```
04
05 import (
06     "fmt"
07 )
08
09 // user 在程序里定义一个用户类型
10 type user struct {
11     name  string
12     email string
13 }
14
15 // notify 实现了一个可以通过 user 类型值的指针
16 // 调用的方法
17 func (u *user) notify() {
18     fmt.Printf("Sending user email to %s<%s>\n",
19     u.name,
20     u.email)
21 }
22
23 // admin 代表一个拥有权限的管理员用户
24 type admin struct {
25     user  // 嵌入类型
26     level string
27 }
28
29 // main 是应用程序的入口
30 func main() {
31     // 创建一个 admin 用户
32     ad := admin{
33         user: user{
34             name:  "john smith",
35             email: "john@yahoo.com",
36         },
37         level: "super",
38     }
39
40     // 我们可以直接访问内部类型的方法
41     ad.user.notify()
42
43     // 内部类型的方法也被提升到外部类型
44     ad.notify()
45 }
```

在代码清单 5-50 中，我们的程序演示了如何嵌入一个类型，并访问嵌入类型的标识符。我们从第 10 行和第 24 行中的两个结构类型的声明开始，如代码清单 5-51 所示。

代码清单 5-51　listing50.go：第 09 行到第 13 行，第 23 行到第 27 行

```
09 // user 在程序里定义一个用户类型
10 type user struct {
11     name  string
12     email string
13 }
```

```
23 // admin 代表一个拥有权限的管理员用户
24 type admin struct {
25     user  // 嵌入类型
26     level string
27 }
```

在代码清单 5-51 的第 10 行，我们声明了一个名为 user 的结构类型。在第 24 行，我们声明了另一个名为 admin 的结构类型。在声明 admin 类型的第 25 行，我们将 user 类型嵌入 admin 类型里。要嵌入一个类型，只需要声明这个类型的名字就可以了。在第 26 行，我们声明了一个名为 level 的字段。注意声明字段和嵌入类型在语法上的不同。

一旦我们将 user 类型嵌入 admin，我们就可以说 user 是外部类型 admin 的内部类型。有了内部类型和外部类型这两个概念，就能更容易地理解这两种类型之间的关系。

代码清单 5-52 展示了使用 user 类型的指针接收者声明名为 notify 的方法。这个方法只是显示一行友好的信息，表示将邮件发给了特定的用户以及邮件地址。

代码清单 5-52　listing50.go：第 15 行到第 21 行

```
15 // notify 实现了一个可以通过 user 类型值的指针
16 // 调用的方法
17 func (u *user) notify() {
18     fmt.Printf("Sending user email to %s<%s>\n",
19     u.name,
20     u.email)
21 }
```

现在，让我们来看一下 main 函数，如代码清单 5-53 所示。

代码清单 5-53　listing50.go：第 30 行到第 45 行

```
30 func main() {
31     // 创建一个 admin 用户
32     ad := admin{
33         user: user{
34             name:  "john smith",
35             email: "john@yahoo.com",
36         },
37         level: "super",
38     }
39
40     // 我们可以直接访问内部类型的方法
41     ad.user.notify()
42
43     // 内部类型的方法也被提升到外部类型
44     ad.notify()
45 }
```

代码清单 5-53 中的 main 函数展示了嵌入类型背后的机制。在第 32 行，创建了一个 admin 类型的值。内部类型的初始化是用结构字面量完成的。通过内部类型的名字可以访问内部类型，

如代码清单 5-54 所示。对外部类型来说，内部类型总是存在的。这就意味着，虽然没有指定内部类型对应的字段名，还是可以使用内部类型的类型名，来访问到内部类型的值。

代码清单 5-54 listing50.go：第 40 行到第 41 行

```
40      // 我们可以直接访问内部类型的方法
41      ad.user.notify()
```

在代码清单 5-54 中第 41 行，可以看到对 notify 方法的调用。这个调用是通过直接访问内部类型 user 来完成的。这展示了内部类型是如何存在于外部类型内，并且总是可访问的。不过，借助内部类型提升，notify 方法也可以直接通过 ad 变量来访问，如代码清单 5-55 所示。

代码清单 5-55 listing50.go：第 43 行到第 45 行

```
43      // 内部类型的方法也被提升到外部类型
44      ad.notify()
45  }
```

代码清单 5-55 的第 44 行中展示了直接通过外部类型的变量来调用 notify 方法。由于内部类型的标识符提升到了外部类型，我们可以直接通过外部类型的值来访问内部类型的标识符。让我们修改一下这个例子，加入一个接口，如代码清单 5-56 所示。

代码清单 5-56 listing56.go

```
01  // 这个示例程序展示如何将嵌入类型应用于接口
02  package main
03
04  import (
05      "fmt"
06  )
07
08  // notifier 是一个定义了
09  // 通知类行为的接口
10  type notifier interface {
11      notify()
12  }
13
14  // user 在程序里定义一个用户类型
15  type user struct {
16      name  string
17      email string
18  }
19
20  // 通过 user 类型值的指针
21  // 调用的方法
22  func (u *user) notify() {
23      fmt.Printf("Sending user email to %s<%s>\n",
24      u.name,
25      u.email)
26  }
```

```
27
28 // admin 代表一个拥有权限的管理员用户
29 type admin struct {
30     user
31     level string
32 }
33
34 // main 是应用程序的入口
35 func main() {
36     // 创建一个 admin 用户
37     ad := admin{
38         user: user{
39             name:  "john smith",
40             email: "john@yahoo.com",
41         },
42         level: "super",
43     }
44
45     // 给 admin 用户发送一个通知
46     // 用于实现接口的内部类型的方法，被提升到
47     // 外部类型
48     sendNotification(&ad)
49 }
50
51 // sendNotification 接受一个实现了 notifier 接口的值
52 // 并发送通知
53 func sendNotification(n notifier) {
54     n.notify()
55 }
```

代码清单 5-56 所示的示例程序的大部分和之前的程序相同，只有一些小变化，如代码清单 5-57 所示。

代码清单 5-57　第 08 行到第 12 行，第 51 行到第 55 行

```
08 // notifier 是一个定义了
09 // 通知类行为的接口
10 type notifier interface {
11     notify()
12 }

51 // sendNotification 接受一个实现了 notifier 接口的值
52 // 并发送通知
53 func sendNotification(n notifier) {
54     n.notify()
55 }
```

在代码清单 5-57 的第 08 行，声明了一个 `notifier` 接口。之后在第 53 行，有一个 `sendNotification` 函数，接受 `notifier` 类型的接口的值。从代码可以知道，`user` 类型之前声明了名为 `notify` 的方法，该方法使用指针接收者实现了 `notifier` 接口。之后，让我们看一下 `main` 函数的改动，如代码清单 5-58 所示。

代码清单 5-58 listing56.go：第 35 行到第 49 行

```
35 func main() {
36     // 创建一个 admin 用户
37     ad := admin{
38         user: user{
39             name:  "john smith",
40             email: "john@yahoo.com",
41         },
42         level: "super",
43     }
44
45     // 给 admin 用户发送一个通知
46     // 用于实现接口的内部类型的方法，被提升到
47     // 外部类型
48     sendNotification(&ad)
49 }
```

这里才是事情变得有趣的地方。在代码清单 5-58 的第 37 行，我们创建了一个名为 ad 的变量，其类型是外部类型 admin。这个类型内部嵌入了 user 类型。之后第 48 行，我们将这个外部类型变量的地址传给 sendNotification 函数。编译器认为这个指针实现了 notifier 接口，并接受了这个值的传递。不过如果看一下整个示例程序，就会发现 admin 类型并没有实现这个接口。

由于内部类型的提升，内部类型实现的接口会自动提升到外部类型。这意味着由于内部类型的实现，外部类型也同样实现了这个接口。运行这个示例程序，会得到代码清单 5-59 所示的输出。

代码清单 5-59 listing56.go 的输出

```
20 // 通过 user 类型值的指针
21 // 调用的方法
22 func (u *user) notify() {
23     fmt.Printf("Sending user email to %s<%s>\n",
24     u.name,
25     u.email)
26 }

Output:
Sending user email to john smith<john@yahoo.com>
```

可以在代码清单 5-59 中看到内部类型的实现被调用。

如果外部类型并不需要使用内部类型的实现，而想使用自己的一套实现，该怎么办？让我们看另一个示例程序是如何解决这个问题的，如代码清单 5-60 所示。

代码清单 5-60 listing60.go

```
01 // 这个示例程序展示当内部类型和外部类型要
02 // 实现同一个接口时的做法
03 package main
```

```
04
05 import (
06     "fmt"
07 )
08
08 // notifier 是一个定义了
09 // 通知类行为的接口
11 type notifier interface {
12     notify()
13 }
14
15 // user 在程序里定义一个用户类型
16 type user struct {
17     name  string
18     email string
19 }
20
21 // 通过 user 类型值的指针
22 // 调用的方法
23 func (u *user) notify() {
24     fmt.Printf("Sending user email to %s<%s>\n",
25         u.name,
26         u.email)
27 }
28
29 // admin 代表一个拥有权限的管理员用户
30 type admin struct {
31     user
32     level string
33 }
34
35 // 通过 admin 类型值的指针
36 // 调用的方法
37 func (a *admin) notify() {
38     fmt.Printf("Sending admin email to %s<%s>\n",
39         a.name,
40         a.email)
41 }
42
43 // main 是应用程序的入口
44 func main() {
45     // 创建一个 admin 用户
46     ad := admin{
47         user: user{
48             name:  "john smith",
49             email: "john@yahoo.com",
50         },
51         level: "super",
52     }
53
54     // 给 admin 用户发送一个通知
55     // 接口的嵌入的内部类型实现并没有提升到
56     // 外部类型
57     sendNotification(&ad)
```

```
58
59      // 我们可以直接访问内部类型的方法
60      ad.user.notify()
61
62      // 内部类型的方法没有被提升
63      ad.notify()
64  }
65
66  // sendNotification 接受一个实现了 notifier 接口的值
67  // 并发送通知
68  func sendNotification(n notifier) {
69      n.notify()
70  }
```

代码清单 5-60 所示的示例程序的大部分和之前的程序相同，只有一些小变化，如代码清单 5-61 所示。

代码清单 5-61 listing60.go：第 35 行到第 41 行

```
35  // 通过 admin 类型值的指针
36  // 调用的方法
37  func (a *admin) notify() {
38      fmt.Printf("Sending admin email to %s<%s>\n",
39          a.name,
40          a.email)
41  }
```

这个示例程序为 admin 类型增加了 notifier 接口的实现。当 admin 类型的实现被调用时，会显示"Sending admin email"。作为对比，user 类型的实现被调用时，会显示"Sending user email"。

main 函数里也有一些变化，如代码清单 5-62 所示。

代码清单 5-62 listing60.go：第 43 行到第 64 行

```
43  // main 是应用程序的入口
44  func main() {
45      // 创建一个 admin 用户
46      ad := admin{
47          user: user{
48              name:  "john smith",
49              email: "john@yahoo.com",
50          },
51          level: "super",
52      }
53
54      // 给 admin 用户发送一个通知
55      // 接口的嵌入的内部类型实现并没有提升到
56      // 外部类型
57      sendNotification(&ad)
58
59      // 我们可以直接访问内部类型的方法
```

```
60      ad.user.notify()
61
62      // 内部类型的方法没有被提升
63      ad.notify()
64  }
```

代码清单 5-62 的第 46 行，我们再次创建了外部类型的变量 ad。在第 57 行，将 ad 变量的地址传给 sendNotification 函数，这个指针实现了接口所需要的方法集。在第 60 行，代码直接访问 user 内部类型，并调用 notify 方法。最后，在第 63 行，使用外部类型变量 ad 来调用 notify 方法。当查看这个示例程序的输出（如代码清单 5-63 所示）时，就会看到区别。

代码清单 5-63　listing60.go 的输出

```
Sending admin email to john smith<john@yahoo.com>
Sending user email to john smith<john@yahoo.com>
Sending admin email to john smith<john@yahoo.com>
```

这次我们看到了 admin 类型是如何实现 notifier 接口的，以及如何由 sendNotification 函数以及直接使用外部类型的变量 ad 来执行 admin 类型实现的方法。这表明，如果外部类型实现了 notify 方法，内部类型的实现就不会被提升。不过内部类型的值一直存在，因此还可以通过直接访问内部类型的值，来调用没有被提升的内部类型实现的方法。

5.6　公开或未公开的标识符

要想设计出好的 API，需要使用某种规则来控制声明后的标识符的可见性。Go 语言支持从包里公开或者隐藏标识符。通过这个功能，让用户能按照自己的规则控制标识符的可见性。在第 3 章讨论包的时候，谈到了如何从一个包引入标识符到另一个包。有时候，你可能不希望公开包里的某个类型、函数或者方法这样的标识符。在这种情况，需要一种方法，将这些标识符声明为包外不可见，这时需要将这些标识符声明为未公开的。

让我们用一个示例程序来演示如何隐藏包里未公开的标识符，如代码清单 5-64 所示。

代码清单 5-64　listing64/

```
counters/counters.go
--------------------------------------------------------------------------
01 // counters 包提供告警计数器的功能
02 package counters
03
04 // alertCounter 是一个未公开的类型
05 // 这个类型用于保存告警计数
06 type alertCounter int

listing64.go
--------------------------------------------------------------------------
01 // 这个示例程序展示无法从另一个包里
02 // 访问未公开的标识符
```

```
03 package main
04
05 import (
06     "fmt"
07
08     "github.com/goinaction/code/chapter5/listing64/counters"
09 )
10
11 // main 是应用程序的入口
12 func main() {
13     // 创建一个未公开的类型的变量
14     // 并将其初始化为 10
15     counter := counters.alertCounter(10)
16
17     // ./listing64.go:15: 不能引用未公开的名字
18     //                 counters.alertCounter
19     // ./listing64.go:15: 未定义：counters.alertCounter
20
21     fmt.Printf("Counter: %d\n", counter)
22 }
```

这个示例程序有两个代码文件。一个代码文件名字为 counters.go，保存在 `counters` 包里；另一个代码文件名字为 listing64.go，导入了 `counters` 包。让我们先从 `counters` 包里的代码开始，如代码清单 5-65 所示。

代码清单 5-65　`counters`/counters.go

```
01 // counters 包提供告警计数器的功能
02 package counters
03
04 // alertCounter 是一个未公开的类型
05 // 这个类型用于保存告警计数
06 type alertCounter int
```

代码清单 5-65 展示了只属于 `counters` 包的代码。你可能会首先注意到第 02 行。直到现在，之前所有的示例程序都使用了 `package main`，而这里用到的是 `package counters`。当要写的代码属于某个包时，好的实践是使用与代码所在文件夹一样的名字作为包名。所有的 Go 工具都会利用这个习惯，所以最好遵守这个好的实践。

在 `counters` 包里，我们在第 06 行声明了唯一一个名为 `alertCounter` 的标识符。这个标识符是一个使用 `int` 作为基础类型的类型。需要注意的是，这是一个未公开的标识符。

当一个标识符的名字以小写字母开头时，这个标识符就是未公开的，即包外的代码不可见。如果一个标识符以大写字母开头，这个标识符就是公开的，即被包外的代码可见。让我们看一下导入这个包的代码，如代码清单 5-66 所示。

代码清单 5-66　listing64.go

```
01 // 这个示例程序展示无法从另一个包里
02 // 访问未公开的标识符
```

```
03 package main
04
05 import (
06     "fmt"
07
08     "github.com/goinaction/code/chapter5/listing64/counters"
09 )
10
11 // main 是应用程序的入口
12 func main() {
13     // 创建一个未公开的类型的变量
14     // 并将其初始化为 10
15     counter := counters.alertCounter(10)
16
17     // ./listing64.go:15: 不能引用未公开的名字
18     //                                          counters.alertCounter
19     // ./listing64.go:15: 未定义：counters.alertCounter
20
21     fmt.Printf("Counter: %d\n", counter)
22 }
```

代码清单 5-66 中的 listing64.go 的代码在第 03 行声明了 `main` 包，之后在第 08 行导入了 `counters` 包。在这之后，我们跳到 `main` 函数里的第 15 行，如代码清单 5-67 所示。

代码清单 5-67　listing64.go：第 13 到 19 行

```
13     // 创建一个未公开的类型的变量
14     // 并将其初始化为 10
15     counter := counters.alertCounter(10)
16
17     // ./listing64.go:15: 不能引用未公开的名字
18     //                                          counters.alertCounter
19     // ./listing64.go:15: 未定义：counters.alertCounter
```

在代码清单 5-67 的第 15 行，代码试图创建未公开的 `alertCounter` 类型的值。不过这段代码会造成第 15 行展示的编译错误，这个编译错误表明第 15 行的代码无法引用 `counters.alertCounter` 这个未公开的标识符。这个标识符是未定义的。

由于 `counters` 包里的 `alertCounter` 类型是使用小写字母声明的，所以这个标识符是未公开的，无法被 listing64.go 的代码访问。如果我们把这个类型改为用大写字母开头，那么就不会产生编译器错误。让我们看一下新的示例程序，如代码清单 5-68 所示，这个程序在 `counters` 包里实现了工厂函数。

代码清单 5-68　listing68/

```
counters/counters.go
-------------------------------------------------------------------------
01 // counters 包提供告警计数器的功能
02 package counters
03
04 // alertCounter 是一个未公开的类型
```

```
05 // 这个类型用于保存告警计数
06 type alertCounter int
07
08 // New 创建并返回一个未公开的
09 // alertCounter 类型的值
10 func New(value int) alertCounter {
11     return alertCounter(value)
12 }
```

```
listing68.go
------------------------------------------------------------------------
01 // 这个示例程序展示如何访问另一个包的未公开的
02 // 标识符的值
03 package main
04
05 import (
06     "fmt"
07
08     "github.com/goinaction/code/chapter5/listing68/counters"
09 )
10
11 // main 是应用程序的入口
12 func main() {
13     // 使用 counters 包公开的 New 函数来创建
14     // 一个未公开的类型的变量
15     counter := counters.New(10)
16
17     fmt.Printf("Counter: %d\n", counter)
18 }
```

这个例子已经修改为使用工厂函数来创建一个未公开的 `alertCounter` 类型的值。让我们先看一下 `counters` 包的代码，如代码清单 5-69 所示。

代码清单 5-69　counters/counters.go

```
01 // counters 包提供告警计数器的功能
02 package counters
03
04 // alertCounter 是一个未公开的类型
05 // 这个类型用于保存告警计数
06 type alertCounter int
07
08 // New 创建并返回一个未公开的
09 // alertCounter 类型的值
10 func New(value int) alertCounter {
11     return alertCounter(value)
12 }
```

代码清单 5-69 展示了我们对 `counters` 包的改动。`alertCounter` 类型依旧是未公开的，不过现在在第 10 行增加了一个名为 `New` 的新函数。将工厂函数命名为 `New` 是 Go 语言的一个习惯。这个 `New` 函数做了些有意思的事情：它创建了一个未公开的类型的值，并将这个值返回给

调用者。让我们看一下 listing68.go 的 `main` 函数，如代码清单 5-70 所示。

代码清单 5-70　listing68.go

```
11 // main 是应用程序的入口
12 func main() {
13     // 使用 counters 包公开的 New 函数来创建
14     // 一个未公开的类型的变量
15     counter := counters.New(10)
16
17     fmt.Printf("Counter: %d\n", counter)
18 }
```

在代码清单 5-70 的第 15 行，可以看到对 `counters` 包里 `New` 函数的调用。这个 `New` 函数返回的值被赋给一个名为 `counter` 的变量。这个程序可以编译并且运行，但为什么呢？`New` 函数返回的是一个未公开的 `alertCounter` 类型的值，而 `main` 函数能够接受这个值并创建一个未公开的类型的变量。

要让这个行为可行，需要两个理由。第一，公开或者未公开的标识符，不是一个值。第二，短变量声明操作符，有能力捕获引用的类型，并创建一个未公开的类型的变量。永远不能显式创建一个未公开的类型的变量，不过短变量声明操作符可以这么做。

让我们看一个新例子，这个例子展示了这些可见的规则是如何影响到结构里的字段，如代码清单 5-71 所示。

代码清单 5-71　listing71/

```
entities/entities.go
------------------------------------------------------------------------
01 // entities 包包含系统中
02 // 与人有关的类型
03 package entities
04
05 // User 在程序里定义一个用户类型
06 type User struct {
07     Name  string
08     email string
09 }

listing71.go
------------------------------------------------------------------------
01 // 这个示例程序展示公开的结构类型中未公开的字段
02 // 无法直接访问
03 package main
04
05 import (
06     "fmt"
07
08     "github.com/goinaction/code/chapter5/listing71/entities"
09 )
10
```

```
11 // main 是应用程序的入口
12 func main() {
13     // 创建 entities 包中的 User 类型的值
14     u := entities.User{
15         Name:  "Bill",
16         email: "bill@email.com",
17     }
18
19     // ./example69.go:16: 结构字面量中结构 entities.User
20     //                 的字段'email'未知
21
22     fmt.Printf("User: %v\n", u)
23 }
```

代码清单 5-71 中的代码有一些微妙的变化。现在我们有一个名为 `entities` 的包，声明了名为 `User` 的结构类型，如代码清单 5-72 所示。

代码清单 5-72　entities/entities.go

```
01 // entities 包包含系统中
02 // 与人有关的类型
03 package entities
04
05 // User 在程序里定义一个用户类型
06 type User struct {
07     Name  string
08     email string
09 }
```

代码清单 5-72 的第 06 行中的 `User` 类型被声明为公开的类型。`User` 类型里声明了两个字段，一个名为 `Name` 的公开的字段，一个名为 `email` 的未公开的字段。让我们看一下 listing71.go 的代码，如代码清单 5-73 所示。

代码清单 5-73　listing71.go

```
01 // 这个示例程序展示公开的结构类型中未公开的字段
02 // 无法直接访问
03 package main
04
05 import (
06     "fmt"
07
08     "github.com/goinaction/code/chapter5/listing71/entities"
09 )
10
11 // main 是程序的入口
12 func main() {
13     // 创建 entities 包中的 User 类型的值
14     u := entities.User{
15         Name:  "Bill",
16         email: "bill@email.com",
17     }
```

```
18
19      // ./example69.go:16: 结构字面量中结构 entities.User
20      //                  的字段'email'未知
21
22      fmt.Printf("User: %v\n", u)
23  }
```

代码清单 5-73 的第 08 行导入了 `entities` 包。在第 14 行声明了 `entities` 包中的公开的类型 `User` 的名为 `u` 的变量，并对该字段做了初始化。不过这里有一个问题。第 16 行的代码试图初始化未公开的字段 `email`，所以编译器抱怨这是个未知的字段。因为 `email` 这个标识符未公开，所以它不能在 `entities` 包外被访问。

让我们看最后一个例子，这个例子展示了公开和未公开的内嵌类型是如何工作的，如代码清单 5-74 所示。

代码清单 5-74　listing74/

```
entities/entities.go
-------------------------------------------------------------------------
01 // entities 包包含系统中
02 // 与人有关的类型
03 package entities
04
05 // user 在程序里定义一个用户类型
06 type user struct {
07      Name  string
08      Email string
09 }
10
11 // Admin 在程序里定义了管理员
12 type Admin struct {
13      user   // 嵌入的类型是未公开的
14      Rights int
15 }


listing74.go
-------------------------------------------------------------------------
01 // 这个示例程序展示公开的结构类型中如何访问
02 // 未公开的内嵌类型的例子
03 package main
04
05 import (
06      "fmt"
07
08      "github.com/goinaction/code/chapter5/listing74/entities"
09 )
10
11 // main 是应用程序的入口
12 func main() {
13      // 创建 entities 包中的 Admin 类型的值
14      a := entities.Admin{
```

```
15          Rights: 10,
16      }
17
18      // 设置未公开的内部类型的
19      // 公开字段的值
20      a.Name = "Bill"
21      a.Email = "bill@email.com"
22
23      fmt.Printf("User: %v\n", a)
24  }
```

现在，在代码清单 5-74 里，`entities` 包包含两个结构类型，如代码清单 5-75 所示。

代码清单 5-75　`entities`/entities.go

```
01 // entities 包包含系统中
02 // 与人有关的类型
03 package entities
04
05 // user 在程序里定义一个用户类型
06 type user struct {
07      Name  string
08      Email string
09 }
10
11 // Admin 在程序里定义了管理员
12 type Admin struct {
13      user   // 嵌入的类型未公开
14      Rights int
15 }
```

在代码清单 5-75 的第 06 行，声明了一个未公开的结构类型 `user`。这个类型包括两个公开的字段 `Name` 和 `Email`。在第 12 行，声明了一个公开的结构类型 `Admin`。`Admin` 有一个名为 `Rights` 的公开的字段，而且嵌入一个未公开的 `user` 类型。让我们看一下 listing74.go 的 `main` 函数，如代码清单 5-76 所示。

代码清单 5-76　listing74.go：第 11 到 24 行

```
11 // main 是应用程序的入口
12 func main() {
13      // 创建 entities 包中的 Admin 类型的值
14      a := entities.Admin{
15          Rights: 10,
16      }
17
18      // 设置未公开的内部类型的
19      // 公开字段的值
20      a.Name = "Bill"
21      a.Email = "bill@email.com"
22
23      fmt.Printf("User: %v\n", a)
24  }
```

让我们从代码清单 5-76 的第 14 行的 `main` 函数开始。这个函数创建了 `entities` 包中的 `Admin` 类型的值。由于内部类型 `user` 是未公开的，这段代码无法直接通过结构字面量的方式初始化该内部类型。不过，即便内部类型是未公开的，内部类型里声明的字段依旧是公开的。既然内部类型的标识符提升到了外部类型，这些公开的字段也可以通过外部类型的字段的值来访问。

因此，在第 20 行和第 21 行，来自未公开的内部类型的字段 `Name` 和 `Email` 可以通过外部类型的变量 `a` 被访问并被初始化。因为 `user` 类型是未公开的，所以这里没有直接访问内部类型。

5.7 小结

- 使用关键字 `struct` 或者通过指定已经存在的类型，可以声明用户定义的类型。
- 方法提供了一种给用户定义的类型增加行为的方式。
- 设计类型时需要确认类型的本质是原始的，还是非原始的。
- 接口是声明了一组行为并支持多态的类型。
- 嵌入类型提供了扩展类型的能力，而无需使用继承。
- 标识符要么是从包里公开的，要么是在包里未公开的。

第 6 章　并发

本章主要内容

- 使用 goroutine 运行程序
- 检测并修正竞争状态
- 利用通道共享数据

通常程序会被编写为一个顺序执行并完成一个独立任务的代码。如果没有特别的需求，最好总是这样写代码，因为这种类型的程序通常很容易写，也很容易维护。不过也有一些情况下，并行执行多个任务会有更大的好处。一个例子是，Web 服务需要在各自独立的套接字（socket）上同时接收多个数据请求。每个套接字请求都是独立的，可以完全独立于其他套接字进行处理。具有并行执行多个请求的能力可以显著提高这类系统的性能。考虑到这一点，Go 语言的语法和运行时直接内置了对并发的支持。

Go 语言里的并发指的是能让某个函数独立于其他函数运行的能力。当一个函数创建为 goroutine 时，Go 会将其视为一个独立的工作单元。这个单元会被调度到可用的逻辑处理器上执行。Go 语言运行时的调度器是一个复杂的软件，能管理被创建的所有 goroutine 并为其分配执行时间。这个调度器在操作系统之上，将操作系统的线程与语言运行时的逻辑处理器绑定，并在逻辑处理器上运行 goroutine。调度器在任何给定的时间，都会全面控制哪个 goroutine 要在哪个逻辑处理器上运行。

Go 语言的并发同步模型来自一个叫作*通信顺序进程*（Communicating Sequential Processes，CSP）的范型（paradigm）。CSP 是一种消息传递模型，通过在 goroutine 之间传递数据来传递消息，而不是对数据进行加锁来实现同步访问。用于在 goroutine 之间同步和传递数据的关键数据类型叫作*通道*（channel）。对于没有使用过通道写并发程序的程序员来说，通道会让他们感觉神奇而兴奋。希望读者使用后也能有这种感觉。使用通道可以使编写并发程序更容易，也能够让并发程序出错更少。

6.1　并发与并行

让我们先来学习一下抽象程度较高的概念：什么是操作系统的*线程*（thread）和*进程*（process）。

这会有助于后面理解 Go 语言运行时调度器如何利用操作系统来并发运行 goroutine。当运行一个应用程序（如一个 IDE 或者编辑器）的时候，操作系统会为这个应用程序启动一个进程。可以将这个进程看作一个包含了应用程序在运行中需要用到和维护的各种资源的容器。

图 6-1 展示了一个包含所有可能分配的常用资源的进程。这些资源包括但不限于内存地址空间、文件和设备的句柄以及线程。一个线程是一个执行空间，这个空间会被操作系统调度来运行函数中所写的代码。每个进程至少包含一个线程，每个进程的初始线程被称作主线程。因为执行这个线程的空间是应用程序的本身的空间，所以当主线程终止时，应用程序也会终止。操作系统将线程调度到某个处理器上运行，这个处理器并不一定是进程所在的处理器。不同操作系统使用的线程调度算法一般都不一样，但是这种不同会被操作系统屏蔽，并不会展示给程序员。

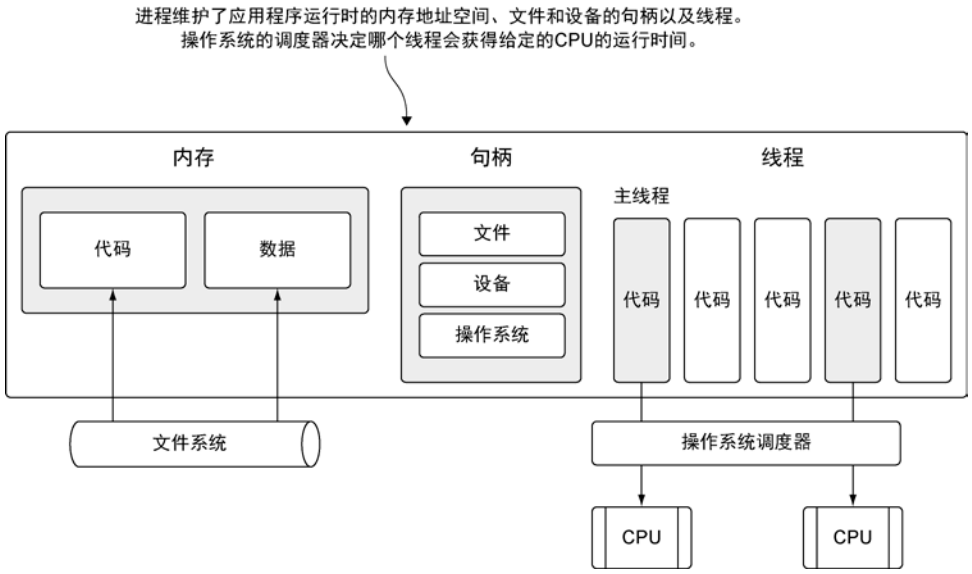

图 6-1　一个运行的应用程序的进程和线程的简要描绘

操作系统会在物理处理器上调度线程来运行，而 Go 语言的运行时会在逻辑处理器上调度goroutine来运行。每个逻辑处理器都分别绑定到单个操作系统线程。在 1.5 版本[①]上，Go语言的运行时默认会为每个可用的物理处理器分配一个逻辑处理器。在 1.5 版本之前的版本中，默认给整个应用程序只分配一个逻辑处理器。这些逻辑处理器会用于执行所有被创建的goroutine。即便只有一个逻辑处理器，Go也可以以神奇的效率和性能，并发调度无数个goroutine。

在图 6-2 中，可以看到操作系统线程、逻辑处理器和本地运行队列之间的关系。如果创建一个 goroutine 并准备运行，这个 goroutine 就会被放到调度器的全局运行队列中。之后，调度器就将这些队列中的 goroutine 分配给一个逻辑处理器，并放到这个逻辑处理器对应的本地运行队列

① 直到目前最新的 1.8 版本都是同一逻辑。可预见的未来版本也会保持这个逻辑。——译者注

中。本地运行队列中的 goroutine 会一直等待直到自己被分配的逻辑处理器执行。

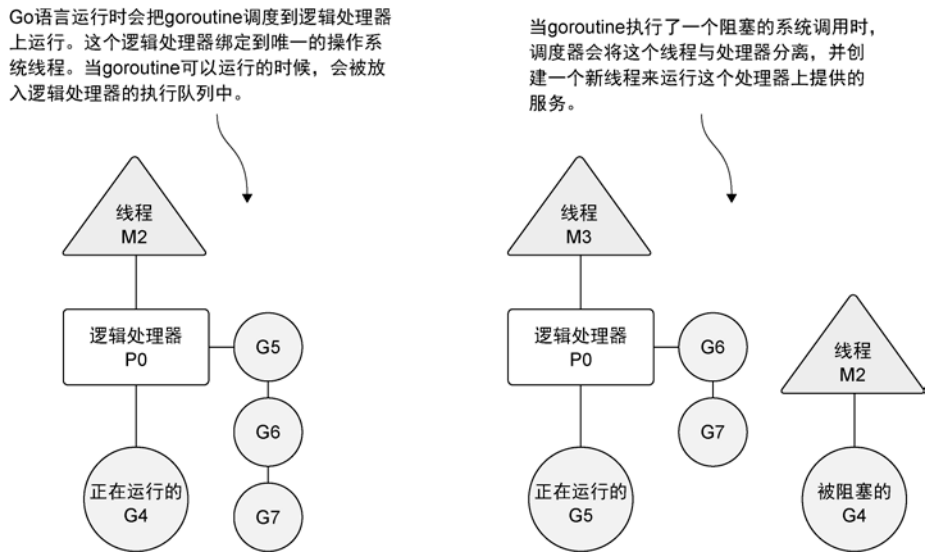

图 6-2　Go 调度器如何管理 goroutine

有时，正在运行的 goroutine 需要执行一个阻塞的系统调用，如打开一个文件。当这类调用发生时，线程和 goroutine 会从逻辑处理器上分离，该线程会继续阻塞，等待系统调用的返回。与此同时，这个逻辑处理器就失去了用来运行的线程。所以，调度器会创建一个新线程，并将其绑定到该逻辑处理器上。之后，调度器会从本地运行队列里选择另一个 goroutine 来运行。一旦被阻塞的系统调用执行完成并返回，对应的 goroutine 会放回到本地运行队列，而之前的线程会保存好，以便之后可以继续使用。

如果一个 goroutine 需要做一个网络 I/O 调用，流程上会有些不一样。在这种情况下，goroutine 会和逻辑处理器分离，并移到集成了网络轮询器的运行时。一旦该轮询器指示某个网络读或者写操作已经就绪，对应的 goroutine 就会重新分配到逻辑处理器上来完成操作。调度器对可以创建的逻辑处理器的数量没有限制，但语言运行时默认限制每个程序最多创建 10 000 个线程。这个限制值可以通过调用 `runtime/debug` 包的 `SetMaxThreads` 方法来更改。如果程序试图使用更多的线程，就会崩溃。

并发（concurrency）不是并行（parallelism）。并行是让不同的代码片段同时在不同的物理处理器上执行。并行的关键是同时做很多事情，而并发是指同时管理很多事情，这些事情可能只做了一半就被暂停去做别的事情了。在很多情况下，并发的效果比并行好，因为操作系统和硬件的总资源一般很少，但能支持系统同时做很多事情。这种“使用较少的资源做更多的事情”的哲学，也是指导 Go 语言设计的哲学。

如果希望让 goroutine 并行，必须使用多于一个逻辑处理器。当有多个逻辑处理器时，调度器

会将 goroutine 平等分配到每个逻辑处理器上。这会让 goroutine 在不同的线程上运行。不过要想真的实现并行的效果，用户需要让自己的程序运行在有多个物理处理器的机器上。否则，哪怕 Go 语言运行时使用多个线程，goroutine 依然会在同一个物理处理器上并发运行，达不到并行的效果。

图 6-3 展示了在一个逻辑处理器上并发运行 goroutine 和在两个逻辑处理器上并行运行两个并发的 goroutine 之间的区别。调度器包含一些聪明的算法，这些算法会随着 Go 语言的发布被更新和改进，所以不推荐盲目修改语言运行时对逻辑处理器的默认设置。如果真的认为修改逻辑处理器的数量可以改进性能，也可以对语言运行时的参数进行细微调整。后面会介绍如何做这种修改。

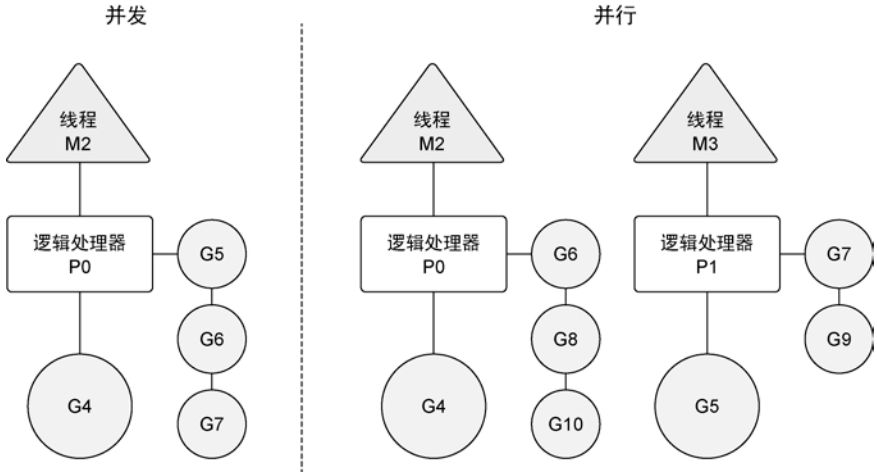

图 6-3　并发和并行的区别

6.2　goroutine

让我们再深入了解一下调度器的行为，以及调度器是如何创建 goroutine 并管理其寿命的。我们会先通过在一个逻辑处理器上运行的例子来讲解，再来讨论如何让 goroutine 并行运行。代码清单 6-1 所示的程序会创建两个 goroutine，以并发的形式分别显示大写和小写的英文字母。

代码清单 6-1　listing01.go

```
01 // 这个示例程序展示如何创建 goroutine
02 // 以及调度器的行为
03 package main
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09 )
10
11 // main 是所有 Go 程序的入口
```

```
12 func main() {
13     // 分配一个逻辑处理器给调度器使用
14     runtime.GOMAXPROCS(1)
15
16     // wg 用来等待程序完成
17     // 计数加 2，表示要等待两个 goroutine
18     var wg sync.WaitGroup
19     wg.Add(2)
20
21     fmt.Println("Start Goroutines")
22
23     // 声明一个匿名函数，并创建一个 goroutine
24     go func() {
25         // 在函数退出时调用 Done 来通知 main 函数工作已经完成
26         defer wg.Done()
27
28         // 显示字母表 3 次
29         for count := 0; count < 3; count++ {
30             for char := 'a'; char < 'a'+26; char++ {
31                 fmt.Printf("%c ", char)
32             }
33         }
34     }()
35
36     // 声明一个匿名函数，并创建一个 goroutine
37     go func() {
38         // 在函数退出时调用 Done 来通知 main 函数工作已经完成
39         defer wg.Done()
40
41         // 显示字母表 3 次
42         for count := 0; count < 3; count++ {
43             for char := 'A'; char < 'A'+26; char++ {
44                 fmt.Printf("%c ", char)
45             }
46         }
47     }()
48
49     // 等待 goroutine 结束
50     fmt.Println("Waiting To Finish")
51     wg.Wait()
52
53     fmt.Println("\nTerminating Program")
54 }
```

在代码清单 6-1 的第 14 行，调用了 runtime 包的 GOMAXPROCS 函数。这个函数允许程序更改调度器可以使用的逻辑处理器的数量。如果不想在代码里做这个调用，也可以通过修改和这个函数名字一样的环境变量的值来更改逻辑处理器的数量。给这个函数传入 1，是通知调度器只能为该程序使用一个逻辑处理器。

在第 24 行和第 37 行，我们声明了两个匿名函数，用来显示英文字母表。第 24 行的函数显示小写字母表，而第 37 行的函数显示大写字母表。这两个函数分别通过关键字 go 创建 goroutine 来执行。根据代码清单 6-2 中给出的输出可以看到，每个 goroutine 执行的代码在一个逻辑处理器

上并发运行的效果。

代码清单 6-2 listing01.go 的输出

```
Start Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m
n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z
Terminating Program
```

第一个 goroutine 完成所有显示需要花时间太短了，以至于在调度器切换到第二个 goroutine 之前，就完成了所有任务。这也是为什么会看到先输出了所有的大写字母，之后才输出小写字母。我们创建的两个 goroutine 一个接一个地并发运行，独立完成显示字母表的任务。

如代码清单 6-3 所示，一旦两个匿名函数创建 goroutine 来执行，`main` 中的代码会继续运行。这意味着 `main` 函数会在 goroutine 完成工作前返回。如果真的返回了，程序就会在 goroutine 有机会运行前终止。因此，在第 51 行，`main` 函数通过 `WaitGroup`，等待两个 goroutine 完成它们的工作。

代码清单 6-3 listing01.go：第 17 行到第 19 行，第 23 行到第 26 行，第 49 行到第 51 行

```
16      // wg 用来等待程序完成
17      // 计数加 2，表示要等待两个 goroutine
18      var wg sync.WaitGroup
19      wg.Add(2)

23      // 声明一个匿名函数，并创建一个 goroutine
24      go func() {
25          // 在函数退出时调用 Done 来通知 main 函数工作已经完成
26          defer wg.Done()

49      // 等待 goroutine 结束
50      fmt.Println("Waiting To Finish")
51      wg.Wait()
```

`WaitGroup` 是一个计数信号量，可以用来记录并维护运行的 goroutine。如果 `WaitGroup` 的值大于 0，`Wait` 方法就会阻塞。在第 18 行，创建了一个 `WaitGroup` 类型的变量，之后在第 19 行，将这个 `WaitGroup` 的值设置为 2，表示有两个正在运行的 goroutine。为了减小 `WaitGroup` 的值并最终释放 `main` 函数，要在第 26 和 39 行，使用 `defer` 声明在函数退出时调用 `Done` 方法。

关键字 `defer` 会修改函数调用时机，在正在执行的函数返回时才真正调用 `defer` 声明的函数。对这里的示例程序来说，我们使用关键字 `defer` 保证，每个 goroutine 一旦完成其工作就调用 `Done` 方法。

基于调度器的内部算法，一个正运行的 goroutine 在工作结束前，可以被停止并重新调度。

调度器这样做的目的是防止某个 goroutine 长时间占用逻辑处理器。当 goroutine 占用时间过长时，调度器会停止当前正运行的 goroutine，并给其他可运行的 goroutine 运行的机会。

图 6-4 从逻辑处理器的角度展示了这一场景。在第 1 步，调度器开始运行 goroutine A，而 goroutine B 在运行队列里等待调度。之后，在第 2 步，调度器交换了 goroutine A 和 goroutine B。由于 goroutine A 并没有完成工作，因此被放回到运行队列。之后，在第 3 步，goroutine B 完成了它的工作并被系统销毁。这也让 goroutine A 继续之前的工作。

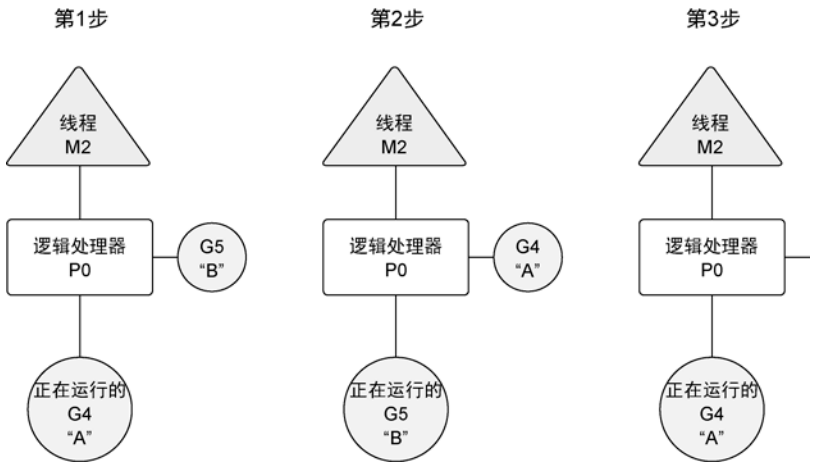

图 6-4　goroutine 在逻辑处理器的线程上进行交换

可以通过创建一个需要长时间才能完成其工作的 goroutine 来看到这个行为，如代码清单 6-4 所示。

代码清单 6-4　listing04.go

```
01 // 这个示例程序展示 goroutine 调度器是如何在单个线程上
02 // 切分时间片的
03 package main
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09 )
10
11 // wg 用来等待程序完成
12 var wg sync.WaitGroup
13
14 // main 是所有 Go 程序的入口
15 func main() {
16     // 分配一个逻辑处理器给调度器使用
17     runtime.GOMAXPROCS(1)
18
```

```
19      // 计数加 2，表示要等待两个 goroutine
20      wg.Add(2)
21
22      // 创建两个 goroutine
23      fmt.Println("Create Goroutines")
24      go printPrime("A")
25      go printPrime("B")
26
27      // 等待 goroutine 结束
28      fmt.Println("Waiting To Finish")
29      wg.Wait()
30
31      fmt.Println("Terminating Program")
32  }
33
34  // printPrime 显示 5000 以内的素数值
35  func printPrime(prefix string) {
36      // 在函数退出时调用 Done 来通知 main 函数工作已经完成
37      defer wg.Done()
38
39  next:
40      for outer := 2; outer < 5000; outer++ {
41          for inner := 2; inner < outer; inner++ {
42              if outer%inner == 0 {
43                  continue next
44              }
45          }
46          fmt.Printf("%s:%d\n", prefix, outer)
47      }
48      fmt.Println("Completed", prefix)
49  }
```

代码清单 6-4 中的程序创建了两个 goroutine，分别打印 1~5000 内的素数。查找并显示素数会消耗不少时间，这会让调度器有机会在第一个 goroutine 找到所有素数之前，切换该 goroutine 的时间片。

在第 12 行中，程序启动的时候，声明了一个 `WaitGroup` 变量，并在第 20 行将其值设置为 2。之后在第 24 行和第 25 行，在关键字 `go` 后面指定 `printPrime` 函数并创建了两个 goroutine 来执行。第一个 goroutine 使用前缀 A，第二个 goroutine 使用前缀 B。和其他函数调用一样，创建为 goroutine 的函数调用时可以传入参数。不过 goroutine 终止时无法获取函数的返回值。查看代码清单 6-5 中给出的输出时，会看到调度器在切换第一个 goroutine。

代码清单 6-5　listing04.go 的输出

```
Create Goroutines
Waiting To Finish
B:2
B:3
...
B:4583
B:4591
```