對本書的讚譽

《*Effective TypeScript*》討論了 *TypeScript* 開發者常見的問題，並提出實用的、以結果為導向的建議。無論你有多少 *TypeScript* 使用經驗，都可以從本書學到新知。

—*Ryan Cavanaugh*，微軟 *TypeScript* 工程主管

《*Effective TypeScript*》第二版為一本已經相當出色的著作加入更多精彩內容。本書包含第一版的絕佳技巧，以及近年來逐漸鞏固的 *TypeScript* 實務建議。無論專案規模多大，只要你打算使用 *TypeScript*，本書的知識都不可或缺。我特別欣賞書中的許多建議皆詳細地解釋它們的使用時機與理由。個人建議打算精通 *TypeScript* 的開發者認真地閱讀本書兩次，並好好地做筆記。

—*Josh Goldberg*，開源開發者、
《*TypeScript* 學習手冊》作者

《*Effective TypeScript*》除了介紹全球最受歡迎的程式語言之外，也透過切合實際的建議，教你如何從型態的角度來思考。你將在每一頁體驗到 *Dan* 的豐富經驗。這本書有很大的野心！它甚至可以改變你對於 *TypeScript* 之外的程式設計的看法。

— *Stefan Baumgartner*，*Dynatrace* 資深產品架構師，
《*TypeScript Cookbook*》作者

對於已經熟悉 *TypeScript* 基礎知識、每天使用 *TypeScript* 來開發，或正在考慮採用 *TypeScript* 的工程師而言，《*Effective TypeScript*》第二版是絕佳的參考書籍。

— *Boris Cherny*，《*TypeScript* 程式設計》作者

在使用語言的同時閱讀範例顯然有絕佳的效果。雖然當時我已經會用 C++ 了，卻第一次感覺它用起來如此自在，並且知道如何考慮它提供的各種選項。後來，當我閱讀 Joshua Bloch 所著的《*Effective Java*》（Addison-Wesley Professional）與 David Herman 所著的《*Effective JavaScript*》（Addison-Wesley Professional）時，也有類似的體驗。

如果你已經習慣使用幾種不同的程式語言，那麼直接深入研究一個新語言的冷門功能可以挑戰你的心智模型，讓你瞭解它有什麼不同之處。在撰寫本書的過程中，我也學到許多關於 TypeScript 的知識。希望你閱讀本書時，也有相同的體驗！

本書結構

本書是由許多「項目（item）」組成的，每一個項目都是一篇簡短的技術文章，針對 TypeScript 的特定層面提出具體的建議。這些項目依照主題分為各個章節，但你可以隨意選擇感興趣的章節跳著閱讀。

每一個項目的標題皆傳達該項目的核心要點，它們都是使用 TypeScript 時應銘記在心的要訣，建議你先快速瀏覽目錄，將它們烙印在你的腦海中。例如，如果你在撰寫文字說明時，糾結著要不要寫入型態資訊，你會知道應該翻到第 31 項：不要在註釋中重複敘述型態資訊。

該項目的內容會進一步說明標題提出的建議，並以具體的例子和技術論述來支持該建議。本書的觀點幾乎都是用範例程式來展示的。當我閱讀技術書籍時，經常會大略看一下文字內容，再仔細研究範例程式，我也預設你將採取類似的做法，希望你閱讀這些文字與解釋！但快速瀏覽範例仍然可以從中掌握要點。

當你閱讀整個項目之後，你應該可以瞭解為何它可協助你更有效地使用 TypeScript。如果它不適合在你所面臨的情況中使用，你也有足夠的知識認知此事。《*Effective C++*》的作者 Scott Meyers 舉了一個令人印象深刻的例子來說明這一點。他曾經和一個飛彈軟體設計團隊開會，該團隊知道他們可以忽略他提出的建議：「不要洩漏原始碼」，因為他們的程式一定會在飛彈擊中目標、硬體被炸碎時終止。我不知道飛彈裡有沒有 JavaScript 執行環境，但 James Webb 太空望遠鏡裡面有，天知道我們會遇到什麼情況！

最後，在每一個項目的結尾都有一個總結該項目重點的「牢記要點」。如果你只想大略瀏覽，可以閱讀這些要點來瞭解該項目的內容，以及確認是否需要仔細地閱讀。建議你閱讀項目的內容！但那些要點可以幫你應急。

TypeScript 的型態系統

TypeScript 可生成程式碼（第 3 項），但是型態系統才是重頭戲，它正是你使用這種語言的原因！

本章將介紹 TypeScript 型態系統的基本元素：如何看待它、如何使用它、你要做出的選擇，以及應該避免使用的功能。TypeScript 的型態系統強大得令人意外，它能夠表達一些你原本以為型態系統無法表達的東西。本章將為你打下堅實的基礎，幫助你在編寫 TypeScript 和閱讀本書其餘內容時更加得心應手。

第 6 項：使用編輯器來訊問和探索型態系統

在安裝 TypeScript 之後，你會得到兩個可執行檔：

- `tsc`，TypeScript 編譯器
- `tsserver`，TypeScript 獨立伺服器

通常你會直接執行 TypeScript 編譯器，但是伺服器也很重要，因為它提供了語言服務，這些服務包括自動補全、檢查、導覽，以及重構。你通常會在編輯器中使用這些服務，如果沒有設置這些功能就虧大了！自動補全之類的服務就是讓 TypeScript 如此好用的因素之一。這些功能除了方便之外，也讓編輯器成為學習與驗證型態系統的最佳工具。它可以協助你瞭解 TypeScript 何時能夠推斷型態，這正是寫出紮實的程式碼、讓它符合語言風格的關鍵（見第 18 項）。

每一款編輯器的操作細節可能各有不同，但你通常可以將游標移到某個代號上，看看 TypeScript 認為它是哪一種型態（圖 2-1）。

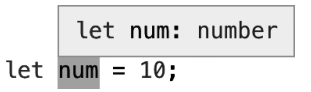

圖 2-1　這個編輯器（VS Code）顯示 num 代號的型態被推斷為 number

你在此並未撰寫 number，但 TypeScript 能夠根據「10」這個值認出它。

你也可以檢查函式，見圖 2-2。

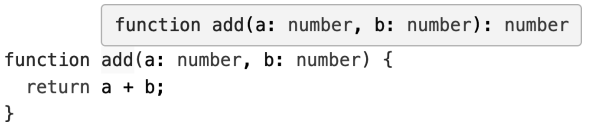

圖 2-2　使用編輯器來顯示被推斷出來的函式回傳型態

其中，特別值得注意的資訊是回傳型態的推斷值（number），如果它和你想的不一樣，你就要加入型態宣告，並找出不一樣的原因（見第 9 項）。

即時查看 TypeScript 推斷出來的變數能夠幫助你直接理解「放寬（widening）」（見第 20 項）與「窄化（narrowing）」（見第 22 項）的機制，觀察一個變數的型態在條件分支中如何變化可以讓你更加信任型態系統（圖 2-3）。

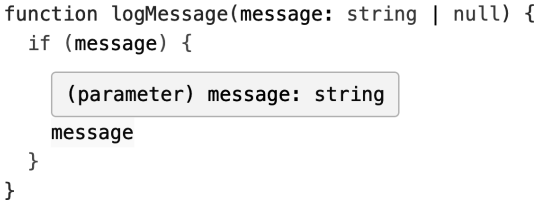

圖 2-3　message 的型態在分支外面是 string | null，但是在裡面是 string

你可以查看更大物件的各個屬性來瞭解 TypeScript 推斷它們是什麼（見圖 2-4）。

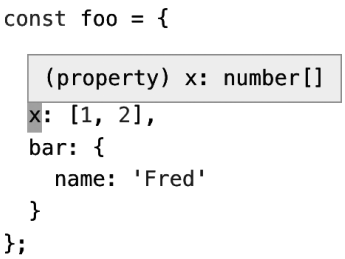

圖 2-4　查看 TypeScript 如何推斷物件內的型態

如果你想讓 x 的型態是 tuple（`[number, number, number]`），那就要使用型態註記（type annotation）。

若要在一條操作鏈（a chain of operations）的中間查看推斷出來的泛型型態，可檢查方法名稱（圖 2-5）。

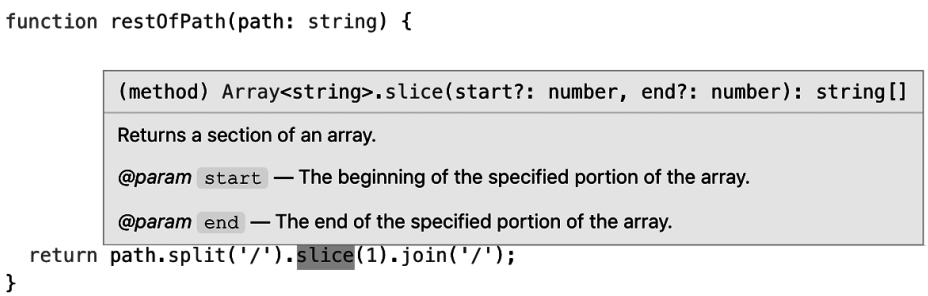

圖 2-5　在方法呼叫鏈之中顯示推斷出來的泛型型態

`Array<string>` 意味著 TypeScript 知道 `split` 產生一個字串陣列。雖然這個例子幾乎沒有什麼模稜兩可之處，但這類資訊在撰寫和偵錯長串的函式呼叫時可能非常關鍵。TypeScript 也顯示 `slice` 方法的文字說明。第 68 項會解釋這是怎麼運作的。

在編輯器中查看型態錯誤也是學習型態系統細節的好策略。例如，這個函式試著使用 `HTMLElement` 的 ID 來取得 `HTMLElement`，或回傳預設的元素，TypeScript 會指出兩項錯誤：

```
function getElement(elOrId: string | HTMLElement | null): HTMLElement {
  if (typeof elOrId === 'object') {
    return elOrId;
    // ~~~ Type 'HTMLElement | null' is not assignable to type 'HTMLElement'
```

```
  } else if (elOrId === null) {
    return document.body;
  }
  elOrId
  // ^? (parameter) elOrId: string
  return document.getElementById(elOrId);
  // ~~~ Type 'HTMLElement | null' is not assignable to type 'HTMLElement'
}
```

第一個 `if` 陳述式分支的目的是篩選出一個物件，即 `HTMLElement`。但奇怪的是，在 JavaScript 中，`typeof null` 的結果是 `"object"`，所以在該分支裡，`elOrId` 也可能是 `null`，你可以把 `null` 檢查寫在最前面來修正這個問題。第二個錯誤是因為 `document.getElementById` 可能回傳 `null`，因此你也要處理這種情況，例如丟出一個例外：

```
function getElement(elOrId: string|HTMLElement|null): HTMLElement {
  if (elOrId === null) {
    return document.body;
  } else if (typeof elOrId === 'object') {
    return elOrId;
    //     ^? (parameter) elOrId: HTMLElement
  }
  const el = document.getElementById(elOrId);
  //                                 ^? (parameter) elOrId: string
  if (!el) {
    throw new Error(`No such element ${elOrId}`);
  }
  return el;
  //     ^? const el: HTMLElement
}
```

TypeScript 的語言服務也提供重構工具。其中一個最簡單但非常實用的功能就是重新命名代號（renaming a symbol）。這項功能比「尋找與取代」更複雜，因為相同的名稱在不同位置可能代表不同的變數。例如，在這段程式碼中，有三個不同的變數都叫做 `i`：

```
let i = 0;
for (let i = 0; i < 10; i++) {
  console.log(i);
  {
    let i = 12;
    console.log(i);
  }
}
console.log(i);
```

在 VS Code 裡，當你點選 `for` 迴圈中的 `i`，然後按下 F2 時，會出現一個文字輸入框，讓你輸入新名稱（圖 2-6）。

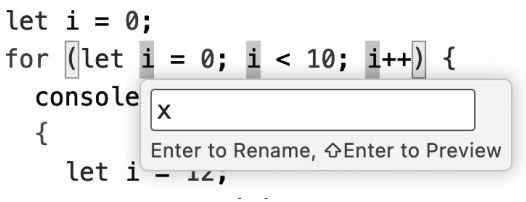

圖 2-6　在編輯器中重新命名一個代號

當你執行這個重構時，只有被你重新命名的那一個 `i` 的參考會被改掉：

```
let i = 0;
for (let x = 0; x < 10; x++) {
  console.log(x);
  {
    let i = 12;
    console.log(i);
  }
}
console.log(i);
```

如果你重新命名一個從其他模組匯入的代號，那些匯入的代號也會被一併更新。此外還有許多其他實用的重構操作，例如重新命名或移動檔案（會自動更新所有匯入的代號），或是將代號移到新檔案中。你也要熟悉這些功能，因為在處理大型 TypeScript 專案時，它們可以大幅提升開發效率。

語言服務也能夠幫助你在自己的程式碼、外部程式庫以及型態宣告之間快速巡覽。假設你看到一段程式碼呼叫了全域的 `fetch` 函式，想要瞭解它的細節，編輯器應該有個「Go to Definition」選項，在我的編輯器中，它長得像圖 2-7 這樣。

圖 2-7　TypeScript 語言服務提供「Go to Definition」功能，你應該也可以在你的編輯器中找到它

選擇這個選項會進入 *lib.dom.d.ts*，也就是被 TypeScript 加入的 DOM 型態宣告檔案：

```
declare function fetch(
  input: RequestInfo | URL, init?: RequestInit
): Promise<Response>;
```

你可以看到 `fetch` 接收兩個引數，並回傳一個 `Promise`。按下 `RequestInfo` 之後，你會看到：

```
type RequestInfo = Request | string;
```

你可以從這裡前往 `Request`：

```
interface Request extends Body {
  // ...
}
declare var Request: {
  prototype: Request;
  new(input: RequestInfo | URL, init?: RequestInit | undefined): Request;
};
```

從這裡可以看到，`Request` 的型態與值是分開定義的（見第 8 項）。你已經看過 `RequestInfo` 了，按下 `RequestInit` 會看到你在建立 `Request` 時可用的所有選項：

```
interface RequestInit {
  body?: BodyInit | null;
  cache?: RequestCache;
  credentials?: RequestCredentials;
  headers?: HeadersInit;
  // ...
}
```

你也可以在這裡查看許多其他型態，你已經知道怎麼做了。型態宣告檔在第一次閱讀時不太容易理解，但它們可以讓你瞭解 TypeScript 能夠做到什麼、你使用的程式庫是怎麼建模的（modeled），以及你可能要如何處理錯誤。第 8 章會進一步說明型態宣告。

牢記要點

- 善用支援 TypeScript 語言服務的編輯器。
- 使用編輯器來瞭解型態系統如何運作，以及 TypeScript 如何推斷型態。
- 熟悉 TypeScript 的重構工具，例如重新命名代號與檔案。
- 瞭解如何跳到型態宣告檔案，並在裡面查看它們如何表達行為。

第六章

泛型與型態級設計

TypeScript 型態系統的設計目的，是為了表達 JavaScript 程式碼的執行期行為。由於 JavaScript 是如此動態且寬鬆，這促使 TypeScript 的型態系統開發出日益強大的功能。如第 15 項所述，這包含在型態之間互相轉換的邏輯。

當你加入泛型型態別名時，TypeScript 的型態系統將非常強大，甚至可以視為一門獨立的程式語言（TypeScript 的型態系統是圖靈完備的（Turing Complete）（*https://oreil.ly/snlm0*），所以嚴格來說確實如此）。你不再是用值來撰寫程式（像使用 JavaScript 時那樣），而是用型態來撰寫程式。換句話說，這是「型態級設計」。型態級設計與 metaprogramming（即設計操作程式的程式）是不一樣的概念，儘管這兩個術語有時被混為一談。

學習新語言是有趣的事情，外界有一些使用 TypeScript 型態系統來打造的許多瘋狂應用，包括遊戲和 SQL 解析器，這波熱潮部分來自 Type Challenges（*https://tsch.js.org*）專案，這個專案提供了數百個難度逐漸提高的型態系統謎題。一邊閱讀這一章，一邊回答這些題目是很好的實戰練習，可幫助你鞏固學到的知識。

本章也會提供一些提醒。就算 TypeScript 提供了一種用來處理型態的程式語言，那也不代表這是一種特別直覺、人性化或好用的語言。你可以在型態層面上撰寫邏輯，也不代表這一定是好的做法。濫用泛型可能寫出難以理解、難以維護的程式。Josh Goldberg 在其著作《*Learning TypeScript*》（O'Reilly）中精闢地指出：

> 儘管使用泛型能夠非常靈活地在程式碼中定義型態，但它們很快就會變得複雜。剛接觸 *TypeScript* 的程式設計師往往會經歷一段過度使用泛型的時期，他們的程式碼會變得難以閱讀、難以操作。*TypeScript* 的最佳用法是在必要時才使用泛型，而且在使用時，應清楚地說明泛型的用途。

第 54 項：使用模板字面型態來表達 DSL 與字串之間的關係

第 35 項曾經建議在自己的程式碼中使用比 `string` 型態更精確的型態。但世界上有太多字串，幾乎無法完全避開。在遇到它們時，TypeScript 提供一項獨特的工具來定義字串的模式與關係：模板字面型態。本項目將探討這項功能的運作方式，以及如何利用它，讓那些原本無法定型的程式碼具備型態安全性。

就像所有程式語言一樣，TypeScript 有一個 `string` 型態，但如同前幾個項目所展示的，它也有字串字面型態，這種型態的值域只有一個字串值，它們經常以聯集的形式出現：

```
type MedalColor = 'gold' | 'silver' | 'bronze';
```

你可以用字串字面型態的聯集來表達有限的字串集合。`string` 本身涵蓋所有可能的字串組合（即無限集合）。模板字面型態可讓你表達介於兩者之間的情況，例如以 `pseudo` 開頭的所有字串：

```
type PseudoString = `pseudo${string}`;
const science: PseudoString = 'pseudoscience';  // ok
const alias: PseudoString = 'pseudonym';  // ok
const physics: PseudoString = 'physics';
//    ~~~~~~~ Type '"physics"' is not assignable to type '`pseudo${string}`'.
```

和 `string` 一樣，`PseudoString` 具備有限的值域（第 7 項）。但與 `string` 不同的是，`PseudoString` 型態的值具有某種結構：它們都以 `pseudo` 開頭。就像其他型態級結構一樣，模板字面型態的語法故意模仿 JavaScript 的模板字面型態（template literals）。

在 JavaScript 裡，結構化字串比比皆是。例如，如果你想讓某個物件有一些既定的屬性，但也允許它有 `data-` 開頭的任何其他屬性呢？（這個模式在 DOM 中很常見）

```
interface Checkbox {
  id: string;
  checked: boolean;
  [key: `data-${string}`]: unknown;
}

const check1: Checkbox = {
  id: 'subscribe',
  checked: true,
  value: 'yes',
// ~~~~ Object literal may only specify known properties,
//        and 'value' does not exist in type 'Checkbox'.
  'data-listIds': 'all-the-lists',  // ok
```

```
};
const check2: Checkbox = {
  id: 'subscribe',
  checked: true,
  listIds: 'all-the-lists',
// ~~~~~~ Object literal may only specify known properties,
//          and 'listIds' does not exist in type 'Checkbox'
};
```

如果使用 `string` 作為索引型態，我們會失去 `check1` 上的額外屬性檢查（見第 11 項），並錯誤地允許 `check2` 中有一個未以 `data-` 作為前綴的屬性：

```
interface Checkbox {
  id: string;
  checked: boolean;
  [key: string]: unknown;
}

const check1: Checkbox = {
  id: 'subscribe',
  checked: true,
  value: 'yes',  // 允許
  'data-listIds': 'all-the-lists',
};
const check2: Checkbox = {
  id: 'subscribe',
  checked: true,
  listIds: 'all-the-lists'  // 也允許，符合索引型態
};
```

模板字面型態很適合用來表達 `string` 的子集合，但它們真正好用之處在於，當你將它們與泛型和型態推斷結合時，能捕捉不同型態之間的關係。

我們以 DOM 所提供的 `querySelector` 函式為例。TypeScript 很聰明，當你查詢選擇器時，它可以提供更具體的 `HTMLElement` 子型態：

```
const img = document.querySelector('img');
//    ^? const img: HTMLImageElement | null
```

這可讓你存取 `img.src`，例如，較不具體的 `Element` 型態就不允許這樣的操作（第 75 項會討論 TypeScript 與 DOM）。

第 74 項：瞭解如何在執行期重建型態

在學習 TypeScript 的過程中，大多數開發者遲早會突然「開竅」，意識到 TypeScript 的型態不是「真的」，在執行期，它們會被移除（見第 3 項）。當他們發現這件事的時候，可能會有不安的情緒：如果說型態不是真的，那又憑什麼信任它們？

「型態」與「執行期行為」互相獨立，是 TypeScript 與 JavaScript 之間的關係之中的關鍵部分（第 1 項）。在大多數情況下，這套系統都可以正確運作。但不可否認，在執行期如果可以取得 TypeScript 型態，有時會非常方便。本項目將探討為什麼會有這種情境，以及你有哪些選擇。

假設你正在實作一個 web 伺服器，並定義了一個 API 端點，可以在部落格文章下新增留言（我們在第 42 項看過這個 API）。你為請求主體定義了一個 TypeScript 型態：

```
interface CreateComment {
  postId: string;
  title: string;
  body: string;
}
```

你的請求處理程式必須驗證請求。部分的驗證程序屬於應用級驗證（例如：`postId` 是否指向一篇存在的貼文，以及使用者能不能留言？），也有一部分是型態級驗證（例如：這個請求是否具有我們預期的所有屬性？屬性的型態是否正確？是否有不該出現的額外屬性？）。

以下可能的寫法之一：

```
app.post('/comment', (request, response) => {
  const {body} = request;
  if (
    !body ||
    typeof body !== 'object' ||
    Object.keys(body).length !== 3 ||
    !('postId' in body) || typeof body.postId !== 'string' ||
    !('title' in body) || typeof body.title !== 'string' ||
    !('body' in body) || typeof body.body !== 'string'
  ) {
    return response.status(400).send('Invalid request');
  }
  const comment = body as CreateComment;
  // ... 應用級驗證與邏輯 ...
  return response.status(200).send('ok');
});
```

即使屬性只有三個，這段驗證程式卻不短。更糟的是，沒有東西可以確保這些檢查是準確的，並且和我們的型態保持一致。沒有任何檢查機制能確保我們正確地拼寫這些屬性名稱，如果我們新增一個屬性，我們也要記得補上檢查邏輯。

這可以說是最糟的程式碼重複，我們有兩份資訊（型態與驗證邏輯）需要保持同步。如果有單一真相來源（single source of truth）就好了。`interface` 似乎是最自然的真相來源，但它在執行期會消失，你不知道該如何在執行期使用它。

我們來看看幾種可能的解決辦法。

從其他來源生成型態

如果你的 API 是以其他形式來定義的，也許使用 GraphQL 或 OpenAPI schema 來定義，那麼你可以將它們當成真相來源，並使用它們來生成 TypeScript 型態。

這通常需要執行一種外部工具來生成型態，可能還要一併生成驗證程式碼。例如，OpenAPI 規格使用 JSON Schema，因此你可以使用 `json-schema-to-typescript` 之類的工具來生成 TypeScript 型態，再搭配 `Ajv` 之類的 JSON Schema 驗證器來驗證請求資料。

這種做法的缺點是，它會增加一些複雜度，並且需要額外的組建步驟，每次 API schema 變動時，都必須重新執行。但如果你原本就透過 OpenAPI 或其他系統來定義 API，這種做法有很大的優點：它不會引入新的真相來源，因此應該是你的首選。

如果這種方式適合你的情況，第 42 項有一個從 schema 生成 TypeScript 型態的範例可供參考。

使用執行期程式庫來定義型態

TypeScript 的設計讓我們無法從靜態型態推導出執行期的值，但反方向的推導（從執行期的值推導出靜態型態）可以透過型態級的 `typeof` 運算子來做到：

```
const val = { postId: '123', title: 'First', body: 'That is all'};
type ValType = typeof val;
//   ^? type ValType = { postId: string; title: string; body: string; }
```

因此，有一種做法是使用執行期結構來定義型態，然後從它們推導出靜態型態。這通常是用程式庫來實現的。有許多程式庫可以做到，目前最受歡迎的是 Zod（React 的 `PropTypes` 是另一個例子）。

第十章

現代化與遷移

你聽說 TypeScript 很棒，你也從痛苦的經驗中知道，維護你 15 年前寫出來的那一套擁有十萬行程式碼的 JavaScript 程式庫一點都不好玩。如果能把它變成 TypeScript 程式庫就好了！

本章將提供一些建議，幫助你在保持理智的情況下，持之以恆地將 JavaScript 專案遷移到 TypeScript。

程式碼越少，遷移起來就越容易。所以在開始進行 TypeScript 遷移之前，先移除已淘汰的功能，並且做一輪死碼清除是個好主意。不過，你可能要將其他形式的現代化暫停，因為等你正式採用 TypeScript 之後，將 jQuery web app 轉換成 React 其實會容易得多。

只有極小型的程式才能一次全部遷移完成，遷移大型專案的關鍵是逐步遷移，第 81 項會討論怎麼做。如果遷移時間較長，追蹤進度並避免回到原點非常重要，這會營造出一種變革正在持續推進、不可逆轉的氣氛。第 82 項會討論如何做到。

將大型專案遷移至 TypeScript 不一定很輕鬆，但它的潛在效益很龐大。在 2017 年的一項研究發現，在 GitHub 上的 JavaScript 專案中，有 15% 的 bug 原本可以用 TypeScript 來預防[1]。更驚人的是，Airbnb 在他們進行的一項為期六個月的事後調查報告中，發現有 38% 的問題是可以透過 TypeScript 來預防的[2]。如果你正在說服你的組織採用 TypeScript，這類數據非常有幫助！進行一些實驗並找到早期採用者也很重要。第 80 項會討論如何在開始遷移之前，進行 TypeScript 的試驗。

1 Z. Gao, C. Bird, and E. T. Barr, "To Type or Not to Type:Quantifying Detectable Bugs in JavaScript" (*https://oreil.ly/4RFfI*), ICSE 2017.

2 Brie Bunge, "Adopting TypeScript at Scale" (*https://oreil.ly/i-L60*), JSConf Hawaii 2019.

由於本章主要討論 JavaScript，有許多範例程式都是純 JavaScript（而且不一定能夠通過型態檢查），或是使用較寬鬆的設定來檢查的（例如關閉 `noImplicitAny`）。本章偶爾會使用作者的 dygraphs 圖表程式庫作為需要進行現代化與遷移的舊程式範例。dygraphs 是歷史悠久的 JavaScript 程式庫，最活躍的開發時期是在 2009 到 2016 年之間。

第 79 項：撰寫現代的 JavaScript

除了檢查你的程式碼是否型態安全之外，TypeScript 還可以將程式碼編譯成任何版本的 JavaScript，甚至可以回溯到 2009 年的 ES5。TypeScript 是最新 JavaScript 的超集合，這意味著你可以將 `tsc` 當作一種「轉譯器（transpiler）」，可以將最新版的 JavaScript 轉換成較舊、被更廣泛地支援的 JavaScript。

從另一個角度來看，這也意味著，當你決定將既有的 JavaScript 程式碼庫轉換成 TypeScript 時，採用所有最新的 JavaScript 功能是萬無一失的。事實上，這樣做好處多多：因為 TypeScript 是為了與現代 JavaScript 配合而設計的，所以將你的 JS 現代化是採用 TypeScript 的絕佳起點。

而且，由於 TypeScript 是 JavaScript 的超集合，學會撰寫更現代、更符合語言習慣的 JavaScript，相當於學會撰寫更好的 TypeScript。

本項目將介紹一些現代 JavaScript 的重點，這裡的「現代」是指從 ES2015（也就是 ES6）之後的功能。這些主題在其他書籍與網路資源中都有更詳盡的說明。如果你不熟悉接下來提到的某些主題，那你應該花時間去深入學習。學會 `async/await` 這樣的新功能之後，你可以從 TypeScript 獲得很大的好處：它應該比你更瞭解這些功能，並能夠引導你正確地使用它們。

這些功能都值得深入瞭解，但在採用 TypeScript 的過程中，最重要的工具莫過於 ECMAScript 模組與 ES2015 類別了。我們會先介紹這兩者，再快速列出其他幾個重點功能。如果你的專案已經在使用這些功能了，你應該感到慶幸！你的遷移過程將會輕鬆許多。

使用 ECMAScript 模組

在 ECMAScript（ES）2015 年版問世之前，將程式碼拆成不同模組沒有標準的做法可選擇。當時有很多解決辦法，例如使用多個 `<script>` 標籤、手動串接、使用 Makefile、Node.js 風格的 `require` 陳述式，或 AMD 風格的 `define` callback。甚至 TypeScript 也有它自己的模組系統（第 72 項）。