```
A:3             ** 切换 goroutine
A:5
...

A:4561
A:4567
B:4603          ** 切换 goroutine
B:4621
...
Completed B
A:4457          ** 切换 goroutine
A:4463
...
A:4993
A:4999
Completed A
Terminating Program
```

goroutine B 先显示素数。一旦 goroutine B 打印到素数 4591，调度器就会将正运行的 goroutine 切换为 goroutine A。之后 goroutine A 在线程上执行了一段时间，再次切换为 goroutine B。这次 goroutine B 完成了所有的工作。一旦 goroutine B 返回，就会看到线程再次切换到 goroutine A 并完成所有的工作。每次运行这个程序，调度器切换的时间点都会稍微有些不同。

代码清单 6-1 和代码清单 6-4 中的示例程序展示了调度器如何在一个逻辑处理器上并发运行多个 goroutine。像之前提到的，Go 标准库的 `runtime` 包里有一个名为 `GOMAXPROCS` 的函数，通过它可以指定调度器可用的逻辑处理器的数量。用这个函数，可以给每个可用的物理处理器在运行的时候分配一个逻辑处理器。代码清单 6-6 展示了这种改动，让 goroutine 并行运行。

代码清单 6-6　如何修改逻辑处理器的数量

```
import "runtime"

// 给每个可用的核心分配一个逻辑处理器
runtime.GOMAXPROCS(runtime.NumCPU())
```

包 `runtime` 提供了修改 Go 语言运行时配置参数的能力。在代码清单 6-6 里，我们使用两个 `runtime` 包的函数来修改调度器使用的逻辑处理器的数量。函数 `NumCPU` 返回可以使用的物理处理器的数量。因此，调用 `GOMAXPROCS` 函数就为每个可用的物理处理器创建一个逻辑处理器。需要强调的是，使用多个逻辑处理器并不意味着性能更好。在修改任何语言运行时配置参数的时候，都需要配合基准测试来评估程序的运行效果。

如果给调度器分配多个逻辑处理器，我们会看到之前的示例程序的输出行为会有些不同。让我们把逻辑处理器的数量改为 2，并再次运行第一个打印英文字母表的示例程序，如代码清单 6-7 所示。

代码清单 6-7　listing07.go

```
01 // 这个示例程序展示如何创建 goroutine
```

```
// 以及 goroutine 调度器的行为
package main

import (
    "fmt"
    "runtime"
    "sync"
)

// main 是所有 Go 程序的入口
func main() {
    // 分配 2 个逻辑处理器给调度器使用
    runtime.GOMAXPROCS(2)

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

    fmt.Println("Start Goroutines")

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

        // 显示字母表 3 次
        for count := 0; count < 3; count++ {
            for char := 'a'; char < 'a'+26; char++ {
                fmt.Printf("%c ", char)
            }
        }
    }()

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

        // 显示字母表 3 次
        for count := 0; count < 3; count++ {
            for char := 'A'; char < 'A'+26; char++ {
                fmt.Printf("%c ", char)
            }
        }
    }()

    // 等待 goroutine 结束
    fmt.Println("Waiting To Finish")
    wg.Wait()

    fmt.Println("\nTerminating Program")
}
```

代码清单6-7中给出的例子在第14行中通过调用GOMAXPROCS函数创建了两个逻辑处理器。这会让goroutine并行运行，输出结果如代码清单6-8所示。

代码清单 6-8　listing07.go 的输出

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

如果仔细查看代码清单6-8中的输出，会看到goroutine是并行运行的。两个goroutine几乎是同时开始运行的，大小写字母是混合在一起显示的。这是在一台8核的电脑上运行程序的输出，所以每个goroutine独自运行在自己的核上。记住，只有在有多个逻辑处理器且可以同时让每个goroutine运行在一个可用的物理处理器上的时候，goroutine才会并行运行。

现在知道了如何创建goroutine，并了解这背后发生的事情了。下面需要了解一下写并发程序时的潜在危险，以及需要注意的事情。

6.3　竞争状态

如果两个或者多个goroutine在没有互相同步的情况下，访问某个共享的资源，并试图同时读和写这个资源，就处于相互竞争的状态，这种情况被称作*竞争状态*（race candition）。竞争状态的存在是让并发程序变得复杂的地方，十分容易引起潜在问题。对一个共享资源的读和写操作必须是原子化的，换句话说，同一时刻只能有一个goroutine对共享资源进行读和写操作。代码清单6-9中给出的是包含竞争状态的示例程序。

代码清单 6-9　listing09.go

```
01 // 这个示例程序展示如何在程序里造成竞争状态
02 // 实际上不希望出现这种情况
03 package main
04
05 import (
06     "fmt"
07     "runtime"
08     "sync"
09 )
10
11 var (
12     // counter 是所有 goroutine 都要增加其值的变量
13     counter int
14
15     // wg 用来等待程序结束
```

```
16      wg sync.WaitGroup
17  )
18
19  // main 是所有 Go 程序的入口
20  func main() {
21      // 计数加 2，表示要等待两个 goroutine
22      wg.Add(2)
23
24      // 创建两个 goroutine
25      go incCounter(1)
26      go incCounter(2)
27
28      // 等待 goroutine 结束
29      wg.Wait()
30      fmt.Println("Final Counter:", counter)
31  }
32
33  // incCounter 增加包里 counter 变量的值
34  func incCounter(id int) {
35      // 在函数退出时调用 Done 来通知 main 函数工作已经完成
36      defer wg.Done()
37
38      for count := 0; count < 2; count++ {
39          // 捕获 counter 的值
40          value := counter
41
42          // 当前 goroutine 从线程退出，并放回到队列
43          runtime.Gosched()
44
45          // 增加本地 value 变量的值
46          value++
47
48          // 将该值保存回 counter
49          counter = value
50      }
51  }
```

对应的输出如代码清单 6-10 所示。

代码清单 6-10　listing09.go 的输出

```
Final Counter: 2
```

变量 `counter` 会进行 4 次读和写操作，每个 goroutine 执行两次。但是，程序终止时，`counter` 变量的值为 2。图 6-5 提供了为什么会这样的线索。

每个 goroutine 都会覆盖另一个 goroutine 的工作。这种覆盖发生在 goroutine 切换的时候。每个 goroutine 创造了一个 `counter` 变量的副本，之后就切换到另一个 goroutine。当这个 goroutine 再次运行的时候，`counter` 变量的值已经改变了，但是 goroutine 并没有更新自己的那个副本的值，而是继续使用这个副本的值，用这个值递增，并存回 `counter` 变量，结果覆盖了另一个 goroutine 完成的工作。

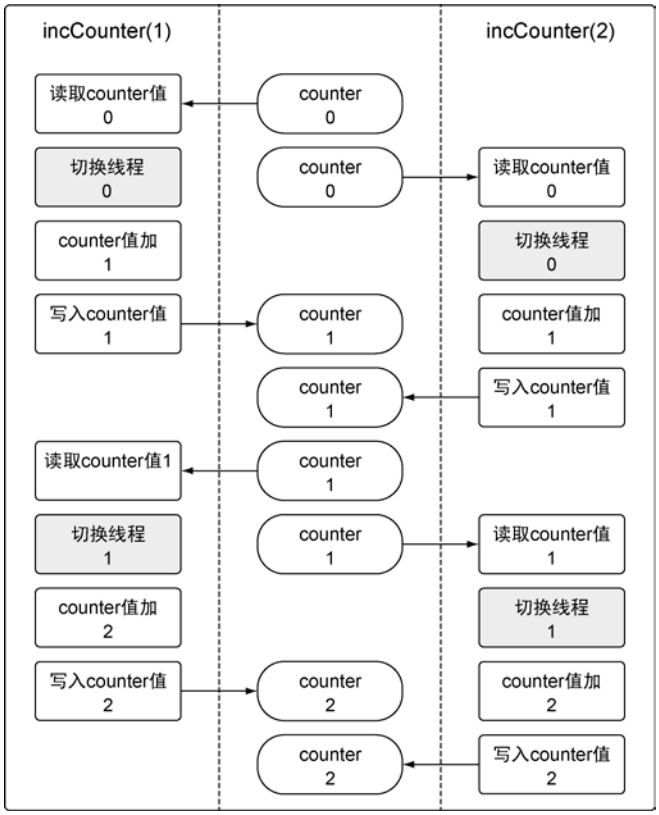

图 6-5　竞争状态下程序行为的图像表达

让我们顺着程序理解一下发生了什么。在第 25 行和第 26 行，使用 `incCounter` 函数创建了两个 goroutine。在第 34 行，`incCounter` 函数对包内变量 `counter` 进行了读和写操作，而这个变量是这个示例程序里的共享资源。每个 goroutine 都会先读出这个 `counter` 变量的值，并在第 40 行将 `counter` 变量的副本存入一个叫作 `value` 的本地变量。之后在第 46 行，`incCounter` 函数对 `value` 的副本的值加 1,最终在第 49 行将这个新值存回到 `counter` 变量。这个函数在第 43 行调用了 `runtime` 包的 `Gosched` 函数，用于将 goroutine 从当前线程退出，给其他 goroutine 运行的机会。在两次操作中间这样做的目的是强制调度器切换两个 goroutine，以便让竞争状态的效果变得更明显。

Go 语言有一个特别的工具，可以在代码里检测竞争状态。在查找这类错误的时候，这个工具非常好用，尤其是在竞争状态并不像这个例子里这么明显的时候。让我们用这个竞争检测器来检测一下我们的例子代码，如代码清单 6-11 所示。

代码清单 6-11　用竞争检测器来编译并执行 listing09 的代码

```
go build -race   // 用竞争检测器标志来编译程序
```

```
./example        // 运行程序

==================
WARNING: DATA RACE
Write by goroutine 5:

  main.incCounter()
      /example/main.go:49 +0x96

Previous read by goroutine 6:
  main.incCounter()
      /example/main.go:40 +0x66

Goroutine 5 (running) created at:
  main.main()
      /example/main.go:25 +0x5c

Goroutine 6 (running) created at:
  main.main()
      /example/main.go:26 +0x73
==================
Final Counter: 2
Found 1 data race(s)
```

代码清单 6-11 中的竞争检测器指出这个例子里面代码清单 6-12 所示的 4 行代码有问题。

代码清单 6-12　竞争检测器指出的代码

```
Line 49: counter = value
Line 40: value := counter
Line 25: go incCounter(1)
Line 26: go incCounter(2)
```

代码清单 6-12 展示了竞争检测器查到的哪个 goroutine 引发了数据竞争，以及哪两行代码有冲突。毫不奇怪，这几行代码分别是对 `counter` 变量的读和写操作。

一种修正代码、消除竞争状态的办法是，使用 Go 语言提供的锁机制，来锁住共享资源，从而保证 goroutine 的同步状态。

6.4　锁住共享资源

Go 语言提供了传统的同步 goroutine 的机制，就是对共享资源加锁。如果需要顺序访问一个整型变量或者一段代码，`atomic` 和 `sync` 包里的函数提供了很好的解决方案。下面我们了解一下 `atomic` 包里的几个函数以及 `sync` 包里的 `mutex` 类型。

6.4.1　原子函数

原子函数能够以很底层的加锁机制来同步访问整型变量和指针。我们可以用原子函数来修正代码清单 6-9 中创建的竞争状态，如代码清单 6-13 所示。

代码清单 6-13　listing13.go

```
// 这个示例程序展示如何使用 atomic 包来提供
// 对数值类型的安全访问
package main

import (
    "fmt"
    "runtime"
    "sync"
    "sync/atomic"
)

var (
    // counter 是所有 goroutine 都要增加其值的变量
    counter int64

    // wg 用来等待程序结束
    wg sync.WaitGroup
)

// main 是所有 Go 程序的入口
func main() {
    // 计数加 2，表示要等待两个 goroutine
    wg.Add(2)

    // 创建两个 goroutine
    go incCounter(1)
    go incCounter(2)

    // 等待 goroutine 结束
    wg.Wait()

    // 显示最终的值
    fmt.Println("Final Counter:", counter)
}

// incCounter 增加包里 counter 变量的值
func incCounter(id int) {
    // 在函数退出时调用 Done 来通知 main 函数工作已经完成
    defer wg.Done()

    for count := 0; count < 2; count++ {
        // 安全地对 counter 加 1
        atomic.AddInt64(&counter, 1)

        // 当前 goroutine 从线程退出，并放回到队列
        runtime.Gosched()
    }
}
```

对应的输出如代码清单 6-14 所示。

代码清单 6-14　listing13.go 的输出

```
Final Counter: 4
```

现在，程序的第 43 行使用了 `atmoic` 包的 `AddInt64` 函数。这个函数会同步整型值的加法，方法是强制同一时刻只能有一个 goroutine 运行并完成这个加法操作。当 goroutine 试图去调用任何原子函数时，这些 goroutine 都会自动根据所引用的变量做同步处理。现在我们得到了正确的值 4。

另外两个有用的原子函数是 `LoadInt64` 和 `StoreInt64`。这两个函数提供了一种安全地读和写一个整型值的方式。代码清单 6-15 中的示例程序使用 `LoadInt64` 和 `StoreInt64` 来创建一个同步标志，这个标志可以向程序里多个 goroutine 通知某个特殊状态。

代码清单 6-15　listing15.go

```
01 // 这个示例程序展示如何使用 atomic 包里的
02 // Store 和 Load 类函数来提供对数值类型
03 // 的安全访问
04 package main
05
06 import (
07     "fmt"
08     "sync"
09     "sync/atomic"
10     "time"
11 )
12
13 var (
14     // shutdown 是通知正在执行的 goroutine 停止工作的标志
15     shutdown int64
16
17     // wg 用来等待程序结束
18     wg sync.WaitGroup
19 )
20
21 // main 是所有 Go 程序的入口
22 func main() {
23     // 计数加 2，表示要等待两个 goroutine
24     wg.Add(2)
25
26     // 创建两个 goroutine
27     go doWork("A")
28     go doWork("B")
29
30     // 给定 goroutine 执行的时间
31     time.Sleep(1 * time.Second)
32
33     // 该停止工作了，安全地设置 shutdown 标志
34     fmt.Println("Shutdown Now")
35     atomic.StoreInt64(&shutdown, 1)
36
```

```
37      // 等待 goroutine 结束
38      wg.Wait()
39  }
40
41  // doWork 用来模拟执行工作的 goroutine，
42  // 检测之前的 shutdown 标志来决定是否提前终止
43  func doWork(name string) {
44      // 在函数退出时调用 Done 来通知 main 函数工作已经完成
45      defer wg.Done()
46
47      for {
48          fmt.Printf("Doing %s Work\n", name)
49          time.Sleep(250 * time.Millisecond)
50
51          // 要停止工作了吗？
52          if atomic.LoadInt64(&shutdown) == 1 {
53              fmt.Printf("Shutting %s Down\n", name)
54              break
55          }
56      }
57  }
```

在这个例子中，启动了两个 goroutine，并完成一些工作。在各自循环的每次迭代之后，在第 52 行中 goroutine 会使用 `LoadInt64` 来检查 `shutdown` 变量的值。这个函数会安全地返回 `shutdown` 变量的一个副本。如果这个副本的值为 1，goroutine 就会跳出循环并终止。

在第 35 行中，`main` 函数使用 `StoreInt64` 函数来安全地修改 `shutdown` 变量的值。如果哪个 `doWork` goroutine 试图在 `main` 函数调用 `StoreInt64` 的同时调用 `LoadInt64` 函数，那么原子函数会将这些调用互相同步，保证这些操作都是安全的，不会进入竞争状态。

6.4.2 互斥锁

另一种同步访问共享资源的方式是使用互斥锁（`mutex`）。互斥锁这个名字来自互斥（mutual exclusion）的概念。互斥锁用于在代码上创建一个临界区，保证同一时间只有一个 goroutine 可以执行这个临界区代码。我们还可以用互斥锁来修正代码清单 6-9 中创建的竞争状态，如代码清单 6-16 所示。

代码清单 6-16　listing16.go

```
01  // 这个示例程序展示如何使用互斥锁来
02  // 定义一段需要同步访问的代码临界区
03  // 资源的同步访问
04  package main
05
06  import (
07      "fmt"
08      "runtime"
09      "sync"
10  )
```

```

var (
    // counter 是所有 goroutine 都要增加其值的变量
    counter int

    // wg 用来等待程序结束
    wg sync.WaitGroup

    // mutex 用来定义一段代码临界区
    mutex sync.Mutex
)

// main 是所有 Go 程序的入口
func main() {
    // 计数加 2，表示要等待两个 goroutine
    wg.Add(2)

    // 创建两个 goroutine
    go incCounter(1)
    go incCounter(2)

    // 等待 goroutine 结束
    wg.Wait()
    fmt.Printf("Final Counter: %d\\n", counter)
}

// incCounter 使用互斥锁来同步并保证安全访问，
// 增加包里 counter 变量的值
func incCounter(id int) {
    // 在函数退出时调用 Done 来通知 main 函数工作已经完成
    defer wg.Done()

    for count := 0; count < 2; count++ {
        // 同一时刻只允许一个 goroutine 进入
        // 这个临界区
        mutex.Lock()
        {
            // 捕获 counter 的值
            value := counter

            // 当前 goroutine 从线程退出，并放回到队列
            runtime.Gosched()

            // 增加本地 value 变量的值
            value++

            // 将该值保存回 counter
            counter = value
        }
        mutex.Unlock()
        // 释放锁，允许其他正在等待的 goroutine
        // 进入临界区
    }
}
```

对 counter 变量的操作在第 46 行和第 60 行的 Lock()和 Unlock()函数调用定义的临界区里被保护起来。使用大括号只是为了让临界区看起来更清晰，并不是必需的。同一时刻只有一个 goroutine 可以进入临界区。之后，直到调用 Unlock()函数之后，其他 goroutine 才能进入临界区。当第 52 行强制将当前 goroutine 退出当前线程后，调度器会再次分配这个 goroutine 继续运行。当程序结束时，我们得到正确的值 4，竞争状态不再存在。

6.5 通道

原子函数和互斥锁都能工作，但是依靠它们都不会让编写并发程序变得更简单，更不容易出错，或者更有趣。在 Go 语言里，你不仅可以使用原子函数和互斥锁来保证对共享资源的安全访问以及消除竞争状态，还可以使用通道，通过发送和接收需要共享的资源，在 goroutine 之间做同步。

当一个资源需要在 goroutine 之间共享时，通道在 goroutine 之间架起了一个管道，并提供了确保同步交换数据的机制。声明通道时，需要指定将要被共享的数据的类型。可以通过通道共享内置类型、命名类型、结构类型和引用类型的值或者指针。

在 Go 语言中需要使用内置函数 make 来创建一个通道，如代码清单 6-17 所示。

代码清单 6-17　使用 make 创建通道

```
// 无缓冲的整型通道
unbuffered := make(chan int)

// 有缓冲的字符串通道
buffered := make(chan string, 10)
```

在代码清单 6-17 中，可以看到使用内置函数 make 创建了两个通道，一个无缓冲的通道，一个有缓冲的通道。make 的第一个参数需要是关键字 chan，之后跟着允许通道交换的数据的类型。如果创建的是一个有缓冲的通道，之后还需要在第二个参数指定这个通道的缓冲区的大小。

向通道发送值或者指针需要用到<-操作符，如代码清单 6-18 所示。

代码清单 6-18　向通道发送值

```
// 有缓冲的字符串通道
buffered := make(chan string, 10)

// 通过通道发送一个字符串
buffered <- "Gopher"
```

在代码清单 6-18 里，我们创建了一个有缓冲的通道，数据类型是字符串，包含一个 10 个值的缓冲区。之后我们通过通道发送字符串"Gopher"。为了让另一个 goroutine 可以从该通道里接收到这个字符串，我们依旧使用<-操作符，但这次是一元运算符，如代码清单 6-19 所示。

代码清单 6-19　从通道里接收值

```
// 从通道接收一个字符串
value := <-buffered
```

当从通道里接收一个值或者指针时，<-运算符在要操作的通道变量的左侧，如代码清单 6-19 所示。

通道是否带有缓冲，其行为会有一些不同。理解这个差异对决定到底应该使用还是不使用缓冲很有帮助。下面我们分别介绍一下这两种类型。

6.5.1　无缓冲的通道

无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何值的通道。这种类型的通道要求发送 goroutine 和接收 goroutine 同时准备好，才能完成发送和接收操作。如果两个 goroutine 没有同时准备好，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。这种对通道进行发送和接收的交互行为本身就是同步的。其中任意一个操作都无法离开另一个操作单独存在。

在图 6-6 里，可以看到一个例子，展示两个 goroutine 如何利用无缓冲的通道来共享一个值。

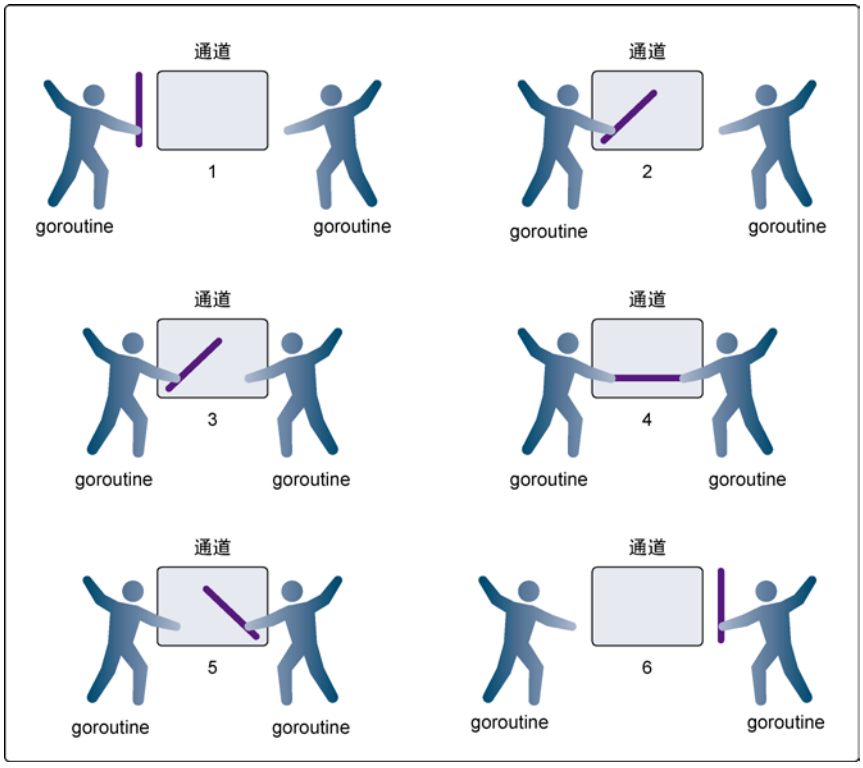

图 6-6　使用无缓冲的通道在 goroutine 之间同步

在第 1 步，两个 goroutine 都到达通道，但哪个都没有开始执行发送或者接收。在第 2 步，左侧的 goroutine 将它的手伸进了通道，这模拟了向通道发送数据的行为。这时，这个 goroutine 会在通道中被锁住，直到交换完成。在第 3 步，右侧的 goroutine 将它的手放入通道，这模拟了从通道里接收数据。这个 goroutine 一样也会在通道中被锁住，直到交换完成。在第 4 步和第 5 步，进行交换，并最终，在第 6 步，两个 goroutine 都将它们的手从通道里拿出来，这模拟了被锁住的 goroutine 得到释放。两个 goroutine 现在都可以去做别的事情了。

为了讲得更清楚，让我们来看两个完整的例子。这两个例子都会使用无缓冲的通道在两个 goroutine 之间同步交换数据。

在网球比赛中，两位选手会把球在两个人之间来回传递。选手总是处在以下两种状态之一：要么在等待接球，要么将球打向对方。可以使用两个 goroutine 来模拟网球比赛，并使用无缓冲的通道来模拟球的来回，如代码清单 6-20 所示。

代码清单 6-20　listing20.go

```
// 这个示例程序展示如何用无缓冲的通道来模拟
// 2 个 goroutine 间的网球比赛
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

// wg 用来等待程序结束
var wg sync.WaitGroup

func init() {
    rand.Seed(time.Now().UnixNano())
}

// main 是所有 Go 程序的入口
func main() {
    // 创建一个无缓冲的通道
    court := make(chan int)

    // 计数加 2，表示要等待两个 goroutine
    wg.Add(2)

    // 启动两个选手
    go player("Nadal", court)
    go player("Djokovic", court)

    // 发球
    court <- 1

    // 等待游戏结束
```

```
35      wg.Wait()
36  }
37
38  // player 模拟一个选手在打网球
39  func player(name string, court chan int) {
40      // 在函数退出时调用 Done 来通知 main 函数工作已经完成
41      defer wg.Done()
42
43      for {
44          // 等待球被击打过来
45          ball, ok := <-court
46          if !ok {
47              // 如果通道被关闭，我们就赢了
48              fmt.Printf("Player %s Won\n", name)
49              return
50          }
51
52          // 选随机数，然后用这个数来判断我们是否丢球
53          n := rand.Intn(100)
54          if n%13 == 0 {
55              fmt.Printf("Player %s Missed\n", name)
56
57              // 关闭通道，表示我们输了
58              close(court)
59              return
60          }
61
62          // 显示击球数，并将击球数加 1
63          fmt.Printf("Player %s Hit %d\n", name, ball)
64          ball++
65
66          // 将球打向对手
67          court <- ball
68      }
69  }
```

运行这个程序会得到代码清单 6-21 所示的输出。

代码清单 6-21　listing20.go 的输出

```
Player Nadal Hit 1
Player Djokovic Hit 2
Player Nadal Hit 3
Player Djokovic Missed
Player Nadal Won
```

在 `main` 函数的第 22 行，创建了一个 `int` 类型的无缓冲的通道，让两个 goroutine 在击球时能够互相同步。之后在第 28 行和第 29 行，创建了参与比赛的两个 goroutine。在这个时候，两个 goroutine 都阻塞住等待击球。在第 32 行，将球发到通道里，程序开始执行这个比赛，直到某个 goroutine 输掉比赛。

在 `player` 函数里，在第 43 行可以找到一个无限循环的 `for` 语句。在这个循环里，是玩游戏的过程。在第 45 行，goroutine 从通道接收数据，用来表示等待接球。这个接收动作会锁住

goroutine，直到有数据发送到通道里。通道的接收动作返回时，第 46 行会检测 `ok` 标志是否为 `false`。如果这个值是 `false`，表示通道已经被关闭，游戏结束。在第 53 行到第 60 行，会产生一个随机数，用来决定 goroutine 是否击中了球。如果击中了球，在第 64 行 `ball` 的值会递增 1，并在第 67 行，将 `ball` 作为球重新放入通道，发送给另一位选手。在这个时刻，两个 goroutine 都会被锁住，直到交换完成。最终，某个 goroutine 没有打中球，在第 58 行关闭通道。之后两个 goroutine 都会返回，通过 `defer` 声明的 `Done` 会被执行，程序终止。

另一个例子，用不同的模式，使用无缓冲的通道，在 goroutine 之间同步数据，来模拟接力比赛。在接力比赛里，4 个跑步者围绕赛道轮流跑（如代码清单 6-22 所示）。第二个、第三个和第四个跑步者要接到前一位跑步者的接力棒后才能起跑。比赛中最重要的部分是要传递接力棒，要求同步传递。在同步接力棒的时候，参与接力的两个跑步者必须在同一时刻准备好交接。

代码清单 6-22　listing22.go

```
01 // 这个示例程序展示如何用无缓冲的通道来模拟
02 // 4 个 goroutine 间的接力比赛
03 package main
04
05 import (
06     "fmt"
07     "sync"
08     "time"
09 )
10
11 // wg 用来等待程序结束
12 var wg sync.WaitGroup
13
14 // main 是所有 Go 程序的入口
15 func main() {
16     // 创建一个无缓冲的通道
17     baton := make(chan int)
18
19     // 为最后一位跑步者将计数加 1
20     wg.Add(1)
21
22     // 第一位跑步者持有接力棒
23     go Runner(baton)
24
25     // 开始比赛
26     baton <- 1
27
28     // 等待比赛结束
29     wg.Wait()
30 }
31
32 // Runner 模拟接力比赛中的一位跑步者
33 func Runner(baton chan int) {
34     var newRunner int
35
```

```
36      // 等待接力棒
37      runner := <-baton
38
39      // 开始绕着跑道跑步
40      fmt.Printf("Runner %d Running With Baton\n", runner)
41
42      // 创建下一位跑步者
43      if runner != 4 {
44          newRunner = runner + 1
45          fmt.Printf("Runner %d To The Line\n", newRunner)
46          go Runner(baton)
47      }
48
49      // 围绕跑道跑
50      time.Sleep(100 * time.Millisecond)
51
52      // 比赛结束了吗?
53      if runner == 4 {
54          fmt.Printf("Runner %d Finished, Race Over\n", runner)
55          wg.Done()
56          return
57      }
58
59      // 将接力棒交给下一位跑步者
60      fmt.Printf("Runner %d Exchange With Runner %d\n",
61          runner,
62          newRunner)
63
64      baton <- newRunner
65  }
```

运行这个程序会得到代码清单 6-23 所示的输出。

代码清单 6-23 listing22.go 的输出

```
Runner 1 Running With Baton
Runner 1 To The Line
Runner 1 Exchange With Runner 2
Runner 2 Running With Baton
Runner 2 To The Line
Runner 2 Exchange With Runner 3
Runner 3 Running With Baton
Runner 3 To The Line
Runner 3 Exchange With Runner 4
Runner 4 Running With Baton
Runner 4 Finished, Race Over
```

在 `main` 函数的第 17 行，创建了一个无缓冲的 `int` 类型的通道 `baton`，用来同步传递接力棒。在第 20 行，我们给 `WaitGroup` 加 1，这样 `main` 函数就会等最后一位跑步者跑步结束。在第 23 行创建了一个 goroutine，用来表示第一位跑步者来到跑道。之后在第 26 行，将接力棒交给这个跑步者，比赛开始。最终，在第 29 行，`main` 函数阻塞在 `WaitGroup`，等候最后一位跑步者完成比赛。

在 `Runner` goroutine 里，可以看到接力棒 `baton` 是如何在跑步者之间传递的。在第 37 行，goroutine 对 `baton` 通道执行接收操作，表示等候接力棒。一旦接力棒传了进来，在第 46 行就会

创建一位新跑步者，准备接力下一棒，直到 goroutine 是第四个跑步者。在第 50 行，跑步者围绕跑道跑 100 ms。在第 55 行，如果第四个跑步者完成了比赛，就调用 `Done`，将 `WaitGroup` 减 1，之后 goroutine 返回。如果这个 goroutine 不是第四个跑步者，那么在第 64 行，接力棒会交到下一个已经在等待的跑步者手上。在这个时候，goroutine 会被锁住，直到交接完成。

在这两个例子里，我们使用无缓冲的通道同步 goroutine，模拟了网球和接力赛。代码的流程与这两个活动在真实世界中的流程完全一样，这样的代码很容易读懂。现在知道了无缓冲的通道是如何工作的，接下来我们会学习有缓冲的通道的工作方法。

6.5.2 有缓冲的通道

有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个值的通道。这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：无缓冲的通道保证进行发送和接收的 goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。

在图 6-7 中可以看到两个 goroutine 分别向有缓冲的通道里增加一个值和从有缓冲的通道里移除一个值。在第 1 步，右侧的 goroutine 正在从通道接收一个值。在第 2 步，右侧的这个 goroutine 独立完成了接收值的动作，而左侧的 goroutine 正在发送一个新值到通道里。在第 3 步，左侧的 goroutine 还在向通道发送新值，而右侧的 goroutine 正在从通道接收另外一个值。这个步骤里的两个操作既不是同步的，也不会互相阻塞。最后，在第 4 步，所有的发送和接收都完成，而通道里还有几个值，也有一些空间可以存更多的值。

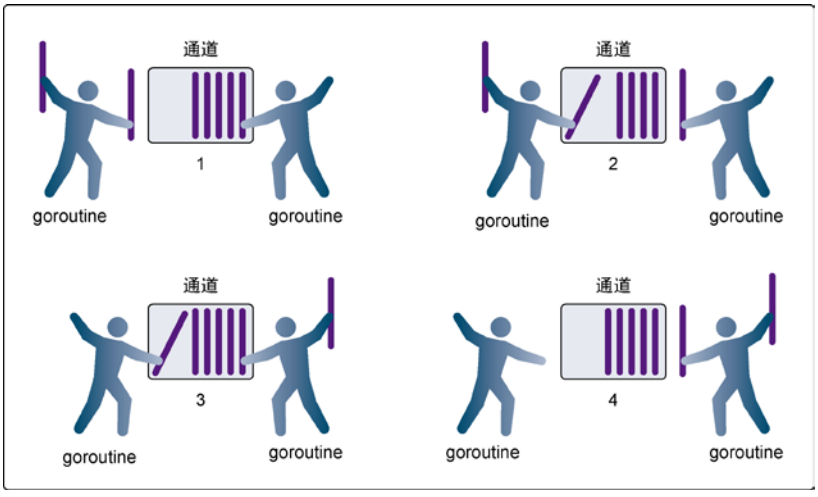

图 6-7　使用有缓冲的通道在 goroutine 之间同步数据

让我们看一个使用有缓冲的通道的例子，这个例子管理一组 goroutine 来接收并完成工作。有缓冲的通道提供了一种清晰而直观的方式来实现这个功能，如代码清单 6-24 所示。

代码清单 6-24　listing24.go

```
01 // 这个示例程序展示如何使用
02 // 有缓冲的通道和固定数目的
03 // goroutine 来处理一堆工作
04 package main
05
06 import (
07     "fmt"
08     "math/rand"
09     "sync"
10     "time"
11 )
12
13 const (
14     numberGoroutines = 4  // 要使用的 goroutine 的数量
15     taskLoad         = 10  // 要处理的工作的数量
16 )
17
18 // wg 用来等待程序完成
19 var wg sync.WaitGroup
20
21 // init 初始化包，Go 语言运行时会在其他代码执行之前
22 // 优先执行这个函数
23 func init() {
24     // 初始化随机数种子
25     rand.Seed(time.Now().Unix())
26 }
27
28 // main 是所有 Go 程序的入口
29 func main() {
30     // 创建一个有缓冲的通道来管理工作
31     tasks := make(chan string, taskLoad)
32
33     // 启动 goroutine 来处理工作
34     wg.Add(numberGoroutines)
35     for gr := 1; gr <= numberGoroutines; gr++ {
36         go worker(tasks, gr)
37     }
38
39     // 增加一组要完成的工作
40     for post := 1; post <= taskLoad; post++ {
41         tasks <- fmt.Sprintf("Task : %d", post)
42     }
43
44     // 当所有工作都处理完时关闭通道
45     // 以便所有 goroutine 退出
46     close(tasks)
47
48     // 等待所有工作完成
49     wg.Wait()
```

```
50 }
51
52 // worker 作为 goroutine 启动来处理
53 // 从有缓冲的通道传入的工作
54 func worker(tasks chan string, worker int) {
55     // 通知函数已经返回
56     defer wg.Done()
57
58     for {
59         // 等待分配工作
60         task, ok := <-tasks
61         if !ok {
62             // 这意味着通道已经空了，并且已被关闭
63             fmt.Printf("Worker: %d : Shutting Down\n", worker)
64             return
65         }
66
67         // 显示我们开始工作了
68         fmt.Printf("Worker: %d : Started %s\n", worker, task)
69
70         // 随机等一段时间来模拟工作
71         sleep := rand.Int63n(100)
72         time.Sleep(time.Duration(sleep) * time.Millisecond)
73
74         // 显示我们完成了工作
75         fmt.Printf("Worker: %d : Completed %s\n", worker, task)
76     }
77 }
```

运行这个程序会得到代码清单 6-25 所示的输出。

代码清单 6-25　listing24.go 的输出

```
Worker: 1 : Started Task : 1
Worker: 2 : Started Task : 2
Worker: 3 : Started Task : 3
Worker: 4 : Started Task : 4
Worker: 1 : Completed Task : 1
Worker: 1 : Started Task : 5
Worker: 4 : Completed Task : 4
Worker: 4 : Started Task : 6
Worker: 1 : Completed Task : 5
Worker: 1 : Started Task : 7
Worker: 2 : Completed Task : 2
Worker: 2 : Started Task : 8
Worker: 3 : Completed Task : 3
Worker: 3 : Started Task : 9
Worker: 1 : Completed Task : 7
Worker: 1 : Started Task : 10
Worker: 4 : Completed Task : 6
Worker: 4 : Shutting Down
Worker: 3 : Completed Task : 9
Worker: 3 : Shutting Down
Worker: 2 : Completed Task : 8
```

```
Worker: 2 : Shutting Down
Worker: 1 : Completed Task : 10
Worker: 1 : Shutting Down
```

由于程序和 Go 语言的调度器带有随机成分，这个程序每次执行得到的输出会不一样。不过，通过有缓冲的通道，使用所有 4 个 goroutine 来完成工作，这个流程不会变。从输出可以看到每个 goroutine 是如何接收从通道里分发的工作。

在 `main` 函数的第 31 行，创建了一个 `string` 类型的有缓冲的通道，缓冲的容量是 10。在第 34 行，给 `WaitGroup` 赋值为 4，代表创建了 4 个工作 goroutine。之后在第 35 行到第 37 行，创建了 4 个 goroutine，并传入用来接收工作的通道。在第 40 行到第 42 行，将 10 个字符串发送到通道，模拟发给 goroutine 的工作。一旦最后一个字符串发送到通道，通道就会在第 46 行关闭，而 `main` 函数就会在第 49 行等待所有工作的完成。

第 46 行中关闭通道的代码非常重要。当通道关闭后，goroutine 依旧可以从通道接收数据，但是不能再向通道里发送数据。能够从已经关闭的通道接收数据这一点非常重要，因为这允许通道关闭后依旧能取出其中缓冲的全部值，而不会有数据丢失。从一个已经关闭且没有数据的通道里获取数据，总会立刻返回，并返回一个通道类型的零值。如果在获取通道时还加入了可选的标志，就能得到通道的状态信息。

在 `worker` 函数里，可以在第 58 行看到一个无限的 `for` 循环。在这个循环里，会处理所有接收到的工作。每个 goroutine 都会在第 60 行阻塞，等待从通道里接收新的工作。一旦接收到返回，就会检查 `ok` 标志，看通道是否已经清空而且关闭。如果 `ok` 的值是 `false`，goroutine 就会终止，并调用第 56 行通过 `defer` 声明的 `Done` 函数，通知 `main` 有工作结束。

如果 `ok` 标志是 `true`，表示接收到的值是有效的。第 71 行和第 72 行模拟了处理的工作。一旦工作完成，goroutine 会再次阻塞在第 60 行从通道获取数据的语句。一旦通道被关闭，这个从通道获取数据的语句会立刻返回，goroutine 也会终止自己。

有缓冲的通道和无缓冲的通道的例子很好地展示了如何编写使用通道的代码。在下一章，我们会介绍真实世界里的一些可能会在工程里用到的并发模式。

6.6 小结

- 并发是指 goroutine 运行的时候是相互独立的。
- 使用关键字 `go` 创建 goroutine 来运行函数。
- goroutine 在逻辑处理器上执行，而逻辑处理器具有独立的系统线程和运行队列。
- 竞争状态是指两个或者多个 goroutine 试图访问同一个资源。
- 原子函数和互斥锁提供了一种防止出现竞争状态的办法。
- 通道提供了一种在两个 goroutine 之间共享数据的简单方法。
- 无缓冲的通道保证同时交换数据，而有缓冲的通道不做这种保证。

第 7 章　并发模式

本章主要内容

- 控制程序的生命周期
- 管理可复用的资源池
- 创建可以处理任务的 goroutine 池

在第 6 章中，我们学习了什么是并发，通道是如何工作的，并学习了可以实际工作的并发代码。本章将通过学习更多代码来扩展这些知识。我们会学习 3 个可以在实际工程里使用的包，这 3 个包分别实现了不同的并发模式。每个包从一个实用的视角来讲解如何使用并发和通道。我们会学习如何用这个包简化并发程序的编写，以及为什么能简化的原因。

7.1　runner

`runner` 包用于展示如何使用通道来监视程序的执行时间，如果程序运行时间太长，也可以用 `runner` 包来终止程序。当开发需要调度后台处理任务的程序的时候，这种模式会很有用。这个程序可能会作为 cron 作业执行，或者在基于定时任务的云环境（如 iron.io）里执行。

让我们来看一下 `runner` 包里的 runner.go 代码文件，如代码清单 7-1 所示。

代码清单 7-1　**runner**/runner.go

```
01 // Gabriel Aszalos 协助完成了这个示例
02 // runner 包管理处理任务的运行和生命周期
03 package runner
04
05 import (
06     "errors"
07     "os"
08     "os/signal"
09     "time"
10 )
11
```

```
12 // Runner 在给定的超时时间内执行一组任务，
13 // 并且在操作系统发送中断信号时结束这些任务
14 type Runner struct {
15     // interrupt 通道报告从操作系统
16     // 发送的信号
17     interrupt chan os.Signal
18
19     // complete 通道报告处理任务已经完成
20     complete chan error
21
22     // timeout 报告处理任务已经超时
23     timeout <-chan time.Time
24
25     // tasks 持有一组以索引顺序依次执行的
26     // 函数
27     tasks []func(int)
28 }
29
30 // ErrTimeout 会在任务执行超时时返回
31 var ErrTimeout = errors.New("received timeout")
32
33 // ErrInterrupt 会在接收到操作系统的事件时返回
34 var ErrInterrupt = errors.New("received interrupt")
35
36 // New 返回一个新的准备使用的 Runner
37 func New(d time.Duration) *Runner {
38     return &Runner{
39         interrupt: make(chan os.Signal, 1),
40         complete:  make(chan error),
41         timeout:   time.After(d),
42     }
43 }
44
45 // Add 将一个任务附加到 Runner 上。这个任务是一个
46 // 接收一个 int 类型的 ID 作为参数的函数
47 func (r *Runner) Add(tasks ...func(int)) {
48     r.tasks = append(r.tasks, tasks...)
49 }
50
51 // Start 执行所有任务，并监视通道事件
52 func (r *Runner) Start() error {
53     // 我们希望接收所有中断信号
54     signal.Notify(r.interrupt, os.Interrupt)
55
56     // 用不同的 goroutine 执行不同的任务
57     go func() {
58         r.complete <- r.run()
59     }()
60
61     select {
62     // 当任务处理完成时发出的信号
63     case err := <-r.complete:
64         return err
```

```
 65
 66      // 当任务处理程序运行超时时发出的信号
 67      case <-r.timeout:
 68          return ErrTimeout
 69      }
 70 }
 71
 72 // run 执行每一个已注册的任务
 73 func (r *Runner) run() error {
 74      for id, task := range r.tasks {
 75          // 检测操作系统的中断信号
 76          if r.gotInterrupt() {
 77              return ErrInterrupt
 78          }
 79
 80          // 执行已注册的任务
 81          task(id)
 82      }
 83
 84      return nil
 85 }
 86
 87 // gotInterrupt 验证是否接收到了中断信号
 88 func (r *Runner) gotInterrupt() bool {
 89      select {
 90      // 当中断事件被触发时发出的信号
 91      case <-r.interrupt:
 92          // 停止接收后续的任何信号
 93          signal.Stop(r.interrupt)
 95          return true
 96
 97      // 继续正常运行
 98      default:
 99          return false
100      }
101 }
```

代码清单 7-1 中的程序展示了依据调度运行的无人值守的面向任务的程序，及其所使用的并发模式。在设计上，可支持以下终止点：

- 程序可以在分配的时间内完成工作，正常终止；
- 程序没有及时完成工作，“自杀”；
- 接收到操作系统发送的中断事件，程序立刻试图清理状态并停止工作。

让我们走查一遍代码，看看每个终止点是如何实现的，如代码清单 7-2 所示。

代码清单 7-2　runner/runner.go：第 12 行到第 28 行

```
12 // Runner 在给定的超时时间内执行一组任务，
13 // 并且在操作系统发送中断信号时结束这些任务
14 type Runner struct {
15      // interrupt 通道报告从操作系统
16      // 发送的信号
```

```
17      interrupt chan os.Signal
18
19      // complete 通道报告处理任务已经完成
20      complete chan error
21
22      // timeout 报告处理任务已经超时
23      timeout <-chan time.Time
24
25      // tasks 持有一组以索引顺序依次执行的
26      // 函数
27      tasks []func(int)
28  }
```

代码清单 7-2 从第 14 行声明 `Runner` 结构开始。这个类型声明了 3 个通道，用来辅助管理程序的生命周期，以及用来表示顺序执行的不同任务的函数切片。

第 17 行的 `interrupt` 通道收发 `os.Signal` 接口类型的值，用来从主机操作系统接收中断事件。`os.Signal` 接口的声明如代码清单 7-3 所示。

代码清单 7-3　golang.org/pkg/os/#Signal

```
// Signal 用来描述操作系统发送的信号。其底层实现通常会
// 依赖操作系统的具体实现：在 UNIX 系统上是
// syscall.Signal
type Signal interface {
    String() string
    Signal()//用来区分其他 Stringer
}
```

代码清单 7-3 展示了 `os.Signal` 接口的声明。这个接口抽象了不同操作系统上捕获和报告信号事件的具体实现。

第二个字段被命名为 `complete`，是一个收发 `error` 接口类型值的通道，如代码清单 7-4 所示。

代码清单 7-4　**runner**/runner.go：第 19 行到第 20 行

```
19      // complete 通道报告处理任务已经完成
20      complete chan error
```

这个通道被命名为 `complete`，因为它被执行任务的 goroutine 用来发送任务已经完成的信号。如果执行任务时发生了错误，会通过这个通道发回一个 `error` 接口类型的值。如果没有发生错误，会通过这个通道发回一个 `nil` 值作为 `error` 接口值。

第三个字段被命名为 `timeout`，接收 `time.Time` 值，如代码清单 7-5 所示。

代码清单 7-5　**runner**/runner.go：第 22 行到第 23 行

```
22      // timeout 报告处理任务已经超时
23      timeout <-chan time.Time
```

这个通道用来管理执行任务的时间。如果从这个通道接收到一个 `time.Time` 的值，这个程