序

網頁開發者需要瞭解許多技術才能做出受歡迎的應用程式。特別是在 React 領域裡有大量的資料可以參考，但這也是問題的一部分，因為並非所有資料都一致，你必須自行摸索，設法從許多教學資源和部落格文章中整理出一套自洽的理論，並且避免遺漏某些知識。你還會經常擔心自己學到的東西是否過時了。

這就是 Tejas Kumar 在書中做出的貢獻。Tejas 在 React 領域有多年的經驗，擅長深入探討那些能夠為你打下深厚知識基礎的主題。他是經驗豐富的工程師，知識的寶庫，即將幫助你遊刃有餘地使用 React。多年來，我有幸在開源領域、會議演講、教育內容創作，以及個人領域和 Tejas 進行各式各樣的合作，希望你明白，在你手上的這本書出自一位才華洋溢的作者。

這本書將帶你深入探討一些應該不會在其他地方接觸到的主題，這些主題將幫助你用正確的心智模式來「以 React 的方式思考」。你將瞭解 React 存在的初衷，在你考慮使用 React 來解決問題時，這會提供一個很棒的參考框架。React 不是憑空發明的，瞭解它的起源可以幫助你明白 React 試圖解決的問題，進而避免你亂用工具。

你將瞭解 JSX、虛擬 DOM、調和以及並行 React 等基本概念，幫助你更有效地使用這個工具。我一直相信，提升工具的使用經驗最好的方法是瞭解工具的運作原理，因此，即使你有多年的 React 使用經驗，這些章節也可以讓你看到新的可能性和新技術，讓你真正理解 React，而非只是將東西拼湊起來、祈望老天給你一個好結果。

你將發現專家用來建構有效且強大抽象的模式。React 本身是一種極其快速的 UI 程式庫，但在建構複雜的應用程式時，有時要進行效能優化，Tejas 會告訴你如何透過 React 的記憶化（memoization）、延遲載入和狀態管理技術來進行優化。此外，你會看到在這個生態系統中最好的程式庫如何運用模式，例如複合組件、算繪 prop、prop

getter、state reducer 和控制 prop…等。它們是建構 React 應用程式的必備工具。即使你只想要使用現成的解決方案，瞭解這些抽象的運作方式也可以幫助你更有效地使用它們。

Tejas 並非只探討 React 的理論層面，他也介紹實用的框架，例如 Remix 和 Next.js，幫助你充分利用 React 的全棧（full stack）能力，以提供最佳的使用者體驗。從一開始，你就可以透過 React 的伺服器算繪能力，在前端和後端皆使用 React 來建構整個使用者體驗，進而掌握自己的命運。你也將瞭解被 React 用來提升使用者體驗的先進技術，例如 Server Components 和 Server Actions。

我保證，在你閱讀 Tejas 整理的內容後，你將知道如何使用 React 來建構優秀的應用程式。祝你在學習 React 這個全球最流行的 UI 程式庫時一切順利，好好享受這段旅程吧！

— Kent C. Dodds
https://kentcdodds.com

前言

這本書不適合打算學習怎麼使用 React 的讀者。如果你對 React 不熟悉，並且正在尋找學習資源，*react.dev* 的 React 文件是很好的起點。然而，這本書適合對於 React 的用法沒那麼感興趣，但是對 *React 如何運作*比較感興趣的人。

在我們相處的時光裡，我們將穿越多個 React 概念，瞭解它們的基本機制，探索它們如何互相配合，好讓你更有效地使用 React 來建立應用程式。在理解基本機制的過程中，我們將發展必要的心智模型，以便高度寫實地思考 React 及其生態系統。

本書假設我們對這句話有一定程度的瞭解：「瀏覽器算繪網頁」。網頁就是 HTML 文件，它的樣式是用 CSS 來定義的，它的互動功能是用 JavaScript 來實現的。本書也假設你對於 React 的用法有一定程度的瞭解，並且曾經寫過一兩個 React 應用程式。理想情況下，你已經有一些 React 應用程式投入使用了。

我們會先介紹 React，回顧它的歷史，將思緒帶回它在 2013 年首次作為開源軟體發表的時刻。我們將從那個時候開始探索 React 的核心概念，包括組件模型、虛擬 DOM 和調和。我們將深入研究 JSX 的編譯器理論，談論 fibers，並深入瞭解它的並行設計模型。這可以讓你得到有用的心得，幫助你更順利地記住該記住的事情，以及透過像 `React.memo` 和 `useTransition` 這類的強大基本元素來延遲應該延遲的算繪工作。

在這本書的後半部分，我們將探索 React 框架，包括它們解決哪些問題，以及它們解決問題的機制。我們將編寫自己的框架來進行探索，該框架可以解決幾乎所有 web 應用程式的三個重大問題：伺服器算繪、路由，和資料提取。

自行解決這些問題可以幫助你瞭解框架如何解決這些問題。我們也會深入探討 React Server Components（RSCs）和伺服器操作（server action）、瞭解次世代工具，例如 bundler 和 isomorphic 路由器的作用。

最後，我們將抽離 React，討論 Vue、Solid、Angular、Qwik 等替代方案。我們將探討訊號和細回應性（fine-grained reactivity）——相對於 React 的那種較粗的回應性模型。我們也會探討 React 對於 signal 的回應：Forget 工具鏈，以及它相較於 signal 表現如何。

需要討論的主題太多了，事不宜遲，我們開始吧！

本書編排慣例

本書使用下列的編排規則：

斜體字（*Italic*）

代表新術語、URL、email 地址、檔名、副檔名。中文以楷體表示。

定寬字（`Constant width`）

用於程式碼，並在文字段落內，用來代表變數、函式名稱、資料庫、資料型態、環境變數、陳述式、關鍵字等程式元素。

定寬粗體字（**`Constant width bold`**）與淡灰文字

在印刷版本的第 10 章中，用來突顯程式碼區塊內的差異。

這個圖案代表一般說明。

致謝

這是我寫的第一本書，很慶幸這趟旅途並非一人獨行。你看到的內容是很多傑出人士一起努力的成果。在此，我想要感謝這些人對書籍內容的貢獻。

請不要跳過這個部分，因為這些人值得你的注目和感謝。先從直接幫助我完成這本書的人開始唱名：

- 第一位感謝的對象必然是我的另一半，Lea。我花很多時間來寫這本書，經常犧牲與家人相處的時間。因為我對於這個主題的興趣，以及想和讀者分享的渴望，我稍微挪用渡假的時間以及陪伴另一半的機會來完成這本書。她一直以來都支持和鼓勵我，在此表達我的感激。

第一章

基本內容

首先，我們來看一個聲明：React 是為所有人而設計的。事實上，你不需要看這本書就可以繼續使用 React 而沒有任何問題！這本書將帶領好奇 React 的基本機制、高級模式和最佳實踐法的人更深入地探討 React。所以這本書比較適合用來瞭解 React 的工作原理，而不是學習如何使用 React。有許多其他書籍的目標是教你如何以最終使用者的身分使用 React。相較之下，這本書是讓你用程式庫或框架作者的視角來瞭解 React，而不是終端使用者。為了貫徹這個主題，我們要從最上層開始深入探討：高級入門主題。我們將從 React 的基本知識談起，然後深入地探討 React 的工作原理細節。

在這一章，我們要討論 React 為什麼存在，它如何運作，以及它可以解決哪些問題。我們將介紹它的最初靈感和設計，並從它出身於 Facebook 的卑微起源開始，一直回顧到今日的普及化解決方案。這一章有點像 meta chapter（沒有惡意），因為在深入探討細節之前，瞭解 React 的背景脈絡非常重要。

為什麼會有 React？

一言以蔽之：更新。在網路時代的蠻荒時期有許多靜態網頁，我們必須填寫表單，按下 Submit，然後載入一個全新的網頁。雖然這種做法有一段時間沒什麼問題，但後來，網路體驗的功能有了顯著的提升。我們希望 web 可以隨著功能的增加而提供更出色的使用者體驗。我們想要立刻看到畫面更新，而不必等待新網頁的算繪和載入。我們希望網路和網頁感覺起來更敏捷和「即時」。問題在於，這些即時更新的很多層面都難以大規模實作：

效能

更新網頁經常造成效能瓶頸，因為我們通常會觸發瀏覽器重新計算網頁佈局（稱為 reflow（重新排列））和重新繪製。

可靠性

追蹤狀態，並確保狀態在豐富的 web 體驗中保持一致並不容易，因為我們必須在多個地方記錄狀態，並確保狀態在所有地方保持一致。這個要求在很多人於同一個碼庫（codebase）上工作時，特別難以實現。

資訊安全

我們必須對注入網頁的所有 HTML 和 JavaScript 進行消毒，以防止跨站命令稿攻擊（XSS）和跨站請求偽造（CSRF）等漏洞。

為了完全瞭解並讚賞 React 為我們解決這些問題的方法，我們來瞭解 React 被創作出來的背景，以及 React 不存在或在 React 問世前的世界。

在 React 問世之前的世界

下面是在 React 出現之前，建構 web 應用程式時可能遇到的重大問題。我們必須釐清如何讓應用程式感覺起來既迅速又即時，並讓數百萬位使用者安全且可靠地運作。例如，我們來考慮按下按鈕的動作：當使用者按下按鈕時，我們想要更新使用者介面，以反映按鈕已被按下。我們至少要考慮四種可能的使用者介面狀態：

按下前

按鈕處於預設狀態，尚未被按下。

已按下但等待處理

按鈕已被按下，但按鈕應執行的操作尚未完成。

按下且成功

按鈕已被按下，且按鈕應執行的操作已完成。接下來，我們可能想要將按鈕恢復成被按下之前的狀態，或讓按鈕變色（變為綠色）以代表成功。

按下且失敗

按鈕已被按下，但按鈕應執行的操作失敗了。接下來，我們可能希望按鈕恢復成按下之前的狀態，或變色（變為紅色）以表示失敗。

有了這些狀態之後，我們要想出如何更新使用者介面以反映它們。一般來說，更新使用者介面需要以下步驟：

1. 使用某種元素定位 API（例如 `document.querySelector` 或 `document.getElementById`），在主控（host）環境中（通常是瀏覽器）找到按鈕。
2. 將事件監聽器附加到按鈕上，以監聽按下事件。
3. 當事件發生時，執行所有狀態更新。
4. 當按鈕離開網頁時，刪除事件監聽器並清理任何狀態。

雖然這個例子很簡單，但它是很好的起點。假設有一個標為「Like」的按鈕，我們想要在使用者按下它時，將它更新為「Liked」，怎麼做？首先，我們會有一個 HTML 元素：

```
<button>Like</button>
```

我們要設法使用 JavaScript 來引用這個按鈕，所以我們給它一個 `id` 屬性：

```
<button id="likeButton">Like</button>
```

好！現在有一個 `id`，JavaScript 可以使用它來讓按鈕有互動性。我們可以使用 `document.getElementById` 來取得按鈕的參考，然後為按鈕附加一個事件監聽器，以監聽按下事件：

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  // 做某些事
});
```

現在有一個事件監聽器，可以在按鈕被按下時執行一些操作。假設我們想在按鈕被按下時，將按鈕的標籤更新為「Liked」，我們可以更新按鈕的文本內容來做到：

```
const likeButton = document.getElementById("likeButton");
likeButton.addEventListener("click", () => {
  likeButton.textContent = "Liked";
});
```

現在有一個「Like」按鈕，當它被按下時，會顯示「Liked」了。問題在於，我們無法將那些東西「unlike（取消喜歡）」，接著來修復這個問題，如果按鈕在「Liked」狀態下再次被按下，就將它更新為顯示「Like」。我們要為按鈕加入一些狀態，以記錄它是否已被按下，所以為按鈕加入一個 `data-liked` 屬性：

```
<button id="likeButton" data-liked="false">Like</button>
```

不可變性也讓強大的開發者工具得以實現，例如使用 Replay.io 等工具來進行時間旅行偵錯，開發者可以在應用程式的狀態改變過程中向前和向後步進，在任何時間點檢查 UI。這只能在每一個狀態更新都被保存成未被修改的唯一快照時才能辦到。

React 實踐不可變狀態更新是一項經過深思熟慮的設計決策，它帶來許多好處。它符合現代泛函編程原則，促成高效率的 UI 更新，可優化效能，降低 bug 的可能性，並改善整體開發者體驗。這種狀態管理方法支持 React 的許多高階功能，並且還會繼續成為 React 發展的基石。

釋出 React

單向資料流徹底翻轉 web 應用程式多年來的建構方式，它也曾經備受質疑。Facebook 是一家擁有大量資源、大量使用者和大量具有不同意見的工程師的大型公司，所以它的發展過程充滿困難和挑戰。React 經過大量的檢驗，在內部取得成功，被 Facebook 採用，然後被 Instagram 採用。

接下來，它在 2013 年被開源，向全世界公開，並遭受巨大的反彈。人們強烈批評 React 使用 JSX，指責 Facebook「把 HTML 放入 JavaScript」並破壞關注點分離。Facebook 成為千夫所指的一家「重新思考最佳實踐法」並破壞 web 的公司。最終，經過 Netflix、Airbnb 和 The New York Times 等公司緩慢且穩定的採用之後，React 成為建構 web 使用者介面的事實標準。

這個故事有很多細節因為超出本書的範圍而被省略，但在深入瞭解 React 的細節之前瞭解它的背景非常重要，特別是 React 企圖解決的技術問題種類。如果你對 React 的故事很感興趣，可以在 YouTube 上免費觀看由 Honeypot 製作的紀錄片「*React.js: The Documentary*」，該片完整記錄了 React 的發展歷史。

由於 Facebook 是這些大規模問題的近距離觀察者，React 開創一種「基於組件」的方法來建構使用者介面，解決了這些問題以及許多其他問題，其中的每一個組件都是一個可以重複使用，也可以和其他組件組合以建構複雜 UI 的獨立程式碼單元。

一年後，React 被當成開放原始碼軟體釋出，Facebook 釋出了 Flux：一種用來管理 React 應用程式內的資料流的模式。Flux 的目的是應對大規模應用程式內的資料流管理挑戰，它也是 React 生態系統的重要成分。我們來看一下 Flux，以及它在 React 裡的作用。

Flux 架構

Flux 是 Facebook（現在的 Meta）提倡的一種建構用戶端 web 應用程式的架構設計模式（圖 1-3）。它強調單向資料流，使 app 內部的資料流更可預測。

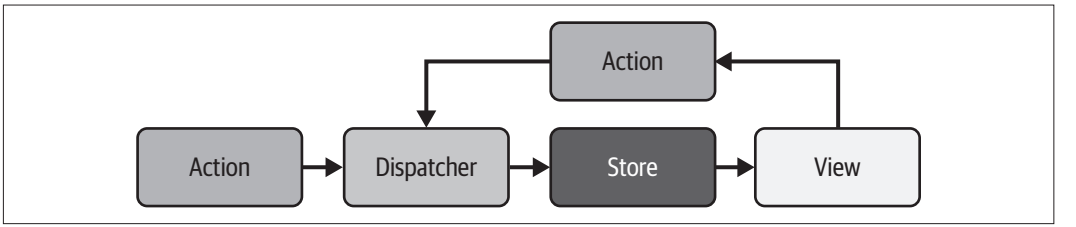

圖 1-3　Flux 架構

以下是 Flux 架構的關鍵概念：

Action

Action 是一個簡單的物件，裡面有新資料，以及用來識別的類型屬性。它們代表系統的外部和內部輸入，例如使用者互動、伺服器回應，和表單輸入。Action 會被中央的 Dispatcher 分給各個 Store：

```
// Action 物件範例
{
  type: 'ADD_TODO',
  text: 'Learn Flux Architecture'
}
```

Dispatcher

Dispatcher 是 Flux 架構的中央樞紐。它接收 Action 並將它們分給已在應用程式中註冊的 Store。它管理一組回呼函式（callback），每一個 Store 都會向 Dispatcher 註冊自己及其回呼函式。當 Action 被分配時，它會被送給所有已註冊的回呼函式：

```
// 分配 Action 的例子
Dispatcher.dispatch(action);
```

Store

Store 裡面有應用程式狀態和邏輯。它們有點類似 MVC 架構中的 model，但它們管理多個物件的狀態。它們向 Dispatcher 註冊並提供回呼函式來處理 Action。當 Store 的狀態被更新時，它會發出變更事件，來讓 View 知道發生了變化：

```
// Store 範例
class TodoStore extends EventEmitter {
  constructor() {
    super();
    this.todos = [];
  }

  handleActions(action) {
    switch (action.type) {
      case "ADD_TODO":
        this.todos.push(action.text);
        this.emit("change");
        break;
      default:
      // no op
    }
  }
}
```

View

View 是 React 組件。它們監聽來自 Store 的變更事件，並在它們依賴的資料改變時更新自己。它們也可以建立新 Action 來更新系統狀態，形成一個單向的資料流循環。

Flux 架構透過系統來促成單向資料流，可方便追蹤變更。這種可預測性可以作為編譯器的基礎來進一步優化程式碼，就像 React Forget 一樣（稍後會詳細介紹）。

Flux 架構的好處

Flux 架構有各種好處，有助於管理複雜性，並讓 web 應用程式更容易維護。以下是一些值得注意的好處：

單一真相來源

Flux 強調讓應用程式狀態擁有單一真相來源，它被儲存在 Store 裡。集中管理狀態可讓應用程式的行為更可預測且更容易理解，它可以消除「多個互相依賴的真相來源」帶來的複雜性，這種複雜性可能導致應用程式中的 bug 和一致性問題。

可測試性

Flux 具備明確的結構及可預測的資料流，讓應用程式高度可測試。將關注點分到系統的不同部分（如 Action、Dispatcher、Store 和 View），可以讓你分別對每一個部分進行單元測試。此外，當資料流是單向、且狀態被儲存在特定且可預測的位置時，測試程式更容易撰寫。

分離關注點（*separation of concerns*，*SoC*）

如前所述，Flux 清楚地將系統不同部分的關注點分開。這種分解使得系統更模組化、更容易維護，也更容易理解。每一個部分都有定義明確的作用，單向資料流則清楚地說明那些部分如何互動。

Flux 架構提供堅實的基礎來建構穩固、可擴展、易維護的 web 應用程式。它強調單向資料流、單一真相來源，和關注點分離，讓應用程式更容易開發、測試和偵錯。

總結：那麼，為什麼 React 這麼紅？

React 之所以這麼紅，是因為它可讓開發者以更容易預測和更可靠的方式來建立 UI，讓你用宣告的方式來表達你想在螢幕上顯示什麼，由 React 負責以高效率的做法，逐步更新 DOM 來實現它。它也鼓勵我們以組件為中心來思考，這可以幫助我們分離關注點，以及更輕鬆地重複使用程式碼。它經歷了 Meta 的實戰測試，並針對大規模使用而設計，而且是開源且免費的。

React 還擁有龐大且活躍的生態系統，開發者可以使用各種工具、程式庫和資源。這個生態系統包括用來測試、偵錯和優化 React 應用程式的工具，以及用來處理常見任務（例如資料管理、路由，和狀態管理）的程式庫。此外，React 社群有極高的參與度並非常積極地提供支援，有許多網路資源、討論區和社群可供開發者學習和成長。

React 是跨平台的，也就是說，它可以為各種平台（包括桌面、行動設備，和虛擬實境）建立 web 應用程式。因為有這種彈性，React 是為多個平台建構應用程式的理想工具，因為開發者可以用一個碼庫來建構可於多個設備上運行的應用程式。

總之，React 的核心價值主張是「基於組件的架構」、宣告性程式設計模型、虛擬 DOM、JSX、龐大的生態系統、跨平台性質，和 Meta 提供的支援。如果開發者想要建立快速、可擴展，和容易維護的 web 應用程式，React 是理想選擇。無論你要建立一個簡單的網站，還是複雜的企業級應用程式，相較於許多其他技術，React 都可以幫助你更有效率地實現目標。接著來回顧本章的內容。

伺服器端的 React

React 自問世以來已經有相當大的發展，雖然它最初是一款用戶端程式庫，但伺服器算繪（SSR）的需求也日漸增長，本章將帶你瞭解其中的原因。我們將一起探索伺服器端的 React，瞭解它與 client-only React 有何不同（譯按：client-only 是指「僅於用戶端算繪」，為了行文流暢，我們使用原文），以及如何用它來加強 React 應用程式。

正如之前的章節所討論的，Meta 開發 React 的初衷是為了製作高效率、可擴展的 UI。在第 3 章，我們看了它如何透過虛擬 DOM 來實現這一點，幫助開發者建立和管理 UI 組件。React 的用戶端策略在 web 上實現了快速、靈敏的使用者體驗。然而，隨著 web 不斷發展，用戶端算繪的限制也變得更加明顯。

用戶端算繪的限制

自從 React 在 2013 年以開源軟體的形式首次發布以來，它就一直被用來建立 UI。後來，這種做法的侷限性開始浮現，這些侷限性最終導致越來越多關注點被移到伺服器端。

SEO

用戶端算繪的主要限制之一，在於它可能讓搜尋引擎爬蟲無法正確地檢索內容，因為有些爬蟲不執行 JavaScript，或者，執行 JavaScript 的爬蟲可能無法以預期的方式工作。

考慮到各種搜尋引擎爬蟲的實作方式，以及許多爬蟲是專門的，且未向大眾公開，client-only 的做法能否接觸特定的網站或應用程式是有疑問的。

儘管如此，Search Engine Land 在 2015 年發表的一篇文章（*https://oreil.ly/r5hF2*），透過一些實驗來測試各種搜尋引擎如何處理 client-only 應用程式，它提到這些內容：

> 我們做了一系列測試來確定 *Google* 能否執行和檢索以多種方式來實現的 *JavaScript*。我們也確認了 *Google* 能夠算繪整個網頁並讀取 *DOM*，從而檢索動態生成的內容。

該文章發現，截至當時，Google 和 Bing 已經可以檢索 client-only 網站，但畢竟這只是廣闊且未知的私有（proprietary）大海中的一個研究專案。

因此，雖然 client-only app 可以和現代搜尋引擎順利合作，但它缺乏伺服器端的對應做法仍然有固有的風險。在傳統的 web 應用程式中，當使用者或搜尋引擎爬蟲請求網頁時，伺服器會算繪網頁的 HTML，並將它送回來，那些 HTML 包含所有內容、連結和資料，使搜尋引擎爬蟲能夠輕鬆地讀取和檢索內容，並產生搜尋結果，因為網頁的內容都只是文本，也就是標記。

然而，在「於用戶端算繪的應用程式」中（通常以 React 之類的程式庫或框架來建構），伺服器會回傳一個幾乎為空的 HTML 檔案，這種檔案的唯一任務是從同一個伺服器或另一個伺服器的另一個 JavaScript 檔案載入 JavaScript，然後 JavaScript 檔案會被下載至瀏覽器中執行，動態算繪網頁內容。雖然這種做法可以提供流暢的使用者體驗，很像原生的應用程式，但它有搜尋引擎優化（SEO）和效能方面的缺點，因為第一次請求並未下載對人類讀者有用的任何內容或標記，而是在網頁載入後立即發送另一個請求，以載入驅動整個網站的 JavaScript。這種做法稱為 network waterfall（網路瀑流）。

因此，client-only 算繪的另一個缺點是效能。我們來討論這一點。

效能

僅於用戶端算繪的應用程式可能有效能問題，尤其是在網路緩慢或設備效能較差的情況下。先下載、解析和執行 JavaScript 再算繪內容可能導致算繪時出現明顯的延遲，這個「多久才能開始互動」的時間是一項重要的指標，因為它直接影響使用者被吸引的程度和跳出率（bounce rate，使用者放棄瀏覽網頁的百分比）。如果網頁載入時間太長，使用者可能離開網頁，進而對網頁的 SEO 排名造成負面影響。

此外，如果設備的效能較低，CPU 能力有限，client-only 算繪也會造成使用者體驗下降，因為設備可能沒有足夠的能力可以快速執行 JavaScript，導致程式緩慢且遲鈍，可能造成使用者不悅，體驗不佳。如果能夠在伺服器端執行 JavaScript 並向用戶端發送最少量的資料或標記，低效能設備就不需要做太多工作，從而提供更好的使用者體驗。

若要探索 app 用戶端包裹的 hydration 步驟，它就像這樣：

```
// 匯入必要的程式庫
import React from "react";
import { hydrateRoot } from "react-dom/client";
// 假設 App 是 app 的主要組件
import App from "./App";

// 在用戶端 hydrate app
hydrateRoot(document, <App />);
```

使用伺服器算繪和用戶端 hydration 之後，app 有完整的互動功能，可以回應使用者的輸入、提取資料，並在必要時更新 DOM。

於 React 中的伺服器算繪 API

在上一節，我們自行使用 Express 和 `ReactDOMServer` 來將伺服器算繪加入 client-only React app。具體來說，我們使用 `ReactDOMServer.renderToString()` 來將 React app 算繪成 HTML 字串。這是將伺服器算繪加入 React app 的基本方法。然而，你也可以用其他方法來將伺服器算繪加入 React app。接下來要深入研究 React 提供的伺服器算繪 API，並瞭解何時以及如何使用它們。

我們接下來要討論 `renderToString` API，探索它的用法、優點、缺點，以及何時適合在 React 應用程式中使用它。具體來說，我們要研究：

- 它是什麼
- 它是如何運作的
- 它如何在日常使用 React 時融入其中

首先，我們來談談它是什麼。

renderToString

`renderToString` 是 React 提供的伺服器算繪 API，它可以讓你在伺服器將一個 React 組件算繪成 HTML 字串。這個 API 是同步的，它會回傳一個完整算繪的 HTML 字串，之後可以當成回應來傳送到用戶端。伺服器算繪的 React 應用程式經常使用 `renderToString` 來提升效能、SEO 和便利性。

本章回顧

本章的結論是，伺服器算繪和 hydration 是強大的技術，可以明顯改善 web 應用程式的效能、使用者體驗，和 SEO。React 提供豐富的伺服器算繪 API，例如 `renderToString` 和 `renderToPipeableStream`，每一個 API 都有其優勢和取捨。瞭解這些 API 並根據應用程式的大小、伺服器環境和開發經驗…等因素來選擇適當的 API，可以優化 React 應用程式的伺服器端和用戶端效能。

我們可以在本章看到，`renderToString` 是一個適用於小型應用程式且簡單直覺的伺服器算繪 API。然而，由於它具有同步性質，且可能阻塞事件迴圈，對於大型應用程式來說，它可能不是最有效率的選擇。另一方面，`renderToPipeableStream` 是一款更高階、更靈活的 API，可讓你更仔細地控制算繪過程，並改善與其他 Node.js 串流的整合，因而比較適合大型應用程式。

複習問題

充分瞭解 React 的伺服器算繪和 hydration 之後，請回答下面的問題來檢驗你的知識。能夠自信地回答這些問題就代表你已經充分地理解 React 裡的機制，可以放心地繼續閱讀。如果不行，建議你仔細地再次閱讀，雖然這不會影響你繼續閱讀本書的體驗。

1. 在 React 應用程式中使用伺服器算繪的主要優勢是什麼？
2. hydration 在 React 中是如何工作的？為什麼它很重要？
3. 什麼是 resumability？它聲稱比 hydration 好在哪裡？
4. client-only 算繪的主要優勢和缺點是什麼？
5. React 的 `renderToReadableStream` 和 `renderToPipeableStream` API 的主要區別是什麼？

下回預告

掌握伺服器算繪和 hydration 之後，你可以開始探索更高階的 React 開發主題了。在下一章，我們將深入研究並行 React。隨著 web 應用程式變得越來越複雜，處理非同步操作對於建立流暢的使用者體驗而言也越來越重要。

學會利用並行 React 有助於創造高效能、可擴展，且方便使用者的應用程式，輕鬆地處理複雜的資料互動。因此，保持專注，做好提升 React 技能的準備，我們將繼續深入探索並行 React 的領域！