe`。可以执行这个程序，并在控制台上显示 “Hello World!”。

如果把这个包名改为 `main` 之外的某个名字，如 `hello`，编译器就认为这只是一个包，而不是命令，如代码清单 3-2 所示。

代码清单 3-2 包含 `main` 函数的无效的 Go 程序

```

01 package hello
02
03 import "fmt"
04
05 func main(){
06     fmt.Println("Hello, World!")
07 }

```

3.2 导入

我们已经了解如何把代码组织到包里，现在让我们来看看如何导入这些包，以便可以访问包内的代码。`import` 语句告诉编译器到磁盘的哪里去找想要导入的包。导入包需要使用关键字 `import`，它会告诉编译器你想引用该位置的包内的代码。如果需要导入多个包，习惯上是 `import` 语句包装在一个导入块中，代码清单 3-3 展示了一个例子。

代码清单 3-3 `import` 声明块

```

import (
    "fmt"
    "strings"
)

```

`strings` 包提供了很多关于字符串的操作，如查找、替换或者变换。可以通过访问 <http://golang.org/pkg/strings/> 或者在终端运行 `godoc strings` 来了解更多关于 `strings` 包的细节。

编译器会使用 Go 环境变量设置的路径，通过引入的相对路径来查找磁盘上的包。标准库中的包会在安装 Go 的位置找到。Go 开发者创建的包会在 `GOPATH` 环境变量指定的目录里查找。`GOPATH` 指定的这些目录就是开发者的个人工作空间。

举个例子。如果 Go 安装在 `/usr/local/go`，并且环境变量 `GOPATH` 设置为 `/home/myproject/home/mylibraries`，编译器就会按照下面的顺序查找 `net/http` 包：

```
/usr/local/go/src/pkg/net/http  ← 这就是标准库源代码所在的位置。  
/home/myproject/src/net/http  
/home/mylibraries/src/net/http
```

一旦编译器找到一个满足 `import` 语句的包，就停止进一步查找。有一件重要的事需要记住，编译器会首先查找 Go 的安装目录，然后才会按顺序查找 `GOPATH` 变量里列出的目录。

如果编译器查遍 `GOPATH` 也没有找到要导入的包，那么在试图对程序执行 `run` 或者 `build` 的时候就会出错。本章后面会介绍如何通过 `go get` 命令来修正这种错误。

3.2.1 远程导入

目前的大势所趋是，使用分布式版本控制系统（Distributed Version Control Systems, DVCS）来分享代码，如 GitHub、Launchpad 还有 Bitbucket。Go 语言的工具链本身就支持从这些网站及类似网站获取源代码。Go 工具链会使用导入路径确定需要获取的代码在网络的什么地方。

例如：

```
import "github.com/spfl3/viper"
```

用导入路径编译程序时，`go build` 命令会使用 `GOPATH` 的设置，在磁盘上搜索这个包。事实上，这个导入路径代表一个 URL，指向 GitHub 上的代码库。如果路径包含 URL，可以使用 Go 工具链从 DVCS 获取包，并把包的源代码保存在 `GOPATH` 指向的路径里与 URL 匹配的目录里。这个获取过程使用 `go get` 命令完成。`go get` 将获取任意指定的 URL 的包，或者一个已经导入的包所依赖的其他包。由于 `go get` 的这种递归特性，这个命令会扫描某个包的源码树，获取能找到的所有依赖包。

3.2.2 命名导入

如果要导入的多个包具有相同的名字，会发生什么？例如，既需要 `network/convert` 包来转换从网络读取的数据，又需要 `file/convert` 包来转换从文本文件读取的数据时，就会同时导入两个名叫 `convert` 的包。这种情况下，重名的包可以通过命名导入来导入。命名导入是指，在 `import` 语句给出的包路径的左侧定义一个名字，将导入的包命名为新名字。

例如，若用户已经使用了标准库里的 `fmt` 包，现在要导入自己项目里名叫 `fmt` 的包，就可以通过代码清单 3-4 所示的命名导入方式，在导入时重新命名自己的包。

代码清单 3-4 重命名导入

```
01 package main  
02  
03 import (  
04     "fmt"  
05     myfmt "mylib/fmt"  
06 )  
07  
08 func main() {
```

```

09     fmt.Println("Standard Library")
10     myfmt.Println("mylib/fmt")
11 }

```

当你导入了一个不在代码里使用的包时，Go 编译器会编译失败，并输出一个错误。Go 开发团队认为，这个特性可以防止导入了未被使用的包，避免代码变得臃肿。虽然这个特性会让人觉得很烦，但 Go 开发团队仍然花了很大的力气说服自己，决定加入这个特性，用来避免其他编程语言里常常遇到的一些问题，如得到一个塞满未使用库的超大可执行文件。很多语言在这种情况下会使用警告做提示，而 Go 开发团队认为，与其让编译器告警，不如直接失败更有意义。每个编译过大型 C 程序的人都知道，在浩如烟海的编译器警告里找到一条有用的信息是多么困难的一件事。这种情况下编译失败会更加明确。

有时，用户可能需要导入一个包，但是不需要引用这个包的标识符。在这种情况下，可以使用空白标识符 `_` 来重命名这个导入。我们下节会讲到这个特性的用法。

空白标识符 下划线字符 (`_`) 在 Go 语言里称为空白标识符，有很多用法。这个标识符用来抛弃不想继续使用的值，如给导入的包赋予一个空名字，或者忽略函数返回的你不感兴趣的值。

3.3 函数 init

每个包可以包含任意多个 `init` 函数，这些函数都会在程序执行开始的时候被调用。所有被编译器发现的 `init` 函数都会安排在 `main` 函数之前执行。`init` 函数用在设置包、初始化变量或者其他要在程序运行前优先完成的引导工作。

以数据库驱动为例，`database` 下的驱动在启动时执行 `init` 函数会将自身注册到 `sql` 包里，因为 `sql` 包在编译时并不知道这些驱动的存在，等启动之后 `sql` 才能调用这些驱动。让我们看看这个过程中 `init` 函数做了什么，如代码清单 3-5 所示。

代码清单 3-5 `init` 函数的用法

```

01 package postgres
02
03 import (
04     "database/sql"
05 )
06
07 func init() {
08     sql.Register("postgres", new(PostgresDriver))
09 }

```

创建一个 postgres 驱动的实例。这里为了展现 `init` 的作用，没有展现其定义细节。

这段示例代码包含在 PostgreSQL 数据库的驱动里。如果程序导入了这个包，就会调用 `init` 函数，促使 PostgreSQL 的驱动最终注册到 Go 的 `sql` 包里，成为一个可用的驱动。

在使用这个新的数据库驱动写程序时，我们使用空白标识符来导入包，以便新的驱动会包含到 `sql` 包。如前所述，不能导入不使用的包，为此使用空白标识符重命名这个导入可以让 `init` 函数发现并被调度运行，让编译器不会因为包未被使用而产生错误。

现在我们可以调用 `sql.Open` 方法来使用这个驱动，如代码清单 3-6 所示。

代码清单 3-6 导入时使用空白标识符作为包的别名

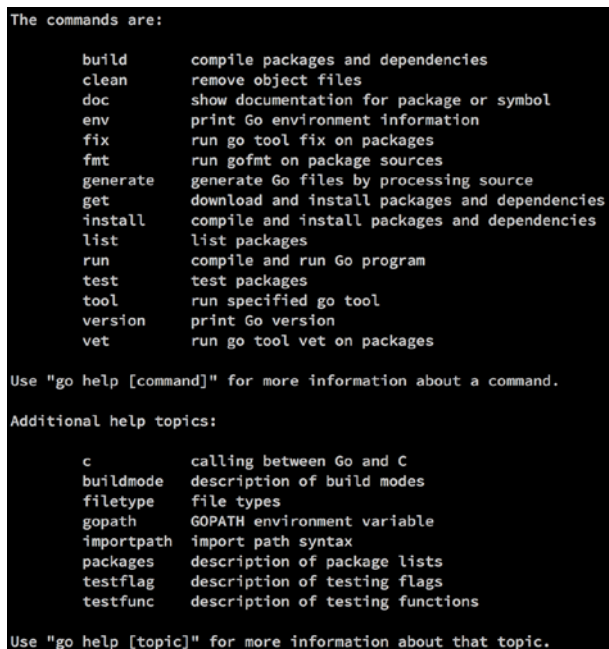
```
01 package main
02
03 import (                                使用空白标识符导入
04     "database/sql"                      包，避免编译错误。
05
06     _ "github.com/goinaction/code/chapter3/dbdriver/postgres"  ←
07 )
08
09 func main() {
10     sql.Open("postgres", "mydb")  ← 调用 sql 包提供的 Open 方法。该方法能
11 }                                  工作的关键在于 postgres 驱动通过自
                                  己的 init 函数将自身注册到了 sql 包。
```

3.4 使用 Go 的工具

在前几章里，我们已经使用过了 `go` 这个工具，但我们还没有探讨这个工具都能做哪些事情。让我们进一步深入了解这个短小的命令，看看都有哪些强大的能力。在命令行提示符下，不带参数直接键入 `go` 这个命令：

```
$ go
```

`go` 这个工具提供了很多功能，如图 3-1 所示。



```
The commands are:

    build      compile packages and dependencies
    clean      remove object files
    doc        show documentation for package or symbol
    env        print Go environment information
    fix        run go tool fix on packages
    fmt        run gofmt on package sources
    generate    generate Go files by processing source
    get        download and install packages and dependencies
    install    compile and install packages and dependencies
    list       list packages
    run        compile and run Go program
    test       test packages
    tool       run specified go tool
    version    print Go version
    vet        run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

    c          calling between Go and C
    buildmode  description of build modes
    filetype   file types
    gopath     GOPATH environment variable
    importpath import path syntax
    packages   description of package lists
    testflag   description of testing flags
    testfunc   description of testing functions

Use "go help [topic]" for more information about that topic.
```

图 3-1 `go` 命令输出的帮助文本

通过输出的列表可以看到，这个命令包含一个编译器，这个编译器可以通过 `build` 命令启动。正如预料的那样，`build` 和 `clean` 命令会执行编译和清理的工作。现在使用代码清单 3-2 里的源代码，尝试执行这些命令：

```
go build hello.go
```

当用户将代码签入源码库里的时候，开发人员可能并不想签入编译生成的文件。可以用 `clean` 命令解决这个问题：

```
go clean hello.go
```

调用 `clean` 后会删除编译生成的可执行文件。让我们看看 `go` 工具的其他一些特性，以及使用这些命令时可以节省时间的方法。接下来的例子中，我们会使用代码清单 3-7 中的样例代码。

代码清单 3-7 使用 `io` 包的工作

```
01 package main
02
03 import (
04     "fmt"
05     "io/ioutil"
06     "os"
07
08     "github.com/goinaction/code/chapter3/words"
09 )
10
11 // main 是应用程序的入口
12 func main() {
13     filename := os.Args[1]
14
15     contents, err := ioutil.ReadFile(filename)
16     if err != nil {
17         fmt.Println(err)
18         return
19     }
20
21     text := string(contents)
22
23     count := words.CountWords(text)
24     fmt.Printf("There are %d words in your text. \n", count)
25 }
```

如果已经下载了本书的源代码，应该可以在 `$GOPATH/src/github.com/goinaction/code/chapter3/words` 找到这个包。确保已经有了这段代码再进行后面的内容。

大部分 `Go` 工具的命令都会接受一个包名作为参数。回顾一下已经用过的命令，会想起 `build` 命令可以简写。在不包含文件名时，`go` 工具会默认使用当前目录来编译。

```
go build
```

因为构建包是很常用的动作，所以也可以直接指定包：

```
go build github.com/goinaction/code/chapter3/wordcount
```

也可以在指定包的时候使用通配符。3 个点表示匹配所有的字符串。例如，下面的命令会编译 `chapter3` 目录下的所有包：

```
go build github.com/goinaction/code/chapter3/...
```

除了指定包，大部分 Go 命令使用短路径作为参数。例如，下面两条命令的效果相同：

```
go build wordcount.go
```

```
go build .
```

要执行程序，需要首先编译，然后执行编译创建的 `wordcount` 或者 `wordcount.exe` 程序。不过这里有一个命令可以在一次调用中完成这两个操作：

```
go run wordcount.go
```

`go run` 命令会先构建 `wordcount.go` 里包含的程序，然后执行构建后的程序。这样可以节省好多录入工作量。

做开发会经常使用 `go build` 和 `go run` 命令。让我们看另外几个可用的命令，以及这些命令可以做什么。

3.5 进一步介绍 Go 开发工具

我们已经学到如何用 `go` 这个通用工具进行编译和执行。但这个好用的工具还有很多其他没有介绍的诀窍。

3.5.1 `go vet`

这个命令不会帮开发人员写代码，但如果开发人员已经写了一些代码，`vet` 命令会帮开发人员检测代码的常见错误。让我们看看 `vet` 捕获哪些类型的错误。

- `Printf` 类函数调用时，类型匹配错误的参数。
- 定义常用的方法时，方法签名的错误。
- 错误的结构标签。
- 没有指定字段名的结构字面量。

让我们看看许多 Go 开发新手经常犯的一个错误。`fmt.Printf` 函数常用来产生格式化输出，不过这个函数要求开发人员记住所有不同的格式化说明符。代码清单 3-8 中给出的就是一个例子。

代码清单 3-8 使用 `go vet`

```
01 package main
02
03 import "fmt"
04
```

```
05 func main() {  
06     fmt.Printf("The quick brown fox jumped over lazy dogs", 3.14)  
07 }
```

这个程序要输出一个浮点数 3.14，但是在格式化字符串里并没有对应的格式化参数。如果对这段代码执行 `go vet`，会得到如下消息：

```
go vet main.go  
  
main.go:6: no formatting directive in Printf call
```

`go vet` 工具不能让开发者避免严重的逻辑错误，或者避免编写充满小错的代码。不过，正像刚才的实例中展示的那样，这个工具可以很好地捕获一部分常见错误。每次对代码先执行 `go vet` 再将其签入源代码库是一个很好的习惯。

3.5.2 Go 代码格式化

`fmt` 是 Go 语言社区很喜欢的一个命令。`fmt` 工具会将开发人员的代码布局成和 Go 源代码类似的风格，不用再为了大括号是不是要放到行尾，或者用 `tab`（制表符）还是空格来做缩进而争论不休。使用 `go fmt` 后面跟文件名或者包名，就可以调用这个代码格式化工具。`fmt` 命令会自动格式化开发人员指定的源代码文件并保存。下面是一个代码执行 `go fmt` 前和执行 `go fmt` 后几行代码的对比：

```
if err != nil { return err }  
  
在对这段代码执行 go fmt 后，会得到：  
  
if err != nil {  
    return err  
}
```

很多 Go 开发人员会配置他们的开发环境，在保存文件或者提交到代码库前执行 `go fmt`。如果读者喜欢这个命令，也可以这样做。

3.5.3 Go 语言的文档

还有另外一个工具能让 Go 开发过程变简单。Go 语言有两种方法为开发者生成文档。如果开发人员使用命令行提示符工作，可以在终端上直接使用 `go doc` 命令来打印文档。无需离开终端，即可快速浏览命令或者包的帮助。不过，如果开发人员认为一个浏览器界面会更有效率，可以使用 `godoc` 程序来启动一个 Web 服务器，通过点击的方式来查看 Go 语言的包的文档。Web 服务器 `godoc` 能让开发人员以网页的方式浏览自己的系统里的所有 Go 语言源代码的文档。

1. 从命令行获取文档

对那种总会打开一个终端和一个文本编辑器（或者在终端内打开文本编辑器）的开发人员来

说，go doc 是很好的选择。假设要用 Go 语言第一次开发读取 UNIX tar 文件的应用程序，想要看看 archive/tar 包的相关文档，就可以输入：

```
go doc tar
```

执行这个命令会直接在终端产生如下输出：

```
PACKAGE DOCUMENTATION
```

```
package tar // import "archive/tar"
```

```
Package tar implements access to tar archives. It aims to cover most of the variations, including those produced by GNU and BSD tars.
```

```
References:
```

```
    http://www.freebsd.org/cgi/man.cgi?query=tar&sektion=5
```

```
    http://www.gnu.org/software/tar/manual/html_node/Standard.html
```

```
    http://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html
```

```
var ErrWriteTooLong = errors.New("archive/tar: write too long") ...
```

```
var ErrHeader = errors.New("archive/tar: invalid tar header")
```

```
func FileInfoHeader(fi os.FileInfo, link string) (*Header, error)
```

```
func NewReader(r io.Reader) *Reader
```

```
func NewWriter(w io.Writer) *Writer
```

```
type Header struct { ... }
```

```
type Reader struct { ... }
```

```
type Writer struct { ... }
```

开发人员无需离开终端即可直接翻看文档，找到自己需要的部分。

2. 浏览文档

Go 语言的文档也提供了浏览器版本。有时候，通过跳转到文档，查阅相关的细节，能更容易理解整个包或者某个函数。在这种情况下，会使用 godoc 作为 Web 服务器。如果想通过 Web 浏览器查看可以点击跳转的文档，下面就是得到这种文档的好方式。

开发人员启动自己的文档服务器，只需要在终端会话中输入如下命令：

```
godoc -http=:6060
```

这个命令通知 godoc 在端口 6060 启动 Web 服务器。如果浏览器已经打开，导航到 <http://localhost:6060> 可以看到一个页面，包含所有 Go 标准库和你的 GOPATH 下的 Go 源代码的文档。

如果图 3-2 显示的文档对开发人员来说很熟悉，并不奇怪，因为 Go 官网就是通过一个略微修改过的 godoc 来提供文档服务的。要进入某个特定包的文档，只需要点击页面顶端的 Packages。

Go 文档工具最棒的地方在于，它也支持开发人员自己写的代码。如果开发人员遵从一个简单的规则来写代码，这些代码就会自动包含在 godoc 生成的文档里。

为了在 godoc 生成的文档里包含自己的代码文档，开发人员需要用下面的规则来写代码和注释。我们不会在本章介绍所有的规则，只会提一些重要的规则。

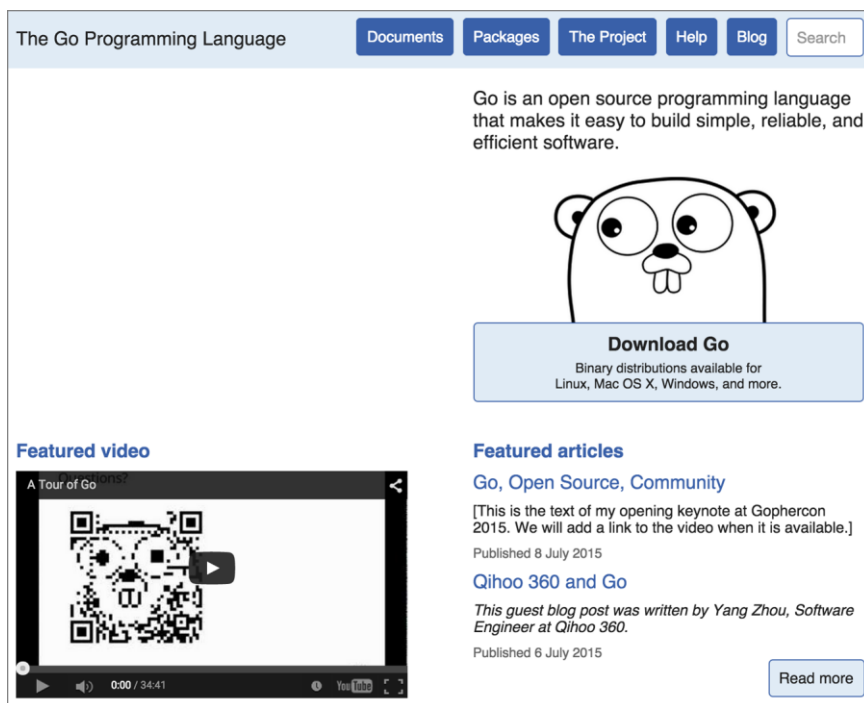


图 3-2 本地 Go 文档

用户需要在标识符之前，把自己想要的文档作为注释加入到代码中。这个规则对包、函数、类型和全局变量都适用。注释可以以双斜线开头，也可以用斜线和星号风格。

```
// Retrieve 连接到配置库，收集各种链接设置、用户名和密码。这个函数在成功时
// 返回 config 结构，否则返回一个错误。
func Retrieve() (config, error) {
    // ... 省略
}
```

在这个例子里，我们展示了在 Go 语言里为一个函数写文档的惯用方法。函数的文档直接写在函数声明之前，使用人类可读的句子编写。如果想给包写一段文字量比较大的文档，可以在工程里包含一个叫作 doc.go 的文件，使用同样的包名，并把包的介绍使用注释加在包名声明之前。

```
/*
    包 usb 提供了用于调用 USB 设备的类型和函数。想要与 USB 设备创建一个新链接，使用 NewConnection
    ...
*/
package usb
```

这段关于包的文档会显示在所有类型和函数文档之前。这个例子也展示了如何使用斜线和星号做注释。可以在 Google 上搜索 golang documentation 来查找更多关于如何给代码创建一个好文档的内容。

3.6 与其他 Go 开发者合作

现代开发者不会一个人单打独斗，而 Go 工具也认可这个趋势，并为合作提供了支持。多亏了 go 工具链，包的概念没有被限制在本地开发环境中，而是做了扩展，从而支持现代合作方式。让我们看看在分布式开发环境里，想要良好合作，需要遵守的一些惯例。

以分享为目的创建代码库

开发人员一旦写了些非常棒的 Go 代码，就会很想把这些代码与 Go 社区的其他人分享。这其实很容易，只需要执行下面的步骤就可以。

1. 包应该在代码库的根目录中

使用 go get 的时候，开发人员指定了要导入包的全路径。这意味着在创建想要分享的代码库的时候，包名应该就是代码库的名字，而且包的源代码应该位于代码库目录结构的根目录。

Go 语言新手常犯的一个错误是，在公用代码库里创建一个名为 code 或者 src 的目录。如果这么做，会让导入公用库的语句变得很长。为了避免过长的语句，只需要把包的源文件放在公用代码库的根目录就好。

2. 包可以非常小

与其他语言相比，Go 语言的包一般相对较小。不要在意包只支持几个 API，或者只完成一项任务。在 Go 语言里，这样的包很常见，而且很受欢迎。

3. 对代码执行 go fmt

和其他开源代码库一样，人们在试用代码前会通过源代码来判断代码的质量。开发人员需要在签入代码前执行 go fmt，这样能让自己的代码可读性更好，而且不会由于一些字符的干扰（如制表符），在不同人的计算机上代码显示的效果不一样。

4. 给代码写文档

Go 开发者用 godoc 来阅读文档，并且会用 <http://godoc.org> 这个网站来阅读开源包的文档。如果按照 go doc 的最佳实践来给代码写文档，包的文档在本地和线上都会很好看，更容易被别人发现。

3.7 依赖管理

从 Go 1.0 发布那天起，社区做了很多努力，提供各种 Go 工具，以便开发人员的工作更轻松。有很多工具专注在如何管理包的依赖关系。现在最流行的依赖管理工具是 Keith Rarik 写的 godep、

Daniel Theophanes 写的 `vender` 和 Gustavo Niemeyer 开发的 `gopkg.in` 工具。`gopkg.in` 能帮助开发人员发布自己的包的多个版本。

作为对社区的回应，Go 语言在 1.5 版本开始试验性提供一组新的构建选项和功能，来为依赖管理提供更好的工具支持。尽管我们还需要等一段时间才能确认这些新特性是否能达成目的，但毕竟现在已经有一些工具以可重复使用的方式提供了管理、构建和测试 Go 代码的能力。

3.7.1 第三方依赖

像 `godep` 和 `vender` 这种社区工具已经使用第三方（`verdoring`）导入路径重写这种特性解决了依赖问题。其思想是把所有的依赖包复制到工程代码库中的目录里，然后使用工程内部的依赖包所在目录来重写所有的导入路径。

代码清单 3-9 展示的是使用 `godep` 来管理工程里第三方依赖时的一个典型的源代码树。

代码清单 3-9 使用 `godep` 的工程

```
$GOPATH/src/github.com/ardanstudios/myproject
|-- Godeps
|   |-- Godeps.json
|   |-- README
|   |-- _workspace
|       |-- src
|           |-- bitbucket.org
|           |-- ww
|               |-- goautoneg
|                   |-- Makefile
|                   |-- README.txt
|                   |-- autoneg.go
|                   |-- autoneg_test.go
|           |-- github.com
|               |-- beorn7
|                   |-- perks
|                       |-- README.md
|                       |-- quantile
|                           |-- bench_test.go
|                           |-- example_test.go
|                           |-- exampledata.txt
|                           |-- stream.go
|-- examples
|-- model
|-- README.md
|-- main.go
```

可以看到 `godep` 创建了一个叫作 `Godeps` 的目录。由这个工具管理的依赖的源代码被放在一个叫作 `_workspace/src` 的目录里。

接下来，如果看一下在 `main.go` 里声明这些依赖的 `import` 语句（如代码清单 3-9 和代码清单 3-10 所示），就能发现需要改动的地方。

代码清单 3-10 在路径重写之前

```
01 package main
02
03 import (
04     "bitbucket.org/ww/goautoneg"
05     "github.com/beorn7/perks"
06 )
```

代码清单 3-11 在路径重写之后

```
01 package main
02
03 import (
04     "github.ardanstudios.com/myproject/Godeps/_workspace/src/
                                bitbucket.org/ww/goautoneg"
05     "github.ardanstudios.com/myproject/Godeps/_workspace/src/
                                github.com/beorn7/perks"
06 )
```

在路径重写之前，`import` 语句使用的是包的正常路径。包对应的代码存放在 `GOPATH` 所指定的磁盘目录里。在依赖管理之后，导入路径需要重写成工程内部依赖包的路径。可以看到这些导入路径非常长，不易于使用。

引入依赖管理将所有构建时依赖的源代码都导入到一个单独的工程代码库里，可以更容易地重新构建工程。使用导入路径重写管理依赖包的另外一个好处是这个工程依旧支持通过 `go get` 获取代码库。当获取这个工程的代码库时，`go get` 可以找到每个包，并将其保存到工程里正确的目录中。

3.7.2 对 gb 的介绍

`gb` 是一个由 Go 社区成员开发的全新的构建工具。`gb` 意识到，不一定要包装 Go 本身的工具，也可以使用其他方法来解决可重复构建的问题。

`gb` 背后的原理源自理解到 Go 语言的 `import` 语句并没有提供可重复构建的能力。`import` 语句可以驱动 `go get`，但是 `import` 本身并没有包含足够的信息来决定到底要获取包的哪个修改的版本。`go get` 无法定位待获取代码的问题，导致 Go 工具在解决重复构建时，不得不使用复杂且难看的方法。我们已经看到过使用 `godep` 时超长的导入路径是多么难看。

`gb` 的创建源于上述理解。`gb` 既不包装 Go 工具链，也不使用 `GOPATH`。`gb` 基于工程将 Go 工具链工作空间的元信息做替换。这种依赖管理的方法不需要重写工程内代码的导入路径。而且导入路径依旧通过 `go get` 和 `GOPATH` 工作空间来管理。

让我们看看上一节的工程如何转换为 `gb` 工程，如代码清单 3-12 所示。

代码清单 3-12 gb 工程的例子

```
/home/bill/devel/myproject ($PROJECT)
|-- src
```

```

| | | | -- cmd
| | | | | | -- myproject
| | | | | | | | -- main.go
| | | | -- examples
| | | | -- model
| | | | -- README.md
| | -- vendor
| | | -- src
| | | | -- bitbucket.org
| | | | | | -- ww
| | | | | | | | -- goautoneg
| | | | | | | | -- Makefile
| | | | | | | | -- README.txt
| | | | | | | | -- autoneg.go
| | | | | | | | -- autoneg_test.go
| | | | -- github.com
| | | | | | -- beorn7
| | | | | | | | -- perks
| | | | | | | | -- README.md
| | | | | | | | -- quantile
| | | | | | | | -- bench_test.go
| | | | -- example_test.go
| | | | -- exampledata.txt
| | | | -- stream.go

```

一个 **gb** 工程就是磁盘上一个包含 **src/**子目录的目录。符号 **\$PROJECT** 导入了工程的根目录中，其下有一个 **src/**的子目录中。这个符号只是一个简写，用来描述工程在磁盘上的位置。**\$PROJECT** 不是必须设置的环境变量。事实上，**gb** 根本不需要设置任何环境变量。

gb 工程会区分开发人员写的代码和开发人员需要依赖的代码。开发人员的代码所依赖的代码被称作**第三方代码**（**vendored code**）。**gb** 工程会明确区分开发人员的代码和第三方代码，如代码清单 3-13 和代码清单 3-14 所示。

代码清单 3-13 工程中存放开发人员写的代码的位置

```
$PROJECT/src/
```

代码清单 3-14 存放第三方代码的位置

```
$PROJECT/vendor/src/
```

gb 一个最好的特点是，不需要重写导入路径。可以看看这个工程里的 **main.go** 文件的 **import** 语句——没有任何需要为导入第三方库而做的修改，如代码清单 3-15 所示。

代码清单 3-15 gb 工程的导入路径

```

01 package main
02
03 import (
04     "bitbucket.org/ww/goautoneg"
05     "github.com/beorn7/perks"
06 )

```

gb 工具首先会在\$PROJECT/src/目录中查找代码,如果找不到,会在\$PROJECT/vendor/src/目录里查找。与工程相关的整个源代码都会在同一代码库里。自己写的代码在工程目录的src/目录中,第三方依赖代码在工程目录的vendor/src子目录中。这样,不需要配合重写导入路径也可以完成整个构建过程,同时可以把整个工程放到磁盘的任意位置。这些特点,让gb成为社区里解决可重复构建的流行工具。

还需要提一点:gb工程与Go官方工具链(包括go get)并不兼容。因为gb不需要设置GOPATH,而Go工具链无法理解gb工程的目录结构,所以无法用Go工具链构建、测试或者获取代码。构建(如代码清单3-16所示)和测试gb工程需要先进入\$PROJECT目录,并使用gb工具。

代码清单 3-16 构建 gb 工程

```
gb build all
```

很多Go工具支持的特性,gb都提供对应的特性。gb还提供了插件系统,可以让社区扩展支持的功能。其中一个插件叫作vender。这个插件可以方便地管理\$PROJECT/vendor/src/目录里的依赖关系,而这个功能Go工具链至今没有提供。想了解更多gb的特性,可以访问这个网站:getgb.io。

3.8 小结

- 在Go语言中包是组织代码的基本单位。
- 环境变量GOPATH决定了Go源代码在磁盘上被保存、编译和安装的位置。
- 可以为每个工程设置不同的GOPATH,以保持源代码和依赖的隔离。
- go工具是在命令行上工作的最好工具。
- 开发人员可以使用go get来获取别人的包并将其安装到自己的GOPATH指定的目录。
- 想要为别人创建包很简单,只要把源代码放到公用代码库,并遵守一些简单规则就可以了。
- Go语言在设计时将分享代码作为语言的核心特性和驱动力。
- 推荐使用依赖管理工具来管理依赖。
- 有很多社区开发的依赖管理工具,如godep、vender和gb。

第 4 章 数组、切片和映射

本章主要内容

- 数组的内部实现和基础功能
- 使用切片管理数据集合
- 使用映射管理键值对

很难遇到要编写一个不需要存储和读取集合数据的程序的情况。如果使用数据库或者文件，或者访问网络，总需要一种方法来处理接收和发送的数据。Go 语言有 3 种数据结构可以让用户管理集合数据：数组、切片和映射。这 3 种数据结构是语言核心的一部分，在标准库里被广泛使用。一旦学会如何使用这些数据结构，用 Go 语言编写程序会变得快速、有趣且十分灵活。

4.1 数组的内部实现和基础功能

了解这些数据结构，一般会从数组开始，因为数组是切片和映射的基础数据结构。理解了数组的工作原理，有助于理解切片和映射提供的优雅和强大的功能。

4.1.1 内部实现

在 Go 语言里，数组是一个长度固定的数据类型，用于存储一段具有相同的类型的元素的连续块。数组存储的类型可以是内置类型，如整型或者字符串，也可以是某种结构类型。

在图 4-1 中可以看到数组的表示。灰色格子代表数组里的元素，每个元素都紧邻另一个元素。每个元素包含相同的类型，这个例子里是整数，并且每个元素可以用一个唯一的索引（也称下标或标号）来访问。

数组是一种非常有用的数据结构，因为其占用的内存是连续分配的。由于内存连续，CPU 能把正在使用的数据缓存更久的时间。而且内存连续很容易计算索引，可以快速迭代数组里的所有元素。数组的类型信息可以提供每次访问一个元素时需要在内存中移动的距离。既然数组的每个元素类型相同，又是连续分配，就可以以固定速度索引数组中的任意数据，速度非常快。

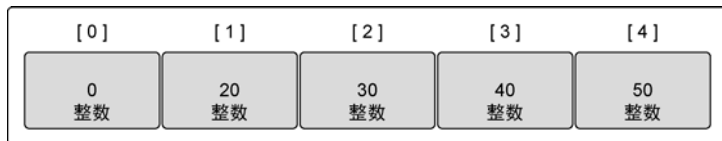


图 4-1 数组的内部实现

4.1.2 声明和初始化

声明数组时需要指定内部存储的数据的类型，以及需要存储的元素的数量，这个数量也称为数组的长度，如代码清单 4-1 所示。

代码清单 4-1 声明一个数组，并设置为零值

```
// 声明一个包含 5 个元素的整型数组  
var array [5]int
```

一旦声明，数组里存储的数据类型和数组长度就都不能改变了。如果需要存储更多的元素，就需要先创建一个更长的数组，再把原来数组里的值复制到新数组里。

在 Go 语言中声明变量时，总会使用对应类型的零值来对变量进行初始化。数组也不例外。当数组初始化时，数组内每个元素都初始化为对应类型的零值。在图 4-2 里，可以看到整型数组里的每个元素都初始化为 0，也就是整型的零值。



图 4-2 声明数组变量后数组的值

一种快速创建数组并初始化的方式是使用数组字面量。数组字面量允许声明数组里元素的数量同时指定每个元素的值，如代码清单 4-2 所示。

代码清单 4-2 使用数组字面量声明数组

```
// 声明一个包含 5 个元素的整型数组  
// 用具体值初始化每个元素  
array := [5]int{10, 20, 30, 40, 50}
```

如果使用...替代表组的长度，Go 语言会根据初始化时数组元素的数量来确定该数组的长度，如代码清单 4-3 所示。

代码清单 4-3 让 Go 自动计算声明数组的长度

```
// 声明一个整型数组  
// 用具体值初始化每个元素
```

```
// 容量由初始化值的数量决定
array := [...]int{10, 20, 30, 40, 50}
```

如果知道数组的长度而是准备给每个值都指定具体值，就可以使用代码清单 4-4 所示的这种语法。

代码清单 4-4 声明数组并指定特定元素的值

```
// 声明一个有 5 个元素的数组
// 用具体值初始化索引为 1 和 2 的元素
// 其余元素保持零值
array := [5]int{1: 10, 2: 20}
```

代码清单 4-4 中声明的数组在声明后，会和图 4-3 所展现的一样。



图 4-3 声明之后数组的值

4.1.3 使用数组

正像之前提到的，因为内存布局是连续的，所以数组是效率很高的数据结构。在访问数组里任意元素的时候，这种高效都是数组的优势。要访问数组里某个单独元素，使用[]运算符，如代码清单 4-5 所示。

代码清单 4-5 访问数组元素

```
// 声明一个包含 5 个元素的整型数组
// 用具体值初始为每个元素
array := [5]int{10, 20, 30, 40, 50}
```

```
// 修改索引为 2 的元素的值
array[2] = 35
```

代码清单 4-5 中声明的数组的值在操作完成后，会和图 4-4 所展现的一样。

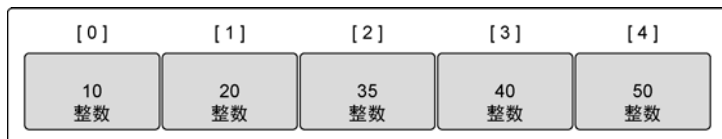


图 4-4 修改索引为 2 的值之后数组的值

可以像第 2 章一样，声明一个所有元素都是指针的数组。使用*运算符就可以访问元素指针所指向的值，如代码清单 4-6 所示。

代码清单 4-6 访问指针数组的元素

```
// 声明包含 5 个元素的指向整数的数组
// 用整型指针初始化索引为 0 和 1 的数组元素
array := [5]*int{0: new(int), 1: new(int)}

// 为索引为 0 和 1 的元素赋值
*array[0] = 10
*array[1] = 20
```

代码清单 4-6 中声明的数组的值在操作完毕后，会和图 4-5 所展现的一样。

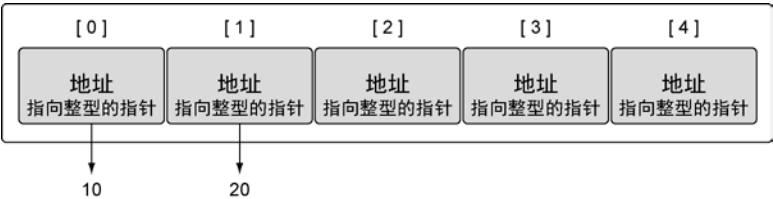


图 4-5 指向整数的指针数组

在 Go 语言里，数组是一个值。这意味着数组可以用在赋值操作中。变量名代表整个数组，因此，同样类型的数组可以赋值给另一个数组，如代码清单 4-7 所示。

代码清单 4-7 把同样类型的一个数组赋值给另外一个数组

```
// 声明第一个包含 5 个元素的字符串数组
var array1 [5]string

// 声明第二个包含 5 个元素的字符串数组
// 用颜色初始化数组
array2 := [5]string{"Red", "Blue", "Green", "Yellow", "Pink"}

// 把 array2 的值复制到 array1
array1 = array2
```

复制之后，两个数组一样，如图 4-6 所示。

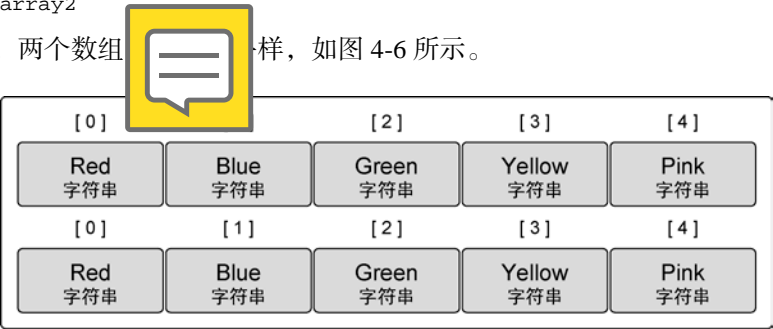


图 4-6 复制之后的两个数组

数组变量的类型包括数组长度和每个元素的类型。只有这两部分都相同的数组，才是类型相

同的数组，才能互相赋值，如代码清单 4-8 所示。

代码清单 4-8 编译器会阻止类型不同的数组互相赋值

```
// 声明第一个包含 4 个元素的字符串数组
var array1 [4]string

// 声明第二个包含 5 个元素的字符串数组
// 使用颜色初始化数组
array2 := [5]string{"Red", "Blue", "Green", "Yellow", "Pink"}

// 将 array2 复制给 array1
array1 = array2

Compiler Error:
cannot use array2 (type [5]string) as type [4]string in assignment
```

复制数组指针，只会复制指针的值，而不会复制指针所指向的值，如代码清单 4-9 所示。

代码清单 4-9 把一个指针数组赋值给另一个

```
// 声明第一个包含 3 个元素的指向字符串的指针数组
var array1 [3]*string

// 声明第二个包含 3 个元素的指向字符串的指针数组
// 使用字符串指针初始化这个数组
array2 := [3]*string{new(string), new(string), new(string)}

// 使用颜色为每个元素赋值
*array2[0] = "Red"
*array2[1] = "Blue"
*array2[2] = "Green"

// 将 array2 复制给 array1
array1 = array2
```

复制之后，两个数组指向同一组字符串，如图 4-7 所示。

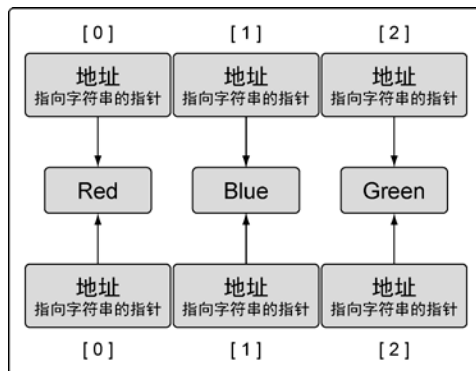


图 4-7 两组指向同样字符串的数组