

TS

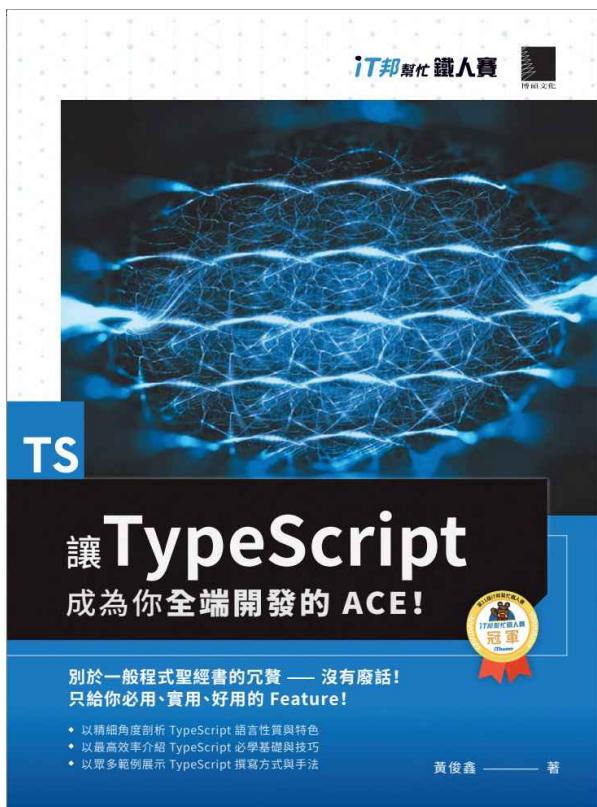
讓 TypeScript 成為你全端開發的 ACE!



別於一般程式聖經書的冗贅 — 沒有廢話！
只給你必用、實用、好用的 Feature！

- ◆ 以精細角度剖析 TypeScript 語言性質與特色
- ◆ 以最高效率介紹 TypeScript 必學基礎與技巧
- ◆ 以眾多範例展示 TypeScript 撰寫方式與手法

黃俊鑫 ————— 著



本書如有破損或裝訂錯誤，請寄回本公司更換

國家圖書館出版品預行編目(CIP)資料

讓 TypeScript 成為你全端開發的ACE！ / 黃俊鑫著.

-- 初版. -- 新北市 : 博碩文化, 2020.06

面； 公分. -- (IT邦幫忙鐵人賽系列書)

ISBN 978-986-434-489-5(平裝)

1.Java Script (電腦程式語言)

2.TypeScript (電腦程式語言)

312.932J36

109006427

Printed in Taiwan



博碩粉絲團

歡迎團體訂購，另有優惠，請洽服務專線
(02) 2696-2869 分機 238、519

作 者：黃俊鑫

責任編輯：蔡瓊慧

董事長：陳來勝

總編輯：陳錦輝

出版：博碩文化股份有限公司

地址：221新北市汐止區新台五路一段112號10樓A棟

電話(02) 2696-2869 傳真(02) 2696-2867

發 行：博碩文化股份有限公司

郵撥帳號：17484299 戶名：博碩文化股份有限公司

博碩網站：<http://www.drmaster.com.tw>

讀者服務信箱：dr26962869@gmail.com

訂購服務專線：(02) 2696-2869 分機 238、519

(週一至週五 09:30~12:00；13:30~17:00)

版 次：2020 年 06 月初版一刷

建議零售價：新台幣 650 元

I S B N : 978-986-434-489-5

律師顧問：鳴權法律事務所 陳曉鳴律師

商標聲明

本書中所引用之商標、產品名稱分屬各公司所有，本書引用純屬介紹之用，並無任何侵害之意。

有限擔保責任聲明

雖然作者與出版社已全力編輯與製作本書，唯不擔保本書及其所附媒體無任何瑕疵；亦不為使用本書而引起之衍生利益損失或意外損毀之損失擔保責任。即使本公司先前已被告知前述損毀之發生。本公司依本書所負之責任，僅限於台端對本書所付之實際價款。

著作權聲明

本書著作權為作者所有，並受國際著作權法保護，未經授權任意拷貝、引用、翻印，均屬違法。

推薦序

＊ 每個想邁向進階的前端開發者都應該要有的本書

為什麼要學習 TypeScript ？

相信網頁開發者都聽過一句話，JavaScript 是個號稱「世界上最被人誤解的程式語言」，但是得 JavaScript 得（前端）天下。

而我認為 JavaScript 本質上是個易用難精的程式語言。不管你有沒有程式相關的經驗，可能用一個週末花點時間就可以快速寫出一些成果，相較於其他程式語言來說，它的門檻是相對低的。

雖然容易開始，但越想深入就會發現它越難。

而 TypeScript 站在 JavaScript 的肩膀上，補強了 JavaScript 的不足之處，除了新增了一部分語法外，本質上仍是與原生的 JavaScript 沒有太大的不同，卻提供了更多好處。像是提供了強型別的語言特性，增加了程式碼的可讀性和可維護性，重構時可自動對程式碼做出型別推論 等。

更重要的是，現在前端三神兵 VAR（Vue/Angular/React）都對 TypeScript 的支援越來越好，甚至 Vue.js 3.0 的底層核心都透過 TypeScript 進行改寫，你還有什麼理由拒絕學習它？

正如 Max 在本書所說，技術的需求以及「進化」才是關鍵，TypeScript 在編譯的過程就可以提前發現程式碼的問題，而不用等到在瀏覽器執行的階段才去打開主控台，尋找錯誤的程式碼，減少犯錯的機會，降低加班的時數。

截至目前為止，在台灣的資訊書界，如果不算中國大陸出品的簡體書籍以及國外翻譯書籍，至今還沒有一本 TypeScript 的專書。

很開心看到這次 Max 參加第 11 屆 iT 邦幫忙鐵人賽冠軍的大作《讓 TypeScript 成為你全端開發的 ACE！》能夠付梓出版，除了為台灣本土資訊業界帶來一注活水，裡頭滿滿的範例以及各種貼心的小提示，手把手帶領讀者學習 TypeScript，相信一定不會讓你失望。

Kuro

Vue.js Taiwan 社群主辦人
《0 陷阱！0 誤解！8 天重新認識 JavaScript！》作者

✿ 前端發展與時俱進

前端生態圈的發展瞬息萬變，當初 JavaScript 的發明者——Brendan Eich 壓根也沒想到短短 10 天內寫出來的 JavaScript 的原型竟然可以發展到今日如此的樣貌——除了可以撰寫各種瀏覽器的動畫外，還可以使用 NodeJS 撰寫後端的程式碼腳本，並且還有專門的技術委員會（TC39，Technical Committee 39）制定 JavaScript 正式的語法標準，並且正名為 ECMAScript。

隨著應用程式開發的需求變化，儘管 JavaScript 相較於 C 或者是 Java，算是淺顯易懂的語言，但在開發上仍然有幾點不方便的地方：

- JavaScript 為弱型別語言（Weakly Typed），也就是說，任何變數（Variable）可以被指派或被不同型別的值（Value）進行覆蓋的動作，而函式（Function）或方法（Method）也不需要檢查傳入的引數（Arguments）之型別；然而，若沒有型別的檢測機制，想要除錯時，通常會暴力地一行一行掃過程式碼，進行人工對照，亦或者是執行 JavaScript 的程式碼查看吐出來的錯誤堆疊（Error Stack），免不了會花很多時間在執行程式碼的層面上除錯。
- 近年來 ES6 的標準下逐漸開始引進了物件導向相關的語法規範，但 JavaScript 本質是原型導向（Prototype-Oriented）的語言，因此在類別（Class）相關的語法方面沒有很完整。嚴格來說，ES6 Class 目前也只能算是語法糖（Syntax Sugar）的一種，並非 JavaScript 原生的語言特色。
- 儘管現在的 ECMAScript 標準讓 JavaScript 擁有很多方便的語法與功能，但不同的環境（如：瀏覽器）不全然會支援所有的語法，因此使用這些新語法標準時會搭配編譯器（Compiler，又或者是轉譯器 Transpiler）負責將語法轉換為 ES5 或更低版本的語法，著名的編譯器就是 Babel；然而，不管是前後端的開發環境的架設通常都很複雜。

當然，還有很多使用 JavaScript 開發的種種零碎心得與開發者經驗方面的主題可以討論，不過作者認為本書將要講的內容——也就是 TypeScript 這個語言，可以解決大部分遇到的上述情形。

基本上，任何開發者開發應用程式時，除了功能外，最重要的莫過於使用者經驗，也就是所謂的 UX (User Experience)；同理，軟體開發工具的發展上，各種不同的工具推陳出新，目的是解決開發過程上的不便，這也就是所謂的 DX (Developer Experience)，而 TypeScript 就是在這樣的背景下出現的。不過在學習 TypeScript 前，作者誠心提醒一下讀者：

工具的發明並不是為了發明而發明，通常一定是要解決某些特定問題才會發展出來的；因此並不是要讀者一看到新的工具就把它學起來，而是碰到了問題，剛好需要時才會去學習它。

重點在於能夠辨別現在你所遇到的問題是什麼以及能不能夠用適當的工具與方法解決，能夠把解法簡單化就儘量去做。

† 本書的內容與方向（還有更多作者的廢話）

回歸本書的主題，作者將要帶領你探索 TypeScript 這門語言，介紹它的種種好處以及比較容易卡關的地方。由於 TypeScript 是建立在 JavaScript 的基礎之上，因此會建議讀者本身就有使用過 JavaScript 經驗，並且至少了解 JavaScript 基本的語法機制，像是：

- JavaScript 的基礎語法包含：變數、基礎資料型態、函式、控制流程、迴圈敘述式等等。
- JavaScript 的進階概念包含：原始資料型態 (Primitives) 與物件 (Objects) 之差別、變數作用域 (Global/Local Scope)、變數或函式提升 (Hoisting)、閉包 (Closure) 的概念、JavaScript 的 this 關鍵字的概念與使用等等。

以上所列的特點在本書的程式碼中，有些出現得頻繁，然而有些儘管較少，但是擁有以上列出的這些基礎將會使得理解本書的範例程式碼的內容更加容易、快速。當然，如果需要補充，作者會在本書會提供額外的註解或參考連結讓讀者自行去發掘。

就語法層面以及 TypeScript 程式的編寫模式（Programming Paradigm），對於任何擁有物件導向（Object-Oriented）程式背景的開發者在學習上會比較友善些，所以擁有 C# 或 Java 等語言背景的讀者可能會覺得某些部分非常簡單，甚至是把這本書當作是物件導向的入門書也不為過，但也僅限於討論類別語法部分以及基礎的物件導向程式設計概念。

此外，TypeScript 的根本是建立在 JavaScript 基礎之上，而 JavaScript 看似入門很簡單，但是要注意的細節也很多，像是：

- null 與 undefined 之間的差別為何？null 為何是 object（也就是物件）類型，而非 null 本身；相較起來，undefined 就是 undefined 類型？
- 判斷式之判斷組合只能用三個字形容：「非常亂」。比如數字 0 偏向於 false 這個布林值（Boolean），空字串也屬於 false 的範疇；然而，只要是物件，不管看起來是不是空的，一律都是偏向於 true 這個值，包含空物件 {} 或空陣列 []；另外，代表非數字的概念的 NaN，儘管 `NaN === NaN` 這一行看起來出現為 true 的結果，但遺憾的是，它的輸出結果是 false，你必須要用 `Number.isNaN` 這個方法來檢測值是不是等於 NaN。
- JavaScript 的作用域規範跟其他大部分的物件導向語言有很大的差異，基本上作者是直接把 JavaScript 函式的使用規則看待成學習另一門新的語言的感覺也不為過。反正只要是談到 JavaScript 的作用域這方面的設計，作者本身感到很無言，但這是無可避免地必須學習的檻。

- 由於瀏覽器背後執行 JavaScript 語言的引擎（最常見的就是 Chrome 的 V8 引擎）運作方式為單線程（Single-threaded）¹，所以了解同步與非同步的執行方式與差異是非常重要的。

以上的部分是要敬告來自非 JavaScript 語言背景的開發者們，JavaScript 看似是很單純的語言，沒有太多很複雜的寫法，不過想要挑出些細節，事實上也挺多的。

反觀如果本身是 JavaScript 語言背景摧殘過後的開發者們，不僅僅只是 TypeScript 招牌的型別系統，最主要也可以從本書學習到的東西是更多物件導向相關細節的內容，尤其物件導向的概念在江湖上被傳頌過很多次，免不了會遇到有些觀念在發展過程中被曲解掉，甚至不知覺地寫出反面模式（Antipattern），也就是會傷害到程式碼的架構品質的模式。而相對反面模式的東西就是俗稱的設計模式（Design Patterns）。

不過由於本書的規劃上主要以學習 **TypeScript** 的基礎語法與概念為主，並不會帶到設計模式這一類的主題，物件導向的進階概念頂多會講到很基礎的設計原則，但是足夠使讀者能夠自行從各種資源或網路上查找、學習、吸收並且驗證更多觀念。（重點是要培養讀者主動驗證知識的正確性的習慣）

1. 瀏覽器執行 JavaScript 程式碼的部分，如果以 Google Chrome 為範例的話，就是所謂的 V8 引擎；而程式碼是單線程的方式執行的，意思就是說，程式碼一定不會在極微小瞬間讀兩行以上的程式碼，因此絕對不會發生同時有兩個以上的來源去修改到同一位址下存取的資料（譬如：同時修改到同一個 JavaScript 的物件）。然而，NodeJS 雖然也是架構在 V8 引擎的基礎上，只是將瀏覽器操作的 DOM 或者是相關的 Web API 替換成操作作業系統下的各種架構，例如檔案系統（File System）；然而，我們可以在 NodeJS 啟用名為 Child Process 這個東西，這樣就可以跳脫單線程的執行限制，所以嚴格來說，到底 JavaScript 是單線程還是多線程（Multi-threaded）還是得根據執行 JavaScript 的環境而定。

畢竟 JavaScript 本身不是物件導向為主的語言，引入初階的物件導向相關語法的 ES6 也才發展幾年的時間而已（儘管才 4 到 5 年左右，但這樣還是算很短了！），但類別相關語法真的是很不完備，並且還要等待 ECMAScript 標準的釋出² 與瀏覽器的實作簡直會等到天荒地老；TypeScript 的發展補足了這一切，擁有完備的類別相關的語法，並且還有介面（Interface）相關的語法系統，簡直是學習物件導向觀念的絕佳工具。因此本身是只有寫過 JavaScript 的讀者，反過來說，學完 TypeScript 後想要嘗試看看稍微複雜些的 Java 或 C# 類型的語言，也可以藉由本書所傳達的概念向外發展跳槽出去唷！

不過作者相信前端的發展將使得 TypeScript 有極高機率成為常用甚至是必學的工具。在這樣的潮流下，作者也會儘量將學習過程簡易化，不過還是會稍微嚴謹些，並且帶出一些實質的應用與挑戰讓讀者能夠更享受寫程式的過程，畢竟寫得過程開心最重要嘛！

另外，這也是第一本由台灣人寫出的 TypeScript 相關書籍，能夠作為推動台灣前端發展的一小步也實在是深感開心。

話不多說～準備好你的電腦或者是筆記，開始這趟征服 TypeScript 的旅程吧～！

2 制定 ECMAScript 標準的 TC39 委員會，通過一項新的提議（Proposal）需要經歷四個階段（嚴格來說有五個）—提案（Proposal）、草稿（Draft）、候選（Candidate）以及釋出（Finished）。有興趣的讀者可以參考 The TC39 Process：<https://tc39.es/process-document/>。

目錄

Contents

► Part I TypeScript 基礎篇

01 TypeScript 的發展與概論

1.1	TypeScript 簡介	1-1
1.2	TypeScript 可以解決什麼樣的問題？	1-4
1.3	學習 TypeScript 的更多好處.....	1-19
1.4	征途路上總是也有跌跌撞撞的時候	1-24
1.5	旅程中的第一小步	1-26

02 TypeScript 型別系統概論

2.1	型別系統的兩大基柱——型別的推論與註記.....	2-1
2.2	型別註記——「註記」與「斷言」的差異性.....	2-5
2.3	綜觀 TypeScript 型別種類.....	2-21

03 深入型別系統 I 基礎篇

3.1	深潛之前的準備.....	3-1
3.2	原始型別 Primitive Types	3-5
3.3	JSON 物件型別 JSON Object Type	3-14
3.4	函式型別 Function Object Type	3-30
3.5	陣列型別 Array Object Type	3-46
3.6	明文型別 Literal Type	3-50

04 深入型別系統 II 進階篇

4.1 元組型別 Tuple Type	4-1
4.2 列舉型別 Enum Type	4-5
4.3 可控索引型別與索引型別 Indexable Type & Index Type	4-18
4.4 複合型別 Composite Type.....	4-27
4.5 Never 型別.....	4-37
4.6 Any 與 Unknown 型別.....	4-42

05 TypeScript 類別基礎

5.1 物件導向基礎概論 OOP Fundamentals	5-1
5.2 TypeScript 類別語法 Class Syntax.....	5-11
5.3 型別系統中的類別	5-56

06 TypeScript 介面

6.1 介面的介紹 Introduction to Interface	6-1
6.2 介面的彈性 Flexibility of Interface.....	6-5
6.3 註記與實踐介面.....	6-13
6.4 詭異的 TypeScript 函式參數型別檢測機制	6-21
6.5 型別化名 V.S. 介面	6-25

07 深入型別系統 III 泛用型別

7.1 泛用型別的介紹 Introduction to Generic Types.....	7-1
7.2 型別泛用化	7-4
7.3 型別參數額外功能	7-14

08 TypeScript 模組系統

8.1 ES6 Import / Export 模組語法	8-1
8.2 命名空間 Namespaces	8-11
8.3 型別宣告 Type Declaration	8-19
8.4 引入純 JavaScript 套件的流程.....	8-23

► Part II TypeScript 應用篇

09 物件導向進階篇章

9.1 物件導向進階概論	9-1
9.2 物件導向設計原則 SOLID Principles.....	9-10
9.3 物件導向延伸應用	9-22

10 常用 ECMAScript 標準語法

10.1 ES6 解構式 Destructuring.....	10-1
10.2 ES7 匯集 - 展開操作符 Rest-Spread Operator.....	10-6
10.3 ES6 Set 與 Map 資料結構	10-11
10.4 ES10 非強制串接操作符 Optional Chaining Operator.....	10-19
10.5 ES10 空值結合操作符 Nullish Coalescing Operator.....	10-21

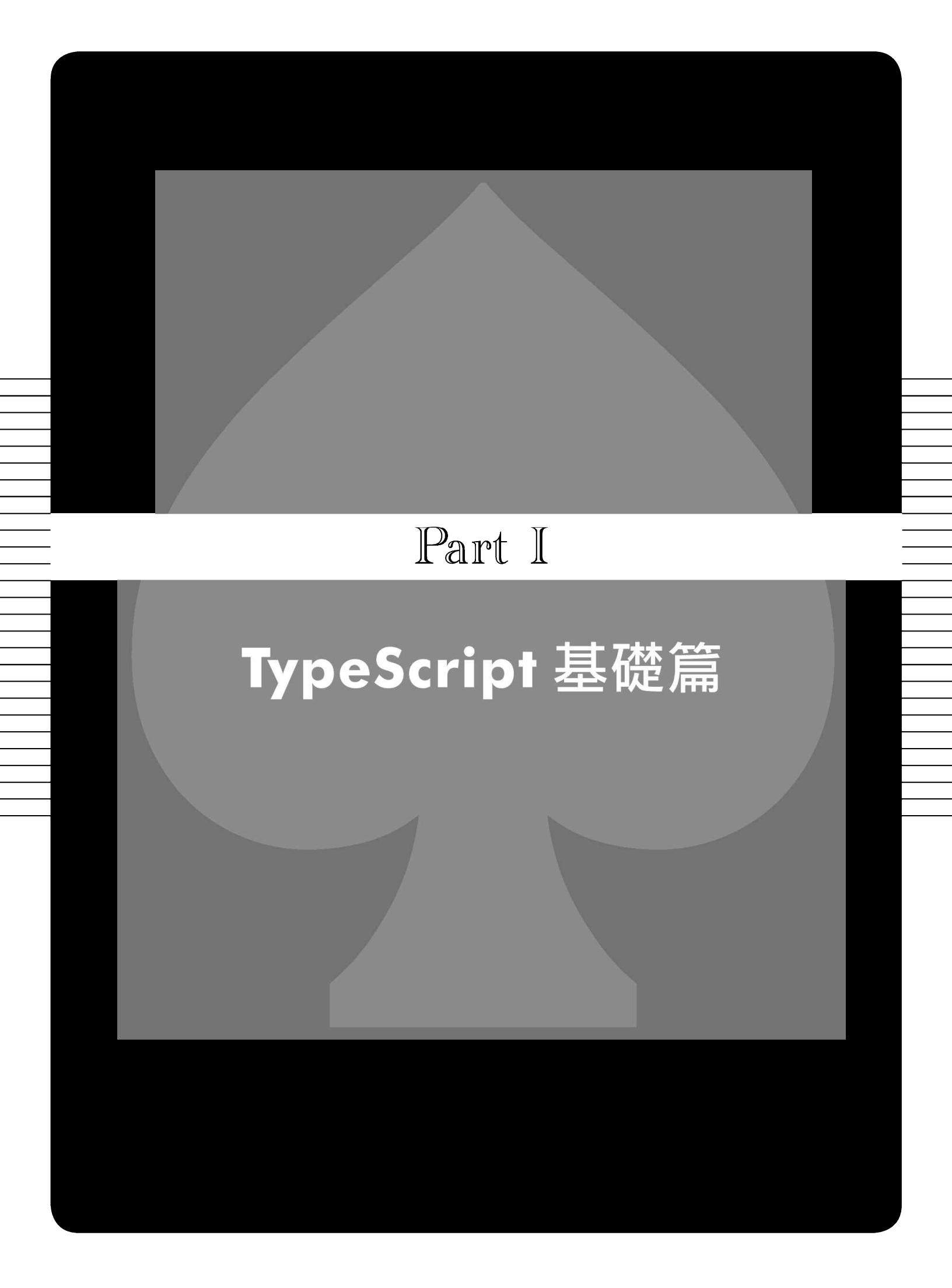
11 常用 ECMAScript 標準語法 非同步程式設計篇

11.1 同步與非同步的概念	11-1
11.2 ES6 Promise 物件	11-6
11.3 ES7 非同步函式 Asynchronous Functions.....	11-17

12 TypeScript 裝飾子

12.1 裝飾子的簡介 Introduction to Decorators.....	12-2
12.2 裝飾子種類	12-8
12.3 裝飾子的運用	12-18

A 解答篇



Part I

TypeScript 基礎篇

01

TypeScript 的發展與概論

► 1.1 TypeScript 簡介

TypeScript 其實早在 2012 年十月的時候發佈了第一版本（版號為 0.8），由微軟（Microsoft）釋出的一款開源的程式語言，主要的開發者為來自丹麥的 Anders Hejlsberg¹，他也是 C# 最初語言架構的設計者，所以 TypeScript 與 C# 在型別系統方面有類似的設計以及對於類別方面的語法支援充足性。

TypeScript 是建立在 JavaScript 的基礎上，除了擁有如其名的型別系統（**Type System**）外，也擁有額外的語法。某些語法是 TypeScript 本身的特色，像是靜態型別方面的宣告與使用；而有些則是符合 ECMAScript 的標準，像是 ES6 箭頭函式（Arrow Functions）、解構式（Destructuring）等等。想知道 TypeScript 對於 ECMAScript 的支援程度，可以參考 ECMAScript Compatibility Table²。

1 <https://twitter.com/ahejlsberg>。

2 <https://kangax.github.io/compat-table/es6>。

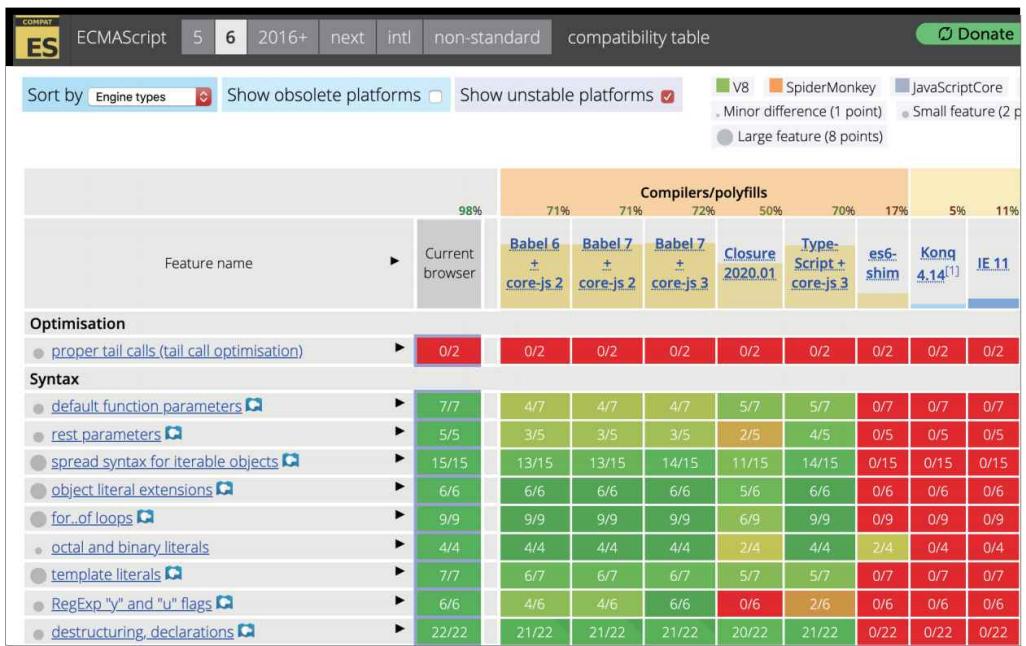
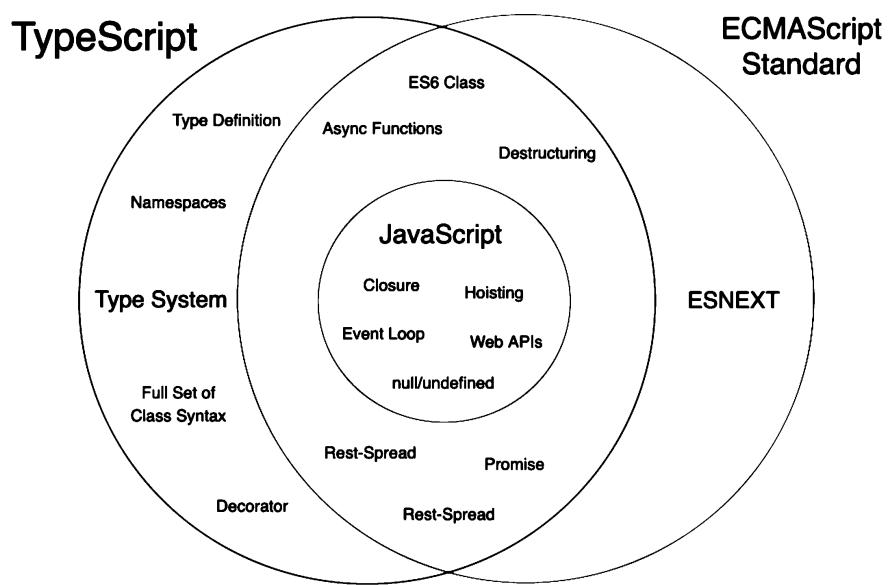


圖 1-1 ECMAScript 相容性列表

基本上，TypeScript 與 ECMAScript 差別就在於——TypeScript 對於支援 ECMAScript 語法標準的程度，僅此而已。另一方面，TypeScript 與 JavaScript 的關係則是——**TypeScript 在語法的規範上是 JavaScript 的超集合 (Superset)**，意即只要是 JavaScript 的程式碼，經過 TypeScript 編譯器還是依然可以編譯成功——但還是有一個小小的前提——該程式碼在型別系統的監控下是沒有任何問題的。(參見圖 1-2)



Illustrated by Maxwell Alexius twitter.com/AlexiusMaxwell

圖 1-2 TypeScript 與 JavaScript 以及 ECMAScript 之間的關係

從圖 1-2 大略可以得知，TypeScript 包含的範圍就是 JavaScript 本身，但只有包含部分的 ECMAScript 標準；通常 TypeScript 還沒支援的 ECMAScript 標準大部分都是所謂的 ESNEXT，也就是未來的 ECMAScript 的規範。

所以根據剛剛提到的情形，由於普通的 JavaScript 程式碼本身屬於 TypeScript 的一部分，基本上是可以直接被 TypeScript 編譯的：

```
const foo = 123;
const bar = 456;
console.log(foo + bar);
```

然而，並不是所有的 JavaScript 程式碼會通過 TypeScript 編譯器的型別檢測。以下這一段雖然也是很簡單的 JavaScript 宣告函式的程式碼，但是會因為 TypeScript 型別系統監測的關係，會出現一些警吶訊息：

```
function addition(x, y) {
    return x + y;
}
```

The screenshot shows a code editor with the following TypeScript code:

```
function addition(x, y) {
    return x + y;
}
```

A tooltip or callout box is overlaid on the code, containing the following text:

(parameter) x: any
Parameter 'x' implicitly has an 'any' type. ts(7006)

[Peek Problem](#) [Quick Fix...](#)

**addition 函式的宣告
出現的警吶訊息**

至於讀者要如何解決上面的問題（讀者可以先猜猜看～），以及判斷哪些時候需要使用 TypeScript 靜態型別系統的語法，這部分就是下一章主要介紹的東西喔～

► 1.2 TypeScript 可以解決什麼樣的問題？

1.2.1 使用 JavaScript 開發時最常遇到的問題

由於 JavaScript 具備動態（Dynamic）語言特性，相信每一位使用 JavaScript 開發的讀者們，不外乎在 Debug 時會重複以下這幾個步驟：

1. 直接執行程式碼，出錯就進行下一步。
2. 查看錯誤訊息與吐出的一大長串錯誤堆疊（Error Stack）。
3. 根據錯誤訊息，可能使用 `console.log` 之類的東西查問題等，並且修改程式碼。
4. 重複步驟 1.。

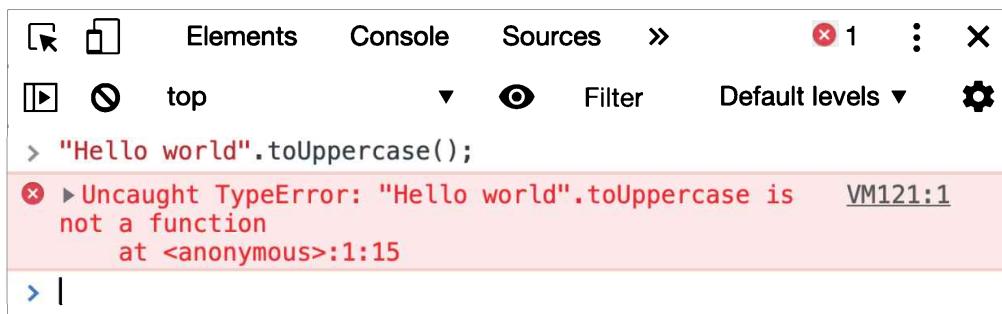
基本上，這是大部分使用動態語言會出現的狀況，因此如果是使用 Ruby、Python 等語言，通常也會是這樣 Debug 的（至於動態與靜態語言的差別，請參見條目 1.2.2）。

只要使用 JavaScript Debug 時，除非使用眼力掃整段程式碼來除 Bug，免不了必須執行程式碼並根據吐出之訊息來除 Bug。

另一個很常見的問題，尤其不管寫任何程式都一定會碰到的就是打錯字（Typo）的情形，比如：

```
"Hello world".toUpperCase();
```

這一行程式碼，相信讀者預期出來的結果就是把字串 "Hello world" 轉換成全部大寫——亦即 "HELLO WORLD"，但是讀者能夠知道這段真的會成功執行嗎？請看執行結果如圖 1-3。

圖 1-3 使用 `toUppercase` 方法結果

事實上，正確的寫法應該是要使用 `toUpperCase`，也就是那個 C 必須要大寫。不知道讀者是否有碰過類似的問題？如果是很明顯的拼錯字就算了，有時候出現大小寫分不清楚，亦或者是物件的性質（Object's Property）或方法的取名方式模糊到錯誤層出不窮，最後還得浪費幾分鐘時間上網查文件。查完還不是結束，我們的腦袋不可能記得所有的東西，因此在未來的某個時間點可能為了使用同個功能結果卻忘記，因此又得再重複查詢。

除非工具提供類似偵測變數、函式、方法名稱等機制，亦或者是通常編輯器會提供的自動補齊（Autocomplete）的功能等；否則，無可避免地，開發過程中出現錯字的機率非常高。

另外，使用 JavaScript 時，變數並沒有紀錄儲存之值的型別的功能，而呼叫函式或方法時，也不會檢查填入之參數（Parameter）的型別為何（除非你在函式或方法裡使用 `typeof` 或 `instanceof` 運算子檢測參數）。這種機制有好也有壞，好處在於變化性可以很大，舉一個很陽春的例子：

```
function plusOperation(x, y) {
  return x + y;
}

plusOperation(1, 2);           // => 3
plusOperation('Hello', 'World'); // => 'Hello World'
```

不管你填入的型別為何，函式都會直接將參數值代入進去。但害處也就藏在這裡，假設今天我們想要的行為單純是數字的加總（Sum），而有一個 API 的設計是會回傳一連串的數字，比如：

```
// 數字的加總函式
function sum(...numbers) {
    return numbers.reduce((acc, cur) => acc + cur, 0);
}

// 使用 Fetch API，該 API 會回傳一連串的數字
function fetchNumbers() {
    return fetch('some_api/numbers').then(response => response.json());
}

fetchNumbers()
    .then(data => sum(...data))
    .then(console.log);

// 假設執行結果印出：
// => 12345678910
```

假設最後印出來的結果是一長串看似是數字的資料，結果除錯時，發現該資料是字串。然而，程式照樣可以動作喔！所以嚴格來說這個程式並沒有發生任何錯誤，而是出現了非你所預期的狀況。

有時候程式執行過程並不會發生任何問題，但可能因為資料格式 / 型別 / 結構錯誤而導致程式出現超常行為（Unexpected Behaviour）。

作者曾遇過上述類似的問題，使用一些開放式的 JSON API 資料時，沒注意到該 API 輸出並且解析 JSON 的結果是——數字部分通通是以字串方式來表示。因此對該字串型別表示的數字做任何的處理會出現很莫名其妙的結果，而程式也不會通報你錯誤的發生，你得自己去抓出這個錯誤。

另外，還有一些很細微的問題，比如：使用如陣列（Array）的 `pop` 的方法，也就是將陣列的最後一個值取出來時，如遇到空陣列情形，在 JavaScript 可是會很安靜地略過，回傳 `undefined` 這個結果。

```
const arr = [1, 2, 3];
arr.pop(); // => 3
arr.pop(); // => 2
arr.pop(); // => undefined
```

```
arr.pop(); // => undefined 此時的陣列是空的
```

對照 Python 這一個性質類似的語言，如果遇到空陣列的狀況，使用 `pop` 方法可是會狠狠地丟出錯誤呢：

```
>>> arr = [1, 2, 3]
>>> arr.pop()
3
>>> arr.pop()
2
>>> arr.pop()
1
>>> arr.pop() # 此時的陣列是空的
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

有些讀者可能會想說：「我又不是來學 Python 或 C# 等其他語言的，為何作者還要提及這些不關乎 TypeScript 或 JavaScript 的東西呢？」

作者認為多去參考其他語言的設計除了可以單純比較差異外，某種程度上可以知道不同語言的設計理念為何。其中，以 Python 這種丟出錯誤的情形，可以對照該語言本身的設計理念（Zen of Python³）的某一句話：

“Error should never pass silently.”

至於什麼是 Zen of JavaScript，就請讀者自行好奇上網看看吧！（包準會看到很多莫名其妙的東西）總而言之，以上的重點是：

有時候任何程式上的行為會出現各種結果——包含呼叫函式或方法、與陌生的第三者服務接觸等，但我們可能會忘記要去處理掉這些邊緣情境（Edge Cases）。

（你是忘記了還是害怕想起來？）

3 https://en.wikipedia.org/wiki/Zen_of_Python。

剛剛舉的範例即是——我們都會使用這種陣列看似簡單的 `pop` 方法，但在操作時，有時卻忘記要去處理掉空陣列使用 `pop` 時會回傳 `undefined` 的這種情形——這就是一種邊緣情境的案例。

通常處理這種邊緣情境，一種就是用判斷敘述式（也就是常見的 `if...else...`）抓出來處理；另一種就是，如果預想到的邊緣情境過多，一一列舉太浪費時間時，我們乾脆使用 `try...catch...` 敘述式來解決也是一種方法。

»》 使用 JavaScript 開發時最常遇到的問題與麻煩點

1. 除錯時，除非用眼力暴力掃描程式碼，否則免不了執行程式的步驟
2. 沒有工具的輔助，單純寫程式，錯字機率本來就很高
3. 由於資料格式 / 型別 / 結構錯誤等，儘管程式執行過程沒錯誤，但出現非人類預期狀況
4. 忘記要處理邊緣情境相關的問題

1.2.2 動態語言 V.S. 靜態語言

從上一個條目我們得知了使用 JavaScript 開發時通常會遇到的問題。其中，有些是由於 JavaScript 具備動態的特性相對造成的結果，而這些問題八九成可以藉由 TypeScript 提供的型別系統來解決。不過請讀者別誤會，並非代表動態語言是不好的，只是能不能善用它的優點特性達到我們的需求、加速開發等。

回過頭來，講到後面就必須有個前提，讀者是否熟悉這幾個名詞呢？

- 靜態型別語言（Statically Typed）與動態型別語言（Dynamically Typed）
- 強型別語言（Strongly Typed）與弱型別語言（Weakly Typed）

我們就先從靜態與動態語言的差異來剖析一下：

»》 動態與靜態語言 Dynamically v.s. Statically Typed Language

1. 動態語言的特色為：在程式運行的狀態下，也就是在英文文章裡會看到的 Run-time 期間，任何變數是經由被代入值來判斷其型別；也就是說，變數的型別會依據存的值本身來判斷。

2. 靜態語言的特色則是：在程式正在編譯時，也就是所謂的 Compilation 期間，根據程式裡宣告的型別（幾乎都是用文字來表示，如 Int、Float 等）來監控型別的狀態。

如果以下面簡單的 JavaScript 程式碼為例子的話：

```
function addition(x, y) {  
    return x + y;  
}
```

可以看到 `addition` 函式要求輸入兩個參數並回傳兩個參數相加的結果。我們不會知道該函式被呼叫時填入之參數型別，必須要執行到的時候才會知道結果如何——而這個就是「動態」這個詞在這裡的意義。因此如果程式執行到下一行是：

```
addition(123, 456); // 運算結果為：數字 579
```

程式看到原來這兩個是數字，因此呼叫 `addition` 時，輸出結果出來是數字。另一方面，如果執行程式時看到的下一行是：

```
addition('123', '456'); // 運算結果為：字串 '123456'
```

執行過程中，看到兩個代入 `addition` 函式的參數變成字串型別，因此輸出結果則變成了字串型別。

好，理解原本的 JavaScript 的動態語言特性後，我們來看看 TypeScript 的部分與 JavaScript 在這方面的差異為何。首先，筆者先祭出簡單的型別註記（Type Annotation）的範例，詳細的語法內容與機制都會在第二章以後討論喔，不急不急～

```
function addition(x: number, y: number): number {  
    return x + y;  
}
```

以上的程式碼應該可以看出多了幾樣東西，我們用文字的方式說明，`addition` 函式必須填入兩個參數 `x` 與 `y`，其型別皆為數字——也就是 `number` 型別的值。另外，在函式的參數宣告的尾端，也就是：

```
function addition(/* 參數宣告 ... */): number { /* ... */ }
```

介於大括弧前的型別代表著 `addition` 函式的輸出必須為 `number` 型別的值。請試著回想剛剛提到的觀念——靜態語言的特色即是在編譯過程中，根據程式自身宣告之型別來進行檢測，而這些宣告通常都是以文字表示。如果仔細去觀察其他靜態型別的語言，作者以某段簡單的 C++ 程式碼為例：

```
int main()
{
    // 宣告三個變數，皆為 int 型別，也就是俗稱為整數的型別
    int num1 = 1;
    int num2 = 2;
    int sum;

    // 加總
    sum = num1 + num2;

    // 印出結果
    cout << num1 << " + " << num2 << " = " << sum;
    return 0;
}
```

以上的範例程式碼使用了 `int` 這個型別宣告，告訴 C++ 編譯器，變數 `num1`、`num2` 與 `sum` 皆為整數型別之值。另外，由於程式是在編譯過程中，並且還未產出可執行的檔案結果前，根據這些型別宣告來判斷程式內容是否有誤——所以連程式都還沒到動態執行的階段就已經在做型別分析了，而這就是「靜態」這個名詞的意義。

好的，那至於 TypeScript 到底又是如何呢？由於它有提供型別系統，並且也可以使用型別註記的方式顯性地告訴程式碼，變數或運算式的型別相關資訊。並且也確實會編譯成純 JavaScript 程式碼，因此有靜態語言的成分；然而是不是靜態語言，我們繼續看下去。接下來，我們可以打開官方 TypeScript 的 Playground⁴ 實驗一些簡單的程式碼。

4 <http://www.typescriptlang.org/play/>。



圖 1-4 TypeScript 官方的 Playground，可以用來體驗一下 TypeScript

如果讀者打開網站的話，沒意外應該會預先出現如圖 1-4 的程式碼，沒有的話可以自行實驗看看，將圖中的程式碼打上去。進到這裡之後，請將滑鼠指到 `message` 這個變數上，會出現如圖 1-5 的結果。

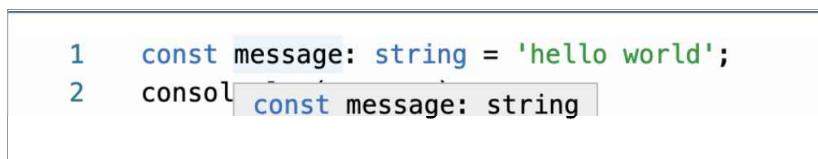


圖 1-5 將滑鼠移動到 `message` 變數上會出現型別的內容訊息

可以看到，TypeScript 會自動偵測到，我們的 `message` 變數是字串型別。這應該是理所當然的事情，畢竟都已經很明確地在 `message` 變數宣告上後面加了個 `string` 的型別註記，這是屬於靜態語言的特性，藉由文字提供的資訊得知變數之型別等。

接下來是重點，請讀者將範例程式碼的註記部分拿掉，變成：

```
const message = "hello world";
console.log(message);
```

將滑鼠移動到 `message` 變數上，會跳出如圖 1-6 的訊息。



圖 1-6 將滑鼠移動到 `message` 變數上會出現型別的內容訊息改變了

首先，作者先破梗一下，TypeScript 針對這種明文（Literal）格式的值也會歸類成一種型別，因此 `message` 變數此時的型別為 "hello world"，但那是因為前面宣告變數的關鍵字是 `const` 造成的，因此建議讀者改成 `let` 來測試看看。（結果如圖 1-7）

```
1 let message = 'hello world';
2 const let message: string
3
```

圖 1-7 將前面的宣告從 `const` 改成 `let` 之結果

讀者有發現任何盲點嗎？首先，我們肉眼看這段程式碼，理應覺得：「確實啊！`message` 確實是字串型別，而且因為不是由 `const` 來宣告變數了，也就是說任何字串都可以代入到 `message` 變數上啊！哪裡有錯呢？」

首先，靜態語言的定義重點在於——程式是根據自身的文字內容判斷變數的型別為何。然而，剛剛測試的程式碼連個型別註記或者是宣告性的文字都沒有：

```
let message = 'hello world';
console.log(message);
```

另外，變數似乎是因為 'hello world' 字串才推斷說它是帶有字串型別的值的變數，那麼這不就是動態語言的特性嗎？這是要搞得我們頭昏腦脹嗎？難道 TypeScript 這麼想腳踏兩條船？

事實上，有一種結合靜態與動態語言的特色之型別檢測機制，被稱之為漸進式型別系統（Gradual Typing⁵）。以下作者就秀一下它的定義：

»》漸進式型別系統 Gradual Typing

兼具動態與靜態語言的特色——程式碼在編譯過程中可能會遇到變數或表達式（Expression）被顯性地型別註記，這些變數會在靜態地編譯過程中檢測並且被監控；某些沒有被註記型別的變數或表達式等，會在程式裡自行推斷（Inference）型別之結果，如果遇到型別對應錯誤時釋放警告。

5 https://en.wikipedia.org/wiki/Gradual_typing。

根據以上的描述，我們可以知道 TypeScript 的設計，其型別系統兼具動態與靜態語言的特色。另外，讀者可能也在上面的漸進式型別系統的定義裡有瞄到這三個字：表達式。關於表達式的方面的東西也會在後續講得更詳細，甚至不熟悉「表達式」本身的讀者也不太需要擔心，先往下看就對了。

由漸進式型別系統的定義可以得知，TypeScript 對於監控型別的機制分成兩種類型：

»》型別註記 & 型別推論

型別註記（**Type Annotation**）為對變數或表達式進行文字敘述上的型別宣告動作。

型別推論（**Type Inference**）則是變數根據被賦予的值之型別來代表該變數之型別；而表達式則是經運算結果的值之型別來代表整個表達式最後的型別結果。簡而言之，撇除掉已經被註記過後的東西，決定其他沒有被註記過後的東西之型別，就是看結果值的型別是什麼就對了。

不得不說這兩個詞的英文很重要，讀者去查官方文件、上 Stack Overflow 查詢問題、看國外相關的 TypeScript 文章等，這兩個詞保證會出現得很頻繁。回過頭來，如果不清楚動態與靜態語言的特性，相信也就不會知道型別推論與註記到底這兩個東西是從何而來的——事實上，型別註記與推論的機制只不過是從很基礎的概念延伸出來罷了，本書會在第二章深究下去唷！

1.2.3 強型別語言 V.S. 弱型別語言

另外再談談很容易讓人搞混甚至誤用的名詞，也就是**強型別**（**Strongly Typed**）以及**弱型別**（**Weakly Typed**）語言。不過用過 JavaScript 的讀者，應該會覺得——既然 JavaScript 這麼隨性，應該也會猜它是弱型別語言吧。至於形容詞「強」跟「弱」到底指得是什麼，以下開始進行探討。

首先，用另類的詞來形容這兩種不同的概念的話，強型別語言比較固執，什麼事情都很一版一眼；弱型別語言相對來講比較能屈能伸，但可能出現過於隱匿的狀況。

以下舉一個超簡單範例，就是加法——你可能以為作者在開玩笑，這樣就可以讓強弱型別特色比較得出來？

```
1 + 1;
```

首先，上面那一行程式碼自然而然可以知道，一加一結果是二。幾乎不管哪個程式語言都會出現同樣的結果：2。

好，那這樣呢？

```
1 + "1";
```

一個是數字，一個是字串型別的數字 "1"，出來結果為何？

如果你直接回答出結果的話那你就錯了～答案是不一定！因為作者沒有告訴你這段程式碼要放到哪個程式語言環境執行。如果是在 JavaScript 的環境裡執行，理應的結果是——由於跟字串相加的特性，因此數字部分的 1 會自動轉型（**Coerce**）成字串型別的 "1"，因此會輸出結果為字串 "11"。

那麼換個另一個性質感覺跟 JavaScript 很相似的 Python 語言（不過 Python 跟 JavaScript 還是差很多，作者知道這當然是廢話），如果你將數字 1 和字串 "1" 相加起來，結果會如何呢～？

```
>>> 1 + '1'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

哇！丟出錯誤了，從以上的情境得知，每個語言的性質底下的細節多多少少還是有差。同理，作者喜歡的另一門語言——Ruby 也具備類似的情形：

```
irb(main):001:0> 1 + '1'  
Traceback (most recent call last):  
  5: from /usr/bin/irb:23:in `<main>'  
  4: from /usr/bin/irb:23:in `load'  
  3: from /Library/Ruby/Gems/2.6.0/gems/irb-1.0.0/exe/irb:11:in `<top  
(required)>'  
  2: from (irb):1
```

```
1: from (irb):1:in `+'  
TypeError (String can't be coerced into Integer)
```

從剛剛的例子可以看到，作者刻意將範例程式碼裡的錯誤訊息強調出來。而這裡，就是強型別與弱型別語言的根本差別——有沒有自動轉型（Coerce）。

» 強型別語言與弱型別語言 Strongly Typed v.s. Weakly Typed Language

弱型別語言對於任意型別的值之間的操作時，會自動轉換成適當的型別的值進行操作。

強型別語言則相對不允許不符合預期的型別的值進行操作，例如在 Python 或 Ruby 裡隨意將非數字型別的值與另一個數值進行加減乘除。

甚至連看起來很嚴格的 C 語言，它反而是偏向於弱型別語言⁶，有所謂的型別轉型（Type Casting）的機制，甚至藏得很隱晦。弱型別語言相較於強型別語言，看起來自由許多，而且省去了自己手動型別轉換的麻煩，但事實上並不是沒轉，而是都已經被語言本身特性自動處理掉了；然而，也因為這樣的自由，使得除錯過程中，多了一種發生錯誤或者是出現非預期性行為的可能性——隱晦的型別轉換。講一個很明顯的例子，大部分剛接觸 JavaScript 的新手（作者當初也是），想說以下的程式碼都已經會印出物件的結構了（結果如圖 1-8）：

```
console.log({ message: 'Hello world!' });
```

```
> console.log({ message: 'Hello world!' })  
▶ {message: "Hello world!"}  
< undefined  
> |
```

圖 1-8 使用 `console.log` 印出 JSON 物件的結果

常人預料之外的狀況則是——如果想要將 JSON 物件轉換為字串的形式來表示的話，以圖 1-9 所示的結果，物件被轉換成奇怪的 "[object Object]" 字串，

⁶ Stack Overflow - Is C considered weakly typed, why? <https://stackoverflow.com/questions/26753483/is-c-considered-weakly-typed-why>

至於原因為何，請自行上網查查吧～畢竟這並不在本書討論範疇。

```
> const jsonObj = { message: 'Hello world!' };
  console.log(`jsonObj is ${jsonObj}`);
    jsonObj is [object Object]
<- undefined
>
```

圖 1-9 直接硬將 JSON 物件自動轉型結果

回過頭來，TypeScript 到底是隸屬於強型別？還是弱型別的語言？事實上，**TypeScript** 也是屬於弱型別語言，它還是會跟原生 JavaScript 一樣，擁有那些古怪的自動轉型行為。至於為何？理由很簡單，用個矛盾反證法（Proof by Contradiction）來證明給大家看看：

»》 假設 TypeScript 為強型別語言的話 ...

首先，根據本書條目 1.1，TypeScript 本身的語法定位是 JavaScript 的超集合概念；也就是說，任何 JavaScript 程式碼可以經由 TypeScript 編譯器編譯成功。

既然所有 JavaScript 程式碼都可以經由 TypeScript 編譯器編譯，那麼根據 JavaScript 這種帶有型別轉換功能的語言，這段程式碼理應要可以成功地被編譯：

```
1 + "1";
```

然而，如果這段程式碼被編譯成功，就代表 TypeScript 一定得具備自動轉換型別的功能，也就是說，TypeScript 屬於弱型別語言——與第一句假設論述矛盾。

簡單吧～所以綜合前一條目 1.2.2 的結論是：

»》 TypeScript 語言本身的設計

TypeScript 是一門擁有漸進式型別系統（Gradual Typing）的弱型別語言（Weakly Typed）。

1.2.4 TypeScript 改善、解決的開發相關的問題

從條目 1.2.1 開頭使用原生 JavaScript 開發時產生的問題種類後，再來是 1.2.2 與 1.2.3 條目介紹 TypeScript 語言的概念與設計——也就是 TypeScript 隸屬於漸進式型別系統的弱型別語言的特質。因此作者總算可以在本條目討論 TypeScript 從原生 JavaScript 改善，甚至可以解決到的問題。

然而，是不是也有些問題就算換成 TypeScript 依然也沒辦法解決的？——答案是有的，不過這受限於 JavaScript 本身的特性，以下且聽作者娓娓道來。

TypeScript 比 JavaScript 最主要多出的功能就是型別系統（**Type System**），況且這個型別系統具備兩種功能：型別的註記以及推論。

首先，JavaScript 本身偏向於動態語言的特性，相對沒有靜態語言的特性；而恰巧 TypeScript 的型別系統補足了這一點，因此可推得：

使用型別註記功能時，靜態層面上監測變數或表達式之型別，所以在程式還未執行前，可以抓出潛在的錯誤。

還記得條目 1.2.1 提到的最關鍵的問題是什麼嗎？作者原封不動地複製這段話：

只要使用 JavaScript Debug 時，除非使用眼力掃整段程式碼來除 Bug，免不了必須執行程式碼並根據吐出之訊息來除 Bug。

往後這本書在講解的大部分內容，請讀者留意一件事情——在學習 TypeScript 的過程中，我們幾乎很少會直接執行程式，而是會查看型別檢測過程中，靜態分析上會不會跳出錯誤訊息！

二來是，既然都已經有型別方面的檢測，資料格式 / 型別 / 結構如有任何變異或潛在性錯誤等，理應來說 TypeScript 會幫你挑出問題來喔！這個部分在講到 JavaScript 物件方面的型別推論與註記機制時會仔細講解，這裡只是先提及一下。

另一方面，TypeScript 同時也有型別推論（Inference）的功能，也就是不需藉由明顯的文字註記性的提示，而是根據變數被指派到的值之型別，又或者是表達式運算結果之值的結果進行型別方面的推論。藉由這種方式，假設今天某個方法被呼叫時可能出現邊緣情境時，TypeScript 也會提醒開發者記得要去處理這方面的情境，以下使用陣列的 `pop` 方法為例，讀者可以跟著上 TypeScript 官網的 Playground 測試看看。

```
const array = [1, 2, 3];
const poppedElement = array.pop();
```

以上的程式碼，我們將陣列的元素 `pop` 出來，如果讀者以為 `poppedElement` 的推論結果是數字型別的話，那你可就大錯特錯了，請看結果圖 1-10。

```
1  const array = [1, 2, 3];
2  const poppedElement: number | undefined
3
```

圖 1-10 `poppedElement` 的推論結果為 `number | undefined`

TypeScript 幫你檢測之結果，除了是正常以為的數字型別外，還另外提醒有可能是為 `undefined` 的邊界情境，是不是很貼心？畢竟你可能無法知道程式執行過程中，陣列在執行 `pop` 方法得當下，到底會不會是空的。

如果搭配型別註記的功能的話：

```
const array = [1, 2, 3];
const poppedElement: number = array.pop();
```

可以看到圖 1-11 編輯器會自動在 `poppedElement` 的宣告下面顯示類似錯誤訊息方面的提示；圖 1-12 則是此錯誤訊息。

```
1  const array = [1, 2, 3];
2  const poppedElement: number = array.pop();
3
```

圖 1-11 宣告 `poppedElement` 且註記為數字型別時出現之底下的波浪狀錯誤提示

```
1 const array = [1, 2, 3];
2 const poppedElement: number = array.pop();
3 const poppedElement: number
4 Type 'number | undefined' is not assignable to type
5 'number'.
6     Type 'undefined' is not assignable to type
7 'number'. (2322)
8
9 Peek Problem No quick fixes available
```

圖 1-12 `poppedElement` 有潛在錯誤的原因是，有可能出現 `undefined` 的情形

所以條目 1.2.1 所提及的這個問題也可以被輕易解決喔！

有時候任何程式上的行為會出現各種結果——包含呼叫函式或方法、與陌生的第三者服務接觸等，但我們可能會忘記要去處理掉這些邊緣情境（Edge Cases）。

那麼有什麼東西是 TypeScript 很遺憾沒辦法告訴你的問題呢？相信這句話已經重複第三次了：「TypeScript 語法規範上是 JavaScript 的超集合」，所以避不了 JavaScript 弱型別語言的特性——享受型別自動轉型的狀況下，也就意味著這邊是最有可能潛藏的問題點。但這也沒辦法，語言特性使然，只得請讀者多多少少留意一下了。

另外，讀者可能還留意到作者少講的一個東西，也就是開發時打錯字（Typo）方面的問題；由於這是開發工具可以解決的問題，因此留待後續的章節來展示給大家看。

► 1.3 學習 TypeScript 的更多好處

讀者可能以為前面的條目就已經解釋很多了 TypeScript 很多的優勢，但事實上還有更多理由可以說服你應該要踏上這趟旅程。

1.3.1 對於各種不同版本 JavaScript 的編譯

首先，前端的開發者最頭痛的問題莫過於對於不同瀏覽器的支援，最惡名昭彰的應該就是 IE 了吧～不過清楚瀏覽器發展歷程的人應該也知道這是很難改變的事實——儘管我們已經有了 ECMAScript 標準規範出 JavaScript 標準的語法規定，但我們無法強制所有瀏覽器的實作必須完全符合到這樣的標準。

一是考量到不同的瀏覽器出自不同廠商或團體，自然而然地內部的實作會有差異，無法保證每個瀏覽器都會跟上標準的進度，標準是 TC39 委員會官方規定的，而瀏覽器的實作則是企業或團體，可能會有一定的時程漸進式的推出功能與支援。不過像是 IE 這種沒在維護的瀏覽器，只支援 ES5 標準的語法，因此開源社群提供的解決工具莫過於著名的 Babel Compiler⁷——負責將 ECMAScript 系列語法轉譯為 ES5 版本的語法。

不過隨著 Babel Compiler 的誕生，我們也會需要自動化的環境來建造專案打包的流程。延伸出的打包專案的工具也是很多種類，像是幫助整合專案的 Webpack⁸ 以及進行排除未用程式碼功能的 Rollup.JS⁹ 等。

而如果使用 TypeScript 的話，它的編譯器設定就會自動幫你把程式碼編譯成你想要的 JavaScript 版本（如果你過往有稍微嘗試過使用 TypeScript，那你應該會知道這些設定會放在一個名為 `tsconfig.json` 的檔案裡）；也就是說，你想要編譯成瀏覽器最基本只支援的 ES5 版本也是可以的喔！

1.3.2 TypeScript 擁有比較完善的物件導向語法

TypeScript 其中一個讓作者比較喜歡的地方在於——它相較於目前 ECMAScript 標準的發展，TypeScript 擁有較為完整的物件導向（Object-Oriented）方面的語

7 Babel 編譯器，請參見官方網站：<https://babeljs.io/>。

8 Webpack，請參見官方網站：<https://webpack.js.org/>。

9 Rollup.JS，請參見官方網站：<https://rollupjs.org/guide/en/>。

法¹⁰，使得我們可以使用 TypeScript 學習跟 OOP 基礎相關的東西。

其實在條目 1.1 最一開始的時候有講過，TypeScript 的主要語言設計者 Anders 也是 C# 語言的設計者，因此在型別系統以及物件導向方面的語法也有跟 C# 這門語言有相似之處。事實上，沒碰過或者不熟悉物件導向的開發者，如果從 TypeScript 開始學習物件導向的概念也是 OK 的，甚至想要延伸自己去學其他物件導向的靜態語言，包含 Java、C# 等也會變得比較容易。

1.3.3 TypeScript 定義檔具備使用說明文件的性質

讀者如果學到 TypeScript 比較深入一點的地方的話，尤其開始要引入第三方套件協作時（可能包含你使用過任何一個套件，如 jQuery、RxJS，亦或者是前端框架，包含 React、Vue 以及 Angular 等），除了上網查官方網站版本的文件外，也可以在自己的本機端，打開你的專案，查詢名為 **TypeScript 定義檔**（Definition File）的東西。

這個定義檔存了所有跟型別化名（Type Alias）、介面規格（Interface）與命名模組（Namespaces）等相關的東西，把它想成專案或套件的規格匯集地——也就是說，我們可以藉由一窺定義檔裡面的內容，反推該專案或套件的使用方式，甚至是連最基礎的東西可以查詢。

譬如你臨時想查 JavaScript 的 **Number** 到底有提供哪些方法，如果查詢定義檔，你會得出 **Number** 的完整定義：

```
interface Number {  
    /**  
     * Returns a string representation of an object.  
     * @param radix Specifies a radix for converting numeric values to  
     *               strings. This value is only used for numbers.  
     */  
    toString(radix?: number): string;
```

10 物件導向語法教學參見本書章節五；物件導向進階篇章參見本書章節九。

```
    /**
     * Returns a string representing a number in fixed-point notation.
     * @param fractionDigits Number of digits after the decimal point. Must
     *                       be in the range 0 - 20, inclusive.
     */
   toFixed(fractionDigits?: number): string;

    /**
     * Returns a string containing a number represented in exponential
     * notation.
     * @param fractionDigits Number of digits after the decimal point. Must
     *                       be in the range 0 - 20, inclusive.
     */
   toExponential(fractionDigits?: number): string;

    /**
     * Returns a string containing a number represented either in exponential
     * or fixed-point notation with a specified number of digits.
     * @param precision Number of significant digits. Must be in the range
     *                  1 - 21, inclusive.
     */
   toPrecision(precision?: number): string;

    /** Returns the primitive value of the specified object. */
   valueOf(): number;
}
```

筆者刻意將提供的方法用最粗的粗體字標出來，也就是說，Number 提供了包含 `toString`、`toFixed`、`toExponential` 等功能。另外，這些方法也都有它必須填入的參數的個數、對應型別以及輸出結果之型別。作者再次強調，這個就是查詢定義檔的好處——定義檔可以作為官方文件查詢提供的功能。

就連最單純的 JavaScript Number 都有定義檔闡述它的規格了，更不用說你還可以查 `String`、`Boolean`、`Object` 等。甚至是 ECMAScript 標準提供的東西，例如：ES6 Promise、迭代器函式（Generators）等規格，除了上網查看使用說明外，

TypeScript 額外提供你多一種方式，那就是查看這些功能在定義檔的規格¹¹。

讀者看到這裡也可以開始想想看，在 TypeScript 還沒盛行之前，以 jQuery 為例，如果假設今天 jQuery 想要開始支援 TypeScript 的話，到底是要把原本的原生 JavaScript 轉譯成 TypeScript 版本呢？還是有其他更簡單的方式去支援？

這些問題的解答關鍵就是 TypeScript 的定義檔啦！

1.3.4 TypeScript 好 Hot !

近年來熱門的套件或者是框架都開始轉移到以 TypeScript 為主要的語言開發了，不過光是看 TypeScript 的 GitHub 星星數，在作者寫書的這個當下也到了 57,000 多顆星星了，這個指標說明 TypeScript 在開源圈是很熱門的語言；對比它的競爭對手——也就是 Facebook 開源的 Flow¹²，雖然也是挺熱門，也看到它這時才剛破 20,000 顆星星，不過跟 TypeScript 相比還是沒過 TypeScript 一半的星星數，熱門程度上是有差別的。

如果按作者主觀感覺，之前接觸過一些 Flow 的經驗，開發體驗跟 TypeScript 相比覺得還是有待加強。再加上 Flow 只有型別宣告與檢測的功能，但 TypeScript 已經是獨立成一門除了型別系統外還附帶很多好用語法的程式語言，所以 TypeScript 熱門並不是沒有原因的。

另外，近年來很多框架都在使用 TypeScript 作為主要的開發語言。不過最早使用 TypeScript 作為基底語言的著名前端框架為 Angular，可能有一部分讀者早就已經在使用並且經驗過 TypeScript 了。而 Angular 除了善用典型的物件導向語法外，也運用了很多 TypeScript 裝飾子（Decorator）¹³的功能。

其他著名的 React 與 Vue 框架也都是近幾年內開始有了結合 TypeScript 的開

11 定義檔（Definition File）請參見在本書條目 8.3 。

12 <https://github.com/facebook/flow> 。

13 TypeScript 裝飾子語法教學參見本書章節十二。

發選項，所以你想要以 TypeScript 來開發前端的單頁式應用程式也已經變得輕鬆容易了；另外，也有熱門套件也選擇從原生 JavaScript 轉移到完全以 TypeScript 作為主要開發語言，其中就以 RxJS¹⁴ 這個套件最著名。

► 1.4 征途路上總是也有跌跌撞撞的時候

學習 TypeScript 免不了一定有遇到挫折的時候，本條目列出一些作者認為學習並且使用 TypeScript 時可能會遇到的一些問題，有些並不是能不能學得會 TypeScript 語法概念的問題，而是一些很細節的東西。

本條目設立的目的就是提前給讀者打一劑強心針。

1.4.1 命名是門哲學，踏入 TypeScript 更是如此

如果你是從 JavaScript 背景或者是一些本身沒有型別系統的語言背景，轉來開始學習 TypeScript 的讀者，你可能必須注意到命名原則（Naming Conventions）這件事情。

讀者可能會想說：「命名絕對難不倒我的！」

嘖嘖，那可就不一定囉～以下作者簡短闡述原因，因為作者就是碰過這個問題才講的。另外，命名原則部分，每個來源講解的規則形式不一定，因此參考多方資源是比較明智的做法。

別小看命名這回事，在 JavaScript 你只需要關心變數、函式或方法、類別等東西的命名就好了；然而在 TypeScript 裡從型別系統裡多出兩樣東西：基礎的型別宣告（Type Declaration）外，也有引入介面（Interface）的語法。

試想一件事情，如果你想要宣告名為狗狗 Dog 與貓貓 Cat 的類別，建立狗狗與

14 RxJS 參見 GitHub：<https://github.com/ReactiveX/rxjs>。

貓貓時，你可能會想要規定這兩個類別都是實踐某一個規格（Speculation，或者是簡寫為 Spec.）——也就是之後你會在本書學到的介面，你可能會將其命名為動物的介面，也就是 `Animal`。

萬一你沒規劃好功能（而且一定時常發生），突然想要改成那些狗狗貓貓都要繼承名為動物的類別，這麼一來，原本你宣告的介面是不是搶走了 `Animal` 的命名了？怎麼辦？所以有一部分人就是將介面統一改成由 I 開頭的命名，變成以 `IAnimal` 作為動物介面的命名。

1.4.2 型別與介面，你們搞得我好亂啊！

這真的也是作者學得很痛苦的點，沒有使用過型別系統與介面的經驗，很難分得清這兩個東西的差別；再加上官方的文件在撰寫本書期間有過期之嫌（但不保證本書出版後官方文件會不會更新），再加上文件敘述的內容艱澀難懂，所以大部分都是靠參考各方文章以及網路上的不同資源彙整出來結果。

本書會將型別與介面這兩樣東西進行定義上的比較以及闡述各自的建議使用時機。此外，在講到跟物件導向方面的概念以及實踐一些設計模式時，會比較清楚型別與介面各自使用的時機。

1.4.3 引入第三方套件

使用 TypeScript 引入第三方套件協作時跟使用純 JavaScript 開發的感覺會差很多；如果你已經會獨立引入並且使用套件提供的東西的話，相當於你已經過了使用 TypeScript 的痛苦期，已經開始朝向游刃有餘的方向囉～

簡而言之，這個關卡很像經過成年禮的概念，因為你不僅僅要懂得型別系統的根本外，你也必須要會獨立查並且看得懂條目 1.3.3 提到的 TypeScript 定義檔。

再者，你在使用每個套件時都一定避不了會有使用的陣痛期（算是你還在摸索怎麼使用套件的時候），而且這個所謂使用 TypeScript 開發時的陣痛的概念遠比你使用原生 JavaScript 還來得大很多；然而，這些種種原因不應當構成你避免使用 TypeScript 的理由，因為 TypeScript 終究的目的是要提升專案的可維護性。

1.4.4 命名空間與模組，避不了的學習門檻

由於 TypeScript 比起 ES6 的標準還要來個早三年多，所以有些語法事實上有跟 ECMAScript 有出入。不過嚴格來說並沒有衝突，但說到 JavaScript 的模組系統（Module System），作者實在是不得不搖頭。

首先，模組系統部分，我們常用到的 `import/export` 語法是 ES6 的特色；如果以後端 Node.js 為例的話，它是以 CommonJS 規範下的 `require/module.exports` 語法為主。然而，你覺得 TypeScript 一開始就有發展出這些語法嗎？

本條目的第一句話是：「**TypeScript** 比起 **ES6** 的標準還要來個早三年多」；也就是說，TypeScript 在 `import/export` 語法釋出前就有一套名為命名空間（Namespaces）與模組（Module）相關的語法，而且 1.5 版以前（但不包含 1.5 版）還分成更困惑的概念，分別是內部模組（Internal Module）與外部模組（Outer Module）。

這些所謂的命名空間與模組相關的語法後來漸漸的比較少見，因為 ES6 提供的 `import/export` 語法已經夠用了。然而，命名空間與模組依然存在於某些套件的定義檔裡面，因為你無法保證，如果直接改這些套件的定義檔，使其支援最新的 ES6 模組語法的話，那麼其他相依（Dependent）的套件與模組也就被跟著牽連到。

只能說，學習這些舊語法的目的並不一定是要會使用它，而是至少要能夠看得懂。

► 1.5 旅程中的第一小步

緊接著我們要開始設定並且寫第一個 Hello World 的程式碼囉～

1.5.1 環境建構

首先呢，要確保你的電腦有 NodeJS 的環境唷～通常要檢查如果 NodeJS 有沒有本來下載到你的主機端，可以打開你的終端機下達以下這個指令：

```
$> which node
```

請讀者不要把以上的指令的 \$> 部分一起打下去，你只要打 `which node` 的部分就好囉；之後只要是下達指令相關的步驟，一率都會以 \$> 為開頭，方便作為識別用途。假設你下達指令出來的結果會顯示 Node 被下載的位置（此為 MacOS 系統顯示的結果）：

```
$> which node
/Users/YOUR_COMPUTER/.nvm/versions/node/NODE_VERSION/bin/node
```

就代表你其實早就已經將 NodeJS 下載好囉～

另外，如果出現類似 `node not found`，就代表你的主機本來就沒有 NodeJS 喔～這時候就要請你上它的官方網站¹⁵並且開始下載。



圖 1-13 NodeJS 官方網站

¹⁵ <https://nodejs.org/en/>。

讀者上網會看到 NodeJS 下載區塊有提供兩種不同版本，其中筆者建議如果不熟悉的讀者可以直接下載圖 1-13 左方版本標有 LTS 的版本。所謂的 LTS 簡單來說就是 Long-Term Support，代表就是維護的主要版本對象。而另一個版本是如果你想要玩玩看一些很實驗性的功能可以下載的；然而本書並不太會用到深入的 NodeJS 方面的功能，因此就不再探究囉。

另外，當 NodeJS 下載完成時，會附帶一個名為 `npm` 的指令——全名為 Node Package Manager——是 NodeJS 圈的模組管理工具。其中，作為 TypeScript 語言的編譯器的終端機指令介面¹⁶也是被包裝為一個 Node 模組。首先，先檢查一下 `npm` 這個指令能不能使用。

```
$> which npm  
/Users/YOUR_COMPUTER/.nvm/versions/node/NODE_VERSION/bin/npm
```

如果出現的是類似 `npm not found` 的訊息，請重開終端機再測試一次。但重開過後還是沒辦法，就表示在下載 NodeJS 的過程中可能有出錯，最簡單的解法可能就是要把 NodeJS 全部砍掉重新下載。

好的，既然 `npm` 可以使用了，我們可以開始下載 TypeScript 的編譯指令工具囉～

```
$> npm install -g typescript
```

其中，`-g` 這個旗標代表的是——我們希望本機的任何地方（也就是全域）都可以使用到 TypeScript 編譯工具¹⁷。

¹⁶ 終端機指令介面就是常見的 CLI，全名 Command Line Interface。

¹⁷ 話雖說是這樣，但一般大型級別的專案（如：使用 Webpack 或者是 React / Vue 專案使用 TypeScript），通常在專案內部裡再下載一次 TypeScript 作為該專案的相依模組（Dependent Module），那是因為 Webpack 打包專案過程中沒辦法連結到本機裡全域版本的 TypeScript 編譯工具，只能把編譯 TypeScript 的工具一併放進去。這部分在後續篇章會再次提到。

下載完後，主要編譯 TypeScript 語言的是一個名為 `tsc` 的指令（全名為 TypeScript Compiler 或 TypeScript Command，兩種都有人在講）。按照剛剛安裝 Node 與 NPM 是的檢測方式，同樣使用 `which` 檢查 `tsc` 指令有沒有被安裝成功。

```
$> which tsc
/Users/YOUR_COMPUTER/.nvm/versions/node/NODE_VERSION/bin/tsc
```

如果出現的是類似 `tsc not found` 的訊息，請重開終端機，再測一次。如果還是沒有出現的話，可能真的要重新下載了¹⁸。

基本上，成功安裝下載的讀者們，我們可以進到下一個條目囉～

1.5.2 編輯器以及設定——建議使用 VSCode

環境背景的東西都處理好了後，接下來要找能夠讓我們寫程式碼的編輯器（Editor）。編輯器形形色色很多種，可能有些早就在用編輯器的人，通常用的是 Notepad++¹⁹、Sublime Text²⁰ 或 Atom²¹。甚至更進階的高手覺得 Vim 的自由性才能滿足到他們。

不過本書會建議，學習 TypeScript 使用的編輯器為 VSCode（全名 Visual Studio Code），可以上到它的官網²² 根據你的主機作業系統下載對應的版本喔～

18 請上網參考 Stack Overflow - tsc Command Not Found in Compiling TypeScript <https://stackoverflow.com/questions/39404922/tsc-command-not-found-in-compiling-typescript>。

19 Notepad++ 請參見 <https://notepad-plus-plus.org/>。

20 Sublime Text 請參見 <https://www.sublimetext.com/>。

21 Atom 請參見 <https://atom.io/>。

22 Visual Studio Code 請參見 <https://code.visualstudio.com/>。

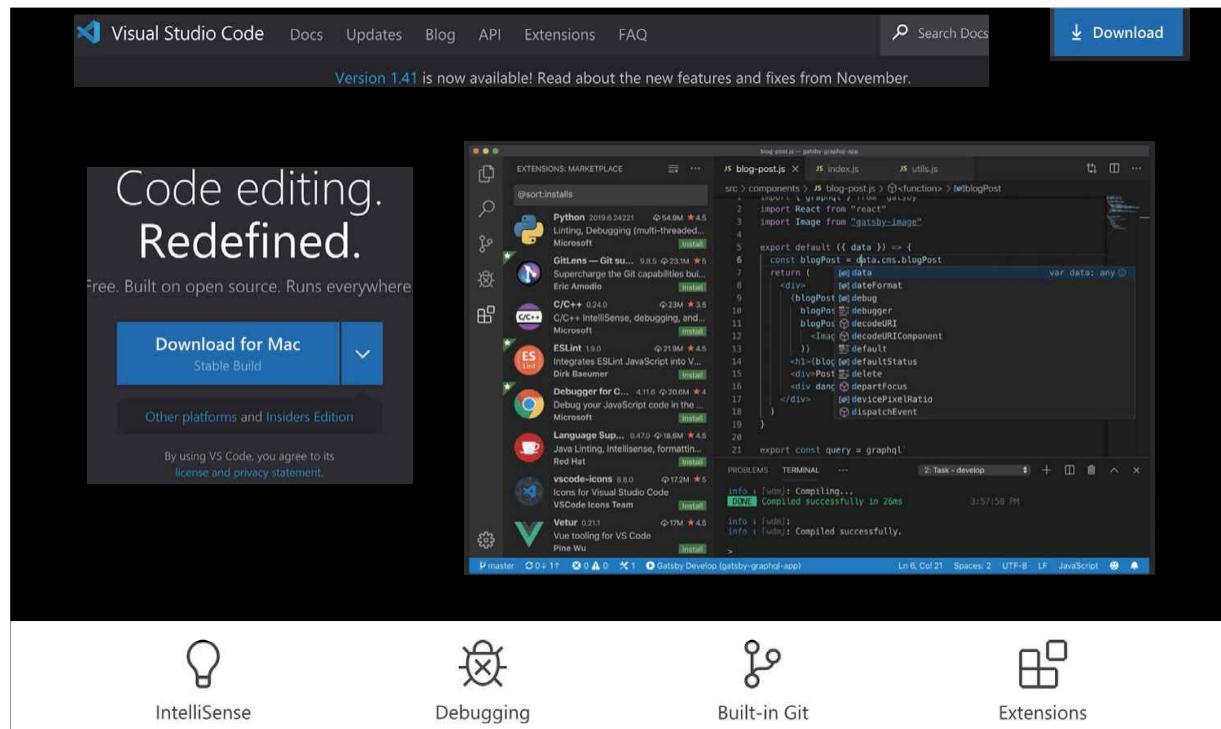


圖 1-14 VSCode 官方網站

以下開始闡述使用 VSCode 對於學習 TypeScript 的好處。

首先，VSCode 同 TypeScript 為微軟官方釋出的工具——自然而然地，**VSCode 對於 TypeScript 的開發者經驗會比較好**（也就是之前提到的 Developer Experience）。之前的條目 1.2.1 闡述到開發 JavaScript 時常遇到的問題中，其中有一點：

沒有工具的輔助，單純寫程式，錯字機率本來就很高。

首先，VSCode 與大部分的編輯器同樣都會有所謂的自動補全（Autocomplete）的功能，比如：如果你先前有定義過變數 message，而編輯器會自動識別你定義過的變數；之後想要使用該變數時，打了該變數的開頭，也就是只有 m 這個字元，就會跳出一個選單，告訴你可以幫你自動補全的選項。（如圖 1-15）



圖 1-15 宣告過的變數 `message`，之後會被記錄並且可以被自動補全

這個功能不僅僅是加快打字開發的速度，而且你可以避免你不小心拼錯 `message` 這個變數的名稱。

讀者可能這時候會問：「可是大部分編輯器都有這個功能啊？我難道不能用其他編輯器嗎？」

不急～我們就在下一個條目——寫我們的第一個 Hello World 程式的過程中告訴讀者——為何 VSCode 是開發使用 TypeScript 的專案的建議編輯器。

1.5.3 從第一個 Hello World 程式，學習正確地使用 VSCode

從 JavaScript 或者其他程式語言背景的讀者，看到這個條目，想說：「嘆嘆嘆，這麼簡單的東西，不就是簡簡單單的 `console.log("Hello world!")` 就好了嗎？乾脆跳過這邊好了～浪費篇幅！」

你跳過這裡就錯了！本條目的重點在於——教會讀者有效率地運用 VSCode 寫 TypeScript。不會這些技巧的話——講到應用方面的篇章，亦或者是讀者開發中途要自行查語法或套件相關的型別內容與定義時，會很莫名其妙的啊！

好的，那麼請讀者就開啟 VSCode 編輯器，一步一步來探索這個工具²³。

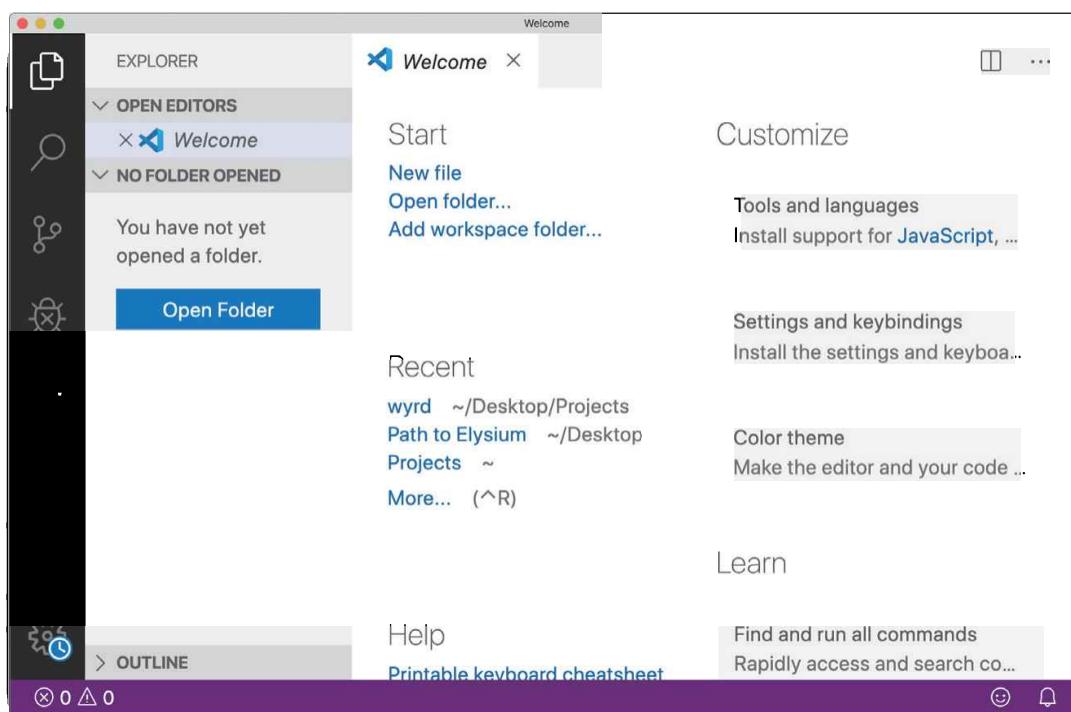


圖 1-16 VSCode 剛打開時的畫面

首先，左方會有一個 Open Folder 的按鈕，其實作者認為讀者可以開任何一個資料夾（比如說放在桌面），並且將該資料夾打開後，就會出現類似如圖 1-17 的畫面喔。



圖 1-17 打開資料夾後，編輯器左方出現資料夾名稱

23 圖 1-16 為打開 VSCode 時的畫面，可能會因為 VSCode 預設的主題一也就是 Theme一有所不同，所以會有不同的樣貌喔，但這不影響後續的教學。另外，圖 1-16 為放大過後版本，為的是比較容易呈現在書中，所以大小不一也沒關係，畫面有像到就好了。

VSCode 功能本身就很繁雜，本書只會涵蓋常用到的東西。

首先，我們就來創建第一個 TypeScript 的檔案。讀者可以在 VSCode 資料夾部分右鍵，就會看到如圖 1-18 的畫面。

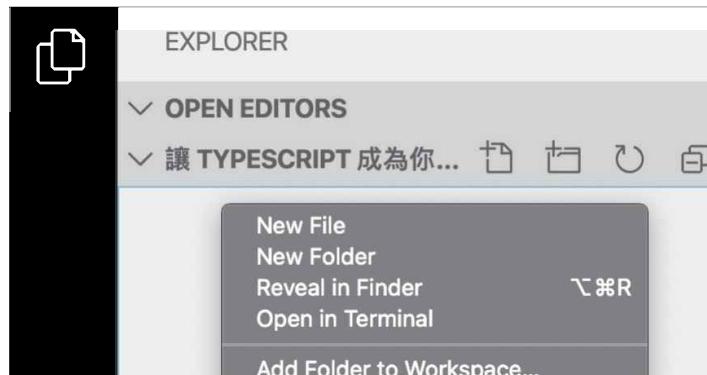


圖 1-18 滑鼠右鍵即可以看到的畫面

點選新增檔案（**New File**）後就可以開始命名檔案囉。這裡要注意的事情是，命名檔案時，**TypeScript** 程式檔都會以 **.ts** 作為結尾，因此這裡作者以 **hello-world.ts** 作為範例。（如圖 1-19）



圖 1-19 命名 TypeScript 檔案

記得命名完之後，要按下 **Enter** 鍵。不過就算你不小心用滑鼠點到其他地方，只要命名檔案的輸入欄位不為空，通常就會自動以該欄位裡的值作為命名創建出檔案。

好的，接下來打開我們的 **hello-world.ts** 檔案就可以在主要的編輯器的地方開始寫程式囉。以下是範例程式碼，而圖 1-20 為編輯器的畫面。

```
function greet(message: string) {  
    console.log(message);  
}
```

```
greet('Hello world!');
```

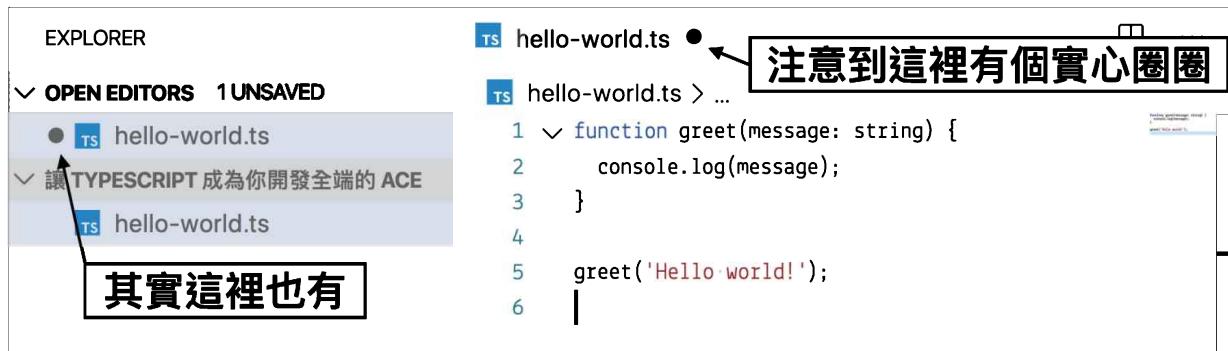


圖 1-20 打下程式碼後，VSCode 的畫面

其實大部分編輯器都有類似的功能，VSCode 也不例外——通常只要檔案有被更動時，該檔案的名稱旁邊會出現一個實心的圓圈——提醒你該檔案還在未被儲存的狀態。通常按一下快捷鍵 Ctrl 加 S (MacOS 則是 ⌘ + S) 就可以將該檔案儲存起來。這裡作者就好心提醒一下，不然辛辛苦苦寫完的程式忘記儲存會留下非常深的懊悔啊。

儲存過後結果如圖 1-21。

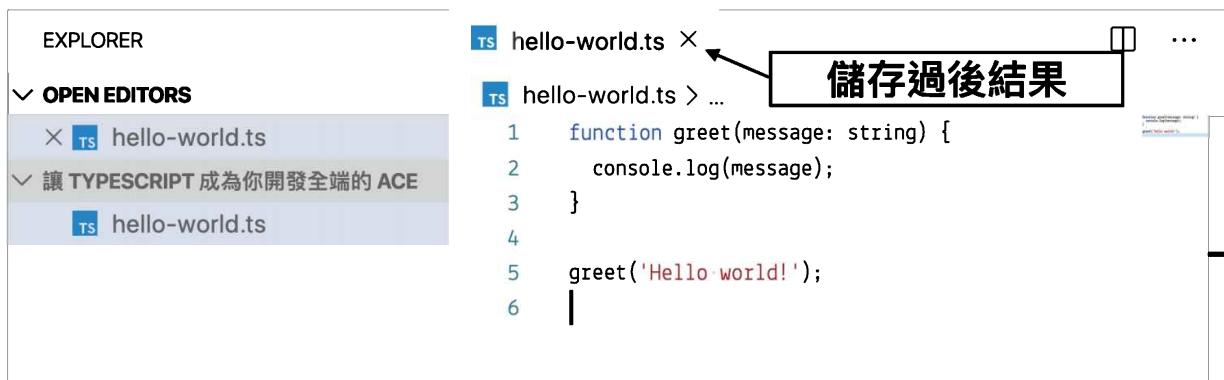


圖 1-21 儲存過後的結果，實心圓圈變正常叉叉

好了，接下來就是試著將我們的 TypeScript 程式碼進行編譯的動作。你可以選擇開啟你的終端機並且進到 `hello-world.ts` 所在的檔案資料夾，亦或者是直接開啟 VSCode 內建的終端機介面。不過作者長時間開發過後，儘管覺得能夠同時在 VSCode 寫程式跟開終端機，但後來還是習慣將這兩個功能分開，因此就看個人喜好。

首先，先進到剛剛創建檔案資料夾的位置（檔案資料夾名稱可能會與讀者不同喔）：

```
$> cd PATH_TO_YOUR_FOLDER/ 讓 TypeScript 成為你全端開發的 ACE
```

想要編譯檔案很簡單，以 `hello-world.ts` 為例，下達以下指令：

```
$> tsc hello-world.ts
```

通常編譯結果沒問題時，`tsc` 不會有額外訊息提醒你。不過請你回去看一下編輯器的狀態，你會發現會出現名為 `hello-world.js`——也就是被編譯過後的檔案。（打開結果如圖 1-22）



圖 1-22 產生的 `hello-world.js` 為編譯過後的結果

仔細比較一下 TypeScript 以及編譯過後的 JavaScript 兩個版本的差異：

```
/* 此為 TypeScript hello-world.ts 版本 */  
function greet(message: string) {  
    console.log(message);  
}  
  
greet('Hello world!');  
  
/* 此為 JavaScript hello-world.js 編譯過後版本 */  
function greet(message) { // 這裡 message: string 的註記被拔掉了  
    console.log(message);  
}  
greet('Hello world!');
```

最明顯的差異是——函式 `greet` 裡的參數 `message` 對應之型別註記部分 `string` 被拔掉了。這應該是可以理解的，畢竟 TypeScript 型別系統相關的語法，JavaScript 理應來說不能讀。

簡單的比較過後，作者要講一下很重點的部分囉～

不過在講到重點之前，先回到我們原本的 `hello-world.ts` 檔案，你會看到函式 `greet` 的宣告下有個錯誤，如果你用滑鼠滑到那邊會出現如圖 1-23 的錯誤訊息²⁴，由於不會造成學習上的困難，因此原因就放在備註裡，有時間或者有興趣再去看。為了方便展示，只得請讀者多做一件事情：先把編譯過後的 `hello-world.js` 刪除掉，剩下我們就繼續看下去。



The screenshot shows a code editor with two tabs: "hello-world.ts" (TypeScript) and "hello-world.js" (JavaScript). The TypeScript tab is active, showing the following code:

```
1  function greet(message: string) {  
2      function greet(message: string): void (+1 overload)  
3  
4      Duplicate function implementation. ts(2393)  
5  ! Peek Problem No quick fixes available  
6
```

A tooltip is visible above the second line of code, indicating a "Duplicate function implementation" error. The status bar at the bottom of the editor window shows the message "greetHelloWorld.js".

圖 1-23 編譯過後，原本的 TypeScript 檔案裡的函式 `greet` 出現錯誤訊息

刪除掉 `hello-world.js` 後，回歸 `hello-world.ts` 檔案。這一次，請讀者將滑鼠移動到函式 `greet` 上面，你可以看到會出現如圖 1-24 的訊息。

24 參照 Stack Overflow，TypeScript guide gives “Duplicate function implementation” warning <https://stackoverflow.com/questions/44192541/typescript-guide-gives-duplicate-function-implementation-warning>。

```
ts hello-world.ts > ...
1  function greet(message: string) {
2      console.log(message);
3  }
4  function greet(message: string): void
5  greet('Hello world!'); ↑
6
7
```

函式 greet 的型別

圖 1-24 將滑鼠滑到函式 greet 上面會顯示出它的型別

首先，滑到的任何一個變數名稱或者是函式，VSCode 會彈出該變數或函式所帶有的型別內容——這被稱之為型別提示（Type Hint），寫 TypeScript 的你一定會超級仰賴這個功能！（如果看不懂該型別到底是做什麼，第二章以後都會討論，所以不急～）

二來是，假設今天你告訴 TypeScript 如果該填入函式 greet 的型別為字串的話，然後你刻意填錯型別，比如數字好了，你就會被 TypeScript 發出警告。（如圖 1-25）

```
ts hello-world.ts > ...
1  fun... . . . . .
2  c Argument of type '123' is not assignable to
3  }     parameter of type 'string'. ts(2345)
4
5  greet(123);
6
7
```

TypeScript 告訴你，123 並非字串

圖 1-25 TypeScript 認定型別有衝突會立馬告訴你錯在哪裡

所以，只要任何 TypeScript 編譯器認定有型別產生衝突的狀況下，在還未執行程式碼前就會提醒你潛在的錯誤發生在哪裡了！這個應該可以被形容為型別衝突方面的偵測。

重點還沒結束，這一次作者刻意將呼叫函式 greet 的程式碼，塞到很後面，第 100 行已經算很後面了吧？（圖 1-26）

```
TS hello-world.ts ●
TS hello-world.ts > ...
97
98
99
100  greet('Hello world!');
101
102
```

圖 1-26 呼叫 `greet` 函式的程式碼被放到很遙遠的下方

讀者云：「作者你在搞什麼！？這還只是 Hello World 程式耶！？」

這裡要模擬的情境是：通常程式碼不會只有單單幾十行，而會是幾百幾千甚至是分散在很多不同的檔案裡，這應該才是常態。假設我們想要快速找到原本宣告該變數或者是函式的位置——我們不需要再檔案叢裡翻來翻去、滑來滑去，我們可以簡簡單單地用幾個快捷鍵解決掉就好。

這裡就以找出函式 `greet` 的宣告位置為例，首先將滑鼠放到目標上，此時按下 `Ctrl`（或者是 MacOS 的 `⌘` 鍵）你會發現呼叫函式 `greet` 的下方有個底線。（如圖 1-27）

```
TS hello-world.ts ●
TS hello-world.ts > ...
97
98
99  function greet(message: string): void
100  greet('Hello world!');
101
102
103
```

圖 1-27 滑鼠移到函式 `greet` 上面並且按下 `Ctrl` 或 `⌘`

出現底線的同時，將滑鼠按下去之後就會自動跳到函式 `greet` 所宣告的位置，這邊在此就不放上圖片了。

這個技巧的好處很多，而且並不侷限於同一個檔案內——如果有使用到 JavaScript 模組方面的語法（Import/Export），並且變數或函式甚至是後面還有型別的宣告與定義都放在其他檔案裡的話，我們可以隨時隨地用剛剛所展示的技巧快速查找到原始宣告的位置喔。

因此後續的篇章中，想要隨時查詢型別宣告與定義的話（而且相信我，你將來會很常這樣做的），剛剛所講的是必備招式啊！

»» VSCode 提供的功能與使用技巧

1. **自動補全功能 (Autocomplete)**：宣告過後的變數、函式等都可以很快速的被自動補全
2. **型別提示 (Type Hint)**：將滑鼠移動到變數、函式等，會跳出型別方面的內容提示
3. **型別衝突偵測**：使用 TypeScript 開發的過程中，如果偵測到型別上的衝突，VSCode 就會主動跳出警訊，提醒開發者要注意有潛在的錯誤發生——這也是使用 TypeScript 開發時，不太需要主動執行程式就可以快速除蟲的重點功能之一
4. **快速查詢**：如果想要快速查詢某變數、函式、型別宣告或定義所在的內容與位置，可以將鼠標移動到想要查詢的目標上，按下 **Ctrl** (**MacOS** 上則是按下 **⌘**) 的同時，並按下滑鼠，此時就可以快速導向到該目標上

02

TypeScript 型別系統概論

► 2.1 型別系統的兩大基柱——型別的推論與註記

2.1.1 運用型別系統必備的核心觀念

在第一章概論部分，條目 1.2.2 探討到動態與靜態語言特色的時候有稍微提到 TypeScript 具備所謂漸進式型別系統（Gradual Typing）的概念——有被文字標示註記的變數或函式等，就會以靜態的方式（也就是未執行程式前）偵測有沒有型別衝突方面的錯誤；而沒有被標示或者主動註記的變數或函式等，就會以所接收到的值（Value）或表達式（Expression）的運算結果的來判斷其型別為何。

前者用文字標明該變數或函式的型別的方式稱之為註記（Annotation）；後者讓程式自己判斷某變數對應其值或運算過後的表達式之結果值的型別，這種模式則是稱之為推論（Inference）。

這裡作者要先給出一個觀念：

儘管我們可以將所有的變數、函式、表達式等，主動地、甚至是積極地註記出它們各自所代表的型別，但這並不一定表示你得這麼做。

你可以寫出一段 TypeScript 程式碼，有些情形是只單純依靠型別的推論機制也可以寫出容易維護甚至還比較簡潔、淺顯易懂的程式碼。

不過這也並不代表你就可以選擇不用註記任何東西，還是要根據各種不同情形，包含：程式的可讀性、上手難度、維護難度、TypeScript 本身型別系統的特性等各種面向來判斷到底在什麼樣的時機適時地使用註記，亦或者是讓程式自己依靠推論機制就好了。

所以，本章重點其實並不僅僅強調你要懂基礎語法而已，你要會在哪個時機用什麼樣的語法同樣也很重要！

一句話「判斷使用推論或者是註記的時機點」就可以簡潔表達本條目的內容。

2.1.2 概觀型別推論與註記——各自的使用時機

筆者舉簡單的例子，以下分成：「100% 積極地註記」、「100% 消極地註記」以及「適時註記、適時推論」三種程式寫法，來檢視看看哪一種方式很容易讀，以及作者所謂「哪些是需要註記」，以及「放著讓程式自行推論也無所謂」的狀況。

以下的分析算是經驗過後整理出來東西，但並不代表全部得按照這樣的步驟，純粹只是作者整理出來的建議流程。

```
/* 100% 積極註記 */
let something: number = 123;
function addSomething(x: number): number {
    return (x + something) as number;
}
const result: number = addSomething(456) as number;
```

首先是以上的 100% 積極註記的案例，如果你看到上面的程式碼，相信一定會頭昏腦脹的吧！這使用得也太過分了些，肉眼看到變數 `something` 被宣告時就已經被指派數值了，所以我們一看就知道是數字型別；二來是函式內部還用到了 `as` 這個關鍵字來強調回傳值是數字型別，由於函式很簡短，所以好像也不太需要刻意強調回傳是數字型別。

最後一行的宣告 `result` 變數時，標明 `result` 是數字型別看起來比較需要，因為我們光靠 `result` 這個變數名稱無法用腦袋判斷型別。然而，後面呼叫

`addSomething` 函式時又再次註記其運算結果之型別時，好像又太過細，根本沒必要到這麼過火。

接下來我們看看下一個案例：完全放爛、不加任何註記的程式碼。（如果是這樣，乾脆回去寫純原生 JavaScript 程式碼還比較輕鬆）

```
/* 100% 消極註記 - 或者是單純只靠推論 */
let something = 123;
function addSomething(x) {
    return x + something;
}
const result = addSomething(456);
```

首先，第一行還好，變數 `something` 被宣告時直接指派數字，這個用頭腦都知道是數字型別，這個就讓 TypeScript 編譯器自己推論就夠了。

函式的宣告部分，尤其是參數（Argument）少了註記就可能有很大的問題了。事實上，如果你親自讓 TypeScript 編譯這段程式碼，它一定會發出警告！這部分的原因就等談論到函式型別的推論與註記（條目 3.4）時就會再往更深的方向探討。

這裡先用人話跟讀者說明，假設該函式 `addSomething` 裡的參數 `x` 沒有被註記時，就相對地代表「任何型別的值¹都可以代入到 `addSomething` 裡」；這聽起來很糟——這裡就可能產生潛在的 Bug。根據上面的函式宣告的方式來看，譬如隨便帶入一個字串的值好了，出來的結果會是字串，但這應該不會是我們想要的結果，也就是第一章有講過的——出人意料之外、非預期性的狀況。

最後一行就是宣告變數 `result` 的部分。如果遮住全部的程式碼，只看變數 `result` 的話，你可以光靠變數名稱猜得出來它的型別會是什麼嗎？如果你猜不出來的話，或者是很難用人類常識來聯想的方式去知道該型別為何，那就表示這可能是使用註記的最佳時機了。因為變數 `result` 可能是指計算出來的數字結果、某個結果的是或否（布林值）、字串的處理結果等太多繁雜可能性。

¹ 任何型別的值，如果有短暫接觸過 TypeScript 的讀者，它指得就是惡名昭彰的 Any 型別（參見本書條目 4.6）。

相反地，假設今天遇到的變數命名方式，像是 `isPositive`——你就可
以聯想出它的型別有很大機會是布林值（型別為 `boolean`）；又或者是
`nameOfSomething`，你就可以聯想出它的型別有很大機會則是字串（型別為
`string`）。因此這裡作者再次強調，有時候很單純、給人一種直覺性的變數命
名就可以讓程式仰賴推論的機制，而不太需要積極註記。

模糊或者是太廣泛的變數命名，譬如剛剛所示範的變數名稱 `result`，亦或者是
`typeOfSomething`²、`info` 等變數，通常還是註記一下會比較好。

最後來看看比較好一些，並且適時使用註記的程式碼範例：

```
/* 適時註記、適時推論 */
let something = 123;
function addSomething(x: number) {
    return x + something;
}
const result: number = addSomething(456);
```

以上的範例，除了解決剛剛所講到的問題——函式的部分限制住代入的參數
只能為數字外，按照條目 1.5.3 所講過的重點，TypeScript 會靜態地偵測程式
碼，如果呼叫函式 `addSomething` 時，代入不符合數字型別的值，就會自動跳
出警告訊息。

二來是宣告變數 `result` 部分時，我們積極註記其為數字型別，這除了可以
補足模糊不清的命名意義、增加可讀性外，假設不小心代入錯誤型別的值，
TypeScript 編譯器照樣會提醒我們並且拋出錯誤訊息。

以上就是比較三種不同情形時的狀況，當然，要判斷什麼時機要註記或者是放
著程式讓它自己跑推論的過程，還有很多東西要注意，這些就是第二章後續條
目要探討的重點。

2 以 `typeOfSomething` 這個名稱為例，它除了會是以字串型別表示外，亦或者會是以本書之
後會講到的列舉或複合型別有關（參見本書條目 4.2 以及 4.4）

»» 型別推論與註記之使用時機

1. 核心守則——TypeScript 的程式碼必須評估各種狀況選擇加上型別註記或者讓程式自行推論。
2. 通常積極作註記是為了限縮型別的各種可能性，使得程式碼可以減少要處理的例外狀況個數、增加模糊變數或方法使用上的可讀性以及讓 TypeScript 編譯器靜態偵測程式，確保型別不會有衝突產生。
3. 如果某段或某行程式碼本身可讀性夠或者是從命名上直覺判斷出型別的機率夠高的話，通常 TypeScript 型別系統的推論機制就已足矣。

► 2.2 型別註記——「註記」與「斷言」的差異性

型別註記部分相信讀者已經覺得在前面好幾個條目講過很多次，應該會很膩，但是作者還沒有完整講過註記的語法層面以及差異性，因為本條目很重要，作者給它打三顆星星可能還是嫌不夠。

事實上，註記再分細一點還有分成兩種：註記（Annotation）與斷言（Assertion）。

首先，作者在此聲明：本書除了本條目 2.2 以外，所有只要在程式中用文字標示的型別的相關語法，統一用「型別的註記」這個名詞涵蓋「註記」跟「斷言」這兩種概念。目的是要減少本書在講語法層面的複雜度。而本條目 2.2 要講的除了是型別註記方面的語法外，最重要的是要講解「註記」跟「斷言」這兩種概念的不同——再次強調：

型別的註記（Annotation）與型別的斷言（Assertion）兩者本質儘管相似卻是完全不一樣的概念。

然而，本書除了條目 2.2 以及部分少數內容有提及到這兩個名詞外，其餘都會以「型別的註記」這個名詞概括這兩個詞彙，這是為了減少後續講解的複雜度。

因此，條目 2.2 就要細講這兩者的差異——所以任何只要讀過本書的讀者，一聽到 TypeScript 的型別註記機制時，你要會判斷該資源或者是對方所講的東西是指「註記」還是「斷言」。

2.2.1 型別註記語法

首先是基本的註記語法，基本上就是讀者看到帶有冒號相關的語法，比如說：

```
let randomNumber: number = Math.random();
const myName: string = 'Maxwell';
const subscribed: boolean = true;
```

如果遇到函式的話，參數（Argument）部分除了可以有類似的註記方式標明輸入的參數型別外，在函式參數宣告結尾也可以註記該函式輸出之型別。以下就以名為 `isPositive` 的函式，示範輸入為數字型別，輸出則是布林型別的函式是如何在宣告的同時註記：

```
/**
 *  isPositive 為檢測輸入值是否為正數的函式
 *  @param input: number 為輸入的數字
 *  @output boolean 為輸出的結果，如果是正數，則輸出為 true
 */
function isPositive(input: number): boolean {
    return input > 0;
}
```

不過 JavaScript 裡，宣告函式的方法非常多種，其中一個是將函式作為值指派到變數裡。此時我們的註記方式會稍微不同，以剛剛的函式 `isPositive` 為例，如果換成是指派型的寫法會變成這樣：

```
const isPositive: (input: number) => boolean = function(input) {
    return input > 0;
}
```

首先，你會看到一個微小變異的地方——函式型別的註記部分不是：

```
(input: number): boolean
```

而是改成用箭頭（Arrow）的方式表示：

```
(input: number) => boolean
```

那是因為在宣告變數 `isPositive` 時，早就已經用過一次冒號了，因此後續為了不要混淆，只得用箭頭的方式表示該函式的型別。另外，這個型別表示的方式與 ES6 箭頭函式（Arrow Function）完全沒有關係——箭頭函式是可以被呼叫的函式，而箭頭方式表示的函式型別是一種註記方式——可被呼叫函式跟註記是完全不相干的東西，只是長得很像罷了！

當然，你可以把變數指派的函式改成 **ES6 箭頭函式³**（Arrow Function）的格式也可以，就只是指派運算子（Assignment Operator，就是指程式裡的等號）左方是函式的型別註記表示法，右方則是普通函式的宣告內容。

```
const isPositive: (input: number) => boolean = input => input > 0;
```

好，你可能嫌這個方式很麻煩、很冗長，拆成兩邊很醜——那我可不可以採用邊宣告函式、邊註記型別的方式定義函式？

答案也是可以的（但嚴格來說變成是讓程式碼推論被指派的函式型別）：

```
const isPositive = function(input: number): boolean {
    return input > 0;
}
```

換成 ES6 箭頭函式寫法也一樣：

```
const isPositive = (input: number): boolean => input > 0;
```

至於差異性在哪？這得留待討論函式型別的推論與註記機制⁴才會詳細討論，到目前為止先知道註記的格式、語法有哪些就夠了！

3 如果來自非 JavaScript 背景的讀者，對於普通函式跟箭頭函式差異不是很了解的話，單純讀這本書應該是不用到知道細微差異在哪；雖然這並非本書討論之範疇，不過還是建議有空好好補足這方面的知識。

4 函式型別部分請參考條目 3.4。

»》 註記的基本語法 Syntax of Type Annotation

- 變數的註記——假設宣告某一變數 `foo`，並且想將其註記為某型別 `T` 時，則基本的註記語法格式如下：

```
<let | const> foo: T = <expression>
```

- 函式的宣告與註記——假設宣告某一函式 `bar`，並且想將其參數 p_1 到 p_n 分別註記為 T_1 到 T_n 時，輸出的型別為 T_{Output} ，則基本的註記語法格式如下：

```
function bar(p1: T1, p2: T2, ..., pn: Tn): TOutput {  
    /* 函式的宣告內容 */  
}
```

- 變數為函式型別的註記——假設宣告某一變數 `bar`，並且想將其註記為某函式型別 `T` 時，且 `T` 之參數 p_1 到 p_n 分別註記為 T_1 到 T_n 時，輸出的型別為 T_{Output} ，則註記於指派運算子左方的註記方式為：

```
const bar: (p1: T1, p2: T2, ..., pn: Tn) => TOutput =  
<function-declaration>;
```

註記於指派運算子右方，並且合併函式的宣告時，註記方式則如下：

```
const bar = function(p1: T1, p2: T2, ..., pn: Tn): TOutput {  
    /* 函式的宣告內容 */  
}
```

換成 ES6 箭頭函式的格式則是：

```
const bar = (p1: T1, p2: T2, ..., pn: Tn): TOutput => {  
    /* 箭頭函式的宣告內容 */  
}
```

2.2.2 型別斷言語法

這裡要開始介紹「型別斷言」的部分，不過再次提醒——如果讀到本書其他篇章部分時，出現這裡所謂的「斷言」相關語法，作者統一會以「註記」涵蓋「斷言」的概念，為的是簡化講解複雜度。

以下開始進行分析：斷言（Assertion）的語法很簡單，看到有使用關鍵字 `as` 或者是 `<T>(...)` 的格式就是斷言的用法了。通常會用到斷言的情境是一程

式沒辦法推論（**Inference**）某表達式（**Expression**）的確切運算結果之型別，我們才會用選擇使出斷言來處理這種情境。

至於什麼樣的情境是程式無法推論的呢？一種簡單的案例是——使用所謂第三方的資源（Third-party resources），譬如說你想使用外來 JSON API 獲得的內容之型別格式、讀取檔案轉成 JSON 物件的結果、使用套件提供的功能、呼叫會回傳未知結果的函式⁵等。

看範例最快：假設有某個函式名為 `returnsUnknown`，程式本身可能不知道它會回傳什麼樣的型別，但你可能是參考過某外在來源、套件提供的文件等第三方資源的資訊，後來得知它絕對會回傳某特定型別——這裡以數字為例，則使用斷言的語法會長這樣：

```
const aNumber = returnsUnknown() as number;
```

或這樣：

```
const aNumber = <number>(returnsUnknown());
```

很簡單吧！但請注意，斷言的語法部分，沒有人斷言在變數的名稱宣告部分——也就是說，如果你這樣寫是錯的：

```
const aNumber as number = returnsUnknown(); // 這是錯誤的！
```

如果仔細思考過「斷言」這個名詞的意義，概念有點像是：「決斷地告訴程式，某表達式的運算結果之型別」。

也就是說，如果你斷言在變數名稱宣告部分，邏輯上本來就有些不合理了，沒有人決斷的說「某變數被運算的型別結果絕對是某某型別」—— 變數本身不是被運算，而是被指派某個東西，而斷言應該是斷在被指派的值或表達式的運算結果上。（「運算後的值」是重點）

複雜一點的情形是，函式的宣告表達式也可以被斷言（畢竟函式作為表達式也

5 這裡泛指 TypeScript 3.0 以後提供的新的型別類型 Unknown，請參考本書條目 4.6。

會被當成值)，不過筆者目前沒遇過需要這種斷言寫法的情況，然而這裡還是以 `isPositive` 函式，將就地示範一下：

```
const isPositive = (input => input > 0) as (input: number) => boolean;
```

亦或者是：

```
const isPositive = <(input: number) => boolean>(input => input > 0);
```

其實讀者進度若已經讀到本書後面提到函式型別的探討部分，上面斷言的寫法效果跟之前示範過基礎註記手法：

```
const isPositive: (input: number) => boolean = input => input > 0;
```

亦或者是：

```
const isPositive = (input: number): boolean => input > 0;
```

以上示範四種不同寫法效果其實根本沒差多少，不過「註記」跟「斷言」的語法真要比的話，意義上的差別可就差很多了！（下個條目（2.2.3）就會談到）

另外，如果讀者真的分不清斷言的語法可以在哪些地方使用的話，這裡就得先補充一個很重要的觀念！

以下開始要進行深論的主題是：

»》敘述式（Statement）與表達式（Expression）的定義與差別

1. 敘述式代表的概念是程式運行的流程，例如：JavaScript 裡的判斷敘述式（`if...else...`）以及迴圈敘述式（`for` 或者是 `while` 復圈）。
 2. 表達式代表的則是程式碼運算的流程，並且會將運算結果回傳（Return）
- 其中，兩者最關鍵的差異是：「敘述式不會回傳值、表達式則是會」。

以下就是常見的表達式案例：

```
/* 運算表達式 Arithmetic Expression */  
1 + 2 * 3; // 回傳結果：7
```

```
/* 邏輯表達式 Logical Expression */
```

```
true && (something === null || myAge < 18) // 回傳結果：可能是 true 或 false

/* 函式（或方法）的呼叫 Function/Method Invocation Expression */
Math.pow(2, 10); // 回傳結果：1024

/* 三元運算子 Ternary Operator */
myAge < 18 ? 'Youngster' : 'Adult'; // 回傳結果：'Youngster'，因為 myAge < 18
```

另外，敘述式的案例通常如下：

```
/* 判斷敘述式 Conditional Statement */
if /* Expression1 */ {
    /* 若 Expression1 為 true 則執行此 */
} else if /* Expression2 */ {
    /* 若 Expression2 為 true 則執行此 */
} else {
    /* 若 Expression1 與 Expression2 皆不為 true 則執行此 */
}

/* 迴圈敘述式 Looping Statement */
while /* Expression */ {
    /* 若 Expression 為 true，則重複執行直到 Expression 為 false 時跳脫 */
}
```

按照上面敘述式的定義——由於敘述式是在敘述運行流程，而不會回傳值，所以你才不會在 JavaScript 裡面看到這樣的寫法⁶：

```
/* JavaScript 並沒有提供這個寫法！ */
const status = if (myAge < 18) {
    return 'Youngster';
} else {
    return 'Adult';
}
```

6 實際上，有些語言允許用表達式的方式寫出判斷敘述過程，不過這邊更精確來說，它已經不再是判斷敘述式了，而是具有判斷功能的判斷表達式（Conditional Expression）；另外，要在 JavaScript 達到類似判斷表達式的效果的唯一方式就是用三元運算子（Ternary Operator）。

有些人對於敘述式跟表達式的判斷依據是有謬誤的——會誤以為「敘述式的呈現方式會是一整區塊（Block-level）的程式碼；而表達式部分由於牽扯運算過程，通常會是以單行的程式碼呈現，畢竟運算式按照剛剛的案例條列下來都是單行運行的」。

首先，作者就要證明剛剛那一段的論述是錯誤的。其中，敘述式不一定是多行式（或區塊式）地呈現的最佳案例是：變數宣告的指派敘述式（Variable Declaration Assignment Statement）。

```
/* (變數宣告的) 指派敘述式 Assignment Statement */  
const foo = 123;
```

這東西真的很細節，很要求讀者觀察 JavaScript 語言特性的各種細節（畢竟你都已經下定決心打算學 TypeScript 了，那順便把 JavaScript 的可愛小角落都看一看，一起完整學起來比較好）。

如果詢問讀者，剛剛的指派式會回傳什麼結果，可能不外乎有兩個答案：「數字 123 或 undefined」。如果你回答後者——也就是 undefined 就代表你答對了。（如圖 2-1）



圖 2-1 壓告變數 foo 並且指派數值的結果回傳的是 undefined

所以可以歸納出，JavaScript 的變數宣告的指派式屬於敘述式，非表達式——這是單行的敘述式在 JavaScript 的案例⁷。由此可知，第一個論述已被作者推翻：

7 本章節後面的練習部分會告訴讀者—JavaScript 也有指派表達式，但此狀況的討論並不在本書主要流程中，因此會以練習題的方式介紹。

敘述式不一定是以程式碼區塊的形式呈現，其中變數宣告指派式就是一種案例。

另外，什麼時候表達式會是以非單行的程式碼，也就是區塊的方式呈現？答：立即呼叫函式表達式⁸（IIFE，全名為 Immediately-Invoked Function Expression）就是著名的案例——英文都已經明講是 Expression 了，不用猶豫一定就是表達式。

```
/* JavaScript IIFE 的寫法 */
const status = (function(myAge) {
    if (myAge < 18) { return 'Youngster'; }
    return 'Adult';
})(16); // 假設填入參數 16，呼叫該函式的結果會回傳 'Youngster'
```

看到了沒～

表達式不一定是以單行運算流程形式呈現，其中立即呼叫函式表達式就是一種案例。

好！可以分得清敘述式和表達式的差別後，回歸型別斷言部分（Type Assertion），以下就是 100% 正確的用法。

»» 斷言的基本語法 Syntax of Type Assertion

斷言的語法只能用在表達式上，因為表達式具備回傳的值，敘述式則沒有——因此可以「斷言該表達式所運算結果之代表型別」。

若想將某表達式斷言為型別 $T_{Assertion}$ ，則寫法為：

`<expression> as $T_{Assertion}$`

另一種斷言的表現形式為：

`< $T_{Assertion}$ >(<expression>)`

8 非 JavaScript 背景的讀者們，看到 IIFE 可能多多少少會有些陌生，這應該是 JavaScript 的特定寫法，你可以想成一宣告函式的剎那間直接呼叫它，既然函式都被呼叫了，它沒有回傳呼叫結果值是不合理的行為，因此被歸納為表達式的一種是合理的喔！

以下的範例程式碼都是正確的斷言語法，這裡就請讀者細細品味。

```
/* 運算表達式 Arithmetic Expression */
(foo + bar * baz) as number;

/* 邏輯表達式 Logical Expression */
(isPositive(num) && isEven(num)) as boolean;

/* 函式（或方法）的呼叫 Function/Method Invocation Expression */
Math.pow(2, 10) as number;

/* 三元運算子 Ternary Operator */
(myAge < 18 ? 'Youngster' : 'Adult') as string;

/* 立即呼叫函式表達式 IIFE */
(function (x, y) {
    return x + y;
})(1, 2) as number;

/* 只要是表達式，就算在其他敘述式、表達式內部都可以使用 */
someFunction(foo as number, bar as string) as boolean;
```

同一系列的案例，更換成另一種斷言形式的寫法也可以，因為只要是表達式，任何表達式都可以被斷言。展示情形如下：

```
/* 運算表達式 Arithmetic Expression */
<number>(foo + bar * baz);

/* 邏輯表達式 Logical Expression */
<boolean>(isPositive(num) && isEven(num));

/* 函式（或方法）的呼叫 Function/Method Invocation Expression */
<number>(Math.pow(2, 10));

/* 三元運算子 Ternary Operator */
<string>(myAge < 18 ? 'Youngster' : 'Adult');

/* 立即呼叫函式表達式 IIFE */
<number>(
    (function (x, y) {
```

```
    return x + y;
})(1, 2)
);

/* 只要是表達式，就算在其他敘述式、表達式內部都可以使用 */
<boolean>(someFunction(<number>foo, <string>bar));
```

2.2.3 註記 Annotation V.S. 斷言 Assertion

作者已經將註記與斷言的語法都在條目 2.2.1 與 2.2.2 都講清楚囉！接下來讀者可能會想問：「明明都是型別方面的註記，都在做給 TypeScript 編譯器看的，差別在哪裡？」

首先，型別註記（Annotation）旨在告訴 TypeScript 編譯器：「任何被註記到的變數、函式的參數等，都必須遵照被註記過後的變數型別」。所以編譯器會隨時隨地監測該變數有沒有出現型別衝突的可能——關鍵字是「遵照」兩字。

譬如：

```
let foo: number = Math.random();
```

就是告訴編譯器：「這個變數 `foo` 被指派的值之型別必須遵照被註記的型別，也就是數字。」所以後面如果你很故意給它塞非數字型別的值，TypeScript 編譯器會警告你不要亂來。（如圖 2-2 與 2-3）

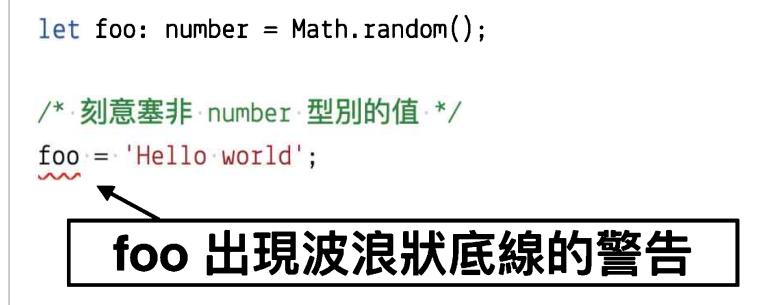


圖 2-2 違背遵照指示就會被記一支警告

```
let foo: number  
Type '"Hello world"' is not assignable to type 'number'. ts(2322)
```



圖 2-3 滑鼠移動到變數 `foo` 上面出現的錯誤訊息，TypeScript 跟你玩真的

另一方面，型別斷言（Assertion）則是無視 TypeScript 編譯器靜態分析整個程式碼的型別推論過程，果斷地告訴 TypeScript 編譯器：「被斷言過後的表達式之運算結果就是某某型別」—— 關鍵意象是「覆蓋」（Overwrite）該表達式的型別推論結果。

其實讀者若聽到「覆蓋」這兩字，應該會有第六感告訴你——這個動作具有潛在的危險性，後面也會示範。這裡作者先作弊一下，使用萬惡的 `any` 型別來示範⁹：

```
function returnsAny(): any {  
    return 123;  
}
```

以上的這個函式，首先讀者看到一個名為 `any` 的特殊型別註記，這裡暫時想成它代表任何型別（數字、字串、布林值、物件等等）。

假設這個函式提供自套件或外在來源——畢竟這個是使用斷言的通常狀況——我們如果呼叫這個函式並且指派其呼叫結果到某變數 `something` 上：

```
let something = returnsAny();
```

讀者可能以為因為函式 `returnsAny` 回傳的值是數字，所以變數 `something` 被推論結果也是數字。但事實上這是錯的，因為 `returnsAny` 在被「註記」（而非斷言）時，輸出結果必須遵照為 `any` 型別的結果，因此變數 `something` 被推論為

9 `Any` 型別，請參考本書條目 4.6。

`any` 型別再正常不過的事情了。(如圖 2-4)

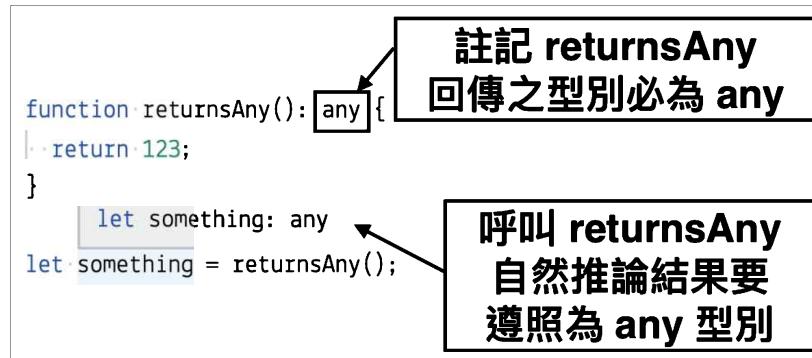


圖 2-4 呼叫函式 `returnsAny` 時，輸出結果必須遵照被註記結果

因此，這裡才是斷言比較適合用的地方——我們肉眼看到該範例都會知道該函式回傳的數字型別，以下使用斷言讓呼叫函式的表達式被複寫為數字型別：

```
let something = returnsAny() as number;
```

變數 `something` 就會被推論為數字型別。(圖 2-5)

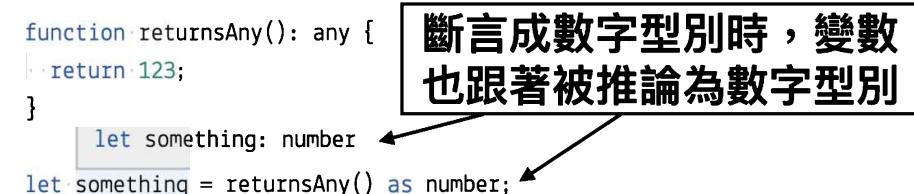


圖 2-5 斷言未知的表達式結果就會直接複寫該表達式運算結果之型別

不過斷言真的要很小心，如果寫出以下的程式碼，專案根本不用維護了。

```
function returnsAny(): any {
    return 123;
}

let something = returnsAny() as string;
```

以上這段程式碼就是斷言比較危險的地方——程式從此以後會認定變數 `something` 為字串型別，但如果你跑以上的程式碼，變數 `something` 實際存的是數字。而以上這段程式碼 TypeScript 也不會出現任何錯誤訊息(如圖 2-6)，完全是人為造成的錯誤。

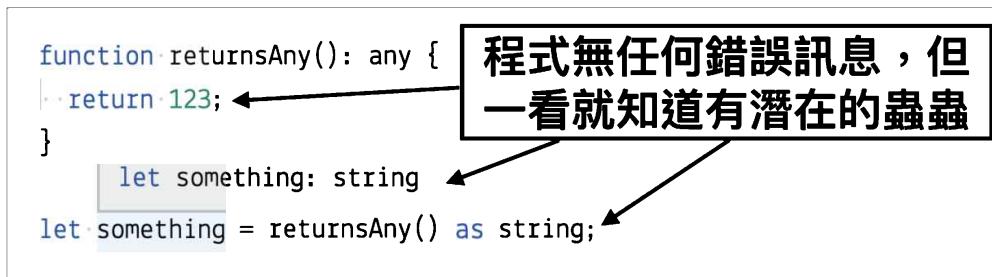


圖 2-6 肉眼看到回傳數字型別的結果，被斷言為字串型別，卻沒有任何警訊

有些讀者可能覺得：「我絕對不會犯這種烏龍錯誤的！誰會去使用一個回傳型別未知的函式或方法呢？」

有的！`JSON.parse` 這個方法在 JavaScript 裡應該算常用的，對不對？它專門負責解析 JSON 物件的字串並轉換成純 JavaScript JSON 物件，它回傳的結果就是 `any` 型別，因為該方法無法保證使用者的輸入長什麼樣子，所以只能將輸出定為 `any` 型別。

因此，作者這裡強調一點：「你一定會用到型別斷言的語法，只是看遇到什麼樣的情況，做出什麼樣的斷言罷了。」

所以，以 `JSON.parse` 為簡單案例：

```
const jsonString = `{
    "name": "Maxwell",
    "age": 18
}`;

let maxwell = JSON.parse(jsonString) as { name: string; age: number };
```

以上運用到所謂物件的明文形式的型別（Literal Type），這個會在條目 3.6 提到。

不過呢～ TypeScript 編譯器還是夠聰明，斷言至少不會誇張到可以讓開發者斷得走火入魔，比如：

```
123 as string;
```

嘖嘖，這擺明當 TypeScript 編譯器是白癡嗎？（圖 2-7）

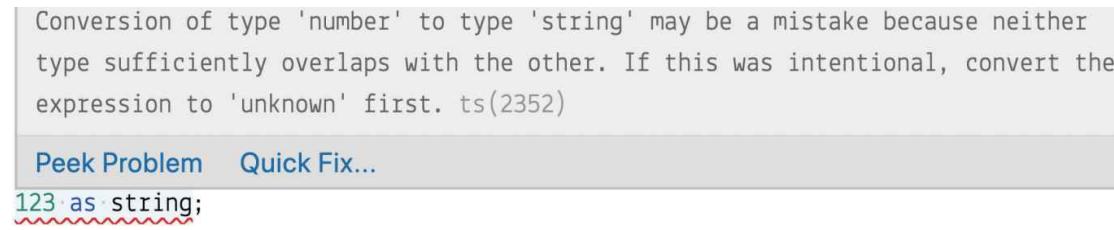


圖 2-7 數字被複寫為字串，實在是太看不起 TypeScript 編譯器了

但是這個錯誤訊息很有趣，作者節選下來給讀者看看：

Conversion of type 'number' to type 'string' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.

兩個重點提前被這個錯誤訊息破梗了。

“Neither type sufficiently overlaps with the other” 這句話暗示著型別有類似集合（Set）的概念，一個最簡單的案例是剛剛講到的萬惡的 `any` 型別——它就涵蓋了字串、數字型別等範疇，所以斷言在這個集合內，不會有任何錯誤訊息發出。

也就是說，如果今天換作是以下這個範例：

```
function devilNumber(): number {
    return 666;
}

devilNumber() as string;
```

請注意喔！這裡的函式 `devilNumber` 已經在宣告時聲明過回傳型別要遵照註記的型別——也就是數字型別，因此如果刻意呼叫它並且複寫為字串型別，當然也會出現同樣的錯誤訊息，畢竟數字跟字串這兩個型別完全各自獨立。（圖 2-8）

```
function devilNumber(): number {  
    return 666;  
}
```

所以說 TypeScript 編譯器至少還是很聰明的好嗎？

Conversion of type 'number' to type 'string' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first. ts(2352)

Peek Problem Quick Fix...

```
devilNumber() as string;
```

圖 2-8 TypeScript 編譯器還是會限制開發者不要亂玩斷言語法

另一句話是——“Convert the expression to 'unknown' first”——這邊有提到在型別系統比較難的 `unknown` 型別，但這個會在第三章詳細提到。

這裡就可以把「註記」跟「斷言」這兩種相似卻不同的概念總結並且告一段落了～（作者在寫這一段也覺得總算解脫了）

備註部分附上一帖優質的 Stack Overflow 問答串¹⁰ 供讀者延伸閱讀。

»》 註記 Annotation V.S. 斷言 Assertion

註記的意義在於——告訴 TypeScript 編譯器，該變數、函式型別必須遵照指定的型別，靜態分析時若出現型別不符的相關衝突，TypeScript 編譯器就會自動拋出警訊。

斷言則是強制覆寫掉被斷言的表達式之型別結果，通常用在回傳未知結果的表達式；使用斷言較有機率產生人為上的錯誤，因為 TypeScript 編譯器會果斷忽略斷言過後的表達式的運算結果之型別，採取斷言的型別作靜態分析。

儘管 TypeScript 編譯器會儘量防止錯誤的斷言發生，但使用時還是得小心些！

¹⁰ 參考自 Stack Overflow，Detailed differences between Type Annotation variable type and Type Assertion <https://stackoverflow.com/questions/47994926/detailed-differences-between-type-annotation-variable-type-and-type-assertion>。

► 2.3 綜觀 TypeScript 型別種類

這裡就要開始抓出所有你可能會在 TypeScript 程式裡看到的各種型別種類與樣貌。對於本來就有 C# 或 Java 相關經驗的讀者，應該會覺得很簡單；然而對只有 JavaScript 背景的讀者，有些可能就很需要時間吸收了。

以下粗略性的分類讓讀者快速地有印象就好。

2.3.1 原始型別 Primitive Types

原始型別就是所謂的資料型別的最小單位概念。譬如說，數字（Number）就是「數字」，沒有人在用字串或物件等其他類型的資料組成數字，因此可以得知：數字算是一種最小的資料表現形式；同理可知，字串（String）、布林值（Boolean）一定也是屬於原始型別的範疇。

而在 JavaScript 中代表未定義的空值 `undefined`（Undefined）以及本身是「代表空值」的值（但同時又是物件）的 `null`（Null）也是隸屬於原始型別的範疇¹¹。

還有一個大部分的人肯定會忘記，但確實被認定是原始型別類型的東西——也就是 `void`；它所代表的意義與 Undefined、Null 有點像，都有空值的意味，不過 `void` 通常會用到函式型別代表沒有輸出的狀態。

最後，還有一個東西也是原始型別（作者你到底要還有多少個！？），也就是 ES6 推出的新的原始型別 `Symbol`¹²，由於它並非在詳細討論範圍，因此將說明放在備註。

11 Undefined 跟 Null 的差異由於並不在本書的討論範疇，如果對於這兩個東西的差別有疑問，可以參考 Stack Overflow - What is the difference between null and undefined in JavaScript <https://stackoverflow.com/questions/5076944/what-is-the-difference-between-null-and-undefined-in-javascript>。

12 Symbol 大略可以看作是獨一無二版本的值存在的形式，也就是說 `Symbol('hello') === Symbol('hello')` 會回傳 `false` 這個結果，儘管內部被帶入相同的值，但 Symbol 會各自建立獨立的參考基準，詳細可參考 MDN 官方使用說明 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol。

總歸來講，以下這些東西都是隸屬於原始型別的範疇，各自互相獨立（請記得：就是誰也不屬於誰的概念）。

- `number`
- `string`
- `boolean`
- `undefined`
- `null`
- `void`
- `symbol`

2.3.2 物件型別 Object Types

其實 JavaScript 物件（Object）的觀念一直以來都是對初學者很令人頭痛的，而 TypeScript 物件型別也不例外，細分之種類也很繁雜。

接觸 JavaScript，你會看到物件的表示方式有幾種，最常見的非 **JSON** 物件莫屬，全名就是 **JavaScript Object Notation**，例如：

```
{  
  name: 'Maxwell',  
  age: 18,  
  interest: ['drawing', 'programming'],  
}
```

第二種在 JavaScript 裡，也是屬於物件的東西——陣列（Array），例如：

```
['Maxwell', 'Taipei', 'TypeScript 101']
```

事實上，JavaScript 的函式（Function）也是屬於物件——但更精確的講法是：「任何非原始型別的資料，在 JavaScript 是屬於物件的範疇」¹³，所以由此可推，函式也是物件的一種。例如：

¹³ 參見 Stack Overflow，“Every object is a function” and “Every function is an object”，which is correct? <https://stackoverflow.com/questions/3449596/every-object-is-a-function-and-every-function-is-object-which-is-correct>。

```
function greet(someone) {  
    console.log(`Hello! ${someone}.`);  
}
```

還有一個較為明顯的東西也是物件——就是從類別（Class）建構出來的物件，它有一個專有名詞，曰：「實體（Instance）也」。

不熟悉類別的讀者看到這裡如果不懂的話，請先不要緊張，這邊只是先給讀者概略講過本書之後會詳細討論的東西；作者這裡只要請讀者記住這一句話：「類別建構出來的物件——也就是實體，也屬於 JavaScript 物件的範疇」。而那些實體的建構方式就是用關鍵字 `new` 出來的，比如說常常會使用到的 `Date`（日期）物件：

```
const foo = new Date('2020-01-01');
```

變數 `foo` 存的就是從 `Date` 類別建構出來的物件，也就是 `Date` 類別的實體。

»» JavaScript 物件範疇

- 只要是非原始型別的資料，通通可以被歸類為 JavaScript 物件。
- JavaScript 物件的表現形式主要分成：**JSON** 物件、陣列、函式與類別建構出來的物件，也就是實體。

這麼多種不同的物件的表現形式，在 TypeScript 裡當然需要有對應的型別表示方法。所以既然我們有 JSON 物件，就會需要對應的 JSON 物件的明文（Literal）表現形式，例如：

```
const info: {  
    name: string;  
    age: number;  
    interest: string[];  
} = {  
    name: 'Maxwell',  
    age: 18,  
    interest: ['drawing', 'programming'],  
};
```

陣列就會有對應的陣列型別（Array Type），格式為 `T[]`，例如：

```
const data: string[] = ['Maxwell', 'Taipei', 'TypeScript 101'];
```

函式就會有對應的函式型別（Function Type），條目 2.2.1 有提到過寫法的極微小差異，不過這裡就還是放上一般註記普遍看到的函式型別之樣貌：

```
const greet: (someone: string) => void = function (someone) {
    console.log(`Hello! ${someone}.`);
};
```

類別創建的實體，就有對應的一—型別化名（Type Alias）的方式來代表該實體的型別，不過型別化名的定義很廣，第三章以後會有更多說明。

```
const foo: Date = new Date('2020-01-01');
```

以上作者將 JavaScript 物件以及對應的註記方式概略上給讀者看過一遍了，有印象後再去深入型別系統就比較不會感到陌生。

根據剛才的討論結果，由於物件型別實在是太過廣泛，所以讀者如果聽到物件型別等相關性詞彙，請記得：到底討論的主題目標是什麼？是 JSON 物件型別、陣列還是函式型別？又或者是類別創建出的實體之型別？這些都得注意。

2.3.3 明文型別 Literal Types

條目 2.3.2 裡面有講述過 JSON 物件的明文（Literal）表現形式，這裡再將明文（Literal）這兩個字的定義講得更清楚：

»》明文的定義 Definition of “Literal”

就是值（Value）的表現方式。

很短吧！明文資料就是指以下的幾種案例：

```
// 數字明文 Number Literal
123;
```

```
// 字串明文 String Literal
'Hello world!';

// 布林值明文 Boolean Literal
true;

// 物件明文 Object Literal
{ name: 'Maxwell', age: 18 }; // ← JSON 物件
[1, 2, 3]; // ← Array 物件
() => 123; // ← 函式物件 (ES6 箭頭函式表示方式)
```

以上的這些值，除了函式外，都可以自成一種型別，於是可以在 TypeScript 被統稱為明文型別（Literal Types）。以下的這種範例寫法：

```
const foo: 1 | 2 | 3 = 1;
```

範例中變數 `foo` 被限定只能被指派為數字 1、2、3 其中一種。另外，讀者從以上的範例可以推論出，想要達到型別上的邏輯「或」（OR）的效果，我們可以使用管線記號（Pipeline），也就是 `|`。這個帶有邏輯「或」的效果等等在條目 2.3.5 會稍微介紹，並於條目 4.4 裡完整分析。

2.3.4 TypeScript 提供型別 TypeScript Provided Types

條目 2.3.1 ~ 2.3.3 基本上學過 JavaScript 的基礎，都會覺得是熟悉的資料面孔。（有這樣的形容方式嗎？）

而 TypeScript 也有引進一些原生 JavaScript 所沒有的型別特色（Feature）——元組（Tuple）與列舉（Enum，全名為 Enumerator）型別。

元組的長相跟陣列很像，差別就在——元組有元素（Element）數量的限制、內部所存的元素順序（Order）與各個元素對應之型別都有嚴格的規定；陣列就是你一般使用 JavaScript 時，你可以不用管內部長短限制，元素只要符合被註記的型別就好。

以下舉個簡單的元組範例：

```
const foo: [number, string, boolean] = [666, 'Devil Number', false];
```

變數 `foo` 被限制為只能存三個元素，第一個必須為數字、再來是字串，最後才是布林值。如果你是想達到的反而是——某陣列可以存放「數字」或「字串」或「布林值」而不需要管順序與長短大小，則根據條目 2.3.3 最後短暫提到的型別邏輯「或」的概念的寫法，可以用以下的範例程式碼達成：

```
const bar: (number | string | boolean)[] = [666, 'Devil Number', false];
```

你換值的順序：

```
const bar: (number | string | boolean)[] = [false, 'Devil Number', 666];
```

或換長短大小：

```
const bar: (number | string | boolean)[] = [true, 123, false, 'Hello', 99];
```

都是沒差的，但元組就是相對陣列而言，比較固執一點的型別就是了。其運作的機制會在條目 4.1 詳細地討論。

另一方面，列舉（Enum）型別是作者主觀認為還蠻不錯的功能。列舉的意涵旨在將相似性質的資料，用文字描述並且匯聚成的一種型別，譬如說：

```
enum Color { Red, Blue, Green, Yellow, White }
```

以上的程式碼宣告一個名為 `Color` 的列舉型別，專門存放跟顏色相關的型別資訊。

使用時，我們可能會這樣做：

```
const baz: Color = Color.Yellow;
```

代表變數 `baz` 為 `Color` 列舉型別，並且存放的鍵（Key）為 `Color.Yellow`。

不過列舉在使用上有很多很細節的地方，註記時筆者也有碰過很令人頭痛的陷阱，還有列舉型別延伸出的列舉成員型別（Enum Member Type），這些細節就放在條目 4.2 慢慢討論吧～

2.3.5 特殊型別 Special Types

本條目的東西，難度會在學習過程中慢慢攀升。特殊型別在 TypeScript 裡是非常重要的課題，如果不想踩到雷、想寫出安全（Type Safe）並且容易維護的程式碼，就得好好把本條目的東西熟悉起來，也就是：

- `any` 型別——專案裡不確定性存在的型別，你會想要儘量根除它的存在，通常它會是專案裡混沌的來源。
- `never` 型別——專案裡如鬼魅般的存在的型別，你不會注意到它的存在，但它是默默存在各種角落的型別（~~講到運作機制你就會知道為何作者要這樣形容子~~）。
- `unknown` 型別——專案裡較為安全的不確定性存在的代表型別，主要用來替代 `any` 型別的狀態。

`any` 型別曾短暫出現在條目 2.2.3 裡，該條目在討論註記跟斷言的差異，其中通常帶有 `any` 型別的值或表達式會用斷言來將 `any` 型別蓋掉，但就還是得小心些。

而 `never` 型別與 `unknown` 型別，就讓它們在條目 4.5、4.6 一起揭開他們的運作機制與代表意義吧～

2.3.6 進階型別 Advanced Types

讀者看到這個標題，肯定會問：「不是上個條目，也就是特殊型別就結束了嗎？怎麼還有啊！？」

如果仔細去查 TypeScript 版本發展過程，綜觀來看，連作者都覺得有些功能很變態。（不過有些是還蠻有趣的）

其中一個必學的實用型別就是泛用型別（Generic Types，簡稱 Generics），它是一種將「型別自身進行參數化（Parameterize）後表現出來的特殊型別」，作者知道這很饒舌，但用很短的一句話就是那樣形容；若想要最精簡版重點，莫屬這三字：「參數化」。

譬如宣告一個名為 echo 的函式，丟進什麼東西，出來的就是什麼東西：

```
function echo<T>(something: T): T { return something; }
```

其中，該函式的宣告中，會發現多了一個很奇怪的東西，那就是 echo<T> 中的 <T>——而那個 T 就是參數化過後的型別，簡而言之就是型別參數（Type Parameter）。讀者可以這麼想，我們在呼叫函式或方法時，會有所謂的可以填入的參數（Parameter），例如我們常見的：

```
console.log('Hello world');
```

字串 'Hello world' 就是方法 log 的參數。同理，假設剛剛的函式 echo<T> 中的 T 填入某個型別，例如數字好了，請讀者在腦袋中把 T 都取代為 number，於是你就得到這個結果：

```
function echo(something: number): number { return something; }
```

該函式必須填入數字型別的值，輸出的值也會跟著要對應成數字型別。同理，如果將 T 替換成字串，就會得到：

```
function echo(something: string): string { return something; }
```

所以，函式 echo<T> 的宣告等於表達了一件事情：「輸入 echo 函式的參數若為型別 T，則其輸出的型別必需等於輸入的值之型別 T」。讀者應該能夠稍微體會到，所謂的型別參數化的概念到底是怎麼樣的感覺了吧～

另外一種特殊型別也很常用，通常是用來鎖定物件的鍵值對（Key-Value Pair）對應的型別的行為¹⁴，這個東西在英文裡謂之“Indexable Type”，中文實在很難翻，就姑且稱之為可控索引型別——畢竟是在控制物件之鍵與值對應的型別。

例如，你想要創建一個 JSON 物件，其中，鍵與值各自都必須為字串，你就可 以這樣寫：

14 學過某些語言的人，如 Python，可以把此處當成是在描述 Dictionary 這種資料型態，亦或者是在 Ruby 裡看到的 Hash 資料型態，以及函數式程式裡常見的 Record、Map 等東西。

```
const dictionary: { [key: string]: string } = {
    name: 'Maxwell',
    description: 'Will always be 18 yrs old.',
    reason: 'You will never know.'
};
```

而很容易跟可控索引型別搞混的另一個雙胞胎，名之“Index Type”，所以中文自然而然只能翻譯成索引型別——它的主要功能為動態地檢測「某鍵值對物件有沒有正確地使用到該物件的屬性（Property）」，通常你看到關鍵字 `keyof` 就會大概知道該程式碼有用到索引型別的技巧——而官方將 `keyof` 稱之為 Index Type Query Operator，也就是索引列隊操作子。

例如：

```
const dictionary = {
    name: 'Maxwell',
    description: 'Will always be 18 yrs old.',
    reason: 'You will never know.'
};

let info: keyof dictionary;
// info 的型別為 ('name' | 'description' | 'reason')
```

這一小段：

```
keyof dictionary;
```

可以動態地將變數 `dictionary` 中的所有鍵進行邏輯「或」起來；此外，它還有一些很細節的應用，作者暫時就先移到下一個進階型別，細節的討論會在後續章節提到。

當然，另一個超常見的進階型別就是前幾個條目不時在範例裡出現的——用邏輯「或」起來的型別；相對「或」這個概念而言，也就會有另一個邏輯「和」（AND）起來的型別。

這裡作者稱它們為複合型別（Composite Type），而如果上網查官方的文件，你會發現，邏輯「或」起來的型別稱之為聯集型別（Union Type）；而邏輯「和」對應的型別稱之為交集型別（Intersection Type）。

從前幾個條目可知，遇到聯集型別的頻率會高很多，但不代表對照的交集型別不會在你的專案中出現；交集型別也可能會被藏到型別系統的某些地方，只是沒了解過底層機制會不清楚狀況，但這並不要緊，對於寫不寫得好的 TypeScript 這方面不會有太大影響。

比較容易用錯的也是交集型別，它是相對於聯集型別使用的管線符號——交集型別則是以小耳朵符號 & 作為其操作子。以下的範例：

```
// "數字" 又或者是 "字串"  
let numOrString: number | string;  
  
// 既是 "數字" 也是 "字串"  
let numAndString: number & string;
```

第一個變數 `numOrString` 可想而知，可以存放數字或字串型別的值；然而，`numAndString` 這個變數的定義很奇怪——它可以存放既是數字也是字串型別的值！？

這樣的案例就是一種典型的型別互斥（Mutually Exclusive）的情形，至於 TypeScript 是如何判定此狀況，亦或者是直接丟個錯誤出來呢？這些都會在後續的章節討論～

以上列出的進階型別，包含：泛用型別、可控索引型別、索引型別以及複合型別是 TypeScript 專案裡，作者認為常見程度偏高的進階型別，到這裡就大概給讀者帶過全面的東西了。

► 本章練習

- 試著描述 TypeScript 型別系統（Type System）的主要概念。（型別推論與註記）。
- 註記的語法中，還有在更細分成「註記」跟「斷言」，兩種語法的用途是什麼？註記方式的關鍵性差異在哪？

3. 使用「斷言」的語法要注意的地方在哪？
4. 試描述敘述式（Statement）與表達式（Expression）的概念。兩者之差異性在哪？
5. 正確的註記方式有哪些，錯誤的註記方式則又是錯在哪？

```
/* 題目 5-1 註記 v.s. 斷言 */
let ex1: number = 123;
let ex2 = 123 as number;
let ex3 as number = 123;
let ex4 = 123 as string;
let ex5 = 123: number;
let ex6 = (123)<number>;
let ex7 = <number>(123);

/* 題目 5-2 函式註記方式 */
let ex1: (x: number, y: string) => string = function(x, y) {
    return x.toString().concat(y);
};

let ex2: (x: number, y: string): string = function(x, y) {
    return x.toString().concat(y);
};

let ex3 = function(x: string, y: string) => string {
    return x.toString().concat(y);
};

let ex4 = function(x: string, y: string): string {
    return x.toString().concat(y);
};

function ex5(x: string, y: string): string {
    return x.toString().concat(y);
};
```

6. 試舉出原始型別（Primitive Types）與物件型別（Object Types）的種類和差異。

7. 【進階題】觀察以下兩種 JavaScript 分別對變數 `foo` 與 `bar` 指派值的程式碼。

```
/* 變數宣告指派式 */
let foo: number = 123;

/* 變數遲滯性指派 Delayed Assignment - 先宣告而後指派 */
let bar: number:

// 純變數指派式
bar = 123;
```

試著用 TypeScript 官方的 Playground 或者是在 VSCode 編輯器實驗並且驗證看看：

- a) JavaScript 裡，以上的變數宣告指派式的範例（也就是宣告變數 `foo` 的那一行程式碼）是屬於敘述式還是表達式？
- b) JavaScript 裡，以上的變數指派式的範例（也就是宣告變數 `bar` 的那一行程式碼）是屬於敘述式還是表達式？
- c) 按照結論 a) 與 b) 以及你對於敘述式以及判斷式的理解，你能夠解釋得出多重指派（Multiple Assignment）的運作原理嗎？

```
/* 先告訴 TypeScript 編譯器 bar 是數字型別 */
let bar: number;

/* 多重指派 */
let foo: number = bar = 1;
```

按照結論 a) 與 b) ，請問以下的註記方式是合理的嗎？

```
/* 先告訴 TypeScript 編譯器 bar 是數字型別 */
let bar: number;

/* 多重指派 */
let foo = (bar = 1) as number;
```

（相信讀者若經過本題的洗禮與摧殘，應該就會知道 JavaScript 的指派式不純然是敘述式或表達式，完全要根據語法使用的情形）

03

深入型別系統 I 基礎篇

► 3.1 深潛之前的準備

3.1.1 TypeScript 編譯設定檔

學習型別系統中，型別本身的特性之前，必須要先注意 TypeScript 本身的編譯設定（Configuration）；如果設定上條件沒有交代清楚的話，後面條目所提到的東西是不精確的，就把它想成編譯設定可以造就出很多種不同 TypeScript 編譯的規則與條件等等。

首先，想要叫出 TypeScript 的編譯設定檔——名為 `tsconfig.json` 的檔案，其中在過往的條目 1.5.1 就已經請讀者下載好的 TypeScript 編譯指令（也就是 `tsc`）可以初始化編譯設定檔：

```
$> tsc --init
```

假設你在某一處（通常是你想要建立 TypeScript 專案的地方）執行此指令時，它就會自動產生 TypeScript 設定檔。圖 3-1 為節選一部分的設定檔內容出來的畫面。

```
{...} tsconfig.json > ...
1  [
2    "compilerOptions": {
3      /* Basic Options */
4      // "incremental": true,
5      // "target": "es5",
6      // "module": "commonjs",
7      // "lib": [],
8      // "allowJs": true,
9      // "checkJs": true,
10     // "jsx": "preserve",
11     // "declaration": true,
12     // "declarationMap": true,
13     // "sourceMap": true,
14     // "outFile": "./",
15     // "outDir": "./",
16     // "rootDir": "./",
17     // "composite": true,
18     // "tsBuildInfoFile": "./",
19     // "removeComments": true,
20     // "noEmit": true,
21     // "importHelpers": true
22   }
23
24   /* Enable incremental compilation */
25   /* Specify ECMAScript target version: 'ES3' (d
26   /* Specify module code generation: 'none', 'co
27   /* Specify library files to be included in the
28   /* Allow javascript files to be compiled. */
29   /* Report errors in .js files. */
30   /* Specify JSX code generation: 'preserve', 'r
31   /* Generates corresponding '.d.ts' file. */
32   /* Generates a sourcemap for each correspondin
33   /* Generates corresponding '.map' file. */
34   /* Concatenate and emit output to single file.
35   /* Redirect output structure to the directory.
36   /* Specify the root directory of input files.
37   /* Enable project compilation. */
38   /* Specify file to store incremental compilati
39   /* Do not emit comments to output. */
40   /* Do not emit outputs. */
41   /* Import emit helpers from 'tslib'. */
42 }
```

圖 3-1 Tsconfig.json 設定檔的部分內容

密密麻麻的設定實在是很多，然而本書不會講這方面的細節，這樣形同把官方的設定直接複製貼在本書，浪費讀者時間；然而讀者若有一些很詳盡的需求，基本上可以在 TypeScript 的官方網站——“Compiler Options”的部分查詢¹。要注意的是，不同版本的 TypeScript² 叫出的初始設定檔可能多多少少也會不同，但應該不至於到完全不一樣。

而本書探討大部分的範例程式碼，型別系統的檢測行為，其中檢查一下 `strict` 選項是不是 `true` 的狀態：

1 <https://www.typescriptlang.org/docs/handbook/compiler-options.html>。

2 作者寫作的當下 TypeScript 為 3.7 版。

```
{  
  "compilerOptions": {  
    /* 其他的設定除非作者提及，否則不變 */  
  
    "true": true,  
  
    /* 其他的設定除非作者提及，否則不變 */  
  }  
}
```

通常初始化時的預設 `strict` 就是 `true`，不過以防萬一還是提醒讀者，想要按照本書的範例程式碼練習的話，請確保先經過以下這幾個步驟：

1. 初始化專案設定檔，使用指令：`tsc --init`。
2. 檢查設定：確認 `strict` 設為 `true`。

這非常重要，若 `strict` 選項沒有開啟，本書講解的程式碼範例之型別判斷基準將會差很多，而且 `strict` 模式也是建議的開發選項。

3.1.2 檢測型別推論與註記的迭代流程

本書在探討並且學習型別系統的過程中，會不停帶領讀者重複這幾個流程：

1. 被動地型別推論（Inference）時，發生的過程與原理。
2. 主動地型別註記（Annotation）的機制與產生的效用。
3. 探討該型別條件下，本身適合被動式推論亦或者積極註記的時機點、優缺等細節。

3.1.3 型別化名介紹——Introduction to Type Alias

最後，也是型別系統的最基礎的成分——型別化名（Type Alias），或者有些人把它稱作型別的別名。簡而言之，就是為型別或者型別的組合取一個新名字的概念。

首先介紹的是基礎的型別化名的宣告方式，假設我們想要有一個代表使用者資料物件的型別 `UserInfo`，以下是簡單的範例程式碼：

```
type UserInfo = {
    name: string;
    age: number;
    interest: string[];
};
```

所以，在條目 2.3.2 時短暫介紹到的 JSON 物件型別的範例：

```
const info: {
    name: string;
    age: number;
    interest: string[];
} = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
};
```

不覺得這樣的寫法很冗長嗎？於是可以在宣告的型別化名 `UserInfo` 上進行簡化的動作！

```
const info: UserInfo = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
};
```

不過型別化名有一個很細微的重點，這些都會與介面（Interface）的介紹時一併討論。

► 3.2 原始型別 Primitive Types

接下來我們要正式進入到型別系統裡很基層的東西——型別本身。作者會儘量按照條目 3.1.2 所敘述的迭代流程進行討論。

3.2.1 型別推論機制

其實要理解或鑽研任何型別的推論機制並不難，就是寫純原生版本的 JavaScript 程式碼——不附帶任何註記的語法，然後檢視 TypeScript 編譯器會對這些值產生什麼樣的反應；因此，探討型別推論的過程，形容得難聽點就是暴力檢測法，而作者也會刻意用些範例刁難編譯器，引出一些可能開發上意想不到的行為，因為某些細節反而是開發過程不經意會遇到的地雷。

首先，條目 2.3.1 已經介紹過原始型別的定義，也就是：表現資料型態的最小單位，而以下的範例，每個變數被指派的值都是屬於原始型別。

```
/* 數字型別 number */
let num = 123;

/* 字串型別 string */
let message = 'Hello world';

/* 布林值型別 boolean */
let truthy = true;

/* Undefined 型別 undefined */
let meansEmpty = undefined;

/* Null 型別 null */
let meansNothing = null;
```

首先，想要知道各個變數被推論的結果，你可以在 VSCode 上將滑鼠移動到該變數，就會跳出 TypeScript 編譯器的推論結果相關訊息。比如說，想要檢測變數 num 被指派之結果，滑鼠放到變數 num 上面就會出現如圖 3-2 的訊息。

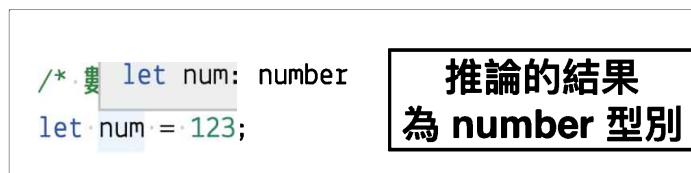


圖 3-2 變數 num 被推論為 number 型別

如果讀者按照這樣的步驟嘗試檢測其他型別之推論結果，一定會得到以下的結果：

- 變數 message 被推論為 string 型別。
- 變數 truthy 被推論為 boolean 型別。
- 變數 meansEmpty 被推論為 undefined 型別。
- 變數 meansNothing 被推論為 null 型別。

以上應該還蠻基本的，那接下來探討變數指派非值（Value），而是其他變數的情形：

```
/* 數字型別 number */
let num = 123;

/* foo 被指派變數 num */
let foo = num;
```

這時的變數 foo 的推論結果如圖 3-2。

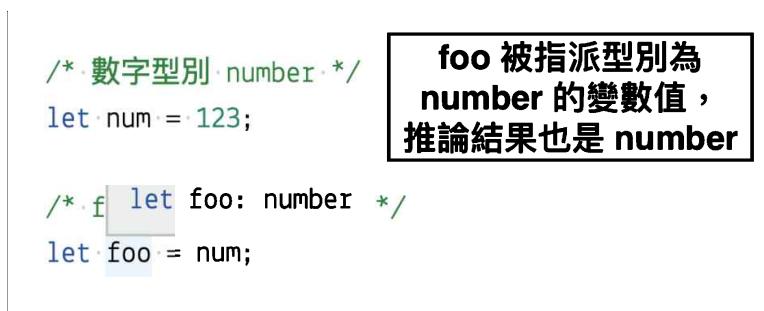


圖 3-2 指派變數 num 的值到變數 foo 裡

也就是說，TypeScript 編譯器型別偵測過程，會沿途傳遞（Propagate）型別的推論結果到後面的程式——這是第一個重點。

另一個問題要問讀者，如果編譯器看到以下的程式碼會怎麼反應？

```
/* 結果是數字型別還是字串型別？——機率各為五成 */
let whatIsThis = (Math.random() > 0.5) ? 123 : '123';
```

首先，`Math.random()` 會回傳數值 $0 \sim 1$ 之間的數值，也就是說，變數 `whatIsThis` 為數字型別的 `123` 或字串型別的 `'123'` 的機率各半。

如果按照人性化的猜測時，根據條目 2.3.6 談到的進階型別中，推論出來的型別應該會以邏輯「或」(OR) 的結果——也就是聯集型別 (Union Type) 來表示。(推論結果如圖 3-3)

```
/* 結果是數字型別還是字串型別？——機率各為五成 */
let whatIsThis = (Math.random() > 0.5) ? 123 : '123';
```

出來的結果的確是 **string**
亦或者是 **number** 的聯集過後之型別

圖 3-3 變數 `whatIsThis` 推論結果為 `number | string`

所以，第二個重點是——遇到類似程式中的分岔點 (Branching Point)，TypeScript 編譯器會將各自的狀況過程與回傳值推論結果，用聯集型別方式來表示。所以以下的程式碼範例將三元運算表達式改成條件敘述式的寫法：

```
/* 結果是數字型別還是字串型別？——機率各為五成 */
let whatIsThis;

if (Math.random() > 0.5) {
    whatIsThis = 123;
} else {
    whatIsThis = '123';
}

whatIsThis; // ← 推論結果為 string | number
```

以上的指派方式，最後一行檢測其推論結果如圖 3-4。

```

if (Math.random() > 0.5) {
    whatIsThis = 123;
} else {
    whatIsThis = '123';
}
let whatIsThis: string | number
whatIsThis;

```

因為兩種情形都有可能發生，
所以推論結果自然而然會有
型別聯集的情形發生

圖 3-4 換成判斷敘述式的寫法也會有類似的效果

另外，假設變數宣告時是使用 `let`，若將被推論過後的變數指派同樣與不同樣的型別的值時：

```

/* 數字型別 number */
let num = 123;

/* num 被指派型別為數字的型別 */
num = 456;

/* num 被指派型別為字串的型別 */
num = 'Hello world!';

```

這裡 TypeScript 編譯器就會出現警告訊息囉，它會在有疑問的程式碼留下紅色波浪狀的記號（如圖 3-5），並且用滑鼠可以檢視訊息（如圖 3-6）。

```

/* num 被指派型別為字串的型別 */
num = 'Hello world!';

```

當 num 變數被指派非 number 型別
之值時，底下出現紅色波浪狀的警告足跡

圖 3-5 紅色波浪狀的記號代表 TypeScript 編譯器對開發者程式碼的疑慮

```

/* 數字型別 number */
let num = 123;
let num: number

Type '"Hello world!"' is not assignable to type 'number'. ts(2322)
Peek Problem No quick fixes available
num = 'Hello world!';

```

num 不給指派除了
number 型別以外的資料

圖 3-6 字串 "Hello world!" 不能夠被指派到被推論為 number 型別的變數 num

而這個就是型別推論的第三個重點，一經推論過後的變數就會只固定接收該型別下的任何值；被指派為不符變數被推論之型別就會出現類似下面的錯誤訊息：

“Type <Some Type> is not assignable to type <Another Type>”

»» TypeScript 型別推論的運作特點

1. **傳遞性**：一經推論過後的變數，會將其推論結果陸續傳遞到後面的程式碼以輔助其他變數、表達式的推論結果。
2. **匯集性**：若變數的推論可能遇到程式中的分岔點（條件敘述、三元運算表達式等）而有所不同，TypeScript 會綜合各個分岔點的推論結果並匯聚成聯集型別（Union Type）。
3. **固定性**：一經宣告與推論過後的變數，後續任何指派到該變數的值必須符合該變數被推論之型別的範疇。

3.2.2 型別註記機制

首先，註記的語法早在條目 2.2 探討過了，甚至還細分成「註記」與「斷言」兩種——但作者早已解釋過兩者之差異性，而往後為了簡化解說的部分，以後「型別註記」就代表作者在講的範圍就涵括「註記」跟「斷言」這兩種喔！

以下的範例就是針對原始型別的簡單註記：

```
/* 數字型別 number */  
let num: number = 123;  
  
/* 字串型別 string */  
let message: string = 'Hello world';  
  
/* 布林值型別 boolean */  
let truthy: boolean = true;  
  
/* Undefined 型別 undefined */  
let meansEmpty: undefined = undefined;
```

```
/* Null 型別 null */  
let meansNothing: null = null;
```

其實看起來沒什麼大不了，就是加上註記語法而已，但是加上去註記某部分程度來說就是增加可讀性、進行補充，亦或者是協助 TypeScript 編譯器在靜態分析的過程。（但過分地使用註記可能導致程式碼反而會讀起來過於冗贅，參見條目 2.1.2）

而 TypeScript 靜態分析過程，若有註記型別對照不符合該值之型別就會有警訊出來，例如以下的範例程式碼，以及圖 3-7 的錯誤訊息畫面：

```
/* 數字型別 number 塞字串 string */  
let num: number = '123';
```



圖 3-7 TypeScript 編譯器的靜態分析部分出現的警訊

當然，註記的效果固然會使得該變數往後被指派的值必須為被註記之型別。

3.2.3 遲滯性指派 Delayed Initialization

這裡要探討一個很細節的情境——遲滯性指派，相信讀者初學 JavaScript 時一定會碰過這種情形：「先宣告變數而後指派值」。譬如：

```
/* 先宣告 num 變數 */  
let num;  
  
/* 而後指派值 */  
num = 123;
```

首先呢，剛開始宣告時的變數推論結果會是如何？由於並沒有值或表達式可以讓 TypeScript 編譯器進行推論，於是就會有一個特殊型別來代替——也就是 **any** 型別³，代表該變數為任意型別都有可能。(如圖 3-8)



圖 3-8 剛宣告時，變數 num 被推論為 any 型別

而經過指派過後，變數 num 的型別就被鎖定為 **number** 型別，原因是有了值或表達式來依據。(如圖 3-9)

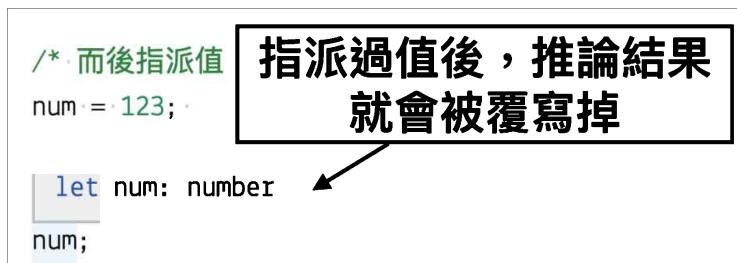


圖 3-9 原本推論為 any 型別之結果被覆寫掉了

另外，在該變數被指派確定的結果前有被使用時，例如：

```

/* 先宣告 num 變數 */
let num;

/* 使用 num */
console.log(num);

/* 而後指派值 */
num = 123;

```

3 特殊型別參見條目 4.6。

當然，在未經過任何處理的狀況，原本被判定為 `any` 的型別的變數 `num`，通常 JavaScript 裡還未被指派值但已經宣告出來的變數初始化的值為 `undefined`，自然而然就是 `undefined` 型別囉。(圖 3-10)



圖 3-10 預設的情形是 `undefined`

以上是宣告時，變數自然被 TypeScript 編譯器推論的過程，看起來沒有什麼大不了。

但接下來若是談論到配合註記的語法就不一定了，如果說範例改成這樣子呢？

```
/* 先宣告 num 變數並且註記為 number 型別 */
let num: number;

/* 而後指派值 */
num = 123;
```

如果讀者將以上的程式碼正常輸入到 VSCode 裡，TypeScript 編譯器事實上是不會出現警告訊息的，變數 `num` 從頭到尾會被認定為型別 `number`。

然而，如果在指派值到該變數前，使用了該變數，TypeScript 編譯器檢測結果會跳出錯誤訊息喔。(如圖 3-11)

```
/* 先宣告 num 變數並且註記為 number 型別 */
let num: number;

/* 使用變數 num */
console.log(num);

/* 而後指派值 */
num = 123;
```

```

/* 先宣告 num */
let num: number
let num: num
Variable 'num' is used before being assigned. ts(2454)

/* 使用變數 */
console.log(num);
Peek Problem No quick fixes available

/* 而後指派值 */
num = 123;

```

變數再指派值之前
被使用就會被丟出警訊

圖 3-11 變數 num 在指派前被使用時，跳出的錯誤訊息

檢視錯誤訊息之內容就是避免開發者在變數還未定前，該變數被註記為型別 `number` 但是卻出來的結果是 `undefined`。

這時若我們想要讓該變數除了為 `number` 型別外，但是也想要有值（例如 `undefined` 或 `null`）代表該變數為空值狀態，此時你可以運用聯集型別的方式進行處理：

```

/* 先宣告 num 變數並且註記為 number | undefined 聯集型別 */
let num: number | undefined;

/* 使用變數 num，此時就沒有錯誤訊息囉！ */
console.log(num);

/* 而後指派值 */
num = 123;

```

以上的範例就由讀者自行試試看吧！

»» 遲滯性指派 Delayed Initialization

1. 定義：變數宣告時不進行指派值的動作，而延後至程式後面的某處再指派值的情形。
2. 若變數採用遲滯性指派的策略，宣告方式如下：

```
let <variable>;
```

該變數 `<variable>` 會直接被推論為 `any` 型別，初始化之值則為 `undefined`。

3. 若變數採用遲滯性指派的策略，並且有被註記特定型別 T，宣告方式如下：

```
let <variable>: T;
```

該變數 `<variable>` 會直接被推論為 T 型別，嚴格上來說，此時並沒有初始化之值可言，因為只要是在被正式指派值前使用該變數時，就會被 TypeScript 靜態分析過後丟出警告訊息如下：

```
Variable '<variable>' is used before being assigned.
```

► 3.3 JSON 物件型別 JSON Object Type

3.3.1 型別推論機制

相信讀者應該有開始慢慢掌握到本書的教學風格，我們就繼續看下個範例：

```
/* 先宣告 info 變數並且指派簡單的 JSON 物件 */
let info = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
};
```

以上的範例（讀者應該覺得有點膩）簡單地展示了 JSON 物件格式的推論機制（如圖 3-12）。

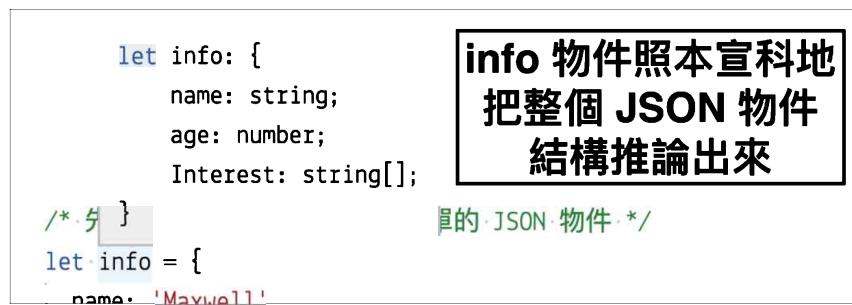


圖 3-12 整個變數 `info` 被推論出來整個 JSON 物件的結構

所以假設我們在開發過程中，臨時想要使用變數 `info` 但卻忘記有哪些物件屬性或方法可以呼叫時，請讀者記得：我們可以檢視它的推論結果！

而根據條目 1.5.2 提及到的部分，我們也可以善用 VSCode 的自動補齊功能（Autocomplete），後續若想要使用變數 `info`，在呼叫該變數的屬性或方法前，就會跳出一個自動補齊的選項視窗。（如圖 3-13）

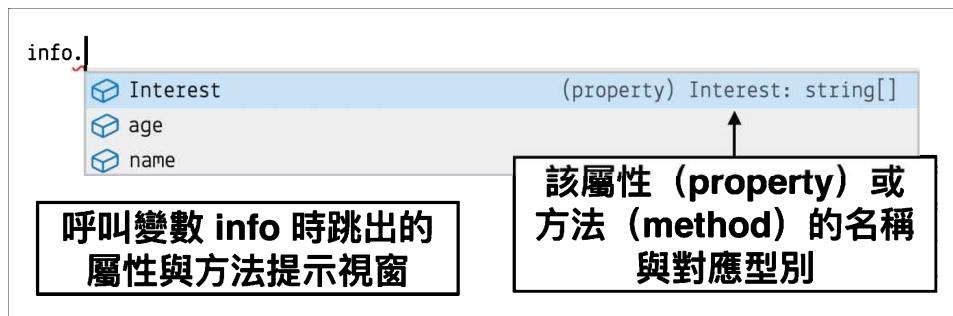


圖 3-13 呼叫變數 `info` 的屬性前，跳出的輔助視窗

另外，由於物件呼叫屬性或方法也是隸屬於表達式（Expression）⁴，因此整個表達式也可以被型別推論出結果來，請看以下的範例：

```
/* 先宣告 info 變數並且指派簡單的 JSON 物件 */
let info = { /* name, age, interest */ };

/* 將 name 屬性拔出來，指派到變數 myName，此時 myName 被推論為 string 型別 */
let myName = info.name;
```

/* 將 let myName: string 派到變數 myName，此時 myName 被推論為 string 型別 */
let myName = info.name;

myName 之推論結果為 string 型別

圖 3-14 變數 `myName` 照樣可以從被指派的表達式推論出正確的型別

3.3.2 型別註記機制

既然 TypeScript 編譯器暴力地把整個 JSON 物件之結構給推論出來，理應來說我們也可以暴力地註記整個物件結構上去：

4 參見條目 2.2.2 敘述式與表達式比較部分。

```
/* 先宣告 info 變數，註記後並且指派簡單的 JSON 物件 */
let info: {
    name: string;
    age: number;
    interest: string[];
} = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
};
```

以上的程式碼推論結果會跟圖 3-12 一樣，因此不放上結果圖，就由讀者自行驗證。

不過考驗讀者的眼力，你有沒有發現 JSON 物件的型別與值的寫法差異。

```
/* 型別寫法的分格是用分號 ";" */
{
    name: string      ; // ← 這是分號
    age: number       ; // ← 這是分號
    interest: string[] ; // ← 講了三次了，這是分號
}

/* JSON 物件值的分格是用逗號 "," */
{
    name: 'Maxwell'           , // ← 這是逗號
    age: 18                  , // ← 這是逗號
    interest: ['drawing', 'programming'] , // ← 講了三次了，這是逗號
};
```

首先，物件的值的寫法用逗號分隔（Separator 或 Delimiter 都有人在講）是絕對必須遵守的事情；但型別部分，在 TypeScript 裡，你想用分號分隔或者是逗號分隔，都可以（看你爽用哪種）：

```
/* 型別寫法的分格用逗號也可以 "," */
{
    name: string      , // ← 逗號也可以
```

```
age: number      ,
interest: string[],
}
```

事實上，TypeScript 相關討論主題的社群還為了這件事情爭論到底適合用哪種寫法，其中一個最扯的原因根本上是因為 TypeScript 編譯過程中，在 1.5 版本（包含）還未支援逗號之前，就算你用逗號分隔，甚至是省略分隔只用換行竟然也都可以編譯成功⁵。

然而正常的開發者應該也沒想過這麼細的問題，本書教的也會是建議寫法，因此就不會在細節上鑽牛角尖（而這些不正常細節也應該是由作者告訴讀者以防讀者踩雷）。

另外，讀者肯定覺得暴力將結構都展示出來的寫法很冗贅，所以早在條目 3.1.3 介紹型別化名（Type Alias）時，藉由獨立出 JSON 物件型別的結構，替它取個新名稱，就可以整理並且簡潔化程式碼的寫法。

```
/* 先宣告名為 PersonalInfo 的型別化名 */
type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};

/* 宣告 info 變數，註記為 PersonalInfo 型別 */
let info: PersonalInfo = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
};
```

不過這時由於是直接告訴編譯器：「變數 `info` 之值必須屬於型別 `PersonalInfo`」，此時將鼠標指到該變數上，就會出現如圖 3-15 的結果。

5 讀者自己實驗玩玩還好，但不要真的在正式專案上亂搞。

```
/* 里 let info: PersonalInfo |sonalInfo 型別 */
> let info: PersonalInfo = { ...  
};
```

儘管確實是 **PersonalInfo**
型別，但結構的描述部份不見了！

圖 3-15 變數 info 被註冊為 PersonalInfo 型別，但是無法檢視內部的結構

上個條目中，圖 3-13 會顯示出完整的結構，但圖 3-15 由於型別化名的宣告與註冊，導致結構訊息不見了。此時若想要查詢 PersonalInfo 型別的結構時，這一次我們可以換成檢視 PersonalInfo 型別本身的推論訊息。(如圖 3-16)

```
type PersonalInfo = {  
    name: string;  
    age: number;  
    interest: string[];  
/* 宣告 in */  
> let info: PersonalInfo = {  
};
```

檢視 **PersonalInfo**
的結構

圖 3-16 型別 PersonalInfo 的結構被看光光了

不過這裡作者用個範例再為難讀者一下：

```
/* 先宣告名為 PersonalInfo 的型別化名 */
type PersonalInfo = { /* name, age, interest */ };

/* 宣告名為 PersonalInfoOrNull 的型別化名 */
type PersonalInfoOrNull = PersonalInfo | null;

/* 宣告 info 變數，註冊為 PersonalInfo 型別 */
let info: PersonalInfoOrNull = { /* name, age, interest */ };
```

從以上範例可以看到，這一次多增加一層型別 PersonalInfoOrNull 代表變數 info 可為 PersonalInfo 或 null 型別。

這時你使用鼠標再次檢視型別 PersonalInfoOrNull 的結果如圖 3-17。

```

/* 宣告 info 變數，註記為 PersonalInfo 型別 */
> let info: PersonalInfoOrNull = ...
};



這一次連結構都不可見了，  
因為多了 null 的聯集


```

圖 3-17 檢視化名 PersonalInfoOrNull 的結果為 PersonalInfo | null

此時想要查詢詳細的型別結構時，就要搬出條目 1.5.3 最後提到——使用 VSCode 提供的快速查詢的技巧。這裡一樣可以複習一下，首先將鼠標滑到型別化名 PersonalInfoOrNull 上，按下 Ctrl (MacOS 則是按下 ⌘) 的同時，點一下滑鼠就會立馬跳到化名 PersonalInfoOrNull 的宣告位置。(圖 3-18)

鼠標會跳到型別化名的宣告處

圖 3-18 使用快速查詢功能就可以跳到型別原本的宣告處

這個快速查詢的功能非常重要，一被導向到化名 PersonalInfoOrNull 的宣告處，你就可以再使用快速查詢技巧，導向到化名 PersonalInfo 的宣告處，檢視其代表之型別結構。

3.3.3 物件的完整性不容許被動搖

JSON 物件的推論跟註記的結果都已經在前兩小條目都討論過後，接下來的重點要放在對這些物件做的操作（Operation）與效果；譬如插入新的屬性或方法。

首先，單純對推論的物件做新增屬性的動作：

```

/* 先宣告 info 變數後指派簡單的 JSON 物件 */
let info = {
  name: 'Maxwell',

```

```
age: 18,  
interest: ['drawing', 'programming'],  
};  
  
/* 新增屬性 */  
info.email = 'example@mail.com';
```

如果讀者試著將該範例讓 TypeScript 編譯器分析，它是不會允許你直接對物件進行新增屬性的動作。

```
/* 先宣告 info 變數後指派簡單的 JSON 物件 */  
let info = {  
    name: 'Maxwell',  
    age: any  
    int  
};  
    Property 'email' does not exist on type '{ name: string; age:  
    number; interest: string[]; }'. ts(2339)  
  
/* 新 Peek Problem No quick fixes available  
info.email = 'example@mail.com';
```

email 屬性本不存在變數 info 代表的物件裡

圖 3-19 JSON 物件在宣告的當下後，不准你在後面亂新增屬性

然而，檢視刪除屬性這個動作時：

```
/* 先宣告 info 變數後指派簡單的 JSON 物件 */  
let info = { /* name, age, interest */ };  
  
/* 刪除屬性 */  
delete info.name;
```

這樣的動作反而不會造成任何問題或警告訊息的產生。(圖 3-20)

```
/* 先宣告 info 變數後指派簡單的 JSON 物件 */  
> let info = { ...  
};  
  
/* 刪除屬性 */  
delete info.name;
```

很安靜，沒有任何警訊

圖 3-20 刪除屬性時卻意外的沒有任何訊息產生

而且如果後面再將範例的 `info.name` 叫出來並指派到其他變數時，該變數的推論結果反而不是 `undefined`，而是 `string` 型別。(圖 3-21)

```
/* 刪除屬性 */
delete info.name;

let myName: string
let myName = info.name;
```

就算刪除了屬性，
推論結果依然存在

圖 3-21 推論結果依然是刪除屬性前的 `string` 型別

作者認為這是很奇怪的現象，畢竟刪掉了屬性理應來說要出現 `undefined` 型別亦或者是出現警訊（但條目 3.3.5 提到的 `readonly` 這個唯讀屬性操作子就會改變刪除屬性沒有出現警訊的怪異行為）。

討論完新增跟刪除的行為，第三種常見的是覆寫（Overwrite）JSON 物件某特定屬性或方法的行為：

```
/* 先宣告 info 變數後指派簡單的 JSON 物件 */
let info = { /* name, age, interest */ };

/**
 * 覆寫屬性的值，其中 info.name 期望被接收的值應該要為
 * string 型別，而 'Martin' 也屬於 string 型別
 */
info.name = 'Martin';
```

事實上，以上覆寫的行為是可以被 TypeScript 接受的，主要原因是覆寫的值 '`Martin`' 為 `string` 型別，而剛好符合 `info.name` 代表的 `string` 型別，這裡就請讀者自行驗證。

而理所當然地，如果遇到型別衝突的情形，以下的案例絕對會出錯：

```
/* 先宣告 info 變數後指派簡單的 JSON 物件 */
let info = { /* name, age, interest */ };

/**
 * 覆寫屬性的值，其中 info.name 期望被接收的值應該要為
```

```
* string 型別，而 123 則屬於 number 型別，因此造成衝突
*/
info.name = 123;
```

錯誤訊息可以參考圖 3-22。

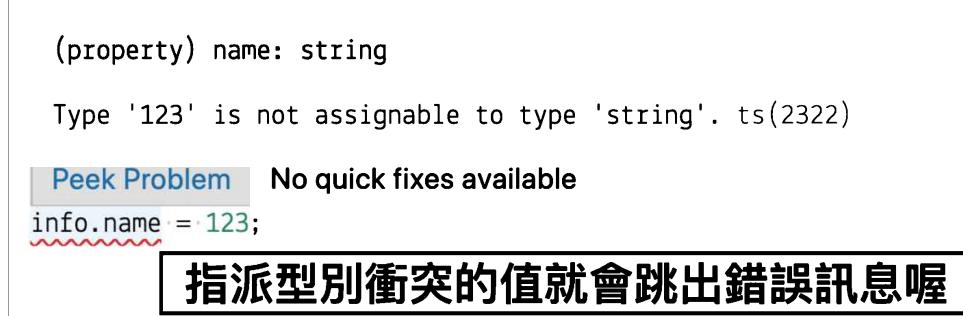


圖 3-22 數值 123 不能被指派到型別推論為 string 的 `info.name` 表達式

覆寫除了是單個屬性或方法覆寫外，還有一種情形是整個 JSON 物件的覆寫：

```
/* 先宣告 info 變數後指派簡單的 JSON 物件 */
let info = { /* name, age, interest */ };

/* 使用暴力覆寫法，整個 JSON 物件連同結構細節丟進去 */
info = {
    name: 'Martin', // ← 特別覆寫掉 name 屬性
    age: info.age,
    interest: info.interest
};

/**
 * 使用 ES7 Rest-Spread，將原本的 info 物件的鍵值對
 * 都複製後，再挑選需要更改的鍵值對
 */
info = { ...info, name: 'Martin' };
```

以上的範例兩種寫法，分別是暴力覆寫法或者是藉由 ES7 出來的匯集 - 展開運算子（Rest-Spread Operator）的寫法，都是等效的。

另外，因為新的覆寫物件結構符合變數 `info` 本身推論的 JSON 物件結構，所以全面覆寫的情形是 TypeScript 編譯器可以接受的行為，請讀者自行驗證。

不過如果出現一點點結構不符合的情形，當然就會跳出警告訊息了。

```
/* 故意將 name 指派為非 string 型別的值 */
info = { ...info, name: 123 };
```

警告訊息跟圖 3-22 差不多，差在顯示的結構不符合的情形比較複雜些。(圖 3-23)

```
(property) name: string
Type 'number' is not assignable to type 'string'. ts(2322)
001.primitive-type-inference.ts(7, 3): The expected type comes from property
'name' which is declared here on type '{ name: string; age: number; interest:
string[]; }'
/* 故意將 name 指派 Peek Problem No quickfix available
info = { ...info, name: 123 };
```

型別衝突就會牽拖出一大串錯誤訊息

圖 3-23 錯誤訊息有點大，所以作者會進行解釋

完整的訊息如下：

```
(property) name: string
Type 'number' is not assignable to type 'string'. ts(2322)
<檔案名稱>(<行數>, <字元位置>): The expected type comes from property
'name' which is declared here on type '{ name: string; age: number; interest:
string[]; }'
```

其實仔細看裡面的內容：屬性 `name` 的型別推論是 `string`，但它偏偏接收到的值是 `number` 型別，所以才會吐出這一串警告訊息。

以上純粹是對 JSON 物件進行操作時，需要注意的重點，由於細節看起來很多，但只要遵照以下的結論，基本上不會有很大的問題出現。

»》操作 JSON 物件時的注意事項——保持 JSON 物件的完整性

一旦某變數被推論為 JSON 物件時，該變數必須遵守以下規則：

1. 不能任意新增屬性。

2. 覆寫掉特定屬性或方法時，覆寫之值的型別不能與該屬性或方法推論過後之型別產生衝突。
3. 覆寫掉整個變數的 JSON 物件值，覆寫之 JSON 物件的值的型別結構不能與變數原先的型別結構產生衝突。

簡而言之，不能夠對物件進行型別結構上的破壞，保持物件的完整性。

唯一的例外是使用 `delete` 關鍵字進行刪除 JSON 物件的屬性或方法的動作。

3.3.4 選用屬性 Optional Property

由於對於 JSON 物件結構的規範本身很嚴格，只要結構不符 TypeScript 編譯器就會發出警訊，因此會有另一個機制讓我們創造出結構相對鬆散些的 JSON 物件。

首先是以下這個範例，假設我們有個型別化名 `PersonalInfo`，不一定要求 JSON 物件有 `age` 屬性（畢竟並不是所有人可能會想要放上一些敏感訊息的），我們可以在該屬性宣告時，旁邊附帶一個問號：

```
/* 先宣告名為 PersonalInfo 的型別化名 */
type PersonalInfo = {
    name: string;
    age?: number;           // ← 注意這裡的屬性 age 被標上 ? 了
    interest: string[];
};

/* 宣告 info 變數，註記為 PersonalInfo 型別，可以省略 age 屬性 */
let info: PersonalInfo = {
    name: 'Maxwell',
    interest: ['drawing', 'programming'],
};
```

以上的範例程式碼是完全沒問題的，若查看化名 `PersonalInfo` 代表的型別結構中，屬性 `age` 對應的 `number` 與 `undefined` 進行了聯集式的複合動作。(如圖 3-24)

```

type PersonalInfo = {
  name: string;
  age?: number | undefined;
  interest: string[];
}

/* 告知 info */
let info: PersonalInfo = {
  name: 'Maxwell';
}
  
```

age 屬性可為 `undefined`
也可為 `number`

即，可以省略 `age` 屬性 */

圖 3-24 `age` 成為了可被省略的屬性

不過可能有些讀者會問，這跟以下的寫法有什麼不同？

```

/* 先宣告名為 PersonalInfo 的型別化名 */
type PersonalInfo = {
  name: string;
  age: number | undefined; // ← 注意這裡的屬性 age 被改成 number | undefined
  interest: string[];
}
  
```

這樣的寫法會讓 TypeScript 編譯器誤以為屬性 `age` 是不可被省略的屬性外，該屬性必須得被指派為型別 `number` 或型別 `undefined` 的值。

重點在「必須得被指派」這幾個字，所以如果改成上述的樣貌，變數 `info` 被宣告時，少了 `age` 屬性依然會被丟出警告。(圖 3-25)

```

/* 先宣告名為 PersonalInfo 的型別化名 */
type PersonalInfo = {
  name: string;
  age: number | undefined;
  interest: string[];
}

let info: PersonalInfo = {
  name: 'Maxwell',
  interest: ['drawing', 'programming'];
}

001.primitive-type-inference.ts(6, 3): 'age' is declared here.

/* └ Peek Problem No quick fixes available
let info: PersonalInfo = {
  name: 'Maxwell',
  interest: ['drawing', 'programming'];
}
  
```

「省略」與「值為 `undefined`」在 JSON 物件的屬性是不同的概念

圖 3-25 錯誤訊息一樣是在描述結構不符的狀況

»» JSON 物件選用屬性

若想在 JSON 物件型別 T_{object} 裡宣告出可以被省略的某特定屬性 $<Prop>$ ，則寫法如下：

```
type Tobject = {
    /* 其他屬性對應型別的宣告 */
    <Prop>?: <TProp>
};
```

其中 $<T_{Prop}>$ 為 $<Prop>$ 屬性對應之預設型別，然而因為 $<Prop>$ 被註記為選用屬性，因此在型別推論時，其推論結果會跟 `undefined` 進行聯集的作用，也就是 $<T_{Prop}> \mid undefined$ 。

因此任何變數被註記為型別 T_{Prop} 時，以下的兩種寫法都可以：

```
let foo: Tobject = {
    /* 對應 Tobject 型別結構宣告的其他屬性 */
    <Prop>: <VProp> // ← 不省略 <Prop> 屬性
};

let bar: Tobject = {
    /* 除了 <Prop> 外，對應 Tobject 型別結構宣告的其他屬性 */
};
```

其中 $<V_{Prop}>$ 為代表型別 $<T_{Prop}>$ 的值。

3.3.5 唯讀屬性 Read-Only Property

JSON 物件除了有條目 3.3.4 講述的選用屬性外，它還有另一個很好用的功能，避免開發者擅自覆寫屬性的功能，那就是唯讀屬性操作符（`Readonly` Property Operator）。

以下面的程式碼為例：

```
/* 先宣告名為 PersonalInfo 的型別化名 */
type PersonalInfo = {
    name: string;
    readonly age: number; // ← 將 age 設定為唯讀屬性
```

```

    interest: string[];
};

/* 宣告 info 變數，註記為 PersonalInfo 型別 */
let info: PersonalInfo = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
};

```

型別 `PersonalInfo` 推論結果會在 `age` 屬性上多出 `readonly` 這個標示。(如圖 3-26)

```

type PersonalInfo = {
    name: string;
    readonly age: number;
    interest: string[];
}

/* 宣告 info */
let info: PersonalInfo = {
    name: 'Maxwell'
}

```

圖 3-26 唯讀屬性的標示出現在型別化名的結構裡

一旦該屬性出現唯讀的標示，理所當然地，對其屬性進行覆寫的動作絕對會讓程式碼吐出錯誤訊息。(圖 3-27)

```

/* 欲覆寫 info.age 之值，然而 age 為唯讀屬性時就會引發錯誤 */
info.age = 20;

```

(property) age: number

Cannot assign to 'age' because it is a read-only property. ts(2540)

/* 欲 Peek Problem No quick fixes available

info.age = 20;

錯誤訊息都跟你說是 read-only 屬性了

圖 3-27 `age` 屬性為唯讀狀態時，硬給它覆寫時會出現的錯誤訊息

另外，唯讀屬性就代表程式碼也禁止你刪掉它。(如圖 3-28)

```
/* 欲刪除 info.age 之值，然而 age 為唯讀屬性時就會引發錯誤 */
delete info.age;
```

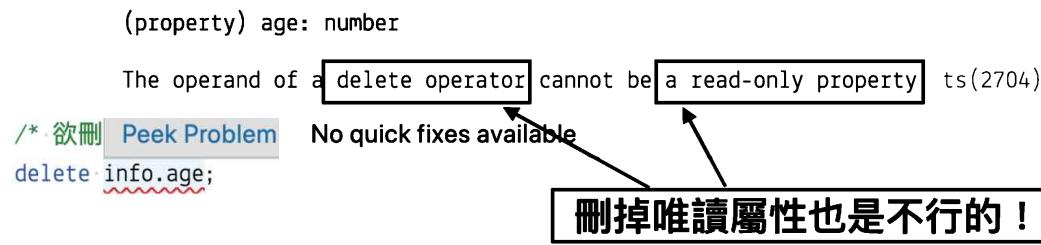


圖 3-28 刪除掉唯讀屬性時跳出之錯誤警訊

不過，儘管裡面有唯讀狀態的屬性，覆寫全部的 JSON 物件基本上依然是允許的行為，畢竟完整覆寫 JSON 物件形同是換掉物件的參照（Reference）⁶，而非物件的值（Value）本身，畢竟唯讀操作符綁定的目標是「值的屬性」。因此以下的範例就是可以接受的操作，就讓讀者自行驗證吧：

```
/* 先宣告 info 變數後註記為 PersonalInfo 型別，指派簡單的 JSON 物件 */
let info: PersonalInfo = { /* name, age, interest */ };

/**
 * 使用暴力覆寫法，整個 JSON 物件連同結構細節丟進去；雖然 age 為唯讀屬性，
 * 但唯讀效果並不是綁在變數本身，所以變數被整個覆寫是可以被接受的
 */
info = {
    name: 'Martin', // ← 特別覆寫掉 name 屬性
    age: info.age, // ← 儘管 age 是唯讀屬性，但這裡只有讀（Read）的操作
    interest: info.interest
};
```

最後，如果是想要讓變數 `info` 中的所有屬性變成唯讀狀態，一種是將型別化名裡宣告的所有屬性通通標上唯讀屬性操作符：

6 JavaScript 物件的傳法是傳它的參照（Reference），也就是說，每個物件的值被建立的那一刻都有對應的一個參照，所以通常看到物件跟物件直接做邏輯相關的比較時，僅管結構相同，例如：`{ a: 123 } === { a: 123 }`，表面上結果是 `true`，但實際會回傳 `false` 的值，原因在於兩個物件代表的參照是不一樣的。

```
/* PersonalInfo 通通都是唯讀屬性 */
type PersonalInfo = {
    readonly name: string;
    readonly age: number;
    readonly interest: string[];
};
```

但不建議這種暴力寫法的原因有幾種：一是麻煩；二是型別屬性若硬給它設成唯讀狀態會造成後續開發時，使用該型別化名的彈性（自由度）降低；另外，如果該型別定義是從第三方套件來的，那麼強行竄改型別的宣告定義反而可能會造成原來套件壞掉的機率很大，你很難保證該套件沒有對註記為該型別的 JSON 物件進行屬性覆寫相關動作。

而如果想要在開發時使用正常的型別化名的定義（如下）：

```
/* PersonalInfo 通通都是可以被任意操作的 */
type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};
```

但又想要避免開發時使用的 JSON 物件太過自由到屬性可以被覆寫，這時就要用到進階型別的寫法；不過作者這邊暫時略過細節，就給讀者看寫法如何：

```
/* 宣告 info 變數，註記為 Readonly<PersonalInfo> 型別 */
let info: Readonly<PersonalInfo> = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
}
```

以上的寫法等效於將 **PersonalInfo** 的化名宣告裡的所有屬性加上唯讀操作符。所以每個屬性都成了唯讀狀態，然而這個 **Readonly<T>** 的細節就此先跳過⁷。

7 **Readonly<T>** 為泛用型別的應用，請參見本書第 7 章。

但是讀者可能覺得這東西根本就是在定義所謂的常數（Constant）吧？也就是不能夠完全被覆寫值的變數。所以下一個範例程式碼照理來說等於 `Readonly<T>` 的寫法？

```
/* 使用 const 告知 info 變數 */
const info: PersonalInfo = { /* name, age, interest */ };
```

如果你這麼想的話，那你可就錯了！關於這個部分的比較，就放在本章的練習好好讓讀者思考一下，到底問題出在哪裡～（關鍵在覆寫方式：覆寫單一屬性與覆寫掉整個變數的值）

»» JSON 物件唯讀屬性

假設宣告 JSON 物件之型別 `Tobject`，裡面的結構中，某個屬性 `<Prop>` 被標上唯讀屬性操作符，寫法如下：

```
type Tobject = {
    /* 其他屬性對應型別的宣告 */
    readonly <Prop>: <TProp>;
};
```

其中，`<Tprop>` 為 `<Prop>` 對應之型別。

任何被註記為 `Tobject` 型別的變數，不能夠覆寫掉 `<Prop>` 屬性。

► 3.4 函式型別 Function Object Type

3.4.1 型別推論機制

「函式型別絕大部分情形是 TypeScript 無法推論的。」

以上這句話可能會令人感到 Shock 來 Shock 去的，但依然還是有些函式——就算不去主動註記函式內部的輸入參數之型別與輸出型別，TypeScript 照樣可以推論出結果。

其中一種最明顯的案例是——函式宣告時無任何參數宣告（Argument）時，就算不註記輸出之型別，大致上可以讓 TypeScript 推論出結果。

```
/* 宣告函式 universalNumber */
function universalNumber() {
    return 42;
}
```

讀者如果把自己的腦袋當成 TypeScript 編譯器，靜態分析時看到以上的程式碼，沒有輸入參數，至於輸出型別你會如何判斷？當然是看函式回傳（return）那一行到底是回傳什麼樣的值（或者是表達式）。

所以該 `universalNumber` 函式的推論結果（如圖 3-29 所示），回傳的型別被推論為 `number` 應該算是還蠻正常的結果。

```
/* 宣告函式 universalNumber(): number
function universalNumber() {
    return 42;
}
```

依據 return 的結果，
推論之輸出結果為 number

圖 3-29 推論結果之型別為 `() => number`

另外，如果遇到程式分岔點（Branching Point），就如同條目 3.2.1 探討過的，出現類似三元運算表達式，又或者是判斷敘述式等，TypeScript 編譯器就會採取匯集型別可能出現的結果並聯集（Union）起來。（以下的程式碼，函式 `numberOrString` 的推論結果如圖 3-30）

```
/* 宣告函式 numberOrString */
function numberOrString() {
    const probability = Math.random();

    if (probability > 0.5) {
        return probability; // ← 五成機率回傳數字型態結果
    } else {
        return probability.toString(); // ← 五成機率回傳字串型態結果
    }
}
```

```
/* 宣告函 function numberOrString(): string | number
function numberOrString() {
    const probability = Math.random();
    if (probability > 0.5) {
```

回傳結果之型別為 string 或 number

圖 3-30 由於遭遇程式的分岔點，推論結果之型別為聯集過後的複合型別

不過如果換成以下的範例，將回傳的結果，原本是未定的數字型別或字串型別的值，改成確定的數字型別值 42 或字串型別值 '42'。

```
/* 宣告函式 numberOrString */
function numberOrString() {
    const probability = Math.random();

    if (probability > 0.5) {
        return 42; // ← 五成機率回傳數字型態結果
    } else {
        return '42'; // ← 五成機率回傳字串型態結果
    }
}
```

讀者可能以為會跟圖 3-4（位於條目 3.2.1）的範例一樣，推論結果為 `number | string`；實際上推論結果（如圖 3-31）為更精確的 `42 | '42'`。

```
/* 宣告函 function numberOrString(): 42 | "42"
function numberOrString() {
    const probability = Math.random();

    if (probability > 0.5) {
        return 42; // ← 五成機率回傳數字型態結果
    } else {
```

推論結果為數字 42
或字串的 '42'

圖 3-31 推論結果反而是確切的值進行聯集的結果

有些人可能覺得這個推論結果是錯誤的，應該是更寬鬆的 `number | string`，但實際上這是合理的推論行為，因為它跟：

```
/* 結果是數字型別還是字串型別？——機率各為五成 */
let numberOrString;
const probability = Math.random();
```

```

if (probability > 0.5) {
    numberOrString = 42;
} else {
    numberOrString = '42';
}

numberOrString; // ← 推論結果為 string | number

```

差就差在：函式的範例裡，無法在函式外部更改函式內部的定義，也就是說，函式內部回傳的值是無法在外面修改它的；相對地，第二個使用變數的範例，你可以等判斷敘述式跑完之後，再度更改變數的值。

因此函式可以將推論結果限縮到直接用明文（Literal）⁸的方式表示（也就是你剛剛看到的推論結果為 `42 | '42'` 的形式）。

輸出部分還有一個案例還沒被討論到，那就是萬一函式本身沒有輸出時，推論結果為何？以下就舉最簡單的例子：

```

/* 宣告函式 greeting */
function greeting() {
    console.log('Hello world!');
}

```

以上的函式沒有任何 `return` 敘述式，推論結果如圖 3-32。

```

/* 宣告函式 greeting()
function greeting(): void
function greeting(){
    console.log('Hello world!');
}

```

**函式若沒有輸出時，
則輸出的型別為 void**

圖 3-32 若函式找不到 `return` 敘述式，就代表回傳的型別是空的，也就是 `void`

推論結果其實很簡單，就是 `() => void`；其中，單字 `Void` 的意思是「虛無」，也就是說函式的回傳是空的，但精確的說法是：「函式回傳的結果根本不重要，或無意義」。

⁸ 明文（Literal）的定義參見條目 2.3.3，而明文型別的完整討論內容參見條目 3.6。

如果是有 `return` 敘述式，但是回傳的結果沒有寫進程式碼，亦或者是指沒有值或表達式的狀況下使用 `return` 的話，也會有相似的結果。(以下的程式碼推論結果如圖 3-33)

```
/* 宣告函式 returnsVoid */
function returnsVoid() { return; } // ← 單純一個 return 敘述式
```

```
/* 宣告函式 returnsVoid()
function returnsVoid(): void
function returnsVoid() { return; }
```

結果是無意義，
所以也是 Void

圖 3-33 空的 `return` 敘述式，該函式之輸出也會是 `void` 型別

這裡就沒有太多東西需要進行補充。

最後我們要討論函式有輸入參數的情形，而輸入參數本質上是無法被推論的。原因是由於輸入參數的型別是要由開發者使用、呼叫該函式時才能確定，就舉以下的例子來說：

```
/* 宣告函式 echo */
function echo(input) {
    return input;
}
```

首先，以上的範例函式 `echo`，單純就是回傳使用者呼叫該函式時所填入的參數；讀者若單純用原生 JavaScript 宣告此函式時，會出現的錯誤訊息如圖 3-34 顯示。

```
(parameter) input: any
Parameter 'input' implicitly has an 'any' type. ts(7006)
/* 宣告函式 echo
function echo(input) {
    return input;
}
```

參數 `input` 被“隱性地”
認定為 `any` 型別

圖 3-34 單純宣告函式時，輸入參數出現的錯誤訊息

錯誤訊息中用了“`implicitly`”（隱性地）這個單字，完整的句子是“`Parameter`

'input' implicitly has an 'any' type." ——也就是說，宣告函式時，輸入的參數都會直接被認定為 **any** 型別。

這不難理解，因為我們呼叫該函式時可以有很多種呼叫方式：

```
/* 呼叫函式 echo */
echo(123); // ← 此時的輸入參數為數字型別
echo('Hello world!'); // ← 此時的輸入參數為字串型別
echo(true); // ← 此時的輸入參數為布林值型別
echo({ foo: 123, bar: 456 }); // ← 此時的輸入參數為物件型別
echo([1, 2, 3]); // ← 此時的輸入參數為陣列型別
echo(echo); // ← 此時的輸入參數為函式型別 (echo 函式物件本身)
```

既然輸入都可以這麼多樣化，理所當然地：「函式宣告時，參數無條件會被視為 **any** 型別的狀態」。

不過這裡要再給讀者另一個很重要的觀念：「有些時候回呼函式（Callback Function）⁹ 的參數，反而 TypeScript 編譯器是可以推論出該函式之輸入參數的型別的喔！」

再三強調前四個字：「有些時候」以及接下來四個字：「回呼函式」。

你可能覺得會很亂——作者一開始說函式的參數會無條件被視為 **any** 型別，現在又說有些情形是可以推論函式的參數型別。

不過看到本條目的讀者，請先記得這個觀念就好：大部分的狀況下，函式的參數沒辦法被推論的主因是由於「函式被呼叫時，參數可能被代入的值之型別有無窮多種狀況，因此 TypeScript 編譯器才會無條件判定輸入參數型別為 **any** 型別。」

9 回呼函式（Callback Functions）指的是將函式當成特殊物件，呼叫其他函式（或方法）時，將其傳遞到其他的函式（或方法）作為參數的手段，目的大多是為了控制或改變該函式內部程式執行的順序或時機點；例如常見的 `setTimeout` 函式，呼叫時第一個參數要求的就是一個函式物件，第二個參數代表延後的時間（微秒）執行第一個參數傳入的函式——其中，第一個函式就是屬於所謂的回呼函式。

等講到泛用型別篇章¹⁰ 時，作者就會講到函式參數可以被 TypeScript 編譯器推論出型別的情形喔！所以請讀者慢慢吸收知識，確保觀念正確後我們再進入進階的部分。

》》函式型別的推論與注意事項

函式物件通常分成兩個部分——輸入（Input）與輸出（Output）。

TypeScript 靜態分析函式物件之型別過程中，尤其是宣告函式時，大多數情形會無條件將函式之輸入參數推論為 `any` 型別；函式之輸出型別則是會看 `return` 敘述式回傳的結果值（或表達式）之型別。

若函式的 `return` 敘述式並無回傳任何值，亦或者該函式並無任何 `return` 敘述式，其輸出型別必為 `void`。

3.4.2 型別註記機制

有關於函式的註記語法，事實上已經在條目 2.2.1 講過了，不過這邊還是將幾種常見的方式稍微帶過一下：

```
/**  
 * 情況一：宣告變數時註記，並將函式當成物件值  
 * 使用箭頭 => 是要避免跟宣告變數時，早就在註記時用過的冒號混淆  
 */  
let addition1: (a: number, b: number) => number = function (a, b) {  
    return a + b;  
};  
  
/**  
 * 情況二：宣告變數時，並將函式當成物件值  
 * 並且在函式宣告輸入輸出時，一併註記  
 */  
let addition2 = function (a: number, b: number): number {  
    return a + b;  
};
```

10 泛用型別（Generics）請參見章節七。

```
/* 情況三：函式宣告敘述式，因此一定是在輸入輸出部分註記 */
function addition3(a: number, b: number): number {
    return a + b;
}

/** 
 * [ 較少用 ] 情況四：宣告變數時，將函式當成物件值
 * 此時由於函式物件值是屬於表達式，因此可以用斷言註記法（參見條目 2.2.2）
 */
let addition4 = function (a, b) {
    return a + b;
} as (a: number, b: number) => number; // ← 注意這裡是用箭頭
```

以上的範例中，所有的註記手法都有相同的功效，都會使得 `addition1` 到 `addition4` 皆被註記為函式型別 `(a: number, b: number) => number`；但以上的範例程式本身運作的差別不外呼跟原生 JavaScript 本來的機制有關，像是將函式當成物件值丟到變數與直接宣告函式等有一些些細微差別，但這並不在本書討論範疇。

大致上複習完函式到底有哪些方式註記時，本條目重點要放在：「什麼時機點或者是建議註記函式型別的情境」；而前個條目 3.4.1 的結論——也就是函式型別的推論運作機制，其實就已經大大地提示出使用註記的時機了：

由於函式宣告時，輸入會被無條件推論為 `any` 型別，因此大部分宣告函式時的情形，函式的輸入參數必須要進行註記的動作。

另外，前個條目還提到，函式的輸出型別通常可以借由 `return` 敘述式回傳的結果進行推論，因此理論上來說，輸出型別通常是不需要註記，除非輸出也是 `any` 型別亦或者是想要特別限制輸出型別，才會需要積極地註記輸出之型別以確保函函式在定義的過程中運作是正確的。

反正我們看幾個案例就知道囉～

首先將前面一系列宣告 `addition` 相關的函式範例拿出來用，先驗證第一、也是最重要的論述：「大多數函式宣告時，輸入型別必須要註記，輸出則不一定」。

```
/* 你一定得註記輸入型別，但不一定需要註記輸出型別 */
function addition(a: number, b: number) {
    return a + b;
}
```

以上的 `addition` 函式之宣告範例，作者把輸出型別的註記部分砍掉，但 TypeScript 依然可以將該函式之輸出型別推論出來。(如圖 3-35)

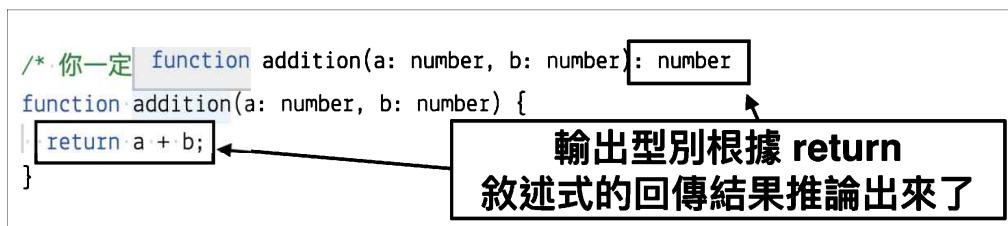


圖 3-35 輸出結果可以被輕易推論出來

當然作者並不是開玩笑的，就算你改變 `return` 敘述式可能回傳的型別結果，照樣還是可以推論出結果來的。(以下的範例程式碼，推論結果如圖 3-36)

```
/* 有了輸入型別的資訊，輸出型別可以根據 return 敘述式被輕易推論出來 */
function addition(a: number, b: number) {
    return (a + b).toString(); // ← 轉型成字串型別
}
```

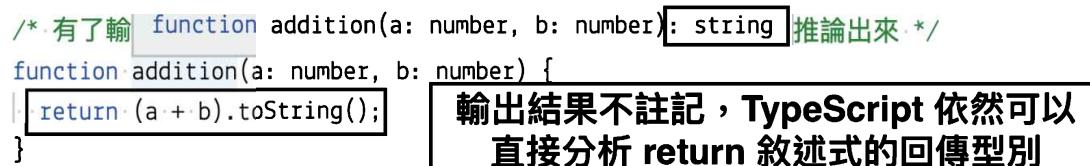


圖 3-36 換成轉型過後的結果依然可以推論出函式的輸出型別

不過呢，開發過程中註記宣告的函式之輸出型別是具有一定的好處的，假設我們想要開發某個函式 `isPositive`，專門檢測輸入的數字是不是正數（大於 0 的數字），剛開始宣告它並且註記時，你可能會這麼寫：

```
/* isPositive: 輸入一個數字，輸出布林值 */
function isPositive(input: number): boolean {
    // TODO: 實作此函式
}
```

目前的 `isPositive` 函式是還未實踐的狀態，但如果你有積極註記該函式宣告時的輸入與輸出型別時，你會看到圖 3-37 的警告訊息。

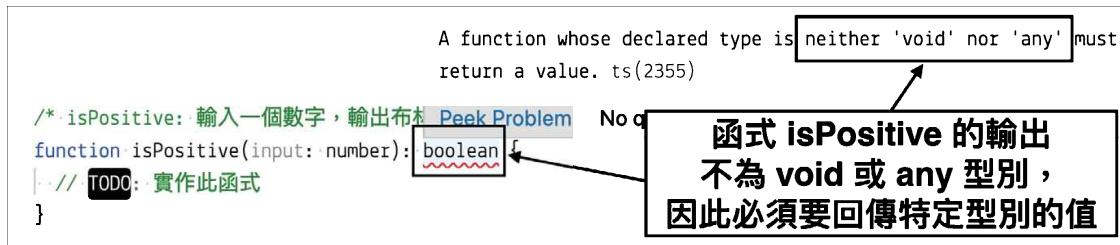


圖 3-37 函式若輸出型別非 `void` 或 `any`，則必須要回傳特定型別之值

圖中的錯誤訊息字可能有些小，作者就放在這邊：

```

A function whose declared type is neither 'void' nor 'any'
must return a value.
  
```

錯誤訊息有時候會流露一些重要的訊息，其中這裡除了出現了 `void` 以外，還多出了個 `any` 型別字樣——也就是說，`any` 型別也包含了 `void` 型別的案例；詳細的特殊型別的討論內容會在後續的條目探討¹¹，這些跟 `any` 型別有關的瑣碎細節，應該也讓讀者稍微摸清楚到底 `any` 型別的機制以及為何要儘量避免的原因了吧！

回過頭來，註記輸出型別部分，反過來說，有助於提醒我們在實踐該函式的過程中，必須將回傳的值必須要符合該函式被註記之輸出型別。

作者的習慣是開發時儘量會註記函式之輸出型別，而函式在實踐完成後，沒有太多功能擴充或改寫的需求¹²，且函式在取名上辨識性高的話，作者就會主動將該函式之輸出型別拔除，剩下讓 TypeScript 編譯器處理它該做的推論行為。

11 特殊型別請參見條目 4.6。

12 其實開發時，也會建議函式一旦被宣告並且實作出行為時，儘量不要更改內部的功能，因為根據開放 - 閉合定律（Open-Closed Principle）參考本書條目 9.2.2——程式寫作時不應該擅自更改內部原有的實作狀態，而是對外部以擴展方式實作出新的功能；唯一允許的內部行為就只有對內部的程式碼進行重構（Refactor）；按照開放 - 閉合定律下的開發模式，函式既然已經實作完成並且確保不會再更改的情形時，將輸出型別的註記拔掉讓 TypeScript 編譯器自行分析也就相對沒有太大的危險性。

什麼叫做辨識性高的函式？譬如說通常會回傳布林值（也就是 `boolean`）的函式取名方式可能為 `isXXX` 或 `hasXXX`，像是 `isPositive` 就代表檢測數值是不是正數、`hasItem` 代表檢測某個容器有沒有存放什麼內容等。這些函式既然可以從命名上，用人腦輕易辨識或直覺想到的型別結果，註記該函式輸出型別的必要性也就減少了。

»» 函式型別的註記時機

大部分的函式在宣告時，必須得註記輸入（input）部分的參數型別；而輸出（output）部分則不一定，因為可以經由 `return` 敘述式讓 TypeScript 判斷推論結果。

然而在開發過程當中，建議可以註記函式輸出結果的目的主要是確保實作過程是符合註記時預期的型別結果。

而宣告出的函式，若命名規則較直覺性等原因（用人腦從命名原則就可以輕易猜出函式輸出之型別），可以考慮拔除函式之註記型別的可能性，畢竟到最後 TypeScript 編譯器依然可以從輸出結果值（也就是從 `return` 敘述式）推論出函式的輸出型別。

3.4.3 選用參數 Optional Function Parameters

選用參數的寫法與規則跟條目 3.3.4 討論的 JSON 物件的選用屬性很相似，但意義上與使用限制上也有些不同。

首先，如果你想要宣告一個函式，其參數不一定需要用到時，你可以在宣告該參數時，旁邊也用 `?` 來註記。例如：

```
/* increment: 回傳累加的結果，若無輸入，則固定累加數字為 1 */
function increment(input1: number, input2?: number) {
    return input1 + (input2 ? input2 : 1);
}
```

其中，第一個參數 `input1` 非選用參數，因此在函式內必定是判定為數字型別，這邊就不附上圖了；而第二個參數在宣告時，參數 `input2` 後面有標上 `?`，代表 `input2` 為選用參數，而其判定型別如圖 3-38。

```
/* increment: 回傳累加的結果，若無輸入，則固定累加數字為 1 */
function increment(parameter) input2: number | undefined
  return input1 + (input2 ? input2 : 1);
}
```

**與選用屬性類似，選用參數會對該參數對應之型別
與 undefined 進行聯集複合**

圖 3-38 選用參數之型別會使得該參數註記的型別與 undefined 進行聯集複合

所以你在使用該函式時，以下兩種呼叫方式都是可以的喔～

```
/* 兩個參數都有提供 */
increment(123, 1);

/* 僅提供一個參數 */
increment(123);
```

選用參數當然可以使用多個，但更重要的是它的使用限制：它只能宣告在函式參數的尾端，而且不能斷斷續續地使用選用參數宣告。意思是說：

```
/* OK：可以宣告一個選用參數，並且是宣告在尾端 */
function sumFrom3Nums(input1: number, input2: number, input3?: number) {
  return input1 + input2 + (input3 ? input3 : 0);
}

/* OK：也可以宣告兩個選用參數，並且是連續性宣告在尾端 */
function sumFrom3Nums(input1: number, input2?: number, input3?: number) {
  return input1 + (input2 ? input2 : 0) + (input3 ? input3 : 0);
}

/* 禁止：選用參數沒有被宣告在尾端 */
function sumFrom3Nums(input1: number, input2?: number, input3: number) {
  return input1 + (input2 ? input2 : 0) + input3;
}

/* 禁止：選用參數斷斷續續宣告 */
function sumFrom3Nums(input1?: number, input2: number, input3?: number) {
  return (input1 ? input1 : 0) + input2 + (input3 ? input3 : 0);
}
```

以上的範例程式碼，前兩個案例是允許的行為；後兩者則是違反選用參數的使用規則（如圖 3-39 與 3-40）。

**如果有宣告選用參數時，
後面不能有任何必用參數**

(parameter) input3: number
A required parameter cannot follow an optional parameter. ts(1016)

```
/* 禁止：選用參數沒有被宣告在尾端 */    Peek Problem  No quick fixes available
function sumFrom3Nums(input1: number, input2?: number, input3: number) {
|· return input1 + (input2 ? input2 : 0) + input3;
}
```

圖 3-39 選用參數的宣告後方不能有任何必用參數（Required Parameter）

**必用參數夾在選用參數
的宣告中間也是不行的**

(parameter) input2: number
A required parameter cannot follow an optional parameter. ts(1016)

```
/* 禁止：選用參數斷斷續續宣告 */    Peek Problem  No quick fixes available
function sumFrom3Nums(input1?: number, input2: number, input3?: number) {
|· return (input1 ? input1 : 0) + input2 + (input3 ? input3 : 0);
}
```

圖 3-40 選用參數必定是要連續性的宣告，中間不能夾雜必用參數

其實觀察圖 3-39 與 3-40 的錯誤訊息：

A required parameter cannot follow an optional parameter.

代表的意思就是「必用參數（Required Parameter）的宣告不能放在選用參數宣告後方」，轉個邏輯思考就是作者講的：「選用參數必須連續性地被宣告在函式參數尾端」。

至於為何要設這項規定的簡單原因是，函式在呼叫時，假設中間的參數是可以忽略的，那以下的語法應該要可以被支援（不過想當然，JavaScript 是沒有支援此語法）：

```
sumFrom3Nums(1, /* 中間略過此參數 */, 3); // ← 這一定會產生錯誤
```

»» 函式宣告時使用選用參數 Optional Parameters

函式的輸入部分宣告時，可以宣告出選用參數，代表該函式被呼叫時，選用參數部分不一定需要代入值。

選用參數宣告時，只能連續性地宣告在函式參數尾端。

假設宣告某一函式 F ，擁有參數 P_1, P_2, \dots, P_N 且各自對應型別為 T_1, T_2, \dots, T_N ；以及選用參數 $P_{\text{option}1}, P_{\text{option}2}, \dots, P_{\text{option}N}$ 且各自對應型別為 $T_{\text{option}1}, T_{\text{option}2}, \dots, T_{\text{option}N}$ ，則寫法如下：

```
function F(  
    P1: T1, P2: T2, ..., PN: TN,  
    Poption1?: Toption1, Poption2?: Toption2, ..., PoptionN?: ToptionN  
) { /* 函式內容 */ };
```

3.4.4 預設參數 Default Function Parameters

如果讀者觀察前一個條目講到的第一個範例程式碼：

```
/* increment: 回傳累加的結果，若無輸入，則固定累加數字為 1 */  
function increment(input1: number, input2?: number) {  
    return input1 + (input2 ? input2 : 1);  
}
```

你會發現該程式碼必須處理選用參數 `input2` 中可能會出現 `undefined` 的情形——也就是使用者呼叫 `increment` 函式時省略第二個參數的情形。

本條目要討論的一種常見情況就是——選用參數的情形有時會搭配相對應的預設值（Default Value）。以上面的函式 `increment` 為例，參數 `input2` 的預設值是數字 `1`。

當然，另一種更常見的寫法是採用短路（Short-Circuiting）寫法來處理預設值：

```
function increment(input1: number, input2?: number) {  
    /* 若 input2 為 Falsy 相關的值，則指派數字 1 */  
    const value = input2 || 1;
```

```
    return input1 + value;
}
```

TypeScript 於 3.7 版推出了新的、也是更安全的空值連結操作符（Nullish Coalescing Operator）¹³ 的寫法：

```
function increment(input1: number, input2?: number) {
  /* 若 input2 為 null 或 undefined，則指派數字 1 */
  const value = input2 ?? 1;

  return input1 + value;
}
```

事實上，預設參數本來就不是 TypeScript 擁有的語法，而是 ECMAScript 標準下的規範，由於開發時時常遇到使用預設參數情形，因此收錄在本條目中。

將以上的範例函式 `increment` 可以改寫成下面的樣子：

```
function increment(input1: number, input2: number = 1) {
  return input1 + input2;
}
```

此時程式碼是不是也因此變乾淨了？欲呼叫上面的函式，由於 `input2` 已提供預設值，所以以下兩種呼叫方式都是可以的：

```
increment(123, 2); // ← 兩個參數都提供值
increment(123);    // ← 僅提供第一個參數的值，第二個參數就會使用預設值，也就是
                  數字 1
```

此外，如果比較「選用參數」與「預設參數」這兩種寫法，函式分別推論結果有很細微的差別。（選用參數範例推論結果如圖 3-41，預測參數則是如圖 3-42）

```
/* 選用參數寫法 */
function increment_1(input1: number, input2?: number) {
```

13 TypeScript 與 ECMAScript 相關章節參見章節十與十一。

```

    return input1 + (input2 || 1);
}

/* 預設參數寫法 */
function increment_2(input1: number, input2: number = 1) {
    return input1 + input2;
}

```

`/* 選用參` `function increment_1(input1: number, input2?: number | undefined): number`

`function increment_1(input1: number, input2?: number) {`

`|· return input1 + (input2 || 1);`

`}`

選用參數會與 undefined 聯集

圖 3-41 選用參數之型別會與 `undefined` 進行聯集複合的動作

`/* 預設參` `function increment_2(input1: number, input2?: number): number`

`function increment_2(input1: number, input2: number = 1) {`

`|· return input1 +`

`}`

預設參數把 undefined 情形鏟除掉了

圖 3-42 預設參數除了自動將參數標上選用記號 ? 外，也不會出現 `undefined` 的情形

另外，預設參數就沒有像是選用參數的限制，可以將預設參數設在函式的任何參數上，不一定要限制在尾端亦或者不連續設定預設參數，畢竟夾雜在中間的參數想要使用預設值時，可以直接填入 `undefined`，函式的參數就會被代入預設值。

不過作者也很少將函式中，帶有預設值的參數隨便亂設在跳來跳去或者是函式開頭的地方：

```

/* 預設參數寫法 */
function increment(input1: number = 1, input2: number, input3: number = 1) {
    return input1 + input2 + input3;
}

```

因為這根本不科學啊要是想呼叫此函式，但想用到前面參數的預設值時，你就得在帶有預設值的參數使用 `undefined`（不過這應該很醜吧）。

```
increment(undefined, 1);
```

至於有沒有同時標註參數為選用並且提供預設值的情形，這些都會在本章練習以及解答部分讓讀者自行去試驗看看喔～

》》函式宣告時提供預設值給參數 Default Function Parameter

1. 函式的輸入部分宣告時，可以宣告出預設參數值，代表該函式被呼叫時，預設參數部分不一定需要代入值。
2. 預設參數由於已提供預設值，因此在函式內該參數被使用時，並不會出現與 undefined 進行聯集複合的情形。
3. 預設參數宣告時，“建議”連續性地宣告在函式參數尾端——要是想要在呼叫函式時，某些參數使用預設值，則必須填入 undefined 。
4. 假設宣告某一函式 F，擁有參數 P₁、P₂、… P_N 且各自對應型別為 T₁、T₂、… T_N；以及選用參數 P_{default1}、P_{default2}、… P_{defaultN} 且各自對應型別為 T_{default1}、T_{default2}、… T_{defaultN}，預設值分別為 V_{default1}、V_{default2}、… V_{defaultN}，則寫法如下：

```
function F(  
    P1: T1, P2: T2, ..., PN: TN,  
    Pdefault1: Tdefault1 = Vdefault1, Pdefault2: Tdefault2 = Vdefault2, ..., PdefaultN: TdefaultN  
    = VdefaultN  
) { /* 函式內容 */ };
```

► 3.5 陣列型別 Array Object Type

3.5.1 型別推論機制

陣列（Array）在 JavaScript 是一個很常見的物件之一，而 TypeScript 也相對應提供了陣列型別的推論機制與註記語法。以下面最單純的例子來說：

```
/* 宣告費伯納契數列 Fibonacci Sequence */  
const fibSeq = [1, 1, 2, 3, 5, 8, 11];
```

推論結果如圖 3-43 所示。

```
/* 宣告 const fibSeq: number[] sequence */
const fibSeq = [1, 1, 2, 3, 5, 8, 11];
```

**陣列型別
附帶 [] 記號**

圖 3-43 附帶空的中括弧就是陣列型別的標誌記號

那麼如果說我們將陣列塞入亂七八糟的值，就會將出現的各種型別全部都聯集起來。

```
/* 亂七八糟的陣列 */
const arr = [123, 'Hello', false, { a: 123, b: 456 }, 1, [4, 5, 6]];
```

所以以上的陣列除了有數字、字串、布林值外，還有條目 3.3 講過的 JSON 物件型別以及本條目要討論的陣列型別。(圖 3-44)

```
const arr: (string | number | boolean | number[] | {
    a: number;
    b: number;
} | [ ])[]

/* 亂 */
const arr = [123, 'Hello', false, { a: 123, b: 456 }, 1, [4, 5, 6]];
```

全部種類的型別聯集起來的結果

圖 3-44 陣列的元素中有什麼型別，就跟該型別進行聯集

其中，最開始的範例程式碼，儲存費伯納契數列的陣列由於僅存放數字型別的值，通常只有存「單個型別」的陣列被稱為同質性陣列（Homogeneous Array）；相反地，如果陣列存放「多種不同型別」的值，我們稱它為異質性陣列（Heterogeneous Array）。

通常以同質性陣列出現在程式碼的頻率較多，因為異質性陣列大多情況下使用上是不方便的，取其元素時必須根據不同型別而有不同的處理方式。但如果講到第四章（條目 4.1）的元組（Tuple）時，此時 JavaScript 的陣列使用上的意義會變得不一樣。

另外，有時候我們會使用到多維陣列（Multi-dimensional Array）。簡單的案例如下，其推論結果如圖 3-45。

```
/* 二維陣列 */
const points = [    // ← 為 ( number[] )[] 型別，所以可以看成 number[][][]
  [1, 2, 3],        // ← 為 number[] 型別
  [4, 5, 6],        // ← 為 number[] 型別
  [7, 8, 9, 10]     // ← 為 number[] 型別
];
```

The diagram shows the TypeScript code for a 2D array. A callout box highlights the type annotation `const points: number[][]`. Another callout box contains the text "二維陣列就是中括弧串在一起". An arrow points from the first callout to the second.

```
/* 二維 */
const points: number[][] = [
  [1, 2, 3],           // ← 為 ( number[] )[] 型別，所以可以看成 number[][][]
  [4, 5, 6],           // ← 為 number[] 型別
  [7, 8, 9, 10]        // ← 為 number[] 型別
];
```

圖 3-45 二維陣列的推論結果

最後，陣列有沒有什麼需要注意的地方呢？那就是——空陣列。

```
/* 空陣列 */
const emptyArr = [];
```

空陣列由於沒有存任何型別的元素，所以 TypeScript 編譯器分析時，只能推論它可能會存任意型別的值，所以又是 `any` 型別出現的情境了。(如圖 3-46)

The diagram shows the TypeScript code for an empty array. A callout box highlights the type annotation `const emptyArr: any[]`. Another callout box contains the text "只有空陣列會出現 any[] 型別". An arrow points from the first callout to the second.

```
/* 空 */
const emptyArr: any[] = [];
```

圖 3-46 由於沒有元素型別可以參考，因此固定被推論為 `any[]`

3.5.2 型別註記機制

陣列的註記手法，其實光是看 3.5.1 講到的推論結果，就大概知道是如何被註記的。

```
/* 宣告費伯納契數列 Fibonacci Sequence */
const fibSeq: number[] = [1, 1, 2, 3, 5, 8, 11];
```

以上的程式碼就是典型的陣列型別註記的範例。

另外，註記的好處就是避免宣告時不小心填入錯誤的值，例如以下的範例程式碼。(出現的錯誤訊息如圖 3-47)

```
/* 宣告數字或字串型別的陣列 */
const arr: (number | string)[] = [123, 'Hello world', true, 'TypeScript'];
```

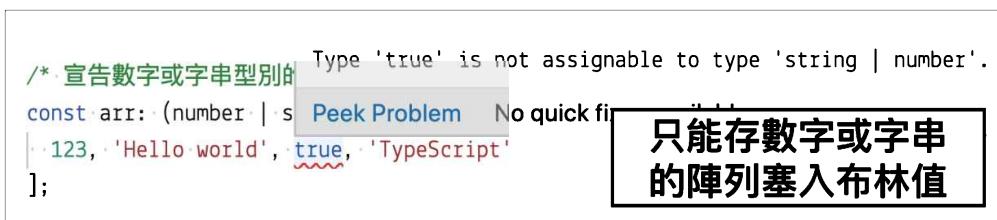


圖 3-47 TypeScript 分析過程會主動提醒開發者不能亂塞型別不符的元素

By the way，請讀者小心異質性陣列的註記方式：

```
/* 異質性陣列註記時，必須要用小括弧將聯集型別部分匡起來 */
const ex1: (number | string)[] = [/* ... */];

/* 以下的註記方式意思是：「數字型別」或者是「僅存字串型別的陣列」 */
const ex2: number | string[] = [/* ... */]; // ← 或者可以放純數字
```

此外，條目 3.5.1 提到空陣列推論結果必定為 `any[]`，因此為了避免此情形，我們必須積極註記空陣列，這是沒辦法避免的事情。

```
/* 空陣列必須主動註記它 */
const emptyArr: number[] = [];
```

陣列型別可以玩的花樣很多請看本章習題你就知道了，不過重點事實上並不

》》 陣列型別的推論與註記之注意事項

陣列型別註記時，後面會尾隨中括弧符號 `[]`，假設宣告陣列儲存型別為 `T` (`T` 可能代表單一型別或複合過後的型別)，寫法為：

```
let <variable>: T[] = [ /* 元素皆為型別 T */ ];
```

其中，空陣列由於沒有任何值可以參考，因此推論型別必為 `any[]`；為了避免 `any` 型別的出現，主動註記空陣列是必要的行為。

► 3.6 明文型別 Literal Type

3.6.1 型別推論機制

明文（Literal）—— 顧名思義就是直接將值表現出來。（條目 2.3.3 的開宗明義）

所以你在程式中看到的，任何值皆可自成一種型別。

```
/* 名為數字 123 的明文數字，但也可以自成一種型別 */
123;
```

```
/* 名為字串 'Maxwell' 的明文字串，但也可以自成一種型別 */
'Maxwell';
```

首先如果遇到指派隸屬於原始資料型態（條目 2.3.1）的明文到變數中，分成兩種情形：如果直接將明文值指派到使用 `let` 關鍵字宣告的變數裡，通常該變數不是明文值型別，而是以該明文本身代表的型別而定；相對地，使用 `const` 關鍵字宣告的變數（又名常數，Constant）普遍來說會直接被推論為明文值型別，畢竟常數的值不能改。（這很像在繞口令）

```
/* 使用 let 告知變數並指派原始型別值，被推論的並不是明文型別，而是明文代表的型別 */
let num = 123;           // ← 推論結果為 number
```

```
/* 使用 const 告知變數並指派原始型別值，被推論的是明文型別 */
const constNum = 123; // ← 推論結果為 123
```

```
/* 使用 let 告知變數並指派原始型別值，被推論的並不是明文型別，而是明文代表的型別 */
let num = 123;           // ← 推論結果為 number
```

圖 3-48 使用 `let` 關鍵字宣告的變數，推論的不是明文本身，而是明文本身的型別

```
/* 使用 const 告知變數並指派原始型別值，被推論的是明文型別 */
const constNum = 123; // ← 推論結果為 123
```

明文表示之型別

圖 3-49 使用 `const` 關鍵字宣告的常數，推論的是明文本身

但是如果是遇到指派物件的明文到變數時，不管是用 `let` 或者是 `const` 關鍵字，都只會描述該物件的結構（JSON 物件、陣列與函式物件皆是），而非物件本身的明文值。

```
/**  
 * 不管是使用 let 或 const 告訴變數並指派物件值，被推論都是明文的 "結構"；  
 * 以下推論結果皆為 { foo: number; bar: string }  
 */  
let jsonObj = { foo: 123, bar: 'Hello' };  
const constJsonObj = { foo: 123, bar: 'Hello' };
```



圖 3-50 儘管看似是明文的 JSON 物件，但仔細一看是 JSON 物件型別

其實這不難理解，畢竟物件的傳遞本身是靠參照（Reference）傳遞，因此表面上物件被宣告為常數，但實際上物件依舊可以更改屬性對應的值，因此 `constJsonObj` 之屬性 `foo` 對應的並非數字 123 的明文型別，而是 `number`；同理，屬性 `bar` 對應的並非字串 'Hello' 的明文型別。

```
constJsonObj.foo = 456; // ← 儘管物件是被宣告為常數，但仍然可更改屬性對應之值
```

這時可能就有讀者想說，有一種方式覺得可以防止物件被竄改，就是使用 `Object.freeze` 來凍結物件被竄改的權限。而且用過這個方法會產出有趣的推論結果（如圖 3-51）：

```
/* 實質意義上的常數物件，完全被凍結的狀態 */  
const freezedJsonObj = Object.freeze({ foo: 123, bar: 'Hello' });
```



圖 3-51 推論結果為，將 JSON 物件型別變成唯讀狀態的型別

把物件凍結，其推論結果就只是把型別部分冠上唯讀狀態的註記，這早在條目 3.3.5 提及。理所當然地，以下的行為就會被 TypeScript 靜態分析時丟出錯誤。

```
freezedJsonObj.foo = 123;  
// → 錯誤訊息：Cannot assign to 'foo' because it is a read-only property
```

3.6.2 型別註記機制

明文型別的註記方式其實有時候覺得挺好笑的：

```
let devilNumber: 666 = 666;
```

其實以上數字指派的範例，其實要這樣寫是可以的，但是光是從條目 3.6.1 的結論就可以知道，它可以直接簡寫為：

```
const devilNumber = 666;
```

畢竟原始型別值作為常數指派的值，該宣告之常數推論結果就等於原始型別值。(又在繞口令)

然而，如果是物件的話，物件的明文值若是這樣指派下去：

```
let jsonObj: { foo: 123, bar: 'Hello' } = { foo: 123, bar: 'Hello' };
```

其實也就等效於前一個條目講到的，將物件凍結的效果是差不多的，畢竟也根本無法更改或重新指派新的值到 jsonObj：

```
jsonObj.foo = 123; // ← 這是可行的，因為符合屬性 foo 對應數值 123  
jsonObj.foo = 456; // ← 這是不行的
```

所以第一個結論很簡單：

明文型別值基本上不需要註記，直接宣告成常數就好了。

如果遇到 JSON 物件或其他種類物件，也可以使用 Object.freeze 將其凍結。

但是明文值並不是說沒有在註記的，有一種情形是會使用明文值，但會搭配聯集複合的技巧，例如我們可以特別宣告一個名為 **Primitives** 的化名，將原始型別的字串表示法放進去：

```
type Primitives = 'number' | 'string' | 'boolean' | 'null' | 'undefined';
```

這樣可以將型別從所有字串類型的值，限縮到特定的字串值的集合。此外，最明顯的另一個範例是 JavaScript 本身就有的 `typeof` 操作符會出現的推論結果（以下的範例程式碼推論結果如圖 3-52）。

```
let something = 123;
let typeofSomething = typeof something;
```

```
let something: number = 123;
let typeofSomething: string = "number";
```

明文型別聯集起來的結果

圖 3-52 全部都是各種不同字串值聯集之結果

»» 明文型別的推論與註記之注意事項

1. 每一個明文值都可以自成一個型別。
2. 通常只要將原始型別值宣告成常數（Constant），該常數的推論結果就是原始型別的明文值。
3. 物件明文值雖然也可以宣告成常數，但由於傳遞的方式是以參照（Reference）傳遞，因此內部的屬性對應的值還是有被竄改的可能，因此推論結果會是物件型別而非物件的明文型別。
4. 若想要實實在在地封住物件被竄改的權限，使用 `Object.freeze` 是一種方式。
5. 明文型別除非搭配聯集複合（Union）的方式，否則是相對沒必要註記的。

3.6.3 互斥聯集 Discriminated Unions

事實上，我們可以更有技巧性地使用 JSON 物件，搭配明文型別與聯集複合技巧建構出一個很有系統性的聯集系統，此技巧又被稱之為互斥聯集¹⁴。

¹⁴ 互斥聯集可以參見 TypeScript 官方 <https://www.typescriptlang.org/docs/handbook/advanced-types.html#discriminated-unions>。

作者想要以 Facebook 推行的 FLUX 架構為範例，相信有寫過 ReactJS 的人就會知道，Redux 就是根據此架構產出的套件——目的除了是統一管理資料的狀態（State）外，也會限制對於資料更新部分可以採取的動作（Actions）。

但請沒有使用過 ReactJS 的讀者們放心，因為 FLUX 架構事實上不一定得學過 ReactJS 才能夠學到的東西，你可以把 FLUX 架構當成只是一種藍圖或演算法，可以獨立出來學習。以下就用簡單的 To Do List 展示給讀者看。

首先 FLUX 架構第一個重點是：統一管理資料狀態，所以第一個主角是狀態，也就是會常常聽到的 State，所以 To Do List 的狀態可能會用很簡單的 JSON 物件表示：

```
let state = {
  items: [
    { id: 1, title: 'Have Lunch', done: true },
    { id: 2, title: 'Learn TypeScript', done: false },
  ],
  trackID: 3, // ← 隨時紀錄下一個 item 被新增時需要指派的 ID 值
};
```

通常在沒有限制資料是如何更新的時候，你可能會寫一大堆亂七八糟的函式，比如：

```
function newItem(title: string) {
  state = {
    items: [
      ...state.items,
      {
        title,
        done: false,
        id: state.trackID,
      }
    ],
    trackID: state.trackID + 1,
  };
}

function removeItem(id: number) {
```

```
// 將某事項給刪除
}

function completeItem(id: number) {
    // 將某事項的 done 設為 true
}

function undoneItem(id: number) {
    // 將某事項的 done 設為 false
}

// 可能有更多方法可以更新狀態 ...
```

請讀者不要誤會一件事情，任何程式都一定有它好或者是不好的地方。以上的程式碼確實是將更新狀態的各種不同動作都包裝在函式裡，這樣在使用上依然還是保有某種程度的可讀性：

```
// 新建一個待辦事項
 newItem('Learn Rust Lang.');

// 刪除 id = 2 的待辦事項
 removeItem(2);

// 將 id = 3 的待辦事項的完成狀態設為 true
 completeItem(3);

// 將 id = 1 的待辦事項的完成狀態設為 false
 undoneItem(1);

// ... 更多不同的更新資料狀態方式
```

然而，這裡依然有些美中不足的地方，如果把每個函式當成一個個更新資料狀態的 API 接口，那這樣多個接口混用的狀態下，讀程式碼可能會比較痛苦些。

有沒有辦法可以統合成單一個接口並且仍然能夠乾淨地將狀態更新？

也就是有沒有辦法寫成類似這樣的樣式：

```
// 新建一個待辦事項
 dispatch({ type: 'NEW_ITEM', payload: { title: 'Learn Rust Lang.' }});
```

```
// 刪除 id = 2 的待辦事項  
dispatch({ type: 'REMOVE_ITEM', payload: { id: 2 } });  
  
// 將 id = 3 的待辦事項的完成狀態設為 true  
dispatch({ type: 'COMPLETE_ITEM', payload: { id: 3 } });  
  
// 將 id = 1 的待辦事項的完成狀態設為 true  
dispatch({ type: 'UNDONE_ITEM', payload: { id: 1 } });
```

儘管說字感覺打得比較多了¹⁵，但你會發現以上的程式碼，論可讀性來說會比較高，除了統一都使用 `dispatch` 函式作為接口外（後面會提到取名的意義），比較以下這兩種寫法：

```
// 將待辦事項的完成狀態設為 true  
completeItem(3);  
  
// 將 "id = 3" 的待辦事項的完成狀態設為 true  
dispatch({ type: 'COMPLETE_ITEM', payload: { id: 3 } });
```

後者比起前者的寫法多了一些資訊，前者我們可能用猜它是將 ID 為 3 的事項完成，但如果在容易錯意的狀態下，有沒有可能會以為是「完成三個代辦事項」的意思？

後者的寫法就可以澄清上述的情形，因為意思若是指「完成三個代辦事項」，那寫法應該會類似：

```
// 將 "三個" 待辦事項的完成狀態設為 true  
dispatch({ type: 'COMPLETE_ITEM', payload: { count: 3 } });
```

解釋完以上的寫法的好處後，接下來要介紹 FLUX 的其他部分。另外，讀者可能以為作者忘記要講本條目的互斥聯集的應用了，以下就會正式應用到此技巧。

FLUX 最主要的目的除了剛剛講到的統一管理資料狀態外，還有限制我們可以採取的資料更新行為。如果以上面的案例來看，目前只能夠有四種方式去更新

15 VSCode 提供的自動補齊（Autocomplete）早就解決這方面的問題，參考本書條目 1.5。

To Do List，也就是新建、刪除代辦事項，以及將代辦事項標註為完成或者是未完成的狀態。

所以我們可以宣告四種不同的行為（Actions），用型別化名的方式宣告：

```
// 新增代辦事項
type NewItemAction = {
  type: 'NEW_ITEM';
  payload: { title: string };
};

// 刪除代辦事項
type RemoveItemAction = {
  type: 'REMOVE_ITEM';
  payload: { id: number };
};

// 完成代辦事項
type CompleteItemAction = {
  type: 'COMPLETE_ITEM';
  payload: { id: number };
};

// 將代辦事項重設為未完成狀態
type UndoneItemAction = {
  type: 'UNDONE_ITEM';
  payload: { id: number };
};
```

從以上的四種不同型別化名，結構都很相像，都是擁有屬性 `type` 以及 `payload`；有些讀者可能不清楚 `Payload` 這個詞是代表什麼意思，直翻成中文的話是「負載」的意思，但這邊更精確一點是提供額外的參數化資訊。

不過想要從這四種型別辨識出其中的任一種型別的話，可以用屬性 `type` 作為辨識基準（畢竟是用明文字串的方式表示），而這些儘管相似但是可以從單一屬性互相排除其他型別的可能性，一但它們聯集在一起就成了一種互斥聯集；所以我們使用一個名為 `Action` 的東西對這四種不同的型別進行聯集：

```
/**
 * Action 為四種不同的型別組成的聯集
 * 而四種不同的型別可以藉由單一屬性，也就是 type，來排除掉為其他型別的可能性
 */
type Action = (
    NewItemAction | 
    RemoveItemAction | 
    CompleteItemAction | 
    UndoneItemAction
);
```

互斥聯集的好處有很多種，首先如果你明確註記某變數為互斥聯集型別，並且程式寫到某個段落會出現提示視窗（如圖 3-53）：

```
const anUnknownAction: Action = {
    type: '', // ← 這裡會出現聯集過後的明文字串提示內容
};
```



圖 3-53 互斥聯集會出現的提示性視窗，告訴你該型別可能有哪幾種情況

而當你確定互斥聯集中的其中一種型別的話，後續就會出現該型別化名的規格方面的提示，圖 3-54 為當屬性 `type` 為 '`NEW_ITEM`' 以及圖 3-55 則是屬性 `type` 為 '`REMOVE_ITEM`' 的兩種情境下，屬性 `payload` 出現的提示訊息。

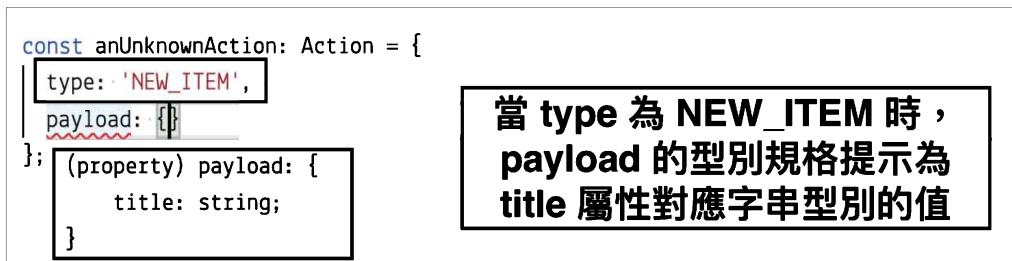


圖 3-54 屬性 `type` 為 '`NEW_ITEM`' 對應之型別提示

```
const anUnknownAction: Action = {
  type: 'REMOVE_ITEM',
  payload: {}  

};  
  (property) payload: {  
    id: number;  
}
```

切換成 `type` 為另一種互斥的型別，以 `REMOVE_ITEM` 為例，`payload` 被改成提示為需要屬性 `id` 對應數字型別值

圖 3-55 屬性 `type` 為 '`REMOVE_ITEM`' 對應之型別提示

另外，儘管我們積極註記變數 `anUnknownAction` 為整個互斥聯集，也就是 `Action` 型別，但是如果依照型別本身互斥的特性，以下面的程式碼為例：

```
const anUnknownAction: Action = {
  type: 'NEW_ITEM', // ← 鎖定為 NewItemAction 的型別
  payload: { title: 'Learn TypeScript' }
};

anUnknownAction; // ← 推論結果會是 NewItemAction 而非 Action
```

變數 `anUnknownAction` 的推論結果會是 `NewItemAction`。(如圖 3-56)

```
const anUnknownAction: Action =  
  type: 'NEW_ITEM',  
  payload: { title: 'Learn TypeScript' }  
.  
const anUnknownAction: NewItemAction  
anUnknownAction;
```

互斥聯集被限縮為
單一特定型別

圖 3-56 變數 `anUnknownAction` 的推論結果

藉由此特性，我們可以設計出統一更新資料的函式 `dispatch`，其結構大致上如下：

```
/* 簡易版本的 dispatch 函式 */
function dispatch(action: Action) {
  switch(action.type) {
    case 'NEW_ITEM':
      newItem(action.payload.title);
      break;
    case 'REMOVE_ITEM':
      removeItem(action.payload.id);
      break;
```

```
case 'COMPLETE_ITEM':  
    completeItem(action.payload.id);  
    break;  
case 'UNDONE_ITEM':  
    undoneItem(action.payload.id);  
    break;  
}  
}
```

當然，以上的 `dispatch` 函式一定有它可以改進的地方，最主要是要能夠學習 TypeScript 的互斥聯集的技巧。

最後，稍微補充一下 FLUX 架構的概念。

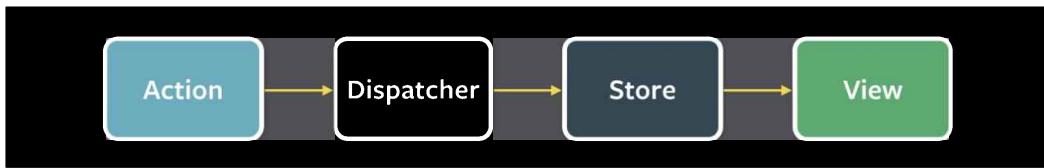


圖 3-57 簡易的 FLUX 架構示意圖¹⁶

FLUX 主要是描述資料管理與更新的流程，其中前面講到的資料狀態 (State)，更精確地來說是指存放資料狀態的地方，也就是 Store。

Store 不可以隨隨便便改變它的狀態，嚴格來說是要用完整覆寫的方式去更新整個 Store，所以本書前面的案例其實是錯誤的行為，僅僅只是示範模擬而已。

另外，想要更新 Store 的狀態必須要宣告不同的行動方式，也就是 Action。這就是為何前面每一個互斥型別的命名都以 Action 相關為主，而每個 Action 會有它的類型 (Type) 與額外被稱之為 Payload 的參數。

最後，每一次要發動一個 Action 更新 Store 時，必須要統一用 Dispatcher (調度者) 來發動更新的行為，而 Dispatcher 其實單純就只是函式而已。(太過 Fancy 的名稱有時候會讓人誤以為是很深奧的技術，但大多數並不是)

16 圖片出自 30-Days-of-React <https://github.com/fullstackreact/30-days-of-react/blob/master/day-18/post.md>。

► 本章練習

- 請讀者試著驗證看看，以下的程式碼，變數 `whatIsThis` 的最後推論結果為何？

```
let whatIsThis;
let randomNumber = Math.random();

if (randomNumber < 0.33) {
    whatIsThis = 123;
} else if (randomNumber < 0.66) {
    whatIsThis = 'string';
} else {
    whatIsThis = true;
}

whatIsThis; // ← 推論結果為何？
```

若假設更換成 `switch...case...` 判斷敘述式，如下的案例，則變數 `whatIsThis` 的最後推論結果為何？

```
let whatIsThis;
let randomNumber = Math.random();

switch (true) {
    case randomNumber < 0.33: whatIsThis = 123;      break;
    case randomNumber < 0.66: whatIsThis = 'string'; break;
    default:                  whatIsThis = true;
}

whatIsThis; // ← 推論結果為何？
```

- 請讀者試著驗證看看，以下的程式碼，變數 `whatIsThis` 的最後推論結果為何？

```
let whatIsThis;

if (true) {
    whatIsThis = 123;
```

```
} else {
    whatIsThis = '123';
}

whatIsThis; // ← 推論結果為何？
```

如果假設把 `if (true)` 的條件換成其他式子，比如 `if (1 === 1)`，變數 `whatIsThis` 的最後推論結果為何？

3. 變數 `variable` 若採取以下的方式宣告，則該變數此時的值與推論出來的型別為何？

```
let variable;
```

4. 以下 TypeScript 的程式碼，有沒有可能出現警告訊息？是什麼原因造成的？

```
let variable: number;

if (Math.random() > 0.5) {
    variable = 123;
}

console.log(variable);
```

5. 如果今天以關鍵字 `let` 宣告某些物件型別的值，例如：

```
let jsonObj = { foo: 123, bar: 'Hello' };
let funcObj = function (a: number, b: number) { return a + b; };
let arrayObj = [1, 2, 3, 4, 5];
```

請問以上三種物件，只要型別結構上是正確的話，可以覆寫任何一個物件型別值嗎？

6. 請解釋以下宣告型別 A 的結構，每個屬性的寫法對應型別的差異性在哪？

```
type A = {
    prop1: string;
    prop2: string | undefined;
    prop3?: string;
```

```
prop4?: string | undefined;
prop5: string | null;
prop6?: string | null;
};
```

7. 宣告 JSON 物件型別時，若同時在某個屬性加上選用屬性與唯讀屬性操作符，這樣的效果如何？有任何操作上的意義嗎？
8. 宣告函式型別中，輸出型別可以根據什麼樣的基準推論出來？至於輸入型別之所以不能無法被推論的主要原因為何？
9. 實驗看看以下的函式的推論結果為何？由此可知，函式輸出型別的判斷基準會因為程式分岔點使得某些地方並不會被執行到而變嗎？

```
function what_is_the_inference_result() {
    if (true) { return 42; }

    /* 以下這一行絕對不會被執行 */
    return '42';
};
```

根據以上的實驗結論，請預測下面的函式之推論結果為何？

```
function what_is_the_inference_result() {
    return true;

    /* 以下的程式都不會被執行到 */
    if (true) { return 42; }

    return '42';
};
```

10. 若遇到函式型別之 `return` 敘述式接的是 `null` 或 `undefined`，請問該函式之輸出型別推論結果為何？
11. 請問以下的函式會推論出什麼樣的結果？

```
function returns_itself_if_positive(input: number) {
    if (input > 0) return input;
}
```

12. 請問以下函式宣告的情形，你會如何簡化它？

```
function greet(message: string | undefined, name: string | undefined) {  
    const msg = message === undefined ? 'Hello' : message;  
  
    if (name === undefined) {  
        console.log(`#${msg}!`);  
        return;  
    }  
    console.log(`#${msg}, ${name}!`);  
}
```

13. 能不能出現同時使用選用參數以及提供預設值的情形？譬如：

```
function increment(input: number, value?: number = 1) {  
    return input + value;  
}
```

14.【進階題】請問以下各種變數推論結果為何？（本題就算有些不會其實也沒差）

```
let ex1 = [ [1, 2, 3], [4, 5, 6] ];  
let ex2 = [ 1, 2, 3, [4, 5, 6] ];  
let ex3 = [ [1, 2, 3], [4, 5, 6], [] ]; // ← 多一個空陣列  
let ex4 = [ [], [], [] ]; // ← 此題答案連作者也意想不到  
  
let ex5 = [  
    { foo: 123, bar: 'Hello' },  
    { foo: 456, bar: 'World' },  
];  
  
let ex6 = [1, , 3, , 4]; // ← 疏鬆陣列 Sparse Array  
let ex7 = [ , , , , ]; // ← 完全疏鬆陣列
```

15. 根據條目 3.2.1 提到的型別推論機制的特點——固定性，其解釋如下：

一經推論過後的變數，並且該變數宣告時是使用 let 關鍵字，後續任何指派到該變數的值必須符合該變數被推論之型別的範疇。

請問 JSON 物件、函式、陣列等是否也符合這一項特性？

```
/* 宣告型別為 { foo: number; bar: string } 的 JSON 物件 */
let jsonObj = { foo: 123, bar: 'Hello' };

/* 是不是可以被同樣格式的 JSON 物件覆寫？ */
jsonObj = { bar: 'World', foo: 456 };

// 同理，函式或陣列是不是也可以按照上面的方式，只要型別符合
// 使用 let 關鍵字宣告的變數也可以被覆寫？
```

16. 請問以下的變數推論結果各為何？

```
let ex1 = 123;
const ex2 = 123;
let ex3 = { foo: 123, bar: 'Hello' };
const ex4 = { foo: 123, bar: 'Hello' };
```

17. 請問以下的兩種寫法，效果會有差嗎？

```
const withoutAnnotation = { hello: 'world' };
const withAnnotation: { hello: 'world' } = { hello: 'world' };
```

18. 假設宣告幾個跟幾何形狀（Geometry）相關的型別化名如下：

```
type Rectangle = { // ← 長方形的面積公式 = width * height
  width: number;
  height: number;
};

type Triangle = { // ← 三角形的面積公式 = base * height / 2
  base: number;
  height: number;
};

type Circle = { // ← 圓形的面積公式為 = Math.PI * radius * radius
  radius: number;
};
```

其中，將這三種不同的型別使用聯集的方式複合起來，宣告出新的型別化名：

```
type Geometry = Rectangle | Triangle | Circle;
```

請設計出函式 `area`，輸入型別為 `Geometry` 的資料（不能為其他型別的輸入），回傳結果為輸入的幾何圖形資料的面積（輸出型別為 `number`）：

```
function area(geometry: Geometry): number {  
    // 實作功能  
}
```

此外，允許額外新增屬性規格到型別化名 `Rectangle`、`Triangle` 以及 `Circle` 上。

所有的型別化名原有的屬性規格不能夠被改變或刪除。

提示：

```
function areaOfRectangle(rect: Rectangle) { // Rectangle 面積算法  
    return rect.width * rect.height;  
}  
  
function areaOfTriangle(tri: Triangle) { // Triangle 面積算法  
    return tri.base * tri.height / 2;  
}  
  
function areaOfCircle(cir: Circle) { // Circle 面積算法  
    return Math.PI * cir.radius * cir.radius;  
}
```

04

深入型別系統 II 進階篇

接下來要進入到型別系統的後半部分，也就是跳脫原生 JavaScript 還可以看得到的型別與語法等。本篇章大部分都是探討 TypeScript 本身提供的語言特色或功能。

► 4.1 元組型別 Tuple Type

4.1.1 元組的意義

元組（Tuple）的定義為：

元組為擁有固定個數、固定的型別順序的元素組合。

比如說，你可以把元組想成在類似表格（Table）中的每一列（Row）資料：

```
const table = [
  ['Maxwell', true, 18],
  ['Martin', false, 23],
  // ... 更多資料，但格式都會是：[ string, boolean, number ]
];
```

儘管它跟陣列（Array）¹有些相似，但這兩種資料型態的意義差很多。陣列的

1 陣列型別，請參考本書條目 3.4。

特點跟元組完全相反：

陣列為可以擁有任何個數、型別的元素組合。

只是光是靠 JavaScript 本身提供的陣列寫法亦或者是 TypeScript 的陣列型別，不太可能可以模擬出元組這種型別。畢竟我們可以簡簡單單地破壞掉剛剛的表格結構：

```
const table = [
  ['Maxwell', true, 18, { foo: 123 }], // ← 多一個元素
  [23, 'Martin', false], // ← 順序變亂
  // ... 更多資料，但格式已不再是：[ string, boolean, number ]
];
```

4.1.2 型別推論機制

元組型別不可能被 TypeScript 分析時推論出來。從前一個條目的範例：

```
const table = [
  ['Maxwell', true, 18],
  ['Martin', false, 23],
  // ... 更多資料，但格式都會是：[ string, boolean, number ]
];
```

TypeScript 會將它推論為二維陣列，最內層為 `string | boolean | number` 這種聯集過後的型別。(如圖 4-1)

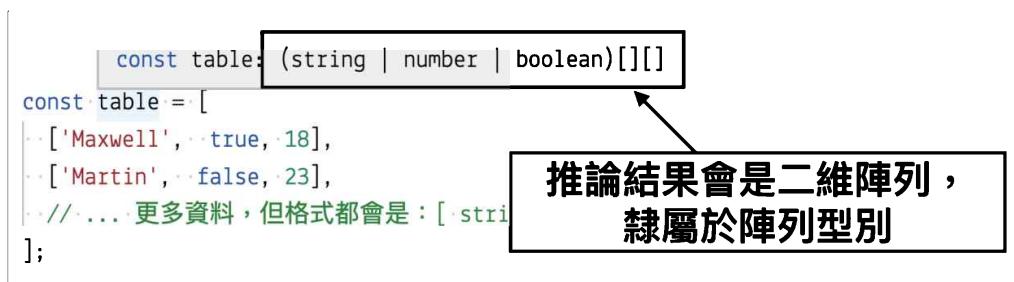


圖 4-1 推論結果絕對會是陣列型別

只要遇到陣列格式當然預設就會以陣列型別為主，因此遇到元組型別的使用情境時，註記是必要的。

4.1.3 型別註記機制

元組型別的註記方式與陣列差異在於，陣列的註記會是將空的中括弧放在元素的型別後方，例如某陣列存取數字或字串型別時：

```
const arr: (number | string)[] = [1, '2', 3, '4', 5];
```

然而，元組由於有個數限制外，也有型別順序的限制，因此元組型別的註記方式會在中括弧裡放置元素型別，並且順序很重要：

```
const tup: [number, string] = [1, 'Hello world'];
```

你如果這樣寫那麼就錯了，因為它已經不是陣列，它就不能夠丟個數不符合的元素。(圖 4-2 為錯誤訊息)

```
const tup: [number, string] = [1, 'Hello world', 123];
```

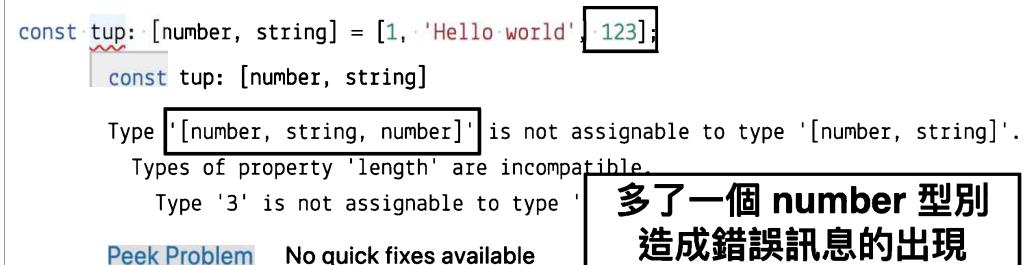


圖 4-2 多了一個數字造成元組型別衝突

當然，順序不對也有差。(圖 4-3)

```
const tup: [number, string] = ['Hello world', 1];
```

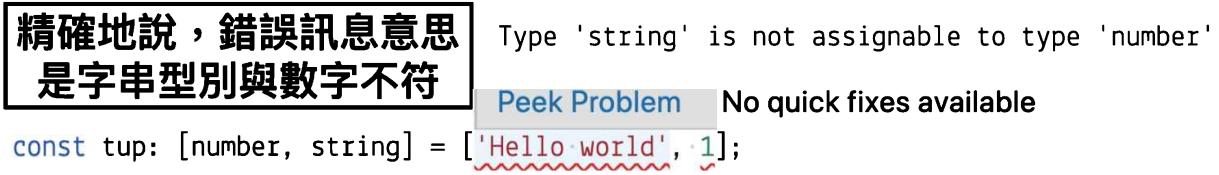


圖 4-3 型別順序不對造成的錯誤

其實如果你根據錯誤訊息來看，它比較的是型別單純有沒有符合在元組內的相對位置對照的元素型別。意思是說，元組內第一個元素若宣告為數字型別，而

值的第一個元素非數字型別就會發出錯誤訊息。

基本上，元組的重點沒有太多需要注意的地方，事實上用的情況也比較少了些。可能從外在類似 CSV 或 Excel 檔案之類的表格資料結構匯出結果變成陣列時，才會有使用到元組的情境。畢竟元組內存的元素若是同一種型別的話，光靠值很難判定內部的資料順序有沒有可能順序錯誤的情形。

另一種常見的使用情境是人類認知上習慣的結構，例如平面座標系統，我們就可以宣告一個名為 `Coordinate` 的化名，第一個元素固定為經度、第二個則是緯度：

```
type Coordinate = [number, number];
```

亦或者是數學裡常見的矩陣（Matrix）系統，如 `Matrix3By3`：

```
type Matrix3By3 = [
  [number, number, number],
  [number, number, number],
  [number, number, number],
];
```

地址（Address）有可能也可以用元組表示，只是就比較麻煩些：

```
type Address = [number, number, string, string, number, string];
// 如：[123, 456, 'Heaven', 'Hell', 12345, 'World']
// No. 123, Rd. 456, Heaven Rd., Hell City 12345, World 之類的英文住址
```

»》元組型別的意義與使用

1. 元組（Tuple）的特點在於它有元素個數限制以及固定的對應型別順序（陣列則相反）
2. 元組無法經由推論而來，畢竟是用 JavaScript 陣列資料型態表示，理應會按照條目 3.4 敘述推論成陣列型別
3. 元組的註記方式為在中括弧裡放置元素之型別，且須注意順序。若宣告某元組型別 `Ttuple`，且內含的元素對應之型別為 $T_1, T_2 \dots, T_N$ ，則寫法如下：

```
type Ttuple = [T1, T2, ..., TN];
```

► 4.2 列舉型別 Enum Type

4.2.1 列舉的意義

另一個 TypeScript 裡常見但並非原生 JavaScript 有的型別就是列舉（Enum）。它可以將性質類似的成員（Member），用物件的鍵（Key）的方式匯聚起來形成的型別。

不知道讀者有沒有見過某些程式在宣告常數時，寫法可能如下：

```
const Colors = {    // ← 告知顏色 Color 的鍵值對
  Red: 'Red',
  Blue: 'Blue',
  Yellow: 'Yellow',
};
```

特別不用純字串值，而是用物件的鍵來代表特別的常數，這樣可以使其在程式裡的可讀性增高，以區別普通的值跟特別的值之差異。這種就是列舉的一種寫法，列舉出相似性質的東西。

而 TypeScript 有提供類似的寫法，而宣告列舉的同時，等同於宣告新的型別化名（就是列舉型別的名稱本身）：

```
/* Colors 既是列舉型別，其名稱本身就是型別化名 */
enum Colors { Red, Blue, Yellow };
```

列舉 Colors 裡面的 Red、Blue 與 Yellow 就是所謂列舉型別裡面的列舉成員²。

這裡要注意的事情是：列舉宣告時，中間不需要夾等號，完全與 C# 的列舉宣告方式相同³。

2 列舉型別（Enum Type）與列舉成員型別（Enum Member Type）是不一樣的東西，請參見 4.2 後續條目。

3 請參考本書條目 1.1。

4.2.2 型別推論機制

首先，列舉可以當成正常的 JSON 物件方式，將宣告的列舉內含的元素當成物件的鍵呼叫，以條目 4.2.1 宣告過後的列舉型別 `Colors` 為例：

```
/* 把列舉型別的化名本身當成 JSON 物件，呼叫 Red 屬性，實質上是列舉的元素之一 */
let selectedColor = Colors.Red;
```

變數之推論結果也會是列舉型別的化名（圖 4-4）。

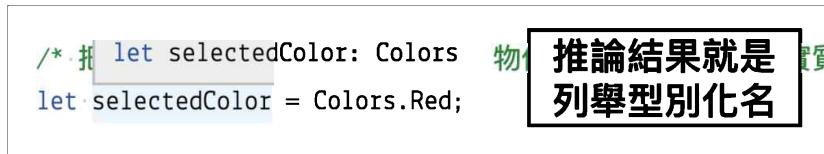


圖 4-4 列舉型別可以被推論得出來

不過應該算是很明顯的吧？畢竟只要看到化名 `Colors` 就會知道它被宣告過為列舉型別化名，理應來說可以這樣推論。

另外，如果是宣告為常數的話：

```
/* 宣告成常數，並且指派列舉型別裡的成員，就會被推論為列舉成員型別 (Enum Member Type) */
const selectedColor = Colors.Red;
```

推論結果會變成列舉成員型別（`Enum Member Type`），而非列舉型別。（圖 4-5）



圖 4-5 推論結果為 `Colors.Red`，為列舉成員型別

另外，讀者可以想成你將列舉型別中的其中一個成員指派到常數中，跟原始型別的值指派到常數的行為很像⁴。簡而言之，列舉成員型別就很像明文型別版本

4 明文型別，參見條目 3.6.1。

的列舉，只是官方將其稱之為列舉成員型別（Enum Member Type）而非列舉明文型別（Enum Literal Type，此名稱並非官方說法）⁵。

4.2.3 型別註記機制

註記部分就是註記列舉型別本身的名字，就像條目 4.2.1 提到的：「宣告列舉的同時，等同於宣告新的型別化名」——用列舉的名稱保證沒錯：

```
let selectedColor: Colors = Colors.Red;
```

當然，你也可以選擇註記列舉成員型別，只是結論會跟條目 3.6.2 提到的結論相似，註記明文型別的意義似乎不大，而註記列舉成員型別也是意義不大。

```
let selectedColor: Colors.Red = Colors.Red;
```

畢竟上面那一行跟前個條目的範例：

```
const selectedColor = Colors.Red;
```

有何不同呢？

基本上作者很少會替變數註記列舉型別（推論就很夠用了），除非是非得註記之情境，譬如宣告函式⁶時，參數必須得註記時，才有可能會註記到列舉型別。

》》列舉型別的意義與使用

1. 列舉（Enum）為 TypeScript 自定義型別，為一系列性質相似之成員（Member）的集合，用 JSON 物件的方式表示，但是有別於 JSON 物件，列舉會特別自成一種型別。
2. 列舉儘管說是 JSON 物件的一種，但內部的屬性正式名稱為列舉成員（Member）

5 但官方確實將呼叫列舉成員的式子稱之為 Enum Literal Expression，所以作者也不清楚名稱這樣亂取來取去是發生了什麼事。

6 通常宣告函式時，由於參數無法被編譯器推論型別結果，因此註記是必要的，函式型別詳細內容參見條目 3.3。

3. 宣告列舉 E 時，假設成員為 M₁、M₂ … 、M_N，則宣告方式如下：

```
enum E { M1, M2, ... MN };
```

4. 列舉型別與列舉成員型別（Enum Member Type）不同，前者是指整個列舉化名背後代表的物件、後者是指列舉的其中一個成員。
5. 若將列舉成員指派到常數（Constant）中，則該常數的推論結果為該列舉成員值的型別，也就是列舉成員型別。

4.2.4 自訂列舉成員 Custom Members

TypeScript 的列舉中，列舉成員事實上有代表值的。要驗證是代表什麼樣的值的最簡單方式是用 `console.log` 將列舉成員印出東西來：

```
enum Colors { Red, Blue, Yellow };

/* 印出列舉成員 */
console.log(Colors.Red);
console.log(Colors.Blue);
console.log(Colors.Yellow);
```

還記得如何編譯 TypeScript 檔案吧⁷？簡簡單單地下達：

```
$> tsc < 檔案名稱 >.ts
```

就可以編譯出結果，並且使用 NodeJS 執行編譯過後的檔案：

```
$> node < 檔案名稱 >.js
```

如果將上面的範例用 `tsc` 編譯並且使用 `node` 執行編譯檔案，就會出現以下的結果：

```
0  
1  
2
```

⁷ TypeScript 編譯方面的指令參見條目 1.5。

事實上，列舉型別在宣告時，預設為數字型列舉型別（Numeric Enum Type），代表列舉成員會對應從 0 開始遞增的數字值。

而如果你扒開編譯結果，其實也不難理解為何列舉的成員對應值是數字：

```
var Colors;  
(function (Colors) {  
    Colors[Colors["Red"] = 0] = "Red";  
    Colors[Colors["Blue"] = 1] = "Blue";  
    Colors[Colors["Yellow"] = 2] = "Yellow";  
})(Colors || (Colors = {}));  
;  
/* 印出列舉成員 */  
// 略 ...
```

以上為將範例程式碼編譯過後的結果，從這裡可以看出列舉預設就是會將每個成員對應從數字 0 持續遞增的值以及列舉的逆向映射性——這部分會在下個條目講到。

而這個數字型列舉的成員對應值是可以被修改的，例如：

```
enum Colors {  
    Red = 1, // ← console.log(Colors.Red) 之結果會是 1，而非預設的 0  
    Blue = 2,  
    Yellow = 3  
};
```

這樣會使得列舉的成員對應值被修改成其他數值。不過以上的寫法甚至可以簡寫成：

```
enum Colors {  
    Red = 1,  
    Blue, // ← Colors.Blue 對應結果，預設為前一個成員值 + 1，也就是 1 + 1 = 2  
    Yellow // ← Colors.Yellow 對應結果，預設為前一個成員值 + 1，也就是 2 + 1 = 3  
};
```

所以你若改成：

```
enum Colors {  
    Red = 1000,
```

```
Blue,  
Yellow  
};
```

這樣 `Colors.Blue` 會預設對應的前一個成員值遞增數字 1，也就是數字 1001，`Colors.Yellow` 則會是數字 1002。

那猜猜看，這樣會不會壞掉？

```
enum Colors {  
    Red,  
    Blue = 0,  
    Yellow  
};
```

事實上，TypeScript 將其編譯結果會是 `Colors.Red` 與 `Colors.Blue` 對應值都會是一樣的，也就是數字 0。所以很諷刺的是，這樣寫的結果就會變成 `true`：

```
console.log(Colors.Red === Colors.Blue);
```

而且 TypeScript 不會幫你抓這種錯誤，不過作者應該也認為讀者不會隨隨便便這樣整自己或者是和你合作專案的同事吧！但你明目張膽地這樣寫，TypeScript 還真的不會跳出任何警告：

```
/* 世上會有什麼樣原因讓你想這樣寫？ */  
enum Colors {  
    Red     = 123,  
    Blue    = 123,  
    Yellow = 123  
};
```

所以這裡得出的結論是——編譯器會認列舉的成員名稱，但不會認你成員的值到底有沒有重複等不合理的地方。

另外，既然作者提到了數字型列舉型別外，理應來說就會有其他形式的列舉型別；其中，字串就是允許的列舉成員對應值——將字串當成列舉對應值的型別又被稱為字串型列舉型別（String Enum Type）。

```
enum Colors {
```

```
Red      = 'Red',
Blue     = 'Blue',
Yellow   = 'Yellow'
};
```

以上的字串型列舉寫法，編譯結果才是完完全全等效於：

```
const Colors = {
  Red: 'Red',
  Blue: 'Blue',
  Yellow: 'Yellow'
};
```

另外，由於列舉型別若沒標示成員之對應值時，會查看前一個列舉成員的值並進行遞增的動作。所以以下的寫法會出現錯誤。(如圖 4-6)

```
enum Colors {
  Red = 'Red',
  Blue,
  Yellow
};
```

```
enum Colors {
  Red = 'Red',
  Blue,
  (enum member) Colors.Blue
};
Enum member must have initializer. ts(1061)
```

**前一個成員對應值若為字串，
後面的成員就必須要指定值**

圖 4-6 可不要以為 Colors.Blue 會有對應值，因為 Colors.Red 不為數字

不過，如果是這樣寫就沒啥問題了：

```
enum Colors {
  Red,
  Blue,
  Yellow = 'Yellow'
};
```

因為前面的 Red 與 Blue 成員值各為數字 0 或 1，最後的 Yellow 對應是自定義不會影響到前面。

這樣寫也不會有問題：

```
enum Colors {  
    Red     = 'Red',  
    Blue    = 1,  
    Yellow  
};
```

因為成員 `Blue` 被指派為數字時，成員 `Yellow` 就可以參考成員 `Blue` 的值並進行遞增的動作。

另外，以上的兩種寫法由於混合了數字或字串型別的值，這種列舉已經不成列舉樣又被稱作異質型列舉型別（Heterogeneous Enum Type）。

而列舉的成員對應值，只能為字串或者是數字型別這兩種，如果你塞這兩種型別以外的值，TypeScript 會禁止你這麼做（就算它是原始型別但不是字串或數字也不行）：

```
enum Colors {  
    Red     = true,           // ← 不能用布林值  
    Blue    = null,           // ← 不能用 null  
    Yellow = { foo: 123 }    // ← 不能用物件  
};
```

不過列舉的自訂成員型別值也可以用前面的成員值進行複雜運算：

```
enum Colors {  
    Red,  
    Blue = Red + 2,          // ← Colors.Blue 的結果會變成 2  
    Yellow = Blue + 3        // ← Colors.Yellow 的結果會變成 5  
};
```

這種可以用表達式指派值到成員的列舉型別還有一套複雜的運算規則⁸，這裡就放備註讓讀者去參考。

8 TypeScript Computed Enum Type：<https://www.typescriptlang.org/docs/handbook/enums.html#computed-and-constant-members>。

4.2.5 逆向映射性 Reverse Mappings

看完了莫名其妙的列舉寫法後，列舉的東西還沒講完。因為剛剛看到範例的程式碼：

```
enum Colors { Red, Blue, Yellow };
```

它的編譯結果為：

```
var Colors;
(function (Colors) {
    Colors[Colors["Red"] = 0] = "Red";
    Colors[Colors["Blue"] = 1] = "Blue";
    Colors[Colors["Yellow"] = 2] = "Yellow";
})(Colors || (Colors = {}));
;
```

有些讀者可能看不懂為何要編譯成這坨很奇怪的東西，作者拆其中一行給讀者看：

```
Colors[Colors["Red"] = 0] = "Red";
```

先來看看最內層：

```
Colors[Colors["Red"] = 0] = "Red";
```

有讀者可能會疑問：「指派式這樣寫是可以的嗎？這樣難道不會出錯嗎？」

別忘了，作者在條目 2.2.2 的部分有深論過表達式（Expression）與敘述式（Statement）的差異，如果是用關鍵字 `let` 或 `const` 的變數宣告式，而由於變數宣告指派式為敘述式的一種，代表以下這一行是不會回傳任何值。

```
let foo = 123;
```

也就是說，這種敘述式是不能這樣被亂塞到任何需要值的地方：

```
Colors[let Colors["Red"] = 0] = "Red";
```

然而，如果讀者做過第二章習題中的第 7 題（或至少看一下解答篇的解析），應該會得到這個結論：「非宣告變數式的指派式為表達式的一種」。

所以以下這一行單純的指派式子為表達式的一種：

```
foo = 123;
```

儘管它看起來很像敘述式（但在 JavaScript 裡它不是敘述式！），它除了會對 `foo` 指派數值 123 外，它還會作為表達式，回傳數值 123。因此：

```
Colors[Colors["Red"] = 0] = "Red";
```

除了會將 `Colors["Red"]` 設為數值 0 外，該表達式還會回傳數值 0。

所以可以轉換成這樣：

```
Colors[0] = "Red";
```

不知道讀者有沒有看到一個亮點：

```
Colors[0] = "Red";
```

上面這一行將 `Colors.Red` 這個成員對應的結果（也就是數字 0）對應到成員的名稱 `Red` 上；這種相互映射的特性，官方將它稱之為逆向映射性（Reverse Mapping）。

所以讀者是可以利用列舉成員值去反向查詢列舉的名稱的：

```
Colors[0] === 'Red'; // ← 在 TypeScript 裡，這樣使用為 true
```

然而，這裡有個前提——如果列舉型別成員對應的值為字串型別，且成員的名稱與字串型別不一樣時，逆向映射性就不一定成立。

首先是成員名稱對應字串型別的情境：

```
enum Colors {  
    Red      = 'Red',  
    Blue     = 'Blue',  
    Yellow   = 'Yellow'  
};
```

以上的編譯結果為：

```
var Color;  
(function (Color) {
```

```
Color["Red"] = "Red";
Color["Blue"] = "Blue";
Color["Yellow"] = "Yellow";
})(Color || (Color = {}));
;
```

節選其中一條，這樣的寫法就有保持到逆向映射的性質喔，畢竟值如果跟屬性一樣，當然也可以反向把值作為屬性參照回去：

```
Color["Red"] = "Red";
```

然而，如果是成員名稱與對應的字串值不一樣的情形時：

```
enum Colors {
    Red      = 'Rot',
    Blue     = 'Blau',
    Yellow   = 'Gelb'
};
```

編譯結果會變成：

```
var Color;
(function (Color) {
    Color["Red"] = "Rot";
    Color["Blue"] = "Blau";
    Color["Yellow"] = "Gelb";
})(Color || (Color = {}));
;
```

節選其中一行時：

```
Color["Red"] = "Rot";
```

由於編譯過後屬性名稱是 "Red"，但對應的值是 "Rot"，你就不能反向使用 "Rot" 來反查成員名稱 "Red"。

»» 列舉型別的逆向映射性 Reverse Mapping

1. 列舉宣告時，成員名稱會與預設的值具備逆向映射性，意思就是說，可以用值去反推成員的名稱。

2. 若列舉成員對照的值為字串型別時，且若成員名稱與值為不同名稱，逆向映射性就不會成立。

4.2.6 常數列舉型別 Constant Enum Type

最後一個要講到的東西，作者認為很重要——列舉也可以被宣告為常數列舉型別。

讀者可能看到下面這一行會覺得：「這沒啥大不了的啊，列舉本來不就是被宣告成常數的樣子嗎？分什麼常數不常數的？」

```
const enum ConstantColors { Red, Blue, Yellow };
```

這一行在 TypeScript 程式碼裡的確就只是多了 `const` 這個關鍵字，宣告其為常數的列舉，但關鍵就在於被編譯過後的結果。

比較一下以下這段範例程式碼：

```
/* 普通的列舉型別 */
enum Colors { Red, Blue, Yellow };
let selectedColor = Colors.Red;

/* 常數版本的列舉型別 */
const enum ConstantColors { Red, Blue, Yellow };
let selectedConstantColors = ConstantColors.Red;
```

以下是其編譯過後的結果：

```
/* 普通的列舉型別 */
var Colors;
(function (Colors) {
    Colors[Colors["Red"] = 0] = "Red";
    Colors[Colors["Blue"] = 1] = "Blue";
    Colors[Colors["Yellow"] = 2] = "Yellow";
})(Colors || (Colors = {}));
;
var selectedColor = Colors.Red;
;
var selectedConstantColors = 0 /* Red */;
```

首先，編譯過後的普通列舉型別非常正常，就是一大堆 JSON 物件在那邊映射來映射去的指派結果。

但換成常數版本的列舉型別，它被編譯後竟然只剩下：

```
; // ← 這個分號也是編譯結果，並不是作者打字多打出來的喔！  
var selectedConstantColors = 0 /* Red */;
```

以上所顯示的這樣。

如果學過編譯器優化（Compiler Optimization）相關主題的讀者們，應該有聽過一種優化技巧叫做 Constant Propagation——也就是常數傳遞相關的演算法。

試想一件事情，今天如果你想要優化下面的 JavaScript 程式碼，你會怎麼處理：

```
const constantNum = 123;  
let mutableNum = constantNum * 456;  
  
console.log(constantNum);  
mutableNum = (constantNum + 1) / 2;
```

宣告常數（Constant）最重要的一件事情是，它可以完全排除掉變異（Mutate）的可能性，也就是說既然上面的程式碼已經告訴你 `constantNum` 已經確定會是數字 123 話，它就可以將此資訊傳遞到後面，將程式碼優化成：

```
let mutableNum = 123 * 456; // ← 將 constantNum 取代為 123  
  
console.log(123); // ← 將 constantNum 取代為 123  
mutableNum = (123 + 1) / 2; // ← 將 constantNum 取代為 123
```

當然以上的結果有些表達式既然已經變成純然數字四則運算，理應來說還可以再更進階地優化，不過這並不是本書討論的範疇。

所以回歸常數列舉型別：

```
/* 常數版本的列舉型別 */  
const enum ConstantColors { Red, Blue, Yellow };  
let selectedConstantColors = ConstantColors.Red;
```

如果按照常數傳遞（Constant Propagation）的邏輯方式來看上面的案例，我們可以將出現在程式碼裡的所有 `ConstantColors.Red` 取代為數字 0；同理，`ConstantColors.Blue` 取代為數字 1、`ConstantColors.Yellow` 取代為數字 2。

》》 常數列舉型別與普通列舉型別差異

1. 普通列舉型別會被編譯成 JSON 物件，會佔掉一些空間。
2. 常數列舉型別需要使用 `const` 關鍵字進行列舉型別的宣告，而編譯結果會將程式後續使用到的列舉成員取代為該成員對應的值，有效減少程式碼產生的量。

前面兩節已經介紹完 TypeScript 提供的兩種偏向直觀的特殊型別後，接下來要介紹的型別長相與模樣會越來越奇怪神奇。

► 4.3 可控索引型別與索引型別 Indexable Type & Index Type

4.3.1 可控索引型別 Indexable Type

讀者看到標題可能認為作者已經放棄先討論型別推論後解析型別註記的機制。其實並不是不討論，是根本沒辦法討論。因為推論的行為情形基本上都已經被第三章各種不同型別 Cover 掉了，所以也不會有什麼新型的推論結果。

回到頭來，可控索引型別（Indexable Type）這種超難翻譯的型別到底是啥呢？這種型別可以控制 JSON 物件的鍵（Key），也就是索引（index）對應的型別，也就是說：

```
type StringDictionary = { [key: string]: string };
```

會嚴格要求任何被註記為 `StringDictionary` 型別的值，其鍵必須要為型別 `string`，而對應之值也是型別 `string`：

```
const obj: StringDictionary = {};  
  
obj[/* 這裡面的東西只能為字串型別的值 */] = 'Hello world';
```

其中的宣告可控索引型別中的 `[key: string]` 這段，官方將其稱之為索引簽章（Index Signature，但應該也可以翻譯成索引記號）。

至於為何會需要這種型別呢？如果你想要控制開發者可以任意新增 JSON 物件的屬性並且限制住值的型別，就可以使用可控索引型別。

不過以下的寫法儘管鍵被輸入數字型別的值，但由於任何鍵都會被轉型為字串，因此是可以被接受的寫法：

```
const obj: StringDictionary = {};  
  
/* 儘管 123 非字串型別，但任何型別的值作為 JSON 物件的鍵會被自動轉型為字串型別 */  
obj[123] = 'Hello world';  
  
/* 以上的寫法等效於： */  
obj['123'] = 'Hello world';
```

基本上通常把鍵設定為字串型別就只有限制物件的值之型別這個效果而已。

另外，鍵也可以被設定為數字型別，此時就會變成像是在模擬陣列（Array）的行為：

```
type StringArray = { [key: number]: string };
```

所以以上的型別化名若註記到變數上，還真的可以這樣寫：

```
const obj: StringArray = ['Hello', 'World', 'TypeScript'];
```

不過傳統的物件寫法也接受：

```
const obj: StringArray = { 0: 'Hello', 1: 'World', 2: 'TypeScript' };
```

然後指派值到型別為 `StringArray` 的變數裡，也必須用數字型別的值來指派：

```
obj[123] = 'Foo';
```

如果換成字串當然就會出錯囉！（錯誤訊息如圖 4-7）

```
obj['123'] = 'Foo';
```

Element implicitly has an 'any' type because index expression is not of type 'number'. ts(7015)

Peek Problem No quick fixes available

```
obj['123'] = 'Foo';
```

圖 4-7 鍵（或者是索引）的型別不為數字型別就會發生錯誤

另外，我們也可以用數字型別的索引模擬陣列型別的樣子：

```
type StringArray = {
  [key: number]: string;
  length: number;
};
```

這樣一來，你就可以模擬陣列型別的樣貌：

```
const obj: StringArray = ['Hello', 'World', 'TypeScript'];
console.log(obj.length); // ← 若讀者自行編譯過後，執行之輸出結果為 3
```

另外，可控索引型別由於是 JSON 物件型別的延伸，所以根據條目 3.3.5，它也可以標上唯讀操作符（Read-Only）：

```
type ReadonlyStringArray = {
  readonly [key: number]: string;
  length: number;
};
```

以下的程式碼，如果變數被註記為 `StringArray`，它的值是可以被竄改；而如果註記為 `ReadonlyStringArray`，就變成所有的值都是唯讀狀態，不能夠隨隨便便被竄改：

```
const obj: StringArray = ['Hello', 'World', 'TypeScript'];
obj[0] = 'Bye Bye!'; // ← 不會發生錯誤
```

```
const readonlyObj: ReadonlyStringArray = ['Hello', 'World', 'TypeScript'];
readonlyObj[0] = 'Bye Bye!'; // ← 唯讀狀態，會被丟出錯誤警訊
```

錯誤提示如圖 4-8。

```
const readonlyObj: ReadonlyStringArray
Index signature in type 'ReadonlyStringArray' only permits reading. ts(2542)
Peek Problem No quick fixes available
readonlyObj[0] = 'Bye Bye!'; // ← 唯讀狀態，會被丟出錯誤警訊
限制為唯讀狀態，因此不容許竄改
```

圖 4-8 “Only permits reading” 代表只有開放讀取的權限

另外，讀者可能想說，那既然可以模擬陣列的話，那這樣就可以用字串的索引方式，額外宣告其他索引存放的值對應型別：

```
type Invalid = {
  [key: string]: string;
  length: number;
};
```

而以上的寫法，很遺憾地，它是錯誤的，原因是屬性 `length` 的名稱本身是字串型別，早已經對應到索引簽章對應到的字串型別的值，但偏偏屬性 `length` 又被宣告為對應數字型別，這樣就變成前後矛盾。（錯誤訊息如圖 4-9）

```
type Invalid = {
  [key: string]: string;
  length: number;
}; (property) length: number
由於索引簽章被固定為對應到字串型別，length 屬性對應型別必須為字串型別
Property 'length' of type 'number' is not assignable to string index type 'string'. ts(2411)
```

圖 4-9 屬性 `length` 對應之型別與索引簽章對應的型別衝突

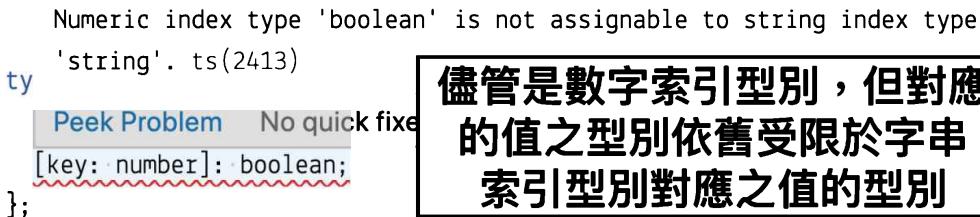
唯一的解法就是開放索引簽章的型別可以多出其他型別出現的可能性：

```
type Valid = {
  [key: string]: string | number; // ← 開放為 string 或 number 型別
  length: number;
};
```

那讀者肯定會想說，那也可以選擇將數字型索引存一種型別，字串則存另外一種：

```
type Invalid = {
  [key: string]: string;
  [key: number]: boolean;
};
```

很抱歉，這也是不行的，因為數字索引經過轉換過會成為字串型別的索引，所以等同於一切都是受制於字串型別的索引簽章對應的型別值。(圖 4-10)



```
Numeric index type 'boolean' is not assignable to string index type
'string'. ts(2413)
ty
  Peek Problem No quick fixe
  [key: number]: boolean;
};
```

儘管是數字索引型別，但對應的值之型別依舊受限於字串索引型別對應之值的型別

圖 4-10 數字索引型別對應之型別只能為字串索引型別對應到的值之型別

解套方式跟前一個案例很像：

```
type Valid = {
  [key: string]: string | boolean; // ← 開放為 string 或 boolean 型別
  [key: number]: boolean;
};
```

最後，索引簽章可不可為其他種型別呢？

```
type Invalid = {
  [key: boolean]: string;
};
```

答案是：不行，它只能為數字型別或字串型別這兩種。(參見圖 4-11 中的錯誤訊息)



```
type Invalid = {
  [key: boolean]: string;
}; (parameter) key: boole
An index signature parameter type must be either 'string' or 'number'.
```

跟你講明不能使用字串或數字以外的型別作為索引簽章裡的型別

圖 4-11 索引簽章裡的型別只能為字串或數字這兩種

»» 可控索引型別 Indexable Type

1. 為 JSON 物件型別的延伸，可以控制鍵值對的型別，並且不受限於擴充屬性的個數。
2. 控制鍵的型別部分被稱之為索引簽章（Index Signature）。
3. 索引簽章只能為字串型別或數字型別這兩種；若索引簽章為字串型別時，可以模擬普通的 Map（或 Dictionary）等資料結構；若索引簽章為數字型別時，則可以模擬陣列資料結構。
4. 若某 JSON 物件型別裡有索引簽章，想要擴充各種屬性對應之型別時，每個屬性的型別都會受限於字串索引型別對應的值之型別。

4.3.2 索引型別 Index Type

另一個東西很容易混淆的東西是索引型別（Index Type），但它的意思不是限制 JSON 物件的索引對應的型別（那是前個條目在做的事情）；本條目的索引型別單純只是將 JSON 物件裡的索引名稱聯集起來變成一種型別而已。

通常索引型別會看到使用關鍵字 `keyof`，最簡單的例子是：

```
type PersonalInfo = {  
    name: string;  
    age: number;  
    interest: string[];  
};  
  
/* 將 PersonalInfo 裡的索引聯集複合 */  
type KeyofPersonalInfo = keyof PersonalInfo;
```

其中，`PersonalInfo` 為一個 JSON 物件型別，而 `KeyofPersonalInfo` 則是將 `PersonalInfo` 裡所有的索引（也就是屬性）進行聯集的動作。（如圖 4-12）

```

type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};

type KeyofPersonalInfo = "name" | "age" | "interest"

type KeyofPersonalInfo = keyof PersonalInfo;

```

將 PersonalInfo 所有屬性聯集起來

圖 4-12 型別 PersonalInfo 的屬性被拔出來形成 'name' | 'age' | 'interest'

另外，運用索引型別，我們也可以叫出 JSON 物件裡的值的型別的聯集。(以下程式碼推論結果如圖 4-13)

```

type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};

type KeyofPersonalInfo = keyof PersonalInfo;

/* 運用索引型別，將值裡的型別進行聯集 */
type KeyofPersonalInfo = PersonalInfo[KeyofPersonalInfo];

```

```

    interest: string[];
};

```

PersonalInfo 裡的值之型別也可以被拔出來

```

type KeyofPersonalInfo = keyof PersonalInfo;
    type ValueOfPersonalInfo = string | number | string[]
type ValueOfPersonalInfo = PersonalInfo[KeyofPersonalInfo];

```

圖 4-13 連 JSON 物件型別裡所有值的型別也可以被拔出來

事實上，精確一點來說，我們也可以單獨叫出 JSON 物件裡某個屬性對應的值之型別——把 JSON 物件型別本身當成 JSON 物件，直接填入屬性也可以。

```

/* 也可以單獨將 JSON 物件裡的特定屬性對應的值的型別拔出來 */
type Interest = PersonalInfo['interest']; // ← 推論結果為 string[]

```

回過頭來，索引型別事實上只是一種名稱，描述的是——使用 `keyof` 將 JSON 物件裡的屬性型別拔出來聯集、以及將 JSON 物件裡屬性對應的值的型別拔出來的技巧而已。

「至於為何會需要用到索引型別呢？」這應該是讀者（以及作者）會問的問題。

有一種情境你可能會需要：假設你引用到的型別是來自外面的套件（以 JS 圈來說，就是所謂的 Node Package）、亦或者是一些專案內建的型別；然而，根據條目 3.3.3，不能輕易物件的完整性代表如果要讓 JSON 物件新增屬性是不太可能的事情，這時如果想要擴充物件屬性的話該怎麼辦呢？

如果說是自己專案內建的型別可能還可以自己竄改，但要是私自竄改外來套件的屬性時，你得承擔整個套件或者是相依套件（Dependency）出現型別錯誤的風險。如果 JSON 物件型別宣告時多了一個鍵，任何使用到該型別的值都得新增對應的鍵以及值。

其中一種解法就是，另外宣告新的 JSON 物件型別，使用本條目講到的索引型別技巧；假設想要擴充型別 `PersonalInfo`，新增 `email` 屬性對應字串型別的值，我們可以這麼做⁹：

```
/**  
 * 將 PersonalInfo 裡的所有索引拔出來後，並將每個索引對應的值之型別  
 * 也對照進去；最後將要新增之屬性與屬性對應的值的型別交集複合起來  
 */  
type ExtendedPersonalInfo = {  
    [Key in keyof PersonalInfo]: PersonalInfo[Key];  
} & {  
    email: string  
};
```

推論結果如圖 4-14。

9 實際上，這個結合 “`keyof`” 與關鍵字 “`in`” 的技巧稱之為 Mapping Type，可以參考：
<https://www.typescriptlang.org/docs/handbook/advanced-types.html#mapped-types>。

```
type ExtendedPersonalInfo = {
  [Key in keyof PersonalInfo]?: PersonalInfo[Key];
} & {
  name: string;
  age: number;
};
interest: string[];
} & {
  email: string;
}
```

**PersonalInfo 的型別
結構被複製出來外，
還可以額外新增屬性**

圖 4-14 結構完整地被複製過去，並且可以新增型別

以上的寫法應該是讀者第一次看到複合型別¹⁰的另外一個形式——交集複合(Intersection)，可以想成是邏輯「和」(AND)的方式將兩個型別的結構複合起來。所以圖 4-14 推論出來的結果是屬性 `name` 對應字串型別、屬性 `age` 對應數字型別、屬性 `interest` 對應字串陣列型別“和”屬性 `email` 對應字串型別。

事實上，以上的範例程式碼如果只是單純複製物件的結構，直接簡寫成這樣就好，不必那麼辛苦：

```
type ExtendedPersonalInfo = PersonalInfo & { email: string };
```

但是如果又有特殊需求，比如你突然想要將所有 `PersonalInfo` 裡的屬性轉換成選用屬性，你可以這麼做：

```
type OptionalPersonalInfo = {
  [Key in keyof PersonalInfo]?: PersonalInfo[Key];
};
```

以上的寫法推論結果如圖 4-15。

```
type OptionalPersonalInfo = {
  name?: string | undefined;
  age?: number | undefined;
  interest?: string[] | undefined;
}
type OptionalPersonalInfo = {
  [Key in keyof PersonalInfo]?: PersonalInfo[Key];
};
```

**全部都被洗成
選用的屬性**

圖 4-15 原來還可以這樣轉換？

¹⁰ 複合型別請參見下一個條目。

當然，物件的唯讀屬性除了使用條目 3.3.5 講過的 `Readonly<T>` 的寫法外，另
一種等效的寫法為：

```
type ReadonlyPersonalInfo = {  
    readonly [Key in keyof PersonalInfo]: PersonalInfo[Key];  
};
```

► 4.4 複合型別 Composite Type

4.4.1 聯集複合 Union Type

總算進到了從第一章看到現在似乎無所不在的複合型別條目；聯集複合的語法
應該看過很多次，所以本條目可以輕鬆快速地帶過。(騙你的)

首先是最簡單的範例：

```
type Primitives = number | string | boolean | null | undefined;
```

以上的範例就是將所有原始型別進行聯集複合的結果，意思就是說任何被註記
為 `Primitives` 的變數都可以填入任何原始型別的值。

聯集型別的標準符號就是管線符號（Pipeline，也就是 `|`），代表邏輯「或」的
概念。

另外，JSON 物件的聯集複合的案例，以條目 3.6.3 講到的互斥聯集
(Discriminated Union) 的技巧最為常用：

```
type Rectangle = {  
    type: 'Rectangle';  
    width: number;  
    height: number;  
};
```

```
type Triangle = {  
    type: 'Triangle';  
    base: number;
```

```
height: number;
};

type Circle = {
    type: 'Circle';
    radius: number;
};

type Shapes = Rectangle | Triangle | Circle;
```

當然你也可以恣意地亂將任意型別進行聯集複合，任何有可能造成邏輯「或」的推論狀況都會用聯集複合的方式表示出型別推論結果。讀者若還記得條目 3.2.1 討論到的型別推論的特點中——「匯集性」的這個特點就是運用聯集複合來解決——當成是遇到分岔點（Branching Point），如條件敘述式等，就有可能會造成型別推論結果用到聯集複合的情形。

```
let numberOrString;

if /* 條件 */ {
    numberOrString = 123;
} else {
    numberOrString = 'Hello world';
}

numberOrString; // ← 推論結果會是 number | string
```

»》 聯集複合型別 Union Type

1. 聯集型別為某變數或表達式可能具備多種不同情形的型別。
2. 通常遇到程式分岔點（Branching Point），就有機會用到聯集型別描述變數或表達式可能的推論結果（根據條目 3.2.1 裡提到的型別推論機制的匯聚性）
3. 若宣告某型別 Tunion 為多個型別 T_1 、 T_2 …、 T_N 的聯集，則寫法如下：

```
type Tunion = T1 | T2 | ... | TN;
```

4.4.2 型別駐防 Type Guard¹¹

通常使用到聯集複合過後的型別，免不了會遇到情境是必須根據聯集的各自型別，會有相對應的處置辦法，例如宣告 `safeAddition` 函式，確保輸入不管是字串還是數字，一率都會轉成數字型別進行加法運算：

```
function safeAddition(input1: number | string, input2: number | string) {
    let safeInput1: number, safeInput2: number;

    /* 如果 input1 為字串，則轉成數字並指派到 safeInput1 */
    if (typeof input1 === 'string') {
        safeInput1 = parseInt(input1, 10); // ← 十進位字串型數字轉純數字
    } else {
        safeInput1 = input1;
    }

    /* 如果 input2 為字串，則轉成數字並指派到 safeInput2 */
    if (typeof input2 === 'string') {
        safeInput2 = parseInt(input2, 10); // ← 十進位字串型數字轉純數字
    } else {
        safeInput2 = input2;
    }

    return safeInput1 + safeInput2;
}
```

其中：

```
if (typeof input1 === 'string') /* ... 略 */
```

以上這一行就是在做所謂的型別駐防，確保變數 `input1` 如果是字串時，應該要轉型成數字然後再指派到 `safeInput1`。

¹¹ Type Guard 似乎沒有正式的中文翻譯名，所以作者在寫網路文章時，之前是會將 Type Guard 翻譯為「型別檢測」，此譯名是根據它的功能—也就是負責檢測某變數或表達式的型別—來翻譯的；而「型別駐防」是作者認為可以兼顧英文直翻中文，但也可以稍微表達到檢測型別功能的最接近翻譯。

仔細看變數 `input1` 在判斷式裡外駐防前後的推論結果，差別如圖 4-16 與 4-17。

```
/* 如果 input1 (parameter) input1: string | number */
if (typeof input1 === 'string') {
    safeInput1 = parseInt(input1, 10); // ← 十進位字串型數字轉純數字
} else {
    safeInput1 = input1;
}
```

**判斷式部分，駐防前，`input1` 推論結果是
string 與 number 的聯集**

圖 4-16 判斷式為駐防的前線，因此推論結果依然是聯集的結果

```
/* 如果 input1 為字串，則轉成數字並指派到 safeInput1 */
if (typeof input1 === 'string') {
    safeInput1 = parseInt(input1, 10); // ← 十進位字串型數字轉純數字
} else {
    safeInput1 = input1;
}
```

**經過判斷式裡的型別駐防過程，
`input1` 變成單純的字串型別**

圖 4-17 經過駐防門檻，自然剩下字串型別的結果

「通常」原始型別的駐防都可以用 `typeof` 操作符去處理，而 `typeof` 旁邊接的東西不管是什麼，一率只會回傳幾種值：

- 原始型別的字串值："number"、"string"、"boolean"、"symbol"、"undefined"。
- 物件型別的字串值："object"、"function"。

`typeof` 操作符部份是原生 JavaScript 擁有但也卻讓作者覺得很弔詭的東西——既然有 "undefined"，那麼 "null" 跑到哪裡去了？

事實上，`typeof null` 的結果是 "object"，但這不是本書重點¹²。

另外，如果是函式物件的話反而可以用 `typeof` 操作符，但作者沒碰過駐防的門檻為函式型別的物件，好像沒有檢測參數結構這種需求或特異功能吧？其他

12 參見 Stackoverflow，Why is `typeof null` “object” ? <https://stackoverflow.com/questions/18808226/why-is-typeof-null-object>。

像是 JSON 物件或陣列對應的就是 "object"，這應該沒有太大爭議。

接下來的東西很細節，這應該是學 JavaScript（或 TypeScript）很麻煩的點，抱怨 JavaScript 關於比較（Comparison）¹³ 方面的聲浪也不曾減弱。

首先，`null` 既然無法用 `typeof` 操作符去判斷的話，直接用嚴格比較操作子（Strict Comparison）就好了（就是三個等號的操作子）：

```
something === null; // ← 若 something 為 null 則 true
```

再來，陣列是特殊的 JavaScript 物件，但不太建議用 `typeof` 操作符檢測是因為很容易跟 JSON 物件混淆；用 `Array.isArray` 這個方法會比較適合喔：

```
Array.isArray(something); // ← 若 something 為陣列則 true
```

其實還有很細節的東西，數字就是一種；讀者可能會問，數字不就是原始型別，而且用本書剛剛講的 `typeof` 操作符不就好了？鬼打牆嗎？

其實嚴格來說，數字的範疇裡還有不是數字的數字（也就是 `NaN`）以及代表無限大的數字（也就是 `Infinity`）；前者可以使用 `Number.isNaN` 檢測，後者則是用 `Number.isFinite` 檢測數字是不是有限的：

```
/* 若 something 為 NaN 則 true */
Number.isNaN(something);

/* 若 something 為 Infinity 或 -Infinity 則 false */
Number.isFinite(something);
```

請不要自作主張想說用 `null` 的判斷手法來判斷 `NaN` 或 `Infinity`：

```
/* NaN 直接與 NaN 比較輸出結果反而是 false */
NaN === NaN;

/* 輸出結果雖然是 true，但 Number.isFinite 的檢測手法會比較 "成熟" 些 */
Infinity === Infinity;
```

13 參見 JavaScript Equality Table，<https://dorey.github.io/JavaScript-Equality-Table/>。

沒辦法，這就是 JavaScript。

另外，物件的比較通常也不用 `typeof` 操作符，而是該物件屬於的類別 (Class)。條目 2.3.2 有提到——類別建構出來的物件被稱之為實體 (Instance)，所以 JavaScript 衍生出 `instanceof` 操作符。以常見的 `Date` 類別為例：

```
/* 從 Date 類別建構出實體，並指派到變數 now 裡 */
const now = new Date();

/* 此為 true，因為 now 存的東西確實是 Date 類別建構出的實體 */
now instanceof Date;
```

類別會在下一個章節談論地更詳細，這裡只是告訴讀者，基本上，通常不會比物件的結構，而是根據物件（即實體）所屬類別來判斷。

不過有些讀者可能有碰過判斷物件屬性存不存在的情境，這時候會用 `in` 操作符或物件通常會有的 `hasOwnProperty` 方法來判斷；只可惜，TypeScript 並沒有這部分的型別駐防功能。(以下的程式碼，駐防的前後結果分別為圖 4-18 與 4-19)

```
// 由於 age 是選用屬性，因此可省略
const info: { name: string; age?: number } = { name: 'Max' };

// 使用 hasOwnProperty 檢測；改用 'age' in info 的寫法也是可以
if (info.hasOwnProperty('age')) {
    console.log(info.age);
}
```

```
const info: {
    name: string;
    age?: number | undefined;
} // 由於 age 是選用屬性，因此可省略
const info: { name: string; age?: number } = { name: 'Max' };

// 使用 hasOwnProperty 檢測；改用 'age' in info 的寫法也是可以
if (info.hasOwnProperty('age')) {
    console.log(info.age);
} // 型別駐防線外，選用屬性 age 可能為 undefined 或數字
```

圖 4-18 駐防線外，`age` 為選用屬性，因此有可能為 `undefined`

```
// 由於 age 是選用屬性，因此可省略
const info: { name: string; } = {
  name: 'Max'
};

// 使用 hasOwnProperty 檢查物件是否有 age 屬性
if (info.hasOwnProperty('age')) {
  console.log(info.age);
}
```

就算是駐防線內，依舊沒辦法將 age 屬性給完全推論為數字型別

圖 4-18 駐防線內，age 為依然為選用屬性

其實如果仔細想過原因的話，以下的這種情形用 `in` 操作符或 `hasOwnProperty` 方法防也防不了的：

```
// 有 age 屬性，但就是用 undefined 來找 TypeScript 麻煩
const info: { name: string; age?: number } = { name: 'Max', age: undefined };
```

》》 型別駐防 Type Guard

1. 由於聯集型別（Union Type）涵蓋多種型別的可能情境，針對不同型別採取不同處理方式時，必須用型別駐防來個別處理不同的型別狀態。
2. 除了 `null` 物件外，原始型別可以用 `typeof` 操作符設立駐防條件。
3. `null` 物件直接用嚴格比較操作子設立駐防條件。
4. 陣列可以使用 `Array.isArray` 方法檢測。
5. 物件通常是用 `instanceof` 操作符檢測，確認該物件是否為特定類別建立出來的實體。
6. 特殊數字如 `NaN` 必須用 `Number.isNaN` 檢測；`Infinity` 則是用 `Number.isFinite` 檢測。

4.4.3 交集複合 Intersection Type

交集複合儘管是少見的複合形式，但仍然有存在的機會，不過大多數的交集複合如果隨機撮合型別交集時，會出現一些很神奇的結果。

首先，TypeScript 的型別系統裡有型別集合（Type Set）的概念；你可以想成本書第三、四章裡出現的型別種類都是屬於整個型別系統內部的元素。（如圖 4-19）

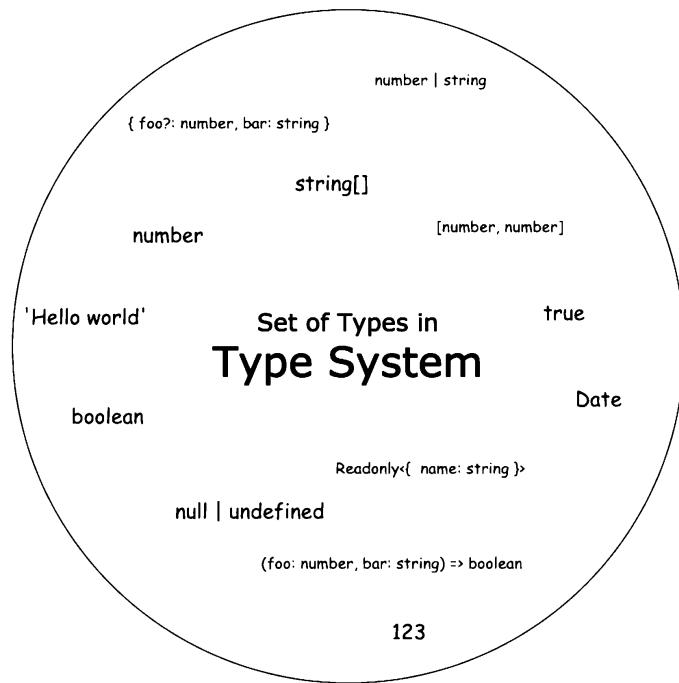


圖 4-19 整個型別系統裡的型別集合概念

集合也有相對層級的概念，譬如說，原始型別（Primitive Types）的集合就有包含數字、字串等型別；而數字的明文（Literal）、字串的明文等也是屬於原始型別的集合裡的元素。（圖 4-20）

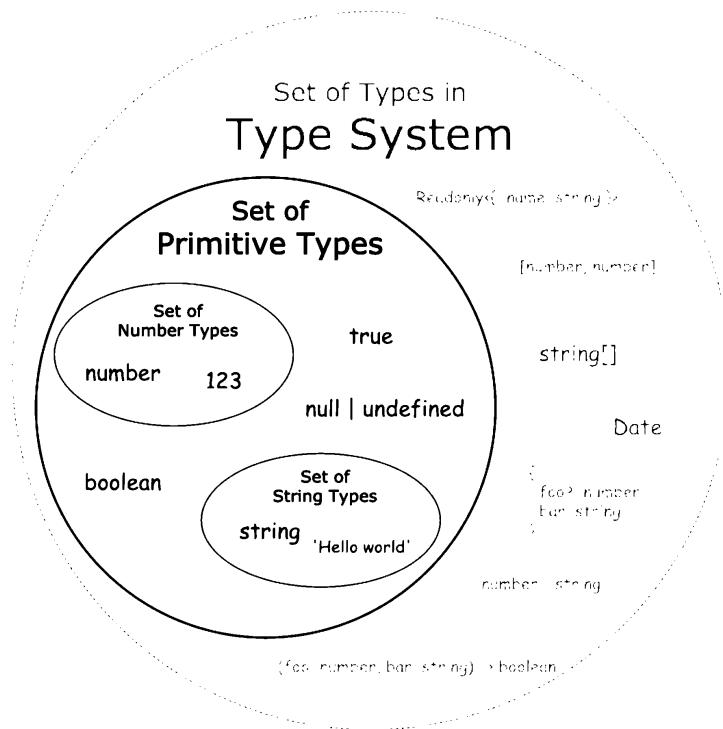


圖 4-20 型別集合裡相對層級的概念

這樣的型別集合可以切分成無限多種情型，理所當然地，交集的組合就可以有很多種；譬如單純地將 `null` 與 `undefined` 進行聯集，在 TypeScript 裡面稱這種聯集結果為 **Nullable Types**。

當然，你也可以將隸屬於不同分類或者是集合區塊、毫無關聯的型別互相聯集在一起，譬如將原始型別與物件型別聯集起來，這樣就會變成是跨不同子集合的聯集；不過如果是交集的話，就得根據交集的兩個型別的特性來討論。

最常用到的交集複合的情境是 JSON 物件的交集複合，例如：

```
type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};

type AccountInfo = {
    email: string;
    username: string;
    subscribed: boolean;
};

type UserAccount = PersonalInfo & AccountInfo;
```

以上的程式碼將兩種 JSON 物件型別 `PersonalInfo` 與 `AccountInfo` 交集複合變成 `UserAccount` 型別。所以任何變數被註記為 `UserAccount` 型別時，內含的物件必須要有型別 `PersonalInfo` 與 `AccountInfo` 的屬性。

```
/* 必須同時要有 PersonalInfo 以及 AccountInfo 的結構 */
const user: UserAccount = {
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
    email: 'example@email.com',
    username: 'Nordic Wyvern',
    subscribed: true,
};
```

如果少了其中一個屬性就會跳出錯誤訊息。(如圖 4-21)

```
type A const user: UserAccount
email Type '{ name: string; age: number; interest: string[]; email: string; username: string; }' is not assignable to type 'UserAccount'.
subs Property 'subscribed' is missing in type '{ name: string; age: number; interest: string[]; email: string; username: string; }' but required in type 'AccountInfo'. ts(2322)
type U hello-world.ts(65, 3): 'subscribed' is declared here.

/* 必須 Peek Problem No quick fixes available
const user: UserAccount = {
  name: 'Maxwell',
  age: 18,
  interest: ['drawing', 'programming'],
  email: 'example@email.com',
  username: 'Nordic Wyvern',
};
```

少了 subscribed 這個屬性
就會出現的錯誤訊息

圖 4-21 找不到屬性 subscribed 產生的錯誤訊息

交集型別使用的符號為 &，具備邏輯「和」(AND) 的概念，因此才會說交集型別是綜合兩個型別的特色融合在一起的結果。

作者其實也想不到除了對物件進行交集複合外的應用，因為通常對物件型別以外的其他型別交集過後就會出現下一個條目會講到的型別結果——never 型別。

»》 交集複合型別 Intersection Type

1. 交集型別代表某變數或表達式同時具備兩個型別特徵。
2. 通常交集型別只會用到 JSON 物件型別的交集，同時將多個物件型別的結構特徵融合在一起；而除了 JSON 物件型別以外的交集，“多數”的交集結果會變成 never 型別。(條目 4.5)
3. 若宣告某型別 $T_{intersection}$ 為多個型別 T_1 、 T_2 ... 、 T_N 的交集，則寫法如下：

```
type Tintersection = T1 & T2 & ... & TN;
```

► 4.5 Never 型別

4.5.1 程式執行不到結尾的應變措施

作者看大部分討論 Never 型別的網路文章等，幾乎都沒有細談到內部的細節。當然如果只提語法與用法時，應該不到一頁就可以終結掉條目 4.5。以下作者就先從淺層帶領讀者認識這個型別。

大部分對於 Never 型別的理解是——只要程式執行不到結尾的地方就會自動推論出 Never 型別；另一個產出 Never 型別的管道是——只要型別本身不可能發生時也會自動推論出 Never 型別喔。

前者說的程式執行不到的地方最主要有兩種，第一種是無窮迴圈。

```
/* 無窮迴圈 */
function forever(): never {
    while (true) {
        console.log('Executing ...');
    }
};
```

但是以上的程式碼如果你沒註記輸出為 `never` 型別時，會按照條目 3.4.1 的函式型別推論手法，看不到 `return` 敘述式，輸出型別為 `void`。而以上的程式碼手動註記 `never` 型別是告訴使用者，該函式不可能有執行終結的一天。(除非你強行終結程式)

另一種比較常見的案例是丟出錯誤訊息：

```
/* 丟出錯誤訊息 */
function throwsError(): never {
    throw new Error('Will never execute next line...');
};
```

與前面的程式碼案例一樣，如果不去註記輸出為 `never` 型別時，輸出型別就會被推論為 `void` 型別。

至於型別不可能發生的地方的 Case——此行為會在條目 4.5.3 討論到。

不去深究 `never` 型別時，其實重點大部分都講完了，就只有用在以上情形而已；不過 `never` 型別事實上重要性遠比上面講到的表面功能還來得大。

4.5.2 整個型別系統裡的基層型別

還記得條目 4.4.3 裡提到，型別系統其實是眾多型別聚集起來的巨型集合嗎？任何型別，甚至包含明文數字、明文字串等這種很細節的型別，最下面還有一個子型別集合——`Never` 型別。

換句話說：

Never 型別包含於所有型別集合裡，就連 `Never` 型別也是屬於 `Never` 型別本身集合。

上面那句話是以數學的集合（Set）意義的方式來解釋 `Never` 型別的特性，用程式語言的意義來形容的話：

任何型別集合都有涵蓋例外事件（Exception）發生的可能。

用一段程式碼就可以證明上面的兩個論述都是正確的：

```
/* 丟出錯誤訊息，但輸出型別為 number */
function throwsError(): number {
    throw new Error('Will never execute next line...');
};
```

以上這段程式碼，僅僅只是把輸出型別改成 `number` 型別，但讀者應該可以很明顯地看到，明明沒有任何 `return` 敘述式回傳跟數字型別有關的值，但是 `TypeScript` 編譯器分析認為這樣的結果是正確的。（圖 4-22）

```
/* 丟出錯誤訊息，但輸出型別為 number */
function throwsError(): number {
    throw new Error('Will never execute next line... ');
};
```

沒有任何波浪形的底線或錯誤警訊

圖 4-22 函式不會回傳任何東西，但註記輸出為數字型別沒有任何錯誤產生

讀者若覺得一頭霧水，不太理解原因的話，將範例改成這樣或許會明瞭許多：

```
/* 有機率丟出錯誤訊息或輸出數字，輸出型別為 number */
function possiblyThrowsErrorOrReturnsNumber(): number {
    if (Math.random() > .5) {
        return 123;
    }
    throw new Error('Will never execute next line...');
};
```

以上的程式碼也不會被發出任何警吶訊息喔！但是這裡就有可能輸出正常的數字型別外，也有機率丟出錯誤訊息，變成 Never 型別的情形。

所以根據以上的範例示範結果，除非是 TypeScript 對於 Never 型別的規範有偏誤，否則就只會有一種結論：

數字型別集合涵蓋例外事件（Exception）發生的可能。

也就是：

數字型別集合裡有包含 Never 型別。

細心的讀者如果去官方¹⁴ 查看 Never 型別的規範，以下這一句話就是形容 Never 型別的基礎特性：

The never type is a **subtype** of, and **assignable** to, **every type**; however, no type is a subtype of, or assignable to, never (except never itself). Even **any** isn't assignable to never.

直接翻譯就是說：「Never 型別是所有型別的子集合，而任何型別（除了 Never 型別外）都不是 Never 型別的子集合」；也就是說，所有型別都有包含 Never 型別（例外處理）的狀態。

¹⁴ 官方對於 Never 型別的描述，<https://www.typescriptlang.org/docs/handbook/basic-types.html#never>。

所以前面的範例，由於沒有回傳值，所以將輸出改型別 `void` 也是可以的：

```
/* 丟出錯誤訊息，但輸出型別為 void，代表無輸出 */
function throwsError(): void {
    throw new Error('Will never execute next line...');
};
```

因為 `Never` 型別屬於任何型別的集合，當然，“任何型別”也包含 `void` 這個東西。

»» Never 型別的意義

1. `Never` 型別為所有型別子集合。`Never` 型別也是 `Never` 型別本身的子集合。
2. `Never` 型別代表的意義廣義來說是程式無法執行到結束的概念，其中以代表例外（Exception）發生的意義為主。
3. 同第 2 點，任何型別都有包含 `Never` 型別這個 Case，以程式的觀點來看，就是指任何型別也涵蓋例外發生的可能性。

4.5.3 Never 型別衍生特性

前個條目探討到 `Never` 型別本身的意義——按照數學的觀點，任何型別都有涵蓋 `Never` 型別，也就是說任何型別與 `Never` 型別進行聯集複合（Union）時，就算任何型別多出 `Never` 型別的可能性，該型別也本身就已經有涵蓋 `Never` 型別，所以：

任何型別與 `Never` 型別發生聯集與沒發生聯集都沒差。

意思是說，任何型別跟 `Never` 型別聯集複合時：

```
type T = <Type> | never;
```

會等效於複合前的結果。（如圖 4-23）

```
type T = <Type>;
```

```
type NumberOrNever = number
type NumberOrNever = number | never;
```

**number 與 never 型別聯集時，
推論結果與沒有和 never 型別聯集的結論一模一樣**

圖 4-23 任何東西跟 Never 型別聯集，就跟沒有聯集的結果一樣

另外，由於反過來時，Never 型別除了自己本身外，並不包含其他型別，也就是說 Never 型別本身與其他型別都是互斥狀態——唯一的共通點只剩 Never 型別本身，所以任何與 Never 型別交集的結果都會是 Never 型別（連 any 型別交集 Never 型別結果也會是 never）。

意思是說，任何型別跟 Never 型別交集複合時：

```
type T = <Type> & never;
```

會等效於 never。（如圖 4-24）

```
type T = never;
```

```
type AbsoluteNever = never
type AbsoluteNever = number & never;
```

任何與 never 型別交集的型別，到最後只剩下 never 型別，因為 never 型別已經是整個型別系統的最底層

圖 4-24 Never 型別為整個系統的最基層型別，因此交集時結果是最基層型別

最後，一開始講述 Never 型別時，條目 4.5.1 有提到型別不可能發生的地方也會產出 Never 型別。所謂的型別不可能發生的地方是指——兩個型別性質本身互斥，只剩下的唯一共通點是 Never 型別時，該型別組合交集結果也會是 never，例如：

```
type NumberAndString = number & string;
```

數字跟字串壓根不可能會有共通點，唯一的共通點就只剩下 Never 型別。（以上程式碼推論結果如圖 4-25）

```
type NumberAndString = never  
type NumberAndString = number & string;
```

**型別本身的特性造成交集時共通點剩下 Never
型別時，推論結果自然而然也是 Never 型別**

圖 4-25 當兩個型別毫無共通點時，交集結果就會是 never

»» Never 型別與型別複合性質

1. **吸收律**：任何與 Never 型別聯集複合的型別 T，結果會是型別 T 吸收掉 Never 型別；相對地，任何與 Never 型別交集複合的型別 T，結果會是 Never 型別吸收掉型別 T。
2. **互斥律**：任意兩種不同型別，若性質本身無任何共通性（也就是互斥時），交集複合結果會是 Never 型別。

► 4.6 Any 與 Unknown 型別

4.6.1 所有型別的聯集——Any 型別

Any 型別是 TypeScript 型別系統裡，相較於 Never 型別，是所有型別的聯集結果。但這也代表 Any 型別由於代表的型別範圍太廣泛，通常會建議能夠避免出現 Any 型別就儘量避免。

從本書第三章看到現在，Any 型別出現的情境如下：

- 遲滯性指派（Delayed Initialization）（條目 3.2.3）
- 函式宣告時的參數型別（條目 3.4）
- 空陣列（條目 3.5.1）
- 會回傳 Any 型別結果的函式或方法，如 `JSON.parse`
- 直接註記變數或表達式為 Any 型別（不過這應該是廢話）

既然 Any 型別都已經跟讀者說是所有型別的聯集了，就表示：

任何型別與 Any 型別發生聯集就等於 Any 型別。

意思是說，任何型別跟 Any 型別聯集複合時：

```
type T = <Type> | any;
```

會被 Any 型別給吸收掉。(如圖 4-26)

```
type T = any;
```

```
type AbsoluteAny = any
type AbsoluteAny = number | any;
```

**任何東西跟 any 型別聯集時，
結果會當然會是 any 型別**

圖 4-26 Any 型別涵蓋全部型別，所以跟任何型別聯集就還是等於 Any 型別

然而，任何與 Any 型別交集的型別，儘管 Any 型別的範圍是整個型別系統的集合，輸出結果仍然為 Any 型別，這是作者實驗出來認為很詭異的現象¹⁵。

意思是說，任何（非 Unknown）型別跟 Any 型別交集複合時：

```
type T = <Non-Unknown-Type> & any;
```

會等效於以下的寫法。(如圖 4-27)

```
type T = any;
```

```
type NumberAndAny = any
type NumberAndAny = number & any;
```

**任何與 Any 交集的型別仍然是 Any，
這是作者認為很奇怪的現象。**

圖 4-27 任何與 Any 型別交集的型別仍然是 Any

15 參見 Stackoverflow：Type Intersection Using Any <https://stackoverflow.com/questions/46673558/type-intersections-using-any>。

4.6.2 安全版本的所有型別的聯集——Unknown 型別

Unknown 型別為 TypeScript 3.0 版釋出的內建型別，它很容易跟 Any 型別搞混（這是廢話），而這個型別的發明最主要是為了開發上能夠兼具 Any 型別的特性，但多了一層防護性的功能——凡是使用 Unknown 型別結果的變數或表達式，TypeScript 靜態分析時會強制開發者使用斷言語法¹⁶ 輔助型別檢測。

以下範例程式碼分別比較 Any 型別與 Unknown 型別的差異。（推論結果如圖 4-28）

```
let anyFoo: any = 123;
let unknownFoo: unknown = 123;

/* 使用 anyFoo 不會發生什麼事情 */
anyFoo + 1;

/* 使用 unknownFoo 就會出現警告訊息喔 */
unknownFoo + 1;
```

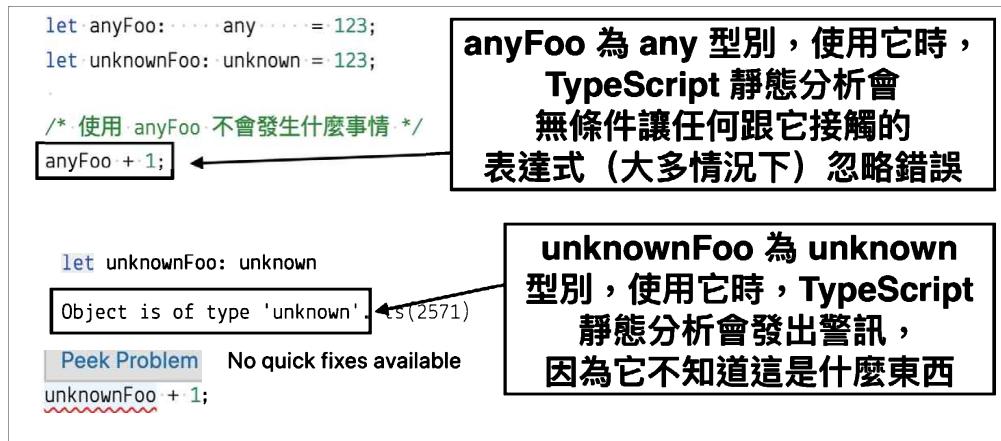


圖 4-28 Unknown 型別具有多一層提醒，告訴開發者必須要輔助性的註記

如果將出現警訊的那一行程式碼改成這樣就不會出現錯誤囉～

```
/* 使用 unknownFoo 時，搭配斷言告訴編譯器，靜態分析時就不會有出現警告訊息喔 */
(unknownFoo as number) + 1;
```

16 型別註記與斷言，請參見本書條目 2.2。

由於 Unknown 型別是建立在某變數是未知型別的前提下使用，因此凡是任何跟未知的東西聯集的結果，注定還是未知的結果——除了 Any 型別。請讀者小心 Any 型別這個東西，因為 Any 型別代表著整個型別系統，所以 Any 型別與 Unknown 型別的聯集會遵守條目 4.6.1 的規則——任何與 Any 型別聯集的型別結果會是 Any 型別，連 Unknown 型別也不例外。

意思是說，任何除了 Any 型別以外的型別跟 Unknown 型別聯集複合時：

```
type T = <Non-Any-Type> | unknown;
```

會變成未知狀態。(如圖 4-29)

```
type T = unknown;
```

```
type BecomesUnknown = unknown
type BecomesUnknown = number | unknown;
```

**任何非 any 型別與
unknown 聯集結果，最後為 unknown 型別**

圖 4-29 非 Any 型別與 Unknown 型別的聯集結果

但是跟 Any 型別聯集在一起逃不掉被 Any 型別吸收的命運——這也是合理的，畢竟一個已知是全部的型別系統和未知的特定型別做聯集時，未知的型別一定會藏身在已知的全部型別系統裡，所以結果當然就會是 Any 型別；因此以下這一行推論結果一定是 Any 型別。(如圖 4-30)

```
type T = any | unknown;
```

```
type AnyOrUnknown = any
type AnyOrUnknown = any | unknown;
```

**已知全部皆可能的型別以及特定的未知型別
聯集時，結果當然會是全部皆可能的型別**

圖 4-30 當 Any 型別與 Unknown 型別聯集時之結果

另外，任何與 Unknown 型別交集的型別，由於一個是未知的型別狀態，另一個是已知的型別狀態，共通點也只能是偏向於已知狀態型別的那一側；因此，

任何型別與 Unknown 型別交集時，結果一定會是前者吸收掉 Unknown 型別。

意思是說，任何型別跟 Unknown 型別交集複合時：

```
type T = <Type> & unknown;
```

會吸收掉 Unknown 型別的狀態。(如圖 4-31)

```
type T = <Type>;
```

```
type AbsolutelyNotUnknown = number
type AbsolutelyNotUnknown = number & unknown;
```

**任何與 Unknown 型別交集的型別，
會自動吸收掉 Unknown 型別**

圖 4-31 與 Unknown 型別交集時，結果會傾向於已知的型別狀態

作者多半建議，若讀者真的遇到非得使用 Any 型別的特性時，不妨使用看看 Unknown 型別，這樣多出了一層型別防護。(譬如以下的安全版本的 JSON.parse)

```
function safeJSONParse(jsonString: string): unknown {
    return JSON.parse(jsonString);
}
```

»» Any 型別與 Unknown 型別

1. Any 型別為整個 TypeScript 型別系統的聯集結果，也就是說，任何型別都可以被指派到 Any 型別的變數下。
2. Unknown 型別有別於 Any 型別，多了未知的意義，代表使用型別為 Unknown 的變數或表達式時，通常會提醒要用斷言語法輔助。

»» Any 型別的複合性質

1. 任何與 Any 型別聯集 (Union) 複合的型別 T 會被 Any 型別吸收，結果為 Any 型別。

2. 任何與 Any 型別交集 (Intersection) 複合的型別 T，且型別 T 不為 unknown 時，會被型別 T 吸收，結果為型別 T。
3. Any 型別與 Unknown 型別交集複合的結果為 Any 型別。

»» Unknown 型別的複合性質

1. 任何與 Unknown 型別聯集 (Union) 複合的型別 T，且型別 T 不為 any 時，會被 Unknown 型別吸收，結果為 Unknown 型別。
2. Unknown 型別與 Any 型別聯集複合的結果為 Any 型別。
3. 任何與 Unknown 型別交集 (Intersection) 複合的型別 T 會被型別 T 吸收，結果為型別 T。

► 本章練習

1. 元組 (Tuple) 與陣列 (Array) 有什麼差異性？
2. 親自實驗一下，以下各個變數之推論結果為何？

```
type TripleNumber = [number, number, number];
```

```
let ex1 = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

```
let ex2 = [
  [1, 2, 3] as TripleNumber,
  [4, 5, 6] as TripleNumber,
  [7, 8, 9] as TripleNumber
];
```

```
let ex3 = [
  [1, 2, 3] as TripleNumber,
  [4, 5, 6] as TripleNumber,
  [7, 8, 9] // ← 少一個斷言
];
```

```
];
```

```
type QuadrupleNumber = [number, number, number, number];
```

```
let ex4 = [
    [1, 2, 3] as TripleNumber,
    [4, 5, 6] as TripleNumber,
    [7, 8, 9, 10] as QuadrupleNumber
];
```

```
let ex5 = [
    [1, 2, 3], // ← 少一個斷言
    [4, 5, 6] as TripleNumber,
    [7, 8, 9, 10] as QuadrupleNumber
];
```

3. 承上題，以上的每個陣列，若根據推論結果，以最外層陣列為主，哪些是同質性陣列？哪些則是異質性陣列？（同質性與異質性陣列參見本書條目 3.5）
4. 請問以下各個變數的推論結果為何？

```
enum Pet { Dog, Cat, Bird, Duck };

let ex1 = Pet.Cat;
const ex2 = Pet.Cat;
let ex3 = Pet[Pet.Cat];
const ex4 = Pet[Pet.Cat];
let ex5 = Pet[2];
const ex6 = Pet[2];
```

5. 請問以下各個變數的推論結果為何？

```
enum Vehicle {
    Car = 'Car',
    Bicycle = 'Bicycle',
    Truck = 'Truck',
    Bus = 'Bus'
};

let ex1 = Vehicle.Truck;
const ex2 = Vehicle.Bus;
let ex3 = Vehicle[Vehicle.Car];
```

```
const ex4 = Vehicle[Vehicle.Bicycle];
```

6. 【進階題】根據條目 4.2.5 提到的列舉型別具備的映射性，其中，如果是以下的情形：

```
enum Vehicle {  
    Car = 'Car',  
    Bicycle = 'Bike',  
    Truck = 'Pickup Truck',  
    Bus = 'Coach'  
};
```

除了 Vehicle.Car 名稱與對應的值 'Car' 一模一樣，因此具備映射性外，其他的成員名稱與對應字串值皆不同，因此不具備映射性。

請思考原因為何？

7. 請問以下的寫法大概會被編譯成什麼樣的結果？列舉有無宣告成常數的差別在哪裡？

```
enum Pet {  
    Dog = 123,  
    Cat,  
    Bird = 'Bird',  
    Duck = 456  
};  
  
const enum Vehicle {  
    Car = 'Car',  
    Bicycle = 99,  
    Truck,  
    Bus = 'Coach'  
};  
  
const foo = Pet.Cat + Vehicle.Truck;  
console.log(Vehicle.Bicycle - Pet.Dog);
```

8. 如何宣告出物件的值（Value）只能為特定型別 T 的 JSON 物件型別？
9. 如何取得某 JSON 物件型別 { a: number; b: string; c: boolean } 裡，所有鍵的明文型別的聯集 'a' | 'b' | 'c' ？以及所有值的型別的聯集 number | string | boolean ？

10. 就讀者目前所學，將 JSON 物件變成唯讀狀態的方式有哪些？

11. 請問以下的型別化名，個別推論結果為何？

```
type Ex1 = number | string | boolean | undefined | null;
type Ex2 = number & string & boolean & undefined & null;
type Ex3 = undefined | null;
type Ex4 = undefined & null;
type Ex5 = number | 123;
type Ex6 = number & 123;
type Ex7 = never | number;
type Ex8 = never & string;
type Ex9 = any | number;
type Ex10 = any & string;
type Ex11 = unknown | number;
type Ex12 = unknown & string;
type Ex13 = never | any;
type Ex14 = never & any;
type Ex15 = never | unknown;
type Ex16 = never & unknown;
type Ex17 = any | unknown;
type Ex18 = any & unknown;
type Ex19 = never | any | unknown;
type Ex20 = never & any & unknown;
```

12. 若註記某變數為 Any 型別時，則任何型別的值或表達式都可以指派到該變數中？

```
let variable: any = /* 任意值或表達式 */;
```

13. 若註記某變數為 Never 型別時，則任何型別的變數或表達式都可以指派到該變數中？

```
let variable: never = /* 任意值或表達式 */;
```

14. 若註記某變數為 Unknown 型別時，則任何型別的變數或表達式都可以指派到該變數中？

```
let variable: unknown = /* 任意值或表達式 */;
```

05

TypeScript 類別基礎

相信讀者從第三、四章看到這裡，可能已經感受到型別系統的深度然後被震懾到了。本篇章要來講到另一門重點主題——類別（Class）。

► 5.1 物件導向基礎概論 OOP Fundamentals

5.1.1 什麼是物件？

相信讀者學到這裡，看到條目標題會翻白眼，不就是這種東西嗎？

```
{  
  name: 'Maxwell',  
  age: 18,  
  interest: ['drawing', 'programming']  
}
```

以上的程式碼確實是原生 JavaScript 的物件沒錯，有屬性也有相對應的值。

但是作者在本條目討論物件（Object）的概念與定義實質上會開始變得嚴格許多。

另外，多數 JavaScript 的開發者，可能有部分的人沒有特別區分「函式」（Function）與「方法」（Method）這兩樣東西——畢竟這也是從物件導向（Object-Oriented Programming）中的概念裡介紹的東西。

只是原生 JavaScript 為原型導向（Prototype-Based）語言，它並沒有物件導向裡的物件的概念；也就是說，JavaScript 的物件單純被定義為：

具備屬性（Property）以及對應的值（Value）的資料結構。

所以你才會看到 JSON 物件的寫法就是屬性與值一對一對的資料結構。

但是物件導向裡的物件的概念，確切的定義為：

內含變數、函式以及多種不同資料結構封裝（Encapsulate）成的一種可被操作的介面——就是物件。

沒有碰過物件導向的讀者一定會看不懂以上的那句話到底在講什麼，這沒什麼大不了，因為本章討論的東西會涵蓋上面那一句話的意思——本章節討論的物件會是嚴謹版本的物件——此物件非原生 JavaScript 物件，請看接下來的條目。

5.1.2 實體與建構子函式 Instance & Constructor Function

我們可以從一些簡單的案例體驗物件導向中，物件的概念。假設有一個虛構的物件：

```
const aCat = new Cat('Julia', 'Scottish Fold'); // ← P.S. 品種為蘇格蘭摺耳貓
```

如果讀者有熟讀前面的章節，條目 2.3.2 的物件型別簡介中就提過這一句話：「從類別（Class）建構出來的物件，它有一個專有名詞，曰：『實體（Instance）也』。」

以上的範例程式碼，變數 `aCat` 存的東西就是類別 `Cat` 建立出來的物件——也就是類別 `Cat` 的實體。

而 `new Cat(...)` 的寫法，其實就是將關鍵字 `new` 串上一個名為 `Cat` 的函式——這個函式就被稱之為建構子函式（Constructor Function）——為建構出類別

`Cat` 的實體的函式。此外，建構新的實體時，一定會先接上關鍵字 `new`。

由於建構子是函式，它當然可以也傳入參數；以上面的程式碼為範例的話，提供兩個字串型別的參數，分別為 '`Julia`' 與 '`Scottish Fold`' 這兩個值。建構式傳入參數的部分通常就是在設定實體的內容。

至於原生 JavaScript 是如何寫出建構子函式呢？基礎宣告寫法如下：

```
function Cat(name: string, breed: string) {
    this.name = name; // ← 貓咪的名字
    this.breed = breed; // ← 貓咪的品種

    return this;
}
```

不熟悉 `this` 的讀者，很遺憾地，儘管它並不是本書討論的重點，但它的 importance 會比你現在正在學的 TypeScript 還要來得重要，本篇條目也時不時會出現關鍵字 `this`，因此建議不熟的讀者也要釐清 `this` 關鍵字¹ 的概念，研讀下去可能會比較安心些。

回過頭來，上面的程式碼確實是用函式的方式宣告出 `Cat` 這個建構子。裡面的參數分別被指派到 `this.name` 以及 `this.breed` 裡面。這樣可以使得我們在一開始指派到變數 `aCat` 裡的實體值，具備兩個屬性（Property），一個就是 `name`，另一個就是 `breed`。

所以你可以使用 `console.log` 取得實體的屬性值：

```
console.log(aCat.name);
// => 'Julia'

console.log(aCat.breed);
// => 'Scottish Fold'
```

1 《 What's THIS in JavaScript? 》 By Kuro Hsu - <https://kuro.tw/posts/2017/10/12/What-is-THIS-in-JavaScript-%E4%B8%8A/>。

實體的屬性，在類別裡的名稱是成員變數（Member Variable），以下的行為就是竄改物件裡成員變數的值。

```
aCat.name = /* 任意字串值 */; // ← 修改物件屬性就是在修改物件的成員變數值
```

當然，我們也可以在建構子函式內部宣告函式，但寫法如下：

```
/**  
 * 請注意：以下是純 JavaScript 程式碼，TypeScript 雖然也可以跑，  
 * 但是會因為 this 被推論為 any 型別跳出警訊——此為 Polymorphic this 的  
 * 推論行為，由於是進階主題，作者也認為放在本書會太過細碎，有興趣可以參見2  
 */  
function Cat(name, breed) {  
    this.name = name; // ← 貓咪的名字  
    this.breed = breed; // ← 貓咪的品種  
  
    /* 介紹貓咪 */  
    this.introduction = function () {  
        return `This is ${this.name} and it belongs to ${this.breed}`;  
    }  
  
    return this;  
}
```

此時的建構出來的實體就會多出了 `introduction` 這個函式，但正確的說法是：「多出了 `introduction` 這個“方法”」；跟成員變數一樣，由於該方法為物件的成員之一，所以又被稱之為成員方法（Member Method）³。

以下是呼叫 `introduction` 這個成員方法的程式碼：

```
console.log(aCat.introduction());  
// => 'This is Julia and it belongs to Scottish Fold'
```

2 Polymorphic this 型別推論機制，請參見 <https://www.typescriptlang.org/docs/handbook/advanced-types.html#polymorphic-this-types>。

3 由於特殊名稱太多，所以進到類別正式的語法說明時會有正式的統整，現在還只是簡介階段。

至於為何實體的屬性會被稱作為成員“變數”的原因其實很簡單——竄改成員變數值，可能會因為這樣的變異（Mutate），呼叫方法的結果也就跟著被改變：

```
/* 名稱從 'Julia' 變異為 'Juliet' */
aCat.name = 'Juliet';

/* 輸出結果也跟著改變 */
console.log(aCat.introduction());
// => 'This is Juliet and it belongs to Scottish Fold'
```

簡而言之，實體就是有它的性質（也就是成員變數）以及它可以做出的行為（也就是呼叫該實體的方法，完整稱呼為成員方法）。

5.1.3 類別的宣告形式 Class Declaration

前面的範例這裡再放一次：

```
function Cat(name, breed) {
    this.name = name;    // ← 貓咪的名字
    this.breed = breed;  // ← 貓咪的品種

    /* 介紹貓咪 */
    this.introduction = function () {
        return `This is ${this.name} and it belongs to ${this.breed}`;
    }

    return this;
}
```

以上是 JavaScript 宣告一個物件導向中的類別（Class）的建構子函式的方式；而建構子函式的名稱 **Cat** 指的就是類別的名稱。而 ECMAScript 第六版標準已經有了基礎的類別語法，使得上面的程式碼等效於以下的寫法⁴：

4 這裡要先請讀者注意，我們還未進入到 TypeScript 的程式碼寫法，因此本範例是使用 ES6 的語法寫出來的程式碼，非 TypeScript 程式碼。

```
class Cat {  
    /* 此為建構子函式，作為初始化實體的地方 */  
    constructor(name, breed) {  
        /* 成員變數的初始化 */  
        this.name = name; // ← 貓咪的名字  
        this.breed = breed; // ← 貓咪的品種  
    }  
  
    /* 告成員方法 introduction */  
    introduction() {  
        return `This is ${this.name} and it belongs to ${this.breed}`;  
    }  
}
```

從以上的類別語法可以看出，宣告類別就是使用 `class` 關鍵字，建構子函式（Constructor Function）則是在類別裡宣告一個名為 `constructor` 的函式，裡面的內容是對實體的初始化（Initialization）行為，譬如設定成員變數的初始值。

成員方法則是另外宣告在類別裡。

5.1.4 物件導向的核心觀念

從條目 5.1.1 看到 5.1.3，可以開始點出物件的主要特性。

類別的宣告可以比擬成物件的藍圖（Blueprint），裡面描述著物件的性質、行為（也就是可以呼叫的方法）等；所以宣告類別 `Cat` 的過程就是在描述 `Cat` 的實體的性質與行為。

從類別建構出來的物件，也就是實體（Instance），具備動態（Dynamic）的特性，因為程式的執行過程中，我們可以任意篡改它的成員變數；有時候類別宣告出來的方法也會改變實體內部的狀態，因此呼叫方法也可能造成實體內部狀態的變異。

事實上，類別本身也是物件的一種，但類別由於是實體的藍圖，相較於容易異變的實體，程式執行過程中，類別相對就不容易被改變，因此類別具備靜態（Static）的特性。（如果藍圖變來變去的話，就應該不能叫做藍圖了，對吧？）

物件導向的核心觀念有三個：封裝（Encapsulation）、繼承（Inheritance）以及多型（Polymorphism）。

封裝的概念事實上很簡單，就是將整個物件的實作過程、細節、變數內容等隱藏在一個黑盒子的概念；唯一可以讓使用者做的事情就是經由類別建立物件實體，並且使用實體的屬性與方法，不需要去管內部的細節。（而實體的屬性與方法即是類別內部的成員變數與成員方法組成的，只是使用物件時不用管到這些細節）

繼承的概念則是將類別的屬性、方法等提供給被繼承者。譬如宣告類別 **Vehicle** 並且下面有不同的交通工具繼承此類別⁵：

```
/* 宣告類別 Vehicle */
class Vehicle {
    constructor(type, wheels) {
        this.type = type; // ← 交通工具種類
        this.wheels = wheels; // ← 輪胎數量
    }

    /* 交通工具發出鈴聲 */
    makeNoise() {
        console.log('Honk honk!');
    }
}

/* 車輛 Car 繼承 Vehicle */
class Car extends Vehicle {
    constructor(brand, color) {
        /* super 函式代表父類別的建構子函式，是為 Vehicle 的 constructor 函式 */
        super('Car', 4); // ← 車輛種類為 'Car' 並且擁有 4 個輪胎
    }
}
```

5 本範例採用的是 ES6 Class 的繼承語法作為範例，原生的 JavaScript 嚴格來說並沒有類別繼承的概念，但要達到相似的功能就得採用原型鍊繼承（Prototype Chain Inheritance）的語法，由於並非本書討論範圍，因此可以參見 MDN Doc: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain。

```
this.brand = brand; // ← 車輛有額外的屬性，這是車輛的品牌
this.color = color; // ← 車輛的顏色
}
}

/* 腳踏車 Bike 繼承 Vehicle */
class Bike extends Vehicle {
constructor(color) {
super('Bike', 2); // ← 車輛種類為 'Bike' 並且擁有 2 個輪胎
this.color = color; // ← 腳踏車額外的屬性，這是腳踏車的顏色
}

/* 覆寫掉 Vehicle 交通工具發出的鈴聲，改成腳踏車的叮叮聲 */
makeNoise() {
console.log('Ding ding!');
}
}
```

以上的範例除了主要的類別 `Vehicle` 外，額外宣告想兩個類別 `Car` 與 `Bike` 並且繼承主類別 `Vehicle` 的功能。

也就是說我們可以這樣使用範例宣告的類別：

```
/* 建立 Car 的實體 */
const car = new Car('Cadillac', 'Black'); // ← P.S. 黑色凱迪拉克

console.log(car.brand); // ← 此為 Car 類別宣告的屬性
// => 'Cadillac'
console.log(car.wheels); // ← 此為 Car 繼承自 Vehicle 的屬性
// => 4
car.makeNoise(); // ← 此為 Car 繼承自 Vehicle 的方法
// => 'Honk honk!'

/* 建立 Bike 的實體 */
const bike = new Bike('Blue');

console.log(bike.color); // ← 此為 Bike 類別宣告的屬性
// => 'Blue'
console.log(bike.type); // ← 此為 Bike 繼承自 Vehicle 的屬性
// => 'Bike'
```

```
bike.makeNoise();           // ← 此為 Bike 繼承自 Vehicle 的方法，但 Bike 覆寫掉了
// => 'Ding ding!'
```

以上大概就是繼承的效果。

最後，多型通常指最主要的兩種功能是——函式超載（Function Overloading）以及覆寫（Overwriting）：

函式超載是指宣告函式時，可以同時宣告不同版本的函式，譬如不同的型別、數量、順序的參數，就會執行不同程式效果的功能。

標準的寫法是無法直接在 JavaScript 展現的，作者只能用簡單的 C++ 語言的函式超載寫法給讀者稍微看看函式超載是長什麼樣子：

```
/* C++ 程式碼範例，宣告 rectangularArea 函式，專門計算長方形的面積 */

// 1. 輸入為兩個整數 integer，輸出為 integer
int rectangularArea(int width, int height) {
    return width * height;
}

// 2. 輸入為單個整數 integer，當成正方形來看待，輸出還是 integer
int rectangularArea(int size) {
    return size * size;
}

// 3. 輸入為一個整數 integer，另一個為浮點數 float，輸出變成 float
float rectangularArea(int width, float height) {
    // 將 width 從 int 轉態成 float 然後再相乘
    return (float)(width) * height;
}
```

以上的程式碼都是在宣告計算長方形面積，共通點是函式的名稱都是 `rectangularArea`，但輸入與對應輸出的形式完全不一樣，經由這樣函式超載的宣告過程，就可以在 C++ 裡這麼使用程式碼：

```
rectangularArea(3, 6); // ← 符合第一種宣告形式
// => 18
```

```
rectangularArea(6);      // ← 符合第二種宣告形式  
// => 36
```

```
rectangularArea(3, 6.0); // ← 符合第三種宣告形式  
// => 18.0
```

JavaScript 沒辦法按照以上的寫法做出函式超載的動作，但講解到 TypeScript 介面⁶時就有類似的超載宣告方式喔～

而覆寫的概念其實就在繼承的範例就已經有稍微展示過，從繼承的範例——也就是類別 `Bike` 覆寫掉繼承類別 `Vehicle` 的 `makeNoise` 方法，使得不同類別建構出來的實體，呼叫同一個方法時會展現不同的行為。

以上就是概略上瀏覽物件導向（Object-Oriented）的整體概念，接下來就是正式進入到 TypeScript 的類別相關語法解析。

》》 物件導向的基礎概念 Fundamentals of Object-Oriented Programming

兩大主角——類別與物件

類別（Class）為物件的藍圖，內部可能包含變數、函式與不同複雜資料結構的組合，主要為描述物件的規格；宣告過後，程式運行過程中由於不異被更動，所以具備靜態（Static）的性質。

物件（Object）則是擁有屬性與方法的資料型態，而物件又被稱作為建立該物件的類別的實體（Instance），具備高度異變性（Mutation），因此才會說具有動態（Dynamic）的性質；通常使用者是不會管物件內部是如何實作的，因為這是物件屬於的類別內部的實作過程，使用者只需要會操作物件提供的屬性與方法就好了。

6 介面的函式超載性宣告，請參見本書條目 6.2.3。

三大概念——封裝、繼承與多型。

封裝（Encapsulation）描述的是類別的實作意義：將詳細的實作內容密封在類別內部，外面的使用者不需要去管內部到底長什麼樣子，直接操作類別提供的屬性與方法。

繼承（Inheritance）描述的是類別的層級關係：可以將類別的實作規格傳遞給繼承的其他類別。

多型（Polymorphism）描述的是類別與物件的操作多樣性：不同類別的物件可以用相同名稱的規格達到不同種類的功能，以函式超載（Function Overloading）與覆寫（Overwriting）的方式達成。

► 5.2 TypeScript 類別語法 Class Syntax

5.2.1 類別的宣告 Class Declaration

事實上，TypeScript 的類別宣告跟 ECMAScript 標準的宣告方式沒有差多少，條目 5.1 在講述物件導向的基礎時就完整地將基礎宣告方式都講完了。

但如果結合讀者從本書第 3 章、第 4 章討論的型別推論與註記方面的知識時，無條件將以下的範例程式碼丟給 TypeScript 編譯器靜態分析，它可是會出現警吿訊息的。（如圖 5-1）

```
class Cat {  
    constructor(name, breed) {  
        this.name = name; // ← 貓咪的名字  
        this.breed = breed; // ← 貓咪的品種  
    }  
  
    introduction() {  
        return `This is ${this.name} and it belongs to ${this.breed}`;  
    }  
}
```

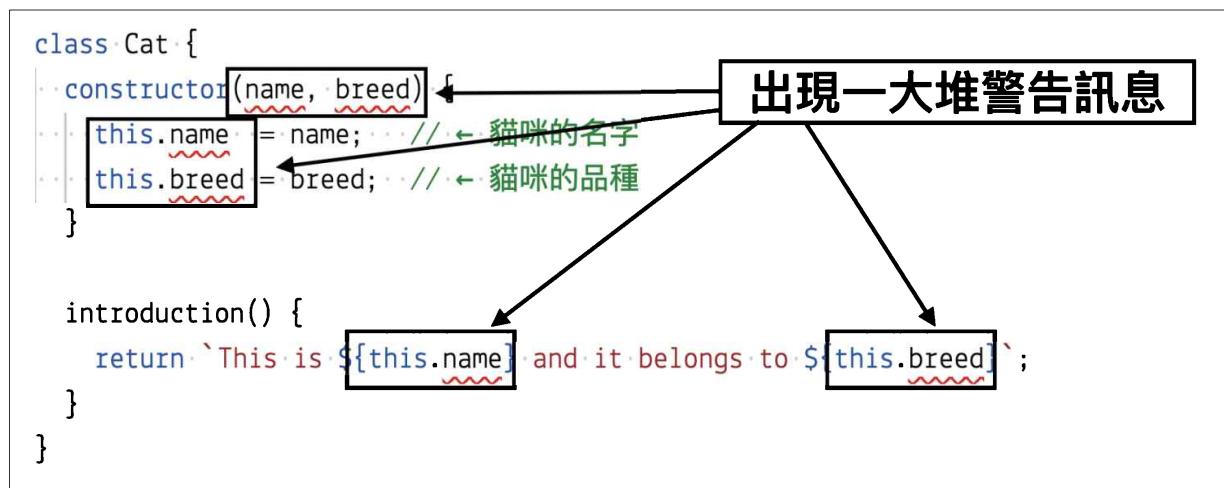


圖 5-1 原封不動將 ECMAScript 標準的類別宣告語法放到 TypeScript 分析環境

由於建構子函式 (Constructor Function) 也是函式⁷的一種，因此如果參數沒有被積極註記時，TypeScript 分析過程當然也不知道類別 `Cat` 裡的建構子函式之參數型別，一律就會被當成 Any 型別。

而一但被當成 Any 型別時，剩餘的推論過程就變得毫無意義啦，因此 TypeScript 就會發出警告，告訴開發者必須要積極註記建構子函式的參數型別。

但以下的程式碼仍然在 TypeScript 會出現錯誤訊息。(如圖 5-2)

```

class Cat {
  /* 建構子函式由於也是函式的宣告，因此參數必須要有對應的型別 */
  constructor(name: string, breed: string) {
    this.name = name; // ← 貓咪的名字
    this.breed = breed; // ← 貓咪的品種
  }

  // 略 ...
}

```

7 函式型別的推論與註記行為，請參見本書條目 3.4。

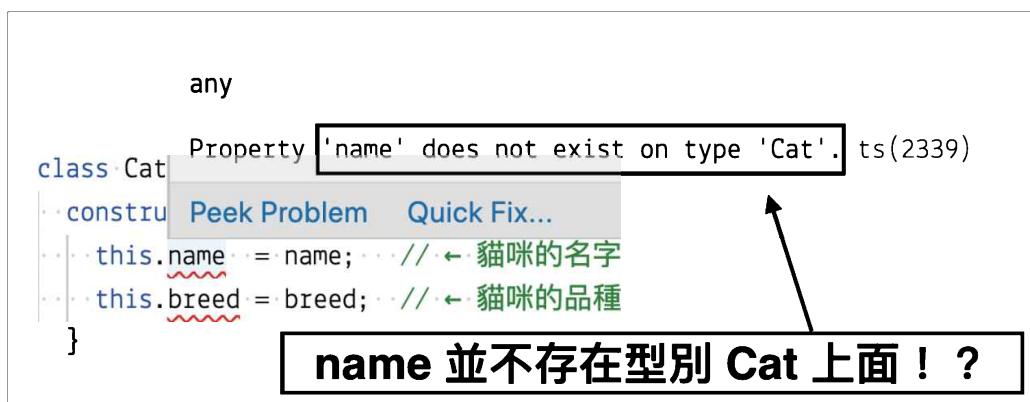


圖 5-2 屬性 name 不存在於型別 Cat 上

以下檢視一下錯誤訊息：

Property 'name' does not exist on type 'Cat'.

我們可以得知兩件事情：

1. 宣告類別的同時，就是在宣告新的型別（所以類別 Cat 是一種型別）。
2. 任何成員變數必須經過宣告才能夠使用。

第二條讀者可能不清楚，意思是說，初始化類別的成員變數時，必須先宣告其型別：

```

class Cat {
  /* 初始化成員變數前，必須先行宣告 */
  name: string;
  breed: string;

  constructor(name: string, breed: string) {
    this.name = name; // ← 貓咪的名字
    this.breed = breed; // ← 貓咪的品種
  }

  // 略 ...
}

```

以上的程式碼就不會出現錯誤訊息；TypeScript 這樣限制的目的不外乎是確保宣告類別時，開發者不會隨隨便便多出新的成員變數，必須經過型別宣告的步驟才得以使用。

關於建構子函式部分，條目 5.1.3 的最後有稍微提到，它的主要目的是初始化 (Initialize) 物件的狀態；因此，建構子函式不應該塞跟初始化過程無關的程式碼在裡面。所以以下的範例並不是好的寫法：

```
class Cat {  
    // 廣告成員變數 ... 略  
  
    /* 建構子函式裡面塞跟初始化無相關的程式碼 */  
    constructor(name: string, breed: string) {  
        // 略 ...  
  
        /* 隨機塞一些亂七八糟的東西 */  
        complexComputation(/* ... */);  
    }  
  
    // 略 ...  
}
```

就算要塞，至少作者建議在類別裡宣告名為 `initialize` 之類的方法名稱，並且在類別建構子裡自行呼叫。(讀者學到下個條目講到的存取修飾子時，作者會建議這裡的 `initialize` 的存取權限設定為 `private`，畢竟你不會希望使用者在外面亂呼叫這個方法)

```
class Cat {  
    // 略 ...  
  
    /* 建構子函式裡面塞跟初始化無相關的程式碼 */  
    constructor(name: string, breed: string) {  
        // 略 ...  
  
        /* 初始化時若遇到複雜的初始化過程，一樣整理到其他地方 */  
        this.initialize();  
    }  
  
    private initialize() {  
        complexComputation(/* ... */);  
    }  
  
    // 略 ...  
}
```

以上大概就是建構子函式必須要注意的地方。

接下來，作者節選建構子函式內部其中一行：

```
this.name = name; // ← 貓咪的名字
```

這一行代表我們在初始化類別的成員變數，此成員的名稱為 `name`。如果按照上面的類別範例，將實體建構出來時，我們也可以呼叫出類別裡的成員變數的值：

```
const aCat = new Cat('Julia', 'Scottish Fold');

aCat.name; // ← 呼叫出類別裡成員變數 name 的值
// => 'Julia'
```

但是讀者應該不會聽到「呼叫物件的成員變數」這種說法，而是在「呼叫物件的屬性（Property）」；類別的宣告過程中，裡面操作的東西名稱是成員變數，但建構出的物件，呼叫出的東西就不叫做成員變數，而是物件屬性。

兩者本身的性質儘管很相似，但名稱與代表意義是不同的。

成員變數除了可以在建構子函式初始化，你也可以放在建構子函式外面初始化：

```
class Cat {
    name: string;
    breed: string;

    /* 額外宣告另一個成員變數 noise，初始化並且指派字串型別的值 */
    noise: string = 'Meow meow meow ~';

    // 略 ...
}
```

不過要注意一件事情，如果你選擇不填入初始化的值時，以下的範例會出錯。
(錯誤訊息如圖 5-3)

```
class Cat {
    name: string;
```

```
breed: string;

/* 額外宣告另一個成員變數 noise，初始化為字串型別，但沒有提供值 */
noise: string;

// 略 ...
}
```

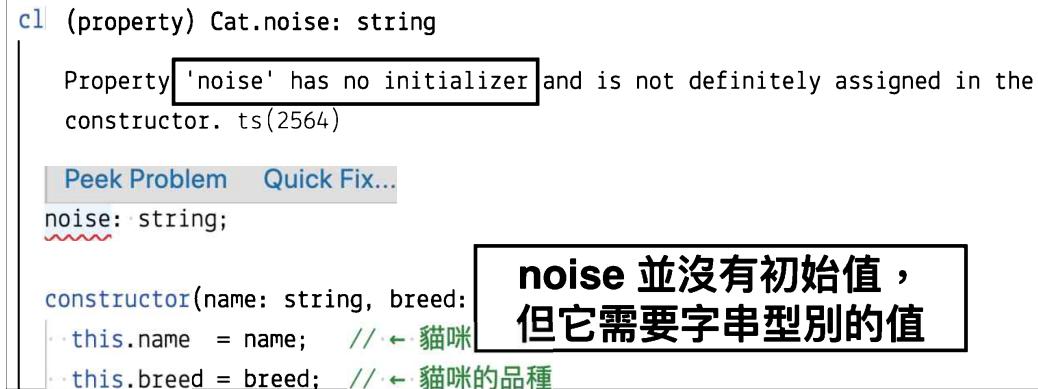


圖 5-3 成員變數 noise 必須要為字串值，但卻沒有初始值

除了最開始直接給它在宣告的同時指派字串值以外，還有兩種解套方式；若真的不需要初始化的值，可以為空值的話，就直接和 `undefined` 進行聯集複合，亦或者乾脆將它設定成選用成員（Optional Member）。

```
class Cat {
    name: string;
    breed: string;

    /* 方法 1. 初始化為選用字串型別 */
    noise?: string;

    // 略 ...
}
```

另一種就是在建構子函式內初始化值。

```
class Cat {
    name: string;
    breed: string;
```

```
noise: string;

constructor(name: string, breed: string) {
    // 略 ...

    /* 方法 2. 在建構子函式內初始化 */
    this.noise = 'Meow meow meow~';
}

// 略 ...
}
```

最後，方法的宣告其實沒有與 ECMAScript 標準差多少，只要記得函式型別篇章提到，參數一定要積極註記的原則就好了：

```
class Cat {
    // 略 ...

    feed(something: string) {
        console.log(`#${this.name} is eating ${something}...`);
    }
}
```

以上是宣告 TypeScript 類別時最基礎的語法介紹。

»» 類別的宣告 Class Declaration

1. 類別的宣告形同宣告一個新的型別。
2. 類別宣告過程，總共有三大部分要注意：成員變數的宣告、初始化過程與成員方法的宣告。
3. 初始化的過程一律都是使用建構子函式，也就是名為 constructor 的函式。
4. 在使用任何成員變數前，一定要先行宣告。
5. 假設宣告類別 C，其成員變數為 V₁、V₂ ... 、V_N，對應型別為 T₁、T₂ ... 、T_N，成員方法則為 M₁、M₂ ... 、M_N，則語法如下：

```
class C {
    V1: T1;
    V2: T2;
```

```
// ...
VN: TN;

constructor(* 建構子函式的參數 *) {
    /* 初始化流程 */
}

M1(* M1 的參數 *) { /* ... */ }
M2(* M2 的參數 *) { /* ... */ }
// ...
MN(* MN 的參數 *) { /* ... */ }
}
```

請記得，如果成員變數的宣告過程，沒有初始化型別對應的值就會出現警告訊息。

除了將成員變數對應的型別與 undefined 進行聯集外，否則宣告的同時亦或者是在建構子函式內指派初始值。

5.2.2 存取修飾子 Access Modifiers

前一個條目完整交代了成員變數與方法的宣告與使用過程，但並不代表每個成員都要開放給外面的人使用，這時候就需要一個名為存取修飾子（Access Modifiers）的東西。

存取修飾子分成：public 、private 以及 protected 三種模式。

- public 模式：不管類別內部還是使用者建立的物件都可以使用的模式
- private 模式：僅限當前類別內部使用的成員
- protected 模式：僅限當前類別以及繼承此類別的其他類別使用的成員

至於為何讀者會需要此功能呢？譬如有一個簡單的提款機 CashMachine 類別，裡面除了有基本的存提款功能外，還有簡單的帳戶確認功能：

```
type UserAccount = {
    username: string;
```

```
password: string;
savings: number;
};

const Users: UserAccount[] = [
  { username: 'Maxwell', password: 'HA$HED', savings: 10000 },
  { username: 'Martin',  password: 'hA$HEd', savings: 20000 },
  { username: 'Mike',    password: 'haSHeD', savings: 30000 },
];

class CashMachine {
  // 宣告 currentUser 為目前使用提款機的使用者，由於可能為空，
  // 因此使用選用型別，代表無使用者狀態，效果跟 undefined 聯集複合差不多
  currentUser?: UserAccount;

  // 登入使用者時，檢查有沒有使用者，若沒有就丟出錯誤訊息
  login(username: string, password: string) {
    for (let i = 0; i < Users.length; i += 1) {
      const user = Users[i];

      // 比對使用者資料，找到使用者時，將結果指派到成員變數並跳出函式
      if (user.username === username) {
        if (user.password !== password) {
          throw new Error('Wrong password!');
        }

        this.currentUser = user;
        return;
      }
    }

    // 還沒跳出函式就代表使用者沒辦法對應到
    throw new Error('User not found!');
  }

  /* 登出功能 */
  logout() {
    this.currentUser = undefined;
  }
}
```

```
/* 存款功能 */
deposit(amount: number) {
    if (this.currentUser === undefined) {
        throw new Error('User haven\'t logged in!');
    }

    this.currentUser.savings += amount;
}

/* 提款功能，別忘了要檢查存款是不是足夠 */
withdraw(amount: number) {
    if (this.currentUser === undefined) {
        throw new Error('User haven\'t logged in!');
    }

    if (this.currentUser.savings < amount) {
        throw new Error('Savings isn\'t enough for withdrawal!');
    }

    this.currentUser.savings -= amount;
}
}
```

雖然說本範例還可以改進的地方很多，不過這裡只是作為示範用途。以下是使用提款機類別的過程：

```
// 初始化 CashMachine 實體
const machine = new CashMachine();

// 登入使用者
machine.login('Maxwell', 'HA$HED');

// 存入金額
machine.deposit(1000);

// 查看目前的帳戶狀態，這裡直接使用斷言的目的在於，machine.currentUser
// 可能為 undefined，但是因為作者很確定這會是 UserAccount 所以才會主動註記它
console.log((machine.currentUser as UserAccount).savings);
// => 11000
```

```
// 登出使用者  
machine.logout();
```

不曉得使用者有沒有注意到這一行：

```
console.log((machine.currentUser as UserAccount).savings);
```

我們可以直接使用類別裡的成員變數查詢目前的使用者的存款狀態，但這也代表我們可以在外面亂竄改使用者的資料，也不會出現任何錯誤訊息。

```
if (machine.currentUser) {  
    machine.currentUser.password = '1234567890';  
    machine.currentUser.savings = 1000000000;  
}
```

這似乎是不樂見的結果，因此我們有必要將提款機內部的成員變數的權限給封住，這時候就是存取修飾子派上用場，以下將成員變數 `currentUser` 設定為私有狀態，也就是 `private` 模式。

```
class CashMachine {  
    // 將 currentUser 的權限設定為私有狀態  
    private currentUser?: UserAccount;  
  
    // 略 ...  
}
```

這麼一來，外面的使用者根本無權取用 `currentUser` 這個屬性。(如圖 5-4)

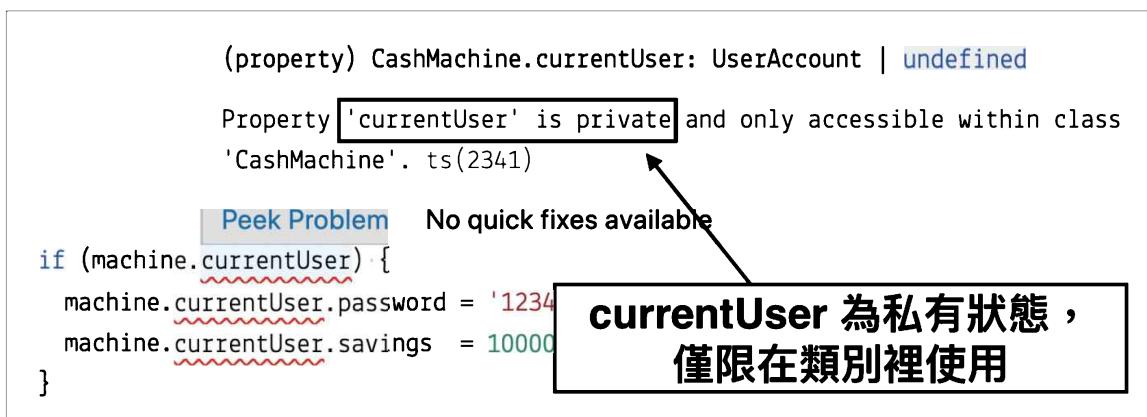


圖 5-4 `currentUser` 為私有狀態時，外面的程式碼就不能亂取用

從以上的範例展示，類別裡宣告的所有成員們，預設都是開放的狀態，也就是 **public** 模式，所以以前一條目的 **Cat** 類別為例，它的宣告方式為：

```
class Cat {  
    name: string;  
    breed: string;  
    noise: string = 'Meow meow meow ~';  
  
    constructor(name: string, breed: string) {  
        this.name = name;  
        this.breed = breed;  
    }  
  
    introduction() {  
        return `This is ${this.name} and it belongs to ${this.breed}`;  
    }  
  
    feed(something: string) {  
        console.log(`${this.name} is eating ${something}...`);  
    }  
}
```

就跟下面的寫法是等效的：

```
class Cat {  
    /* 所有成員預設都是 public 模式 */  
    public name: string;  
    public breed: string;  
    public noise: string = 'Meow meow meow ~';  
  
    constructor(name: string, breed: string) {  
        this.name = name;  
        this.breed = breed;  
    }  
  
    /* 成員方法也可以設定存取權限喔！ */  
    public introduction() {  
        return `This is ${this.name} and it belongs to ${this.breed}`;  
    }  
  
    public feed(something: string) {
```

```
    console.log(` ${this.name} is eating ${something}...`);  
}  
}
```

另外，建構子函式內的初始化過程，由於是直接將參數的值指派到成員變數中，它還有精簡的寫法：

```
class Cat {  
    public noise: string = 'Meow meow meow ~';  
  
    /**  
     * 同時宣告 name 與 breed 為成員變數外，建構時的第一個與第二個參數  
     * 分別為 name 與 breed 初始化的值  
     */  
    constructor(  
        public name: string,  
        public breed: string  
    ) {}  
  
    // 略 ...  
}
```

以上的寫法是不是更簡單？

另外，如果搭配函式參數預設值⁸，除了可以同時宣告成員變數外，還可以再建構子函式內的參數宣告部分預設初始化的值。

```
class Cat {  
    /* 同時宣告 noise 為成員變數並且作為建構子函式的參數外，也預設它的初始值 */  
    constructor(  
        public name: string,  
        public breed: string,  
        public noise: string = 'Meow meow meow ~'  
    ) {}  
  
    // 略 ...  
}
```

8 預設參數值的行為，請參見本書條目 3.4.4。

所以根據以上的範例程式碼，要建構 `Cat` 實體有兩種方式：

```
// 使用預設的貓叫聲
const cat1 = new Cat('Julia', 'Scottish Fold');

// 初始化不同的貓叫聲
const cat2 = new Cat('Irene', 'Ocicat', 'MEOwwwWWWW!'); // ← P.S. 品種為
歐西貓
```

最後，目前還沒出現過保護模式，也就是 `protected` 模式；不過它跟 `private` 模式差別就差在繼承的部分。簡而言之，類別內的成員若為保護模式時，繼承此類別的其他類別也可以使用這些保護模式下的成員⁹。

»» 存取修飾子 Access Modifiers

1. 存取修飾子主要在控制類別成員的使用權限。
2. 分成三種模式：開放 `public`、私有 `private` 以及保護 `protected` 模式。
3. 若類別內的成員沒有指定任何存取模式，一切都預設為開放模式，代表該類別成員可以在類別內部與外面被使用。
4. 若僅限定某類別成員在類別內使用時，可以標注為私有模式。
5. 若僅限定某類別成員在類別內或繼承該類別的其他類別使用時，可以標注為保護模式。
6. 若要標示存取修飾子在某類別 C 的成員時，寫法如下：

```
class C {
    /* 宣告成員變數 */
    <public | private | protected> V: T;

    constructor(/* 建構子函式的參數 */) {
        /* 初始化流程 */
    }
}
```

⁹ 保護模式會在類別繼承部分詳細解說，請參見本書條目 5.2.6。

```
/* 告成員方法 */
<public | private | protected> M(/* M 的參數 */) { /* ... */ }
}
```

若建構子函式有參數可以直接初始化成員變數時，可以將寫法簡化如下：

```
class C {
  constructor(
    /* 告成員函式的參數同時，宣告類別的成員變數 */
    <public | private | protected> V: T;

    // 其餘建構子函式的參數 ...
  ) {/* 初始化流程 */}

  <public | private | protected> M(/* M 的參數 */) { /* ... */ }
}
```

5.2.3 存取方法 Accessors

另外一個很容易因為名稱與存取修飾子很相近而搞混的是存取方法（Accessors）。

作者以前一篇章節講述到的提款機類別 `CashMachine` 的案例延伸下去。為了防止外面的使用者竄改提款機的成員變數 `currentUser` 的資料，因此就得將 `currentUser` 設定為私有狀態（Private）。

因此如果想要讓使用者可以查詢目前的存款狀態時，可以額外宣告方法讓使用者可以檢視：

```
class CashMachine {
  private currentUser?: UserAccount;

  getAccountInfo() {
    // 記得要用條目 4.4.2 教過的型別駐防（Type Guard）處理聯集型別的不同情形
    if (this.currentUser === undefined) {
      return 'Current user hasn\'t logged in!';
    }
  }
}
```

```
// 使用 ES6 解構語法將 username 與 savings 兩個屬性的值拔出來
const { username, savings } = this.currentUser;
return `${username} has savings: \$${savings}`;
}

// 略 ...
}
```

這樣就可以這樣使用我們的提款機檢視使用者的存款狀態：

```
const machine = new CashMachine();
machine.login('Maxwell', 'HA$HED');
machine.deposit(1000);

// 查詢狀態
console.log(machine.getAccountInfo());
// => 'Maxwell has savings: $11000'

// 登出使用者
machine.logout();

// 再次查詢狀態
console.log(machine.getAccountInfo());
// => 'Current user hasn\'t logged in!'
```

不過如果能夠將查詢狀態的成員方法 `getAccountInfo` 設計得像物件的屬性 `accountInfo` 而非方法的話，是不是感覺會自然許多？

```
// 呼叫方法的方式查詢狀態
console.log( machine.getAccountInfo() );

// 呼叫屬性的方式查詢狀態
console.log( machine.accountInfo ); // ← 呼叫屬性比起呼叫方法感覺似乎乾淨些
```

類別有提供一個功能叫做取值方法（Getter Methods），可以動態模擬物件屬性：

```
class CashMachine {
    private currentUser?: UserAccount;
```

```
/**
 * 將 getAccountInfo 取代為取值方法 accountInfo，使得建構出的實體
 * 可以動態模擬出 accountInfo 屬性
 */
get accountInfo() { /* 略... */ }

// 略...
}
```

取值方法宣告方式很簡單，就是在成員方法前面標上 **get** 關鍵字。

經由取值方法的宣告，程式碼的使用過程會變得非常簡潔喔～

```
const machine = new CashMachine();
machine.login('Maxwell', 'HA$HED');
machine.deposit(1000);

// 物件可以使用 accountInfo 這種動態屬性來查詢狀態
console.log(machine.accountInfo);
// => 'Maxwell has savings: $11000'

// 登出使用者
machine.logout();

// 再次查詢狀態，屬性結果被改變
console.log(machine.accountInfo);
// => 'Current user hasn\'t logged in!'
```

由於取值方法是在模擬物件的屬性，而呼叫屬性時不可能像函式一樣代入任何參數，因此取值方法有任何參數是禁止的行為。(如圖 5-5)

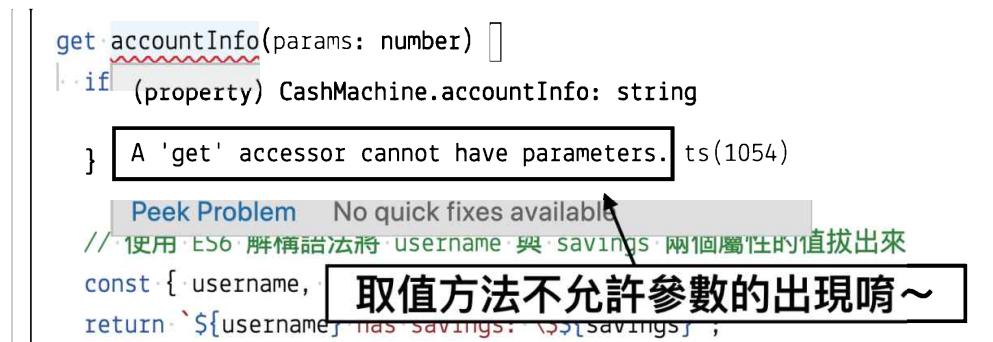


圖 5-5 取值方法出現參數時出現的錯誤訊息

另外，既然有取值方法，就會有對應的存值方法（Setter Methods）。

比如說想要更改帳戶的使用者名稱，但是因為 `currentUser` 被設定為私有狀態，所以這樣暴力改帳戶名稱是不行的：

```
machine.currentUser.username = 'Maximilian';
```

一種解法當然就是新增一個名為 `setUsername` 的成員方法供外面使用：

```
class CashMachine {  
    private currentUser?: UserAccount;  
  
    setUsername(input: string) {  
        if (this.currentUser === undefined) {  
            throw new Error('Current user isn\'t logged in!');  
        }  
  
        // 類別裡就可以取用 currentUser  
        this.currentUser.username = input;  
    }  
  
    // 略 ...  
}
```

這樣使用者要在外面更改帳戶名稱時，可以這麼做：

```
const machine = new CashMachine();  
machine.login('Maxwell', 'HA$HED');  
  
console.log(machine.accountInfo);  
// => 'Maxwell has savings: $10000'  
  
// 更改使用者名稱  
machine.setUsername('Maximilian');  
  
// 再次查詢狀態，屬性結果被改變  
console.log(machine.accountInfo);  
// => 'Maximilian has savings: $10000'
```

如果想要模擬覆寫物件屬性的行為，可以採用存值方法的宣告方式：

```

class CashMachine {
    private currentUser?: UserAccount;

    /**
     * 將 setUsername 取代為存值方法 username，使得建構出的實體
     * 可以動態模擬出 username 屬性
     */
    set username(input: string) { /* 略 ... */ }

    // 略 ...
}

```

所以以下這一行：

```
machine.setUsername('Maximilian');
```

就可以被改寫為：

```
machine.username = 'Maximilian';
```

是不是像極了操作普通 JSON 物件的感覺，只是黑盒子裡面是複雜的類別藍圖架構。

另外，由於存值方法模擬的物件屬性的覆寫指派行為，只會接收到指派的值，所以存值方法一定要出現、但只能出現一個參數，如果沒有參數或太多參數就會出現錯誤訊息喔。(如圖 5-6)

```

    set username(input: string, tooManyParams: number) {
        if (property) CashMachine.username: string
    } A 'set' accessor must have exactly one parameter. ts(1049)
    Peek Problem No quick fixes available
    // 類別裡就可以取用 currentUser
    this.currentUser.use
}

```

存值方法只能接收一個參數！

圖 5-6 存值方法只能接收一個參數，如果不是就會出現圖中的錯誤訊息

另外，如果你想要模擬物件的唯讀屬性（Readonly Property），最常見的寫法就是只取值方法，但不宣告存值方法。

以上的存值方法與取值方法，合稱存取方法（Accessors），不過不同的物件導向語言會用不同的名稱，你可能會看到：

- 取值方法儘管在 JavaScript 被稱為 Getter Methods，但可能在某些物件導向語言會將取值方法直接用 Accessors 稱呼。
- 存值方法在 JavaScript 裡被稱為 Setter Methods，但相對於某部分語言果斷將取值方法稱為 Accessors，相對應地存值方法就被稱作為 Modifiers（亦或者是 Mutators），還有 Modifier Methods（Mutator Methods）。

對，存取方法的英文除了跟存取修飾子很容易搞混之外，連使用的語言不同，名稱也會指涉不同的東西。舉例來說，使用的是 JavaScript 或 TypeScript，Accessors 就同時指得是存值與取值方法這兩種；但如果是 Java 這門語言時，存值方法的英文是 Accessors，取值方法則是 Mutators。

»» 存取方法 Accessors

1. 存取方法主要是動態模擬物件屬性的存值與取值行為。
2. 取值方法（Getter Methods）的宣告，就是在類別成員方法前面標上 get 關鍵字，並且該方法的宣告不能有任何參數。
3. 存值方法（Setter Methods）的宣告，就是在類別成員方法前面標上 set 關鍵字，並且該方法的宣告一定要有一個參數。
4. 存取方法可以被標上存取修飾子（Access Modifiers），意思是說存取方法可以被設定為開放、私有與保護模式。

5.2.4 靜態成員 Static Members

看到本條目的標題中出現「靜態」這兩個字，想必讀者心裡已經有一個底了。

靜態成員到底是什麼？相對於靜態，所謂的“動態”成員存不存在？事實上從條目 5.2.1 討論到 5.2.3，我們接觸並且宣告的類別成員都是動態的成員，因為這些類別裡的成員變數、方法等，都是實體會間接接觸到、變異到的東西。

靜態成員就是類別（Class）本身的成員，所以如果你聽到類別屬性（Class

Property) 與類別方法 (Class Methods) 等名稱多半指的是靜態成員，主角從物件本身轉移到類別（也就是創建實體的藍圖）。況且條目 5.1.4 都已經強調了，由於類別在程式運行中被動到的機率很低，相對於實體而言，具備靜態的屬性，所以被取名為靜態成員應該可以理解。

以提款機類別 `CashMachine` 為例，除了提款機外，還有一個很陽春的使用者資料庫 `Users`。事實上我們可以選擇將這些資料納入類別裡面，將其封裝進去，並且設定為私有狀態，因為你也不會希望外來人亂動裡面的資料：

```
class CashMachine {  
    private currentUser?: UserAccount;  
    private Users: UserAccount[] = [  
        { username: 'Maxwell', password: 'HA$HED', savings: 10000 },  
        { username: 'Martin', password: 'hA$HEd', savings: 20000 },  
        { username: 'Mike', password: 'haSHeD', savings: 30000 },  
    ];  
  
    // 略 ...  
}
```

由於資料庫變成類別裡的成員變數，因此登入的函式 `login` 必須改寫為：

```
class CashMachine {  
    // 略 ...  
  
    // 原本只有 Users 必須改成用 this.Users 的寫法  
    login(username: string, password: string) {  
        for (let i = 0; i < this.Users.length; i += 1) {  
            const user = this.Users[i];  
  
            // 略 ...  
        }  
  
        // 還沒跳出函式就代表使用者沒辦法對應到  
        throw new Error('User not found!');  
    }  
  
    // 略 ...  
}
```

首先，如果讀者改寫成以上的程式碼時，每一次建立一個 `CashMachine` 這個實體時，都會重複建立 `Users` 這個成員變數對應的物件：

```
const machine1 = new CashMachine(); // 建立整個 this.Users 物件
const machine2 = new CashMachine(); // 再次建立整個 this.Users 物件
const machine3 = new CashMachine(); // ... 一直不停的建立 this.Users 物件
const machine4 = new CashMachine();
const machine5 = new CashMachine();
```

另外，如果讀者對於物件的參照（Reference）方式傳遞行為很熟的話，上面 `CashMachine` 的宣告，使用時會出現嚴重錯誤。

```
const machine1 = new CashMachine(); // 建立整個 this.Users 物件
const machine2 = new CashMachine(); // 再次建立整個 this.Users 物件

/* 第一個提款機，存 $1000 進到 Maxwell 這個帳戶 */
machine1.login('Maxwell', 'HA$HED');
machine1.deposit(1000);
console.log(machine1.accountInfo);
// => 'Maxwell has savings: $11000'

machine1.logout();

/* 進到第二個提款機，結果查詢 Maxwell 這個帳戶是未存款前的狀態 */
machine2.login('Maxwell', 'HA$HED');
console.log(machine2.accountInfo);
// => 'Maxwell has savings: $10000'
```

因為每一次從 `CashMachine` 建立出新的實體時，`Users` 這個成員變數的值會重複建構一次，使得每一次建構的新的提款機實體會是最初版本的資料型態。

為了要避免每一次建構新的實體時，整個物件被重新建構，我們可以將 `Users` 這個成員變數改成靜態的類別成員：

```
class CashMachine {
    private currentUser?: UserAccount;

    /* 使用 static 關鍵字宣告靜態成員 */
    static Users: UserAccount[] = [
        { username: 'Maxwell', password: 'HA$HED', savings: 10000 },
```

```
{ username: 'Martin', password: 'hA$HEd', savings: 20000 },
{ username: 'Mike', password: 'haSHeD', savings: 30000 },
];
// 略 ...
}
```

由於 `Users` 變成靜態成員了，以前在類別裡的 `this.Users` 的寫法就會出現錯誤，因為那是普通成員變數或方法的寫法。那這時候要改寫成什麼樣子呢？

記得，靜態成員是指類別本身的成員，也就是說，就是用類別 `CashMachine` 去呼叫該成員：

```
class CashMachine {
    // 略 ...

    // this.Users 必須改成用 CashMachine.Users 的寫法，因為是在呼叫類別本身的成員
    login(username: string, password: string) {
        for (let i = 0; i < CashMachine.Users.length; i += 1) {
            const user = CashMachine.Users[i];

            // 略 ...
        }

        // 還沒跳出函式就代表使用者沒辦法對應到
        throw new Error('User not found!');
    }

    // 略 ...
}
```

這樣的話，以下的範例程式碼就會正常運作囉！

```
const machine1 = new CashMachine(); // 建立整個 this.Users 物件
const machine2 = new CashMachine(); // 再次建立整個 this.Users 物件

/* 第一個提款機，存 $1000 進到 Maxwell 這個帳戶 */
machine1.login('Maxwell', 'HA$HED');
machine1.deposit(1000);
console.log(machine1.accountInfo);
```

```
// => 'Maxwell has savings: $11000'

machine1.logout();

/* 進到第二個提款機，結果查詢 Maxwell 這個帳戶是存款過後的狀態 */
machine2.login('Maxwell', 'HA$HED');
console.log(machine2.accountInfo);
// => 'Maxwell has savings: $11000'
```

另外，美中不足的地方是在於該靜態成員依然可以在類別外面被使用，最後再補上條目 5.2.2 講到的存取修飾子，調整為私有狀態就行囉～

```
class CashMachine {
    private currentUser?: UserAccount;

    /* 使用 private static 關鍵字宣告私有靜態成員 */
    private static Users: UserAccount[] = [ /* 略 ... */ ];

    // 略 ...
}
```

另外，靜態成員也可能被應用在設計一系列的工具介面，譬如我們常在原生 JavaScript 看到的 `Math` 物件就是一種案例。想要模擬 `Math` 物件的設計方式是可以這麼寫的，以 `Math.pow` 和 `Math.abs` 方法為例。

```
class MyMath {
    /* Math.pow 為指數運算的函式 */
    static pow(base: number, exponent: number) {
        return base ** exponent;
    }

    /* Math.abs 為絕對值運算的函式 並不是指腹肌的英文 */
    static abs(input: number) {
        return input >= 0 ? input : -input;
    }
}
```

以上的宣告方式就可以這樣使用囉～

```
console.log(MyMath.pow(2, 4));
// => 16
```

```
console.log(MyMath.abs(-1));
// => 1
```

»» 靜態成員 Static Members

1. 靜態成員主要為類別本身的屬性與方法。
2. 靜態成員的宣告必須使用關鍵字 `static`。
3. 靜態成員也可以標上存取修飾子（Access Modifiers），使得該靜態成員的模式為開放、私有或者是保護狀態。

5.2.5 唯讀成員 Read-Only Members

條目 3.3.5 有針對 JSON 物件的唯讀屬性（Read-Only Properties）的設置，理所當然地，在類別裡也會有相對應的唯讀成員喔。

因此假設我們將之前的範例類別 `Cat` 中的成員變數，標注上 `readonly` 屬性時，也可以將其設定為唯讀屬性的成員：

```
class Cat {
  constructor(
    public readonly name: string,
    public readonly breed: string
  ) {}

  // 略 ...
}
```

以下的行為，將值指派並覆寫成員變數的行為就會被禁止。（如圖 5-7）

```
const cat = new Cat('Julia', 'Scottish Fold');
cat.name = 'Juliet';
(property) Cat.name: string
Cannot assign to 'name' because it is a read-only property. ts(2540)
Peek Problem No quick fixes available
```

由於 `name` 為唯讀屬性，因此不能夠覆寫掉

圖 5-7 不能夠覆寫掉唯讀屬性的成員喔

另外，靜態成員屬性，也就是類別屬性也可以標注上唯讀屬性操作符，就是在 `static` 關鍵字旁邊接上 `readonly` 這個操作符就可以囉。

»» 唯讀成員變數 Read-Only Member Variables

1. 類別宣告成員變數時，可以附上唯讀屬性操作符 `readonly` 使該成員被設定為唯讀狀態。
2. 類別靜態屬性也可以附上唯讀屬性操作符。

5.2.6 類別繼承 Class Inheritance

繼承的概念在條目 5.1.4 稍微提過，任何類別都可以被其他類別繼承，繼承的類別就擁有被繼承類別的所有開放（Public）或保護（Protected）模式狀態下的成員。

以下就用之前有的範例程式碼為主，宣告交通工具類別 `Vehicle` 如下：

```
class Vehicle {  
    constructor(  
        private type: string,  
        private wheels: number,  
    ) {}  
  
    get info() {  
        return `Type: ${this.type}; Wheels: ${this.wheels}`;  
    }  
  
    public makeNoise() {  
        console.log('Honk honk!');  
    }  
}
```

相信經過前面條目的說明，應該可以看得懂以上的程式碼。類別 `Vehicle` 宣告兩個私有模式下的成員變數 `type` 以及 `wheels`，以及開放狀態下的成員方法 `makeNoise`，並且使用取值方法讓使用者間接知道該交通工具的資訊。

首先，作者希望做到的功能是，從主要的類別 `Vehicle` 延伸出各種不同的交通

工具，使得每一個交通工具具備類別 **Vehicle** 的性質但是不需要再重複定義一遍。

第一步就是繼承的語法：

```
class Car extends Vehicle {}
```

類別 **Car** 與它的父類別（Parent Class）**Vehicle** 使用時會有一模一樣的效果：

```
const vehicle = new Vehicle('Car', 4); // 建立新的 Vehicle 實體
const car      = new Car('Car', 4);      // 建立新的 Car 實體，並且填入同樣的參數

/* 由於 Car 繼承自 Vehicle，剛繼承時，兩者使用效果會一模一樣 */
console.log(vehicle.info);
// => 'Type: Car; Wheels: 4'

console.log(car.info);
// => 'Type: Car; Wheels: 4'

vehicle.makeNoise();
// => 'Honk honk!'

car.makeNoise();
// => 'Honk honk!'
```

另外，英文裡除了稱 **Vehicle** 為類別 **Car** 的父類別，也就是 Parent Class，亦或者用 Superclass 代表；相對地，類別 **Car** 為 **Vehicle** 的子類別，也就是 Child Class，亦或者用 Subclass 代表。

不過有些讀者可能看到這一行，應該會覺得程式碼有冗贅的感覺：

```
const car = new Car('Car', 4); // 建立新的 Car 實體，並且填入同樣的參數
```

宣告 **Car** 的實體時，類別名稱本身和成員變數 **type** 重複了；另外，一般車輛大多數是 4 個輪胎，所以應該可以在類別建構實體的初始化（Initialization）過程中自動填入這些參數的。

相信讀者看到「初始化」這三個字腦袋就會直接連結建構子函式，也就是 **constructor** 函式。但是以下的寫法是錯誤的喔！

```
class Car extends Vehicle {  
    constructor(  
        private type: string = 'Car',  
        private wheels: number = 4,  
    ) {}  
}
```

而且以上的寫法很不合理的地方在於，它重複跟父類別的宣告同樣的成員變數。

但如果換成這樣的寫法，也是不對的：

```
class Car extends Vehicle {  
    constructor() {  
        this.type = 'Car';  
        this.wheels = 4;  
    }  
}
```

儘管看似可以這樣初始化裡面的成員變數，但別忘記了，父類別 `Vehicle` 將成員變數 `type` 以及 `wheels` 都設定成私有狀態，就連子類別也沒辦法輕易讀取與竄改。

類別的繼承裡有一種特殊的關鍵字可以直接連結到父類別的建構子函式——`super`，也就是 Super Class 裡的 `super` 這個單字。你可以想成呼叫 `super` 函式就跟在類別外面呼叫 `new Vehicle` 的概念有些相似，但重點在於，你是在子類別初始化的過程中，呼叫它的父類別的建構子函式。

```
class Car extends Vehicle {  
    constructor() {  
        super('Car', 4);  
    }  
}
```

以上的寫法就可以這樣使用 `Car` 類別：

```
/* 以下兩種分別使用父類別 Vehicle 和子類別 Car 建立實體的方式是等效的 */  
const vehicle = new Vehicle('Car', 4);  
const car = new Car();
```

```
console.log(vehicle.info);
// => 'Type: Car; Wheels: 4'

/* 子類別一樣可以呼叫父類別提供的存取方法，前提是它是 public 模式 */
console.log(car.info);
// => 'Type: Car; Wheels: 4'

vehicle.makeNoise();
// => 'Honk honk!'

/* 子類別一樣可以呼叫父類別提供的成員方法，前提是它是 public 模式 */
car.makeNoise();
// => 'Honk honk!'
```

我們當然也可以在子類別任意新增成員：

```
class Car extends Vehicle {
    constructor(
        public brand: string,
        public color: string
    ) {
        super('Car', 4);
    }
}
```

也可以利用多型（Polymorphism）的特點¹⁰，覆寫（Overwriting）父類別的成員。

```
class Car extends Vehicle {
    // 略 ...

    public makeNoise() {
        console.log('Baaaaaaaaah!');
    }
}
```

¹⁰ 物件導向的核心觀念，請參見本書條目 5.1.4。

但是如果不小心使用到父類別的私有成員就會被 TypeScript 警告的！（如圖 5-7）

```
class Car extends Vehicle {
    // 略 ...

    get info() {
        /* color 與 brand 為 Car 自己宣告的成員變數，但 type 則是父類別的私有成員變數 */
        return `${this.type} Brand: ${this.color} ${this.brand}`;
    }
}
```



圖 5-7 私有狀態的成員除了外面的人不能使用，連繼承的類別也不能

這裡就是**保護（Protected）**模式派上用場的時刻，將父類別 **Vehicle** 的成員變數 **type** 設定為 **Protected** 模式：

```
class Vehicle {
    constructor(
        /* 將 type 設定為 protected 模式 */
        protected type: string,
        private wheels: number,
    ) {}

    // 略 ...
}
```

這麼一來，錯誤訊息就消失掉了。（如圖 5-8）

```

        console.log('Baaaaaaaaah!');
    }

    get info() {
        /* color 與 brand 是父類別的私有成員變數 */
        return `${this.type} ${Brand}: ${this.color} ${this.brand}`;
    }
}

```

子類別使用保護模式下的父類別 Vehicle 的成員變數

圖 5-8 保護模式下的成員除了可以在當前類別內使用外，繼承的類別也可以使用喔

以上大致上就是繼承語法的寫法與特性介紹；另外，如果用久了物件導向中的繼承寫法，它會造成程式上父子類別的相依程度（Dependency，或耦合程度，Coupling）大，這些進階討論的細節將會在物件導向中的應用篇章¹¹ 討論到，讀者看到這裡目前只要知道類別繼承的語法與特性就可以了。

»» 類別繼承 Class Inheritance

1. 類別的繼承過程中，被繼承的類別為父類別（Parent Class，或 Super Class），繼承的類別相對來說就是子類別（Child Class，或 Subclass）。
2. 繼承過後的類別將擁有父類別的所有開放模式（Public）與保護模式（Protected）下的成員。
3. 繼承過後的類別，若想要在初始化過程中，初始化父類別的成員變數，可以使用關鍵字 `super` 函式，因為它就是父類別的建構子函式。
4. 類別繼承的語法必須使用關鍵字 `extends`，宣告方式為：

```
class <subclass> extends <super-class> {}
```

5.2.7 抽象類別 Abstract Class

其實對於初學者來說，5.2 前面的小條目已經算是很夠用的類別語法了，最後要介紹的東西對於初學者會稍微難一些些。（當然，會了就覺得還好，就是學習過程難而已）

¹¹ 物件導向進階篇章，請參見本書章節九。

抽象類別是一種將類別的實踐過程抽象化的概念，聽起來很像是廢話（因為這當然是廢話），但是這裡要講的東西很重要——電腦科學中，抽象化是什麼？

抽象化（Abstraction）的概念為——一系列複雜的過程簡化的步驟。

其實關鍵字就是「簡化」，但這麼無聊的名詞，總得給一些讓人會感到興奮的名稱嘛，所以正式的名稱就被取成抽象化。（是嗎？）

讀者剛學 JavaScript 時，宣告函式（Functions）的目的不外乎就是將重複流程給整理到一個區塊，但更深層的意義是將複雜的流程簡化成一個呼叫一個函式而已。

就好比你去餐廳吃飯，你絕對不會跟服務員說你要的食材原料、烹調順序等等，你會直接根據服務員拿給你的菜單，看上面有什麼就點什麼來吃——菜單上的所有料理都是複雜的烹飪料理過程抽象化的結果；相對於「抽象」這個名詞，應該就是「具象」這個詞彙，所以那些烹調的過程就是將餐點具象化的過程，而餐點的名稱就是抽象詞彙。

熟悉「抽象」這個東西的概念後，作者開始來舉例，慢慢延伸出抽象類別的應用。

假設今天想要設計一個排序器（Sorter），專門排序輸入的東西，以下的範例程式碼宣告類別 Sorter，除了擁有一個成員變數 input，成員方法 sort 使用的排序演算法為泡泡排序法：

```
class Sorter {
    constructor(public input: number[]) {}

    public sort() {
        /* 使用泡泡排序法 Bubble Sort */
        for (let i = this.input.length; i > 0; i -= 1) {
            for (let j = 0; j < i - 1; j += 1) {
                /* 若前面的元素值大於後面的元素值，則交換兩元素 */
                if (this.input[j] > this.input[j + 1]) {
                    const temp = this.input[j];
                    this.input[j] = this.input[j + 1];
                    this.input[j + 1] = temp;
                }
            }
        }
    }
}
```

```
        this.input[j + 1] = temp;
    }
}
}
}
}
```

以下是類別 Sorter 的使用方式：

```
const sorter = new Sorter([8, 5, 2, 6, 1, 4, 3, 9]);

/* 排序前的輸入結果 */
console.log(sorter.input);
// => [8, 5, 2, 6, 1, 4, 3, 9]

/* 開始排序 */
sorter.sort();

/* 排序後的輸入結果 */
console.log(sorter.input);
// => [1, 2, 3, 4, 5, 6, 8, 9]
```

很簡單，就是很單純的排序功能而已；然而，如果我們想要讓此排序器可以接收字串類型的資料排序時，單純改寫成以下的情形：

```
class Sorter {
    constructor(public input: number[] | string) {}

    public sort() {
        /* 使用條目 4.4.2 的型別駐防，處理輸入為數字型陣列型別的情形 */
        if (Array.isArray(this.input)) {
            for (let i = this.input.length; i > 0; i -= 1) {
                for (let j = 0; j < i - 1; j += 1) {
                    if (this.input[j] > this.input[j + 1]) {
                        const temp = this.input[j];
                        this.input[j] = this.input[j + 1];
                        this.input[j + 1] = temp;
                    }
                }
            }
        }
    }
}
```

```
}

/* 其餘情形為字串型別的狀況 */
else {
    for (let i = this.input.length; i > 0; i -= 1) {
        for (let j = 0; j < i - 1; j += 1) {
            /**
             * 如果前者字元以文字排序來看是相對後面的話，就交換位置，
             * 譬如字元 b 比 a 還要後面，所以交換兩者
             */
            if (this.input[j].localeCompare(this.input[j + 1]) === 1) {
                this.input = this.input.slice(0, j) +
                    this.input[j + 1] +
                    this.input[j] +
                    this.input.slice(j + 2);
            }
        }
    }
}
}
```

字串裡字元的交換有些複雜，而判斷依據使用 `String.prototype.localeCompare` 這個方法¹²。

所以我們也可以這樣對字串進行排序：

```
const sorter = new Sorter('helloworld');

/* 排序前的輸入結果 */
console.log(sorter.input);
// => 'helloworld'

/* 開始排序 */
sorter.sort();
```

12 `String.prototype.localeCompare`，請參見 MDN：https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare。

```
/* 排序後的輸入結果 */
console.log(sorter.input);
// => 'dehllloorw'
```

以上的範例程式碼實踐排序器的過程儘管是正確的，但是程式碼本身很難看，於是作者開始進行重構（Refactor）的動作，首先我們可以特別將比對元素的過程給拆出來，變成：

```
class Sorter {
    constructor(public input: number[] | string) {}

    public sort() {
        if (Array.isArray(this.input)) {
            for (let i = this.input.length; i > 0; i -= 1) {
                for (let j = 0; j < i - 1; j += 1) {
                    if (this.compare(j, j + 1)) { /* 交換元素的邏輯略 ... */ }
                }
            }
        } else {
            for (let i = this.input.length; i > 0; i -= 1) {
                for (let j = 0; j < i - 1; j += 1) {
                    /**
                     * 如果前者字元以文字排序來看是相對後面的話，就交換位置，
                     * 譬如字元 b 比 a 還要後面，所以交換兩者
                     */
                    if (this.compare(j, j + 1)) { /* 交換元素的邏輯略 ... */ }
                }
            }
        }
    }

    public compare(index1: number, index2: number): boolean {
        // 數字型陣列型別時的判斷邏輯
        if (Array.isArray(this.input)) {
            return this.input[index1] > this.input[index2];
        }
    }
}
```

```
// 字串型別的判斷邏輯
else {
    return this.input[index1].localeCompare(this.input[index2]) === 1;
}
}
```

第二個部分是，由於數字的陣列與字串的元素交換方式不同，一樣將他們的邏輯拆出來：

```
class Sorter {
    constructor(public input: number[] | string) {}

    public sort() {
        if (Array.isArray(this.input)) {
            for (let i = this.input.length; i > 0; i -= 1) {
                for (let j = 0; j < i - 1; j += 1) {
                    /* 簡化成 swap 方法來交換元素 */
                    if (this.compare(j, j + 1)) this.swap(j, j + 1);
                }
            }
        }
    }

    /* 其餘情形為字串型別的狀況 */
    else {
        for (let i = this.input.length; i > 0; i -= 1) {
            for (let j = 0; j < i - 1; j += 1) {
                /* 簡化成 swap 方法來交換元素 */
                if (this.compare(j, j + 1)) this.swap(j, j + 1);
            }
        }
    }
}

public compare(index1: number, index2: number): boolean { /* 略 */ }

public swap(index1: number, index2: number) {
    // 數字型陣列型別的元素交換邏輯
    if (Array.isArray(this.input)) {
        const temp = this.input[index1];
```

```
        this.input[index1] = this.input[index2];
        this.input[index2] = temp;
    }

    // 字串型別的元素交換邏輯
    else {
        this.input = this.input.slice(0, index1) +
                    this.input[index2] +
                    this.input[index1] +
                    this.input.slice(index2 + 1);
    }
}
}
```

讀者仔細看上面的程式碼，應該會察覺到一件事情，成員方法 `sort` 裡面的邏輯不管是字串型別還是數字型陣列型別都是一模一樣的。也就是說，程式碼可以簡化成如下：

```
class Sorter {
    constructor(public input: number[] | string) {}

    public sort() {
        /* 完全只描述泡泡演算法的過程 */
        for (let i = this.input.length; i > 0; i -= 1) {
            for (let j = 0; j < i - 1; j += 1) {
                /* 簡化成 swap 方法來交換元素 */
                if (this.compare(j, j + 1)) this.swap(j, j + 1);
            }
        }
    }

    public compare(index1: number, index2: number): boolean { /* 略 */ }

    public swap(index1: number, index2: number) { /* 略 */ }
}
```

主要排序的演算法是不是乾淨許多呢？事實上這樣的寫法好處是——`sort` 方法可以專心描述泡泡演算法的內容，而不會因為不同型別的輸入要寫兩份不同的邏輯內容。

現在比較有問題的部分是 `compare` 與 `swap` 方法，因為這兩個方法必須根據輸入型別的不同而有不同的處置方式。

```
class Sorter {
    // 略 ...

    public compare(index1: number, index2: number): boolean {
        // 數字型陣列型別的元素的判斷邏輯
        if (Array.isArray(this.input)) { /* 略 ... */ }

        // 字串型別的元素的判斷邏輯
        else { /* 略 ... */ }
    }

    public swap(index1: number, index2: number) {
        // 數字型陣列型別的元素交換邏輯
        if (Array.isArray(this.input)) { /* 略 ... */ }

        // 字串型別的元素交換邏輯
        else { /* 略 ... */ }
    }
}
```

於是這一次作者選擇拆分成專門排序數字型陣列型別的 `NumberSorter`，以及專門排序字串型別的 `StringSorter`。

```
class NumberSorter {
    /* 由於輸入只限 number[]，因此可以簡化 compare 與 swap 的型別駐防過程 */
    constructor(public input: number[]) {}

    public sort() {
        /* 完全只描述泡泡演算法的過程 */
        for (let i = this.input.length; i > 0; i -= 1) {
            for (let j = 0; j < i - 1; j += 1) {
                /* 簡化成 swap 方法來交換元素 */
                if (this.compare(j, j + 1)) this.swap(j, j + 1);
            }
        }
    }
}
```

```
public compare(index1: number, index2: number): boolean {
    return this.input[index1] > this.input[index2];
}

public swap(index1: number, index2: number) {
    const temp = this.input[index1];
    this.input[index1] = this.input[index2];
    this.input[index2] = temp;
}
}

class StringSorter {
    /* 由於輸入只限 string 因此可以簡化 compare 與 swap 的型別駐防過程 */
    constructor(public input: string) {}

    public sort() {
        /* 完全只描述泡泡演算法的過程 */
        for (let i = this.input.length; i > 0; i -= 1) {
            for (let j = 0; j < i - 1; j += 1) {
                /* 簡化成 swap 方法來交換元素 */
                if (this.compare(j, j + 1)) this.swap(j, j + 1);
            }
        }
    }

    public compare(index1: number, index2: number): boolean {
        return this.input[index1].localeCompare(this.input[index2]) === 1;
    }

    public swap(index1: number, index2: number) {
        this.input = this.input.slice(0, index1) +
                    this.input[index2] +
                    this.input[index1] +
                    this.input.slice(index2 + 1);
    }
}
```

以上的兩個不同的排序器類別，可以這樣使用：

```
const sorter1 = new NumberSorter([8, 5, 2, 6, 1, 4, 3, 9]);
const sorter2 = new StringSorter('helloworld');
```

```
/* 排序前的輸入結果 */
console.log(sorter1.input);
// => [8, 5, 2, 6, 1, 4, 3, 9]
console.log(sorter2.input);
// => 'helloworld'

/* 開始排序 */
sorter1.sort();
sorter2.sort();

/* 排序後的輸入結果 */
console.log(sorter1.input);
// => [1, 2, 3, 4, 5, 6, 8, 9]
console.log(sorter2.input);
// => 'dehllloorw'
```

到這裡，程式碼拆分成 `NumberSorter` 與 `StringSorter` 的結果看起來很不錯，但就還是差那麼一點點，這一次重複的東西變成是成員方法 `sort` 裡面的泡泡排序演算法。

我們可能可以用條目 5.2.6 講到的類別繼承方式解決掉這樣的問題，讓 `NumberSorter` 與 `StringSorter` 繼承到同一個父類別，而該父類別擁有泡泡排序的演算法。

但這裡最頭痛的一點在於如果直接要使用類別繼承的方式處理，父類別就必須先要實踐所有的成員方法 `sort`、`compare` 以及 `swap` 確保子類別會擁有共同的介面；然後子類別再根據不同情形覆寫掉父類別提供的方法。

格式就會長得很像以下的樣子：

```
class Sorter {
    /* 成員變數必須設為 any，因為繼承的類別可能會排序不同種類的型別 */
    constructor(public input: any) {}

    public sort() { /* 泡泡排序法 */ }

    public compare(index1: number, index2: number): boolean {
```

```
/* 預設的父類別實作方式 */
}

public swap(index1: number, index2: number) {
    /* 預設的父類別實作方式 */
}
}

class NumberSorter extends Sorter {
    constructor(input: number[]) {
        super(input); // ← 父類別早就有 input 成員變數，使用 super 初始化
    }

    // 假設父類別預設是排序給數字型陣列，就不用再實作 sort、compare 與 swap 成員方法
}

class StringSorter extends Sorter {
    constructor(input: string) {
        super(input); // ← 父類別早就有 input 成員變數，使用 super 初始化
    }

    // 父類別已提供了泡泡排序演算法，所以 sort 沒必要再覆寫
    // 然而，父類別的 compare 與 swap 成員實作方式並非 StringSorter 預期的，  

    // 因此必須覆寫
    public compare(index1: number, index2: number): boolean { /* 略 ... */ }
    public swap(index1: number, index2: number) { /* 略 ... */ }
}
```

這樣看似是解決問題了，但是寫法算合理嗎？`Sorter` 照理來說應該要由開發者宣告類別後繼承它，並且必須由開發者宣告裡面的成員——也就是排序的判斷基準（也就是 `compare` 方法）以及元素交換的邏輯（也就是 `swap` 方法）。

總算可以開始正式介紹抽象類別的用法了，看到這裡的讀者實在是辛苦你們了！

宣告抽象類別的關鍵情境是：

- 某類別專門設計是被繼承，不能被直接使用。

- 繼承該類別時，強制要求開發者宣告特定規格的成員（這些成員被稱作抽象成員）。

`Sorter` 符合上述兩種情形：專門用來被繼承外，必須要求開發者實踐 `compare` 與 `swap` 方法；所以我們可以得知，`Sorter` 是我們要宣告的抽象類別，而 `compare` 與 `swap` 方法都是我們要求開發者宣告的成員方法，而它們就被稱作為抽象成員（Abstract Member）。

抽象類別 `Sorter` 的宣告方式如下：

```
/* 抽象類別宣告時，使用 abstract 關鍵字 */
abstract class Sorter {
    /* 成員變數必須設為 any，因為繼承的類別可能會排序不同種類的型別 */
    constructor(public input: any) {}

    /* sort 演算法不管任何繼承的型別為何，通通都會是一樣的，所以不是抽象成員，必須要實踐 */
    public sort() { /* 泡泡排序法 */ }

    /* compare 必須由繼承的類別宣告其行為，所以為抽象方法 */
    abstract compare(index1: number, index2: number): boolean;

    /* swap 也必須由繼承的類別宣告其行為，所以為抽象方法，請注意輸出型別要註記為 void */
    abstract swap(index1: number, index2: number): void;
}
```

從以上的範例可以得知，抽象類別裡的抽象成員是不需要被實踐的，因為那些成員的實踐會是當其他類別繼承抽象類別 `Sorter` 時才需要去實踐；你也可以將抽象成員假想成早就實踐過的成員，在抽象類別裡可以自由使用那些抽象成員。

另外，注意到這一行：

```
abstract swap(index1: number, index2: number): void;
```

`swap` 方法儘管是函式型別，但是由於抽象成員並沒有實作過程，因此連輸出型別都無從推斷，所以不僅僅只是參數要註記，連同輸出結果都要註記型別！

如果你沒有註記輸出型別就會出現錯誤訊息喔。(如圖 5-9)

```

    if (this.compare(j, j + 1)) this.swap(j, j + 1);
}
}      (method) Sorter.swap(index1: number, index2: number): any
}
'swap', which lacks return-type annotation, implicitly has an 'any' return
type. ts(7010)

abstract Peek Problem No quick fixes available
abstract swap(index1: number, index2: number);
}

為註記輸出型別，輸出會被無條件推論為 Any 型別

```

圖 5-9 抽象成員若為方法時，不管是輸入還是輸出，都要註記型別

由於抽象類別生來就是要被繼承的，也就是說，如果把抽象類別當成普通類別建構物件時會出現錯誤訊息，因為這行為本身不合理。(如圖 5-10)

```

Cannot create an instance of an abstract class. ts(2511)

Peek Problem No quick fixes available
const invalidSorter = new Sorter([8, 5, 2, 6, 1, 4, 3, 9]);

```

抽象類別不是讓你建構物件實體的！

圖 5-10 抽象類別當成普通類別建構實體時出現的錯誤訊息

當抽象類別宣告好時，就可以將程式碼重構成最終理想結果囉！以下是連同抽象類別，完整的實踐過程：

```

abstract class Sorter {
  constructor(public input: any) {}

  public sort() {
    /* 泡泡演算法 */
    for (let i = this.input.length; i > 0; i -= 1) {
      for (let j = 0; j < i - 1; j += 1) {
        if (this.compare(j, j + 1)) this.swap(j, j + 1);
      }
    }
  }

  abstract compare(index1: number, index2: number): boolean;

```

```
abstract swap(index1: number, index2: number): void;
}

class NumberSorter extends Sorter {
    constructor(input: number[]) {
        super(input);
    }

    // 繼承抽象類別後，抽象成員都得實踐出來！
    public compare(index1: number, index2: number): boolean {
        return this.input[index1] > this.input[index2];
    }

    public swap(index1: number, index2: number) {
        const temp = this.input[index1];
        this.input[index1] = this.input[index2];
        this.input[index2] = temp;
    }
}

class StringSorter extends Sorter {
    constructor(input: string) {
        super(input);
    }

    // 繼承抽象類別後，抽象成員都得實踐出來！
    public compare(index1: number, index2: number): boolean {
        return this.input[index1].localeCompare(this.input[index2]) === 1;
    }

    public swap(index1: number, index2: number) {
        this.input = this.input.slice(0, index1) +
                    this.input[index2] +
                    this.input[index1] +
                    this.input.slice(index2 + 1);
    }
}
```

以上的範例幾乎不會看到重複的部分，況且如果假設有新的資料型態需要進行排序時，我們就不需要重複實踐整個排序演算法，只需要繼承抽象類別 Sorter

後，演算法自然而然就可以被重複利用，這時可以專心想該資料型態裡的元素的比較過程（也就是 `compare` 方法）以及元素的交換邏輯就夠了（也就是 `swap` 方法）！

此外，如果開發者忘記實踐任何一個抽象成員時，就會出現錯誤訊息提醒開發者記得實踐該成員喔。（如圖 5-11）

```

class StringSorter extends Sorter {
  ...cons
  ...
}
} Non-abstract class 'StringSorter' does not implement inherited abstract member
'swap' from class 'Sorter'. ts(2515)

publ Peek Problem Quick Fix...
  ...
  ...
}

// public swap(index1: number, index2: number) {
//   ...
//   ...
// }

```

沒有實踐抽象類別裡的抽象方法時，就會出現錯誤訊息提醒

圖 5-11 成員方法 `swap` 註解掉，由於是被宣告為抽象方法，因此必須被實踐

另外，有些讀者（包含作者）可能會覺得還有一個東西很礙眼：

```

abstract class Sorter {
  constructor(public input: any) {} // ← 這個 Any 型別的輸入

  ...
}

```

事實上，將該 `Any` 型別取代掉的方法是有的，講到第 7 章的泛用型別（Generic Types）時，讀者就會學到處理掉這個 `Any` 型別的方式喔！

»» 抽象類別 Abstract Class

1. 當某個類別被設計為專門被繼承時，以及強制要求開發者必須實踐特定的規格或成員時，該類別可以被宣告為抽象類別。
2. 抽象類別既然是專門被繼承用，理論上單獨使用是不合理的，因此不能拿抽象類別建立實體。
3. 任何繼承抽象類別的子類別都必須實踐抽象類別裡的抽象成員。

4. 抽象類別以及任何抽象成員必須使用關鍵字 abstract 進行宣告。
5. 由於抽象成員被規定是必須實作的規格，因此抽象成員並無存取權限狀態；也就是說，私有或保護模式的抽象成員是不存在的。

► 5.3 型別系統中的類別

5.3.1 型別推論機制

接下來要探討，TypeScript 類別在型別系統裡的角色。條目 5.2.1 有短暫地提到類別本身自成一種型別，所以若以下面的程式碼為例：

```
class Cat {  
    constructor(public name: string, public breed: string) {}  
  
    public introduction() { /* 略 ... */ }  
}  
  
/* aCat 會被推論為型別 Cat */  
const aCat = new Cat('Julia', 'Scottish Fold');
```

變數 aCat 會被自動推論為 Cat 型別，這應該很明顯。畢竟編譯過程只要看到類別名稱大概就推論得出結果。

另外，不管是類別到底繼承自誰，只要看到是哪個類別初始化實體，就會推論為哪種類別：

```
class Vehicle {  
    constructor(public type: string, public wheels: number) {}  
  
    public makeNoise() { /* 略 ... */ }  
}  
  
class Car extends Vehicle {  
    constructor(public brand: string, public color: string) {  
        super('Car', 4);  
    }  
}
```

```
}
```

```
/* vehicle 會被推論為型別 Vehicle */
const vehicle = new Vehicle('Bus', 8);

/* car 則會被推論為型別 Car */
const car = new Car('Cadillac', 'Black');
```

基本上推論的部分沒有太多需要注意的地方。

5.3.2 型別註記機制

由於類別是一種型別，理應來說，我們可以拿類別名稱來註記。以條目 5.3.1 的類別 Cat 為例：

```
/* definitelyACat 被註記為 Cat 型別 */
const definitelyACat: Cat = new Cat('Julia', 'Scottish Fold');
```

另外，如果是父子繼承關係的類別，子類別除了可以註記為該子類別的型別外，也可以選擇註記為父類別的型別。以條目 5.3.1 的類別 Vehicle 與類別 Car 為例：

```
/* Car 為 Vehicle 的子類別，註記為 Car 或 Vehicle */
const car1: Car      = new Car('Cadillac', 'Black');
const car2: Vehicle = new Car('Cadillac', 'Black');
```

子類別被註記為子類別型別應該可以理解；而子類別由於繼承了父類別的特性，因此當然也可以將子類別建構的實體指派到註記為父類別的變數中。

但如果是相反的情形，父類別建構的實體能不能指派到註記為子類別的變數呢？儘管說作者不建議讀者這麼做，也未曾想過要這麼做，但這裡會延伸出一個很重要的觀念——任何物件能否被指派到某個被註記過後的變數關鍵在於物件的資料結構。

如果說今天父子類別的宣告方式為：

```
class Parent {
```

```
constructor(public propA: string, public propB: number) {}

public methodA() { /* 略 ... */ }

}

class Child extends Parent {
  constructor() {
    super('Hello world', 12345); // 沒有任何新的成員變數、方法宣告在子類別上
  }
}
```

由於子類別 `Child` 除了繼承 `Parent` 之外並沒有額外宣告任何新的成員，代表父子類別的結構大致上相同。

所以以下的程式碼，將父類別建構的實體指派到子類別去，聽起來會出錯，但實質上並不會出現錯誤訊息的。(如圖 5-12)

```
// 明明是子類別型別的變數，指派父類別的值也不會出錯
const shouldBeChild: Child = new Parent('this is parent', 54321);
```

事實證明，只要結構符合，什麼型別的物件都可以被接受

圖 5-12 由於父子類別結構上並無差異，因此父類別實體可以丟到子類別型別的變數

然而，如果將子類別改寫成：

```
class Child extends Parent {
  constructor() {
    super('Hello world', 12345);
  }

  // 多了父類別沒有的成員方法
  public methodB() { /* 略 ... */ }
}
```

因為子類別多了父類別所沒有的成員，把父類別的值塞到子類別型別的變數時，就會發生錯誤。(如圖 5-13)

```
// 子類別比起父類別多一個成員，指派父類別的值就會出錯
const shouldBeChild: Child = new Parent('this is parent', 54321);
const shouldBeChild: Child

Property 'methodB' is missing in type 'Parent' but required in type
'Child'. ts(2741)

hello-world
Peek Pro
```

methodB 並沒有在父類別宣告過，因此結構上，子類別與父類別是不一樣的東西

圖 5-13 此時父子類別的結構有差異，因此父類別實體不能指派到子類別型別的變數

簡而言之，判斷物件能不能被指派到某個被註記的變數裡，一切都是看結構，並不是說父類別的東西不能丟到註記為子類別型別的變數。討論到下一個章節，也就是介面（Interface）時，這個重點會被更加強調出來。

»》類別的推論與註記機制 Type Inference & Annotation of Class

- 當為註記的變數被指派某類別建構的實體時，推論結果就是該類別建構子的名稱。
- 子類別建構出的實體，絕對可以指派到被註記為子類別或者是父類別型別的變數中。
- 父類別建構出的實體，除了可以被指派到被註記為父類別的變數外，如果繼承的子類別並沒有額外宣告新的成員時，父類別實體也可以被指派到該子類別型別的變數，關鍵都是在看父子類別結構有沒有相等。

► 本章練習

- 類別裡的成員變數（Member Variables）與建構出的實體屬性（Property）差異性在哪？
- 建構子函式（Constructor Function）是什麼？
- 存取修飾子（Access Modifiers）是什麼？

4. 試簡化以下的程式碼。

```
class Example {  
    public prop1: string;  
    private prop2: number;  
    protected prop3: boolean = true;  
  
    constructor(input1: number, input2: string) {  
        this.prop1 = input1;  
        this.prop2 = input2.toUpperCase();  
    }  
}
```

5. 試宣告類別長方形 Rectangle 的程式碼，並且擁有以下的規格：

- 擁有兩個私有狀態的成員 width 與 height，皆為數字型別的值。
- 擁有兩個開放狀態的成員方法 calcArea 與 calcCircumference，分別為計算長方形面積與周長的方法。
面積公式為：width * height
周長公式為：(width + height) * 2
- 初始化時第一個參數初始化成員變數 width 的值，第二個參數初始化成員變數 height 的值。

宣告完的類別長方形 Rectangle 的程式碼的使用方式如下：

```
const rect = new Rectangle(4, 5); // width 為 4 與 height 為 5 的長方形  
  
/* 以下這兩行會出現錯誤，因為屬性都是私有狀態 */  
// rect.width;  
// rect.height;  
  
/* 長方形的面積與周長 */  
console.log(rect.calcArea());  
// => 20  
  
console.log(rect.calcCircumference());  
// => 18
```

6. 承第 5 題，想要讓 Rectangle 實體可以使用 setWidth 或 setHeight，更改類別內部成員變數的值，你會如何修改 Rectangle 內部的宣告方式？（以下為 Rectangle 修改後的使用方式）

```
const rect = new Rectangle(4, 5); // width 為 4 與 height 為 5 的長方形

console.log(rect.calcArea());
// => 20
console.log(rect.calcCircumference());
// => 18

/* 修改長方形的 width 和 height */
rect.setWidth(7); // ← width 被修改為 7
rect.setHeight(4); // ← height 被修改為 4

console.log(rect.calcArea());
// => 28
console.log(rect.calcCircumference());
// => 22
```

7. 承第 6 題，如何在不宣告新的成員變數與成員方法的條件下，實踐出新的版本的 Rectangle 類別，使得呼叫實體的屬性 area 檢視長方形的面積、屬性 circumference 檢視長方形的周長也會隨 width 或 height 改變時更新？

```
const rect = new Rectangle(4, 5); // width 為 4 與 height 為 5 的長方形

console.log(rect.area);
// => 20
console.log(rect.circumference);
// => 18

/* 修改長方形的 width 和 height */
rect.setWidth(7); // ← width 被修改為 7
rect.setHeight(4); // ← height 被修改為 4

console.log(rect.area);
// => 28
console.log(rect.circumference);
// => 22
```

8. 承第 7 題，假設今天想要實踐出 Rectangle 實體以下的行為：

```
const rect = new Rectangle(4, 5); // width 為 4 與 height 為 5 的長方形

console.log(rect.area);
// => 20
console.log(rect.circumference);
// => 18

/** 
 * 指派 [number, number] 型別的值進到 dimension 屬性， 
 * 第一個值可以改掉 width，第二個值可以改掉 height
 */
rect.dimension = [7, 4] as [number, number];
// width 被修改為 7, height 被修改為 4

console.log(rect.area);
// => 28
console.log(rect.circumference);
// => 22
```

當指派元組型別 [number, number] 的值到實體的屬性 dimension 就會修改 width 或 height 的值，試修改類別 Rectangle 的宣告方式。

9. 承第 8 題，試宣告正方形類別 Square，其特徵是只有一個輸入 size，並且計算面積與周長的方式跟類別 Rectangle 一模一樣，差異在於——類別 Square 的屬性 size 為 Rectangle 中的成員變數 width 與 height 皆相等的情形。(註：本題的 size 不一定要宣告為 Square 的成員變數)

以下是類別 Square 的使用情形，以及等效的 Rectangle 使用情形：

```
const sq = new Square(5);           // size 為 5 的正方形
const rect = new Rectangle(5, 5); // width 為 5 與 height 為 5 的長方形

console.log(sq.area);
// => 25
console.log(rect.area);
// => 25
```

10. 承第 8 題，試改成新的版本的類別 Rectangle，滿足以下的使用方式。

```
// 計算 width 為 4 以及 height 為 5 的長方形面積
console.log(Rectangle.area(4, 5));
// => 20

// 計算 width 為 4 以及 height 為 5 的長方形周長
console.log(Rectangle.circumference(4, 5));
// => 18
```

11. 類別的普通成員與靜態成員差異性在哪裡？靜態成員跟普通成員在命名上可以重複嗎？

12. 任何普通成員與靜態成員皆可以標注存取修飾子？

13. 若類別 Parent 與類別 Child 的宣告方式如下：

```
class Parent {
    constructor(public propA: number, public propB: string) {}

    public methodA() {}
}

class Child extends Parent {
    constructor(inputA: number, inputB: string) {
        super(inputA, inputB);
    }

    public methodA() {}
}
```

請問以下的指派行為會有錯誤嗎？

```
const annotatedAsChild: Child = new Parent(123, 'Hello world');
const annotatedAsParent: Parent = new Child(123, 'Hello world');
```

14. 承 13 題，若 Parent 宣告方式不變，Child 宣告方式為：

```
class Child extends Parent {
    constructor(inputA: number, inputB: string) {
        super(inputA, inputB);
```

```
}

public methodA() {}
public methodB() {} // ← 多出成員方法 methodB
}
```

這樣會造成第 13 題的指派行為出錯嗎？

15. 承 13 題，若 Parent 宣告方式不變，Child 宣告方式為：

```
class Child extends Parent {
  constructor(inputA: number, inputB: string) {
    super(inputA, inputB);
  }

  public methodA() {}
  private methodB() {} // ← 多出私有成員方法 methodB
}
```

這樣會造成第 13 題的指派行為出錯嗎？

16. 承 13 題，若 Parent 宣告方式不變，Child 宣告方式為：

```
class Child extends Parent {
  constructor(inputA: number, inputB: string) {
    super(inputA, inputB);
  }

  public methodA() {}
  static methodB() {} // ← 多出類別方法 methodB
}
```

這樣會造成第 13 題的指派行為出錯嗎？

06

TypeScript 介面

介面（Interface）的部分已經算是基礎篇章後面的章節，也算是很容易跟型別化名（Type Alias）搞混的章節。

► 6.1 介面的介紹 Introduction to Interface

6.1.1 介面的定義 Definition of Interface

講到介面就要清楚「介面」兩字的真實意義。

將複雜的系統套上人性化的設計，這應該就是介面兩字的真諦。阿~~們~~

就好比讀者打開筆記型電腦，眼前看到的鍵盤與螢幕就是一種可以被人操作的介面；拆掉筆記型電腦，裡面一定都是密密麻麻的電路板，鑲嵌各種不同的IC晶片、電子原件等等，但人類想當然不會直接接觸到電路底層的東西，而是藉由操作介面——也就是鍵盤來操作電腦的指令與程式。

介面的概念跟條目 5.2.7 提到的抽象類別中的抽象化（Abstraction）概念有關，抽象化是將複雜的流程簡化的過程；介面與之有相似的概念，兩者比較起來是：

將複雜系統簡化的過程叫做抽象化。

將複雜系統簡化的結果叫做介面。

一個講究過程、另一個講究結果，廣義來說，我們可以認為函式的宣告是在對某段複雜程式進行的抽象化，而函式本身提供了輸入參數的介面。

以下宣告一個泡泡排序演算法的函式 `bubbleSort`：

```
function bubbleSort(input: number[]) {
    const result = [...input]; // ← 深度複製整個陣列

    for (let i = result.length; i > 0; i -= 1) {
        for (let j = 0; j < i - 1; j += 1) {
            if (result[j] > result[j + 1]) {
                const temp = result[j];
                result[j] = result[j + 1];
                result[j + 1] = temp;
            }
        }
    }

    return result;
}
```

使用該函式的時候：

```
bubbleSort([2, 3, 6, 1, 4, 5]);
// => [1, 2, 3, 4, 5, 6]
```

你不需要知道該函式內部是怎麼被實作的，因為已經被它給抽象化了；你只需要知道該函式的用途以及使用方法，而該函式提供一個參數接口，這個參數接口就相當於該函式 `bubbleSort` 的介面的概念。

6.1.2 TypeScript 介面的基礎宣告 Interface Declaration

條目 6.1.1 講到的東西比較偏向於介面的廣義概念，所以作者強調，程式碼不一定需要 TypeScript 的介面語法來幫助你宣告程式碼，前提在於程式碼本身的

可讀性與維護性足夠的狀態下。(老實說，專案要能夠達成可讀性與維護性也是挺需要經驗)

回歸正題，TypeScript 的介面有別於條目 6.1.1 講到的廣義介面的概念差異性到底在哪裡呢？

TypeScript 提供的的介面語法著重在描述任何物件 (Object) 或類別 (Class) 的規格 (Specification)。

作者果斷向讀者說：

「TypeScript Interface 形同規格的制定。」

其實也不為過。

一般介面的宣告跟普通的 JSON 物件型別化名的宣告非常像，差別在於使用的關鍵字是 **interface**。

```
interface PersonalInfo {  
    name: string;  
    age: number;  
    interest: string[];  
}
```

很像在描述物件吧！基本上條目 3.3 宣告 JSON 物件型別的方式，宣告介面時也都可以使用，譬如說如果用介面描述條目 5.2.1 曾出現過的類別 **Cat** 的規格：

```
interface Cat {  
    /* 普通成員變數的規格 */  
    name: string;  
    breed: string;  
    noise: string;  
  
    /* 普通成員方法的規格，使用函式型別格式 */  
    makeNoise(): void;  
    feed(something: string): void;  
}
```

選用屬性（Optional Property）當然也可以作為介面宣告的規格：

```
interface Cat {  
    // 其他的規格略 ...  
  
    // 由於貓咪不一定有主人，因此將 owner 設定為選用屬性  
    owner?: PersonalInfo;  
}
```

設定唯讀屬性（Read-Only Property）也是可以的：

```
interface Cat {  
    // 其他的規格略 ...  
  
    // 難道讀者有養過性別會變來變去的貓咪嗎！？  
    readonly sex: 'Male' | 'Female';  
}
```

另外，也可以設定像條目 4.3 的可控索引型別（Indexable Type）一樣的條件：

```
interface StringDictionary {  
    [key: string]: string;  
}
```

還有一種宣告方式是單純宣告函式的介面：

```
interface BinaryOperation {  
    (operand1: any, operand2: any): any;  
}
```

作者本身很少用到單純函式宣告成介面的用法，不過也是根據專案性質，看有沒有需要用到¹。

不過這時讀者可能會問：「用條目 3.1.3 講到的型別化名（Type Alias）不就夠了嗎？」

介面最重要的性質就在下一個條目。

¹ 講到第 7 章的泛用型別時，介面的宣告方式也會更變態有彈性。

»» 介面的定義與宣告 Definition and Declaration of Interface

1. 介面廣義來說，即一般複雜程式簡化成比較人性化的結果。
2. TypeScript 提供的介面語法偏向於制定物件或類別的規格 (Speculation)。
3. 宣告介面時，使用關鍵字 `interface` :

```
interface I { /* 介面 I 的規格內容 */ }
```

4. 由於 TypeScript 介面的定位是實作規格，因此並不包含實作過程與內容。
5. 宣告介面規格的方式與宣告物件型別或可控索引型別的方式類似。

► 6.2 介面的彈性 Flexibility of Interface

本條目講的東西是 TypeScript 介面與型別化名差異性的最大關鍵，記得：介面擁有條目 6.2 的特性，型別化名則沒有²。

6.2.1 介面的延展 Interface Extension

關於介面延展這個功能，有些資源也會用介面繼承 (Interface Inheritance) 稱呼這個功能，這兩種說法都可以，只是作者偏好前者的說法，因為會用到關鍵字 `extends`，不過讀者覺得爽就好。

就用條目 6.1.2 裡面的使用者資料的介面 `PersonalInfo` 為例：

```
interface PersonalInfo {
    name: string;
    age: number;
    interest: string[];
}
```

2 就算型別化名有本條目講的功能，也不應該在型別化名宣告時使用，因為意義上來說不合理。

假設今天我們想要宣告一個新的使用者帳戶的介面為 `UserAccount`，除了有一些跟帳戶資料相關的規格屬性外，還要包含介面 `PersonalInfo` 裡的規格時，我們可以使用介面延展的功能：

```
interface UserAccount extends PersonalInfo {
    email: string;
    password: string;
    subscribed: boolean;
}
```

這樣一來，`UserAccount` 除了有它自己本身定義的規格外，它還擁有 `PersonalInfo` 裡的規格，因此等效於：

```
interface UserAccount {
    // UserAccount 定義規格
    email: string;
    password: string;
    subscribed: boolean;

    // 從 PersonalInfo 延展而來的規格
    name: string;
    age: number;
    interest: string[];
}
```

由於這個行為很像是類別在繼承的行為，所以才會有介面繼承這樣的說法。

但介面延展不僅僅只能延展單個介面，多個介面是可以同時延展的，譬如再多新增一個 `SocialLinks` 介面專門描述使用者的社群網站連結。

```
interface SocialLinks {
    facebook?: string;
    twitter?: string;
    linkedin?: string;
    websites: ({ name: string; url: string })[];
}
```

`UserAccount` 介面可以同時延展 `PersonalInfo` 以及 `SocialLinks` 介面：

```
interface UserAccount extends PersonalInfo, SocialLinks {
```

```
/* UserAccount 定義規格內容略 ... */  
}
```

»» 介面的延展 Interface Extension

1. 介面可以從其他宣告過的介面延展，使得該介面可以擁有其他介面已宣告過的規格。
2. 介面的延展不限於一個介面，可以延展多個介面。
3. 使用介面延展時，使用關鍵字 `extends`：

```
interface I extends I1, I2, ..., IN { /* 介面 I 的規格內容 */ }
```

其中，介面 I_1, I_2, \dots, I_N 為被延展的介面。

6.2.2 介面的融合 Interface Merging

介面還有另一種特性，儘管本條目的標題叫做「介面的融合」，但精確一點的名稱叫做「宣告的融合」(Declaration Merging)³，只是因為常看到的情況都是介面的宣告方面的融合，所以標題才會特別指介面的融合這種情形。

融合的條件只有一個——同一個介面名稱下重複宣告的情形，譬如：

```
interface PersonalInfo {  
    name: string;  
    age: number;  
    interest: string[];  
}  
  
interface PersonalInfo {  
    gender: 'Male' | 'Female';  
    married: boolean;  
}
```

³ 除了介面外，TypeScript 模組系統中的命名空間 (Namespacing) 也是宣告的融合中的另一主題，這會在本書條目 8.2 討論到。

介面重複宣告不等同於覆寫（Overwrite）的行為，而是進行宣告性融合。以上的範例程式碼的等效寫法為：

```
interface PersonalInfo {  
    // 原先宣告時的規格  
    name: string;  
    age: number;  
    interest: string[];  
  
    // 重複宣告時的規格  
    gender: 'Male' | 'Female';  
    married: boolean;  
}
```

另外，由於介面規格的宣告是嚴謹的，所以介面進行定義融合時，若出現屬性與型別定義衝突時，就會出現錯誤訊息。（以下的程式碼範例產生錯誤訊息如圖 6-1）

```
interface A {  
    propA: string;  
    propB: number; // ← 介面 A 原先宣告 propB 屬性對應型別為 number  
}  
  
interface A {  
    propA: string;  
    propB: string; // ← 介面 A 重複宣告時，propB 屬性對應型別為 string  
}
```

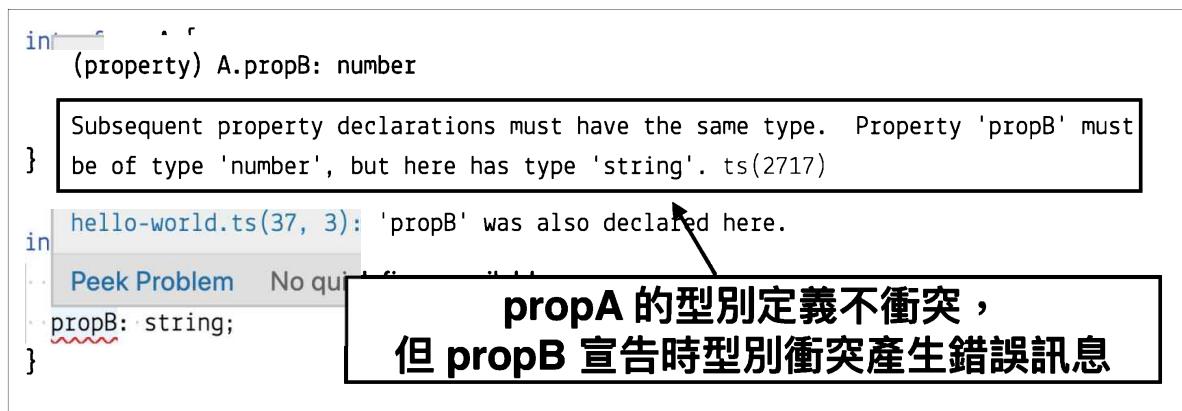


圖 6-1 介面中的規格產生宣告性衝突的錯誤訊息

»» 介面的融合 Interface Merging

1. 介面的融合為定義的融合（Declaration Merging）中的一種，也是最常見的融合性宣告手法。
2. 介面的宣告時，若先前有重複宣告過同個介面時，就會啟動介面融合機制。
3. 介面由於是嚴格宣告規格內容，因此融合過程中產生型別衝突時，介面的融合就會失敗。

6.2.3 函式超載性宣告 Function Declaration Overloading

還記得討論物件導向的概念時，條目 5.1.4 有提到其中一個功能是函式超載（Function Overloading），意思是指同個函式會根據不同的輸入參數的個數、型別與順序等，而有不同的行為。

TypeScript 沒有這種直接實踐函式超載的功能，畢竟函式若重複宣告在 JavaScript 程式碼時，等同於覆寫掉先前宣告的函式。然而，如果說是宣告性的超載就可不一定，譬如以下的範例程式碼。

```
function safeAddition(  
    input1: number | string,  
    input2: number | string  
): number {  
    const num1 = typeof input1 === 'number' ? input1 : parseInt(input1, 10);  
    const num2 = typeof input2 === 'number' ? input2 : parseInt(input2, 10);  
    return num1 + num2;  
}
```

以上的函式 `safeAddition` 的用途是，輸入參數不管是字串還是數字都會用正常的數字方式去做加法運算。

而 `safeAddition` 的超載式的函式宣告寫法為：

```
function safeAddition(input1: number, input2: number): number;  
function safeAddition(input1: string, input2: number): number;  
function safeAddition(input1: number, input2: string): number;  
function safeAddition(input1: string, input2: string): number;
```

對，你沒看錯，就是以上的宣告方式，只是你在實踐（Implement）以上的超載後的函式型別規格時，得照樣得按照原先的方式實踐。完整程式碼為：

```
/* 規格，超載性函式型別宣告 */
function safeAddition(input1: number, input2: number): number;
function safeAddition(input1: string, input2: number): number;
function safeAddition(input1: number, input2: string): number;
function safeAddition(input1: string, input2: string): number;

/* 實踐 Implementation */
function safeAddition(
    input1: number | string,
    input2: number | string
): number {
    const num1 = typeof input1 === 'number' ? input1 : parseInt(input1, 10);
    const num2 = typeof input2 === 'number' ? input2 : parseInt(input2, 10);
    return num1 + num2;
}
```

讀者可能會懷疑為何還要先宣告超載的函式規格，難道不用直接實踐函式就好了嗎？

其實宣告規格的好處一來是避免遺漏掉某個條目，二來是函式的實踐過程中，程式碼的參數是用複合型別⁴的方式表示，比起超載性宣告，一條一條乾淨的寫法還要來得複雜。

基本上，函式超載性宣告不是給程式碼執行的，是要給人類看的東西。另外，函式的超載性宣告完畢後，一定要馬上接函式的實踐過程，如果是以下的情形，就算中間只是塞個 `console.log` 還是會出錯。（如圖 6-2）

```
/* 規格，超載性函式型別宣告 */
function safeAddition(input1: number, input2: number): number;
// 略 ...

console.log('Hello world!'); // ← 介於函式的宣告與實踐之間會出現錯誤訊息
```

4 複合型別（Composite Types），請參見本書條目 4.4。

```
/* 實踐 Implementation */
function safeAddition(
    input1: number | string,
// 略 ...
```

```
function safeAddition(input1: string, input2: string): number (+4 overloads)
function: Function implementation is missing or not immediately following the
function: declaration. ts(2391)
function: Peek Problem No quick fixes available
function: safeAddition(input1: string, input2: string): number;

console.log('Hello World!');
```

函式的實踐必須馬上接在宣告的後方

```
function safeAddition(input1: number | string, input2: number | string): number {
```

圖 6-2 函式規格的超載性宣告與實踐中間摻雜亂七八糟的程式碼

讀者可能會懷疑為何作者要在介面的部分突然塞一個不是介面宣告類型的主題；事實上，條目 6.1.2 的結尾有提到純函式的介面宣告手法就可以搭配函式超載性宣告：

```
interface Addition {
    (input1: number, input2: number): number;
    (input1: string, input2: number): number;
    (input1: number, input2: string): number;
    (input1: string, input2: string): number;
}
```

是的，介面內部可以存在函式的超載性宣告，而且比起將函式的超載性宣告的程式碼寫在外面，將其包裝在介面就可以隨意地將其放在任意的地方，函式的實踐自然就沒必要強制性的馬上宣告囉！

以下的程式碼就不會出現如圖 6-2 的錯誤訊息。

```
interface SafeAddition {
    (input1: number, input2: number): number;
    // 其他的超載性宣告略 ...
}
```

```
console.log('Hello world!'); // ← 函式的實踐就可以放到很後面喔～
```

```
const safeAddition: SafeAddition = function (
  input1: number | string,
  // 實踐過程略 ...
```

實際上，真的有案例是這樣宣告函式的型別，寫前端的人最常用到的建立 DOM 節點的 `document.createElement` 方法的規格就可以這樣寫喔～

```
// 以下的寫法不一定是官方的宣告寫法，但宣告方式近似
interface CreateElement {
  (tagName: 'p'): HTMLParagraphElement;
  (tagName: 'a'): HTMLAnchorElement;
  (tagName: 'button'): HTMLButtonElement;
  (tagName: 'input'): HTMLInputElement;
  // 其他的超載性宣告略 ...
}
```

讀者想想看，要是沒有超載性宣告，以上的等效程式碼可能會變成這一坨災難：

```
interface CreateElement {
  (tagName: 'p' | 'a' | 'button' | 'input' | /* 更多不同的元素略 ... */):
    HTMLParagraphElement |
    HTMLAnchorElement |
    HTMLButtonElement |
    HTMLInputElement |
  // 其他的超載性宣告略 ...
};
```

而且這種聯集複合的宣告手法並非可以像函式超載性的宣告手法有相同效果，超載的宣告是一條一條的宣告，以下面的宣告方式為例：

```
(tagName: 'p'): HTMLParagraphElement;
```

它很明確地指出，輸入若為明文字串 'p' 時，對應的輸出為 `HTMLParagraphElement` 這個型別的值。

但是後面的複合型別的宣告手法使得輸入與輸出的宣告被纏繞在一起，搞得好像是輸入為 'p' 時，輸出為 `HTMLParagraphElement | HTMLAnchorElement | ...` 等型別。

»» 函式的超載性宣告 Function Declaration Overloading

1. 函式型別的宣告可以進行超載，也就是說，可以預先略過實踐的過程，宣告輸入以及輸出型別，而後再去實踐完整的結果。
2. 函式型別進行超載性宣告後，必須馬上實踐。
3. 若將函式超載性宣告的過程包裝在介面裡，就不需要遵守第 2 點的限制。

► 6.3 註記與實踐介面

繼前幾章討論到的型別化名與類別，介面是最後一種隸屬於型別的東西（然而，請不要把介面和型別化名這兩個東西劃上等號，後面的章節會說明兩者的不同）。

既然介面也是型別的一種，理所當然，本條目要討論註記介面、實踐介面的過程。

6.3.1 JSON 物件與介面

最明顯，可以註記的東西就是 JSON 物件，它跟介面的宣告格式、長相都差不多，如果不能註記那 TypeScript 真是毫無路用的神奇的語言。以條目 6.1.2 的介面 `PersonalInfo` 的註記使用為例：

```
const maxwell: PersonalInfo = {
  name: 'Maxwell',
  age: 18,
  interest: ['drawing', 'programming']
};
```

它跟註記型別化名（Type Alias）的效果幾乎沒兩樣。（不過意義上有差別）

另外，如果你沒有實踐 `PersonalInfo` 介面裡宣告的規格，就會出現錯誤訊息並且提醒缺少的屬性。(如圖 6-3)

The screenshot shows a TypeScript code editor with the following code:

```
const maxwell: PersonalInfo
Property 'interest' is missing in type '{ name: string; age: number; }' but required in type 'PersonalInfo'. ts(2741)
    hello-world.ts(4, 3): 'interest' is declared here.
Peek Problem No quick fixes available
```

The error message "Property 'interest' is missing in type '{ name: string; age: number; }' but required in type 'PersonalInfo'. ts(2741)" is highlighted with a red box. An arrow points from this box to a callout box containing the text "interest 屬性為 PersonalInfo 介面的規格，缺少就會出現錯誤訊息".

```
const maxwell: PersonalInfo = {
  name: 'Maxwell',
  age: 18,
  // 省略其中一個規格
  // interest: ['drawing', 'programming']
};
```

圖 6-3 `PersonalInfo` 介面的規格沒有完整實踐時，出現的錯誤訊息

6.3.2 函式與介面

事實上，在條目 6.2.3 討論的函式超載性宣告裡的 `safeAddition` 函式案例就已經有範例程式碼將函式介面註記在變數上，並且也有實踐過程等等。

不過要注意以下兩種介面宣告寫法：

```
/* 純函式超載性宣告介面 */
interface AdditionFunction {
  (input1: number, input2: number): number;
  (input1: string, input2: number): number;
  (input1: number, input2: string): number;
  (input1: string, input2: string): number;
}

/* 看起來像純函式宣告但實質上是屬性為函式型別的超載性宣告 */
interface AdditionFunctionProperty {
  addition(input1: number, input2: number): number;
  addition(input1: string, input2: number): number;
  addition(input1: number, input2: string): number;
  addition(input1: string, input2: string): number;
}
```

前面是純函式介面的宣告手法，註記使用方式為指派函式型別的值並實踐該介面的內容：

```
const safeAddition: AdditionFunction = function /* 實踐過程略 ... */;
```

但是 `AdditionFunctionProperty` 的宣告方式是物件型別的宣告方式，意思是說，變數被註記此介面時必須這樣實踐：

```
const safeAdditionObj: AdditionFunctionProperty = {
    addition: function /* 實踐過程略 ... */,
};
```

如果把介面 `AdditionFunctionProperty` 的宣告手法改成以下這樣，應該就可以看得出該介面的使用方法。

```
/* 看起來像純函式宣告但實質上是屬性為函式型別的超載性宣告 */
interface AdditionFunctionProperty {
    addition(input1: number, input2: number): number;
    addition(input1: string, input2: number): number;
    addition(input1: number, input2: string): number;
    addition(input1: string, input2: string): number;

    // 額外宣告其他規格
    propB: string;
}
```

因此，實踐 `AdditionFunctionProperty` 介面的程式碼會變成：

```
const safeAdditionObj: AdditionFunctionProperty = {
    addition: function /* 實踐過程略 ... */,
    propB: 'Hello world'
};
```

另外，作者不會在本書討論以下這種混合式介面（Hybrid Interface）⁵ 告手法：

⁵ 混合式介面的宣告與使用，請參見官方文件：<https://www.typescriptlang.org/docs/handbook/interfaces.html#hybrid-types>。

```
/* 混合純函式與普通屬性的混合式宣告手法 */
interface HybridAdditionFunctionProperty {
  (input1: number, input2: number): number;
  (input1: string, input2: number): number;
  (input1: number, input2: string): number;
  (input1: string, input2: string): number;

  // 額外宣告其他規格
  propB: string;
}
```

簡短的原因是——作者主觀覺得很難用、也不實用，沒必要用這個來折騰讀者。

6.3.3 類別實踐介面 Interface Implementation

事實上，介面如果拿來做條目 6.3.1 的事情實在是很浪費，通常用普通的型別化名（Type Alias）就夠了；除非你想善用條目 6.2 講到的介面的彈性機制時，才有宣告介面的必要。

介面反而是在物件導向語言中的類別（Class）有很大的協作用途⁶。

回顧一下條目 6.1.2 對於 TypeScript 介面的描述：

「TypeScript Interface 形同規格的制定。」

規格（Speculation，通常簡稱 Spec）的特性就是軟體產出的標準，用在類別上就是描述類別該實踐的成員樣貌。

由於類別內部的成員變數與方法可以宣告成不同種類的模式（即存取修飾子⁷），但規格本身的性質是開放的，藏在黑盒子裡的東西是不需要特別強調或標注在介面的規格上。實作過程（Implementation）跟規格設計兩者看似息息相關，其實是兩回事。

6 更多進階應用請參見本書章節九。

7 類別存取修飾子（Access Modifiers），請參見本書條目 5.2.2。

一般設計規格的時候，不會管實作的細節，只管使用者是如何操作物件外部提供的屬性或方法（其實就是指物件的「操作介面」）。

而實作的目標與結果要符合規格制定的樣貌，所以偏向於從屬關係。（實作結果終究要遵照規格的設計，不然規格制定完卻不照規格走，制定心酸的嗎？）

既然規格只規範外部的行為，也就是說：

TypeScript Interface 宣告的規格，對於類別裡的成員來說，是開放狀態的。

TypeScript 介面不會規範類別的私有或保護狀態的成員，它只規範開放狀態（也就是 Public）的成員。

以條目 6.1.2 的範例介面 Cat 為例，不過這一次名稱取 CatInterface，避免跟類別名稱衝突。

```
interface CatInterface {  
    /* 普通成員變數的規格 */  
    name: string;  
    breed: string;  
    noise: string;  
  
    /* 普通成員方法的規格，使用函式型別格式 */  
    makeNoise(): void;  
    feed(something: string): void;  
}
```

以上的介面規範三種不同的（開放狀態的）成員變數 name、breed 與 noise，以及兩個（開放狀態的）成員方法 makeNoise 與 feed。

若類別想要實踐此介面，必須使用關鍵字 implements，這下子讀者應該也知道為何標題要下「類別實踐介面」而非「類別與介面」之類的標題吧。

```
/* 實踐 Cat Interface 介面的範例 */  
class Cat implements CatInterface {  
    public name: string;  
    public breed: string;  
    public noise: string = 'Meow meow meow ~';
```

```
constructor(name: string, breed: string) {
  this.name = name;
  this.breed = breed;
}

public makeNoise() {
  console.log(this.noise);
}

public feed(something: string) {
  console.log(`${this.name} is eating ${something}...`);
}
}
```

首先，如果類別實踐介面的規格中，任何一個成員不見或沒有實踐時，會出現提醒訊息。(如圖 6-4)

The screenshot shows a TypeScript code editor with the following code:

```
nois
  .
  .
  .
  class Cat

  /* 實踐 Class 'Cat' incorrectly implements interface 'CatInterface'.
  make   Property 'noise' is missing in type 'Cat' but required in type
  feed   'CatInterface'. ts(2420)
  }

  /* 實踐 Peek Problem Quick Fix...
  class Cat implements CatInterface {
    public name: string;
    public breed: string;
    // 將 noise 這個成員拔除
    // public noise: string = 'Meow·meow·meow~';
  }
```

A tooltip box contains the text: "noise 為 CatInterface 要求的成員規格，被拔除時，就會出現錯誤訊息". An arrow points from this tooltip to the removed 'noise' member in the code.

圖 6-4 實踐結果不符出現的錯誤訊息

另外，如果說成員規格不為開放模式時，也會出現錯誤訊息喔。(如圖 6-5)

```

/* 錯誤
class Cat
make
feed Class 'Cat' incorrectly implements interface 'CatInterface'.
}
Property 'noise' is private in type 'Cat' but not in type
'CatInterface'. ts(2420)

/* 實踐 Peek Problem No
class Cat implements CatI
· public name: string;
· public breed: string;
· // 將 noise 這個成員變成私有狀態
private noise: string = 'Meow·meow·meow·~';

```

noise 是私有成員，但介面規定的規格是開放狀態

圖 6-5 規格規定的就是開放狀態的成員，私有或保護模式下的成員是不符的

介面由於具備高度的延展彈性，因此類別可以同時實踐多個介面。

```

class C implements I1, I2, ..., IN {
    /* 類別的宣告 */
}

```

另外，就算是經過繼承的子類別也可以實踐介面：

```

class Child extends Parent implements I1, I2, ..., IN {
    /* 類別的宣告 */
}

```

不過從這裡可以稍微看出類別繼承與類別實踐介面的差異性：

子類別只能繼承一個父類別。⁸

任何類別的宣告可以實踐多個介面。

所以一旦使用類別繼承的話，父類別功能要是被改掉，子類別也就很容易壞掉，這就是時常聽到的程式耦合度太高以至於很難開發下去的原因。

⁸ 通常類別繼承只能繼承一個父類別而已，但少數語言支援多重類別繼承，譬如 Python 語言的 Multiple Inheritance 就是一種。

介面具備這種可以組來組去、類別可以實踐多個介面的特性，耦合度降低，相對來說就不會把程式寫得死死的。

另外，任何類別實踐介面時，以本條目一開始出現的介面 `CatInterface` 與類別 `Cat` 為例，兩者關係為類別 `Cat` 實踐介面 `CatInterface`。

```
class Cat implements CatInterface { /* 實踐過程略 ... */ }
```

由於從類別 `Cat` 建構出來的實體符合介面 `CatInterface` 描述的規格，因此任何註記為型別 `CatInterface` 的變數可以被指派類別 `Cat` 的值：

```
const cat: CatInterface = new Cat('Julia', 'Scottish Fold');
```

此外，如果遇到類別繼承的情形，若父類別有實踐特定的介面：

```
/* 父類別的宣告 */
class Parent implements ParentInterface { /* 實踐過程略 ... */ }

/* 子類別繼承父類別 */
class Child extends Parent { /* 實踐過程略 ... */ }
```

由於子類別繼承自父類別，所以子類別擁有父類別所有開放狀態（與保護狀態）的成員，並且父類別符合 `ParentInterface` 介面的規格，因此子類別自然而然也有 `ParentInterface` 描述的規格，所以以下的程式碼是成立的。

```
const child: ParentInterface = new Child(/* 建構子函式參數 ... */);
```

簡而言之，子類別繼承自父類別外，也順便符合父類別所實踐的介面規格。

»》 類別宣告與介面實踐 Class Declaration & Interface Implementation

1. 類別雖然只能繼承單個父類別，但類別可以實踐多種不同的介面。
2. 類別必須宣告介面裡的規格為開放狀態（Public）的成員。
3. 類別宣告時實踐介面，使用關鍵字 `implements`。

► 6.4 詭異的 TypeScript 函式參數型別檢測機制

這裡要探討一個 TypeScript 檢測機制裡的特例，也是網路上最多人覺得困惑的東西，因此特地使用一個條目來講解。

6.4.1 明文 JSON 物件作為函式參數

以下面的程式碼為例，宣告某個函式 `printInfo`，其參數接收特定介面型別 `PersonalInfo` 的值：

```
interface PersonalInfo {
    name: string;
    age: number;
    interest: string[];
}

function printInfo(info: PersonalInfo) {
    console.log(`"${info.name}" is ${info.age} years old.`);
    console.log(`"${info.name}" is interested in ${info.interest.join(', ')}.`);
}
```

根據條目 6.3.1 講過的東西，普通的 JSON 物件若符合介面要求的格式，理所當然會正常地通過型別檢測：

```
// printInfo 函式接收的參數必須為 PersonalInfo 型別的值
printInfo({
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
});

// => Maxwell is 18 years old.
// => Maxwell is interested in drawing, programming.
```

如果輸入的參數，物件多一個屬性（或少一個屬性），由於不符合介面 `PersonalInfo` 描述的情形，就會出現錯誤訊息。（如圖 6-6）

```
function printInfo(info: PersonalInfo) {  
    // Argument of type '{ gender: string; name: string; age: number; interest:  
} string[]; }' is not assignable to parameter of type 'PersonalInfo'.  
    Object literal may only specify known properties, and 'gender' does not exist  
    in type 'PersonalInfo'. ts(2345)  
//  
pr Peek Problem No quick fixes available  
gender: 'Male',  
// 多出一個 gender 屬性  
name: 'Maxwell',  
age: 18,  
interest: ['drawing', 'programming'],  
});
```

由於多出一個 gender 屬性，判定不符合介面 PersonalInfo 的規格

圖 6-6 明文 JSON 物件的值，多出規格原先沒有規範的屬性就會出現錯誤

這應該看起來是理所當然的行為。

6.4.2 變數作為函式參數

如果遇到以下的情形，將物件的值指派到不經過註記的變數 maxwell 中，並且將該變數作為參數傳入函式 printInfo 裡。

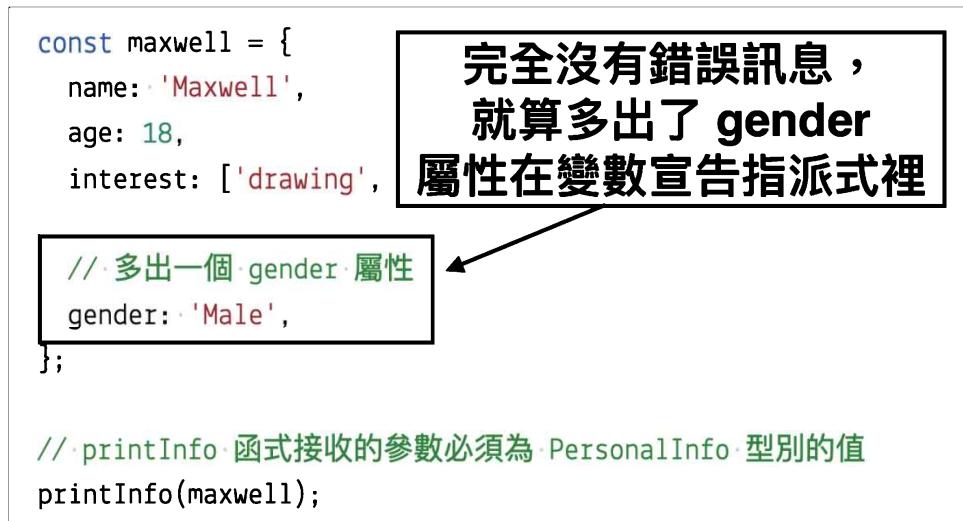
```
const maxwell = {  
    name: 'Maxwell',  
    age: 18,  
    interest: ['drawing', 'programming'],  
};  
  
// printInfo 函式接收的參數必須為 PersonalInfo 型別的值  
printInfo(maxwell);  
  
// => Maxwell is 18 years old.  
// => Maxwell is interested in drawing, programming.
```

以上的範例理所當然也會成立，不過將變數 maxwell 新增介面 PersonalInfo 沒有規範的規格時：

```
const maxwell = {  
    name: 'Maxwell',
```

```
age: 18,  
interest: ['drawing', 'programming'],  
  
// 新增 gender 屬性，但該屬性並沒有規範到 PersonalInfo 介面  
gender: 'male',  
};  
  
// printInfo 函式接收的參數必須為 PersonalInfo 型別的值  
printInfo(maxwell);  
  
// => Maxwell is 18 years old.  
// => Maxwell is interested in drawing, programming.
```

以上的程式碼反而不會發生條目 6.4.1 中，如圖 6-6 的錯誤訊息。(如圖 6-7)



```
const maxwell = {  
    name: 'Maxwell',  
    age: 18,  
    interest: ['drawing',  
        // 多出一個 gender 屬性  
        gender: 'Male',  
    ];  
  
    // printInfo 函式接收的參數必須為 PersonalInfo 型別的值  
    printInfo(maxwell);
```

圖 6-7 整個明文物件抽離出來，指派到變數並作為參數傳入函式時就不會出現錯誤

關鍵差異在：

明文物件型別的值作為參數時，就會是嚴格檢測模式，代表明文物件的規格必須完全符合參數的型別結構（指得也就是介面的規格）。

若函式的參數被傳入非明文物件型別以外的東西，例如變數或表達式，就會變成鴨子型別的檢測模式（Duck-Typing）。

6.4.3 鴨子型別檢測模式 Duck-Typing

「走路像鴨子、叫得也像鴨子的話，那麼它就是鴨子。」

以上的這句話就是一般對於鴨子型別的描述⁹。

鴨子型別的概念很簡單，轉換成 TypeScript 的想法，只要物件值的結構至少符合特定介面的規格，那麼該物件就是符合該介面的規格。

以下為介面 Duck 與變數 cat 的宣告程式碼。

```
interface Duck {  
    noise: string;  
    feetCount: number;  
}  
  
const cat = {  
    noise: 'Meow meow meow~~~~~',  
    feetCount: 4,  
  
    // 新增 nocturnal (夜行性) 屬性，但該屬性不屬於 Duck 介面  
    nocturnal: true,  
};
```

對於以上的宣告手法，變數 cat 儘管比起介面 Duck 多出了一個屬性，但是 cat 本身也擁有介面 Duck 規範的東西，所以 cat 仍然可以被當成 Duck。

所以以下這一行是反而是成立的，因為指派為變數時，會基於鴨子型別的檢測手法通過這一行變數宣告指派式：

```
const duck: Duck = cat; // ← 這一行不會出錯，因為是基於鴨子型別的判斷準則
```

但如果你是強行註記原先宣告的變數 cat 為 Duck 介面時，就代表物件值必須嚴格實踐介面 Duck 所規範的型別，因此就會出現錯誤訊息。

⁹ 鴨子型別模式，參見維基百科：https://en.wikipedia.org/wiki/Duck_typing。

```
const cat: Duck = {           // ← 變數被註記時，就會轉換成嚴格檢測模式
    noise: 'Meow meow meow~~~~~',
    feetCount: 4,
    // 新增 nocturnal (夜行性) 屬性，但該屬性不屬於 Duck 介面
    nocturnal: true,
};
```

同理，根據條目 6.4.2 的情形：

```
printInfo(maxwell); // ← 接收值為變數或表達式，為鴨子型別檢測模式

printInfo({           // ← 接收值為明文物件型別，為嚴格檢測模式，此範例會出現錯誤
    name: 'Maxwell',
    age: 18,
    interest: ['drawing', 'programming'],
    // 多出 PersonalInfo 原先沒規範的屬性，造成錯誤
    gender: 'Male',
});
```

函式接收參數若為明文物件型別，就必須嚴格遵守參數型別的格式或規格；若為普通變數或表達式，則檢測基準變成鴨子型別檢測模式。

»》嚴格檢測模式 v.s. 鴨子型別檢測模式

1. 嚴格檢測模式代表物件的值之屬性規格必須完全符合介面的規範。
2. 鴨子型別檢測模式代表物件的值之屬性規格至少符合介面的規範。
3. 一般變數或表達式若指派到被註記的變數亦或者是作為函式參數時，評判基準為鴨子型別模式；若為明文物件型別時，則為嚴格檢測模式。

► 6.5 型別化名 V.S. 介面

學習 TypeScript 絕對會碰到的問題一定是分不清型別化名與介面宣告的差異性，本條目就來了解一下它們兩者的各自的意義與使用情境差異。

6.5.1 型別化名推論時會“蒸發掉”

基本上一般型別化名與介面宣告方式看起來都是在宣告物件格式的型別：

```
/* 型別化名的宣告 */
type ObjType = {
    /* 規格內容 ... */
};

/* 介面的宣告 */
interface ObjInterface {
    /* 規格內容 ... */
}
```

型別化名可以表示任何複雜型別的組成結構，但精確一點的說法是一—型別化名單純就只是幫該型別結構取名字而已。以下面的程式碼範例為例：

```
type NumberOrString = number | string;

let ex1: NumberOrString = 123;
let ex2: number | string = ex1;
```

型別化名為 `NumberOrString` 為複合型別 `number | string` 的名稱，也就是說上面的程式碼完全等效於：

```
let ex1: number | string = 123;
let ex2: number | string = ex1;
```

所以宣告型別化名並不是創建新的型別，而是用那個名字代表宣告化名背後代表的型別結構而已。官方的說法如下¹⁰：

Aliasing **doesn't actually create a new type** - it creates a **new name to refer to that type**. Aliasing a primitive is not terribly useful, though it can be used as a form of documentation.

¹⁰ 參見 TypeScript 官方文件：<https://www.typescriptlang.org/docs/handbook/advanced-types.html#type-aliases>。

以上裡面的重點翻譯的結果為：「化名不代表宣告新的型別，…，它主要是用新的名字去代表宣告的特定型別結構」。

以下面的程式碼為例：

```
/* 宣告型別化名 */
type AliasA = { num: number };
type AliasB = AliasA;
```

根據官方的說法，若型別化名僅僅是一個名稱參照背後代表的型別結構時，理應來說，化名 `AliasB` 的推論結果並不是化名 `AliasA`，而是 `AliasA` 背後代表的型別，也就是 `{ num: number }`。(推論結果如圖 6-8)

```
// 宣告型別化名 AliasA
type AliasA = { num: number };

type AliasB = {
    num: number;
// AliasA
type AliasB = AliasA;
```

**型別化名只是個名稱
代表型別結構，因此
AliasB 的代表
型別不是 AliasA**

圖 6-8 型別化名只是個名稱，推論時會將化名替代成實際型別結構

所以以上的範例等效程式碼事實上是：

```
type AliasB = { num: number };
```

介面則相反地，會在任何地方使用名稱代表推論結果。以下面的程式碼為範例：

```
/* 宣告介面 */
interface InterfaceA {
    num: number;
};

type AliasB = InterfaceA;
```

此時 `AliasB` 的推論結果就變成 `InterfaceA` 而不是整個型別結構 `{num: number}`。(參見圖 6-9)

```
/* 宣告介面 */
interface InterfaceA {
    num: number;
};

type AliasB = InterfaceA
type AliasB = InterfaceA;
```

介面的宣告就會相對而言
產生新的名稱來指涉

圖 6-9 介面的名稱就會保存起來

所以這裡可以總結，TypeScript 編譯器在推論型別的過程中，任何型別化名的宣告只是用名稱間接參照背後代表的型別，實質上在型別推論過程中，名稱會蒸發掉（作者想不到更好的詞彙，所以用“蒸發”這個很生動的動詞）。

反之，介面的名稱則會在推論過程中保存起來。

6.5.2 意義上的差別

一般的型別化名除了基本的 JSON 物件型別的結構可以宣告外，各種型別包含元組（條目 4.1）、列舉（條目 4.2）型別，甚至是原始型別（條目 3.2）或任何型別的聯集、交集複合（條目 4.4）都可以表示。

另外，既然條目 6.5.1 得知的結論是型別化名會直接蒸發掉變成該化名背後代表的型別結構，而型別結構也不可能具備變化性，相對介面可以輕易地被延展或組合的彈性而言，型別化名比較適合用來表示靜態的資料結構。比如說用型別化名的方式宣告 PersonalInfo：

```
type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};
```

這是在告訴開發者，型別化名 PersonalInfo 的結構如上，不可以多一個屬性或少一個屬性，所以結構是靜態的。

而雖然介面僅限於制定物件的規格，它沒辦法單純表示出原始型別或者是特殊型別如元組、列舉等，但由於具備條目 6.2 談論的延展與融合相關的彈性，因

此介面宣告的結構變化性相對型別化名而言是動態的。

如果你將 `PersonalInfo` 宣告成介面時：

```
interface PersonalInfo {  
    name: string;  
    age: number;  
    interest: string[];  
};
```

意義就會變成——任何東西實踐 `PersonalInfo` 介面裡的規格時，至少符合裡面規範的功能。

讀者應該稍微感覺得到意義的差異：型別化名單純只是宣告資料結構（Structure）；介面則是宣告一個系統（可為單純 JSON 物件、類別的成員規格、純函式等東西）的規格（Spec.），而規格明顯具備高度彈性，可以組來組去、拆來拆去。

所以以下的範例程式碼：

```
interface PersonalInfo {  
    name: string;  
    age: number;  
    interest: string[];  
};  
  
interface AccountInfo {  
    email: string;  
    password: string;  
    subscribed: boolean;  
};  
  
interface UserAccount extends PersonalInfo, AccountInfo {}
```

它可以很清楚地告訴使用者，想要實踐介面 `UserAccount` 這個介面，就必須要滿足兩個介面規範的規格，一個是 `PersonalInfo`、另一個是 `AccountInfo`。

不過有些讀者可能會覺得，介面雖然具備高度的延展性，但是普通的型別化名也可以利用條目 4.4 談到的複合型別中的交集形式來達到相似的功能：

```
type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};

type AccountInfo = {
    email: string;
    password: string;
    subscribed: boolean;
};

type UserAccount = PersonalInfo & AccountInfo;
```

官方確實說可以用這種方式達到相似於介面的延展功能，只是變成是型別化名的形式，但就意義層面的解讀上會變成——宣告的靜態資料結構必須同時有 `PersonalInfo` 以及 `AccountInfo` 兩種靜態資料結構。

型別化名與介面的宣告意義上的差異是有的，而且根據開放封閉準則（Open-Closed Principle）¹¹——「任何程式的設計對於擴展是開放的，但對於修改是封閉的」；也就是說，我們忌諱在程式內部裡進行修改的動作，但對於擴展功能而言是可以接受的行為。

既然鼓勵擴展功能的行為，理應來說，使用介面是更恰當的選擇。

讀者看完本章節，目前對於介面的使用上可能會有一定程度上的模糊，本書第 9 章討論到物件導向的進階篇章時就會有更多實際案例喔。

»》 型別化名 V.S. 介面

1. 型別化名規範單純的資料結構，因此宣告出來的名稱在推論過程中會蒸發掉，變成它所代表的型別結構。
2. 介面規範的是實踐功能的基本規格，因此為了達到方便的抽象化，相對型別化名而言，彈性固然比較高。

¹¹ 開放封閉準則，參見本書條目 9.2.2，亦或者參見維基百科：https://en.wikipedia.org/wiki/Open/Closed_principle。

► 本章練習

1. 普通介面的宣告與純函式介面的宣告差異性在哪？
2. 若類別實踐介面時，以下的實踐方式是可以接受的嗎？

```
interface ICircle {  
    radius: number;  
    area: number;  
}  
  
class Circle implements ICircle {  
    constructor(public radius: number) {}  
  
    get area(): number {  
        return Math.PI * (this.radius ** 2);  
    }  
}
```

3. 以下兩種介面的宣告方式會有什麼樣的效果？

```
interface Addition1 {  
    (input1: number, input2: number): number;  
}  
  
interface Addition2 {  
    addition(input1: number, input2: number): number;  
}
```

4. 以下介面的宣告方式會有什麼樣的效果？

```
interface Addition {  
    (input1: number, input2: number): number;  
}  
  
interface Addition {  
    (input1: string, input2: string): number;  
}
```

5. 以下兩種介面的宣告方式是等效的嗎？

```
interface Addition1 {
    (input1: number | string, input2: number | string): number;
}

interface Addition2 {
    (input1: number, input2: number): number;
    (input1: string, input2: string): number;
}
```

6. 宣告一個介面與型別化名的差異性在哪？

```
type T = /* ... */;
interface I { /* ... */ }
```

07

深入型別系統 III 泛用型別

泛用型別（Generic Types）算是型別系統裡，可以跟介面並列為重點的型別宣告機制；在 TypeScript 型別系統裡，幾乎任何型別都可以轉換成泛用的形式，本章節以下開始講述泛用型別的機制。

► 7.1 泛用型別的介紹 Introduction to Generic Types

7.1.1 型別參數化 Type Parameterization

泛用型別有點像是型別版本的函式的概念，最簡單的範例就是以下的型別化名 `Identity<T>` 的宣告：

```
/* 宣告簡單的泛用型別化名 Identity<T> */
type Identity<T> = T;
```

化名 `Identity<T>` 的宣告裡中的 `T` 為型別變數（Type Parameter），可以套入各種型別，譬如說：

```
let num: Identity<number> = 123;
let str: Identity<string> = 'Hello world!';
```

就等效於以下的寫法：

```
let num: number = 123;
let str: string = 'Hello world!';
```

這種型別參數化的技巧就是泛用型別的特徵。

7.1.2 推論未來情境的泛用型別

型別參數化可以輔助編譯器的靜態分析過程，造就編譯器提前預知型別的能力。（聽起來很像黑魔法）

以普通的函式型別 `TypeConversion<T, U>` 的宣告方式為例：

```
/* 宣告簡單的泛用函式型別 TypeConversion<T, U> */
type TypeConversion<T, U> = (input: T) => U;
```

該型別化名的參數有兩個，分別是 `T` 與 `U`，就代表必須接收兩種不同的型別作為該泛用型別的型別參數。比如說：

```
/* T = number, U = string */
let numberToString: TypeConversion<number, string>;

/* T = number, U = boolean */
let isPositive: TypeConversion<number, boolean>;
```

就會等效於：

```
/* T = number, U = string */
let numberToString: (input: number) => string;

/* T = number, U = boolean */
let isPositive: (input: number) => boolean;
```

因此可以用同樣的型別化名實踐出兩種不同函式型別的結果：

```
/* 數字轉型成字串結果的函式型別 */
let numberToString: TypeConversion<number, string> =
  function (input: number): string {
    return input.toString();
```

```
};

/* 數字轉型成布林值的函式型別，此為檢測數字是否為正數 */
let isPositive: TypeConversion<number, boolean> =
  function (input: number): boolean {
    return input > 0;
};
```

另外，儘管本書在條目 3.4.1 函式型別中提到，宣告函式時，參數必須要積極註記的目的是因為呼叫函式時，使用者可以填入任意型別的值，也就是說參數就會無條件被視為 Any 型別，這是使用 TypeScript 比較不樂見的情形。

然而，如果經過泛用型別的宣告與註記手法，省略輸入參數的型別註記反而是可以接受的：

```
/* 數字轉型成字串結果的函式型別 */
let numberToString: TypeConversion<number, string> = function (input) {
  return input.toString();
};

/* 數字轉型成布林值的函式型別，此為檢測數字是否為正數 */
let isPositive: TypeConversion<number, boolean> = function (input) {
  return input > 0;
};
```

原因是因為，泛用型別早就可以藉由型別參數提供的資訊，間接推論出宣告的函式之參數型別值。如圖 7-1 為函式 `isPositive` 的宣告時，參數 `input` 沒有被註記時的推論狀態。

```
/* 告知簡單的泛用函式型別 TypeConversion<T, U> */
type TypeConversion<T, U> = (input: T) => U;

/* 數字轉型成布林值的函式型別，此為檢測數字是否為正數 */
let isPositive: TypeConversion<number, boolean> = function (input) {
  return input > 0;
};
```

輸入就算沒有被積極註記，仍然可以
藉由型別參數提供的資訊間接推論出來

圖 7-1 函式宣告時的參數經由泛用型別中，提供的型別參數資訊間接推論出來

後續的條目談到更多泛用型別的形式時會一再強調這方面的特質喔。

»» 泛用型別 Generic Types

1. 泛用型別的特徵為搭配參數化（Parameterization）的型別宣告機制。
2. 泛用型別可以用單個型別宣告表示式，呈現多種類型的型別樣貌。

► 7.2 型別泛用化

7.2.1 泛用基礎型別 Generic Basic Types

除了條目 7.1.1 提到的最簡單的型別化名 `Identity<T>` 的宣告方式外，只要是任意型別的組合或宣告（除了列舉的宣告不能改成泛用型別外）基本上都是可以轉換成泛用型別的形式喔。以下是各種泛用型別的宣告模式：

```
/* 宣告簡單的泛用陣列型別 */
type TypedArray<T> = T[];
type TypedTwoDimensionalArray<T> = T[][];

/* 宣告簡單的泛用元組型別 */
type Coordinate<T> = [T, T];

/* 宣告簡單的泛用複合型別 */
type Either<T, U> = T | U;
type And<T, U> = T & U;

/* 宣告簡單的泛用 JSON 物件型別 */
type Obj<T> = { prop: T };

/* 宣告簡單的泛用 JSON 物件之可控索引型別 */
type Dictionary<T> = { [key: string]: T };
```

事實上，TypeScript 本身有內建一個名為 `Array<T>` 的泛用型別化名，它的宣告方式就是以上的範例程式碼中的型別化名 `TypedArray<T>` 的宣告方式。也就是說，條目 3.5 中討論的陣列註記行為時，以下的寫法在 TypeScript 裡都是可以

接受的：

```
/* 以下的宣告註記寫法都是正確的 */
let numberArray1: number[]      = [1, 2, 3, 4, 5];
let numberArray2: Array<number> = [1, 2, 3, 4, 5];
```

另外，型別化名的宣告，當然也可以跟其他的泛用型別進行組合，譬如說，宣告二維陣列的型別就可以這麼宣告：

```
type TwoDimensionalArray<T> = Array<Array<T>>;
```

》》泛用型別化名的宣告 Generic Type Alias Declaration

宣告泛用型別化名時，可以在型別化名名稱後方直接接上型別參數宣告，必須使用符號 `<>` 包圍住。

7.2.2 泛用函式型別 Generic Function Types

函式型別的泛用形式也已經在條目 7.1.2 中的函式型別 `TypeConversion<T, U>` 的宣告展示過了。

不過這裡還是稍微說一下，由於函式型別分成參數的型別與輸出的型別宣告，而型別化名中的型別參數可以出現在函式的參數與輸出的宣告部分。

```
/* 型別參數可以宣告在函式的參數與輸出結果 */
type IdentityTypedFunction<T> = (input: T) => T;

/* 函式的參數輸入與輸出型別當然也可以為不同的型別參數 */
type TypeConversion<T, U> = (input: T) => U;
```

》》泛用函式型別的宣告 Generic Function Type Declaration

1. 宣告泛用函式型別時，可以在函式名稱後方直接接上型別參數宣告，必須使用符號 `<>` 包圍住。
2. 函式型別的型別參數除了可以在函式參數的宣告中使用外，輸出型別也可以使用型別參數。

7.2.3 泛用類別宣告 Generic Class Declaration

泛用型別的機制如果搭配類別的話，將會變得超級無敵非常實用！

事實上，陣列可以用泛用類別的宣告方式模擬出來：

```
class TypedArray<T> {
    constructor(public elements: T[]) {}

    public at(index: number): T {
        return this.elements[index];
    }

    public map(func: (input: T) => T): TypedArray<T> {
        const result = new TypedArray<T>([]);

        for (let i = 0; i < this.elements.length; i += 1) {
            const currentElement = this.at(i);
            const mappedResult = func(currentElement);
            result.elements.push(mappedResult);
        }

        return result;
    }

    // 可以自由宣告更多不同的成員 ...
}
```

以上的 `TypedArray<T>` 類別具備一個型別變數 `T`，裡面的成員也可以使用該型別參數。因此若建構型別為 `TypedArray<number>` 的實體，則它會擁有的成員對應型別如下面所列：

1. 成員變數 `elements` 對應的型別為 `number[]`
2. 成員方法 `at` 對應的型別為 `(index: number) => number`
3. 成員方法 `map` 對應的型別為 `(func: (input: number) => number) => TypedArray<number>`

其實就是暴力地將型別變數 `T` 取代為 `number` 而已。

以上宣告的 `TypedArray<T>` 的使用範例程式碼如下：

```
let numberArray = new TypedArray<number>([1, 2, 3, 4, 5]);

console.log(numberArray.at(2));
// => 3

// 將 numberArray 內部的所有元素乘以二，並建立新的 TypedArray<number> 實體
console.log(
  numberArray
    .map(function (x) { return x * 2 })
    .elements
);
// => [2, 4, 6, 8, 10]
```

另外，根據條目 7.1.2 講述的特性，如果型別參數被指定為 `number` 型別，但 `TypedArray<number>` 初始化時，填入建構子函式的參數型別不符合應該要對應的 `number[]` 時，就會出現錯誤訊息喔！（如圖 7-2）

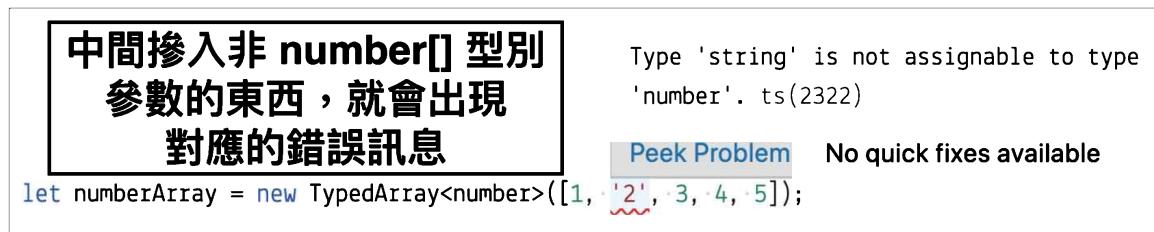


圖 7-2 填入 `(number | string)[]`（而非 `number[]`）出現的錯誤訊息

另外，如果是以下這一行的宣告方式：

```
let numberArray = new TypedArray([1, 2, 3, 4, 5]);
```

就算你不指定型別參數對應的型別，TypeScript 編譯器有能力可以反推該型別參數對應的型別。（如圖 7-3）

**根據輸入參數的型別為 `number[]`，
TypeScript 反推型別參數 T 代表的型別為 `number`**

```
let numberArray: TypedArray<number>
let numberArray = new TypedArray([1, 2, 3, 4, 5]);
```

圖 7-3 泛用型別的推論可以從被指派的值反推型別參數對應的型別

泛用型別推論絕對不是單向的，實質上是雙向的。

不過泛用型別跟類別結合起來時，由於牽扯到眾多型別結構，因此要非常注意寫法，例如，將剛剛的 `TypedArray<T>` 範例多宣告一個新的成員：

```
class TypedArray<T> {
    constructor(public elements: T[]) {}

    public at(index: number): T {
        return this.elements[index];
    }

    public map(func: (input: T) => T): TypedArray<T> {
        // 實踐過程與前個案例的 map 方法相同
    }

    /**
     * 此 mapToType<U> 成員方法也有另一個型別參數 U，跟前面的成員方法 map 用途差在
     * map 方法會建立新的 TypedArray<T>，而 mapToType<U> 則是建立新的 TypedArray<U>
     */
    public mapToType<U>(func: (input: T) => U): TypedArray<U> {
        const result = new TypedArray<U>([]);

        for (let i = 0; i < this.elements.length; i += 1) {
            // ... 實踐過程跟 map 方法裡面的 for 迴圈相同
        }

        return result;
    }
    // 可以自由宣告更多不同的成員 ...
}
```

使用差異可以從以下的程式碼比較：

```
let numberArray = new TypedArray<number>([1, 2, 3, 4, 5]);

// 將 numberArray 內部的所有元素乘以二，並建立新的 TypedArray<number> 實體
console.log(
    numberArray
        .map(function (x) { return x * 2 })
```

```
.elements
);
// => [2, 4, 6, 8, 10]

// 但如果是想要將所有的元素轉成 string 型別時，不能使用 map 方法，一定要使用
// mapToType<string> 這個成員方法喔！
console.log(
    numberArray
        .mapToType<string>(function (x) { return x.toString(); })
        .elements
);
// => ['1', '2', '3', '4', '5']
```

至於類別這樣的宣告方式：

```
class TypedArray<T> {
    // 略 ...

    /**
     * 以下的 mapToType 型別變數 T 會覆蓋掉 TypedArray<T> 類別的型別變數 T
     */
    public mapToType<T>(func: (input: T) => T): TypedArray<T> {
        // 略 ...
    }
}
```

事實上是可以的，但是非常不建議這麼做，此時的 T 就不代表類別的型別參數，意義上是錯的，這種寫法會讓很多人賭爛。

另外，牽扯到類別的繼承時，第一種情況是：

```
class Child extends Parent<TYPE> { /* 子類別的宣告 */ }
```

這種純子類別繼承泛用父類別時，父類別的型別參數必須指定型別，比如：

```
class Child extends Parent<number> { /* 子類別的宣告 */ }
```

但你不能寫成在宣告型別參數的寫法：

```
class Child extends Parent<T> { /* 子類別的宣告 */ }
```

因為宣告的重點是子類別，應該是由子類別宣告型別參數後，將子類別的型別參數套入父類別（又或者是父類別填入另一種已知型別）。

意思是說，以下兩種寫法都是可以接受的：

```
/* 子類別的宣告中，型別參數 T 可以套入父類別 */
class Child<T> extends Parent<T> { /* 子類別的宣告 */ }

/* 子類別的宣告中，父類別也可以選擇套入其他已知型別 */
class Child<T> extends Parent<number> { /* 子類別的宣告 */ }
```

此外，抽象類別¹的宣告理所當然地可以轉換成泛用的形式。以之前的範例抽象類別 Sorter 為例：

```
abstract class Sorter {
    constructor(public input: any) {}

    public sort() {
        /* 泡泡演算法 */
        for (let i = this.input.length; i > 0; i -= 1) {
            for (let j = 0; j < i - 1; j += 1) {
                if (this.compare(j, j + 1)) this.swap(j, j + 1);
            }
        }
    }

    abstract compare(index1: number, index2: number): boolean;
    abstract swap(index1: number, index2: number): void;
}
```

還記得本範例還卡在一個美中不足的地方嗎？

```
abstract class Sorter {
    constructor(public input: any) {}

    // 其他成員略 ...
}
```

¹ 抽象類別，請參見本書條目 5.2.7。

以上的成員變數 `input` 為 `Any` 型別若想要避免的話，我們可以將抽象類別 `Sorter` 轉換成泛用的形式，也就是 `Sorter<T>`：

```
abstract class Sorter<T> {
    constructor(public input: T) {}

    public sort() {
        /* 泡泡演算法，注意裡面的 this.input.length 被改寫為 this.length */
        for (let i = this.length; i > 0; i -= 1) {
            for (let j = 0; j < i - 1; j += 1) {
                if (this.compare(j, j + 1)) this.swap(j, j + 1);
            }
        }
    }

    /**
     * 由於 input 的 T 中的 T 不一定為陣列型別，input 自然而然就不一定
     * 擁有 length 屬性，因此必須強制宣告 length 成員的規格
     */
    abstract length: number;

    abstract compare(index1: number, index2: number): boolean;
    abstract swap(index1: number, index2: number): void;
}
```

因此，條目 5.2.7 裡，宣告數字型陣列的排序器類別 `NumberSorter` 的繼承方式就可以寫成：

```
class NumberSorter extends Sorter<number[]> {
    constructor(input: number[]) {
        super(input);
    }

    // 請讀者注意：此範例必須多宣告以下的 length 取值方法
    get length() { return this.input.length; }

    // 其餘宣告方式一模一樣 ...
}
```

同理，`StringSorter` 的繼承方式就可以寫成：

```
class StringSorter extends Sorter<string> {
    constructor(input: string) {
        super(input);
    }

    // 請讀者注意：此範例必須多宣告以下的 length 取值方法
    get length() { return this.input.length; }

    // 略 ...
}
```

》》泛用類別的宣告 Generic Class Declaration

1. 宣告泛用類別時，可以在類別名稱後方直接接上型別參數宣告，必須使用符號 `<>` 包圍住。
2. 泛用類別的型別參數除了可以在類別內部使用外，也可以填入繼承的父類別（若為泛用類別的話）或者是實踐的介面（若為泛用介面的話）。

7.2.4 泛用介面宣告 Generic Interface Declaration

介面也是可以宣告成泛用的形式，例如將條目 7.2.3 裡的 `TypedArray<T>` 轉換成介面的宣告格式如下：

```
interface TypedArray<T> {
    elements: Array<T>;
    at(index: number): T;
    map(func: (input: T) => T): T;
    mapToType<U>(func: (input: T) => U): T;
}
```

此外，介面由於擁有延展的特性，所以它也可能會出現以下的宣告方式：

```
/* 普通介面延展自泛用介面 */
interface I1 extends I2<number> {
    /* 介面規格宣告略 ... */
}
```

```
/* 泛用介面延展自另一個泛用介面 */
interface I3<T> extends I4<T> {
    /* 介面規格宣告略 ... */
}
```

它跟泛用類別繼承時有類似的想法，若是以上面的案例中的前者，普通介面延展自泛用介面時，泛用介面必須指定特定型別，因為型別參數必須為主要的普通介面進行宣告。

而如果是後者的案例，宣告泛用介面擁有型別參數的宣告時，才可以將該參數套入延展的泛用介面。

另外，通常介面的宣告會伴隨類別實踐該介面，如果是純類別的宣告，沒有型別參數提供時，實踐的泛用介面之型別參數必須為已知型別：

```
/* 普通類別實踐泛用介面時，泛用介面必須為已知型別 */
class C implements I<number> {
    /* 類別成員規格宣告略 ... */
}
```

若類別本身為泛用的宣告形式，則實踐的泛用介面可以選擇填入已知型別外，也可以填入類別宣告的型別參數：

```
/* 泛用類別實踐泛用介面時，泛用介面可為已知型別外，也可以為泛用類別的型別參數 */
class C1<T> implements I<number> {
    /* 類別成員規格宣告略 ... */
}

class C2<T> implements I<T> {
    /* 類別成員規格宣告略 ... */
}
```

以上的兩種寫法都是可以的，原因是型別參數的宣告主要是綁定在類別宣告身上。

此外，類別由於可以同時間進行繼承與實踐介面，因此你也可能會看到以下的組合：

```
/* 普通類別繼承泛用類別與實踐泛用介面 */
class Child extends Parent<number> implements Interface<number> {
    /* 類別成員規格宣告略 ... */
}

/* 泛用類別繼承泛用類別與實踐泛用介面 */
class GenericChild<T> extends Parent<T> implements Interface<T> {
    /* 類別成員規格宣告略 ... */
}
```

別忘了，類別一次可以實踐多種不同的介面，反正讀者只要知道類別與介面有哪些組合寫法，就有可能可以宣告成泛用的形式。

► 7.3 型別參數額外功能

7.3.1 預設型別參數 Default Type Parameter

既然函式的參數具備預設值的功能，對應的泛用型別之型別參數也有類似的機制。以下面的程式碼為範例：

```
/* 宣告簡單的泛用 JSON 物件之可控索引型別，其中 Dictionary 的型別參數值預設為字串
   型別 */
type Dictionary<T = string> = { [key: string]: T };
```

預設型別的語法跟函式參數的預設值的語法差不多，都是用等號並且指派預設型別值到型別參數旁。因此以下的範例程式碼，兩種宣告方式都會得到一模一樣的結果：

```
/* Dictionary 的型別參數為 string */
type StringDictionary1 = Dictionary<string>;  
  
/* Dictionary 的型別參數為預設型別值，也就是 string */
type StringDictionary2 = Dictionary;
```

因此，如果變數被註記為沒有填入型別參數值的 **Dictionary** 的型別，若出現非字串型別（也就是預設型別）的值，就會出現如圖 7-4 的錯誤訊息。

```
const stringDictionary: Dictionary<string> = {  
    hello: 'world',  
    num: 123,  
}; // (property) num: number
```

Dictionary 預設的型別為 string，因此對應值若為非字串型別值就會出現錯誤

```
Type 'number' is not assignable to type 'string'. ts(2322)
```

```
hello-world.ts(2, 33): The expected type comes from this index signature.
```

```
Peek Problem No quick fixes available
```

圖 7-4 Dictionary 的型別參數預設為字串型別，裡面的值為數字型別出現的警訊

此外，跟函式的預設參數概念相向，型別參數的預設值不能斷斷續續地預設，而且要設置在宣告型別參數部分的尾端。因此以下除了最後的範例外，其他宣告方式是錯誤的：

```
/* 尾端沒有指派預設型別 */  
type Alias1<T = string, U = number, V> = /* 泛用型別宣告 */;  
  
/* 斷斷續續地制派預設型別 */  
type Alias2<T = string, U, V = number> = /* 泛用型別宣告 */;  
  
/* 可以被接受的預設型別手法 */  
type Alias3<T, U = string, V = number> = /* 泛用型別宣告 */;
```

»》預設型別參數 Default Type Parameter

1. 宣告任何泛用型別時，皆可以指派預設型別到宣告的型別參數。
2. 型別參數的預設必須連續性地設置在尾端。

7.3.2 型別參數限制 Type Constraint

泛用型別通常在宣告時，不一定希望使用者填任何型別；也就是說，我們會想要在型別參數部分設置一個可以填入的型別範圍，這個被稱之為型別參數限制。

假設條目 7.3.1 中的 `Dictionary<T>` 的範例中，若想限制型別參數只能填非 `null` 或 `undefined` 的原始型別值時，也就是限制 `Dictionary<T>` 只能為數字、

字串或布林值時，可以參考以下範例：

```
/* NonNullPrimitives 為非 null 或 undefined 的原始型別 */
type NonNullPrimitives = number | string | boolean;

/* 宣告簡單的泛用 JSON 物件之可控索引型別，限制型別參數只能為 NonNullPrimitives */
type Dictionary<T extends NonNullPrimitives> = { [key: string]: T };
```

這樣就可以防止使用者填非 `NonNullPrimitives` 範疇以外的型別作為泛用型別 `Dictionary<T>` 的型別參數值。(如圖 7-5)

```
/* 在 Dictionary 的型別參數部分填入 { message: string; } 型別會出現錯誤訊息 */
type ObjectDictionary = Dictionary<{ message: string; }>;
```

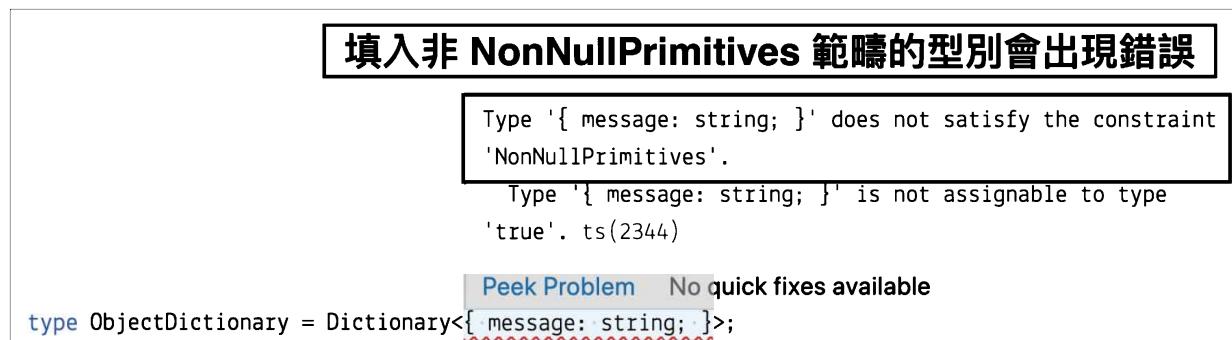


圖 7-5 填入的型別參數限制住後，使用者就不能夠亂填任何型別

限制的部分也可以用介面的方式進行參數限制，假設宣告的泛用型別想要的物件至少符合某些規格，我們可以這麼做：

```
interface ContainsDescription {
  description: string;
}

/* 任何填入 logDetail 函式的參數必須符合 ContainsDescription 介面的規格 */
function logDetail<T extends ContainsDescription>(something: T) {
  console.log(something.description);
}
```

所以任何有 `description` 屬性的物件都可以被填到該函式裡，但讀者可能會想說，以上這一段 `logDetail` 宣告到底跟以下 `logDetailInterface` 的宣告方式差在哪裡？

```
function logDetailInterface(something: ContainsDescription) {
    console.log(something.description);
}
```

前者 `logDetail` 是指，凡是至少有符合 `ContainsDescription` 介面的物件都可以指派到該函式李，這是使用型別參數限制最常見的宣告手法。

```
/* logDetail 可以填入至少符合 ContainsDescription 介面的值，這一行不會出現錯誤
訊息 */
logDetail({ hello: 'world', description: 'Hello world!' });
```

然而，如果是 `logDetailInterface` 的函式宣告寫法，參數直接被註記為 `ContainsDescription` 介面就會回歸到條目 6.4 討論的狀態——如果填入該函式的東西是變數或表達式就會變成鴨子模式的檢測標準，但填入明文 JSON 物件到函式裡，則必須完全符合 `ContainsDescription` 介面的規格，為嚴格局模式檢測手法。

```
/**
 * logDetailInterface 的宣告方式，填入的明文 JSON 物件參數則必須要變成完全符合
 * ContainsDescription 介面描述的規格
 */
logDetailInterface({ hello: 'world', description: 'Hello world!' });

/* 若填入的值為變數或表達式，評判標準為鴨子型別 */
const obj = { hello: 'world', description: 'Hello world!' };
logDetailInterface(obj);
```

以上的範例會出現的錯誤訊息如圖 7-6。

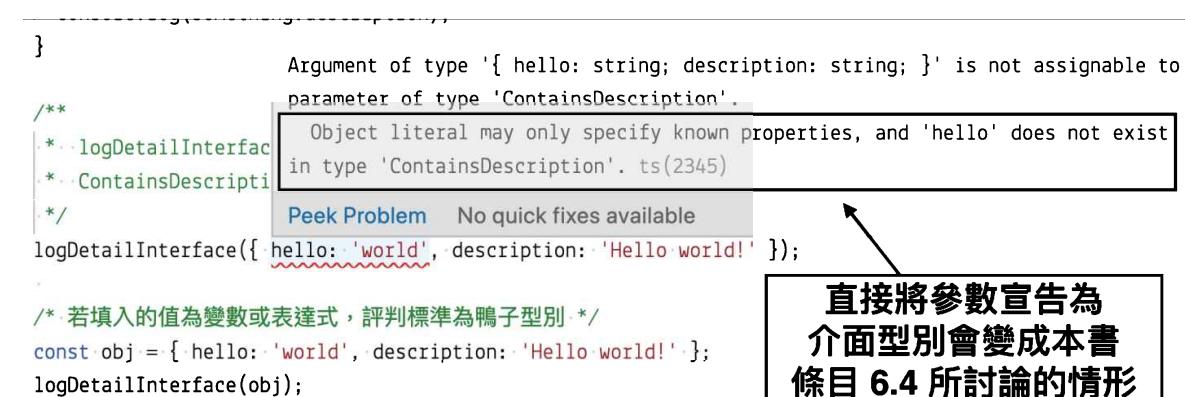


圖 7-6 明文 JSON 物件填入介面型別的參數時，必須完全符合介面規格

此外，我們當然可以同時結合預設型別值與型別限制技巧，寫法是先限制型別後指派預設型別值：

```
/* 先限制為 NonNullPrimitives 的限制，而後預設為 string 型別 */
type Dictionary<T extends NonNullPrimitives = string> = { [key: string]: T };
```

》》型別參數限制 Type Constraints

1. 泛用型別之型別參數宣告時，可以使用關鍵字 **extends** 限制被填入的參數型別範圍。
2. 若想要在限制範圍的同時指派預設型別值，記得先限制範圍、後指派預設值的順序。
3. 使用型別參數限制時，若限制型別為介面時，就代表填入型別參數之型別只要至少符合介面的規格就算達標。

► 本章練習

1. 檢視以下的範例，若想將 `isTruthy` 函式型別改成泛用的形式，要求輸入參數 `input` 可以為設定成特定的型別時，要如何修改？

```
type isTruthy = (input: any) => boolean;
```

2. 承上題，若想要限制宣告的型別參數範圍只能為原始型別（Primitive Type）類型的值，則該如何修改泛型宣告？
3. 一般使用 JavaScript 陣列（也就是 `Array`）相關的方法時，其中一個最常用到的方法是 `Array.prototype.forEach`，為陣列的元素迭代方法。請在編輯器（也就是 VSCode）裡檢視並且實驗以下的程式碼：

```
const arr = [1, 2, 3, 4, 5];

// 迭代每個元素並且將每個元素印出來
arr.forEach(function callback(el) {
    console.log(el);
});
```

試著想想看，為何裡面的回呼函式 `callback` 之參數不用註記，仍然不會出現錯誤訊息？

4. 本書條目 3.3.5 JSON 物件型別的唯讀屬性部分，講到 `Readonly<T>` 這個 TypeScript 內建的應用型別（Utility Type），它會將所有物件裡的屬性全部轉換成唯讀狀態，請問等效的宣告方式是什麼？
5. 請問以下的泛用介面的宣告方式，`method1`、`method2` 與 `method3` 三種規格的差異性在哪？

```
interface I<T> {
    method1(input: T): T;
    method2<U>(input: U): T;
    method3<T>(input: T): T;
}
```

6. 請問以下兩種函式的宣告方式，差異性在哪裡？

```
interface Lengthwise {
    length: number;
}

function ex1(input: Lengthwise) { /* 函式實踐內容略 ... */ }
function ex2<T extends Lengthwise>(input: T) { /* 函式實踐內容略 ... */ }
```

7. 【進階題】以下的程式碼用原生 JavaScript 宣告一個 `valuesOfJSON` 函式，輸入一個 JSON 物件，輸出為 JSON 物件的值的陣列：

```
function valuesOfJSON(input) {
    const keys = Object.keys(input); // ← 將 JSON 的鍵 (key) 全部拔出來
    const result = [];
    for (let i = 0; i < keys.length; i += 1) {
        result.push(obj[keys[i]]);
    }
    return result;
}
```

函式 `valuesOfJSON` 的使用範例如下：

```
// valuesOfJSON 會將 JSON 物件的值全部都輸出
```

```
console.log(valuesOfJSON({ hello: 'world', num: 123 }));
// => ['world', 123]

console.log(valuesOfJSON({ prop1: true, prop2: 100, prop3: 'Max' }));
// => [true, 100, 'Max']
```

試著將函式 valuesOfJSON 轉換成 TypeScript 的宣告寫法，除了必須註記輸入外，輸出也必須註記，格式如下：

```
function valuesOfJSON(input: /* 輸入型別 */): /* 輸出型別 */ {
    /* 實踐過程略 ... */
}
```

如有必要可以將 valuesOfJSON 宣告成泛用型式，也可以額外宣告其他型別化名或介面進行輔助。(不過本題基本上很難不用到泛用型式的宣告)

08

TypeScript 模組系統

本書已經將 TypeScript 的型別系統裡的化名、類別與介面等基本型別都涵蓋得差不多了，基礎篇剩下的最後一片拼圖就是能夠讓程式中的型別模組化的機制。如果本身是熟悉 ES6 Import 與 Export 語法的讀者，應該可以很快看過條目 8.1。

► 8.1 ES6 Import / Export 模組語法

8.1.1 型別宣告與實踐過程的分離

TypeScript 比起原生 JavaScript 多出了型別系統，因此編寫程式的過程中，型別宣告的部分與主程式塞在同個檔案，難免覺得有些複雜：

```
/* 使用條目 3.6.3 講到的互斥聯集 */
type Shape = Circle | Rectangle | Triangle;

type Circle = {
    type: 'Circle';
    radius: number;
};

type Rectangle = {
    type: 'Rectangle';
    // 略 ...
}
```

```
};

type Triangle = {
    type: 'Triangle';
    // 略 ...
};

function areaOfShape(info: Shape): number { /* 實踐過程略 ... */ }
```

ECMAScript 在第六版提供了基礎的 Import 與 Export 相關的語法，使得程式碼可以拆分成多個 JavaScript 檔案。同理，TypeScript 的型別化名與介面也是可以被引入與輸出。

以下的範例為將上面的程式碼拆成型別宣告部分（假設檔案位置與命名為 ./Shape.ts）與實踐部分（./index.ts）的結果：

```
// 檔案位置：./Shape.ts
/* 使用 export 關鍵字將型別輸出 */
export type Shape = Circle | Rectangle | Triangle;

export type Circle = {
    type: 'Circle';
    radius: number;
};

export type Rectangle = {
    type: 'Rectangle';
    // 略 ...
};

export type Triangle = {
    type: 'Triangle';
    // 略 ...
};

// 檔案位置：./index.ts
/* 使用 import 關鍵字將型別輸入，記得要參照正確的檔案位置喔！ */
import { Shape } from './Shape';

function areaOfShape(info: Shape): number { /* 實踐過程略 ... */ }
```

另外，如果實踐過程不僅僅會用到 `Shape` 這個型別時，我們當然也可以將其他需要用到的型別引入：

```
// 檔案位置： ./index.ts
/* 可以引入多種不同的型別 */
import { Shape, Circle, Rectangle, Triangle } from './Shape';

function areaOfShape(info: Shape): number { /* 實踐過程略 ... */ }
```

以下假設將 `Shape.ts` 裡的 `Circle` 的 `export` 關鍵字刪除；如果型別宣告的部分沒有標上 `export` 關鍵字時，該型別就是不能被輸出的狀態。(錯誤訊息如圖 8-1)

```
// 檔案位置： ./Shape.ts
// 其他型別宣告略 ...

/* 若沒有 export 關鍵字，型別形同沒有輸出 */
type Circle = { /* 略 ... */ };
```

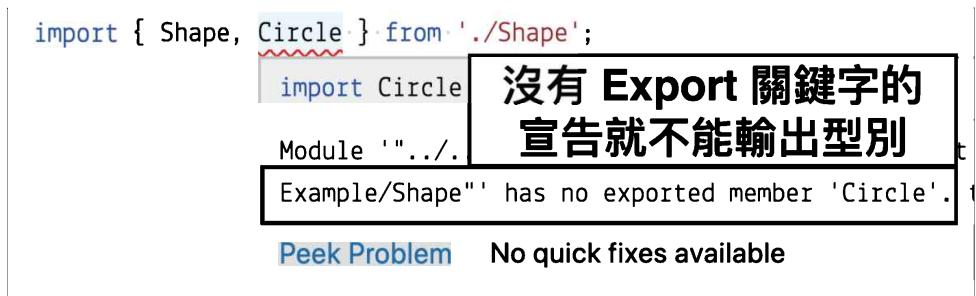


圖 8-1 引入沒有被宣告輸出的東西會出現錯誤訊息

另外，介面與類別的宣告也可以經由 ES6 Export 進行輸出的動作（輸入也是同樣的道理）：

```
/* 型別化名的輸出 */
export type Shape = Circle | Rectangle | Triangle;

/* 介面的輸出 */
export interface PersonalInfo {
    name: string;
    age: number;
    interest: string[];
}
```

```
/* 類別的輸出 */
export class UserAccount implements PersonalInfo { /* 實踐過程略 ... */ }
```

8.1.2 預設輸出 Default Export

另一種常見的輸出方式是預設輸出，宣告方式為 `export default` 兩個關鍵字的組合，比如說我們將條目 8.1.1 講到的 `Shape` 型別改成預設輸出：

```
// 檔案位置：./Shape.ts
/* 預設輸出型別為 Shape */
type Shape = Circle | Rectangle | Triangle;
export default Shape; // ← 請注意：不能使用 export default type 的宣告方式

/* 非預設的輸出型別 Circle */
export type Circle = { /* 略 ... */ };

// 略 ...
```

這裡要請讀者注意的是：你不能宣告型別的同時設置預設輸出，因此如果出現以下的寫法就會出現如圖 8-2 的錯誤訊息。

```
export default type Shape = Circle | Rectangle | Triangle;
```

The screenshot shows a TypeScript error message. The code line `export default type Shape = Circle | Rectangle | Triangle;` is highlighted with red squiggly underlines under `default`, `type`, and the entire type definition. A callout box with a black border and white background contains the text "出現宣告錯誤" (Declaration Error) in bold. Below it, the error message "any" is followed by the error code "';' expected. ts(1005)". At the bottom, another message "Cannot find name 'Shape'. ts(2304)" is shown.

圖 8-2 錯誤訊息看起來是一般的語法解析過程錯誤

事實上，圖 8-2 的錯誤訊息不是 TypeScript 在型別機制上的錯誤，而是因為以下的寫法：

```
export default type;
```

是指——`type` 並非作為型別宣告，而是單純被 JavaScript 程式碼當作變數並且設為預設輸出的寫法，所以為了澄清此狀況，你不得不拆分成兩行：

```
type Shape = Circle | Rectangle | Triangle;
export default Shape;
```

接下來，引入預設輸出的型別時，就不需要用大括弧進行引入的動作：

```
// 檔案位置：./index.ts
/* 引入預設輸出的型別時不需要大括弧 */
import Shape from './Shape';

function areaOfShape(info: Shape): number { /* 實踐過程略 ... */ }
```

這樣做可以讓模組的引入看起來像是整合過的寫法，而預設輸出的型別，在引入時命名可以任意更改：

```
// 檔案位置：./index.ts
/**
 * 引入預設輸出的型別可以更改為其他的命名；以下將 ./Shape.ts 裡預設輸出的 Shape
 * 更名為 Dimension 不會有太大差異，仍然會指涉到 ./Shape.ts 裡預設輸出的型別
 */
import Dimension from './Shape';

function areaOfShape(info: Dimension): number { /* 實踐過程略 ... */ }
```

預設輸出既然可以在引入時自定義名稱，相對代表著檔案只能預設輸出一個型別，而非多個。如果出現多個預設輸出的宣告時就會出現如圖 8-3 的錯誤訊息。

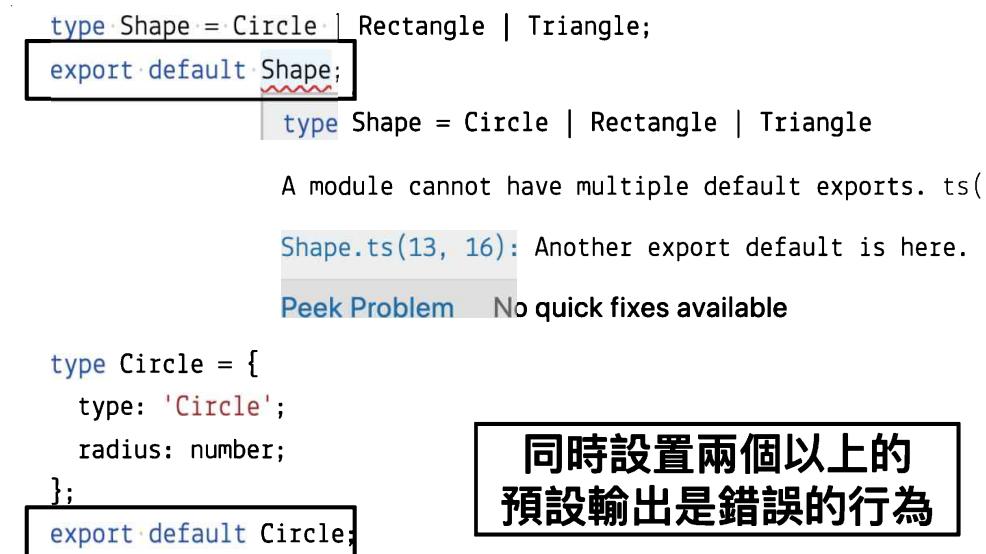


圖 8-3 一個模組不能有多個預設輸出

因為如果假設有模組涵蓋兩個預設輸出時，引入的名稱就不知要指涉哪一個預設輸出模組。

另外，你會發現模組裡的型別可以同時擁有預設輸出跟普通的輸出模式，如果想要同時引入預設輸出的型別以及需要用到的普通輸出模式的型別時，可以這麼寫：

```
// 檔案位置：./index.ts
/* 同時引入預設輸出與普通模式輸出下的東西 */
import Dimension, { Circle, Rectangle, Triangle } from './Shape';

function areaOfShape(info: Dimension): number { /* 實踐過程略 ... */ }
```

普通輸出模式下的任何東西照樣需要大括弧包住，跟條目 8.1.1 的引入方式差不多。

不過呢，預設輸出模式的型別也可以用大括弧包住，只是名稱就得用 `default` 代表，並且強制使用關鍵字 `as` 進行更名：

```
// 檔案位置：./index.ts
/* 在大括弧內引入預設輸出的東西時，只能用 default 這個名稱指涉 */
import { default as Dimension, Circle, Rectangle, Triangle } from './Shape';

function areaOfShape(info: Dimension): number { /* 實踐過程略 ... */ }
```

講到更名，引入的普通輸出模式下的型別更換名稱也是可以的，照樣使用 `as` 關鍵字，以下是簡單的範例：

```
// 檔案位置：./index.ts
/* 將引入的普通模式輸出型別更換名稱 */
import { Circle as C, Rectangle as R, Triangle as T } from './Shape';
```

這樣就可以在註記時使用更換過的型別名稱：

```
let circle: C = { /* ... 略 */ }; // ← 代表 ./Shape.ts 裡的 Circle 型別
```

8.1.3 引入所有模組的輸出 Import All

如果檔案引入的是模組所有輸出的東西時，可以用更簡潔的語法方式引入：

```
// 檔案位置：./index.ts
/* 引入模組輸出的所有東西，並且將整個東西命名為 shapes */
import * as shapes from './Shape';

// 略 ...
```

這樣如果想要使用普通模式輸出下的型別，可以這樣註記：

```
// 檔案位置：./index.ts
import * as shapes from './Shape';

// 使用 shapes.[ 輸出的型別名稱 ] 指涉
let circle: shapes.Circle = {
  type: 'Circle',
  radius: 3
};
```

另外，預設輸出的型別可以用 `default` 這個屬性來指涉：

```
// 檔案位置：./index.ts
import * as shapes from './Shape';

let circle: shapes.Circle = { /* ... 略 */ };

// 預設輸出必須要用 shapes.default 指涉
let rectangle: shapes.default = {
  type: 'Rectangle',
  width: 4,
  height: 5
};
```

不過用 `default` 這個屬性的註記方式實在太醜，或不直觀，因此會建議用條目 8.1.2 講到的方式，為預設輸出的型別取個名稱會比較好：

```
// 檔案位置：./index.ts
import Shape, * as shapes from './Shape';
```

```
// 預設輸出可以用名稱 Shape 替代 shapes.default
let rectangle: Shapes = { /* ... 略 */ };
```

8.1.4 輸出引入的模組 Export From Import

有時候寫套件會用到的型別很多種，散落在各種不同的檔案，造成引入時會不太方便，或者是寫起來不夠乾淨。比如某檔案資料夾 `shapes` 裡涵蓋三個不同的型別宣告各自獨立成一個檔案模組：

```
// 檔案位置： ./shapes/Circle.ts
export type Circle = {
    type: 'Circle';
    radius: number;
};

// 檔案位置： ./shapes/Rectangle.ts
export type Rectangle = {
    type: 'Rectangle';
    width: number;
    height: number;
};

// 檔案位置： ./shapes/Triangle.ts
export type Triangle = {
    type: 'Triangle';
    base: number;
    height: number;
};
```

這樣子使用起來會變成以下的引入方式：

```
// 檔案位置： ./index.ts
import { Circle } from './shapes/Circle';
import { Rectangle } from './shapes/Rectangle';
import { Triangle } from './shapes/Triangle';

// 略 ...
```

所以以後每新增一個跟 `shapes` 相關的型別就得再新增一行引入的語法；若有幾個檔案需要用到 `shapes` 相關的型別，就得翻來翻去，寫程式效率很明顯會降低。

因此通常會有一個檔案專門彙整相關的型別，統一輸出去；以 `shapes` 為例，我們可以額外新增 `shapes/index.ts` 的檔案，並且將所有型別進行彙整輸出：

```
// 檔案位置：./shapes/index.ts
import { Circle } from './Circle';
import { Rectangle } from './Rectangle';
import { Triangle } from './Triangle';

export { Circle, Rectangle, Triangle };
```

這樣就可以在引入這些跟 `shapes` 相關的型別，可以搭配條目 8.1.3 的技巧，固定都是全部輸出，你也不需要擔心有沒有檔案缺乏哪些 `shapes` 相關的型別。

```
// 檔案位置：./index.ts
import * as shapes from './shapes/index';

// 用 shapes.[ 型別 ] 的方式指涉
let circle: shapes.Circle = {
  type: 'Circle',
  radius: 3
};
```

另外，以上的程式碼有些地方可以簡化，第一個是：只要某個資料夾檔案命名為 `index` 時（注意：一定要是 `index` 這個名稱），從該檔案載入的路徑可以省略 `index` 這個名稱，因為檔案路徑如果找到的結果是資料夾而非檔案時，預設會認定是該資料夾內的 `index.ts` 這個檔案！

```
// 檔案位置：./index.ts
/* 由於 ./shapes 為資料夾名稱，TypeScript 編譯時認定載入檔案為 ./shapes/index.ts */
import * as shapes from './shapes';

// 略 ...
```

另外，載入與輸出的程式碼，也就是如下的樣貌：

```
// 檔案位置：./shapes/index.ts
import { Circle } from './Circle';
import { Rectangle } from './Rectangle';
import { Triangle } from './Triangle';

export { Circle, Rectangle, Triangle };
```

由於這樣的模式很常見，因此 ES6 提供了輸出載入的模組語法，也就是 `export ... from ...`，改寫方式如下：

```
// 檔案位置：./shapes/index.ts
export { Circle } from './Circle';
export { Rectangle } from './Rectangle';
export { Triangle } from './Triangle';
```

所以之後有任何型別要被新增到 `shapes` 時，就可以用一行的方式解決掉。

最後要提到的細節是，如果 `shapes` 資料夾裡的模組，每一個是用預設模式輸出時，也就是：

```
// 檔案位置：./shapes/Circle.ts
type Circle = { /* 略 ... */ };
export default Circle;

// 檔案位置：./shapes/Rectangle.ts
type Rectangle = { /* 略 ... */ };
export default Rectangle;

// 檔案位置：./shapes/Triangle.ts
type Triangle = { /* 略 ... */ };
export default Triangle;
```

這樣的話，輸出載入的語法就會變成：

```
// 檔案位置：./shapes/index.ts
export { default as Circle } from './Circle';
export { default as Rectangle } from './Rectangle';
export { default as Triangle } from './Triangle';
```

以上是常見的 ES6 模組相關的語法規則。

»» ES6 Import / Export 語法

1. ES6 Import / Export 語法為 JavaScript 標準的模組語法規範。
2. 如果程式碼複雜度提升時，可以試著將型別宣告與實踐的部分進行分離。
3. 任何模組要輸出東西時，一定要使用 `export` 關鍵字；從其他模組引入東西時，則使用 `import` 關鍵字，並且將要輸入的東西用大括弧匡住。
4. 預設輸出時，使用 `export default` 方式進行輸出；每個模組一次只能擁有一個預設輸出。
5. 引入預設輸出的東西不需要用大括弧匡住，也不一定要按照預設輸出的名稱命名。
6. 使用 `import * as <NAME> from '<MODULE>'` 的方式可以將模組裡的所有東西引入。
7. 任何普通模式輸入的東西，可以藉由 `as` 關鍵字改名。
8. 使用 `export { ... } from '<MODULE>'` 的方式可以直接輸出從模組引入的東西。
9. 若載入的模組路徑為資料夾時，預設會讀取該資料夾內的 `index.ts` 檔案的內容。

► 8.2 命名空間 Namespaces

8.2.1 TypeScript 原先提供的模組化解法

事實上，有關於模組化的語法，讀者熟悉條目 8.1 的內容即可，因為你不太會用到本條目討論到的東西，但是在翻一些套件的原始碼的過程，依然有遇到的可能性。

由於條目 8.1 討論的 ECMAScript 模組標準為第六版，出版年代為 2015 年，而 TypeScript 出生的年代更早，差不多是 2012 年左右。想當然，那時候 Import / Export 的語法都還沒正式釋出，TypeScript 理應會需要一套規範專門處理模組化相關的機制，於是就有所謂的 `namespaces` 以及 `module` 的模組宣告式語法。

官方基本上將 `module` 這種宣告方式列為 Legacy Feature，最主要原因是 Import / Export 語法已經正式地標準化，模組的概念已成 JavaScript 原生應當具備的特色¹。所以本條目會專注於被官方稱為 TypeScript-Specific (TypeScript 特色) 的命名空間。

8.2.2 命名空間的意義 Namespaces

命名空間可以拆成兩個名詞——「命名」與「空間」。(這應該是廢話)

想像一下，在 TypeScript 的專案裡引入各種不同來源的模組或套件，假設有兩個套件同時都有相同命名的型別，豈不是就產生衝突了嗎？

於是為了避免型別在命名宣告上產生衝突，於是 TypeScript 提供了模組化的宣告方式，謂之命名空間。以下宣告名為 `Shapes` 的命名模組，綜合了三種不同的型別：

```
/* 宣告命名空間 Shapes */
namespace Shapes {
    export type Circle = {
        type: 'Circle';
        radius: number;
    };

    export type Rectangle = { /* 略 ... */ };

    export type Triangle = { /* 略 ... */ };
}
```

注意，宣告命名空間時，由於它本身也是模組的一種，因此輸出型別時也需要使用關鍵字 `export`！

使用命名空間輸出的型別，跟呼叫物件屬性的概念很像：

1 除去那些沒有支援 ES6 標準的瀏覽器，或者是支援不同模組規範（如：CommonJS）的引擎。

```

namespace Shapes {
  export type Circle = {
    type: 'Circle';
    radius: number;
  };

  // 略 ...
}

// 使用 Shapes.Circle 型別
let circle: Shapes.Circle = { /* ... 略 */ };

```

如果你忘記了關鍵字 `export` 就會出現如圖 8-4 的錯誤訊息。

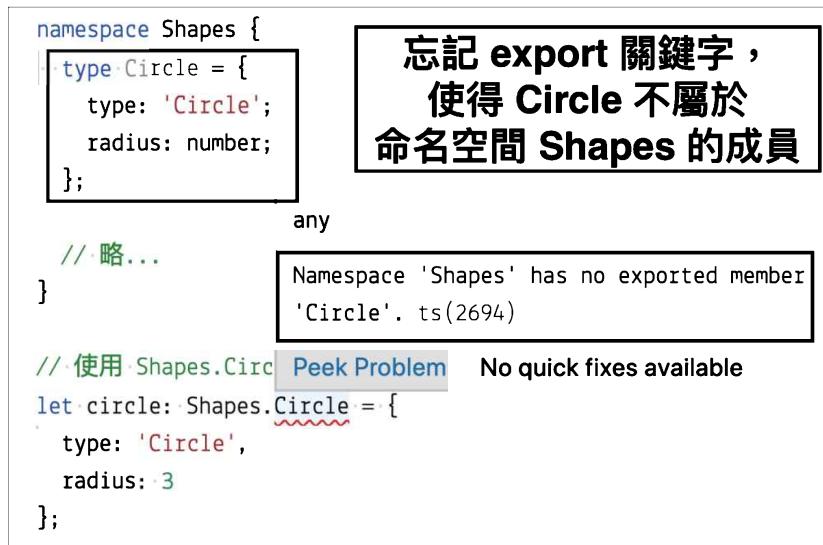


圖 8-4 沒有將型別從命名空間輸出時出現的錯誤訊息

理所當然地，介面、類別等宣告也都可以被輸出：

```

namespace SomeNamespace {
  export interface SomeInterface { /* 介面的宣告 */ }

  export class SomeClass { /* 類別的宣告 */ }
}

```

基本上，有了命名空間，任何模組都會互相獨立，因此：

```

namespace Shapes {
  export type Circle = {

```

```
type: 'Circle';
radius: number;
};

// 略 ...

}

namespace Dimensions {
export type Circle = {
    type: 'Circle';
    radius: number;
    sideCount: Infinity; // 圓形的邊數事實上有無限多個邊，變數變多，因此變成圓形
};

// 略 ...
}
```

以上的範例程式碼中的 `Shapes.Circle` 與 `Dimensions.Circle` 皆指涉為不同的型別，儘管重複了 `Circle` 這個命名。

8.2.3 命名空間的融合 Namespaces Merging

與介面的融合（Interface Merging）² 相似，命名空間若重複宣告的話，並非覆寫掉過往宣告的命名空間，而是進行宣告的融合（Declaration Merging）。

因此呢，以下的宣告方式：

```
/* 宣告命名空間 Shapes */
namespace Shapes {
    export type Circle = {
        type: 'Circle';
        radius: number;
    };
}
```

2 介面的融合，參見本書條目 6.2.2。

```
/* 重複宣告命名空間 Shapes 並不會覆寫掉前面的宣告 */
namespace Shapes {
    export type Rectangle = { /* 略 ... */ };

    export type Triangle = { /* 略 ... */ };
}
```

宣告的效果等效於：

```
/* 命名空間 Shapes 進行宣告性融合 */
namespace Shapes {
    export type Circle = {
        type: 'Circle';
        radius: number;
    };

    export type Rectangle = { /* 略 ... */ };

    export type Triangle = { /* 略 ... */ };
}
```

命名空間的宣告融合過程中，若出現型別化名（Type Alias）之名稱重複時，就會出現如圖 8-5 的錯誤訊息。

```
namespace Shapes {
    export type Circle = {
        type: 'Circle';
        radius: number;
    };
}

export type Rectangle = [
    type: 'Rectangle'
]
```

Duplicate identifier 'Circle'. ts(2300)

Peek Problem No quick fixes available

namespace Shapes {
 export type Circle = {
 type: 'Circle';
 radius: number;
 };
}

重複的型別化名名稱會出現
Duplication 錯誤

```
export type Rectangle = [
    type: 'Rectangle'
]
```

圖 8-5 不能重複宣告型別化名

類別（Class）³ 也不能夠使用同一個名稱重複宣告。（錯誤訊息如圖 8-6）

The screenshot shows a portion of a TypeScript file. It starts with a namespace declaration for 'Shapes'. Inside, there is an 'export class Example {}' statement. Below it, there is another 'export class Example {}' statement. A tooltip box highlights the second 'Example' with the text 'Duplicate identifier 'Example''. The code editor interface includes a 'Peek Problem' button and a note 'No quick fixes available'. A large callout box at the bottom right contains the text '類別也不能重複宣告'.

```
namespace Shapes {
    export class Example {}
    export class Example {} // Duplicate identifier 'Example'. ts(2300)
}
namespace Shapes {
    export class Example {}
```

圖 8-6 類別不能夠重複宣告

然而，若是宣告為介面，且重複使用介面時，由於介面也屬於定義的融合範疇，因此會按照條目 6.2.2 講的方式進行融合。

```
/* 命名空間裡的介面 Example 會按照條目 6.2.2 討論過的結果進行介面融合 */
namespace Shapes {
    export interface Example {
        propA: string;
    };
}

namespace Shapes {
    export interface Example {
        propB: number;
    };
}
```

所以以上的寫法等效於：

```
namespace Shapes {
    export interface Example {
```

3 實際上，在命名空間裡也可以輸出函式等物件，而這些物件也不能用同一個名稱重複宣告。

```
    propA: string;
    propB: number;
  };
}
```

8.2.4 巢狀命名空間 Nested Namespaces

命名空間裡也可以宣告命名空間，不過跟型別的宣告等，要輸出就得使用關鍵字 `export`，否則外面就沒辦法使用內層的命名空間裡的型別。

```
/* 巢狀命名空間的宣告 */
namespace Outer {
  export namespace Inner {
    export type Example = {
      propA: string;
      propB: number;
    };
  }
}

/* 使用巢狀命名空間內部輸出的型別 */
let obj: Outer.Inner.Example = {
  propA: 'Hello world',
  propB: 123
};
```

由於命名空間本身具備定義融合性質，因此以下的宣告手法：

```
namespace Outer {
  export namespace Inner {
    export type Example1 = /* Example1 的型別化名宣告 */;
  }
}

namespace Outer {
  export namespace Inner {
    export type Example2 = /* Example2 的型別化名宣告 */;
  }
}
```

等效於融合過後的寫法：

```
namespace Outer {
    export namespace Inner {
        export type Example1 = /* Example1 的型別化名宣告 */;
        export type Example2 = /* Example2 的型別化名宣告 */;
    }
}
```

8.2.5 引入命名空間 Import Namespaces

事實上，條目 8.1 講到的 ES6 Import / Export 也可以引入 TypeScript 的命名空間，只要宣告命名空間時，記得附上 `export` 關鍵字的宣告就好：

```
// 檔案名稱 ./Shapes.ts

// 記得要把命名空間輸出
export namespace Shapes {
    export type Circle = {
        type: 'Circle';
        radius: number;
    };

    export type Rectangle = { /* 略 ... */ };

    export type Triangle = { /* 略 ... */ };
}

// 檔案位置：./index.ts
import { Shapes } type './Shapes';

// 使用命名空間提供的 Circle 型別
let circle: Shapes.Circle = { /* ... 略 */ };
```

► 8.3 型別宣告 Type Declaration

8.3.1 連結 JavaScript 與 TypeScript 的橋樑

早期 TypeScript 還在成長狀態時，大部分的套件都是原生 JavaScript 寫出來的（這是廢話），因此想要和 TypeScript 協作幾乎是不可能的事情。

所以想要支援 TypeScript 專案時，暴力的方法就是將整個套件從原生 JavaScript 灌入 TypeScript 型別系統的宣告，並且適時的註記與處理邊緣情境（Edge Cases）；想當然，誰有那個時間將開源專案全部變成 TypeScript 版本的套件？

從模組化系統出來後，TypeScript 提供了一個管道，讓開發者不需將整個 JavaScript 寫成的套件轉成 TypeScript，仍然可以和 TypeScript 共同協作，這個中間溝通的橋樑被稱之為型別宣告（Type Declaration）。

讀者可能覺得本書已經在很前面的章節介紹過型別化名、介面的宣告等等東西，現在又多一個很模稜兩可的詞——「型別宣告」——是想整人嗎？

作者沒有存心整人，官方用的就是 Type Declaration 這個詞。

而語法層面上就是使用 `declare` 關鍵字：

```
/* 宣告一個字串型別的型別化名 message */
declare type message = string;
```

讀者可能又覺得宣告都不需要用 `declare` 關鍵字了，生出這種東西是想怎樣？

主要是為了解決可以在 TypeScript 程式碼引入原生 JavaScript 寫成的模組。

以下就由作者娓娓道來。

8.3.2 引入原生 JavaScript 寫成的模組

作者的目標是，能夠讓以下的程式碼被引入到 TypeScript 的專案裡：

```
// 檔案路徑：./hello-world.js ← 注意：是 JavaScript 檔案
const message = 'Hello world';

/* Node Module 輸出的方式 */
module.exports = { message };
```

想當然，如果你直接在 TypeScript 的專案裡引用 JavaScript 檔案，絕對無法編譯。

```
// 檔案路徑：./index.ts
import { message } from './hello-world';

// 印出 message 存的值
console.log(message);
```

如圖 8-7 為錯誤訊息。

The screenshot shows a code editor with two files open:

- hello-world.js**:
A plain JavaScript file containing:

```
const message = 'Hello world';
module.exports = { message };
```

A callout box highlights the line `module.exports = { message };` with the text "hello-world.js 中的 message 為 Any 型別".
- index.ts**:
A TypeScript file containing:

```
import { message } from './hello-world';
console.log(message);
```

A callout box highlights the import statement `import { message } from './hello-world';` with the text "Peek Problem No quick fixes available".

Both files show error messages in the status bar:

- hello-world.js**: Could not find a declaration file for module './hello-world'. '/Users/pro/Desktop/TypeScript Example/hello-world.js' implicitly has an 'any' type. ts(7016)
- index.ts**: No quick fixes available

圖 8-7 從 TypeScript 的檔案引入 JavaScript 程式碼，出現的錯誤訊息

從錯誤訊息可以得知一件事情：從 **JavaScript** 裡引入的東西是無法被 **TypeScript** 分析過程時推論，因此我們只有一個解決方式——主動告訴 **TypeScript** 從原生 **JavaScript** 套件引入的東西之型別。

8.3.3 型別定義檔 Type Definition File

想要告訴 **TypeScript** 我們引入的東西是什麼的話，最簡單的方式是宣告型別的定義（Declaration of Type Definition），這就得用到條目 8.3.1 講到的 **declare** 關鍵字。

通常宣告型別定義時，我們會將它們放到名為定義檔（Definition File）的地方，檔案的命名結尾為 **.d.ts**，就是多了 **.d** 這個東西。由於引入的 **JavaScript** 套件名稱為 **hello-world.ts**，相對應建造它的定義檔，命名為 **hello-world.d.ts**。

```
// 檔案路徑： ./hello-world.d.ts ← 宣告型別定義的檔案

// 宣告變數 message 的型別定義，其對應型別為字串
export declare const message: string;
```

這麼一來我們的程式碼可以正常運行囉！（圖 8-8 為變數 **message** 在 **TypeScript** 的推論結果）

```
hello-world.js
1 // 檔案路徑： ./hello-world.js ← 注意： ...
2 const message = 'Hello world';
3
4 /* Node Module 輸出的方式 */
5 module.exports = { message };
6
7
8

hello-world.d.ts
1 export declare const message: string;
2
3
4
5
6
7
8

index.ts
1 import { mes (alias) const message: string
2           import message
3           console.log(message);
4
5
6
7
8
```

message 變數
型別定義

經由型別定義檔案，可以推論
出 message 變數之型別

圖 8-8 利用型別定義檔告訴 **TypeScript**，**JavaScript** 套件裡的東西之型別

這就是我們可以在不將原生 JavaScript 轉換成 TypeScript 程式碼的情形下，就由單純的型別定義讓 TypeScript 程式碼使用 JavaScript 程式提供的功能。

不過讀者仔細比較有 `declare` 關鍵字跟沒有 `declare` 關鍵字的宣告情形下，差異在於，型別宣告部分（也就是有 `declare` 關鍵字的宣告）只有單純宣告型別的作用。

前面的 `hello-world` 套件的案例是告訴 TypeScript 變數 `message` 的型別資訊；而 `declare` 關鍵字理所當然可以用在各種型別的宣告：

```
/* 變數型別宣告 */
declare var foo: number;
declare let bar: string;
declare const baz: boolean;

/* 函式型別宣告（超載也可以喔！）*/
declare function addition(input1: number, input2: number): number;
declare function addition(input1: string, input2: string): number;

/* 類別宣告 */
declare class Person {
    /* 各種存取修飾模式都可以宣告 */
    public name: string;
    private age: number;
    protected interest: string[];

    /* 成員方法也可以宣告 */
    public getInfo(): string;
}

/* 型別化名宣告 */
declare type PersonalInfo = {
    name: string;
    age: number;
    interests: string[];
};

/* 介面宣告 */
declare interface UserAccount {
```

```
email: string;  
password: string;  
subscribed: boolean;  
}
```

尤其在函式與類別的 `declare` 型別宣告部分，你不需要去實踐內容；原因很簡單，因為型別宣告會被塞在定義檔，而定義檔的目的只是描述那些 JavaScript 程式碼提供的功能對應之型別介面。(實踐部分早就被原生 JavaScript 做掉了)

另外，如果該定義宣告需要輸出時，一定要加上 `export` 關鍵字喔。

»» 型別宣告 Type Declaration

1. 使用 `declare` 關鍵字可以進行型別的宣告。
2. 型別宣告的東西不需要進行實踐，它僅只具備提供型別資訊輔助 TypeScript 的型別檢測與編譯過程而已。
3. 想要從原生 JavaScript 寫出來的模組，引入到 TypeScript 專案時，必須提供型別定義檔 (Type Definition File)；型別定義檔固定會有 `.d.ts` 的後綴。

► 8.4 引入純 JavaScript 套件的流程

8.4.1 開源社群提供之型別定義檔

既然條目 8.3 已經闡述完接軌原生 JavaScript 到 TypeScript 的流程了，想當然，型別定義檔是個重要的東西。而熱心的開源社群將幾乎常見的原生 JavaScript 對應的型別定義檔也集中放置在名為 **DefinitelyTyped** 的 GitHub 資料庫 (Repository)⁴；一般如果會用 TypeScript 應該都要知道這個資料庫，不然沒辦法使用第三方套件輔助開發。

4 GitHub Definitely Typed：<https://github.com/DefinitelyTyped/DefinitelyTyped>。

少數已經都翻譯成 TypeScript 的開源套件，就不需要按照本條目所闡述的安裝手法，而是直接安裝就可以在 TypeScript 專案使用。(例如：RxJS 在第五版已經全面改用 TypeScript 開發，因此可以直接受到套件而不需要管一些其他細節)

8.4.2 安裝原生 JavaScript 開發的套件

本條目會以 JQuery 為例，展示如何安裝原生 JavaScript 的套件到 TypeScript 專案裡；而 JQuery 的型別定義檔已經由開源社群的人幫你處理好了。(你不會真的想將 JQuery 裡所有的型別定義手打出來吧！？)

讀者可以先到任何檔案資料夾自行試試看。一般建立 JavaScript 專案的流程會先把 `package.json` 進行初始化的動作；該檔案涵蓋專案的基本設定以及套件內容等，最簡單的指令就是：

```
$> npm init -y
```

假設作者進到的檔案資料夾路徑是 `<PATH_NAME>/test`，初始化的指令會產出以下的內容：

```
Wrote to <PATH_NAME>/test/package.json:
```

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

基本上，`package.json` 檔案的內容就是以上所示。

由於本條目要用 JQuery 作為示範，我們必須下載該套件：

```
$> npm install jquery
```

下載完後，你的 `package.json` 應該要在 `dependencies` 部分顯示出套件的資訊：

```
/* package.json 的設定 */
{
  /* 略 ... */,
  "dependencies": {
    "jquery": "JQUERY_VERSION" // ← 版本不一定，看最新版本為何
  },
  /* 略 ... */
}
```

接下來，新增 `index.ts` 檔案並填入簡單的測試程式碼：

```
/* 將 JQuery 套件引入 */
import $ from 'jquery';

/* 當瀏覽器的 document 載入時，console 就會印出 Hello world 字串 */
$(document).ready(function () {
  console.log('Hello world!');
});
```

想當然，因為 JQuery 套件本身是用純 JavaScript 寫出來的，並沒有 TypeScript 型別系統的指引，因此使用 JQuery 相關的方法，出現的推論結果基本上都會被視為 Any 型別。(如圖 8-9)



圖 8-9 使用 JQuery 套件，沒有型別定義檔時，推論結果都會是 Any 型別

這時候為了讓 TypeScript 專案能夠順利開發，我們就需要依賴 JQuery 的型別定義檔。在 TypeScript 這個領域，型別定義檔儘管都會集中在 **DefinitelyTyped** 這個資料庫裡；然而，下載這些型別定義檔時，固定會用以下的格式下載：

```
$> npm install @types/<PACKAGE_NAME>
```

也就是說，JQuery 套件的型別定義檔可以用以下的指令下載：

```
$> npm install @types/jquery
```

前面的 **@types** 為 NPM 套件的命名空間⁵，任何有 **@types** 名稱前綴的套件通常就是型別定義檔。

下載完之後你可以看到 **node_modules** 的資料夾會多出 **@types** 的資料夾，並且存放著 JQuery 的型別定義檔。(如圖 8-10)

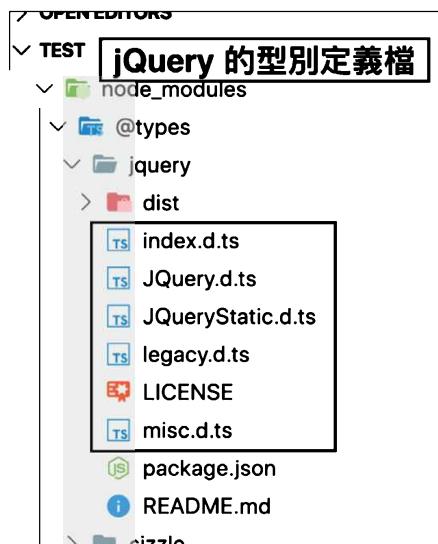


圖 8-10 **@types** 資料夾內的存放下載的套件對應之型別定義檔

當型別定義檔下載過後，VSCode 裡，JQuery 程式碼相關的型別提示就會出來囉！(如圖 8-11)

5 請注意，這裡講的不是 TypeScript 的命名空間，是 NPM 套件系統的命名空間，又被稱作為 Package Scope，請參見：<https://docs.npmjs.com/about-scopes>。

```
import $ from 'jquery';

$(document).ready(function () {
    console.log(method) JQuery<Document>.ready(handler: ($: JQueryStatic) => void): JQuery<Docu
});
```

藉由型別定義檔提供的資訊，TypeScript
可以將 JQuery 方法的型別提示內容展示出來

Specify a function to execute when the DOM is fully loaded.

@param handler — A function to execute after the DOM is ready.

圖 8-10 正常的型別提示內容出來了

»》型別定義檔的下載

1. 開源社群裡，大部分原生 JavaScript 的套件對應的 TypeScript 型別定義檔會放在名為 **Definitely Typed** 的 GitHub 資料庫裡。
2. 套件對應的型別定義檔，固定會有 @types 的前綴命名。
3. 大部分的套件（原生 JavaScript 開發或編譯出來的結果），若想要和 TypeScript 專案協作就必須同時下載該套件以及該套件對應的型別定義檔。
4. 少數本身套件若為 TypeScript 開發出來的，則不需要經歷第 3 點講到的步驟。

► 本章練習

1. 以下兩種引入東西的方式差異性在哪？

```
import A from './module_A';
import { A } from './module_B';
```

2. 請問可以在型別宣告的同時進行輸出嗎？

```
export type T = /* T 之型別內容 */;
```

3. 如何引用此模組內的所有東西？（試著展示各種不同的寫法）

```
type T = /* T 之型別內容 */;
type U = /* U 之型別內容 */;
```

```
export { T };
export default U;
```

4. 若提供一個原生 JavaScript 的套件，其內容為：

```
// example-module.js

// input1 與 input2 皆可能為數字或字串型別的值
export function addition(input1, input2) {
    const value1 = typeof input1 === 'number' ?
        input1 :
        parseInt(input1, 10);

    const value2 = typeof input2 === 'number' ?
        input2 :
        parseInt(input2, 10);

    return value1 + value2;
}
```

你會如何將其引入 TypeScript 專案裡？

5. Lodash 為常見的 JavaScript 套件之一，以下提供簡單的範例程式碼：

```
import _ from 'lodash';

_.forEach([1, 2, 3, 4, 5], function (value) {
    console.log(value);
});
```

以上的程式碼呼叫 Lodash 提供的 forEach 方法，用來迭代聚合物 (Collection) 相關的物件。

試著將以上的程式碼成功地在 TypeScript 環境裡執行，並且讓 VSCode 推論 forEach 方法時不會出現 Any 型別，而是型別推論出的結果。

提示：Lodash 為原生 JavaScript 寫出來的套件。

Part II

TypeScript 應用篇

09

物件導向進階篇章

本條目的設立就是要讓讀者深入第 5、6 章講到的類別（Class）與介面（Interface）的使用方法；然而，如果單純從這語法角度切入說明，視野也未免太狹隘，反而會建議讀者從雙方角度，也就是語法（Syntax）以及物件導向（Object-Oriented）程式設計本身的概念一起切入；畢竟物件導向的概念滋生出了類別與介面的語法，而沒有類別與介面的語法就無法實踐物件導向的概念，兩者是相向並存的。

不過本章強烈要求讀者至少先把語法部分（也就是第 5、6 章的內容）大略了解之後再深入探究；然而本章也會將物件導向設計比較精華的部分儘量以簡單的方式呈現給讀者看，但不代表本章就可以涵蓋所有進階主題（這樣也太誇張），因此會儘量打好讀者的基礎。

► 9.1 物件導向進階概論

9.1.1 「介面為主・實踐為輔」的設計思維

本書在第 5 章介紹類別（Class）語法的時候，特別指出物件（Object）這個東西的實際意義：

內含變數、函式以及多種不同資料結構封裝（Encapsulate）成的一種可被操作的介面——就是物件。

然而，這時作者想問一個問題：

為何要以物件導向中的物件（Object）的設計為主？

憑什麼要用類別建構物件而不直接用 JavaScript JSON 物件就夠了？

JSON 可以封裝變數（也就是 JSON 屬性）以及方法，也比整個類別的語法簡單許多，何苦繞遠路還要學一個看起來很冗贅的 Class 語法？

我們思考一個情境，假設我們想要將一坨資料或者是聚合物（Collection）進行檢視內容的動作，該資料可能是陣列，也有可能是集合（Set）¹，那你可能想說可以宣告簡單的 `inspect<T>` 函式、傳入陣列，並且用個簡單的 For 迴圈迭代印出結果：

```
function inspect<T>(collection: Array<T>) {
  for (let i = 0; i < collection.length; i += 1) {
    console.log(collection[i]);
  }
}

inspect<number>([1, 2, 3, 4, 5]);
// => 1
// => 2
// => 3
// => 4
// => 5
```

甚至還不用 JSON 物件，就可以直接達成目標。

這個實現功能的過程就是時常聽到的「實踐」（Implementation）兩字的概念。
(作者在講廢話)

1 ES6 集合 Set，參見本書條目 10.3。

然而，這個 `inspect<T>` 函式非常之脆弱，假設今天我們有另一個資料結構是鏈結串列（Linked List）²，其簡單的結構如下：

```
/* 鏈結串列會不停指涉下一個串列節點（List Node），若無下一個值，就會指向 null */
type ListNode<T> = {
    value: T;
    next: ListNode<T> | null;
};

/**
 * 以下的鏈結串列可以圖解成：(1) → (2) → (3) → (null)
 * 每一個值代表鏈結串列中的一個結點，若串列下一個地方沒元素時，就指向 null
 */
let list: ListNode<number> = {
    value: 1,
    next: {
        value: 2,
        next: {
            value: 3,
            next: null
        }
    }
};
```

這時你要如何讓 `inspect<T>` 同時支援陣列或以上的鏈結串列的迭代檢視過程？

你可能想說我們可以用個簡單的判斷式，根據傳入的不同的資料型態進行處理：

```
function inspect<T>(collection: Array<T> | LinkedList<T>) {
    /* 使用型別駐防的方式檢測陣列型別 */
    if (Array.isArray(collection)) {
        for (let i = 0; i < collection.length; i += 1) {
            console.log(collection[i]);
        }
    } else {
        for (let node = collection; node !== null; node = node.next) {
            console.log(node.value);
        }
    }
}
```

2 鏈結串列為常見的基礎資料結構之一，參見 Wikipedia：https://en.wikipedia.org/wiki/Linked_list。

```
    }
} else {
let node = collection;

// 先印出第一個節點的值
console.log(node.value);

// 檢測有沒有下一個節點，若無則跳出
while (node.next !== null) {
    node = node.next;
    console.log(node.value);
}
}

inspect<number>(list);
// => 1
// => 2
// => 3
```

看看以上程式碼的實踐方式，這實在是很糟糕，為了延伸出一個功能，讓函式的宣告多出了約莫十行的程式碼，這樣每一次要支援不同資料型態的輸入只會是一場惡夢。

另外，`inspect<T>` 這個函式名稱實在是很糟糕，是在檢視（Inspect）什麼樣的東西或是什麼樣的資料結構、物件？但如果又改成 `inspectArray` 又會更怪，所以變成我們還會有 `inspectSet` 或甚至 `inspectLinkedList` 等函式以及對應實踐過程出來嗎？

但是產生更多方法又會違背我們原先將這些功能整合成單一 `inspect` 方法的需求，所以到底該怎麼處理這方面的問題呢？

如果要是有一個統一的操作介面（Interface）`Collection`，該介面規定任何物件都必須有 `inspect` 方法時，此時就可以改寫成以下的形式：

```
/* 以下展示 Pseudo Code 範例，意思為偽程式碼 */
let collection1: Collection = new TypedArray<number>([1, 2, 3, 4, 5]);
collection1.inspect();
```

```
// => 1
// => 2
// => 3
// => 4
// => 5

let collection2: Collection = new LinkedList<number>(list);
collection2.inspect();
// => 1
// => 2
// => 3
```

看起來是不是乾淨許多呢？

若我們規範類別的宣告必須符合 `Collection` 的設計，我們就可以不用管資料結構的細節，只管可以如何規範介面的行為以及使用介面提供的功能就好了。

因此，物件導向的核心概念不是膚淺地叫你單純設計物件而已，而是叫你：

「專注於系統的介面設計，而非專注於實踐過程」³

「物件」只是一種名稱或者是媒介，被建構出來並且被操作的東西罷了；真正的重點是——「類別」的宣告就是描述物件可以做出的行為。

此外，不提倡直接實踐的原因非常簡單，因為就會發生剛剛宣告 `inspect<T>` 函式發生的慘劇，整個程式碼變成一坨義大利麵⁴；而且要實踐的功能可能性實在是過多，每次改個規格，很可能面臨過往實踐的程式碼全部被重改的困境。

因此物件導向設計提倡專注於可以被操作的介面行為的設計，因此「實踐過程」被取代為「操作、組合類別規範出的行為」；這樣一來，功能可以被重複

3 原文為“Program to an interface, not an implementation”，出自《Design Patterns: Elements of Reusable Object-Oriented Software》這本書。

4 Spaghetti Code，意旨功能摻雜在一起過度複雜化的程式碼，延伸新功能以及維護起來皆不易。

利用外，改動率可以儘量變小，要是需要延伸出功能，就是在類別的宣告實作上定義更多物件可以被操作的行為而已⁵。

9.1.2 抽象化的概念 Abstraction

「這東西不是之前就講過了嗎？在講抽象類別⁶的時候就已經提到了！」

本條目當然要再深入，不可能草草一句話就帶過：「抽象的概念是將一系列步驟包裝並且簡化的過程。」

事實上，整個物件導向設計的概念都在對整個實踐過程進行抽象化的動作，只是將其抽象化成前一個條目講述到——專注於介面設計——的概念。

也就是說，「實踐過程」抽象化的結果就是：「被設計出來的介面」。

如果說，條目 9.1.1 的 `inspect<T>` 函式的宣告是一段實踐過程，那麼簡化成的 `Collection` 類別並且建構、操作物件就是被抽象化過後的結果。

例如：我們可以說車子的儀表板，就是將整個車子的運作狀態抽象化過後的結果；也就是說，開車的速度、油量、里程數等，這些數據當然會被車子運作過程用機械的運作方式呈現，但人哪會看著輪子就能夠猜到轉速的數值，因此這些東西必須靠儀表板上面的數據呈現出來，而儀表板上的數據就是抽象化的結果。

又或者是：消毒水含有一定的酒精含量，但人的嗅覺不可能靈敏到可以聞出酒精濃度，因此才會發明酒精濃度計，專門來測量酒精濃度；酒精濃度上呈現的數值就是消毒水本身含有的酒精濃度被抽象化成數據的結果。

回歸 `Collection` 介面的設計，你會發現範例程式碼：

```
let collection1: Collection = new TypedArray<number>([1, 2, 3, 4, 5]);
```

5 Open-Closed Principle，請參見條目 9.2.2。

6 抽象類別（Abstract Class），參見本書條目 5.2.7。

```
collection1.inspect();

let collection2: Collection = new LinkedList<number>(list);
collection2.inspect();
```

建構符合 `Collection` 介面的實體（`Instance`⁷）時，使用 `inspect` 方法不會特別關注到底是陣列還是鏈結串列，只管說凡事符合 `Collection` 介面的東西，可以使用名為 `inspect` 的方法。

因此代表 `Collection` 介面的資料型態原本是啥根本不重要！我們只要記得把不同的資料型態轉換成（或者是抽象化成）統一的介面來操作就夠囉！

9.1.3 內聚與耦合 Cohesion & Coupling

另一個讀者看各種跟物件導向程式設計相關，一定會看到「內聚」與「耦合」這兩個詞彙，兩者互為相對關係（以數學名詞來形容就是成反比關係）。

「內聚性高」的程式碼代表「耦合程度低」，依此來推。

內聚描述的是將程式碼裡各種功能、函式等東西打包成的一個模組的狀態，而這個模組可以為任意形式的東西，例如：類別、物件甚至是簡單的函式也可以。

事實上，作者在條目 9.1.1 示範的 `inspect<T>` 的實作方式就是一種內聚的表現：

```
function inspect<T>(collection: Array<T> | LinkedList<T>) {
  if (Array.isArray(collection)) {
    // 實踐過程略 ...
  } else {
    // 實踐過程略 ...
  }
}
```

它負責將檢視陣列或鏈結串列內容的過程實踐進同一個函式裡。

⁷ 類別建構出的物件又被稱之為實體（`Instance`），參見條目 5.2。

內聚的形式方法有很多種，以上的範例程式碼屬於邏輯性內聚（Logical Cohesion），因為用的是判斷敘述式的手法針對不同的資料型態有不同的行為，而這也是最常見、但最容易產出不好的程式碼的寫法；畢竟判斷式儘管可以進行分流，但要是出現多種不同的情境時，程式碼會暴增到幾百幾千行只是遲早的事情，這樣反而造成程式碼維護變得很困難。

那麼有沒有好的內聚的寫法案例？有的，最常見的是順序性內聚（Sequential Cohesion）的寫法。

順序性內聚要理解非常簡單，譬如說描述一道料理製作過程的食譜即順序性內聚的典型案例，它會告訴你步驟與過程，但是每個步驟或過程都會是互相獨立的。

```
// 註：實際的烤蛋糕食譜請還是參照其他資源為主
function cake(flavor: 'Chocolate' | 'Vanilla' | 'Strawberry') {
    let milk = new Milk(2);           // ← 兩公升的牛奶
    let butter = milk.whip();        // ← 製作奶油（其實應該是要製作麵團才對）
    butter.mix(flavor);            // ← 摻入風味

    let bakingPan = new Pan(butter); // ← 將奶油倒入烤盤

    let oven = new Oven(bakingPan);      // ← 將烤盤放入烤箱
    oven.cook({ temperature: 100, duration: 30 * 60 }); // ← 烤箱溫度與時長

    // 從烤箱取出蛋糕
    return oven.result;
}
```

你會發現以上的 `cake` 函式也是描述烤蛋糕的過程，而描述步驟的過程屬於順序性內聚的寫法，而此寫法被歸類為好的內聚形式；事實上內聚的表現形式當然不只這兩種，其他形式的內聚可以參照⁸，讓讀者延伸鑽研。

8 內聚的種類非常多種，請參見 Wikipedia：[https://zh.wikipedia.org/wiki/內聚性_\(計算機科學\)](https://zh.wikipedia.org/wiki/內聚性_(計算機科學))。

另外，相對於內聚的概念就是「耦合」——衡量模組與模組之間的相依程度（Dependency），譬如兩個模組共同使用一個全域變數（Global Variable）、函式、物件、類別等東西，這些東西就是該模組相依的東西。

也就是說，以剛剛的 `cake` 函式來說，由於它有使用到很多類別，諸如 `Milk`、`Pan` 以及 `Oven` 類別，我們就可以說 `cake` 函式直接相依（或直接耦合）於這三個類別；若是其他的食譜，比如說製作冰淇淋 `icecream` 的函式剛好也需要 `Milk` 類別，我們可以說 `icecream` 也相依 `Milk` 類別。

程式碼的相依程度越高會造成開發上會越來越複雜，譬如說如果今天你想將食譜原物料中的牛奶 `Milk` 改成羊奶 `GoatMilk`，這樣就會變成你要改原本的程式碼，不然就是複製原本的程式碼、貼到其他地方、改掉函式名稱並且將裡面的牛奶 `Milk` 改成羊奶 `GoatMilk` 類別。

因此不建議直接耦合的行為，但是模組與模組間的溝通勢必還是要進行耦合或者相依的行為，這時會建議改採抽象耦合的模式；用 TypeScript 來說，與其直接相依物件的類別（Class），不如相依於介面（Interface）⁹，你可能會將 `cake` 函式多出一個參數，填入奶類製品 `DairyProduct` 介面，並且讓牛奶 `Milk` 與羊奶 `GoatMilk` 都實踐 `DairyProduct` 介面，這樣就可以改採抽象相依的模式，避免直接相依造成程式碼沒辦法輕易修改的窘境。

```
interface DairyProduct {  
    // DairyProduct 奶製品介面的規格  
}  
  
class Milk implements DairyProduct {  
    // Milk 實踐 DairyProduct 的規格  
}  
  
class GoatMilk implements DairyProduct {  
    // GoatMilk 也實踐 DairyProduct 的規格  
}
```

⁹ 介面（Interface），請參見本書第 6 章。

```
// 烤蛋糕可以選擇口味外，還可以選擇採用的奶製品原料
function cake(flavor: /* flavor 的型別 */, dairyType: DairyProduct) {
    // 略 ...
}

// 冰淇淋可以選擇口味外，還可以選擇採用的奶製品原料
function icecream(flavor: /* flavor 的型別 */, dairyType: DairyProduct) {
    // 略 ...
}

// 製作巧克力風味的蛋糕，原物料為牛奶
cake('Chocolate', new Milk(/* Milk 類別的建構子函式參數 */));

// 製作香草風味的冰淇淋，原物料為羊奶
icecream('Vanilla', new GoatMilk(/* GoatMilk 類別的建構子函式參數 */));
```

講完內聚與耦合大致上的觀念：好的內聚通常是將程式中不相關的東西整合在一起（譬如：做完一件事情的步驟，步驟與步驟兼毫無相干，此為順序性內聚）；但是不可能所有的軟體程式設計過程都是不相關的，因此這時我們可能對這些相似性質的程式採取抽象耦合的策略，將程式相關的東西用抽象介面的方式進行耦合的動作（譬如：一段程式中某一個步驟，發現該步驟有多種方式可以替換）。

► 9.2 物件導向設計原則 SOLID Principles

基本上，本條目彙整的是從物件導向的概念延伸出最基礎的五大原則¹⁰；以下條列的五大原則英文開頭分別就是 S、O、L、I、D 這五個字母，故又另稱為 SOLID 原則。

10 由 Robert C. Martin 提出的五大設計原則，出自《Agile Software Development, Principles, Patterns, and Practices》。

9.2.1 單一職責原則 Single-Responsibility Principle

原文的定義是：

“A class should have only one reason to change.”

然而，作者認為前面看似只有講說類別（Class）必須遵守外，實質上它所涵蓋的意涵應該包含所有的東西，包含函式、類別、介面的設計等。

「單一職責原則」——光是看名稱大概就可以猜得出它的意思，主要建議任何類別（或函式、方法、資料等）只要能夠簡單地完成一項事情就可以了。譬如說，你不會在一個蛋糕的食譜裡，寫出冰淇淋的製作過程；然而，要是遇到要製作冰淇淋蛋糕時，與其將冰淇淋的製作過程一併寫進冰淇淋蛋糕的食譜，還不如直接引用冰淇淋的食譜，專心講蛋糕部分的製作過程就好了。（亦或者是同時引用冰淇淋與蛋糕的食譜，加入步驟不同的地方與細節等）

然而，如果是另一種情形：類別（或函式、物件、介面等）本身要負擔的責任太大時，你可能會考慮將該事情切分成更細的細節處理；例如一個在描述動物的類別 `Animal`，它可能是鳥類所以會飛，可能是魚類會游泳等：

```
class Animal {  
    public canFly: boolean;  
    public canSwim: boolean;  
  
    public fly(): void { /* 飛！ */ }  
    public swim(): void { /* 游泳！ */ }  
  
    // 其他成員變數或成員方法的宣告  
}
```

不過這樣子就必須先判斷到底是能不能飛 `canFly` 或能不能游泳 `canSwim`，並且再使用 `fly` 或 `swim` 方法；儘管說動物看似除了可能會飛或者會游泳是合理的行為，但是這個類別負擔的責任實在太大，因此可能會將 `Animal` 設計成介面（Interface），並且分別實踐出 `Bird` 或者是 `Fish` 類別。

```
interface Animal {  
    /* 動物介面的基本規格略 ... */  
    fly(): void;  
    swim(): void;  
}  
  
/* 各種類型的動物去實踐 Animal 介面 */  
class Bird implements Animal { /* 鳥類的類別實踐 */ }  
class Fish implements Animal { /* 魚類的類別實踐 */ }
```

當然，以上只是一種想法，實踐過程還是得根據規格或需求來訂，如果責任負擔不大，再拆更細碎反而會有過度設計的問題。

再更深入一點，單一職責原則更精確來說是在建議一段程式儘量增強內聚力，不相關的東西應該要拆到其他的介面或類別實踐，並且用抽象耦合的方式處理；也就是說，也許動物類別 `Animal` 介面的設計，會飛或者是會游泳都是在描述動物可以做的行為，乾脆將其再拆成 `Flyable` 介面（代表可以飛）或者是 `Swimmable` 介面（代表可以游泳），並且將鳥類同時實踐 `Animal` 與 `Flyable` 介面，依此類推。

```
interface Animal {  
    /* 動物介面的基本規格略 ... */  
}  
  
interface Flyable {  
    /* Flyable 其他基礎規格略 ... */  
    fly(): void;  
}  
  
interface Swimmable {  
    /* Swimmable 其他基礎規格率 ... */  
    swim(): void;  
}  
  
/* 各種類型的動物去實踐 Animal 介面，有需要再去額外實踐其他功能 */  
class Bird implements Animal, Flyable { /* 鳥類的類別實踐 */ }  
class Fish implements Animal, Swimmable { /* 魚類的類別實踐 */ }
```

9.2.2 開放封閉原則 Open-Closed Principle

初次深入物件導向的設計原則的讀者，看到這標題肯定覺得很模糊甚至感到有些矛盾，不過那是過度簡化解讀的結果；實際上開放封閉原則（有時候簡寫為開閉原則）的定義是：

“Software entities (class, modules, functions, etc.) should be **open for extension**, but **closed for modification**.”

簡而言之就是，一段程式碼的模組（可能是類別、方法等）對於擴充（Extension）的行為是開放的；對於修改（Modification）的行為是封閉的。

本原則就是在避免「改程式碼」的行為，相信改程式碼的過程很痛苦，更何況是改別人寫的程式碼；不過讀者可能覺得真的有辦法做到用新增、擴充程式碼的方式取代掉修改程式碼嗎？

其實在條目 9.1.3 討論的 `cake` 函式的實作過程中，如果內部將奶製品統一使用牛奶類別 `Milk`，這樣就是直接耦合造成寫死的窘境。因此更換成羊奶 `GoatMilk` 就變成可能你會面臨到函式必須新增參數，並且用判斷式來判斷奶製品的種類：

```
function cake(flavor: /* 口味的型別 */, isMilk: boolean) {
    let milk;
    if (isMilk) {
        milk = new Milk(2);      // ← 兩公升的牛奶
    } else {
        milk = new GoatMilk(2); // ← 兩公升的羊奶
    }

    // 實踐過程略 ...
}
```

讀者應該可以很清楚地看到，這種邏輯型內聚（Logical Cohesion）¹¹ 寫法是不

11 內聚與耦合，請參見本書條目 9.1.3。

好的；另外，如果奶製品種類還有其他種，該判斷式只會越長越大，因此我們改採宣告奶製品介面 `DairyProduct` 的方式，所有的奶製品（包含 `Milk` 與 `GoatMilk`）必須實踐此介面，並且用抽象耦合的方式實踐 `cake` 函式¹²。

你會發現，一但我們新增了新的奶製品種類，與其修改 `cake` 函式的實踐過程，我們已經可以改成新增程式碼——也就是實踐 `DairyProduct` 介面，就可以多出新的奶製品選項，而 `cake` 函式不需要更動。

因此本條目強調的東西除了是不鼓勵修改程式碼的實踐外，重點在於條目 9.1.1 強調的設計思維：

「介面為主 · 實踐為輔」

結合抽象化的技巧，避免程式碼的實作被寫死，才相對可以有擴充的空間；因此，開閉原則講述的「擴充程式碼」以取代「修改程式碼」的這個過程是可以達成的行為。

9.2.3 里氏替換原則 Liskov Substitution Principle

作者認為里氏替換原則算是描述類別繼承¹³ 或者是介面實踐必須遵守的原則。原文的定義是：

“Subtypes must be substitutable for their base types.”

所謂的 `Subtype` 指的就是像子類別的角色，而 `Base Type` 指的就是父類別（Parent Class）或者是 TypeScript 的介面（Interface）這些角色。

本原則的概念非常簡單：凡是父類別的物件，子類別可以替換掉它，但仍然可以操作父類別規範的行為，而不影響整體程式的執行。其實這就是在描述作者

12 請參見條目 9.1.3 後面的範例程式碼。

13 類別的繼承（Class Inheritance），請參見本書條目 5.2.6。

在條目 9.1.1 的 `inspect<T>` 函式範例的行為，假設 `Collection` 為 TypeScript 介面：

```
/* 假設 TypedArray 類別實踐 Collection 介面 */
let collection1: Collection = new TypedArray<number>([1, 2, 3, 4, 5]);
collection1.inspect();

/* 假設 LinkedList 類別也實踐 Collection 介面 */
let collection2: Collection = new LinkedList<number>(list);
collection2.inspect();
```

類別 `TypedArray<T>` 以及 `LinkedList<T>` 皆有實踐 `Collection` 介面的話，我們就可以把這兩個類別當成（或者替換成）`Collection` 介面規格的物件進行操作。

相信以上的例子有些模糊，因此再舉以下的案例：

```
interface Geometry {
    area(): number;           // ← 計算幾何圖形的面積
    circumference(): number;  // ← 計算幾何圖形的周長
}

class Circle implements Geometry {
    // radius 為圓形的半徑
    constructor(public radius: number) {}

    // 實踐 Geometry 的規格
    area() {
        return Math.PI * (this.radius ** 2);
    }

    circumference() {
        return 2 * Math.PI * this.radius;
    }
}

class Rectangle implements Geometry {
    // width, height 為長方形的長與寬
    constructor(public width: number, public height: number) {}
```

```
// 實踐 Geometry 的規格
area() {
    return this.width * this.height;
}

circumference() {
    return (this.width + this.height) * 2;
}
}
```

該範例宣告簡單的 `Geometry` 介面，為里氏替換原則的 Base Type；相對來說，類別 `Circle` 與 `Rectangle` 由於實踐 `Geometry` 介面，因此它們的角色為 Subtypes。

里氏原則告訴我們必須遵守 Base Type 的物件可以被替換為 Subtypes，意思是說：

```
let someShape1: Geometry = new Circle(3);
let someShape2: Geometry = new Rectangle(4, 5);
```

以上的 `someShape` 系列的變數儘管指向的是 `Geometry` 介面描述的物件型別，但是它可以指向或者被取代為 Subtypes 建構的物件，即類別 `Circle` 與 `Rectangle` 建構出的實體（Instance）。

然後，里氏替換原則強調：Base Type 被替換成 Subtypes 時，可以使用 Base Type 所規範的操作行為，也就是說我們不管 `someShape` 等變數到底是指向哪些東西，一定都可以使用 `Geometry` 介面規範的東西：

```
// 假設不知道 someShape1 是什麼樣的值，但你一定知道它可以呼叫 Geometry 介面所規範的行為
someShape1.area();
someShape1.circumference();

// 同理，someShape2 也一定可以呼叫 Geometry 介面所規範的行為
someShape2.area();
someShape2.circumference();
```

反之，里氏替換行為被違反的話，就表示 Subtypes 可能沒有正確地實踐出 Base Type 所規範的行為。

畢竟，從本章節開始就一直強調抽象耦合的重要性，而我們操作的東西都是抽象連結起來的物件，所以我們看不到連結到的實體長什麼樣子，只能依靠抽象介面（或父類別的規格）而定；理應來說 Subtypes（亦或者是被繼承或者被實踐的類別）就必須要符合抽象規格，這樣操作起來才不會有問題。

9.2.4 介面隔離原則 Interface Segregation Principle

本原則的原文定義是：

“No client should be forced to depend on methods it does not use.”

ISP 主要的目的是避免過度地將不必要的規格塞在單一個介面，就好比條目 9.2.1 講過的動物的介面設計 **Animal**：

```
interface Animal {  
    /* 動物介面的基礎規格略 ... */  
    fly(): void;  
    swim(): void;  
}  
  
/* 各種類型的動物去實踐 Animal 介面 */  
class Bird implements Animal { /* 鳥類的類別實踐 */ }  
class Fish implements Animal { /* 魚類的類別實踐 */ }
```

並不是所有的動物要會飛或者是要會游泳，因此才再被拆成 **Flyable** 或 **Swimmable** 介面；而類別實踐動物 **Animal** 介面的時候，視規格需求再去實踐其他被分離出的介面。

本原則強調抽象過程的精確性——基本上就是設計介面的過程中判斷規格有沒有存在該特定介面的必要性，若無必要時就隔離出新的介面。

而這些被隔離出的特定介面被稱作角色介面（Role Interface）。然而，譯作配角應該會更適合，畢竟如果以動物 **Animal** 介面的範例為主的話——主介面是 **Animal**，而它的配角介面就是 **Flyable** 或 **Swimmable** 介面；畢竟如果類別單純實踐配角介面（例如 **Flyable**），但沒有實踐主介面（例如：**Animal** 介面）本身就是不合理的行為。

簡而言之，你能夠想像本身不是動物但是有長翅膀的類別嗎？

9.2.5 依賴倒轉原則 Dependency Inversion Principle

本條目的原則如果看不懂，它還是會回歸到本章節開篇第一句話：

「**介面為主 · 實踐為輔**」。

正式的依賴倒轉原則的原文定義分成兩段：

“High-level modules should not depend on low-level modules. Both should **depend on abstractions.**”

“Abstractions should not depend on details. Details should **depend on abstractions.**”

你可以看到它的共通點是——模組的實踐過程中必須仰賴抽象介面；而實踐細節也是仰賴抽象介面——「**抽象化**」的概念簡直超重要的啊！（昏倒）

其實本原則在講的東西就是如何實現「**介面為主 · 實踐為輔**」這句話 (Program to an Interface, not an implementation) 的理念，白話文就是千萬不要寫死程式碼。

它跟條目 9.2.2 講到的開放封閉原則講的東西儘管很像，都是避免寫死程式碼，然而開閉原則強調避免寫死的原因是為了方便擴充程式碼；依賴倒轉原則強調如何實現「不要寫死程式碼」的過程與解法。(前者是強調寫死程式碼的「後果」，也就是程式碼無法擴充；後者強調如何避免寫死程式碼的「方法」，這樣才可以實現開閉原則中的擴充程式碼的效果)

首先第一段——“High-level modules should not depend on low-level modules. Both should **depend on abstractions.**” 其實講的東西還是可以用條目 9.1.3 舉的 `cake` 函式作為簡單的範例，其中 `cake` 最初的寫法就是直接依賴於牛奶 `Milk` 類別：

```
function cake(flavor: /* 口味的型別 */) {  
    let milk = new Milk(2); // ← 食譜中，奶製品被寫死 Milk 類別
```

```
// 略 ...  
}
```

因此如果畫成圖就會如圖 9-1 所示，`cake` 函式直接耦合 `Milk` 類別。

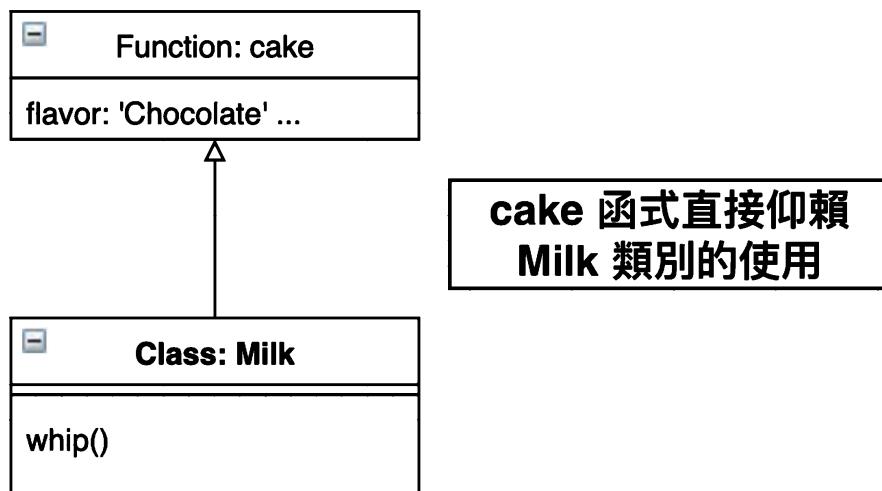


圖 9-1 `cake` 函式直接仰賴 `Milk` 類別的使用

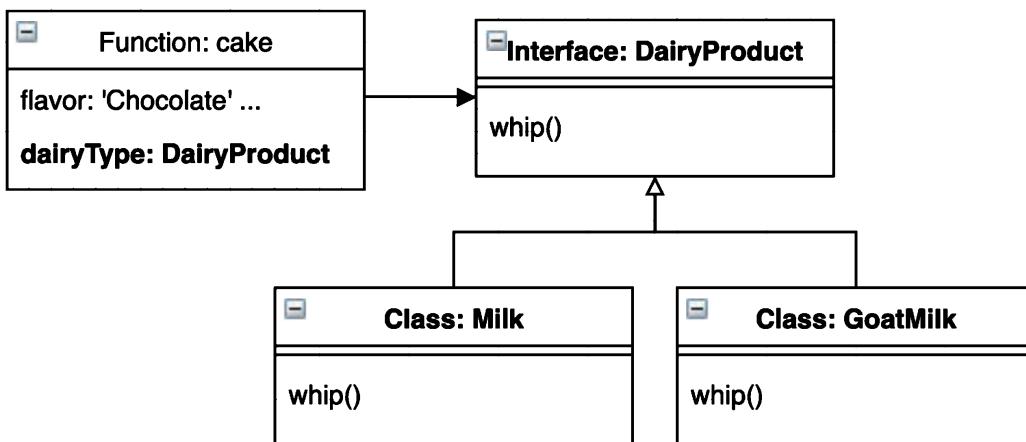
圖 9-1 就可以看得出，所謂的 High-Level Module 指的角色是 `cake` 函式，而 Low-Level Module 則是 `Milk` 類別；然而，依賴倒轉原則不鼓勵上下模組（也就是類別、函式等東西）直接相依（或耦合）的情形。

要是想避免這種直接耦合的現象，我們於是就宣告 `DairyProduct` 奶製品介面，並且讓 `Milk` 與 `GoatMilk` 類別，`cake` 函式就可以藉由奶製品介面進行抽象耦合：

```
interface DairyProduct {  
    // DiaryProduct 奶製品介面的規格  
}  
  
class Milk implements DairyProduct {  
    // Milk 實踐 DairyProduct 的規格  
}  
  
class GoatMilk implements DairyProduct {  
    // GoatMilk 也實踐 DairyProduct 的規格  
}
```

```
function cake(flavor: /* flavor 的型別 */, dairyType: DairyProduct) {  
    // 略 ...  
}
```

其中，這樣的關係可以在圖 9-2 示意。



cake 函式藉由抽象耦合的方式，從介面 DairyProduct 連結到跟奶製品相關的類別

圖 9-2 抽象耦合取代直接耦合的過程

你會發現圖 9-1 到 9-2 的過程會變成不管是 High-Level 或 Low-Level 模組，想要建立關聯（也就是進行耦合時）必須要同時改成依賴於抽象的介面。

這樣一來，想要使用奶製品相關的模組（此為 High-Level 的模組）可以直接依賴 **DairyProduct** 介面；相對地，想要擴充奶製品的種類（也就是 Low-Level 的模組）只要按照 **DairyProduct** 介面的規格實踐就好了。

至於 “Abstractions should not depend on details. Details should depend on abstractions.” 這句話其實意思非常簡單，而且也很基本，就是規格的制定不可能會根據實踐的結果而定；而實踐的結果必須根據規格而定。

想想看，設計一件產品的過程是「先把產品做出來再把規格補上去」還是「將產品的規格規劃好後再做出來」？理應來說應該是後者，產品做出來前一定有所規劃才會有一套標準流程做出來。

9.2.6 統整 SOLID 原則

相信讀者把條目 9.2 中的各個原則讀過一遍之後，離不開幾個關鍵重點：

1. 程式碼模組（Module）必須具備高聚合性（High Cohesion）。
2. 模組必須有相依時建議進行抽象耦合（Abstract Coupling）。
3. 忌諱修改程式碼。

而所謂的模組，範圍可以小至一個函式，大則可以大到代表一個類別的實踐。

而 SOLID 原則各自也有相似的地方，作者簡單將將這幾個原則的意義並統整如下：

SRP：即單一職責原則，代表模組儘量傾向於高聚合性的設計。（不要一次承攬太多無關緊要的責任，這樣反而會降低聚合性）

OCP：即開放封閉原則，代表模組藉由抽象耦合的方式進行程式碼的擴充，以防止程式碼內部的修改。

LSP：即里氏替換原則，代表某介面（或某父類別）的東西，可以被實踐該介面（或者繼承父類別的子類別）的物件給替換；畢竟達成抽象耦合的條件就是實踐介面（或類別的繼承）時，不能有任何例外狀況出現。

ISP：即介面隔離原則，強調介面職責分配的精確性；跟 SRP 差異在於，ISP 強調非必要的職責可以再獨立隔離成其他角色介面（Role Interface）。（然而，角色介面不能獨立實踐，必須配合主要介面一併使用）

DIP：即依賴倒轉原則，解釋了抽象耦合的方法，也就是任何模組互相依賴時，必須依靠中間的抽象介面進行耦合；而任何實踐出來的東西必須遵照抽象的規格。

SRP 與 ISP 差異在於前者提倡模組應具備高聚合性；後者則是聚合過程中，不必要的聚合，再特地隔離出新的介面。

LSP、DIP 與 OCP 都有講到抽象耦合的機制，然而 LSP 講的是達成抽象耦合的基本要件；DIP 則是抽象耦合的實踐方式；最後，OCP 則是提倡開發過程中鼓勵用抽象耦合的方式——擴充程式碼、防止修改程式碼。

這樣應該很清楚 SOLID 原則五者的描述差異性了吧～！

► 9.3 物件導向延伸應用

9.3.1 耦合與解耦 Coupling & Decoupling

讀者看到這標題可能覺得關於耦合（Coupling）的議題不就在前面的條目一直提過了嗎？

事實上，作者個人認為將程式模組（即類別、函式、介面與物件等）的耦合進行解構（簡稱解耦）是物件導向程式設計的核心概念之一；不過討論到“解耦”這個動詞還是得小心些，有些解耦的實際情形是真的把兩個耦合的程式模組完全拆掉、有些是根據 SOLID 原則變成抽象耦合關係、有些是直接變成單一內聚關係的模組，不過共通點是：

「解耦」泛指解構程式模組中的直接耦合的現象。

所以如果你認為轉換成抽象耦合的模組關係就不是解耦那就是錯誤的，解耦是針對直接耦合的關係進行解構。

那麼「直接耦合」的案例有哪些？

最簡單也最直觀的案例就是類別的繼承（Class Inheritance）。

```
class Parent { /* 父類別的實踐過程 */ }
class Child extends Parent { /* 子類別的實踐過程 */ }
```

根據條目 9.2.3 里氏替換原則，父類別所規劃出來的方法，子類別也必須跟進實踐；因此父類別一但新增了成員，子類別可能也會連帶運動到內部的設計。

另外，子類別由於繼承自父類別，而類別一次只能繼承自一個父類別，因此子類別的規格也就會跟父類別綁得死死的；程式碼想要擴展功能的唯一地點就是父類別內部的設計，而子類別可能也會隨著父類別的設計改變，行為也就跟著改變。

還有一種常見的案例就是直接違反依賴倒轉原則（條目 9.2.5），也就是程式模組直接使用其他程式模組實踐出來的東西，這就是產生出直接耦合的證據。以條目 9.1.3 曾出現過的 cake 函式之宣告：

```
function cake(flavor: 'Chocolate' | 'Vanilla' | 'Strawberry') {  
    let milk = new Milk(2); // ← 兩公升的牛奶  
  
    // 其他步驟 ...  
}
```

該函式與類別 Milk 直接耦合，這應該沒什麼話好講。

相對來說，解耦就是將以上出現的直接耦合情形解構掉，就這麼簡單。（但請記得，解耦過程並不一定簡單啊！）

9.3.2 物件的委任 Object Delegation

常見的解耦方式就是所謂的物件的委任（Object Delegation），以下舉簡單的案例進行探討。

```
/* 角色 */  
class Character {  
    constructor(  
        public name: string,  
    ) {}  
  
    public attack(enemy: Character) {  
        console.log(`"${this.name}" is attacking "${enemy.name}"`);  
    }  
}  
  
/* 戰士 */  
class Warrior extends Character {  
    constructor(name: string) {  
        super(name);  
    }  
  
    public attack(enemy: Character) {
```

```
    console.log(` ${this.name} pierces through ${enemy.name} using a sword.`);
}
}

/* 巫師 */
class Wizard extends Character {
  constructor(name: string) {
    super(name);
  }

  public attack(enemy: Character) {
    console.log(` ${this.name} freezes ${enemy.name} using a spell.`);
  }
}
```

以上的範例分別宣告三種不同的類別，分別是 `Character`、`Warrior` 與 `Wizard`。其中，`Character` 為 `Warrior` 與 `Wizard` 的父類別，也就是說後兩的類別與 `Character` 是直接耦合的關係，使用方式如下：

```
/* 戰士角色的建構 */
let max = new Warrior('Max');

/* 巫師角色的建構 */
let martin = new Wizard('Martin');

/* 互相傷害 */
max.attack(martin);
// => 'Max pierces through Martin using a sword.'

martin.attack(max);
// => 'Martin freezes Max using a spell.'
```

表面上這樣的程式架構似乎沒有太大的問題，不過一旦功能變多了，角色類別 `Character` 的設計可能會更加複雜：

```
/* 角色 */
class Character {
  constructor(
    public name: string,
```

```
public healthPoint: number, // ← 生命值
public strength: number, // ← 力量
public intelligence: number, // ← 智慧
) {}

public attack(enemy: Character) {
    console.log(` ${this.name} is attacking ${enemy.name}. `);
}

/*
戰士 */
class Warrior extends Character {
    constructor(name: string, healthPoint: number, strength: number) {
        super(name, healthPoint, strength, 0);
    }

    public attack(enemy: Character) {
        console.log(` ${this.name} pierces through ${enemy.name} using a sword. `);

        /* 戰士攻擊時會對對方造成一定程度的物理性傷害 */
        enemy.healthPoint -= this.strength;
    }
}

/*
巫師 */
class Wizard extends Character {
    constructor(name: string, healthPoint: number, intelligence: number) {
        super(name, healthPoint, 0, intelligence);
    }

    public attack(enemy: Character) {
        console.log(` ${this.name} freezes ${enemy.name} using a spell. `);

        /* 法師攻擊時會對對方造成一定程度的魔法性傷害 */
        enemy.healthPoint -= this.intelligence;
    }
}
```

首先，以上的程式碼，由於類別繼承的特性，父類別一但有新的成員倍增加時，子類別也不得不作出一些變化。

因此，為了解耦，此時就有一種稱作為物件委任（Object Delegation）技巧，做法是這樣：

將要被解耦的目標模組（在這裡是指子類別）替換為主模組（父類別）的成員變數指涉目標；主模組的任何行為皆由被指涉的目標主導，也就是說，將主模組的行為委任給被指涉的模組執行。

剛開始看不懂是正常的，用程式碼來解釋就是將原本 `Character` 類別的設計改成以下的模式：

```
interface CharacterInfo {
    healthPoint: number;
    strength: number;
    intelligence: number;

    attack(character: Character, enemy: Character): void;
}

class Character {
    constructor(
        public name: string,
        public character: CharacterInfo // ← 指涉目標為 CharacterInfo 介面
    ) {}

    public attack(enemy: Character) {
        this.character.attack(this, enemy);
    }
}
```

首先，本範例額外宣告了新的介面 `CharacterInfo` 代表的是 `Warrior` 與 `Wizard` 的基礎規格（生命值、力量等能力值），將 `Character` 原本直接耦合的類別（也就是本案例要解耦的目標類別 `Warrior` 與 `Wizard`），改成使用成員變數方式與 `CharacterInfo` 介面進行抽象耦合。

另外，原本 `Character` 可以做的行為是呼叫 `attack` 方法進行攻擊；這一次的改寫則是委任被指涉的抽象目標（也就是 `CharacterInfo`）進行攻擊。

你會發現 `Character` 不再管細節層面的資料，例如生命值（Health Point）以及力量（Strength）等能力值，它只會專注描述一個角色可以做的事情（例如：攻擊敵人），符合條目 9.2.1 講到的單一職責原則。（`Character` 的責任減輕了！）

而我們只要按照里氏替換原則，根據 `CharacterInfo` 介面設計出 `Warrior` 與 `Wizard` 這兩個類別，就可以將這兩個類別與 `Character` 進行抽象耦合：

```
class Warrior implements CharacterInfo {
    public intelligence: number = 0;

    constructor(
        public healthPoint: number,
        public strength: number
    ) {}

    public attack(character: Character, enemy: Character) {
        console.log(
            `${character.name} pierces through ${enemy.name} using a sword.`);
    }

    /* 戰士攻擊時會對對方造成一定程度的物理性傷害 */
    enemy.character.healthPoint -= this.strength;
}

class Wizard implements CharacterInfo {
    public strength: number = 0;

    constructor(
        public healthPoint: number,
        public intelligence: number
    ) {}

    public attack(character: Character, enemy: Character) {
        console.log(
            `${character.name} freezes ${enemy.name} using a spell.`);
    }
}
```

```
/* 法師攻擊時會對對方造成一定程度的魔法性傷害 */
enemy.character.healthPoint -= this.intelligence;
}
}
```

我們專注 `Warrior` 程式碼中的建構子函式前後改寫的差異：

```
/* 改寫前 ... */
class Warrior extends Character {
  constructor(name: string, healthPoint: number, strength: number) {
    super(name, healthPoint, strength, 0);
  }

  // 其他成員略 ...
}

/* 改寫後 ... */
class Warrior implements CharacterInfo {
  public intelligence: number = 0;

  constructor(
    public healthPoint: number,
    public strength: number
  ) {}

  // 其他成員略 ...
}
```

你會發現，原本 `Warrior` 與它的父類別 `Character` 是很親密的關係（因為必須使用 `super` 關鍵字初始化父類別的成員變數值）；經過物件委任的結果，`Warrior` 與 `Character` 正式脫離了直接耦合的繼承關係，取而代之的是——`Warrior` 專注於描述戰士的能力性質與攻擊的手法，剩下就是根據 `CharacterInfo` 介面與父類別 `Character` 進行抽象耦合。（圖 9-4 為原本實踐的類別關係圖，圖 9-5 為抽象耦合後的關係圖）

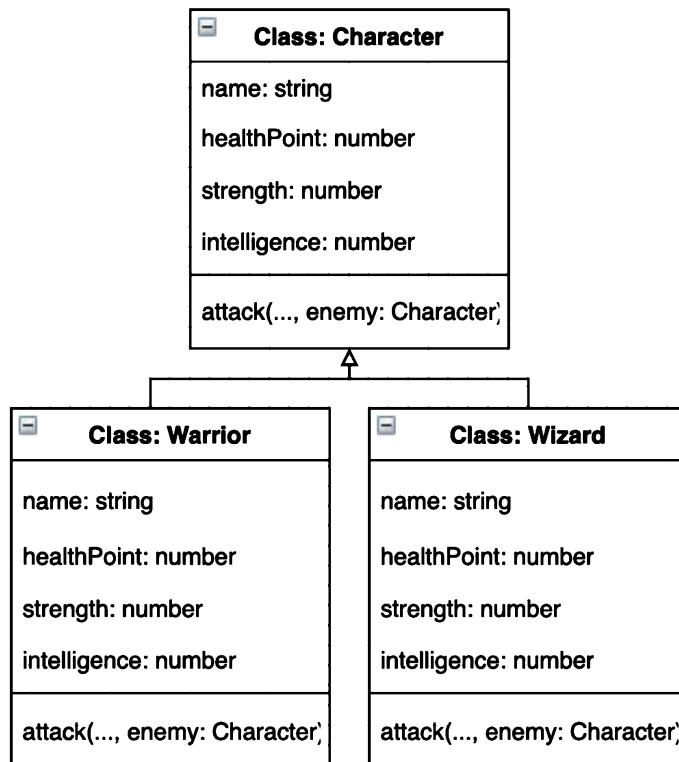


圖 9-4 Character 與其他子類別直接耦合的關係圖

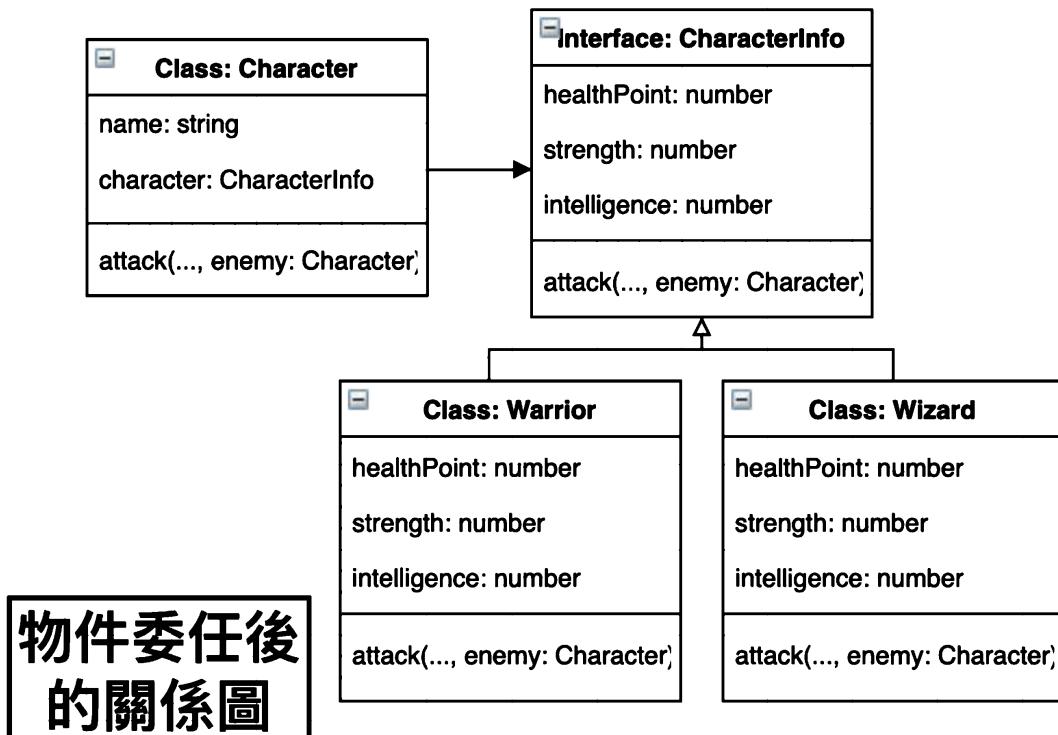


圖 9-5 採用物件的委任後，抽象耦合的結果

仔細觀察圖 9-5，不覺得這個模式看起很眼熟嗎？這不就跟條目 9.2.5 中的圖 9-2 簡直很相向，儘管該篇章講的案例是 `cake` 函式與乳製品相關的介面 `DairyProduct` 進行抽象耦合的關係，但我們也可以把該案例想成，`cake` 函式裡凡是跟乳製品相關的行為，我們都會藉由 `DairyProduct` 介面，將要執行的行為委任給該類別指涉的物件來執行！

另外，你會發現圖 9-5 比起圖 9-4，抽象耦合過後的寫法，`Character` 的設計變得更為簡單，而 `Warrior` 與 `Wizard` 類別也不會與 `Character` 的設計綁在一起；這也相對驗證出，物件的委任是基於依賴倒轉原則延伸出來的技巧。

9.3.3 依賴的注入 Dependency Injection

此外，物件的委任又跟另一個技巧很像 —— 依賴的注入（Dependency Injection），不過作者認為物件的委任實質上就是依賴的注入，只是解釋的看法角度不同而已。

譬如說，資料庫的設計會有代表資料庫連線（Database Connection）的物件，任何要跟資料庫溝通的行為都必須依靠此物件，假設陽春的資料庫類別長這樣：

```
class Database {  
    // 資料庫的連線物件  
    public connection = new SQLDatabase();  
  
    // 執行資料庫的指令並回傳結果  
    public execute(query: string) {  
        return this.connection.execute(query);  
    }  
  
    // 關掉與資料庫的連線  
    public close() {  
        this.connection.close();  
    }  
  
    // 其他成員方法 ...  
}
```

你會發現，它跟物件的委任很像的點在於，`Database` 類別若要執行一些方法（諸如 `execute` 或 `close` 方法），都會委任 `connection` 成員所指涉的物件來執行。

然而，這裡並非為物件委任的寫法，原因是 `connection` 指涉的是一個實體（`Instance`），因此造成直接耦合的情形；因此如果想要改成 `Database` 可以用各種不同版本的資料庫連線方式，必須改成：

```
interface DatabaseConnection {
    close(): void;
    execute(query: string): /* execute 方法的回傳型別 ... */;

    /* DatabaseConnection 的其他規格 ... */
}

class Database {
    // 資料庫的連線物件可以為各種不同的東西，只要符合所謂 DatabaseConnection 這個介面
    constructor(public connection: DatabaseConnection) {}

    // 成員方法略 ...
}
```

這樣 `Database` 就可以擁有不只一種資料庫連接的版本：

```
let db1 = new Database(new SQLDatabase());
let db2 = new Database(new MySQLDatabase());
let db3 = new Database(new PostgreSQLDatabase());
// ... 更多不同版本的資料庫實體 ...
```

這種寫法就好像是將相依的模組（這裡指的是 `SQLDatabase`、`MySQLDatabase` 以及 `PostgreSQLDatabase`）從外面注入（Inject）到 `Database` 類別裡，單單這樣的技巧就得到一個正式名稱——依賴的注入。

依賴注入的解耦手法僅僅是把直接依賴（或直接耦合）的部分，藉函式傳參數的方式注入到主要的模組裡，只是注入的物件必須遵從同一個介面的設計；如果注入的東西，其介面都長得不一樣，這樣子主模組要如何使用這些被注入的東西呢？

看看這兩個名詞：依賴倒轉（Dependency Inversion）以及依賴注入（Dependency Injection）是不是長得很像？

有些網路資訊會直接將這兩個名詞劃成等號，但作者認為它們並不全等，甚至代表性質與意義差很多；它們的性質分別如下：

依賴倒轉（Dependency Inversion）是一個原則（Principle），描述的是模組解耦的方法。（將 High-Level 與 Low-Level 的模組共同依賴於一個抽象介面）依賴注入（Dependency Injection）則是將外來的模組注入到主要的模組裡，最常見的手法就是用參數傳遞方式注入，避免模組直接耦合的情形。

因此依賴倒轉跟依賴注入的關係是：

依賴倒轉原則描述的解耦手法中——其中一種就是依賴的注入。

前者是原則（Principle），後者是技法（Technique）；正如同純理論以及對應的實際應用的關係。（所以請不要把「依賴倒轉」與「依賴注入」劃上全等號囉！）

回歸前面的 `Character` 類別的範例，當角色要進行某些行為（比如呼叫 `attack` 方法）時，就會委託 `character` 指涉到的 `CharacterInfo` 執行。不過這裡用依賴注入的角度來解釋也是可以的——將 `CharacterInfo`（也就是 `Warrior` 與 `Wizard` 類別）注入到 `Character` 類別裡，使得 `Character` 具備更多彈性。

至於想要用什麼樣的角度看待物件的委任或依賴的注入，基本上觀念對就沒有太大的問題。

10

常用 ECMAScript 標準語法

本章節要探討的東西是 ECMAScript 標準釋出的語法，讀者可能以為這早在學習原生 JavaScript 的時候應該接觸到很熟了；然而，在 TypeScript 裡搭配了基礎篇章講到的型別系統又會是從另一個層面學習 ECMAScript 標準，尤其又以第七章講到的泛用型別的應用更常看到。

► 10.1 ES6 解構式 Destructuring

10.1.1 解構式語法基礎

基本上，熟悉如何在原生 JavaScript 使用 ES6 解構式基礎語法的讀者可以跳過本條目，但如果是直接初學的人或者是從非 JavaScript 背景的人可以稍微瀏覽看看喔。

作者認為自己最常使用也最依賴的東西就是解構式語法。以下面的範例為例：

```
let maxwell = {
  name: 'Maxwell',
  age: 18,
  interest: ['drawing', 'programming']
};
```

宣告的物件 `maxwell` 就是一般常見的 JSON 物件，如果說想要宣告一個函式，專門填入該物件並且印出所有細節時：

```
function logInfo(person) {
    const name = person.name;
    const age = person.age;
    const interest = person.interest;

    const interestStr = interest.join('.');
    console.log(
        `${name} is ${age} years old, and interested in ${interestStr}`
    );
}
```

仔細看這三行的寫法：

```
const name = person.name;
const age = person.age;
const interest = person.interest;
```

以上的做法是將物件的值取出來（也就是呼叫對應屬性）並且指派結果到變數裡。但最常見的變數命名方式是將變數命名的結果與呼叫的對應屬性相同，才最造成以上三行的寫法。

不過這種方式挺冗贅的，因此 ECMAScript 標準提供更簡潔的寫法，就是解構式。以上三行的程式碼可以簡化成如下：

```
const { name } = person;
const { age } = person;
const { interest } = person;
```

做法很簡單，因為解構的東西是 JSON 物件，所以運用 JSON 物件的寫法，將屬性解構出來之後宣告成變數。由於 JSON 物件可以容納多個屬性（廢話），所以解構式當然可以從同一個物件一次指派多個屬性出來：

```
const { name, age, interest } = person;
```

但是若遇到這種案例，也就是變數名稱不等於屬性名稱，該如何處理？

```
const username = person.name;
```

解構式語法可以使用冒號的方式更改解構屬性的變數名稱：

```
const { name: username } = person; // ← 解構的 name 屬性被指派到變數 username
```

此外，解構式中，使用者也可以指派預設值，以防解構的屬性是不存在物件的喔！

```
const { name = 'Max' } = person; // ← 解構的 name 屬性若不存在時，預設為 'Max'
```

當然，以上的更改指派變數名稱以及預設值可以合在一起使用：

```
const { name: username = 'Max' } = person;
```

另一方面，陣列也是可以被解構的物件，但是作者很少使用，因為不太直觀。以下的範例是解構陣列裡的第一個與第二個元素的寫法：

```
const arr = [1, 2, 3, 4, 5];
const [firstElement, secondElement] = arr;

console.log(firstElement); // => 1
console.log(secondElement); // => 2
```

講完基礎解構式後，最前面的案例，也就是函式 `logInfo` 的宣告可以改寫成：

```
function logInfo(person) {
  const { name, age, interest } = person;

  const interestStr = interest.join('.');
  console.log(
    `${name} is ${age} years old, and interested in ${interestStr}`
  );
}
```

是不是看起來簡潔多了？

不過還有更簡易的寫法是在函式的參數部分直接解構也是可以的喔！

```
/* 將 person 參數解構成 name, age 以及 interest */
function logInfo({ name, age, interest }) {
  // 以下略 ...
}
```

以上是常見的解構式語法的介紹。

10.1.2 解構的同時進行型別註記

作者真正想講的重點不是解構式語法有多好多棒（它本來就很好很棒），而是使用解構式的同時能夠進行型別的註記（Annotation）嗎？

讀者可能想說如果你寫成這樣：

```
const { name: number, age: string, interest: string[] } = person;
```

根據條目 10.1.1 講述的解構是規則，這樣就會等效於以下的程式碼：

```
const number = person.name;
const string = person.age;
const string[] = person.interest;
```

撇除前兩個屬性的解構式寫法沒問題，第三個解構的結果根本是不合理的啊！

```
const string[] = person.interest; // ← 這根本是錯誤的解構指派式啊
```

那這樣還可以同時解構物件並且進行註記嗎？

答案是可以的，你必須用到 JSON 物件型別的寫法，而非單純型別一個個對解構的屬性進行註記。

正確的註記寫法如下：

```
const { name, age, interest }: {
    name: string;
    age: number;
    interest: string[];
} = person;
```

當然，如果該 JSON 物件型別有用型別化名¹的方式宣告時，可以用該化名取代以上的寫法。

¹ 型別化名（Type Alias）的宣告，請參見本書條目 3.1.3。

```
type PersonalInfo = {
    name: string;
    age: number;
    interest: string[];
};

const { name, age, interest }: PersonalInfo = person;
```

如果萬一遇到解構的屬性之變數更名的情形時，註記的型別之 JSON 物件型別結構請以被指派的物件的結構為主。假設將解構屬性 `name` 的名稱，指派到變數 `username` 時，就算是這樣，你還是要以被指派的物件（也就是 `person` 物件）的結構來註記：

```
const { name: username, age, interest }: {
    /**
     * 就算解構屬性 name 被更名為 username，還是要按照被指派的物件，
     * 也就是 person 本身的結構描述註記的型別
     */
    name: string;
    age: number;
    interest: string[];
} = person;
```

如果解構的東西為函式的參數，也可以遵照同樣的想法同時解構並且註記函式參數的型別結構：

```
/* 將 person 參數解構成 name, age 以及 interest，並且註記為 PersonalInfo 屬性 */
function logInfo({ name, age, interest }: PersonalInfo) {
    // 以下略 ...
}
```

至於陣列的解構，對應的註記型別可能是陣列型別（注意「可能」兩字），例如：

```
const arr = [1, 2, 3, 4, 5];
const [firstElement, secondElement]: number[] = arr;
```

畢竟根據註記的型別必須符合被指派的物件值之型別的規則，變數 `arr` 為 `number[]` 型別，理應來說，解構同時必須註記的型別為 `number[]`。

然而，如果說解構的東西是元組型別²的話，理應來說，解構的同時必須註記為元組型別：

```
type FiveNumbers = [number, number, number, number, number];
const arr: FiveNumbers = [1, 2, 3, 4, 5];
const [firstElement, secondElement]: FiveNumbers = arr;
```

然而，如果你改成以下的寫法還是可以被接受的，不過作者還是建議根據被指派的變數值而來註記解構式的型別。

```
const arr: FiveNumbers = [1, 2, 3, 4, 5];
const [firstElement, secondElement]: number[] = arr;
```

► 10.2 汇集 - 展開操作符 Rest-Spread Operator

10.2.1 汇集 - 展開操作符語法基礎

與條目 10.1.1 的概念相同，熟悉此語法在 JavaScript 運作過程的讀者，大致上可以跳過本條目喔～

匯集 - 展開操作符英文是 Rest-Spread，所以有資源會翻譯成「其餘（剩餘） - 展開操作符」，這個譯名也是可以的！不過作者是根據使用本條目要討論的操作符之功能特性，採用「匯集」這個詞作為翻譯。

匯集 - 展開操作符的符號是三個點點，也就是 ...。可是它有兩種用途——「匯集」以及「展開」，這就是為何此操作符有這麼冗長的名稱。

先講後者的功能，也就是「展開」的部分——什麼樣的東西是可以被展開的呢？應該也只有 JSON 物件或陣列看起來像是可以被展開的東西吧！

2 元組型別 (Tuple Type)，請參見本書條目 4.1。

譬如說，`Math.max` 這個函式專門在找一系列數字裡的最大數字：

```
console.log(Math.max(5, 1, 7, 2, 4));  
// => 7
```

但是如果今天遇到的輸入是一個陣列時，以下的寫法就是錯誤的，畢竟 `Math.max` 都展示過就只接受一系列的數字作為參數了。

```
console.log(Math.max([5, 1, 7, 2, 4])); // ← 錯誤的寫法
```

通常解法就是用 `Function.prototype.apply` 方法來處理這一類需求：

```
console.log(Math.max.apply(undefined, [5, 1, 7, 2, 4])); // ← 錯誤的寫法
```

只是這樣的寫法真的很不直觀，於是我們可以藉由展開操作符（Spread Operator）幫助我們將輸入的陣列展開成為函式的參數：

```
console.log(Math.max(...[5, 1, 7, 2, 4])); // ← 使用展開操作符
```

此外，對陣列進行展開運算符的相關操作的另一個用途是複製新的陣列出來：

```
const originalArray = [1, 2, 3, 4, 5];  
const copiedArray = [...originalArray]; // ← 複製並產出新的陣列
```

再者，陣列也可以運用此操作符進行結合（Concatenation）的動作：

```
const arr1 = [1, 2, 3];  
const arr2 = [4, 5, 6];  
const concatenatedArray = [...arr1, ...arr2]; // => [1, 2, 3, 4, 5, 6]
```

另一方面，展開運算符也可以用在 JSON 物件上，例如複製物件：

```
const originalJSON = { propA: 123, propB: 'Hello world' };  
const copiedJSON = { ...originalJSON }; // ← 複製並產出新的物件
```

但請注意，如果遇到 JSON 物件有多層物件的情形，這種複製手法並不是深度複製，而是俗稱的淺層複製（Shallow Copy）：

```
const originalJSON = { propA: { propB: 'Hello world' } };  
const copiedJSON = { ...originalJSON }; // ← 複製並產出新的物件
```

```
console.log(originalJSON.propA.propB); // => 'Hello world'

/* 修改複製過後的物件，但如果竄改深層的值依然會影響到複製前的物件 */
copiedJSON.propA.propB = 123;
console.log(originalJSON.propA.propB); // => 123
```

當然，也可以利用展開操作符，將多個物件的屬性合併重組起來：

```
const json1 = { propA: 123, propB: 'Hello world' };
const json2 = { propC: true, propD: [1, 2, 3] };

/* 合併 json1 與 json2 的屬性 */
const combinedJSON = { ...json1, ...json2 };
// => { propA: 123, propB: 'Hello world', propC: true, propD: [1, 2, 3] }
```

相對於展開操作符，匯集操作符（Rest Operator）的效果是相反的，而且多半會與條目 10.1 講述到的解構式語法使用。主要目的就是匯集（陣列的）元素或（物件的）屬性的概念。

假設今天我們想要將陣列裡的第一個拔出來，並且將剩餘的元素匯集成另一個陣列時，我們可以這麼做：

```
const arr = [1, 2, 3, 4, 5];
const [firstElement, ...restElement] = arr;

console.log(firstElement); // => 1
console.log(restElement); // => [2, 3, 4, 5]
```

當然，匯集操作符的功用也可以套用在解構 JSON 物件的案例，譬如以條目 10.1.1 的 `maxwell` 物件作為案例，假設僅拔出 `interest` 屬性，其餘的屬性匯集成獨立的 JSON 物件：

```
const { interest, ...otherProps } = maxwell;

console.log(interest); // => ['drawing', 'programming']
console.log(otherProps); // => { name: 'Maxwell', age: 18 }
```

此外，函式的部分，如果說想要設計類似 `Math.max` 這一類的方法，必須要用

到以下的方式進行宣告：

```
function max() {
    let result = -Infinity;

    for (let i = 0; i < arguments.length; i += 1) {
        if (arguments[i] > result) {
            result = arguments[i];
        }
    }

    return result;
}
```

就算是作者，宣告此函式時，偶爾還是會查一下特殊關鍵字 `arguments` 的用法。這樣的寫法稍嫌不友善了些，不過習慣過後覺得也是還好。

而匯集操作符可以將以上的函式宣告改寫成更平易近人的寫法：

```
function max(...input) {
    let result = -Infinity;

    for (let i = 0; i < input.length; i += 1) {
        if (input[i] > result) {
            result = input[i];
        }
    }

    return result;
}
```

此函式的參數 `input` 會將填入的參數匯集起來，成為普通的陣列型別，於是你就可以直接省去原生 JavaScript 裡的 `arguments` 寫法³（更何況 `arguments` 本身並不是陣列，只是看起來可以用陣列的方式取值）。

³ 函式宣告裡的特殊關鍵字 `arguments`，參見 MDN：<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments>。

10.2.2 結合型別系統

一般的展開操作符的寫法，註記型別時，不太會有什麼需要注意的地方，畢竟以下的寫法應該很直觀吧！

```
const originalArray = [1, 2, 3, 4, 5];
const copiedArray: number[] = [...originalArray];
```

匯集操作符與解構式的搭配，並且註記型別時，規則跟條目 10.1.2 討論的規則一模一樣，不管你是解構成什麼樣的樣子，一律都是以被指派的值之型別為主：

```
const arr = [1, 2, 3, 4, 5];
const [firstElement, ...restElement]: number[] = arr;
```

所以就算前一個案例，解構式的型別：

`[firstElement, ...restElement]` 的型別看起來是（但實際上不是）`[number, number[]]`

但是要以後面的 `arr` 對應的型別為主，也就是 `number[]` 型別。

如果說解構的東西是 JSON 物件，以條目 10.1.1 的 `maxwell` 物件作為範例：

```
type PersonalInfo = {
  name: string;
  age: number;
  interest: string[];
};

const { interest, ...otherProps }: PersonalInfo = maxwell;
```

這時註記的寫法依然還是按照被指派的物件本身的結構註記型別，儘管解構式使用到匯集操作符。

另外，若匯集操作子放在函式部分時，以條目 10.2.1 出現的 `max` 函式例子，裡面的參數 `input` 的註記方式為：

```
function max(...input: number[]) {
  let result = -Infinity;
```

```
// 實作過程略 ...
}
```

我們希望使用者填入函式 `max` 的參數都是數字型別的值，然而因為運用到匯集操作符的語法，`input` 的型別必須為陣列型別。(如果不是的話就會出現如圖 10-1 的錯誤訊息)

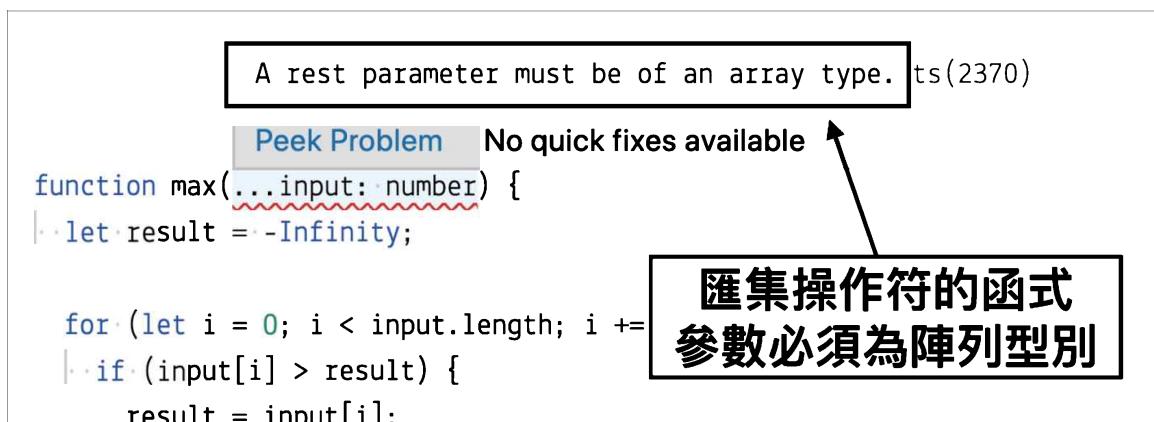


圖 10-1 使用匯集操作符的參數之註記型別必須為陣列型別

註記過後當然就可以防止使用者亂填任何不正確的值。(如圖 10-2)



圖 10-2 參數的型別註記過後，可以防止使用者亂填不符合規格的參數

► 10.3 ES6 Set 與 Map 資料結構

在進入本主題前，由於 `Set` 與 `Map` 為 ES6 標準裡新增的物件，在 `TypeScript` 設定檔——也就是 `tsconfig.json` 裡，必須要在 `lib` 設定部分（預設只有 `dom`）新增 `es6`（或 `es2015`）這個選項喔！

```
{  
  "compileOptions": {  
    // ... 其他 TypeScript 編譯器選項  
  
    "lib": ["dom", "ES2015"],  
  
    // ... 其他 TypeScript 編譯器選項  
  }  
}
```

10.3.1 Set 與 Map 的使用基礎

與條目 10.1.1 的概念相同，熟悉此語法在 JavaScript 運作過程的讀者，大致上可以跳過本條目喔～

ES6 出了兩種很好用的資料結構，分別為 **Set**（集合）與 **Map**（鍵值對集合）。

集合（Set）儘管跟陣列相似，都是一系列元素的集合，但最大差異在於——集合內部的元素不重複、並且不會有順序。（數學上也是這麼定義集合的）

```
const anArray = [1, 1, 2, 2, 2, 3, 4, 4, 5];  
const aSet = new Set([1, 1, 2, 2, 2, 3, 4, 4, 5]);  
  
console.log(anArray); // => [1, 1, 2, 2, 2, 3, 4, 4, 5]  
console.log(aSet); // => Set { 1, 2, 3, 4, 5 } 集合內部的元素不會重複
```

集合由於沒有順序之分，所以不會有陣列的 `Array.prototype.push` 或 `unshift` 方法（分別是插入最後一個以及第一個值），集合只有一個 `Set.prototype.add` 方法，純粹只是加入一個新的元素到集合內部。

```
aSet.add(6);  
console.log(aSet); // => Set { 1, 2, 3, 4, 5, 6 }
```

就算你新增集合內部原本就有的元素，按照集合的定義，元素並不會重複新增下去：

```
aSet.add(1);  
console.log(aSet); // => Set { 1, 2, 3, 4, 5, 6 }
```

集合還有提供多個不同的方法⁴讓你可以刪除或檢視集合內有沒有特定的元素：

```
aSet.delete(2);      // 刪除元素 2
console.log(aSet); // => Set { 1, 3, 4, 5, 6 }

// 檢視集合內有沒有元素 3
console.log(aSet.has(3)); // => true

// 檢視集合內有沒有元素 2
aSet.remove(2);
console.log(aSet.has(2)); // => false
```

另一個對應的資料結構為 ES6 Map⁵，它可以儲存鍵值對的東西；不過讀者可能感覺它跟 JSON 物件沒有什麼差別，但事實上，Map 支援的鍵是任何型別的值；相對的，JSON 的鍵只能為字串型別的值，就算你用其他型別的值作為 JSON 物件的鍵，它都會先轉型成字串型別作為其屬性。

```
const aJSONObject = { hello: 'world', num: 123 };

/* Map 建構子接收的是二維的陣列，內部為鍵值對的組合 */
const aMapObject = new Map([
  ['hello', 'world'],
  ['num', 123]
]);
```

Map 可以使用 `Map.prototype.set` 方法進行鍵值對的設置：

```
aMapObject.set('subscribed', true); // ← 設置新的鍵值對 ['subscribed', true]
```

當然也可以刪除屬性，利用 `Map.prototype.delete` 方法：

```
aMapObject.delete('hello'); // ← 刪除 'hello' 這個鍵
```

4 ES6 Set，參見 MDN：https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set。

5 ES6 Map，參見 MDN：https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map。

Map 也可以檢視鍵的存在性：

```
console.log(aMapObject.has('num')); // => true  
console.log(aMapObject.has('hello'));
```

而由於 Map 物件它可以接收的鍵是任何型別的值，也就是說數字、物件等等都可以作為 Map 的鍵，所以如果以下面的案例來看：

```
aMapObject.set(123, 456); // ← 設置新的鍵值對 [123, 456]  
aMapObject.set('123', 789); // ← 設置新的鍵值對 ['123', 789]
```

這個功能是普通的 JSON 辦不到的事情，因為屬性就算是數字型別的值，一樣會先轉型成字串型別後再設為屬性，因此以下的寫法，第二個字串 '123' 作為屬性宣告的結果會覆蓋掉前面的數字型 123 屬性對應的結果。

```
console.log({  
    123: 456,  
    '123': 789 // ← 這個宣告會覆蓋掉前面的屬性宣告值  
});  
// => { '123': 789 }
```

10.3.2 泛用類別 Set

是的，ES6 Set 與 Map 當然被 TypeScript 支援，至於支援的方式就是採用第 7 章討論到的泛用型別的形式呈現。

讀者可以運用到條目 1.5.3 討論到的 VSCode 快速查詢技巧，查詢 Set 與 Map 的型別宣告。本條目先從 Set 類別開始討論。

以下是 Set 的實體與建構子函式的介面宣告，為 Set 的型別定義⁶。

```
interface Set<T> {  
    add(value: T): this;  
    clear(): void;  
    delete(value: T): boolean;
```

⁶ 型別定義 (Type Definition)，請參見本書條目 8.3。

```

    forEach(
        callbackfn: (value: T, value2: T, set: Set<T>) => void,
        thisArg?: any
    ): void;
    has(value: T): boolean;
    readonly size: number;
}

/** 
 * 此為 Set 建構子函式的介面，此為類別型別 Class Types 的設計手法，
 * 由於本書並無討論此議題，因此放在備註7，有興趣的讀者再去深究
 */
interface SetConstructor {
    new <T = any>(values?: readonly T[] | null): Set<T>;
    readonly prototype: Set<any>;
}

declare var Set: SetConstructor;

```

這就是使用 TypeScript 的其中一個好處，我們可以直接在 VSCode 輕易地扒開類別的型別定義，檢視類別內部的規格。假設臨時忘記如何刪除 Set 裡的元素的方法到底是 `remove` 還是 `delete` 時，藉由扒開 Set 的定義宣告，我們得知是 `delete` 方法，因為規格裡有這麼一行：

```

interface Set<T> {
    // 略 ...
    delete(value: T): boolean;
    // 略 ...
}

```

此外，你可以看到 Set 被宣告的介面規格時，它有附帶一個型別參數 `T`；這個參數代表著集合 Set 內部的元素之型別，也就是說以下的建構子函式註記手法：

```
const aSet = new Set<number>([1, 1, 2, 2, 2, 3, 4, 4, 5]);
```

⁷ 類別型別 (Class Types)，請參見 TypeScript 官方文件：<https://www.typescriptlang.org/docs/handbook/interfaces.html#class-types>。

就代表 Set 裡面的元素不能為數字型別以外的東西。(如圖 10-3)



圖 10-3 偶然摻入字串型別的值造成的錯誤訊息

不過根據條目 7.2.3 討論的過程，提到：泛用型別推論絕對不是單向的，實質上是雙向的。這一點也可以在宣告 Set 的時候成立，TypeScript 的推論機制並沒有那麼笨。(參見圖 10-4)



圖 10-4 假設不去註記時，照樣可以推論出型別，不過建議多多少少註記一下

10.3.3 泛用類別 Map

另一方面，由於 Map 結構有分成鍵與值的部分，理應來說，它應該會有兩種不同的型別參數。

以下是 ES6 Map 的型別定義宣告。

```
interface Map<K, V> {
  clear(): void;
  delete(key: K): boolean;
  forEach(
    callbackfn: (value: V, key: K, map: Map<K, V>) => void,
    thisArg?: any
```

```

): void;
get(key: K): V | undefined;
has(key: K): boolean;
set(key: K, value: V): this;
readonly size: number;
}

/**
 * Map 的建構子函式本身的介面，看起來很複雜，讀者看不懂可以暫時跳過，
 * 有興趣再研究，畢竟這不是本書討論的範圍，請參考備註 7
 */
interface MapConstructor {
  new(): Map<any, any>;
  new<K, V>(entries?: readonly (readonly [K, V])[] | null): Map<K, V>;
  readonly prototype: Map<any, any>;
}

declare var Map: MapConstructor;

```

基本上，條目 10.3.1 談到跟 Map 有關的方法都可以藉由扒開型別定義的部分檢視規格，不需要再上網慢慢爬文件，省去幾分鐘的時間。

使用 Map 時與 Set 有類似概念，不過它有兩個型別參數，第一個代表 Map 的鍵之型別，第二個則是值的型別。

```

/* 以下的 numMapObject 之鍵必須為字串，值必須為數字型別 */
const numMapObject = new Map<string, number>([
  ['lucky number', 14],
  ['unlucky number', 13]
]);

```

如果填入不符的型別就會出現錯誤。

```

const numMapObject = new Map<string, number>([
  ['lucky number', 14],
  ['unlucky number', 13],

  [123, 456],           // ← 鍵不為字串型別
  ['hello', 'world'] // ← 值不為數字型別
]);

```

不過，如果沒有特別註記型別參數時，Map 會根據初始化時的值，反推鍵與值對應的型別。

```
const numMapObject: Map<string, number>
const numMapObject = new Map([
  ['lucky number', 14],
  ['unlucky number', 13],
]);
```

推論結果為一鍵必須為字串型別，值則是數字型別

圖 10-5 Map 物件沒有填入型別參數的推論結果

但是 Map 如果出現鍵值有兩種不同的型別時，它反而不會推論成型別聯集複合的結果，而是出現錯誤訊息。(如圖 10-6)

```
const numMapObject = new Map([
  ['lucky number', 14], // ← 值為 number 型別
  ['unlucky number', '13'] // ← 值為 string 型別
]);
```

```
type 'IteratorResult<readonly [unknown, unknown], any>' is missing
  Type 'IteratorYieldResult<(string | number)[]>' is missing
type 'IteratorResult<readonly [unknown, unknown], any>' is missing
  Type 'IteratorYieldResult<(string | number)[]>' is missing
  Type 'IteratorYieldResult<readonly [unknown, unknown], any>' is missing
    Type '(string | number)[]' is missing
const numMapObject = new Map([
  ['lucky number', 14], // ← 值為 number 型別
  ['unlucky number', '13'] // ← 值為 string 型別
]);
```

圖 10-6 Map 型別若沒有特別填入泛用型別參數，仍然不能出現兩種不同型別的值

改成以下的方式就沒有問題了：

```
const numMapObject = new Map<string, number | string>([
  ['lucky number', 14], // ← 值為 number 型別
  ['unlucky number', '13'] // ← 值為 string 型別
]);
```

► 10.4 ES10 非強制串接操作符 Optional Chaining Operator

本條目探討的東西是 TypeScript (版本 3.7) 於作者寫作時剛釋出的 ECMAScript 標準的語法支援，而 Optional Chaining 是眾多 JavaScript 開發者期盼的功能，寫起來讓人感到療癒。

有時候 JavaScript 程式裡，物件的屬性不一定會存在值，如果要確認物件內部的狀態時，你必須得這麼寫：

```
if (obj && obj.propA && obj.propA.propB) {  
    console.log('obj.propA.propB exists!');  
}
```

以上的範例就是先確保 `obj` 的存在（也就是 `obj` 不為 `null` 或 `undefined` 等偏向於 `False` 這一類的值），然後在相繼確認 `obj.propA` 與 `obj.propA.propB` 的存在。

運用非強制串接操作符時，也就是 `?.` 的記號，等效結果如下：

```
if (obj?.propA?.propB) { /* 略 ... */ }
```

不過要注意的事情是，由於 `False` 這一類的值包含數字 0 或者是空字串，但這樣的說法是不精確的。以下是運用串接操作符以及實際的程式碼對照：

```
/* 非強制串接操作符 */  
if (obj?.propA?.propB) { /* 略 ... */ }  
  
/* 以上的程式碼實際的編譯結果 */  
if (  
    (obj === null || obj === undefined) ? undefined :  
    (obj.propA === null || obj.propA === undefined) ? undefined :  
    obj.propA.propB  
) { /* 略 ... */ }
```

實際上，非強制串接操作符的效果是會先檢測串接前的值是否為 `null` 或 `undefined`，如果不是的話就會繼續串接下去。

另外，非強制串接操作不一定只有操作屬性，它可以用來處理呼叫陣列索引的行為：

```
/**  
 * 將非強制串接操作在陣列上，會變成先檢查陣列是否為 undefined 或 null；  
 * 若陣列存在，就可以直接串接並且呼叫該陣列的索引  
 */  
let arr = [1, 2, 3, 4, 5];  
arr?.[0];
```

當然，字串由於也具備索引的特徵，因此改成以下的形式也可以：

```
let message = 'Hello world';  
message?.[0];
```

不過，如果變數被確定並不是陣列或字串值時，TypeScript 也沒有傻到就直接串接上索引方式的呼叫行為。(以下的程式碼，錯誤訊息如圖 10-7)

```
let luckyNumber = 14;  
luckyNumber?.[0];
```

```
let luckyNumber = 14;  
luckyNumber?.[0];  
let luckyNumber: number  
  
Element implicitly has an 'any' type because expression of type '0' can't be  
used to index type 'Number'.  
Property '0' does not exist on type 'Number'. ts(7053)
```

數字不能使用索引的方式取值

圖 10-7 索引呼叫方式當然不能隨便亂呼叫

不過如果用條目 4-3 講到的可控索引型別的方式，模擬出陣列的索引呼叫行為時，也可以用索引呼叫方式進行非強制串接：

```
interface ListOfString {  
    [key: number]: string;  
}  
  
let strings: ListOfString = ['hello', 'world', 'TypeScript'];  
strings?.[0]; // ← 利用 Indexable Type 模擬出的陣列
```

既然索引呼叫的方式可以串接時，那麼函式的呼叫行為也可以改成非強制串接的方式呼叫：

```
aFunction?.(/* 參數 ... */);
```

當然，以上的寫法等效於：

```
aFunction === null || aFunction === undefined ?  
    undefined :  
    aFunction(/* 參數 ... */);
```

► 10.5 ES10 空值結合操作符 Nullish Coalescing Operator

另一個 TypeScript 3.7 版本支援的 ECMAScript 語法標準為空值結合操作符。此操作符通常用在預設值（Default Value）的設置。

JavaScript 裡有一種被稱作為短路語法（Short-Circuiting），比如：

```
let selectedNumber = obj.value && 14;
```

以上的程式碼會先確認如果 `obj.value` 不為偏向於 False 的值（如：`null`、`undefined`、數字 0 或空字串），就會將 `obj.value` 指派到變數 `selectedNumber` 裡，否則會將數值 `14` 指派到變數裡。

而數值 `14` 就是該變數的預設值。

不過呢，假設變數 `selectedNumber` 是指任何數字皆可以被指派的話，那麼數字 0 永遠不可能被指派進去，因為數字 0 在 JavaScript 隸屬於 False 值，因此必須改寫成以下的形式：

```
let selectedNumber = obj.value && (obj.value === 0 ? 0 : 14);
```

以上的寫法就變成是多增加一層確認 `obj.value` 是否為數字 0 的步驟。

但這種短路寫法寫久了，應該會希望有操作符是專門擋 `null` 或 `undefined` 這兩種值的短路語法，這就是 ECMAScript 出的空值結合操作符的主要功能。以上的範例可以改寫成下面的結果，效果相似：

```
let selectedNumber = obj.value ?? 14;
```

其中，符號 `??` 就是空值結合操作符，以上的程式碼編譯結果等效於以下的形式：

```
let selectedNumber = (obj.value === null || obj.value === undefined) ?  
    14 :  
    obj.value;
```

另外，假設你連 `obj.value` 也不確定 `obj` 的存在與否時，也可以結合條目 10.4 講到的非強制串接操作符，寫出以下形式的程式碼：

```
let selectedNumber = obj?.value ?? 14;
```

是不是看起來簡潔許多？不過一開始不習慣也是正常的，畢竟臨時多出一大堆問號的操作符出來，作者不敢保證所有剛接觸這兩個語法特色的人可以快速上手；用習慣了以後，可讀性也很高的。

► 本章練習

1. 請問 JSON 物件若含有多層，也可以使用解構式解構嗎？註記方式又為何？例如以下的程式碼，`maxwell.socialLinks.personalWebsite` 可以被解構出來嗎？

```
let maxwell = {  
    name: 'Maxwell',  
    age: 18,  
    socialLinks: { personalWebsite: 'example.com' },  
};
```

2. 請問以下的程式碼效果為何？

```
let arr = [1, 2];
[arr[1], arr[0]] = [arr[0], arr[1]];
```

3. 承上題，如果換成以下的程式碼，會有什麼結果？

```
let arr = [1, 2];
[arr[1], arr[0]] = arr;
```

4. 匯集 - 展開操作符可以對 JSON 物件進行展開的操作同時，將屬性再匯集到另一個 JSON 物件裡，因此你可能會看到利用匯集 - 展開操作符重新建構物件的語法：

```
let obj1 = { foo: 'Hello', bar: 123, baz: { message: 'world' } };
let obj2 = { ...obj1 };

console.log(obj1 === obj2); // => false
```

然而，此語法實際上只會進行淺層的物件複製（英文為 Shallow Copy），內層的屬性如果對到的也是物件時（比如 obj1.bar），複製出的物件之內層仍然會指向同一個物件：

```
console.log(obj1.baz === obj2.baz); // => true
```

因此如果對內層物件進行竄改時，原先被複製的物件對照的內層物件資料也會被竄改：

```
obj2.baz.message = 'goodbye';
console.log(obj1.baz); // => { message: 'goodbye' }
```

試寫出一個簡單的函式 deepCopy，深度複製（Deep Copy）輸入的物件，使得輸出物件的內層物件不會與複製前的物件有任何參照（Reference）上的關聯：

```
type JSONObj = { [key: string]: any };
function deepCopy(obj: JSONObj): JSONObj {
    // 實踐過程 ...
}
```

```
let obj1 = { foo: 'Hello', bar: 123, baz: { message: 'world' } };
let obj2 = deepCopy(obj1);

/* 對 obj2 進行竄改，obj1 不會受到影響 */
obj2.baz.message = 'goodbye';
console.log(obj1.baz); // => { message: 'world' }
console.log(obj2.baz); // => { message: 'goodbye' }

console.log(obj1 === obj2); // => false
```

本題不考慮 JSON 物件以外的情形，例如陣列或者是函式等，因此讀者不需要管輸入到 `deepCopy` 函式裡的 JSON 物件裡的值有陣列或函式以外的情形。

5. 請問陣列 Array 與 ES6 Set 有何差異性？
6. 請問 JSON 物件與 ES6 Map 有何差異性？
7. 提供任一組數字陣列（如：`[2, 3, 6, 1, 3, 6, 5, 9]`），請計算陣列的元素種類個數（本範例中，元素種類有 6 種，分別為：`1, 2, 3, 5, 6, 9`）。

11

常用 ECMAScript 標準語法 非同步程式設計篇

本章節要探討的東西也隸屬於 ECMAScript 標準釋出的語法範圍，但又是使用 JavaScript（或 TypeScript）開發時多多少少會見到的東西。使用 TypeScript 開發會多出型別系統的概念，因此作者會針對型別系統的部分對這些非同步程式設計進行補充。

另外，非 JavaScript 背景的讀者，如果直接學 TypeScript 時，建議一定要熟悉本章節的內容，畢竟 JavaScript 比較進階的東西就是同步與非同步程式的概念與差異。

► 11.1 同步與非同步的概念

11.1.1 同步的概念 **Synchronous**

「同步」表面上看起來是指很多程式同時執行，很容易被誤會，實際上它是指：

程式執行的過程與讀到的程式碼呈現同步狀態。

以下面的程式碼範例為主：

```
let foo = 123;           // ← 第一行：指派 123 到變數 foo 裡
let bar = 456;           // ← 第二行：指派 456 到變數 bar 裡

console.log(foo);        // ← 第四行：將 foo 的值印到 console 裡
console.log(bar);        // ← 第五行：將 bar 的值印到 console 裡

let baz = foo + bar;     // ← 第七行：指派 foo 與 bar 的總和到變數 baz 裡
console.log(baz);        // ← 第八行：將 baz 的值印到 console 裡
```

該範例會一行一行依照順序地執行；也就是說，執行順序不會跳來跳去，因此印出來的結果應該如下：

```
123   // ← 為 foo 的值
456   // ← 為 bar 的值
579   // ← 為 baz 的值，baz 為 foo 與 bar 相加總和
```

第二個關鍵點是：同步的程式具有阻塞性質（Blocking）。

你可以這麼想，由於程式碼在同步狀態是依序執行地，也就是說前一行若還沒執行完畢，絕對會卡在該行；一直到該行程式碼完成時才會讀取下一行程式碼繼續執行。

因此以下的程式碼，呼叫一個簡單的 loops 函式，該函式會毫無目的地執行數次：

```
function loops(times: number) {
  let count = 0;

  while (count < times) {
    count++;
  }
}

console.log('Loops started');
console.time('Duration');      // ← 設置計時器

loops(1000000);               // ← 讓迴圈執行 "好多" 次
```

```
console.timeEnd('Duration'); // ← 結束計時器  
console.log('Loops ended');
```

讀者可以任意將 `loops` 函式裡的值隨意更改，但你永遠會得到類似以下的結果：

```
Loops started  
Duration: 103.071ms           // ← 執行時間可能會有微差別  
Loops ended
```

這裡依舊可以連結到同步程式碼執行順序不變的特性，因此也就會延伸出程式有可能因為某行的執行過程很繁複，造成程式阻塞的可能性。

11.1.2 非同步的概念 Asynchronous

儘管大多數的程式可以按照同步程式碼的概念實踐出來，但仍然會有些應用是不太可能按照同步的特性實現，有可能的原因是實踐效果不理想、成本太大、亦或者是有很大的機率會碰到程式阻塞的問題，造成程式沒有效率。

前端的程式設計，很常遇到處理使用者與瀏覽器介面複雜的互動機制，這就是事件處理，或者是監聽事件（Event Listener）的註冊。

試想一件事情，如果事件處理過程是同步的程式達成的，會造成什麼樣的後果？

「今天開開心心的上 FxxxBook 時，結果網站載入時，竟然卡住了！後來發現原來是按鈕正等待著被我按下去，否則就會一直卡住。」——真是蠢到爆的網站設計。

因此，非同步的程式在 JavaScript 非常常見，在 JavaScript 裡註冊一個事件的語法使用的是 `addEventListener` 的方法：

```
/* 假設網頁 HTML 裡有按鍵元素：<button id="btn"> ... </button> */  
const $btn = document.getElementById('btn');  
  
if ($btn === undefined) {  
    throw new Error('Button is not found.');
```

```
}

console.log('Before event registered');

const clickEventHandler = function () {
    console.log('Click event triggered!');
};

/* 事件的註冊，填入回呼函式 clickEventHandler */
$btn.addEventListener('click', clickEventHandler);

console.log('Event registered!');
```

程式的執行過程中，印出來的結果一定會是以下的順序：

```
Before event registered
Event registered!
Click event triggered! // ← 如果使用者有按按鈕的話，按幾次就會印出幾次
Click event triggered!
Click event triggered!
...
...
```

以上的程式碼，儘管看似函式 `clickEventHandler` 會在印完 '`Before event registered`' 字串後立馬執行，但是實際上它以函式物件形式，作為監聽按鈕事件之接收者（Handler）的角色；因此執行順序就會隨著使用者按下按鈕後才會觸發函式裡的程式，而且按下幾次、就會觸發幾次，所以這裡相對來說並沒有按照程式碼一行一行的順序執行，理所當然就是非同步的執行狀態。

而且非同步程式並沒有阻塞問題，註冊完事件之後，JavaScript 的背景會監控使用者有沒有觸發此事件，然後再去排定事件觸發歷史，呼叫對應的回呼函式¹。

1 JavaScript 中的 Event Loop 是處理非同步程式的核心；每個非同步事件（或 API，例如：`setTimeout`）的註冊會記錄在所謂的 Event Table，而使用者會背景觸發到某個事件時，就會將事件排定在 Event Queue 裡（先進先出，First-in-first-out）；當程式碼中的 Call Stack 完全為空時，也就是整段 JavaScript 程式碼執行完畢後，會開始將 Event Queue 裡所有的觸發事件依序拉出來執行，這個依序執行的過程就是 Event Loop。

»» 同步與非同步的概念差異 Synchronous v.s. Asynchronous

1. 同步的概念是指程式會按照順序依序執行，但遇到複雜需要花時間的運算過程時，就會出現阻塞（Blocking）的情形。
2. 非同步的概念則是程式不一定按照順序執行，有可能會將某些程式放在背景執行，因此通常要避免阻塞的狀況，非同步程式的設計就會變得比較重要。

11.1.3 回呼函式地獄 Callback Hell

非同步的程式結構，以回呼函式（Callback Function）的使用最為明顯，畢竟用函式作為物件傳遞時，可以選擇在程式的任何一個時刻執行該函式——藉由回呼函式可以輕易改變一段程式碼執行的順序。

```
console.log('Before execution');

setTimeout(function callback() {
    console.log('Execute Callback');
}, 1000);

console.log('After execution');

// => Before execution
// => After execution

// 以下這一段會在程式執行約 1000ms 後出現
// => Execute Callback
```

回呼函式很容易使得程式碼形成層狀結構，通常一兩層看起來還好，但如果超過三層以上可能就得考慮程式碼重構的必要性。

```
/* 回呼函式中的回呼函式中的回呼函式中的回呼函式 */
setTimeout(function () {
    console.log('Layer 1');

    setTimeout(function () {
        console.log('Layer 2');
```

```
setTimeout(function () {
    console.log('Layer 3');

    setTimeout(function () {
        console.log('Layer 4');

        setTimeout(function () {
            console.log('Layer 5');
            }, 1000);
        }, 1000);
    }, 1000);
}, 1000);

// => Layer 1 ← 約第 1 秒出現
// => Layer 2 ← 約第 2 秒出現
// => Layer 3 ← 約第 3 秒出現
// => Layer 4 ← 約第 4 秒出現
// => Layer 5 ← 約第 5 秒出現
```

為了能夠輕易解決掉這個俗稱回呼函式地獄（有時候你會聽到厄運金字塔，也就是 Pyramid of Doom 這個稱呼）的問題，ECMAScript 的標準發展到現在，提供了一系列好用的工具幫助我們解決它。

► 11.2 ES6 Promise 物件

在討論本條目 Promise 物件之前，由於 Promise 屬於 ES6 標準下的物件，因此在 TypeScript 設定檔——也就是 `tsconfig.json` 裡，必須要在 `lib` 設定部分（預設只有 `dom`）新增 `es6`（或 `es2015`）這個選項喔！

```
{
  "compileOptions": {
    // ... 其他 TypeScript 編譯器選項

    "lib": [ "dom", "ES2015" ],
```

```
// ... 其他 TypeScript 編譯器選項
}
}
```

11.2.1 ES6 Promise 語法基礎

Promise 物件是整個非同步程式設計中最基礎的單位，最簡單的範例程式碼如下：

```
let p = new Promise(function (resolve, reject) {
    resolve('Hello Promise!');
});

/** 
 * 若 p 內部的程式執行過程沒有問題，並且呼叫了 resolve 函式時， 
 * 會執行 then 裡面的回呼函式
 */
p.then(function (value) {
    console.log(value);
});

/** 
 * 若 p 內部的程式執行過程出現錯誤，或者是呼叫了 reject 函式時， 
 * 會執行 catch 裡面的回呼函式
 */
p.catch(function (error) {
    console.log(error);
});
```

以上的程式碼，`p` 內部的程式會直接呼叫 `resolve` 方法並且傳入特定的值（也就是字串 '`Hello Promise!`'）；當 `resolve` 方法被呼叫時，`p.then` 裡的回呼函式就會執行，也就是將 `resolve` 函式傳遞的東西印出來。

但要是 `p` 內部的程式改成呼叫 `reject` 方法時：

```
let p = new Promise(function (resolve, reject) {
    reject('Errored!');           // ← 從 resolve 改成 reject
});
```

就會執行 `p1.catch` 裡的回呼函式，也就是把遇到的錯誤訊息印出來。

以上的程式碼可以用更簡潔的方式表達：

```
new Promise(function (resolve, reject) {
    resolve('Hello Promise!');
}).then(function (value) {
    console.log(value);
}).catch(function (error) {
    console.log(error);
});
```

注意：`resolve` 或 `reject` 可以被改名，畢竟它是作為 `Promise` 物件裡傳入函式中的參數，所以你可能會看到有程式碼取得名稱更簡潔，例如 `res` 代表 `resolve` 等；本書為了在講解時減少複雜度，固定只會取 `resolve` 或 `reject` 這兩個名詞。

11.2.2 Promise 串接鏈 Promise Chain

`Promise` 還有另一個重要性質——在 `then`（或者是 `catch`）方法裡，傳入的回呼函式可以回傳另一個 `Promise` 物件，使得 `Promise` 鏈可以不停地串接下去。

假設我們有一個函式 `resolveAfter` 專門回傳一個會在若干時間 `resolve` 的 `Promise` 物件，其定義方式如下：

```
/**
 * resolveAfter 會回傳一個 Promise 物件，
 * 該 Promise 物件會在特定時間才會 resolve
 */
function resolveAfter(milliseconds: number) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve(`Promise resolved after ${milliseconds}ms!`);
        }, milliseconds);
    });
}
```

假設如果想要實現一連串 Promise 依照順序執行的程式，一種寫法是：

```
resolveAfter(2000)
  .then(function (value) {
    console.log(value);

    /* 執行下一個 Promise 物件 */
    resolveAfter(3000).then(function () {
      console.log(value);

      /* 再執行下一個 Promise 物件 */
      resolveAfter(1000).then(function (value) {
        console.log(value);
      });
    });
  });
}

// => Promise resolved after 2000ms! // ← 程式從開始執行到這裡約 2 秒時間
// => Promise resolved after 3000ms! // ← 程式從開始執行到這裡約 5 秒時間
// => Promise resolved after 1000ms! // ← 程式從開始執行到這裡約 6 秒時間
```

然而，以上的寫法當然又是重蹈回呼函式地獄的覆轍，因此 **Promise** 允許你改寫成串接的形式：

```
resolveAfter(2000)
  .then(function (value) {
    console.log(value);

    /* 回傳另一個 Promise 物件 */
    return resolveAfter(3000);
  })
  .then(function () {
    console.log(value);

    /* 再回傳另一個 Promise 物件 */
    return resolveAfter(1000);
  })
  .then(function (value) {
    console.log(value);
});
```

```
// => Promise resolved after 2000ms! // ← 程式從開始執行到這裡約 2 秒時間  
// => Promise resolved after 3000ms! // ← 程式從開始執行到這裡約 5 秒時間  
// => Promise resolved after 1000ms! // ← 程式從開始執行到這裡約 6 秒時間
```

這樣攤平化後，就可以防止巢狀的部分繼續長下去。

11.2.3 Promise 物件狀態機 State Machine

Promise 物件的狀態分成三種：Pending（執行中，或等待結果狀態）、Resolved（完成）以及 Rejected（失敗）。

理所當然地，Pending 狀態就是在等待最後到底是被 `resolve` 函式或 `reject` 函式呼叫，而 `resolve` 函式就會觸發 Resolved 狀態；相對地，只要呼叫 `reject` 函式，就會觸發 Rejected 狀態。

最後就是，在 `then` 或 `catch` 的回呼函式裡，只要回傳新的 Promise 物件，又會再以該 Promise 物件重新回到 Pending 狀態。（如圖 11-1）

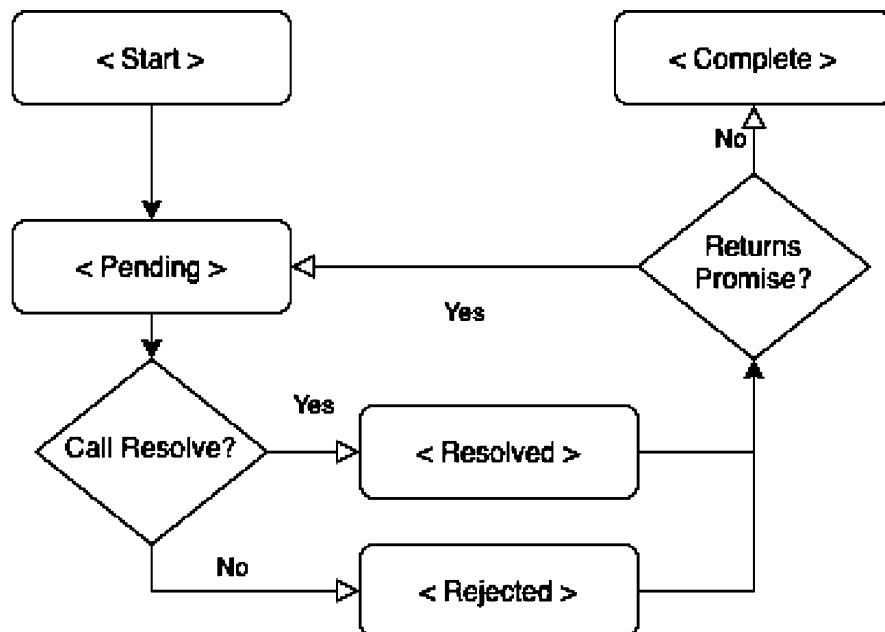


圖 11-1 Promise 物件簡易的狀態機流程圖

11.2.4 Promise 物件為泛用型別

前面的條目大概簡介了 Promise 物件的語法與性質，而在 TypeScript 的型別系統裡，Promise 物件本身是一種泛用類別²；作者鼓勵主動用條目 1.5.3 講到的技巧，檢視 Promise 物件在 TypeScript 型別宣告裡的規格。(如圖 11-2 為片段)



圖 11-2 為翻開 Promise 的定義檔部分的宣告

Promise 物件主要有一個型別參數，用途是註記 `resolve` 函式回傳的型別值。

譬如說，Promise 物件若在建立時，註記型別參數為字串型別時，`resolve` 函式回傳的東西也必須是字串型別值：

```
/* Promise<string> 必須 resolve 字串型別值 */
new Promise<string>(function (resolve, reject) {
    resolve('Hello Promise!');
});
```

如果你填入非字串型別的值到 `resolve` 函式，會出現如圖 11-3 的錯誤訊息。

2 泛用型別 (Generic Types)，請參見本書第 7 章。

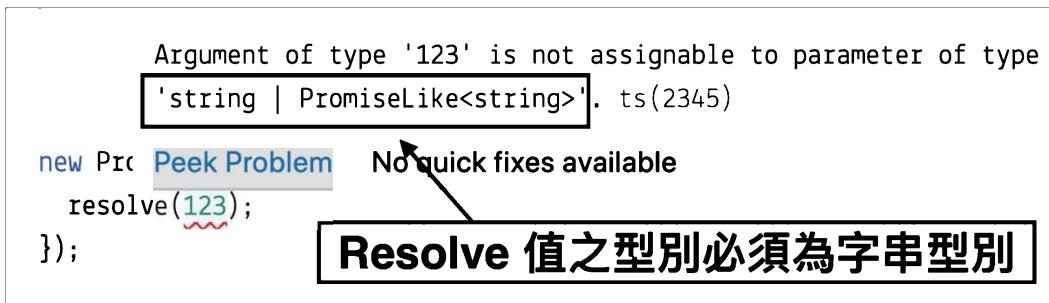


圖 11-3 填入非字串型別的值會出現的錯誤訊息

從錯誤訊息可以看到，如果 `resolve` 函式輸入非字串型別值，會告誡你必須傳入字串型別或 `PromiseLike<string>` 型別的值。

讀者可能很好奇 `PromiseLike<string>` 到底是什麼，不過由於這是進階話題，作者打算放在本章練習部分進行延伸。

不過，如果型別不去註記，那麼 `Promise` 物件會怎麼被推論呢？（如圖 11-4）

```
let p = new Promise(function (resolve, reject) {
    resolve('Hello Promise!');
});
```

```
let p: Promise<unknown>
let p = new Promise(function (resolve, reject) {
    resolve(123);
});
```

推論結果為 `unknown` 型別

圖 11-4 `Promise` 物件若不指定型別參數之值時，推論結果為 `Promise<unknown>`

這代表著如果要在 `then` 的回呼函式裡使用 `resolve` 所傳入的值，就必須積極註記，畢竟 `unknown` 型別³ 為的就是強迫開發者主動斷言它的型別⁴。

```
p.then(function (value) {
    /* 使用 as <TYPE> 進行型別斷言 */
```

3 `Unknown` 型別，請參見本書條目 4-6。

4 型別註記與斷言，請參見本書條目 2-2。

```
    console.log((value as string).toUpperCase());
});
```

»» ES6 Promise 狀態轉換

1. 通常想要將同步程式轉換成非同步程式時，可以將該程式包裝至 Promise 物件裡。
2. Promise 在建構的那一瞬間就會轉換到 Pending 狀態。
3. Promise 的建構子函式需要傳入一個回呼函式，裡面包含兩個參數；第一個參數為非同步程式執行完畢時，用以轉換到 Resolved 狀態的函式；第二個則是如果非同步函式出現錯誤時，可以呼叫並且轉換到 Rejected 狀態的函式。
4. 當轉換至 Resolved 狀態時，Promise 物件若有 then 方法，就會執行內部的函式，並且附帶 Resolve 的參數值。
5. 當轉換至 Rejected 狀態時，Promise 物件若有 catch 方法，就會執行內部的函式，並且附帶 Reject 的參數值。
6. Promise 物件的 then（與 catch）方法內的回呼函式，若回傳新的 Promise 物件，則會以該物件為主體，可以繼續串接 then（與 catch）方法下去，這個特性被稱之為 **Promise 串接鍊**。

11.2.5 常用的 Promise 物件 API

Promise 物件的使用方式事實上有很多種，本書列出常見的 API 語法。

有時候單純想要將某個值轉成 Promise 的格式直接進行 `resolve`（亦或者是 `reject`），如以下的範例：

```
/* 直接 Resolve */
let p1 = new Promise(function (resolve, reject) {
  resolve('Hello Promise!');
});

/* 直接 Reject */
let p2 = new Promise(function (resolve, reject) {
  reject('Error from Promise!');
});
```

其實可以用 Promise 物件提供的靜態方法⁵ `Promise.resolve`（以及 `Promise.reject`）簡化以上的寫法：

```
/* 直接 Resolve */
let p1 = Promise.resolve('Hello Promise!');

/* 直接 Reject */
let p2 = Promise.reject('Error from Promise');
```

這就是將原本是同步程式處理的值改以非同步方式將值包裝在 Promise 的簡化語法。

另外，有時候你希望程式執行的方式並不是序列（Series）的方式（也就是 Promise 串接鍊，按照順序一個個執行的程式），而是並列（Parallel）的方式執行；這時你可以採用 `Promise.all` 這個靜態方法，傳入多個不同的 Promise 物件，並且會等待所有 Promise 進入 **Resolved** 狀態，才會完整地 **Resolve**。

使用條目 11.2.2 宣告過的 `resolveAfter` 函式為例：

```
/* 全部一起等待完畢後一起 Resolve */
Promise.all([
    resolveAfter(3000),      // ← 3 秒後 Resolve
    resolveAfter(1000),      // ← 1 秒後 Resolve
    resolveAfter(5000),      // ← 5 秒後 Resolve
]).then(function (result) {
    console.log(result);
});

// 約 5000ms 會印出以下訊息
// => [
//     'Promise resolved after 3000ms!',
//     'Promise resolved after 1000ms!',
//     'Promise resolved after 5000ms!',
// ]
```

⁵ 類別靜態成員（Static Members），請參見本書條目 5.2.4。

另外，`Promise.all` 由於是等待多個 Promise 物件同時 Resolve，因此只要其中一個 Promise 物件進入 Rejected 狀態，就等於整個 Promise 會進入 Rejected 狀態：

```
/* 全部一起等待完畢後一起 Resolve */
Promise.all([
    resolveAfter(3000),           // ← 3 秒後 Resolve
    resolveAfter(1000),           // ← 1 秒後 Resolve
    resolveAfter(5000),           // ← 5 秒後 Resolve
    Promise.reject('Rejected')   // ← 整個會被 Reject 掉
]).then(function (result) {
    console.log(result);
}).catch(function (error) {
    console.log(`Error: ${error}`);
});

// 立即印出以下訊息
// => Error: Rejected
```

另外，由於 `Promise.all` 會將所有的值進行 Resolve，`then` 裡面傳的回呼函式，推論結果會是元組型別⁶ 的 Promise 物件。(以下的程式碼，`then` 方法裡的回呼函式之參數推論結果如圖 11-5)

```
Promise.all([
    Promise.resolve(123),          // ← 為 Promise<number> 型別
    Promise.resolve('Hello world'), // ← 為 Promise<string> 型別
    Promise.resolve(true),          // ← 為 Promise<boolean> 型別
]).then(function (result) {
    /* result 的型別推論為 [number, string, boolean] 型別 */
    console.log(result);
});
```

6 元組型別 (Tuple Type)，請參見本書條目 4.1。

```

Promise.all([
    Promise.resolve(123), ..... // ← 為 Promise<number> 型別
    Promise.resolve('Hello world') // ← 為 Promise<string> 型別
    Promise.resolve( (parameter) result: [number, string, boolean]
]).then(function (result) {
    /* result 的型別推論為 [number, string, boolean] 型別 */
    console.log(result);
});

```

元組型別的推論結果

圖 11-5 `Promise.all` Resolve 過後的結果為元組型別

另外，還有一種 Promise API 為 `Promise.race`，代表所有傳入的 Promise 物件會進行比賽，最先被 `Resolve` 的 Promise 就會以該 `Resolve` 結果作為整個 `Promise.race` 的結果。

```

/* 所有 Promise 比賽看誰最先被 Resolve */
Promise.race([
    resolveAfter(3000),      // ← 3 秒後 Resolve
    resolveAfter(1000),      // ← 1 秒後 Resolve
    resolveAfter(5000),      // ← 5 秒後 Resolve
]).then(function (result) {
    console.log(result);
});

// 沒有任何意外的話，會是 1000ms 的 Promise 物件最先被 Resolve
// => 'Promise resolved after 1000ms!'

```

而 `Promise.race` 既然可能會有任何一個 Promise 被 `Resolve`，因此 `then` 方法裡的回呼函式的值推論結果為所有傳入 Promise 的 `Resolve` 型別進行聯集（Union）複合的結果。（以下的程式碼範例，`then` 方法裡的函式參數推論結果如圖 11-6。）

```

Promise.race([
    Promise.resolve(123),           // ← 為 Promise<number> 型別
    Promise.resolve('Hello world'), // ← 為 Promise<string> 型別
    Promise.resolve(true),          // ← 為 Promise<boolean> 型別
]).then(function (result) {
    /* result 的型別推論為 number | string | boolean 型別 */
}

```

```
    console.log(result);
});
```

```
Promise.race([
  Promise.resolve(123), ..... // ← 為 Promise<number> 型別
  Promise.resolve('Hello world') // ← 為 Promise<string> 型別
  Promise.resolve( parameter ) result: string | number | boolean
]).then(function (result) {
  /* result 的型別推論 */
  console.log(result);
});
```

Promise.race 會將所有可能 Resolve 的型別進行聯集複合

圖 11-6 `Promise.race` 裡任何東西都可能 `Resolve`，因此推論為聯集複合的形式

»» ES6 Promise 常見 API

1. `Promise.resolve` 可以回傳一個立即進入 `Resolved` 狀態的 `Promise` 物件。
2. `Promise.reject` 可以回傳一個立即進入 `Rejected` 狀態的 `Promise` 物件。
3. `Promise.all` 可以傳入一系列的 `Promise` 物件，並且待所有物件進入 `Resolved` 狀態時，最外層的 `Promise.all` 回傳的 `Promise` 才會正式進入 `Resolved` 狀態；否則為 `Rejected` 狀態。
4. `Promise.race` 則是傳入一系列的 `Promise` 物件，最先進入 **Resolved**（或 **Rejected**）狀態的 `Promise` 物件就會以該 `Promise` 的狀態作為整個 `Promise.race` 的狀態。

► 11.3 ES7 非同步函式 Asynchronous Functions

11.3.1 以 `Promise` 為基礎的函式

普通的 JavaScript 函式以及非同步函式差異性到底在哪？以最簡單的例子來呈現：

```
/* 普通函式宣告 */
function syncMessage() {
  return 'Sync: Hello World!';
```

```

}

/* 非同步函式宣告時，使用 async 關鍵字 */
async function asyncMessage() {
    return 'Async: Hello World!';
}

```

以上面的例子來說，普通函式的推論結果想當然一定是 `() => string`，畢竟函式型別篇章⁷ 講過，輸出的型別可以經由 `return` 敘述式回傳的型別來推斷；然而，`asyncMessage` 儘管也是回傳字串型別的函式，可是推論結果就不是字串型別。(如圖 11-7)



圖 11-7 非同步函式回傳的型別為 Promise 物件的泛用型別形式

也就是說，使用非同步函式的感覺有點像是在呼叫一個產生 `Promise` 物件的函式，因此可以接上 `then` (或者是 `catch`) 方法。

```

asyncMessage()
.then(function (value) {
    console.log(value);
});

// => Async: Hello World!

```

因此，`asyncMessage` 的宣告等效於以下普通函式的寫法，只是回傳的東西包覆一層 `Promise` 物件，但是條目 11.2.4 有講過，由於沒有註記時會被認定為 `Promise<unknown>` 型別，因此回傳的 `Promise` 物件必須註記型別才能完全等效。

⁷ 函式型別，請參見本書條目 3-4。

```
/* 非同步函式 asyncMessage 等效的宣告手法 */
function asyncMessage() {
    return new Promise<string>(function (resolve, reject) {
        resolve('Async: Hello World!');
    });
}
```

這時，可能某些邪惡的讀者會想說，以下對非同步函式註記的寫法豈不就讓 TypeScript 的型別系統壞掉了嗎？

```
/* 故意將非同步函式的回傳型別寫為非 Promise 物件的型別 */
async function asyncMessage(): string {
    return 'Async: Hello World!';
}
```

TypeScript 當然知道你是故意的。(如圖 11-8)

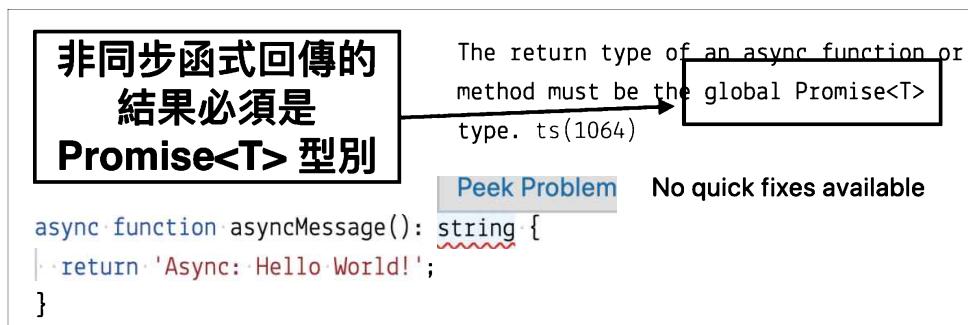


圖 11-8 非同步函式絕對會回傳 Promise 物件相關的型別

11.3.2 自由操作 Promise 的函式

如果單純只是將回傳結果包裝成 Promise，那麼這樣的語法糖實在太牽強。

而非同步函式的另一個關鍵字 `await` 可以負責將 Promise 物件 `resolve` 出來的結果拔出來，因此條目 11.2.2 曾出現的範例程式碼：

```
resolveAfter(2000)
    .then(function (value) {
        console.log(value);
        return resolveAfter(3000);
    })
    .then(function () {
```

```
    console.log(value);
    return resolveAfter(1000);
})
.then(function (value) {
    console.log(value);
});
```

可以改寫為：

```
async function example() {
    const result1 = await resolveAfter(2000);
    console.log(result1);

    const result2 = await resolveAfter(3000);
    console.log(result2);

    const result3 = await resolveAfter(1000);
    return result3;
}

example().then(function (value) {
    console.log(value);
});
```

以上的寫法看起來是不是比起 Promise 串接鏈的寫法，在可讀性上是不是更加清楚呢？

要注意的一件事情是，`await` 關鍵字接的東西是 Promise 相關的物件，畢竟非同步函式裡，通常需要被等待的東西是另一個非同步程式；因此，如果 `await` 接的東西並非 Promise 相關物件時，雖然說不會出現警告訊息，但他還是會提醒你這樣做是沒太多意義的。(如圖 11-9)

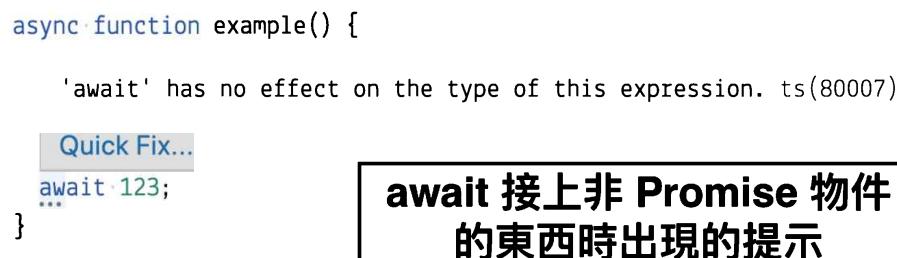


圖 11-9 `await` 接的東西不是 Promise 物件時，基本上沒太大意義

11.3.3 輕鬆進行 Promise 例外處理的函式

另外，還記得探討非同步的程式設計過程中，它們最容易遇到的問題就是執行順序會跳來跳去，造成程式的可讀性大打折扣。

而條目 11.2.2 介紹的 Promise 串接鍊基本上除了攤平化回呼函式地獄的寫法外，串接下去的順序也是固定的，不會說跳來跳去執行，所以解決大部分的問題。

然而，如果我們想要幫 Promise 進行例外處理（Exception Handling）相關的事物，會非常麻煩，請看下面的範例。

```
/* 回傳一半機率會 Resolve 或 Reject 的 Promise 物件 */
function possiblyReject(message: string) {
    return new Promise(function (resolve, reject) {
        if (Math.random() > 0.5) {
            resolve(`Resolved: ${message}`);
        } else {
            reject(`Rejected: ${message}`);
        }
    });
}

possiblyReject('Promise 1')
    .then(function () {
        return possiblyReject('Promise 2');
    })
    .catch(function (error) {
        /* 處理錯誤必須要集體處理 */
        if (error === 'Rejected: Promise 1') {
            // Promise 1 失敗時的處理
        } else {
            // Promise 2 失敗時的處理
        }
    });
}
```

以上的程式碼，你會發現 Promise 串接鍊有個相對來說比較不好的地方，例外處理的部分會根據串接不同的 Promise 物件會需要進行分流的狀態。

這樣寫真的會很痛苦，因此我們可以將以上的範例程式碼，運用例外處理的 `try...catch...` 語法，將其改寫成以下的等效寫法：

```
async function example() {
  try {
    await possiblyReject('Promise 1');
  } catch (error) {
    console.log(`#${error}`);
    return;
  }

  try {
    await possiblyReject('Promise 2');
  } catch (error) {
    console.log(`#${error}`);
    return;
  }
}
```

以上的程式碼除了可讀性高以外，寫起來的樣子跟同步的程式碼很像；儘管它描述的是非同步的行為，但是它會按照順序執行，在 `await` 關鍵字部分等待非同步程式的執行結果。

► 本章練習

1. 試描述 JavaScript 裡同步（Synchronous）與非同步（Asynchronous）概念與差異性。
2. 試問以下的程式碼會以什麼樣的順序印出值？

```
console.log('Log 1');

// 設定 3 秒計時後印出值
setTimeout(function () {
  console.log('Log 2');
}, 3000);
```

```
console.log('Log 3');

// 設定 0 秒計時後印出值
setTimeout(function () {
    console.log('Log 4');
}, 0);

console.log('Log 5');
```

3. Promise 物件初始化的過程中，會是如何執行程式碼？試分析下面的範例程式碼。

```
new Promise<string>(function (resolve, reject) {
    console.log('Log 1');
    resolve('Hello world!');

    console.log('Log 2');           // ← Resolve 過後，這東西會印出來嗎？
    resolve('Hello world again!'); // ← 可以再度 Resolve 嗎？

}).then(function (result) {
    console.log('Log 3');
    console.log(result);
});

console.log('Log 4');           // ← 是 Promise 裡面的函式，還是這個先執行？
```

4. 在討論函式型別篇章（條目 3.4）時，作者強調過通常函式在宣告時，參數必須要積極註記；然而宣告一個新的 Promise 物件時，為何本章節大部分的範例程式碼，Promise 物的 then 方法裡的參數不太有註記的需要？
5. 假設今天宣告一個 timeoutAfter 函式如下：

```
function timeoutAfter(milliseconds: number) {
    return new Promise<string>(function (resolve, reject) {
        setTimeout(function () {
            reject('Timeout Error');
        }, milliseconds);
    });
}
```

```
});  
}
```

Timeout 通常代表著一段程式執行時間過長，必須要設一段時間進行中斷程式的動作。

運用 timeoutAfter 函式，搭配哪一種 Promise API 就可以達到當一個非同步程式運作時間過長，可以提前終止並拋出錯誤訊息的動作。

6. 非同步函式可以在內部 await 另一個非同步函式嗎？那麼也可以 await Promise API 之類的東西嗎？（譬如：await Promise.all(...) 等東西）
7. 【進階題】條目 11.2.4 討論 Promise 物件的泛型特性時，提到註記 Promise 物件的型別參數 T，就代表 resolve 函式也必須要填入型別 T 的值，但也可以填入 **PromiseLike<T>** 這種型別的值。

TypeScript 事實上具有 **<Type>-Like** 相關的型別（例如本書沒提到過的：**ArrayLike<T>**），讀者可以想想看這類型型別的意義在哪嗎？

12

TypeScript 裝飾子

裝飾子（Decorator）這個東西是物件導向裡、設計模式¹中的一種包裹類別或類別成員的輕量化程式碼，通常也是為了使用類別繼承（Inheritance）而產生直接耦合的情況。不過 TypeScript 提供的裝飾子比較偏向於語法糖（Syntax Sugar）²的寫法，而非物件導向相關的語法（諸如類別、介面等語法）。

另外，裝飾子事實上在 ECMAScript 標準的規劃中還在籌備階段³，因此 TypeScript 的裝飾子在官方被稱作為——比較偏向於“實驗性”特色（Experimental Feature）的語法；然而知名的框架，如 Angular 以及一些常見的 NodeJS 後端框架，如 ExpressJS 也都有使用到此裝飾子的語法糖。

裝飾子運作過程感覺起來很神奇，不過讀者學過之後就會發現，裝飾子僅僅只是一個超級簡單的函式，然而卻可以寫出讓人意料之外的應用。

1 物件導向進階篇章，參見本書第 9 章。

2 語法糖主要專注於簡化程式碼、增進可讀性，而非語言額外的特性；畢竟語法糖可以根據語言本身特性替換成原本的寫法，只是原本的寫法可能會比較複雜些。

3 作者正在寫作本書時，ECMAScript Decorators 還在 Stage-2，詳請可以參見 <https://github.com/tc39/proposal-decorators>。

► 12.1 裝飾子的簡介 Introduction to Decorators

12.1.1 啟用實驗性設定

由於裝飾子本身是實驗性的語法特色，因此在正式進到裝飾子語法的討論之前，請讀者記得在專案裡的 TypeScript 設定檔（也就是 tsconfig.json）啟用兩個選項，分別是：`experimentalDecorators` 以及 `emitDecoratorMetadata`。

```
/* tsconfig.json */
{
    // 其他設定略 ...

    /* Experimental Options */
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true

    // 其他設定略 ...
}
```

12.1.2 Hello! Decorator!

裝飾子的語法到底長什麼樣子呢？以下就用非常簡單的範例：

```
class Cat {
    public name: string;
    public breed: string;

    constructor(name: string, breed: string) {
        this.name = name;
        this.breed = breed;
    }

    get info() {
        return `${this.name} is a ${this.breed}`;
    }

    // 放上裝飾子 decorate
}
```

```
@decorate
public makeNoise() {
    console.log('Meow meow meow~~~');
}
}

// 裝飾子函式的宣告
function decorate(target: any, key: string, desc: PropertyDescriptor) {
    console.log('Hello world!');

    // 印出裝飾子的參數
    console.log(target);
    console.log(key);
    console.log(desc);
}
```

以上宣告一個貓咪類別 `Cat` 以及一個普通函式 `decorate`，而類別 `Cat` 的成員方法 `makeNoise` 被安裝了 `decorate` 裝飾子；如果你將其編譯並且執行時，就會出現以下的訊息：

```
Hello world!
Cat { info: [Getter], makeNoise: [Function] }
makeNoise
{
    value: [Function],
    writable: true,
    enumerable: true,
    configurable: true
}
```

相信第一行出現的 `Hello world!` 訊息應該很清楚，單純就是裝飾子函式 `decorate` 被呼叫時第一次 `console.log` 出現的訊息。

第二行出現的 `Cat { info: [Getter] ... }` 則是類別 `Cat` 的原型（也就是 `Prototype`，此時的 `target` 參數存有 `Cat.prototype` 這個物件），而 JavaScript 裡的原型敘述的內容為該類別的成員；也就是說，裝飾子函式裡的第一個參數為被裝飾的程式碼所屬類別的原型。

第三行出現的 `makeNoise` 則是被裝飾的成員名稱，所以裝飾子函式裡第二個參數為被裝飾到的東西（可為類別成員的方法、屬性，或類別本身甚至於方法裡的參數等）的名稱。

最後出現的是一大堆 JSON 物件的資料，包含 `value`、`writable`、`enumerable` 以及 `configurable` 四種屬性，此物件又被稱作為屬性描述器（Property Descriptor），而 TypeScript 有內建其型別為 `PropertyDescriptor`；也就是說，最後一個參數為被裝飾到的成員對應之屬性描述器。

至於屬性描述器到底是什麼呢？本書的條目 12.3 會仔細剖析並講述應用。

另外，讀者可能覺得很奇怪，為何裝飾子要有這三個參數？一定得有這三個參數嗎？事實上，除了省掉最後一個參數外，前面的兩個參數必須存在，否則裝飾子會被 TypeScript 發出警告。（如圖 12-1）

The screenshot shows a code editor with the following TypeScript code:

```
    }
    function decorate(): void
    'decorate' accepts too few arguments to be used as a decorator here.
    Did you mean to call it first and write '@decorate()'? ts(1329)
    Peek Problem Quick Fix
    @decorate
    public makeNoise(a: any)
        console.log('Meow~meo');
    };
}

function decorate() {
    console.log('Hello~world!');
}
```

A tooltip box is overlaid on the code, containing the following text:

裝飾子函式若沒有參數時，儘管 TypeScript 會告訴你該函式擁有不夠的參數，然而沒有參數的裝飾子也是沒啥意義

圖 12-1 裝飾子函式若沒有足夠的參數時，TypeScript 會自動發出警訊

不過裝飾子在使用時最關鍵的東西是前兩個參數提供的資訊；沒有使用到前兩個參數的裝飾子基本上沒什麼用處，不如不用該裝飾子也沒差。

12.1.3 裝飾子等效語法特性

裝飾子在使用時，只會被呼叫一次——而唯一的那一次就是在宣告類別的時候呼叫。其實這個邏輯很簡單，前面的範例中，類別 `Cat` 的宣告就已經展示過此特性。

```
class Cat {  
    // 略 ...  
  
    // 放上裝飾子 decorate  
    @decorate  
    public makeNoise() {  
        console.log('Meow meow meow~~~');  
    };  
}  
  
// 裝飾子函式的宣告  
function decorate(target: any, key: string, desc: PropertyDescriptor) {  
    // 略 ...  
}
```

就算我們沒有建立任何類別 `Cat` 的實體（Instance），該類別被宣告並且成員方法 `makeNoise` 被裝飾 `decorate` 的那一剎那就會執行 `decorate` 函式的內容。

這應該很好理解。不過如果讀者真的使用 `tsc` 進行編譯，並且檢視結果時，大致上的結構如下：

```
var __decorate = (this && this.__decorate) ||  
function (decorators, target, key, desc) {  
    // 裡面內容超複雜，因此略 ...  
    return c > 3 && r && Object.defineProperty(target, key, r);  
};  
  
var Cat = /** @class */ (function () {  
    // 略 ...  
  
    Cat.prototype.makeNoise = function (a) {  
        console.log('Meow meow meow~~~');  
    };
```

```
;  
_decorate([  
    decorate  
], Cat.prototype, "makeNoise");  
return Cat;  
}());  
  
function decorate(target, key, desc) {  
    // 略 ...  
}
```

首先讀者應該可以看到，TypeScript 在實踐裝飾子的特性時，使用 `_decorate` 函式將類別 `Cat` 以及成員名稱 `makeNoise` 套入名為 `decorate` 的裝飾子：

```
// 略 ...  
  
_decorate([  
    decorate  
], Cat.prototype, "makeNoise");  
  
// 略 ...
```

這裡肯定會有一個疑問，到底 `_decorate` 函式在做什麼？如果將編譯過後的結果，拔掉一些判斷敘述等過程，簡化之等效結果如下：

```
function _decorate(decorators, target, key) {  
    // 先取得 key 之屬性描述器  
    let desc = Object.getOwnPropertyDescriptor(target, key);  
  
    // 將每個裝飾子函式，套入原型、成員名稱以及對應之屬性描述器  
    for (let i = 0; i < decorators.length; i += 1) {  
        const decorator = decorators[i];  
  
        decorator(target, key, desc);  
    }  
}
```

當然，以上是極為簡化、較好理解的裝飾子運作過程；讀者應該可以發現，除了將屬性描述器取出外，裝飾子語法僅僅就是將原型、被裝飾的東西名稱和對

應之屬性描述器丟進裝飾子函式罷了。

另外，從以上的簡化程式碼可以得知，同一個成員目標可以附上多個裝飾子，因為它是用 For 迴圈對多個裝飾子進行迭代；也就是說，以下的寫法是可以的：

```
class Cat {  
    // 略 ...  
  
    // 放上多個裝飾子 decorate1  
    @decorate1  
    @decorate2  
    public makeNoise() {  
        console.log('Meow meow meow~~~');  
    };  
}  
  
// 裝飾子函式的宣告  
function decorate1(target: any, key: string, desc: PropertyDescriptor) {  
    console.log('From decorator 1');  
}  
  
function decorate2(target: any, key: string, desc: PropertyDescriptor) {  
    console.log('From decorator 2');  
}
```

是可以接受的寫法。但如果讀者認為是 'From decorator 1' 這個訊息先印出來的話，那麼你就錯了！

以上的範例運作結果為：

```
From decorator 2  
From decorator 1
```

“越靠近”被裝飾的目標的裝飾子函式會先被觸發；也就是說，由於 `decorate2` 裝飾的位置靠近 `makeNoise` 成員，因此它最先觸發。

事實上，要理解裝飾子，除了其基本的等效語法過程外，裝飾子被觸發的順序也會是本章的重點之一。

► 12.2 裝飾子種類

裝飾子的種類其實不單純只是 12.1 講述的這種形式，它可能會以各種樣貌出現，只是 12.1 是比較常見的裝飾子函式用法。

12.2.1 成員方法裝飾子

基本上，成員方法裝飾子（Member Method Decorator）早就在條目 12.1.2 探討過了，其裝飾子函式案例如下：

```
class SomeClass {  
    // 其他成員略 ...  
  
    /* 使用類別成員方法裝飾子 */  
    @decorate  
    method() { /* 略 ... */ }  
}  
  
/* 成員方法裝飾子 */  
function decorate(target: any, key: string, desc: PropertyDescriptor) {  
    console.log(target); // ← 所屬類別的原型 Prototype  
    console.log(key); // ← 成員方法名稱  
    console.log(desc); // ← 成員方法之屬性描述器  
}
```

基本上會有三個參數，假設裝飾在 `SomeClass` 這個類別的成員方法時，`decorate` 函式之第一個參數是 `SomeClass` 的原型物件（也就是 `SomeClass.prototype`）、第二個 `key` 則是被裝飾的方法名稱 '`method`'（型別為字串），最後一個則是該方法的屬性描述器。

讀者可能會覺得第一個參數對應的型別為 `any`⁴ 可能感到疑惑，為何不要使用 `SomeClass` 或者其他特定的型別像是 `unknown` 呢？

4 Any 與 Unknown 型別，請參見本書條目 4.6。

原因其實很簡單，因為裝飾子通常不會知道到底會被裝飾在哪一個類別上；二來是如果真的限定在哪一個類別時，則該裝飾子可以被使用的範圍就大大地下降了；最後則是我們很難註記 `target` 之型別為類別之原型（Prototype）物件型別。

再強調一次：`target` 參數代表的是原型物件，而非類別對應的實體型別；前者所指的原型物件為 `SomeClass.prototype`，後者則為 `new SomeClass(...)`。

12.2.2 存取方法裝飾子

事實上，類別的成員方法外，存取方法（Accessor Method）⁵ 也可以使用裝飾子，以下延用條目 12.1.2 的 `Cat` 類別進行示範。

```
class Cat {  
    // 略 ...  
  
    /* 存取方法可以放上裝飾子 */  
    @decorate  
    get info() {  
        return `${this.name} is a ${this.breed}`;  
    }  
  
    // 略 ...  
}  
  
// 裝飾子函式的宣告  
function decorate(target: any, key: string, desc: PropertyDescriptor) {  
    // 印出裝飾子的參數  
    console.log(target);  
    console.log(key);  
    console.log(desc);  
}
```

⁵ 類別存取方法，請參見本書條目 5.2.3。

印出的結果如下：

```
Cat { info: [Getter], makeNoise: [Function] }
info
{
  get: [Function: get],
  set: undefined,
  enumerable: true,
  configurable: true
}
```

前兩行分別印出的是 `Cat` 類別的原型物件 `Cat.prototype` 以及裝飾到的取值方法（Getter Method）名稱 `info`；不過你會發現，在屬性描述器的部分，有別於條目 12.1.2 裝飾的成員方法 `makeNoise` 所出現的結果，也就是：

```
{
  value: [Function],
  writable: true,
  enumerable: true,
  configurable: true
}
```

裝飾到取值方法 `info` 則是出現了以下的結果：

```
{
  get: [Function: get], // ← 取值方法
  set: undefined,      // ← 由於沒有宣告存值方法，因此此為 undefined
  enumerable: true,
  configurable: true
}
```

也就是說，裝飾到普通成員方法跟存取方法時，屬性描述器的值會有些微的差別；至於屬性描述器的細節，這部分會在後面的章節談論到。

不過基本上，讀者應該可以猜到——該屬性描述器提供的訊息，大意是說 `info` 只能夠被取值（也就是只有取值方法），並沒有方法是可以進行存值的。

12.2.3 成員變數裝飾子

不曉得讀者有沒有注意到，前面的條目中，裝飾子函式的第一個參數 `target` 指涉到的類別原型物件裡，似乎只有顯示成員方法但沒有成員變數？

以條目 12.1.2 裡的 `Cat` 類別為例，裝飾在 `makeNoise` 方法時，第一個參數 `target` 印出來的結果為：

```
Cat { info: [Getter], makeNoise: [Function] }
```

從以上的結果發現，除了成員方法與存取方法外，成員變數的資訊並沒有出現在上面；基本上，成員變數綁定的時機點為物件被初始化（也就是建構物件實體時）時，成員變數的值才會出來，畢竟有時候初始化值是在建構子函式（Constructor Function）裡進行初始化的。

```
class Cat {  
    public name: string;  
    public breed: string;  
  
    /* 成員變數除了宣告類別成員時可以指派外，大多數情形會在 constructor 函式裡  
    初始化 */  
    constructor(name: string, breed: string) {  
        this.name = name;  
        this.breed = breed;  
    }  
  
    // 略 ...  
}
```

這對裝飾子函式的寫法會有什麼需要注意的地方呢？

如果你將條目 12.2.1 或 12.2.2 的裝飾子函式直接安裝到成員變數時，一定會出現錯誤。(如圖 12-2)

```
class Cat {  
    /* 裝飾到成員變數的宣告上 */  
    @decorate  
    public name: string;
```

```
// 略 ...
}

// 裝飾子函式的宣告
function decorate(target: any, key: string, desc: PropertyDescriptor) {
    // 略 ...
}

class Cat {
    @decorate
    function decorate(target: any, key: string, desc: PropertyDescriptor): void
        Unable to resolve signature of property decorator when called as an
        expression. ts(1240)
    > + Peek Problem No quick fixes available
    > + get info() { ...
    }
    > + public makeNoise(a: any) { ...

```

成員方法裝飾子的格式
不能裝飾在成員變數上

圖 12-2 成員方法裝飾子沒辦法安裝到成員變數上

理由非常簡單，屬性描述器裡具有一個屬性為 `value`，指涉的東西是該成員對應的值；以下面的案例為例：

```
class Cat {
    // 略 ...

    // 放上裝飾子 decorate
    @decorate
    public makeNoise() {
        console.log('Meow meow meow~~~');
    };
}

// 裝飾子函式的宣告
function decorate(target: any, key: string, desc: PropertyDescriptor) {
    // 略 ...
}
```

`makeNoise` 的屬性描述結果為：

```
{  
  value: [Function],    // ← makeNoise 的屬性描述器有一個 value 屬性  
  writable: true,  
  enumerable: true,  
  configurable: true  
}
```

該 `value` 屬性對應的東西就是 `Cat.prototype.makeNoise` 這個函式。

如果讀者細心的話，應該留意到前面有講過：「成員變數的值會在實體被建構過後才會初始化初值」；另外，根據條目 12.1 討論的結果，裝飾子函式會在類別宣告後執行一次。

裝飾子函式會在類別剛宣告時執行，但成員變數的值要等到實體被建構時才會確定，這時候裝飾子函式裡面的內容早就被執行過了；這也意味著，想要知道成員變數的屬性描述狀態是根本不可能的事情。

於是我們可以得出很簡單的結論：成員變數的裝飾子事實上就是省略掉成員方法修飾子的最後一個參數！

```
class Cat {  
  /* 裝飾到成員變數的宣告上 */  
  @memberVariableDecorate  
  public name: string;  
  
  // 略 ...  
}  
  
// 成員變數的裝飾子函式的宣告，只有兩個參數  
function memberVariableDecorate(target: any, key: string) {  
  // 印出裝飾子的參數  
  console.log(target);  
  console.log(key);  
}
```

至於印出來的結果會是什麼呢？相信讀者應該可以猜得出來：

```
Cat { info: [Getter], makeNoise: [Function] }  
name
```

12.2.4 靜態成員裝飾子

靜態成員部分分成靜態方法、靜態存取方法與靜態屬性，跟普通成員的差別就只差在靜態成員是在類別被宣告時就出現的成員，綁定在類別本身，而非由類別建構的實體上。⁶

靜態成員當然也可以被裝飾，其中，靜態方法與靜態存取方法分別與條目 12.2.1 與 12.2.2 的裝飾子函式的宣告方式一模一樣；靜態屬性則是跟 12.2.3 講到的裝飾子函式的宣告方式相同。

```
class Circle {  
    /* 裝飾到靜態屬性的宣告上 */  
    @decorateStaticProperty  
    public static PI: number = Math.PI;  
  
    /* 裝飾到靜態方法的宣告上 */  
    @decorateStaticMethod  
    public static area(radius: number) {  
        return Circle.PI * (radius ** 2)  
    }  
}  
  
// 靜態屬性裝飾子  
function decorateStaticProperty(target: any, key: string) {  
    // 印出裝飾子的參數  
    console.log(target);  
    console.log(key);  
}
```

6 靜態成員（Static Member），請參見本書條目 5.2.4。

```
// 靜態方法以及靜態存取方法裝飾子
function decorateStaticMethod(
    target: any,
    key: string,
    desc: PropertyDescriptor
) {
    // 印出裝飾子的參數
    console.log(target);
    console.log(key);
    console.log(desc);
}
```

以上的範例印出的結果如下：

```
[Function: Circle] { area: [Function], PI: 3.141592653589793 }
PI
[Function: Circle] { area: [Function], PI: 3.141592653589793 }
area
{
    value: [Function],
    writable: true,
    enumerable: true,
    configurable: true
}
```

你會發現靜態成員的裝飾子，`target` 指涉的東西變成了類別建構子函式（Constructor Function）本身；靜態成員與普通成員裝飾子中，第一個參數差異在於，前者是類別建構子函式，後者則是該被別的原型物件（Prototype）。

12.2.5 參數裝飾子

類別裡方法所宣告的參數（Parameter）也可以被裝飾，所以又被稱作參數裝飾子（Parameter Decorator）。

然而，參數裝飾子並沒有所謂的屬性描述器，取而代之的是該參數所在的位置。譬如以下的範例，我們將 `Cat` 類別中的 `makeNoise` 方法改寫，並且在其參數宣告旁加上一個裝飾子 `decorateParameter`：

```
class Cat {  
    // 略 ...  
  
    public makeNoise(  
        @decorateParameter noise: string = 'Meow meow meow~~~'  
    ) {  
        console.log(noise);  
    };  
}  
  
// 參數裝飾子函式的宣告，第三個參數 index 為被裝飾之參數位置  
function decorateParameter(target: any, key: string, index: number) {  
    // 印出裝飾子的參數  
    console.log(target);  
    console.log(key);  
    console.log(index);  
}
```

你應該會發現參數裝飾子的寫法有點冗長，因為真的是裝飾在參數旁，如果再搭配參數型別的註記以及預設值，理所當然地會有些複雜。

而以上的程式碼運作結果，參數 `target` 應該可以猜得出來是什麼，畢竟從 12.2.1 講到 12.2.5 (也就是本條目)，其代表的值就是 `Cat` 類別的原型物件。

不過呢，第二個參數 `key` 可能會誤以為就是指參數的名稱，所以會是 '`noise`' 這個字串值，但實際上參數裝飾子函式之第二個參數的值是該參數所宣告在的方法名稱，所以結果是 '`makeNoise`' 喔！

因此，同一個方法裡的多個參數都有被裝上參數裝飾子時，該裝飾子函式的第二個參數之值永遠不會變，也就是那些參數被宣告在的函式名稱！

所以為了區隔掉參數的種類，就出現了第三個參數 `index`，也就是代表參數的位置，跟陣列的索引邏輯一樣，參數位置由數字 0 開始計數。因此本條目裡的範例會印出以下的結果：

```
Cat { info: [Getter], makeNoise: [Function] }  
makeNoise // ← 注意：並非參數名稱 noise，而是參數所在的方法名稱 makeNoise  
0          // ← noise 為 makeNoise 方法裡的第一個參數，所以 0
```

12.2.6 類別裝飾子

讀者肯定覺得裝飾子種類怎麼那麼多！？

作者認為裝飾子難在它可以裝飾在類別裡的任何地方（甚至連類別本身，也就是本條目所要講的東西），若沒搞清楚類別裡宣告的東西的特性，裝飾子的宣告與使用會變成一場惡夢；起初你可能以為可以裝飾在類別成員方法的裝飾子，在成員變數上卻行不通，因為裝飾子的格式限制所導致的錯誤。

回歸正題，類別裝飾子就是裝飾在類別宣告的裝飾子；既然是裝飾在類別的宣告上，裝飾子就只會（而且絕對）需要一個參數，代表的是該類別的建構子函式，以下舉類別 `Cat` 為範例：

```
@decorateClass
class Cat { /* 略 ... */ }

// 類別裝飾子的宣告
function decorateClass(constructor: Function) {
    // 印出類別裝飾子的參數
    console.log(constructor);
}
```

類別裝飾子的函式宣告簡單許多，畢竟跟類別成員不一樣的地方除了本身沒有屬性描述器外，想要知道被裝飾的類別名稱，可以使用 `constructor.name` 來代表；因此，你可能想說之前講過的裝飾子都需要 `key` 這個參數，代表被裝飾的東西的名稱，而這次不需要宣告的理由就是因為 `constructor`（也就是類別裝飾子的第一個參數）所提供的資訊已足夠。

► 12.3 裝飾子的運用

12.3.1 屬性描述器的介紹 Property Descriptor

首先根據前面的條目討論結果，裝飾子函式中，屬性描述器只會出現在裝飾於成員方法、存取方法（以及靜態方法與靜態存取方法）中⁷；畢竟在本章節開頭有表明，裝飾子函式的目的是避免程式碼直接耦合的情形，而改採用裝飾子包裝的方式處理。

通常需要被裝飾子包裝、避免直接耦合的程式碼，自然而然就是指函式、方法類型的東西，而不會是純屬性類型的東西。（屬性就是一個值而已，但函式、方法等可以經由組合的方式產生不同結果）

因此只有成員方法等類型的裝飾子函式才會提供屬性描述器這個物件；至於屬性描述器到底是什麼呢？屬性描述器長相有兩種版本，而且這兩種版本是互不相容的，分別是資料描述器（Data Descriptor）以及存取描述器（Accessor Descriptor）。

想要取得物件的屬性描述器，可以使用 `Object.getOwnPropertyDescriptor` 這個方法（這名稱很長，不過一直以來 JavaScript 裡提供的 API 在取名上好像都是這樣）；以下面的 JSON 物件為例，假設我們想要取得 `maxwell` 物件的 `name` 屬性的屬性描述器，我們可以這麼做：

```
let maxwell = {
  name: 'Maxwell',
  age: 18,
  interests: ['Programming', 'Drawing']
};

/* 取得 maxwell.name 之屬性描述器 */
```

7 成員方法裝飾子，請參見本書條目 12.2.1；存取方法裝飾子，請參見本書條目 12.2.2；靜態成員裝飾子，請參見本書條目 12.2.4。

```
console.log(Object.getOwnPropertyDescriptor(maxwell, 'name'));
```

以上的 `console.log` 印出來的結果為：

```
{
  value: 'Maxwell',
  writable: true,
  enumerable: true,
  configurable: true
}
```

粗體標示部分為資料描述器的特徵，以本程式碼案例就是在敘述 `maxwell.name` 這個屬性的資料敘述。

資料描述器有四大屬性——其中 `value` 是為該物件屬性對應的值，也就是說，`maxwell.name` 的值為字串 '`Maxwell`'；而 `writable` 為該物件之屬性是否為可被覆寫的狀態，如果 `writable` 為 `false` 時，則 `maxwell.name` 就是不可覆寫狀態，俗稱唯讀狀態（Read-Only）。

資料描述器裡的 `enumerable` 則代表該屬性（也就是 `maxwell.name` 裡的 '`name`' 屬性）是否可以在 `for...in...` 迴圈被迭代，意思是說：

```
let maxwell = {
  name: 'Maxwell',
  age: 18,
  interests: ['Programming', 'Drawing']
};

for (let prop in maxwell) {
  console.log(prop);
}
// => 'name'
// => 'age'
// => 'interests'
```

以上的範例程式碼使用 `for...in...` 迴圈，可以迭代出 `maxwell` 這個物件裡所有的屬性；相對地，如果 `enumerable` 為 `false`，`for...in...` 迴圈就不會出現該屬性。

最後，`configurable` 則是指能不能夠修改該屬性的描述器資訊，亦或者是刪掉該屬性。

假設我們想要讓 `maxwell.age` 修改成唯讀狀態、不能夠被迭代，並且不能夠被修改，我們可以使用 `Object.defineProperty`，第一個參數填上 `maxwell` 物件、第二個則是目標屬性 `age`，最後則是屬性的資料描述器：

```
let maxwell = {
  name: 'Maxwell',
  age: 18,
  interests: ['Programming', 'Drawing']
};
```

```
Object.defineProperty(maxwell, 'age', {
  value: 18,
  writable: false,
  enumerable: false,
  configurable: false,
});
```

這麼一來，想要覆寫掉 `maxwell.age` 就會出現大串錯誤訊息：

```
maxwell.age = 123;
^
TypeError: Cannot assign to read only property 'age' of object '#<Object>'
at ... 錯誤訊息略 ...
```

想要使用 `for...in...` 迴圈檢視物件時，`maxwell.age` 就會被跳過：

```
for (let prop in maxwell) {
  console.log(prop);
}
// => 'name'
// => 'interests'
```

最後，想要重新宣告 `maxwell.age` 的資料描述性質時，也會出現錯誤訊息：

```
// ...
Object.defineProperty(maxwell, 'age', {
```

```
// ... 略
configurable: false, // ← 設定為不能夠重新設定屬性描述
});

/* 重新設定 maxwell.age 屬性描述性質 */
Object.defineProperty(maxwell, 'age', {
    value: 18,
    writable: true,
    enumerable: false,
    configurable: true,
});
// 以下為錯誤訊息：
Object.defineProperty(maxwell, 'age', {
    ^
TypeError: Cannot redefine property: age
    at ...
```

以上為資料描述器的簡單介紹；另一種屬性描述器的形式為存取描述器，以下面 JSON 物件的例子為範例：

```
let circle = {
    _radius: 0,
    area: 0,

    /* 告訴 radius 為存取方法 */
    get radius() { return this._radius; },
    set radius(value: number) {
        this._radius = value;
        this.area = Math.PI * (value ** 2);
    },
};
```

以上的範例可以發現，`circle.radius` 為存取方法，亦可以被視為動態的物件屬性，與類別存取方法的宣告方式近似，而使用起來的效果也很接近：

```
/* 初始值 */
console.log(circle.radius); // => 0
console.log(circle.area); // => 0
```

```
/* 動態設定 radius 屬性 */
circle.radius = 5;
console.log(circle.radius);    // => 5
console.log(circle.area);     // => 78.53981633974483
```

如果我們用 `Object.getOwnPropertyDescriptor` 檢視 `circle.radius` 屬性時，會發現存取描述器的結構如下：

```
{
  get: [Function: get radius],
  set: [Function: set radius],
  enumerable: true,
  configurable: true
}
```

粗體的部分是存取描述器有別於資料描述器的特徵，存取描述器會描述存取方法（也就是 `get` 以及 `set` 屬性）以及該屬性是否可以被 `for...in...` 迴圈迭代（`enumerable`）以及是否可以修改該存取描述器的性質（`configurable`）。

其他有關於更多跟屬性描述器有關的東西會在本章練習的習題中深入討論，讓讀者自行試試看唷～

12.3.2 善用屬性描述器

其實光是從前一個條目講到屬性描述器的特點就可以知道，我們可以修改成員方法（以及存取方法等）的性質，如將該成員方法設定成唯讀狀態；延用條目 12.1.2 的 `Cat` 類別，以裡面宣告的成員方法 `makeNoise` 為例：

```
class Cat {
  public name: string;
  public breed: string;

  // 略 ...

  @readonly
  public makeNoise() {
    console.log('Meow meow meow~~~');
```

```
};

}

// readonly 裝飾子函式的宣告
function readonly(target: any, key: string, desc: PropertyDescriptor) {
    desc.writable = false;
}
```

如果你建構出 `Cat` 實體時，必且使用 `Object.getOwnPropertyDescriptor` 檢視 `makeNoise` 方法的屬性描述器，就會發現 `writable` 就被竄改了：

```
let cat = new Cat('Julia', 'Scottish Fold');

// 檢視 cat.makeNoise 的屬性描述
console.log(Object.getOwnPropertyDescriptor(cat, 'makeNoise'));
```

因此，如果程式碼擅自想要覆寫掉 `makeNoise` 方法時，就會出現一長串錯誤訊息：

```
let cat = new Cat('Julia', 'Scottish Fold');

cat.makeNoise = function () { /* ... */ };
// c.makeNoise = function () { /* ... */ }
//           ^
// TypeError: Cannot assign to read only property 'makeNoise' of object
//           '#<Cat>'
//     at ...
```

另外，類別在宣告的過程中，我們也可以寫個簡單的裝飾子，專門輔助我們除 Bug，例如：

```
class SomeClass {
    // 略 ...

    @debug
    public someMethod(message: string) {
        throw new Error(message);
    };
}
```

```
// debug 裝飾子函式的宣告
function debug(target: any, key: string, desc: PropertyDescriptor) {
    /* 取得原本的方法 */
    const method = desc.value;

    /* 包裝原本的方法到 try ... catch ... 敘述式 */
    desc.value = function (...params: any[]) {
        try {
            /* 如果呼叫方法時出現錯誤，就會進入到 catch 敘述式那邊 */
            method(...params);
        } catch (err) {
            const className = target.constructor.name;
            console.log(`Error occurred in ${className}.prototype.${key}`);
            console.log(`Error: ${err.message}`);
        }
    }
}
```

以上的範例有點複雜，不過可以發現，該範例就是從屬性描述器中，將原本的方法拔出來，並且用新的函式宣告覆寫掉原本的方法。

而新的函式宣告就是將一個簡單的 `try...catch...` 敘述式包在舊的函式宣告裡，因此如果建構出 `SomeClass` 的實體並且呼叫被 `@debug` 裝飾子包裝過後的 `someMethod` 方法時，就會出現以下的結果：

```
let instance = new SomeClass();

instance.someMethod('Hello world!');
// Error occurred in SomeClass.prototype.someMethod
// Error: Hello world!
```

裝飾子的好處即是——簡簡單單一行裝飾在類別成員上，就可以重複利用這些裝飾子內部的程式碼。

12.3.3 裝飾子工廠函式

裝飾子的另一種用法是搭配工廠方法（Factory Method）模式，也就是所謂的裝飾子工廠（Decorator Factory）。

它的概念很簡單，也可以將其理解為裝飾子函式參數化（Parameterization）的一種手段；以前面的裝飾子 `@readonly` 為例，假設今天想要改成，`@readonly` 可以填入一個布林值參數，作為設定成唯讀模式的依據：

```
class Cat {  
    public name: string;  
    public breed: string;  
  
    // 略 ...  
  
    @readonly(true)      // ← 設定成唯讀模式  
    public makeNoise() {  
        console.log('Meow meow meow~~~');  
    };  
}
```

這種情況下，原本的 `@readonly` 裝飾子函式宣告方式為：

```
function readonly(target: any, key: string, desc: PropertyDescriptor) {  
    desc.writable = false;  
}
```

由於 `@readonly` 想要改成 `@readonly(active: boolean)` 這種情形，勢必不能用以上的寫法；此時我們可以將其改成以下的宣告形式：

```
// 宣告 readonly 裝飾子工廠函式  
function readonly(active: boolean) {  
  
    /* 回傳正式的裝飾子函式 */  
    return function (target: any, key: string, desc: PropertyDescriptor) {  
        desc.writable = active;  
    };  
}
```

其實裝飾子工廠函式——正如其名，就是回傳裝飾子函式的工廠函式，頂多就是增加了可被參數化的性質。

12.3.4 裝飾子執行順序

最後，裝飾子最重要的東西是它的執行順序，以下先從最單純的案例探討：

```
@decorateClass
class Cat {
    @decorateProperty
    public name: string;
    public breed: string;

    constructor(name: string, breed: string) {
        this.name = name;
        this.breed = breed;
    }

    @decorateMethod
    get info() {
        return `${this.name} is a ${this.breed}`;
    }

    @decorateMethod
    public makeNoise() {
        console.log('Meow meow meow~~~');
    };
}

function decorateClass(constructor: Function) {
    const name = constructor.name;
    console.log(`Class Decorator Invoked: ${name}`);
}

function decorateProperty(target: any, key: string) {
    console.log(`Property Decorator Invoked: ${key}`);
}

function decorateMethod(target: any, key: string, desc: PropertyDescriptor) {
    console.log(`Member Method Decorator Invoked: ${key}`);
}
```

以上的程式碼簡單測試常見的裝飾子，其印出結果為：

```
Property Decorator Invoked: name           // ← 屬性裝飾子
Member Method Decorator Invoked: info        // ← 成員方法裝飾子（存取方法 info）
Member Method Decorator Invoked: makeNoise   // ← 成員方法裝飾子（方法 makeNoise）
Class Decorator Invoked: Cat                 // ← 類別裝飾子
```

順序看起來很複雜，不過讀者只要記得一個重點——類別裝飾子裡的程式最後執行，其餘皆按照成員先後宣告順序執行。

也就是說，裡面的成員變數、方法等，先被宣告並且被裝飾的東西會最先執行裝飾子的程式，最後才是類別裝飾子；原因其實不難理解，如果類別裝飾子先執行的話，裝飾子函式裡的參數——也就是建構子函式等於是在類別成員未宣告完畢的狀況下執行，這樣會造成建構子函式形同空殼子，沒有什麼用處。

另外，如果是多個裝飾子的情形，如以下範例所示：

```
class Cat {
  // 略 ...

  @decorateMethod1
  @decorateMethod2
  public makeNoise() {
    console.log('Meow meow meow~~~');
  };
}

function decorateMethod1(target: any, key: string, desc: PropertyDescriptor) {
  console.log(`Member Method Decorator 1 Invoked: ${key}`);
}

function decorateMethod2(target: any, key: string, desc: PropertyDescriptor) {
  console.log(`Member Method Decorator 2 Invoked: ${key}`);
}
```

這時候，裝飾子的執行順序是內層優先於外層；也就是說，儘管 `@decorateMethod1` 先於 `@decorateMethod2` 裝飾在 `makeNoise` 方法上，但是執行順序反而是相反的：

```
Member Method Decorator 2 Invoked: makeNoise  
Member Method Decorator 1 Invoked: makeNoise
```

由於裝飾子的概念具備層層包裝的效果，因此越內層的裝飾子才會優先於外層的裝飾子執行。

以上就是常見的裝飾子語法特性。

A

解答篇

► 第二章 TypeScript 型別系統概論

1. TypeScript 型別系統主要分成型別推論（Inference）與註記（Annotation）的行為：推論行為具有類似動態語言的特色，藉由資料本身的值反推型別；註記則是使用者主動告訴 TypeScript 編譯器，特定的變數代表的型別，屬於類似靜態語言的特色。兩種機制綜合起來的系統，即漸進式型別系統（Gradual Typing）。
2. 註記（Annotation）型別的用意是在告知 TypeScript 編譯器，宣告出來的變數（Variable）或函式裡的參數（Parameter）所代表之型別；斷言（Assertion）則是由於某些原因 TypeScript 編譯器可能無法推論某表達式（Expression）運算結果，所以進行人為告知編譯器，該表達式所代表之型別結果。
3. 由於斷言是人為強制覆蓋掉一個表達式的型別推論結果，因此必須要在很確定程式碼的運作過程時，進行型別斷言。
4. 敘述式（Statement）與表達式（Expression）最關鍵差別在於：後者運算結果會有值出來，前者則沒有。
5. 變數註記方式中，以下深色字為錯誤的部分並且有備註錯誤在哪裡；淺色字則代表沒有問題：

```
/* 題目 5-1 註記 v.s. 斷言 */
let ex1: number = 123;
let ex2 = 123 as number;

/* 型別斷言語法不能放在宣告變數部分 */
let ex3 as number = 123;

/* 型別斷言語法正確，然而數字 number 被斷言為字串 string 本身是不合邏輯的 */
let ex4 = 123 as string;

/* 註記語法不能使用在表達式上，要改就必須改成 123 as number 或 <number>(123) */
let ex5 = 123: number;

/* 純粹錯誤的斷言語法，改成 123 as number 或 <number>(123) */
let ex6 = (123)<number>;

let ex7 = <number>(123);
```

函式註記方式中，以下深色字為錯誤的部分並且有備註錯誤在哪裡；淺色字則代表沒有問題：

```
/* 題目 5-2 函式註記方式 */
let ex1: (x: number, y: string) => string = function(x, y) {
    return x.toString().concat(y);
};

/**
 * 函式型別被註記在變數上時，輸出部分必須用箭頭而非冒號，故應該成：
 * let ex2: (x: ...,) => string = function (x, y) { ...
 */
let ex2: (x: number, y: string) => string = function(x, y) {
    return x.toString().concat(y);
};

/**
 * 此為函式直接作為值，輸出型別直接以冒號接，故須改成：
 * let ex3 = function(x: ...,): string { ...
 */
let ex3 = function(x: string, y: string) => string {
    return x.toString().concat(y);
```

```
};

let ex4 = function(x: string, y: string): string {
    return x.toString().concat(y);
};

function ex5(x: string, y: string): string {
    return x.toString().concat(y);
};
```

6. 原始型別（Primitive Types）為數字（Number）、字串（String）、布林值（Boolean）以及 Undefined、Null；其餘皆為物件型別，如陣列、JSON 物件等。
7. (a) 變數宣告指派式為敘述式，因為回傳結果為 undefined，為敘述式典型特徵。
(b) 純變數指派式會回傳指派的值作為結果，因此為表達式。
(c) 多重指派式的部份，變數 foo 是宣告的同時被指派值（是為敘述式），而被指派的東西為變數 bar 的純指派式子（是為表達式），bar 被指派的值為 1，同時也會回傳數字 1；也就是說，foo 會被指派變數 bar 被指派的值（即該表達式回傳的值），也就是數字 1。
(d) 是，因為純指派式子為表達式，理所當然也可以被型別斷言。

► 第三章 深入型別系統 I 基礎篇

1. 第一段程式碼，whatIsThis 會被推論為 string | number | boolean；第二段程式碼答案跟前一段程式碼一樣。
2. 第一段程式碼，whatIsThis 會被推論為 number；然而，第二段程式碼則是會被推論為 string | number，TypeScript 只對純布林值排除型別可能出現的結果，遇到邏輯表達式則會涵蓋所有可能推論結果。
3. variable 為遲至性指派的宣告，宣告的當下，值為 undefined，所代表型別則是 any。

4. 由於判斷式有機會不會對 variable 指派值，因此會出現錯誤訊息：Variable 'variable' is used before being assigned.
5. 是。
6. 以下是對 JSON 物件型別 A 中的每個屬性註記的型別進行解釋：

```
type A = {
    /* prop1 為必要屬性，必須要有字串型別值 */
    prop1: string;

    /**
     * prop2 為必要屬性，必須要有字串型別值，或者是 undefined，省略掉
     * prop2 是不行的
     */
    prop2: string | undefined;

    /* prop3 為選用屬性，可以省略 prop3，但 prop3 若沒省略時則必須為字串型別值 */
    prop3?: string;

    /* prop4 為選用屬性，跟 prop2 差別在於，prop2 不能省略，但 prop4 可以省略 */
    prop4?: string | undefined;

    /* prop5 為必要屬性，可以為字串型別值或 null，但不能為 undefined */
    prop5: string | null;

    /* prop6 為選用屬性，若有使用時則必須為字串型別值或 null */
    prop6?: string | null;
};
```

7. 選用屬性與唯讀屬性不可能同時存在，假設某個屬性被設定為唯讀狀態時，設置為選用屬性沒有任何意義。
8. 一般函式的宣告下，輸出型別可以從 return 敘述式回傳的值推論出來；相對的，輸入參數之型別則不能被推論的原因則是會受到使用者輸入的值影響，故必須要註記輸入型別。
9. 儘管 what_is_the_inference_result 之函式內部有判斷式壟斷掉輸出部分為數字型別，但是仍然會推斷為 number | string 這種聯集複合的型別，故可

以得出一般函式的輸出型別可以將所有 return 敘述式回傳的值之型別聯集起來：

```
function what_is_the_inference_result(): 42 | "42"
function what_is_the_inference_result() {
  if (true) { return 42; }

  /* 以下這一行絕對不會被執行 */
  return '42';
};
```

第二段程式碼中，函式 `what_is_the_inference_result` 由於多出了回傳布林值 `true` 的 `return` 敘述式，因此推論結果為 `number | string | boolean`。

10. 推論之函式回傳型別就是 `null`（亦或者是 `undefined`），並非 `void`。
11. `number | undefined`，因為判斷式有可能會跳過，因此回傳空值。
12. 以下為參考之簡化結果：

```
function greet(message: string = 'Hello', name?: string) {
  if (name === undefined) {
    console.log(`"${message}"`);
    return;
  }
  console.log(`${message}, ${name}`);
}
```

13. 函式之參數不能同時存在選用參數以及預設值的情形，更精確來說是冗贅的情形，畢竟選用參數若沒有被填入值時，就會自動被預設值取代，程式碼的效果等同於只有預設參數值，因此本題程式碼可以簡化成：

```
function increment(input: number, value: number = 1) {
  return input + value;
}
```

14. 推論結果分別如下：

```
/* 推論結果為：number[][] */
let ex1 = [ [1, 2, 3], [4, 5, 6] ];

/* 推論結果為：(number | number[][][])[] */
let ex2 = [ 1, 2, 3, [4, 5, 6] ];

/* 推論結果為：number[][] */
let ex3 = [ [1, 2, 3], [4, 5, 6], [] ]; // ← 多一個空陣列

/* 推論結果為：never[][] */
let ex4 = [ [], [], [] ]; // ← 此題答案連作者也意想不到

/* 推論結果為：({ foo: number; bar: string })[] */
let ex5 = [
  { foo: 123, bar: 'Hello' },
  { foo: 456, bar: 'World' },
];

/* 推論結果為：(number | undefined)[] */
let ex6 = [1, , 3, , 4]; // ← 疏鬆陣列 Sparse Array

/* 推論結果為：undefined[] */
let ex7 = [ , , , , ]; // ← 完全疏鬆陣列
```

15. 是。

16. 推論結果分別如下：

```
/* 推論結果為：number */
let ex1 = 123;

/* 推論結果為：123 */
const ex2 = 123;

/* 推論結果為：{ foo: number; bar: string } */
let ex3 = { foo: 123, bar: 'Hello' };

/* 推論結果為：{ foo: number; bar: string } */
const ex4 = { foo: 123, bar: 'Hello' };
```

17. 前者部分，儘管變數是宣告常數狀態，但在 JavaScript 裡，JSON 物件的屬性對應值仍然可以修改，因此推論結果為 { hello: string }；後者註記部分除了強調是 { hello: string } 這種 JSON 物件型別結構外，還強調 hello 屬性對應的值只能為 'world'，故後者的物件完全是型態是為鎖死的狀態。
18. 將 Rectangle、Triangle、Circle 型別部分新增 kind 這個屬性，使得 Geometry 變成互斥聯集後，剩下就是實作細節，程式碼參考如下：

```
type Rectangle = {
    kind: 'Rectangle';
    width: number;
    height: number;
};

type Triangle = {
    kind: 'Triangle';
    base: number;
    height: number;
};

type Circle = {
    kind: 'Circle';
    radius: number;
};

type Geometry = Rectangle | Triangle | Circle;

function area(geometry: Geometry): number {
    if (geometry.kind === 'Rectangle') {
        return geometry.width * geometry.height;
    } else if (geometry.kind === 'Triangle') {
        return geometry.base * geometry.height / 2;
    }

    /* 剩下為計算圓形的面積 */
    return Math.PI * geometry.radius * geometry.radius;
}
```

► 第四章 深入型別系統 II 進階篇

1. 陣列（Array）無關乎順序，元素的數量也沒有任何限制；元組的元素型別順序與個數則有特定限制。
2. 以下為各個範例之推論結果：

```
type TripleNumber = [number, number, number];

/* 推論結果為：number[][] */
let ex1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

/* 推論結果為：TripleNumber[] */
let ex2 = [
    [1, 2, 3] as TripleNumber,
    [4, 5, 6] as TripleNumber,
    [7, 8, 9] as TripleNumber
];

/* 推論結果為：number[] */
let ex3 = [
    [1, 2, 3] as TripleNumber,
    [4, 5, 6] as TripleNumber,
    [7, 8, 9] // ← 少一個斷言
];

type QuadrupleNumber = [number, number, number, number];

/* 推論結果為：(TripleNumber | QuadrupleNumber)[] */
let ex4 = [
    [1, 2, 3] as TripleNumber,
    [4, 5, 6] as TripleNumber,
    [7, 8, 9, 10] as QuadrupleNumber
];

/* 推論結果為：number[][][] */
```

```
let ex5 = [
  [1, 2, 3], // ← 少一個斷言
  [4, 5, 6] as TripleNumber,
  [7, 8, 9, 10] as QuadrupleNumber
];
```

3. 除了 ex4 的推論結果為 `(TripleNumber | QuadrupleNumber)[]`，為異質性陣列外，其餘皆為同質性陣列；同質或異質性陣列的判斷基準以推論之陣列的型別種類個數為基準。
4. 以下為各個範例之推論結果：

```
enum Pet { Dog, Cat, Bird, Duck };

/* 推論結果為：Pet */
let ex1 = Pet.Cat;

/* 推論結果為：Pet.Cat，為列舉成員型別 */
const ex2 = Pet.Cat;

/* 推論結果為：string */
let ex3 = Pet[Pet.Cat];

/* 推論結果為：string */
const ex4 = Pet[Pet.Cat];

/* 推論結果為：string */
let ex5 = Pet[2];

/* 推論結果為：string */
const ex6 = Pet[2];
```

5. 以下為各個範例之推論結果：

```
enum Vehicle {
  Car = 'Car',
  Bicycle = 'Bicycle',
  Truck = 'Truck',
  Bus = 'Bus'
};
```

```
/* 推論結果為：Vehicle */
let ex1 = Vehicle.Truck;

/* 推論結果為：Vehicle.Bus，為列舉成員型別 */
const ex2 = Vehicle.Bus;

/** 
 * 推論結果為：Vehicle，原因是 Vehicle.Car 為字串 'Car'，然後變成
 * Vehicle['Car']，於是等效於 Vehicle.Car
 */
let ex3 = Vehicle[Vehicle.Car];

/** 
 * 推論結果為：Vehicle.Bicycle，原因是 Vehicle.Bicycle 為字串 'Bicycle'，
 * 然後變成 Vehicle['Bicycle']，於是等效於 Vehicle.Bicycle
 */
const ex4 = Vehicle[Vehicle.Bicycle];
```

6. 除了 Vehicle.Car 等於字串值 'Car' 以外，其餘的列舉成員值不等於列舉的成員名稱，使得無法用 Vehicle[Vehicle.[成員]] 方式逆向映射。
7. 常數列舉部分會直接將列舉的成員值帶入到任何使用常數列舉的成員部分，省略掉建構列舉對應的 JSON 物件；“大略”的編譯結果如下（隨著版本不同可能會有異動，但大致上不會有太多影響）：

```
var Pet;
(function (Pet) {
    Pet[Pet["Dog"] = 123] = "Dog";
    Pet[Pet["Cat"] = 124] = "Cat";
    Pet["Bird"] = "Bird";
    Pet[Pet["Duck"] = 456] = "Duck";
})(Pet || (Pet = {}));

var foo = Pet.Cat + 100 /* Truck */;
console.log(99 - Pet.Dog);
```

8. 最簡單的方式是：

```
type Dictionary = { [key: string]: <特定型別> }
```

9. 參考程式碼：

```
type example = { a: number; b: string; c: boolean };

/* 取得 example 所有的鍵 'a' | 'b' | 'c' */
type keyofExample = keyof example;

/* 取得 example 所有的值之型別聯集 */
type valueofExample = example[keyofExample];
```

10. 使用 `readonly` 關鍵字宣告 JSON 物件的屬性；使用 `Readonly<T>` 這個泛用型別；使用 `Object.freeze` 方法。

11. 以下為各個範例之推論結果：

```
/* 推論結果與自身一模一樣 */
type Ex1 = number | string | boolean | undefined | null;
type Ex2 = number & string & boolean & undefined & null;
type Ex3 = undefined | null;
type Ex4 = undefined & null;

/* 推論結果為： number */
type Ex5 = number | 123;

/* 推論結果為： 123 */
type Ex6 = number & 123;

/* 推論結果為： number，因為 never 為任何型別的基底型別 */
type Ex7 = never | number;

/* 推論結果為： never */
type Ex8 = never & string;

/* 推論結果為： any */
type Ex9 = any | number;

/* 推論結果為： any，為作者認定的例外 */
type Ex10 = any & string;

/* 推論結果為： unknown */
type Ex11 = unknown | number;
```

```
/* 推論結果為：string */
type Ex12 = unknown & string;

/* 推論結果為：any */
type Ex13 = never | any;

/* 推論結果為：never，因為 never 仍然是 any 的基底型別 */
type Ex14 = never & any;

/* 推論結果為：unknown */
type Ex15 = never | unknown;

/* 推論結果為：never，因為 never 仍然是 unknown 的基底型別 */
type Ex16 = never & unknown;

/* 推論結果為：any */
type Ex17 = any | unknown;

/* 推論結果為：any */
type Ex18 = any & unknown;

/* 推論結果為：any */
type Ex19 = never | any | unknown;

/* 推論結果為：never */
type Ex20 = never & any & unknown;
```

12. 是。

13. 否，因為 never 為任何值對應之型別之基底型別，因此反推 never 不屬於任何除了 never 以外的型別。

14. 是。

► 第五章 TypeScript 類別基礎

- 成員變數（Member Variable）與建立出的實體屬性（Property）儘管差異不大，但是實體屬性可以視為開放（Public）狀態的成員變數。

2. 類別建構實體，也就是初始化物件的函式。
3. 存取修飾子（Access Modifiers）可以調整成員的封裝狀態，分成開放（Public）、私有（Private）以及保護（Protected）模式；開放狀態泛指成員可以在任何地方使用、私有則是只能在類別宣告內部使用，而保護模式下的成員除了可以在類別內部使用外，也可以在繼承的子類別下使用該成員。
4. 參考程式碼：

```
class Example {  
    private prop2: number;  
    protected prop3: boolean = true;  
  
    constructor(public input1: number, input2: string) {  
        this.prop2 = input2.toUpperCase();  
    }  
}
```

5. 參考程式碼：

```
class Rectangle {  
    constructor(private width: number, private height: number) {}  
  
    public calcArea() {  
        return this.width * this.height;  
    }  
  
    public calcCircumference() {  
        return (this.width + this.height) * 2;  
    }  
}
```

6. 參考程式碼：

```
class Rectangle {  
    constructor(private width: number, private height: number) {}  
  
    public calcArea() {  
        return this.width * this.height;  
    }
```

```
}

public calcCircumference() {
    return (this.width + this.height) * 2;
}

public setWidth(width: number) {
    this.width = width;
}

public setHeight(height: number) {
    this.height = height;
}
}
```

7. 參考程式碼：

```
class Rectangle {
    constructor(private width: number, private height: number) {}

    get area() {
        return this.width * this.height;
    }

    get circumference() {
        return (this.width + this.height) * 2;
    }

    public setWidth(width: number) {
        this.width = width;
    }

    public setHeight(height: number) {
        this.height = height;
    }
}
```

8. 參考程式碼：

```
class Rectangle {
    constructor(private width: number, private height: number) {}
```

```
get area() {
    return this.width * this.height;
}

get circumference() {
    return (this.width + this.height) * 2;
}

set dimension(input: [number, number]) {
    this.width = input[0];
    this.height = input[1];
}
}
```

9. 參考程式碼：

```
class Rectangle {
    /* 同第 8 題宣告方式 */
}

class Square extends Rectangle {
    constructor(size: number) {
        super(size, size);
    }
}
```

10. 參考程式碼：

```
class Rectangle {
    static area(width: number, height: number) {
        return width * height;
    }

    static circumference(width: number, height: number) {
        return (width + height) * 2;
    }
}
```

11. 類別普通成員與靜態成員差異在於，宣告普通成員為建構出的實體之介面，而靜態成員則是類別建構子函式本身的成員，因此靜態成員只會建構一次，也就是類別被宣告的那個剎那；因為兩者性質不同，故命名可以重複。

12. 是，但存取修飾子必須優先於 static 關鍵字。
13. 事實上不會，因為子類別與父類別建構出的實體結構一樣，故不會有問題。
14. 子類別建構的實體若指派到父類別的型別變數上是絕對可以的；但子類別有額外宣告父類別所沒有的成員，父類別的實體就不能指派到子類別型別的變數上。
15. 經過實驗結果，不管成員是否為開放模式，只要子類別有額外宣告父類別所沒有的成員，父類別的實體就不能指派到子類別型別的變數上。
16. 由於靜態成員與普通成員性質不同，因此不會有問題。

► 第六章 TypeScript 類別基礎

1. 普通介面為代表物件性質（Property）以及函式（Method）的組合；純函式介面的宣告僅止於純函式規格的宣告。
2. 是，類別使用存取方法（Accessors）依然能夠達到模擬物件屬性的目的，差別在於內部的實作方式與普通屬性不一樣。
3. Addition1 為純函式介面的宣告，因為沒有特別命名；Addition2 為普通介面的宣告，其規格為在物件上實踐名為 addition 的方法而已。
4. 同一個名稱的介面重複宣告時會進行融合，而剛好都是宣告純函式介面，因此融合過後會進行函式的超載性宣告（Function Overloading）。
5. 否，Addition1 中的函式介面，兩個參數可以為數字或字串型別，因此有四種輸入組合（數字 - 數字、字串 - 字串、數字 - 字串、字串 - 數字）；而後者 Addition2 中的函式介面，經過超載性宣告的結果，只會有兩種輸入組合（數字 - 數字、字串 - 字串）。
6. 使用上儘管差異看起來不大，不過可以從幾個點簡單點出差別：
 - 型別化名可以表示任何一種資料型態與結構；介面除了原始型別與特殊型別（如：元組、列舉等）之外，只能表達物件型別的結構，包含

JSON 物件、函式與類別的規格。

- 型別化名的意義是宣告靜態型資料結構，作者傾向不鼓勵頻繁的更動型別化名的宣告內容；介面則是宣告物件應實踐出的規格，使用上比較靈活，可以自由組合
- 型別化名在 TypeScript 裡的運作，事實上並非宣告出全新的型別，而是代替某複雜結構的“名稱”；介面的宣告則在 TypeScript 編譯過程中，用該介面的名稱代表某物件或函式規格
- 型別化名嚴格來說不能夠像介面一般進行延伸，不過仍然可以藉由交集複合（Intersection）的方式處理；介面則可以自由延伸（亦或者被稱作是介面的繼承）

► 第七章 深入型別系統 III 泛用型別

1. 以下為參考程式碼：

```
type isTruthy<T> = (input: T) => boolean;
```

2. 以下為參考程式碼：

```
type Primitives = number | string | boolean | null | undefined;
type isTruthy<T extends Primitives> = (input: T) => boolean;
```

3. 變數 arr 被推論之型別為 number[]，而 TypeScript 內建陣列的另一種泛用型別表示方式為 Array<T>，而此時 arr 中的 T 為數字型別；由於泛用型別中的型別參數可以間接表達方法應該填入參數型別，而檢視程式碼中 arr.forEach 的型別提示就已經告訴開發者，該方法內部的回呼函式，參數會自動被推論為數字型別：

```
/* 以下為 VSCode 編輯器應當出現的提示內容： */
(method) Array<number>.forEach(
    callbackfn: (
        value: number,
        index: number,
        array: number[])
    ) => void,
```

```
thisArg?: any  
): void
```

4. 以下為參考程式碼：

```
type CustomReadonly<T> = {  
    readonly [Key in keyof T]: T[Key]  
}
```

5. 介面 `I<T>` 只有一個型別參數 `T`，而該型別參數所代表之型別會自動代入到 `method1` 中該型別參數存在的地方，以及 `method2<U>` 的輸出型別；而 `method2<U>` 的型別參數 `U` 會代入到輸入部分；最後，`method3<T>` 的型別參數恰好跟介面 `I<T>` 的型別參數命名一樣，不過它會以 `method3<T>` 的型別參數的值為主，代入到 `method3<T>` 的輸入與輸出部分。
6. 函式 `ex1` 的輸入部分，如果用明文物件形式代入值時，就只能出現一個屬性，也就是 `length`；`ex2` 的輸入部分，只要出現 `length` 屬性的物件（包含陣列）都可以代入。

```
/* ex1 只能代入以下的值 */  
ex1({ length: 123 });  
  
/* ex2 可以代入至少有 length 屬性的值 */  
ex2({ length: 123 });  
ex2({ length: 123, hello: 'world' });  
ex1([1, 2, 3]);
```

7. 以下為參考程式碼：

```
interface Dictionary<T = any> {  
    [key: string]: T;  
}
```

```
function valuesOfJSON<  
    T extends Dictionary,  
    U extends keyof T  
>(obj: T): T[U][] {  
    /* ... */  
}
```

► 第八章 TypeScript 模組系統

1. 前者為引入預設輸出（Default Export）的模組；後者則是引入特定的模組。

2. 可以。

3. 以下為參考程式碼：

```
/* 方法 1: */
import U, { T } from '...';

/* 方法 2: */
import { default as U, T } from '...';

/* 方法 3: */
import * as M from '...';
// M.default 為 U; M.T 為 T
```

4. 撰寫型別定義檔並且必須命名為 `example-module.d.ts`，而後將 `addition` 函式規格定義出來：

```
/* example-module.d.ts */
declare function addition(
    input1: number | string,
    input2: number | string
): number;
```

5. 必須額外安裝 Lodash 的型別定義檔，也就是 `@types/lodash` 套件。

► 第十章 常用 ECMAScript 標準語法

1. 以下為參考程式碼：

```
/* 淺層解構法： */
let {
    personalWebsite: ex1
}: { personalWebsite: string } = maxwell.socialLinks;

/* 深層解構法： */
```

```
let {
  socialLinks: { personalWebsite: ex2 }
}: {
  socialLinks: { personalWebsite: string }
} = maxwell;

console.log(ex1); // => 'example.com'
```

2. 本題的程式碼可以被看作成：

```
let arr = [1, 2];
[arr[1], arr[0]] = [1, 2];
```

經過解構式的解讀，arr 的第 0 個元素被指派數值 2、第 1 個元素被指派數值 1，因此結果為：

```
console.log(arr);
// => [2, 1]
```

因此推論效果為交換該陣列裡的兩個元素；本題程式碼等效於：

```
let arr = [1, 2];

/* 交換兩個元素 */
let temp = arr[1];
arr[1] = arr[0];
arr[0] = temp;
```

3. 本題為陷阱題，答案結果是 [1, 1]，因為本題的等效程式碼為：

```
let arr = [1, 2];
arr[1] = arr[0]; // ← 此時 arr[0] 為 1，被指派到 arr[1]
arr[0] = arr[1]; // ← 此時 arr[1] 為 1，被指派到 arr[0]
```

與前一題差別在於，解構過程中是新建一個物件後解構：

```
[arr[1], arr[0]] = [arr[0], arr[1]];
~~~~~ 新建一個物件解構
```

然而，本題卻是直接用同一個物件參照進行解構，而後再塞到同一個物件參照對應的值，因此才會造成不同結果。

```
[arr[1], arr[0]] = arr;
```

~~~ 用同一個物件參照解構並指派到同一個物件參照

4. 以下為參考程式碼：

```
function deepCopy(input: JSONObj): JSONObj {  
    // 淺層複製  
    let result = { ...input };  
  
    // 使用遞迴 Recursion：  
    // 如果複製的結果，裡面的值為物件時，再進行淺層複製，一直遞迴下去  
    for (let prop in input) {  
        if (typeof prop === 'object') {  
            result[prop] = deepCopy(result[prop]);  
        }  
    }  
  
    return result;  
}
```

5. 最大的差異在於，陣列的元素可以重複，而 ES6 Set 內的元素不一樣；二來是陣列與 ES6 Set 提供的 API 具有很大的差異，像是陣列元素順序有差異，因此會有 Push 或 Unshift 之類的方法新增元素，而 ES6 Set 的元素無關乎順序，因此新增元素才會只有 Add 方法。
6. JSON 物件的鍵只能為字串或數字型別的值，而 ES6 Map 物件則可以以任何物件值作為鍵。
7. 使用 ES6 Set 即可：

```
function distinctNumberCount(input: Array<number>): number {  
    return new Set(input).size;  
}
```

## ► 第十一章 常用 ECMAScript 標準語法 非同步程式設計篇

1. 同步的程式運作過程，會按照程式碼順序一行一行地執行，也就是說，後面的程式碼必須得等到前面的程式執行完畢才會接續，因此遇到運算耗

費量過大的程式碼容易造成阻塞（Blocking）問題；非同步程式碼在運作上，順序則不一定，由於 JavaScript 引擎（如：V8）設計上只有單線程（Single-Threaded），因此程式運作上是併發（Concurrent）執行的，不太會遇到阻塞問題。

2. 印出訊息順序為：Log 1 → Log 3 → Log 5 → Log 4 → Log 2
3. 實現運行整段程式碼，其印出訊息順序為：Log 1 → Log 2 → Log 4 → Log 3 → Hello world!；Promise 物件在一開始初始化時，就會執行函式的內容，因此最先印出 Log 1 訊息；然而，遇到 resolve（或 reject）函式時，儘管看似應該要終止了，但 Promise 物件內的函式依然會繼續執行，於是印出 Log 2；由於 Promise 物件本身是非同步程式，根據事件迴圈（Event Loop）的原理，程式碼會先執行到結束——也就是印出 Log 4 後，Promise 中的 then 方法才會開始執行，而 then 方法裡依序印出 Log 3 以及參數值，為 Hello world! 字串。
4. Promise 物件本身為泛用型別，而 `Promise<T>` 中的型別參數 T 就代表了 then 方法裡回呼函式中的參數型別。
5. 使用 `Promise.race`，只要 `timeoutAfter` 回傳的 Promise 物件 `reject` 時就會將整個 `Promise.race` 終結。
6. 是，只要是任何回傳 Promise 物件相關的資源都可以在非同步函式裡，用 `await` 關鍵字進行等待該非同步程式結束的情形。
7. `<Type>-Like` 類型的型別可以避免輸入過於限制的情形，比如一個函式要求輸入型別為 `ArrayLike<T>` 以及另一個函式則是輸入型別為 `Array<T>`，兩者相比，前者只要有長得像陣列的資料結構，譬如具有 `length` 性質的物件等就可以輸入，但後者強調一定要陣列型的資料結構輸入才行；

詳細可以參見：<https://stackoverflow.com/questions/43712705/why-does-typescript-use-like-types>



## 內容簡介

本書內容改編自第 11 屆 iT邦幫忙鐵人賽，Modern Web 組冠軍網路系列文章——《讓 TypeScript 成為你全端開發的 ACE !》——除了是單純入門 TypeScript 語言的技術書籍外，也是第一本屬於台灣本土的 TypeScript 專書；由微軟（Microsoft）研發出的 TypeScript，為近年來逐漸熱門的技術，是軟體社群上擁有眾多廠商與開發者青睞的前、後端開發工具，本書旨在介紹 TypeScript 這門語言的使用方法與技巧。

## 四大重點

### ◆ 漸進式型別系統：原始、物件、泛用與常用進階型別等。

本書一大重點在於型別系統的重要性以及使用方式；除此之外，可以從型別系統來認識一門語言的特性、運作過程、效果與細節，對於程式語言本身的設計會有更深層的理解。

### ◆ TypeScript 高效技巧：除錯、規格查詢、結合 JavaScript 專案的方法。

好的 TypeScript 程式碼除了可以提升開發效率外，除錯（Debug）的痛苦程度會大大的降低；並且也會介紹原生 JavaScript 專案是如何不需全部重寫成 TypeScript 就可以結合 TypeScript 專案的秘密。

### ◆ 標準物件導向程式設計入門：類別、介面、SOLID 原則。

TypeScript 對於物件導向程式設計方面的語法支援完備度較現階段的 JavaScript 以及 ECMAScript 標準高，因此讀者可以藉由 TypeScript 學習到物件導向程式設計的觀念。

### ◆ TypeScript & ECMAScript：語法糖的使用、非同步程式設計。

本書不僅會討論到 JavaScript、ECMAScript 以及 TypeScript 之間的關係，並且也會介紹在 TypeScript 使用 ECMAScript 標準語法時需要注意的事項。

## 好評推薦

「很開心看到這次 Max 參加第11屆iT邦幫忙鐵人賽冠軍的大作《讓 TypeScript 成為你全端開發的 ACE !》能夠付梓出版，除了為台灣本土資訊業界帶來一注活水，裡頭滿滿的範例以及各種貼心的小提示，手把手帶領讀者學習 TypeScript ，相信一定不會讓你失望。」

——Kuro

Vue.js Taiwan 社群主辦人  
《0 陷阱！0 誤解！8 天重新認識 JavaScript !》作者

ISBN 978-986-434-489-5  
MP22016

博碩文化股份有限公司  
建議書區 網頁開發・JavaScript  
定價・650元

