前言

我曾經在第一版寫道：

> 我本來想要把這本書命名為 *Boring Go*，因為正確的 Go 程式碼很無趣。
>
> 無趣並非瑣碎、無價值（trivial）。要正確地使用 Go，你必須瞭解它的功能是怎麼互相搭配的。雖然你可以寫出看似 Java 或 Python 的 Go 程式碼，但你應該不會滿意它們的效果，甚至懷疑寫出那些東西有何意義。這就是本書的目的，它將帶你瞭解 Go 的各種功能，解釋如何以最好的方式使用它們，以寫出可擴展且符合慣例的程式碼。

Go 仍然是一種功能有限的小型語言。它仍然沒有繼承、特性導向設計（aspect-oriented programming）、函式多載、運算子多載、模式比對、例外、具名參數，以及讓其他語言那麼複雜的許多額外功能。那麼，為什麼這本介紹無趣語言的書要出第二版？

之所以出這一版有幾個原因。首先，無趣不代表瑣碎無價值，也不意味著一成不變。在過去三年裡，Go 語言加入許多功能、工具，和程式庫，以及一些改進，例如結構化日誌紀錄（logging）、fuzzing（模糊化）、工作區，和漏洞檢查，以幫助 Go 開發者寫出可靠、持久，且容易維護的程式。隨著 Go 開發者累積多年的泛型使用經驗，標準程式庫開始納入型態約束和泛型函式，以減少重複的程式碼。即使是 unsafe 套件也被更新，來讓它更安全一些。Go 開發者需要一本書來解釋如何善用這些新功能。

其次，第一版對一些層面的介紹不夠充分。我不滿意第一版導論的敘事順序，且第一版並未探討豐富的 Go 工具生態系統。此外，第一版的讀者希望我加入習題和更多範例程式。這一版試圖彌補這些不足。

最後，Go 團隊加入一些新的，甚至容我大膽地說，令人興奮的功能。現在當你進行長期的軟體工程專案時，能夠在維持回溯相容的同時，加入不回溯相容的變更，以解決長期存在的設計缺陷。Go 1.22 加入的 `for` 迴圈變數作用域規則就是利用這種做法的第一種功能。

Go 依然無趣，但它也仍然很卓越，甚至比以往更好。希望你喜歡這本第二版。

誰應該閱讀這本書？

本書的目標讀者是希望學習第二種語言的開發者（或第五種）。本書特別重視剛接觸 Go 的人，這個族群包括只知道 Go 有一隻可愛吉祥物的人，以及完成過一門 Go 課程，甚至寫過一些 Go 程式的讀者。*Learning Go* 並非只介紹如何使用 Go 來寫出程式，它也教你如何以符合 *Go 語言慣例*的寫法來寫出程式。對於比較有經驗的 Go 開發者來說，這本書提供如何善用新功能的建議。最重要的是，本書會教你如何寫出看起來像 Go 的 Go 程式。

我們假設讀者已經熟悉業界常用的開發工具了，例如版本控制系統（最好是 Git）和 IDE。讀者應該對基本的計算機科學有一定程度的瞭解，例如並行（concurrency）和抽象，因為本書會解釋這些概念在 Go 中是如何運作的。你可以從 GitHub 下載一些範例程式，並且在 The Go Playground 上試驗數十個範例。雖然使用它們時不需要網路，但是當你觀察可執行的範例時，有網路將更加方便。由於 Go 通常被用來建構和呼叫 HTTP 伺服器，有一些範例假設讀者已經熟悉基本的 HTTP 概念。

雖然 Go 的大部分功能在其他語言中也能找到，但 Go 做了不同的取捨，因此 Go 程式的結構將有所不同。*Learning Go* 會先介紹如何設置 Go 開發環境，接著討論變數、型態、控制結構，和函式。如果你想要跳過這些章節，請稍安勿躁，並仔細閱讀。讓 Go 程式碼更符合慣例的關鍵往往潛伏在這些細節之中。當你深入探究時，或許你會發現看似理所當然的內容隱藏著微妙且令人驚訝之處。

本書編排慣例

本書使用下列的編排方式：

斜體字（*Italic*）

代表新術語、URL、email 地址、檔名，與副檔名。中文以楷體表示。

第六章

指標

瞭解變數和函式之後，接下來該學習指標語法了。我將藉著比較其他語言的類別的行為來解釋 Go 的指標的行為。你還將學習如何使用指標、使用它的時機、Go 是如何配置記憶體的，以及如何正確地使用指標和值來讓 Go 程式跑得更快且更有效率。

指標快速入門

指標是一種變數，它保存了值在記憶體內的位置，如果你修過計算機科學課程，你應該看過關於「變數是怎麼存放在記憶體內的」的圖表。下面這兩個變數可以用圖 6-1 來表示：

```
var x int32 = 10
var y bool = true
```

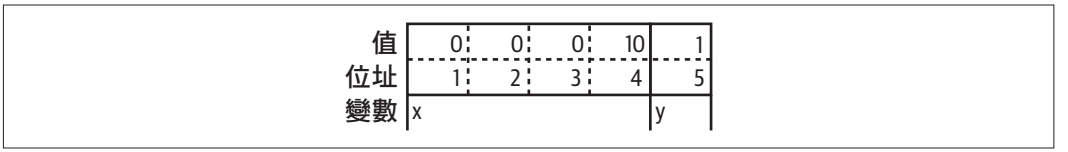

圖 6-1　將兩個變數儲存於記憶體內

每一個變數都被存放在一個或多個連續的記憶體位置內，那些位置稱為位址。不同型態的變數可能占用不同數量的記憶體。這個例子有兩個變數，x 是 32-bit int，y 是布林。儲存 32-bit int 需要 4 bytes，所以 x 的值被存放在 4 bytes 內，從位址 1 到位址 4。布林只需要 1 byte（雖然表示 `true` 與 `false` 只需要 1 bit，但是可以獨立定址的最小記憶體是 1 byte），所以 y 值被儲存在位址 5 的 1 byte，用值 1 來代表 `true`。

指標是一個變數，它的內容是另一個變數的儲存位址。圖 6-2 是下面的指標在記憶體裡面的情況：

```
var x int32 = 10
var y bool = true
pointerX := &x
pointerY := &y
var pointerZ *string
```

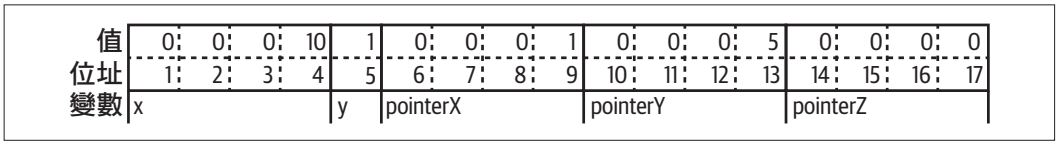

圖 6-2　在記憶體內儲存指標

雖然不同型態的變數可能占用不同數量的記憶體位置，但是每一個指標都有相同數量的記憶體位置，無論它指向哪一種型態。本章的範例使用 4-byte 指標，但許多現代計算機的指標使用 8 bytes。指標保存一個數字，它是指標所指的資料在記憶體內的位置。該數字稱為位址。指向 x 的指標 `pointerX` 被存放在位置 6，它的值是 1，即 x 的位址。同理，指向 y 的指標 `pointerY` 被存放在位置 10，它的值是 5，即 y 的位址。最後一個指標 `pointerZ` 被存放在位置 14，它的值是 0，因為它沒有指向任何東西。

指標的零值是 `nil`。你已經看過 `nil` 好幾次了，它也是 slice、map，與函式的零值，這些型態都是用指標來實作的（另外兩種型態，channel 與 interface 也是用指標來實作的。我們會在第 161 頁的「介面快速入門」與第 299 頁的「channel」介紹它們）。如第 3 章所述，`nil` 是一種 untyped 代號，代表某種型態無值的情況。與 C 的 `NULL` 不同的是，`nil` 不是 0 的別稱，你不能在它與數字之間來回轉換。

第 4 章說過，`nil` 是在全域區塊裡定義的值，所以它可能被遮蔽。絕對不要將變數或函式命名為 `nil`，除非你想要陷害你的同事，或你不在乎年度考核。

Go 的指標語法部分借鑒了 C 與 C++。因為 Go 有記憶體回收程序，所以你幾乎完全不需要做痛苦的記憶體管理工作，此外，Go 不能做一些可在 C 與 C++ 裡用指標來操作的技巧，包括指標算術。

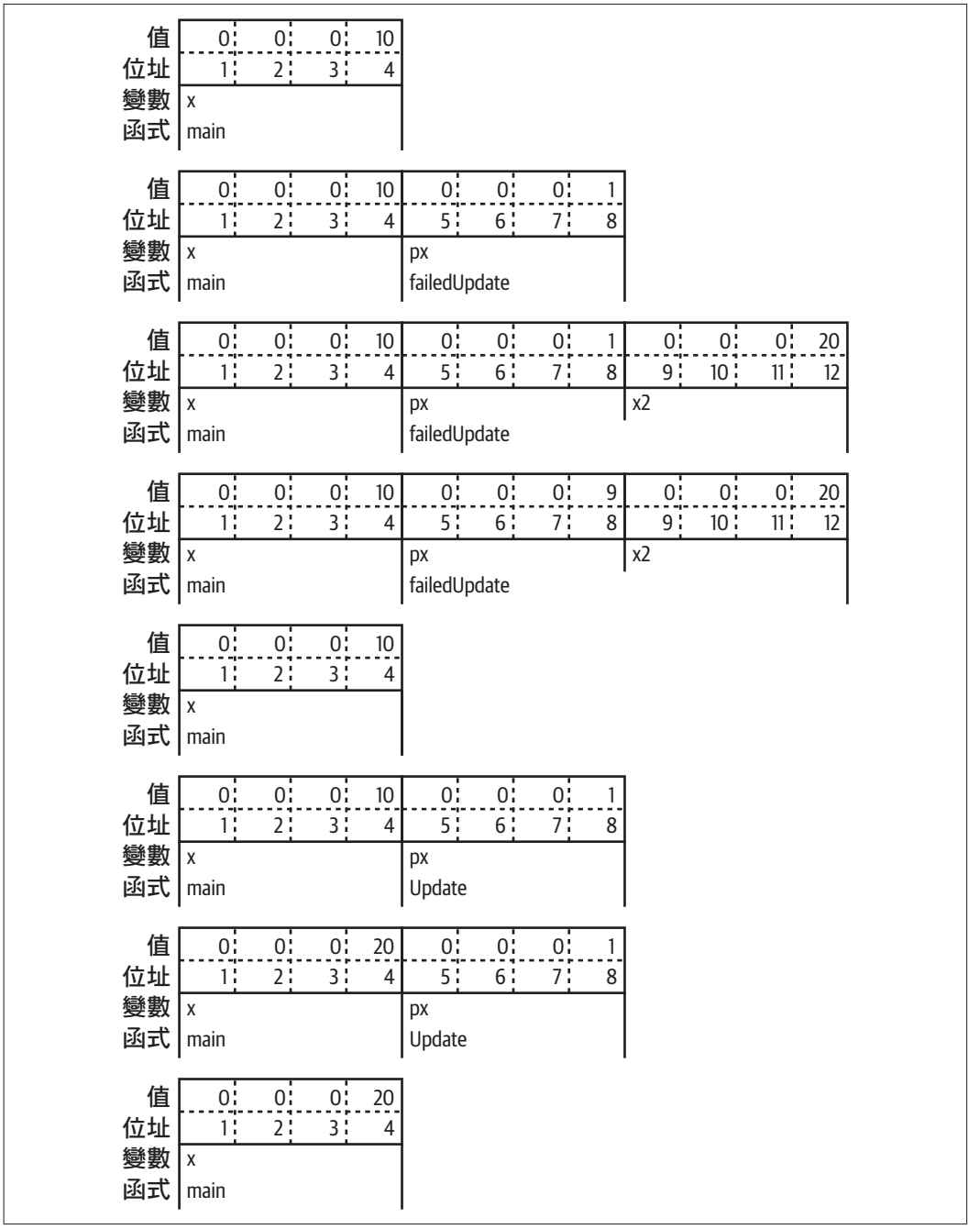

圖 6-4　更新指標的正確方式與錯誤方式

指標是最終手段

話雖如此，在 Go 裡面使用指標時要很小心，如前所述，它們會讓資料流難以理解，而且可能會讓記憶體回收程序增加額外的工作負擔。如果你要填寫一個 struct，你應該讓函式實例化 struct 並回傳它，而不是將一個指向 struct 的指標傳給函式（見範例 6-3 與 6-4）。

範例 *6-3*　別這樣做

```
func MakeFoo(f *Foo) error {
  f.Field1 = "val"
  f.Field2 = 20
  return nil
}
```

範例 *6-4*　而是要這樣做

```
func MakeFoo() (Foo, error) {
  f := Foo{
    Field1: "val",
    Field2: 20,
  }
  return f, nil
}
```

只有在函式期望接收介面時，你才可以使用指標參數來修改變數，你會在使用 JSON 時看到這種模式（我們將在第 337 頁的「encoding/json」詳細討論 Go 標準程式庫對於 JSON 的支援）：

```
f := struct {
  Name string `json:"name"`
  Age int `json:"age"`
}{}
err := json.Unmarshal([]byte(`{"name": "Bob", "age": 30}`), &f)
```

`Unmarshal` 函式可將包含 JSON 的 byte slice 的內容填入一個變數。它接收一個 byte slice 和一個 `any` 參數。被傳入 `any` 參數的值必須是一個指標，如果不是，函式會回傳一個錯誤。

為什麼要將指標傳入 `Unmarshal`，而不是讓它回傳一個值？原因有兩個。首先，這個函式在 Go 引入泛型之前就被設計出來了，沒有泛型（第 8 章會詳細介紹泛型）就無法知道該建立並回傳哪種型態的值。

第二個原因是傳入指標可讓你控制記憶體配置。遍歷資料並將它從 JSON 轉換為 Go struct 是一種常見的設計模式，所以 Go 為此優化了 `Unmarshal`。如果 `Unmarshal` 函式回傳一個值，而且你在迴圈中呼叫 `Unmarshal`，那麼每一次的迴圈迭代都會建立一個 struct 實例。這會給資料回收程序帶來更多工作，從而讓程式跑得更慢。你將在第 139 頁的「將 slice 當成緩衝區」看到這個模式的另一個使用案例，我也會在第 140 頁的「減少記憶體回收程序的工作負擔」中談到更多關於有效使用記憶體的內容。

因為 JSON 整合是如此常見，Go 新人可能認為使用這個 API 是常態，而不是罕見的例外。

當你從函式回傳值時，應該優先使用值型態。除非在資料型態裡面有狀態需要更改，否則不要回傳指標型態。當你在第 329 頁的「io 與朋友」瞭解 I/O 時，你會看到用來讀寫資料的緩衝區。此外，有一些與並行（concurrency）一起使用的資料型態一定要用指標來傳遞，你會在第 12 章看到它們。

指標傳遞效能

如果 struct 夠大，將 struct 的指標當成輸入參數或回傳值可以改善效能。無論資料多大，將指標傳入函式的時間都是固定的，大約是 1 奈秒，這很合理，因為所有資料型態的指標都一樣大。將值傳入函式會隨著資料越大而花越多時間，當值有 10 MB 左右的資料時，大概需要 0.7 毫秒。

回傳指標 vs. 回傳值的行為更有意思。如果資料結構小於 10 MB，回傳指標型態其實比回傳值更慢。例如，100-byte 的資料結構大約需要 10 奈秒來回傳，但是該資料結構的指標大約需要 30 奈秒。隨著資料結構變大，效能優勢會反轉，回傳 10 MB 的資料大約要花 1.5 毫秒，但回傳它的指標略少於半毫秒。

它們是很短的時間，對大多數的情況而言，使用指標與使用值的差異不會影響程式的效能。但是如果你要在函式之間傳遞幾 MB 的資料，請考慮使用指標，即使資料是不可變的。

以上的數據來自配備 32 GB RAM 的 i7-8700 電腦。不同的 CPU 可能會產生不同的臨界點。例如，在配備 16 GB RAM 的 Apple M1 CPU 上，當值的大小是 100 KB 左右時，回傳指標的速度（5 微秒）比回傳值（8 微秒）更快。你可以使用第 6 章版本庫（*https://oreil.ly/riOYA*）的 *sample_code/pointer_perf* 目錄內的程式來自行進行效能測試。你可以執行命令 `go test ./… -bench=.` 來查看結果（第 405 頁的「使用效能評測」會詳細介紹效能評測）。

第八章

泛型

「Don't repeat yourself（不要重複自己做過的事）」是軟體工程界的老生常談。重複建立資料結構或函式不如重複利用它們，因為同步更新重複的程式碼很難持續做下去。在 Go 這樣的強定型語言中，每一個函式參數和每一個 struct 欄位的型態在編譯時必須是已知的。雖然這種嚴格的規定能夠讓編譯器協助你驗證程式碼是否正確，但有時你希望用不同的型態來重複利用函式內的邏輯或 struct 內的欄位。Go 透過型態參數來提供這項功能，它通常稱為泛型。本章要說明為何需要泛型、Go 的泛型能夠做什麼、不能做什麼，以及如何正確使用它們。

泛型可減少重複的程式碼，並提高型態安全性

Go 是一種靜態定型語言，這意味著變數和參數的型態是在編譯時檢查的。內建型態（如 map、slice、channel）和函式（如 `len`、`cap` 或 `make`）可以接受和回傳不同具體型態的值，但是在 Go 1.18 之前，使用者定義的 Go 型態和函式無法做到這一點。

如果你習慣使用動態定型語言，你可能無法理解泛型為何備受關注，甚至不明白泛型的概念，因為那些語言的型態在程式碼執行時才被算出來。你可以將它們想成「型態參數」。在本書之前的內容中，你已經看過一些附帶參數的函式，那些參數的值是在呼叫函式時指定的。在第 100 頁的「多個回傳值」中，函式 `divAndRemainder` 接收兩個 `int` 型態的參數，並回傳兩個 `int` 值：

有了泛型之後，你可以為多個型態實作一個資料結構，並在編譯時檢測不相容的資料。接下來要告訴你如何正確地使用它們。

雖然無泛型的資料結構很不方便，但最麻煩的是在寫函式的時候。在 Go 標準程式庫裡，有幾個實作決策是在這個語言還沒有泛型的時候做出來的。例如，Go 並未寫出多個函式來處理不同的數值型態，而是使用 `float64` 參數來實作 `math.Max`、`math.Min` 和 `math.Mod` 等函式，因為 `float64` 有夠大的範圍，可以精確地表示幾乎所有其他數值型態（但值大於 $2^{53} - 1$ 或小於 $-2^{53} - 1$ 的 `int`、`int64` 或 `uint` 例外）。

有些事情沒有泛型無法做到，例如，你無法建立以介面來指定的變數的新實例，也無法指定相同介面型態的兩個參數必須是同一個具體型態。沒有泛型就一定要使用 reflection 才能寫出可處理任意型態的 slice 的函式，這將犧牲效能，以及編譯期型態安全性（這正是 `sort.Slice` 的做法）。這意味著，在 Go 加入泛型之前，Go 必須為每一種 slice 型態重複製作操作 slice 的函式（例如 `map`、`reduce` 和 `filter`）。雖然複製簡單的演算法很容易，但許多（應該是絕大多數的）軟體工程師認為，僅僅因為編譯器沒有自動處理的能力而必須編寫重複的程式並不是好事。

Go 泛型簡介

自從 Go 最初發表以來，大眾就呼籲將泛型加入這種語言中。Go 的開發主管 Russ Cox 在 2009 年寫了一篇部落格文章（*https://oreil.ly/U4huA*）來解釋為何最初未加入泛型。Go 重視快速的編譯器、易讀的程式碼，以及優秀的執行時間，當時他們所瞭解的泛型實作方案都無法同時滿足這三個要求。在花了十年的時間研究這個問題後，Go 團隊提出一個可行的解決方案，其概述可在「Type Parameters Proposal」（*https://oreil.ly/31ay7*）中看到。

你可以藉著觀察堆疊來瞭解 Go 中的泛型是如何運作的。跟沒有計算機科學背景的讀者解釋一下，堆疊是一種資料型態，它的值是按照後進先出（LIFO）的順序加入和移除的。它就像是一疊等待清洗的盤子，最早放進去的盤子位於最下面，你只能藉著先處理後來才放入的盤子，才能拿到最下面的盤子。接著來看看如何使用泛型來建立堆疊。

```
type Stack[T any] struct {
    vals []T
}

func (s *Stack[T]) Push(val T) {
    s.vals = append(s.vals, val)
}
```

用型態元素來限制常數

型態元素也可以指定哪些常數可以被指派給泛型型態的變數。就像運算子一樣，這些常數對型態元素的所有型態項而言都必須是有效的。沒有常數可以指派給 `Ordered` 介面中列出的每一個型態，因此你無法將常數指派給這種泛型型態的變數。如果你使用 `Integer` 介面，以下程式碼將無法編譯，因為你不能將值 1,000 指派給 8 位元的整數：

```
// 無效！
func PlusOneThousand[T Integer](in T) T {
    return in + 1_000
}
```

然而，這段程式碼是有效的：

```
// 有效
func PlusOneHundred[T Integer](in T) T {
    return in + 100
}
```

結合泛型函式與泛型資料結構

我們回到二元樹的例子，看看如何結合你學到的知識來建立一棵適用於任何具體型態的樹。

重點在於，你要意識到這棵樹需要一個能夠比較兩個值並回傳它們的順序的泛型函式：

```
type OrderableFunc [T any] func(t1, t2 T) int
```

有了 `OrderableFunc` 之後，你可以稍微修改樹的實作。首先，將它拆成兩個型態，`Tree` 和 `Node`：

```
type Tree[T any] struct {
    f    OrderableFunc[T]
    root *Node[T]
}

type Node[T any] struct {
    val         T
    left, right *Node[T]
}
```

```
        return v
    }).Reduce(0, func(acc int, cur int) int {
        return acc + cur
    })
```

對泛函編程的愛好者來說，很遺憾，這種寫法無效。你無法將方法呼叫串接起來，而是要嵌入函式呼叫，或使用更易讀的寫法，也就是一次呼叫一個函式，並將中間結果指派給變數。在型態參數提案（type parameter proposal）中，你可以找到 Go 排除參數化方法的原因。

Go 也沒有可變數量（variadic）型態參數。正如第 99 頁的「可變數量的輸入參數與 slice」所討論的那樣，要實作能夠接收可變數量參數的函式，你必須指明最後一個參數型態以 `...` 開頭。例如，你無法指定這些可變數量參數具有某種型態模式，例如交替出現的 `string` 和 `int`。所有可變數量參數必須符合所宣告的單一型態，該型態可以是泛型或非泛型的。

Go 泛型排除的其他功能則比較艱深，包括：

專門化（*Specialization*）

除了泛型版本外，函式或方法也能夠使用一個或多個特定型態的版本來多載。由於 Go 不支援多載，這個功能不列入考慮範圍。

Currying

在一個泛型函式或型態之上指定部分的型態參數來部分實例化另一個函式或型態。

Metaprogramming

指定在編譯時執行的程式碼，以便產生在程式執行時執行的程式碼。

符合慣例的 Go 與泛型

加入泛型顯然改變了一些關於如何以慣用方式來編寫 Go 程式碼的建議。使用 `float64` 來表示任意數值型態這種做法即將成為過去式。你應該使用 `any` 來取代 `interface{}` 以表示資料結構或函式參數中的未指定之型態。你可以用單一函式來處理不同的 slice 型態。但不需要急著將所有程式改成使用型態參數。隨著新設計模式的出現和完善，你的舊程式仍然可以運行。

在標準程式庫中加入泛型

Go 1.18 發表的初版泛型相當保守。它將新介面 `any` 和 `comparable` 加到全域區塊中，但沒有修改標準程式庫中的任何 API 來支援泛型。它改變了一些風格，在標準程式庫中，將幾乎所有的 `interface{}` 都換成 `any`。

隨著 Go 社群對泛型的接受度逐漸提高，我們開始看到更多變化。從 Go 1.21 開始，標準程式庫加入一些使用泛型來實作常見演算法以處理 slice、map，和並行的函式。第 3 章曾經介紹 `slices` 和 `maps` 套件中的 `Equal` 和 `EqualFunc` 函式。在這些套件中的其他函式簡化了針對 slice 和 map 的操作。在 `slices` 套件中的 `Insert`、`Delete`，和 `DeleteFunc` 函式可避免開發者編寫一些意外複雜的 slice 處理程式碼。`maps.Clone` 函式利用 Go runtime 來更加速地建立 map 淺複本。你將在第 317 頁的「只執行程式一次」中學習 `sync.OnceValue` 和 `sync.OnceValues`，它們使用泛型來建構只會呼叫一次並回傳一個或兩個值的函式。建議你優先使用這些套件的函式，而不是自行編寫。標準程式庫的未來版本可能加入更多利用泛型的新函式和型態。

未來解鎖的功能

泛型可能成為其他未來功能的基礎。其中一種可能的功能是 *sum types*（和型態）。就像型態元素被用來指定可以替代型態參數的型態一樣，它們也可以用於可變參數裡的介面，這將啟用一些有趣的功能。目前，Go 無法處理一種 JSON 的常見情況：某個欄位可能是單一值或值的串列。即使有泛型，處理這種情況的唯一方法是使用 `any` 型態的欄位。Go 加入和型態後，你可以建立介面來指定一個欄位只能是字串或字串 slice。如此一來，型態切換（type switch）可以完整列舉每一個有效型態，提升型態安全性。這種指定型態的有限集合的功能，讓許多現代語言（包括 Rust 和 Swift）能夠使用和型態來表示列舉（enums）。由於目前 Go 的列舉功能比較弱，這或許是一個有吸引力的解決方案，但評估和探索這些想法需要時間醞釀。

習題

你已經瞭解泛型的工作原理了，試著應用它們來解決以下問題。解答位於第 8 章版本庫（*https://oreil.ly/E0Ay8*）的 *exercise_solutions* 目錄裡。

第十一章

Go 工具

程式語言並非遺世獨立，為了讓語言提供實際的幫助，開發者必須透過工具的協助，來將原始碼轉換成可執行文件。Go 的目的是解決現代軟體工程師面臨的問題，並幫助他們建立高品質的軟體，因此 Go 工具組經過精心設計，簡化了在其他開發平台上往往難以執行的工作，包括組建、格式化、更新、驗證、分發等程序，甚至改進了使用者安裝程式碼的做法。

我已經介紹許多 Go 的同綑工具了，包括 `go vet`、`go fmt`、`go mod`、`go get`、`go list`、`go work`、`go doc` 和 `go build`。`go test` 為測試提供非常廣泛的支援，我們會在第 15 章單獨介紹它。這一章要介紹讓 Go 的開發過程如此美好的其他工具，它們有的來自 Go 團隊，有的來自第三方。

使用 go run 來試驗小型程式

Go 是一種編譯型語言，這意味著在執行 Go 程式碼之前，你必須將它轉換成可執行檔。相較之下，直譯型語言（例如 Python 或 JavaScript）可讓你快速地編寫一個腳本來測試想法並立即執行。快速的回饋循環非常重要，因此 Go 透過 `go run` 命令來提供類似的功能，讓你可以在一個步驟之中組建並執行程式。我們回到第 1 章的第一個程式，將它放入一個名為 *hello.go* 的檔案中：

```
package main

import "fmt"

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

（你也可以在第 11 章版本庫（*https://oreil.ly/Z_Fpg*）的 *sample_code/gorun* 目錄裡找到這段程式。）

儲存檔案後，使用 `go run` 命令來組建並執行它：

```
go run hello.go
Hello, world!
```

在執行 `go run` 命令之後，你會發現目錄內沒有任何二進制檔案，在目錄裡的唯一檔案是你剛才建立的 *hello.go* 檔案。可執行檔案跑（go）哪裡去了？（不是要故意使用雙關語）

`go run` 命令確實會將你的程式碼編譯成二進制檔案，不過是在一個臨時的目錄裡組建那一個二進制檔。`go run` 命令會在那個臨時目錄裡組建二進制檔案，在該目錄執行二進制檔案，並在程式結束後刪除該二進制檔案。因此，`go run` 命令適合用來測試小型程式，或將 Go 當成腳本語言來使用。

當你想將 Go 程式當作腳本來運行，並立即執行原始碼時，可使用 `go run`。

使用 go install 來加入第三方工具

雖然有些人選擇將 Go 程式當成預先編譯的二進制檔案來發布，但你也可以透過 `go install` 命令來組建以 Go 編寫的工具的原始碼，並安裝到你的電腦上。

正如你在第 258 頁的「發布你的模組」中看到的，Go 是用模組的原始碼版本庫來識別它們的。`go install` 命令接受一個引數，該引數是模組原始碼版本庫裡的主套件路徑，尾隨 `@` 和你想要使用的工具版本（如果你只想要取得最新版本，可使用 `@latest`）。然後它會下載、編譯，並安裝該工具。

第十二章

Go 的並行

並行（*concurrency*）是一個電腦科學術語，它的意思是將單一處理程序拆成獨立的元件，並定義如何讓那些元件安全地共用資料。許多語言都透過程式庫來提供並行功能，它們使用作業系統級的執行緒，藉著透過資料鎖的獲取（acquire lock）來共用資料。Go 不一樣，它的主並行模型（可說是 Go 最具代表性的功能）是以 Communicating Sequential Processes（CSP）作為基礎。這種並行風格是 Tony Hoare（他是快速排序演算法的發明者）在 1978 年的一篇論文中提出的（*https://oreil.ly/x1IVG*）。用 CSP 實作的模式與標準模式的效果一樣強大，但是容易理解許多。

本章將快速介紹 Go 並行的核心功能：goroutine、channel 與 `select` 關鍵字，然後討論一些常見的 Go 並行模式，並介紹一些比較適合採用低階技術的情況。

使用並行的時機

首先是個警告聲明：請必先確定使用並行對你的程式而言是有幫助的。當 Go 新手開始試用並行時，他們往往經歷一系列的階段：

1. 這個功能太棒了，我想要把所有東西都放在 goroutine 裡面！
2. 程式並沒有比較快，所以我要幫 channel 加入緩衝區（buffer）。
3. 我的 channel 塞住了，而且我被鎖死了。我要使用具有很大的緩衝區的 channel。
4. 我的 channel 依然塞住，那我改用 mutex 好了。
5. 算了，我放棄並行了。

goroutine

goroutine 是 Go 的並行模式的核心概念。為了說明 goroutine，我們要來定義一些術語。第一個術語是*程序*（*process*）。程序是在電腦的作業系統上運行的程式（program）實例。作業系統會將一些資源（例如記憶體）分配給程序，並確保其他程序不能操作它們。程序是由一或多個*執行緒*（*thread*）組成的。執行緒是一個執行單位，作業系統會幫它指定一段運行時間。在一個程序裡面的執行緒都共用同樣的資源。CPU 可以同時執行一或多個執行緒的命令，取決於它有多少顆核心。作業系統會安排執行緒待在 CPU 裡的時間，以確保每一個程序（以及程序內的每一個執行緒）都有機會可以運行。

goroutine 可以視為由 Go runtime 管理的輕量級執行緒。當 Go 程式開始執行時，Go runtime 會建立一些執行緒，並且啟動一個 goroutine 來執行你的程式。你的程式建立的所有 goroutine，包括最初的那一個，都會被 Go runtime 排程器（scheduler）自動指派給這些執行緒，就像作業系統在 CPU 核心之間調度執行緒一樣。雖然這看起來是多餘的工作，因為底層的作業系統已經有一個排程器負責管理執行緒與程序了，但是它有幾個好處：

- 建立 goroutine 的速度比建立執行緒的速度快很多，因為沒有建立作業系統等級的資源。
- goroutine 的初始堆疊比執行緒堆疊更小，而且可以隨需擴展，所以 goroutine 更具記憶體效率。
- 在 goroutine 之間切換比在執行緒之間切換更快，因為它是在程序裡面發生的，避開（相對）緩慢的作業系統呼叫。
- goroutine 排程器可以將它的決策最佳化，因為它是 Go 程序的一部分。排程器可和網路輪詢器（poller）合作，偵測 goroutine 是否被 I/O 塞住，進而取消排程。它也會和記憶體回收程序整合，確保分配給所有 Go 程序的所有作業系統執行緒的工作負擔是平衡的。

這些優點使得 Go 程式可以同時產生上百、上千，甚至上萬個 goroutine。如果你在具備原生執行緒機制的語言裡面啟動上千個執行緒，程式的速度將會大幅下降，近乎停滯。

如果你想要瞭解排程器如何執行工作，可觀看 Kavya Joshi 在 2018 年的 GopherCon 裡的演說「The Scheduler Saga」（*https://oreil.ly/879mk*）。

channel

goroutine 使用 channel 來溝通。如同 slice 和 map，channel 是用 `make` 函式來建立的內建型態：

```
ch := make(chan int)
```

channel 是參考型態，與 map 一樣。當你將 channel 傳給函式時，你傳遞的是 channel 的指標。同樣如同 map 和 slice，channel 的零值是 `nil`。

讀取、寫入與緩衝

你要使用 `<-` 運算子來與 channel 互動。若要讀取 channel，將 `<-` 運算子放在 channel 變數的左邊，若要寫入 channel，則是放在它的右邊：

```
a := <-ch // 從 ch 讀值，並將它指派給 a
ch <- b   // 將 b 的值寫入 ch
```

被寫入 channel 的每一個值都只能被讀取一次。如果有多個 `goroutine` 從同一個 channel 讀取資料，被寫入該 channel 的值只會被其中一個 goroutine 讀取。

同一個 goroutine 幾乎不會對著同一個 channel 進行讀取和寫入。當你將一個 channel 指派給一個變數或欄位，或將它傳給函式時，請在 `chan` 關鍵字的前面使用箭頭（`ch <-chan int`）來指示 goroutine 只讀取 channel。在 `chan` 關鍵字的後面使用箭頭（`ch chan<- int`）來說明該 goroutine 只寫入 channel。這樣做可讓 Go 編譯器確保一個 channel 只被一個函式讀取或寫入。

在預設情況下，channel 是無緩衝區的（*unbuffered*）。每次 goroutine 寫入一個打開的、無緩衝區的 channel 時，它都會暫停，直到有另一個 goroutine 讀取同一個 channel 為止。同理，每次 goroutine 讀取一個打開的、無緩衝區的 channel 時，它會暫停，直到另一個 goroutine 寫入同一個 channel 為止。也就是說，你必須至少使用兩個並行運行的 goroutine 才能對一個無緩衝區的 channel 進行寫入或讀取。

Go 也有有緩衝區（*buffered*）的 channel。這種 channel 會緩衝有限次數的寫入操作而不會塞住。如果緩衝區在任何 goroutine 讀取 channel 之前滿了，接下來對 channel 進行寫入的 goroutine 都會暫停，直到 channel 被讀取為止。如同對緩衝區已滿的 channel 進行寫入會塞住，讀取緩衝區清空的 channel 也會塞住。

這帶來一個問題：如何讓 Done channel 回傳一個值？這個動作是透過 *context* 取消來觸發的。在 main 裡，我們藉著使用 context 套件內的 WithCancel 函式來建立一個 context 和一個 cancel 函式。接著使用 defer 在 main 函式退出時呼叫 cancel。這會關閉 Done 回傳的 channel，因為被關閉的 channel 一定回傳一個值，確保運行 countTo 的 goroutine 會退出。

使用 context 來終止 goroutine 是很常見的模式。它可讓你根據呼叫堆疊裡較早的函式中的某些條件來停止 goroutine。第 369 頁的「取消」將告訴你如何使用 context 來確定一個或多個 goroutine 該關閉了。

決定何時該使用有緩衝區與無緩衝區的 channel

在 Go 的並行中，最複雜的技能，是決定何時該使用有緩衝區的 channel。在預設情況下，channel 是無緩衝區的，它們很容易理解：有一個 goroutine 進行寫入，並等待另一個 goroutine 取得它的成果，很像接力賽跑用的接力棒。有緩衝區的 channel 複雜許多，你必須選擇大小，因為有緩衝區的 channel 絕不使用無限大的緩衝區。要正確地使用有緩衝區的 channel，你必須處理兩種情況：緩衝區已滿的情況，以及進行寫入的 goroutine 為了等待進行讀取的 goroutine 而塞住的情況。那麼，如何正確地使用有緩衝區的 channel？

有緩衝的 channel 的情況很微妙，有緩衝區的 channel 適合在你已經知道你啟動了多少 goroutine、你想要限制以後啟動的 goroutine 數量，或你想要限制排隊等待的工作數量時使用。

當你想要從一組已被你啟動的 goroutine 收集資料，或想要限制並行的使用時，很適合使用有緩衝區的 channel。它們也有助於管理系統排隊等候的工作量，防止你的服務落後進度，與不堪重負。接下來的幾個例子將告訴你如何使用它們。

在第一個例子裡，我們要處理一個 channel 的前 10 個結果。為此，我們要啟動 10 個 goroutine，每一個都將它的結果寫入有緩衝區的 channel：

```
func processChannel(ch chan int) []int {
    const conc = 10
    results := make(chan int, conc)
    for i := 0; i < conc; i++ {
        go func() {
            v := <- ch
            results <- process(v)
        }()
    }
```

第十六章

惡龍禁地：Reflect、Unsafe 與 Cgo

已知世界的邊緣地帶令人毛骨聳然，古代的地圖會在未開拓的蠻荒地區畫上龍和獅子，在之前的內容中，我強調 Go 是一種安全的語言，它使用 typed 變數來說明你正在使用哪一種資料，並使用記憶體回收機制來管理記憶體，就連它的指標也是人畜無害的，因為你無法像在 C 與 C++ 裡那樣濫用它們。

上面說的都是真的，在你將會寫出來的大多數 Go 程式中，你可以相信 Go runtime 會保護你。但是 Go 仍然有一些密道，有時 Go 程式需要冒險進入一些未知領域。在這一章，我們要看看如何處理無法用一般的 Go 程式來解決的情況，例如，當資料的型態無法在編譯期決定時，你可以使用 `reflect` 套件支援的 reflection 來與資料互動，甚至建構資料。當你需要利用 Go 的資料型態的記憶體布局時，你可使用 `unsafe` 套件。如果你想用的功能只能在 C 的程式庫找到，你可以用 `cgo` 來呼叫 C 程式碼。

你可能想問，為何在一本以 Go 新手為對象的書裡介紹這些進階概念？原因有二。首先，開發者在尋找問題的解決方案時，可能會發現他們一知半解的技術（並將它複製貼上），所以在你將可能造成問題的進階技術加入你的碼庫（codebase）之前先稍微瞭解它們比較好。第二，這些工具都很有趣，因為它們可以讓你做 Go 通常做不到的事，稍微玩玩它們，看看你能夠做哪些事情，也是一件有趣的事情。