Go语言圣经（中文版）

The Go Programming Language

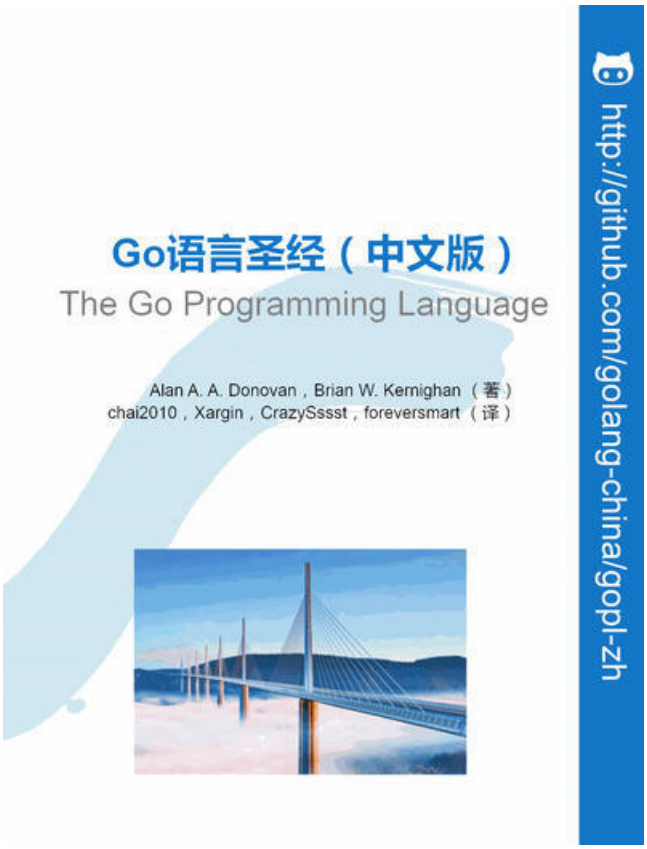
Alan A. A. Donovan，Brian W. Kernighan（著）
chai2010，Xargin，CrazySssst，foreversmart（译）

目錄

Go語言聖經（中文版）

Go語言聖經 《The Go Programming Language》 中文版本，僅供學習交流之用。

- 項目主頁：http://github.com/golang-china/gopl-zh
- 項目進度：http://github.com/golang-china/gopl-zh/blob/master/progress.md
- 參與人員：http://github.com/golang-china/gopl-zh/blob/master/CONTRIBUTORS.md
- 離線版本：http://github.com/golang-china/gopl-zh/archive/gh-pages.zip
- 在線預覽：http://golang-china.github.io/gopl-zh
- 原版官網：http://gopl.io



從源文件構建

先安裝NodeJS和GitBook命令行工具(`npm install gitbook-cli -g` 命令)。

1. 下載 https://github.com/golang-china/gopl-zh/archive/master.zip ，獲取源文件。
2. 切換到 `gopl-zh` 目録，運行 `gitbook install` ，安裝GitBook插件。
3. 運行 `gitbook build` ，生成 `_book` 目録。
4. 打開 `_book/index.html` 文件。

簡體/繁體轉換

切片到 `gopl-zh` 目録：

- `make zh2tw` 或 `go run zh2tw.go . "\.md$" zh2tw` ，轉繁體。
- `make tw2zh` 或 `go run zh2tw.go . "\.md$" tw2zh` ，轉簡體。

Markdown 格式預覽

- SUMMARY-github.md

版權聲明

譯者序

在上個世紀70年代，貝爾實驗室的Ken Thompson 和Dennis M. Ritchie 合作發明了UNIX 操作繫統，同時Dennis M. Ritchie 爲了解決UNIX 繫統的移植性問題而發明了C語言，貝爾實驗室的UNIX 和C語言兩大發明奠定了整個現代IT行業最重要的軟件基礎（目前的三大桌面操作繫統的中Linux 和Mac OS X 都是源於UINX 繫統，兩大移動平台的操作繫統iOS和Android也都是源於UNIX 繫統。C繫家族的編程語言占據統治地位達幾十年之久）。在UINX 和C語言發明40年之後，目前已經在Google工作的Ken Thompson 和Rob Pike （他們在貝爾實驗室時就是同事）、還有Robert Griesemer （設計了V8引擎和HotSpot虛擬機）一起合作，爲了解決在21世紀多核和網絡化環境下越來越複雜的編程問題而發明了Go語言。從Go語言庫早期代碼庫日誌可以看出它的演化歷程（Git用 `git log --before={2008-03-03} --reverse` 命令查看）：

```
C:\go\go-tip>hg log -r 0:4
changeset:   0:f6182e5abf5e
user:        Brian Kernighan <bwk>
date:        Tue Jul 18 19:05:45 1972 -0500
summary:     hello, world

changeset:   1:b66d0bf8da3e
user:        Brian Kernighan <bwk>
date:        Sun Jan 20 01:02:03 1974 -0400
summary:     convert to C

changeset:   2:ac3363d7e788
user:        Brian Kernighan <research!bwk>
date:        Fri Apr 01 02:02:04 1988 -0500
summary:     convert to Draft-Proposed ANSI C

changeset:   3:172d32922e72
user:        Brian Kernighan <bwk@research.att.com>
date:        Fri Apr 01 02:03:04 1988 -0500
summary:     last-minute fix: convert to ANSI C

changeset:   4:4e9a5b095532
user:        Robert Griesemer <gri@golang.org>
date:        Sun Mar 02 20:47:34 2008 -0800
summary:     Go spec starting point.


C:\go\go-tip>
```

從早期提交日誌中也可以看出，Go語言是從Ken Thompson 發明的B語言、Dennis M. Ritchie 發明的C語言逐步演化過來的，是C語言家族的成員，因此很多人將Go語言稱爲21世紀的C語言。縱觀這幾年來的發展趨勢，Go語言已經成爲雲計算、雲存儲時代最重要的基礎編程語言。

在C語言發明之後約5年的時間之後（1978年），Brian W. Kernighan 和Dennis M. Ritchie 合作編寫出版了C語言方面的經典教材《The C Programming Language 》，該書被譽爲C語言程序員的聖經，作者也被大家親切地稱爲K&R 。同樣在Go語言正式發布（2009年）約5年之後（2014年開始寫作，2015年出版），由Go語言核心糰隊成員Alan A. A. Donovan 和K&R 中的Brian W. Kernighan 合作編寫了Go語言方面的經典教材《The Go Programming Language 》。Go語言被譽爲21世紀的C語言，如果說K&R 所著的是聖經的舊約，那麼D&K所著的必將成爲聖經的新約。該書介紹了Go語言幾乎全部特性，併且隨着語言的深入層層遞進，對每個細節都解讀得非常細致，每一節內容都精綵不容錯過，是廣大Gopher的必讀書目。同時，大部分Go語言核心糰隊的成員都參與了該書校對工作，因此該書的質量是可以完全放心的。

同時，單憑閱讀和學習其語法結構併不能眞正地掌握一門編程語言，必鬚進行足夠多的編程實踐——親自編寫一些程序併研究學習別人寫的程序。要從利用Go語言良好的特性使得程序模塊化，充分利用Go的標準函數庫以Go語言自己的風格來編寫程序。書中包含了上百個精心挑選的習題，希望大家能先用自己的方式嚐試完成習題，然後再參考官方給出的解決方案。

該書英文版約從2015年10月開始公開發售，同時同步發售的還有日文版本。不過比較可惜的是，中文版併沒有在同步發售之列，甚至連中文版是否會引進、是由哪個出版社引進、即使引進將由何人來翻譯、何時能出版都成了一個祕密。中国的Go語言社區是全球最大的Go語言社區，我們從一開始就始終緊跟着Go語言的發展腳步。我們應該也完全有能力以中国Go語言社區的力量同步完成Go語言聖經中文版的翻譯工作。與此同時，国內有很多Go語言愛好者也在積極關註該書（本人也在第一時間購買了紙質版本，亞馬遜價格314人民幣 ）。爲了Go語言的學習和交流，大家決定合作免費翻譯該書。

翻譯工作從2015年11月20日前後開始，到2016年1月底初步完成，前後歷時約2個月時間。其中，chai2010 翻譯了前言、第2~4章、第10~13章，Xargin 翻譯了第1章、第6章、第8~9章，CrazySssst 翻譯了第5章，foreversmart 翻譯了第7章，大家共同參與了基本的校驗工作，還有其他一些朋友提供了積極的反饋建議。如果大家還有任何問題或建議，可以直接到中文版項目頁面提

交Issue，如果發現英文版原文在勘誤 中未提到的任何錯誤，可以直接去英文版項目 提交。

最後，希望這本書能夠幫助大家用Go語言快樂地編程。

2016年 1月 於 武漢

前言

“Go是一個開源的編程語言，它很容易用於構建簡單、可靠和高效的軟件。”（摘自Go語言官方網站：http://golang.org ）

Go語言由來自Google公司的Robert Griesemer ， Rob Pike 和Ken Thompson 三位大牛於2007年9月開始設計和實現，然後於2009年的11月對外正式發布（譯註：關於Go語言的創世紀過程請參考 http://talks.golang.org/2015/how-go-was-made.slide ）。語言及其配套工具的設計目標是具有表達力，高效的編譯和執行效率，有效地編寫高效和健壯的程序。

Go語言有着和C語言類似的語法外表，和C語言一樣是專業程序員的必備工具，可以用最小的代價獲得最大的戰果。 但是它不僅僅是一個更新的C語言。它還從其他語言借鑒了很多好的想法，同時避免引入過度的複雜性。 Go語言中和併發編程相關的特性是全新的也是有效的，同時對數據抽象和面向對象編程的支持也很靈活。 Go語言同時還集成了自動垃圾收集技術用於更好地管理內存。

Go語言尤其適合編寫網絡服務相關基礎設施，同時也適合開發一些工具軟件和繫統軟件。 但是Go語言確實是一個通用的編程語言，它也可以用在圖形圖像驅動編程、移動應用程序開發 和機器學習等諸多領域。目前Go語言已經成爲受歡迎的作爲無類型的腳本語言的替代者： 因爲Go編寫的程序通常比腳本語言運行的更快也更安全，而且很少會發生意外的類型錯誤。

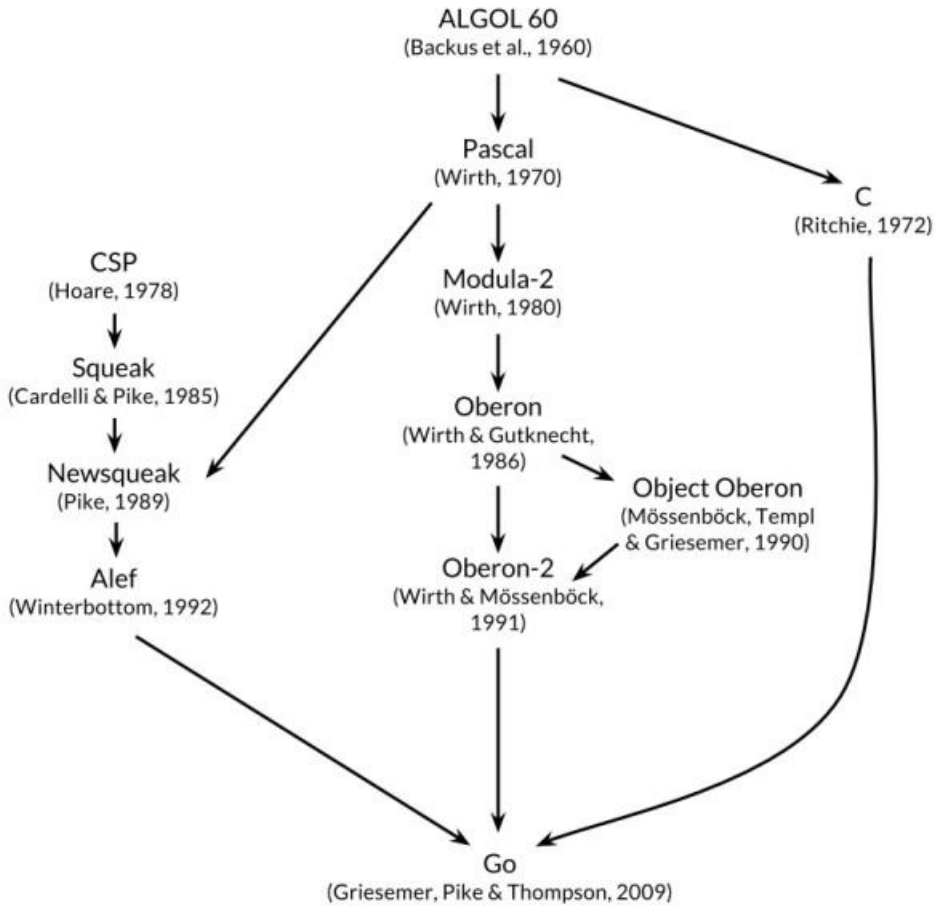
Go語言還是一個開源的項目，可以免費獲編譯器、庫、配套工具的源代碼。 Go語言的貢獻者來自一個活躍的全球社區。Go語言可以運行在類UNIX 繫統—— 比如Linux 、FreeBSD 、OpenBSD 、Mac OSX ——和Plan9 繫統和Microsoft Windows 操作繫統之上。 Go語言編寫的程序無需脩改就可以運行在上面這些環境。

本書是爲了幫助你開始以有效的方式使用Go語言，充分利用語言本身的特性和自帶的標準庫去編寫清晰地道的Go程序。

Go語言起源

編程語言的演化就像生物物種的演化類似，一個成功的編程語言的後代一般都會繼承它們祖先的優點；當然有時多種語言雜合也可能會產生令人驚訝的特性；還有一些激進的新特性可能併沒有先例。我們可以通過觀察編程語言和軟硬件環境是如何相互促進、相互影響的演化過程而學到很多。

下圖展示了有哪些早期的編程語言對Go語言的設計產生了重要影響。



Go語言有時候被描述爲“C類似語言”，或者是“21世紀的C語言”。Go從C語言繼承了相似的表達式語法、控製流結構、基礎數據類型、調用參數傳值、指針等很多思想，還有C語言一直所看中的編譯後機器碼的運行效率以及和現有操作繫統的無縫適配。

但是在Go語言的家族樹中還有其它的祖先。其中一個有影響力的分支來自Niklaus Wirth 所設計的Pascal)語言。然後Modula-2 語言激發了包的概念。然後Oberon)語言摒棄了模塊接口文件和模塊實現文件之間的區別。第二代的Oberon-2)語言直接影響了包的導入和聲明的語法，還有Oberon)語言的面向對象特性所提供的方法的聲明語法等。

Go語言的另一支祖先，帶來了Go語言區別其他語言的重要特性，靈感來自於貝爾實驗室的Tony Hoare 於1978年發表的鮮爲外界所知的關於併發研究的基礎文獻 *順序通信進程*（ *communicating sequential processes* ，縮寫爲CSP ）。在CSP 中，程序是一組中間沒有共享狀態的平行運行的處理過程，它們之間使用管道進行通信和控製同步。不過Tony Hoare 的CSP 隻是一個用於描述併發性基本概念的描述語言，併不是一個可以編寫可執行程序的通用編程語言。

接下來，Rob Pike和其他人開始不斷嚐試將CSP 引入實際的編程語言中。他們第一次嚐試引入CSP 特性的編程語言叫Squeak（老鼠間交流的語言），是一個提供鼠標和鍵盤事件處理的編程語言，它的管道是靜態創建的。然後是改進版的Newsqueak 語言，提供了類似C語言語句和表達式的語法和類似Pascal)語言的推導語法。Newsqueak是一個帶垃圾迴收的純函數式語言，它再次針對鍵盤、鼠標和窗口事件管理。但是在Newsqueak語言中管道是動態創建的，屬於第一類值, 可以保存到變量中。

在Plan9操作繫統中，這些優秀的想法被吸收到了一個叫Alef 的編程語言中。Alef試圖將Newsqueak語言改造爲繫統編程語言，但是因爲缺少垃圾迴收機製而導致併發編程很痛苦。（譯註：在Aelf之後還有一個叫Limbo 的編程語言，Go語言從其中借鑒了很多特性。 具體請參考Pike的講稿：http://talks.golang.org/2012/concurrency.slide#9 ）

Go語言的其他的一些特性零散地來自於其他一些編程語言；比如iota語法是從APL)語言借鑒，詞法作用域與嵌套函數來自於Scheme)語言（和其他很多語言）。當然，我們也可以從Go中發現很多創新的設計。比如Go語言的切片爲動態數組提供了有效的隨機存取的性能，這可能會讓人聯想到鏈表的底層的共享機製。還有Go語言新發明的defer語句。

Go語言項目

所有的編程語言都反映了語言設計者對編程哲學的反思，通常包括之前的語言所暴露的一些不足地方的改進。Go項目是在Google公司維護超級複雜的幾個軟件繫統遇到的一些問題的反思（但是這類問題絶不是Google公司所特有的）。

正如Rob Pike 所説，“軟件的複雜性是乘法級相關的”，通過增加一個部分的複雜性來脩複問題通常將慢慢地增加其他部分的複雜性。通過增加功能和選項和配置是脩複問題的最快的途徑，但是這很容易讓人忘記簡潔的內涵，卽使從長遠來看，簡潔依然是好軟件的關鍵因素。

簡潔的設計需要在工作開始的時候舍棄不必要的想法，併且在軟件的生命週期內嚴格區別好的改變或壞的改變。通過足夠的努力，一個好的改變可以在不破壞原有完整概念的前提下保持自適應，正如Fred Brooks 所説的“概念完整性”；而一個壞的改變則不能達到這個效果，它們僅僅是通過膚淺的和簡單的妥協來破壞原有設計的一致性。隻有通過簡潔的設計，才能讓一個繫統保持穩定、安全和持續的進化。

Go項目包括編程語言本身，附帶了相關的工具和標準庫，最後但併非代表不重要的，關於簡潔編程哲學的宣言。就事後諸葛的角度來看，Go語言的這些地方都做的還不錯：擁有自動垃圾迴收、一個包繫統、函數作爲一等公民、詞法作用域、繫統調用接口、隻讀的UTF8字符串等。但是Go語言本身隻有很少的特性，也不太可能添加太多的特性。例如，它沒有隱式的數值轉換，沒有構造函數和析構函數，沒有運算符重載，沒有默認參數，也沒有繼承，沒有泛型，沒有異常，沒有宏，沒有函數脩飾，更沒有線程局部存儲。但是語言本身是成熟和穩定的，而且承諾保證向後兼容：用之前的Go語言編寫程序可以用新版本的Go語言編譯器和標準庫直接構建而不需要脩改代碼。

Go語言有足夠的類型繫統以避免動態語言中那些粗心的類型錯誤，但是Go語言的類型繫統相比傳統的強類型語言又要簡潔很多。雖然有時候這會導致一個“無類型”的抽象類型概念，但是Go語言程序員併不需要像C++或Haskell程序員那樣糾結於具體類型的安全屬性。在實踐中Go語言簡潔的類型繫統給了程序員帶來了更多的安全性和更好的運行時性能。

Go語言鼓勵當代計算機繫統設計的原則，特別是局部的重要性。它的內置數據類型和大多數的準庫數據結構都經過精心設計而避免顯式的初始化或隱式的構造函數，因爲很少的內存分配和內存初始化代碼被隱藏在庫代碼中了。Go語言的聚合類型（結構體和數組）可以直接操作它們的元素，隻需要更少的存儲空間、更少的內存分配，而且指針操作比其他間接操作的語言也更有效率。由於現代計算機是一個併行的機器，Go語言提供了基於CSP的併發特性支持。Go語言的動態棧使得輕量級線程goroutine的初始棧可以很小，因此創建一個goroutine的代價很小，創建百萬級的goroutine完全是可行的。

Go語言的標準庫（通常被稱爲語言自帶的電池），提供了清晰的構建模塊和公共接口，包含I/O操作、文本處理、圖像、密碼學、網絡和分布式應用程序等，併支持許多標準化的文件格式和編解碼協議。庫和工具使用了大量的約定來減少額外的配置和解釋，從而最終簡化程序的邏輯，而且每個Go程序結構都是如此的相似，因此Go程序也很容易學習。使用Go語言自帶工具構建Go語言項目隻需要使用文件名和標識符名稱, 一個偶爾的特殊註釋來確定所有的庫、可執行文件、測試、基準測試、例子、以及特定於平台的變量、項目的文檔等；Go語言源代碼本身就包含了構建規范。

本書的組織

我們假設你已經有一個或多個其他編程語言的使用經歷，不管是類似C、c++或Java的編譯型語言，還是類似Python、Ruby、JavaScript的腳本語言，因此我們不會像對完全的編程語言初學者那樣解釋所有的細節。因爲Go語言的 變量、常量、表達式、控製流和函數等基本語法也是類似的。

第一章包含了本教程的基本結構，通過十幾個程序介紹了用Go語言如何實現 類似讀寫文件、文本格式化、創建圖像、網絡客戶端和服務器通訊等日常工作。

第二章描述了一個Go語言程序的基本元素結構、變量、定義新的類型、包和文件、以及作用域的概念。第三章討論了數字、布爾值、字符串和常量，併演示了如何顯示和處理Unicode字符。第四章描述了複合類型，從簡單的數組、字典、切片到動態列表。第五章涵蓋了函數，併討論了錯誤處理、panic和recover，還有defer語句。

第一章到第五章是基礎部分，對於任何主流命令式編程語言這個部分都是類似的。雖然有時候Go語言的語法和風格會有自己的特色，但是大多數程序員將能很快地適應。剩下的章節是Go語言中特有地方：方法、接口、併發、包、測試和反射等語言特性。

Go語言的面向對象是不同尋常的。它沒有類層次結構，甚至可以說沒有類；僅僅是通過組合（而不是繼承）簡單的對象來構建複雜的對象。方法不僅僅可以定義在結構體上, 而且可以定義在任何用戶自己定義的類型上；併且具體類型和抽象類型（接口）之間的關繫是隱式的，所以很多類型的設計者可能併不知道該類型到底滿足了哪些接口。方法將在第六章討論，接口將在第七章將討論。

第八章討論了基於順序通信進程(CSP)概念的併發編程，通過使用goroutines和channels處理併發編程。第九章則討論了更爲傳統的基於共享變量的併發編程。

第十章描述了包機製和包的組織結構。這一章還展示了如何有效的利用Go自帶的工具，通過一個命令提供了編譯、測試、基準測試、代碼格式化、文檔和許多其他任務。

第十一章討論了單元測試，Go語言的工具和標準庫中集成的輕量級的測試功能，從而避免了采用強大但複雜的測試框架。測試庫提供一些基本的構件，如果有必要可以用來構建更複雜的測試構件。

第十二章討論了反射，一個程序在運行期間來審視自己的能力。反射是一個強大的編程工具，不過要謹慎地使用；這一章通過用利用反射機製實現一些重要的Go語言庫函數來展示了反射的強大用法。第十三章解釋了底層編程的細節，通過使用unsafe包來繞過Go語言安全的類型繫統，當然有時這是必要的。

有些章節的後面可能會有一些練習，你可以根據你對Go語言的理解，然後脩改書中的例子來探索Go語言的其他用法。

書中所有的代碼都可以從 http://gopl.io 上的Git倉庫下載。go get命令可以根據每個例子的其導入路徑智能地獲取、構建併安裝。你隻需要選擇一個目録作爲工作空間，然後將GOPATH環境指向這個工作目録。

Go語言工具將在必要時創建的相應的目録。例如：

```
$ export GOPATH=$HOME/gobook    # 選擇工作目録
$ go get gopl.io/ch1/helloworld # 獲取/編譯/安裝
$ $GOPATH/bin/helloworld        # 運行程序
Hello, 世界                     # 這是中文
```

要運行這些例子, 你需要安裝Go1.5以上的版本.

```
$ go version
go version go1.5 linux/amd64
```

如果你用的是其他的操作繫統, 請參考 https://golang.org/doc/install 提供的説明安裝。

更多的信息

最佳的幫助信息來自Go語言的官方網站，https://golang.org，它提供了完善的參考文檔，包括編程語言規范和標準庫等諸多權威的幫助信息。同時也包含了如何編寫更地道的Go程序的基本教程，還有各種各樣的在線文本資源和視頻資源，它們是本書最有價值的補充。Go語言的官方博客 https://blog.golang.org 會不定期發布一些Go語言最好的實踐文章，包括當前語言的發展狀態、未來的計劃、會議報告和Go語言相關的各種會議的主題等信息（譯註： http://talks.golang.org/ 包含了官方收録的各種報告的講稿）。

在線訪問的一個有價值的地方是可以從web頁面運行Go語言的程序（而紙質書則沒有這麼便利了）。這個功能由來自 https://play.golang.org 的 Go Playground 提供，併且可以方便地嵌入到其他頁面中，例如 https://golang.org 的主頁，或 godoc 提供的文檔頁面中。

Playground可以簡單的通過執行一個小程序來測試對語法、語義和對程序庫的理解，類似其他很多語言提供的REPL即時運行的工具。同時它可以生成對應的url，非常適合共享Go語言代碼片段，滙報bug或提供反饋意見等。

基於 Playground 構建的 Go Tour，https://tour.golang.org，是一個繫列的Go語言入門教程，它包含了諸多基本概念和結構相關的併可在線運行的互動小程序。

當然，Playground 和 Tour 也有一些限製，它們隻能導入標準庫，而且因爲安全的原因對一些網絡庫做了限製。如果要在編譯和運行時需要訪問互聯網，對於一些更複製的實驗，你可能需要在自己的電腦上構建併運行程序。幸運的是下載Go語言的過程很簡單，從 https://golang.org 下載安裝包應該不超過幾分鐘（譯註：感謝偉大的長城，讓大陸的Gopher們都學會了自己打洞的基本生活技能，下載時間可能會因爲洞的大小等因素從幾分鐘到幾天或更久），然後就可以在自己電腦上編寫和運行Go程序了。

Go語言是一個開源項目，你可以在 https://golang.org/pkg 閱讀標準庫中任意函數和類型的實現代碼，和下載安裝包的代碼完全一致。這樣你可以知道很多函數是如何工作的， 通過挖掘找出一些答案的細節，或者僅僅是出於欣賞專業級Go代碼。

致謝

Rob Pike 和Russ Cox，以及很多其他Go糰隊的核心成員多次仔細閱讀了本書的手稿，他們對本書的組織結構和表述用詞等給出了很多寶貴的建議。在準備日文版翻譯的時候，Yoshiki Shibata更是仔細地審閱了本書的每個部分，及時發現了諸多英文和代碼的錯誤。我們非常感謝本書的每一位審閱者，併感謝對本書給出了重要的建議的Brian Goetz、Corey Kosak、Arnold Robbins、Josh Bleecher Snyder和Peter Weinberger等人。

我們還感謝Sameer Ajmani、Ittai Balaban、David Crawshaw、Billy Donohue、Jonathan Feinberg、Andrew Gerrand、Robert Griesemer、John Linderman、Minux Ma（譯註：中国人，Go糰隊成員。）、Bryan Mills、Bala Natarajan、Cosmos Nicolaou、Paul Staniforth、Nigel Tao（譯註：好像是陶哲軒的兄弟）以及Howard Trickey給出的許多有價值的建議。我們還要感謝David Brailsford和Raph Levien關於類型設置的建議。

我們的來自Addison-Wesley的編輯Greg Doench收到了很多幫助，從最開始就得到了越來越多的幫助。來自AW生產糰隊的John Fuller、Dayna Isley、Julie Nahil、Chuti Prasertsith到Barbara Wood，感謝你們的熱心幫助。

Alan Donovan 特別感謝：Sameer Ajmani、Chris Demetriou、Walt Drummond和Google公司的Reid Tatge允許他有充裕的時間去寫本書；感謝Stephen Donovan的建議和始終如一的鼓勵，以及他的妻子Leila Kazemi併沒有讓他爲了家庭瑣事而分心，併熱情堅定地支持這個項目。

Brian Kernighan 特別感謝：朋友和同事對他的耐心和寬容，讓他慢慢地梳理本書的寫作思路。同時感謝他的妻子Meg和其他很多朋友對他寫作事業的支持。

2015年 10月 於 紐約

第1章 入門

本章會介紹Go語言里的一些基本組件。我們希望用信息和例子盡快帶你入門。本章和之後章節的例子都是針對眞實的開發案例給出。本章我們隻是簡單地爲你介紹一些Go語言的入門例子，從簡單的文件處理、圖像處理到互聯網併發客戶端和服務端程序。當然，在第一章我們不會詳盡地一一去説明細枝末節，不過用這些程序來學習一門新語言肯定是很有效的。

當你學習一門新語言時，你會用這門新語言去重寫自己以前熟悉語言例子的傾向。在學習Go語言的過程中，盡量避免這麼做。我們會向你演示如何才能寫出好的Go語言程序，所以請使用這里的代碼作爲你寫自己的Go程序時的指南。

1.1. Hello, World

我們以1978年出版的C語言聖經《The C Programming Language》 中經典的“hello world”案例來開始吧（譯註：本書作者之一Brian W. Kernighan也是C語言聖經一書的作者）。C語言對Go語言的設計產生了很多影響。用這個例子，我們來講解一些Go語言的核心特性：

```
gopl.io/ch1/helloworld
package
 main

import
 "fmt"



func
 main() {
    fmt.Println("Hello, 世界"
)
}
```

Go是一門編譯型語言，Go語言的工具鏈將源代碼和其依賴一起打包，生成機器的本地指令(譯註：靜態編譯)。Go語言提供的工具可以通過go命令下的一繫列子命令來調用。最簡單的一個子命令就是run。這個命令會將一個或多個文件名以.go結尾的源文件，和關聯庫鏈接到一起，然後運行最終的可執行文件。（本書將用$表示命令行的提示符。）

```
$ go run helloworld.go
```

毫無意外，這個命令會輸出：

```
Hello, 世界
```

Go語言原生支持Unicode標準，所以你可以用Go語言處理世界上的任何自然語言。

如果你希望自己的程序不隻是簡單的一次性實驗，那麽你一定會希望能夠編譯這個程序，併且能夠將編譯結果保存下來以備將來之用。這個可以用build子命令來實現：

```
$ go build helloworld.go
```

這會創建一個名爲helloworld的可執行的二進製文件（譯註：在Windows繫統下生成的可執行文件是helloworld.exe，增加了.exe後綴名），之後你可以在任何時間去運行這個二進製文件，不需要其它的任何處理(譯註：因爲是靜態編譯，所以也不用擔心在繫統庫更新的時候衝突，幸福感滿滿)。

下面是運行我們的編譯結果樣例（譯註：在Windows繫統下在命令行直接輸入helloworld.exe命令運行）：

```
$ ./helloworld
Hello, 世界
```

本書中我們所有的例子都做了一個特殊標記，你可以通過這些標記在 http://gopl.io 在線網站上找到這些樣例代碼，比如這個

```
gopl.io/ch1/helloworld
```

如果你執行 `go get gopl.io/ch1/helloworld` 命令，go命令能夠自己從網上獲取到這些代碼（譯註：需要先安裝Git或Hg之類的版本管理工具，併將對應的命令添加到PATH環境變量中），併且將這些代碼放到對應的目録中（譯註：序言已經提及，需要先設置好GOPATH環境變量，下載的代碼會放在 $GOPATH/src/gopl.io/ch1/helloworld 目録）。更詳細的介紹在2.6和10.7章節中。

我們來討論一下程序本身。Go語言的代碼是通過package來組織的，package的概念和你知道的其它語言里的libraries或者modules概念比較類似。一個package會包含一個或多個.go結束的源代碼文件。每一個源文件都是以一個package xxx的聲明語句開頭的，比如我們的例子里就是package main。這行聲明語句表示該文件是屬於哪一個package，緊跟着是一繫列import的package名，表示這個文件中引入的package。再之後是本文件本身的代碼。

Go的標準庫已經提供了100多個package，用來完成一門程序語言的一些常見的基本任務，比如輸入、輸出、排序或者字符串/文本處理。比如fmt這個package，就包括接收輸入、格式化輸出的各種函數。Println是其中的一個常用的函數，可以用這個函數來打印一個或多個值，該函數會將這些參數用空格隔開進行輸出，併在輸出完畢之後在行末加上一個換行符。

package main是一個比較特殊的package。這個package里會定義一個獨立的程序，這個程序是可以運行的，而不是像其它package一樣對應一個library。在main這個package里，main函數也是一個特殊的函數，這是我們整個程序的入口（譯註：其實C繫語言差不多都是這樣）。main函數所做的事情就是我們程序做的事情。當然了，main函數一般是通過是調用其它packge里的函數來完成自己的工作，比如fmt.Println。

我們必鬚告訴編譯器如何要正確地執行這個源文件，需要用到哪些package，這就是import在這個文件里扮演的角色。上述的hello world例子隻用到了一個其它的package，就是fmt。一般情況下，需要import的package可能不隻一個。

這也正是因爲go語言必鬚引入所有要用到的package的原則，假如你沒有在代碼里import需要用到的package，程序將無法編譯通過，同時當你import了沒有用到的package，也會無法編譯通過（譯註：Go語言編譯過程沒有警告信息，爭議特性之一）。

import聲明必鬚跟在文件的package聲明之後。在import語句之後，則是各種方法、變量、常量、類型的聲明語句(分別用關鍵字func, var, const, type來進行定義)。這些內容的聲明順序併沒有什麼規定，可以隨便調整順序(譯註：最好還是定一下規范)。我們例子里的程序比較簡單，隻包含了一個函數。併且在該函數里也隻調用了一個其它函數。爲了節省空間，有些時候的例子我們會省略package和import聲明，但是讀者需要註意這些聲明是一定要包含在源文件里的。

一個函數的聲明包含func這個關鍵字、函數名、參數列表、返迴結果列表（我們例子里的main函數參數列表和返迴值都是空的）以及包含在大括號里的函數體。關於函數的更詳細描述在第五章。

Go語言是一門不需要分號作爲語句或者聲明結束的語言，除非要在一行中將多個語句、聲明隔開。然而在編譯時，編譯器會主動在一些特定的符號（譯註：比如行末是，一個標識符、一個整數、浮點數、虛數、字符或字符串文字、關鍵字break、continue、fallthrough或return中的一個、運算符和分隔符++、--、)、]或}中的一個） 後添加分號，所以在哪里加分號合適是取決於Go語言代碼的。例如：在Go語言中的函數聲明和 { 大括號必鬚在同一行，而在x + y這樣的表達式中，在+號後換行可以，但是在+號前換行則會有問題（譯註：以+結尾的話不會被插入分號分隔符，但是以x結尾的話則會被分號分隔符，從而導致編譯錯誤）。

Go語言在代碼格式上采取了很強硬的態度。gofmt工具會將你的代碼格式化爲標準格式（譯註：這個格式化工具沒有任何可以調整代碼格式的參數，Go語言就是這麼任性），併且go工具中的fmt子命令會自動對特定package下的所有.go源文件應用gofmt工具格式化。如果不指定package，則默認對當前目録下的源文件進行格式化。本書中的所有代碼已經是執行過gofmt後的標準格式代碼。你應該在自己的代碼上也執行這種格式化。規定一種標準的代碼格式可以規避掉無盡的無意義的撕逼（譯註：也導致了Go語言的TIOBE排名較低，因爲缺少撕逼的話題）。當然了，這可以避免由於代碼格式導致的邏輯上的歧義。

很多文本編輯器都可以設置爲保存文件時自動執行gofmt，所以你的源代碼應該總是會被格式化。這里還有一個相關的工具，goimports，會自動地添加你代碼里需要用到的import聲明以及需要移除的import聲明。這個工具併沒有包含在標準的分發包中，然而你可以自行安裝：

```
$ go get golang.org/x/tools/cmd/goimports
```

對於大多數用戶來說，下載、build package、運行測試用例、顯示Go語言的文檔等等常用功能都是可以用go的工具來實現的。這些工具的詳細介紹我們會在10.7節中提到。

1.2. 命令行參數

大多數的程序都是處理輸入，產生輸出；這也正是“計算”的定義。但是一個程序要如何獲取輸入呢？一些程序會生成自己的數據，但通常情況下，輸入都來自於程序外部：比如文件、網絡連接、其它程序的輸出、用戶的鍵盤、命令行的參數或其它類似輸入源。下面幾個例子會討論其中的一些輸入類型，首先是命令行參數。

os這個package提供了操作繫統無關（跨平台）的，與繫統交互的一些函數和相關的變量，運行時程序的命令行參數可以通過os包中一個叫Args的這個變量來獲取；當在os包外部使用該變量時，需要用os.Args來訪問。

os.Args這個變量是一個字符串（string）的slice（譯註：slice和Python語言中的切片類似，是一個簡版的動態數組），slice在Go語言里是一個基礎的數據結構，之後我們很快會提到。現在可以先把slice當一個簡單的元素序列，可以用類似s[i]的下標訪問形式獲取其內容，併且可以用形如s[m:n]的形式來獲取到一個slice的子集(譯註：和python里的語法差不多)。其長度可以用len(s)函數來獲取。和其它大多數編程語言類似，Go語言里的這種索引形式也采用了左閉右開區間，包括m~n的第一個元素，但不包括最後那個元素（譯註：比如a = [1, 2, 3, 4, 5], a[0:3] = [1, 2, 3]，不包含最後一個元素）。這樣可以簡化我們的處理邏輯。比如s[m:n]這個slice，$0 \le m \le n \le len(s)$，包含n-m個元素。

os.Args的第一個元素，即os.Args[0]是命令行執行時的命令本身；其它的元素則是執行該命令時傳給這個程序的參數。前面提到的切片表達式，s[m:n]會返迴第m到第n-1個元素，所以下一個例子里需要用到的os.Args[1:len(os.Args)]即是除了命令本身外的所有傳入參數。如果我們省略s[m:n]里的m和n，那麼默認這個表達式會填入0:len(s)，所以這里我們還可以省略掉n，寫成os.Args[1:]。

下面是一個Unix里echo命令的實現，這個命令會在單行內打印出命令行參數。這個程序import了兩個package，併且用括號把這兩個package包了起來，這是分別import各個package聲明的簡化寫法。當然了你分開來寫import也沒有什麼問題，隻是一般爲了方便我們都會像下面這樣來導入多個package。我們自己寫的導入順序併不重要，因爲gofmt工具會幫助我們按照字母順序來排列好這些導入包名。（本書中如果一個例子有多種版本時，我們會用編號標記出來）

```
gopl.io/ch1/echo1
// Echo1 prints its command-line arguments.

package
 main

import
 (
    "fmt"

    "os"

)

func
 main() {
    var
 s, sep string

    for
 i := 1
; i < len
(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "

    }
    fmt.Println(s)
}
```

Go語言里的註釋是以//來表示。//之後的內容一直到行末都是這條註釋的一部分，併且這些註釋會被編譯器忽略。

按照慣例，我們會在每一個package前面放上這個package的詳盡的註釋對其進行說明；對於一個main package來說，一般這段評論會包含幾句話來說明這個項目/程序整體是做什麽用的。

var關鍵字用來做變量聲明。這個程序聲明了s和sep兩個string變量。變量可以在聲明期間直接進行初始化。如果沒有顯式地初始化的話，Go語言會隱式地給這些未初始化的變量賦予對應其類型的零值，比如數值類型就是0，字符串類型就是空字符串“”。在這個例子里的s和sep被隱式地賦值爲了空字符串。在第2章中我們會更詳細地講解變量和聲明。

對於數字類型，Go語言提供了常規的數值計算和邏輯運算符。而對於string類型，+號表示字符串的連接（譯註：和C++或者js是一樣的）。所以下面這個表達式：

```
sep + os.Args[i]
```

表示將sep字符串和os.Args[i]字符串進行連接。我們在程序里用的另外一個表達式：

```
s += sep + os.Args[i]
```

會將sep與os.Args[i]連接，然後再將得到的結果與s進行連接併賦值運給s，這種方式和下面的表達是等價的：

```
s = s + sep + os.Args[i]
```

運算符+=是一個賦值運算符(assignment operator)，每一種數值和邏輯運算符，例如*或者+都有其對應的賦值運算符。

echo程序可以每循環一次輸出一個參數，不過我們這里的版本是不斷地將其結果連接到一個字符串的末尾。s這個字符串在聲明的時候是一個空字符串，而之後循環每次都會被在末尾添加一段字符串；第一次迭代之後，一個空格會被插入到字符串末尾，所以每插入一個新值，都會和前一個中間有一個空格隔開。這是一種非線性的操作，當我們的參數數量變得龐大的時候（當然不是說這里的echo，一般echo也不會有太多參數）其運行開銷也會變得龐大。下面我們會介紹一繫列的echo改進版，來應對這里說到的運行效率低下。

在for循環中，我們用到了i來做下標索引，可以看到我們用了:=符號來給i進行初始化和賦值，這是var xxx=yyy的一種簡寫形式，Go語言會根據等號右邊的值的類型自動判斷左邊的值類型，下一章會對這一點進行詳細說明。

自增表達式i++會爲i加上1；這和i += 1以及i = i + 1都是等價的。對應的還有i--是給i減去1。這些在Go語言里是語句，而不像C繫的其它語言里是表達式。所以在Go語言里j = i++是非法的，而且++和--都隻能放在變量名後面，因此--i也是非法的。

在Go語言里隻有for循環一種循環。當然了爲了滿足需求，Go的for循環有很多種形式，下面是其中的一種：

```
for
 initialization; condition; post {
    // zero or more statements

}
```

這里需要註意，for循環的兩邊是不需要像其它語言一樣寫括號的。併且左大括號需要和for語句在同一行。

initialization部分是可選的，如果你寫了這部分的話，在for循環之前這部分的邏輯會被執行。需要註意的是這部分必鬚是一個簡單的語句，也就是說是一個簡短的變量聲明，一個賦值語句，或是一個函數調用。condition部分必鬚是一個結果爲boolean值的表達式，在每次循環之前，語言都會檢查當前是否滿足這個條件，如果不滿足的話便會結束循環；post部分的語句則是在每次循環迭代結束之後被執行，之後conditon部分會在下一次執行前再被執行，依此往複。當condition條件里的判斷結果變爲false之後，循環即結束。

上面提到是for循環里的三個部分都是可以被省略的，如果你把initialization和post部分都省略的話，那麼連中間隔離他們的分號也是可以被省略的，比如下面這種for循環，就和傳統的while循環是一樣的：

```
// a traditional "while" loop

for
 condition {
    // ...

}
```

當然了，如果你連唯一的條件都省了，那麼for循環就會變成一個無限循環，像下面這樣：

```
// a traditional infinite loop

for
 {
    // ...

}
```

在無限循環中，你還是可以靠break或者return語句來終止掉循環。

如果你的遍歷對象是string或者slice類型值的話，還有另外一種循環的寫法，我們來看看另一個版本的echo：

```
gopl.io/ch1/echo2
// Echo2 prints its command-line arguments.

package
 main

import
 (
    "fmt"

)

func
 main() {
    s, sep := ""
, ""

    for
 _, arg := range
 os.Args[1
:] {
        s += sep + arg
        sep = " "

    }
    fmt.Println(s)
}
```

每一次循環迭代，range都會返迴一對結果；當前迭代的下標以及在該下標處的元素的值。在這個例子里，我們不需要這個下標，但是因爲range的處理要求我們必鬚要同時處理下標和值。我們可以在這里聲明一個接收index的臨時變量來解決這個問題，但是Go語言又不允許隻聲明而在後續代碼里不使用這個變量，如果你這樣做了編譯器會返迴一個編譯錯誤。

在Go語言中，應對這種情況的解決方法是用空白標識符，對，就是上面那個下劃線_。空白標識符可以在任何你接收自己不需要處理的值時使用。在這里，我們用它來忽略掉range返迴的那個沒用的下標值。大多數的Go程序員都會像上面這樣來寫類似的os.Args遍歷，由於遍歷os.Args的下標索引是隱式自動生成的，可以避免因顯式更新索引導致的錯誤。

上面這個版本將s和sep的聲明和初始化都放到了一起，但是我們可以等價地將聲明和賦值分開來寫，下面這些寫法都是等價的

```
s := ""

var
 s string

var
 s = ""

var
 s string
 = ""
```

那麼這些等價的形式應該怎麼做選擇呢？這里提供一些建議：第一種形式，隻能用在一個函數內部，而package級別的變量，禁止用這樣的聲明方式。第二種形式依賴於string類型的內部初始化機製，被初始化爲空字符串。第三種形式使用得很少，除非同時聲明多個變量。第四種形式會顯式地標明變量的類型，在多變量同時聲明時可以用到。實踐中你應該隻使用上面的前兩種形式，顯式地指定變量的類型，讓編譯器自己去初始化其值，或者直接用隱式初始化，表明初始值怎麼樣併不重要。

像上面提到的，每次循環迭代中字符串s都會得到一個新內容。+=語句會分配一個新的字符串，併將老字符串連接起來的值賦予給它。而目標字符串的老字面值在得到新值以後就失去了用處，這些臨時值會被Go語言的垃圾收集器斡掉。

如果不斷連接的數據量很大，那麼上面這種操作就是成本非常高的操作。更簡單併且有效的一種方式是使用strings包提供的Join函數，像下面這樣：

```
gopl.io/ch1/echo3
func
 main() {
    fmt.Println(strings.Join(os.Args[1
:], " "
))
}
```

最後，如果我們對輸出的格式也不是很關心，隻是想簡單地輸出值得的話，還可以像下面這麼寫，Println函數會爲我們自動格式化輸出。

```
fmt.Println(os.Args[1
:])
```

這個輸出結果和前面的string.Join得到的結果很相似，隻是被自動地放到了一個方括號里，對slice調用Println函數都會被打印成這樣形式的結果。

練習 1.1： 脩改echo程序，使其能夠打印os.Args[0]。

練習 1.2： 脩改echo程序，使其打印value和index，每個value和index顯示一行。

練習 1.3： 上手實踐前面提到的strings.Join和直接Println，併觀察輸出結果的區别。

1.3. 查找重複的行

文件拷貝、文件打印、文件蒐索、文件排序、文件統計類的程序一般都會有比較相似的程序結構：一個處理輸入的循環，在每一個輸入元素上執行計算處理，在處理的同時或者處理完成之後進行結果輸出。我們會展示一個叫dup程序的三個版本；這個程序的靈感來自於linux的uniq命令，我們的程序將會找到相鄰的重複的行。這個程序提供的模式可以很方便地被脩改來完成不同的需求。

第一個版本的dup會輸出標準輸入流中的出現多次的行，在行內容前會有其出現次數的計數。這個程序將引入if表達式，map內置數據結構和bufio的package。

```
gopl.io/ch1/dup1
// Dup1 prints the text of each line that appears more than

// once in the standard input, preceded by its count.

package
 main

import
 (
    "bufio"

    "fmt"

    "os"

)

func
 main() {
    counts := make
(map
[string
]int
)
    input := bufio.NewScanner(os.Stdin)
    for
 input.Scan() {
        counts[input.Text()]++
    }
    // NOTE:
 ignoring potential errors from input.Err()

    for
 line, n := range
 counts {
        if
 n > 1
 {
            fmt.Printf("%d\t%s\n"
, n, line)
        }
    }
}
```

和我們前面提到的for循環一樣，在if條件的兩邊，我們也不需要加括號，但是if表達式後的邏輯體的花括號是不能省略的。如果需要的話，像其它編程語言一樣，這個if表達式也可以有else部分，這部分邏輯會在if中的條件結果爲false時被執行。

map是Go語言內置的key/value型數據結構，這個數據結構能夠提供常數時間的存儲、獲取、測試操作。key可以是任意數據類型，隻要該類型能夠用==運算符來進行比較，string是最常用的key類型。而value類型的范圍就更大了，基本上什麽類型都是可以的。這個例子中的key都是string類型，value用的是int類型。我們用內置make函數來創建一個空的map，當然了，make方法還可

以有別的用處。在4.3章中我們還會對map進行更深入的討論。

dup程序每次讀取輸入的一行，這一行的內容會被當做一個map的key，而其value值會被+1。counts[input.Text()]++這個語句和下面的兩句是等價的：

```
line := input.Text()
counts[line] = counts[line] + 1
```

當然了，在這個例子里我們併不用擔心map在沒有當前的key時就對其進行++操作會有什麼問題，因爲Go語言在碰到這種情況時，會自動將其初始化爲0，然後再進行操作。

在這里我們又用了一個range的循環來打印結果，這次range是被用在map這個數據結構之上。這一次的情況和上次比較類似，range會返迴兩個值，一個key和在map對應這個key的value。對map進行range循環時，其迭代順序是不確定的，從實踐來看，很可能每次運行都會有不一樣的結果（譯註：這是Go語言的設計者有意爲之的，因爲其底層實現不保證插入順序和遍歷順序一致，也希望程序員不要依賴遍歷時的順序，所以榦脆直接在遍歷的時候做了隨機化處理，醉了。補充：好像說隨機序可以防止某種類型的攻擊，雖然不太明白，但是感覺還蠻厲害的），來避免程序員在業務中依賴遍歷時的順序。

然後輪到我們例子中的bufio這個package了，這個package主要的目的是幫助我們更方便有效地處理程序的輸入和輸出。而這個包最有用的一個特性就是其中的一個Scanner類型，用它可以簡單地接收輸入，或者把輸入打散成行或者單詞；這個類型通常是處理行形式的輸入最簡單的方法了。

本程序中用了一個短變量聲明，來創建一個buffio.Scanner對象：

```
input := bufio.NewScanner(os.Stdin)
```

scanner對象可以從程序的標準輸入中讀取內容。對input.Scanner的每一次調用都會調入一個新行，併且會自動將其行末的換行符去掉；其結果可以用input.Text()得到。Scan方法在讀到了新行的時候會返迴true，而在沒有新行被讀入時，會返迴false。

例子中還有一個fmt.Printf，這個函數和C繫的其它語言里的那個printf函數差不多，都是格式化輸出的方法。fmt.Printf的第一個參數卽是輸出內容的格式規約，每一個參數如何格式化是取決於在格式化字符串里出現的“轉換字符”，這個字符串是跟着%號後的一個字母。比如%d表示以一個整數的形式來打印一個變量，而%s，則表示以string形式來打印一個變量。

Printf有一大堆這種轉換，Go語言程序員把這些叫做verb（動詞）。下面的表格列出了常用的動詞，當然了不是全部，但基本也夠用了。

```
%d          int變量
%x, %o, %b  分別爲16進製，8進製，2進製形式的int
%f, %g, %e  浮點數： 3.141593 3.141592653589793 3.141593e+00
%t          布爾變量：true 或 false
%c          rune (Unicode碼點)，Go語言里特有的Unicode字符類型
%s          string
%q          帶雙引號的字符串 "abc" 或 帶單引號的 rune 'c'
%v          會將任意變量以易讀的形式打印出來
%T          打印變量的類型
%%          字符型百分比標誌（%符號本身，沒有其他操作）
```

dup1中的程序還包含了一個\t和\n的格式化字符串。在字符串中會以這些特殊的轉義字符來表示不可見字符。Printf默認不會在輸出內容後加上換行符。按照慣例，用來格式化的函數都會在末尾以f字母結尾（譯註：f後綴對應format或fmt縮寫），比如log.Printf，fmt.Errorf，同時還有一繫列對應以ln結尾的函數（譯註：ln後綴對應line縮寫），這些函數默認以%v來格式化他們的參數，併且會在輸出結束後在最後自動加上一個換行符。

許多程序從標準輸入中讀取數據，像上面的例子那樣。除此之外，還可能從一繫列的文件中讀取。下一個dup程序就是從標準輸入中讀到一些文件名，用os.Open函數來打開每一個文件獲取內容的。

```
gopl.io/ch1/dup2
// Dup2 prints the count and text of lines that appear more than once

// in the input.  It reads from stdin or from a list of named files.

package
 main

import
 (
    "bufio"

    "fmt"

    "os"

)

func
 main() {
    counts := make
(map
[string
]int
)
    files := os.Args[1
:]
    if
 len
(files) == 0
 {
        countLines(os.Stdin, counts)
    } else
 {
        for
 _, arg := range
 files {
            f, err := os.Open(arg)
            if
 err != nil
 {
                fmt.Fprintf(os.Stderr, "dup2: %v\n"
, err)
                continue

            }
            countLines(f, counts)
            f.Close()
        }
```

```
    }
    for
 line, n := range
 counts {
        if
 n > 1
 {
            fmt.Printf("%d\t%s\n"
, n, line)
        }
    }
}

func
 countLines(f *os.File, counts map
[string
]int
) {
    input := bufio.NewScanner(f)
    for
 input.Scan() {
        counts[input.Text()]++
    }
    // NOTE:
 ignoring potential errors from input.Err()

}
```

os.Open函數會返迴兩個值。第一個值是一個打開的文件類型(*os.File)，這個對象在下面的程序中被Scanner讀取。

os.Open返迴的第二個值是一個Go語言內置的error類型。如果這個error和內置值的nil（譯註：相當於其它語言里的NULL）相等的話，説明文件被成功的打開了。之後文件被讀取，一直到文件的最後，文件的Close方法關閉該文件，併釋放相應的占用一切資源。另一方面，如果err的值不是nil的話，那説明在打開文件的時候出了某種錯誤。這種情況下，error類型的值會描述具體的問題。我們例子里的簡單錯誤處理會在標準錯誤流中用Fprintf和%v來格式化該錯誤字符串。然後繼續處理下一個文件；continue語句會直接跳過之後的語句，直接開始執行下一個循環迭代。

我們在本書中早期的例子中做了比較詳盡的錯誤處理，當然了，在實際編碼過程中，像os.Open這類的函數是一定要檢查其返迴的error值的；爲了減少例子程序的代碼量，我們姑且簡化掉這些不太可能返迴錯誤的處理邏輯。後面的例子里我們會跳過錯誤檢查。在5.4節中我們會對錯誤處理做更詳細的闡述。

讀者可以再觀察一下上面的例子，我們的countLines函數是在其聲明之前就被調用了。在Go語言里，函數和包級別的變量可以以任意的順序被聲明，併不影響其被調用。（譯註：最好還是遵循一定的規范）

再來講講map這個數據結構，map是用make函數創建的數據結構的一個引用。當一個map被作爲參數傳遞給一個函數時，函數接收到的是一份引用的拷貝，雖然本身併不是一個東西，但因爲他們指向的是同一塊數據對象（譯註：類似於C++里的引用傳遞），所以你在函數里對map里的值進行脩改時，原始的map內的值也會改變。在我們的例子中，我們在countLines函數中插入到counts這個map里的值，在主函數中也是看得到的。

上面這個版本的dup是以流的形式來處理輸入，併將其打散爲行。理論上這些程序也是可以以二進製形式來處理輸入的。我們也可以一次性的把整個輸入內容全部讀到內存中，然後再把其分割爲多行，然後再去處理這些行內的數據。下面的dup3這個例子就是以這種形式來進行操作的。這個例子引入了一個新函數ReadFile（從io/ioutil包提供），這個函數會把一個指定名字的文件內容一次性調入，之後我們用strings.Split函數把文件分割爲多個子字符串，併存儲到slice結構中。（Split函數是strings.Join的逆函數，Join函數之前提到過）

我們簡化了dup3這個程序。首先，它隻讀取命名的文件，而不去讀標準輸入，因爲ReadFile函數需要一個文件名參數。其次，我們將行計數邏輯移迴到了main函數，因爲現在這個邏輯隻有一個地方需要用到。

```
gopl.io/ch1/dup3
package
 main

import
 (
    "fmt"

    "io/ioutil"

    "os"

    "strings"

)

func
 main() {
    counts := make
(map
[string
]int
)
    for
 _, filename := range
 os.Args[1
:] {
        data, err := ioutil.ReadFile(filename)
        if
 err != nil
 {
            fmt.Fprintf(os.Stderr, "dup3: %v\n"
, err)
            continue

        }
        for
 _, line := range
 strings.Split(string
(data), "\n"
) {
            counts[line]++
        }
    }
    for
 line, n := range
 counts {
        if
 n > 1
```

```
 {
            fmt.Printf("%d\t%s\n"
, n, line)
        }
    }
}
```

ReadFile函數返迴一個byte的slice，這個slice必鬚被轉換爲string，之後才能夠用string.Split方法來進行處理。我們在3.5.4節中會更詳細地講解string和byte slice（字節數組）。

在更底層一些的地方，bufio.Scanner，ioutil.ReadFile和ioutil.WriteFile使用的是*os.File的Read和Write方法，不過一般程序員併不需要去直接了解到其底層實現細節，在bufio和io/ioutil包中提供的方法已經足夠好用。

練習 1.4： 脩改dup2，使其可以打印重複的行分別出現在哪些文件。

1.4. GIF動畫

下面的程序會演示Go語言標準庫里的image這個package的用法，我們會用這個包來生成一繫列的bit-mapped圖，然後將這些圖片編碼爲一個GIF動畫。我們生成的圖形名字叫利薩如圖形(Lissajous figures)，這種效果是在1960年代的老電影里出現的一種視覺特效。它們是協振子在兩個緯度上振動所産生的麕線，比如兩個sin正絃波分別在x軸和y軸輸入會産生的麕線。圖1.1是這樣的一個例子：

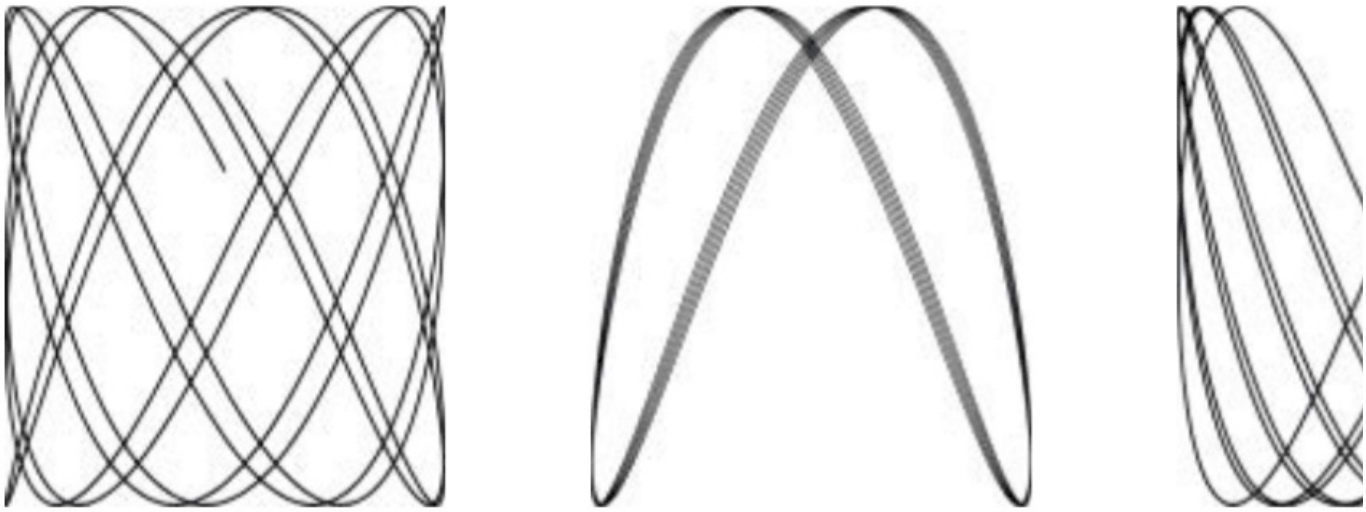
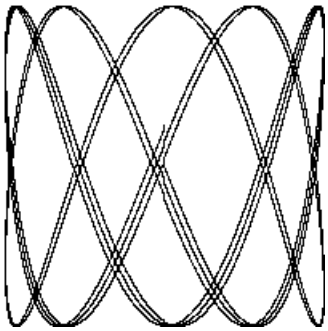


Figure 1.1. Four Lissajous

譯註：要看這個程序的結果，需要將標準輸出重定向到一個GIF圖像文件（使用 `./lissajous > output.gif` 命令）。下面是GIF圖像動畫效果：



這段代碼里我們用了一些新的結構，包括const聲明，struct結構體類型，複合聲明。和我們舉的其它的例子不太一樣，這一個例子包含了浮點數運算。這些概念我們隻在這里簡單地説明一下，之後的章節會更詳細地講解。

```
gopl.io/ch1/lissajous

// Lissajous generates GIF animations of random Lissajous figures.

package
 main

import
 (
    "image"

    "image/color"
```

```
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)
var
 palette = []color.Color{color.White, color.Black}
const
 (
    whiteIndex = 0
 // first color in palette
    blackIndex = 1
 // next color in palette
)
func
 main() {
    lissajous(os.Stdout)
}
func
 lissajous(out io.Writer) {
    const
 (
        cycles  = 5
     // number of complete x oscillator revolutions
        res     = 0.001
 // angular resolution
        size    = 100
   // image canvas covers [-size..+size]
        nframes = 64
    // number of animation frames
        delay   = 8
     // delay between frames in 10ms units
    )
    freq := rand.Float64() * 3.0
```

```
 // relative frequency of y oscillator

    anim := gif.GIF{LoopCount: nframes}
    phase := 0.0
 // phase difference

    for
 i := 0
; i < nframes; i++ {
        rect := image.Rect(0
, 0
, 2
*size+1
, 2
*size+1
)
        img := image.NewPaletted(rect, palette)
        for
 t := 0.0
; t < cycles*2
*math.Pi; t += res {
            x := math.Sin(t)
            y := math.Sin(t*freq + phase)
            img.SetColorIndex(size+int
(x*size+0.5
), size+int
(y*size+0.5
),
bla    kIndex)
        }
        phase += 0.1

        anim.Delay = append
(anim.Delay, delay)
        anim.Image = append
(anim.Image, img)
    }
    gif.EncodeAll(out, &anim) // NOTE:
 ignoring encoding errors

}
```

當我們import了一個包路徑包含有多個單詞的package時，比如image/color（image和color兩個單詞），通常我們隻需要用最後那個單詞表示這個包就可以。所以當我們寫color.White時，這個變量指向的是image/color包里的變量，同理gif.GIF是屬於image/gif包里的變量。

這個程序里的常量聲明給出了一繫列的常量值，常量是指在程序編譯後運行時始終都不會變化的值，比如圈數、幀數、延遲值。常量聲明和變量聲明一般都會出現在包級別，所以這些常量在整個包中都是可以共享的，或者你也可以把常量聲明定義在函數體內部，那麼這種常量就隻能在函數體內用。目前常量聲明的值必鬚是一個數字值、字符串或者一個固定的boolean值。

[]color.Color{...}和gif.GIF{...}這兩個表達式就是我們說的複合聲明（4.2和4.4.1節有說明）。這是實例化Go語言里的複合類型的一種寫法。這里的前者生成的是一個slice切片，後者生成的是一個struct結構體。

gif.GIF是一個struct類型（參考4.4節）。struct是一組值或者叫字段的集合，不同的類型集合在一個struct可以讓我們以一個統一的單元進行處理。anim是一個gif.GIF類型的struct變量。這種寫法會生成一個struct變量，併且其內部變量LoopCount字段會被設置爲nframes；而其它的字段會被設置爲各自類型默認的零值。struct內部的變量可以以一個點(.)來進行訪問，就像在最後兩個賦值語句中顯式地更新了anim這個struct的Delay和Image字段。

lissajous函數內部有兩層嵌套的for循環。外層循環會循環64次，每一次都會生成一個單獨的動畫幀。它生成了一個包含兩種顏色的201&201大小的圖片，白色和黑色。所有像素點都會被默認設置爲其零值（也就是palette里的第0個值），這里我們設置的是白色。每次外層循環都會生成一張新圖片，併將一些像素設置爲黑色。其結果會append到之前結果之後。這里我們用到了append(參考4.2.1)這個內置函數，將結果appen到anim中的幀列表末尾，併會設置一個默認的80ms的延遲值。最終循環結束，所有的延遲值也被編碼進了GIF圖片中，併將結果寫入到輸出流。out這個變量是io.Writer類型，這個類型讓我們可以可以讓我們把輸出結果寫到很多目標，很快我們就可以看到了。

內存循環設置了兩個偏振。x軸偏振使用的是一個sin函數。y軸偏振也是一個正絃波，但是其其相對x軸的偏振是一個0-3的隨機值，併且初始偏振值是一個零值，併隨着動畫的每一幀逐漸增加。循環會一直跑到x軸完成五次完整的循環。每一步它都會調用SetColorIndex來爲(x, y)點來染黑色。

main函數調用了lissajous函數，併且用它來向標準輸出中打印信息，所以下面這個命令會像圖1.1中產生一個GIF動畫。

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

練習 1.5： 脩改前面的Lissajous程序里的調色闆，由緑色改爲黑色。我們可以用color.RGBA{0xRR, 0xGG, 0xBB}來得到#RRGGBB這個色值，三個十六進製的字符串分別代表紅、緑、藍像素。

練習 1.6： 脩改Lissajous程序，脩改其調色闆來生成更豐富的顔色，然後脩改SetColorIndex的第三個參數，看看顯示結果吧。

1.5. 獲取URL

對於很多現代應用來説，訪問互聯網上的信息和訪問本地文件繫統一樣重要。Go語言在net這個強大package的幫助下提供了一繫列的package來做這件事情，使用這些包可以更簡單地用網絡收發信息，還可以建立更底層的網絡連接，編寫服務器程序。在這些情景下，Go語言原生的併發特性（在第八章中會介紹）就顯得尤其好用了。

爲了最簡單地展示基於HTTP獲取信息的方式，下面給出一個示例程序fetch，這個程序將獲取對應的url，併將其源文本打印出來；這個例子的靈感來源於curl工具（譯註：unix下的一個網絡相關的工具）。當然了，curl提供的功能更爲複雜豐富，這里我們隻編寫最簡單的樣例。之後我們還會在本書中經常用到這個例子。

```
gopl.io/ch1/fetch
// Fetch prints the content found at a URL.

package
 main

import
 (
    "fmt"

    "io/ioutil"

    "net/http"

    "os"

)

func
 main() {
    for
 _, url := range
 os.Args[1
:] {
        resp, err := http.Get(url)
        if
 err != nil
 {
            fmt.Fprintf(os.Stderr, "fetch: %v\n"
, err)
            os.Exit(1
)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if
 err != nil
 {
            fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n"
, url, err)
            os.Exit(1
)
        }
        fmt.Printf("%s"
, b)
    }
}
```

這個程序從兩個package中導入了函數，net/http和io/ioutil包，http.Get函數是創建HTTP請求的函數，如果獲取過程沒有出錯，那麼會在resp這個結構體中得到訪問的請求結果。resp的Body字段包括一個可讀的服務器響應流。這之後ioutil.ReadAll函數從response中讀取到全部內容；其結果保存在變量b中。resp.Body.Close這一句會關閉resp的Body流，防止資源洩露，Printf函數會將結果b寫出到標準輸出流中。

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>title>
...
```

HTTP請求如果失敗了的話，會得到下面這樣的結果：

```
$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host
```

譯註：在大天朝的網絡環境下很容易重現這種錯誤，下面是Windows下運行得到的錯誤信息：

```
$ go run main.go http://gopl.io
fetch: Get http://gopl.io: dial tcp: lookup gopl.io: getaddrinfow: No such host is known.
```

無論哪種失敗原因，我們的程序都用了os.Exit函數來終止進程，併且返迴一個status錯誤碼，其值爲1。

練習 1.7： 函數調用io.Copy(dst, src)會從src中讀取內容，併將讀到的結果寫入到dst中，使用這個函數替代掉例子中的ioutil.ReadAll來拷貝響應結構體到os.Stdout，避免申請一個緩衝區（例子中的b）來存儲。記得處理io.Copy返迴結果中的錯誤。

練習 1.8： 脩改fetch這個范例，如果輸入的url參數沒有 `http://` 前綴的話，爲這個url加上該前綴。你可能會用到strings.HasPrefix這個函數。

練習 1.9： 脩改fetch打印出HTTP協議的狀態碼，可以從resp.Status變量得到該狀態碼。

1.6. 併發獲取多個URL

Go語言最有意思併且最新奇的特性就是其對併發編程的支持了。併發編程是一個大話題，在第八章和第九章中會專門講到。這里我們隻淺嚐輒止地來體驗一下Go語言里的goroutine和channel。

下面的例子fetchall，和上面的fetch程序所要做的工作是一致的，但是這個fetchall的特別之處在於它會同時去獲取所有的URL，所以這個程序的獲取時間不會超過執行時間最長的那一個任務，而不會像前面的fetch程序一樣，執行時間是所有任務執行時間之和。這次的fetchall程序隻會打印獲取的內容大小和經過的時間，不會像上面那樣打印出獲取的內容。

```
gopl.io/ch1/fetchall
// Fetchall fetches URLs in parallel and reports their times and sizes.

package
 main

import
 (
    "fmt"

    "io"

    "io/ioutil"

    "net/http"

    "os"

    "time"

)

func
 main() {
    start := time.Now()
    ch := make
(chan
 string
)
    for
 _, url := range
 os.Args[1
:] {
        go
 fetch(url, ch) // start a goroutine

    }
    for
 range
 os.Args[1
:] {
        fmt.Println(<-ch) // receive from channel ch

    }
```

```
    }
    fmt.Printf("%.2fs elapsed\n"
, time.Since(start).Seconds())
}

func
 fetch(url string
, ch chan
<- string
) {
    start := time.Now()
    resp, err := http.Get(url)
    if
 err != nil
 {
        ch <- fmt.Sprint(err) // send to channel ch

        return

    }
    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // don't leak resources

    if
 err != nil
 {
        ch <- fmt.Sprintf("while reading %s: %v"
, url, err)
        return

    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs  %7d  %s"
, secs, nbytes, url)
}
```

下面是一個使用的例子

```
$ go build gopl.io/ch1/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s     6852  https://godoc.org
0.16s     7261  https://golang.org
0.48s     2475  http://gopl.io
0.48s elapsed
```

goroutine是一種函数的併發執行方式，而channel是用來在goroutine之間進行參數傳遞。main函數也是運行在一個goroutine中，而go function則表示創建一個新的goroutine，併在這個這個新的goroutine里執行這個函数。

main函數中用make函數創建了一個傳遞string類型參數的channel，對每一個命令行參數，我們都用go這個關鍵字來創建一個goroutine，併且讓函數在這個goroutine異步執行http.Get方法。這個程序里的io.Copy會把響應的Body內容拷貝到ioutil.Discard輸出流中（譯註：這是一個垃圾桶，可以向里面寫一些不需要的數據），因爲我們需要這個方法返迴的字節數，但是又不想要其內容。每當請求返迴內容時，fetch函數都會往ch這個channel里寫入一個字符串，由main函數里的第二個for循環來處理併打印channel里的這個字符串。

當一個goroutine嚐試在一個channel上做send或者receive操作時，這個goroutine會阻塞在調用處，直到另一個goroutine往這個channel里寫入、或者接收了值，這樣兩個goroutine才會繼續執行操作channel完成之後的邏輯。在這個例子中，每一個fetch函數在執行時都會往channel里發送一個值(ch <- expression)，主函數接收這些值(<-ch)。這個程序中我們用main函數來所有fetch函數傳迴的字符串，可以避免在goroutine異步執行時同時結束。

練習 1.10： 找一個數據量比較大的網站，用本小節中的程序調研網站的緩存策略，對每個URL執行兩遍請求，查看兩次時間是否有較大的差別，併且每次獲取到的響應內容是否一致，脩改本節中的程序，將響應結果輸出，以便於進行對比。

1.7. Web服務

Go語言的內置庫讓我們寫一個像fetch這樣例子的web服務器變得異常地簡單。在本節中，我們會展示一個微型服務器，這個服務的功能是返迴當前用戶正在訪問的URL。也就是說比如用戶訪問的是 http://localhost:8000/hello ，那麼響應是URL.Path = "hello"。

```
gopl.io/ch1/server1
// Server1 is a minimal "echo" server.

package
 main

import
 (
    "fmt"

    "log"

    "net/http"

)

func
 main() {
    http.HandleFunc("/"
, handler) // each request calls handler

    log.Fatal(http.ListenAndServe("localhost:8000"
, nil
))
}

// handler echoes the Path component of the request URL r.

func
 handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n"
, r.URL.Path)
}
```

我們隻用了八九行代碼就實現了一個個Web服務程序，這都是多虧了標準庫里的方法已經幫我們處理了大量的工作。main函數會將所有發送到/路徑下的請求和handler函數關聯起來，/開頭的請求其實就是所有發送到當前站點上的請求，我們的服務跑在了8000端口上。發送到這個服務的“請求”是一個http.Request類型的對象，這個對象中包含了請求中的一繫列相關字段，其中就包括我們需要的URL。當請求到達服務器時，這個請求會被傳給handler函數來處理，這個函數會將/hello這個路徑從請求的URL中解析出來，然後把其發送到響應中，這里我們用的是標準輸出流的fmt.Fprintf。Web服務會在第7.7節中詳細闡述。

讓我們在後台運行這個服務程序。如果你的操作繫統是Mac OS X或者Linux，那麼在運行命令的末尾加上一個&符號，即可讓程序簡單地跑在後台，而在windows下，你需要在另外一個命令行窗口去運行這個程序了。

```
$ go run src/gopl.io/ch1/server1/main.go &
```

現在我們可以通過命令行來發送客戶端請求了：

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"
```

另外我們還可以直接在瀏覽器里訪問這個URL，然後得到返迴結果，如圖1.2：

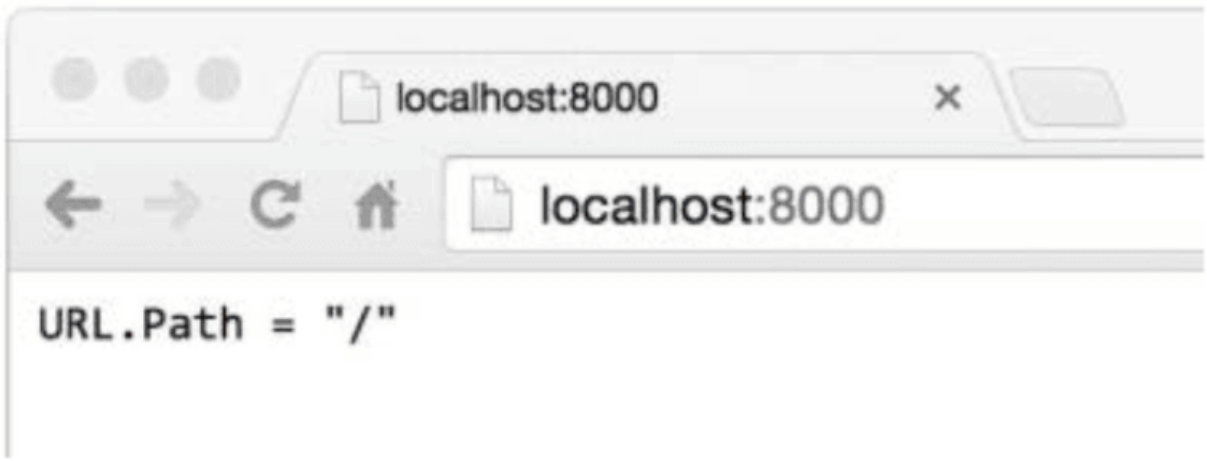


Figure 1.2. A response from the echo server.

在這個服務的基礎上疊加特性是很容易的。一種比較實用的脩改是爲訪問的url添加某種狀態。比如，下面這個版本輸出了同樣的內容，但是會對請求的次數進行計算；對URL的請求結果會包含各種URL被訪問的總次數，直接對/count這個URL的訪問要除外。

```
gopl.io/ch1/server2
// Server2 is a minimal "echo" and counter server.

package
 main

import
 (
    "fmt"

    "log"

    "net/http"

    "sync"

)

var
 mu sync.Mutex
```

```
var
 count int


func
 main() {
    http.HandleFunc("/"
, handler)
    http.HandleFunc("/count"
, counter)
    log.Fatal(http.ListenAndServe("localhost:8000"
, nil
))
}

// handler echoes the Path component of the requested URL.

func
 handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n"
, r.URL.Path)
}

// counter echoes the number of calls so far.

func
 counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n"
, count)
    mu.Unlock()
}
```

這個服務器有兩個請求處理函數，請求的url會決定具體調用哪一個：對/count這個url的請求會調用到count這個函數，其它所有的url都會調用默認的處理函數。如果你的請求pattern是以/結尾，那麼所有以該url爲前綴的url都會被這條規則匹配。在這些代碼的背後，服務器每一次接收請求處理時都會另起一個goroutine，這樣服務器就可以同一時間處理多數請求。然而在併發情況下，假如眞的有兩個請求同一時刻去更新count，那麼這個值可能併不會被正確地增加；這個程序可能會被引發一個嚴重的bug：競態條件（參見9.1）。爲了避免這個問題，我們必鬚保證每次脩改變量的最多隻能有一個goroutine，這也就是代碼里的mu.Lock()和mu.Unlock()調用將脩改count的所有行爲包在中間的目的。第九章中我們會進一步講解共享變量。

下面是一個更爲豐富的例子，handler函數會把請求的http頭和請求的form數據都打印出來，這樣可以讓檢查和調試這個服務更爲方便：

```
gopl.io/ch1/server3
// handler echoes the HTTP request.

func
 handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n"
, r.Method, r.URL, r.Proto)
    for
 k, v := range
 r.Header {
        fmt.Fprintf(w, "Header[%q] = %q\n"
, k, v)
    }
    fmt.Fprintf(w, "Host = %q\n"
, r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n"
, r.RemoteAddr)
    if
 err := r.ParseForm(); err != nil
 {
        log.Print(err)
    }
    for
 k, v := range
 r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n"
, k, v)
    }
}
```

我們用http.Request這個struct里的字段來輸出下面這樣的內容：

```
GET /?q=query HTTP/1.1
Header["Accept-Encoding"] = ["gzip, deflate, sdch"] Header["Accept-Language"] = ["en-US,en;q=0.8"]
Header["Connection"] = ["keep-alive"]
Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."] Header["User-Agent"] = ["Mozilla/5.0
(Macintosh; Intel Mac OS X 10_7_5)..."] Host = "localhost:8000"
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]
```

可以看到這里的ParseForm被嵌套在了if語句中。Go語言允許這樣的一個簡單的語句結果作爲循環的變量聲明出現在if語句的最前面，這一點對錯誤處理很有用處。我們還可以像下面這樣寫（當然看起來就長了一些）：

```
err := r.ParseForm()
if
 err != nil
 {
    log.Print(err)
}
```

用if和ParseForm結合可以讓代碼更加簡單，併且可以限製err這個變量的作用域，這麽做是很不錯的。我們會在2.7節中講解作用域。

在這些程序中，我們看到了很多不同的類型被輸出到標準輸出流中。比如前面的fetch程序，就把HTTP的響應數據拷貝到了os.Stdout，或者在lissajous程序里我們輸出的是一個文件。fetchall程序則完全忽略到了HTTP的響應體，隻是計算了一下響應體的大小，這個程序中把響應體拷貝到了ioutil.Discard。在本節的web服務器程序中則是用fmt.Fprintf直接寫到了http.ResponseWriter中。

盡管這三種具體的實現流程併不太一樣，他們都實現一個共同的接口，即當它們被調用需要一個標準流輸出時都可以滿足。這個接口叫作io.Writer，在7.1節中會詳細討論。

Go語言的接口機製會在第7章中講解，爲了在這里簡單說明接口能做什麽，讓我們簡單地將這里的web服務器和之前寫的lissajous函數結合起來，這樣GIF動畫可以被寫到HTTP的客戶端，而不是之前的標準輸出流。隻要在web服務器的代碼里加入下面這幾行。

```
handler := func
(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
}
http.HandleFunc("/"
, handler)
```

或者另一種等價形式：

```
http.HandleFunc("/"
, func
(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
})
```

HandleFunc函數的第二個參數是一個函數的字面值，也就是一個在使用時定義的匿名函數。這些內容我們會在5.6節中講解。

做完這些脩改之後，在瀏覽器里訪問 http://localhost:8000 。每次你載入這個頁面都可以看到一個像圖1.3那樣的動畫。

練習 1.12： 脩改Lissajour服務，從URL讀取變量，比如你可以訪問 http://localhost:8000/?cycles=20 這個URL，這樣訪問可以將程序里的cycles默認的5脩改爲20。字符串轉換爲數字可以調用strconv.Atoi函數。你可以在dodoc里查看strconv.Atoi的詳細説明。

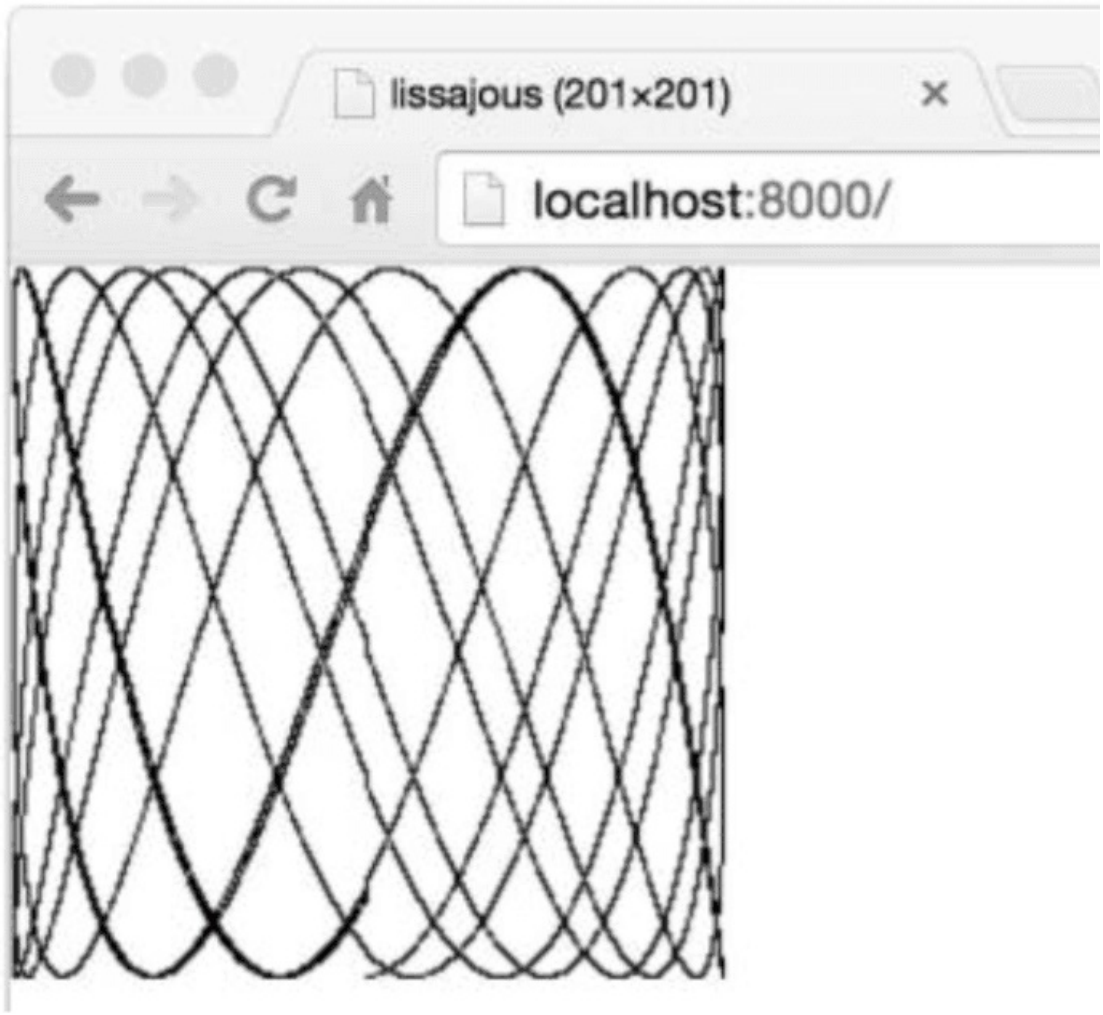


Figure 1.3. Animated Lissajous figures i

1.8. 本章要點

本章中對Go語言做了一些介紹，實際上Go語言還有很多方面在這有限的篇幅中還沒有覆蓋到。這里我們會把沒有講到的內容也做一些簡單的介紹，這樣讀者在之後看到完整的內容之前，也可以有個簡單印象。

控製流： 在本章我們隻介紹了if控製和for，但是沒有提到switch多路選擇。這里是一個簡單的switch的例子：

```
switch
 coinflip() {
case
 "heads"
:
    heads++
case
 "tails"
:
    tails++
default
:
    fmt.Println("landed on edge!"
)
}
```

在翻轉硬幣的時候，例子里的coinflip函數返迴幾種不同的結果，每一個case都會對應個返迴結果，這里需要註意，Go語言併不需要顯式地去在每一個case後寫break，語言默認執行完case後的邏輯語句會自動退出。當然了，如果你想要相鄰的幾個case都執行同一邏輯的話，需要自己顯式地寫上一個fallthrough語句來覆蓋這種默認行爲。不過fallthrough語句在一般的編程中用到得很少。

Go語言里的switch還可以不帶操作對象（譯註：switch不帶操作對象時默認用true值代替，然後將每個case的表達式和true值進行比較）；可以直接羅列多種條件，像其它語言里面的多個if else一樣，下面是一個例子：

```
func
 Signum(x int
) int
 {
    switch
 {
    case
 x > 0
:
        return
 +1

    default
:
        return
 0

    case
 x < 0
:
        return
 -1

    }
}
```

這種形式叫做無tag switch(tagless switch)；這和switch true是等價的。

像for和if控製語句一樣，switch也可以緊跟一個簡短的變量聲明，一個自增表達式、賦值語句，或者一個函數調用。

break和continue語句會改變控製流。和其它語言中的break和continue一樣，break會中斷當前的循環，併開始執行循環之後的內容，而continue會中跳過當前循環，併開始執行下一次循環。這兩個語句除了可以控製for循環，還可以用來控製switch和select語句(之後會講到)，在1.3節中我們看到，continue會跳過是內層的循環，如果我們想跳過的是更外層的循環的話，我們可以在相應的位置加上label，這樣break和continue就可以根據我們的想法來continue和break任意循環。這看起來甚至有點像goto語句的作用了。當然，一般程序員也不會用到這種操作。這兩種行爲更多地被用到機器生成的代碼中。

命名類型： 類型聲明使得我們可以很方便地給一個特殊類型一個名字。因爲struct類型聲明通常非常地長，所以我們總要給這種struct取一個名字。本章中就有這樣一個例子，二維點類型：

```
type
 Point struct
 {
    X, Y int

}
var
 p Point
```

類型聲明和命名類型會在第二章中介紹。

指針： Go語言提供了指針。指針是一種直接存儲了變量的內存地址的數據類型。在其它語言中，比如C語言，指針操作是完全不受約束的。在另外一些語言中，指針一般被處理爲“引用”，除了到處傳遞這些指針之外，併不能對這些指針做太多事情。Go語言在這兩種范圍中取了一種平衡。指針是可見的內存地址，&操作符可以返迴一個變量的內存地址，併且*操作符可以獲取指針指向的變量內容，但是在Go語言里沒有指針運算，也就是不能像c語言里可以對指針進行加或減操作。我們會在2.3.2中進行詳細介紹。

方法和接口： 方法是和命名類型關聯的一類函數。Go語言里比較特殊的是方法可以被關聯到任意一種命名類型。在第六章我們會詳細地講方法。接口是一種抽象類型，這種類型可以讓我們以同樣的方式來處理不同的固有類型，不用關心它們的具體實現，而隻需要關註它們提供的方法。第七章中會詳細説明這些內容。

包（packages）： Go語言提供了一些很好用的package，併且這些package是可以擴展的。Go語言社區已經創造併且分享了很多很多。所以Go語言編程大多數情況下就是用已有的package來寫我們自己的代碼。通過這本書，我們會講解一些重要的標準庫內的package，但是還是有很多我們沒有篇幅去説明，因爲我們沒法在這樣的厚度的書里去做一部代碼大全。

在你開始寫一個新程序之前，最好先去檢查一下是不是已經有了現成的庫可以幫助你更高效地完成這件事情。你可以在 https://golang.org/pkg 和 https://godoc.org 中找到標準庫和社區寫的package。godoc這個工具可以讓你直接在本地命令行閲讀標準庫的文檔。比如下面這個例子。

```
$ go doc http.ListenAndServe
package http // import "net/http"
func ListenAndServe(addr string, handler Handler) error
    ListenAndServe listens on the TCP network address addr and then
    calls Serve with handler to handle requests on incoming connections.
...
```

註釋： 我們之前已經提到過了在源文件的開頭寫的註釋是這個源文件的文檔。在每一個函數之前寫一個説明函數行爲的註釋也是一個好習慣。這些慣例很重要，因爲這些內容會被像godoc這樣的工具檢測到，併且在執行命令時顯示這些註釋。具體可以參考10.7.4。

多行註釋可以用 `/* ... */` 來包裹，和其它大多數語言一樣。在文件一開頭的註釋一般都是這種形式，或者一大段的解釋性的註釋文字也會被這符號包住，來避免每一行都需要加//。在註釋中//和/*是沒什麽意義的，所以不要在註釋中再嵌入註釋。

第2章 程序結構

Go語言和其他編程語言一樣，一個大的程序是由很多小的基礎構件組成的。變量保存值，簡單的加法和減法運算被組合成較複雜的表達式。基礎類型被聚合爲數組或結構體等更複雜的數據結構。然後使用if和for之類的控製語句來組織和控製表達式的執行流程。然後多個語句被組織到一個個函數中，以便代碼的隔離和複用。函數以源文件和包的方式被組織。

我們已經在前面章節的例子中看到了很多例子。在本章中，我們將深入討論Go程序基礎結構方面的一些細節。每個示例程序都是刻意寫的簡單，這樣我們可以減少複雜的算法或數據結構等不相關的問題帶來的榦擾，從而可以專註於Go語言本身的學習。

2.1. 命名

Go語言中的函數名、變量名、常量名、類型名、語句標號和包名等所有的命名，都遵循一個簡單的命名規則：一個名字必鬚以一個字母（Unicode字母）或下劃線開頭，後面可以跟任意數量的字母、數字或下劃線。大寫字母和小寫字母是不同的：heapSort和Heapsort是兩個不同的名字。

Go語言中類似if和switch的關鍵字有25個；關鍵字不能用於自定義名字，隻能在特定語法結構中使用。

```
break      default       func     interface   select
case       defer         go       map         struct
chan       else          goto     package     switch
const      fallthrough   if       range       type
continue   for           import   return      var
```

此外，還有大約30多個預定義的名字，比如int和true等，主要對應內建的常量、類型和函數。

```
內建常量: true false iota nil

內建類型: int int8 int16 int32 int64
          uint uint8 uint16 uint32 uint64 uintptr
          float32 float64 complex128 complex64
          bool byte rune string error

內建函數: make len cap new append copy close delete
          complex real imag
          panic recover
```

這些內部預先定義的名字併不是關鍵字，你可以再定義中重新使用它們。在一些特殊的場景中重新定義它們也是有意義的，但是也要註意避免過度而引起語義混亂。

如果一個名字是在函數內部定義，那麼它的就隻在函數內部有效。如果是在函數外部定義，那麼將在當前包的所有文件中都可以訪問。名字的開頭字母的大小寫決定了名字在包外的可見性。如果一個名字是大寫字母開頭的（譯註：必鬚是在函數外部定義的包級名字；包級函數名本身也是包級名字），那麼它將是導出的，也就是說可以被外部的包訪問，例如fmt包的Printf函數就是導出的，可以在fmt包外部訪問。包本身的名字一般總是用小寫字母。

名字的長度沒有邏輯限製，但是Go語言的風格是盡量使用短小的名字，對於局部變量尤其是這樣；你會經常看到i之類的短名字，而不是冗長的theLoopIndex命名。通常來說，如果一個名字的作用域比較大，生命週期也比較長，那麼用長的名字將會更有意義。

在習慣上，Go語言程序員推薦使用 **駝峯式** 命名，當名字有幾個單詞組成的時優先使用大小寫分隔，而不是優先用下劃線分隔。因此，在標準庫有QuoteRuneToASCII和parseRequestLine這樣的函數命名，但是一般不會用quote_rune_to_ASCII和parse_request_line這樣的命名。而像ASCII和HTML這樣的縮略詞則避免使用大小寫混合的寫法，它們可能被稱爲htmlEscape、HTMLEscape或escapeHTML，但不會是escapeHtml。

2.2. 聲明

聲明語句定義了程序的各種實體對象以及部分或全部的屬性。Go語言主要有四種類型的聲明語句：var、const、type和func，分別對應變量、常量、類型和函數實體對象的聲明。這一章我們重點討論變量和類型的聲明，第三章將討論常量的聲明，第五章將討論函數的聲明。

一個Go語言編寫的程序對應一個或多個以.go爲文件後綴名的源文件中。每個源文件以包的聲明語句開始，說明該源文件是屬於哪個包。包聲明語句之後是import語句導入依賴的其它包，然後是包一級的類型、變量、常量、函數的聲明語句，包一級的各種類型的聲明語句的順序無關緊要（譯註：函數內部的名字則必鬚先聲明之後才能使用）。例如，下面的例子中聲明了一個常量、一個函數和兩個變量：

```
gopl.io/ch2/boiling
// Boiling prints the boiling point of water.

package
 main

import
 "fmt"


const
 boilingF = 212.0


func
 main() {
    var
 f = boilingF
    var
 c = (f - 32
) * 5
 / 9

    fmt.Printf("boiling point = %g°F or %g°C\n"
, f, c)
    // Output:

    // boiling point = 212°F or 100°C

}
```

其中常量boilingF是在包一級范圍聲明語句聲明的，然後f和c兩個變量是在main函數內部聲明的聲明語句聲明的。在包一級聲明語句聲明的名字可在整個包對應的每個源文件中訪問，而不是僅僅在其聲明語句所在的源文件中訪問。相比之下，局部聲明的名字就隻能在函數內部很小的范圍被訪問。

一個函數的聲明由一個函數名字、參數列表（由函數的調用者提供參數變量的具體值）、一個可選的返迴值列表和包含函數定義的函數體組成。如果函數沒有返迴值，那麼返迴值列表是省略的。執行函數從函數的第一個語句開始，依次順序執行直到遇到renturn返迴語句，如果沒有返迴語句則是執行到函數末尾，然後返迴到函數調用者。

我們已經看到過很多函數聲明和函數調用的例子了，在第五章將深入討論函數的相關細節，這里隻簡單解釋下。下面的fToC函數封裝了溫度轉換的處理邏輯，這樣它隻需要被定義一次，就可以在多個地方多次被使用。在這個例子中，main函數就調用了兩次fToC函數，分別是使用在局部定義的兩個常量作爲調用函數的參數。

```
gopl.io/ch2/ftoc
// Ftoc prints two Fahrenheit-to-Celsius conversions.

package
 main

import
 "fmt"


func
 main() {
    const
 freezingF, boilingF = 32.0
, 212.0

    fmt.Printf("%g°F = %g°C\n"
, freezingF, fToC(freezingF)) // "32°F = 0°C"

    fmt.Printf("%g°F = %g°C\n"
, boilingF, fToC(boilingF))   // "212°F = 100°C"

}

func
 fToC(f float64
) float64
 {
    return
 (f - 32
) * 5
 / 9

}
```

2.3. 變量

var聲明語句可以創建一個特定類型的變量，然後給變量附加一個名字，併且設置變量的初始值。變量聲明的一般語法如下：

```
var
 變量名字 類型 = 表達式
```

其中“*類型*”或“= *表達式*”兩個部分可以省略其中的一個。如果省略的是類型信息，那麼將根據初始化表達式來推導變量的類型信息。如果初始化表達式被省略，那麼將用零值初始化該變量。 數值類型變量對應的零值是0，布爾類型變量對應的零值是false，字符串類型對應的零值是空字符串，接口或引用類型（包括slice、map、chan和函數）變量對應的零值是nil。數組或結構體等聚合類型對應的零值是每個元素或字段都是對應該類型的零值。

零值初始化機製可以確保每個聲明的變量總是有一個良好定義的值，因此在Go語言中不存在未初始化的變量。這個特性可以簡化很多代碼，而且可以在沒有增加額外工作的前提下確保邊界條件下的合理行爲。例如：

```
var
 s string

fmt.Println(s) // ""
```

這段代碼將打印一個空字符串，而不是導致錯誤或産生不可預知的行爲。Go語言程序員應該讓一些聚合類型的零值也具有意義，這樣可以保證不管任何類型的變量總是有一個合理有效的零值狀態。

也可以在一個聲明語句中同時聲明一組變量，或用一組初始化表達式聲明併初始化一組變量。如果省略每個變量的類型，將可以聲明多個類型不同的變量（類型由初始化表達式推導）：

```
var
 i, j, k int
                 // int, int, int

var
 b, f, s = true
, 2.3
, "four"
 // bool, float64, string
```

初始化表達式可以是字面量或任意的表達式。在包級別聲明的變量會在main入口函數執行前完成初始化（§2.6.2），局部變量將在聲明語句被執行到的時候完成初始化。

一組變量也可以通過調用一個函數，由函數返迴的多個返迴值初始化：

```
var
 f, err = os.Open(name) // os.Open returns a file and an error
```

2.3.1. 簡短變量聲明

在函數內部，有一種稱爲簡短變量聲明語句的形式可用於聲明和初始化局部變量。它以“名字 := 表達式”形式聲明變量，變量的類型根據表達式來自動推導。下面是lissajous函數中的三個簡短變量聲明語句（§1.4）：

```
anim := gif.GIF{LoopCount: nframes}
freq := rand.Float64() * 3.0

t := 0.0
```

因爲簡潔和靈活的特點，簡短變量聲明被廣泛用於大部分的局部變量的聲明和初始化。var形式的聲明語句往往是用於需要顯式指定變量類型地方，或者因爲變量稍後會被重新賦值而初始值無關緊要的地方。

```
i := 100
                  // an int

var
 boiling float64
 = 100
 // a float64

var
 names []string

var
 err error
var
 p Point
```

和var形式聲明變語句一樣，簡短變量聲明語句也可以用來聲明和初始化一組變量：

```
i, j := 0
, 1
```

但是這種同時聲明多個變量的方式應該限製隻在可以提高代碼可讀性的地方使用，比如for語句的循環的初始化語句部分。

請記住“:=”是一個變量聲明語句，而“=‘是一個變量賦值操作。也不要混淆多個變量的聲明和元組的多重賦值（§2.4.1），後者是將右邊各個的表達式值賦值給左邊對應位置的各個變量：

```
i, j = j, i // 交換 i 和 j 的值
```

和普通var形式的變量聲明語句一樣，簡短變量聲明語句也可以用函數的返迴值來聲明和初始化變量，像下面的os.Open函數調用將返迴兩個值：

```
f, err := os.Open(name)
if
 err != nil
 {
    return
 err
}
// ...use f...

f.Close()
```

這里有一個比較微妙的地方：簡短變量聲明左邊的變量可能併不是全部都是剛剛聲明的。如果有一些已經在相同的詞法域聲明過了（§2.7），那麼簡短變量聲明語句對這些已經聲明過的變量就隻有賦值行爲了。

在下面的代碼中，第一個語句聲明了in和err兩個變量。在第二個語句隻聲明了out一個變量，然後對已經聲明的err進行了賦值操作。

```
in, err := os.Open(infile)
// ...

out, err := os.Create(outfile)
```

簡短變量聲明語句中必鬚至少要聲明一個新的變量，下面的代碼將不能編譯通過：

```
f, err := os.Open(infile)
// ...

f, err := os.Create(outfile) // compile error: no new variables
```

解決的方法是第二個簡短變量聲明語句改用普通的多重賦值語言。

簡短變量聲明語句隻有對已經在同級詞法域聲明過的變量才和賦值操作語句等價，如果變量是在外部詞法域聲明的，那麼簡短變量聲明語句將會在當前詞法域重新聲明一個新的變量。我們在本章後面將會看到類似的例子。

2.3.2. 指針

一個變量對應一個保存了變量對應類型值的內存空間。普通變量在聲明語句創建時被綁定到一個變量名，比如叫x的變量，但是還有很多變量始終以表達式方式引入，例如x[i]或x.f變量。所有這些表達式一般都是讀取一個變量的值，除非它們是出現在賦值語句的左邊，這種時候是給對應變量賦予一個新的值。

一個指針的值是另一個變量的地址。一個指針對應變量在內存中的存儲位置。併不是每一個值都會有一個內存地址，但是對於每一個變量必然有對應的內存地址。通過指針，我們可以直接讀或更新對應變量的值，而不需要知道該變量的名字（如果變量有名字的話）。

如果用“var x int”聲明語句聲明一個x變量，那麼&x表達式（取x變量的內存地址）將產生一個指向該整數變量的指針，指針對應的數據類型是 `*int` ，指針被稱之爲“指向int類型的指針”。如果指針名字爲p，那麼可以說“p指針指向變量x”，或者說“p指針保存了x變量的內存地址”。同時 `*p` 表達式對應p指針指向的變量的值。一般 `*p` 表達式讀取指針指向的變量的值，這里爲int類型的值，同時因爲 `*p` 對應一個變量，所以該表達式也可以出現在賦值語句的左邊，表示更新指針所指向的變量的值。

```
x := 1

p := &x         // p, of type *int, points to x

fmt.Println(*p) // "1"

*p = 2
          // equivalent to x = 2

fmt.Println(x)  // "2"
```

對於聚合類型每個成員——比如結構體的每個字段、或者是數組的每個元素——也都是對應一個變量，因此可以被取地址。

變量有時候被稱爲可尋址的值。即使變量由表達式臨時生成，那麼表達式也必鬚能接受 `&` 取地址操作。

任何類型的指針的零值都是nil。如果 `p != nil` 測試爲眞，那麼p是指向某個有效變量。指針之間也是可以進行相等測試的，隻有當它們指向同一個變量或全部是nil時才相等。

```
var
 x, y int

fmt.Println(&x == &x, &x == &y, &x == nil
) // "true false false"
```

在Go語言中，返迴函數中局部變量的地址也是安全的。例如下面的代碼，調用f函數時創建局部變量v，在局部變量地址被返迴之後依然有效，因爲指針p依然引用這個變量。

```
var
 p = f()

func
 f() *int
 {
    v := 1

    return
 &v
}
```

每次調用函數都將返回不同的結果：

```
fmt.Println(f() == f()) // "false"
```

因爲指針包含了一個變量的地址，因此如果將指針作爲參數調用函數，那將可以在函數中通過該指針來更新變量的值。例如下面這個例子就是通過指針來更新變量的值，然後返回更新後的值，可用在一個表達式中（譯註：這是對C語言中 `++v` 操作的模擬，這里隻是爲了説明指針的用法，incr函數模擬的做法併不推薦）：

```
func
 incr(p *int
) int
 {
    *p++ // 非常重要：隻是增加p指向的變量的值，併不改變p指針！！！

    return
 *p
}

v := 1

incr(&v)              // side effect: v is now 2

fmt.Println(incr(&v)) // "3" (and v is 3)
```

每次我們對一個變量取地址，或者複製指針，我們都是爲原變量創建了新的別名。例如， `*p` 就是是 變量v的別名。指針特別有價值的地方在於我們可以不用名字而訪問一個變量，但是這是一把雙刃劍：要找到一個變量的所有訪問者併不容易，我們必鬚知道變量全部的別名（譯註：這是Go語言的垃圾迴收器所做的工作）。不僅僅是指針會創建別名，很多其他引用類型也會創建別名，例如slice、map和chan，甚至結構體、數組和接口都會創建所引用變量的別名。

指針是實現標準庫中flag包的關鍵技術，它使用命令行參數來設置對應變量的值，而這些對應命令行標誌參數的變量可能會零散分布在整個程序中。爲了説明這一點，在早些的echo版本中，就包含了兩個可選的命令行參數： `-n` 用於忽略行尾的換行符， `-s sep` 用於指定分隔字符（默認是空格）。下面這是第四個版本，對應包路徑爲gopl.io/ch2/echo4。

```
gopl.io/ch2/echo4
// Echo4 prints its command-line arguments.

package
 main

import
 (
    "flag"

    "fmt"

    "strings"

)

var
 n = flag.Bool("n"
, false
, "omit trailing newline"
)
var
 sep = flag.String("s"
, " "
, "separator"
)

func
 main() {
    flag.Parse()
    fmt.Print(strings.Join(flag.Args(), *sep))
    if
 !*n {
        fmt.Println()
    }
}
```

調用flag.Bool函數會創建一個新的對應布爾型標誌參數的變量。它有三個屬性：第一個是的命令行標誌參數的名字“n”，然後是該標誌參數的默認值（這里是false），最後是該標誌參數對應的描述信息。如果用戶在命令行輸入了一個無效的標誌參數，或者輸入 `-h` 或 `-help` 參數，那麼將打印所有標誌參數的名字、默認值和描述信息。類似的，調用flag.String函數將於創建一個對應字符串類型的標誌參數變量，同樣包含命令行標誌參數對應的參數名、默認值、和描述信息。程序中的 `sep` 和 `n` 變量分別是指向對應命令行標誌參數變量的指針，因此必鬚用 `*sep` 和 `*n` 形式的指針語法間接引用它們。

當程序運行時，必鬚在使用標誌參數對應的變量之前調用先flag.Parse函數，用於更新每個標誌參數對應變量的值（之前是默認值）。對於非標誌參數的普通命令行參數可以通過調用flag.Args()函數來訪問，返回值對應對應一個字符串類型的slice。如果在flag.Parse函數解析命令行參數時遇到錯誤，默認將打印相關的提示信息，然後調用os.Exit(2)終止程序。

讓我們運行一些echo測試用例：

```
$ go build gopl.io/ch2/echo4
$ ./echo4 a bc def
a bc def
$ ./echo4 -s / a bc def
a/bc/def
$ ./echo4 -n a bc def
a bc def$
$ ./echo4 -help
Usage of ./echo4:
  -n    omit trailing newline
  -s string
        separator (default " ")
```

2.3.3. new函數

另一個創建變量的方法是調用用內建的new函數。表達式new(T)將創建一個T類型的匿名變量，初始化爲T類型的零值，然後返回變量地址，返回的指針類型爲 `*T` 。

```
p := new
(int
)   // p, *int 類型, 指向匿名的 int 變量

fmt.Println(*p) // "0"

*p = 2
          // 設置 int 匿名變量的值爲 2

fmt.Println(*p) // "2"
```

用new創建變量和普通變量聲明語句方式創建變量沒有什麼區別，除了不需要聲明一個臨時變量的名字外，我們還可以在表達式中使用new(T)。換言之，new函數類似是一種語法醣，而不是一個新的基礎概念。

下面的兩個newInt函數有着相同的行爲：

```
func
 newInt() *int
 {               func
 newInt() *int
 {
    return
 new
(int
)                    var
 dummy int

}                                    return
 &dummy
                                  }
```

每次調用new函數都是返迴一個新的變量的地址，因此下面兩個地址是不同的：

```
p := new
(int
)
q := new
(int
)
fmt.Println(p == q) // "false"
```

當然也可能有特殊情況：如果兩個類型都是空的，也就是說類型的大小是0，例如 `struct{}` 和 `[0]int` ，有可能有相同的地址（依賴具體的語言實現）（譯註：請謹慎使用大小爲0的類型，因爲如果類型的大小位0好話，可能導致Go語言的自動垃圾迴收器有不同的行爲，具體請查看 `runtime.SetFinalizer` 函數相關文檔）。

new函數使用常見相對比較少，因爲對應結構體來說，可以直接用字面量語法創建新變量的方法會更靈活（§4.4.1）。

由於new隻是一個預定義的函數，它併不是一個關鍵字，因此我們可以將new名字重新定義爲別的類型。例如下面的例子：

```
func
 delta(old, new
 int
) int
 { return
 new
 - old }
```

由於new被定義爲int類型的變量名，因此在delta函數內部是無法使用內置的new函數的。

2.3.4. 變量的生命週期

變量的生命週期指的是在程序運行期間變量有效存在的時間間隔。對於在包一級聲明的變量來說，它們的生命週期和整個程序

的運行週期是一致的。而相比之下，在局部變量的聲明週期則是動態的：從每次創建一個新變量的聲明語句開始，直到該變量不再被引用爲止，然後變量的存儲空間可能被迴收。函數的參數變量和返迴值變量都是局部變量。它們在函數每次被調用的時候創建。

例如，下面是從1.4節的Lissajous程序摘録的代碼片段：

```
for
 t := 0.0
; t < cycles*2
*math.Pi; t += res {
    x := math.Sin(t)
    y := math.Sin(t*freq + phase)
    img.SetColorIndex(size+int
(x*size+0.5
), size+int
(y*size+0.5
),
        blackIndex)
}
```

譯註：函數的有右小括弧也可以另起一行縮進，同時爲了防止編譯器在行尾自動插入分號而導致的編譯錯誤，可以在末尾的參數變量後面顯式插入逗號。像下面這樣：

```
for
 t := 0.0
; t < cycles*2
*math.Pi; t += res {
    x := math.Sin(t)
    y := math.Sin(t*freq + phase)
    img.SetColorIndex(
        size+int
(x*size+0.5
), size+int
(y*size+0.5
),
        blackIndex, // 最後插入的逗號不會導致編譯錯誤，這是Go編譯器的一個特性

    )               // 小括弧另起一行縮進，和大括弧的風格保存一致

}
```

在每次循環的開始會創建臨時變量t，然後在每次循環迭代中創建臨時變量x和y。

那麼垃Go語言的自動圾收集器是如何知道一個變量是何時可以被迴收的呢？這里我們可以避開完整的技術細節，基本的實現思路是，從每個包級的變量和每個當前運行函數的每一個局部變量開始，通過指針或引用的訪問路徑遍歷，是否可以找到該變量。如果不存在這樣的訪問路徑，那麼説明該變量是不可達的，也就是説它是否存在併不會影響程序後續的計算結果。

因爲一個變量的有效週期隻取決於是否可達，因此一個循環迭代內部的局部變量的生命週期可能超出其局部作用域。同時，局部變量可能在函數返迴之後依然存在。

編譯器會自動選擇在棧上還是在堆上分配局部變量的存儲空間，但可能令人驚訝的是，這個選擇併不是由用var還是new聲明變量的方式決定的。

```
var
 global *int


func
 f() {                 func
 g() {
    var
 x int
                  y := new
(int
)
    x = 1
                      *y = 1

    global = &x           }
}
```

這里的x變量必鬚在堆上分配，因爲它在函數退出後依然可以通過包一級的global變量找到，雖然它是在函數內部定義的；用Go語言的術語說，這個x局部變量從函數f中逃逸了。相反，當g函數返迴時，變量 `*y` 將是不可達的，也就是說可以馬上被迴收的。因此， `*y` 併沒有從函數g中逃逸，編譯器可以選擇在棧上分配 `*y` 的存儲空間（譯註：也可以選擇在堆上分配，然後由Go語言的GC迴收這個變量的內存空間），雖然這里用的是new方式。其實在任何時候，你併不需爲了編寫正確的代碼而要考慮變量的逃逸行爲，要記住的是，逃逸的變量需要額外分配內存，同時對性能的優化可能會産生細微的影響。

Go語言的自動垃圾收集器對編寫正確的代碼是一個鉅大的幫助，但也併不是說你完全不用考慮內存了。你雖然不需要顯式地分配和釋放內存，但是要編寫高效的程序你依然需要了解變量的生命週期。例如，如果將指向短生命週期對象的指針保存到具有長生命週期的對象中，特別是保存到全局變量時，會阻止對短生命週期對象的垃圾迴收（從而可能影響程序的性能）。

2.4. 賦值

使用賦值語句可以更新一個變量的值，最簡單的賦值語句是將要被賦值的變量放在=的左邊，新值的表達式放在=的右邊。

```
x = 1
                        // 命令變量的賦值

*p = true
                    // 通過指針間接賦值

person.name = "bob"
         // 結構體字段賦值

count[x] = count[x] * scale // 數組、slice或map的元素賦值
```

特定的二元算術運算符和賦值語句的複合操作有一個簡潔形式，例如上面最後的語句可以重寫爲：

```
count[x] *= scale
```

這樣可以省去對變量表達式的重複計算。

數值變量也可以支持 `++` 遞增和 `--` 遞減語句（譯註：自增和自減是語句，而不是表達式，因此 `x = i++` 之類的表達式是錯誤的）：

```
v := 1

v++    // 等價方式 v = v + 1; v 變成 2

v--    // 等價方式 v = v - 1; v 變成 1
```

2.4.1. 元組賦值

元組賦值是另一種形式的賦值語句，它允許同時更新多個變量的值。在賦值之前，賦值語句右邊的所有表達式將會先進行求值，然後再統一更新左邊對應變量的值。這對於處理有些同時出現在元組賦值語句左右兩邊的變量很有幫助，例如我們可以這樣交換兩個變量的值：

```
x, y = y, x

a[i], a[j] = a[j], a[i]
```

或者是計算兩個整數值的的最大公約數（GCD）（譯註：GCD不是那個敏感字，而是greatest common divisor的縮寫，歐幾里德的GCD是最早的非平凡算法）：

```
func
 gcd(x, y int
) int
 {
    for
 y != 0
 {
        x, y = y, x%y
    }
    return
 x
}
```

或者是計算斐波納契數列（Fibonacci）的第N個數：

```
func
 fib(n int
) int
 {
    x, y := 0
, 1

    for
 i := 0
; i < n; i++ {
        x, y = y, x+y
    }
    return
 x
}
```

元組賦值也可以使一繫列瑣碎賦值更加緊湊（譯註: 特別是在for循環的初始化部分），

```
i, j, k = 2
, 3
, 5
```

但如果表達式太複雜的話，應該盡量避免過度使用元組賦值；因爲每個變量單獨賦值語句的寫法可讀性會更好。

有些表達式會産生多個值，比如調用一個有多個返迴值的函數。當這樣一個函數調用出現在元組賦值右邊的表達式中時（譯註：右邊不能再有其它表達式），左邊變量的數目必鬚和右邊一致。

```
f, err = os.Open("foo.txt"
) // function call returns two values
```

通常，這類函數會用額外的返迴值來表達某種錯誤類型，例如os.Open是用額外的返迴值返迴一個error類型的錯誤，還有一些是用來返迴布爾值，通常被稱爲ok。在稍後我們將看到的三個操作都是類似的用法。如果map查找（§4.3）、類型斷言（§7.10）或通道接收（§8.4.2）出現在賦值語句的右邊，它們都可能會產生兩個結果，有一個額外的布爾結果表示操作是否成功：

```
v, ok = m[key]             // map lookup

v, ok = x.(T)              // type assertion

v, ok = <-ch               // channel receive
```

譯註：map查找（§4.3）、類型斷言（§7.10）或通道接收（§8.4.2）出現在賦值語句的右邊時，併不一定是產生兩個結果，也可能隻產生一個結果。對於值產生一個結果的情形，map查找失敗時會返迴零值，類型斷言失敗時會發送運行時panic異常，通道接收失敗時會返迴零值（阻塞不算是失敗）。例如下面的例子：

```
v = m[key]                // map查找，失敗時返迴零值

v = x.(T)                 // type斷言，失敗時panic異常

v = <-ch                  // 管道接收，失敗時返迴零值（阻塞不算是失敗）


_, ok = m[key]            // map返迴2個值

_, ok = mm[""
], false
     // map返迴1個值

_ = mm[""
]                // map返迴1個值
```

和變量聲明一樣，我們可以用下劃線空白標識符 `_` 來丟棄不需要的值。

```
_, err = io.Copy(dst, src) // 丟棄字節數

_, ok = x.(T)              // 隻檢測類型，忽略具體值
```

2.4.2. 可賦值性

賦值語句是顯式的賦值形式，但是程序中還有很多地方會發生隱式的賦值行爲：函數調用會隱式地將調用參數的值賦值給函數的參數變量，一個返迴語句將隱式地將返迴操作的值賦值給結果變量，一個複合類型的字面量（§4.2）也會産生賦值行爲。例如下面的語句：

```
medals := []string
{"gold"
, "silver"
, "bronze"
}
```

隱式地對slice的每個元素進行賦值操作，類似這樣寫的行爲：

```
medals[0
] = "gold"

medals[1
] = "silver"

medals[2
] = "bronze"
```

map和chan的元素，雖然不是普通的變量，但是也有類似的隱式賦值行爲。

不管是隱式還是顯式地賦值，在賦值語句左邊的變量和右邊最終的求到的值必鬚有相同的數據類型。更直白地說，隻有右邊的值對於左邊的變量是可賦值的，賦值語句才是允許的。

可賦值性的規則對於不同類型有着不同要求，對每個新類型特殊的地方我們會專門解釋。對於目前我們已經討論過的類型，它的規則是簡單的：類型必鬚完全匹配，nil可以賦值給任何指針或引用類型的變量。常量（§3.6）則有更靈活的賦值規則，因爲這樣可以避免不必要的顯式的類型轉換。

對於兩個值是否可以用 `==` 或 `!=` 進行相等比較的能力也和可賦值能力有關繫：對於任何類型的值的相等比較，第二個值必鬚是對第一個值類型對應的變量是可賦值的，反之依然。和前面一樣，我們會對每個新類型比較特殊的地方做專門的解釋。

2.5. 類型

變量或表達式的類型定義了對應存儲值的屬性特徵，例如數值在內存的存儲大小（或者是元素的bit個數），它們在內部是如何表達的，是否支持一些操作符，以及它們自己關聯的方法集等。

在任何程序中都會存在一些變量有着相同的內部結構，但是卻表示完全不同的概念。例如，一個int類型的變量可以用來表示一個循環的迭代索引、或者一個時間戳、或者一個文件描述符、或者一個月份；一個float64類型的變量可以用來表示每秒移動幾米的速度、或者是不同溫度單位下的溫度；一個字符串可以用來表示一個密碼或者一個顏色的名稱。

一個類型聲明語句創建了一個新的類型名稱，和現有類型具有相同的底層結構。新命名的類型提供了一個方法，用來分隔不同概念的類型，這樣即使它們底層類型相同也是不兼容的。

```
type
 類型名字 底層類型
```

類型聲明語句一般出現在包一級，因此如果新創建的類型名字的首字符大寫，則在外部包也可以使用。

譯註：對於中文漢字，Unicode標誌都作爲小寫字母處理，因此中文的命名默認不能導出；不過国內的用戶針對該問題提出了不同的看法，根據RobPike的迴複，在Go2中有可能會將中日韓等字符當作大寫字母處理。下面是RobPik在 Issue763 的迴複：

> A solution that's been kicking around for a while:
>
> For Go 2 (can't do it before then): Change the definition to “lower case letters and *are package-local; all else is exported”. Then with non-cased languages, such as Japanese, we can write 日本語 for an exported name and* 日本語 for a local name. This rule has no effect, relative to the Go 1 rule, with cased languages. They behave exactly the same.

爲了說明類型聲明，我們將不同溫度單位分別定義爲不同的類型：

```
gopl.io/ch2/tempconv0
// Package tempconv performs Celsius and Fahrenheit temperature computations.

package
 tempconv

import
 "fmt"


type
 Celsius float64
    // 攝氏溫度

type
 Fahrenheit float64
 // 華氏溫度


const
 (
    AbsoluteZeroC Celsius = -273.15
 // 絶對零度

    FreezingC     Celsius = 0
       // 結冰點溫度

    BoilingC      Celsius = 100
     // 沸水溫度

)

func
 CToF(c Celsius) Fahrenheit { return
 Fahrenheit(c*9
/5
 + 32
) }

func
 FToC(f Fahrenheit) Celsius { return
 Celsius((f - 32
) * 5
 / 9
) }
```

我們在這個包聲明了兩種類型：Celsius和Fahrenheit分別對應不同的溫度單位。它們雖然有着相同的底層類型float64，但是它們是不同的數據類型，因此它們不可以被相互比較或混在一個表達式運算。刻意區分類型，可以避免一些像無意中使用不同單位的溫度混合計算導致的錯誤；因此需要一個類似Celsius(t)或Fahrenheit(t)形式的顯式轉型操作才能將float64轉爲對應的類型。

Celsius(t)和Fahrenheit(t)是類型轉換操作，它們併不是函數調用。類型轉換不會改變值本身，但是會使它們的語義發生變化。另一方面，CToF和FToC兩個函數則是對不同溫度單位下的溫度進行換算，它們會返迴不同的值。

對於每一個類型T，都有一個對應的類型轉換操作T(x)，用於將x轉爲T類型（譯註：如果T是指針類型，可能會需要用小括弧包裝T，比如 `(*int)(0)` ）。隻有當兩個類型的底層基礎類型相同時，才允許這種轉型操作，或者是兩者都是指向相同底層結構的指針類型，這些轉換隻改變類型而不會影響值本身。如果x是可以賦值給T類型的值，那麼x必然也可以被轉爲T類型，但是一般沒有這個必要。

數值類型之間的轉型也是允許的，併且在字符串和一些特定類型的slice之間也是可以轉換的，在下一章我們會看到這樣的例子。這類轉換可能改變值的表現。例如，將一個浮點數轉爲整數將丟棄小數部分，將一個字符串轉爲 `[]byte` 類型的slice將拷貝一個字符串數據的副本。在任何情況下，運行時不會發生轉換失敗的錯誤（譯註:錯誤隻會發生在編譯階段）。

底層數據類型決定了內部結構和表達方式，也決定是否可以像底層類型一樣對內置運算符的支持。這意味着，Celsius和Fahrenheit類型的算術運算行爲和底層的float64類型是一樣的，正如我們所期望的那樣。

```
fmt.Printf("%g\n"
, BoilingC-FreezingC) // "100" °C

boilingF := CToF(BoilingC)
fmt.Printf("%g\n"
, boilingF-CToF(FreezingC)) // "180" °F

fmt.Printf("%g\n"
, boilingF-FreezingC)       // compile error: type mismatch
```

比較運算符 `==` 和 `<` 也可以用來比較一個命名類型的變量和另一個有相同類型的變量，或有着相同底層類型的未命名類型的值之間做比較。但是如果兩個值有着不同的類型，則不能直接進行比較：

```
var
 c Celsius
var
 f Fahrenheit
fmt.Println(c == 0
)          // "true"

fmt.Println(f >= 0
)          // "true"

fmt.Println(c == f)          // compile error: type mismatch

fmt.Println(c == Celsius(f)) // "true"!
```

註意最後那個語句。盡管看起來想函數調用，但是Celsius(f)是類型轉換操作，它併不會改變值，僅僅是改變值的類型而已。測試爲眞的原因是因爲c和g都是零值。

一個命名的類型可以提供書寫方便，特別是可以避免一遍又一遍地書寫複雜類型（譯註：例如用匿名的結構體定義變量）。雖然對於像float64這種簡單的底層類型沒有簡潔很多，但是如果是複雜的類型將會簡潔很多，特別是我們即將討論的結構體類型。

命名類型還可以爲該類型的值定義新的行爲。這些行爲表示爲一組關聯到該類型的函數集合，我們稱爲類型的方法集。我們將在第六章中討論方法的細節，這里值説寫簡單用法。

下面的聲明語句，Celsius類型的參數c出現在了函數名的前面，表示聲明的是Celsius類型的一個叫名叫String的方法，該方法返迴該類型對象c帶着°C溫度單位的字符串：

```
func
 (c Celsius) String() string
 { return
 fmt.Sprintf("%g°C"
, c) }
```

許多類型都會定義一個String方法，因爲當使用fmt包的打印方法時，將會優先使用該類型對應的String方法返迴的結果打印，我們將在7.1節講述。

```
c := FToC(212.0
)
fmt.Println(c.String()) // "100°C"

fmt.Printf("%v\n"
, c)   // "100°C"; no need to call String explicitly

fmt.Printf("%s\n"
, c)   // "100°C"

fmt.Println(c)          // "100°C"

fmt.Printf("%g\n"
, c)   // "100"; does not call String

fmt.Println(float64
(c)) // "100"; does not call String
```

2.6. 包和文件

Go語言中的包和其他語言的庫或模塊的概念類似，目的都是爲了支持模塊化、封裝、單獨編譯和代碼重用。一個包的源代碼保存在一個或多個以.go爲文件後綴名的源文件中，通常一個包所在目録路徑的後綴是包的導入路徑；例如包gopl.io/ch1/helloworld對應的目録路徑是$GOPATH/src/gopl.io/ch1/helloworld。

每個包都對應一個獨立的名字空間。例如，在image包中的Decode函數和在unicode/utf16包中的 Decode函數是不同的。要在外部引用該函數，必鬚顯式使用image.Decode或utf16.Decode形式訪問。

包還可以讓我們通過控製哪些名字是外部可見的來隱藏內部實現信息。在Go語言中，一個簡單的規則是：如果一個名字是大寫字母開頭的，那麼該名字是導出的（譯註：因爲漢字不區分大小寫，因此漢字開頭的名字是沒有導出的）。

爲了演示包基本的用法，先假設我們的溫度轉換軟件已經很流行，我們希望到Go語言社區也能使用這個包。我們該如何做呢？

讓我們創建一個名爲gopl.io/ch2/tempconv的包，這是前面例子的一個改進版本。（我們約定我們的例子都是以章節順序來編號的，這樣的路徑更容易閱讀）包代碼存儲在兩個源文件中，用來演示如何在一個源文件聲明然後在其他的源文件訪問；雖然在現實中，這樣小的包一般隻需要一個文件。

我們把變量的聲明、對應的常量，還有方法都放到tempconv.go源文件中：

```
gopl.io/ch2/tempconv
// Package tempconv performs Celsius and Fahrenheit conversions.

package
 tempconv

import
 "fmt"


type
 Celsius float64

type
 Fahrenheit float64


const
 (
    AbsoluteZeroC Celsius = -273.15

    FreezingC     Celsius = 0

    BoilingC      Celsius = 100

)

func
 (c Celsius) String() string
    { return
 fmt.Sprintf("%g°C"
, c) }
func
 (f Fahrenheit) String() string
 { return
 fmt.Sprintf("%g°F"
, f) }
```

轉換函數則放在另一個conv.go源文件中：

```
package
 tempconv

// CToF converts a Celsius temperature to Fahrenheit.

func
 CToF(c Celsius) Fahrenheit { return
 Fahrenheit(c*9
/5
 + 32
) }

// FToC converts a Fahrenheit temperature to Celsius.

func
 FToC(f Fahrenheit) Celsius { return
 Celsius((f - 32
) * 5
 / 9
) }
```

每個源文件都是以包的聲明語句開始，用來指名包的名字。當包被導入的時候，包內的成員將通過類似tempconv.CToF的形式訪問。而包級別的名字，例如在一個文件聲明的類型和常量，在同一個包的其他源文件也是可以直接訪問的，就好像所有代碼都在一個文件一樣。要註意的是tempconv.go源文件導入了fmt包，但是conv.go源文件併沒有，因爲這個源文件中的代碼併沒有用到fmt包。

因爲包級別的常量名都是以大寫字母開頭，它們可以像tempconv.AbsoluteZeroC這樣被外部代碼訪問：

```
fmt.Printf("Brrrr! %v\n"
, tempconv.AbsoluteZeroC) // "Brrrr! -273.15°C"
```

要將攝氏溫度轉換爲華氏溫度，需要先用import語句導入gopl.io/ch2/tempconv包，然後就可以使用下面的代碼進行轉換了：

```
fmt.Println(tempconv.CToF(tempconv.BoilingC)) // "212°F"
```

在每個源文件的包聲明前僅跟着的註釋是包註釋（§10.7.4）。通常，包註釋的第一句應該先是包的功能概要說明。一個包通常隻有一個源文件有包註釋（譯註：如果有多個包註釋，目前的文檔工具會根據源文件名的先後順序將它們鏈接爲一個包註釋）。如果包註釋很大，通常會放到一個獨立的doc.go文件中。

練習 2.1： 向tempconv包添加類型、常量和函數用來處理Kelvin絕對溫度的轉換，Kelvin 絕對零度是−273.15°C，Kelvin絕對溫度1K和攝氏度1°C的單位間隔是一樣的。

2.6.1. 導入包

在Go語言程序中，每個包都是有一個全局唯一的導入路徑。導入語句中類似"gopl.io/ch2/tempconv"的字符串對應包的導入路徑。Go語言的規范併沒有定義這些字符串的具體含義或包來自哪里，它們是由構建工具來解釋的。當使用Go語言自帶的go工具箱

時（第十章），一個導入路徑代表一個目錄中的一個或多個Go源文件。

除了包的導入路徑，每個包還有一個包名，包名一般是短小的名字（併不要求包名是唯一的），包名在包的聲明處指定。按照慣例，一個包的名字和包的導入路徑的最後一個字段相同，例如gopl.io/ch2/tempconv包的名字一般是tempconv。

要使用gopl.io/ch2/tempconv包，需要先導入：

```
gopl.io/ch2/cf
// Cf converts its numeric argument to Celsius and Fahrenheit.

package
 main

import
 (
    "fmt"

    "os"

    "strconv"


    "gopl.io/ch2/tempconv"

)

func
 main() {
    for
 _, arg := range
 os.Args[1
:] {
        t, err := strconv.ParseFloat(arg, 64
)
        if
 err != nil
 {
            fmt.Fprintf(os.Stderr, "cf: %v\n"
, err)
            os.Exit(1
)
        }
        f := tempconv.Fahrenheit(t)
        c := tempconv.Celsius(t)
        fmt.Printf("%s = %s, %s = %s\n"
,
            f, tempconv.FToC(f), c, tempconv.CToF(c))
    }
}
```

導入語句將導入的包綁定到一個短小的名字，然後通過該短小的名字就可以引用包中導出的全部內容。上面的導入聲明將允許我們以tempconv.CToF的形式來訪問gopl.io/ch2/tempconv包中的內容。在默認情況下，導入的包綁定到tempconv名字（譯註：這包聲明語句指定的名字），但是我們也可以綁定到另一個名稱，以避免名字衝突（§10.4）。

cf程序將命令行輸入的一個溫度在Celsius和Fahrenheit溫度單位之間轉換：

```
$ go build gopl.io/ch2/cf
$ ./cf 32
32°F = 0°C, 32°C = 89.6°F
$ ./cf 212
212°F = 100°C, 212°C = 413.6°F
$ ./cf -40
-40°F = -40°C, -40°C = -40°F
```

如果導入了一個包，但是又沒有使用該包將被當作一個編譯錯誤處理。這種強製規則可以有效減少不必要的依賴，雖然在調試期間可能會讓人討厭，因爲刪除一個類似log.Print("got here!")的打印語句可能導致需要同時刪除log包導入聲明，否則，編譯器將會發出一個錯誤。在這種情況下，我們需要將不必要的導入刪除或註釋掉。

不過有更好的解決方案，我們可以使用golang.org/x/tools/cmd/goimports導入工具，它可以根據需要自動添加或刪除導入的包；許多編輯器都可以集成goimports工具，然後在保存文件的時候自動運行。類似的還有gofmt工具，可以用來格式化Go源文件。

練習 2.2： 寫一個通用的單位轉換程序，用類似cf程序的方式從命令行讀取參數，如果缺省的話則是從標準輸入讀取參數，然後做類似Celsius和Fahrenheit的單位轉換，長度單位可以對應英尺和米，重量單位可以對應磅和公斤等。

2.6.2. 包的初始化

包的初始化首先是解決包級變量的依賴順序，然後安照包級變量聲明出現的順序依次初始化：

```
var
 a = b + c // a 第三個初始化, 爲 3

var
 b = f()   // b 第二個初始化, 爲 2, 通過調用 f (依賴c)

var
 c = 1
     // c 第一個初始化, 爲 1


func
 f() int
 { return
 c + 1
 }
```

如果包中含有多個.go源文件，它們將按照發給編譯器的順序進行初始化，Go語言的構建工具首先會將.go文件根據文件名排序，然後依次調用編譯器編譯。

對於在包級別聲明的變量，如果有初始化表達式則用表達式初始化，還有一些沒有初始化表達式的，例如某些表格數據初始化併不是一個簡單的賦值過程。在這種情況下，我們可以用一個特殊的init初始化函數來簡化初始化工作。每個文件都可以包含多個init初始化函數

```
func
 init() { /* ... */
 }
```

這樣的init初始化函數除了不能被調用或引用外，其他行爲和普通函數類似。在每個文件中的init初始化函數，在程序開始執行時按照它們聲明的順序被自動調用。

每個包在解決依賴的前提下，以導入聲明的順序初始化，每個包隻會被初始化一次。因此，如果一個p包導入了q包，那麼在p包初始化的時候可以認爲q包必然已經初始化過了。初始化工作是自下而上進行的，main包最後被初始化。以這種方式，可以確保在main函數執行之前，所有依然的包都已經完成初始化工作了。

下面的代碼定義了一個PopCount函數，用於返迴一個數字中含二進製1bit的個數。它使用init初始化函數來生成輔助表格pc，pc表格用於處理每個8bit寬度的數字含二進製的1bit的bit個數，這樣的話在處理64bit寬度的數字時就沒有必要循環64次，隻需要8次查表就可以了。（這併不是最快的統計1bit數目的算法，但是它可以方便演示init函數的用法，併且演示了如果預生成輔助表格，這是編程中常用的技術）。

```
gopl.io/ch2/popcount
package
 popcount

// pc[i] is the population count of i.

var
 pc [256
]byte


func
 init() {
    for
 i := range
 pc {
        pc[i] = pc[i/2
] + byte
(i&1
)
    }
}

// PopCount returns the population count (number of set bits) of x.

func
 PopCount(x uint64
) int
 {
    return
 int
(pc[byte
(x>>(0
*8
```

```
))] +
        pc[byte
(x>>(1
*8
))] +
        pc[byte
(x>>(2
*8
))] +
        pc[byte
(x>>(3
*8
))] +
        pc[byte
(x>>(4
*8
))] +
        pc[byte
(x>>(5
*8
))] +
        pc[byte
(x>>(6
*8
))] +
        pc[byte
(x>>(7
*8
))])
}
```

譯註：對於pc這類需要複雜處理的初始化，可以通過將初始化邏輯包裝爲一個匿名函數處理，像下面這樣：

```
// pc[i] is the population count of i.

var
 pc [256
]byte
 = func
() (pc [256
]byte
) {
    for
 i := range
 pc {
        pc[i] = pc[i/2
] + byte
(i&1
)
    }
}()
```

要註意的是在init函數中，range循環隻使用了索引，省略了沒有用到的值部分。循環也可以這樣寫：

```
for
 i, _ := range
 pc {
```

我們在下一節和10.5節還將看到其它使用init函數的地方。

練習 2.3： 重寫PopCount函數，用一個循環代替單一的表達式。比較兩個版本的性能。（11.4節將展示如何繫統地比較兩個不同實現的性能。）

練習 2.4： 用移位算法重寫PopCount函數，每次測試最右邊的1bit，然後統計總數。比較和查表算法的性能差異。

練習 2.5： 表達式 `x&(x-1)` 用於將x的最低的一個非零的bit位清零。使用這個算法重寫PopCount函數，然後比較性能。

2.7. 作用域

一個聲明語句將程序中的實體和一個名字關聯，比如一個函數或一個變量。聲明語句的作用域是指源代碼中可以有效使用這個名字的范圍。

不要將作用域和生命週期混爲一談。聲明語句的作用域對應的是一個源代碼的文本區域；它是一個編譯時的屬性。一個變量的生命週期是指程序運行時變量存在的有效時間段，在此時間區域內它可以被程序的其他部分引用；是一個運行時的概念。

語法塊是由花括弧所包含的一繫列語句，就像函數體或循環體花括弧對應的語法塊那樣。語法塊內部聲明的名字是無法被外部語法塊訪問的。語法決定了內部聲明的名字的作用域范圍。我們可以這樣理解，語法塊可以包含其他類似組批量聲明等沒有用花括弧包含的代碼，我們稱之爲語法塊。有一個語法塊爲整個源代碼，稱爲全局語法塊；然後是每個包的包語法決；每個for、if和switch語句的語法決；每個switch或select的分支也有獨立的語法決；當然也包括顯式書寫的語法塊（花括弧包含的語句）。

聲明語句對應的詞法域決定了作用域范圍的大小。對於內置的類型、函數和常量，比如int、len和true等是在全局作用域的，因此可以在整個程序中直接使用。任何在在函數外部（也就是包級語法域）聲明的名字可以在同一個包的任何源文件中訪問的。對於導入的包，例如tempconv導入的fmt包，則是對應源文件級的作用域，因此隻能在當前的文件中訪問導入的fmt包，當前包的其它源文件無法訪問在當前源文件導入的包。還有許多聲明語句，比如tempconv.CToF函數中的變量c，則是局部作用域的，它隻能在函數內部（甚至隻能是局部的某些部分）訪問。

控製流標號，就是break、continue或goto語句後面跟着的那種標號，則是函數級的作用域。

一個程序可能包含多個同名的聲明，隻要它們在不同的詞法域就沒有關繫。例如，你可以聲明一個局部變量，和包級的變量同名。或者是像2.3.3節的例子那樣，你可以將一個函數參數的名字聲明爲new，雖然內置的new是全局作用域的。但是物極必反，如果濫用不同詞法域可重名的特性的話，可能導致程序很難閱讀。

當編譯器遇到一個名字引用時，如果它看起來像一個聲明，它首先從最內層的詞法域向全局的作用域查找。如果查找失敗，則報告“未聲明的名字”這樣的錯誤。如果該名字在內部和外部的塊分別聲明過，則內部塊的聲明首先被找到。在這種情況下，內部聲明屏蔽了外部同名的聲明，讓外部的聲明的名字無法被訪問：

```
func f() {}

var g = "g"

func main() {
    f := "f"

    fmt.Println(f) // "f"; local var f shadows package-level func f

    fmt.Println(g) // "g"; package-level var

    fmt.Println(h) // compile error: undefined: h

}
```

在函數中詞法域可以深度嵌套，因此內部的一個聲明可能屏蔽外部的聲明。還有許多語法塊是if或for等控製流語句構造的。下面的代碼有三個不同的變量x，因爲它們是定義在不同的詞法域（這個例子隻是爲了演示作用域規則，但不是好的編程風格）。

```
func
 main() {
    x := "hello!"

    for
 i := 0
; i < len
(x); i++ {
        x := x[i]
        if
 x != '!'
 {
            x := x + 'A'
 - 'a'

            fmt.Printf("%c"
, x) // "HELLO" (one letter per iteration)

        }
    }
}
```

在 `x[i]` 和 `x + 'A' - 'a'` 聲明語句的初始化的表達式中都引用了外部作用域聲明的x變量，稍後我們會解釋這個。（註意，後面的表達式與unicode.ToUpper併不等價。）

正如上面例子所示，併不是所有的詞法域都顯式地對應到由花括弧包含的語句；還有一些隱含的規則。上面的for語句創建了兩個詞法域：花括弧包含的是顯式的部分是for的循環體部分詞法域，另外一個隱式的部分則是循環的初始化部分，比如用於迭代變量i的初始化。隱式的詞法域部分的作用域還包含條件測試部分和循環後的迭代部分（ `i++` ），當然也包含循環體詞法域。

下面的例子同樣有三個不同的x變量，每個聲明在不同的詞法域，一個在函數體詞法域，一個在for隱式的初始化詞法域，一個在for循環體詞法域；隻有兩個塊是顯式創建的：

```
func
 main() {
    x := "hello"

    for
 _, x := range
 x {
        x := x + 'A'
 - 'a'

        fmt.Printf("%c"
, x) // "HELLO" (one letter per iteration)

    }
}
```

和for循環類似，if和switch語句也會在條件部分創建隱式詞法域，還有它們對應的執行體詞法域。下面的if-else測試鏈演示了x和y的有效作用域范圍：

```
if
 x := f(); x == 0
 {
    fmt.Println(x)
} else
 if
 y := g(x); x == y {
    fmt.Println(x, y)
} else
 {
    fmt.Println(x, y)
}
fmt.Println(x, y) // compile error: x and y are not visible here
```

第二個if語句嵌套在第一個內部，因此第一個if語句條件初始化詞法域聲明的變量在第二個if中也可以訪問。switch語句的每個分支也有類似的詞法域規則：條件部分爲一個隱式詞法域，然後每個是每個分支的詞法域。

在包級別，聲明的順序併不會影響作用域范圍，因此一個先聲明的可以引用它自身或者是引用後面的一個聲明，這可以讓我們定義一些相互嵌套或遞歸的類型或函數。但是如果一個變量或常量遞歸引用了自身，則會產生編譯錯誤。

在這個程序中：

```
if
 f, err := os.Open(fname); err != nil
 { // compile error: unused: f

    return
 err
}
f.ReadByte() // compile error: undefined f

f.Close()    // compile error: undefined f
```

變量f的作用域隻有在if語句內，因此後面的語句將無法引入它，這將導致編譯錯誤。你可能會收到一個局部變量f沒有聲明的錯誤提示，具體錯誤信息依賴編譯器的實現。

通常需要在if之前聲明變量，這樣可以確保後面的語句依然可以訪問變量：

```
f, err := os.Open(fname)
if
 err != nil
 {
    return
 err
}
f.ReadByte()
f.Close()
```

你可能會考慮通過將ReadByte和Close移動到if的else塊來解決這個問題：

```
if
 f, err := os.Open(fname); err != nil
 {
    return
 err
} else
 {
    // f and err are visible here too

    f.ReadByte()
    f.Close()
}
```

但這不是Go語言推薦的做法，Go語言的習慣是在if中處理錯誤然後直接返迴，這樣可以確保正常執行的語句不需要代碼縮進。

要特別註意短變量聲明語句的作用域范圍，考慮下面的程序，它的目的是獲取當前的工作目録然後保存到一個包級的變量中。這可以本來通過直接調用os.Getwd完成，但是將這個從主邏輯中分離出來可能會更好，特別是在需要處理錯誤的時候。函數log.Fatalf用於打印日誌信息，然後調用os.Exit(1)終止程序。

```
var
 cwd string


func
 init() {
    cwd, err := os.Getwd() // compile error: unused: cwd

    if
 err != nil
 {
        log.Fatalf("os.Getwd failed: %v"
, err)
    }
}
```

雖然cwd在外部已經聲明過，但是 `:=` 語句還是將cwd和err重新聲明爲新的局部變量。因爲內部聲明的cwd將屏蔽外部的聲明，因此上面的代碼併不會正確更新包級聲明的cwd變量。

由於當前的編譯器會檢測到局部聲明的cwd併沒有本使用，然後報告這可能是一個錯誤，但是這種檢測併不可靠。因爲一些小的代碼變更，例如增加一個局部cwd的打印語句，就可能導致這種檢測失效。

```
var
 cwd string


func
 init() {
    cwd, err := os.Getwd() // NOTE:
 wrong!

    if
 err != nil
 {
        log.Fatalf("os.Getwd failed: %v"
, err)
    }
    log.Printf("Working directory = %s"
, cwd)
}
```

全局的cwd變量依然是沒有被正確初始化的，而且看似正常的日誌輸出更是讓這個BUG更加隱晦。

有許多方式可以避免出現類似潛在的問題。最直接的方法是通過單獨聲明err變量，來避免使用 `:=` 的簡短聲明方式：

```
var
 cwd string


func
 init() {
    var
 err error
    cwd, err = os.Getwd()
    if
 err != nil
 {
        log.Fatalf("os.Getwd failed: %v"
, err)
    }
}
```

我們已經看到包、文件、聲明和語句如何來表達一個程序結構。在下面的兩個章節，我們將探討數據的結構。

第3章 基礎數據類型

雖然從底層而言，所有的數據都是由比特組成，但計算機一般操作的是固定大小的數，如整數、浮點數、比特數組、內存地址等。進一步將這些數組織在一起，就可表達更多的對象，例如數據包、像素點、詩歌，甚至其他任何對象。Go語言提供了豐富的數據組織形式，這依賴於Go語言內置的數據類型。這些內置的數據類型，兼顧了硬件的特性和表達複雜數據結構的便捷性。

Go語言將數據類型分爲四類：基礎類型、複合類型、引用類型和接口類型。本章介紹基礎類型，包括：數字、字符串和布爾型。複合數據類型——數組（§4.1）和結構體（§4.2）——是通過組合簡單類型，來表達更加複雜的數據結構。引用類型包括指針（§2.3.2）、切片（§4.2)）字典（§4.3）、函數（§5）、通道（§8），雖然數據種類很多，但它們都是對程序中一個變量或狀態的間接引用。這意味着對任一引用類型數據的脩改都會影響所有該引用的拷貝。我們將在第7章介紹接口類型。

3.1. 整型

Go語言的數值類型包括幾種不同大小的整形數、浮點數和複數。每種數值類型都決定了對應的大小范圍和是否支持正負符號。讓我們先從整形數類型開始介紹。

Go語言同時提供了有符號和無符號類型的整數運算。這里有int8、int16、int32和int64四種截然不同大小的有符號整形數類型，分別對應8、16、32、64bit大小的有符號整形數，與此對應的是uint8、uint16、uint32和uint64四種無符號整形數類型。

這里還有兩種一般對應特定CPU平台機器字大小的有符號和無符號整數int和uint；其中int是應用最廣泛的數值類型。這兩種類型都有同樣的大小，32或64bit，但是我們不能對此做任何的假設；因爲不同的編譯器即使在相同的硬件平台上可能產生不同的大小。

Unicode字符rune類型是和int32等價的類型，通常用於表示一個Unicode碼點。這兩個名稱可以互換使用。同樣byte也是uint8類型的等價類型，byte類型一般用於強調數值是一個原始的數據而不是一個小的整數。

最後，還有一種無符號的整數類型uintptr，沒有指定具體的bit大小但是足以容納指針。uintptr類型隻有在底層編程是才需要，特別是Go語言和C語言函數庫或操作繫統接口相交互的地方。我們將在第十三章的unsafe包相關部分看到類似的例子。

不管它們的具體大小，int、uint和uintptr是不同類型的兄弟類型。其中int和int32也是不同的類型，即使int的大小也是32bit，在需要將int當作int32類型的地方需要一個顯式的類型轉換操作，反之亦然。

其中有符號整數采用2的補碼形式表示，也就是最高bit位用作表示符號位，一個n-bit的有符號數的值域是從 -2^{n-1} 到 $2^{n-1}-1$。無符號整數的所有bit位都用於表示非負數，值域是0到 $2^{n}-1$。例如，int8類型整數的值域是從-128到127，而uint8類型整數的值域是從0到255。

下面是Go語言中關於算術運算、邏輯運算和比較運算的二元運算符，它們按照先級遞減的順序的排列：

```
*      /      %      <<       >>     &       &^
+      -      |      ^
==     !=     <      <=       >      >=
&&
||
```

二元運算符有五種優先級。在同一個優先級，使用左優先結合規則，但是使用括號可以明確優先順序，使用括號也可以用於提陞優先級，例如 `mask & (1 << 28)` 。

對於上表中前兩行的運算符，例如+運算符還有一個與賦值相結合的對應運算符+=，可以用於簡化賦值語句。

算術運算符+、-、 `*` 和 `/` 可以適用與於整數、浮點數和複數，但是取模運算符%僅用於整數間的運算。對於不同編程語言，%取模運算的行爲可能併不相同。在Go語言中，%取模運算符的符號和被取模數的符號總是一致的，因此 `-5%3` 和 `-5%-3` 結果都是-2。除法運算符 `/` 的行爲則依賴於操作數是否爲全爲整數，比如 `5.0/4.0` 的結果是1.25，但是5/4的結果是1，因爲整數除法會向着0方向截斷餘數。

如果一個算術運算的結果，不管是有符號或者是無符號的，如果需要更多的bit位才能正確表示的話，就說明計算結果是溢出了。超出的高位的bit位部分將被丟棄。如果原始的數值是有符號類型，而且最左邊的bit爲是1的話，那麼最終結果可能是負的，例如int8的例子：

```
var
 u uint8
 = 255

fmt.Println(u, u+1
, u*u) // "255 0 1"


var
 i int8
 = 127

fmt.Println(i, i+1
, i*i) // "127 -128 1"
```

兩個相同的整數類型可以使用下面的二元比較運算符進行比較；比較表達式的結果是布爾類型。

```
==    equal to
!=    not equal to
<     less than
<=    less than or equal to
>     greater than
>=    greater than or equal to
```

事實上，布爾型、數字類型和字符串等基本類型都是可比較的，也就是說兩個相同類型的值可以用==和!=進行比較。此外，整數、浮點數和字符串可以根據比較結果排序。許多其它類型的值可能是不可比較的，因此也就可能是不可排序的。對於我們遇到的每種類型，我們需要保證規則的一致性。

這里是一元的加法和減法運算符：

```
+      一元加法（無效果）
-      負數
```

對於整數，+x是0+x的簡寫，-x則是0-x的簡寫；對於浮點數和複數，+x就是x，-x則是x 的負數。

Go語言還提供了以下的bit位操作運算符，前面4個操作運算符併不區分是有符號還是無符號數：

```
&      位運算 AND
|      位運算 OR
^      位運算 XOR
&^     位清空 (AND NOT)
<<     左移
>>     右移
```

位操作運算符 `^` 作爲二元運算符時是按位異或（XOR），當用作一元運算符時表示按位取反；也就是説，它返迴一個每個bit位都取反的數。位操作運算符 `&^` 用於按位置零（AND NOT）：表達式 `z = x &^ y` 結果z的bit位爲0，如果對應y中bit位爲1的話，否則對應的bit位等於x相應的bit位的值。

下面的代碼演示了如何使用位操作解釋uint8類型值的8個獨立的bit位。它使用了Printf函數的%b參數打印二進製格式的數字；其中%08b中08表示打印至少8個字符寬度，不足的前綴部分用0填充。

```
var
 x uint8
 = 1
<<1
 | 1
<<5

var
 y uint8
 = 1
<<1
 | 1
<<2


fmt.Printf("%08b\n"
, x) // "00100010", the set {1, 5}

fmt.Printf("%08b\n"
, y) // "00000110", the set {1, 2}


fmt.Printf("%08b\n"
, x&y)  // "00000010", the intersection {1}

fmt.Printf("%08b\n"
, x|y)  // "00100110", the union {1, 2, 5}

fmt.Printf("%08b\n"
, x^y)  // "00100100", the symmetric difference {2, 5}

fmt.Printf("%08b\n"
, x&^y) // "00100000", the difference {5}


for
 i := uint
(0
); i < 8
; i++ {
    if
 x&(1
<<i) != 0
 { // membership test
```

```
        fmt.Println(i) // "1", "5"

    }
}

fmt.Printf("%08b\n"
, x<<1
) // "01000100", the set {2, 6}

fmt.Printf("%08b\n"
, x>>1
) // "00010001", the set {0, 4}
```

（6.5節給出了一個可以遠大於一個字節的整數集的實現。）

在 `x<<n` 和 `x>>n` 移位運算中，決定了移位操作bit數部分必鬚是無符號數；被操作的x數可以是有符號或無符號數。算術上，一個 `x<<n` 左移運算等價於乘以 2^n，一個 `x>>n` 右移運算等價於除以 2^n。

左移運算用零填充右邊空缺的bit位，無符號數的右移運算也是用0填充左邊空缺的bit位，但是有符號數的右移運算會用符號位的值填充左邊空缺的bit位。因爲這個原因，最好用無符號運算，這樣你可以將整數完全當作一個bit位模式處理。

盡管Go語言提供了無符號數和運算，卽使數值本身不可能出現負數我們還是傾向於使用有符號的int類型，就像數組的長度那樣，雖然使用uint無符號類型似乎是一個更合理的選擇。事實上，內置的len函數返迴一個有符號的int，我們可以像下面例子那樣處理逆序循環。

```
medals := []string
{"gold"
, "silver"
, "bronze"
}
for
 i := len
(medals) - 1
; i >= 0
; i-- {
    fmt.Println(medals[i]) // "bronze", "silver", "gold"

}
```

另一個選擇對於上面的例子來說將是災難性的。如果len函數返迴一個無符號數，那麼i也將是無符號的uint類型，然後條件 `i >= 0` 則永遠爲眞。在三次迭代之後，也就是 `i == 0` 時，i--語句將不會産生-1，而是變成一個uint類型的最大值（可能是 $2^{64} - 1$），然後medals[i]表達式將發生運行時panic異常（§5.9），也就是試圖訪問一個slice范圍以外的元素。

出於這個原因，無符號數往往隻有在位運算或其它特殊的運算場景才會使用，就像bit集合、分析二進製文件格式或者是哈希和加密操作等。它們通常併不用於僅僅是表達非負數量的場合。

一般來說，需要一個顯式的轉換將一個值從一種類型轉化位另一種類型，併且算術和邏輯運算的二元操作中必鬚是相同的類型。雖然這偶爾會導致需要很長的表達式，但是它消除了所有和類型相關的問題，而且也使得程序容易理解。

在很多場景，會遇到類似下面的代碼通用的錯誤：

```
var
 apples int32
 = 1

var
 oranges int16
 = 2

var
 compote int
 = apples + oranges // compile error
```

當嚐試編譯這三個語句時，將產生一個錯誤信息：

```
invalid operation: apples + oranges (mismatched types int32 and int16)
```

這種類型不匹配的問題可以有幾種不同的方法脩複，最常見方法是將它們都顯式轉型爲一個常見類型：

```
var
 compote = int
(apples) + int
(oranges)
```

如2.5節所述，對於每種類型T，如果轉換允許的話，類型轉換操作T(x)將x轉換爲T類型。許多整形數之間的相互轉換併不會改變數值；它們隻是告訴編譯器如何解釋這個值。但是對於將一個大尺寸的整數類型轉爲一個小尺寸的整數類型，或者是將一個浮點數轉爲整數，可能會改變數值或丟失精度：

```
f := 3.141
 // a float64

i := int
(f)
fmt.Println(f, i) // "3.141 3"

f = 1.99

fmt.Println(int
(f)) // "1"
```

浮點數到整數的轉換將丟失任何小數部分，然後向數軸零方向截斷。你應該避免對可能會超出目標類型表示范圍的數值類型轉換，因爲截斷的行爲可能依賴於具體的實現：

```
f := 1e100
  // a float64

i := int
(f) // 結果依賴於具體實現
```

任何大小的整數字面值都可以用以0開始的八進製格式書寫，例如0666；或用以0x或0X開頭的十六進製格式書寫，例如0xdeadbeef。十六進製數字可以用大寫或小寫字母。如今八進製數據通常用於POSIX操作繫統上的文件訪問權限標誌，十六進製數字則更強調數字值的bit位模式。

當使用fmt包打印一個數值時，我們可以用%d、%o或%x參數控製輸出的進製格式，就像下面的例子：

```
o := 0666

fmt.Printf("%d %[1]o %#[1]o\n"
, o) // "438 666 0666"

x := int64
(0xdeadbeef
)
fmt.Printf("%d %[1]x %#[1]x %#[1]X\n"
, x)
// Output:

// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF
```

請註意fmt的兩個使用技巧。通常Printf格式化字符串包含多個%參數時將會包含對應相同數量的額外操作數，但是%之後的 `[1]` 副詞告訴Printf函數再次使用第一個操作數。第二，%後的 `#` 副詞告訴Printf在用%o、%x或%X輸出時生成0、0x或0X前綴。

字符面值通過一對單引號直接包含對應字符。最簡單的例子是ASCII中類似'a'寫法的字符面值，但是我們也可以通過轉義的數值來表示任意的Unicode碼點對應的字符，馬上將會看到這樣的例子。

字符使用 `%c` 參數打印，或者是用 `%q` 參數打印帶單引號的字符：

```
ascii := 'a'

unicode := '国'

newline := '\n'

fmt.Printf("%d %[1]c %[1]q\n"
, ascii)   // "97 a 'a'"

fmt.Printf("%d %[1]c %[1]q\n"
, unicode) // "22269 国 '国'"

fmt.Printf("%d %[1]q\n"
, newline)       // "10 '\n'"
```

3.2. 浮點數

Go語言提供了兩種精度的浮點數，float32和float64。它們的算術規范由IEEE754浮點數国際標準定義，該浮點數規范被所有現代的CPU支持。

這些浮點數類型的取值范圍可以從很微小到很鉅大。浮點數的范圍極限值可以在math包找到。常量math.MaxFloat32表示float32能表示的最大數值，大約是 3.4e38；對應的math.MaxFloat64常量大約是1.8e308。它們分別能表示的最小值近似爲1.4e-45和4.9e-324。

一個float32類型的浮點數可以提供大約6個十進製數的精度，而float64則可以提供約15個十進製數的精度；通常應該優先使用float64類型，因爲float32類型的纍計計算誤差很容易擴散，併且float32能精確表示的正整數併不是很大（譯註：因爲float32的有效bit位隻有23個，其它的bit位用於指數和符號；當整數大於23bit能表達的范圍時，float32的表示將出現誤差）：

```
var
 f float32
 = 16777216
 // 1 << 24

fmt.Println(f == f+1
)    // "true"!
```

浮點數的字面值可以直接寫小數部分，像這樣：

```
const
 e = 2.71828
 // (approximately)
```

小數點前面或後面的數字都可能被省略（例如.707或1.）。很小或很大的數最好用科學計數法書寫，通過e或E來指定指數部分：

```
const
 Avogadro = 6.02214129e23
  // 阿伏伽德羅常數

const
 Planck   = 6.62606957e-34
 // 普朗剋常數
```

用Printf函數的%g參數打印浮點數，將采用更緊湊的表示形式打印，併提供足夠的精度，但是對應表格的數據，使用%e（帶指數）或%f的形式打印可能更合適。所有的這三個打印形式都可以指定打印的寬度和控製打印精度。

```
for
 x := 0
; x < 8
; x++ {
    fmt.Printf("x = %d e^x = %8.3f\n"
, x, math.Exp(float64
(x)))
}
```

上面代碼打印e的冪，打印精度是小數點後三個小數精度和8個字符寬度：

```
x = 0       e^x =    1.000
x = 1       e^x =    2.718
x = 2       e^x =    7.389
x = 3       e^x =   20.086
x = 4       e^x =   54.598
x = 5       e^x =  148.413
x = 6       e^x =  403.429
x = 7       e^x = 1096.633
```

math包中除了提供大量常用的數學函數外，還提供了IEEE754浮點數標準中定義的特殊值的創建和測試：正無窮大和負無窮大，分別用於表示太大溢出的數字和除零的結果；還有NaN非數，一般用於表示無效的除法操作結果0/0或Sqrt(-1).

```
var
 z float64

fmt.Println(z, -z, 1
/z, -1
/z, z/z) // "0 -0 +Inf -Inf NaN"
```

函數math.IsNaN用於測試一個數是否是非數NaN，math.NaN則返迴非數對應的值。雖然可以用math.NaN來表示一個非法的結果，但是測試一個結果是否是非數NaN則是充滿風險的，因爲NaN和任何數都是不相等的（譯註：在浮點數中，NaN、正無窮大和負無窮大都不是唯一的，每個都有非常多種的bit模式表示）：

```
nan := math.NaN()
fmt.Println(nan == nan, nan < nan, nan > nan) // "false false false"
```

如果一個函數返迴的浮點數結果可能失敗，最好的做法是用單獨的標誌報告失敗，像這樣：

```
func
 compute() (value float64
, ok bool
) {
    // ...

    if
 failed {
        return
 0
, false

    }
    return
 result, true

}
```

接下來的程序演示了通過浮點計算生成的圖形。它是帶有兩個參數的z = f(x, y)函數的三維形式，使用了可縮放矢量圖形（SVG）格式輸出，SVG是一個用於矢量線繪製的XML標準。圖3.1顯示了sin(r)/r函數的輸出圖形，其中r是sqrt(*xx*+*y* y)。

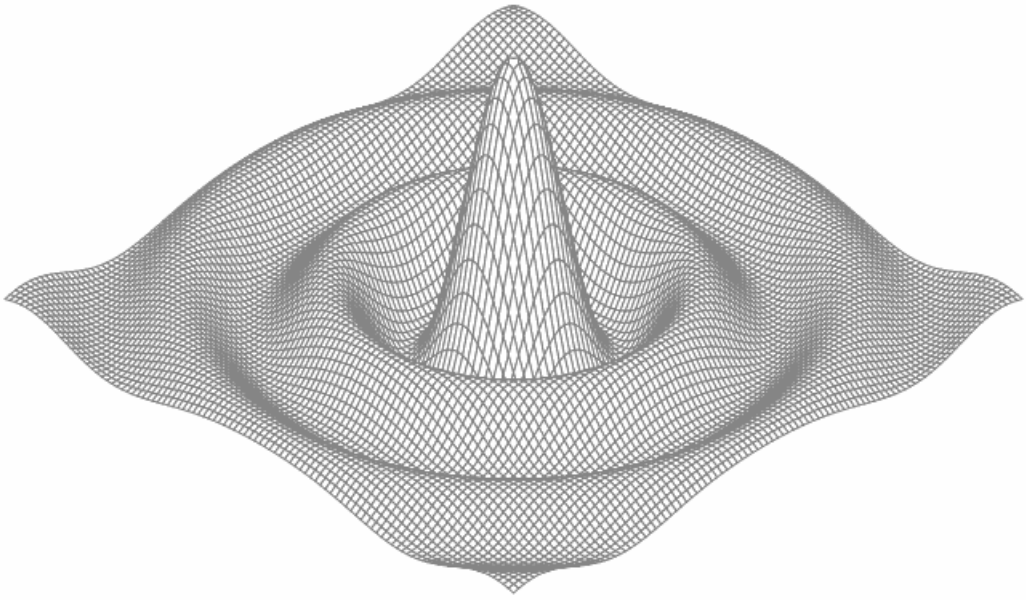


Figure 3.1. A surface plot of the function `sin(r)/r`.

```
gopl.io/ch3/surface
// Surface computes an SVG rendering of a 3-D surface function.

package
 main

import
 (
    "fmt"

    "math"
```

```
)

const
 (
    width, height = 600
, 320
            // canvas size in pixels

    cells         = 100
                 // number of grid cells

    xyrange       = 30.0
                // axis ranges (-xyrange..+xyrange)

    xyscale       = width / 2
 / xyrange // pixels per x or y unit

    zscale        = height * 0.4
        // pixels per z unit

    angle         = math.Pi / 6
         // angle of x, y axes (=30°)

)

var
 sin30, cos30 = math.Sin(angle), math.Cos(angle) // sin(30°), cos(30°)


func
 main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "
+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "
+
        "width='%d' height='%d'>"
, width, height)
    for
 i := 0
; i < cells; i++ {
        for
 j := 0
; j < cells; j++ {
            ax, ay := corner(i+1
, j)
            bx, by := corner(i, j)
            cx, cy := corner(i, j+1
)
            dx, dy := corner(i+1
, j+1
)
```

```
            fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g'/>\n"
,
                ax, ay, bx, by, cx, cy, dx, dy)
        }
    }
    fmt.Println("</svg>"
)
}

func
 corner(i, j int
) (float64
, float64
) {
    // Find point (x,y) at corner of cell (i,j).

    x := xyrange * (float64
(i)/cells - 0.5
)
    y := xyrange * (float64
(j)/cells - 0.5
)

    // Compute surface height z.

    z := f(x, y)

    // Project (x,y,z) isometrically onto 2-D SVG canvas (sx,sy).

    sx := width/2
 + (x-y)*cos30*xyscale
    sy := height/2
 + (x+y)*sin30*xyscale - z*zscale
    return
 sx, sy
}

func
 f(x, y float64
) float64
 {
    r := math.Hypot(x, y) // distance from (0,0)

    return
 math.Sin(r) / r
}
```

要註意的是corner函數返迴了兩個結果，分別對應每個網格頂點的坐標參數。

要解釋這個程序是如何工作的需要一些基本的幾何學知識，但是我們可以跳過幾何學原理，因爲程序的重點是演示浮點數運算。程序的本質是三個不同的坐標繫中映射關繫，如圖3.2所示。第一個是100x100的二維網格，對應整數整數坐標(i,j)，從遠處的(0, 0)位置開始。我們從遠處向前面繪製，因此遠處先繪製的多邊形有可能被前面後繪製的多邊形覆蓋。

第二個坐標繫是一個三維的網格浮點坐標(x,y,z)，其中x和y是i和j的線性函數，通過平移轉換位網格單元的中心，然後用xyrange繫數縮放。高度z是函數f(x,y)的值。

第三個坐標繫是一個二維的畫布，起點(0,0)在左上角。畫布中點的坐標用(sx, sy)表示。我們使用等角投影將三維點

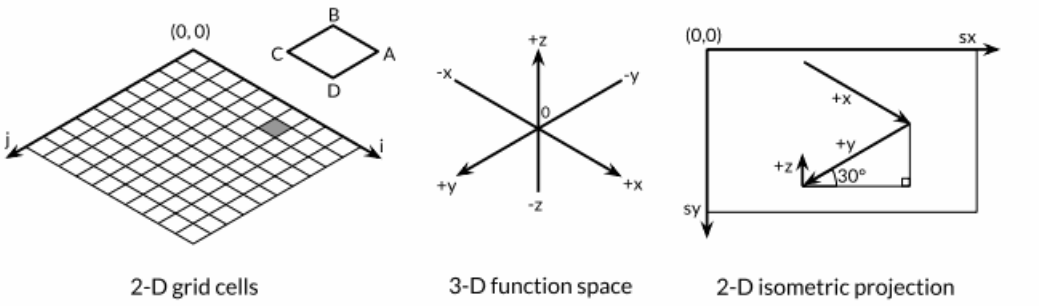


Figure 3.2. Three different coordinate systems.

(x,y,z)投影到二維的畫布中。畫布中從遠處到右邊的點對應較大的x值和較大的y值。併且畫布中x和y值越大，則對應的z值越小。x和y的垂直和水平縮放繫數來自30度角的正絃和餘絃值。z的縮放繫數0.4，是一個任意選擇的參數。

對於二維網格中的每一個網格單元，main函數計算單元的四個頂點在畫布中對應多邊形ABCD的頂點，其中B對應(i,j)頂點位置，A、C和D是其它相鄰的頂點，然後輸出SVG的繪製指令。

練習 3.1： 如果f函數返迴的是無限製的float64值，那麼SVG文件可能輸出無效的多邊形元素（雖然許多SVG渲染器會妥善處理這類問題）。脩改程序跳過無效的多邊形。

練習 3.2： 試驗math包中其他函數的渲染圖形。你是否能輸出一個egg box、moguls或a saddle圖案?

練習 3.3： 根據高度給每個多邊形上色，那樣峯值部將是紅色(#ff0000)，谷部將是藍色(#0000ff)。

練習 3.4： 參考1.7節Lissajous例子的函數，構造一個web服務器，用於計算函數麴面然後返迴SVG數據給客戶端。服務器必鬚設置Content-Type頭部：

```
w.Header().Set("Content-Type"
, "image/svg+xml"
)
```

（這一步在Lissajous例子中不是必鬚的，因爲服務器使用標準的PNG圖像格式，可以根據前面的512個字節自動輸出對應的頭部。）允許客戶端通過HTTP請求參數設置高度、寬度和顏色等參數。

3.3. 複數

Go語言提供了兩種精度的複數類型：complex64和complex128，分別對應float32和float64兩種浮點數精度。內置的complex函數用於構建複數，內建的real和imag函數分別返回複數的實部和虛部：

```
var
 x complex128
 = complex
(1
, 2
) // 1+2i

var
 y complex128
 = complex
(3
, 4
) // 3+4i

fmt.Println(x*y)                // "(-5+10i)"

fmt.Println(real
(x*y))           // "-5"

fmt.Println(imag
(x*y))           // "10"
```

如果一個浮點數面值或一個十進製整數面值後面跟着一個i，例如3.141592i或2i，它將構成一個複數的虛部，複數的實部是0：

```
fmt.Println(1i
 * 1i
) // "(-1+0i)", i^2 = -1
```

在常量算術規則下，一個複數常量可以加到另一個普通數值常量（整數或浮點數、實部或虛部），我們可以用自然的方式書寫複數，就像1+2i或與之等價的寫法2i+1。上面x和y的聲明語句還可以簡化：

```
x := 1
 + 2i

y := 3
 + 4i
```

複數也可以用==和!=進行相等比較。隻有兩個複數的實部和虛部都相等的時候它們才是相等的（譯註：浮點數的相等比較是危險的，需要特別小心處理精度問題）。

math/cmplx包提供了複數處理的許多函數，例如求複數的平方根函數和求冪函數。

```
fmt.Println(cmplx.Sqrt(-1
)) // "(0+1i)"
```

下面的程序使用complex128複數算法來生成一個Mandelbrot圖像。

```
gopl.io/ch3/mandelbrot

// Mandelbrot emits a PNG image of the Mandelbrot fractal.

package
 main

import
 (
    "image"

    "image/color"

    "image/png"

    "math/cmplx"

    "os"

)


func
 main() {
    const
 (
        xmin, ymin, xmax, ymax = -2
, -2
, +2
, +2

        width, height          = 1024
, 1024

    )

    img := image.NewRGBA(image.Rect(0
, 0
, width, height))
    for
 py := 0
; py < height; py++ {
        y := float64
```

```
(py)/height*(ymax-ymin) + ymin
        for
 px := 0
; px < width; px++ {
            x := float64
(px)/width*(xmax-xmin) + xmin
            z := complex
(x, y)
            // Image point (px, py) represents complex value z.

            img.Set(px, py, mandelbrot(z))
        }
    }
    png.Encode(os.Stdout, img) // NOTE:
 ignoring errors

}

func
 mandelbrot(z complex128
) color.Color {
    const
 iterations = 200

    const
 contrast = 15



    var
 v complex128

    for
 n := uint8
(0
); n < iterations; n++ {
        v = v*v + z
        if
 cmplx.Abs(v) > 2
 {
            return
 color.Gray{255
 - contrast*n}
        }
    }
    return
 color.Black
}
```

用於遍歷1024x1024圖像每個點的兩個嵌套的循環對應-2到+2區間的複數平面。程序反複測試每個點對應複數值平方值加一個增量值對應的點是否超出半徑爲2的圓。如果超過了，通過根據預設置的逃逸迭代次數對應的灰度顏色來代替。如果不是，那麼該點屬於Mandelbrot集合，使用黑色顏色標記。最終程序將生成的PNG格式分形圖像圖像輸出到標準輸出，如圖3.3所示。

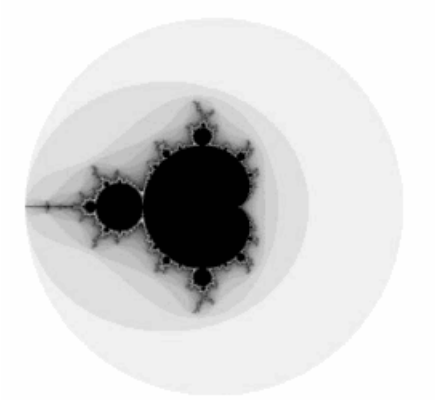


Figure 3.3. The Mandelbrot set.

練習 3.5： 實現一個綵色的Mandelbrot圖像，使用image.NewRGBA創建圖像，使用color.RGBA或color.YCbCr生成顏色。

練習 3.6： 陞采樣技術可以降低每個像素對計算顏色值和平均值的影響。簡單的方法是將每個像素分層四個子像素，實現它。

練習 3.7： 另一個生成分形圖像的方式是使用牛頓法來求解一個複數方程，例如 $z_4 - 1 = 0$ 。每個起點到四個根的迭代次數對應陰影的灰度。方程根對應的點用顏色表示。

練習 3.8： 通過提高精度來生成更多級別的分形。使用四種不同精度類型的數字實現相同的分形：complex64、complex128、big.Float和big.Rat。（後面兩種類型在math/big包聲明。Float是有指定限精度的浮點數；Rat是無效精度的有理數。）它們間的性能和內存使用對比如何？當渲染圖可見時縮放的級別是多少？

練習 3.9： 編寫一個web服務器，用於給客戶端生成分形的圖像。運行客戶端用過HTTP參數參數指定x,y和zoom參數。

3.4. 布爾型

一個布爾類型的值隻有兩種：true和false。if和for語句的條件部分都是布爾類型的值，併且==和<等比較操作也會産生布爾型的值。一元操作符 `!` 對應邏輯非操作，因此 `!true` 的值爲 `false` ，更羅嗦的説法是 `(!true==false)==true` ，雖然表達方式不一樣，不過我們一般會采用簡潔的布爾表達式，就像用x來表示 `x==true` 。

布爾值可以和&&（AND）和||（OR）操作符結合，併且可能會有短路行爲：如果運算符左邊值已經可以確定整個布爾表達式的值，那麽運算符右邊的值將不在被求值，因此下面的表達式總是安全的：

```
s != ""
 && s[0
] == 'x'
```

其中s[0]操作如果應用於空字符串將會導致panic異常。

因爲 `&&` 的優先級比 `||` 高（助記： `&&` 對應邏輯乘法， `||` 對應邏輯加法，乘法比加法優先級要高），下面形式的布爾表達式是不需要加小括弧的：

```
if
 'a'
 <= c && c <= 'z'
 ||
    'A'
 <= c && c <= 'Z'
 ||
    '0'
 <= c && c <= '9'
 {
    // ...ASCII letter or digit...

}
```

布爾值併不會隱式轉換爲數字值0或1，反之亦然。必鬚使用一個顯式的if語句輔助轉換：

```
i := 0

if
 b {
    i = 1

}
```

如果需要經常做類似的轉換, 包裝成一個函數會更方便:

```
// btoi returns 1 if b is true and 0 if false.

func
 btoi(b bool
) int
 {
    if
 b {
        return
 1

    }
    return
 0

}
```

數字到布爾型的逆轉換則非常簡單, 不過爲了保持對稱, 我們也可以包裝一個函數:

```
// itob reports whether i is non-zero.

func
 itob(i int
) bool
 { return
 i != 0
 }
```

3.5. 字符串

一個字符串是一個不可改變的字節序列。字符串可以包含任意的數據，包括byte值0，但是通常是用來包含人類可讀的文本。文本字符串通常被解釋爲采用UTF8編碼的Unicode碼點（rune）序列，我們稍後會詳細討論這個問題。

内置的len函數可以返迴一個字符串中的字節數目（不是rune字符數目），索引操作s[i]返迴第i個字節的字節值，i必鬚滿足$0 \le i < len(s)$條件約束。

```
s := "hello, world"

fmt.Println(len
(s))     // "12"

fmt.Println(s[0
], s[7
]) // "104 119" ('h' and 'w')
```

如果試圖訪問超出字符串索引范圍的字節將會導致panic異常：

```
c := s[len
(s)] // panic: index out of range
```

第i個字節併不一定是字符串的第i個字符，因爲對於非ASCII字符的UTF8編碼會要兩個或多個字節。我們先簡單説下字符的工作方式。

子字符串操作s[i:j]基於原始的s字符串的第i個字節開始到第j個字節（併不包含j本身）生成一個新字符串。生成的新字符串將包含j-i個字節。

```
fmt.Println(s[0
:5
]) // "hello"
```

同樣，如果索引超出字符串范圍或者j小於i的話將導致panic異常。

不管i還是j都可能被忽略，當它們被忽略時將采用0作爲開始位置，采用len(s)作爲結束的位置。

```
fmt.Println(s[:5
]) // "hello"

fmt.Println(s[7
:]) // "world"

fmt.Println(s[:])  // "hello, world"
```

其中+操作符將兩個字符串鏈接構造一個新字符串：

```
fmt.Println("goodbye"
 + s[5
:]) // "goodbye, world"
```

字符串可以用==和<進行比較；比較通過逐個字節比較完成的，因此比較的結果是字符串自然編碼的順序。

字符串的值是不可變的：一個字符串包含的字節序列永遠不會被改變，當然我們也可以給一個字符串變量分配一個新字符串值。可以像下面這樣將一個字符串追加到另一個字符串：

```
s := "left foot"

t := s
s += ", right foot"
```

這併不會導致原始的字符串值被改變，但是變量s將因爲+=語句持有一個新的字符串值，但是t依然是包含原先的字符串值。

```
fmt.Println(s) // "left foot, right foot"

fmt.Println(t) // "left foot"
```

因爲字符串是不可脩改的，因此嚐試脩改字符串內部數據的操作也是被禁止的：

```
s[0
] = 'L'
 // compile error: cannot assign to s[0]
```

不變性意味如果兩個字符串共享相同的底層數據的話也是安全的，這使得複製任何長度的字符串代價是低廉的。同樣，一個字符串s和對應的子字符串切片s[7:]的操作也可以安全地共享相同的內存，因此字符串切片操作代價也是低廉的。在這兩種情況下都沒有必要分配新的內存。 圖3.4演示了一個字符串和兩個字串共享相同的底層數據。

3.5.1. 字符串面值

字符串值也可以用字符串面值方式編寫，隻要將一繫列字節序列包含在雙引號即可：

```
"Hello, 世界"
```

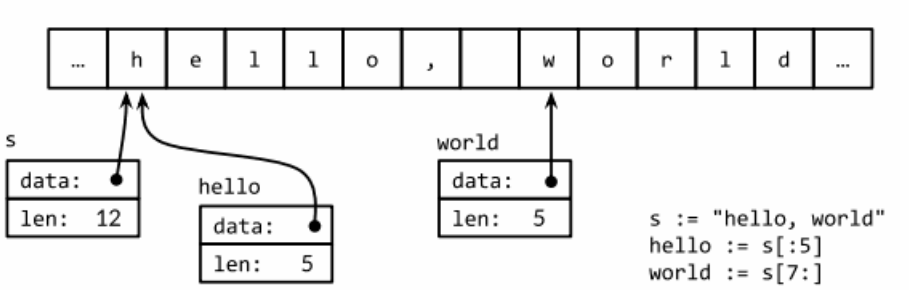


Figure 3.4. The string "hello, world" and two substrings.

因爲Go語言源文件總是用UTF8編碼，併且Go語言的文本字符串也以UTF8編碼的方式處理，因此我們可以將Unicode碼點也寫到字符串面值中。

在一個雙引號包含的字符串面值中，可以用以反斜槓 `\` 開頭的轉義序列插入任意的數據。下面的換行、迴車和製表符等是常見的ASCII控製代碼的轉義方式：

```
\a      響鈴
\b      退格
\f      換頁
\n      換行
\r      迴車
\t      製表符
\v      垂直製表符
\'      單引號（隻用在 '\'' 形式的rune符號面值中）
\"      雙引號（隻用在 "..." 形式的字符串面值中）
\\      反斜槓
```

可以通過十六進製或八進製轉義在字符串面值包含任意的字節。一個十六進製的轉義形式是\xhh，其中兩個h表示十六進製數字（大寫或小寫都可以）。一個八進製轉義形式是\ooo，包含三個八進製的o數字（0到7），但是不能超過 `\377` （譯註：對應一個字節的范圍，十進製爲255）。每一個單一的字節表達一個特定的值。稍後我們將看到如何將一個Unicode碼點寫到字符串面值中。

一個原生的字符串面值形式是 `...` ，使用反引號 代替雙引號。在原生的字符串面值中，沒有轉義操作；全部的內容都是字面的意思，包含退格和換行，因此一個程序中的原生字符串面值可能跨越多行（譯註：在原生字符串面值內部是無法直接寫 字符的，可以用八進製或十六進製轉義或+"```"鏈接字符串常量完成）。唯一的特殊處理是會刪除迴車以保證在所有平台上的值都是一樣的，包括那些把迴車也放入文本文件的繫統（譯註：Windows繫統會把迴車和換行一起放入文本文件中）。

原生字符串面值用於編寫正則表達式會很方便，因爲正則表達式往往會包含很多反斜槓。原生字符串面值同時被廣泛應用於HTML模闆、JSON面值、命令行提示信息以及那些需要擴展到多行的場景。

```
const
 GoUsage = `Go is a tool for managing Go source code.

Usage:
    go command [arguments]
...`
```

3.5.2. Unicode

在很久以前，世界還是比較簡單的，起碼計算機世界就隻有一個ASCII字符集：美国信息交換標準代碼。ASCII，更準確地説是美国的ASCII，使用7bit來表示128個字符：包含英文字母的大小寫、數字、各種標點符號和設置控製符。對於早期的計算機程序來説，這些就足夠了，但是這也導致了世界上很多其他地區的用戶無法直接使用自己的符號繫統。隨着互聯網的發展，混合多種語言的數據變得很常見（譯註：比如本身的英文原文或中文翻譯都包含了ASCII、中文、日文等多種語言字符）。如何有效處理這些包含了各種語言的豐富多樣的文本數據呢？

答案就是使用Unicode（ http://unicode.org ），它收集了這個世界上所有的符號繫統，包括重音符號和其它變音符號，製表符和迴車符，還有很多神祕的符號，每個符號都分配一個唯一的Unicode碼點，Unicode碼點對應Go語言中的rune整數類型（譯註：rune是int32等價類型）。

在第八版本的Unicode標準收集了超過120,000個字符，涵蓋超過100多種語言。這些在計算機程序和數據中是如何體現的呢？通用的表示一個Unicode碼點的數據類型是int32，也就是Go語言中rune對應的類型；它的同義詞rune符文正是這個意思。

我們可以將一個符文序列表示爲一個int32序列。這種編碼方式叫UTF-32或UCS-4，每個Unicode碼點都使用同樣的大小32bit來表示。這種方式比較簡單統一，但是它會浪費很多存儲空間，因爲大數據計算機可讀的文本是ASCII字符，本來每個ASCII字符隻需要8bit或1字節就能表示。而且即使是常用的字符也遠少於65,536個，也就是説用16bit編碼方式就能表達常用字符。但是，還有其它更好的編碼方法嗎？

3.5.3. UTF-8

UTF8是一個將Unicode碼點編碼爲字節序列的變長編碼。UTF8編碼由Go語言之父Ken Thompson和Rob Pike共同發明的，現在已經是Unicode的標準。UTF8編碼使用1到4個字節來表示每個Unicode碼點，ASCII部分字符隻使用1個字節，常用字符部分使用2或3個字節表示。每個符號編碼後第一個字節的高端bit位用於表示總共有多少編碼個字節。如果第一個字節的高端bit爲0，則表示對應7bit的ASCII字符，ASCII字符每個字符依然是一個字節，和傳統的ASCII編碼兼容。如果第一個字節的高端bit是110，則説明需要2個字節；後續的每個高端bit都以10開頭。更大的Unicode碼點也是采用類似的策略處理。

```
0xxxxxxx                            runes 0-127    (ASCII)
110xxxxx 10xxxxxx                   128-2047       (values <128 unused)
1110xxxx 10xxxxxx 10xxxxxx          2048-65535     (values <2048 unused)
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx  65536-0x10ffff (other values unused)
```

變長的編碼無法直接通過索引來訪問第n個字符，但是UTF8編碼獲得了很多額外的優點。首先UTF8編碼比較緊湊，完全兼容ASCII碼，併且可以自動同步：它可以通過向前迴朔最多2個字節就能確定當前字符編碼的開始字節的位置。它也是一個前綴編碼，所以當從左向右解碼時不會有任何歧義也併不需要向前查看（譯註：像GBK之類的編碼，如果不知道起點位置則可能會出現歧義）。沒有任何字符的編碼是其它字符編碼的子串，或是其它編碼序列的字串，因此蒐索一個字符時隻要蒐索它的字節編碼序列即可，不用擔心前後的上下文會對蒐索結果産生榦擾。同時UTF8編碼的順序和Unicode碼點的順序一致，因此可以直接排序UTF8編碼序列。同時因爲沒有嵌入的NUL(0)字節，可以很好地兼容那些使用NUL作爲字符串結尾的編程語言。

Go語言的源文件采用UTF8編碼，併且Go語言處理UTF8編碼的文本也很出色。unicode包提供了諸多處理rune字符相關功能的函數（比如區分字母和數組，或者是字母的大寫和小寫轉換等），unicode/utf8包則提供了用於rune字符序列的UTF8編碼和解碼的功能。

有很多Unicode字符很難直接從鍵盤輸入，併且還有很多字符有着相似的結構；有一些甚至是不可見的字符（譯註：中文和日文就有很多相似但不同的字）。Go語言字符串面值中的Unicode轉義字符讓我們可以通過Unicode碼點輸入特殊的字符。有兩種形式：\uhhhh對應16bit的碼點值，\Uhhhhhhhh對應32bit的碼點值，其中h是一個十六進製數字；一般很少需要使用32bit的形式。每一個對應碼點的UTF8編碼。例如：下面的字母串面值都表示相同的值：

```
"世界"
"\xe4\xb8\x96\xe7\x95\x8c"
"\u4e16\u754c"
"\U00004e16\U0000754c"
```

上面三個轉義序列都爲第一個字符串提供替代寫法，但是它們的值都是相同的。

Unicode轉義也可以使用在rune字符中。下面三個字符是等價的：

```
'世' '\u4e16' '\U00004e16'
```

對於小於256碼點值可以寫在一個十六進製轉義字節中，例如'\x41'對應字符'A'，但是對於更大的碼點則必鬚使用\u或\U轉義形式。因此，'\xe4\xb8\x96'併不是一個合法的rune字符，雖然這三個字節對應一個有效的UTF8編碼的碼點。

得益於UTF8編碼優良的設計，諸多字符串操作都不需要解碼操作。我們可以不用解碼直接測試一個字符串是否是另一個字符串的前綴：

```
func
 HasPrefix(s, prefix string
) bool
 {
    return
 len
(s) >= len
(prefix) && s[:len
(prefix)] == prefix
}
```

或者是後綴測試：

```
func
 HasSuffix(s, suffix string
) bool
 {
    return
 len
(s) >= len
(suffix) && s[len
(s)-len
(suffix):] == suffix
}
```

或者是包含子串測試：

```
func
 Contains(s, substr string
) bool
 {
    for
 i := 0
; i < len
(s); i++ {
        if
 HasPrefix(s[i:], substr) {
            return
 true

        }
    }
    return
 false

}
```

對於UTF8編碼後文本的處理和原始的字節處理邏輯是一樣的。但是對應很多其它編碼則併不是這樣的。（上面的函數都來自strings字符串處理包，眞實的代碼包含了一個用哈希技術優化的Contains 實現。）

另一方面，如果我們眞的關心每個Unicode字符，我們可以使用其它處理方式。考慮前面的第一個例子中的字符串，它包混合了中西兩種字符。圖3.5展示了它的內存表示形式。字符串包含13個字節，以UTF8形式編碼，但是隻對應9個Unicode字符：

```
import
 "unicode/utf8"


s := "Hello, 世界"

fmt.Println(len
(s))                    // "13"

fmt.Println(utf8.RuneCountInString(s)) // "9"
```

爲了處理這些眞實的字符，我們需要一個UTF8解碼器。unicode/utf8包提供了該功能，我們可以這樣使用：

```
for
 i := 0
; i < len
(s); {
    r, size := utf8.DecodeRuneInString(s[i:])
    fmt.Printf("%d\t%c\n"
, i, r)
    i += size
}
```

每一次調用DecodeRuneInString函數都返回一個r和長度，r對應字符本身，長度對應r采用UTF8編碼後的編碼字節數目。長度可以用於更新第i個字符在字符串中的字節索引位置。但是這種編碼方式是笨拙的，我們需要更簡潔的語法。幸運的是，Go語言的range循環在處理字符串的時候，會自動隱式解碼UTF8字符串。下面的循環運行如圖3.5所示；需要註意的是對於非ASCII，索引更新的步長將超過1個字節。

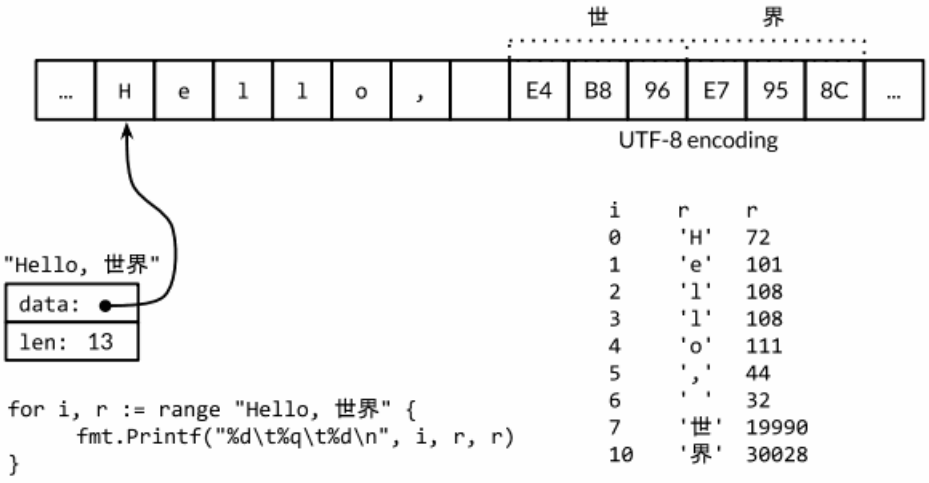


Figure 3.5. A range loop decodes a UTF-8-encoded string.

```
for
 i, r := range
 "Hello, 世界"
 {
    fmt.Printf("%d\t%q\t%d\n"
, i, r, r)
}
```

我們可以使用一個簡單的循環來統計字符串中字符的數目，像這樣：

```
n := 0

for
 _, _ = range
 s {
    n++
}
```

像其它形式的循環那樣，我們也可以忽略不需要的變量：

```
n := 0

for
 range
 s {
    n++
}
```

或者我們可以直接調用utf8.RuneCountInString(s)函數。

正如我們前面提到的，文本字符串采用UTF8編碼隻是一種慣例，但是對於循環的眞正字符串併不是一個慣例，這是正確的。如果用於循環的字符串隻是一個普通的二進製數據，或者是含有錯誤編碼的UTF8數據，將會發送什麼呢？

每一個UTF8字符解碼，不管是顯式地調用utf8.DecodeRuneInString解碼或是在range循環中隱式地解碼，如果遇到一個錯誤的UTF8編碼輸入，將生成一個特別的Unicode字符'\uFFFD'，在印刷中這個符號通常是一個黑色六角或鑽石形狀，里面包含一個白色的問號（?）。當程序遇到這樣的一個字符，通常是一個危險信號，說明輸入併不是一個完美沒有錯誤的UTF8字符串。

UTF8字符串作爲交換格式是非常方便的，但是在程序內部采用rune序列可能更方便，因爲rune大小一致，支持數組索引和方便切割。

string接受到[]rune的類型轉換，可以將一個UTF8編碼的字符串解碼爲Unicode字符序列：

```
// "program" in Japanese katakana

s := "プログラム"

fmt.Printf("% x\n"
, s) // "e3 83 97 e3 83 ad e3 82 b0 e3 83 a9 e3 83 a0"

r := []rune
(s)
fmt.Printf("%x\n"
, r)  // "[30d7 30ed 30b0 30e9 30e0]"
```

（在第一個Printf中的 `% x` 參數用於在每個十六進製數字前插入一個空格。）

如果是將一個[]rune類型的Unicode字符slice或數組轉爲string，則對它們進行UTF8編碼：

```
fmt.Println(string
(r)) // "プログラム"
```

將一個整數轉型爲字符串意思是生成以隻包含對應Unicode碼點字符的UTF8字符串：

```
fmt.Println(string
(65
))     // "A", not "65"

fmt.Println(string
(0x4eac
)) // "京"
```

如果對應碼點的字符是無效的，則用'\uFFFD'無效字符作爲替換：

```
fmt.Println(string
(1234567
)) // "(?)"
```

3.5.4. 字符串和Byte切片

標準庫中有四個包對字符串處理尤爲重要：bytes、strings、strconv和unicode包。strings包提供了許多如字符串的查詢、替換、比較、截斷、拆分和合併等功能。

bytes包也提供了很多類似功能的函數，但是針對和字符串有着相同結構的[]byte類型。因爲字符串是隻讀的，因此逐步構建字符串會導致很多分配和複製。在這種情況下，使用bytes.Buffer類型將會更有效，稍後我們將展示。

strconv包提供了布爾型、整型數、浮點數和對應字符串的相互轉換，還提供了雙引號轉義相關的轉換。

unicode包提供了IsDigit、IsLetter、IsUpper和IsLower等類似功能，它們用於給字符分類。每個函數有一個單一的rune類型的參數，然後返迴一個布爾值。而像ToUpper和ToLower之類的轉換函數將用於rune字符的大小寫轉換。所有的這些函數都是遵循Unicode標準定義的字母、數字等分類規范。strings包也有類似的函數，它們是ToUpper和ToLower，將原始字符串的每個字符都做相應的轉換，然後返迴新的字符串。

下面例子的basename函數靈感於Unix shell的同名工具。在我們實現的版本中，basename(s)將看起來像是繫統路徑的前綴刪除，同時將看似文件類型的後綴名部分刪除：

```
fmt.Println(basename("a/b/c.go"
)) // "c"

fmt.Println(basename("c.d.go"
))   // "c.d"

fmt.Println(basename("abc"
))      // "abc"
```

第一個版本併沒有使用任何庫，全部手工硬編碼實現：

```
gopl.io/ch3/basename1
// basename removes directory components and a .suffix.

// e.g., a => a, a.go => a, a/b/c.go => c, a/b.c.go => b.c

func
 basename(s string
) string
 {
    // Discard last '/' and everything before.

    for
 i := len
(s) - 1
; i >= 0
; i-- {
        if
 s[i] == '/'
 {
            s = s[i+1
:]
            break

        }
    }
    // Preserve everything before last '.'.

    for
 i := len
(s) - 1
; i >= 0
; i-- {
        if
 s[i] == '.'
 {
            s = s[:i]
            break

        }
    }
    return
 s
}
```

簡化個版本使用了strings.LastIndex庫函數：

```
gopl.io/ch3/basename2

func
 basename(s string
) string
 {
    slash := strings.LastIndex(s, "/"
) // -1 if "/" not found

    s = s[slash+1
:]
    if
 dot := strings.LastIndex(s, "."
); dot >= 0
 {
        s = s[:dot]
    }
    return
 s
}
```

path和path/filepath包提供了關於文件路徑名更一般的函數操作。使用斜槓分隔路徑可以在任何操作繫統上工作。斜槓本身不應該用於文件名，但是在其他一些領域可能會用於文件名，例如URL路徑組件。相比之下，path/filepath包則使用操作繫統本身的路徑規則，例如POSIX繫統使用/foo/bar，而Microsoft Windows使用c:\foo\bar等。

讓我們繼續另一個字符串的例子。函數的功能是將一個表示整值的字符串，每隔三個字符插入一個逗號分隔符，例如“12345”處理後成爲“12,345”。這個版本隻適用於整數類型；支持浮點數類型的支持留作練習。

```
gopl.io/ch3/comma

// comma inserts commas in a non-negative decimal integer string.

func
 comma(s string
) string
 {
    n := len
(s)
    if
 n <= 3
 {
        return
 s
    }
    return
 comma(s[:n-3
]) + ","
 + s[n-3
:]
}
```

輸入comma函數的參數是一個字符串。如果輸入字符串的長度小於或等於3的話，則不需要插入逗分隔符。否則，comma函數將在最後三個字符前位置將字符串切割爲兩個兩個子串併插入逗號分隔符，然後通過遞歸調用自身來出前面的子串。

一個字符串是包含的隻讀字節數組，一旦創建，是不可變的。相比之下，一個字節slice的元素則可以自由地脩改。

字符串和字節slice之間可以相互轉換：

```
s := "abc"

b := []byte
(s)
s2 := string
(b)
```

從概念上講，一個[]byte(s)轉換是分配了一個新的字節數組用於保存字符串數據的拷貝，然後引用這個底層的字節數組。編譯器的優化可以避免在一些場景下分配和複製字符串數據，但總的來說需要確保在變量b被脩改的情況下，原始的s字符串也不會改變。將一個字節slice轉到字符串的string(b)操作則是構造一個字符串拷貝，以確保s2字符串是隻讀的。

爲了避免轉換中不必要的內存分配，bytes包和strings同時提供了許多實用函數。下面是strings包中的六個函數：

```
func
 Contains(s, substr string
) bool

func
 Count(s, sep string
) int

func
 Fields(s string
) []string

func
 HasPrefix(s, prefix string
) bool

func
 Index(s, sep string
) int

func
 Join(a []string
, sep string
) string
```

bytes包中也對應的六個函數：

```
func
 Contains(b, subslice []byte
) bool

func
 Count(s, sep []byte
) int

func
 Fields(s []byte
) [][]byte

func
 HasPrefix(s, prefix []byte
) bool

func
 Index(s, sep []byte
) int

func
 Join(s [][]byte
, sep []byte
) []byte
```

它們之間唯一的區別是字符串類型參數被替換成了字節slice類型的參數。

bytes包還提供了Buffer類型用於字節slice的緩存。一個Buffer開始是空的，但是隨着string、byte或[]byte等類型數據的寫入可以動態增長，一個bytes.Buffer變量併不需要處理化，因爲零值也是有效的：

```
gopl.io/ch3/printints

// intsToString is like fmt.Sprint(values) but adds commas.

func
 intsToString(values []int
) string
 {
    var
 buf bytes.Buffer
    buf.WriteByte('['
)
    for
 i, v := range
 values {
        if
 i > 0
 {
            buf.WriteString(", "
)
        }
        fmt.Fprintf(&buf, "%d"
, v)
    }
    buf.WriteByte(']'
)
    return
 buf.String()
}

func
 main() {
    fmt.Println(intsToString([]int
{1
, 2
, 3
})) // "[1, 2, 3]"

}
```

當向bytes.Buffer添加任意字符的UTF8編碼時，最好使用bytes.Buffer的WriteRune方法，但是WriteByte方法對於寫入類似'['和']'等ASCII字符則會更加有效。

bytes.Buffer類型有着很多實用的功能，我們在第七章討論接口時將會涉及到，我們將看看如何將它用作一個I/O的輸入和輸出對象，例如當做Fprintf的io.Writer輸出對象，或者當作io.Reader類型的輸入源對象。

練習 3.10： 編寫一個非遞歸版本的comma函數，使用bytes.Buffer代替字符串鏈接操作。

練習 3.11： 完善comma函數，以支持浮點數處理和一個可選的正負號的處理。

練習 3.12： 編寫一個函數，判斷兩個字符串是否是是相互打亂的，也就是說它們有着相同的字符，但是對應不同的順序。

3.5.5. 字符串和數字的轉換

除了字符串、字符、字節之間的轉換，字符串和數值之間的轉換也比較常見。由strconv包提供這類轉換功能。

將一個整數轉爲字符串，一種方法是用fmt.Sprintf返迴一個格式化的字符串；另一個方法是用strconv.Itoa(“整數到ASCII”)：

```
x := 123

y := fmt.Sprintf("%d"
, x)
fmt.Println(y, strconv.Itoa(x)) // "123 123"
```

FormatInt和FormatUint函數可以用不同的進製來格式化數字：

```
fmt.Println(strconv.FormatInt(int64
(x), 2
)) // "1111011"
```

fmt.Printf函數的%b、%d、%o和%x等參數提供功能往往比strconv包的Format函數方便很多，特別是在需要包含附加額外信息的時候：

```
s := fmt.Sprintf("x=%b"
, x) // "x=1111011"
```

如果要將一個字符串解析爲整數，可以使用strconv包的Atoi或ParseInt函數，還有用於解析無符號整數的ParseUint函數：

```
x, err := strconv.Atoi("123"
)             // x is an int

y, err := strconv.ParseInt("123"
, 10
, 64
) // base 10, up to 64 bits
```

ParseInt函數的第三個參數是用於指定整型數的大小；例如16表示int16，0則表示int。在任何情況下，返迴的結果y總是int64類型，你可以通過強製類型轉換將它轉爲更小的整數類型。

有時候也會使用fmt.Scanf來解析輸入的字符串和數字，特別是當字符串和數字混合在一行的時候，它可以靈活處理不完整或不規則的輸入。

3.6. 常量

常量表達式的值在編譯期計算，而不是在運行期。每種常量的潛在類型都是基礎類型：boolean、string或數字。

一個常量的聲明語句定義了常量的名字，和變量的聲明語法類似，常量的值不可脩改，這樣可以防止在運行期被意外或惡意的脩改。例如，常量比變量更適合用於表達像π之類的數學常數，因爲它們的值不會發生變化：

```
const
 pi = 3.14159
 // approximately; math.Pi is a better approximation
```

和變量聲明一樣，可以批量聲明多個常量；這比較適合聲明一組相關的常量：

```
const
 (
    e  = 2.71828182845904523536028747135266249775724709369995957496696763

    pi = 3.14159265358979323846264338327950288419716939937510582097494459

)
```

所有常量的運算都可以在編譯期完成，這樣可以減少運行時的工作，也方便其他編譯優化。當操作數是常量時，一些運行時的錯誤也可以在編譯時被發現，例如整數除零、字符串索引越界、任何導致無效浮點數的操作等。

常量間的所有算術運算、邏輯運算和比較運算的結果也是常量，對常量的類型轉換操作或以下函數調用都是返迴常量結果：len、cap、real、imag、complex和unsafe.Sizeof（§13.1）。

因爲它們的值是在編譯期就確定的，因此常量可以是構成類型的一部分，例如用於指定數組類型的長度：

```
const
 IPv4Len = 4


// parseIPv4 parses an IPv4 address (d.d.d.d).

func
 parseIPv4(s string
) IP {
    var
 p [IPv4Len]byte

    // ...

}
```

一個常量的聲明也可以包含一個類型和一個值，但是如果沒有顯式指明類型，那麼將從右邊的表達式推斷類型。在下面的代碼中，time.Duration是一個命名類型，底層類型是int64，time.Minute是對應類型的常量。下面聲明的兩個常量都是time.Duration類型，可以通過%T參數打印類型信息：

```
const
 noDelay time.Duration = 0

const
 timeout = 5
 * time.Minute
fmt.Printf("%T %[1]v\n"
, noDelay)     // "time.Duration 0"

fmt.Printf("%T %[1]v\n"
, timeout)     // "time.Duration 5m0s"

fmt.Printf("%T %[1]v\n"
, time.Minute) // "time.Duration 1m0s"
```

如果是批量聲明的常量，除了第一個外其它的常量右邊的初始化表達式都可以省略，如果省略初始化表達式則表示使用前面常量的初始化表達式寫法，對應的常量類型也一樣的。例如：

```
const
 (
    a = 1

    b
    c = 2

    d
)

fmt.Println(a, b, c, d) // "1 1 2 2"
```

如果隻是簡單地複製右邊的常量表達式，其實併沒有太實用的價值。但是它可以帶來其它的特性，那就是iota常量生成器語法。

3.6.1. iota 常量生成器

常量聲明可以使用iota常量生成器初始化，它用於生成一組以相似規則初始化的常量，但是不用每行都寫一遍初始化表達式。在一個const聲明語句中，在第一個聲明的常量所在的行，iota將會被置爲0，然後在每一個有常量聲明的行加一。

下面是來自time包的例子，它首先定義了一個Weekday命名類型，然後爲一週的每天定義了一個常量，從週日0開始。在其它編程語言中，這種類型一般被稱爲枚舉類型。

```
type
 Weekday int


const
 (
    Sunday Weekday = iota

    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

週一將對應0，週一爲1，如此等等。

我們也可以在複雜的常量表達式中使用iota，下面是來自net包的例子，用於給一個無符號整數的最低5bit的每個bit指定一個名字：

```
type
 Flags uint


const
 (
    FlagUp Flags = 1
 << iota
 // is up

    FlagBroadcast            // supports broadcast access capability

    FlagLoopback             // is a loopback interface

    FlagPointToPoint         // belongs to a point-to-point link

    FlagMulticast            // supports multicast access capability

)
```

隨着iota的遞增，每個常量對應表達式1 << iota，是連續的2的冪，分別對應一個bit位置。使用這些常量可以用於測試、設置或清除對應的bit位的值：

```
gopl.io/ch3/netflag

func
 IsUp(v Flags) bool
     { return
 v&FlagUp == FlagUp }
func
 TurnDown(v *Flags)     { *v &^= FlagUp }
func
 SetBroadcast(v *Flags) { *v |= FlagBroadcast }
func
 IsCast(v Flags) bool
   { return
 v&(FlagBroadcast|FlagMulticast) != 0
 }

unc main() {
    var
 v Flags = FlagMulticast | FlagUp
    fmt.Printf("%b %t\n"
, v, IsUp(v)) // "10001 true"

    TurnDown(&v)
    fmt.Printf("%b %t\n"
, v, IsUp(v)) // "10000 false"

    SetBroadcast(&v)
    fmt.Printf("%b %t\n"
, v, IsUp(v))   // "10010 false"

    fmt.Printf("%b %t\n"
, v, IsCast(v)) // "10010 true"

}
```

下面是一個更複雜的例子，每個常量都是1024的冪：

```
const
 (
    _ = 1
 << (10
 * iota
)
    KiB // 1024

    MiB // 1048576

    GiB // 1073741824

    TiB // 1099511627776             (exceeds 1 << 32)

    PiB // 1125899906842624

    EiB // 1152921504606846976

    ZiB // 118059162071741130342    (exceeds 1 << 64)

    YiB // 120892581961462917470617
```

不過iota常量生成規則也有其局限性。例如，它併不能用於産生1000的冪（KB、MB等），因爲Go語言併沒有計算冪的運算符。

練習 3.13： 編寫KB、MB的常量聲明，然後擴展到YB。

3.6.2. 無類型常量

Go語言的常量有個不同尋常之處。雖然一個常量可以有任意有一個確定的基礎類型，例如int或float64，或者是類似time.Duration這樣命名的基礎類型，但是許多常量併沒有一個明確的基礎類型。編譯器爲這些沒有明確的基礎類型的數字常量提供比基礎類型更高精度的算術運算；你可以認爲至少有256bit的運算精度。這里有六種未明確類型的常量類型，分別是無類型的布爾型、無類型的整數、無類型的字符、無類型的浮點數、無類型的複數、無類型的字符串。

通過延遲明確常量的具體類型，無類型的常量不僅可以提供更高的運算精度，而且可以直接用於更多的表達式而不需要顯式的類型轉換。例如，例子中的ZiB和YiB的值已經超出任何Go語言中整數類型能表達的范圍，但是它們依然是合法的常量，而且可以像下面常量表達式依然有效（譯註：YiB/ZiB是在編譯期計算出來的，併且結果常量是1024，是Go語言int變量能有效表示的）：

```
fmt.Println(YiB/ZiB) // "1024"
```

另一個例子，math.Pi無類型的浮點數常量，可以直接用於任意需要浮點數或複數的地方：

```
var
 x float32
 = math.Pi
var
 y float64
 = math.Pi
var
 z complex128
 = math.Pi
```

如果math.Pi被確定爲特定類型，比如float64，那麼結果精度可能會不一樣，同時對於需要float32或complex128類型值的地方則會強製需要一個明確的類型轉換：

```
const
 Pi64 float64
 = math.Pi

var
 x float32
 = float32
(Pi64)
var
 y float64
 = Pi64
var
 z complex128
 = complex128
(Pi64)
```

對於常量面值，不同的寫法可能會對應不同的類型。例如0、0.0、0i和'\u0000'雖然有着相同的常量值，但是它們分別對應無類型的整數、無類型的浮點數、無類型的複數和無類型的字符等不同的常量類型。同樣，true和false也是無類型的布爾類型，字符串面值常量是無類型的字符串類型。

前面説過除法運算符/會根據操作數的類型生成對應類型的結果。因此，不同寫法的常量除法表達式可能對應不同的結果：

```
var
 f float64
 = 212

fmt.Println((f - 32
) * 5
 / 9
)     // "100"; (f - 32) * 5 is a float64

fmt.Println(5
 / 9
 * (f - 32
))     // "0";   5/9 is an untyped integer, 0

fmt.Println(5.0
 / 9.0
 * (f - 32
)) // "100"; 5.0/9.0 is an untyped float
```

隻有常量可以是無類型的。當一個無類型的常量被賦值給一個變量的時候，就像上面的第一行語句，或者是像其餘三個語句中右邊表達式中含有明確類型的值，無類型的常量將會被隱式轉換爲對應的類型，如果轉換合法的話。

```
var
 f float64
 = 3
 + 0i
 // untyped complex -> float64

f = 2
                  // untyped integer -> float64

f = 1e123
              // untyped floating-point -> float64

f = 'a'
                // untyped rune -> float64
```

上面的語句相當於:

```
var
 f float64
 = float64
(3
 + 0i
)
f = float64
(2
)
f = float64
(1e123
)
f = float64
('a'
)
```

無論是隱式或顯式轉換，將一種類型轉換爲另一種類型都要求目標可以表示原始值。對於浮點數和複數，可能會有舍入處理：

```
const
 (
    deadbeef = 0xdeadbeef
 // untyped int with value 3735928559

    a = uint32
(deadbeef)  // uint32 with value 3735928559

    b = float32
(deadbeef) // float32 with value 3735928576 (rounded up)

    c = float64
(deadbeef) // float64 with value 3735928559 (exact)

    d = int32
(deadbeef)   // compile error: constant overflows int32

    e = float64
(1e309
)    // compile error: constant overflows float64

    f = uint
(-1
)          // compile error: constant underflows uint

)
```

對於一個沒有顯式類型的變量聲明語法（包括短變量聲明語法），無類型的常量會被隱式轉爲默認的變量類型，就像下面的例子：

```
i := 0
      // untyped integer;        implicit int(0)

r := '\000'
 // untyped rune;           implicit rune('\000')

f := 0.0
    // untyped floating-point; implicit float64(0.0)

c := 0i
     // untyped complex;        implicit complex128(0i)
```

註意默認類型是規則的：無類型的整數常量默認轉換爲int，對應不確定的內存大小，但是浮點數和複數常量則默認轉換爲float64和complex128。Go語言本身併沒有不確定內存大小的浮點數和複數類型，而且如果不知道浮點數類型的話將很難寫出正確的數值算法。

如果要給變量一個不同的類型，我們必鬚顯式地將無類型的常量轉化爲所需的類型，或給聲明的變量指定明確的類型，像下面例子這樣：

```
var
 i = int8
(0
)
var
 i int8
 = 0
```

當嚐試將這些無類型的常量轉爲一個接口值時（見第7章），這些默認類型將顯得尤爲重要，因爲要靠它們明確接口對應的動態類型。

```
fmt.Printf("%T\n"
, 0
)      // "int"

fmt.Printf("%T\n"
, 0.0
)    // "float64"

fmt.Printf("%T\n"
, 0i
)     // "complex128"

fmt.Printf("%T\n"
, '\000'
) // "int32" (rune)
```

現在我們已經講述了Go語言中全部的基礎數據類型。下一步將演示如何用基礎數據類型組合成數組或結構體等複雜數據類型，然後構建用於解決實際編程問題的數據結構，這將是第四章的討論主題。

第四章 複合數據類型

在第三章我們討論了基本數據類型，它們可以用於構建程序中數據結構，是Go語言的世界的原子。在本章，我們將討論複合數據類型，它是以不同的方式組合基本類型可以構造出來的複合數據類型。我們主要討論四種類型——數組、slice、map和結構體——同時在本章的最後，我們將演示如何使用結構體來解碼和編碼到對應JSON格式的數據，併且通過結合使用模闆來生成HTML頁面。

數組和結構體是聚合類型；它們的值由許多元素或成員字段的值組成。數組是由同構的元素組成——每個數組元素都是完全相同的類型——結構體則是由異構的元素組成的。數組和結構體都是有固定內存大小的數據結構。相比之下，slice和map則是動態的數據結構，它們將根據需要動態增長。

4.1. 數組

數組是一個由固定長度的特定類型元素組成的序列，一個數組可以由零個或多個元素組成。因爲數組的長度是固定的，因此在Go語言中很少直接使用數組。和數組對應的類型是Slice（切片），它是可以增長和收縮動態序列，slice功能也更靈活，但是要理解slice工作原理的話需要先理解數組。

數組的每個元素可以通過索引下標來訪問，索引下標的范圍是從0開始到數組長度減1的位置。內置的len函數將返迴數組中元素的個數。

```
var
 a [3
]int
             // array of 3 integers

fmt.Println(a[0
])        // print the first element

fmt.Println(a[len
(a)-1
]) // print the last element, a[2]


// Print the indices and elements.

for
 i, v := range
 a {
    fmt.Printf("%d %d\n"
, i, v)
}

// Print the elements only.

for
 _, v := range
 a {
    fmt.Printf("%d\n"
, v)
}
```

默認情況下，數組的每個元素都被初始化爲元素類型對應的零值，對於數字類型來說就是0。我們也可以使用數組字面值語法用一組值來初始化數組：

```
var
 q [3
]int
 = [3
]int
{1
, 2
, 3
}
var
 r [3
]int
 = [3
]int
{1
, 2
}
fmt.Println(r[2
]) // "0"
```

在數組字面值中，如果在數組的長度位置出現的是“...”省略號，則表示數組的長度是根據初始化值的個數來計算。因此，上面q數組的定義可以簡化爲

```
q := [...]int
{1
, 2
, 3
}
fmt.Printf("%T\n"
, q) // "[3]int"
```

數組的長度是數組類型的一個組成部分，因此[3]int和[4]int是兩種不同的數組類型。數組的長度必鬚是常量表達式，因爲數組的長度需要在編譯階段確定。

```
q := [3
]int
{1
, 2
, 3
}
q = [4
]int
{1
, 2
, 3
, 4
} // compile error: cannot assign [4]int to [3]int
```

我們將會發現，數組、slice、map和結構體字面值的寫法都很相似。上面的形式是直接提供順序初始化值序列，但是也可以指定一個索引和對應值列表的方式初始化，就像下面這樣：

```
type
 Currency int


const
 (
    USD Currency = iota
 // 美元

    EUR                 // 歐元

    GBP                 // 英鎊

    RMB                 // 人民幣

)

symbol := [...]string
{USD: "$"
, EUR: "€"
, GBP: "£"
, RMB: "¥"
}

fmt.Println(RMB, symbol[RMB]) // "3 ¥"
```

在這種形式的數組字面值形式中，初始化索引的順序是無關緊要的，而且沒用到的索引可以省略，和前面提到的規則一樣，未指定初始值的元素將用零值初始化。例如，

```
r := [...]int
{99
: -1
}
```

定義了一個含有100個元素的數組r，最後一個元素被初始化爲-1，其它元素都是用0初始化。

如果一個數組的元素類型是可以相互比較的，那麼數組類型也是可以相互比較的，這時候我們可以直接通過==比較運算符來比較兩個數組，隻有當兩個數組的所有元素都是相等的時候數組才是相等的。不相等比較運算符!=遵循同樣的規則。

```
a := [2
]int
{1
, 2
}
b := [...]int
{1
, 2
}
c := [2
]int
{1
, 3
}
fmt.Println(a == b, a == c, b == c) // "true false false"

d := [3
]int
{1
, 2
}
fmt.Println(a == d) // compile error: cannot compare [2]int == [3]int
```

作爲一個眞實的例子，crypto/sha256包的Sum256函數對一個任意的字節slice類型的數據生成一個對應的消息摘要。消息摘要有256bit大小，因此對應[32]byte數組類型。如果兩個消息摘要是相同的，那麼可以認爲兩個消息本身也是相同（譯註：理論上有HASH碰撞的情況，但是實際應用可以基本忽略）；如果消息摘要不同，那麼消息本身必然也是不同的。下面的例子用SHA256算法分別生成“x”和“X”兩個信息的摘要：

```
gopl.io/ch4/sha256

import
 "crypto/sha256"



func
 main() {
    c1 := sha256.Sum256([]byte
("x"
))
    c2 := sha256.Sum256([]byte
("X"
))
    fmt.Printf("%x\n%x\n%t\n%T\n"
, c1, c2, c1 == c2, c1)
    // Output:

    // 2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881

    // 4b68ab3847feda7d6c62c1fbcbeebfa35eab7351ed5e78f4ddadea5df64b8015

    // false

    // [32]uint8

}
```

上面例子中，兩個消息雖然隻有一個字符的差異，但是生成的消息摘要則幾乎有一半的bit位是不相同的。需要註意Printf函數的%x副詞參數，它用於指定以十六進製的格式打印數組或slice全部的元素，%t副詞參數是用於打印布爾型數據，%T副詞參數是用於顯示一個值對應的數據類型。

當調用一個函數的時候，函數的每個調用參數將會被賦值給函數內部的參數變量，所以函數參數變量接收的是一個複製的副本，併不是原始調用的變量。因爲函數參數傳遞的機製導致傳遞大的數組類型將是低效的，併且對數組參數的任何的脩改都是發生在複製的數組上，併不能直接脩改調用時原始的數組變量。在這個方面，Go語言對待數組的方式和其它很多編程語言不同，其它編程語言可能會隱式地將數組作爲引用或指針對象傳入被調用的函數。

當然，我們可以顯式地傳入一個數組指針，那樣的話函數通過指針對數組的任何脩改都可以直接反饋到調用者。下面的函數用於給[32]byte類型的數組清零：

```
func
 zero(ptr *[32
]byte
) {
    for
 i := range
 ptr {
        ptr[i] = 0

    }
}
```

其實數組字面值[32]byte{}就可以生成一個32字節的數組。而且每個數組的元素都是零值初始化，也就是0。因此，我們可以將上面的zero函數寫的更簡潔一點：

```
func
 zero(ptr *[32
]byte
) {
    *ptr = [32
]byte
{}
}
```

雖然通過指針來傳遞數組參數是高效的，而且也允許在函數內部脩改數組的值，但是數組依然是殭化的類型，因爲數組的類型包含了殭化的長度信息。上面的zero函數併不能接收指向[16]byte類型數組的指針，而且也沒有任何添加或刪除數組元素的方法。由於這些原因，除了像SHA256這類需要處理特定大小數組的特例外，數組依然很少用作函數參數；相反，我們一般使用slice來替代數組。

練習 4.1： 編寫一個函數，計算兩個SHA256哈希碼中不同bit的數目。（參考2.6.2節的PopCount函數。）

練習 4.2： 編寫一個程序，默認打印標準輸入的以SHA256哈希碼，也可以通過命令行標準參數選擇SHA384或SHA512哈希算法。

4.2. Slice

Slice（切片）代表變長的序列，序列中每個元素都有相同的類型。一個slice類型一般寫作[]T，其中T代表slice中元素的類型；slice的語法和數組很像，隻是沒有固定長度而已。

數組和slice之間有着緊密的聯繫。一個slice是一個輕量級的數據結構，提供了訪問數組子序列（或者全部）元素的功能，而且slice的底層確實引用一個數組對象。一個slice由三個部分構成：指針、長度和容量。指針指向第一個slice元素對應的底層數組元素的地址，要註意的是slice的第一個元素併不一定就是數組的第一個元素。長度對應slice中元素的數目；長度不能超過容量，容量一般是從slice的開始位置到底層數據的結尾位置。內置的len和cap函數分別返迴slice的長度和容量。

多個slice之間可以共享底層的數據，併且引用的數組部分區間可能重疊。圖4.1顯示了表示一年中每個月份名字的字符串數組，還有重疊引用了該數組的兩個slice。數組這樣定義

```
months := [...]string
{1
: "January"
, /* ... */
, 12
: "December"
}
```

因此一月份是months[1]，十二月份是months[12]。通常，數組的第一個元素從索引0開始，但是月份一般是從1開始的，因此我們聲明數組時直接第0個元素，第0個元素會被自動初始化爲空字符串。

slice的切片操作s[i:j]，其中0 ≤ i≤ j≤ cap(s)，用於創建一個新的slice，引用s的從第i個元素開始到第j-1個元素的子序列。新的slice將隻有j-i個元素。如果i位置的索引被省略的話將使用0代替，如果j位置的索引被省略的話將使用len(s)代替。因此，months[1:13]切片操作將引用全部有效的月份，和months[1:]操作等價；months[:]切片操作則是引用整個數組。讓我們分別定義表示第二季度和北方夏天月份的slice，它們有重疊部分：

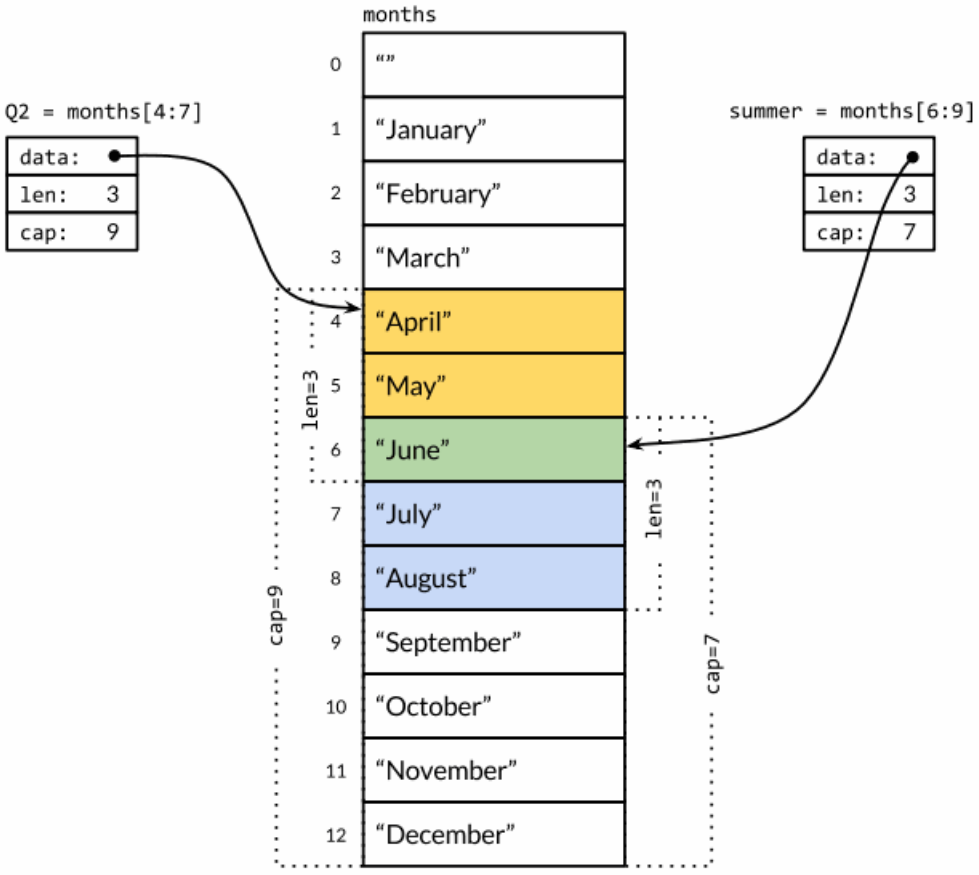


Figure 4.1. Two overlapping slices of an array of months.

```
Q2 := months[4
:7
]
summer := months[6
:9
]
fmt.Println(Q2)     // ["April" "May" "June"]

fmt.Println(summer) // ["June" "July" "August"]
```

兩個slice都包含了六月份，下面的代碼是一個包含相同月份的測試（性能較低）：

```
for
 _, s := range
 summer {
    for
 _, q := range
 Q2 {
        if
 s == q {
            fmt.Printf("%s appears in both\n"
, s)
        }
    }
}
```

如果切片操作超出cap(s)的上限將導致一個panic異常，但是超出len(s)則是意味着擴展了slice，因爲新slice的長度會變大：

```
fmt.Println(summer[:20
]) // panic: out of range


endlessSummer := summer[:5
] // extend a slice (within capacity)

fmt.Println(endlessSummer)  // "[June July August September October]"
```

另外，字符串的切片操作和[]byte字節類型切片的切片操作是類似的。它們都寫作x[m:n]，併且都是返迴一個原始字節繫列的子序列，底層都是共享之前的底層數組，因此切片操作對應常量時間複雜度。x[m:n]切片操作對於字符串則生成一個新字符串，如果x是[]byte的話則生成一個新的[]byte。

因爲slice值包含指向第一個slice元素的指針，因此向函數傳遞slice將允許在函數內部脩改底層數組的元素。換句話說，複製一個slice隻是對底層的數組創建了一個新的slice別名（§2.3.2）。下面的reverse函數在原內存空間將[]int類型的slice反轉，而且它可以用於任意長度的slice。

```
gopl.io/ch4/rev

// reverse reverses a slice of ints in place.

func
 reverse(s []int
) {
    for
 i, j := 0
, len
(s)-1
; i < j; i, j = i+1
, j-1
 {
        s[i], s[j] = s[j], s[i]
    }
}
```

這里我們反轉數組的應用：

```
a := [...]int
{0
, 1
, 2
, 3
, 4
, 5
}
reverse(a[:])
fmt.Println(a) // "[5 4 3 2 1 0]"
```

一種將slice元素循環向左鏇轉n個元素的方法是三次調用reverse反轉函數，第一次是反轉開頭的n個元素，然後是反轉剩下的元素，最後是反轉整個slice的元素。（如果是向右循環鏇轉，則將第三個函數調用移到第一個調用位置就可以了。）

```
s := []int
{0
, 1
, 2
, 3
, 4
, 5
}
// Rotate s left by two positions.

reverse(s[:2
])
reverse(s[2
:])
reverse(s)
fmt.Println(s) // "[2 3 4 5 0 1]"
```

要註意的是slice類型的變量s和數組類型的變量a的初始化語法的差異。slice和數組的字面值語法很類似，它們都是用花括弧包含一繫列的初始化元素，但是對於slice併沒有指明序列的長度。這會隱式地創建一個合適大小的數組，然後slice的指針指向底層的數組。就像數組字面值一樣，slice的字面值也可以按順序指定初始化值序列，或者是通過索引和元素值指定，或者的兩種風格的混合語法初始化。

和數組不同的是，slice之間不能比較，因此我們不能使用==操作符來判斷兩個slice是否含有全部相等元素。不過標準庫提供了高度優化的bytes.Equal函數來判斷兩個字節型slice是否相等（[]byte），但是對於其他類型的slice，我們必鬚自己展開每個元素進行比較：

```
func
 equal(x, y []string
) bool
 {
    if
 len
(x) != len
(y) {
        return
 false

    }
    for
 i := range
 x {
        if
 x[i] != y[i] {
            return
 false

        }
    }
    return
 true

}
```

上面關於兩個slice的深度相等測試，運行的時間併不比支持==操作的數組或字符串更多，但是爲何slice不直接支持比較運算符呢？這方面有兩個原因。第一個原因，一個slice的元素是間接引用的，一個slice甚至可以包含自身。雖然有很多辦法處理這種情形，但是沒有一個是簡單有效的。

第二個原因，因爲slice的元素是間接引用的，一個固定值的slice在不同的時間可能包含不同的元素，因爲底層數組的元素可能會被脩改。併且Go語言中map等哈希表之類的數據結構的key隻做簡單的淺拷貝，它要求在整個聲明週期中相等的key必鬚對相同的元素。對於像指針或chan之類的引用類型，==相等測試可以判斷兩個是否是引用相同的對象。一個針對slice的淺相等測試的==操作符可能是有一定用處的，也能臨時解決map類型的key問題，但是slice和數組不同的相等測試行爲會讓人睏惑。因此，安全的做飯是直接禁止slice之間的比較操作。

slice唯一合法的比較操作是和nil比較，例如：

```
if
 summer == nil
 { /* ... */
 }
```

一個零值的slice等於nil。一個nil值的slice併沒有底層數組。一個nil值的slice的長度和容量都是0，但是也有非nil值的slice的長度和容量也是0的，例如[]int{}或make([]int, 3)[3:]。與任意類型的nil值一樣，我們可以用[]int(nil)類型轉換表達式來生成一個對應類型slice的nil值。

```
var
 s []int
    // len(s) == 0, s == nil

s = nil
        // len(s) == 0, s == nil

s = []int
(nil
) // len(s) == 0, s == nil

s = []int
{}    // len(s) == 0, s != nil
```

如果你需要測試一個slice是否是空的，使用len(s) == 0來判斷，而不應該用s == nil來判斷。除了和nil相等比較外，一個nil值的slice的行爲和其它任意0產長度的slice一樣；例如reverse(nil)也是安全的。除了文檔已經明確說明的地方，所有的Go語言函數應該以相同的方式對待nil值的slice和0長度的slice。

內置的make函數創建一個指定元素類型、長度和容量的slice。容量部分可以省略，在這種情況下，容量將等於長度。

```
make
([]T, len
)
make
([]T, len
, cap
) // same as make([]T, cap)[:len]
```

在底層，make創建了一個匿名的數組變量，然後返迴一個slice；隻有通過返迴的slice才能引用底層匿名的數組變量。在第一種語句中，slice是整個數組的view。在第二個語句中，slice隻引用了底層數組的前len個元素，但是容量將包含整個的數組。額外的元素是留給未來的增長用的。

4.2.1. append函數

內置的append函數用於向slice追加元素：

```
var
 runes []rune

for
 _, r := range
 "Hello, 世界"
 {
    runes = append
(runes, r)
}
fmt.Printf("%q\n"
, runes) // "['H' 'e' 'l' 'l' 'o' ',' ' ' '世' '界']"
```

在循環中使用append函數構建一個由九個rune字符構成的slice，當然對應這個特殊的問題我們可以通過Go語言內置的[]rune("Hello, 世界")轉換操作完成。

append函數對於理解slice底層是如何工作的非常重要，所以讓我們仔細查看究竟是發生了什麼。下面是第一個版本的appendInt函數，專門用於處理[]int類型的slice：

```
gopl.io/ch4/append


func
 appendInt(x []int
, y int
) []int
 {
    var
 z []int

    zlen := len
(x) + 1

    if
 zlen <= cap
(x) {
        // There is room to grow.  Extend the slice.

        z = x[:zlen]
    } else
 {
        // There is insufficient space.  Allocate a new array.

        // Grow by doubling, for amortized linear complexity.

        zcap := zlen
        if
 zcap < 2
*len
(x) {
            zcap = 2
 * len
(x)
        }
        z = make
([]int
, zlen, zcap)
        copy
(z, x) // a built-in function; see text

    }
    z[len
(x)] = y
    return
 z
}
```

每次調用appendInt函數，必鬚先檢測slice底層數組是否有足夠的容量來保存新添加的元素。如果有足夠空間的話，直接擴展slice（依然在原有的底層數組之上），將新添加的y元素複製到新擴展的空間，併返迴slice。因此，輸入的x和輸出的z共享相同的底層數組。

如果沒有足夠的增長空間的話，appendInt函數則會先分配一個足夠大的slice用於保存新的結果，先將輸入的x複製到新的空間，然後添加y元素。結果z和輸入的x引用的將是不同的底層數組。

雖然通過循環複製元素更直接，不過內置的copy函數可以方便地將一個slice複製另一個相同類型的slice。copy函數的第一個參數是要複製的目標slice，第二個參數是源slice，目標和源的位置順序和 `dst = src` 賦值語句是一致的。兩個slice可以共享同一個底層數組，甚至有重疊也沒有問題。copy函數將返迴成功複製的元素的個數（我們這里沒有用到），等於兩個slice中較小的長度，所以我們不用擔心覆蓋會超出目標slice的范圍。

爲了提高內存使用效率，新分配的數組一般略大於保存x和y所需要的最低大小。通過在每次擴展數組時直接將長度翻倍從而避免了多次內存分配，也確保了添加單個元素操的平均時間是一個常數時間。這個程序演示了效果：

```
func
 main() {
    var
 x, y []int

    for
 i := 0
; i < 10
; i++ {
        y = appendInt(x, i)
        fmt.Printf("%d cap=%d\t%v\n"
, i, cap
(y), y)
        x = y
    }
}
```

每一次容量的變化都會導致重新分配內存和copy操作：

```
0  cap=1    [0]
1  cap=2    [0 1]
2  cap=4    [0 1 2]
3  cap=4    [0 1 2 3]
4  cap=8    [0 1 2 3 4]
5  cap=8    [0 1 2 3 4 5]
6  cap=8    [0 1 2 3 4 5 6]
7  cap=8    [0 1 2 3 4 5 6 7]
8  cap=16   [0 1 2 3 4 5 6 7 8]
9  cap=16   [0 1 2 3 4 5 6 7 8 9]
```

讓我們仔細查看i=3次的迭代。當時x包含了[0 1 2]三個元素，但是容量是4，因此可以簡單將新的元素添加到末尾，不需要新的內存分配。然後新的y的長度和容量都是4，併且和x引用着相同的底層數組，如圖4.2所示。

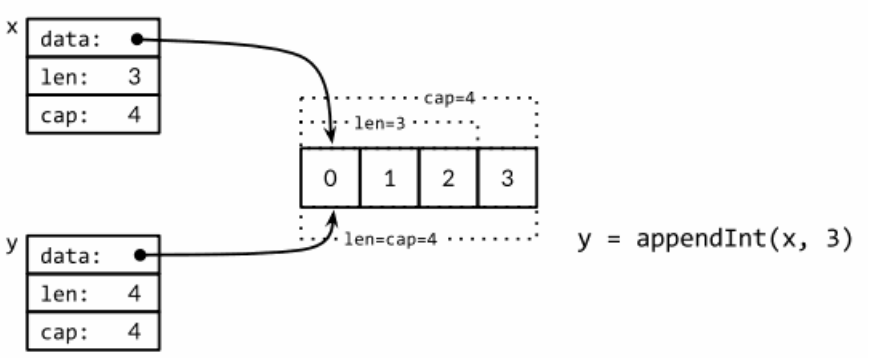


Figure 4.2. Appending with room to grow.

在下一次迭代時i=4，現在沒有新的空餘的空間了，因此appendInt函數分配一個容量爲8的底層數組，將x的4個元素[0 1 2 3]複製到新空間的開頭，然後添加新的元素i，新元素的值是4。新的y的長度是5，容量是8；後面有3個空閒的位置，三次迭代都不需要分配新的空間。當前迭代中，y和x是對應不同底層數組的view。這次操作如圖4.3所示。

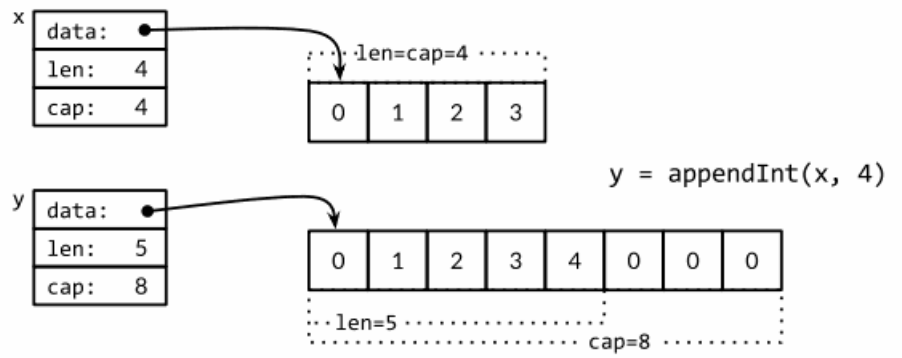


Figure 4.3. Appending without room to grow.

內置的append函數可能使用比appendInt更複雜的內存擴展策略。因此，通常我們併不知道append調用是否導致了內存的重新分配，因此我們也不能確認新的slice和原始的slice是否引用的是相同的底層數組空間。同樣，我們不能確認在原先的slice上的操作是否會影響到新的slice。因此，通常是將append返迴的結果直接賦值給輸入的slice變量：

```
runes = append
(runes, r)
```

更新slice變量不僅對調用append函數是必要的，實際上對應任何可能導致長度、容量或底層數組變化的操作都是必要的。要正確地使用slice，需要記住盡管底層數組的元素是間接訪問的，但是slice對應結構體本身的指針、長度和容量部分是直接訪問的。要更新這些信息需要像上面例子那樣一個顯式的賦值操作。從這個角度看，slice併不是一個純粹的引用類型，它實際上是一個類似下面結構體的聚合類型：

```
type
 IntSlice struct
 {
    ptr      *int

    len
, cap
 int

}
```

我們的appendInt函數每次隻能向slice追加一個元素，但是內置的append函數則可以追加多個元素，甚至追加一個slice。

```
var
 x []int

x = append
(x, 1
)
x = append
(x, 2
, 3
)
x = append
(x, 4
, 5
, 6
)
x = append
(x, x...) // append the slice x

fmt.Println(x)      // "[1 2 3 4 5 6 1 2 3 4 5 6]"
```

通過下面的小脩改，我們可以可以達到append函數類似的功能。其中在appendInt函數參數中的最後的“...”省略號表示接收變長的參數爲slice。我們將在5.7節詳細解釋這個特性。

```
func
 appendInt(x []int
, y ...int
) []int
 {
    var
 z []int

    zlen := len
(x) + len
(y)
    // ...expand z to at least zlen...

    copy
(z[len
(x):], y)
    return
 z
}
```

爲了避免重複，和前面相同的代碼併沒有顯示。

4.2.2. Slice內存技巧

讓我們看看更多的例子，比如鏇轉slice、反轉slice或在slice原有內存空間脩改元素。給定一個字符串列表，下面的nonempty函數將在原有slice內存空間之上返迴不包含空字符串的列表：

```
gopl.io/ch4/nonempty

// Nonempty is an example of an in-place slice algorithm.

package
 main

import
 "fmt"


// nonempty returns a slice holding only the non-empty strings.

// The underlying array is modified during the call.

func
 nonempty(strings []string
) []string
 {
    i := 0

    for
 _, s := range
 strings {
        if
 s != ""
 {
            strings[i] = s
            i++
        }
    }
    return
 strings[:i]
}
```

比較微妙的地方是，輸入的slice和輸出的slice共享一個底層數組。這可以避免分配另一個數組，不過原來的數據將可能會被覆蓋，正如下面兩個打印語句看到的那樣：

```
data := []string
{"one"
, ""
, "three"
}
fmt.Printf("%q\n"
, nonempty(data)) // `["one" "three"]`

fmt.Printf("%q\n"
, data)           // `["one" "three" "three"]`
```

因此我們通常會這樣使用nonempty函數： `data = nonempty(data)` 。

nonempty函數也可以使用append函數實現：

```
func
 nonempty2(strings []string
) []string
 {
    out := strings[:0
] // zero-length slice of original

    for
 _, s := range
 strings {
        if
 s != ""
 {
            out = append
(out, s)
        }
    }
    return
 out
}
```

無論如何實現，以這種方式重用一個slice一般都要求最多爲每個輸入值產生一個輸出值，事實上很多這類算法都是用來過濾或合併序列中相鄰的元素。這種slice用法是比較複雜的技巧，雖然使用到了slice的一些技巧，但是對於某些場合是比較清晰和有效的。

一個slice可以用來模擬一個stack。最初給定的空slice對應一個空的stack，然後可以使用append函數將新的值壓入stack：

```
stack = append
(stack, v) // push v
```

stack的頂部位置對應slice的最後一個元素：

```
top := stack[len
(stack)-1
] // top of stack
```

通過收縮stack可以彈出棧頂的元素

```
stack = stack[:len
(stack)-1
] // pop
```

要删除slice中間的某個元素併保存原有的元素順序，可以通過內置的copy函數將後面的子slice向前依次移動一位完成：

```
func
 remove(slice []int
, i int
) []int
 {
    copy
(slice[i:], slice[i+1
:])
    return
 slice[:len
(slice)-1
]
}

func
 main() {
    s := []int
{5
, 6
, 7
, 8
, 9
}
    fmt.Println(remove(s, 2
)) // "[5 6 8 9]"

}
```

如果删除元素後不用保持原來順序的話，我們可以簡單的用最後一個元素覆蓋被删除的元素：

```
func
 remove(slice []int
, i int
) []int
 {
    slice[i] = slice[len
(slice)-1
]
    return
 slice[:len
(slice)-1
]
}

func
 main() {
    s := []int
{5
, 6
, 7
, 8
, 9
}
    fmt.Println(remove(s, 2
)) // "[5 6 9 8]

}
```

練習 4.3： 重寫reverse函數，使用數組指針代替slice。

練習 4.4： 編寫一個rotate函數，通過一次循環完成鏇轉。

練習 4.5： 寫一個函數在原地完成消除[]string中相鄰重複的字符串的操作。

練習 4.6： 編寫一個函數，原地將一個UTF-8編碼的[]byte類型的slice中相鄰的空格（參考unicode.IsSpace）替換成一個空格返迴

練習 4.7： 脩改reverse函數用於原地反轉UTF-8編碼的[]byte。是否可以不用分配額外的內存？

4.3. Map

哈希表是一種巧妙併且實用的數據結構。它是一個無序的key/value對的集合，其中所有的key都是不同的，然後通過給定的key可以在常數時間複雜度內檢索、更新或刪除對應的value。

在Go語言中，一個map就是一個哈希表的引用，map類型可以寫爲map[K]V，其中K和V分別對應key和value。map中所有的key都有相同的類型，所以的value也有着相同的類型，但是key和value之間可以是不同的數據類型。其中K對應的key必鬚是支持==比較運算符的數據類型，所以map可以通過測試key是否相等來判斷是否已經存在。雖然浮點數類型也是支持相等運算符比較的，但是將浮點數用做key類型則是一個壞的想法，正如第三章提到的，最壞的情況是可能出現的NaN和任何浮點數都不相等。對於V對應的value數據類型則沒有任何的限製。

內置的make函數可以創建一個map：

```
ages := make
(map
[string
]int
) // mapping from strings to ints
```

我們也可以用map字面值的語法創建map，同時還可以指定一些最初的key/value：

```
ages := map
[string
]int
{
    "alice"
:   31
,
    "charlie"
: 34
,
}
```

這相當於

```
ages := make
(map
[string
]int
)
ages["alice"
] = 31

ages["charlie"
] = 34
```

因此，另一種創建空的map的表達式是 `map[string]int{}` 。

Map中的元素通過key對應的下標語法訪問：

```
ages["alice"
] = 32

fmt.Println(ages["alice"
]) // "32"
```

使用內置的delete函數可以刪除元素：

```
delete
(ages, "alice"
) // remove element ages["alice"]
```

所有這些操作是安全的，即使這些元素不在map中也沒有關繫；如果一個查找失敗將返迴value類型對應的零值，例如，即使map中不存在“bob”下面的代碼也可以正常工作，因爲ages["bob"]失敗時將返迴0。

```
ages["bob"
] = ages["bob"
] + 1
 // happy birthday!
```

而且 `x += y` 和 `x++` 等簡短賦值語法也可以用在map上，所以上面的代碼可以改寫成

```
ages["bob"
] += 1
```

更簡單的寫法

```
ages["bob"
]++
```

但是map中的元素併不是一個變量，因此我們不能對map的元素進行取址操作：

```
_ = &ages["bob"
] // compile error: cannot take address of map element
```

禁止對map元素取址的原因是map可能隨着元素數量的增長而重新分配更大的內存空間，從而可能導致之前的地址無效。

要想遍歷map中全部的key/value對的話，可以使用range風格的for循環實現，和之前的slice遍歷語法類似。下面的迭代語句將在每次迭代時設置name和age變量，它們對應下一個鍵/值對：

```
for
 name, age := range
 ages {
    fmt.Printf("%s\t%d\n"
, name, age)
}
```

Map的迭代順序是不確定的，併且不同的哈希函數實現可能導致不同的遍歷順序。在實踐中，遍歷的順序是隨機的，每一次遍歷的順序都不相同。這是故意的，每次都使用隨機的遍歷順序可以強製要求程序不會依賴具體的哈希函數實現。如果要按順序遍歷key/value對，我們必鬚顯式地對key進行排序，可以使用sort包的Strings函數對字符串slice進行排序。下面是常見的處理方式：

```
import
 "sort"


var
 names []string

for
 name := range
 ages {
    names = append
(names, name)
}
sort.Strings(names)
for
 _, name := range
 names {
    fmt.Printf("%s\t%d\n"
, name, ages[name])
}
```

因爲我們一開始就知道names的最終大小，因此給slice分配一個合適的大小將會更有效。下面的代碼創建了一個空的slice，但是slice的容量剛好可以放下map中全部的key：

```
names := make
([]string
, 0
, len
(ages))
```

在上面的第一個range循環中，我們隻關心map中的key，所以我們忽略了第二個循環變量。在第二個循環中，我們隻關心names中的名字，所以我們使用“_”空白標識符來忽略第一個循環變量，也就是迭代slice時的索引。

map類型的零值是nil，也就是沒有引用任何哈希表。

```
var
 ages map
[string
]int

fmt.Println(ages == nil
)    // "true"

fmt.Println(len
(ages) == 0
) // "true"
```

map上的大部分操作，包括查找、删除、len和range循環都可以安全工作在nil值的map上，它們的行爲和一個空的map類似。但是向一個nil值的map存入元素將導致一個panic異常：

```
ages["carol"
] = 21
 // panic: assignment to entry in nil map
```

在向map存數據前必鬚先創建map。

通過key作爲索引下標來訪問map將産生一個value。如果key在map中是存在的，那麽將得到與key對應的value；如果key不存在，那麽將得到value對應類型的零值，正如我們前面看到的ages["bob"]那樣。這個規則很實用，但是有時候可能需要知道對應的元素是否眞的是在map之中。例如，如果元素類型是一個數字，你可以需要區分一個已經存在的0，和不存在而返迴零值的0，可以像下面這樣測試：

```
age, ok := ages["bob"
]
if
 !ok { /* "bob" is not a key in this map; age == 0. */
 }
```

你會經常看到將這兩個結合起來使用，像這樣：

```
if
 age, ok := ages["bob"
]; !ok { /* ... */
 }
```

在這種場景下，map的下標語法將產生兩個值；第二個是一個布爾值，用於報告元素是否眞的存在。布爾變量一般命名爲ok，特別適合馬上用於if條件判斷部分。

和slice一樣，map之間也不能進行相等比較；唯一的例外是和nil進行比較。要判斷兩個map是否包含相同的key和value，我們必鬚通過一個循環實現：

```
func
 equal(x, y map
[string
]int
) bool
 {
    if
 len
(x) != len
(y) {
        return
 false

    }
    for
 k, xv := range
 x {
        if
 yv, ok := y[k]; !ok || yv != xv {
            return
 false

        }
    }
    return
 true

}
```

要註意我們是如何用!ok來區分元素缺失和元素不同的。我們不能簡單地用xv != y[k]判斷，那樣會導致在判斷下面兩個map時產生錯誤的結果：

```
// True if equal is written incorrectly.

equal(map
[string
]int
{"A"
: 0
}, map
[string
]int
{"B"
: 42
})
```

Go語言中併沒有提供一個set類型，但是map中的key也是不相同的，可以用map實現類似set的功能。爲了説明這一點，下面的dedup程序讀取多行輸入，但是隻打印第一次出現的行。（它是1.3節中出現的dup程序的變體。）dedup程序通過map來表示所有的輸入行所對應的set集合，以確保已經在集合存在的行不會被重複打印。

```
gopl.io/ch4/dedup

func
 main() {
    seen := make
(map
[string
]bool
) // a set of strings

    input := bufio.NewScanner(os.Stdin)
    for
 input.Scan() {
        line := input.Text()
        if
 !seen[line] {
            seen[line] = true

            fmt.Println(line)
        }
    }

    if
 err := input.Err(); err != nil
 {
        fmt.Fprintf(os.Stderr, "dedup: %v\n"
, err)
        os.Exit(1
)
    }
}
```

Go程序員將這種忽略value的map當作一個字符串集合，併非所有 `map[string]bool` 類型value都是無關緊要的；有一些則可能會同時包含tue和false的值。

有時候我們需要一個map或set的key是slice類型，但是map的key必鬚是可比較的類型，但是slice併不滿足這個條件。不過，我們可以通過兩個步驟繞過這個限製。第一步，定義一個輔助函數k，將slice轉爲map對應的string類型的key，確保隻有x和y相等時k(x) == k(y)才成立。然後創建一個key爲string類型的map，在每次對map操作時先用k輔助函數將slice轉化爲string類型。

下面的例子演示了如何使用map來記録提交相同的字符串列表的次數。它使用了fmt.Sprintf函數將字符串列表轉換爲一個字符串以用於map的key，通過%q參數忠實地記録每個字符串元素的信息：

```
var
 m = make
(map
[string
]int
)

func
 k(list []string
) string
 { return
 fmt.Sprintf("%q"
, list) }

func
 Add(list []string
)       { m[k(list)]++ }
func
 Count(list []string
) int
 { return
 m[k(list)] }
```

使用同樣的技術可以處理任何不可比較的key類型，而不僅僅是slice類型。這種技術對於想使用自定義key比較函數的時候也很有用，例如在比較字符串的時候忽略大小寫。同時，輔助函數k(x)也不一定是字符串類型，它可以返迴任何可比較的類型，例如整數、數組或結構體等。

這是map的另一個例子，下面的程序用於統計輸入中每個Unicode碼點出現的次數。雖然Unicode全部碼點的數量鉅大，但是出現在特定文檔中的字符種類併沒有多少，使用map可以用比較自然的方式來跟蹤那些出現過字符的次數。

```
gopl.io/ch4/charcount

// Charcount computes counts of Unicode characters.

package
 main

import
 (
    "bufio"

    "fmt"

    "io"

    "os"

    "unicode"

    "unicode/utf8"
```

```
)

func
 main() {
    counts := make
(map
[rune
]int
)    // counts of Unicode characters

    var
 utflen [utf8.UTFMax + 1
]int
 // count of lengths of UTF-8 encodings

    invalid := 0
                    // count of invalid UTF-8 characters


    in := bufio.NewReader(os.Stdin)
    for
 {
        r, n, err := in.ReadRune() // returns rune, nbytes, error

        if
 err == io.EOF {
            break

        }
        if
 err != nil
 {
            fmt.Fprintf(os.Stderr, "charcount: %v\n"
, err)
            os.Exit(1
)
        }
        if
 r == unicode.ReplacementChar && n == 1
 {
            invalid++
            continue

        }
        counts[r]++
        utflen[n]++
    }
    fmt.Printf("rune\tcount\n"
)
    for
 c, n := range
```

```
 counts {
        fmt.Printf("%q\t%d\n"
, c, n)
    }
    fmt.Print("\nlen\tcount\n"
)
    for
 i, n := range
 utflen {
        if
 i > 0
 {
            fmt.Printf("%d\t%d\n"
, i, n)
        }
    }
    if
 invalid > 0
 {
        fmt.Printf("\n%d invalid UTF-8 characters\n"
, invalid)
    }
}
```

ReadRune方法執行UTF-8解碼併返迴三個值：解碼的rune字符的值，字符UTF-8編碼後的長度，和一個錯誤值。我們可預期的錯誤值隻有對應文件結尾的io.EOF。如果輸入的是無效的UTF-8編碼的字符，返迴的將是unicode.ReplacementChar表示無效字符，併且編碼長度是1。

charcount程序同時打印不同UTF-8編碼長度的字符數目。對此，map併不是一個合適的數據結構；因爲UTF-8編碼的長度總是從1到utf8.UTFMax（最大是4個字節），使用數組將更有效。

作爲一個實驗，我們用charcount程序對英文版原稿的字符進行了統計。雖然大部分是英語，但是也有一些非ASCII字符。下面是排名前10的非ASCII字符：

```
°  27  世  15  界  14  é  13  ×  10  ≤  5  ×  5  国  4  4  □  3
```

下面是不同UTF-8編碼長度的字符的數目：

```
len count
1   765391
2   60
3   70
4   0
```

Map的value類型也可以是一個聚合類型，比如是一個map或slice。在下面的代碼中，圖graph的key類型是一個字符串，value類型map[string]bool代表一個字符串集合。從概念上將，graph將一個字符串類型的key映射到一組相關的字符串集合，它們指向新的graph的key。

```
gopl.io/ch4/graph

var
 graph = make
(map
[string
]map
[string
]bool
)

func
 addEdge(from, to string
) {
    edges := graph[from]
    if
 edges == nil
 {
        edges = make
(map
[string
]bool
)
        graph[from] = edges
    }
    edges[to] = true

}

func
 hasEdge(from, to string
) bool
 {
    return
 graph[from][to]
}
```

其中addEdge函數惰性初始化map是一個慣用方式，也就是説在每個值首次作爲key時才初始化。addEdge函數顯示了如何讓map的零值也能正常工作；即使from到to的邊不存在，graph[from][to]依然可以返迴一個有意義的結果。

練習 4.8： 脩改charcount程序，使用unicode.IsLetter等相關的函數，統計字母、數字等Unicode中不同的字符類別。

練習 4.9： 編寫一個程序wordfreq程序，報告輸入文本中每個單詞出現的頻率。在第一次調用Scan前先調用input.Split(bufio.ScanWords)函數，這樣可以按單詞而不是按行輸入。

4.4. 結構體

結構體是一種聚合的數據類型，是由零個或多個任意類型的值聚合成的實體。每個值稱爲結構體的成員。用結構體的經典案例處理公司的員工信息，每個員工信息包含一個唯一的員工編號、員工的名字、家庭住址、出生日期、工作崗位、薪資、上級領導等等。所有的這些信息都需要綁定到一個實體中，可以作爲一個整體單元被複製，作爲函數的參數或返迴值，或者是被存儲到數組中，等等。

下面兩個語句聲明了一個叫Employee的命名的結構體類型，併且聲明了一個Employee類型的變量dilbert：

```
type
 Employee struct
 {
    ID        int

    Name      string

    Address   string

    DoB       time.Time
    Position  string

    Salary    int

    ManagerID int

}

var
 dilbert Employee
```

dilbert結構體變量的成員可以通過點操作符訪問，比如dilbert.Name和dilbert.DoB。因爲dilbert是一個變量，它所有的成員也同樣是變量，我們可以直接對每個成員賦值：

```
dilbert.Salary -= 5000
 // demoted, for writing too few lines of code
```

或者是對成員取地址，然後通過指針訪問：

```
position := &dilbert.Position
*position = "Senior "
 + *position // promoted, for outsourcing to Elbonia
```

點操作符也可以和指向結構體的指針一起工作：

```
var
 employeeOfTheMonth *Employee = &dilbert
employeeOfTheMonth.Position += " (proactive team player)"
```

相當於下面語句

```
(*employeeOfTheMonth).Position += " (proactive team player)"
```

下面的EmployeeByID函數將根據給定的員工ID返回對應的員工信息結構體的指針。我們可以使用點操作符來訪問它里面的成員：

```
func
 EmployeeByID(id int
) *Employee { /* ... */
 }

fmt.Println(EmployeeByID(dilbert.ManagerID).Position) // "Pointy-haired boss"


id := dilbert.ID
EmployeeByID(id).Salary = 0
 // fired for... no real reason
```

後面的語句通過EmployeeByID返回的結構體指針更新了Employee結構體的成員。如果將EmployeeByID函數的返回值從 `*Employee` 指針類型改爲Employee值類型，那麼更新語句將不能編譯通過，因爲在賦值語句的左邊併不確定是一個變量（譯註：調用函數返回的是值，併不是一個可取地址的變量）。

通常一行對應一個結構體成員，成員的名字在前類型在後，不過如果相鄰的成員類型如果相同的話可以被合併到一行，就像下面的Name和Address成員那樣：

```
type
 Employee struct
 {
    ID            int

    Name, Address string

    DoB           time.Time
    Position      string

    Salary        int

    ManagerID     int

}
```

結構體成員的輸入順序也有重要的意義。我們也可以將Position成員合併（因爲也是字符串類型），或者是交換Name和Address出現的先後順序，那樣的話就是定義了不同的結構體類型。通常，我們隻是將相關的成員寫到一起。

如果結構體成員名字是以大寫字母開頭的，那麽該成員就是導出的；這是Go語言導出規則決定的。一個結構體可能同時包含導出和未導出的成員。

結構體類型往往是冗長的，因爲它的每個成員可能都會占一行。雖然我們每次都可以重寫整個結構體成員，但是重複會令人厭煩。因此，完整的結構體寫法通常隻在類型聲明語句的地方出現，就像Employee類型聲明語句那樣。

一個命名爲S的結構體類型將不能再包含S類型的成員：因爲一個聚合的值不能包含它自身。（該限製同樣適應於數組。）但是S類型的結構體可以包含 `*S` 指針類型的成員，這可以讓我們創建遞歸的數據結構，比如鏈表和樹結構等。在下面的代碼中，我們使用一個二叉樹來實現一個插入排序：

```
gopl.io/ch4/treesort

type
 tree struct
 {
    value       int

    left, right *tree
}

// Sort sorts values in place.

func
 Sort(values []int
) {
    var
 root *tree
    for
 _, v := range
 values {
        root = add(root, v)
    }
```

```
    appendValues(values[:0
], root)
}

// appendValues appends the elements of t to values in order

// and returns the resulting slice.

func
 appendValues(values []int
, t *tree) []int
 {
    if
 t != nil
 {
        values = appendValues(values, t.left)
        values = append
(values, t.value)
        values = appendValues(values, t.right)
    }
    return
 values
}

func
 add(t *tree, value int
) *tree {
    if
 t == nil
 {
        // Equivalent to return &tree{value: value}.

        t = new
(tree)
        t.value = value
        return
 t
    }
    if
 value < t.value {
        t.left = add(t.left, value)
    } else
 {
        t.right = add(t.right, value)
    }
    return
 t
}
```

結構體類型的零值是每個成員都對是零值。通常會將零值作爲最合理的默認值。例如，對於bytes.Buffer類型，結構體初始值就是一個隨時可用的空緩存，還有在第9章將會講到的sync.Mutex的零值也是有效的未鎖定狀態。有時候這種零值可用的特性是自

然獲得的，但是也有些類型需要一些額外的工作。

如果結構體沒有任何成員的話就是空結構體，寫作struct{}。它的大小爲0，也不包含任何信息，但是有時候依然是有價值的。有些Go語言程序員用map帶模擬set數據結構時，用它來代替map中布爾類型的value，隻是強調key的重要性，但是因爲節約的空間有限，而且語法比較複雜，所有我們通常避免避免這樣的用法。

```
seen := make
(map
[string
]struct
{}) // set of strings

// ...

if
 _, ok := seen[s]; !ok {
    seen[s] = struct
{}{}
    // ...first time seeing s...

}
```

4.4.1. 結構體面值

結構體值也可以用結構體面值表示，結構體面值可以指定每個成員的值。

```
type
 Point struct
{ X, Y int
 }

p := Point{1
, 2
}
```

這里有兩種形式的結構體面值語法，上面的是第一種寫法，要求以結構體成員定義的順序爲每個結構體成員指定一個面值。它要求寫代碼和讀代碼的人要記住結構體的每個成員的類型和順序，不過結構體成員有細微的調整就可能導致上述代碼不能編譯。因此，上述的語法一般隻在定義結構體的包內部使用，或者是在較小的結構體中使用，這些結構體的成員排列比較規則，比如image.Point{x, y}或color.RGBA{red, green, blue, alpha}。

其實更常用的是第二種寫法，以成員名字和相應的值來初始化，可以包含部分或全部的成員，如1.4節的Lissajous程序的寫法：

```
anim := gif.GIF{LoopCount: nframes}
```

在這種形式的結構體面值寫法中，如果成員被忽略的話將默認用零值。因爲，提供了成員的名字，所有成員出現的順序併不重要。

兩種不同形式的寫法不能混合使用。而且，你不能企圖在外部包中用第一種順序賦值的技巧來偷偷地初始化結構體中未導出的成員。

```
package
 p
type
 T struct
{ a, b int
 } // a and b are not exported


package
 q
import
 "p"

var
 _ = p.T{a: 1
, b: 2
} // compile error: can't reference a, b

var
 _ = p.T{1
, 2
}       // compile error: can't reference a, b
```

雖然上面最後一行代碼的編譯錯誤信息中併沒有顯式提到未導出的成員，但是這樣企圖隱式使用未導出成員的行爲也是不允許的。

結構體可以作爲函數的參數和返迴值。例如，這個Scale函數將Point類型的值縮放後返迴：

```
func
 Scale(p Point, factor int
) Point {
    return
 Point{p.X * factor, p.Y * factor}
}

fmt.Println(Scale(Point{1
, 2
}, 5
)) // "{5 10}"
```

如果考慮效率的話，較大的結構體通常會用指針的方式傳入和返迴，

```
func
 Bonus(e *Employee, percent int
) int
 {
    return
 e.Salary * percent / 100

}
```

如果要在函數內部脩改結構體成員的話，用指針傳入是必鬚的；因爲在Go語言中，所有的函數參數都是值拷貝傳入的，函數參數將不再是函數調用時的原始變量。

```
func
 AwardAnnualRaise(e *Employee) {
    e.Salary = e.Salary * 105
 / 100

}
```

因爲結構體通常通過指針處理，可以用下面的寫法來創建併初始化一個結構體變量，併返迴結構體的地址：

```
pp := &Point{1
, 2
}
```

它是下面的語句是等價的

```
pp := new
(Point)
*pp = Point{1
, 2
}
```

不過&Point{1, 2}寫法可以直接在表達式中使用，比如一個函數調用。

4.4.2. 結構體比較

如果結構體的全部成員都是可以比較的，那麼結構體也是可以比較的，那樣的話兩個結構體將可以使用==或!=運算符進行比較。相等比較運算符==將比較兩個結構體的每個成員，因此下面兩個比較的表達式是等價的：

```
type
 Point struct
{ X, Y int
 }

p := Point{1
, 2
}
q := Point{2
, 1
}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"

fmt.Println(p == q)                   // "false"
```

可比較的結構體類型和其他可比較的類型一樣，可以用於map的key類型。

```
type
 address struct
 {
    hostname string

    port     int

}

hits := make
(map
[address]int
)
hits[address{"golang.org"
, 443
}]++
```

4.4.3. 結構體嵌入和匿名成員

在本節中，我們將看到如何使用Go語言提供的不同尋常的結構體嵌入機製讓一個命名的結構體包含另一個結構體類型的匿名成員，這樣就可以通過簡單的點運算符x.f來訪問匿名成員鏈中嵌套的x.d.e.f成員。

考慮一個二維的繪圖程序，提供了一個各種圖形的庫，例如矩形、橢圓形、星形和輪形等幾何形狀。這里是其中兩個的定義：

```
type
 Circle struct
 {
    X, Y, Radius int

}

type
 Wheel struct
 {
    X, Y, Radius, Spokes int

}
```

一個Circle代表的圓形類型包含了標準圓心的X和Y坐標信息，和一個Radius表示的半徑信息。一個Wheel輪形除了包含Circle類型所有的全部成員外，還增加了Spokes表示徑向輻條的數量。我們可以這樣創建一個wheel變量：

```
var
 w Wheel
w.X = 8

w.Y = 8

w.Radius = 5

w.Spokes = 20
```

隨着庫中幾何形狀數量的增多，我們一定會註意到它們之間的相似和重複之處，所以我們可能爲了便於維護而將相同的屬性獨立出來：

```
type
 Point struct
 {
    X, Y int

}

type
 Circle struct
 {
    Center Point
    Radius int

}

type
 Wheel struct
 {
    Circle Circle
    Spokes int

}
```

這樣改動之後結構體類型變的清晰了，但是這種脩改同時也導致了訪問每個成員變得繁瑣：

```
var
 w Wheel
w.Circle.Center.X = 8

w.Circle.Center.Y = 8

w.Circle.Radius = 5

w.Spokes = 20
```

Go語言有一個特性讓我們隻聲明一個成員對應的數據類型而不指名成員的名字；這類成員就叫匿名成員。匿名成員的數據類型必鬚是命名的類型或指向一個命名的類型的指針。下面的代碼中，Circle和Wheel各自都有一個匿名成員。我們可以說Point類型被嵌入到了Circle結構體，同時Circle類型被嵌入到了Wheel結構體。

```
type
 Circle struct
 {
    Point
    Radius int

}

type
 Wheel struct
 {
    Circle
    Spokes int

}
```

得意於匿名嵌入的特性，我們可以直接訪問葉子屬性而不需要給出完整的路徑：

```
var
 w Wheel
w.X = 8
            // equivalent to w.Circle.Point.X = 8

w.Y = 8
            // equivalent to w.Circle.Point.Y = 8

w.Radius = 5
       // equivalent to w.Circle.Radius = 5

w.Spokes = 20
```

在右邊的註釋中給出的顯式形式訪問這些葉子成員的語法依然有效，因此匿名成員併不是眞的無法訪問了。其中匿名成員Circle和Point都有自己的名字——就是命名的類型名字——但是這些名字在點操作符中是可選的。我們在訪問子成員的時候可以忽略任何匿名成員部分。

不幸的是，結構體字面值併沒有簡短表示匿名成員的語法， 因此下面的語句都不能編譯通過：

```
w = Wheel{8
, 8
, 5
, 20
}                       // compile error: unknown fields

w = Wheel{X: 8
, Y: 8
, Radius: 5
, Spokes: 20
} // compile error: unknown fields
```

結構體字面值必鬚遵循形狀類型聲明時的結構，所以我們隻能用下面的兩種語法，它們彼此是等價的：

```
gopl.io/ch4/embed

w = Wheel{Circle{Point{8
, 8
}, 5
}, 20
}

w = Wheel{
    Circle: Circle{
        Point:  Point{X: 8
, Y: 8
},
        Radius: 5
,
    },
    Spokes: 20
, // NOTE:
 trailing comma necessary here (and at Radius)

}

fmt.Printf("%#v\n"
, w)
// Output:

// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}


w.X = 42


fmt.Printf("%#v\n"
, w)
// Output:

// Wheel{Circle:Circle{Point:Point{X:42, Y:8}, Radius:5}, Spokes:20}
```

需要註意的是Printf函數中%v參數包含的#副詞，它表示用和Go語言類似的語法打印值。對於結構體類型來説，將包含每個成員的名字。

因爲匿名成員也有一個隱式的名字，因此不能同時包含兩個類型相同的匿名成員，這會導致名字衝突。同時，因爲成員的名字是由其類型隱式地決定的，所有匿名成員也有可見性的規則約束。在上面的例子中，Point和Circle匿名成員都是導出的。即使它們不導出（比如改成小寫字母開頭的point和circle），我們依然可以用簡短形式訪問匿名成員嵌套的成員

```
w.X = 8
 // equivalent to w.circle.point.X = 8
```

但是在包外部，因爲circle和point沒有導出不能訪問它們的成員，因此簡短的匿名成員訪問語法也是禁止的。

到目前未知，我們看到匿名成員特性隻是對訪問嵌套成員的點運算符提供了簡短的語法醣。稍後，我們將會看到匿名成員併不要求是結構體類型；其實任何命令的類型都可以作爲結構體的匿名成員。但是爲什麼要嵌入一個沒有任何子成員類型的匿名成員類型呢？

答案是匿名類型的方法集。簡短的點運算符語法可以用於選擇匿名成員嵌套的成員，也可以用於訪問它們的方法。實際上，外層的結構體不僅僅是獲得了匿名成員類型的所有成員，而且也獲得了該類型導出的全部的方法。這個機製可以用於將一個有簡單行爲的對象組合成有複雜行爲的對象。組合是Go語言中面向對象編程的核心，我們將在6.3節中專門討論。

4.5. JSON

JavaScript對象表示法（JSON）是一種用於發送和接收結構化信息的標準協議。在類似的協議中，JSON併不是唯一的一個標準協議。 XML（§7.14）、ASN.1和Google的Protocol Buffers都是類似的協議，併且有各自的特色，但是由於簡潔性、可讀性和流行程度等原因，JSON是應用最廣泛的一個。

Go語言對於這些標準格式的編碼和解碼都有良好的支持，由標準庫中的encoding/json、encoding/xml、encoding/asn1等包提供支持（譯註：Protocol Buffers的支持由 github.com/golang/protobuf 包提供），併且這類包都有着相似的API接口。本節，我們將對重要的encoding/json包的用法做個概述。

JSON是對JavaScript中各種類型的值——字符串、數字、布爾值和對象——Unicode本文編碼。它可以用有效可讀的方式表示第三章的基礎數據類型和本章的數組、slice、結構體和map等聚合數據類型。

基本的JSON類型有數字（十進製或科學記數法）、布爾值（true或false）、字符串，其中字符串是以雙引號包含的Unicode字符序列，支持和Go語言類似的反斜槓轉義特性，不過JSON使用的是\Uhhhh轉義數字來表示一個UTF-16編碼（譯註：UTF-16和UTF-8一樣是一種變長的編碼，有些Unicode碼點較大的字符需要用4個字節表示；而且UTF-16還有大端和小端的問題），而不是Go語言的rune類型。

這些基礎類型可以通過JSON的數組和對象類型進行遞歸組合。一個JSON數組是一個有序的值序列，寫在一個方括號中併以逗號分隔；一個JSON數組可以用於編碼Go語言的數組和slice。一個JSON對象是一個字符串到值的映射，寫成以繫列的name:value對形式，用花括號包含併以逗號分隔；JSON的對象類型可以用於編碼Go語言的map類型（key類型是字符串）和結構體。例如：

```
boolean         true
number          -273.15
string          "She said \"Hello, BF\""
array           ["gold", "silver", "bronze"]
object          {"year": 1980,
                 "event": "archery",
                 "medals": ["gold", "silver", "bronze"]}
```

考慮一個應用程序，該程序負責收集各種電影評論併提供反饋功能。它的Movie數據類型和一個典型的表示電影的值列表如下所示。（在結構體聲明中，Year和Color成員後面的字符串面值是結構體成員Tag；我們稍後會解釋它的作用。）

```
gopl.io/ch4/movie

type
 Movie struct
 {
    Title  string

    Year   int
  `json:"released"`

    Color  bool
 `json:"color,omitempty"`

    Actors []string

}

var
 movies = []Movie{
    {Title: "Casablanca"
, Year: 1942
, Color: false
,
        Actors: []string
{"Humphrey Bogart"
, "Ingrid Bergman"
}},
    {Title: "Cool Hand Luke"
, Year: 1967
, Color: true
,
        Actors: []string
{"Paul Newman"
}},
    {Title: "Bullitt"
, Year: 1968
, Color: true
,
        Actors: []string
{"Steve McQueen"
, "Jacqueline Bisset"
}},
    // ...

}
```

這樣的數據結構特別適合JSON格式，併且在兩種之間相互轉換也很容易。將一個Go語言中類似movies的結構體slice轉爲JSON的過程叫編組（marshaling）。編組通過調用json.Marshal函數完成：

```
data, err := json.Marshal(movies)
if
 err != nil
 {
    log.Fatalf("JSON marshaling failed: %s"
, err)
}
fmt.Printf("%s\n"
, data)
```

Marshal函數返還一個編碼後的字節slice，包含很長的字符串，併且沒有空白縮進；我們將它摺行以便於顯示：

```
[{"Title":"Casablanca","released":1942,"Actors":["Humphrey Bogart","Ingr
id Bergman"]},{"Title":"Cool Hand Luke","released":1967,"color":true,"Ac
tors":["Paul Newman"]},{"Title":"Bullitt","released":1968,"color":true,"
Actors":["Steve McQueen","Jacqueline Bisset"]}]
```

這種緊湊的表示形式雖然包含了全部的信息，但是很難閱讀。爲了生成便於閱讀的格式，另一個json.MarshalIndent函數將產生整齊縮進的輸出。該函數有兩個額外的字符串參數用於表示每一行輸出的前綴和每一個層級的縮進：

```
data, err := json.MarshalIndent(movies, ""
, "    "
)
if
 err != nil
 {
    log.Fatalf("JSON marshaling failed: %s"
, err)
}
fmt.Printf("%s\n"
, data)
```

上面的代碼將產生這樣的輸出（譯註：在最後一個成員或元素後面併沒有逗號分隔符）：

```
[
    {
        "Title"
: "Casablanca"
,
        "released"
: 1942
,
        "Actors"
: [
            "Humphrey Bogart"
,
            "Ingrid Bergman"
```

```
        ]
    },
    {
        "Title"
: "Cool Hand Luke"
,
        "released"
: 1967
,
        "color"
: true
,
        "Actors"
: [
            "Paul Newman"

        ]
    },
    {
        "Title"
: "Bullitt"
,
        "released"
: 1968
,
        "color"
: true
,
        "Actors"
: [
            "Steve McQueen"
,
            "Jacqueline Bisset"

        ]
    }
]
```

在編碼時，默認使用Go語言結構體的成員名字作爲JSON的對象（通過reflect反射技術，我們將在12.6節討論）。隻有導出的結構體成員才會被編碼，這也就是我們爲什麽選擇用大寫字母開頭的成員名稱。

細心的讀者可能已經註意到，其中Year名字的成員在編碼後變成了released，還有Color成員編碼後變成了小寫字母開頭的color。這是因爲構體成員Tag所導致的。一個構體成員Tag是和在編譯階段關聯到該成員的元信息字符串：

```
Year  int  `json:"released"`
Color bool `json:"color,omitempty"`
```

結構體的成員Tag可以是任意的字符串面值，但是通常是一繫列用空格分隔的key:"value"鍵值對序列；因爲值中含義雙引號字符，因此成員Tag一般用原生字符串面值的形式書寫。json開頭鍵名對應的值用於控製encoding/json包的編碼和解碼的行爲，併且encoding/...下面其它的包也遵循這個約定。成員Tag中json對應值的第一部分用於指定JSON對象的名字，比如將Go語言中的

TotalCount成員對應到JSON中的total_count對象。Color成員的Tag還帶了一個額外的omitempty選項，表示當Go語言結構體成員爲空或零值時不生成JSON對象（這里false爲零值）。果然，Casablanca是一個黑白電影，併沒有輸出Color成員。

編碼的逆操作是解碼，對應將JSON數據解碼爲Go語言的數據結構，Go語言中一般叫unmarshaling，通過json.Unmarshal函數完成。下面的代碼將JSON格式的電影數據解碼爲一個結構體slice，結構體中隻有Title成員。通過定義合適的Go語言數據結構，我們可以選擇性地解碼JSON中感興趣的成員。當Unmarshal函數調用返迴，slice將被隻含有Title信息值填充，其它JSON成員將被忽略。

```
var
 titles []struct
{ Title string
 }
if
 err := json.Unmarshal(data, &titles); err != nil
 {
    log.Fatalf("JSON unmarshaling failed: %s"
, err)
}
fmt.Println(titles) // "[{Casablanca} {Cool Hand Luke} {Bullitt}]"
```

許多web服務都提供JSON接口，通過HTTP接口發送JSON格式請求併返迴JSON格式的信息。爲了說明這一點，我們通過Github的issue查詢服務來演示類似的用法。首先，我們要定義合適的類型和常量：

```
gopl.io/ch4/github
// Package github provides a Go API for the GitHub issue tracker.

// See https://developer.github.com/v3/search/#search-issues.

package
 github

import
 "time"


const
 IssuesURL = "https://api.github.com/search/issues"


type
 IssuesSearchResult struct
 {
    TotalCount int
 `json:"total_count"`

    Items          []*Issue
}

type
 Issue struct
```

```
 {
    Number    int

    HTMLURL   string
 `json:"html_url"`

    Title     string

    State     string

    User      *User
    CreatedAt time.Time `json:"created_at"`

    Body      string
    // in Markdown format

}

type
 User struct
 {
    Login   string

    HTMLURL string
 `json:"html_url"`

}
```

和前面一樣，即使對應的JSON對象名是小寫字母，每個結構體的成員名也是聲明爲大小字母開頭的。因爲有些JSON成員名字和Go結構體成員名字併不相同，因此需要Go語言結構體成員Tag來指定對應的JSON名字。同樣，在解碼的時候也需要做同樣的處理，GitHub服務返迴的信息比我們定義的要多很多。

SearchIssues函數發出一個HTTP請求，然後解碼返迴的JSON格式的結果。因爲用戶提供的查詢條件可能包含類似 `?` 和 `&` 之類的特殊字符，爲了避免對URL造成衝突，我們用url.QueryEscape來對查詢中的特殊字符進行轉義操作。

```
gopl.io/ch4/github
package
 github

import
 (
    "encoding/json"

    "fmt"

    "net/http"

    "net/url"

    "strings"
```

```
)

// SearchIssues queries the GitHub issue tracker.

func
 SearchIssues(terms []string
) (*IssuesSearchResult, error) {
    q := url.QueryEscape(strings.Join(terms, " "
))
    resp, err := http.Get(IssuesURL + "?q="
 + q)
    if
 err != nil
 {
        return
 nil
, err
    }

    // We must close resp.Body on all execution paths.

    // (Chapter 5 presents 'defer', which makes this simpler.)

    if
 resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return
 nil
, fmt.Errorf("search query failed: %s"
, resp.Status)
    }

    var
 result IssuesSearchResult
    if
 err := json.NewDecoder(resp.Body).Decode(&result); err != nil
 {
        resp.Body.Close()
        return
 nil
, err
    }
    resp.Body.Close()
    return
 &result, nil

}
```

在早些的例子中，我們使用了json.Unmarshal函數來將JSON格式的字符串解碼爲字節slice。但是這個例子中，我們使用了基於流式的解碼器json.Decoder，它可以從一個輸入流解碼JSON數據，盡管這不是必須的。如您所料，還有一個針對輸出流的json.Encoder編碼對象。

我們調用Decode方法來填充變量。這里有多種方法可以格式化結構。下面是最簡單的一種，以一個固定寬度打印每個issue，但是在下一節我們將看到如果利用模闆來輸出複雜的格式。

```
gopl.io/ch4/issues

// Issues prints a table of GitHub issues matching the search terms.

package
 main

import
 (
    "fmt"

    "log"

    "os"


    "gopl.io/ch4/github"

)

func
 main() {
    result, err := github.SearchIssues(os.Args[1
:])
    if
 err != nil
 {
        log.Fatal(err)
    }
    fmt.Printf("%d issues:\n"
, result.TotalCount)
    for
 _, item := range
 result.Items {
        fmt.Printf("#%-5d %9.9s %.55s\n"
,
            item.Number, item.User.Login, item.Title)
    }
}
```

通過命令行參數指定檢索條件。下面的命令是查詢Go語言項目中和JSON解碼相關的問題，還有查詢返迴的結果：

```
$ go build gopl.io/ch4/issues
$ ./issues repo:golang/go is:open json decoder
13 issues:
#5680    eaigner encoding/json: set key converter on en/decoder
#6050  gopherbot encoding/json: provide tokenizer
#8658  gopherbot encoding/json: use bufio
#8462  kortschak encoding/json: UnmarshalText confuses json.Unmarshal
#5901        rsc encoding/json: allow override type marshaling
#9812  klauspost encoding/json: string tag not symmetric
#7872  extempora encoding/json: Encoder internally buffers full output
#9650    cespare encoding/json: Decoding gives errPhase when unmarshalin
#6716  gopherbot encoding/json: include field name in unmarshal error me
#6901  lukescott encoding/json, encoding/xml: option to treat unknown fi
#6384    joeshaw encoding/json: encode precise floating point integers u
#6647    btracey x/tools/cmd/godoc: display type kind of each named type
#4237  gjemiller encoding/base64: URLEncoding padding is optional
```

GitHub的Web服務接口 https://developer.github.com/v3/ 包含了更多的特性。

練習 4.10： 脩改issues程序，根據問題的時間進行分類，比如不到一個月的、不到一年的、超過一年。

練習 4.11： 編寫一個工具，允許用戶在命令行創建、讀取、更新和刪除GitHub上的issue，當必要的時候自動打開用戶默認的編輯器用於輸入文本信息。

練習 4.12： 流行的web漫畫服務xkcd也提供了JSON接口。例如，一個 https://xkcd.com/571/info.0.json 請求將返迴一個很多人喜愛的571編號的詳細描述。下載每個鏈接（隻下載一次）然後創建一個離線索引。編寫一個xkcd工具，使用這些離線索引，打印和命令行輸入的檢索詞相匹配的漫畫的URL。

練習 4.13： 使用開放電影數據庫的JSON服務接口，允許你檢索和下載 https://omdbapi.com/ 上電影的名字和對應的海報圖像。編寫一個poster工具，通過命令行輸入的電影名字，下載對應的海報。

4.6. 文本和HTML模闆

前面的例子，隻是最簡單的格式化，使用Printf是完全足夠的。但是有時候會需要複雜的打印格式，這時候一般需要將格式化代碼分離出來以便更安全地脩改。這寫功能是由text/template和html/template等模闆包提供的，它們提供了一個將變量值填充到一個文本或HTML格式的模闆的機製。

一個模闆是一個字符串或一個文件，里面包含了一個或多個由雙花括號包含的 `{{action}}` 對象。大部分的字符串隻是按面值打印，但是對於actions部分將觸發其它的行爲。每個actions都包含了一個用模闆語言書寫的表達式，一個action雖然簡短但是可以輸出複雜的打印值，模闆語言包含通過選擇結構體的成員、調用函數或方法、表達式控製流if-else語句和range循環語句，還有其它實例化模闆等諸多特性。下面是一個簡單的模闆字符串：

```
gopl.io/ch4/issuesreport

const
 templ = `{{.TotalCount}} issues:
{{range .Items}}----------------------------------------
Number: {{.Number}}
User:   {{.User.Login}}
Title:  {{.Title | printf "%.64s"}}
Age:    {{.CreatedAt | daysAgo}} days
{{end}}`
```

這個模闆先打印匹配到的issue總數，然後打印每個issue的編號、創建用戶、標題還有存在的時間。對於每一個action，都有一個當前值的概念，對應點操作符，寫作“.”。當前值“.”最初被初始化爲調用模闆是的參數，在當前例子中對應github.IssuesSearchResult類型的變量。模闆中 `{{.TotalCount}}` 對應action將展開爲結構體中TotalCount成員以默認的方式打印的值。模闆中 `{{range .Items}}` 和 `{{end}}` 對應一個循環action，因此它們直接的內容可能會被展開多次，循環每次迭代的當前值對應當前的Items元素的值。

在一個action中， `|` 操作符表示將前一個表達式的結果作爲後一個函數的輸入，類似於UNIX中管道的概念。在Title這一行的action中，第二個操作是一個printf函數，是一個基於fmt.Sprintf實現的內置函數，所有模闆都可以直接使用。對於Age部分，第二個動作是一個叫daysAgo的函數，通過time.Since函數將CreatedAt成員轉換爲過去的時間長度：

```
func
 daysAgo(t time.Time) int
 {
    return
 int
(time.Since(t).Hours() / 24
)
}
```

需要註意的是CreatedAt的參數類型是time.Time，併不是字符串。以同樣的方式，我們可以通過定義一些方法來控製字符串的格式化（§2.5），一個類型同樣可以定製自己的JSON編碼和解碼行爲。time.Time類型對應的JSON值是一個標準時間格式的字符串。

生成模闆的輸出需要兩個處理步驟。第一步是要分析模闆併轉爲內部表示，然後基於指定的輸入執行模闆。分析模闆部分一般隻需要執行一次。下面的代碼創建併分析上面定義的模闆templ。註意方法調用鏈的順序：template.New先創建併返迴一個模闆；Funcs方法將daysAgo等自定義函數註冊到模闆中，併返迴模闆；最後調用Parse函數分析模闆。

```
report, err := template.New("report"
).
    Funcs(template.FuncMap{"daysAgo"
: daysAgo}).
    Parse(templ)
if
 err != nil
 {
    log.Fatal(err)
}
```

因爲模闆通常在編譯時就測試好了，如果模闆解析失敗將是一個致命的錯誤。template.Must輔助函數可以簡化這個致命錯誤的處理：它接受一個模闆和一個error類型的參數，檢測error是否爲nil（如果不是nil則發出panic異常），然後返迴傳入的模闆。我們將在5.9節再討論這個話題。

一旦模闆已經創建、註冊了daysAgo函數、併通過分析和檢測，我們就可以使用github.IssuesSearchResult作爲輸入源、os.Stdout作爲輸出源來執行模闆：

```
var
 report = template.Must(template.New("issuelist"
).
    Funcs(template.FuncMap{"daysAgo"
: daysAgo}).
    Parse(templ))

func
 main() {
    result, err := github.SearchIssues(os.Args[1
:])
    if
 err != nil
 {
        log.Fatal(err)
    }
    if
 err := report.Execute(os.Stdout, result); err != nil
 {
        log.Fatal(err)
    }
}
```

程序輸出一個純文本報告：

```
$ go build gopl.io/ch4/issuesreport
$ ./issuesreport repo:golang/go is:open json decoder
13 issues:
----------------------------------------
Number: 5680
User:      eaigner
Title:     encoding/json: set key converter on en/decoder
Age:       750 days
----------------------------------------
Number: 6050
User:      gopherbot
Title:     encoding/json: provide tokenizer
Age:       695 days
----------------------------------------
...
```

現在讓我們轉到html/template模闆包。它使用和text/template包相同的API和模闆語言，但是增加了一個將字符串自動轉義特性，這可以避免輸入字符串和HTML、JavaScript、CSS或URL語法產生衝突的問題。這個特性還可以避免一些長期存在的安全問題，比如通過生成HTML註入攻擊，通過構造一個含有惡意代碼的問題標題，這些都可能讓模闆輸出錯誤的輸出，從而讓他們控製頁面。

下面的模闆以HTML格式輸出issue列表。註意import語句的不同：

```
gopl.io/ch4/issueshtml

import
 "html/template"



var
 issueList = template.Must(template.New("issuelist"
).Parse(`
<h1>{{.TotalCount}} issues</h1>
<table>
<tr style='text-align: left'>
  <th>#</th>
  <th>State</th>
  <th>User</th>
  <th>Title</th>
</tr>
{{range .Items}}
<tr>
  <td><a href='{{.HTMLURL}}'>{{.Number}}</td>
  <td>{{.State}}</td>
  <td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>
  <td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>
</tr>
{{end}}
</table>
`
))
```

下面的命令將在新的模闆上執行一個稍微不同的查詢：

```
$ go build gopl.io/ch4/issueshtml
$ ./issueshtml repo:golang/go commenter:gopherbot json encoder >issues.html
```

圖4.4顯示了在web瀏覽器中的效果圖。每個issue包含到Github對應頁面的鏈接。

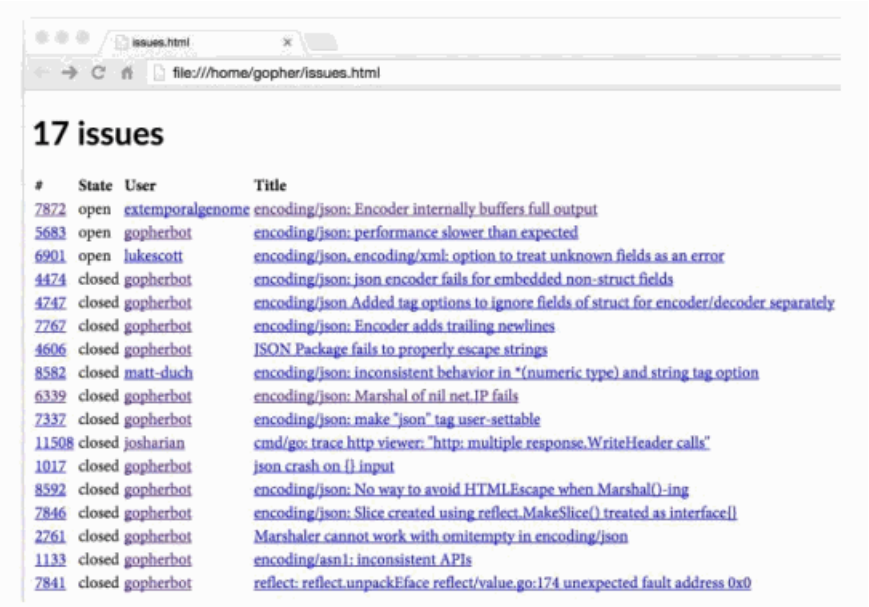


Figure 4.4. An HTML table of Go project issues relating to JSON encoding.

圖4.4中issue沒有包含會對HTML格式產生衝突的特殊字符，但是我們馬上將看到標題中含有 `&` 和 `<` 字符的issue。下面的命令選擇了兩個這樣的issue：

```
$ ./issueshtml repo:golang/go 3133 10535 >issues2.html
```

圖4.5顯示了該查詢的結果。註意，html/template包已經自動將特殊字符轉義，因此我們依然可以看到正確的字面值。如果我們使用text/template包的話，這2個issue將會產生錯誤，其中“<”四個字符將會被當作小於字符“<”處理，同時“<link>”字符串將會被當作一個鏈接元素處理，它們都會導致HTML文檔結構的改變，從而導致有未知的風險。

我們也可以通過對信任的HTML字符串使用template.HTML類型來抑製這種自動轉義的行爲。還有很多采用類型命名的字符串類型分別對應信任的JavaScript、CSS和URL。下面的程序演示了兩個使用不同類型的相同字符串產生的不同結果：A是一個普通字符串，B是一個信任的template.HTML字符串類型。

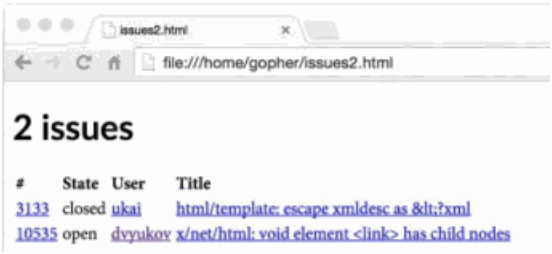


Figure 4.5. HTML metacharacters in issue titles are correctly displayed.

```
gopl.io/ch4/autoescape

func
 main() {
    const
 templ = `<p>A: {{.A}}</p><p>B: {{.B}}</p>`

    t := template.Must(template.New("escape"
).Parse(templ))
    var
 data struct
 {
        A string
        // untrusted plain text

        B template.HTML // trusted HTML

    }
    data.A = "<b>Hello!</b>"

    data.B = "<b>Hello!</b>"

    if
 err := t.Execute(os.Stdout, data); err != nil
 {
        log.Fatal(err)
    }
}
```

圖4.6顯示了出現在瀏覽器中的模闆輸出。我們看到A的黑體標記被轉義失效了，但是B沒有。

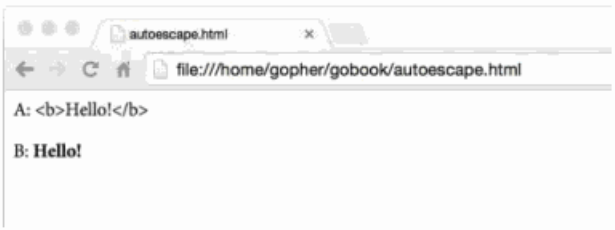


Figure 4.6. String values are HTML-escaped but `template.HTML` values are not.

我們這里隻講述了模闆繫統中最基本的特性。一如既往，如果想了解更多的信息，請自己查看包文檔：

```
$ go doc text/template
$ go doc html/template
```

練習 4.14： 創建一個web服務器，查詢一次GitHub，然後生成BUG報告、里程碑和對應的用戶信息。

第五章 函數

函數可以讓我們將一個語句序列打包爲一個單元，然後可以從程序中其它地方多次調用。函數的機製可以讓我們將一個大的工作分解爲小的任務，這樣的小任務可以讓不同程序員在不同時間、不同地方獨立完成。一個函數同時對用戶隱藏了其實現細節。由於這些因素，對於任何編程語言來説，函數都是一個至關重要的部分。

我們已經見過許多函數了。現在，讓我們多花一點時間來徹底地討論函數特性。本章的運行示例是一個網絡蜘蛛，也就是web蒐索引擎中負責抓取網頁部分的組件，它們根據抓取網頁中的鏈接繼續抓取鏈接指向的頁面。一個網絡蜘蛛的例子給我們足夠的機會去探索遞歸函數、匿名函數、錯誤處理和函數其它的很多特性。

5.1. 函數聲明

函數聲明包括函數名、形式參數列表、返迴值列表（可省略）以及函數體。

```
func
 name(parameter-list) (result-list) {
    body
}
```

形式參數列表描述了函數的參數名以及參數類型。這些參數作爲局部變量，其值由參數調用者提供。返迴值列表描述了函數返迴值的變量名以及類型。如果函數返迴一個無名變量或者沒有返迴值，返迴值列表的括號是可以省略的。如果一個函數聲明不包括返迴值列表，那麼函數體執行完畢後，不會返迴任何值。 在hypot函數中,

```
func
 hypot(x, y float64
) float64
 {
    return
 math.Sqrt(x*x + y*y)
}
fmt.Println(hypot(3
,4
)) // "5"
```

x和y是形參名,3和4是調用時的傳入的實數，函數返迴了一個float64類型的值。 返迴值也可以像形式參數一樣被命名。在這種情況下，每個返迴值被聲明成一個局部變量，併根據該返迴值的類型，將其初始化爲0。 如果一個函數在聲明時，包含返迴值列表，該函數必鬚以 return語句結尾，除非函數明顯無法運行到結尾處。例如函數在結尾時調用了panic異常或函數中存在無限循環。

正如hypot一樣，如果一組形參或返迴值有相同的類型，我們不必爲每個形參都寫出參數類型。下面2個聲明是等價的：

```
func
 f(i, j, k int
, s, t string
)                 { /* ... */
 }
func
 f(i int
, j int
, k int
,  s string
, t string
) { /* ... */
 }
```

下面，我們給出4種方法聲明擁有2個int型參數和1個int型返迴值的函數.blank identifier(譯者註：即下文的_符號)可以強調某個參數未被使用。

```
func
 add(x int
, y int
) int
   {return
 x + y}
func
 sub(x, y int
) (z int
)   { z = x - y; return
}
func
 first(x int
, _ int
) int
 { return
 x }
func
 zero(int
, int
) int
      { return
 0
 }

fmt.Printf("%T\n"
, add)   // "func(int, int) int"

fmt.Printf("%T\n"
, sub)   // "func(int, int) int"

fmt.Printf("%T\n"
, first) // "func(int, int) int"

fmt.Printf("%T\n"
, zero)  // "func(int, int) int"
```

函數的類型被稱爲函數的標識符。如果兩個函數形式參數列表和返迴值列表中的變量類型一一對應，那麽這兩個函數被認爲有相同的類型和標識符。形參和返迴值的變量名不影響函數標識符也不影響它們是否可以以省略參數類型的形式表示。

每一次函數調用都必鬚按照聲明順序爲所有參數提供實參（參數值）。在函數調用時，Go語言沒有默認參數值，也沒有任何方法可以通過參數名指定形參，因此形參和返迴值的變量名對於函數調用者而言沒有意義。

在函數體中，函數的形參作爲局部變量，被初始化爲調用者提供的值。函數的形參和有名返迴值作爲函數最外層的局部變量，被存儲在相同的詞法塊中。

實參通過值的方式傳遞，因此函數的形參是實參的拷貝。對形參進行修改不會影響實參。但是，如果實參包括引用類型，如指針，slice(切片)、map、function、channel等類型，實參可能會由於函數的簡介引用被修改。

你可能會偶爾遇到沒有函數體的函數聲明，這表示該函數不是以Go實現的。這樣的聲明定義了函數標識符。

```
package
 math

func
 Sin(x float64
) float //implemented in assembly language
```

5.2. 遞歸

函數可以是遞歸的，這意味着函數可以直接或間接的調用自身。對許多問題而言，遞歸是一種強有力的技術，例如處理遞歸的數據結構。在4.4節，我們通過遍歷二叉樹來實現簡單的插入排序，在本章節，我們再次使用它來處理HTML文件。

下文的示例代碼使用了非標準包 golang.org/x/net/html，解析HTML。golang.org/x/... 目録下存儲了一些由Go糰隊設計、維護，對網絡編程、国際化文件處理、移動平台、圖像處理、加密解密、開發者工具提供支持的擴展包。未將這些擴展包加入到標準庫原因有二，一是部分包仍在開發中，二是對大多數Go語言的開發者而言，擴展包提供的功能很少被使用。

例子中調用golang.org/x/net/html的部分api如下所示。html.Parse函數讀入一組bytes.解析後，返迴html.node類型的HTML頁面樹狀結構根節點。HTML擁有很多類型的結點如text（文本）,commnets（註釋）類型，在下面的例子中，我們 隻關註< name key='value' >形式的結點。

```
golang.org/x/net/html
package
 html

type
 Node struct
 {
    Type                    NodeType
    Data                    string

    Attr                    []Attribute
    FirstChild, NextSibling *Node
}

type
 NodeType int32



const
 (
    ErrorNode NodeType = iota

    TextNode
    DocumentNode
    ElementNode
    CommentNode
    DoctypeNode
)

type
 Attribute struct
 {
    Key, Val string

}

func
 Parse(r io.Reader) (*Node, error)
```

main函數解析HTML標準輸入，通過遞歸函數visit獲得links（鏈接），併打印出這些links：

```
gopl.io/ch5/findlinks1
// Findlinks1 prints the links in an HTML document read from standard input.

package
 main

import
 (
    "fmt"

    "os"


    "golang.org/x/net/html"

)

func
 main() {
    doc, err := html.Parse(os.Stdin)
    if
 err != nil
 {
        fmt.Fprintf(os.Stderr, "findlinks1: %v\n"
, err)
        os.Exit(1
)
    }
    for
 _, link := range
 visit(nil
, doc) {
        fmt.Println(link)
    }
}
```

visit函數遍歷HTML的節點樹，從每一個anchor元素的href屬性獲得link,將這些links存入字符串數組中，併返迴這個字符串數組。

```
// visit appends to links each link found in n and returns the result.

func
 visit(links []string
, n *html.Node) []string
 {
    if
 n.Type == html.ElementNode && n.Data == "a"
 {
        for
 _, a := range
 n.Attr {
            if
 a.Key == "href"
 {
                links = append
(links, a.Val)
            }
        }
    }
    for
 c := n.FirstChild; c != nil
; c = c.NextSibling {
        links = visit(links, c)
    }
    return
 links
}
```

爲了遍歷結點n的所有後代結點，每次遇到n的孩子結點時，visit遞歸的調用自身。這些孩子結點存放在FirstChild鏈表中。

讓我們以Go的主頁（golang.org）作爲目標，運行findlinks。我們以fetch（1.5章）的輸出作爲findlinks的輸入。下面的輸出做了簡化處理。

```
$ go build gopl.io/ch1/fetch
$ go build gopl.io/ch5/findlinks1
$ ./fetch https://golang.org | ./findlinks1
#
/doc/
/pkg/
/help/
/blog/
http://play.golang.org/
//tour.golang.org/
https://golang.org/dl/
//blog.golang.org/
/LICENSE
/doc/tos.html
http://www.google.com/intl/en/policies/privacy/
```

註意在頁面中出現的鏈接格式，在之後我們會介紹如何將這些鏈接，根據根路徑（ https://golang.org ）生成可以直接訪問的url。

在函數outline中，我們通過遞歸的方式遍歷整個HTML結點樹，併輸出樹的結構。在outline內部，每遇到一個HTML元素標籤，就將其入棧，併輸出。

```
gopl.io/ch5/outline
func
 main() {
    doc, err := html.Parse(os.Stdin)
    if
 err != nil
 {
        fmt.Fprintf(os.Stderr, "outline: %v\n"
, err)
        os.Exit(1
)
    }
    outline(nil
, doc)
}
func
 outline(stack []string
, n *html.Node) {
    if
 n.Type == html.ElementNode {
        stack = append
(stack, n.Data) // push tag

        fmt.Println(stack)
    }
    for
 c := n.FirstChild; c != nil
; c = c.NextSibling {
        outline(stack, c)
    }
}
```

有一點值得註意：outline有入棧操作，但沒有相對應的出棧操作。當outline調用自身時，被調用者接收的是stack的拷貝。被調用者的入棧操作，脩改的是stack的拷貝，而不是調用者的stack,因對當函數返迴時,調用者的stack併未被脩改。

下面是 https://golang.org 頁面的簡要結構:

```
$ go build gopl.io/ch5/outline
$ ./fetch https://golang.org | ./outline
[html]
[html head]
[html head meta]
[html head title]
[html head link]
[html body]
[html body div]
[html body div]
[html body div div]
[html body div div form]
[html body div div form div]
[html body div div form div a]
...
```

正如你在上面實驗中所見，大部分HTML頁面隻需幾層遞歸就能被處理，但仍然有些頁面需要深層次的遞歸。

大部分編程語言使用固定大小的函數調用棧，常見的大小從64KB到2MB不等。固定大小棧會限製遞歸的深度，當你用遞歸處理大量數據時，需要避免棧溢出；除此之外，還會導致安全性問題。與相反,Go語言使用可變棧，棧的大小按需增加(初始時很小)。這使得我們使用遞歸時不必考慮溢出和安全問題。

練習**5.1** :脩改findlinks代碼中遍歷n.FirstChild鏈表的部分，將循環調用visit，改成遞歸調用。

練習**5.2** : 編寫函數，記録在HTML樹中出現的同名元素的次數。

練習**5.3** : 編寫函數輸出所有text結點的內容。註意不要訪問 `<script>` 和 `<style>` 元素,因爲這些元素對瀏覽者是不可見的。

練習**5.4** : 擴展vist函數，使其能夠處理其他類型的結點，如images、scripts和style sheets。

5.3. 多返迴值

在Go中，一個函數可以返迴多個值。我們已經在之前例子中看到，許多標準庫中的函數返迴2個值，一個是期望得到的返迴值，另一個是函數出錯時的錯誤信息。下面的例子會展示如何編寫多返迴值的函數。

下面的程序是findlinks的改進版本。脩改後的findlinks可以自己發起HTTP請求，這樣我們就不必再運行fetch。因爲HTTP請求和解析操作可能會失敗，因此findlinks聲明了2個返迴值：鏈接列表和錯誤信息。一般而言，HTML的解析器可以處理HTML頁面的錯誤結點，構造出HTML頁面結構，所以解析HTML很少失敗。這意味着如果findlinks函數失敗了，很可能是由於I/O的錯誤導致的。

```
gopl.io/ch5/findlinks2
func
 main() {
    for
 _, url := range
 os.Args[1
:] {
        links, err := findLinks(url)
        if
 err != nil
 {
            fmt.Fprintf(os.Stderr, "findlinks2: %v\n"
, err)
            continue

        }
        for
 _, link := range
 links {
            fmt.Println(link)
        }
    }
}

// findLinks performs an HTTP GET request for url, parses the

// response as HTML, and extracts and returns the links.

func
 findLinks(url string
) ([]string
, error) {
    resp, err := http.Get(url)
    if
 err != nil
 {
        return
 nil
, err
    }
    if
 resp.StatusCode != http.StatusOK {
        resp.Body.Close()
```

```
        resp.Body.Close()
        return
 nil
, fmt.Errorf("getting %s: %s"
, url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if
 err != nil
 {
        return
 nil
, fmt.Errorf("parsing %s as HTML: %v"
, url, err)
    }
    return
 visit(nil
, doc), nil

}
```

在findlinks中，有4處return語句，每一處return都返迴了一組值。前三處return，將http和html包中的錯誤信息傳遞給findlinks的調用者。第一處return直接返迴錯誤信息，其他兩處通過fmt.Errorf（§7.8）輸出詳細的錯誤信息。如果findlinks成功結束，最後的return語句將一組解析獲得的連接返迴給用戶。

在finallinks中，我們必鬚確保resp.Body被關閉，釋放網絡資源。雖然Go的垃圾迴收機製會迴收不被使用的內存，但是這不包括操作繫統層面的資源，比如打開的文件、網絡連接。因此我們必鬚顯式的釋放這些資源。

調用多返迴值函數時，返迴給調用者的是一組值，調用者必鬚顯式的將這些值分配給變量:

```
links, err := findLinks(url)
```

如果某個值不被使用，可以將其分配給blank identifier:

```
links, _ := findLinks(url) // errors ignored
```

一個函數內部可以將另一個有多返迴值的函數作爲返迴值，下面的例子展示了與findLinks有相同功能的函數，兩者的區別在於下面的例子先輸出參數：

```
func
 findLinksLog(url string
) ([]string
, error) {
    log.Printf("findLinks %s"
, url)
    return
 findLinks(url)
}
```

當你調用接受多參數的函數時，可以將一個返迴多參數的函數作爲該函數的參數。雖然這很少出現在實際生產代碼中，但這個特性在debug時很方便，我們隻需要一條語句就可以輸出所有的返迴值。下面的代碼是等價的：

```
log.Println(findLinks(url))
links, err := findLinks(url)
log.Println(links, err)
```

準確的變量名可以傳達函數返迴值的含義。尤其在返迴值的類型都相同時，就像下面這樣：

```
func
 Size(rect image.Rectangle) (width, height int
)
func
 Split(path string
) (dir, file string
)
func
 HourMinSec(t time.Time) (hour, minute, second int
)
```

雖然良好的命名很重要，但你也不必爲每一個返迴值都取一個適當的名字。比如，按照慣例，函數的最後一個bool類型的返迴值表示函數是否運行成功，error類型的返迴值代表函數的錯誤信息，對於這些類似的慣例，我們不必思考合適的命名，它們都無需解釋。

如果一個函數將所有的返迴值都顯示的變量名，那麽該函數的return語句可以省略操作數。這稱之爲bare return。

```
// CountWordsAndImages does an HTTP GET request for the HTML

// document url and returns the number of words and images in it.

func
 CountWordsAndImages(url string
) (words, images int
, err error) {
    resp, err := http.Get(url)
    if
 err != nil
 {
        return

    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if
 err != nil
 {
        err = fmt.Errorf("parsing HTML: %s"
, err)
    return

    }
    words, images = countWordsAndImages(doc)
    return

}
func
 countWordsAndImages(n *html.Node) (words, images int
) { /* ... */
 }
```

按照返迴值列表的次序，返迴所有的返迴值，在上面的例子中，每一個return語句等價於：

```
return
 words, images, err
```

當一個函數有多處return語句以及許多返迴值時，bare return 可以減少代碼的重複，但是使得代碼難以被理解。舉個例子，如果你沒有仔細的審查代碼，很難發現前2處return等價於 return 0,0,err（Go會將返迴值 words和images在函數體的開始處，根據它們的類型，將其初始化爲0），最後一處return等價於 return words，image，nil。基於以上原因，不宜過度使用bare return。

練習 5.5： 實現countWordsAndImages。（參考練習4.9如何分詞）

練習 5.6： 脩改gopl.io/ch3/surface (§3.2) 中的corner函數，將返迴值命名，併使用bare return。

5.4. 錯誤

在Go中有一部分函數總是能成功的運行。比如string.Contains和strconv.FormatBool函數，對各種可能的輸入都做了良好的處理，使得運行時幾乎不會失敗，除非遇到災難性的、不可預料的情況，比如運行時的內存溢出。導致這種錯誤的原因很複雜，難以處理，從錯誤中恢複的可能性也很低。

還有一部分函數隻要輸入的參數滿足一定條件，也能保證運行成功。比如time.Date函數，該函數將年月日等參數構造成time.Time對象，除非最後一個參數（時區）是nil。這種情況下會引發panic異常。panic是來自被調函數的信號，表示發生了某個已知的bug。一個良好的程序永遠不應該發生panic異常。

對於大部分函數而言，永遠無法確保能否成功運行。這是因爲錯誤的原因超出了程序員的控製。舉個例子，任何進行I/O操作的函數都會面臨出現錯誤的可能，隻有沒有經驗的程序員才會相信讀寫操作不會失敗，即時是簡單的讀寫。因此，當本該可信的操作出乎意料的失敗後，我們必鬚弄清楚導致失敗的原因。

在Go的錯誤處理中，錯誤是軟件包API和應用程序用戶界面的一個重要組成部分，程序運行失敗僅被認爲是幾個預期的結果之一。

對於那些將運行失敗看作是預期結果的函數，它們會返迴一個額外的返迴值，通常是最後一個，來傳遞錯誤信息。如果導致失敗的原因隻有一個，額外的返迴值可以是一個布爾值，通常被命名爲ok。比如，cache.Lookup失敗的唯一原因是key不存在，那麼代碼可以按照下面的方式組織：

```
value, ok := cache.Lookup(key)
if
 !ok {
    // ...cache[key] does not exist…


}
```

通常，導致失敗的原因不止一種，尤其是對I/O操作而言，用戶需要了解更多的錯誤信息。因此，額外的返迴值不再是簡單的布爾類型，而是error類型。

內置的error是接口類型。我們將在第七章了解接口類型的含義，以及它對錯誤處理的影響。現在我們隻需要明白error類型可能是nil或者non-nil。nil意味着函數運行成功，non-nil表示失敗。對於non-nil的error類型,我們可以通過調用error的Error函數或者輸出函數獲得字符串類型的錯誤信息。

```
fmt.Println(err)
fmt.Printf("%v"
, err)
```

通常，當函數返迴non-nil的error時，其他的返迴值是未定義的(undefined),這些未定義的返迴值應該被忽略。然而，有少部分函數在發生錯誤時，仍然會返迴一些有用的返迴值。比如，當讀取文件發生錯誤時，Read函數會返迴可以讀取的字節數以及錯誤信息。對於這種情況，正確的處理方式應該是先處理這些不完整的數據，再處理錯誤。因此對函數的返迴值要有清晰的說明，以便於其他人使用。

在Go中，函數運行失敗時會返迴錯誤信息，這些錯誤信息被認爲是一種預期的值而非異常（exception），這使得Go有別於那些將函數運行失敗看作是異常的語言。雖然Go有各種異常機製，但這些機製僅被使用在處理那些未被預料到的錯誤，即bug，而不是那些在健壯程序中應該被避免的程序錯誤。對於Go的異常機製我們將在5.9介紹。

Go這樣設計的原因是由於對於某個應該在控製流程中處理的錯誤而言，將這個錯誤以異常的形式拋出會混亂對錯誤的描述，這通常會導致一些糟糕的後果。當某個程序錯誤被當作異常處理後，這個錯誤會將堆棧根據信息返迴給終端用戶，這些信息複雜且無用，無法幫助定位錯誤。

正因此，Go使用控製流機製（如if和return）處理異常，這使得編碼人員能更多的關註錯誤處理。

5.4.1. 錯誤處理策略

當一次函數調用返迴錯誤時，調用者有應該選擇何時的方式處理錯誤。根據情況的不同，有很多處理方式，讓我們來看看常用的五種方式。

首先，也是最常用的方式是傳播錯誤。這意味着函數中某個子程序的失敗，會變成該函數的失敗。下面，我們以5.3節的findLinks函數作爲例子。如果findLinks對http.Get的調用失敗，findLinks會直接將這個HTTP錯誤返迴給調用者：

```
resp, err := http.Get(url)
if
 err != nil
{
    return
 nill, err
}
```

當對html.Parse的調用失敗時，findLinks不會直接返迴html.Parse的錯誤，因爲缺少兩條重要信息：1、錯誤發生在解析器；2、url已經被解析。這些信息有助於錯誤的處理，findLinks會構造新的錯誤信息返迴給調用者：

```
doc, err := html.Parse(resp.Body)
resp.Body.Close()
if
 err != nil
 {
    return
 nil
, fmt.Errorf("parsing %s as HTML: %v"
, url,err)
}
```

fmt.Errorf函數使用fmt.Sprintf格式化錯誤信息併返迴。我們使用該函數前綴添加額外的上下文信息到原始錯誤信息。當錯誤最終由main函數處理時，錯誤信息應提供清晰的從原因到後果的因果鏈，就像美国宇航局事故調查時做的那樣：

```
genesis: crashed: no parachute: G-switch failed: bad relay orientation
```

由於錯誤信息經常是以鏈式組合在一起的，所以錯誤信息中應避免大寫和換行符。最終的錯誤信息可能很長，我們可以通過類似grep的工具處理錯誤信息（譯者註：grep是一種文本蒐索工具）。

編寫錯誤信息時，我們要確保錯誤信息對問題細節的描述是詳盡的。尤其是要註意錯誤信息表達的一致性，即相同的函數或同包內的同一組函數返迴的錯誤在構成和處理方式上是相似的。

以OS包爲例，OS包確保文件操作（如os.Open、Read、Write、Close）返迴的每個錯誤的描述不僅僅包含錯誤的原因（如無權限，文件目録不存在）也包含文件名，這樣調用者在構造新的錯誤信息時無需再添加這些信息。

一般而言，被調函數f(x)會將調用信息和參數信息作爲發生錯誤時的上下文放在錯誤信息中併返迴給調用者，調用者需要添加一些錯誤信息中不包含的信息，比如添加url到html.Parse返迴的錯誤中。

讓我們來看看處理錯誤的第二種策略。如果錯誤的發生是偶然性的，或由不可預知的問題導致的。一個明智的選擇是重新嚐試失敗的操作。在重試時，我們需要限製重試的時間間隔或重試的次數，防止無限製的重試。

```
gopl.io/ch5/wait
// WaitForServer attempts to contact the server of a URL.

// It tries for one minute using exponential back-off.

// It reports an error if all attempts fail.

func
 WaitForServer(url string
) error {
    const
 timeout = 1
 * time.Minute
    deadline := time.Now().Add(timeout)
    for
 tries := 0
; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if
 err == nil
 {
            return
 nil
 // success

        }
        log.Printf("server not responding (%s);retrying…"
, err)
        time.Sleep(time.Second << uint
(tries)) // exponential back-off

    }
    return
 fmt.Errorf("server %s failed to respond after %s"
, url, timeout)
}
```

如果錯誤發生後，程序無法繼續運行，我們就可以采用第三種策略：輸出錯誤信息併結束程序。需要註意的是，這種策略隻應在main中執行。對庫函數而言，應僅向上傳播錯誤，除非該錯誤意味着程序內部包含不一致性，即遇到了bug，才能在庫函數中結束程序。

```
// (In function main.)

if
 err := WaitForServer(url); err != nil
 {
    fmt.Fprintf(os.Stderr, "Site is down: %v\n"
, err)
    os.Exit(1
)
}
```

調用log.Fatalf可以更簡潔的代碼達到與上文相同的效果。log中的所有函數，都默認會在錯誤信息之前輸出時間信息。

```
if
 err := WaitForServer(url); err != nil
 {
    log.Fatalf("Site is down: %v\n"
, err)
}
```

長時間運行的服務器常采用默認的時間格式，而交互式工具很少采用包含如此多信息的格式。

```
2006/01/02 15:04:05 Site is down: no such domain:
bad.gopl.io
```

我們可以設置log的前綴信息屏蔽時間信息，一般而言，前綴信息會被設置成命令名。

```
log.SetPrefix("wait: "
)
log.SetFlags(0
)
```

第四種策略：有時，我們隻需要輸出錯誤信息就足夠了，不需要中斷程序的運行。我們可以通過log包提供函數

```
if
 err := Ping(); err != nil
 {
    log.Printf("ping failed: %v; networking disabled"
,err)
}
```

或者標準錯誤流輸出錯誤信息。

```
if
 err := Ping(); err != nil
 {
    fmt.Fprintf(os.Stderr, "ping failed: %v; networking disabled\n"
, err)
}
```

log包中的所有函數會爲沒有換行符的字符串增加換行符。

第五種，也是最後一種策略：我們可以直接忽略掉錯誤。

```
dir, err := ioutil.TempDir(""
, "scratch"
)
if
 err != nil
 {
    return
 fmt.Errorf("failed to create temp dir: %v"
,err)
}
// ...use temp dir…

os.RemoveAll(dir) // ignore errors; $TMPDIR is cleaned periodically
```

盡管os.RemoveAll會失敗，但上面的例子併沒有做錯誤處理。這是因爲操作繫統會定期的清理臨時目録。正因如此，雖然程序沒有處理錯誤，但程序的邏輯不會因此受到影響。我們應該在每次函數調用後，都養成考慮錯誤處理的習慣，當你決定忽略某個錯誤時，你應該在清晰的記録下你的意圖。

在Go中，錯誤處理有一套獨特的編碼風格。檢查某個子函數是否失敗後，我們通常將處理失敗的邏輯代碼放在處理成功的代碼之前。如果某個錯誤會導致函數返迴，那麼成功時的邏輯代碼不應放在else語句塊中，而應直接放在函數體中。Go中大部分函數的代碼結構幾乎相同，首先是一繫列的初始檢查，防止錯誤發生，之後是函數的實際邏輯。

5.4.2. 文件結尾錯誤（EOF）

函數經常會返迴多種錯誤，這對終端用戶來説可能會很有趣，但對程序而言，這使得情況變得複雜。很多時候，程序必鬚根據錯誤類型，作出不同的響應。讓我們考慮這樣一個例子：從文件中讀取n個字節。如果n等於文件的長度，讀取過程的任何錯誤都表示失敗。如果n小於文件的長度，調用者會重複的讀取固定大小的數據直到文件結束。這會導致調用者必鬚分別處理由文件結束引起的各種錯誤。基於這樣的原因，io包保證任何由文件結束引起的讀取失敗都返迴同一個錯誤——io.EOF，該錯誤在io包中定義：

```
package
 io

import
 "errors"


// EOF is the error returned by Read when no more input is available.

var
 EOF = errors.New("EOF"
)
```

調用者隻需通過簡單的比較，就可以檢測出這個錯誤。下面的例子展示了如何從標準輸入中讀取字符，以及判斷文件結束。（4.3的chartcount程序展示了更加複雜的代碼）

```
in := bufio.NewReader(os.Stdin)
for
 {
    r, _, err := in.ReadRune()
    if
 err == io.EOF {
        break
 // finished reading

    }
    if
 err != nil
 {
        return
 fmt.Errorf("read failed:%v"
, err)
    }
    // ...use r…

}
```

因爲文件結束這種錯誤不需要更多的描述，所以io.EOF有固定的錯誤信息——“EOF”。對於其他錯誤，我們可能需要在錯誤信息中描述錯誤的類型和數量，這使得我們不能像io.EOF一樣采用固定的錯誤信息。在7.11節中，我們會提出更繁統的方法區分某些固定的錯誤值。

5.5. 函數值

在Go中，函數被看作第一類值（first-class values）：函數像其他值一樣，擁有類型，可以被賦值給其他變量，傳遞給函數，從函數返迴。對函數值（function value）的調用類似函數調用。例子如下：

```
    func
 square(n int
) int
 { return
 n * n }
    func
 negative(n int
) int
 { return
 -n }
    func
 product(m, n int
) int
 { return
 m * n }

    f := square
    fmt.Println(f(3
)) // "9"



    f = negative
    fmt.Println(f(3
))     // "-3"

    fmt.Printf("%T\n"
, f) // "func(int) int"



    f = product // compile error: can't assign func(int, int) int to func(int) int
```

函數類型的零值是nil。調用值爲nil的函數值會引起panic錯誤：

```
    var
 f func
(int
) int

    f(3
) // 此處f的值爲nil,會引起panic錯誤
```

函數值可以與nil比較：

```
    var
 f func
(int
) int

    if
 f != nil
 {
        f(3
)
    }
```

但是函數值之間是不可比較的，也不能用函數值作爲map的key。

函數值使得我們不僅僅可以通過數據來參數化函數，亦可通過行爲。標準庫中包含許多這樣的例子。下面的代碼展示了如何使用這個技巧。string.Map對字符串中的每個字符調用add1函數，併將每個add1函數的返迴值組成一個新的字符串返迴給調用者。

```
    func
 add1(r rune
) rune
 { return
 r + 1
 }

    fmt.Println(strings.Map(add1, "HAL-9000"
)) // "IBM.:111"

    fmt.Println(strings.Map(add1, "VMS"
))      // "WNT"

    fmt.Println(strings.Map(add1, "Admix"
))    // "Benjy"
```

5.2節的findLinks函數使用了輔助函數visit,遍歷和操作了HTML頁面的所有結點。使用函數值，我們可以將遍歷結點的邏輯和操作結點的邏輯分離，使得我們可以複用遍歷的邏輯，從而對結點進行不同的操作。

```
gopl.io/ch5/outline2
// forEachNode針對每個結點x,都會調用pre(x)和post(x)。

// pre和post都是可選的。

// 遍歷孩子結點之前,pre被調用

// 遍歷孩子結點之後，post被調用

func
 forEachNode(n *html.Node, pre, post func
(n *html.Node)) {
    if
 pre != nil
 {
        pre(n)
    }
    for
 c := n.FirstChild; c != nil
; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if
 post != nil
 {
        post(n)
    }
}
```

該函數接收2個函數作爲參數，分別在結點的孩子被訪問前和訪問後調用。這樣的設計給調用者更大的靈活性。舉個例子，現在我們有startElemen和endElement兩個函數用於輸出HTML元素的開始標籤和結束標籤 `<b>...</b>` ：

```
var
 depth int

func
 startElement(n *html.Node) {
    if
 n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n"
, depth*2
, ""
, n.Data)
        depth++
    }
}
func
 endElement(n *html.Node) {
    if
 n.Type == html.ElementNode {
        depth--
        fmt.Printf("%*s</%s>\n"
, depth*2
, ""
, n.Data)
    }
}
```

上面的代碼利用fmt.Printf的一個小技巧控製輸出的縮進。`%*s` 中的 `*` 會在字符串之前填充一些空格。在例子中,每次輸出會先填充 `depth*2` 數量的空格，再輸出""，最後再輸出HTML標籤。

如果我們像下面這樣調用forEachNode：

```
forEachNode(doc, startElement, endElement)
```

與之前的outline程序相比，我們得到了更加詳細的頁面結構：

```
$ go build gopl.io/ch5/outline2
$ ./outline2 http://gopl.io
<html>
  <head>
    <meta>
    </meta>
    <title>
    </title>
    <style>
    </style>
  </head>
  <body>
    <table>
      <tbody>
        <tr>
          <td>
            <a>
              <img>
              </img>
...
```

練習 5.7： 完善startElement和endElement函數，使其成爲通用的HTML輸出器。要求：輸出註釋結點，文本結點以及每個元素的屬性（< a href='...'>）。使用簡略格式輸出沒有孩子結點的元素（即用 `<img/>` 代替 `<img></img>` ）。編寫測試，驗證程序輸出的格式正確。（詳見11章）

練習 5.8： 脩改pre和post函數，使其返迴布爾類型的返迴值。返迴false時，中止forEachNoded的遍歷。使用脩改後的代碼編寫ElementByID函數，根據用戶輸入的id查找第一個擁有該id元素的HTML元素，查找成功後，停止遍歷。

```
func
 ElementByID(doc *html.Node, id string
) *html.Node
```

練習 5.9： 編寫函數expand，將s中的"foo"替換爲f("foo")的返迴值。

```
func
 expand(s string
, f func
(string
) string
) string
```

5.6. 匿名函數

擁有函數名的函數隻能在包級語法塊中被聲明，通過函數字面量（function literal），我們可繞過這一限製，在任何表達式中表示一個函數值。函數字面量的語法和函數聲明相似，區別在於func關鍵字後沒有函數名。函數值字面量是一種表達式，它的值被成爲匿名函數（anonymous function）。

函數字面量允許我們在使用時函數時，再定義它。通過這種技巧，我們可以改寫之前對strings.Map的調用：

```
strings.Map(func
(r rune
) rune
 { return
 r + 1
 }, "HAL-9000"
)
```

更爲重要的是，通過這種方式定義的函數可以訪問完整的詞法環境（lexical environment），這意味着在函數中定義的內部函數可以引用該函數的變量，如下例所示：

```
gopl.io/ch5/squares
// squares返回一個匿名函數。

// 該匿名函數每次被調用時都會返回下一個數的平方。

func
 squares() func
() int
 {
    var
 x int

    return
 func
() int
 {
        x++
        return
 x * x
    }
}
func
 main() {
    f := squares()
    fmt.Println(f()) // "1"

    fmt.Println(f()) // "4"

    fmt.Println(f()) // "9"

    fmt.Println(f()) // "16"

}
```

函數squares返回另一個類型爲 func() int 的函數。對squares的一次調用會生成一個局部變量x併返回一個匿名函數。每次調用時匿名函數時，該函數都會先使x的值加1，再返回x的平方。第二次調用squares時，會生成第二個x變量，併返回一個新的匿名函數。新匿名函數操作的是第二個x變量。

squares的例子證明，函數值不僅僅是一串代碼，還記録了狀態。在squares中定義的匿名內部函數可以訪問和更新squares中的局部變量，這意味着匿名函數和squares中，存在變量引用。這就是函數值屬於引用類型和函數值不可比較的原因。Go使用閉包（closures）技術實現函數值，Go程序員也把函數值叫做閉包。

通過這個例子，我們看到變量的生命週期不由它的作用域決定：squares返回後，變量x仍然隱式的存在於f中。

接下來，我們討論一個有點學術性的例子，考慮這樣一個問題：給定一些計算機課程，每個課程都有前置課程，隻有完成了前置課程才可以開始當前課程的學習；我們的目標是選擇出一組課程，這組課程必鬚確保按順序學習時，能全部被完成。每個課程的前置課程如下：

```
gopl.io/ch5/toposort
// prereqs記錄了每個課程的前置課程

var
 prereqs = map
[string
][]string
{
    "algorithms"
: {"data structures"
},
    "calculus"
: {"linear algebra"
},
    "compilers"
: {
        "data structures"
,
        "formal languages"
,
        "computer organization"
,
    },
    "data structures"
:       {"discrete math"
},
    "databases"
:             {"data structures"
},
    "discrete math"
:         {"intro to programming"
},
    "formal languages"
:      {"discrete math"
},
    "networks"
:              {"operating systems"
},
    "operating systems"
:     {"data structures"
, "computer organization"
},
    "programming languages"
: {"data structures"
, "computer organization"
},
}
```

這類問題被稱作拓撲排序。從概念上說，前置條件可以構成有向圖。圖中的頂點表示課程，邊表示課程間的依賴關繫。顯然，圖中應該無環，這也就是說從某點出發的邊，最終不會迴到該點。下面的代碼用深度優先蒐索了整張圖，獲得了符合要求的課程序列。

```
func
 main() {
    for
 i, course := range
 topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n"
, i+1
, course)
    }
}

func
 topoSort(m map
[string
][]string
) []string
 {
    var
 order []string

    seen := make
(map
[string
]bool
)
    var
 visitAll func
(items []string
)
    visitAll = func
(items []string
) {
        for
 _, item := range
 items {
            if
 !seen[item] {
                seen[item] = true

                visitAll(m[item])
                order = append
(order, item)
            }
        }
    }
    var
 keys []string
```

```
    for
 key := range
 m {
    keys = append
(keys, key)
    }
    sort.Strings(keys)
    visitAll(keys)
    return
 order
}
```

當匿名函數需要被遞歸調用時，我們必鬚首先聲明一個變量（在上面的例子中，我們首先聲明了 visitAll），再將匿名函數賦值給這個變量。如果不分成兩部，函數字面量無法與visitAll綁定，我們也無法遞歸調用該匿名函數。

```
visitAll := func
(items []string
) {
    // ...

    visitAll(m[item]) // compile error: undefined: visitAll

    // ...

}
```

在topsort中，首先對prereqs中的key排序，再調用visitAll。因爲prereqs映射的是切片而不是更複雜的map，所以數據的遍歷次序是固定的，這意味着你每次運行topsort得到的輸出都是一樣的。 topsort的輸出結果如下:

```
1: intro to programming
2: discrete math
3: data structures
4: algorithms
5: linear algebra
6: calculus
7: formal languages
8: computer organization
9: compilers
10: databases
11: operating systems
12: networks
13: programming languages
```

讓我們迴到findLinks這個例子。我們將代碼移動到了links包下，將函數重命名爲Extract，在第八章我們會再次用到這個函數。新的匿名函數被引入，用於替換原來的visit函數。該匿名函數負責將新連接添加到切片中。在Extract中，使用forEachNode遍歷HTML頁面，由於Extract隻需要在遍歷結點前操作結點，所以forEachNode的post參數被傳入nil。

```
gopl.io/ch5/links
```

```
// Package links provides a link-extraction function.

package
 links
import
 (
    "fmt"

    "net/http"

    "golang.org/x/net/html"

)
// Extract makes an HTTP GET request to the specified URL, parses

// the response as HTML, and returns the links in the HTML document.

func
 Extract(url string
) ([]string
, error) {
    resp, err := http.Get(url)
    if
 err != nil
 {
        return
 nil
, err
    }
    if
 resp.StatusCode != http.StatusOK {
    resp.Body.Close()
        return
 nil
, fmt.Errorf("getting %s: %s"
, url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if
 err != nil
 {
        return
 nil
, fmt.Errorf("parsing %s as HTML: %v"
, url, err)
    }
    var
 links []string

    visitNode := func
(n *html.Node) {
```

```
        if
n.Type == html.ElementNode && n.Data == "a"
{
            for
_, a := range
n.Attr {
                if
a.Key != "href"
{
                    continue

                }
                link, err := resp.Request.URL.Parse(a.Val)
                if
err != nil
{
                    continue
// ignore bad URLs

                }
                links = append
(links, link.String())
            }
        }
    }
    forEachNode(doc, visitNode, nil
)
    return
 links, nil

}
```

上面的代碼對之前的版本做了改進，現在links中存儲的不是href屬性的原始值，而是通過resp.Request.URL解析後的值。解析後，這些連接以絕對路徑的形式存在，可以直接被http.Get訪問。

網頁抓取的核心問題就是如何遍歷圖。在topoSort的例子中，已經展示了深度優先遍歷，在網頁抓取中，我們會展示如何用廣度優先遍歷圖。在第8章，我們會介紹如何將深度優先和廣度優先結合使用。

下面的函數實現了廣度優先算法。調用者需要輸入一個初始的待訪問列表和一個函數f。待訪問列表中的每個元素被定義爲string類型。廣度優先算法會爲每個元素調用一次f。每次f執行完畢後，會返迴一組待訪問元素。這些元素會被加入到待訪問列表中。當待訪問列表中的所有元素都被訪問後，breadthFirst函數運行結束。爲了避免同一個元素被訪問兩次，代碼中維護了一個map。

```
gopl.io/ch5/findlinks3
// breadthFirst calls f for each item in the worklist.

// Any items returned by f are added to the worklist.

// f is called at most once for each item.

func
 breadthFirst(f func
(item string
) []string
, worklist []string
) {
    seen := make
(map
[string
]bool
)
    for
 len
(worklist) > 0
 {
        items := worklist
        worklist = nil

        for
 _, item := range
 items {
            if
 !seen[item] {
                seen[item] = true

                worklist = append
(worklist, f(item)...)
            }
        }
    }
}
```

就像我們在章節3解釋的那樣，append的參數“f(item)...”，會將f返迴的一組元素一個個添加到worklist中。

在我們網頁抓取器中，元素的類型是url。crawl函數會將URL輸出，提取其中的新鏈接，併將這些新鏈接返迴。我們會將crawl作爲參數傳遞給breadthFirst。

```
func
 crawl(url string
) []string
 {
    fmt.Println(url)
    list, err := links.Extract(url)
    if
 err != nil
 {
        log.Print(err)
    }
    return
 list
}
```

爲了使抓取器開始運行，我們用命令行輸入的參數作爲初始的待訪問url。

```
func
 main() {
    // Crawl the web breadth-first,

    // starting from the command-line arguments.

    breadthFirst(crawl, os.Args[1
:])
}
```

讓我們從 https://golang.org 開始，下面是程序的輸出結果：

```
$ go build gopl.io/ch5/findlinks3
$ ./findlinks3 https://golang.org
https://golang.org/
https://golang.org/doc/
https://golang.org/pkg/
https://golang.org/project/
https://code.google.com/p/go-tour/
https://golang.org/doc/code.html
https://www.youtube.com/watch?v=XCsL89YtqCs
http://research.swtch.com/gotour
```

當所有發現的鏈接都已經被訪問或電腦的內存耗盡時，程序運行結束。

練習5.10： 重寫topoSort函數，用map代替切片併移除對key的排序代碼。驗證結果的正確性（結果不唯一）。

練習5.11： 現在線性代數的老師把微積分設爲了前置課程。完善topSort，使其能檢測有向圖中的環。

練習 5.12： gopl.io/ch5/outline2（5.5節）的startElement和endElement共用了全局變量depth，將它們脩改爲匿名函數，使其共享outline中的局部變量。

練習 5.13： 脩改crawl，使其能保存發現的頁面，必要時，可以創建目録來保存這些頁面。隻保存來自原始域名下的頁面。假設初始頁面在golang.org下，就不要保存vimeo.com下的頁面。

練習 5.14： 使用breadthFirst遍歷其他數據結構。比如，topoSort例子中的課程依賴關繫（有向圖）,個人計算機的文件層次結構（樹），你所在城市的公交或地鐵線路（無向圖）。

5.6.1. 警告：捕獲迭代變量

本節，將介紹Go詞法作用域的一個陷阱。請務必仔細的閱讀，弄清楚發生問題的原因。即使是經驗豐富的程序員也會在這個問題上犯錯誤。

考慮這個樣一個問題：你被要求首先創建一些目録，再將目録刪除。在下面的例子中我們用函數值來完成刪除操作。下面的示例代碼需要引入os包。爲了使代碼簡單，我們忽略了所有的異常處理。

```
var
 rmdirs []func
()
for
 _, d := range
 tempDirs() {
    dir := d // NOTE:
 necessary!

    os.MkdirAll(dir, 0755
) // creates parent directories too

    rmdirs = append
(rmdirs, func
() {
        os.RemoveAll(dir)
    })
}
// ...do some work…

for
 _, rmdir := range
 rmdirs {
    rmdir() // clean up

}
```

你可能會感到睏惑，爲什麽要在循環體中用循環變量d賦值一個新的局部變量，而不是像下面的代碼一樣直接使用循環變量dir。需要註意，下面的代碼是錯誤的。

```
var
 rmdirs []func
()
for
 _, dir := range
 tempDirs() {
    os.MkdirAll(dir, 0755
)
    rmdirs = append
(rmdirs, func
() {
        os.RemoveAll(dir) // NOTE:
 incorrect!

    })
}
```

問題的原因在於循環變量的作用域。在上面的程序中，for循環語句引入了新的詞法塊，循環變量dir在這個詞法塊中被聲明。在該循環中生成的所有函數值都共享相同的循環變量。需要註意，函數值中記録的是循環變量的內存地址，而不是循環變量某一時刻的值。以dir爲例，後續的迭代會不斷更新dir的值，當刪除操作執行時，for循環已完成，dir中存儲的值等於最後一次迭代的值。這意味着，每次對os.RemoveAll的調用刪除的都是相同的目録。

通常，爲了解決這個問題，我們會引入一個與循環變量同名的局部變量，作爲循環變量的副本。比如下面的變量dir，雖然這看起來很奇怪，但卻很有用。

```
for
 _, dir := range
 tempDirs() {
    dir := dir // declares inner dir, initialized to outer dir

    // ...

}
```

這個問題不僅存在基於range的循環，在下面的例子中，對循環變量i的使用也存在同樣的問題：

```
var
 rmdirs []func
()
dirs := tempDirs()
for
 i := 0
; i < len
(dirs); i++ {
    os.MkdirAll(dirs[i], 0755
) // OK

    rmdirs = append
(rmdirs, func
() {
        os.RemoveAll(dirs[i]) // NOTE:
 incorrect!

    })
}
```

如果你使用go語句（第八章）或者defer語句（5.8節）會經常遇到此類問題。這不是go或defer本身導致的，而是因爲它們都會等待循環結束後，再執行函數值。

5.7. 可變參數

參數數量可變的函數稱爲爲可變參數函數。典型的例子就是fmt.Printf和類似函數。Printf首先接收一個的必備參數，之後接收任意個數的後續參數。

在聲明可變參數函數時，需要在參數列表的最後一個參數類型之前加上省略符號“...”，這表示該函數會接收任意數量的該類型參數。

```
gopl.io/ch5/sum
func
 sum(vals...int
) int
 {
    total := 0

    for
 _, val := range
 vals {
        total += val
    }
    return
 total
}
```

sum函數返迴任意個int型參數的和。在函數體中,vals被看作是類型爲[] int的切片。sum可以接收任意數量的int型參數：

```
fmt.Println(sum()) // "0"

fmt.Println(sum(3
)) // "3"

fmt.Println(sum(1
, 2
, 3
, 4
)) // "10"
```

在上面的代碼中，調用者隱式的創建一個數組，併將原始參數複製到數組中，再把數組的一個切片作爲參數傳給被調函數。如果原始參數已經是切片類型，我們該如何傳遞給sum？隻需在最後一個參數後加上省略符。下面的代碼功能與上個例子中最後一條語句相同。

```
values := []int
{1
, 2
, 3
, 4
}
fmt.Println(sum(values...)) // "10"
```

雖然在可變參數函數內部，...int 型參數的行爲看起來很像切片類型，但實際上，可變參數函數和以切片作爲參數的函數是不同的。

```
func
 f(...int
) {}
func
 g([]int
) {}
fmt.Printf("%T\n"
, f) // "func(...int)"

fmt.Printf("%T\n"
, g) // "func([]int)"
```

可變參數函數經常被用於格式化字符串。下面的errorf函數構造了一個以行號開頭的，經過格式化的錯誤信息。函數名的後綴f是一種通用的命名規范，代表該可變參數函數可以接收Printf風格的格式化字符串。

```
func errorf(linenum int, format string, args...interface{})
{
    fmt.Fprintf(os.Stderr, "Line %d: ", linenum)
    fmt.Fprintf(os.Stderr, format, args…)
    fmt.Fprintln(os.Stderr)
}
linenum, name := 12, "count"
errorf(linenum, "undefined: %s", name) // "Line 12:
undefined: count"
```

interfac{}表示函數的最後一個參數可以接收任意類型，我們會在第7章詳細介紹。

練習 5.15： 編寫類似sum的可變參數函數max和min。考慮不傳參時，max和min該如何處理，再編寫至少接收1個參數的版本。

練習 5.16： 編寫多參數版本的strings.Join。

練習 5.17： 編寫多參數版本的ElementsByTagName，函數接收一個HTML結點樹以及任意數量的標籤名，返迴與這些標籤名匹配的所有元素。下面給出了2個例子：

```
func
 ElementsByTagName(doc *html.Node, name...string
)
[]*html.Node
images := ElementsByTagName(doc, "img"
)
headings := ElementsByTagName(doc, "h1"
, "h2"
, "h3"
, "h4"
)
```

5.8. Deferred函數

TODO

5.9. Panic异常

TODO

5.10. Recover捕獲異常

TODO

第六章 方法

從90年代早期開始，面向對象編程(OOP)就成爲了稱霸工程界和教育界的編程范式，所以之後幾乎所有大規模被應用的語言都包含了對OOP的支持，go語言也不例外。

盡管沒有被大衆所接受的明確的OOP的定義，從我們的理解來講，一個對象其實也就是一個簡單的值或者一個變量，在這個對象中會包含一些方法，而一個方法則是一個一個和特殊類型關聯的函數。一個面向對象的程序會用方法來表達其屬性和對應的操作，這樣使用這個對象的用戶就不需要直接去操作對象，而是借助方法來做這些事情。

在早些的章節中，我們已經使用了標準庫提供的一些方法，比如time.Duration這個類型的Seconds方法：

```
const
 day = 24
 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

併且在2.5節中，我們定義了一個自己的方法，Celsius類型的String方法:

```
func
 (c Celsius) String() string
 { return
 fmt.Sprintf("%g°C"
, c) }
```

在本章中，OOP編程的第一方面，我們會向你展示如何有效地定義和使用方法。我們會覆蓋到OOP編程的兩個關鍵點，封裝和組合。

6.1. 方法聲明

在函數聲明時，在其名字之前放上一個變量，即是一個方法。這個附加的參數會將該函數附加到這種類型上，即相當於爲這種類型定義了一個獨占的方法。

下面來寫我們第一個方法的例子，這個例子在package geometry下：

```
gopl.io/ch6/geometry
package
 geometry

import
 "math"



type
 Point struct
{ X, Y float64
 }

// traditional function

func
 Distance(p, q Point) float64
 {
    return
 math.Hypot(q.X-p.X, q.Y-p.Y)
}



// same thing, but as a method of the Point type

func
 (p Point) Distance(q Point) float64
 {
    return
 math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

上面的代碼里那個附加的參數p，叫做方法的接收器(receiver)，早期的面向對象語言留下的遺產將調用一個方法稱爲“向一個對象發送消息”。

在Go語言中，我們併不會像其它語言那樣用this或者self作爲接收器；我們可以任意的選擇接收器的名字。由於接收器的名字經常會被使用到，所以保持其在方法間傳遞時的一致性和簡短性是不錯的主意。這里的建議是可以使用其類型的第一個字母，比如這里使用了Point的首字母p。

在方法調用過程中，接收器參數一般會在方法名之前出現。這和方法聲明是一樣的，都是接收器參數在方法名字之前。下面是例子：

```
p := Point{1
, 2
}
q := Point{4
, 6
}
fmt.Println(Distance(p, q)) // "5", function call

fmt.Println(p.Distance(q))  // "5", method call
```

可以看到，上面的兩個函數調用都是Distance，但是卻沒有發生衝突。第一個Distance的調用實際上用的是包級別的函數geometry.Distance，而第二個則是使用剛剛聲明的Point，調用的是Point類下聲明的Point.Distance方法。

這種p.Distance的表達式叫做選擇器，因爲他會選擇合適的對應p這個對象的Distance方法來執行。選擇器也會被用來選擇一個struct類型的字段，比如p.X。由於方法和字段都是在同一命名空間，所以如果我們在這里聲明一個X方法的話，編譯器會報錯，因爲在調用p.X時會有歧義(譯註：這里確實挺奇怪的)。

因爲每種類型都有其方法的命名空間，我們在用Distance這個名字的時候，不同的Distance調用指向了不同類型里的Distance方法。讓我們來定義一個Path類型，這個Path代表一個線段的集合，併且也給這個Path定義一個叫Distance的方法。

```
// A Path is a journey connecting the points with straight lines.

type
 Path []Point
// Distance returns the distance traveled along the path.

func
 (path Path) Distance() float64
 {
    sum := 0.0

    for
 i := range
 path {
        if
 i > 0
 {
            sum += path[i-1
].Distance(path[i])
        }
    }
    return
 sum
}
```

Path是一個命名的slice類型，而不是Point那樣的struct類型，然而我們依然可以爲它定義方法。在能夠給任意類型定義方法這一點上，Go和很多其它的面向對象的語言不太一樣。因此在Go語言里，我們爲一些簡單的數值、字符串、slice、map來定義一些附加行爲很方便。方法可以被聲明到任意類型，隻要不是一個指針或者一個interface。

兩個Distance方法有不同的類型。他們兩個方法之間沒有任何關繫，盡管Path的Distance方法會在內部調用Point.Distance方法來計算每個連接鄰接點的線段的長度。

讓我們來調用一個新方法，計算三角形的週長：

```
perim := Path{
    {1
, 1
},
    {5
, 1
},
    {5
, 4
},
    {1
, 1
},
}
fmt.Println(perim.Distance()) // "12"
```

在上面兩個對Distance名字的方法的調用中，編譯器會根據方法的名字以及接收器來決定具體調用的是哪一個函數。第一個例子中path[i-1]數組中的類型是Point，因此Point.Distance這個方法被調用；在第二個例子中perim的類型是Path，因此Distance調用的是Path.Distance。

對於一個給定的類型，其內部的方法都必鬚有唯一的方法名，但是不同的類型卻可以有同樣的方法名，比如我們這里Point和Path就都有Distance這個名字的方法；所以我們沒有必要非在方法名之前加類型名來消除歧義，比如PathDistance。這里我們已經看到了方法比之函數的一些好處：方法名可以簡短。當我們在包外調用的時候這種好處就會被放大，因爲我們可以使用這個短名字，而可以省略掉包的名字，下面是例子：

```
import
 "gopl.io/ch6/geometry"


perim := geometry.Path{{1
, 1
}, {5
, 1
}, {5
, 4
}, {1
, 1
}}
fmt.Println(geometry.PathDistance(perim)) // "12", standalone function

fmt.Println(perim.Distance())             // "12", method of geometry.Path
```

譯註：如果我們要用方法去計算perim的distance，還需要去寫全geometry的包名，和其函數名，但是因爲Path這個變量定義了一個可以直接用的Distance方法，所以我們可以直接寫perim.Distance()。相當於可以少打很多字，作者應該是這個意思。因爲在Go里包外調用函數需要帶上包名，還是挺麻煩的。

6.2. 基於指針對象的方法

當調用一個函數時，會對其每一個參數值進行拷貝，如果一個函數需要更新一個變量，或者函數的其中一個參數實在太大我們希望能夠避免進行這種默認的拷貝，這種情況下我們就需要用到指針了。對應到我們這里用來更新接收器的對象的方法，當這個接受者變量本身比較大時，我們就可以用其指針而不是對象來聲明方法，如下：

```
func
 (p *Point) ScaleBy(factor float64
) {
    p.X *= factor
    p.Y *= factor
}
```

這個方法的名字是 `(*Point).ScaleBy` 。這里的括號是必鬚的；沒有括號的話這個表達式可能會被理解爲 `*(Point.ScaleBy)` 。

在現實的程序里，一般會約定如果Point這個類有一個指針作爲接收器的方法，那麼所有Point的方法都必鬚有一個指針接收器，即使是那些併不需要這個指針接收器的函數。我們在這里打破了這個約定隻是爲了展示一下兩種方法的異同而已。

隻有類型(Point)和指向他們的指針(*Point)，才是可能會出現在接收器聲明里的兩種接收器。此外，爲了避免歧義，在聲明方法時，如果一個類型名本身是一個指針的話，是不允許其出現在接收器中的，比如下面這個例子：

```
type
 P *int

func
 (P) f() { /* ... */
 } // compile error: invalid receiver type
```

想要調用指針類型方法 `(*Point).ScaleBy` ，隻要提供一個Point類型的指針即可，像下面這樣。

```
r := &Point{1
, 2
}
r.ScaleBy(2
)
fmt.Println(*r) // "{2, 4}"
```

或者這樣：

```
p := Point{1
, 2
}
pptr := &p
pptr.ScaleBy(2
)
fmt.Println(p) // "{2, 4}"
```

或者這樣:

```
p := Point{1
, 2
}
(&p).ScaleBy(2
)
fmt.Println(p) // "{2, 4}"
```

不過後面兩種方法有些笨拙。幸運的是，go語言本身在這種地方會幫到我們。如果接收器p是一個Point類型的變量，併且其方法需要一個Point指針作爲接收器，我們可以用下面這種簡短的寫法：

```
p.ScaleBy(2
)
```

編譯器會隱式地幫我們用&p去調用ScaleBy這個方法。這種簡寫方法隻適用於“變量”，包括struct里的字段比如p.X，以及array和slice內的元素比如perim[0]。我們不能通過一個無法取到地址的接收器來調用指針方法，比如臨時變量的內存地址就無法獲取得到：

```
Point{1
, 2
}.ScaleBy(2
) // compile error: can't take address of Point literal
```

但是我們可以用一個 `*Point` 這樣的接收器來調用Point的方法，因爲我們可以通過地址來找到這個變量，隻要用解引用符號 `*` 來取到該變量即可。編譯器在這里也會給我們隱式地插入 `*` 這個操作符，所以下面這兩種寫法等價的：

```
pptr.Distance(q)
(*pptr).Distance(q)
```

這里的幾個例子可能讓你有些睏惑，所以我們總結一下：在每一個合法的方法調用表達式中，也就是下面三種情況里的任意一種情況都是可以的：

不論是接收器的實際參數和其接收器的形式參數相同，比如兩者都是類型T或者都是類型 `*T` ：

```
Point{1
, 2
}.Distance(q) //  Point

pptr.ScaleBy(2
)         // *Point
```

或者接收器形參是類型T，但接收器實參是類型 `*T` ，這種情況下編譯器會隱式地爲我們取變量的地址：

```
p.ScaleBy(2
) // implicit (&p)
```

或者接收器形參是類型 `*T` ，實參是類型T。編譯器會隱式地爲我們解引用，取到指針指向的實際變量：

```
pptr.Distance(q) // implicit (*pptr)
```

如果類型T的所有方法都是用T類型自己來做接收器(而不是 `*T`)，那麼拷貝這種類型的實例就是安全的；調用他的任何一個方法也就會產生一個值的拷貝。比如time.Duration的這個類型，在調用其方法時就會被全部拷貝一份，包括在作爲參數傳入函數的時候。但是如果一個方法使用指針作爲接收器，你需要避免對其進行拷貝，因爲這樣可能會破壞掉該類型內部的不變性。比如你對bytes.Buffer對象進行了拷貝，那麼可能會引起原始對象和拷貝對象隻是別名而已，但實際上其指向的對象是一致的。緊接着對拷貝後的變量進行脩改可能會有讓你意外的結果。

譯註：作者這里説的比較繞，其實有兩點：

1.不管你的method的receiver是指針類型還是非指針類型，都是可以通過指針/非指針類型進行調用的，編譯器會幫你做類型轉換
2.在聲明一個method的receiver該是指針還是非指針類型時，你需要考慮兩方面的內部，第一方面是這個對象本身是不是特別大，如果聲明爲非指針變量時，調用會產生一次拷貝；第二方面是如果你用指針類型作爲receiver，那麼你一定要註意，這種指針類型指向的始終是一塊內存地址，就算你對其進行了拷貝。熟悉C或者C++的人這里應該很快能明白。

6.2.1. Nil也是一個合法的接收器類型

就像一些函數允許nil指針作爲參數一樣，方法理論上也可以用nil指針作爲其接收器，尤其當nil對於對象來説是合法的零值時，比如map或者slice。在下面的簡單int鏈表的例子里，nil代表的是空鏈表：

```
// An IntList is a linked list of integers.

// A nil *IntList represents the empty list.

type
 IntList struct
 {
    Value int

    Tail  *IntList
}
// Sum returns the sum of the list elements.

func
 (list *IntList) Sum() int
 {
    if
 list == nil
 {
        return
 0

    }
    return
 list.Value + list.Tail.Sum()
}
```

當你定義一個允許nil作爲接收器值的方法的類型時，在類型前面的註釋中指出nil變量代表的意義是很有必要的，就像我們上面例子里做的這樣。

下面是net/url包里Values類型定義的一部分。

```
net/url
package
 url

// Values maps a string key to a list of values.

type
 Values map
[string
][]string

// Get returns the first value associated with the given key,

// or "" if there are none.

func
 (v Values) Get(key string
) string
 {
     if
 vs := v[key]; len
(vs) > 0
 {
         return
 vs[0
]
     }
     return
 ""

}
// Add adds the value to key.

// It appends to any existing values associated with key.

func
 (v Values) Add(key, value string
) {
    v[key] = append
(v[key], value)
}
```

這個定義向外部暴露了一個map的類型的變量，併且提供了一些能夠簡單操作這個map的方法。這個map的value字段是一個string的slice，所以這個Values是一個多維map。客戶端使用這個變量的時候可以使用map固有的一些操作(make，切片，m[key]等等)，也可以使用這里提供的操作方法，或者兩者併用，都是可以的：

```
gopl.io/ch6/urlvalues
m := url.Values{"lang"
: {"en"
}} // direct construction

m.Add("item"
, "1"
)
m.Add("item"
, "2"
)

fmt.Println(m.Get("lang"
)) // "en"

fmt.Println(m.Get("q"
))    // ""

fmt.Println(m.Get("item"
)) // "1"      (first value)

fmt.Println(m["item"
])     // "[1 2]"  (direct map access)


m = nil

fmt.Println(m.Get("item"
)) // ""

m.Add("item"
, "3"
)         // panic: assignment to entry in nil map
```

對Get的最後一次調用中，nil接收器的行爲即是一個空map的行爲。我們可以等價地將這個操作寫成Value(nil).Get("item")，但是如果你直接寫nil.Get("item")的話是無法通過編譯的，因爲nil的字面量編譯器無法判斷其準備類型。所以相比之下，最後的那行m.Add的調用就會產生一個panic，因爲他嚐試更新一個空map。

由於url.Values是一個map類型，併且間接引用了其key/value對，因此url.Values.Add對這個map里的元素做任何的更新、刪除操作對調用方都是可見的。實際上，就像在普通函數中一樣，雖然可以通過引用來操作內部值，但在方法想要脩改引用本身是不會影響原始值的，比如把他置爲nil，或者讓這個引用指向了其它的對象，調用方都不會受影響。(譯註：因爲傳入的是存儲了內存地址的變量，你改變這個變量是影響不了原始的變量的，想想C語言，是差不多的)

6.3. 通過嵌入結構體來擴展類型

來看看ColoredPoint這個類型：

```
gopl.io/ch6/coloredpoint
import
 "image/color"

type
 Point struct
{ X, Y float64
 }
type
 ColoredPoint struct
 {
    Point
    Color color.RGBA
}
```

我們完全可以將ColoredPoint定義爲一個有三個字段的struct，但是我們卻將Point這個類型嵌入到ColoredPoint來提供X和Y這兩個字段。像我們在4.4節中看到的那樣，內嵌可以使我們在定義ColoredPoint時得到一種句法上的簡寫形式，併使其包含Point類型所具有的一切字段，然後再定義一些自己的。如果我們想要的話，我們可以直接認爲通過嵌入的字段就是ColoredPoint自身的字段，而完全不需要在調用時指出Point，比如下面這樣。

```
var
 cp ColoredPoint
cp.X = 1

fmt.Println(cp.Point.X) // "1"

cp.Point.Y = 2

fmt.Println(cp.Y) // "2"
```

對於Point中的方法我們也有類似的用法，我們可以把ColoredPoint類型當作接收器來調用Point里的方法，即使ColoredPoint里沒有聲明這些方法：

```
red := color.RGBA{255
, 0
, 0
, 255
}
blue := color.RGBA{0
, 0
, 255
, 255
}
var
 p = ColoredPoint{Point{1
, 1
}, red}
var
 q = ColoredPoint{Point{5
, 4
}, blue}
fmt.Println(p.Distance(q.Point)) // "5"

p.ScaleBy(2
)
q.ScaleBy(2
)
fmt.Println(p.Distance(q.Point)) // "10"
```

Point類的方法也被引入了ColoredPoint。用這種方式，內嵌可以使我們定義字段特別多的複雜類型，我們可以將字段先按小類型分組，然後定義小類型的方法，之後再把它們組合起來。

讀者如果對基於類來實現面向對象的語言比較熟悉的話，可能會傾向於將Point看作一個基類，而ColoredPoint看作其子類或者繼承類，或者將ColoredPoint看作"is a" Point類型。但這是錯誤的理解。請註意上面例子中對Distance方法的調用。Distance有一個參數是Point類型，但q併不是一個Point類，所以盡管q有着Point這個內嵌類型，我們也必鬚要顯式地選擇它。嚐試直接傳q的話你會看到下面這樣的錯誤：

```
p.Distance(q) // compile error: cannot use q (ColoredPoint) as Point
```

一個ColoredPoint併不是一個Point，但他"has a"Point，併且它有從Point類里引入的Distance和ScaleBy方法。如果你喜歡從實現的角度來考慮問題，內嵌字段會指導編譯器去生成額外的包裝方法來委託已經聲明好的方法，和下面的形式是等價的：

```
func
 (p ColoredPoint) Distance(q Point) float64
 {
    return
 p.Point.Distance(q)
}

func
 (p *ColoredPoint) ScaleBy(factor float64
) {
    p.Point.ScaleBy(factor)
}
```

當Point.Distance被第一個包裝方法調用時，它的接收器值是p.Point，而不是p，當然了，在Point類的方法里，你是訪問不到ColoredPoint的任何字段的。

在類型中內嵌的匿名字段也可能是一個命名類型的指針，這種情況下字段和方法會被間接地引入到當前的類型中(譯註：訪問需要通過該指針指向的對象去取)。添加這一層間接關繫讓我們可以共享通用的結構併動態地改變對象之間的關繫。下面這個ColoredPoint的聲明內嵌了一個*Point的指針。

```
type
 ColoredPoint struct
 {
    *Point
    Color color.RGBA
}

p := ColoredPoint{&Point{1
, 1
}, red}
q := ColoredPoint{&Point{5
, 4
}, blue}
fmt.Println(p.Distance(*q.Point)) // "5"

q.Point = p.Point                 // p and q now share the same Point

p.ScaleBy(2
)
fmt.Println(*p.Point, *q.Point) // "{2 2} {2 2}"
```

一個struct類型也可能會有多個匿名字段。我們將ColoredPoint定義爲下面這樣：

```
type
 ColoredPoint struct
 {
    Point
    color.RGBA
}
```

然後這種類型的值便會擁有Point和RGBA類型的所有方法，以及直接定義在ColoredPoint中的方法。當編譯器解析一個選擇器到方法時，比如p.ScaleBy，它會首先去找直接定義在這個類型里的ScaleBy方法，然後找被ColoredPoint的內嵌字段們引入的方法，然後去找Point和RGBA的內嵌字段引入的方法，然後一直遞歸向下找。如果選擇器有二義性的話編譯器會報錯，比如你在同一級里有兩個同名的方法。

方法隻能在命名類型(像Point)或者指向類型的指針上定義，但是多虧了內嵌，有些時候我們給匿名struct類型來定義方法也有了手段。

下面是一個小trick。這個例子展示了簡單的cache，其使用兩個包級別的變量來實現，一個mutex互斥量(§9.2)和它所操作的cache：

```
var
 (
    mu sync.Mutex // guards mapping

    mapping = make
(map
[string
]string
)
)

func
 Lookup(key string
) string
 {
    mu.Lock()
    v := mapping[key]
    mu.Unlock()
    return
 v
}
```

下面這個版本在功能上是一致的，但將兩個包級吧的變量放在了cache這個struct一組內：

```
var
 cache = struct
 {
    sync.Mutex
    mapping map
[string
]string

}{
    mapping: make
(map
[string
]string
),
}


func
 Lookup(key string
) string
 {
    cache.Lock()
    v := cache.mapping[key]
    cache.Unlock()
    return
 v
}
```

我們給新的變量起了一個更具表達性的名字：cache。因爲sync.Mutex字段也被嵌入到了這個struct里，其Lock和Unlock方法也就都被引入到了這個匿名結構中了，這讓我們能夠以一個簡單明了的語法來對其進行加鎖解鎖操作。

6.4. 方法值和方法表達式

我們經常選擇一個方法，併且在同一個表達式里執行，比如常見的p.Distance()形式，實際上將其分成兩步來執行也是可能的。p.Distance叫作“選擇器”，選擇器會返迴一個方法"值"->一個將方法(Point.Distance)綁定到特定接收器變量的函數。這個函數可以不通過指定其接收器即可被調用；即調用時不需要指定接收器(譯註：因爲已經在前文中指定過了)，隻要傳入函數的參數即可：

```
p := Point{1
, 2
}
q := Point{4
, 6
}

distanceFromP := p.Distance        // method value

fmt.Println(distanceFromP(q))      // "5"

var
 origin Point                  // {0, 0}

fmt.Println(distanceFromP(origin)) // "2.23606797749979", sqrt(5)


scaleP := p.ScaleBy // method value

scaleP(2
)           // p becomes (2, 4)

scaleP(3
)           //      then (6, 12)

scaleP(10
)          //      then (60, 120)
```

在一個包的API需要一個函數值、且調用方希望操作的是某一個綁定了對象的方法的話，方法"值"會非常實用(=_=眞是繞)。舉例來說，下面例子中的time.AfterFunc這個函數的功能是在指定的延遲時間之後來執行一個(譯註：另外的)函數。且這個函數操作的是一個Rocket對象r

```
type
 Rocket struct
 { /* ... */
 }
func
 (r *Rocket) Launch() { /* ... */
 }
r := new
(Rocket)
time.AfterFunc(10
 * time.Second, func
() { r.Launch() })
```

直接用方法"值"傳入AfterFunc的話可以更爲簡短：

```
time.AfterFunc(10
 * time.Second, r.Launch)
```

譯註：省掉了上面那個例子里的匿名函數。

和方法"值"相關的還有方法表達式。當調用一個方法時，與調用一個普通的函數相比，我們必鬚要用選擇器(p.Distance)語法來指定方法的接收器。

當T是一個類型時，方法表達式可能會寫作T.f或者(*T).f，會返迴一個函數"值"，這種函數會將其第一個參數用作接收器，所以可以用通常(譯註：不寫選擇器)的方式來對其進行調用：

```
p := Point{1
, 2
}
q := Point{4
, 6
}

distance := Point.Distance   // method expression

fmt.Println(distance(p, q))  // "5"

fmt.Printf("%T\n"
, distance) // "func(Point, Point) float64"


scale := (*Point).ScaleBy
scale(&p, 2
)
fmt.Println(p)            // "{2 4}"

fmt.Printf("%T\n"
, scale) // "func(*Point, float64)"


// 譯註：這個Distance實際上是指定了Point對象爲接收器的一個方法func (p Point) Distance()，

// 但通過Point.Distance得到的函數需要比實際的Distance方法多一個參數，

// 即其需要用第一個額外參數指定接收器，後面排列Distance方法的參數。

// 看起來本書中函數和方法的區別是指有沒有接收器，而不像其他語言那樣是指有沒有返迴值。
```

當你根據一個變量來決定調用同一個類型的哪個函數時，方法表達式就顯得很有用了。你可以根據選擇來調用接收器各不相同的方法。下面的例子，變量op代表Point類型的addition或者subtraction方法，Path.TranslateBy方法會爲其Path數組中的每一個Point來調用對應的方法：

```
type
 Point struct
{ X, Y float64
 }

func
 (p Point) Add(q Point) Point { return
 Point{p.X + q.X, p.Y + q.Y} }
func
 (p Point) Sub(q Point) Point { return
 Point{p.X - q.X, p.Y - q.Y} }

type
 Path []Point

func
 (path Path) TranslateBy(offset Point, add bool
) {
    var
 op func
(p, q Point) Point
    if
 add {
        op = Point.Add
    } else
 {
        op = Point.Sub
    }
    for
 i := range
 path {
        // Call either path[i].Add(offset) or path[i].Sub(offset).

        path[i] = op(path[i], offset)
    }
}
```

6.5. 示例: Bit數組

Go語言里的集合一般會用map[T]bool這種形式來表示，T代表元素類型。集合用map類型來表示雖然非常靈活，但我們可以以一種更好的形式來表示它。例如在數據流分析領域，集合元素通常是一個非負整數，集合會包含很多元素，併且集合會經常進行併集、交集操作，這種情況下，bit數組會比map表現更加理想。(譯註：這里再補充一個例子，比如我們執行一個http下載任務，把文件按照16kb一塊劃分爲很多塊，需要有一個全局變量來標識哪些塊下載完成了，這種時候也需要用到bit數組)

一個bit數組通常會用一個無符號數或者稱之爲“字”的slice或者來表示，每一個元素的每一位都表示集合里的一個值。當集合的第i位被設置時，我們才說這個集合包含元素i。下面的這個程序展示了一個簡單的bit數組類型，併且實現了三個函數來對這個bit數組來進行操作：

```
gopl.io/ch6/intset
// An IntSet is a set of small non-negative integers.

// Its zero value represents the empty set.

type
 IntSet struct
 {
    words []uint64

}

// Has reports whether the set contains the non-negative value x.

func
 (s *IntSet) Has(x int
) bool
 {
    word, bit := x/64
, uint
(x%64
)
    return
 word < len
(s.words) && s.words[word]&(1
<<bit) != 0

}

// Add adds the non-negative value x to the set.

func
 (s *IntSet) Add(x int
) {
    word, bit := x/64
, uint
(x%64
)
    for
 word >= len
(s.words) {
```

```
            s.words = append
(s.words, 0
)
        }
        s.words[word] |= 1
     << bit
}

// UnionWith sets s to the union of s and t.

func
 (s *IntSet) UnionWith(t *IntSet) {
        for
 i, tword := range
 t.words {
            if
 i < len
(s.words) {
                s.words[i] |= tword
            } else
 {
                s.words = append
(s.words, tword)
            }
        }
}
```

因爲每一個字都有64個二進製位，所以爲了定位x的bit位，我們用了x/64的商作爲字的下標，併且用x%64得到的值作爲這個字內的bit的所在位置。UnionWith這個方法里用到了bit位的“或”邏輯操作符號|來一次完成64個元素的或計算。(在練習6.5中我們還會程序用到這個64位字的例子。)

當前這個實現還缺少了很多必要的特性，我們把其中一些作爲練習題列在本小節之後。但是有一個方法如果缺失的話我們的bit數組可能會比較難混：將IntSet作爲一個字符串來打印。這里我們來實現它，讓我們來給上面的例子添加一個String方法，類似2.5節中做的那樣：

```
// String returns the set as a string of the form "{1 2 3}".

func
 (s *IntSet) String() string
 {
    var
 buf bytes.Buffer
    buf.WriteByte('{'
)
    for
 i, word := range
 s.words {
        if
 word == 0
 {
            continue


        }
        for
 j := 0
; j < 64
; j++ {
            if
 word&(1
<<uint
(j)) != 0
 {
                if
 buf.Len() > len
("{"
) {
                    buf.WriteByte('}'
)
                }
                fmt.Fprintf(&buf, "%d"
, 64
*i+j)"}"
)}}
            }
        }
    }
    buf.WriteByte('}'
)
    return
 buf.String()
}
```

這里留意一下String方法，是不是和3.5.4節中的intsToString方法很相似；bytes.Buffer在String方法里經常這麼用。當你爲一個複雜的類型定義了一個String方法時，fmt包就會特殊對待這種類型的值，這樣可以讓這些類型在打印的時候看起來更加友好，而不是直接打印其原始的值。fmt會直接調用用戶定義的String方法。這種機製依賴於接口和類型斷言，在第7章中我們會詳細介紹。

現在我們就可以在實戰中直接用上面定義好的IntSet了：

```
var
 x, y IntSet
x.Add(1
)
x.Add(144
)
x.Add(9
)
fmt.Println(x.String()) // "{1 9 144}"


y.Add(9
)
y.Add(42
)
fmt.Println(y.String()) // "{9 42}"


x.UnionWith(&y)
fmt.Println(x.String()) // "{1 9 42 144}"

fmt.Println(x.Has(9
), x.Has(123
)) // "true false"
```

這里要註意：我們聲明的String和Has兩個方法都是以指針類型*IntSet來作爲接收器的，但實際上對於這兩個類型來說，把接收器聲明爲指針類型也沒什麼必要。不過另外兩個函數就不是這樣了，因爲另外兩個函數操作的是s.words對象，如果你不把接收器聲明爲指針對象，那麼實際操作的是拷貝對象，而不是原來的那個對象。因此，因爲我們的String方法定義在IntSet指針上，所以當我們的變量是IntSet類型而不是IntSet指針時，可能會有下面這樣讓人意外的情況：

```
fmt.Println(&x)         // "{1 9 42 144}"

fmt.Println(x.String()) // "{1 9 42 144}"

fmt.Println(x)          // "{[4398046511618 0 65536]}"
```

在第一個Println中，我們打印一個*IntSet的指針，這個類型的指針確實有自定義的String方法。第二Println，我們直接調用了x變量的String()方法；這種情況下編譯器會隱式地在x前插入&操作符，這樣相當遠我們還是調用的IntSet指針的String方法。在第三個Println中，因爲IntSet類型沒有String方法，所以Println方法會直接以原始的方式理解併打印。所以在這種情況下&符號是不能忘的。在我們這種場景下，你把String方法綁定到IntSet對象上，而不是IntSet指針上可能會更合適一些，不過這也需要具體問題具體分析。

練習6.1: 爲bit數組實現下面這些方法

```
func
 (*IntSet) Len() int
      // return the number of elements

func
 (*IntSet) Remove(x int
)  // remove x from the set

func
 (*IntSet) Clear()        // remove all elements from the set

func
 (*IntSet) Copy() *IntSet // return a copy of the set
```

練習6.2: 定義一個變參方法(*IntSet).AddAll(...int)，這個方法可以爲一組IntSet值求和，比如s.AddAll(1,2,3)。

練習6.3: (*IntSet).UnionWith會用|操作符計算兩個集合的交集，我們再爲IntSet實現另外的幾個函數IntersectWith(交集：元素在A集合B集合均出現),DifferenceWith(差集：元素出現在A集合，未出現在B集合),SymmetricDifference(併差集：元素出現在A但沒有出現在B，或者出現在B沒有出現在A)。 練習6.4: 實現一個Elems方法，返迴集合中的所有元素，用於做一些range之類的遍歷操作。

練習6.5: 我們這章定義的IntSet里的每個字都是用的uint64類型，但是64位的數值可能在32位的平台上不高效。脩改程序，使其使用uint類型，這種類型對於32位平台來説更合適。當然了，這里我們可以不用簡單粗暴地除64，可以定義一個常量來決定是用32還是64，這里你可能會用到平台的自動判斷的一個智能表達式：32 << (^uint(0) >> 63)

6.6. 封裝

一個對象的變量或者方法如果對調用方是不可見的話，一般就被定義爲“封裝”。封裝有時候也被叫做信息隱藏，同時也是面向對象編程最關鍵的一個方面。

Go語言隻有一種控製可見性的手段：大寫首字母的標識符會從定義它們的包中被導出，小寫字母的則不會。這種限製包內成員的方式同樣適用於struct或者一個類型的方法。因而如果我們想要封裝一個對象，我們必鬚將其定義爲一個struct。

這也就是前面的小節中IntSet被定義爲struct類型的原因，盡管它隻有一個字段：

```
type
 IntSet struct
 {
    words []uint64

}
```

當然，我們也可以把IntSet定義爲一個slice類型，盡管這樣我們就需要把代碼中所有方法里用到的s.words用*s替換掉了：

```
type
 IntSet []uint64
```

盡管這個版本的IntSet在本質上是一樣的，他也可以允許其它包中可以直接讀取併編輯這個slice。換句話說，相對*s這個表達式會出現在所有的包中，s.words隻需要在定義IntSet的包中出現(譯註：所以還是推薦後者吧的意思)。

這種基於名字的手段使得在語言中最小的封裝單元是package，而不是像其它語言一樣的類型。一個struct類型的字段對同一個包的所有代碼都有可見性，無論你的代碼是寫在一個函數還是一個方法里。

封裝提供了三方面的優點。首先，因爲調用方不能直接脩改對象的變量值，其隻需要關註少量的語句併且隻要弄懂少量變量的可能的值即可。

第二，隱藏實現的細節，可以防止調用方依賴那些可能變化的具體實現，這樣使設計包的程序員在不破壞對外的api情況下能得到更大的自由。

把bytes.Buffer這個類型作爲例子來考慮。這個類型在做短字符串疊加的時候很常用，所以在設計的時候可以做一些預先的優化，比如提前預留一部分空間，來避免反複的內存分配。又因爲Buffer是一個struct類型，這些額外的空間可以用附加的字節數組來保存，且放在一個小寫字母開頭的字段中。這樣在外部的調用方隻能看到性能的提陞，但併不會得到這個附加變量。Buffer和其增長算法我們列在這里，爲了簡潔性稍微做了一些精簡：

```
type
 Buffer struct
 {
    buf     []byte

    initial [64
]byte

    /* ... */

}

// Grow expands the buffer's capacity, if necessary,

// to guarantee space for another n bytes. [...]

func
 (b *Buffer) Grow(n int
) {
    if
 b.buf == nil
 {
        b.buf = b.initial[:0
] // use preallocated space initially

    }
    if
 len
(b.buf)+n > cap
(b.buf) {
        buf := make
([]byte
, b.Len(), 2
*cap
(b.buf) + n)
        copy
(buf, b.buf)
        b.buf = buf
    }
}
```

封装的第三個優點也是最重要的優點，是阻止了外部調用方對對象內部的值任意地進行脩改。因爲對象內部變量隻可以被同一個包內的函數脩改，所以包的作者可以讓這些函數確保對象內部的一些值的不變性。比如下面的Counter類型允許調用方來增加counter變量的值，併且允許將這個值reset爲0，但是不允許隨便設置這個值(譯註：因爲壓根就訪問不到)：

```
type
 Counter struct
 { n int
 }
func
 (c *Counter) N() int
     { return
 c.n }
func
 (c *Counter) Increment() { c.n++ }
func
 (c *Counter) Reset()     { c.n = 0
 }
```

隻用來訪問或脩改內部變量的函數被稱爲setter或者getter，例子如下，比如log包里的Logger類型對應的一些函數。在命名一個getter方法時，我們通常會省略掉前面的Get前綴。這種簡潔上的偏好也可以推廣到各種類型的前綴比如Fetch，Find或者Lookup。

```
package
 log
type
 Logger struct
 {
    flags  int

    prefix string

    // ...

}
func
 (l *Logger) Flags() int

func
 (l *Logger) SetFlags(flag int
)
func
 (l *Logger) Prefix() string

func
 (l *Logger) SetPrefix(prefix string
)
```

Go的編碼風格不禁止直接導出字段。當然，一旦進行了導出，就沒有辦法在保證API兼容的情況下去除對其的導出，所以在一開始的選擇一定要經過深思熟慮併且要考慮到包內部的一些不變量的保證，未來可能的變化，以及調用方的代碼質量是否會因爲包的一點脩改而變差。

封装併不總是理想的。 雖然封装在有些情況是必要的，但有時候我們也需要暴露一些內部內容，比如：time.Duration將其表現暴露爲一個int64數字的納秒，使得我們可以用一般的數值操作來對時間進行對比，甚至可以定義這種類型的常量：

```
const
 day = 24
 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

另一個例子，將IntSet和本章開頭的geometry.Path進行對比。Path被定義爲一個slice類型，這允許其調用slice的字面方法來對其內部的points用range進行迭代遍歷；在這一點上，IntSet是沒有辦法讓你這麼做的。

這兩種類型決定性的不同：geometry.Path的本質是一個坐標點的序列，不多也不少，我們可以預見到之後也併不會給他增加額外的字段，所以在geometry包中將Path暴露爲一個slice。相比之下，IntSet僅僅是在這里用了一個[]uint64的slice。這個類型還可以用[]uint類型來表示，或者我們甚至可以用其它完全不同的占用更小內存空間的東西來表示這個集合，所以我們可能還會需要額外的字段來在這個類型中記録元素的個數。也正是因爲這些原因，我們讓IntSet對調用方透明。

在這章中，我們學到了如何將方法與命名類型進行組合，併且知道了如何調用這些方法。盡管方法對於OOP編程來説至關重要，但他們隻是OOP編程里的半邊天。爲了完成OOP，我們還需要接口。Go里的接口會在下一章中介紹。

第七章 接口

接口類型是對其它類型行爲的抽象和概括；因爲接口類型不會和特定的實現細節綁定在一起，通過這種抽象的方式我們可以讓我們的函數更加靈活和更具有適應能力。

很多面向對象的語言都有相似的接口概念，但Go語言中接口類型的獨特之處在於它是滿足隱式實現的。也就是說，我們沒有必要對於給定的具體類型定義所有滿足的接口類型；簡單地擁有一些必需的方法就足夠了。這種設計可以讓你創建一個新的接口類型滿足已經存在的具體類型卻不會去改變這些類型的定義；當我們使用的類型來自於不受我們控製的包時這種設計尤其有用。

在本章，我們會開始看到接口類型和值的一些基本技巧。順着這種方式我們將學習幾個來自標準庫的重要接口。很多Go程序中都盡可能多的去使用標準庫中的接口。最後,我們會在(§7.10)看到類型斷言的知識，在(§7.13)看到類型開關的使用併且學到他們是怎樣讓不同的類型的概括成爲可能。

7.1. 接口約定

目前爲止，我們看到的類型都是具體的類型。一個具體的類型可以準確的描述它所代表的值併且展示出對類型本身的一些操作方式就像數字類型的算術操作，切片類型的索引、附加和取范圍操作。具體的類型還可以通過它的方法提供額外的行爲操作。總的來說，當你拿到一個具體的類型時你就知道它的本身是什麼和你可以用它來做什麼。

在Go語言中還存在着另外一種類型：接口類型。接口類型是一種抽象的類型。它不會暴露出它所代表的對象的內部值的結構和這個對象支持的基礎操作的集合；它們隻會展示出它們自己的方法。也就是說當你有看到一個接口類型的值時，你不知道它是什麼，唯一知道的就是可以通過它的方法來做什麼。

在本書中，我們一直使用兩個相似的函數來進行字符串的格式化：fmt.Printf它會把結果寫到標準輸出和fmt.Sprintf它會把結果以字符串的形式返迴。得益於使用接口，我們不必可悲的因爲返迴結果在使用方式上的一些淺顯不同就必需把格式化這個最睏難的過程複製一份。實際上，這兩個函數都使用了另一個函數fmt.Fprintf來進行封裝。fmt.Fprintf這個函數對它的計算結果會被怎麼使用是完全不知道的。

```
package
 fmt
func
 Fprintf(w io.Writer, format string
, args ...interface
{}) (int
, error)
func
 Printf(format string
, args ...interface
{}) (int
, error) {
    return
 Fprintf(os.Stdout, format, args...)
}
func
 Sprintf(format string
, args ...interface
{}) string
 {
    var
 buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return
 buf.String()
}
```

Fprintf的前綴F表示文件(File)也表明格式化輸出結果應該被寫入第一個參數提供的文件中。在Printf函數中的第一個參數os.Stdout是*os.File類型；在Sprintf函數中的第一個參數&buf是一個指向可以寫入字節的內存緩衝區，然而它 併不是一個文件類型盡管它在某種意義上和文件類型相似。

即使Fprintf函數中的第一個參數也不是一個文件類型。它是io.Writer類型這是一個接口類型定義如下：

```
package
 io
// Writer is the interface that wraps the basic Write method.

type
 Writer interface
 {
    // Write writes len(p) bytes from p to the underlying data stream.

    // It returns the number of bytes written from p (0 <= n <= len(p))

    // and any error encountered that caused the write to stop early.

    // Write must return a non-nil error if it returns n < len(p).

    // Write must not modify the slice data, even temporarily.

    //

    // Implementations must not retain p.

    Write(p []byte
) (n int
, err error)
}
```

io.Writer類型定義了函數Fprintf和這個函數調用者之間的約定。一方面這個約定需要調用者提供具體類型的值就像*os.File和*bytes.Buffer，這些類型都有一個特定籤名和行爲的Write的函數。另一方面這個約定保證了Fprintf接受任何滿足io.Writer接口的值都可以工作。Fprintf函數可能沒有假定寫入的是一個文件或是一段內存，而是寫入一個可以調用Write函數的值。

因爲fmt.Fprintf函數沒有對具體操作的值做任何假設而是僅僅通過io.Writer接口的約定來保證行爲，所以第一個參數可以安全地傳入一個任何具體類型的值隻需要滿足io.Writer接口。一個類型可以自由的使用另一個滿足相同接口的類型來進行替換被稱作可替換性(LSP里氏替換)。這是一個面向對象的特徵。

讓我們通過一個新的類型來進行校驗，下面*ByteCounter類型里的Write方法，僅僅在丟失寫向它的字節前統計它們的長度。(在這個+=賦值語句中，讓len(p)的類型和*c的類型匹配的轉換是必髮的。)

```
// gopl.io/ch7/bytecounter

type
 ByteCounter int

func
 (c *ByteCounter) Write(p []byte
) (int
, error) {
    *c += ByteCounter(len
(p)) // convert int to ByteCounter

    return
 len
(p), nil

}
```

因爲*ByteCounter滿足io.Writer的約定，我們可以把它傳入Fprintf函數中；Fprintf函數執行字符串格式化的過程不會去關註ByteCounter正確的纍加結果的長度。

```
var
 c ByteCounter
c.Write([]byte
("hello"
))
fmt.Println(c) // "5", = len("hello")

c = 0
 // reset the counter

var
 name = "Dolly"

fmt.Fprintf(&c, "hello, %s"
, name)
fmt.Println(c) // "12", = len("hello, Dolly")
```

除了io.Writer這個接口類型，還有另一個對fmt包很重要的接口類型。Fprintf和Fprintln函數向類型提供了一種控製它們值輸出的途徑。在2.5節中，我們爲Celsius類型提供了一個String方法以便於可以打印成這樣"100°C"，在6.5節中我們給*IntSet添加一個String方法，這樣集合可以用傳統的符號來進行表示就像"{1 2 3}"。給一個類型定義String方法，可以讓它滿足最廣泛使用之一的接口類型fmt.Stringer：

```
package
 fmt
// The String method is used to print values passed

// as an operand to any format that accepts a string

// or to an unformatted printer such as Print.

type
 Stringer interface
 {
    String() string

}
```

我們會在7.10節解釋fmt包怎麼發現哪些值是滿足這個接口類型的。

練習7.1:使用來自ByteCounter的思路，實現一個針對對單詞和行數的計數器。你會發現bufio.ScanWords非常的有用。

練習7.2:寫一個帶有如下函數籤名的函數CountingWriter，傳入一個io.Writer接口類型，返迴一個新的Writer類型把原來的Writer封裝在里面和一個表示寫入新的Writer字節數的int64類型指針

```
func
 CountingWriter(w io.Writer) (io.Writer, *int64
)
```

練習7.3:爲在gopl.io/ch4/treesort (§4.4)的*tree類型實現一個String方法去展示tree類型的值序列。

7.2. 接口類型

接口類型具體描述了一繫列方法的集合，一個實現了這些方法的具體類型是這個接口類型的實例。

io.Writer類型是用的最廣泛的接口之一，因爲它提供了所有的類型寫入bytes的抽象，包括文件類型，內存緩衝區，網絡鏈接，HTTP客戶端，壓縮工具，哈希等等。io包中定義了很多其它有用的接口類型。Reader可以代表任意可以讀取bytes的類型，Closer可以是任意可以關閉的值，例如一個文件或是網絡鏈接。（到現在你可能註意到了很多Go語言中單方法接口的命名習慣）

```
package
 io
type
 Reader interface
 {
    Read(p []byte
) (n int
, err error)
}
type
 Closer interface
 {
    Close() error
}
```

在往下看，我們發現有些新的接口類型通過組合已經有的接口來定義。下面是兩個例子：

```
type
 ReadWriter interface
 {
    Reader
    Writer
}
type
 ReadWriteCloser interface
 {
    Reader
    Writer
    Closer
}
```

上面用到的語法和結構內嵌相似，我們可以用這種方式以一個簡寫命名另一個接口，而不用聲明它所有的方法。這種方式本稱爲接口內嵌。盡管略失簡潔，我們可以像下面這樣，不使用內嵌來聲明io.Writer接口。

```
type
 ReadWriter interface
 {
    Read(p []byte
) (n int
, err error)
    Write(p []byte
) (n int
, err error)
}
```

或者甚至使用種混合的風格：

```
type
 ReadWriter interface
 {
    Read(p []byte
) (n int
, err error)
    Writer
}
```

上面3種定義方式都是一樣的效果。方法的順序變化也沒有影響，唯一重要的就是這個集合里面的方法。

練習7.4:strings.NewReader函數通過讀取一個string參數返迴一個滿足io.Reader接口類型的值（和其它值）。實現一個簡單版本的NewReader，併用它來構造一個接收字符串輸入的HTML解析器（§5.2）

練習7.5:io包里面的LimitReader函數接收一個io.Reader接口類型的r和字節數n，併且返迴另一個從r中讀取字節但是當讀完n個字節後就表示讀到文件結束的Reader。實現這個LimitReader函數：

```
func
 LimitReader(r io.Reader, n int64
) io.Reader
```

7.3. 實現接口的條件

一個類型如果擁有一個接口需要的所有方法，那麼這個類型就實現了這個接口。例如，*os.File類型實現了io.Reader，Writer，Closer，和ReadWriter接口。* bytes.Buffer實現了Reader，Writer，和ReadWriter這些接口，但是它沒有實現Closer接口因爲它不具有Close方法。Go的程序員經常會簡要的把一個具體的類型描述成一個特定的接口類型。舉個例子，*bytes.Buffer是io.Writer；* os.Files是io.ReadWriter。

接口指定的規則非常簡單：表達一個類型屬於某個接口隻要這個類型實現這個接口。所以：

```
var
 w io.Writer
w = os.Stdout           // OK: *os.File has Write method

w = new
(bytes.Buffer)   // OK: *bytes.Buffer has Write method

w = time.Second         // compile error: time.Duration lacks Write method


var
 rwc io.ReadWriteCloser
rwc = os.Stdout         // OK: *os.File has Read, Write, Close methods

rwc = new
(bytes.Buffer) // compile error: *bytes.Buffer lacks Close method
```

這個規則甚至適用於等式右邊本身也是一個接口類型

```
w = rwc                 // OK: io.ReadWriteCloser has Write method

rwc = w                 // compile error: io.Writer lacks Close method
```

因爲ReadWriter和ReadWriteCloser包含所有Writer的方法，所以任何實現了ReadWriter和ReadWriteCloser的類型必定也實現了Writer接口

在進一步學習前，必鬚先解釋表示一個類型持有一個方法當中的細節。迴想在6.2章中，對於每一個命名過的具體類型T；它一些方法的接收者是類型T本身然而另一些則是一個*T的指針。還記得在T類型的參數上調用一個* T的方法是合法的，隻要這個參數是一個變量；編譯器隱式的獲取了它的地址。但這僅僅是一個語法醣：T類型的值不擁有所有*T指針的方法，那這樣它就可能隻實現更少的接口。

舉個例子可能會更清晰一點。在第6.5章中，IntSet類型的String方法的接收者是一個指針類型，所以我們不能在一個不能尋址的IntSet值上調用這個方法：

```
type
 IntSet struct
 { /* ... */
 }
func
 (*IntSet) String() string

var
 _ = IntSet{}.String() // compile error: String requires *IntSet receiver
```

但是我們可以在一個IntSet值上調用這個方法：

```
var
 s IntSet
var
 _ = s.String() // OK: s is a variable and &s has a String method
```

然而，由於隻有*IntSet類型有String方法，所有也隻有*IntSet類型實現了fmt.Stringer接口：

```
var
 _ fmt.Stringer = &s // OK

var
 _ fmt.Stringer = s  // compile error: IntSet lacks String method
```

12.8章包含了一個打印出任意值的所有方法的程序，然後可以使用godoc -analysis=type tool(§10.7.4)展示每個類型的方法和具體類型和接口之間的關繫

就像信封封裝和隱藏信件起來一樣，接口類型封裝和隱藏具體類型和它的值。即使具體類型有其它的方法也隻有接口類型暴露出來的方法會被調用到：

```
os.Stdout.Write([]byte
("hello"
)) // OK: *os.File has Write method

os.Stdout.Close()                // OK: *os.File has Close method


var
 w io.Writer
w = os.Stdout
w.Write([]byte
("hello"
)) // OK: io.Writer has Write method

w.Close()                // compile error: io.Writer lacks Close method
```

一個有更多方法的接口類型，比如io.ReadWriter，和少一些方法的接口類型,例如io.Reader，進行對比；更多方法的接口類型會告訴我們更多關於它的值持有的信息，併且對實現它的類型要求更加嚴格。那麼關於interface{}類型，它沒有任何方法，請講出哪些具體的類型實現了它？

這看上去好像沒有用，但實際上interface{}被稱爲空接口類型是不可或缺的。因爲空接口類型對實現它的類型沒有要求，所以我們可以將任意一個值賦給空接口類型。

```
var
 any interface
{}
any = true

any = 12.34

any = "hello"

any = map
[string
]int
{"one"
: 1
}
any = new
(bytes.Buffer)
```

盡管不是很明顯，從本書最早的的例子中我們就已經在使用空接口類型。它允許像fmt.Println或者5.7章中的errorf函數接受任何類型的參數。

對於創建的一個interface{}值持有一個boolean，float，string，map，pointer，或者任意其它的類型；我們當然不能直接對它持有的值做操作，因爲interface{}沒有任何方法。我們會在7.10章中學到一種用類型斷言來獲取interface{}中值的方法。

因爲接口實現隻依賴於判斷的兩個類型的方法，所以沒有必要定義一個具體類型和它實現的接口之間的關繫。也就是說，嚐試文檔化和斷言這種關繫幾乎沒有用，所以併沒有通過程序強製定義。下面的定義在編譯期斷言一個*bytes.Buffer的值實現了io.Writer接口類型:

```
// *bytes.Buffer must satisfy io.Writer

var
 w io.Writer = new
(bytes.Buffer)
```

因爲任意*bytes.Buffer的值，甚至包括nil通過(*bytes.Buffer)(nil進行顯示的轉換都實現了這個接口，所以我們不必分配一個新的變量。併且因爲我們絶不會引用變量w，我們可以使用空標識符來來進行代替。總的看，這些變化可以讓我們得到一個更樸素的版本：

```
// *bytes.Buffer must satisfy io.Writer

var
 _ io.Writer = (*bytes.Buffer)(nil
)
```

非空的接口類型比如io.Writer經常被指針類型實現，尤其當一個或多個接口方法像Write方法那樣隱式的給接收者帶來變化的時候。一個結構體的指針是非常常見的承載方法的類型。

但是併不意味着隻有指針類型滿足接口類型，甚至連一些有設置方法的接口類型也可能會被Go語言中其它的引用類型實現。我們已經看過slice類型的方法(geometry.Path, §6.1)和map類型的方法(url.Values, §6.2.1)，後面還會看到函數類型的方法的例子(http.HandlerFunc, §7.7)。甚至基本的類型也可能會實現一些接口；就如我們在7.4章中看到的time.Duration類型實現了fmt.Stringer接口。

一個具體的類型可能實現了很多不相關的接口。考慮在一個組織出售數字文化產品比如音樂，電影和書籍的程序中可能定義了下列的具體類型：

```
Album
Book
Movie
Magazine
Podcast
TVEpisode
Track
```

我們可以把每個抽象的特點用接口來表示。一些特性對於所有的這些文化產品都是共通的，例如標題，創作日期和作者列表。

```
type
 Artifact interface
 {
    Title() string

    Creators() []string

    Created() time.Time
}
```

其它的一些特性隻對特定類型的文化產品才有。和文字排版特性相關的隻有books和magazines，還有隻有movies和TV劇集和屏幕分辨率相關。

```
type
 Text interface
 {
    Pages() int

    Words() int

    PageSize() int

}
type
 Audio interface
 {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string
 // e.g., "MP3", "WAV"

}
type
 Video interface
 {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string
 // e.g., "MP4", "WMV"

    Resolution() (x, y int
)
}
```

這些接口不止是一種有用的方式來分組相關的具體類型和表示他們之間的共同特定。我們後面可能會發現其它的分組。舉例，如果我們發現我們需要以同樣的方式處理Audio和Video，我們可以定義一個Streamer接口來代表它們之間相同的部分而不必對已經存在的類型做改變。

```
type
 Streamer interface
 {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string

}
```

每一個具體類型的組基於它們相同的行爲可以表示成一個接口類型。不像基於類的語言，他們一個類實現的接口集合需要進行顯式的定義，在Go語言中我們可以在需要的時候定義一個新的抽象或者特定特點的組，而不需要脩改具體類型的定義。當具體的類型來自不同的作者時這種方式會特別有用。當然也確實沒有必要在具體的類型中指出這些共性。

7.4. flag.Value接口

在本章，我們會學到另一個標準的接口類型flag.Value是怎麼幫助命令行標記定義新的符號的。思考下面這個會休眠特定時間的程序：

```
// gopl.io/ch7/sleep

var
 period = flag.Duration("period"
, 1
*time.Second, "sleep period"
)

func
 main() {
    flag.Parse()
    fmt.Printf("Sleeping for %v..."
, *period)
    time.Sleep(*period)
    fmt.Println()
}
```

在它休眠前它會打印出休眠的時間週期。fmt包調用time.Duration的String方法打印這個時間週期是以用戶友好的註解方式，而不是一個納秒數字：

```
$ go build gopl.io/ch7/sleep
$ ./sleep
Sleeping for 1s...
```

默認情況下，休眠週期是一秒，但是可以通過 -period 這個命令行標記來控製。flag.Duration函數創建一個time.Duration類型的標記變量併且允許用戶通過多種用戶友好的方式來設置這個變量的大小，這種方式還包括和String方法相同的符號排版形式。這種對稱設計使得用戶交互良好。

```
$ ./sleep -period 50ms
Sleeping for 50ms...
$ ./sleep -period 2m30s
Sleeping for 2m30s...
$ ./sleep -period 1.5h
Sleeping for 1h30m0s...
$ ./sleep -period "1 day"
invalid value "1 day" for flag -period: time: invalid duration 1 day
```

因爲時間週期標記值非常的有用，所以這個特性被構建到了flag包中；但是我們爲我們自己的數據類型定義新的標記符號是簡單容易的。我們隻需要定義一個實現flag.Value接口的類型，如下：

```
package
 flag

// Value is the interface to the value stored in a flag.

type
 Value interface
 {
    String() string

    Set(string
) error
}
```

String方法格式化標記的值用在命令行幫組消息中；這樣每一個flag.Value也是一個fmt.Stringer。Set方法解析它的字符串參數併且更新標記變量的值。實際上，Set方法和String是兩個相反的操作，所以最好的辦法就是對他們使用相同的註解方式。

讓我們定義一個允許通過攝氏度或者華氏溫度變換的形式指定溫度的celsiusFlag類型。註意celsiusFlag內嵌了一個Celsius類型(§2.5)，因此不用實現本身就已經有String方法了。爲了實現flag.Value，我們隻需要定義Set方法：

```
// gopl.io/ch7/tempconv

// *celsiusFlag satisfies the flag.Value interface.

type
 celsiusFlag struct
{ Celsius }

func
 (f *celsiusFlag) Set(s string
) error {
    var
 unit string

    var
 value float64

    fmt.Sscanf(s, "%f%s"
, &value, &unit) // no error check needed

    switch
 unit {
    case
 "C"
, "°C"
:
        f.Celsius = Celsius(value)
        return
 nil

    case
 "F"
, "°F"
:
        f.Celsius = FToC(Fahrenheit(value))
        return
 nil

    }
    return
 fmt.Errorf("invalid temperature %q"
, s)
}
```

調用fmt.Sscanf函數從輸入s中解析一個浮點數（value）和一個字符串（unit）。雖然通常必鬚檢查Sscanf的錯誤返迴，但是在這個例子中我們不需要因爲如果有錯誤發生，就沒有switch case會匹配到。

下面的CelsiusFlag函數將所有邏輯都封裝在一起。它返迴一個內嵌在celsiusFlag變量f中的Celsius指針給調用者。Celsius字段是一個會通過Set方法在標記處理的過程中更新的變量。調用Var方法將標記加入應用的命令行標記集合中，有異常複雜命令行接口的全局變量flag.CommandLine.Programs可能有幾個這個類型的變量。調用Var方法將一個*celsiusFlag參數賦值給一個flag.Value參數*,

導致編譯器去檢查 celsiusFlag是否有必鬚的方法。

```
// CelsiusFlag defines a Celsius flag with the specified name,

// default value, and usage, and returns the address of the flag variable.

// The flag argument must have a quantity and a unit, e.g., "100C".

func
 CelsiusFlag(name string
, value Celsius, usage string
) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return
 &f.Celsius
}
```

現在我們可以開始在我們的程序中使用新的標記：

```
// gopl.io/ch7/tempflag

var
 temp = tempconv.CelsiusFlag("temp"
, 20.0
, "the temperature"
)

func
 main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

下面是典型的場景：

```
$ go build gopl.io/ch7/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
invalid value "273.15K" for flag -temp: invalid temperature "273.15K"
Usage of ./tempflag:
  -temp value
        the temperature (default 20°C)
$ ./tempflag -help
Usage of ./tempflag:
  -temp value
        the temperature (default 20°C)
```

練習 7.6： 對tempFlag加入支持開爾文溫度。

練習 7.7： 解釋爲什麽幫助信息在它的默認值是20.0沒有包含°C的情況下輸出了°C。

7.5. 接口值

概念上講一個接口的值，接口值，由兩個部分組成，一個具體的類型和那個類型的值。它們被稱爲接口的動態類型和動態值。對於像Go語言這種靜態類型的語言，類型是編譯期的概念；因此一個類型不是一個值。在我們的概念模型中，一些提供每個類型信息的值被稱爲類型描述符，比如類型的名稱和方法。在一個接口值中，類型部分代表與之相關類型的描述符。

下面4個語句中，變量w得到了3個不同的值。（開始和最後的值是相同的）

```
var
 w io.Writer
w = os.Stdout
w = new
(bytes.Buffer)
w = nil
```

讓我們進一步觀察在每一個語句後的w變量的值和動態行爲。第一個語句定義了變量w:

```
var
 w io.Writer
```

在Go語言中，變量總是被一個定義明確的值初始化，即使接口類型也不例外。對於一個接口的零值就是它的類型和值的部分都是nil（圖7.1）。

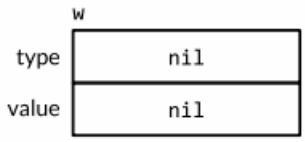


Figure 7.1. A nil interface value.

一個接口值基於它的動態類型被描述爲空或非空，所以這是一個空的接口值。你可以通過使用w==nil或者w!=nil來判讀接口值是否爲空。調用一個空接口值上的任意方法都會產生panic:

```
w.Write([]byte
("hello"
)) // panic: nil pointer dereference
```

第二個語句將一個*os.File類型的值賦給變量w:

```
w = os.Stdout
```

這個賦值過程調用了一個具體類型到接口類型的隱式轉換，這和顯式的使用io.Writer(os.Stdout)是等價的。這類轉換不管是顯式的還是隱式的，都會刻畫出操作到的類型和值。這個接口值的動態類型被設爲*os.Stdout指針的類型描述符，它的動態值持有os.Stdout的拷貝；這是一個代表處理標準輸出的os.File類型變量的指針（圖7.2）。

Figure 7.2. An interface value containing an `*os.File` pointer.

調用一個包含*os.File類型指針的接口值的Write方法，使得(*os.File).Write方法被調用。這個調用輸出“hello”。

```
w.Write([]byte
("hello"
)) // "hello"
```

通常在編譯期，我們不知道接口值的動態類型是什麼，所以一個接口上的調用必鬚使用動態分配。因爲不是直接進行調用，所以編譯器必鬚把代碼生成在類型描述符的方法Write上，然後間接調用那個地址。這個調用的接收者是一個接口動態值的拷貝，os.Stdout。效果和下面這個直接調用一樣：

```
os.Stdout.Write([]byte
("hello"
)) // "hello"
```

第三個語句給接口值賦了一個*bytes.Buffer類型的值

```
w = new
(bytes.Buffer)
```

現在動態類型是*bytes.Buffer併且動態值是一個指向新分配的緩衝區的指針（圖7.3）。

Figure 7.3. An interface value containing a `*bytes.Buffer` pointer.

Write方法的調用也使用了和之前一樣的機製：

```
w.Write([]byte
("hello"
)) // writes "hello" to the bytes.Buffers
```

這次類型描述符是*bytes.Buffer，所以調用了(*bytes.Buffer).Write方法，併且接收者是該緩衝區的地址。這個調用把字符串“hello”添加到緩衝區中。

最後，第四個語句將nil賦給了接口值：

```
w = nil
```

這個重置將它所有的部分都設爲nil值，把變量w恢複到和它之前定義時相同的狀態圖，在圖7.1中可以看到。

一個接口值可以持有任意大的動態值。例如，表示時間實例的time.Time類型，這個類型有幾個對外不公開的字段。我們從它上面創建一個接口值，

```
var
 x interface
{} = time.Now()
```

結果可能和圖7.4相似。從概念上講，不論接口值多大，動態值總是可以容下它。（這隻是一個概念上的模型；具體的實現可能會非常不同）

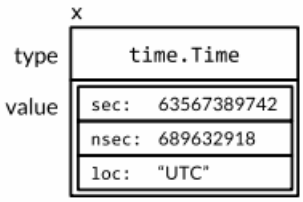


Figure 7.4. An interface value holding a `time.Time` struct.

接口值可以使用＝＝和！＝來進行比較。兩個接口值相等僅當它們都是nil值或者它們的動態類型相同併且動態值也根據這個動態類型的＝＝操作相等。因爲接口值是可比較的，所以它們可以用在map的鍵或者作爲switch語句的操作數。

然而，如果兩個接口值的動態類型相同，但是這個動態類型是不可比較的（比如切片），將它們進行比較就會失敗併且panic:

```
var
 x interface
{} = []int
{1
, 2
, 3
}
fmt.Println(x == x) // panic: comparing uncomparable type []int
```

考慮到這點，接口類型是非常與衆不同的。其它類型要麽是安全的可比較類型（如基本類型和指針）要麽是完全不可比較的類型（如切片，映射類型，和函數），但是在比較接口值或者包含了接口值的聚合類型時，我們必鬚要意識到潛在的panic。同樣的風險也存在於使用接口作爲map的鍵或者switch的操作數。隻能比較你非常確定它們的動態值是可比較類型的接口值。

當我們處理錯誤或者調試的過程中，得知接口值的動態類型是非常有幫助的。所以我們使用fmt包的%T動作:

```
var
 w io.Writer
fmt.Printf("%T\n"
, w) // "<nil>"

w = os.Stdout
fmt.Printf("%T\n"
, w) // "*os.File"

w = new
(bytes.Buffer)
fmt.Printf("%T\n"
, w) // "*bytes.Buffer"
```

在fmt包內部，使用反射來獲取接口動態類型的名稱。我們會在第12章中學到反射相關的知識。

7.5.1. 警告：一個包含nil指針的接口不是nil接口

一個不包含任何值的nil接口值和一個剛好包含nil指針的接口值是不同的。這個細微區別產生了一個容易絆倒每個Go程序員的陷阱。

思考下面的程序。當debug變量設置爲true時，main函數會將f函數的輸出收集到一個bytes.Buffer類型中。

```
const
 debug = true


func
 main() {
    var
 buf *bytes.Buffer
    if
 debug {
        buf = new
(bytes.Buffer) // enable collection of output

    }
    f(buf) // NOTE:
 subtly incorrect!

    if
 debug {
        // ...use buf...

    }
}

// If out is non-nil, output will be written to it.

func
 f(out io.Writer) {
    // ...do something...

    if
 out != nil
 {
        out.Write([]byte
("done!\n"
))
    }
}
```

我們可能會預計當把變量debug設置爲false時可以禁止對輸出的收集，但是實際上在out.Write方法調用時程序發生了panic：

```
if
 out != nil
 {
    out.Write([]byte
("done!\n"
)) // panic: nil pointer dereference

}
```

當main函數調用函數f時，它給f函數的out參數賦了一個*bytes.Buffer的空指針，所以out的動態值是nil。然而，它的動態類型是*bytes.Buffer，意思就是out變量是一個包含空指針值的非空接口（如圖7.5），所以防禦性檢查out!=nil的結果依然是true。

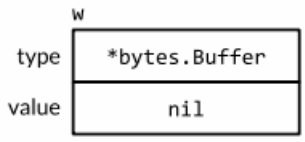


Figure 7.5. A non-nil interface containing a nil pointer.

動態分配機製依然決定(*bytes.Buffer).Write的方法會被調用，但是這次的接收者的值是nil。對於一些如*os.File的類型，nil是一個有效的接收者(§6.2.1)，但是*bytes.Buffer類型不在這些類型中。這個方法會被調用，但是當它嚐試去獲取緩衝區時會發生panic。

問題在於盡管一個nil的*bytes.Buffer指針有實現這個接口的方法，它也不滿足這個接口具體的行爲上的要求。特別是這個調用違反了(*bytes.Buffer).Write方法的接收者非空的隱含先覺條件，所以將nil指針賦給這個接口是錯誤的。解決方案就是將main函數中的變量buf的類型改爲io.Writer，因此可以避免一開始就將一個不完全的值賦值給這個接口：

```
var
 buf io.Writer
if
 debug {
    buf = new
(bytes.Buffer) // enable collection of output

}
f(buf) // OK
```

現在我們已經把接口值的技巧都講完了，讓我們來看更多的一些在Go標準庫中的重要接口類型。在下面的三章中，我們會看到接口類型是怎樣用在排序，web服務，錯誤處理中的。

7.6. sort.Interface接口

TODO

7.7. http.Handler接口

TODO

7.8. error接口

TODO

7.9. 示例: 表達式求值

TODO

7.10. 類型斷言

TODO

7.11. 基於類型斷言識別錯誤類型

TODO

7.12. 通過類型斷言查詢接口

TODO

7.13. 類型分支

TODO

7.14. 示例: 基於標記的XML解碼

TODO

7.15. 補充幾點

TODO

第八章 Goroutines和Channels

併發程序指的是同時做好幾件事情的程序，隨着硬件的發展，併發程序顯得越來越重要。Web服務器會一次處理成韆上萬的請求。平闆電腦和手機app在渲染用戶動畫的同時，還會後台執行各種計算任務和網絡請求。即使是傳統的批處理問題--讀取數據，計算，寫輸出--現在也會用併發來隱藏掉I/O的操作延遲充分利用現代計算機設備的多核，盡管計算機的性能每年都在增長，但併不是線性。

Go語言中的併發程序可以用兩種手段來實現。這一章會講解goroutine和channel，其支持“順序進程通信”(communicating sequential processes)或被簡稱爲CSP。CSP是一個現代的併發編程模型，在這種編程模型中值會在不同的運行實例(goroutine)中傳遞，盡管大多數情況下被限製在單一實例中。第9章會覆蓋到更爲傳統的併發模型：多線程共享內存，如果你在其它的主流語言中寫過併發程序的話可能會更熟悉一些。第9章同時會講一些本章不會深入的併發程序帶來的重要風險和陷阱。

盡管Go對併發的支持是衆多強力特性之一，但大多數情況下跟蹤併發程序還是很睏難，併且在線性程序中我們的直覺往往還會讓我們誤入歧途。如果這是你第一次接觸併發，那麽我推薦你稍微多花一些時間來思考這兩個章節中的樣例。

8.1. Goroutines

在Go語言中，每一個併發的執行單元叫作一個goroutine。設想這里有一個程序有兩個函數，一個函數做一些計算，另一個輸出一些結果，假設兩個函數沒有相互之間的調用關繫。一個線性的程序會先調用其中的一個函數，然後再調用來一個，但如果是在有兩個甚至更多個goroutine的程序中，對兩個函數的調用就可以在同一時間。我們馬上就會看到這樣的一個程序。

如果你使用過操作繫統或者其它語言提供的線程，那麼你可以簡單地把goroutine類比作一個線程，這樣你就可以寫出一些正確的程序了。goroutine和線程的本質區別會在9.8節中講。

當一個程序啟動時，其主函數卽在一個單獨的goroutine中運行，我們叫它main goroutine。新的goroutine會用go語句來創建。在語法上，go語句是一個普通的函數或方法調用前加上關鍵字go。go語句會使其語句中的函數在一個新創建的goroutine中運行。而go語句本身會迅速地完成。

```
f()    // call f(); wait for it to return

go
 f() // create a new goroutine that calls f(); don't wait
```

在下面的例子中，main goroutine會計算第45個菲波那契數。由於計算函數使用了效率非常低的遞歸，所以會運行相當可觀的一段時間，在這期間我們想要讓用戶看到一個可見的標識來表明程序依然在正常運行，所以顯示一個動畫的小圖標：

```
gopl.io/ch8/spinner
func
 main() {
    go
 spinner(100
 * time.Millisecond)
    const
 n = 45

    fibN := fib(n) // slow

    fmt.Printf("\rFibonacci(%d) = %d\n"
, n, fibN)
}

func
 spinner(delay time.Duration) {
    for
 {
        for
 _, r := range
 `-\|/`
 {
            fmt.Printf("\r%c"
, r)
            time.Sleep(delay)
        }
    }
}

func
 fib(x int
) int
 {
    if
 x < 2
 {
        return
 x
    }
    return
 fib(x-1
) + fib(x-2
)
}
```

動畫顯示了幾秒之後，fib(45)的調用成功地返迴，併且打印結果：

```
Fibonacci(45) = 1134903170
```

然後主函數返迴。當主函數返迴時，所有的goroutine都會直接打斷，程序退出。除了從主函數退出或者直接退出程序之外，沒有其它的編程方法能夠讓一個goroutine來打斷另一個的執行，但是我們之後可以看到，可以通過goroutine之間的通信來讓一個goroutine請求請求其它的goroutine，併讓其自己結束執行。

註意這里的兩個獨立的單元是如何進行組合的，spinning和菲波那契的計算。每一個都是寫在獨立的函數中，但是每一個函數都會併發地執行。

8.2. 示例: 併發的Clock服務

網絡編程是併發大顯身手的一個領域，由於服務器是最典型的需要同時處理很多連接的程序，這些連接一般來自遠彼此獨立的客戶端。在本小節中，我們會講解go語言的net包，這個包提供編寫一個網絡客戶端或者服務器程序的基本組件，無論兩者間通信是使用TCP，UDP或者Unix domain sockets。在第一章中我們已經使用過的net/http包里的方法，也算是net包的一部分。

我們的第一個例子是一個順序執行的時鐘服務器，它會每隔一秒鐘將當前時間寫到客戶端：

```
gopl.io/ch8/clock1
// Clock1 is a TCP server that periodically writes the time.

package
 main

import
 (
    "io"

    "log"

    "net"

    "time"

)

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

    for
 {
        conn, err := listener.Accept()
        if
 err != nil
 {
            log.Print(err) // e.g., connection aborted

            continue

        }
        handleConn(conn) // handle one connection at a time

    }
}
```

```
func
 handleConn(c net.Conn) {
    defer
 c.Close()
    for
 {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"
))
        if
 err != nil
 {
            return
 // e.g., client disconnected

        }
        time.Sleep(1
 * time.Second)
    }
}
```

Listen函數創建了一個net.Listener的對象，這個對象會監聽一個網絡端口上到來的連接，在這個例子里我們用的是TCP的localhost:8000端口。listener對象的Accept方法會直接阻塞，直到一個新的連接被創建，然後會返迴一個net.Conn對象來表示這個連接。

handleConn函數會處理一個完整的客戶端連接。在一個for死循環中，將當前的時候用time.Now()函數得到，然後寫到客戶端。由於net.Conn實現了io.Writer接口，我們可以直接向其寫入內容。這個死循環會一直執行，直到寫入失敗。最可能的原因是客戶端主動斷開連接。這種情況下handleConn函數會用defer調用關閉服務器側的連接，然後返迴到主函數，繼續等待下一個連接請求。

time.Time.Format方法提供了一種格式化日期和時間信息的方式。它的參數是一個格式化模闆標識如何來格式化時間，而這個格式化模闆限定爲Mon Jan 2 03:04:05PM 2006 UTC-0700。有8個部分(週幾，月份，一個月的第幾天，等等)。可以以任意的形式來組合前面這個模闆；出現在模闆中的部分會作爲參考來對時間格式進行輸出。在上面的例子中我們隻用到了小時、分鐘和秒。time包里定義了很多標準時間格式，比如time.RFC1123。在進行格式化的逆向操作time.Parse時，也會用到同樣的策略。(譯註：這是go語言和其它語言相比比較奇葩的一個地方。。你需要記住格式化字符串是1月2日下午3點4分5秒零六年UTC-0700，而不像其它語言那樣Y-m-d H:i:s一樣，當然了這里可以用1234567的方式來記憶，倒是也不麻煩)

爲了連接例子里的服務器，我們需要一個客戶端程序，比如netcat這個工具(nc命令)，這個工具可以用來執行網絡連接操作。

```
$ go build gopl.io/ch8/clock1
$ ./clock1 &
$ nc localhost 8000
13:58:54
13:58:55
13:58:56
13:58:57
^C
```

客戶端將服務器發來的時間顯示了出來，我們用Control+C來中斷客戶端的執行，在Unix繫統上，你會看到^C這樣的響應。如果你的繫統沒有裝nc這個工具，你可以用telnet來實現同樣的效果，或者也可以用我們下面的這個用go寫的簡單的telnet程序，用net.Dial就可以簡單地創建一個TCP連接：

```
gopl.io/ch8/netcat1
// Netcat1 is a read-only TCP client.

package
 main

import
 (
    "io"

    "log"

    "net"

    "os"

)

func
 main() {
    conn, err := net.Dial("tcp"
, "localhost:8000"
)
    if
 err != nil
 {
        log.Fatal(err)
    }
    defer
 conn.Close()
    mustCopy(os.Stdout, conn)
}

func
 mustCopy(dst io.Writer, src io.Reader) {
    if
 _, err := io.Copy(dst, src); err != nil
 {
        log.Fatal(err)
    }
}
```

這個程序會從連接中讀取數據，併將讀到的內容寫到標準輸出中，直到遇到end of file的條件或者發生錯誤。mustCopy這個函數我們在本節的幾個例子中都會用到。讓我們同時運行兩個客戶端來進行一個測試，這里可以開兩個終端窗口，下面左邊的是其中的一個的輸出，右邊的是另一個的輸出：

```
$ go build gopl.io/ch8/netcat1
$ ./netcat1
13:58:54                                $ ./netcat1
13:58:55
13:58:56
^C
                                        13:58:57
                                        13:58:58
                                        13:58:59
                                        ^C
$ killall clock1
```

killall命令是一個Unix命令行工具，可以用給定的進程名來殺掉所有名字匹配的進程。

第二個客戶端必鬚等待第一個客戶端完成工作，這樣服務端才能繼續向後執行；因爲我們這里的服務器程序同一時間隻能處理一個客戶端連接。我們這里對服務端程序做一點小改動，使其支持併發：在handleConn函數調用的地方增加go關鍵字，讓每一次handleConn的調用都進入一個獨立的goroutine。

```
gopl.io/ch8/clock2
for
 {
    conn, err := listener.Accept()
    if
 err != nil
 {
        log.Print(err) // e.g., connection aborted

        continue

    }
    go
 handleConn(conn) // handle connections concurrently

}
```

現在多個客戶端可以同時接收到時間了：

```
$ go build gopl.io/ch8/clock2
$ ./clock2 &
$ go build gopl.io/ch8/netcat1
$ ./netcat1
14:02:54                                $ ./netcat1
14:02:55                                14:02:55
14:02:56                                14:02:56
14:02:57                                ^C
14:02:58
14:02:59                                $ ./netcat1
14:03:00                                14:03:00
14:03:01                                14:03:01
^C                                      14:03:02
                                        ^C
$ killall clock2
```

練習8.1: 脩改clock2來支持傳入參數作爲端口號，然後寫一個clockwall的程序，這個程序可以同時與多個clock服務器通信，從多服務器中讀取時間，併且在一個表格中一次顯示所有服務傳迴的結果，類似於你在某些辦公室里看到的時鐘牆。如果你有地理學上分布式的服務器可以用的話，讓這些服務器跑在不同的機器上面；或者在同一台機器上跑多個不同的實例，這些實例監聽不同的端口，假裝自己在不同的時區。像下面這樣：

```
$ TZ=US/Eastern    ./clock2 -port 8010 &
$ TZ=Asia/Tokyo    ./clock2 -port 8020 &
$ TZ=Europe/London ./clock2 -port 8030 &
$ clockwall NewYork=localhost:8010 Tokyo=localhost:8020 London=localhost:8030
```

練習8.2: 實現一個併發FTP服務器。服務器應該解析客戶端來的一些命令，比如cd命令來切換目録，ls來列出目録內文件，get和send來傳輸文件，close來關閉連接。你可以用標準的ftp命令來作爲客戶端，或者也可以自己實現一個。

8.3. 示例: 併發的Echo服務

clock服務器每一個連接都會起一個goroutine。在本節中我們會創建一個echo服務器，這個服務在每個連接中會有多個goroutine。大多數echo服務僅僅會返迴他們讀取到的內容，就像下面這個簡單的handleConn函數所做的一樣：

```
func
 handleConn(c net.Conn) {
    io.Copy(c, c) // NOTE:
 ignoring errors

    c.Close()
}
```

一個更有意思的echo服務應該模擬一個實際的echo的“迴響”，併且一開始要用大寫HELLO來表示“聲音很大”，之後經過一小段延遲返迴一個有所緩和的Hello，然後一個全小寫字母的hello表示聲音漸漸變小直至消失，像下面這個版本的handleConn(譯註：笑看作者腦洞大開)：

```
gopl.io/ch8/reverb1
func
 echo(c net.Conn, shout string
, delay time.Duration) {
    fmt.Fprintln(c, "\t"
, strings.ToUpper(shout))
    time.Sleep(delay)
    fmt.Fprintln(c, "\t"
, shout)
    time.Sleep(delay)
    fmt.Fprintln(c, "\t"
, strings.ToLower(shout))
}

func
 handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for
 input.Scan() {
        echo(c, input.Text(), 1
*time.Second)
    }
    // NOTE:
 ignoring potential errors from input.Err()

    c.Close()
}
```

我們需要陞級我們的客戶端程序，這樣它就可以發送終端的輸入到服務器，併把服務端的返迴輸出到終端上，這使我們有了使用併發的另一個好機會：

```
gopl.io/ch8/netcat2
func
 main() {
    conn, err := net.Dial("tcp"
, "localhost:8000"
)
    if
 err != nil
 {
        log.Fatal(err)
    }
    defer
 conn.Close()
    go
 mustCopy(os.Stdout, conn)
    mustCopy(conn, os.Stdin)
}
```

當main goroutine從標準輸入流中讀取內容併將其發送給服務器時，另一個goroutine會讀取併打印服務端的響應。當main goroutine碰到輸入終止時，例如，用戶在終端中按了Control-D(^D)，在windows上是Control-Z，這時程序就會被終止，盡管其它goroutine中還有進行中的任務。(在8.4.1中引入了channels後我們會明白如何讓程序等待兩邊都結束)。

下面這個會話中，客戶端的輸入是左對齊的，服務端的響應會用縮進來區別顯示。 客戶端會向服務器“喊三次話”：

```
$ go build gopl.io/ch8/reverb1
$ ./reverb1 &
$ go build gopl.io/ch8/netcat2
$ ./netcat2
Hello?
    HELLO?
    Hello?
    hello?
Is there anybody there?
    IS THERE ANYBODY THERE?
Yooo-hooo!
    Is there anybody there?
    is there anybody there?
    YOOO-HOOO!
    Yooo-hooo!
yooo-hooo!
^D
$ killall reverb1
```

註意客戶端的第三次shout在前一個shout處理完成之前一直沒有被處理，這貌似看起來不是特別“現實”。眞實世界里的迴響應該是會由三次shout的迴聲組合而成的。爲了模擬眞實世界的迴響，我們需要更多的goroutine來做這件事情。這樣我們就再一次地需要go這個關鍵詞了，這次我們用它來調用echo：

```
gopl.io/ch8/reverb2
func
 handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for
 input.Scan() {
        go
 echo(c, input.Text(), 1
*time.Second)
    }
    // NOTE:
 ignoring potential errors from input.Err()

    c.Close()
}
```

go後跟的函數的參數會在go語句自身執行時被求值；因此input.Text()會在main goroutine中被求值。 現在迴響是併發併且會按時間來覆蓋掉其它響應了：

```
$ go build gopl.io/ch8/reverb2
$ ./reverb2 &
$ ./netcat2
Is there anybody there?
    IS THERE ANYBODY THERE?
Yooo-hooo!
    Is there anybody there?
    YOOO-HOOO!
    is there anybody there?
    Yooo-hooo!
    yooo-hooo!
^D
$ killall reverb2
```

讓服務使用併發不隻是處理多個客戶端的請求，甚至在處理單個連接時也可能會用到，就像我們上面的兩個go關鍵詞的用法。然而在我們使用go關鍵詞的同時，需要慎重地考慮net.Conn中的方法在併發地調用時是否安全，事實上對於大多數類型來說也確實不安全。我們會在下一章中詳細地探討併發安全性。

8.4. Channels

如果説goroutine是Go語音程序的併發體的話，那麼channels它們之間的通信機製。一個channels是一個通信機製，它可以讓一個goroutine通過它給另一個goroutine發送值信息。每個channel都有一個特殊的類型，也就是channels可發送數據的類型。一個可以發送int類型數據的channel一般寫爲chan int。

使用內置的make函數，我們可以創建一個channel：

```
ch := make
(chan
 int
) // ch has type 'chan int'
```

和map類似，channel也一個對應make創建的底層數據結構的引用。當我嗎複製一個channel或用於函數參數傳遞時，我嗎隻是拷貝了一個channel引用，因此調用者何被調用者將引用同一個channel對象。和其它的引用類型一樣，channel的零值也是nil。

兩個相同類型的channel可以使用==運算符比較。如果兩個channel引用的是相通的對象，那麼比較的結果爲眞。一個channel也可以和nil進行比較。

一個channel有發送和接受兩個主要操作，都是通信行爲。一個發送語句將一個值從一個goroutine通過channel發送到另一個執行接收操作的goroutine。發送和接收兩個操作都是用 `<-` 運算符。在發送語句中， `<-` 運算符分割channel和要發送的值。在接收語句中， `<-` 運算符寫在channel對象之前。一個不使用接收結果的接收操作也是合法的。

```
ch <- x  // a send statement

x = <-ch // a receive expression in an assignment statement

<-ch     // a receive statement; result is discarded
```

Channel還支持close操作，用於關閉channel，隨後對基於該channel的任何發送操作都將導致panic異常。對一個已經被close過的channel之行接收操作依然可以接受到之前已經成功發送的數據；如果channel中已經沒有數據的話講産生一個零值的數據。

使用內置的close函數就可以關閉一個channel：

```
close
(ch)
```

以最簡單方式調用make函數創建的時一個無緩存的channel，但是我們也可以指定第二個整形參數，對應channel的容量。如果channel的容量大於零，那麼該channel就是帶緩存的channel。

```
ch = make
(chan
 int
)    // unbuffered channel

ch = make
(chan
 int
, 0
) // unbuffered channel

ch = make
(chan
 int
, 3
) // buffered channel with capacity 3
```

我們將先討論無緩存的channel，然後在8.4.4節討論帶緩存的channel。

8.4.1. 不帶緩存的Channels

一個基於無緩存Channels的發送操作將導致發送者goroutine阻塞，直到另一個goroutine在相同的Channels上執行接收操作，當發送的值通過Channels成功傳輸之後，兩個goroutine可以繼續執行後面的語句。反之，如果接收操作先發生，那麼接收者goroutine也將阻塞，直到有另一個goroutine在相同的Channels上執行發送操作。

基於無緩存Channels的發送和接收操作將導致兩個goroutine做一次同步操作。因爲這個原因，無緩存Channels有時候也被稱爲同步Channels。當通過一個無緩存Channels發送數據時，接收者收到數據發生在喚醒發送者goroutine之前（譯註：*happens before*，這是Go語言併發內存模型的一個關鍵術語！）。

在討論併發編程時，當我們說x事件在y事件之前發生（*happens before*），我們併不是說x事件在時間上比y時間更早；我們要表達的意思是要保證在此之前的事件都已經完成了，例如在此之前的更新某些變量的操作已經完成，你可以放心依賴這些已完成的事件了。

當我們說x事件既不是在y事件之前發生也不是在y事件之後發生，我們就說x事件和y事件是併發的。這併不是意味着x事件和y事件就一定是同時發生的，我們隻是不能確定這兩個事件發生的先後順序。在下一章中我們將看到，當兩個goroutine併發訪問了相同的變量時，我們有必要保證某些事件的執行順序，以避免出現某些併發問題。

在8.3節的客戶端程序，它在主goroutine中（譯註：就是執行main函數的goroutine）將標準輸入複製到server，因此當客戶端程序關閉標準輸入時，後台goroutine可能依然在工作。我們需要讓主goroutine等待後台goroutine完成工作後再退出，我們使用了一個channel來同步兩個goroutine：

```
gopl.io/ch8/netcat3
func
 main() {
    conn, err := net.Dial("tcp"
, "localhost:8000"
)
    if
 err != nil
 {
        log.Fatal(err)
    }
    done := make
(chan
 struct
{})
    go
 func
() {
        io.Copy(os.Stdout, conn) // NOTE:
 ignoring errors

        log.Println("done"
)
        done <- struct
{}{} // signal the main goroutine

    }()
    mustCopy(conn, os.Stdin)
    conn.Close()
    <-done // wait for background goroutine to finish

}
```

當用戶關閉了標準輸入，主goroutine中的mustCopy函數調用將返迴，然後調用conn.Close()關閉讀和寫方向的網絡連接。關閉網絡鏈接中的寫方向的鏈接將導致server程序收到一個文件（end-of-file）結束的信號。關閉網絡鏈接中讀方向的鏈接將導致後台goroutine的io.Copy函數調用返迴一個“read from closed connection”（“從關閉的鏈接讀”）類似的錯誤，因此我們臨時移除了錯誤日誌語句；在練習8.3將會提供一個更好的解決方案。（需要註意的是go語句調用了一個函數字面量，這Go語言中啟動goroutine常用的形式。）

在後台goroutine返迴之前，它先打印一個日誌信息，然後向done對應的channel發送一個值。主goroutine在退出前先等待從done對應的channel接收一個值。因此，總是可以在程序退出前正確輸出“done”消息。

基於channels發送消息有兩個重要方面。首先每個消息都有一個值，但是有時候通訊的事實和發生的時刻也同樣重要。當我們更希望強調通訊發生的時刻時，我們將它稱爲**消息事件**。有些消息事件併不攜帶額外的信息，它僅僅是用作兩個goroutine之間的同步，這時候我們可以用 `struct{}` 空結構體作爲channels元素的類型，雖然也可以使用bool或int類型實現同樣的功能， `done <- 1` 語句也比 `done <- struct{}{}` 更短。

練習 8.3： 在netcat3例子中，conn雖然是一個interface類型的值，但是其底層眞實類型是 `*net.TCPConn` ，代表一個TCP鏈接。一個TCP鏈接有讀和寫兩個部分，可以使用CloseRead和CloseWrite方法分別關閉它們。脩改netcat3的主goroutine代碼，隻關閉網絡鏈接中寫的部分，這樣的話後台goroutine可以在標準輸入被關閉後繼續打印從reverb1服務器傳迴的數據。（要在reverb2服務器也完成同樣的功能是比較睏難的；參考**練習 8.4**。）

8.4.2. 串聯的Channels（Pipeline）

Channels也可以用於將多個goroutine鏈接在一起，一個Channels的輸出作爲下一個Channels的輸入。這種串聯的Channels就是所謂的管道（pipeline）。下面的程序用兩個channels將三個goroutine串聯起來，如圖8.1所示。

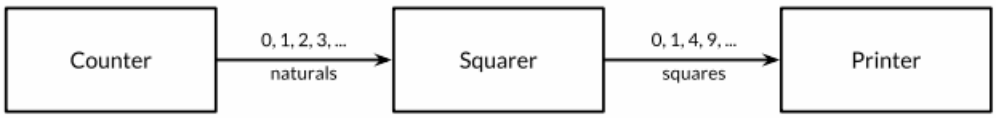


Figure 8.1. A three-stage pipeline.

第一個goroutine是一個計數器，用於生成0、1、2、……形式的整數序列，然後通過channel將該整數序列發送給第二個goroutine；第二個goroutine是一個求平方的程序，對收到的每個整數求平方，然後將平方後的結果通過第二個channel發送給第三個goroutine；第三個goroutine是一個打印程序，打印收到的每個整數。爲了保持例子清晰，我們有意選擇了非常簡單的函數，當然三個goroutine的計算很簡單，在現實中確實沒有必要爲如此簡單的運算構建三個goroutine。

```
gopl.io/ch8/pipeline1

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

    // Counter

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

    // Squarer

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

    // Printer (in main goroutine)

    for
 {
        fmt.Println(<-squares)
    }
}
```

如您所料，上面的程序將生成0、1、4、9、……形式的無窮數列。像這樣的串聯Channels的管道（Pipelines）可以用在需要長時間運行的服務中，每個長時間運行的goroutine可能會包含一個死循環，在不同goroutine的死循環內部使用串聯的Channels來通信。但是，如果我們希望通過Channels隻發送有限的數列該如何處理呢？

如果發送者知道，沒有更多的值需要發送到channel的話，那麽讓接收者也能及時知道沒有多餘的值可接收將是有用的，因爲接收者可以停止不必要的接收等待。這可以通過內置的close函數來關閉channel實現：

```
close
(naturals)
```

當一個channel被關閉後，再向該channel發送數據將導致panic異常。當一個被關閉的channel中已經發送的數據都被成功接收後，後續的接收操作將不再阻塞，它們會立即返回一個零值。關閉上面例子中的naturals變量對應的channel併不能終止循環，它依然會收到一個永無休止的零值序列，然後將它們發送給打印者goroutine。

沒有辦法直接測試一個channel是否被關閉，但是接收操作有一個變體形式：它多接收一個結果，多接收的第二個結果是一個布爾值ok，ture表示成功從channels接收到值，false表示channels已經被關閉併且里面沒有值可接收。使用這個特性，我們可以脩改squarer函數中的循環代碼，當naturals對應的channel被關閉併沒有值可接收時跳出循環，併且也關閉squares對應的channel.

```
// Squarer

go
 func
() {
    for
 {
        x, ok := <-naturals
        if
 !ok {
            break
 // channel was closed and drained

        }
        squares <- x * x
    }
    close
(squares)
}()
```

因爲上面的語法是笨拙的，而且這種處理模式很場景，因此Go語言的range循環可直接在channels上面迭代。使用range循環是上面處理模式的簡潔語法，它依次從channel接收數據，當channel被關閉併且沒有值可接收時跳出循環。

在下面的改進中，我們的計數器goroutine隻生成100個含數字的序列，然後關閉naturals對應的channel，這將導致計算平方數的squarer對應的goroutine可以正常終止循環併關閉squares對應的channel。（在一個更複雜的程序中，可以通過defer語句關閉對應的channel。）最後，主goroutine也可以正常終止循環併退出程序。

```
gopl.io/ch8/pipeline2

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

    // Counter

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

    // Squarer

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

    // Printer (in main goroutine)

    for
 x := range
 squares {
        fmt.Println(x)
    }
}
```

其實你併不需要關閉每一個channel。隻要當需要告訴接收者goroutine，所有的數據已經全部發送時才需要關閉channel。不管一個channel是否被關閉，當它沒有被引用時將會被Go語言的垃圾自動迴收器迴收。（不要將關閉一個打開文件的操作和關閉一個channel操作混淆。對於每個打開的文件，都需要在不使用的使用調用對應的Close方法來關閉文件。）

視圖重複關閉一個channel將導致panic異常，視圖關閉一個nil值的channel也將導致panic異常。關閉一個channels還會觸發一個廣播機製，我們將在8.9節討論。

8.4.3. 單方向的Channel

隨着程序的增長，人們習慣於將大的函數拆分爲小的函數。我們前面的例子中使用了三個goroutine，然後用兩個channels連鏈接它們，它們都是main函數的局部變量。將三個goroutine拆分爲以下三個函數是自然的想法：

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

其中squarer計算平方的函數在兩個串聯Channels的中間，因此擁有兩個channels類型的參數，一個用於輸入一個用於輸出。每個channels都用有相同的類型，但是它們的使用方式想反：一個隻用於接收，另一個隻用於發送。參數的名字in和out已經明確表示了這個意圖，但是併無法保證squarer函數向一個in參數對應的channels發送數據或者從一個out參數對應的channels接收數據。

這種場景是典型的。當一個channel作爲一個函數參數是，它一般總是被專門用於隻發送或者隻接收。

爲了表明這種意圖併防止被濫用，Go語言的類型繫統提供了單方向的channel類型，分別用於隻發送或隻接收的channel。類型 `chan<- int` 表示一個隻發送int的channel，隻能發送不能接收。相反，類型 `<-chan int` 表示一個隻接收int的channel，隻能接收不能發送。（箭頭 `<-` 和關鍵字chan的相對位置表明了channel的方向。）這種限製將在編譯期檢測。

因爲關閉操作隻用於斷言不再向channel發送新的數據，所以隻有在發送者所在的goroutine才會調用close函數，因此對一個隻接收的channel調用close將是一個編譯錯誤。

這是改進的版本，這一次參數使用了單方向channel類型：

```
gopl.io/ch8/pipeline3
func
 counter(out chan
<- int
) {
    for
 x := 0
; x < 100
; x++ {
        out <- x
    }
    close
(out)
```

```
}

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

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

func
 main() {
    naturals := make
(chan
 int
)
    squares := make
(chan
 int
)
    go
 counter(naturals)
    go
 squarer(squares, naturals)
    printer(squares)
}
```

調用counter(naturals)將導致將 `chan int` 類型的naturals隱式地轉換爲 `chan<- int` 類型隻發送型的channel。調用printer(squares)也會導致相似的隱式轉換，這一次是轉換爲 `<-chan int` 類型隻接收型的channel。任何雙向channel向單向channel變量的賦值操作都將導致該隱式轉換。這里併沒有反向轉換的語法：也就是不能一個將類似 `chan<- int` 類型的單向型的channel轉換爲 `chan int` 類型的雙向型的channel。

8.4.4. 帶緩存的Channels

帶緩存的Channel內部持有一個元素隊列。隊列的最大容量是在調用make函數創建channel時通過第二個參數指定的。下面的語句創建了一個可以持有三個字符串元素的帶緩存Channel。圖8.2是ch變量對應的channel的圖形表示形式。

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

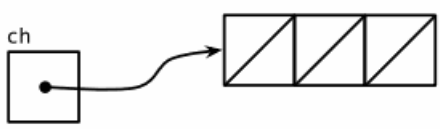


Figure 8.2. An empty buffered channel.

向緩存Channel的發送操作就是向內部緩存隊列的尾部插入原因，接收操作則是從隊列的頭部刪除元素。如果內部緩存隊列是滿的，那麼發送操作將阻塞直到因另一個goroutine執行接收操作而釋放了新的隊列空間。相反，如果channel是空的，接收操作將阻塞直到有另一個goroutine執行發送操作而向隊列插入元素。

我們可以在無阻塞的情況下連續向新創建的channel發送三個值：

```
ch <- "A"

ch <- "B"

ch <- "C"
```

此刻，channel的內部緩存隊列將是滿的（圖8.3），如果有第四個發送操作將發生阻塞。

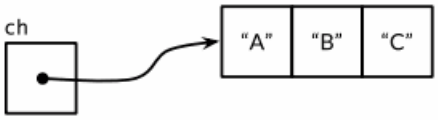


Figure 8.3. A full buffered channel.

如果我們接收一個值，

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

那麼channel的緩存隊列將不是滿的也不是空的（圖8.4），因此對該channel執行的發送或接收操作都不會發送阻塞。通過這種方式，channel的緩存隊列解耦了接收和發送的goroutine。

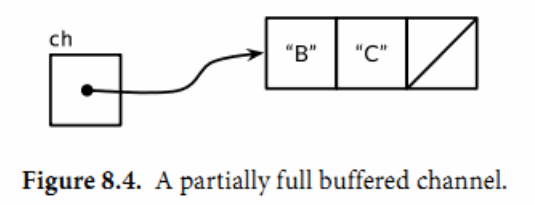


Figure 8.4. A partially full buffered channel.

在某些特殊情況下，程序可能需要知道channel內部緩存的容量，可以用內置的cap函數獲取：

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

同樣，對於內置的len函數，如果傳入的是channel，那麼將返迴channel內部緩存隊列中有效元素的個數。因爲在併發程序中該信息會隨着接收操作而失效，但是它對某些故障診斷和性能優化會有幫助。

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

在繼續執行兩次接收操作後channel內部的緩存隊列將又成爲空的，如果有第四個接收操作將發生阻塞：

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

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

在這個例子中，發送和接收操作都發生在同一個goroutine中，但是在眞是的程序中它們一般由不同的goroutine執行。Go語言新手有時候會將一個帶緩存的channel當作同一個goroutine中的隊列使用，雖然語法看似簡單，但實際上這是一個錯誤。Channel和goroutine的調度器機製是緊密相連的，一個發送操作——或許是整個程序——可能會永遠阻塞。如果你隻是需要一個簡單的隊列，使用slice就可以了。

下面的例子展示了一個使用了帶緩存channel的應用。它併發地向三個鏡像站點發出請求，三個鏡像站點分散在不同的地理位置。它們分別將收到的響應發送到帶緩存channel，最後接收者隻接收第一個收到的響應，也就是最快的那個響應。因此mirroredQuery函數可能在另外兩個響應慢的鏡像站點響應之前就返迴了結果。（順便說一下，多個goroutines併發地向同一個channel發送數據，或從同一個channel接收數據都是常見的用法。）

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

}

func
 request(hostname string
) (response string
) { /* ... */
 }
```

如果我們使用了無緩存的channel，那麼兩個慢的goroutines將會因爲沒有人接收而被永遠卡住。這種情況，稱爲goroutines洩漏，這將是一個BUG。和垃圾變量不同，洩漏的goroutines併不會被自動迴收，因此確保每個不再需要的goroutine能正常退出是重要的。

關於無緩存或帶緩存channels之間的選擇，或者是帶緩存channels的容量大小的選擇，都可能影響程序的正確性。無緩存channel更強地保證了每個發送操作與相應的同步接收操作；但是對於帶緩存channel，這些操作是解耦的。同樣，即使我們知道將要發送到一個channel的信息的數量上限，創建一個對應容量大小帶緩存channel也是不現實的，因爲這要求在執行任何接收操作之前緩存所有已經發送的值。如果未能分配足夠的緩衝將導致程序死鎖。

Channel的緩存也可能影響程序的性能。想象一家蛋糕店有三個廚師，一個烘焙，一個上醣衣，還有一個將每個蛋糕傳遞到它下一個廚師在生產線。在狹小的廚房空間環境，每個廚師在完成蛋糕後必鬚等待下一個廚師已經準備好接受它；這類似於在一個無緩存的channel上進行溝通。

如果在每個廚師之間有一個放置一個蛋糕的額外空間，那麼每個廚師就可以將一個完成的蛋糕臨時放在那里而馬上進入下一個蛋糕在製作中；這類似於將channel的緩存隊列的容量設置爲1。隻要每個廚師的平均工作效率相近，那麼其中大部分的傳輸工作將是迅速的，個體之間細小的效率差異將在交接過程中彌補。如果廚師之間有更大的額外空間——也是就更大容量的緩存隊列——將可以在不停止生產線的前提下消除更大的效率波動，例如一個廚師可以短暫地休息，然後在加快趕上進度而不影響其其他人。

另一方面，如果生產線的前期階段一直快於後續階段，那麼它們之間的緩存在大部分時間都將是滿的。相反，如果後續階段比前期階段更快，那麼它們之間的緩存在大部分時間都將是空的。對於這類場景，額外的緩存併沒有帶來任何好處。

生產線的隱喻對於理解channels和goroutines的工作機製是很有幫助的。例如，如果第二階段是需要精心製作的複雜操作，一個廚師可能無法跟上第一個廚師的進度，或者是無法滿足第階段廚師的需求。要解決這個問題，我們可以雇傭另一個廚師來幫助完成第二階段的工作，他執行相同的任務但是獨立工作。這類似於基於相同的channels創建另一個獨立的goroutine。

我們沒有太多的空間展示全部細節，但是gopl.io/ch8/cake包模擬了這個蛋糕店，可以通過不同的參數調整。它還對上面提到的幾種場景提供對應的基準測試（§11.4）。

8.5. 併發的循環

本節中，我們會探索一些用來在併行時循環迭代的常見併發模型。我們會探究從全尺寸圖片生成一些縮略圖的問題。gopl.io/ch8/thumbnail包提供了ImageFile函數來幫我們拉伸圖片。我們不會說明這個函數的實現，隻需要從gopl.io下載它。

```
gopl.io/ch8/thumbnail
package
 thumbnail
// ImageFile reads an image from infile and writes

// a thumbnail-size version of it in the same directory.

// It returns the generated file name, e.g., "foo.thumb.jpg".

func
 ImageFile(infile string
) (string
, error)
```

下面的程序會循環迭代一些圖片文件名，併爲每一張圖片生成一個縮略圖：

```
gopl.io/ch8/thumbnail
// makeThumbnails makes thumbnails of the specified files.

func
 makeThumbnails(filenames []string
) {
    for
 _, f := range
 filenames {
        if
 _, err := thumbnail.ImageFile(f); err != nil
 {
            log.Println(err)
        }
    }
}
```

顯然我們處理文件的順序無關緊要，因爲每一個圖片的拉伸操作和其它圖片的處理操作都是彼此獨立的。像這種子問題都是完全彼此獨立的問題被叫做易併行問題(譯註：embarrassingly parallel，直譯的話更像是尷尬併行)。易併行問題是最容易被實現成併行的一類問題(廢話)，併且是最能夠享受併發帶來的好處，能夠隨着併行的規模線性地擴展。

下面讓我們併行地執行這些操作，從而將文件IO的延遲隱藏掉，併用上多核cpu的計算能力來拉伸圖像。我們的第一個併發程序隻是使用了一個go關鍵字。這里我們先忽略掉錯誤，之後再進行處理。

```
// NOTE:
 incorrect!

func
 makeThumbnails2(filenames []string
) {
    for
 _, f := range
 filenames {
        go
 thumbnail.ImageFile(f) // NOTE:
 ignoring errors

    }
}
```

這個版本運行的實在有點太快，實際上，由於它比最早的版本使用的時間要短得多，即使當文件名的slice中隻包含有一個元素。這就有點奇怪了，如果程序沒有併發執行的話，那爲什麽一個併發的版本還是要快呢？答案其實是makeThumbnails在它還沒有完成工作之前就已經返迴了。它啟動了所有的goroutine，沒一個文件名對應一個，但沒有等待它們一直到執行完畢。

沒有什麽直接的辦法能夠等待goroutine完成，但是我們可以改變goroutine里的代碼讓其能夠將完成情況報告給外部的goroutine知曉，使用的方式是向一個共享的channel中發送事件。因爲我們已經知道內部的goroutine隻有len(filenames)，所以外部的goroutine隻需要在返迴之前對這些事件計數。

```
// makeThumbnails3 makes thumbnails of the specified files in parallel.

func
 makeThumbnails3(filenames []string
) {
    ch := make
(chan
 struct
{})
    for
 _, f := range
 filenames {
        go
 func
(f string
) {
            thumbnail.ImageFile(f) // NOTE:
 ignoring errors

            ch <- struct
{}{}
        }(f)
    }
    // Wait for goroutines to complete.

    for
 range
 filenames {
        <-ch
    }
}
```

註意我們將f的值作爲一個顯式的變量傳給了函數，而不是在循環的閉包中聲明：

```
for
 _, f := range
 filenames {
    go
 func
() {
        thumbnail.ImageFile(f) // NOTE:
 incorrect!

        // ...

    }()
}
```

迴憶一下之前在5.6.1節中，匿名函數中的循環變量快照問題。上面這個單獨的變量f是被所有的匿名函數值所共享，且會被連續的循環迭代所更新的。當新的goroutine開始執行字面函數時，for循環可能已經更新了f併且開始了另一輪的迭代或者(更有可能的)已經結束了整個循環，所以當這些goroutine開始讀取f的值時，它們所看到的值已經是slice的最後一個元素了。顯式地添加這個參數，我們能夠確保使用的f是當go語句執行時的“當前”那個f。

如果我們想要從每一個worker goroutine往主goroutine中返迴值時該怎麼辦呢？當我們調用thumbnail.ImageFile創建文件失敗的時候，它會返迴一個錯誤。下一個版本的makeThumbnails會返迴其在做拉伸操作時接收到的第一個錯誤：

```
// makeThumbnails4 makes thumbnails for the specified files in parallel.

// It returns an error if any step failed.

func
 makeThumbnails4(filenames []string
) error {
    errors := make
(chan
 error)

    for
 _, f := range
 filenames {
        go
 func
(f string
) {
            _, err := thumbnail.ImageFile(f)
            errors <- err
        }(f)}
    }

    for
 range
 filenames {
        if
 err := <-errors; err != nil
 {
            return
 err // NOTE:
 incorrect: goroutine leak!

        }
    }

    return
 nil

}
```

這個程序有一個微秒的bug。當它遇到第一個非nil的error時會直接將error返迴到調用方，使得沒有一個goroutine去排空errors channel。這樣剩下的worker goroutine在向這個channel中發送值時，都會永遠地阻塞下去，併且永遠都不會退出。這種情況叫做goroutine洩露(§8.4.4)，可能會導致整個程序卡住或者跑出out of memory的錯誤。

最簡單的解決辦法就是用一個具有合適大小的buffered channel，這樣這些worker goroutine向channel中發送測向時就不會被阻塞。(一個可選的解決辦法是創建一個另外的goroutine，當main goroutine返迴第一個錯誤的同時去排空channel)

下一個版本的makeThumbnails使用了一個buffered channel來返迴生成的圖片文件的名字，附帶生成時的錯誤。

```
// makeThumbnails5 makes thumbnails for the specified files in parallel.

// It returns the generated file names in an arbitrary order,

// or an error if any step failed.


func
 makeThumbnails5(filenames []string
) (thumbfiles []string
, err error) {
    type
 item struct
 {
        thumbfile string

        err       error
    }

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

    for
 range
 filenames {
        it := <-ch
        if
 it.err != nil
```

```
 {
            return
 nil
, it.err
        }
        thumbfiles = append
(thumbfiles, it.thumbfile)
    }

    return
 thumbfiles, nil

}
```

我們最後一個版本的makeThumbnails返迴了新文件們的大小總計數(bytes)。和前面的版本都不一樣的一點是我們在這個版本里沒有把文件名放在slice里，而是通過一個string的channel傳過來，所以我們無法對循環的次數進行預測。

爲了知道最後一個goroutine什麼時候結束(最後一個結束併不一定是最後一個開始)，我們需要一個遞增的計數器，在每一個goroutine啟動時加一，在goroutine退出時減一。這需要一種特殊的計數器，這個計數器需要在多個goroutine操作時做到安全併且提供提供在其減爲零之前一直等待的一種方法。這種計數類型被稱爲sync.WaitGroup，下面的代碼就用到了這種方法：

```
// makeThumbnails6 makes thumbnails for each file received from the channel.

// It returns the number of bytes occupied by the files it creates.

func
 makeThumbnails6(filenames <-chan
 string
) int64
 {
    sizes := make
(chan
 int64
)
    var
 wg sync.WaitGroup // number of working goroutines

    for
 f := range
 filenames {
        wg.Add(1
)
        // worker

        go
 func
(f string
) {
            defer
 wg.Done()
            thumb, err := thumbnail.ImageFile(f)
```

```
            if
 err != nil
 {
                log.Println(err)
                return

            }
            info, _ := os.Stat(thumb) // OK to ignore error

            sizes <- info.Size()
        }(f)
    }

    // closer

    go
 func
() {
        wg.Wait()
        close
(sizes)
    }()


    var
 total int64

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

註意Add和Done方法的不對策。Add是爲計數器加一，必鬚在worker goroutine開始之前調用，而不是在goroutine中；否則的話我們沒辦法確定Add是在"closer" goroutine調用Wait之前被調用。併且Add還有一個參數，但Done卻沒有任何參數；其實它和Add(-1)是等價的。我們使用defer來確保計數器即使是在出錯的情況下依然能夠正確地被減掉。上面的程序代碼結構是當我們使用併發循環，但又不知道迭代次數時很通常而且很地道的寫法。

sizes channel攜帶了每一個文件的大小到main goroutine，在main goroutine中使用了range loop來計算總和。觀察一下我們是怎樣創建一個closer goroutine，併讓其等待worker們在關閉掉sizes channel之前退出的。兩步操作：wait和close，必鬚是基於sizes的循環的併發。考慮一下另一種方案：如果等待操作被放在了main goroutine中，在循環之前，這樣的話就永遠都不會結束了，如果在循環之後，那麼又變成了不可達的部分，因爲沒有任何東西去關閉這個channel，這個循環就永遠都不會終止。

圖8.5 表明了makethumbnails6函數中事件的序列。縱列表示goroutine。窄線段代表sleep，粗線段代表活動。斜線箭頭代表用來同步兩個goroutine的事件。時間向下流動。註意main goroutine是如何大部分的時間被喚醒執行其range循環，等待worker發送值或者closer來關閉channel的。

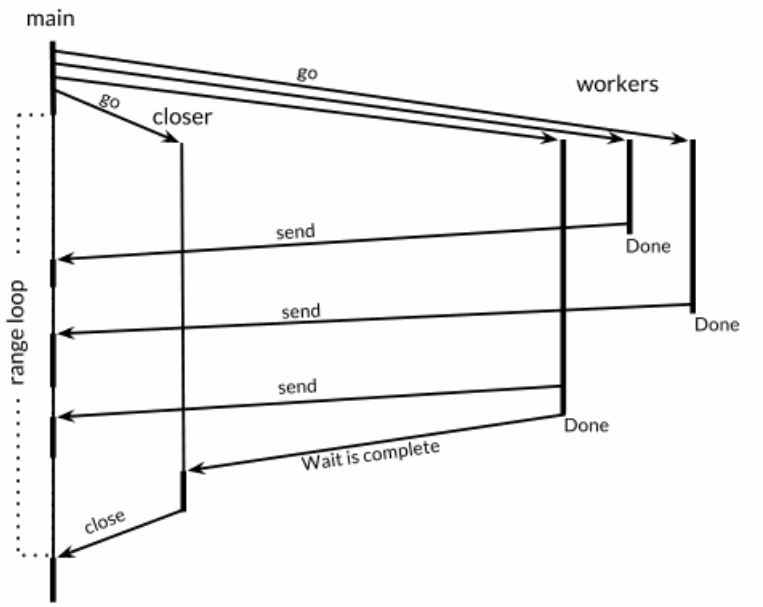


Figure 8.5. The sequence of events in `makeThumbnails6`.

練習8.4: 脩改reverb2服務器，在每一個連接中使用sync.WaitGroup來計數活躍的echo goroutine。當計數減爲零時，關閉TCP連接的寫入，像練習8.3中一樣。驗證一下你的脩改版netcat3客戶端會一直等待所有的併發“喊叫”完成，即使是在標準輸入流已經關閉的情況下。

練習8.5: 使用一個已有的CPU綁定的順序程序，比如在3.3節中我們寫的Mandelbrot程序或者3.2節中的3-D surface計算程序，併將他們的主循環改爲併發形式，使用channel來進行通信。在多核計算機上這個程序得到了多少速度上的改進？使用多少個goroutine是最合適的呢？

8.6. 示例: 併發的Web爬蟲

在5.6節中，我們做了一個簡單的web爬蟲，用bfs(廣度優先)算法來抓取整個網站。在本節中，我們會讓這個這個爬蟲併行化，這樣每一個彼此獨立的抓取命令可以併行進行IO，最大化利用網絡資源。crawl函數和gopl.io/ch5/findlinks3中的是一樣的。

```
gopl.io/ch8/crawl1
func
 crawl(url string
) []string
 {
    fmt.Println(url)
    list, err := links.Extract(url)
    if
 err != nil
 {
        log.Print(err)
    }
    return
 list
}
```

主函數和5.6節中的breadthFirst(深度優先)類似。像之前一樣，一個worklist是一個記録了需要處理的元素的隊列，每一個元素都是一個需要抓取的URL列表，不過這一次我們用channel代替slice來做這個隊列。每一個對crawl的調用都會在他們自己的goroutine中進行併且會把他們抓到的鏈接發送迴worklist。

```
func
 main() {
    worklist := make
(chan
 []string
)

    // Start with the command-line arguments.

    go
 func
() { worklist <- os.Args[1
:] }()

    // Crawl the web concurrently.

    seen := make
(map
[string
]bool
)
    for
 list := range
 worklist {
        for
 _, link := range
 list {
            if
 !seen[link] {
                seen[link] = true

                go
 func
(link string
) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

註意這里的crawl所在的goroutine會將link作爲一個顯式的參數傳入，來避免“循環變量快照”的問題(在5.6.1中有講解)。另外註意這里將命令行參數傳入worklist也是在一個另外的goroutine中進行的，這是爲了避免在main goroutine和crawler goroutine中同時向另一個goroutine通過channel發送內容時發生死鎖(因爲另一邊的接收操作還沒有準備好)。當然，這里我們也可以用buffered channel來解決問題，這里不再贅述。

現在爬蟲可以高併發地運行起來，併且可以產生一大坨的URL了，不過還是會有倆問題。一個問題是在運行一段時間後可能會出現在log的錯誤信息里的：

```
$ go build gopl.io/ch8/crawl1
$ ./crawl1 http://gopl.io/
http://gopl.io/
https://golang.org/help/
https://golang.org/doc/
https://golang.org/blog/
...
2015/07/15 18:22:12 Get ...: dial tcp: lookup blog.golang.org: no such host
2015/07/15 18:22:12 Get ...: dial tcp 23.21.222.120:443: socket:
                                                    too many open files
...
```

最初的錯誤信息是一個讓人莫名的DNS查找失敗，即使這個域名是完全可靠的。而隨後的錯誤信息揭示了原因：這個程序一次性創建了太多網絡連接，超過了每一個進程的打開文件數限製，既而導致了在調用net.Dial像DNS查找失敗這樣的問題。

這個程序實在是太他媽併行了。無窮無盡地併行化併不是什麽好事情，因爲不管怎麽説，你的繫統總是會有一個些限製因素，比如CPU核心數會限製你的計算負載，比如你的硬盤轉軸和磁頭數限製了你的本地磁盤IO操作頻率，比如你的網絡帶寬限製了你的下載速度上限，或者是你的一個web服務的服務容量上限等等。爲了解決這個問題，我們可以限製併發程序所使用的資源來使之適應自己的運行環境。對於我們的例子來説，最簡單的方法就是限製對links.Extract在同一時間最多不會有超過n次調用，這里的n是fd的limit-20，一般情況下。這個一個夜店里限製客人數目是一個道理，隻有當有客人離開時，才會允許新的客人進入店內(譯註：作者你個老流氓)。

我們可以用一個有容量限製的buffered channel來控製併發，這類似於操作繫統里的計數信號量概念。從概念上講，channel里的n個空槽代表n個可以處理內容的token(通行證)，從channel里接收一個值會釋放其中的一個token，併且生成一個新的空槽位。這樣保證了在沒有接收介入時最多有n個發送操作。(這里可能我們拿channel里填充的槽來做token更直觀一些，不過還是這樣吧~)。由於channel里的元素類型併不重要，我們用一個零值的struct{}來作爲其元素。

讓我們重寫crawl函數，將對links.Extract的調用操作用獲取、釋放token的操作包裹起來，來確保同一時間對其隻有20個調用。信號量數量和其能操作的IO資源數量應保持接近。

```
gopl.io/ch8/crawl2
// tokens is a counting semaphore used to

// enforce a limit of 20 concurrent requests.

var
 tokens = make
(chan
 struct
{}, 20
)

func
 crawl(url string
) []string
 {
    fmt.Println(url)
    tokens <- struct
{}{} // acquire a token

    list, err := links.Extract(url)
    <-tokens // release the token

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

第二個問題是這個程序永遠都不會終止，即使它已經爬到了所有初始鏈接衍生出的鏈接。(當然，除非你慎重地選擇了合適的初始化URL或者已經實現了練習8.6中的深度限製，你應該還沒有意識到這個問題)。爲了使這個程序能夠終止，我們需要在worklist爲空或者沒有crawl的goroutine在運行時退出主循環。

```
func
 main() {
    worklist := make
(chan
 []string
)
    var
 n int
 // number of pending sends to worklist

    // Start with the command-line arguments.

    n++
```

```
    n++
    go
 func
() { worklist <- os.Args[1
:] }()


    // Crawl the web concurrently.

    seen := make
(map
[string
]bool
)

    for
 ; n > 0
; n-- {
        list := <-worklist
        for
 _, link := range
 list {
            if
 !seen[link] {
                seen[link] = true

                n++
                go
 func
(link string
) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

這個版本中，計算器n對worklist的發送操作數量進行了限製。每一次我們發現有元素需要被發送到worklist時，我們都會對n進行++操作，在向worklist中發送初始的命令行參數之前，我們也進行過一次++操作。這里的操作++是在每啟動一個crawler的goroutine之前。主循環會在n減爲0時終止，這時候説明沒活可榦了。

現在這個併發爬蟲會比5.6節中的深度優先蒐索版快上20倍，而且不會出什麽錯，併且在其完成任務時也會正確地終止。

下面的程序是避免過度併發的另一種思路。這個版本使用了原來的crawl函數，但沒有使用計數信號量，取而代之用了20個長活的crawler goroutine，這樣來保證最多20個HTTP請求在併發。

```
func
 main() {
    worklist := make
(chan
 []string
```

```
[]string
)  // lists of URLs, may have duplicates

    unseenLinks := make
(chan
 string
) // de-duplicated URLs



    // Add command-line arguments to worklist.

    go
 func
() { worklist <- os.Args[1
:] }()

    // Create 20 crawler goroutines to fetch each unseen link.

    for
 i := 0
; i < 20
; i++ {
        go
 func
() {
            for
 link := range
 unseenLinks {
                foundLinks := crawl(link)
                go
 func
() { worklist <- foundLinks }()
            }
        }()
    }

    // The main goroutine de-duplicates worklist items

    // and sends the unseen ones to the crawlers.

    seen := make
(map
[string
]bool
)
    for
 list := range
 worklist {
        for
 _, link := range
 list {
            if
 !seen[link] {
```

```
                seen[link] = true

                unseenLinks <- link
            }
        }
    }
}
```

所有的爬蟲goroutine現在都是被同一個channel-unseenLinks餵飽的了。主goroutine負責拆分它從worklist里拿到的元素，然後把沒有抓過的經由unseenLinks channel發送給一個爬蟲的goroutine。

seen這個map被限定在main goroutine中；也就是說這個map隻能在main goroutine中進行訪問。類似於其它的信息隱藏方式，這樣的約束可以讓我們從一定程度上保證程序的正確性。例如，內部變量不能夠在函數外部被訪問到；變量(§2.3.4)在沒有被轉義的情況下是無法在函數外部訪問的；一個對象的封裝字段無法被該對象的方法以外的方法訪問到。在所有的情況下，信息隱藏都可以幫助我們約束我們的程序，使其不發生意料之外的情況。

crawl函數爬到的鏈接在一個專有的goroutine中被發送到worklist中來避免死鎖。爲了節省空間，這個例子的終止問題我們先不進行詳細闡述了。

練習8.6: 爲併發爬蟲增加深度限製。也就是說，如果用戶設置了depth=3，那麽隻有從首頁跳轉三次以內能夠跳到的頁面才能被抓取到。

練習8.7: 完成一個併發程序來創建一個線上網站的本地鏡像，把該站點的所有可達的頁面都抓取到本地硬盤。爲了省事，我們這里可以隻取出現在該域下的所有頁面(比如golang.org結尾，譯註：外鏈的應該就不算了。)當然了，出現在頁面里的鏈接你也需要進行一些處理，使其能夠在你的鏡像站點上進行跳轉，而不是指向原始的鏈接。

譯註： 拓展閱讀： http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/

8.7. 基於select的多路複用

下面的程序會進行火箭發射的倒計時。time.Tick函數返回一個channel，程序會週期性地像一個節拍器一樣向這個channel發送事件。每一個事件的值是一個時間戳，不過更有意思的是其傳送方式。

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

現在我們讓這個程序支持在倒計時中，用戶按下return鍵時直接中斷發射流程。首先，我們啟動一個goroutine，這個goroutine會嚐試從標準輸入中調入一個單獨的byte併且，如果成功了，會向名爲abort的channel發送一個值。

```
gopl.io/ch8/countdown2
abort := make
(chan
 struct
{})
go
 func
() {
    os.Stdin.Read(make
([]byte
, 1
)) // read a single byte

    abort <- struct
{}{}
}()
```

現在每一次計數循環的迭代都需要等待兩個channel中的其中一個返迴事件了：ticker channel當一切正常時(就像NASA jorgon的"nominal"，譯註：這梗估計我們是不懂了)或者異常時返迴的abort事件。我們無法做到從每一個channel中接收信息，如果我們這麼做的話，如果第一個channel中沒有事件發過來那麼程序就會立刻被阻塞，這樣我們就無法收到第二個channel中發過來的事件。這時候我們需要多路複用(multiplex)這些操作了，爲了能夠多路複用，我們使用了select語句。

```
select
 {
case
 <-ch1:
    // ...

case
 x := <-ch2:
    // ...use x...

case
 ch3 <- y:
    // ...

default
:
    // ...

}
```

上面是select語句的一般形式。和switch語句稍微有點相似，也會有幾個case和最後的default選擇支。每一個case代表一個通信操作(在某個channel上進行發送或者接收)併且會包含一些語句組成的一個語句塊。一個接收表達式可能隻包含接收表達式自身(譯註：不把接收到的值賦值給變量什麽的)，就像上面的第一個case，或者包含在一個簡短的變量聲明中，像第二個case里一樣；第二種形式讓你能夠引用接收到的值。

select會等待case中有能夠執行的case時去執行。當條件滿足時，select才會去通信併執行case之後的語句；這時候其它通信是不會執行的。一個沒有任何case的select語句寫作select{}，會永遠地等待下去。

讓我們迴到我們的火箭發射程序。time.After函數會立即返迴一個channel，併起一個新的goroutine在經過特定的時間後向該channel發送一個獨立的值。下面的select語句會會一直等待到兩個事件中的一個到達，無論是abort事件或者一個10秒經過的事件。如果10秒經過了還沒有abort事件進入，那麽火箭就會發射。

```
func
 main() {
    // ...create abort channel...


    fmt.Println("Commencing countdown.  Press return to abort."
)
    select
 {
    case
 <-time.After(10
 * time.Second):
        // Do nothing.

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

    }
    launch()
}
```

下面這個例子更微秒。ch這個channel的buffer大小是1，所以會交替的爲空或爲滿，所以隻有一個case可以進行下去，無論i是奇數或者偶數，它都會打印0 2 4 6 8。

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

    case
 ch <- i:
    }
}
```

如果多個case同時就緒時，select會隨機地選擇一個執行，這樣來保證每一個channel都有平等的被select的機會。增加前一個例子的buffer大小會使其輸出變得不確定，因爲當buffer既不爲滿也不爲空時，select語句的執行情況就像是拋硬幣的行爲一樣是隨機的。

下面讓我們的發射程序打印倒計時。這里的select語句會使每次循環迭代等待一秒來執行退出操作。

```
gopl.io/ch8/countdown3
func
 main() {
    // ...create abort channel...



    fmt.Println("Commencing countdown.  Press return to abort."
)
    tick := time.Tick(1
 * time.Second)
    for
 countdown := 10
; countdown > 0
; countdown-- {
        fmt.Println(countdown)
        select
 {
        case
 <-tick:
            // Do nothing.

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

        }
    }
    launch()
}
```

time.Tick函數表現得好像它創建了一個在循環中調用time.Sleep的goroutine，每次被喚醒時發送一個事件。當countdown函數返迴時，它會停止從tick中接收事件，但是ticker這個goroutine還依然存活，繼續徒勞地嚐試從channel中發送值，然而這時候已經沒有其它的goroutine會從該channel中接收值了--這被稱爲goroutine洩露(§8.4.4)。

Tick函數挺方便，但是隻有當程序整個生命週期都需要這個時間時我們使用它才比較合適。否則的話，我們應該使用下面的這種模式：

```
ticker := time.NewTicker(1
 * time.Second)
<-ticker.C    // receive from the ticker's channel

ticker.Stop() // cause the ticker's goroutine to terminate
```

有時候我們希望能夠從channel中發送或者接收值，併避免因爲發送或者接收導致的阻塞，尤其是當channel沒有準備好寫或者讀時。select語句就可以實現這樣的功能。select會有一個default來設置當其它的操作都不能夠馬上被處理時程序需要執行哪些邏輯。

下面的select語句會在abort channel中有值時，從其中接收值；無值時什麼都不做。這是一個非阻塞的接收操作；反複地做這樣的操作叫做“輪詢channel”。

```
select
 {
case
 <-abort:
    fmt.Printf("Launch aborted!\n"
)
    return

default
:
    // do nothing

}
```

channel的零值是nil。也許會讓你覺得比較奇怪，nil的channel有時候也是有一些用處的。因爲對一個nil的channel發送和接收操作會永遠阻塞，在select語句中操作nil的channel永遠都不會被select到。

這使得我們可以用nil來激活或者禁用case，來達成處理其它輸入或輸出事件時超時和取消的邏輯。我們會在下一節中看到一個例子。

練習8.8: 使用select來改造8.3節中的echo服務器，爲其增加超時，這樣服務器可以在客戶端10秒中沒有任何喊話時自動斷開連接。

8.8. 示例: 併發的字典遍歷

在本小節中，我們會創建一個程序來生成指定目録的硬盤使用情況報告，這個程序和Unix里的du工具比較相似。大多數工作用下面這個walkDir函數來完成，這個函數使用dirents函數來枚舉一個目録下的所有入口。

```
gopl.io/ch8/du1
// walkDir recursively walks the file tree rooted at dir

// and sends the size of each found file on fileSizes.

func
 walkDir(dir string
, fileSizes chan
<- int64
) {
    for
 _, entry := range
 dirents(dir) {
        if
 entry.IsDir() {
            subdir := filepath.Join(dir, entry.Name())
            walkDir(subdir, fileSizes)
        } else
 {
            fileSizes <- entry.Size()
        }
    }
}

// dirents returns the entries of directory dir.

func
 dirents(dir string
) []os.FileInfo {
    entries, err := ioutil.ReadDir(dir)
    if
 err != nil
 {
        fmt.Fprintf(os.Stderr, "du1: %v\n"
, err)
        return
 nil

    }
    return
 entries
}
```

ioutil.ReadDir函數會返迴一個os.FileInfo類型的slice，os.FileInfo類型也是os.Stat這個函數的返迴值。對每一個子目録而言，walkDir會遞歸地調用其自身，併且會對每一個文件也遞歸調用。walkDir函數會向fileSizes這個channel發送一條消息。這條消息包含了文件的字節大小。

下面的主函數，用了兩個goroutine。後台的goroutine調用walkDir來遍歷命令行給出的每一個路徑併最終關閉fileSizes這個channel。主goroutine會對其從channel中接收到的文件大小進行纍加，併輸出其和。

```
package
 main

import
 (
    "flag"

    "fmt"

    "io/ioutil"

    "os"

    "path/filepath"

)

func
 main() {
    // Determine the initial directories.

    flag.Parse()
    roots := flag.Args()
    if
 len
(roots) == 0
 {
        roots = []string
{"."
}
    }

    // Traverse the file tree.

    fileSizes := make
(chan
 int64
)
    go
 func
() {
        for
 _, root := range
 roots {
            walkDir(root, fileSizes)
```

```
        }
        close
(fileSizes)
    }()

    // Print the results.

    var
 nfiles, nbytes int64

    for
 size := range
 fileSizes {
        nfiles++
        nbytes += size
    }
    printDiskUsage(nfiles, nbytes)
}

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

這個程序會在打印其結果之前卡住很長時間。

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

如果在運行的時候能夠讓我們知道處理進度的話想必更好。但是，如果簡單地把printDiskUsage函數調用移動到循環里會導致其打印出成百上韆的輸出。

下面這個du的變種會間歇打印內容，不過隻有在調用時提供了-v的flag才會顯示程序進度信息。在roots目録上循環的後台goroutine在這里保持不變。主goroutine現在使用了計時器來每500ms生成事件，然後用select語句來等待文件大小的消息來更新總大小數據，或者一個計時器的事件來打印當前的總大小數據。如果-v的flag在運行時沒有傳入的話，tick這個channel會保持爲nil，這樣在select里的case也就相當於被禁用了。

```
gopl.io/ch8/du2
var
 verbose = flag.Bool("v"
, false
, "show verbose progress messages"
)

func
 main() {
    // ...start background goroutine...



    // Print the results periodically.

    var
 tick <-chan
 time.Time
    if
 *verbose {
        tick = time.Tick(500
 * time.Millisecond)
    }
    var
 nfiles, nbytes int64

loop:
    for
 {
        select
 {
        case
 size, ok := <-fileSizes:
            if
 !ok {
                break
 loop // fileSizes was closed

            }
            nfiles++
            nbytes += size
        case
 <-tick:
            printDiskUsage(nfiles, nbytes)
        }
    }
    printDiskUsage(nfiles, nbytes) // final totals

}
```

由於我們的程序不再使用range循環，第一個select的case必鬚顯式地判斷fileSizes的channel是不是已經被關閉了，這里可以用到channel接收的二值形式。如果channel已經被關閉了的話，程序會直接退出循環。這里的break語句用到了標籤break，這樣可以同時終結select和for兩個循環；如果沒有用標籤就break的話隻會退出內層的select循環，而外層的for循環會使之進入下一輪select循環。

現在程序會悠閒地爲我們打印更新流：

```
$ go build gopl.io/ch8/du2
$ ./du2 -v $HOME /usr /bin /etc
28608 files  8.3 GB
54147 files  10.3 GB
93591 files  15.1 GB
127169 files  52.9 GB
175931 files  62.2 GB
213201 files  62.7 GB
```

然而這個程序還是會花上很長時間才會結束。無法對walkDir做併行化處理沒什麼別的原因，無非是因爲磁盤繫統併行限製。下面這個第三個版本的du，會對每一個walkDir的調用創建一個新的goroutine。它使用sync.WaitGroup (§8.5)來對仍舊活躍的walkDir調用進行計數，另一個goroutine會在計數器減爲零的時候將fileSizes這個channel關閉。

```
gopl.io/ch8/du3
func
 main() {
    // ...determine roots...

    // Traverse each root of the file tree in parallel.

    fileSizes := make
(chan
 int64
)
    var
 n sync.WaitGroup
    for
 _, root := range
 roots {
        n.Add(1
)
        go
 walkDir(root, &n, fileSizes)
    }
    go
 func
() {
        n.Wait()
        close
(fileSizes)
    }()
    // ...select loop...

}
```

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

由於這個程序在高峯期會創建成百上韆的goroutine，我們需要脩改dirents函數，用計數信號量來阻止他同時打開太多的文件，就像我們在8.7節中的併發爬蟲一樣：

```
// sema is a counting semaphore for limiting concurrency in dirents.

var
 sema = make
(chan
 struct
{}, 20
)

// dirents returns the entries of directory dir.

func
 dirents(dir string
) []os.FileInfo {
    sema <- struct
{}{}        // acquire token

    defer
 func
() { <-sema }() // release token

    // ...
```

這個版本比之前那個快了好幾倍，盡管其具體效率還是和你的運行環境，機器配置相關。

練習8.9: 編寫一個du工具，每隔一段時間將root目録下的目録大小計算併顯示出來。

8.9. 併發的退出

有時候我們需要通知goroutine停止它正在幹的事情，比如一個正在執行計算的web服務，然而它的客戶端已經斷開了和服務端的連接。

Go語言併沒有提供在一個goroutine中終止另一個goroutine的方法，由於這樣會導致goroutine之間的共享變量落在未定義的狀態上。在8.7節中的rocket launch程序中，我們往名字叫abort的channel里發送了一個簡單的值，在countdown的goroutine中會把這個值理解爲自己的退出信號。但是如果我們想要退出兩個或者任意多個goroutine怎麼辦呢？

一種可能的手段是向abort的channel里發送和goroutine數目一樣多的事件來退出它們。如果這些goroutine中已經有一些自己退出了，那麼會導致我們的channel里的事件數比goroutine還多，這樣導致我們的發送直接被阻塞。另一方面，如果這些goroutine又生成了其它的goroutine，我們的channel里的數目又太少了，所以有些goroutine可能會無法接收到退出消息。一般情況下我們是很難知道在某一個時刻具體有多少個goroutine在運行着的。另外，當一個goroutine從abort channel中接收到一個值的時候，他會消費掉這個值，這樣其它的goroutine就沒法看到這條信息。爲了能夠達到我們退出goroutine的目的，我們需要更靠譜的策略，來通過一個channel把消息廣播出去，這樣goroutine們能夠看到這條事件消息，併且在事件完成之後，可以知道這件事已經發生過了。

迴憶一下我們關閉了一個channel併且被消費掉了所有已發送的值，操作channel之後的代碼可以立即被執行，併且會產生零值。我們可以將這個機製擴展一下，來作爲我們的廣播機製：不要向channel發送值，而是用關閉一個channel來進行廣播。

隻要一些小脩改，我們就可以把退出邏輯加入到前一節的du程序。首先，我們創建一個退出的channel，這個channel不會向其中發送任何值，但其所在的閉包內要寫明程序需要退出。我們同時還定義了一個工具函數，cancelled，這個函數在被調用的時候會輪詢退出狀態。

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

func
 cancelled() bool
 {
    select
 {
    case
 <-done:
        return
 true

    default
:
        return
 false

    }
}
```

下面我們創建一個從標準輸入流中讀取內容的goroutine，這是一個比較典型的連接到終端的程序。每當有輸入被讀到(比如用戶按了迴車鍵)，這個goroutine就會把取消消息通過關閉done的channel廣播出去。

```
// Cancel traversal when input is detected.

go
 func
() {
    os.Stdin.Read(make
([]byte
, 1
)) // read a single byte

    close
(done)
}()
```

現在我們需要使我們的goroutine來對取消進行響應。在main goroutine中，我們添加了select的第三個case語句，嚐試從done channel中接收內容。如果這個case被滿足的話，在select到的時候即會返迴，但在結束之前我們需要把fileSizes channel中的內容“排”空，在channel被關閉之前，舍棄掉所有值。這樣可以保證對walkDir的調用不要被向fileSizes發送信息阻塞住，可以正確地完成。

```
for
 {
    select
 {
    case
 <-done:
        // Drain fileSizes to allow existing goroutines to finish.

        for
 range
 fileSizes {
            // Do nothing.

        }
        return

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

    }
}
```

walkDir這個goroutine一啟動就會輪詢取消狀態，如果取消狀態被設置的話會直接返迴，併且不做額外的事情。這樣我們將所有在取消事件之後創建的goroutine改變爲無操作。

```
func
 walkDir(dir string
, n *sync.WaitGroup, fileSizes chan
<- int64
) {
    defer
 n.Done()
    if
 cancelled() {
        return

    }
    for
 _, entry := range
 dirents(dir) {
        // ...

    }
}
```

在walkDir函數的循環中我們對取消狀態進行輪詢可以帶來明顯的益處，可以避免在取消事件發生時還去創建goroutine。取消本身是有一些代價的；想要快速的響應需要對程序邏輯進行侵入式的脩改。確保在取消發生之後不要有代價太大的操作可能會需要脩改你代碼里的很多地方，但是在一些重要的地方去檢查取消事件也確實能帶來很大的好處。

對這個程序的一個簡單的性能分析可以揭示瓶頸在dirents函數中獲取一個信號量。下面的select可以讓這種操作可以被取消，併且可以將取消時的延遲從幾百毫秒降低到幾十毫秒。

```
func
 dirents(dir string
) []os.FileInfo {
    select
 {
    case
 sema <- struct
{}{}: // acquire token

    case
 <-done:
        return
 nil
 // cancelled

    }
    defer
 func
() { <-sema }() // release token

    // ...read directory...

}
```

現在當取消發生時，所有後台的goroutine都會迅速停止併且主函數會返迴。當然，當主函數返迴時，一個程序會退出，而我們又無法在主函數退出的時候確認其已經釋放了所有的資源(譯註：因爲程序都退出了，你的代碼都沒法執行了)。這里有一個方便的竅門我們可以一用：取代掉直接從主函數返迴，我們調用一個panic，然後runtime會把每一個goroutine的棧dump下來。如果main goroutine是唯一一個剩下的goroutine的話，他會清理掉自己的一切資源。但是如果還有其它的goroutine沒有退出，他們可能沒辦法被正確地取消掉，也有可能被取消但是取消操作會很花時間；所以這里的一個調研還是很有必要的。我們用panic來獲取到足夠的信息來驗證我們上面的判斷，看看最終到底是什麼樣的情況。

練習8.10: HTTP請求可能會因http.Request結構體中Cancel channel的關閉而取消。脩改8.6節中的web crawler來支持取消http請求。

提示: http.Get併沒有提供方便地定製一個請求的方法。你可以用http.NewRequest來取而代之，設置它的Cancel字段，然後用http.DefaultClient.Do(req)來進行這個http請求。

練習8.11:緊接着8.4.4中的mirroredQuery流程，實現一個併發請求url的fetch的變種。當第一個請求返迴時，直接取消其它的請求。

8.10. 示例: 聊天服務

我們用一個聊天服務器來終結本章節的內容，這個程序可以讓一些用戶通過服務器向其它所有用戶廣播文本消息。這個程序中有四種goroutine。main和broadcaster各自是一個goroutine實例，每一個客戶端的連接都會有一個handleConn和clientWriter的goroutine。broadcaster是select用法的不錯的樣例，因爲它需要處理三種不同類型的消息。 下面演示的main goroutine的工作，是listen和accept(譯註：網絡編程里的概念)從客戶端過來的連接。對每一個連接，程序都會建立一個新的handleConn的goroutine，就像我們在本章開頭的併發的echo服務器里所做的那樣。

```
gopl.io/ch8/chat
func
 main() {
    listener, err := net.Listen("tcp"
, "localhost:8000"
)
    if
 err != nil
 {
        log.Fatal(err)
    }
    go
 broadcaster()
    for
 {
        conn, err := listener.Accept()
        if
 err != nil
 {
            log.Print(err)
            continue

        }
        go
 handleConn(conn)
    }
}
```

然後是broadcaster的goroutine。他的內部變量clients會記錄當前建立連接的客戶端集合。其記錄的內容是每一個客戶端的消息發出channel的"資格"信息。

```
type
 client chan
<- string
 // an outgoing message channel


var
 (
    entering = make
(chan
 client)
```

```
    leaving  = make
(chan
 client)
    messages = make
(chan
 string
) // all incoming client messages

)

func
 broadcaster() {
    clients := make
(map
[client]bool
) // all connected clients

    for
 {
        select
 {
        case
 msg := <-messages:
            // Broadcast incoming message to all

            // clients' outgoing message channels.

            for
 cli := range
 clients {
                cli <- msg
            }
        case
 cli := <-entering:
            clients[cli] = true



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

broadcaster監聽來自全局的entering和leaving的channel來獲知客戶端的到來和離開事件。當其接收到其中的一個事件時，會更新clients集合，當該事件是離開行爲時，它會關閉客戶端的消息發出channel。broadcaster也會監聽全局的消息channel，所有的客戶端都會向這個channel中發送消息。當broadcaster接收到什麼消息時，就會將其廣播至所有連接到服務端的客戶端。

現在讓我們看看每一個客戶端的goroutine。handleConn函數會爲它的客戶端創建一個消息發出channel併通過entering channel來通知客戶端的到來。然後它會讀取客戶端發來的每一行文本，併通過全局的消息channel來將這些文本發送出去，併爲每條消息帶上發送者的前綴來標明消息身份。當客戶端發送完畢後，handleConn會通過leaving這個channel來通知客戶端的離開併關閉連接。

```
func
 handleConn(conn net.Conn) {
    ch := make
(chan
 string
) // outgoing client messages

    go
 clientWriter(conn, ch)

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

    entering <- ch

    input := bufio.NewScanner(conn)
    for
 input.Scan() {
        messages <- who + ": "
 + input.Text()
    }
    // NOTE:
 ignoring potential errors from input.Err()


    leaving <- ch
    messages <- who + " has left"

    conn.Close()
}

func
 clientWriter(conn net.Conn, ch <-chan
 string
) {
    for
 msg := range
 ch {
    fmt.Fprintln(conn, msg) // NOTE:
 ignoring network errors

    }
}
```

另外，handleConn爲每一個客戶端創建了一個clientWriter的goroutine來接收向客戶端發出消息channel中發送的廣播消息，併將它們寫入到客戶端的網絡連接。客戶端的讀取方循環會在broadcaster接收到leaving通知併關閉了channel後終止。

下面演示的是當服務器有兩個活動的客戶端連接，併且在兩個窗口中運行的情況，使用netcat來聊天：

```
$ go build gopl.io/ch8/chat
$ go build gopl.io/ch8/netcat3
$ ./chat &
$ ./netcat3
You are 127.0.0.1:64208               $ ./netcat3
127.0.0.1:64211 has arrived           You are 127.0.0.1:64211
Hi!
127.0.0.1:64208: Hi!
127.0.0.1:64208: Hi!
                                      Hi yourself.
127.0.0.1:64211: Hi yourself.         127.0.0.1:64211: Hi yourself.
^C
                                      127.0.0.1:64208 has left
$ ./netcat3
You are 127.0.0.1:64216               127.0.0.1:64216 has arrived
                                      Welcome.
127.0.0.1:64211: Welcome.             127.0.0.1:64211: Welcome.
                                      ^C
127.0.0.1:64211 has left”
```

當與n個客戶端保持聊天session時，這個程序會有2n+2個併發的goroutine，然而這個程序卻併不需要顯式的鎖(§9.2)。clients這個map被限製在了一個獨立的goroutine中，broadcaster，所以它不能被併發地訪問。多個goroutine共享的變量隻有這些channel和net.Conn的實例，兩個東西都是併發安全的。我們會在下一章中更多地解決約束，併發安全以及goroutine中共享變量的含義。

練習8.12: 使broadcaster能夠將arrival事件通知當前所有的客戶端。爲了達成這個目的，你需要有一個客戶端的集合，併且在entering和leaving的channel中記録客戶端的名字。 練習8.13: 使聊天服務器能夠斷開空閒的客戶端連接，比如最近五分鐘之後沒有發送任何消息的那些客戶端。提示：可以在其它goroutine中調用conn.Close()來解除Read調用，就像input.Scanner()所做的那樣。 練習8.14: 脩改聊天服務器的網絡協議這樣每一個客戶端就可以在entering時可以提供它們的名字。將消息前綴由之前的網絡地址改爲這個名字。 練習8.15: 如果一個客戶端沒有及時地讀取數據可能會導致所有的客戶端被阻塞。脩改broadcaster來跳過一條消息，而不是等待這個客戶端一直到其準備好寫。或者爲每一個客戶端的消息發出channel建立緩衝區，這樣大部分的消息便不會被丟掉；broadcaster應該用一個非阻塞的send向這個channel中發消息。

第九章 基於共享變量的併發

前一章我們介紹了一些使用goroutine和channel這樣直接而自然的方式來實現併發的方法。然而這樣做我們實際上屏蔽掉了在寫併發代碼時必鬚處理的一些重要而且細微的問題。

在本章中，我們會細致地了解併發機製。尤其是在多goroutine之間的共享變量，併發問題的分析手段，以及解決這些問題的基本模式。最後我們會解釋goroutine和操作繫統線程之間的技術上的一些區別。

9.1. 競爭條件

TODO

9.2. sync.Mutex互斥鎖

TODO

9.3. sync.RWMutex讀寫鎖

在100刀的存款消失時不做記録多少還是會讓我們有一些恐慌，Bob寫了一個程序，每秒運行幾百次來檢查他的銀行餘額。他會在家，在工作中，甚至會在他的手機上來運行這個程序。銀行註意到這些陡增的流量使得存款和取款有了延時，因爲所有的餘額查詢請求是順序執行的，這樣會互斥地獲得鎖，併且會暫時阻止其它的goroutine運行。

由於Balance函數隻需要讀取變量的狀態，所以我們同時讓多個Balance調用併發運行事實上是安全的，隻要在運行的時候沒有存款或者取款操作就行。在這種場景下我們需要一種特殊類型的鎖，其允許多個隻讀操作併行執行，但寫操作會完全互斥。這種鎖叫作“多讀單寫”鎖(multiple readers, single writer lock)，Go語言提供的這樣的鎖是sync.RWMutex：

```
var
 mu sync.RWMutex
var
 balance int

func
 Balance() int
 {
    mu.RLock() // readers lock

    defer
 mu.RUnlock()
    return
 balance
}
```

Balance函數現在調用了RLock和RUnlock方法來獲取和釋放一個讀取或者共享鎖。Deposit函數沒有變化，會調用mu.Lock和mu.Unlock方法來獲取和釋放一個寫或互斥鎖。

在這次脩改後，Bob的餘額查詢請求就可以彼此併行地執行併且會很快地完成了。鎖在更多的時間范圍可用，併且存款請求也能夠及時地被響應了。

RLock隻能在臨界區共享變量沒有任何寫入操作時可用。一般來說，我們不應該假設邏輯上的隻讀函數/方法也不會去更新某一些變量。比如一個方法功能是訪問一個變量，但它也有可能會同時去給一個內部的計數器+1(譯註：可能是記録這個方法的訪問次數啥的)，或者去更新緩存--使卽時的調用能夠更快。如果有疑惑的話，請使用互斥鎖。

RWMutex隻有當獲得鎖的大部分goroutine都是讀操作，而鎖在競爭條件下，也就是説，goroutine們必鬚等待才能獲取到鎖的時候，RWMutex才是最能帶來好處的。RWMutex需要更複雜的內部記録，所以會讓它比一般的無競爭鎖的mutex慢一些。

9.4. 内存同步

你可能比較糾結爲什麼Balance方法需要用到互斥條件，無論是基於channel還是基於互斥量。畢竟和存款不一樣，它隻由一個簡單的操作組成，所以不會碰到其它goroutine在其執行"中"執行其它的邏輯的風險。這里使用mutex有兩方面考慮。第一Balance不會在其它操作比如Withdraw“中間”執行。第二(更重要)的是"同步"不僅僅是一堆goroutine執行順序的問題；同樣也會涉及到內存的問題。

在現代計算機中可能會有一堆處理器，每一個都會有其本地緩存(local cache)。爲了效率，對內存的寫入一般會在每一個處理器中緩衝，併在必要時一起flush到主存。這種情況下這些數據可能會以與當初goroutine寫入順序不同的順序被提交到主存。像channel通信或者互斥量操作這樣的原語會使處理器將其聚集的寫入flush併commit，這樣goroutine在某個時間點上的執行結果才能被其它處理器上運行的goroutine得到。

考慮一下下面代碼片段的可能輸出：

```
var
 x, y int

go
 func
() {
    x = 1
 // A1

    fmt.Print("y:"
, y, " "
) // A2

}()
go
 func
() {
    y = 1
                    // B1

    fmt.Print("x:"
, x, " "
) // B2

}()
```

因爲兩個goroutine是併發執行，併且訪問共享變量時也沒有互斥，會有數據競爭，所以程序的運行結果沒法預測的話也請不要驚訝。我們可能希望它能夠打印出下面這四種結果中的一種，相當於幾種不同的交錯執行時的情況：

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

第四行可以被解釋爲執行順序A1,B1,A2,B2或者B1,A1,A2,B2的執行結果。 然而實際的運行時還是有些情況讓我們有點驚訝：

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

但是根據所使用的編譯器，CPU，或者其它很多影響因子，這兩種情況也是有可能發生的。那麼這兩種情況要怎麼解釋呢？

在一個獨立的goroutine中，每一個語句的執行順序是可以被保證的；也就是說goroutine是順序連貫的。但是在不使用channel且不使用mutex這樣的顯式同步操作時，我們就沒法保證事件在不同的goroutine中看到的執行順序是一致的了。盡管goroutine A中一定需要觀察到x=1執行成功之後才會去讀取y，但它沒法確保自己觀察得到goroutine B中對y的寫入，所以A還可能會打印出y的一個舊版的值。

盡管去理解併發的一種嚐試是去將其運行理解爲不同goroutine語句的交錯執行，但看看上面的例子，這已經不是現代的編譯器和cpu的工作方式了。因爲賦值和打印指向不同的變量，編譯器可能會斷定兩條語句的順序不會影響執行結果，併且會交換兩個語句的執行順序。如果兩個goroutine在不同的CPU上執行，每一個核心有自己的緩存，這樣一個goroutine的寫入對於其它goroutine的Print，在主存同步之前就是不可見的了。

所有併發的問題都可以用一致的、簡單的既定的模式來規避。所以可能的話，將變量限定在goroutine內部；如果是多個goroutine都需要訪問的變量，使用互斥條件來訪問。

9.5. sync.Once初始化

TODO

9.6. 競爭條件檢測

即使我們小心到不能再小心，但在併發程序中犯錯還是太容易了。幸運的是，Go的runtime和工具鏈爲我們裝備了一個複雜但好用的動態分析工具，競爭檢查器(the race detector)。

隻要在go build，go run或者go test命令後面加上-race的flag，就會使編譯器創建一個你的應用的“脩改”版或者一個附帶了能夠記録所有運行期對共享變量訪問工具的test，併且會記録下每一個讀或者寫共享變量的goroutine的身份信息。另外，脩改版的程序會記録下所有的同步事件，比如go語句，channel操作，以及對(*sync.Mutex).Lock，(*sync.WaitGroup).Wait等等的調用。(完整的同步事件集合是在The Go Memory Model文檔中有説明，該文檔是和語言文檔放在一起的。譯註：https://golang.org/ref/mem)

競爭檢查器會檢查這些事件，會尋找在哪一個goroutine中出現了這樣的case，例如其讀或者寫了一個共享變量，這個共享變量是被另一個goroutine在沒有進行幹預同步操作便直接寫入的。這種情況也就表明了是對一個共享變量的併發訪問，即數據競爭。這個工具會打印一份報告，內容包含變量身份，讀取和寫入的goroutine中活躍的函數的調用棧。這些信息在定位問題時通常很有用。9.7節中會有一個競爭檢查器的實戰樣例。

競爭檢查器會報告所有的已經發生的數據競爭。然而，它隻能檢測到運行時的競爭條件；併不能證明之後不會發生數據競爭。所以爲了使結果盡量正確，請保證你的測試併發地覆蓋到了你到包。

由於需要額外的記録，因此構建時加了競爭檢測的程序跑起來會慢一些，且需要更大的內存，即時是這樣，這些代價對於很多生産環境的工作來説還是可以接受的。對於一些偶發的競爭條件來説，讓競爭檢查器來幹活可以節省無數日夜的debugging。(譯註：多少服務端C和C艹程序員爲此盡摺腰)

9.7. 示例: 併發的非阻塞緩存

TODO

9.8. Goroutines和線程

在上一章中我們說goroutine和操作繫統的線程區別可以先忽略。盡管兩者的區別實際上隻是一個量的區別，但量變會引起質變的道理同樣適用於goroutine和線程。現在正是我們來區分開兩者的最佳時機。

9.8.1. 動態棧

每一個OS線程都有一個固定大小的內存塊(一般會是2MB)來做棧，這個棧會用來存儲當前正在被調用或掛起(指在調用其它函數時)的函數的內部變量。這個固定大小的棧同時很大又很小。因爲2MB的棧對於一個小小的goroutine來說是很大的內存浪費，比如對於我們用到的，一個隻是用來WaitGroup之後關閉channel的goroutine來說。而對於go程序來說，同時創建成百上韆個gorutine是非常普遍的，如果每一個goroutine都需要這麼大的棧的話，那這麼多的goroutine就不太可能了。除去大小的問題之外，固定大小的棧對於更複雜或者更深層次的遞歸函數調用來說顯然是不夠的。脩改固定的大小可以提陞空間的利用率允許創建更多的線程，併且可以允許更深的遞歸調用，不過這兩者是沒法同時兼備的。

相反，一個goroutine會以一個很小的棧開始其生命週期，一般隻需要2KB。一個goroutine的棧，和操作繫統線程一樣，會保存其活躍或掛起的函數調用的本地變量，但是和OS線程不太一樣的是一個goroutine的棧大小併不是固定的；棧的大小會根據需要動態地伸縮。而goroutine的棧的最大值有1GB，比傳統的固定大小的線程棧要大得多，盡管一般情況下，大多goroutine都不需要這麼大的棧。

練習 9.4: 創建一個流水線程序，支持用channel連接任意數量的goroutine，在跑爆內存之前，可以創建多少流水線階段？一個變量通過整個流水線需要用多久？(這個練習題翻譯不是很確定。。)

9.8.2. Goroutine調度

OS線程會被操作繫統內核調度。每幾毫秒，一個硬件計時器會中斷處理器，這會調用一個叫作scheduler的內核函數。這個函數會掛起當前執行的線程併保存內存中它的寄存器內容，檢查線程列表併決定下一次哪個線程可以被運行，併從內存中恢複該線程的寄存器信息，然後恢複執行該線程的現場併開始執行線程。因爲操作繫統線程是被內核所調度，所以從一個線程向另一個“移動”需要完整的上下文切換，也就是說，保存一個用戶線程的狀態到內存，恢複另一個線程的到寄存器，然後更新調度器的數據結構。這幾步操作很慢，因爲其局部性很差需要幾次內存訪問，併且會增加運行的cpu週期。

Go的運行時包含了其自己的調度器，這個調度器使用了一些技術手段，比如m:n調度，因爲其會在n個操作繫統線程上多工(調度)m個goroutine。Go調度器的工作和內核的調度是相似的，但是這個調度器隻關註單獨的Go程序中的goroutine(譯註：按程序獨立)。

和操作繫統的線程調度不同的是，Go調度器併不是用一個硬件定時器而是被Go語言"建築"本身進行調度的。例如當一個goroutine調用了time.Sleep或者被channel調用或者mutex操作阻塞時，調度器會使其進入休眠併開始執行另一個goroutine直到時機到了再去喚醒第一個goroutine。因爲因爲這種調度方式不需要進入內核的上下文，所以重新調度一個goroutine比調度一個線程代價要低得多。

練習 9.5: 寫一個有兩個goroutine的程序，兩個goroutine會向兩個無buffer channel反複地發送ping-pong消息。這樣的程序每秒可以支持多少次通信？

9.8.3. GOMAXPROCS

Go的調度器使用了一個叫做GOMAXPROCS的變量來決定會有多少個操作繫統的線程同時執行Go的代碼。其默認的值是運行機器上的CPU的核心數，所以在一個有8個核心的機器上時，調度器一次會在8個OS線程上去調度GO代碼。(GOMAXPROCS是前面說的m:n調度中的n)。在休眠中的或者在通信中被阻塞的goroutine是不需要一個對應的線程來做調度的。在I/O中或繫統調用中或調用非Go語言函數時，是需要一個對應的操作繫統線程的，但是GOMAXPROCS併不需要將這幾種情況計數在內。

你可以用GOMAXPROCS的環境變量呂顯式地控製這個參數，或者也可以在運行時用runtime.GOMAXPROCS函數來脩改它。我們在下面的小程序中會看到GOMAXPROCS的效果，這個程序會無限打印0和1。

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

$ GOMAXPROCS=1
 go
 run hacker-cliché go

11111111111111111111000000000000000000000011111.
..

$ GOMAXPROCS=2
 go
 run hacker-cliché go

01010101010101010101100110010101101001010 0110.
..
```

在第一次執行時，最多同時隻能有一個goroutine被執行。初始情況下隻有main goroutine被執行，所以會打印很多1。過了一段時間後，GO調度器會將其置爲休眠，併喚醒另一個goroutine，這時候就開始打印很多0了，在打印的時候，goroutine是被調度到操作繫統線程上的。在第二次執行時，我們使用了兩個操作繫統線程，所以兩個goroutine可以一起被執行，以同樣的頻率交替打印0和1。我們必鬚強調的是goroutine的調度是受很多因子影響的，而runtime也是在不斷地發展演進的，所以這里的你實際得到的結果可能會因爲版本的不同而與我們運行的結果有所不同。

練習9.6: 測試一下計算密集型的併發程序(練習8.5那樣的)會被GOMAXPROCS怎樣影響到。在你的電腦上最佳的值是多少？你的電腦CPU有多少個核心？

9.8.4. Goroutine沒有ID號

在大多數支持多線程的操作繫統和程序語言中，當前的線程都有一個獨特的身份(id)，併且這個身份信息可以以一個普通值的形式被被很容易地獲取到，典型的可以是一個integer或者指針值。這種情況下我們做一個抽象化的thread-local storage(線程本地存儲，多線程編程中不希望其它線程訪問的內容)就很容易，隻需要以線程的id作爲key的一個map就可以解決問題，每一個線程以其id就能從中獲取到值，且和其它線程互不衝突。

goroutine沒有可以被程序員獲取到的身份(id)的概念。這一點是設計上故意而爲之，由於thread-local storage總是會被濫用。比如說，一個web server是用一種支持tls的語言實現的，而非常普遍的是很多函數會去尋找HTTP請求的信息，這代表它們就是去其存儲層(這個存儲層有可能是tls)查找的。這就像是那些過分依賴全局變量的程序一樣，會導致一種非健康的“距離外行爲”，在這種行爲下，一個函數的行爲可能不是由其自己內部的變量所決定，而是由其所運行在的線程所決定。因此，如果線程本身的身份會改變--比如一些worker線程之類的--那麼函數的行爲就會變得神祕莫測。

Go鼓勵更爲簡單的模式，這種模式下參數對函數的影響都是顯式的。這樣不僅使程序變得更易讀，而且會讓我們自由地向一些給定的函數分配子任務時不用擔心其身份信息影響行爲。

你現在應該已經明白了寫一個Go程序所需要的所有語言特性信息。在後面兩章節中，我們會迴顧一些之前的實例和工具，支持我們寫出更大規模的程序：如何將一個工程組織成一繫列的包，如果獲取，構建，測試，性能測試，剖析，寫文檔，併且將這些包分享出去。

第十章 包和工具

現在隨便一個小程序的實現都可能包含超過10000個函數。然而作者一般隻需要考慮其中很小的一部分和做很少的設計，因爲絶大部分代碼都是由他人編寫的，它們通過類似包或模塊的方式被重用。

Go語言有超過100個的標準包（譯註：可以用 `go list std | wc -l` 命令查看標準包的具體數目），標準庫爲大多數的程序提供了必要的基礎構件。在Go的社區，有很多成熟的包被設計、共享、重用和改進，目前互聯網上已經發布了非常多的Go語音開源包，它們可以通過 http://godoc.org 檢索。在本章，我們將演示如果使用已有的包和創建新的包。

Go還自帶了工具箱，里面有很多用來簡化工作區和包管理的小工具。在本書開始的時候，我們已經見識過如何使用工具箱自帶的工具來下載、構件和運行我們的演示程序了。在本章，我們將看看這些工具的基本設計理論和嚐試更多的功能，例如打印工作區中包的文檔和查詢相關的元數據等。在下一章，我們將探討探索包的單元測試用法。

10.1. 包簡介

任何包繫統設計的目的都是爲了簡化大型程序的設計和維護工作，通過將一組相關的特性放進一個獨立的單元以便於理解和更新，在每個單元更新的同時保持和程序中其它單元的相對獨立性。這種模塊化的特性允許每個包可以被其它的不同項目共享和重用，在項目范圍內、甚至全球范圍統一的分發和複用。

每個包一般都定義了一個不同的名字空間用於它內部的每個標識符的訪問。每個名字空間關聯到一個特定的包，讓我們給類型、函數等選擇簡短明了的名字，這樣可以避免在我們使用它們的時候減少和其它部分名字的衝突。

每個包還通過控製包內名字的可見性和是否導出來實現封裝特性。通過限製包成員的可見性併隱藏包API的具體實現，將允許包的維護者在不影響外部包用戶的前提下調整包的內部實現。通過限製包內變量的可見性，還可以強製用戶通過某些特定函數來訪問和更新內部變量，這樣可以保證內部變量的一致性和併發時的互斥約束。

當我們脩改了一個源文件，我們必鬚重新編譯該源文件對應的包和所有依賴該包的其他包。即使是從頭構建，Go語言編譯器的編譯速度也明顯快於其它編譯語言。Go語言的閃電般的編譯速度主要得益於三個語言特性。第一點，所有導入的包必鬚在每個文件的開頭顯式聲明，這樣的話編譯器就沒有必要讀取和分析整個源文件來判斷包的依賴關繫。第二點，禁止包的環狀依賴，因爲沒有循環依賴，包的依賴關繫形成一個有向無環圖，每個包可以被獨立編譯，而且很可能是被併發編譯。第三點，編譯後包的目標文件不僅僅記録包本身的導出信息，目標文件同時還記録了包的依賴關繫。因此，在編譯一個包的時候，編譯器隻需要讀取每個直接導入包的目標文件，而不需要遍歷所有依賴的的文件（譯註：很多都是重複的間接依賴）。

10.2. 導入路徑

每個包是由一個全局唯一的字符串所標識的導入路徑定位。出現在import語句中的導入路徑也是字符串。

```
import
 (
    "fmt"

    "math/rand"

    "encoding/json"


    "golang.org/x/net/html"


    "github.com/go-sql-driver/mysql"

)
```

就像我們在2.6.1節提到過的，Go語言的規范併沒有指明包的導入路徑字符串的具體含義，導入路徑的具體含義是由構建工具來解釋的。在本章，我們將深入討論Go語言工具箱的功能，包括大家經常使用的構建測試等功能。當然，也有第三方擴展的工具箱存在。例如，Google公司內部的Go語言碼農，他們就使用內部的多語言構建繫統（譯註：Google公司使用的是類似Bazel的構建繫統，支持多種編程語言，目前該構件繫統還不能完整支持Windows環境），用不同的規則來處理包名字和定位包，用不同的規則來處理單元測試等等，因爲這樣可以更緊密適配他們內部環境。

如果你計劃分享或發布包，那麼導入路徑最好是全球唯一的。爲了避免衝突，所有非標準庫包的導入路徑建議以所在組織的互聯網域名爲前綴；而且這樣也有利於包的檢索。例如，上面的import語句導入了Go糰隊維護的HTML解析器和一個流行的第三方維護的MySQL驅動。

10.3. 包聲明

在每個Go語音源文件的開頭都必鬚有包聲明語句。包聲明語句的主要目的是確定當前包被其它包導入時默認的標識符（也稱爲包名）。

例如，math/rand包的每個源文件的開頭都包含 `package rand` 包聲明語句，所以當你導入這個包，你就可以用rand.Int、rand.Float64類似的方式訪問包的成員。

```
package main

import (
    "fmt"

    "math/rand"

)

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

通常來說，默認的包名就是包導入路徑名的最後一段，因此即使兩個包的導入路徑不同，它們依然可能有一個相同的包名。例如，math/rand包和crypto/rand包的包名都是rand。稍後我們將看到如何同時導入兩個有相同包名的包。

關於默認包名一般采用導入路徑名的最後一段的約定也有三種例外情況。第一個例外，包對應一個可執行程序，也就是main包，這時候main包本身的導入路徑是無關緊要的。名字爲main的包是給go build（§10.7.3）構建命令一個信息，這個包編譯完之後必鬚調用連接器生成一個可執行程序。

第二個例外，包所在的目録中可能有一些文件名是以*test.go爲後綴的Go源文件（譯註：前面必鬚有其它的字符，因爲以*``前綴的源文件是被忽略的），併且這些源文件聲明的包名也是以_test爲後綴名的。這種目録可以包含兩種包：一種普通包，加一種則是測試的外部擴展包。所有以_test爲後綴包名的測試外部擴展包都由go test命令獨立編譯，普通包和測試的外部擴展包是相互獨立的。測試的外部擴展包一般用來避免測試代碼中的循環導入依賴，具體細節我們將在11.2.4節中介紹。

第三個例外，一些依賴版本號的管理工具會在導入路徑後追加版本號信息，例如"gopkg.in/yaml.v2"。這種情況下包的名字併不包含版本號後綴，而是yaml。

10.4. 導入聲明

可以在一個Go語言源文件包聲明語句之後，其它非導入聲明語句之前，包含零到多個導入包聲明語句。每個導入聲明可以單獨指定一個導入路徑，也可以通過圓括號同時導入多個導入路徑。下面兩個導入形式是等價的，但是第二種形式更爲常見。

```
import "fmt"

import "os"


import (
    "fmt"

    "os"

)
```

導入的包之間可以通過添加空行來分組；通常將來自不同組織的包獨自分組。包的導入順序無關緊要，但是在每個分組中一般會根據字符串順序排列。（gofmt和goimports工具都可以將不同分組導入的包獨立排序。）

```
import (
    "fmt"

    "html/template"

    "os"


    "golang.org/x/net/html"

    "golang.org/x/net/ipv4"

)
```

如果我們想同時導入兩個有着名字相同的包，例如math/rand包和crypto/rand包，那麼導入聲明必鬚至少爲一個同名包指定一個新的包名以避免衝突。這叫做導入包的重命名。

```
import
 (
    "crypto/rand"

    mrand "math/rand"
 // alternative name mrand avoids conflict

)
```

導入包的重命名隻影響當前的源文件。其它的源文件如果導入了相同的包，可以用導入包原本默認的名字或重命名爲另一個完全不同的名字。

導入包重命名是一個有用的特性，它不僅僅隻是爲了解決名字衝突。如果導入的一個包名很笨重，特別是在一些自動生成的代碼中，這時候用一個簡短名稱會更方便。選擇用簡短名稱重命名導入包時候最好統一，以避免包名混亂。選擇另一個包名稱還可以幫助避免和本地普通變量名產生衝突。例如，如果文件中已經有了一個名爲path的變量，那麼我們可以將"path"標準包重命名爲pathpkg。

每個導入聲明語句都明確指定了當前包和被導入包之間的依賴關繫。如果遇到包循環導入的情況，Go語言的構建工具將報告錯誤。

10.5. 包的匿名導入

如果隻是導入一個包而併不使用導入的包將會導致一個編譯錯誤。但是有時候我們隻是想利用導入包而産生的副作用：它會計算包級變量的初始化表達式和執行導入包的init初始化函數（§2.6.2）。這時候我們需要抑製“unused import”編譯錯誤，我們可以用下劃線 `_` 來重命名導入的包。像往常一樣，下劃線 `_` 爲空白標識符，併不能被訪問。

```
import
 _ "image/png"
 // register PNG decoder
```

這個被稱爲包的匿名導入。它通常是用來實現一個編譯時機製，然後通過在main主程序入口選擇性地導入附加的包。首先，讓我們看看如何使用該特性，然後再看看它是如何工作的。

標準庫的image圖像包包含了一個 `Decode` 函數，用於從 `io.Reader` 接口讀取數據併解碼圖像，它調用底層註冊的圖像解碼器來完成任務，然後返迴image.Image類型的圖像。使用 `image.Decode` 很容易編寫一個圖像格式的轉換工具，讀取一種格式的圖像，然後編碼爲另一種圖像格式：

```
gopl.io/ch10/jpeg
// The jpeg command reads a PNG image from the standard input

// and writes it as a JPEG image to the standard output.

package
 main

import
 (
    "fmt"

    "image"

    "image/jpeg"

    _ "image/png"
 // register PNG decoder

    "io"

    "os"

)

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

```
        os.Exit(1
)
    }
}

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

如果我們將 `gopl.io/ch3/mandelbrot` （§3.3）的輸出導入到這個程序的標準輸入，它將解碼輸入的PNG格式圖像，然後轉換爲JPEG格式的圖像輸出（圖3.3）。

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

要註意image/png包的匿名導入語句。如果沒有這一行語句，程序依然可以編譯和運行，但是它將不能正確識別和解碼PNG格式的圖像：

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

下面的代碼演示了它的工作機製。標準庫還提供了GIF、PNG和JPEG等格式圖像的解碼器，用戶也可以提供自己的解碼器，但是爲了保持程序體積較小，很多解碼器併沒有被全部包含，除非是明確需要支持的格式。image.Decode函數在解碼時會依次查詢支持的格式列表。每個格式驅動列表的每個入口指定了四件事情：格式的名稱；一個用於描述這種圖像數據開頭部分模式的字符串，用於解碼器檢測識別；一個Decode函數用於完成解碼圖像工作；一個DecodeConfig函數用於解碼圖像的大小和顏色空間的信息。每個驅動入口是通過調用image.RegisterFormat函數註冊，一般是在每個格式包的init初始化函數中調用，例如image/png包是這樣註冊的：

```
package
 png // image/png


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

func
 init() {
    const
 pngHeader = "\x89PNG\r\n\x1a\n"

    image.RegisterFormat("png"
, pngHeader, Decode, DecodeConfig)
}
```

最終的效果是，主程序隻需要匿名導入特定圖像驅動包就可以用image.Decode解碼對應格式的圖像了。

數據庫包database/sql也是采用了類似的技術，讓用戶可以根據自己需要選擇導入必要的數據庫驅動。例如：

```
import
 (
    "database/mysql"

    _ "github.com/lib/pq"
              // enable support for Postgres

    _ "github.com/go-sql-driver/mysql"
 // enable support for MySQL

)

db, err = sql.Open("postgres"
, dbname) // OK

db, err = sql.Open("mysql"
, dbname)    // OK

db, err = sql.Open("sqlite3"
, dbname)  // returns error: unknown driver "sqlite3"
```

練習 10.1： 擴展jpeg程序，以支持任意圖像格式之間的相互轉換，使用image.Decode檢測支持的格式類型，然後通過flag命令行標誌參數選擇輸出的格式。

練習 10.2： 設計一個通用的壓縮文件讀取框架，用來讀取ZIP（archive/zip）和POSIX tar（archive/tar）格式壓縮的文檔。使用類似上面的註冊技術來擴展支持不同的壓縮格式，然後根據需要通過匿名導入選擇導入要支持的壓縮格式的驅動包。

10.6. 包和命名

在本節中，我們將提供一些關於Go語言獨特的包和成員命名的約定。

當創建一個包，一般要用短小的包名，但也不能太短導致難以理解。標準庫中最常用的包有bufio、bytes、flag、fmt、http、io、json、os、sort、sync和time等包。

它們的名字都簡潔明了。例如，不要將一個類似imageutil或ioutilis的通用包命名爲util，雖然它看起來很短小。要盡量避免包名使用可能被經常用於局部變量的名字，這樣可能導致用戶重命名導入包，例如前面看到的path包。

包名一般采用單數的形式。標準庫的bytes、errors和strings使用了複數形式，這是爲了避免和預定義的類型衝突，同樣還有go/types是爲了避免和type關鍵字衝突。

要避免包名有其它的含義。例如，2.5節中我們的溫度轉換包最初使用了temp包名，雖然併沒有持續多久。但這是一個糟糕的嚐試，因爲temp幾乎是臨時變量的同義詞。然後我們有一段時間使用了temperature作爲包名，雖然名字併沒有表達包的眞實用途。最後我們改成了和strconv標準包類似的tempconv包名，這個名字比之前的就好多了。

現在讓我們看看如何命名包的成員。由於是通過包的導入名字引入包里面的成員，例如fmt.Println，同時包含了包名和成員名信息。因此，我們一般併不需要關註Println的具體內容，因爲fmt包名已經包含了這個信息。當設計一個包的時候，需要考慮包名和成員名兩個部分如何很好地配合。下面有一些例子：

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

我們可以看到一些常用的命名模式。strings包提供了和字符串相關的諸多操作：

```
package
 strings

func
 Index(needle, haystack string
) int



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

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

字符串string本身併沒有出現在每個成員名字中。因爲用戶會這樣引用這些成員strings.Index、strings.Replacer等。

其它一些包，可能隻描述了單一的數據類型，例如html/template和math/rand等，隻暴露一個主要的數據結構和與它相關的方法，還有一個以New命名的函數用於創建實例。

```
package
 rand // "math/rand"


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

這可能導致一些名字重複，例如template.Template或rand.Rand，這就是爲什麽這些種類的包名往往特別短的原因之一。

在另一個極端，還有像net/http包那樣含有非常多的名字和種類不多的數據類型，因爲它們都是要執行一個複雜的複合任務。盡管有將近二十種類型和更多的函數，但是包中最重要的成員名字卻是簡單明了的：Get、Post、Handle、Error、Client、Server等。

10.7. 工具

本章剩下的部分將討論Go語言工具箱的具體功能，包括如何下載、格式化、構建、測試和安裝Go語言編寫的程序。

Go語言的工具箱集合了一繫列的功能的命令集。它可以看作是一個包管理器（類似於Linux中的apt和rpm工具），用於包的查詢、計算的包依賴關繫、從遠程版本控製繫統和下載它們等任務。它也是一個構建繫統，計算文件的依賴關繫，然後調用編譯器、滙編器和連接器構建程序，雖然它故意被設計成沒有標準的make命令那麼複雜。它也是一個單元測試和基準測試的驅動程序，我們將在第11章討論測試話題。

Go語言工具箱的命令有着類似“瑞士軍刀”的風格，帶着一打子的子命令，有一些我們經常用到，例如get、run、build和fmt等。你可以運行go或go help命令查看內置的幫助文檔，爲了查詢方便，我們列出了最常用的命令：

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

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

爲了達到零配置的設計目標，Go語言的工具箱很多地方都依賴各種約定。例如，根據給定的源文件的名稱，Go語言的工具可以找到源文件對應的包，因爲每個目録隻包含了單一的包，併且到的導入路徑和工作區的目録結構是對應的。給定一個包的導入路徑，Go語言的工具可以找到對應的目録中沒個實體對應的源文件。它還可以根據導入路徑找到存儲代碼倉庫的遠程服務器的URL。

10.7.1. 工作區結構

對於大多數的Go語言用戶，隻需要配置一個名叫GOPATH的環境變量，用來指定當前工作目録即可。當需要切換到不同工作區的時候，隻要更新GOPATH就可以了。例如，我們在編寫本書時將GOPATH設置爲 `$HOME/gobook` ：

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

當你用前面介紹的命令下載本書全部的例子源碼之後，你的當前工作區的目録結構應該是這樣的：

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

GOPATH對應的工作區目録有三個子目録。其中src子目録用於存儲源代碼。每個包被保存在與$GOPATH/src的相對路徑爲包導入路徑的子目録中，例如gopl.io/ch1/helloworld相對應的路徑目録。我們看到，一個GOPATH工作區的src目録中可能有多個獨立的版本控製繫統，例如gopl.io和golang.org分別對應不同的Git倉庫。其中pkg子目録用於保存編譯後的包的目標文件，bin子目録用於保存編譯後的可執行程序，例如helloworld可執行程序。

第二個環境變量GOROOT用來指定Go的安裝目録，還有它自帶的標準庫包的位置。GOROOT的目録結構和GOPATH類似，因此存放fmt包的源代碼對應目録應該爲$GOROOT/src/fmt。用戶一般不需要設置GOROOT，默認情況下Go語言安裝工具會將其設置爲安裝的目録路徑。

其中 `go env` 命令用於查看Go語音工具涉及的所有環境變量的值，包括未設置環境變量的默認值。GOOS環境變量用於指定目標操作繫統（例如android、linux、darwin或windows），GOARCH環境變量用於指定處理器的類型，例如amd64、386或arm等。雖然GOPATH環境變量是唯一必需要設置的，但是其它環境變量也會偶爾用到。

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

10.7.2. 下載包

使用Go語言工具箱的go命令，不僅可以根據包導入路徑找到本地工作區的包，甚至可以從互聯網上找到和更新包。

使用命令 `go get` 可以下載一個單一的包或者用 `...` 下載整個子目録里面的每個包。Go語言工具箱的go命令同時計算併下載所依賴的每個包，這也是前一個例子中golang.org/x/net/html自動出現在本地工作區目録的原因。

一旦 `go get` 命令下載了包，然後就是安裝包或包對應的可執行的程序。我們將在下一節再關註它的細節，現在隻是展示整個下載過程是如何的簡單。第一個命令是獲取golint工具，它用於檢測Go源代碼的編程風格是否有問題。第二個命令是用golint命令對2.6.2節的gopl.io/ch2/popcount包代碼進行編碼風格檢查。它友好地報告了忘記了包的文檔：

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

`go get` 命令支持當前流行的託管網站GitHub、Bitbucket和Launchpad，可以直接向它們的版本控製繫統請求代碼。對於其它的網站，你可能需要指定版本控製繫統的具體路徑和協議，例如 Git或Mercurial。運行 `go help importpath` 獲取相關的信息。

`go get` 命令獲取的代碼是眞實的本地存儲倉庫，而不僅僅隻是複製源文件，因此你依然可以使用版本管理工具比較本地代碼的變更或者切換到其它的版本。例如golang.org/x/net包目録對應一個Git倉庫：

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

需要註意的是導入路徑含有的網站域名和本地Git倉庫對應遠程服務地址併不相同，眞實的Git地址是go.googlesource.com。這其實是Go語言工具的一個特性，可以讓包用一個自定義的導入路徑，但是眞實的代碼卻是由更通用的服務提供，例如googlesource.com或github.com。因爲頁面 https://golang.org/x/net/html 包含了如下的元數據，它告訴Go語言的工具當前包眞實的Git倉庫託管地址：

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

如果指定 `-u` 命令行標誌參數， `go get` 命令將確保所有的包和依賴的包的版本都是最新的，然後重新編譯和安裝它們。如果不包含該標誌參數的話，而且如果包已經在本地存在，那麼代碼那麼將不會被自動更新。

`go get -u` 命令隻是簡單地保證每個包是最新版本，如果是第一次下載包則是比較很方便的；但是對於發布程序則可能是不合適的，因爲本地程序可能需要對依賴的包做精確的版本依賴管理。通常的解決方案是使用vendor的目録用於存儲依賴包的固定版本的源代碼，對本地依賴的包的版本更新也是謹慎和持續可控的。在Go1.5之前，一般需要脩改包的導入路徑，所以複製後golang.org/x/net/html導入路徑可能會變爲gopl.io/vendor/golang.org/x/net/html。最新的Go語言命令已經支持vendor特性，但限於篇幅這里併不討論vendor的具體細節。不過可以通過 `go help gopath` 命令查看Vendor的幫助文檔。

練習 10.3: 從 http://gopl.io/ch1/helloworld?go-get=1 獲取內容，查看本書的代碼的眞實託管的網址（ `go get` 請求HTML頁面時包含了 `go-get` 參數，以區別普通的瀏覽器請求）。

10.7.3. 構建包

`go build` 命令編譯命令行參數指定的每個包。如果包是一個庫，則忽略輸出結果；這可以用於檢測包的可以正確編譯的。如果包的名字是main， `go build` 將調用連接器在當前目録創建一個可執行程序；以導入路徑的最後一段作爲可執行程序的名字。

因爲每個目録隻包含一個包，因此每個對應可執行程序或者叫Unix術語中的命令的包，會要求放到一個獨立的目録中。這些目録有時候會放在名叫cmd目録的子目録下面，例如用於提供Go文檔服務的golang.org/x/tools/cmd/godoc命令就是放在cmd子目録（§10.7.4）。

每個包可以由它們的導入路徑指定，就像前面看到的那樣，或者用一個相對目録的路徑知指定，相對路徑必鬚以 `.` 或 `..` 開頭。如果沒有指定參數，那麽默認指定爲當前目録對應的包。 下面的命令用於構建同一個包, 雖然它們的寫法各不相同:

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

或者：

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

或者：

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

但不能這樣：

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

也可以指定包的源文件列表，這一般這隻用於構建一些小程序或做一些臨時性的實驗。如果是main包，將會以第一個Go源文件的基礎文件名作爲最終的可執行程序的名字。

```
$ cat quoteargs.go
package main

import (
    "fmt"
    "os"
)

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

特別是對於這類一次性運行的程序，我們希望盡快的構建併運行它。 `go run` 命令實際上是結合了構建和運行的兩個步驟：

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

第一行的參數列表中，第一個不是以 `.go` 結尾的將作爲可執行程序的參數運行。

默認情況下， `go build` 命令構建指定的包和它依賴的包，然後丟棄除了最後的可執行文件之外所有的中間編譯結果。依賴分析和編譯過程雖然都是很快的，但是隨着項目增加到幾十個包和成韆上萬行代碼，依賴關繫分析和編譯時間的消耗將變的可觀，有時候可能需要幾秒種，即使這些依賴項沒有改變。

`go install` 命令和 `go build` 命令很相似，但是它會保存每個包的編譯成果，而不是將它們都丟棄。被編譯的包會被保存到$GOPATH/pkg目録下，目録路徑和 src目録路徑對應，可執行程序被保存到$GOPATH/bin目録。（很多用戶會將$GOPATH/bin添加到可執行程序的蒐索列表中。）還有， `go install` 命令和 `go build` 命令都不會重新編譯沒有發生變化的包，這可以使後續構建更快捷。爲了方便編譯依賴的包， `go build -i` 命令將安裝每個目標所依賴的包。

因爲編譯對應不同的操作繫統平台和CPU架構， `go install` 命令會將編譯結果安裝到GOOS和GOARCH對應的目録。例如，在Mac繫統，golang.org/x/net/html包將被安裝到$GOPATH/pkg/darwin_amd64目録下的golang.org/x/net/html.a文件。

針對不同操作繫統或CPU的交叉構建也是很簡單的。隻需要設置好目標對應的GOOS和GOARCH，然後運行構建命令卽可。下面交叉編譯的程序將輸出它在編譯時操作繫統和CPU類型：

```
gopl.io/ch10/cross

func
 main() {
    fmt.Println(runtime.GOOS, runtime.GOARCH)
}
```

下面以64位和32位環境分別執行程序：

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

有些包可能需要針對不同平台和處理器類型使用不同版本的代碼文件，以便於處理底層的可移植性問題或提供爲一些特定代碼提供優化。如果一個文件名包含了一個操作繫統或處理器類型名字，例如net_linux.go或asm_amd64.s，Go語言的構建工具將隻在對應的平台編譯這些文件。還有一個特別的構建註釋註釋可以提供更多的構建過程控製。例如，文件中可能包含下面的註釋：

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

在包聲明和包註釋的前面，該構建註釋參數告訴 `go build` 隻在編譯程序對應的目標操作繫統是Linux或Mac OS X時才編譯這個文件。下面的構建註釋則表示不編譯這個文件：

```
// +build ignore
```

更多細節，可以參考go/build包的構建約束部分的文檔。

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

10.7.4. 包文檔

Go語言的編碼風格鼓勵爲每個包提供良好的文檔。包中每個導出的成員和包聲明前都應該包含目的和用法說明的註釋。

Go語言中包文檔註釋一般是完整的句子，第一行是包的摘要說明，註釋後僅跟着包聲明語句。註釋中函數的參數或其它的標識符併不需要額外的引號或其它標記註明。例如，下面是fmt.Fprintf的文檔註釋。

```
// Fprintf formats according to a format specifier and writes to w.

// It returns the number of bytes written and any write error encountered.

func
 Fprintf(w io.Writer, format string
, a ...interface
{}) (int
, error)
```

Fprintf函數格式化的細節在fmt包文檔中描述。如果註釋後僅跟着包聲明語句，那註釋對應整個包的文檔。包文檔對應的註釋隻能有一個（譯註：其實可以有多個，它們會組合成一個包文檔註釋），包註釋可以出現在任何一個源文件中。如果包的註釋內容比較長，一般會放到一個獨立的源文件中；fmt包註釋就有300行之多。這個專門用於保存包文檔的源文件通常叫doc.go。

好的文檔併不需要面面俱到，文檔本身應該是簡潔但可不忽略的。事實上，Go語言的風格更喜歡簡潔的文檔，併且文檔也是需要像代碼一樣維護的。對於一組聲明語句，可以用一個精鍊的句子描述，如果是顯而易見的功能則併不需要註釋。

在本書中，隻要空間允許，我們之前很多包聲明都包含了註釋文檔，但你可以從標準庫中發現很多更好的例子。有兩個工具可以幫到你。

首先是 `go doc` 命令，該命令打印包的聲明和每個成員的文檔註釋，下面是整個包的文檔：

```
$ go doc time
package time // import "time"

Package time provides functionality for measuring and displaying time.

const Nanosecond Duration = 1 ...
func After(d Duration) <-chan Time
func Sleep(d Duration)
func Since(t Time) Duration
func Now() Time
type Duration int64
type Time struct { ... }
...many more...
```

或者是某個具體包成員的註釋文檔：

```
$ go doc time.Since
func Since(t Time) Duration

    Since returns the time elapsed since t.
    It is shorthand for time.Now().Sub(t).
```

或者是某個具體包的一個方法的註釋文檔：

```
$ go doc time.Duration.Seconds
func (d Duration) Seconds() float64

    Seconds returns the duration as a floating-point number of seconds.
```

該命令併不需要輸入完整的包導入路徑或正確的大小寫。下面的命令將打印encoding/json包的 `(*json.Decoder).Decode` 方法的文檔：

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

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

第二個工具，名字也叫godoc，它提供可以相互交叉引用的HTML頁面，但是包含和 `go doc` 命令相同以及更多的信息。10.1節演示了time包的文檔，11.6節將看到godoc演示可以交互的示例程序。godoc的在線服務 https://godoc.org，包含了成韆上萬的開源包的檢索工具。

你也可以在自己的工作區目録運行godoc服務。運行下面的命令，然後在瀏覽器查看 http://localhost:8000/pkg 頁面：

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

其中 `-analysis=type` 和 `-analysis=pointer` 命令行標誌參數用於打開文檔和代碼中關於靜態分析的結果。

10.7.5. 內部包

在Go語音程序中，包的封裝機製是一個重要的特性。沒有導出的標識符隻在同一個包內部可以訪問，而導出的標識符則是面向全宇宙都是可見的。

有時候，一個中間的狀態可能也是有用的，對於一小部分信任的包是可見的，但併不是對所有調用者都可見。例如，當我們計劃將一個大的包拆分爲很多小的更容易維護的子包，但是我們併不想將內部的子包結構也完全暴露出去。同時，我們可能還希望在內部子包之間共享一些通用的處理包，或者我們隻是想實驗一個新包的還併不穩定的接口，暫時隻暴露給一些受限製的用戶使用。

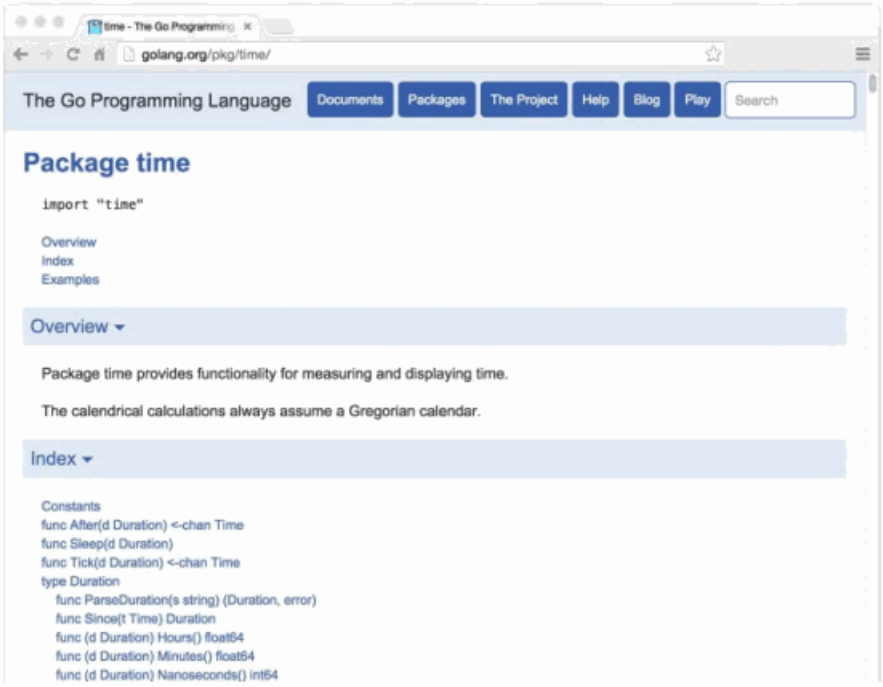


Figure 10.1. The `time` package in godoc.

爲了滿足這些需求，Go語言的構建工具對包含internal名字的路徑段的包導入路徑做了特殊處理。這種包叫internal包，一個internal包隻能被和internal目録有同一個父目録的包所導入。例如，net/http/internal/chunked內部包隻能被net/http/httputil或net/http包導入，但是不能被net/url包導入。不過net/url包卻可以導入net/http/httputil包。

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

10.7.6. 查詢包

`go list` 命令可以查詢可用包的信息。其最簡單的形式，可以測試包是否在工作區併打印它的導入路徑：

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

`go list` 命令的參數還可以用 `"..."` 表示匹配任意的包的導入路徑。我們可以用它來列表工作區中的所有包：

```
$ go list ...
archive/tar
archive/zip
bufio
bytes
cmd/addr2line
cmd/api
...many more...
```

或者是特定子目録下的所有包：

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

或者是和某個主題相關的所有包:

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

`go list` 命令還可以獲取每個包完整的元信息，而不僅僅隻是導入路徑，這些元信息可以以不同格式提供給用戶。其中 `-json` 命令行參數表示用JSON格式打印每個包的元信息。

```
$ go list -json hash
{
    "Dir": "/home/gopher/go/src/hash",
    "ImportPath": "hash",
    "Name": "hash",
    "Doc": "Package hash provides interfaces for hash functions.",
    "Target": "/home/gopher/go/pkg/darwin_amd64/hash.a",
    "Goroot": true,
    "Standard": true,
    "Root": "/home/gopher/go",
    "GoFiles": [
            "hash.go"
    ],
    "Imports": [
        "io"
    ],
    "Deps": [
        "errors",
        "io",
        "runtime",
        "sync",
        "sync/atomic",
        "unsafe"
    ]
}
```

命令行參數 `-f` 則允許用戶使用text/template包（§4.6）的模闆語言定義輸出文本的格式。下面的命令將打印strconv包的依賴的包，然後用join模闆函數將結果鏈接爲一行，連接時每個結果之間用一個空格分隔：

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

譯註：上面的命令在Windows的命令行運行會遇到 `template: main:1: unclosed action` 的錯誤。產生這個錯誤的原因是因爲命令行對命令中的 `" "` 參數進行了轉義處理。可以按照下面的方法解決轉義字符串的問題：

```
$ go list -f "{{join .Deps \" \"}}" strconv
```

下面的命令打印compress子目録下所有包的依賴包列表：

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

譯註：Windows下有同樣有問題，要避免轉義字符串的幹擾：

```
$ go list -f "{{.ImportPath}} -> {{join .Imports \" \"}}" compress/...
```

`go list` 命令對於一次性的交互式查詢或自動化構建或測試腳本都很有幫助。我們將在11.2.4節中再次使用它。每個子命令的更多信息，包括可設置的字段和意義，可以用 `go help list` 命令查看。

在本章，我們解釋了Go語言工具中除了測試命令之外的所有重要的子命令。在下一章，我們將看到如何用 `go test` 命令去運行Go語言程序中的測試代碼。

練習 10.4： 創建一個工具，根據命令行指定的參數，報告工作區所有依賴指定包的其它包集合。提示：你需要運行 `go list` 命令兩次，一次用於初始化包，一次用於所有包。你可能需要用encoding/json（§4.5）包來分析輸出的JSON格式的信息。

第十一章 測試

Maurice Wilkes, 第一個存儲程序計算機 EDSAC 的設計者, 1949年在他的實驗室爬樓梯時有一個頓悟. 在《計算機先驅迴憶録》(Memoirs of a Computer Pioneer)里, 他迴憶到: "忽然間有一種醍醐灌頂的感覺, 我整個後半生的美好時光都將在尋找程序BUG中度過了.". 肯定從那之後的每一個存儲程序的碼農都可以同情 Wilkes 的想法, 雖然也許不是沒有人睏惑於他對軟件開發的難度的天眞看法.

現在的程序已經遠比 Wilkes 時代的更大也更複雜, 也有許多技術可以讓軟件的複雜性可得到控製. 其中有兩種技術在實踐中證明是比較有效的. 第一種是代碼在被正式部署前需要進行代碼評審. 第二種是測試, 是本章的討論主題.

我們説測試的時候一般是指自動化測試, 也就是寫一些小的程序用來檢測被測試代碼(産品代碼)的行爲和預期的一樣, 這些通常都是精心挑選的執行某些特定的功能或者是通過隨機性的輸入要驗證邊界的處理.

軟件測試是一個鉅大的領域. 測試的任務一般占據了一些程序員的部分時間和另一些程序員的全部時間. 和軟件測試技術相關的圖書或博客文章有成韆上萬之多. 每一種主流的編程語言, 都有一打的用於測試的軟件包, 也有大量的測試相關的理論, 每種都吸引了大量技術先驅和追隨者. 這些都足以説服那些想要編寫有效測試的程序員重新學習一套全新的技能.

Go語言的測試技術是相對低級的. 它依賴一個 'go test' 測試命令, 和一組按照約定方式編寫的測試函數, 測試命令可以運行測試函數. 編寫相對輕量級的純測試代碼是有效的, 而且它很容易延伸到基準測試和示例文檔.

在實踐中, 編寫測試代碼和編寫程序本身併沒有多大區别. 我們編寫的每一個函數也是針對每個具體的任務. 我們必鬚小心處理邊界條件, 思考合適的數據結構, 推斷合適的輸入應該産生什麽樣的結果輸出. 編程測試代碼和編寫普通的Go代碼過程是類似的; 它併不需要學習新的符號, 規則和工具.

11.1. go test

`go test` 是一個按照一定的約定和組織的測試代碼的驅動程序. 在包目録內, 以 `_test.go` 爲後綴名的源文件併不是 `go build` 構建包的以部分, 它們是 `go test` 測試的一部分.

早 `*_test.go` 文件中, 有三種類型的函數: 測試函數, 基準測試函數, 例子函數. 一個測試函數是以 Test 爲函數名前綴的函數, 用於測試程序的一些邏輯行爲是否正確; `go test` 會調用這些測試函數併報告測試結果是 PASS 或 FAIL. 基準測試函數是以 Benchmark爲函數名前綴的函數, 用於衡量一些函數的性能; `go test` 會多次運行基準函數以計算一個平均的執行時間. 例子函數是以Example爲函數名前綴的函數, 提供一個由機器檢測正確性的例子文檔. 我們將在 11.2 節 討論測試函數的細節, 在 11.4 節討論基準測試函數的細節, 在 11.6 討論例子函數的細節.

`go test` 命令會遍歷所有的 `*_test.go` 文件中上述函數, 然後生成一個臨時的main包調用相應的測試函數, 然後構建併運行, 報告測試結果, 最後清理臨時文件.

11.2. 測試函數

每個測試函數必鬚導入 testing 包. 測試函數有如下的籤名:

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

}
```

測試函數的名字必鬚以Test開頭, 可選的後綴名必鬚以大寫字母開頭:

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

其中 t 參數用於報告測試失敗和附件的日誌信息. 讓我們頂一個一個實例包 gopl.io/ch11/word1, 隻有一個函數 IsPalindrome 用於檢查一個字符串是否從前向後和從後向前讀都一樣. (這個實現對於一個字符串是否是迴文字符串前後重複測試了兩次; 我們稍後會再討論這個問題.)

```
gopl.io/ch11/word1
// Package word provides utilities for word games.

package
 word

// IsPalindrome reports whether s reads the same forward and backward.

// (Our first attempt.)

func
 IsPalindrome(s string
) bool
 {
    for
 i := range
 s {
        if
 s[i] != s[len
(s)-1
-i] {
            return
 false

        }
    }
    return
 true

}
```

在相同的目録下, word_test.go 文件包含了 TestPalindrome 和 TestNonPalindrome 兩個測試函數. 每一個都是測試 IsPalindrome 是否給出正確的結果, 併使用 t.Error 報告失敗:

```
package
 word

import
 "testing"


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

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

`go test` (或 `go build`) 命令 如果沒有參數指定包那麼將默認采用當前目録對應的包. 我們可以用下面的命令構建和運行測試.

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

還比較滿意, 我們運行了這個程序, 不過沒有提前退出是因爲還沒有遇到BUG報告. 一個法国名爲 Noelle Eve Elleon 的用戶抱怨 IsPalindrome 函數不能識別 ‘‘été.’’. 另外一個來自美国中部用戶的抱怨是不能識別 ‘‘A man, a plan, a canal: Panama.’’. 執行特殊和小的BUG報告爲我們提供了新的更自然的測試用例.

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

func
 TestCanalPalindrome(t *testing.T) {
    input := "A man, a plan, a canal: Panama"

    if
 !IsPalindrome(input) {
        t.Errorf(`IsPalindrome(%q) = false`
, input)
    }
}
```

爲了避免兩次輸入較長的字符串, 我們使用了提供了有類似 Printf格式化功能的 Errorf函數來滙報錯誤結果.

當添加了這兩個測試用例之後, `go test` 返迴了測試失敗的信息.

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

先編寫測試用例併觀察到測試用例觸發了和用戶報告的錯誤相同的描述是一個好的測試習慣. 隻有這樣, 我們才能定位我們要眞正解決的問題.

先寫測試用例的另好處是, 運行測試通常會比手工描述報告的處理更快, 這讓我們可以進行快速地迭代. 如果測試集有很多運行緩慢的測試, 我們可以通過隻選擇運行某些特定的測試來加快測試速度.

參數 `-v` 用於打印每個測試函數的名字和運行時間:

```
$ go test -v
=== RUN TestPalindrome
--- PASS: TestPalindrome (0.00s)
=== RUN TestNonPalindrome
--- PASS: TestNonPalindrome (0.00s)
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1  0.017s
```

參數 `-run` 是一個正則表達式, 隻有測試函數名被它正確匹配的測試函數才會被 `go test` 運行:

```
$ go test -v -run="French|Canal"
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1  0.014s
```

當然, 一旦我們已經脩複了失敗的測試用例, 在我們提交代碼更新之前, 我們應該以不帶參數的 `go test` 命令運行全部的測試用例, 以確保更新沒有引入新的問題.

我們現在的任務就是脩複這些錯誤. 簡要分析後發現第一個BUG的原因是我們采用了 byte 而不是 rune 序列, 所以像 "été" 中的 é 等非 ASCII 字符不能正確處理. 第二個BUG是因爲沒有忽略空格和字母的大小寫導致的.

針對上述兩個BUG, 我們仔細重寫了函數:

```
gopl.io/ch11/word2
// Package word provides utilities for word games.

package
 word

import
 "unicode"


// IsPalindrome reports whether s reads the same forward and backward.

// Letter case is ignored, as are non-letters.

func
 IsPalindrome(s string
) bool
 {
    var
 letters []rune

    for
 _, r := range
 s {
        if
 unicode.IsLetter(r) {
            letters = append
(letters, unicode.ToLower(r))
        }
    }
    for
 i := range
 letters {
        if
 letters[i] != letters[len
(letters)-1
-i] {
            return
 false

        }
    }
    return
 true

}
```

同時我們也將之前的所有測試數據合併到了一個測試中的表格中.

```
func
```

```
func
 TestIsPalindrome(t *testing.T) {
    var
 tests = []struct
 {
        input string

        want     bool

    }{
        {""
, true
},
        {"a"
, true
},
        {"aa"
, true
},
        {"ab"
, false
},
        {"kayak"
, true
},
        {"detartrated"
, true
},
        {"A man, a plan, a canal: Panama"
, true
},
        {"Evil I did dwell; lewd did I live."
, true
},
        {"Able was I ere I saw Elba"
, true
},
        {"été"
, true
},
        {"Et se resservir, ivresse reste."
, true
},
        {"palindrome"
, false
}, // non-palindrome

        {"desserts"
, false
},   // semi-palindrome

    }
    for
```

```
 _, test := range
 tests {
        if
 got := IsPalindrome(test.input); got != test.want {
            t.Errorf("IsPalindrome(%q) = %v"
, test.input, got)
        }
    }
}
```

我們的新測試阿都通過了:

```
$ go test gopl.io/ch11/word2
ok      gopl.io/ch11/word2      0.015s
```

這種表格驅動的測試在Go中很常見的. 我們很容易想表格添加新的測試數據, 併且後面的測試邏輯也沒有冗餘, 這樣我們可以更好地完善錯誤信息.

失敗的測試的輸出併不包括調用 t.Errorf 時刻的堆棧調用信息. 不像其他語言或測試框架的 assert 斷言, t.Errorf 調用也沒有引起 panic 或停止測試的執行. 卽使表格中前面的數據導致了測試的失敗, 表格後面的測試數據依然會運行測試, 因此在一個測試中我們可能了解多個失敗的信息.

如果我們眞的需要停止測試, 或許是因爲初始化失敗或可能是早先的錯誤導致了後續錯誤等原因, 我們可以使用 t.Fatal 或 t.Fatalf 停止測試. 它們必鬚在和測試函數同一個 goroutine 內調用.

測試失敗的信息一般的形式是 "f(x) = y, want z", f(x) 解釋了失敗的操作和對應的輸出, y 是實際的運行結果, z 是期望的正確的結果. 就像前面檢查迴文字符串的例子, 實際的函數用於 f(x) 部分. 如果顯示 x 是表格驅動型測試中比較重要的部分, 因爲同一個斷言可能對應不同的表格項執行多次. 要避免無用和冗餘的信息. 在測試類似 IsPalindrome 返迴布爾類型的函數時, 可以忽略併沒有額外信息的 z 部分. 如果 x, y 或 z 是 y 的長度, 輸出一個相關部分的簡明總結卽可. 測試的作者應該要努力幫助程序員診斷失敗的測試.

練習 11.1: 爲 4.3節 中的 charcount 程序編寫測試.

練習 11.2: 爲 (§6.5)的 IntSet 編寫一組測試, 用於檢查每個操作後的行爲和基於內置 map 的集合等價 , 後面 練習11.7 將會用到.

11.2.1. 隨機測試

表格驅動的測試便於構造基於精心挑選的測試數據的測試用例. 另一種測試思路是隨機測試, 也就是通過構造更廣泛的隨機輸入來測試探索函數的行爲.

那麼對於一個隨機的輸入, 我們如何能知道希望的輸出結果呢? 這里有兩種策略. 第一個是編寫另一個函數, 使用簡單和清晰的算法, 雖然效率較低但是行爲和要測試的函數一致, 然後針對相同的隨機輸入檢查兩者的輸出結果. 第二種是生成的隨機輸入的數據遵循特定的模式, 這樣我們就可以知道期望的輸出的模式.

下面的例子使用的是第二種方法: randomPalindrome 函數用於隨機生成迴文字符串.

```
import
 "math/rand"


// randomPalindrome returns a palindrome whose length and contents
```

```
// are derived from the pseudo-random number generator rng.

func
 randomPalindrome(rng *rand.Rand) string
 {
    n := rng.Intn(25
) // random length up to 24

    runes := make
([]rune
, n)
    for
 i := 0
; i < (n+1
)/2
; i++ {
        r := rune
(rng.Intn(0x1000
)) // random rune up to '\u0999'

        runes[i] = r
        runes[n-1
-i] = r
    }
    return
 string
(runes)
}

func
 TestRandomPalindromes(t *testing.T) {
    // Initialize a pseudo-random number generator.

    seed := time.Now().UTC().UnixNano()
    t.Logf("Random seed: %d"
, seed)
    rng := rand.New(rand.NewSource(seed))



    for
 i := 0
; i < 1000
; i++ {
        p := randomPalindrome(rng)
        if
 !IsPalindrome(p) {
            t.Errorf("IsPalindrome(%q) = false"
, p)
        }
    }
}
```

雖然隨機測試有不確定因素, 但是它也是至關重要的, 我們可以從失敗測試的日誌獲取足夠的信息. 在我們的例子中, 輸入IsPalindrome 的 p 參數將告訴我們眞實的數據, 但是對於函數將接受更複雜的輸入, 不需要保存所有的輸入, 隻要日誌中簡單地記録隨機數種子即可(像上面的方式). 有了這些隨機數初始化種子, 我們可以很容易脩改測試代碼以重現失敗的隨機測試.

通過使用當前時間作爲隨機種子, 在整個過程中的每次運行測試命令時都將探索新的隨機數據. 如果你使用的是定期運行的自動化測試集成繫統, 隨機測試將特別有價值.

練習 11.3: TestRandomPalindromes 隻測試了迴文字符串. 編寫新的隨機測試生成器, 用於測試隨機生成的非迴文字符串.

練習 11.4: 脩改 randomPalindrome 函數, 以探索 IsPalindrome 對標點和空格的處理.

11.2.2. 測試一個命令

對於測試包 `go test` 是一個的有用的工具, 但是稍加努力我們也可以用它來測試可執行程序. 如果一個包的名字是 main, 那麽在構建時會生成一個可執行程序, 不過 main 包可以作爲一個包被測試器代碼導入.

讓我們爲 2.3.2節 的 echo 程序編寫一個測試. 我們先將程序拆分爲兩個函數: echo 函數完成眞正的工作, main 函數用於處理命令行輸入參數和echo可能返迴的錯誤.

```
gopl.io/ch11/echo
// Echo prints its command-line arguments.

package
 main

import
 (
    "flag"

    "fmt"

    "io"

    "os"

    "strings"

)

var
 (
    n = flag.Bool("n"
, false
, "omit trailing newline"
)
    s = flag.String("s"
, " "
, "separator"
)
)

var
 out io.Writer = os.Stdout // modified during testing
```

```
func
 main() {
    flag.Parse()
    if
 err := echo(!*n, *s, flag.Args()); err != nil
 {
        fmt.Fprintf(os.Stderr, "echo: %v\n"
, err)
        os.Exit(1
)
    }
}

func
 echo(newline bool
, sep string
, args []string
) error {
    fmt.Fprint(out, strings.Join(args, sep))
    if
 newline {
        fmt.Fprintln(out)
    }
    return
 nil

}
```

在測試中嗎我們可以用各種參數和標標誌調用 echo 函數, 然後檢測它的輸出是否正確, 我們通過增加參數來減少 echo 函數對全局變量的依賴. 我們還增加了一個全局名爲 out 的變量來替代直接使用 os.Stdout, 這樣測試代碼可以根據需要將 out 脩改爲不同的對象以便於檢查. 下面就是 echo_test.go 文件中的測試代碼:

```
package
 main

import
 (
    "bytes"

    "fmt"

    "testing"

)

func
 TestEcho(t *testing.T) {
    var
 tests = []struct
 {
```

```
 i
        newline bool

        sep     string

        args    []string

        want    string

    }{
        {true
, ""
, []string
{}, "\n"
},
        {false
, ""
, []string
{}, ""
},
        {true
, "\t"
, []string
{"one"
, "two"
, "three"
}, "one\ttwo\tthree\n"
},
        {true
, ","
, []string
{"a"
, "b"
, "c"
}, "a,b,c\n"
},
        {false
, ":"
, []string
{"1"
, "2"
, "3"
}, "1:2:3"
},
    }
    for
 _, test := range
 tests {
        descr := fmt.Sprintf("echo(%v, %q, %q)"
,
            test.newline, test.sep, test.args)

        out = new
```

```
(bytes.Buffer) // captured output

        if
 err := echo(test.newline, test.sep, test.args); err != nil
 {
            t.Errorf("%s failed: %v"
, descr, err)
            continue

        }
        got := out.(*bytes.Buffer).String()
        if
 got != test.want {
            t.Errorf("%s = %q, want %q"
, descr, got, test.want)
        }
    }
}
```

要註意的是測試代碼和產品代碼在同一個包. 雖然是main包, 也有對應的 main 入口函數, 但是在測試的時候 main 包隻是 TestEcho 測試函數導入的一個普通包, 里面 main 函數併沒有被導出是被忽略的.

通過將測試放到表格中, 我們很容易添加新的測試用例. 讓我通過增加下面的測試用例來看看失敗的情況是怎麼樣的:

```
{true
, ","
, []string
{"a"
, "b"
, "c"
}, "a b c\n"
}, // NOTE:
 wrong expectation!
```

`go test` 輸出如下:

```
$ go test gopl.io/ch11/echo
--- FAIL: TestEcho (0.00s)
    echo_test.go:31: echo(true, ",", ["a" "b" "c"]) = "a,b,c", want "a b c\n"
FAIL
FAIL        gopl.io/ch11/echo         0.006s
```

錯誤信息描述了嚐試的操作(使用Go類似語法), 實際的行爲, 和期望的行爲. 通過這樣的錯誤信息, 你可以在檢視代碼之前就很容易定位錯誤的原因.

要註意的是在測試代碼中併沒有調用 log.Fatal 或 os.Exit, 因爲調用這類函數會導致程序提前退出; 調用這些函數的特權應該放在 main 函數中. 如果眞的有以外的事情導致函數發送 panic, 測試驅動應該嚐試 recover, 然後將當前測試當作失敗處理. 如果是可預期的錯誤, 例如非法的用戶輸入, 找不到文件, 或配置文件不當等應該通過返迴一個非空的 error 的方式處理. 幸運的是(上面的意

外隻是一個插麭), 我們的 echo 示例是比較簡單的也沒有需要返迴非空error的情況.

11.2.3. 白盒測試

一個測試分類的方法是基於測試者是否需要了解被測試對象的內部工作原理. 黑盒測試隻需要測試包公開的文檔和API行爲, 內部實現對測試代碼是透明的. 相反, 白盒測試有訪問包內部函數和數據結構的權限, 因此可以做到一下普通客戶端無法實現的測試. 例如, 一個飽和測試可以在每個操作之後檢測不變量的數據類型. (白盒測試隻是一個傳統的名稱, 其實稱爲 clear box 會更準確.)

黑盒和白盒這兩種測試方法是互補的. 黑盒測試一般更健壯, 隨着軟件實現的完善測試代碼很少需要更新. 它們可以幫助測試者了解眞是客戶的需求, 可以幫助發現API設計的一些不足之處. 相反, 白盒測試則可以對內部一些棘手的實現提供更多的測試覆蓋.

我們已經看到兩種測試的例子. TestIsPalindrome 測試僅僅使用導出的 IsPalindrome 函數, 因此它是一個黑盒測試. TestEcho 測試則調用了內部的 echo 函數, 併且更新了內部的 out 全局變量, 這兩個都是未導出的, 因此它是白盒測試.

當我們開發TestEcho測試的時候, 我們脩改了 echo 函數使用包級的 out 作爲輸出對象, 因此測試代碼可以用另一個實現代替標準輸出, 這樣可以方便對比 echo 的輸出數據. 使用類似的技術, 我們可以將産品代碼的其他部分也替換爲一個容易測試的僞對象. 使用僞對象的好處是我們可以方便配置, 容易預測, 更可靠, 也更容易觀察. 同時也可以避免一些不良的副作用, 例如更新生産數據庫或信用卡消費行爲.

下面的代碼演示了爲用戶提供網絡存儲的web服務中的配額檢測邏輯. 當用戶使用了超過 90% 的存儲配額之後將發送提醒郵件.

```
gopl.io/ch11/storage1

package
 storage

import
 (
    "fmt"

    "log"

    "net/smtp"

)

func
 bytesInUse(username string
) int64
 { return
 0
 /* ... */
 }

// Email sender configuration.

// NOTE:
 never put passwords in source code!

const
 sender = "notifications@example.com"

const
```

```
const
 password = "correcthorsebatterystaple"

const
 hostname = "smtp.example.com"



const
 template = `Warning: you are using %d bytes of storage,
%d%% of your quota.`



func
 CheckQuota(username string
) {
    used := bytesInUse(username)
    const
 quota = 1000000000
 // 1GB

    percent := 100
 * used / quota
    if
 percent < 90
 {
        return
 // OK

    }
    msg := fmt.Sprintf(template, used, percent)
    auth := smtp.PlainAuth(""
, sender, password, hostname)
    err := smtp.SendMail(hostname+":587"
, auth, sender,
        []string
{username}, []byte
(msg))
    if
 err != nil
 {
        log.Printf("smtp.SendMail(%s) failed: %s"
, username, err)
    }
}
```

我們想測試這個代碼, 但是我們併不希望發送眞實的郵件. 因此我們將郵件處理邏輯放到一個私有的 notifyUser 函數.

```
gopl.io/ch11/storage2

var
 notifyUser = func
(username, msg string
) {
    auth := smtp.PlainAuth(""
, sender, password, hostname)
    err := smtp.SendMail(hostname+":587"
, auth, sender,
        []string
{username}, []byte
(msg))
    if
 err != nil
 {
        log.Printf("smtp.SendEmail(%s) failed: %s"
, username, err)
    }
}

func
 CheckQuota(username string
) {
    used := bytesInUse(username)
    const
 quota = 1000000000
 // 1GB

    percent := 100
 * used / quota
    if
 percent < 90
 {
        return
 // OK

    }
    msg := fmt.Sprintf(template, used, percent)
    notifyUser(username, msg)
}
```

現在我們可以在測試中用僞郵件發送函數替代眞實的郵件發送函數. 它隻是簡單記録要通知的用戶和郵件的內容.

```
package
 storage

import
 (
    "strings"
```

```
    "strings"

    "testing"

)
func
 TestCheckQuotaNotifiesUser(t *testing.T) {
    var
 notifiedUser, notifiedMsg string

    notifyUser = func
(user, msg string
) {
        notifiedUser, notifiedMsg = user, msg
    }

    // ...simulate a 980MB-used condition...


    const
 user = "joe@example.org"

    CheckQuota(user)
    if
 notifiedUser == ""
 && notifiedMsg == ""
 {
        t.Fatalf("notifyUser not called"
)
    }
    if
 notifiedUser != user {
        t.Errorf("wrong user (%s) notified, want %s"
,
            notifiedUser, user)
    }
    const
 wantSubstring = "98% of your quota"

    if
 !strings.Contains(notifiedMsg, wantSubstring) {
        t.Errorf("unexpected notification message <<%s>>, "
+
            "want substring %q"
, notifiedMsg, wantSubstring)
    }
}
```

這里有一個問題: 當測試函數返迴後, CheckQuota 將不能正常工作, 因爲 notifyUsers 依然使用的是測試函數的僞發送郵件函數. (當更新全局對象的時候總會有這種風險.) 我們必鬚脩改測試代碼恢複 notifyUsers 原先的狀態以便後續其他的測試沒有影響, 要確保所有的執行路徑後都能恢複, 包括測試失敗或 panic 情形. 在這種情況下, 我們建議使用 defer 處理恢複的代碼.

```
func
 TestCheckQuotaNotifiesUser(t *testing.T) {
    // Save and restore original notifyUser.

    saved := notifyUser
    defer
 func
() { notifyUser = saved }()

    // Install the test's fake notifyUser.

    var
 notifiedUser, notifiedMsg string

    notifyUser = func
(user, msg string
) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ...rest of test...

}
```

這種處理模式可以用來暫時保存和恢複所有的全局變量, 包括命令行標誌參數, 調試選項, 和優化參數; 安裝和移除導致生産代碼産生一些調試信息的鉤子函數; 還有有些誘導生産代碼進入某些重要狀態的改變, 比如 超時, 錯誤, 甚至是一些刻意製造的併發行爲.

以這種方式使用全局變量是安全的, 因爲 go test 併不會同時併發地執行多個測試.

11.2.4. 擴展測試包

考慮下這兩個包: net/url 包, 提供了 URL 解析的功能; net/http 包, 提供了web服務和HTTP客戶端的功能. 如我們所料, 上層的 net/http 包依賴下層的 net/url 包. 然後, net/url 包中的一個測試是演示不同URL和HTTP客戶端的交互行爲. 也就是說, 一個下層包的測試代碼導入了上層的包.

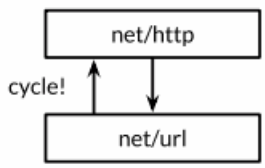


Figure 11.1. A test of `net/url` depends on `net/http`.

這樣的行爲在 net/url 包的測試代碼中會導致包的循環依賴, 正如 圖11.1中向上箭頭所示, 同時正如我們在 10.1節所說, Go語言規范是禁止包的循環依賴的.

我們可以通過測試擴展包的方式解決循環依賴的問題, 也就是在 net/url 包所在的目録聲明一個 url_test 測試擴展包. 其中測試擴展包名的 `_test` 後綴告訴 go test 工具它應該建立一個額外的包來運行測試. 我們將這個擴展測試包的導入路徑視作是 net/url_test 會更容易理解, 但實際上它併不能被其他任何包導入.

因爲測試擴展包是一個獨立的包, 因此可以導入測試代碼依賴的其他的輔助包; 包内的測試代碼可能無法做到. 在設計層面, 測試擴展包是在所以它依賴的包的上層, 正如 圖11.2所示.

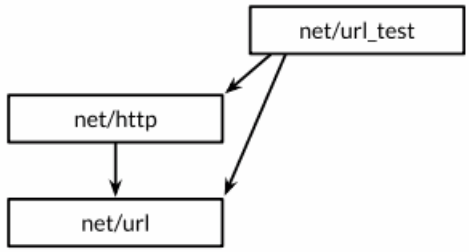


Figure 11.2. External test packages break dependency cycles.

通過迴避循環導入依賴, 擴展測試包可以更靈活的測試, 特別是集成測試(需要測試多個組件之間的交互), 可以像普通應用程序那樣自由地導入其他包.

我們可以用 go list 工具查看包對應目録中哪些Go源文件是産品代碼, 哪些是包內測試, 還哪些測試擴展包. 我們以 fmt 包作爲一個例子. GoFiles 表示産品代碼對應的Go源文件列表; 也就是 go build 命令要編譯的部分:

```
$ go list -f={{.GoFiles}} fmt
[doc.go format.go print.go scan.go]
```

TestGoFiles 表示的是 fmt 包內部測試測試代碼, 以 _test.go 爲後綴文件名, 不過隻在測試時被構建:

```
$ go list -f={{.TestGoFiles}} fmt
[export_test.go]
```

包的測試代碼通常都在這些文件中, 不過 fmt 包併非如此; 稍後我們再解釋 export_test.go 文件的作用.

XTestGoFiles 表示的是屬於測試擴展包的測試代碼, 也就是 fmt_test 包, 因此它們必鬚先導入 fmt 包. 同樣, 這些文件也隻是在測試時被構建運行:

```
$ go list -f={{.XTestGoFiles}} fmt
[fmt_test.go scan_test.go stringer_test.go]
```

有時候測試擴展包需要訪問被測試包內部的代碼, 例如在一個爲了避免循環導入而被獨立到外部測試擴展包的白盒測試. 在這種情況下, 我們可以通過一些技巧解決: 我們在包內的一個 _test.go 文件中導出一個內部的實現給測試擴展包. 因爲這些代碼隻有在測試時才需要, 因此一般放在 export_test.go 文件中.

例如, fmt 包的 fmt.Scanf 需要 unicode.IsSpace 函數提供的功能. 但是爲了避免太多的依賴, fmt 包併沒有導入包含鉅大表格數據的 unicode 包; 相反fmt包有一個叫 isSpace 內部的簡易實現.

爲了確保 fmt.isSpace 和 unicode.IsSpace 函數的行爲一致, fmt 包謹慎地包含了一個測試. 是一個在測試擴展包內的測試, 因此是無法直接訪問到 isSpace 內部函數的, 因此 fmt 通過一個祕密出口導出了 isSpace 函數. export_test.go 文件就是專門用於測試擴展包的祕密出口.

```
package fmt

var IsSpace = isSpace
```

這個測試文件併沒有定義測試代碼; 它隻是通過 fmt.IsSpace 簡單導出了內部的 isSpace 函數, 提供給測試擴展包使用. 這個技巧可以廣泛用於位於測試擴展包的白盒測試.

11.2.5. 編寫有效的測試

許多Go新人會驚異與它的極簡的測試框架. 很多其他語言的測試框架都提供了識別測試函數的機製(通常使用反射或元數據), 通過設置一些 ‘‘setup’’ 和 ‘‘teardown’’ 的鉤子函數來執行測試用例運行的初始化或之後的清理操作, 同時測試工具箱還提供了很多類似assert斷言, 比較值, 格式化輸出錯誤信息和停止一個識別的測試等輔助函數(通常使用異常機製). 雖然這些機製可以使得測試非常簡潔, 但是測試輸出的日誌卻像火星文一般難以理解. 此外, 雖然測試最終也會輸出 PASS 或 FAIL 的報告, 但是它們提供的信息格式卻非常不利於代碼維護者快速定位問題, 因爲失敗的信息的具體含義是非常隱患的, 比如 "assert: 0 == 1" 或 成頁的海量跟蹤日誌.

Go語言的測試風格則形成鮮明對比. 它期望測試者自己完成大部分的工作, 定義函數避免重複, 就像普通編程那樣. 編寫測試併不是一個機械的填充過程; 一個測試也有自己的接口, 盡管它的維護者也是測試僅有的一個用戶. 一個好的測試不應該引發其他無關的錯誤信息, 它隻要清晰簡潔地描述問題的癥狀即可, 有時候可能還需要一些上下文信息. 在理想情況下, 維護者可以在不看代碼的情況下就能根據錯誤信息定位錯誤産生的原因. 一個好的測試不應該在遇到一點小錯誤就立刻退出測試, 它應該嚐試報告更多的測試, 因此我們可能從多個失敗測試的模式中發現錯誤産生的規律.

下面的斷言函數比較兩個值, 然後生成一個通用的錯誤信息, 併停止程序. 它很方便使用也確實有效果, 但是當識別的時候, 錯誤時打印的信息幾乎是沒有價值的. 它併沒有爲解決問題提供一個很好的入口.

```
import
 (
    "fmt"

    "strings"

    "testing"

)
// A poor assertion function.

func
 assertEqual(x, y int
) {
    if
 x != y {
        panic
(fmt.Sprintf("%d != %d"
, x, y))
    }
}
func
 TestSplit(t *testing.T) {
    words := strings.Split("a:b:c"
, ":"
)
    assertEqual(len
(words), 3
)
    // ...

}
```

從這個意義上說, 斷言函數犯了過早抽象的錯誤: 僅僅測試兩個整數是否相同, 而放棄了根據上下文提供更有意義的錯誤信息的做法. 我們可以根據具體的錯誤打印一個更有價值的錯誤信息, 就像下面例子那樣. 測試在隻有一次重複的模式出現時引入抽象.

```
func
 TestSplit(t *testing.T) {
    s, sep := "a:b:c"
, ":"

    words := strings.Split(s, sep)
    if
 got, want := len
(words), 3
; got != want {
        t.Errorf("Split(%q, %q) returned %d words, want %d"
,
            s, sep, got, want)
    }
    // ...

}
```

現在的測試不僅報告了調用的具體函數, 它的輸入, 和結果的意義; 併且打印的眞實返迴的值和期望返迴的值; 併且卽使斷言失敗依然會繼續嚐試運行更多的測試. 一旦我們寫了這樣結構的測試, 下一步自然不是用更多的if語句來擴展測試用例, 我們可以用像IsPalindrome 的表驅動測試那樣來準備更多的 s, sep 測試用例.

前面的例子併不需要額外的輔助函數, 如果如果有可以使測試代碼更簡單的方法我們也樂意接受. (我們將在 13.3節 看到一個reflect.DeepEqual 輔助函數.) 開始一個好的測試的關鍵是通過實現你眞正想要的具體行爲, 然後才是考慮然後簡化測試代碼. 最好的結果是直接從庫的抽象接口開始, 針對公共接口編寫一些測試函數.

練習 11.5: 用表格驅動的技術擴展TestSplit測試, 併打印期望的輸出結果.

11.2.6. 避免的不穩定的測試

如果一個應用程序對於新出現的但有效的輸入經常失敗說明程序不夠穩健; 同樣如果一個測試僅僅因爲聲音變化就會導致失敗也是不合邏輯的. 就像一個不夠穩健的程序會挫敗它的用戶一樣, 一個脆弱性測試同樣會激怒它的維護者. 最脆弱的測試代碼會在程序沒有任何變化的時候産生不同的結果, 時好時壞, 處理它們會耗費大量的時間但是併不會得到任何好處.

當一個測試函數産生一個複雜的輸出如一個很長的字符串, 或一個精心設計的數據結構, 或一個文件, 它可以用於和預設的“golden”結果數據對比, 用這種簡單方式寫測試是誘人的. 但是隨着項目的發展, 輸出的某些部分很可能會發生變化, 盡管很可能是一個改進的實現導致的. 而且不僅僅是輸出部分, 函數複雜複製的輸入部分可能也跟着變化了, 因此測試使用的輸入也就不在有效了.

避免脆弱測試代碼的方法是隻檢測你眞正關心的屬性. 保存測試代碼的簡潔和內部結構的穩定. 特別是對斷言部分要有所選擇. 不要檢查字符串的全匹配, 但是尋找相關的子字符串, 因爲某些子字符串在項目的發展中是比較穩定不變的. 通常編寫一個重複雜的輸出中提取必要精華信息以用於斷言是值得的, 雖然這可能會帶來很多前期的工作, 但是它可以幫助迅速及時脩複因爲項目演化而導致的不合邏輯的失敗測試.

11.3. 測試覆蓋率

就其性質而言, 測試不可能是完整的. 計算機科學家 Edsger Dijkstra 曾說過: "測試可以顯示存在缺陷, 但是併不是說沒有BUG." 再多的測試也不能證明一個包沒有BUG. 在最好的情況下, 測試可以增強我們的信息, 包在我們測試的環境是可以正常工作的.

由測試驅動觸發運行到的被測試函數的代碼數目稱爲測試的覆蓋率. 測試覆蓋率併不能量化 — 甚至連最簡單的動態程序也難以精確測量 — 但是可以啟發併幫助我們編寫的有效的測試代碼.

這些幫助信息中語句的覆蓋率是最簡單和最廣泛使用的. 語句的覆蓋率是指在測試中至少被運行一次的代碼占總代碼數的比例. 在本節中, 我們使用 `go test` 中集成的測試覆蓋率工具, 來度量下面代碼的測試覆蓋率, 幫助我們識別測試和我們期望間的差距.

The code below is a table-driven test for the expression evaluator we built back in Chapter 7:

下面的代碼是一個表格驅動的測試, 用於測試第七章的表達式求值程序:

```
gopl.io/ch7/eval


func
 TestCoverage(t *testing.T) {
    var
 tests = []struct
 {
        input string

        env   Env
        want  string
 // expected error from Parse/Check or result from Eval

    }{
        {"x % 2"
, nil
, "unexpected '%'"
},
        {"!true"
, nil
, "unexpected '!'"
},
        {"log(10)"
, nil
, `unknown function "log"`
},
        {"sqrt(1, 2)"
, nil
, "call to sqrt has 2 args, want 1"
},
        {"sqrt(A / pi)"
, Env{"A"
: 87616
, "pi"
: math.Pi}, "167"
},
        {"pow(x, 3) + pow(y, 3)"
, Env{"x"
```

```
: 9
, "y"
: 10
}, "1729"
},
        {"5 / 9 * (F - 32)"
, Env{"F"
: -40
}, "-40"
},
    }

    for
 _, test := range
 tests {
        expr, err := Parse(test.input)
        if
 err == nil
 {
            err = expr.Check(map
[Var]bool
{})
        }
        if
 err != nil
 {
            if
 err.Error() != test.want {
                t.Errorf("%s: got %q, want %q"
, test.input, err, test.want)
            }
            continue

        }
        got := fmt.Sprintf("%.6g"
, expr.Eval(test.env))
        if
 got != test.want {
            t.Errorf("%s: %v => %s, want %s"
,
                test.input, test.env, got, test.want)
        }
    }
}
```

首先, 我們要確保所有的測試都正常通過:

```
$ go test -v -run=Coverage gopl.io/ch7/eval
=== RUN TestCoverage
--- PASS: TestCoverage (0.00s)
PASS
ok      gopl.io/ch7/eval         0.011s
```

下面這個命令可以顯示測試覆蓋率工具的用法信息:

```
$ go tool cover
Usage of 'go tool cover':
Given a coverage profile produced by 'go test':
    go test -coverprofile=c.out

Open a web browser displaying annotated source code:
    go tool cover -html=c.out
...
```

`go tool` 命令運行Go工具鏈的底層可執行程序. 這些底層可執行程序放在 $GOROOT/pkg/tool/${GOOS}_${GOARCH} 目録. 因爲 `go build` 的原因, 我們很小直接調用這些底層工具.

現在我們可以用 `-coverprofile` 標誌參數重新運行:

```
$ go test -run=Coverage -coverprofile=c.out gopl.io/ch7/eval
ok      gopl.io/ch7/eval         0.032s      coverage: 68.5% of statements
```

這個標誌參數通過插入生成鉤子代碼來統計覆蓋率數據. 也就是説, 在運行每個測試前, 它會脩改要測試代碼的副本, 在每個塊都會設置一個布爾標誌變量. 當被脩改後的被測試代碼運行退出時, 將統計日誌數據寫入 c.out 文件, 併打印一部分執行的語句的一個總結. (如果你需要的是摘要,使用 `go test -cover` .)

如果使用了 `-covermode=count` 標誌參數, 那麽將在每個代碼塊插入一個計數器而不是布爾標誌量. 在統計結果中記録了每個塊的執行次數, 這可以用於衡量哪些是被頻繁執行的熱點代碼.

爲了收集數據, 我們運行了測試覆蓋率工具, 打印了測試日誌, 生成一個HTML報告, 然後在瀏覽器中打開(圖11.3).

```
$ go tool cover -html=c.out
```

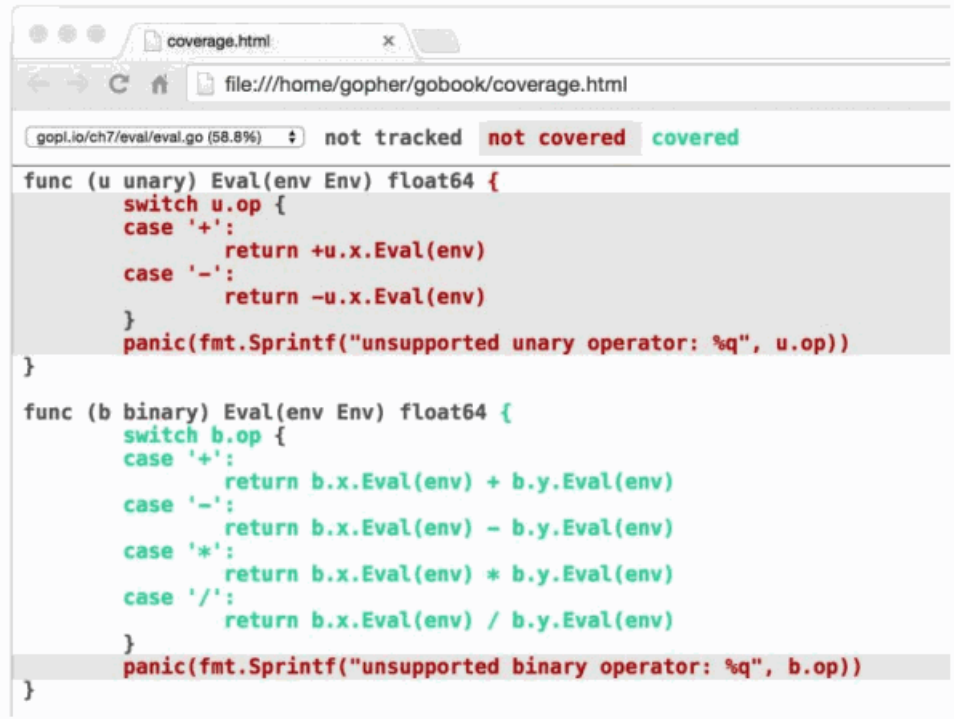


Figure 11.3. A coverage report.

綠色的代碼塊被測試覆蓋到了, 紅色的則表示沒有被覆蓋到. 爲了清晰起見, 我們將的背景紅色文本的背景設置成了陰影效果. 我們可以馬上發現 unary 操作的 Eval 方法併沒有被執行到. 如果我們針對這部分未被覆蓋的代碼添加下面的測試, 然後重新運行上面的命令, 那麼我們將會看到那個紅色部分的代碼也變成綠色了:

```
{"-x * -x", eval.Env{"x": 2}, "4"}
```

不過兩個 panic 語句依然是紅色的. 這是沒有問題的, 因爲這兩個語句併不會被執行到.

實現 100% 的測試覆蓋率聽起來很好, 但是在具體實踐中通常是不可行的, 也不是值得推薦的做法. 因爲那隻能說明代碼被執行過而已, 併不意味着代碼是沒有BUG的; 因爲對於邏輯複雜的語句需要針對不同的輸入執行多次. 有一些語句, 例如上面的 panic 語句則永遠都不會被執行到. 另外, 還有一些隱晦的錯誤在現實中很少遇到也很難編寫對應的測試代碼. 測試從本質上來說是一個比較務實的工作, 編寫測試代碼和編寫應用代碼的成本對比是需要考慮的. 測試覆蓋率工具可以幫助我們快速識別測試薄弱的地方, 但是設計好的測試用例和編寫應用代碼一樣需要嚴密的思考.

11.4. 基準測試

基準測試是測量一個程序在固定工作負載下的性能. 在Go語言中, 基準測試函數和普通測試函數類似, 但是以Benchmark爲前綴名, 併且帶有一個 `*testing.B` 類型的參數; `*testing.B` 除了提供和 `*testing.T` 類似的方法, 還有額外一些和性能測量相關的方法. 它還提供了一個整數N, 用於指定操作執行的循環次數.

下面是 IsPalindrome 函數的基準測試, 其中循環將執行N次.

```
import
 "testing"


func
 BenchmarkIsPalindrome(b *testing.B) {
    for
 i := 0
; i < b.N; i++ {
        IsPalindrome("A man, a plan, a canal: Panama"
)
    }
}
```

我們用下面的命令運行基準測試. 和普通測試不同的是, 默認情況下不運行任何基準測試. 我們需要通過 `-bench` 命令行標誌參數手工指定要運行的基準測試函數. 該參數是一個正則表達式, 用於匹配要執行的基準測試函數的名字, 默認值是空的. 其中 ‘‘.’’ 模式將可以匹配所有基準測試函數, 但是這里總共隻有一個基準測試函數, 因此 和 `-bench=IsPalindrome` 參數是等價的效果.

```
$ cd $GOPATH/src/gopl.io/ch11/word2
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000               1035 ns/op
ok      gopl.io/ch11/word2      2.179s
```

基準測試名的數字後綴部分, 這里是8, 表示運行時對應的 GOMAXPROCS 的值, 這對於一些和併發相關的基準測試是重要的信息.

報告顯示每次調用 IsPalindrome 函數花費 1.035微秒, 是執行 1,000,000 次的平均時間. 因爲基準測試驅動器併不知道每個基準測試函數運行所花的時候, 它會嚐試在眞正運行基準測試前先嚐試用較小的 N 運行測試來估算基準測試函數所需要的時間, 然後推斷一個較大的時間保證穩定的測量結果.

循環在基準測試函數內實現, 而不是放在基準測試框架內實現, 這樣可以讓每個基準測試函數有機會在循環啟動前執行初始化代碼, 這樣併不會顯著影響每次迭代的平均運行時間. 如果還是擔心初始化代碼部分對測量時間帶來榦擾, 那麽可以通過 testing.B 參數的方法來臨時關閉或重置計時器, 不過這些一般很少會用到.

現在我們有了一個基準測試和普通測試, 我們可以很容易測試新的讓程序運行更快的想法. 也許最明顯的優化是在 IsPalindrome 函數中第二個循環的停止檢查, 這樣可以避免每個比較都做兩次:

```
n := len
(letters)/2

for
 i := 0
; i < n; i++ {
    if
 letters[i] != letters[len
(letters)-1
-i] {
        return
 false

    }
}
return
 true
```

不過很多情況下, 一個明顯的優化併不一定就能代碼預期的效果. 這個改進在基準測試中值帶來了 4% 的性能提陞.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000              992 ns/op
ok      gopl.io/ch11/word2      2.093s
```

另一個改進想法是在開始爲每個字符預先分配一個足夠大的數組, 這樣就可以避免在 append 調用時可能會導致內存的多次重新分配. 聲明一個 letters 數組變量, 併指定合適的大小, 像這樣,

```
letters := make
([]rune
, 0
, len
(s))
for
 _, r := range
 s {
    if
 unicode.IsLetter(r) {
        letters = append
(letters, unicode.ToLower(r))
    }
}
```

這個改進提陞性能約 35%, 報告結果是基於 2,000,000 次迭代的平均運行時間統計.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 2000000                      697 ns/op
ok      gopl.io/ch11/word2      1.468s
```

如這個例子所示, 快的程序往往是有很少的內存分配. `-benchmem` 命令行標誌參數將在報告中包含內存的分配數據統計. 我們可以比較優化前後內存的分配情況:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    1000000   1026 ns/op    304 B/op  4 allocs/op
```

這是優化之後的結果:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome    2000000    807 ns/op    128 B/op  1 allocs/op
```

一次內存分配代替多次的內存分配節省了75%的分配調用次數和減少近一半的內存需求.

這個基準測試告訴我們所需的絶對時間依賴給定的具體操作, 兩個不同的操作所需時間的差異也是和不同環境相關的. 例如, 如果一個函數需要 1ms 處理 1,000 個元素, 那麼處理 10000 或 1百萬 將需要多少時間呢? 這樣的比較揭示了漸近增長函數的運行時間. 另一個例子: I/O 緩存該設置爲多大呢? 基準測試可以幫助我們選擇較小的緩存但能帶來滿意的性能. 第三個例子: 對於一個確定的工作那種算法更好? 基準測試可以評估兩種不同算法對於相同的輸入在不同的場景和負載下的優缺點.

比較基準測試都是結構類似的代碼. 它們通常是采用一個參數的函數, 從幾個標誌的基準測試函數入口調用, 就像這樣:

```
func
 benchmark(b *testing.B, size int
) { /* ... */
 }
func
 Benchmark10(b *testing.B)         { benchmark(b, 10
) }
func
 Benchmark100(b *testing.B)        { benchmark(b, 100
) }
func
 Benchmark1000(b *testing.B)       { benchmark(b, 1000
) }
```

通過函數參數來指定輸入的大小, 但是參數變量對於每個具體的基準測試都是固定的. 要避免直接脩改 b.N 來控製輸入的大小. 除非你將它作爲一個固定大小的迭代計算輸入, 否則基準測試的結果將毫無意義.

基準測試對於編寫代碼是很有幫助的, 但是即使工作完成了應應當保存基準測試代碼. 因爲隨着項目的發展, 或者是輸入的增加, 或者是部署到新的操作繫統或不同的處理器, 我們可以再次用基準測試來幫助我們改進設計.

練習 11.6: 爲 2.6.2節 的 練習 2.4 和 練習 2.5 的 PopCount 函數編寫基準測試. 看看基於表格算法在不同情況下的性能.

練習 11.7: 爲 *IntSet (§6.5) 的 Add, UnionWith 和 其他方法編寫基準測試, 使用大量隨機出入. 你可以讓這些方法跑多快? 選擇字的大小對於性能的影響如何? IntSet 和基於內建 map 的實現相比有多快?

11.5. 剖析

測量基準對於衡量特定操作的性能是有幫助的, 但是, 當我們視圖讓程序跑的更快的時候, 我們通常併不知道從哪里開始優化. 每個碼農都應該知道 Donald Knuth 在1974年的 ‘‘Structured Programming with go to Statements’’ 上所說的格言. 雖然經常被解讀爲不重視性能的意思, 但是從原文我們可以看到不同的含義:

> 毫無疑問, 效率會導致各種濫用. 程序員需要浪費大量的時間思考, 或者擔心, 被部分程序的速度所榦擾, 實際上這些嚐試提陞效率的行爲可能産生強烈的負面影響, 特別是當調試和維護的時候. 我們不應該過度糾結於細節的優化, 應該説約97%的場景: 過早的優化是萬惡之源.
>
> 我們當然不應該放棄那關鍵的3%的機會. 一個好的程序員不會因爲這個理由而滿足, 他們會明智地觀察和識別哪些是關鍵的代碼; 但是隻有在關鍵代碼已經被確認的前提下才會進行優化. 對於判斷哪些部分是關鍵代碼是經常容易犯經驗性錯誤的地方, 因此程序員普通使用的測量工具, 使得他們的直覺很不靠譜.

當我們想仔細觀察我們程序的運行速度的時候, 最好的技術是如何識別關鍵代碼. 自動化的剖析技術是基於程序執行期間一些抽樣數據, 然後推斷後面的執行狀態; 最終産生一個運行時間的統計數據文件.

Go語言支持多種類型的剖析性能分析, 每一種關註不同的方面, 但它們都涉及到每個采樣記録的感興趣的一繫列事件消息, 每個事件都包含函數調用時函數調用堆棧的信息. 內建的 `go test` 工具對幾種分析方式都提供了支持.

CPU分析文件標識了函數執行時所需要的CPU時間. 當前運行的繫統線程在每隔幾毫秒都會遇到操作繫統的中斷事件, 每次中斷時都會記録一個分析文件然後恢複正常的運行.

堆分析則記録了程序的內存使用情況. 每個內存分配操作都會觸發內部平均內存分配例程, 每個 512KB 的內存申請都會觸發一個事件.

阻塞分析則記録了goroutine最大的阻塞操作, 例如繫統調用, 管道發送和接收, 還有獲取鎖等. 分析庫會記録每個goroutine被阻塞時的相關操作.

在測試環境下隻需要一個標誌參數就可以生成各種分析文件. 當一次使用多個標誌參數時需要當心, 因爲分析操作本身也可能會影像程序的運行.

```
$ go test -cpuprofile=cpu.out
$ go test -blockprofile=block.out
$ go test -memprofile=mem.out
```

對於一些非測試程序也很容易支持分析的特性, 具體的實現方式和程序是短時間運行的小工具還是長時間運行的服務會有很大不同, 因此Go的runtim運行時包提供了程序運行時控製分析特性的接口.

一旦我們已經收集到了用於分析的采樣數據, 我們就可以使用 pprof據來分析這些數據. 這是Go工具箱自帶的一個工具, 但併不是一個日常工具, 它對應 `go tool pprof` 命令. 該命令有許多特性和選項, 但是最重要的有兩個, 就是生成這個概要文件的可執行程序和對於的分析日誌文件.

爲了提高分析效率和減少空間, 分析日誌本身併不包含函數的名字; 它隻包含函數對應的地址. 也就是説pprof需要和分析日誌對於的可執行程序. 雖然 `go test` 命令通常會丟棄臨時用的測試程序, 但是在啟用分析的時候會將測試程序保存爲 foo.test 文件, 其中 foo 部分對於測試包的名字.

下面的命令演示了如何生成一個CPU分析文件. 我們選擇 `net/http` 包的一個基準測試. 通常是基於一個已經確定了是關鍵代碼的部分進行基準測試. 基準測試會默認包含單元測試, 這里我們用 -run=NONE 禁止單元測試.

```
$ go test -run=NONE -bench=ClientServerParallelTLS64 \
    -cpuprofile=cpu.log net/http
 PASS
 BenchmarkClientServerParallelTLS64-8  1000
    3141325 ns/op  143010 B/op  1747 allocs/op
ok       net/http       3.395s

$ go tool pprof -text -nodecount=10 ./http.test cpu.log
2570ms of 3590ms total (71.59%)
Dropped 129 nodes (cum <= 17.95ms)
Showing top 10 nodes out of 166 (cum >= 60ms)
    flat  flat%   sum%     cum   cum%
  1730ms 48.19% 48.19%  1750ms 48.75%  crypto/elliptic.p256ReduceDegree
   230ms  6.41% 54.60%   250ms  6.96%  crypto/elliptic.p256Diff
   120ms  3.34% 57.94%   120ms  3.34%  math/big.addMulVVW
   110ms  3.06% 61.00%   110ms  3.06%  syscall.Syscall
    90ms  2.51% 63.51%  1130ms 31.48%  crypto/elliptic.p256Square
    70ms  1.95% 65.46%   120ms  3.34%  runtime.scanobject
    60ms  1.67% 67.13%   830ms 23.12%  crypto/elliptic.p256Mul
    60ms  1.67% 68.80%   190ms  5.29%  math/big.nat.montgomery
    50ms  1.39% 70.19%    50ms  1.39%  crypto/elliptic.p256ReduceCarry
    50ms  1.39% 71.59%    60ms  1.67%  crypto/elliptic.p256Sum
```

參數 `-text` 標誌參數用於指定輸出格式, 在這里每行是一個函數, 根據使用CPU的時間來排序. 其中 `-nodecount=10` 標誌參數限製了隻輸出前10行的結果. 對於嚴重的性能問題, 這個文本格式基本可以幫助查明原因了.

這個概要文件告訴我們, HTTPS基準測試中 `crypto/elliptic.p256ReduceDegree` 函數占用了將近一般的CPU資源. 相比之下, 如果一個概要文件中主要是runtime包的內存分配的函數, 那麽減少內存消耗可能是一個值得嚐試的優化策略.

對於一些更微妙的問題, 你可能需要使用 pprof 的圖形顯示功能. 這個需要安裝 GraphViz 工具, 可以從 www.graphviz.org 下載. 參數 `-web` 用於生成一個有向圖文件, 包含CPU的使用和最特點的函數等信息.

這一節我們隻是簡單看了下Go語言的分析據工具. 如果想了解更多, 可以閱讀 Go官方博客的 ‘‘Profiling Go Programs’’ 一文.

11.6. 示例函數

第三種 `go test` 特別處理的函數是示例函數, 以 Example 爲函數名開頭. 示例函數沒有函數參數和返迴值. 下面是 IsPalindrome 函數對應的示例函數:

```
func
 ExampleIsPalindrome() {
    fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"
))
    fmt.Println(IsPalindrome("palindrome"
))
    // Output:

    // true

    // false

}
```

示例函數有三個用處. 最主要的一個是用於文檔: 一個包的例子可以更簡潔直觀的方式來演示函數的用法, 會文字描述會更直接易懂, 特別是作爲一個提醒或快速參考時. 一個例子函數也可以方便展示屬於同一個接口的幾種類型或函數直接的關繫, 所有的文檔都必鬚關聯到一個地方, 就像一個類型或函數聲明都統一到包一樣. 同時, 示例函數和註釋併不一樣, 示例函數是完整眞是的Go代碼, 需要介紹編譯器的編譯時檢查, 這樣可以保證示例代碼不會腐爛成不能使用的舊代碼.

根據示例函數的後綴名部分, godoc 的web文檔會將一個示例函數關聯到某個具體函數或包本身, 因此 ExampleIsPalindrome 示例函數將是 IsPalindrome 函數文檔的一部分, Example 示例函數將是包文檔的一部分.

示例文檔的第二個用處是在 `go test` 執行測試的時候也運行示例函數測試. 如果示例函數內含有類似上面例子中的 `// Output:` 這樣的註釋, 那麼測試工具會執行這個示例函數, 然後檢測這個示例函數的標準輸出和註釋是否匹配.

示例函數的第三個目的提供一個眞實的演練場. golang.org 是由 dogoc 提供的服務, 它使用了 Go Playground 技術讓用戶可以在瀏覽器中在線編輯和運行每個示例函數, 就像 圖 11.4 所示的那樣. 這通常是學習函數使用或Go語言特性的最快方式.

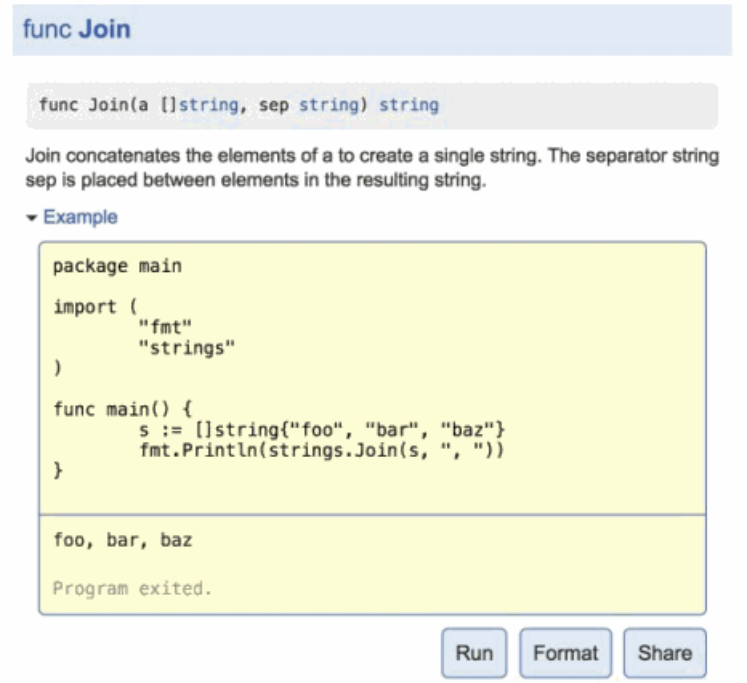


Figure 11.4. An interactive example of `strings.Join` in godoc.

本書最後的兩掌是討論 reflect 和 unsafe 包, 一般的Go用於很少需要使用它們. 因此, 如果你還沒有寫過任何眞是的Go程序的話, 現在可以忽略剩餘部分而直接編碼了.

第十二章 反射

Go提供了一種機製在運行時更新變量和檢查它們的值, 調用它們的方法, 和它們支持的內在操作, 但是在編譯時併不知道這些變量的類型. 這種機製被稱爲反射. 反射也可以讓我們將類型本身作爲第一類的值類型處理.

在本章, 我們將探討Go語言的反射特性, 看看它可以給語言增加哪些表達力, 以及在兩個至關重要的API是如何用反射機製的: 一個是 fmt 包提供的字符串格式功能, 另一個是類似 encoding/json 和 encoding/xml 提供的針對特定協議的編解碼功能. 對於我們在4.6節中看到過的 text/template 和 html/template 包, 它們的實現也是依賴反射技術的. 然後, 反射是一個複雜的內省技術, 而應該隨意使用, 因此, 盡管上面這些包都是用反射技術實現的, 但是它們自己的API都沒有公開反射相關的接口.

12.1. 爲何需要反射?

有時候我們需要編寫一個函數能夠處理一類併不滿足普通公共接口的類型的值, 也可能它們併沒有確定的表示方式, 或者在我們設計該函數的時候還這些類型可能還不存在, 各種情況都有可能.

一個大家熟悉的例子是 fmt.Fprintf 函數提供的字符串格式化處理邏輯, 它可以用例對任意類型的值格式化打印, 甚至是用戶自定義的類型. 讓我們來嚐試實現一個類似功能的函數. 簡單起見, 我們的函數隻接收一個參數, 然後返迴和 fmt.Sprint 類似的格式化後的字符串, 我們的函數名也叫 Sprint.

我們使用了 switch 分支首先來測試輸入參數是否實現了 String 方法, 如果是的話就使用該方法. 然後繼續增加測試分支, 檢查是否是每個基於 string, int, bool 等基礎類型的動態類型, 併在每種情況下執行適當的格式化操作.

```
func
 Sprint(x interface
{}) string
 {
    type
 stringer interface
 {
        String() string

    }
    switch
 x := x.(type
) {
    case
 stringer:
        return
 x.String()
    case
 string
:
        return
 x
    case
 int
:
        return
 strconv.Itoa(x)
    // ...similar cases for int16, uint32, and so on...

    case
 bool
:
        if
 x {
            return
 "true"

        }
        return
 "false"
```

```
    default
:
        // array, chan, func, map, pointer, slice, struct

        return
 "???"

    }
}
```

但是我們如何處理其它類似 []float64, map[string][]string 等類型呢? 我們當然可以添加更多的測試分支, 但是這些組合類型的數目基本是無窮的. 還有如何處理 url.Values 等命令的類型呢? 雖然類型分支可以識別出底層的基礎類型是 map[string][]string, 但是它併不匹配 url.Values 類型, 因爲這是兩種不同的類型, 而且 switch 分支也不可能包含每個類似 url.Values 的類型, 這會導致對這些庫的依賴.

沒有一種方法來檢查未知類型的表示方式, 我們被卡住了. 這就是我們爲何需要反射的原因.

12.2. reflect.Type和reflect.Value

反射是由 reflect 包提供支持. 它定義了兩個重要的類型, Type 和 Value. 一個 Type 表示一個Go類型. 它是一個接口, 有許多方法來區分類型和檢查它們的組件, 例如一個結構體的成員或一個函數的參數等. 唯一能反映 reflect.Type 實現的是接口的類型描述信息(§7.5), 同樣的實體標識了動態類型的接口值.

函數 reflect.TypeOf 接受任意的 interface{} 類型, 併返迴對應動態類型的reflect.Type:

```
t := reflect.TypeOf(3
)  // a reflect.Type

fmt.Println(t.String()) // "int"

fmt.Println(t)          // "int"
```

其中 TypeOf(3) 調用將值 3 作爲 interface{} 類型參數傳入. 迴到 7.5節 的將一個具體的值轉爲接口類型會有一個隱式的接口轉換操作, 它會創建一個包含兩個信息的接口值: 操作數的動態類型(這里是int)和它的動態的值(這里是3).

因爲 reflect.TypeOf 返迴的是一個動態類型的接口值, 它總是返迴具體的類型. 因此, 下面的代碼將打印 "*os.File" 而不是 "io.Writer". 稍後, 我們將看到 reflect.Type 是具有識別接口類型的表達方式功能的.

```
var
 w io.Writer = os.Stdout
fmt.Println(reflect.TypeOf(w)) // "*os.File"
```

要註意的是 reflect.Type 接口是滿足 fmt.Stringer 接口的. 因爲打印動態類型值對於調試和日誌是有幫助的, fmt.Printf 提供了一個簡短的 %T 標誌參數, 內部使用 reflect.TypeOf 的結果輸出:

```
fmt.Printf("%T\n"
, 3
) // "int"
```

reflect 包中另一個重要的類型是 Value. 一個 reflect.Value 可以持有一個任意類型的值. 函數 reflect.ValueOf 接受任意的 interface{} 類型, 併返迴對應動態類型的reflect.Value. 和 reflect.TypeOf 類似, reflect.ValueOf 返迴的結果也是對於具體的類型, 但是 reflect.Value 也可以持有一個接口值.

```
v := reflect.ValueOf(3
) // a reflect.Value

fmt.Println(v)          // "3"

fmt.Printf("%v\n"
, v)   // "3"

fmt.Println(v.String()) // NOTE:
 "<int Value>"
```

和 reflect.Type 類似, reflect.Value 也滿足 fmt.Stringer 接口, 但是除非 Value 持有的是字符串, 否則 String 隻是返迴具體的類型. 相同, 使用 fmt 包的 %v 標誌參數, 將使用 reflect.Values 的結果格式化.

調用 Value 的 Type 方法將返迴具體類型所對應的 reflect.Type:

```
t := v.Type()           // a reflect.Type

fmt.Println(t.String()) // "int"
```

逆操作是調用 reflect.ValueOf 對應的 reflect.Value.Interface 方法. 它返迴一個 interface{} 類型表示 reflect.Value 對應類型的具體值:

```
v := reflect.ValueOf(3
) // a reflect.Value

x := v.Interface()      // an interface{}

i := x.(int
)            // an int

fmt.Printf("%d\n"
, i)   // "3"
```

一個 reflect.Value 和 interface{} 都能保存任意的值. 所不同的是, 一個空的接口隱藏了值對應的表示方式和所有的公開的方法, 因此隻有我們知道具體的動態類型才能使用類型斷言來訪問內部的值(就像上面那樣), 對於內部值併沒有特別可做的事情. 相比之下, 一個 Value 則有很多方法來檢查其內容, 無論它的具體類型是什麼. 讓我們再次嚐試實現我們的格式化函數 format.Any.

我們使用 reflect.Value 的 Kind 方法來替代之前的類型 switch. 雖然還是有無窮多的類型, 但是它們的kinds類型卻是有限的: Bool, String 和 所有數字類型的基礎類型; Array 和 Struct 對應的聚合類型; Chan, Func, Ptr, Slice, 和 Map 對應的引用類似; 接口類型; 還有表示空值的無效類型. (空的 reflect.Value 對應 Invalid 無效類型.)

```
gopl.io/ch12/format
package
 format

import
```

```
 (
    "reflect"

    "strconv"

)

// Any formats any value as a string.

func
 Any(value interface
{}) string
 {
    return
 formatAtom(reflect.ValueOf(value))
}

// formatAtom formats a value without inspecting its internal structure.

func
 formatAtom(v reflect.Value) string
 {
    switch
 v.Kind() {
    case
 reflect.Invalid:
        return
 "invalid"

    case
 reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return
 strconv.FormatInt(v.Int(), 10
)
    case
 reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return
 strconv.FormatUint(v.Uint(), 10
)
    // ...floating-point and complex cases omitted for brevity...

    case
 reflect.Bool:
        return
 strconv.FormatBool(v.Bool())
    case
 reflect.String:
        return
 strconv.Quote(v.String())
    case
```

```
 reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
        return
 v.Type().String() + " 0x"
 +
            strconv.FormatUint(uint64
(v.Pointer()), 16
)
    default
: // reflect.Array, reflect.Struct, reflect.Interface

        return
 v.Type().String() + " value"

    }
}
```

到目前未知, 我們的函數將每個值視作一個不可分割沒有內部結構的, 因此它叫 formatAtom. 對於聚合類型(結構體和數組)個接口隻是打印類型的值, 對於引用類型(channels, functions, pointers, slices, 和 maps), 它十六進製打印類型的引用地址. 雖然還不夠理想, 但是依然是一個重大的進步, 併且 Kind 隻關心底層表示, format.Any 也支持新命名的類型. 例如:

```
var
 x int64
 = 1

var
 d time.Duration = 1
 * time.Nanosecond
fmt.Println(format.Any(x))                  // "1"

fmt.Println(format.Any(d))                  // "1"

fmt.Println(format.Any([]int64
{x}))         // "[]int64 0x8202b87b0"

fmt.Println(format.Any([]time.Duration{d})) // "[]time.Duration 0x8202b87e0"
```

12.3. Display遞歸打印

接下來，讓我們看看如何改善聚合數據類型的顯示。我們併不想完全剋隆一個fmt.Sprint函數，我們隻是像構建一個用於調式用的Display函數，給定一個聚合類型x，打印這個值對應的完整的結構，同時記録每個發現的每個元素的路徑。讓我們從一個例子開始。

```
e, _ := eval.Parse("sqrt(A / pi)"
)
Display("e"
, e)
```

在上面的調用中，傳入Display函數的參數是在7.9節一個表達式求值函數返迴的語法樹。Display函數的輸出如下：

```
Display e (eval.call):
e.fn = "sqrt"

e.args[0
].type
 = eval.binary
e.args[0
].value.op = 47

e.args[0
].value.x.type
 = eval.Var
e.args[0
].value.x.value = "A"

e.args[0
].value.y.type
 = eval.Var
e.args[0
].value.y.value = "pi"
```

在可能的情況下，你應該避免在一個包中暴露和反射相關的接口。我們將定義一個未導出的display函數用於遞歸處理工作，導出的是Display函數，它隻是display函數簡單的包裝以接受interface{}類型的參數：

```
gopl.io/ch12/display

func
 Display(name string
, x interface
{}) {
    fmt.Printf("Display %s (%T):\n"
, name, x)
    display(name, reflect.ValueOf(x))
}
```

在display函數中，我們使用了前面定義的打印基礎類型——基本類型、函數和chan等——元素值的formatAtom函數，但是我們會使用reflect.Value的方法來遞歸顯示聚合類型的每一個成員或元素。在遞歸下降過程中，path字符串，從最開始傳入的起始值（這里是“e”），將逐步增長以表示如何達到當前值（例如“e.args[0].value”）。

因爲我們不再模擬fmt.Sprint函數，我們將直接使用fmt包來簡化我們的例子實現。

```
func
 display(path string
, v reflect.Value) {
    switch
 v.Kind() {
    case
 reflect.Invalid:
        fmt.Printf("%s = invalid\n"
, path)
    case
 reflect.Slice, reflect.Array:
        for
 i := 0
; i < v.Len(); i++ {
            display(fmt.Sprintf("%s[%d]"
, path, i), v.Index(i))
        }
    case
 reflect.Struct:
        for
 i := 0
; i < v.NumField(); i++ {
            fieldPath := fmt.Sprintf("%s.%s"
, path, v.Type().Field(i).Name)
            display(fieldPath, v.Field(i))
        }
    case
 reflect.Map:
        for
 _, key := range
 v.MapKeys() {
            display(fmt.Sprintf("%s[%s]"
, path,
```

```
                formatAtom(key)), v.MapIndex(key))
        }
    case
 reflect.Ptr:
        if
 v.IsNil() {
            fmt.Printf("%s = nil\n"
, path)
        } else
 {
            display(fmt.Sprintf("(*%s)"
, path), v.Elem())
        }
    case
 reflect.Interface:
        if
 v.IsNil() {
            fmt.Printf("%s = nil\n"
, path)
        } else
 {
            fmt.Printf("%s.type = %s\n"
, path, v.Elem().Type())
            display(path+".value"
, v.Elem())
        }
    default
: // basic types, channels, funcs

        fmt.Printf("%s = %s\n"
, path, formatAtom(v))
    }
}
```

讓我們針對不同類型分別討論。

Slice和數組： 兩種的處理邏輯是一樣的。Len方法返迴slice或數組值中的元素個數，Index(i)活動索引i對應的元素，返迴的也是一個reflect.Value類型的值；如果索引i超出范圍的話將導致panic異常，這些行爲和數組或slice類型內建的len(a)和a[i]等操作類似。display針對序列中的每個元素遞歸調用自身處理，我們通過在遞歸處理時向path附加“[i]”來表示訪問路徑。

雖然reflect.Value類型帶有很多方法，但是隻有少數的方法對任意值都是可以安全調用的。例如，Index方法隻能對Slice、數組或字符串類型的值調用，其它類型如果調用將導致panic異常。

結構體： NumField方法報告結構體中成員的數量，Field(i)以reflect.Value類型返迴第i個成員的值。成員列表包含了匿名成員在內的全部成員。通過在path添加“.f”來表示成員路徑，我們必鬚獲得結構體對應的reflect.Type類型信息，包含結構體類型和第i個成員的名字。

Maps: MapKeys方法返迴一個reflect.Value類型的slice，每一個都對應map的可以。和往常一樣，遍歷map時順序是隨機的。MapIndex(key)返迴map中key對應的value。我們向path添加“[key]”來表示訪問路徑。（我們這里有一個未完成的工作。其實map的key的類型併不局限於formatAtom能完美處理的類型；數組、結構體和接口都可以作爲map的key。針對這種類型，完善key的顯示信息是練習12.1的任務。）

指針： Elem方法返迴指針指向的變量，還是reflect.Value類型。技術指針是nil，這個操作也是安全的，在這種情況下指針是Invalid無效類型，但是我們可以用IsNil方法來顯式地測試一個空指針，這樣我們可以打印更合適的信息。我們在path前面添加“*”，併用括弧包含以避免歧義。

接口： 再一次，我們使用IsNil方法來測試接口是否是nil，如果不是，我們可以調用v.Elem()來獲取接口對應的動態值，併且打印對應的類型和值。

現在我們的Display函數總算完工了，讓我們看看它的表現吧。下面的Movie類型是在4.5節的電影類型上演變來的：

```
type
 Movie struct
 {
    Title, Subtitle string

    Year            int

    Color           bool

    Actor           map
[string
]string

    Oscars          []string

    Sequel          *string

}
```

讓我們聲明一個該類型的變量，然後看看Display函數如何顯示它：

```
strangelove := Movie{
    Title:    "Dr. Strangelove"
,
    Subtitle: "How I Learned to Stop Worrying and Love the Bomb"
,
    Year:     1964
,
    Color:    false
,
    Actor: map
[string
]string
{
        "Dr. Strangelove"
:            "Peter Sellers"
,
        "Grp. Capt. Lionel Mandrake"
: "Peter Sellers"
,
        "Pres. Merkin Muffley"
:       "Peter Sellers"
,
        "Gen. Buck Turgidson"
:        "George C. Scott"
,
        "Brig. Gen. Jack D. Ripper"
:  "Sterling Hayden"
,
        `Maj. T.J. "King" Kong`
:      "Slim Pickens"
,
    },

    Oscars: []string
{
        "Best Actor (Nomin.)"
,
        "Best Adapted Screenplay (Nomin.)"
,
        "Best Director (Nomin.)"
,
        "Best Picture (Nomin.)"
,
    },
}
```

Display("strangelove", strangelove)調用將顯示（strangelove電影對應的中文名是《奇愛博士》）：

```
Display strangelove (display.Movie):
strangelove.Title = "Dr. Strangelove"

strangelove.Subtitle = "How I Learned to Stop Worrying and Love the Bomb"

strangelove.Year = 1964

strangelove.Color = false

strangelove.Actor["Gen. Buck Turgidson"
] = "George C. Scott"

strangelove.Actor["Brig. Gen. Jack D. Ripper"
] = "Sterling Hayden"

strangelove.Actor["Maj. T.J. \"King\" Kong"
] = "Slim Pickens"

strangelove.Actor["Dr. Strangelove"
] = "Peter Sellers"

strangelove.Actor["Grp. Capt. Lionel Mandrake"
] = "Peter Sellers"

strangelove.Actor["Pres. Merkin Muffley"
] = "Peter Sellers"

strangelove.Oscars[0
] = "Best Actor (Nomin.)"

strangelove.Oscars[1
] = "Best Adapted Screenplay (Nomin.)"

strangelove.Oscars[2
] = "Best Director (Nomin.)"

strangelove.Oscars[3
] = "Best Picture (Nomin.)"

strangelove.Sequel = nil
```

我們也可以使用Display函數來顯示標準庫中類型的內部結構，例如 `*os.File` 類型：

```
Display("os.Stderr"
, os.Stderr)
// Output:

// Display os.Stderr (*os.File):

// (*(*os.Stderr).file).fd = 2

// (*(*os.Stderr).file).name = "/dev/stderr"

// (*(*os.Stderr).file).nepipe = 0
```

要註意的是，結構體中未導出的成員對反射也是可見的。需要當心的是這個例子的輸出在不同操作繫統上可能是不同的，併且隨着標準庫的發展也可能導致結果不同。（這也是將這些成員定義爲私有成員的原因之一！）我們深圳可以用Display函數來顯示reflect.Value，來查看 `*os.File` 類型的內部表示方式。 `Display("rV", reflect.ValueOf(os.Stderr))` 調用的輸出如下，當然不同環境得到的結果可能有差異：

```
Display rV (reflect.Value):
(*rV.typ).size = 8

(*rV.typ).hash = 871609668

(*rV.typ).align = 8

(*rV.typ).fieldAlign = 8

(*rV.typ).kind = 22

(*(*rV.typ).string
) = "*os.File"



(*(*(*rV.typ).uncommonType).methods[0
].name) = "Chdir"

(*(*(*(*rV.typ).uncommonType).methods[0
].mtyp).string
) = "func() error"

(*(*(*(*rV.typ).uncommonType).methods[0
].typ).string
) = "func(*os.File) error"

...
```

觀察下面兩個例子的區別：

```
var
 i interface
{} = 3


Display("i"
, i)
// Output:

// Display i (int):

// i = 3


Display("&i"
, &i)
// Output:

// Display &i (*interface {}):

// (*&i).type = int

// (*&i).value = 3
```

在第一個例子中，Display函數將調用reflect.ValueOf(i)，它返迴一個Int類型的值。正如我們在12.2節中提到的，reflect.ValueOf總是返迴一個值的具體類型，因爲它是從一個接口值提取的內容。

在第二個例子中，Display函數調用的是reflect.ValueOf(&i)，它返迴一個指向i的指針，對應Ptr類型。在switch的Ptr分支中，通過調用Elem來返迴這個值，返迴一個Value來表示i，對應Interface類型。一個間接獲得的Value，就像這一個，可能代表任意類型的值，包括接口類型。內部的display函數遞歸調用自身，這次它將打印接口的動態類型和值。

目前的實現，Display如果顯示一個帶環的數據結構將會陷入死循環，例如首位項鏈的鏈表：

```
// a struct that points to itself

type
 Cycle struct
{ Value int
; Tail *Cycle }
var
 c Cycle
c = Cycle{42
, &c}
Display("c"
, c)
```

Display會永遠不停地進行深度遞歸打印：

```
Display c (display.Cycle):
c.Value = 42

(*c.Tail).Value = 42

(*(*c.Tail).Tail).Value = 42

(*(*(*c.Tail).Tail).Tail).Value = 42

...ad infinitum...
```

許多Go語言程序都包含了一些循環的數據結果。Display支持這類帶環的數據結構是比較棘手的，需要增加一個額外的記録訪問的路徑；代價是昂貴的。一般的解決方案是采用不安全的語言特性，我們將在13.3節看到具體的解決方案。

帶環的數據結構很少會對fmt.Sprint函數造成問題，因爲它很少嚐試打印完整的數據結構。例如，當它遇到一個指針的時候，它隻是簡單第打印指針的數值。雖然，在打印包含自身的slice或map時可能遇到睏難，但是不保證處理這種是罕見情況卻可以避免額外的麻煩。

練習 12.1： 擴展Displayhans，以便它可以顯示包含以結構體或數組作爲map的key類型的值。

練習 12.2： 增強display函數的穩健性，通過記録邊界的步數來確保在超出一定限製前放棄遞歸。（在13.3節，我們會看到另一種探測數據結構是否存在環的技術。）

12.4. 示例: 編碼S表達式

Display是一個用於顯示結構化數據的調試工具，但是它併不能將任意的Go語言對象編碼爲通用消息然後用於進程間通信。

正如我們在4.5節中中看到的，Go語言的標準庫支持了包括JSON、XML和ASN.1等多種編碼格式。還有另一種依然被廣泛使用的格式是S表達式格式，采用類似Lisp語言的語法。但是和其他編碼格式不同的是，Go語言自帶的標準庫併不支持S表達式，主要是因爲它沒有一個公認的標準規范。

在本節中，我們將定義一個包用於將Go語言的對象編碼爲S表達式格式，它支持以下結構：

```
42          integer
"hello"     string (with Go-style quotation)
foo         symbol (an unquoted name)
(1 2 3)     list   (zero or more items enclosed in parentheses)
```

布爾型習慣上使用t符號表示true，空列表或nil符號表示false，但是爲了簡單起見，我們暫時忽略布爾類型。同時忽略的還有chan管道和函數，因爲通過反射併無法知道它們的確切狀態。我們忽略的還浮點數、複數和interface。支持它們是練習12.3的任務。

我們將Go語言的類型編碼爲S表達式的方法如下。整數和字符串以自然的方式編碼。Nil值編碼爲nil符號。數組和slice被編碼爲一個列表。

結構體被編碼爲成員對象的列表，每個成員對象對應一個個僅有兩個元素的子列表，其中子列表的第一個元素是成員的名字，子列表的第二個元素是成員的值。Map被編碼爲鍵值對的列表。傳統上，S表達式使用點狀符號列表(key . value)結構來表示key/value對，而不是用一個含雙元素的列表，不過爲了簡單我們忽略了點狀符號列表。

編碼是由一個encode遞歸函數完成，如下所示。它的結構本質上和前面的Display函數類似：

```
gopl.io/ch12/sexpr

func
 encode(buf *bytes.Buffer, v reflect.Value) error {
    switch
 v.Kind() {
    case
 reflect.Invalid:
        buf.WriteString("nil"
)

    case
 reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        fmt.Fprintf(buf, "%d"
, v.Int())

    case
 reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        fmt.Fprintf(buf, "%d"
, v.Uint())

    case
 reflect.String:
        fmt.Fprintf(buf, "%q"
```

```
, v.String())

    case
 reflect.Ptr:
        return
 encode(buf, v.Elem())

    case
 reflect.Array, reflect.Slice: // (value ...)

        buf.WriteByte('('
)
        for
 i := 0
; i < v.Len(); i++ {
            if
 i > 0
 {
                buf.WriteByte(' '
)
            }
            if
 err := encode(buf, v.Index(i)); err != nil
 {
                return
 err
            }
        }
        buf.WriteByte(')'
)

    case
 reflect.Struct: // ((name value) ...)

        buf.WriteByte('('
)
        for
 i := 0
; i < v.NumField(); i++ {
            if
 i > 0
 {
                buf.WriteByte(' '
)
            }
            fmt.Fprintf(buf, "(%s "
, v.Type().Field(i).Name)
            if
 err := encode(buf, v.Field(i)); err != nil
 {
                return
 err
            }
```

```
            buf.WriteByte(')'
)
        }
        buf.WriteByte(')'
)

    case
 reflect.Map: // ((key value) ...)

        buf.WriteByte('('
)
        for
 i, key := range
 v.MapKeys() {
            if
 i > 0
 {
                buf.WriteByte(' '
)
            }
            buf.WriteByte('('
)
            if
 err := encode(buf, key); err != nil
 {
                return
 err
            }
            buf.WriteByte(' '
)
            if
 err := encode(buf, v.MapIndex(key)); err != nil
 {
                return
 err
            }
            buf.WriteByte(')'
)
        }
        buf.WriteByte(')'
)

    default
: // float, complex, bool, chan, func, interface

        return
 fmt.Errorf("unsupported type: %s"
, v.Type())
    }
    return
 nil
```

```
}
```

Marshal函數是對encode的保證，以保持和encoding/...下其它包有着相似的API：

```
// Marshal encodes a Go value in S-expression form.

func
 Marshal(v interface
{}) ([]byte
, error) {
    var
 buf bytes.Buffer
    if
 err := encode(&buf, reflect.ValueOf(v)); err != nil
 {
        return
 nil
, err
    }
    return
 buf.Bytes(), nil

}
```

下面是Marshal對12.3節的strangelove變量編碼後的結果：

```
((Title "Dr. Strangelove") (Subtitle "How I Learned to Stop Worrying and Lo
ve the Bomb") (Year 1964) (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sell
ers") ("Pres. Merkin Muffley" "Peter Sellers") ("Gen. Buck Turgidson" "Geor
ge C. Scott") ("Brig. Gen. Jack D. Ripper" "Sterling Hayden") ("Maj. T.J. \
"King\" Kong" "Slim Pickens") ("Dr. Strangelove" "Peter Sellers"))) (Oscars
("Best Actor (Nomin.)" "Best Adapted Screenplay (Nomin.)" "Best Director (N
omin.)" "Best Picture (Nomin.)")) (Sequel nil))
```

整個輸出編碼爲一行中以減少輸出的大小，但是也很難閱讀。這里有一個對S表達式格式化的約定。編寫一個S表達式的格式化函數將作爲一個具有挑戰性的練習任務；不過 http://gopl.io 也提供了一個簡單的版本。

```
((Title "Dr. Strangelove")
 (Subtitle "How I Learned to Stop Worrying and Love the Bomb")
 (Year 1964)
 (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sellers")
         ("Pres. Merkin Muffley" "Peter Sellers")
         ("Gen. Buck Turgidson" "George C. Scott")
         ("Brig. Gen. Jack D. Ripper" "Sterling Hayden")
         ("Maj. T.J. \"King\" Kong" "Slim Pickens")
         ("Dr. Strangelove" "Peter Sellers")))
 (Oscars ("Best Actor (Nomin.)"
          "Best Adapted Screenplay (Nomin.)"
          "Best Director (Nomin.)"
          "Best Picture (Nomin.)"))
 (Sequel nil))
```

和fmt.Print、json.Marshal、Display函數類似，sexpr.Marshal函數處理帶環的數據結構也會陷入死循環。

在12.6節中，我們將給出S表達式解碼器的實現步驟，但是在那之前，我們還需要先了解如果通過反射技術來更新程序的變量。

練習 12.3： 實現encode函數缺少的分支。將布爾類型編碼爲t和nil，浮點數編碼爲Go語言的格式，複數1+2i編碼爲#C(1.0 2.0)格式。接口編碼爲類型名和值對，例如("[]int" (1 2 3))，但是這個形式可能會造成歧義：reflect.Type.String方法對於不同的類型可能返迴相同的結果。

練習 12.4： 脩改encode函數，以上面的格式化形式輸出S表達式。

練習 12.5： 脩改encode函數，用JSON格式代替S表達式格式。然後使用標準庫提供的json.Unmarshal解碼器來驗證函數是正確的。

練習 12.6： 脩改encode，作爲一個優化，忽略對是零值對象的編碼。

練習 12.7： 創建一個基於流式的API，用於S表達式的解碼，和json.Decoder(§4.5)函數功能類似。

12.5. 通過reflect.Value脩改值

到目前爲止，反射還隻是程序中變量的另一種訪問方式。然而，在本節中我們將重點討論如果通過反射機製來脩改變量。

迴想一下，Go語言中類似x、x.f[1]和*p形式的表達式都可以表示變量，但是其它如x + 1和f(2)則不是變量。一個變量就是一個可尋址的內存空間，里面存儲了一個值，併且存儲的值可以通過內存地址來更新。

對於reflect.Values也有類似的區別。有一些reflect.Values是可取地址的；其它一些則不可以。考慮以下的聲明語句：

```
x := 2
                   // value   type    variable?

a := reflect.ValueOf(2
)  // 2       int     no

b := reflect.ValueOf(x)  // 2       int     no

c := reflect.ValueOf(&x) // &x      *int    no

d := c.Elem()            // 2       int     yes (x)
```

其中a對應的變量則不可取地址。因爲a中的值僅僅是整數2的拷貝副本。b中的值也同樣不可取地址。c中的值還是不可取地址，它隻是一個指針 `&x` 的拷貝。實際上，所有通過reflect.ValueOf(x)返迴的reflect.Value都是不可取地址的。但是對於d，它是c的解引用方式生成的，指向另一個變量，因此是可取地址的。我們可以通過調用reflect.ValueOf(&x).Elem()，來獲取任意變量x對應的可取地址的Value。

我們可以通過調用reflect.Value的CanAddr方法來判斷其是否可以被取地址：

```
fmt.Println(a.CanAddr()) // "false"

fmt.Println(b.CanAddr()) // "false"

fmt.Println(c.CanAddr()) // "false"

fmt.Println(d.CanAddr()) // "true"
```

每當我們通過指針間接地獲取的reflect.Value都是可取地址的，即使開始的是一個不可取地址的Value。在反射機製中，所有關於是否支持取地址的規則都是類似的。例如，slice的索引表達式e[i]將隱式地包含一個指針，它就是可取地址的，即使開始的e表達式不支持也沒有關繫。以此類推，reflect.ValueOf(e).Index(i)對於的值也是可取地址的，即使原始的reflect.ValueOf(e)不支持也沒有關繫。

要從變量對應的可取地址的reflect.Value來訪問變量需要三個步驟。第一步是調用Addr()方法，它返迴一個Value，里面保存了指向變量的指針。然後是在Value上調用Interface()方法，也就是返迴一個interface{}，里面通用包含指向變量的指針。最後，如果我們知道變量的類型，我們可以使用類型的斷言機製將得到的interface{}類型的接口強製環爲普通的類型指針。這樣我們就可以通過這個普通指針來更新變量了：

```
x := 2

d := reflect.ValueOf(&x).Elem()   // d refers to the variable x

px := d.Addr().Interface().(*int
) // px := &x

*px = 3
                            // x = 3

fmt.Println(x)                    // "3"
```

或者，不使用指針，而是通過調用可取地址的reflect.Value的reflect.Value.Set方法來更新對於的值：

```
d.Set(reflect.ValueOf(4
))
fmt.Println(x) // "4"
```

Set方法將在運行時執行和編譯時類似的可賦值性約束的檢查。以上代碼，變量和值都是int類型，但是如果變量是int64類型，那麼程序將拋出一個panic異常，所以關鍵問題是要確保改類型的變量可以接受對應的值：

```
d.Set(reflect.ValueOf(int64
(5
))) // panic: int64 is not assignable to int
```

通用對一個不可取地址的reflect.Value調用Set方法也會導致panic異常：

```
x := 2

b := reflect.ValueOf(x)
b.Set(reflect.ValueOf(3
)) // panic: Set using unaddressable value
```

這里有很多用於基本數據類型的Set方法：SetInt、SetUint、SetString和SetFloat等。

```
d := reflect.ValueOf(&x).Elem()
d.SetInt(3
)
fmt.Println(x) // "3"
```

從某種程度上説，這些Set方法總是盡可能地完成任務。以SetInt爲例，隻要變量是某種類型的有符號整數就可以工作，即使是一些命名的類型，隻要底層數據類型是有符號整數就可以，而且如果對於變量類型值太大的話會被自動截斷。但需要謹慎的是：對於一個引用interface{}類型的reflect.Value調用SetInt會導致panic異常，即使那個interface{}變量對於整數類型也不行。

```
x := 1

rx := reflect.ValueOf(&x).Elem()
rx.SetInt(2
)                     // OK, x = 2

rx.Set(reflect.ValueOf(3
))       // OK, x = 3

rx.SetString("hello"
)            // panic: string is not assignable to int

rx.Set(reflect.ValueOf("hello"
)) // panic: string is not assignable to int


var
 y interface
{}
ry := reflect.ValueOf(&y).Elem()
ry.SetInt(2
)                     // panic: SetInt called on interface Value

ry.Set(reflect.ValueOf(3
))       // OK, y = int(3)

ry.SetString("hello"
)            // panic: SetString called on interface Value

ry.Set(reflect.ValueOf("hello"
)) // OK, y = "hello"
```

當我們用Display顯示os.Stdout結構時，我們發現反射可以越過Go語言的導出規則的限製讀取結構體中未導出的成員，比如在類Unix繫統上os.File結構體中的fd int成員。然而，利用反射機製併不能脩改這些未導出的成員：

```
stdout := reflect.ValueOf(os.Stdout).Elem() // *os.Stdout, an os.File var

fmt.Println(stdout.Type())                  // "os.File"

fd := stdout.FieldByName("fd"
)
fmt.Println(fd.Int()) // "1"

fd.SetInt(2
)          // panic: unexported field
```

一個可取地址的reflect.Value會記錄一個結構體成員是否是未導出成員，如果是的話則拒絕脩改操作。因此，CanAddr方法併不能正確反映一個變量是否是可以被脩改的。另一個相關的方法CanSet是用於檢查對應的reflect.Value是否是可取地址併可被脩改的：

```
fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"
```

12.6. 示例: 解碼S表達式

標準庫中encoding/...下每個包中提供的Marshal編碼函數都有一個對應的Unmarshal函數用於解碼。例如，我們在4.5節中看到的，要將包含JSON編碼格式的字節slice數據解碼爲我們自己的Movie類型（§12.3），我們可以這樣做：

```
data := []byte
{/* ... */
}
var
 movie Movie
err := json.Unmarshal(data, &movie)
```

Unmarshal函數使用了反射機製類脩改movie變量的每個成員，根據輸入的內容爲Movie成員創建對應的map、結構體和slice。

現在讓我們爲S表達式編碼實現一個簡易的Unmarshal，類似於前面的json.Unmarshal標準庫函數，對應我們之前實現的sexpr.Marshal函數的逆操作。我們必鬚提醒一下，一個健壯的和通用的實現通常需要比例子更多的代碼，爲了便於演示我們采用了精簡的實現。我們隻支持S表達式有限的子集，同時處理錯誤的方式也比較粗暴，代碼的目的是爲了演示反射的用法，而不是構造一個實用的S表達式的解碼器。

詞法分析器lexer使用了標準庫中的text/scanner包將輸入流的字節數據解析爲一個個類似註釋、標識符、字符串面值和數字面值之類的標記。輸入掃描器scanner的Scan方法將提前掃描和返迴下一個記號，對於rune類型。大多數記號，比如“(”，對應一個單一rune可表示的Unicode字符，但是text/scanner也可以用小的負數表示記號標識符、字符串等由多個字符組成的記號。調用Scan方法將返迴這些記號的類型，接着調用TokenText方法將返迴記號對應的文本內容。

因爲每個解析器可能需要多次使用當前的記號，但是Scan會一直向前掃描，所有我們包裝了一個lexer掃描器輔助類型，用於跟蹤最近由Scan方法返迴的記號。

```
gopl.io/ch12/sexpr

type
 lexer struct
 {
    scan  scanner.Scanner
    token rune
 // the current token

}

func
 (lex *lexer) next()        { lex.token = lex.scan.Scan() }
func
 (lex *lexer) text() string
 { return
 lex.scan.TokenText() }

func
 (lex *lexer) consume(want rune
) {
    if
 lex.token != want { // NOTE:
 Not an example of good error handling.

        panic
(fmt.Sprintf("got %q, want %q"
, lex.text(), want))
    }
    lex.next()
}
```

現在讓我們轉到語法解析器。它主要包含兩個功能。第一個是read函數，用於讀取S表達式的當前標記，然後根據S表達式的當前標記更新可取地址的reflect.Value對應的變量v。

```
func
 read(lex *lexer, v reflect.Value) {
    switch
 lex.token {
    case
 scanner.Ident:
        // The only valid identifiers are

        // "nil" and struct field names.

        if
 lex.text() == "nil"
 {
            v.Set(reflect.Zero(v.Type()))
            lex.next()
```

```
            lex.next()
            return

        }
    case
 scanner.String:
        s, _ := strconv.Unquote(lex.text()) // NOTE:
 ignoring errors

        v.SetString(s)
        lex.next()
        return

    case
 scanner.Int:
        i, _ := strconv.Atoi(lex.text()) // NOTE:
 ignoring errors

        v.SetInt(int64
(i))
        lex.next()
        return

    case
 '('
:
        lex.next()
        readList(lex, v)
        lex.next() // consume ')'

        return

    }
    panic
(fmt.Sprintf("unexpected token %q"
, lex.text()))
}
```

我們的S表達式使用標識符區分兩個不同類型，結構體成員名和nil值的指針。read函數值處理nil類型的標識符。當遇到scanner.Ident爲“nil”是，使用reflect.Zero函數將變量v設置爲零值。而其它任何類型的標識符，我們都作爲錯誤處理。後面的readList函數將處理結構體的成員名。

一個“(”標記對應一個列表的開始。第二個函數readList，將一個列表解碼到一個聚合類型中（map、結構體、slice或數組），具體類型依然於傳入待填充變量的類型。每次遇到這種情況，循環繼續解析每個元素直到遇到於開始標記匹配的結束標記“)”，endList函數用於檢測結束標記。

最有趣的部分是遞歸。最簡單的是對數組類型的處理。直到遇到“)”結束標記，我們使用Index函數來獲取數組每個元素的地址，然後遞歸調用read函數處理。和其它錯誤類似，如果輸入數據導致解碼器的引用超出了數組的范圍，解碼器將拋出panic異常。slice也采用類似方法解析，不同的是我們將爲每個元素創建新的變量，然後將元素添加到slice的末尾。

在循環處理結構體和map每個元素時必鬚解碼一個(key value)格式的對應子列表。對於結構體，key部分對於成員的名字。和數組類似，我們使用FieldByName找到結構體對應成員的變量，然後遞歸調用read函數處理。對於map，key可能是任意類型，對元素的處理方式和slice類似，我們創建一個新的變量，然後遞歸填充它，最後將新解析到的key/value對添加到map。

```
func
 readList(lex *lexer, v reflect.Value) {
    switch
 v.Kind() {
    case
 reflect.Array: // (item ...)

        for
 i := 0
; !endList(lex); i++ {
            read(lex, v.Index(i))
        }

    case
 reflect.Slice: // (item ...)

        for
 !endList(lex) {
            item := reflect.New(v.Type().Elem()).Elem()
            read(lex, item)
            v.Set(reflect.Append(v, item))
        }

    case
 reflect.Struct: // ((name value) ...)

        for
 !endList(lex) {
            lex.consume('('
)
            if
 lex.token != scanner.Ident {
                panic
(fmt.Sprintf("got token %q, want field name"
, lex.text()))
            }
            name := lex.text()
            lex.next()
            read(lex, v.FieldByName(name))
            lex.consume(')'
)
        }

    case
 reflect.Map: // ((key value) ...)

        v.Set(reflect.MakeMap(v.Type()))
        for
 !endList(lex) {
            lex.consume('('
)
```

```
            key := reflect.New(v.Type().Key()).Elem()
            read(lex, key)
            value := reflect.New(v.Type().Elem()).Elem()
            read(lex, value)
            v.SetMapIndex(key, value)
            lex.consume(')'
)
        }

    default
:
        panic
(fmt.Sprintf("cannot decode list into %v"
, v.Type()))
    }
}

func
 endList(lex *lexer) bool
 {
    switch
 lex.token {
    case
 scanner.EOF:
        panic
("end of file"
)
    case
 ')'
:
        return
 true

    }
    return
 false

}
```

最後，我們將解析器包裝爲導出的Unmarshal解碼函數，隱藏了一些初始化和清理等邊緣處理。內部解析器以panic的方式拋出錯誤，但是Unmarshal函數通過在defer語句調用recover函數來捕獲內部panic（§5.10），然後返迴一個對panic對應的錯誤信息。

```
// Unmarshal parses S-expression data and populates the variable

// whose address is in the non-nil pointer out.

func
 Unmarshal(data []byte
, out interface
{}) (err error) {
    lex := &lexer{scan: scanner.Scanner{Mode: scanner.GoTokens}}
    lex.scan.Init(bytes.NewReader(data))
    lex.next() // get the first token

    defer
 func
() {
        // NOTE:
 this is not an example of ideal error handling.

        if
 x := recover
(); x != nil
 {
            err = fmt.Errorf("error at %s: %v"
, lex.scan.Position, x)
        }
    }()
    read(lex, reflect.ValueOf(out).Elem())
    return
 nil

}
```

生產實現不應該對任何輸入問題都用panic形式報告，而且應該報告一些錯誤相關的信息，例如出現錯誤輸入的行號和位置等。盡管如此，我們希望通過這個例子來展示類似encoding/json等包底層代碼的實現思路，以及如何使用反射機製來填充數據結構。

練習 12.8： sexpr.Unmarshal函數和json.Unmarshal一樣，都要求在解碼前輸入完整的字節slice。定義一個和json.Decoder類似的sexpr.Decoder類型，支持從一個io.Reader流解碼。脩改sexpr.Unmarshal函數，使用這個新的類型實現。

練習 12.9： 編寫一個基於標記的API用於解碼S表達式，參考xml.Decoder（7.14）的風格。你將需要五種類型的標記：Symbol、String、Int、StartList和EndList。

練習 12.10： 擴展sexpr.Unmarshal函數，支持布爾型、浮點數和interface類型的解碼，使用 **練習 12.3：** 的方案。（提示：要解碼接口，你需要將name映射到每個支持類型的reflect.Type。）

12.7. 獲取結構體字段標識

在4.5節我們使用構體成員標籤用於設置對應JSON對應的名字。其中json成員標籤讓我們可以選擇成員的名字和抑製零值成員的輸出。在本節，我們將看到如果通過反射機製類獲取成員標籤。

對於一個web服務，大部分HTTP處理函數要做的第一件事情就是展開請求中的參數到本地變量中。我們定義了一個工具函數，叫params.Unpack，通過使用結構體成員標籤機製來讓HTTP處理函數解析請求參數更方便。

首先，我們看看如何使用它。下面的search函數是一個HTTP請求處理函數。它定義了一個匿名結構體類型的變量，用結構體的每個成員表示HTTP請求的參數。其中結構體成員標籤指明了對於請求參數的名字，爲了減少UTRL的長度這些參數名通常都是神祕的縮略詞。Unpack將請求參數填充到合適的結構體成員中，這樣我們可以方便地通過合適的類型類來訪問這些參數。

```
gopl.io/ch12/search

import
 "gopl.io/ch12/params"


// search implements the /search URL endpoint.

func
 search(resp http.ResponseWriter, req *http.Request) {
    var
 data struct
 {
        Labels     []string
 `http:"l"`

        MaxResults int
      `http:"max"`

        Exact      bool
     `http:"x"`

    }
    data.MaxResults = 10
 // set default

    if
 err := params.Unpack(req, &data); err != nil
 {
        http.Error(resp, err.Error(), http.StatusBadRequest) // 400

        return

    }

    // ...rest of handler...

    fmt.Fprintf(resp, "Search: %+v\n"
, data)
}
```

下面的Unpack函數主要完成三件事情。第一，它調用req.ParseForm()來解析HTTP請求。然後，req.Form將包含所有的請求參數，不管HTTP客戶端使用的是GET還是POST請求方法。

下一步，Unpack函數將構建每個結構體成員有效參數名字到成員變量的映射。如果結構體成員有成員標籤的話，有效參數名字可能和實際的成員名字不相同。reflect.Type的Field方法將返迴一個reflect.StructField，里面含有每個成員的名字、類型和可選的成員標籤等信息。其中成員標籤信息對應reflect.StructTag類型的字符串，併且提供了Get方法用於解析和根據特定key提取的子串，例如這里的http:"..."形式的子串。

```
gopl.io/ch12/params
```

```
// Unpack populates the fields of the struct pointed to by ptr

// from the HTTP request parameters in req.

func
 Unpack(req *http.Request, ptr interface
{}) error {
    if
 err := req.ParseForm(); err != nil
 {
        return
 err
    }

    // Build map of fields keyed by effective name.

    fields := make
(map
[string
]reflect.Value)
    v := reflect.ValueOf(ptr).Elem() // the struct variable

    for
 i := 0
; i < v.NumField(); i++ {
        fieldInfo := v.Type().Field(i) // a reflect.StructField

        tag := fieldInfo.Tag           // a reflect.StructTag

        name := tag.Get("http"
)
        if
 name == ""
 {
            name = strings.ToLower(fieldInfo.Name)
        }
        fields[name] = v.Field(i)
    }

    // Update struct field for each parameter in the request.

    for
 name, values := range
 req.Form {
        f := fields[name]
        if
 !f.IsValid() {
            continue
 // ignore unrecognized HTTP parameters

        }
        for
```

```
 _, value := range
 values {
            if
 f.Kind() == reflect.Slice {
                elem := reflect.New(f.Type().Elem()).Elem()
                if
 err := populate(elem, value); err != nil
 {
                    return
 fmt.Errorf("%s: %v"
, name, err)
                }
                f.Set(reflect.Append(f, elem))
            } else
 {
                if
 err := populate(f, value); err != nil
 {
                    return
 fmt.Errorf("%s: %v"
, name, err)
                }
            }
        }
    }
    return
 nil

}
```

最後，Unpack遍歷HTTP請求的name/valu參數鍵值對，併且根據更新相應的結構體成員。迴想一下，同一個名字的參數可能出現多次。如果發生這種情況，併且對應的結構體成員是一個slice，那麼就將所有的參數添加到slice中。其它情況，對應的成員值將被覆蓋，隻有最後一次出現的參數值才是起作用的。

populate函數小心用請求的字符串類型參數值來填充單一的成員v（或者是slice類型成員中的單一的元素）。目前，它僅支持字符串、有符號整數和布爾型。其中其它的類型將留做練習任務。

```
func
 populate(v reflect.Value, value string
) error {
    switch
 v.Kind() {
    case
 reflect.String:
        v.SetString(value)

    case
 reflect.Int:
        i, err := strconv.ParseInt(value, 10
, 64
)
        if
 err != nil
 {
            return
 err
        }
        v.SetInt(i)

    case
 reflect.Bool:
        b, err := strconv.ParseBool(value)
        if
 err != nil
 {
            return
 err
        }
        v.SetBool(b)

    default
:
        return
 fmt.Errorf("unsupported kind %s"
, v.Type())
    }
    return
 nil

}
```

如果我們上上面的處理程序添加到一個web服務器，則可以產生以下的會話：

```
$ go build gopl.io/ch12/search
$ ./search &
$ ./fetch 'http://localhost:12345/search'
Search: {Labels:[] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming&max=100'
Search: {Labels:[golang programming] MaxResults:100 Exact:false}
$ ./fetch 'http://localhost:12345/search?x=true&l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:true}
$ ./fetch 'http://localhost:12345/search?q=hello&x=123'
x: strconv.ParseBool: parsing "123": invalid syntax
$ ./fetch 'http://localhost:12345/search?q=hello&max=lots'
max: strconv.ParseInt: parsing "lots": invalid syntax
```

練習 12.11： 編寫相應的Pack函數，給定一個結構體值，Pack函數將返迴合併了所有結構體成員和值的URL。

練習 12.12： 擴展成員標籤以表示一個請求參數的有效值規則。例如，一個字符串可以是有效的email地址或一個信用卡號碼，還有一個整數可能需要是有效的郵政編碼。脩改Unpack函數以檢查這些規則。

練習 12.13： 脩改S表達式的編碼器（§12.4）和解碼器（§12.6），采用和encoding/json包（§4.5）類似的方式使用成員標籤中的sexpr:"..."字串。

12.8. 顯示一個類型的方法集

我們的最後一個例子是使用reflect.Type來打印任意值的類型和枚舉它的方法：

```
gopl.io/ch12/methods


// Print prints the method set of the value x.


func
 Print(x interface
{}) {
    v := reflect.ValueOf(x)
    t := v.Type()
    fmt.Printf("type %s\n"
, t)


    for
 i := 0
; i < v.NumMethod(); i++ {
        methType := v.Method(i).Type()
        fmt.Printf("func (%s) %s%s\n"
, t, t.Method(i).Name,
            strings.TrimPrefix(methType.String(), "func"
))
    }
}
```

reflect.Type和reflect.Value都提供了一個Method方法。每次t.Method(i)調用將一個reflect.Method的實例，對應一個用於描述一個方法的名稱和類型的結構體。每次v.Method(i)方法調用都返迴一個reflect.Value以表示對應的值（§6.4），也就是一個方法是幫到它的接收者的。使用reflect.Value.Call方法（我們之類沒有演示），將可以調用一個Func類型的Value，但是這個例子中隻用到了它的類型。

這是屬於time.Duration和 `*strings.Replacer` 兩個類型的方法：

```
methods.Print(time.Hour)
// Output:

// type time.Duration

// func (time.Duration) Hours() float64

// func (time.Duration) Minutes() float64

// func (time.Duration) Nanoseconds() int64

// func (time.Duration) Seconds() float64

// func (time.Duration) String() string


methods.Print(new
(strings.Replacer))
// Output:

// type *strings.Replacer

// func (*strings.Replacer) Replace(string) string

// func (*strings.Replacer) WriteString(io.Writer, string) (int, error)

`
```

12.9. 幾點忠告

雖然反射提供的API遠多於我們講到的，我們前面的例子主要是給出了一個方向，通過反射可以實現哪些功能。反射是一個強大併富有表達力的工具，但是它應該被小心地使用，原因有三。

第一個原因是，基於反射的代碼是比較脆弱的。對於每一個會導致編譯器報告類型錯誤的問題，在反射中都有與之相對應的問題，不同的是編譯器會在構建時馬上報告錯誤，而反射則是在眞正運行到的時候才會拋出panic異常，可能是寫完代碼很久之後的時候了，而且程序也可能運行了很長的時間。

以前面的readList函數（§12.6）爲例，爲了從輸入讀取字符串併填充int類型的變量而調用的reflect.Value.SetString方法可能導致panic異常。絶大多數使用反射的程序都有類似的風險，需要非常小心地檢查每個reflect.Value的對於值的類型、是否可取地址，還有是否可以被脩改等。

避免這種因反射而導致的脆弱性的問題的最好方法是將所有的反射相關的使用控製在包的內部，如果可能的話避免在包的API中直接暴露reflect.Value類型，這樣可以限製一些非法輸入。如果無法做到這一點，在每個有風險的操作前指向額外的類型檢查。以標準庫中的代碼爲例，當fmt.Printf收到一個非法的操作數是，它併不會拋出panic異常，而是打印相關的錯誤信息。程序雖然還有BUG，但是會更加容易診斷。

```
fmt.Printf("%d %s\n"
, "hello"
, 42
) // "%!d(string=hello) %!s(int=42)"
```

反射同樣降低了程序的安全性，還影響了自動化重構和分析工具的準確性，因爲它們無法識別運行時才能確認的類型信息。

避免使用反射的第二個原因是，即使對應類型提供了相同文檔，但是反射的操作不能做靜態類型檢查，而且大量反射的代碼通常難以理解。總是需要小心翼翼地爲每個導出的類型和其它接受interface{}或reflect.Value類型參數的函數維護説明文檔。

第三個原因，基於反射的代碼通常比正常的代碼運行速度慢一到兩個數量級。對於一個典型的項目，大部分函數的性能和程序的整體性能關繫不大，所以使用反射可能會使程序更加清晰。測試是一個特別適合使用反射的場景，因爲每個測試的數據集都很小。但是對於性能關鍵路徑的函數，最好避免使用反射。

第13章 底層編程

Go語言的設計包含了諸多安全策略，限製了可能導致程序運行出現錯誤的用法。編譯時類型檢查檢查可以發現大多數類型不匹配的操作，例如兩個字符串做減法的錯誤。字符串、map、slice和chan等所有的內置類型，都有嚴格的類型轉換規則。

對於無法靜態檢測到的錯誤，例如數組訪問越界或使用空指針，運行時動態檢測可以保證程序在遇到問題的時候立即終止併打印相關的錯誤信息。自動內存管理（垃圾內存自動迴收）可以消除大部分野指針和內存洩漏相關的問題。

Go語言的實現刻意隱藏了很多底層細節。我們無法知道一個結構體眞實的內存布局，也無法獲取一個運行時函數對應的機器碼，也無法知道當前的goroutine是運行在哪個操作繫統線程之上。事實上，Go語言的調度器會自己決定是否需要將某個goroutine從一個操作繫統線程轉移到另一個操作繫統線程。一個指向變量的指針也併沒有展示變量眞實的地址。因爲垃圾迴收器可能會根據需要移動變量的內存位置，當然變量對應的地址也會被自動更新。

總的來說，Go語言的這些特性使得Go程序相比較低級的C語言來說更容易預測和理解，程序也不容易崩潰。通過隱藏底層的實現細節，也使得Go語言編寫的程序具有高度的可移植性，因爲語言的語義在很大程度上是獨立於任何編譯器實現、操作繫統和CPU繫統結構的（當然也不是完全絶對獨立：例如int等類型就依賴於CPU機器字的大小，某些表達式求值的具體順序，還有編譯器實現的一些額外的限製等）。

有時候我們可能會放棄使用部分語言特性而優先選擇更好具有更好性能的方法，例如需要與其他語言編寫的庫互操作，或者用純Go語言無法實現的某些函數。

在本章，我們將展示如何使用unsafe包來襬脫Go語言規則帶來的限製，講述如何創建C語言函數庫的綁定，以及如何進行繫統調用。

本章提供的方法不應該輕易使用（譯註：屬於黑魔法，雖然可能功能很強大，但是也容易誤傷到自己）。如果沒有處理好細節，它們可能導致各種不可預測的併且隱晦的錯誤，甚至連有經驗的的C語言程序員也無法理解這些錯誤。使用unsafe包的同時也放棄了Go語言保證與未來版本的兼容性的承諾，因爲它必然會在有意無意中會使用很多實現的細節，而這些實現的細節在未來的Go語言中很可能會被改變。

要註意的是，unsafe包是一個采用特殊方式實現的包。雖然它可以和普通包一樣的導入和使用，但它實際上是由編譯器實現的。它提供了一些訪問語言內部特性的方法，特別是內存布局相關的細節。將這些特性封裝到一個獨立的包中，是爲在極少數情況下需要使用的時候，同時引起人們的註意（譯註：因爲看包的名字就知道使用unsafe包是不安全的）。此外，有一些環境因爲安全的因素可能限製這個包的使用。

不過unsafe包被廣泛地用於比較低級的包, 例如runtime、os、syscall還有net包等，因爲它們需要和操作繫統密切配合，但是對於普通的程序一般是不需要使用unsafe包的。

13.1. unsafe.Sizeof, Alignof 和 Offsetof

unsafe.Sizeof函數返迴操作數在內存中的字節大小，參數可以是任意類型的表達式，但是它併不會對表達式進行求值。一個Sizeof函數調用是一個對應uintptr類型的常量表達式，因此返迴的結果可以用作數組類型的長度大小，或者用作計算其他的常量。

```
import
 "unsafe"

fmt.Println(unsafe.Sizeof(float64
(0
))) // "8"
```

Sizeof函數返迴的大小隻包括數據結構中固定的部分，例如字符串對應結構體中的指針和字符串長度部分，但是併不包含指針指向的字符串的內容。Go語言中非聚合類型通常有一個固定的大小，盡管在不同工具鏈下生成的實際大小可能會有所不同。考慮到可移植性，引用類型或包含引用類型的大小在32位平台上是4個字節，在64位平台上是8個字節。

計算機在加載和保存數據時，如果內存地址合理地對齊的將會更有效率。例如2字節大小的int16類型的變量地址應該是偶數，一個4字節大小的rune類型變量的地址應該是4的倍數，一個8字節大小的float64、uint64或64-bit指針類型變量的地址應該是8字節對齊的。但是對於再大的地址對齊倍數則是不需要的，即使是complex128等較大的數據類型最多也隻是8字節對齊。

由於地址對齊這個因素，一個聚合類型（結構體或數組）的大小至少是所有字段或元素大小的總和，或者更大因爲可能存在內存空洞。內存空洞是編譯器自動添加的沒有被使用的內存空間，用於保證後面每個字段或元素的地址相對於結構或數組的開始地址能夠合理地對齊（譯註：內存空洞可能會存在一些隨機數據，可能會對用unsafe包直接操作內存的處理産生影響）。

類型	大小
bool	1個字節
intN, uintN, floatN, complexN	N/8個字節(例如float64是8個字節)
int, uint, uintptr	1個機器字
*T	1個機器字
string	2個機器字(data,len)
[]T	3個機器字(data,len,cap)
map	1個機器字
func	1個機器字
chan	1個機器字
interface	2個機器字(type,value)

Go語言的規范併沒有要求一個字段的聲明順序和內存中的順序是一致的，所以理論上一個編譯器可以隨意地重新排列每個字段的內存位置，隨然在寫作本書的時候編譯器還沒有這麼做。下面的三個結構體雖然有着相同的字段，但是第一種寫法比另外的兩個需要多50%的內存。

```
                            // 64-bit  32-bit

struct
{ bool
; float64
; int16
 } // 3 words 4words

struct
{ float64
; int16
; bool
 } // 2 words 3words

struct
{ bool
; int16
; float64
 } // 2 words 3words
```

關於內存地址對齊算法的細節超出了本書的范圍，也不是每一個結構體都需要擔心這個問題，不過有效的包裝可以使數據結構更加緊湊（譯註：未來的Go語言編譯器應該會默認優化結構體的順序，當然用於應該也能夠指定具體的內存布局，相同討論請參考 Issue10014 ），內存使用率和性能都可能會受益。

`unsafe.Alignof` 函數返迴對應參數的類型需要對齊的倍數. 和 Sizeof 類似, Alignof 也是返迴一個常量表達式, 對應一個常量. 通常情況下布爾和數字類型需要對齊到它們本身的大小(最多8個字節), 其它的類型對齊到機器字大小.

`unsafe.Offsetof` 函數的參數必鬚是一個字段 `x.f` , 然後返迴 `f` 字段相對於 `x` 起始地址的偏移量, 包括可能的空洞.

圖 13.1 顯示了一個結構體變量 x 以及其在32位和64位機器上的典型的內存. 灰色區域是空洞.

```
var
 x struct
 {
    a bool

    b int16

    c []int

 }
```

下面顯示了對x和它的三個字段調用unsafe包相關函數的計算結果：

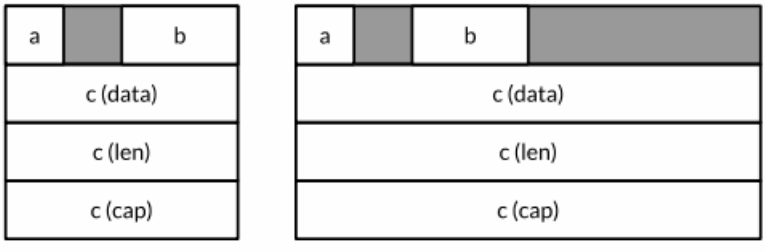


Figure 13.1. Holes in a struct.

32位繫統：

```
Sizeof(x)   = 16  Alignof(x)   = 4
Sizeof(x.a) = 1   Alignof(x.a) = 1 Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2 Offsetof(x.b) = 2
Sizeof(x.c) = 12  Alignof(x.c) = 4 Offsetof(x.c) = 4
```

64位繫統：

```
Sizeof(x)   = 32  Alignof(x)   = 8
Sizeof(x.a) = 1   Alignof(x.a) = 1 Offsetof(x.a) = 0
Sizeof(x.b) = 2   Alignof(x.b) = 2 Offsetof(x.b) = 2
Sizeof(x.c) = 24  Alignof(x.c) = 8 Offsetof(x.c) = 8
```

雖然這幾個函數在不安全的unsafe包，但是這幾個函數調用併不是眞的不安全，特別在需要優化內存空間時它們返迴的結果對於理解原生的內存布局很有幫助。

13.2. unsafe.Pointer

大多數指針類型會寫成 `*T` ，表示是“一個指向T類型變量的指針”。unsafe.Pointer是特別定義的一種指針類型（譯註：類似C語言中的 `void*` 類型的指針），它可以包含任意類型變量的地址。當然，我們不可以直接通過 `*p` 來獲取unsafe.Pointer指針指向的眞實變量的值，因爲我們併不知道變量的具體類型。和普通指針一樣，unsafe.Pointer指針也是可以比較的，併且支持和nil常量比較判斷是否爲空指針。

一個普通的 `*T` 類型指針可以被轉化爲unsafe.Pointer類型指針，併且一個unsafe.Pointer類型指針也可以被轉迴普通的指針，被轉迴普通的指針類型併不需要和原始的 `*T` 類型相同。通過將 `*float64` 類型指針轉化爲 `*uint64` 類型指針，我們可以查看一個浮點數變量的位模式。

```
package
 math

func
 Float64bits(f float64
) uint64
 { return
 *(*uint64
)(unsafe.Pointer(&f)) }

fmt.Printf("%#016x\n"
, Float64bits(1.0
)) // "0x3ff0000000000000"
```

通過轉爲新類型指針，我們可以更新浮點數的位模式。通過位模式操作浮點數是可以的，但是更重要的意義是指針轉換語法讓我們可以在不破壞類型繫統的前提下向內存寫入任意的值。

一個unsafe.Pointer指針也可以被轉化爲uintptr類型，然後保存到指針型數值變量中（譯註：這隻是和當前指針相同的一個數字值，併不是一個指針），然後用以做必要的指針數值運算。（第三章內容，uintptr是一個無符號的整型數，足以保存一個地址）這種轉換雖然也是可逆的，但是將uintptr轉爲unsafe.Pointer指針可能會破壞類型繫統，因爲併不是所有的數字都是有效的內存地址。

許多將unsafe.Pointer指針轉爲原生數字，然後再轉迴爲unsafe.Pointer類型指針的操作也是不安全的。比如下面的例子需要將變量x的地址加上b字段地址偏移量轉化爲 `*int16` 類型指針，然後通過該指針更新x.b：

```
//gopl.io/ch13/unsafeptr

var
 x struct
 {
    a bool

    b int16

    c []int

}

// 和 pb := &x.b 等價

pb := (*int16
)(unsafe.Pointer(
    uintptr
(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
*pb = 42

fmt.Println(x.b) // "42"
```

上面的寫法盡管很繁瑣，但在這里併不是一件壞事，因爲這些功能應該很謹慎地使用。不要試圖引入一個uintptr類型的臨時變量，因爲它可能會破壞代碼的安全性（譯註：這是眞正可以體會unsafe包爲何不安全的例子）。下面段代碼是錯誤的：

```
// NOTE:
 subtly incorrect!

tmp := uintptr
(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
pb := (*int16
)(unsafe.Pointer(tmp))
*pb = 42
```

產生錯誤的原因很微妙。有時候垃圾迴收器會移動一些變量以降低內存碎片等問題。這類垃圾迴收器被稱爲移動GC。當一個變量被移動，所有的保存改變量舊地址的指針必鬚同時被更新爲變量移動後的新地址。從垃圾收集器的視角來看，一個unsafe.Pointer是一個指向變量的指針，因此當變量被移動是對應的指針也必鬚被更新；但是uintptr類型的臨時變量隻是一個普通的數字，所以其值不應該被改變。上面錯誤的代碼因爲引入一個非指針的臨時變量tmp，導致垃圾收集器無法正確識別這個是一個指向變量x的指針。當第二個語句執行時，變量x可能已經被轉移，這時候臨時變量tmp也就不再是現在的 `&x.b` 地址。第三個向之前無效地址空間的賦值語句將徹底摧毀整個程序！

還有很多類似原因導致的錯誤。例如這條語句：

```
pT := uintptr
(unsafe.Pointer(new
(T))) // 提示: 錯誤!
```

這里併沒有指針引用 `new` 新創建的變量，因此該語句執行完成之後，垃圾收集器有權馬上迴收其內存空間，所以返迴的pT將是無效的地址。

雖然目前的Go語言實現還沒有使用移動GC（譯註：未來可能實現），但這不該是編寫錯誤代碼僥幸的理由：當前的Go語言實現已經有移動變量的場景。在5.2節我們提到goroutine的棧是根據需要動態增長的。當發送棧動態增長的時候，原來棧中的所以變量可能需要被移動到新的更大的棧中，所以我們併不能確保變量的地址在整個使用週期內是不變的。

在編寫本文時，還沒有清晰的原則來指引Go程序員，什麼樣的unsafe.Pointer和uintptr的轉換是不安全的（參考 Issue7192 ）. 譯註: 該問題已經關閉），因此我們強烈建議按照最壞的方式處理。將所有包含變量地址的uintptr類型變量當作BUG處理，同時減少不必要的unsafe.Pointer類型到uintptr類型的轉換。在第一個例子中，有三個轉換——字段偏移量到uintptr的轉換和轉迴unsafe.Pointer類型的操作——所有的轉換全在一個表達式完成。

當調用一個庫函數，併且返迴的是uintptr類型地址時（譯註：普通方法實現的函數不盡量不要返迴該類型。下面例子是reflect包的函數，reflect包和unsafe包一樣都是采用特殊技術實現的，編譯器可能給它們開了後門），比如下面反射包中的相關函數，返迴的結果應該立卽轉換爲unsafe.Pointer以確保指針指向的是相同的變量。

```
package
 reflect

func
 (Value) Pointer() uintptr

func
 (Value) UnsafeAddr() uintptr

func
 (Value) InterfaceData() [2
]uintptr
 // (index 1)
```

13.3. 示例: 深度相等判斷

來自reflect包的DeepEqual函數可以對兩個值進行深度相等判斷。DeepEqual函數使用內建的==比較操作符對基礎類型進行相等判斷，對於複合類型則遞歸該變量的每個基礎類型然後做類似的比較判斷。因爲它可以工作在任意的類型上，甚至對於一些不支持==操作運算符的類型也可以工作，因此在一些測試代碼中廣泛地使用該函數。比如下面的代碼是用DeepEqual函數比較兩個字符串數組是否相等。

```
func
 TestSplit(t *testing.T) {
    got := strings.Split("a:b:c"
, ":"
)
    want := []string
{"a"
, "b"
, "c"
};
    if
 !reflect.DeepEqual(got, want) { /* ... */
 }
}
```

盡管DeepEqual函數很方便，而且可以支持任意的數據類型，但是它也有不足之處。例如，它將一個nil值的map和非nil值但是空的map視作不相等，同樣nil值的slice 和非nil但是空的slice也視作不相等。

```
var
 a, b []string
 = nil
, []string
{}
fmt.Println(reflect.DeepEqual(a, b)) // "false"


var
 c, d map
[string
]int
 = nil
, make
(map
[string
]int
)
fmt.Println(reflect.DeepEqual(c, d)) // "false"
```

我們希望在這里實現一個自己的Equal函數，用於比較類型的值。和DeepEqual函數類似的地方是它也是基於slice和map的每個元素進行遞歸比較，不同之處是它將nil值的slice（map類似）和非nil值但是空的slice視作相等的值。基礎部分的比較可以基於reflect包完成，和12.3章的Display函數的實現方法類似。同樣，我們也定義了一個內部函數equal，用於內部的遞歸比較。讀者目前不用關心seen參數的具體含義。對於每一對需要比較的x和y，equal函數首先檢測它們是否都有效（或都無效），然後檢測它們是否是相同的類型。剩下的部分是一個鉅大的switch分支，用於相同基礎類型的元素比較。因爲頁面空間的限製，我們省略了一些相似的分支。

```
gopl.io/ch13/equal
func
 equal(x, y reflect.Value, seen map
[comparison]bool
) bool
 {
    if
 !x.IsValid() || !y.IsValid() {
        return
 x.IsValid() == y.IsValid()
    }
    if
 x.Type() != y.Type() {
        return
 false


    }

    // ...cycle check omitted (shown later)...



    switch
 x.Kind() {
    case
 reflect.Bool:
        return
 x.Bool() == y.Bool()
    case
 reflect.String:
        return
 x.String() == y.String()

    // ...numeric cases omitted for brevity...



    case
 reflect.Chan, reflect.UnsafePointer, reflect.Func:
        return
 x.Pointer() == y.Pointer()
    case
 reflect.Ptr, reflect.Interface:
        return
 equal(x.Elem(), y.Elem(), seen)
    case
 reflect.Array, reflect.Slice:
```

```
        if
 x.Len() != y.Len() {
            return
 false

        }
        for
 i := 0
; i < x.Len(); i++ {
            if
 !equal(x.Index(i), y.Index(i), seen) {
                return
 false

            }
        }
        return
 true


    // ...struct and map cases omitted for brevity...

    }
    panic
("unreachable"
)
}
```

和前面的建議一樣，我們併不公開reflect包相關的接口，所以導出的函數需要在內部自己將變量轉爲reflect.Value類型。

```
// Equal reports whether x and y are deeply equal.

func
 Equal(x, y interface
{}) bool
 {
    seen := make
(map
[comparison]bool
)
    return
 equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)
}

type
 comparison struct
 {
    x, y unsafe.Pointer
    treflect.Type
}
```

爲了確保算法對於有環的數據結構也能正常退出，我們必鬚記録每次已經比較的變量，從而避免進入第二次的比較。Equal函數分配了一組用於比較的結構體，包含每對比較對象的地址（unsafe.Pointer形式保存）和類型。我們要記録類型的原因是，有些不同的變量可能對應相同的地址。例如，如果x和y都是數組類型，那麼x和x[0]將對應相同的地址，y和y[0]也是對應相同的地址，這可以用於區分x與y之間的比較或x[0]與y[0]之間的比較是否進行過了。

```
// cycle check

if
 x.CanAddr() && y.CanAddr() {
    xptr := unsafe.Pointer(x.UnsafeAddr())
    yptr := unsafe.Pointer(y.UnsafeAddr())
    if
 xptr == yptr {
        return
 true
 // identical references

    }
    c := comparison{xptr, yptr, x.Type()}
    if
 seen[c] {
        return
 true
 // already seen

    }
    seen[c] = true

}
```

這是Equal函數用法的例子:

```
fmt.Println(Equal([]int
{1
, 2
, 3
}, []int
{1
, 2
, 3
}))        // "true"

fmt.Println(Equal([]string
{"foo"
}, []string
{"bar"
}))      // "false"

fmt.Println(Equal([]string
(nil
), []string
{}))             // "true"

fmt.Println(Equal(map
[string
]int
(nil
), map
[string
]int
{})) // "true"
```

Equal函數甚至可以處理類似12.3章中導致Display陷入陷入死循環的帶有環的數據。

```
// Circular linked lists a -> b -> a and c -> c.

type
 link struct
 {
    value string

    tail *link
}
a, b, c := &link{value: "a"
}, &link{value: "b"
}, &link{value: "c"
}
a.tail, b.tail, c.tail = b, a, c
fmt.Println(Equal(a, a)) // "true"

fmt.Println(Equal(b, b)) // "true"

fmt.Println(Equal(c, c)) // "true"

fmt.Println(Equal(a, b)) // "false"

fmt.Println(Equal(a, c)) // "false"
```

練習 13.1： 定義一個深比較函數，對於十億以內的數字比較，忽略類型差異。

練習 13.2： 編寫一個函數，報告其參數是否循環數據結構。

13.4. 通過cgo調用C代碼

Go程序可能會遇到要訪問C語言的某些硬件驅動函數的場景，或者是從一個C++語言實現的嵌入式數據庫查詢記録的場景，或者是使用Fortran語言實現的一些線性代數庫的場景。C語言作爲一個通用語言，很多庫會選擇提供一個C兼容的API，然後用其他不同的編程語言實現（譯者：Go語言需要也應該擁抱這些鉅大的代碼遺産）。

在本節中，我們將構建一個簡易的數據壓縮程序，使用了一個Go語言自帶的叫cgo的用於支援C語言函數調用的工具。這類工具一般被稱爲 *foreign-function interfaces*（簡稱ffi）, 併且在類似工具中cgo也不是唯一的。SWIG（ http://swig.org ）是另一個類似的且被廣泛使用的工具，SWIG提供了很多複雜特性以支援C++的特性，但SWIG併不是我們要討論的主題。

在標準庫的 `compress/...` 子包有很多流行的壓縮算法的編碼和解碼實現，包括流行的LZW壓縮算法（Unix的compress命令用的算法）和DEFLATE壓縮算法（GNU gzip命令用的算法）。這些包的API的細節雖然有些差異，但是它們都提供了針對 io.Writer類型輸出的壓縮接口和提供了針對io.Reader類型輸入的解壓縮接口。例如：

```
package
 gzip // compress/gzip

func
 NewWriter(w io.Writer) io.WriteCloser
func
 NewReader(r io.Reader) (io.ReadCloser, error)
```

bzip2壓縮算法，是基於優雅的Burrows-Wheeler變換算法，運行速度比gzip要慢，但是可以提供更高的壓縮比。標準庫的compress/bzip2包目前還沒有提供bzip2壓縮算法的實現。完全從頭開始實現是一個壓縮算法是一件繁瑣的工作，而且 http://bzip.org 已經有現成的libbzip2的開源實現，不僅文檔齊全而且性能又好。

如果是比較小的C語言庫，我們完全可以用純Go語言重新實現一遍。如果我們對性能也沒有特殊要求的話，我們還可以用os/exec包的方法將C編寫的應用程序作爲一個子進程運行。隻有當你需要使用複雜而且性能更高的底層C接口時，就是使用cgo的場景了（譯註：用os/exec包調用子進程的方法會導致程序運行時依賴那個應用程序）。下面我們將通過一個例子講述cgo的具體用法。

譯註：本章采用的代碼都是最新的。因爲之前已經出版的書中包含的代碼隻能在Go1.5之前使用。從Go1.6開始，Go語言已經明確規定了哪些Go語言指針可以之間傳入C語言函數。新代碼重點是增加了bz2alloc和bz2free的兩個函數，用於bz_stream對象空間的申請和釋放操作。下面是新代碼中增加的註釋，説明這個問題：

```
// The version of this program that appeared in the first and second

// printings did not comply with the proposal rules for passing

// pointers between Go and C, described here:

// https://github.com/golang/proposal/blob/master/design/12416-cgo-pointers.md

//

// The rules forbid a C function like bz2compress from storing 'in'

// and 'out' (pointers to variables allocated by Go) into the Go

// variable 's', even temporarily.

//

// The version below, which appears in the third printing, has been

// corrected.  To comply with the rules, the bz_stream variable must

// be allocated by C code.  We have introduced two C functions,

// bz2alloc and bz2free, to allocate and free instances of the

// bz_stream type.  Also, we have changed bz2compress so that before

// it returns, it clears the fields of the bz_stream that contain

// pointers to Go variables.
```

要使用libbzip2，我們需要先構建一個bz_stream結構體，用於保持輸入和輸出緩存。然後有三個函數：BZ2_bzCompressInit用於初始化緩存，BZ2_bzCompress用於將輸入緩存的數據壓縮到輸出緩存，BZ2_bzCompressEnd用於釋放不需要的緩存。（目前不要擔心包的具體結構,這個例子的目的就是演示各個部分如何組合在一起的。）

我們可以在Go代碼中直接調用BZ2_bzCompressInit和BZ2_bzCompressEnd，但是對於BZ2_bzCompress，我們將定義一個C語言的包裝函數，用它完成眞正的工作。下面是C代碼，對應一個獨立的文件。

```
gopl.io/ch13/bzip

/* This file is gopl.io/ch13/bzip/bzip2.c,         */

/* a simple wrapper for libbzip2 suitable for cgo. */

#include
 <bzlib.h>




int
 bz2compress
(bz_stream *s, int
 action,
                char
 *in, unsigned
 *inlen, char
 *out, unsigned
 *outlen)

{
    s->next_in = in;
    s->avail_in = *inlen;
    s->next_out = out;
    s->avail_out = *outlen;
    int
 r = BZ2_bzCompress(s, action);
    *inlen -= s->avail_in;
    *outlen -= s->avail_out;
    s->next_in = s->next_out = NULL
;
    return
 r;
}
```

現在讓我們轉到Go語言部分，第一部分如下所示。其中 `import "C"` 的語句是比較特別的。其實併沒有一個叫C的包，但是這行語句會讓Go編譯程序在編譯之前先運行cgo工具。

```
// Package bzip provides a writer that uses bzip2 compression (bzip.org).

package
 bzip

/*
#cgo CFLAGS: -I/usr/include
#cgo LDFLAGS: -L/usr/lib -lbz2
#include <bzlib.h>
#include <stdlib.h>
```

```
#include <stdlib.h>
bz_stream* bz2alloc() { return calloc(1, sizeof(bz_stream)); }
int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen);
void bz2free(bz_stream* s) { free(s); }
*/

import
 "C"


import
 (
    "io"

    "unsafe"

)

type
 writer struct
 {
    w      io.Writer // underlying output stream

    stream *C.bz_stream
    outbuf [64
 * 1024
]byte

}

// NewWriter returns a writer for bzip2-compressed streams.

func
 NewWriter(out io.Writer) io.WriteCloser {
    const
 blockSize = 9

    const
 verbosity = 0

    const
 workFactor = 30

    w := &writer{w: out, stream: C.bz2alloc()}
    C.BZ2_bzCompressInit(w.stream, blockSize, verbosity, workFactor)
    return
 w
}
```

在預處理過程中，cgo工具爲生成一個臨時包用於包含所有在Go語言中訪問的C語言的函數或類型。例如C.bz_stream和C.BZ2_bzCompressInit。cgo工具通過以某種特殊的方式調用本地的C編譯器來發現在Go源文件導入聲明前的註釋中包含的C頭文件中的內容（譯註：`import "C"`語句前僅捱着的註釋是對應cgo的特殊語法，對應必要的構建參數選項和C語言代碼）。

在cgo註釋中還可以包含#cgo指令，用於給C語言工具鏈指定特殊的參數。例如CFLAGS和LDFLAGS分別對應傳給C語言編譯器的編譯參數和鏈接器參數，使它們可以特定目録找到bzlib.h頭文件和libbz2.a庫文件。這個例子假設你已經在/usr目録成功安裝了bzip2庫。如果bzip2庫是安裝在不同的位置，你需要更新這些參數（譯註：這里有一個從純C代碼生成的cgo綁定，不依賴bzip2靜態庫和操作繫統的具體環境，具體請訪問 https://github.com/chai2010/bzip2 ）。

NewWriter函數通過調用C語言的BZ2_bzCompressInit函數來初始化stream中的緩存。在writer結構中還包括了另一個buffer，用於輸出緩存。

下面是Write方法的實現，返迴成功壓縮數據的大小，主體是一個循環中調用C語言的bz2compress函數實現的。從代碼可以看到，Go程序可以訪問C語言的bz_stream、char和uint類型，還可以訪問bz2compress等函數，甚至可以訪問C語言中像BZ_RUN那樣的宏定義，全部都是以C.x語法訪問。其中C.uint類型和Go語言的uint類型併不相同，即使它們具有相同的大小也是不同的類型。

```
func
 (w *writer) Write(data []byte
) (int
, error) {
    if
 w.stream == nil
 {
        panic
("closed"
)
    }
    var
 total int
 // uncompressed bytes written


    for
 len
(data) > 0
 {
        inlen, outlen := C.uint
(len
(data)), C.uint
(cap
(w.outbuf))
        C.bz2compress(w.stream, C.BZ_RUN,
            (*C.char)(unsafe.Pointer(&data[0
])), &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        total += int
(inlen)
        data = data[inlen:]
        if
 _, err := w.w.Write(w.outbuf[:outlen]); err != nil
 {
            return
 total, err
        }
    }
    return
 total, nil

}
```

在循環的每次迭代中，向bz2compress傳入數據的地址和剩餘部分的長度，還有輸出緩存w.outbuf的地址和容量。這兩個長度信息通過它們的地址傳入而不是值傳入，因爲bz2compress函數可能會根據已經壓縮的數據和壓縮後數據的大小來更新這兩個值。每個塊壓縮後的數據被寫入到底層的io.Writer。

Close方法和Write方法有着類似的結構，通過一個循環將剩餘的壓縮數據刷新到輸出緩存。

```
// Close flushes the compressed data and closes the stream.

// It does not close the underlying io.Writer.

func
 (w *writer) Close() error {
    if
 w.stream == nil
 {
        panic
("closed"
)
    }
    defer
 func
() {
        C.BZ2_bzCompressEnd(w.stream)
        C.bz2free(w.stream)
        w.stream = nil

    }()
    for
 {
        inlen, outlen := C.uint
(0
), C.uint
(cap
(w.outbuf))
        r := C.bz2compress(w.stream, C.BZ_FINISH, nil
, &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        if
 _, err := w.w.Write(w.outbuf[:outlen]); err != nil
 {
            return
 err
        }
        if
 r == C.BZ_STREAM_END {
            return
 nil

        }
    }
}
```

壓縮完成後，Close方法用了defer函數確保函數退出前調用C.BZ2_bzCompressEnd和C.bz2free釋放相關的C語言運行時資源。此刻w.stream指針將不再有效，我們將它設置爲nil以保證安全，然後在每個方法中增加了nil檢測，以防止用戶在關閉後依然錯誤使用相關方法。

上面的實現中，不僅僅寫是非併發安全的，甚至併發調用Close和Write方法也可能導致程序的的崩潰。脩複這個問題是練習13.3的內容。

下面的bzipper程序，使用我們自己包實現的bzip2壓縮命令。它的行爲和許多Unix繫統的bzip2命令類似。

```
gopl.io/ch13/bzipper

// Bzipper reads input, bzip2-compresses it, and writes it out.

package
 main

import
 (
    "io"

    "log"

    "os"

    "gopl.io/ch13/bzip"

)

func
 main() {
    w := bzip.NewWriter(os.Stdout)
    if
 _, err := io.Copy(w, os.Stdin); err != nil
 {
        log.Fatalf("bzipper: %v\n"
, err)
    }
    if
 err := w.Close(); err != nil
 {
        log.Fatalf("bzipper: close: %v\n"
, err)
    }
}
```

在上面的場景中，我們使用bzipper壓縮了/usr/share/dict/words繫統自帶的詞典，從938,848字節壓縮到335,405字節。大約是原始數據大小的三分之一。然後使用繫統自帶的bunzip2命令進行解壓。壓縮前後文件的SHA256哈希碼是相同了，這也説明了我們的壓縮工具是正確的。（如果你的繫統沒有sha256sum命令，那麽請先按照練習4.2實現一個類似的工具）

```
$ go build gopl.io/ch13/bzipper
$ wc -c < /usr/share/dict/words
938848
$ sha256sum < /usr/share/dict/words
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
$ ./bzipper < /usr/share/dict/words | wc -c
335405
$ ./bzipper < /usr/share/dict/words | bunzip2 | sha256sum
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
```

我們演示了如何將一個C語言庫鏈接到Go語言程序。相反, 將Go編譯爲靜態庫然後鏈接到C程序，或者將Go程序編譯爲動態庫然後在C程序中動態加載也都是可行的（譯註：在Go1.5中，Windows繫統的Go語言實現併不支持生成C語言動態庫或靜態庫的特性。不過好消息是，目前已經有人在嚐試解決這個問題，具體請訪問 Issue11058 ）。這里我們隻展示的cgo很小的一些方面，更多的關於內存管理、指針、迴調函數、中斷信號處理、字符串、errno處理、終結器，以及goroutines和繫統線程的關繫等，有很多細節可以討論。特別是如何將Go語言的指針傳入C函數的規則也是異常複雜的（譯註：簡單來説，要傳入C函數的Go指針指向的數據本身不能包含指針或其他引用類型；併且C函數在返迴後不能繼續持有Go指針；併且在C函數返迴之前，Go指針是被鎖定的，不能導致對應指針數據被移動或棧的調整），部分的原因在13.2節有討論到，但是在Go1.5中還沒有被明確（譯註：Go1.6將會明確cgo中的指針使用規則）。如果要進一步閱讀，可以從 https://golang.org/cmd/cgo 開始。

練習 13.3： 使用sync.Mutex以保證bzip2.writer在多個goroutines中被併發調用是安全的。

練習 13.4： 因爲C庫依賴的限製。 使用os/exec包啟動/bin/bzip2命令作爲一個子進程，提供一個純Go的bzip.NewWriter的替代實現（譯註：雖然是純Go實現，但是運行時將依賴/bin/bzip2命令，其他操作繫統可能無法運行）。

13.5. 幾點忠告

我們在前一章結尾的時候，我們警告要謹慎使用reflect包。那些警告同樣適用於本章的unsafe包。

高級語言使得程序員不用在關心眞正運行程序的指令細節，同時也不再需要關註許多如內存布局之類的實現細節。因爲高級語言這個絶緣的抽象層，我們可以編寫安全健壯的，併且可以運行在不同操作繫統上的具有高度可移植性的程序。

但是unsafe包，它讓程序員可以透過這個絶緣的抽象層直接使用一些必要的功能，雖然可能是爲了獲得更好的性能。但是代價就是犧牲了可移植性和程序安全，因此使用unsafe包是一個危險的行爲。我們對何時以及如何使用unsafe包的建議和我們在11.5節提到的Knuth對過早優化的建議類似。大多數Go程序員可能永遠不會需要直接使用unsafe包。當然，也永遠都會有一些需要使用unsafe包實現會更簡單的場景。如果確實認爲使用unsafe包是最理想的方式，那麼應該盡可能將它限製在較小的范圍，那樣其它代碼就忽略unsafe的影響。

現在，趕緊將最後兩章拋入腦後吧。編寫一些實實在在的應用是眞理。請遠離reflect的unsafe包，除非你確實需要它們。

最後，用Go快樂地編程。我們希望你能像我們一樣喜歡Go語言。

附録：作者/譯者

英文作者

- **Alan A. A. Donovan** is a member of Google's Go team in New York. He holds computer science degrees from Cambridge and MIT and has been programming in industry since 1996. Since 2005, he has worked at Google on infrastructure projects and was the co-designer of its proprietary build system, Blaze . He has built many libraries and tools for static analysis of Go programs, including oracle , `godoc -analysis` , eg, and gorename .
- **Brian W. Kernighan** is a professor in the Computer Science Department at Princeton University. He was a member of technical staff in the Computing Science Research Center at Bell Labs from 1969 until 2000, where he worked on languages and tools for Unix . He is the co-author of several books, including The C Programming Language, Second Edition (Prentice Hall, 1988) , and The Practice of Programming (Addison-Wesley, 1999) .

中文譯者

中文譯者	章節
`chai2010 <chaishushan@gmail.com>`	前言/第2~4章/第10~13章
`CrazySssst`	第5章
`foreversmart <njutree@gmail.com>`	第7章
`Xargin <cao1988228@163.com>`	第1章/第6章/第8~9章

譯文授權