

选项 `-run` 的参数是一个正则表达式，它可以使得 `go test` 只运行那些测试函数名称匹配给定模式的函数：

```
$ go test -v -run="French|Canal"
==> RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
==> RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL      gopl.io/ch11/word1  0.014s
```

当然，一旦我们使得选择的测试用例通过之后，在我们提交更改之前，我们必须重新使用不带开关的 `go test` 来运行一次整个测试套件。

现在的任务是修复 bug。我们经过迅速地调查发现第一个 bug 的原因是函数 `IsPalindrome` 使用字节序列而不是字符序列来比较，因此那些非 ASCII 字符（例如 "été" 中的 "é"）就使得程序困惑了。第二个 bug 的原因是没有忽略空格、标点符号和字母大小写。

有了教训后，我们仔细地重写了这个函数：

```
gopl.io/ch11/word2
// 包 word 提供了文字游戏相关的工具函数
package word

import "unicode"

// IsPalindrome 判断一个字符串是否是回文字符串
// 忽略字母大小写，以及非字母字符
func IsPalindrome(s string) bool {
    var letters []rune
    for _, r := range s {
        if unicode.IsLetter(r) {
            letters = append(letters, unicode.ToLower(r))
        }
    }
    for i := range letters {
        if letters[i] != letters[len(letters)-1-i] {
            return false
        }
    }
    return true
}
```

我们还写了更加全面的测试用例，把前面的用例和新的用例结合到一个表里面。

```
func TestIsPalindrome(t *testing.T) {
    var tests = []struct {
        input string
        want  bool
    }{
        {"", true},
        {"a", true},
        {"aa", true},
        {"ab", false},
        {"kayak", true},
        {"detartrated", true},
        {"A man, a plan, a canal: Panama", true},
        {"Evil I did dwell; lewd did I live.", true},
    }
```

```
    {"Able was I ere I saw Elba", true},
    {"été", true},
    {"Et se resservir, ivresse reste.", true},
    {"palindrome", false}, // 非回文
    {"desserts", false}, // 半回文
}
for _, test := range tests {
    if got := IsPalindrome(test.input); got != test.want {
        t.Errorf("IsPalindrome(%q) = %v", test.input, got)
    }
}
}
```

新的测试可以通过了：

```
$ go test gopl.io/ch11/word2
ok      gopl.io/ch11/word2      0.015s
```

这种基于表的测试方式在 Go 里面很常见。根据需要添加新的表项目很直观，并且由于断言逻辑没有重复，因此我们可以花点精力让输出的错误消息更好看一点。

当前调用 `t.Errorf` 输出的失败的测试用例信息没有包含整个跟踪栈信息，也不会导致程序宕机或者终止执行，这和很多其他语言的测试框架中的断言不同。测试用例彼此是独立的。如果测试表中的一个条目造成测试失败，那么其他的条目仍然会继续测试，这样我们就可以在一次测试过程中发现多个失败的情况。

如果我们真的需要终止一个测试函数，比如由于初始化代码失败或者避免已有的错误产生令人困惑的输出，我们可以使用 `t.Fatal` 或者 `t.Fatalf` 函数来终止测试。这些函数的调用必须和 `Test` 函数在同一个 goroutine 中，而不是在测试创建的其他 goroutine 中。

测试错误消息一般格式是 "`f(x)=y, want z`"，这里 `f(x)` 表示需要执行的操作和它的输入，`y` 是实际的输出结果，`z` 是期望得到的结果。出于方便，对于 `f(x)` 我们会使用 Go 的语法，比如在上面回文的例子中，我们使用 Go 的格式化来显示较长的输入，避免重复输入。在基于表的测试中，输出 `x` 是很重要的，因为一条断言语句会在不同的输入情况下执行多次。错误消息要避免样板文字和冗余信息。在测试一个布尔函数的时候，比如上面的 `IsPalindrome`，省略 "`want z`" 部分，因为它没有给出有用信息。如果 `x`、`y`、`z` 都比较长，可以输出准确代表各部分的概要信息。在程序员诊断一个测试失败的时候，测试用例的作者必须努力帮助程序员。

练习 11.1：为 4.3 节的 `charcount` 程序编写测试用例。

练习 11.2：为 6.5 节的 `IntSet` 编写测试用例用来检测它的行为在每一次操作之后和基于内置的 `map` 实现的 `Set` 一致。保存你的实现，我们将在练习 11.7 中会进行基准测试。

11.2.1 随机测试

基于表的测试方便针对精心选择的输入检测函数是否工作正常，以测试逻辑上引人关注的用例。另外一种方式是随机测试，通过构建随机输入来扩展测试的覆盖范围。

如果给出的输入是随机的，我们怎么知道函数输出什么内容呢？这里有两种策略。一种方式就是额外写一个函数，这个函数使用低效但是清晰的算法，然后检查这两种实现的输出是否一致。另外一种方式是构建符合某种模式的输入，这样我们可以知道它们对应的输出是

什么。

下面的例子使用了第二种方式，`randomPalindrome` 函数产生一系列的回文字符串，这些输出在构建的时候就确定是回文字符串了。

```
import "math/rand"

// randomPalindrome 返回一个回文字符串，它的长度和内容都是随机数生成器//rng 生成的
func randomPalindrome(rng *rand.Rand) string {
    n := rng.Intn(25) // 随机字符串最大长度是 24
    runes := make([]rune, n)
    for i := 0; i < (n+1)/2; i++ {
        r := rune(rng.Intn(0x1000)) // 随机字符最大是 '\u0999'
        runes[i] = r
        runes[n-1-i] = r
    }
    return string(runes)
}

func TestRandomPalindromes(t *testing.T) {
    // 初始化一个伪随机数生成器
    seed := time.Now().UTC().UnixNano()
    t.Logf("Random seed: %d", seed)
    rng := rand.New(rand.NewSource(seed))
    for i := 0; i < 1000; i++ {
        p := randomPalindrome(rng)
        if !IsPalindrome(p) {
            t.Errorf("IsPalindrome(%q) = false", p)
        }
    }
}
```

由于随机测试的不确定性，在遇到测试用例失败的情况下，一定要记录足够的信息以便于重现这个问题。在该例子中，函数 `IsPalindrome` 的输入 `p` 告诉我们所需要知道的所有信息，但是对于那些拥有更复杂输入的函数来说，记录伪随机数生成器的种子（如我们所做的那样）会比转储整个输入数据结构要简单得多。有了随机数的种子，我们可以简单地修改测试代码来准确地重现错误。

通过使用当前时间作为伪随机数的种子源，在测试的整个生命周期中，每次运行的时候都会得到新的输入。如果你的项目使用自动化系统来间周期地运行测试，这一点很重要。

练习 11.3： `TestRandomPalindromes` 函数仅测试回文字符串。编写一个随机测试用来产生并检测非回文字符串。

练习 11.4： 修改 `randomPalindrome` 来测试 `IsPalindrome` 函数对标点符号和空格的处理。

11.2.2 测试命令

`go test` 工具对测试库代码包很有用，但是也可以将它用于测试命令。包名 `main` 一般会产生可执行文件，但是也可以当做库来导入。

为 2.3.2 节的 `echo` 程序写一个测试。把程序分为两个函数，`echo` 执行逻辑，而 `main` 用来读取和解析命令行参数以及报告 `echo` 函数可能返回的错误。

```
gopl.io/ch11/echo
// Echo 输出它的命令行参数
package main

import (
    "flag"
    "fmt"
    "io"
    "os"
    "strings"
)
var (
    n = flag.Bool("n", false, "omit trailing newline")
    s = flag.String("s", " ", "separator")
)
var out io.Writer = os.Stdout // 测试过程中被更改

func main() {
    flag.Parse()
    if err := echo(!*n, *s, flag.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "echo: %v\n", err)
        os.Exit(1)
    }
}

func echo(newline bool, sep string, args []string) error {
    fmt.Fprint(out, strings.Join(args, sep))
    if newline {
        fmt.Fprintln(out)
    }
    return nil
}
```

在测试中，我们会通过不同的参数和开关来调用 `echo`，以检查它在每种测试用例下得到正确的输出，所以我们还为 `echo` 函数添加了参数以避免依赖全局变量。也就是说，我们还引入了另外一个全局变量 `out`，该变量是 `io.Writer` 类型，所有的结果都将输出到这里。通过将 `echo` 输出到这个变量而不是直接输出到 `os.Stdout`，测试用例还可以用其他的替代 `Writer` 实现来记录写入的内容以便于后面检查。下面是测试代码（在文件 `echo_test.go` 中）：

```
package main

import (
    "bytes"
    "fmt"
    "testing"
)

func TestEcho(t *testing.T) {
    var tests = []struct {
        newline bool
        sep     string
        args   []string
        want   string
    }{
        {true, "", []string{}, "\n"},
        {false, "", []string{}, ""},
        {true, "\t", []string{"one", "two", "three"}, "one\ttwo\tthree\n"},
        {true, ",", []string{"a", "b", "c"}, "a,b,c\n"},
        {false, ":"}, []string{"1", "2", "3"}, "1:2:3"},
    }
}
```

```

for _, test := range tests {
    descr := fmt.Sprintf("echo(%v, %q, %q)",
        test.newline, test.sep, test.args)
    out = new(bytes.Buffer) // 捕获的输出
    if err := echo(test.newline, test.sep, test.args); err != nil {
        t.Errorf("%s failed: %v", descr, err)
        continue
    }
    got := out.(*bytes.Buffer).String()
    if got != test.want {
        t.Errorf("%s = %q, want %q", descr, got, test.want)
    }
}
}

```

注意，测试代码和产品代码在一个包里面。尽管包的名称叫作 `main`，并且里面定义了一个 `main` 函数，但是在测试过程中，这个包当作库来测试，并且将函数 `TestEcho` 递送到测试驱动程序，而 `main` 函数则被忽略了。

通过表来组织测试用例，我们可以很容易地添加新的测试用例。下面添加一行到测试用例表中，来看看测试失败的时候发生了什么。

```
{true, "", []string{"a", "b", "c"}, "a b c\n"}, // 注意，预期结果是错误的
```

运行 `go test` 输出：

```
$ go test gopl.io/ch11/echo
--- FAIL: TestEcho (0.00s)
    echo_test.go:31: echo(true, "", ["a" "b" "c"]) = "a,b,c", want "a b c\n"
FAIL
FAIL    gopl.io/ch11/echo    0.006s
```

错误消息描述了想要进行的操作（使用了类似 Go 的语法），然后依次是实际行为和预期的结果。有了如此详细的错误消息，你在定位测试的源代码之前就很容易了解错误的根源了。

记住，在测试的代码里面不要调用 `log.Fatal` 或者 `os.Exit`，因为这两个调用会阻止跟踪的过程，这两个函数的调用可以认为是 `main` 函数的特权。如果有时候发生了未预期错误或者函数崩溃了，即使测试用例本身失败了，测试驱动程序也可以继续工作。预期的错误（比如用户输入的内容不合法、文件不存在、配置不正确等）应该通过返回一个非空的 `error` 值来报告。幸运的是，`echo` 程序很简单，它不会返回一个非空的 `error` 值。

11.2.3 白盒测试

测试的分类方式之一是基于对所要进行测试的包的内部了解程度。黑盒测试假设测试者对包的了解仅通过公开的 API 和文档，而包的内部逻辑则是不透明的。相反，白盒测试可以访问包的内部函数和数据结构，并且可以做一些常规用户无法做到的观察和改动。例如，白盒测试可以检查包的数据类型不可变性在每次操作后都是经过维护的。（白盒这个名字是传统说法，净盒（clear box）的说法或许更准确。）

这两种方法是互补的。黑盒测试通常更加健壮，每次程序更新后基本不需要修改。它们也会帮助测试的作者关注包的用户并且能够发现 API 设计的缺陷。反之，白盒测试可以对实现的特定之处提供更详细的覆盖测试。

上面已经给出了这两种测试方法的例子。`TestIsPalindrome` 函数仅调用导出的函数 `IsPalindrome`，所以它是一个黑盒测试。`TestEcho` 函数调用 `echo` 函数并且更新全局变量 `out`，无论函数 `echo` 还是变量 `out` 都是未导出的，所以它是一个白盒测试。

在开发 TestEcho 的时候，我们修改了 echo 函数，从而在输出结果时使用一个包级别的变量，以便该测试用一个额外的实现代替标准输出来记录后面要检查的数据。通过同样的技术，我们可以使用易于测试的伪实现来替换部分产品代码。这种伪实现的优点是更易于配置、预测和观察，并且更可靠。它们还能够避免带来副作用，比如更新产品数据库或者刷信用卡。

下面的代码演示了向用户提供存储服务的 Web 服务中的限额逻辑。当用户使用的额度超过 90% 的时候，系统自动发送一封告警邮件。

```
gopl.io/ch11/storage1
package storage

import (
    "fmt"
    "log"
    "net/smtp"
)

func bytesInUse(username string) int64 { return 0 /* ... */ }

// 邮件发送者配置
// 注意：永远不要把密码放到源代码中
const sender = "notifications@example.com"
const password = "correcthorsebatterystaple"
const hostname = "smtp.example.com"

const template = `Warning: you are using %d bytes of storage,
%d% of your quota.`

func CheckQuota(username string) {
    used := bytesInUse(username)
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
    if percent < 90 {
        return // OK
    }
    msg := fmt.Sprintf(template, used, percent)
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
        []string{username}, []byte(msg))
    if err != nil {
        log.Printf("smtp.SendMail(%s) failed: %s", username, err)
    }
}
```

我们想测试这个功能，但是并不想真的发送邮件出去。所以我们把发送邮件的逻辑移动到独立的函数中，并且把它存储到一个不可导出的包级别的变量 notifyUser 中。

```
gopl.io/ch11/storage2
var notifyUser = func(username, msg string) {
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
        []string{username}, []byte(msg))
    if err != nil {
        log.Printf("smtp.SendEmail(%s) failed: %s", username, err)
    }
}

func CheckQuota(username string) {
    used := bytesInUse(username)
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
```

```

if percent < 90 {
    return // OK
}
msg := fmt.Sprintf(template, used, percent)
notifyUser(username, msg)
}

```

现在我们可以写个简单的测试，这个测试用伪造的通知机制而不是发送一封真实的邮件。这个测试记录需要通知的用户和通知的内容。

```

package storage

import (
    "strings"
    "testing"
)
func TestCheckQuotaNotifiesUser(t *testing.T) {
    var notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ...模拟已使用 980MB 的情况...
    const user = "joe@example.org"
    CheckQuota(user)
    if notifiedUser == "" && notifiedMsg == "" {
        t.Fatalf("notifyUser not called")
    }
    if notifiedUser != user {
        t.Errorf("wrong user (%s) notified, want %s",
            notifiedUser, user)
    }
    const wantSubstring = "98% of your quota"
    if !strings.Contains(notifiedMsg, wantSubstring) {
        t.Errorf("unexpected notification message <<%s>>, "+
            "want substring %q", notifiedMsg, wantSubstring)
    }
}

```

这里有一个问题，在这个测试函数返回之后，`CheckQuota` 因为仍然使用该测试的伪通知实现 `notifyUser`，所以再次在其他测试中调用它时就不能正常工作了。（对于全局变量的更新一直都是存在风险的）。我们必须修改这个测试让它恢复 `notifyUser` 原来的值，这样后面的测试才不会受影响。我们必须在所有的测试执行路径上面都这样做，包括测试失败和宕机。通常这种情况下建议使用 `defer`。

```

func TestCheckQuotaNotifiesUser(t *testing.T) {
    // 保存留待恢复的notifyUser
    saved := notifyUser
    defer func() { notifyUser = saved }()
    // 设置测试的伪通知notifyUser
    var notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ...测试其余的部分...
}

```

这种方式可以用来临时保存并恢复各种全局变量，包括命令行选项、调试参数，以及性能参数；也可以用来安装和移除钩子程序来让产品代码调用测试代码；或者将产品代码设置

为少见却很重要的状态，比如超时、错误，甚至是交叉并行执行。

以这种方式来使用全局变量是安全的，因为 `go test` 一般不会并发执行多个测试。

11.2.4 外部测试包

考虑包 `net/url`，这个包提供了 URL 解析功能，还有 `net/http`，这个包提供了 Web 服务器和 HTTP 客户端库。如我们所知，高级的 `net/http` 包依赖于低级的 `net/url` 包。然而，`net/url` 包中的一个测试是用来演示 URL 和 HTTP 库之间进行交互的例子。换句话说，低级别包的测试导入了高级别包。

在 `net/url` 包中声明这个测试函数会导致包循环引用，如图 11-1 中向上的箭头所示，但是 10.1 节讲过，Go 规范禁止循环引用。

我们通过将这个测试函数定义在外部测试包中来解决这个问题。也就是说，在 `net/url` 目录中，有一个文件，它的包声明是 `url_test`。这个额外的后缀 `_test` 告诉 `go test` 工具，它应该单独地编译一个包，这个包仅包含这些文件，然后运行它的测试。为了帮助理解，你可以认为这个外部测试包的导入路径是 `net/url_test`，但事实上它无法通过该路径以及其他任何路径导入。

由于外部测试在一个单独的包里面，因此它们也可以引用一些依赖于被测试包的帮助包；这个是包内测试无法做到的。从设计层次来看，外部测试包逻辑上在它所依赖的两个包之上，如图 11-2 所示。

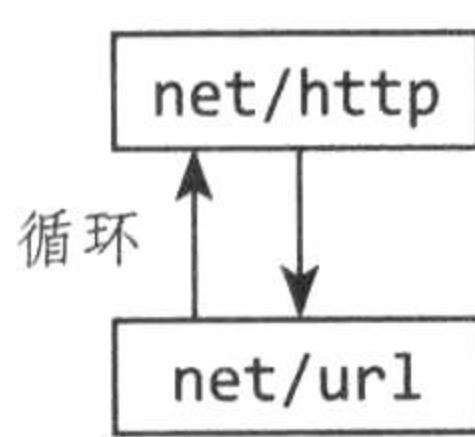


图 11-1 `net/url` 的一个测试依赖 `net/http`

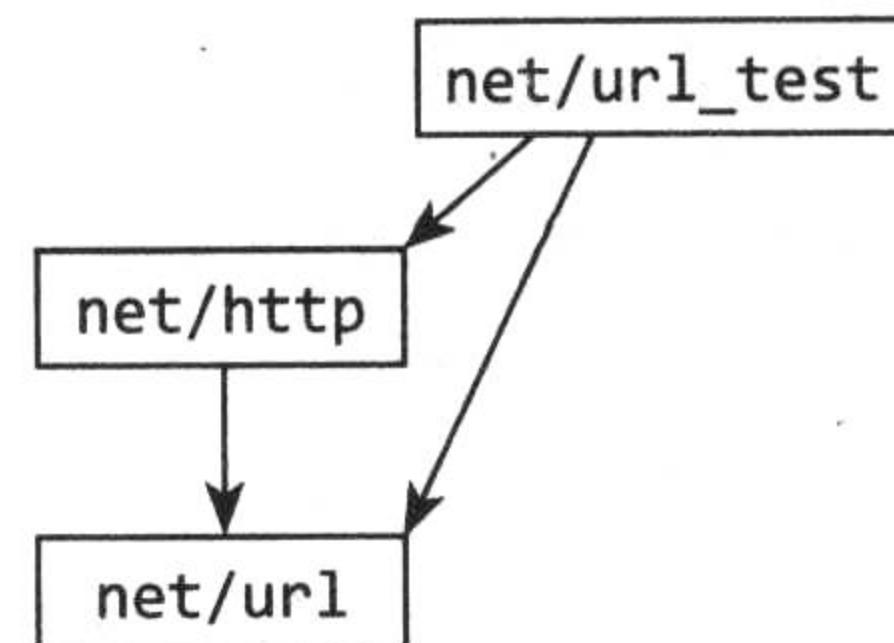


图 11-2 外部测试包打破了循环引用

为了避免包循环导入，外部测试包允许测试用例，尤其是集成测试用例（用来测试多个组件的交互），自由地导入其他的包，就像一个应用程序那样。

可以使用 `go list` 工具来汇总一个包目录中哪些是产品代码，哪些是包内测试以及哪些是外部测试。我们用 `fmt` 包来作为例子。`GoFiles` 是包含产品代码的文件列表，这些文件是 `go build` 命令将编译进你程序的代码。

```
$ go list -f={{.GoFiles}} fmt
[doc.go format.go print.go scan.go]
```

`TestGoFiles` 是也属于包 `fmt` 的文件列表，但是这些以 `_test.go` 结尾的文件仅在编译测试的时候才会使用。

```
$ go list -f={{.TestGoFiles}} fmt
[export_test.go]
```

包的测试用例通常位于这些文件中，而 `fmt` 包却并不是这样。后面会解释文件 `export_test.go` 的作用。

`XTestGoFiles` 是包外部测试文件列表，比如 `fmt_test`，所以这些文件必须引用 `fmt` 包才能使用它。同样，它们仅用在测试过程中。

```
$ go list -f={{.XTestGoFiles}} fmt
[fmt_test.go scan_test.go stringer_test.go]
```

有时候，外部测试包需要对待测试包拥有特殊的访问权限，例如为了避免循环引用，一个白盒测试必须存在于一个单独的包中。在这种情况下，我们使用一种小技巧：在包内测试文件 _test.go 中添加一些函数声明，将包内部的功能暴露给外部测试。这些文件也因此为测试提供了包的一个“后门”。如果一个源文件存在的唯一目的就在于此，并且自己不包含任何测试，它们一般称作 export_test.go。

例如，包 `fmt` 的实现需要功能 `unicode.isSpace` 作为 `fmt.Scanf` 的一部分。为了避免创建不合理的依赖，`fmt` 没有导入 `unicode` 包及其巨大的数据表，而是包含了一个更加简单的实现 `isSpace`。

为了确保 `fmt.isSpace` 和 `unicode.IsSpace` 的功能一致，`fmt` 添加了一个测试。这个是包外部测试，所以它不能够直接访问 `isSpace`，所以 `fmt` 通过定义一个可导出变量来引用 `isSpace` 函数。这个就是 `fmt` 包中文件 `export_test.go` 的内容。

```
package fmt

var IsSpace = isSpace
```

这个测试文件没有定义测试；它仅定义了一个导出符号 `fmt.IsSpace` 用来给外部测试使用。这个技巧在任何外部测试需要使用白盒测试技术的时候都可以使用。

11.2.5 编写有效测试

很多初学 Go 的人都对 Go 测试框架的极简主义感到惊奇。其他语言的框架提供了识别测试函数的机制（一般通过反射或者元数据），在测试前后执行测试“启动”和“销毁”的钩子，以及为常规的断言、值比较、错误消息格式化和终止失败的测试（一般通过抛出异常的方式）提供工具方法的库。虽然这些机制可以让测试编写更加精细，但是导致的结果是这些测试看上去像是用一门其他的语言编写的。而且，尽管它们也能准确地报告测试结果是 `PASS` 还是 `FAIL`，但是报告的方式对可怜的维护者来讲或许并不友好，比如模糊的错误消息“`assert: 0==1`”或者一页页的跟踪栈信息。

Go 对测试的看法是完全不同的。它期望测试的编写者自己来做大部分的工作，通过定义函数来避免重复，就像他们为普通程序所做的那样。测试的过程不是死记硬背地填表格；测试也是有用户界面的，虽然它的用户也是它的维护者。一个好的测试不会在发生错误时崩溃而是输出该问题一个简洁、清晰的现象描述，以及其他与上下文相关的信息。理想情况下，维护者不需要再通过阅读源代码来探究测试失败的原因。一个好的测试不应该在发现一次测试失败后就终止，而是要在一次运行中尝试报告多个错误，因为错误发生的方式本身会揭露错误的原因。

下面的断言函数比较两个值，构建一条一般的错误消息，并且停止程序。这个测试是正确的并且易于使用，但是当它运行失败的时候，所输出的错误消息毫无用处。它没有解决一个重要问题，那就是提供一个好的用户界面。

```
import (
    "fmt"
    "strings"
    "testing"
)
```

```
// 一个糟糕的断言函数
func assertEquals(x, y int) {
    if x != y {
        容机(fmt.Sprintf("%d != %d", x, y))
    }
}

func TestSplit(t *testing.T) {
    words := strings.Split("a:b:c", ":")
    assertEquals(len(words), 3)
    // ...
}
```

在这种情况下，断言函数过早抽象，把这个特殊测试的失败当作两个整数之间的不同。我们丧失了提供有意义语境的机会。我们可以通过从具体的信息开始来提供一个更好的错误输出，如下面的示例所示。只有重复一次的模式出现在一组测试中时才可以引入抽象。

```
func TestSplit(t *testing.T) {
    s, sep := "a:b:c", ":"
    words := strings.Split(s, sep)
    if got, want := len(words), 3; got != want {
        t.Errorf("Split(%q, %q) returned %d words, want %d",
            s, sep, got, want)
    }
    // ...
}
```

现在测试函数报告了调用的函数名称、它的输入以及输出表示的含义；它显式地区别出实际值和期望值；并且即使在测试失败的情况下，它也可以继续执行。当我们写出这种测试之后，下一步自然不是定义一个函数来替代整个 `if` 语句，而是在一个循环中执行这个测试，其中 `s`、`sep`、`want` 的值每次都不同，使用的是如 `IsPalindrome` 那样基于表的测试方式。

前面的例子并不需要任何工具函数，但是这并不阻止我们在为了使得代码更简洁的情况下引入工具函数（13.3 节会给出一个工具函数 `reflect.DeepEqual`）。一个好测试的关键是首先实现你所期望的具体行为，并且仅在这个时候再使用工具函数来使代码简洁并且避免重复。好的结果很少是从抽象的、通用的测试函数开始的。

练习 11.5：扩展 `TestSplit` 函数，以使用基于表的输入和期望输出。

11.2.6 避免脆弱的测试

如果一个应用在遇到新的合法输入的情况下经常崩溃，那么这个程序是有缺陷的；如果在程序发生可靠的改动的时候测试用例奇怪地失败了，那么这个测试用例也是脆弱的。如同有缺陷的程序会让用户感到沮丧，脆弱的测试也会激怒它的维护者。最脆弱的测试在产品代码发生任何改动的时候都会失败，无论这些改动是好是坏，这些测试通常称为变化探测器（change detector）或现状探测器（status quo test），并且处理它们花费的时间将会使得它们曾经带来的好处消失殆尽。

如果一个被测试的函数产生了一个复杂的输出，比如一个长字符串、一个详细的数据结构或者一个文件，比较吸引人的做法是检查输出完全匹配在测试阶段预期的一些“幸运值”。然而，随着程序的进化，输出的部分内容或许以好的方式将会发生变化，但是会发生改变。而且，不仅仅是输出会变化，拥有复杂输入的函数经常崩溃，由于测试中使用的输入不再合法。

避免写出脆弱测试的最简单方法就是仅检查你关心的属性。首先测试程序中越来越简单和稳定的接口，然后是它们的内部函数。选择性地设置断言。例如不要检查字符串精确匹配，而是寻找在程序进化过程中不会发生改变的子串。通常情况下，很值得写一个稳定的函数来从复杂的输出中提取核心内容，只有这样断言才会可靠。虽然看起来预先会做很多工作，但是这是值得的，否则这些时间就会被花在修复那些奇怪地失败的测试上面。

11.3 覆盖率

从本质上讲，测试从来不会结束。著名计算机科学家 Edsger Dijkstra 说，“测试旨在发现 bug，而不是证明其不存在。”无论有多少测试都无法证明一个包是没有 bug 的。在最好的情况下，它们增强了我们的信心，这些包是可以在很多重要的场景下使用的。

一个测试套件覆盖待测试包的比例称为测试的覆盖率。覆盖率无法直接通过数量来衡量，任何事情都是动态的，即使最微小的程序都无法精确地测量。但还是有办法帮助我们将测试精力放到最有潜力的地方。

语句覆盖率是一种最简单的且广泛使用的方法之一。一个测试套件的语句覆盖率是指部分语句在一次执行中至少执行一次。本节将使用 Go 的 `cover` 工具，这个工具被集成到了 `go test` 中，用来衡量语句覆盖率并帮助识别测试之间的明显差别。

下面的代码是基于表的测试，用来测试第 7 章中创建的表达式求值器。

```
gopl.io/ch7/eval

func TestCoverage(t *testing.T) {
    var tests = []struct {
        input string
        env   Env
        want  string // Parse/Check 返回的错误或者 Eval 返回的结果
    }{
        {"x % 2", nil, "unexpected '%'"},
        {"!true", nil, "unexpected '!'"},
        {"log(10)", nil, `unknown function "log"`},
        {"sqrt(1, 2)", nil, "call to sqrt has 2 args, want 1"},
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
    }

    for _, test := range tests {
        expr, err := Parse(test.input)
        if err == nil {
            err = expr.Check(map[Var]bool{})
        }
        if err != nil {
            if err.Error() != test.want {
                t.Errorf("%s: got %q, want %q", test.input, err, test.want)
            }
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        if got != test.want {
            t.Errorf("%s: %v => %s, want %s",
                    test.input, test.env, got, test.want)
        }
    }
}
```

首先，检测测试可以通过：

```
$ go test -v -run=Coverage gopl.io/ch7/eval
==== RUN TestCoverage
--- PASS: TestCoverage (0.00s)
PASS
ok      gopl.io/ch7/eval    0.011s
```

这个命令输出了覆盖工具的使用方法：

```
$ go tool cover
Usage of 'go tool cover':
Given a coverage profile produced by 'go test':
  go test -coverprofile=c.out

Open a web browser displaying annotated source code:
  go tool cover -html=c.out
...
```

命令 `go tool` 运行 Go 工具链里面的一个可执行文件。这些程序位于 `$GOROOT/pkg/tool/${GOOS}_${GOARCH}`。多亏了 `go build` 工具，我们很少直接调用它们。

现在带上 `-coverprofile` 标记来运行测试：

```
$ go test -run=Coverage -coverprofile=c.out gopl.io/ch7/eval
ok      gopl.io/ch7/eval    0.032s  coverage: 68.5% of statements
```

这个标记通过检测产品代码，启用了覆盖数据收集。也就是说，它修改了源代码的副本，这样在每个语句块执行之前，设置一个布尔变量，每个语句块都对应一个变量。在修改的程序退出之前，它将每个变量的值都写入到指定的 `c.out` 日志文件中并且输出被执行语句的汇总信息（如果你只需要汇总信息，那么使用 `go test -cover`）。

如果 `go test` 命令指定了 `-convermode=count` 标记，每个语句块的检测会递增一个计数器而不是设置布尔量。关于每个块的执行次数的日志使得量化比较成为可能，可由此识别出执行频率较高的“热块”或者相反的“冷块”。

在生成数据之后，我们运行 `cover` 工具，来处理生成的日志，生成一个 HTML 报告，并在一个新的浏览器窗口打开它（见图 11-3）。

```
$ go tool cover -html=c.out
```

界面中，每个用绿色（图中显示为浅灰色）标记的语句块表示它被覆盖了，而红色（图中为加阴影的深灰色）的则表示它没有被覆盖。为了清晰起见，我们给红色的文字加了阴影。我们可以立即看到，这里的输入都没有执行一元操作符 `Eval` 方法。如果添加一个新的测试用例到表格中并且重新运行前面的两条命令，一元表达式代码将变成绿色。

```
{"+x * -x", eval.Env{"x": 2}, "-4"}
```

然而，两行 `panic` 语句仍然是红色。这个并不奇怪，因为这些代码不应该执行到。

实现语句的 100% 覆盖听上去很宏伟，但是在实际情况下这并不可行，也不会行之有效。因为语句得以执行并不意味着这是没有 bug 的，拥有复杂表达式的语句块必须使用不同的输入执行多次来覆盖相关用例。有一些语句（如上面的 `panic` 语句）就永远不会被执行到。其他的（比如处理少见错误的代码）也很难检测并且实际上也很少会执行。测试基本上是实用主义行为，在编写测试的代价和本可以通过测试避免的错误造成的代价之间进行平衡。覆盖工具可以帮助识别最薄弱的点，但是和编程一样，设计好的测试用例通常需要一丝不苟的精神。

```

coverage.html
file:///home/gopher/gobook/coverage.html
gopl.io/ch7/eval/eval.go (58.8%) not tracked not covered covered

func (u unary) Eval(env Env) float64 {
    switch u.op {
    case '+':
        return +u.x.Eval(env)
    case '-':
        return -u.x.Eval(env)
    }
    panic(fmt.Sprintf("unsupported unary operator: %q", u.op))
}

func (b binary) Eval(env Env) float64 {
    switch b.op {
    case '+':
        return b.x.Eval(env) + b.y.Eval(env)
    case '-':
        return b.x.Eval(env) - b.y.Eval(env)
    case '*':
        return b.x.Eval(env) * b.y.Eval(env)
    case '/':
        return b.x.Eval(env) / b.y.Eval(env)
    }
    panic(fmt.Sprintf("unsupported binary operator: %q", b.op))
}

```

图 11-3 一个覆盖率报告

11.4 Benchmark 函数

基准测试就是在一定工作负载之下检测程序性能的一种方法。在 Go 里面，基准测试函数看上去像一个测试函数，但是前缀是 `Benchmark` 并且拥有一个 `*testing.B` 参数用来提供大多数和 `*testing.T` 相同的方法，额外增加了一些与性能检测相关方法。它还提供了一个整型成员 `N`，用来指定被检测操作的执行次数。

这里是 `IsPalindrome` 函数的基准测试，它在一个循环中调用了 `IsPalindrome` 共 `N` 次。

```

import "testing"

func BenchmarkIsPalindrome(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IsPalindrome("A man, a plan, a canal: Panama")
    }
}

```

我们使用下面的命令执行它。和测试不同，默认情况下不会运行任何基准测试。标记 `-bench` 的参数指定了要运行的基准测试。它是一个匹配 `Benchmark` 函数名称的正则表达式，它的默认值不匹配任何函数。模式“.”使它匹配包 `word` 中所有的基准测试函数，因为这里只有一个基准测试函数，所以和指定 `-bench=IsPalindrome` 效果一样。

```

$ cd $GOPATH/src/gopl.io/ch11/word2
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000          1035 ns/op
ok      gopl.io/ch11/word2      2.179s

```

基准测试名称的数字后缀 8 表示 `GOMAXPROCS` 的值，这个对于并发基准测试很重要。

报告告诉我们每次 `IsPalindrome` 调用耗费 1.035ms，这个是 1 000 000 次调用的平均值。

因为基准测试运行器开始的时候并不清楚这个操作的耗时长短，所以开始的时候它使用了比较小的 N 值来做检测，然后为了检测稳定的运行时间，推断出足够大的 N 值。

使用基准测试函数来实现循环而不是在测试驱动程序中调用代码的原因是，在基准测试函数中在循环外面可以执行一些必要的初始化代码并且这段时间不加到每次迭代的时间中。如果初始化代码干扰了结果，参数 `testing.B` 提供了方法用来停止、恢复和重置计时器，但是这些方法很少用到。

既然有了基准测试和功能测试，这时就很容易想到让程序更快一点。或许最明显的优化是使得 `IsPalindrome` 函数的第二次循环在中间停止检测，以避免比较两次：

```
n := len(letters)/2
for i := 0; i < n; i++ {
    if letters[i] != letters[len(letters)-1-i] {
        return false
    }
}
return true
```

但是通常情况下，优化并不能总是带来期望的好处。这个优化在一次实验中仅带来了 4% 的性能提升。

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000          992 ns/op
ok      gopl.io/ch11/word2      2.093s
```

另外一个主意是为 `letters` 预分配一个容量足够大的数组，而不是通过连续的 `append` 调用来自加。像这样声明一个合适长度的数组 `letters`：

```
letters := make([]rune, 0, len(s))
for _, r := range s {
    if unicode.IsLetter(r) {
        letters = append(letters, unicode.ToLower(r))
    }
}
```

这个改进带来了 35% 的性能提升，另外基准测试运行器报告了 2 000 000 次运行时间的平均值。

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 2000000          697 ns/op
ok      gopl.io/ch11/word2      1.468s
```

如上面的例子所示，最快的程序通常是那些进行内存分配次数最少的程序。命令行标记 `-benchmem` 在报告中包含了内存分配统计数据。这里和优化之前的内存分配进行比较：

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    1000000  1026 ns/op  304 B/op  4 allocs/op
```

优化之后：

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    2000000   807 ns/op  128 B/op  1 allocs/op
```

在一次 `make` 调用中分配完全部所需的内存减少了 75% 的分配次数并且减少了一半的内

存分配。

这种基准测试告诉我们给定操作的绝对耗时，但是在很多情况下，引起关注的性能问题是两个不同操作之间的相对耗时。例如，如果一个函数需要 1ms 来处理 1000 个元素，那么它处理 10 000 个或者 100 万个元素需要多久呢？另外一个例子：I/O 缓冲区的最佳大小是多少。对一个应用使用一系列的大小进行基准测试可以帮助我们选择最小的缓冲区并带来最佳的性能表现。第三个例子：对于一个任务来讲，哪种算法表现最佳？对两个不同的算法使用相同的输入，在重要的或者具有代表性的工作负载下，进行基准测试通常可以显示出每个算法的优点和缺点。

性能比较函数只是普通的代码。它们的表现形式通常是带有一个参数的函数，被多个不同的 Benchmark 函数传入不同的值来调用，如下所示：

```
func benchmark(b *testing.B, size int) { /* ... */ }
func Benchmark10(b *testing.B) { benchmark(b, 10) }
func Benchmark100(b *testing.B) { benchmark(b, 100) }
func Benchmark1000(b *testing.B) { benchmark(b, 1000) }
```

参数 size 指定了输入的大小，每个 Benchmark 函数传入的值都不同但是在每个函数内部是一个常量。不要使用 b.N 作为输入的大小。除非把它当作固定大小输入的循环次数，否则该基准测试的结果毫无意义。

基准测试比较揭示的模式在程序设计阶段很有用处，但是即使程序正常工作了，我们也不会丢掉基准测试。随着的程序演变，或者它的输入增长了，或者它被部署在其他的操作系统上并拥有一些新特性，我们仍然可以重用基准测试来回顾当初的设计决策。

练习 11.6：编写基准测试来比较 2.6.2 节实现的 PopCount 和练习 2.4 和 2.5 的答案。在何种情况下，基于表的测试方法付出和收益均衡。

练习 11.7：使用大型伪随机输入，为 6.5 节中 *IntSet 的 Add、UnionWith 和其他的方法编写基准测试。你能让这些方法运行多快？单词长度的选择对性能具有什么影响？这个 IntSet 的性能比使用内置 map 类型实现的功能快多少？

11.5 性能剖析

基准测试对检测具体操作的性能很有用，但是当我们在尝试使得一个程序变得更快的时候，我们经常不知道从何做起。每个程序员都了解关于唐纳德·克努斯的不要过早优化的箴言，这句话出现在 1974 年的“Structured Programming with go to Statements”一文中。虽然经常被误解为性能并不重要，但是我们可以从原始的语境中得出如下信息：

毫无疑问对性能的崇拜会导致滥用。程序员们浪费了大量的时间来思考或担心他们非关键部分代码的执行速度，并且在考虑到程序的调试和维护的时候这些优化的尝试事实上会带来负面影响。我们必须忘记微小的性能提升，必须说在 97% 的情况下，过早优化是万恶之源。

然而我们不可以错过那关键的 3% 的情况。一个好的程序员不会因为这个就自满，明智的方法是他应该仔细地查看关键代码；当然仅在关键代码明确之后。通常情况下先入为主地认定程序哪些部分是关键代码是错误的，使用了检测工具的程序员会发现的普遍经验就是他们的直觉是错的。

当我们希望仔细地查看程序的速度时，发现关键代码的最佳技术就是性能剖析。性能剖

析是通过自动化手段在程序执行过程中基于一些性能事件的采样来进行性能评测，然后再从这些采样中推断分析，得到的统计报告就称作为性能剖析（profile）。

Go 支持很多种性能剖析方式，每一个都和一个不同方面的性能指标相关，但是它们都需要记录一些相关的事件，每一个都有一个相关的栈信息——在事件发生时活跃的函数调用栈。工具 `go test` 内置支持一些类别的性能剖析。

CPU 性能剖析识别出执行过程中需要 CPU 最多的函数。在每个 CPU 上面执行的线程都每隔几毫秒会定期地被操作系统中断，在每次中断过程中记录一个性能剖析事件，然后恢复正常执行。

堆性能剖析识别出负责分配最多内存的语句。性能剖析库对协程内部内存分配调用进行采样，因此每个性能剖析事件平均记录了分配的 512KB 内存。

阻塞性能剖析识别出那些阻塞协程最久的操作，例如系统调用，通道发送和接收数据，以及获取锁等。性能分析库在一个 goroutine 每次被上述操作之一阻塞的时候记录一个事件。

获取待测试代码的性能剖析报告很容易，只需要像下面一样指定一个标记即可。当一次使用多个标记的时候需要注意，获取性能分析报告的机制是当获取其中一个类别的报告时会覆盖掉其他类别的报告。

```
$ go test -cpuprofile=cpu.out  
$ go test -blockprofile=block.out  
$ go test -memprofile=mem.out
```

尽管具体的做法对于短暂的命令行工具和长时间运行的服务器程序有所不同，但是为非测试程序添加性能剖析支持也很容易。性能剖析对于长时间运行的程序尤其有用，所以 Go 运行时的性能剖析特性可以让程序员通过 `runtime API` 来启用。

在我们获取性能剖析结果后，我们需要使用 `pprof` 工具来分析它。这是 Go 发布包的标准部分，但是因为不经常使用，所以通过 `go tool pprof` 间接来使用它。它有很多特性和选项，但是基本的用法只有两个参数，产生性能剖析结果的可执行文件和性能剖析日志。

为了使得性能剖析过程高效并且节约空间，性能剖析日志里面没有包含函数名称而是使用它们的地址。这就意味着 `pprof` 工具需要可执行文件才能理解数据内容。虽然通常情况下 `go test` 工具在测试完成之后就丢弃了用于测试而临时产生的可执行文件，在性能剖析启用的时候，它保存并把可执行文件命名为 `foo.test`，其中 `foo` 是被测试包的名字。

下面的命令演示如何获取和显示简单的 CPU 性能剖析。我们选择了 `net/http` 包中的一个基准测试。通常情况下最好对我们关心的具有代表性的具体负载而构建的基准测试进行性能性能剖析。对测试用例进行基准测试永远没有代表性，这也是我们使用过滤器 `-run=None` 来禁用它们的原因。

```
$ go test -run=None -bench=ClientServerParallelTLS64 \  
-cpuprofile=cpu.log net/http  
PASS  
BenchmarkClientServerParallelTLS64-8 1000  
 3141325 ns/op 143010 B/op 1747 allocs/op  
ok    net/http      3.395s  
  
$ go tool pprof -text -nodecount=10 ./http.test cpu.log  
2570ms of 3590ms total (71.59%)  
Dropped 129 nodes (cum <= 17.95ms)
```

```
Showing top 10 nodes out of 166 (cum >= 60ms)
      flat  flat%  sum%  cum  cum%
1730ms 48.19% 48.19% 1750ms 48.75% crypto/elliptic.p256ReduceDegree
 230ms 6.41% 54.60% 250ms 6.96% crypto/elliptic.p256Diff
120ms 3.34% 57.94% 120ms 3.34% math/big.addMulVW
110ms 3.06% 61.00% 110ms 3.06% syscall.Syscall
 90ms 2.51% 63.51% 1130ms 31.48% crypto/elliptic.p256Square
 70ms 1.95% 65.46% 120ms 3.34% runtime.scanobject
 60ms 1.67% 67.13% 830ms 23.12% crypto/elliptic.p256Mul
 60ms 1.67% 68.80% 190ms 5.29% math/big.nat.montgomery
 50ms 1.39% 70.19% 50ms 1.39% crypto/elliptic.p256ReduceCarry
 50ms 1.39% 71.59% 60ms 1.67% crypto/elliptic.p256Sum
```

标记 `-text` 指定输出的格式，在这个例子中，首先出现的是一个文本表格，表格中每行一个函数，这些函数是根据消耗 CPU 最多的规则排序的“热函数”。标记 `-nodecount=10` 限制输出的结果共 10 行。对于较明显的性能问题，这个文本格式的输出或许已经足够暴露问题了。

这个性能剖析结果告诉我们对于这个特定的 HTTPS 基准测试的性能来说椭圆曲线密码学很重要。作为对比，如果一个性能剖析结果主要由 `runtime` 包中的内存分配函数控制，那么减少内存消耗是一个有价值的优化。

对于更微妙的问题，最好使用 `pprof` 的图形显示格式之一。这些格式需要 `GraphViz`，它可以从 www.graphviz.org 下载。标记 `-web` 渲染了程序中函数的有向图，并标记出函数的 CPU 消耗数值，用颜色突出“热函数”。

这里只讨论了 Go 的性能剖析工具的皮毛。要了解更多内容，可以阅读 Go 博客文章“[Go 程序性能检测](#)”。

11.6 Example 函数

被 `go test` 特殊对待的第三种函数就是示例函数，它们的名字以 `Example` 开头。它既没有参数也没有结果。这里有 `IsPalindrome` 的一个示例函数：

```
func ExampleIsPalindrome() {
    fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"))
    fmt.Println(IsPalindrome("palindrome"))
    // 输出：
    // true
    // false
}
```

示例函数有三个目的。首要目的是作为文档；比起乏味的描述，举一个好的例子是描述库函数功能最简洁直观的方式。示例也可以用来演示同一 API 中的类型和函数之间的交互，而文档则总是要重点介绍某个点，要么是类型，要么是函数或者整个包。和带注释的例子不同，示例函数是真实的 Go 代码，必须通过编译时检查，所以随着代码的进化它们也不会过时。

基于 `Example` 函数的后缀，基于 Web 的文档服务器 `godoc` 可以将示例函数和它所演示的函数或包相关联，因此 `ExampleIsPalindrome` 将和函数 `IsPalindrome` 的文档显示在一起，同时如果有一个示例函数就叫 `Example`，那么它就和包 `word` 关联在一起。

示例函数的第二个目的是它们是可以通过 `go test` 运行的可执行测试。如果一个示例函数最后包含一个类似这样的注释 `// 输出：`，测试驱动程序将执行这个函数并且检查输出到终

端的内容匹配这个注释中的文本。

示例函数的第三个目的是提供手动实验代码。在 golang.org 上的 godoc 文档服务器使用 Go Playground 来让用户在 Web 浏览器上面编辑和运行每个示例函数，如图 11-4 所示。这个通常是了解特定函数功能或者了解语言特性最快捷的方法。

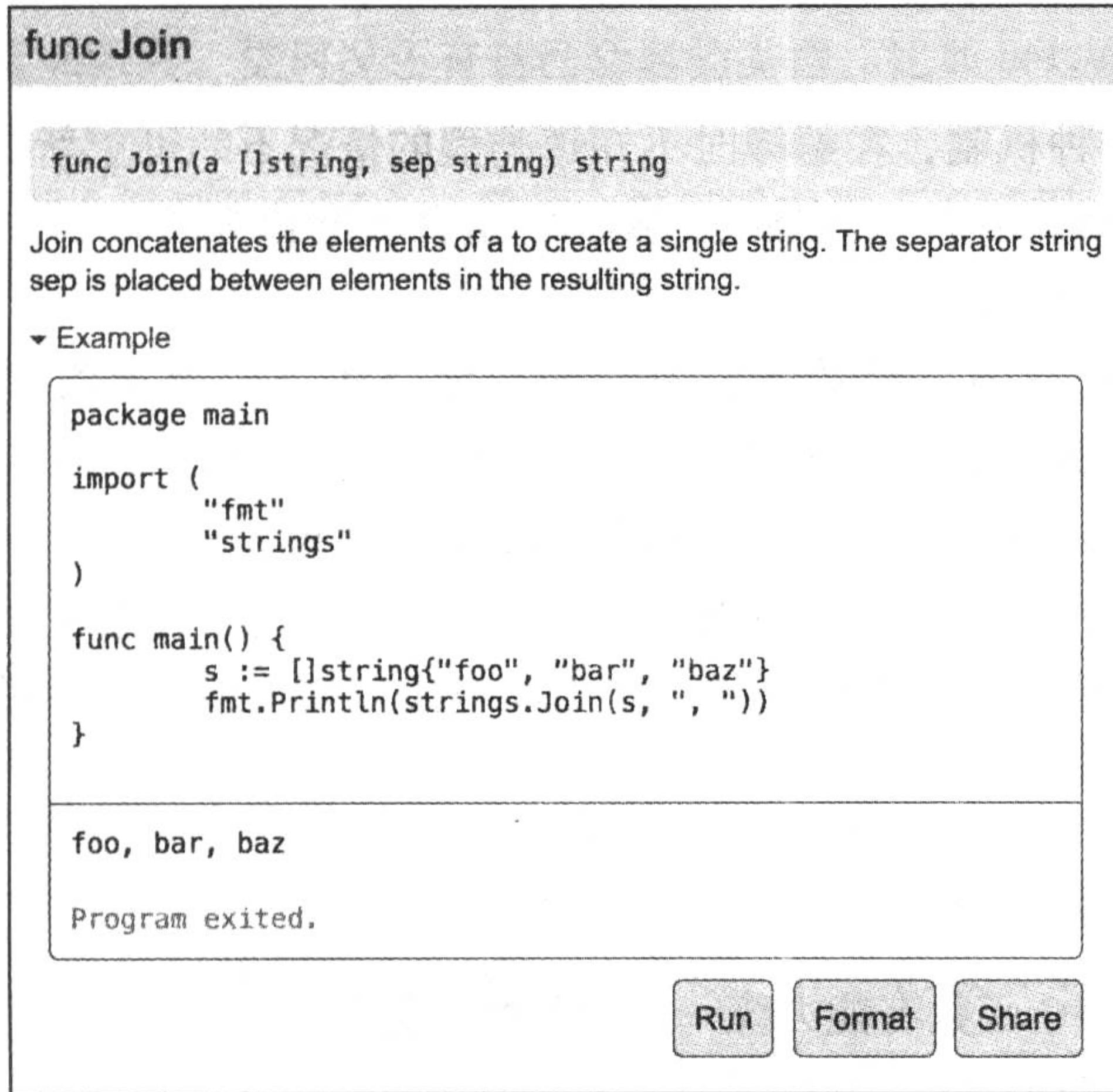


图 11-4 `strings.Join` 在 godoc 中的交互式例子

本书最后两章讲解 `reflect` 包和 `unsafe` 包，这两个包很少会有 Go 程序员使用——当然也很少会需要用到。如果读到这里，你还没有写过任何 Go 程序，现在就可以开始了。

反 射

Go 语言提供了一种机制，在编译时不知道类型的情况下，可更新变量、在运行时查看值、调用方法以及直接对它们的布局进行操作，这种机制称为反射（reflection）。反射也让我们可以把类型当做头等值。

本章将探讨 Go 语言的反射功能以及它如何增强语言的表达能力，特别是它在实现两个重要 API 中的关键作用。这两个 API 分别是 `fmt` 包提供的字符串格式化功能，以及 `encoding/json` 和 `encoding/xml` 这种包提供的协议编码功能。反射在 `text/template` 和 `html/template` 包提供的模板机制（参见 4.6 节）中也很重要。另外，反射的推导比较复杂，也不是为了随意使用设计的，因此尽管这些包使用反射来实现，但它们并没有在自己的 API 中暴露反射。

12.1 为什么使用反射

有时我们需要写一个函数有能力统一处理各种值类型的函数，而这些类型可能无法共享同一个接口，也可能布局未知，也有可能这个类型在我们设计函数时还不存在，甚至这个类型会同时存在上面三种问题。

一个熟悉的例子是 `fmt.Printf` 中的格式化逻辑，它可以输出任意类型的任意值，甚至是用户自定义的一个类型。让我们先尝试用我们已学到的知识来实现一个类似的函数。为了简化起见，该函数只接受一个参数，并且与 `fmt.Sprint` 一样返回一个字符串，所以我们称这个函数为 `Sprint`。

我们先用一个类型分支来判断这个参数是否定义了一个 `String` 方法，如果已定义则直接调用它。然后添加一些 `switch` 分支来判断参数的动态类型是否是基本类型（比如 `string`、`int`、`bool` 等），再对每种类型采用不同的格式化操作。

```
func Sprint(x interface{}) string {
    type stringer interface {
        String() string
    }
    switch x := x.(type) {
    case stringer:
        return x.String()
    case string:
        return x
    case int:
        return strconv.Itoa(x)
    // ...对 int16、uint32 等类型做类似的处理
    case bool:
        if x {
            return "true"
        }
        return "false"
```

```
default:  
    // array、chan、func、map、pointer、slice、struct  
    return "???"  
}  
}
```

但我们如何处理类似 `[]float64`、`map[string][]string` 的其他类型呢？可以添加更多的分支，但这样的类型有无限种。更何况还有自己命名的类型，比如 `url.Values`？因为即使我们有一个分支来处理 `map[string][]string` (`url.Values` 的底层类型)，这个分支仍然不会处理 `url.Values`，因为这两个类型不是完全一致的。更何况根本不可能引入 `url.Values` 的处理分支？因为这会导致库依赖于库的客户（即循环引用）。

当我们无法透视一个未知类型的布局时，这段代码就无法继续，这时我们就需要反射了。

12.2 reflect.Type 和 reflect.Value

反射功能由 `reflect` 包提供，它定义了两个重要的类型：`Type` 和 `Value`。`Type` 表示 Go 语言的一个类型，它是一个有很多方法的接口，这些方法可以用来识别类型以及透视类型的组成部分，比如一个结构的各个字段或者一个函数的各个参数。`reflect.Type` 接口只有一个实现，即类型描述符（见 7.5 节），接口值中的动态类型也是类型描述符。

`reflect.TypeOf` 函数接受任何的 `interface{}` 参数，并且把接口中的动态类型以 `reflect.Type` 形式返回。

```
t := reflect.TypeOf(3) // 一个 reflect.Type  
fmt.Println(t.String()) // "int"  
fmt.Println(t) // "int"
```

上面的 `TypeOf(3)` 调用把数值 3 赋给 `interface{}` 参数。回想一下 7.5 节的内容，把一个具体值赋给一个接口类型时会发生一个隐式类型转换，转换会生成一个包含两部分内容的接口值：动态类型部分是操作数的类型（`int`），动态值部分是操作数的值（3）。

因为 `reflect.TypeOf` 返回一个接口值对应的动态类型，所以它返回总是具体类型（而不是接口类型）。比如下面的代码输出的是 `"*os.File"` 而不是 `"io.Writer"`。后面我们会看到如何让 `reflect.Type` 也表示一个接口类型。

```
var w io.Writer = os.Stdout  
fmt.Println(reflect.TypeOf(w)) // "*os.File"
```

注意，`reflect.Type` 满足 `fmt.Stringer`。因为输出一个接口值的动态类型在调试和日志中很常用，所以 `fmt.Printf` 提供了一个简写方式 `%T`，内部实现就使用了 `reflect.TypeOf`：

```
fmt.Printf("%T\n", 3) // "int"
```

`reflect` 包的另一个重要类型就是 `Value`。`reflect.Value` 可以包含一个任意类型的值。

`reflect.ValueOf` 函数接受任意的 `interface{}` 并将接口的动态值以 `reflect.Value` 的形式返回。与 `reflect.TypeOf` 类似，`reflect.ValueOf` 的返回值也都是具体值，不过 `reflect.Value` 也可以包含一个接口值。

```
v := reflect.ValueOf(3) // 一个 reflect.Value  
fmt.Println(v) // "3"  
fmt.Printf("%v\n", v) // "3"  
fmt.Println(v.String()) // 注意: "<int Value>"
```

另一个与 `reflect.Type` 类似的是，`reflect.Value` 也满足 `fmt.Stringer`，但除非 `Value` 包含的是一个字符串，否则 `String` 方法的结果仅仅暴露了类型。通常，你需要使用 `fmt` 包的 `%v` 功能，它对 `reflect.Value` 会进行特殊处理。

调用 `Value` 的 `Type` 方法会把它的类型以 `reflect.Type` 方式返回：

```
t := v.Type()          // 一个 reflect.Type
fmt.Println(t.String()) // "int"
```

`reflect.ValueOf` 的逆操作是 `reflect.Value.Interface` 方法。它返回一个 `interface{}` 接口值，与 `reflect.Value` 包含同一个具体值。

```
v := reflect.ValueOf(3) // a reflect.Value
x := v.Interface()    // an interface{}
i := x.(int)           // an int
fmt.Printf("%d\n", i)  // "3"
```

`reflect.Value` 和 `interface{}` 都可以包含任意的值。二者的区别是空接口 (`interface{}`) 隐藏了值的布局信息、内置操作和相关方法，所以除非我们知道它的动态类型，并用一个类型断言来渗透进去（上面的代码就用了类型断言），否则我们对所包含值能做的事情很少。作为对比，`Value` 有很多方法可以用来分析所包含的值，而不用知道它的类型。使用这些技术，我们可以第二次尝试写一个通用的格式化函数，它称为 `format.Any`。

不用类型分支，我们用 `reflect.Value` 的 `Kind` 方法来区分不同的类型。尽管有无限种类型，但类型的分类 (`kind`) 只有少数几种：基础类型 `Bool`、`String` 以及各种数字类型；聚合类型 `Array` 和 `Struct`；引用类型 `Chan`、`Func`、`Ptr`、`Slice` 和 `Map`、接口类型 `Interface`；最后还有 `Invalid` 类型，表示它们还没有任何值。（`reflect.Value` 的零值就属于 `Invalid` 类型。）

```
gopl.io/ch12/format
package format

import (
    "reflect"
    "strconv"
)

// Any 把任何值格式化为一个字符串
func Any(value interface{}) string {
    return formatAtom(reflect.ValueOf(value))
}

// formatAtom 格式化一个值，且不分析它的内部结构
func formatAtom(v reflect.Value) string {
    switch v.Kind() {
    case reflect.Invalid:
        return "invalid"
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return strconv.FormatUint(v.Uint(), 10)
    // ...为简化起见，省略了浮点数和复数的分支...
    case reflect.Bool:
        return strconv.FormatBool(v.Bool())
    case reflect.String:
        return strconv.Quote(v.String())
    case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
        return v.Type().String() + " 0x" +
            strconv.FormatUint(uint64(v.Pointer()), 16)
    }
}
```

```

    default: // reflect.Array, reflect.Struct, reflect.Interface
        return v.Type().String() + " value"
    }
}

```

到现在为止，该函数把每个值当做一个没有内部结构且不可分割的物体（所以才叫 `formatAtom`）。对于聚合类型（结构体和数组）以及接口，它只输出了值的类型；对于引用类型（通道、函数、指针、slice 和 map），它输出了类型和以十六进制表示的引用地址。这个结果仍然不够理想，但确实是一个很大的进步。因为 `Kind` 只关心底层实现，所以 `format.Any` 对命名类型的效果也很好。比如：

```

var x int64 = 1
var d time.Duration = 1 * time.Nanosecond
fmt.Println(format.Any(x)) // "1"
fmt.Println(format.Any(d)) // "1"
fmt.Println(format.Any([]int64{x})) // "[]int64 0x8202b87b0"
fmt.Println(format.Any([]time.Duration{d})) // "[]time.Duration 0x8202b87e0"

```

12.3 Display：一个递归的值显示器

接下来我们看一下如何改善组合类型的显示。这次，我们不再实现一个 `fmt.Sprint`，而是实现一个称为 `Display` 的调试工具函数，这个函数对给定的任意一个复杂值 `x`，输出这个复杂值的完整结构，并对找到的每个元素标上这个元素的路径。下面先看一个例子。

```

e, _ := eval.Parse("sqrt(A / pi)")
Display("e", e)

```

在上面的调用中，`Display` 的参数是一个从表达式求值器生成的语法树（参考 7.9 节）。`Display` 的输出如下所示：

```

Display e (eval.call):
e.fn = "sqrt"
e.args[0].type = eval.binary
e.args[0].value.op = 47
e.args[0].value.x.type = eval.Var
e.args[0].value.x.value = "A"
e.args[0].value.y.type = eval.Var
e.args[0].value.y.value = "pi"

```

我们应当尽可能避免在包的 API 里边暴露反射相关的内容。我们将定义一个未导出的函数 `display` 来做真正的递归处理，再暴露 `Display`，而 `Display` 则只是一个简单的封装，并且接受一个 `interface{}` 参数：

gopl.io/ch12/display

```

func Display(name string, x interface{}) {
    fmt.Printf("Display %s (%T):\n", name, x)
    display(name, reflect.ValueOf(x))
}

```

在 `display` 中，我们使用之前定义的 `formatAtom` 函数来输出基础值（基础类型、函数和通道），使用 `reflect.Value` 的一些方法来递归展示复杂类型的每个组成部分。当递归深入时，`path` 字符串（之前用来表示起始值，比如 "e"）会增长，以表示如何找到当前值（比如 "`e.args[0].value`"）。

因为我们不用再假装正在实现 `fmt.Sprint`，所以接下来我们将使用 `fmt` 包来简化该示例：

```
func display(path string, v reflect.Value) {
    switch v.Kind() {
    case reflect.Invalid:
        fmt.Printf("%s = invalid\n", path)
    case reflect.Slice, reflect.Array:
        for i := 0; i < v.Len(); i++ {
            display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))
        }
    case reflect.Struct:
        for i := 0; i < v.NumField(); i++ {
            fieldPath := fmt.Sprintf("%s.%s", path, v.Type().Field(i).Name)
            display(fieldPath, v.Field(i))
        }
    case reflect.Map:
        for _, key := range v.MapKeys() {
            display(fmt.Sprintf("%s[%s]", path,
                formatAtom(key)), v.MapIndex(key))
        }
    case reflect.Ptr:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            display(fmt.Sprintf("( *%s )", path), v.Elem())
        }
    case reflect.Interface:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            fmt.Printf("%s.type = %s\n", path, v.Elem().Type())
            display(path+".value", v.Elem())
        }
    default: // 基本类型、通道、函数
        fmt.Printf("%s = %s\n", path, formatAtom(v))
    }
}
```

接下来将对这些分支逐一讲解。

slice 与数组：两者的逻辑一致。`Len` 方法会返回 slice 或者数组中元素的个数，`Index(i)` 会返回第 `i` 个元素，返回的元素类型为 `reflect.Value`（如果 `i` 越界会崩溃）。这两个方法与内置的 `len(a)` 和 `a[i]` 序列操作类似。在每个序列元素上递归调用了 `display` 函数，只是在路径后边加上了 "[i]"。

尽管 `reflect.Value` 有很多方法，但对于每个值，只有少量的方法可以安全调用。比如，`Index` 方法可以在 `Slice`、`Array`、`String` 类型的值上安全调用，但对于其他类型则会崩溃。

结构体：`NumField` 方法可以报告结果中的字段数，`Field(i)` 会返回第 `i` 个字段，返回的字段类型为 `reflect.Value`。字段列表包括了从匿名字段中做了类型提升的字段。要追加一个类似 ".f" 的字段选择标记到路径中，我们必须先获得结构体的 `reflect.Type` 才能获到第 `i` 个字段的名称。

map : `MapKeys` 方法返回一个元素类型为 `reflect.Value` 的 slice，每个元素都是一个 map 的键。与平常遍历 map 的结果类似，顺序是不固定的。`MapIndex(key)` 返回 `key` 对应的值。我们追加下标记号 "[key]" 到路径中。（此处忽略了一些情形。map 的键类型有可能超出 `formatAtom` 能处理好的类型，比如数组、结构体、接口都可以是合法的字典键。在练习 12.1 中会有输出完整键的内容。）

指针: `Elem` 方法返回指针指向的变量，同样也是以 `reflect.Value` 类型返回。这个方法在指针是 `nil` 时也能正确处理，但返回的结果属于 `Invalid` 类型，所以我们用 `IsNil` 来显式检测空指针，方便输出一条更合适的消息。为了避免二义性，在路径前加了一个 “`*`”，外边再加上一对圆括号。

接口: 我们再次使用 `IsNil` 来判断接口是否为空，如果非空，我们通过 `v.Elem()` 来获取动态值，进一步输出它的类型和值。

既然 `Display` 函数完成了，接下来我们就实际使用一下。下面的 `Movie` 类型引自 4.5 节，但略有修改：

```
type Movie struct {
    Title, Subtitle string
    Year            int
    Color           bool
    Actor           map[string]string
    Oscars          []string
    Sequel          *string
}
```

下面声明这个类型的一个值，并查看 `Display` 如何处理这个值：

```
strangelove := Movie{
    Title:      "Dr. Strangelove",
    Subtitle:   "How I Learned to Stop Worrying and Love the Bomb",
    Year:       1964,
    Color:      false,
    Actor: map[string]string{
        "Dr. Strangelove":           "Peter Sellers",
        "Grp. Capt. Lionel Mandrake": "Peter Sellers",
        "Pres. Merkin Muffley":       "Peter Sellers",
        "Gen. Buck Turgidson":        "George C. Scott",
        "Brig. Gen. Jack D. Ripper":  "Sterling Hayden",
        `Maj. T.J. "King" Kong`:     "Slim Pickens",
    },
    Oscars: []string{
        "Best Actor (Nomin.)",
        "Best Adapted Screenplay (Nomin.)",
        "Best Director (Nomin.)",
        "Best Picture (Nomin.)",
    },
}
```

调用 `Display("strangelove", strangelove)` 会输出：

```
Display strangelove (display.Movie):
strangelove.Title = "Dr. Strangelove"
strangelove.Subtitle = "How I Learned to Stop Worrying and Love the Bomb"
strangelove.Year = 1964
strangelove.Color = false
strangelove.Actor["Gen. Buck Turgidson"] = "George C. Scott"
strangelove.Actor["Brig. Gen. Jack D. Ripper"] = "Sterling Hayden"
strangelove.Actor["Maj. T.J. \"King\" Kong"] = "Slim Pickens"
strangelove.Actor["Dr. Strangelove"] = "Peter Sellers"
strangelove.Actor["Grp. Capt. Lionel Mandrake"] = "Peter Sellers"
strangelove.Actor["Pres. Merkin Muffley"] = "Peter Sellers"
strangelove.Oscars[0] = "Best Actor (Nomin.)"
strangelove.Oscars[1] = "Best Adapted Screenplay (Nomin.)"
strangelove.Oscars[2] = "Best Director (Nomin.)"
strangelove.Oscars[3] = "Best Picture (Nomin.)"
strangelove.Sequel = nil
```

我们可以使用 `Display` 来显示标准库类型的内部结构，比如 `*os.File`:

```
Display("os.Stderr", os.Stderr)
// 输出:
// 显示 os.Stderr (*os.File):
// (*os.Stderr).fd = 2
// (*os.Stderr).name = "/dev/stderr"
// (*os.Stderr).nepipe = 0
```

注意，即使非导出字段在反射下也是可见的。当然，这个例子的输出在各个平台上可能会有差异，也可能随着库的演进而改变。(毕竟把字段设为私有是由原因的！)

我们还可以把 `Display` 作用在 `reflect.Value` 上，并且观察它如何遍历 `*os.File` 的类型描述符的内部结构。调用 `Display("rV", reflect.ValueOf(os.Stderr))` 的输出如下所示，当然，每人得到的结果可能会有不同：

```
Display rV (reflect.Value):
(*rV.typ).size = 8
(*rV.typ).hash = 871609668
(*rV.typ).align = 8
(*rV.typ).fieldAlign = 8
(*rV.typ).kind = 22
(*rV.typ).string) = "*os.File"
(*rV.uncommonType.methods[0].name) = "Chdir"
(*rV.uncommonType.methods[0].mtyp.string) = "func() error"
(*rV.uncommonType.methods[0].typ.string) = "func(*os.File) error"
...
```

注意如下两个例子的差异：

```
var i interface{} = 3

Display("i", i)
// 输出:
// 显示 i (int):
// i = 3

Display("&i", &i)
// 输出:
// 显示 &i (*interface {}):
// (&i).type = int
// (&i).value = 3
```

在第一个例子中，`Display` 调用 `reflect.ValueOf(i)`，返回值的类型为 `Int`。正如 12.2 节提到的，因为 `reflect.ValueOf` 从接口值中提取值部分，所以它永远返回一个具体类型的 `Value`。

在第二个例子中，`Display` 调用 `reflect.ValueOf(&i)`，其返回值的类型为 `Ptr`，并且是一个指向 `i` 的指针。在 `Display` 函数的 `Ptr` 分支中，会调用这个值的 `Elem` 方法，返回一个代表变量 `i` 的 `Value`，其类型为 `Interface`。类似这种间接获得的 `Value` 可以代表任何值，包括接口。这时 `display` 函数递归调用自己，输出接口的动态类型和动态值。

在当前的实现中，`Display` 在对象图中存在循环引用时不会自行终止，比如处理一个首尾相接的链表时：

```
// 一个指向自己的结构体
type Cycle struct{ Value int; Tail *Cycle }
var c Cycle
c = Cycle{42, &c}
Display("c", c)
```

`Display` 输出了一个持续增长的展开式：

```
Display c (display.Cycle):
c.Value = 42
(*c.Tail).Value = 42
(**c.Tail).Tail.Value = 42
(**(*c.Tail).Tail).Tail = 42
...无穷无尽...
```

很多 Go 程序都至少包含一些循环引用的数据。让 `Display` 能鲁棒地处理这些循环引用要一些小技巧，需要记录所有曾经被访问过的引用，当然成本也不低。一个通用的解决方案需要 `unsafe` 语言特性，13.3 节将介绍这个特性。

循环引用在 `fmt.Sprintf` 中不构成一个大问题，因为它很少尝试输出整个结构体。比如，当它遇到一个指针时，它会输出指针的数字值，这样就打破了循环引用。但如果遇到一个 slice 或者 map 包含自身，它还是会卡住，只是不值得为了这种罕见的案例而去承担处理循环引用的成本。

练习 12.1：扩展 `Display`，让它可以处理 map 中键为结构体或者数组的情形。

练习 12.2：通过限制递归的层数，让 `Display` 能安全处理循环引用的数据结构。（在 13.3 节中，我们可以看到另外一个检测循环引用的方法。）

12.4 示例：编码 S 表达式

`Display` 是一个用于显示数据结构的调试例程，但是只要对它稍加修改，就可以用它来对任意 Go 对象进行编码或编排，使之成为适用于进程间通信的可移植记法中的消息。

正如我们在 4.5 节见到的，Go 的标准库支持各种格式，包括 JSON、XML 和 ASN.1。另一种广泛使用的格式是 Lisp 语言中的 S 表达式。与其他格式不同的是，S 表达式还没被 Go 标准库支持，这是因为尽管有几次标准化的尝试并存在很多实现，但它们仍然没有被广泛接受的严格定义。

在本节中，我们会定义一个包，它使用 S 表达式来编码任意的 Go 对象，这个 S 表达式需要支持下面的表达式：

42	integer
"hello"	string (转义方法与 Go一致)
foo	symbol (直接使用名字，不加引号)
(1 2 3)	list (用括号括起来的零个及以上元素)

布尔值一般用符号 `t` 表示真，用空列表 `()` 或者符号 `nil` 表示假，但为了简化起见，这个实现直接忽略布尔值。通道和函数也被忽略了，因为它们的状态对于反射来说是不透明的。这个实现还忽略了实数、复数和接口，在练习 12.3 中我们会加上这些支持。

我们将按如下思路来把 Go 语言的值编码为 S 表达式。整数和字符串的编码方式是显而易见的。空值直接编码为符号 `nil`，数组和 slice 则用列表记法来编码。

结构被编码为一个关于字段绑定（field binding）的列表，每个字段绑定都是一个两个元素的列表，其中第一个元素（使用符号）是字段名，第二个元素是字段值。`map` 也编码为元素对的列表，每个元素对都是 `map` 中一项的键和值。按照传统，S 表达式使用形式为 `(key . value)` 的单个构造单元（cons cell）来表示键值对，而不是用双元素的列表，但为了简化解码过程，我们将忽略带“.”的列表表示法。

编码用如下的单个递归调用函数 `encode` 来实现。它的结构与上一节的 `Display` 在本质上

是一致的：

```
gopl.io/ch12/sexpr

func encode(buf *bytes.Buffer, v reflect.Value) error {
    switch v.Kind() {
    case reflect.Invalid:
        buf.WriteString("nil")

    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        fmt.Fprintf(buf, "%d", v.Int())

    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        fmt.Fprintf(buf, "%d", v.Uint())

    case reflect.String:
        fmt.Fprintf(buf, "%q", v.String())

    case reflect.Ptr:
        return encode(buf, v.Elem())

    case reflect.Array, reflect.Slice: // (value ...)
        buf.WriteByte('(')
        for i := 0; i < v.Len(); i++ {
            if i > 0 {
                buf.WriteByte(' ')
            }
            if err := encode(buf, v.Index(i)); err != nil {
                return err
            }
        }
        buf.WriteByte(')')
    }

    case reflect.Struct: // ((name value) ...)
        buf.WriteByte('(')
        for i := 0; i < v.NumField(); i++ {
            if i > 0 {
                buf.WriteByte(' ')
            }
            fmt.Fprintf(buf, " (%s ", v.Type().Field(i).Name)
            if err := encode(buf, v.Field(i)); err != nil {
                return err
            }
            buf.WriteByte(')')
        }
        buf.WriteByte(')')

    case reflect.Map: // ((key value) ...)
        buf.WriteByte('(')
        for i, key := range v.MapKeys() {
            if i > 0 {
                buf.WriteByte(' ')
            }
            buf.WriteByte('(')
            if err := encode(buf, key); err != nil {
                return err
            }
            buf.WriteByte(' ')
        }
        buf.WriteByte(')')
```

```
    if err := encode(buf, v.MapIndex(key)); err != nil {
        return err
    }
    buf.WriteByte(')')
}
buf.WriteByte(')')

default: // float, complex, bool, chan, func, interface
    return fmt.Errorf("unsupported type: %s", v.Type())
}
return nil
}
```

Marshal 函数把上面的编码器封装成一个 API，它类似于其他 encoding/... 包里的 API：

```
// Marshal 把 Go 的值编码为 S 表达式的形式
func Marshal(v interface{}) ([]byte, error) {
    var buf bytes.Buffer
    if err := encode(&buf, reflect.ValueOf(v)); err != nil {
        return nil, err
    }
    return buf.Bytes(), nil
}
```

下面是 12.3 节的 strangelove 应用 Marshal 后的输出：

```
((Title "Dr. Strangelove") (Subtitle "How I Learned to Stop Worrying and Love the Bomb") (Year 1964) (Actor ((Grp. Capt. Lionel Mandrake" "Peter Sellers") ("Pres. Merkin Muffley" "Peter Sellers") ("Gen. Buck Turgidson" "George C. Scott") ("Brig. Gen. Jack D. Ripper" "Sterling Hayden") ("Maj. T.J. \ "King\ Kong" "Slim Pickens") ("Dr. Strangelove" "Peter Sellers")))) (Oscars ("Best Actor (Nomin.)" "Best Adapted Screenplay (Nomin.)" "Best Director (Nomin.)" "Best Picture (Nomin.)")) (Sequel nil))
```

整个输出都在一行且使用了最少的空格数，导致读起来很困难。根据 S 表达式的习惯手动格式化后的结果如下所示。把编写 S 表达式的美化打印器留作练习（有点挑战），从 gopl.io 上可以下载一个简单的版本。

```
((Title "Dr. Strangelove")
(Subtitle "How I Learned to Stop Worrying and Love the Bomb")
(Year 1964)
(Actor ((Grp. Capt. Lionel Mandrake" "Peter Sellers")
("Pres. Merkin Muffley" "Peter Sellers")
("Gen. Buck Turgidson" "George C. Scott")
("Brig. Gen. Jack D. Ripper" "Sterling Hayden")
("Maj. T.J. \ "King\ Kong" "Slim Pickens")
("Dr. Strangelove" "Peter Sellers")))
(Oscars ("Best Actor (Nomin.)"
"Best Adapted Screenplay (Nomin.)"
"Best Director (Nomin.)"
"Best Picture (Nomin.)"))
(Sequel nil))
```

与 fmt.Print、json.Marshal、Display 这些函数类似，sexpr.Marshal 在遇到循环应用的数据时也会无限循环。

12.6 节会概述 S 表达式解码函数的实现，但在那之前，我们需要先了解一下如何用反射来更新程序中的变量。

练习 12.3：实现 encode 函数缺失的功能。把布尔值编码为 t 和 nil，浮点数则用 Go 语

言的表示法，像 `1+2i` 这种复数则编码为 `#c(1.0 2.0)`。接口编码为成对的类型名和值，比如 `("[]int"(1 2 3))`，但要注意这个方法是有二义性的，因为 `reflect.Type.String` 方法可能会对不同的类型生成同样的字符串。

练习 12.4：修改 `encode` 函数，输出如上所示的美化后的 S 表达式。

练习 12.5：改写 `encode` 函数，从输出 S 表达式改为输出 JSON。使用标准库的解码器 `json.Unmarshal` 来测试编码器。

练习 12.6：改写 `encode` 函数，优化输出，如果字段值是其类型的零值则不须编码。

练习 12.7：参考 `json.encoder`（参见 4.5 节）的风格，创建一个 S 表达式编码器的流式 API。

12.5 使用 `reflect.Value` 来设置值

到现在为止，在程序中反射还只用来解析变量值。而本节的重点则是如何改变值。

回想一下 Go 语言的表达式，比如 `x`、`x.f[1]`、`*p` 这样的表达式表示一个变量，而 `x+1`、`f(2)` 之类的表达式则不表示变量。一个变量是一个可寻址的存储区域，其中包含了一个值，并且它的值可以通过这个地址来更新。

对 `reflect.Value` 也有一个类似的区分，某些是可寻址的，而其他的并非如此。比如如下的变量声明：

```
x := 2          // 值类型变量?
a := reflect.ValueOf(2) // 2      int    no
b := reflect.ValueOf(x) // 2      int    no
c := reflect.ValueOf(&x) // &x    *int   no
d := c.Elem()         // 2      int    yes (x)
```

`a` 里边的值是不可寻址的，它包含的仅仅是整数 2 的一个副本。`b` 也是如此。`c` 里边的值也是不可寻址的，它包含的是指针 `&x` 的一个副本。事实上，通过 `reflect.ValueOf(x)` 返回的 `reflect.Value` 都是不可寻址的。但 `d` 是通过对 `c` 中的指针提领得来的，所以它是可寻址的。可以通过这个方法，调用 `reflect.ValueOf(&x).Elem()` 来获得任意变量 `x` 可寻址的 `Value` 值。

可以通过变量的 `CanAddr` 方法来询问 `reflect.Value` 变量是否可寻址：

```
fmt.Println(a.CanAddr()) // "false"
fmt.Println(b.CanAddr()) // "false"
fmt.Println(c.CanAddr()) // "false"
fmt.Println(d.CanAddr()) // "true"
```

我们可以通过一个指针来间接获取一个可寻址的 `reflect.Value`，即使这个指针是不可寻址的。可寻址的常见规则都在反射包里边有对应项。比如，slice 的脚标表达式 `e[i]` 隐式地做了指针去引用，所以即使 `e` 是不可寻址的，这个表达式仍然是可寻址的。类似地，`reflect.ValueOf(e).Index(i)` 代表一个变量，尽管 `reflect.ValueOf(e)` 不是可寻址的，这个变量也是可寻址的。

从一个可寻址的 `reflect.Value()` 获取变量需要三步。首先，调用 `Addr()`，返回一个 `Value`，其中包含一个指向变量的指针，接下来，在这个 `Value` 上调用 `Interface()`，会返回一个包含这个指针的 `interface{}` 值。最后，如果我们知道变量的类型，我们可以使用类型断言来把接口内容转换为一个普通指针。之后就可以通过这个指针来更新变量了：

```
x := 2
d := reflect.ValueOf(&x).Elem()      // d 代表变量 x
px := d.Addr().Interface().(*int)    // px := &x
*px = 3                            // x = 3
fmt.Println(x)                      // "3"
```

还可以直接通过可寻址的 `reflect.Value` 来更新变量，不用通过指针，而是直接调用 `reflect.Value.Set` 方法：

```
d.SetFloat(4)
fmt.Println(x) // "4"
```

平常由编译器来检查的那些可赋值性条件，在这种情况下则是在运行时由 `Set` 方法来检查。上面的变量和值都是 `int` 类型，但如果变量类型是 `int64`，这个程序就会崩溃。所以确保这个值对于变量类型是可赋值的是很重要的一件事。

```
d.SetFloat(int64(5)) // 崩溃: int64 不可赋值给 int
```

当然，在一个不可寻址的 `reflect.Value` 上调用 `Set` 方法也会崩溃：

```
x := 2
b := reflect.ValueOf(x)
b.SetInt(3) // 崩溃: 在不可寻址的值上使用 Set
```

我们还有为一些基本类型特化的 `Set` 变种：`SetInt`、`SetUint`、`SetString`、`SetFloat` 等：

```
d := reflect.ValueOf(&x).Elem()
d.SetInt(3)
fmt.Println(x) // "3"
```

这些方法还有一定程度的容错性。只要变量类型是某种带符号的整数，比如 `SetInt`，甚至是底层类型为带符号整数的命名类型，都可以成功。如果值太大了还会无提示地截断它。但需要注意的是，在指向 `interface{}` 变量的 `reflect.Value` 上调用 `SetInt` 会奔溃（尽管使用 `Set` 就没有问题）。

```
x := 1
rx := reflect.ValueOf(&x).Elem()
rx.SetInt(2) // OK, x = 2
rx.SetString("hello") // 崩溃: 字符串不能赋给整数
rx.SetInt(reflect.ValueOf("hello")) // 崩溃: 字符串不能赋给整数

var y interface{}
ry := reflect.ValueOf(&y).Elem()
ry.SetInt(2) // 崩溃: 在指向接口的 Value 上调用 SetInt
ry.SetString("hello") // 崩溃: 在指向接口的 Value 上调用 SetString
ry.SetString(reflect.ValueOf("hello")) // OK, y = "hello"
```

在把 `Display` 作用于 `os.Stdout` 时，我们发现反射可以读取到未导出结构字段的值，通过 Go 语言的常规方法这些值是无法读取的。比如 `os.File` 结构在类 UNIX 平台上的 `fd` `int` 字段。但反射不能更新这些值：

```
stdout := reflect.ValueOf(os.Stdout).Elem() // *os.Stdout, 一个 os.File 变量
fmt.Println(stdout.Type()) // "os.File"
fd := stdout.FieldByName("fd")
fmt.Println(fd.Int()) // "1"
fd.SetInt(2) // 崩溃: 未导出字段
```

一个可寻址的 `reflect.Value` 会记录它是否是通过遍历一个未导出字段来获得的，如果是这样，则不允许修改。所以，在更新变量前用 `CanAddr` 来检查并不能保证正确。`CanSet` 方法才能正确地报告一个 `reflect.Value` 是否可寻址且可更改：

```
fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"
```

12.6 示例：解码 S 表达式

对于标准库 `encoding/...` 提供的每一个 `Marshal` 函数，都有一个对应的 `Unmarshal` 函数来做解码。正如在 4.5 节中所见到的，对于一个包含编码的 JSON 数据的字节 slice，我们可以按下面的方法解码为 `Movie` 类型（参见 12.3 节）：

```
data := []byte{/* ... */}
var movie Movie
err := json.Unmarshal(data, &movie)
```

`Unmarshal` 函数使用反射来修改已存在的 `movie` 变量的字段，根据 `Movie` 类型和输入数据来创建新的 map、结构和 slice。

现在为 S 表达式实现一个简单的 `Unmarshal` 函数，这个函数与上面使用过的标准 `json.Unmarshal` 函数类似，与之前的 `sexpr.Marshal` 则正好相反。我必须先提醒你，一个鲁棒且通用的实现需要的代码量远超这个示例能容纳的量（尽管这个示例已经很长了），所以我们必须走一些捷径。我们仅支持了 S 表达式一个有限的子集，并且没有优雅地处理错误。代码的目的是阐释反射，而不是语法分析。

词法分析程序使用 `text/scanner` 包提供的扫描器 `Scanner` 类型来把输入流分解成一系列的标记（`token`），包括注释、标识符、字符串字面量和数字字面量。扫描器的 `Scan` 方法向前推进扫描位置并且返回下一个标记（类型为 `rune`）。大部分标记（比如 '()'）都只包含单个 `rune`，但 `text/scanner` 包则用 `rune` 类型的小负数区域来表示那些多字符的标记，比如 `Ident`、`String`、`Int`。调用 `Scan` 会返回标记的类型，调用 `TokenText` 则会返回标记对应的文本。

因为一个典型的分析器需要多次分析当前的标记，但 `Scan` 方法会一直推进扫描位置，所以我们把扫描器封装到一个 `lexer` 辅助类型中，其中保存了 `Scan` 最近返回的标记。

gopl.io/ch12/seexpr

```
type lexer struct {
    scan scanner.Scanner
    token rune // 当前标记
}

func (lex *lexer) next() { lex.token = lex.scan.Scan() }
func (lex *lexer) text() string { return lex.scan.TokenText() }

func (lex *lexer) consume(want rune) {
    if lex.token != want { // 注意：这不是一个好的错误处理示例
        panic(fmt.Sprintf("got %q, want %q", lex.text(), want))
    }
    lex.next()
}
```

让我们先看一下分析器。它有两个主要的函数，第一个是 `read`，它读取从当前标记开始的 S 表达式，并更新由可寻址的 `reflect.Value` `v` 指向的变量。

```
func read(lex *lexer, v reflect.Value) {
    switch lex.token {
        // 仅有的有效标识符是 "nil" 和结构体的字段名
        // "nil" and struct field names.
        if lex.text() == "nil" {
            v.Set(reflect.Zero(v.Type()))
            lex.next()
            return
        }
    }
}
```

```

case scanner.String:
    s, _ := strconv.Unquote(lex.text()) // 注意：错误被忽略
    v.SetString(s)
    lex.next()
    return
case scanner.Int:
    i, _ := strconv.Atoi(lex.text()) // 注意：错误被忽略
    v.SetInt(int64(i))
    lex.next()
    return
case '(':
    lex.next()
    readList(lex, v)
    lex.next() // consume ')'
    return
}
panic(fmt.Sprintf("unexpected token %q", lex.text()))
}

```

S 表达式为两个不同的目的使用标识符：结构体的字段名和指针的 nil 值。read 函数只处理后一种情形。当它遇到 scanner.Ident "nil" 时，通过 reflect.Zero 函数把 v 设置为其类型的零值。对于其他标识符，则产生一个错误^Θ。readList 函数（接下来马上要看到）则把标识符处理为结构字段名。

一个 '(' 标记代表一个列表的开始。第二个函数 readList 可把列表解码为多种类型：map、结构体、slice 或者数组，主要根据当前正在处理的 Go 变量类型。对于每种情形，都会循环解析内容直到遇到匹配的右括号 ')'，这个是由 endList 函数来检测的。

比较有趣的地方是递归。最简单的例子是一个数组。在遇到 ')' 之前，我们使用 Index 方法来获得数组的一个元素，再递归调用 read 来填充数据。与其他错误处理类似，如果输入数据导致解码器的下标超过了数组的大小，解码器崩溃。slice 的流程与数组比较类似，不同之处是先创建每一个元素变量，再填充，最后追加到 slice 中。

结构体和 map 在循环的每一轮中都必须解析一个关于 (key value) 的子列表。对于结构体，key 是用来定位字段的符号。与数组的情形类似，我们通过 FieldByName 函数来获得结构体字段的现有变量，再递归调用 read 来填充。对于 map，key 可以是任何类型。与 slice 类似，先创建新变量，递归地填充，最后再把新的键值对插入映射表中。

```

func readList(lex *lexer, v reflect.Value) {
    switch v.Kind() {
    case reflect.Array: // (item ...)
        for i := 0; !endList(lex); i++ {
            read(lex, v.Index(i))
        }
    case reflect.Slice: // (item ...)
        for !endList(lex) {
            item := reflect.New(v.Type().Elem()).Elem()
            read(lex, item)
            v.Set(reflect.Append(v, item))
        }
    }
}

```

^Θ 根据代码，应该是直接忽略了。——译者注

```

case reflect.Struct: // ((name value) ...)
    for !endList(lex) {
        lex.consume('(')
        if lex.token != scanner.Ident {
            panic(fmt.Sprintf("got token %q, want field name", lex.text()))
        }
        name := lex.text()
        lex.next()
        read(lex, v.FieldByName(name))
        lex.consume(')')
    }

case reflect.Map: // ((key value) ...)
    v.Set(reflect.MakeMap(v.Type()))
    for !endList(lex) {
        lex.consume('(')
        key := reflect.New(v.Type().Key()).Elem()
        read(lex, key)
        value := reflect.New(v.Type().Elem()).Elem()
        read(lex, value)
        v.SetMapIndex(key, value)
        lex.consume(')')
    }

default:
    panic(fmt.Sprintf("cannot decode list into %v", v.Type()))
}

func endList(lex *lexer) bool {
    switch lex.token {
    case scanner.EOF:
        panic("end of file")
    case ')':
        return true
    }
    return false
}

```

最后，把解析器封装成如下所示的一个导出函数 `Unmarshal`，隐藏了实现中很多不完美之处。比如在解析过程中遇到错误会崩溃，因此 `Unmarshal` 使用一个延迟调用来从崩溃中恢复（见 5.10 节），并且返回一条错误消息。

```

// Unmarshal 解析 S 表达式数据并且填充到非 nil 指针 out 指向的变量
func Unmarshal(data []byte, out interface{}) (err error) {
    lex := &lexer{scan: scanner.Scanner{Mode: scanner.GoTokens}}
    lex.scan.Init(bytes.NewReader(data))
    lex.next() // 获取第一个标记
    defer func() {
        // 注意：这不是一个好的错误处理示例
        if x := recover(); x != nil {
            err = fmt.Errorf("error at %s: %v", lex.scan.Position, x)
        }
    }()
    read(lex, reflect.ValueOf(out).Elem())
    return nil
}

```

一个具备用于生产环境的质量的实现对任何的输入都不应当崩溃，而且应当对每次错误详细报告信息，可能的话，应当包含行号或者偏移量。无论如何，我们希望这个示例有助于

了解 `encoding/json` 这类包的底层机制，以及如何使用反射来填充数据结构。

练习 12.8：类似于 `json.Unmarshal` 函数，`sexpr.Unmarshal` 函数在解码之前就需要完整的字节 slice。仿照 `json.Decoder`，定义一个 `sexpr.Decoder` 类型，允许从一个 `io.Reader` 接口解码一系列的值。使用这个新类型来重新实现 `sexpr.Unmarshal`。

练习 12.9：仿照 `xml.Decoder`（参考 7.14 节），写一个基于标记的 S 表达式解码 API。你需要 5 个类型的标记：`Symbol`、`String`、`Int`、`StartList` 和 `Endlist`。

练习 12.10：扩展 `sexpr.Unmarshal`，以处理练习 12.3 中按你的答案编码的布尔值、浮点数和接口。（提示：为了解码接口，你需要一个 map，其中包含每个支持类型从名字到 `reflect.Type` 的映射。）

12.7 访问结构体字段标签

在 4.5 节我们用结构体字段标签来修改 Go 结构值的 JSON 编码方式。`json` 字段标签让我们可以选择其他的字段名以及忽略输出的空字段。本节将讨论如何用反射来获取字段标签。

在一个 Web 服务器中，绝大部分 HTTP 处理函数的第一件事就是提取请求参数到局部变量中。我们将定义一个工具函数 `params.Unpack`，使用结构体字段标签来简化 HTTP 处理程序（参考 7.7 节）的编写。

首先，展示如何使用这个方法。下面的 `search` 函数就是一个 HTTP 处理函数，它定义一个变量 `data`，`data` 的类型是一个字段与 HTTP 请求参数对应的匿名结构。结构体的字段标签指定参数名称，这些名称一般比较短，含义也比较模糊，毕竟 URL 长度有限，不能随便浪费。`Unpack` 函数从请求中提取数据来填充这个结构体，这样不仅可以更方便地访问，还避免了手动转换类型。

```
gopl.io/ch12/search
import "gopl.io/ch12/params"

// search 用于处理 /search URL endpoint.
func search(resp http.ResponseWriter, req *http.Request) {
    var data struct {
        Labels      []string `http:"l"`
        MaxResults int      `http:"max"`
        Exact       bool     `http:"x"`
    }
    data.MaxResults = 10 // 设置默认值
    if err := params.Unpack(req, &data); err != nil {
        http.Error(resp, err.Error(), http.StatusBadRequest) // 400
        return
    }

    // ...其他处理代码...
    fmt.Fprintf(resp, "Search: %+v\n", data)
}
```

下面的 `Unpack` 函数做了三件事情。首先，调用 `req.ParseForm()` 来解析请求。在这之后，`req.Form` 就有了所有的请求参数，这个方法对 HTTP GET 和 POST 请求都适用。

接着，`Unpack` 函数构造了一个从每个有效字段名到对应字段变量的映射。在字段有标签时有效字段名与实际字段名可能会有差别。`reflect.Type` 的 `Field` 方法会返回一个 `reflect.StructField` 类型，这个类型提供了每个字段的名称、类型以及一个可选的标签。它的 `Tag` 字

段类型为 `reflect.StructTag`, 底层类型为字符串, 提供了一个 `Get` 方法用于解析和提取对于一个特定键的子串, 比如这个例子中用到的 `http:"..."`。

gopl.io/ch12/params

```
// Unpack 从 HTTP 请求 req 的参数中提取数据填充到 ptr 指向结构体的各个字段
// from the HTTP request parameters in req.
func Unpack(req *http.Request, ptr interface{}) error {
    if err := req.ParseForm(); err != nil {
        return err
    }

    // 创建字段映射表, 键为有效名称
    fields := make(map[string]reflect.Value)
    v := reflect.ValueOf(ptr).Elem() // 结构变量
    for i := 0; i < v.NumField(); i++ {
        fieldInfo := v.Type().Field(i) // a reflect.StructField
        tag := fieldInfo.Tag          // a reflect.StructTag
        name := tag.Get("http")
        if name == "" {
            name = strings.ToLower(fieldInfo.Name)
        }
        fields[name] = v.Field(i)
    }

    // 对请求中的每个参数更新结构体中对应的字段
    for name, values := range req.Form {
        f := fields[name]
        if !f.IsValid() {
            continue // 忽略不能识别的 HTTP 参数
        }
        for _, value := range values {
            if f.Kind() == reflect.Slice {
                elem := reflect.New(f.Type().Elem()).Elem()
                if err := populate(elem, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
                f.Set(reflect.Append(f, elem))
            } else {
                if err := populate(f, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
            }
        }
    }
    return nil
}
```

最后, `Unpack` 遍历 HTTP 参数中的所有键值对, 并且更新对应的结构体字段。注意, 同一个参数可能会出现多次。如果有这种情况并且字段是 slice 类型, 则这个参数的所有值都会追加到 slice 里。如果不是, 则这个字段会被多次覆盖, 仅有最后一个值才是有效的。

`populate` 函数负责从单个 HTTP 请求参数值填充单个字段 `v` (或者 slice 字段中的单个元素)。现在, 它仅支持字符串、有符号整数和布尔值。支持其他类型则留作练习。

```
func populate(v reflect.Value, value string) error {
    switch v.Kind() {
    case reflect.String:
        v.SetString(value)
```

```

case reflect.Int:
    i, err := strconv.ParseInt(value, 10, 64)
    if err != nil {
        return err
    }
    v.SetInt(i)

case reflect.Bool:
    b, err := strconv.ParseBool(value)
    if err != nil {
        return err
    }
    v.SetBool(b)

default:
    return fmt.Errorf("unsupported kind %s", v.Type())
}
return nil
}

```

接着把 `server` 处理程序添加到一个 Web 服务器中。下面就是一个典型的交互过程：

```

$ go build gopl.io/ch12/search
$ ./search &
$ ./fetch 'http://localhost:12345/search'
Search: {Labels:[] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming&max=100'
Search: {Labels:[golang programming] MaxResults:100 Exact:false}
$ ./fetch 'http://localhost:12345/search?x=true&l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:true}
$ ./fetch 'http://localhost:12345/search?q=hello&x=123'
x: strconv.ParseBool: parsing "123": invalid syntax
$ ./fetch 'http://localhost:12345/search?q=hello&max=lots'
max: strconv.ParseInt: parsing "lots": invalid syntax

```

练习 12.11：写一个与 `Unpack` 对应的 `Pack` 函数。给定一个结构体的值，`Pack` 应当返回一个 URL，这个 URL 的参数与输入的结构体对应。

练习 12.12：扩展字段标签语法来支持参数有效性检验。比如，一个字符串应当是一个有效的 `email` 地址或者有效的信用卡号码，一个整数应当是一个有效的美国邮编^Θ。修改 `Unpack` 函数来支持这些功能。

练习 12.13：修改 S 表达式编码器（参考 12.4 节）和解码器（参考 12.6 节），支持 `sexpr:"..."` 形式的字段标签，标签含义同 `encoding/json` 包（参考 4.5 节）。

12.8 显示类型的方法

最后一个反射示例使用 `reflect.Type` 来显示一个任意值的类型并枚举它的方法：

```

gopl.io/ch12/methods
// Print 输出值 x 的所有方法
func Print(x interface{}) {
    v := reflect.ValueOf(x)
    t := v.Type()
    fmt.Printf("type %s\n", t)
}

```

^Θ 美国邮编为 5 位整数。——译者注

```

    for i := 0; i < v.NumMethod(); i++ {
        methType := v.Method(i).Type()
        fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,
            strings.TrimPrefix(methType.String(), "func"))
    }
}

```

`reflect.Type` 和 `reflect.Value` 都有一个叫作 `Method` 的方法。每个 `t.Method(i)` (从 `reflect.Type` 调用) 都会返回一个 `reflect.Method` 类型的实例，这个结构类型描述了这个方法的名称和类型。而每个 `v.Method(i)` (从 `reflect.Value` 调用) 都会返回一个 `reflect.Value`，代表一个方法值 (6.4 节)，即一个已绑定接收者的方法。使用 `reflect.Value.Call` 方法可以调用 `Func` 类型的 `Value` (为节省版面，这里就不演示了)，但这个程序只需要它的类型。

下面就是两个类型 `time.Duration` 和 `*strings.Replacer` 的方法列表：

```

methods.Print(time.Hour)
// 输出:
// type time.Duration
// func (time.Duration) Hours() float64
// func (time.Duration) Minutes() float64
// func (time.Duration) Nanoseconds() int64
// func (time.Duration) Seconds() float64
// func (time.Duration) String() string

methods.Print(new(strings.Replacer))
// 输出:
// type *strings.Replacer
// func (*strings.Replacer) Replace(string) string
// func (*strings.Replacer) WriteString(io.Writer, string) (int, error)

```

12.9 注意事项

还有很多反射 API，但限于篇幅原因，这里不再展示，但之前的示例揭示了反射能做哪些事情。反射是一个功能和表达能力都很强大的工具，但应该谨慎使用它，具体有三个原因。

第一个原因是基于反射的代码是很脆弱的。能导致编译器报告类型错误的每种写法，在反射中都有一个对应的误用方法。编译器在编译时就能向你报告这个错误，而反射错误则要等到执行时才以崩溃的方式来报告，而这可能是代码写好很久以后，甚至是代码开始执行很久以后才会发生的事。

比如，如果 `readList` 函数 (参考 12.6 节) 尝试从输入读取一个字符串然后填充一个 `int` 类型的变量，那么调用 `reflect.Value.SetString` 就会崩溃。很多使用反射的程序都有类似的风险，所以对每一个 `reflect.Value` 都需要仔细注意它的类型、是否可寻址、是否可设置。

回避这种缺陷的最好办法是确保反射的使用完整地封装在包里边，并且如果可能，在包的 API 中避免使用 `reflect.Value`，尽量使用特定的类型来确保输入是合法的值。如果做不到这点，那就需要在每个危险操作前都做额外的动态检查。作为标准库中的一个示例，当 `fmt.Printf` 遇到操作数类型不合适时，它不会莫名其妙地崩溃，而是输出一条描述性的错误消息。尽管程序仍然有 bug，但定位起来就简单多了。

```
fmt.Printf("%d %s\n", "hello", 42) // "%!d(string=hello) %!s(int=42)"
```

反射还降低了自动重构和分析工具的安全性与准确度，因为它们无法检测到类型信息。

避免使用反射的第二个原因是类型其实也算是某种形式的文档，而反射的相关操作则无法做静态类型检查，所以大量使用反射的代码是很难理解的。对于接受 `interface{}` 或者 `reflect.Value` 的函数，一定要写清楚期望的参数类型和其他限制条件（即不变量）。

第三个原因是基于反射的函数会比为特定类型优化的函数慢一两个数量级。在一个典型的程序中，大部分函数与整体性能无关，所以为了让程序更清晰可以使用反射。测试就很合适使用反射，因为大部分测试都使用小数据集。但对于关键路径上的函数，则最好避免使用反射。

低级编程

Go 语言的设计确保了一些安全的属性从而限制了 Go 程序可能“出错”的途径。在编译期间，类型检查检测那些尝试把结果赋给不正确类型的操作，例如，从一个字符串减去另一个字符串。严格的数据转换规则阻止了直接对内置类型字符串、map、slice 和通道的内部访问。

对于那些无法静态检测的错误，例如数组访问越界或者 nil 指针引用，动态检测确保程序在一个禁止的操作发生的时候立即终止并给出错误提示信息。自动内存管理（垃圾回收）防止了“释放后使用”的 bug，以及大多数的内存泄漏。

很多实现的细节是无法通过 Go 程序来访问的。对于聚合类型（如结构体）的内存布局或者一个函数对应的机器码，就无法了解识别出当前运行的 goroutine 所在的线程也不可行。事实上，Go 协程调度器可以自由地将 goroutine 从一个线程移动到另一个线程。指针会识别所引用的变量，而不用暴露出变量的地址。在垃圾回收的过程中，变量地址会被移动，同时指针也会透明地更新。

总之，这些特性使得 Go 程序（尤其是那些出错的程序）比起 C 来行为更加可预测并且减少了神秘性。通过隐藏底层细节，它们可以使 Go 程序高度可移植，因为语言的语义很大程度上独立于任何特定的编译器、操作系统或者 CPU 架构。（并非完全独立，有一些细节漏掉了，例如处理器的字宽度，某些表达式的计算顺序，以及强加给编译器的限制性实现。）

偶尔，我们会选择放弃一些有益的保障来实现最可能的高性能，和以其他语言编写的库进行交互或者实现一个无法使用纯 Go 描述的函数。

本章将揭示包 `unsafe` 如何让我们打破常规，以及如何使用 `cgo` 工具来为 C 函数库和系统调用创建 Go 语言的绑定。

本章所讲述的内容不可以滥用。如果对细节部分不深思熟虑，将会带来各种不可预测的、奇怪的、非局部性错误，C 语言的程序员经常碰到这些问题。使用 `unsafe` 包中的内容也无法保证和 Go 未来的发布版兼容，因为无论是无意还是有意，这个包里面的内容都会依赖一些未知的实现细节，而它们可能发生未知的变化。

包 `unsafe` 是很神奇的，虽然它像普通的包并且像普通包那样导入，但是它事实上是由编译器实现的。它提供了对语言内置特性的访问功能，而这些特性一般是不可见的，因为它们暴露了 Go 详细的内存布局。把这些特性单独放在一个包中，就使得它们的本来就不频繁的使用场合变得更加引人注目。另外，一些环境下，出于安全原因，`unsafe` 包的使用是受限制的。

包 `unsafe` 广泛使用在和操作系统交互的低级包（比如 `runtime`、`os`、`syscall` 和 `net`）中，但是普通程序从来不需要使用它。

13.1 `unsafe.Sizeof`、`Alignof` 和 `Offsetof`

函数 `unsafe.Sizeof` 报告传递给它的参数在内存中占用的字节长度，这个参数可以是任何类型的表达式，不会计算表达式。`Sizeof` 调用返回一个 `uintptr` 类型的常量表达式，所以

这个结果可以作为数组类型的维度或者用于计算其他的常量。

```
import "unsafe"  
fmt.Println(unsafe.Sizeof(float64(0))) // "8"
```

`Sizeof` 仅报告每个数据结构固定部分的内存占用的字节长度，例如指针或者字符串的长度，但是不会报告诸如字符串内容这种间接内容。非聚合类型的典型长度如下所示，当然准确的长度随工具链的不同而不同。为了可移植性，我们将以字来表示引用类型（或者包含引用的类型）的长度。在 32 位系统上，字的长度是 4 个字节；而在 64 位系统上，字的长度是 8 个字节。

在类型的值在内存中对齐的情况下，计算机的加载或者写入会很高效。例如，一个两字节值（如 `int16`）的地址应该是一个偶数，一个四字节值（如 `rune`）的地址应该是 4 的整数倍，一个八字节值（如 `float64`、`uint64` 或者 64 位指针）的地址应该是 8 的整数倍。更大整数倍的对齐很少见，即使是像 `complex128` 这种大的数据类型。

因此，聚合类型（结构体或数组）的值的长度至少是它的成员或元素长度之和，并且由于“内存间隙”的存在，或许比这个更大一些。内存空位是由编译器添加的未使用的内存地址，用来确保连续的成员或者元素相对于结构体或数组的起始地址是对齐的。

类型	大小
<code>bool</code>	1个字
<code>intN</code> 、 <code>uintN</code> 、 <code>floatN</code> 、 <code>complexN</code>	$N / 8$ 字节（例如 <code>float64</code> 是 8 字节）
<code>int</code> 、 <code>uint</code> 、 <code>uintptr</code>	1个字
<code>*T</code>	1个字
<code>string</code>	2个字（数据、长度）
<code>[]T</code>	3个字（数据、长度、容量）
<code>map</code>	1个字
<code>func</code>	1个字
<code>chan</code>	1个字
<code>interface</code>	两个字（类型、值）

语言规范并没有要求成员声明的顺序对应内存中的布局顺序，所以在理论上，编译器可以自由安排。尽管这样说，但是实际上没人这样做。如果结构体成员的类型是不同的，那么将相同类型的成员定义在一起可以更节约内存空间。例如下面的三个结构体拥有相同的成员，但是第一个定义比其他两个定义要多占至多 50% 的内存。

```
// 64位 32位  
struct{ bool; float64; int16 } // 3 个字 4 个字  
struct{ float64; int16; bool } // 两个字 3 个字  
struct{ bool; int16; float64 } // 两个字 3 个字
```

对齐算法的细节已经超出本书的范围了，同时不值得担心每个结构体的内存布局，但是在为高效组合的数据结构经常分配内存的时候可以更加紧凑、快速。

函数 `unsafe.Alignof` 报告它参数类型所要求的对齐方式。和 `Sizeof` 一样，它的参数可以是任意类型的表达式，并且返回一个常量。典型地，布尔类型和数值类型对齐到它们的长度（最大 8 字节），而其他的类型则按字对齐。

函数 `unsafe.Offsetof`，计算成员 `f` 相对于结构体 `x` 起始地址的偏移值，如果有内存空位，也计算在内，该函数的操作数必须是一个成员选择器 `x.f`。

图 13-1 演示了一个结构体变量 `x` 和它在典型的 32 位和 64 位系统上的内存布局。灰色

的部分都是内存空位。

```
var x struct {
    a bool
    b int16
    c []int
}
```

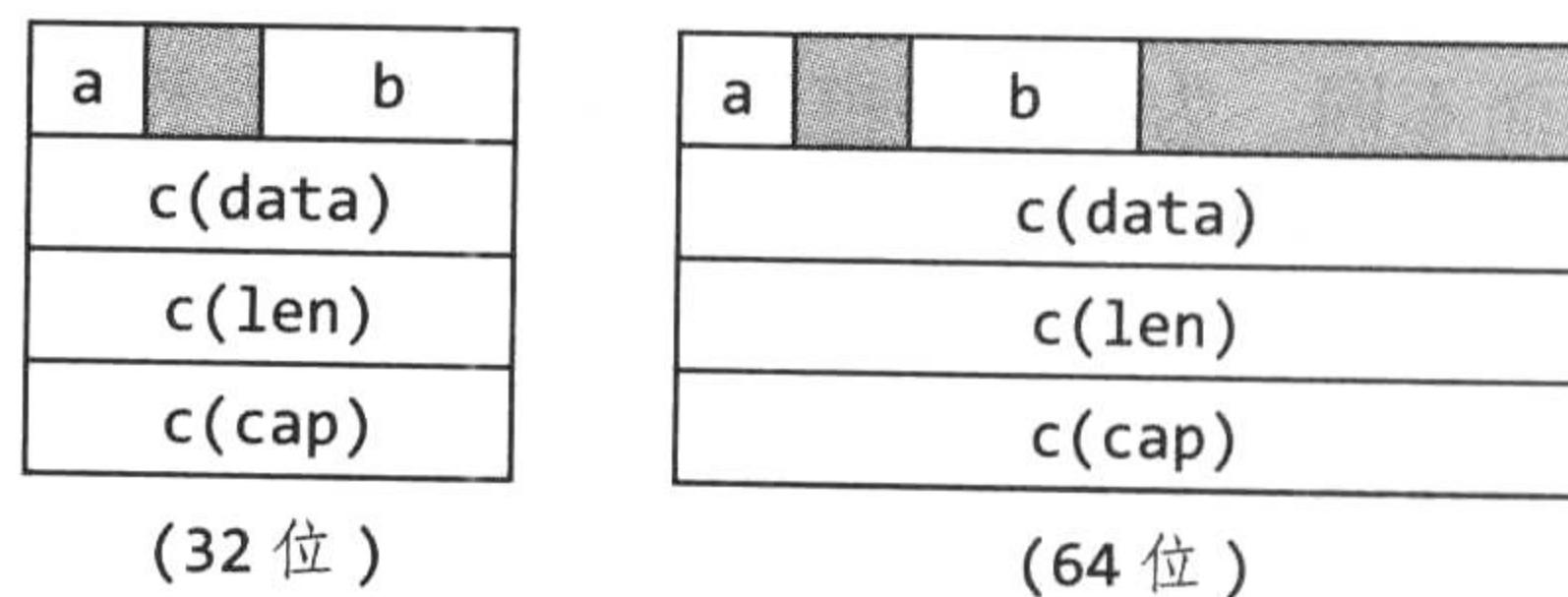


图 13-1 结构体中的内存空位

下面的代码演示了对结构体 `x` 以及 `x` 的三个成员调用 `unsafe` 的三个函数的结果：在典型的 32 位平台上：

```
Sizeof(x) = 16 Alignof(x) = 4
Sizeof(x.a) = 1 Alignof(x.a) = 1 Offsetof(x.a) = 0
Sizeof(x.b) = 2 Alignof(x.b) = 2 Offsetof(x.b) = 2
Sizeof(x.c) = 12 Alignof(x.c) = 4 Offsetof(x.c) = 4
```

在典型的 64 位平台上：

```
Sizeof(x) = 32 Alignof(x) = 8
Sizeof(x.a) = 1 Alignof(x.a) = 1 Offsetof(x.a) = 0
Sizeof(x.b) = 2 Alignof(x.b) = 2 Offsetof(x.b) = 2
Sizeof(x.c) = 24 Alignof(x.c) = 8 Offsetof(x.c) = 8
```

虽然它们的名字叫作 `unsafe`，但是这些函数本身是安全的，并且在做内存优化的时候，它们对理解程序底层内存布局很有帮助。

13.2 unsafe.Pointer

很多指针类型都写作 `*T`，意思是“一个指向 `T` 类型变量的指针”。`unsafe.Pointer` 类型是一种特殊类型的指针，它可以存储任何变量的地址。当然，我们无法间接地通过一个 `unsafe.Pointer` 变量来使用 `*p`，因为我们不知道这个表达式的具体类型。和普通的指针一样，`unsafe.Pointer` 类型的指针是可比较的并且可以和 `nil` 做比较，`nil` 是指针类型的零值。

一个普通的指针 `*T` 可以转换为 `unsafe.Pointer` 类型的指针，另外一个 `unsafe.Pointer` 类型的指针也可以转换回普通指针，而且可以不必和原来的类型 `*T` 相同。例如，通过转换一个 `*float64` 类型的指针到 `*uint64` 类型，可以查看一下浮点类型变量的位模式：

```
package math

func Float64bits(f float64) uint64 { return *(*uint64)(unsafe.Pointer(&f)) }
fmt.Printf("%#016x\n", Float64bits(1.0)) // "0x3ff0000000000000"
```

也可以通过结果指针来更新位模式。这个对一个浮点类型的变量来说是无害的，但是通常使用 `unsafe.Pointer` 进行类型转换可以让我们将任意值写入内存中，并因此破坏了类型系统。

`unsafe.Pointer` 类型也可以转换为 `uintptr` 类型，`uintptr` 类型保存了指针所指向地址的数值，这就可以让我们对地址进行数值计算。（回忆一下第 3 章，`uintptr` 类型是一个足够大

的无符号整型，可以用来表示任何地址。) 这种转换当然也可以反过来，但是这种从 `uintptr` 到 `unsafe.Pointer` 的转换也会破坏类型系统，因为并不是所有的数值都是合法的内存地址。

很多 `unsafe.Pointer` 类型的值都是从普通指针到原始内存地址以及再从内存地址到普通指针进行转换的中间值。下面的例子获取变量 `x` 的地址，然后加上其成员 `b` 的地址偏移量，并将结果转换为 `*int16` 指针类型，接着通过这个指针更新 `x.b` 的值。

```
gopl.io/ch13/unsafeptr
var x struct {
    a bool
    b int16
    c []int
}

// 等价于 to pb := &x.b
pb := (*int16)(unsafe.Pointer(
    uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
*pb = 42

fmt.Println(x.b) // "42"
```

虽然这种语法看上去很冗长，这或许也不是坏事，因为这些特性不应该被随意使用，但是不要尝试引入 `uintptr` 类型的临时变量来破坏整行代码。下面这段代码是不正确的：

```
// 注意：很微妙的错误
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
pb := (*int16)(unsafe.Pointer(tmp))
*pb = 42
```

原因很微妙。一些垃圾回收器在内存中把变量移来移去以减少内存碎片或者为了进行簿记工作。这种类型的垃圾回收器称为移动的垃圾回收器。当一个变量在内存中移动后，该变量所指向旧地址的所有指针都需要更新以指向新地址。从垃圾回收器的角度看，`unsafe.Pointer` 是一个变量的指针，当变量移动的时候，它的值也需要改变，而 `uintptr` 仅仅是一个数值，所以它的值是不会变的。上面的错误代码使得垃圾回收器无法通过到非指针变量 `tmp` 了解它背后的指针。当第二条语句执行的时候，变量 `x` 可能在内存中已经移动了，这个时候 `tmp` 中的值就不是变量 `&x.b` 的地址了。第三条语句将向一个任意的内存地址写入值 42。

上面的问题会导致很多其他的错误写法。例如在这条语句执行之后，将没有指针指向 `new` 创建的那个变量。

```
pT := uintptr(unsafe.Pointer(new(T))) // 注意：错误
```

在这种情况下，垃圾回收器将会在语句执行结束后回收内存，在这之后，`pT` 存储的是变量旧的地址，不过这个时候这个地址对应的已经不是那个变量了。

当前版本的 Go 实现没有使用移动垃圾回收器（尽管未来可能会），但是不要暗自庆幸，因为当前版本的 Go 确实会在内存中移动变量。回忆一下 5.2 节，goroutine 栈会根据需要增长。这个时候，旧栈上面的所有变量会重新分配到新的、更大的栈上面，所以我们不能指望变量的地址值在它的整个生命周期都不会变。

在本书撰写的时候，在将 `unsafe.Pointer` 转换为 `uintptr` 之后，并没有什么清晰的指导意见可以让 Go 程序员参考（可以看 Go issue 7192），所以我们强烈建议你遵守最小可用原则。可认为所有的 `uintptr` 值都包含一个变量的旧地址，并且减少 `unsafe.Pointer` 到 `uintptr` 之间的转换到使用这个 `uintptr` 之间的操作次数。在上面的第一个例子中，转换为 `uintptr`，

加上成员地址偏移量，然后再转换回来，都是在一条语句中实现的。

当调用一个返回 `uintptr` 类型的库函数时，例如下面 `reflect` 包中函数所返回的值，这些结果应该立刻转换为 `unsafe.Pointer` 来确保它们在接下来的代码中继续指向同一个变量。

```
package reflect

func (Value) Pointer() uintptr
func (Value) UnsafeAddr() uintptr
func (Value) InterfaceData() [2]uintptr // (索引 1)
```

13.3 示例：深度相等

包 `reflect` 中的 `DeepEqual` 函数用来报告两个变量的值是否“深度”相等。`DeepEqual` 函数对基本类型使用内置的 `==` 操作符进行比较；对于组合类型，它逐层深入比较相应的元素。因为这个函数适合于任意一对变量值，甚至是那些无法通过 `==` 来进行比较的值，所以这个函数在测试中广泛使用。下面的测试使用 `DeepEqual` 来比较两个 `[]string` 类型的值：

```
func TestSplit(t *testing.T) {
    got := strings.Split("a:b:c", ":")
    want := []string{"a", "b", "c"};
    if !reflect.DeepEqual(got, want) { /* ... */ }
}
```

虽然 `DeepEqual` 很方便，但是它的特点就是判断过于武断。例如，它不会认为一个值为 `nil` 的 `map` 和一个值不为 `nil` 的空 `map` 相等，也不会判断出一个值为 `nil` 的 `slice` 和一个值不为 `nil` 的空 `slice` 相等。

```
var a, b []string = nil, []string{}
fmt.Println(reflect.DeepEqual(a, b)) // "false"

var c, d map[string]int = nil, make(map[string]int)
fmt.Println(reflect.DeepEqual(c, d)) // "false"
```

在本节，我们将定义一个函数 `Equal` 来比较两个任意类型的值。和 `DeepEqual` 类似，它基于值来比较 `slice` 和 `map`，但是和 `DeepEqual` 不同的是，它认为一个值为 `nil` 的 `slice` 或 `map` 和一个值不为 `nil` 的空 `slice` 或 `map` 相等。对参数的基本递归检查可以通过反射来实现，方式和 12.3 节看过的 `Display` 程序类似。和平常一样，我们定义一个未导出函数 `equal` 用来进行递归检查。现在不用关心参数 `seen`。对于每对进行比较的值 `x` 和 `y`，`equal` 函数检查两者是否合法以及它们是否具有相同的类型。函数的结果通过 `switch` 的 `case` 语句返回，在 `case` 语句中比较两个相同类型的值。为了节约篇幅，我们目前已经很熟悉的类型就省略了。

gopl.io/ch13/equal

```
func equal(x, y reflect.Value, seen map[comparison]bool) bool {
    if !x.IsValid() || !y.IsValid() {
        return x.IsValid() == y.IsValid()
    }
    if x.Type() != y.Type() {
        return false
    }
    // ...省略了循环检查（参见后面）...

    switch x.Kind() {
    case reflect.Bool:
        return x.Bool() == y.Bool()
```

```

case reflect.String:
    return x.String() == y.String()

// ...为了简洁，数值类型就省略了...

case reflect.Chan, reflect.UnsafePointer, reflect.Func:
    return x.Pointer() == y.Pointer()

case reflect.Ptr, reflect.Interface:
    return equal(x.Elem(), y.Elem(), seen)

case reflect.Array, reflect.Slice:
    if x.Len() != y.Len() {
        return false
    }
    for i := 0; i < x.Len(); i++ {
        if !equal(x.Index(i), y.Index(i), seen) {
            return false
        }
    }
    return true

// ...为了简洁，结构体和map类型就省略了...
}
panic("unreachable")
}

```

和通常一样，在 API 中不暴露使用反射的细节，所以导出的 Equal 函数必须对参数显式调用 reflect.ValueOf 函数。

```

// Equal 函数检查 x 和 y 是否深度相等
func Equal(x, y interface{}) bool {
    seen := make(map[comparison]bool)
    return equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)
}

type comparison struct {
    x, y unsafe.Pointer
    t    reflect.Type
}

```

为了确保算法终止甚至可以对循环数据结构进行比较，它必须记录哪两对变量已经比较过了，并且避免再次进行比较。Equal 函数分配了一个叫作 comparison 的结构体集合，每个元素都包含两个变量的地址（使用 unsafe.Pointer 值表示）以及比较的类型。除了变量的地址外，还需要记录比较的类型，因为不同的变量可能拥有相同的地址。例如，如果 x 和 y 都是数组，x 和 x[0] 的地址是一样的，当然 y 和 y[0] 的地址也一样，这个时候区分开是比较了 x 和 y 还是比较了 x[0] 和 y[0] 就很重要。

当 equal 发现它的两个参数类型相同的时候，在执行 switch 语句进行比较之前，它检查这两个变量是否已经比较过了，如果已经比较过，则终止这次递归比较。

```

// 循环检查
if x.CanAddr() && y.CanAddr() {
    xptr := unsafe.Pointer(x.UnsafeAddr())
    yptr := unsafe.Pointer(y.UnsafeAddr())
    if xptr == yptr {
        return true // identical references
    }
    c := comparison{xptr, yptr, x.Type()}

```

```

if seen[c] {
    return true // already seen
}
seen[c] = true
}

```

这里是 Equal 函数的实际例子：

```

fmt.Println(Equal([]int{1, 2, 3}, []int{1, 2, 3}))           // "true"
fmt.Println(Equal([]string{"foo"}, []string{"bar"}))         // "false"
fmt.Println(Equal([]string(nil), []string{}))                // "true"
fmt.Println(Equal(map[string]int(nil), map[string]int{}))   // "true"

```

它甚至适用于循环输入，这和 12.3 节使得 Display 函数在循环中卡住的循环输入类似。

```

// 循环链表 a -> b -> a 和 c -> c
type link struct {
    value string
    tail  *link
}
a, b, c := &link{value: "a"}, &link{value: "b"}, &link{value: "c"}
a.tail, b.tail, c.tail = b, a, c
fmt.Println(Equal(a, a)) // "true"
fmt.Println(Equal(b, b)) // "true"
fmt.Println(Equal(c, c)) // "true"
fmt.Println(Equal(a, b)) // "false"
fmt.Println(Equal(a, c)) // "false"

```

练习 13.1： 定义一个深度比较函数，该函数把数值（任何类型）之间的差小于 10^{-9} 视为相等。

练习 13.2： 编写一个函数来判断它的参数是否是一个循环数据结构体。

13.4 使用 cgo 调用 C 代码

一个 Go 程序或许需要调用用 C 实现的硬件驱动程序，查询一个用 C++ 实现的嵌入式数据库，或者使用一些以 Fortran 实现的线性代数协程。C 作为一种编程混合语言已经很久了，所以无论那些广泛使用的包是哪种语言实现的，它们都导出了和 C 兼容的 API。

在本节，我们将使用 cgo 来构建一个简单的数据压缩程序，cgo 是用来为 C 函数创建 Go 绑定的工具。诸如此类的工具都叫作外部函数接口 (FFI)，并且 cgo 不是 Go 程序唯一的工具。SWIG (swig.org) 是另一个工具；它提供了更加复杂的特性用来集成 C++ 的类，但是这里不打算演示。

标准库的 compress/... 子包中提供了流行压缩算法的压缩器和解压缩器，包括 LZW (UNIX 工具 compress 使用的算法) 和 DEFLATE (GNU 工具 gzip 使用的算法)。这些包中的 API 有些许的不同，但是它们都提供一个对 io.Writer 的封装用来对写入的数据进行压缩，并且还有一个对 io.Reader 的封装，当从中读取数据的同时进行解压缩。例如：

```

package gzip // compress/gzip
func NewWriter(w io.Writer) io.WriteCloser
func NewReader(r io.Reader) (io.ReadCloser, error)

```

bzip2 算法基于优雅的 Burrows-Wheeler 变换，它比 gzip 运行起来慢但是可以得到更好的压缩效果。包 compress/bzip2 提供了 bzip2 的解压缩器，但是目前该包还没有提供压缩功能。从头开始开发工作量较大，但是恰好有一个文档完善且高性能的开源 C 语言实现：来

自 bzip.org 的 `libbzip2` 包。

如果 C 库很小，我们可以使用纯 Go 语言来移植它，并且如果性能对我们来说不是很关键，我们最好使用包 `os/exec` 以辅助子进程的方式来调用 C 程序。仅当你需要使用拥有有限 C API 并且复杂的、性能关键的库时，使用 `cgo` 来把它们包装成 Go 语言的绑定才有意义。本节剩下的部分将通过一个例子来说明。

从 C 的包 `libbzip2` 中，我们需要结构体类型 `bz_stream`，这个结构体包含输入和输出缓冲区，以及三个 C 函数：`BZ2_bzCompressInit`，它用来分配流的缓冲区；`BZ2_bzCompress`，它用来压缩输入缓冲区中的数据并写出到输出缓冲区；以及 `BZ2_bzCompressEnd`，它用来释放缓冲区。（不用担心 `libbzip2` 包的工作机制，本例的目的就是演示各部分如何一起工作。）

我们从 Go 语言中直接调用 C 函数 `BZ2_bzCompressInit` 和 `BZ2_bzCompressEnd`，但是对于 `BZ2_bzCompress` 函数，我们使用 C 语言定义个了包装函数，来演示它如何使用。下面的 C 源文件和 Go 代码都在包中：

```
gopl.io/ch13/bzip
/* 文件是 gopl.io/ch13/bzip/bzip2.c */
/* 对 libbzip2 的简单包装适合 cgo      */
#include <bzlib.h>

int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen) {
    s->next_in = in;
    s->avail_in = *inlen;
    s->next_out = out;
    s->avail_out = *outlen;
    int r = BZ2_bzCompress(s, action);
    *inlen -= s->avail_in;
    *outlen -= s->avail_out;
    return r;
}
```

现在我们来看 Go 代码，第一部分如下所示。声明 `import "c"` 很特别。没有包的名字是 `c`，但是这个导入会在 Go 编译器看到它之前促使 `go build` 利用 `cgo` 工具预处理文件。

```
// 包 bzip 封装了一个使用 bzip2 压缩算法的 writer(bzip.org)
package bzip

/*
#cgo CFLAGS: -I/usr/include
#cgo LDFLAGS: -L/usr/lib -lbz2
#include <bzlib.h>
int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen);
*/
import "C"
import (
    "io"
    "unsafe"
)

type writer struct {
    w      io.Writer // 基本输出流
    stream *C.bz_stream
    outbuf [64 * 1024]byte
}
```

```
// NewWriter 对于 bzip2 压缩的流返回一个 writer
func NewWriter(out io.Writer) io.WriteCloser {
    const (
        blockSize = 9
        verbosity = 0
        workFactor = 30
    )
    w := &writer{w: out, stream: C.bz2alloc()}
    C.BZ2_bzCompressInit(w.stream, blockSize, verbosity, workFactor)
    return w
}
```

在预处理过程中，`cgo` 产生一个临时包，这个包里面包含了所有 C 函数和类型对应的 Go 语言声明，例如 `C.bz_stream` 和 `C.BZ2_bzCompressInit`。`cgo` 工具通过以一种特殊的方式调用 C 编译器 `import "c"` 声明之前的注释来发现这些类型。

这些注释还可以包含 `#cgo` 指令用来指定 C 工具链中其他的选项。`CFLAGS` 和 `LDLDFLAGS` 的值将为编译器和链接器命令指定额外的参数，用来发现头文件 `bzlib.h` 和归档库 `libbz2.a`。这个例子假定它们都在系统的 `/usr` 目录下。根据个人的安装情况，你或许需要修改或者删除这些标记。

`NewWriter` 调用 C 函数 `BZ2_bzCompressInit` 来初始化流的缓冲区。这个 `writer` 类型包含一个额外的缓冲区用来耗尽解压缩器的输出缓冲区。

下面所示的 `Write` 方法将未压缩的数据写入压缩器中，然后在一个循环中调用 `bz2compress` 函数，直到所有的数据压缩完毕。注意，Go 程序可以访问 C 的类型（比如 `bz_stream`、`char` 和 `uint`），C 的函数（比如 `bz2compress`），甚至是类似 C 的预处理宏的对象（例如 `BZ_RUN`），都通过 `c.x` 的方式来访问。即使类型 `C.uint` 和 Go 的 `uint` 的长度相同，它们的类型也不同。

```
func (w *writer) Write(data []byte) (int, error) {
    if w.stream == nil {
        panic("closed")
    }
    var total int // 写入的未压缩字节
    for len(data) > 0 {
        inlen, outlen := C.uint(len(data)), C.uint(cap(w.outbuf))
        C.bz2compress(w.stream, C.BZ_RUN,
                      (*C.char)(unsafe.Pointer(&data[0])), &inlen,
                      (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        total += int(inlen)
        data = data[inlen:]
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return total, err
        }
    }
    return total, nil
}
```

每次循环都会将剩余 `data` 的地址和长度，以及 `w.outbuf` 的地址和容量传递给函数 `bz2compress`。两个表示长度的变量是通过它们的地址来传递的，而不是值，这样 C 函数就可以更新它们以此了解压缩了多少数据以及生成了多少压缩后的数据。然后把每块压缩后的数据写入底层 `io.Writer`。

`Close` 方法和 `Write` 方法结构相似，使用一个循环来将任何剩余的压缩后的数据从输出

流缓冲区写入底层。

```
// Close 方法清空压缩的数据并关闭流
// 它不会关闭底层的 io.Writer
func (w *writer) Close() error {
    if w.stream == nil {
        panic("closed")
    }
    defer func() {
        C.BZ2_bzCompressEnd(w.stream)
        C.bz2free(w.stream)
        w.stream = nil
    }()
    for {
        inlen, outlen := C.uint(0), C.uint(cap(w.outbuf))
        r := C.bz2compress(w.stream, C.BZ_FINISH, nil, &inlen,
                           (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return err
        }
        if r == C.BZ_STREAM_END {
            return nil
        }
    }
}
```

完成之后，`Close` 方法调用 `C.BZ2_bzCompressEnd` 来释放流缓冲区，使用 `defer` 来确保所有路径返回后一定会释放资源。在这个时候，`w.stream` 指针就不能安全地解引用了。为了安全，把它设置为 `nil`，并且为每个方法调用都添加显式的 `nil` 检查，这样如果用户在 `Close` 之后错误地调用一个方法该程序就会宕机。

不但 `writer` 不是并发安全的，而且并发调用 `close` 和 `Write` 也会导致 C 代码崩溃。在练习 13.3 中修复它。

下面的程序 `bzipper` 是一个使用了新包的 bzip2 压缩器命令。它和很多 UNIX 系统上面的 `bzip2` 命令相似。

```
gopl.io/ch13/bzipper
// Bzipper 读取输入，使用 bzip2 压缩然后输出数据
package main

import (
    "io"
    "log"
    "os"
    "gopl.io/ch13/bzip"
)

func main() {
    w := bzip.NewReader(os.Stdout)
    if _, err := io.Copy(w, os.Stdin); err != nil {
        log.Fatalf("bzipper: %v\n", err)
    }
    if err := w.Close(); err != nil {
        log.Fatalf("bzipper: close: %v\n", err)
    }
}
```

在下面的部分，使用 `bzipper` 来压缩 `/usr/share/dict/words` 文件，这个文件是系统的字典，我们把它从 938 848 个字节压缩到 335 495 个字节，将近是原来的 1/3 大小，然后再使

用系统命令 `bunzip2` 来解压缩它。我们检查压缩前和解压缩后的文件发现它们的 SHA256 散列值是一致的，因此我们相信我们实现的压缩器是正确的。（如果你系统上面没有 `sha256sum` 命令，那么使用练习 4.2 的答案。）

```
$ go build gopl.io/ch13/bzipper
$ wc -c < /usr/share/dict/words
938848
$ sha256sum < /usr/share/dict/words
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
$ ./bzipper < /usr/share/dict/words | wc -c
335405
$ ./bzipper < /usr/share/dict/words | bunzip2 | sha256sum
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
```

我们演示了如何将 C 库链接进 Go 程序中。反过来，可以将 Go 程序编译为静态库然后链接进 C 程序中，也可以编译为动态库通过 C 程序来加载和共享。这里仅讲解了 `cgo` 很浅显的知识，另外关于内存管理、指针、回调、信号处理、字符串、错误处理、析构器以及 `goroutine` 和系统线程的关系等还有很多内容，其中很多内容都很微妙。尤其是，正确地从 GO 传递指针给 C 以及反向传递的过程都很复杂，原因和 13.2 节讨论过的内容相似，并且当前也还没有权威的解释。如果想了解更多的内容，可以访问 <https://golang.org/cmd/cgo>。

练习 13.3： 使用 `sync.Mutex` 来使得 `bzip.Writer` 在多 `goroutine` 的情况下可以安全使用。

练习 13.4： 依赖 C 函数库的实现是有缺点的。请使用另外一个 `bzip.NewWriter` 的纯 Go 实现，它使用 `os/exec` 包将 `/bin/bzip2` 作为一个子进程执行。

13.5 关于安全的注意事项

上一章结尾对反射接口的使用方法给出了警告。这些警告对于本章讲解的包 `unsafe` 更加适用。

高级语言将程序、程序员和神秘的机器指令集隔离开来，并且也隔离了诸如变量在内存中的存储位置，数据类型有多大，数据结构的内存布局，以及关于机器的其他实现细节。由于这个隔离层的存在，我们可以编写安全健壮的代码并且不加改动就可以在任何操作系统上运行。

包 `unsafe` 可以让程序员穿透这层隔离开去使用一些关键的但通过其他方式无法使用到的特性，或者是为了实现更高的性能。付出的代价通常就是程序的可移植性和安全性，所以当你使用 `unsafe` 的时候就得自己承担风险。对于如何使用以及何时使用 `unsafe` 包的功能，建议参考 11.5 节引用的 Knuth 对于过早优化的评论。大多数程序员永远都不需要使用 `unsafe` 包。当然，偶尔还是存在这种情况，其中一些关键代码最好还是通过 `unsafe` 来写。如果在仔细研究和评估后确认 `unsafe` 包是最佳的选择，那么还是尽可能地限制在小范围内使用，这样大多数的程序就不会了解它在哪里使用。

从现在开始，可以将最后两章放到脑后了。开始写一些 Go 程序，避免使用 `reflect` 和 `unsafe` 包，只在你必须使用的时候再复习这两章。

开始快乐地使用 Go 编程吧。我们希望你和我们一样喜欢用 Go 来编程。



Go程序设计语言

The Go Programming Language

“Go是一种开源的程序设计语言，它旨在使得人们能够方便地构建简单、可靠、高效的软件。”

—— Go官网golang.org

本书是Go程序员的权威教程，旨在帮助人们熟练掌握Go语言并充分利用Go的语言特性和标准库来撰写清晰、高效的程序，从而解决现实问题。

本书主要内容

- 第1章介绍Go语言的基础概念，通过十几个完成日常任务（包括读写文件、格式化文本、创建图像以及在Internet客户端和服务器之间通信）的程序来介绍这门语言。
- 接下来讲述Go程序的组成元素（语法、控制流、数据类型），以及程序的组织（包、文件和函数）；后面的几章详细解释了包机制，以及如何高效地利用go工具来构建、测试和维护项目。
- 第6章和第7章介绍Go如何以一种不同寻常的方式来实现面向对象的程序设计，其中方法可以关联到任何用户定义的类型。具体类型和抽象类型（即接口）之间的关系是隐式的，所以一个具体类型可能会实现该类型设计者所没有意识到的接口。
- 第8章和第9章深入讨论并发性方面的重要内容。第8章介绍goroutine和通道的基本机制，并解释CSP模型。第9章讨论并发性中较传统的内容，使用共享变量来实现并发。
- 最后两章探讨Go的低级特性。第12章讲解使用反射的元编程艺术。第13章展示如何运用unsafe包来绕过Go的类型系统，以及如何使用cgo工具来调用C代码。

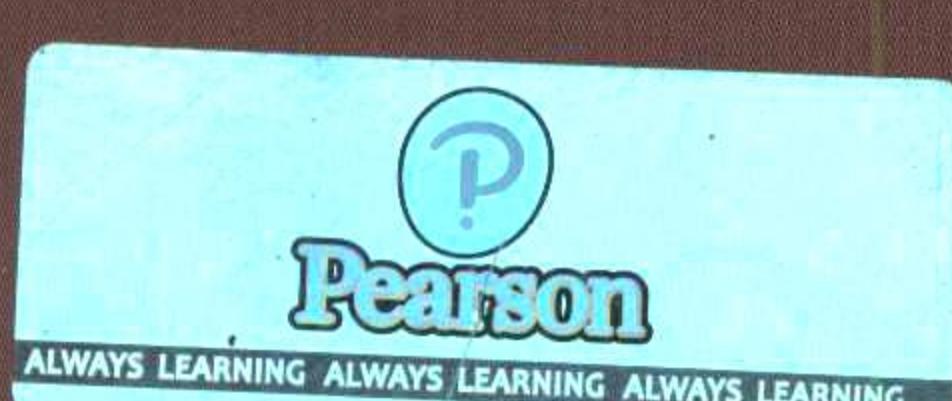
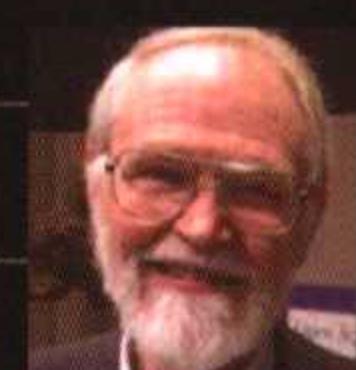
本书包含规范的代码，配有数百个典型示例，涵盖整个Go语言及其最重要的包和广泛的应用。每章都附有一定数量的练习，可以帮助读者加深对Go基础知识的理解。源代码可以从位于http://www.gopl.io/的公开Git仓库下载，并且能够方便地使用go get命令获取、构建和安装。

作者简介

艾伦 A. A. 多诺万 (Alan A. A. Donovan) 谷歌公司Go开发团队成员。他拥有剑桥大学和麻省理工学院计算机科学学士和硕士学位，从1996年开始就在工业界从事软件研发和编程工作。2005年起，他开始在谷歌公司工作，从事基础架构项目研发，是谷歌软件构建工具Blaze的联合设计师。他还创建了用于Go程序静态分析的许多库和工具，包括oracle、godoc -analysis、eg和gorename。

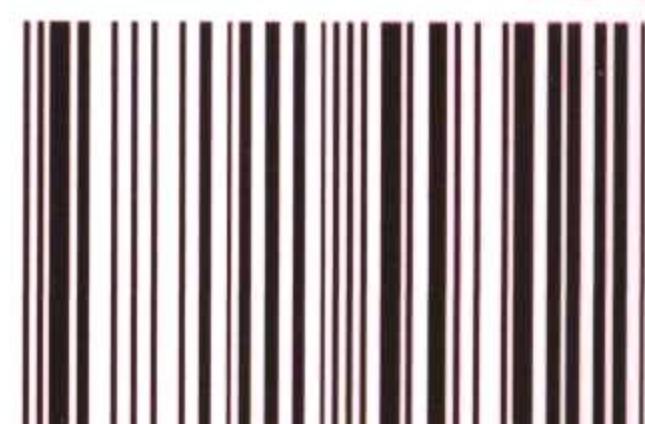


布莱恩 W. 柯尼汉 (Brian W. Kernighan) 现为普林斯顿大学计算机科学系教授。他与C语言的发明人Dennis Ritchie共同合作撰写了《C程序设计语言》。1969~2000年间，他是贝尔实验室计算机科学研究中心技术团队的成员，同时他也是开发UNIX的主要贡献者。他是AWK和AMPL编程语言的作者之一，AWK中的K说的就是Kernighan。



上架指导：计算机/程序设计

ISBN 978-7-111-55842-2



9 787111 558422 >

定价：79.00元



www.pearson.com

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn