前言

即使印刷技術已經問世許久了，出版書籍仍然是有挑戰性的工作。是的，通常會有一位（或一群）作者隨時隨地撰寫內容，但也會有一位內容編輯者負責協助作者，將他們的想法轉換成不致於太枯燥並富有吸引力的故事——如果書籍與技術或商業有關，更要特別小心。我們還有技術校閱，他們是高警覺性的專家，負責找出嚴重的技術定義或解釋的錯誤。當然，最後還有文字編輯，他們是確保文字和語法正確的最後一道防線。但是到目前為止，我們只談到皮毛而已：上述的所有人員大致上都處理書本的內容，但是製作書籍也有其他的面向。例如排版人員，他們的工作是確保書籍在付梓時有良好的外觀——妥善處理參差不齊的程式碼等等。有些人負責設計封面，也有人負責審核初稿的目錄，這樣才能提供合約給作者。此外，有些人要負責監督書籍的出版過程，通常稱為“生管”。印好書籍之後，還要配送它們。最後，書籍才會被放到書架上（實體或其他的），並開始販售，直到終於有人買了這本書並開始閱讀它。整個購買與交易的過程甚至可以寫成一本書。

這個過程複雜得讓人難以置信，然而，對參與其中的每一個人而言，事情並沒有那麼複雜。例如，作者只要每天寫幾百個字就可以了，這複雜嗎？如此劃分這個程序是有原因的：我們不擅長處理高度複雜的事項。對參與出版書籍這種龐大專案（或商務企業）的每一個人來說，將它拆成各項單一的責任（例如“編寫內容”、“改善文字排列”、“校閱技術問題”、“修正文法錯誤”、“排版”，或“處理販售程序”）可讓他們更容易工作。

出版書籍只是一個例子——我們可以對幾乎任何事項做同樣的事情。從你的桌上拿起一個東西，任何東西都行，想一下它是怎麼來的，接著把目光拉遠，想一下：它是怎麼做的？它的材料是什麼？有多少人製作每一個組件、將它們組裝起來、讓它看起來是完美的，再將它送到你購買它的店裡面？那個東西是水果嗎？有多少人種植它、消滅害蟲、修剪枝葉、包裝它，再把它送到商店？

軟體沒有太大的不同，我們的身邊充斥著各種複雜的事物。把目標拉到最近，你可以發現以物理常數描述的限制，例如光速、各個位元，以及硬體、中斷呼叫、組合語言等等。把目標拉遠，我們可以發現技術領域的大型結構，負責處理搜尋查詢指令與付款處理等事項。我們這些開發者，以及我們負責的專案就身處這些複雜事項之中。

我們不太可能停下來思考每日看到的物件與互動的底層的複雜性，因為這樣做會造成癱瘓。相反，我們會將解決方案藏在抽象介面之後，將它們（在我們的心中）**化成**介面。有些介面可良好地對映抽象實作，讓人覺得好用，有些則無法良好地對映實作，讓人覺得困惑和挫折。軟體沒有什麼不同，我們不想考慮整個系統，幾乎所有我們用到的事物都被隱藏在比底層的實作更容易使用與理解的介面之後。

誰該閱讀這本書？

本書適合具備 JavaScript 與 ES6 應用知識的開發者、業餘愛好者和專業人士[1]。想要知道如何寫出易理解、易維護與可擴展程式的開發者（即使不使用 JavaScript 語言）也可以從中獲益。

為何將 JavaScript 模組化？

最初，我只想玩玩 Node.js，在不知不覺的情況下，我與 JavaScript 建立緊密的關係，在此同時，我發現開放原始碼，並愛上它的做法。Node.js 周圍的 open source（開放原始碼）生態系統讓來自 C# 這種

1 ES6 深深地影響 JavaScript 語言的變化，它加入了多種語法改善以及一些新方法。本書假設你已經熟悉 JavaScript ES6 之後的版本了。你可以到 Pony Foo blog（*https://mjavascript.com/out/es6*）閱讀速成教學來深入瞭解 ES6 語法。

closed-source（封閉原始碼）領域的我大開眼界，並且熱切地釐清“如何編寫穩健的程式，讓別人可以愉快地使用”。在這種背景之下，我發現自己總是在思考如何定義介面、誰該使用它、他們會不會花時間做別的事情，搞不清楚我們原本希望他們做的事情。

本書期望以循循善誘的方式讓你成為成功的模組作者。寫出 JavaScript 模組不難，但是採取合理的做法來提供簡單、靈活，而且在大部分的情況下都很容易使用（但是在必要時也很靈活），同時又能控制內部複雜性的模組並不簡單。我曾經在《*JavaScript Application Design*》[2] 以及 Pony Foo 部落格之中斷斷續續地寫過一些關於正確的應用程式設計的見解，但我一直渴望出版一本完全討論模組化程式的設計與編寫的書籍。

雖然我還沒有看到任何一本從 JavaScript 的角度來討論這個主題的書籍，但你可以找到與模組化程式這個主題有關的其他書籍，例如 Steve McConnell 的《*Code Complete*》（Microsoft Press） 與 Robert C. Martin 的《*Clean Code*》（Prentice Hall），並且在你的 JavaScript 開發工作中採取他們的做法。本書試著將你的注意力從別人認為你該做的事情上移開，讓你自行決定你應該做什麼，以及為何如此——而不是強加一堆規則，因為這只會導致人們自行宣稱他們的程式是“簡潔的”。

本書試著以不刻意造作的方式來解釋如何編寫模組化程式。我們會試著闡明模組化架構背後的原理，以及它在 JavaScript 的歷史，讓你更容易瞭解編寫模組化程式的意義以及好處。

雖然坊間已經有許多說明如何設計正確的應用程式設計的書籍了，但是探討模組化應用程式設計的書籍並不多，更不用說模組化 JavaScript 應用程式設計了，所以你才會看到這本書。雖然本書中大多數的建議、理念與教學都不是 JavaScript 專屬的，但是把注意力放在 JavaScript 上，意味著你不但會學到如何編寫模組化的網路應用程式，在過程中也會記得一些將網路變成獨特的平台以及讓 JavaScript 在許多方面都如此特別的奇特功能。

2 《*JavaScript Application Design*》（*https://mjavascript.com/out/jad*）是 Manning 在 2015 年為我出版的書籍。它的內容與組建程序有關，但也有一些章節討論複雜性的管理、理想的非同步流程控制程式、REST API 設計，與 JavaScript 的測試問題。

希望你將本書的內容用在你試著處理的問題上，並且評估各種做法的優劣得失，以產生自己的見解，而不是依賴長篇大論、深入的分析以及具體的例子。軟體沒有一體適用的解決方案，你通常要活用自己的判斷力來決定如何編寫它。所有的軟體都得適應它周圍的環境，如何你做過任何涉及軟體部署或發表的工作，肯定知道將同一段程式塞到不同的執行環境裡面有多麼困難。

這本書的目標與《*Practical Modern JavaScript*》一樣，也是慢慢地建立一條基線。當我們從《*Practical Modern JavaScript*》學到最新的語言功能之後，要從這本書瞭解模組化設計思維。你可以在這兩本書的各章與各節中看到這種漸進式與模組化的做法。

本書架構

第 1 章討論模組化在 JavaScript 中的演進，從早期在 `onclick` 屬性中嵌入 JavaScript，到 CommonJS，以及最後的原生 ECMAScript 模組。接著說明編寫獨立且完善的程式的好處，以及在系統的每一個層面這樣做的好處，這些層面包括服務、應用程式、元件、模組、函式、區塊等等。

第 2 章討論模組化設計的基本要求，幫你打下一個基礎，讓你在這個基礎上編寫具備良好 API 的模組，並且知道它會被如何使用（在所有可能的情況下）、責任的歸屬，以及哪些屬於介面。

第 3 章大部分的內容都是關於讓你瞭解你應該解決的問題類型、如何在解決那些問題的同時密切關注模組與介面的演變，以及在等待清晰的模式浮現時盡量不要做抽象化。這一章會深入討論關於文件化、錯誤處理的最佳做法，以及根據你自己的理解，對想要解決的問題採取你自己的做法。

第 4 章會輕鬆地討論內部複雜性、緊密耦合，並評估框架與規範的優點。這一章主要討論重構程式來減少複雜性的各種做法。接著我們會討論狀態與複雜性之間的關係，以及如何減輕複雜性。在此資料結構也扮演重要的角色，因為在控制複雜性時，選擇正確的資料結構雖然有挑戰性，卻可帶來巨大的回報。

第 5 章專門討論 JavaScript，詳細介紹如何利用現代的語言結構來編寫簡潔的程式。這一章也會探討繼承與組合之類的模式，進而討論如何根據你的使用案例來選擇適當的選項。本章的最後也會討論經典的模式，例如顯示模組、物件工廠、事件發射器與 JSON 訊息傳遞。

第 6 章介紹身經百戰的模組開發者是如何思考的，探討安全問題與依賴關係管理、建立及整合程序、介面和抽象，一般來說，這是個關於模組設計的建議和最佳做法的大雜燴。

如果你已經熟悉與 JavaScript 有關的模組化歷史，至少可以瀏覽一下第 1 章的歷史課。如果你喜歡跳著閱讀各個章節，我仍然建議你閱讀每一章，因為這本薄書比較類似講述合理程序的故事書，而不是簡單的配方食譜。

本書編排慣例

本書使用以下的編排規則：

斜體字（*Italic*）

代表新的術語、URL、電子郵件地址、檔案名稱及副檔名。中文以楷體表示。

定寬字（`Constant width`）

代表程式，也在文章中代表程式元素，例如變數或函式名稱、資料庫、資料類型、環境變數、陳述式與關鍵字。

這個圖示代表一般注意事項。

致謝

本書得以完成需要感謝許多人。首先是負責本書與 O'Reilly 的 Modular JavaScript 系列的主編 Virginia Wilson，她提供了寶貴的經驗，當我進度開始落後、寫作的速度像涓涓細流般緩慢時，她非常體諒我，始終保持非常積極的態度！

第一章

模組思維

如前言所言，當我們進行軟體專案時，複雜性似乎總是圍繞四周。抽象也是如此，它將複雜性藏在我們不敢碰觸的石頭底下，那些石頭是另一個世界的介面，讓我們可以遠離它、幾乎不會想到它。JavaScript 也不例外，儘管動態語言很強大，但是當我們使用它們時，往往更容易（甚至嘗試）寫出複雜的程式。

我們接下來會先討論如何在工作中更妥善地應用抽象、介面與它們底層的概念，如此一來，當我們執行專案、處理某項功能時，就可以將需要注意的複雜性最小化，直到將它變成單一功能的分支。

1.1 模組思維簡介

擁有**模組思維**就是認知複雜性是不可避免的，並且用介面來消除複雜性，以免再次看到它或想到它。但是你必須妥善地設計介面（這是棘手的部分），以免讓它的使用者感受挫折，這種挫折甚至會讓人忍不住窺探引擎蓋下的世界，當他們看到比令人氣餒的介面複雜得多的內部程式之後，或許會發現，如果沒有那個介面的話，程式搞不好還比較容易維護與閱讀。

系統可以顆粒化（granular）：我們可以將它們分成專案、用許多應用程式構成系統、在系統裡面設置一些應用程式等級的階層，在每一個階層內使用上百個模組，用上千個函式組成這些模組等等。顆粒化可以協助我們把相當的注意力放在模組化上，同時保持理智，協助我們寫出容易瞭解與維護的程式。在第 11 頁的 1.4 節“模組顆粒化”中，我們會討論如何執行顆粒化來建立模組化的應用程式。

當我們描述元件時，都會提供一個讓系統的其他部分操作的公開介面。這個介面（或 API）裡面有元件公開的方法或屬性，那些方法或屬性也可以稱為**接觸點**，也就是可在介面中公開互動的東西。介面的接觸點越少，表面積就越小，介面就越簡單。表面積大的介面有高度的彈性，但是因為這種介面公開大量的功能，所以很可能難以理解與使用。

介面有兩種用途，它可以讓我們開發元件的新功能，只公開已經準備好了、可供大家使用的功能，同時保留不想讓其他元件使用的私有內容。同時，它可讓使用者（使用介面的元件或系統）受惠於公開的功能，而不需要考慮那項功能的細節究竟如何實作。

要隔離複雜的程式，讓別人在使用它的時候不需要知道任何實作細節，最好的做法之一就是寫出強健、文件化的介面。有系統地安排強健的介面可以逐漸形成一個階層，例如企業應用程式的服務或資料層。採取這種做法時，我們在很大程度上可將邏輯隔離與限制在其中一層，同時將處理表象的程式與嚴格的商業或持久保存相關的程式分開。這種強而有力的隔離是高效的做法，因為它可讓各個元件保持整齊，並且讓階層保持一致。從開發者的角度來看，一致的階層（以模式或外觀相似的元件組成的）可提供熟悉感，讓人能夠持續直觀地使用它，並且隨著時間的流逝更加熟悉 API 的外貌。

由於設計合適的介面是不容易的事情，使用一致的 API 外貌是增加生產力的好方法。當我們持續使用類似的 API 外貌，就不需要每次都重新擬定新的設計，使用者也可以放心地相信你沒有每次都重新發明輪子。我們會在接下來的章節更詳細地討論 API 設計。

1.2 模組化簡史

在 JavaScript 中，模組化是個現代的概念。本節要快速地回顧並總結 JavaScript 世界的模組化演進里程碑。這一節只簡介主流做法在 JavaScript 歷史中的演化，不會詳盡地說明它們。

1.2.1 腳本標籤與 Closure

在早期，JavaScript 是被嵌在 HTML `<script>` 裡面的，它頂多會被放到專屬的腳本檔案裡面，全都共用一個全域範圍。

在這些檔案或行內腳本內宣告的任何變數或綁定都會被印到全域的 `window` 物件，因而在彼此無關的腳本之間造成資訊洩漏，這可能會導致衝突甚至破壞體驗，因為在一個腳本裡面的變數可能會不小心取代另一個腳本使用的全域變數：

```
<script>
  var initialized = false

  if (!initialized) {
    init()
  }

  function init() {
    initialized = true
    console.log('init')
  }
</script>

<script>
  if (initialized) {
    console.log('was initialized!')
  }

  // 就連 `init` 都默默地變成全域變數
  console.log('init' in window)
</script>
```

隨著網路應用程式開始變大與變複雜，限定範圍的概念與全域範圍的危險越來越明顯並開始受到關注。於是 Immediately Invoked Function Expressions（IIFE）問世了，並且立刻成為主流。IIFE 的做法是將整個檔案或部分的檔案包在一個函式裡面，在求值（evaluation）之後立刻執行。JavaScript 的每一個函式都會建立一層新的範圍，也就是說，使用 `var` 變數的賦值會被包在 IIFE 裡面。雖然變數的宣告會被懸吊（hoist）在它們的範圍上面，但是拜 IIFE 包裝之賜，它們永遠不會變成隱晦的全域變數，因此可以降低隱晦的 JavaScript 全域變數造成的脆弱性。

下面的範例是一些 IIFE 的做法，這裡面的 IIFE 程式是各自獨立的，只能用明確的陳述式轉換成全域環境，例如 `window.fromIIFE = true`：

```
(function() {
  console.log('IIFE using parenthesis')
})()

~function() {
  console.log('IIFE using a bitwise operator')
}()

void function() {
  console.log('IIFE using the void operator')
}()
```

程式庫通常使用 IIFE 模式來公開再重複使用 `window` 物件的單一綁定，以盡量減少全域命令空間的汙染。下面的程式展示如何使用（採取 IIFE 模式的）程式庫裡面的 `sum` 方法來建立 `mathlib` 元件。如果我們想要在 `mathlib` 裡面加入更多模組，可以將它們分別放在一個 IIFE，將它們自己的方法放到 `mathlib` 公開介面，同時讓其他的程式對於定義新功能的元件來說都維持私用。

```
void function() {
  window.mathlib = window.mathlib || {}
  window.mathlib.sum = sum

  function sum(...values) {
    return values.reduce((a, b) => a + b, 0)
  }
}()

mathlib.sum(1, 2, 3)
// <- 6
```

碰巧這種模式也促使 JavaScript 工具快速成長，讓開發者（初次）將每一個 IIFE 模組都串連成一個檔案，這種做法可以減少網路的壓力，前提是當時的原始捆綁方法能夠在不破壞應用程式邏輯的情況下自動插入分號與縮小內容。

IIFE 方法的問題在於它沒有明確的依賴關係樹，開發者必須按照精確的順序來製作元件檔案清單，在使用依賴項目的模組被載入之前，先載入依賴項目（遞迴地）。

1.2.2 RequireJS、AngularJS 與依賴注入

自從 RequireJS 之類的模組系統與 AngularJS 的依賴注入機制出現以來，我們幾乎不用考慮這個問題，因為它們都可讓我們明確定義各個模組的依賴關係。

下面的範例展示如何使用 RequireJS 的 `define` 函式（已被加入全域範圍）來定義 *mathlib/sum.js* 程式庫。接下來 `define` 回呼的回傳值會被當成模組的公開介面使用：

```
define(function() {
  return sum

  function sum(...values) {
    return values.reduce((a, b) => a + b, 0)
  }
})
```

接下來我們可以用一個 *mathlib.js* 模組來收集想要納入程式庫的所有功能。在這個例子中，依賴項目只有 *mathlib/sum*，但是我們可以用同樣的方式列出任意數量的項目。我們可以在陣列裡面以依賴項目路徑來列出它們，並且從被傳入回呼的參數按相同的順序取得它們的公開介面：

```
define(['mathlib/sum'], function(sum) {
  return { sum }
})
```

我們定義程式庫之後，就可以用 `require` 來使用它了。注意這段程式是怎麼幫我們解析依賴關係鏈的：

```
require(['mathlib'], function(mathlib) {
  mathlib.sum(1, 2, 3)
  // <- 6
})
```

這是 RequireJS 與它的依賴樹的優點。無論我們的應用程式有沒有上百或上千個模組，RequireJS 都可以解析依賴樹，不會用到需要謹慎管理的清單。因為我們已經將依賴項目列在需要使用它們的地方了，所以不需要列出一長串的元件以及它們彼此的關係，因此不需要做“維護清單”這種容易出錯的工作。消除複雜性的大量來源只是附帶的好處而已，不是主要的好處。

在模組層級明確地宣告依賴關係可讓我們清楚地知道元件與應用程式的其他部分有什麼關係，明白這一點之後，我們可以繼續做更大規模的模組化，這在過往是很難有效率地做到的，因為依賴鏈很難追隨。

RequireJS 並非沒有任何問題，它的整個模式都與非同步載入模組有密切的關係，這對生產部署來說是不明智的做法，因為這種做法的執行效果很差。當我們使用非同步載入機制時，在多數的程式執行之前，要先以瀑流（waterfall）形式發出上百個網路請求，我們必須使用不同的工具來優化產品的組建，接著有一堆冗長的元素，最後你會得到一長串的依賴項目、RequireJS 函式呼叫式，以及讓你的模組使用的回呼，此外還有一些 RequireJS 函式與呼叫這些函式的方法，讓它們的使用更複雜。這種 API 不是最直觀的，此外還有許多方法可以做同樣的事情：用依賴項目來宣告模組。

AngularJS 的依賴注入系統也有許多同樣的問題。當時它是一種優雅的解決方案，可巧妙地解析字串來避免使用依賴項目陣列，並改用函式參數名稱來解析依賴關係。這種機制與 manifier 不相容，因為 minifier 會將參數的名稱改成單一字元，因而破壞注入機制。

AngularJS v1 在後期加入一種組建工作來將這種程式碼：

```
module.factory('calculator', function(mathlib) {
  // ...
})
```

轉換成下面的格式，因為它加入了明確的依賴項目清單，所以可以安全地簡化：

```
module.factory('calculator', ['mathlib', function(mathlib) {
  // ...
}])
```

因為這種默默無聞的組建工具太晚出現了，而且它要用額外的組建步驟來恢復不該被破壞的東西，讓大家不想要使用這種只帶來些微好處的模式。大部分的開發者依然使用他們熟悉的，類似 RequireJS 那種寫死依賴關係的陣列格式。

1.2.3 Node.js 與 CommonJS 的問世

在 Node.js 引發的諸多創新之中，出現一種 CommonJS 模組系統，簡稱 CJS。由於 CommonJS 活用“Node.js 程式可以存取檔案系統”這個事實，所以它更符合傳統的模組載入機制。在 CommonJS 中，每一個檔案都是一個模組，有它們自己的範圍與環境。它使用同步的 `require` 函式來載入依賴項目，這個函式可在模組的生命週期的任何時刻動態呼叫，例如這段程式：

```
const mathlib = require('./mathlib')
```

CommonJS 很像 RequireJS 與 AngularJS，都用路徑名稱來引用依賴關係。它們之間的主要差異在於 CommonJS 沒有 boilerplate[譯註]函式與依賴關係陣列了，而且你可以將模組的介面指派給變數，或在可使用 JavaScript 運算式的任何地方使用它。

譯註 在程式設計中，boilerplate code 或 boilerplate 指的是在許多地方重複出現，而且只被少量修改或完全相同的程式。通常它們被用來代表累贅的程式。boilerplate 可譯為“樣板”，但中文無法傳達這個意思，故直接引用原文。

與 RequireJS 或 AngularJS 不同的是，CommonJS 相當嚴格。在 RequireJS 與 AngularJS 裡面，每個檔案都可以有許多動態定義的模組，但是在 CommonJS 中，檔案與模組的關係是一對一的。同時，RequireJS 有許多種宣告模組的方式，AngularJS 則有許多種工廠、服務、供應器等等，而且 AngularJS 的依賴注入機制與這種框架本身緊密地耦合。相較之下，CommonJS 只有一種宣告模組的方式，任何 JavaScript 檔案都是一個模組，呼叫 `require` 會載入依賴項目，而且指派給 `module.exports` 的任何東西都是它的介面，因此它是更好的工具，也可以做程式自我檢查（introspection），更方便我們用來瞭解 CommonJS 元件系統階層。

最後，Browserify 的發明彌合了 Node.js 伺服器上的 CommonJS 模組與瀏覽器之間的鴻溝，只要使用 `browserify` 命令列介面程式並將入口模組的路徑傳給它，就可以將數量多到不可思議的模組結合成一個可供瀏覽器使用的包裹。CommonJS 的殺手級功能 npm 套件註冊表更在它企圖主宰模組載入生態系統時發揮決定性作用。

當然，並非只有 CommonJS 模組或 JavaScript 套件可以使用 npm，但是在當時與現在，CommonJS 模組都是 npm 的主要使用案例。使用 CommonJS 時，你只要按幾次滑鼠就可以在網路 app 中使用上千種套件（現在超過 50 萬種，而且還在穩定成長中），而且可以在 Node.js 伺服器與每個用戶端的瀏覽器上重複使用大部分的系統，這些優勢讓其他的系統難以望其項背。

1.2.4 ES6、import、Babel 與 Webpack

隨著 ES6 在 2015 年 6 月標準化，而且 Babel 早就可將 ES6 轉換成 ES5 了，我們很快迎來一場新的革命。ES6 規格加入了 JavaScript 原生的模組語法，通常稱為 ECMAScript 模組（ESM）。

ESM 在很大程度上受到 CJS 及其前身的影響，提供了一種靜態的宣告式 API，以及採用 promise 的動態可程式 API，如下所示：

```
import mathlib from './mathlib'
import('./mathlib').then(mathlib => {
  // ...
})
```

同樣的，在 ESM 中，每一個檔案都是一個模組，有它自己的範圍與環境。ESM 比 CJS 好的主要優勢之一在於 ESM 可以靜態匯入依賴項目（並且鼓勵使用）。靜態匯入大幅改善模組系統的自我檢查能力，因為模組系統可以被靜態分析，並且可用詞彙從系統的各個模組的抽象語法樹（AST）取出。ESM 的靜態匯入只限於模組的最頂層，可進一步簡化解析與自我檢查。ESM 比 CommonJS`require()` 好的另一個地方在於 ESM 有一種進行非同步模組載入的方法，也就是說，你可以視需求同時或惰性（lazy）載入應用程式的部分依賴關係圖來回應特定的事件。雖然在我寫這本書時，大多數的環境都還沒有實作這項功能，但有強烈的跡象指出 Node.js 將來應該會納入它[1]。

在 Node.js v8.5.0 中，如果 ESM 模組使用 *.mjs* 副檔名的話，你要在 `--experimental-modules` 旗標的後面加入它，大多數的長青瀏覽器都不需要旗標就可以支援 ESM 了。

Webpack 是 Browserify 的接班人，由於它具備廣泛的功能，所以在很大程度上繼承了通用模組包裝器的角色。如同 Babel 與 ES6，Webpack 長期以來一直藉由它的 `import` 與 `export` 靜態陳述式以及 `import()` 動態函式之類的運算式來支援 ESM。它採用 ESM 帶來的效益極高，這在很大程度上歸功於它採用“程式碼分割”的機制，因此可以將應用程式分割成不同的包裝，以提升初次載入體驗的效能[2]。

由於 ESM 是語言原生的工具（與 CJS 相較之下），我們預期它在接下來幾年會完全主宰模組生態系統。

1 你可以閱讀 Node.js 團隊成員 Myles Borins 著作的《The Current State of Implementation and Planning for ESModules》*https://mjavascript.com/out/esm-node*），來深入瞭解具體的細節。

2 程式碼分割（*https://mjavascript.com/out/code-splitting*）可根據不同的入口將應用程式拆成許多不同的包裝，也可以將各個包裝共用的依賴項目放到一個可重複使用的包裝裡面。