前言

自從 10 多年前我編寫了第一版《JavaScrip 設計模式學習手冊》以來，JavaScript 的世界已經取得了長足的進步。那時，我正在開發大型 web 應用程式，並發現 JavaScript 程式碼所缺乏的結構和組織，使得維護和擴展這些應用程式變得十分困難。

快轉到今天，web 開發格局發生了翻天覆地的變化。JavaScript 已成為世界上最流行的程式設計語言之一，可用來開發從簡單的腳本到複雜的 web 應用程式等所有事物上。JavaScript 語言已經發展到包含模組、promise 和 `async/await`，而這顯著影響我們建構應用程式的方式。開發人員編寫元件的方式，例如使用 React，也顯著影響他們對可維護性（maintainability）的看法，讓人不得不考慮這些新變化的現代模式問世。

隨著 React、Vue 和 Angular 等現代程式庫和框架的興起，開發人員現在正在建構比以往任何時刻都更加複雜的應用程式。我認知到必須更新《JavaScrip 設計模式學習手冊》，以反映 JavaScript 和 web 應用程式開發中的變化。

在《JavaScrip 設計模式學習手冊》第二版中，我旨在幫助開發人員將現代設計模式應用到他們的 JavaScript 程式碼和 React 應用程式中。本書涵蓋建構可維護和可擴展的應用程式所必備的 20 多種設計模式（design pattern）。且本書不僅涉及設計模式，還涉及渲染（rendering）和效能（performance）模式，它們對現代 web 應用程式的成功也至關重要。

本書第一版側重於經典的設計模式，例如模組（Module）模式、觀察者（Observer）模式和中介者（Mediator）模式。這些模式在今天仍然很重要也相關，但 web 開發世界在過去十年中發生重大變化，並且出現新的模式。二版涵蓋這些新模式，例如 promise、

`async/await` 和模組模式的新變化，也會介紹 MVC、MVP 和 MVVM 等架構模式，並討論現代框架在哪些方面適合這些架構模式。

現今的開發人員接觸到許多特定於程式庫或特定於框架的設計模式。React 成熟的生態系統以及利用較新的 JS 原語（primitive），對於在框架或程式庫的上下文中討論最佳實務和模式是不錯的開始。除了經典的設計模式外，本書還涵蓋現代 React 模式，例如 Hook、Higher-Order Component 和 Render Prop，這些模式都是 React 特有的，對於使用這個流行框架來建構現代 web 應用程式至關重要。

這本書不只是關於模式的；它還與最佳實務有關。我們涵蓋了程式碼組織、效能和渲染等主題，這些主題對於建構高品質的 web 應用程式也一樣重要。您將瞭解動態匯入（dynamic import）、程式碼拆分（code-splitting）、伺服器端渲染（server-side rendering）、水合（hydration）和 Islands 架構，所有這些對於建構快速且響應性的 web 應用程式都是必不可少的。

到本書結束時，您將深入瞭解設計模式，以及如何將它們應用到您的 JavaScript 程式碼和 React 應用程式中；您還將知道哪些模式會和現代網路相關，哪些又不相關。本書不只是模式的參考，它還是建構高品質 web 應用程式的指南。您將學習如何建構具有最大可維護性和可擴展性的程式碼，以及如何優化程式碼以提高效能。

本書架構

本書分為 15 章，目標是結合已更新的語言特性和特定於 React 的模式，從現代角度帶您瞭解 JavaScript 設計模式；每一章都建立在前一章的基礎上，讓您能夠逐步增長知識，並有效地應用：

- 第 1 章，〈設計模式簡介〉：瞭解設計模式的歷史，以及它們在程式設計世界中的意義。
- 第 2 章，〈「模式」性測試、原型模式和三法則〉：瞭解評估和改進設計模式的過程。
- 第 3 章，〈建構和編寫模式〉：學習對編寫良好模式的剖析，以及建立模式的方法。
- 第 4 章，〈反模式〉：瞭解何謂反模式？以及如何在您的程式碼中避免它們。
- 第 5 章，〈現代 JavaScript 語法和特性〉：探索最新的 JavaScript 語言特性，及其對設計模式的影響。

- 第 6 章，〈設計模式的分類〉：深入研究設計模式的不同類別：建立型、結構型和行為型。
- 第 7 章，〈JavaScript 設計模式〉：研究 JavaScript 的 20 多種經典設計模式，及其現代的改編版本。
- 第 8 章，〈JavaScript MV* 模式〉：瞭解 MVC、MVP 和 MVVM 等架構模式，及其在現代 web 開發中的重要性。
- 第 9 章，〈非同步程式設計模式〉：瞭解 JavaScript 中非同步程式設計的強大功能，以及處理非同步程式設計的各種模式。
- 第 10 章，〈模組式 JavaScript 設計模式〉：發現組織和模組化您的 JavaScript 程式碼的模式。
- 第 11 章，〈命名空間化模式〉：學習各種命名空間化您的 JavaScript 程式碼，以避免全域名稱空間污染的技術。
- 第 12 章，〈React.js 設計模式〉：探索特定於 React 的模式，包括 Higher-Order Component、Render Prop 和 Hook。
- 第 13 章，〈渲染模式〉：瞭解不同的渲染技術，例如客戶端渲染、伺服器端渲染、漸進式水合和 Islands 架構。
- 第 14 章，〈React.js 的應用程式結構〉：瞭解如何建構 React 應用程式，以實現更好的組織、可維護性和可擴展性。
- 第 15 章，〈結論〉：總結本書的要點和最終想法。

這整本書都提供實際範例來說明所討論的模式和概念。在旅程結束時，您將對 JavaScript 設計模式有深入的瞭解，並有能力編寫優雅、可維護和可擴展的程式碼。

無論您是經驗豐富還是剛剛起步的 web 開發人員，本書都將為您提供建構現代、可維護和可擴展的 web 應用程式所需的知識和工具。我希望這本書會成為您繼續發展技能，和建構令人驚嘆的 web 應用程式的寶貴資源。

第 4 章

反模式

工程師都可能遇到這樣的情況：在截止日期前交付解決方案，或者程式碼在沒有程式碼審查的情況下被包含在一系列補丁內。這種情況下的程式碼可能並不總是經過深思熟慮的，並且可能會傳播所謂的**反模式**（*anti-pattern*）。本章將介紹反模式，以及必須理解和識別它們的原因，並且將研究 JavaScript 中的一些典型反模式。

什麼是反模式？

如果模式代表最佳實務，反模式就代表提出的模式出錯時所吸取的教訓。受到 GoF 著作《*Design Patterns*》的啟發，Andrew Koenig 在 1995 年發表於 *Journal of Object-Oriented Programming* 第 8 卷的文章[1]中，首次創造出反模式一詞。他將反模式描述為：

> 反模式就像模式一樣，但它不是解決方案，它呈現的是表面上看起來像解決方案，但實際上不是解決方案的東西。

他提出反模式的兩個概念：

- 描述某個會導致不利情況發生的特定問題**劣質**解決方案
- 描述擺脫上述情況並尋求良好解決方案的**方式**

1 *https:// oreil.ly/Megyr*

說到此，Alexander 也提到在良好設計結構和良好上下文之間，取得良好平衡的困難：

> 這些筆記是關於設計過程的；發明實體事物的過程，這些事物顯示出新的實體秩序、組織、形式，來回應功能。……每個設計問題都始於努力達成兩個實體之間的適合性：所討論的形式及其上下文。形式是解決問題的辦法，上下文則定義了問題。

瞭解反模式與瞭解設計模式一樣重要，分析背後的原因可知，建立應用程式時，專案的生命週期就從建構開始。在這個階段，您可能會從可用的良好設計模式中選擇您認為合適的；但是在初始發布之後，還需要維護它。

維護已投入生產的應用程式可能特別具有挑戰性。以前沒有開發過該應用程式的開發人員可能會不小心將糟糕的設計引入專案中，如果這些不良實務已經識別為反模式，開發人員就能夠提前認出它們，並避免已知的常見錯誤。這類似於應用設計模式知識，來識別出那些可以應用已知且有用標準技術的領域。

隨著解決方案的發展，其品質可能是好是壞，這取決於團隊在其中投入的技能水準和時間。這裡的好壞以上下文而定，如果應用在錯誤的上下文時，再怎麼「完美」的設計，也可能認定為反模式。

總而言之，反模式是一種應該要記錄下來的糟糕設計。

JavaScript 中的反模式

開發人員有時會故意選擇捷徑和臨時解決方案，以加快程式碼交付腳步，這些東西往往會成為永久性，且基本上是由反模式組成的技術債務而累積起來。JavaScript 是一種弱型別或無型別語言，因此可以更輕鬆地採用某些捷徑，以下是您可能在 JavaScript 中遇到的一些反模式範例：

- 透過在全域上下文中定義大量變數，來污染全域命名空間。
- 將字串而不是函數傳遞給 `setTimeout` 或 `setInterval`，因為這會在內部觸發 `eval()` 的使用。
- 修改 `Object` 類別原型（這是一個極為糟糕的反模式）。
- 以內聯（inline）形式使用 JavaScript，因為這是沒有彈性的。

- 在使用原生文件物件模型（Document Object Model, DOM）替代方案，如 `document.createElement` 更合適的地方，使用 `document.write`。`document.write` 多年來一直遭到嚴重濫用，並且有很多缺點。如果它在頁面載入後執行，會覆寫所在的頁面，這讓 `document.createElement` 成為比較好的選擇，可存取以下連結來獲得實際操作範例[2]。它也不適用於 XHTML，這是選擇對 DOM 更友善方法，例如 `document.createElement` 的另一個原因。

瞭解反模式是成功的關鍵。一旦學會識別這種反模式，就可以重構程式碼來否定它們，這樣能立即提高解決方案的整體品質。

總結

本章介紹了可能衍生出的反模式問題模式和 JavaScript 反模式範例。在詳細介紹 JavaScript 設計模式之前，必須觸及一些關鍵的現代 JavaScript 概念，因為這些概念與對模式的討論息息相關。這就是下一章的主題，介紹現代 JavaScript 的特性和語法。

2 *https://oreil.ly/kc1c0*

第 5 章

現代 JavaScript 語法和特性

JavaScript 已經存在了幾十年，並且經歷多次修訂。本書探索現代 JavaScript 上下文中的設計模式，並在所有討論的範例中使用現代 ES2015+ 語法；之所以在本章討論 ES2015+ JavaScript 特性和語法，是因為這對進一步討論當前 JavaScript 上下文中的設計模式來說至關重要。

ES2015 對 JavaScript 語法引入一些根本性的變化，與關於模式的討論密切相關。BabelJS ES2015 指南[1] 詳細介紹了這些內容。

本書依賴現代 JavaScript 語法。TypeScript 可能讓您好奇不已，它是 JavaScript 的靜態型別超集合，提供 JavaScript 所沒有的多種語言特性，包括強型別、介面、列舉（enum）和進階型別推理（type inference），而且也會影響設計模式。要瞭解更多 TypeScript 及其優勢的資訊，可以查看 O'Reilly 書籍，例如 Boris Cherny 所撰寫的《*TypeScript 程式設計*》（*Programming TypeScript*）。

解耦應用程式的重要性

模組化 JavaScript 允許您在邏輯上把應用程式分成更小的部分，稱為模組。一個模組可以被其他模組匯入，而這些模組反過來可以被更多的模組匯入。因此，應用程式可以由許多巢套模組所組成。

1 *https://oreil.ly/V09r_*

在可擴展的 JavaScript 世界中，若說一個應用程式是**模組化**（*modular*），通常指它是由一組高度解耦、不同的功能模組組成。鬆散耦合（loose coupling）有助於透過在可能的情況下消除依賴關係，從而更輕鬆地維護應用程式。如果有效地實作，可以看出它對系統這個部分的更改，會如何影響另一個部分。

和一些更為傳統的程式設計語言不同，在 ES5（標準 ECMA-262 5.1 版[2]）之前，JavaScript 的舊版本並未為開發人員提供乾淨地組織和匯入程式碼模組的方法。直到最近幾年，隨著對更有組織的 JavaScript 應用程式需求日益明顯，它才成為規範的關注點之一。非同步模組定義（Asynchronous Module Definition, AMD）[3] 和 CommonJS[4] 模組是 JavaScript 初始版本中，最流行的解耦應用程式模式。

這些問題的原生解決方案隨著 ES6 或 ES2015[5] 一起出現。負責定義 ECMAScript 及其未來版本語法語意演變的標準機構 TC39[6]，一直在密切關注 JavaScript 在大規模開發中的使用狀況演變，並敏銳地意識到，需要用更好的語言特性，才能編寫更模組化的 JS。

隨著 ES2015 中 ECMAScript 模組的發布，在 JavaScript 建立模組的語法也已開發並標準化。今天，所有主流瀏覽器都支援 JavaScript 模組。它們已經成為在 JavaScript 中實作現代模組化程式設計的實際方法。本節也將使用 ES2015+ 中的模組語法，來探索程式碼範例。

具有匯入和匯出功能的模組

模組可以讓應用程式的程式碼分成獨立單元，每個單元包含功能的一個層面的程式碼。模組也鼓勵程式碼的可重用性，並揭露可以整合到不同應用程式中的功能。

一種語言應該具有這項功能：允許您 `import` 模組依賴項並 `export` 模組介面（允許其他模組使用的公共 API／變數），以支援模組化程式設計。ES2015 將 JavaScript 模組（即

2 *https://oreil.ly/w2WxN*

3 *https://oreil.ly/W5XPd*

4 *https://oreil.ly/lgw0w*

5 *https://oreil.ly/rPxFL*

6 *https://oreil.ly/GJduA*

ES 模組）[7] 的支援導入至 JavaScript 中，允許您使用 `import` 關鍵字來指明模組依賴項。同樣的，您也可以使用 `export` 關鍵字，從模組中匯出任何內容：

- `import` 宣告將模組的匯出綁定為區域變數，並且可以重新命名以避免名稱重複／衝突。
- `export` 宣告會宣告模組的區域綁定是外部可見的，這樣其他模組可以讀取匯出但不能修改它們。有趣的是，模組可以匯出子模組，但不能匯出已在別處定義的模組；也可以重新命名匯出，好讓它們的外部名稱不同於區域名稱。

.mjs 是用於 JavaScript 模組的副檔名，可以幫助區分模組檔案和經典腳本（*.js*）。*.mjs* 副檔名確保相對應的檔案執行時期（runtime）和建構工具，例如 Node.js[8]、Babel[9] 能將其解析為模組。

以下範例顯示麵包店員工的三個模組、他們在烘焙時執行的功能、以及麵包店本身，可以看出一個模組要如何匯入和使用另一個模組匯出的功能：

```
// 檔名：staff.mjs
// ========================================
// 指明其他模組可以使用的（公開）匯出
export const baker = {
   bake(item) {
      console.log( `Woo! I just baked ${item}` );
    }
};

// 檔名：cakeFactory.mjs
// ========================================
// 指明依賴項
import baker from "/modules/staff.mjs";

export const oven = {
    makeCupcake(toppings) {
       baker.bake( "cupcake", toppings );
    },
    makeMuffin(mSize) {
        baker.bake( "muffin", size );
```

7 *https://oreil.ly/kd-pu*

8 *https://oreil.ly/E9oRS*

9 *https://oreil.ly/fkQAL*

```
    }
}

// 檔名：bakery.mjs
// ========================================
import {cakeFactory} from "/modules/cakeFactory.mjs";
cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

通常，一個模組檔案會包含幾個相關的函數、常數和變數，您可以使用單一匯出敘述，在檔案末尾集中式的匯出這些內容，後面以逗號分隔要匯出的模組資源串列：

```
// 檔名：staff.mjs
// ========================================
const baker = {
  // 烘焙師函數
};
const pastryChef = {
  // 糕點師函數
};
const assistant = {
  // 助手函數
};

export { baker, pastryChef, assistant };
```

同樣的，您可以只匯入需要的函數：

```
import {baker, assistant} from "/modules/staff.mjs";
```

也可以透過指明值為 module 的 type 屬性，來要求瀏覽器接受包含 JavaScript 模組的 <script> 標記：

```
<script type= module src= main.mjs ></script>
<script nomodule src= fallback.js ></script>
```

nomodule 屬性告訴現代瀏覽器不要將經典腳本作為模組載入，這對不使用模組語法的後饋（fallback）腳本很有用。它允許您在 HTML 中使用模組語法，並讓它在不支援它的瀏覽器中工作，就很多層面例如效能來說，這都很有用。現代瀏覽器不需要 polyfilling 來實現現代功能，就能允許您單獨為舊版瀏覽器提供更大的轉譯程式碼。

模組物件

有一種更簡潔的方法可以匯入和使用模組資源，就是將模組作為物件來匯入，這會將所有的匯出內容變成該物件的成員。：

```
// 檔名：cakeFactory.mjs

import * as Staff from "/modules/staff.mjs";

export const oven = {
    makeCupcake(toppings) {
       Staff.baker.bake( "cupcake", toppings );
    },
    makePastry(mSize) {
        Staff.pastryChef.make( "pastry", type );
    }
}
```

從遠端來源載入的模組

ES2015+ 還支援遠端模組，例如第三方程式庫，這讓它要從外部位置載入模組相形之下更為簡單。下面是一個帶入之前定義的模組並使用的範例：

```
import {cakeFactory} from "https://example.com/modules/cakeFactory.mjs";
// 急切載入的靜態匯入

cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

靜態匯入

剛剛討論的匯入類型稱為靜態匯入（static import）。在主要程式碼執行之前，需要使用靜態匯入以下載並執行模組圖，這有時會導致初始頁面在載入時，需要預先載入大量程式碼，而耗費不少時間，並且延遲關鍵功能的啟用時間：

```
import {cakeFactory} from "/modules/cakeFactory.mjs";
// 急切載入的靜態匯入

cakeFactory.oven.makeCupcake( "sprinkles" );
cakeFactory.oven.makeMuffin( "large" );
```

動態匯入

有時，您不想預先載入模組，而是在需要時再依需求載入。惰性載入（lazy-loading）模組允許您在需要時才載入所需的內容，例如使用者點擊連結或按鈕時，這能提高初始載入時的效能；引入動態匯入（dynamic import）[10] 則讓此事更為可行。

動態匯入引入一種全新、類似於函數的匯入形式。`import(url)` 傳回對請求模組的模組命名空間物件承諾，而該請求模組在獲取、實例化和評估所有模組依賴項以及模組本身之後創建。以下是 `cakeFactory` 模組的動態匯入範例：

```
form.addEventListener("submit", e => {
  e.preventDefault();
  import("/modules/cakeFactory.js")
    .then((module) => {
      // 用模組做些事。
      module.oven.makeCupcake("sprinkles");
      module.oven.makeMuffin("large");
    });
});
```

還可以使用 `await` 關鍵字來支援動態匯入：

```
let module = await import("/modules/cakeFactory.js");
```

透過動態匯入，只會在使用模組時才下載和評估模組圖。

流行模式，如互動匯入（Import on Interaction）和可見性匯入（Import on Visibility），可以使用動態匯入功能於普通 JavaScript 中輕鬆實作。

互動匯入

有些程式庫可能只在使用者開始與網頁上特定功能互動時才發揮作用，最典型的例子就是聊天小部件（widget）、複雜對話框或視訊嵌入。這些功能的程式庫不需要在頁面載入時匯入，但可以在使用者與它們互動時，用諸如點擊元件外觀或占位符等方式載入。這樣的操作會觸發相應程式庫的動態匯入，進而呼叫函數來啟動所需功能。

10 *https://oreil.ly/fqR6v*

例如，可以使用動態載入的外部 `lodash.sortby` 模組[11]，來實作螢幕排序功能：

```
const btn = document.querySelector('button');

btn.addEventListener('click', e => {
  e.preventDefault();
  import('lodash.sortby')
    .then(module => module.default)
    .then(sortInput()) // 使用匯入的依賴項
    .catch(err => { console.log(err) });
});
```

可見性匯入

許多元件在初始頁面載入時不可見，但使用者向下捲動時就能見到。由於使用者不一定會一直向下捲動，因此可以在看見這些元件所對應的模組時，再進行惰性載入。IntersectionObserver API[12] 可以偵測元件占位符的可見時機，此時動態匯入就可以載入相對應的模組。

伺服器模組

Node[13] 15.3.0 以上版本支援 JavaScript 模組，它們在沒有實驗旗標的情況下執行，並且與 npm 套件生態系統的其餘部分相容。Node[14] 把以 *.mjs* 和 *.js* 結尾且最上層型別欄位值為模組（module）的檔案，視為 JavaScript 模組：

```
{
  "name": "js-modules",
  "version": "1.0.0",
  "description": "A package using JS Modules",
  "main": "index.js",
  "type": "module",
  "author": "",
  "license": "MIT"
}
```

11 *https://oreil.ly/VUgnM*

12 *https:// oreil.ly/wXwgi*

13 *https://oreil.ly/4Bh_O*

14 *https://oreil.ly/q1Jzl*

使用模組的優勢

模組化程式設計和使用模組能提供幾個獨特優勢，例如以下：

只評估一次模組腳本

經典腳本每次添加到 DOM 時都需要評估，但瀏覽器只會評估模組腳本一次。這意味著對於 JS 模組而言，如果有一個依賴模組的擴展階層，就會先評估依賴於最內層模組的模組，這是一件好事，因為這意味著最裡面的模組會第一個評估，並且可以存取依賴於這個模組的匯出。

模組可自動延遲

與其他腳本檔案不同，如果您不想立即載入這些腳本，則必須包含 `defer` 屬性，但瀏覽器會自動延遲模組的載入。

模組易於維護和重用

模組促進程式碼的解耦，這些程式碼可以獨立維護，而無需大幅度更改其他模組；而且允許您在多個不同的函數中重用相同程式碼。

模組提供命名空間

模組為相關變數和常數建立一個私有空間，以便可以透過模組來引用它們，而不會污染全域命名空間。

模組可以消除死程式碼

在引入模組之前，必須手動從專案中刪除未使用的程式碼檔案。透過匯入模組，webpack[15] 和 Rollup[16] 等捆包器（bundler）可以自動識別未使用的模組並刪除它，所以死程式碼在添加到捆包之前，就會遭到刪除，稱為 tree-shaking。

所有現代瀏覽器都支援模組匯入[17]和匯出[18]，您可以在沒有任何後饋的情況下使用它們。

15 *https:// oreil.ly/37e9F*

16 *https://oreil.ly/rUWiB*

17 *https://oreil.ly/IauTK*

18 *https://oreil.ly/NubAY*

具有建構子、getter 和 setter 的類別

除了模組之外，ES2015+ 還允許使用建構子（constructor）和一些隱私感來定義類別。JavaScript 類別使用 class 關鍵字定義，下面的範例定義了一個 Cake 類別，它有一個建構子和兩個 getter 和 setter：

```
class Cake{

    // 可以使用關鍵字 constructor 以及
    // 一串類別變數來定義
    // 類別建構子的本體。
    constructor( name, toppings, price, cakeSize ){
        this.name = name;
        this.cakeSize = cakeSize;
        this.toppings = toppings;
        this.price = price;
    }

    // 作為一部分 ES2015+ 對減少不必要的功能使用的努力，
    // 您會注意到它在以下情況遭到刪除。
    // 這裡一個識別符後面跟著一個參數串列和一個主體定義了一個新方法。

    addTopping( topping ){
        this.toppings.push( topping );
    }

    // 可以透過在識別符／方法名稱和大括號所包圍的主體之前宣告 get 來定義 getter。
    get allToppings(){
        return this.toppings;
    }

    get qualifiesForDiscount(){
        return this.price > 5;
    }

    // 與 getter 類似，setter 可以透過在識別符前使用 set 關鍵字來定義
    set size( size ){
        if ( size < 0){
            throw new Error( "Cake must be a valid size: " +
                                "either small, medium or large");
        }
       this.cakeSize = size;
    }
}

// 用法
let cake = new Cake( "chocolate", ["chocolate chips"], 5, "large" );
```

建立在原型之上的 JavaScript 類別，是一種特殊的 JavaScript 函數，需要先定義它們才能參照。

您還可以使用 extends 關鍵字，來指出一個類別是繼承自另一個類別：

```
class BirthdayCake extends Cake {
  surprise() {
    console.log(`Happy Birthday!`);
  }
}

let birthdayCake = new BirthdayCake( "chocolate", ["chocolate chips"], 5,
  "large" );
birthdayCake.surprise();
```

所有現代瀏覽器和 Node 都支援 ES2015 類別，它們還與 ES6 中導入的新型類別語法[19]相容。

JavaScript 模組和類別之間的區別在於，模組是匯入的[20]和匯出的[21]，而類別可使用 class 關鍵字定義。

讀過一遍之後，您可能還會注意到前面的範例缺少「function」一詞。這不是打字錯誤：TC39 已經有意識地努力減少對 function 這個關鍵字的濫用，希望它能協助簡化編寫程式碼的方式。

JavaScript 類別還支援 super 關鍵字，它允許您呼叫父類別的建構子[22]，這對實作自我繼承模式很有用。您可以使用 super 來呼叫超類別的方法：

```
class Cookie {
  constructor(flavor) {
    this.flavor = flavor;
  }

  showTitle() {
    console.log(`The flavor of this cookie is ${this.flavor}.`);
  }
}
```

19 *https:// oreil.ly/9c9jm*

20 *https://oreil.ly/IauTK*

21 *https://oreil.ly/NubAY*

22 *https://oreil.ly/gYxxw*

```
class FavoriteCookie extends Cookie {
  showTitle() {
    super.showTitle();
    console.log(`${this.flavor} is amazing.`);
  }
}

let myCookie = new FavoriteCookie('chocolate');
myCookie.showTitle();
// 這塊餅乾是巧克力口味的。
// 巧克力太棒了。
```

現代 JavaScript 支援公共和私有類別成員，其他類別可以存取公共類別成員，私有類別成員只能由定義它們的類別存取。預設情況下，類別欄位是公共的，私有類別欄位[23] 可以使用 # 字首來建立：

```
class CookieWithPrivateField {
  #privateField;
}

class CookieWithPrivateMethod {
  #privateMethod() {
    return 'delicious cookies';
  }
}
```

JavaScript 類別使用 `static` 關鍵字來支援靜態方法和屬性，可以在不實例化類別的情況下參照靜態成員。您可以使用靜態方法來建立實用程序函數，並使用靜態屬性來保存配置或快取資料：

```
class Cookie {
  constructor(flavor) {
    this.flavor = flavor;
  }
  static brandName = "Best Bakes";
  static discountPercent = 5;
}
console.log(Cookie.brandName); // 輸出 = "Best Bakes"
```

23 *https:// oreil.ly/SXsES*

JavaScript 框架中的類別

在過去幾年裡，一些現代 JavaScript 程式庫和框架，尤其是 React 導入了類別的替代品，React Hook 可以在沒有 ES2015 類別元件的情況下，使用 React 狀態和生命週期方法。在 Hook 之前，React 開發人員必須將功能元件重構為類別元件，以便處理狀態和生命週期方法；這樣通常很棘手，因為需要瞭解 ES2015 類別的工作原理。React Hook 是允許您在不依賴類別的情況下，管理元件的狀態和生命週期方法的函數。

請注意，其他幾種為 web 建構的方法，例如 Web Components[24] 社群，持續使用類別作為元件開發的基礎。

總結

本章介紹模組和類別的 JavaScript 語言語法，這些特性能讓人在編寫程式碼的同時，堅持物件導向的設計和模組化程式設計原則，還能夠使用這些概念，來分類和描述不同設計模式；下一章就將討論這些不同類別的設計模式。

相關閱讀

- v8 上的 JavaScript 模組：*https://oreil.ly/IEuAq*
- MDN 上的 JavaScript 模組：*https://oreil.ly/OAL9O*

24 *https://oreil.ly/ndfeb*