

在构建自己的网络服务时，有几种方法可以在不运行服务的情况下，调用服务的功能进行测试。

9.1.1 基础单元测试

让我们看一个单元测试的例子，如代码清单 9-1 所示。

代码清单 9-1 listing01_test.go

```
01 // 这个示例程序展示如何写基础单元测试
02 package listing01
03
04 import (
05     "net/http"
06     "testing"
07 )
08
09 const checkMark = "\u2713"
10 const ballotX = "\u2717"
11
12 // TestDownload 确认 http 包的 Get 函数可以下载内容
13 func TestDownload(t *testing.T) {
14     url := "http://www.goinggo.net/feeds/posts/default?alt=rss"
15     statusCode := 200
16
17     t.Log("Given the need to test downloading content.")
18     {
19         t.Logf("\tWhen checking \"%s\" for status code \"%d\"",
20             url, statusCode)
21         {
22             resp, err := http.Get(url)
23             if err != nil {
24                 t.Fatalf("\t\tShould be able to make the Get call.",
25                     ballotX, err)
26             }
27             t.Log("\t\tShould be able to make the Get call.",
28                 checkMark)
29
30             defer resp.Body.Close()
31
32             if resp.StatusCode == statusCode {
33                 t.Logf("\t\tShould receive a \"%d\" status. %v",
34                     statusCode, checkMark)
35             } else {
36                 t.Errorf("\t\tShould receive a \"%d\" status. %v %v",
37                     statusCode, ballotX, resp.StatusCode)
38             }
39         }
40     }
41 }
```

代码清单 9-1 展示了测试 http 包的 Get 函数的单元测试。测试的内容是确保可以从网络正常下载 goinggo.net 的 RSS 列表。如果通过调用 `go test -v` 来运行这个测试（-v 表示提供冗

余输出)，会得到图 9-1 所示的测试结果。

```
$ go test -v
=== RUN TestDownload
--- PASS: TestDownload (0.43s)
    listing01_test.go:17: Given the need to test downloading content.
    listing01_test.go:20:   When checking "http://www.goinggo.net/feeds/posts
    listing01_test.go:28:   Should be able to make the Get call. ✓
    listing01_test.go:34:   Should receive a "200" status. ✓
PASS
ok      github.com/goinaction/code/chapter9/listing01    0.435s
```

图 9-1 基础单元测试的输出

这个例子背后发生了很多事情，来确保测试能正确工作，并显示结果。让我们从测试文件的文件名开始。如果查看代码清单 9-1 一开始的部分，会看到测试文件的文件名是 `listing01_test.go`。Go 语言的测试工具只会认为以 `_test.go` 结尾的文件是测试文件。如果没有遵从这个约定，在包里运行 `go test` 的时候就可能会报告没有测试文件。一旦测试工具找到了测试文件，就会查找里面的测试函数并执行。

让我们仔细看看 `listing01_test.go` 测试文件里面的代码，如代码清单 9-2 所示。

代码清单 9-2 listing01_test.go: 第 01 行到第 10 行

```
01 // 这个示例程序展示如何写基础单元测试
02 package listing01
03
04 import (
05     "net/http"
06     "testing"
07 )
08
09 const checkMark = "\u2713"
10 const ballotX = "\u2717"
```

在代码清单 9-2 里，可以看到第 06 行引入了 `testing` 包。这个 `testing` 包提供了从测试框架到报告测试的输出和状态的各种测试功能的支持。第 09 行和第 10 行声明了两个常量，这两个常量包含写测试输出时会用到的对号（✓）和叉号（×）。

接下来，让我们看一下测试函数的声明，如代码清单 9-3 所示。

代码清单 9-3 listing01_test.go: 第 12 行到第 13 行

```
12 // TestDownload 确认 http 包的 Get 函数可以下载内容
13 func TestDownload(t *testing.T) {
```

在代码清单 9-3 的第 13 行中，可以看到测试函数的名字是 `TestDownload`。一个测试函数必须是公开的函数，并且以 `Test` 单词开头。不但函数名字要以 `Test` 开头，而且函数的签名必须接收一个指向 `testing.T` 类型的指针，并且不返回任何值。如果没有遵守这些约定，测试框架就不会认为这个函数是一个测试函数，也不会让测试工具去执行它。

指向 `testing.T` 类型的指针很重要。这个指针提供的机制可以报告每个测试的输出和状态。

测试的输出格式没有标准要求。我更喜欢使用 Go 写文档的方式，输出容易读的测试结果。对我来说，测试的输出是代码文档的一部分。测试的输出需使用完整易读的语句，来记录为什么需要这个测试，具体测试了什么，以及测试的结果是什么。让我们来看一下更多的代码，了解我是如何完成这些测试的，如代码清单 9-4 所示。

代码清单 9-4 listing01_test.go: 第 14 行到第 18 行

```
14     url := "http://www.goinggo.net/feeds/posts/default?alt=rss"
15     statusCode := 200
16
17     t.Log("Given the need to test downloading content.")
18     {
```

可以看到，在代码清单 9-4 的第 14 行和第 15 行，声明并初始化了两个变量。这两个变量包含了要测试的 URL，以及期望从响应中返回的状态。在第 17 行，使用方法 `t.Log` 来输出测试的消息。这个方法还有一个名为 `t.Logf` 的版本，可以格式化消息。如果执行 `go test` 的时候没有加入冗余选项（`-v`），除非测试失败，否则我们是看不到任何测试输出的。

每个测试函数都应该通过解释这个测试的给定要求（`given need`），来说明为什么应该存在这个测试。对这个例子来说，给定要求是测试能否成功下载数据。在声明了测试的给定要求后，测试应该说明被测试的代码应该在什么情况下被执行，以及如何执行。

代码清单 9-5 listing01_test.go: 第 19 行到第 21 行

```
19         t.Logf("\tWhen checking \"%s\" for status code \"%d\"",
20             url, statusCode)
21     {
```

可以在代码清单 9-5 的第 19 行看到测试执行条件的说明。它特别说明了要测试的值。接下来，让我们看一下被测试的代码是如何使用这些值来进行测试的。

代码清单 9-6 listing01_test.go: 第 22 行到第 30 行

```
22         resp, err := http.Get(url)
23         if err != nil {
24             t.Fatal("\t\tShould be able to make the Get call.",
25                 ballotX, err)
26         }
27         t.Log("\t\tShould be able to make the Get call.",
28             checkMark)
29
30         defer resp.Body.Close()
```

代码清单 9-6 中的代码使用 `http` 包的 `Get` 函数来向 `goinggo.net` 网络服务器发起请求，请求下载该博客的 RSS 列表。在 `Get` 调用返回之后，会检查错误值，来判断调用是否成功。在每种情况下，我们都会说明测试应有的结果。如果调用失败，除了结果，还会输出叉号以及得到的错误值。如果测试成功，会输出对号。

如果 Get 调用失败,使用第 24 行的 `t.Fatal` 方法,让测试框架知道这个测试失败了。`t.Fatal` 方法不但报告这个单元测试已经失败,而且会向测试输出写一些消息,而后立刻停止这个测试函数的执行。如果除了这个函数外还有其他没有执行的测试函数,会继续执行其他测试函数。这个方法对应的格式化版本名为 `t.Fatalf`。

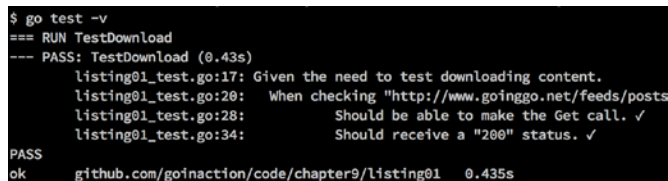
如果需要报告测试失败,但是并不想停止当前测试函数的执行,可以使用 `t.Error` 系列方法,如代码清单 9-7 所示。

代码清单 9-7 listing01_test.go: 第 32 行到第 41 行

```
32         if resp.StatusCode == statusCode {
33             t.Logf("\t\tShould receive a \"%d\" status. %v",
34                 statusCode, checkMark)
35         } else {
36             t.Errorf("\t\tShould receive a \"%d\" status. %v %v",
37                 statusCode, ballotX, resp.StatusCode)
38         }
39     }
40 }
41 }
```

在代码清单 9-7 的第 32 行,会将响应返回的状态码和我们期望收到的状态码进行比较。我们再次声明了期望测试返回的结果是什么。如果状态码匹配,我们就使用 `t.Logf` 方法输出信息;否则,就使用 `t.Errorf` 方法。因为 `t.Errorf` 方法不会停止当前测试函数的执行,所以,如果在第 38 行之后还有测试,单元测试就会继续执行。如果测试函数执行时没有调用过 `t.Fatal` 或者 `t.Error` 方法,就会认为测试通过了。

如果再看一下测试的输出(如图 9-2 所示),你会看到这段代码组合在一起的效果。



```
$ go test -v
=== RUN TestDownload
--- PASS: TestDownload (0.43s)
    listing01_test.go:17: Given the need to test downloading content.
    listing01_test.go:20:   When checking "http://www.goinggo.net/feeds/posts
    listing01_test.go:28:   Should be able to make the Get call. ✓
    listing01_test.go:34:   Should receive a "200" status. ✓
PASS
ok      github.com/goinaction/code/chapter9/listing01  0.435s
```

图 9-2 基础单元测试的输出

在图 9-2 中能看到这个测试的完整文档。下载给定的内容,当检测获取 URL 的内容返回的状态码时(在图中被截断),我们应该能够成功完成这个调用并收到状态 200。测试的输出很清晰,能描述测试的目的,同时包含了足够的信息。我们知道具体是哪个单元测试被运行,测试通过了,并且运行消耗的时间是 435 毫秒。

9.1.2 表组测试

如果测试可以接受一组不同的输入并产生不同的输出的代码,那么应该使用表组测试的方法

进行测试。表组测试除了会有一组不同的输入值和期望结果之外，其余部分都很像基础单元测试。测试会依次迭代不同的值，来运行要测试的代码。每次迭代的时候，都会检测返回的结果。这便于在一个函数里测试不同的输入值和条件。让我们看一个表组测试的例子，如代码清单 9-8 所示。

代码清单 9-8 listing08_test.go

```
01 // 这个示例程序展示如何写一个基本的表组测试
02 package listing08
03
04 import (
05     "net/http"
06     "testing"
07 )
08
09 const checkMark = "\u2713"
10 const ballotX = "\u2717"
11
12 // TestDownload 确认 http 包的 Get 函数可以下载内容
13 // 并正确处理不同的状态
14 func TestDownload(t *testing.T) {
15     var urls = []struct {
16         url      string
17         statusCode int
18     }{
19         {
20             "http://www.goinggo.net/feeds/posts/default?alt=rss",
21             http.StatusOK,
22         },
23         {
24             "http://rss.cnn.com/rss/cnn_topstbadurl.rss",
25             http.StatusNotFound,
26         },
27     }
28
29     t.Log("Given the need to test downloading different content.")
30     {
31         for _, u := range urls {
32             t.Logf("\tWhen checking \"%s\" for status code \"%d\"",
33                 u.url, u.statusCode)
34             {
35                 resp, err := http.Get(u.url)
36                 if err != nil {
37                     t.Fatal("\t\tShould be able to Get the url.",
38                         ballotX, err)
39                 }
40                 t.Log("\t\tShould be able to Get the url",
41                     checkMark)
42
43                 defer resp.Body.Close()
44
45                 if resp.StatusCode == u.statusCode {
46                     t.Logf("\t\tShould have a \"%d\" status. %v",
```

```

47         u.statusCode, checkMark)
48     } else {
49         t.Errorf("\t\tShould have a \"%d\" status %v %v",
50             u.statusCode, ballotX, resp.StatusCode)
51     }
52 }
53 }
54 }
55 }

```

在代码清单 9-8 中，我们稍微改动了之前的基础单元测试，将其变为表组测试。现在，可以使用一个测试函数来测试不同的 URL 以及 http.Get 方法的返回状态码。我们不需要为每个要测试的 URL 和状态码创建一个新测试函数。让我们看一下，和之前相比，做了哪些改动，如代码清单 9-9 所示。

代码清单 9-9 listing08_test.go: 第 12 行到第 27 行

```

12 // TestDownload 确认 http 包的 Get 函数可以下载内容
13 // 并正确处理不同的状态
14 func TestDownload(t *testing.T) {
15     var urls = []struct {
16         url      string
17         statusCode int
18     }{
19         {
20             "http://www.goinggo.net/feeds/posts/default?alt=rss",
21             http.StatusOK,
22         },
23         {
24             "http://rss.cnn.com/rss/cnn_topstbadurl.rss",
25             http.StatusNotFound,
26         },
27     }

```

在代码清单 9-9 中，可以看到和之前同名的测试函数 TestDownload，它接收一个指向 testing.T 类型的指针。但这个版本的 TestDownload 略微有些不同。在第 15 行到第 27 行，可以看到表组的实现代码。表组的第一个字段是 URL，指向一个给定的互联网资源，第二个字段是我们请求资源后期望收到的状态码。

目前，我们的表组只配置了两组值。第一组值是 goinggo.net 的 URL，响应状态为 OK，第二组值是另一个 URL，响应状态为 NotFound。运行这个测试会得到图 9-3 所示的输出。

```

$ go test -v
=== RUN TestDownload
--- PASS: TestDownload (0.72s)
    listing02_test.go:29: Given the need to test downloading different content
    listing02_test.go:33:   When checking "http://www.goinggo.net/feeds/posts/
    listing02_test.go:41:     Should be able to Get the url. ✓
    listing02_test.go:47:     Should have a "200" status. ✓
    listing02_test.go:33:   When checking "http://rss.cnn.com/rss/cnn_topstba
    listing02_test.go:41:     Should be able to Get the url. ✓
    listing02_test.go:47:     Should have a "404" status. ✓
PASS
ok      github.com/goinaction/code/chapter9/listing02  0.724s

```

图 9-3 表组测试的输出

图 9-3 所示的输出展示了如何迭代表组里的值，并使用其进行测试。输出看起来和基础单元测试的输出很像，只是每次都会输出两个不同的 URL 及其结果。测试又通过了。

让我们看一下我们是如何让表组测试工作的，如代码清单 9-10 所示。

代码清单 9-10 listing08_test.go: 第 29 行到第 34 行

```
29     t.Log("Given the need to test downloading different content.")
30     {
31         for _, u := range urls {
32             t.Logf("\tWhen checking \"%s\" for status code \"%d\"",
33                 u.url, u.statusCode)
34             {
```

代码清单 9-10 的第 31 行的 `for range` 循环让测试迭代表组里的值，使用不同的 URL 运行测试代码。测试的代码与基础单元测试的代码相同，只不过这次使用的是表组内的值进行测试，如代码清单 9-11 所示。

代码清单 9-11 listing08_test.go: 第 35 行到第 55 行

```
35         resp, err := http.Get(u.url)
36         if err != nil {
37             t.Fatal("\t\tShould be able to Get the url.",
38                 ballotX, err)
39         }
40         t.Log("\t\tShould be able to Get the url",
41             checkMark)
42
43         defer resp.Body.Close()
44
45         if resp.StatusCode == u.statusCode {
46             t.Logf("\t\tShould have a \"%d\" status. %v",
47                 u.statusCode, checkMark)
48         } else {
49             t.Errorf("\t\tShould have a \"%d\" status %v %v",
50                 u.statusCode, ballotX, resp.StatusCode)
51         }
52     }
53 }
54 }
55 }
```

代码清单 9-11 的第 35 行中展示了代码如何使用 `u.url` 字段来做 URL 调用。在第 45 行中，`u.statusCode` 字段被用于和实际的响应状态码进行比较。如果以后需要扩展测试，只需要将新的 URL 和状态码加入表组就可以，不需要改动测试的核心代码。

9.1.3 模仿调用

我们之前写的单元测试都很好，但是还有些瑕疵。首先，这些测试需要访问互联网，才能保

证测试运行成功。图 9-4 展示了如果没有互联网连接，运行基础单元测试会测试失败。

```
$ go test -v
=== RUN TestDownload
--- FAIL: TestDownload (0.00s)
    listing01_test.go:17: Given the need to test downloading content.
    listing01_test.go:20: When checking "http://www.goinggo.net/feeds/posts
    listing01_test.go:25: Should be able to make the Get call. X Get
    http://www.goinggo.net/feeds/posts/default?alt=rss: dial tcp: lookup www.goinggo.
    net: no such host
FAIL
exit status 1
FAIL    github.com/goinaction/code/chapter9/listing01    0.009s
```

图 9-4 由于没有互联网连接导致测试失败

不能总是假设运行测试的机器可以访问互联网。此外，依赖不属于你的或者你无法操作的服务来进行测试，也不是一个好习惯。这两点会严重影响测试持续集成和部署的自动化。如果突然断网，导致测试失败，就没办法部署新构建的程序。

为了修正这个问题，标准库包含一个名为 `httptest` 的包，它让开发人员可以模仿基于 HTTP 的网络调用。模仿（mocking）是一个很常用的技术手段，用来在运行测试时模拟访问不可用的资源。包 `httptest` 可以让你能够模仿互联网资源的请求和响应。在我们的单元测试中，通过模仿 `http.Get` 的响应，我们可以解决在图 9-4 中遇到的问题，保证在没有网络的时候，我们的测试也不会失败，依旧可以验证我们的 `http.Get` 调用正常工作，并且可以处理预期的响应。让我们看一下基础单元测试，并将其改为模仿调用 `goinggo.net` 网站的 RSS 列表，如代码清单 9-12 所示。

代码清单 9-12 listing12_test.go: 第 01 行到第 41 行

```
01 // 这个示例程序展示如何内部模仿 HTTP GET 调用
02 // 与本书之前的例子有些差别
03 package listing12
04
05 import (
06     "encoding/xml"
07     "fmt"
08     "net/http"
09     "net/http/httptest"
10     "testing"
11 )
12
13 const checkMark = "\u2713"
14 const ballotX = "\u2717"
15
16 // feed 模仿了我们期望接收的 XML 文档
17 var feed = `<?xml version="1.0" encoding="UTF-8"?>
18 <rss>
19 <channel>
20     <title>Going Go Programming</title>
21     <description>Golang : https://github.com/goinggo</description>
22     <link>http://www.goinggo.net/</link>
```



```

23     <item>
24         <pubDate>Sun, 15 Mar 2015 15:04:00 +0000</pubDate>
25         <title>Object Oriented Programming Mechanics</title>
26         <description>Go is an object oriented language.</description>
27         <link>http://www.goinggo.net/2015/03/object-oriented</link>
28     </item>
29 </channel>
30 </rss>`
31
32 // mockServer 返回用来处理请求的服务器的指针
33 func mockServer() *httptest.Server {
34     f := func(w http.ResponseWriter, r *http.Request) {
35         w.WriteHeader(200)
36         w.Header().Set("Content-Type", "application/xml")
37         fmt.Fprintln(w, feed)
38     }
39
40     return httptest.NewServer(http.HandlerFunc(f))
41 }

```

代码清单 9-12 展示了如何模仿对 goinggo.net 网站的调用，来模拟下载 RSS 列表。在第 17 行中，声明了包级变量 feed，并初始化为模仿服务器返回的 RSS XML 文档的字符串。这是实际 RSS 文档的一小段，足以完成我们的测试。在第 33 行中，我们声明了一个名为 mockServer 的函数，这个函数利用 httptest 包内的支持来模拟对互联网上真实服务器的调用，如代码清单 9-13 所示。

代码清单 9-13 listing12_test.go: 第 32 行到第 41 行

```

32 // mockServer 返回用来处理调用的服务器的指针
33 func mockServer() *httptest.Server {
34     f := func(w http.ResponseWriter, r *http.Request) {
35         w.WriteHeader(200)
36         w.Header().Set("Content-Type", "application/xml")
37         fmt.Fprintln(w, feed)
38     }
39
40     return httptest.NewServer(http.HandlerFunc(f))
41 }

```

代码清单 9-13 中声明的 mockServer 函数，返回一个指向 httptest.Server 类型的指针。这个 httptest.Server 的值是整个模仿服务的关键。函数的代码一开始声明了一个匿名函数，其签名符合 http.HandlerFunc 函数类型，如代码清单 9-14 所示。

代码清单 9-14 golang.org/pkg/net/http/#HandlerFunc

```
type HandlerFunc func(ResponseWriter, *Request)
```

HandlerFunc 类型是一个适配器，允许常规函数作为 HTTP 的处理函数使用。如果函数 f 具有合适的签名，HandlerFunc(f) 就是一个处理 HTTP 请求的 Handler 对象，内部通过调用 f 处理请求

遵守这个签名，让匿名函数成了处理函数。一旦声明了这个处理函数，第 40 行就会使用这

个匿名函数作为参数来调用 `httpptest.NewServer` 函数，创建我们的模仿服务器。之后在第 40 行，通过指针返回这个模仿服务器。

我们可以通过 `http.Get` 调用来使用这个模仿服务器，用来模拟对 `goinggo.net` 网络服务器的请求。当进行 `http.Get` 调用时，实际执行的是处理函数，并用处理函数模仿对网络服务器的请求和响应。在第 35 行，处理函数首先设置状态码，之后在第 36 行，设置返回内容的类型 `Content-Type`，最后，在第 37 行，使用包含 XML 内容的字符串 `feed` 作为响应数据，返回给调用者。

现在，让我们看一下模仿服务器与基础单元测试是怎么整合在一起的，以及如何将 `http.Get` 请求发送到模仿服务器，如代码清单 9-15 所示。

代码清单 9-15 listing12_test.go: 第 43 行到第 74 行

```
43 // TestDownload 确认 http 包的 Get 函数可以下载内容
44 // 并且内容可以被正确地反序列化并关闭
45 func TestDownload(t *testing.T) {
46     statusCode := http.StatusOK
47
48     server := mockServer()
49     defer server.Close()
50
51     t.Log("Given the need to test downloading content.")
52     {
53         t.Logf("\tWhen checking \"%s\" for status code \"%d\"",
54             server.URL, statusCode)
55         {
56             resp, err := http.Get(server.URL)
57             if err != nil {
58                 t.Fatalf("\t\tShould be able to make the Get call.",
59                     ballotX, err)
60             }
61             t.Log("\t\tShould be able to make the Get call.",
62                 checkMark)
63
64             defer resp.Body.Close()
65
66             if resp.StatusCode != statusCode {
67                 t.Fatalf("\t\tShould receive a \"%d\" status. %v %v",
68                     statusCode, ballotX, resp.StatusCode)
69             }
70             t.Logf("\t\tShould receive a \"%d\" status. %v",
71                 statusCode, checkMark)
72         }
73     }
74 }
```

在代码清单 9-15 中再次看到了 `TestDownload` 函数，不过这次它在请求模仿服务器。在第 48 行和第 49 行，调用 `mockServer` 函数生成模仿服务器，并安排在测试函数返回时执行服务器的 `Close` 方法。之后，除了代码清单 9-16 所示的这一行代码，这段测试代码看上去和基础单

元测试的代码一模一样。

代码清单 9-16 listing12_test.go: 第 56 行

```
56         resp, err := http.Get(server.URL)
```

这次由 `httptest.Server` 值提供了请求的 URL。当我们使用由模仿服务器提供的 URL 时, `http.Get` 调用依旧会按我们预期的方式运行。`http.Get` 方法调用时并不知道我们的调用是否经过互联网。这次调用最终会执行, 并且我们自己的处理函数最终被执行, 返回我们预先准备好的 XML 文档和状态码 `http.StatusOK`。

在图 9-5 里, 如果在没有互联网连接的时候运行测试, 可以看到测试依旧可以运行并通过。这张图展示了程序是如何再次通过测试的。如果仔细看用于调用的 URL, 会发现这个 URL 使用了 `localhost` 作为地址, 端口是 52065。这个端口号每次运行测试时都会改变。包 `http` 与包 `httptest` 和模仿服务器结合在一起, 知道如何通过 URL 路由到我们自己的处理函数。现在, 我们可以在没有触碰实际服务器的情况下, 测试请求 `goinggo.net` 的 RSS 列表。

```
$ go test -v
=== RUN TestDownload
--- PASS: TestDownload (0.00s)
    listing03_test.go:51: Given the need to test downloading content.
    listing03_test.go:54:   When checking "http://127.0.0.1:52065" for status code "200"
    listing03_test.go:62:     Should be able to make the Get call. ✓
    listing03_test.go:71:     Should receive a "200" status. ✓
    listing03_test.go:79:     Should be able to unmarshal the response. ✓
    listing03_test.go:83:     Should have "1" item in the feed. ✓
PASS
ok      github.com/goinaction/code/chapter9/listing03  0.007s
```

图 9-5 没有互联网接入情况下测试成功

9.1.4 测试服务端点

服务端点 (endpoint) 是指与服务宿主信息无关, 用来分辨某个服务的地址, 一般是不包含宿主的一个路径。如果在构造网络 API, 你会希望直接测试自己的服务的所有服务端点, 而不用启动整个网络服务。包 `httptest` 正好提供了做到这一点的机制。让我们看一个简单的包含一个服务端点的网络服务的例子, 如代码清单 9-17 所示, 之后你会看到如何写一个单元测试, 来模仿真正的调用。

代码清单 9-17 listing17.go

```
01 // 这个示例程序实现了简单的网络服务
02 package main
03
04 import (
05     "log"
06     "net/http"
07
08     "github.com/goinaction/code/chapter9/listing17/handlers"
09 )
10
```

```

11 // main 是应用程序的入口
12 func main() {
13     handlers.Routes()
14
15     log.Println("listener : Started : Listening on :4000")
16     http.ListenAndServe(":4000", nil)
17 }

```

代码清单 9-17 展示的代码文件是整个网络服务的入口。在第 13 行的 main 函数里，代码调用了内部 handlers 包的 Routes 函数。这个函数为托管的网络服务设置了一个服务端点。在 main 函数的第 15 行和第 16 行，显示服务监听的端口，并且启动网络服务，等待请求。

现在让我们来看一下 handlers 包的代码，如代码清单 9-18 所示。

代码清单 9-18 handlers/handlers.go

```

01 // handlers 包提供了用于网络服务的服务端点
02 package handlers
03
04 import (
05     "encoding/json"
06     "net/http"
07 )
08
09 // Routes 为网络服务设置路由
10 func Routes() {
11     http.HandleFunc("/sendjson", SendJSON)
12 }
13
14 // SendJSON 返回一个简单的 JSON 文档
15 func SendJSON(rw http.ResponseWriter, r *http.Request) {
16     u := struct {
17         Name string
18         Email string
19     }{
20         Name: "Bill",
21         Email: "bill@ardanstudios.com",
22     }
23
24     rw.Header().Set("Content-Type", "application/json")
25     rw.WriteHeader(200)
26     json.NewEncoder(rw).Encode(&u)
27 }

```

代码清单 9-18 里展示了 handlers 包的代码。这个包提供了实现好的处理函数，并且能为网络服务设置路由。在第 10 行，你能看到 Routes 函数，使用 http 包里默认的 http.ServeMux 来配置路由，将 URL 映射到对应的处理代码。在第 11 行，我们将 /sendjson 服务端点与 SendJSON 函数绑定在一起。

从第 15 行起，是 SendJSON 函数的实现。这个函数的签名和之前看到代码清单 9-14 里 http.HandlerFunc 函数类型的签名一致。在第 16 行，声明了一个匿名结构类型，使用这个结构创建了一个名为 u 的变量，并赋予一组初值。在第 24 行和第 25 行，设置了响应的内容类型

和状态码。最后，在第 26 行，将 `u` 值编码为 JSON 文档，并发送回发起调用的客户端。

如果我们构建了一个网络服务，并启动服务器，就可以像图 9-6 和图 9-7 展示的那样，通过服务获取 JSON 文档。

```
$ ./listing17
2015/06/13 18:53:06 listener : Started : Listening on :4000
```

图 9-6 启动网络服务

```
localhost:4000/sendjson
{"Name": "Bill", "Email": "bill@ardanstudios.com"}
```

图 9-7 网络服务提供的 JSON 文档

现在有了包含一个服务端点的可用的网络服务，我们可以写单元测试来测试这个服务端点，如代码清单 9-19 所示。

代码清单 9-19 handlers/handlers_test.go

```
01 // 这个示例程序展示如何测试内部服务端点
02 // 的执行效果
03 package handlers_test
04
05 import (
06     "encoding/json"
07     "net/http"
08     "net/http/httptest"
09     "testing"
10
11     "github.com/goinaction/code/chapter9/listing17/handlers"
12 )
13
14 const checkMark = "\u2713"
15 const ballotX = "\u2717"
16
17 func init() {
18     handlers.Routes()
19 }
20
21 // TestSendJSON 测试/sendjson 内部服务端点
22 func TestSendJSON(t *testing.T) {
23     t.Log("Given the need to test the SendJSON endpoint.")
24     {
25         req, err := http.NewRequest("GET", "/sendjson", nil)
26         if err != nil {
27             t.Fatal("\tShould be able to create a request.",
28                 ballotX, err)
29         }
30         t.Log("\tShould be able to create a request.",
31             checkMark)
32
33         rw := httptest.NewRecorder()
34         http.DefaultServeMux.ServeHTTP(rw, req)
35
36         if rw.Code != 200 {
37             t.Fatal("\tShould receive \"200\"", ballotX, rw.Code)
```

```

38     }
39     t.Log("\tShould receive \"200\"", checkMark)
40
41     u := struct {
42         Name string
43         Email string
44     }{}
45
46     if err := json.NewDecoder(rw.Body).Decode(&u); err != nil {
47         t.Fatal("\tShould decode the response.", ballotX)
48     }
49     t.Log("\tShould decode the response.", checkMark)
50
51     if u.Name == "Bill" {
52         t.Log("\tShould have a Name.", checkMark)
53     } else {
54         t.Error("\tShould have a Name.", ballotX, u.Name)
55     }
56
57     if u.Email == "bill@ardanstudios.com" {
58         t.Log("\tShould have an Email.", checkMark)
59     } else {
60         t.Error("\tShould have an Email.", ballotX, u.Email)
61     }
62 }
63 }

```

代码清单 9-19 展示了对 `/sendjson` 服务端点的单元测试。注意，第 03 行包的名字和其他测试代码的包的名字不太一样，如代码清单 9-20 所示。

代码清单 9-20 `handlers/handlers_test.go`: 第 01 行到第 03 行

```

01 // 这个示例程序展示如何测试内部服务端点
02 // 的执行效果
03 package handlers_test

```

正如在代码清单 9-20 里看到的，这次包的名字也使用 `_test` 结尾。如果包使用这种方式命名，测试代码只能访问包里公开的标识符。即便测试代码文件和被测试的代码放在同一个文件夹中，也只能访问公开的标识符。

就像直接运行服务时一样，需要为服务端点初始化路由，如代码清单 9-21 所示。

代码清单 9-21 `handlers/handlers_test.go`: 第 17 行到第 19 行

```

17 func init() {
18     handlers.Routes()
19 }

```

在代码清单 9-21 的第 17 行，声明的 `init` 函数里对路由进行初始化。如果没有在单元测试运行之前初始化路由，那么测试就会遇到 `http.StatusNotFound` 错误而失败。现在让我们看一下 `/sendjson` 服务端点的单元测试，如代码清单 9-22 所示。

代码清单 9-22 handlers/handlers_test.go: 第 21 行到第 34 行

```

21 // TestSendJSON 测试/sendjson 内部服务端点
22 func TestSendJSON(t *testing.T) {
23     t.Log("Given the need to test the SendJSON endpoint.")
24     {
25         req, err := http.NewRequest("GET", "/sendjson", nil)
26         if err != nil {
27             t.Fatal("\tShould be able to create a request.",
28                 ballotX, err)
29         }
30         t.Log("\tShould be able to create a request.",
31             checkMark)
32     }
33     rw := httptest.NewRecorder()
34     http.DefaultServeMux.ServeHTTP(rw, req)

```

代码清单 9-22 展示了测试函数 TestSendJSON 的声明。测试从记录测试的给定要求开始，然后在第 25 行创建了一个 http.Request 值。这个 Request 值使用 GET 方法调用 /sendjson 服务端点的响应。由于这个调用使用的是 GET 方法，第三个发送数据的参数被传入 nil。

之后，在第 33 行，调用 httptest.NewRecorder 函数来创建一个 http.ResponseRecorder 值。有了 http.Request 和 http.ResponseRecorder 这两个值，就可以在第 34 行直接调用服务默认的多路选择器（mux）的 ServeHTTP 方法。调用这个方法模仿了外部客户端对 /sendjson 服务端点的请求。

一旦 ServeHTTP 方法调用完成，http.ResponseRecorder 值就包含了 SendJSON 处理函数的响应。现在，我们可以检查这个响应的内容，如代码清单 9-23 所示。

代码清单 9-23 handlers/handlers_test.go: 第 36 行到第 39 行

```

36     if rw.Code != 200 {
37         t.Fatal("\tShould receive \"200\"", ballotX, rw.Code)
38     }
39     t.Log("\tShould receive \"200\"", checkMark)

```

首先，在第 36 行检查了响应的状态。一般任何服务端点成功调用后，都会期望得到 200 的状态码。如果状态码是 200，之后将 JSON 响应解码成 Go 的值。

代码清单 9-24 handlers/handlers_test.go: 第 41 行到第 49 行

```

41     u := struct {
42         Name string
43         Email string
44     }{}
45
46     if err := json.NewDecoder(rw.Body).Decode(&u); err != nil {
47         t.Fatal("\tShould decode the response.", ballotX)
48     }
49     t.Log("\tShould decode the response.", checkMark)

```

在代码清单 9-24 的第 41 行，声明了一个匿名结构类型，使用这个类型创建了名为 u 的变量，

并初始化为零值。在第 46 行，使用 json 包将响应的 JSON 文档解码到变量 u 里。如果解码失败，单元测试结束；否则，我们会验证解码后的值是否正确，如代码清单 9-25 所示。

代码清单 9-25 handlers/handlers_test.go: 第 51 行到第 63 行

```
51     if u.Name == "Bill" {
52         t.Log("\tShould have a Name.", checkMark)
53     } else {
54         t.Error("\tShould have a Name.", ballotX, u.Name)
55     }
56
57     if u.Email == "bill@ardanstudios.com" {
58         t.Log("\tShould have an Email.", checkMark)
59     } else {
60         t.Error("\tShould have an Email.", ballotX, u.Email)
61     }
62 }
63 }
```

代码清单 9-25 展示了对收到的两个值的检测。在第 51 行，我们检测 Name 字段的值是否为 "Bill"，之后在第 57 行，检查 Email 字段的值是否为 "bill@ardanstudios.com"。如果这些值都匹配，单元测试通过；否则，单元测试失败。这两个检测使用 Error 方法来报告失败，所以不管检测结果如何，两个字段都会被检测。

9.2 示例

Go 语言很重视给代码编写合适的文档。专门内置了 godoc 工具来从代码直接生成文档。在第 3 章中，我们已经学过如何使用 godoc 工具来生成包的文档。这个工具的另一个特性是示例代码。示例代码给文档和测试都增加了一个可以扩展的维度。

如果使用浏览器来浏览 json 包的 Go 文档，会看到类似图 9-8 所示的文档。



图 9-8 包 json 的示例代码列表

包 json 含有 5 个示例，这些示例都会在这个包的 Go 文档里有展示。如果选中第一个示例，

会看到一段示例代码，如图 9-9 所示。

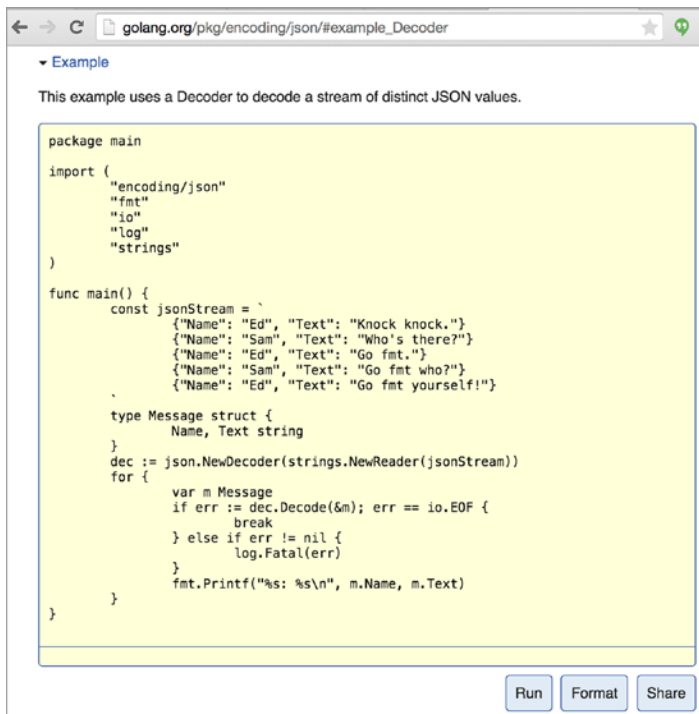


图 9-9 Go 文档里显示的 Decoder 示例视图

开发人员可以创建自己的示例，并且在包的 Go 文档里展示。让我们看一个来自前一节例子的 `SendJSON` 函数的示例，如代码清单 9-26 所示。

代码清单 9-26 handlers_example_test.go

```
01 // 这个示例程序展示如何编写基础示例
02 package handlers_test
03
04 import (
05     "encoding/json"
06     "fmt"
07     "log"
08     "net/http"
09     "net/http/httptest"
10 )
11
12 // ExampleSendJSON 提供了基础示例
13 func ExampleSendJSON() {
14     r, _ := http.NewRequest("GET", "/sendjson", nil)
15     rw := httptest.NewRecorder()
16     http.DefaultServeMux.ServeHTTP(rw, r)
```

```

17
18     var u struct {
19         Name string
20         Email string
21     }
22
23     if err := json.NewDecoder(w.Body).Decode(&u); err != nil {
24         log.Println("ERROR:", err)
25     }
26
27     // 使用 fmt 将结果写到 stdout 来检测输出
28     fmt.Println(u)
29     // Output:
30     // {Bill bill@ardanstudios.com}
31 }

```

示例基于已经存在的函数或者方法。我们需要使用 `Example` 代替 `Test` 作为函数名的开始。在代码清单 9-26 的第 13 行中，示例代码的名字是 `ExampleSendJSON`。

对于示例代码，需要遵守一个规则。示例代码的函数名字必须基于已经存在的公开的函数或者方法。我们的示例的名字基于 `handlers` 包里公开的 `SendJSON` 函数。如果没有使用已经存在的函数或者方法，这个示例就不会显示在包的 Go 文档里。

写示例代码的目的是展示某个函数或者方法的特定使用方法。为了判断测试是成功还是失败，需要将程序最终的输出和示例函数底部列出的输出做比较，如代码清单 9-27 所示。

代码清单 9-27 `handlers_example_test.go`: 第 27 行到第 31 行

```

27     // 使用 fmt 将结果写到 stdout 来检测输出
28     fmt.Println(u)
29     // Output:
30     // {Bill bill@ardanstudios.com}
31 }

```

在代码清单 9-27 的第 28 行，代码使用 `fmt.Println` 输出变量 `u` 的值到标准输出。变量 `u` 的值在调用 `/sendjson` 服务端点之前使用零值初始化。在第 29 行中，有一段带有 `Output:` 的注释。

这个 `Output:` 标记用来在文档中标记出示例函数运行后期望的输出。Go 的测试框架知道如何比较注释里的期望输出和标准输出的最终输出。如果两者匹配，这个示例作为测试就会通过，并加入到包的 Go 文档里。如果输出不匹配，这个示例作为测试就会失败。

如果启动一个本地的 `godoc` 服务器 (`godoc -http=":3000"`)，并找到 `handlers` 包，就能看到包含示例的文档，如图 9-10 所示。

在图 9-10 里可以看到 `handlers` 包的文档里展示了 `SendJSON` 函数的示例。如果选中这个 `SendJSON` 链接，文档就会展示这段代码，如图 9-11 所示。

图 9-11 展示了示例的一组完整文档，包括代码和期望的输出。由于这个示例也是测试的一部分，可以使用 `go test` 工具来运行这个示例函数，如图 9-12 所示。

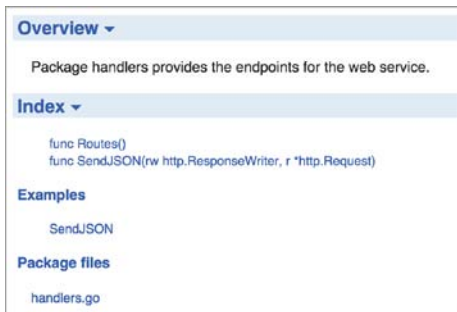


图 9-10 handlers 包的 godoc 视图

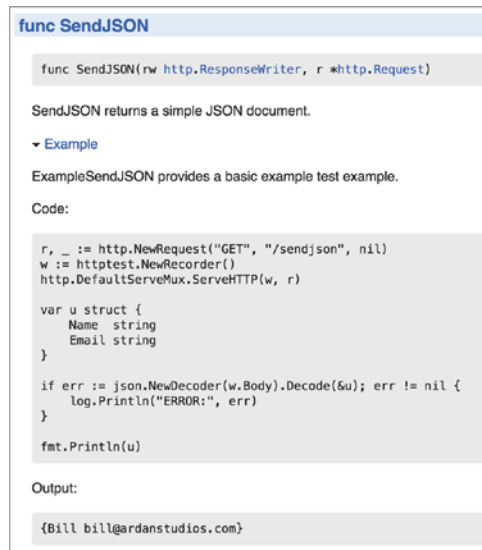


图 9-11 在 godoc 里显示完整的示例代码

```
$ go test -v -run="ExampleSendJSON"
=== RUN: ExampleSendJSON
--- PASS: ExampleSendJSON (0.00s)
PASS
ok      github.com/goinaction/code/chapter9/listing17/handlers 0.008s
```

图 9-12 运行示例代码

运行测试后，可以看到测试通过了。这次运行测试时，使用 `-run` 选项指定了特定的函数 `ExampleSendJSON`。`-run` 选项接受任意的正则表达式，来过滤要运行的测试函数。这个选项既支持单元测试，也支持示例函数。如果示例运行失败，输出会与图 9-13 所示的样子类似。

```
$ go test -v -run="ExampleSendJSON"
=== RUN: ExampleSendJSON
--- FAIL: ExampleSendJSON (0.00s)
got:
{Lisa lisa@gmail.com}
want:
{Bill bill@ardanstudios.com}
FAIL
exit status 1
FAIL    github.com/goinaction/code/chapter9/listing17/handlers 0.006s
```

图 9-13 示例运行失败

如果示例运行失败，`go test` 会同时展示出生成的输出，以及期望的输出。

9.3 基准测试

基准测试是一种测试代码性能的方法。想要测试解决同一问题的不同方案的性能，以及查看哪种解决方案的性能更好时，基准测试就会很有用。基准测试也可以用来识别某段代码的 CPU

或者内存效率问题，而这段代码的效率可能会严重影响整个应用程序的性能。许多开发人员会用基准测试来测试不同的并发模式，或者用基准测试来辅助配置工作池的数量，以保证能最大化系统的吞吐量。

让我们看一组基准测试的函数，找出将整数值转为字符串的最快方法。在标准库里，有 3 种方法可以将一个整数值转为字符串。

代码清单 9-28 展示了 listing28_test.go 基准测试开始的几行代码。

代码清单 9-28 listing28_test.go: 第 01 行到第 10 行

```
01 // 用来检测要将整数值转为字符串，使用哪个函数会更好的基准
02 // 测试示例。先使用 fmt.Sprintf 函数，然后使用
03 // strconv.FormatInt 函数，最后使用 strconv.Itoa
04 package listing28_test
05
06 import (
07     "fmt"
08     "strconv"
09     "testing"
10 )
```

和单元测试文件一样，基准测试的文件名也必须以 _test.go 结尾。同时也必须导入 testing 包。接下来，让我们看一下其中一个基准测试函数，如代码清单 9-29 所示。

代码清单 9-29 listing28_test.go: 第 12 行到第 22 行

```
12 // BenchmarkSprintf 对 fmt.Sprintf 函数
13 // 进行基准测试
14 func BenchmarkSprintf(b *testing.B) {
15     number := 10
16
17     b.ResetTimer()
18
19     for i := 0; i < b.N; i++ {
20         fmt.Sprintf("%d", number)
21     }
22 }
```

在代码清单 9-29 的第 14 行，可以看到第一个基准测试函数，名为 BenchmarkSprintf。基准测试函数必须以 Benchmark 开头，接受一个指向 testing.B 类型的指针作为唯一参数。为了让基准测试框架能准确测试性能，它必须在一段时间内反复运行这段代码，所以这里使用了 for 循环，如代码清单 9-30 所示。

代码清单 9-30 listing28_test.go: 第 19 行到第 22 行

```
19     for i := 0; i < b.N; i++ {
20         fmt.Sprintf("%d", number)
21     }
22 }
```

代码清单 9-30 第 19 行的 for 循环展示了如何使用 b.N 的值。在第 20 行，调用了 fmt 包

里的 `Sprintf` 函数。这个函数是将要测试的将整数值转为字符串的函数。

基准测试框架默认会在持续 1 秒的时间内，反复调用需要测试的函数。测试框架每次调用测试函数时，都会增加 `b.N` 的值。第一次调用时，`b.N` 的值为 1。需要注意，一定要将所有要进行基准测试的代码都放到循环里，并且循环要使用 `b.N` 的值。否则，测试的结果是不可靠的。

如果我们只希望运行基准测试函数，需要加入 `-bench` 选项，如代码清单 9-31 所示。

代码清单 9-31 运行基准测试

```
go test -v -run="none" -bench="BenchmarkSprintf"
```

在这次 `go test` 调用里，我们给 `-run` 选项传递了字符串 `"none"`，来保证在运行制订的基准测试函数之前没有单元测试会被运行。这两个选项都可以接受正则表达式，来决定需要运行哪些测试。由于例子里没有单元测试函数的名字中有 `none`，所以使用 `none` 可以排除所有的单元测试。发出这个命令后，得到图 9-14 所示的输出。

```
$ go test -v -run="none" -bench="BenchmarkSprintf"
testing: warning: no tests to run
PASS
BenchmarkSprintf      5000000      258 ns/op
ok      github.com/goinaction/code/chapter9/listing28  1.562s
```

图 9-14 运行单个基准测试

这个输出一开始明确了没有单元测试被运行，之后开始运行 `BenchmarkSprintf` 基准测试。在输出 `PASS` 之后，可以看到运行这个基准测试函数的结果。第一个数字 5000000 表示在循环中的代码被执行的次数。在这个例子里，一共执行了 500 万次。之后的数字表示代码的性能，单位为每次操作消耗的纳秒（ns）数。这个数字展示了这次测试，使用 `Sprintf` 函数平均每次花费了 258 纳秒。

最后，运行基准测试输出了 `ok`，表明基准测试正常结束。之后显示的是被执行的代码文件的名称。最后，输出运行基准测试总共消耗的时间。默认情况下，基准测试的最小运行时间是 1 秒。你会看到这个测试框架持续运行了大约 1.5 秒。如果想让运行时间更长，可以使用另一个名为 `-benchtime` 的选项来更改测试执行的最短时间。让我们再次运行这个测试，这次持续执行 3 秒（见图 9-15）。

```
$ go test -v -run="none" -bench="BenchmarkSprintf" -benchtime="3s"
testing: warning: no tests to run
PASS
BenchmarkSprintf      2000000      256 ns/op
ok      github.com/goinaction/code/chapter9/listing28  5.384s
```

图 9-15 使用 `-benchtime` 选项来运行基准测试

这次 `Sprintf` 函数运行了 2000 万次，持续了 5.384 秒。这个函数的执行性能并没有太大的变化，这次的性能是每次操作消耗 256 纳秒。有时候，增加基准测试的时间，会得到更加精确的性能结果。对大多数测试来说，超过 3 秒的基准测试并不会改变测试的精确度。只是每次基准测试的结果会稍有不同。

让我们看另外两个基准测试函数，并一起运行这 3 个基准测试，看看哪种将整数值转换为字符串的方法最快，如代码清单 9-32 所示。

代码清单 9-32 listing28_test.go: 第 24 行到第 46 行

```
24 // BenchmarkFormat 对 strconv.FormatInt 函数
25 // 进行基准测试
26 func BenchmarkFormat(b *testing.B) {
27     number := int64(10)
28
29     b.ResetTimer()
30
31     for i := 0; i < b.N; i++ {
32         strconv.FormatInt(number, 10)
33     }
34 }
35
36 // BenchmarkItoa 对 strconv.Itoa 函数
37 // 进行基准测试
38 func BenchmarkItoa(b *testing.B) {
39     number := 10
40
41     b.ResetTimer()
42
43     for i := 0; i < b.N; i++ {
44         strconv.Itoa(number)
45     }
46 }
```

代码清单 9-32 展示了另外两个基准测试函数。函数 `BenchmarkFormat` 测试了 `strconv` 包里的 `FormatInt` 函数，而函数 `BenchmarkItoa` 测试了同样来自 `strconv` 包的 `Itoa` 函数。这两个基准测试函数的模式和 `BenchmarkSprintf` 函数的模式很类似。函数内部的 `for` 循环使用 `b.N` 来控制每次调用时迭代的次数。

我们之前一直没有提到这 3 个基准测试里面调用 `b.ResetTimer` 的作用。在代码开始执行循环之前需要进行初始化时，这个方法用来重置计时器，保证测试代码执行前的初始化代码，不会干扰计时器的结果。为了保证得到的测试结果尽量精确，需要使用这个函数来跳过初始化代码的执行时间。

让这 3 个函数至少运行 3 秒后，我们得到图 9-16 所示的结果。



```
$ go test -v -run="none" -bench=. -benchtime="3s"
testing: warning: no tests to run
PASS
BenchmarkSprintf      20000000      257 ns/op
BenchmarkFormat 100000000      45.9 ns/op
BenchmarkItoa  100000000      49.4 ns/op
ok      github.com/goinaction/code/chapter9/listing28  15.057s
```

图 9-16 运行所有 3 个基准测试

这个结果展示了 `BenchmarkFormat` 测试函数运行的速度最快，每次操作耗时 45.9 纳秒。紧随其后的是 `BenchmarkItoa`，每次操作耗时 49.4 ns。这两个函数的性能都比 `Sprintf` 函数快得多。

运行基准测试时，另一个很有用的选项是 `-benchmem` 选项。这个选项可以提供每次操作分配内存的次数，以及总共分配内存的字节数。让我们看一下如何使用这个选项（见图 9-17）。

```
$ go test -v -run="none" -bench=. -benchtime="3s" -benchmem
testing: warning: no tests to run
PASS
BenchmarkSprintf    200000000      255 ns/op      16 B/op      2 allocs/op
BenchmarkFormat    1000000000      45.8 ns/op       2 B/op      1 allocs/op
BenchmarkItoa      1000000000      49.5 ns/op       2 B/op      1 allocs/op
ok      github.com/goinaction/code/chapter9/listing28    15.008s
```

图 9-17 使用 `-benchmem` 选项来运行基准测试

这次输出的结果会多出两组新的数值：一组数值的单位是 `B/op`，另一组的单位是 `allocs/op`。单位为 `allocs/op` 的值表示每次操作从堆上分配内存的次数。你可以看到 `Sprintf` 函数每次操作都会从堆上分配两个值，而另外两个函数每次操作只会分配一个值。单位为 `B/op` 的值表示每次操作分配的字节数。你可以看到 `Sprintf` 函数两次分配总共消耗了 16 字节的内存，而另外两个函数每次操作只会分配 2 字节的内存。

在运行单元测试和基准测试时，还有很多选项可以用。建议读者查看一遍所有选项，以便在编写自己的包和工程时，充分利用测试框架。社区希望包的作者在正式发布包的时候提供足够的测试。

9.4 小结

- 测试功能被内置到 Go 语言中，Go 语言提供了必要的测试工具。
- `go test` 工具用来运行测试。
- 测试文件总是以 `_test.go` 作为文件名的结尾。
- 表组测试是利用一个测试函数测试多组值的好办法。
- 包中的示例代码，既能用于测试，也能用于文档。
- 基准测试提供了探查代码性能的机制。

Go 语言实战

即便不处理类似可扩展的 Web 并发或者实时性能等复杂的系统编程问题，应用程序开发也是一件非常困难的事情。尽管使用一些工具和框架也可以解决这些常见的问题，但 Go 语言却以一种更加自然且高效的方式正确处理了这类问题。由谷歌公司开发的 Go 语言，为在基础设施中非常依赖高性能服务的初创公司和大企业提供了足够的能力。

本书目标读者是已经有一定其他编程语言经验，想要开始学习 Go 语言或者更深入了解 Go 语言及其内部机制的中级开发者。本书会提供一个专注、全面且符合习惯的视角。本书关注 Go 语言的规范和实现，涉及的内容包括语法、Go 的类型系统、并发、通道和测试等主题。

本书主要内容

- Go 语言规范和实现。
- Go 语言的类型系统。
- Go 语言的数据结构的内部实现。
- 测试和基准测试。

本书假设读者是熟练使用其他语言（如 Java、Ruby、Python、C# 或者 C++）的开发者。

William Kennedy 是一位熟练的软件开发人员，也是博客 GoingGo.Net 的作者。**Brian Ketelsen** 和 **Erik St. Martin** 是全球 Go 语言大会 GopherCon 的组织者，也是 Go 语言框架 Skynet 的联合作者。



异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机 / 程序设计 / Go 语言
人民邮电出版社网址：www.ptpress.com.cn

“探索、学习并使用 Go 语言的简洁而全面的指导手册。”

——摘自 Hugo 创立者 Steven Francia
为本书写的序

“这本权威的书为所有想要开始学习 Go 语言的人提供了一站式的指引。”

——Sam Zaydel, RackTop Systems

“写得太好了！完整介绍了 Go 语言。强烈推荐。”

——Adam McKay, SUEZ

“这本书把 Go 语言不寻常的部分讲得通俗易懂。”

——Alex Vidal, Atlassian 的 HipChat 团队

ISBN 978-7-115-44535-3



9 787115 445353 >