序就会试图清理状态并停止工作。

最后一个字段被命名为 tasks，是一个函数值的切片，如代码清单 7-6 所示。

代码清单 7-6 **runner**/runner.go：第 25 行到第 27 行

```
25      // tasks 持有一组以索引顺序依次执行的
26      // 函数
27      tasks []func(int)
```

这些函数值代表一个接一个顺序执行的函数。会有一个与 main 函数分离的 goroutine 来执行这些函数。

现在已经声明了 Runner 类型，接下来看一下两个 error 接口变量，这两个变量分别代表不同的错误值，如代码清单 7-7 所示。

代码清单 7-7 **runner**/runner.go：第 30 行到第 34 行

```
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")
```

第一个 error 接口变量名为 ErrTimeout。这个错误值会在收到超时事件时，由 Start 方法返回。第二个 error 接口变量名为 ErrInterrupt。这个错误值会在收到操作系统的中断事件时，由 Start 方法返回。

现在我们来看一下用户如何创建一个 Runner 类型的值，如代码清单 7-8 所示。

代码清单 7-8 **runner**/runner.go：第 36 行到第 43 行

```
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 }
```

代码清单 7-8 展示了名为 New 的工厂函数。这个函数接收一个 time.Duration 类型的值，并返回 Runner 类型的指针。这个函数会创建一个 Runner 类型的值，并初始化每个通道字段。因为 task 字段的零值是 nil，已经满足初始化的要求，所以没有被明确初始化。每个通道字段都有独立的初始化过程，让我们探究一下每个字段的初始化细节。

通道 interrupt 被初始化为缓冲区容量为 1 的通道。这可以保证通道至少能接收一个来自语言运行时的 os.Signal 值，确保语言运行时发送这个事件的时候不会被阻塞。如果 goroutine 没有准备好接收这个值，这个值就会被丢弃。例如，如果用户反复敲 Ctrl+C 组合键，程序只会

在这个通道的缓冲区可用的时候接收事件，其余的所有事件都会被丢弃。

通道 complete 被初始化为无缓冲的通道。当执行任务的 goroutine 完成时，会向这个通道发送一个 error 类型的值或者 nil 值。之后就会等待 main 函数接收这个值。一旦 main 接收了这个 error 值，goroutine 就可以安全地终止了。

最后一个通道 timeout 是用 time 包的 After 函数初始化的。After 函数返回一个 time.Time 类型的通道。语言运行时会在指定的 duration 时间到期之后，向这个通道发送一个 time.Time 的值。

现在知道了如何创建并初始化一个 Runner 值，我们再来看一下与 Runner 类型关联的方法。第一个方法 Add 用来增加一个要执行的任务函数，如代码清单 7-9 所示。

代码清单 7-9　**runner**/runner.go：第 45 行到第 49 行

```
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 }
```

代码清单 7-9 展示了 Add 方法，这个方法接收一个名为 tasks 的可变参数。可变参数可以接受任意数量的值作为传入参数。这个例子里，这些传入的值必须是一个接收一个整数且什么都不返回的函数。函数执行时的参数 tasks 是一个存储所有这些传入函数值的切片。

现在让我们来看一下 run 方法，如代码清单 7-10 所示。

代码清单 7-10　**runner**/runner.go：第 72 行到第 85 行

```
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 }
```

代码清单 7-10 的第 73 行的 run 方法会迭代 tasks 切片，并按顺序执行每个函数。函数会在第 81 行被执行。在执行之前，会在第 76 行调用 gotInterrupt 方法来检查是否有要从操作系统接收的事件。

代码清单 7-11 中的方法 gotInterrupt 展示了带 default 分支的 select 语句的经典用法。

代码清单 7-11　**runner**/runner.go：第 87 行到第 101 行

```
 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 }
```

在第 91 行，代码试图从 `interrupt` 通道去接收信号。一般来说，`select` 语句在没有任何要接收的数据时会阻塞，不过有了第 98 行的 `default` 分支就不会阻塞了。`default` 分支会将接收 `interrupt` 通道的阻塞调用转变为非阻塞的。如果 `interrupt` 通道有中断信号需要接收，就会接收并处理这个中断。如果没有需要接收的信号，就会执行 `default` 分支。

当收到中断信号后，代码会通过在第 93 行调用 `Stop` 方法来停止接收之后的所有事件。之后函数返回 `true`。如果没有收到中断信号，在第 99 行该方法会返回 `false`。本质上，`gotInterrupt` 方法会让 goroutine 检查中断信号，如果没有发出中断信号，就继续处理工作。

这个包里的最后一个方法名为 `Start`，如代码清单 7-12 所示。

代码清单 7-12　**runner**/runner.go：第 51 行到第 70 行

```
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 }
```

方法 `Start` 实现了程序的主流程。在代码清单 7-12 的第 52 行，`Start` 设置了 `gotInterrupt`

方法要从操作系统接收的中断信号。在第 56 行到第 59 行，声明了一个匿名函数，并单独启动 goroutine 来执行。这个 goroutine 会执行一系列被赋予的任务。在第 58 行，在 goroutine 的内部调用了 `run` 方法，并将这个方法返回的 `error` 接口值发送到 `complete` 通道。一旦 `error` 接口的值被接收，该 goroutine 就会通过通道将这个值返回给调用者。

创建 goroutine 后，`Start` 进入一个 `select` 语句，阻塞等待两个事件中的任意一个。如果从 `complete` 通道接收到 `error` 接口值,那么该 goroutine 要么在规定的时间内完成了分配的工作，要么收到了操作系统的中断信号。无论哪种情况，收到的 `error` 接口值都会被返回，随后方法终止。如果从 `timeout` 通道接收到 `time.Time` 值，就表示 goroutine 没有在规定的时间内完成工作。这种情况下，程序会返回 `ErrTimeout` 变量。

现在看过了 `runner` 包的代码，并了解了代码是如何工作的，让我们看一下 main.go 代码文件中的测试程序，如代码清单 7-13 所示。

代码清单 7-13 **runner**/main/main.go

```
01 // 这个示例程序演示如何使用通道来监视
02 // 程序运行的时间，以在程序运行时间过长
03 // 时如何终止程序
03 package main
04
05 import (
06     "log"
07     "time"
08
09     "github.com/goinaction/code/chapter7/patterns/runner"
10 )
11
12 // timeout 规定了必须在多少秒内处理完成
13 const timeout = 3 * time.Second
14
15 // main 是程序的入口
16 func main() {
17     log.Println("Starting work.")
18
19     // 为本次执行分配超时时间
20     r := runner.New(timeout)
21
22     // 加入要执行的任务
23     r.Add(createTask(), createTask(), createTask())
24
25     // 执行任务并处理结果
26     if err := r.Start(); err != nil {
27         switch err {
28         case runner.ErrTimeout:
29             log.Println("Terminating due to timeout.")
30             os.Exit(1)
31         case runner.ErrInterrupt:
32             log.Println("Terminating due to interrupt.")
33             os.Exit(2)
```

```
34          }
35      }
36
37      log.Println("Process ended.")
38  }
39
40  // createTask 返回一个根据 id
41  // 休眠指定秒数的示例任务
42  func createTask() func(int) {
43      return func(id int) {
44          log.Printf("Processor - Task #%d.", id)
45          time.Sleep(time.Duration(id) * time.Second)
46      }
47  }
```

代码清单 7-13 的第 16 行是 `main` 函数。在第 20 行，使用 `timeout` 作为超时时间传给 `New` 函数，并返回了一个指向 `Runner` 类型的指针。之后在第 23 行，使用 `createTask` 函数创建了几个任务，并被加入 `Runner` 里。在第 42 行声明了 `createTask` 函数。这个函数创建的任务只是休眠了一段时间，用来模拟正在进行工作。增加完任务后，在第 26 行调用了 `Start` 方法，`main` 函数会等待 `Start` 方法的返回。

当 `Start` 返回时，会检查其返回的 `error` 接口值，并存入 `err` 变量。如果确实发生了错误，代码会根据 `err` 变量的值来判断方法是由于超时终止的，还是由于收到了中断信号终止。如果没有错误，任务就是按时执行完成的。如果执行超时，程序就会用错误码 1 终止。如果接收到中断信号，程序就会用错误码 2 终止。其他情况下，程序会使用错误码 0 正常终止。

7.2 pool

本章会介绍`pool`包[①]。这个包用于展示如何使用有缓冲的通道实现资源池，来管理可以在任意数量的goroutine之间共享及独立使用的资源。这种模式在需要共享一组静态资源的情况（如共享数据库连接或者内存缓冲区）下非常有用。如果goroutine需要从池里得到这些资源中的一个，它可以从池里申请，使用完后归还到资源池里。

让我们看一下 `pool` 包里的 pool.go 代码文件，如代码清单 7-14 所示。

代码清单 7-14 **`pool`**/pool.go

```
01 // Fatih Arslan 和 Gabriel Aszalos 协助完成了这个示例
02 // 包 pool 管理用户定义的一组资源
03 package pool
04
05 import (
06     "errors"
07     "log"
```

① 本书是以 Go 1.5 版本为基础写作而成的。在 Go 1.6 及之后的版本中，标准库里自带了资源池的实现（`sync.Pool`）。推荐使用。——译者注

```
08     "io"
09     "sync"
10 )
11
12 // Pool 管理一组可以安全地在多个 goroutine 间
13 // 共享的资源。被管理的资源必须
14 // 实现 io.Closer 接口
15 type Pool struct {
16     m          sync.Mutex
17     resources  chan io.Closer
18     factory    func() (io.Closer, error)
19     closed     bool
20 }
21
22 // ErrPoolClosed 表示请求（Acquire）了一个
23 // 已经关闭的池
24 var ErrPoolClosed = errors.New("Pool has been closed.")
25
26 // New 创建一个用来管理资源的池。
27 // 这个池需要一个可以分配新资源的函数，
28 // 并规定池的大小
29 func New(fn func() (io.Closer, error), size uint) (*Pool, error) {
30     if size <= 0 {
31         return nil, errors.New("Size value too small.")
32     }
33
34     return &Pool{
35         factory:   fn,
36         resources: make(chan io.Closer, size),
37     }, nil
38 }
39
40 // Acquire 从池中获取一个资源
41 func (p *Pool) Acquire() (io.Closer, error) {
42     select {
43     // 检查是否有空闲的资源
44     case r, ok := <-p.resources:
45         log.Println("Acquire:", "Shared Resource")
46         if !ok {
47             return nil, ErrPoolClosed
48         }
49         return r, nil
50
51     // 因为没有空闲资源可用，所以提供一个新资源
52     default:
53         log.Println("Acquire:", "New Resource")
54         return p.factory()
55     }
56 }
57
58 // Release 将一个使用后的资源放回池里
59 func (p *Pool) Release(r io.Closer) {
60     // 保证本操作和 Close 操作的安全
61     p.m.Lock()
```

```
62      defer p.m.Unlock()
63
64      // 如果池已经被关闭，销毁这个资源
65      if p.closed {
66          r.Close()
67          return
68      }
69
70      select {
71      // 试图将这个资源放入队列
72      case p.resources <- r:
73          log.Println("Release:", "In Queue")
74
75      // 如果队列已满，则关闭这个资源
76      default:
77          log.Println("Release:", "Closing")
78          r.Close()
79      }
80 }
81
82 // Close 会让资源池停止工作，并关闭所有现有的资源
83 func (p *Pool) Close() {
84      // 保证本操作与 Release 操作的安全
85      p.m.Lock()
86      defer p.m.Unlock()
87
88      // 如果 pool 已经被关闭，什么也不做
89      if p.closed {
90          return
91      }
92
93      // 将池关闭
94      p.closed = true
95
96      // 在清空通道里的资源之前，将通道关闭
97      // 如果不这样做，会发生死锁
98      close(p.resources)
99
100     // 关闭资源
101     for r := range p.resources {
102         r.Close()
103     }
104 }
```

代码清单 7-14 中的 `pool` 包的代码声明了一个名为 `Pool` 的结构，该结构允许调用者根据所需数量创建不同的资源池。只要某类资源实现了 `io.Closer` 接口，就可以用这个资源池来管理。让我们看一下 `Pool` 结构的声明，如代码清单 7-15 所示。

代码清单 7-15　`pool`/pool.go：第 12 行到第 20 行

```
12 // Pool 管理一组可以安全地在多个 goroutine 间
13 // 共享的资源。被管理的资源必须
```

```
14 // 实现 io.Closer 接口
15 type Pool struct {
16     m         sync.Mutex
17     resources chan io.Closer
18     factory   func() (io.Closer, error)
19     closed    bool
20 }
```

Pool 结构声明了 4 个字段，每个字段都用来辅助以 goroutine 安全的方式来管理资源池。在第 16 行，结构以一个 sync.Mutex 类型的字段开始。这个互斥锁用来保证在多个 goroutine 访问资源池时，池内的值是安全的。第二个字段名为 resources，被声明为 io.Closer 接口类型的通道。这个通道是作为一个有缓冲的通道创建的，用来保存共享的资源。由于通道的类型是一个接口，所以池可以管理任意实现了 io.Closer 接口的资源类型。

factory 字段是一个函数类型。任何一个没有输入参数且返回一个 io.Closer 和一个 error 接口值的函数，都可以赋值给这个字段。这个函数的目的是，当池需要一个新资源时，可以用这个函数创建。这个函数的实现细节超出了 pool 包的范围，并且需要由包的使用者实现并提供。

第 19 行中的最后一个字段是 closed 字段。这个字段是一个标志，表示 Pool 是否已经被关闭。现在已经了解了 Pool 结构的声明，让我们看一下第 24 行声明的 error 接口变量，如代码清单 7-16 所示。

代码清单 7-16　pool/pool.go：第 22 行到第 24 行

```
22 // ErrPoolClosed 表示请求（Acquire）了一个
23 // 已经关闭的池
24 var ErrPoolClosed = errors.New("Pool has been closed.")
```

Go 语言里会经常创建 error 接口变量。这可以让调用者来判断某个包里的函数或者方法返回的具体的错误值。当调用者对一个已经关闭的池调用 Acquire 方法时，会返回代码清单 7-16 里的 error 接口变量。因为 Acquire 方法可能返回多个不同类型的错误，所以 Pool 已经关闭时会关闭时返回这个错误变量可以让调用者从其他错误中识别出这个特定的错误。

既然已经声明了 Pool 类型和 error 接口值，我们就可以开始看一下 pool 包里声明的函数和方法了。让我们从池的工厂函数开始，这个函数名为 New，如代码清单 7-17 所示。

代码清单 7-17　pool/pool.go：第 26 行到第 38 行

```
26 // New 创建一个用来管理资源的池。
27 // 这个池需要一个可以分配新资源的函数，
28 // 并规定池的大小
29 func New(fn func() (io.Closer, error), size uint) (*Pool, error) {
30     if size <= 0 {
31         return nil, errors.New("Size value too small.")
32     }
33
34     return &Pool{
```

```
35          factory:   fn,
36          resources: make(chan io.Closer, size),
37      }, nil
38  }
```

代码清单 7-17 中的 New 函数接受两个参数，并返回两个值。第一个参数 fn 声明为一个函数类型，这个函数不接受任何参数，返回一个 io.Closer 和一个 error 接口值。这个作为参数的函数是一个工厂函数，用来创建由池管理的资源的值。第二个参数 size 表示为了保存资源而创建的有缓冲的通道的缓冲区大小。

第 30 行检查了 size 的值，保证这个值不小于等于 0。如果这个值小于等于 0，就会使用 nil 值作为返回的 pool 指针值，然后为该错误创建一个 error 接口值。因为这是这个函数唯一可能返回的错误值，所以不需要为这个错误单独创建和使用一个 error 接口变量。如果能够接受传入的 size，就会创建并初始化一个新的 Pool 值。在第 35 行，函数参数 fn 被赋值给 factory 字段，并且在第 36 行，使用 size 值创建有缓冲的通道。在 return 语句里，可以构造并初始化任何值。因此，第 34 行的 return 语句用指向新创建的 Pool 类型值的指针和 nil 值作为 error 接口值，返回给函数的调用者。

在创建并初始化 Pool 类型的值之后，接下来让我们来看一下 Acquire 方法，如代码清单 7-18 所示。这个方法可以让调用者从池里获得资源。

代码清单 7-18　**pool**/pool.go：第 40 行到第 56 行

```
40  // Acquire 从池中获取一个资源
41  func (p *Pool) Acquire() (io.Closer, error) {
42      select {
43      // 检查是否有空闲的资源
44      case r, ok := <-p.resources:
45          log.Println("Acquire:", "Shared Resource")
46          if !ok {
47              return nil, ErrPoolClosed
48          }
49          return r, nil
50
51      // 因为没有空闲资源可用，所以提供一个新资源
52      default:
53          log.Println("Acquire:", "New Resource")
54          return p.factory()
55      }
56  }
```

代码清单 7-18 包含了 Acquire 方法的代码。这个方法在还有可用资源时会从资源池里返回一个资源，否则会为该调用创建并返回一个新的资源。这个实现是通过 select/case 语句来检查有缓冲的通道里是否还有资源来完成的。如果通道里还有资源，如第 44 行到第 49 行所写，就取出这个资源，并返回给调用者。如果该通道里没有资源可取，就会执行 default 分支。在这个示例中，在第 54 行执行用户提供的工厂函数，并且创建并返回一个新资源。

如果不再需要已经获得的资源，必须将这个资源释放回资源池里。这是 Release 方法的任

务。不过在理解 Release 方法的代码背后的机制之前，我们需要先看一下 Close 方法，如代码清单 7-19 所示。

代码清单 7-19　pool/pool.go：第 82 行到第 104 行

```
 82 // Close 会让资源池停止工作，并关闭所有现有的资源
 83 func (p *Pool) Close() {
 84     // 保证本操作与 Release 操作的安全
 85     p.m.Lock()
 86     defer p.m.Unlock()
 87
 88     // 如果 pool 已经被关闭，什么也不做
 89     if p.closed {
 90         return
 91     }
 92
 93     // 将池关闭
 94     p.closed = true
 95
 96     // 在清空通道里的资源之前，将通道关闭
 97     // 如果不这样做，会发生死锁
 98     close(p.resources)
 99
100     // 关闭资源
101     for r := range p.resources {
102         r.Close()
103     }
104 }
```

一旦程序不再使用资源池，需要调用这个资源池的 Close 方法。代码清单 7-19 中展示了 Close 方法的代码。在第 98 行到第 101 行，这个方法关闭并清空了有缓冲的通道，并将缓冲的空闲资源关闭。需要注意的是，在同一时刻只能有一个 goroutine 执行这段代码。事实上，当这段代码被执行时，必须保证其他 goroutine 中没有同时执行 Release 方法。你一会儿就会理解为什么这很重要。

在第 85 行到第 86 行，互斥量被加锁，并在函数返回时解锁。在第 89 行，检查 closed 标志，判断池是不是已经关闭。如果已经关闭，该方法会直接返回，并释放锁。如果这个方法第一次被调用，就会将这个标志设置为 true，并关闭且清空 resources 通道。

现在我们可以看一下 Release 方法，看看这个方法是如何和 Close 方法配合的，如代码清单 7-20 所示。

代码清单 7-20　pool/pool.go：第 58 行到第 80 行

```
58 // Release 将一个使用后的资源放回池里
59 func (p *Pool) Release(r io.Closer) {
60     // 保证本操作和 Close 操作的安全
61     p.m.Lock()
62     defer p.m.Unlock()
63
```

```
64      // 如果池已经被关闭，销毁这个资源
65      if p.closed {
66          r.Close()
67          return
68      }
69
70      select {
71      // 试图将这个资源放入队列
72      case p.resources <- r:
73          log.Println("Release:", "In Queue")
74
75      // 如果队列已满，则关闭这个资源
76      default:
77          log.Println("Release:", "Closing")
78          r.Close()
79      }
80  }
```

在代码清单 7-20 中可以找到 `Release` 方法的实现。该方法一开始在第 61 行和第 62 行对互斥量进行加锁和解锁。这和 `Close` 方法中的互斥量是同一个互斥量。这样可以阻止这两个方法在不同 goroutine 里同时运行。使用互斥量有两个目的。第一，可以保护第 65 行中读取 `closed` 标志的行为，保证同一时刻不会有其他 goroutine 调用 `Close` 方法写同一个标志。第二，我们不想往一个已经关闭的通道里发送数据，因为那样会引起崩溃。如果 `closed` 标志是 `true`，我们就知道 `resources` 通道已经被关闭。

在第 66 行，如果池已经被关闭，会直接调用资源值 `r` 的 `Close` 方法。因为这时已经清空并关闭了池，所以无法将资源重新放回到该资源池里。对 `closed` 标志的读写必须进行同步，否则可能误导其他 goroutine，让其认为该资源池依旧是打开的，并试图对通道进行无效的操作。

现在看过了池的代码，了解了池是如何工作的，让我们看一下 main.go 代码文件里的测试程序，如代码清单 7-21 所示。

代码清单 7-21　`pool`/main/main.go

```
01 // 这个示例程序展示如何使用 pool 包
02 // 来共享一组模拟的数据库连接
03 package main
04
05 import (
06      "log"
07      "io"
08      "math/rand"
09      "sync"
10      "sync/atomic"
11      "time"
12
13      "github.com/goinaction/code/chapter7/patterns/pool"
14 )
15
16 const (
```

```
    maxGoroutines   = 25 // 要使用的 goroutine 的数量
    pooledResources = 2  // 池中的资源的数量
)

// dbConnection 模拟要共享的资源
type dbConnection struct {
    ID int32
}

// Close 实现了 io.Closer 接口，以便 dbConnection
// 可以被池管理。Close 用来完成任意资源的
// 释放管理
func (dbConn *dbConnection) Close() error {
    log.Println("Close: Connection", dbConn.ID)
    return nil
}

// idCounter 用来给每个连接分配一个独一无二的 id
var idCounter int32

// createConnection 是一个工厂函数，
// 当需要一个新连接时，资源池会调用这个函数
func createConnection() (io.Closer, error) {
    id := atomic.AddInt32(&idCounter, 1)
    log.Println("Create: New Connection", id)

    return &dbConnection{id}, nil
}

// main 是所有 Go 程序的入口
func main() {
    var wg sync.WaitGroup
    wg.Add(maxGoroutines)

    // 创建用来管理连接的池
    p, err := pool.New(createConnection, pooledResources)
    if err != nil {
        log.Println(err)
    }

    // 使用池里的连接来完成查询
    for query := 0; query < maxGoroutines; query++ {
        // 每个 goroutine 需要自己复制一份要
        // 查询值的副本，不然所有的查询会共享
        // 同一个查询变量
        go func(q int) {
            performQueries(q, p)
            wg.Done()
        }(query)
    }

    // 等待 goroutine 结束
    wg.Wait()

```

```
71      // 关闭池
72      log.Println("Shutdown Program.")
73      p.Close()
74  }
75
76  // performQueries 用来测试连接的资源池
77  func performQueries(query int, p *pool.Pool) {
78      // 从池里请求一个连接
79      conn, err := p.Acquire()
80      if err != nil {
81          log.Println(err)
82          return
83      }
84
85      // 将该连接释放回池里
86      defer p.Release(conn)
87
88      // 用等待来模拟查询响应
89      time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
90      log.Printf("QID[%d] CID[%d]\n", query, conn.(*dbConnection).ID)
91  }
```

代码清单 7-21 展示的 main.go 中的代码使用 `pool` 包来管理一组模拟数据库连接的连接池。代码一开始声明了两个常量 `maxGoroutines` 和 `pooledResource`，用来设置 goroutine 的数量以及程序将要使用资源的数量。资源的声明以及 `io.Closer` 接口的实现如代码清单 7-22 所示。

代码清单 7-22　`pool`/main/main.go：第 21 行到第 32 行

```
21  // dbConnection 模拟要共享的资源
22  type dbConnection struct {
23      ID int32
24  }
25
26  // Close 实现了 io.Closer 接口，以便 dbConnection
27  // 可以被池管理。Close 用来完成任意资源的
28  // 释放管理
29  func (dbConn *dbConnection) Close() error {
30      log.Println("Close: Connection", dbConn.ID)
31      return nil
32  }
```

代码清单 7-22 展示了 `dbConnection` 结构的声明以及 `io.Closer` 接口的实现。`dbConnection` 类型模拟了管理数据库连接的结构，当前版本只包含一个字段 `ID`，用来保存每个连接的唯一标识。`Close` 方法只是报告了连接正在被关闭，并显示出要关闭连接的标识。

接下来我们来看一下创建 `dbConnection` 值的工厂函数，如代码清单 7-23 所示。

代码清单 7-23　`pool`/main/main.go：第 34 行到第 44 行

```
34  // idCounter 用来给每个连接分配一个独一无二的 id
```

```
35 var idCounter int32
36
37 // createConnection 是一个工厂函数，
38 // 当需要一个新连接时，资源池会调用这个函数
39 func createConnection() (io.Closer, error) {
40     id := atomic.AddInt32(&idCounter, 1)
41     log.Println("Create: New Connection", id)
42
43     return &dbConnection{id}, nil
44 }
```

代码清单 7-23 展示了 `createConnection` 函数的实现。这个函数给连接生成了一个唯一标识，显示连接正在被创建，并返回指向带有唯一标识的 `dbConnection` 类型值的指针。唯一标识是通过 `atomic.AddInt32` 函数生成的。这个函数可以安全地增加包级变量 `idCounter` 的值。现在有了资源以及工厂函数，我们可以配合使用 `pool` 包了。

接下来让我们看一下 `main` 函数的代码，如代码清单 7-24 所示。

代码清单 7-24　`pool`/main/main.go：第 48 行到第 55 行

```
48     var wg sync.WaitGroup
49     wg.Add(maxGoroutines)
50
51     // 创建用来管理连接的池
52     p, err := pool.New(createConnection, pooledResources)
53     if err != nil {
54         log.Println(err)
55     }
```

在第 48 行，`main` 函数一开始就声明了一个 `WaitGroup` 值，并将 `WaitGroup` 的值设置为要创建的 goroutine 的数量。之后使用 `pool` 包里的 `New` 函数创建了一个新的 `Pool` 类型。工厂函数和要管理的资源的数量会传入 `New` 函数。这个函数会返回一个指向 `Pool` 值的指针，并检查可能的错误。现在我们有了一个 `Pool` 类型的资源池实例，就可以创建 goroutine，并使用这个资源池在 goroutine 之间共享资源，如代码清单 7-25 所示。

代码清单 7-25　`pool`/main/main.go：第 57 行到第 66 行

```
57     // 使用池里的连接来完成查询
58     for query := 0; query < maxGoroutines; query++ {
59         // 每个 goroutine 需要自己复制一份要
60         // 查询值的副本，不然所有的查询会共享
61         // 同一个查询变量
62         go func(q int) {
63             performQueries(q, p)
64             wg.Done()
65         }(query)
66     }
```

代码清单 7-25 中用一个 `for` 循环创建要使用池的 goroutine。每个 goroutine 调用一次 `performQueries` 函数然后退出。`performQueries` 函数需要传入一个唯一的 `ID` 值用于做日

志以及一个指向 Pool 的指针。一旦所有的 goroutine 都创建完成，main 函数就等待所有 goroutine 执行完毕，如代码清单 7-26 所示。

代码清单 7-26　pool/main/main.go：第 68 行到第 73 行

```
68      // 等待 goroutine 结束
69      wg.Wait()
70
71      // 关闭池
72      log.Println("Shutdown Program.")
73      p.Close()
```

在代码清单 7-26 中，main 函数等待 WaitGroup 实例的 Wait 方法执行完成。一旦所有 goroutine 都报告其执行完成，就关闭 Pool，并且终止程序。接下来，让我们看一下 performQueries 函数。这个函数使用了池的 Acquire 方法和 Release 方法，如代码清单 7-27 所示。

代码清单 7-27　pool/main/main.go：第 76 行到第 91 行

```
76 // performQueries 用来测试连接的资源池
77 func performQueries(query int, p *pool.Pool) {
78      // 从池里请求一个连接
79      conn, err := p.Acquire()
80      if err != nil {
81          log.Println(err)
82          return
83      }
84
85      // 将该连接释放回池里
86      defer p.Release(conn)
87
88      // 用等待来模拟查询响应
89      time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
90      log.Printf("QID[%d] CID[%d]\n", query, conn.(*dbConnection).ID)
91 }
```

代码清单 7-27 展示了 performQueries 的实现。这个实现使用了 pool 的 Acquire 方法和 Release 方法。这个函数首先调用了 Acquire 方法，从池里获得 dbConnection。之后会检查返回的 error 接口值，在第 86 行，再使用 defer 语句在函数退出时将 dbConnection 释放回池里。在第 89 行和第 90 行，随机休眠一段时间，以此来模拟使用 dbConnection 工作时间。

7.3　work

work 包的目的是展示如何使用无缓冲的通道来创建一个 goroutine 池，这些 goroutine 执行并控制一组工作，让其并发执行。在这种情况下，使用无缓冲的通道要比随意指定一个缓冲区大小的有缓冲的通道好，因为这个情况下既不需要一个工作队列，也不需要一组 goroutine 配合执

行。无缓冲的通道保证两个 goroutine 之间的数据交换。这种使用无缓冲的通道的方法允许使用者知道什么时候 goroutine 池正在执行工作，而且如果池里的所有 goroutine 都忙，无法接受新的工作的时候，也能及时通过通道来通知调用者。使用无缓冲的通道不会有工作在队列里丢失或者卡住，所有工作都会被处理。

让我们来看一下 work 包里的 work.go 代码文件，如代码清单 7-28 所示。

代码清单 7-28 work/work.go

```
01 // Jason Waldrip 协助完成了这个示例
02 // work 包管理一个 goroutine 池来完成工作
03 package work
04
05 import "sync"
06
07 // Worker 必须满足接口类型，
08 // 才能使用工作池
09 type Worker interface {
10     Task()
11 }
12
13 // Pool 提供一个 goroutine 池，这个池可以完成
14 // 任何已提交的 Worker 任务
15 type Pool struct {
16     work chan Worker
17     wg   sync.WaitGroup
18 }
19
20 // New 创建一个新工作池
21 func New(maxGoroutines int) *Pool {
22     p := Pool{
23         work: make(chan Worker),
24     }
25
26     p.wg.Add(maxGoroutines)
27     for i := 0; i < maxGoroutines; i++ {
28         go func() {
29             for w := range p.work {
30                 w.Task()
31             }
32             p.wg.Done()
33         }()
34     }
35
36     return &p
37 }
38
39 // Run 提交工作到工作池
40 func (p *Pool) Run(w Worker) {
41     p.work <- w
42 }
43
```

```
44 // Shutdown 等待所有 goroutine 停止工作
45 func (p *Pool) Shutdown() {
46     close(p.work)
47     p.wg.Wait()
48 }
```

代码清单 7-28 中展示的 work 包一开始声明了名为 Worker 的接口和名为 Pool 的结构，如代码清单 7-29 所示。

代码清单 7-29 **work**/work.go：第 07 行到第 18 行

```
07 // Worker 必须满足接口类型，
08 // 才能使用工作池
09 type Worker interface {
10     Task()
11 }
12
13 // Pool 提供一个 goroutine 池，这个池可以完成
14 // 任何已提交的 Worker 任务
15 type Pool struct {
16     work chan Worker
17     wg   sync.WaitGroup
18 }
```

代码清单 7-29 的第 09 行中的 Worker 接口声明了一个名为 Task 的方法。在第 15 行，声明了名为 Pool 的结构，这个结构类型实现了 goroutine 池，并实现了一些处理工作的方法。这个结构类型声明了两个字段，一个名为 work（一个 Worker 接口类型的通道），另一个名为 wg 的 sync.WaitGroup 类型。

接下来，让我们来看一下 work 包的工厂函数，如代码清单 7-30 所示。

代码清单 7-30 **work**/work.go：第 20 行到第 37 行

```
20 // New 创建一个新工作池
21 func New(maxGoroutines int) *Pool {
22     p := Pool{
23         work: make(chan Worker),
24     }
25
26     p.wg.Add(maxGoroutines)
27     for i := 0; i < maxGoroutines; i++ {
28         go func() {
29             for w := range p.work {
30                 w.Task()
31             }
32             p.wg.Done()
33         }()
34     }
35
36     return &p
37 }
```

代码清单 7-30 展示了 New 函数，这个函数使用固定数量的 goroutine 来创建一个工作池。goroutine 的数量作为参数传给 New 函数。在第 22 行，创建了一个 Pool 类型的值，并使用无缓冲的通道来初始化 work 字段。

之后，在第 26 行，初始化 WaitGroup 需要等待的数量，并在第 27 行到第 34 行，创建了同样数量的 goroutine。这些 goroutine 只接收 Worker 类型的接口值，并调用这个值的 Task 方法，如代码清单 7-31 所示。

代码清单 7-31　work/work.go：第 28 行到第 33 行

```
28          go func() {
29              for w := range p.work {
30                  w.Task()
31              }
32              p.wg.Done()
33          }()
```

代码清单 7-31 里的 for range 循环会一直阻塞，直到从 work 通道收到一个 Worker 接口值。如果收到一个值，就会执行这个值的 Task 方法。一旦 work 通道被关闭，for range 循环就会结束，并调用 WaitGroup 的 Done 方法。然后 goroutine 终止。

现在我们可以创建一个等待并执行工作的 goroutine 池了。让我们看一下如何向池里提交工作，如代码清单 7-32 所示。

代码清单 7-32　work/work.go：第 39 行到第 42 行

```
39 // Run 提交工作到工作池
40 func (p *Pool) Run(w Worker) {
41     p.work <- w
42 }
```

代码清单 7-32 展示了 Run 方法。这个方法可以向池里提交工作。该方法接受一个 Worker 类型的接口值作为参数，并将这个值通过 work 通道发送。由于 work 通道是一个无缓冲的通道，调用者必须等待工作池里的某个 goroutine 接收到这个值才会返回。这正是我们想要的，这样可以保证调用的 Run 返回时，提交的工作已经开始执行。

在某个时间点，需要关闭工作池。这是 Shutdown 方法所做的事情，如代码清单 7-33 所示。

代码清单 7-33　work/work.go：第 44 行到第 48 行

```
44 // Shutdown 等待所有 goroutine 停止工作
45 func (p *Pool) Shutdown() {
46     close(p.work)
47     p.wg.Wait()
48 }
```

代码清单 7-33 中的 Shutdown 方法做了两件事，首先，它关闭了 work 通道，这会导致所有池里的 goroutine 停止工作，并调用 WaitGroup 的 Done 方法；然后，Shutdown 方法调用 WaitGroup 的 Wait 方法，这会让 Shutdown 方法等待所有 goroutine 终止。

我们看了 work 包的代码，并了解了它是如何工作的，接下来让我们看一下 main.go 源代码文件中的测试程序，如代码清单 7-34 所示。

代码清单 7-34　work/main/main.go

```
01 // 这个示例程序展示如何使用 work 包
02 // 创建一个 goroutine 池并完成工作
03 package main
04
05 import (
06     "log"
07     "sync"
08     "time"
09
10     "github.com/goinaction/code/chapter7/patterns/work"
11 )
12
13 // names 提供了一组用来显示的名字
14 var names = []string{
15     "steve",
16     "bob",
17     "mary",
18     "therese",
19     "jason",
20 }
21
22 // namePrinter 使用特定方式打印名字
23 type namePrinter struct {
24     name string
25 }
26
27 // Task 实现 Worker 接口
28 func (m *namePrinter) Task() {
29     log.Println(m.name)
30     time.Sleep(time.Second)
31 }
32
33 // main 是所有 Go 程序的入口
34 func main() {
35     // 使用两个 goroutine 来创建工作池
36     p := work.New(2)
37
38     var wg sync.WaitGroup
39     wg.Add(100 * len(names))
40
41     for i := 0; i < 100; i++ {
42         // 迭代 names 切片
43         for _, name := range names {
44             // 创建一个 namePrinter 并提供
45             // 指定的名字
46             np := namePrinter{
47                 name: name,
48             }
```

```
49
50              go func() {
51                  // 将任务提交执行。当 Run 返回时
52                  // 我们就知道任务已经处理完成
53                  p.Run(&np)
54                  wg.Done()
55              }()
56          }
57      }
58
59      wg.Wait()
60
61      // 让工作池停止工作，等待所有现有的
62      // 工作完成
63      p.Shutdown()
64  }
```

代码清单 7-34 展示了使用 work 包来完成名字显示工作的测试程序。这段代码一开始在第 14 行声明了名为 names 的包级的变量，这个变量被声明为一个字符串切片。这个切片使用 5 个名字进行了初始化。然后声明了名为 namePrinter 的类型，如代码清单 7-35 所示。

代码清单 7-35　**work**/main/main.go：第 22 行到第 31 行

```
22 // namePrinter 使用特定方式打印名字
23 type namePrinter struct {
24     name string
25 }
26
27 // Task 实现 Worker 接口
28 func (m *namePrinter) Task() {
29     log.Println(m.name)
30     time.Sleep(time.Second)
31 }
```

在代码清单 7-35 的第 23 行，声明了 namePrinter 类型，接着是这个类型对 Worker 接口的实现。这个类型的工作任务是在显示器上显示名字。这个类型只包含一个字段，即 name，它包含要显示的名字。Worker 接口的实现 Task 函数用 log.Println 函数来显示名字，之后等待 1 秒再退出。等待这 1 秒只是为了让测试程序运行的速度慢一些，以便看到并发的效果。

有了 Worker 接口的实现，我们就可以看一下 main 函数内部的代码了，如代码清单 7-36 所示。

代码清单 7-36　**work**/main/main.go：第 33 行到第 64 行

```
33 // main 是所有 Go 程序的入口
34 func main() {
35     // 使用两个 goroutine 来创建工作池
36     p := work.New(2)
37
38     var wg sync.WaitGroup
39     wg.Add(100 * len(names))
```

```
40
41      for i := 0; i < 100; i++ {
42          // 迭代 names 切片
43          for _, name := range names {
44              // 创建一个 namePrinter 并提供
45              // 指定的名字
46              np := namePrinter{
47                  name: name,
48              }
49
50              go func() {
51                  // 将任务提交执行。当 Run 返回时
52                  // 我们就知道任务已经处理完成
53                  p.Run(&np)
54                  wg.Done()
55              }()
56          }
57      }
58
59      wg.Wait()
60
61      // 让工作池停止工作，等待所有现有的
62      // 工作完成
63      p.Shutdown()
64  }
```

在代码清单 7-36 第 36 行，调用 `work` 包里的 `New` 函数创建一个工作池。这个调用传入的参数是 2，表示这个工作池只会包含两个执行任务的 goroutine。在第 38 行和第 39 行，声明了一个 `WaitGroup`，并初始化为要执行任务的 goroutine 数。在这个例子里，`names` 切片里的每个名字都会创建 100 个 goroutine 来提交任务。这样就会有一堆 goroutine 互相竞争，将任务提交到池里。

在第 41 行到第 43 行，内部和外部的 `for` 循环用来声明并创建所有的 goroutine。每次内部循环都会创建一个 `namePrinter` 类型的值，并提供一个用来打印的名字。之后，在第 50 行，声明了一个匿名函数，并创建一个 goroutine 执行这个函数。这个 goroutine 会调用工作池的 `Run` 方法，将 `namePrinter` 的值提交到池里。一旦工作池里的 goroutine 接收到这个值，`Run` 方法就会返回。这也会导致 goroutine 将 `WaitGroup` 的计数递减，并终止 goroutine。

一旦所有的 goroutine 都创建完成，`main` 函数就会调用 `WaitGroup` 的 `Wait` 方法。这个调用会等待所有创建的 goroutine 提交它们的工作。一旦 `Wait` 返回，就会调用工作池的 `Shutdown` 方法来关闭工作池。`Shutdown` 方法直到所有的工作都做完才会返回。在这个例子里，最多只会等待两个工作的完成。

7.4 小结

- 可以使用通道来控制程序的生命周期。
- 带 `default` 分支的 `select` 语句可以用来尝试向通道发送或者接收数据，而不会阻塞。

- 有缓冲的通道可以用来管理一组可复用的资源。
- 语言运行时会处理好通道的协作和同步。
- 使用无缓冲的通道来创建完成工作的 goroutine 池。
- 任何时间都可以用无缓冲的通道来让两个 goroutine 交换数据，在通道操作完成时一定保证对方接收到了数据。

第 8 章　标准库

本章主要内容

- 输出数据以及记录日志
- 对 JSON 进行编码和解码
- 处理输入/输出，并以流的方式处理数据
- 让标准库里多个包协同工作

什么是 Go 标准库？为什么这个库这么重要？Go 标准库是一组核心包，用来扩展和增强语言的能力。这些包为语言增加了大量不同的类型。开发人员可以直接使用这些类型，而不用再写自己的包或者去下载其他人发布的第三方包。由于这些包和语言绑在一起发布，它们会得到以下特殊的保证：

- 每次语言更新，哪怕是小更新，都会带有标准库；
- 这些标准库会严格遵守向后兼容的承诺；
- 标准库是 Go 语言开发、构建、发布过程的一部分；
- 标准库由 Go 的构建者们维护和评审；
- 每次 Go 语言发布新版本时，标准库都会被测试，并评估性能。

这些保证让标准库变得很特殊，开发人员应该尽量利用这些标准库。使用标准库里的包可以使管理代码变得更容易，并且保证代码的稳定。不用担心程序无法兼容不同的 Go 语言版本，也不用管理第三方依赖。

如果标准库包含的包不够好用，那么这些好处实际上没什么用。Go 语言社区的开发者会比其他语言的开发者更依赖这些标准库里的包的原因是，标准库本身是经过良好设计的，并且比其他语言的标准库提供了更多的功能。社区里的 Go 开发者会依赖这些标准库里的包做更多其他语言中开发者无法做的事情，例如，网络、HTTP、图像处理、加密等。

本章中我们会大致了解标准库的一部分包。之后，我们会更详细地探讨 3 个非常有用的包：`log`、`json` 和 `io`。这些包也展示了 Go 语言提供的重要且有用的机制。

8.1 文档与源代码

标准库里包含众多的包，不可能在一章内把这些包都讲一遍。目前，标准库里总共有超过100个包，这些包被分到38个类别里，如代码清单8-1所示。

代码清单8-1 标准库里的顶级目录和包

```
archive   bufio      bytes     compress  container  crypto    database
debug     encoding   errors    expvar    flag       fmt       go
hash      html       image     index     io         log       math
mime      net        os        path      reflect    regexp    runtime
sort      strconv    strings   sync      syscall    testing   text
time      unicode    unsafe
```

代码清单8-1里列出的许多分类本身就是一个包。如果想了解所有包以及更详细的描述，Go语言团队在网站上维护了一个文档，参见http://golang.org/pkg/。

`golang`网站的`pkg`页面提供了每个包的`godoc`文档。图8-1展示了`golang`网站上`io`包的文档。

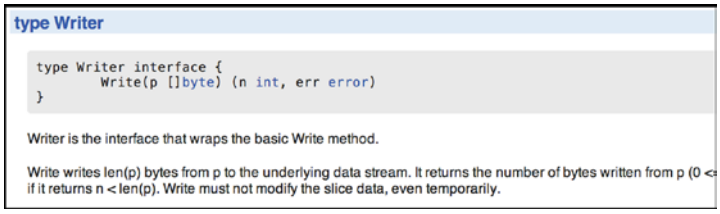

图8-1 golang.org/pkg/io/#Writer

如果想以交互的方式浏览文档，Sourcegraph索引了所有标准库的代码，以及大部分包含Go代码的公开库。图8-2是Sourcegraph网站的一个例子，展示的是`io`包的文档。

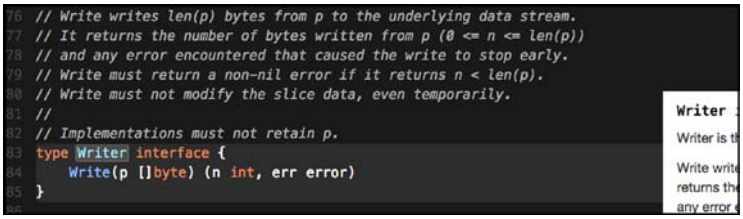

图8-2 sourcegraph.com/code.google.com/p/go/.GoPackage/io/.def/Writer

不管用什么方式安装Go，标准库的源代码都会安装在$GOROOT/src/pkg文件夹中。拥有标准库的源代码对Go工具正常工作非常重要。类似`godoc`、`gocode`甚至`go build`这些工具，都需要读取标准库的源代码才能完成其工作。如果源代码没有安装在以上文件夹中，或者无法通