

源中搜索数据。

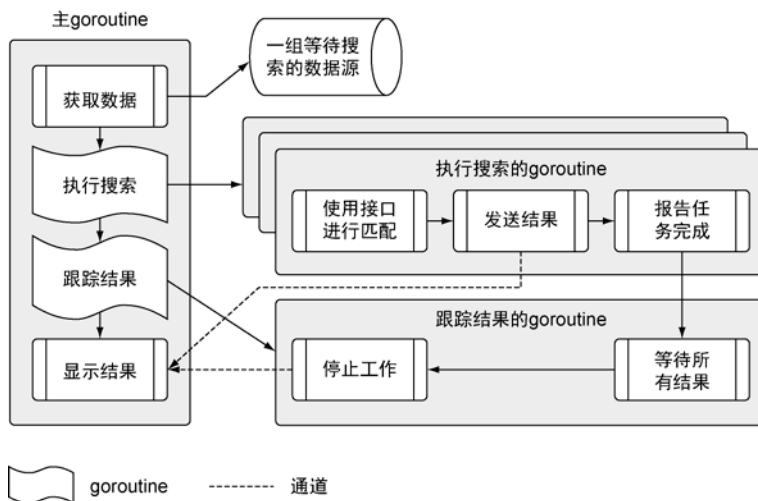


图 2-1 程序架构流程图

这个程序分成多个不同步骤，在多个不同的 goroutine 里运行。我们会根据流程展示代码，从主 goroutine 开始，一直到执行搜索的 goroutine 和跟踪结果的 goroutine，最后回到主 goroutine。首先来看一下整个项目的结构，如代码清单 2-1 所示。

代码清单 2-1 应用程序的项目结构

```
cd $GOPATH/src/github.com/goinaction/code/chapter2

- sample
  - data
    data.json  -- 包含一组数据源
  - matchers
    rss.go     -- 搜索 rss 源的匹配器
  - search
    default.go -- 搜索数据用的默认匹配器
    feed.go    -- 用于读取 json 数据文件
    match.go   -- 用于支持不同匹配器的接口
    search.go  -- 执行搜索的主控制逻辑
  main.go     -- 程序的入口
```

这个应用的代码使用了 4 个文件夹，按字母顺序列出。文件夹 data 中有一个 JSON 文档，其内容是程序要拉取和处理的数据源。文件夹 matchers 中包含程序里用于支持搜索不同数据源的代码。目前程序只完成了支持处理 RSS 类型的数据源的匹配器。文件夹 search 中包含使用不同匹配器进行搜索的业务逻辑。最后，父级文件夹 sample 中有个 main.go 文件，这是整个程序的入口。

现在了解了如何组织程序的代码，可以继续探索并了解程序是如何工作的。让我们从程序的入口开始。

2.2 main 包

程序的主入口可以在 `main.go` 文件里找到，如代码清单 2-2 所示。虽然这个文件只有 21 行代码，依然有几点需要注意。

代码清单 2-2 `main.go`

```
01 package main
02
03 import (
04     "log"
05     "os"
06
07     _ "github.com/goinaction/code/chapter2/sample/matchers"
08     "github.com/goinaction/code/chapter2/sample/search"
09 )
10
11 // init 在 main 之前调用
12 func init() {
13     // 将日志输出到标准输出
14     log.SetOutput(os.Stdout)
15 }
16
17 // main 是整个程序的入口
18 func main() {
19     // 使用特定的项做搜索
20     search.Run("president")
21 }
```

每个可执行的 Go 程序都有两个明显的特征。一个特征是第 18 行声明的名为 `main` 的函数。构建程序在构建可执行文件时，需要找到这个已经声明的 `main` 函数，把它作为程序的入口。第二个特征是程序的第 01 行的包名 `main`，如代码清单 2-3 所示。

代码清单 2-3 `main.go`：第 01 行

```
01 package main
```

可以看到，`main` 函数保存在名为 `main` 的包里。如果 `main` 函数不在 `main` 包里，构建工具就不会生成可执行的文件。

Go 语言的每个代码文件都属于一个包，`main.go` 也不例外。包这个特性对于 Go 语言来说很重要，我们会在第 3 章中接触到更多细节。现在，只要简单了解以下内容：一个包定义一组编译过的代码，包的名字类似命名空间，可以用来间接访问包内声明的标识符。这个特性可以把不同包中定义的同名标识符区别开。

现在，把注意力转到 `main.go` 的第 03 行到第 09 行，如代码清单 2-4 所示，这里声明了所有的导入项。

代码清单 2-4 main.go: 第 03 行到第 09 行

```
03 import (  
04     "log"  
05     "os"  
06  
07     _ "github.com/goinaction/code/chapter2/sample/matchers"  
08     "github.com/goinaction/code/chapter2/sample/search"  
09 )
```



顾名思义，关键字 `import` 就是导入一段代码，让用户可以访问其标识符，如类型、函数、常量和接口。在这个例子中，由于第 08 行的导入，`main.go` 里的代码就可以引用 `search` 包里的 `Run` 函数。程序的第 04 行和第 05 行导入标准库里的 `log` 和 `os` 包。

所有处于同一个文件夹里的代码文件，必须使用同一个包名。按照惯例，包和文件夹同名。就像之前说的，一个包定义一组编译后的代码，每段代码都描述包的一部分。如果回头去看看代码清单 2-1，可以看看第 08 行的导入是如何指定那个项目里名叫 `search` 的文件夹的。

读者可能注意到第 07 行导入 `matchers` 包的时候，导入的路径前面有一个下划线，如代码清单 2-5 所示。

代码清单 2-5 main.go: 第 07 行

```
07     _ "github.com/goinaction/code/chapter2/sample/matchers"
```

这个技术是为了让 Go 语言对包做初始化操作，但是并不使用包里的标识符。为了让程序的可读性更强，Go 编译器不允许声明导入某个包却不使用。下划线让编译器接受这类导入，并且调用对应包内的所有代码文件里定义的 `init` 函数。对这个程序来说，这样做的目的是调用 `matchers` 包中的 `rss.go` 代码文件里的 `init` 函数，注册 RSS 匹配器，以便后用。我们后面会展示具体的工作方式。

代码文件 `main.go` 里也有一个 `init` 函数，在第 12 行到第 15 行中声明，如代码清单 2-6 所示。

代码清单 2-6 main.go: 第 11 行到第 15 行

```
11 // init 在 main 之前调用  
12 func init() {  
13     // 将日志输出到标准输出  
14     log.SetOutput(os.Stdout)  
15 }
```

程序中每个代码文件里的 `init` 函数都会在 `main` 函数执行前调用。这个 `init` 函数将标准库里日志类的输出，从默认的标准错误 (`stderr`)，设置为标准输出 (`stdout`) 设备。在第 7 章，我们会进一步讨论 `log` 包和标准库里其他重要的包。

最后，让我们看看 `main` 函数第 20 行那条语句的作用，如代码清单 2-7 所示。

代码清单 2-7 main.go: 第 19 行到第 20 行

```
19    // 使用特定的项做搜索
20    search.Run("president")
```

可以看到，这一行调用了 `search` 包里的 `Run` 函数。这个函数包含程序的核心业务逻辑，需要传入一个字符串作为搜索项。一旦 `Run` 函数退出，程序就会终止。

现在，让我们看看 `search` 包里的代码。

2.3 search 包

这个程序使用的框架和业务逻辑都在 `search` 包里。这个包由 4 个不同的代码文件组成，每个文件对应一个独立的职责。我们会逐步分析这个程序的逻辑，到时再说明各个代码文件的作用。

由于整个程序都围绕匹配器来运作，我们先简单介绍一下什么是匹配器。这个程序里的匹配器，是指包含特定信息、用于处理某类数据源的实例。在这个示例程序中，有两个匹配器。框架本身实现了一个无法获取任何信息的默认匹配器，而在 `matchers` 包里实现了 `RSS` 匹配器。`RSS` 匹配器知道如何获取、读入并查找 `RSS` 数据源。随后我们会扩展这个程序，加入能读取 `JSON` 文档或 `CSV` 文件的匹配器。我们后面会再讨论如何实现匹配器。

2.3.1 search.go

代码清单 2-8 中展示的是 `search.go` 代码文件的前 9 行代码。之前提到的 `Run` 函数就在这个文件里。

代码清单 2-8 search/search.go: 第 01 行到第 09 行

```
01 package search
02
03 import (
04     "log"
05     "sync"
06 )
07
08 // 注册用于搜索的匹配器的映射
09 var matchers = make(map[string]Matcher)
```

可以看到，每个代码文件都以 `package` 关键字开头，随后跟着包的名字。文件夹 `search` 下的每个代码文件都使用 `search` 作为包名。第 03 行到第 06 行代码导入标准库的 `log` 和 `sync` 包。

与第三方包不同，从标准库中导入代码时，只需要给出要导入的包名。编译器查找包的时候，总是会到 `GOROOT` 和 `GOPATH` 环境变量（如代码清单 2-9 所示）引用的位置去查找。

代码清单 2-9 GOROOT 和 GOPATH 环境变量

```
GOROOT="/Users/me/go"  
GOPATH="/Users/me/spaces/go/projects"
```

log 包提供打印日志信息到标准输出 (stdout)、标准错误 (stderr) 或者自定义设备的功能。sync 包提供同步 goroutine 的功能。这个示例程序需要用到同步功能。第 09 行是全书第一次声明一个变量, 如代码清单 2-10 所示。

代码清单 2-10 search/search.go: 第 08 行到第 09 行

```
08 // 注册用于搜索的匹配器的映射  
09 var matchers = make(map[string]Matcher)
```

这个变量没有定义在任何函数作用域内, 所以会被当成包级变量。这个变量使用关键字 var 声明, 而且声明为 Matcher 类型的映射 (map), 这个映射以 string 类型值作为键, Matcher 类型值作为映射后的值。Matcher 类型在代码文件 matcher.go 中声明, 后面再讲这个类型的用途。这个变量声明还有一个地方要强调一下: 变量名 matchers 是以小写字母开头的。

在 Go 语言里, 标识符要么从包里公开, 要么不从包里公开。当代码导入了一个包时, 程序可以直接访问这个包中任意一个公开的标识符。这些标识符以大写字母开头。以小写字母开头的标识符是不公开的, 不能被其他包中的代码直接访问。但是, 其他包可以间接访问不公开的标识符。例如, 一个函数可以返回一个未公开类型的值, 那么这个函数的任何调用者, 哪怕调用者不是在这个包里声明的, 都可以访问这个值。

这行变量声明还使用赋值运算符和特殊的  make 初始化了变量, 如代码清单 2-11 所示。

代码清单 2-11 构建一个映射

```
make(map[string]Matcher)
```

map 是 Go 语言里的一个引用类型, 需要使用 make 来构造。如果不先构造 map 并将构造后的值赋值给变量, 会在试图使用这个 map 变量时收到出错信息。这是因为 map 变量默认的零值是 nil。在第 4 章我们会进一步了解关于映射的细节。

在 Go 语言中, 所有变量都被初始化为其零值。对于数值类型, 零值是 0; 对于字符串类型, 零值是空字符串; 对于布尔类型, 零值是 false; 对于指针, 零值是 nil。对于引用类型来说, 所引用的底层数据结构会被初始化为对应的零值。但是被声明为其零值的引用类型的变量, 会返回 nil 作为其值。

现在, 让我们看看之前在 main 函数中调用的 Run 函数的内容, 如代码清单 2-12 所示。

代码清单 2-12 search/search.go: 第 11 行到第 57 行

```
11 // Run 执行搜索逻辑  
12 func Run(searchTerm string) {  
13     // 获取需要搜索的数据源列表  
14     feeds, err := RetrieveFeeds()
```

```

15     if err != nil {
16         log.Fatal(err)
17     }
18
19     // 创建一个无缓冲的通道，接收匹配后的结果
20     results := make(chan *Result)
21
22     // 构造一个 waitGroup，以便处理所有的数据源
23     var waitGroup sync.WaitGroup
24
25     // 设置需要等待处理
26     // 每个数据源的 goroutine 的数量
27     waitGroup.Add(len(feeds))
28
29     // 为每个数据源启动一个 goroutine 来查找结果
30     for _, feed := range feeds {
31         // 获取一个匹配器用于查找
32         matcher, exists := matchers[feed.Type]
33         if !exists {
34             matcher = matchers["default"]
35         }
36
37         // 启动一个 goroutine 来执行搜索
38         go func(matcher Matcher, feed *Feed) {
39             Match(matcher, feed, searchTerm, results)
40             waitGroup.Done()
41         }(matcher, feed)
42     }
43
44     // 启动一个 goroutine 来监控是否所有的工作都做完了
45     go func() {
46         // 等候所有任务完成
47         waitGroup.Wait()
48
49         // 用关闭通道的方式，通知 Display 函数
50         // 可以退出程序了
51         close(results)
52     }()
53
54     // 启动函数，显示返回的结果，并且
55     // 在最后一个结果显示完后返回
56     Display(results)
57 }

```

Run 函数包括了这个程序最主要的控制逻辑。这段代码很好地展示了如何组织 Go 程序的代码，以便正确地并发启动和同步 goroutine。先来一步一步考察整个逻辑，再考察每步实现代码的细节。

先来看看 Run 函数是怎么定义的，如代码清单 2-13 所示。

代码清单 2-13 search/search.go: 第 11 行到第 12 行

```

11 // Run 执行搜索逻辑
12 func Run(searchTerm string) {

```

Go 语言使用关键字 `func` 声明函数，关键字后面紧跟着函数名、参数以及返回值。对于 `Run` 这个函数来说，只有一个参数，是 `string` 类型的，名叫 `searchTerm`。这个参数是 `Run` 函数要搜索的搜索项，如果回头看看 `main` 函数（如代码清单 2-14 所示），可以看到如何传递这个搜索项。

代码清单 2-14 `main.go`：第 17 行到第 21 行

```
17 // main 是整个程序的入口
18 func main() {
19     // 使用特定的项做搜索
20     search.Run("president")
21 }
```

`Run` 函数做的第一件事情就是获取数据源 `feeds` 列表。这些数据源从互联网上抓取数据，之后对数据使用特定的搜索项进行匹配，如代码清单 2-15 所示。

代码清单 2-15 `search/search.go`：第 13 行到第 17 行

```
13 // 获取需要搜索的数据源列表
14 feeds, err := RetrieveFeeds()
15 if err != nil {
16     log.Fatal(err)
17 }
```

这里有几个值得注意的重要概念。第 14 行调用了 `search` 包的 `RetrieveFeeds` 函数。这个函数返回两个值。第一个返回值是一组 `Feed` 类型的切片。切片是一种实现了一个动态数组的引用类型。在 Go 语言里可以用切片来操作一组数据。第 4 章会进一步深入了解有关切片的细节。

第二个返回值是一个错误值。在第 15 行，检查返回的值是不是真的是一个错误。如果真的发生错误了，就会调用 `log` 包里的 `Fatal` 函数。`Fatal` 函数接受这个错误的值，并将这个错误在终端窗口里输出，随后终止程序。

不仅仅是 Go 语言，很多语言都允许一个函数返回多个值。一般会像 `RetrieveFeeds` 函数这样声明一个函数返回一个值和一个错误值。如果发生了错误，永远不要使用该函数返回的另一个值^①。这时必须忽略另一个值，否则程序会产生更多的错误，甚至崩溃。

让我们仔细看看从函数返回的值是如何赋值给变量的，如代码清单 2-16 所示。

代码清单 2-16 `search/search.go`：第 13 行到第 14 行

```
13 // 获取需要搜索的数据源列表
14 feeds, err := RetrieveFeeds()
```

这里可以看到简化变量声明运算符 (`:=`)。这个运算符用于声明一个变量，同时给这个变量

① 这个说法并不严格成立，Go 标准库中的 `io.Reader.Read` 方法就允许同时返回数据和错误。但是，如果是自己实现的函数，要尽量遵守这个原则，保持含义足够明确。——译者注

赋予初始值。编译器使用函数返回值的类型来确定每个变量的类型。简化变量声明运算符只是一种简化记法，让代码可读性更高。这个运算符声明的变量和其他使用关键字 `var` 声明的变量没有任何区别。

现在我们得到了数据源列表，进入到后面的代码，如代码清单 2-17 所示。

代码清单 2-17 search/search.go: 第 19 行到第 20 行

```
19 // 创建一个无缓冲的通道，接收匹配后的结果
20 results := make(chan *Result)
```

在第 20 行，我们使用内置的 `make` 函数创建了一个无缓冲的通道。我们使用简化变量声明运算符，在调用 `make` 的同时声明并初始化该通道变量。根据经验，如果需要声明初始值为零值的变量，应该使用 `var` 关键字声明变量；如果提供确切的非零值初始化变量或者使用函数返回值创建变量，应该使用简化变量声明运算符。

在 Go 语言中，通道（channel）和映射（map）与切片（slice）一样，也是引用类型，不过通道本身实现的是一组带类型的值，这组值用于在 goroutine 之间传递数据。通道内置同步机制，从而保证通信安全。在第 6 章中，我们会介绍更多关于通道和 goroutine 的细节。

之后两行是为了防止程序在全部搜索执行完之前终止，如代码清单 2-18 所示。

代码清单 2-18 search/search.go: 第 22 行到第 27 行

```
22 // 构造一个 wait group，以便处理所有的数据源
23 var waitGroup sync.WaitGroup
24
25 // 设置需要等待处理
26 // 每个数据源的 goroutine 的数量
27 waitGroup.Add(len(feeds))
```

在 Go 语言中，如果 main 函数返回，整个程序也就终止了。Go 程序终止时，还会关闭所有之前启动且还在运行的 goroutine。写并发程序的时候，最佳做法是，在 main 函数返回前，清理并终止所有之前启动的 goroutine。编写启动和终止时的状态都很清晰的程序，有助减少 bug，防止资源异常。

这个程序使用 `sync` 包的 `WaitGroup` 跟踪所有启动的 goroutine。非常推荐使用 `WaitGroup` 来跟踪 goroutine 的工作是否完成。`WaitGroup` 是一个计数信号量，我们可以利用它来统计所有的 goroutine 是不是都完成了工作。

在第 23 行我们声明了一个 `sync` 包里的 `WaitGroup` 类型的变量。之后在第 27 行，我们将 `WaitGroup` 变量的值设置为将要启动的 goroutine 的数量。马上就能看到，我们为每个数据源都启动了一个 goroutine 来处理数据。每个 goroutine 完成其工作后，就会递减 `WaitGroup` 变量的计数值，当这个值递减到 0 时，我们就知道所有的工作都做完了。

现在让我们来看看为每个数据源启动 goroutine 的代码，如代码清单 2-19 所示。

代码清单 2-19 search/search.go: 第 29 行到第 42 行

```

29    // 为每个数据源启动一个 goroutine 来查找结果
30    for _, feed := range feeds {
31        // 获取一个匹配器用于查找
32        matcher, exists := matchers[feed.Type]
33        if !exists {
34            matcher = matchers["default"]
35        }
36
37        // 启动一个 goroutine 来执行搜索
38        go func(matcher Matcher, feed *Feed) {
39            Match(matcher, feed, searchTerm, results)
40            waitGroup.Done()
41        }(matcher, feed)
42    }

```

第 30 行到第 42 行迭代之前获得的 feeds，为每个 feed 启动一个 goroutine。我们使用关键字 `for range` 对 feeds 切片做迭代。关键字 `range` 可以用于迭代数组、字符串、切片、映射和通道。使用 `for range` 迭代切片时，每次迭代会返回两个值。第一个值是迭代的元素在切片里的索引位置，第二个值是元素值的一个副本。

如果仔细看一下第 30 行的 `for range` 语句，会发现再次使用了下划线标识符，如代码清单 2-20 所示。

代码清单 2-20 search/search.go: 第 29 行到第 30 行

```

29    // 为每个数据源启动一个 goroutine 来查找结果
30    for _, feed := range feeds {

```

这是第二次看到使用了下划线标识符。第一次是在 `main.go` 里导入 `matchers` 包的时候。这次，下划线标识符的作用是占位符，占据了保存 `range` 调用返回的索引值的变量的位置。如果要调用的函数返回多个值，而又不需要其中的某个值，就可以使用下划线标识符将其忽略。在我们的例子里，我们不需要使用返回的索引值，所以就使用下划线标识符把它忽略掉。

在循环中，我们首先通过 `map` 查找到一个可用于处理特定数据源类型的数据的 `Matcher` 值，如代码清单 2-21 所示。

代码清单 2-21 search/search.go: 第 31 行到第 35 行

```

31        // 获取一个匹配器用于查找
32        matcher, exists := matchers[feed.Type]
33        if !exists {
34            matcher = matchers["default"]
35        }

```

我们还没有说过 `map` 里面的值是如何获得的。一会儿就会在程序初始化的时候看到如何设置 `map` 里的值。在第 32 行，我们检查 `map` 是否含有符合数据源类型的值。查找 `map` 里的键时，

有两个选择：要么赋值给一个变量，要么为了精确查找，赋值给两个变量。赋值给两个变量时第一个值和赋值给一个变量时的值一样，是 `map` 查找的结果值。如果指定了第二个值，就会返回一个布尔标志，来表示查找的键是否存在于 `map` 里。如果这个键不存在，`map` 会返回其值类型的零值作为返回值，如果这个键存在，`map` 会返回键所对应值的副本。

在第 33 行，我们检查这个键是否存在于 `map` 里。如果不存在，使用默认匹配器。这样程序在不知道对应数据源的具体类型时，也可以执行，而不会中断。之后，启动一个 `goroutine` 来执行搜索，如代码清单 2-22 所示。

代码清单 2-22 search/search.go: 第 37 行到第 41 行

```
37      // 启动一个 goroutine 来执行搜索
38      go func(matcher Matcher, feed *Feed) {
39          Match(matcher, feed, searchTerm, results)
40          waitGroup.Done()
41      }(matcher, feed)
```

我们会在第 6 章进一步学习 `goroutine`，现在只要知道，一个 `goroutine` 是一个独立于其他函数运行的函数。使用关键字 `go` 启动一个 `goroutine`，并对这个 `goroutine` 做并发调度。在第 38 行，我们使用关键字 `go` 启动了一个匿名函数作为 `goroutine`。匿名函数是指没有明确声明名字的函数。在 `for range` 循环里，我们为每个数据源，以 `goroutine` 的方式启动了一个匿名函数。这样可以并发地独立处理每个数据源的数据。

匿名函数也可以接受声明时指定的参数。在第 38 行，我们指定匿名函数要接受两个参数，一个类型为 `Matcher`，另一个是指向一个 `Feed` 类型值的指针。这意味着变量 `feed` 是一个指针变量。指针变量可以方便地在函数之间共享数据。使用指针变量可以让函数访问并修改一个变量的状态，而这个变量可以在其他函数甚至是其他 `goroutine` 的作用域里声明。

在第 41 行，`matcher` 和 `feed` 两个变量的值被传入匿名函数。在 Go 语言中，所有的变量都以值的方式传递。因为指针变量的值是所指向的内存地址，在函数间传递指针变量，是在传递这个地址值，所以依旧被看作以值的方式在传递。

在第 39 行到第 40 行，可以看到每个 `goroutine` 是如何工作的，如代码清单 2-23 所示。

代码清单 2-23 search/search.go: 第 39 行到第 40 行

```
39      Match(matcher, feed, searchTerm, results)
40      waitGroup.Done()
```

`goroutine` 做的第一件事是调用一个叫 `Match` 的函数，这个函数可以在 `match.go` 文件里找到。`Match` 函数的参数是一个 `Matcher` 类型的值、一个指向 `Feed` 类型值的指针、搜索项以及输出结果的通道。我们一会儿再看这个函数的内部细节，现在只要知道，`Match` 函数会搜索数据源的数据，并将匹配结果输出到 `results` 通道。

一旦 `Match` 函数调用完毕，就会执行第 40 行的代码，递减 `WaitGroup` 的计数。一旦每个 `goroutine` 都执行调用 `Match` 函数和 `Done` 方法，程序就知道每个数据源都处理完成。调用 `Done`

方法这一行还有一个值得注意的细节：WaitGroup 的值没有作为参数传入匿名函数，但是匿名函数依旧访问到了这个值。

Go 语言支持闭包，这里就应用了闭包。实际上，在匿名函数内访问 searchTerm 和 results 变量，也是通过闭包的形式访问的。因为有了闭包，函数可以直接访问到那些没有作为参数传入的变量。匿名函数并没有拿到这些变量的副本，而是直接访问外层函数作用域中声明的这些变量本身。因为 matcher 和 feed 变量每次调用时值不相同，所以并没有使用闭包的方式访问这两个变量，如代码清单 2-24 所示。

代码清单 2-24 search/search.go: 第 29 行到第 32 行

```
29    // 为每个数据源启动一个 goroutine 来查找结果
30    for _, feed := range feeds {
31        // 获取一个匹配器用于查找
32        matcher, exists := matchers[feed.Type]
```

可以看到，在第 30 行到第 32 行，变量 feed 和 matcher 的值会随着循环的迭代而改变。如果我们使用闭包访问这些变量，随着外层函数里变量值的改变，内层的匿名函数也会感知到这些改变。所有的 goroutine 都会因为闭包共享同样的变量。除非我们以函数参数的形式传值给函数，否则绝大部分 goroutine 最终都会使用同一个 matcher 来处理同一个 feed——这个值很有可能是 feeds 切片的最后一个值。

随着每个 goroutine 搜索工作的运行，将结果发送到 results 通道，并递减 waitGroup 的计数，我们需要一种方法来显示所有的结果，并让 main 函数持续工作，直到完成所有的操作，如代码清单 2-25 所示。

代码清单 2-25 search/search.go: 第 44 行到第 57 行

```
44    // 启动一个 goroutine 来监控是否所有的工作都做完了
45    go func() {
46        // 等候所有任务完成
47        waitGroup.Wait()
48
49        // 用关闭通道的方式，通知 Display 函数
50        // 可以退出程序了
51        close(results)
52    }()
53
54    // 启动函数，显示返回的结果，
55    // 并且在最后一个结果显示完后返回
56    Display(results)
57 }
```

第 45 行到第 56 行的代码解释起来比较麻烦，等我们看完 search 包里的其他代码后再来解释。我们现在只解释表面的语法，随后再来解释底层的机制。在第 45 行到第 52 行，我们以 goroutine 的方式启动了另一个匿名函数。这个匿名函数没有输入参数，使用闭包访问了 WaitGroup 和

results 变量。这个 goroutine 里面调用了 WaitGroup 的 Wait 方法。这个方法会导致 goroutine 阻塞，直到 WaitGroup 内部的计数到达 0。之后，goroutine 调用了内置的 close 函数，关闭了通道，最终导致程序终止。

Run 函数的最后一段代码是第 56 行。这行调用了 match.go 文件里的 Display 函数。一旦这个函数返回，程序就会终止。而之前的代码保证了所有 results 通道里的数据被处理之前，Display 函数不会返回。

2.3.2 feed.go

现在已经看过了 Run 函数，让我们继续看看 search.go 文件的第 14 行中的 RetrieveFeeds 函数调用背后的代码。这个函数读取 data.json 文件并返回数据源的切片。这些数据源会输出内容，随后使用各自的匹配器进行搜索。代码清单 2-26 给出的是 feed.go 文件的前 8 行代码。

代码清单 2-26 feed.go: 第 01 行到第 08 行

```
01 package search
02
03 import (
04     "encoding/json"
05     "os"
06 )
07
08 const dataFile = "data/data.json"
```

这个代码文件在 search 文件夹里，所以第 01 行声明了包的名字为 search。第 03 行到第 06 行导入了标准库中的两个包。json 包提供编解码 JSON 的功能，os 包提供访问操作系统的功能，如读文件。

读者可能注意到了，导入 json 包的时候需要指定 encoding 路径。不考虑这个路径的话，我们导入包的名字叫作 json。不管标准库的路径是什么样的，并不会改变包名。我们在访问 json 包内的函数时，依旧是指定 json 这个名字。

在第 08 行，我们声明了一个叫作 dataFile 的常量，使用内容是磁盘上根据相对路径指定的数据文件名的字符串做初始化。因为 Go 编译器可以根据赋值运算符右边的值来推导类型，声明常量的时候不需要指定类型。此外，这个常量的名称使用小写字母开头，表示它只能在 search 包内的代码里直接访问，而不暴露到包外面。

接着我们来看看 data.json 数据文件的部分内容，如代码清单 2-27 所示。

代码清单 2-27 data.json

```
[
  {
    "site" : "npr",
    "link" : "http://www.npr.org/rss/rss.php?id=1001",
    "type" : "rss"
```

```

    },
    {
        "site" : "cnn",
        "link" : "http://rss.cnn.com/rss/cnn_world.rss",
        "type" : "rss"
    },
    {
        "site" : "foxnews",
        "link" : "http://feeds.foxnews.com/foxnews/world?format=xml",
        "type" : "rss"
    },
    {
        "site" : "nbcnews",
        "link" : "http://feeds.nbcnews.com/feeds/topstories",
        "type" : "rss"
    }
]

```

为了保证数据的有效性，代码清单 2-27 只选用了 4 个数据源，实际数据文件包含的数据要比这 4 个多。数据文件包括一个 JSON 文档数组。数组的每一项都是一个 JSON 文档，包含获取数据的网站名、数据的链接以及我们期望获得的数据类型。

这些数据文档需要解码到一个结构组成的切片里，以便我们能在程序里使用这些数据。来看看用于解码数据文档的结构类型，如代码清单 2-28 所示。

代码清单 2-28 feed.go: 第 10 行到第 15 行

```

10 // Feed 包含我们需要处理的数据源的信息
11 type Feed struct {
12     Name string `json:"site"`
13     URI  string `json:"link"`
14     Type string `json:"type"`
15 }

```

在第 11 行到第 15 行，我们声明了一个名叫 Feed 的结构类型。这个类型会对外暴露。这个类型里面声明了 3 个字段，每个字段的类型都是字符串，对应于数据文件中各个文档的不同字段。每个字段的声明最后 ` ` 引号里的部分被称作标记 (tag)。这个标记里描述了 JSON 解码的元数据，用于创建 Feed 类型值的切片。每个标记将结构类型里字段对应到 JSON 文档里指定名字的字段。

现在可以看看 search.go 代码文件的第 14 行中调用的 RetrieveFeeds 函数了。这个函数读取数据文件，并将每个 JSON 文档解码，存入一个 Feed 类型值的切片里，如代码清单 2-29 所示。

代码清单 2-29 feed.go: 第 17 行到第 36 行

```

17 // RetrieveFeeds 读取并反序列化源数据文件
18 func RetrieveFeeds() ([]*Feed, error) {
19     // 打开文件
20     file, err := os.Open(dataFile)
21     if err != nil {
22         return nil, err
23     }

```

```

24
25     // 当函数返回时
26     // 关闭文件
27     defer file.Close()
28
29     // 将文件解码到一个切片里
30     // 这个切片的每一项是一个指向一个 Feed 类型值的指针
31     var feeds []*Feed
32     err = json.NewDecoder(file).Decode(&feeds)
33
34     // 这个函数不需要检查错误，调用者会做这件事
35     return feeds, err
36 }

```

让我们从第 18 行的函数声明开始。这个函数没有参数，会返回两个值。第一个返回值是一个切片，其中每一项指向一个 `Feed` 类型的值。第二个返回值是一个 `error` 类型的值，用来表示函数是否调用成功。在这个代码示例里，会经常看到返回 `error` 类型值来表示函数是否调用成功。这种用法在标准库里也很常见。

现在让我们看看第 20 行到第 23 行。在这几行里，我们使用 `os` 包打开了数据文件。我们使用相对路径调用 `Open` 方法，并得到两个返回值。第一个返回值是一个指针，指向 `File` 类型的值，第二个返回值是 `error` 类型的值，检查 `Open` 调用是否成功。紧接着第 21 行就检查了返回的 `error` 类型错误值，如果打开文件真的有问题，就把这个错误值返回给调用者。

如果成功打开了文件，会进入到第 27 行。这里使用了关键字 `defer`，如代码清单 2-30 所示。

代码清单 2-30 feed.go: 第 25 行到第 27 行

```

25     // 当函数返回时
26     // 关闭文件
27     defer file.Close()

```

关键字 `defer` 会安排随后的函数调用在函数返回时才执行。在使用完文件后，需要主动关闭文件。使用关键字 `defer` 来安排调用 `Close` 方法，可以保证这个函数一定会被调用。哪怕函数意外崩溃终止，也能保证关键字 `defer` 安排调用的函数会被执行。关键字 `defer` 可以缩短打开文件和关闭文件之间间隔的代码行数，有助于提高代码可读性，减少错误。

现在可以看看这个函数的最后几行，如代码清单 2-31 所示。先来看一下第 31 行到第 35 行的代码。

代码清单 2-31 feed.go: 第 29 行到第 36 行

```

29     // 将文件解码到一个切片里
30     // 这个切片的每一项是一个指向一个 Feed 类型值的指针
31     var feeds []*Feed
32     err = json.NewDecoder(file).Decode(&feeds)
33
34     // 这个函数不需要检查错误，调用者会做这件事
35     return feeds, err
36 }

```


在第 31 行我们声明了一个名字叫 `feeds`，值为 `nil` 的切片，这个切片包含一组指向 `Feed` 类型值的指针。之后在第 32 行我们调用 `json` 包的 `NewDecoder` 函数，然后在其返回值上调用 `Decode` 方法。我们使用之前调用 `Open` 返回的文件句柄调用 `NewDecoder` 函数，并得到一个指向 `Decoder` 类型的值的指针。之后再调用这个指针的 `Decode` 方法，传入切片的地址。之后 `Decode` 方法会解码数据文件，并将解码后的值以 `Feed` 类型值的形式存入切片里。

根据 `Decode` 方法的声明，该方法可以接受任何类型的值，如代码清单 2-32 所示。

代码清单 2-32 使用空 interface

```
func (dec *Decoder) Decode(v interface{}) error
```

`Decode` 方法接受一个类型为 `interface{}` 的数。这个类型在 Go 语言里很特殊，一般会配合 `reflect` 包里提供的反射功能一起使用。

最后，第 35 行给函数的调用者返回了切片和 。在这个例子里，不需要对 `Decode` 调用之后的错误做检查。函数执行结束，这个函数的调用者可以检查这个错误值，并决定后续如何处理。

现在让我们看看搜索的代码是如何支持不同类型的数据源的。让我们去看看匹配器的代码。

2.3.3 match.go/default.go

`match.go` 代码文件包含创建不同类型匹配器的代码，这些匹配器用于在 `Run` 函数里对数据进行搜索。让我们回头看看 `Run` 函数里使用不同匹配器执行搜索的代码，如代码清单 2-33 所示。

代码清单 2-33 search/search.go: 第 29 行到第 42 行

```
29 // 为每个数据源启动一个 goroutine 来查找结果
30 for _, feed := range feeds {
31     // 获取一个匹配器用于查找
32     matcher, exists := matchers[feed.Type]
33     if !exists {
34         matcher = matchers["default"]
35     }
36
37     // 启动一个 goroutine 执行查找
38     go func(matcher Matcher, feed *Feed) {
39         Match(matcher, feed, searchTerm, results)
40         waitGroup.Done()
41     }(matcher, feed)
42 }
```

代码的第 32 行，根据数据源类型查找一个匹配器值。这个匹配器值随后会用于在特定的数据源里处理搜索。之后在第 38 行到第 41 行启动了一个 `goroutine`，让匹配器对数据源的数据进行搜索。让这段代码起作用的关键是这个架构使用一个接口类型来匹配并执行具有特定实现的匹配器。这样，就能使用这段代码，以一致且通用的方法，来处理不同类型的匹配器值。让我们看一

下 `match.go` 里的代码，看看如何才能实现这一功能。

代码清单 2-34 给出的是 `match.go` 的前 17 行代码。

代码清单 2-34 search/match.go: 第 01 行到第 17 行

```
01 package search
02
03 import (
04     "log"
05 )
06
07 // Result 保存搜索的结果
08 type Result struct {
09     Field   string
10     Content string
11 }
12
13 // Matcher 定义了要实现的
14 // 新搜索类型的行为
15 type Matcher interface {
16     Search(feed *Feed, searchTerm string) ([]*Result, error)
17 }
```

让我们看一下第 15 行到第 17 行，这里声明了一个名为 `Matcher` 的接口类型。之前，我们只见过声明结构类型，而现在看到如何声明一个 `interface`（接口）类型。我们会在第 5 章介绍接口的更多细节，现在只需要知道，`interface` 关键字声明了一个接口，这个接口声明了结构类型或者具名类型需要实现的行为。一个接口的行为最终由在这个接口类型中声明的方法决定。

对于 `Matcher` 这个接口来说，只声明了一个 `Search` 方法，这个方法输入一个指向 `Feed` 类型值的指针和一个 `string` 类型的搜索项。这个方法返回两个值：一个指向 `Result` 类型值的指针的切片，另一个是错误值。`Result` 类型的声明在第 08 行到第 11 行。

命名接口的时候，也需要遵守 Go 语言的命名惯例。如果接口类型只包含一个方法，那么这个类型的名字以 `er` 结尾。我们的例子里就是这么做的，所以这个接口的名字叫作 `Matcher`。如果接口类型内部声明了多个方法，其名字需要与其行为关联。

如果能让一个用户定义的类型实现一个接口，这个用户定义的类型要实现接口类型里声明的所有方法。让我们切换到 `default.go` 代码文件，看看默认匹配器是如何实现 `Matcher` 接口的，如代码清单 2-35 所示。

代码清单 2-35 search/default.go: 第 01 行到第 15 行

```
01 package search
02
03 // defaultMatcher 实现了默认匹配器
04 type defaultMatcher struct{}
05
06 // init 函数将默认匹配器注册到程序里
07 func init() {
```



```

08     var matcher defaultMatcher
09     Register("default", matcher)
10 }
11
12 // Search 实现了默认匹配器的行为
13 func (m defaultMatcher) Search(feed *Feed, searchTerm string) ([]*Result, error) {
14     return nil, nil
15 }

```

在第 04 行，我们使用一个空结构声明了一个名叫 `defaultMatcher` 的结构类型。空结构在创建实例时，不会分配任何内存。这种结构很适合创建没有任何状态的类型。对于默认匹配器来说，不需要维护任何状态，所以我们只要实现对应的接口就行。

在第 13 行到第 15 行，可以看到 `defaultMatcher` 类型实现 `Matcher` 接口的代码。实现接口的方法 `Search` 只返回两个 `nil` 值。其他的实现，如 `RSS` 匹配器的实现，会在这个方法里使用特定的业务逻辑规则来处理搜索。

`Search` 方法的声明也声明了 `defaultMatcher` 类型的值的接收者，如代码清单 2-36 所示。

代码清单 2-36 search/default.go: 第 13 行

```

13 func (m defaultMatcher) Search

```

如果声明函数的时候带有接收者，则意味着声明了一个方法。这个方法会和指定的接收者的类型绑在一起。在我们的例子里，`Search` 方法与 `defaultMatcher` 类型的值绑在一起。这意味着我们可以使用 `defaultMatcher` 类型的值或者指向这个类型值的指针来调用 `Search` 方法。无论我们是使用接收者类型的值来调用这个方法，还是使用接收者类型值的指针来调用这个方法，编译器都会正确地引用或者解引用对应的值，作为接收者传递给 `Search` 方法，如代码清单 2-37 所示。

代码清单 2-37 调用方法的例子

```

// 方法声明为使用 defaultMatcher 类型的值作为接收者
func (m defaultMatcher) Search(feed *Feed, searchTerm string)

// 声明一个指向 defaultMatcher 类型值的指针
dm := new(defaultMatch)

// 编译器会解开 dm 指针的引用，使用对应的值调用方法
dm.Search(feed, "test")

// 方法声明为使用指向 defaultMatcher 类型值的指针作为接收者
func (m *defaultMatcher) Search(feed *Feed, searchTerm string)

// 声明一个 defaultMatcher 类型的值
var dm defaultMatch

// 编译器会自动生成指针引用 dm 值，使用指针调用方法
dm.Search(feed, "test")

```

因为大部分方法在被调用后都需要维护接收者的值的状态，所以，一个最佳实践是，将方法的接收者声明为指针。对于 `defaultMatcher` 类型来说，使用值作为接收者是因为创建一个 `defaultMatcher` 类型的值不需要分配内存。由于 `defaultMatcher` 不需要维护状态，所以不需要指针形式的接收者。

与直接通过值或者指针调用方法不同，如果通过接口类型的值调用方法，规则有很大不同，如代码清单 2-38 所示。使用指针作为接收者声明的方法，只能在接口类型的值是一个指针的时候被调用。使用值作为接收者声明的方法，在接口类型的值为值或者指针时，都可以被调用。

代码清单 2-38 接口方法调用所受限制的例子

```
// 方法声明为使用指向 defaultMatcher 类型值的指针作为接收者
func (m *defaultMatcher) Search(feed *Feed, searchTerm string)

// 通过 interface 类型的值来调用方法
var dm defaultMatcher
var matcher Matcher = dm // 将值赋值给接口类型
matcher.Search(feed, "test") // 使用值来调用接口方法

> go build
cannot use dm (type defaultMatcher) as type Matcher in assignment

// 方法声明为使用 defaultMatcher 类型的值作为接收者
func (m defaultMatcher) Search(feed *Feed, searchTerm string)

// 通过 interface 类型的值来调用方法
var dm defaultMatcher
var matcher Matcher = &dm // 将指针赋值给接口类型
matcher.Search(feed, "test") // 使用指针来调用接口方法

> go build
Build Successful
```

除了 `Search` 方法，`defaultMatcher` 类型不需要为实现接口做更多的事情了。从这段代码之后，不论是 `defaultMatcher` 类型的值还是指针，都满足 `Matcher` 接口，都可以作为 `Matcher` 类型的值使用。这是代码可以工作的关键。`defaultMatcher` 类型的值和指针现在还可以作为 `Matcher` 的值，赋值或者传递给接受 `Matcher` 类型值的函数。

让我们看看 `match.go` 代码文件里实现 `Match` 函数的代码，如代码清单 2-39 所示。这个函数在 `search.go` 代码文件的第 39 行中由 `Run` 函数调用。

代码清单 2-39 search/match.go: 第 19 行到第 33 行

```
19 // Match 函数，为每个数据源单独启动 goroutine 来执行这个函数
20 // 并发地执行搜索
21 func Match(matcher Matcher, feed *Feed, searchTerm string, results chan<- *Result) {
22     // 对特定的匹配器执行搜索
23     searchResults, err := matcher.Search(feed, searchTerm)
24     if err != nil {
```

```

25         log.Println(err)
26         return
27     }
28
29     // 将结果写入通道
30     for _, result := range searchResults {
31         results <- result
32     }
33 }

```

这个函数使用实现了 `Matcher` 接口的值或者指针，进行真正的搜索。这个函数接受 `Matcher` 类型的值作为第一个参数。只有实现了 `Matcher` 接口的值或者指针能被接受。因为 `defaultMatcher` 类型使用值作为接收者，实现了这个接口，所以 `defaultMatcher` 类型的值或者指针可以传入这个函数。

在第 23 行，调用了传入函数的 `Matcher` 类型值的 `Search` 方法。这里执行了 `Matcher` 变量中特定的 `Search` 方法。`Search` 方法返回后，在第 24 行检测返回的错误值是否真的是一个错误。如果是一个错误，函数通过 `log` 输出错误信息并返回。如果搜索并没有返回错误，而是返回了搜索结果，则把结果写入通道，以便正在监听通道的 `main` 函数就能收到这些结果。

`match.go` 中的最后一部分代码就是 `main` 函数在第 56 行调用的 `Display` 函数，如代码清单 2-40 所示。这个函数会阻止程序终止，直到接收并输出了搜索 `goroutine` 返回的所有结果。

代码清单 2-40 search/match.go: 第 35 行到第 43 行

```

35 // Display 从每个单独的 goroutine 接收到结果后
36 // 在终端窗口输出
37 func Display(results chan *Result) {
38     // 通道会一直被阻塞，直到有结果写入
39     // 一旦通道被关闭，for 循环就会终止
40     for result := range results {
41         fmt.Printf("%s:\n%s\n\n", result.Field, result.Content)
42     }
43 }

```

当通道被关闭时，通道和关键字 `range` 的行为，使这个函数在处理完所有结果后才会返回。让我们再来简单看一下 `Run` 函数的代码，特别是关闭 `results` 通道并调用 `Display` 函数那段，如代码清单 2-41 所示。

代码清单 2-41 search/search.go: 第 44 行到第 57 行

```

44 // 启动一个 goroutine 来监控是否所有的工作都做完了
45 go func() {
46     // 等候所有任务完成
47     waitGroup.Wait()
48
49     // 用关闭通道的方式，通知 Display 函数
50     // 可以退出程序了
51     close(results)
52 }()

```

```

53
54     // 启动函数，显示返回的结果，
55     // 并且在最后一个结果显示完后返回
56     Display(results)
57 }

```

第 45 行到第 52 行定义的 goroutine 会等待 waitGroup，直到搜索 goroutine 调用了 Done 方法。一旦最后一个搜索 goroutine 调用了 Done，Wait 方法会返回，之后第 51 行的代码会关闭 results 通道。一旦通道关闭，goroutine 就会终止，不再工作。

在 match.go 代码文件的第 30 行到第 32 行，搜索结果会被写入通道，如代码清单 2-42 所示。

代码清单 2-42 search/match.go: 第 29 行到第 32 行

```

29     // 将结果写入通道
30     for _, result := range searchResults {
31         results <- result
32     }

```

如果回头看一看 match.go 代码文件的第 40 行到第 42 行的 for range 循环，如代码清单 2-43 所示，我们就能把写入结果、关闭通道和处理结果这些流程串在一起。

代码清单 2-43 search/match.go: 第 38 行到第 42 行

```

38     // 通道会一直被阻塞，直到有结果写入
39     // 一旦通道被关闭，for 循环就会终止
40     for result := range results {
41         fmt.Printf("%s:\n%s\n\n", result.Field, result.Content)
42     }

```

match.go 代码文件的第 40 行的 for range 循环会一直阻塞，直到有结果写入通道。在某个搜索 goroutine 向通道写入结果后（如在 match.go 代码文件的第 31 行所见），for range 循环被唤醒，读出这些结果。之后，结果会立刻写到日志中。看上去这个 for range 循环会无限循环下去，但其实不然。一旦 search.go 代码文件第 51 行关闭了通道，for range 循环就会终止，Display 函数也会返回。

在我们去看 RSS 匹配器的实现之前，再看一下程序开始执行时，如何初始化不同的匹配器。为此，我们需要先回头看看 default.go 代码文件的第 07 行到第 10 行，如代码清单 2-44 所示。

代码清单 2-44 search/default.go: 第 06 行到第 10 行

```

06 // init 函数将默认匹配器注册到程序里
07 func init() {
08     var matcher defaultMatcher
09     Register("default", matcher)
10 }

```

在代码文件 default.go 里有一个特殊的函数，名叫 init。在 main.go 代码文件里也能看到同名的函数。我们之前说过，程序里所有的 init 方法都会在 main 函数启动前被调用。让我们再

看看 main.go 代码文件导入了哪些代码，如代码清单 2-45 所示。

代码清单 2-45 main.go: 第 07 行到第 08 行

```
07 _ "github.com/goinaction/code/chapter2/sample/matchers"
08 "github.com/goinaction/code/chapter2/sample/search"
```

第 8 行导入 search 包，这让编译器可以找到 default.go 代码文件里的 init 函数。一旦编译器发现 init 函数，它就会给这个函数优先执行的权限，保证其在 main 函数之前被调用。

代码文件 default.go 里的 init 函数执行一个特殊的任务。这个函数会创建一个 defaultMatcher 类型的值，并将这个值传递给 search.go 代码文件里的 Register 函数，如代码清单 2-46 所示。

代码清单 2-46 search/search.go: 第 59 行到第 67 行

```
59 // Register 调用时，会注册一个匹配器，提供给后面的程序使用
60 func Register(feedType string, matcher Matcher) {
61     if _, exists := matchers[feedType]; exists {
62         log.Fatalln(feedType, "Matcher already registered")
63     }
64
65     log.Println("Register", feedType, "matcher")
66     matchers[feedType] = matcher
67 }
```

这个函数的职责是，将一个 Matcher 值加入到保存注册匹配器的映射中。所有这种注册都应该在 main 函数被调用前完成。使用 init 函数可以非常完美地完成这种初始化时注册的任务。

2.4 RSS 匹配器

最后要看的一部分代码是 RSS 匹配器的实现代码。我们之前看到的代码搭建了一个框架，以便能够实现不同的匹配器来搜索内容。RSS 匹配器的结构与默认匹配器的结构很类似。每个匹配器为了匹配接口，Search 方法的实现都不同，因此匹配器之间无法互相替换。

代码清单 2-47 中的 RSS 文档是一个例子。当我们访问数据源列表里 RSS 数据源的链接时，期望获得的数据就和这个例子类似。

代码清单 2-47 期望的 RSS 数据源文档

```
<rss xmlns:npr="http://www.npr.org/rss/" xmlns:nprml="http://api"
  <channel>
    <title>News</title>
    <link>...</link>
    <description>...</description>

    <language>en</language>
```

```

<copyright>Copyright 2014 NPR - For Personal Use
<image>...</image>
<item>
  <title>
    Putin Says He'll Respect Ukraine Vote But U.S.
  </title>
  <description>
    The White House and State Department have called on the
  </description>

```

如果用浏览器打开代码清单 2-47 中的任意一个链接，就能看到期望的 RSS 文档的完整内容。RSS 匹配器的实现会下载这些 RSS 文档，使用搜索项来搜索标题和描述域，并将结果发送给 results 通道。让我们先看看 rss.go 代码文件的前 12 行代码，如代码清单 2-48 所示。

代码清单 2-48 matchers/rss.go: 第 01 行到第 12 行

```

01 package matchers
02
03 import (
04     "encoding/xml"
05     "errors"
06     "fmt"
07     "log"
08     "net/http"
09     "regexp"
10
11     "github.com/goinaction/code/chapter2/sample/search"
12 )

```

和其他代码文件一样，第 1 行定义了包名。这个代码文件处于名叫 matchers 的文件夹中，所以包名也叫 matchers。之后，我们从标准库中导入了 6 个库，还导入了 search 包。再一次，我们看到有些标准库的包是从标准库所在的子文件夹导入的，如 xml 和 http。就像 json 包一样，路径里最后一个文件夹的名字代表包的名字。

为了让程序可以使用文档里的数据，解码 RSS 文档的时候需要用到 4 个结构类型，如代码清单 2-49 所示。

代码清单 2-49 matchers/rss.go: 第 14 行到第 58 行

```

14 type (
15     // item 根据 item 字段的标签，将定义的字
16     // 与 rss 文档的字段关联起来
17     item struct {
18         XMLName      xml.Name `xml:"item"`
19         PubDate       string  `xml:"pubDate"`
20         Title          string  `xml:"title"`
21         Description    string  `xml:"description"`
22         Link           string  `xml:"link"`
23         GUID           string  `xml:"guid"`
24         GeoRssPoint    string  `xml:"georss:point"`
25     }

```

```

26
27 // image 根据 image 字段的标签，将定义的字段
28 // 与 rss 文档的字段关联起来
29 image struct {
30     XMLName xml.Name `xml:"image"`
31     URL      string   `xml:"url"`
32     Title    string   `xml:"title"`
33     Link     string   `xml:"link"`
34 }
35
36 // channel 根据 channel 字段的标签，将定义的字段
37 // 与 rss 文档的字段关联起来
38 channel struct {
39     XMLName      xml.Name `xml:"channel"`
40     Title        string   `xml:"title"`
41     Description  string   `xml:"description"`
42     Link         string   `xml:"link"`
43     PubDate      string   `xml:"pubDate"`
44     LastBuildDate string `xml:"lastBuildDate"`
45     TTL          string   `xml:"ttl"`
46     Language     string   `xml:"language"`
47     ManagingEditor string `xml:"managingEditor"`
48     WebMaster    string   `xml:"webMaster"`
49     Image        image    `xml:"image"`
50     Item         []item   `xml:"item"`
51 }
52
53 // rssDocument 定义了与 rss 文档关联的字段
54 rssDocument struct {
55     XMLName xml.Name `xml:"rss"`
56     Channel channel   `xml:"channel"`
57 }
58 )

```

如果把这些结构与任意一个数据源的 RSS 文档对比，就能发现它们的对应关系。解码 XML 的方法与我们在 feed.go 代码文件里解码 JSON 文档一样。接下来我们可以看看 rssMatcher 类型的声明，如代码清单 2-50 所示。

代码清单 2-50 matchers/rss.go: 第 60 行到第 61 行

```

60 // rssMatcher 实现了 Matcher 接口
61 type rssMatcher struct{}

```

再说明一次，这个声明与 defaultMatcher 类型的声明很像。因为不需要维护任何状态，所以我们使用了一个空结构来实现 Matcher 接口。接下来看看匹配器 init 函数的实现，如代码清单 2-51 所示。

代码清单 2-51 matchers/rss.go: 第 63 行到第 67 行

```

63 // init 将匹配器注册到程序里
64 func init() {
65     var matcher rssMatcher

```

```

66     search.Register("rss", matcher)
67 }

```

就像在默认匹配器里看到的一样，`init` 函数将 `rssMatcher` 类型的值注册到程序里，以备后用。让我们再看一次 `main.go` 代码文件里的导入部分，如代码清单 2-52 所示。

代码清单 2-52 `main.go`: 第 07 行到第 08 行

```

07     _ "github.com/goinaction/code/chapter2/sample/matchers"
08     "github.com/goinaction/code/chapter2/sample/search"

```

`main.go` 代码文件里的代码并没有直接使用任何 `matchers` 包里的标识符。不过，我们依旧需要编译器安排调用 `rss.go` 代码文件里的 `init` 函数。在第 07 行，我们使用下划线标识符作为别名导入 `matchers` 包，完成了这个调用。这种方法可以让编译器在导入未被引用的包时不报错，而且依旧会定位到包内的 `init` 函数。我们已经看过了所有的导入、类型和初始化函数，现在来看看最后两个用于实现 `Matcher` 接口的方法，如代码清单 2-53 所示。

代码清单 2-53 `matchers/rss.go`: 第 114 行到第 140 行

```

114 // retrieve 发送 HTTP Get 请求获取 rss 数据源并解码
115 func (m rssMatcher) retrieve(feed *search.Feed) (*rssDocument, error) {
116     if feed.URI == "" {
117         return nil, errors.New("No rss feed URI provided")
118     }
119
120     // 从网络获得 rss 数据源文档
121     resp, err := http.Get(feed.URI)
122     if err != nil {
123         return nil, err
124     }
125
126     // 一旦从函数返回，关闭返回的响应链接
127     defer resp.Body.Close()
128
129     // 检查状态码是不是 200，这样就能知道
130     // 是不是收到了正确的响应
131     if resp.StatusCode != 200 {
132         return nil, fmt.Errorf("HTTP Response Error %d\n", resp.StatusCode)
133     }
134
135     // 将 rss 数据源文档解码到我们定义的结构类型里
136     // 不需要检查错误，调用者会做这件事
137     var document rssDocument
138     err = xml.NewDecoder(resp.Body).Decode(&document)
139     return &document, err
140 }

```

方法 `retrieve` 并没有对外暴露，其执行的逻辑是从 `RSS` 数据源的链接拉取 `RSS` 文档。在第 121 行，可以看到调用了 `http` 包的 `Get` 方法。我们会在第 8 章进一步介绍这个包，现在只需要知道，使用 `http` 包，`Go` 语言可以很容易地进行网络请求。当 `Get` 方法返回后，我们可以