EXPERT INSIGHT

Mastering Go

Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures

Second Edition

Mihalis Tsoukalos

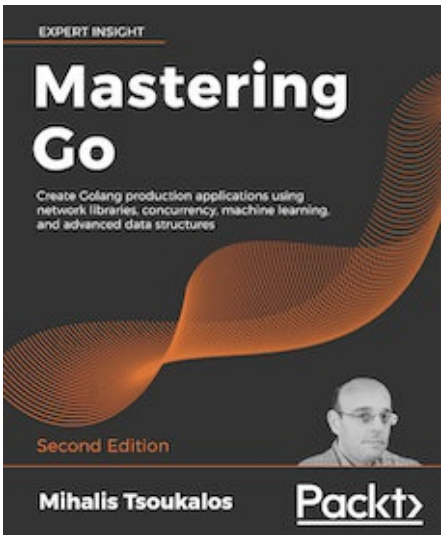
Packt

目　录

介绍

Mastering_Go_Second_Edition_Zh_CN

Mastering Go 第二版中文版来袭！



英文版第二版的 Mastering Go不知不觉上线了，https://www.packtpub.com/programming/mastering-go-second-edition

第一版在广大Golang爱好者的大力支持下已经完成release，我们开始第二版的翻译。

翻译规则目前先参考第一版，https://github.com/hantmac/Mastering_Go_ZH_CN

领取翻译章节的话，先起一个issue，表明领取的章节，然后给一个大致的DDL，及时同步翻译进展，以方便其他人了解进度。

项目刚刚启动，对初学者来说可以 强迫自己读一些英文原著，也是不错的学习机会；golang大佬也可以对翻译的效果给予指导。

有兴趣的可加入交流群，或者通过邮件沟通：hantmac@outlook.com（发邮件领取第二版英文电子版即可参与）

交流社区

有兴趣的读者可加golang交流群，大家一起交流。

关注公众号Go_Official_Blog 了解更多官方资讯（公众号中回复 `加群` 即可）。

翻译进度

持续更新中。。。。

规则&&Fork&&PR

- 章节命名规则：举例，第一章第一节，01.1.md,如果第一节下面还有分支，01.1.1,依次类推;
- 联系邮箱，取得电子版，获得安排的翻译章节，Fork分支，提交PR;
- 由多人审核后，合并

致谢

- 本书原作者：Mihalis Tsoukalos
- 参与翻译人员
    - Jeremy
    - calmbryan
    - newlife

授权许可

Go学习资料及社区（持续更新中。。。）

- Go By Example 英文网站
- Go By Example 中文网站
- GOCN Forum
- Go语言中文网
- Go walker 强大的Go在线API文档
- jsonTOGo 好用的json转go struct工具
- Go web框架beego
- 官方代码规范指导
- xorm支持 MySQL、PostgreSQL、SQLite3 以及 MsSQL
- mgoMongoDB 官方推荐驱动
- gorm全功能 ORM (无限接近) 支持 MySQL、PostgreSQL、SQLite3 以及 MsSQL

1 Go与操作系统

Go和操作系统

本章介绍Go语言的各种主题，这些主题初学者会发现很有用。有经验的Go开发者也可以把这章当做基础知识的复习。因为这些都是实践性的主题，所以最好的理解方法就是去实验他们。在这里，实验就是自己写代码，自己发现错误，并且从错误中学习。不要被错误信息和bug吓倒。

本章中，你将学到以下这些：

- Go语言的历史和未来
- Go的优势
- 编译Go代码
- 执行Go代码
- 下载和使用Go语言的外部包
- UNIX的标准输入，输出和错误
- 在屏幕上打印数据
- 获取用户输入
- 打印数据到标准错误
- 使用日志文件
- 使用Docker来编译和执行Go源码文件
- Go语言的错误处理

01.1 Go的历史

Go的历史

Go语言是一种现代的，多用途的开源编程语言，发布于2009年末。Go一开始的时候是Google的内部实验项目，受到很其他语言的启发，包括 C, Pascal, Alef和Oberon。Go语言的精神教父是专业的程序员Robert Griesemer, Ken Thomson, and Rob Pike.

他们为那些希望构建可靠，健壮，高效的软件系统的专业程序员设计了Go语言。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1.13。不过，即使你的Go版本比这个要高，本书的内容依然是不会过时的。

如果你是第一次安装Go语言，你可以从访问https://golang.org/dl/开始。但是，有很大的可能你的unix系统已经提供了一个Go语言的安装包，所以可以用你喜欢包管理器来安装它。

01.2 Go的未来

Go的未来

Go社区已经在讨论Go语言的下一个主版本，这个主版本将被称为Go 2 ，但是目前还没有什么定论。

Go 1团队希望Go 2更加的社区化。一般意义上来说，这是一个好主意，但是这也是很危险的，因为很多人试图在Go语言重大变化上发表意见，而这个项目一开始是由几个伟大的程序员设计和开发的内部项目。

01.3.2 什么是预处理器

01.3.1 Go是完美的么

Go是完美的么？

没有完美的编程语言，Go也不例外。然而，某些编程语言在某些编程领域比其他要好，或者我们更喜欢他们。就我个人而言，我不喜欢Java（me too），而且我以前喜欢C++，现在不喜欢了。作为一个编程语言，C++变得太复杂了，而Java的代码太难看了。
Go的一些不足之处：

- Go没有直接支持面向对象编程，这对那些以前以面向对象的方式来写代码的程序员可能是个问题。当然，你可以用组合来模拟继承。
- 对于一些人而言，Go永远不能取代C
- 对系统编程而言，C依然比其他语言都快，主要是因为unix是C写的

尽管如此，Go依然是一个十分优雅的编程语言，如果你花时间学习和使用它，它不会让你失望的。

01.3.3 godoc

godoc工具

Go语言自带很多可以使你的程序员生活更加容易的工具。其中一个就是godoc，它可以帮助你在没有网络连接的情况下查看现存包和函数的文档。
godoc工具即可以像一般命令行命令那样直接在终端显示结果，也可以在命令行下启动一个web服务器。启动web服务器的，你需要一个浏览器来查看Go的文档。
第一种方式就像用man命令一样，不过我们查询的是Go的函数和包。因此，为了获取 fmt包中 Printf() 函数的信息，你可以执行下面的命令：

```
$ go doc fmt.Printf
```

同样的，你用下面这个命令可以找到fmt包的信息：

```
$ go doc fmt
```

第二种需要执行godoc命令，并且带有**-http**参数：

```
$ godoc -http=:8001
```

上面命令中的数值，就是那个8001，是HTTP服务器监听的端口号。你可以使用任何你有权限，并且可用的值。然而，注意0到1023是被限制的，只有root用户可用，所以最好是避免使用这之间的一个，选个别的，并且这个端口没有被其他进程占用。
你也可以没有上面程序中那个**=**，用一个空格代替。下面这个命令和上面的命令是完全一样的：

```
$ godoc -http :8001
```

然后，你就可以打开浏览器，访问http://localhost:8001/pkg/,就可以浏览Go的文档了。

01.4 编译Go代码

编译Go代码

在这一节中你将会学习如何编译Go代码。好消息是你可以在命令行编译你的Go代码，不需要图形界面。更好的是，Go不关心代码文件的名字，只要你的包名是main,并且包中有且只有一个函数叫main。这是因为main()函数是程序执行开始的地方。因此，在一个项目中不能有多个函数的名字是main()。

我们首先编译一个叫aSourceFile.go的文件，这个文件中包含以下内容：

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

注意，Go社区倾向源码文件的名字是source_file.go,而不是aSourceFile.go。不管你使用哪种，请保持一致。

为了编译aSourceFile.go这个文件，并且生成一个静态链接的可执行文件，你需要执行下面的命令：

```
$ go build aSourceFile.go
```

然后，你就有了一个叫做aSourceFile的可执行文件:

```
$ file aSourceFile
aSourceFile: Mach-O 64-bit executable x86_64
$ ls -l aSourceFile
-rwxr-xr-x 1 mtsouk staff 2007576 Jan 10 21:10 aSourceFile $ ./aSourceFile
This is a sample Go program!
```

这个文件体积这么大的主要原因是它是静态链接的，运行的时候不需要外部库。

2 理解 Go 的内部构造

理解 Go 的内部构造

你在上一章中学到的所有 Go 功能都非常方便，你将一直使用它们。但是，更有价值的是看到和理解这背后发生的事情以及 Go 在幕后的操作方式。

在本章中，你将学习 Go 垃圾收集器及其工作方式。此外，你还将了解如何从 Go 程序中调用 C 代码，这在某些情况下可能是必不可少的。当然，因为 Go 是一种非常强大的编程语言，你可能不需要经常使用此功能。

此外，你还将学习如何从 C 代码中调用 Go 函数，以及如何使用 `panic()` 和 `restore()` 函数和 `defer` 关键字。

本章将会涵盖：

- Go 编译器
- Go 的垃圾回收是如何工作的
- 如何检查垃圾收集器的操作
- 从 Go 代码调用 C 代码
- 从 C 代码调用 Go 代码
- `panic()` 和 `recover()` 函数
- `unsafe` 包
- 方便但棘手的 `defer` 关键字
- `strace(1)` Linux 工具
- `dtrace(1)` 实用程序，可以在 FreeBSD 系统(包括 macOS Mojave)中找到
- 查找有关你的 Go 环境的信息
- Go 创建的节点树
- 从 Go 创建 WebAssembly 代码
- Go 汇编器

Go 编译器

Go 编译器

Go 编译器在 go 工具的帮助下执行，该工具不仅生成可执行文件，还执行更多的操作。

> Tip: 本节中使用的 `unsafe.go` 文件不包含任何内容特殊代码–所提供的命令将在每个有效的 Go 源文件上运行。

你可以通过 `go tool compile` 命令编译 Go 源代码。你将得到一个目标文件，这将是一个以 `.o` 扩展结尾的文件。在 macOS Mojave 环境下执行的下一个命令的输出中对此进行了说明：

```
$ go tool compile unsafe.go
$ ls -l unsafe.o
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:39 unsafe.o
$ file unsafe.o
unsafe.o: current ar archive
```

目标文件是包含目标代码的文件，该目标代码是可重定位格式的机器代码，在大多数情况下，这些代码不能直接执行。可重定位格式的最大优点是在链接阶段它只需要较少的内存。
如果你在运行 `go tool compile` 的时候加上了 `-pack` 标志，你将会得到压缩文件而非目标文件：

```
$ go tool compile -pack unsafe.go
$ ls -l unsafe.a
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:40 unsafe.a
$ file unsafe.a
unsafe.a: current ar archive
```

压缩文件是包含一个或多个文件的二进制文件，主要是
用于将多个文件分组为一个文件。这些格式之一是 `ar` ，它是 Go 使用的格式。
你可以像这样打印出一个 `.a` 压缩文件的内容：

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

`go tool compile` 另一个真正有价值的命令行标志是 `-race` ，它使你能够检测竞争条件。你将在第10章Go的并发性-高级主题中了解有关竞争条件以及为什么要避免竞争条件的更多信息。
本章的最后讨论汇编语言和节点树的时候，你还会学到一个更多有用的 `go tool compile` 命令。不过目前

你可以尝试一下下面的命令：

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

你可能会发现，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。

Go 的垃圾回收

Go 的垃圾回收

垃圾回收是释放未使用的内存空间的过程。换句话说，垃圾收集器查看哪些对象不在范围内并且无法再引用，并释放它们消耗的内存空间。此过程在 Go 程序运行时(而不是在程序执行之前或之后)以并行方式发生。 Go 垃圾收集器实现的文档指出以下内容：

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 goroutine 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"
    "runtime"
    "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() {
  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(5 * time.Second)
        }
    }
    printStats(mem)
}
```

在 macOS Mojave 上面 `gColl.go` 的输出如下：

```
$ go run gColl.go
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
-----
```

尽管你不会一直检查 Go 垃圾收集器的操作，但是从长远来看，能够观察它在运行缓慢的应用程序上的运行方式可以节省大量时间。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，将会很值得。

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+0.065+0.018 ms clock,
  0.021+0.040/0.057/0.003+0.14 ms cpu, 47->47->0 MB, 48 MB goal, 8 P
gc 17 @30.103s 0%: 0.004+0.080+0.019 ms clock,
  0.033+0/0.076/0.071+0.15 ms cpu, 95->95->0 MB, 96 MB goal, 8 P
```

前面的输出告诉我们有关垃圾回收过程中堆大小的更多信息。因此，让我们以 `47 -> 47 -> 0 MB` 为例。第一个数字是垃圾收集器即将运行时的堆大小。第二个值是垃圾收集器结束其操作时的堆大小。最后一个值是活动堆的大小。

三色算法

三色算法

Go 基于三色算法的垃圾回收器。

> Tip: 请注意，三色算法不是 Go 语言独有的，可以在其他编程语言中使用。

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)。
该算法背后的核心思想来自Edsger W.Dijkstra，Leslie Lamport，A.J.Martin，C.S.Scholten 和 E.F.M.Steffens，并首次在名为On-the-fly Garbage Collection：An Exercise in Cooperation的论文中得到了说明。
三色标记和清除算法背后的主要原理是，它根据堆的颜色将堆的对象划分为三个不同的集合，这些颜色由算法分配。现在是时候讨论每种颜色集的含义了。保证黑色集合的对象没有指向白色集合的任何对象的指针。
但是，白色集合的对象可以具有指向黑色集的对象的指针，因为这对垃圾收集器的操作没有影响。灰色集合的对象可能具有指向白色集合的某些对象的指针。最后，白色集合的对象是垃圾收集的候选对象。
请注意，没有对象可以直接从黑色集合转到白色集合，这允许算法进行操作并能够清除白色集合上的对象。此外，黑色集合的任何对象都不能直接指向白色集合的对象。
此后，垃圾收集器选择一个灰色对象，使其变为黑色，然后开始查看该对象是否具有指向白色集合中其他对象的指针。这意味着在扫描灰色对象以寻找指向其他对象的指针时，它会被涂成黑色。如果该扫描发现该特定对象具有一个或多个指向白色对象的指针，则会将该白色对象放入灰色集合中。只要灰色集合中存在对象，此过程就会持续进行。之后，白色集合中的对象将无法访问，并且它们的存储空间可以被重用。因此，在这一点上，我们说白色集合里的元素被垃圾回收了。

> Tip: 请注意，如果在垃圾回收周期的某个时刻灰色集合的对象变得不可访问，则不会在该垃圾回收周期中将其收集，而是在下一个垃圾回收中将其收集！尽管这不是最佳情况，但还不错。

在此过程中，正在运行的应用程序称为修改器(mutator)。 mutator 运行一个名为**写屏障(write barrier)**的小函数，每次修改堆中的指针时都会执行该函数。如果修改了堆中某个对象的指针，这意味着该对象现在可以访问，则写入屏障将其着色为灰色并将其放入灰色集合中。

> 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！

结果堆显示为已连接对象的图形，如图 2.1 所示，该图演示了垃圾回收周期的单个阶段。

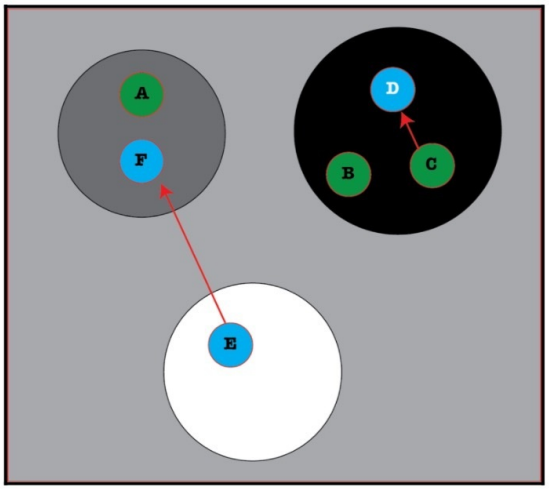


Figure 2.1: The Go garbage collector represents the heap of a program as a graph

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go代码。你可以进一步了解这个，如果你想了解更多关于垃圾回收操作的东西。

> Tip: 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。

有关 Go 垃圾收集器操作的更多信息

有关 Go 垃圾收集器操作的更多信息

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。

> Tip: 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。

Maps, silces 与 Go 垃圾回收器

Maps, silces 与 Go 垃圾回收器

在本节中，我将向你提供一些示例，这些示例说明了为什么你对于垃圾收集器的操作应保持谨慎。本节的重点是要了解，存储指针的方式对垃圾收集器的性能有很大影响，尤其是在处理大量指针时。

> Tip: 所提供的示例使用了指针，切片和映射，它们都是原生 Go 数据类型。你将在第 3 章，使用基本 Go 数据类型了解有关指针，切片和映射的更多信息。

使用 slice

这在一节的例子中我们将使用slice来存储大量的结构体数据。每一个结构体数据存储两个整数值。

`sliceGC.go` 中的 Go 代码如下：

```
package main
import (
  "runtime"
)
type data struct {
  i, j int
}
func main() {
  var N = 40000000
  var structure []data
  for i := 0; i < N; i++ {
    value := int(i)
    structure = append(structure, data{value, value})
  }
  runtime.GC()
  _ = structure[0]
}
```

最后一条语句 `(_ = structure[0])` 用于防止垃圾回收器过早地垃圾收集结构体变量，因为未在 `for` 循环之外对其进行引用或使用。随后的三个 Go 程序将使用相同的技术。除此重要细节外， `for` 循环用于将所有值放入存储在 slice 中的结构中。

使用 pointers 操作 map

在本小节中，我们将使用映射将所有指针存储为整数。该程序的名称为 `mapStar.go` ，其中包含以下 Go 代码：

```
package main
```

```
import (
  "runtime"
)

func main() {
  var N = 40000000
  myMap := make(map[int]*int)
  for i := 0; i < N; i++ {
    value := int(i)
    myMap[value] = &value
  }
  runtime.GC()
  _ = myMap[0]
}
```

存储整数指针的 map 的名称为 `myMap`， `for` 循环用于将整数值放入 map。

不使用 pointers 操作 map

在本小节中，我们将使用一个存储无指针纯值的 map， `mapNoStar.go` 的 Go 代码如下：

```
package main
  import (
    "runtime"
  )
func main() {
  var N = 40000000
  myMap := make(map[int]int)
  for i := 0; i < N; i++ {
    value := int(i)
    myMap[value] = value
  }
  runtime.GC()
  _ = myMap[0]
}
```

和之前一样，使用 for 循环将整数值放入 map 中。

分割 map

本小节的实现会将 map 拆分为 maps，这也称为分片。本小节的程序另存为 `mapSplit.go` ，将分两部分介绍。 mapSplit.go 的第一部分包含以下 Go 代码：

```
package main
import (
  "runtime"
)
func main() {
```

```
    var N = 40000000
    split := make([]map[int]int, 200)
```

这是定义哈希的哈希值的地方。

第二段代码如下：

```
    for i := range split {
      split[i] = make(map[int]int)
    }
    for i := 0; i < N; i++ {
      value := int(i)
      split[i%200][value] = value
    }
    runtime.GC()
    _ = split[0][0]
  }
```

这次，我们使用了两个 `for` 循环：一个用于创建哈希散列的 `for` 循环，以及另一个用于在哈希哈希中存储所需数据。

Comparing the performance of the presented techniques

由于这四个程序都使用巨大复杂的数据结构，因此它们正在消耗大量内存。占用大量内存空间的程序会更频繁地触发 Go 垃圾收集器。因此，在本小节中，我们将使用 `time(1)` 命令比较这四个实现中每个实现的性能。输出中重要的不是确切的数字，而是四种不同方法之间的时间差:

```
    $ time go run sliceGC.go
    real  1.50s
    user  1.72s
    sys   0.71s
    $ time go run mapStar.go
    real  13.62s
    user  23.74s
    sys   1.89s
    $ time go run mapNoStar.go
    real  11.41s
    user  10.35s
    sys   1.15s
    $ time go run mapSplit.go
    real  10.60s
    user  10.01s
    sys   0.74s
```

因此，事实证明，maps 会减慢 Go 垃圾收集器的速度，而 slices 则可以更好地协作。这里应该注意，这不是 map 的问题，而是 Go 垃圾收集器工作方式的结果。但是，除非你要处理的是存储大量数据的 map，否则此问

题在你的程序中将不会变得明显。

> Tip: 你将在第 11 章代码测试，优化和性能分析中学习有关基准测试的更多信息。此外，在第 3 章使用基本 Go 数据类型中，你将学到更专业的方法来测量 Go 中执行命令或程序所花费的时间。

你了解垃圾收集已经足够多了；下一节的主题将是 `unsafe code` 和 `unsafe` Go package。

Unsafe code

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！

Unsafe code 的使用将在 unsafe.go 程序中进行说明，该程序分为三个部分。

`unsafe.go` 第一部分

```
package main
import (
  "fmt"
  "unsafe"
)
```

你会注意到，为了使用 unsafe code，你将需要导入不安全的标准 Go package。

第二部分 Go 代码：

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

请注意此处使用 `unsafe.Pointer()` 函数，该函数使我们自己承担创建一个名为 `p2` 的 `int32` 指针的风险，该指针指向一个名为 `value` 的 `int64` 变量，可以使用 `p1` 指针对其进行访问。任何 Go 指针都可以转换为 `unsafe.Pointer` 。

> Tip: 类型为 `unsafe.Pointer` 的指针可以覆盖 Go 的类型系统。这无疑是很快的，但是如果使用不正确或不小心，也会带来危险。此外，它使开发人员可以更好地控制数据。

最后一部分 Go 代码：

```
  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)
```

```
}
```

> 你可以取消引用指针，并使用星号(*)获取，使用或设置其值。

如果你执行 `unsafe.go` ，你会得到以下的输出：

```
$ go run unsafe.go
*p1:  5
*p2:  5
54341234123124312l2
*p2:  -930866580
54341234
*p2:  54341234
```

此输出告诉我们什么？它告诉我们 32 位指针不能存储 64 位整数。

正如你将在下一节中看到的那样， `unsafe` 包的功能可以用内存做更多有趣的事情。

有关 unsafe 包

有关 unsafe 包

现在你已经看到了 `unsafe` 包在起作用，现在该讨论更多有关使其成为特殊程序包的方法了。首先，如果你查看 `unsafe` 包的源代码，你可能会有些惊讶。在使用 Homebrew(https://brew.sh/)安装的 Go 版本为 1.11.4 的 macOS Mojave 系统上， `unsafe` 包的源代码位于 `/usr/local/Cellar/go/1.11.4/libexec/src/unsafe/unsafe.go` ，不包含注释，它的内容如下：

```
$ cd /usr/local/Cellar/go/1.11.4/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
```

那么， `unsafe` 包的其余 Go 代码在哪里？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。

> Tip: 许多系统库，例如 `runtime`,`syscall` 和 `os` 会经常使用到 `unsafe` 库

另一个 usafe 包的例子

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.Sizeof(array[0])
  for i := 0; i < len(array)-1; i++ {
    pointer = (*int)(unsafe.Pointer(memoryAddress))
    fmt.Print(*pointer, " ")
    memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])
  }
```

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。

`unsafe.Sizeof(array[0])` 的值使你可以进入数组的下一个元素，因为这是每个数组元素占用的内存。因此，该值将在 for 循环的每次迭代中添加到 `memoryAddress` 变量中，从而使你可以获取下一个数组元素的内存地址。 `*pointer` 表示法取消引用指针并返回存储的整数值。

第三部分代码：

```
fmt.Println()
pointer = (*int)(unsafe.Pointer(memoryAddress))
fmt.Print("One more: ", *pointer, " ")
memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])
fmt.Println()
```

在最后一部分中，我们尝试使用指针和内存地址访问数组中不存在的元素。由于使用了 `unsafe` 包，Go 编译器无法捕获此类逻辑错误，因此将返回不正确的内容。

执行 `moreUnsafe.go` 将会输出：

```
$ go run moreUnsafe.go
0 1 -2 3 4
One more: 824634208008
```

你刚刚使用指针访问了 Go 数组的所有元素。但是，这里的真正问题是，当你尝试访问无效的数组元素时，程序没有抛出错误，而是返回了一个随机数。

从 Go 调用 C 代码

从 Go 调用 C 代码

与 C 相比，尽管 Go 的编程体验更好，但是 C 仍然是一种非常有用的编程语言。这意味着在某些情况下(例如，使用数据库或使用 C 编写的设备驱动程序)仍需要使用 C，这意味着你将需要在 Go 项目中使用 C 代码。

> Tip: 如果发现你在同一项目中多次使用此功能，则可能需要重新考虑你的方法或选择的编程语言是不是有问题。

在同一文件用 Go 调用 C 代码

在同一文件用 Go 调用 C 代码

从 Go 程序调用 C 代码的最简单方法是将 C 代码包含在 Go 源文件中。这需要特殊处理，但是速度很快，而且没有那么困难。

包含 C 和 Go 代码的 Go 源文件的名称为 `cGo.go` ，将分为三部分。

第一部分的代码：

```
package main
//#include <stdio.h>
//void callC() {
//    printf("Calling C code!\n");
//}
import "C"
```

> Tip: 如你所看到的，C 代码包含在 Go 程序的注释中。但是，由于使用了 `c` Go package， `go` 知道如何处理此类注释。

第二部分代码：

```
import "fmt"

func main() {
```

所有其他的包都必须被单独导入。

最后一部分代码：

```
  fmt.Println("A Go statement")
  C.callC()
  fmt.Println("Another Go Statement!")
}
```

为了执行 `callC()` c 函数，你需要像这样 `C.callC()` 调用.

执行 `cGo.go` 你会得到这样的输出：

```
  $ go run cGo.go
  A Go statement!
  Calling C code!
  Another Go statement!
```

在单独的文件用 Go 调用 C 代码

在单独的文件用 Go 调用 C 代码

现在让我们继续讨论当 C 代码位于单独的文件中时如何从 Go 程序中调用 C 代码。
首先，我将解释用程序解决的假想的问题。然后我们需要使用已经实现好的两个 C 函数，这些函数我们不希望或无法在 Go 中重写。

C 代码部分

本小节将为你提供示例的 C 代码。它有两个文件： `callC.h` 和 `callC.c` 。文件 `(callC.h)` 包含以下代码：

```
#ifndef CALLC_H
#define CALLC_H
void cHello();
void printMessage(char* message);
#endif
```

文件 `(callC.c)` 包含以下代码：

```
#include <stdio.h>
#include "callC.h"
void cHello() {
  printf("Hello from C!\n");
}
void printMessage(char* message) {
  printf("Go send me %s\n", message);
}
```

`callC.c` 和 `callC.h` 文件都存储在单独的目录中，我们把该目录命名为 `callClib` 。但是，你可以使用你想使用的任何目录名称。

> Tip: 只要你使用正确的类型和数量的参数调用正确的 C 函数，那么的 C 代码的具体实现并不重要。C 代码中没有任何内容告诉我们它将在 Go 程序中使用。You should look at the Go code for the juicy part.

Go 代码部分

本小节将为你提供示例的 Go 源代码，该源代码名为 `callC.go` ，并将分三部分向你介绍。
第一部分的代码：

```
package main
// #cgo CFLAGS: -I${SRCDIR}/callClib
```

```
// #cgo LDFLAGS: ${SRCDIR}/callC.a
// #include <stdlib.h>
// #include <callC.h>
import "C"
```

整个 Go 源文件中最重要的 Go 语句是使用单独的 `import` 语句包含 C 包。但是， `C` 是一个虚拟的 Go 包，它只是告诉 `go build` 在 go 编译器处理文件之前使用 `cgo` 工具对其输入文件进行预处理。你仍然可以看到你需要使用注释来告知 Go 程序有关 C 代码的信息。在这种情况下，你告诉 `callC.go` 在哪里可以找到 `callC.h` 文件以及在哪里可以找到我们将在一段时间内创建的 `callC.a` 库文件，就像以 `#cgo` 开头的一段代码。

第一部分的代码：

```
import (
  "fmt"
  "unsafe"
)
func main() {
  fmt.Println("Going to call a C function!")
  C.cHello()
```

最后一部分代码：

```
  fmt.Println("Going to call another C function!")
  myMessage := C.CString("This is Mihalis!")
  defer C.free(unsafe.Pointer(myMessage))
  C.printMessage(myMessage)
  fmt.Println("All perfectly done!")
}
```

为了从 Go 将字符串传递给 C 函数，你将需要使用 `C.CString()` 创建一个 C 字符串。此外，你将需要一个 `defer` 语句，以便在不再需要 C 字符串时释放其存储空间。 defer 语句包括对 `C.free()` 和 `unsafe.Pointer()` 的调用。

在下一部分中，你将看到如何编译和执行 `callC.go` 。

混合 Go 和 C 代码

现在你已经有了 C 代码和 Go 代码，现在该学习下一步以执行调用 C 代码的 Go 文件了。

由于所有的关键信息都包含在 Go 文件中了，所以你唯一需要做的就是编译 C 代码以生成一个库:

```
$ ls -l callClib/
total 16
-rw-r--r--@ 1 mtsouk  staff  162 Jan 10 09:17 callC.c
-rw-r--r--@ 1 mtsouk  staff   89 Jan 10 09:17 callC.h
```

```
$ gcc -c callClib/*.c
$ ls -l callC.o
-rw-r--r--  1 mtsouk  staff  952 Jan 22 22:03 callC.o
$ file callC.o
callC.o: Mach-O 64-bit object x86_64
$ /usr/bin/ar rs callC.a *.o
ar: creating archive callC.a
$ ls -l callC.a
-rw-r--r--  1 mtsouk  staff  4024 Jan 22 22:03 callC.a
$ file callC.a
callC.a: current ar archive
$ rm callC.o
```

之后，你将在与 callC.go 文件相同的目录中拥有一个名为 `callC.a` 的文件。 gcc 可执行命令是C 编译器的名称。

现在，你可以使用 Go 代码编译文件创建一个新的可执行文件：

```
$ go build callC.go
$ ls -l callC
-rwxr-xr-x  1 mtsouk  staff  2403184 Jan 22 22:10 callC
$ file callC
callC: Mach-O 64-bit executable x86_64
```

执行 `callC` 可执行文件，你将得到输出：

```
$ ./callC
Going to call a C function!
Hello from C!
Going to call another C function!
Go send me This is Mihalis!
All perfectly done!
```

> Tip: 如果要调用少量的 C 代码，强烈建议 C 和 Go 代码在单个 Go 文件中。但是，如果你要进行更复杂和更高级的操作，则首选静态 C 库。

从 C 调用 Go 代码

从 C 调用 Go 代码

也可以从C代码中调用Go函数。因此，本节将为你提供一个小示例，它将从C程序中调用两个Go函数。 Go包将转换为C共享库，该库将在C程序中使用。

Go 包

Go 包

本小节将向你提供将在 C 程序中使用的 Go package 的代码。 Go package 的名称必须是 `main` ，但其文件名可以是你想要的任何名称；在这我们将文件命名为为 `useByC.go` ，并分为三部分。

> Tip: 你将在第 6 章-你可能不了解 Go package 和 Go 函数中了解有关 Go package 的更多信息。

第一部分代码：

```
package main
import "C"
import (
  "fmt"
)
```

正如我之前所说，必须将 Go 包命名为 `main` 。你还需要在 Go 代码中导入 C 包。
第二部分代码：

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

必须将 C 代码准备调用的 Go 函数导出。这意味着你应在导出之前放置一个以 `// export` 开头的注释行。在 `// export` 的后面你将需要输入函数的名称，因为这是 C 代码将使用的名称。
最后一部分代码：

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

func main() {
}
```

`usedByC.go` 的 `main()` 函数不需要代码，因为它不会被导出，因此不会被 C 程序使用。此外，由于还希望导出 `Multiply()` 函数，因此需要写上 `// export Multiply` 。
之后，你需要通过执行以下命令从 Go 代码生成 C 共享库：

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

前面的命令将生成两个名为 `usedByC.h` 和 `usedByC.o` 的文件：

```
$ ls -l usedByC.*
-rw-r--r--@ 1 mtsouk  staff      204 Jan 10 09:17 usedByC.go
-rw-r--r--  1 mtsouk  staff     1365 Jan 22 22:14 usedByC.h
-rw-r--r--  1 mtsouk  staff  2329472 Jan 22 22:14 usedByC.o
$ file usedByC.o
usedByC.o: Mach-O 64-bit dynamically linked shared library x86_64
```

你不应对 usedByC.h 进行任何更改。

C 代码

C 代码

可以在 `willUseGo.c` 源文件中找到相关的 C 代码，该文件分为两部分。接下来是 `willUseGo.c` 的第一部分：

```
#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();
```

如果你已经知道 C，那么你将理解为什么需要包含 usedByC.h；这就是 C 代码了解库的可用功能的方式。

第二部分的代码：

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

`GoInt p` 表示使用 `(int)p` 转换为 C 整数变量，是从 Go 函数获取整数值的必要方式。

在 macOS Mojave 机器上编译并执行 `willUseGo.c` 将输出：

```
$ 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

`defer` 关键字将函数的执行推迟到周围的函数返回之前，这在文件输入和输出操作中被广泛使用，因为它使你不必记住何时关闭打开的文件。使用 `defer` 关键字，可以将关闭已打开文件的函数调用放在打开该文件的函数调用附近。在第 8 章“告诉 UNIX 系统该做什么”中你将学习 `defer` 在与文件相关的操作，在本节将介绍 `defer` 的两种不同用法。你还将在讨论 `panic()` 和 `restore()` 内置 Go 函数的部分以及与日志记录相关的部分中看到 `defer` 。

重要的是要记住，延迟函数在返回周围函数后以后进先出(LIFO)的顺序执行。简而言之，这意味着如果在同一个周围函数中先延迟函数 `f1()` ，然后再延迟函数 `f2()` ，然后再延迟函数 `f3()` ，则当周围函数即将返回时，将执行函数 `f3()` ，然后 `f2()` ，最后才是 `f3()` 。

由于对 `defer` 的定义尚不清楚，我认为你可以通过查看 Go 代码和 `defer.go` 程序的输出来更好地理解 `defer` 的用法，该代码将分为三部分。

第一部分的代码：

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

除了 import 块，前面的 Go 代码还实现了一个名为 `d1()` 的函数。包含有 `for` 循环和 `defer` 语句，该语句将执行 3 次。

第二部分的代码：

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

在这部分代码中，你可以看到另一个函数的实现，该函数名为 `d2()` 。 `d2()` 函数还包含一个 `for` 循环和

一个 `defer` 语句，它们还将执行 3 次。但是，这一次，将 defer 关键字应用于匿名函数，而不是单个 `fmt.Print()` 语句。此外，匿名函数不带任何参数。

最后一部分的代码：

```
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()
}
```

除了调用 `d1(),d2()和d3()` 函数的 `main()` 函数之外，你还可以看到 `d3()` 函数的实现，该函数具有一个 `for` 循环，该循环使用匿名的 `defer` 关键字功能。但是，这一次，匿名函数需要一个名为 `n` 的整数参数。Go 代码告诉我们， `n` 参数从 `for` 循环中使用的 `i` 变量中获取其值。

执行 `defer.go` 将会得的输出：

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

你很可能会发现生成的输出非常复杂且难以理解，这证明如果代码不清晰，则操作和使用 `defer` 的结果可能会很棘手。

我们将从 `d1()` 函数生成的第一行输出 `(1 2 3)` 开始。 `d1()` 中 `i` 的值依次为 `3、2和1` 。

`d1()` 中延迟的函数是 fmt.Print()语句；结果，当 `d1()` 函数将要返回时，你将以相反的顺序获得 for 循环的 i 变量的三个值，因为延迟的函数以 LIFO 顺序执行。

现在，让我解释一下 `d2()` 函数产生的第二行输出。我们得到三个零而不是 `1 2 3` 真是奇怪。但是，原因很简单。

在 `for` 循环结束之后， `i` 的值为 0，因为正是 `i` 的值使 `for` 循环终止。但是，这里的棘手之处在于，因为没有参数，所以在 for 循环结束后评估了延迟的匿名函数，这意味着对于 `i` 值为 0，它会被评估三次，因此生成了输出。这种令人困惑的代码可能会导致在你的项目中创建烦人的错误，因此请避免使用它。

最后，我们将讨论输出的第三行，它是由 `d3()` 函数生成的。由于匿名函数的参数，每次推迟匿名函数时，它

都会获取并因此使用 `i` 的当前值。结果，匿名函数的每次执行都需要处理不同的值，因此产生的输出也不同。所以你应该清楚，使用 `defer` 的最佳方法是第三种方法，它出现在 `d3()` 函数中，这种方法以易于理解的方式在匿名函数中传递了所需的变量。

用 defer 打印日志

用 defer 打印日志

本节将介绍与日志记录相关的 `defer` 应用。该技术的目的是帮助你更好地组织功能的日志记录信息。该试例程序的名称是 `logDefer.go` ，它将分为三个部分。

第一部分代码：

```
package main
import (
  "fmt"
  "log"
  "os"
)

var LOGFILE = "/tmp/mGo.log"

func one(aLog *log.Logger) {
  aLog.Println("-- FUNCTION one ------")
  defer aLog.Println("-- FUNCTION one ------")

  for i := 0; i < 10; i++ {
    aLog.Println(i)
  }
}
```

名为 `one()` 的函数正在使用 `defer` 来确保第二个 `aLog.Println()` 调用将在该函数即将返回之前执行。因此，该函数的所有日志消息都将被嵌入在打开 `aLog.Println()` 和关闭 `aLog.Println()` 之间。这样，在日志文件中查看该功能的日志消息就会容易得多。

第二部分代码：

```
func two(aLog *log.Logger) {
  aLog.Println("---- FUNCTION two")
  defer aLog.Println("FUNCTION two ------")

  for i := 10; i > 0; i-- {
    aLog.Println(i)
  }
}
```

名为 `two()` 的函数仍然使用 `defer` 方便地对其日志消息进行分组。但是，这次 `two()` 使用的消息格式与函数 `one()` 略有不同，不过这完全取决于你。

最后一部分代码：

```
func main() {
  f, err := os.OpenFile(LOGFILE,os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
  if err != nil {
    fmt.Println(err)
    return
  }
  defer f.Close()

  iLog := log.New(f, "logDefer ", log.LstdFlags)
  iLog.Println("Hello there!")
  iLog.Println("Another log entry!")

  one(iLog)
  two(iLog)
}
```

执行 `logDefer.go` 将不会在控制台产生输出。但是，查看程序正在使用的日志文件 `/tmp/mGo.log` 的内容，你就知道 `defer` 到底有多方便了：

```
$ cat /tmp/mGo.log
logDefer 2019/01/19 21:15:11 Hello there!
logDefer 2019/01/19 21:15:11 Another log entry!
logDefer 2019/01/19 21:15:11 -- FUNCTION one ------
logDefer 2019/01/19 21:15:11 0
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 -- FUNCTION one ------
logDefer 2019/01/19 21:15:11 ---- FUNCTION two
logDefer 2019/01/19 21:15:11 10
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 FUNCTION two ------
```

Panic 和 Recover

Panic 和 Recover

本节将向你介绍上一章中首先提到的技巧。该技术涉及 `panic()` 和 `recover()` 函数的使用，将在 `panicRecover.go` 中进行介绍，同样分为三部分。

第一段代码：

```
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` ，此部分还包括 `a()` 函数的实现。函数 `a()` 的最重要部分是延迟代码块，该代码块实现了一个匿名函数，当调用 `panic()` 时将调用该匿名函数。

第二段代码：

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

最后一段代码：

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

执行 `panicRecover.go` 产生如下输出：

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

这个输出结果让人有点惊讶。因为，从输出中可以看到， `a()` 函数没有正常结束，它的最后两个语句没有得到执行：

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

不过，好在 `panicRecover.go` 程序会按我们的意愿结束而不会发生崩溃，因为在 `defer` 中使用的匿名函数控制了异常情况。还要注意，函数 `b()` 对函数 `a()` 一无所知；但是，函数 `a()` 包含处理函数 `b()` 异常情况的 Go 代码。

单独使用 Panic 函数

单独使用 Panic 函数

你也可以单独使用 `panic()` 函数，而无需和recover配合，该小节将使用 `justPanic.go` 的 Go 代码显示其结果，该代码分为两部分。

第一部分代码：

```
package main
import (
  "fmt"
  "os"
)
```

使用 `panic()` 不要求引入任何的 Go package。

第二部分代码：

```
func main() {
  if len(os.Args) == 1 {
    panic("Not enough arguments!")
  }
  fmt.Println("Thanks for the argument(s)!")
}
```

如果你的 Go 程序没有接收一个命令行参数，它将调用 `panic()` 函数。 `panic()` 函数需要一个参数，这是你要在屏幕上打印的错误消息。

在 macOS Mojave 机器上执行 `justPanic.go` 将输出：

```
$ go run justPanic.go
panic: Not enough arguments!
goroutine 1 [running]:
main.main()
  /Users/mtsouk/ch2/code/justPanic.go:10 +0x91
exit status 2
```

因此，单独使用 `panic()` 函数将终止 Go 程序，而不会给你处理异常机会。因此，组合使用 `panic()` 和 `recovery()` 比仅使用 `panic()` 更实用且显得专业。

> Tip: `panic()` 函数的输出类似于日志包中 `Panic()` 函数的输出。但是， `panic()` 函数不会将任何内容发送到 UNIX 计算机的日志记录服务。

两个好用的 UNIX 工具

两个好用的 UNIX 工具

有时 UNIX 程序由于某种未知原因而失败或无法正常运行，并且你想找出原因，但不想重写代码并添加大量调试语句。

本节将介绍两个命令行实用程序，使你可以查看由可执行文件执行的 C 系统调用。这两个工具的名称分别为 `strace()` 和 `dtrace()` ，它们使你可以检查程序的运行情况。

> Tip: 请记住，所有在 UNIX 计算机上运行的程序最终都将使用 C 系统调用来与 UNIX 内核进行通信并执行大部分任务。

尽管这两个工具都可以使用 `go run` 命令，但是如果你首先使用 `go build` 创建可执行文件并使用该文件，则得到的无关输出会更少。发生这种情况的主要原因是 `go run` 在实际运行 Go 代码之前会生成各种临时文件。

strace

strace

`strace(1)` 命令行实用程序允许你跟踪系统调用和信号。由于 `strace(1)` 仅适用于 Linux 计算机，因此本节将使用 Debian Linux 计算机展示 `strace(1)` 。

`strace(1)` 生成的输出如下所示：

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 15 vars */]) = 0
brk(0)                                  = 0x186c000
fstat(3, {st_mode=S_IFREG|0644, st_size=35288, ...}) = 0
```

`strace(1)` 输出显示每个系统调用及其参数和返回值。请注意，在 UNIX 世界中，返回值为 0 是一件好事。为了处理二进制文件，你需要将 `strace(1)` 命令放在要处理的可执行文件的前面。但是，你将需要自己解释输出，以便从中得出有用的结论。好消息是，像 `grep(1)` 这样的工具可以为你提供你真正想要的输出：

```
$ strace find /usr 2>&1 | grep ioctl
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59c50) = -1 ENOTTY (Inappropriate ioctl for device)
ioctl(1, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59be0) = -1 ENOTTY (Inappropriate ioctl for device)
```

与 `-c` 命令行选项一起使用时， `strace(1)` 工具可以为每个系统调用打印计数时间，调用信息和错误信息：

```
$ strace -c find /usr 1>/dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------
82.88     0.063223           2      39228             getdents
16.60     0.012664           1      19587             getdents
0.16      0.000119           0      19618             13 open
```

由于普通程序输出以标准输出打印，而 `strace(1)` 的输出以标准错误打印，因此上一条命令将丢弃所检查命令的输出，并显示 `strace(1)` 的输出。从输出的最后一行可以看到， `open(2)` 系统调用被调用了 19,618 次，产生了 13 个错误，并且花费了整个命令执行时间的 0.16%大约 0.000119 秒。

dtrace

dtrace

尽管诸如 `strace(1)` 和`truss(1)之类的调试实用程序可以跟踪进程产生的系统调用，但是它们可能很慢，因此不适合解决繁忙的 UNIX 系统上的性能问题。另一个名为DTrace的工具可让你在系统范围内查看幕后发生的情况，而无需修改或重新编译任何内容。它还使你可以在生产环境上工作，并动态监视正在运行的程序或服务器进程，同时又不会造成很大的开销。

> Tip: 尽管有 `dtrace(1)` 有 Linux 版本，但 `dtrace(1)` 工具在 macOS 和其他 FreeBSD 变体上效果最佳。

本小节将使用 macOS 自带的 `dtruss(1)` 命令行实用程序，它只是一个 `dtrace(1)` 脚本，该脚本显示了进程的系统调用，使我们不必编写 `dtrace(1)` 代码。请注意， `dtrace(1)和` dtruss(1)`都需要 root 权限才能运行。

```
$ sudo dtruss godoc
ioctl(0x3, 0x80086804, 0x7FFEEFBFEC20)          = 0 0
close(0x3)          = 0 0
access("/AppleInternal/XBS/.isChrooted\0", 0x0, 0x0) = -1 Err#2
thread_selfid(0x0, 0x0, 0x0) = 1895378 0
geteuid(0x0, 0x0, 0x0) = 0 0
getegid(0x0, 0x0, 0x0) = 0 0
```

`dtruss(1)` 的工作方式与 `strace(1)` 实用程序相同。与 `strace(1)` 类似，当与 `-c` 参数一起使用时， `dtruss(1)` 将打印系统调用计数：

```
$ sudo dtruss -c go run unsafe.go 2>&1
CALL                                        COUNT
access                                          1
bsdthread_register                              1
getuid                                          1
ioctl                                           1
issetugid                                       1
kqueue                                          1
write                                           1
mkdir                                           2
read                                          244
kevent                                        474
fcntl                                         479
lstat64                                       553
psynch_cvsignal                               649
```

```
psynch_cvwait                                   654
```

上面的输出将迅速告知你 Go 代码中的潜在瓶颈，甚至可以帮你比较两个不同的命令行程序的性能。

> Tip: 诸如 `strace(1)` ，dtrace(1)和 dtruss(1)`之类的实用工具需要一段时间适应熟悉，但是这样的工具可以使我们的生活变得更加轻松和舒适，我强烈建议你立即开始学习至少一个这样的工具。

你可以阅读 Brendan Gregg 和 Jim Mauro 写的DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X和FreeBSD ，或访问http://dtrace.org/，以了解有关 `dtrace(1)` 实用程序的更多信息。
虽然， `dtrace(1)` 比 `strace(1)` 功能强大得多，因为它具有自己的编程语言。但是，当你要做的只是监视可执行文件的系统调用时， `strace(1)` 的用途更加广泛。

配置 Go 开发环境

配置 Go 开发环境

本节将讨论使用 `runtime` package 的功能和属性查找有关当前 Go 环境的信息。本节中将程序命名为 `goEnv.go` ，它将分为两部分。

第一部分：

```
package main
import (
  "fmt"
  "runtime"
)
```

`runtime` package 包含获取 runtime 信息的函数和属性。 `goEnv.go` 的第二个代码部分包含 `main()` 函数的实现：

```
func main() {
  fmt.Print("You are using ", runtime.Compiler, " ")
  fmt.Println("on a", runtime.GOARCH, "machine")
  fmt.Println("Using Go version", runtime.Version())
  fmt.Println("Number of CPUs:", runtime.NumCPU())
  fmt.Println("Number of Goroutines:", runtime.NumGoroutine())
}
```

在装有 Go 1.11.4 的 macOS Mojave 机器上执行 `goEnv.go` 将输出：

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.11.4
Number of CPUs: 8
Number of Goroutines: 1
```

同样的代码在装有 Go 1.3.3 的 Debian Linux 机器上输出如下：

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.3.3
Number of CPUs: 1
Number of Goroutines: 4
```

在名为 `requiredVersion.go` 的程序中说明了获取到有关 Go 环境的信息有什么用处，该程序会告诉你是否使用的是 Go1.8 或更高版本：

```
package main
import (
  "fmt"
  "runtime"
  "strconv"
  "strings"
)
func main() {
  myVersion := runtime.Version()
  major := strings.Split(myVersion, ".")[0][2]
  minor := strings.Split(myVersion, ".")[1]
  m1, _ := strconv.Atoi(string(major))
  m2, _ := strconv.Atoi(minor)
  if m1 == 1 && m2 < 8 {
    fmt.Println("Need Go version 1.8 or higher!")
    return
  }
  fmt.Println("You are using Go version 1.8 or higher!")
}
```

`strings` Go 标准包用于拆分从 `runtime.Version()` 获得的 Go 版本字符串，以获取其前两个部分，而 `strconv.Atoi()` 函数用于将字符串转换为整数。

在 macOS Mojave 机器上执行 `requiredVersion.go` 将输出：

```
$ go run requiredVersion.go
You are using Go version 1.8 or higher!
```

但如果你在 Debian Linux 机器上运行 `requiredVersion.go` ，它将输出：

```
$ go run requiredVersion.go
Need Go version 1.8 or higher!
```

因此，通过调用程序 `requiredVersion.go` ，你能够确定 UNIX 计算机是否具有所需的 Go 版本。

go env 命令

go env 命令

如果需要获取 Go 和 Go 编译器支持的所有环境变量及其当前值的列表，你可以执行 `go env` 。

在使用 Go1.11.4 的 macOS Mojave 上， `go env` 的功能非常丰富，如下所示：

```
$ go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/Users/mtsouk/Library/Caches/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/mtsouk/go"
GOPROXY=""
GORACE=""
GOROOT="/usr/local/Cellar/go/1.11.4/libexec"
GOTMPDIR=""
GOTOOLDIR="/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64"
GCCGO="gccgo"
CC="clang"
CXX="clang++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-
arguments -fmessage-length=0 -fdebug-prefix-
map=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-
build790367620=/tmp/go-build -gno-record-gcc-switches -fno-common"
```

请注意，如果你使用其他 Go 版本，用户名不是 `mtsouk` ，在其他硬件上使用其他 UNIX 变体或在使用 Go 模块(GOMOD)时，其中某些环境变量可能会不是默认值。

Go 汇编器

Go 汇编器

本节将简要讨论汇编语言和 Go 汇编器，这是一个 Go 工具，可让你查看 Go 编译器使用的汇编语言。
例如，通过执行以下，你可以看到在本章上一节中看到的 `goEnv.go` 程序的汇编语言：

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S goEnv.go
```

`GOOS` 变量的值定义目标操作系统的名称，而 `GOARCH` 变量的值定义编译体系结构。以上的命令在 macOS Mojave 机器上执行，因此将 `darwin` 值赋予 `GOOS` 变量。
即使对于像 `goEnv.go` 这样的简单程序，以上命令的输出也非常复杂：

```
"".main STEXT size=859 args=0x0 locals=0x118
  0x0000 00000 (goEnv.go:8)   TEXT    "".main(SB), $280-0
  0x00be 00190 (goEnv.go:9)   PCDATA  $0, $1
  0x0308 00776 (goEnv.go:13)  PCDATA  $0, $5
  0x0308 00776 (goEnv.go:13)  CALL    runtime.convT2E64(SB)
"".init STEXT size=96 args=0x0 locals=0x8
  0x0000 00000 (<autogenerated>:1) TEXT    "".init(SB), $8-0
  0x0000 00000 (<autogenerated>:1) MOVQ    (TLS), CX
  0x001d 00029 (<autogenerated>:1) FUNCDATA        $0,
gclocals d4dc2f11db048877dbc0f60a22b4adb3(SB)
  0x001d 00029 (<autogenerated>:1) FUNCDATA        $1,
  gclocals 33cdeccccebe80329f1fdbee7f5874cb(SB)
```

包含 `FUNCDATA` 和 `PCDATA` 指令的行由 Go 垃圾收集器读取和使用，并由 Go 编译器自动生成。
这条命令和以上命令是等效的：

```
$ GOOS=darwin GOARCH=amd64 go build -gcflags -S goEnv.go
```

`GOOS` 变量可能的取值包括 `android, darwin, dragonfly, freebsd, linux, nacl, netbsd, openbsd, plan9, solaris`。`GOARCH` 可能的取值包括 `386, amd64, amd64p32, arm, armbe, arm64, arm64be, ppc64, ppc64le, mips, mipsle, m`。

> Tip: 如果你真的对Go汇编器感兴趣，并且想要更多信息，请访问https://golang.org/doc/asm。

节点树

节点树

Go 节点是包含大量属性的结构。你将在第 4 章“复合类型的使用”中学习有关定义和使用 Go struct的更多信息。Go 编译器的模块根据 Go 编程语言的语法对 Go 程序中的所有内容进行解析和分析。此分析的最终结果是一棵树，该树对应于所提供的 Go 代码，并以适合编译器方式表示程序结构。

> Tip: 需要注意的是， `go tool 6g -W test.go` 在更新的 Go 版本中无法正常运行。取而代之你应该使用 `go tool compile -W test.go` 。

本节将首先使用以下 Go 代码(保存为 `nodeTree.go`)作为示例，以查看 go 工具可以为我们提供的底层信息：

```
package main
import (
  "fmt"
)
func main() {
  fmt.Println("Hello there!")
}
```

nodeTree.go 的 Go 代码非常容易理解，因此你不会对它感到惊讶其输出，接下来是：

```
$ go run nodeTree.go
Hello there!
```

通过执行下一个命令来查看一些 Go 的内部工作方式：

```
$ go tool compile -W nodeTree.go
before walk main
.   CALLFUNC l(8) tc(1) STRUCT-(int, error)
.   .   NAME-fmt.Println a(true) l(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
.   .   DDDARG l(8) esc(no) PTR64-*[1]interface {}
.   CALLFUNC-list
.   .   CONVIFACE l(8) esc(h) tc(1) implicit(true) INTER-interface {}
.   .   .   NAME-main.statictmp_0 a(true) l(8) x(0) class(PEXTERN) tc(1)
used string
.   VARKILL l(8) tc(1)
.   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) used
ARRAY-[1]interface {}
```

```
after walk main
.   CALLFUNC-init
.   .   AS l(8) tc(1)
.   .   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
.. AS l(8) tc(1)
.. . NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned
used PTR64-*[1]interface {}
.. . ADDR l(8) tc(1) PTR64-*[1]interface {}
.. . . NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) addrtak
en assigned used ARRAY-[1]interface {}
.   .   BLOCK l(8)
.   .   BLOCK-list
.   .   .   AS l(8) tc(1) hascall
.   .   .   .   INDEX l(8) tc(1) assigned bounded hascall INTER-interface
{}
.   .   .   .   .   IND l(8) tc(1) implicit(true) assigned hascall ARRAY-
[1]interface {}
.   .   .   .   .   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO)
esc(N) tc(1) assigned used PTR64-*[1]interface {}
.   .   .   .   .   LITERAL-0 l(8) tc(1) int
. .
. .
. . uint8
.. . . . ADDR l(8) tc(1) PTR64-*string
.. . . . . NAME-main.statictmp_0 a(true) l(8) x(0) class(PEXTERN) tc(1) addrtak
en used string
.   .
.   .   .   ADDR a(true) l(8) tc(1) PTR64-*uint8
.   .   .   .   NAME-type.string a(true) x(0) class(PEXTERN) tc(1)
EFACE l(8) tc(1) INTER-interface {}
.   .   BLOCK l(8)
.   .   BLOCK-list
.   .   .   AS l(8) tc(1) hascall
.   .   .   .   NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
.. . . SLICEARR l(8) tc(1) hascall SLICE-[]interface {}
.. . . . NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assig
ned used PTR64-*[1]interface {}
.   CALLFUNC l(8) tc(1) hascall STRUCT-(int, error)
.   .   NAME-fmt.Println a(true) l(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
.   .   DDDARG l(8) esc(no) PTR64-*[1]interface {}
.   CALLFUNC-list
.   .   AS l(8) tc(1)
.   .   .   INDREGSP-SP a(true) l(8) x(0) tc(1) addrtaken main.__ SLICE-
[]interface {}
.   .   .   NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
.   VARKILL l(8) tc(1)
```

```
.   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1)
addrtaken assigned used ARRAY-[1]interface {}
before walk init
.   IF l(1) tc(1)
.   .   GT l(1) tc(1) bool
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   RETURN l(1) tc(1)
.   IF l(1) tc(1)
.   .   EQ l(1) tc(1) bool
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   CALLFUNC l(1) tc(1)
.   .   .   NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-1 l(1) tc(1) uint8
.   CALLFUNC l(1) tc(1)
.   .   NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-2 l(1) tc(1) uint8
.   RETURN l(1) tc(1)
after walk init
.   IF l(1) tc(1)
.   .   GT l(1) tc(1) bool
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   RETURN l(1) tc(1)
.   IF l(1) tc(1)
.   .   EQ l(1) tc(1) bool
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   CALLFUNC l(1) tc(1) hascall
.   .   .   NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
```

```
.   .   LITERAL-1 l(1) tc(1) uint8
.   CALLFUNC l(1) tc(1) hascall
.   .   NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-2 l(1) tc(1) uint8
.   RETURN l(1) tc(1)
```

Go 编译器及其工具在后台做很多事情，即使对于像 `nodeTree.go` 这样的简单程序也是如此。

> Tip: `-W` 参数告诉 go 编译命令在类型检查后打印debug parse tree。

查看下面两个命令的输出：

```
$ go tool compile -W nodeTree.go | grep before
before walk main
before walk init
$ go tool compile -W nodeTree.go | grep after
after walk main
after walk init
```

由以上输出可以知道， `before` 关键字是关于函数执行开始的。如果你的程序具有更多功能，你将获得更多输出，比如：

```
$ go tool compile -W defer.go | grep before
before d1
before d2
before d3
before main
before d2.func1
before d3.func1
before init
before type..hash.[2]interface {}
before type..eq.[2]interface {}
```

前面的示例使用 `defer.go` 的 Go 代码，该代码比 `nodeTree.go` 复杂得多。但是，很显然， `init()` 函数是 Go 自动生成的，因为它也在 `go tool compile -W` 的两个输出中(`nodeTree.go` 和 `defer.go`)。现在，我将向你展示一个名为 `nodeTreeMore.go` 输出，它是一个多版本的 `nodeTree.go` ：

```
package main
import (
  "fmt"
```

```
)
func functionOne(x int) {
  fmt.Println(x)
}
func main() {
  varOne := 1
  varTwo := 2
  fmt.Println("Hello there!")
  functionOne(varOne)
  functionOne(varTwo)
}
```

`nodeTreeMore.go` 程序有两个变量，分别名为 `varOne` 和 `varTwo` ，以及一个名为 `functionOne` 的附加函数。在 `go tool compile -W` 输出中搜索 `varOne, varTwo` 和 `functionOne` 将显示以下信息：

```
$ go tool compile -W nodeTreeMore.go  | grep functionOne | uniq
before walk functionOne
after walk functionOne
.   .   NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used
FUNC-func(int)
$ go tool compile -W nodeTreeMore.go  | grep varTwo | uniq
.   .   NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used
int
.   .   .   NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1)
used int
$ go tool compile -W nodeTreeMore.go  | grep varOne | uniq
.   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used
int
.   .   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1)
used int
```

因此， `varOne` 表示为 `NAME-main.varOne` ，而 `varTwo` 表示为 `NAME-main.varTwo` 。 `functionOne()` 函数被称为 `NAME-main.functionOne` ， `main()` 函数被表示为 `NAME-main` 。现在，让我们看看 `nodeTreeMore.go` 的 debug parse tree：

```
before walk functionOne
.   AS l(8) tc(1)
.   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1)
assigned used int
.   .   NAME-main.x a(true) g(1) l(7) x(0) class(PPARAM) tc(1) used int
```

这段信息与 `functionOne()` 的定义有关。 `l(8)` 字符串告诉我们，可以在第八行(即在读取第七行之后)找到该节点的定义。 `NAME-main..autotmp_2` 整数变量由编译器自动生成。

debug parse tree 输出的下一部分将在此处进行说明：

```
.   CALLFUNC l(15) tc(1)
.   .   NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used
FUNC-func(int)
.   CALLFUNC-list
.   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used
int
```

第一行说在程序的第15行(由 `l(15)` 指定)，你将调用 `NAME-main.functionOne` ，该名称在程序的第7行定义，由 `l(7)` 指定，即一个需要一个整数参数的函数，该函数由 `FUNC func(int)` 指定。在 `CALLFUNC-list` 之后指定的参数函数列表包括 `NAME-main.varOne` 变量，该变量在程序的第12行定义，如 `l(12)` 所示。

进一步了解 Go 构建

进一步了解 Go 构建

如果要了解执行 `go build` 命令之后的更多信息，则应在其中添加 `-x` 标志：

```
$ go build -x defer.go
WORK=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-build254573394
mkdir -p $WORK/b001/
cat >$WORK/b001/importcfg.link << 'EOF' # internal
packagefile command-line-arguments=/Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d
packagefile fmt=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/fmt.a
packagefile
runtime=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime.a
packagefile
errors=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/errors.a
packagefile io=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/io.a
packagefile
math=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math.a
packagefile os=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/os.a
packagefile
reflect=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/reflect.a
packagefile
strconv=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/strconv.a
packagefile
sync=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync.a
packagefile
unicode/utf8=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode/u
tf8.a
packagefile
internal/bytealg=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/inter
nal/bytealg.a
packagefile
internal/cpu=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/
cpu.a
packagefile
runtime/internal/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd6
4/runtime/internal/atomic.a
packagefile
runtime/internal/sys=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/r
untime/internal/sys.a
packagefile
sync/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync/atomi
c.a
packagefile
internal/poll=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal
```

```
/poll.a
packagefile
internal/syscall/unix=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/
internal/syscall/unix.a
packagefile
internal/testlog=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/inter
nal/testlog.a
packagefile
syscall=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/syscall.a
packagefile
time=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/time.a
packagefile
unicode=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode.a
packagefile
math/bits=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math/bits.a
packagefile
internal/race=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal
/race.a
EOF
mkdir -p $WORK/b001/exe/
cd .
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/link -o
$WORK/b001/exe/a.out -importcfg $WORK/b001/importcfg.link -buildmode=exe -
buildid=nkFdi6n3HGYZXDdCOju1/VfKOjehfe3PSzik3cZom/OthUDj9rThOtZPf-2627/nkFd
i6n3HGYZXDdCOju1 -extld=clang /Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/buildid -w
$WORK/b001/exe/a.out # internal
mv $WORK/b001/exe/a.out defer
rm -r $WORK/b001/
```

虽然了解编译的详细过程对你说是好事，但是大多数情况下你都不需要了解这些详细的编译过程信息。

创建 WebAssembly 代码

创建 WebAssembly 代码

Go可以让你借助go工具创建WebAssembly代码。在说明过程之前，我先分享有关WebAssembly的更多信息。

对 Webassembly 的简单介绍

对 Webassembly 的简单介绍

WebAssembly(Wasm)是针对虚拟机的机器模型和可执行格式。它旨在提高速度减小文件大小。这意味着你可以在任何平台上使用WebAssembly二进制文件，而无需进行任何更改。

WebAssembly有两种格式：纯文本格式和二进制格式。纯文本格式的WebAssembly文件扩展名为.wat，而二进制文件的扩展名为.wasm。请注意，一旦有了WebAssembly二进制文件，就必须使用JavaScript API加载和使用它。

除了Go之外，还可以从其他支持静态类型的编程语言(包括Rust，C和C ++)中生成WebAssembly。

为什么 WebAssembly 很重要

为什么 WebAssembly 很重要

这里有几个主要的原因：

- WebAssembly 代码的运行速度非常接近 native，这意味着 WebAssembly 速度很快。
- 你可以使用多种编程语言来创建 WebAssembly 代码，这些语言可能包括你已经知道的编程语言。
- 大多数现代 Web 浏览器本身都支持 WebAssembly，而无需插件或安装其他任何软件。
- WebAssembly 的运行效率比Javascript快得多。

Go 与 WebAssembly

Go 与 WebAssembly

对于 Go，WebAssembly 只是另一种体系结构。因此，你可以使用 Go 的交叉编译功能来创建 WebAssembly 代码。

你将在第 11 章，“代码测试，优化和分析”中学习有关 Go 的交叉编译功能的更多信息。现在，请注意将 Go 代码编译到 WebAssembly 时正在使用的 `GOOS` 和 `GOARCH` 环境变量的值。

示例

示例

在本节中，我们将了解如何将 Go 程序编译为 WebAssembly 代码。 `toWasm.go` 的 Go 代码如下：

```
package main
import (
  "fmt"
)
func main() {
  fmt.Println("Creating WebAssembly code from Go!")
}
```

这里要注意的是此代码并没有用到 WebAssembly，并且 toWasm.go 可以单独编译和执行，这意味着它没有与 WebAssembly 相关的外部依赖关系。

创建 WebAssembly 代码所需执行的最后一步是执行以下命令：

```
$ GOOS=js GOARCH=wasm go build -o main.wasm toWasm.go
$ ls -l
total 4760
-rwxr-xr-x  1 mtsouk  staff  2430633 Jan 19 21:00 main.wasm
-rw-r--r--@ 1 mtsouk  staff      100 Jan 19 20:53 toWasm.go
$ file main.wasm
main.wasm: , created: Thu Oct 25 20:41:08 2007, modified: Fri May 28 13:51:43 2
032
```

因此，在第一个命令中通过 `GOOS` 和 `GOARCH` 的值告诉 Go 创建 WebAssembly 代码。如果未输入正确的 `GOOS` 和 `GOARCH` 值，则编译将不会生成 WebAssembly 代码。

使用创建好的 WebAssembly 代码

使用创建好的 WebAssembly 代码

到目前为止，我们只生成了一个 WebAssembly 二进制文件。所以，你仍然需要采取一些步骤，才能使用该 WebAssembly 二进制文件并在 Web 浏览器的窗口中查看其结果。

如果你使用Google Chrome 浏览器，则有一个标志可让你启用 Liftoff，Liftoff 是 WebAssembly 的编译器，从理论上讲，它可以提高 WebAssembly 代码的运行效率。尝试以下没什么副作用，你可以访问 chrome:// flags/ #enable-webassembly-baseline 来开启它。

第一步是将 `main.wasm` 复制到 Web 服务器的目录中。接下来，你将需要执行以下命令：

```
$ cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" .
```

这会将 Go 安装的 `wasm_exec.js` 复制到当前目录中。你应该将该文件放在与 `main.wasm` 相同的 Web 服务器目录中。

这里用到的 `index.html` 的代码：

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>Go and WebAssembly</title>
  </head>
  <body>
    <script src="wasm_exec.js"></script>
    <script>
      if (!WebAssembly.instantiateStreaming) {
        // polyfill
        WebAssembly.instantiateStreaming = async (resp, importObject) => {
          const source = await (await resp).arrayBuffer()
          return await WebAssembly.instantiate(source, importObject)
        }
      }
      const go = new Go()
      let mod, inst
      WebAssembly.instantiateStreaming(fetch('main.wasm'), go.importObject)
        .then(result => {
          mod = result.module
          inst = result.instance
          document.getElementById('runButton').disabled = false
        })
        .catch(err => {
          console.error(err)
        })
```

```
      async function run() {
        console.clear()
        await go.run(inst)
        inst = await WebAssembly.instantiate(mod, go.importObject)
      }
    </script>
    <button onClick="run();" id="runButton" disabled>Run</button>
  </body>
</html>
```

请注意，由 HTML 代码创建的 `Run` 按钮只有在 WebAssembly 代码加载完成的情况下点击有效。
下图显示了 WebAssembly 代码的输出，如 Google Chrome Web 浏览器的 JavaScript 控制台中所示。其他 Web 浏览器将显示类似的输出。

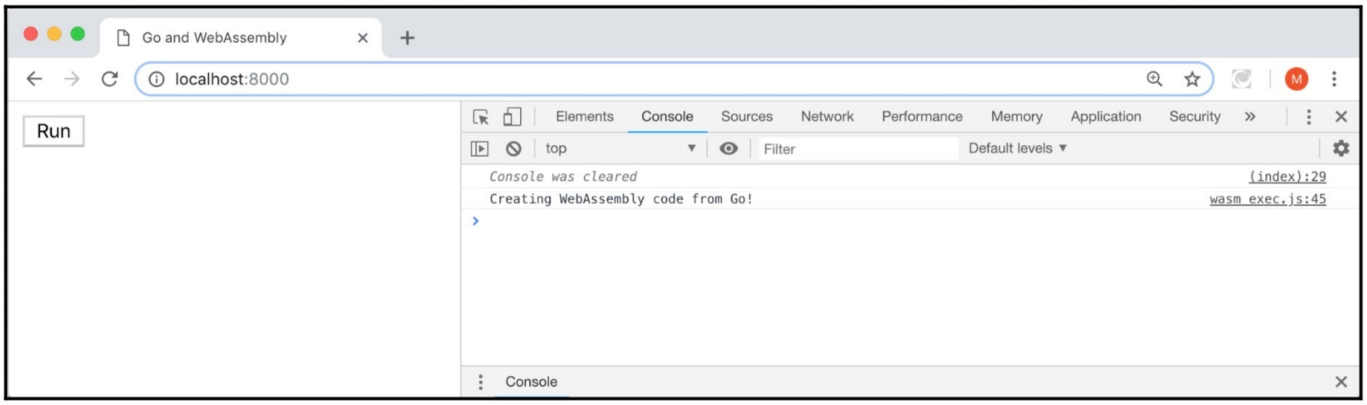


Figure 2.2: Serving WebAssembly code generated by Go

> Tip: 在第 12 章“Go 中的网络编程基础”中，你将学习如何在 Go 中开发自己的 Web 服务器。

但其实有一种测试 WebAssembly 应用程序的简便得多的方法，那就是使用Node.js。不需要 Web 服务器，因为 Node.js 是基于 Chrome V8 的 JavaScript 引擎构建的JavaScript运行时。
如果你已经在本地计算机上安装了Node.js，则可以执行以下命令：

```
$ export PATH="$PATH:$(go env GOROOT)/misc/wasm"
$ GOOS=js GOARCH=wasm go run .
Creating WebAssembly code from Go!
```

第二个命令的输出将验证WebAssembly代码是否正确并生成所需的消息。请注意，第一个命令并不是必须的，因为它只是更改PATH环境变量的当前值，以便包括当前Go安装程序存储与WebAssembly相关的文件的目录。

Go 编码风格建议

Go 编码风格建议

以下是编码风格建议，可帮助你编写更好的 Go 代码：

- 如果你在 Go 函数中有错误，请记录或返回它；除非你有充分的理由这样做，否则请不要两者都做。
- Go 接口定义的是行为，不是数据和数据结构。
- 尽可能使用 `io.Reader` 和 `io.Writer` 接口，因为它们使你的代码更具可扩展性。
- 确保仅在需要时才将指向变量的指针传递给函数。其余时间，只需传递变量的值即可。
- 表示程序错误的变量不是 `string` 变量，是 `error` 变量！
- 除非有充分的理由，否则请不要在生产环境上测试 Go 代码。
- 如果你真的不了解 Go 功能，请在首次使用它之前对其进行测试，尤其是当你正在开发将被大量用户使用的应用程序或实用程序时。
- 如果你害怕犯错误，那么你很可能最终会无所事事。尽可能多地尝试！

练习和相关链接

练习和相关链接

- 访问 https://golang.org/pkg/unsafe/ 上的文档页面，以了解有关 `unsafe` package标准Go软件包的更多信息。
- 访问 DTrace 网站 http://dtrace.org/。
- 在 Linux 机器上使用 `strace(1)` 检查某些标准 UNIX 实用程序(例如 `cp(1)` 和 `ls(1)`)的操作，你发现了了什么？
- 如果使用的是 macOS 计算机，请使用 `dtruss(1)` 查看 `sync(8)` 实用程序的工作方式。
- 编写你自己的示例，在其中使用 Go 程序调用 C 代码。编写 Go 函数并在 C 程序中使用它。
- 你可以通过以下方式找到有 `runtime` package 的更多信息：https://golang.org/pkg/runtime/。
- 阅读研究论文可能很困难，但很有收获。请下载“On-the-Fly Garbage Collection: An Exercise in Cooperation”论文并阅读。可以在许多地方找到该论文，包括 https://dl.acm.org/citation.cfm?id=359655。
- 访问 https://github.com/gasche/gc-latency-experiment 以查找各种编程语言的垃圾收集器的基准测试代码。
- 可以通过 https://nodejs.org/en/ 访问 Node.js 网站。
- 你可以在 https://webassembly.org/ 上了解有关 WebAssembly 的更多信息。
- 如果你想了解有关垃圾收集的更多信息，请务必访问http://gchandbook.org/。
- 访问 cgo 的文档页面，网址为https://golang.org/cmd/cgo/。

本章小结

本章小结

本章讨论了许多有趣的 Go 相关主题，包括有关 Go 垃圾收集器的理论和实践信息。如何从 Go 程序中调用 C 代码；方便但不那么好掌握的 `defer` 关键字； `panic()` 和 `recovery()` 函数； `strace(1)` ， `dtrace(1)和 dtruss(1)` UNIX 工具；使用 `unsafe package` ；如何从 Go 生成 WebAssembly 代码；和 Go 生成的汇编代码。使用 `runtime` package 查看了有关 Go 环境的信息，并展示了如何输出和解释 Go 程序的节点树。最后，给了你一些不错的 Go 编码建议。

在本章中，你应该记住的是， `unsafe` package 和从 Go 调用 C 代码的功能通常在以下三种情况下使用：首先，当你想要最佳性能时，又想为此牺牲一些 Go 安全性;其次，当你想与另一种编程语言交流时；第三，当你想要实现 Go 中无法实现的功能时。

在下一章中，我们将开始学习 Go 附带的基本数据类型，包括arrays, slices, maps。尽管它们很简单，但是这些数据类型几乎是每个 Go 应用程序的构建基础，因为它们是更复杂的数据结构的基础，它使你可以在 Go 项目中存储数据和移动信息。

此外，你还将学习pointers, Go loop，这和在其他编程语言中类似。以及 Go 与日期和时间配合使用的特有方式。

3 Go基本数据类型

Go基本的数据类型

前面的章节讨论了很多有趣迷人的话题，这其中包括Go的垃圾回收机制， `panic()、recover()` 函数的使用，不安全包，如何使用Go调用C代码以及如何使用C程序调用Go代码，还包含了Go编译器在编译Go时形成的程序树的介绍。

本章的主要话题是Go的基本数据类型，这包括数值类型，数组，切片，和映射（maps). 尽管它们很简单，但是这些数据类型可以帮助你进行数值计算。你还可以以一种非常方便快捷的方式存储、检索和更改程序的数据。这一章还包括指针、常量、循环以及如何在Go中处理日期和时间。

本章你将会学习的主题：

- 数值类型
- Go数组
- Go切片以及为什么切片比数组要好
- 如何在已经存在的切片中加入数组
- Go映射
- Go指针
- Go循环
- Go常量
- 使用时间工作
- 测量命令和函数的执行时间
- 操作日期

03.1 Go循环

Go数值类型

Go原生支持整数、浮点数和复数. 接下来的小节将深入介绍每一个Go所支持的数值类型.

03.1.1 for循环

整数

Go支持四种不同大小的有符号整数和无符号整数，分别是 `int8、int16、int32` 和 `int64` ，以及 `uint8、uint16、uint32、uint64` 。每种类型末尾的数字显示用于表示每种类型的位数。
另外，对于当前运算平台， `int` 和 `uint` 分别代表最有效的有符号整数和无符号整数。因此，如果有疑问，可以使用 `int` 和 `uint` ，但要记住这些类型的大小取决于计算机结构。
有符号整数和无符号整数之间的区别如下：如果一个整数有8位并且没有符号，那么它的值可以是二进制的 `00000000(0)` 到二进制的 `11111111(255)` 。如果它有一个符号，那么它的值可以是-128（ `原文是-127，应该不对 补码10000000是-128`)到127。这意味着你有7个二进制数字来存储你的数字，因为第8位用于保存整数的符号。同样的规则也适用于其他大小的无符号整数。

03.1.2 while循环

浮点数

Go只支持两种类型的浮点数： `float32、float64` 。第一个提供了大约小数点后6位的精度，而第二个提供了15位精度。

03.1.3 range关键字

复数

和浮点数相似，Go提供了两种复数类型： `complex64` 和 `complex128` 。第一个使用两个32位浮点数：一个用于实部，另一个用于复数的虚部，而 `complex128` 使用两个64位浮点数。复数以 `a + bi` 的形式表示，其中 `a` 和 `b` 是实数，i 是方程 $x^2 =-1$ 的解。
所有这些数字数据类型都在 `numbers.go` 中分为三个部分进行说明。
第一部分的代码如下：

```
package main

import (
"fmt"
)

func main() {
    c1 := 12 + 1i
    c2 := complex(5, 7)
    fmt.Printf("Type of c1: %T\n", c1)
    fmt.Printf("Type of c2: %T\n", c2)
    var c3 complex64 = complex64(c1 + c2)
    fmt.Println("c3:", c3)
    fmt.Printf("Type of c3: %T\n", c3)
    cZero := c3 - c3
    fmt.Println("cZero:", cZero)
```

这部分使用了一些复数进行计算。有两种方法去创建复数：与 `c1` 和 `c2` 一样直接创建，或间接地通过计算现有的复数得到，例如 `c3` 和 `cZero` 。

> Tip: 如果你错误地尝试将一个复数创建为 `aComplex:= 12 + 2 * i` ，那么将会有两种可能的结果，因为这个语句告诉Go你想要执行一个加法和一个乘法。如果当前作用域中没有名为i的数值变量，该语句会出现语法错误，你的Go代码编译将失败。但是，如果已经定义了一个名为i的数值变量，那么计算将会成功，但是你将不会得到 需的复数(bug)。

第二部分代码如下：

```
    x := 12
    k := 5
    fmt.Println(x)
    fmt.Printf("Type of x: %T\n", x)
    div := x / k
```

```
    fmt.Println("div", div)
```

在这一部分中，我们使用带符号的整数。请注意，如果你想对两个整数做除法，Go认为你想得到整数答案并将计算返回整数除法的商。11除以2得到的是整数5而不是5.5。

> Tip: 当你将浮点数转换为整数时，浮点数的分数将被丢弃且被截断为零，这意味着一些数据可能会在处理过程中丢失。

最后一部分代码：

```
    var m, n float64
    m = 1.223
    fmt.Println("m, n:", m, n)
    y := 4 / 2.3
    fmt.Println("y:", y)
    divFloat := float64(x) / float64(k)
    fmt.Println("divFloat", divFloat)
    fmt.Printf("Type of divFloat: %T\n", divFloat)
}
```

在程序的最后一部分中，我们将使用浮点数。在做除法时, 你可以看到如何使用 `float64()` 来告诉Go去创建一个 `floating-point number` 。如果你只是使用 `divFloat: = float64 (x) / k` ，然后运行代码时你会得到以下错误消息:

```
$ go run numbers.go
# command-line-arguments
./numbers.go:35:25: invalid operation:
float64(x) / k (mismatched types float64 and int)
```

执行 `numbers.go` ，结果输出如下：

```
Type of c1: complex128
Type of c2: complex128
c3: (17+8i)
Type of c3: complex64
cZero: (0+0i)
12
Type of x: int
div 2
m, n: 1.223 0
y: 1.7391304347826086
divFloat 2.4
Type of divFloat: float64
```

03.1.4 for循环代码示例

for循环代码示例

在撰写本文时，有人建议对Go处理数值字面量(number literals)的方式进行更改。数值字面量与你在编程语言中定义和使用数字的方式有关。这个特别的建议与二进制整数字面量、八进制整数字面量、数字分隔符和十六进制浮点数有关。你可以在网站上找到更多关于Go 2和数值字面量的信息 https://golang.org/design/19308-number-literals。

> Tip: 开发人员维护了详细发布页面,你可以查看 https://dev.golang.org/release。

03.3 Go切片

Go数组

数组是最流行的数据结构之一，原因有两个。第一个原因是它简单易懂,而第二个原因是,他们非常多样化,可以存储许多不同种类的数据。我们可以通过如下代码声明一个存储四个整数的数组:

```
anArray: = [4]int{1, 2, 4, 4}
```

数组元素类型前的数字表示数组的大小。你可以通过函数： `len(anArray)` 得到一个数组的长度。
数组任意维度的第一个元素的索引为0；第二个元素的数组的索引为1，以此类推。这意味着一个命名为a的一维数组,有效的索引从 `0` 到 `len(a) - 1` 。
在其他编程语言中，尽管你可能熟悉访问数组的元素和使用一个for循环以及一个或多个数值变量，但是在Go中有更多访问数组所有元素的惯用方法。它们涉及 `range` 关键字的使用和允许你不使用 `len()` 函数。
`loops.go` 代码中举了这样的例子。

03.3.1 切片基本操作

多维数组

数组可以有多个维度。然而，在没有严肃理由的情况下，使用超过三个维度会使你的程序难于阅读，并可能产生bug。

> Tip: 数组可以存储所有类型的元素，这里我们只用整数讲解，因为他们更容易理解和类型。

下面的Go代码演示了如何创建一个二维数组(twoD)和另一个三维数组(threeD):

```
twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12},{13, 14, 15, 16}}
threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

访问、分配或打印这两个数组中的一个元素是很容易的。例如，twoD数组的第一个元素是 `twoD[0][0]` ，它的值是1。

因此，使用多层循环遍历threeD 的每一个元素可以通过下面的代码实现：

```
for i := 0; i < len(threeD); i++ {
    for j := 0; j < len(v); j++ {
        for k := 0; k < len(m); k++ {
        }
    }
}
```

我们看到，我们需要和数组维度一样的循环的数量来遍历所有元素。这个规则对切片 `slice` 也适用，下章将会讲到。这里使用 `x, y, z` 替代 `i, j, k` 会比较好。

`usingArrays.go` 展示了如何在Go中使用数组，下面分成三部分进行讲解。

第一部分的代码如下：

```
package main
import (
"fmt"
)
func main() {
    anArray := [4]int{1, 2, 4, -4}
    twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14,15,
16}}
    threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5,-1},{7, 0}}}
```

这里我们定义了三个数组变量分别叫 anArray, twoD 和 threeD。

第二部分的代码如下：

```
    fmt.Println("The length of", anArray, "is", len(anArray))
    fmt.Println("The first element of", twoD, "is", twoD[0][0])
    fmt.Println("The length of", threeD, "is", len(threeD))
    for i := 0; i < len(threeD); i++ {
        v := threeD[i]
        for j := 0; j < len(v); j++ {
            m := v[j]
            for k := 0; k < len(m); k++ {
                fmt.Print(m[k], " ")
            }
        }
        fmt.Println()
    }
```

我们从第一个for循环中得到一个二维数组（ `threeD[i]` ），同理，第二个for循环中得到一个一维数组。最后一个for循环遍历得到的一维数组得到里面的每一个元素。

最后一部分代码：

The last code part comes with the next Go code:

```
    for _, v := range threeD {
        for _, m := range v {
            for _, s := range m {
                fmt.Print(s, " ")
            }
        }
        fmt.Println()
    }
}
```

`range` 关键字的作用与上一个代码段的for循环中使用的迭代变量完全相同，但它的作用更优雅、更清晰。然而，如果想要提前知道将要执行的迭代次数，那么就不能使用 `range` 关键字。

> `range` 关键字也适用于Go映射，这使得它非常方便，也是我喜欢的迭代方式。正如将在第9章中看到的Go - Goroutines、channel和pipeline中的并发性，range关键字也适用于channel。

执行 `usingArrays.go` 结果输出如下：

```
$ go run usingArrays.go
The length of [1 2 4 -4] is 4
The first element of [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16]] is 1
```

```
The length of [[[1 0] [-2 4]] [[5 -1] [7 0]]] is 2
1 0 -2 4
5 -1 7 0
1 0 -2 4
5 -1 7 0
```

数组最大的问题之一是越界错误，这意味着试图访问数组中不存在的元素。这就像试图访问一个只有5个元素的数组的第6个元素。Go编译器将此认作编译检测的编译问题，因为这有助于开发工作流。因此，Go编译器能检测数组越界错误并给出如下错误提示：

```
./a.go:10: invalid array index -1 (index must be non-negative)
./a.go:10: invalid array index 20 (out of bounds for 2-element array)
```

03.3.2 切片的扩容

Go数组的缺点

Go数组有很多缺点，这将使你重新考虑在Go项目中使用它们。首先，一旦你定义了一个数组，你就不能改变它的大小，这意味着Go数组不是动态的。简单地说，如果需要将一个元素添加到一个已经没有空间的现有数组中，那么我们需要创建一个更大的数组，并将旧数组中的所有元素复制到新数组中。另外，当将数组作为参数传递给函数时，实际上传递的是数组的一个副本，这意味着对函数内部数组所做的任何更改都将在函数退出后丢失。最后，将一个大数组传递给一个函数可能会非常慢，这主要是因为Go必须创建一个数组的副本。所有这些问题的解决方案是使用Go切片，这将在下一节中介绍。

> Tip: 由于它们的缺点，在Go中很少使用数组!

03.3.3 字节切片

03.3.4 copy()函数

03.3.5 多维切片

03.3.6 使用切片的代码示例

03.3.7 使用sort.Slice()排序

03.4 Go 映射(map)

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。

> Tip: 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。

> Bryan：就和C/C++中的array和vector一样，vector会自动翻倍扩充。

03.4.1 Map值为nil的坑

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初始值，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, 12345)
```

`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。

> Tip: 假设有一个数组a1，你可以执行 `s1 := a1[:]` 来创建一个引用a1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，只要较小的重新切片存在，来自原始切片的底层数组就会被保存在内存中，因为较小的重新切片引用了原始切片, 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，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。

03.4.2 何时该使用Map?

切片的自动扩容

切片有两个主要的属性：容量(cap)和长度(len)，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，你再往该切片追加一个元素，当然长度增加1，但是切片的容量会变为原来的两倍。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(aSlice))
    aSlice = append(aSlice, -100)
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n",cap(aSlice),len(aSlice))
```

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(aSlice))
}
```

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。

03.5 Go 常量

Go 映射(maps)

一个 Go map（映射，下文不做翻译）就是在其它编程语言中众所周知的哈希表。map数据结构的主要优势就是其可以使用任意数据类型作为键值，但是对于Go map来说并不是所有的数据类型都能作为键值，只有可比较的类型才可以，意思是Go编译器能够区分不同的键值。或者简单来说，Go map的键值必须支持 `==` 操作符。显而易见，使用 `bool` 类型作为map的键值是非常不灵活的。另外，由于不同机器和操作系统的浮点数精度定义不同，使用浮点数作为键值可能会出现异常。

> Tip: Go map的底层指向了一个哈希表！Go已经隐藏了哈希表的实现及其复杂性，你将在第五章学习如何使用Go实现一个哈希表。

下面使用 `make()` 函数以 `string` 为键类型，以 `int` 作为值类型创建一个空的map:

```
iMap := make(map[string]int)
```

同样也可以使用map字面量创建并初始化一个map:

```
anotherMap := map[string]int {
            "k1": 12,
            "k2": 13
            }
```

可以使用 `anotherMap["k1"]` 可以获得对应的值，使用 `delete()` 删除一个键值对：

```
delete(anotherMap, "k1")
```

遍历map中得元素可使用如下代码:

```
for key, value := range iMap {
    fmt.Priintln(key, value)
}
```

`usingMaps.go` 中的代码将会更加详细地展示map的用法。代码将会分为3部分，第一部分是:

```
package main

import (
```

```
    "fmt"
)

func main() {
    iMap := make(map[string]int)
    iMap["k1"] = 12
    iMap["k2"] = 13
    fmt.Println("iMap:", iMap)

    anotherMap := map[string]int {
        "k1": 12,
        "k2": 13,
    }
```

第二部分是：

```
    fmt.Println("anotherMap:",anotherMap)
    delete(anotherMap,"k1")
    delete(anotherMap, "k1")
    delete(anotherMap, "k1")
    fmt.Println("anotherMap:",anotherMap)

    _, ok := iMap["doseItExist"]
    if ok {
        fmt.Println("Exist!")
    } else {
        fmt.Println("dose NOT exist")
    }
```

这里你将学习到如何判断map中拥有某个键值对，这是很重要的知识点，如果不了解的话，你将无法判断一个map是否拥有你想要的信息。

> 需要注意的是，当你尝试使用一个并不存在的键去获取值的时候，返回值是0，但是你并不知道到底是某个键对应的值是0，还是由于所访问的键不存在而返回的0，这是为什么我们使用 `_, ok` .

另外，代码中多次调用delete()去删除同一个元素并不会导致异常或者警告。
最后一部分代码展示了使用 `range` 关键字遍历map是非常简洁和方便的:

```
    for key, value := range iMap {
        fmt.Println(key, value)
    }
}
```

执行 `usingMaps.go` 将会得到下面的输出:

```
$ go run usingMaps.go

iMap: map[k1:12 k2:13]
anotherMap: map[k1:12 k2:13]
anotherMap: map[k2:13]
dose NOT exist
k1 12
k2 13
```

> Tip: 你不能也不应该期望键值对是按顺序打印的，因为遍历map时其顺序是随机的。

03.5.1 常量生成器：iota

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
```

这意味着试图向nil map中插入值是不行的，但是查找、删除、长度以及使用range循环是可以的。

03.6 Go 指针

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。

> Tip: 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 //注意这里是float64
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或者结构体中。

03.7 时间与日期的处理技巧

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。

> Tip: 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。

> Tip: 在Go中字符串是数值类型而不是指针这和C语言不一样。

03.7.1 解析时间

为什么使用指针？

在你的程序中使用指针有两个主要原因:

- 指针允许你共享数据，特别是在Go函数之间。
- 当你希望区分零值和未设置的值时，指针非常有用。

03.7.2 解析时间的代码示例

03.7.3 解析日期

03.7.4 解析日期的代码示例

03.7.5 格式化时间与日期

03.8 延伸阅读

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那么重要，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 19-01-12T15:59:39+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！

03.9 练习

有用的链接和练习

- 使用 `iota` 编写常量生成器表示4的指数
- 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`
- 阅读官方的 `time` 包：https://golang.org/pkg/time https://golang.org/pkg/time/.
- 你也可以访问GitHub的页面，那里正在讨论Go 2和数字字面量的变化，这将帮助你了解Go正在发生的变化以及它们是如何发生的：https://github.com/golang/proposal/blob/master/design/19308-number-literals.md.

03.10 本章小结

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。

JSON是一种非常流行的文本格式，因此下一章还将讨论如何在Go中创建、导入和导出JSON数据。

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。

9 并发-Goroutines,Channel和Pipeline

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。系统编程中，前面章节未涉及的两个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（ **`goroutine`** ）和通道（ **`channels`** ）。协程是Go程序中可独立执行的最小实体单元，而通道是协程间并发有效获取数据的方式，这允许协程间具有引用点并可以相互通信。Go中的所有内容都是使用协程执行的，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，单个协程调用 `main()` 函数，该函数执行实际的程序。

本章的内容和代码都比较简单，你应该可以很容易地理解它们。 `goroutines` 和 `channels` 中更高级的部分留到第10章中。

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
- `Erlang` 和 `Rust` 中的并发模型
- 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

09.1 关于进程，线程和Go协程

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, 以及包含其运行时获得的其他类型资源的程序执行环境，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。区分线程和进程的一个简单的方式是：假如进程是正在运行的二进制文件，线程就是其子集。

`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` 存在于Unix进程的线程中，它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，线程和协程的基本知识，下一小节我们聊聊Go调度器。

09.1.1 Go调度器

Go调度器

Unix内核调度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 scheduling` 的调度技术负责执行 `goroutine` ，通过多路复用使n个操作系统线程执行m个 `goroutine` 。Go调度器是Go中负责Go程序中 `goroutine` 的执行方式和执行顺序的组件。这使得Go调度器成为Go编程语言中非常重要的一部分，因为所有的Go程序都是以 `goroutine` 的形式执行的。

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。

`Chapter 10, Concurrency in Go – Advanced Topics` 将讨论更多关于Go调度器的更多细节。

09.1.2 并发与并行

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，而并发是构建你的组件的一种方式，以便它们可以在可能的情况下独立执行。

当你的操作系统和硬件允许时,只有当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，好的并行性来自于更好的并发表达和解决问题的方式。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。

即使你无法在 `Unix` 机器上并行运行你的函数，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！

09.2.2 创建多个Goroutine

09.2.1 创建一个Goroutine

创建一个Goroutine

在本小节中，你将学习两种创建 `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 - 这两种方法是等价的。

本小节所展示的程序文件为 `simple.go` ，它分为三个部分。

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，你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。
正如你将在下一节中看到的，你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 15 16 17 18 19 0123456789

$ go run simple.go
10 11 12 13 14 15 16 0117 2345678918 19

$ go run simple.go
10 11 12 012345678913 14 15 16 17 18 19
```

尽管对于你的程序你想要的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，三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。

09.3 优雅地结束goroutines

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。

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。

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，你可以使用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，以防止出现任何形式的竞争。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。
最后一部分代码如下：

```
    fmt.Printf("%#v\n", waitGroup)
    waitGroup.Wait()
    fmt.Println("\nExiting...")
}
```

`sync.Wait()` 调用将阻塞主程序，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, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x14,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema: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, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
1 0 4 5 17 7 8 9 10 11 12 13 2 sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]
uint8{0x0, 0x0, 0x0, 0x0, 0x17, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
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, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
```

```
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x1e,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema: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 21 14 3 17 18 27
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。

09.3.1 当Add()和Done()的数量不匹配时会发生什么？

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，本节将告诉你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{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x15, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 10 11 12 13 17 18 8 5 4 6 14 1 0 7 3 2 15 9 16 fatal error: all
goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc4200120bc)
/usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc4200120b0)
/usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
/Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:28 +0x2d7
exit status 2
```

错误消息是很清楚的: `fatal error: all goroutines are asleep - deadlock!` .，这是因为你通过调用 `sync.Add(1)` 函数 `n+1` 次来告诉程序等待 `n+1` 个 `goroutine` ，而 `n` 个 `goroutine` 只执行了 `n` 个 `sync.Done()` 语句。因此， `sync.Wait()` 调用将无限期地等待一个或多个对 `sync.Done()` 的调用，而不会有任何结果，这显然是死锁的情况。

如果使用的 `sync.Add()` 调用比 `sync.Done()` 调用少，那么可以通过在 `syncGo.go` 的for循环之后添加waitGroup.Done()语句来进行模拟。那么执行后输出类似如下的结果:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x12, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
```

```
19 6 1 2 9 7 8 15 13 0 14 16 17 3 11 4 5 12 18 10 panic: sync: negative
WaitGroup counter
goroutine 22 [running]:
sync.(*WaitGroup).Add(0xc4200120b0, 0xffffffffffffffff)
/usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:75 +0x134
sync.(*WaitGroup).Done(0xc4200120b0)
/usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:100 +0x34
main.main.func1(0xc4200120b0, 0x11)
/Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:25 +0xd8
created by main.main
/Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:21 +0x206
exit status 2
```

这次问题的根源也非常清楚： `panic: sync: negative WaitGroup counter` 。虽然这两种情况下的错误消息都非常具体，可以帮助你解决实际的问题，但是你应该非常小心地处理放入程序中的 `sync.Add()` 和 `sync.Done()` 调用的数量。另外，请注意，第二个错误情况(`panic: sync: negative WaitGroup counter`)可能并不总是出现。

09.4 Channel(通道)

通道(Channel)

通道(`channel`)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：当你使用通道作为函数参数时，你可以指定其方向; 也就是说，该通道是用于发送数据或是接收数据。在我看来，如果你事先知道一个通道的用途，你应该使用这个功能，因为它会使你的程序更健壮，也更安全。你将无法意外地将数据发送到应该只从其接收数据的通道，或从应该只向其发送数据的通道接收数据。因此，如果你声明一个通道函数参数将被用于只读，并尝试对其进行写操作，那么你将得到一条错误消息，它可能会帮助你从讨厌的bug中解救出来。我们将在本章后面讨论这个问题。

> Tip: 作者说第十章一定要学习呀，会有更有很好的理解。

09.4.1 通道的写入

通道的写入

这小节的代码将教你怎样往通道写入数据。把 `x` 值写到 `c` 通道，通过 `c <- x` 就可以实现，这非常简单。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` 表达式允许你写 `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。一旦你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。

09.4.2 从通道接收数据

从通道接收数据

这小节，你将了解到如何从通道读取数据。你可以执行 `<-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` 的程序帮你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。如果你想要保存数据到名为 `k` 的变量而不只是打印它的话，你可以使用 `k := <-c` 表达式。第二个 `time.Sleep(1 * time.Second)` 语句给你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，你能看到一个判断一个通道是打开还是关闭的技巧。当通道关闭时表明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。如果你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
2 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，因为当你从通道读取数据时，它就被解除阻塞了。

> Bryan: 我的实验结果是第一个结果出现的概率大些。

09.4.3 通道作为函数参数传递

从关闭的channel中读数据会发生什么

来看代码：

```
package main
import (
"fmt"
)
func main() {
    willClose := make(chan int, 10)
    willClose <- -1
    willClose <- 0
    willClose <- 2

    <-willClose
    <-willClose
    <-willClose

    close(willClose)
    read := <-willClose
    fmt.Println(read)
}
```

代码创建了一个int类型的通道 `willClose` ，并且向其中写入三个值，然后再依次读出来。后面关闭这个通道后，再尝试从其中读数据，执行这个程序输出结果：

```
$ go run readClose.go
0
```

这个结果表明，如果我们尝试从关闭的通道中读取数据，其会返回基本类型的初始值。

09.5 管道

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，因为你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。你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。你已经对 `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。

09.6 延展阅读

竞态(Race conditions)

`pipeline.go` 并不完美，它包含一个逻辑错误，在并发术语中称为竞态。这可以通过执行以下命令来揭示:

```
$ go run -race pipeline.go 1 10
2 2 ==================
WARNING: DATA RACE
Write at 0x00000122bae8 by goroutine 7:
main.second()
/Users/mtsouk/ch09/pipeline.go:34 +0x15c
Previous read at 0x00000122bae8 by goroutine 6:
main.first()
/Users/mtsouk/ch09/pipeline.go:21 +0xa3
Goroutine 7 (running) created at:
main.main()
/Users/mtsouk/ch09/pipeline.go:72 +0x2a1
Goroutine 6 (running) created at:
main.main()
/Users/mtsouk/ch09/pipeline.go:71 +0x275
==================
2
The sum of the random numbers is 2.
Found 1 data race(s)
exit status 66
```

这里的问题是，当 `first()` 函数读取CLOSEA变量时，执行 `second()` 函数的 `goroutine` 可能会更改CLOSEA变量的值。因为先发生什么和后发生什么是不确定的，所以被认为是竞态。为了修正这个竞态条件，我们需要使用一个信号通道和 `select` 关键字。

> Tip：第十章会有更多 `select` 的介绍。

`diff(1)` 命令会揭示新代码 `plNoRace.go` 和之前代码的区别：

```
$ diff pipeline.go plNoRace.go
14a15,16
> var signal chan struct{}
>
21c23,24
<    if CLOSEA {
---
>    select {
>    case <-signal:
```

```
23a27
>   case out <- random(min, max):
25d28
<   out <- random(min, max)
31d33
<   fmt.Print(x, " ")
34c36
<   CLOSEA = true
---
>   signal <- struct{}{}
35a38
>   fmt.Print(x, " ")
61d63
<
66a69,70
>   signal = make(chan struct{})
>
```

`plNoRace.go` 的正确性可以通过如下命令验证：

```
$ go run -race plNoRace.go 1 10
8 1 4 9 3
The sum of the random numbers is 25.
```

09.7 练习

比较Go和Rust的并发模型

Rust是一种非常流行的系统编程语言，它也支持并发编程。简要来说,一些Rust的特点和的并发模型如下:

- Rust线程是UNIX线程，这意味着他们不是轻量的，但可以做许多事情。
- Rust支持消息传递和共享状态并发，就像Go支持通道、互斥锁和共享变量一样。
- 基于其严格的类型和所有制、Rust提供了一个安全线程可变状态。这些规则由Rust编译器强制执行。
- 有些Rust的结构允许你共享状态。
- 如果一个线程开始行为不正常，系统将不会崩溃。这种情况可以被处理和控制。
- Rust语言在不断发展，这可能会阻止一些人用它，他们可能需要修改现有代码。

所以，Rust有一个灵活的并发模型，它甚至比Go的并发模型还要灵活。然而，你为这种灵活性付出的代价是你不得不接受Rust的独特特性。

09.8 本章小结

比较Go和Erlang并发模型

Erlang是一种非常流行的并发函数式编程语言，它在设计时考虑了高可用性。简单地说，Erlang和Erlang并发模型的主要特点如下:

- Erlang是一个成熟的编程语言——这也适用于它的并发模型。
- 如果你不喜欢Erlang代码的工作方式，你总是可以试着使用Elixir - 基于Erlang并使用Erlang虚拟机，但其代码更加简洁。
- Erlang只使用异步通信。
- Erlang使用错误处理来开发健壮的并发系统。
- Erlang进程可以崩溃，但是如果崩溃是妥善处理，该系统可以继续工作。
- 就像 `goroutine` ，Erlang进程之间是隔离的，意味着它们之间没有共享状态。Erlang进程之间通信的唯一途径是通过消息传递。- Erlang线程是轻量级的，就像Go goroutines一样。这意味着你可以创建尽可能多的过程。

总之，只要愿意使用Erlang并发方法，Erlang和Elixir都是可靠和高可用性系统开发的不错选择。