前言

這本書是為各行各業的程式設計師所寫的，例如專業的 JavaScript 工程師、C# 群眾、Java 支持者、Python 愛好者、Ruby 粉絲、Haskell 技術迷等等。不管你是使用何種語言來撰寫程式，只要你有一些程式設計的經驗，並且懂得函式、變數、類別和錯誤的基本知識，這本書就很適合你。若有使用 JavaScript 的經驗，包括了解 Document Object Model（DOM）的基礎，那麼在閱讀的過程中，將會很有幫助，雖然我們沒有深入探討那些概念，但它們可是絕佳範例的源泉，如果你不熟悉它們，可能就無法完全理解某些例子背後的意義。

無論你以前用過什麼程式語言，我們大家都有的，就是調查例外、逐行追查程式碼以找出錯誤原因及如何修正的共通經驗。這就是 TypeScript 可以幫助你避開的經驗，方法是自動檢查你的程式碼，指出你可能忽略的缺失。

如果你之前未曾用過靜態定型（statically typed）的語言，那也沒關係。我會教導你型別的相關知識，還有如何有效運用它們來減少你程式當掉的機會、如何為你的程式碼編寫更好的說明文件，以及如何拓展你應用程式的規模，涵蓋更多使用者，並跨越更多工程師和伺服器。我會盡量避免包山包海的含糊用詞，以直覺、容易記住且務實的方式來解說概念，並在過程中使用大量的範例來讓那些概念更為具體。

這就是 TypeScript 的特色：跟許多具有型別的其他語言不同，TypeScript 非常實用。它發明了全新的概念來幫助你更簡潔且精確地表達，讓你以有趣、現代且安全的方式撰寫應用程式。

本書的架構

本書有兩個目標：讓你對 TypeScript 語言的運作方式（理論）有深入的理解，並提供大把的實用建議，告訴你如何寫出能夠正式上線的 TypeScript 程式碼（實務）。

因為 TypeScript 是如此務實的語言，理論很快就能轉成實務，所以本書大部分都是這兩者的混合體，只有最前面幾章幾乎完全介紹理論和後面少數幾章完全涵蓋實務。

一開始我會先講解編譯器（compilers）、型別檢查器（typecheckers）和型別（types）是什麼東西。然後為 TypeScript 中的不同型別和型別運算子（type operators）提供一個概觀，說明它們的用途，以及如何使用。奠基於我們所學過的，接著我會涵蓋一些進階的主題，像是 TypeScript 最精密的型別系統功能、錯誤處理，以及非同步程式設計（asynchronous programming）。最後，我會介紹如何搭配使用 TypeScript 和你最愛的框架（前端或後端）、將你現有的 JavaScript 專案遷移至 TypeScript，以及在正式環境執行你的 TypeScript 應用程式。

每章結尾幾乎都會有一組練習題目。請試著自己做做看，它們會為你帶來僅是閱讀所比不上的深刻直覺。這些章節習題的答案可在線上找到：*https://github.com/bcherny/programming-typescript-answers*。

本書的風格

本書各處，我都會試著維持單一的程式碼風格（code style）。這種風格的某些面向相當個人化，例如：

- 我只在必要時使用分號（semicolons）。
- 我以兩個空格（two spaces）進行縮排。
- 我會在程式只是快速的範例，或是程式的結構比細節更重要時，使用簡短的變數名稱，像是 `a`、`f` 或 `_`。

而這種程式碼風格的另一些面向，則是我認為你也應該照著做的。幾個例子包括：

- 你應該使用最新的 JavaScript 語法和功能（最新的 JavaScript 版本通常單純稱作「esnext」）。這會讓你的程式碼與最新的標準保持一致，增進可互通性，以及 Google 搜尋的可行性，這能加快新雇員工上手的速度。這也能讓你享受到強大的現代 JavaScript 功能所帶來的好處，例如箭號函式（arrow functions）、承諾（promises）和產生器（generators）。

- 大多數情況下，你都應該使用分散（...）來讓你的資料結構保持不可變（immutable）。[1]
- 你應該確定每個東西都有一個型別，並盡可能推論出來。要小心別濫用明確型別（explicit types），這可以幫助你的程式碼維持清晰和簡潔，並藉由揭露不正確的型別，增進安全性，而非只是東貼西補，撐過就好。
- 你應該讓你的程式碼是可重複使用（reusable）並且泛用（generic）的。多型（polymorphism）是你的最佳幫手（請參閱第 4 章的「多型」）。

當然，這些風格算不上什麼新東西。不過當你遵循這些風格，TypeScript 就能運作得特別好。TypeScript 的內建降層編譯器（downlevel compiler）、對唯讀型別（read-only types）的支援、強大的型別推論（type inference）、對多型的深入支援，以及結構完整的型別系統都鼓勵良好的編程風格（coding style），同時該語言仍然具備非常好的表達能力，並與底層的 JavaScript 互通無礙。

開始前還有幾件事要說明。

JavaScript 並不對外提供指標（pointers）和參考（references），而是具有值（value）與參考的型別。值是不可變的，包括了字串、數字和 boolean（邏輯真假值）等，而參考則指向經常是可變的（mutable）的資料結構，例如陣列、物件和函式。在本書中使用「值（value）」這個詞的時候，我通常泛指的是一個 JavaScript 值或一個參考。

最後，與 JavaScript 或型別不對的第三方程式庫或舊版程式碼（legacy code）交互運作時，或是你很趕時間的時候，你可能會發現自己寫出的是不甚理想的 TypeScript 程式碼。本書主要呈現的就是撰寫 TypeScript *應該*有的方式，並據理論述，試圖說服你，為何你應該盡全力避免這種妥協。不過實務上，你的程式碼有多正確，取決於你和你的團隊。

本書編排慣例

本書使用下列的排版慣例：

1 如果你沒有 JavaScript 背景，這裡有個例子：如果你有一個物件 `o`，而你想要新增帶有值 `3` 的一個特性 `k` 給它，你可以直接變動 `o`，即 `o.k = 3`，或是將變更套用到 `o`，並建立一個新的物件作為結果：`let p = {...o, k: 3}`。

第一章

簡介

所以，你決定購買一本關於 TypeScript 的書，原因何在？

或許是因為你厭倦了 JavaScript 那些古怪的 `cannot read property blah of undefined` 錯誤。又或是你聽說過 TypeScript 能夠幫助你的程式碼更好的拓展規模，於是想要一探究竟。又或者你是寫 C# 的，但一直想試試這些甚囂塵上的 JavaScript 玩意兒。或者你是函式型程式設計師（functional programmer），覺得該是時候將技能提升到下一個層次了。又或者你老闆受夠了你的程式碼造成的產品問題，於是送了這本書給你當作聖誕禮物（如果我太超過了請制止我）。

不管你的理由為何，你所聽到的都是真的。TypeScript 會是下一代 Web apps、行動 apps、NodeJS 專案和 IoT（物聯網，Internet of Things）裝置的主力語言。它可藉由檢查常見的錯誤讓你的程式更加安全、作為你自己與未來工程師的說明文件、讓重構（refactoring）輕鬆一點，並使你幾乎半數的單元測試（unit tests）變得沒必要（「什麼單元測試？」）。TypeScript 將會使你作為程式設計師的生產力加倍，也能讓你有機會跟對街那可愛的咖啡廳店員約會。

但在你急著過街之前，讓我們稍加說明，先從這個開始：我說「更安全」時，代表的到底是什麼呢？當然，我所說的，是指**型別安全性**（*type safety*）。

型別安全性

使用型別來防止程式做出無效的事情。[1]

這裡有幾個無效的例子：

- 把一個數字乘以一個串列（list）
- 使用由字串組成的一個串列呼叫一個函式，但它實際上需要的是由物件組成的串列
- 在一個物件上呼叫某個方法（method），但實際上該物件並沒有那個方法
- 匯入一個最近移動過的模組（module）

有些程式語言會試著讓這樣的錯誤發揮最大的用處，嘗試在你做了某些無效的事情時，找出你真正的意圖為何，畢竟我們總得盡力而為，不是嗎？以 JavaScript 為例：

```
3 + []              // 估算（evaluate）為字串 "3"

let obj = {}
obj.foo             // 估算為 undefined

function a(b) {
  return b/2
}
a("z")              // 估算為 NaN
```

特別注意到，JavaScript 並不會在你試著進行明顯無效的事情時擲出例外（exception），而是全力善用它，並盡可能避免例外。JavaScript 有幫忙嗎？當然有。它有讓你更輕易且快速的抓出臭蟲嗎？或許沒有。

現在想像一下，如果 JavaScript 是擲出更多的例外，而非安靜地從我們給它的東西裡面努力榨出一些汁液，那我們可能會得到像這樣的回應：

```
3 + []              // 錯誤：你真的想把一個數字與一個陣列相加嗎？

let obj = {}
obj.foo             // 錯誤：你忘記在 obj 上定義特性 "foo" 了。

function a(b) {
```

1 取決於你所用的靜態型別語言，「無效（invalid）」代表的意義也可能隨之而變，從碰到時會使程式整個當掉的錯誤，到雖然不會當掉，但顯然不合理的東西都有。

```
  return b/2
}
a("z")              // 錯誤：函式 "a" 預期一個數字，
                    // 但你給了它一個字串。
```

別誤會我的意思：就程式語言來說，嘗試為我們修正錯誤，是一種很棒的功能（如果不僅限於程式那就更好了！）。但 JavaScript 的這項功能，讓「你在程式碼中犯了錯誤」與「你**發現**你在程式碼中犯了錯誤」之間失去了關聯。通常，那意味著，你第一次知道自己犯了錯誤時，是從別人口中聽到的。

所以會有這種問題出現：JavaScript 到底什麼時候會告訴你，你犯了錯呢？

沒錯：正是在你**執行**（*run*）程式的時候。你的程式可能會在於瀏覽器中測試時，或使用者拜訪你的網站時，或你進行單元測試時被執行。若你是個有紀律的人，撰寫了很多單元測試及端對端的測試（end-to-end tests），會在發布程式碼之前進行煙霧測試（smoke test），並且會在提供給使用者之前，於內部好好測試一段時間，你大概就能在使用者發現之前，先找到你的錯誤。但如果你不是呢？

這就是 TypeScript 派上用場的地方了。比「TypeScript 會提供你有用的錯誤訊息」這件事更酷的是它給你這些訊息的**時機**：TypeScript 會在**你的文字編輯器中**提供錯誤訊息，就在**你輸入程式碼的同時**。這表示你不必仰賴單元測試或煙霧測試或同事，就能捕捉到這類問題：TypeScript 會為你捕捉它們，並在你撰寫程式的過程中提醒你它們的存在。讓我們看一下 TypeScript 會對前面的例子說些什麼：

```
3 + []              // Error TS2365：運算子 '+' 不能套用到型別 '3'
                    // 和 'never[]' 上。

let obj = {}
obj.foo             // Error TS2339：特性 'foo' 在型別 '{}' 之上並不存在。

function a(b: number) {
  return b / 2
}
a("z")              // Error TS2345：型別為 '"z"' 的引數（argument）不可指定（assign）給
                    // 型別為 'number' 的參數（parameter）。
```

除了能夠消除與型別有關的一整類臭蟲，這其實也會改變你撰寫程式碼的方式。你發現自己會先在型別的層次（type level）描述出程式的輪廓，然後於值的層次（value level）填入細節[2]，你會在設計程式的過程中思考邊緣情況（edge cases），而非事後才那麼做。你會設計出較為簡單、快速、容易理解的程式，維護起來也比較輕鬆。

你準備好開始這段旅程了嗎？我們出發吧！

2 如果你不確定這裡的「型別層次」是什麼意思，別擔心。我們會在後續的章節中深入說明。

第二章

TypeScript：10_000 英尺高的觀點

在接下來幾章中，我會介紹 TypeScript 這個語言，為你提供 TypeScript Compiler（TSC）運作方式的一個概觀，並帶你一覽 TypeScript 的各項功能，以及藉由這些特色所能發展的模式。我們會先從編譯器（compiler）開始。

編譯器

取決於你過去（也就是在你決定購買本書並承諾過著具有型別安全性的生活之前）所用過的程式語言，你對程式如何運作，會有不同的見解。TypeScript 的運作方式與其他主流語言（例如 JavaScript 或 Java）比起來較為不尋常，所以繼續前行之前，很重要的就是先理解這一點。

讓我們先做廣義的說明：程式是含有一些文字的檔案，而這些文字的撰寫人就是你，也就是程式設計師。這些文字會由一個特殊的程式進行剖析（parse），這個程式就叫做**編譯器**（*compiler*），它會將之變換為一個**抽象語法樹**（*abstract syntax tree*，AST），這是忽略了空白、註解和你對於「tabs vs. spaces 爭論」所持立場的一種資料結構。接著編譯器會將那個 AST 轉換為一種低階的表現方式（lower-level representation），稱作 *bytecode*。你可以將這種 bytecode 餵入給叫做 *runtime*（**執行環境**）的另一個程式，以對它進行估算（evaluate）並取得結果。因此，當你執行一個程式，你實際上所做的事情是告訴那個 runtime 去估算編譯器所產生的 bytecode，而這個 bytecode 產自你的原始碼（source code）剖析之後得到的 AST。細節可能有所不同，但對大多數的語言來說，這都算是一種準確的高階觀點。

再一次，這些步驟為：

1. 程式被剖析為一個 AST.
2. AST 被編譯為 bytecode。
3. Bytecode 由 runtime 來估算。

TypeScript 的特殊之處在於，它不是直接編譯成 bytecode，TypeScript 會被編譯為…JavaScript 程式碼！然後你會以正常的方式執行這個 JavaScript 程式碼，在瀏覽器中，或者使用 NodeJS，或以紙筆手動進行（對機器反叛之後才讀到這一段的那些人來說）。

此時你可能會想：「等等！上一章你說過 TypeScript 會讓我的程式碼變得更安全！那是在何時發生？」

很好的問題。實際上我跳過了關鍵的一步：在 TypeScript Compiler 為你的程式產生一個 AST 之後，但在它發出程式碼之前，它會為你的程式碼進行**型別檢查**（*typecheck*）。

> **型別檢查器（Typechecker）**
>
> 會驗證你的程式碼是否具備型別安全性的一種特殊程式。

這個型別檢查動作就是 TypeScript 背後的魔法。TypeScript 正是藉此確保你的程式是否如預期運作、是否沒有明顯的錯誤，以及對街那個可愛店員答應之後，是否真的會打電話給你（別擔心，她或許只是在忙）。

所以如果我們納入型別檢查和 JavaScript 程式碼的發出動作（emission），TypeScript 的編譯過程現在看起來大概就像圖 2-1 那樣：

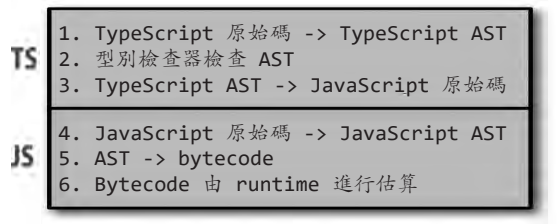

圖 2-1 編譯並執行 TypeScript

步驟 1 到 3 由 TSC 進行，而 4 到 6 則由你所用的瀏覽器、NodeJS 或其他 JavaScript 引擎中的 JavaScript runtime 進行。

JavaScript 編譯器和 runtime 通常會被結合成單一個叫做**引擎**（*engine*）的程式，作為程式設計師，這就是你一般會與之互動的對象。這是 V8（NodeJS、Chrome 和 Opera 背後的引擎）、SpiderMonkey（Firefox）、JSCore（Safari），以及 Chakra（Edge）運作的方式，也是 JavaScript 看起來像是一個**直譯式**（*interpreted*）語言的原因。

在這個過程中，步驟 1 到 2 使用你程式的型別，步驟 3 則否。這值得重述一次：***TSC 將你的程式碼從 TypeScript 編譯為 JavaScript 的時候，它並不會查看你的型別***。這表示你程式的型別永遠都不會影響到你程式所產生的輸出，它們只被用來進行型別檢查。這種特色能讓你毫無疑慮地把玩你程式的型別，你可以更新、改善它們，而不用擔心會弄壞你應用程式。

型別系統

現代語言有著各式各樣不同的**型別系統**（*type systems*）。

> **型別系統**
>
> 型別檢查器用來為你的程式指定型別的一組規則。

一般來說，型別系統有兩種：你必須以明確的語法告知編譯器每樣東西的型別的型別系統，以及會自動為你推論出東西型別的型別系統。這兩種做法都有各自必須取捨之處。[1]

這兩種型別系統都有影響到 TypeScript：你可以明確地標注你的型別，或讓 TypeScript 為你推論出大部分的型別。

1 在這兩端之間有各種語言存在：JavaScript、Python 與 Ruby 會在執行時期（runtime）推論型別。Haskell 和 OCaml 會在編譯時期（compile time）推論並檢查缺少的型別。Scala 和 TypeScript 需要一些明確的型別，並會在編譯時期推論並檢查其餘的型別。Java 及 C 則需要為幾乎所有的東西加上明確的注釋，並會在編譯時期檢查它們。

要向 TypeScript 明確指出型別為何，就使用注釋（annotation）。注釋的形式為 *value: type*，它們告知型別檢查器：「嘿！你有看到這個 *value* 嗎？它的型別為 *type*」。讓我們看幾個例子（每行後的註解是 TypeScript 所推論出來的實際型別）：

```
let a: number = 1                 // a 是一個數字
let b: string = 'hello'           // b 是一個字串
let c: boolean[] = [true, false]  // c 是由 boolean 所組成的一個陣列
```

如果你希望 TypeScript 為你推論出型別，那就省略它們，讓 TypeScript 工作：

```
let a = 1                         // a 是一個數字
let b = 'hello'                   // b 是一個字串
let c = [true, false]             // c 是由 boolean 所組成的一個陣列
```

你馬上就能注意到 TypeScript 多擅長為你推論出型別。如果你省略那些注釋，推論出來的型別是一樣的！在本書各處，我們都只會在必要時使用注釋，而盡可能讓 TypeScript 為我們施展推論魔法。

一般來說，讓 TypeScript 為你推論出盡可能多的型別，使明確指出型別的程式碼維持在最少，是一種良好的風格。

TypeScript vs. JavaScript

讓我們深入探討 TypeScript 的型別系統，以及它與 JavaScript 的型別系統比起來如何。表 2-1 提供了一個概觀。要好好了解它們的差異，關鍵是建造出 TypeScript 如何運作的心智模型。

表 2-1 比較 JavaScript 和 TypeScript 的型別系統

型別系統的功能	JavaScript	TypeScript
型別是如何繫結（bind）的呢？	動態	靜態
型別會自動轉換嗎？	是	否（大多如此）
型別何時檢查？	執行時期	編譯時期
錯誤何時浮現？	執行時期（大多如此）	編譯時期（大多如此）

型別如何繫結？

動態的型別繫結代表 JavaScript 必須實際執行你的程式才能知道其中各個東西的型別。執行程式之前，JavaScript 沒辦法知道你的型別是什麼。

TypeScript 是一種**逐步定型**（*gradually typed*）的語言。這表示，TypeScript 運作得最好的時候，是它在編譯時期知道你程式中每樣東西的型別之時，但它並不一定得要知道每個型別才能編譯你的程式。即使是在未具型別的程式中，TypeScript 也能為你推論出一些型別並捕捉某些錯誤，但因為不知道每樣東西的型別，它會讓很多的錯誤溜到使用者那邊。

這種逐步定型很適合用來將未具型別的 JavaScript 舊有源碼庫（legacy codebases）移植到具型別的 TypeScript（更多資訊請參閱第 11 章的「逐步從 JavaScript 遷移至 TypeScript」），但除非你正在移植源碼庫，否則你應該追求 100% 的型別涵蓋率。除了特別註明之處，那也會是本書所採用的途徑。

型別會自動轉換嗎？

JavaScript 是弱型別（weakly typed）語言，這代表如果你做了某些無效的事，像是相加一個數字與一個陣列（如我們在第 1 章中做的那樣），它會套用一組規則來找出你真正的意圖為何，如此它才能用你所給的來盡力做些事情。讓我們逐步檢視 JavaScript 如何估算 `3 + [1]` 這個具體的例子：

1. JavaScript 注意到 `3` 是一個數字，而 `[1]` 是一個陣列。
2. 因為我們用的是 `+`，它假設我們想要將兩者串接（concatenate）。
3. 它隱含地將 `3` 轉為一個字串，產出 `"3"`。
4. 它隱含地將 `[1]` 轉為一個字串，產出 `"1"`。
5. 它串接結果，產出 `"31"`。

我們也可以更明確地這麼做（讓 JavaScript 避免步驟 1、3 和 4）：

```
3 + [1];                          // 估算為 "31"

(3).toString() + [1].toString() // 估算為 "31"
```

JavaScript 會為你進行聰明的型別轉換，試著想要幫上忙，而 TypeScript 會在你做了無效的事情時立即向你抱怨。當你把相同的 JavaScript 程式碼餵給 TSC，你會得到錯誤訊息：

```
3 + [1];                          // Error TS2365：運算子 '+' 無法套用到
                                  // 型別 '3' 和 'number[]'。

(3).toString() + [1].toString()  // 估算為 to "31"
```

如果你做了看起來不對的事，TypeScript 會向你抱怨，但若你的意圖有明確表達，TypeScript 就不會擋路。這種行為很合理：神智清醒的人誰會試著相加一個數字和一個陣列，還預期結果要是一個字串（當然，你新創公司地下室憑藉著燭光整天寫程式的 JavaScript 巫師 Bavmorda 或許除外）？

JavaScript 進行的這種隱含轉換可能會使錯誤來源十分難以追查，也是許多 JavaScript 程式設計師的禍根。這不僅讓工程師個人的工作難以完成，更會使得程式碼難以擴展到大型團隊的規模，因為那要求每名工程師都得知道你的程式碼所做的隱含假設。

簡而言之，如果你非得轉換型別，就明確地進行。

型別何時檢查？

大多數情況下，JavaScript 都不在意你給的是什麼型別，而是試著盡力將你給它的，轉成它所預期的東西。

另一方面，TypeScript 則會在編譯時期檢查你的程式碼（記得本章開頭所列的步驟 2 嗎？），所以你不需要實際執行你的程式碼就能看到前面例子的 `Error`。TypeScript 會**靜態分析**（*statically analyzes*）你的程式碼，以找出像這樣的錯誤，並在你執行程式前顯示它們。如果你的程式碼無法順利編譯，你就可以很清楚的知道你犯了錯誤，而你應該在嘗試執行程式碼之前加以修正。

圖 2-2 顯示我把上一個程式碼範例輸入到 VSCode（我所選的編輯器）時會發生什麼事。

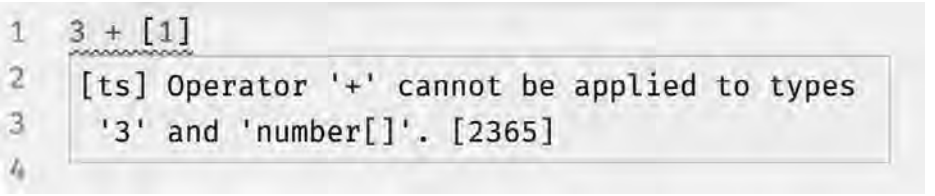

圖 2-2 VSCode 所回報的 TypeError

只要為你偏好的編輯器找到優良的 TypeScript 擴充功能，錯誤就會在你輸入程式碼的同時顯現出來，以紅色的扭曲線條標示。這可以讓「撰寫程式碼，發現你犯了個錯，更新程式碼來修正該錯誤」的回饋循環速度大幅提升。

錯誤何時浮現？

JavaScript 會在執行時期（runtime）擲出例外或進行隱含的型別轉換[2]。這代表你必須實際執行程式方能獲得「你做了一些無效的事」的實用訊號。在最好的情況下，那代表在單元測試的過程中發生；在最糟情況下，那意味著來自使用者的憤怒 email。

TypeScript 則會在編譯時期（compile time）丟出語法相關的錯誤及型別相關的錯誤。實務上，那表示這些種類的錯誤都會顯示在你的程式碼編輯器中，就在你打入程式碼的同時，如果你之前從未用過漸進式編譯（incrementally compiled）的靜態型別語言，這將是一種神奇的體驗[3]。

儘管如此，還是有很多錯誤是 TypeScript 沒辦法在編譯時期為你找出的，像是堆疊溢位（stack overflows）、網路連線壞掉，以及格式不對的使用者輸入等，這些都依然會導致執行時期的例外。TypeScript 所做的就是讓在純 JavaScript 世界中原本會是執行期錯誤的大部分錯誤在編譯期就能消除。

程式碼編輯器的設定

現在你對 TypeScript 編譯器和型別系統的運作方式有些直觀理解了，現在讓我們來設定你的程式碼編輯器（code editor），以便開始深入討論一些真實的程式碼。

首先是下載你撰寫程式用的程式碼編輯器。我喜歡 VSCode 是因為它提供了很不錯的 TypeScript 編輯體驗，但你也可以使用 Sublime Text、Atom、Vim、WebStorm 或任何你喜歡的編輯器。工程師對於 IDE 通常會很挑剔，所以我留給你自行決定。如果你想要使用 VSCode，請依循網站（*https://code.visualstudio.com/*）上的指示來設置。

2 當然，JavaScript 會在剖析你的程式後，並在執行前指出語法錯誤和少數的一些臭蟲（例如相同範疇中以同一個名字進行了多次的 `const` 宣告）。若你有在建置過程（build process）剖析你的 JavaScript（例如使用 Babel），你就能在建置時期發現那些錯誤。

3 漸進式編譯的語言可在你做了小型變更後，快速重新編譯，而不必重新編譯整個程式（即包括你沒動到的那些部分）。

TSC 本身是一個以 TypeScript 撰寫的命令列應用程式（command-line application）[4]，這表示你需要 NodeJS 才能執行它。請依照 NodeJS 官方網站（*https://nodejs.org*）上的指示取得 NodeJS 並在你的機器上執行。

隨著 NodeJS 而來的是 NPM，它是用來管理你專案依存關係（dependency）和策畫建置過程的套件管理程式（package manager）。我們會先用它來安裝 TSC 和 TSLint（用於 TypeScript 的 linter）。請開啟你的終端機程式，並建立一個新的檔案夾，然後在其中初始化一個新的 NPM 專案：

```
# 建立一個新的檔案夾
mkdir chapter-2
cd chapter-2

# 初始化一個新的 NPM 專案（依照提示進行）
npm init

# 安裝 TSC、TSLint 及 NodeJS 的型別宣告
npm install --save-dev typescript tslint @types/node
```

tsconfig.json

每個 TypeScript 專案的根目錄（root directory）都應該有一個叫做 *tsconfig.json* 的檔案。這個 *tsconfig.json* 正是 TypeScript 專案定義應該要編譯什麼檔案、要編譯到哪個目錄，以及要發出哪個版本的 JavaScript 這些設定的地方。

在你的根目錄創建一個新的檔案，叫做 *tsconfig.json*（`touch tsconfig.json`）[5]，然後在你的程式碼編輯器中打開它，輸入下列內容：

```
{
  "compilerOptions": {
    "lib": ["es2015"],
    "module": "commonjs",
    "outDir": "dist",
    "sourceMap": true,
    "strict": true,
    "target": "es2015"
```

4 這使得 TSC 被歸入一種神秘的編譯器類別，叫做 *self-hosting compilers*（自我提供的編譯器），或是「編譯自身的編譯器」。

5 就這個練習而言，我們是手動建立一個 *tsconfig.json*。未來你設置 TypeScript 專案時，你可以使用 TSC 內建的初始化命令來為你產生這個檔案：`./node_modules/.bin/tsc --init`。

```
    },
    "include": [
      "src"
    ]
  }
```

讓我們簡略地看過這些選項及它們的意義（表 2-2）：

表 2-2 tsconfig.json 的選項

選項	說明
include	TSC 應該查看哪些檔案夾以找出你的 TypeScript 檔案？
lib	TSC 應該假設你要執行程式碼的環境有哪些 API？這包括了像是 ES5 的 Function.prototype.bind、ES2015 的 Object.assign 及 DOM 的 document.querySelector 這些東西。
module	TSC 應該將你的程式碼編譯為使用哪個模組系統（CommonJS、SystemJS、ES2015 等）？
outDir	TSC 應該把產出的 JavaScript 程式碼放在哪個檔案夾？
strict	檢查無效程式碼時盡可能嚴格（strict）。這個選項強制要求你的程式碼全都得有適當的型別。我們會為本書中所有的範例使用這個選項，而你也應該為你的 TypeScript 專案使用它。
target	TSC 應該將你的程式碼編譯為哪個 JavaScript 版本（ES3、ES5、ES2015、ES2016）？

這些只是可用的選項中的少數幾個，*tsconfig.json* 支援數十個選項，而且不時會加入新的。實務上，你不會常常變更它們，除了換到新的 module bundler 時調整一下 module 與 target 的設定、寫 TypeScript 給瀏覽器用（你會在第 12 章學到這個主題的更多資訊）的時候新增 "dom" 到 lib，或將你現有的 JavaScript 程式碼移植到 TypeScript 時調整 strict 的程度（參閱第 11 章的「逐步從 JavaScript 遷移至 TypeScript」）。要找完整且最新的支援選項清單，請查閱 TypeScript 網站（*http://bit.ly/2JWfsgY*）的官方說明文件。

注意到，雖然使用一個 *tsconfig.json* 檔案來設定 TSC 很方便，因為它能讓我們檢查對於原始碼控制的配置，但其實大多數的 TSC 選項也都能從命令列設定。執行 ./node_modules/.bin/tsc --help 來查看可用的命令列選項。

tslint.json

你的專案也應該有一個 tslint.json 檔案，其中包含你的 TSLint 配置，規範了你希望程式碼使用的任何風格慣例（stylistic conventions，例如要用 tab 或 space 等）。

TSLint 的使用是選擇性的，但我們強烈建議所有的 TypeScript 專案都這麼做，以施加一致的編程風格。更重要的是，審閱程式碼時，這可以省下你跟同事爭論哪種風格比較好的時間。

下列命令會產生帶有預設 TSLint 配置的一個 `tslint.json` 檔案：

```
./node_modules/.bin/tslint --init
```

然後你可以為此加上覆寫（override）來符合你自己的編程風格。例如，我的 *tslint.json* 看起來像這樣：

```
{
  "defaultSeverity": "error",
  "extends": [
    "tslint:recommended"
  ],
  "rules": {
    "semicolon": false,
    "trailing-comma": false
  }
}
```

要找可用規則的完整列表，請查閱 TSLint 的說明文件（*https://palantir.github.io/tslint/rules/*）。你也能夠新增自訂的規則，或安裝預先設置好的額外規則（例如 ReactJS 用的：*https://www.npmjs.com/package/tslint-react*）。

index.ts

現在你設定好了你的 *tsconfig.json* 和 *tslint.json*，就建立一個 *src* 檔案夾放置你的第一個 TypeScript 檔案：

```
mkdir src
touch src/index.ts
```

你專案的檔案夾結構看起來應該像這樣：

```
chapter-2/
├── node_modules/
├── src/
│   └── index.ts
├── package.json
├── tsconfig.json
└── tslint.json
```

在你的程式碼編輯器開啟 *src/index.ts*，然後輸入下列 TypeScript 程式碼：

```
console.log('Hello TypeScript!')
```

然後，編譯並執行你的 TypeScript 程式碼：

```
# 以 TSC 編譯你的 TypeScript
./node_modules/.bin/tsc

# 以 NodeJS 執行你的程式碼
node ./dist/index.js
```

如果你有照這裡所有的步驟進行，你的程式碼就應該可以執行，而你應該會在你的主控台（console）中看到單一筆 log：

```
Hello TypeScript!
```

這樣就完成了，你從頭開始設置並執行了你的第一個 TypeScript 專案，幹得好！

既然這可能是你第一次從無到有設置一個 TypeScript 專案，我有試著說明了每個步驟，讓你對所有的組成部分有個感覺。下一次你可以採用幾個捷徑來讓事情快一點：

- 安裝 `ts-node`（*https://npmjs.org/package/ts-node*），透過它使用單一道指令來編譯並執行你的 TypeScript。
- 使用鷹架（scaffolding）工具，像是 `typescript-node-starter`（*https://github.com/Microsoft/TypeScript-Node-Starter*）快速為你產生檔案夾結構。

習題

既然環境已經設置好了，請在你的程式碼編輯器中開啟 *src/index.ts*，並輸入下列程式碼：

```
let a = 1 + 2
let b = a + 3
let c = {
apple: a,
banana: b
}
let d = c.apple * 4
```

現在用滑鼠游標在 a、b、c、d 上方移動，注意到 TypeScript 為你推論出了這些變數的型別：a 是一個 number，b 是一個 number，c 是具有特殊形狀的一個物件，而 d 也是一個 number（圖 2-3）。

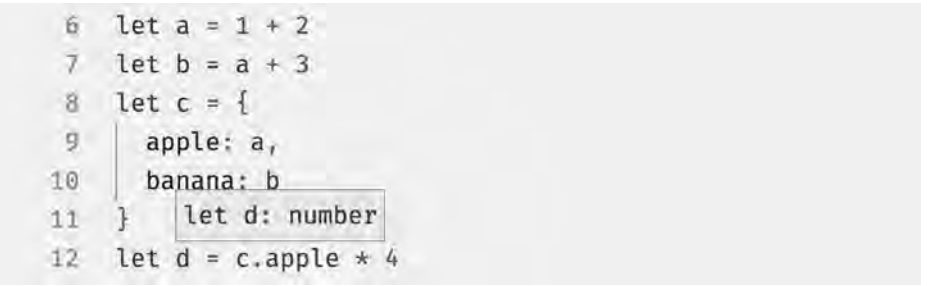

圖 2-3 TypeScript 為你推論出型別

把玩一下你的程式碼，看看你是否能夠：

- 做出一些無效的事，讓 TypeScript 為你呈現出紅色的扭曲標示（我們稱這為「擲出一個 TypeError」）。
- 閱讀那個 TypeError，試著理解它的意義。
- 修正那個 TypeError，看看紅色的扭曲標示是否消失。

若你富有野心，試試看能不能寫出一段 TypeScript 無法推論出型別的程式碼。