Table of Contents

前言

这本书能够帮助你成为一个更棒的Go开发者。

我尽最大努力将这本书理论与实战兼顾，但是只有你的反馈才是评价这本书成功与否的标准。另外，本书的代码示例都是独立的，意味着你可以单独使用或者以其为模板创建更加复杂的应用。

每章最后的练习题要亲自动手去敲，有什么问题尽管联系我，因为你的反馈与建议可以让本书的下一版本更加完善！

章节概览

第一章，Go与操作系统，首先讨论了Go的历史、Go的优势、及*godoc*的用法，并且教你如何编译和执行一个Go程序；接下来阐述了用户输入输出，如何使用命令行参数以及log文件；本章最后一个主题是错误处理，其在Go里具有举足轻重的地位。

第二章，深入剖析Go的各种内部原理，讨论了Go垃圾回收机制及其内部原理；然后展示了一些不安全的代码和包，C代码与Go的互相调用，defer关键字及strace、dtrace两个小工具的使用示例；本章最后你将学习如何使用Go汇编器获取环境变量的信息。

第三章，Go基本数据类型，讨论Go提供的基本数据类型，包括数组，切片，映射，指针，常量，循环以及时间与日期的基本操作。我想你不会愿意错过这一章的！

第四章，组合类型的使用，以Go结构体与struct关键字开始本章，之后讨论了元组、字符串、rune、字节切片以及字符串字面量；最后讲解了正则表达式与模式匹配，switch语句、strings包、math/big包，以及如何使用Go实现简单的k-v存储。

第五章，数据结构的Go描述，当Go提供的内置基本类型不能满足你的需求时，你可以实现自己的数据类型，本章将教你实现包括二叉树、链表、哈希表、栈、队列在内的数据结构并了解它们的优势及具体使用场景。最后，你将学会如何使用Go产生随机数。

第六章， Go package中不为人知的知识，本章包括Go的包和函数的知识，如init()函数，syscall标准库，text/template和html/template包。这一章将会让你成为更棒的Go开发者。

第七章，反射和接口，我们将讨论更高级的概念，包括反射，接口，类型方法。Go的面向对象编程也会在本章出现！

第八章，Go UNIX系统编程，本章关于如何使用Go进行UNIX系统编程，包括使用flag包获取命令行参数，处理UNIX信号量，文件的输入输出，bytes包以及io.Reader和io.Writer接口。我之前提到过，如果你想在系统编程进行更加深入的研究，读完本书后可以阅读*Go系统编程*一书。

第九章，并发-Goroutines,Channel和Pipeline，本章讨论Goroutines,Channel和Pipeline，Go使用这些技术实现并发操作。你将了解到进程、线程、goroutine之间的区别，sync包和调度器原理也会涉及。

第十章，Go 并发-进阶讨论，在上一章的基础上进一步讨论Go并发编程，旨在帮助你成为goroutine和channel的专家！主要内容有，Go调度器深入讲解，强大的select关键字的使用，channel的不同类型，共享内存，互斥锁，sync.Mutex与syncRWMutex类型的使用。最后将会讨论context包，工作池以及如何检测竞争代码的状态。

十一章，代码测试，优化以及分析，本章内容包括代码测试，优化，代码分析以及交叉编译，创建文档，基准测试等内容。

十二章， Go网络编程基础，主要内容是net/http包的使用以及如何使用Go设计实现web客户端与服务器。除此之外涉及http.Response，http.Request的使用，http.Transport结构和http.NewServerMux类型的讲解。通过本章学习，你将能够使用Go开发完整的网站！最后，你会了解如何使用Go读取网络接口配置和实现DNS轮询。

十三章， 网络编程-构建服务器与客户端，本章讨论如何基于net包实现UDP和TCP客户端与服务器，如何创建RPC客户端与服务器，实现并发的TCP服务器以及读取原生网络包。

更多信息

为了更好地学习这本书，你需要拥有一台类UNIX系统的机器，Mac OS和Linux均可，并且安装较新版本的Go。本书大部分的代码可以在Microsoft windows机器上运行。

你应当尽可能地在你的程序中用上本书的知识，以验证哪些可以正常运行哪些不能。就像我之前说的那样，你应当亲自动手解决每章最后的练习题，或者编写自己的程序。

代码规范约定

下面是关于本书的代码规范约定。

文本框中的内容：代码，数据库表名，文件夹名，文件名，扩展名，路径名，URL，用户输入等。

下面是代码块示例：

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("This is a sample Go program!")
}
```

代码块中强调的代码，会用下划线或者加粗的形式给出：

> import ("fmt") func main() {
>
> **fmt.Println("This is a sample Go program!")**
>
> }

命令行输入输出的格式如下所示：

```
$ date

Sat Oct 21 20:09:20 EEST 2017

$ go version

go version go1.9.1 darwin/amd64
```

粗体：意味着这是重要的内容，或者是从屏幕中输出的内容。

本书结构

《Mastering Go》此本书可以从逻辑上分为三部分。第一部分由前四章构成，会快速的浏览一下Go语言的重要概念，包括用户输入输出、下载与使用第三方Go的包、如何编译Go的代码、如何在Go代码中调用C的代码，以及如何操作与使用Go的基本类型与组合类型等。

第二部分包括三个章节，主要介绍Go的代码如何组织、Go项目如何设计，以及Go语言的高级特性。

第三部分包括六个章节，主要涵盖Go语言实践过程中的高级话题，包括Go语言的系统编程、Go语言的并发，代码测试、优化与审计。本书最后两章会涉及网络编程的相关的概念。

本书展示的示例代码较少，主要有两个方面的原因：一方面，在了解一项技术实现的时候，不会被无止境的代码绕晕；另一方面，示例代码只是起到抛砖引玉的作用，你可以将它作为一个简单的开始，来编写你自己的应用。

本书主要以类Unix操作系统为例，但是并不代表Go的代码不能在Windows的操作系统中运行，因为Go的代码是可移植的！之所以这样介绍，是因为本书的示例代码在类Unix操作系统中，特别是Mac OS(版本为High Sierra)与Debian Linux系统中测试通过而已。

Go的历史

Go是当下比较流行的开源的语言之一，第一次发布是在2009年年底。它开始于Google内部的项目，并且吸收了C、Pascal和Alef等语言的优质特性，最早由Robert Griesemer、Ken Thomson和Rob Pike联合撰写。三位作者的初衷是基于Go，能够开发一个可靠的和高效的软件系统。除了语法和标准函数之外，Go还提供了相当丰富的标准库的实现。

编写此书的时候，Go的稳定版本是1.9.1，1.9.2版本也即将发布：

```
$ date
Sat Oct 21 20:09:20 EEST 2017
$ go version
go version go1.9.1 darwin/amd64
```

我非常确信在本书真正出版的时候，通过命令**go version**命令，输出的结果跟上面一定不一样！但是，本书介绍的内容至少在若干年内不会过时，即使Go版本迭代如此之迅速。

如果你是第一次安装Go，你可以访问 Golang学习。 还有一个很简单的办法，在类Unix系统中，已经可以通过包管理器安装Go编程语言包，你唯一要做的就是选择你喜欢的包管理器。

为什么要学习Go

Go是当下比较热门和流行的语言，可以帮助你编写更加安全的代码、更少的bug。当然，有一些bug是非常难发现的！通常，Go语言希望成就更多、更快乐的编程者，因此Go的代码通俗易懂、很有吸引力，而且容易编写。

下一部分，我们会谈论Go的很多优秀的特点。

Go的优点

Go语言具有很多优点——其中一些是Go所特有的，有一些是与其他优秀的语言所共有的。

Go最具优势的特性如下：

- Go是由很多丰富经验的开发者所开发；
- Go最早用于Google内部开发生产系统所使用；
- Go的代码容易编写、易于阅读；
- Go希望带给开发者快乐，快乐的心情容易编写优美的代码；
- Go编译器会打印警告和错误信息，能够帮助开发者快速解决问题；
- Go代码是可以移植的，特别是类Unix系统之间；
- Go已经支持面向过程、并发和分布式编程；
- Go支持垃圾回收机制，所以开发者无需关心内存分配与释放的问题；
- Go没有预处理器，所以编译速度很快。因此，可以作为脚本语言使用；
- Go可以构建Web应用，提供简单的Web服务器，供测试使用；
- Go提供很多标准的库和包，简化开发者的开发工作。这些标准库和包经过大量的测试，可以安全使用；
- Go默认启用静态链接，二进制文件很容易在同种操作系统间移植。一旦Go代码编译后，无需关心它所依赖的其他库，就可以很方便的在其他地方执行该二进制文件；
- Go提供命令行式编译命令，无需GUI操作，深受Unix开发者的青睐；
- Go支持Unicode编码，意味着不需要编写很多代码适配多种语言；
- go的多个特性之间都是正交的，保证语言的稳定性和简单性。

Go是完美的么

没有任何语言是完美的，Go也不例外。但是有些语言在某些方面具有特定的优势。就个人而言，我不喜欢Java，过去习惯于使用C++，但是现在一点也不喜欢它了。其中的原因是，我发现Java代码和C++代码不是很让人舒适。

当然，Go也是有缺点的：

- Go不直接支持面向对象编程，这就使得那些已经习惯于面向对象编程的开发者来说非常痛苦。然而，你可以用组合的方式去模拟面向对象的实现方式。
- 对于一些开发者来说，仍然偏爱C，Go不会完全取代C!
- C仍然是处理最快的系统语言，因为类Unix系统就是用C开发的。

  然而，这并不阻碍Go变得越来越流行！

认识GO的内部机制

在前面章节中提到的所有的Go特性都非常简便，你会一直使用它们。然而，更有价值的事情是看到并理解Go程序背后的运行机制。

本章将会介绍垃圾回收以及它的运行机制、如何在Go程序中调用C的代码。在某些情况下在Go中使用C代码是必不可缺的，但是在大多数情况下你并不会用到，因为Go本身就是一种非常强大的编程语言。同样的，如何在C程序中调用Go代码也将会介绍。最后将谈谈如何使用 `panic()`、`recover()` 函数和 `defer` 关键字。

本章你将会学习的主题：

- Go编译器
- Go的垃圾回收是如何工作的
- 如何检测垃圾回收的运转情况
- 在Go中调用C
- 在C中调用Go
- `panic()` 和 `recover()` 函数
- `unsafe` 包
- 方便但又棘手的关键字 `defer`
- Linux工具 `strace`
- `FreeBSD` 和 `macOS High Sierra` 常用的 `dtrace` 工具
- 查找Go环境的信息
- 节点树
- Go汇编

Go 编译器

Go编译器需要在 `go tool` 的帮助下执行。 `go tool` 除了生成可执行文件之外还有很多其他功能。

本章节中使用的 `unsafe.go` 文件不包含任何特殊代码-所提供的命令将适用于每个有效的Go源文件。接下来你将会看到 `unsafe.go` 的内容。

如果你使用 `go tool compile` 命令编译 `unsafe.go` 文件，会得到一个扩展名为 `.o` 的目标文件。下面的展示这个命令在macOS操作系统中执行的结果：

```
$ go tool compile unsafe.go
$ ls -l unsafe.o
-rw-r--r-- 1 mstouk staff 5495 Oct 30 19:51 unsafe.o
$ file unsafe.o
unsafe.o: data
```

目标文件是包含了目标代码的二进制文件，它以可重定位格式的形式展现，很多时候不能被直接执行。可重定位格式的最大好处是在链接阶段它需要尽可能少的内存。

如果执行 `go tool compile -pack` 命令，将会得到一个压缩文件。

```
$ go tool compile -pack unsafe.go
$ ls -l unsafe.a
-rw-r--r-- 1 mtsouk staff 5680 Oct 30 19:52 unsafe.a
$ file unsafe.a
unsafe.a: current ar archive
```

压缩文件也是一个二进制文件，但是它包含了一个或者多个文件，主要的目的是把多个文件组织到单个文件中。Go使用的压缩文件被称为 `ar` 。

使用下面的命令可以列出 `.a` 压缩文件的内容：

```
$ ar t unsafe.a
__.PKGDEF
_go_.o
```

`go tool compile` 的另一个真正有价值的命令行标志是 `-race` ，它允许您检测竞态条件。在第十章中你会学到更多关于竞态条件以及为什么要避免竞态条件的知识点。

本章的最后讨论汇编语言和节点树的时候，你还会学到 一个同样有用的 `go tool compile` 命令。现在为了撩动你，尝试一下下面的命令吧：

```
$ go tool compile -S unsafe.go
```

你可能发现，以上命令的输出会让你难以理解，这正说明了Go可以很好地帮你隐藏不必要的复杂性，除非你要求Go展示出来。

垃圾回收

垃圾回收是释放掉那些不再被使用的的内存空间的过程。换句话说，垃圾回收器会去检查哪些对象超出范围并且不会再被引用到，然后它会去释放掉那些对象占用的内存空间。这个过程是在go程序运行中以并发的方式去进行的，不是go程序执行之前，也不是go程序执行之后。go垃圾回收器实现的说明文档给出了如下声明(runtime下的mgc.go中)：

The GC runs concurrently with mutator threads, is type accurate (aka precise), allows multiple GC thread to run in parallel. It is a concurrent mark and sweep that uses a write barrier. It is non-generational and non-compacting. Allocation is done using size segregated per P allocation areas to minimize fragmentation while eliminating locks in the common case.

垃圾回收是和go线程同时运行的，它是类型精确的，而且多个垃圾回收线程可以并行运行。它是一种使用了写屏障的并发标记清除的垃圾回收方式。它是非分代和非压缩的。使用按P分配区域隔离的大小来完成分配，以最小化碎片，同时消除常见情况下的锁。

这里面有很多术语，我一会儿会解释。但是首先，我会为你展示一种查看垃圾回收过程的参数的方式。幸运的是，Go标准库提供了方法，允许你去学习垃圾回收器的操作以及了解更多关于垃圾回收器在背后所做的事情。相关的代码保存在了gColl.go中，它有三个部分。

gColl.go的第一部分代码段如下所示：

```
package main

import (
    "fmt"
    "os"
    "runtime"
    "runtime/trace"
    "time"
)

func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC)
    fmt.Println("-----")
}
```

注意到每次你都需要获取更多最近的垃圾回收统计信息，你会需要调用 `runtime.ReadMemStats()` 方法。 `printStats()` 方法的目的是去避免每次要写相同代码。

第二部分代码如下：

```
func main() {
    f, err := os.Create("/tmp/traceFile.out")
    if err != nil {
        panic(err)
    }
    defer f.Close()
    err = trace.Start(f)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer trace.Stop()

    var mem runtime.MemStats
    printStats(mem)
    for i := 0; i < 10; i++ {
        s := make([]byte, 50000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
    }
    printStats(mem)
```

for循环里创建一堆大的go slices，目的是为了进行大量内存分配来触发垃圾回收。

最后一部分是接下来 `gColl.go` 的代码，用go slices进行了更多的内存分配。

```
    for i := 0; i < 10; i++ {
        s := make([]byte, 100000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
        time.Sleep(time.Millisecond)
    }
    printStats(mem)
}
```

在macOS High Sierra上面gColl.go的输出如下：

```
mem.Alloc: 66024
mem.TotalAlloc: 66024
mem.HeapAlloc: 66024
mem.NumGC: 0
-----
mem.Alloc: 50078496
mem.TotalAlloc: 500117056
mem.HeapAlloc: 50078496
mem.NumGC: 10
-----
mem.Alloc: 76712
mem.TotalAlloc: 1500199904
mem.HeapAlloc: 76712
mem.NumGC: 20
-----
```

尽管你不会每次都去检查go垃圾收集器的操作，但是在一个慢的应用程序上可以看到go垃圾回收器的工作方式，长时间的运行里会节省你很多时间。我很确定，花点时间去整体学习了解垃圾回收，尤其是了解go垃圾回收器的工作方式，你不会后悔的。

这里有一个技巧可以让你得到更多关于go 垃圾收集器操作的细节，使用下面这个命令。

```
GODEBUG=gctrace=1 go run gColl.go
```

所以，如果你在任何 `go run` 命令前面加上 `GODEBUG=gctrace=1` ，go就会去打印关于垃圾回收操作的一些分析数据。生成的数据是如下的形式：

```
gc 4 @0.025s 0%: 0.002+0.65+0.018 ms clock, 0.021+0.040/0.057/0.
gc 17 @30.103s 0%: 0.004+0.080+0.019ms clock, 0.033+0/0.076/0.07
```

这些数据给你提供了更多垃圾回收过程中的堆内存大小的信息。让我们以 `47->47->0 MB` 这三个值为例。第一个数值是垃圾回收器要去运行时候的堆内存大小。第二个值是垃圾回收器操作结束时候的堆内存大小。最后一个值就是生存堆的大小。

三色算法

go垃圾回收器的操作都是基于**三色算法**，这一章节主要来说明此算法。

三色算法并不是**go**独有的，它也会在其它编程语言中使用到

严格来说，在Go中这个算法的官方名称是叫做**三色标记清除算法（tricolor mark-and-sweep algorithm）**。它可以和程序一起并发工作并且使用**写屏障（write barrier）**。这就意味着，当Go程序员运行起来，go调度器去负责应用程序的调度，而垃圾回收器会像调度器处理常规应用程序一样，去使用多个**goroutines**去进行工作。你可以在第九章了解更多关于goroutines以及go调度器的相关内容，包括*Go 并发 - Goroutines，Channels，和管道（Piplines）*。

这个算法背后的核心思想是由Edsger W. Dijkstra，Leslie Lamport，A.J.Martin，C.S.Scholten和E.F.M.Steffens这些大佬提出的。算法首先发表在论文*On-the-fly Garbage Collection：An Exercise in Cooperation*上面。三色标记清除算法背后的首要原则就是它把堆中的对象根据它们的颜色分到不同集合里面，颜色是根据算法进行标记的，我会在这一章的*更多关于go垃圾回收*这部分对其算法进行进一步说明。

现在让我们来谈谈每种颜色集合代表的含义。**黑色集合**是为了确保没有任何指针指向白色集合。但是**白色集合**中的对象允许有指针指向黑色集合，因为这不会对垃圾回收器的操作产生影响。**灰色集合**可能会有指针指向白色集合里的对象。白色集合中的对象就是垃圾回收的候选对象。

注意到没有任何对象可以从黑色集合进到白色集合，这允许算法能够去操作并且清除白色集合里的对象。此外，没有任何黑色集合里的指针对象能够直接指向白色集合中的对象。

当垃圾回收开始，全部对象标记为白色，然后垃圾回收器会遍历所有根对象并把它们标记为灰色。**根对象**就是程序能直接访问到的对象，包括全局变量以及栈里面的东西。这些对象大多数取决于特定程序的go代码。在这之后，垃圾回收器选取一个灰色的对象，把它变为黑色，然后开始寻找去确定这个对象是否有指针指向白色集合的对象。这意味着当一个灰色对象由于被其它对象的指针所指而扫描到的时候，这个灰色对象会被标记为黑色。如果扫描发现这个灰色对象有一个或者更多指针指向白色对象时，会把所指向的白色对象放到灰色集合里。只要有灰色集合对象存在，这个过程就会一直进行下去。之后，白色集合里的对象就是没人访问的对象，并且它们所占用的内存可以被回收重用。因此，在这个点上，我们说白色集合里的元素被垃圾回收了。

如果垃圾回收过程中，一个灰色对象在某些情况变为不可达状态，它在那次垃圾回收中就不会被回收了，但是不是说下次也不会回收！尽管这不是最佳情况，但也没有那么糟

在这个过程中，运行应用程序被叫做**修改器（mutator）**。mutator去运行一个小的方法叫做**写屏障(write barrier)**，每次堆中的指针被修改写屏障都会去执行。如果堆中对象的指针被修改，就意味着那个对象现在是可触达的，写屏障会把它标记为灰色并把它放到灰色集合中。

mutator负责保持黑色集合中没有任何元素的指针去指向白色集合中的元素。这是在写屏障方法的帮助下完成的。如果维持这个不变状态失败的话，会毁坏垃圾回收过程，并且很可能会以一种丑陋和非预期的方式破坏你的程序。

堆可以看成许多连接对象的图，如下所示，展示了单独一个垃圾回收的过程。

因此，我们有三种不同颜色：黑色、白色和灰色。当算法开始的时候，所有对象标记为白色。随着算法继续进行，白色对象移到了其它两种颜色集合的一种里面。最后留在白色集合里面的对象会在将来某个时间点被清理掉。

在前面的图里，你可以看到白色对象E，它是在白色集合里而且可以访问对象F，E不会被任何其它的对象访问到因为没有其它指向E的指针，这使得E成为了垃圾回收的最佳候选对象！另外，对象A、B和C是根对象而且总是可达的，因此它们不会被垃圾回收掉。

接下来，算法会去处理留下的灰色集合元素，这意味着对象A和F会进入到黑色集合里。对象A会进入到黑色集合是因为它是一个根元素，而F会进入黑色集合是因为它没有指向任何其它对象但是是在灰色集合里。在对象A被垃圾回收之后，对象F会变成不可达状态并且会在下一次垃圾回收器的处理循环中被回收掉。

go垃圾回收也可以应用于其它变量例如**channel**！当垃圾回收器发现一个channel是不可达的而且channel变量再也不会被访问到，它就会释放掉它的资源甚至说channel还没被关闭！你将会了解更多channels的东西在第九章里，*Go并发 - Groutines，Channel和Pipelines*。

Go允许你通过在你的Go代码里放一个 `runtime.GC()` 的声明来手动去开启一次垃圾回收。但是，要记住一点， `runtime.GC()` 会阻塞调用器，并且它可能会阻塞整个程序，尤其是如果你想运行一个非常繁忙的而且有很多对象的go程序。这种情况发生，主要是因为你不能在其他任何事都在频繁变化的时候去处理垃圾回收，因为这种情况不会给垃圾回收器机会，去清楚地确定白色、黑色和灰色集合里的成员。这种垃圾回收状态也被称作是**垃圾回收安全点(garbage collection safe-point)**。

你可以在https://github.com/golang/go/blob/master/src/runtime/mgc.go里找到垃圾回收器相关的高级go代码。你可以学习这个如果你想了解更多关于垃圾回收操作的东西。

go团队一直在优化垃圾回收器，主要是通过降低垃圾回收器所需要处理三种集合上数据的扫描次数来让它更快。但是尽管进行各种优化，其背后算法的整体原理还是一样的。

垃圾回收器背后的更多操作

这一小节会深入探索go垃圾回收器。

go垃圾回收器的主要关注点是低延迟，也就是说为了进行实时操作它会有短暂的暂停。另一方面，创建新对象然后使用指针操作存活对象是程序始终在做的事情，这个过程可能最终会创建出不会再被访问到的对象，因为没有指向那些对象的指针。这种对象即为垃圾对象，它们等待被垃圾回收器清理然后释放它们的空间。之后它们释放的空间可以再次被使用。

垃圾回收中使用的最简单的算法就是经典的**标记清除算法(mark-and-sweep)**：算法为了遍历和标记堆中所有可触达对象，会把程序停下来(**stop the world**)。之后，它会去清扫（sweeps）不可触达的对象。在算法的标记(mark)阶段，每个对象被标记为白色、灰色或黑色。灰色的子对象标记为灰色，而原始的对象此时会标记为黑色。没有更多灰色对象去检查的话就会开始清扫阶段。这个技术适用是因为没有从黑色指向白色的指针，这是算法的基本不变要素。

尽管标记清除算法很简单，但是它会暂停程序的运行，这意味着实际过程中它会增加延迟。go会通过把垃圾回收器作为一个并发的处理过程，同时使用前一节讲的三色算法，来降低这种延迟。但是，在垃圾回收器并发运行时候，其它的过程可能会移动指针或者创建对象，这会让垃圾回收器处理非常困难。所以，让三色算法并发运行的关键点就是，维持标记清除算法的不变要素即没有黑色的对象能够指向白色集合对象。

因此，新对象必须进入到灰色集合，因为这种方式下标记清除的不变要素不会被改变。另外，当程序的一个指针移动，要把指针所指的对象标记为灰色。你可以说灰色集合是白色集合和黑色集合中间的“屏障”。最后，每次一个指针移动，会自动执行一些代码，也就是我们之前提到的**写屏障**，它会去进行重新标色。

为了能够并发运行垃圾回收器，写屏障代码产生的延迟是必要的代价。

注意**Java**程序语言有许多垃圾回收器，它们在各种参数下能够进行高度配置。其中一种垃圾回收器叫G1，推荐在低延迟应用的程序使用它。

*一定要记住，**Go**垃圾回收器是一个实时的垃圾回收器，它是和其他**goroutines**一起并发运行的*

在第11章，代码测试，优化以及分析，你会学习到如何能够用图表的方式呈现程序的性能。这一章节也会包括一些关于Go垃圾回收器操作的一些信息。

下一节会介绍**不安全代码(unsafe code)和 `unsafe` 的go标准库

Unsafe code

Unsafe code是一种绕过go类型安全和内存安全检查的Go代码。大多数情况，unsafe code是和指针相关的。但是要记住使用unsafe code有可能会损害你的程序，所以，如果你不完全确定是否需要用到unsafe code就不要使用它。

以下面的 `unsafe.go` 为例，看一下unsafe code的使用

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    var value int64 = 5
    var p1 = &value
    var p2 = (*int32)(unsafe.Pointer(p1))
```

这里使用了 `unsafe.Pointer()` 方法，这个方法能让你创造一个 `int32` 的 `p2` 指针去指向一个 `int64` 的 `value` 变量，而这个变量是使用 `p1` 指针去访问的，注意这种做法是有风险的。

任何go指针都可以转化为 `unsafe.Pointer` 指针。

*`unsafe.Pointer` 类型的指针可以覆盖掉**go**的系统类型。这毫无疑问很快，但是如果不小心或者不正确使用的话就会很危险，它给了开发者更多选择去掌控数据。*

```

```

```
    fmt.Println("*p1: ", *p1)
    fmt.Println("*p2: ", *p2)
    *p1 = 5434123412312431212
    fmt.Println(value)
    fmt.Println("*p2: ", *p2)
    *p1 = 54341234
    fmt.Println(value)
    fmt.Println("*p2: ", *p2)
```

}

```
###### *你可以使用一个星号(***)来解引用一个指针*

运行```unsafe.go```，会得到如下的输出
```

p1: 5 p2: 5 5434123412312431212 *p2: -930866580 54341234* p2: 54341234 ```

那么这个输出说明了什么呢？它告诉了我们，使用32-bit的指针无法存一个64-bit的整数型

下一节你会看到，使用 `unsafe` 库中的方法可以做许多有趣的事情

关于**unsafe**包

你已经实际操作过 `unsafe` 包的东西了，现在来看一下为什么这个库这么特别。

首先，如果你看了 `unsafe` 包的源码，你可能会感到惊讶。在macOS Hight Sierra系统上，可以使用**Homebrew**安装1.9.1版本的Go 。 `unsafe` 源码路径
在 `/usr/local/Cellar/go/1.9.1/libexec/src/unsafe/unsafe.go` 下面，不包含注释，它的内容如下

```
$ cd /usr/local/Cellar/go/1.9.1/libexec/src/unsafe/
$ grep -v '^//' unsafe.go|grep -v '^$'
package unsafe
type ArbitraryType int
type Pointer *ArbitraryType
func Sizeof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr
```

OK，其它的 `unsafe` 包的go代码去哪里了？答案很简单：当你import到你程序里的时候，Go编译器实现了这个unsafe库。

许多系统库，例如 `runtime` , `syscall` 和 `os` 会经常使用到 `usafe` 库

另一个usafe包的例子

在这一节，你会了解到更多关于 `unsafe` 库的东西，以及通过一个 `moreUnsafe.go` 的小程序来了解unsafe库的兼容性。 `moreUnsafe.go` 做的事情就是使用指针来访问数组里的所有元素。

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    array := [...]int{0, 1, -2, 3, 4}
    pointer := &array[0]
    fmt.Print(*pointer, " ")
    memoryAddress := uintptr(unsafe.Pointer(pointer)) + unsafe.S
    for i := 0; i < len(array)-1; i++ {
        pointer = (*int)(unsafe.Pointer(memoryAddress))
        fmt.Print(*pointer, " ")
        memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsaf
    }
```

首先， `pointer` 变量指向 `array[0]` 的地址， `array[0]` 是整型数组的第一个元素。接下来指向整数值的 `pointer` 变量会传入 `unsafe.Pointer()` 方法，然后传入 `uintptr` 。最后结果存到了 `memoryAddress` 里。

```
后面部分代码如下：
```

```
    fmt.Println()
    pointer = (*int)(unsafe.Pointer(memoryAddress))
    fmt.Print("One more: ", *pointer, " ")
    memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof
    fmt.Println()
```

```
}
```

这里，我们尝试使用指针和内存地址去访问一个不存在的数组元素。由于使用``

执行```moreUnsafe.go```，会产生如下的输出：

$ go run moreUnsafe.go 0 1 -2 3 4 One more: 824634191624 ``` 现在，你使用指针访问了Go数组里的所有元素。但是，这里真正的问题是，当你尝试访问无效的数组元素，程序并不会出错而是会返回一个随机的数字。

C代码中调用Go函数

在C中可以调用Go函数，接下来，呈现两个C语言的例子，在这两个例子中，调用了两个Go函数，其中Go package将会被转换成这两个C程序的共享库。

Go Package

本节将展示C示例程序用到的 `Go package` 代码。 `Go package` 的名字必须是 `main` ，但是文件名可以随意，我们的例子中，文件名是 `usedByC.go` ，分三部分展示。

你可能不了解 `Go package` 机制，在第六章会详细的介绍。

第一部分的 `Go package` 代码：

```
package main

import "C"

import (
    "fmt"
)
```

前面提到， `Go package` 的名字必须是 `main` ，同样的也需要导入 `"C"` `package` 。

第二部分的代码如下：

```
//export PrintMessage
func PrintMessage() {
    fmt.Println("A Go function!")
}
```

如果一个Go函数想要被C语言调用，必须要先导出。你应该在函数实现的开始部分，添加一行以 `export` 开头的注释， `export` 后面要加上函数的名字，这样C程序才知道怎么使用。

最后一部分的代码：

```
//export Multiply
func Multiply(a, b int) int {
    return a * b
}
func main() {
}
```

这个 `main()` 函数不需要任何的函数体，因为不需要导出给C代码使用，同样的，要导出 `Multiply` 函数，也需要将 `//export Multiply` 注释添加到函数实现之前。

接着，你需要用Go代码生成一个C共享库，命令如下：

```
$ go build -o usedByC.o -buildmode=c-shared usedByC.go
```

上面的命令会产生 `usedByC.h` 和 `usedByC.o` 两个文件。

```
$ ls -l usedByC.*
-rw-r--r--@ 1 mtsouk staff
204
-rw-r--r-- 1 mtsouk staff
1365
-rw-r--r-- 1 mtsouk staff 2329472
$ file usedByC.o
usedByC.o: Mach-O 64-bit dynamically
Oct 31 20:37 usedByC.go
Oct 31 20:40 usedByC.h
Oct 31 20:40 usedByC.o
linked shared library x86_64
```

注意：不要修改 `usedByC.h` 文件。

C代码

C代码在 `willUseGo.c` 文件中，分两部分展示，第一部分如下：

```
#include <stdio.h>
#include "usedByC.h"
int main(int argc, char **argv) {
    GoInt x = 12;
    GoInt y = 23;
    printf("About to call a Go function!\n");
    PrintMessage();
```

导入 `usedByC.h` 之后，就可以使用其中实现的函数。

第二部分代码：

```
    GoInt p = Multiply(x,y);
    printf("Product: %d\n",(int)p);
    printf("It worked!\n");
    return 0;
}
```

变量 `GoInt p` 是从Go函数中获取一个整数值，使用 `(int)p` 可以把它转换成C语言的整数。

在Mac机器上编译和执行 `willUseGo.c` ，输出如下：

```
$ gcc -o willUseGo willUseGo.c ./usedByC.o
$ ./willUseGo
About to call a Go function!
A Go function!
Product: 276
It worked!
```

defer关键字

defer关键字的作用是当外围函数返回之后才执行被推迟的函数。在文件输入输出操作中经常可以见到defer关键字，因为它使您不必记住何时关闭已打开的文件：defer关键字调用文件关闭函数关闭已打开的文件时，可以紧靠着文件打开函数之后。在第八章“告诉Unix系统要做什么”中，会介绍如何在文件相关操作中使用defer关键字，本节将介绍defer的其他用法。您还将在 `panic()` 和 `recover()` 两个Go内置函数中看到defer操作。

首先记住一点重要的原则：defer函数在外围函数返回之后，以后进先出(LIFO)的原则执行。简单点说，在一个外围函数中有3个defer函数： `f1()` 最先出现，然后 `f2()` ，最后 `f3()` ，当外围函数执行返回之后， `f3()` 最先被执行，接着是 `f2()` ，最后是 `f1()` 。

如果你感觉这个定义不是很清晰，查看并执行 `defer.go` 源码，这应该会让你理解的更清楚些。下面分三部分呈现这段源码。

源码第一部分：

```
package main
import (
    "fmt"
)
func d1() {
    for i := 3; i > 0; i-- {
        defer fmt.Print(i, " ")
    }
}
```

上面的Go代码实现了一个名为 `d1()` 函数，函数里面有一个 `for` 循环和一个 `defer` 语句，这个 `defer` 语句将会执行三次。

第二部分源码：

```
func d2() {
    for i := 3; i > 0; i-- {
        defer func() {
            fmt.Print(i, " ")
        }()
    }
    fmt.Println()
}
```

这部分代码实现了 `d2()` 函数，它也包含了一个 `for` 循环和一个 `defer` 语句，这个 `defer` 语句也会执行三次，但是这个 `defer` 执行的是匿名函数，并且匿名函数没有带参数。

最后一部分源码：

```
func d3() {
    for i := 3; i > 0; i-- {
        defer func(n int) {
            fmt.Print(n, " ")
        }(i)
    }
}
func main() {
    d1()
    d2()
    fmt.Println()
    d3()
    fmt.Println()
}
```

这部分代码，定义了 `d3()` 函数，它包含了 `for` 循环 `defer` 语句，这个 `defer` 执行带参数的匿名函数，其中参数n使用的是循环中变量i的值。 `main()` 函数调用这三个函数。

执行源码会得到如下输出：

```
$ go run defer.go
1 2 3
0 0 0
1 2 3
```

您很可能会发现生成的输出很复杂且难以理解。 这表明，如果您的代码不清晰或不明确，执行 `defer` 操作可能会产生非常棘手的结果。

让我们分析一下上面的结果，更进一步了解如果不密切关注自己的代码， `defer` 将会变得多么棘手。第一行是 `d1()` 函数输出的（1 2 3），变量i的值在这个函数中顺序是3 、2和1 ，在 `d1()` 中延迟的函数是 `fmt.Print()` 。 因此，当 `d1()` 函数即将返回时，您将以相反的顺序获取for循环中变量i的三个值，因为被延迟的函数以 `LIFO` 顺序执行。

第二行是 `d2()` 函数的输出，很奇怪的是我们得到了单个0，而不是想象中的1、2和3 ，原因很简单，在循环结束后，i的值为0，因为是0值使循环终止。但是，这里棘手的部分是在循环结束后会执行被延迟的匿名函数。因为它没有参数，这意味着，将值为0的i进行三次打印输出！ 在您的项目中，这种令人困惑的代码可能导致令人讨厌的错误，所以尽量避免它！

第三行是 `d3()` 函数的输出，因为匿名函数的参数的存在，每次匿名函数被推迟执行的时候，它都会获取并使用变量i当前的值，所以每次执行匿名函数都有不同的值输出。

到这里，你应该清楚了，使用 `defer` 的最好的方式就是第三个函数展示的方法，这是因为您故意以易于理解的方式在匿名函数中传递所需的变量。

Panic和Recover

本节将向您介绍第1章“Go和操作系统”中首次提到的技术。这种技术涉及使用 `panic()` 和 `recover()` 函数，代码在 `panicRecover.go` 中，将分三部分进行讨论。

严格来说， `panic()` 是一个内置的Go函数，它终止Go程序的当前流程并开始 `panicking` ！ 另一方面， `recover()` 函数也是一个内置的Go函数，允许你收回那些使用了 `panic()` 函数的 `goroutine` 的控制权。

第一部分的程序如下：

```
package main
import (
    "fmt"
)
func a() {
    fmt.Println("Inside a()")
    defer func() {
        if c := recover(); c != nil {
            fmt.Println("Recover inside a()!")
           }
    }()
    fmt.Println("About to call b()")
    b()
    fmt.Println("b() exited!")
    fmt.Println("Exiting a()")
}
```

除了import语句块，这部分代码实现了一个函数 `a()` ，这个函数最重要的部分是 `defer` 代码块，它实现了一个匿名函数，当 `panic()` 函数被调用的时候，这个匿名函数就会被调用。

第二部分的代码：

```
func b() {
    fmt.Println("Inside b()")
    panic("Panic in b()!")
    fmt.Println("Exiting b()")
}
```

最后一部分代码解释了 `panic()` 和 `recover()` 函数：

```
func main() {
    a()
    fmt.Println("main() ended!")
}
```

执行完整的代码，会得到如下的输出：

```
$ go run panicRecover.go
Inside a()
About to call b()
Inside b()
Recover inside a()!
main() ended!
```

这里发生的事真的令人印象深刻！ 但是，正如您从输出中看到的那样， `a` 函数没有正常结束，因为它的最后两个语句没有执行：

```
fmt.Println("b() exited!")
fmt.Println("Exiting a()")
```

然而，好消息是 `panicRecover.go` 根据我们的意愿结束而没有 `panicking` ，因为 `defer` 中使用的匿名函数控制了局面！ 另请注意，函数 `b` 对函数 `a` 一无所知。 但是，函数 `a` 包含处理 `b` 函数的panic情况的Go代码！

Go循环

每个编程语言都有一种进行循环的方式，Go也不例外。Go提供了for循环，用来对多种数据类型进行遍历。

> Go没有提供while关键字。但是，Go的for循环语句完全可以替代while循环。

for循环

for循环是编程中最常见的循环了，它允许你迭代指定次数，或者在for循环开始的时候计算一个初始值然后只要条件满足就一直迭代下去，这些值包括切片、数组、字典等的长度。这意味着使用for循环是获取一个数组、切片或者字典所有元素的最通用的方式。另外一个方式就是使用range关键字。

下面的代码展示了一个for循环的最简单的使用方式，让一个变量在一定范围内进行遍历：

```
for i := 0; i < 100; i++ {

}
```

在前面的循环中，i的取值将是从0到99。一旦i达到了100，for循环将终止执行。在这段代码中，i是一个本地的临时变量，这意味着在for循环的执行终止之后，i会在未来某个时刻被垃圾回收从而消失。然而，如果i是在for循环之外定义的，那它将在for循环之后继续存在，并且它的值将是100。

你可以使用break关键字来退出for循环。break关键字同时也允许你创建一个没有像i < 100的这样的终止条件的for循环，因为终止条件可以放在for循环内部的代码块中。在for循环中允许有多个退出循环的地方。

此外，你可以使用continue关键字在for循环中跳过一次循环。continue关键字会停止执行当前的代码块，跳回顶部的for循环，然后继续进行下一次循环。

while 循环

正如先前提到的，Go没有提供while关键字来进行while循环的编写，但是它允许你使用for循环来替代while循环。这一节将用两个例子来展示for循环是怎么替代while循环的工作的。

下面是一个典型的你会写while(condition）的例子：

```
for {

}
```

注意此时开发者需要使用break关键字来退出这个for循环！

此外，for循环还能模拟一个在其他编程语言中会出现的do...while循环。下面的代码给了一个Go语言中等价于do...while(anExpression）的操作：

```
for ok:=true; ok; ok = anExpression {

}
```

一旦变量ok的值是false，则for循环终止执行。

range关键字

Go同时提供了range关键字，它能配合for循环以帮助你在遍历Go的数据类型时写出更易于理解的代码。

range关键字的主要优势在于并不需要知道一个切片或者字典的长度就可以一个接一个的对它们的元素进行操作。稍后你会看到range的示例。

for循环代码示例

这一节将展示几个for循环的例子。这个代码示例的文件名是loops.go, 将被分为四部分进行展示。第一部分如下：

```
package main

import (
   "fmt"
)

func main() {
   for i := 0; i < 100; i++ {
       if i%20 == 0 {
           continue
       }
       if i == 95 {
           break
       }
       fmt.Print(i, " ")
```

上面的代码展示了一个典型的for循环的例子以及关键字continue和break的使用。

接下来的代码片段是：

```
   fmt.Println()
   i := 10
   for {
       if i < 0 {
           break
       }
       fmt.Print(i, " ")
       i--
   }
   fmt.Println()
```

上面的代码模拟了一个典型的while循环。注意使用了关键字break来退出for循环。

第三段代码如下：

```
    i = 0
    anExpression := true
    for ok := true; ok; ok = anExpression {
      if i > 10 {
          anExpression = false
        }
       fmt.Print(i, " ")
       i++
   }
    fmt.Println()
```

在这段代码中，你能看到前面所说的使用for循环来进行do...while循环的工作的操作。

loops.go的最后一部分展示如下：

```
    anArray := [5]int{0, 1, -1, 2, -2}
    for i, value := range anArray {
        fmt.Println("index:", i, "value: ", value)
    }
}
```

对一个数组使用range关键字会返回两个值：一个是数组的索引，一个是该索引上的值。你可以两个值都使用，或者使用其中一个，甚至如果你只是想单纯的计算一下数组的元素个数的话，也可以两个值都不使用。

执行loops.go会产生如下的输出：

```
$ go run loops.go
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23 24
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11
index: 0 value:  0
index: 1 value:  1
index: 2 value:  -1
index: 3 value:  2
index: 4 value:  -2
```

Go 切片

Go的切片十分强大，可以毫不夸张地说切片完全能够取代数组。只有非常少的情况下，你才需要创建数组而非切片，最常见的场景就是你非常确定你所存储的元素数量。

> 切片的底层是数组，这意味着Go为每一个切片创建一个底层数组

切片作为函数的形参时是传引用操作，传递的是指向切片的内存地址，这意味着在函数中对切片的任何操作都会在函数结束后体现出来。另外，函数中传递切片要比传递同样元素数量的数组高效，因为Go只是传递指向切片的内存地址，而非拷贝整个切片。

切片基本操作

使用下面的代码可创建一个切片字面量：

```
aSliceLiteral := []int{1, 2, 3, 4, 5}
```

与定义数组相比，**切片字面量**只是没有指定元素数量。如果你在**[]**中填入数字，你将得到的是数组。

也可以使用***make()***创建一个空切片，并指定切片的长度和容量。容量这个参数可以省略，在这种情况下容量等于长度。

下面定义一个长度和容量均为**20**的空切片，并且在其需要的时候会自动扩容：

```
integer := make([]int, 20)
```

Go自动将空切片的元素初始化为对应元素的零值，意味着切片初始化时的值是由切片类型决定的。

使用下面的代码遍历切片中的元素：

```
for i :=0; i < len(integer); i++ {

fmt.Println(i)

}
```

切片变量的零值是**nil**, 下面代码可将已有的切片置为空：

```
aSliceLiteral = nil
```

可以使用***append()***函数追加元素到切片，此操作将触发切片自动扩容。

```
integer = append(integer, -5900)
```

integer[0]代表切片**integer**的第一个元素，**integer[len(integer)-1]**代表最后一个元素。

同时，使用**[:]**操作可以获取连续多个元素，下面的代码表示获取第**2**、**3**个元素：

```
integer[1:3]
```

[:]操作也可以帮助你从现有的切片或数组中创建新的切片或数组：

```
s2 := integer[1:3]
```

这种操作叫做**re-slicing**,在某种情况下可能会导致**bug**：

```
package main

import "fmt"

func main() {
 s1 := make([]int, 5)
 reSlice := s1[1:3]
 fmt.Println(s1)
 fmt.Println(reSlice)
 reSlice[0] = -100
 reSlice[1] = 123456
 fmt.Println(s1)
 fmt.Println(reSlice)
}
```

我们使用[:]操作获取第2、3个元素。

> 假设有一个数组a1，你可以执行 `s1 := a1[:]` 来创建一个以a1为引用的切片

将上述代码保存为reslice.go并执行，将得到以下输出;

```
$ go run reslice.go
[0 0 0 0 0]
[0 0]
[0 -100 123456 0 0 ]
[-100 123456]
```

可以看到切片s1的输出是[0 -100 123456 0 0]，但是我们并没有直接改变s1。这说明通过re-slicing操作得到的切片，与原切片指向同一片内存地址！

re-slicing操作的第二个问题是，尽管你可能是想通过使用re-slicing从原切片中得到较小的一个切片，但是原切片的底层数组仍然会跟随着re-slicing得到的小切片，这对于小切片来说并不是什么严重问题，但是在这种情况下就可能导致bug：你将大文件的内容读到切片中，但是你只是想使用其中一小部分。

切片的自动扩容

切片有两个主要的属性：`容量` 和 `长度`，而且这两个属性的值往往是不一样的。一个与数组拥有相同数量元素的切片，二者的长度是相同的，且均可以通过函数 `len()` 获得。切片的容量是指切片能够容纳的元素空间，可以通过 `cap()` 函数获得。由于切片的大小是动态变化的，如果一个切片超出了其设置的容量，Go会自动将该切片的长度变为原来的两倍，以存放超出的元素。

简单来说，当切片的容量和长度相等时，你再往该切片追加元素，切片的容量就会变为原来的两倍，当然长度增加1。然而这种操作可能对于小的切片效果比较好，对于大型切片来说会占用的内存会超出你预期。

下面的代码lenCap.go分三部分，清楚地阐述了切片的容量与长度的变化规律，第一部分代码：

> package main
>
> ```
> func printSlice(x []int)  {
>    for _, number := range x {
>       fmt.Println(number," ")
>    }
>    fmt.Println()
> }
> ```

printSlice()打印一个slice的所有元素。

第二部分代码：

> ```
> func main()  {
>    aSlice := []int{-1, 0, 4}
>    fmt.Printf("aSlice: ")
>    printSlice(aSlice)
>
>    fmt.Printf("Cap: %d, Length: %d\n",cap(aSlice),len(aSlic
>    aSlice = append(aSlice, -100)
>    fmt.Printf("aSlice: ")
>    printSlice(aSlice)
>    fmt.Printf("Cap: %d, Length: %d\n",cap(aSlice),len(aSlic
> ```

这部分代码中，我们往aSlice中添加元素以出发其长度和容量的变化。

第三部分代码：

```
aSlice = append(aSlice, -2)
   aSlice = append(aSlice, -3)
   aSlice = append(aSlice, -4)
   printSlice(aSlice)
   fmt.Printf("Cap: %d, Length: %d\n",cap(aSlice),len(aSlic
}
```

以上代码的执行结果是：

```
$ go run lenCap.go

aSlice: -1
0
4

Cap: 3, Length: 3 aSlice: -1
0
4
-100

Cap: 6, Length: 4 -1
0
4
-100
-2
-3
-4

Cap: 12, Length: 7
```

正如输出所示，初始的切片长度和容量均是3，在添加一个元素之后，其长度变为4，然后容量变为6。继续往切片中追加元素，其长度变为7，容量再一次翻倍即变为12。

字节切片

类型为byte的切片成为字节切片，可通过下面的代码创建一个字节切片：

```
s := make([]byte,5)
```

字节切片的操作与其他类型的切片并没有什么区别，但是在输入输出中（网络，文件流等）使用的非常多，你将在 第八章 大量使用字节切片。

copy()函数

你可以从现有的数组中创建一个切片，你也可以使用 `copy()` 函数复制一个切片。然而， `copy()` 的用法可能会让你觉得非常奇怪，下面的copySlice.go代码将清楚地阐述其使用方法。

> 使用copy()时你应小心翼翼，因为内建函数copy(dst,src)会以len(dst)和len(src)中的最小值为复制长度。

第一部分代码是：

```
package main

import "fmt"

func main()  {
   a6 := []int{-10, 1, 2, 3, 4, 5}
   a4 := []int{-1, -2, -3, -4}
   fmt.Println("a6:", a6)
   fmt.Printf("a4:", a4)

   copy(a6,a4)
   fmt.Println("a6:", a6)
   fmt.Printf("a4:", a4)
   fmt.Println()
```

上面的代码，我们定义了两个切片分别是a6和a4,打印之后，将a4拷贝到a6。由于a6比a4拥有更多的元素，a4中所有的元素将会拷贝至a6，a6中剩下的两个元素会保持原状。

第二部分代码：

```
b6 := []int{-10, 1, 2, 3, 4, 5}
b4 := []int{-1, -2, -3, -4}
fmt.Println("b6:", b6)
fmt.Printf("b4:", b4)

copy(b4,b6)
fmt.Println("b6:", b6)
fmt.Printf("b4:", b4)
fmt.Println()
```

在这一部分代码中，由于a4的长度为4，所以b6的前四个元素才会拷贝到a4。

copySlice.go的第三部分代码：

```
fmt.Println()
array4 := [4]int{4, -4, 4, -4}
s6 := []int{1,-1,1,-1,-5,5}

copy(s6, array4[0:])
fmt.Println("array4:", array4[0:])
fmt.Printf("s6:", s6)
fmt.Println()
```

在这部分代码，我们尝试将具有四个元素的数组拷贝到具有6哥元素的切片，注意，我们使用[:]操作将数组转为切片（array4[0:]）。

最后一部分代码：

```
array5 := [5]int{5,-5,5,-5,5}
s7 := []int{7,7,-7,7,-7,7}
copy(array5[0:],s7)
fmt.Println("array5:", array5)
fmt.Printf("s7:", s7)
fmt.Println()
```

我们将一个切片拷贝至具有五个元素的切片，由于copy()函数只接收切片，所以依然需要[:]操作将数组转为切片。

如果你尝试将一个数组拷贝至切片或者将切片拷贝至数组，就会得到下面的编译错误：

```
./copySlice.go:37:6: first argument to copy should be slice; have [5]int
```

最后，给出执行copySlice.go的输出：

```
a6: [-10 1 2 3 4 5] a4:%!(EXTRA []int=[-1 -2 -3 -4])

a6: [-1 -2 -3 -4 4 5] a4: [-1 -2 -3 -4] b6: [-10 1 2 3 4 5] b4:[-1 -2 -3 -4]

b6: [-10 1 2 3 4 5] b4:[-10 1 2 3]

array4: [4 -4 4 -4] s6:[4 -4 4 -4 -5 5] array5: [7 7 -7 7 -7] s7:[7 7 -7 7 -7 7]
```

多维切片

跟数组一样，切片同样可以是多维的，下面代码创建一个二维切片：

```
s1 := make([][]int, 4)
```

> 如果你发现你的代码中出现很多多维切片，你就要考虑你的代码设计是否合理并且使用不需要多维切片的更简单的设计。

下一小结中你会看到一段使用多维切片的代码示例。

使用切片的代码示例

希望slices.go中的代码能够扫除你对切片所有的困惑，本节代码分为5个部分。

第一部分包含切片的定义以及初始值的说明:

```
package main

import "fmt"

func main()  {
   aSlice := []int{1, 2, 3, 4, 5}
   fmt.Println(aSlice)
   integer := make([]int,2)
   fmt.Println(integer)
   integer = nil
   fmt.Println(integer)
```

第二部分展示了如何使用 `[:]` 以数组创建切片。注意， `[:]` 操作只是将引用指向数组，并没有创建一份数组的拷贝:

```
anArray := [5]int{-1, -2, -3, -4, -5}
refAnArray := anArray[:]

fmt.Println(anArray)
fmt.Println(refAnArray)
anArray[4] = -100
fmt.Println(refAnArray)
```

第三部分代码使用 `make()` 创建一个二维切片:

```
s := make([]byte,5)
fmt.Println(s)
twoD := make([][]int,3)
fmt.Println(twoD)
fmt.Println()
```

Go自动将切片的元素值初始化为对应切片类型的零值，例如int类型的零值是 `0` ，切片的零值是 `nil` 。记住，多维切片的元素类型是切片！

slices.go的第四部分，你将看到初始化二维数组的方法:

```
for i := 0; i < len(twoD);i++ {
   for j := 0; j < 2; j++ {
      twoD[i] = append(twoD[i], i*j)
   }
}
```

上述代码使用append()函数往切片中追加元素，切记不要使用不存在的索引值，否则你将看到 `panic:runtime error: index out of range` 。

最后一部分代码展示了如何使用 `range` 关键字打印二维切片的所有元素:

```
for _, x := range twoD {
      for i,y := range x {
         fmt.Println("i:",i, "value:",y)
      }

      fmt.Println()
   }

}
```

执行slices.go，你将看到如下输出:

```
$ go run slices.go
[1 2 3 4 5][0 0] [][-1 -2 -3 -4 -5] [-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -100]
[0 0 0 0 0][[]
i: 0 value: 0 i: 1 value: 0
i: 0 value: 0 i: 1 value: 1
i: 0 value: 0 i: 1 value: 2
```

由于切片的零值是 `nil` ，所以二维切片中的切片元素被初始化为 `nil` ，打印出来就是空的。

使用sort.slice()排序

本节介绍 `sort.Slice()` ，这个函数是在Go 1.8中被初次引入的。这意味着 `sortSlice.go` 中的代码不能在低于1.8版本的Go环境中运行。这部分代码将分三部分解释，第一部分是:

```
package main

import (
   "fmt"
   "sort"
)

type aStructure struct {
   person string
   height int
   weight int
}
```

你可能是第一次在本书中看到 `Go structure` ，在第四章*组合类型的使用*中，你将全面了解 `Go structure` 的知识。现在你只需要记住，结构体是拥有不同类型的多个变量的数据类型。

下面是第二部分代码：

```
sort.Slice(mySlice, func(i, j int) bool {
      return  mySlice[i].height < mySlice[j].height
   })
   fmt.Println("<:", mySlice)

   sort.Slice(mySlice, func(i, j int) bool {
      return mySlice[i].height > mySlice[j].height
   })
   fmt.Println(">:",mySlice)
}
```

在这里你创建了一个名为 `mySlice` 的切片，元素是 `aStructure` 结构体。

最后一部分代码：

```
sort.Slice(mySlice, func(i, j int) bool {
      return  mySlice[i].height < mySlice[j].height
   })
   fmt.Println("<:", mySlice)

   sort.Slice(mySlice, func(i, j int) bool {
      return mySlice[i].height > mySlice[j].height
   })
   fmt.Println(">:",mySlice)
}
```

我们使用了 `sort.Slice()` 及两个匿名函数对 `mySlice` 进行排序，匿名函数使用了 `aStructure` 的 `height` 字段。

> sort.Slice()函数根据匿名排序函数对切片中的元素进行排序。

执行 `sort.Slice.go` 之后，将会得到下面的输出：

> $ go run sortSlice.go
>
> 0: [{Mihalis 180 90} {Bill 134 45} {Merietta 155 45} {Epifanios 144 50} {Athina 134 40}]
>
> <: [{Bill 134 45} {Athina 134 40} {Epifanios 144 50} {Merietta 155 45} {Mihalis 180 90}]
>
> : [{Mihalis 180 90} {Merietta 155 45} {Epifanios144 50} {Bill 134 45} {Athina 134 40}]

如果你在Go版本低于1.8的UNIX系统中执行 `sort.Slice()` ，就会得到下面的错误信息：

> $ go version
>
> go version g01.3.3 linux/amd64
>
> $ go run sortSlice.go
>
> ./sortSlice.go:24: undefined: sort.Slice
>
> ./sortSlice.go:24: undefined: sort.Slice

Map值为nil的坑

下面的代码会正常工作:

```
aMap := map[string]int{}

aMap["test"] = 1
```

然而下面的代码是不能工作的，因为你将 `nil` 赋值给map:

```
aMap := map[string]int{}

aMap = nil

fmt.Println(aMap)

aMap["test"] = 1
```

将以上代码保存至 `failMap.go` ,执行后会产生下面的错误信息（事实上，如果你用IDE编程，IDE就会提醒你有错误）:

```
$ go run failMap.go

map[]

panic: assiment to entry in nil map
```

何时该使用Map?

与切片与数组相比，map（映射）的功能要强大的多，但是具有灵活性的同时也伴随着性能损耗，实现Go map往往需要更多的处理能力。但是不用担心，Go的内置数据结构是非常高效的，所以当你需要map的时候就尽情地去用吧！

你应该记住的是Go map是非常方便的而且能存储多种不同的数据类型，同时其用法易于理解且上手迅速。

Go 常量

常量是的值是不能改变的，Go使用关键字 `const` 定义常量。

> 通常来说，常量是全局变量。因此，当你的代码中出现大量在局部定义的常量时，你就应该考虑重新设计你的代码了。

显而易见，使用常量的好处就是保证了该值不会在程序运行过程中被修改！

严格来说，常量的值在编译期间就被确定了。在这种情况下，Go可以使用布尔类型、字符串、或者数字类型存储常量的值。

你可以使用下面的代码定义常量：

```
const HEIGHT = 200
```

另外，你还可以一次性定义多个常量：

```
const (
C1 = "C1C1C1"
C2 = "C2C2C2"
C3 = "C3C3C3"
)
```

下面这三种声明变量的方式在Go看来是一样的：

```
s1 := "My String"
var s2 = "My String"
var s3 string = "My String"
```

以上三个变量的声明并没有使用 `const` 关键字，所以它们并不是常量。这并不意味着你不能使用相似的方式定义两个常量：

```
const s1 = "My String"
const s2 string = "My String"
```

尽管 `s1` 和 `s2` 都是常量，但是 `s2` 定义时声明了其类型，意味着它比常量 `s1` 的定义更加严格。这是因为一个声明类型的Go常量必须遵循与声明过类型的变量相同的严格规则，换句话说，未声明类型的常量无需遵循严格规则，使用起来会更加自由。但是，即使在定义常量时没有声明其类型，Go会根据其值判断其类型，因为你不想在使用该常量时考虑所有的规则。下面我们将用一个简单的例子来说明，当你为常量赋予具体类型时会遇到哪些问题:

> const s1 = 123
>
> const s2 float64 = 123
>
> var v1 float32 = s1*12
>
> var v2 float32 = s2*12

编译器正常通过 `v1` 的声明及初始化，但是由于 `s2` 和 `v2` 的类型不同，编译器就会报错：

> $ go run a.go
>
> $ command-line-argument
>
> ./a.go:12:6: canot use s2 * 12 (type float64) as type float32 in assignment
>
> 代码建议：如果你要用到许多常量，最好将它们定义到同一个包中。

常量生成器：iota

常量生成器iota使用递增的数字来声明一系列相关的值，且无需明确定义其类型。

大部分与 `const` 关键字相关的概念，包括常量生成器iota，将会分成四部分，在 `constants.go` 中阐述。

第一部分代码：

```
package main

import "fmt"

type Digit int
type Power2 int

const PI  = 3.1415926

const (

   C1 = "C1C1C1"

   C2 = "C2C2C2"

   C3 = "C3C3C3"

)
```

这部分自定义了两个类型，分别叫做 `Digit` 和 `Power2` ,以及四个常量。

> Go可以使用现有的类型自定义新的类型，目的是从名字上区分可能使用相同数据类型，但是具有不同含义的变量。

第二部分的代码：

```
func main() {

   const s1  = 123
   var v1 float32 = s1 * 12
   fmt.Println(v1)
   fmt.Println(PI)
```

这部分声明了一个新的常量 `s1` ,并且在 `v1` 的声明中用到了它。

第三部分代码：

```
const   (
   Zero Digit = iota
   One
   Two
   Three
   Four
)
fmt.Println(One)
fmt.Println(Two)
```

在这里我们使用常量生成器定义类型为 `Digit` 的常量，等同于下面的代码：

```
const (
Zero = 0
One = 1
Two = 2
Three = 3
Four = 4
)
```

最后一部分代码：

```
const (
      p2_0 Power2 = 1 << iota
      _
      p2_2
      _
      p2_4
      _
      p2_6
   )

   fmt.Println("2^0:", p2_0)
   fmt.Println("2^2:", p2_2)
   fmt.Println("2^4:", p2_4)
   fmt.Println("2^6:", p2_6)


}
```

这个常量生成器的使用与第三部分的略有不同，首先你会留意到 `_` 符号，意思是跳过本次常量声明，其次说明iota的递增属性是可以用到表达式中的。

接下来让我们深入 `const` 块中一探究竟。对于 `p2_0` 来说， `iota` 的值是 `0``,p2_0`的值是 1 ,对于 p2_2 来说 iota的值是2， `p2_2` 的值是表达式 `1 << 2` 的运算结果，用二进制表示左移2位即 `00000100` ，相应十进制的值就是4。同样的道理， `p2_4` 的值是 `16` ， `p2_6` 的值是 `64` 。

如你所见， `iota` 的使用能大大提高开发效率！

执行 `constants.go` 后得到下面的输出：

```
$ go run constants.go

1476 3.1415926 1 2 2^0: 1 2^2: 4 2^4: 16 2^6: 64
```

Go 指针

Go支持指针！**指针**是内存地址，它能够提升代码运行效率但是增加了代码的复杂度，C程序员深受指针的折磨。在 `第2章` 中当我们讨论不安全的代码时，就使用过指针，这一节哦我们将深入介绍Go指针的一些难点。另外，当你足够了解原生Go指针时，其安全性大可放心。

使用指针时， `*` 可以获取指针的值，此操作成为**指针的解引用**， `*` 也叫取值操作符； `&` 可以获取非指针变量的地址，叫做取地址操作符。

> 通常来说，经验较少的开发者应该尽量少使用指针，因为指针很容易产生难以察觉的bug。

你可以创建一个参数为指针的函数：

```
func getPointer(n *int) {

}
```

同样，一个函数的返回值也可以为指针：

```
func returnPointer(n int) *int {

}
```

`pointers.go` 展示了如何安全地使用Go指针，该文件分为4部分，其中第一部分是：

```
package main

import "fmt"

func getPointer(n *int) {
   *n = *n * *n
}

func returnPointer(n int) *int  {
   v := n * n
   return &v
}
```

`getPointer()` 的作用是修改传递来的参数，而无需返回值。这是因为传递的参数是指针，其指向了变量的地址，所以能够将变量值的改变反映到原值上。

`returnPointer()` 的参数是一个整数，返回值是指向整数的指针，尽管这样看起来并没有什么用处，但是在第四章中，当我们讨论指向结构体的指针以及其他复杂数据结构时，你就会发现这种操作的优势。

`getPointer()` 和 `returnPointer()` 函数的作用都是求一个整数的平方，区别在于 `getPointer()` 使用传递来的参数存储计算结果，
而 `returnPointer()` 函数重新声明了一个变量来存储运算结果。

第二部分:

```
func main() {
   i := -10
   j := 25

   pI := &i
   pJ := &j

   fmt.Println("pI memory:", pI)
   fmt.Println("pJ memory:", pJ)
   fmt.Println("pI value:", *pI)
   fmt.Println("pJ memory:", *pJ)
```

`i` 和 `j` 是整数， `pI` 和 `pJ` 分别是指向 `i` 和 `j` 的指针， `pI` 是变量的内存地址， `*pI` 是变量的值。

第三部分:

```
*pI = 123456
*pI--
fmt.Println("i:",i)
```

这里我们使用指针 `pI` 改变了变量 `i` 的值。

最后一部分代码:

```
getPointer(pJ)
   fmt.Println("j:",j)
   k := returnPointer(12)
   fmt.Println(*k)
   fmt.Println(k)

}
```

根据前面的讨论，我们通过修改 `pJ` 的值就可以将改变反映到 `j` 上，因为 `pJ` 指向了 `j` 变量。我们将 `returnPointer()` 的返回值赋值给指针变量 `k` 。

运行 `pointers.go` 的输出是:

```
$ go run pointers.go

pI memory: 0xc420014088 pJ memory: 0xc420014090 pI value:
-10 pJ memory: 25 i: 123455 j: 625 144 0xc4200140c8
```

你可能对 `pointers.go` 中的某些代码感到困惑，因为我们在第六章才开始讨论函数及函数定义，可以去了解关于函数的更多信息。

时间与日期的处理技巧

本节你将学习到如何解析时间与日期字符串、格式化日期与时间、以你期望的格式打印时间与日期。你可能会觉得这部分内容没有意义，但是当你想要实现多任务同步或者从文本、用户读取日期时，就会发现这一节的作用。

Go自带一个处理时间与日期的神器-`time`包，这里将介绍几个实用的函数。

在学习如何将字符串解析为时间和日期之前，先看一段简单的代码`usingTime.go`以对`time`包有个简单的了解，代码分为三个部分，第一部分引入了我们准备使用的包:

```
package main

import (
   "fmt"
   "time"
)
```

第二部分：

```
func main() {
   fmt.Println("Epoch Time:", time.Now().Unix())
   t := time.Now()
   fmt.Println(t,t.Format(time.RFC3339))
   fmt.Println(t.Weekday(), t.Day(), t.Month(), t.Year())

   time.Sleep(time.Second)
   t1 := time.Now()
   fmt.Println("Time difference:", t1.Sub(t))
```

`time.Now().Unix()`返回UNIX时间（UNIX时间是计算了从00:00:00 UTC，1970年1月1日以来的秒数）。`Format()`能够将`time`类型的变量转换成其他格式，例如`RFC3339`格式。

你会发现`time.Sleep()`在本书中频繁出现，这是一种最简单的产生延时的函数。`time.Second`意思是1秒，如果你想产生10s的延迟，只需将 time.Second*10 即可。对于 time.Nanosecond 、 time.Microsecond 、 time.minute 、 time.Hour 是同样的道理。使用 time 包能够定义的最小时间间隔是1纳秒。最后， time.Sub()`函数能够得到两个时间之间的时间差。

第三部分：

```
formatT := t.Format("01 January 2006")
   fmt.Println(formatT)
   loc, _ := time.LoadLocation("Europe/Paris")
   LondonTime := t.In(loc)
   fmt.Println("Paris:",LondonTime)

}
```

我们使用 `time.Format` 定义了一个新的日期格式，并且得到指定时区的时间。

执行 `usingTime.go` 的输出如下：

```
$ go run usingTime.go

Epoch Time: 1547279979 2019-01-12 15:59:39.959594352 +0800 CST m=+0.000392272 2019-01-12T15:59:39+08:00 Saturday 12 January 2019 Time difference: 1.000820609s 01 January 2019 Paris: 2019-01-12 08:59:39.959594352 +0100 CET
```

现在你应该对 `time` 包有了一个基本的了解，是时候去深入了解 `time` 更多的功能了！

解析时间

将时间类型的变量转换成其他格式的时间与日期是非常简单的，但是当你想要将 `string` 类型转换成时间类型，以检查其是否是一个有效的时间格式时，就会比较麻烦。 `time` 包提供了 `time.parse()` 函数帮助你解析时间与日期字符串，将其转换成 `time` 类型。 `time.Parse()` 接收两个参数，第一个参数是你期望得到的时间格式，第二个参数是你要解析的时间字符串。第一个参数是Go与时间解析相关的一系列常量。

> 这些常量可以用来创建你期望的时间日期格式，具体介绍见 `https://golang.org./src/time/format.go` 。Go并没有定义一些时间格式像 `DDYYMM` 、 `%D%Y%M` 等，最开始你可能会觉得Go的这种处理方式很蠢，但是用多了后你肯定会喜欢上这种简洁的方式。

这些与时间处理相关的常量分别是， `15` 代表解析小时， `04` 代表解析分钟， `05` 解析秒，同时你可以使用 `PM` 将字符串中的字母转为大写， `pm` 转为小写。

> 开发者只需要将上面的常量按照字符串中日期的顺序排放就可以了。

其实你可以将 `time.Parse()` 的第一个参数看做正则表达式。

解析时间的代码示例

本节将介绍 `parseTime.go`，这段代码接收一个命令行参数，并将其由 `string` 类型转换成 `time` 类型。该代码分为三部分，仅是为了演示 `time.Parse()`，可能会因为你的输入格式不正确而报错。

第一部分：

```
package main

import (
   "fmt"
   "os"
   "path/filepath"
   "time"
)
```

第二部分：

```
func main() {
   var myTime string

   if len(os.Args) != 2 {
      fmt.Printf("Usage: %s string\n",
         filepath.Base(os.Args[0]))
      os.Exit(0)
   }

   myTime = os.Args[1]
```

最后一部分是展现神奇的代码：

```
d,err := time.Parse("15:04",myTime)
   if err == nil {
      fmt.Println("Full",d)
      fmt.Println("Time", d.Hour(), d.Minute())
   } else {
      fmt.Println(err)
   }
}
```

可以看到，为了解析一个包含小时和分钟的字符串，你需要使用常量来构建格式"15:04"。返回值 `err` 能够告诉你我们的解析是否成功。

执行 `parseTime.go` 后我们得到：

```
$ go run parseTime.go

usage: parseTime string

exit status 1

$ go run parseTime.go 12:10

Full 0000-01-01 12:10:00 +0000 UTC Time 12 10
```

可以看到Go将整个时间（Full 0000-01-01 12:10:00 +0000 UTC）都打印出来了，这是因为 `time.Parse()` 返回值是时间类型的变量。如果你只关心具体时间而不是日期，你应该只打印你关心的 `time` 变量部分。

如果你没有按照Go规定的时间常量来格式化你的字符串，例如使用 `22:04` 作为第一个参数，你就会得到下面的错误信息：

```
$ go run parseTime.go

parsing time "12:10" as "22:04" : cannot parse ":10" as "2"
```

或者，如果你使用了用来处理月份的 `11` 来格式化，可能得到下面的错误信息：

```
$ go run parseTime.go 12:10

parsing time "12:10": month out of range
```

解析日期

在本节中你将学习如何将字符串转为日期，同样是使用 `time.Parse()` 。

Go解析日期的常量是： `Jan` 用来解析月份（英文月份简写）， `2006` 用来解析年， `02` 用来解析天， `Mon` 用来解析周几（如果是 `Monday` ，那就是周几的英文全称），同样如果你使用 `January` 而不是 `Jan` ，你将会得到月份的英文全称而不是三个字母的简写。

解析日期的代码示例

本节的代码 `parseDate.go` ，分两部分讲解。

第一部分：

```
package main

import (
   "fmt"
   "os"
   "path/filepath"
   "time"
)

func main() {
   var myDate string

   if len(os.Args) != 2 {
      fmt.Printf("Usage: %s date\n",
         filepath.Base(os.Args[0]))
      os.Exit(0)
   }

   myDate = os.Args[1]
```

第二部分：

```
d,err := time.Parse("02 January 2006",myDate)
   if err == nil {
      fmt.Println("Full",d)
      fmt.Println("Time", d.Day(), d.Month(), d.Year())
   } else {
      fmt.Println(err)
   }
}
```

如果你想要在月份和年之间加入"-"，只需要将格式“02 January 2006“改成“02 January-2006”即可。

执行 `parseDate.go` 你将得到下面的输出：

```
$ go run parseDate.go

usage: parseDate string

$ go run parseDate.go "12 January 2019"

Full 2019-01-12 00:00:00 +0000 UTC Time 12 January 2019
```

格式化时间与日期

在这一节中你将处理既有时间又有日期的字符串，这种格式的时间在web服务器中最常见到，例如**Apache**，**Nginx**等。由于目前还没有讲到文件IO，所以这里我将文本硬编码写到程序中，这不会影响程序的功能。

本节代码 `timeDate.go` 将分3部分展示，其中第一部分：

```
package main

import (
   "fmt"
   "regexp"
   "time"
)

func main() {

   logs := []string{"127.0.0.1 - - [16/Nov/2017:10:49:46 +0
      "127.0.0.1 - - [16/Nov/2017:10:16:41 +0200] \"GET /CV
      "127.0.0.1 200 9412 - - [12/Nov/2017:06:26:05 +0200]
      "[12/Nov/2017:16:27:21 +0300]",
      "[12/Nov/2017:20:88:21 +0200]",
      "[12/Nov/2017:20:21 +0200]",
   }
```

由于我们不能确定数据的格式，所以样本数据尽量覆盖不同格式的数据，包括像 `"[12/Nov/2017:20:21 +0200]"` 这种不完整的数据， `[12/Nov/2017:20:88:21 +0200]` 这种本身存在错误的数据（秒数是88）。

第二部分代码：

```
for _, logEntry := range logs {
   r := regexp.MustCompile(`.*\[(\d\d\/\w+/\d\d\d\d:\d\d:\d
   if r.MatchString(logEntry) {
      match := r.FindStringSubmatch(logEntry)
```

我们使用正则表达式来匹配正确的时间格式，在拿到时间字符串后，剩下的任务就交给 `time.Parse()` 好了。

最后一部分代码：

```
dt, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1
         if err == nil {
            newFormat := dt.Format(time.RFC850)
            fmt.Println(newFormat)
         } else {
            fmt.Println("Not a valid date time format!")
         }
      } else {
         fmt.Println("Not a match!")
      }
   }
}
```

一旦正则表达式匹配到了时间字符串， `time.Parse()` 就会去解析判断其是否是一个有效的时间。如果是， `timeDate.go` 就会将时间以**RFC850**格式打印出来。

执行 `timeDate.go` ，你将得到下面的输出：

```
$ go run timeDate.go

Thursday, 16-Nov-17 10:49:46 +0200 Thursday, 16-Nov-17
10:16:41 +0200 Sunday, 12-Nov-17 06:26:05 +0200 Sunday, 12-
Nov-17 16:27:21 +0300 Not a valid date time format! Not a match!
```

延展阅读

可以阅读下面的资源：

- 阅读官方的 `time` 包：https://golang.org/pkg/time
- 官方讲解正则表达式的包 `regexp` ：https://golang.org/pkg/regexp

练习

- 使用iota编写常量生成器表示数字0-4
- 使用iota编写常量生成器表示一周的天数
- 编写程序将数组转为map
- 自己动手编写parseTime.go，不要忘记写测试
- 编写timeDate.go使其能够处理两种格式的时间与日期
- 自己动手编写parseDate.go

本章小结

本章你学习了很多有趣的Go知识，包括映射（map）、数组、切片、指针、常量、循环以及Go处理时间与日期的技巧。学到现在，你应该能够理解为什么切片要优于数组。

下一章将介绍构建与使用组合类型的知识，主要是使用 `struct` 关键字创建的结构体，之后会讨论 `string` 变量和元组。

另外，下一章还会涉及到正则表达式和模式匹配，这是一个比较tricky的主题，不仅针对Go，对其他所有语言都是一样的，合理地使用正则表达式能够大大简化你的工作，是非常值得学习一下的。

你还会了解关于 `switch` 关键字的知识，以及使用 `strings` 包处理**UTF-8**字符串的技巧。

组合类型的使用

上一章我们讨论了数组、切片、映射、指针、常量、`for`循环、`range`关键字的使用，以及处理始建于日期的技巧。这一章我们将探索更加高级的Go特性，比如元组和字符串，标准库`strings`，`switch`语句，以及使用`struct`关键字创建结构体。另外一部分比较重要的内容使用Go实现正则表达式以及模式匹配。完成上面的章节后，我们将实现一个简单的K-V存储。

下面是本章的内容概览：

- Go结构体和`struct`关键字
- Go元组
- Go 字符串，runes，字节切片，以及字符串字面量
- Go的正则表达式
- Go的模式匹配
- `switch`语句
- 关于标准库`strings`的使用
- 计算高精度的**PI**值
- 实现一个**K-V存储**

关于组合类型

尽管Go提供的类型是非常方便，快速和灵活的，但是不可能满足开发者所有的需求。Go通过提供结构体来解决此问题，开发者可以实现自定义类型。另外，Go还支持元组操作，意思是允许函数返回多个值，这些值并不需要提前声明结构体存放。

结构体

数组，切片，映射这类数据结构的确很有用，但是它们不能将多个值聚合到同一个地方。如果你想要将不同类型的不同变量聚合到一个地方以创建一个新的类型，那么结构体会满足你的需要。结构体重的不同元素叫做**结构体的字段**。

本章我们以一个简单的结构体为例来讲解，上一章的 `sortSlie.go` 中我们定义过一个结构体:

```
type sStructure struct {
person string
height int
weight int
}
```

其实结构体的字段通常是大写字母开头，这个原因我们将在第六章中具体说明，这个习惯会贯穿本书接下来的内容。

这个结构体有三个字段，分别是 `person,height,weight` 。现在，你可以创建一个类型为 `astructure` 的变量了:

```
var s1 aStructure
```

另外，我们使用 `.` 操作符来访问结构体中的字段，例如我们可以使用 `s1.person` 来获取 `person` 字段的值。

一个**结构体字面量**可以这样定义：

```
p1 := aStructure{"fmt",12,-2}
```

然而你可能觉得记住结构体中字段的声明顺序实在是太难了，莫慌，我们有另外一种定义结构体字面量的方式:

```
p1 := aStructure{weight: 12, height:-2}
```

这种方式你无须初始化结构体中的所有字段。

现在你已经了解了结构体的基本操作，让我们尝试一些实战性更强的代码吧，这部分代码保存在 `structures.go` 中，分为三部分。

第一部分代码：

```
package main

import "fmt"

func main(){
   type XYZ struct {
      X int
      Y int
      Z int
   }

   var s1 XYZ
   fmt.Println(s1.Y, s1.Z)
```

> Go结构体通常来说是定义在 `main()` 函数外面的，这样在整个Go package中可以拥有全局的属性，除非你不想让你的结构体在任何地方都被访问到，这种情况下你可以定义局部的结构体。

第二部分代码：

```
p1 := XYZ{23,12, -2}
p2 := XYZ{Z:12,Y:13}

fmt.Println(p1)
fmt.Println(p2)
```

我们使用两种定义结构体字面量的方式定义了 `p1` 与 `p2` ,并且打印出来。

最后一部分：

```
pSlice := [4]XYZ{}
   pSlice[2] = p1
   pSlice[0] = p2

   fmt.Println(pSlice)

   p2 = XYZ{1,2,3}
   fmt.Println(pSlice)
}
```

最后一部分代码中，我们创建了一个结构体数组 `pSlice` ，当你将一个结构体分配给结构体数组，那么这个结构体就会被深拷贝至这个数组，这意味着改变原结构体是对数组中的结构体没有影响的，从下面的打印输出中我们能够看出来：

```
$ go run structures.go

0 0 {23 12 -2} {0 13 12} [{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]

[{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]
```

注意，结构体中字段的定义顺序是有意义的，简单来说，就算两个结构体拥有相同的字段，但是字段的声明顺序不同，那么这两个结构体也是不相等的。

从程序输出中我们能够知道，结构体中的变量是初始化为其类型的零值。

结构体指针

之前的章节我们已经讨论过指针，本节我们会继续指针的讨论并且提供一个与**结构体指针**的示例，`pointerStruct.go`。

第一部分代码：

```
package main

import "fmt"

type myStructure struct {
   Name string
   Surname string
   Height int32
}

func createStructure(n,s string, h int32) *myStructure {
   if h > 300 {
      h = 0
   }
   return &myStructure{n, s, h}
}
```

`相对于直接初始化一个结构体变量，createStructure()`提供了一种更加优雅的方法，不仅能够检查赋予结构体字段的值的正确性和有效性，而且当出问题之后，能够很快找到问题所在。注意一下该函数的命名，命名为`NewStructure()`是比较妥当的。

> 对于有C/C++背景的开发者来说，Go函数返回局部变量的内存地址是很容易理解的，所以Go的这种设计皆大欢喜！

第二部分：

```
func retStructure(n,s string, h int32) myStructure {
   if h > 300 {
      h = 0
   }
   return myStructure{n, s, h}
}
```

这部分实现了`createStructure()`的无指针版本，两个函数的效果都是一样的，所以选择哪一种就看开发者的喜好了。其实这两个函数命名为`NewStructurePointer()`和`NewStructure()`是比较规范的。

最后一部分代码：

```
func main() {
   s1 := createStructure("Mihalis","Tsoukalos",123)
   s2 := retStructure("Mihalis","Tsoukalos",123)
   fmt.Println((*s1).Name)
   fmt.Println(s2.Name)
   fmt.Println(s1)
   fmt.Println(s2)
}
```

执行 `pointerStruct.go` 我们得到下面的输出：

```
$ go run pointerStruct.go
Mihalis
Mihalis
&{Mihalis Tsoukalos 123}
{Mihalis Tsoukalos 123}
```

通过输出你能够发现 `createStructure()` 和 `retStructure()` 的主要区别，就是前者返回指向结构体的指针，这意味着你在想获取结构体内字段值的时候，就必须先解引用，有些人会觉得这种操作不太优雅。

> 结构体是一个非常重要的概念，在实战中会经常用到，它能够帮你把很多不同类型的变量聚合到一起，然后依次处理。

使用new关键字

Go支持使用 `new` 关键创建新的对象，必须要注意的是，这种方式返回的对象的指针！

你可以创建一个 `aStructure` 类型的变量：

```
pS := new(aStructure)
```

执行上述代码后，你得到的是值为 `nil` 的对象，并没有初始化。

> `new` 和 `make` 最大的区别就是： `new` 返回的是空的内存地址，即没有做初始化。另外， `make` 仅可以用来创建映射，切片和通道，而且并不是返回指针。

下面的代码将会创建一个指向切片的指针，并且值为 `nil` ：

```
sP := new([]aStructure)
```

元组

严格来讲，元组是由多个部分组成的有序列表，最重要的是Go本身不支持元组类型，虽然Go官方并不关心元组，但实际上它提供了元组的某些操作。

有趣的是我们在第一章已经接触过Go的元组操作，像下面的这种操作，使用一条语句获取两个返回值：

```
min,_ := strconv.ParseFloat(arguments[1], 64)
```

`tuples.go` 中的代码将会分成三部分来解释Go的元组，请注意下面代码中的函数将返回值以元组的形式返回。

第一部分：

```
package main

import "fmt"

func retThree(x int) (int, int, int) {
      return 2 * x, x*x,-x
}
```

`retThree()` 函数返回了包含三个整数元素的元组，这种能力使得函数能够返回一组数据，而无需将它们聚合到结构体重或者返回一个结构体变量。

在第六章你将会学习到如何给函数的返回值命名，这是一个非常方便的特性。

第二部分代码：

```
func main() {
   fmt.Println(retThree(10))
   n1, n2, n3 := retThree(20)
   fmt.Println(n1,n2,n3)
```

这里我们使用了两次 `retThree()` 函数。第一次我没让你并没有将其返回值保存，第二次使用三个变量保存返回值，在Go的术语中这叫做元组赋值，看到这里，你是不是有了一种 `Go支持元组！` 的错觉。如果有些返回值你并不关心，可以使用 `_` 操作符忽略掉它们。要知道，在Go的代码里声明了但是未使用的代码是会导致编译错误的。

第三部分代码：

```
n1, n2 = n2, n1
   fmt.Println(n1, n2, n3)

   x1, x2, x3 := n1*2,n1*n1, -n1

   fmt.Println(x1,x2,x3)
}
```

可以看到，依靠这种元组操作，我们无需借助 `temp` 变量就可以实现两个数字的交换。

执行 `tuples.go` 可得到如下输出：

```
$ go run tuples.go

20 100 -10

40 400 -20

400 40 -20

800 160000 -400
```

正则表达式与模式匹配

模式匹配在Go中占有重要的地位，是能够根据正则表达式查询出特定的字符集的技术。模式匹配成功后你就可以提取出你想要的字符串，并对其进行替换、删除等进一步操作。

Go标准库中已经帮你设计好了用于正则表达式的包-`regexp`，我们将会在下面深入讨论。

> 当你在代码中使用正则表达式时，定义符合条件的正则表达式是重中之重。

理论知识

每一个正则表达式都通过一种叫做**有限自动机**的状态转换图表，被编译成识别器。一个有限自动机可以使确定性的也可以是不确定性的。不确定性的意思是一种输入可能会产生多种输出状态。识别器是一种程序，它接收一个字符串 `x` 并且能够分辨出 `x` 是何种语言写的。

语法能够以一种格式化的语言为字符串生成一系列产生式规则，产生式规则描述了根据词法生成有效字符串的方法。语法器只能描述字符串的结构，不能描述一个字符串的含义或者描述该字符串能在什么上下文中用来做什么。要注意，语法是定义或者使用正则表达式的核心。

> 尽管正则表达式可以用来解决棘手的问题，但是不要任何问题都使用正则表达式，一定要在正确的地方选择正确的工具！

接下来的这一小节将会展示三个正则表达式和模式匹配的例子。

简单的正则表达式示例

本节你将学习从一行文本中选择特定列的技巧。为了让这事儿更有趣，我还会教你如何逐行地读取文本，文本I/O是第8章的内容，你可以参考一下。

本节代码示例 `selectColumn.go` 将会分为4部分阐述，需要至少两个命令行参数进行操作；第一个是你想选择的第几列，第二个是你要处理的文本所在路径。当然，你可以处理多个文件， `selectColumn.go` 将会帮你一个一个处理。

第一部分代码：

```
package main

import (
   "bufio"
   "fmt"
   "io"
   "os"
   "strconv"
   "strings"
)

func main() {
   arguments := os.Args
   if len(arguments) < 2 {
      fmt.Printf("usage: use selectColumn column [file1] [f
      os.Exit(1)
   }

   temp , err := strconv.Atoi(arguments[i])
   if err != nil  {
      fmt.Printf("column value is not an integer", temp)
      return
   }

   column := temp
   if column < 0 {
      fmt.Println("Invalid column value")
      os.Exit(1)
   }
```

首先判断输入的参数数量是否满足要求（len(arguments) < 2），然后需要判断输入的column值是否是合法的，即 `>0` 。

第二部分代码：

```
for _, fileName := range arguments[2:]{
   fmt.Println("\t\t",fileName)
   f, err := os.Open(fileName)
   if err != nil {
      fmt.Printf("error opening file %s\n",err)
      continue
   }
   f.Close()
```

这部分代码判断所给的文件是否存在并且可读，`os.Open()`用来读取文本文件的。记住，UNIX系统的文件有些你是没有读权限的。

第三部分代码：

```
r := bufio.NewReader(f)
for {
   line, err := r.ReadString('\n')
   if err == io.EOF {
      break
   } else if err != nil{
      fmt.Printf("error reading file %s\n",err)
   }
```

你将在第8章中学习到`bufio.ReadString()`函数的机制是读取到参数中的字符就停止，所以`bufio.ReadString('\n')`的意思就是逐行读取文本文件，因为`\n`在UNIX系统中是换行符。`bufio.ReadString()`函数返回的是字节切片。

最后一部分代码是：

```
data := strings.Fields(line)
         if len(data) > column {
            fmt.Println(data[column-1])
         }
      }

   }

}
```

这个程序背后的逻辑是很简单的：逐行读取文本并选择你想要的列。然而，由于你不能确定当前行是否拥有你想要的列，所以你会在输出之前检查一下。其实这是一个非常简单的模式匹配，因为空格天然地充当了每一行中的分隔符。

如果你想要探究更多切分一行文本的技巧，可以查看 `strings.Fields()` 的源码，你就会发现该函数基于空白分隔符将一行文本分割成一个字符串切片，空白分隔符是由 `unicode.IsSpace()` 定义的。

执行 `selectColumn.go` 的输出如下：

```
$ go run selectColumn.go 15 /tmp/swtag.log /tmp/adobegc.log
AdobeDjkdgj
Successfully
Initializiing
Stream
*AdobeGC
```

高级的正则表达式示例

本节你将学习到如何在Apache web服务器的日志文件中匹配特定格式的时间与日期字符串。同时，你也会了解到将不同格式的时间与日期写入日志文件中。与上一节一样，我们需要逐行读取Apache日志文件。

本节的代码 `changeDT.go` 将分为4部分展示，可以发现 `changeDT.go` 是第三章中 `timeDate.go` 的升级版本，只不过是使用两个正则表达式匹配不同的时间与日期。

> 要注意的是不要试图在程序的第一个版本就达到尽善尽美，最好的方法就是小版本快速迭代。

第一部分代码：

```
package main

import (
   "bufio"
   "fmt"
   "io"
   "os"
   "regexp"
   "strings"
   "time"
)

func main() {
   arguments := os.Args
   if len(arguments) ==1 {
      fmt.Println("Please provide one text file to process"
      os.Exit(1)
   }
   fileName := arguments[1]
   f, err := os.Open(fileName)
   if err != nil {
      fmt.Printf("error opening file %s ",err)
      os.Exit(1)
   }
   defer f.Close()

   notAmatch := 0
   r := bufio.NewReader(f)
   for{
      line, err := r.ReadString('\n')
      if err == io.EOF {
         break
      } else if err != nil{
         fmt.Printf("error reading file %s,",err)

      }
```

首先我们要打开要读取的文件，并逐行读取其内容。变量 `notAmatch` 存储输入文件中不匹配两个正则表达式的条目。

第三部分代码：

```
r1 := regexp.MustCompile(`.*\[(\d\d\/\w+/\d\d\d\d:\d\d:\d\d
if r1.MatchString(line) {
   match := r1.FindStringSubmatch(line)
   d1, err := time.Parse("02/Jan/2006:15:04:05 -0700", matc
   if err == nil {
      newFormat := d1.Format(time.Stamp)
      fmt.Printf(strings.Replace(line,match[1],newFormat,1)
   } else {
      notAmatch++
      continue
   }

}
```

可以看出只要代码执行到 `if` 中，就会执行 `continue` ，这意味着程序会继续执行本代码块中的逻辑。第一个正则表达式会匹配格式为 `21/Nov/2017:19:28:09 +0200` 的时间与日期字符串。

函数 `regexp.MustCompile()` 与 `regex.Compile()` 作用相同，只不过在解析失败时会触发panic。这种情况下你只能实现一种匹配，所以就要使用 `regexp.FindStringSubmatch()` 。

第三部分代码：

```
r2 := regexp.MustCompile(`.*\[(\w+\-\d\d-\d\d:\d\d:\d\d:\d\
   if r2.MatchString(line) {
      match := r2.FindStringSubmatch(line)
      d1, err := time.Parse("Jan-02-06:15:04:05 -0700", mat
      if err == nil {
         newFormat := d1.Format(time.Stamp)
         fmt.Print(strings.Replace(line, match[1], newForma
      } else {
         notAmatch++
      }
      continue
   }
}
```

匹配的第二种时间格式是 `Jun-21-17:19:28:09 +0200` ，尽管本程序实现了两种格式的匹配，但当你掌握正则表达式之后，你可以实现任意形式的匹配。

最后一部分代码将会打印出日志中没有匹配的条目数量：

```
    fmt.Println(notAmatch, "lines did not match!")
    }
```

执行 `ChangDT.go` 后得到下面的输出：

```
$ go run changDT.go
```

正则匹配IPv4地址

一个**IPv4**地址被 `.` 号分成了四部分，每一部分都由8位二进制组成，十进制范围是0-255，即二进制的范围是 `00000000` - `11111111` 。

> *IPv6*的格式更加复杂，本节的代码不适用于IPv6。

本节的代码 `findIPv4.go` 将分为4个部分，第一部分：

```
package main

import (
   "bufio"
   "fmt"
   "io"
   "net"
   "os"
   "path/filepath"
   "regexp"
)

func findIp(input string) string {
   partIp := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])
   grammer := partIp+"\\."+partIp+"\\."+partIp+"\\."+partIp
   matchMe := regexp.MustCompile(grammer)
   return matchMe.FindString(input)
}
```

`findIPv4.go` 的代码相对之前的复杂一些，所以需要使用的包比较多。

这部分代码定义的正则表达式，能够匹配IPv4地址，是整个程序的核心，如果你定义的正则表达式不正确，将永远找不到你想要的IPv4地址！

代码中的正则表达式稍后我会解释，在此之前必须了解IPv4地址的结构，比如IPv4的点分十进制表示方法以及每一部分都不能超过255，只有这样你能写出恰当的正则表达式。

正则表达式 `partIp` 定义了在一个IPv4地址中，每一部分可能出现的情况，比如可能是250-255的三位数，也可能是200-249，100-199，或者10-99两位数，0-9的个位数，以上的情况必须都考虑在内。

`grammer` 变量定义了IPv4地址的四个部分，每一部分必须匹配 `partIp` 。

> `findIPv4.go` 可以在任意文本文件中帮助你找到IPv4地址。

如果你有特殊需求，比如要搜索特定的IP地址，直接修改 `findIPv4.go` 的正则表达式即可。

第二部分：

```
func main() {
   arguments := os.Args
   if len(arguments) < 2 {
      fmt.Printf("usage: %s logfile\n",filepath.Base(argume
      os.Exit(1)
   }
   for _, filename := range arguments[1:] {
      f,err := os.Open(filename)
      if err != nil {
         fmt.Printf("error openning file %s\n",err)
         os.Exit(-1)
      }
      defer f.Close()
```

首先通过 `os.Args` 获取搜索的文件，然后检查命令行参数的长度是否符合要求。接下来通过一个for`循环迭代处理。

第四部分代码：

```
r := bufio.NewReader(f)
for {
   line,err := r.ReadString('\n')
   if err == io.EOF {
      break
   } else  if err != nil {
      fmt.Printf("error openning file %s\n",err)
      break
   }
```

类似于 `selectColumn.go` 的代码，我们使用 `bufio.ReadString*()` 逐行读取文本。

最后一部分：

```
ip := findIp(line)
            trail := net.ParseIP(ip)
            if trail.To4() == nil {
               continue
            } else {
               fmt.Println(ip)
            }
         }
      }
   }
```

对逐行读取的文本执行 `findIp()` 函数， `netParseIP()` 会再次确保获取的是有效的IPv4地址。

执行 `findIPv4.go` 得到下面的输出：

```
$ go run findIPv4.go auth.log

116.112.10.1 151.1.51.5 192.168.1.1
```

其实 `findIPv4.go` 会打印出很多重复的行（如果文件中有很多重复的IPv4地址）。我们可以配合UNIX命令对输出进一步处理，可以获得更加清晰直观的结果：

```
go run findIPv4.go auth.log | sort -rn | uniq -c | sort -rn

38 191.168.1.1

22 182.12.5.1

....

9 10.18.2.64
```

`sort -rn` 将 `findIPv4.go` 的输出作为输入，排序后倒序输出， `uniq -c` 计算重复ip的出现次数，最后 `sort -rn` 按照重复ip的出现次数倒叙输出。

> 再次强调， `findIPv4.go` 的核心是正则表达四的实现。如果正则表达式定义错误了，就会匹配不到你想要的数据，或者匹配到错误的数据。

字符串

严格来说，Go字符串并不是组合类型，但是Go提供了很多直接操作字符串的函数，*第3章*提到过Go字符串是值类型，并非像C字符串需要靠指针实现。另外，Go默认支持UTF-8字符串，处理Unicode编码的字符就会非常方便。下面的小节也会讲到字符、rune、字节之间的区别，以及字符串与字符串字面量的不同。

Go字符串实际上是一个字节切片，可以存储任意类型、任意长度的字节。

可以像下面一样定义一个字符串：

```
const sLiteral = "\x99\x42\x32"
```

也可以使用字符串字面量：

```
s2 := "rfsdf"
```

可以使用 `len()` 得到字符串的长度。

`strings.go` 将会分4部分展示一些Go字符串的标准操作。

第一部分：

```
package main

import (
   "fmt"
)

func main() {
   const sLiteral= "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9
   fmt.Println(sLiteral)
   fmt.Printf("x: %x\n", sLiteral)

   fmt.Printf("sLiteral length: %d\n", len(sLiteral))
```

每一个 `\xAB` 都代表 `sLiteral` 的一个字符，所以调用 `len()` 就会得到 `sLiteral` 的字符数量。 `%x` 会得到 `\xAB` 中的 `AB` 部分。

第二部分代码：

```
for i := 0; i < len(sLiteral); i++ {
   fmt.Printf("%x ", sLiteral[i])
}
fmt.Println()

fmt.Printf("q: %q\n", sLiteral)
fmt.Printf("+q: %+q\n", sLiteral)
fmt.Printf(" x: % x\n", sLiteral)

fmt.Printf("s: As a string: %s\n", sLiteral)
```

如代码所示，你可以像操作一个切片那样去操作字符串。使用 `%q` 作为字符串格式化参数，可以安全地打印出带双引号的字符串， `%+q` 可以保证输出是ASCII格式。

最后， `% x` （注意%与x之间的空格）将会在输出的字符之间加上空格，如果你想打印字符串格式，就要使用 `%s` 。

第三部分代码：

```
s2 := "€£³"
for x, y := range s2 {
   fmt.Printf("%#U starts at byte position %d\n", y, x)
}

fmt.Printf("s2 length: %d\n", len(s2))
```

在这里定义了字符串 `s2` ，内容是3个unicode字符。使用 `%#U` 可以打印出 `U+0058` 格式的字符， `range` 关键字能够迭代包含Unicode字符的字符串，这样就可以逐个处理Unicode字符。

`len(s2)` 的输出可能会令你困惑，解释一下， `s2` 包含的是Unicode字符，Unicode字符的字节数量是大于该字符串中的元素数量的，
而 `len()` 函数计算的是字节数量，所以 `len(s2)` 的值是7而不是3。

最后一部分：

```
const s3= "ab12AB"
   fmt.Println("s3:", s3)
   fmt.Printf("x: % x\n", s3)

   fmt.Printf("s3 length: %d\n", len(s3))

   for i := 0; i < len(s3); i++ {
      fmt.Printf("%x ", s3[i])
   }
   fmt.Println()

}
```

执行 `strings.go` 的输出是：

```
go run strings.go
```

```
B2UP5#P)
```

```
x: 9942325550352350299c
```

```
sLiteral length: 10
```

```
99 42 32 55 50 35 23 50 29 9c
```

```
q: "\x99B2UP5#P)\x9c"
```

```
+q: "\x99B2UP5#P)\x9c"
```

```
x: 99 42 32 55 50 35 23 50 29 9c
```

```
s: As a string: B2UP5#P)
```

```
U+20AC '€' starts at byte position 0
```

```
U+00A3 '£' starts at byte position 3
```

```
U+00B3 '³' starts at byte position 5
```

```
s2 length: 7
```

```
s3: ab12AB
```

```
x: 61 62 31 32 41 42
```

```
s3 length: 6
```

```
61 62 31 32 41 42
```

在不清楚Unicode和UTF-8机制的情况下，你可能会对本节的输出干到困惑，不必恐慌，因为这些在平时的开发中很少使用，基本的 `fmt.Printf()` 和 `fmt.Println()` 就能满足你的需求。

rune是什么？

rune是一个类型为 `int32` 的值，因此他主要用来代表一个Unicode码点。Unicode码点是一个代表Unicode字符的数值。

> **NOTE:**你可以认为字符串是一系列rune的集合

rune字面量实际上是一个用单引号括起来的字符，并且与与Unicode码点的概念相关联。

`rune.go` 将分两部分阐述rune的使用，第一部分是：

```
package main

import (
   "fmt"
)

func main() {
   const r1 = '€'
   fmt.Println("(int32) r1:", r1)
   fmt.Printf("(HEX) r1: %x\n", r1)
   fmt.Printf("(as a String) r1: %s\n", r1)
   fmt.Printf("(as a character) r1: %c\n", r1)
```

首先定义了一个rune字面量 `r1` ,然后使用不同的方式去打印，分别是int32、十六进制、字符串、字符，最终你会发现使用字符格式打印出的与定义 `r1` 的值相同。

第二部分：

```
fmt.Println("A string is a collection of runes:", []byte("M
   aString := []byte("Mihalis")
   for x, y := range aString {
      fmt.Println(x, y)
      fmt.Printf("Char: %c\n", aString[x])
   }
   fmt.Printf("%s\n", aString)
}
```

显而易见，**字节切片**实际上就是一系列runes的集合，并且如果你使用 `fmt.Println()` 打印字节切片，结果很可能不会符合你的预期。 `fmt.Printf()` 语句结合 `%c` 可以将runes转换为字符输出；如果想要以字符串的形式输出字节数组，应使用 `fmt.Printf()` 结合 `%s` 。

执行runes.go得到下面的输出：

> go run runes.go

```
(int32) r1: 8364
(HEX) r1: 20ac
(as a String) r1: %!s(int32=8364)
(as a character) r1: €
A string is a collection of runes: [77 105 104 97 108 105 1
0 77
Char: M
1 105
Char: i
2 104
Char: h
3 97
Char: a
4 108
Char: l
5 105
Char: i
6 115
Char: s
Mihalis
```

最后，举一个产生 `illegal rune literal` 错误的例子：在导包的时候使用单引号。

> $ cat a.go
>
> package main
>
> import (
>
> 'fmt'
>
>)
>
> func main(){
>
> }
>
> $ go run a.go
>
> package main:a.go:4:2: illegal rune literal

关于Unicode的包

Go提供的 `unicode` 标准库提供了很多便捷的函数，其中 `unicode.IsPrint()` 能帮助你判断字符串的某一部分是否能够以rune的类型打印出来。接下来将会在代码 `unicode.go` 中分两部分展示该函数的用法。

第一部分：

```
package main

import (
   "fmt"
   "unicode"
)

func main() {
   const sL= "\x99\x00ab\x50\x00\x23\x50\x29\x9c"
```

第二部分：

```
for i := 0; i < len(sL); i++ {
      if unicode.IsPrint(rune(sL[i])) {
         fmt.Printf("%c\n", sL[i])
      } else {
         fmt.Println("Not printable!")
      }
   }
}
```

`unicode.IsPrint()` 函数将检查字符串 `sL` 的每个元素是否是rune类型，如果是的话将返回 `true` 否则返回false。如果你需要更多操作Unicode字符的方法，可以参考官方 `unicode` 包的介绍。

执行 `unicode.go` 将会打印：

> $ go run unicode.go

```
Not printable!
Not printable!
a
b
P
Not printable!
#
P
)
Not printable!
```

关于字符串处理的包

Go在 `strings` 包里面提供了很多操作UTF-8字符串的强大工具，大部分的工具将会在 `useStrings.go` 介绍，与文件IO相关的 操作将会在*第8章*详细介绍。

`useStrings.go` 的第一部分代码：

```
package main

import (
   "fmt"
   s "strings"
   "unicode"
)

var f = fmt.Printf
```

> `s "strings"` 的导包方式是非常推荐的，意味着我们给strings包起了一个简短的包名，这样就可以使用 `s.FunctionName()` 而不必使用 `strings.FunctionName()` 。

当你频繁使用某个包中的函数的时候，也可以将该函数赋值给一个变量，像上面的代码一样， `var f = fmt.Printf` ，接下来就可以用 `f` 代替 `fmt.Printf()` 。要注意的是不要过度使用该特性，不然你的代码可读性会变得很差。

第二部分代码：

```
unc main() {
   upper := s.ToUpper("Hello there!")
   f("To Upper: %s\n", upper)
   f("To Lower: %s\n", s.ToLower("Hello THERE"))

   f("%s\n", s.Title("tHis wiLL be A title!"))

   f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHAlis"))
   f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHAli"))
```

这段代码提供了很多玩转字符串的例子， `strings.EqualFold()` 函数能够判断两个字符串是否相同。

第三部分代码：

```
f("Prefix: %v\n", s.HasPrefix("Mihalis", "Mi"))
f("Prefix: %v\n", s.HasPrefix("Mihalis", "mi"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "is"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "IS"))

f("Index: %v\n", s.Index("Mihalis", "ha"))
f("Index: %v\n", s.Index("Mihalis", "Ha"))
f("Count: %v\n", s.Count("Mihalis", "i"))
f("Count: %v\n", s.Count("Mihalis", "I"))
f("Repeat: %s\n", s.Repeat("ab", 5))

f("TrimSpace: %s\n", s.TrimSpace(" \tThis is a line. \n"))
f("TrimLeft: %s", s.TrimLeft(" \tThis is a\t line. \n", "\n
f("TrimRight: %s\n", s.TrimRight(" \tThis is a\t line. \n",
```

`strings.Count()` 函数能够计算第二个参数（字符串类型），在第一个参数（字符串类型）中出现的次数；`strings.HasPrefix()` 函数判断字符串是否是以某字符串开头，如果是的话便返回 `true` ;同样，`strings.HasSuffix()` 判断字符串是否以某字符串结尾。

第四部分代码：

```
f("Compare: %v\n", s.Compare("Mihalis", "MIHALIS"))
f("Compare: %v\n", s.Compare("Mihalis", "Mihalis"))
f("Compare: %v\n", s.Compare("MIHALIS", "MIHalis"))

f("Fields: %v\n", s.Fields("This is a string!"))
f("Fields: %v\n", s.Fields("Thisis\na\tstring!"))

f("%s\n", s.Split("abcd efg", ""))
f("%s\n", s.Replace("abcd efg", "", "_", -1))
f("%s\n", s.Replace("abcd efg", "", "_", 4))
f("%s\n", s.Replace("abcd efg", "", "_", 2))
```

这部分展示了 `strings` 比较高级的一些函数。`strings.Split()` 能够以特定字符分割字符串，并返回一个**字符串切片**。

`strings.Compare()` 函数用于比较两个字符串是否相等，相等返回 `true` ，否则返回1或者-1（strings.Compare(a,b),如果ab,意思是出现在字母表中的顺序，译者加）。

`strings.Fields()` 按照空格将字符串分割。

> `strings.Split()` 非常有用，你迟早会在自己的程序中用到它！

最后一部分代码：

```
lines := []string{"Line 1", "Line 2", "Line 3"}
   f("Join: %s\n", s.Join(lines, "+++"))

   f("SplitAfter: %s\n", s.SplitAfter("123++432++", "++"))

   trimFunction := func(c rune) bool {
      return !unicode.IsLetter(c)
   }
   f("TrimFunc: %s\n", s.TrimFunc("123 abc ABC \t .", trimF
}
```

最后一部分代码提供的方法非常好理解，而且功能强大：

`strings.Replace()` 函数需要4个参数，第一个是你要处理的字符串，第二个是你准备替换的字符，第三个是你要用该字符去替换，第四个参数是你要替换的数量，如果是 `-1` 就意味着你会替换所有要替换的字符。

`strings.TrimFunc()` 函数可按照自定义函数获取你感兴趣的内容；

`strings.SplitAfter()` 函数基于指定分隔符将字符串分割。

执行 `useStrings.go` 的输出如下：

> $ go run useStrings.go

```
To Upper: HELLO THERE!
To Lower: hello there
THis WiLL Be A Title!
EqualFold: true
EqualFold: false
Prefix: true
Prefix: false
Suffix: true
Suffix: false
Index: 2
Index: -1
Count: 2
Count: 0
Repeat: abababababab
TrimSpace: This is a line.
TrimLeft: This is a      line.
TrimRight:      This is a        line.
Compare: 1
Compare: 0
Compare: -1
Fields: [This is a string!]
Fields: [Thisis a string!]
[a b c d   e f g]
_a_b_c_d_ _e_f_g_
_a_b_c_d efg
_a_bcd efg
Join: Line 1+++Line 2+++Line 3
SplitAfter: [123++ 432++ ]
TrimFunc: abc ABC
```

官方 `strings` 包中的方法远比 `useStrings.go` 中所展示的多，如果你正在写的代码是关于文本处理的，那么你应该阅读] `strings` 的官方文档以了解更多。

switch语句

在本节介绍 `switch` 语句主要是因为 `switch` 可以用于正则表达式！首先看一段 `switch` 代码块：

```
switch asString {
    case "1":
        fmt.Println("One")
    case "0":
        fmt.Println("Zero")
    default:
        fmt.Println("Do not care!")
}
```

这段代码能够区分不同的 `asString` 值所对应的不同操作。

> *`switch` 代码块设置default子句是非常棒的实践。由于 `switch` 的 `case` 语句是依赖顺序的，所以 `default` 子句总是在最后声明。*

`switch` 的使用还可以更加灵活：

```
switch {
case number < 0:
 fmt.Println("Less than zero")
case number > 0:
 fmt.Println("Bigger than zero")
default:
 fmt.Println("zero")
}
```

上面的代码块能够在某个数字正数、负数、以及0的情况下执行不同的任务。如你所见， `switch` 的分支语句可以是条件语句，那么其分支语句同样也可以是正则表达式！

关于 `switch` 的用法将在 `switch.go` 中分5部分展示。

第一部分：

```
package main

import (
    "fmt"
    "os"
    "regexp"
    "strconv"
)

func main() {
    arguments := os.Args
    if len(arguments) <2 {
        fmt.Println("Usage: switch number")
        os.Exit(1)
    }
```

`regex` 包用于生成正则表达。

第二部分代码：

```
number, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("The value is not an integer",number)
    }else {
        switch {
        case number<0:
            fmt.Println("Less than zero")
        case number >0:
            fmt.Println("Bigger than zero")
        default:
            fmt.Println("Zero")
        }
    }
```

第三部分代码：

```
asString := arguments[1]
    switch  asString{
    case "5":
        fmt.Println("Five")
    case "0":
        fmt.Println("Zero")
    default:
        fmt.Println("Do not care")
    }
```

这部分代码说明case 子句可以包含硬编码的变量，这种情况通常是 switch`后跟有一个变量。

第四部分代码：

```
var negative = regexp.MustCompile(`-`)
    var floatingPoint = regexp.MustCompile(`\d?\.\d`)
    var mail = regexp.MustCompile(`^[^@]+@[^@.]+\.[^@.]+`)
    switch {
    case negative.MatchString(asString):
        fmt.Println("Negative number")
    case floatingPoint.MatchString(asString):
        fmt.Println("Floating Point")
    case mail.MatchString(asString):
        fmt.Println("It is an email")
        fallthrough
    default:
        fmt.Println("Something else")
    }
```

这部分代码十分有趣。首先，我们定义了3个正则表达 `negative` , `floatingPoint` ,以及 `mail` 。然后使用 `regexp.MatchString()` 在 `switch` 中匹配不同的情况。

最后， `fallthrough` 关键字告诉Go执行接下来的分支，即 `default` 分支。这意味着无论 `mail.MatchString(asString)` 是否成功匹配， `default` 子句都会执行。

最后一部分：

```
var aType error = nil
    switch aType.(type) {
    case nil:
        fmt.Println("It is a nil interface")
    default:
        fmt.Println("It it not a nil interface")

    }

}
```

这段代码说明 `switch` 能够区分不同类型，你将在第7章中了解到接口的知识。执行 `switch.go` 将会产生如下输出：

```
$ go run switch.go
Usage: switch number
exit status 1
hanshanjiedeMacBook-Pro:chapter4 hanshanjie$ go run switch.
The value is not an integer 0
Do not care
It is an email
Something else
It is a nil interface
hanshanjiedeMacBook-Pro:chapter4 hanshanjie$ go run switch.
Bigger than zero
Five
Something else
It is a nil interface
hanshanjiedeMacBook-Pro:chapter4 hanshanjie$ go run switch.
Zero
Zero
Something else
It is a nil interface
hanshanjiedeMacBook-Pro:chapter4 hanshanjie$ go run switch.
The value is not an integer 0
Do not care
Floating Point
It is a nil interface
hanshanjiedeMacBook-Pro:chapter4 hanshanjie$ go run switch.
The value is not an integer 0
Do not care
Negative number
It is a nil interface
```

计算Pi的精确值

在本小节中你将学习到如何使用Go标准库 `math/big` 以及其提供的特殊类型，并计算高精度的**Pi**值。

> *本节的代码是我所见过的最丑陋的Go代码，甚至用Java写看起来都会好一些。*

`calculatePi.go` 使用Bellard规则计算Pi值，代码将分4部分展示。

第一部分：

```
package main

import (
   "fmt"
   "math"
   "math/big"
   "os"
   "strconv"
)

var precision uint = 0
```

`precision` 变量代表你想得到的Pi值精度，其声明为全局变量保证在整个程序中都可以被访问到。

第二部分：

```
func Pi(accuracy uint) *big.Float {
   k := 0
   pi := new(big.Float).SetPrec(precision).SetFloat64(0)
   k1k2k3 := new(big.Float).SetPrec(precision).SetFloat64(0
   k4k5k6 := new(big.Float).SetPrec(precision).SetFloat64(0
   temp := new(big.Float).SetPrec(precision).SetFloat64(0)
   minusOne := new(big.Float).SetPrec(precision).SetFloat64
   total := new(big.Float).SetPrec(precision).SetFloat64(0)

   two2Six := math.Pow(2, 6)
   two2SixBig := new(big.Float).SetPrec(precision).SetFloat
```

`new(big.Float)` 创建一个 `big.Float` 类型的变量，并调用 `SetPrec()` 函数将精度设置为参数 `precision` 。

第三部分是贝拉算法计算精确Pi值的函数 `Pi()` 实现（关于贝拉算法可查看文末简介）：

```
for {
      if k > int(accuracy) {
         break
      }
      t1 := float64(float64(1) / float64(10*k+9))
      k1 := new(big.Float).SetPrec(precision).SetFloat64(t1
      t2 := float64(float64(64) / float64(10*k+3))
      k2 := new(big.Float).SetPrec(precision).SetFloat64(t2
      t3 := float64(float64(32) / float64(4*k+1))
      k3 := new(big.Float).SetPrec(precision).SetFloat64(t3
      k1k2k3.Sub(k1, k2)
      k1k2k3.Sub(k1k2k3, k3)

      t4 := float64(float64(4) / float64(10*k+5))
      k4 := new(big.Float).SetPrec(precision).SetFloat64(t4
      t5 := float64(float64(4) / float64(10*k+7))
      k5 := new(big.Float).SetPrec(precision).SetFloat64(t5
      t6 := float64(float64(1) / float64(4*k+3))
      k6 := new(big.Float).SetPrec(precision).SetFloat64(t6
      k4k5k6.Add(k4, k5)
      k4k5k6.Add(k4k5k6, k6)
      k4k5k6 = k4k5k6.Mul(k4k5k6, minusOne)
      temp.Add(k1k2k3, k4k5k6)

      k7temp := new(big.Int).Exp(big.NewInt(-1), big.NewInt
      k8temp := new(big.Int).Exp(big.NewInt(1024), big.NewI

      k7 := new(big.Float).SetPrec(precision).SetFloat64(0)
      k7.SetInt(k7temp)
      k8 := new(big.Float).SetPrec(precision).SetFloat64(0)
      k8.SetInt(k8temp)

      t9 := float64(256) / float64(10*k+1)
      k9 := new(big.Float).SetPrec(precision).SetFloat64(t9
      k9.Add(k9, temp)
      total.Mul(k9, k7)
      total.Quo(total, k8)
      pi.Add(pi, total)

      k = k + 1
   }
   pi.Quo(pi, two2SixBig)
   return pi
}
```

这部分代码是贝拉算法的Go实现，你必须借助于 `math/big` 包中特定的函数进行计算，因为这些函数能够实现你想要达到的数字精度，可以说如果不使用 `big.Float` 、 `big.Int` 变量以及 `math/big` 中的函数，高精度的Pi值根本无法计算。

最后一部分代码：

```
func main() {
   arguments := os.Args
   if len(arguments) == 1 {
      fmt.Println("Please provide one numeric argument!")
      os.Exit(1)
   }

   temp, _ := strconv.ParseUint(arguments[1], 10, 32)
   precision = uint(temp) * 3

   PI := Pi(precision)
   fmt.Println(PI)
}
```

执行 `calculatePi.go` 得到如下输出：

```
$ go run calculatePi.go

Please provide one numeric argument!

exit status 1

$ go run calculatePi.go 20

3.14159265358979324

$ go run calculatePi.go 200

3.14159265358979325696039936173876240401918315624857324349317928357104645024891346711851178431761535428201792941629280905081393787528343561058631336354860243676804770648983892438192 9
```

本节的代码需要使用很多不同的数据类型，务必保证数据类型的正确使用！

课外阅读

文中提到的贝拉算法简介：

Fabrice Bellard在圆周率算法方面也有着惊人的成就，1997年FabriceBellard提出最快圆周率算法公式。在计算圆周率的过程中，Fabrice Bellard使用改良后的查德诺夫斯基方程算法来进行圆周率的计算，并使用贝利-波温-劳夫算法来验证计算的结果。为了纪念他对圆周率算法所作出的杰出贡献，Fabrice Bellard所使用的改良型算法被命名为Fabrice Bellard算法，这种算法是目前所有圆周率算法中最快的一种，这个计算N位PI的公式比传统的BBQ算法要快47%。

2009年的最后一天，Fabr ice Bellard宣布另一重大突破：他用桌面电脑打破了由超级计算机保持的圆周率运算记录。这是一个壮举， 他将PI计算到了小数点后2.7万亿位！更令人惊讶的是， 他使用的不过是价格不到2000欧元的个人PC，仅用了116天，就计算出了PI的小数点后第2.7万亿位，超过了排名世界第47位的T2K Open超级计算机于2009年8月17日创造的世界纪录。新纪录比原纪录多出1200亿位，然而，他使用的这台桌面电脑的配置仅为：2.93GHz Core i7 CPU，6GB内存，7.5TB硬盘! 不过这次为了加快计算完成的速度保住排名第一的位置，Fabrice Bel lard使用了9台联网的电脑来对数据进行验证， 若使用一台电脑来验证计算结果的话， 则需要额外增加13天的计算时间。 Fabrice Bellard在圆周率方面的辉煌成就， 使他创造多次圆周率单一位计算的世界纪录（计算10的整次幂位）， 也曾因此而登上《科学美国人》法文版。

实现简单的K-V存储

本节你将学习使用Go实现**K-V存储**的简单实现，其背后的思想是易于理解的：尽可能快速地发出请求并给出响应，并将其转化成对应的数据结构。

本节代码将实现K-V存储的四个基本功能：

1. 添加新元素
2. 基于key删除已有的元素
3. 给定key查找对应value
4. 修改key对应的value

我们将这四个功能命名为 `ADD` ，`DELETE` ，`LOOKUP` ，`CHANGE` ，完成这四个基本功能，你将会对K-V存储的实现有一个全面的了解。另外，当你输入 `STOP` 时整个程序就会停止，输入 `PRINT` 命令就会打印出当前K-V存储的内容。

本节的 `keyValue.go` 将分为5个代码段解释。

第一部分：

```
package main

import (
   "bufio"
   "fmt"
   "os"
   "strings"
)

type myElement struct {
   Name string
   SurName string
   Id string
}

var DATA = make(map[string]myElement)
```

我们使用原生的Go map来实现K-V存储，因为内置的数据结构往往执行效率更高。 `map` 变量被声明为全局变量，其k为 `string` 类型，v为 `myElement` 类型， `myElement` 是自定义的结构体。

第二部分代码：

```
func ADD(k string,n myElement) bool {
   if k == "" {
      return false
   }
   if LOOKUP(k) == nil {
      DATA[k] = n
      return true
   }
   return false
}

func DELETE(k string) bool {
   if LOOKUP(k) != nil {
      delete(DATA, k)
      return true
   }
   return false
}
```

这部分代码实现了命令行 `ADD` 和 `DELETE` ，用户在执行 `ADD` 命令时，如果没有携带足够的参数，我们要保证该操作不会失败，意味着 `myElement` 中对应的字段为空字符串。然而如果你要添加的key已经存在了，就会报错（K-V存储不允许重复key出现）而不是修改对应的值。

第三部分代码：

```
func LOOKUP(k string) *myElement  {
   _, ok := DATA[k]
   if ok {
      n := DATA[k]
      return &n
   } else {
      return nil
   }
}

func CHANGE(k string, n myElement) bool {
   DATA[k] = n
   return true
}

func PRINT()  {
   for k, v := range DATA {
      fmt.Printf("key: %s value: %v",k,v)
   }
}
```

该代码段实现了 `LOOKUP` 与 `CHANGE` 功能，如果你要修改的key不存储，程序会自动将其存储。

`PRINT` 命令能够打印出目前所有K-V存储的内容。

第四部分代码：

```
func main() {
   scanner := bufio.NewScanner(os.Stdin)
   for scanner.Scan() {
      text := scanner.Text()
      text = strings.TrimSpace(text)
      tokens := strings.Fields(text)

      switch len(tokens) {
      case 0:
         continue
      case 1:
         tokens = append(tokens,"")
         tokens = append(tokens,"")
         tokens = append(tokens,"")
         tokens = append(tokens,"")
      case 2:
         tokens = append(tokens,"")
         tokens = append(tokens,"")
         tokens = append(tokens,"")
      case 3:
         tokens = append(tokens,"")
         tokens = append(tokens,"")
      case 4:
         tokens = append(tokens,"")

      }
```

该部分代码读取用户的输入。首先， `for` 循环将保证程序一直等待用户的输入，接下来使用 `tokens` 切片保证至少有5个元素的输入，即使只有 `ADD` 命令需要5个参数。如果用户不想在使用 `ADD` 命令时出现空字符串值，就需要这样输入： `ADD aKey Field1 Field2 Field3` 。

最后一部分代码：

```
switch tokens[0] {
      case "PRINT":
         PRINT()
      case "STOP":
         return
      case "DELETE":
         if !DELETE(tokens[1]) {
            fmt.Println("Delete operations failed")
         }
      case "ADD":
         n := myElement{tokens[2],tokens[3],tokens[4]}
         if !ADD(tokens[1],n) {
            fmt.Println("Add operation failed")
         }
      case "LOOKUP":
         n := LOOKUP(tokens[1])
         if n != nil {
            fmt.Printf("%v\n",n)
         }

      case "CHANGE":
         n := myElement{tokens[2],tokens[3],tokens[4]}
         if !CHANGE(tokens[1],n) {
            fmt.Println("Update operation failed")
         }

      default:
         fmt.Println("Unknown command - please try again!")

      }
   }
}
```

这部分代码处理用户的输入。`switch` 的使用使得程序的逻辑设计看上去十分清晰，能够将程序员从冗余的 `if...else` 中拯救出来。

执行 `keyValue.go` 得到如下输出：

```
$ go run keyValue.go
UNKNOWN
Unknown command - please try again!

ADD 123 1 2 3
ADD 234 2 3 4
ADD 234
Add operation failed
ADD 345
PRINT
key: 123 value: {1 2 3}key: 234 value: {2 3 4}key: 345 valu
CHANGE 345 3 4 5
PRINT
key: 345 value: {3 4 5}key: 123 value: {1 2 3}key: 234 valu
DELETE 345
PRINT
key: 123 value: {1 2 3}key: 234 value: {2 3 4}
ADD 567 -5 -6 -7
PRINT
key: 123 value: {1 2 3}key: 234 value: {2 3 4}key: 567 valu
CHANGE 567
PRINT
key: 567 value: {  }key: 123 value: {1 2 3}key: 234 value:
STOP
```

学习 第8章 后你将学会如何支持K-V存储的数据持久化功能。然而，在单用户应用使用goroutines和channel是没有任何实际意义的，但是如果你是通过**TCP/IP**实现K-V存储，那么使用goroutines和channel会帮助你实现多连接、服务多用户功能。你将在 第9、10章 学习goroutines和channel的知识，并且在 12、13章 学习网络编程的相关知识，尽情期待吧！

延展阅读

- `regexp` 标准库的详细讲解-https://golang.org/pkg/regexp
- 阅读 `grep` 工具的使用说明
- 使用Go开发一个程序，能够找出log文件中所有的IPv4地址
- 关于 `math/big` 包的详细讲解-https://golang.org/pkg/math/big
- `unicode` 标准库的官方文档-https://golang.org/pkg/unicode
- 开始阅读官方的Go语言规范吧-https://golang.org/ref/spec

练习

- 写一个Go程序，识别无效的IPv4地址
- 说说 `make` 和 `new` 的区别
- 说说字符、字节和 `rune` 之间的区别
- 在不使用UNIX工具的前提下，修改 `findIPv4.go` 使之能够打印出出现次数最多的IPv4地址
- 写一个Go程序，能够识别出log文件中产生404错误的IPv4地址
- 使用 `math/big` 包计算高精度平方根，算法自选
- 写一个Go程序，从给定时间与日期中寻找特定格式的时间。
- 写一个正则表达式匹配200-400之间的整数
- 为 `keyValue.go` 添加日志打印
- 修改 `findIPv4.go` 中的 `findIP()` 函数，保证 `findIp()` 多次调用时正则表达式只编译一次

本章小结

在本章中，你学到了Go的很多实用特性，包括创建与使用结构体，元组，字符串，runes以及Unicode标准库。另外，你也掌握了模式匹配与正则表达式，`switch`语句，`strings`标准库；并且，你实现了Go版本的K-V存储，使用`math/big`包计算出高精度的Pi值。

在下一章中，你将学习使用更加高级的数据结构来操作数据，比如二叉树，链表，双向链表，队列，栈，哈希表以及`container`标准库中的数据结构。下一章最后的主题将是**随机数**，我们将生成难以破解的密码字符串。

数据结构

在上一章中，我们讨论了可以通过 `struct` 关键字定义的组合类型，正则表达式，模式匹配，元组， `runes` ，字符串， `unicode` 和 `strings` 包，之后我们开发了一个简单的 `key-value` 存储。但是有时候编程语言提供的这些结构不适合一些特定的问题。有些时候我们需要自己创建的数据结构，以便以准确，专业的方式去存储，搜索，接收数据。因此，本章将介绍在Go中开发和使用众所周知的数据结构，包括二叉树，链表，散列表，堆栈和队列以及了解它们的优点。由于没有比图像更好地描述数据结构，因此本章将会看到许多图解！本章最后一部分将讨论随机数的生成，它将会帮你生成难以猜测的密码！下面是本章的内容概览：

- 图和节点
- 分析算法复杂度
- Go的二叉树
- Go的哈希表
- Go的链表
- Go的双端链表
- Go的队列
- Go的栈
- Go标准库 `container` 包提供的数据结构
- Go生成随机数
- 构建随机字符串用作难以破解的密码

图和节点

图G(V,E)是由一组有限的，非空的顶点集V（或者节点）以及一组边E组成。有两种主要的类型：有环图和无环图。有环图是指其全部或者部分顶点集存在闭环。在无环中，没有闭环。有向图是其边有方向性，有向无环图是没有闭环的有向图。

通常一个节点（node)包含很多信息，所以node一般使用Go的数据结构来实现

算法复杂度

算法的效率是由计算的复杂度来判定的，这主要是和算法完成工作需要访问其输入数据的次数有关。

在计算机科学中通常用大 `O` 表示法来描述算法的复杂度。因此，只需要访问线性次输入数据的O(n)算法要更优于$O(n^2)$的算法以及$O(n^3)$的算法等等。然而最坏的算法就是O(n!)，想输入超过300个元素是不可能的。

最后，Go大多数内置数据类型的查找操作，例如map中通过key查找value或者访问数组元素都是常量时间内完成即O(1)。这就意味着内置数据类型通常比自定义类型更快，所以你应该优先使用它们，除非你想完全控制幕后发生的事情。此外并非所有的数据结构都相同，通常来讲数组操作数据比map要快，这也是你必须为map的多功能性付出的代价。

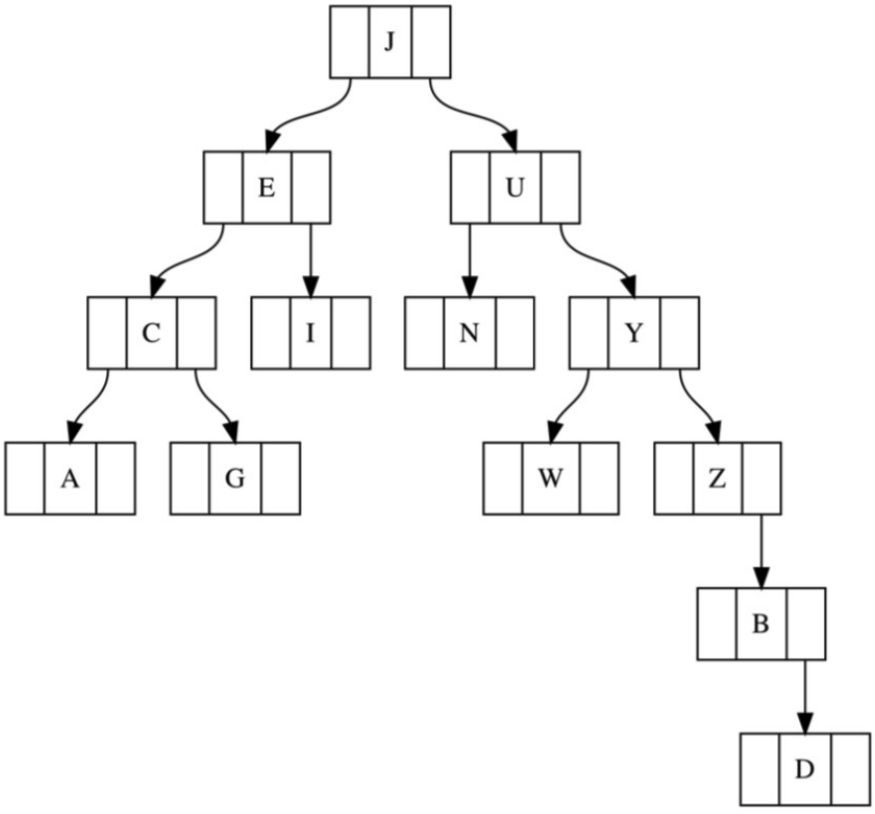
尽管每种算法都有他的缺点，但是如果你的数据量不是很大的时候，只要算法能够准确的执行所需的工作，那么这些就不是很重要了。

Go 语言中的二叉树

二叉树是每个节点最多有两个两个分支的数据结构。“最多”表示一个节点可以连接一至两个子节点，或者不连接其他子节点。**树的根节点**是树结构中的第一个节点。**树的深度**，又称为树的高度，是从树的根节点到所有节点的路径中最长的一个。而节点的深度是该节点到树的根所经过的路径中边的数量。**叶节点**是没有子节点的节点。

若一个树的根节点到任意两个叶节点的距离之差不大于 1，则认为这个树是**平衡的**。顾名思义，非平衡树不是平衡的。树的平衡操作困难又缓慢，所以最好从一开始就让你的树保持平衡，而不是在建好树之后再去调整，尤其是这个树有很多节点的时候。

下图是一个非平衡二叉树。它的节根点是 J，叶节点包括 A、G、W 和 D。



An unbalanced binary tree

Go 语言实现二叉树

本节介绍了如何使用 Go 语言实现一个二叉树，示例代码在 `binTree.go` 中。 下面将 `binTree.go` 的内容分成五个部分来介绍。第一部分如下：

```
package main

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

type Tree struct {
    Left  *Tree
    Value int
    Right *Tree
}
```

这里是使用 Go 语言中的结构体定义的树的节点。由于我们没有真实的数据，所以将使用 `math/rand` 包向树中填充随机的数值。

`binTree.go` 中代码的第二部分如下：

```
func traverse(t *Tree) {
    if t == nil {
        return
    }
    traverse(t.Left)
    fmt.Println(t.Value, " ")
    traverse(t.Right)
}
```

`traverse()` 函数展示了如何使用递归访问二叉树上的所有节点。

`binTree.go` 中的第三个代码段如下：

```
func create(n int) *Tree {
    var t *Tree
    rand.Seed(time.Now().Unix())
    for i := 0; i < 2*n; i++ {
        temp := rand.Intn(n * 2)
        t = insert(t, temp)
    }
    return t
}
```

`create()` 函数仅用于向树中填充随机的数值。

该程序的第四部分如下：

```
func insert(t *Tree, v int) *Tree {
    if t == nil {
        return &Tree{nil, v, nil}
    }
    if v == t.Value {
        return t
    }
    if v < t.Value {
        t.Left = insert(t.Left, v)
        return t
    }
    t.Right = insert(t.Right, v)
    return t
}
```

`insert()` 函数使用 `if` 语句做了很多重要的事。第一个 `if` 语句检查要操作的树是否为空。如果是空树，那么通过 `&Tree{nil, v, nil}` 创建的新节点将成为该树的根节点。第二个 `if` 语句判断二叉树上是否已经存在将要插入的值。如果值已经存在，那么函数将什么也不做然后返回。第三个 `if` 语句判断对于当前节点，被插入的值是在节点的左侧还是右侧，然后执行相应的操作。

注意，这里展示的实现创建的是非平衡二叉树。

`binTree.go` 的最后一部分包含如下的 Go 代码：

```
func main() {
    tree := create(10)
    fmt.Println("The value of the root of the tree is", tree.Val
    traverse(tree)
    fmt.Println()
    tree = insert(tree, -10)
    tree = insert(tree, -2)
    traverse(tree)
    fmt.Println()
    fmt.Println("The value of the root of the tree is", tree.Val
}
```

执行 `binTree.go` 将生成类似如下的输出：

```
$ go run binTree.go
The value of the root of the tree is 18
0 3 4 5 7 8 9 10 11 14 16 17 18 19
-10 -2 0 3 4 5 7 8 9 10 11 14 16 17 18 19
The value of the root of the tree is 18
```

二叉树的优点

当你要描述多层次数据的时候，树是最好的选择。因此，树被广泛应用于编程语言的编译器解析计算机程序的过程。

此外，树本身就具有**有序性**，所以你不必另外对其进行排序。只要插入到了正确的位置，那么树就能保持有序。然而，由于树的构造方式，删除元素的操作有时至关重要。

如果一个二叉树是平衡的，它的查找、插入、删除操作需要 *log(n)* 步，这里的 n 是树上元素的总数量。此外，平衡二叉树的高度约为 $log_2(n)$，这意味着一个拥有 10,000 个元素的平衡树的高度大约是 14。类似的，拥有 100,000 个元素的平衡树的高度大约是 17，拥有 1,000,000 个元素的平衡树的高度大约是 20。也就是说，向一个平衡二叉树中插入大量的元素后，对树进行操作的速度并不会大幅变化。换个说法，你只需要不到 20 步就能到达一个拥有 1,000,000 个节点的树上的任意一个节点！

二叉树最大的一个缺点是树的形状很依赖元素插入的顺序。如果树上元素的键又长又复杂，那么插入和查找元素的过程需要进行大量的匹配，从而变得很慢。最后，如果一个树不是平衡的，那么就很难对其性能进行预测。

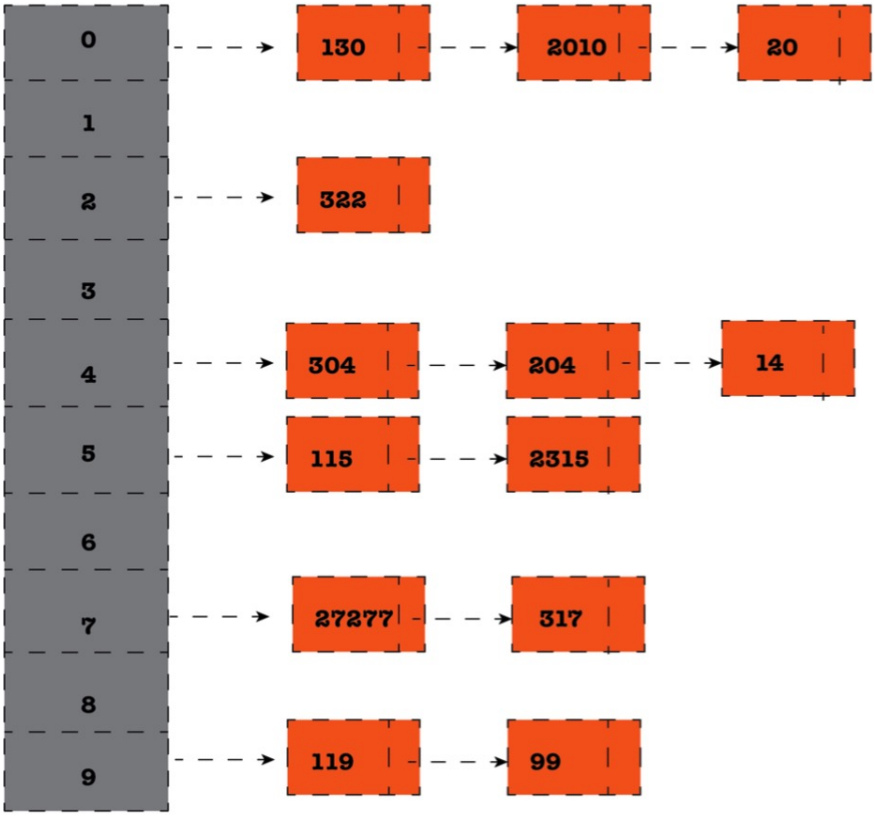
尽管你能更快地创建链表或数组，但是二叉树在查找操作中的灵活性值得付出额外的开销并进行维护。在二叉树上查找一个元素时，你会比较要搜索的元素则值与当前节点值的大小，然后决定在哪个子树上继续搜索，这样做可以节约大量的时间。

Go 语言中的哈希表

严格来讲，**哈希表**这种数据结构会存储一至多个键值对，并使用一个哈希函数计算出存放正确值的桶或槽的索引。理想情况下，**哈希函数**应该将每个键分配到唯一的桶中，但通常你事先得有足够数量的桶。

一个好的哈希函数一定要能够输出均匀分布的哈希值，如果有多余的桶或者桶之间的基数差值较大，操作的效率就会变低。此外，这个哈希函数应该具有一致性，对于同一个键始终输出相同的哈希值，否则就不能准确定位需要的数据。

下图展示了一个拥有 10 个桶的哈希表。



A hash table with 10 buckets

Go 语言实现哈希表

下面将 `hashTable.go` 中的 Go 语言代码分成五个部分来讲解哈希表。

下面是 `hashTable.go` 中的第一部分：

```
package main

import (
    "fmt
)

const SIZE = 15

type Node struct {
    Value int
    Next  *Node
}
```

这一部分是哈希表中节点的定义，毫无疑问，我们使用 Go 的结构体进行了实现。`SIZE` 常量表示哈希表中桶的数量。

下面的 Go 代码是 `hashTable.go` 中的第二段：

```
type HashTable struct {
    Table map[int]*Node
    Size  int
}

func hashFunction(i, size int) int {
    return (i % size)
}
```

在这个代码段中，你可以看到在这个哈希表中使用的哈希函数。`hashFunction()` 使用了**模运算符**。这里使用模运算符的主要原因是这个哈希表要处理的是整数值的键。如果你要应对的是字符串或浮点数的键，那么你的哈希函数的逻辑应该与这不同。

真正的哈希表存在一个 `HashTable` 结构体中，它有两个字段。第二个字段表示哈希表的大小，第一个字段则是连接整数和链表（`*Node`）的 map。因此，这个哈希表中链表的数量与桶的数量相等。这也意味着哈希表上每个桶中的节点都会使用链表存储。之后会进一步对链表进行介绍。

`hashTable.go` 的第三部分如下：

```
func insert(hash *HashTable, value int) int {
    index := hashFunction(value, hash.Size)
    element := Node{Value: value, Next: hash.Table[index]}
    hash.Table[index] = &element
    return index
}
```

`insert()` 函数用于向哈希表中插入元素。注意，当前 `insert()` 的实现不会检查值是否重复。

下面是 `hashTable.go` 的第四部分：

```
func traverse(hash *HashTable) {
    for k := range hash.Table {
        if hash.Table[k] != nil {
            t := hash.Table[k]
            for t != nil {
                fmt.Printf("%d -> ", t.Value)
                t = t.Next
            }
        }
        fmt.Println()
    }
}
```

`traverse()` 函数用于输出哈希表所有的值。这个函数访问哈希表中所有链表，然后依次输出每个链表上存储的值。

`hashTable.go` 的最后一部分代码如下：

```
func main() {
    table := make(map[int]*Node, SIZE)
    hash := &HashTable{Table: table, Size: SIZE}
    fmt.Println("Numbder of spaces:", hash.Size)
    for i := 0; i < 120; i++ {
        insert(hash, i)
    }
    traverse(hash)
}
```

在这部分代码中，你将创建一个新的、命名为 `hash` 的哈希表，它接收一个 `table` 变量，这个变量是一个保存了哈希表中所有桶的 map。正如你所知道的，这个哈希表的槽是使用链表实现的。我们使用 map 保存哈希表中的链表，而没用切片或数组，这主要是因为切片或数组的键只能是正整数，但 map 的键则几乎可以满足你的所有需求。

执行 `hashTable.go` 将生成如下输出：

```
$ go run hashTable.go
108 -> 93 -> 78 -> 63 -> 48 -> 33 -> 18 -> 3 ->
109 -> 94 -> 79 -> 64 -> 49 -> 34 -> 19 -> 4 ->
117 -> 102 -> 87 -> 72 -> 57 -> 42 -> 27 -> 12 ->
119 -> 104 -> 89 -> 74 -> 59 -> 44 -> 29 -> 14 ->
111 -> 96 -> 81 -> 66 -> 51 -> 36 -> 21 -> 6 ->
112 -> 97 -> 82 -> 67 -> 52 -> 37 -> 22 -> 7 ->
116 -> 101 -> 86 -> 71 -> 56 -> 41 -> 26 -> 11 ->
114 -> 99 -> 84 -> 69 -> 54 -> 39 -> 24 -> 9 ->
118 -> 103 -> 88 -> 73 -> 58 -> 43 -> 28 -> 13 ->
105 -> 90 -> 75 -> 60 -> 45 -> 30 -> 15 -> 0 ->
106 -> 91 -> 76 -> 61 -> 46 -> 31 -> 16 -> 1 ->
107 -> 92 -> 77 -> 62 -> 47 -> 32 -> 17 -> 2 ->
110 -> 95 -> 80 -> 65 -> 50 -> 35 -> 20 -> 5 ->
113 -> 98 -> 83 -> 68 -> 53 -> 38 -> 23 -> 8 ->
115 -> 100 -> 85 -> 70 -> 55 -> 40 -> 25 -> 10 ->
```

这个哈希表已经完美平衡，因为它所需要做的是将连续数放入**模运算**所计算出的槽中。但是现实世界的问题中可能不会出现这么方便的结果！

> 在欧几里得除法中，两个自然数 a 和 b 之间的余数可以通过公式 a = bq + r 进行计算，这里的 q 是商，r 是余数。余数的值的范围是 0 到 b-1，这个范围中的值都是模运算可能的结果。

注意，如果多次运行 `hashTable.go`，你所得到的那些输出的行的顺序很可能不同，因为 Go 的 map 输出键值对的顺序是完全随机的！

实现查找功能

在这一节中，你将会实现一个 `lookup()` 函数，使用这个函数可以检查哈希表中是否已经存在某个给定的元素。 `lookup()` 函数基于 `traverse()` 函数实现，如下所示：

```
func lookup(hash *HashTable, value int) bool {
    index := hashFunction(value, hash.Size)
    if hash.Table[index] != nil {
        t := hash.Table[index]
        for t != nil {
            if t.Value == value {
                return true
            }
            t = t.Next
        }
    }
    return false
}
```

你可以在 `hashTableLookup.go` 源文件中找到上面的代码。执行 `hashTableLookup.go` 将得到下面的输出：

```
$ go run hashTableLookup.go
120 is not in the hash table!
121 is not in the hash table!
122 is not in the hash table!
123 is not in the hash table!
124 is not in the hash table!
```

以上的输出表示 `lookup()` 函数效果很棒！

哈希表的优点

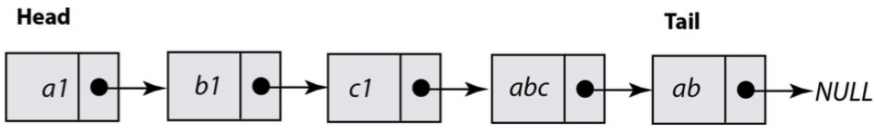
如果你认为哈希表不实用、不方便或者不巧妙，请考虑如下情形：当一个哈希表有 n 个键和 k 个桶时，查找 n 个键的时间复杂度将从使用线性查找的 *O(n)* 下降至 *O(n/k)*！尽管这点改进看起来不起眼，你也必须意识到对于一个有 20 个槽的哈希数组，查找的次数将会下降为原来的 1/20！这让哈希表非常适合用于搭建词典或者其他这类需要大量数据查找的应用。

Go 语言中的链表

链表是一种包含有限个元素的数据结构，其中每个元素至少占用两个存储单元，一个用于存储真正的数据，另一个用于存储链接当前元素和下一个元素的指针，从而建立了一个元素序列构成的链表。

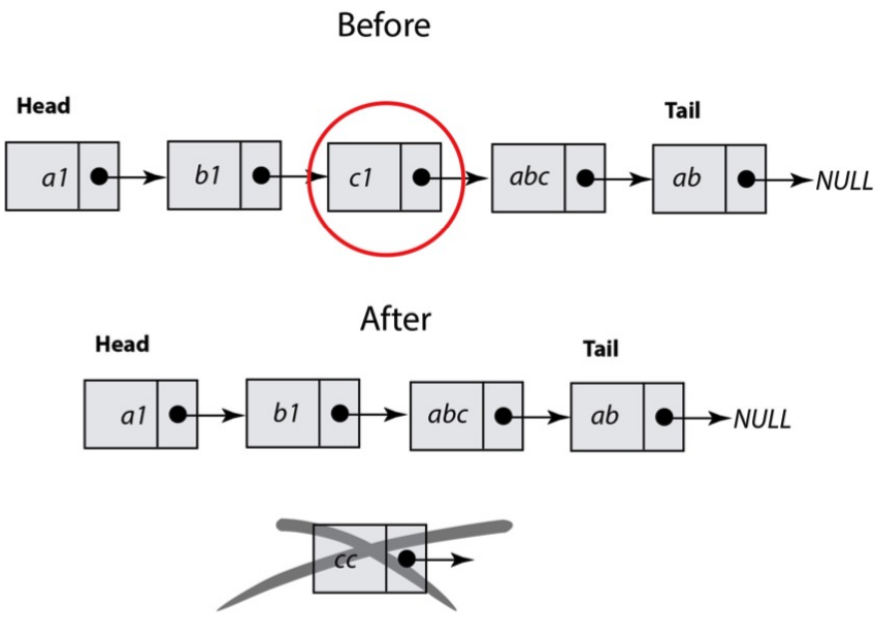
链表中的第一个元素称为头，最后一个元素称为尾。在定义链表的过程中，你需要做的第一件事就是将链表头用单独的变量存储，因为链表头是你访问整个链表的唯一媒介。注意，如果弄丢了指向单向链表第一个节点的指针，你就没法再找到它了。

下图展示了一个拥有五个节点的链表：



A linked list with five nodes

下图描述了如何从链表中移除一个节点，它将帮助你更好地理解这个过程所涉及的步骤。你要做的主要是修改被删除节点左侧的那个节点，将它指向下一个元素的指针指向被删除节点右侧的那个节点。



Removing a node from a linked list

下面的链表实现已经相对简化了，并且不含删除节点的功能，这个功能的实现将留给读者作为练习题。

Go 语言实现链表

链表相关实现的 Go 源文件是 `linkedList.go`，我们将分为五个部分来介绍。

`linkedList.go` 中的第一个代码段如下：

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var root = new(Node)
```

在这部分代码中，我们定义了用作链表节点的 `Node` 结构类型和保存链表第一个元素的全局变量 `root`，在代码的其他任何地方我们都能访问这个变量。

`linkedList.go` 的第二部分是如下的 Go 代码：

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        t.Next = &Node{v, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

依据工作原理，链表通常不含重复的项。此外，如果不是有序链表，新的节点通常会被添加在链表的尾部。

因此，`addNode()` 函数的功能就是向链表中添加新的节点。这个实现中使用 `if` 语句检查了三种不同的情况。第一种情况，检查要处理的链表是否为空。第二种情况，检查将要插入的值是否已经存在。第三种情况，检查是否已经到达了链表的末端，这时，使用 `t.Next = &Node{v, nil}` 将含有给定值的新节点加入链表的末端。如果所有的情况都不满足，则使用 `return addNode(t.Next, v)` 对链表中下一个节点调用 `addNode()`，重复上面的过程。

`linkedList.go` 程序的第三个代码片段包含了 `traverse()` 函数的实现：

```
func traverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

`linkedList.go` 的第四部分如下：

```
func lookupNode(t *Node, v int) bool {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}

func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    i := 0
    for t != nil {
        i++
        t = t.Next
    }
    return i
}
```

这一部分包含了两个非常方便的函数——`lookupNode()` 和 `size()` 的实现。前一个函数检查链表中是否存在给定的元素，后一个函数返回链表的大小，也就是链表中节点的数量。

`lookupNode()` 函数实现背后的逻辑很容易理解：依次访问单向链表中所有的元素来查找你想要的值。如果你从头到尾都没找到想要的值，那就说明链表中没有这个值。

`linkedList.go` 的最后一部分包含 `main()` 函数的实现：

```
func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, -1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 45)
    addNode(root, 5)
    addNode(root, 5)
    traverse(root)
    addNode(root, 100)
    traverse(root)

    if lookupNode(root, 100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }

    if lookupNode(root, -100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }
}
```

执行 `linkedList.go` 将生成如下的输出：

```
$ go run linkedList.go
&{0 <nil>}
-> Empty list!
1 -> -1 ->
Node already exists: 5
Node already exists: 5
1 -> -1 -> 10 -> 5 -> 45 ->
1 -> -1 -> 10 -> 5 -> 45 -> 100 ->
Node exists!
Node does not exist!
```

链表的优点

链表最大的优点就是容易理解和实现。链表的通用性使得它们可以用于很多不同的情形。这意味着可以用链表对各种不同类型的数据建模。此外，在链表中使用指针进行顺序检索的速度非常快。

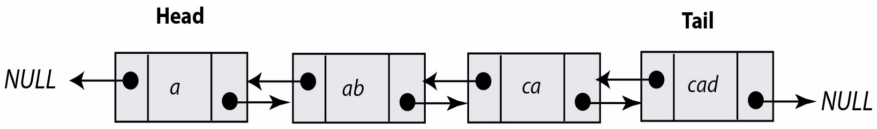
链表不仅可以对数据排序，还可以在插入和删除元素之后保持数据的有序性。从有序链表中删除一个节点的操作与无序链表中相同，而向有序链表中插入新的节点则与无序链表中不同，为了保持链表的有序性，必须将新节点放到正确的位置。实际上，这意味着如果你有大量数据并且会频繁删除数据，那么相对于哈希表和二叉树，使用链表是一个更好的选择。

最后，有很多技术可以优化**有序链表**中搜索和查找节点的过程。其中最常用的一种就是保存一个指向有序链表中间节点的指针，之后每次都从这个节点开始查找。这个简单的优化方法可以将查找操作的时间减少一半！

Go 语言中的双向链表

双向链表中的每个节点都既有指向前一个元素的指针，又有指向下一个元素的指针。

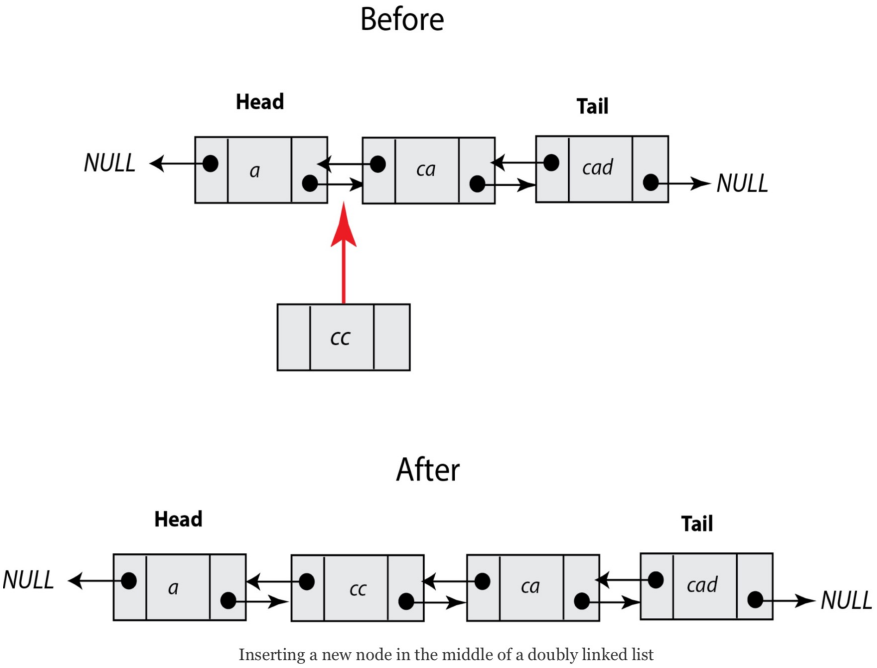
双向链表形如下图：



A doubly linked list

因此，在一个双向链表中，第一个节点的后链接指向第二个节点，而它的前链接指向 `nil`（也称为 **NULL**）。类似的，最后一个节点的后链接指向 `nil`，而它的前链接指向双向链表中的倒数第二个节点。

本章的最后一个插图阐明了双向链表中增加节点的操作。可想而知，这个过程中的主要任务是处理新节点、新节点左侧节点、新节点右侧节点这三个节点的指针。



Inserting a new node in the middle of a doubly linked list

所以，单向链表和双向链表的主要区别实际上只是双向链表的操作更冗杂。这是你为了能够从两个方向都能访问双向链表所必须付出的代价。

Go 语言实现双向链表

实现了双向链表的 Go 程序是 `doublyLList.go`，我们将分为五个部分来介绍。双向链表背后的基本思想和单向链表相同。不过由于双向链表中每个节点都有两个指针，所以操作更冗杂。

`doublyLList.go` 的第一部分如下：

```
package main

import (
    "fmt"
)

type Node struct {
    Value    int
    Previous *Node
    Next     *Node
}
```

这部分中使用 Go 的结构体定义了双向链表的节点。不过这次的 `struct` 中有两个指针字段，原因就不必多说了。

`doublyLList.go` 的第二部分包含如下的 Go 代码：

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        temp := t
        t.Next = &Node{v, temp, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

就像单向链表中一样，新的节点都被添加在当前双向链表的末端。这并不是强制性的，你也可以选择对这个双向链表进行排序。

`doublyLList.go` 的第三部分如下：

```
func traverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

func reverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    temp := t
    for t != nil {
        temp = t
        t = t.Next
    }

    for temp.Previous != nil {
        fmt.Printf("%d -> ", temp.Value)
        temp = temp.Previous
    }
    fmt.Printf("%d -> ", temp.Value)
    fmt.Println()
}
```

这是 `traverse()` 和 `reverse()` 函数的 Go 代码。`traverse()` 的实现和 `linkedList.go` 中的一样。但是 `reverse()` 背后的逻辑很有趣。因为没有指向双向链表尾节点的指针，所以我们必须先从头向后访问，直到找到尾节点，之后才能反向遍历所有的节点。

`doublyLList.go` 的第四部分包含如下的 Go 代码：

```
func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    n := 0
    for t != nil {
        n++
        t = t.Next
    }
    return n
}

func lookupNode(t *Node, v int) bool {
    if root == nil {
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}
```

`doublyLList.go` 的最后一个代码段包含如下的 Go 代码：

```
var root = new(Node)

func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, 1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 0)
    addNode(root, 0)
    traverse(root)
    addNode(root, 100)
    fmt.Println("Size:", size(root))
    traverse(root)
    reverse(root)
}
```

执行 `doublyLList.go`，你将得到如下输出：

```
$ go run doublyLList.go
&{0 <nil> <nil>}
-> Empty list!
Node already exists: 1
1 ->
Node already exists: 0
1 -> 10 -> 5 -> 0 ->
Size: 5
1 -> 10 -> 5 -> 0 -> 100 ->
100 -> 0 -> 5 -> 10 -> 1 ->
```

如你所见，`reverse()` 函数效果很好！

双向链表的优点

双向链表的功能比单向链表更加丰富，你可以从任意方向遍历双向链表，也可更容易地在双向链表中增删元素。此外，即使弄丢了指向双向链表头节点的指针，你也可以重新找回它。然而，这种多功能性的代价就是每个节点需要维护两个指针。开发者需要考虑这种额外的复杂性是否值得。

总之，你的音乐播放器中的歌单用的可能就是双向链表，所以你才能切换到上一首歌和下一首歌！

Go 语言中的队列

队列是一种特殊的链表，它总是在链表头添加新的元素，在链表尾删除元素。我们不必用插图来描述队列。想象一下银行中的情形，你必须等到比你先来的人完成交易之后才能和出纳员交谈。这就是个队列！

队列最大的优点就是简单！你只需要两个函数就能访问一个队列，这意味着你需要担心的事情更少，并且你只用完成这两个函数就能实现一个队列！

Go 语言实现队列

`queue.go` 程序描述了 Go 语言的队列实现，我们将分为五个部分来介绍。注意，这里队列的实现使用了链表。 `Push()` 函数和 `Pop()` 函数分别用于队列中元素的增删。

`queue.go` 中的第一部分代码如下：

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var size = 0
var queue = new(Node)
```

用一个参数（ `size` ）保存队列中节点的数量很实用但不是必须的。不过为了简化操作，这里展示的实现还是提供了这个功能。

`queue.go` 的第二部分包含如下的 Go 代码：

```
func Push(t *Node, v int) bool {
    if queue == nil {
        queue = &Node{v, nil}
        size++
        return true
    }

    t = &Node{v, nil}
    t.Next = queue
    queue = t
    size++

    return true
}
```

这部分展示了 `Push()` 函数的实现，这个实现很好理解。如果队列是空的就会被新的节点取代。如果队列不为空，那么新节点就会被插入到当前队列的前面，然后新节点会变成队列的头节点。

`queue.go` 的第三部分包含如下 Go 代码：

```
func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        queue = nil
        size--
        return t.Value, true
    }

    temp := t
    for (t.Next) != nil {
        temp = t
        t = t.Next
    }

    v := (temp.Next).Value
    temp.Next = nil

    size--
    return v, true
}
```

以上代码展示了 `Pop()` 函数的实现，该函数将最老的元素从队列中移除。如果队列为空（ `size == 0` ）就没有值可以返回。如果队列只有一个节点，那么将返回这个节点的值，队列也会变成空的。其他情况下将取出队列中的最后一个元素，并将移除最后一个节点，然后对节点的指针进行必要的修改，最后返回被删除节点的值。

`queue.go` 的第四个代码段包含如下的 Go 代码：

```
func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Queue!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

严格来说，`traverse()` 函数对于队列的操作并不是必须的，但是它提供了一种实用的方法来查看队列中的所有节点。

`queue.go` 的最后一个代码段如下：

```
func main() {
    queue = nil
    Push(queue, 10)
    fmt.Println("Size:", size)
    traverse(queue)

    v, b := Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    for i := 0; i < 5; i++ {
        Push(queue, i)
    }
    traverse(queue)
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)
    traverse(queue)
}
```

`main()` 中几乎所有的代码都是对队列操作的检查。其中最重要的代码是两个 `if` 语句，它能让你知道 `Pop()` 函数返回了一个确切的值还是因队列为空而没有返回任何值。

执行 `queue.go` 将产生如下输出：

```
Size: 1
10 ->
Pop: 10
Size: 0
4 -> 3 -> 2 -> 1 -> 0 ->
Size: 5
Pop: 0
Size: 4
Pop: 1
Size: 3
4 -> 3 -> 2 ->
```

Go 语言中的栈

栈是一种看起来像一堆盘子的数据结构。你在需要一个新盘子时会先使用最后一个被放在这堆盘子顶端的那个盘子。

栈最大的优点就是简单，因为你只需要实现两个函数就能使用栈，之后你可以向栈中添加新节点或者删除栈中的节点。

Go 语言实现栈

现在是时候看看如何使用 Go 语言实现栈了。相关的细节在 `stack.go` 源文件中。同样，栈的实现也会用到链表。如你所知，你需要两个函数：一个 `Push()` 函数将元素放入栈中，一个 `Pop()` 函数从栈中删除元素。单独使用一个变量保存栈中元素的数量是个很实用的方法，这样即使不访问链表也能判断栈是否为空，不过这不是必须的。

`stack.go` 中的源代码将分四个部分介绍。第一部分如下：

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var size = 0
var stack = new(Node)
```

`stack.go` 的第二部分包含了 `Push()` 函数的实现：

```
func Push(v int) bool {
    if stack == nil {
        stack = &Node{v, nil}
        size = 1
        return true
    }

    temp := &Node{v, nil}
    temp.Next = stack
    stack = temp
    size++
    return true
}
```

如果栈为空就创建一个新节点（`temp`）并将它放在当前栈的最前面，然后新节点会成为栈的头节点。这个版本的 `Push()` 函数永远返回 `true`。对于存储空间有限的栈，你可以修改一下，在栈将要溢出时返回 `false`。

第三部分包含 `Pop()` 函数的实现：

```
func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        size = 0
        stack = nil
        return t.Value, true
    }

    stack = stack.Next
    size--
    return t.Value, true
}
```

`stack.go` 的第四个代码段如下：

```
func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Stack!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

由于这里的栈是使用链表实现的，所以也用链表的方式进行遍历。

`stack.go` 的最后一部分如下：

```
func main() {

    stack = nil
    v, b := Pop(stack)
    if b {
        fmt.Print(v, " ")
    } else {
        fmt.Println("Pop() failed!")
    }

    Push(100)
    traverse(stack)
    Push(200)
    traverse(stack)

    for i := 0; i < 10; i++ {
        Push(i)
    }

    for i := 0; i < 15; i++ {
        v, b := Pop(stack)
        if b {
            fmt.Print(v, " ")
        } else {
            break
        }
    }
    fmt.Println()
    traverse(stack)
}
```

如你所见，`stack.go` 的源代码比 `queue.go` 的稍微短一点，这主要是因为栈背后的思想比队列更简单。

执行 `stack.go` 将生成如下输出：

```
Pop() failed!
100 ->
200 -> 100 ->
9 8 7 6 5 4 3 2 1 0 200 100
Empty Stack!
```

> 现在为止，你已经知道了如何使用链表实现哈希表、队列和栈。这些例子会让你意识到，在一般情况下链表对于编程和计算机科学来说是多么的有效和重要！

container 包

在这一节中，我会介绍 Go 语言标准库 `container` 的用法。`container` 包支持三种数据结构：`heap`、`list` 和 `ring`。`container/heap`、`container/list` 和 `container/ring` 分别实现了这些数据结构。

有的人可能对环不太熟悉，环就是一个**循环链表**，也就是说环上的最后一个元素中的指针指向环上的第一个元素。从根本上来说，这意味着环上的每个节点都是平等的，并且没有开头和结尾。因此，从环上任一元素开始都能遍历整个环。

后面的三个小节将分别详细介绍 `container` 包中的每一个包。有个合理的建议：你应该先判断 Go 语言标准库 `container` 的功能是否满足你的需求，如果满足再加以使用，否则你应该使用自己实现的数据结构。

使用 container/heap

在这一小节中，你将了解 `container/heap` 包中提供的功能。首先，你应该知道的是 `container/heap` 实现的堆是一个树结构，树上的每个节点都是其所在子树上最小的元素。注意，我这里说的是最小的元素而不是最小的值是为了突出堆不止支持数值。

不过你可能已经猜到了，为了使用 Go 语言实现堆积树，你需要自己开发一种用来比较两个元素大小的方法。这种情况下，使用 Go 语言中的接口来定义比较合适。

这意味着 `container/heap` 包比 `container` 包中的其他两个包更加先进，而且你需要先完成一些定义之后才能使用 `container/heap` 包的功能。更准确地说， `container/heap` 包要求你实现 `container/heap.Interface` ，其定义如下：

```
type Interface struct {
    sort.Interface
    Push(x interface{}) // 将 x 添加为第 Len() 个元素
    Pop() interface{}   // 删除并返回第 Len() - 1 个元素。
}
```

第 7 章“反射和接口”中会更详细地介绍接口。目前，你只用记住实现 Go 语言的接口只需要实现接口中的函数和其中组合的其他接口，比如上面的例子中的 `sort.Interface` 、 `Push()` 函数和 `Pop()` 函数。 `sort.Interface` 中需要实现的函数包括 `Len()` 、 `Less()` 和 `Swap()` ，实现这些函数很有必要，因为实现排序的功能必须要先实现交换两个元素、计算需要排序的对象的值以及根据前面计算的值判断两元素大小这些功能。尽管你可能认为这工作量很大，但大多数情况下这些函数的实现要么很琐碎，要么很简单。

由于这一节的目的是为了阐明 `container/heap` 的用法而不是为了让你头疼，例子中元素的数据类型都将使用 `float32` 。

`conHeap.go` 中的 Go 语言代码将分为五个部分介绍。第一部分如下：

```
package main

import (
    "container/heap"
    "fmt"
)

type heapFloat32 []float32
```

`conHeap.go` 的第二部分是如下的 Go 代码：

```
func (n *heapFloat32) Pop() interface{} {
    old := *n
    x := old[len(old)-1]
    new := old[0 : len(old)-1]
    *n = new
    return x
}

func (n *heapFloat32) Push(x interface{}) {
    *n = append(*n, x.(float32))
}
```

尽管这里定义了 `Pop()` 和 `Push()` 这两个函数，但这只是为了遵循接口的要求。当你需要在堆中增删元素的时候还是应该分别调用 `heap.Push()` 和 `heap.Pop()` 。

`conHeap.go` 的第三个代码段包含如下的 Go 代码：

```
func (n heapFloat32) Len() int {
    return len(n)
}

func (n heapFloat32) Less(a, b int) bool {
    return n[a] < n[b]
}


func (n heapFloat32) Swap(a, b int) {
    n[a], n[b] = n[b], n[a]
}
```

这一部分实现了 `sort.Interface` 接口所需要的三个函数。

`conHeap.go` 的第四部分如下：

```
func main() {
    myHeap := &heapFloat32{1.2, 2.1, 3.1, -100.1}
    heap.Init(myHeap)
    size := len(*myHeap)
    fmt.Printf("Heap size: %d\n", size)
    fmt.Printf("%v\n", myHeap)
```

`conHeap.go` 的最后一个代码段如下：

```
    myHeap.Push(float32(-100.2))
    myHeap.Push(float32(0.2))
    fmt.Printf("Heap size: %d\n", len(*myHeap))
    fmt.Printf("%v\n", myHeap)
    heap.Init(myHeap)
    fmt.Printf("%v\n", myHeap)
}
```

`conHeap.go` 的最后一部分中使用 `heap.Push()` 向 `myHeap` 中添加了两个新元素。然而，为了让堆重新有序排列，你需要再次调用 `heap.Init()`

执行 `conHeap.go` 将生成如下输出：

```
$ go run conHeap.go
Heap size: 4
&[-100.1 1.2 3.1 2.1]
Heap size: 6
&[-100.1 1.2 3.1 2.1 -100.2 0.2]
&[-100.2 -100.1 0.2 2.1 1.2 3.1]
```

输出的最后一行中的 `2.1 1.2 3.1` 这三个数的顺序看上去有些不对，这是因为堆不是按照线性逻辑进行排序的。请记住，堆是一个树状结构，而不是像数组或切片这样的线性结构。

使用 container/list

在这一小节中将通过 `conList.go` 中的 Go 代码来阐明如何使用 `container/list` 包，下面分为三个部分介绍。

> `container/list` 包实现了链表

`conList.go` 的第一部分包含如下的 Go 代码：

```
package main

import (
    "container/list"
    "fmt"
    "strconv"
)

func printList(l *list.List) {
    for t := l.Back(); t != nil; t = t.Prev() {
        fmt.Print(t.Value, " ")
    }
    fmt.Println()

    for t := l.Front(); t != nil; t = t.Next() {
        fmt.Print(t.Value, " ")
    }

    fmt.Println()
}
```

这里有个函数叫 `printList()`，你可以传入一个 `list.List` 变量作为指针，然后输出其中所有的内容。这段 Go 代码展示了如何以从头到尾和从尾到头的两个方向输出 `list.List` 中的所有元素。通常，在你的程序中只需要使用其中的一种方式。你可以用 `Prev()` 和 `Next()` 函数来反向或正向迭代链表中的元素。

`conList.go` 中的第二个代码段如下：

```
func main() {

    values := list.New()

    e1 := values.PushBack("One")
    e2 := values.PushBack("Two")
    values.PushFront("Three")
    values.InsertBefore("Four", e1)
    values.InsertAfter("Five", e2)
    values.Remove(e2)
    values.Remove(e2)
    values.InsertAfter("FiveFive", e2)
    values.PushBackList(values)

    printList(values)

    values.Init()
```

你可以用 `list.PushBack()` 函数在链表尾部插入对象，也可以使用 `list.PushFront()` 函数在链表的头部插入对象。这两个函数的返回值都是链表中新插入的对象。如果你想在指定的元素后面插入新的元素，那么可以使用 `list.InsertAfter()` 函数。类似的，你应该使用 `list.InsertBefore()` 在指定元素的前面插入新元素，如果指定的元素不存在链表就不会改变。 `list.PushBackList()` 将一个链表的副本插入到另一个链表的后面。 `list.PushFrontList()` 函数将一个链表的副本插入到另一个链表的前面。 `list.Remove()` 函数从链表中删除一个指定的元素。

注意，使用 `values.Init()` 将会清空一个现有的链表或者初始化一个新的链表。

`conList.go` 的最后一部分代码如下：

```
    fmt.Printf("After Init(): %v\n", values)

    for i := 0; i < 20; i++ {
        values.PushFront(strconv.Itoa(i))
    }

    printList(values)
}
```

这里使用 `for` 循环创建了一个新的链表， `strconv.Itoa()` 函数将整数值转化为一个字符串。

总之， `container/list` 包中函数的用法很好理解，结果也不出意外。

执行 `conList.go` 将会生成如下的输出：

```
$ go run conList.go
Five One Four Three Five One Four Three
Three Four One Five Three Four One Five
After Init(): &{_{0xc420074180 0xc420074180 <nil> <nil>} 0}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

使用 container/ring

这一节中将使用 conRing.go 中的 Go 语言代码阐述 container/ring 包的用法，下面分为四个部分介绍。注意， container/ring 比 container/list 和 container/heap 都简单多了，也就是说这个包中的函数比另外两个包中的要少一些。

conRing.go 中的第一个代码段如下：

```
package main

import (
    "container/ring"
    "fmt"
)

var size int = 10
```

size 变量存储了要创建的环的大小。

conRing.go 的第二部分包含如下 Go 代码：

```
func main() {
    myRing := ring.New(size + 1)
    fmt.Println("Empty ring:", *myRing)

    for i := 0; i < myRing.Len()-1; i++ {
        myRing.Value = i
        myRing = myRing.Next()
    }

    myRing.Value = 2
```

从上面可知，创建新的环需要使用 ring.New() 函数，它需要接受一个提供环的大小的参数。最后的 myRing.Value = 2 语句向环中加入了 2 这个值。不过前面的 for 循环中已经向环中加入了那个值。最后，环的零值指的是只有一个值为 nil 的元素的环。

conRing.go 的第三部分如下：

```
    sum := 0
    myRing.Do(func(x interface{}) {
        t := x.(int)
        sum = sum + t
    })
    fmt.Println("Sum:", sum)
```

`ring.Do()` 函数可以对环上的每个元素依次调用一个函数。然而 `ring.Do()` 没有定义对环进行修改的行为。`x.(int)` 语句称为**类型断言**。第 7 章“反射和接口”中将详细介绍类型断言。目前，你只用知道这表示 `x` 是 `int` 类型的就行了。

`conRing.go` 的最后一部分程序如下：

```
    for i := 0; i < myRing.Len()+2; i++ {
        myRing = myRing.Next()
        fmt.Print(myRing.Value, " ")
    }
    fmt.Println()
}
```

使用环会遇到的唯一的问题就是你可以无限调用 `ring.Next()`，所以你需要找到停下来的办法。这种情况下就需要用到 `ring.Len()` 函数。就个人而言，我比较倾向于使用 `ring.Do()` 函数来迭代环上的所有元素，因为这样代码更简洁，但用 `for` 循环其实也不错！

执行 `conRing.go` 将会生成如下输出：

```
$ go run conRing.go
Empty ring: {0x42000a080 0xc42000a1a0 <nil>}
Sum: 47
0 1 2 3 4 5 6 7 8 9 2 0 1
```

输出的结果证明环上可以存在重复的值，也就是说你只能通过 `ring.Len()` 函数才能安全地获取到环的大小。

生成随机数

随机数的生成是计算机科学的一个研究领域，同时也是一种艺术。这是因为计算机是纯粹的逻辑机器，所以使用计算机生成随机数异常困难！

你可以用 `math/rand` 包来生成随机数。开始生成随机数之前首先需要一个种子，种子用于整个过程的初始化，它相当重要。因为如果你每次都用同样的种子初始化，那么就总是会得到相同的随机数序列。这意味着每个人都可以重新生成同一个序列，那这个序列就根本不能再算是随机的了！

我们将用来生成随机数的程序名为 `randomNumbers.go`，下面分成四个部分进行介绍。这个程序需要几个参数，分别是生成随机数的下限、生成随机数的上限、生成随机数的数量。如果你还用了第四个命令参数，那么程序会将它作为随机数生成器的种子。在测试的时候，你就可以再次使用这个参数生成相同的数列。

程序的第一部分如下所示：

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

`random()` 函数完成了所有的工作，它通过根据指定的范围调用 `rand.Intn()` 生成随机数。

这个命令行程序的第二部分如下：

```
func main() {
    MIN := 0
    MAX := 100
    TOTAL := 100
    SEED := time.Now().Unix()

    arguments := os.Args
```

这一部分对程序中将会用到的变量进行了初始化。

`randomNumbers.go` 的第三部分包含如下的 Go 代码：

```
    switch len(arguments) {
    case 2:
        fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
        MIN, _ = strconv.Atoi(arguments[1])
        MAX = MIN + 100
    case 3:
        fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
        MIN, _ = strconv.Atoi(arguments[1])
        MAX, _ = strconv.Atoi(arguments[2])
    case 4:
        fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
        MIN, _ = strconv.Atoi(arguments[1])
        MAX, _ = strconv.Atoi(arguments[2])
        TOTAL, _ = strconv.Atoi(arguments[3])
    case 5:
        MIN, _ = strconv.Atoi(arguments[1])
        MAX, _ = strconv.Atoi(arguments[2])
        TOTAL, _ = strconv.Atoi(arguments[3])
        SEED, _ = strconv.ParseInt(arguments[4], 10, 64)
    default:
        fmt.Println("Using default values!")
    }
```

`switch` 代码块背后的逻辑相对简单：根据命令行参数的数量决定程序中的参数是使用缺省的默认值还是使用用户提供的值。为了简化程序，`strconv.Atoi()` 和 `strconv.ParseInt()` 函数返回的 `error` 参数使用下划线字符接收，然后被忽略。如果是商业程序就千万不能忽略 `strconv.Atoi()` 和 `strconv.ParseInt()` 函数返回的 `error` 参数！

最后，使用 `strconv.ParseInt()` 对 `SEED` 变量赋新的值是因为 `rand.Seed()` 函数要求的参数类型是 `int64` 。`strconv.ParseInt()` 的第一个参数是要解析的字符串，第二个参数是输出数的基数，第三个参数是输出数的位数。由于我们想要生成的是一个 64 位的十进制整数，所以使用 `10` 作为基数，使用 `64` 作为位数。注意，如果你想解析无符号的整数的话应该使用 `strconv.ParseUint()` 函数代替。

`randomNumbers.go` 的最后一部分是如下的 Go 代码：

```
    rand.Seed(SEED)
    for i := 0; i < TOTAL; i++ {
        myrand := random(MIN, MAX)
        fmt.Print(myrand)
        fmt.Print(" ")
    }
    fmt.Println()
}
```

> 除了使用 Unix 时间戳作为随机数生成器的种子，你还可以使用 `/dev/random` 这个系统设备。你可以在第 8 章“Go Unix系统编程”中了解 `/dev/random` 的相关内容。

执行 `randomNumbers.go` 将会生成如下输出：

```
$ go run randomNumbers.go
75 69 15 75 62 67 64 8 73 1 83 92 7 34 8 70 22 58 38 8 54 91 65
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
1 1
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
2 2
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
```

如果你对随机数生成真的很有兴趣，那么你应该先读一下 *Donald E.Knuth* 写的 *The Art of Computer Programming (Addison-Wesley Professional, 2011)* 的第二卷。

如果需要用 Go 生成更安全的随机数的话，你可以使用 `crypto/rand` 包。这个包中实现了满足密码学安全的伪随机数生成器。你可以在 https://golang.org/pkg/crypto/rand/ 文档页面了解更多关于 `crypto/rand` 包的信息。

生成随机字符串

一旦你知道了计算机是如何呈现出单个字符，从随机数过渡到随机字符串就不难了。这一节将会介绍如何使用前一节中 `randomNumbers.go` 的代码生成难以猜出的密码。用于完成这个任务的 Go 程序叫做 `generatePassword.go`，下面将分四个部分进行介绍。这个程序只需要一个命令行参数，就是你需要生成的密码的长度。

`generatePassword.go` 的第一部分包含如下的 Go 代码：

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

`generatePassword.go` 的第二个代码段如下：

```
func main() {
    MIN := 0
    MAX := 94
    SEED := time.Now().Unix()
    var LENGTH int64 = 8

    arguments := os.Arg
```

我们只想得到可打印的 ASCII 字符，所以对生成随机数的范围进行了限制。ASCII 字符表中可打印的字符一共有 94 个。这意味着该程序生成的随机数的范围应该是从 0 到 94 且不包括 94。

`generatePassword.go` 的第三个代码段如下：

```
    switch len(arguments) {
    case 2:
        LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
    default:
        fmt.Println("Using default values!")
    }

       rand.Seed(SEED)
```

`generatePassword.go` 的最后一部分如下：

```
    startChar := "!"
    var i int64 = 1
    for {
        myRand := random(MIN, MAX)
        newChar := string(startChar[0] + byte(myRand))
        fmt.Print(newChar)
        if i == LENGTH {
            break
        }
        i++
    }
    fmt.Println()
}
```

`startChar` 参数保存了这个程序可以生成的第一个 ASCII 字符，也就是十进制 ASCII 值为 33 的感叹号。已知该程序生成的随机数小于 94，可以计算出生成的最大的 ASCII 值为 93 + 33，等于 126，也就是 `~` 的 ASCII 值。下面的输出是含有十进制数值的 ASCII 字符表：

```
The decimal set:

  0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack
  8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so
 16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn
 24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs
 32 sp    33  !    34  "    35  #    36  $    37  %    38  &
 40  (    41  )    42  *    43  +    44  ,    45  -    46  .
 48  0    49  1    50  2    51  3    52  4    53  5    54  6
 56  8    57  9    58  :    59  ;    60  <    61  =    62  >
 64  @    65  A    66  B    67  C    68  D    69  E    70  F
 72  H    73  I    74  J    75  K    76  L    77  M    78  N
 80  P    81  Q    82  R    83  S    84  T    85  U    86  V
 88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^
 96  `    97  a    98  b    99  c   100  d   101  e   102  f   1
104  h   105  i   106  j   107  k   108  l   109  m   110  n   1
112  p   113  q   114  r   115  s   116  t   117  u   118  v   1
120  x   121  y   122  z   123  {   124  |   125  }   126  ~   1
```

> 在你最喜欢的 Unix shell 中输入 `man ascii` 就能生成易读的 ASCII 字符表。

执行 `generatePassword.go` 并传入合适的命令行参数将生成如下输出：

```
$ go run generatePassword.go
Using default values!
ugs$5mv1
$ go run generatePassword.go
Using default values!
PA/8hA@?
$ go run generatePassword.go 20
HBR+=3\UA'B@ExT4QG|o
$ go run generatePassword.go 20
XLcr|R{*pX/::'t2u^T'
```

延展阅读

下面这些资源很有用：

- 仔细查看 **Graphviz** 程序的网页。你可以使用这个程序提供的语言画图：http://graphviz.org。
- 查看 https://golang.org/pkg/container 并阅读 Go 标准库 `container` 中子包的文档页面。
- 如果你想进一步了解数据结构的话可以阅读 *Alfred V.Aho*、*John E.Hopcroft* 和 *Jeffrey D.Ullman* 写的 *The Design and Analysis of Computer Algorithms (Addison-Wesley, 1974)*，这本书非常棒！
- 还有基本关于算法和数据结构的书也很有趣，分别是 *Programming Pearls (Addison-Wesley Professional, 1999)* 和 *More Programming Pearls: Confessions of a Coder (Addison-Wesley Professional, 1988)*，这两本书都是 *John Bentley* 写的。好好读这两本书可以让你成为一个更优秀的程序员！

练习

- 尝试修改将 `generatePassword.go` 背后的逻辑，新的程序将使用当前系统时间或日期生成一个 Go 切片，然后使用这个切片生成一个密码列表，再从中挑选一个密码。
- 修改 `queue.go` 的代码使其能够存储浮点数而不是整数。
- 修改 `stack.go` 的代码，使其节点拥有 `Value` 、 `Number` 和 `Seed` 三个整型的数据字段。除了明显要对 `Nodestruct` 的定义进行修改，你还需要对剩下的程序做出什么改动？
- 你能试试修改 `linkedList.go` 的代码，使其能够保持链表中的节点有序吗？
- 类似的，你能修改 `doublyLList.go` 中代码使其能保持节点有序吗？你能开发一个删除已存在节点的函数吗？
- 修改 `hashTableLookup.go` 的代码使得哈希表中没有重复的元素。你可以使用 `lookup()` 函数！
- 你能修改 `generatePassword.go` 中的代码使生成的密码只包含大写字母吗？
- 试试修改 `conHeap.go` 使其除了能支持 `float32` 元素外还能支持自定义的、更复杂的结构。
- 实现 `linkedList.go` 中缺失的删除节点的功能。
- 你觉得如果使用双向链表的话是否能改善 `queue.go` 程序中的代码呢？试着用双向链表代替单向链表实现一个队列。

本章小结

在这一章中我们讨论了很多 Go 语言数据结构相关的有趣又实用的话题，包括链表、双向链表、哈希表、队列和栈的实现，还有 Go 语言标准库 `container` 的用法以及如何使用 Go 生成随机数和难以猜测的密码。通过学习这一章你应该记住，每种数据结构的基础都是其中节点的定义和实现。

我确信你会觉得下一章是本书中最有趣并且最有价值的一章。下一章的主题是 Go 语言的包。下一章还介绍了如何在你的程序中定义并使用各种类型的 Go 函数。所以别耽误时间了，继续往后看吧！

Go package中不为人知的知识

上章，我们讨论了开发和使用自定义数据结构，如链表，字节树，哈希表，还有生产随机数和用 Go 猜测密码。

这章主要讲 Go package，它是 Go 组织，交付，使用代码的方式。Go package 非常通用的组成部分是 函数，它相当的灵活。这章的最后部分您将看到一些高级的 Go 标准库 package，会更好地理解创建 Go package 的不同方式。

这章，您将了解到如下主题：

- 用 Go 编写函数
- 匿名函数
- 返回多值函数
- 命名函数的返回值
- 返回其他函数的函数
- 函数作为参数的函数
- 编写 Go package
- 私有和公有 package 对象
- 在包中使用 `init()` 函数
- 复杂的 `html/templates` 标准 package
- `text/template` 标准 package，和另一个真正复杂的有自己语言的 Go package
- `syscall` 标准包，它是一个底层的包，尽管您可能很少有机会使用它，但它被其他包广泛使用。

关于Go packages

在 Go 中所有东西都以 packages 形式传递。Go **package** 是 Go 源文件，它以 `package` 关键字开头后面跟着这个 package 的名字。一些 packages 有结构。例如， `net` package 有几个子目录，分别是 `http`, `mail`, `rpc`, `smtp`, `textproto` 和 `url` ，它们应该分别依 `net/http`, `net/mail`, `net/rpc`, `net/smtp`, `net/textproto` 和 `net/url` 引入。

packages 主要用于组合相关函数，变量和常量，以便您可以在自己的程序中简单地传递和使用它们。注意 `main` package 里的内容，Go packages 不是自控制程序并且不能被编译为可执行文件。这意味着他们需要直接或间接的在 `main` package 里调用才能使用。因此，如果您把它当成自控制程序来执行，您会失望的：

```
$go run aPakcage.go
go run: cannot run non-main package
```

Go函数

函数 在任何编程语言中都是重要的元素，因为它可以让您把大块的程序分成比较小而且好管理的小块。函数必须尽可能的彼此独立，而且必须完成单一功能，最好只做一个功能。因此，如果您发现您写的函数做了多个功能，那么您需要考虑用多个函数来取代它！Go 中唯一最常用的函数是 `main()`，它被用在每个独立的 Go 程序中。您应该已经知道了函数的定义用 `func` 关键字开始。

匿名函数

匿名函数 可以在不需要名称的情况下直接定义，它们通常用于实现只需很少代码量的功能。Go 中，函数可以返回一个匿名函数或把匿名函数作为它的参数。另外，匿名函数可以分配给变量。

注意匿名函数也叫**闭包**，特别是在函数式编程术语中。

> *匿名函数好的表现是实现内容小并聚焦局部。如果一个匿名函数没有聚焦局部，那么您可能需要把它改为正常函数*

当匿名函数适合一个功能时，它极其方便和容易；只是在没有好的原因时不要使用太多匿名函数。您稍后会看到匿名函数是如何工作的。

多返回值的函数

如您已经从 `strconv.atoi()` 等函数所知，Go 函数可以返回多个不同的值，这就避免了您必须创建一个专用的结构，以便能够同时从一个函数接收多个值。您可以声明一个如下返回四个值的函数，两个 `int` 值，一个 `float64` 值和一个 `string`：

```
func aFunction() (int, int, float64, string){
}
```

现在来以 `functions.go` 做例子来说明匿名函数和返回多值的函数。相关代码分五部分来介绍。

`functions.go` 的第一段代码如下：

```
package main

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

`functions.go` 的第二段代码如下：

```
func doubleSquare(x int) (int, int) {
    return x * 2, x * x
}
```

这里您可以看到一个名为 `doubleSquare()` 函数的定义和实现，它只需要一个 `int` 参数并返回两个 `int` 值。

`functions.go` 的第三部分如下：

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("The program needs 1 argument!")
        return
    }
    y, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
```

上面的代码处理程序的命令行参数。

`functions.go` 的第四部分包含如下代码：

```
    square := func(s int) int {
        return s * s
    }
    fmt.Println("The squre of", y, "is", square(y))

    double := func(s int) int {
        return s + s
    }
    fmt.Println("The double of", y, "is", double(y))
```

`square` 和 `double` 变量分别对应一个匿名函数。糟糕的部分是 `squre`， `double` 的值或任何其他对应匿名函数的变量值可以被修改，这意味着，这些变量能修改和计算其他的。

> 不要认为可以修改匿名函数变量的代码是好的程序实现，因为这可能是错误的根源！

`functions.go` 的最后一部分如下：

```
    fmt.Println(doubleSquare(y))
    d, s := doubleSquare(y)
    fmt.Println(d, s)
}
```

您可以打印如 `doubleSquare()` 函数的返回值，也可以把它们分配给其他变量。

执行 `functions.go` 产生如下输出：

```
$go run functions.go 1 21
The program needs 1 argument!
rMackBook: code mtsouk
$go run functions.go 10
The square of 10 is 100
The double of 10 is 20
20 100
20 100
```

可命名的函数返回值

和 C 不同， Go 可以命名函数的返回值。另外，当一个函数有个没有任何参数的 `return` 语句时，函数自动返回每个命名的返回值的当前值，返回顺序与函数定义中声明的顺序相同。

> 命名返回值是一个非常方便的特性，它可以避免各种错误，因此使用它吧！我的个人建议是命名函数的返回值，除非有好的理由不使用它们。

说明命名函数返回值的源码是 `returnNames.go` ，分为三部分介绍。

`returnNames.go` 第一部分如下：

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

func namedMinMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return
}
```

在这段代码中，您可以看到 `namedMinMax()` 函数的实现，它使用了命名返回值参数。然而，有一点： `namedMinMax()` 函数没有明确的返回任何变量在 `return` 语句。不过，由于这个函数命令了返回值， `min` 和 `max` 被自动返回，以它们被定义的顺序。

`returnNames.go` 的第二段代码如下：

```
func minMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return min, max
}
```

`minMax()` 函数也使用了命名返回值，但它的 `return` 语句指定了返回值的顺序和变量。

`returnNames.go` 的最后一段代码如下：

```
func main() {
    arguments := os.Args
    if len(arguments) < 3 {
        fmt.Println("The program needs at least 2 arguments!")
        return
    }
    a1, _ := strconv.Atoi(arguments[1])
    a2, _ := strconv.Atoi(arguments[2])

    fmt.Println(minMax(a1, a2))
    min, max := minMax(a1, a2)
    fmt.Println(min, max)

    fmt.Println(namedMinMax(a1, a2))
    min, max = namedMinMax(a1, a2)
    fmt.Println(min, max)
}
```

`main()` 函数中的代码目的是验证所有方法产生相同的结果。

执行 `returnNames.go` 将输出如下：

```
$go run returnNames.go -20 1
-20 1
-20 1
-20 1
-20 1
```

参数为指针的函数

函数可以有指针类型的参数。`ptrFun.go` 的代码将说明指针作为函数参数的使用。

`ptfFun.go` 的第一部分如下：

```
package main

import (
    "fmt"
)

func getPtr(v *float64) float64 {
    return *v * *v
}
```

`getPrt()` 函数接收一个指向 `float64` 值的指针。

第二段代码如下：

```
func main() {
    x := 12.2
    fmt.Println(getPrt(&x))
    x = 12
    fmt.Println(getPtr(&x))
}
```

这里您需要传递变量的地址给 `getPtr()` 函数，因为它需要一个指针参数，传地址需要在变量前放一个 & 符号（&x）。

执行 `ptrFun.go` 产生如下输出：

```
$go run ptrFun.go
148.83999999999997
144
```

如果您直接传值如：`12.2` 给 `getPtr()`，并调用它如：`getPtr(12.12)`，那么程序编译失败，显示下面的错误信息：

```
$go run ptrFun.go
# command-line-arguments
./ptrFun.go:15:21: cannot use 12.12 (type float64) as type *floa
```

返回值为指针的函数

如您在第四章了解到的，在 `pointerStruct.go` 程序中介绍的*复合类型的使用*，它是一个非常好的练习，使用一个独立的函数来创建一个新的结构变量并返回指向它的指针。因此，函数返回指针是非常常见的。通常讲，函数简化程序结构并让开发者把较重要的处理逻辑集中起来，而不是总是复制相同的代码。这节将使用一个非常简单的例子， `returnPtr.go` 。

`returnPtr.go` 的第一部分代码如下：

```
package main

import (
    "fmt"
)

func returnPtr(x int) *int {
    y := x * x
    return &y
}
```

除了必须的引入部分，这段代码定义了一个返回 `int` 变量指针的新函数。唯一要记住的是使用 `&y` 在 `return` 表达式来返回 `y` 变量的内存地址。

`returnPtr.go` 的第二部分如下：

```
func main() {
    sq := returnPtr(10)
    fmt.Println("sq:", *sq)
```

如您所知， `*` 符号**解引用一个指针变量**，就是它返回存储在内存地址里的实际值而不是内存地址本身。

`returnPtr.go` 的最后一段代码如下：

```
    fmt.Println("sq:", sq)
}
```

这段代码返回 `sq` 变量的内存地址，而不是存储在它里面的 `int` 值。

执行 `returnPtr.go` 将看到如下输出（内存地址可能会不同）：

```
$go run returnPtr.go
sq: 100
sq: 0xc420014088
```

闭包

这节，您将了解到使用 `returnFunction.go` 的代码怎样实现闭包，分为三部分来介绍。

`returnFunction.go` 的第一部分如下：

```
package main

import (
    "fmt"
)

func funReturnFun() func() int {
    i := 0
    return func() int {
        i++
        return i * i
    }
}
```

如您所见， `funReturnFun()` 函数返回一个匿名函数！

`returnFunction.go` 的第二段代码如下：

```
func main(){
    i := funReturnFun()
    j := funReturnFun()
```

这段代码，调用 `funReturnFun()` 俩次并分配函数类型的返回值给名为 `i` 和 `j` 的两个特定变量。从程序的输出可以看出，这两个变量是完全无关的。

`returnFunction.go` 的最后一段代码如下：

```
    fmt.Println("1:", i())
    fmt.Println("2:", i())
    fmt.Println("j1:", j())
    fmt.Println("j2:", j())
    fmt.Println("3:", i())
}
```

这段代码里，以 `i()` 使用 `i` 变量三次，以 `j()` 使用 `j` 变量两次。重要的是尽管 `i` 和 `j` 是由调用 `funReturnFun()` 创建，但它们完全独立，不共享任何内容。所以，尽管它们从同样的序列返回值但它们彼此

不干扰。

执行 `returnFunction.go` 产生如下输出：

```
$go run returnFunction.go
1: 1
2: 4
j1: 1
j2: 4
3: 9
```

从 `returnFunction.go` 的输出可以看到， `funReturnFun()` 函数里的 `i` 值保持增加并不会在每次调用 `i()` 或 `j()` 后变为 `0`。

函数作为参数

Go 函数可以接收另一个 Go 函数作为参数，这个功能为您用 Go 函数增加了非常广泛的用途。这个功能最为常用的用途就是闭包。这里介绍的 `funFun.go`，是一个非常简单的处理整数值的例子。相关代码分为三部分介绍。

`funFun.go` 的第一部分代码如下：

```
package main

import "fmt"

func function1(i int) int {
    return i + i
}

func function2(i int) int {
    return i * i
}
```

这里是两个接收 `int` 并返回 `int` 的函数。它们一会用于作为另个函数的参数。

`funFun.go` 的第二段函数包含如下代码：

```
func funFun(f func(int) int, v int) int {
    return f(v)
}
```

`funFun()` 函数接收两个参数，一个名为 `f` 的函数和一个 `int` 值。`f` 参数是一个接收 `int` 参数并返回 `int` 值的函数。

`funFun.go` 的最后一段如下：

```
func main() {
    fmt.Println("function1:", funFun(function1, 123))
    fmt.Println("function2:", funFun(function2, 123))
    fmt.Println("Inline:", funFun(func(i int) int {return i * i
}
```

第一个 `fmt.Println()` 调用使用 `function1` 作为第一个参数的 `funFun()`，第二个 `fmt.Println()` 调用使用 `function2` 作为第一个参数的 `funFun()`。

最后一个 `fmt.Println()` 表达式，奇妙的是：这个函数参数的实现定义在 `funFun()` 调用里！

尽管这个方法可以运行简单的，小巧的函数参数，对于多行代码的函数还是尽量不要这样使用。

执行 `funFun.go` 将产生如下输出：

```
$go run funFun.go
function1: 246
function2: 15129
Inline: 1860867
```

设计你的Go packages

Go 包的源码可以包括多个文件和多个目录，可以在以包名称命名的单个目录找到，除了 `main` 包，它可以放在任意位置。

这章的目标是开发一个名为 `aPackage` 的简单 Go 包。这个包的源文件命名为 `aPackage.go` ，它的源码分为两部分介绍。

`aPackage.go` 的第一部分如下：

```
package aPackage

import (
    "fmt"
)

func A() {
    fmt.Println("This is function A!")
}
```

`aPackage.go` 的第二部分代码如下：

```
func B() {
    fmt.Println("privateConstant:", privateConstant)
}

const MyConstant = 123
const privateConstant = 21
```

如您所见，开发一个新包是相当简单的！现在您不能单独使用这个包，需要创建一个名为 `main` 有 `main()` 函数的包来创建执行文件。这里，使用 `aPackage` 的程序名是 `useAPackage.go` ，代码如下：

```
package main

import (
    "aPackage"
    "fmt"
)

func main() {
    fmt.Println("Using aPackage!")
    aPackage.A()
    aPackage.B()
    fmt.Println(aPackage.MyConstant)
}
```

如果现在执行 `useAPackage.go`，将会得到一个错误信息，告诉您不能执行：

```
$go run useAPackage.go
useAPackage.go:4:2: cannot find package "aPackage" in any of:
    /use/local/Cellar/go/1.9.2/libexec/src/aPackage (from $GOROO
    /Users/mtsouk/go/src/aPackage (from $GOPATH)
```

还有一件事需要处理，正如在 *第一章——Go与操作系统* 了解到的，Go 需要从 Unix shell 执行特定命令来安装所有外部包，也包括您自己开发的。因此，您需要把之前的包放在正确的目录下并对当前 Unix 用户可用。所以安装您自己的包，需要执行如下命令：

```
$mkdir ~/go/src/aPackage
$cp aPackage.go ~/go/src/aPackage/
$go install aPackage
$cd ~/go/pkg/dawin_amd64/
$ls -l aPackage.a
-rw-r--r-- 1 mtsouk staff 4980 Dec 22 06:12 aPackage.a
```

如果 `~/go` 不存在，您需要使用 `mkdir(1)` 命令创建它。您也需要有相同的 `~/go/src` 目录。

使用 `useAPackage.go` 将产生如下输出：

```
$go run useAPackage.go
Using aPackage!
This is function A!
privateConstant: 21
123
```

类型方法

Go的**类型方法**是带有特殊参数的函数。在一个声明为普通函数的方法函数名前面，增加一个特定参数，该特定参数将函数与该参数的类型进行关联，该特定参数被称为**方法的接收器**。

下面的Go代码是在https://golang.org/src/os/file_plan9.go中的 `Close()` 函数的实现：

```
func (f *File) Close() error {
    if err := f.checkValid("close"); err != nil {
        return err
    }
    return f.file.close()
}
```

`Close()` 函数就是**类型方法**，因为函数名称前面和 `func` 关键字后面有 `(f *File)` 参数。 `f` 参数被称为**方法的接收器**；在面向对象编程术语中，这个过程可以描述为向**对象**发送消息。在Go语言中，**方法的接收器**是使用常规变量名定义的，通常使用单个字母，而不需要使用专用关键字，如 `this` 或 `self` 。

接下来我们使用 `methods.go` 文件的Go代码来呈现一个完整的示例，包含以下四部分。

```
> ```go
> package main
>
> import (
>     "fmt"
> )
>
> type twoInts struct {
>     X int64
>     Y int64
> }
>
```

在前面的代码中，定义了结构体 `twoInts` ，该结构体包含两个 `int64` 类型的字段。

```
> ```go
> func regularFunction(a, b twoInts) twoInts {
>     temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}
>     return temp
> }
>
```

在本部分中，定义了一个名为 `regularFunction()` 的函数，该函数接收两个 `twoInts` 类型的参数，并返回一个 `twoInts` 类型的值。

```
> ```go
> func (a twoInts) method(b twoInts) twoInts {
>     temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}
>     return temp
> }
>
```

```
> _这里真正有趣的是，**类型方法**```method()```与普通函数```regular

```methods.go```的最后一个代码段如下：

> ```go
> func main() {
>     i := twoInts{X: 1, Y: 2}
>     j := twoInts{X: -5, Y: -2}
>     fmt.Println(regularFunction(i, j))
>     fmt.Println(i.method(j))
> }
>
```

如您所见，您调用**类型方法**（ `i.method(j)` ）的方式是与普通函数（ `regularFunction(i, j)` ）的调用方式不同。

执行 `methods.go` 的输出如下：

```
$ go run methods.go
{-4 0}
{-4 0}
```

值得注意的是，**类型方法**也与**接口**相关，下一节中将对**接口**进行讲解，稍后也将看到更多**类型方法**的使用。
```

Go的接口

严格来说，Go的 `interface` 类型是一组需要被其他数据类型实现的函数方法的集合。对于需要满足接口的数据类型，它需要实现接口所需的所有方法。简单地说，接口是定义了一组需要被实现的函数方法的**抽象类型**，实现接口的数据类型可以视为接口的实例。如果满足以上情况，我们说数据类型实现了这个接口。因此，一个接口包含两部分：一组接口方法和一个接口类型，这样就可以定义其它数据类型的行为。

使用接口的最大优势，就是在定义函数时使用接口作为参数，那么在调用函数时，任何实现该接口类型的变量都可以作为函数的参数被传递。接口可以作为抽象类型的能力，是其最实际的价值。

> 如果发现在同一个Go包中定义了一个接口及其实现，那么可能使用了错误的接口实现方式！

两个非常常见的Go接口是 `io.Reader` 和 `io.Writer` ，用于文件输入和输出操作。具体来说， `io.Reader` 用于读取文件，而 `io.Writer` 用于写入文件。

在代码https://golang.org/src/io/io.go中， `io.Reader` 的定义如下：

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

为了使类型满足 `io.Reader` 接口，需要实现接口中描述的 `Read()` 方法。

同样，在代码https://golang.org/src/io/io.go中的 `io.Writer` 的定义如下：

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

为满足 `io.Writer` 接口，只需要实现一个名为 `Write()` 的方法。

尽管每个 `io.Reader` 和 `io.Writer` 接口都只需要实现一个方法，但是这两个接口功能都非常强大。他们功能的强大都源于他们的简单！一般来说，接口应该尽可能简单。

在下一章节中，将讲解如何定义一个接口，以及如何在其他Go包中使用。需要注意的是一个接口只要做想要实现的事情，不需要过于花哨或令人印象深刻。

> 简单地说，当需要一个Go元素确保满足某些条件或者达到某些预期行为时，应该使用接口。

类型断言

类型断言的表示方法是 `x.(T)` ，其中 `x` 是接口类型的变量， `T` 是要判断的类型。也就是说，存储在 `x` 接口类型的变量中的实际值是 `T` 类型，并且 `T` 必须满足 `x` 的接口类型变量！以下的段落和代码示例，将澄清**类型断言**的这个相对不容易理解的概念。

类型断言做两件事：第一件事是检查接口类型变量是否是特定的类型，这样使用时，类型断言返回两个值：基础值和 `bool` 值。虽然基础值是您可能想要使用的值，但是布尔值告诉您类型断言是否成功！

类型断言所做的第二件事是允许您使用存储在接口中的具体值或将其分配给新变量。这意味着如果接口中有一个 `int` 变量，可以使用类型断言获取该值。

但是，如果类型断言不成功，并且没有处理该失败，那么您的程序将触发 `panic` 异常。接下来我们分两部分介绍 `assertion.go` 程序。第一部分包含以下Go代码：

```
package main

import (
    "fmt"
)

func main() {
    var myInt interface{} = 123
    k, ok := myInt.(int)
    if ok {
        fmt.Println("Success:", k)
    }
    v, ok := myInt.(float64)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("Failed without panicking!")
    }
```

首先，声明 `myInt` 变量，该变量具有动态类型 `int` 和值 `123` 。然后使用类型断言测试两次 `myInt` 变量的接口类型 — 分别是 `int` 类型和 `float64` 类型。

由于 `myInt` 变量不包含 `float64` 值，因此类型断言 `myInt.(float64)` 执行时，如果没有恰当的处理，则会引发错误。因此在这种情况下，使用 `ok` 变量来判断类型断言是否成功，将使程序免于 `panic` 异常。

以下Go代码是 `assertion.go` 程序的第二部分：

```
    i := myInt.(int)
    fmt.Println("No cheking:", i)
    j := myInt.(bool)
    fmt.Println(j)
}
```

这里有两种类型的断言。第一个类型断言是成功的，因此不会有任何问题。但是，让我们进一步回顾一下这个特定类型的断言。变量 `i` 的类型为 `int` ，其值为 `123` ，存储在 `myInt` 中。因此由于 `int` 满足 `myInt` 接口，并且 `myInt` 接口不需要实现接口函数，所以 `myInt.(int)` 的值是一个 `int` 值。

然而第二个类型断言 `myInt.(bool)` 将触发 `panic` 异常，因为 `myInt` 的基础值不是布尔值（ `bool` ）。因此执行 `assertion.go` 将生成以下输出：

```
$ go run assertion.go
Success: 123
Failed without panicking!
No cheking: 123
panic: interface conversion: interface {} is int, not bool

goroutine 1 [running]:
main.main()
      /Users/mtsouk/Desktop/masterGo/ch/ch7/code/assertion.
exit status 2
```

程序的执行结果明确的给出了触发 `panic` 异常的原因： `interface{} is int, not bool` 。

一般来说，在使用接口时，也应该使用类型断言。您马上会在 `useInterface.go` 程序中看到更多类型断言的使用案例。

设计接口

在本节中，您将学习如何设计接口。只要您知道要设计的接口行为，这个过程是相对简单的。

该章节将使用Go代码 `myInterface.go` 进行说明，这个代码将创建一个接口来辅助平面的几何图形的相关运算。

```
>```go
> package myInterface
>
> type Shape interface {
>     Area() float64
>     Perimeter() float64
> }
>
```

接口 `shape` 的定义是非常简单直接的，它只需要实现两个名为 `Area()` 和 `Perimeter()` 的函数，两个函数都返回 `float64` 值。第一个函数将用于计算平面形状的面积，第二个函数用于计算平面形状的周长。之后，您需要安装 `myInterface.go` 包，并使其对当前用户可用。正如你已经知道的，安装过程涉及以下Unix命令的执行：

```
$ mkdir ~/go/src/myInterface
$ cp myInterface.go ~/go/src/myInterface
$ go install myInterface
```

接口的使用

本小节将教您如何在名为 `useInterface.go` 的Go程序中使用 `myInterface.go` 中定义的接口，该程序将分五部分介绍。

```
>```go
> package main
>
> import (
>     "fmt"
>     "math"
>     "myInterface"
> )
>
> type square struct {
>     X float64
> }
>
> type circle struct {
>     R float64
> }
>
```

由于所需的接口定义在它自己的包中，所以需要导入 `myInterface` 包。

```
>```go
> func (s square) Area() float64 {
>     return s.X * s.X
> }
>
> func (s square) Perimeter() float64 {
>     return 4 * s.X
> }
>
```

在本部分中，类型 `square` 实现了 `shape` 接口。

第三部分包含以下Go代码：

```go
func (s circle) Area() float64 {
    return s.R * s.R * math.Pi
}

func (s circle) Perimeter() float64 {
    return 2 * s.R * math.Pi
}
```

在本部分中，类型 `circle` 实现了 `shape` 接口。

```
>```go
> func Calculate(x myInterface.Shape) {
>     _, ok := x.(circle)
>     if ok {
>         fmt.Println("Is a circle!")
>     }
>
>     v, ok := x.(square)
>     if ok {
>         fmt.Println("Is a square:", v)
>     }
>
>     fmt.Println(x.Area())
>     fmt.Println(x.Perimeter())
> }
>
```

在上面的代码中，实现了一个需要 `shape` 参数（ `myInterface.Shape` ）的函数。这里需要理解注意的是函数参数是任何 `shape` 类型的参数，也就是实现 `shape` 接口的类型！

函数开头的代码展示了如何区分实现接口的数据类型。在第二个代码块中，将看到如何获取存储在 `square` 参数中的值。对实现 `myInterface.Shape` 的任何类型都可以使用这样的方式。

最后一个代码段包括以下代码：

```go
func main() {
    x := square{X: 10}
    fmt.Println("Perimeter:", x.Perimeter())
    Calculate(x)
    y := circle{R: 5}
    Calculate(y)
}
```

在本部分中，将展示如何使用 `circle` 和 `square` 变量作为前面实现的 `Calculate()` 函数的参数。

如果执行 `useInterface.go` ，将得到以下输出：

```
$ go run useInterface.go
Perimeter: 40
Is a square: {10}
100
40
Is a circle!
78.53981633974483
31.41592653589793
```

Switch用于类型判断

在本小节中，将在Go代码 `switch.go` 中讲解如何使用 `switch` 语句来区分不同的数据类型，这将分为四个部分进行介绍。 `switch.go` 的Go代码部分基于 `useInterface.go` ，但它将添加另一个名为 `rectangle` 的类型，不需要实现任何接口的方法。

```
>```go
> package main
>
> import (
>     "fmt"
> )
>
```

由于 `switch.go` 中的代码不需要实现 `myInterface.go` 中定义的接口，因此不需要导入 `myInterface` 包。

```
>```go
> type square struct {
>     X float64
> }
>
> type circle struct {
>     R float64
> }
>
> type rectangle struct {
>     X float64
>     Y float64
> }
>
```

这三种类型都很简单。

```
>```go
> func tellInterface(x interface{}) {
>     switch v := x.(type) {
>     case square:
>         fmt.Println("This is a square!")
>     case circle:
>         fmt.Printf("%v is a circle!\n", v)
>     case rectangle:
>         fmt.Println("This is a rectangle!")
>     default:
>         fmt.Printf("Unknown type %T!\n", v)
>     }
> }
>
```

在这里，实现了一个名为 `tellInterface()` 的函数，该函数有一个类型为 `interface` 的参数 `x` 。

这个函数实现体现了区分不同数据类型的 `x` 参数。通过使用返回 `x` 元素类型的 `x.(type)` 语句来实现这样的效果。在 `fmt.Printf()` 函数中使用的格式化字符串 `%v` 来获取类型的值。

```
>```go
> func main() {
>     x := circle{R: 10}
>     tellInterface(x)
>     y := rectangle{X: 4, Y: 1}
>     tellInterface(y)
>     z := square{X: 4}
>     tellInterface(z)
>     tellInterface(10)
> }
>
```

执行 `switch.go` 将生成以下的输出：

```
$ go run switch.go
{10} is a circle!
This is a rectangle!
This is a square!
Unknown type int!
```

反射

反射是一种高级的Go语言特性，它允许动态地获取任意对象的类型及其结构信息。Go语言通过 `reflect` 包提供用于处理反射的能力。不过应该记住的是，您很可能不需要在每个Go程序中使用反射。

可能会出现的前两个问题是，为什么反射是必要的，什么时候应该使用它？

对于诸如 `fmt` 、 `text/template` 和 `html/template` 等包的实现，反射是必需的。在 `fmt` 包中，反射可以避免必须显式地处理现有的每种数据类型。即使您有耐心编写代码来处理您知道的每种数据类型，您仍然无法处理所有可能的类型！所以在这种情况下，反射使 `fmt` 包的方法能够同新类型协同工作！

因此，当想编写尽可能通用的代码时，或当希望确保能够处理编写时不存在的数据类型代码，都可以使用反射。此外，反射在处理不需要实现公共接口的类型变量的值时，也非常方便。

> 反射帮助您处理未知类型和未知类型值。

`reflect` 包的核心有两种类型，分别为 `reflect.Value` 和 `reflect.Type` 。前一种类型用于存储任何类型的值，而后一种类型用于表示任意Go类型。

使用反射的简单示例

本节将介绍一个相对简单的反射示例，以便开始熟悉这个高级的Go语言功能。

示例的Go程序的名称是 `reflection.go` ，将分四部分介绍。 `reflection.go` 的目的是检查“未知类型”的结构变量，并在运行时了解更多有关它的信息。为了加深理解，程序将定义两个新的 `struct` 类型。基于这两种类型，它还将定义两个新变量；但是，示例程序只检查其中一个。如果程序没有命令行参数，它将检查第一个参数；否则，它将探索第二个。实际上，这意味着程序不会预先知道它将要处理的 `struct` 变量的类型。

```
>
```

> package main
>
> import ("fmt" "os" "reflect")
>
> type a struct { X int Y float64 Z string }
>
> type b struct { F int G int H string I float64 } ```

在程序的这一部分中，您可以看到将要使用的 `struct` 数据类型的定义。

```
>
```

> func main() { x := 100 xRefl := reflect.ValueOf(&x).Elem() xType := xRefl.Type() fmt.Printf("The type of x is %s.\n", xType) ```

上面的Go代码提供了一个小的、简单的反射示例。首先声明一个名为 `x` 的变量，然后调用 `reflect.ValueOf(&x).Elem()` 函数。接下来调用 `xRefl.Type()` 以获取存储在 `xType` 中的变量类型。这三行代码说明了如何使用反射获取变量的数据类型。如果只关心变量的数据类型，那么可以改为调用 `reflect.TypeOf(x)` 。

```
>
```

```
A := a{100, 200.12, "Struct a"}
B := b{1, 2, "Struct b", -1.2}
var r reflect.Value

arguments := os.Args
if len(arguments) == 1 {
    r = reflect.ValueOf(&A).Elem()
} else {
    r = reflect.ValueOf(&B).Elem()
}
```

```

在本部分中，声明了两个名为 `A` 和 `B` 的变量。 `A` 变量的类型为 `a` ， `B` 变量的类型为 `b` 。 `r` 变量的类型定义为 `reflect.Value` ，因为这是函数 `reflect.ValueOf()` 返回的结果。 `Elem()` 方法返回反射接口（ `reflect.Value` ）中包含的值。

```
>
```

```
iType := r.Type()
fmt.Printf("i Type: %s\n", iType)
fmt.Printf("The %d fields of %s are:\n", r.NumField(), iTyp

for i := 0; i < r.NumField(); i++ {
    fmt.Printf("Field name: %s ", iType.Field(i).Name)
    fmt.Printf("with type: %s ", r.Field(i).Type())
    fmt.Printf("and value %v\n", r.Field(i).Interface())
}
```

}```

在程序的这一部分中，使用了 `reflect` 包中适当的功能来获取所需的信息。 `NumField()` 方法返回 `reflect.Value` 结构中的字段个数，而 `Field()` 函数返回结构中的指定字段。 `Interface()` 函数的作用是以接口类型，返回 `reflect.Value` 结构字段的值。

两次执行 `reflection.go` 将得到以下输出：
```

```
$ go run reflection.go 1
The type of x is int.
i Type: main.b
The 4 fields of main.b are:
Field name: F with type: int and value 1
Field name: G with type: int and value 2
Field name: H with type: string and value Struct b
Field name: I with type: float64 and value -1.2
$ go run reflection.go
The type of x is int.
i Type: main.a
The 3 fields of main.a are:
Field name: X with type: int and value 100
Field name: Y with type: float64 and value 200.12
Field name: Z with type: string and value Struct a
```

值得注意的是我们看到Go使用其内部表示来打印变量 `A` 和 `B` 的数据类型，分别是 `main.a` 和 `main.b` 。但是，变量 `x` 不是这样的，它是系统内置的 `int` 类型。

反射进阶

在本节中，我们将探讨反射的更高级的用法，将使用相对较小的 `advRefl.go` 的Go代码进行演示。

```
>
```

```
package main

import ( "fmt" "os" "reflect" )

type t1 int type t2 int ```
```

可以注意到尽管 `t1` 和 `t2` 类型都基于 `int` 类型，因此本质上也与 `int` 类型相同，但Go语言将它们视为完全不同的类型。它们在Go编译器解析后的内部表示分别是 `main.t1` 和 `main.t2` 。

```
>
```

```
type a struct { X int Y float64 Text string }

func (a1 a) compareStruct(a2 a) bool { r1 := reflect.ValueOf(&a1).Elem() r2 := reflect.ValueOf(&a2).Elem()

    for i := 0; i < r1.NumField(); i++ {
        if r1.Field(i).Interface() != r2.Field(i).Interface() {
            return false
        }
    }
    return true

} ```
```

在这个代码段中，我们定义了一个名为 `a` 的Go结构类型，并实现了一个名为 `compareStruct()` 的Go函数。这个函数的目的是找出类型 `a` 的两个变量是否完全相同。如您所见， `compareStruct()` 使用 `reflection.go` 中的Go代码来执行其任务。

```
>
```

> func printMethods(i interface{}) { r := reflect.ValueOf(i) t := r.Type() fmt.Printf("Type to examine: %s\n", t)
>
> ```
> for j := 0; j < r.NumMethod(); j++ {
>     m := r.Method(j).Type()
>     fmt.Println(t.Method(j).Name, "-->", m)
> }
> ```
>
> } ```

```advRefl.go```的第四段包含以下Go代码：

>

> func main() { x1 := t1(100) x2 := t2(100) fmt.Printf("The type of x1 is %s\n", reflect.TypeOf(x1)) fmt.Printf("The type of x2 is %s\n", reflect.TypeOf(x2))
>
> ```
> var p struct{}
> r := reflect.New(reflect.ValueOf(&p).Type()).Elem()
> fmt.Printf("The type of r is %s\n", reflect.TypeOf(r))
> ```
>
> ```

>

> ```
> a1 := a{1, 2.1, "A1"}
> a2 := a{1, -2, "A2"}
>
> if a1.compareStruct(a1) {
>   fmt.Println("Equal!")
> }
>
> if !a1.compareStruct(a2) {
>   fmt.Println("Not Equal!")
> }
>
> var f *os.File
> printMethods(f)
> ```
>
> } ```
```

正如您稍后将看到的，`a1.compareStruct(a1)` 调用返回 `true`，因为我们正在比较 `a1` 与自身，而 `a1.compareStruct(a2)` 调用将返回 `false`，因为 `a1` 和 `a2` 变量的值不同。

执行 `advRefl.go` 将得到以下输出：

```
$ go run advRefl.go
The type of x1 is main.t1
The type of x2 is main.t2
The type of r is reflect.Value
Equal!
Not Equal!
Type to examine: *os.File
Chdir --> func() error
Chmod --> func(os.FileMode) error
Chown --> func(int, int) error
Close --> func() error
Fd --> func() uintptr
Name --> func() string
Read --> func([]uint8) (int, error)
ReadAt --> func([]uint8, int64) (int, error)
Readdir --> func(int) ([]os.FileInfo, error)
Readdirnames --> func(int) ([]string, error)
Seek --> func(int64, int) (int64, error)
Stat --> func() (os.FileInfo, error)
Sync --> func() error
Truncate --> func(int64) error
Write --> func([]uint8) (int, error)
WriteAt --> func([]uint8, int64) (int, error)
WriteString --> func(string) (int, error)
```

可以看到由 `reflect.New()` 返回的 `r` 变量的类型是 `reflect.Value`。另外，`printMethods()` 方法的输出可以看到 `*os.File` 类型支持很多的方法，例如：`Chdir()`、`Chmod()` 等。

反射的三个缺点

毫无疑问，反射是Go语言一个强大的功能。然而，由于所有工具都使用反射，因此应谨慎使用，主要有三个原因。

第一个原因是反射的广泛使用会使程序难以阅读和维护。这个问题的一个潜在解决方案是编写好的文档；但众所周知的是开发人员并不愿意花时间来编写维护文档。

第二个原因是使用反射的Go代码会使程序运行变慢。一般来说，用于特定数据类型的Go代码总是比使用反射动态处理Go数据类型的代码运行速度快。此外，这样的动态代码将使工具很难重构或分析代码。

最后一个原因是反射错误在编译时无法捕获，并且运行时会造成 `panic` 异常。这意味着反射的错误可能会使程序崩溃！这可能发生在Go程序开发完成后的数月甚至数年后！这个问题的一个解决办法是在危险函数调用之前进行全面的测试。然而，这将为Go程序增加更多的代码，使程序运行的更慢。

Go的OOP思想

现在应该了解Go语言缺少继承，但它支持组合，并且Go接口提供了一种多态性。所以，尽管Go不是面向对象的编程语言，但它有一些特性可以让您模拟面向对象的编程。

> 如果您真的想使用面向对象的方法开发应用程序，那么Go语言可能不是您的最佳选择。由于我不是很喜欢Java，我建议看一下C++或Python。当然，Go禁止构建复杂而深入的类型层次结构，也是因为这些深层次结构很难使用和维护！

首先，让我解释这一部分中将使用的两种Go语言技术。第一种技术使用方法将函数与类型关联起来。这意味着，在某种程度上，您可以考虑函数和数据类型构造了一个对象。

使用第二种技术，可以将数据类型嵌入到新的结构类型中，以便创建一种层次结构。

还有第三种技术，即使用Go接口生成两个或多个元素——一个类的多个对象。因为本章前面的章节刚刚进行了论述，本节将不介绍这种技术。这个技术的关键点是Go接口允许您在不同的元素之间定义一个共同的行为（函数方法），这样所有这些不同的元素都可以共享一个对象的特性，因此可以认为这些不同的元素是同一个类的对象。然而，实际的面向对象编程语言的对象和类可以做更多的事情。

前两种技术将在 `ooo.go` 程序中进行说明，该程序将分四部分介绍。 `ooo.go` 的第一段包含以下Go代码：

```
package main

import (
    "fmt"
)

type a struct {
    XX int
    YY int
}

type b struct {
    AA string
    XX int
}
```

```
>```go
> type c struct {
>     A a
>     B b
> }
>
```

在Go语言中，组合技术允许使用多个 `struct` 类型创建一个新的结构。在以上代码示例中，数据类型 `c` 由一个结构体 `a` 的变量和一个结构体 `b` 的变量组合而成。

以下Go代码是 `ooo.go` 的第三部分：

```
func (A a) A() {
    fmt.Println("Function A() for A")
}

func (B b) A() {
    fmt.Println("Function A() for B")
}
```

此处定义的两个方法可以使用相同的名称（ `A()` ），因为它们具有不同的函数头。第一种方法使用 `a` 变量，而第二种方法使用 `b` 变量。此技术允许您在多个类型之间共享相同的函数名。

代码 `ooo.go` 的最后一部分如下：

```
func main() {
    var i c
    i.A.A()
    i.B.A()
}
```

与实现**抽象类**和**继承**的面向对象编程语言代码相比， `ooo.go` 中的所有Go代码都非常简单。而且，它对于生成具有结构的类型和元素，以及具有相同方法名的不同数据类型来说是足够的。

执行 `ooo.go` 程序的输出如下：

```
$ go run ooo.go
Function A() for A
Function A() for B
```

以下代码则说明了组合与继承的差异，结构体 `first` 对结构体 `second` 调用 `shared()` 函数，对其所做的更改一无所知：

```
package main

import (
    "fmt"
)

type first struct{}

func (a first) F() {
    a.shared()
}

func (a first) shared() {
    fmt.Println("This is shared() from first!")
}

type second struct {
    first
}

func (a second) shared() {
    fmt.Println("This is shared() from second!")
}

func main() {
    first{}.F()
    second{}.shared()
    i := second{}
    j := i.first
    j.F()
}
```

通过以上代码可以看到结构体 `second` 嵌入了结构体 `first` ，这两个结构体共享一个名为 `shared()` 的函数。

以上Go代码保存为goCoIn.go并执行，将生成以下输出：

```
$ go run goCoIn.go
This is shared() from first!
This is shared() from second!
This is shared() from first!
```

`first{}.F()` 和 `second{}.shared()` 的函数调用生成了预期的结果。尽管结构体 `second` 重新实现了 `shared()` 函数，对 `j.F()` 的调用仍然调用了 `first.shared()` 而不是 `second.shared()` 函数。在面向对象语言的术语中这样的操作叫做 `方法重写` 。

另外可以将 `j.F()` 调用可以简写成 `(i.first).F()` 或 `(second{}.first).F()` ，而不需要定义太多的变量。为了更容易理解，上面的调用将这个过程分成三行代码。

延展阅读

您会发现以下两项资源非常方便：

- 访问Go标准包文档中 `reflect` 包的页面，可在 https://golang.org/pkg/reflect/中找到。 `relfect` 包有比本章介绍更多的功能。
- 如果你真的对反射很感兴趣并想了解更多关于反射的知识，你可以参考Mitchell Hashimoto的 `reflectwalk` 库，网址为 https://github.com/mitchell/reflectwalk。 `reflectwalk` 库允许您在Go语言中使用反射遍历复杂值。如果有时间，可以研究下这个库的Go代码！

练习

- 编写一个接口，并在另一个Go程序中使用它。然后说明你编写的接口的作用。
- 编写一个接口，计算三维图形的体积，例如立方体和球体。
- 编写一个借口，计算线段长度和平面上两点之间的距离。
- 编写一个使用反射的例子。
- 反射如何在Go映射上工作？
- 如果你擅长数学，试着编写一个接口来实现实数和复数的四个基本数学运算。不要使用 `complex64` 和 `complex128` 这些标准Go类型，用来定义支持复数的结构。

本章小结

在本章中，您学习了类似契约的接口，以及Go中的类型方法、类型断言和反射。您还学习了如何遵循面向对象编程原则编写Go代码。

虽然反射是一个非常强大的Go特性，但它可能会减慢Go程序的速度，因为它在运行时增加了一层复杂性。此外，如果使用反射时不够仔细，Go程序可能会崩溃。

如果要只记住本章中唯一一件事，那就是Go不是面向对象的编程语言，而是可以模仿面向对象编程语言提供的一些功能，例如Java和C++。这意味着，如果计划使用面向对象的编程范式开发软件，那么最好选择Go以外的面向对象的编程语言。当然，面向对象编程并不是万能的，如果您选择像Go语言这样的编程语言，那么一样会创建一个更好、更干净、更健壮的设计！

虽然这一章的理论可能比你预期的要多，但下一章会让你的耐心得到回报，因为它将涉及Go语言中的系统编程，将要讨论文件I/O、Unix系统文件的使用、Unix信号的处理以及Unix管道的支持。另外，下一章将讨论如何使用 `flag` 包支持命令行工具中的多个命令行参数和选项、遍历目录结构、Unix文件权限以及Go标准库的 `syscall` 包提供的一些高级用法。

如果你真的喜欢用Go进行系统编程，那么阅读我的《Go Systems Programming，Packt Publishing，2017》这本书会是个好主意。

Go UNIX系统编程

上一章，我们讨论了两个高级的，但有点偏理论的 Go 主题：接口和反射。这章的代码只是理论性的。

因为这章的主题是**系统编程**，毕竟 Go 是一门严格的系统编程语言；它的创造者不满意于他们编写系统软件所选择的语言，所以它们决定创造一门新的编程语言。

> *如果您真的对用 Go做 **系统编程**感兴趣，那么 2017 出版的《Go 系统编程》绝对可以帮助到您。但是，这章包含到一些有趣并有点高级的主题没有出现在我的《Go 系统编程》这本书中*

《玩转 Go》的这章，您会了解到如下主题：

- Unix 进程
- `flag` 包
- `io.Reader` 和 `io.Writer` 接口的使用
- 用 Go 的 `os/signal` 包处理 Unix 信号
- 在您的 Unix 系统工具中支持 Unix 管道
- 读取文本文件
- 读取 CSV 文件
- 写入文件
- `bytes` 包
- `syscall` 包的进阶使用
- 遍历目录结构
- Unix 文件权限

关于UNIX进程

严格讲，**进程**是一个执行环境，包含指令，用户数据，系统数据和运行中获取的资源。另外，**程序**是一个二进制文件，包含指令和初始化指令时用到的数据，还有进程的用户数据。每次运行进程都被一个无符号整数唯一标识，它叫做**进程 ID**。

进程有三种分类：用户进程，守护进程，内核进程。**用户进程**运行在用户空间并且通常没有特殊访问权限。**内存进程**仅在内存空间执行并且可以完全访问所有内存数据结构。**守护进程**是运行在用户空间的程序并且运行在后台不需要一个终端。

> *C 调用 `fork()` 创建新进程。 `fork()` 的返回值用于程序员在父进程和子进程之间进行分辨。相反，Go 不支持类似功能而是提供 goroutines*

flag包

标识是特殊格式的字符串，可以传入程序以控制其行为。如果希望支持多个标识，那么自己单独处理标识可能会很困难。因此，如果你正在开发Unix系统命令行实用程序，你会发现flag包非常有趣和有用。在它的其他特性中，flag包对命令行参数和选项的顺序没有任何假设，并且在命令行实用程序执行过程中出现错误时，它会打印有用的信息。

flag包的最大好处是，它是*Go*标准库的一部分，意味着它经过了大量的测试和调试。

我将介绍两个使用flag包的Go程序：一个简单的和一个复杂的。

第一个程序，称为 `simpleFlag.go` ，分为四部分。 `simpleFlag.go` 程序识别两个命令行选项：第一个是 `Boolean` 选项，第二个是整数值。

`simpleFlag.go` 的第一部分如下：

```
import (
    "flag"
    "fmt"
)
```

`simpleFlag.go` 的第二部分如下：

```
func main() {
    minusK := flag.Bool("k", true, "k")
    minusO := flag.Int("O", 1, "O")
    flag.Parse()
```

`flag.Bool("k", true, "k")` 语句定义了名为 `k` 的 `Boolean` 型的命令行选项，默认值为 `true` 。语句的最后一个参数是将与程序的用法信息一起显示的字符串。类似， `flag.Int()` 函数的作用是增加对整数命令行选项的支持。

通常，在定义了命令行选项后，需要调用 `flag.Parse()` 。

`simpleFlag.go` 的第三部分程序如下：

```
  valueK := *minusK
    valueO := *minusO
    valueO++
```

在前面的 `Go` 代码中，你看到了如何获取选项的值。好处是 `flag` 自动将 `flag.Int()` 标识关联的输入转换为整数值。即你不用再做转换操作。另外， `flag` 包确保它得到一个可接受的整数值。

`simpleFlag.go` 的剩余代码如下：

```
  fmt.Println("-k:", valueK)
    fmt.Println("-O:", valueO)
}
```

获取到需要的参数后，你就可以使用它们了。

与 `simpleFlag.go` 交互将创建如下输出：

```
$ go run simpleFlag.go -O 100
-k: true
-O: 101
$ go run simpleFlag.go -O=100
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k false
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k=false
-k: false
-O: 101
```

如果在执行 `simpleFlag.go` 中遇到错误，则会得到如下错误信息：

```
$ go run simpleFlag.go -O=notAnInteger
invalid value "notAnInteger" for flag -O: strconv.ParseInt: pars
Usage of /var/folders/sk/ltk8cnw501zdtr2hxcj5sv2m0000gn/T/go-bui
  -O int
        O (default 1)
  -k    k (default true)
exit status 2
```

注意：当程序的命令行选项中出现错误时，将自动打印常规用法信息。

现在该是展示一个使用 `flag` 包更实际、更高级的程序了。程序 `funWithFlag.go` 分为五部分。 `funWithFlag.go` 识别各种选项，包括一个由逗号分隔的多个值的选项。此外，它还将说明如何访问位于可执行文件末尾且不属于任何选项的命令行参数。

`funWithFlag.go` 中使用的 `flag.Var()` 函数创建一个满足 `flag.Value` 接口的任意类型的标识，`flag.Value` 接口定义如下：

```
type Value interface {
    String() string
    Set(string) error
}
```

`funWithFlag.go` 第一部分代码如下：

```
package main

import(
    "flag"
    "fmt"
    "strings"
)

type NamesFlag struct {
    Names []string
}
```

`NamesFlag` 数据结构用于 `flag.Value` 接口。

`funWithFlag.go` 的第二部分如下：

```
func (s *NamesFlag) GetNames() []string{
    return s.Names
}
func (s *NamesFlag) String() string {
    return fmt.Sprint(s.Names)
}
```

`funWithFlag.go` 的第三部分代码如下：

```
func (s *NamesFlag) Set(v string) error {
    if len(s.Names) > 0 {
        return fmt.Errorf("Cannot use names flag more than once!
    }

    names := strings.Split(v, ",")
    for _, item := range names {
        s.Names = append(s.Names, item)
    }
    return nil
}
```

首先，`Set()` 方法确保相关命令行选项没有被设置。之后，获取输入并使用 `strings.Split()` 函数来分隔参数。最后，参数被保存在 `NamesFlag` 结构的 `Names` 字段。

`funWithFlag.go` 的第四部分代码如下：

```
func main() {
    var manyNames NamesFlag
    minusK := flag.Int("k:", 0, "An int")
    minusO := flag.String("o", "Mihalis", "The name")
    flag.Var(&manyNames, "names", "Comma-separated list")

    flag.Parse()
    fmt.Println("-k:", *minusK)
    fmt.Println("-o:", *minusO)
```

`funWithFlag.go` 的最后部分如下：

```
    for i, item := range manyNames.GetNames() {
        fmt.Println(i, item)
    }
    fmt.Println("Remaing command-line arugments:")
    for index, val := range flag.Args() {
        fmt.Println(index, ":", val)
    }
}
```

`flag.Args()` 切片保留命令行参数，而 `manyNames` 变量保留来自 `flag.Var()` 命令行选项的值。

执行 `funWithFlag.go` 产生如下输出：

> *除非你开发一个不需要选项的命令行小工具，否则你极需要使用 `flag` 包来处理您的程序的命令行选项。*

io.Reader和io.Writer接口

如前所述，实现 `io.Reader` 接口需要实现 `Read()` 方法， `io.Writer()` 接口需要实现 `Write()` 方法。这两个接口在 `Go` 中非常常见，我们将在稍后使用它们。

缓冲和无缓冲的文件输入和输出

缓冲文件输入和输出发生在读取或写入数据之前有一个临时存储数据的缓冲区。因此，你可以一次读取多个字节，而不是逐字节地读取一个文件。你将它放在一个缓冲区中，等待调用者以所需的方式读取它。非缓冲文件输入和输出发生在实际读取或写入数据之前没有临时存储数据的缓冲区时。

你可能会问的下一个问题是，如何决定何时使用缓冲文件输入和输出，以及何时使用非缓冲文件输入和输出。在处理关键数据时，非缓冲文件输入和输出通常是更好的选择，因为缓冲读取可能导致过期的数据，而缓冲写入可能在计算机电源中断时导致数据丢失。然而，大多数时候，这个问题没有明确的答案。这意味着你可以使用任何使任务更容易实现的工具。

bufio包

顾名思义，`bufio`包是关于缓冲输入输出的。但是，`bufio`包在内部仍然使用`io.Reader`和`io.Writer`对象，它封装这些对象分别创建`bufio.Reader`和`bufio.Writer`对象。

下一节中，我们会看到，`bufio`包非常适合读取文本文件。

读取文本文件

文本文件是 `Unix` 系统中最常见的一种文件。在本节中，你将学习如何以三种方式读取文本文件：逐行读取、逐词读取、逐字符读取。如你所见，逐行读取是访问文本文件的最简单方法，而逐词读取是最困难的方法。

如果你仔细观察 `byLine.go` 、 `byWord.go` 、 `byCharacter.go` 程序，你会发现它们的 `Go` 代码有许多相似之处。首先，它们都是逐行读取输入文件。其次，除了 `main()` 函数的 `for` 循环中调用的函数不同之外，这三个实用程序都具有相同的 `main()` 函数。最后，除了函数的实现部分不同之外，处理输入文本文件的三个函数几乎是相同的。

逐行读取文本文件

逐行读取文本文件是最常用的方式。这也是我们首先介绍它的原因。 `byLine.go` 程序分为三部分，将帮助你理解这个技巧。

`byLine.go` 第一部分代码如下：

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

引入 `bufio` 包表示我们将使用缓冲区输入。

```
func lineByLine(file string) error {
    var err error

    f, err := os.Open(file)
    if err != nil {

        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            break
        }
        fmt.Printf(line)
    }
    return nil
}
```

所有的实现都在 `lineByLine()` 函数中。在确保可以打开指定的文件名进行读取之后，你调用 `bufio.NewReader()` 创建一个新的读实例，然后你可以调用 `bufio.ReadString()` 逐行读取文件。行分隔符通过 `bufio.ReadString()` 参数指定，它指示 `bufio.ReadString()` 一直读取，直到碰到行分隔符为止。当参数是换行符时，不断调用 `bufio.ReadString()` 会逐行读取输入文件！注意，使用 `fmt.Print()` 而不是 `fmt.Println()` 输出读取行，来显示每个输入行中都包含换行符。

`byLine.go` 第三部分代码如下：

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byLine <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := lineByLine(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

执行 `byLine.go`，并使用 `wc(1)` 处理输出会产生如下的输出内容：

```
$ go run byLine.go /tmp/swtag.log /tmp/adobegc.log | wc
        4761   88521   568402
```

如下的命令会校验前述输出的精确性：

```
$ wc /tmp/swtag.log /tmp/adobegc.log
         131     693     8440     /tmp/swtag.log
        4630   87828   559962     /tmp/adobegc.log
        4761   88521   568402     total
```

逐词读取文本文件

本节中展示的技术将通过 `byWord.go` 文件演示，它由四部分组成。正如你在 `Go` 代码中看到的，分隔一行中的单词可能比较棘手。程序的第一部分如下：

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
    "regexp"
)
```

`byWord.go` 的第二部分代码如下：

```
func wordByWord(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
```

`wordByWord()` 函数的这部分代码和 `byLine.go` 程序的 `lineByLine()` 函数一样。

`byWord.go` 第三部分代码如下：

```
        r := regexp.MustCompile("[^\\s]+")
          words := r.FindAllString(line, -1)
          for i := 0; i < len(words); i++ {
              fmt.Printf(words[i])
          }
    }
    return nil
}
```

`wordByWord()` 函数的剩余代码是全新的，并使用正则表达式对输入的每行进行单词分割。正则表达式 `regexp.MustCompile("[^\\s]+")` 使用空格分割单词。

`byWord.go` 的最后一部分代码如下：

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byWord <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := wordByWord(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

执行 `byWord.go` 会产生如下的输出：

```
$ go run byWord.go /tmp/adobegc.log
01/08/18
20:25:09:669
|
[INFO]
```

可以使用 `wc(1)` 验证 `byWord.go` 的正确性：

```
$ go run byWord.go /tmp/adobegc.log | wc
    91591     91591     559005
$ wc /tmp/adobegc.log
    4831      91591     583454     /tmp/adobegc.log
```

如你所见， `wc(1)` 计算所得的单词数和 `byWord.go` 一致。

逐字符读取文本文件

在本节中，你将学会如何逐字符读取文本文件，这是一个非常少见的需求，除非你想创建文本编辑器。相关的 `Go` 代码参考 `byCharacter.go`，将分为四部分介绍。

`byCharacter.go` 第一部分代码如下：

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

如你所见，本程序不需要使用正则表达式。

`byCharacter.go` 第二部分代码如下：

```
func charByChar(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
```

`byCharacter.go` 第三部分代码如下：

```
        for _, x := range line {
            fmt.Println(string(x))
        }
    }
    return nil
}
```

此处，你读取每一行，并使用 `range` 分割。 `range` 返回两个值：丢弃第一个值，它是 `line` 变量中当前字符的位置，并使用第二个值。但是，该值不是字符-这就是为什么必须使用 `string()` 函数将其转换为字符的原因。

注意，由于 `fmt.Println(string(x))` 语句，每个字符都按行打印，这意味着程序的输出将很大。如果想要压缩输出，应该使用 `fmt.Print()` 函数。

`byCharacter.go` 的最后一部分代码如下：

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byChar <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := charByChar(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

执行 `byCharacter.go` 会产生如下输出：

```
$ go run byCharacter.go /tmp/adobegc.log
0
1
/
0
8
/
1
8
```

注意，上述 `Go` 代码也能用来统计输入文件的字符个数，即是 `Go` 版本的 `wc(1)` 命令行实现。

从/dev/random中读取

在本节中，你将学习如何从 `/dev/random` 系统设备读取数据。 `/dev/random` 系统设备的目的是生成随机数据，你可以使用这些数据来测试程序，或者，在本例中，你将为随机数生成器生成随机数种子。从 `/dev/random` 获取数据可能有些棘手，这也是要在这里特别讨论它的主要原因。

在 `macOS High Sierra` 机器上， `/dev/random` 文件具有如下的权限：

```
$ ls -l /dev/random
crw-rw-rw-  1 root  wheel   14,   0  Jan 8 20:24 /dev/random
```

同样，在 `Debian Linux` 机器上， `/dev/random` 系统设备具有如下的 `Unix` 文件权限：

```
$ ls -l /dev/random
crw-rw-rw-  1 root  root   1,   8  Jan 13 12:19 /dev/random
```

这意味着 `/dev/random` 文件在这两种 `Unix` 变体系统上具有类似的文件权限。惟一区别是它们的文件组权限不同， `macOS` 是 `wheel` ， `Debian Linux` 上是 `root` 。

本节主题的程序是 `devRandom.go` ，分为三部分。第一部分代码如下：

```
package main

import (
    "encoding/binary"
    "fmt"
    "os"
)
```

为了从 `dev/random` 中读取数据，需要引入 `encoding/binary` 标准包，因为 `/dev/random` 返回二进制数据，需要解码。

`devRandom.go` 第二部分代码如下：

```
func main() {
    f, err := os.Open("/dev/random")
    defer f.Close()

    if err != nil {
        fmt.Println(err)
        return
    }
```

和以往一样，打开 `/dev/random` ，因为Unix中一切皆是文件。

`devRandom.go` 最后一部分代码如下：

```
    var seed int64
    binary.Read(f, binary.LittleEndian, &seed)
    fmt.Println("Seed:", seed)
}
```

调用 `binary.Read()` 函数从 `/dev/random` 系统设备中读取数据，`binary.Read()` 函数需要三个参数：第二个参数(`binary.LittleEndian`)指定了小端字节序，另一个选项是 `binary.BigEndian` ，在计算机使用大端字节序时使用。

执行 `devRandom.go` 得到如下输出：

```
$ go run devRandom.go
Seed: -2044736418491485077
$ go run devRandom.go
Seed: -517485437251490328
$ go run devRandom.go
Seed: 7702177874251412774
```

从文件中读取所需的数据量

在本节中，你将学习如何准确读取所需的数据量。这种技术在读取二进制文件时特别有用，在二进制文件中，必须以特定的方式解码读取的数据。不过，这种技术仍然适用于文本文件。

这种技术背后的逻辑并不难：创建一个所需大小的字节切片，并使用该字节切片进行读取。为了更有趣一点，我们将使用一个函数实现这个功能，这个函数具有两个参数。一个参数用于指定需要读取的数据量，另一个参数将使用 `*os.File` 文件类型，用于访问所需的文件。该函数的返回值将是所读取的数据。

本节的实现文件是 `readSize.go`，分为四部分。程序接受一个简单的参数，为字节切片的大小。

当使用当前技术时，这个特定的程序可以帮助你使用所需的缓冲区大小复制任何文件。

`readSize.go` 的第一部分代码如下：

```
package main

import (
    "fmt"
    "io"
    "os"
    "strconv"
)
```

`readSize.go` 的第二部分代码如下：

```
func readSize(f *os.File, size int) []byte {
    buffer := make([]byte, size)

    n, err := f.Read(buffer)
    if err == io.EOF {
        return nil
    }

    if err != nil {
        fmt.Println(err)
        return nil
    }

    return buffer[0:n]
}
```

这即是前面讨论的函数。尽管代码相当直接，但仍有一点需要解释。`io.Reader.Read()`方法返回两个参数：读取的字节数以及`error`变量。

`readSize()`函数的作用是：使用`io.Read()`的第一个返回值返回字节切片大小。虽然这是一个很小的细节，而且只有在到达文件末尾时才重要，但是它确保实用程序的输出与输入相同，并且不包含任何额外的字符。最后，还要检查`io.EOF`,表示已经到达文件的末尾。当发生错误时，函数返回。

代码的第三部分如下：

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("<buffer size> <filename>")
        return
    }

    bufferSize, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    file := os.Args[2]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

`readSize.go` 的最后一部分代码如下：

```
    for {
        readData := readSize(f, bufferSize)
        if readData != nil {
            fmt.Println(string(readData))
        } else {
            break
        }
    }
}
```

所以，你需要一直读取输入文件，直接返回错误或者 `nil` 。

执行 `readSize.go` ，传入处理的二进制文件，并使用 `wc(1)` 处理它的输出，来验证程序的正确性。

```
$ go run readSize.go 1000 /bin/ls | wc
     80     1032    38688
$ wc /bin/ls
     80     1032    38688    /bin/ls
```

为什么我们使用二进制格式

在前一节中，`readSize.go`程序演示了如何逐字节读取文件，这是一种应用于二进制文件的技术。那么，你可能会问，既然文本格式更容易理解，为什么要读取二进制格式的数据呢？主要原因是节省空间。假设你想将数字`20`作为字符串存储到文件中。很容易理解，使用`ASCII`字符存储`20`需要两个字节，一个用于存储`2`，另一个用于存储`0`。以二进制格式存储`20`只需要一个字节，因为`20`可以用二进制表示为`00010100`，也可以用十六进制表示为`0x14`。

在处理少量数据时，这种差异可能看起来微不足道，但在处理应用程序（如数据库服务器）中的数据时，这种差异可能非常显著。

读取CSV文件

`CSV` 文件是纯文本文件。在本节中，你将学习如何读取包含平面点的文本文件，这意味着每一行将包含一对坐标。此外，你还将使用一个名为 `Glot` 的外部 `Go` 库，它将帮助你创建从 `CSV` 文件中读取的点的图表。注意， `Glot` 使用 `Gnuplot` ，这意味着你需要在 `Unix` 机器上安装 `Gnuplot` 才能使用 `Glot` 。

本章的程序是 `CSVplot.go` ，分为五部分。第一部分代码如下：

```
package main

import (
    "encoding/csv"
    "fmt"
    "github.com/Arafatk/glot"
    "os"
    "strconv"
)
```

`CSVplot.go` 第二部分代码如下：

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a data file!")
        return
    }

    file := os.Args[1]
    _, err := os.Stat(file)
    if err != nil {
        fmt.Println("Cannot stat", file)
        return
    }
```

在本部分中，你将看到一种使用强大的 `os.Stat()` 函数检查文件是否已经存在的技术。

`CSVplot.go` 第三部分代码如下：

```
    f, err := os.Open(file)
    if err != nil {
        fmt.Println("Cannot open", file)
        fmt.Println(err)
        return
    }
    defer f.Close()

    reader := csv.NewReader(f)
    reader.FieldsPerRecord = -1
    allRecords, err := reader.ReadAll()
    if err != nil {
        fmt.Println(err)
        return
    }
```

`CSVplot.go` 第四部分代码如下：

```
    xP := []float64{}
    yP := []float64{}

    for _, rec := range allRecords {
        x, _ := strconv.ParseFloat(rec[0], 64)
        y, _ := strconv.ParseFloat(rec[1], 64)
        xP = append(xP, x)
        yP = append(yP, y)
    }


    points := [][]float64{}
    points = append(points, xP)
    points = append(points, yP)
    fmt.Println(points)
```

此处，你将字符串转为数字，并追加到二维切片 `points` 中。

`CSVplot.go` 最后一部分代码如下：

```
    dimensions := 2
    persist := true
    debug := false
    plot, _ := glot.NewPlot(dimensions, persist, debug)

    plot.SetTitle("Using Glot with CSV data")
    plot.SetXLabel("X-Axis")
    plot.SetYLabel("Y-Axis")
    style := "circle"
    plot.AddPointGroup("Circle:", style, points)
    plot.SavePlot("output.png")
}
```

在前面的 `go` 代码中，你了解了如何使用 `Glot` 库及其 `Glot.SavePlot()` 函数创建 `PNG` 文件。

可以猜到，在编译和执行 `CSVplot.go` 之前，需要下载 `Glot Go` 库，它需要从Unix shell执行以下命令：

```
$ go get github.com/Arafatk/glot
```

你可以通过查看 `~/go` 目录查看当前命令的执行结果:

```
$ ls -l ~/go/pkg/darwin_amd64/github.com/Arafatk/
total 240
-rw-r--r-- 1 mtsouk staff 119750 Jan 22 22:12 glot.a
$ ls -l ~/go/src/github.com/Arafatk/glot/
total 120
-rw-r--r--  1 mtsouk  staff   1818   Jan  22 22:12 README.md
-rw-r--r--  1 mtsouk  staff   6092   Jan  22 22:12 common.go
-rw-r--r--  1 mtsouk  staff    552   Jan  22 22:12 common_test.g
-rw-r--r--  1 mtsouk  staff   3162   Jan  22 22:12 core.go
-rw-r--r--  1 mtsouk  staff    138   Jan  22 22:12 core_test.go
-rw-r--r--  1 mtsouk  staff   3049   Jan  22 22:12 function.go
-rw-r--r--  1 mtsouk  staff    511   Jan  22 22:12 function_test
-rw-r--r--  1 mtsouk  staff   4955   Jan  22 22:12 glot.go
-rw-r--r--  1 mtsouk  staff    220   Jan  22 22:12 glot_test.go
-rw-r--r--  1 mtsouk  staff  10536   Jan  22 22:12 pointgroup.go
-rw-r--r--  1 mtsouk  staff    378   Jan  22 22:12 pointgroup_te
```

包含要绘制的点的 `CSV` 数据文件具有以下格式：

```
$ cat /tmp/dataFile
1,2
2,3
3,3
4,4
5,8
6,5
-1,12
-2,10
-3,10
-4,10
```

执行 `CSVplot.go` 会产生如下的输出：

```
$ go run CSVplot.go /tmp/doesNoExits
Cannot stat /tmp/doesNoExits
$ go run CSVplot.gp /tmp/dataFile
[[1 2 3 4 5 6 -1 -2 -3 -4] [2 3 3 4 8 5 12 10 10 10]]
```

你可以用下图查看 `CSVplot.go` 的结果。 

写入文件

一般来说，你可以使用 `io.Writer` 接口将数据写入磁盘上的文件。然而， `save.go` 代码将向你展示五种将数据写入文件的方法。 `save.go` 程序分为六部分。

`save.go` 第一部分代码如下：

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)
```

`save.go` 第二部分代码如下：

```
func main() {
    s := []byte("Data to write\n")


    f1, err := os.Create("f1.txt")
    if err != nil {
        fmt.Println("Cannot create file", err)
        return
    }
    defer f1.Close()
    fmt.Fprintf(f1, string(s))
```

注意，在这个 `Go` 程序中涉及到写入的每一行都将使用 `s` 字节切片。此外， `fmt.Fprintf()` 函数可以帮助你使用所需的格式将数据写入自己的日志文件。在本例中， `fmt.Fprintf()` 将数据写入 `f1` 标识的文件。

`save.go` 第三部分代码如下：

```
    f2, err := os.Create("f2.txt")
    if err != nil {
        fmt.Println("Cannot create file", err)
        return
    }
    defer f2.Close()
    n, err := f2.WriteString(string(s))
    fmt.Printf("wrote %d bytes\n", n)
```

其中，`f2.WriteString()` 用于将数据写入文件。

`save.go` 第四部分代码如下：

```
    f3, err := os.Create("f3.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    w := bufio.NewWriter(f3)
    n, err = w.WriteString(string(s))
    fmt.Printf("wrote %d bytes\n", n)
    w.Flush()
```

其中，`bufio.NewWriter()` 打开文件，并调用 `bufio.WriteString()` 写入数据。

`save.go` 第五部分代码将教你写入文件的其他方法：

```
    f4 := "f4.txt"
    err = ioutil.WriteFile(f4, s, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
```

此方法只需要简单调用 `ioutil.WriteFile()` 函数，而无需使用 `os.Create()` 。

`save.go` 最后一部分代码如下：

```
    f5, err := os.Create("f5.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    n, err = io.WriteString(f5, string(s))
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("wrote %d bytes\n", n)
}
```

该技术使用 `io.WriteString()` 将所需数据写入文件。

执行 `save.go` 将产生如下的输出：

```
$ go run save.go
wrote 14 bytes
wrote 14 bytes
wrote 14 bytes
$ ls -l f?.txt
-rw-r--r--  1 mtsouk  staff  14  Jan  23 20:30 f1.txt
-rw-r--r--  1 mtsouk  staff  14  Jan  23 20:30 f2.txt
-rw-r--r--  1 mtsouk  staff  14  Jan  23 20:30 f3.txt
-rw-r--r--  1 mtsouk  staff  14  Jan  23 20:30 f4.txt
-rw-r--r--  1 mtsouk  staff  14  Jan  23 20:30 f5.txt
$ cat f?.txt
Data to write
Data to write

Data to write
Data to write
Data to write
```

下一节将向你展示如何通过 `Go` 标准库中的专用包功能将数据保存到文件中。

从磁盘加载和保存数据

你还记得第 4 节的 keyValue.go 应用吗？复合类型的使用，它还远远没有完成，所以在本节中，你将学习如何将键值存储的数据保存在磁盘上，以及如何在下一次启动应用程序时将其加载到内存。

我们准备创建两个新函数， save() 保存数据到磁盘， load() 从磁盘加载数据。因此，我们将用 diff(1) Unix 命令行程序显示 keyValue.go 和 kvSaveLoad.go 之间的代码差异。

> *当你希望发现两个文本文件之间的差异时， diff(1) Unix 命令行实用程序非常方便。通过在 Unix shell 命令行上执行 man 1 diff ，你可以了解更多关于它的信息。*

如果你考虑这里要实现的任务，你将认识到你需要的是一种简单的方法来将 Go 映射的内容保存到磁盘，以及一种方法来加载文件中的数据并将其放入 Go 映射。

将数据转换为字节流的过程称为序列化。读取数据文件并将其转换为对象的过程称为反序列化。 encoding/gob 标准 Go 包将用于 kvSaveLoad.go 程序。它将有助于序列化和反序列化过程。 encoding/gob 包使用 gob 格式存储其数据。这种格式的官方名称是流编码。 gob 格式的好处是， Go 完成了所有的繁琐工作，因此你不必担心编码和解码阶段。

其他 Go 包可以帮助你序列化和反序列化数据， encoding/xml 使用 XML 格式， encoding/json 使用 JSON 格式。

下面的输出将显示 kvSaveLoad.go 和 keyValue.go 之间的代码更改，不包括 save() 和 load() 函数的实现，这些函数将在这里完整地展示：

```
$ diff keyValue.go kvSaveLoad.go
4a5
>. "encoding/gob"
16a18,55
> var DATAFILE = "/tmp/dataFile.gob"
> func save() error {
>
>    return nil
> }
>
> func load() error {
>
> }
59a99,104
>
>.   err := load()
>    if err != nil {
>        fmt.Println(err)
>    }
>
88a134,137
>            err = save()
>            if err != nil {
>                 fmt.Println(err)
>            }
```

`diff(1)` 输出的一个重要部分是 `DATAFILE` 全局变量的定义，该变量保存键值存储使用的文件路径。除此之外，还可以看到 `load()` 函数的调用位置以及 `save()` 函数的调用位置。 `load()` 函数首先在 `main()` 函数中使用，而 `save()` 函数在用户发出 `STOP` 命令时执行。

`save()` 函数实现如下：

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}
```

注意，`save()` 函数做的第一件事是使用 `os.Remove()` 函数删除现有数据文件，以便稍后创建它。

`save()` 函数所做的最关键的任务之一是确保你可以实际创建并写入所需的文件。尽管有很多方法可以做到这一点，但是 `save()` 函数是最简单的方法，即检查 `os.Create()` 函数返回的错误值。如果该值不是 `nil`，那么就会出现问题，`save()` 函数在不保存任何数据的情况下结束。

`load()` 函数实现如下：

```
func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}
```

`load()` 函数的任务之一是确保要读取的文件确实存在，并且可以毫无问题地读取它。`load()` 函数再次使用最简单的方法，即查看 `os.Open()` 函数的返回值。如果返回的错误值等于 `nil`，则一切正常。

在读取数据之后关闭文件也很重要，因为稍后 `save()` 函数将覆盖该文件。文件的释放由 `defer loadFrom.Close()` 语句完成。

执行 `kvSaveLoad.go` 会产生如下的输出：

```
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
ADD 1 2 3
ADD 4 5 6
STOP
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 1 value: {2 3 }
key: 3 value: {5 6 }
DELETE 1
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
rMacBook:code mtsouk$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
$ ls -l /tmp/dataFile.gob
-rw-r--r-- 1 mtsouk wheel 80 Jan 22 11:22 /tmp/dataFile.gob
$ file /tmp/dataFile.gob
/tmp/dataFile.gob: data
```

在第13章，网络编程—构建服务器和客户端，你将看到键值存储的最终版本，它将能够在TCP/IP连接上运行，并将使用 `goroutines` 服务于多个网络客户端。

再看strings包

我们在第4章“复合类型的使用”中首先讨论了 `strings` 包。本节将讨论与文件输入和输出相关的 `strings` 包。

`str.go` 第一部分代码如下：

```
package main

import (
    "fmt"
    "io"
    "os"
    "strings"
)
```

`str.go` 的第二段代码如下：

```
func main() {
    r := strings.NewReader("test")
    fmt.Println("r length", r.Len())
```

`strings.NewReader()` 函数从字符串创建只读 `Reader` 。 `strings.Reader` 对象实现了 `io.Reader` 、 `io.ReaderAt` 、 `io.Seeker` 、 `io.WriterTo` 、 `io.ByteScanner` 和 `io.RuneScanner` 接口。

`str.go` 第三部分代码如下：

```
    b := make([]byte, 1)
    for {
        n, err := r.Read(b)
        if err == io.EOF {
            break
        }

        if err != nil {
            fmt.Println(err)
            continue
        }

        fmt.Printf("Read %s Bytes: %d\n", b, n)
    }
```

此处，你可以看到如何使用 `strings.Reader` 作为 `io.Reader` 接口，从而使用 `Read()` 函数逐字节读取字符串。

`str.go` 的最后一段代码如下：

```
    s := strings.NewReader("This is an error!\n")
    fmt.Println("r length:", s.Len())
    n, err := s.WriteTo(os.Stderr)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Wrote %d bytes to os.Stderr\n", n)
}
```

在这段代码中，你可以看到如何在 `strings` 包的帮助下编写标准错误。

```
$ go run str.go
r length 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
This is an error!
Wrote 18 bytes to os.Stderr
$ go run str.go 2>/dev/null
r length 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
Wrote 18 bytes to os.Stderr
```

关于bytes包

`byte` 包包含处理字节切片的函数，方法与帮助处理字符串的 `strings` 包方法相同。 `Go` 源代码文件的名称是 `bytes.go` 。它分为三部分。

> *这部分内容不包含在 `Go` 系统编程中，Packet出版，2017*

`byte.go` 第一部分代码如下：

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "os"
)
```

`byte.go` 第二部分代码如下：

```
func main() {
    var buffer bytes.Buffer

    buffer.Write([]byte("This is"))
    fmt.Fprintf(&buffer, " a string!\n")
    buffer.WriteTo(os.Stdout)
    buffer.WriteTo(os.Stdout)
```

首先，创建一个新的 `bytes.Buffer` 变量，通过 `buffer.Write()` 和 `fat.Fprintf()` 写入数据。然后调用 `buffer.WriteTo()` 两次。

第一次调用 `buffer.WriteTo()` 将打印 `buffer` 的内容。然而，第二次调用 `buffer.WriteTo()` 将不会有任何输出，因为在第一次调用后， `buffer` 内容为空。

`bytes.go` 最后一部分代码如下：

```
    buffer.Reset()
    buffer.Write([]byte("Mastering Go!"))
    r := bytes.NewReader([]byte(buffer.String()))
    fmt.Println(buffer.String())
    for {
        b := make([]byte, 3)
        n, err := r.Read(b)
        if err == io.EOF {
            break
        }

        if err != nil {
            fmt.Println(err)
            continue
        }

        fmt.Printf("Read %s Bytes: %d\n", b, n)
    }
}
```

`Reset()` 方法重置 `buffer` 变量，通过 `Write()` 方法再次写入一些数据。然后你可以通过 `bytes.NewReader()` 创建新的读对象，再使用 `io.Reader` 接口方法 `Read()` 从 `buffer` 变量读取数据。

执行 `bytes.go` 会产生如下的输出：

```
$ go run byte.go
This is a string!
Mastering Go!
Read Mas Bytes: 3
Read ter Bytes: 3
Read ing Bytes: 3
Read  Go Bytes: 3
Read ! Bytes: 1
```

文件权限

`Unix` 系统编程中的一个热门话题是 `Unix` 文件权限。在本节中，假设你有足够的 `Unix` 权限，你将学习如何输出任意文件的权限！程序名为 `permission.go` ，分为三部分。

`permission.go` 的第一部分代码如下：

```
package main

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

`permission.go` 的第二部分代码如下：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("usage:permissions filename\n")
        return
    }
```

最后一部分代码如下：

```
    filename := arguments[1]
    info, _ := os.Stat(filename)
    mode := info.Mode()
    fmt.Println(filename, "mode is", mode.String()[1:10])
}
```

`os.Stat(filename)` 调用返回一个大结构体，其中包含了许多数据。因为我们只对文件的权限感兴趣，我们调用 `Mode()` 方式并打印输出。实际上，我们通过 `mode.String()[1:10]` 打印了输出的一部分，因为它即是我们感兴趣的那部分。

执行 `permission.go` 将产生如下输出：

```
$ go run permission.go /tmp/adobegc.log
/tmp/adobegc.log mode is rw-rw-rw-
$ go run permissions.go /dev/random
/dev/random mode is crw-rw-rw
```

`ls(1)` 输出可以验证 `permission.go` 的正确性：

```
$ ls -l /dev/random /tmp/adobegc.log
crw-rw-rw- 1 root wheel 14,   0 Jan 8 20:24 /dev/random
-rw-rw-rw- 1 root wheel  583454 Jan 16 19:12 /tmp/adobegc.log
```

处理 Unix 信号

Go 为程序员提供了 os/signal 包来处理信号。本节将向你显示如何处理 Unix 信号。

首先，让我们介绍 Unix 信号的一些有用信息。你是否使用 Ctrl+C 停止正在运行的程序？如果是，则它们已经比较相似，因为 Ctrl+C 向程序发送 SIGINT signal 。严格来讲， Unix 信号是可以通过名称或数字访问的软件中断，它们提供了在 Unix 系统上处理异步事件的方法。有两种方式发送信号：通过名字或者通过数字。通常来说，通过名字发送信号比较安全，因为你不太可能不小心发出错误的信号。

一个程序是不可能处理所有的信号的：一些信号既不能被捕获，也不能被忽略。 SIGKILL 和 SIGSTOP 信号不能被捕获、阻塞或忽略。这样做的原因是，它们为内核和根用户提供了一种停止任何进程的方法。 SIGKILL 信号，即数字 9 ，通常在需要响应迅速的极端条件下调用。因此，它是唯一一个通常由数字调用的信号，这仅仅是因为这样做更快。

> *signal.SIGINFO 在 Linux 机器上不可用。这意味着，如果你要在一个 Linux 机器上运行包含它的 Go 程序，则需要用另一个信号替换它，否则 Go 程序将无法编译和执行。*

给进程发送信号的最常用方式是通过 kill(1) 方法。默认情况下， kill(1) 发送 SIGTERM 信号。如果你想查看 Unix 机器上支持的所有信号，可以执行 kill -l 命令。

如果你在无权限的情况下给一个进程发送信号， kill(1) 并不会执行，且会返回类似如下的错误提示：

```
$ kill 1210
-bash: kill: (1210) - Operation not permitted
```

处理两种信号

在本小节中，你将学习如何在 Go 程序中处理两种信号，代码见 handleTwo.go ,分为四部分。 handleTwo.go 处理的信号是 SIGINFO 和 SIGINT ，在Golang中称为 syscall.SIGINFO 和 os.Interrupt 。

> *如果你查看 os 包文档，会发现在所有系统上只保证存在两个 siganal ，分别是 syscall.SIGKILL 和 syscall.SIGINT ，在 Go 中也定义为 os.Kill 和 os.Interrupt 。*

handleTwo.go 第一部分包含如下代码：

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)
```

handleTwo.go 第二部分代码如下：

```
func handleSignal(signal os.Signal) {
    fmt.Println("handleSignal() Caught:", signal)
}
```

handleSignal 函数用于处理 syscall.SIGINFO 信号，而 os.interrupt 信号将被内联处理。 handleTwo.go 第三部分代码如下：

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, os.Interrupt, syscall.SIGINFO)

    go func() {
        for {
            sig := <-sigs
            switch sig {
            case os.Interrupt:
                fmt.Println("Caught:", sig)
            case syscall.SIGINFO:
                handleSignal(sig)
                return
            }
        }
    }()
```

本技术工作原理如下：首先，你需要定义一个通道 `sigs` 用于传递数据。然后调用 `signal.Notify()` 声明你感兴趣的信号。下一步，你实现一个匿名函数，作为 `goroutine` 运行以便在收到关心的任何一个信号时进行操作。你需要等待 `Chapter 9,Go Concurrency-Goroutines,Channels,and Pipelines` ，学习 `goroutine` 和 `channels` 。

`handleTwo.go` 最后一部分程序如下：

```
    for {
        fmt.Printf(".")
        time.Sleep(20 * time.Second)
    }
}
```

`time.Sleep()` 调用用于阻止程序结束。在实际应用中，不需要使用类似代码。

在调用 `kill(1)` 时，我们需要程序的进程 `ID` ，我们首先编译 `handleTwo.go` ，并运行可执行文件，而不是 `go run handleTwo.go` 。 `handleTwo` 输出如下：

```
$ go build handleTwo.go
$ ls -l handleTwo
-rwxr-xr-x 1 mtsouk staff 2005200 Jan 18 07:49 handleTwo
$ ./handleTwo
.^CCaught: interrupt
.Caught: interrupt
handleSignal() Caught: information request
.Killed:9
```

注意你需要另一个终端和 `handleTwo.go` 交互，并获取输出。在终端执行命令如下：

```
$ ps ax | grep ./handleTwo | grep -v grep
47988 s003 S+  0:00.00 ./handleTwo
$ kill -s INT 47988
$ kill -s INFO 47988
$ kill -s USR1 47988
$ kill -9 47988
```

第一条命令用于查找 `handleTwo` 的进程 `ID`，剩余的命令用于向进程发送信号。信号 `SIGUSR1` 被忽略了，在输出中没有显示。

`handleTwo.go` 的问题是，如果它得到一个未被编程处理的信号，它将忽略它。因此，在下一节中，你将看到一种使用相对不同的方法以更有效的方式处理信号的技术。

处理所有信号

在本小节，你将学习如何处理所有信号，但只对真正感兴趣的信号作出响应。和上一节相比，它技术更好，并更安全。该技术代码参考 `handleAll.go` ，分为四部分。

`handleAll.go` 第一部分代码如下：

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func handle(signal os.Signal) {
    fmt.Println("Received:", signal)
}
```

`handleAll.go` 第二部分代码如下：

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs)
```

所以，所有的魔法都在 `signal.Notify(sigs)` 调用上。由于没有指定信号，所有输入信号都将被处理。

> *你可以在同一程序中使用不同的通道和相同的信号多次调用 `signal.Notify` 。在这种情况下，每个相关通道都将收到一份它要处理的信号副本！*

`handleAll.go` 第三部分代码如下：

```
        for {
            sig := <-sigs
            switch sig {
            case os.Interrupt:
                handle(sig)
            case syscall.SIGTERM:
                handle(sig)
                os.Exit(0)
            case syscall.SIGUSR2:
                fmt.Println("Handling syscall.SIGUSR2!")
            default:
                fmt.Println("Ignoring:", sig)
            }
        }
    }()
```

使用其中一个信号退出程序非常方便。

这给了你在需要时在程序中做一些内务管理的机会。此时，`syscall.SIGERM`信号被用于这个目的。但这并不妨碍你使用`SIGKILL`来终止程序。

`handleAll.go`剩余代码如下：

```
    for {
        fmt.Printf(".")
        time.Sleep(30 * time.Second)
    }
}
```

你仍需要调用`time.Sleep()`以阻止程序立即退出。

同样，最好使用`go build`工具编译`handleAll.go`生成可执行文件。在新的终端中执行`handleAll`会产生如下的输出：

```
$ go build handleAll.go
$ ls -l handleAll
-rwxr-xr-x 1 mtsouk staff 2005216 Jan 18 08:25 handleAll
$ ./handleAll
.Ignoring: hangup
Handling syscall.SIGUSR2!
Ignoring: user defined signal 1
Received: interrupt
^CReceived: interrupt
Received: terminated
```

另一个终端命令输出如下：

```
$ ps ax | grep ./handleAll | grep -v grep
49902 s003 S+ 0:00.00 ./handleAll
$ kill -s HUP 49902
$ kill -s USR2 49902
$ kill -s USR1 49902
$ kill -s INT 49902
$ kill -s TERM 49902
```

`Unix` 管道编程

根据 `Unix` 原理， `Unix` 命令行实用程序应该仅且仅执行一个任务！在实践中，这意味着，与其开发执行大量任务的大型实用程序，不如开发多个较小的程序，这些程序组合在一起执行所需的工作。两个或多个 `Unix` 命令行实用程序通信的最常见方式是使用管道。在 `Unix` 管道中，一个命令行实用程序的输出是另一个命令行实用程序的输入。此过程可能涉及两个以上的程序！ `Unix` 管道的标识是字符 `|` 。

管道有两个严重的限制：第一，它们通常是单向通信；第二，它们只能在具有相同祖先的进程之间使用。 `Unix` 管道实现背后的理念是，如果没有要处理的文件，那么应该等待从标准输入中获取输入。同样，如果不要求将输出保存到文件中，则应将输出写入标准输出，以便用户查看或由其他程序处理。

在第一节 `Go与操作系统` 中，你学习了如何从标准输入中读取，以及如何写入到标准输出。如果你对这两个操作仍有疑问，最好花点时间复习下代码 `stdOUT.go` 和 `stdIN.go` 。

遍历目录树

本节将介绍 `Go` 编写的 `find(1)` 命令行程序的一个相对简单的版本。

程序名为 `gofind.go` ，分为四部分介绍。 `gofind.go` 的代码将支持两个命令行选项。 `-d` 选项在作为目录的路径前面打印一个星形字符，而 `-f` 选项在作为常规文件的路径前面打印一个加号字符。

`goFind.go` 第一部分代码如下：

```
package main

import (
    "flag"
    "fmt"
    "os"
    "path/filepath"
)
```

如你所见， `goFind.go` 程序使用 `flag` 包高效地处理命令行参数。

`goFind.go` 第二部分代码如下：

```
var minusD bool = false
var minusF bool = false

func walk(path string, info os.FileInfo, err error) error {
    fileInfo, err := os.Stat(path)
    if err != nil {
        return err
    }
    mode := fileInfo.Mode()
    if mode.IsRegular() && minusF {
        fmt.Println("+", path)
        return nil
    }
    if mode.IsDir() && minusD {
        fmt.Println("*", path)
        return nil
    }
    fmt.Println(path)
    return nil
}
```

由于 `minusD` 和 `minusF` 应该可以从程序中的任何地方访问，包括 `walk()` 函数，我决定将它们设置为全局变量。 `IsDir()` 函数用于识别目录， `isRegular()` 用于识别常规文件。

`goFind.go` 第三部分代码如下：

```
func main() {
    starD := flag.Bool("d", false, "Signify directories")
    plusF := flag.Bool("f", false, "Signify regular files")
    flag.Parse()
    flags := flag.Args()
    Path := "."
    if len(flags) == 1 {
        Path = flags[0]
    }
```

`goFind.go` 最后一部分代码如下：

```
    minusD = *starD
    minusF = *plusF
    err := filepath.Walk(Path, walk)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```

执行 `goFind.go` 将创建如下的输出：

```
$ go run goFind.go -d -f /tmp/
* /tmp/
+ /tmp/.keystone_install_lock
* /tmp/5580C65A-E7E2-4B27-AD91-506F85545E1D
* /tmp/569A57CB-8FD3-4879-A6A3-B86116CB0116
+ /tmp/ExmanProcessMutex
+ /tmp/adobegc.log
/tmp/com.adobe.AdobeIPCBroker.ctrl-mtsouk
* /tmp/com.apple.launchd.h3Izgq45dz
/tmp/com.apple.launchd.h3Izgq45dz/Listeners
* /tmp/lilo.46843
+ /tmp/swtag.log
/tmp/textmate-501.sock
$ go run goFind.go -f /tmp/
/tmp/
+ /tmp/.keystone_install_lock
/tmp/5580C65A-E7E2-4B27-AD91-506F85545E1D
/tmp/569A57CB-8FD3-4879-A6A3-B86116CB0116
+ /tmp/ExmanProcessMutex
+ /tmp/adobegc.log
/tmp/com.adobe.AdobeIPCBroker.ctrl-mtsouk
/tmp/com.apple.launchd.h3Izgq45dz
/tmp/com.apple.launchd.h3Izgq45dz/Listeners
/tmp/lilo.46843
+ /tmp/swtag.log
/tmp/textmate-501.sock
$ go run goFind.go /tmp/
/tmp/
/tmp/.keystone_install_lock
/tmp/5580C65A-E7E2-4B27-AD91-506F85545E1D
/tmp/569A57CB-8FD3-4879-A6A3-B86116CB0116
/tmp/ExmanProcessMutex
/tmp/adobegc.log
/tmp/com.adobe.AdobeIPCBroker.ctrl-mtsouk
/tmp/com.apple.launchd.h3Izgq45dz
/tmp/com.apple.launchd.h3Izgq45dz/Listeners
/tmp/lilo.46843
/tmp/swtag.log
/tmp/textmate-501.sock
```

`goFind.go` 的输出包含了指定根目录下的所有文件类型，包括套接字和符号链接。如果对walk（）函数进行必要的更改，你可以告诉 `gofind.go` 只打印你真正感兴趣的信息。这是留给你做的练习。

> *现在想象一下使用 `C` 编程语言的标准库函数来实现相同的程序有多困难！*

使用 ePBF

eBPF 是增强的伯克利包过滤器，它是一个内核的虚拟机，集成到 Linux 内核中，可用于 Linux 跟踪。为了能够使用 eBPF ，需要使用 CONFIG_BPF_SYSCALL 选项编译内核，该选项在 Ubuntu Linux 上自动激活。

> *eBPF 在内核版本相对较新的 Linux 机器上工作，但不在 macOS 或 Mac OS X 机器上工作。*

你可以在 https://github.com/iovisor/bcc 上了解更多关于 eBPF 的信息，在 https://kinvolk.io/blog/2016/11/introducing-gobpf---using-ebpf-from-go/ 上了解 eBPF 和 Go 。 gobpf 包可在 https://github.com/iovisor/gobpf/ 上找到。

不幸的是，对 eBPF 和 Go 的进一步讨论超出了本书的范围。

关于 syscall.PtraceRegs

你可能以为你已经完全掌握了 syscall 标准库，但是你错了！在本节中，我们将使用 syscall.PtraceRegs ，它是一个保存寄存器状态信息的结构。

> *本节不包含在我的《 Go 系统编程》书籍中， Packt 出版社，2017 年*

现在，你将学习如何使用 ptraceRegs.go 在屏幕上打印如下所有寄存器的值，该代码将分为四部分。 ptraceRegs.go 主要是调用 syscall.PtraceGetRegs() 函数-还有 syscall.PtraceSetRegs() 、 syscall.PtraceAttach() 、 syscall.PtracPeekData() 和 syscall.PtracePokeData() 函数，这些函数可以帮助你使用寄存器，但 ptraceRegs.go 中不会使用这些函数。

ptraceRegs.go 的第一部分代码如下：

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
    "time"
)
```

ptraceRegs.go 的第二部分代码如下：

```
func main() {
    var r syscall.PtraceRegs
    cmd := exec.Command(os.Args[1], os.Args[2:]...)
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
```

ptraceRegs.go 的第三部分代码如下：

```
    cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}
    err := cmd.Start()
    if err != nil {
        fmt.Println("Start:", err)
        return
    }
    err = cmd.Wait()
    fmt.Printf("State: %v\n", err)
    wpid := cmd.Process.Pid
```

在前面的 `Go` 代码中，你调用一个外部命令，该命令在程序的命令行参数中指定，并找到其进程 `ID`，该 `ID` 将在 `syscall.PtraceGetRegs()` 调用中使用。语句 `&syscall.SysProcAttr{Ptrace:true}` 指定要在子进程上使用 `ptrace`。

`ptraceRegs.go` 最后一部分代码如下：

```
    err = syscall.PtraceGetRegs(wpid, &r)
    if err != nil {
        fmt.Println("PtraceGetRegs:", err)
        return
    }
    fmt.Printf("Registers: %#v\n", r)
    fmt.Printf("R15=%d, Gs=%d\n", r.R15, r.Gs)
    time.Sleep(2 * time.Second)
}
```

此处调用 `syscall.PtraceGetRegs()`，并打印存储在变量 `r` 中的结果，该变量应作为指针传递。

在 `macOS High Sierra` 机器上执行 `ptraceRegs.go`，输出如下：

```
$ go run ptraceRegs.go
# command-line-arguments
./ptraceRegs.go:11:8: undefined: syscall.PtraceRegs
./ptraceRegs.go:14:9: undefined: syscall.PtraceGetRegs
```

这意味着程序不支持运行在 `macOS` 和 `OS X` 上。

在 `Debian Linux` 机器上执行 `ptraceRegs.go`，输出如下：

```
$ go version
go version go1.3.3 linux/amd64
$ go run ptraceRegs.go echo "Mastering Go!"
State: stop signal: trace/breakpoint trap
Registers: syscall.PtraceRegs{R15:0x0, R14:0x0, R13:0x0, R12:0x0
Rbx:0x0, R11:0x0, R10:0x0, R9:0x0, R8:0x0, Rax:0x0, Rcx:0x0, Rdx
Rdi:0x0, Orig_rax:0x3b, Rip:0x7f8b1200e130, Cs:0x33, Eflags:0x20
Rsp:0x7ffd9d53f320, Ss:0x2b, Fs_base:0x0, Gs_base:0x0, Ds:0x0, E
Gs:0x0}
R15=0, Gs=0
Mastering Go!
```

你也可以在 `syscall` 包的文档中找到寄存器列表。

跟踪系统调用

本节将介绍一种非常先进的技术，它使用 `syscall` 包，并允许你监视在 `Go` 程序中执行的系统调用。

> *本节不包含在我的《`Go` 系统编程》书籍中，`Packt` 出版社，2017 年*

`Go` 程序名为 `traceSyscall.go`，并分为五部分。`traceSyscall.go` 第一部分代码如下：

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "os/exec"
    "strings"
    "syscall"
)

var maxSyscalls = 0

const SYSCALLFILE = "SYSCALLS"
```

你将很快学习 `SYSCALLFILE` 变量的作用。

`traceSyscall.go` 第二部分代码如下：

```
func main() {
    var SYSTEMCALLS []string
    f, err := os.Open(SYSCALLFILE)
    defer f.Close()
    if err != nil {
        fmt.Println(err)
        return
    }

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        line := scanner.Text()
        line = strings.Replace(line, " ", "", -1)
        line = strings.Replace(line, "SYS_", "", -1)
        temp := strings.ToLower(strings.Split(line, "=")[0])
        SYSTEMCALLS = append(SYSTEMCALLS, temp)
        maxSyscalls++
    }
```

请注意，`SYSCALLS` 文件的信息取自 `syscall` 包的文档，它将每个系统调用与一个数字相关联，该数字是系统调用的内部 `Go` 表示形式。该文件主要用于打印被跟踪程序所使用的系统调用的名称。

`SYSCALLS` 文件的格式如下：

```
SYS_READ = 0
SYS_WRITE = 1
SYS_OPEN = 2
SYS_CLOSE = 3
SYS_STAT = 4
```

在读取文本文件后，程序创建名为 `SYSTEMCALLS` 的切片来存储信息。

`traceSyscall.go` 第三部分代码如下：

```
    COUNTER := make([]int, maxSyscalls)
    var regs syscall.PtraceRegs
    cmd := exec.Command(os.Args[1], os.Args[2:]...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}
    err = cmd.Start()
    err = cmd.Wait()
    if err != nil {
        fmt.Println("Wait:", err)
    }
    pid := cmd.Process.Pid
    fmt.Println("Process ID:", pid)
```

`COUNTER` 切片存储在被跟踪的程序中每个系统调用的次数。

`traceSyscall.go` 的第四部分代码如下：

```
    before := true
    forCount := 0
    for {
        if before {
            err := syscall.PtraceGetRegs(pid, &regs)
            if err != nil {
                break
            }
            if regs.Orig_rax > uint64(maxSyscalls) {
                fmt.Println("Unknown:", regs.Orig_rax)
                return
            }
            COUNTER[regs.Orig_rax]++
            forCount++
        }
        err = syscall.PtraceSyscall(pid, 0)
        if err != nil {
            fmt.Println("PtraceSyscall:", err)
            return
        }
        _, err = syscall.Wait4(pid, nil, 0, nil)
        if err != nil {
            fmt.Println("Wait4:", err)
            return
        }
        before = !before
    }
```

`syscall.PtraceSyscall()` 函数的作用是：告诉 `Go` 继续执行正在被跟踪的程序，但是当程序进入或退出系统调用时停止执行，这正是我们想要的！由于每个系统调用在被调用之前和完成其工作之后都会被跟踪，因此我们使用 `before` 变量来计算每个系统调用仅一次。

`traceSyscall.go` 的最后一部分代码如下：

```
    for i, x := range COUNTER {
        if x != 0 {
            fmt.Println(SYSTEMCALLS[i], "->", x)
        }
    }
    fmt.Println("Total System Calls:", forCount)
}
```

在这一部分中，我们打印切片 `COUNTER` 的内容。切片 `SYSTEMCALLS` 用于在知道系统调用的 `Go` 数字表示时，来查找系统调用的名称。

在 `macOS High Sierra` 机器上执行 `traceSyscall.go` 会创建如下的输出：

```
$ go run traceSyscall.go
# command-line-arguments
./traceSyscall.go:36:11: undefined: syscall.PtraceRegs
./traceSyscall.go:57:11: undefined: syscall.PtraceGetRegs
./traceSyscall.go:70:9: undefined: syscall.PtraceSyscall
```

同样， `traceSyscall.go` 程序不能在 `macOS` 和 `Mac OS X` 上运行。

在 `Debian Linux` 机器上执行程序会创建如下的输出：

```
$ go run traceSyscall.go ls /tmp/
Wait: stop signal: trace/breakpoint trap
Process ID: 5657
go-build084836422 test.go upload_progress_cache
read -> 11
write -> 1
open -> 37
close -> 27
stat -> 1
fstat -> 25
mmap -> 39
mprotect -> 16
munmap -> 4
brk -> 3
rt_sigaction -> 2
rt_sigprocmask -> 1
ioctl -> 2
access -> 9
execve -> 1
getdents -> 2
getrlimit -> 1
statfs -> 2
arch_prctl -> 1
futex -> 1
set_tid_address -> 1
openat -> 1
set_robust_list -> 1
Total System Calls: 189
```

在程序结束时， `traceSyscall.go` 打印程序中调用每个系统调用的次数！ `traceSyscall.go` 的正确性通过 `strace -c` 程序的输出进行验证。

```
$ strace -c ls /tmp
test.go upload_progress_cache
% time seconds usecs/call calls errors syscall
------ ----------- ----------- --------- --------- ----------------
0.00  0.000000          0    11         read
0.00  0.000000          0     1         write
0.00  0.000000          0    37      13 open
0.00  0.000000          0    27         close
0.00  0.000000          0     1         stat
0.00  0.000000          0    25         fstat
0.00  0.000000          0    39         mmap
0.00  0.000000          0    16         mprotect
0.00  0.000000          0     4         munmap
0.00  0.000000          0     3         brk
0.00  0.000000          0     2         rt_sigaction
0.00  0.000000          0     1         rt_sigprocmask
0.00  0.000000          0     2         ioctl
0.00  0.000000          0     9       9 access
0.00  0.000000          0     1         execve
0.00  0.000000          0     2         getdents
0.00  0.000000          0     1         getrlimit
0.00  0.000000          0     2       2 statfs
0.00  0.000000          0     1         arch_prctl
0.00  0.000000          0     1         futex
0.00  0.000000          0     1         set_tid_address
0.00  0.000000          0     1         openat
0.00  0.000000          0     1         set_robust_list
------ ----------- ----------- --------- --------- ----------------
100.00 0.000000              189        24 total
```

User ID 和 group ID

在本章的最后一节中，你将学习如何查找当前用户的用户 ID 以及当前用户所属的组 ID 。用户 ID 和组 ID 都是保存在 UNIX 系统文件中的正整数。

程序名为 ids.go ，分为两部分。第一部分代码如下：

```
package main

import (
    "fmt"
    "os"
    "os/user"
)

func main() {
    fmt.Println("User id:", os.Getuid())
```

查找当前用户的用户 ID 与调用 os.Getuid() 函数一样简单。

ids.go 第二部分代码如下：

```
    var u *user.User
    u, _ = user.Current()
    fmt.Print("Group ids: ")
    groupIDs, _ := u.GroupIds()
    for _, i := range groupIDs {
        fmt.Print(i, " ")
    }
    fmt.Println()
}
```

另一方面，查找用户所属的组 ID 是一项更棘手的任务。

执行 ids.go 输出如下：

```
$ go run ids.go
User id: 501
Group ids: 20 701 12 61 79 80 81 98 33 100 204 250 395 398 399
```

其他资源

你会发现如下 `Web` 资源非常有用：

- `io` 包文档可以参考 `https://golang.org/pkg/io/` 。
- 通过访问 `ts` 官网 `https://github.com/Arafatk/glot` ，可以了解更多关于 `Glot` 绘图库的信息。
- 访问 `https://golang.org/pkg/encoding/binary/` 了解更多关于编码/二进制标准包的信息。
- 访问 `https://golang.org/pkg/encoding/gob/` 获取 `encoding/gob` 包的文档
- 访问 `http://www.brendangregg.com/ebpf.html` 学习 `eBPF` 。你也可以观看视频 `https://www.youtube.com/watch?v=JRFNIKUROPE and https://www.youtube.com/watch? v=w8nFRoFJ6EQ` 。
- 你可以在很多地方学习 `Endianness` ，如 `https://en.wikipedia.org/wiki/Endianness` 。
- 访问 `https://golang.org/pkg/flag/` 查看 `flag` 包的文档。

练习

- 编写一个 `Go` 程序，它有三个参数：一个文本文件的名称，和两个字符串。程序应使用第二个字符串替换文件中每次出现的第一个字符串。由于安全原因，最终输出将打印在屏幕上，也就是说原始文本文件将保持完整。
- 使用 `encoding/gob` 包序列化和反序列化 `Go` 映射以及结构体的切片。
- 创建一个 `Go` 程序，处理你选择的任意三个信号。
- 在 `Go` 中创建一个程序，将文本文件中的所有制表符替换为命令行中指定数量的空格符。同样，输出将打印在屏幕上。
- 开发一个程序，逐行读取文本文件并使用 `strings.TrimSpace()` 函数从每一行中删除空格字符。
- 修改 `kvSaveLoad.go` 以支持单个命令行参数，该参数是个文件名，用于加载和保存数据。
- 你能创建 `wc(1)` 程序的 `Go` 版本吗？看一下 `wc(1)` 的手册页，找到有关命令行选项的信息支持。
- 修改 `goFind.go` 的代码只打印普通文件。这意味着它不应该打印目录、套接字和符号链接。
- 你能编写一个使用 `Glot` 来绘制函数的程序吗？
- 修改 `traceSyscall.go` 以便在跟踪时显示每个系统调用。
- 修改 `cat.go` 仅支持 `io.Copy(os.Stdout,f)` ，直接复制文件的内容，而不是扫描所有的内容。
- 你还可以使用 `bufio.Newscanner()` 和 `bufio.ScanWords` 逐字逐句地读一行。了解如何创建一个新版本的 `byWord.go` 程序。

总结

本章谈到了很多有趣的话题，包括阅读文件、写入文件、以及使用 `flag` 包。然而，有很多与系统编程相关的话题未在本章提及，如使用目录、复制、删除和重命名文件、处理 `Unix` 用户、组和 `Unix` 进程、使用环境变量如 `PATH` 、改变 `Unix` 文件权限、生成稀疏文件、读取和保存 `JSON` 数据、锁定和创建文件，使用和旋转日志文件，以及 `os.Stat()` 调用返回的结构中的信息。

GO并发-协程，通道和管道

上一章我们讨论了Go系统编程，包括Go函数和与操作系统通信的技术。前一章未涉及的两个系统编程领域是并发编程以及创建和管理多个线程。这两个主题将在本章和下一章中讨论。

GO提供了自己独特而新颖的方式来实现并发，这就是协程和通道。协程Goroutine是Go程序中可独立执行的最小实体单元，而通道Channel是Goroutine间并发且有效地获取数据的方式，从而允许goroutine具有引用点并相互通信。Go中的所有内容都是使用goroutines执行的，这是完全合理的，因为Go是一种并发编程语言。因此，当Go程序开始执行时，它的单个goroutine调用main（）函数，该函数执行实际的程序。

本章的主要内容如下：

- 进程，线程和Go协程之间的区别
- Go调度器
- 并发与并行
- 创建Go协程
- 创建通道
- 从通道读取或接收数据
- 往通道里写或发送数据
- 创建管道
- 等待你的Go协程结束

关于进程，线程和Go协程

进程是包含计算机指令，用户数据和系统数据的程序执行环境，以及包含其运行时获得的其他类型资源。而程序是一个文件，其中包含用于初始化进程的指令和用户数据部分的指令和数据。

线程相对于进程是更加小巧而轻量的实体，线程由进程创建且包含自己的控制流和栈，线程和进程的区别在于：进程是正在运行的二进制文件，而线程是进程的子集。

Goroutine是Go程序并发执行的最小单元，因为Goroutine不是像Unix进程那样是自治的实体，Goroutine生活在Unix进程的线程中。Goroutine的主要优点是非常轻巧，运行成千上万或几十万都没有问题。

总结一下，Goroutine比线程更轻量，而线程比进程更轻量。实际上，一个进程可以有多个线程以及许多Goroutine，而Goroutine需要一个进程的环境才能存在。因此，为了创建一个Goroutine，你需要有一个进程且这个进程至少有一个线程--Unix负责进程和线程管理，而Go工程师只需要处理Goroutine，这极大的降低了开发的成本。

到现在为止，你知道了关于进程，线程和Goroutine的基本知识，下一小节我们聊聊Go调度器。

Go调度器

Unix内核调度程序负责程序线程的执行。另一方面，Go运行时也有自己的调度程序，它使用称为 m:n 的调度技术负责执行Goroutine，使用多路复用使n个操作系统线程执行m个Goroutine。Go调度程序是Go的组件，负责Go程序中Goroutine的执行方式和执行顺序。这使得Go调度程序成为Go编程语言中非常重要的一部分，因为Go程序中很多都是作为Goroutine执行的。

需要留意的是，由于Go调度程序仅处理单个程序的Goroutine，因此其操作比内核调度程序的操作更简单，更轻量，更快。

并发与并行

有一个非常普遍的误解，认为并发与并行是一回事 - 其实不然！并行是同时执行某种类型的多个实体，而并发是构建组件的一种方式，以便它们可以在可能的情况下独立执行。

当您的操作系统和硬件允许时,只有当您并发地构建软件组件时，才能以并行地方式安全执行。早在CPU拥有多个内核且计算机拥有大量RAM之前，Erlang编程语言就有过类似的实践。

在有效的并发设计中，添加并发实体会使整个系统运行得更快，因为可以并行运行更多内容。因此，期望的并行性来自更好的并发表达和问题的实现。开发人员负责在系统的设计阶段考虑并发性，并从系统组件的潜在并行执行中受益。因此，开发人员不应该考虑并行性，而应该将程序分解为独立的组件，以便于通过组合来解决前面提到的问题。

即使您无法在Unix机器上并行运行功能，有效的并发设计仍可以改进程序的设计和可维护性。换句话说，并发性优于并行性！

Goroutines

在Go语言中使用go关键字后跟函数名称或定义完整的匿名函数即可开启一个新的Goroutine，使用go关键字调用函数后会立即返回，该函数在后台作为Goroutine运行，程序的其余部分继续执行

但是，如上所述，您无法控制您的Goroutine的执行顺序，因为这取决于操作系统的调度程序，Go调度程序以及操作系统的负载。

创建一个Goroutine

在本小节中，您将学习两种创建goroutine的方法。第一种方法是使用常规的函数，而第二种方法是使用匿名函数 - 这两种方法其实是类似的。

本小节所展示的程序文件为simple.go，它分为三个部分。

simple.go的第一部分代码如下：

```
package main

import (
    "fmt"
    "time"
)

func function() {
    for i := 0; i < 10; i++ {
        fmt.Print(i)
    }
    fmt.Println()
}
```

除了import包之外，上面的代码定义了一个名为function()的函数，该函数将在下面的代码内使用。

接下来是simple.go的第二部分代码：

```
func main() {
    go function()
```

上面的代码启用一个新的Goroutine来运行function()函数。然后主程序会继续执行，而function()函数开始在后台运行。

simple.go的最后一部分代码如下：

```
go func() {
    for i := 10; i < 20; i++ {
        fmt.Print(i, " ")
    }
}()

    time.Sleep(1 * time.Second)
}
```

如上所示，您也可以使用匿名函数创建Goroutine。此方法适合相对较小的功能。如果函数体有大量代码，最好使用go关键字创建常规函数来执行它。

正如您将在下一节中看到的，您可以按照自己的方式创建多个Goroutine，当然也可以使用for循环。

执行simple.go两次后的输出如下：

```
$ go run simple.go
10 11 12 13 14 0123456789
15 16 17 18 19

$ go run simple.go
10 11 12 13 14 15 16 17 18 19 0123456789
```

尽管您想要的程序是对于同一个输入有相同的输出，但从上面的执行结果来看，两次的输出并不是相同的。我们可以总结一下：在不做额外工作的自然情况下我们是无法控制Goroutine的执行顺序的，如果要控制它，我们需要编写额外的代码。在下一章我们将学习到这部分内容。

优雅地结束goroutines

本节内容将介绍如何使用go标准库中的sync包来解决上一节提到的Goroutine中的任务还未执行完成，main()函数就提前结束的问题。 本节的代码文件为syncGo.go,我们基于上一节的create.go来扩展syncGo.go。

syncGo.go的第一部分代码如下：

```
package main

import (
    "flag"
    "fmt"
    "sync"
)
```

如上所示，我们不再需要time包，我们将使用sync包中的功能来等待所有的Goroutine执行完成。

在第10章“并发 - 高级主题”中，我们将会学习两种方式来对Goroutine进行超时处理。 syncGo.go的第二部分代码如下：

```
func main() {
    n := flag.Int("n", 20, "Number of goroutines")
    flag.Parse()
    count := *n
    fmt.Printf("Going to create %d goroutines.\n", count)

    var waitGroup sync.WaitGroup
```

在上面的代码中，我们定义了sync.WaitGroup类型的变量，查看sync包的源码我们可以发现，waitgroup.go文件位于sync目录中，sync.WaitGroup的定义只不过是一个包含三个字段的结构体：

```
type WaitGroup struct {
        noCopy noCopy
        state1 [12]byte
        sema   uint32
    }
```

syncGo.go的输出将显示有关sync.WaitGroup变量工作方式的更多信息。

syncGo.go的第三部分代码如下：

```
fmt.Printf("%#v\n", waitGroup)
    for i := 0; i < count; i++ {
        waitGroup.Add(1)
        go func(x int) {
            defer waitGroup.Done()
            fmt.Printf("%d ", x)
        }(i)
    }
```

在这里，您可以使用for循环创建所需数量的Goroutine。（当然，也可以写多个顺序的Go语句。）

每次调用sync.Add()都会增加sync.WaitGroup变量中的计数器。需要注意的是，在go语句之前调用sync.Add(1)非常重要，以防止出现任何竞争条件。当每个Goroutine完成其工作时，将执行sync.Done()函数，以减少相同的计数器。

syncGo.go的最后一部分代码如下：

```
fmt.Printf("%#v\n", waitGroup)
    waitGroup.Wait()
    fmt.Println("\nExiting...")
}
```

sync.Wait()调用将阻塞，直到sync.WaitGroup变量中的计数器为零，从而保证所有Goroutine能执行完成。

syncGo.go的输出如下：

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0,
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0,
19 7 8 9 10 11 12 13 14 15 16 17 0 1 2 5 18 4 6 3
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0,
1 0 4 5 17 7 8 9 10 11 12 13 2 sync.WaitGroup{noCopy:sync.noCopy
29 15 6 27 24 25 16 22 14 23 18 26 3 19 20 28 21
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0,
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0,
29 1 7 8 2 9 10 11 12 4 13 15 0 6 5 22 25 23 16 28 26 20 19 24 2
Exiting...
```

syncGo.go的输出因执行情况而异。另外，当Goroutines的数量为30时，一些Goroutine可能会在第二个fmt.Printf（“％＃v \ n”，waitGroup）语句之前完成它们的工作。最后需要注意sync.WaitGroup中的state1字段是一个保存计数器的元素，该计数器根据sync.Add()和sync.Done()调用而增加和减少。

当Add()和Done()的数量不匹配时会发生什么？

当sync.Add()和sync.Done()调用的数量相等时，程序会正常运行。但是，本节将告诉您当调用数量不一致时会发生什么。

假如我们执行sync.Add()的次数大于执行sync.Done()的次数，这种情况下，通过在第一个fmt.Printf（“％＃v \ n”，waitGroup）之前添加waitGroup.Add(1)语句，然后执行go run的输出如下：

```
$ go run syncGo.go Going to create 20 goroutines. sync.WaitGroup
```

Channel(通道)

通道是Go提供的一种通信机制，允许Goroutines之间进行数据传输。但也有一些明确的规则，首先，每个通道只允许交换指定类型的数据，也称为通道的元素类型，其次，要使通道正常运行，还需要保证通道有数据接收方。使用chan关键字即可声明一个新通道，且可以使用close()函数来关闭通道。

最后，有一个非常重要的细节：当您使用通道作为函数参数时，您可以指定其方向; 也就是说，该通道是用于发送数据或是接收数据。

通道的写入

这小节的代码将教您怎样往通道写入数据。写 `x` 值到 `c` 通道如同 `c <- x` 一样容易。这个箭头表示值的方向，只要 `x` 和 `c` 是相同的类型用这个表达式就不会有问题。这节的示例保存在 `writeCh.go` 中，并分三部分介绍。

`writeCh.go` 的第一段代码如下：

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println(x)
    c <- x
    close(c)
    fmt.Println(x)
}
```

`chan` 关键字是用于声明函数参数 `c` 是一个通道，并且伴随通道 (`int`) 类型。`c <-x` 表达式允许您写 `x` 值到 `c` 通道，并用 `close()` 函数关闭这个通道；那样就不会再和它通信了。

`writeCh.go` 的第二部分代码如下：

```
func main() {
    c := make(chan int)
```

上面的代码定义了一个名为 `c` 的通道变量，第一次在这章使用 `make()` 函数和 `chan` 关键字。所有的通道都有一个指定的类型。

`writeCh.go` 的其余代码如下：

```
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
}
```

这里以 goroutine 的方式执行 `writeToChannel()` 函数并调用 `time.Sleep()` 来给 `writeToChannel()` 函数足够的时间来执行。

执行 `writeCh.go` 将产生如下输出：

```
$go run writeCh.go
10
```

奇怪的是 `writeToChannel()` 函数只打印了一次给定的值。这是由于第二个 `fmt.Println(x)` 表达式没有执行。一旦您理解了通道的工作原理，这个原因就非常简单了：`c <- x` 表达式阻塞了 `writeChannel()` 函数下面的执行，因为没人读取 `c` 通道内写入的值。所以，当 `time.Sleep(1 * time.Second)` 表达式结束的时候，程序没有等待 `writeChannel()` 就结束了。

下节将说明怎么从通道读数据。

从通道接收数据

这小节，您将了解到如何从通道读取数据。您可以执行 `<-c` 从名为 `c` 的通道读取一个值。如此，箭头方向是从通道到外部。

我将使用名为 `readCh.go` 的程序帮您理解怎样从通道读取数据，并它分为三部分介绍。

`readCh.go` 的第一段代码如下：

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println("1", x)
    c <- x
    close(c)
    fmt.Println("2", x)
}
```

`writeToChannel()` 函数的实现与之前一样。

`readCh.go` 的第二部分如下：

```
func main() {
    c := make(chan int)
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
    fmt.Println("Read:", <-c)
    time.Sleep(1 * time.Second)
```

上面的代码，使用 `<-c` 语法从 `c` 通道读取数据。如果您想要保存数据到名为 `k` 的变量而不只是打印它的话，您可以使用 `k := <-c` 表达式。第二个 `time.Sleep(1 * time.Second)` 语句给您时间来读取通道数据。

`readCh.go` 的最后一段代码如下：

```
    _, ok := <-c
    if ok {
        fmt.Println("Channel is open!")
    }else {
        fmt.Println("Channel is closed!")
    }
}
```

从上面的代码，您能看到一个判断一个通道是打开还是关闭的技巧。当通道关闭时，当前代码运行的还不错。但是，如果通道被打开，这里的代码就会丢弃从通道读取的值，因为在 `_, ok := <-c` 语句中使用了 `_` 字符。如果您也想在通道打开时读取通道的值，就使用一个有意义的变量名代替 `_`。

执行 `readCh.go` 产生如下输出：

```
$go run readCh.go
1 10
Read: 10
Channel is closed!
$go run readCh.go
1 10
2 10
Read: 10
Channel is closed!
```

尽管输出不确定，但 `writeToChannel()` 函数的两个 `fmt->Println(x)` 表达式都被执行了，因为当您从通道读取数据时，它就被解除阻塞了。

通道作为函数参数传递

虽然 `readCh.go` 和 `writeCh.go` 没有使用这一功能，但 Go 允许您在把通道作为函数的参数时指定它的方向；那就是它是否用于读取或写入数据。通道有两种类型，**无向通道**和默认的双向通道。

查看下面两个函数代码：

```
func f1(c chan int, x int) {
    fmt.Println(x)
    c <- x
}

func f2(c chan<- int, x int){
    fmt.Println(x)
    c <- x
}
```

虽然两个函数实现了相同的功能，但它们的定义略有不同。`f2()` 函数的 `chan` 关键字右侧有个 `<-` 符号。这说明过 `c` 通道只能用于写数据。如果 Go 代码试图从一个只读通道（只发通道）读取数据的话，Go 编译器就会产生如下错误信息：

```
# command-line-arguments
a.go:19:11: invalid operation: range in (recevie from send-only
```

同样，有下面的函数定义：

```
func f1(out chan int64, in chan int64) {
    fmt.Println(x)
    c <- x
}

func f2(out chan<- int64, in <-chan int64) {
    fmt.Println(x)
    c <- x
}
```

`f2` 定义含有名为 `in` 的只读通道和 `out` 的只写通道。如果您试图向函数的一个只读通道（只收通道）参数写数据和关闭它时，会得到如下错误信息：

```
# command-line-arguments
a.go:13:7: invalid operation: out <- i(send to receive-only type
a.go:15:7: invalid operation: close(out)(cannot close receive-on
```

管道

管道是一个虚拟的方法用来连接 goroutines 和 通道，使一个 goroutine 的输出成为另一个的输入，使用通道传递数据。

使用管道的一个好处是程序中有不变的数据流，因此 goroutine 和 通道不必等所有都就绪才开始执行。另外，因为您不必把所有内容都保存为变量，就节省了变量和内存空间的使用。最后，管道简化了程序设计并提升了维护性。

我们使用 `pipeline.go` 代码来说明管道的使用。这个程序分六部分来介绍。 `pipeling.go` 程序执行的任务是在给定范围内产生随机数，当任何数字随机出现第二次时就结束。但在终止前，程序将打印第一个随机数出现第二次之前的所有随机数之和。您需要三个函数来连接程序的通道。程序的逻辑在这三个函数中，但数据流在管道的通道内。

这个程序有两个通道。第一个（通道A）用于从第一个函数获取随机数并发送它们到第二个函数。第二个（通道B）被第二个函数用来发送可接受的随机数到第三个函数。第三个函数负责从通道 B 获取数据，并计算结果和展示。

`pipeline.go` 的第一段代码如下：

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

var CLOSEA = false
var DATA = make(map[int]bool)
```

因为 `second()` 函数需要通过一种方式告诉 `first()` 函数关闭第一个通道，所以我使用一个全局变量 `CLOSEA` 来处理。 `CLOSEA` 变量只在 `first()` 函数中检查，并只能在 `second()` 函数中修改。

`pipeline.go` 的第二段代码如下：

```
func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func first(min, max int, out chan<- int) {
    for {
        if CLOSEA {
            close(out)
            return
        }
        out <- random(min, max)
    }
}
```

上面的代码展示了 `random` 和 `first` 函数的实现。您已经对 `random()` 函数比较熟悉了，它在一定范围内产生随机数。但真正有趣的是 `first()` 函数。它使用 `for` 循环持续运行，直到 `CLOSEA` 变为 `true` 。那样的话，它就关闭 `out` 通道。

`pipeline.go` 的第三段代码如下：

```
func second(out chan<- int, in <-chan int) {
    for x := range in {
        fmt.Print(x, " ")
        _, ok := DATA[x]

        if ok {
            CLOSEA = true
        }else {
            DATA[x] = true
            out <- x
        }
    }
    fmt.Println()
    close(out)
}
```

`second()` 函数从 `in` 通道接收数据，并发送该数据到 `out` 通道。但是， `second()` 函数一旦发现 `DATA` map 中已经存在了该随机数，它就把 `CLOSEA` 全局变量设为 `true` 并停止发送任何数据到 `out` 通道。然后，它就关闭 `out` 通道。

`pipeline.go` 的第四段代码如下：

```
func third(in <-chan int){
    var sum int
    sum = 0
    for x2 := range in {
        sum = sum + x2
    }
    fmt.Println("The sum of the random numbers is %d\n", sum)
}
```

`third` 函数持续从 `in` 通道读取数据。当通道被 `second()` 函数关闭时， `for` 循环停止获取数据，函数打印输出。从这很清楚的看到 `second()` 函数控制许多事情。

`pipeline.go` 的第五段代码如下：

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer paramters!")
        os.Exit(1)
    }
    n1, _ := strconv.Atoi(os.Args[1])
    n2, _ := strconv.Atoi(os.Args[2])

    if n1 > n2 {
        fmt.Printf("%d should be smaller than %d\n", n1, n2)
        return
    }
```

`pipeline.go` 的最后一段如下：

```
    rand.Seed(time.Now().UnixNano())
    A := make(chan int)
    B := make(chan int)

    go first(n1, n2, A)
    go second(B, A)
    third(B)
}
```

这里，定义了需要的通道，并执行两个 goroutines 和一个函数。 `third()` 函数阻止 `main` 返回，因为它不作为 goroutine 执行。

执行 `pipeline.go` 产生如下输出：

```
$go run pipeline.go 1 10
2 2
The sum of the random numbers is 2
$go run pipeline.go 1 10
9 7 8 4 3 3
The sum of the random numbers is 31
$go run pipeline.go 1 10
1 6 9 7 1
The sum of the random numbers is 23
$go run pipeline.go 10 20
16 19 16
The sum of the random numbers is 35
$go run pipeline.go 10 20
10 16 17 11 15 10
The sum of the random numbers is 69
$go run pipeline.go 10 20
12 11 14 15 10 15
The sum of the random numbers is 62
```

这里重点是尽管 `first` 函数按自己的节奏持续产生随机数，并且 `second()` 函数把它们打印在屏幕上，不需要的随机数也就是已经出现的随机数，不会发送给 `third()` 函数，因此就不会包含在最终的总和中。

其他学习资源

访问以下有用资源：

- `sync` 包的文档页在https://golang.org/pkg/sync。
- 还是 `sync` 包的文档页。关注 `sync.Mutext` 和 `sync.RWMutex` 类型，它们将在下章出现。

练习题

- 创建一个管道来读取文本文件，找到每个文件里给定短语的出现次数，并计算所有文件中该短语出现的总数。
- 创建一个管道来计算给定范围的所有自然数的平方和。
- 从 `simple.go` 程序总移除 `time.Sleep(1 * time.Second)` 表达式，看看会发生什么。为什么？
- 修改 `pipeling.go` 的代码来创建一个管道，用五个函数和适当的通道。
- 修改 `pipleline.go` 代码来找出当您忘记关闭 `first()` 函数中的 `out` 通道时会发生什么。

本章小结

在这章里，您了解到了许多 Go 的独特功能，包括 goroutines，通道和管道。另外，您学到了使用 `sync` 包提供的功能来给 goroutines 提供足够的时间去完成它们的任务。最后，您了解到了通道可以作为函数的参数。这允许开发者创建数据流管道。

下章将通过介绍 `select` 关键字来继续讨论 Go 的并发。这个关键字可以让通道执行许多有趣的任务，我想您会真正的被它的强大所震惊。之后，您将看到两个技巧，用于处理一个或多个因为某些原因而超时的 goroutines。然后，您将了解空通道，信号通道，通道的通道和缓冲通道，还有 `context` 包。

在下章您也会了解到**共享内存**，它是同一个 Unix 进程中的线程间共享信息的传统方式，它也适用于 goroutines。不过，共享内存在 Go 开发者中并不流行，因为 Go 提供了更好，更安全和更快速的方法给 goroutines 来交换数据。

Go 并发-进阶讨论

上一章介绍了 goroutines——Go 中最重要的特性，channels 和 pipelines。这章在讨论共享变量，`sync.Mutex` 和 `sync.RWMutex` 类型前，将继续从上章留下的这点出发来了解更多关于 goroutines，channels 和 `select` 关键字。这章也包括一些代码示例用于说明信号 channels，缓冲 channels，空 channels 和 channels 的 channels 的使用。另外，这章的前期您将了解俩个技巧用于一个 goroutine 在给定时间后的超时处理，因为没人能保证所有的 goroutines 能在期望的时间前都完成。这章将以检查竞争条件，`context` 标准包和工作池来结束。

在*玩转 Go* 的本章中，您将了解到如下主题：

- `select` 关键字
- Go 调度器如何工作
- 俩个技巧用于一个完成时长超过预期的 goroutine 的超时处理
- 信号 channels
- 缓冲 channels
- 空 channels
- 监控 goroutines
- channels 的 channels
- 共享内存和互斥器
- `sync.Mutex` 和 `sync.RWMutex` 类型
- `contenxt` 包和它的功能
- 工作池
- 探测竞争条件

重温调度器

调度器负责以有效的方式在可用资源上分配要完成的工作量。这节，我们将检查 Go 调度器的操作方式比上一章更深入些。您已经知道了 Go 工作使用 **m:n** 调度器（或**M:N 调度器**）来调度 goroutines——它比系统线程（使用系统线程）更轻量。然而先上我们来复习一些必要理论和一些有用的术语定义。

Go 使用 **fork-join 并发**模型。这个模型的 fork 部分代表一个程序在任何时间点创建的子分支。类似地，Go 并发模型的 join 部分是子分支将要接受的位置并加入它的父级。除此之外，收集 goroutines 结果的 `sync.Wait()` 和 channles 都是连接点，而任何新 goroutine 都会创建子分支。

> *fork-join 模型的 fork 阶段和 C 语言中的 `fork(2)` 调用完全是两回事*

公平调度策略是相当直接的，并有一个简单的实现，它在可用的处理器间均匀地共享所有负载。首先，这可能看起来是完美的策略，因为当它保持所有的处理器被均等占用时不必考虑太多事情。事实证明情况并非如此，因为大多数分布式任务通常依赖于其他任务。因此，最终一些处理器使用不足或者说一些处理器使用过剩。

在 Go 中 goroutine 是一个**任务**，而调用 goroutine 之后的所有内容是一个延续。在 Go 调度器使用的 **work stealing 策略** 中，一个未充分利用的（逻辑）处理器会从其他处理搜寻额外的工作。当它找到了这些工作，它就会从其他处理器偷取它们，这就是 *work stealing 策略*名称当由来。还有 Go 队列和窃取延续的 work-stealing 算法。一个**间歇接合点**，如其名所示，是一个执行线程在接合处间歇的点并开始寻找其他工作去做。尽管所有的任务窃取和延续窃取都有间歇接合点，但延续部分发生的更频繁；因此，Go 算法与延续部分工作而不是任务。

延续窃取的主要缺点是它需要来自编程语言的编译器的额外工作。幸运地，Go 提供了额外的帮助，因此在它的 work-stealing 策略中使用 **延续窃取**。

延续窃取的一个好处是当您尽使用函数或具有多个 goroutine 的单个线程时，您可用获得相同的结果。这是完全合理的，因为这这两种情况下，在任何给定的时间都只执行一件事。

现在，让我们回到 Go 中使用的 **m:n 调度**算法。严格讲，在任何时候，您都有 *m* 个 goroutines 被执行并计划运行在 `n` 个系统线程上，使用最多 `GOMAXPROCS` 个逻辑处理器。稍后您将了解 `GOMAXPROCS` 。

Go 调度器使用三种主要实体工作：系统线程（M）与使用的操作系统有关，goroutines（G）和 **逻辑处理器**（P）。Go 程序能使用的处理器数量由 **GOMAXPROCS 环境变量**定义——在任何给定时间里有最多 `GOMAXPROCS` 个处理器。

下图说明了这点：

这个插图告诉我们有俩个不同类型的队列：一个全局队列和一个关联到每个逻辑处理器的本地队列。全局队列里的 Goroutines 为了执行会被分配到逻辑处理器的队列中。因此，Go 调度器为了避免正在执行的 goroutines 只在每个逻辑处理器的本地队列中，需要检查全局队列。然而，全局队列不是总在被检查，意思是它没有比本地队列更有优势。另外，每个逻辑处理器有多个线程，并且窃取发生在可用逻辑处理器的本地队列之间。最后，注意当需要时， Go 调度器被允许创建跟多的系统线程。然而，系统线程相当昂贵，就是说处理系统线程太多会使您的 Go 应用变慢。

注意在程序中为了性能使用更多的 goroutines 不是万能方法，因为除了各种调用 `sync.Add()`， `sync.Wait()`，和 `sync.Done()` 外，更多的 goroutines 由于需要 Go 调度器做额外工作会使您的程序变慢。

> *Go 调度器，作为大多数 Go 组件总是在进化，就是说负责 Go 调度器的人不断努力改进它的性能，通过对它的工作方式做微小改变。然而这个核心原理保持一致*

所有这些细节有用吗？我想是的！为了使用 goroutines 写代码您需要知道所有这些吗？当然不！然而，当奇怪的事情发生时，或者您对 Go 调度器是如果工作的好奇，知道场景背后发生了什么一定能帮到您。这当然能使您成为更好的开发者！

环境变量 GOMAXPROCS

这个 `GOMAXPROCS` 环境变量（和Go 函数）允许您限制操作系统线程数，它能同时执行用户级 Go 代码。Go 1.5 版本开始， `GOMAXPROCS` 的默认值应该是您 Unix 机器的内核数。

如果您决定分配给 `GOMAXPROCS` 的值小于您的 Unix 机器的内核数的话，可能会影响您的程序的性能。然而，使用比可用内核大的 `GOMAXPROCS` 值也一定不会使您的程序运行的快。

您可用通过编程的方式来发现 `GOMAXPROCS` 环境变量的值，即相关代码。在下面的 `maxprocs.go` 程序中找到：

```
package main

import (
    "fmt"
    "runtime"
)

func getGOMAXPROCS() int {
    return runtime.GOMAXPROCS(0)
}

func main() {
    fmt.Printf("GOMAXPROCS:%d\n", getGOMAXPROCS())
}
```

在一台 Intel i7 处理器的机器上执行 `maxprocs.go` 产生如下输出：

```
$go run maxprocs.go
GOMAXPROCS: 8
```

然而，您可以在程序执行前，通过改变 `GOMAXPROCS` 环境变量的值来修改上面的输出。在 `bash(1)` Unix shell 里执行下面的命令：

```
$go version
go version go1.9.4 darwin/amd64
$export GOMAXPROCS=800; go run maxprocs.go
GOMAXPROCS: 800
$export GOMAXPROCS=4; go run maxprocs.go
GOMAXPROCS: 4
```

select 关键字

您很快就会了解到，`select` 关键字非常强大，它可以在各种情况下做很多事情。Go中的 `select` 语句看起来像 channels 的 `switch` 语句。实际上，这意味着 `select` 允许 goroutine 等待多个通信操作。因此，您从 `select` 获得的主要好处就是它使您能够使用一个 `select` 块处理多个 channels。因此，您可以在 channels 上进行非阻塞操作。

> 使用多 channels 和 `select` 关键字的最大问题是 死锁。就是说为了避免此类死锁，您在设计和开发过程中要格外的小心。

`select.go` 的代码将阐明 `select` 关键字的用法。这个程序分五部分来介绍。

`select.go` 的第一部分展示如下代码：

```
package main

import(
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

来自 `select.go` 的第二部分代码如下：

```
func gen(min, max int, createNumber chan int, end chan bool) {
    for {
        select {
            case createNumber <- rand.Intn(max-min) + min:
            case <- end:
                close(end)
                return
            case <- time.After(4 * time.Second):
                fmt.Println("\ntime.After()!")
        }
    }
}
```

那么，在 `select` 块中的代码真正发生了什么？这个特别的 `select` 语句有三种情况。注意 `select` 语句不需要 `default` 分支。您可以把这个 `select` 语句的第三个分支当作 `default` 分支。`time.After()` 函

数在指定时间过后返回，因此它将在其他 channels 被阻塞时解锁 `select` 语句。

`select` 语句不是按顺序计算的，因为所有的 channels 都是同时检查的。如果在 `select` 语句中没有 channels 是准备好的，那么 `select` 语句就会阻塞，直到有一个 channels 准备好。如果 `select` 语句中有多个 channels 准备好，那么 Go 运行时就会在这些准备好的 channles 中随机选择一个。Go 运行时在这些准备好的 channels 之间做随机选择时尽量做到一致和公平。

`select.go` 的第三部分如下：

```
func main() {
    rand.Seed(time.Now().Unix())
    createNumber := make(chan int)
    end := make(chan bool)

    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }
```

`select.go` 程序的第四部分包含如下代码：

```
    n, _ := strconv.Atoi(os.Args[1])
    fmt.Printf("Going to create %d random numbers.\n", n)
    go gen(0, 2*n, createNumber, end)

    for i := 0; i < n; i++ {
        fmt.Printf("%d ", <-createNumber)
    }
```

> *没有检查 `strconv.Atoi()` 返回的错误值是为节省一些空间。您在真实应用中不应该这样做*

`select.go` 程序的剩余代码如下：

```
    time.Sleep(5 * time.Second)
    fmt.Println("Exting...")
    end <- true
}
```

`time.Sleep(5 * time.Second)` 语句的主要目地是给 `gen()` 中的 `time.After()` 函数足够的时间返回，从而激活 `select` 语句中的相关分支。

`main()` 函数最后的一条语句是通过激活在 `gen()` 里的 `select` 语句中的 `case <-end` 分支来终止程序并执行相关代码。

执行 `select.go` 将产生如下输出：

```
$go run select.go 10
Going to create 10 random nubmers.
12 17 8 14 19 9 2 0 19 5
time.After()!
Exiting...
```

> *`select` 最大优点就是它可以连接，编排和管理多个 channels。当 channels 连接 goroutines 时，`select` 连接那些连接 goroutines 的 channels。因此，如果 `select` 语句不是 Go 并发模型中唯一重要的部分，也是之一。*

goroutine超时检查的两种方式

这节介绍两个非常重要的技巧来帮助您处理 goroutines 超时。简单说，这两个技巧帮您解决不得不一直等待一个 goroutine 完成它的工作，并且让您完全控制等待 goroutine 结束的时间。这两个技巧都使用方便的 `select` 关键字的功能并结合上节中使用过的 `time.After()` 函数。

方式1

第一个技巧的源码保存在 `timeOut1.go` 中，并分为四部分介绍。

`timeOut1.go` 的第一部分显示如下代码：

```
package main

import (
    "fmt"
    "time"
)
```

`timeOut1.go` 第二段代码如下：

```
func main() {
    c1 := make(chan string)
    go func () {
        time.Sleep(time.Second * 3)
        c <- "c1 OK"
    }()
```

`time.Sleep()` 调用用于模拟函数完成其工作通常需要的时间。在这个例子里，匿名函数以 goroutine 方式执行，它将在大约 3 秒（ `time.Second * 3` ）后写消息到 `c1` channel。

`timeOut1.go` 的第三段包含如下代码：

```
    select {
        case res := <- c1:
            fmt.Println(res)
        case <- time.After(time.Second * 1):
            fmt.Println("timeout c1")
    }
```

`time.After()` 函数调用的目地是等待指定时间。这个例子中，您不用关心 `time.After()` 返回的实际值，但实际上 `time.After()` 函数调用介绍意味着超时了。例子中由于 `time.After()` 函数指定的值小于上节以 goroutine 方式执行的 `time.Sleep()` 调用中使用的值，您极有可能得到一个超时信息。

`timeOut1.go` 的其余代码如下：

```
    c2 := make(chan string)
    go func() {
        time.Sleep(3 * time.Second)
        c2 <- "c2 OK"
    }()

    select {
        case res := <-c2:
            fmt.Println(res)
        case <- time.After(4 * time.Second):
            fmt.Println("timeout c2")
    }
}
```

上面这段代码执行一个 goroutine 并使用 `time.After(4 * time.Second)` 定义了一个 4 秒的超时期限，由于 `time.Sleep()` 的调用，这个 goroutine 将执行大约 3 秒。如果 `time.After(4 * time.Second)` 调用的返回在您从 `select` 块的第一个分支里的 c2 channel 获得值之后，那么就不会超时；否则，将超时！然而，在这个例子中， `time.After()` 调用的值提供了足够的时间对于 `time.Sleep()` 调用的返回，因此您极可能不会得到一个超时信息。

执行 `timeOut1.go` 产生如下输出：

```
$go run timeOut1.go
timeout c1
c2 OK
```

正如期望的，第一个 goroutine 没有完成它的工作，而第二哥有足够的时间去完成。

方式2

第二个技巧的源码保存在 `timeOut2.go` 中，并分为五部分来介绍。这次，超时时间作为命令行参数提供给程序。

`timeOut2.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)
```

`timeOut2.go` 的第二段代码如下：

```
func timeout(w *sync.WaitGroup, t time.Duration) bool {
    temp := make(chan int)
    go func() {
        time.Sleep(5 * time.Second)
        defer close(temp)

        w.Wait()
    }()

    select {
        case <-temp:
            return false
        case <-time.After(t):
            return true
    }
}
```

上面的代码里，`time.After()` 调用使用的时间周期是 `timeout()` 函数的一个参数，就是说它可变。再次，`select` 块实现了超时逻辑。另外，`w.Wait()` 调用使 `timeout()` 函数无期限的等待一个匹配的 `sync.Done()` 函数来结束。当 `w.Wait()` 调用返回，`select` 表达式的第一个分支就会执行。

`timeOut2.go` 的第三段代码如下：

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need a time duration!")
        return
    }

    var w sync.WaitGroup
    w.Add(1)
    t, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
```

`timeOut2.go` 的第四部分如下：

```
    duration := time.Duration(int32(t)) * time.Millisecond
    fmt.Printf("Timeout period is %s\n", duration)

    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
```

`time.Duration()` 函数转换一个整数值给一个之后使用到的 `time.Duration` 变量。

`timeOut2.go` 的其余代码如下：

```
    w.Done()
    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
}
```

一旦 `w.Done()` 调用执行，之前的 `timeout()` 函数将返回。然而第二次调用 `timeout()` 没有要等待的 `sync.Done()` 语句。

执行 `timeOut2.go` 产生如下输出：

```
$go run timeOut2.go 10000
Timeout period is 10s
Timed out!
OK!
```

执行 `timeOut2.go` 的超时周期比匿名 goroutine 的 `time.Sleep(5 * time.Second)` 长。然而，没有必要的 `w.Done()` 调用，匿名 goroutine 不能返回因为 `time.After()` 调用先结束了，所以第一个 `if` 表达式的 `timeout()` 函数返回 `true` 。第二个 `if` 表达式，匿名函数不必等待，因为 `time.Sleep(5 * time.Second)` 将先于 `time.After(t)` 结束，所以 `timeout()` 函数返回 `false`：

```
$go run timeOut2.go 100
Timeout period is 100ms
Timed out!
Timed out!
```

然而，第二次执行，这个超时周期太短了，所以这两个 `timeout()` 的执行都没有足够的时间完成，因此都超时了。

所以，当定义一个超时周期时，确保您选择了一个恰当的值，否则您的结果可能不是您期望的那样。

重温Channel（通道）

一旦掌握了 `select` 关键字，Go channels 可以以几种独特的方式做更多事要比您在第9章（并发-Goroutines,Channel和Pipeline）学到的。这节将揭晓 Go channels 的这些使用方法。

要记住 channel 类型的零值是 `nil`，并且如果您发送一个消息给以关闭的 channel，程序将崩溃。然而，如果您尝试从已关闭的 channel 读取的话，会得到 channel 类型的零值。因此，关闭 channel 后，您不能再往里写，但您能一直读。

为了能关闭 channel, channel 不必是只接受。另外，一个 `nil` channel 总是阻塞的。channels 的这个特性非常有用，当您想要禁用 `select` 表达式的一个分支时，可以分配一个 `nil` 值给一个 channel 变量。

最后，如果您要关闭一个 `nil` channel，程序就会崩溃。最好的说明是下面这个 `closeNilChannel.go` 程序：

```
package main

func main() {
    var c chan string
    close(c)
}
```

执行 `closeNilChannel.go` 产生如下输出：

```
$go run closeNilChannel.go
panic: close of nil channel

goroutine 1 [running]:
main.main()
    /Users/mtsouk/closeNilChannel.go:5 +0x2a
exit status 2
```

信号通道

信号通道 是一个仅用来发送信号的通道。简单说，当您想要通知别人一些事情时可以使用信号通道。信号通道不能用来传输数据。

> *您不能混淆信号通道和在第8章（Go UNIX系统编程）讨论的* ***Unix 信号处理***，*它们是完全不同的概念。*

在这章后面的 *指定 goroutines 的执行顺序* 一节，您将看到使用信号通道的代码例子。

可缓冲通道

这小节的主题是**可缓冲通道**。这些通道允许 Go 调度器快速把任务放入队列，为了能够处理更多的请求。而且，您可以使用缓冲通道作为 **信号量** 来限制整个应用程序。

这里介绍的技术工作如下： 所有进入的请求被转发到通道里，由它来逐个处理。当通道处理完一个请求后，它就发送消息给原来当调用者说它准备处理新当请求了。因此这个通道的缓冲能力限制它能保存的并发请求数。

这个技术用 `bufChannel.go` 中的代码来帮助介绍，分为四个部分。

`bufChannel.go` 的第一部分代码如下：

```
package main

import (
    "fmt"
)
```

`bufChannel.go` 的第二部分代码如下：

```
func main() {
    numbers := make(chan int, 5)
    counter := 10
```

这个 `numbers` 通道的定义给它提供了存储五个整数的空间。

`bufChannel.go` 的第三部分代码显示如下：

```
    for i:= 0; i < counter; i++ {
        select{
            case numbers <- i:
            default:
                fmt.Println("Not enough space for", i)
        }
    }
```

上面这段代码里，我们试图放 10 个整数到 `numbers` 通道。然而，由于 `numbers` 通道只有 5 个整数的空间，您就不能把 10 个整数都存入进去。

`bufChannel.go` 的其余代码如下：

```
    for i := 0; i < counter + 5; i++ {
        select {
            case num := <- numbers:
                fmt.Println(num)
            default:
                fmt.Println("Nothing more to be done!")
                break
        }
    }
}
```

上面的代码里，我们试着使用 `for` 循环和 `select` 表达式读取 `numbers` 通道中的内容。只要 `numbers` 通道里有内容可读， `select` 表达式的第一个分支就会执行。如果 `numbers` 通道是空的，这个 `default` 分支就会执行。

执行 `bufChannel.go` 产生如下输出：

```
$go run bufChannel.go
Not enough space for 5
Not enough space for 6
Not enough space for 7
Not enough space for 8
Not enough space for 9
0
1
2
3
4
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
```

值为nil的通道

这节，您将了解到 **值为 nil 的通道**。这些是通道的特殊类型，因为它们总是阻塞的。空通道的使用说明在 `nilChannel.go` 中， 分为四个代码片段来介绍。

`nilChannel.go` 的第一部分如下：

```
package main

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

`nilChannel.go` 的第二段代码显示如下：

```
func add(c chan int) {
    sum := 0
    t := time.NewTimer(time.Second)
    for {
        select {
            case input := <-c:
                sum = sum + input
            case <-t.c:
                c = nil
                fmt.Println(sum)
        }
    }
}
```

`add()` 函数展示了如何使用空通道。 `<-t.c` 表达式按在 `time.NewTimer()` 调用中指定的时间阻塞 `t` 计时器的 `c` 通道。不要混淆函数的参数 `c` 通道和 `t` 计时器的 `t.c` 通道。它会触发 `select` 表达式的相关分支的执行，给 `c` 通道分配 `nil` 值并打印 `sum` 变量。

`nilChannel.go` 的第三段代码如下：

```
func send(c chan int) {
    for {
        c <- rand.Intn(10)
    }
}
```

`send()` 函数的目的是产生随机数并持续发送它们到一个通道里，只要这个通道是打开的。

`nilChannel.go` 的其余代码如下：

```
func main() {
    c := make(chan int)
    go add(c)
    go send(c)
    time.Sleep(3 * time.Second)
}
```

使用 `time.Sleep()` 函数是为了给这俩个 goroutines 足够的操作时间。

执行 `nilChannel.go` 将产生如下输出：

```
$go run nilChannel.go
13167523
$go run nilChannel.go
12988362
```

由于执行 `add()` 函数中的 `select` 表达式的第一个分支的次数不固定，所以您执行 `nilChannel.go` 会得到不同的结果。

传送channel的通道

传送channel的通道 是一种特殊的通道变量，它与通道工作而不是其他类型的变量。不过，您仍然要给一个传送 channel 的通道声明一个数据类型。您可以在一行使用 `chan` 关键字两次定义一个传送 channel 的通道，如下所示：

```
c1 := make(chan chan int)
```

> 这章介绍的其他类型的通道都比传递channel的通道要简单方便。

用 `chSquare.go` 代码来说明传递channel的通道的使用，分为四个部分来介绍。

`chSquare.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "time"
)

var times int
```

`chSquare.go` 的第二段代码如下：

```
func f1(cc chan chan int, f chan bool) {
    c := make(chan int)
    cc <- c
    defer close(c)

    sum := 0
    select {
        case x := <-c:
            for i := 0; i <= x;  i++ {
                sum = sum + i
            }
            c <- sum
        case <-f:
            return
    }
}
```

声明一个常规的 `int` 通道，您把它发送给传递通道的通道变量。然后您使用 `select` 表达式从常规的 `int` 通道读取数据和使用 `f` 信号通道来退出函数。

一旦您从 `c` 通道读取了信号值，便开始了一个 `for` 循环来计算从 `0` 到信号值的所有整数之和。接下来，您发送这个计算值给 `c int` 通道，执行结束。

`chSquare.go` 的第三部分如下：

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need just one integer argument!")
        return
    }

    times, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    cc := make(chan chan int)
```

上面代理里的最后一句表达式声明了一个名为 `cc` 的传递通道的通道变量，它是这个程序的开始，因为一切都依赖与它： `cc` 变量传递给 `f1()` 函数，并被接下来的 `for` 循环中使用。

`chSquare.go` 的其他代码如下：

```
    for i := 1; i < times + 1; i++ {
        f := make(chan bool)
        go f1(cc, f)
        ch := <-cc
        ch <- i
        for sum := range ch {
            fmt.Print("Sum(", i, ")=", sum)
        }
        fmt.Println()
        time.Sleep(time.Second)
        close(f)
    }
}
```

`f` 通道是一个**信号通道**，用来当真正的工作完成时结束 goroutine。`ch := <-cc` 表达式允许您从传递通道的通道获得一个常规的通道，使用 `ch <-i` 发送一个 `int` 值给它。之后，您使用 `for` 循环从它读取数据。尽管 `f1()` 函数编写为发送一个值回来，但您也可以读取多个值。注意 `i` 的每个值都被不同的 goroutine 执行。

信号通道的类型可以是任何您想要的类型，包括上面用到的 `bool` 和下节要用到的 `struct{}` 类型的信号通道。一个 `struct{}` 信号通道的优点是不能发送数据给它，这能减少错误和错误的想法。

执行 `chSquare.go` 将产生如下输出：

```
$go run chSquare.go 4
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
$go run chSquare.go 6
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
Sum(5)=15
Sum(6)=21
```

指定通道的执行顺序

尽管您不应该对 goroutines 的执行顺序做任何假设，但当您需要控制这个顺序时是可以的。这小节介绍一个使用 **信号通道** 的技巧。

> 您可能会问为什么简单函数能够更容易的控制执行顺序，还要选择按顺序执行 goroutines。这个答案很简单：goroutines 能够操作并发并等待其他 goroutines 结束，而在普通的函数却不能！

说明这个主题的 Go 程序命名为 `defineOrder.go`，并分了五部分来介绍。

`defineOrder.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "time"
)

func A(a, b chan struct{}) {
    <-a
    fmt.Println("A()!")
    time.Sleep(time.Second)
    close(b)
}
```

`A()` 函数能被存储在 `a` 参数的通道中并阻塞。一旦在 `main` 函数中通道被解除阻塞，函数 `A()` 将开始工作。最后，它关闭 `b` 通道，解除在函数 `B()` 中的另个函数的阻塞。

`defineOrder.go` 的第二段代码如下：

```
func B(a, b chan struct{}) {
    <-a
    fmt.Println("B()!")
    close(b)
}
```

`B()` 中的逻辑和函数 `A()` 相同。这个函数被阻塞直到 `a` 通道关闭。然后它做自己的工作并关闭 `b` 通道。注意 `a` 和 `b` 通道值这个函数的参数名。

`defineOrder.go` 的第三段代码如下：

```
func C(a chan struct{}) {
    <-a
    fmt.Println("C()!")
}
```

`C()` 函数被阻塞并等待 `a` 通道关闭再开始工作。

`defineOrder.go` 的第四部分如下：

```
func main() {
    x := make(chan struct{})
    y := make(chan struct{})
    z := make(chan struct{})
```

这三个通道作为三个函数的参数。

`defineorder.go` 的最后一段如下：

```
    go C(z)
    go A(x, y)
    go C(z)
    go B(y, z)
    go C(z)

    close(x)
    time.Sleep(3 * time.Second)
}
```

执行 `defineOrder.go` 将产生期望的输出，即使 `C()` 函数被调用了多次：

```
$go run defineOrder.go

A()!
B()!
C()!
C()!
C()!
C()!
```

以 goroutines 调用 `C()` 函数多次会工作的很好，因为 `C()` 没有关闭任何通道。然而，如果您调用 `A()` 或 `B()` 多次的话，您可能会得到如下错误信息：

如您所见， `A()` 函数被调用了俩次。然而，由于 `A()` 函数关闭了一个通道，它的 goroutines 中的一个将发现通道已经关闭并产生一个崩溃。如果您调用 `B` 多次会得到同样的崩溃信息。

通过共享变量来共享内存

通过共享变量来共享内存 是 Unix 线程间彼此通信非常普通的方式。一个 **互斥** 变量是一个互相排斥变量的简称，它主要用于线程同步和在有多个写操作同时发生时保护共享数据。互斥工作类似于容量为一的 **缓冲通道**，它允许最多一个 goroutine 在任何给定时间访问共享变量。这就是说没有办法让俩个或更多 goroutines 同时去更新一个变量。

一个并发程序的 **关键部分** 是代码不能被所有进程，线程或如当前的所有 goroutines 同时执行。代码需要用互斥体保护起来。因此，识别代码的关键部分会使整个程序处理非常简单，您应该注意这个任务。

> *当两个关键部分使用相同的 `sync.Mutex` 或 `sync.RWMutex` 变量时，您不能把其中一个嵌入另一个。不惜任何代价一定要避免跨函数传播互斥体，因为那会很难看出您是否做了嵌套。*

下面两小节将说明 `sync.Mutex` 和 `sync.RWMutex` 类型的使用。

sync.Mutex 类型

`sync.Mutex` 类型是 Go 实现的一个互斥体。它的定义可以在 `sync` 目录的 `mutex.go` 文件中找到，内容如下：

`sync.Mutex` 类型的定义没有什么特别的。所有的工作都是由 `sync.Lock()` 和 `sync.Unlock()` 来做的，它们分别能加锁和解锁 `sync.Mutex` 互斥体。给互斥体上锁意味着没人可以操作它，直到使用 `sync.Unlock()` 函数解锁。

`mutex.go` 程序分为五部分介绍，来说明 `sync.Mutex` 类型的使用。

`mutex.go` 的第一段代码如下：

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    m sync.Mutex
    v1 int
)
```

`mutex.go` 的第二段代码如下：

```
func change(i int) {
    m.Lock()
    time.Sleep(time.Second)
    v1 = v1 + 1
    if v1 % 10 == 0 {
        v1 = v1 - 10*i
    }
    m.Unlock()
}
```

这个函数的关键部分是 `m.Lock()` 和 `m.Unlock()` 之间的代码。

`mutex.go` 的第三部分如下：

```
func read() int {
    m.Lock()
    a := v1
    m.Unlock()
    return a
}
```

同样，这个函数的关键部分也由 `m.Lock()` 和 `m.Unlock()` 表达式限定。

`mutex.go` 的第四部分代码如下：

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }

    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    var waitGroup sync.WaitGroup
```

`mutex.go` 的最后一段代码如下：

```
    fmt.Printf("%d ", read())
    for i := 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func(i int) {
            defer waitGroup.Done()
            change(i)
            fmt.Printf("-> %d", read())
        }(i)
    }

    waitGroup.Wait()
    fmt.Printf("-> %d\n", read())
}
```

执行 `mutex.go` 将产生如下输出：

如果您从 `change()` 函数移除 `m.Lock()` 和 `m.Unlock()` 表达式，这个程序会产生类似如下输出：

输出中发生改变的原因是所有 goroutines 同时去修改共享变量，这也是随机输出的主要原因。

忘记解锁mutex的后果

这节，您会看到如果您忘记解锁 `sync.Mutex` 的后果。您将使用 `forgetMutex.go` 代码来做这个，把它分两部分来介绍。

`forgetMutex.go` 第一段代码如下：

```
package main

import (
    "fmt"
    "sync"
)

var m sync.Mutex

func function() {
    m.Lock()
    fmt.Println("Locked!")
}
```

这个程序的所有问题是由于开发者忘记释放 `m sync.Mutex` 互斥体的锁导致的。然而，如果您的程序只调用 `function()` 一次，那一切正常！

`forgetMutex.go` 的第二段如下：

```
func main() {
    var w sync.WaitGroup

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    w.Wait()
}
```

`main()` 函数没有任何问题，它创建两个 goroutines 并等它们完成。

执行 `forgetMutex.go` 产生如下输出：

```
$go run forgetMutex.go
Locked!
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001209c)
    /usr/local/Cellar/go/1.9.4/libexec/src/runtime/sema.go:56 +0
sync.(*WaitGroup).Wait(0xc420012090)
    /usr/local/Cellar/go/1.9.4/libexec/src/sync/waitgroup.go:131
    /Users/mtsouk/forgetMutex.go:30 +0xb6

goroutine 5 [semacquire]:
sync.runtime_SemacquireMutex(0x115c6fc, 0x0)
    /usr/local/Cellar/go/1.9.4/libexec/src/runtime/sema.go:71 +0
sync.(*Mutex).Lock(0x115c6f8)
    /usr/local/Cellar/go/1.9.4/libexec/src/sync/mutex.go:134 +0x
    /Users/mtsouk/forgetMutex.go:11 +0x2d main.main.func1(0xc420
    /Users/mtsouk/forgetMutex.go:20 +0x48 created by main.main
    /Users/mtsouk/forgetMutex.go:18 +0x58 exit status 2
```

所以，忘记解锁 `sync.Mutex` 互斥体会产生崩溃即使是最简单的程序。这同样适用于您将在下一节中使用的 `sync.RWMutex` 类型的互斥锁。

sync.RWMutex类型

`sync.RWMutex` 类型是另外一种互斥体，它是 `sync.Mutex` 的改进版，在 `sync` 目录的 `rwmutex.go` 文件中定义如下：

```
type RWMutex struct {
    w           Mutex
    writerSem   uint32
    readerSem   uint32
    readerCount int32
    readerWait  int32
}
```

换句话说， `sync.RWMutex` 是基于 `sync.Mutex` 做了必要的添加和改进。

现在让我们来说说 `sync.RWMutex` 是如果改进 `sync.Mutex` 的。尽管使用 `sync.RWMutex` 互斥锁只允许有一个函数执行写操作，但您能有多个属于 `sync.RWMutex` 互斥锁的读取者。然而，有一件事您要注意：除非 `sync.RWMutex` 互斥锁的所有读取者解锁，您不能为写操作给它上锁，这也是为了实现多互斥锁读取而付出的代价。

`RLock()` 和 `RUnlock()` 是出于读取的目的帮您与 `sync.RWMutex` 互斥体工作的函数，它们分别用于上锁和解锁互斥体。当您为了写操作想要上锁和解锁一个 `sync.RWMutex` 互斥体时，用在 `sync.Mutex` 互斥体的 `Lock()` 和 `Unlock()` 函数仍适用。因此为了读操作 `RLock()` 函数应该与 `RUnlock()` 配对调用。最后，显而易见，您不应该修改在 `RLock()` 和 `RUnlock()` 代码块间的任何共享变量。

`rwMutex.go` 代码说明了 `sync.RWMutex` 类型的使用和通途。程序分六部分介绍，并且相同的函数有俩个稍有不同的版本。第一个为读操作使用 `sync.RWMutex` 互斥体，第二个为读操作使用 `sync.Mutex` 互斥体。这两个函数之间的不同表现会帮助您理解为了读取操作使用 `sync.RWMutex` 互斥体的好处更多。

`rwMutex.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "os"
    "sync"
    "time"
)

var Password = secret{password: "myPassword"}

type secret struct {
    RWM         sync.RWMutex
    M           sync.Mutex
    password    string
}
```

`secret` 结构有一个共享变量，一个 `sync.RWMutex` 互斥锁和一个 `sync.Mutex` 互斥锁。

`rwMutex.go` 的第二段代码如下：

```
func Change(c *secret, pass string) {
    c.RWM.Lock()
    fmt.Println("LChange")
    time.Sleep(10 * time.Second)
    c.password = pass
    c.RWM.Unlock()
}
```

`Change()` 函数修改共享变量，意味着您需要使用一个排他锁，这就是使用 `Lock()` 和 `Unlock()` 函数的原因。当做变更时离不开使用排他锁！

`rwMutex.go` 的第三部分如下：

```
func show(c *secret) string {
    c.RWM.RLock()
    fmt.Println("show")
    time.Sleep(3 * time.Second)
    defer c.RWM.RUnlock()
    return c.password
}
```

`show()` 函数使用 `RLock()` 和 `RUnlock()` 函数是因为它的关键部分是用来读取共享变量的。因此，尽管 goroutines 能够读取共享变量，但是在不使用 `Lock()` 和 `Unlock()` 函数的情况下，所有 goroutines 都不能修

改共享变量。但是，只要有人使用互斥锁读取共享变量，`Lock()` 函数就会被一直阻塞。

`rwMutex.go` 的第四段代码如下：

```
func showWithLock(c *secret) string {
    c.M.Lock()
    fmt.Println("showWithLock")
    time.Sleep(3 * time.Second)
    defer c.M.Unlock()
    return c.password
}
```

`showWithLock()` 函数和 `show()` 函数之间唯一的不同是 `showWithLock()` 函数为了读操作使用了排他锁，这意味着只有一个 `showWithLock()` 函数能读取 `secret` 结构体的 `password` 字段。

`rwMutex.go` 的第五段代码如下：

```
func main() {
    var showFunction = func(c *secret) string {return ""}
    if len(os.Args) != 2 {
        fmt.Println("Using sync.RWMutex!")
        showFunction = show
    } else {
        fmt.Println("Using sync.Mutex!")
        showFunction = showWithLock
    }

    var waitGroup sync.WaitGroup
    fmt.Println("Pass:", showFunction(&Password))
```

`rwMutex.go` 的其余代码如下：

```
    for i := 0 ; i < 15; i++ {
        waitGroup.Add(1)
        go func() {
            defer waitGroup.Done()
            fmt.Println("Go Pass:", showFunction(&Password))
        }()

        go func(){
            waitGroup.Add(1)
            defer waitGroup.Done()
            Change(&Password, "123456")
        }()
        waitGroup.Wait()
        fmt.Println("Pass:", showFunction(&Password))
    }
}
```

执行 `rwMutex.go` 两次并使用 `time(1)` 命令行工具测试这个程序的两个版本将产生如下输出：

```
$time go run rwMutex.go 10 >/dev/null

real 0m51.206s
user 0m0.130s
sys  0m0.074s
$time go run rwMutex.go >/dev/null


real 0m22.191s
user 0m0.135s
sys  0m0.071s
```

注意上面命令结尾处的 `> /dev/null` 是为了忽略这两个命令的输出。因此，使用 `sync.RWMutex` 互斥体的版本要比使用 `sync.Mutex` 的版本快很多。

通过goroutine共享内存

本主题的最后一小节说明了如何使用专用的goroutine共享数据。尽管，共享内存是线程间彼此通信的传统方法，但 Go 具有内置的同步功能，允许单个 goroutine 拥有共享数据。这意味着其他 goroutines 必须发送消息给这个拥有共享数据的单独 goroutine，这可以防止数据损坏。这样的 goroutine 叫 **监视器 goroutine**。这 Go 的术语中，这是通过通信来共享而不是通过共享来通信。

该技术通过使用 `monitor.go` 源文件来说明，分五部分来介绍。 `monitor.go` 程序使用监视器 goroutine 随机产生数字。

`monitor.go` 的第一部分如下：

```
package main

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

var readValue = make(chan int)
var writeValue = make(chan int)
```

`readValue` 管道用于读取随机数，而 `writeValue` 管道用于得到新随机数。

`monitor.go` 的第二段代码如下：

```
func set(newValue int) {
    writeValue <-newValue
}

func read() int {
    return <-readValue
}
```

`set()` 函数的目的是共享变量的值，而 `read()` 函数的目的是读取已保存变量的值。

`monitor.go` 程序的第三段代码如下：

```
func monitor() {
    var value int

    for {
        select {
            case newValue := <-writeValue:
                value = newValue
                fmt.Printf("%d", value)
            case readValue <- value:
        }
    }
}
```

这个程序的所有逻辑都在这个 `monitor()` 函数里了。最具体地说， `select` 语句编排了整个程序的操作。当您有读请求时， `read()` 函数试图从由 `monitor()` 函数控制的 `readValue` 管道读取数据。返回保存在 `value` 变量中的当前值。另一方面，当您想要修改存储值时，可以调用 `set()` 。往 `writeValue` 管道写数据也由 `select` 语句处理。结果就是，不使用 `monitor()` 函数没人可以处理 `value` 共享变量。

`monitor.go` 的第四部分代码如下：

```
func main() {
    if len(os.Args) != 2 {

        fmt.Println("Please give an integer!")
        return
    }
    n, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("Going to create %d random numbers.\n", n)
    rand.Seed(time.Now().Unix())
    go monitor()
```

`monitor.go` 的最后一段代码如下：

```
    var w sync.WaitGroup
    for r := 0; r < n; r++ {
        w.Add(1)
        go func() {
            defer w.Done()
            set(rand.Intn(10 * n))
        }()
    }
    w.Wait()
    fmt.Printf("\nLast value: %d\n", read())
}
```

执行 `monitor.go` 产生如下输出：

```
$go run monitor.go 20
Going to create 20 random numbers.
89 88 166 42 149 89 20 84 44 178 184 28 52 121 62 91 31c117 140
Last value: 106
$go run monitor.go 10
Going to create 10 random numbers.
30 16 66 70 65 45 31 57 62 26
Last value: 26
```

> *个人而言，我更喜欢使用监视器 goroutine 而不是传统的共享内存技术，因为使用监视器 goroutine 的实现更安全，更贴合 Go 的哲学思想*

竞争状态

数据竞争状态 是两个多个运行中的如线程和 goroutines 试图控制或修改共享的资源或程序变量。严格讲，数据竞争发生在当两个或多个指令访问同一个内存地址，它们中至少有一个在此执行写操作。

当运行或构建 Go 源文件时使用 `-race` 标志会开启 Go **竞争监视器**，它会使编译器创建一个典型的可执行文件的修改版。这个修改版可以记录所有对共享变量的访问以及发生的同步事件，包括调用 `sync.Mutex` 和 `sync.WaitGroup` 。分析相关事件后，竞争监视器打印一份报告来帮助您识别潜在问题，这样您就可以修正它们了。

请看下面的 Go 代码，它保存为 `racec.go` 。这个程序分三部分来介绍。

`raceC.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
)

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")

        os.Exit(1)
    }
    numberGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
```

`raceC.go` 的第二段代码如下：

```
    var waitGroup sync.WaitGroup
    var i int

    k := make(map[int]int)
    k[1] = 12
    for i = 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func() {
            defer waitGroup.Done()
            k[i] = i
        }()
    }
```

`raceC.go` 的其余代码如下：

```
    k[2] = 10
    waitGroup.Wait()
    fmt.Println("k = %v\n", k)
}
```

好像许多 goroutines 同时访问 `k` map 还不够，我们在调用 `sync.Wait()` 函数前再添加一个访问 `k` map 的语句。

如果您执行 `raceC.go`，会获得如下没有任何警告或错误信息的输出：

```
$go run raceC.go 10
k = map[int]int{7:10, 2:10, 10:10, 1:12}
$go run raceC.go 10
k = map[int]int{2:10, 10:10, 1:12, 8:8, 9:9}
$go run raceC.go 19
k = map[int]int{10:10, 1:12, 6:7, 7:7, 2:10}
```

如果您仅执行一次 `raceC.go`，当打印 `k` map 内容的时候尽管您没有得到期望的，但一切看起来正常。然而，多次执行 `raceC.go` 告诉我们这有些错误，主要是每次执行产生不同的输出。

如果我们决定使用 Go 竞争监视器分析 `raceC.go`，我们可以获得更多信息及意外输出：

竞争监视器发现两处数据竞争。每个都在它的输出里用 `WARNING: DATA RACE` 消息开头。

第一个 **数据竞争** 发生在 `main.main.func1()` 内，它由一个goroutine 执行的 `for` 循环调用。这的问题由 `Previous write` 消息表示。检查相关代码后，很容易看到实际问题是匿名函数没带参数，意思是在 `for` 循

环中使用的 `i` 值不同准确识别，因为这是个写操作，在 `for` 循环内不断改变。

第二处数据竞争信息是 `Write at 0x00c420074180 by goroutine 7` 。如果您阅读相关输出，会看到这个数据竞争是关于写操作的，并且至少有两个 goroutines 在执行。因为这两个 goroutine 有相同的名字（ `main.main.func1()` ），这表明我们谈论的是同一个 goroutine。这两个 goroutine 试图写同一个变量，这就是数据竞争状态！

> *Go 用 `main.main.func1()` 记号命名一个内部地匿名函数。如果您有不同地匿名函数，它们的名字同样会不同*

您可以问之际，现在为了修正这两个数居竞争引起的问题我能做什么？

好的，您可以重写 `raceC.go` 的 `main()` 函数如下：

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
        os.Exit(1)
    }
    numGR, err := srconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    var waitGroup syncWaitGroup
    var i int

    k := make(map[int]int)
    k[1] = 12

    for i = 0; i < numGR; i++ {
        waitGroup.Add(1)
        go runc(j int) {
            defer waitGroup.Done()
            aMutex.Lock()
            k[j] = j
            aMutex.Unlock()
        }(i)
    }

    waitGroup.Wait()
    k[2] = 10
    fmt.Printf("k = %#v\n", k)
}
```

`aMutex` 变量是一个定义在 `main()` 函数外的全局 `sync.Mutex` 变量，它可以在程序的任何地方访问到。尽管这不必要，不过有这样一个全局变量可以免得您总是把它传给函数。

把这个新版本 `raceC.go` 保存为 `noRaceC.go` 并执行它产生如下输出：

```
$go run noRaceC.go 10
k = map[int]int{1:1, 0:0, 5:5, 3:3, 6:6, 9:9, 2:10, 4:4, 7:7, 8:
```

用 Go 竞争监测器运行 `noRaceC.go` 产生如下输出：

```
$go run -race noRaceC.go 10
k = map[int]int{5:5, 7:7, 9:9, 1:1, 0:0, 4:4, 6:6, 8:8, 2:10, 3:
```

注意当访问 `k` map 时，您需要一个锁机制。如果您没有使用这样的机制并且只修改了由 goroutine 执行的匿名函数的实现的话，您会从 `go run noRaceC.go` 获得如下输出：

这个根本问题是显而易见的：`并发 map 写操作` 。

关于context包

`context` 包的主要用途是定义 `Context` 类型和支持 **取消**！是的，您没听错； `context` 包的主要用途是支持取消操作，因为有时为了某种原因，您想要废弃正在做的操作。然而，能提供一些关于您取消决定的额外信息是非常有帮助的。 `context` 包就能让您做到这点！

如果您浏览一下 `context` 包点源码，就会认识到它的实现是相当的简单——甚至 `Context` 类型的实现也非常简单，而 `context` 包是非常重要的。

> *`context` 包以前是作为外部 Go 包存在，但在 Go 1.7版本，它第一次作为标准 Go 包出现。因此，如果您有个较老的 Go 版本，再没先下载 `context` 包前，您不能跟着这节学习。*

`Context` 类型是一个有四个方法的接口，方法名为 `Deadline()` ， `Done()` ， `Err()` ， `Value()` 。好消息是您不需要实现 `Context` 接口的所有方法——您只需要使用如 `context.WithCancel()` 和 `context.WithTimeout()` 函数修改 `Context` 变量就行。

下面是使用 `context` 包的示例，用 `simpleContext.go` 文件源码，分六部分来介绍。

`simpleContext.go` 的第一部分代码如下：

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "time"
)
```

`simpleContext.go` 的第二部分如下：

```
func f1(t int) {
    c1 := context.Background()
    c1, cancel := context.WithCancel(c1)
    defer cancel()

    go func(){
        time.Sleep(4 * time.Second)
        cancel()
    }()
```

`f1()` 函数只需要一个时延参数，因为其他的都定义在函数里了。注意 `cancel` 变量的类型是 `context.CancelFunc` 。

您需要调用 `context.Background()` 函数来初始化一个空 `Context` 参数。 `context.WithCancel()` 函数使用一个存在的 `Context` 创建一个子类并执行取消操作。 `context.WithCancel()` 函数也会创建一个 `Done` 通道，如上面的代码所示当 `cancel()` 函数被调用时，或者当父 context 的 `Done` 通道关闭时，它会被关闭。

`simpleContext.go` 的第三部分包含 `f1()` 函数的其余部分：

```
    select {
        case <- c1.Done():
            fmt.Println("f1():", c1.Err())
            return
        case r := <-time.After(time.Duration(t) * time.Second):
            fmt.Println("f1():", r)
    }
    return
}
```

这里您看到了 `Context` 变量的 `Done()` 函数的使用。当这个函数被调用时，您有一个取消操作。 `Context.Done()` 的返回值是一个通道，否则您就不能在 `select` 语句中使用它了。

`simpleContext.go` 的第四部分如下：

```
func f2(t int) {
    c2 := context.Background()
    c2, cancel := context.WithTimeout(c2, time.Duration(t)*time.
    defer cancel()

    go func(){
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
        case <-c2.Done():
            fmt.Println("f2():", c2.Err())
            return
        case r := <-time.After(time.Duration(t)*time.Second):
            fmt.Println("f2():", r)
    }
    return
}
```

这部分展示了 `context.WithTimeout()` 函数的使用，它需要两个参数：`Context` 参数和 `time.Duration` 参数。当超时到达时，`cancel()` 函数自动调用。

`simpleContext.go` 的第五部分如下：

```
func f3(t int) {
    c3 := context.Background()
    deadline := time.Now().Add(time.Duration(2*t) * time.Second)
    c3, cancel := context.WithDeadline(c3, deadline)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
        case <-c3.Done():
            fmt.Println("f3():", c3.Err())
            return
        case r := <-time.After(time.Duration(t) * time.Second):
            fmt.Println("f3():", r)
    }
    return
}
```

上面的代码说明了 `context.WithDeadline()` 函数的使用，它需要两个参数： `Context` 变量和一个表示操作将要截止的时间。当期限到了， `cancel()` 函数自动调用。

`simpleContext.go` 的最后一段代码如下：

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a delay!")
        return
    }

    delay, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Delay:", delay)

    f1(delay)
    f2(delay)
    f3(delay)
}
```

执行 `simpleContext.go` 产生如下输出：

```
$go run simpleContext.go 4
Delay: 4
f1(): 2018-02-13 23:30:00.271587 +0200 EET m=+4.003435078
f2(): 2018-02-13 23:30:00.272678 +0200 EET m=+8.004706996
f3(): 2018-02-13 23:30:00.273738 +0200 EET m=+12.005937567
$go run simpleContext.go 10
Delay: 10
f1(): context canceled
f2(): context canceled
f3(): context canceled
```

输出较长的行是 `time.After()` 函数调用的返回值。它们代表程序正常操作。意味着如果该程序执行超时就会立刻被取消。

这与使用 `context` 包一样简单，因为介绍的代码没有对 `Context` 接口做任何重要的工作。不过，下节的 Go 代码将介绍一个更真实的例子。

context使用的高级示例

使用 `useContext.go` 程序代码将更好，更深入的说明 `context` 包的功能，该代码分为五部分来介绍。

这个例子中，您将创建一个快速响应的 HTTP 客户端，这是一个很常见的需求。事实上，几乎所有的 HTTP 客户端都支持这个功能，您将学习另一个 HTTP 请求超时的技巧，在第12章（Go网络编程基础）。

`useContext.go` 程序需要两个命令行参数：要连接的服务器 URL 和应该等待的延迟。如果该程序只有一个命名行参数，那么延迟将是 5 秒。

`useContext.go` 的第一段代码如下：

```
package main

import (
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    myUrl   string
    delay   int = 5
    w       sync.WaitGroup
)

type myData struct {
    r   *http.Response
    err error
}
```

`myUrl` 和 `delay` 都是全局变量，因此它们能从代码的任意位置访问。另外，有一个名为 `w` 的 `sync..WaitGroup` 变量也是全局的，还有一个名为 `myData` 的结构体用于把 web 服务器的响应和一个错误变量绑在一起。

`useContext.go` 的第二部分代码如下：

```
func connect(c context.Context) error {
    defer w.Done()
    data := make(chan myData, 1)
    tr := &http.Transport{}
    httpClient := &http.Client{Transport: tr}

    req, _ := http.NewRequest("GET", myUrl, nil)
```

上面的代码处理 HTTP 连接。

> 您会了解更多关于开发 web 服务器和客户端的内容，在第12章（Go网络编程基础）。

`useContext.go` 的第三段代码如下：

```
    go func() {
        response, err := httpClient.Do(req)
        if err != nil {
            fmt.Println(err)
            data <- myData{nil, err}
            return
        } else {
            pack := myData{response, err}
            data <- pack
        }
    }()
```

`useContext.go` 的第四段代码如下：

```
    select {
        case <-c.Done():
            tr.CancelRequest(req)
            <-data
            fmt.Println("The request was cancelled!")
            return c.Err()
        case ok := <-data:
            err := ok.err
            resp := ok.r
            if err != nil {
                fmt.Println("Error select:", err)
                return err
            }
            defer resp.Body.Close()

            realHTTPData, err := ioutil.ReadAll(resp.Body)
            if err != nil {
                fmt.Println("Error select:", err)
                return err
            }
            fmt.Println("Server Response: %s\n", realHTTPData)
    }
    return nil
}
```

`useContext.go` 的其余代码实现了 `main()` 函数，如下：

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need a URL and a delay!")
        return
    }

    myUrl = os.Args[1]
    if len(os.Args) == 3 {
        t. err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println(err)
            return
        }
        delay = t
    }

    fmt.Println("Delay:", delay)
    c := context.Background()
    c, cancel := contedxt.WithTimeout(c, time.Duration(delay)*ti
    defer cancel()

    fmt.Printf("Connecting to %s \n", myUrl)
    w.Add(1)
    go connect(c)
    w.Wait()
    fmt.Println("Exiting...")
}
```

`context.WithTimeout()` 方法定义了超时期限。`connect()` 函数作为一个 goroutine 执行，将会正常终止或 `cancel()` 函数执行时终止。

尽管没必要知道服务端的操作，但看一下 Go 版本的随机减速的 web 服务器也是好的；一个随机数生成器决定您的 web 服务器有多慢。`slowWWW.go` 的源码内容如下：

```
package main

import (
    "fmt"
    "math/rand"
    "net/http"
    "os"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func myHandler(w http.ResponseWriter, r *http.Request) {
    delay := random(0, 15)
    time.Sleep(time.Duration(delay) * time.Second)

    fmt.Fprintf(w, "Servring: %s\n", r.URL.Path)
    fmt.Fprintf(w, "Dealy: %d\n", delay)
    fmt.Printf("Served: %s\n", r.Host)
}

func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port nubmer: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", myHandler)
    err := http.ListenAndServer(PORT, nil)
    if err != nil {
        fmt.Println(err)
        os.Exit(10)
    }
}
```

如您所见，您不需要在 `slowWWW.go` 文件中使用 `context` 包，因为那是 web 客户端的工作，它会决定需要多长时间等待响应。

`myHandler()` 函数的代码负责 web 服务器的减速处理。如介绍的那样，延迟可以由 `random(0,15)` 函数在 0 到 14 秒随机产生。

如果您使用如 `wget(1)` 的工具访问 `slowWWW.go` 服务器的话，会收到如下输出：

```
$wget -qO- http://localhost:8001/
Serving: /
Delay: 4
$wget -qO- http://localhost:8001/
Serving: /
Delay: 13
```

这是因为 `wget(1)` 的默认超时时间比较大。

当 `slowWWW.go` 已经在另一个 Unix shell 里运行后，用方便的 `time(1)` 工具处理执行 `useContext.go` 的输出如下：

这个输出显示只有第三个命令确实从 HTTP 服务器获得了响应——第一和第二个命令都超时了！

工作池

通常讲，**工作池** 是一组分配了处理任务的线程。或多或少，Apache web 服务器是这样工作的：主进程接受所有传入请求，然后转发到工作进程以获得服务。一旦，一个工作进程完成工作，它就准备为新客户端提供服务。不过，在这有些不同，因为我们的工作池使用 goroutines 代替线程。另外，线程不能在服务完请求后经常销毁，因为结束线程和创建线程的代价太高，而 goroutines 在完成它的任务后就可以销毁。

很快您就会看到，Go 中的工作池用 **缓冲通道** 来实现的，因为它允许您限制同时执行的 goroutines 数量。

接下来的程序 `workPool.go` 分为五部分来介绍。这个程序实现一个简单的任务：它将处理整数并打印它们的平方，使用一个单独 goroutine 来服务每个请求。尽管 `workerPool.go` 比较简单，但是它基本可以作为实现更复杂任务的程序模板。

> 这是一个高级技巧，它能帮您在 Go 中使用 goroutines 创建服务进程来接收并服务多个客户端！

`workerPool.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

type Client struct {
    id int
    integer int
}

type Data struct {
    job Client
    square int
}
```

这里您可以看到一个技巧，使用 `Client` 结构来分配一个唯一标识给每个要处理的请求。 `Data` 结构用于把由程序产生实际结果的客户端数据组合起来。简单说， `Client` 结构持有每个请求的输入数据，而 `Data` 结构有请求的结果。

`workerPool.go` 的第二段代码如下：

```
var (
    size = 10
    clients = make(chan Client, size)
    data = make(chan Data, size)
)

func worker(w *sync.WaitGroup) {
    for c := range clients {
        squre := c.integer * c.integer
        output := Data{c, square}
        data <- output
        time.Sleep(time.Second)
    }
    w.Done()
}
```

上面的代码由两处有趣的地方。第一处是创建了三个全局变量。`clients` 和 `data` 缓冲通道分别用于获得新的客户端请求和写入结果。如果您想要程序运行的快些，您可以增加这个 `size` 参数的值。

第二处是 `worker()` 函数的实现，它读取 `clients` 通道来获得新的请求去服务。一旦处理完成，结果就会写入 `data` 通道。使用 `time.Sleep(time.Second)` 语句的延迟不是必要的，但它使您更好地了解生成的输出的打印方式。

`workerPool.go' 的第三部分包含如下代码：

```
func makeWP(n int) {
    var w sync.WaitGroup
    for i := 0; i < n; i++ {
        w.Add(1)
        go worker(&w)
    }
    w.Wait()
    close(data)
}

func create(n int) {
    for i := 0; i < n; i++ {
        c := Client{i, i}
        clients <- c
    }
    close(clients)
}
```

上面的代码实现了两个名为 `makeWP()` 和 `create()` 的函数。`makeWP()` 函数的目的是为了处理所有请求生成需要的 `worker()` goroutines。虽然 `w.Add(1)` 函数在 `makeWP()` 中调用，但 `w.Done()` 是在 `worker()` 函数里调用的，当 worker 完成它的任务时。

`create()` 函数的目的是使用 `Client` 类型恰当地创建所有的请求，然后把它们写入 `clients` 通道进行处理。注意 `clients` 通道是被 `worker()` 函数读取的。

`workerPool.go` 的第四部分代码如下：

```
func main() {
    fmt.Println("Capacity of clients:", cap(clients))
    fmt.Println("Capacity of data:", cap(data))

    if len(os.Args) != 3 {
        fmt.Println("Need #jobs and #workers!")
        os.Exit(1)
    }

    nJobs, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    nWorkers, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
```

在上面的代码里，您看到了读取命令行参数之前，会使用 `cap()` 函数获取通道的容量。

如果 worker 的数量大于 `clients` 缓冲通道容量，那么 goroutines 的数量将会增加到与 `clients` 通道的大小相同。简单讲，任务数量比 worker 大，任务将以较小的数量供应。

这个程序允许您使用命令行参数定义 worker 和 任务的数量。但为了修改 `clients` 和 `data` 通道的大小，您需要修改源代码。

`workerPool.go` 的其余部分如下：

```
    go create(nJobs)
    finished := make(chan interface{})
    go func() {
        for d := range data {
            fmt.Println("Client ID: %d\tint: ", d.job.id)
            fmt.Println("%dtsquare: %d\n", d.job.integer, d.squa
        }
        finished <- true
    }()
    makeWP(nWorkers)
    fmt.Println(": %v\n", <-finished)
}
```

首先，您调用 `create()` 函数模拟要处理的客户端请求。一个匿名 goroutine 用于读取 `data` 通道并打印输出到屏幕上。 `finished` 通道用于阻塞程序直到匿名 goroutine 读完 `data` 通道。因此，这个 `finished` 通道不需要指定类型！最后，您调用 `makeWP()` 函数来真正处理请求。 `fmt.Printf()` 语句块里的 `<-finished` 语句的意思是不允许程序结束直到有往 `finished` 通道里写数据。写数据的是 `main()` 函数中的匿名 goroutine。另外，虽然这个匿名函数往 `finished` 通道里写 `true` 值，但您也可以往里写 `false` 并同样解除 `main()` 函数的阻塞。您可以自己试一下！

执行 `workerPool.go` 产生如下输出：

```
$go run workerPool.go 15 5
Capacity of clients: 10
Capacity of data: 10
ClientID: 0     int: 0      square:0
ClientID: 4     int: 4      square:16
ClientID: 1     int: 1      square:1
ClientID: 3     int: 3      square:9
ClientID: 2     int: 2      square:4
ClientID: 5     int: 5      square:25
ClientID: 6     int: 6      square:36
ClientID: 7     int: 7      square:49
ClientID: 8     int: 8      square:64
ClientID: 9     int: 9      square:81
ClientID: 10    int: 10     square:100
ClientID: 11    int: 11     square:121
ClientID: 12    int: 12     square:144
ClientID: 13    int: 13     square:169
ClientID: 14    int: 14     square:196
: true
```

当您希望在 `main()` 函数中为每个单独的请求提供服务而不希望得到它的响应时，就像在 `workerpool.go` 中发生的那样，您需要担心的事情就很少了。在 `main()` 函数中既要使用 goroutines 处理请求，又要从它们获得响应，一个简单的办法是使用共享内存或者一个监视进程来搜集数据而不只是打印它们到屏幕上。

最后， `workerPool.go` 程序的工作是非常的简单，因为 `worker()` 函数不会失败。当您必须在计算机网络上工作或使用其他可能会失败的资源时，情况就不是这样了。

延展阅读

下面是非常有用的资源：

- 可以在https://golang.org/pkg/sync/查看 `sync` 包文档。
- 可以在https://golang.org/pkg/context/查看 `context` 包文档。
- 访问https://golang.org/src/runtime/proc.go了解更多关于 Go 调度器多实现。
- 可以在浏览 Go 调度器的设计文档。

练习

- 使用缓冲通道实现一个并发版本的 `wc(1)`。
- 接着，使用共享内存实现一个并发版本的 `wx(1)`。
- 最后，使用监视 goroutine 实现一个并发版本的 `wx(1)`。
- 修改 `workerPool.go` 的源码来保存结果到文件里。处理文件时，使用互斥锁和关键部分，或者一个监视 goroutine 来保证在磁盘上写入您的数据。
- 当 `workerPool.go` 的 `size` 全局变量为 `1` 时会发生什么？为什么？
- 修改 `workerPool.go` 的源码来实现 `wc(1)` 支持命令行功能。
- 修改 `workerPool.go` 的源码来实现 `clients` 和 `data` 缓冲通道的大小可以通过命令行参数来定义。
- 使用监视 goroutine 写一个并发版本的 `find(1)` 命令行工具。
- 修改 `simpleContext.go` 源码，把在 `f1()`，`f2()`，`f3()` 中使用的匿名函数改为一个单独的函数。这个改动的难点是什么？
- 修改 `simpleContext.go` 的源码，让 `f1()`，`f2()`，`f3()` 函数使用外部创建的 `Context` 变量而不是它们自己的。
- 修改 `useContext.go` 源码，使用 `context.WithDeadline()` 或者 `context.WithCancel()` 来代替 `context.WithTimeout()`。
- 最后，使用 `sync.Mutex` 互斥锁时间一个并发版本的 `find(1)` 命令行工具。

本章小结

这章解决了许多关于 goroutine 的重要主题。然而，主要的是阐明 `select` 语句的作用。另外是说明 `context` 标准包的使用。

由于 `select` 语句的功能，通道是非常好的 Go 方式来互连 Go 程序的组件。

并发程序又许多规则；但是，最重要的规则是您应该避免共享，除非有万不得已的情况。并发程序中，共享数据是所有 bug 的根本原因。

从这章您必须记住尽管共享内存是在同一个进程的线程之上交换数据的唯一方法，但 Go 提供了更好的方法给每个 goroutine 来通信。因此，在你的代码里使用共享内存前，考虑一下 Go 的语法。虽然如此，如果您真的不得不使用共享内存，您可能会考虑使用监视 goroutine 来代替。

下章的主要话题是代码测试，代码优化和代码分析。除了这些主题，您会了解到基准测试，交叉编译和找出隐藏代码。

下章的最后，您也会了解到怎样文档化您的代码并使用 `godoc` 工具生成 HTML 输出。

Go网络编程基础

在前一章中，我们讨论了使用 `benchmark` 函数对Go代码进行性能测试、如何为Go代码编写单元测试、示例函数的编写、交叉编译和Go代码的调优，以及如何生成Go代码的文档。

本章主要介绍Go语言的网络编程，包括如何创建Web应用程序，并使其可以在计算机网络或互联网上运行。在下一章则介绍如何开发TCP和UDP的应用程序。

为了顺利地完成本章和下一章的内容，本章还介绍一些关于HTTP协议、网络以及网络工作原理的知识。

在这一章中，将学习以下主题：

- TCP/IP简介及其重要性
- IPv4和IPv6相关协议
- 命令行工具**netcat**
- 在Go中实现**DNS**查询
- `net/http` 包简介
- `http.Response` 、 `http.Request` 和 `http.Transport` 结构简介
- 使用Go创建Web服务器
- 使用Go进行Web客户端编程
- 使用Go创建网站
- `http.NewServeMux` 类型介绍
- **Wireshark**和**tshark**介绍
- HTTP连接的超时处理（客户端和服务端）

对于本章介绍的一些底层知识相对独立，可以跳过学习——在需要的时候可以重新复习这些内容。

关于 net/http，net 和 http.RoundTripper

本章的核心是 net/http 包，它提供了用于开发强大的Web客户端和服务端的函数功能。在这个包中， http.Get() 和 https.Get() 方法作为客户端，可以用来发送HTTP和HTTPS请求，而 http.ListenAndServe() 函数可用于创建Web服务器，并且指定服务器监听的IP地址和TCP端口号，然后在该函数中处理传入的请求。

除了 net/http 包，我们还在本章介绍的一些程序中使用 net 包。在第13章*网络编程-构建服务器与客户端*中，将会详细的介绍 net 包的功能。

最后，接口 http.RoundTripper 可以使Go的元素能够很方便的拥有执行HTTP事务的能力。简单地说，这意味着Go元素可以为 http.Request 返回 http.Response 结构。稍后将详细介绍 http.Response 和 http.Request 结构。

`http.Response` 类型

在文件https://golang.org/src/net/http/response.go中可以找到 `http.Response` 结构的定义如下：

```
type Response struct {
    Status string // e.g. "200 OK"
    StatusCode int // e.g. 200
    Proto string // e.g. "HTTP/1.0"
    ProtoMajor int // e.g. 1
    ProtoMinor int // e.g. 0
    Header Header
    Body io.ReadCloser
    ContentLength int64
    TransferEncoding []string
    Close bool
    Uncompressed bool
    Trailer Header
    Request *Request
    TLS *tls.ConnectionState
}
```

这个复杂的 `http.Response` 类型可以用来表示HTTP请求的响应。这个结构的每个字段，都可以在以上的源文件找到更多的信息。在标准Go库中，大多数 `struct` 类型都是基于一个结构，以及与结构字段相关的函数操作。

`http.Request` 类型

源代码[https://golang.org/src/net/http/request.go](https://golang

> ```go
> type Request struct {
>     Method string
>     URL *url.URL
>     Proto string // "HTTP/1.0"
>     ProtoMajor int // 1
>     ProtoMinor int // 0
>     Header Header
>     Body io.ReadCloser
>     GetBody func() (io.ReadCloser, error)
>     ContentLength int64
>     TransferEncoding []string
>     Close bool
>     Host string
>     Form url.Values
>     PostForm url.Values
>     MultipartForm *multipart.Form
>     Trailer Header
>     RemoteAddr string
>     RequestURI string
>     TLS *tls.ConnectionState
>     Cancel <-chan struct{}
>     Response *Response
>     ctx context.Context
> }
>

`http.Transport` 类型

> ```go
> type Transport struct {
>     idleMu sync.Mutex
>     wantIdle bool
>     idleConn map[connectMethodKey][]*persistConn
>     idleConnCh map[connectMethodKey]chan *persistConn
>     idleLRU connLRU
>     reqMu sync.Mutex
>     reqCanceler map[*Request]func(error)
>     altMu sync.Mutex
>     altProto atomic.Value
>     Proxy func(*Request) (*url.URL, error)
>     DialContext func(ctx context.Context, network, addr string
>     Dial func(network, addr string) (net.Conn, error)
>     DialTLS func(network, addr string) (net.Conn, error)
>     TLSClientConfig *tls.Config
>     TLSHandshakeTimeout time.Duration
>     DisableKeepAlives bool
>     DisableCompression bool
>     MaxIdleConns int
>     MaxIdleConnsPerHost int
>     IdleConnTimeout time.Duration
>     ResponseHeaderTimeout time.Duration
>     ExpectContinueTimeout time.Duration
>     TLSNextProto map[string]func(authority string, c *tls.Conn
>     ProxyConnectHeader Header
>     MaxResponseHeaderBytes int64
>     nextProtoOnce sync.Once
>     h2transport *http2Transport
> }
>

如您所见， `http.Transport` 是一个包含大量字段的复杂结构。好消息是在编写HTTP相关程序时，并不需要经常使用 `http.Transport` 结构，并且在使用时不需要处理它的所有字段。

`http.Transport` 结构实现了 `http.RoundTripper` 接口，并且支持HTTP、HTTPS和HTTP代理的模式。不过 `http.Transport` 是一个低级别的结构，本章中使用的 `http.Client` 结构则是一个高级别的HTTP客户端实现。

关于TCP/IP

TCP/IP是保障互联网运行的一组基础协议。它的名字来自它最著名的两个协议：**TCP**协议和**IP**协议。

TCP协议的全称是**传输控制协议(Transmission Control Protocol)**。TCP软件使用称为TCP报文的数据段，在机器之间传输数据。TCP协议可靠得传输能力是其主要的特点，也就是说在不需要程序员进行额外编码的情况下，该协议可以确保发送的数据包送达接收方。如果数据包发送没有被确认，TCP协议将重新发送该数据包。也就是说，TCP协议可以用于建立连接、传输数据、发送确认和关闭连接。

当两台主机之间建立TCP连接时，这两台机器之间会创建一个与电话呼叫类似的全双工虚拟链路。这两台主机会通过不断地通信，以确保数据发送和接收正确。如果由于某种原因连接中断，两台主机都会向上层应用程序返回错误。

IP协议的全称是**Internet协议(Internet Protocol)**。IP协议本质上不是一个可靠的协议。IP协议通过封装通过TCP/IP网络传输的数据，根据IP地址将数据报文从源主机传送到目标主机。IP协议还具备将数据报文有效发送到目的地的寻址方法。尽管路由器作为专用设备执行IP路由转发，但每个TCP/IP设备都或多或少执行一些基本的路由转发。

UDP协议的全称是**用户数据报协议(User Datagram Protocol)**，该协议基于IP协议，因此不确保可靠传输。简而言之，UDP协议比TCP协议要简单，因为该协议从设计上就没有考虑可靠性。因此，UDP消息可能会丢失、重传或乱序。此外，UDP数据报文的到达速度可以比接收方的处理能力快。因此当性能比可靠性更重要时，更多使用UDP协议。

关于IPv4和IPv6

IP协议的第一个版本称为**IPv4**；为表示区分，最新版本的IP协议被称为**IPv6**。

当前IPv4协议的主要问题是IP地址即将耗尽，这也是创建IPv6协议的主要原因。IPv4协议地址耗尽的原因是一个IPv4地址只使用32位来表示，它可以表示出2的32次方（4294967296）个不同的IP地址。而IPv6使用128位定义一个地址。

IPv4地址的格式为 `10.20.32.245` （用点分隔的四段），而IPv6地址的格式为 `3fce:1706:4523:3:150:f8ff:fe21:56cf` （用冒号分隔的八段）。

命令行工具netcat

命令行工具 `nc(1)` ，全称 `netcat(1)` ，在测试TCP/IP的客户端和服务端时非常便利。本节将介绍它的一些常见用法。

```
> ```shell
> nc 10.10.1.123 1234
>
```

命令行工具 `nc(1)` 默认使用TCP协议。如果使用UDP协议，可以在执行 `nc(1)` 命令时携带 `-u` 参数。

如果要使用 `netcat(1)` 模拟服务器，可以携带 `-l` 参数， `netcat(1)` 将监听指定端口号的连接。

如果希望 `netcat(1)` 生成详细的输出，可以使用 `-v` 和 `-vv` 参数，这些输出对排查网络连接故障提供了很大的便利。

`netcat(1)` 不仅可以测试HTTP应用程序，在第13章*网络编程 - 构建服务器与客户端*中，它同样可以灵活的使用，用于开发TCP和UDP协议的客户机和服务器。在本章中后续的一个例子中， `netcat(1)` 将作为案例使用。

读取网络接口的配置文件

网络配置有四个核心元素：接口的IP地址、接口的网络掩码、主机的DNS服务器配置以及主机的默认网关或默认路由配置。这里存在一个问题：目前没有原生并可移植Go代码来获取以上信息。这表明在UNIX系统的主机上，没有可移植的代码来查询DNS配置和默认网关信息。

因此在本节中，将介绍如何使用Go代码读取UNIX系统的网络接口配置。为此，我将介绍两个可移植的程序，用于查询有关网络接口的配置。

第一个程序 `netConfig.go` 的源代码由三部分组成。

以下Go代码是 `netConfig.go` 的第一部分：

```
package main

import (
    "fmt"
    "net"
)

func main() {
    interfaces, err := net.Interfaces()
    if err != nil {
        fmt.Println(err)
        return
    }
```

函数 `net.Interfaces()` 的作用是将当前计算机的所有接口信息通过一个切片数据结构返回，切片的数据元素类型为 `net.Interface` 。此切片将用于获取接口的信息。

代码 `netConfig.go` 的第二个部分包含以下Go代码：

```
    for _, i := range interfaces {
        fmt.Printf("Interface: %v\n", i.Name)
        byName, err := net.InterfaceByName(i.Name)
        if err != nil {
            fmt.Println(err)
        }
```

在上面的代码中，使用 `net.Interface` 类型来访问切片的每个元素，获取所需的信息。

```
> ```go
>         addresses, err := byName.Addrs()
>         for k, v := range addresses {
>             fmt.Printf("Interface Address #%v: %v\n", k, v.Str
>         }
>         fmt.Println()
>     }
> }
>
```

在操作系统为macOS High Sierra的主机上，使用版本为1.10的Go运行环境执行 `netConfig.go` 会得到以下输出：

```
$ go run netConfig.go
Interface: lo0
Interface Address #0: 127.0.0.1/8
Interface Address #1: ::1/128
Interface Address #2: fe80::1/64

Interface: gif0

Interface: stf0

Interface: XHC20

Interface: en0
Interface Address #0: fe80::18fa:901a:ea9:eb5f/64
Interface Address #1: 192.168.1.200/24
Interface Address #2: 2a02:587:3006:b800:1cb8:bf1b:b154:4d0
Interface Address #3: 2a02:587:3006:b800:d84a:f0c:c932:35d1

Interface: en1

Interface: p2p0

Interface: awdl0

Interface: en2

Interface: en3

Interface: bridge0

Interface: utun0
Interface Address #0: fe80::2514:c3a3:ca83:e1c6/64

Interface: utun1
Interface Address #0: fe80::4e0b:a9a6:9abe:81a4/64

Interface: en5
Interface Address #0: fe80::1cb4:a29e:97bc:6fb5/64
Interface Address #1: 169.254.72.59/16
```

如上，因为现代计算机配置很多网络接口，同时程序支持IPv4和IPv6协议，`netConfig.go`返回了很多接口和IP地址信息。

在操作系统为Debian Linux的主机上，使用版本为1.7.4的Go运行环境执行`netConfig.go`会得到以下输出：

```
$ go run netConfig.go
Interface: lo
Interface Address #0: 127.0.0.1/8
Interface Address #1: ::1/128

Interface: dummy0

Interface: eth0
Interface Address #0: 10.74.193.253/24
Interface Address #1: 2a01:7e00::f03c:91ff:fe69:1381/64
Interface Address #2: fe80::f03c:91ff:fe69:1381/64

Interface: teql0

Interface: tunl0

Interface: gre0

Interface: gretap0

Interface: erspan0

Interface: ip_vti0

Interface: ip6_vti0

Interface: sit0

Interface: ip6tnl0

Interface: ip6gre0
```

可以看到有的网络接口没有显示网络地址，可能的主要原因是接口是关闭的，或者该接口没有进行配置。

> 并非所有列出的网络接口都关联了真正的硬件网络设备。最典型的例子是 `lo0` 接口，它是环回设备。环回设备是一种特殊的虚拟网络接口，主机可以通过该接口与自身通信。

下一个Go程序 `netCapabilities.go` 的代码也分为三部分。程序 `netCapabilities.go` 的目的是打印UNIX操作系统的主机上每个网络接口的功能。

程序 `netCapabilities.go` 使用结构 `net.Interface` 的字段，定义如下：

```
type Interface struct {
    Index int
    MTU int
    Name string
    HardwareAddr HardwareAddr
    Flags Flags
}
```

Go程序 `netCapabilities.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "net"
)
```

```
> ```go
> func main() {
>     interfaces, err := net.Interfaces()
>
>     if err != nil {
>         fmt.Print(err)
>         return
>     }
>
```

```
> ```go
>     for _, i := range interfaces {
>         fmt.Printf("Name: %v\n", i.Name)
>         fmt.Println("Interface Flags:", i.Flags.String())
>         fmt.Println("Interface MTU:", i.MTU)
>         fmt.Println("Interface Hardware Address:", i.HardwareA
>         fmt.Println()
>     }
> }
>
```

在操作系统为macOS High Sierra的主机上运行 `netCapabilities.go` 生成以下输出：

```
$ go run netCapabilities.go
Name: lo0
Interface Flags: up|loopback|multicast
Interface MTU: 16384
Interface Hardware Address:

Name: gif0
Interface Flags: pointtopoint|multicast
Interface MTU: 1280
Interface Hardware Address:

Name: stf0
Interface Flags: 0
Interface MTU: 1280
Interface Hardware Address:

Name: XHC20
Interface Flags: 0
Interface MTU: 0
Interface Hardware Address:

Name: en0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 98:5a:eb:d7:84:cd

Name: en1
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: d0:03:4b:cf:84:d3

Name: p2p0
Interface Flags: broadcast|multicast
Interface MTU: 2304
Interface Hardware Address: 02:03:4b:cf:84:d3

Name: awdl0
Interface Flags: broadcast|multicast
Interface MTU: 1484
Interface Hardware Address: 02:ac:d4:3b:d9:29

Name: en2
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 0a:00:00:a5:32:b0

Name: en3
Interface Flags: up|broadcast|multicast
```

```
Interface MTU: 1500
Interface Hardware Address: 0a:00:00:a5:32:b1

Name: bridge0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 0a:00:00:a5:32:b0

Name: utun0
Interface Flags: up|pointtopoint|multicast
Interface MTU: 2000
Interface Hardware Address:

Name: utun1
Interface Flags: up|pointtopoint|multicast
Interface MTU: 1380
Interface Hardware Address:

Name: en5
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 6e:72:e7:1b:cd:5f
```

在Debian Linux操作系统的主机上执行 `netCapabilities.go` 将输出类似的结果。

最后，如果对查找主机的默认网关感兴趣，可以在 `shell` 运行环境中执行 `netstat -nr` 命令查看，或在Go程序中使用 `exec.Command()` 执行该命令，并通过 `pipe` 或 `exec.CombinedOutput()` 以文本的形式获取其输出，并打印。然而，这种方式既不优雅也不完美。

实现DNS查询

DNS全称**Domain Name System（域名系统）**，它的作用是将IP地址转换为类似 `packt.com` 的域名，或者将域名转换为IP地址。本节中开发的 `DNS.go` 程序的处理逻辑非常简单：如果程序执行时的命令行参数是一个有效的IP地址，则程序将查询该IP地址对应的主机名；其他情况下，程序将假定它处理的是一个主机名，并将其转换成一个或多个IP地址。

程序 `DNS.go` 的代码将分三部分介绍。第一部分程序包含以下Go代码：

```
package main

import (
    "fmt"
    "net"
    "os"
)

func lookIP(address string) ([]string, error) {
    hosts, err := net.LookupAddr(address)
    if err != nil {
        return nil, err
    }
    return hosts, nil
}

func lookHostname(hostname string) ([]string, error) {
    IPs, err := net.LookupHost(hostname)
    if err != nil {
        return nil, err
    }
    return IPs, nil
}
```

函数 `lookIP()` 将一个IP地址作为输入，然后返回与该IP地址匹配的主机列表，这个功能通过函数 `net.LookupAddr()` 的帮助来实现。

而函数 `lookHostname()` 将主机名作为输入，使用 `net.LookupHost()` 函数进行处理，返回一个相关IP地址的列表。

程序 `DNS.go` 的第二部分是以下Go代码：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide an argument!")
        return
    }

    input := arguments[1]
    IPaddress := net.ParseIP(input)
```

函数 `net.ParseIP()` 可以将输入字符串解析为IPv4或IPv6地址。如果输入一个非法的IP地址，函数 `net.ParseIP()` 将返回 `nil` 。

程序 `DNS.go` 的剩余Go代码如下：

```
    if IPaddress == nil {
        IPs, err := lookHostname(input)
        if err == nil {
            for _, singleIP := range IPs {
                fmt.Println(singleIP)
            }
        }
    } else {
        hosts, err := lookIP(input)
        if err == nil {
            for _, hostname := range hosts {
                fmt.Println(hostname)
            }
        }
    }
}
```

执行 `DNS.go` 程序，并携带不同的输入参数，将生成以下输出：

```
$ go run DNS.go 127.0.0.1
localhost
$ go run DNS.go 192.168.1.1
cisco
$ go run DNS.go packtpub.com
83.166.169.231
$ go run DNS.go google.com
2a00:1450:4001:816::200e
216.58.210.14
$ go run DNS.go www.google.com
2a00:1450:4001:816::2004
216.58.214.36
$ go run DNS.go cnn.com
2a04:4e42::323
2a04:4e42:600::323
2a04:4e42:400::323
2a04:4e42:200::323
151.101.193.67
151.101.1.67
151.101.129.67
151.101.65.67
```

可以看到 `go run DNS.go 192.168.1.1` 命令的输出来自 `/etc/hosts` 文件，因为在 `/etc/hosts` 文件中配置了IP地址 `192.168.1.1` 的别名 `cisco` 。

最后一个命令的输出演示了域名（ `cnn.com` ）可能有多个公网IP地址映射。请特别注意*公网*这个词，尽管 `www.google.com` 有多个IP地址，但是只有地址（ `216.58.214.36` ）是公网IP地址。

获取域名的 NS 记录

一个 DNS 请求常见的作用是找到域名的 **名称服务**，它们被存储在域名的 **NS 记录**中。这个功能将会在 `NSrecords.go` 的代码中介绍。

`NSrecords.go` 的代码将由俩部分展示。第一部分如下：

```
package main

import(
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
```

在这部分，您要检查是否至少有一个命令行参数，以便有可以运行的参数。

`NSrecords.go` 的剩余代码如下：

```
    domain := arguments[1]
    NSs, err := net.LookupNS(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, NS := range NSs {
        fmt.Println(NS.Host)
    }
}
```

`net.LookupNS()` 函数做了所有的工作，它将域名的 **NS 记录**作为 `net.NS` 类型的 slice 变量返回。这是打印 slice 的每个 `net.NS` 元素的 `Host` 字段的原因。

执行 `NSrecords.go` 将产生如下输出：

```
$ go run NSrecords.go mtsoukalos.eu
ns5.linode.com.
ns4.linode.com.
ns1.linode.com.
ns2.linode.com.
ns3.linode.com.
$ go run NSrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

您可以用 `host(1)` 工具验证上面输出的正确性：

```
$ host -t ns www.mtsoukalos.eu
www.mtsoukalos.eu has no NS record
$ host -t ns mtsoukalos.eu
mtsoukalos.eu name server ns3.linode.com.
mtsoukalos.eu name server ns1.linode.com.
mtsoukalos.eu name server ns4.linode.com.
mtsoukalos.eu name server ns2.linode.com.
mtsoukalos.eu name server ns5.linode.com.
```

获取域名的 MX 记录

另一个 DNS 请求常见的作用是获取域名的 **MX** 记录。MX 记录指定域名的邮件服务。`MXrecords.go` 工具将用 Go 履行这一任务。

`MXrecords.go` 工具的第一部分展示在下面的 Go 代码中：

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
```

`MXrecords.go` 的第二部分包含如下 Go 代码：

```
    domain := arguments[1]
    MXs, err := net.LookupMX(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, MX := range MXs {
        fmt.Println(MX.Host)
    }
}
```

`MXrecords.go` 的工作方式与上节的 `NXrecords.go` 类似。唯一的不同是使用 `net.LookupMX()` 函数替代 `net.LookupNS()` 函数。

执行 `MXrecords.go` 将产生如下输出：

```
$ go run MXrecords.go golang.com
aspmx.1.google.com.
alt3.aspmx.1.google.com.
alt1.aspmx.1.google.com.
alt2.aspmx.1.google.com.
$ go run MXrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

再次，您可以在 `host(1)` 工具的帮助下验证前面输出的有效性：

```
$ host -t mx golang.com
golang.com mail is handled by 2 alt3.aspmx.1.google.com.
golang.com mail is handled by 1 aspmx.1.google.com.
golang.com mail is handled by 2 alt1.aspmx.1.google.com.
golang.com mail is handled by 2 alt2.aspmx.1.google.com.
$ host -t mx www.mtsoukalos.eu
www.mtsoukalos.eu has no MX record
```

Go实现web服务器

Go 允许您使用它的标准库函数自己实现一个 web 服务器。在这本书中您第一次看到一个用 Go 实现的 web 服务器应用程序是在第10章，Go 并发-进阶讨论，我们讨论 `context` 包的时候。

> *尽管用 Go 实现的 web 服务器能做需要高效安全的事情，但如果您真正需要一个强大的 web 服务器，支持模块，多站点和虚拟主机的话，您最好使用如 **Apache** 或 **Nginx** 这样的 web 服务器*

这个例子程序命名为 `www.go`，由五部分展示。它的第一部分包含期望的 `import` 声明：

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)
```

对于 web 服务器不需要操作 `time` 包。然而，由于服务器要发送时间和日期给客户端，所以这个例子里需要它。

`www.go` 的第二段代码如下：

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

这是该程序的第一个处理函数的实现。一个处理函数取决于使用的配置可以服务一个或多个 URL。

`www.go` 的第三部分包含如下代码：

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Fprintf(w, "Served time for: %s\n", r.Host)
}
```

从上面这段 Go 代码，您能看到该程序的第二哥处理函数的实现。该函数输出动态内容。

我们的 web 服务器代码的第四部分处理命令行参数和定义支持的 URL：

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    }else{
        PORT = ":" + arguments[1]
    }
    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/", myHandler)
```

`http.HandleFunc()` 函数把一个URL和一个处理函数关联起来。

`www.go` 程序的最后一段如下：

```
    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

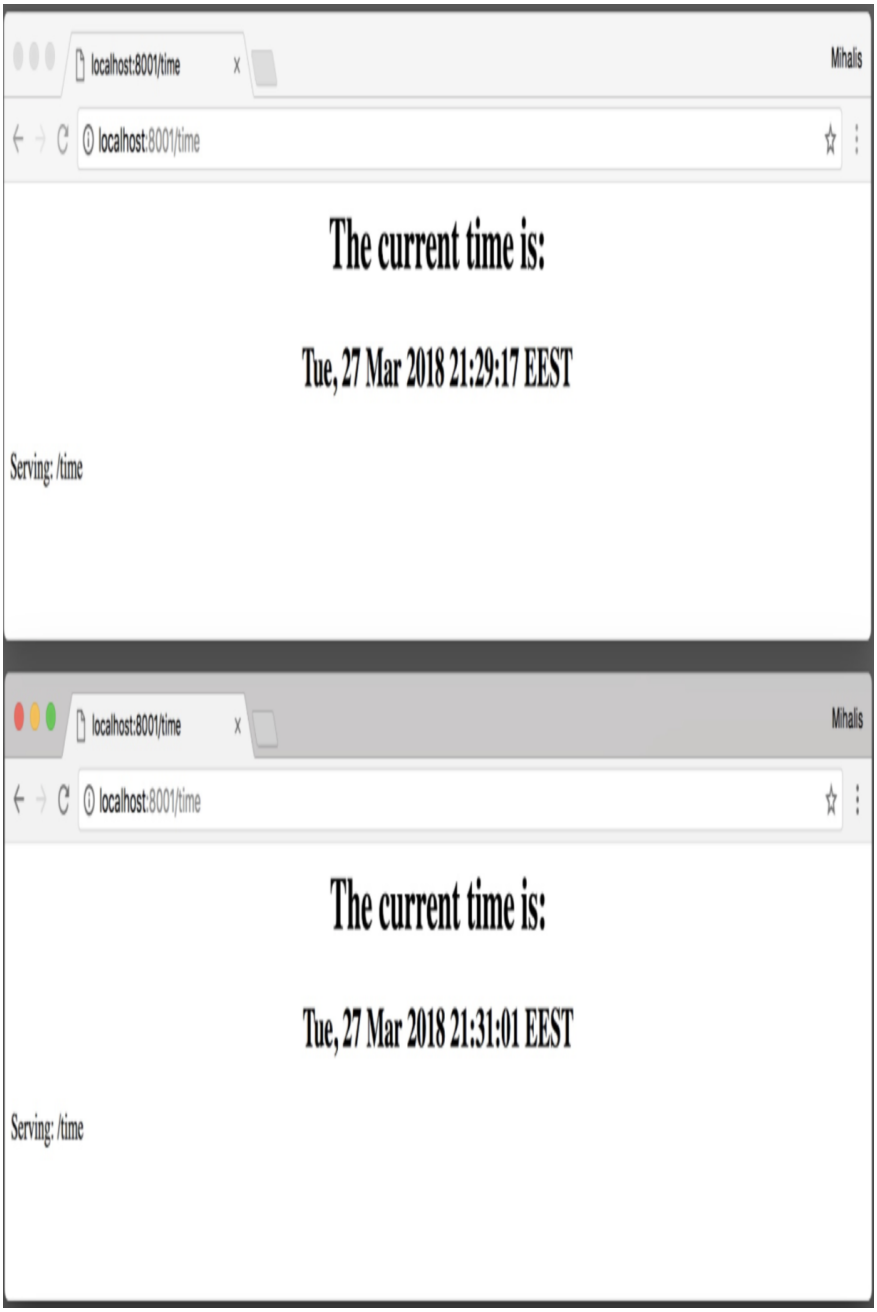
您将在 `http.ListenAndServe()` 函数的帮助下使用期望的端口号启动 web 服务器。

执行 `www.go` 并连接 web 服务器将生成如下输出：

```
$ go run www.go
Using default port number: :8001
Served: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served: localhost:8001
Served: localhost:8001
```

尽管这个程序的输出提供了一些方便的信息，但我想您更愿意用您喜欢的浏览器看到真正的输出。下面的截图显示了我们的 web 服务器的 `myhandler()` 函数的输出，显示在 **Google Chrome:**

下面的截图显示 `www.go` 也能生成动态页面。在这个例子，它是注册在 `/time` 下的 web 页面，显示当前时间和日期：



这里真正有趣的是除了 `/time` 所有的 URL 都由 `myHandler()` 函数处理，因为它是 `/` 的第一个参数，匹配所有没有与另一个 handler 匹配的 URL。

分析 HTTP 服务

如您在第11章（代码测试，优化和分析）中了解到的，有个名为 `net/http/pprof` 标准包，当您想要用它自己的 HTTP 服务分析一个 Go 应用时，可以使用它。因此，导入 `net/http/pprof` 将在 `/debug/pprof/` URL 下安装各种处理程序。您很快就会看到更多的情况。现在，您记住 `net/http/pprof` 包应该用于用 HTTP 服务分析 web 应用就足够了，而 `runtime/pprof` 标准包应该用于分析其他类型的应用。

注意，如果您的分析器工作在 `http://localhost:8080` 地址，您将从 如下 web 链接获得支持：

- http://localhost:8080/debug/pprof/goroutine
- http://localhost:8080/debug/pprof/heap
- http://localhost:8080/debug/pprof/theadcreate
- http://localhost:8080/debug/pprof/block
- http://localhost:8080/debug/pprof/mutex
- http://localhost:8080/debug/pprof/profile
- http://localhost:8080/debug/pprof/trace?seconds=5

下面被展示的程序将使用 `www.go` 做为起始点，并添加必要的 Go 代码允许您去分析它。这个新程序命名为 `wwwProfile.go` ，分为四部分展示。

注意， `wwwProfile.go` 将使用一个 `http.NewServeMux` 变量用于注册程序支持的路径。这样做的主要原因是，使用 `http.NewServeMux` 需要手动定义 HTTP 端点。如果您决定不使用 `http.NewServeMux` ，那么 HTTP 端点将自动注册，这也意味着您将使用 `_` 字符在引入的 `net/http/pprof` 包前。

`wwwProfile.go` 第一部分包含的 Go 代码如下：

```
package main

import (
    "fmt"
    "net/http"
    "net/http/pprof"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

这俩个处理函数的实现与之前的一样。

`wwwProfile.go` 的第二段代码如下：

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    }else{
        PORT = ":" + arguments[1]
        fmt.Println("Using port number: ", PORT)
    }

    r := http.NewServeMux()
    r.HandleFunc("/time", timeHandler)
    r.HandleFunc("/", myHandler)
```

在上面的 Go 代码中，您使用 `http.NewServeMux()` 和 `HandleFunc()` 定义您的 web 服务支持的 URL。

`wwwProfile.go` 的第三段代码如下：

```
    r.HandleFunc("/debug/pprof/", pprof.Index)
    r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
    r.HandleFunc("/debug/pprof/profile", pprof.Profile)
    r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
    r.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

上面的 Go 代码定义与分析相关的 HTTP 端点。没有它们，您将不能分析您的 web 应用。

余下的 Go 代码如下：

```
    err := http.ListenAndServe(PORT, r)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

这段代码启动 web 服务，并允许它服务来自 HTTP 客户端的连接。您会注意到 `http.ListenAndServe()` 的第二个参数不再是 `nil` 。

如您所见， `wwwProfile.go` 没有定义 `/debug/pprof/goroutine` HTTP 端点，这意味着 `wwwProfile.go` 不使用任何 goroutines！

执行 `wwwProfile.go` 将产生如下输出：

```
$ go run wwwProfile.go 1234
Using port number: :1234
Served time for: localhost:1234
```

使用 Go 分析器获取数据是相当简单的任务，执行下面的命令将带您自动进入 Go 分析器的 shell。

如您在第11章（代码测试，优化和分析）中了解到的，您现在可以使用这个分析数据，并使用 `go tool pprof` 分析它。

> *您可以访问 `http://HOSTNAME:PORTNUMBER/debug/pprof` ，从那可以看到分析结果。当 `HOSTNAME` 的值是 `localhost` ， `PORTNUMBER` 的值是 `1234` 时，您应该访问 `http://localhost:1234/debug/pprof/` 。*

您应该希望测试您的 web 服务应用的性能，您可以使用 `ab(1)` 工具，它做为 **Apache HTTP server benchmarking tool** 广为人知，用于创建流量和基准。这也允许 `go tool pprof` 收集更多的准确数据，如下：

用 Go 创建网站

您还记得第四章（组合类型的使用）中的 `keyValue.go` 应用，和第八章（Go UNIX系统编程）中的 `kvSaveLoad.go` 吗？在这节，您将学习使用 Go 标准库给它们创建 web 接口。这个新的 Go 源代码文件命名为 `kvWeb.go`，并分为6部分展示。

这个 `kvWeb.go` 和之前的 `www.go` 的第一个不同是，`kvWeb.go` 使用 `http.NewServeMux` 来处理 HTTP 请求，因为对于稍复杂点的 web 应用来说，它有更多的功能。

`kvWeb.go` 的第一部分如下：

```
package main

import (
    "encoding/gob"
    "fmt"
    "html/template"
    "net/http"
    "os"
)

type myElement struct {
    Name    string
    Surname string
    Id      string
}

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

在第八章（Go UNIX系统编程）中，您已经在 `kvSaveLoad.go` 见过上面的代码了。

`kvWeb.go` 的第二部分代码如下：

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }
    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}
```

```
func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Println("key: %s value: %v\n", k, d)
    }
}
```

您应该也熟悉上面这段代码，因为它曾第一次出现在第八章的 `kvSaveLoad.go` 中。

`kvWeb.go` 的第三部分如下：

```
func homePage(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Serving", r.Host, "for", r.URL.Path)
    myT := template.Must(template.ParseGlob("home.gohtml"))
    myT.ExecuteTemplate(w, "home.gohtml", nil)
}

func listAll(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Listing the contents of the KV store!")
    fmt.Fprintf(w, "<a href=\"/\" style=\"margin-right: 20px;\">
    fmt.Fprintf(w, "<a href=\"/list\" style=\"margin-right: 20px
    fmt.Fprintf(w, "<a href=\"/change\" style=\"margin-right: 20
    fmt.Fprintf(w, "<a href=\"/insert\" style=\"margin-right: 20

    fmt.Fprintf(w, "<h1>The contents of the KV store are:</h1>")
    fmt.Fprintf(w, "<ul>")
    for k, v := range DATA {
        fmt.Fprintf(w, "<li>")
        fmt.Fprintf(w, "<strong>%s</strong> with value: %v\n", k
        fmt.Fprintf(w, "</li>")
    }
    fmt.Fprintf(w, "</ul>")
}
```

`listAll()` 函数没有使用任何 Go 模版来生成动态输出，而是使用 Go 动态生成的。您可以把这当作一个例外，因为 web 应用通常使用 HTML 模版和 `html/templates` 标准库比较好。

`kvWeb.go` 的第四部分包含如下代码：

```
func changeElement(w http.ResponseWriter, r *http.Request){
    fmt.Println("Change an element of the KV store!")
    tmpl := template.Must(template.ParseFiles("update.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:    r.FormValue("name"),
        Surname: r.FormValue("surname"),
        Id:      r.FormValue("id"),
    }

    if !CHANGE(key, n) {
        fmt.Println("Update operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct {Struct bool}{true})
    }
}
```

从上面这段代码，您能看到在 `FormValue()` 函数的帮助下怎么读取一个 HTML 表单中字段的值。这个 `template.Must()` 函数是一个帮助函数，用于确保提供的模版文件不包含错误。

`kvWeb.go` 的第五段代码如下：

```
func insertElement(w http.ResponseWriter, r *http.Request){
    fmt.Println("Inserting an element to the KV store!")
    tmpl := template.Must(template.ParseFiles("insert.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement {
        Name:    r.FormValue("name"),
        Surname: r.FormValue("surname"),
        Id:      r.FormValue("id"),
    }

    if !ADD(key, n) {
        fmt.Println("Add operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{Success bool}{true})
    }
}
```

余下代码如下：

```
func main() {
    err := load()
    if err != nil {
        fmt.Println(err)
    }

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", homePage)
    http.HandleFunc("/change", changeElement)
    http.HandleFunc("/list", listAll)
    http.HandleFunc("/insert", insertElement)
    err = http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
    }
}
```

这个 `kvWeb.go` 的 `main()` 函数要比第八章（Go UNIX系统编程）的 `kvSaveLoad.go` 的 `main()` 函数简单，因为这俩个程序有完全不同的设计。

现在看一下 `gohtml` 文件，这个工程使用的起始文件是如下的 `home.gohtml`：

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</
<a href="/change" style="margin-right: 20px;">Change an elements
<a href="/insert" style="margin-right: 20px;">Insert an elements

<h2>Welcome to the Go KV store!</h2>
</body>
</html>
```

这个 `home.gohtml` 文件是静态的，就是说它的内容没有改变。而，其他的 `gohtml` 文件是动态显示信息。

`update.gohtml` 的内容如下：

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</
<a href="/change" style="margin-right: 20px;">Change an elements
<a href="/insert" style="margin-right: 20px;">Insert an elements

 <h1>Element updated!<h1>
<h1>Please fill in the fields:</h1>
    <form method="POST">
        <label>Key:</label><br/>
        <input type="text" name="key"><br/>
        <label>Name:</label><br/>
        <input type="text" name="name"><br/>
        <label>Surname:</label><br/>
        <input type="text" name="surname"><br/>
        <label>Id:</label><br/>
        <input type="text" name="id"><br/>
        <input type="submit">
    </form>


</body>
</html>
```

上面是主要的 HTML 代码。它最有趣的部分是 `if` 声明，它定义了您应该看到表单还是 `Element updated!` 信息。

最后， `insert.gohtml` 的内容如下：

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</
<a href="/change" style="margin-right: 20px;">Change an elements
<a href="/insert" style="margin-right: 20px;">Insert an elements


    <h1>Element inserted!</h1>

    <h1>Please fill in the fields:</h1>
    <form method="POST">
        <label>Key:</label><br/>
        <input type="text" name="key"><br/>
        <label>Name:</label><br/>
        <input type="text" name="name"><br/>
        <label>Surname:</label><br/>
        <input type="text" name="surname"><br/>
        <label>Id:</label><br/>
        <input type="text" name="id"><br/>
        <input type="submit">
    </form>


</body>
</html>
```

您能从 `<title>` 标签的内容看出， `insert.gohtml` 和 `update.gohtml` 是完全相同的！

在 Unix 命令行中执行 `kvWeb.go` 将产生如下输出：

```
$ go run kvWeb.go
Loading /tmp/dataFile.gob
Using default port number:  :8001
Serving localhost:8001 for /
Serving localhost:8001 for /favicon.ico
Listing the contents of the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Add operation failed!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Saving /tmp/dataFile.gob
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Changing an element of the KV store!
Serving localhost:8001 for /vavicon.ico
```

另外，真正有趣的是您可以用 web 浏览器和 `kvWeb.go` 交互。这个网站的首页，定义在 `home.gohtml` 显示截屏如下：

下面的截屏显示的是 key-value 存储的内容：

下面的截屏显示的是使用 `kvWeb.go` 应用的 web 接口添加新的数据到 key-value 存储中。

下面的截屏显示的是使用 `kvWeb.go` 应用的 web 接口更新已存在 key 数据的值。

这个 `kvWeb.go` web 应用还不够完美，所以把它作为一个练习尽可能完善它。

> 这节说明了您如何使用 Go 开放整个网址和 web 应用。尽管您的需求无疑是多样的，但是和 `kvWeb.go` 使用的技术是相同的。注意，自己做的网站被认为要比那些由流行的管理系统创建的更安全。

追踪 HTTP

Go 支持用 `net/http/httptraces` 标准包**追踪 HTTP**！这个包允许您追踪一个 HTTP 请求的解析。 `net/http/httptraces` 标准包的使用将在 `httpTrace.go` 中加以说明，它分为五部分进行讲解。

`httpTrace.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptrace"
    "os"
)
```

如您所料，您需要引入 `net/http/httptrace` 包来追踪 HTTP。

`httpTrace.go` 的第二部分代码如下：

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: URL\n")
        return
    }
    URL := os.Args[1]
    client := http.Client{}
```

在这部分，我们读取命令行参数并创建一个新的 `http.Client` 变量。由于这是您第一次在执行中看到 `http.Client` 对象，所以关于它我们就多讲一些。 `http.Client` 对象发送请求到服务器并获得一个响应。它的 `Transport` 字段允许您设置各种 HTTP 细节来替代默认值。

注意，您不应该在发布的软件中使用 `http.Client` 对象的默认值，因为这些值没有指定请求超时，它会影响程序的性能以及 goroutines的运行。而且，根据设计 `http.Client` 对象能够安全的用于并发程序。

`httpTrace.go` 的第三段代码如下：

```
    req, _ := http.NewRequest("GET", URL, nil)
    trace := &httptrace.ClientTrace{
        GotFirstResponseByte: func() {
            fmt.Println("First response byte!")
        },
        GotConn: func(connInfo httptrace.GotConnInfo) {
            fmt.Printf("Got Conn: %+v\n", connInfo)
        },
        DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
            fmt.Printf("DNS Info: %+v\n", dnsInfo)
        },
        ConnectStart: func(network, addr string) {
            fmt.Println("Dial start")
        },
        ConnectDone: func(network, addr string, err error) {
            fmt.Println("Dial done")
        },
        WroteHeaders: func() {
            fmt.Println("Wrote headers")
        },
    }
```

上面这段就是全部追踪 HTTP 请求的代码了。`httptrace.ClientTrace` 对象定义了使我们感兴趣的事件。当一个事件发生时，相关代码就会被执行。您能够在 `net/http/httptrace` 包的文档中找到更多信息，关于支持的事件和它们的目的。

`httpTrace.go` 的第四部分如下：

```
    req = req.WithContext(httptrace.WithClientTrace(req.Context(
    fmt.Println("Requesting data from server!")
    _, err := http.DefaultTransport.RoundTrip(req)
    if err != nil {
        fmt.Println(err)
        return
    }
```

`httptrace.WithClientTrace()` 函数基于给定的父上下文返回一个新的上下文；而 `http.DefaultTransport.RoundTrip()` 方法包裹 `http.DefaultTransport.RoundTrip()` 目的是告诉它要追踪当前的请求。注意，Go HTTP 跟踪被设计为追踪单个 `http.Transport.RoundTrip` 的事件。然而，当服务一个单独的HTTP 请求时，由于您可能有多个 URL 重定向，所以您需要能够识别出当前的请求。

`httpTrace.go` 的剩余代码如下：

```
    response, err := client.Do(req)
    if err != nil {
        fmt.Println(err)
        return
    }
    io.Copy(os.Stdout, response.Body)
}
```

最后这部分是使用 `Do()` 执行实际的请求到 web 服务器，获取 HTTP 数据，并把它显示在屏幕上。

执行 `httpTrace.go` 将产生如下非常翔实的输出：

```
$ go run httpTrace.go http://localhost:8001/
Requesting data from server!
DNS Info: {Addrs: [{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil
Dial start
Dial done
Got Conn: {Conn:0xc42014200 Reused:false WasIdel:false IdleTime:
Wrote headers
First response byte!
DNS Info: {Addrs: [{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil
Dial start
Dial done
Got Conn: {Conn:0xc420142008 Reused:false WasIdle:false IdleTime
Wrote headers
First response byte!
Serving: /
```

考虑到 `httpTrace.go` 打印从 HTTP 服务器返回的全部 HTML 数据，当您用一个真实的 web 服务测试时，您可能会得到太多输出，所以在这我们就使用 `www.go` 这个 web 服务器了。

> *如果您有时间看 `net/http/httptrace` 包的源码，在 http://golang.org/src/net/http/httptrace/trace.go，您会立马意识到 `net/http/httptrace` 是个相当低级的包，它使用 `context`，`reflect` 和 `internal/nettrace` 包实现它的功能。*

测试 HTTP handler

在这节我们将学习如果用 Go 测试 HTTP handlers，这是一个用 Go 测试的特别例子。我们将已 `www.go` 的代码为基础，修改需要调整的地方。

新版本的 `www.go` 命名为 `testWWW.go`，并分为三部分展示。`testWWW.go` 的第一部分代码如下：

```
package main

import (
    "fmt"
    "net/http"
    "os"
)

func CheckStatusOK(w http.ResponseWriter, r *http.Request){
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, `Fine!`)
}
```

`testWWW.go` 的第二部分代码如下：

```
func StatusNotFound(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func MyHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

`testWWW.go` 余下代码如下：

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number:", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/CheckStatusOK", CheckStatusOK)
    http.HandleFunc("/StatusNotFound", StatusNotFound)
    http.HandleFunc("/", MyHandler)

    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

我们现在需要开始测试 `testWWW.go` 了，为此我们应该创建一个 `testWWW_test.go` 文件。这个文件的内容由四部分展示。

`testWWW_test.go` 的第一部分包容如下代码：

```
package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
    "testing"
)
```

注意，为了用 Go 测试 web 应用，您需要引入 `net/http/httptest` 标准包。

`testWWW_tesst.go` 的第二部分如下：

```
func TestCheckStatusOK(t *testing.T) {
    req, err := http.NewRequest("GET", "/CheckStatusOK", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(CheckStatusOK)
    handler.ServeHTTP(rr, req)
```

`httptest.NewRecorder()` 函数返回一个 `httptest.ResponseRecorder` 对象，它被用于记录 HTTP 响应。

`testWWW_test.go` 的第三部分如下：

```
    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("handler returned %v", status)
    }

    expect := `Fine!`
    if rr.Body.String() != expect {
        t.Errorf("handler returned %v", rr.Body.String())
    }
}
```

您首先检查返回码是所期望的，然后再验证返回消息体也是您期望的。

`testWWW_test.go` 的余下代码如下：

```
func TestStatusNotFound(t *testing.T) {
    req, err := http.NewRequest("GET", "/StatusNotFound", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(StatusNotFound)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusNotFound {
        t.Errorf("handler returned %v", status)
    }
}
```

这个测试函数验证 `main` 包中的 `StatusNotFound()` 函数是否如期运行。

执行 `testWWW_test.go` 的这俩个测试函数将产生如下输出：

```
$ go test testWWW.go testWWW_test.go -v
=== RUN     TestCheckStatusOK
--- PASS:   TestCheckStatusOK (0.00s)
=== RUN     TestStatusNotFound
--- PASS:   TestStatusNotFound (0.00s)
PASS
ok      command-line-arguments  (cached)
```

Go 实现 web 客户端

在这节，将学习更多关于使用 Go 开发 web 客户端。这个 web 客户端工具的名字是 `webClient.go`，分为四部分展示。

`webClient.go` 的第一部分包含如下代码：

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)
```

`webClient.go` 的第二部分是读取命令行参数获得期望的 URL：

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL := os.Args[1]
```

`webClient.go` 的第三部分是实际操作发生的地方：

```
    data, err := http.Get(URL)

    if err != nil {
        fmt.Println(err)
        return
```

`http.Get()` 调用做了所有的工作，当您不想要处理参数和选项时它是相当的方便。然而，这样的调用方式在处理过程中没有灵活性。注意，`http.Get()` 返回一个 `http.Response` 变量。

余下的代码如下：

```
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

上面这段代码复制 `http.Response` 结构的 `Body` 字段内容到标准输出。

执行 `webClient.go` 将产生如下输出：

> *只有已小部分展示在这*

`webClient.go` 的主要问题是几乎不能控制处理过程——您要么获得全部 HTML 输出，要么什么都没有！

Go web客户端进阶

由于上一节的 web 客户端相当简单并没有任何灵活性，在这节您将学习如何更优雅的读取一个 URL，不使用 http.Get() 函数并且没有更多选项。这个演示程序命名为 `advancedWelClient.go`，并分为五个部分展示。

`advancedWebClient.go` 的第一部份包含如下代码：

```
package main

import (
    "fmt"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
    "path/filepath"
    "strings"
    "time"
)
```

`advancedWebClient.go` 的第二部分如下：

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
    URL, err := url.Parse(os.Args[1])
    if err != nil {
        fmt.Println("Error in parsing:", err)
        return
    }
```

`advancedWebClient.go` 的第三部分代码如下：

```
    c := &http.Client{
        Timeout: 15 * time.Second,
    }

    request, err := http.NewRequest("GET", URL.String(), nil)
    if err != nil {
        fmt.Println("Get:", err)
        return
    }
    httpData, err := c.Do(request)
    if err != nil {
        fmt.Println("Error in Do():", err)
        return
    }
```

`http.NewRequest()` 函数返回一个 `http.Request` 对象，它被赋予一个请求方法，一个 URL 和一个可选的消息体。`http.Do()` 函数使用 `http.Client` 对象发送一个 HTTP 请求(htt.Request)，并获得一个 HTTP 响应(http.Response)。`http.Do()` 以一种更易理解的方式做了 `http.Get()` 的工作。

在 `http.NewRequest()` 使用的 `GET` 字符串可以用 `http.MethodGet` 替换。

`advancedWebClient.go` 的第四部分包含代码如下：

```
    fmt.Println("Status code:", httpData.Status)
    header, _ := httputil.DumpResponse(httpData, false)
    fmt.Println(string(header))

    contentType := httpData.Header.Get("Content-Type")
    characterSet := strings.SplitAfter(contentType, "charset=")
    if len(characterSet) > 1 {
        fmt.Println("Character Set:", characterSet[1])
    }

    if httpData.ContentLength == -1 {
        fmt.Println("ContentLength is unknown!")
    } else {
        fmt.Println("ContentLength:", httpData.ContentLength)
    }
```

上面这段代码，您能看到如何开始搜索服务器响应来找到我们想要的。

`advancedWebClient.go` 的最后一部分如下：

```
    length := 0
    var buffer [1024]byte
    r := httpData.Body
    for {
        n, err := r.Read(buffer[0:])
        if err != nil {
            fmt.Println(err)
            break
        }
        length = length + n
    }
    fmt.Println("Calculated response data length:", length)
}
```

上面这段代码，您能看到一个计算服务器 HTTP 响应大小的技巧。如果您想显示这个 HTML 输出在您的屏幕上，您可以打印这个 `r` buffer 变量内容。

使用 `advancedWebClient.go` 访问一个 web 页面将产生如下比之前更丰富的输出：

```
$ go run advancedWebClient.go http://www.mtsoukalos.eu
Status code: 200 OK
HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 0
Cache-Control: no-cache, must-revalidate
Connection: keep-alive
Content-Language: en
Content-Type: text/html; charset=utf-8
Date: Sat, 24 Mar 2018 18:52:17 GMT
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Server: Apache/2.4.25 (Debian) PHP/5.6.33-0+deb8u1 mod_wsgi/4.5.
Vary: Accept-Encoding
Via: 1.1 varnish (Varnish/5.0)
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Generator: Drupal 7 (http://drupal.org)
X-Powered-By: PHP/5.6.33-0+deb8u1
X-Varnish: 886025

Character Set: utf-8
ContentLength is unknown!
EOF
Calculated response data length: 50176
```

执行 `advancedWebClient.go` 访问一个不同的 URL 将返回一个稍有不同的输出：

```
$ go run advancedWebClient.go http://www.google.com
Status code: 200 OK
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-7
Date: Sat, 24 Mar 2018 18:52:38 GMT
Expires: -1
P3p: CP="This is not a P3P policy! See g.co/p3phelp for more inf
Server: gws
Set-Cookie: 1P_JAR=2018-03-24-18; expires=Mon, 23-Apr-2018 18:52
Set-Cookie:
NID=126=csX1_koD30SJcC_1jAfcM2V8kTfRkppmAdmLjINLfclracMxuk6JGe4g
X-Frame-OPtions: SAMEORIGIN
X-Xss-Protection: 1; mode=block

Character Set: ISO-8859-7
ContentLength in unknown!
EOF
Calculated response data length: 10240
```

如果您使用 `advancedWebClient.go` 试图获取一个错误的 URL，将获得以下输出：

```
$ go run advancedWebClient.go http://www.google
Error in Do(): Get http://www.google: dial tcp: lookup www.googl
$ go run advancedWebClient.go www.google.com
Error in Do(): Get www.google.com: unsupported protocol scheme "
```

随意修改 `advancedWebClient.go` 以达到您想要的输出！

HTTP连接超时

这节将介绍一个处理网络连接超时的技巧，网络连接超时是花了太长的时间也没有连接上。记住您已经知道了一个技巧，是从第十章（Go 并发-进阶讨论），我们讨论 `context` 标准包时学到的。这个技巧展示在 `useContext.go` 源码文件中。

在这节介绍的方法实现起来非常简单。相关代码保存在 `clientTimeOut.go` 文件中，分为四个部分来介绍。这个程序接收俩个命令行参数，一个是 URL 另一个是超时秒数。注意第二个参数是可选的。

`clientTimeOut.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "io"
    "net"
    "net/http"
    "os"
    "path/filepath"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)
```

`clientTimeOut.go` 的第二段代码如下：

```
func Timeout(network, host string) (net.Conn, error) {
    conn, err := net.DialTimeout(network, host,timeout)
    if err != nil {
        return nil, err
    }
    conn.SetDeadline(time.Now().Add(timeout))
    return conn, nil
}
```

在下节，您将学习更多关于 `SetDeadline()` 的功能。 `Timeout()` 函数用在 `http.Transport` 变量的 `Dial` 字段。

`clientTimeOut.go` 的第三部分代码如下：

```
func main() {
    if len(os.Args) == 1 {
        fmt.Printf("Usage: %s URL TIMEOUT\n", filepath.Base(os.A
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(time.Duration(temp) * time.S
        }
    }

    URL := os.Args[1]
    t := http.Transport{
        Dial: Timeout,
    }
```

clientTimeOut.go 的其余代码如下：

```
    client := http.Client {
        Transport: &t,
    }

    data, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

使用在第十章（Go 并发-进阶讨论）开发的 slowWWW.go web 服务器测试 clientTimeOut.go web 客户端。

执行 clientTimeOut.go 俩次将产生如下输出：

```
$ go run clientTimeOut.go http://localhost:8001
Serving: /
Delay:  0
$ go run clientTimeOut.go http://localhost:8001
Get http://localhost:8001: read tcp[::1]:57397->[::1]:8001: i/o
```

从上面的输出您能看出，第一个请求连接所期望的 web 服务器没有问题。然而，第二个 `http.Get()` 请求花的时间比预期长，所以超时了！

SetDeadline 介绍

`net` 包使用 `SetDeadline()` 函数设置给定网络连接的读写截止时间。由于 `SetDeadline()` 函数的工作方式，您需要在任何读写操作前调用 `SetDeadline()` 。请记住，Go 使用截止日期来实现超时，因此您不需要在应用程序每次接收或发送任何数据时重置它。

服务端设置超时时间

在这小节，您将学习如何在服务端超时连接。您需要这样做，因为有时客户端结束 HTTP 连接的时间比预期的要长得多。这通常由于俩个原因而发生：第一个原因是客户端软件有 bug，第二个原因是服务进程正经历攻击！

在 `serverTimeOut.go` 源码文件中实现了这个技巧，并分四部分介绍。

`serverTimeOut.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Fprintf("Served: %s\n", r.Host)
}
```

`serverTimeOut.go` 的第二部分代码如下：

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for :%s\n", r.Host)
}
```

`serverTimeOut.go` 的第三段代码如下：

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    }

    m := http.NewServeMux()
    srv := &http.Server {
        Addr: PORT,
        Handler: m,
        ReadTimeout: 3 * time.Second,
        WriteTimeout: 3 * time.Second,
    }
```

在这个例子中，我们使用 `http.Server` 结构, 它的字段支持两类超时。第一个叫 `ReadTimeout`，第二个叫 `WriteTimeout`。`ReadTimeout` 字段的值定义读取整个请求，包括消息体的最大持续时间。

`WriteTimeout` 字段的值定义超时写入响应之前的最大持续时间。简单说，这是从请求头读取结束到响应写入结束的时间。

`serverTimeOut.go` 的剩余代码如下：

```
    m.HandleFunc("/time". timeHandler)
    m.HandleFunc("/", myHandler)

    err := srv.ListenAndServe()
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

我们现在执行 `serverTimeOut.go`，使用 `nc(1)` 与之交互。

```
$ go run serverTimeOut.go
Listening on http://0.0.0.0:8001
```

在 `nc(1)` 部分（在这个例子中作为一个虚拟 HTTP 客户端），您应该发出下一个命令来连接到 `serverTimeOut.go`：

```
$ time nc localhost 8001

real 0m3.012s
user 0m0.001s
sys  0m0.002s
```

由于我们没有发出任何命令，HTTP 服务器结束了连接。`time(1)` 程序的输出验证了服务器关闭连接所用时间。

设置超时的另外一种方法

这小节将介绍另外一种从客户端处理 HTTP 连接超时的方法。如您所见，这是最简单的超时处理形式，因为您只需要定义一个 `http.Client` 对象并设置它的 `Timeout` 字段为期望的超时值。

展示最后一种超时类型的程序命名为 `anotherTimeOut.go`，并分为四个部分来介绍。

`anotherTimeOut.go` 的第一部分如下：

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)
```

`anotherTimeOut.go` 的第二部分代码如下：

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please provide a URL")
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(time.Duration(temp) * time.S
        }
    }

    URL := os.Args[1]
```

`anotherTimeOut.go` 的第三部分代码如下：

```
    client := http.Client{
        Timeout: timeout,
    }
    client.Get(URL)
```

这是使用 `http.Client` 变量的 `Timeout` 字段定义超时周期的地方。

`anotherTimeOut.go` 的最后一段代码如下：

```
    data, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

执行 `anotherTimeOut.go` 并和在第十章（Go 并发-进阶讨论）开发的 `slowWWW.go` web 服务器交互，将产生如下输出：

```
$ go run anotherTimeOut.go http://localhost:8001
Get http://localhost:8001: net/http: request canceled (Client.Ti
$ go run anotherTimeOut.go http://localhost:8001 15
Serving: /
Delay: 8
```

抓包工具Wireshark和tshark

这节将简单的提及强大的 Wireshark 和 tshark 工具。**Wireshark** 是一个图形应用，是分析任何类型的网络流量的主流工具。尽管 Wireshark 非常强大，但有时您可能需要一个非图形界面的，可远程执行的轻量级应用。这种情况下，您可以使用 **tshark**，它是 Wireshark 的命令行版本。

很遗憾，Wireshark 和 tshark 的讨论超出了本章的范畴。

延展阅读

下面的资料一定会扩展您的视野，所以请找时间看一下：

- Apache web 服务器的官方网站http://httpd.apache.org。
- Nginx web 服务器的官方网站http://nginx.org。
- 您可能希望了解更多关于网络，TCP/IP 和它的各种服务，可以从阅读 **RFC** 文档开始。您可以在http://www.rfc-archive.org上找到此类文档。
- 浏览 Wireshark 和 tshark 的网站https://www.wireshark.org/了解更多信息。
- 浏览 `net` 标准包文档，在https://golang.org/pkg/net/。它是 Go 官方文档中最大的之一。
- 浏览 `net/http` 包文档https://golang.org/pkg/net/http/
- 如果您不想写任何 Go 代码就创建一个网站，您可以试一试 **Hugo utility**，它是用 Go 写的！您可以在https://gohugo.io(https://gohugo.io)了解更多关于 Hugo 框架。然而，对于每一个 Go 程序员真正有趣和有教育意义的是浏览它的源码，它在https://github.com/gohugoio/hugo。
- 您可以在https://golang.org/pkg/net/http/httptrace/浏览 `net/http/httptrace` 文档。
- 您可以在https://golang.org/pkg/net/http/pprof/找到 `net/http/pprof` 文档。
- 浏览 `nc(1)` 命令行工具的手册了解更多它的能力和各种命令行选项。
- 您可以在https://github.com/davecheney/httpstat找到由 Dave Cheney 开发的 `httpstat` 工具。它是使用 `net/http/httptrace` 包追踪 HTTP 最好的例子。
- 您可以浏览 `ab(1)` 手册https://httpd.apache.org/docs/2.4/programs/ab.html找到更多关于它的信息。

练习

- 自己不看这章的代码用 Go 写个 web 客户端
- 合并 `MXrecords.go` 和 `NSrecords.go` 的代码，创建一个基于命令行参数具备全部功能的单独程序
- 修改 `MXrecords.go` 和 `NSrecords.go` 的代码，使其也能接收 IP 地址作为输入
- 修改 `advancedWebClient.go` 使其保存 HTML 输出到一个外部文件
- 试着使用 goroutines 自己写个简单版本的 `ab(1)`
- 修改 `kvWeb.go` 使其支持在 key-value 存储的原来版本中的 `DELETE` 和 `LOOKUP` 操作
- 修改 `httpTrace.go` 使其有个标志能禁用 `io.Copy(os.Stdout, response.Body)` 表达式的执行

本章小结

本章讨论了用于 web 客户端和 web 服务器编程以及用 Go 创建网站的 Go 代码！您也了解了允许您定义 HTTP 连接参数的 `http.Response`， `http.Request` 和 `http.Transport` 结构。

另外，您也了解到怎样使用 Go 代码获取 Unix 机器的网络配置和怎样使用 Go 程序执行 DNS 查询，包括获取一个域名的 NS 和 MX 记录。

最后，我们讨论了 **Wireshark** 和 **tshark**，这俩个非常流行的工具可以让您捕获和（最重要的）分析网络流量。在这章开始，我们也提及了 `nc(1)` 工具。

这本书的最后章节，我们将继续讨论 Go 语言的网络编程。而这次，我们将介绍低级别的 Go 代码，它可以让您开发 TCP 客户端和服务器以及 UDP 客户端和服务进程。另外，您将了解到创建 RCP 客户端和服务器。希望您会喜欢！

网络编程 - 构建服务端和客户端

之前的章节讨论了关于网络编程的内容; 包括开发 web 客户端，web 服务端和 web 应用; DNS 查询; 和 HTTP 连接超时。

这章将带您更进一步，向您展示怎样编写您自己的 TCP 客户端和服务端，还有 UDP 客户端和服务端。此外，通过俩个例子展示怎样编写一个并发的 TCP 服务端。第一个例子相当简单，并发的 TCP 服务端只返回斐波那契序列数字。而第二个例子将使用第四章（组合类型的使用）的 keyValue.go 应用的代码，在它的基础上，把**key-value 存储**转变为一个并发 TCP 应用，可以在不需要浏览器的情况下操作。

在玩转 Go的这一章中，您将了解到如下内容：

1. net 标准包
2. TCP 客户端和服务端
3. 实现并发 TCP 服务端
4. UDP 客户端和服务端
5. 修改 第八章的 kvSaveLoad.go，告诉一个 *Unix 系统* 做什么，才能通过 TCP 连接提供请求
6. RCP 客户端和服务端

Go 标准库-net

在 Go 语言中，您不使用 net 包提供的功能是无法创建一个 TCP 或 UDP 的客户端或服务器的。

net.Dial() 方法作为客户端连接网络，而 net.Listen() 方法作为服务端告诉 Go 程序接收网络连接。这俩个方法只有第一个参数相同，都是网络类型。net.Dial() 和 net.Listen() 方法的返回值，是实现了 io.Reader 和 io.Writer 接口的 net.Conn 类型。

TCP 客户端

由于从之前的章节，您已经知道了，TCP 代表**传输控制协议**，并且它的主要特点是可靠性。每个包的 TCP 头部包含**源端口**和**目标端口**字段。这俩个字段，加上源和目标 IP 地址，组合成唯一标识单个 TCP 连接。这节实现的 TCP 客户端命名为 TCPclient.go，它由以下四部分构成。第一部分代码：

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

第二部分代码：

```
func main(){
    arguments := os.Args

    if len(arguments) == 1 {
        fmt.Println("Please provide host:port.")
        return
    }
    CONNECT := arguments[1]
    c, err := net.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
```

net.Dial() 方法连接远端服务器。这个方法的第一个参数定义使用的网络连接类型，第二个参数定义服务器地址，并且必须包含端口号。第一个参数的有效值为 tcp，tcp4 **(IPv4-only)**，tcp6 **(IPv6-ony)**，udp， udp4 **(IPv4-only)**，udp6 **(IPv6-only)**，ip，ip4 **(IPv4-only)**，ip6 **(IPv6-only)**，Unix **(Unix sockets)**，Unixgram 和 Unixpacket。

第三部分代码：

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(c, text + "\n")
```

前面这段代码用于从用户获取输入，使用 `os.Stdin` 文件进行验证读取。忽略从 `reader.ReadString()` 返回的 `error` 值不是一个好的实现，但它节省了空间。当然，你永远不要在正式软件中这样做。

最后一段代码：

```
        message, _ := bufio.NewReader(c).ReadString('\n')
        fmt.Print("->: " + message)
        if strings.TrimSpace(string(text)) == "STOP" {
            fmt.Println("TCP client exiting...")
            return
        }
    }
}
```

为了测试， `TCPclient.go` 将连接由 `netcat(1)` 实现的TCP 服务器，它将产生以下输出：

```
$ go run TCPclient.go 8001
dial tcp: address 8001: missing port in address
$ go run TCPclient.go localhost:8001
>> Hello from TCPclient.go!
->: Hi from nc!

>> STOP
->:
TCP client exiting...
```

请注意，给定协议如 TCP 和 UDP 的客户端本质上是可以通用的，这意味着它能够与支持其协议的多种服务器通信。您很快会看到，并不是使用指定协议的服务端应用必须实现指定的功能

另一个版本的 TCP 客户端

Go 提供了一个不同的函数集，也可以开发 TCP 客户端和服务器。在这一节，您将学习使用这些函数来编写 TCP 客户端。

TCP 客户端命名为 otherTCPclient.go，它有以下四部分。第一段代码：

```
package main

import(
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

第二段代码：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a server:port string!")
        return

    }

    CONNECT := arguments[1]
    tcpAddr, err := net.ResolveTCPAddr("tcp4", CONNECT)
    if err != nil {
        fmt.Println("ResolveTCPAddr:", err.Error())
        return
    }
```

net.ResolveTCPAddr() 函数返回一个 TCP 终点地址（类型是 `TCPAddr`），并且只能用于 TCP 网络。

第三段代码：

```
    conn, err := net.DialTCP("tcp4", nil, tcpAddr)
    if err != nil {
        fmt.Println("DialTCP:", err.Error())
        return
    }
```

net.DialTCP() 函数相当于 net.Dial() 对 TCP 网络。

最后的代码：

```
    for {
        reader := bufio.NewReader(os.Stdin)
        fmt.Print(">> ")
        text, _ := reader.ReadString('\n')
        fmt.Fprintf(conn, text + "\n")

        message, _ := bufio.NewReader(conn).ReadString('\n')
        fmt.Print("->: " + message)
        if strings.TrimSpace(string(text)) == "STOP" {
            fmt.Println("TCP client exiting...")
            conn.Close()
            return
        }
    }
}
```

执行 otherTCPclient.go，并与 TCP 服务交互将产生如下输出：

```
$ go run otherTCPclient.go localhost:8001
>> Hello from otherTCPclient.go!
->: Hi from netcat!
>> STOP
->:
TCP client exiting...
```

对于这个例子，TCP 服务器使用 netcat(1) 工具，执行输出如下：

```
$ nc -l 127.0.0.1 8001
Hello from otherTCPclient.go!

Hi from netcat!
STOP
```

TCP 服务器

这节编写一个 TCP 服务器，给客户端返回一个当前日期和时间的网络包。在实践中，这意味着服务器在接收到一个客户端连接后，将从 Unix 系统获取时间和日期，并把这个数据返回给客户端。

这个工具命名为 TCPserver.go，并由 4 部分构成。

第一部分：

```
package main

import(
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
    "time"
)
```

第二部分包含以下代码：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide port number")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp" , PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
```

net.Listen() 函数用于监听连接。如果第二个参数不包含 IP 地址，只有端口号的话，net.Listen() 将监听本地系统上所有可用的 IP 地址。

TCPserver.go 的第三段如下：

```
    c, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }
```

Accept() 函数等待接收连接并返回一个通用的 `Conn` 变量。这个特定的 TCP 服务器有个错误是它只能接收第一个与它建立连接的 TCP 客户端，因为 Accept() 函数是在接下来的 `for` 循环外调用的。

TCPserver.go 的剩余代码如下：

```
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }
        if strings.TrimSpace(string(netData)) == "STOP" {
            fmt.Println("Exiting TCP server!")
            return
        }
        fmt.Print("-> ", string(netData))
        t := time.Now()
        myTime := t.Format(time.RFC3339) + "\n"
        c.Write([]byte(myTime))
    }
}
```

执行 TCPServer.go 并使用一个 TCP 客户端应用和它交互，将产生如下输出：

```
$ go run TCPServer.go 8001
-> HELLO
Exiting TCP server!
```

在客户端这边，您将看到如下输出：

```
$ nc 127.0.0.1 8001
HELLO
2018-05-07T14:40:05+03:00
STOP
```

如果这个 TCPServer.go 工具试图使用一个被其他 Unix 进程占用的 TCP 端口，您将看到下面的错误信息：

```
$ go run TCPserver.go 9000
listen tcp :9000: bind: address already in use
```

最后，如果这个 TCPServer.go 工具试图使用一个在 1-1024 范围内需要 root 权限的 TCP 端口，您将看到下面的错误信息：

```
$ go run TCPserver.go 80
listen tcp :80: bind: permission denied
```

另一个版本的 TCP 服务器

这节，您将看到另一个 Go 编写的 TCP 服务器实现。这次，TCP 服务器实现一个原样返回客户端发来数据的 **Echo 服务**。这个程序命名为 otherTCPserver.go，它将分为四个部分。

第一部分如下：

```
package main

import(
    "fmt"
    "net"
    "os"
    "strings"
)
```

第二部分如下：

```
func main() {
    arguments := os.Args
    if len(arguments == 1) {
        fmt.Println("Please provide a port number!")
        return

    }
    SERVER := "localhost" + ":" + arguments[1]
    s, err := net.ResolveTCPAddr("tcp", SERVER)
    if err != nil {
        fmt.Println(err)
        return
    }
    l, err := net.ListenTCP("tcp", s)
    if err != nil {
        fmt.Println(err)
        return
    }
```

net.ListenTCP() 函数相当于 net.Listen() 对 TCP 网络。

第三部分如下：

```
    buffer := make([]byte, 1024)
    conn, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }
```

otherTCPserver.go 余下代码如下：

```
    for {
        n, err := conn.Read(buffer)
        if err != nil {
            fmt.Println(err)
            return
        }
        if strings.TrimSpace(string(buffer[0:n]) == "STOP") {
            fmt.Println("Exiting TCP server!")
            conn.Close()
            return
        }

        fmt.Print("> ", string(buffer[0:n-1]))
        _, err = conn.Write(buffer)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

执行 otherTCPserver.go 并用一个客户端和它交互将产生如下输出：

```
$ go run otherTCPserver.go 8001
> 1
> 2
> Hello!
> Exiting TCP server!
```

这个例子用 otherTCPclient.go 作为客户端，您将看到如下输出：

```
$ go run otherTCPclient.go localhost:8001
>> 1
->: 1
>> 2
->: 2
>> Hello!
->: Hello!
>> ->:
>> STOP
->: TCP client exiting...
```

最后，我将演示一个方法，用来在 Unix 机器上找到监听给定 TCP 或 UDP 端口的进程名。如果您想知道是哪个进程使用了 8001 的 TCP 端口号，您可以执行如下命令：

```
$ sudo lsof -n -i :8001
COMMAND    PID     USER FD TYPE             DEVICE SIZE/OFF NODE
TCPserver  87775 mtsouk 3u IPv6 0x98d55014e6c9360f      0t0  TCP
```

UDP 客户端

如果您知道怎么开放一个 TCP 客户端，那么由于 UDP 协议的简单性，对于您来说开发一个 UDP 客户端应该更简单。

> *UDP 和 TCP 最大的不同是 UDP 的不可靠性。这也意味着，UDP 要比 TCP 简单，因为 UDP 不需要保持一个 UDP 连接的状态。*

展现这个主题的工具命名为 UDPclient.go，它分成四个代码片段。
UDPclient.go 的第一段如下：

```
package main

import(
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

UDPclient.go 的第二段如下：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:post string")
        return
    }
    CONNECT := arguments[1]
    s, err := net.ResolveUDPAddr("udp4", CONNECT)
    c, err := net.DialUDP("udp4", nil, s)

    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("The UDP server is %s\n", c.RemoteAddr().String()
    defer c.Close()
```

net.ResolveUDPAddr() 函数返回一个由第二个参数定义的 UDP 终点地址。第一个参数（ `udp4` ）规定了程序只能支持 IPv4 协议。

net.DialUDP() 函数相当于 net.Dial() 对 UDP 网络。

UDPclient.go 的第三段代码如下：

```
    for {
        reader := bufio.NewReader(os.Stdin)
        fmt.Print(">> ")
        text, _ := reader.ReadString("\n")
        data := []byte(text + "\n")
        _, err = c.Write(data)
        if strings.TrimSpace(string(data)) == "STOP" {
            fmt.Println("Exiting UDP client!")
            return
        }
```

上面这段代码需要使用者输入一些文本，发送给 UDP 服务器。使用 `buio.NewReader(os.Stdin)` 从标准输入中读取输入的文本。`Write(data)` 方法通过 UDP 网络连接发送数据。

剩下的代码如下：

```
        if err != nil {
            fmt.Println(err)
            return
        }

        buffer := make([]byte, 1024)
        n, _, err := c.ReadFromUDP(buffer)
        if err != nil {
            fmt.Println(err)

            return
        }
        fmt.Printf("Reply: %s\n", string(buffer[0:n]))
    }
}
```

一旦客户端数据被发送出去后， 您必须等待 UDP 服务器发送的数据，使用 `ReadFromUDP()` 读取。

执行 `UDPclient.go` 并使用 `netcat(1)` 工具作为 UDP 服务器与之交互，将产生如下输出：

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: Hi there!

>> Have to leave - bye!
Reply: OK.

>> STOP
Exiting UDP client!
```

在 UDP 服务器这边，输出如下：

```
$ nc -v -u -l 127.0.0.0 8001
Hello!

Hi there!
Have to leave - bye!

OK.
STOP

^C
```

因为当 `nc(1)` 收到输入的 `STOP` 字符串时，没有任何代码能告诉它终止，所以键入 `Control + C` 去停止它。

UDP 服务器

这节开发的 UDP 服务器的目的是给它的 UDP 客户端返回从 1 到 1,000 的随机数。 这个程序命名为 UDPserver.go， 并分为 4 个部分。

UDPserver.go 的第一部分如下：

```
package main

import(
    "fmt"
    "math/rand"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)

func random(min, max, int) int {
    return rand.Intn(max-min) + min
}
```

UDPserver.go 的第二部分如下：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }
    PORT := ":" + arguments[1]

    s, err := net.ResolveUDPAddr("udp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
```

UDPServer.go 的第三部分如下：

```
    connection, err := net.ListenUDP("udp4", s)
    if err != nil {
        fmt.Println(err)
        return
    }

    defer connection.Close()
    buffer := make([]byte, 1024)
    rand.Seed(time.Now().Unix())
```

对于 UDP 网络，`net.ListenUDP()` 函数作用类似于 `net.ListenTCP()`。

UDPserver.go 余下代码如下：

```
    for {
        n, addr, err := connection.ReadFromUDP(buffer)
        fmt.Print("->", string(buffer[0:n-1]))

        if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
            fmt.Println("Exiting UDP server!")
            return
        }

        data := []byte(strconv.Itoa(random[1, 1001]))
        fmt.Printf("data: %s\n", string(data))
        _, err = connection.WriteToUDP(data, addr)
        if err != nil {
            fmt.Println(err)
            return
        }

    }
}
```

`ReadFromUDP()` 函数允许您使用一个字节切片缓冲区从 UDP 连接读取数据。

执行 UDPserver.go 并用 UDPclient.go 连接它，将产生如下输出：

```
$ go run UDPserver.go 8001
-> Hello!
data: 156
-> Another random number please :)
data: 944
-> Leaving...
data: 491
-> STOP
Exiting UDP server!
```

客户端输出如下：

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: 156
>> Another random number please :)
Reply: 944
>> Leaving...
Reply: 491
>> STOP
Exiting UDP client!
```

并发 TCP 服务器

在这节，您将学习使用 goroutines 开放一个**并发 TCP 服务器**。TCP 服务器将给每个接入的连接开启一个新的 goroutines 来处理请求。一个并发 TCP 服务器可以接收更多请求，能同时为多个客户端提供服务。

这个 TCP 并发服务器的工作是接收一个正整数并从斐波纳切序列返回一个自然数。如果输入发送错误，则返回值为-1。由于斐波纳切序列数的计算慢，我们将使用曾首次在第11章出现的一个算法，代码测试，优化和分析都包含在 `benchmarkMe.go` 文件中。另外，这次用到的算法将会详细的讲解。

我们把程序命名为 `fiboTCP.go`，并把代码分成五部分。由于把 web 服务的端口号定义为命令行参数被认为是良好的实现方式，这次 `fiboTCP.go` 将完全按此来做。

`fiboTCP.go` 的第一部分如下：

```
package main

import(
    "bufio"
    "fmt"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)
```

`fiboTCP.go` 的第二部分如下：

```
func f(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

上面这段代码，您能看到 `f()` 函数实现了斐波纳切序列自然数的生成。一开始看这个算法很难理解，但它非常有效且运行速度也很快。首先，`f()` 函数使用了一个被命名为 `fn` 的字典，这在计算斐波纳切序列数时很不寻常。第二，`f()` 函数使用了一个 `for` 循环，这也相当不寻常。最后，`f()` 函数没有使用递归，这也是它执行快的主要原因。

在 `f()` 函数中用到的算法思想使用了动态规划技巧，每当一个斐波纳切数被计算后，就把它放入 `fn` 字典中，这样它就不会再被计算了。这个简单的想法节省了很多时间，特别是需要计算较大斐波纳切数时，因为您不必对相同的斐波纳切数计算多次。

`fiboTCP.go` 的第三部分如下：

```
func handleConnection(c net.Conn) {
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }

        temp := strings.TrimSpace(string(netData))
        if temp == "STOP" {
            break
        }

        fibo := "-1\n"
        n, err := strconv.Atoi(temp)
        if err == nil {
            fibo = strconv.Itoa(f(n)) + "\n"
        }
        c.Write([]byte(string(fibo)))
    }
    time.Sleep(5 * time.Second)


    c.Close()
}
```

`handleConnection()` 函数处理并发 TCP 服务器的每个客户端。

`fiboTCP.go` 的第四部分如下：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
```

`fiboTCP.go` 的剩余代码如下：

```
    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        go handleConnection(c)
    }
}
```

`go handleConnection(c)` 声明实现了程序的并发性，每次连接一个新 TCP 客户端时它就开启一个新的 goroutine。goroutine 被并发执行，使服务器有机会服务更多的客户端。

执行 `fiboTCP.go` 并使用 `netcat(1)` 和 `TCPclient.go` 在俩个不同的终端窗口和它交互，输出如下：

```
$ go run fiboTCP.go 9000
n: 10
fibo: 55
n: 0
fibo: 1
n: -1
fibo: 0
n: 100
fibo: 3736710778780434371
n: 12
fibo: 144
n: 12
fibo: 144
```

在 `TCPclient.go` 这边的输出如下：

```
$ go run TCPclient.go localhost:9000
>> 12
->: 144
>> a
->: -1
>> STOP
->: TCP client exiting...
```

在 `netcat(1)` 这边的输出如下：

```
$ nc localhost 9000
10
55
0
1
-1
0
100
3736710778780434371
ads
-1
STOP
```

当您发送 `STOP` 字符串给服务进程时。为指定 TCP 客户端服务的 goroutine 将终止，这将引起连接关闭。

最后，值得关注的是俩个客户端被同时提供服务，这可以通过下面的命令输出来验证：

```
$ netstat -anp TCP | grep 9000
tcp4    0 0 127.0.0.1 9000  127.0.0.1.57309     ESTABLISHED
tcp4    0 0 127.0.0.1.57309 127.0.0.1.9000      ESTABLISHED
tcp4    0 0 127.0.0.1 9000  127.0.0.1.57305     ESTABLISHED
tcp4    0 0 127.0.0.1 57305 127.0.0.1.9000      ESTABLISHED
tcp4    0 0 *.9000          *.*                 LISTEN
```

上面命令的最后一行输出告诉我们有一个进程在监听 `9000` 端口，这意味着您仍能连接 `9000` 端口。输出的头俩行显示有个客户端使用 `57309` 端口与服务进程通信。第三和第四行证明有另一个客户端连接到监听 `9000` 端口的服务上。这个客户端使用的 TCP 端口是 `57035` 。

简洁的并发TCP服务器

尽管上节的并发 TCP 服务器运作良好，但是它还不能为实际应用提供服务。因此，在这节您将学到怎样把第四章中的 `keyValue.go` 文件对于复杂类型的使用转化为一个功能齐全的并发 TCP 应用。

为了能够和网络中的 key-value 存储交互，我们创建自定义的 TCP 协议。您将需要为 key-value 存储的每一个函数定义关键字。为简单起见，每个关键字都跟着相关数据。大多数命令的结果将是成功或失败消息。

> 设计您自己的 TCP 或 UDP 协议不是一个简单的工作。这意味着设计一个新协议时，您必须特别细致小心。这里的关键是在您开始编写生产代码之前文档化所有内容。

这个主题使用的工具命名为 `kvTCP.go`，它被分为六个部分。

`kvTCP.go` 的第一部分如下：

```
package main

import (
    "bufio"
    "encoding/gob"
    "fmt"
    "net"
    "os"
    "strings"
)

type myElement struct {
    Name string
    Surname string
    Id string
}

const welcome = "Welcome to the Key-value store!\n"

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

`kvTCP.go` 的第二部分如下：

```
func handleConnection(c net.Conn) {
    c.Write([]byte(welcome))
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }
        command := strings.TrimSpace(string(netData))
        tokens := strings.Fields(command)
        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 3:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 4:
            tokens = append(tokens, "")
        }

        switch tokens[0] {
        case "STOP":
            err = save()
            if err != nil {
                fmt.Println(err)
            }
            c.Close()
            return
        case "PRINT":
            PRINT(c)
        case "DELETE":
            if !DELETE(tokens[1]) {
                netData := "Delete operation failed!\n"
                c.Write([]byte(netData))
            }else{
                netData := "Delete operation successful!\n"
                c.Write([]byte(netData))
            }
        case "ADD":
```

```
            n := myElement{tokens[2], tokens[3], tokens[4]}
            if !ADD(tokens[1], n) {
                netData := "Add operation failed!\n"
                c.Write([]byte(netData))
            } else {
                netData := "Add operation successful!\n"
                c.Write([]byte(netData))
            }
            err = save()
            if err != nil {
                fmt.Println(err)
            }
        case "LOOKUP":
            n := LOOKUP(tokens[1])
            if n != nil {
                netData := fmt.Sprintf("%v\n", *n)
                c.Write([]byte(netData))
            } else {
                netData := "Did not find key!\n"
                c.Write([]byte(netData))
            }
        case "CHANGE":
            n := myElement{tokens[2], tokens[3], tokens[4]}
            if !CHANGE(tokens[1], n) {
                netData := "Update operation failed!\n"
                c.Write([]byte(netData))
            } else {
                netData := "Update operation successful!\n"
                c.Write([]byte(netData))
            }
            err = save()
            if err != nil {
                fmt.Println(err)
            }
        default:
            netData := "Unknown command - please try again!\n"
            c.Write([]byte(netData))
        }
    }
}
```

`handleConnection()` 函数和每个 TCP 客户端交互并解析客户端的输入。

`kvTCP.go` 的第三部分包含如下代码：

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}
```

`kvTCP.go` 的第四段如下：

```
func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}
```

上面的这些函数实现与 `keyValue.go` 一样。它们没有直接和 TCP 客户端交互。

`kvTCP.go` 的第五部分包含代码如下：

```
func PRINT(c net.Conn) {
    for k, d := range DATA {
        netData := fmt.Sprintf("key: %s value: %v\n", k, d)
        c.Write([]byte(netData))
    }
}
```

`PRINT()` 函数直接发送数据给 TCP 客户端，一次一行。

这个程序的剩余代码如下：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

    err = load()
    if err != nil {
        fmt.Println(err)
    }

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }
        go handleConnection(c)
    }
}
```

执行 `kvTCP.go` 将产生如下输出：

```
$ go run kvTCP.go 9000
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
Saving /tmp/dataFile.gob
```

为了这节的目的， `netcat(1)` 工具用来作为 `kvTCP.go` 的客户端：

```
$ nc localhost 9000
Welcome to the Key-value store!
PRINT
LOOKUP 1
Did not find key!
ADD 1 2 3 4
Add operation successful!
LOOKUP 1
{2 3 4}
ADD 4 -1 -2 -3
Add operation successful!
PRINT
key: 1 value: {2 3 4}
key: 4 value: {-1 -2 -3}
STOP
```

`kvTCP.go` 文件是一个使用 goroutines 的并发应用，它能够同时服务多个 TCP 客户端。然而，所有的 TCP 客户端共享相同的数据！

远程调用（RPC）

远程调用（**RPC**）是一种使用 TCP/IP 的进程间通信的 客户端-服务器机制。 RPC 客户端和 RPC 服务器都使用下面这个命名为 `sharedRPC.go` 的包开发。

```
package sharedRPC

type MyFloats struct {
    A1, A2 float64
}

type MyInterface interface {
    Multiply(arguments *MyFloats, reply *float64) error
    Power(arguments *MyFloats, reply *float64) error
}
```

`sharedRPC` 包定义了一个名为 `MyInterface` 的接口和一个名为 `MyFloats` 的结构，客户端和服务器都将会使用到。然后，只有 RPC 服务器需要实现这个接口。

之后，您需要执行如下命令安装 `sharedRPC.go` 包：

```
$ mkdir  -p ~/go/src/sharedRPC
$ cp sharedRPC.go ~/go/src/sharedRPC/
$ go install sharedRPC
```

RPC 客户端

这节，您将看到被存为 `RPCclient.go` 的 RPC 客户端代码，它被分为四部分进行讲解。

`RPCclient.go` 的第一部分：

```
package main

import (
    "fmt"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

`RPCclient.go` 第二部分包含的代码如下：

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port sting!")
        return
    }


    CONNECT := arguments[1]
    c, err := rpc.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
```

注意，尽管 RPC 服务器使用 TCP，但使用 `rpc.Dial()` 函数代替 `net.Dial()` 连接 RPC 服务器。

`RPCclient.go` 的第三部分如下：

```
    args := sharedRPC.MyFloats(16, -0.5)
    var reply float64

    err = c.Call("MyInterface.Multiply", args, &reply)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Reply (Multiply): %f\n", reply)
```

RPC 客户端和 RPC 服务器之间通过 `Call()` 函数交换函数名，参数和函数返回结果，而 RPC 客户端对函数的具体实现一无所知。

`RPCclient.go` 的其余代码如下：

```
    err = c.Call("MyInterface.Power", argus, &reply)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Reply (Power): %f\n", reply)
}
```

如果您试图在没有一个运行的 RPC 服务器的情况下执行 `RPCclient.go` 将会得到如下错误信息：

```
$ go run RPCclient.go localhost:1234
dial tcp [::1]:1234: connect: connection refused
```

RPC 服务器

RPC 服务器被保存为 `RPCserver.go`，并分为五个部分来介绍。

`RPCserver.go` 的第一部分如下：

```
package main

import(
    "fmt"
    "math"
    "net"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

`RPCserver.go` 的第二部分代码如下：

```
type MyInterface struct {}

func Power(x, y float64) float64 {
    return math.Pow(x, y)
}

func (t *MyInterface) Multiply(arguments *sharedRPC.MyFloats, re
    *reply = arguments.A1 * argumentsA2
    return nil
}

func (t *MyInterface) Power(arguments *sharedRPC.MyFloats, reply
    *reply = Power(arguments.A1, arguments.A2)
    return nil
}
```

上面这段代码，RPC 服务器实现了所需接口以及名为 `Power()` 的辅助函数。

`RPCserver.go` 的第三段如下：

```
func main() {
    PORT := ":1234"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
```

`RPCserver.go` 的第四部分代码如下：

```
    myInterface := new(MyInterface)
    rpc.Register(myInterface)
    t, err := net.ResolveTCPAddr("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    l, err := net.ListenTCP("tcp4", t)
    if err != nil {
        fmt.Println(err)
        return
    }
```

`rpc.Register()` 函数的调用使这个程序成为 RPC 服务器。但是，由于 RPC 服务器使用 TCP 协议，它仍需要调用 `net.ResolveTCPAddr()` 和 `net.ListenTCP()` 。

`RPCserver.go` 的其余代码如下：

```
    for {
        c, err := l.Accept()
        if err != nil {
            continue
        }
        fmt.Printf("%s\n", c.RemoteAddr())
        rpc.ServerConn(c)
    }
}
```

`RemoteAddr()` 函数返回接入的 RPC 客户端 IP 地址和端口。 `rpc.ServerConn()` 函数为 RPC 客户端提供服务。

执行 `RPCserver.go` 并等待 `RPCclient.go` 连接将产生如下输出：

```
$ go run RPCserver.go
127.0.0.1:52289
```

执行 `RPCclient.go` 将产生如下输出：

```
$ go run RPCclient.go localhost:1234
Reply (Multiply): -8.000000
Reply (Power): 0.250000
```

底层网络编程

虽然 `http.Transport` 结构允许您修改网络连接的底层参数，但您可以编写允许您读取网络包原始数据的 Go 代码。这有两个棘手的问题：

- 网络包采用二进制格式，这要求您查找特定类型的网络数据包，而不仅仅是任何类型的网络数据包。
- 为了发送一个网络数据包，您必须自己构建。

接下来要展示的是 `lowLevel.go`，并把它分为三个部分。注意 `lowLevel.go` 捕获 ICMP 数据包，使用 IPv4 协议并打印包的内容。另外，由于安全原因操作原始网络数据需求 root 权限。

`lowLevel.go` 的第一段如下：

```
package main

import(
    "fmt"
    "net"
)
```

`lowLevel.go` 的第二段代码如下：

```
func main() {
    netaddr, err := net.ResolveIPAddr("ip4", "127.0.0.1")
    if err != nil {
        fmt.Println(err)
        return
    }

    conn, err := net.ListenIP("ip4:icmp", netaddr)
    if err != nil {
        fmt.Println(err)
        return
    }
```

ICMP 协协议被定义在 `net.ListenIP()` 函数的第一个参数的第二部分。此外，`ip4` 部分告诉程序只捕获 IPv4 流量。

`lowLevel.go` 的其余 Go 代码如下：

```
    buffer := make([]byte, 1024)
    n, _, err := conn.ReadFrom(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("% X\n", buffer[0:n])
}
```

上面这段代码告诉 `lowLevel.go` 只读取一个网络包，因为没有 `for` 循环。

ICMP 协议由 `ping(1)` 和 `traceroute(1)` 命令使用，所以为了产生 ICMP 流量，可以使用它们中的任何一个。当 `lowLevel.go` 已经运行后，在所有的 Unix 机器上使用如下命令就会产生 ICMP 网络流量。

```
$ ping -c 5 localhost
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.037 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.117 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.052 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.049 ms

--- localhost ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.037/0.059/0.117/0.030 ms
$ traceroute localhost
traceroute to localhost (127.0.0.1), 64 hops max, 52 byte packet
1   localhost(127.0.0.1)  0.255 ms  0.048 ms  0.067 ms
```

在 macOS High Sierra 机器上用 root 权限执行 `lowLevel.go` 将产生如下输出：

```
$ sudo go run lowlevel.go
03 03 CD DA 00 00 00 00 45 00 34 00 B4 0F 00 00 01 00 00 7F 00 0
$ sudo go run lowLevel.go
00 00 0B 3B 20 34 00 00 5A CB 5C 15 00 04 32 A9 08 09 0A 0B 0C 0
```

第一个输出的例子是 `ping(1)` 命令产生的，第二个是 `traceroute(1)` 命令产生的。

在 Debian Linux 机器上运行 `lowLevel.go` 将产生如下输出：

```
$ uname -a
Linux mail 4.14.12-x86_64-linode92 #1 SMP Fri Jan 5 15:34:44 UTC
# go run lowLevel.go
08 00 61 DD 3F BA 00 01 9A 5D CB 5A 00 00 00 00 26 DC 0B 00 00 0
# go run lowLevel.go
03 03 BB B8 00 00 00 00 45 00 00 3C CD 8D 00 00 01 11 EE 21 7F 0
```

`uname(1)` 命令打印出 Linux 系统的有用信息。注意，在当前的 Linux 机器上，您应该在执行 `ping(1)` 命令时使用 -4 标志来告诉它使用 IPv4 协议。

获取ICMP数据

这节，您将学习怎样使用 `syscall` 库获取原始 ICMP 网络数据，和 `syscall.SetsockoptInt()` 去设置 socket 选项。牢记发送原始 ICMP 数据是非常困难的，因为您不得不自己构造原始网络包。这个程序的名称是 `syscallNet.go`，并显示在四个部分中。

`syscallNet.go` 的第一部分如下：

```
package main

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

`syscallNet.go` 的第二部分包含代码如下：

```
func main() {
    fd, err := syscall.Socket(syscall.AF_INET, syscall.SOCK_RAW,
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }
    f := os.NewFile(uintptr(fd), "captureICMP")
    if f == nil {
        fmt.Println("Error is os.NewFile:", err)
        return
    }
```

`syscall.AF_INET` 参数告诉 `syscall.Socket()` 您想使用 IPv4。`syscall.SOCK_RAW` 参数使生成的 socket 成为原始 socket。这最后一个参数，`syscall.IPPROTO_ICMP`，告诉 `syscall.Socket()` 您只对 ICMP 通信感兴趣。

`syscallNet.go` 的第三部分如下：

```
    err = syscall.SetsockoptInt(fd, syscall.SOL_SOCKET, syscall.
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }
```

调用 `syscall.SetsockoptInt()` 设置 socket 的接收 buffer 大小为 256。`syscall.SOL_SOCKET` 参数是为了说明您想要在 socket 层级上工作。

`syscallNet.go` 的其余 Go 代码如下：

```
    for {
        buf := make([]byte, 1024)
        numRead, err := f.Read(buf)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Printf("% X\n", buf[:numRead])
    }
}
```

由于 `for` 循环，`syscallNet.go` 将一直抓取 ICMP 网络包直到您手动终止它。

在 macOS High Sierra 机器上执行 `syscallNet.go` 将产生如下输出：

```
$ sudo go run syscallNet.go
45 00 40 00 BC B6 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
45 00 40 00 62 FB 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
45 00 40 00 9A 5F 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
45 00 40 00 6E 0D 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
45 00 40 00 3A 07 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
45 00 24 00 45 55 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
45 00 24 00 E8 1E 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
45 00 24 00 2A 4B 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 0
```

在 Debian Linux 机器上运行 `syscallNet.go` 将产生如下输出：

```
# go run syscallNet.go
45 00 00 54 7F E9 40 00 40 01 BC BD 7F 00 00 01 7F 00 00 01 08 0
45 00 00 54 7F EA 00 00 40 01 FC BC 7F 00 00 01 7F 00 00 01 00 0
45 C0 00 44 68 54 00 00 34 01 8B 8E 86 77 DC 57 6D 4A C1 FD 03 0
45 00 00 54 80 4E 40 00 40 01 BC 58 7F 00 00 01 7F 00 00 01 08 0
45 00 00 54 80 4F 00 00 40 01 FC 57 7F 00 00 01 7F 00 00 01 00 0
45 00 00 54 80 9B 40 00 40 01 BC 0B 7F 00 00 01 7F 00 00 01 08 0
```

接下来的任务

从哲学上讲，没有一本编程书是完美的，这本也不例外！我有没谈及到的 Go 主题吗？当然有！为什么呢？因为一本书中总有更多的主题要涉及，所以如果我试图涵盖所有的主题，那么这本书就永远不会出版！这种情况在某种程度上类似于一个程序的规范——你可以随时添加新的、令人兴奋的特性，但是如果你不冻结它的具体特性，这个程序将永远处于开发阶段，它将永远无法为你的目标观众做好准备。我留下的一些 Go 主题可能会出现在这本书的第二版！

读完这本书后，好处是您可以自学了，您能从任何一本好的计算机编程书中获取最大的好处。这本书的主要目的是帮您学习怎么用 Go 语言编程，并获得一些相关经验。然而，没有什么可以代替您自己去探索，并且失败往往成为学习一门编程语言的唯一方式，因此您必须持续不断的拓展一些不同寻常的知识。

对于我来说，这是另一本 Go 语言书的结束，但对您是一段旅程的开始！感谢您购买本书，现在准备开始用 Go 编写您自己的软件并学习新知识吧！

延展阅读

查看以下资源：

- 访问 Go 标准库 `net` 文档，在 `https://golang.org/pkg/net` 可以找到。这是 Go 文档中最大的文档之一。
- 尽管这本书谈到了 **RPC**，但没谈到 **gRPC**，这个开源的，高性能的 RPC 框架。用 Go 语言实现的 gRPC 包在 `https://github.com/grpc/grpc-go` 。
- IPv4的 ICMP 协议定义在 RFC 792。它可以在很多地方找到，包括 `https://tools.ietf.org/html/rfc792` 。
- **WebSocket** 是客户端和远程主机间双向通信的协议。一个 Go 实现的 WebSocket 在 `https://github.com/gorilla/websocket` 。您可以在 `http://www.rfc-editor.org/rfc/rfc6455.txt` 学到更多关于 WebSocket。
- 如果您真对网络编程感兴趣并想使用 RAW TCP 数据包，您可以在 `https://github.com/google/gopacket` 找到有趣并有帮助的信息和工具。
- 在 `https://github.com/mdlayher/raw` 的 `raw` 包可以让您在网络设备的设备驱动级别读写数据。

练习

- 用 Go 开发一个 **FTP 客户端**
- 接下来用 Go 开发一个 FTP 服务器。实现 FTP 客户端 和 FTP 服务器哪个更困难？为什么？
- 试着实现一个 Go 版本的 `nc(1)` 工具。编写如此复杂工具的秘诀是在实现所有可能选项前，先从实现一个原工具的基础功能版本开始。
- 修改 `TCPserver.go`，使其返回一个网络包中的数据和另一个网络包中的时间
- 修改 `TCPserver.go`，使其可以按顺序为多个客户机提供服务。注意，这和同时服务多个请求不同。简单讲，使用 `for` 循环，以便可以多次执行 `Accept()` 调用。
- TCP 服务器，如 `fiboTCP.go`，在接收到给定的信号时往往会终止，因此在 `fiboTCP.go` 中添加信号处理代码，如第8章所述，告诉UNIX系统该怎么做。
- 使用 Go 中普通的 TCP 代替 `http.ListenAndServe()` 函数，开发一个您自己的小型 web server。

本章小结

这章解决了很多有趣的事情，包括开发 UDP 和 TCP 客户端和服务器，它们是工作在 TCP/IP 计算机网络之上的应用。

这是这本书最后一章，我向您表示祝贺并为您选择这本书表示感谢。我希望您能觉得它有用并在您的 Go 之旅中把它作为参考继续使用。

Soli Deo gloria