

`done <- struct{}{}` 要短。

练习 8.3：在 `netcat3` 中，`conn` 接口有一个具体的类型 `*net.TCPConn`，它代表一个 TCP 连接。TCP 链接由两半边组成，可以通过 `CloseRead` 和 `CloseWrite` 方法分别关闭。修改主 goroutine，仅仅关闭连接的写半边，这样程序可以继续执行来输出来自 `reverb1` 服务器的回声，即使标准输入已经关闭。（对 `reverb2` 程序来说更难一些，见练习 8.4。）

8.4.2 管道

通道可以用来连接 goroutine，这样一个的输出是另一个的输入。这个叫管道 (pipeline)。下面的程序由三个 goroutine 组成，它们被两个通道连接起来，如图 8-1 所示。

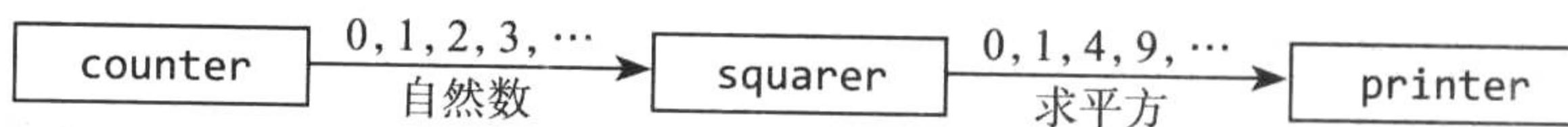


图 8-1 一个三级管道

第一个 goroutine 是 `counter`，产生一个 `0, 1, 2, ...` 的整数序列，然后通过一个管道发送给第二个 goroutine (叫 `square`)，计算数值的平方，然后将结果通过另一个通道发送给第三个 goroutine (叫 `printer`)，接收值并输出它们。为了简化例子，我们特意选择了非常简单的函数，尽管它们太简单以至于在现实程序中不可能有自己的 goroutine。

```

gopl.io/ch8/pipeline1

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // counter
    go func() {
        for x := 0; ; x++ {
            naturals <- x
        }
    }()

    // squarer
    go func() {
        for {
            x := <-naturals
            squares <- x * x
        }
    }()

    // printer (在主 goroutine 中)
    for {
        fmt.Println(<-squares)
    }
}
  
```

正如所期望的那样，程序输出无限的平方序列 `0, 1, 4, 9, ...`。像这样的管道出现在长期运行的服务器程序中，其中通道用于在包含无限循环的 goroutine 之间整个生命周期中的通信。如果要通过管道发送有限的数字怎么办？

如果发送方知道没有更多的数据要发送，告诉接收者所在 goroutine 可以停止等待是很有用的。这可以通过调用内置的 `close` 函数来关闭通道：

```
close(naturals)
```

在通道关闭后，任何后续的发送操作将会导致应用崩溃。当关闭的通道被读完（就是最

后一个发送的值被接收) 后, 所有后续的接收操作顺畅进行, 只是获取到的是零值。关闭 `naturals` 通道导致计算平方的循环快速运转, 并将结果 0 传递给 `printer` goroutine。

没有一个直接的方式来判断是否通道已经关闭, 但是这里有接收操作的一个变种, 它产生两个结果: 接收到的通道元素, 以及一个布尔值 (通常称为 `ok`), 它为 `true` 的时候代表接收成功, `false` 表示当前的接收操作在一个关闭的并且读完的通道上。使用这个特性, 可以修改 `square` 的循环, 当 `naturals` 通道读完以后, 关闭 `squares` 通道。

```
// square
go func() {
    for {
        x, ok := <-naturals
        if !ok {
            break // 通道关闭并且读完
        }
        squares <- x * x
    }
    close(squares)
}()
```

因为上面的语法比较笨拙, 而模式又比较通用, 所以该语言也提供了 `range` 循环语法以在通道上迭代。这个语法更方便接收在通道上所有发送的值, 接收完最后一个值后关闭循环。

下面的管道中, 当 `counter` goroutine 在 100 个元素后结束循环时, 它关闭 `naturals` 通道, 导致 `square` 结束循环并关闭 `squares` 通道。(在更复杂的程序中, 将 `counter` 和 `square` 的 goroutine 的 `close` 调用延迟到外层, 也是可以的。) 最终, 主 goroutine 结束, 然后程序退出。

```
gopl.io/ch8/pipeline2
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // counter
    go func() {
        for x := 0; x < 100; x++ {
            naturals <- x
        }
        close(naturals)
    }()

    // square
    go func() {
        for x := range naturals {
            squares <- x * x
        }
        close(squares)
    }()

    // printer (在主 goroutine 中)
    for x := range squares {
        fmt.Println(x)
    }
}
```

结束时, 关闭每一个通道不是必需的。只有在通知接收方 goroutine 所有的数据都发送完毕的时候才需要关闭通道。通道也是可以通过垃圾回收器根据它是否可以访问来决定是否回收它, 而不是根据它是否关闭。(不要将这个 `close` 操作和对于文件的 `close` 操作混淆。当结束的时候对每一个文件调用 `close` 方法是非常重要的。)

试图关闭一个已经关闭的通道会导致宕机，就像关闭一个空通道一样。关闭通道还可以作为一个广播机制，将在 8.9 节进行讨论。

8.4.3 单向通道类型

当程序演进时，将大的函数拆分为多个更小的是很自然的。上一个例子使用了三个 goroutine，两个通道用来通信，它们都是 `main` 的局部变量。程序自然划分为三个函数：

```
func counter(out chan int)
func squarer(out, in chan int)
func printer(in chan int)
```

`squarer` 函数处于管道的中间，使用两个参数：输入通道和输出通道。它们有相同的类型，但是用途是相反的：`in` 仅仅用来接收，`out` 仅仅用来发送，`in` 和 `out` 两个名字是特意使用的，但是没有什么东西阻碍 `squarer` 函数通过 `in` 来发送或者通过 `out` 来接收。

这是一个典型的安排，当一个通道用做函数的形参时，它几乎总是被有意地限制不能发送或不能接收。

将这种意图文档化可以避免误用，Go 的类型系统提供了单向通道类型，仅仅导出发送或接收操作。类型 `chan<- int` 是一个只能发送的通道，允许发送但不允许接收。反之，类型 `<-chan int` 是一个只能接收的 `int` 类型通道，允许接收但是不能发送。（`<-` 操作符相对于 `chan` 关键字的位置是一个帮助记忆的点）。违反这个原则会在编译时被检查出来。

因为 `close` 操作说明了通道上没有数据再发送，仅仅在发送方 goroutine 上才能调用它，所以试图关闭一个仅能接收的通道在编译时会报错。

这里我们又一次看到平方管道，这次我们使用单向通道类型：

```
gopl.io/ch8/pipeline3

func counter(out chan<- int) {
    for x := 0; x < 100; x++ {
        out <- x
    }
    close(out)
}

func squarer(out chan<- int, in <-chan int) {
    for v := range in {
        out <- v * v
    }
    close(out)
}

func printer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    go counter(naturals)
    go squarer(squares, naturals)
    printer(squares)
}
```

`counter(naturals)` 的调用隐式地将 `chan int` 类型转化为参数要求的 `chan<- int` 类型。调

用 `printer(squares)` 做了类似 `<-chan int` 的转变。在任何赋值操作中将双向通道转换为单向通道都是允许的，但是反过来是不行的，一旦有一个像 `chan<- int` 这样的单向通道，是没有办法通过它获取到引用同一个数据结构的 `chan int` 数据类型的。

8.4.4 缓冲通道

缓冲通道有一个元素队列，队列的最大长度在创建的时候通过 `make` 的容量参数来设置。下面的语句创建一个可以容纳三个字符串的缓冲通道。图 8-2 展示了 `ch` 和指向它的引用。



图 8-2 一个空的缓冲通道

```
ch = make(chan string, 3)
```

缓冲通道上的发送操作在队列的尾部插入一个元素，接收操作从队列的头部移除一个元素。如果通道满了，发送操作会阻塞所在的 goroutine 直到另一个 goroutine 对它进行接收操作来留出可用的空间。反过来，如果通道是空的，执行接收操作的 goroutine 阻塞，直到另一个 goroutine 在通道上发送数据。

可以在当前通道上无阻塞地发送三个值：

```
ch <- "A"
ch <- "B"
ch <- "C"
```

这时，通道是满的，如图 8-3 所示，第四个发送语句将会阻塞。

如果接收一个值：

```
fmt.Println(<-ch) // "A"
```

通道既不满也不空，如图 8-4 所示，所以这时一个接收或发送操作都不会阻塞。通过这种方式，通道的缓冲区将发送和接收 goroutine 进行解耦。

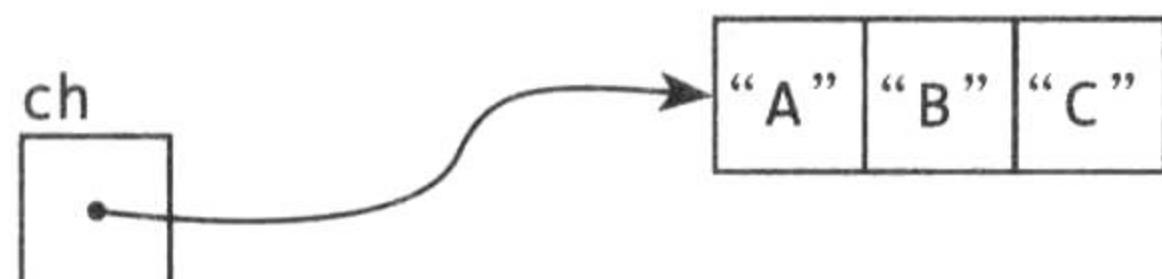


图 8-3 一个满的缓冲通道

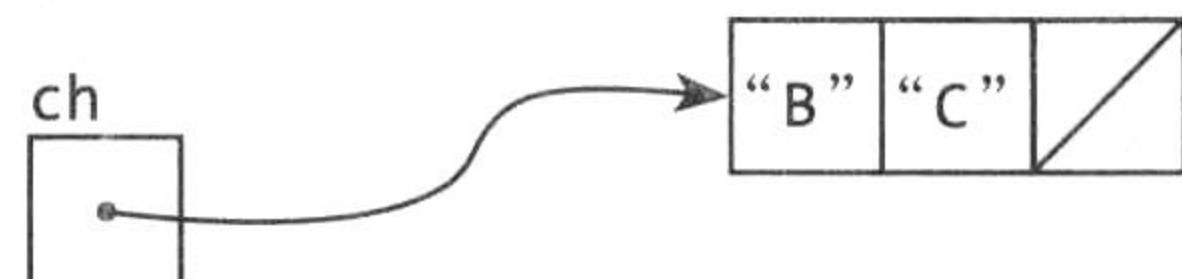


图 8-4 一个部分填满的缓冲通道

不太常见的一个情况是，程序需要知道通道缓冲区的容量，可以通过调用内置的 `cap` 函数获取它：

```
fmt.Println(cap(ch)) // "3"
```

当使用内置的 `len` 函数时，可以获取当前通道内的元素个数。因为在并发程序中这个信息会随着检索操作很快过时，所以它的价值很低，但是它在错误诊断和性能优化的时候很有用。

```
fmt.Println(len(ch)) // "2"
```

通过接下去的两次接收操作，通道又变空了，第四次接收会被阻塞：

```
fmt.Println(<-ch) // "B"
fmt.Println(<-ch) // "C"
```

这个例子中，发送和接收操作都由同一个 goroutine 执行，但在真实的程序中通常由不同的 goroutine 执行。因为语法简单，新手有时候粗暴地将缓冲通道作为队列在单个 goroutine 中使用，但是这是个错误。通道和 goroutine 的调度深度关联，如果没有另一个 goroutine 从通

道进行接收，发送者（也许是整个程序）有被永久阻塞的风险。如果仅仅需要一个简单的队列，使用 slice 创建一个就可以。

下面的例子展示一个使用缓冲通道的应用。它并发地向三个镜像地址发请求，镜像指相同但分布在不同地理区域的服务器。它将它们的响应通过一个缓冲通道进行发送，然后只接收第一个返回的响应，因为它是最早到达的。所以 `mirroredQuery` 函数甚至在两个比较慢的服务器还没有响应之前返回了一个结果。（偶然情况下，会出现像这个例子中几个 goroutine 同时在一个通道上并发发送，或者同时从一个通道接收的情况。）

```
func mirroredQuery() string {
    responses := make(chan string, 3)
    go func() { responses <- request("asia.gopl.io") }()
    go func() { responses <- request("europe.gopl.io") }()
    go func() { responses <- request("americas.gopl.io") }()
    return <-responses // return the quickest response
}
func request(hostname string) (response string) { /* ... */ }
```

如果使用一个无缓冲通道，两个比较慢的 goroutine 将被卡住，因为在它们发送响应结果到通道的时候没有 goroutine 来接收。这个情况叫作 goroutine 泄漏，它属于一个 bug。不像回收变量，泄露的 goroutine 不会自动回收，所以确保 goroutine 在不再需要的时候可以自动结束。

无缓冲和缓冲通道的选择、缓冲通道容量大小的选择，都会对程序的正确性产生影响。无缓冲通道提供强同步保障，因为每一次发送都需要和一次对应的接收同步；对于缓冲通道，这些操作则是解耦的。如果我们知道要发送的值数量的上限，通常会创建一个容量是使用上限的缓冲通道，在接收第一个值前就完成所有的发送。在内存无法提供缓冲容量的情况下，可能导致程序死锁。

通道的缓冲也可能影响程序的性能。想象蛋糕店里的三个厨师，在生产线上，在把每一个蛋糕传递给下一个厨师之前，一个烤，一个加糖衣，一个雕刻。在空间比较小的厨房，每一个厨师完成一个蛋糕流程，必须等待下一个厨师准备好接受它；这个场景类似于使用无缓冲通道来通信。

如果在厨师之间有可以放一个蛋糕的位置，一个厨师可以将制作好的蛋糕放到这里，然后立即开始制作下一个，这类似于使用一个容量为 1 的缓冲通道。只要厨师们以相同的速度工作，大多数工作就可以快速处理，消除他们各自之间的速率差异。如果在厨师之间有更多的空间——更长的缓冲区——就可以消除更大的暂态速率波动而不影响组装流水线，比如当一个厨师稍作休息时，后面再抓紧跟上进度。

另一方面，如果生产线的上游持续比下游快，缓冲区满的时间占大多数。如果后续的流程更快，缓冲区通常是空的。这时缓冲区的存在是没有价值的。

组装流水线是对于通道和 goroutine 合适的比喻。例如，如果第二段更加复杂，一个厨师可能跟不上第一个厨师的供应，或者跟不上第三个厨师的需求。为了解决这个问题，我们可以雇用另一个厨师来帮助第二段流程，独立地执行同样的任务。这个类似于创建另外一个 goroutine 使用同一个通道来通信。

这里没有空间来展示细节，但是 `gopl.io/ch8/cake` 包模拟蛋糕店，并且有几个参数可以调节。它包含了上面描述场景的一些性能基准参照（参考 11.4 节）。

8.5 并行循环

这一节探讨一些通用的并行模式，来并行执行所有的循环迭代。考虑生成一批全尺寸图像的缩略图的问题。gopl.io/ch8/thumbnail 包提供 `ImageFile` 函数，它可以缩放单个图像。这里不展示它的实现细节，它可以从 gopl.io 进行下载。

```
gopl.io/ch8/thumbnail
package thumbnail

// ImageFile 从 infile 中读取一幅图像并把它的缩略图写入同一个目录中
// 它返回生成的文件名，比如 "foo.thumb.jpg".
func ImageFile(infile string) (string, error)
```

下面的程序在一个图像文件名字列表上进行循环，然后给每一个图像产生一幅缩略图：

```
gopl.io/ch8/thumbnail
// makeThumbnails 生成指定文件的缩略图
func makeThumbnails(filenames []string) {
    for _, f := range filenames {
        if _, err := thumbnail.ImageFile(f); err != nil {
            log.Println(err)
        }
    }
}
```

很明显，处理文件的顺序没有关系，因为每一个缩放操作和其他的操作独立。像这样由一些完全独立的子问题组成的问题称为高度并行。高度并行的问题是最容易实现并行的，有许多并行机制来实现线性扩展。

并行执行这些操作，忽略文件 I/O 的延迟和对同一文件使用多个 CPU 进行图像 slice 计算。第一个并行版本准备仅仅添加 `go` 关键字。现在将忽略错误，后面再处理：

```
// 注意：不正确
func makeThumbnails2(filenames []string) {
    for _, f := range filenames {
        go thumbnail.ImageFile(f) // 注意：忽略错误
    }
}
```

这一版运行真得太快了，事实上，即使在文件名称 slice 中只有一个元素的情况下，它也比原始版本要快。如果这里没有并行机制，并发的版本怎么可能运行得更快？答案是 `makeThumbnails2` 在没有完成想要完成的事情之前就返回了。它启动了所有的 goroutine，每个文件一个，但是没有等它们执行完毕。

没有一个直接的访问等待 goroutine 结束，但是可以修改内层 goroutine，通过一个共享的通道发送事件来向外层 goroutine 报告它的完成。因为我们知道 `len(filenames)` 内层 goroutine 的确切个数，所以外层 goroutine 只需要在返回前对完成事件进行计数：

```
// makeThumbnails3 并行生成指定文件的缩略图
func makeThumbnails3(filenames []string) {
    ch := make(chan struct{})
    for _, f := range filenames {
        go func(f string) {
            thumbnail.ImageFile(f) // 注意：此处忽略了可能的错误
            ch <- struct{}{}
        }(f)
    }
}
```

```
// 等待 goroutine 完成
for range filenames {
    <-ch
}
}
```

注意，这里作为一个字面量函数的显式参数传递 `f`，而不是在 `for` 循环中声明 `f`：

```
for _, f := range filenames {
    go func() {
        thumbnail.ImageFile(f) // 注意：不正确
        // ...
    }()
}
```

回想 5.6.1 节描述的在内部匿名函数中获取循环变量的问题。上面单变量 `f` 的值被所有的匿名函数值共享并且被后续的迭代所更新。这时新的 goroutine 执行字面量函数，`for` 循环可能已经更新 `f`，并且开始另一个迭代或者已经完全结束，所以当这些 goroutine 读取 `f` 的值时，它们所看到的都是 slice 的最后一个元素。通过添加显式参数，可以确保当 `go` 语句执行的时候，使用 `f` 的当前值。

我们想让每一个工作 goroutine 中向主 goroutine 返回什么？如果调用 `thumbnail.ImageFile` 无法创建一个文件，它返回一个错误。下一个版本的 `makeThumbnails` 返回第一个它从扩展的操作中接收到的错误：

```
// makeThumbnails4 为指定文件并行地生成缩略图
// 如果任何步骤出错它返回一个错误
func makeThumbnails4(filenames []string) error {
    errors := make(chan error)
    for _, f := range filenames {
        go func(f string) {
            _, err := thumbnail.ImageFile(f)
            errors <- err
        }(f)
    }
    for range filenames {
        if err := <-errors; err != nil {
            return err // 注意：不正确，goroutine 泄漏
        }
    }
    return nil
}
```

这个函数有一个微妙的缺陷，当遇到第一个非 `nil` 的错误时，它将错误返回给调用者，这样没有 goroutine 继续从 `errors` 返回通道上进行接收，直至读完。每一个现存的工作 goroutine 在试图发送值到此通道的时候永久阻塞，永不终止。这种情况下 goroutine 泄漏（参考 8.4.4 节）可能导致整个程序卡住或者系统内存耗尽。

最简单的方案是使用一个有足够容量的缓冲通道，这样没有工作 goroutine 在发送消息时候阻塞。（另一个方案是在主 goroutine 返回第一个错误的同时，创建另一个 goroutine 来读完通道。）

下一个版本的 `makeThumbnails` 使用一个缓冲通道来返回生成的图像文件的名称以及任何错误消息：

```
// makeThumbnails5 为指定文件并行地生成缩略图
// 它以任意顺序返回生成的文件名
// 如果任何步骤出错就返回一个错误
func makeThumbnails5(filenames []string) ([]string, error) {
    type item struct {
        thumbfile string
        err        error
    }

    ch := make(chan item, len(filenames))
    for _, f := range filenames {
        go func(f string) {
            var it item
            it.thumbfile, it.err = thumbnail.ImageFile(f)
            ch <- it
        }(f)
    }

    for range filenames {
        it := <-ch
        if it.err != nil {
            return nil, it.err
        }
        thumbfiles = append(thumbfiles, it.thumbfile)
    }
}

return thumbfiles, nil
}
```

makeThumbnails 的终极版本（参见下面）返回新文件所占用的总字节数。不像前一个版本，它不是使用 slice 接收文件名，而是借助一个字符串通道，这样我们不能预测迭代的次数。

为了知道什么时候最后一个 goroutine 结束（它不一定是最后启动的），需要在每一个 goroutine 启动前递增计数，在每一个 goroutine 结束时递减计数。这需要一个特殊类型的计数器，它可以被多个 goroutine 安全地操作，然后有一个方法一直等到它变为 0。这个计数器类型是 sync.WaitGroup，下面的代码展示如何使用它：

```
// makeThumbnails6 为从通道接收到的每个文件生成缩略图
// 它返回其生成的文件占用的字节数
func makeThumbnails6(filenames <-chan string) int64 {
    sizes := make(chan int64)
    var wg sync.WaitGroup // 工作 goroutine 的个数
    for f := range filenames {
        wg.Add(1)
        // worker
        go func(f string) {
            defer wg.Done()
            thumb, err := thumbnail.ImageFile(f)
            if err != nil {
                log.Println(err)
                return
            }
            info, _ := os.Stat(thumb) // 可以忽略错误
            sizes <- info.Size()
        }(f)
    }
}
```

```
// closer
go func() {
    wg.Wait()
    close(sizes)
}()

var total int64
for size := range sizes {
    total += size
}
return total
}
```

注意 `Add` 和 `Done` 方法的不对称性。`Add` 递增计数器，它必须在工作 goroutine 开始之前执行，而不是在中间。另一方面，不能保证 `Add` 会在关闭者 goroutine 调用 `Wait` 之前发生。另外，`Add` 有一个参数，但 `Done` 没有，它等价于 `Add(-1)`。使用 `defer` 来确保在发送错误的情况下计数器可以递减。在不知道迭代次数的情况下，上面的代码结构是通用的，符合习惯的并行循环模式。

`sizes` 通道将每一个文件的大小带回主 goroutine，它使用 `range` 循环进行接收然后计算总和。注意，在关闭者 goroutine 中，在关闭 `sizes` 通道之前，等待所有的工作者结束。这里两个操作（等待和关闭）必须和在 `sizes` 通道上面的迭代并行执行。考虑替代方案：如果我们将等待操作放在循环之前的主 goroutine 中，因为通道会满，它将永不结束；如果放在循环后面，它将不可达，因为没有任何东西可用来关闭通道，循环可能永不结束。

图 8-5 说明 `makeThumbnails6` 函数中的事件序列。垂直线表示 goroutine。细片段表示休眠，粗片段表示活动。斜箭头表示 goroutine 通过事件进行了同步。时间从上向下流动。注意，主 goroutine 把大多数时间花在 `range` 循环休眠上，等待工作者发送值或等待 `closer` 来关闭通道。

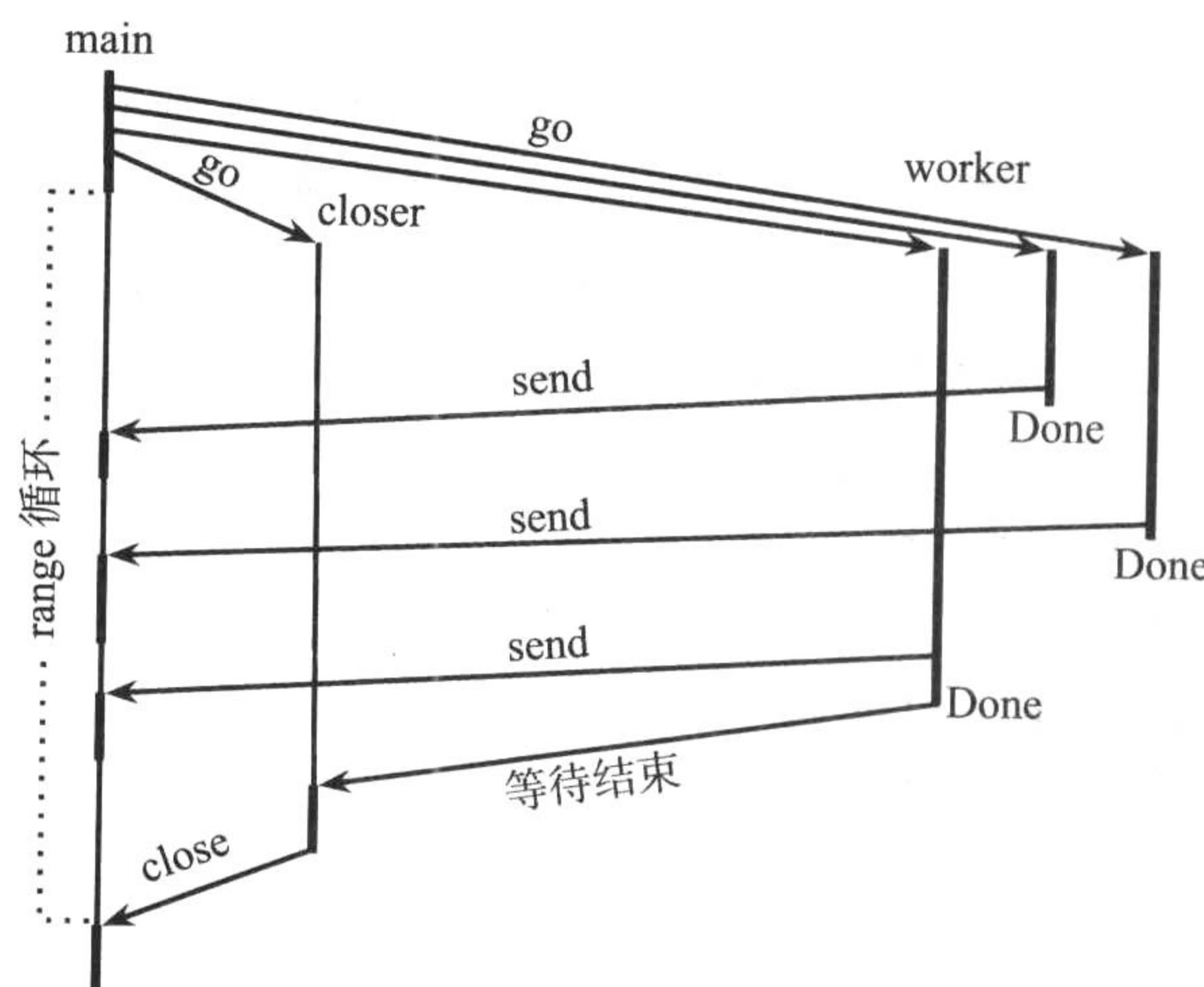


图 8-5 `makeThumbnails6` 中的事件序列

练习 8.4：修改 `reverb2` 程序来使用 `sync.WaitGroup` 来计算每一个连接上面的活动的回声 goroutine 的个数。当它变成 0 时，关闭练习 8.3 中描述的写半边的 TCP 链接。验证你修改好的 `netcat3` 客户端，等待最后几个并发的呼喊回声，即使标准输入已经关闭。

练习 8.5：使用一个已有的 CPU 绑定的顺序程序，例如 3.3 节的 `Mandelbrot` 程序，或者 3.2 节的 3D 平面计算，在主循环中并行执行它们，使用通道来通信。在多 CPU 的机器上它的运行速度有多快？goroutine 的最优数量是多少？

第一条消息比较令人意外，因为它报告的是对一个可靠的域名出现了解析失败。接下去的错误消息说明程序同时创建了太多的网络连接，超过了程序能打开文件数的限制，导致类似于 DNS 查询和 `net.Dial` 的连接失败。

程序的并行度太高了，无限制的并行通常不是一个好的主意，因为系统中总有限制因素，例如，对于计算型应用 CPU 的核数，对于磁盘 I/O 操作磁头和磁盘的个数，下载流所使用的网络带宽，或者 Web 服务本身的容量。解决方法是根据资源可用情况限制并发的个数，以匹配合适的并行度。该例子中有一个简单的办法是确保对于 `links.Extract` 的同时调用不超过 n 个，即比文件描述符所规定的 20 个少得多。这种方式类似于一个拥挤夜店的门卫只有在有客人离开的时候才允许其他客人进去。

我们可以使用容量为 n 的缓冲通道来建立一个并发原语，称为计数信号量。概念上，对于缓冲通道中的 n 个空闲槽，每一个代表一个令牌，持有者可以执行。通过发送一个值到通道中来领取令牌，从通道中接收一个值来释放令牌，创建一个新的空闲槽。这保证了在没有接收操作的时候，最多同时有 n 个发送。（尽管使用已填充槽比令牌更直观，但使用空闲槽在创建通道缓冲区之后可以省掉填充的过程。）因为通道的元素类型在这里不重要，所以我们使用 `struct{}`，它所占用的空间大小是 0。

重写 `crawl` 函数，使用令牌的获取和释放操作来包括对 `links.Extract` 函数的调用，这样保证最多同时 20 个调用可以进行。保持信号量操作离它所约束的 I/O 操作越近越好——这是一个好的实践：

```
gopl.io/ch8/crawl2

// 令牌是一个计数信号量
// 确保并发请求限制在 20 个以内
var tokens = make(chan struct{}, 20)

func crawl(url string) []string {
    fmt.Println(url)
    tokens <- struct{}{} // 获得令牌
    list, err := links.Extract(url)
    <-tokens // 释放令牌

    if err != nil {
        log.Print(err)
    }
    return list
}
```

第二个问题是这个程序永远不会结束，即使它已经从初始 URL 发现了所有的可达链接（当然，你可能注意不到这个问题，除非你精心选择初始 URL 或者像练习 8.6 一样使用深度限制策略）。为了让程序终止，当任务列表为空且爬取 goroutine 都结束以后，需要从主循环退出：

```
func main() {
    worklist := make(chan []string)
    var n int // 等待发送到任务列表的数量

    // 从命令行参数开始
    n++
    go func() { worklist <- os.Args[1:] }()

    // 并发爬取 Web
    seen := make(map[string]bool)
    for ; n > 0; n-- {
        list := <-worklist
```

```

        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                n++
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}

```

这个版本中，计数器 `n` 跟踪发送到任务列表中的任务个数。每次知道一个条目被发送到任务列表时，就递增变量 `n`，第一次递增是在发送初始化命令行参数之前，第二次递增是在每次启动一个新的爬取 goroutine 的时候。主循环从 `n` 减到 0，这时再没有任务需要完成。

现在，并发爬虫的速度大约比 5.6 节中广度优先版本快 20 倍，在它完成任务的时候，应该没有错误出现，并且正确退出。

下面的程序展示一个替代方案，解决过度并发的问题。这个版本使用最初的 `crawl` 函数，它没有计数信号量，但是通过 20 个长期存活的爬虫 goroutine 来调用它，这样确保最多 20 个 HTTP 请求并发执行：

```

gopl.io/ch8/crawl3
func main() {
    worklist := make(chan []string) // 可能有重复的 URL 列表
    unseenLinks := make(chan string) // 去重后的 URL 列表

    // 向任务列表中添加命令行参数
    go func() { worklist <- os.Args[1:] }()

    // 创建 20 个爬虫 goroutine 来获取每个不可见链接
    for i := 0; i < 20; i++ {
        go func() {
            for link := range unseenLinks {
                foundLinks := crawl(link)
                go func() { worklist <- foundLinks }()
            }
        }()
    }

    // 主 goroutine 对 URL 列表进行去重
    // 并把没有爬取过的条目发送给爬虫程序
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                unseenLinks <- link
            }
        }
    }
}

```

爬取 goroutine 使用同一个通道 `unseenLinks` 进行接收。主 goroutine 负责对从任务列表接收到的条目进行去重，然后发送每一个没有爬取过的条目到 `unseenLinks` 通道，然后被爬取 goroutine 接收。

`seen map` 被限制在主 goroutine 里面，它仅仅需要被这个 goroutine 访问。与其他形式的信息隐藏一样，范围限制可以帮助我们推导程序的正确性。例如，局部变量不能在声明它的

函数之外通过名字引用；没有从函数中逃逸（见 2.3.4 节）的变量不能从函数外面访问；一个对象的封装域只能被对象自己的方法访问。所有的场景中，信息隐藏帮助限制程序不同部分之间不经意的交互。

`crawl` 发现的链接通过精心设计的 goroutine 发送到任务列表来避免死锁。

为了节省空间，不在这讨论这个例子中的终止问题。

练习 8.6：对并发爬虫添加深度限制。如果用户设置 `-depth=3`，那么仅最多通过三个链接可达的 URL 能被找到。

练习 8.7：写一个并发程序来创建一个网站的本地镜像，获取它每一个可达的页面，然后将它们写到本地磁盘上的目录。只能获取本域的页面（例如，`golang.org`）。镜像页面内的 URL 按需调整，因为它们应该引用镜像页面，而不是原始页面。

8.7 使用 select 多路复用

下面的程序对火箭发射进行倒计时。`time.Tick` 函数返回一个通道，它定期发送事件，像一个节拍器一样。每个事件的值是一个时间戳，但我们更感兴趣它能带来的东西：

```
gopl.io/ch8/countdown1
func main() {
    fmt.Println("Commencing countdown.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        <-tick
    }
    launch()
}
```

让我们通过在倒计时进行时按下回车键来取消发射过程的能力。第一步，启动一个 goroutine 试图从标准输入中读取一个字符，如果成功，发送一个值到 `abort` 通道：

```
gopl.io/ch8/countdown2
abort := make(chan struct{})
go func() {
    os.Stdin.Read(make([]byte, 1)) // 读取单个字节
    abort <- struct{}{}
}()


```

现在每一次倒计时迭代需要等待事件到达两个通道中的一个：计时器通道，前提是一切顺利（nominal）；或者中止事件前提是“异常”（anomaly）。不能只从一个通道上接收，因为哪一个操作都会在完成前阻塞。所以需要多路复用那些操作过程，为了实现这个目的，需要一个 `select` 语句：

```
select {
case <-ch1:
    // ...
case x := <-ch2:
    // ...use x...
case ch3 <- y:
    // ...
default:
    // ...
}
```

上面展示的是 `select` 语句的通用形式。像 `switch` 语句一样，它有一系列的情况和一个

可选的默认分支。每一个情况指定一次通信（在一些通道上进行发送或接收操作）和关联的一段代码块。接收表达式操作可能出现在它本身上，像第一个情况，或者在一个短变量声明中，像第二个情况；第二种形式可以让你引用所接收的值。

`select` 一直等待，直到一次通信来告知有一些情况可以执行。然后，它进行这次通信，执行此情况所对应的语句；其他的通信将不会发生。对于没有对应情况的 `select`, `select{}` 将永远等待。

让我们回到火箭发射程序。`time.After` 函数立即返回一个通道，然后启动一个新的 goroutine 在间隔指定时间后，发送一个值到它上面。下面的 `select` 语句等两个事件中第一个到达的事件，中止事件或者指示事件过去 10s 的事件。如果过了 10s 没有中止，开始发射：

```
func main() {
    // ... 创建中止通道...
    fmt.Println("Commencing countdown. Press return to abort.")
    select {
        case <-time.After(10 * time.Second):
            // 不执行任何操作
        case <-abort:
            fmt.Println("Launch aborted!")
            return
    }
    launch()
}
```

下面的例子更微妙一些。通道 `ch` 的缓冲区大小为 1，它要么是空的，要么是满的，因此只有在其中一个状况下可以执行，要么在 `i` 是偶数时发送，要么在 `i` 是奇数时接收。它总是输出 `0 2 4 6 8`:

```
ch := make(chan int, 1)
for i := 0; i < 10; i++ {
    select {
        case x := <-ch:
            fmt.Println(x) // "0" "2" "4" "6" "8"
        case ch <- i:
    }
}
```

如果多个情况同时满足，`select` 随机选择一个，这样保证每一个通道有相同的机会被选中。在前一个例子中增加缓冲区的容量，会使输出变得不可确定，因为当缓冲既不空也不满的情况，相当于 `select` 语句在扔硬币做选择。

让该发射程序输出倒计时。下面的 `select` 语句使每一次迭代使用 1s 来等待中止，但不会更长：

gopl.io/ch8/countdown3

```
func main() {
    // ... 创建中止通道...
    fmt.Println("Commencing countdown. Press return to abort.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        select {
            case <-tick:
                // 什么操作也不执行
        }
    }
}
```

```

        case <-abort:
            fmt.Println("Launch aborted!")
            return
    }
}
launch()
}

```

`time.Tick` 函数的行为很像创建一个 goroutine 在循环里面调用 `time.Sleep`，然后在它每次醒来时发送事件。当上面的倒计时函数返回时，它停止从 `tick` 通道中接收事件，但是计时器 goroutine 还在运行，徒劳地向一个没有 goroutine 在接收的通道不断发送——发生 goroutine 泄漏（参考 8.4.4 节）。

`Tick` 函数很方便使用，但是它仅仅在应用的整个生命周期中都需要时才合适。否则，我们需要使用这个模式：

```

ticker := time.NewTicker(1 * time.Second)
<-ticker.C // 从 ticker 的通道接收
ticker.Stop() // 造成 ticker 的 goroutine 终止

```

有时候我们试图在一个通道上发送或接收，但是不想在通道没有准备好的情况下被阻塞——非阻塞通信。这使用 `select` 语句也可以做到。`select` 可以有一个默认情况，它用来指定在没有其他的通信发生时可以立即执行的动作。

下面的 `select` 语句从尝试从 `abort` 通道中接收一个值，如果没有值，它什么也不做。这是一个非阻塞的接收操作；重复这个动作称为对通道轮询：

```

select {
case <-abort:
    fmt.Printf("Launch aborted!\n")
    return
default:
    // 不执行任何操作
}

```

通道的零值是 `nil`。令人惊讶的是，`nil` 通道有时候很有用。因为在 `nil` 通道上发送和接收将永远阻塞，对于 `select` 语句中的情况，如果其通道是 `nil`，它将永远不会被选择。这次让我们用 `nil` 来开启或禁用特性所对应的情况，比如超时处理或者取消操作，响应其他的输入事件或者发送事件。我们将在下一节看到这个例子。

练习 8.8： 使用 `select` 语句，给 8.3 节的回声服务器加一个超时，这样可以断开 10s 内没有任何呼叫的客户端。

8.8 示例：并发目录遍历

这一节中，我们构建一个程序，根据命令行指定的输入，报告一个或多个目录的磁盘使用情况，类似于 UNIX `du` 命令。大多数的工作由下面的 `walkDir` 函数完成，它使用 `dirents` 辅助函数来枚举目录中的条目。

gopl.io/ch8/du1

```

// walkDir 递归地遍历以 dir 为根目录的整个文件树
// 并在 fileSizes 上发送每个已找到的文件的大小
func walkDir(dir string, fileSizes chan<- int64) {

```

```
for _, entry := range dirents(dir) {
    if entry.IsDir() {
        subdir := filepath.Join(dir, entry.Name())
        walkDir(subdir, fileSizes)
    } else {
        fileSizes <- entry.Size()
    }
}

// dirents 返回 dir 目录中的条目
func dirents(dir string) []os.FileInfo {
    entries, err := ioutil.ReadDir(dir)
    if err != nil {
        fmt.Fprintf(os.Stderr, "du1: %v\n", err)
        return nil
    }
    return entries
}
```

ioutil.ReadDir 函数返回一个 os.FileInfo 类型的 slice，针对单个文件同样的信息可以通过调用 os.Stat 函数来返回。对每一个子目录，walkDir 递归调用它自己，对于每一个文件，walkDir 发送一条消息到 fileSizes 通道。消息是文件所占用的字节数。

如下所示，main 函数使用两个 goroutine。后台 goroutine 调用 walkDir 遍历命令行上指定的每一个目录，最后关闭 fileSizes 通道。主 goroutine 计算从通道中接收的文件的大小的和，最后输出总数。

```
// du1 计算目录中文件占用的磁盘空间大小
package main

import (
    "flag"
    "fmt"
    "io/ioutil"
    "os"
    "path/filepath"
)

func main() {
    // 确定初始目录
    flag.Parse()
    roots := flag.Args()
    if len(roots) == 0 {
        roots = []string{".."}
    }

    // 遍历文件树
    fileSizes := make(chan int64)
    go func() {
        for _, root := range roots {
            walkDir(root, fileSizes)
        }
        close(fileSizes)
    }()

    // 输出结果
    var nfiles, nbytes int64
    for size := range fileSizes {
        nfiles++
        nbytes += size
    }
    printDiskUsage(nfiles, nbytes)
}
```

```
func printDiskUsage(nfiles, nbytes int64) {
    fmt.Printf("%d files %.1f GB\n", nfiles, float64(nbytes)/1e9)
}
```

在输出结果前，程序等待较长时间：

```
$ go build gopl.io/ch8/du1
$ ./du1 $HOME /usr /bin /etc
213201 files 62.7 GB
```

如果程序可以通知它的进度，将会更友好。但是仅把 `printDiskUsage` 调用移动到循环内部会使它输出数千行结果。

下面这个 `du` 的变种周期性地输出总数，只有在 `-v` 标识指定的时候才输出，因为不是所有的用户都想看进度消息。后台 goroutine 依然从根部开始迭代。主 goroutine 现在使用一个计时器每 500ms 定期产生事件，使用一个 `select` 语句或者等待一个关于文件大小的消息，这时它更新总数，或者等待一个计时事件，这时它输出当前的总数。如果 `-v` 标识没有指定，`tick` 通道依然是 `nil`，它对应的情况在 `select` 中实际上被禁用。

```
gopl.io/ch8/du2
var verbose = flag.Bool("v", false, "show verbose progress messages")
func main() {
    // ...启动后台 goroutine...
    // 定期输出结果
    var tick <-chan time.Time
    if *verbose {
        tick = time.Tick(500 * time.Millisecond)
    }
    var nfiles, nbytes int64
loop:
    for {
        select {
        case size, ok := <-fileSizes:
            if !ok {
                break loop // fileSizes 关闭
            }
            nfiles++
            nbytes += size
        case <-tick:
            printDiskUsage(nfiles, nbytes)
        }
    }
    printDiskUsage(nfiles, nbytes) // 最终总数
}
```

因为这个程序没有使用 `range` 循环，所以第一个 `select` 情况必须显式判断 `fileSizes` 通道是否已经关闭，使用两个返回值的形式进行接收操作。如果通道已经关闭，程序退出循环。标签化的 `break` 语句将跳出 `select` 和 `for` 循环的逻辑；没有标签的 `break` 只能跳出 `select` 的逻辑，导致循环的下一次迭代。

程序提供给我们一个从容不迫的更新流：

```
$ go build gopl.io/ch8/du2
$ ./du2 -v $HOME /usr /bin /etc
28608 files 8.3 GB
54147 files 10.3 GB
93591 files 15.1 GB
127169 files 52.9 GB
```

```
175931 files 62.2 GB
213201 files 62.7 GB
```

但是它依然耗费太长的时间。这里没有理由不能并发调用 `walkDir` 从而充分利用磁盘系统的并行机制。第三个版本的 `du` 为每一个 `walkDir` 的调用创建一个新的 goroutine。它使用 `sync.WaitGroup`（参考 8.5 节）来为当前存活的 `walkDir` 调用计数，一个关闭者 goroutine 在计数器减为 0 的时候关闭 `fileSizes` 通道。

```
gopl.io/ch8/du3

func main() {
    // ... 确定根目录 ...

    // 并行遍历每一个文件树
    fileSizes := make(chan int64)
    var n sync.WaitGroup
    for _, root := range roots {
        n.Add(1)
        go walkDir(root, &n, fileSizes)
    }
    go func() {
        n.Wait()
        close(fileSizes)
    }()
    // ... 选择循环 ...
}

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            n.Add(1)
            subdir := filepath.Join(dir, entry.Name())
            go walkDir(subdir, n, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}
```

因为程序在高峰时创建数千个 goroutine，所以我们不得不修改 `dirents` 函数来使用计数信号量，以防止它同时打开太多的文件，就像我们在 8.6 节中为 Web 爬虫所做的：

```
// sema 是一个用于限制目录并发数的计数信号量
var sema = make(chan struct{}, 20)

// dirents 返回 directory 目录中的条目
func dirents(dir string) []os.FileInfo {
    sema <- struct{}{}           // 获取令牌
    defer func() { <-sema }() // 释放令牌
    // ...
```

尽管系统与系统之间有很多的不同，但是这个版本的速度比前一个版本快几倍。

练习 8.9：写一个 `du` 版本，它可以为每一个指定的 `root` 目录计算和定期输出各自占用的总空间。

8.9 取消

有时候我们需要让一个 goroutine 停止它当前的任务，例如，一个 Web 服务器对客户请

求处理到一半的时候客户端断开了。

一个 goroutine 无法直接终止另一个，因为这样会让所有的共享变量状态处于不确定状态。在 8.7 节的火箭发射程序中，我们给 `abort` 通道发送一个值，倒计时 goroutine 把它理解为停止自己的请求。但是怎样才能取消两个或者指定个数的 goroutine 呢？

一个可能是给 `abort` 通道发送和要取消的 goroutine 同样多的事件。如果一些 goroutine 已经自己终止了，这样计数就多了，然后发送过程会卡住。如果那些 goroutine 可以自我繁殖，数量又会太少，其中一些 goroutine 依然不知道要取消。通常，任何时刻都很难知道有多少 goroutine 正在工作。更多情况下，当一个 goroutine 从 `abort` 通道接收到值时，它利用这个值，这样其他的 goroutine 接收不到这个值。对于取消操作，我们需要一个可靠的机制在一个通道上广播一个事件，这样很多 goroutine 可以认为它发生了，然后可以看到它已经发生。

回忆一下，当一个通道关闭且已取完所有发送的值之后，接下来的接收操作立即返回，得到零值。我们可以利用它创建一个广播机制：不在通道上发送值，而是关闭它。

我们在前一节的 `du` 程序中加入取消机制。第一步，创建一个取消通道，在它上面不发送任何值，但是它的关闭表明程序需要停止它正在做的事情。也定义了一个工具函数 `cancelled`，在它被调用的时候检测或轮询取消状态。

```
gopl.io/ch8/du4
var done = make(chan struct{})

func cancelled() bool {
    select {
    case <-done:
        return true
    default:
        return false
    }
}
```

接下来，创建一个读取标准输入的 goroutine，它通常连接到终端。一旦开始读取任何输入（例如，用户按回车键）时，这个 goroutine 通过关闭 `done` 通道来广播取消事件。

```
// 当检测到输入时取消遍历
go func() {
    os.Stdin.Read(make([]byte, 1)) // 读一个字节
    close(done)
}()
```

现在我们需要让 goroutine 来响应取消操作。在主 goroutine 中，添加第三个情况到 `select` 语句中，它尝试从 `done` 通道接收。如果选择这个情况，函数将返回，但是在返回之前它必须耗尽 `fileSizes` 通道，丢弃它所有的值，直到通道关闭。做这些是为了保证所有的 `walkDir` 调用可以执行完，不会卡在向 `fileSizes` 通道发送消息上。

```
for {
    select {
    case <-done:
        // 耗尽 fileSizes 以允许已有的 goroutine 结束
        for range fileSizes {
            // 不执行任何操作
        }
        return
    }
```

```

    case size, ok := <-fileSizes:
        // ...
    }
}

```

`walkDir` goroutine 在开始的时候轮询取消状态，如果设置状态，什么都不做立即返回。它让在取消后创建的 goroutine 什么不做：

```

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    if cancelled() {
        return
    }
    for _, entry := range dirents(dir) {
        // ...
    }
}

```

在 `walkDir` 循环中来进行取消状态轮询也许是划算的，它避免在取消后创建新的 goroutine。取消需要权衡：更快的响应通常需要更多的程序逻辑变更入侵。确保在取消事件以后没有更多昂贵的操作发生，可能需要更新代码中很多的地方，但通常我们可以通过在少量重要的地方检查取消状态来达到目的。

程序的一点性能剖析揭示了它的瓶颈在于 `dirents` 中获取信号量令牌的操作。下面的 `select` 让取消操作的延迟从数百毫秒减为几十毫秒：

```

func dirents(dir string) []os.FileInfo {
    select {
    case sema <- struct{}{}: // 获取令牌
    case <-done:
        return nil // 取消
    }
    defer func() { <-sema }()
    // ...read directory...
}

```

现在，当取消事件发生时，所有的后台 goroutine 迅速停止，然后 `main` 函数返回。当然，当 `main` 返回时，程序随之退出，不过这里没有谁在后面通知 `main` 函数来进行清理。在测试中有一个技巧：如果在取消事件到来的时候 `main` 函数没有返回，执行一个 `panic` 调用，然后运行时将转储程序中所有 goroutine 的栈。如果主 goroutine 是最后一个剩下的 goroutine，它需要自己进行清理。但如果还有其他的 goroutine 存活，它们可能还没有合适地取消，或者它们已经取消，可是需要的时间比较长；多一点调查总是值得的。崩溃转储信息通常含有足够的信息来分辨这些情况。

练习 8.10：HTTP 请求可以通过关闭 `http.Request` 结构中可选的 `cancel` 通道进行取消。修改 8.6 节的网页爬虫使其支持取消操作。

提示：`http.Get` 便利函数没有提供定制 `Request` 的机会。使用 `http.NewRequest` 创建请求，设置它的 `Cancel` 字段，然后调用 `http.DefaultClient.Do(req)` 来执行请求。

练习 8.11：使用 8.4.4 节的 `mirroredQuery` 程序中的方法，实现 `fetch` 的一个变种，它并发请求多个 URL。当第一个响应返回的时候，取消其他的请求。

8.10 示例：聊天服务器

我们用聊天服务器来结束本章，它可以在几个用户之间相互广播文本消息。这个程序里有4个goroutine。主goroutine和广播(broadcaster)goroutine，每一个连接里面有一个连接处理(handleConn)goroutine和一个客户写入(clientWriter)goroutine。广播器(broadcaster)是关于如何使用select的一个规范说明，因为它需要对三种不同的消息进行响应。

如下所示，主goroutine的工作是监听端口，接受连接客户端的网络连接。对每一个连接，它创建一个新的handleConn goroutine，就像本章开始时并发回声服务器中那样。

```
gopl.io/ch8/chat

func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }

    go broadcaster()
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err)
            continue
        }
        go handleConn(conn)
    }
}
```

下一个广播器，它使用局部变量clients来记录当前连接的客户集合。每个客户唯一被记录的信息是其对外发送消息通道的ID，下面是细节：

```
type client chan<- string // 对外发送消息的通道

var (
    entering = make(chan client)
    leaving = make(chan client)
    messages = make(chan string) // 所有接收的客户消息
)

func broadcaster() {
    clients := make(map[client]bool) // 所有连接的客户端
    for {
        select {
        case msg := <-messages:
            // 把所有接收的消息广播给所有的客户
            // 发送消息通道
            for cli := range clients {
                cli <- msg
            }

        case cli := <-entering:
            clients[cli] = true

        case cli := <-leaving:
            delete(clients, cli)
            close(cli)
        }
    }
}
```

广播者监听两个全局的通道entering和leaving，通过它们通知客户的到来和离开，如

果它从其中一个接收到事件，它将更新 `clients` 集合。如果客户离开，那么它关闭对应客户对外发送消息的通道。广播者也监听来自 `messages` 通道的事件，所有的客户都将消息发送到这个通道。当广播者接收到其中一个事件时，它把消息广播给所有客户。

现在来看一下每个客户自己的 goroutine。`handleConn` 函数创建一个对外发送消息的新通道，然后通过 `entering` 通道通知广播者新客户到来。接着，它读取客户发来的每一行文本，通过全局接收消息通道将每一行发送给广播者，发送时在每条消息前面加上发送者 ID 作为前缀。一旦从客户端读取完毕消息，`handleConn` 通过 `leaving` 通道通知客户离开，然后关闭连接。

```
func handleConn(conn net.Conn) {
    ch := make(chan string) // 对外发送客户消息的通道
    go clientWriter(conn, ch)

    who := conn.RemoteAddr().String()
    ch <- "You are " + who
    messages <- who + " has arrived"
    entering <- ch

    input := bufio.NewScanner(conn)
    for input.Scan() {
        messages <- who + ":" + input.Text()
    }
    // 注意，忽略 input.Err() 中可能的错误

    leaving <- ch
    messages <- who + " has left"
    conn.Close()
}

func clientWriter(conn net.Conn, ch <-chan string) {
    for msg := range ch {
        fmt.Fprintln(conn, msg) // 注意，忽略网络层面的错误
    }
}
```

另外，`handleConn` 函数还为每一个客户创建了写入 (`clientWriter`) goroutine，它接收消息，广播到客户的发送消息通道中，然后将它们写到客户的网络连接中。客户写入者的循环在广播者收到 `leaving` 通知并且关闭客户的发送消息通道后终止。

下面的信息展示了同一个机器上的一个服务器和两个客户端，它们使用 `netcat` 程序来聊天：

```
$ go build gopl.io/ch8/chat
$ go build gopl.io/ch8/netcat3

$ ./chat &
$ ./netcat3
You are 127.0.0.1:64208
127.0.0.1:64211 has arrived
Hi!
127.0.0.1:64208: Hi!
127.0.0.1:64211: Hi yourself.
^C
$ ./netcat3
You are 127.0.0.1:64216
127.0.0.1:64208 has left
127.0.0.1:64211 has arrived
Welcome.
```

```
127.0.0.1:64211: Welcome.  
127.0.0.1:64211: Welcome.  
^C  
127.0.0.1:64211 has left
```

当有 n 个客户 session 在连接的时候，程序并发运行着 $2n+2$ 个相互通信的 goroutine，它不需要隐式的加锁操作（参考 9.2 节）。clients map 限制在广播器这一个 goroutine 中被访问，所以不会并发访问它。唯一被多个 goroutine 共享的变量是通道以及 net.Conn 的实例，它们又都是并发安全的。关于限制、并发安全，以及跨 goroutine 的变量共享的含义，将在下一章进行更多的讨论。

练习 8.12：让广播者在每一个新客户到来的时候通知当前存在的客户。这也要求 clients 集合以及 entering 和 leaving 通道记录客户的名字。

练习 8.13：使聊天服务器可以断掉长期空置的连接，例如在过去 5 分钟里没有发送过消息的连接。提示：在另一个 goroutine 中调用 conn.Close()，可以让当前阻塞的读操作变成非阻塞，就像 input.Scan() 输入完成的读操作一样。

练习 8.14：改变聊天服务器的网络交互协议，让客户端可以输入它的名字。使用名字来代替网络地址作为发送者的 ID，作为每一条消息的前缀。

练习 8.15：任何客户程序读取数据的时间很长最终会造成所有的客户卡住。修改广播者，使它满足如果一个向客户写入的通道没有准备好接受它，那么跳过这条消息。还可以给每一个给向客户发送消息的通道增加缓冲，这样大多数的消息不会丢弃；广播者在这个通道上应该使用非阻塞的发送方式。

使用共享变量实现并发

上一章用几个程序来演示了如何使用 goroutine 和通道来实现一种直接和自然的并发方式。当然，我们避开了一些微妙的要点，这些点是写并发代码时必须铭记在心的。

本章将深入到并发机制内部，特别是与多个 goroutine 共享变量相关的问题，以及识别这些问题的分析技术，还有解决这些问题的模式。最后将解释一下 goroutine 和操作系统线程的差别。

9.1 竞态

在串行程序中（即一个程序只有一个 goroutine），程序中各个步骤的执行顺序由程序逻辑来决定。比如，在一系列语句中，第一句在第二句之前执行，以此类推。当一个程序有两个或者多个 goroutine 时，每个 goroutine 内部的各个步骤也是顺序执行的，但我们无法知道一个 goroutine 中的事件 x 和另外一个 goroutine 中的事件 y 的先后顺序。如果我们无法自信地说一个事件肯定先于另外一个事件，那么这两个事件就是并发的。

考虑一个能在串行程序中正确工作的函数。如果这个函数在并发调用时仍然能正确工作，那么这个函数是并发安全（concurrency-safe）的，在这里并发调用是指，在没有额外同步机制的情况下，从两个或者多个 goroutine 同时调用这个函数。这个概念也可以推广到其他函数，比如方法或者作用于特定类型的一些操作。如果一个类型的所有可访问方法和操作都是并发安全时，则它可称为并发安全的类型。

让一个程序并发安全并不需要其中的每一个具体类型都是并发安全的。实际上，并发安全的类型其实是特例而不是普遍存在的，所以仅在文档指出类型是安全的情况下，才可以并发地访问一个变量。对于绝大部分变量，如要回避并发访问，要么限制变量只存在于一个 goroutine 内，要么维护一个更高层的互斥不变量。本章将详细解释这些概念。

与之对应的是，导出的包级别函数通常可以认为是并发安全的。因为包级别的变量无法限制在一个 goroutine 内，所以那些修改这些变量的函数就必须采用互斥机制。

函数并发调用时不工作的原因有很多，包括死锁、活锁（livelock）^Θ以及资源耗尽。我们没有足够的时间来讨论所有的情形，因此接下来会重点讨论最重要的一种情形，即竞态。

竞态是指在多个 goroutine 按某些交错顺序执行时程序无法给出正确的结果。竞态对于程序是致命的，因为它们可能会潜伏在程序中，出现频率也很低，有可能仅在高负载环境或者在使用特定的编译器、平台和架构时才出现。这些都让竞态很难再现和分析。

我们常用一个经济损失的隐喻来解释竞态的严重性，在这里也先考虑一个简单的银行账户程序：

```
// bank 包实现了一个只有一个账户的银行
package bank
```

^Θ 比如多个线程在尝试绕开死锁，却由于过分同步导致反复冲突。——译者注

```

var balance int

func Deposit(amount int) { balance = balance + amount }

func Balance() int { return balance }

```

(`Deposit` 的函数体也可以写作等价的 `balance += amount`, 但比较长的形式解释起来比较方便。)

对于一个如此简单的程序, 我们一眼就可以看出, 任意串行地调用 `Deposit` 和 `Balance` 都可以得到正确的结果。即 `Balance` 会输出之前存入的金额总数。但如果这些函数的调用顺序不是串行而是并行, `Balance` 就不保证输出正确结果了。考虑如下两个 goroutine, 它们代表对一个共享账户的两笔交易:

```

// Alice:
go func() {
    bank.Deposit(200)           // A1
    fmt.Println("=", bank.Balance()) // A2
}()

// Bob:
go bank.Deposit(100)           // B

```

`Alice` 存入 200 美元, 然后查询她的余额, 与此同时 `Bob` 存入了 100 美元。`A1`、`A2` 两步与 `B` 是并发进行的, 我们无法预测实际的执行顺序。直觉来看, 可能存在三种不同的顺序, 分别称为“`Alice` 先”、“`Bob` 先”和“`Alice/Bob/Alice`”。下面的表格显示了每个步骤之后 `balance` 变量的值。带引号的字符串代表输出的账户余额。

<code>Alice</code> 先	<code>Bob</code> 先	<code>Alice/Bob/Alice</code>
0	0	0
A1 200	B 100	A1 200
A2 "= 200"	A1 300	B 300
B 300	A2 "= 300"	A2 "= 300"

在所有情况下最终的账户余额都是 300 美元。唯一不同的是 `Alice` 看到的账户余额是否包含了 `Bob` 的交易, 但客户对所有情况都不会有不满。

但这种直觉是错的。这里还有第四种可能, `Bob` 的存款在 `Alice` 的存款操作中间执行, 晚于账户余额读取 (`balance+amount`), 但早于余额更新 (`balance = ...`), 这会导致 `Bob` 存的钱消失了。这是因为 `Alice` 的存款操作 `A1` 实际上是串行的两个操作, 读部分和写部分, 我们称之为 `A1r` 和 `A1w`。下面就是有问题的执行顺序:

数据竞态	
	0
A1r 0	... = balance + amount
B 100	
A1w 200	balance = ...
A2 "= 200"	

在 `A1r` 之后, 表达式 `balance + amount` 求值结果为 200, 这个值在 `A1w` 步骤中用于写入, 完全没理会中间的存款操作。最终的余额为仅有 200 美元, 银行从 `Bob` 手上挣了 100 美元。

程序中的这种状况是竞态中的一种, 称为数据竞态 (data race)。数据竞态发生于两个 goroutine 并发读写同一个变量并且至少其中一个写入时。

当发生数据竞态的变量类型是大于一个机器字长的类型 (比如接口、字符串或 slice) 时, 事情就更复杂了。下面的代码并发把 `x` 更新为两个不同长度的 slice。

```

var x []int
go func() { x = make([]int, 10) }()
go func() { x = make([]int, 1000000) }()
x[999999] = 1 // 注意：未定义行为，可能造成内存异常

```

最后一个表达式中 `x` 的值是未定义的，它可能是 `nil`、一个长度为 10 的 slice 或者一个长度为 1 000 000 的 slice。回想一下 slice 的三个部分：指针、长度和容量。如果指针来自于第一个 `make` 调用而长度来自第二个 `make` 调用，那么 `x` 会变成一个嵌合体，它名义上长度为 1 000 000 但底层的数组只有 10 个元素。在这种情况下，尝试存储到第 999 999 个元素会伤及很遥远的一段内存，其恶果无法预测，问题也很难调试和定位。这种语义上的雷区称为未定义行为，C 程序员应当对此很熟悉了。幸运的是，相比之下 Go 语言很少有这种问题。

并行程序是几个串行程序交错执行这个观念也是一个错觉。在 9.4 节中可以看到，数据竞态可能由更奇怪的原因来引发。很多程序员（甚至是相当聪明的程序员）偶尔也会为自己程序中的数据竞态找借口：比如“互斥机制的成本太高了”“这段逻辑只用于输出日志”“我不在意丢掉一些消息”等。在给定的编译器和平台下不存在问题也给了他们盲目的自信。一个好的习惯是根本就没有什么温和的数据竞态。所以如何在程序中避免数据竞态呢？

再回顾一下定义（因为定义非常重要）：数据竞态发生于两个 goroutine 并发读写同一个变量并且至少其中一个时写入时。从定义不难看出，有三种方法来避免数据竞态。

第一种方法是不要修改变量。考虑如下的 map，它进行了延迟初始化，对于每个键，在第一次访问时才触发加载。如果 `Icon` 的调用是串行的，那么程序能正常工作，但如果 `Icon` 的调用是并发的，在访问 map 时就存在数据竞态。

```

var icons = make(map[string]image.Image)

func loadIcon(name string) image.Image

// 注意：并发不安全
func Icon(name string) image.Image {
    icon, ok := icons[name]
    if !ok {
        icon = loadIcon(name)
        icons[name] = icon
    }
    return icon
}

```

如果在创建其他 goroutine 之前就用完整的数据来初始化 map，并且不再修改。那么无论多少 goroutine 也可以安全地并发调用 `Icon`，因为每个 goroutine 都只读取这个 map。

```

var icons = map[string]image.Image{
    "spades.png": loadIcon("spades.png"),
    "hearts.png": loadIcon("hearts.png"),
    "diamonds.png": loadIcon("diamonds.png"),
    "clubs.png": loadIcon("clubs.png"),
}

// 并发安全
func Icon(name string) image.Image { return icons[name] }

```

在上面的例子中，`icons` 变量的赋值发生在包初始化时，也就是在程序的 `main` 函数开始运行之前。一旦初始化完成后，`icons` 就不再修改。那些从不修改的数据结构以及不可变数据结构本质上是并发安全的，也不需要做任何同步。但显然我们不能把这个方法用在必然会有更新的场景，比如一个银行账号。

第二种避免数据竞态的方法是避免从多个 goroutine 访问同一个变量。上一章的很多程序都采用了这个方法。比如，并发的 Web 爬虫（见 8.6 节）中主 goroutine 是唯一一个能访问 `seen map` 的 goroutine；聊天服务器（见 8.10 节）中的 `broadcaster goroutine` 是唯一一个能访问 `clients map` 的 goroutine。这些变量都限制在单个 goroutine 内部。

由于其他 goroutine 无法直接访问相关变量，因此它们就必须使用通道来向受限 goroutine 发送查询请求或者更新变量。这也是这句 Go 箴言的含义：“不要通过共享内存来通信，而应该通过通信来共享内存”。使用通道请求来代理一个受限变量的所有访问的 goroutine 称为该变量的监控 goroutine（monitor goroutine）。比如，`broadcaster goroutine` 监控了对 `clients map` 的访问。

下面就是重写的银行案例，用一个叫 `teller` 的监控 goroutine 限制 `balance` 变量：

```
gopl.io/ch9/bank1
// bank 包提供了一个只有一个账户的并发安全银行
package bank

var deposits = make(chan int) // 发送存款额
var balances = make(chan int) // 接收余额

func Deposit(amount int) { deposits <- amount }
func Balance() int          { return <-balances }

func teller() {
    var balance int // balance 被限制在 teller goroutine 中
    for {
        select {
        case amount := <-deposits:
            balance += amount
        case balances <- balance:
        }
    }
}

func init() {
    go teller() // 启动监控 goroutine
}
```

即使一个变量无法在整个生命周期受限于单个 goroutine，加以限制仍然可以是解决并发访问的好方法。比如一个常见的场景，可以通过借助通道来把共享变量的地址从上一步传到下一步，从而在流水线上的多个 goroutine 之间共享该变量。在流水线中的每一步，在把变量地址传给下一步后就不再访问该变量了，这样所有对这个变量的访问都是串行的。换个说法，这个变量先受限于流水线的一步，再受限于下一步，以此类推。这种受限有时也称为串行受限。

在下面的例子中，`Cakes` 是串行受限的，首先受限于 `baker` goroutine，然后受限于 `icer` goroutine。

```
type Cake struct{ state string }

func baker(cooked chan<- *Cake) {
    for {
        cake := new(Cake)
        cake.state = "cooked"
        cooked <- cake // baker 不再访问 cake 变量
    }
}
```

```

func icer(iced chan<- *Cake, cooked <-chan *Cake) {
    for cake := range cooked {
        cake.state = "iced"
        iced <- cake // icer 不再访问 cake 变量
    }
}

```

第三种避免数据竞态的办法是允许多个 goroutine 访问同一个变量，但在同一时间只有一个 goroutine 可以访问。这种方法称为互斥机制，这是下一节的主题。

练习 9.1：向 `gopl.io/ch9/bank1` 程序添加一个函数 `Withdraw(amount int) bool`。结果应当反映交易成功还是由于余额不足而失败。函数发送到监控 goroutine 的消息应当包含取款金额和一个新的通道，这个通道用于监控 goroutine 把布尔型的结果发送回 `Withdraw` 函数。

9.2 互斥锁：`sync.Mutex`

在 8.6 节，使用一个缓冲通道实现了一个计数信号量，用于确认同时发起 HTTP 请求的 goroutine 数量不超过 20。使用同样的理念，也可以用一个容量为 1 的通道来保证同一时间最多有一个 goroutine 能访问共享变量。一个计数上限为 1 的信号量称为二进制信号量 (binary semaphore)。

```

gopl.io/ch9/bank2
var (
    sema     = make(chan struct{}, 1) // 用来保护 balance 的二进制信号量
    balance int
)

func Deposit(amount int) {
    sema <- struct{}{} // 获取令牌
    balance = balance + amount
    <-sema // 释放令牌
}

func Balance() int {
    sema <- struct{}{} // 获取令牌
    b := balance
    <-sema // 释放令牌
    return b
}

```

互斥锁模式应用非常广泛，所以 `sync` 包有一个单独的 `Mutex` 类型来支持这种模式。它的 `Lock` 方法用于获取令牌 (token，此过程也称为上锁)，`Unlock` 方法用于释放令牌：

```

gopl.io/ch9/bank3
import "sync"

var (
    mu      sync.Mutex // 保护 balance
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}

func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
    return b
}

```

一个 goroutine 在每次访问银行的变量（此处仅有 `balance`）之前，它都必须先调用互斥量的 `Lock` 方法来获取一个互斥锁。如果其他 goroutine 已经取走了互斥锁，那么操作会一直阻塞到其他 goroutine 调用 `Unlock` 之后（此时互斥锁再度可用）。互斥量保护共享变量。按照惯例，被互斥量保护的变量声明应当紧接在互斥量的声明之后。如果实际情况不是如此，请确认已加了注释来说明此事。

在 `Lock` 和 `Unlock` 之间的代码，可以自由地读取和修改共享变量，这一部分称为临界区域。在锁的持有人调用 `Unlock` 之前，其他 goroutine 不能获取锁。所以很重要的一点是，goroutine 在使用完成后就应当释放锁，另外，需要包括函数的所有分支，特别是错误分支。

上面的银行程序展现了一个典型的并发模式。几个导出函数封装了一个或多个变量，于是只能通过这些函数来访问这些变量（对于一个对象的变量，则用方法来封装）。每个函数在开始时申请一个互斥锁，在结束时再释放掉，通过这种方式来确保共享变量不会被并发访问。这种函数、互斥锁、变量的组合方式称为监控（monitor）模式。（之前在监控 goroutine 中也使用了监控（monitor）这个词，都代表使用一个代理人（broker）来确保变量按顺序访问。）

因为 `Deposit` 和 `Balance` 函数中的临界区域都很短（只有一行，也没有分支），所以直接在函数结束时调用 `Unlock` 也很方便。在更复杂的临界场景中，特别是必须通过提前返回来处理错误的场景，很难确定在所有的分支中 `Lock` 和 `Unlock` 都成对执行了。Go 语言的 `defer` 语句就可以解决这个问题：通过延迟执行 `Unlock` 就可以把临界区域隐式扩展到当前函数的结尾，避免了必须在一个或者多个远离 `Lock` 的位置插入一条 `unlock` 语句。

```
func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}
```

在上面的例子中，`Unlock` 在 `return` 语句已经读完 `balance` 变量之后执行，所以 `Balance` 函数就是并发安全的。另外，我们也不需要使用局部变量 `b` 了。

而且，在临界区域崩溃时延迟执行的 `Unlock` 也会正确执行，这在使用 `recover`（参考 5.10 节）的情况下尤其重要。当然，`defer` 的执行成本比显式调用 `Unlock` 略大一些，但不足以成为代码不清晰的理由。在处理并发程序时，永远应当优先考虑清晰度，并且拒绝过早优化。在可以使用的地方，就尽量使用 `defer` 来让临界区域扩展到函数结尾处。

考虑如下 `Withdraw` 函数。当成功时，余额减少了指定的数量，并且返回 `true`，但如果余额不足，无法完成交易，`Withdraw` 恢复余额并且返回 `false`。

```
// 注意：不是原子操作
func Withdraw(amount int) bool {
    Deposit(-amount)
    if Balance() < 0 {
        Deposit(amount)
        return false // 余额不足
    }
    return true
}
```

这个函数最终能给出正确的结果，但它有一个不良的副作用。在尝试进行超额提款时，在某个瞬间余额会降到 0 以下。这有可能会导致一个小额的取款会不合逻辑地被拒绝掉。所

以当 Bob 尝试购买一辆跑车时，却会导致 Alice 无法支付早上的咖啡。`Withdraw` 的问题在于不是原子操作：它包含三个串行的操作，每个操作都申请并释放了互斥锁，但对于整个序列没有上锁。

理想情况下，`Withdraw` 应当为整个操作申请一次互斥锁。但如下尝试是正确的：

```
// 注意：不正确的实现
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    Deposit(-amount)
    if Balance() < 0 {
        Deposit(amount)
        return false // 余额不足
    }
    return true
}
```

`Deposit` 会通过调用 `mu.Lock()` 来尝试再次获取互斥锁，但由于互斥锁是不能再入的（无法对一个已经上锁的互斥量再上锁），因此这会导致死锁，`Withdraw` 会一直被卡住。

Go 语言的互斥量是不可再入的，具体理由见后。互斥量的目的是在程序执行过程中维持基于共享变量的特定不变量（invariant）。其中一个不变量是“没有 goroutine 正在访问这个共享变量”，但有可能互斥量也保护针对数据结构的其他不变量。当 goroutine 获取一个互斥锁的时候，它可能会假定这些不变量是满足的。当它获取到互斥锁之后，它可能会更新共享变量的值，这样可能会临时不满足之前的不变量。当它释放互斥锁时，它必须保证之前的不变量已经还原且又能重新满足。尽管一个可重入的互斥量可以确保没有其他 goroutine 可以访问共享变量，但是无法保护这些变量的其他不变量。

一个常见的解决方案是把 `Deposit` 这样的函数拆分为两部分：一个不导出的函数 `deposit`，它假定已经获得互斥锁，并完成实际的业务逻辑；以及一个导出的函数 `Deposit`，它用来获取锁并调用 `deposit`。这样我们就可以用 `deposit` 来实现 `Withdraw`：

```
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    deposit(-amount)
    if balance < 0 {
        deposit(amount)
        return false // 余额不足
    }
    return true
}

func Deposit(amount int) {
    mu.Lock()
    defer mu.Unlock()
    deposit(amount)
}

func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}

// 这个函数要求已获取互斥锁
func deposit(amount int) { balance += amount }
```

当然，这里的 `deposit` 函数代码太少了，所以实际上 `Withdraw` 函数可以不用调用这个函数，但无论如何通过这个例子我们很好地演示了这个规则。

封装（参考 6.6 节）即通过在程序中减少对数据结构的非预期交互，来帮助我们保证数据结构中的不变量。因为类似的原因，封装也可以用来保持并发中的不变性。所以无论是为了保护包级别的变量，还是结构中的字段，当你使用一个互斥量时，都请确保互斥量本身以及被保护的变量都没有导出。

9.3 读写互斥锁：`sync.RWMutex`

Bob 的 100 美元存款消失了，没留下任何线索，Bob 感到很焦虑，为了解决这个问题，Bob 写了一个程序，每秒钟查询数百次他的账户余额。这个程序同时在他家里、公司里和他的手机上运行。银行注意到快速增长的业务请求正在拖慢存款和取款操作，因为所有的 `Balance` 请求都是串行运行的，持有互斥锁并暂时妨碍了其他 goroutine 运行。

因为 `Balance` 函数只须读取变量的状态，所以多个 `Balance` 请求其实可以安全地并发运行，只要 `Deposit` 和 `Withdraw` 请求没有同时运行即可。在这种场景下，我们需要一种特殊类型的锁，它允许只读操作可以并发执行，但写操作需要获得完全独享的访问权限。这种锁称为多读单写锁，Go 语言中的 `sync.RWMutex` 可以提供这种功能：

```
var mu sync.RWMutex
var balance int

func Balance() int {
    mu.RLock() // 读锁
    defer mu.RUnlock()
    return balance
}
```

`Balance` 函数现在可以调用 `RLock` 和 `RUnlock` 方法来分别获取和释放一个读锁（也称为共享锁）。`Deposit` 函数无须更改，它通过调用 `mu.Lock` 和 `mu.Unlock` 来分别获取和释放一个写锁（也称为互斥锁）。

经过上面的修改之后，Bob 的绝大部分 `Balance` 请求可以并行运行且能更快完成。因此，锁可用的时间比例会更大，`Deposit` 请求也能得到更及时的响应。

`RLock` 仅可用于在临界区域内对共享变量无写操作的情形。一般来讲，我们不应当假定那些逻辑上只读的函数和方法不会更新一些变量。比如，一个看起来只是简单访问器的方法可能会递增内部使用的计数器，或者更新一个缓存来让重复的调用更快。如果你有疑问，那么久应当使用独享版本的 `Lock`。

仅在绝大部分 goroutine 都在获取读锁并且锁竞争比较激烈时（即，goroutine 一般都需要等待后才能获到锁），`RWMutex` 才有优势。因为 `RWMutex` 需要更复杂的内部簿记工作，所以在竞争不激烈时它比普通的互斥锁慢。

9.4 内存同步

你可能会对 `Balance` 方法也需要互斥锁（不管是基于通道的锁还是基于互斥量的锁）感到奇怪。毕竟，与 `Deposit` 不一样，它只包含单个操作，所以并不存在另外一个 goroutine 插在中间执行的风险。其实需要互斥锁的原因有两个。第一个是防止 `Balance` 插到其他操作中间也是很重要的，比如 `Withdraw`。第二个原因更微妙，因为同步不仅涉及多个 goroutine 的

执行顺序问题，同步还会影响到内存。

现代的计算机一般都会有多个处理器，每个处理器都有内存的本地缓存。为了提高效率，对内存的写入是缓存在每个处理器中的，只在必要时才刷回内存。甚至刷回内存的顺序都可能与 goroutine 的写入顺序不一致。像通道通信或者互斥锁操作这样的同步原语都会导致处理器把累积的写操作刷回内存并提交，所以这个时刻之前 goroutine 的执行结果就保证了对运行在其他处理器的 goroutine 可见。

考虑如下代码片段的可能输出：

```
var x, y int
go func() {
    x = 1           // A1
    fmt.Println("y:", y, " ") // A2
}()
go func() {
    y = 1           // B1
    fmt.Println("x:", x, " ") // B2
}()
```

由于这两个 goroutine 并发运行且在没使用互斥锁的情况下访问共享变量，所以这里会有数据竞态。于是我们对程序每次的输出不一样不应该感到奇怪。根据对程序中标注语句不同的交错模式，我们可能会期望能看到如下四个结果中的一个：

```
y:0 x:1
x:0 y:1
x:1 y:1
y:1 x:1
```

第四行可以由 A1,B1,A2,B2 或 B1,A1,A2,B2 这样的执行顺序来产生。但是，程序产生的如下两个输出就在我们的意料之外了：

```
x:0 y:0
y:0 x:0
```

但在某些特定的编译器、CPU 或者其他情况下，这些确实可能发生。上面四个语句以什么样的顺序交错执行才能解释这个结果呢？

在单个 goroutine 内，每个语句的效果保证按照执行的顺序发生，也就是说，goroutine 是串行一致的 (sequentially consistent)。但在缺乏使用通道或者互斥量来显式同步的情况下，并不能保证所有的 goroutine 看到的事件顺序都是一致的。尽管 goroutine A 肯定能在读取 y 之前能观察到 x=1 的效果，但它并不一定能观察到 goroutine B 对 y 写入的效果，所以 A 可能会输出 y 的一个过期值。

尽管很容易把并发简单理解为多个 goroutine 中语句的某种交错执行方式，但正如上面的例子所显示的，这并不是一个现代编译器和 CPU 的工作方式。因为赋值和 Print 对应不同的变量，所以编译器就可能会认为两个语句的执行顺序不会影响结果，然后就交换了这两个语句的执行顺序。CPU 也有类似的问题，如果两个 goroutine 在不同的 CPU 上执行，每个 CPU 都有自己的缓存，那么一个 goroutine 的写入操作在同步到内存之前对另外一个 goroutine 的 Print 语句是不可见的。

这些并发问题都可以通过采用简单、成熟的模式来避免，即在可能的情况下，把变量限制到单个 goroutine 中，对于其他变量，使用互斥锁。

9.5 延迟初始化: sync.Once

延迟一个昂贵的初始化步骤到有实际需求的时刻是一个很好的实践。预先初始化一个变量会增加程序的启动延时，并且如果实际执行时有可能根本用不上这个变量，那么初始化也不是必需的。回到本章之前提到的 `icons` 变量：

```
var icons map[string]image.Image
```

这个版本的 `Icon` 使用了延迟初始化：

```
func loadIcons() {
    icons = map[string]image.Image{
        "spades.png": loadIcon("spades.png"),
        "hearts.png": loadIcon("hearts.png"),
        "diamonds.png": loadIcon("diamonds.png"),
        "clubs.png": loadIcon("clubs.png"),
    }
}

// 注意：并发不安全
func Icon(name string) image.Image {
    if icons == nil {
        loadIcons() // 一次性地初始化
    }
    return icons[name]
}
```

对于那些只被一个 `goroutine` 访问的变量，上面的模式是没有问题的，但对于这个例子，在并发调用 `Icon` 时这个模式就是不安全的。类似于银行例子中最早版本的 `Deposit` 函数，`Icon` 也包含多个步骤：检测 `icons` 是否为空，再加载图标，最后更新 `icons` 为一个非 `nil` 值。直觉可能会告诉你，竞态带来的最严重问题可能就是 `loadIcons` 函数会被调用多遍。当第一个 `goroutine` 正忙于加载图标时，其他 `goroutine` 进入 `Icon` 函数，会发现 `icons` 仍然是 `nil`，所以仍然会调用 `loadIcons`。

但这个直觉仍然是错的（我希望你现在已经有一个关于并发的新直觉，那就是关于并发的直觉都不可靠）。回想一下 9.4 节关于内存的讨论，在缺乏显式同步的情况下，编译器和 CPU 在能保证每个 `goroutine` 都满足串行一致性的基础上可以自由地重排访问内存的顺序。`loadIcons` 一个可能的语句重排结果如下所示。它在填充数据之前把一个空 `map` 赋给 `icons`：

```
func loadIcons() {
    icons = make(map[string]image.Image)
    icons["spades.png"] = loadIcon("spades.png")
    icons["hearts.png"] = loadIcon("hearts.png")
    icons["diamonds.png"] = loadIcon("diamonds.png")
    icons["clubs.png"] = loadIcon("clubs.png")
}
```

因此，一个 `goroutine` 发现 `icons` 不是 `nil` 并不意味着变量的初始化肯定已经完成。

保证所有 `goroutine` 都能观察到 `loadIcons` 效果最简单的正确方法就是用一个互斥锁来做同步：

```
var mu sync.Mutex // 保护 icons
var icons map[string]image.Image
```

```
// 并发安全
func Icon(name string) image.Image {
    mu.Lock()
    defer mu.Unlock()
    if icons == nil {
        loadIcons()
    }
    return icons[name]
}
```

采用互斥锁访问 `icons` 的额外代价是两个 goroutine 不能并发访问这个变量，即使在变量已经安全完成初始化且不再更改的情况下，也会造成这个后果。使用一个可以并发读的锁就可以改善这个问题：

```
var mu sync.RWMutex // 保护 icons
var icons map[string]image.Image

// 并发安全
func Icon(name string) image.Image {
    mu.RLock()
    if icons != nil {
        icon := icons[name]
        mu.RUnlock()
        return icon
    }
    mu.RUnlock()

    // 获取互斥锁
    mu.Lock()
    if icons == nil { // 注意：必须重新检查 nil 值
        loadIcons()
    }
    icon := icons[name]
    mu.Unlock()
    return icon
}
```

这里有两个临界区域。goroutine 首先获取一个读锁，查阅 map，然后释放这个读锁。如果条目能找到（常见情况），就返回它。如果条目没找到，goroutine 再获取一个写锁。由于不先释放一个共享锁就无法直接把它升级到互斥锁，为了避免在过渡期其他 goroutine 已经初始化了 `icons`，所以我们必须重新检查 `nil` 值。

上面的模式具有更好的并发性，但它更复杂并且更容易出错。幸运的是，`sync` 包提供了针对一次性初始化问题的特化解决方案：`sync.Once`。从概念上来讲，`Once` 包含一个布尔变量和一个互斥量，布尔变量记录初始化是否已经完成，互斥量则负责保护这个布尔变量和客户端的数据结构。`Once` 的唯一方法 `Do` 以初始化函数作为它的参数。让我们看一下 `Once` 简化后的 `Icon` 函数：

```
var loadIconsOnce sync.Once
var icons map[string]image.Image

// 并发安全
func Icon(name string) image.Image {
    loadIconsOnce.Do(loadIcons)
    return icons[name]
}
```

每次调用 `Do(loadIcons)` 时会先锁定互斥量并检查里边的布尔变量。在第一次调用时，

这个布尔变量为假，`Do` 会调用 `loadIcons` 然后把变量设置为真。后续的调用相当于空操作，只是通过互斥量的同步来保证 `loadIcons` 对内存产生的效果（在这里就是 `icons` 变量）对所有的 goroutine 可见。以这种方式来使用 `sync.Once`，可以避免变量在正确构造之前就被其他 goroutine 分享。

练习 9.2：重写 2.6.2 节的 `PopCount` 示例，使用 `sync.Once` 来把查找表的初始化延迟到第一次使用时。（从实际效果来看，像 `PopCount` 这种既小又经高度优化的函数无法承担同步的成本。）

9.6 竞态检测器

即使以最大努力的仔细，仍然很容易在并发上犯错误。幸运的是，Go 语言运行时和工具链装备了一个精致并易于使用的动态分析工具：竞态检测器（race detector）。

简单地把 `-race` 命令行参数加到 `go build`、`go run`、`go test` 命令里边即可使用该功能。它会让编译器为你的应用或测试构建一个修改后的版本，这个版本有额外的手法用于高效记录在执行时对共享变量的所有访问，以及读写这些变量的 goroutine 标识。除此之外，修改后的版本还会记录所有的同步事件，包括 `go` 语句、通道操作、`(*sync.Mutex).Lock` 调用、`(*sync.WaitGroup).Wait` 调用等。（完整的同步事件集合可以在语言规范中的“*The Go Memory Model*”文档中找到。）

竞态检测器会研究事件流，找到那些有问题的案例，即一个 goroutine 写入一个变量后，中间没有任何同步的操作，就有另外一个 goroutine 读写了该变量。这种案例表明有对共享变量的并发访问，即数据竞态。这个工具会输出一份报告，包括变量的标识以及读写 goroutine 当时的调用栈。通常情况下这些信息足以定位问题了。在 9.7 节就有一个竞态检测器的示例。

竞态检测器报告所有实际运行了的数据竞态。然而，它只能检测到那些在运行时发生的竞态，无法用来保证肯定不会发生竞态。为了获得最佳效果，请确保你的测试包含了并发使用包的场景。

由于存在额外的簿记工作，带竞态检测功能的程序在执行时需要更长的时间和更多的内存，但即使对于很多生产环境的任务，这种额外开支也是可以接受的。对于那些不常发生的竞态，使用竞态检测器可以帮你节省数小时甚至数天的调试时间。

9.7 示例：并发非阻塞缓存

在本节中，我们会创建一个并发非阻塞的缓存系统，它能解决在并发实战很常见但已有的库也不能很好地解决的一个问题：函数记忆（memoizing）[⊖] 问题，即缓存函数的结果，达到多次调用但只须计算一次的效果。我们的解决方案将是并发安全的，并且要避免简单地对整个缓存使用单个锁而带来的锁争夺问题。

我们将使用下面的 `httpGetBody` 函数作为示例来演示函数记忆。它会发起一个 HTTP GET 请求并读取响应体。调用这个函数相当昂贵，所以我们希望避免不必要的重复调用。

```
func httpGetBody(url string) (interface{}, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
}
```

[⊖] 关于函数记忆的详细信息，可以参考 <https://en.wikipedia.org/wiki/Memoization>。——译者注

```

    defer resp.Body.Close()
    return ioutil.ReadAll(resp.Body)
}

```

最后一行略有一些微妙，`ReadAll` 返回两个结果，一个`[]byte` 和一个`error`，因为它们分别可以直接赋给 `httpGetBody` 声明的结果类型 `interface{}` 和一个`error`，所以我们可以直接返回这个结果而不用做额外的处理。`httpGetBody` 选择这样的结果类型是为了满足我们要做的缓存系统的设计。

下面是缓存的初始版本：

```

gopl.io/ch9/memo1
// memo包提供了一个对类型 Func 并发不安全的函数记忆功能
package memo

// Memo缓存了调用 Func 的结果
type Memo struct {
    f      Func
    cache map[string]result
}

// Func是用于记忆的函数类型
type Func func(key string) (interface{}, error)

type result struct {
    value interface{}
    err   error
}

func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]result)}
}

// 注意：非并发安全
func (memo *Memo) Get(key string) (interface{}, error) {
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    return res.value, res.err
}

```

一个 `Memo` 实例包含了被记忆的函数 `f`（类型为 `Func`），以及缓存，类型为从字符串到 `result` 的一个映射表。每个 `result` 都是调用 `f` 产生的结果对：一个值和一个错误。在设计的推进过程中我们会展示 `Memo` 的几种变体，但所有变体都会遵守这些基本概念。

下面的例子展示如何使用 `Memo`。对于一串请求 URL 中的每个元素，首先调用 `Get`，记录延时和它返回的数据长度：

```

m := memo.New(httpGetBody)
for url := range incomingURLs() {
    start := time.Now()
    value, err := m.Get(url)
    if err != nil {
        log.Println(err)
    }
    fmt.Printf("%s, %s, %d bytes\n",
        url, time.Since(start), len(value.([]byte)))
}

```

我们可以使用 `testing` 包（这是第 11 章的主题）来系统地调查一下记忆的效果。从下面的测试结果来看，我们可以看到 URL 流有重复项，尽管每个 URL 第一次调用 `(*Memo).Get` 都会消耗数百毫秒的时间，但对这个 URL 的第二次请求在 $1\mu\text{s}$ 内就返回了同样的结果。

```
$ go test -v gopl.io/ch9/memo1
== RUN Test
https://golang.org, 175.026418ms, 7537 bytes
https://godoc.org, 172.686825ms, 6878 bytes
https://play.golang.org, 115.762377ms, 5767 bytes
http://gopl.io, 749.887242ms, 2856 bytes
https://golang.org, 721ns, 7537 bytes
https://godoc.org, 152ns, 6878 bytes
https://play.golang.org, 205ns, 5767 bytes
http://gopl.io, 326ns, 2856 bytes
--- PASS: Test (1.21s)
PASS
ok gopl.io/ch9/memo1 1.257s
```

这次测试中所有的 `Get` 都是串行运行的。

因为 HTTP 请求用并发来改善的空间很大，所以我们修改测试来让所有请求并发进行。这个测试使用 `sync.WaitGroup` 来做到等最后一个请求完成后再返回的效果。

```
m := memo.New(httpGetBody)
var n sync.WaitGroup
for url := range incomingURLs() {
    n.Add(1)
    go func(url string) {
        start := time.Now()
        value, err := m.Get(url)
        if err != nil {
            log.Println(err)
        }
        fmt.Printf("%s, %s, %d bytes\n",
            url, time.Since(start), len(value.([]byte)))
        n.Done()
    }(url)
}
n.Wait()
```

这次的测试运行起来快很多，但是它并不是每一次都能正常运行。我们可能注意到意料之外的缓存无效，以及缓存命中后返回错误的结果，甚至崩溃。

更糟糕的是，有的时候它能正常运行，所以我们可能甚至都没有注意到它会有问题。但如果我们在加上 `-race` 标志后再运行，那么竞态检测器（参考 9.6 节）经常会输出与下面类似的一份报告：

```
$ go test -run=TestConcurrent -race -v gopl.io/ch9/memo1
== RUN TestConcurrent
...
WARNING: DATA RACE
Write by goroutine 36:
    runtime.mapassign1()
    ~/go/src/runtime/hashmap.go:411 +0x0
gopl.io/ch9/memo1.(*Memo).Get()
    ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
...
Previous write by goroutine 35:
    runtime.mapassign1()
    ~/go/src/runtime/hashmap.go:411 +0x0
```

```

gopl.io/ch9/memo1.(*Memo).Get()
~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
...
Found 1 data race(s)
FAIL    gopl.io/ch9/memo1    2.393s

```

上面提到的 memo.go:32 告诉我们两个 goroutine 在没使用同步的情况下更新了 cache map。整个 Get 函数其实不是并发安全的：它存在数据竞态。

```

28 func (memo *Memo) Get(key string) (interface{}, error) {
29     res, ok := memo.cache[key]
30     if !ok {
31         res.value, res.err = memo.f(key)
32         memo.cache[key] = res
33     }
34     return res.value, res.err
35 }

```

让缓存并发安全最简单的方法就是用一个基于监控的同步机制。我们需要的是给 Memo 加一个互斥量，并在 Get 函数的开头获取互斥锁，在返回前释放互斥锁，这样两个 cache 相关的操作就发生在临界区域了：

```

gopl.io/ch9/memo2
type Memo struct {
    f      Func
    mu    sync.Mutex // 保护cache
    cache map[string]result
}

// Get 是并发安全的
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    memo.mu.Unlock()
    return res.value, res.err
}

```

现在即使并发运行测试，竞态检测器也没有报警。但是这次对 Memo 的修改让我们之前对性能的优化失效了。由于每次调用 f 时都上锁，因此 Get 把我们希望并行的 I/O 操作串行化了。我们需要的是一个非阻塞的缓存，一个不会把他需要记忆的函数串行运行的缓存。

在下面一个版本的 Get 实现中，主调 goroutine 会分两次获取锁：第一次用于查询，第二次用于在查询无返回结果时进行更新。在两次之间，其他 goroutine 也可以使用缓存。

```

gopl.io/ch9/memo3
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    memo.mu.Unlock()

```

```

    if !ok {
        res.value, res.err = memo.f(key)

        // 在两个临界区域之前，可能会有多个 goroutine 来计算 f(key) 并且
        // 更新 map
        memo.mu.Lock()
        memo.cache[key] = res
        memo.mu.Unlock()
    }
    return res.value, res.err
}

```

性能再度得到提升，但我们注意到某些 URL 被获取了两次。在两个或者多个 goroutine 几乎同时调用 Get 来获取同一个 URL 时就会出现这个问题。两个 goroutine 都首先查询缓存，发现缓存中没有需要的数据，然后调用那个慢函数 f，最后又都用获得的结果来更新 map，其中一个结果会被另外一个覆盖。

在理想情况下我们应该避免这种额外的处理。这个功能有时称为重复抑制 (duplicate suppression)。在下面的 Memo 版本中，map 的每个元素是一个指向 entry 结构的指针。除了与之前一样包含一个已经记住的函数 f 调用结果之外，每个 entry 还新加了一个通道 ready。在设置 entry 的 result 字段后，通道会关闭，正在等待的 goroutine 会收到广播 (参考 8.9 节)，然后就可以从 entry 读取结果了。

```

gopl.io/ch9/memo4

type entry struct {
    res  result
    ready chan struct{} // res 准备好后会被关闭
}

func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]*entry)}
}

type Memo struct {
    f      Func
    mu    sync.Mutex // 保护 cache
    cache map[string]*entry
}

func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    e := memo.cache[key]
    if e == nil {
        // 对 key 的第一次访问，这个 goroutine 负责计算数据和广播数据
        // 已准备完毕的消息
        e = &entry{ready: make(chan struct{})}
        memo.cache[key] = e
        memo.mu.Unlock()

        e.res.value, e.res.err = memo.f(key)

        close(e.ready) // 广播数据已准备完毕的消息
    } else {
        // 对这个 key 的重复访问
        memo.mu.Unlock()

        <-e.ready // 等待数据准备完毕
    }
    return e.res.value, e.res.err
}

```

现在调用 Get 回会先获取保护 cache map 的互斥锁，再从 map 中查询一个指向已有 entry 的指针，如果没有查找到，就分配并插入一个新的 entry，最后释放锁。如果要查询的

`entry` 存在，那么它的值可能还没准备好（另外一个 goroutine 有可能还在调用慢函数 `f`），所以主调 goroutine 就需要等待 `entry` 准备好才能读取 `entry` 中的 `result` 数据，具体的实现方法就是从 `ready` 通道读取数据，这个操作会一直阻塞到通道关闭。

如果要查询的 `entry` 不存在，那么当前的 goroutine 就需要新插入一个没有准备好的 `entry` 到 `map` 里，并负责调用慢函数 `f`，更新 `entry`，最后向其他正在等待的 goroutine 广播数据已准备完毕的消息。

注意，`entry` 中的变量 `e.res.value` 和 `e.res.err` 被多个 goroutine 共享。创建 `entry` 的 goroutine 设置了这两个变量的值，其他 goroutine 在收到数据准备完毕的广播后开始读这两个变量。尽管变量被多个 goroutine 访问，但此处不需要加上互斥锁。`ready` 通道的关闭先于其他 goroutine 收到广播事件，所以第一个 goroutine 的变量写入事件也先于后续多个 goroutine 的读取事件。在这个情况下数据竞态不存在。

这里的并发、重复抑制、非阻塞缓存就完成了。

上面的 `Memo` 代码使用一个互斥量来保护被多个调用 `Get` 的 goroutine 访问的 `map` 变量。接下来会对比另外一种设计，在新的设计中，`map` 变量限制在一个监控 goroutine 中，而 `Get` 的调用者则不得不改为发送消息。

`Func`、`result`、`entry` 的声明与之前一致：

```
// Func是用于记忆的函数类型
type Func func(key string) (interface{}, error)

// result是调用 Func 的返回结果
type result struct {
    value interface{}
    err   error
}

type entry struct {
    res  result
    ready chan struct{} // 当 res 准备好后关闭该通道
}
```

尽管 `Get` 的调用者通过这个通道来与监控 goroutine 通信，但是 `Memo` 类型现在包含一个通道 `requests`。该通道的元素类型是 `request`。通过这种数据结构，`Get` 的调用者向监控 goroutine 发送被记忆函数的参数（`key`），以及一个通道 `response`，结果在准备好后就通过 `response` 通道发回。这个通道仅会传输一个值。

gopl.io/ch9/memo5

```
// request是一条请求消息，key 需要用 Func 来调用
type request struct {
    key      string
    response chan<- result // 客户端需要单个 result
}

type Memo struct{ requests chan request }

// New返回f的函数记忆，客户端之后需要调用 Close.
func New(f Func) *Memo {
    memo := &Memo{requests: make(chan request)}
    go memo.server(f)
    return memo
}

func (memo *Memo) Get(key string) (interface{}, error) {
    response := make(chan result)
    memo.requests <- request{key, response}
    res := <-response
    return res.value, res.err
}

func (memo *Memo) Close() { close(memo.requests) }
```

上面的 `Get` 方法创建了一个响应 (`response`) 通道，放在了请求里边，然后把它发送给监控 goroutine，再马上从响应通道中读取。

如下所示，`cache` 变量被限制在监控 goroutine (即 `(*Memo).server`) 中。监控 goroutine 从 `request` 通道中循环读取，直到该通道被 `Close` 方法关闭。对于每个请求，它先查询缓存，如果没找到则创建并插入一个新的 `entry`。

```
func (memo *Memo) server(f Func) {
    cache := make(map[string]*entry)
    for req := range memo.requests {
        e := cache[req.key]
        if e == nil {
            // 对这个 key 的第一次请求
            e = &entry{ready: make(chan struct{})}
            cache[req.key] = e
            go e.call(f, req.key) // 调用 f(key)
        }
        go e.deliver(req.response)
    }
}

func (e *entry) call(f Func, key string) {
    // 执行函数
    e.res.value, e.res.err = f(key)
    // 通知数据已准备完毕
    close(e.ready)
}

func (e *entry) deliver(response chan<- result) {
    // 等该数据准备完毕
    <-e.ready
    // 向客户端发送结果
    response <- e.res
}
```

与基于互斥锁的版本类似，对于指定键的一次请求负责在该键上调用函数 `f`，保存结果到 `entry` 中，最后通过关闭 `ready` 通道来广播准备完毕状态。这个流程通过 `(*entry).call` 来实现。

对同一个键的后续请求会在 `map` 中找到已有的 `entry`，然后等待结果准备好，最后通过响应通道把结果发回给调用 `Get` 的客户端 goroutine。其中 `call` 和 `deliver` 方法都需要在独立的 goroutine 中运行，以确保监控 goroutine 能持续处理新请求。

上面的例子展示了可以使用两种方案来构建并发结构：共享变量并上锁，或者通信顺序进程 (communicating sequential process)，这两者也都不复杂。

在给定的情况下也许很难判定哪种方案更好，但了解这两种方案的对照关系是很有价值的。有时候从一种方案切换到另外一种能够让代码更简单。

练习 9.3：扩展 `Func` 类型和 `(*Memo).Get` 方法，让调用者可选择性地提供一个 `done` 通道，方便取消操作 (参考 8.9 节)。不要缓存被取消的 `Func` 调用结果。

9.8 goroutine 与线程

上一章提到可以先忽略 goroutine 与操作系统 (OS) 线程的差异。尽管它们之间的差异本质上是属于量变，但一个足够大的量变会变成质变。下面讨论如何区分它们。

9.8.1 可增长的栈

每个 OS 线程都有一个固定大小的栈内存（通常为 2MB），栈内存区域用于保存在其他函数调用期间那些正在执行或临时暂停的函数中的局部变量。这个固定的栈大小既太大又太小。对于一个小的 goroutine，2MB 的栈是一个巨大的浪费，比如有的 goroutine 仅仅等待一个 `WaitGroup` 再关闭一个通道。在 Go 程序中，一次创建十万左右的 goroutine 也不罕见，对于这种情况，栈就太大了。另外，对于最复杂和深度递归的函数，固定大小的栈始终不够大。改变这个固定大小可以提高空间效率并允许创建更多的线程，或者也可以容许更深的递归函数，但无法同时做到上面的两点。

作为对比，一个 goroutine 在生命周期开始时只有一个很小的栈，典型情况下为 2KB。与 OS 线程类似，goroutine 的栈也用于存放那些正在执行或临时暂停的函数中的局部变量。但与 OS 线程不同的是，goroutine 的栈不是固定大小的，它可以按需增大和缩小。goroutine 的栈大小限制可以达到 1GB，比线程典型的固定大小栈高几个数量级。当然，只有极少的 goroutine 会使用这么大的栈。

练习 9.4： 使用通道构造一个把任意多个 goroutine 串联在一起的流水线程序。在内存耗尽之前你能创建的最大流水线级数是多少？一个值穿过整个流水线需要多久？

9.8.2 goroutine 调度

OS 线程由 OS 内核来调度。每隔几毫秒，一个硬件时钟中断发到 CPU，CPU 调用一个叫调度器的内核函数。这个函数暂停当前正在运行的线程，把它的寄存器信息保存到内存，查看线程列表并决定接下来运行哪一个线程，再从内存恢复线程的注册表信息，最后继续执行选中的线程。因为 OS 线程由内核来调度，所以控制权限从一个线程到另外一个线程需要一个完整的上下文切换（context switch）：即保存一个线程的状态到内存，再恢复另外一个线程的状态，最后更新调度器的数据结构。考虑这个操作涉及的内存局部性以及涉及的内存访问数量，还有访问内存所需的 CPU 周期数量的增加，这个操作其实是很慢的。

Go 运行时包含一个自己的调度器，这个调度器使用一个称为 *m:n* 调度的技术（因为它可以复用 / 调度 *m* 个 goroutine 到 *n* 个 OS 线程）。Go 调度器与内核调度器的工作类似，但 Go 调度器只需关心单个 Go 程序的 goroutine 调度问题。

与操作系统的线程调度器不同的是，Go 调度器不是由硬件时钟来定期触发的，而是由特定的 Go 语言结构来触发的。比如当一个 goroutine 调用 `time.Sleep` 或被通道阻塞或对互斥量操作时，调度器就会将这个 goroutine 设为休眠模式，并运行其他 goroutine 直到前一个可重新唤醒为止。因为它不需要切换到内核语境，所以调用一个 goroutine 比调度一个线程成本低很多。

练习 9.5： 写一个程序，两个 goroutine 通过两个无缓冲通道来互相转发消息。这个程序能维持每秒多少次通信？

9.8.3 GOMAXPROCS

Go 调度器使用 `GOMAXPROCS` 参数来确定需要使用多少个 OS 线程来同时执行 Go 代码。默认值是机器上的 CPU 数量，所以在一个有 8 个 CPU 的机器上，调度器会把 Go 代码同时调度到 8 各 OS 线程上。（`GOMAXPROCS` 是 *m:n* 调度中的 *n*。）正在休眠或者正被通道通信阻塞的 goroutine 不需要占用线程。阻塞在 I/O 和其他系统调用中或调用非 Go 语言写的函数的

goroutine 需要一个独立的 OS 线程，但这个线程不计算在 `GOMAXPROCS` 内。

可以用 `GOMAXPROCS` 环境变量或者 `runtime.GOMAXPROCS` 函数来显式控制这个参数。可以用一个小程序来看看 `GOMAXPROCS` 的效果，这个程序无止境地输出 0 和 1：

```
for {
    go fmt.Println(0)
    fmt.Println(1)
}

$ GOMAXPROCS=1 go run hacker-cliché.go
1111111111111111111100000000000000000000011111...
$ GOMAXPROCS=2 go run hacker-cliché.go
0101010101010101011001100101011010010100110...
```

在第一次运行时，每次最多只能有一个 goroutine 运行。最开始是主 goroutine，它输出 1。在一段时间以后，Go 调度器让主 goroutine 休眠，并且唤醒另一个输出 0 的 goroutine，让它有机会执行。在第二次运行时，这里有两个可用的 OS 线程，所以两个 goroutine 可以同时运行，以一个差不多的速率输出两个数字。我们必须强调影响 goroutine 调度的因素很多，运行时也在不断演化，所以你的结果可能与上面展示的结果会有所不同。

练习 9.6： 测量计算密集型并行程序（见练习 8.5）在 `GOMAXPROCS` 参数变化时的性能变化。在你的计算机上最优值是多少？你的计算机有多少个 CPU？

9.8.4 goroutine 没有标识

在大部分支持多线程的操作系统和编程语言里，当前线程都有一个独特的标识，它通常可以取一个整数或者指针。这个特性让我们可以轻松构建一个线程的局部存储，它本质上就是一个全局的 map，以线程的标识作为键，这样每个线程都可以独立地用这个 map 存储和获取值，而不受其他线程干扰。

goroutine 没有可供程序员访问的标识。这个是由设计来决定的，因为线程局部存储有一种被滥用的倾向。比如，当一个 Web 服务器用一个支持线程局部存储的语言来实现时，很多函数都会通过访问这个存储来查找关于 HTTP 请求的信息。但就像那些过度依赖于全局变量的程序一样，这也会导致一种不健康的“超距作用”，即函数的行为不仅取决于它的参数，还取决于运行它的线程标识。因此，在线程的标识需要改变的场景（比如需要使用工作线程时），这些函数的行为就会变得诡异莫测。

Go 语言鼓励一种更简单的编程风格，其中，能影响一个函数行为的参数应当是显式指定的。这不仅让程序更易阅读，还让我们能自由地把一个函数的子任务分发到多个不同的 goroutine 而无需担心这些 goroutine 的标识。

你现在已经学习了写 Go 程序所需的所有语言特性。在接下来的两章中，我们将回退一步，从一个更大的尺度去了解支撑编程的一些实践和工具：比如如何把一个项目分为多个包，如何获取、编译、测试、归档、分享这些包，以及对这些包进行基准测试、性能分析。

包和 go 工具

今天一个中等规模的程序可能包含 10 000 个函数。但是作者只须思考它们其中的一部分，甚至不需要设计函数，因为绝大部分都是其他人来写的，然后通过包来复用。

Go 自带 100 多个包，可以为大多数应用程序提供基础。Go 社区是一个茁壮成长的生态环境，其中鼓励包设计、共享、重用以及改进，已经发布了很多的包，可以在 <http://godoc.org> 找到可以搜索的索引。本章展示如何使用已有的包和创建新包。

Go 还有配套的 `go` 工具，一个复杂但是容易使用的命令行工具，用来管理 Go 包的工作空间。本书开篇展示了如何使用 `go` 工具来下载、构建、运行样例程序。本章讨论这个工具所隐含的概念，展示它更多的功能，其中包括输出文档和在包的工作空间中查询包的元数据。下一章探索它的测试特性。

10.1 引言

任何包管理系统的目的是通过对关联的特性进行分类，组织成便于理解和修改的单元，使其与程序的其他包保持独立，从而有助于设计和维护大型的程序。模块化允许包在不同的项目中共享、复用，在组织中发布，或者在全世界范围内使用。

每个包定义了一个不同的命名空间作为它的标识符。每个名字关联一个具体的包，它让我们在为类型、函数等选取短小而且清晰的名字的同时，不与程序的其他部分冲突。

包通过控制名字是否导出使其对包外可见来提供封装能力。限制包成员的可见性，从而隐藏 API 后面的辅助函数和类型，允许包的维护者修改包的实现而不影响包外部的代码。限制变量的可见性也可以隐藏变量，这样使用者仅可以通过导出函数来对其访问和更新，他们可以保留自己的不变量以及在并发程序中实现互斥的访问。

当我们修改一个文件时，我们必须重新编译文件所在的包和所有潜在依赖它的包。众所周知，Go 程序的编译比其他语言要快，即便从零开始编译也如此。这里有三个主要原因。第一，所有的导入都必须在每一个源文件的开头进行显式列出，这样编译器在确定依赖性的时候就不需要读取和处理整个文件；第二，包的依赖性形成有向无环图，因为没有环，所以包可以独立甚至并行编译。第三，Go 包编译输出的目标文件不仅记录它自己的导出信息，还记录它所依赖包的导出信息。当编译一个包时，编译器必须从每一个导入中读取一个目标文件，但是不会超出这些文件。

10.2 导入路径

每一个包都通过一个唯一的字符串进行标识，它称为导入路径，它们用在 `import` 声明中。

```
import (
    "fmt"
    "math/rand"
    "encoding/json"
    "golang.org/x/net/html"
    "github.com/go-sql-driver/mysql"
)
```

如 2.6.1 节所述，Go 语言的规范没有定义字符串的含义或如何确定一个包的导入路径，它通过工具来解决这些问题。本章详细讨论 `go` 工具如何理解它们，`go` 工具也是 Go 程序员用来构建、测试程序的主要工具，尽管还有其他工具存在。例如，Go 程序员使用 Google 内部的多语言构建系统，遵循不同的命名和包定位规则，具体化的测试案例等，这更加匹配那个系统的惯例。

对于准备共享或公开的包，导入路径需要全局唯一。为了避免冲突，除了标准库中的包之外，其他包的导入路径应该以互联网域名（组织机构拥有的域名或用于存放包的域名）作为路径开始，这样也方便查找包。例如上面例子导入 Go 团队维护的一个 HTML 解析器和一个流行的第三方 MySQL 数据库驱动程序。

10.3 包的声明

在每一个 Go 源文件的开头都需要进行包声明。主要的目的是当该包被其他包引入的时候作为其默认的标识符（称为包名）。

例如，`math/rand` 包中每一个文件的开头都是 `package rand`，这样当你导入这个包时，可以访问它的成员，比如 `rand.Int`、`rand.Float64` 等。

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println(rand.Int())
}
```

通常，包名是导入路径的最后一段，于是，即使导入路径不同的两个包，二者也可以拥有同样的名字。例如，两个包的导入路径分别是 `math/rand` 和 `crypto/rand`，而包的名字都是 `rand`。我们将看到如何在同一个程序中同时使用它们。

关于“最后一段”的惯例，这个有三个例外。第一个例外是：不管包的导入路径是什么，如果该包定义一条命令（可执行的 Go 程序），那么它总是使用名称 `main`。这是告诉 `go build`（见 10.7.3 节）的信号，它必须调用连接器生成可执行文件。

第二个例外是：目录中可能有一些文件名字以 `_test.go` 结尾，包名中会出现以 `_test` 结尾。这样一个目录中有两个包：一个普通的，加上一个外部测试包。`_test` 后缀告诉 `go test` 两个包都需要构建，并且指明文件属于哪个包。外部测试包用来避免测试所依赖的导入图中的循环依赖。11.2.4 节会进行更细致的讲述。

第三个例外是：有一些依赖管理工具会在包导入路径的尾部追加版本号后缀，如 `"gopkg.in/yaml.v2"`。包名不包含后缀，因此这个情况下包名为 `yaml`。

10.4 导入声明

一个 Go 源文件可以在 `package` 声明的后面和第一个非导入声明语句前面紧接着包含零个或多个 `import` 声明。每一个导入可以单独指定一条导入路径，也可以通过圆括号括起来的列表一次导入多个包。下面两种形式是等价的，但第二种形式更常见。

```
import "fmt"
import "os"

import (
    "fmt"
    "os"
)
```

导入的包可以通过空行进行分组；这类分组通常表示不同领域和方面的包。导入顺序不重要，但按照惯例每一组都按照字母进行排序。（`gofmt` 和 `goimports` 工具都会自动进行分组并排序。）

```
import (
    "fmt"
    "html/template"
    "os"

    "golang.org/x/net/html"
    "golang.org/x/net/ipv4"
)
```

如果需要把两个名字一样的包（如 `math/rand` 和 `crypto/rand`）导入到第三个包中，导入声明就必须至少为其中的一个指定一个替代名字来避免冲突。这叫作重命名导入。

```
import (
    "crypto/rand"
    mrand "math/rand" // 通过指定一个不同的名称 mrand 就避免了冲突
)
```

替代名字仅影响当前文件。其他文件（即便是同一个包中的文件）可以使用默认名字来导入包，或者一个替代名字也可以。

重命名导入在没有冲突时也是非常有用的。如果有时用到自动生成的代码，导入的包名字非常冗长，使用一个替代名字可能更方便。同样的缩写名字要一直用下去，以避免产生混淆。使用一个替代名字有助于规避常见的局部变量冲突。例如，如果一个文件可以包含许多以 `path` 命名的变量，我们就可以使用 `pathpkg` 这个名字导入一个标准的 "`path`" 包。

每个导入声明从当前包向导入的包建立一个依赖。如果这些依赖形成一个循环，`go build` 工具会报错。

10.5 空导入

如果导入的包的名字没有在文件中引用，就会产生一个编译错误。但是，有时候，我们必须导入一个包，这仅仅是为了利用其副作用：对包级别的变量执行初始化表达式求值，并执行它的 `init` 函数（见 2.6.2 节）。为了防止“未使用的导入”错误，我们必须使用一个重命名导入，它使用一个替代的名字 `_`，这表示导入的内容为空白标识符。通常情况下，空白标识不可能被引用。

```
import _ "image/png" // 注册 PNG 解码器
```

这称为空白导入。多数情况下，它用来实现一个编译时的机制，使用空白引用导入额外

的包，来开启主程序中可选的特性。首先我们来看如何使用它，然后看它是如何工作的。

标准库的 `image` 包导出了 `Decode` 函数，它从 `io.Reader` 读取数据，并且识别使用哪一种图像格式来编码数据，调用适当的解码器，返回 `image.Image` 对象作为结果。使用 `image.Decode` 可以构建一个简单的图像转换器，读取某一种格式的图像，然后输出为另外一个格式：

```
gopl.io/ch10/jpeg
// jpeg 命令从标准输入读取 PNG 图像
// 并把它作为 JPEG 图像写到标准输出
package main

import (
    "fmt"
    "image"
    "image/jpeg"
    _ "image/png" // 注册 PNG 解码器
    "io"
    "os"
)

func main() {
    if err := toJPEG(os.Stdin, os.Stdout); err != nil {
        fmt.Fprintf(os.Stderr, "jpeg: %v\n", err)
        os.Exit(1)
    }
}

func toJPEG(in io.Reader, out io.Writer) error {
    img, kind, err := image.Decode(in)
    if err != nil {
        return err
    }
    fmt.Println(os.Stderr, "Input format =", kind)
    return jpeg.Encode(out, img, &jpeg.Options{Quality: 95})
}
```

如果将 `gopl.io/ch3/mandelbrot`（参考 3.3 节）的输出作为这个转换程序的输入，它检测 PNG 个数的输入，然后输出 JPEG 格式的图 3-3。

```
$ go build gopl.io/ch3/mandelbrot
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
Input format = png
```

注意空白导入 `image/png`。没有这一行，程序可以正常编译和链接，但是不能识别和解码 PNG 格式的输入：

```
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
jpeg: image: unknown format
```

这里解释它是如何工作的。标准库提供 GIF、PNG、JPEG 等格式的解码库，用户自己可以提供其他格式的，但是为了使可执行程序简短，除非明确需要，否则解码器不会被包含进应用程序。`image.Decode` 函数查阅一个关于支持格式的表格。每一个表项由 4 个部分组成：格式的名字；某种格式中所使用的相同的前缀字符串，用来识别编码格式；一个用来解码被编码图像的函数 `Decode`；以及另一个函数 `DecodeConfig`，它仅仅解码图像的元数据，比如尺寸和色域。对于每一种格式，通常通过在其支持的包的初始化函数中来调用 `image.RegisterFormat` 来向表格添加项，例如 `image/png` 中的实现如下：

```

package png // image/png

func Decode(r io.Reader) (image.Image, error)
func DecodeConfig(r io.Reader) (image.Config, error)

func init() {
    const pngHeader = "\x89PNG\r\n\x1a\x00"
    image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}

```

这个效果就是，一个应用只需要空白导入格式化所需的包，就可以让 `image.Decode` 函数具备应对格式的解码能力。

`database/sql` 包使用类似的机制让用户按需加入想要的数据库驱动程序。例如：

```

import (
    "database/sql"
    _ "github.com/lib/pq"           // 添加 Postgres 支持
    _ "github.com/go-sql-driver/mysql" // 添加 MySQL 支持
)

db, err = sql.Open("postgres", dbname) // OK
db, err = sql.Open("mysql", dbname)    // OK
db, err = sql.Open("sqlite3", dbname)  // 返回错误消息: unknown driver "sqlite3"

```

练习 10.1：扩展 `jpeg` 程序，使其可以把任意支持的输入格式转换为任意输出格式，使用 `image.Decode` 来检测输入格式，并且添加一个标记来选择输出格式。

练习 10.2：定义一个通用的归档文件读取函数，它可以读取 ZIP (`archive/zip`) 文件和 POSIX tar (`archive/tar`) 文件。使用一个类似前面描述的注册机制，使用空白导入以插件方式支持各种文件格式。

10.6 包及其命名

本节将提供一些建议，指出如何遵从 Go 的习惯来给包和它的成员进行命名。

当创建一个包时，使用简短的名字，但是不要短到像加了密一样。在标准库中最常用的包包括：`bufio`、`bytes`、`flag`、`fmt`、`http`、`io`、`json`、`os`、`sort`、`sync` 和 `time` 等。

尽可能保持可读性和无歧义。例如，不要把一个辅助工具包命名为 `util`，使用 `imageutil` 或 `ioutil` 等名称更具体和清晰。避免选择经常用于相关的局部变量的包名，或者迫使使用者使用重命名导入，例如使用以 `path` 命名的包。

包名通常使用统一的形式。标准包 `bytes`、`errors` 和 `strings` 使用复数来避免覆盖响应的预声明类型，使用 `go/types` 这个形式，来避免和关键字的冲突。

避免使用有其他含义的包名。例如，我们一开始在 2.5 节中使用 `temp` 作为温度转换包的名字，但是它没有继续这么用。这是一个非常糟糕的主意，因为“`temp`”大多数情况下代表“`temporary`”（临时性的）。我们在一小段时间里面使用 `temperature` 作为包名，但是它太长了，并且不能说明它究竟可以做什么。最后，它变成了 `tempconv`，它更短并且和 `strconv` 等类似。

现在讨论包成员的命名。因为对其他包成员的每一个引用使用一个具体的标识符，例如 `fmt.Println`，描述包的成员和描述包名与成员名同样繁杂。我们不需要在 `Println` 中引用格式化的概念，因为包名 `fmt` 还没有准备好。当设计一个包时，要考虑两个有意义的部分如何一起工作，而不只是成员名。这里有一些具体的例子：

`bytes.Equal` `flag.Int` `http.Get` `json.Marshal`

我们可以识别出一些通用的命名模式。`strings` 包提供一系列操作字符串的独立函数：

```

package strings

func Index(needle, haystack string) int

type Replacer struct{ /* ... */ }
func NewReplacer(oldnew ...string) *Replacer

type Reader struct{ /* ... */ }
func NewReader(s string) *Reader

```

`string` 这个词不会出现在任何名字中。客户通过 `strings.Index`、`strings.Replacer` 等引用它们。

其他的一些包可以描述为单一类型包，例如 `html/template` 和 `math/rand`，这些包导出一个数据类型及其方法，通常有一个 `New` 函数用来创建实例。

```

package rand // "math/rand"

type Rand struct{ /* ... */ }
func New(source Source) *Rand

```

这可能造成重复，例如在 `template.Template` 或 `rand.Rand` 中，这也是为什么这类包名通常都比较短。

在其他极端情况下，像 `net/http` 这样的包有很多的名字，但是没有很多的结构，因为它们执行复杂的任务。尽管有超过 20 种类型和更多的函数，但是包中最重要的成员使用最简单的命名：`Get`、`Post`、`Handle`、`Error`、`Client`、`Server`。

10.7 go 工具

下面的章节主要讨论 `go` 工具（`go tool`），它用来下载、查询、格式化、构建、测试以及安装 Go 代码包。

`go` 工具将不同种类的工具集合并为一个命名集。它是一个包管理器（类似于 `apt` 或 `rpm`），它可以查询包的作者，计算它们的依赖关系，从远程版本控制系统下载它们。它是一个构建系统，可计算文件依赖，调用编译器、汇编器和链接器，尽管它没有标准的 UNIX `make` 命令完备。它还是一个测试驱动程序，第 11 章将介绍它。

它的命令行接口使用“瑞士军刀”风格，有十几个子命令，其中有一些我们已经见过，例如 `get`、`run`、`build` 和 `fmt`。可以运行 `go help` 来查看内置文档的索引。仅仅为了引用，我们已经列出了最常用的命令：

```

$ go
...
build      compile packages and dependencies
clean      remove object files
doc        show documentation for package or symbol
env        print Go environment information
fmt        run gofmt on package sources
get        download and install packages and dependencies
install    compile and install packages and dependencies
list       list packages
run        compile and run Go program
test       test packages
version    print Go version
vet        run go tool vet on packages

Use "go help [command]" for more information about a command.
...

```

为了让配置操作最小化，`go` 工具非常依赖惯例。例如，给定一个 Go 源文件，该工具可以找到它所在的包，因为每一个目录包含一个包，并且包的导入路径对应于工作空间的目录结构。给定一个包的导入路径，该工具可以找到存放目标文件的对应目录。它也可以找到存储源代码仓库的服务器的 URL。

10.7.1 工作空间的组织

大部分用户必须进行的唯一的配置是 `GOPATH` 环境变量，它指定工作空间的根。当需要切换到不同的工作空间时，更新 `GOPATH` 变量的值即可。例如，当写这本书的时候，我们设置 `GOPATH` 为 `$HOME/gobook`：

```
$ export GOPATH=$HOME/gobook
$ go get gopl.io/...
```

在你使用上面的命令下载了本书所有的程序之后，你的工作空间将包含如下一个层次结构：

```
GOPATH/
  src/
    gopl.io/
      .git/
      ch1/
        helloworld/
          main.go
        dup/
          main.go
        ...
    ...
    golang.org/x/net/
      .git/
      html/
        parse.go
        node.go
      ...
  bin/
    helloworld
    dup
  pkg/
    darwin_amd64/
    ...
```

`GOPATH` 有三个子目录。`src` 子目录包含源文件。每一个包放在一个目录中，该目录相对于 `$GOPATH/src` 的名字是包的导入路径，如 `gopl.io/ch1/helloworld`。注意，一个 `GOPATH` 工作空间在 `src` 下包含多个源代码版本控制仓库，例如 `gopl.io` 或 `golang.org`。`pkg` 子目录是构建工具存储编译后的包的位置，`bin` 子目录放置像 `helloworld` 这样的可执行程序。

第二个环境变量是 `GOROOT`，它指定 Go 发行版的根目录，其中提供所有标准库的包。`GOROOT` 下面的目录结构类似于 `GOPATH`，这样 `fmt` 包的源代码放在 `$GOROOT/src/fmt` 目录中。用户无须设置 `GOROOT`，因为默认情况下 `go` 工具使用它的安装路径。

`go env` 命令输出与工具链相关的已经设置有效值的环境变量及其所设置值，还会输出未设置有效值的环境变量及其默认值。`GOOS` 指定目标操作系统（例如，`android`、`linux`、`darwin` 或者 `windows`），`GOARCH` 指定目标处理器架构，比如 `amd64`、`386` 或 `arm`。尽管 `GOPATH` 是为一个必须设置的变量，但是其他的变量也会偶尔在我们的解释中出现。

```
$ go env
GOPATH="/home/gopher/gobook"
GOROOT="/usr/local/go"
GOARCH="amd64"
GOOS="darwin"
...
```

10.7.2 包的下载

如果使用 `go` 工具，包的导入路径不仅指示了如何在本地工作空间中找到它的位置，还指明了通过互联网使用 `go get` 来获取和更新它的位置。

`go get` 命令可以下载单一的包，也可以使用 ... 符号来下载子树或仓库，像前一节提到的那样。该工具也计算并下载初始包所有的依赖性，这也是为什么前一个例子中 `golang.org/x/net/html` 包会出现在工作空间中。

在 `go get` 完成包的下载之后，它会构建它们，然后安装库和相应的命令。这些内容会在下一节详细讨论，一个例子将展示整个流程是如何进行的。下面的第一条命令获取 `golint` 工具，它用来检查 Go 源码中的风格问题。第二条命令执行 `golint` 来检查 2.6.2 节中的 `gopl.io/ch2/popcount` 代码。它会报告我们忘了给这个包写文档注释：

```
$ go get github.com/golang/lint/golint
$ $GOPATH/bin/golint gopl.io/ch2/popcount
src/gopl.io/ch2/popcount/main.go:1:1:
    package comment should be of the form "Package popcount ..."
```

`go get` 命令已经支持多个流行的代码托管站点，如 GitHub、Bitbucket 和 Launchpad，并且可以向版本控制系统发出合适的请求。对于不知名的网站，你也许需要指出导入路径使用的是哪种版本控制协议，比如 Git 或 Mercurial。执行 `go help importpath` 来获取更多细节。

`go get` 创建的目录是远程仓库的真实客户端，而不仅仅是文件的副本，这样可以使用版本控制命令来查看本地编辑的差异或者更新到不同的版本。例如，`golang.org/x/net` 目录是一个 Git 客户端：

```
$ cd $GOPATH/src/golang.org/x/net
$ git remote -v
origin https://go.googlesource.com/net (fetch)
origin https://go.googlesource.com/net (push)
```

注意，包导入路径中明显的域名 (`golang.org`) 不同于 Git 服务器的实际域名 (`go.googlesource.com`)。这是 `go` 工具的一个特性，如果位置由诸如 `googlesource.com` 或 `github.com` 之类通用服务托管，包可以在其导入路径中使用自定义域名。在 `https://golang.org/x/net/html` 下面的 HTML 网页包含如下元数据，它重定向 `go` 工具到实际托管地址的 Git 仓库：

```
$ go build gopl.io/ch1/fetch
$ ./fetch https://golang.org/x/net/html | grep go-import
<meta name="go-import"
      content="golang.org/x/net git https://go.googlesource.com/net">
```

如果你指定 `-u` 开关，`go get` 将确保它访问的所有包（包括它们的依赖性）更新到最新版本，然后再构建和按照。如果没有这个标记，已经存在于本地的包不会更新。

`go get -u` 命令通常获取每个包的最新版本，它在你刚刚开始的时候很方便；但是在需要部署的项目中（其中，发布版本需要精准的版本控制），就不太适合使用它。通常的解决

方案是加一层 vendor 目录，构建一个关于所有必需依赖的本地副本，然后非常小心地更新这个副本。在 Go 1.5 之前，这需要改变包的导入路径，这样 `golang.org/x/net/html` 的副本会变成 `gopl.io/vendor/golang.org/x/net/html`。几乎所有最近版本的 go 工具都支持加 vendor 目录，但这里不允许我们展开所有的细节了。请使用 `go help gopath` 来查看 vendor 目录的详细信息。

练习 10.3： 使用 `fetch http://gopl.io/ch1/helloworld?go-get=1`，找到本书的示例代码是由那个服务托管的（`go get` 发出的 HTTP 请求包含 `go-get` 参数，这样服务器可以区分出普通的浏览器请求。）

10.7.3 包的构建

`go build` 命令编译每一个命令行参数中的包。如果包是一个库，结果会被舍弃；对于没有编译错误的包几乎不做检查。如果包的名字是 `main`，`go build` 调用链接器在当前目录中创建可执行程序，可执行程序的名字取自包的导入路径的最后一段。

每一个目录包含一个包，每一个可执行程序或者 UNIX 命令都需要自己的目录。这些目录可能是 `cmd` 目录的子目录，比如 `golang.org/x/tools/cmd/godoc` 命令，为 Go 包的文档提供 Web 访问接口（参考 10.7.4 节）。

如前所述，包可以指定通过目录来指定，可以使用导入路径或者一个相对目录名，目录必须以 `.` 或 `..` 开头，即使这不经常需要。如果没有提供参数，会使用当前目录作为参数。所以，以下命令：

```
$ cd $GOPATH/src/gopl.io/ch1/helloworld  
$ go build
```

和以下命令：

```
$ cd anywhere  
$ go build gopl.io/ch1/helloworld
```

以及以下命令：

```
$ cd $GOPATH  
$ go build ./src/gopl.io/ch1/helloworld
```

构建同样的包（尽管每次写入的目录是 `go build` 命令运行时所在的目录）。但以下命令编译不同的包：

```
$ cd $GOPATH  
$ go build src/gopl.io/ch1/helloworld  
Error: cannot find package "src/gopl.io/ch1/helloworld".
```

包也可以使用一个文件列表来指定（尽管这只是针对小型的程序和一次性的实验）。如果包名是 `main`，可执行程序的名字来自第一个 `.go` 文件名的主体部分。

```
$ cat quoteargs.go  
package main  
  
import (  
    "fmt"  
    "os"  
)
```

```

func main() {
    fmt.Printf("%q\n", os.Args[1:])
}
$ go build quoteargs.go
$ ./quoteargs one "two three" four\ five
["one" "two three" "four five"]

```

特别是对于这类即用即抛型的程序，我们需要在构建之后尽快运行。`go run` 命令将这两步合并起来：

```
$ go run quoteargs.go one "two three" four\ five
["one" "two three" "four five"]
```

第一个不是以 `.go` 文件结尾的参数会作为 Go 可执行程序的参数列表的开始。

默认情况下，`go build` 命令构建所有需要的包以及它们所有的依赖性，然后丢弃除了最终可执行程序之外的所有编译后的代码。依赖性分析和编译本身都非常快，但是当项目增长到数十个包和数十万行代码的时候，重新编译依赖性的时间明显变慢，也许数秒钟的时间，即使依赖的部分根本没有改变过。

`go install` 命令和 `go build` 非常相似，区别是它会保存每一个包的编译代码和命令，而不是把它们丢弃。编译后的包保存在 `$GOPATH/pkg` 目录中，它对应于存放源文件的 `src` 目录，可执行的命令保存在 `$GOPATH/bin` 目录中。（许多用户将 `$GOPATH/bin` 加入他们的可执行搜索路径中。）这样，`go build` 和 `go install` 对于没有改变的包和命令不需要重新编译，从而使后续的构建更加快速。惯例上，`go build -i` 可以将包安装在独立于构建目标的地方。

因为编译包根据操作系统平台和 CPU 体系结构不同而不同，所以 `go install` 将保存它们的目录命名为与 `GOOS` 和 `GOARCH` 变量的值相关。例如，在 Mac 上面 `golang.org/x/net/html` 编译后的文件 `golang.org/x/net/html.a` 放在 `$GOPATH/pkg/darwin_amd64` 目录下面。

```
gopl.io/ch10/cross
func main() {
    fmt.Println(runtime.GOOS, runtime.GOARCH)
}
```

下面的命令分别生成 64 位和 32 位的可执行程序：

```

$ go build gopl.io/ch10/cross
$ ./cross
darwin amd64
$ GOARCH=386 go build gopl.io/ch10/cross
$ ./cross
darwin 386

```

例如，为了处理底层的可移植性问题或为重要的例程提供优化版本，有一些包需要为特定的平台或者处理器编译不同版本的代码。如果一个文件名包含操作系统或处理器体系结构名字（如 `net_linux.go` 或 `asm_amd64.s`），`go` 工具只会在构建指定规格的目标文件的时候才进行编译。叫作构建标签的特殊注释，提供更细粒度的控制。例如，如果一个文件包含下面的注释：

```
// +build linux darwin
```

注释在包的声明之前（它是文档注释），`go build` 只会在构建 Linux 或 Mac OS X 系统应用的时候才会对它进行编译，下面的注释指出任何时候都不要编译这个文件：

```
// +build ignore
```

更多的细节可以在 `go/build` 包的文档中的 *Build Constraints* 节找到：

```
$ go doc go/build
```

10.7.4 包的文档化

Go 风格强烈鼓励有良好的包 API 文档。每一个导出的包成员的声明以及包声明自身应该立刻使用注释来描述它的目的和用途。

Go 文档注释总是完整的语句，使用声明的包名作为开头的第一句注释通常是总结。函数参数和其他的标识符无须括起来或者特别标注。例如，`fmt.Fprintf` 的文档注释如下：

```
// Fprintf 根据格式说明符格式化并写入 w  
// 返回写入的字节数及可能遇到的错误  
func Fprintf(w io.Writer, format string, a ...interface{}) (int, error)
```

`Fprintf` 的格式化细节在 `fmt` 包自身的文档注释中进行解释。包声明的前面的文档注释被认为是整个包的文档注释。尽管它可以出现在任何文件中，但是必须只有一个。比较长的包注释可以使用一个单独的注释文件，`fmt` 的注释超过 300 行，文件名通常叫 `doc.go`。

好的文档不一定是洋洋洒洒的，而简明是文档一个不可替代的优点。事实上，Go 在文档保持简练和简单方面的惯例和其他所有的东西一样，因为文档像代码一样也需要维护。许多声明可以在一个通顺的句子中解释清楚，并且如果这个行为非常明确，就不需要注释。

在全书中篇幅允许时，就会使用 `doc` 注释来进行前置声明，但是在你浏览标准库时你会发现更好的示例。如果你想这样做，有两个工具可以给你提供更多方便。

`go doc` 工具输出在命令行上指定的内容的声明和整个文档注释，这也许是一个包：

```
$ go doc time  
package time // 导入 "time"  
  
Package time provides functionality for measuring and displaying time.  
  
const Nanosecond Duration = 1 ...  
func After(d Duration) <-chan Time  
func Sleep(d Duration)  
func Since(t Time) Duration  
func Now() Time  
type Duration int64  
type Time struct { ... }  
... 更多 ...
```

或者是一个包成员：

```
$ go doc time.Since  
func Since(t Time) Duration  
  
Since returns the time elapsed since t.  
It is shorthand for time.Now().Sub(t).
```

或者是一个方法：

```
$ go doc time.Duration.Seconds  
func (d Duration) Seconds() float64  
  
Seconds returns the duration as a floating-point number of seconds.
```

该工具不需要完整的导入路径或者正确的标识符。这个命令输出来自 `encoding/json` 包的 `(*json.Decoder).Decode` 的文档：

```
$ go doc json.decode
func (dec *Decoder) Decode(v interface{}) error

Decode reads the next JSON-encoded value from its input and stores
it in the value pointed to by v.
```

有点迷惑的是，第二个工具名字叫 `godoc`，它提供相互链接的 HTML 页面服务，进而提供不少于 `go doc` 命令的信息。在 `https://golang.org/pkg` 的 `godoc` 服务器覆盖了标准库。图 10-1 展示了 `time` 包的文档，在 11.6 节中，我们将看到 `godoc` 中交互式显示的程序示例。在 `https://godoc.org` 的 `godoc` 服务器提供数千个可搜索的开源包索引。

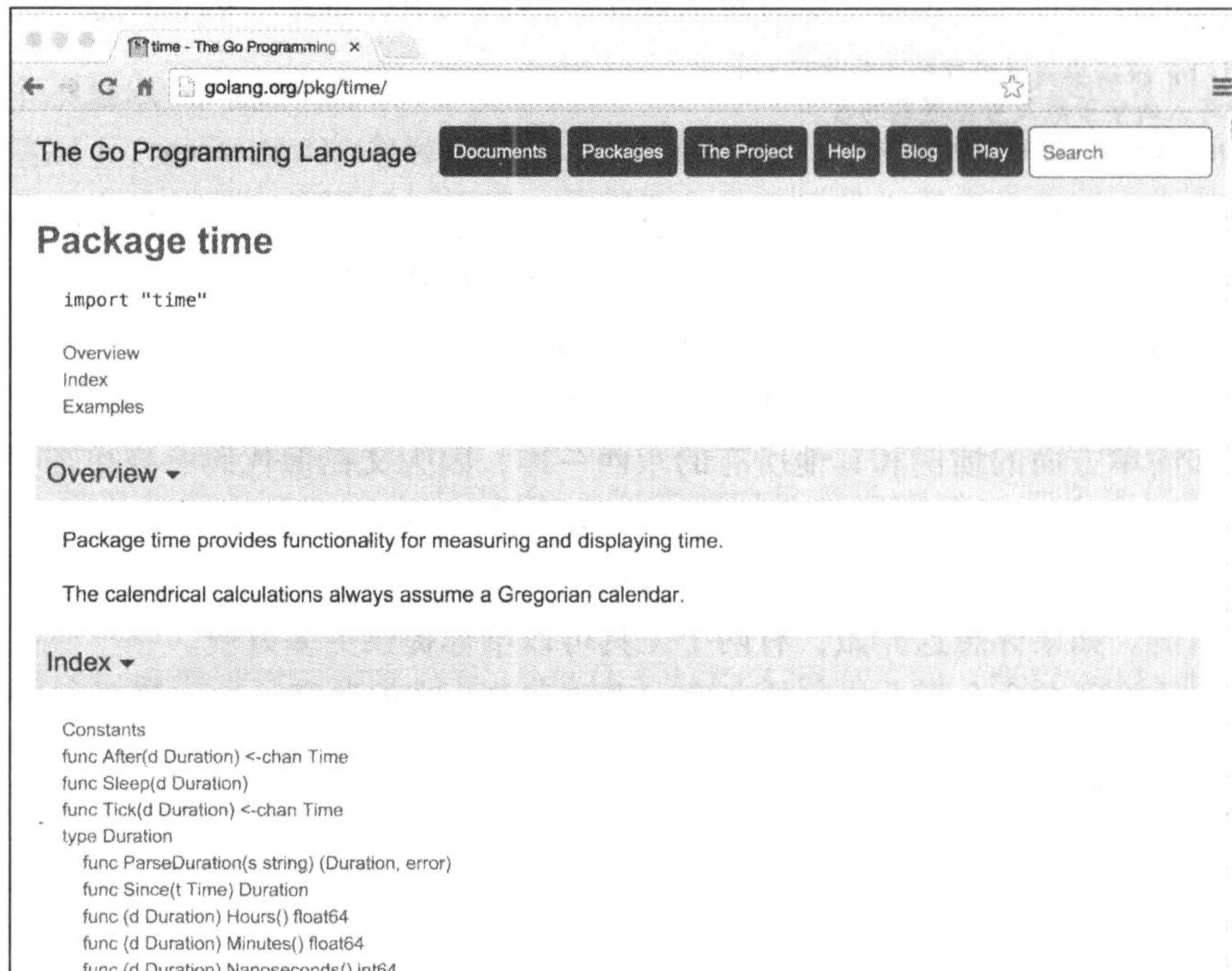


图 10-1 godoc 中的 `time` 包

如果你想浏览你自己的包，你也可以在你的工作空间中运行一个 `godoc` 实例。在执行下面的命令后，在浏览器中访问 `http://localhost:8000/pkg`：

```
$ godoc -http :8000
```

加上 `-analysis=type` 和 `-analysis=pointer` 标记使文档内容丰富，同时提供源代码的高级静态分析结果。

10.7.5 内部包

这是包用来封装 Go 程序最重要的机制。没有导出的标识符只能在同一个包内访问，导出的标识符可以在世界任何地方访问。

有时，有一个中间地带是很有帮助的，这种方式定义标识符可以被一个小的可信任的包集合访问，但不是所有人可以访问。例如，当我们将一个大包分解为多个可托管的小包时，我们不想对其他的包显露这些包之间的关系。或者我们想在不进行导出的情况下，在项目的一些包中间共享一些工具函数。或者我们只是想试验一个新的包，而不是永久地提交给它的

API，可以通过加上一个允许访问的有限客户列表来实现。

为了解决这些需求，`go build` 工具会特殊对待导入路径中包含路径片段 `internal` 的情况。这些包叫内部包。内部包只能被另一个包导入，这个包位于以 `internal` 目录的父目录为根目录的树中。例如，给定下面的包，`net/http/internal/chunked` 可以从 `net/http/httputil` 或 `net/http` 导入，但是不能从 `net/url` 进行导入。然而，`net/url` 可以导入 `net/http/httputil`。

```
net/http
net/http/internal/chunked
net/http/httputil
net/url
```

10.7.6 包的查询

`go list` 工具上报可用包的信息。通过最简单的形式，`go list` 判断一个包是否存在于工作空间中，如果存在输出它的导入路径：

```
$ go list github.com/go-sql-driver/mysql
github.com/go-sql-driver/mysql
```

`go list` 命令的参数可以包含“...”通配符，它用来匹配包的导入路径中的任意子串。可以使用它枚举一个 Go 工作空间中的所有包：

```
$ go list ...
archive/tar
archive/zip
bufio
bytes
cmd/addr2line
cmd/api
... 更多其他的包 ...
```

或者一个指定的子树中的所有包：

```
$ go list gopl.io/ch3/...
gopl.io/ch3/basename1
gopl.io/ch3/basename2
gopl.io/ch3/comma
gopl.io/ch3/mandelbrot
gopl.io/ch3/netflag
gopl.io/ch3/printints
gopl.io/ch3/surface
```

或者一个具体的主题：

```
$ go list ...xml...
encoding/xml
gopl.io/ch7/xmlselect
```

`go list` 命令获取每一个包的完整元数据，而不仅仅是导入路径，并且提供各种对于用户或者其他工具可访问的格式。`-json` 标记使 `go list` 以 JSON 格式输出每一个包的完整记录：

```
$ go list -json hash
{
  "Dir": "/home/gopher/go/src/hash",
  "ImportPath": "hash",
  "Name": "hash",
  "Doc": "Package hash provides interfaces for hash functions.",
  "Target": "/home/gopher/go/pkg/darwin_amd64/hash.a",
```

```
"Goroot": true,  
"Standard": true,  
"Root": "/home/gopher/go",  
"GoFiles": [  
    "hash.go"  
,  
"Imports": [  
    "io"  
,  
"Deps": [  
    "errors",  
    "io",  
    "runtime",  
    "sync",  
    "sync/atomic",  
    "unsafe"  
,  
]  
}
```

-f 标记可以让用户通过 `text/template` 包提供的模板语言来定制输出格式（参考 4.6 节）。这个命令输出 `strconv` 包的依赖过渡关系记录，记录之间由空格分割：

```
$ go list -f '{{join .Deps " "}}' strconv  
errors math runtime unicode/utf8 unsafe
```

这条命令输出标准库的 `compress` 子树中每个包的直接导入记录：

```
$ go list -f '{{.ImportPath}} -> {{join .Imports " "}}' compress/...  
compress/bzip2 -> bufio io sort  
compress/flate -> bufio fmt io math sort strconv  
compress/gzip -> bufio compress/flate errors fmt hash hash/crc32 io time  
compress/lzw -> bufio errors fmt io  
compress/zlib -> bufio compress/flate errors fmt hash hash/adler32 io
```

`go list` 对于一次性的交互查询和构建、测试脚本都非常有用。我们将在 11.2.4 节再次使用它。更多的参数以及它们的含义等信息，可以通过执行 `go help list` 来获取。

本章讨论了 `go` 工具中几乎所有重要的子命令（除了一个子命令外）。下一章讲述如何使用 `go test` 命令测试 Go 程序。

练习 10.4：构建一个工具，它可以汇报工作空间中所有包的过度依赖中，是否含有参数中指定的包。提示：你将需要执行两次 `go list`，一次是针对初始包，一次是针对所有包。你也许想使用 `encoding/json` 包（参考 4.5 节）来解析它的 JSON 格式输出内容。

测 试

莫里斯·威尔克斯（Maurice Wilkes）设计和制造了世界上第一台存储程序式计算机 EDSAC，在 1949 年有一次实验室爬楼梯的时候，针对测试讲述了一番颇有洞察力的话。在《一个计算机先驱的回忆》（Memoirs of A Computer Pioneer）一书中，他回忆道：“我强烈地意识到我余生的很大一部分时间都将用来寻找我程序中的错误。”当然，从那时开始，虽然或许也会对他对软件构建复杂度认识的不足感到困惑，但是每一个存储程序式计算机程序员都会赞同他这番话。

今天的软件项目比威尔克斯年代的要庞大、复杂得多，并且在使软件复杂度可以控制的技术上面，人们投入了大量的精力。其中有两种技术尤其有效，第一是软件发布之前的例行同行评审，另一个就是本章的主题：测试。

测试是自动化测试的简称，即编写简单的程序来确保程序（产品代码）在该测试中针对特定输入产生预期的输出。这些测试通常要么是经过精心设计之后用来检测某种功能，要么是随机性的，用来扩大测试的覆盖面。

软件测试领域内容很广泛。测试任务几乎占据了所有程序员的一部分时间，有时候甚至是一些程序员所有的时间。关于测试的资料有数千本书和数百万字的博文。在每一门主流的程序设计语言中，都有很多软件包专门用来构建测试，其中有一些还包含很多理论，并且这个领域吸引了很多拥有众多拥趸的先知。这些足够使得程序员相信为了写好测试，他们必须掌握一种新的技能。

Go 的测试方法看上去相对比较低级。它依赖于命令 `go test` 和一些能用 `go test` 运行的测试函数的编写约定。这个相对轻量级的机制对单纯的测试很有效，并且这种方式也很自然地扩展到基准测试和文档系统的示例。

实际上，编写测试代码和编写原始程序并没有什么不同。我们编写聚焦于任务的部分功能的简单函数。我们必须谨防条件边界，思考数据结构，并且合理地设计如何根据合适的输入得到输出。这和编写常规的 Go 代码没有区别，这不需要新的注解、约定和工具。

11.1 go test 工具

`go test` 子命令是 Go 语言包的测试驱动程序，这些包根据某些约定组织在一起。在一个包目录中，以 `_test.go` 结尾的文件不是 `go build` 命令编译的目标，而是 `go test` 编译的目标。

在 `*_test.go` 文件中，三种函数需要特殊对待，即功能测试函数、基准测试函数和示例函数。功能测试函数是以 `Test` 前缀命名的函数，用来检测一些程序逻辑的正确性，`go test` 运行测试函数，并且报告结果是 `PASS` 还是 `FAIL`。基准测试函数的名称以 `Benchmark` 开头，用来测试某些操作的性能，`go test` 汇报操作的平均执行时间。示例函数的名称，以 `Example` 开头，用来提供机器检查过的文档。11.2 节讲解功能测试函数，11.4 节讲解基准测试函数，11.6 节讲解示例函数。

`go test` 工具扫描 `*_test.go` 文件来寻找特殊函数，并生成一个临时的 `main` 包来调用它

们，然后编译和运行，并汇报结果，最后清空临时文件。

11.2 Test 函数

每一个测试文件必须导入 `testing` 包。这些函数的函数签名如下：

```
func TestName(t *testing.T) {
    // ...
}
```

功能测试函数必须以 `Test` 开头，可选的后缀名称必须以大写字母开头：

```
func TestSin(t *testing.T) { /* ... */ }
func TestCos(t *testing.T) { /* ... */ }
func TestLog(t *testing.T) { /* ... */ }
```

参数 `t` 提供了汇报测试失败和日志记录的功能。定义一个示例包 `gopl.io/ch11/word1`，这个包包含一个函数 `IsPalindrome`，用来判断一个字符串是否是回文字符串（这个函数在字符串是回文字符串的情况下对于每个字节检查了两次，后面会实现简短的版本）。

```
gopl.io/ch11/word1
// 包 word 提供了文字游戏相关的工具函数
package word

// IsPalindrome 判断一个字符串是否是回文字符串
// (Our first attempt.)
func IsPalindrome(s string) bool {
    for i := range s {
        if s[i] != s[len(s)-1-i] {
            return false
        }
    }
    return true
}
```

在同一个目录中，文件 `word_test.go` 包含了两个功能测试函数 `TestPalindrome` 和 `TestNonPalindrome`。两个函数都检查 `isPalindrome` 是否针对单个输入参数给出了正确的结果，并且用 `t.Error` 来报错。

```
package word

import "testing"

func TestPalindrome(t *testing.T) {
    if !IsPalindrome("detartrated") {
        t.Error(`IsPalindrome("detartrated") = false`)
    }
    if !IsPalindrome("kayak") {
        t.Error(`IsPalindrome("kayak") = false`)
    }
}

func TestNonPalindrome(t *testing.T) {
    if IsPalindrome("palindrome") {
        t.Error(`IsPalindrome("palindrome") = true`)
    }
}
```

`go test`（或者 `go build`）命令在不指定包参数的情况下，以当前目录所在的包为参数。可以用下面的命令来编译和运行测试：

```
$ cd $GOPATH/src/gopl.io/ch11/word1
$ go test
ok    gopl.io/ch11/word1  0.008s
```

测试通过，我们发布了程序，但是午餐聚会的客人们离开不久之后，就开始有 bug 提交过来了。有个法国用户 Noelle Eve Elleon 抱怨说 `IsPalindrome` 函数无法识别 “été”。另一个来自中美洲的用户对程序无法判断出 “A man, a plan, a canal: Panama” 也是回文感到失望。这些特定的小 bug 自然导致了新的测试用例的产生。

```
func TestFrenchPalindrome(t *testing.T) {
    if !IsPalindrome("été") {
        t.Error(`IsPalindrome("été") = false`)
    }
}

func TestCanalPalindrome(t *testing.T) {
    input := "A man, a plan, a canal: Panama"
    if !IsPalindrome(input) {
        t.Errorf(`IsPalindrome(%q) = false`, input)
    }
}
```

由于 `input` 很长，为了避免写两次，这里使用了函数 `Errorf`，这个函数和 `Printf` 一样提供了格式化功能。

当添加这两个新的测试之后，`go test` 命令失败了，给出如下错误消息：

```
$ go test
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
FAIL    gopl.io/ch11/word1  0.014s
```

比较好的实践是先写测试然后发现它触发的错误和用户 bug 报告里面的一致。只有这个时候，我们才能确信我们修复的内容是针对这个出现的问题的。

另外，运行 `go test` 比手动测试 bug 报告中的内容要快得多，测试可以让我们顺序地检查内容。如果一个测试套件（test suite）里面有很多测试用例，我们可以选择性地测试用例来加加测试过程。

选项 `-v` 可以输出包中每个测试用例的名称和执行的时间：

```
$ go test -v
==== RUN TestPalindrome
--- PASS: TestPalindrome (0.00s)
==== RUN TestNonPalindrome
--- PASS: TestNonPalindrome (0.00s)
==== 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  0.017s
```