# 第 3 章

# Go 模块

欢迎来到第 3 章！现在你已经了解了 Go 语言的基本构建块，你可能认为是时候开始实现更复杂的应用程序了。然而，在此之前，学习如何与较大的程序员社区协同工作是很重要的。要做到这一点，必须学习如何使用和创建包（package），从而能够大规模地部署代码。

阅读本章后，可以得到以下问题的答案。

- 什么是包？为什么它们对编程语言的生态系统很重要？
- 包管理器是如何工作的？它在开发大型应用程序中起到了什么作用？
- Go 模块是如何构建的？
- 如何在应用程序中使用内置和第三方的模块？
- 如何构建自己的 Go 模块？

大多数编程语言都使用包管理系统对代码进行部署，如果使用得当，那么包管理系统能够使程序员的工作更高效且更具有协作性。例如，Python 的 PIP、Swift 的包管理器、Ruby 的 RubyGems，当然，还有 Go 的 Go modules。

此外，当一个令人称赞的包生态系统（无论是内置的还是第三方社区支持的）具有独特的语言特性（如 Go 独特的并发性和快速编译特性）时，编程就会变得简单而有趣。

编程语言实际上是由其提供的功能“包”定义的。以 Python 为例，Python 通常通过大量经过良好优化的内置包来扩展其标准库，例如，json、urllib、pickle、os、sys、sqlite 等。这些包是 Python 成功的关键，尽管从技术角度来看它是有缺陷的，但你只需要安装 Python 语言，就可以发送和接收 HTTP 请求、解析 JSON 和 CSV 文件、序列化数据到磁盘、控制 OS 特性，甚至使用 SQLite 数据库。

另一个关键是，它不仅使用内置包提供了基本功能，而且为其他人发布自己的包和使用他人的包提供了一种简单的方法。如果是一个不易使用、不集中，并且不是与语言本身一起构建的系统，那么最终可能会得到一个松散、混乱、滋生 bug 的生态系统，例如 Java 或 iOS 以外平台的旧版本 Swift。

## 3.1 使用内置包

我们探索的第一个示例是一个相对简单的应用程序，它可以从 OMDb API 中获取数据。OMDb 代表 the Open Movie Database，顾名思义，它们提供了大量关于电影的可编程访问信息。

具体来说，我们会构建一个可以根据电影名称搜索电影的应用程序。我们将以常规的方式开始构建应用程序，再加上一些其他东西（不

用担心，我们将在稍后解释它们的含义）。

```
package main

import (
    "encoding/json"
    "errors"
    "io/ioutil"
    "net/http"
    "net/url"
    "strings"
)
```

当然，第一行代码只是告诉 Go 我们正在为哪个包编写代码。然后，让我们来看看导入（import）。到目前为止，你只使用过 fmt 库，但这次会使用许多其他库，如表 3.1 所示。

表 3.1　在 OMDb 应用程序中导入各包的含义

| 包 | 含　义 |
|---|---|
| encoding/json | 用于编码/解码（封装/解封）JSON 对象 |
| errors | 用于引发错误 |
| io/ioutil | 输入/输出程序（用于从 REST API 读取数据流） |
| net/http | HTTP 客户端和服务器的实现（我们只使用客户端） |
| net/url | 处理 URL 应用程序的实现（包括 HTTP URL） |
| strings | 字符串应用程序的实现 |

这些包将提供构建块，我们可以在其上实现自己的逻辑。

**备注**：需要记住的是，在 Go 中，必须用到每个 import；否则，Go 将抛出 error 并拒绝编译程序，而其他编程语言要么静默地继续，要么抛出警告，这是为了保证速度与安全，因为未使用的 import 可能存在 bug。

导入之后，我们将创建一个名为 APIKEY 的新常量，用于存储 OMDB API 键。OMDb API 键用于告诉 API 你是谁，以便服务可以实现如评级限制、权限和演员表等功能。

```
//omdbapi.com API key
const APIKEY = "193ef3a"
```

在继续之前，先简单介绍一下 API：我们将实现两个函数，用其调用众多 API 端点(endpoint)中的两个，这两个函数都将实现电影搜索功能，一个将实现按标题搜索，另一个将实现按 Movie ID(IMDB 中电影的唯一标识符)搜索。

如果调用这两个端点中的任何一个，将得到一个 JSON 响应，其中包含一个符合特定规范(specification)的对象。我们现在要做的是在 Go 语言的结构体中实现相同的规范。这样，当得到 JSON 字符串作为响应时，我们就可以告诉 Go 将 JSON 封装到结构体中。

以下是我们将构建的结构体：

```
//The structure of the returned JSON from omdbapi.com
//To keep this example short, some of the values are not
//mapped into the structure
type MovieInfo struct {
Title string `json:"Title"`
Year string `json:"Year"`
Rated string `json:"Rated"`
Released string `json:"Released"`
Runtime string `json:"Runtime"`
Genre string `json:"Genre"`
Writer string `json:"Writer"`
Actors string `json:"Actors"`
Plot string `json:"Plot"`
Language string `json:"Language"`
Country string `json:"Country"`
```

```
    Awards string `json:"Awards"`
    Poster string `json:"Poster"`
    ImdbRating string `json:"imdbRating"`
    ImdbID string `json:"imdbID"`
    }
```

我们还可以从 API 中获得相当多的其他数据，但是为了简洁起见，我们在结构体中删掉了这些数据。在这个结构体中，还有一些没有介绍过的其他语法——位于类型标注之后，用反引号括起来。例如：

```
    Actors string `json:"Actors"`
```

`json:"Actors"`用来告诉 JSON 包，当你需要将 JSON 字符串封装到这个结构体时，Actors 键的值应该被置于这个变量("Actors"字符串)中，这允许你在 JSON 中为键设置一个与结构体中存储的键的变量名不同的名称。

关于这一点，我们在第 2 章中介绍了简化版的结构体，其中说明了需要在结构体中提供的变量名和类型标注，但是 Go 还可以接收另一部分的变量信息(尽管这是可选的)，这就是所谓的“标签”。标签包含关于变量的其他信息，以及代码的不同部分所需的变量信息。

在本例中，我们使用包含 JSON 键的标签支持 JSON 的封装处理。

现在，我们有了一个包含从 API 获得的信息的结构体，下面实现对 API 的调用。但是在此之前，我们必须编写一个实际负责向 API 发送 HTTP GET 请求的函数。下面是实现这个逻辑的一个简单函数：

```
func sendGetRequest(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }
```

```
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }

    if resp.StatusCode != 200 {
        return string(body), errors.New(resp.Status)
    }
    return string(body), nil
}
```

这个函数的工作方式很简单，它看起来可能很复杂，这是因为它是你在 Go 中处理的第一个较大的函数。让我们从解析函数签名开始：

```
func sendGetRequest(url string) (string, error)
```

这个签名很简单。该函数只接收一个字符串类型的参数 url。但是，它会返回两个独立的值：一个字符串，它将请求的响应表示为字符串；一个 error 类型，error 应该是 nil，但如果在请求或解析响应的某个地方发生了错误，则会包含一个值。

签名之后是函数的主体，因为这是你接触到的第一个真正的函数，所以让我们在表 3.2 中分别介绍每一部分吧。

表 3.2 sendGetRequest 中每部分代码的功能

| 代 码 | 功 能 |
| --- | --- |
| `resp, err := http.Get(url)`<br>`if err != nil {`<br>`  return "", err`<br>`}` | 使用 http 模块中的 GET 函数运行实际的 GET 请求，并将该函数的 url 作为参数传递给它。然后，将响应存储在 resp 和 err 中，并确保 err 为 nil。如果不是，则提前从该函数返回一个空响应和此 error |

续表

| 代　码 | 功　能 |
| --- | --- |
| defer resp.Body.Close()<br>body, err := ioutil.ReadAll<br>(resp.Body)<br>if err != nil {<br>　return "", err<br>} | 告诉 Go“在将这个函数返回给它的调用者之前，确保从响应中关闭 Body 输入流”。然后，使用 io 包读取从响应 Body 中获得的字节，这可能会返回一个 error，因此运行与第一部分中类似的逻辑——如果有 error，则提前返回一个空响应和我们得到的 error |
| if resp.StatusCode != 200 {<br>　return string(body),<br>　errors.New(resp.<br>Status)<br>} | 检查我们得到的状态码，如果它不是 200（意味着一切正常），那么返回响应的 body（其中可能包含有用的信息）并创建一个新的 error，其描述了响应的整个状态 |
| return string(body), nil | 最后一部分是我们希望得到的结果，这意味着管道中没有 error，并且可以将 body 作为字符串返回，没有 error |

完成此函数后，我们现在就能够发送 GET 请求并处理可能出现的一些常见错误。下面让我们实现搜索。

请记住，我们需要实现两种搜索方式：通过标题搜索和通过电影 ID 搜索。让我们先从按标题或姓名搜索开始。

```
func SearchByName(name string) (*MovieInfo, error) {
    parms := url.Values{}
    parms.Set("apikey", APIKEY)
    parms.Set("t", name)
    siteURL := "http://www.omdbapi.com/?" + parms.Encode()
    body, err := sendGetRequest(siteURL)
    if err != nil {
        return nil, errors.New(err.Error() + "\nBody:" + body)
    }
    mi := &MovieInfo{}
    return mi, json.Unmarshal([]byte(body), mi)
}
```

这个函数看起来更简单一些，我们依然从函数签名开始。

```
func SearchByName(name string) (*MovieInfo, error)
```

函数签名乍一看很简单：只是创建了一个接收单个参数并返回两个值的函数。然而，仔细观察后，你会发现返回的第一个值，即从 JSON 响应中解析出的 MovieInfo 结构体实际上是一个指针。

为什么会这样？这是因为我们还有另一个返回值——error。当有一个 error 时，这意味着没有返回一个 MovieInfo 结构体。那么，用什么来代替它呢？我们可以返回一个带有一堆占位符值的空结构体，但这并不美观。你可能会问，为什么不直接返回 nil 呢？如果函数的类型注解只是 MovieInfo，其作为一个值而不是指针，那么 Go 就不允许返回 nil，这是因为没有办法在内存中表示它。

当我们将其声明为指针时，Go 则允许返回 nil，这是因为指针允许指向任何内容。如果我们需要返回一个 error，则可以返回一个带有 error 的 nil 值；如果没有问题，也可以返回一个带有 nil error 的值。

就函数逻辑而言，其本质上遵循以下步骤：

(1) 构建一组 URL 参数，其中包含 API 和我们想要搜索的电影名称；

(2) 组合我们想要查询的 REST API URL，以及步骤(1)中的参数；

(3) 向站点发出请求，如果有错误，则返回 nil 指针和 error；

(4) 创建一个新的 MovieInfo 值，获取指向该值的指针，返回该指针，并通过该指针将响应字符串解析为值；如果数据解析产生错误，则会返回 error。

在所有步骤中，步骤(4)可能是唯一一个比较复杂的，因为它与函数中的最后两行代码相关，让我们来看一下更多的细节。

```
mi := &MovieInfo{}
return mi, json.Unmarshal([]byte(body), mi)
```

代码的第一行很简单：创建结构体，通过取地址操作符“&”获取指针，并将该地址存储在 mi 变量中。第二行有些复杂，如果仔细分析一下，你会发现需要返回的第一个值是一个很容易解析的表达式，它只是变量中的一个值。

但是，需要返回的第二个值就不那么容易解析了。你需要调用一个函数（在本例中是 Unmarshal）获取返回值，然后将其作为此函数的返回值。所以，这是内部发生的事情：

（1）创建一个指向新的 MovieInfo 结构体的指针；

（2）将 body 字符串转换为字节数组；

（3）以字节的形式传递 body 和指向 json 包中的 Unmarshal 函数的指针。Unmarshal 函数的响应是 error 类型，并没有一个真正的“值”被返回，这是因为我们给它传递了一个指针，所以它只是把我们期望它“返回”的值放在指针指向的内存中而已；

（4）该函数返回指向刚刚解析的内存的指针，以及 Unmarshal 函数可能返回的 error。

同样，第一次看可能有些不太直观，但一旦习惯了，它就会更容易理解。

然后，我们对按唯一标识符搜索电影的函数做以下操作：

```
func SearchById(id string) (*MovieInfo, error) {
    parms := url.Values{}
    parms.Set("apikey", APIKEY)
    parms.Set("i", id)
    siteURL := "http://www.omdbapi.com/?" + parms.Encode()
    body, err := sendGetRequest(siteURL)
    if err != nil {
        return nil, errors.New(err.Error() + "\nBody:" + body)
    }
    mi := &MovieInfo{}
```

```
    return mi, json.Unmarshal([]byte(body), mi)
}
```

除了函数名之外，这里唯一的区别在代码的第 4 行。参数的名称不是 t 表示的标题，而是 i 表示的 ID。

以上就是全部代码！现在我们就可以查询 OMDb 并获取有关电影的信息。让我们用 main 函数来测试一下。

```
func main() {
    body, _ := SearchById("tt3896198")
    fmt.Println(body.Title)
    body, _ = SearchByName("Game of")
    fmt.Println(body.Title)
}
```

我们在 main 函数中所做的是按 ID 搜索，然后按 Name 搜索。尝试在这里输入值，看看你会得到哪个电影。要记住的是，我们忽略了这些函数的第二个返回值，它们都是 error 类型的。这意味着，如果此时得到一个错误，则不会处理它。根据错误发生的位置，可能导致应用程序崩溃或打印出一个无用的、无意义的值。

当然，在实际的产品代码中，我们会处理这些错误。

当运行这段代码时，你会看到以下输出：

```
Guardians of the Galaxy Vol. 2
Game of Thrones
```

第一行是 ID 为 tt3896198 的结果的标题，第二行是在 OMDb 中搜索“Game of”时得到的第一个结果的标题。

下面是完整的代码清单。

### 代码清单 3.1　使用 OMDb API 获取电影信息

```
package main

/*
Example of only using many built-in packages in Go to reach out
to a rest API to retrieve movie detail.
*/

import (
    "encoding/json"
    "errors"
    "fmt"
    "io/ioutil"
    "net/http"
    "net/url"
    "strings"
)

//omdbapi.com API key
const APIKEY = "193ef3a"

//The structure of the returned JSON from omdbapi.com
//To keep this example short, some of the values are not
//mapped into the structure

type MovieInfo struct {
    Title string `json:"Title"`
    Year string `json:"Year"`
    Rated string `json:"Rated"`
    Released string `json:"Released"`
    Runtime string `json:"Runtime"`
    Genre string `json:"Genre"`
    Writer string `json:"Writer"`
    Actors string `json:"Actors"`
    Plot string `json:"Plot"`
    Language string `json:"Language"`
    Country string `json:"Country"`
```

```
    Awards string `json:"Awards"`
    Poster string `json:"Poster"`
    ImdbRating string `json:"imdbRating"`
    ImdbID string `json:"imdbID"`
}

func main() {
    body, _ := SearchById("tt3896198")
    fmt.Println(body.Title)
    body, _ = SearchByName("Game of")
    fmt.Println(body.Title)
}

func SearchByName(name string) (*MovieInfo, error) {
    parms := url.Values{}
    parms.Set("apikey", APIKEY)
    parms.Set("t", name)
    siteURL := "http://www.omdbapi.com/?" + parms.Encode()
    body, err := sendGetRequest(siteURL)
    if err != nil {
        return nil, errors.New(err.Error() + "\nBody:" + body)
    }
    mi := &MovieInfo{}
    return mi, json.Unmarshal([]byte(body), mi)
}

func SearchById(id string) (*MovieInfo, error) {
    parms := url.Values{}
    parms.Set("apikey", APIKEY)
    parms.Set("i", id)
    siteURL := "http://www.omdbapi.com/?" + parms.Encode()
    body, err := sendGetRequest(siteURL)
    if err != nil {
        return nil, errors.New(err.Error() + "\nBody:" + body)
    }
    mi := &MovieInfo{}
    return mi, json.Unmarshal([]byte(body), mi)
}
```

```
func sendGetRequest(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }

    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }

    if resp.StatusCode != 200 {
        return string(body), errors.New(resp.Status)
    }
    return string(body), nil
}
```

以上就是一个仅使用内置包构建真实的、有用的应用程序的示例。然而，当你使用社区编写的代码时，编程语言的美就会显现出来。为此，我们将使用 Go 模块。

## 3.2　使用第三方包

对 Go 模块的支持始于 Go 1.11 版本。通过模块，Go 可以无缝地处理第三方包，使程序员能够协作和共享代码。

Go 模块可以由 IDE 处理，也可以通过命令行手动处理。因为我们的目的是尽可能以与平台无关的方式学习 Go，所以只会介绍使用命令行的方法。

处理模块的主要命令是“go mod”。例如，如果你在命令行中运行这个命令：

```
go mod help
```

就会看到一个帮助页面，从中可以了解使用 Go 模块做的所有事情。

Go 模块有以下两种使用方式。

（1）“全局”安装 Go 模块。你需要下载该模块的代码并将其存储在所有项目都可以访问的路径中。这样做的好处是，你只需要获取它一次，然后你的所有项目就都可以访问该模块了。其主要的缺点是有时可能会很烦琐，例如当你需要某个包的特定版本或一个独立的包环境时。

（2）如果创建自己的 Go 模块，则可以在自己的模块中安装第三方 Go 模块，这样可以防止其他项目和模块访问你下载的模块，它将被存储在项目文件夹中。

让我们先编写一些使用第三方 Go 模块的代码，然后探究这两种选择方式。首先，我们建议在项目中安装模块，除非有非常特殊的原因，否则需要在全局安装它们。

我们将构建一个应用程序，它通过命令行从用户处获取一个数字，并打印出该数字是否是素数。这个例子非常简单，所以我们可以很容易地自己编写代码。但是，我们将使用 GitHub 上的一个开源模块完成。

我们将使用的包可以在 www.github.com/otiai10/primes 上找到。整个代码文件如下。

**代码清单 3.2　使用 otiai 10 包检测数字是否为素数**

```
package main

import (
    "fmt"
    "github.com/otiai10/primes"
```

```
    "os"
    "strconv"
)

func main() {
    args := os.Args[1:]
    if len(args) != 1 {
        fmt.Println("Usage:", os.Args[0], "<number>")
        os.Exit(1)
    }
    number, err := strconv.Atoi(args[0])
    if err != nil {
        panic(err)
    }
    f := primes.Factorize(int64(number))
    fmt.Println("primes:", len(f.Powers()) == 1)
}
```

正如你看到的，使用第三方模块和使用内置包几乎是一样的。一个明显的区别是，当运行 import 时，需要指定到模块所在的 GitHub repo（存储库）的链接。

从技术上讲，Go 模块不需要 GitHub 链接即可工作，只需要一个链接到任意远程的 Git 存储库（可能托管在 GitHub、Gitlab、Bitbucket 等），你甚至可以使用自己的 Git 服务器实例。

这使得 Go 模块系统能够处理版本控制。例如，它会专门记住你调用了哪个模块编译了你的代码，从而使编码环境的一致性更好。如果采用刚才提到的第二种方式，即创建自己的包，那么这些信息会存储在一个名为 go.sum 的文件中。

回到我们的代码，如果你此时编译它（使用 go build 命令），将遇到以下错误：

```
main.go:14:2: cannot find package "github.com/otiai10/primes" in any of:
```

```
    /usr/local/Cellar/go/1.14.4/libexec/src/github.com/otiai10/
    primes (from $GOROOT)
    /Users/tanmaybakshi/go/src/github.com/otiai10/primes
    (from $GOPATH)
```

这时 Go 会告诉你“我找不到你试图导入的模块”，它会抛出一个编译器错误。如果你决定使用第一种方法安装软件包（我们并不推荐），那么可以在命令行中运行以下命令：

```
go get github.com/otiai10/primes
```

这会将包下载到你的主文件夹（home folder），如果构建并运行代码，那么它应该可以正常工作。

但是，我们建议执行以下命令：

```
go mod init primechecker
```

这个命令只会做一件事情：在当前目录下，它将创建一个名为 primechecker 的新模块。在本例中，名称并不重要，它只在其他人想使用你的包时才会有用（这是他们用来引用你的模块的名称）。

现在，当运行 go build 命令时，你将看到代码编译成功，这是因为 Go 能够将所需的模块下载到你自己的新模块中。

当运行 go build 命令时，你会看到如下输出：

```
> go build .
go: finding module for package github.com/otiai10/primes go:
found github.com/otiai10/primes in github.com/otiai10/primes
v0.0.0-20180210170552-f6d2a1ba97c4
```

在 go.sum 文件中，你会看到以下内容：

```
github.com/otiai10/primes v0.0.0-20180210170552-f6d2a1ba97c4/
go.mod h1:UmSP7QeU3XmAdGu5+dnrTJqjBc+IscpVZkQzk473cjM=
```

## 3.3　构建自己的包

正如我们已经提到的，在 Go 中制作和构建自己的自定义包非常简单。

现在，为了让其他人使用你的模块，你必须非常具体地命名这个模块。在上面的示例中，你可以将模块命名为任意名称，但是，如果你想将该模块推送到远程 Git 服务器供其他人使用，则必须将包命名为其他人可以在其代码中导入的远程 Git 库。

例如，让我们编写一个允许用户检查一个数字是否为素数的包。可以像下面这样初始化包：

```
go mod init github.com/Tanmay-Teaches/golang/chapter3/example3
```

这样，我们的 Go 模块就可以被想要使用我们代码的人自动导入了。

下面编写这个包的代码。当然，我们将在 main.go 文件中进行编码。但是这次，我们将把包命名为 example3，而不是 package main，这表示当人们在自己的代码中引用这个包时，将把它称为 example3。

**代码清单 3.3　自定义素数检测包**

```
package example3

func IsPrime(n int) bool {
    if n <= 1 {
        return false
```

```
    } else if n <= 3 {
        return true
    } else if n %2 == 0 || n %3 == 0 {
        return false
    }

    i := 5
    for i * i <n {
        if n %i == 0 || n %(i + 2) == 0 {
            return false
        }
        i += 6
    }
    return true
}
```

IsPrime 函数背后的逻辑非常简单：有几个条件用来检查返回真或假的情况，然后从 5 循环到我们想要检查的数字的平方根。如果输入的数字能被循环中的任一数字整除，则返回 false；否则返回 true。

现在我们可以将代码送入 GitHub 仓库。如果这个 GitHub 仓库恰巧是公开的，那么本章的其余部分应该就不言自明了，你应该能够找到它。你需要做的就是提交修改并送入到你的仓库——或者你的 Git 工作流可能需要的其他东西。

你可能会想：如果 GitHub 的仓库是私有的，我们该怎么办？或者你在一家使用 GitHub Enterprise 的公司工作，你有自己的 GitHub 实例，又该如何告诉 Go 对特定的 GitHub 实例进行身份验证呢？

通常，可以使用 SSH 密钥对 Git 进行身份验证。SSH 密钥易于使用，非常安全，并且是行业标准。

要告诉 Go 使用 SSH，必须从一个 Git 命令开始。具体来说，我们必须告诉 git，当看到 HTTPS URL 时，它必须用 SSH 请求替换 HTTPS 请求。下面是设置全局 Git 配置的方法：

```
git config - - global url." git @ github. com:". insteadOf " https://
github.com/"
```

然后，你需要告诉 Go 不要对要克隆的私有存储库或内部存储库进行 sum check 校验，这是因为 sum check 与私有存储库和内部存储库不兼容。

```
export GONOSUMDB=github.com/Tanmay-Teaches/golang
```

此时，你应该能够创建一个新项目，在该项目中引用你在私有 Git 存储库中创建的模块，然后获取该模块并在自己的代码中使用它。

首先，创建一个新的 Go 项目（创建一个新文件夹）并初始化一个新模块，如下所示：

```
go mod init example4
```

同样，当你为自己创建模块而不打算发布时，模块的名称并不重要。

除了为自己创建的模块之外，我们还将使用 labstack 的另一个名为 echo 的第三方模块，这个模块将帮助我们创建一个快速 Web（HTTP）服务器。总之，我们的应用程序将：

（1）在一个特定的端口打开一个 HTTP 服务器；

（2）监听来自端口的 GET 请求；

（3）如果它收到了一个 GET 请求，其中 URL 的路径是一个数字，则检测数字是否为素数：

① 如果是，则对 GET 请求返回 true 作为响应；

② 如果不是，对 GET 请求返回 false 作为响应。

下面是该应用程序的所有代码。

### 代码清单 3.4 使用自定义素数检测包

```
package main

import (
    "net/http"
    "github.com/Tanmay-Teaches/golang/chapter3/example3"
    "github.com/labstack/echo/v4"
    "strconv"
)

func main() {
    e := echo.New()
    e.GET("/:number", func(c echo.Context) error {
        nstr := c.Param("number")
        n, err := strconv.Atoi(nstr)
        if err != nil {
            return c.String(http.StatusBadRequest, err.Error())
        }
return c.String(http.StatusOK,
            ↪ strconv.FormatBool(example3.IsPrime(n)))
    })
    e.Logger.Fatal(e.Start(":1323"))
}
```

在运行代码之前，我们先来了解一下它是如何工作的。当启动 main 函数时，我们只需要调用 echo.New()，就可以生成一个新的 HTTP Server 实例。之后，我们针对该服务器注册一个 GET 端点。

通过传递一个字符串"/:number"注册这个 GET 端点，这意味着在 URL 的根节点之后，如果你看到的唯一一个东西是我们称为“number”的特定字符序列，那么我们就可以处理请求。