第一章

歡迎來到 React

什麼指標可以代表一個好的 JavaScript 函式庫？是 GitHub 上的星星數量嗎？是 npm 上的下載次數嗎？是推特上那些意見領袖的討論頻率嗎？我們究竟該如何選擇最佳的工具來建構專案？又要如何確認這些工具值得花時間研究？又要如何知道它能否勝任呢？

React 團隊的 Pete Hunt 寫了一篇文章〈*Why React*？〉——這篇文章希望讀者可以在批評 React 的設計理念太惡搞前，先給 React 五分鐘。

你擔心的沒錯，React 只是一個相對小型的函式庫，它可能沒有包含所有你需要的功能——但是你可以先給 React 五分鐘嗎？

你擔心的沒錯，在 React 中，你會在 JavaScript 裡寫出看起來像是 HTML 的標籤；你擔心的沒錯，那些 HTML 標籤必須經過預處理（Preprocessing）才可以在瀏覽器中運作；此外，你可能還需要一個像是 webpack 的建構工具來進行預處理——但是你可以先給 React 五分鐘嗎？

儘管你可能存在許多疑慮，但事實證明，隨著 React 問世將滿十年，許多團隊都一致承認它真的很棒——因為他們願意給 React 五分鐘。這些團隊包含了像是 Uber、Twitter 與 Airbnb 這種等級的軟體巨頭。在試用過後，他們相信 React 可以用更快的速度，創造出更好的產品。

這不就是我們在這裡討論 React 的原因嗎？不是因為 GitHub 上的星星或下載數。重點在於：我們想要使用自己喜愛的工具，創造出連自己都愛不釋手的產品，並驕傲地宣稱本人正是產品的開發者。

如果你有這樣的目標，你將很有可能愛上 React。

堅實的基礎

不論你是全然的 React 新手，或只是想要學習 React 最新的功能，我們希望這本書能為你建構堅實的 React 基礎。我們設計了一個有效的學習途徑，避免你自行摸索可能產生的困惑。

然而，在開始探討 React 前，你必須先瞭解 JavaScript——你不需要對所有語法都瞭若指掌，但至少要熟悉陣列、物件與函式的語法。

在下一個章節中，我們將會帶你熟悉 JavaScript 的最新語法與功能——尤其是那些 React 中經常使用的部分。接著，我們會為你介紹函式導向的 JavaScript——這是 React 賴以建構的基礎。在探索 React 的過程中，你也會逐漸領悟到如何透過更好的設計模式，來創造出高可讀性、可重用且可測試的程式碼，進而成為一個更厲害的 JavaScript 工程師。

接著，我們要探討如何使用 React 元件（Component）建構出使用者介面，並在其上架構邏輯、屬性（Property）與狀態（State）。我們還會介紹 React Hook——這將使我們可以重複使用元件中的狀態與邏輯。

在前述的基礎上，我們將創建一個調色應用，讓使用者可以新增、編輯以及刪除顏色，並透過 Hook 與 Suspense 語法來取用資料。在建構應用的過程中，我們將會介紹各種工具組，來處理軟體開發中常見的問題，包含路由、測試以及後端渲染（Server-side Rendering）。

我們希望這樣的教學順序能使你快速且深入地掌握 React，並創建出具有實務價值的應用程式。

React 的過去與未來

React 的創造者是 Facebook 的工程師 Jordan Walke。React 在 2011 年被納入 Facebook 的塗鴉牆專案中；接著用於 2012 年被收購的 Instagram。在 2013 年的 JavaScript 研討會（JSConf）中 React 公開了原始碼，正式成為前端領域百家爭鳴中的一員。同時期的競爭對手包含了 jQuery、Angular、Dojo、Meteor 等等。在這個時期，React 仍被認為只是專用於處理使用者介面（也就是 MVC 模式中的 View）的 JavaScript 工具。

在開源後，社群開始著手改造 React。2015 年 1 月，Netflix 宣布他們正在使用 React 來架構使用者介面；幾個月後，用以創造行動應用程式的 React Native 問世了；稍後，臉書公開了 ReactVR，使得 React 的渲染功能跨出了使用者介面以外的範疇；在 2015 年至 2016 年間，更多更泛用的工具被公開發表，例如用以處理路由功能的 React Router，以及用於狀態管理的 Redux 以及 Mobx。此後，React 逐漸被認定成一個用以創造一系列特定功能的「函式庫」，而非一個固定結構的「框架」。

2017 年推出的 Fiber 是 React 發展史上另一個重大事件。Fiber 以一種近似魔幻的實作手法改寫了 React 的渲染引擎，這個改動雖然沒有影響大部分已公開的程式介面，卻從內部徹底改變了 React 的運作邏輯。這一切的目的在於：使 React 的效能更好，且更符合現代應用程式設計需求。

在 2019 年，React 推出了 Hook，用以在元件之間新增或共用狀態邏輯。同時期也推出了 Suspense，用以最佳化非同步的資料存取行為。

在未來，我們可以預見 React 仍會持續改動。React 的成功源自於背後強大且有經驗的開發團隊。他們兼具進取與謹慎，在前瞻思考的同時，也能仔細考量各種變動可能對既有使用者帶來的影響。

既然改動不會停止，本書所提供的範例程式碼亦有可能受到影響。為了讓讀者可以放心地操作這些範例，我們會在 *pakcage.json* 檔案中加入明確的版本資訊，請依照正確的版本號碼安裝模組即可。

除了本書外，你也可以追蹤 React 的官方部落格（*https://facebook.github.io/react/blog*）以取得最新資訊。當新版本發佈時，團隊也會在此發文說明更新的內容。這個部落格支援多國語言，即便你的母語不是英文，仍然可以閱讀翻譯後的文件（*https://reactjs.org/languages*）。

《React 學習指南》第二版更動說明

這是《*React 學習指南*》的第二版。有鑒於 React 進化迅速，我們必須將此書自第一版改寫。我們將專注於探討那些被 React 團隊倡導的現代最佳化的實作模式，但也會提供一些舊有的語法作為參考，以協助你維護那些舊有的專案。

使用範例檔案

在本段落中，我們將介紹如何使用本書提供的範例檔案，以及一些有用的 React 工具。

GitHub 儲存庫

本書的所有檔案皆已上傳至 GitHub 儲存庫（*https://github.com/moonhighway/learning-react*）。

React Developer Tools

我們強烈建議你安裝 React 開發者工具（React Developer Tools）來輔助開發。你可以在 Chrome 以及 Firefox 中找到瀏覽器插件，或是下載獨立的應用程式（這適用於 Safari、IE 以及本地端的 React）。安裝完畢後，你可以透過 React Developer Tools 來檢視 React 元件樹；瀏覽各項屬性與狀態；甚至可以得知哪一個網站是使用 React 製作的——這將大幅提升你的除錯效率，並且協助你從實務網站中學習如何建構專案。

你可以透過以下網址來安裝瀏覽器插件。

- GitHub 頁面（*https://oreil.ly/5tizT*）
- Chrome（*https://oreil.ly/Or3pH*）
- Firefox（*https://oreil.ly/uw3uv*）

安裝完成後，只要網址列旁邊的 React 圖示亮起（見圖 1-1），就代表這個頁面正在使用 React。

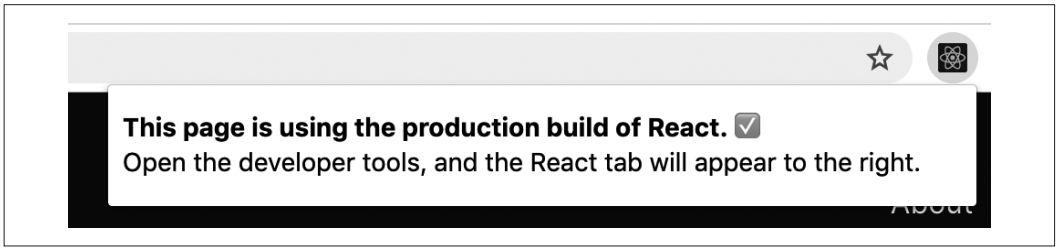

圖 1-1　在 Chrome 中使用 React Developer Tools

此時，只要打開瀏覽器的開發者工具，你會看到一個名為 React 的新標籤（見圖 1-2）。點擊該標籤將會顯示當前頁面的元件結構。

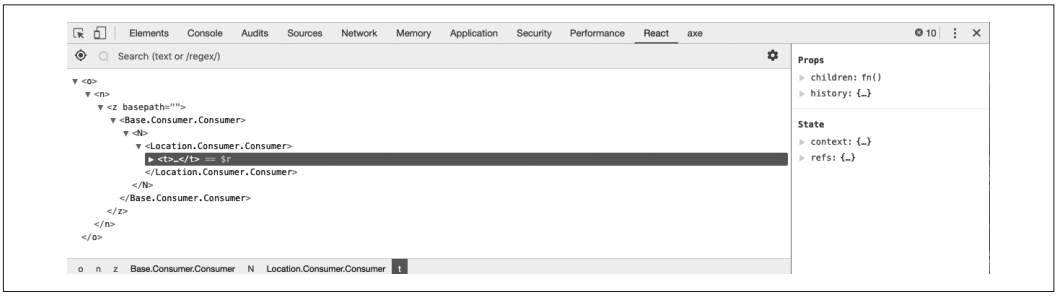

圖 1-2　使用 React 開發者工具檢視 DOM 模型

安裝 Node.js

Node.js 是一個 JavaScript 的執行環境，可用來建構包含前端到後端的應用。Node 不僅開源，也可以在絕大多數的作業系統上運作，例如 Windows、macOS 以及 Linux。我們將在第 12 章中使用 Node 來建構 Express 伺服器。

你必須要先安裝好 Node（但開發 React 應用並不需要精通 Node）。如果你不確定自己的系統是否已經安裝 Node，可以在命令列中輸入：

```
node -v
```

執行該命令後，你會見到目前安裝的 Node 版本號碼——這個數字必須大於 8.6.2。如果回報的是諸如「Command not found（指令不存在）」等錯誤，則代表系統尚未安裝 Node。你可以透過官方網站（*http://nodejs.org*）來安裝 Node。依照指示完成後，請再次使用 `node -v` 指令來確認版本號碼。

npm

在安裝 Node.js 時，你也同時安裝了 Node 的套件管理程式 npm。在 JavaScript 社群中，為了避免重複開發已存在的框架、函式庫或是各種工具函式，開發者會將原始碼開源並彼此共享——React 本身即是 npm 中的一個開源函式庫。在本書中，我們將透過 npm 來安裝套件。

大部分 JavaScript 專案都包含了一系列分類過的檔案以及一個 *package.json* 檔。它記載了該專案的各項資訊以及其相依性套件（dependencies）。當你在含有 *package.json* 的資料夾中執行 `npm install` 時，npm 將會自動安裝所有被條列出來的函式庫。

如果你打算從零開始建立專案並納入相依性套件，只需要執行：

```
npm init -y
```

這個指令會初始化專案並建立 *package.json* 檔。此後，你就可以透過 npm 管理相依性套件。可以透過以下指令安裝特定套件：

```
npm install package-name
```

或是透過以下指令移除套件：

```
npm remove package-name
```

Yarn

Yarn 是 npm 的替代選項。Facebook 在 2016 年將其公開，其他參與開發的公司還包含了 Google、 Exponent 以及 Tilde。如果你已經熟悉了 npm，使用 Yarn 將不會是件難事。首先，必須使用 npm 安裝 Yarn：

```
npm install -g yarn
```

接著，當你要安裝 *package.json* 中指定的相依性套件時，可以使用 `yarn` 來取代 `npm install`：

```
yarn add package-name
```

你可以透過以下指令來移除套件：

```
yarn remove package-name
```

Facebook 在他們的實務專案中使用了 Yarn；其他專案諸如 React、React Native 以及 Create React App 也納入了 Yarn。如果你在專案資料夾中發現了名為 *yarn.lock* 的檔案，代表著該專案採用了 Yarn。正如 `npm install` 指令一樣，你可以透過 `yarn` 指令來安裝所有使用到的套件。

討論到這裡，代表你已經設定好開發環境，可以開始學習 React 了。在下一章中，我們將為你介紹最現代、且 React 最常使用到的 JavaScript 語法。

第二章

JavaScript 與 React

JavaScript 發表於 1995 年，一路上經歷了許多變動。起初，開發者使用 JavaScript 在網頁中建構互動性的元素，例如按鈕點擊、游標懸停與表單檢查等等；接著，JavaScript 在 DHTML 以及 AJAX 等應用模式上逐漸成熟；在現代，隨著 Node.js 的問世，JavaScript 正式成為一個泛用的程式語言，可用於建構從前端到後端的各式應用——JavaScript 無所不在。

軟體公司、瀏覽器開發商以及開發者社群都影響了 JavaScript 的演化。ECMA（European Computer Manufacturers Association，歐洲電腦製造商協會）是主導 JS 變革的領頭羊，而修正提案則由開發者社群驅動——任何人都可以向 ECMA 的委員會遞交提案（*https://tc39.github.io/process-document*），委員會為其排出優先順序，並決定何者將被正式採用。

第一版的 ECMAScript（ECMAScript1）發佈於 1997 年；緊接著 ECMAScript2 發佈於 1998 年；ECMAScript3 則於 1999 年問世——該版本新增了正規表達式以及字串處理等功能；ECMAScript4 的協議過程則陷入了種種政治糾葛以及巨大的混亂中，因此從未正式發佈。在 2009 年，ECMAScript5（ES5）終於問世，並帶來了多種新功能，例如擴充的陣列方法、物件屬性以及支援 JSON 的函式庫。

在此之後，JavaScript 仍然充滿活力。在 ECMAScript6（發表於 2015 年，因此又稱 ES2015）後，JS 維持著一年一次的更新頻率。任何被 ECMA 委員會採用並進入計畫階段（Stage Proposal）的新提案都統稱為 ESNext——這個名詞讓我們得以簡單地指涉那些未來即將落地的功能。

所有新提案都將經歷五個清楚的工作階段（Stage）。最低的是階段零（Stage 0），代表全新的提議；最高的則是階段四（Stage 4），代表已經規劃完成的計畫。當一個計畫受到青睞時，將由瀏覽器開發商（例如 Chrome 以及 Firefox）來負責實作。

以 const 為例，在舊版的 JavaScript 中，我們總是使用 var 來宣告變數。然而，ECMA 委員會決定納入新的關鍵字 const 用以宣告常數（相關細節將於本章中詳述）。在 const 剛被社群提案時，你無法在瀏覽器腳本中使用 const。然而在此刻，因為 ECMA 委員會早已正式接納 const 並完成了該提案，各大瀏覽器也完成了支援的實作，因此該功能就可以順利運作了。

本章節將討論的許多功能都已被最新的瀏覽器支援，但我們還是會介紹如何「編譯」JavaScript 程式碼。透過編譯，我們可以將某些支援度較低的新語法轉化為支援度較高的舊語法。*kangax* 相容性表格（*https://oreil.ly/oe7la*）詳細呈現了最新 JavaScript 語法在各大瀏覽器中的支援狀態。

在接下來的內容中，我們將介紹本書中常用到的 JavaScript 新版語法。如果你尚未熟悉這些新功能，可以藉此奠定良好的學習基礎；但如果你已經相當熟練，請直接閱讀下一章即可。

宣告變數

在 ES2015 前，使用 var 是宣告變數唯一的方式。現在，我們擁有更多不同功能的選項。

關鍵字：const

常數（Constant）是無法被覆寫的變數，一旦宣告，你將無法修改它的值。在 JavaScript 中，許多變數理論上是不該被修改的，因此我們會頻繁地使用 const。常數的概念早已被許多語言採納，而 JavaScript 則到 ES6 才開始支援。

ES2015 之前，我們只能使用 var 來宣告變數，且無法限制值的修改：

```
var pizza = true;
pizza = false;
console.log(pizza); // false
```

我們無法重設 const 變數的值——如果堅持這麼做，將會產生錯誤（見圖 2-1）。

```
const pizza = true;
pizza = false;
```

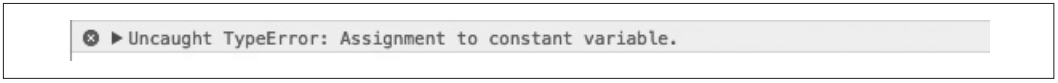

圖 2-1　為 const 變數指派新的值將會回報錯誤

關鍵字：let

現代的 JavaScript 實作了**語法變數域**（*Lexical Variable Scope*；又稱靜態變數域）。使用大括號 { } 會創建一個程式碼區塊（Code Block）。在函式的宣告裡，由大括號所創建的程式碼區塊會產生一個新的**變數作用域**（Variable Scope）隔開內部與外部透過 var 宣告的變數。然而，當我們使用 if/else 語法時，大括號卻不會如同函式一般，產生區隔 var 變數域的效果。如果你熟悉其他程式語言，這應該會讓你感到相當困惑——這樣的問題直到 let 關鍵字問世後才得以解決。

在以下範例中，if/else 的程式碼區塊內宣告的變數，其作用域並沒有被侷限在大括號中。

```
var topic = "JavaScript";

if (topic) {
  var topic = "React";         // <= 注意這行使用 var
  console.log("block", topic); // 會印出 block React
}

console.log("global", topic);  // 會印出 global React
```

然而，只要使用 let 關鍵字，我們就可以將變數侷限於程式碼區塊內，避免取用或修改到全域變數：

```
var topic = "JavaScript";

if (topic) {
  let topic = "React";         // <= 注意這行使用 let
  console.log("block", topic); // 會印出 block React
}

console.log("global", topic);  // 會印出 global JavaScript
```

另舉一個 for 迴圈的例子示範，大括號並無法隔開 var 變數的作用域：

```
var div,
  container = document.getElementById("container");

for (var i = 0; i < 5; i++) {
  div = document.createElement("div");
  div.onclick = function() {
    alert("This is box #" + i);
  };
  container.appendChild(div);
}
```

在以上程式碼中，我們建構了五個 div 色塊並將之新增於 container 中。每個 div 色塊都被指派了一個點擊事件的處理器（Onclick Event Handler），會觸發警告視窗並顯示色塊的編號。在 for 迴圈中我們宣告了全域變數 i，並持續遞增 i 直到 5 為止。然而，不論點擊哪一個色塊，警告視窗都會回報 #5——因為全域變數 i 最後的值是 5（見圖 2-2）。

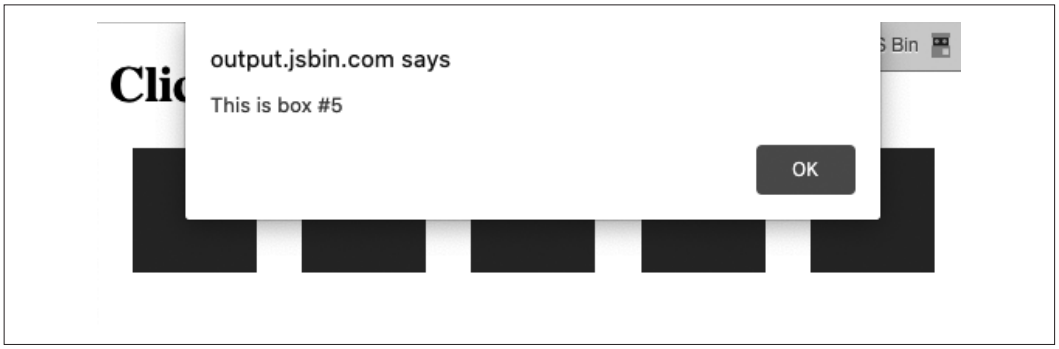

圖 2-2　每一個色塊都會回報 #5

反之，如果在 for 迴圈中使用 let 取代 var 用以宣告 i，這會使 i 的變數域侷限於迴圈的程式碼區塊中。如此一來，在我們點擊色塊時，警告視窗將會回報迴圈區塊內部的 i 值，也就是色塊各自的編號（見圖 2-3）。

```
const container = document.getElementById("container");
let div;
for (let i = 0; i < 5; i++) {
  div = document.createElement("div");
  div.onclick = function() {
    alert("This is box #: " + i);
```

```
  };
  container.appendChild(div);
}
```

圖 2-3　使用 let 限縮 i 的變數域

樣板字串

樣板字串（*Template String*）提供了一個新的連結字串的方法，使我們得以將變數嵌入字串中。它在英文裡又被稱作 Template Literal 或是 String Template。

在傳統的語法裡，我們會使用加號 + 來連結字串與變數：

```
console.log(lastName + ", " + firstName + " " + middleName);
```

現在，我們可以建構一個樣板字串並將變數包覆於 ${ } 之中，直接將值嵌入：

```
console.log(`${lastName}, ${firstName} ${middleName}`);
```

我們也可以在樣板字串中透過 ${ } 呼叫任何具有回傳值的函式。

樣板字串能舒服地顯示空白字元，這使我們得以透過簡潔的語法，建立像是電子郵件草稿、程式碼範例、任何頻繁使用到空白字元的文字或是一個具有多個換行字元的段落，卻不會弄髒程式碼：

```
const email = `
Hello ${firstName},

Thanks for ordering ${qty} tickets to ${event}.

Order Details
```

```
${firstName} ${middleName} ${lastName}
     ${qty} x $${price} = $${qty*price} to ${event}

You can pick your tickets up 30 minutes before
the show.

Thanks,

${ticketAgent}
`
```

在樣板字串問世前，使用 JavaScript 編寫包含 HTML 語法的字串是很痛苦的，因為我們必須將其全部擠在同一行內。現在，因為樣板字串對空白與換行字元良好的支援，開發者可以優雅地編寫出高可讀性且內嵌變數的程式碼：

```
document.body.innerHTML = `
<section>
  <header>
      <h1>The React Blog</h1>
  </header>
  <article>
      <h2>${article.title}</h2>
      ${article.body}
  </article>
  <footer>
      <p>copyright ${new Date().getYear()} | The React Blog</p>
  </footer>
</section>
`;
```

創建函式

函式可以用來執行重複的任務。在本節中，我們將檢視幾種不同建構函式的語法，並逐一解析。

函式宣告

在透過函式宣告來建構函式時，我們會以 function 關鍵字起手，並提供函式名稱（在本例中為 logCompliment），其餘陳述句則放在大括號之中：

```
function logCompliment() {
  console.log("You're doing great!");
}
```

宣告完成後，可以透過呼叫來執行函式：

```
function logCompliment() {
  console.log("You're doing great!");
}

logCompliment();
```

執行以上程式碼之後，我們會見到主控台印出一段文字。

函式表達式

另一個建構函式的方式是使用**函式表達式**（Function Expression）——我們先創建一個函式，再將之指派給變數：

```
const logCompliment = function() {
  console.log("You're doing great!");
};

logCompliment();
```

以上程式碼將產生和前例一樣的效果，主控台會印出一段文字。

以上我們示範了兩種創建函式的方法，值得注意的是：函式宣告會被**提吊**（Hoist），但函式指派則不會。換言之，在使用函式宣告時，你可以在函式的語法區塊**之前**就呼叫它；然而，如果你使用函式表達式，則一定要在函式的語法區塊**之後**才能呼叫它——如果違反這個規則，將會產生錯誤。

```
// OK！在函式宣告區塊之前就呼叫
hey();

// 函式宣告
function hey() {
  alert("hey!");
}
```

以上程式碼可以正常運作，你會順利看到警告視窗。這樣的語法之所以合法，是因為 JavaScript 將函式「提吊」了——它實際上被移動到檔案上方的區域。然而，如果我們使用表達式建構函式，則會得到錯誤：

```
// Error！在函式宣告區塊之前就呼叫
hey();
// 函式指派
const hey = function() {
  alert("hey!");
```

```
};
TypeError: hey is not a function
```

這樣的差異也許很微小，但在引入檔案或函式時往往會造成非預期的 TypeError。如果見到類似的情形，可以考慮使用函式宣告的語法將程式碼進行重構。

傳遞引數

之前建構的 logCompliment 函式並沒有接受任何的引數（Argument）。如果我們希望函式可以接受動態的輸入值，可以在小括號中命名一個或多個參數（Parameter）。以下範例示範了如何新增 firstName 作為函式的參數：

```
const logCompliment = function(firstName) {
  console.log(`You're doing great, ${firstName}`);
};

logCompliment("Molly");
```

如此一來，當我們在呼叫函式時，控制台印出的訊息將會依照輸入的人名而有所不同。

我們可以繼續為函式新增 message 參數。這麼一來，就不必再將訊息寫死：

```
const logCompliment = function(firstName, message) {
  console.log(`${firstName}: ${message}`);
};

logCompliment("Molly", "You're so cool");
```

函式回傳值

目前的 logCompliment 函式會將訊息印出至控制台中，然而在大部分的使用情境中，我們會使用 return 關鍵字讓函數回傳一個值。為了表明程式碼的意圖，可以將原本的函式改名為 createCompliment：

```
const createCompliment = function(firstName, message) {
  return `${firstName}: ${message}`;
};

createCompliment("Molly", "You're so cool");
```

如果想要確認結果，可以使用 console.log 函式將回傳結果印出：

```
console.log(createCompliment("You're so cool", "Molly"));
```

為參數提供預設值

在 C++ 與 Python 中，開發者可以為參數提供預設的值——當函式被呼叫，卻沒有傳入值時，就會自動使用預設值進行運算。JavaScript 在 ES6 也補上了這個功能。

舉例來說，我們可以為以下函式的 name 以及 activity 提供預設的字串：

```
function logActivity(name = "Shane McConkey", activity = "skiing") {
  console.log(`${name} loves ${activity}`);
}
```

當我們呼叫 logActivity 卻沒有傳入引數時，預設值將會被自動帶入——預設值可以是任何資料型別，不侷限於字串：

```
const defaultPerson = {
  name: {
    first: "Shane",
    last: "McConkey"
  },
  favActivity: "skiing"
};

function logActivity(person = defaultPerson) {
  console.log(`${person.name.first} loves ${person.favActivity}`);
}
```

箭頭函式

箭頭函式（*Arrow Function*）是 ES6 中提供的新功能。有了它，我們可以在不使用 function 關鍵字的情況下創建函式。在某些狀況中，甚至連 return 也可以省略。以下我們將創建一個函式，它接受 firstName 並回傳一段加工過後的字串：

```
const lordify = function(firstName) {
  return `${firstName} of Canterbury`;
};

console.log(lordify("Dale")); // Dale of Canterbury
console.log(lordify("Gail")); // Gail of Canterbury
```

使用箭頭函式，我們可以將以上語法大幅簡化：

```
const lordify = firstName => `${firstName} of Canterbury`;
```

在以上程式碼中，我們將所有語法濃縮成一行。既沒有使用 function 關鍵字；也沒有使用 return——語法中的箭頭直接指向了應當被回傳的值。此外，如果函式只使用一個參數，我們可以不必使用小括號。

如果箭頭函式包含了一個以上的參數，我們必須使用小括號 () 將之包覆：

```
// 使用標準的函式語法
const lordify = function(firstName, land) {
  return `${firstName} of ${land}`;
};

// 使用箭頭函式語法
const lordify = (firstName, land) => `${firstName} of ${land}`;

console.log(lordify("Don", "Piscataway")); // Don of Piscataway
console.log(lordify("Todd", "Schenectady")); // Todd of Schenectady
```

在以上程式碼中，因為函式只包含一個陳述句，我們仍然可以透過箭頭函式將語法壓縮在一行之內。如果你想設計的功能沒有這麼單純。你可以使用大括號 {} 來包覆多行程式碼：

```
const lordify = (firstName, land) => {
  if (!firstName) {
    throw new Error("A firstName is required to lordify");
  }

  if (!land) {
    throw new Error("A lord must have a land");
  }

  return `${firstName} of ${land}`;
};

console.log(lordify("Kelly", "Sonoma")); // Kelly of Sonoma
console.log(lordify("Dave")); // ! JAVASCRIPT ERROR
```

以上函式包含了多個陳述句，因此必須使用大括號包覆起來。儘管如此，箭頭函式的語法仍然較傳統的寫法簡潔不少。

回傳物件

如果我們想要令箭頭函式回傳一個物件，該怎麼做呢？想像有一個名為 person 的函式，它接收參數 firstName 以及 lastName 並回傳一個物件：

```
const person = (firstName, lastName) =>
    {
        first: firstName,
        last: lastName
    }

console.log(person("Brad", "Janson"));
```

如果我們執行以上程式碼，將會得到錯誤 Uncaught SyntaxError: Unexpected token ':'。要修正這個問題，只需要使用小括號包覆物件語法即可：

```
const person = (firstName, lastName) => ({
  first: firstName,
  last: lastName
});

console.log(person("Flad", "Hanson"));
```

這個小錯誤常常在 JavaScript 與 React 中發生，值得留意。

箭頭函式的作用域

標準函式並不會靜態地決定 this 的作用域（Scope）。請參考以下範例，程式碼中 setTimeout 的回呼函式中的 this 實際上並不是 tahoe 物件：

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
    setTimeout(function() { // <= 標準函式作為回呼，this 是動態決定的
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

tahoe.print(); // Uncaught TypeError: Cannot read property 'join' of undefined
```

以上程式碼的錯誤意味著：this 物件並不具有 .join 方法。如果我們試著印出 this，會發現它其實是 Window 物件：

```
console.log(this); // Window {}
```

要解決這個問題，我們可以使用箭頭函式，使 this 被固定在靜態的作用域中。

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
```

```
    setTimeout(() => { // <= 箭頭函式作為回呼，this 是靜態決定的
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

tahoe.print(); // Freel, Rose, Tallac, Rubicon, Silver
```

在以上程式碼中，print 方法會如預期般地運作，印出 mountains 陣列中文字的結合。在使用 JavaScript 時，時時留心作用域的概念是很重要的。

值得注意的是，箭頭函式並沒有自己的 this 作用域：

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: (delay = 1000) => { // <= 箭頭函式作為物件方法
    setTimeout(() => {
      console.log(this.mountains.join(", "));
    }, delay);
  }
};

tahoe.print(); // Uncaught TypeError: Cannot read property 'join' of undefined
```

在以上程式碼中，使用箭頭函式作為物件方法會導致 this 指向 Window（也就是 tahoe 作用域中的 this）[譯註 1]。

編譯 JavaScript

當新的 JavaScript 提案得到支持且順利成案，開發者社群往往希望在瀏覽器廣泛支援前就可以先行使用。要滿足這個需求，唯一的方法就是將新版的程式碼轉換成較舊但支援度較佳的語法——這個過程稱為**編譯**（*Compile*）。Babel（*http://www.babeljs.io*）是其中最受歡迎的工具。

Babel 所執行的「編譯」並不是傳統概念上像 C 語言那般的編譯——我們的 JavaScript 程式碼並沒有被編寫成二進位碼；而只是被轉換成舊版的 JS 語法。

舉例來說，以下語法使用了箭頭函式加上預設參數：

```
const add = (x = 5, y = 10) => console.log(x + y);
```

[譯註 1] JavaScript 的作用域以及 this 本身就是一個糾葛的議題，本書的主題是 React，礙於篇幅無法完整地著墨。如果讀者想要更完整地了解，可以參見 MDN Web Docs 中對 this 的專文說明。

使用 Babel 編譯後，會得到以下結果：

```
"use strict";

var add = function add() {
  var x =
    arguments.length <= 0 || arguments[0] === undefined ? 5 : arguments[0];
  var y =
    arguments.length <= 1 || arguments[1] === undefined ? 10 : arguments[1];
  return console.log(x + y);
};
```

在以上編譯結果中，Babel 新增了 `use strict` 宣告；`x` 與 `y` 參數也成了 `arguments` 陣列的成員（你也許有學過類似的技巧）。編譯後的程式碼可以確保最廣泛的支援度。

如果你想要深入了解 Babel 的工作原理，可以參考官方網站的 REPL（Read-Eval-Print Loop，互動式程式設計環境）工具（*https://babeljs.io/repl*）：只要在介面左側輸入程式碼，就可以在右側對照編譯結果。

在實務中，編譯的過程通常會透過諸如 webpack 或是 Parcel 等工具加以自動化，我們會在稍後的章節內詳細討論。

物件與陣列

自 ES2016 起，JavaScript 支援了一些新穎的語法，用以拆解物件與陣列中的資料。這些語法被廣泛地用在 React 的開發中。以下我們將介紹幾個最常見的用例，包含物件解構（Object Destructuring）、物件語法強化（Object Literal Enhancement）以及延展運算子（Spread Operator）。

物件解構

解構賦值（Destructuring Assignment）使我們可以從物件中拆解出特定欄位的值並指派給變數。舉例來說，以下 `sandwich` 物件具有四個鍵值，但我們只需要其中的 `bread` 以及 `meat`：

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};
```

```
const { bread, meat } = sandwich;

console.log(bread, meat); // dutch crunch tuna
```

以上程式碼將 bread 以及 meat 從物件中取出並創建了新的變數。

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};

let { bread, meat } = sandwich;

bread = "garlic";
meat = "turkey";

console.log(bread); // garlic
console.log(meat); // turkey

console.log(sandwich.bread, sandwich.meat); // dutch crunch tuna
```

我們也可以將函式的引數解構。以下為例，箭頭函式將印出某人的姓氏，但是採用了傳統的物件取值的作法，語法相當冗長：

```
const lordify = regularPerson => {
  console.log(`${regularPerson.firstname} of Canterbury`);
};

const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
};

lordify(regularPerson); // Bill of Canterbury
```

透過物件解構，我們可以直接將 firstname 從物件中「解構」出來，而不需進行標準的物件取值：

```
const lordify = ({ firstname }) => {
  console.log(`${firstname} of Canterbury`);
};

const regularPerson = {
  firstname: "Bill",
```

```
  lastname: "Wilson"
};

lordify(regularPerson); // Bill of Canterbury
```

物件解構還可以有更複雜的用法。以下為例，regularPerson 是一個巢狀的物件結構，其中 spouse 是物件中的物件[譯註 2]：

```
const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson",
  spouse: {
    firstname: "Phil",
    lastname: "Wilson"
  }
};
```

如果我們想要解構出 spouse 的 firstname，可以使用雙重的大括號 {} 以及冒號：將函式的參數修改如下：

```
const lordify = ({ spouse: { firstname } }) => {
  console.log(`${firstname} of Canterbury`);
};

lordify(regularPerson); // Phil of Canterbury
```

陣列解構

我們也可以從陣列中解構出值。以下為例，只取出陣列的第一個值，並將之指派給變數：

```
const [firstAnimal] = ["Horse", "Mouse", "Cat"];
console.log(firstAnimal); // Horse
```

還可以透過逗號來略過不需要的值。以下為例，只取用陣列中的第三個值，並使用逗號略過前兩個：

```
const [, , thirdAnimal] = ["Horse", "Mouse", "Cat"];
console.log(thirdAnimal); // Cat
```

在後續的範例中，我們將進一步展示如何結合陣列解構以及延展運算子，來編寫出簡練的程式碼。

譯註 2 spouse 是配偶的意思。