React 與 TypeScript 深度理解

從「會寫」到「懂為什麼」

資料流向 · 重新渲染 · Hooks 原理 · 元件設計 · 型別與泛型

這份講義為誰而寫

寫給已經在工作中使用 React 與 TypeScript 一段時間、能照著範例完成需求，但希望真正理解「每段程式碼為什麼要這樣設計」的工程師。讀完之後，目標是讓你面對公司的程式碼時，能看懂設計背後的思路，並且有能力對自己的寫法做出有依據的取捨。

本講義的五個核心提問

- 資料在 React 應用程式裡是怎麼流動的？狀態應該放在哪裡？
- 重新渲染到底什麼時候發生？`memo`、`useMemo`、`useCallback` 在解決什麼問題？
- Hooks 為什麼被設計成這樣？`useEffect` 的依賴陣列真正的意義是什麼？
- 什麼是好的元件設計？該怎麼決定拆分與 props 介面？
- 型別、泛型、interface 在真實專案裡解決什麼問題？

版本 1.0 · 2026 年 8 月 · 內容以 React 18+ 與 TypeScript 5 為基準

目錄

課程導讀 **5**

1 React 的心智模型：UI 是狀態的函數 **8**

1.1 命令式的困境：手動把畫面「修」成對的樣子 8

1.2 宣告式：只描述「每個狀態下畫面長什麼樣」 9

1.3 Virtual DOM 與協調：讓「整個重畫」變得付得起 10

1.4 不可變性：不是程式碼潔癖，是變更偵測的協議 11

1.5 把模型帶著走 12

自我檢測 12

2 資料流向：單向資料流與狀態的位置 **14**

2.1 單向資料流：資料往下，事件往上 14

2.2 狀態該放在哪一層 15

2.3 不要儲存算得出來的東西：衍生資料 16

2.4 Context：跨層傳遞的通道，不是全域變數 17

2.5 客戶端狀態與伺服器狀態 18

自我檢測 19

3 重新渲染的時機：何時發生、如何控制 **20**

3.1 觸發重新渲染的三個來源 20

3.2 一次渲染是一張快照 21

3.3 協調與 key：React 怎麼認出「同一個」元件 21

3.4 React.memo：把「傳染」擋在邊界上 23

3.5 useMemo 與 useCallback：讓引用跨渲染存活 23

3.6 Context 與重新渲染 24

自我檢測 25

4 Hooks：為什麼存在、為什麼長這樣 **27**

4.1 Hooks 之前的世界：class 元件的三個痛點 . . . 27
4.2 為什麼 hooks 有那些「奇怪的規則」 . . . 28
4.3 useState 進階：狀態存在哪裡、閉包會過期 . . . 29
4.4 useEffect：是「同步」，不是「生命週期」 . . . 30
4.5 useRef：不參與渲染的可變盒子 . . . 31
4.6 自訂 Hook：邏輯復用的正確單位 . . . 32
4.7 useReducer：把「狀態怎麼變」集中成一張規則表 . . . 33
自我檢測 . . . 34

5 元件設計：拆分、組合與 API 思維 35
5.1 好元件的三個可檢驗特徵 . . . 35
5.2 Props 設計：從使用端往回設計 . . . 36
5.3 組合優於設定：children 與插槽 . . . 36
5.4 受控與非受控：狀態主導權的設計決策 . . . 38
5.5 邏輯與畫面分離：hook 管邏輯，元件管長相 . . . 38
5.6 Compound Components：一組元件共享隱式狀態 . . . 39
5.7 讓不可能的狀態無法出現 . . . 40
自我檢測 . . . 41

6 TypeScript 型別系統：從「標註」到「建模」 42
6.1 型別在做什麼：把值分成集合 . . . 42
6.2 結構型別：看形狀，不看名字 . . . 43
6.3 type 與 interface：怎麼選 . . . 43
6.4 聯集與收窄：讓編譯器跟著你的 if 走 . . . 44
6.5 標記聯集：實戰中最重要的型別模式 . . . 44
6.6 泛型：型別層級的參數 . . . 45
6.7 讓值成為型別的唯一來源：typeof、keyof、as const . . . 46
6.8 Utility Types：型別的標準函式庫 . . . 47
6.9 any、unknown、never：三個特殊集合 . . . 48
自我檢測 . . . 48

7 TypeScript × React：型別的實戰應用 50

7.1 Props 的型別：元件的公開合約 . . . 50
7.2 事件型別：用 IDE 查，不用背 . . . 51
7.3 useState 與 useRef 的型別 . . . 51
7.4 泛型元件：讓「一份 UI 邏輯」適用所有資料型別 . . . 52
7.5 用標記聯集設計 props：把使用規則寫進型別 . . . 52
7.6 包裝原生元素：ComponentPropsWithoutRef . . . 53
7.7 自訂 hook 的回傳型別 . . . 53
7.8 API 層：型別的單一來源與執行期的守門員 . . . 54
自我檢測 . . . 55

8 元件實戰：從零打造一個浮動視窗 57
8.1 需求與介面：先設計「用起來的樣子」 . . . 57
8.2 Portal：DOM 出走，資料流留下 . . . 58
8.3 位置狀態與第一版拖曳：同步模型的推導 . . . 60
8.4 setPointerCapture：讓事件回到元素身上 . . . 62
8.5 每秒上百次 setPosition：誰在重新渲染？ . . . 63
8.6 疊層：點一下，浮到最上面 . . . 64
8.7 縮放與 usePointerDrag：同一支舞，只跳一次 . . . 65
8.8 鍵盤、觸控與可及性 . . . 66
8.9 完整組裝與決策對帳 . . . 67
8.10 距離元件庫還有多遠 . . . 70
自我檢測 . . . 71

9 綜合實戰：把觀念組裝成一個完整功能 72
9.1 分層與檔案結構 . . . 72
9.2 types.ts：先把「什麼是合法的」說清楚 . . . 72
9.3 api.ts：請求邏輯寫一次 . . . 73
9.4 useUsers.ts：這一頁「怎麼運作」的全部 . . . 73
9.5 呈現層：UserTable 與搜尋框 . . . 74
9.6 UserPage.tsx：接線，然後追蹤一次資料流 . . . 75
9.7 閱讀公司程式碼的提問清單 . . . 76
9.8 常見 code smell 對照表 . . . 77

9.9 接下來的路 77
自我檢測 78

附錄：速查表、練習與資源 **79**

A 「該用哪個 hook」決策速查 79
B 高頻型別速查 80
C 動手練習：把公司程式碼當教材 80
D 延伸資源 81

課程導讀

你現在在哪裡，這份講義要帶你去哪裡

用了三個月的 React 與 TypeScript 之後，多數人會停在一個共同的位置：`useState`、`useEffect`、`useMemo`、`useCallback` 都會用，照著公司現有的範例可以把需求做完，但心裡其實有一連串沒有答案的問題——

- 為什麼這個狀態放在這一層，而不是放在用到它的那個元件裡？
- 為什麼這裡要包 `useCallback`，隔壁看起來一樣的函式卻沒包？
- 為什麼資深同事看到 `useEffect` 裡 `setState` 會皺眉頭？
- 這個泛型 `<T>` 到底在參數化什麼？拿掉會怎樣？

這些問題的共同點是：它們都不是「語法問題」，而是「設計問題」。語法查文件三分鐘就有答案；設計的理由卻很少被寫下來，因為它藏在 React 與 TypeScript 本身的設計取捨裡。這份講義的目標，就是把這些取捨攤開來講清楚——**讓你之後看公司的程式碼時，能還原出「當初為什麼這樣寫」的推理過程**。

內容地圖

全書分成四個部分，順序刻意安排成「先建立模型，再談實踐」：

部分	章節	回答的核心問題
React 執行模型	第 1 章 心智模型	React 為什麼是「宣告式」？UI = f(state) 是什麼意思？
	第 2 章 資料流向	資料怎麼流動？狀態該放在哪裡？
	第 3 章 重新渲染	渲染何時發生？`memo` 系列工具在解決什麼？
	第 4 章 Hooks	Hooks 為什麼存在？`useEffect` 的正確心智模型是什麼？
設計實踐	第 5 章 元件設計	怎麼設計 props？何時拆元件？什麼是好的元件 API？
TypeScript	第 6 章 型別系統	型別、泛型、interface 背後的模型是什麼？
	第 7 章 TS × React	型別如何落地到元件、hooks 與 API 層？
實戰	第 8 章 浮動視窗	用 portal、全域事件、焦點與拖曳，從零打造一個完整的互動元件。
	第 9 章 綜合實戰	把全部觀念組裝成一個完整功能，並學會閱讀既有程式碼。

建議的使用方式

每章三步驟。每一章都建議走完這個循環：

1. **讀懂主張**：每章都圍繞少數幾個核心主張（用「核心觀念」框標出），先確定你能用自己的話複述。
2. **跟著程式碼推理**：範例刻意寫成「公司程式碼會出現的樣子」，讀的時候不要只看結果，要能說出「換一種寫法會發生什麼事」。
3. **回到公司的程式碼找對應**：這是最重要的一步。每章的「自我檢測」之後，打開你正在維護的專案，找出一段用得上本章觀念的程式碼，試著解釋它的設計（或指出它的問題）。

建議進度。一週一章、九週完成是舒服的節奏；第 3、4、6 章份量較重，必要時可以各拆成兩週。如果你想快速補齊某個主題，各章也可以獨立閱讀，但第 1 章的心智模型是全書的地基，建議無論如何先讀。

動手環境。準備三樣工具：一個可以隨手實驗的專案（`npm create vite@latest --template react-ts`）、瀏覽器的 React DevTools 擴充功能（第 3 章會用到 Profiler）、以及 TypeScript Playground（`https://www.typescriptlang.org/play`，第 6 章的型別實驗都可以在這裡做）。

版式說明

內文會穿插四種資訊框，各有明確的用途：

資訊框	用途
核心觀念（藍）	本章的關鍵主張，值得反覆咀嚼、能用自己的話複述的程度。
常見誤解（紅）	實務上流傳很廣、但不正確或不精確的說法，以及正確的版本。
實務建議（綠）	可以直接帶回工作中的具體做法與判斷準則。
深入原理（紫）	實作層面的補充，跳過不影響主線，但讀懂會讓你更有把握。

術語約定

為了閱讀順暢，全書採以下用語：render 譯為「渲染」、re-render 為「重新渲染」、component 為「元件」、state 為「狀態」；props、hook、context、effect 等已成慣用語的名詞保留英文。程式碼註解使用中文，變數命名維持英文，與多數公司的實際情況一致。

第 1 章

React 的心智模型：UI 是狀態的函數

本章目標——讀完後你應該能回答

- 「宣告式」和「命令式」的差別是什麼？React 為什麼選擇宣告式？
- Virtual DOM 存在的真正目的是什麼？（提示：不是「比較快」）
- 為什麼直接修改（mutate）狀態，畫面不會更新？
- 「渲染」和「更新 DOM」是同一件事嗎？

React 所有的設計——單向資料流、hooks、不可變性、`memo`——都是從同一個核心決定推導出來的。這一章先把這個核心講清楚，後面每一章遇到「為什麼要這樣」的時候，答案幾乎都能追溯回這裡。

1.1 命令式的困境：手動把畫面「修」成對的樣子

先看一段沒有框架的程式碼。需求很日常：一個登入表單，email 和密碼都有值時按鈕才能按，送出時按鈕顯示「登入中…」並停用，失敗時顯示錯誤訊息。

```
// 命令式（vanilla JS）：每個事件都要「手動修畫面」
emailInput.addEventListener('input', () => {
  submitBtn.disabled = !emailInput.value || !passwordInput.value;
  errorBox.hidden = true;            // 別忘了：使用者重新輸入時要清掉舊錯誤
});
passwordInput.addEventListener('input', () => {
  submitBtn.disabled = !emailInput.value || !passwordInput.value;
  errorBox.hidden = true;            // 同一段邏輯，第二份複本
});
submitBtn.addEventListener('click', async () => {
  submitBtn.disabled = true;
  submitBtn.textContent = '登入中…';
  try {
    await login(emailInput.value, passwordInput.value);
  } catch (e) {
    errorBox.hidden = false;
    errorBox.textContent = '帳號或密碼錯誤';
    submitBtn.textContent = '登入';
    submitBtn.disabled = false;      // 漏掉這行，按鈕就永遠鎖死
  }
```

```
});
```

注意每個事件處理器在做的事：它必須知道「畫面現在可能長什麼樣」，然後下達一連串 DOM 指令，把畫面**從目前的樣子修正到應該有的樣子**。這就是命令式（imperative）：你描述的是「怎麼改」，而不是「該長怎樣」。

問題在於規模。畫面的樣子由多個狀態共同決定——這個例子裡至少有「兩個欄位是否有值」「是否送出中」「是否有錯誤」。k 個獨立的布林狀態就有 2^k 種可能的畫面；而命令式程式要維護的不是這 2^k 種畫面本身，是**畫面之間的轉移**：每個事件發生時，從任何一種畫面出發都要能改到正確的下一種。狀態一多，轉移數量的成長遠比畫面數量快，漏寫任何一條（例如上面「漏掉重新啟用按鈕」）就是一個 bug。jQuery 時代大型前端難以維護，根本原因就是這個。

1.2　宣告式：只描述「每個狀態下畫面長什麼樣」

同樣的需求，React 的寫法：

```tsx
function LoginForm() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [submitting, setSubmitting] = useState(false);
  const [error, setError] = useState<string | null>(null);

  async function handleSubmit() {
    setSubmitting(true);
    setError(null);
    try {
      await login(email, password);
    } catch {
      setError('帳號或密碼錯誤');
    } finally {
      setSubmitting(false);
    }
  }

  const canSubmit = email !== '' && password !== '' && !submitting;

  return (
    <form>
      <input value={email} onChange={(e) => setEmail(e.target.value)} />
      <input value={password} onChange={(e) => setPassword(e.target.value)} />
      {error !== null && <p className="error">{error}</p>}
      <button disabled={!canSubmit} onClick={handleSubmit}>
        {submitting ? '登入中…' : '登入'}
      </button>
    </form>
```

```
  );
}
```

結構上發生了一個關鍵的職責分離：

- **事件處理器只改狀態**。`handleSubmit` 裡沒有任何一行碰畫面，它只負責描述「發生了什麼事、狀態因此變成什麼」。
- **回傳值描述完整畫面**。JSX 的部分是一個純粹的對應關係：給定這一刻的狀態，畫面就長這樣。按鈕該不該停用、錯誤訊息在不在，全部是從狀態**算出來**的，不需要「記得去改」。

「使用者重新輸入時要清掉舊錯誤」這種容易漏的轉移邏輯去哪了？它消失了——因為畫面不再被「修改」，而是每次都從狀態重新推導，只要狀態對，畫面就不可能停留在舊的樣子。你維護的東西從 $O(\text{狀態} \times \text{事件})$ 條轉移規則，變成 $O(\text{狀態})$ 條對應規則加上每個事件對狀態的影響。

核心觀念：UI = f(state)

React 的核心主張是把介面寫成狀態的**純函數**：渲染就是「用目前的狀態計算畫面的描述」，事件處理就是「更新狀態」。兩者徹底分離——事件不碰畫面，渲染不碰狀態。你之後在公司程式碼看到的絕大多數設計慣例，都是在維護這條公式的成立。

這也直接解釋了公司程式碼的幾個「長相」：

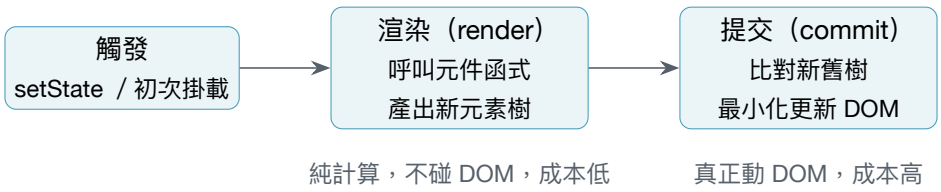
- **為什麼 JSX 裡到處是三元運算子、`&&`、`map`**——因為渲染是「計算」，條件與迴圈自然用運算式表達，而不是用指令去增刪節點。
- **為什麼元件函式頂部常有一排 `const`**——那些是「每次渲染都重新計算的中間值」（例如上面的 `canSubmit`），它們不是狀態，只是狀態的推導結果。這個區分在第 2 章會正式展開。
- **為什麼事件處理器通常短小、幾乎只呼叫 `setXxx`**——命令式的部分被刻意壓縮到最小。

1.3　Virtual DOM 與協調：讓「整個重畫」變得付得起

「每次狀態改變就重新計算整個畫面」聽起來很浪費。如果每次都真的把 DOM 全部拆掉重建，效能確實不可接受——DOM 操作牽動瀏覽器的版面計算（layout）與重繪（paint），是前端最昂貴的操作之一。

React 的解法是在你和真實 DOM 之間插入一層便宜的描述：

1. 你的元件函式回傳的不是 DOM，而是 **React 元素樹**——一種輕量的 JavaScript 物件，描述「畫面應該長什麼樣」。JSX 只是建立這些物件的語法糖（`<div>` 會被編譯成對 `jsx('div', ...)` 的呼叫）。建立幾千個 JS 物件的成本，遠低於操作幾個 DOM 節點。
2. React 把新的元素樹跟上一次的樹做比對（這個過程叫**協調**，reconciliation），找出真正有差異的地方。
3. 只把差異的部分套用到真實 DOM 上。



常見誤解：Virtual DOM 是為了「比直接操作 DOM 快」

不是。針對特定更新手工寫出的精準 DOM 操作，永遠可以比「產生整棵樹再 diff」更快——diff 本身就是額外成本。Virtual DOM 真正的目的，是讓「每次都整個重算」這個 **好維護的心智模型**，在效能上變得可以接受。React 用一點執行期成本，換走了你手上所有「手動同步畫面」的維護成本。理解這一點，你才能正確看待第 3 章的效能優化：優化的對象是「計算太多次」，而不是「DOM 更新太多次」——後者 React 已經替你最小化了。

另外一個值得記住的細節：JSX 產生的元素是「描述」而不是「實體」。`<UserCard />` 這行執行時只是建立一個 `{ type: UserCard, props: ... }` 的物件，**React 才是真正決定何時呼叫 `UserCard` 函式的人**。這就是為什麼元件函式的執行時機不由你控制、為什麼它必須保持「純」——同樣的 props 和狀態要算出同樣的結果，不能依賴呼叫順序或次數（第 3 章的 StrictMode 會再回到這件事）。

1.4　不可變性：不是程式碼潔癖，是變更偵測的協議

三個月的 React 經驗裡，你多半已經被糾正過這種寫法：

```
// 錯誤：直接 mutate，畫面不會更新
function addTodo(todo: Todo) {
  todos.push(todo);      // 修改了原陣列
  setTodos(todos);       // 傳回去的還是「同一個」陣列
}

// 正確：建立新陣列
function addTodo(todo: Todo) {
  setTodos([...todos, todo]);
}
```

為什麼第一種不行？這不是語法規定，而是 React 變更偵測策略的直接結果。React 需要在很多地方快速回答「這個值變了嗎」：`setState` 之後要不要重新渲染、`useEffect` 的依賴變了沒、`memo` 的 props 變了沒。它一律用同一個辦法：**`Object.is` 引用比較**——比「是不是同一個物件」，不比內容。

為什麼選擇引用比較？

理論上 React 也可以深度比較物件內容，或像某些框架用 Proxy 攔截每一次屬性修改。但深度比較的成本隨資料大小線性成長，大型狀態每次更新都深比會拖垮效能；攔截式的響

應系統則讓「什麼時候會觸發更新」變得隱晦。React 選了最便宜、最可預測的協議：**開發者用「新的引用」來宣告「內容變了」**。引用比較是 $O(1)$，代價是把「建立新物件」的責任交給你。所以不可變更新不是風格偏好，而是你和 React 之間的通訊協定——mutate 等於對 React 說「什麼都沒變」。

常用的不可變更新模式

- 改物件欄位：`setUser({ ...user, name: newName })`
- 陣列新增：`setItems([...items, newItem])`
- 陣列刪除：`setItems(items.filter((i) => i.id !== id))`
- 陣列更新單筆：`setItems(items.map((i) => i.id === id ? { ...i, done: true } : i))`

巢狀太深導致 spread 展開痛苦時，通常代表狀態結構該攤平了，或者考慮用 Immer（公司若使用 Redux Toolkit，其內建的正是 Immer）。

1.5 把模型帶著走

這一章建立的模型可以濃縮成三句話，後面每一章都會反覆用到：

1. **渲染是計算**：元件函式用目前的狀態算出畫面描述，這個計算應該是純的。
2. **事件改狀態，不改畫面**：畫面永遠是推導結果，沒有「順手改一下 DOM」這種事。
3. **新引用等於有變化**：React 用 `Object.is` 判斷一切「變了沒」的問題。

本章重點回顧

- 命令式維護「狀態之間的轉移」，數量隨狀態數爆炸；宣告式只維護「狀態到畫面的對應」加上「事件對狀態的影響」。
- UI = f(state)：渲染是純計算，事件處理只更新狀態，兩者分離。
- Virtual DOM 的目的不是絕對速度，而是讓宣告式模型的成本可接受；「渲染」（算描述）和「提交」（動 DOM）是兩個階段。
- React 用 `Object.is` 引用比較偵測變化，所以更新必須產生新引用；mutate 對 React 而言等於「沒變」。

自我檢測

1. **同事問你：「React 是宣告式的」到底是什麼意思？**
   參考答案：命令式描述「怎麼一步步修改畫面」，宣告式描述「每種狀態下畫面該長什麼樣」。React 元件回傳的 JSX 就是後者：它是狀態到畫面的對應函數，畫面之間的轉移由 React 負責，開發者只維護狀態。

2. **「用了 Virtual DOM 所以 React 比較快」這句話哪裡有問題？**
   參考答案： Virtual DOM 有 diff 成本，針對單一更新永遠不會比手寫精準 DOM 操作快。它的價值是讓「每次整個重算」的宣告式模型在效能上可行，用小量執行期成本換取大量維護成本的下降。

3. **為什麼 `user.name = 'Amy'; setUser(user)` 之後畫面沒更新？**
   參考答案： `setUser` 收到的引用與目前狀態是同一個物件，React 用 `Object.is` 比較認定「沒有變化」，直接略過重新渲染。必須傳入新物件：`setUser({ ...user, name: 'Amy' })`。

4. **「這個元件渲染了三次」等於「DOM 被改了三次」嗎？**
   參考答案：不等於。渲染是呼叫元件函式產出新的元素樹（純計算）；之後 React 比對新舊樹，只有真正有差異的節點才會在 commit 階段更新 DOM。三次渲染若算出相同結果，DOM 可能一次都沒動。

第 2 章

資料流向：單向資料流與狀態的位置

本章目標——讀完後你應該能回答

- 「單向資料流」具體指什麼？它換來了哪些好處？
- 決定「狀態放在哪一層」的原則是什麼？
- 什麼樣的資料**不該**成為狀態？
- Context 解決什麼問題，代價是什麼，何時不該用？

第 1 章講的是單一元件內部：狀態算出畫面。真實應用有幾十層元件，這一章回答的問題是：**狀態在元件樹裡怎麼流動、應該住在哪裡**。公司程式碼裡那些「為什麼狀態放這層」「為什麼要傳 `onXxx` 回呼」的疑惑，答案都在這裡。

2.1 單向資料流：資料往下，事件往上

React 的規則只有一條，但貫徹得非常徹底：

核心觀念：單向資料流

狀態由某一個元件**擁有**；資料透過 props 往下傳給子孫，子孫想改變資料時，只能呼叫父層傳下來的回呼函式「往上報告事件」，由擁有者決定狀態怎麼變。資料的流向永遠是單向的：**向下傳值、向上傳事件**。




```tsx
function SearchPage({ products }: { products: Product[] }) {
  const [keyword, setKeyword] = useState('');
  const filtered = products.filter((p) => p.name.includes(keyword));

  return (
    <>
      <SearchInput value={keyword} onChange={setKeyword} />
      <ProductList items={filtered} />
```

```
    </>
  );
}

// 子元件不擁有資料，只負責顯示與回報
function SearchInput({ value, onChange }: {
  value: string;
  onChange: (next: string) => void;
}) {
  return <input value={value} onChange={(e) => onChange(e.target.value)} />;
}
```

為什麼要規定得這麼死？因為它買到三個對大型程式至關重要的性質：

- **除錯有固定路徑**。畫面錯了→找到顯示這個值的元件→往上找到狀態的擁有者→能改這個狀態的只有擁有者手上的 `setXxx`，全部呼叫點一目瞭然。對比雙向綁定盛行的年代（如 Angular 1）：任何一層都可能改資料，一個值被改動時你無法回答「是誰改的」，改一處會連鎖觸發十處。
- **元件可以獨立理解**。`SearchInput` 的行為完全由 props 決定，不需要知道外面的世界；同一個元件因此可以放到任何地方重用。
- **資料只有一份**。`keyword` 只存在於 `SearchPage`，輸入框顯示的、清單過濾用的，保證是同一個值，不存在「兩邊不同步」的問題。這就是**單一事實來源**（single source of truth）。

順帶解釋一個常見的命名慣例：props 介面上的回呼叫 `onXxx`（事件語意，「發生了什麼」），元件內部的實作叫 `handleXxx`（處理語意）。子元件說「發生了 change」，父元件決定「所以狀態要怎麼變」——**子元件回報事實，父元件決定政策**。這個分工讓子元件不必知道自己被用在什麼場景。

2.2　狀態該放在哪一層

實務上最常見的資料流問題就是「狀態放錯層」。判斷原則只有兩條，而且有優先順序：

1. **預設放在最靠近使用處的地方**（colocation）。只有一個元件用到的狀態，就放在那個元件裡。好處：改動的影響範圍小、重新渲染的範圍小（第 3 章詳述）、元件搬移或刪除時狀態跟著走，不留孤兒。
2. **多個元件需要同一份狀態時，提升到它們最近的共同祖先**（lifting state up）。上面的 `SearchPage` 就是例子：輸入框和清單都依賴 `keyword`，所以它住在兩者的共同父層。

新手常犯的錯是反過來：讓兩個元件「各自持有一份」，再想辦法同步。

```
// 反模式：兄弟元件各自持有狀態，再用 effect 互相同步
function SearchInput({ onKeywordChange }: Props) {
  const [keyword, setKeyword] = useState('');   // 第一份
```

```
  useEffect(() => {
    onKeywordChange(keyword);                    // 用副作用把值「推」出去
  }, [keyword, onKeywordChange]);
  // ...
}
function SearchPage() {
  const [keyword, setKeyword] = useState('');    // 第二份（複本）
  // ...
}
```

兩份狀態代表兩個事實來源：它們會有短暫不一致的瞬間（effect 在渲染之後才跑）、多一次不必要的渲染，而且任何一邊多了新的更新路徑，同步邏輯就得跟著長。正確做法永遠是：**把狀態提升到共同祖先，讓下面的元件變成「無狀態的顯示者」**。如果提升之後發現要穿過很多層，那是另一個問題（本章後段的 Context 與第 5 章的組合技巧處理它），不要用「多存一份」來逃避。

判斷流程

為新資料決定住處時，依序問：(1) 只有一個元件用？→放那個元件。(2) 幾個鄰近元件共用？→提升到最近共同祖先。(3) 整個應用程式到處都要（登入者、主題、語系）？→ Context。(4) 它其實是伺服器資料的快取？→交給資料抓取層（本章 2.5 節）。(5) 它需要「重新整理後還在」或「可以分享網址」？→放 URL（query string），不放 React。

2.3　不要儲存算得出來的東西：衍生資料

狀態設計第二常見的錯誤：把**可以從現有狀態算出來的值**也存成狀態。這是公司程式碼審查裡出現頻率最高的問題之一，值得完整看一次。

```
// 反模式：filtered 是 keyword + products 的推導結果，卻被存成獨立狀態
const [keyword, setKeyword] = useState('');
const [filtered, setFiltered] = useState<Product[]>([]);

useEffect(() => {
  setFiltered(products.filter((p) => p.name.includes(keyword)));
}, [products, keyword]);
```

這段程式碼「能動」，但有三個實質問題：

- **多一次渲染**：`keyword` 改變→渲染一次（此時 `filtered` 還是舊的！）→ effect 執行→ `setFiltered` →再渲染一次。中間那次渲染，畫面上是「新關鍵字配舊清單」的不一致狀態。
- **兩個事實來源**：`filtered` 的正確性依賴「有人記得同步它」。未來若新增清空、重設等更新路徑，每一條都要記得維護同步，漏一條就是 bug。
- **閱讀成本**：讀者必須追完 effect 才知道 `filtered` 從哪來。

正確版本在第 1 章其實已經出現過——衍生資料就在渲染時直接算：

```
const [keyword, setKeyword] = useState('');
const filtered = products.filter((p) => p.name.includes(keyword));
// 覺得計算昂貴時才加上 useMemo（先量測再說，見第 3 章）：
// const filtered = useMemo(
//   () => products.filter((p) => p.name.includes(keyword)),
//   [products, keyword]
// );
```

核心觀念：最小狀態原則

狀態是「無法從其他資料推導出來的最小事實集合」。每次要寫 `useState` 前先問：這個值能不能從現有的 props 或狀態算出來？能，就用 `const` 在渲染時算；不能，才配得上一個 state。狀態越少，需要保持一致的東西越少，bug 的棲息地就越小。

順帶一提，「用 `useEffect` 同步兩個狀態」幾乎永遠是這個反模式的變形。你會在第 4 章看到 `useEffect` 的正當用途清單，「改了 A 狀態之後去改 B 狀態」不在其中。

2.4　Context：跨層傳遞的通道，不是全域變數

單向資料流有個實際的痛點：狀態放得高、用得深，中間每一層都要幫忙轉傳 props——即使那一層自己根本不關心這個值。這叫 **prop drilling**。五層十層地轉傳登入者資訊，每個中間元件的介面都被污染，重構時牽一髮動全身。

Context 就是為此存在的：**讓深處的元件直接讀取遠祖先提供的值**，跳過中間層。

```
// auth-context.tsx —— 公司程式碼裡的典型寫法
interface AuthContextValue {
  user: User | null;
  logout: () => void;
}
const AuthContext = createContext<AuthContextValue | null>(null);

export function AuthProvider({ children }: { children: React.ReactNode }) {
  const [user, setUser] = useState<User | null>(null);
  // ...登入、登出邏輯...
  return (
    <AuthContext.Provider value={{ user, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

// 慣例：用自訂 hook 包住 useContext，並擋掉「忘了包 Provider」的誤用
```

```ts
export function useAuth(): AuthContextValue {
  const ctx = useContext(AuthContext);
  if (ctx === null) {
    throw new Error('useAuth 必須在 <AuthProvider> 之內使用');
  }
  return ctx;
}
```

這段範本值得注意兩個設計：context 的預設值給 `null` 而不是假資料，配合 `useAuth` 裡的檢查，「忘記包 Provider」會在開發期立刻炸出明確錯誤，而不是默默用假資料渲染；以及**對外只暴露 `useAuth`**，使用端完全不需要知道 context 物件本身——這是「用 hook 封裝實作細節」的標準手法，第 4 章會正式討論。

但 Context 有兩個常被忽略的代價：

- **更新的廣播成本**：Provider 的 `value` 一變（依然是 `Object.is` 比較），**所有**讀取這個 context 的元件都會重新渲染，無論它用到的欄位有沒有變。所以 context 適合「低頻變動」的資料：登入者、主題、語系、權限。高頻變動的資料（輸入中的表單值、滑鼠位置）放進 context 會讓整個訂閱網一起抖動。
- **隱式依賴**：元件透過 props 拿資料時，依賴寫在介面上，一眼可見；透過 context 拿資料時，依賴藏在實作裡，元件也因此離不開特定的 Provider 環境（測試時得多包一層）。

常見誤解：「要避免 prop drilling，所以都用 Context」

穿透兩三層的 props 不是問題，是單向資料流的正常樣貌——依賴明確、可追蹤，成本只是多打幾個字。Context 是為「幾乎整棵樹都需要、且變動不頻繁」的資料準備的例外通道。此外，第 5 章會示範另一個常被遺忘的解法：用**元件組合**（把組好的元件當 children 傳下去）讓中間層根本不需要轉傳資料。

2.5　客戶端狀態與伺服器狀態

最後一個對讀懂公司程式碼很關鍵的區分。很多專案會用 React Query、SWR 或 RTK Query 這類函式庫，新手常困惑：「這跟我自己 `useState` + `useEffect` 抓資料差在哪？」

差別在於**資料的擁有者**。「彈窗開不開」的事實來源就在前端，你設什麼就是什麼；但「使用者列表」的事實來源在伺服器——前端存的永遠只是**某個時間點的快取**。快取就帶來一整組固有問題：什麼時候算過期？兩個元件同時要這份資料，要抓兩次嗎？使用者切換分頁回來要不要重新驗證？更新成功後相關列表怎麼失效？這些問題與「元件狀態」無關，用 `useState` 硬做，就是在自己重新發明一個快取系統（而且通常漏掉一半的邊界情況）。

資料	特徵	建議住處
輸入中的表單值、彈窗開關	單一元件使用、生命週期短	元件內 `useState`
搜尋關鍵字＋結果清單	鄰近元件共用	提升到共同祖先
登入者、主題、語系	全域、低頻變動	Context
目前頁碼、篩選條件	需要可分享、可重新整理	URL query string
商品列表、使用者資料	事實來源在伺服器（快取）	React Query 等資料層

讀公司程式碼時，先分清楚眼前的狀態屬於哪一類，再判斷它的住處合不合理——很多「這程式碼好亂」的直覺，追根究柢是伺服器狀態被當成客戶端狀態管理了。

本章重點回顧

- 單向資料流：狀態有唯一擁有者，資料經 props 向下，事件經回呼向上；換來可追蹤、可重用、單一事實來源。
- 狀態位置：預設 colocation，共用時提升到最近共同祖先；不要複製狀態再同步。
- 最小狀態原則：算得出來的就不要存，衍生資料在渲染時計算。
- Context 是低頻全域資料的跨層通道；它有廣播式重渲染與隱式依賴的代價，不是 prop drilling 的萬靈丹。
- 伺服器資料本質是快取，交給專門的資料層管理，別用 `useState` 重新發明。

自我檢測

1. **為什麼 React 不允許子元件直接修改 props？**
   參考答案： props 唯讀是單向資料流的基石：狀態只能被擁有者透過 `setState` 修改，所以任何值的變化都能追溯到唯一的源頭。若子元件能改 props，資料的變動來源就不再唯一，「誰改的」變得不可考，除錯與推理的基礎就崩了。

2. **你接手的元件裡有 `useEffect` 監聽狀態 A 去 `set` 狀態 B，該怎麼判斷它有沒有問題？**
   參考答案：先問 B 是否能從 A（與其他資料）直接算出來。可以的話這是衍生資料反模式：改成渲染時計算（必要時 `useMemo`），可消除多餘渲染與不同步窗口。只有 B 確實是獨立事實（無法推導）時，才考慮其他設計。

3. **什麼樣的資料適合放 Context？舉一個不適合的例子。**
   參考答案：適合：整棵樹大範圍需要、低頻變動，例如登入者、主題、語系。不適合：高頻變動的資料，例如正在輸入的表單內容——value 每次改變都會讓所有 consumer 重新渲染。

4. **同事說「這個列表資料我用 useState 存起來就好，幹嘛引 React Query」，你怎麼回應？**
   參考答案：列表的事實來源在伺服器，前端存的是快取。自己管快取意味著自己處理過期、重新驗證、去重、失效、載入與錯誤狀態等問題；資料層函式庫就是把這組固有問題標準化。小專案可以手寫，但要意識到自己在承擔哪些責任。

第 3 章

重新渲染的時機：何時發生、如何控制

本章目標——讀完後你應該能回答

- 重新渲染到底由什麼觸發？(答案只有三種來源)
- 為什麼「props 變了所以重新渲染」是不精確的說法？
- `key` 在協調過程中扮演什麼角色？為什麼用陣列索引當 key 會出 bug？
- `React.memo`、`useMemo`、`useCallback` 各自的定位，以及為什麼「全部都包起來」是錯的？

「這個元件為什麼一直重新渲染」大概是前端工作裡最常見的效能疑問，而多數人是靠著口耳相傳的規則在猜。這一章把規則追到源頭：觸發渲染的來源其實非常少，把它們列乾淨，剩下的行為都能推導出來。

3.1 觸發重新渲染的三個來源

除了初次掛載，一個元件重新渲染，原因**只可能**是下列三者之一：

1. **自己的狀態更新了**：呼叫 `setState`（或 `dispatch`），且新值與舊值經 `Object.is` 比較不相等。
2. **父元件重新渲染了**：父元件重新執行時，會重新建立子元件的元素，預設情況下子元件就會跟著重新渲染——**跟 props 有沒有變無關**。
3. **訂閱的 context 更新了**：元件用 `useContext` 讀取的 Provider，其 `value` 換了新引用。

常見誤解：「props 改變會觸發重新渲染」

這句話因果顛倒了。props 是父元件渲染時傳進來的引數——**props 會變，正是因為父元件重新渲染了**；而只要父元件重新渲染，子元件預設就會重新渲染，即使 props 一個字都沒變。React 預設**根本不比較 props**。「props 沒變就跳過渲染」是 `React.memo` 提供的**選配行為**，不是預設值。把這個因果關係弄對，後面 memo 系列工具的定位才會清楚。

所以預設行為是「渲染會沿著樹往下傳染」：根元件一動，整棵子樹跟著重算。聽起來很浪費？回想第 1 章：渲染只是重新執行函式、產出新的描述物件，真正貴的 DOM 更新已經被 commit 階段最小化了。對絕大多數應用程式，這個「浪費」根本量測不出來——這是 React 刻意的取捨：**預設選擇簡單正確（全部重算），把優化留給真正需要的地方**。

3.2　一次渲染是一張快照

進入優化工具之前，必須先建立一個常被跳過、卻解釋了無數怪異行為的觀念：**每次渲染擁有自己固定不變的 props、狀態與函式**。

元件函式每執行一次，函式裡的所有東西——變數、事件處理器、傳給 hook 的閉包——都是「那一次執行」的產物，捕捉的是那一刻的狀態值。看這個經典例子：

```
function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
    // 點一下之後 count 是多少？
  }

  return <button onClick={handleClick}>{count}</button>;
}
```

答案是 1，不是 3。在這次渲染裡 `count` 就是常數 `0`，三行都是 `setCount(0 + 1)`——你排進佇列的是三次「把狀態設成 1」。狀態不是「即時變數」，是**渲染之間的快照**；`setCount` 也不會就地改變 `count`，它只是「預約下一次渲染」。

如果意圖是「在前一個結果上繼續算」，要用**函數式更新**，把「如何從舊值算新值」交給 React：

```
setCount((c) => c + 1);   // 收到的 c 是佇列中最新的值
setCount((c) => c + 1);
setCount((c) => c + 1);   // 結果是 3
```

另一個推論：連續呼叫多次 `setState` 並不會渲染多次。React 會**批次處理**（batching）——把同一個事件（React 18 之後包含 `setTimeout`、promise 回呼等所有地方）裡的狀態更新收集起來，只安排一次重新渲染。所以「盡量把多個 set 合併成一個以減少渲染」這種手工優化，多數時候是不必要的。

還有一個守門機制：`setState` 的新值若與目前值 `Object.is` 相等，React 會直接跳過這次更新，不重新渲染。這再次呼應第 1 章——mutate 之後 `set` 回同一個物件，就是在這裡被擋掉的。

3.3　協調與 key：React 怎麼認出「同一個」元件

渲染產出新的元素樹之後，React 要決定哪些舊節點可以沿用。規則很簡單：

- **同一個位置、同一種型別**（同一個標籤名或同一個元件函式）→視為同一個實例：保留它的

狀態與 DOM，只更新變動的屬性。

- **型別不同**→舊子樹整棵卸載（狀態全部銷毀），新子樹重新掛載。

「位置」這個詞在列表裡會出問題：列表重新排序或插入刪除時，「第 2 個位置」前後指的可能不是同一筆資料。這就是 `key` 的工作——**key 讓 React 用「身分」而不是「位置」來配對新舊節點**。

用陣列索引當 key，等於把「位置」再包裝一次，配對就會出錯：

```
// 反模式：index 當 key，且列表項有自己的狀態（輸入框）
{todos.map((todo, i) => (
  <TodoRow key={i} todo={todo} />   // TodoRow 內含 <input>
))}
```

假設畫面上有三列，你在第一列的輸入框打了字（這是 `TodoRow` 的內部狀態），然後刪除第一筆資料。新列表裡原本的第二筆變成 `key=0`——React 一比對：「`key=0` 的 `TodoRow` 還在」，於是**保留了第一列的元件實例（連同你打的字）**，只把 props 換成第二筆資料。結果：輸入的文字「黏」在錯誤的資料列上。改用穩定的識別（`key={todo.id}`），React 才能正確地銷毀第一列、保留其餘。

key 的選用準則

- 用資料的穩定識別碼（資料庫 id、uuid）。
- 陣列索引只在「列表永遠不重排、不插刪、項目無狀態」時勉強可用——但需求會變，預設就用 id 最安全。
- 不要用 `Math.random()` 當 key：每次渲染都是新 key，等於強迫整列卸載重建。
- 反過來利用：**刻意換 key 可以強制重置元件狀態**，例如 `<UserForm key={userId} />`，切換使用者時讓表單整個重來，比在 effect 裡手動清空欄位乾淨得多。

「型別不同就整棵重建」還解釋了一個真實世界的 bug 模式：

```
// Bug：在元件內定義元件
function Parent() {
  const [text, setText] = useState('');
  // 每次 Parent 渲染，Child 都是「新建立的函式」
  function Child() {
    return <input />;   // 使用者打的字會一直消失！
  }
  return (
    <>
      <input value={text} onChange={(e) => setText(e.target.value)} />
      <Child />
    </>
  );
}
```

每次 `Parent` 渲染都會建立一個**新的** `Child` 函式。對協調演算法而言，新舊 `Child` 是兩個不同的元件型別，於是每次渲染都把 `<Child />` 整棵卸載重建，內部狀態（焦點、輸入值）全部蒸發。規則：**元件永遠定義在模組最上層**，需要參數就用 props 傳。

3.4 React.memo：把「傳染」擋在邊界上

回到觸發來源第 2 條：父元件渲染，子元件預設跟著渲染。當某個子樹確實很重（大表格、複雜圖表），而且它的 props 常常沒變，就可以用 `React.memo` 在它門口設一道檢查哨：

```
const UserTable = React.memo(function UserTable({ users, onSelect }: Props) {
  // 只有 users 或 onSelect（淺比較）變了才重新渲染
  return /* 幾百列的表格 */;
});
```

`memo` 的規則：父元件渲染時，對新舊 props **逐一做 `Object.is` 淺比較**，全部相等就跳過這次渲染，直接沿用上次的結果。

注意「`Object.is` 淺比較」這幾個字——它就是 `memo` 在實務上「失效」的全部原因：

```
function Dashboard() {
  const [tab, setTab] = useState('overview');
  return (
    <UserTable
      users={allUsers.filter((u) => u.active)}  // 每次渲染都是新陣列！
      onSelect={(id) => navigate(`/users/${id}`)} // 每次渲染都是新函式！
    />
  );
}
```

`Dashboard` 每次因為 `tab` 改變而渲染時，`filter` 產生新陣列、箭頭函式是新函式——兩個 props 的引用都變了，`memo` 的比較永遠不會通過，檢查哨形同虛設。這就輪到另外兩個工具出場了。

3.5 useMemo 與 useCallback：讓引用跨渲染存活

這兩個 hook 常被籠統地理解成「效能優化」，但它們真正做的事非常具體：**在依賴不變的前提下，讓「同一個值 / 函式的引用」跨越多次渲染**。

```
function Dashboard() {
  const [tab, setTab] = useState('overview');

  // 依賴 allUsers 沒變 → 回傳「上一次的那個陣列」（同一個引用）
  const activeUsers = useMemo(
    () => allUsers.filter((u) => u.active),
```

```
    [allUsers]
  );

  // 依賴為空 → 永遠是同一個函式引用
  const handleSelect = useCallback((id: string) => {
    navigate(`/users/${id}`);
  }, []);

  return <UserTable users={activeUsers} onSelect={handleSelect} />;
}
```

現在 `tab` 改變時，`UserTable` 收到的兩個 props 引用都與上次相同，`memo` 的淺比較通過，大表格不再陪跑。順帶記住一個等式，可以少背一個 API：`useCallback(fn, deps)` 完全等價於 `useMemo(() => fn, deps)`——前者快取函式本身，後者快取函式的回傳值。

它們的第二個用途一樣是「穩定引用」，只是消費者換成 hook 的依賴陣列：如果某個物件 / 函式要放進 `useEffect` 的依賴，沒有穩定引用的話，effect 每次渲染都會重跑（第 4 章會再遇到這件事）。

核心觀念：三件套的分工

`React.memo` 是**檢查哨**：props 沒變就不渲染。`useMemo` / `useCallback` 是**保引用**：依賴沒變就給同一個引用。後兩者常常是為了讓前者（或依賴陣列）的 `Object.is` 比較能夠通過而存在。單獨使用 `useCallback` 而下游沒有任何 memo 元件或依賴陣列在消費它的穩定性，通常是白付成本。

常見誤解：「全部包上 memo / useMemo / useCallback 最保險」

每個 `useMemo` / `useCallback` 都有成本：快取佔記憶體、每次渲染要做依賴比較、程式碼變長變難讀，而且只要**一個** props 破功（例如漏掉一個 inline 物件），整組 memo 就無效。正確的順序是：先確認有量測得到的問題（React DevTools 的 Profiler 可以錄下每次互動觸發了哪些元件渲染、各花多少時間），找到真正的熱點，再沿著「memo 檢查哨＋穩定該元件的所有 props」佈防。順帶一提，很多「重渲染災難」的更好解法是**搬狀態**：把高頻變動的狀態往下移（colocation，第 2 章），讓它天生只影響一小塊子樹，根本不需要 memo。

3.6　Context 與重新渲染

第 2 章提過 context 的廣播性質，現在可以講精確了：Provider 重新渲染時，若 `value` 的引用與上次不同，所有 `useContext` 它的元件都會重新渲染。而最常見的失誤是：

```
function AuthProvider({ children }: { children: React.ReactNode }) {
  const [user, setUser] = useState<User | null>(null);
  // 反模式：value 是每次渲染新建的物件
  // AuthProvider 因任何原因重新渲染，全部 consumer 都跟著渲染
  return (
```

```
    <AuthContext.Provider value={{ user, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

// 修正：讓 value 的引用只在內容真正改變時更新
const value = useMemo(() => ({ user, logout }), [user, logout]);
```

若 context 同時裝了「資料」和「更新函式」，另一個常見手法是**拆成兩個 context**（`StateContext` 與 `DispatchContext`）：只呼叫動作、不讀資料的元件訂閱後者，資料更新時就不會被波及。

本章重點回顧

- 重新渲染只有三個來源：自己 setState（且值不同）、父元件渲染、訂閱的 context 更新。props 改變是父元件渲染的「結果」，不是獨立的觸發源。
- 每次渲染是一張快照：狀態在該次渲染內是常數；`setState` 是預約下次渲染；需要接續計算用函數式更新；更新會自動批次。
- 協調用「位置＋型別」配對節點，列表用 key 改成「身分」配對；index key 在會變動的列表是 bug 源；換 key 可刻意重置狀態；元件不可定義在元件內。
- `memo` 擋渲染、`useMemo`/`useCallback` 穩引用，三者配套才有效；先量測、先考慮搬狀態，再談 memo。
- context 的 `value` 要用 `useMemo` 穩定引用，必要時把 state 與 dispatch 拆開。

自我檢測

1. **一個沒被 memo 的元件，props 完全沒變，會不會重新渲染？**
   參考答案：會，只要它的父元件重新渲染。React 預設不比較 props；「props 沒變就跳過」是 `React.memo` 提供的選配行為。

2. **`setCount(count + 1)` 寫三次只加一，為什麼？怎麼改？**
   參考答案：同一次渲染內 `count` 是固定快照，三次都是「設成同一個值」。改用函數式更新 `setCount((c) => c + 1)`，讓每次更新基於佇列中的最新值。

3. **為什麼幫 `UserTable` 包了 `React.memo` 之後，它還是每次都重新渲染？**
   參考答案：幾乎可以肯定有 props 的引用每次都在變：inline 物件 / 陣列 / 函式，或 children。用 `useMemo`/`useCallback` 穩定那些 props，或檢查是否傳了每次都新建的 JSX。memo 的比較是 `Object.is` 淺比較，一個引用破功就整體失效。

4. **什麼情況下你會「故意」改變一個元件的 key？**
   參考答案：想強制重置元件的內部狀態時。例如 `<UserForm key={userId}>`：切換使用者時讓表單卸載重建，全部欄位自然歸零，比用 effect 逐欄清空更不容易漏。

5. **同事把專案裡每個函式都包上 `useCallback`，說是「效能好習慣」。你的看法？**

參考答案： useCallback 本身有記憶體與比較成本，且只有在下游有 memo 元件或依賴陣列消費「引用穩定性」時才有意義。無差別包裹是負優化且傷可讀性。應先用 Profiler 找熱點，或考慮搬移狀態縮小渲染範圍。

第 4 章

Hooks：為什麼存在、為什麼長這樣

本章目標——讀完後你應該能回答

- Hooks 是為了解決什麼問題而誕生的？（不是「比 class 潮」）
- 為什麼 hooks 只能在元件頂層呼叫、不能放進 if 裡？
- `useEffect` 的正確心智模型是什麼？依賴陣列到底是什麼意思？
- 什麼時候該把邏輯抽成自訂 hook？`useRef` 和 `useReducer` 的定位？

你每天都在用 hooks，但「為什麼 React 要發明 hooks」這個問題的答案，決定了你會把它們用在對的地方，還是只是照抄範例。這一章先回到 hooks 誕生前的世界，看清楚它要解決的問題，再逐一拆解每個 hook 的設計理由。

4.1 Hooks 之前的世界：class 元件的三個痛點

2019 年之前，有狀態的元件只能寫成 class。它有三個越用越痛的問題：

痛點一：有狀態的邏輯難以復用。「追蹤視窗大小」「訂閱聊天室」「表單欄位管理」這類邏輯都帶著自己的狀態與生命週期，偏偏 class 元件的狀態與生命週期都焊死在元件本體上，抽不出來。社群發明了兩種 workaround，各有致命傷：高階元件（HOC，如 `withRouter(withTheme(withAuth(Page)))`）讓元件樹多出一堆包裝層，props 從哪個包裝層來的無從查起，還會撞名；render props（`<Mouse render={(pos) => <Cat pos={pos}/>}/>`）一旦要同時復用三種邏輯，JSX 就變成三層巢狀的金字塔。兩者的共同缺陷：**為了復用「邏輯」，卻被迫改變「元件樹的結構」。**

痛點二：生命週期方法把「一件事」切成三段，又把「三件事」混在一段。

```
class ChatRoom extends React.Component<Props> {
  componentDidMount() {
    this.subscribe(this.props.roomId);       // 訂閱：第 1 段
    trackPageView();                         // 順便混進了無關的埋點
  }
  componentDidUpdate(prev: Props) {
    if (prev.roomId !== this.props.roomId) { // 訂閱：第 2 段
      this.unsubscribe(prev.roomId);         //   （這段最常被忘記寫）
      this.subscribe(this.props.roomId);
    }
  }
  componentWillUnmount() {
```

```
    this.unsubscribe(this.props.roomId);     // 訂閱：第 3 段
  }
}
```

「維持聊天室訂閱」是一個完整的關注點，卻被按**時間點**切散在三個方法裡；而 `componentDidMount` 裡又常常同時塞著訂閱、埋點、抓資料等**互不相干**的事。想讀懂「訂閱是怎麼運作的」，你得在三個方法之間跳來跳去，還要自行過濾雜訊。

痛點三：`this` 與樣板碼。忘記 `bind`、建構子裡一排 `this.handleClick = this.handleClick.bind(this)`、`this.setState` 的淺合併規則……這些都是與業務無關的認知負擔。

核心觀念：Hooks 的核心主張

Hooks 讓「有狀態的邏輯」變成**可以用函式呼叫來組合的單位**：

- 程式碼按**關注點**組織（一個 `useEffect` 管一件事、一個自訂 hook 封裝一個能力），而不是按生命週期的時間點切割。
- 復用邏輯＝抽一個函式（自訂 hook），**完全不改變元件樹**——對比 HOC 與 render props 的最大差異。

上面的 `ChatRoom`，用 hooks 寫就是一個 `useEffect`（訂閱＋清理＋換房間，三段合一）加一個獨立的 `useEffect`（埋點），每個關注點自成一格。

4.2 為什麼 hooks 有那些「奇怪的規則」

「只能在最頂層呼叫，不能放進條件、迴圈」——這條規則不是風格偏好，而是 hooks 儲存機制的直接結果，值得花三分鐘徹底弄懂。

`useState` 是個普通函式，它怎麼知道「這次呼叫」對應哪一份狀態？答案：React 在每個元件實例（fiber 節點）上維護一個 **hook 狀態串列**，渲染時**按呼叫順序**一格一格對應——第 1 次 hook 呼叫拿第 1 格、第 2 次拿第 2 格，不看變數名，只看順序。

儲存格	第一次渲染	第二次渲染（若 if 成立時跳過一個 hook）
0	`useState('Amy')` →存名字	`useState('Amy')` →讀到名字（✓ 正確）
1	`useState(0)` →存計數	（被 if 跳過）
2	`useEffect(訂閱)`	`useEffect(訂閱)` →讀到「計數」的格子，全面錯位

只要某次渲染少呼叫（或多呼叫）了一個 hook，後面所有 hook 都會對到錯的格子——狀態張冠李戴，直接崩潰。所以規則的真正內容是：**每次渲染的 hook 呼叫順序必須完全一**

致，「不放進 if / 迴圈 / early return 之後」只是保證這件事的手段。需要條件行為時，條件寫在 hook **裡面**（例如 effect 內先 `if (!enabled) return`），而不是有條件地呼叫 hook。`eslint-plugin-react-hooks` 就是在自動檢查這件事，公司專案幾乎一定裝了——它報的錯不要用註解壓掉，那通常代表你正要製造一個錯位或過期閉包的 bug。

為什麼不設計成「給每個狀態取名字」？

理論上可以要求 `useState('count', 0)` 用字串當 key，就不受順序限制。但名字會在複製貼上時撞名、在兩個自訂 hook 內部互相衝突——「順序即身分」反而讓每個自訂 hook 天然擁有**完全隔離的狀態空間**，這是「自訂 hook 可以任意組合而互不干擾」的基礎。規則是這個設計換來的代價，而且能被 lint 工具完全自動檢查。

4.3 useState 進階：狀態存在哪裡、閉包會過期

第 3 章建立了「渲染即快照」；配上剛才的儲存機制，可以把模型補完整：**狀態存在 React 的 fiber 上，元件函式只是每次渲染被重新執行的「配方」**。函式裡的變數、閉包每次渲染都是新的，但 `useState` 從 fiber 讀回來的值是延續的。

這個模型直接解釋了最常見的 hooks bug——**過期閉包**（stale closure）：

```
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1);   // 這個閉包捕捉的是「掛載那次渲染」的 count = 0
    }, 1000);
    return () => clearInterval(id);
  }, []);   // 空依賴：effect 只在掛載時建立一次

  return <p>{count}</p>;    // 永遠停在 1
}
```

interval 的回呼是掛載那次渲染建立的閉包，它捕捉的 `count` 永遠是 `0`，所以每秒都在執行 `setCount(0 + 1)`。畫面卡在 1，不多不少。三種修法，各對應一種思路：

```
// 修法一（最好）：函數式更新——根本不需要讀 count
setCount((c) => c + 1);

// 修法二：誠實申報依賴——count 變了就重建 interval（每秒拆裝一次，可行但較粗）
useEffect(() => { ... }, [count]);

// 修法三：用 ref 保存「最新值」（進階，見 4.5 節）
```

實務建議

看到 `setState` 時第一個反射：新值需要用到舊值嗎？需要就寫函數式更新 `set((prev) => ...)`。這個習慣同時消滅「快照疊加」（第 3 章）和「過期閉包」兩類 bug，而且讓函式可以安全地從依賴陣列中省略對狀態值的依賴。

4.4 useEffect：是「同步」，不是「生命週期」

`useEffect` 是被誤解最深的 hook，因為多數人帶著 class 的舊地圖來認路：「空陣列＝`componentDidMount`，有依賴＝`componentDidUpdate`」。這個對照表能讓你寫出會動的程式碼，但它是錯的地圖，遲早把你帶進溝裡。

正確的模型：**useEffect 描述「如何讓某個外部系統與 React 的狀態保持同步」**。外部系統指 React 管不到的東西：網路連線、訂閱、計時器、DOM API、第三方函式庫。

- **effect 本體**：如何開始同步（建立連線、掛上監聽器）。
- **cleanup**：如何停止同步（斷線、移除監聽器）。
- **依賴陣列**：這段同步邏輯**讀取了哪些會變的值**——不是「什麼時候執行」的開關，而是「同步的內容取決於誰」的誠實申報。依賴變了 = 舊的同步已經過時 = React 先跑 cleanup 再跑新的 effect。

用這個模型看聊天室，三段生命週期合成一段自然的敘述：

```
useEffect(() => {
  const conn = connect(roomId);    // 開始同步：連上這個房間
  return () => conn.close();       // 停止同步：離開這個房間
}, [roomId]);                      // 同步的對象取決於 roomId
```

`roomId` 從 A 變成 B 時，React 自動執行「`close(A)` → `connect(B)`」。class 版本裡最容易忘記寫的 `componentDidUpdate` 分支，在這個模型下**根本不存在**——你沒有機會忘記。這就是「按關注點組織」的實際威力。

常見誤解：「依賴陣列是用來控制執行時機的」

把依賴陣列當「時機開關」的人，會為了「我只想跑一次」而故意留空陣列、或為了「我不想它重跑」而從陣列裡偷偷拿掉某個值。後者製造的就是上一節的過期閉包：effect 讀著舊渲染的值繼續工作。規矩只有一條：**effect 裡讀到的所有 reactive 值（props、state、渲染中算出的變數）都必須列進依賴**。如果「誠實申報之後 effect 跑太頻繁」，問題永遠出在別處——值的引用不穩定（用 `useMemo`/`useCallback` 穩定它，第 3 章）、不需要 reactive 的值混了進來（搬出去或用 ref）、或這段邏輯根本不該是 effect（見下方清單）。

抓資料的競態問題也是同步模型的直接應用。依賴變了會先跑 cleanup——這正是取消過期請求的正確位置：

```
useEffect(() => {
  let ignore = false;              // 本次同步的「已過時」旗標
  fetchUser(userId).then((user) => {
    if (!ignore) setUser(user);    // 慢回來的舊請求會被這裡擋掉
  });
  return () => { ignore = true; };
}, [userId]);
```

沒有這層防護，使用者快速從使用者 A 切到 B 時，若 A 的回應比 B 晚到，畫面就會顯示 B 的網址配 A 的資料。（用 `AbortController` 是更完整的版本：cleanup 時直接 `abort()` 取消請求。）這也是 React 18 StrictMode 在開發模式故意「掛載→卸載→再掛載」跑兩次 effect 的原因：**逼你把 cleanup 寫對**——寫對的 effect 跑兩次結果不變，寫錯的當場現形。所以「開發環境 API 都打兩次」不是 bug，是體檢。

最後是同樣重要的反面清單。**以下情況不該用 effect**：

- **從現有狀態算另一個值**——那是衍生資料，渲染時直接算（第 2 章）。
- **回應使用者操作**——「按下按鈕要送出訂單」的邏輯屬於事件處理器；塞進 effect（「監聽某狀態變 true 就送出」）會讓因果鏈斷裂：讀的人無法從事件追到後果，而且狀態被其他路徑改到時會誤觸發。
- **「狀態 A 變了就去 set 狀態 B」的連鎖**——通常是狀態設計錯誤的訊號，回到第 2 章的最小狀態原則重新想。

判斷準則一句話：**effect 是給「與外部系統同步」用的逃生口；不涉及外部系統的邏輯，幾乎都有更好的位置**。

4.5　useRef：不參與渲染的可變盒子

`useRef` 回傳一個在**所有渲染之間共享**的盒子 `{ current: ... }`：改它不觸發渲染，讀它永遠拿到最新值——正好是 state 的鏡像互補。

	useState	useRef
改變時重新渲染？	會	不會
渲染中讀到的值	該次渲染的快照	永遠是最新值（同一個盒子）
適合存放	會反映在畫面上的資料	與畫面無關的機制性資料

三種典型用途：

```
// 1. 操作 DOM：宣告式模型的正式逃生口（聚焦、捲動、量尺寸）
const inputRef = useRef<HTMLInputElement>(null);
<input ref={inputRef} />;          // 之後可呼叫 inputRef.current?.focus()
```

```
// 2. 保存機制性資料：計時器 id、WebSocket 實例、上一次的值
const timerRef = useRef<number | null>(null);

// 3. 讓長壽命閉包讀到最新值（解過期閉包的第三條路）
const latestQuery = useRef(query);
useEffect(() => { latestQuery.current = query; });  // 每次渲染後更新
```

判斷「該 state 還是該 ref」只需一個問題：**這個值改變時，畫面需要跟著變嗎？**需要→ state；不需要（純機制）→ ref。反過來的錯誤也常見：把該進 ref 的計時器 id 放進 state，結果每次設定計時器都觸發一次毫無意義的重新渲染。

4.6　自訂 Hook：邏輯復用的正確單位

把前面所有觀念收攏起來的，就是自訂 hook——**一個以 `use` 開頭、內部可以呼叫其他 hooks 的普通函式**。它是 hooks 系統回應「痛點一」的正式答案：復用有狀態的邏輯，不動元件樹。

看一個工作中幾乎必然遇到的例子——搜尋防抖：

```
// useDebounce.ts：把「值的防抖」封裝成可復用的能力
export function useDebounce<T>(value: T, delayMs = 300): T {
  const [debounced, setDebounced] = useState(value);

  useEffect(() => {
    const id = setTimeout(() => setDebounced(value), delayMs);
    return () => clearTimeout(id);   // 值又變了：取消上一輪倒數
  }, [value, delayMs]);

  return debounced;
}

// 使用端：讀起來像宣告一個事實，細節完全隱形
function SearchPage() {
  const [keyword, setKeyword] = useState('');
  const debouncedKeyword = useDebounce(keyword, 300);
  const results = useSearchResults(debouncedKeyword);
  // ...
}
```

幾個值得內化的觀察：

- **封裝的是「意圖」**：使用端只看到「防抖後的關鍵字」這個概念，`setTimeout` 與 cleanup 的舞步被藏起來。好的自訂 hook 讓元件讀起來像需求描述。
- **每次呼叫都是獨立實例**：兩個元件各自呼叫 `useDebounce`，狀態完全隔離（回想「順序即身分」的儲存機制）——這是 mixin 與 HOC 做不到的乾淨。

- **泛型 `<T>` 讓它對任何型別可用**且不丟失型別資訊——傳入 `string` 就回傳 `string`。這是第 6 章泛型的預告。
- **名字必須 `use` 開頭**不是裝飾：它讓 lint 工具知道「這個函式內部有 hooks」，從而檢查呼叫順序規則。

什麼時候抽自訂 hook？

兩個訊號：(1) 兩個元件出現重複的「state + effect」組合；(2) 單一元件裡，某群 state 與 effect 明顯屬於同一個關注點、與其他邏輯糾纏在一起（就算不復用，抽出來也讓兩邊都更好讀、更好測）。公司程式碼裡的 `useAuth`、`usePagination`、`useTableSort` 都是這樣長出來的——先有具體重複，再有抽象，順序不要反過來。

4.7 useReducer：把「狀態怎麼變」集中成一張規則表

多個 `useState` 總是一起變、更新邏輯散落在各個事件處理器、新同事不知道「還有哪些地方會改這包狀態」——這是 `useReducer` 出場的訊號：

```
type CartAction =
  | { type: 'add'; item: Item }
  | { type: 'remove'; itemId: string }
  | { type: 'clear' };

function cartReducer(state: CartState, action: CartAction): CartState {
  switch (action.type) {
    case 'add':    return { ...state, items: [...state.items, action.item] };
    case 'remove': return { ...state, items: state.items.filter((i) => i.id !==
    action.itemId) };
    case 'clear':  return { ...state, items: [] };
  }
}

const [cart, dispatch] = useReducer(cartReducer, initialCart);
// 事件處理器只剩「宣告發生了什麼」：dispatch({ type: 'add', item })
```

它買到的東西：**所有合法的狀態轉移集中在一個純函式裡**（一眼讀完、可以單獨測試、新增轉移時 TypeScript 的窮盡檢查會提醒你漏了哪個 case，見第 6 章）；事件處理器退化成「報告事件」；而且 `dispatch` 的引用天生穩定，往下傳不需要 `useCallback`。你會發現這正是 Redux 的核心思想縮小到單一元件——讀懂這個，公司若在用 Redux/RTK，模型是同一套。

本章重點回顧

- Hooks 解決 class 的核心痛：邏輯按關注點組織、復用邏輯不改變元件樹。
- 「只能頂層呼叫」源自「順序即身分」的儲存機制；條件寫在 hook 內部，不是有條件地呼叫 hook。

- 狀態存在 fiber 上，函式每次渲染重跑；閉包捕捉快照→過期閉包；函數式更新是第一線解法。
- `useEffect` = 與外部系統同步：本體開始、cleanup 停止、依賴陣列是「同步讀了誰」的誠實申報；衍生資料與事件邏輯不屬於 effect。
- `useRef` 是不觸發渲染的可變盒子：DOM、機制性資料、最新值。
- 自訂 hook 封裝「state+effect 的關注點」；`useReducer` 把狀態轉移集中成可測試的規則表。

自我檢測

1. **為什麼 hooks 不能寫在 if 裡？用儲存機制解釋。**
   參考答案： React 按「呼叫順序」把每次 hook 呼叫對應到 fiber 上的儲存格。if 讓某次渲染的呼叫次序改變，後續所有 hook 都會對到錯的格子，狀態全面錯位。所以規則的本質是「每次渲染的 hook 序列必須一致」。

2. **`useEffect(() => {...}, [])` 裡的程式碼讀了 `props.userId`，lint 要你把它加進依賴，你卻只想「跑一次」。怎麼辦？**
   參考答案：先回到同步模型：effect 讀了 userId，代表同步內容取決於它，userId 變了本來就該重新同步——加進依賴通常才是對的。若這段邏輯真的「只該發生一次、與後續變化無關」，它可能根本不是同步（例如應該搬到事件處理器），或應改用 ref 讀非 reactive 的值。硬留空陣列就是自製過期閉包。

3. **什麼樣的值該放 `useRef` 而不是 `useState`？**
   參考答案：改變時不需要畫面更新的機制性資料：計時器 id、外部連線實例、「最新值」的鏡像、上一次的值。判斷句：這個值變了畫面要跟著變嗎？不用就 ref。

4. **開發環境每個 API 都打兩次，QA 問是不是 bug。你怎麼解釋？**
   參考答案：是 React 18 StrictMode 在開發模式故意將元件「掛載→卸載→重掛載」，驗證 effect 的 cleanup 是否寫對。cleanup 正確（取消請求或忽略過期回應）時，行為與一次相同；生產環境不會發生。該修的不是雙跑，而是沒寫 cleanup 的 effect。

5. **同事在 effect 裡監聽 `isSubmitClicked` 狀態變 true 就呼叫送出 API，然後再 set 回 false。你會怎麼重構？**
   參考答案：這是把事件邏輯寫成 effect 的反模式：因果被拆散、狀態繞了一圈、還可能被其他路徑誤觸發。直接把送出邏輯放回按鈕的事件處理器，刪掉 isSubmitClicked 這個假狀態。

第 5 章

元件設計：拆分、組合與 API 思維

本章目標——讀完後你應該能回答

- 「好的元件」有哪些可判斷的特徵？何時該拆、何時不該拆？
- 為什麼資深工程師偏好「組合」勝過一直加 props？
- 受控與非受控元件的本質差異是什麼？公司的表單元件為什麼都設計成受控？
- 邏輯與畫面怎麼分離？Compound Components 解決什麼問題？

前四章講 React 的運作規則；這一章開始講**設計**——同樣的需求有很多種寫法都「能動」，差別在半年後好不好改。元件設計的核心視角轉換只有一句話：**元件是一個 API，props 是它的介面，使用它的同事（包括未來的你）是它的用戶。**

5.1 好元件的三個可檢驗特徵

「高內聚低耦合」這類口號難以操作，換成三個看得見的檢查點：

1. **單一職責**：能用一句不含「和」「以及」的話說出它做什麼。「顯示使用者卡片」通過；「顯示使用者卡片以及管理追蹤狀態以及抓取推薦名單」該拆了。
2. **介面誠實**：看 props 型別就能正確使用，不需要打開實作。隱藏的前提（「要先包在某個 Provider 裡」「兩個 props 只能擇一傳」）要嘛消除、要嘛用型別表達出來（第 7 章會做到「擇一傳」的型別）。
3. **行為可預測**：同樣的 props 與狀態渲染出同樣的畫面，沒有偷讀全域變數、沒有渲染中的副作用——這是第 1 章純函數模型的延伸。

至於「何時拆元件」，行數只是最粗糙的訊號。更可靠的三個拆分理由：

- **職責出現了邊界**：一段 JSX 有自己的名字（「篩選列」「結果卡片」），抽出來後父元件讀起來像目錄。
- **狀態需要隔離**：第 3 章講過，狀態放低影響範圍小。高頻變動的輸入框抽成子元件，它的每一次按鍵就不會拖著整頁重算。
- **確實要復用**。注意順序——復用是拆分的理由之一，不是唯一理由；反過來，「還沒有第二個使用者」也不構成不拆的理由。

拆過頭同樣有成本：每多一層就多一次 props 轉手、多一個要命名的東西、多一跳閱讀距離。**拆分的收益是「每一塊變簡單」，代價是「塊與塊的接線變多」**，兩邊都要算。

5.2 Props 設計：從使用端往回設計

寫元件最容易犯的順序錯誤：先寫實作，props 是實作「剛好需要什麼就加什麼」的沉積物。方向應該相反——**先寫出你希望使用端長什麼樣子，再回頭實作它**（跟先寫測試同一種思維）。

一個很快就會失控的真實模式，布林 props 的堆積：

```
// 每來一個需求就長一顆布林，排列組合無人能懂：
<Dialog isOpen showCloseButton hideFooter isCompact isDanger
        centerTitle noPadding ... />
```

布林 props 的問題不只是醜：n 顆布林有 2^n 種組合，其中大半是沒人測過、甚至語意矛盾的（`isCompact` 加 `noPadding` 會怎樣？誰知道）。兩個標準修法：

修法一：互斥的布林收斂成一個變體（variant）。當幾顆布林其實在表達「同一個維度上的不同選項」，用字面量聯集（literal union，第 6 章詳述）取代：

```
// size 只會是三者之一，不存在「同時 compact 又 large」的非法組合
interface DialogProps {
  size?: 'compact' | 'default' | 'large';
  tone?: 'neutral' | 'danger';
}
```

修法二：內容相關的設定改用組合（composition）。這是下一節的主題。

props 介面的幾條經驗法則

- 以**領域語言**命名，不以實作細節命名：`tone="danger"` 優於 `redBorder`。
- 回呼參數傳「領域資料」不傳整包內部狀態：`onSelect(userId)` 優於 `onSelect(event, row, index, ...)`——洩漏越少，日後越好改。
- 相關的 props 一起變動時，考慮合成一個物件；獨立變動時保持分開。
- 預設值讓「最常見的用法」寫起來最短：`<Dialog>` 不帶任何 props 就該是最普通的對話框。

5.3 組合優於設定：children 與插槽

回到 `Dialog` 的例子。需求持續演化：頁尾有時一顆按鈕、有時兩顆、有時要放連結、有時整個不要……用「設定」的思路，你會不斷加 props（`footerButtons`、`footerAlign`、`hideFooter`…），元件變成一台越來越難操作的萬能機器。

「組合」的思路是：**元件只負責結構與外框，把「洞」開出來讓使用端自己填**。React 給這個模式的原生支援就是 `children`，以及「接受 JSX 的 props」（俗稱插槽，slots）：

```
interface DialogProps {
  title: string;
```

```
  footer?: React.ReactNode;    // 插槽：整塊頁尾由使用端決定
  children: React.ReactNode;   // 主內容
}

function Dialog({ title, footer, children }: DialogProps) {
  return (
    <div className="dialog">
      <header>{title}</header>
      <main>{children}</main>
      {footer !== undefined && <footer>{footer}</footer>}
    </div>
  );
}

// 使用端：頁尾要長怎樣是使用端的自由，Dialog 不需要知道
<Dialog title="刪除確認" footer={
  <>
    <Button tone="danger" onClick={confirm}>刪除</Button>
    <Button onClick={close}>取消</Button>
  </>
}>
  確定要刪除這 3 個項目嗎？
</Dialog>
```

這樣設計之後，**新需求不再需要修改 `Dialog`**——它對「頁尾可能長什麼樣」保持開放，卻對自己的結構保持封閉。公司 UI 元件庫裡 `ReactNode` 型別的 props 這麼多，就是這個原因。

組合還有一個常被忽略的妙用：**解 prop drilling**。第 2 章說過中間層轉傳 props 的痛，除了 context，另一個解法是 **在高處把元件組好，讓中間層只負責擺放**：

```
// 中間層 Layout 完全不知道（也不需要知道）user 的存在
function Page() {
  const user = useCurrentUser();
  return <Layout sidebar={<UserPanel user={user} />} />;
}
function Layout({ sidebar }: { sidebar: React.ReactNode }) {
  return <div className="layout"><aside>{sidebar}</aside><main>...</main></div>;
}
```

資料 `user` 從擁有者直接跳到使用者，中間層的介面乾乾淨淨。遇到 prop drilling 時，先想「能不能在上面組好再傳下來」，再考慮 context。

5.4　受控與非受控：狀態主導權的設計決策

這組概念從表單來，但它其實是一個所有互動元件都要回答的通用問題：**「這個元件的狀態，由誰擁有？」**

- **非受控**（uncontrolled）：元件自己管狀態，外界頂多給個初始值。原生 `<input defaultValue="..." />` 就是：值存在 DOM 裡，React 不過問。
- **受控**（controlled）：狀態由外部擁有，元件只負責「顯示外部給的值＋回報使用者的動作」。`<input value={x} onChange={...} />`：畫面上永遠顯示 `x`，使用者打字只會觸發 `onChange`，**外部不更新狀態，畫面就不會變**。

公司的表單元件幾乎清一色設計成受控，理由回到第 2 章的單一事實來源：值住在 React 狀態裡，於是「即時驗證」「欄位聯動」（選了國家，城市清單跟著變）「送出時收集所有值」「一鍵重設」全都是對狀態的普通操作；值若散落在各個 DOM 節點裡，每個需求都要用 ref 去 DOM 裡撈。代價是每個欄位都要接 `value`/`onChange` 的樣板碼——這正是 React Hook Form 這類函式庫要優化的點（它用非受控＋ ref 換效能，表單很大時每次按鍵不會整張表單重新渲染——你看，又是第 3 章的取捨）。

設計**自己的**可互動元件（開關、分頁籤、日期選擇器）時，同一個決策會找上你。通用建議：**對外提供受控介面（`value` + `onChange`）為主**，因為受控元件可以被任何外部邏輯指揮，組合性最強；行有餘力再補非受控模式（`defaultValue`）方便簡單場景。成熟元件庫的「`value` / `defaultValue` 雙介面」慣例就是這麼來的。

5.5　邏輯與畫面分離：hook 管邏輯，元件管長相

老 React 教材講「容器元件 / 展示元件」（container / presentational）分離；hooks 出現後，這個模式演化成更輕的形式：**把「這個功能怎麼運作」抽進自訂 hook，讓元件只剩「這個功能長什麼樣」**。

```
// useUserSearch.ts —— 邏輯層：狀態、防抖、抓資料、競態防護
export function useUserSearch() {
  const [keyword, setKeyword] = useState('');
  const debounced = useDebounce(keyword, 300);
  const { data: users, status } = useQuery(searchUsersQuery(debounced));
  return { keyword, setKeyword, users: users ?? [], status };
}

// UserSearchPage.tsx —— 畫面層：讀起來只剩結構
function UserSearchPage() {
  const { keyword, setKeyword, users, status } = useUserSearch();
  return (
    <>
      <SearchInput value={keyword} onChange={setKeyword} />
      {status === 'pending' ? <Spinner /> : <UserList users={users} />}
```

```
    </>
  );
}
```

收益具體可感：邏輯可以脫離畫面單獨測試（測 hook 不用管 CSS）；換畫面不動邏輯（桌面版、行動版共用同一個 hook）；以及**閱讀的分層**——追 bug 時，畫面錯找元件、行為錯找 hook，一半的程式碼可以直接跳過。你在公司程式碼看到的 `useXxxPage`、`useXxxForm` 命名，多半就是這個模式。

5.6 Compound Components：一組元件共享隱式狀態

當一個元件的「內部結構」本身需要高度自由——最典型的是分頁籤（Tabs）——連插槽都不夠用了。設定式 API 會長成這樣：

```
// 設定式：結構被鎖死在 items 的形狀裡，想在籤上加個徽章都沒地方寫
<Tabs items={[
  { key: 'a', label: '總覽', content: <Overview /> },
  { key: 'b', label: '成員', content: <Members /> },
]} />
```

Compound components（複合元件）改成**一組互相配合的元件**，外形交還給使用端，「目前選到哪個籤」這個共享狀態則由最外層元件用 context 隱式提供：

```
<Tabs defaultKey="a">
  <Tabs.List>
    <Tabs.Tab tabKey="a">總覽</Tabs.Tab>
    <Tabs.Tab tabKey="b">成員 <Badge count={3} /></Tabs.Tab>
  </Tabs.List>
  <Tabs.Panel tabKey="a"><Overview /></Tabs.Panel>
  <Tabs.Panel tabKey="b"><Members /></Tabs.Panel>
</Tabs>
```

```
// 實作骨架：Tabs 擁有狀態，透過 context 發給子元件
const TabsContext = createContext<{
  active: string;
  setActive: (key: string) => void;
} | null>(null);

function Tabs({ defaultKey, children }: TabsProps) {
  const [active, setActive] = useState(defaultKey);
  const value = useMemo(() => ({ active, setActive }), [active]);
  return <TabsContext.Provider value={value}>{children}</TabsContext.Provider>;
}
```

```
function Tab({ tabKey, children }: TabProps) {
  const ctx = useTabsContext();          // 內部 hook，含 null 檢查
  return (
    <button
      aria-selected={ctx.active === tabKey}
      onClick={() => ctx.setActive(tabKey)}
    >{children}</button>
  );
}
```

你可以在這個骨架裡認出前面所有章節的零件：狀態提升（`active` 住在 `Tabs`）、context 跨層傳遞、`useMemo` 穩定 provider value、用 hook 封裝 context 存取。Radix、Headless UI、Ant Design 等元件庫大量使用這個模式——讀懂它，那些元件庫的文件會突然變得眼熟。

5.7　讓不可能的狀態無法出現

元件設計的最後一塊拼圖與 TypeScript 銜接。看這個到處都是的寫法：

```
// 三顆獨立布林：2 x 2 x 2 = 8 種組合，合法的只有 4 種
const [isLoading, setIsLoading] = useState(false);
const [isError, setIsError] = useState(false);
const [data, setData] = useState<User[] | null>(null);
//  「isLoading 且 isError」是什麼意思？忘記重設某一顆時就會出現。
```

三顆布林彼此其實不獨立——它們是**同一台狀態機的碎片**。碎片化的代價：每條更新路徑都要記得維護所有碎片的一致性，漏一顆就進入非法狀態。修法是把狀態合併成**一個**值，讓非法組合在型別上根本寫不出來：

```
type FetchState =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success'; data: User[] }   // data 只在成功時存在
  | { status: 'error'; error: string };   // error 只在失敗時存在

const [state, setState] = useState<FetchState>({ status: 'idle' });
```

「載入中卻有錯誤」現在連寫都寫不出來；而且 `data` 綁在 `success` 分支上，「忘記檢查 loading 就用 data」會直接變成型別錯誤。這個技術叫**標記聯集**（discriminated union），是第 6 章的重頭戲——設計章在這裡先給你動機：**型別系統是元件設計的工具，好的型別讓錯誤的用法無法通過編譯**。

本章重點回顧

- 元件是 API：單一職責、介面誠實、行為可預測；拆分看職責邊界、狀態隔離與復用，不是行數。
- props 從使用端往回設計；互斥布林收斂成 variant 聯集。
- 組合優於設定：children / 插槽讓元件對變化開放、對修改封閉；在高處組好元件也能解 prop drilling。
- 受控 / 非受控是「狀態誰擁有」的決策；可重用元件以受控介面為主。
- 自訂 hook 裝邏輯、元件裝畫面；compound components 用 context 讓一組元件共享隱式狀態。
- 用單一狀態機取代一把布林，讓不可能的狀態無法表達——型別是設計工具。

自我檢測

1. **某個 `Card` 元件已經有 12 個 props，其中 5 個是控制頁尾的。你會怎麼重構？**
   參考答案：把頁尾整塊改成插槽：`footer?: ReactNode`，刪掉那 5 顆設定 props，頁尾的長相交還給使用端。元件對內容保持開放、對結構保持封閉，之後頁尾再怎麼變都不用改 Card。

2. **什麼時候該把一段 JSX 抽成子元件？說出三個與「行數」無關的判斷。**
   參考答案：(1) 這段 JSX 有清楚的領域名字與職責邊界；(2) 它包含高頻變動的狀態，抽出去可以隔離重新渲染範圍；(3) 有真實的復用需求。反之，抽出去只是搬家、接線變多，就先不拆。

3. **你要做一個公司內部的 `Switch` 開關元件，介面要設計成受控還是非受控？**
   參考答案：以受控為主：`checked` + `onChange`，讓外部邏輯（表單、權限、聯動）能完全指揮它；可加 `defaultChecked` 支援簡單場景。只做非受控會讓它無法參與任何跨元件邏輯。

4. **`isLoading`、`isError`、`data` 三個 state 有什麼隱患？怎麼改？**
   參考答案：三者是同一狀態機的碎片，存在 8 種組合但只有少數合法，每條更新路徑都可能漏改造成非法狀態。改成單一的標記聯集 `{status: 'idle' | 'loading' | 'success' | 'error', ...}`，讓非法組合無法表達、資料綁定在對應分支上。

第 6 章

TypeScript 型別系統：從「標註」到「建模」

本章目標——讀完後你應該能回答

- TypeScript 的型別檢查依據什麼原則？「結構型別」是什麼意思？
- `type` 和 `interface` 到底該用哪個？
- 泛型在參數化什麼？`<T>` 要怎麼讀？
- 標記聯集、`keyof`、utility types 這些工具各自解決什麼實際問題？

用了三個月 TypeScript 的人，多半把型別當成「寫完程式後補的標註」，遇到紅線就想辦法讓它消失（必要時 `as any`）。這一章要完成一個視角升級：**型別是建模工具——先用型別把「什麼是合法的資料與操作」描述出來，讓編譯器替你守住這個模型**。第 5 章結尾的「讓不可能的狀態無法表達」就是這個視角的實戰預告。

6.1 型別在做什麼：把值分成集合

一個型別就是**一群值的集合**：`string` 是所有字串、`'a' | 'b'` 是只有兩個成員的集合、`never` 是空集合。型別檢查回答的問題永遠是同一個：**「這裡出現的值，保證屬於這裡要求的集合嗎？」**賦值、傳參數、回傳，全部是這一個問題的變形。

用集合觀點，很多語法瞬間變得可推理：

- `A | B`（聯集）：兩個集合的併集——值可能是 A 也可能是 B，所以**只能安全使用兩者共有的成員**，用其他成員前要先收窄（6.4 節）。
- `A & B`（交集）：同時滿足兩者——物件要同時具備兩邊的欄位。
- `string` 比 `'save' | 'cancel'` **大**：小集合可以放進大集合的位置，反之不行。這就是為什麼函式參數宣告成字面量聯集後，亂傳字串會被擋下。

而 TypeScript 的價值主張，說穿了是把一類錯誤的發現時間往前搬：**執行期的 `undefined is not a function`，變成編譯期（其實是「打字當下」）的紅線**。附帶的紅利在日常工作裡甚至更有感——型別是**不會過期的文件**（註解會騙人，型別不會，因為它被強制執行）、是**重構的安全網**（改個欄位名，所有沒改到的使用點立刻亮紅）、也是 **IDE 自動補全的資料來源**。

型別會在編譯後蒸發（type erasure）

TypeScript 的型別只存在於編譯期，編譯成 JavaScript 後**全部消失**。這帶來一個工作上必須內化的推論：**型別不會在執行期保護你**。你宣告 API 回應是 `User[]`，只是「向編譯器承諾」，後端多回一個欄位、少一個欄位，執行期不會有任何警報。所以資料**跨越系統邊界**（API 回應、`JSON.parse`、`localStorage`）時，需要執行期驗證（第 7 章會用 zod 示範）。邊界之內，才是型別系統說了算的世界。

6.2 結構型別：看形狀，不看名字

TypeScript 與 Java / C# 最根本的差異：型別相容性 **只看結構（有哪些欄位、什麼型別），不看名字或宣告關係**。

```
interface Point { x: number; y: number }

function draw(p: Point) { /* ... */ }

const spot = { x: 1, y: 2, label: 'home' };
draw(spot);   // OK！spot 沒有宣告自己「是」Point，
              // 但它的形狀滿足 Point 的要求（多的欄位不礙事）
```

`spot` 從沒說過自己實作了 `Point`——不需要。形狀對了就相容，這讓 TypeScript 能自然地描述「本來就存在的 JavaScript 程式碼」，也解釋了為什麼你可以把後端回應直接餵給任何「形狀合得上」的函式。

但有一個著名的特例會讓新手以為規則矛盾了：

```
draw({ x: 1, y: 2, label: 'home' });
//               ^^^^^ 錯誤：物件字面量只能指定已知屬性
```

同樣的物件，先存變數再傳可以，直接以字面量傳卻報錯？這是刻意設計的**多餘屬性檢查**（excess property check）：「當場寫出來的字面量」多出的欄位不可能有其他用途，幾乎可以肯定是拼錯欄位名，所以 TypeScript 特別從嚴。理解成「字面量享受拼字檢查特權」即可。

6.3 type 與 interface：怎麼選

先說結論：**兩者在「描述物件形狀」這個用途上幾乎可以互換**，多數團隊的選擇是慣例問題而不是對錯問題。有實質差異的地方：

	interface	type
描述物件 / 函式形狀	可以	可以
聯集、交集、字面量、mapped types	不行	**只有 type 能寫**
擴充	`extends`（錯誤訊息較友善）	`&` 交集
同名再宣告	**自動合併**（declaration merging）	重複宣告是錯誤

實務選擇

- 元件 props、公開 API 的物件形狀：`interface`（慣例常見、`extends` 擴充自然）。
- 聯集（`FetchState`）、字面量（`'sm' | 'md'`）、從其他型別演算出來的型別：**只能用 `type`**，這也是實戰中 `type` 越用越多的原因。
- declaration merging 幾乎只在一種場景有價值：擴充第三方套件的型別（例如幫 `express` 的 `Request` 加欄位）。自己的程式碼不要依賴它。
- 最重要的一條：**跟隨團隊現有慣例**，一致性比個人偏好值錢。

6.4 聯集與收窄：讓編譯器跟著你的 if 走

聯集型別（union）表達「這個值是其中之一」。關鍵技能是**收窄**（narrowing）：用執行期檢查說服編譯器「在這個分支裡，值只可能是某幾種」。

```
function formatId(id: string | number): string {
  // 這裡 id 是 string | number，只能用兩者共有的操作
  if (typeof id === 'number') {
    return id.toFixed(0);   // 這個分支裡 id: number
  }
  return id.toUpperCase();  // if 沒中，所以這裡 id: string
}
```

TypeScript 會**沿著控制流程**追蹤每個分支裡型別的變化——這叫 control flow analysis，是它最聰明的部分。常用的收窄手段：`typeof x === 'string'`（原始型別）、`x instanceof Date`（class）、`'items' in x`（依欄位存在區分）、`x !== null` / truthiness 檢查（去掉 `null | undefined`），以及對字面量欄位的相等比較——這最後一種威力最大，值得獨立成節。

6.5 標記聯集：實戰中最重要的型別模式

第 5 章已經給過動機（三顆布林的碎片化狀態），現在補上完整的機制。**標記聯集**（discriminated union）＝聯集的每個成員都帶著同一個「標記欄位」（通常叫 `status`、`type` 或 `kind`），其值是

互不重疊的字面量：

```
type FetchState<T> =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success'; data: T }
  | { status: 'error'; error: string };

function render(state: FetchState<User[]>) {
  switch (state.status) {
    case 'loading': return <Spinner />;
    case 'success': return <UserList users={state.data} />;
    //                                      ^^^^ 只有這個分支拿得到 data
    case 'error':   return <Alert message={state.error} />;
    case 'idle':    return null;
  }
}
```

檢查標記欄位（`switch` 或 `if`）時，編譯器自動把整個物件收窄成對應的成員——**「檢查狀態」與「取得該狀態才有的資料」變成同一個動作**，「忘記檢查就取用」從 runtime bug 變成編譯錯誤。

再加一個免費的保險——**窮盡檢查**（exhaustiveness check）：

```
default: {
  // 若上面的 case 涵蓋了所有成員，這裡的 state 已被收窄到 never（空集合）
  const unreachable: never = state;
  throw new Error(`未處理的狀態: ${unreachable}`);
}
```

半年後有人幫 `FetchState` 加了 `'cancelled'` 成員，所有漏掉這個 case 的 `switch` 會**同時亮起編譯錯誤**，一個都跑不掉——這就是「型別是重構安全網」最具體的體現。公司程式碼裡的 Redux action、WebSocket 訊息、表單流程狀態，清一色是這個模式，認得它等於拿到一把萬用鑰匙。

6.6　泛型：型別層級的參數

泛型是多數人「看得懂但寫不出來」的分水嶺。突破口是理解它的動機——看一個最小的例子，取陣列第一個元素：

```
// 方案一：寫死型別 → 每種型別複製一份，違反 DRY
function firstOfStrings(arr: string[]): string | undefined { return arr[0]; }
function firstOfUsers(arr: User[]): User | undefined { return arr[0]; }

// 方案二：用 any → 型別資訊在出口蒸發，等於自廢武功
function firstAny(arr: any[]): any { return arr[0]; }
```

```ts
const u = firstAny(users);   // u: any ——— 從這行起編譯器幫不了你了

// 方案三：泛型 → 「進來什麼，出去就是什麼」這個【關係】被保留下來
function first<T>(arr: T[]): T | undefined { return arr[0]; }
const user = first(users);      // user: User | undefined（自動推斷，不用手寫）
const name = first(['a', 'b']); // name: string | undefined
```

核心觀念：泛型保留的是「關係」

函式參數把「值」參數化：同一段邏輯適用於不同的值。泛型把「型別」參數化：同一段邏輯適用於不同的型別，並且**把輸入與輸出型別之間的關係保留下來**（進來 `T[]`，出去 `T`）。讀泛型的口訣：**尖括號裡是宣告（引入一個型別變數），後面出現的同名字母是使用**。呼叫端通常不需要手動指定——編譯器會從實際傳入的值**推斷** `T` 是什麼。

泛型預設對 `T` 一無所知，所以 `T` 上什麼操作都不能做。需要「T 至少長什麼樣」時，用 `extends` 加上**約束**：

```ts
// K extends keyof T：K 必須是「T 的欄位名」這個集合的成員
function pluck<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}
const user = { id: 1, name: 'Amy' };
pluck(user, 'name');   // 回傳型別自動是 string
pluck(user, 'email');  // 編譯錯誤：'email' 不是 user 的欄位
```

短短一行裡有三個可以帶走的零件：`keyof T`（把 T 的欄位名變成字面量聯集 `'id' | 'name'`）、`extends` 約束（限制型別參數的範圍）、`T[K]`（索引存取：欄位 K 的型別）。它們組合出「傳錯欄位名直接編譯錯誤、回傳型別隨欄位自動變化」——這正是公司程式碼裡那些「神奇地知道你要什麼」的工具函式的原理。

實戰中你天天在消費泛型而不自知：`useState<User | null>`、`Promise<Response>`、`User[]`（它就是 `Array<T>`）、第 4 章的 `useDebounce<T>`、以及所有 API 包裝層的 `ApiResponse<T>`。泛型不是進階炫技，是「不複製貼上、也不丟失型別」的唯一路徑。

6.7　讓值成為型別的唯一來源：typeof、keyof、as const

實務上常見的重複：同一份「選項清單」，值寫一遍、型別再寫一遍，兩邊要人工同步：

```ts
// 反模式：值與型別各自維護，加一個尺寸要改兩個地方
const SIZES = ['sm', 'md', 'lg'];
type Size = 'sm' | 'md' | 'lg';
```

TypeScript 提供從「值」推導「型別」的工具鏈，讓事實只有一份：

```ts
const SIZES = ['sm', 'md', 'lg'] as const;
// as const：型別從 string[] 收緊為 readonly ['sm', 'md', 'lg']
type Size = (typeof SIZES)[number];   // 'sm' | 'md' | 'lg'（自動推導）

const BUTTON_TONE = { neutral: 'gray', danger: 'red' } as const;
type Tone = keyof typeof BUTTON_TONE; // 'neutral' | 'danger'
```

拆解一下這串咒語：`as const` 告訴編譯器「這個值永遠不變，請用最窄的字面量型別記住它」；`typeof`（型別語境版）取出值的靜態型別；`[number]` / `keyof` 再從中萃取成員。之後新增尺寸只改 `SIZES` 一處，所有相關型別自動更新——**單一事實來源**原則在型別層的實踐。（順帶：這一套通常比 `enum` 更受現代 TS 團隊青睞——不引入額外的執行期結構，跟字面量聯集無縫相容。）

6.8　Utility Types：型別的標準函式庫

TypeScript 內建一批「吃型別、吐型別」的工具。先看最常用的四個在實戰裡的長相：

```ts
interface User { id: string; name: string; email: string; role: Role }

// 建立時還沒有 id（由後端配發）：
type NewUser = Omit<User, 'id'>;
// 編輯表單只允許改部分欄位：
type UserPatch = Partial<Pick<User, 'name' | 'email'>>;
// 以 id 索引的查找表：
type UsersById = Record<string, User>;
```

重點不是背表，而是看穿它們**不是魔法**——自己寫一次 `Partial` 就除魅了：

```ts
// mapped type：對 T 的每個欄位 P 逐一改造
type MyPartial<T> = {
  [P in keyof T]?: T[P];   // in keyof T = 走訪所有欄位；? = 加上可選
};
type MyPick<T, K extends keyof T> = { [P in K]: T[P] };
```

`[P in keyof T]` 讀作「對 T 的每個欄位名 P」——型別層的 `map` 迴圈。懂了這個，讀公司程式碼裡自訂的工具型別（以及 `Omit = Pick + Exclude` 這類組合）就只是普通的閱讀，不再是黑盒。

高頻 utility types 速查

`Partial<T>` 全部變可選（patch 更新）；`Required<T>` 反向；`Pick<T, K>` / `Omit<T, K>` 取 / 去欄位（從完整模型裁出子集）；`Record<K, V>` 鍵值表；`ReturnType<F>` / `Parameters<F>` 從函式反推型別（跟著既有函式走，不重複宣告）；`Exclude<T, U>` / `Extract<T, U>` 對聯集做減法 / 交集。共同精神：**從既有型別演算出新型別，而不是再手寫一份。**

6.9　any、unknown、never：三個特殊集合

- `any`：**「請編譯器閉嘴」**。它不只是「任意值」，而是雙向免檢——什麼都能塞給它、它也能塞給任何地方，而且會**傳染**：一個 `any` 流過的每個變數都跟著失去保護。每寫一個 `any`，就是在型別安全網上剪一個洞。
- `unknown`：「任意值」的**安全版**。什麼都能放進來，但**用之前必須收窄**——編譯器強迫你先檢查。「還不知道是什麼」的值（`JSON.parse` 的結果、`catch (e)` 的 e、未驗證的外部輸入）一律用它：

```
const parsed: unknown = JSON.parse(raw);
if (typeof parsed === 'object' && parsed !== null && 'id' in parsed) {
  // 收窄之後才能使用 —— any 會讓你跳過這一步，然後在執行期爆炸
}
```

- `never`：空集合，「不可能有值」。兩個實戰角色：窮盡檢查的哨兵（6.5 節）、以及永不回傳的函式（`throw` / 無限迴圈）的回傳型別。

最後是 `as` 型別斷言。它的語意是**「編譯器，你先讓開，我對這個值的了解比你多」**——斷言不做任何轉換或檢查，只是單方面壓過編譯器的判斷。正當用途確實存在（DOM API 的泛化回傳、測試裡構造假資料），但每個 `as` 都該視為一句需要理由的宣示。特別警惕連環斷言 `x as unknown as T`：它能把任何東西「變成」任何型別，出現時幾乎總是設計出了問題。**優先順序永遠是：收窄 > 型別守衛 > 斷言。**

本章重點回顧

- 型別＝值的集合；檢查＝集合相容性；聯集是併集、交集要同時滿足；小集合可放進大集合。
- TypeScript 是結構型別：形狀對就相容；物件字面量另享「多餘屬性檢查」。
- type 能表達聯集與型別演算，interface 有 extends 與宣告合併；物件形狀兩者皆可，跟隨團隊慣例。
- 標記聯集＋收窄＋窮盡檢查＝「不可能的狀態無法表達、漏處理的分支編譯期現形」。
- 泛型參數化型別並保留輸入輸出的關係；`extends` 加約束；`keyof` / `T[K]` 精準描述物件操作。
- `as const`＋`typeof`＋`keyof` 讓值成為型別的單一事實來源；utility types 是型別演算的標準庫，本質是 mapped types。
- 不明的值用 `unknown` 不用 `any`；`as` 是最後手段。

自我檢測

1. **為什麼一個從未宣告「implements Point」的物件可以傳給要求 `Point` 的函式？**
   參考答案： TypeScript 是結構型別系統：相容性只看形狀（欄位與其型別）是否滿足要求，不看名字或宣告關係。只要物件具備 `Point` 要求的欄位即相容，多的欄位不影響（但直接寫物件字面量時

會觸發多餘屬性檢查）。

2. **`function first<T>(arr: T[]): T` 相比 `(arr: any[]) => any` 好在哪？**
   參考答案： any 版本讓型別資訊在出口消失，之後的程式碼全部失去檢查與補全；泛型版本保留「進來什麼、出去什麼」的關係，且呼叫端由推斷自動得到精確型別，不需要手動標註。

3. **標記聯集裡的「標記」需要什麼條件？switch 之後為什麼能直接用 `state.data`？**
   參考答案：每個成員要有同名欄位（如 status），其型別是互不重疊的字面量。對標記做相等檢查時，control flow analysis 會把整個物件收窄到對應成員，所以 `case 'success'` 分支裡編譯器知道這個物件必有 `data`。

4. **`catch (e)` 裡的 `e` 該當成什麼型別處理？為什麼不是 any？**
   參考答案： unknown。任何東西都可能被 throw（不只 Error），unknown 強迫你先收窄（如 `e instanceof Error`）再使用，any 則允許直接 `e.message`，在 e 不是 Error 時執行期爆炸。

5. **同事寫了 `const user = res.data as User`，這行有什麼風險？有什麼替代方案？**
   參考答案： as 不做任何執行期檢查：後端格式一變，錯誤會延遲到某次取用欄位時才爆，且離源頭很遠。跨系統邊界的資料應做執行期驗證（如 zod 的 `safeParse`），驗證通過後型別自然成立，不需要斷言。

第 7 章

TypeScript × React：型別的實戰應用

本章目標——讀完後你應該能回答

- 元件 props、事件處理器、hooks 的型別該怎麼寫、怎麼查？
- 怎麼寫一個泛型元件？什麼時候需要？
- 怎麼用型別表達「這兩個 props 只能擇一」這種規則？
- 公司 UI 元件常見的 `ComponentPropsWithoutRef` 是在做什麼？
- API 層的型別怎麼組織？型別和執行期驗證的分工是什麼？

第 6 章建立了型別系統的模型，這一章把它接到 React 上——目標是讓你讀懂（並寫出）公司程式碼裡那些型別技巧的「為什麼」。

7.1 Props 的型別：元件的公開合約

```
interface UserCardProps {
  user: User;                          // 必填
  showEmail?: boolean;                 // 可選：呼叫端可以不管它
  onSelect?: (userId: string) => void;
  children?: React.ReactNode;          // 「任何可以渲染的東西」
}

function UserCard({ user, showEmail = false, onSelect, children }: UserCardProps) {
  // 可選 props 的預設值在解構時給，取代已棄用的 defaultProps
  ...
}
```

幾個值得說「為什麼」的點：

- **props 介面就是文件**。第 5 章說「介面誠實」，落實方式就是把規則寫進型別：必填 / 可選、回呼的參數是什麼，IDE 會在呼叫端即時執行這份文件。
- `children` 的型別用 `React.ReactNode`——它是「任何合法渲染內容」的聯集（元素、字串、數字、陣列、`null`…）。另一個常見選項 `React.ReactElement` 窄得多（只接受 JSX 元素，字串都不行），用在「只能塞元件」的插槽。分不清時：容器用 `ReactNode`，九成情況正確。
- **回呼的參數用領域資料**（`userId: string`）而不是 React 事件——呼應第 5 章：洩漏越少，元件越可重用。

7.2 事件型別：用 IDE 查，不用背

需要碰原生事件（拿 `e.target.value`、`preventDefault`）時才會遇到事件型別。不用背——**在 JSX 上寫 inline handler，滑鼠停在 `e` 上，IDE 直接告訴你答案**，把它抄出來即可。高頻的三個：

場景	型別
輸入框 `onChange`	`React.ChangeEvent<HTMLInputElement>`
按鈕 `onClick`	`React.MouseEvent<HTMLButtonElement>`
表單 `onSubmit`	`React.FormEvent<HTMLFormElement>`

讀法本身是複習：`ChangeEvent<HTMLInputElement>` 是泛型——「change 事件，發生在 input 元素上」，所以 `e.target` 的型別精確到 `HTMLInputElement`，`e.target.value` 才拿得到。泛型在這裡保留的正是「事件與元素種類」的關係。

7.3 useState 與 useRef 的型別

`useState` 多數時候靠推斷就好（`useState(0)` 自然是 `number`）。**需要手動標註的訊號只有一個：初始值無法代表完整的型別**。最典型的兩種：

```
// 1. 初始值是 null，但之後會裝 User —— 不標註的話型別被推斷成 null
const [user, setUser] = useState<User | null>(null);

// 2. 初始值是空陣列 —— 不標註會推斷成 never[]，之後什麼都塞不進去
const [items, setItems] = useState<Item[]>([]);
```

`<User | null>` 這個聯集不是累贅，是誠實：「還沒載入」是真實存在的狀態，型別把它寫出來之後，每個使用 `user` 的地方都被迫先處理 `null`——於是「忘記處理載入中」從線上錯誤變成紅線。（進一步的版本你已經在第 5、6 章看過：多狀態時直接用標記聯集 `FetchState<T>`。）

`useRef` 的兩種簽名對應第 4 章的兩種用途：

```
const inputRef = useRef<HTMLInputElement>(null);  // DOM ref：交給 ref 屬性
const timerId  = useRef<number | null>(null);     // 可變盒子：自己讀寫 current
```

7.4　泛型元件：讓「一份 UI 邏輯」適用所有資料型別

第 6 章的泛型函式概念，原封不動搬到元件上。需求場景：公司裡一定有的通用列表——載入、空狀態、逐項渲染，邏輯對 `User[]`、`Order[]` 完全相同，只差「一項長怎樣」：

```
interface ListProps<T> {
  items: T[];
  keyOf: (item: T) => string;              // 身分函式（回想第 3 章的 key）
  renderItem: (item: T) => React.ReactNode; // 每一項的長相交給呼叫端
  empty?: React.ReactNode;
}

function List<T>({ items, keyOf, renderItem, empty }: ListProps<T>) {
  if (items.length === 0) return <>{empty ?? '沒有資料'}</>;
  return <ul>{items.map((it) => <li key={keyOf(it)}>{renderItem(it)}</li>)}</ul>;
}

// 呼叫端：T 由 items 推斷為 User，於是 renderItem 的參數自動是 User
<List
  items={users}
  keyOf={(u) => u.id}
  renderItem={(u) => <UserCard user={u} />}  // u: User，全程有補全
/>
```

注意型別在其中鎖住的關係：`items`、`keyOf`、`renderItem` 三個 props **被同一個 `T` 綁在一起**——傳 `users` 進來，`renderItem` 就不可能收到別的型別。沒有泛型的話，這三者只能各自宣告成 `any` 系列，關係就斷了。這個「插槽＋泛型」組合（`renderItem` 這種 props 叫 render prop）是元件庫 Table、Select、Autocomplete 的標準骨架，公司程式碼裡一定找得到。

.tsx 檔的一個小陷阱

在 `.tsx` 裡寫**箭頭函式版**的泛型會撞到語法歧義：`const fn = <T>(x: T) => x` 的 `<T>` 會被當成 JSX 標籤而報錯。解法：加個逗號 `<T,>`，或改用 `function` 宣告（泛型元件慣用後者）。知道這個，就不會在第一次寫泛型元件時卡半小時。

7.5　用標記聯集設計 props：把使用規則寫進型別

第 5 章遺留了一個問題：「兩個 props 只能擇一」怎麼表達？場景：`Button` 既可以是真按鈕（要 `onClick`），也可以是連結的樣子（要 `href`），但不能兩個都給。用可選欄位寫 `onClick?: ...; href?: ...` 的話，「都給」「都不給」這兩種錯誤用法照樣通過編譯。標記聯集登場：

```
type ButtonProps = { tone?: 'neutral' | 'danger'; children: React.ReactNode } & (
  | { as: 'button'; onClick: () => void; href?: never }
  | { as: 'link'; href: string; onClick?: never }
);
```

```
<Button as="button" onClick={save}>儲存</Button>   // OK
<Button as="link" href="/docs">文件</Button>        // OK
<Button as="link" onClick={save} href="/x">壞</Button> // 編譯錯誤！
```

拆解：共用欄位放交集（&）的左邊；互斥的部分做成以 as 為標記的聯集；href?: never 是慣用技——「這個欄位在這個變體裡**不准出現**」（never 是空集合，你給任何值都塞不進去）。使用端一寫 as="link"，IDE 的補全立刻切換成該變體的欄位。**元件的使用規則不再寫在註解或口頭交接裡，而是編譯器代為執法。**

7.6 包裝原生元素：ComponentPropsWithoutRef

公司 UI 庫的 Button、Input 幾乎都要「支援原生元素的所有屬性」（disabled、aria-*、form …上百個）。手抄是不可能的，標準做法是**直接繼承原生元素的 props 型別**：

```
type ButtonProps = React.ComponentPropsWithoutRef<'button'> & {
  tone?: 'neutral' | 'danger';       // 再交集上自己的擴充
  isLoading?: boolean;
};

function Button({ tone = 'neutral', isLoading, children, ...rest }: ButtonProps) {
  return (
    <button className={toneClass[tone]} disabled={isLoading} {...rest}>
      {isLoading ? <Spinner /> : children}
    </button>
  );
}
```

ComponentPropsWithoutRef<'button'> 讀作「原生 <button> 能接受的全部 props（不含 ref）」。搭配 ...rest 轉發，這個元件自動支援所有原生屬性與未來新增的屬性，型別與行為一次到位——這就是你在公司元件庫裡看到這個模式的原因。（對應地，ComponentProps<typeof SomeComponent> 可以取出**任何元件**的 props 型別，在「包裝第三方元件」時同樣好用。）

7.7 自訂 hook 的回傳型別

自訂 hook 是普通函式，但有一個專屬陷阱——回傳「陣列對」的時候：

```
function useToggle(initial = false) {
  const [on, setOn] = useState(initial);
  const toggle = useCallback(() => setOn((v) => !v), []);
  return [on, toggle] as const;
  // 沒有 as const：推斷成 (boolean | (() => void))[] ——— 順序與長度都丟了
  // 有 as const：readonly [boolean, () => void] ——— 精確的 tuple
}
```

規則：**回傳兩個以內、順序自明的值，用 tuple + `as const`**（模仿 `useState` 的手感，呼叫端可自由命名）；**回傳三個以上或語意需要名字的，改回傳物件**（如第 5 章的 `useUserSearch` 回傳 `{ keyword, setKeyword, users, status }`）。

7.8 API 層：型別的單一來源與執行期的守門員

把前面所有工具組裝成公司專案裡最重要的一層。原則有三條：

（一）領域型別集中定義、單點維護。 `User`、`Order` 這些型別放在共用的 `types.ts`（或按領域拆檔），元件、hook、API 全部 import 同一份——後端改欄位時，改這一處，編譯器把所有受影響的位置標紅給你（重構安全網的日常用法）。

（二）用泛型寫一次請求邏輯。

```
async function api<T>(path: string, init?: RequestInit): Promise<T> {
  const res = await fetch(`${BASE_URL}${path}`, init);
  if (!res.ok) throw new ApiError(res.status, await res.text());
  return res.json() as Promise<T>;
}

// 端點層：每個函式就一行，型別關係全部自動流動
export const fetchUser = (id: string) => api<User>(`/users/${id}`);
export const searchUsers = (q: string) => api<Paginated<User>>(`/users?q=${q}`);
```

（三）記住 `Promise<T>` 是承諾，不是驗證。上面 `as Promise<T>` 那行就是第 6 章講的「跨系統邊界」——型別在這裡是**宣告**出來的，不是檢查出來的。對格式穩定的內部 API，多數團隊接受這個信任；對關鍵路徑或第三方 API，用 zod 這類函式庫把「驗證」與「型別」合一：

```
import { z } from 'zod';

const UserSchema = z.object({
  id: z.string(),
  name: z.string(),
  email: z.string().email(),
});
type User = z.infer<typeof UserSchema>;   // 型別直接從 schema 推導，單一來源

const data = UserSchema.parse(await res.json());
// 通過 = data: User（貨真價實）；不通過 = 立刻在邊界丟出清楚的錯誤，
// 而不是讓髒資料流進深處，在某個元件裡以 undefined 錯誤的形式引爆
```

tsconfig 的一個開關值得你確認

打開公司專案的 `tsconfig.json` 看 `strict` 是否為 `true`。它裡面最有價值的是 `strictNullChecks`：關掉時，`null` / `undefined` 可以塞進任何型別，「`user` 可能是 null」這種資訊型別系統根本表達不出來——本章一半的技巧會失效。新專案一律開 `strict`；舊專案若沒開，讀碼時要自行提高警覺（型別寫著 `User`，實際可能是 `null`）。

本章重點回顧

- props 介面是元件的公開合約：children 用 `ReactNode`，回呼參數用領域資料；預設值在解構時給。
- 事件型別用 IDE hover 查；`ChangeEvent<HTMLInputElement>` 的泛型參數保留了「事件 × 元素」的關係。
- `useState` 只在「初始值不能代表全型別」時手動標註（`<User | null>`、`<Item[]>`）。
- 泛型元件用一個 `T` 把 `items` / `keyOf` / `renderItem` 綁定成一致的整體。
- 標記聯集＋`never` 欄位可表達「擇一」的 props 規則；`ComponentPropsWithoutRef` 讓包裝元件繼承原生屬性。
- hook 回傳 tuple 要 `as const`；三個以上的值改回傳物件。
- 領域型別單一來源；泛型 fetcher 讓型別自動流動；邊界資料用 zod 把驗證與型別合一。

自我檢測

1. **`useState(null)` 之後想 `setUser(user)` 被紅線，為什麼？怎麼修？**
   參考答案：沒有標註時初始值 null 被推斷成型別 null，之後任何非 null 的值都塞不進去。改成 `useState<User | null>(null)`，同時這也誠實表達了「尚未載入」狀態，強迫所有使用點處理 null。

2. **泛型元件 `List<T>` 裡，為什麼呼叫端的 `renderItem` 參數自動有正確型別？**
   參考答案：items、keyOf、renderItem 共用同一個型別參數 T；呼叫端傳入 `users: User[]` 時，編譯器從 items 推斷 T = User，於是同一份 T 流進 renderItem 的參數型別，三個 props 被鎖定成一致。

3. **怎麼用型別表達「`href` 和 `onClick` 不能同時出現」？**
   參考答案：用標記聯集拆成兩個變體（如 `as: 'link'` 與 `as: 'button'`），各自要求自己的欄位，並用 `other?: never` 禁止對方的欄位出現；共用欄位用交集掛在外層。錯誤組合會直接編譯失敗。

4. **`api<T>()` 回傳的 `Promise<User>`，和 zod 驗證後得到的 `User`，可信度差在哪？**
   參考答案：前者是編譯期的單方面宣告（`res.json()` 實際回什麼沒人檢查），後端改格式時型別不會報警；後者經過執行期驗證，資料真的符合 schema 才會通過，且型別由 `z.infer` 從 schema 推導，驗證與型別是同一份事實。

5. **公司元件 `<Input />` 能接受 `maxLength`、`aria-label` 等所有原生屬性，通常是怎麼做到的？**

參考答案： props 型別宣告為 `ComponentPropsWithoutRef<'input'> & {自訂欄位}`，實作中把解構剩下的 `...rest` 轉發給原生 `<input>`，型別與行為都自動涵蓋全部原生屬性。

第 8 章

元件實戰：從零打造一個浮動視窗

本章目標——讀完後你應該能回答

- 浮動層（對話框、下拉選單、浮動視窗）為什麼幾乎都用 `createPortal`？它搬走了什麼、留下了什麼？
- 拖曳這種每秒上百次事件的互動，狀態該怎麼設計才不會拖垮渲染？
- `setPointerCapture` 如何讓拖曳不需要 effect？什麼時候仍然該用 effect 版？
- 多個視窗的疊層順序誰說了算？「點一下置頂」怎麼做才不弄髒渲染的純粹性？

前面章節的元件，工作大多是「資料進、畫面出」；浮動視窗是另一種生物——它要跟瀏覽器的座標系統、指標事件、疊層順序打交道，還得在高頻互動下保持流暢。這類元件最容易寫成「能動就好」的一團事件處理器，也因此最能檢驗你對 React 模型的掌握。這一章從需求開始，完整打造一個 `FloatingWindow`：可拖曳、可縮放、可堆疊、支援鍵盤與觸控。每一步都不是新魔法，而是前七章觀念的落地。建議打開導讀提到的實驗專案跟著做——這章的程式碼全部可以直接執行。

8.1 需求與介面：先設計「用起來的樣子」

需求清單，來自浮動視窗在真實產品裡的典型用途（樓層平面圖、迷你播放器、客服對話、監控面板）：

- 內容任意——它是個容器，裝什麼由使用端決定。
- 標題列可拖曳移動；右下角可拖曳縮放；有關閉鍵。
- 可以同時開好幾個；點到哪個，哪個浮到最上面。
- 拖不出視口太遠（至少留一角抓得回來）；按 ESC 可關閉；觸控裝置也能拖。

第 5 章的順序在這裡直接套用：**先寫使用端想像圖，再回頭實作**。

```
// 使用端想像圖：先寫這段，再實作 FloatingWindow
function ReportPage() {
  const [showPlan, setShowPlan] = useState(false);
  return (
    <>
      <button onClick={() => setShowPlan(true)}>樓層平面圖</button>
      {showPlan && (
        <FloatingWindow
```

```
          title="樓層平面圖"
          defaultPosition={{ x: 120, y: 96 }}
          onClose={() => setShowPlan(false)}
        >
          <FloorPlan floor={3} />
        </FloatingWindow>
      )}
    </>
  );
}
```

從想像圖反推出 props 介面：

```
export interface Point { x: number; y: number }
export interface Size  { width: number; height: number }

export interface FloatingWindowProps {
  title: string;
  onClose: () => void;          // 關閉的決定權在外面：視窗只回報意圖
  defaultPosition?: Point;      // 非受控：位置由視窗自己管
  defaultSize?: Size;
  minSize?: Size;               // 縮放的下限
  children: React.ReactNode;    // 內容是插槽（第 5 章）
}
```

這份介面裡藏著兩個第 5 章「受控 / 非受控」的決策，值得攤開講：

視窗的「存在」是受控的。 `FloatingWindow` 沒有 `isOpen` 這種內部開關——開著與否由父層的條件渲染決定，視窗只透過 `onClose` 回報「使用者想關」。理由：開關幾乎必然要跟外部聯動（按鈕、路由、快捷鍵），主導權該在外面；而且「關閉＝卸載」讓內部狀態自然歸零（第 3 章的協調規則），下次打開回到初始位置——「記住上次位置」是之後的擴充，8.10 節再談。

位置與尺寸是非受控的。 多數使用端根本不想指揮它們，給個 `defaultPosition` 起步就好。刻意取名 `defaultPosition` 而不是 `initialPosition`，是替未來的受控介面（`position`＋`onPositionChange`）留下名字空間——成熟元件庫 `value` / `defaultValue` 雙介面的慣例（第 5 章）。

8.2　Portal：DOM 出走，資料流留下

第一個實作決策跟畫面無關，跟「住址」有關。`FloatingWindow` 在 React 樹裡自然住在擁有它的功能底下（上例的 `ReportPage`），資料才拿得順手。但它的 **DOM 節點**若也住在 `ReportPage` 的版面裡，三件麻煩會找上門：

- 祖先的 `overflow: hidden` 會把拖出去的視窗**裁掉**。
- 祖先建立的 stacking context 會困住 `z-index`——它只在同一個 stacking context 內比大

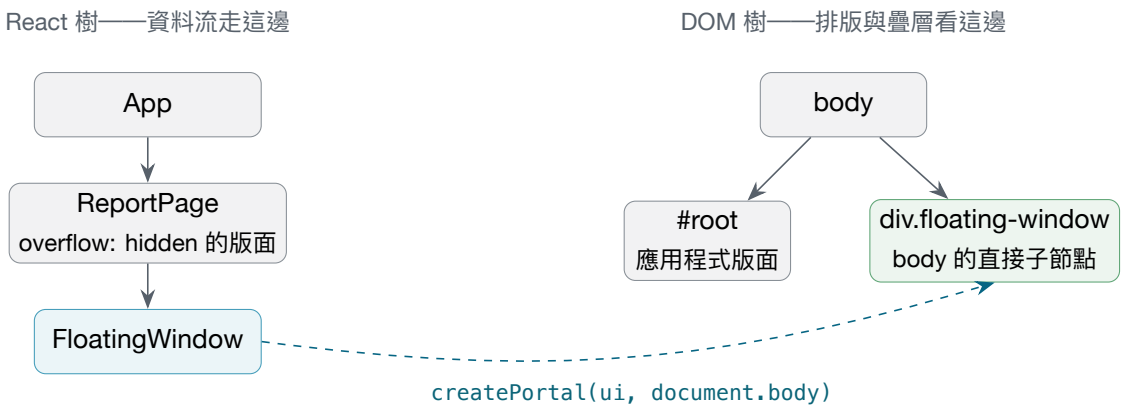
小，內層寫 9999 也翻不出去。

- 冷知識：祖先有 `transform`（或 `filter`）時，`position: fixed` 會改以那個祖先為定位基準——「fixed 突然不 fixed 了」的元兇多半是它。

「元件樹上住在邏輯歸屬處」與「DOM 上浮在整頁頂層」是兩個矛盾的需求，`createPortal` 正是解這個矛盾的工具：

```tsx
import { createPortal } from 'react-dom';

function FloatingWindow(props: FloatingWindowProps) {
  return createPortal(
    <div className="floating-window">...</div>,    // 畫面，接下來逐步填入
    document.body,                                  // DOM 的落點
  );
}
```



核心觀念：Portal 只搬 DOM，不搬資料流

- 狀態、props、context 全部照 **React 樹**走：`FloatingWindow` 照常讀得到 `ReportPage` 之上的任何 Provider。
- 事件也照 React 樹**冒泡**：視窗內的點擊會冒泡到 `ReportPage` 的處理器——即使 DOM 上兩者毫無祖孫關係。
- 所以 portal 元件寫起來與一般元件**沒有任何差別**；差別只發生在瀏覽器的排版、裁切與疊層。這也是元件庫的對話框、下拉選單、tooltip 全家都 portal 的原因。

掛載點的兩個實務細節

(1) SSR 專案裡，`document` 在伺服器端不存在，直接寫 `document.body` 會在伺服器渲染時炸掉。慣用解法是「掛載後才渲染 portal」：用一個 effect 把 `mounted` 設為 true，之前先回傳 `null`。單頁應用沒有這個問題。(2) 許多專案不直接掛 `body`，而是掛到自己準備的 `<div id="overlay-root">`：方便統一控制浮動層的順序、樣式作用域與測試時的清理。機制完全相同。

8.3 位置狀態與第一版拖曳：同步模型的推導

位置與尺寸會直接反映在畫面上——第 4 章的判準（值變了，畫面要跟著變嗎？）答案是「要」，所以是 state：

```
function FloatingWindow({
  title, onClose, children,
  defaultPosition = { x: 80, y: 80 },
  defaultSize = { width: 360, height: 260 },
}: FloatingWindowProps) {
  // 初始值已能代表完整型別，不需要手動標註（第 7 章）
  const [position, setPosition] = useState(defaultPosition);
  const [size, setSize] = useState(defaultSize);

  return createPortal(
    <div
      className="floating-window"    // 陰影、圓角等外觀交給 CSS，這裡只放機制
      style={{
        position: 'fixed', left: 0, top: 0,
        transform: `translate(${position.x}px, ${position.y}px)`,
        width: size.width, height: size.height,
      }}
    >
      <header className="fw-titlebar">{title}</header>
      <main className="fw-content">{children}</main>
    </div>,
    document.body,
  );
}
```

兩個容易被忽略的決定：

- **用 `transform` 移動，而不是改 `left` / `top`**。後者參與排版，每次改動都觸發瀏覽器重新 layout；`transform` 只動合成層。拖曳每秒改上百次位置，這筆帳值得算。狀態（`position`）是畫面事實，`transform` 只是呈現手段——兩者分離。
- `useState(defaultPosition)` 的初始值**只在首次渲染被讀取**，之後父層再改這個 prop 也不會跟著動——這正是「非受控」在程式碼上的意思。

接著讓標題列動起來。拖曳是三段式的手勢：**按下**（記住「抓住點離視窗左上角多遠」）→ **移動**（指標座標減抓點偏移＝視窗新位置）→ **放開**。按下發生在標題列上，這沒問題；但移動呢？指標一拖快就會甩出標題列的範圍，掛在標題列上的 `onPointerMove` 就收不到事件了。第一直覺：監聽整個 `window`。於是需求變成「**拖曳中**這段期間，`window` 上要存在一組監聽器；結束就要消失」——「讓外部系統跟上狀態」，這是第 4 章 effect 同步模型的標準劇本：

```
const [dragging, setDragging] = useState(false);
```

```
const grabOffset = useRef<Point>({ x: 0, y: 0 });  // 抓住點離視窗左上角多遠

function onTitlePointerDown(e: React.PointerEvent) {
  grabOffset.current = { x: e.clientX - position.x, y: e.clientY - position.y };
  setDragging(true);
}

// 同步：「拖曳中」時 window 上存在監聽器；結束時消失
useEffect(() => {
  if (!dragging) return;
  function onMove(e: PointerEvent) {
    setPosition({                       // 絕對式計算：不讀舊 position
      x: e.clientX - grabOffset.current.x,
      y: e.clientY - grabOffset.current.y,
    });
  }
  function onUp() { setDragging(false); }
  window.addEventListener('pointermove', onMove);
  window.addEventListener('pointerup', onUp);
  return () => {                        // 依賴變了或卸載：拆掉這一輪的監聽器
    window.removeEventListener('pointermove', onMove);
    window.removeEventListener('pointerup', onUp);
  };
}, [dragging]);

// JSX:<header onPointerDown={onTitlePointerDown} ...>
```

順帶一提，用 pointer 系列事件而不是 mouse 系列，是因為它把滑鼠、觸控、觸控筆統一成同一組事件——一套程式碼三種輸入。逐項對帳這段程式碼的決策：

- `grabOffset` 是**機制資料**：拖曳全程都用它換算，但畫面上沒有任何東西顯示它——判準答案是「不用跟著變」，所以住 ref（第 4 章）。放 state 的話，每次按下標題列都會白白多渲染一次。
- **依賴陣列誠實嗎？**effect 讀到的 reactive 值只有 `dragging`。`grabOffset.current` 是 ref（非 reactive）、`setPosition` 引用天生穩定；最關鍵的是 `onMove` **沒有讀 `position`**——新位置由指標座標減抓點偏移「絕對式」算出，不在舊位置上累加。誠實。

常見誤解：拖曳邏輯裡的過期閉包

把 `onMove` 寫成累加式，是浮動視窗最經典的 bug：`setPosition({ x: position.x + e.movementX, ... })`。`dragging` 變 true 的那次渲染建立了這個閉包，它捕捉的 `position` 是那一刻的快照（第 3、4 章）；之後每次 move 都在同一個舊值上疊加，視窗動一步就卡住或不斷跳回原地。修法：函數式更新 `setPosition((p) => ...)`，或像上面那樣改用絕對式計算，從根本不讀 `position`。把 `position` 加進依賴陣列也「對」，但代價是每動一格就拆掛一次 window 監

> 聽器——誠實但粗糙。

8.4 setPointerCapture：讓事件回到元素身上

effect 版能用，也是通用解，但指標事件有一個**為拖曳而生**的 API：元素的 `setPointerCapture`。把事件的 `pointerId` 交給它之後，這顆指標後續的 move / up 一律派送給該元素——無論指標飄到頁面哪裡。「事件甩出元素」的問題直接消失，window 監聽器與那顆 effect 也就沒有存在的理由：

```
const grab = useRef<Point | null>(null);   // null = 沒在拖；有值 = 抓點偏移

function onTitlePointerDown(e: React.PointerEvent<HTMLElement>) {
  grab.current = { x: e.clientX - position.x, y: e.clientY - position.y };
  e.currentTarget.setPointerCapture(e.pointerId);  // 這顆指標之後的事件都送回來
}
function onTitlePointerMove(e: React.PointerEvent<HTMLElement>) {
  if (!grab.current) return;                       // 沒抓住，只是滑過標題列
  setPosition(keepReachable(
    { x: e.clientX - grab.current.x, y: e.clientY - grab.current.y },
    size,
  ));
}
function onTitlePointerUp() { grab.current = null; }  // capture 會自動釋放

// JSX:
// <header onPointerDown={onTitlePointerDown}
//         onPointerMove={onTitlePointerMove}
//         onPointerUp={onTitlePointerUp}>
```

注意 `dragging` 這個 state **整個消失了**：「是否拖曳中」不再影響任何畫面輸出，它退化成「`grab.current` 是否為 null」——ref 就夠了（判準始終是同一條）。之後若想在拖曳中換游標樣式或加陰影，再把它升級回 state 即可。

上面順手加了 `keepReachable`：把位置夾在「至少留一角抓得回來」的範圍內。它是純函式，跟 React 無關，可以單獨測試：

```
const clamp = (v: number, lo: number, hi: number) =>
  Math.min(Math.max(v, lo), hi);

const EDGE = 48;   // 拖出視口時，至少留這麼多可以抓回來

function keepReachable(pos: Point, size: Size): Point {
  return {
    x: clamp(pos.x, EDGE - size.width, window.innerWidth - EDGE),
    y: clamp(pos.y, 0, window.innerHeight - EDGE),  // 標題列不准穿出上緣
```

```
  };
}
```

刻意「允許推出去大半、留一角」而不是「完全不准出去」：使用者常想把視窗推到旁邊只露一角。`y` 的下限是 0，因為標題列一旦穿出上緣就再也抓不回來了。另外注意它讀了 `window.innerWidth`——在**事件路徑**裡讀外部環境是正常的；要純的是**渲染**（第 1 章的界線）。

effect 版與 capture 版怎麼選

- **capture 版**：事件都能派送回元素的手勢（拖曳、縮放、畫布塗抹）首選。少一個 state、少一顆 effect，事件的生命週期天然跟著元件卸載走。
- **effect + window 監聽版**：當事件本來就不發生在自己身上——全域快捷鍵、點擊外部關閉、監聽 `window` 的 resize——同步模型仍是通用解。
- 公司程式碼兩種都會遇到；能認出它們是「同一件事的兩種載體」，讀碼就不慌。這也複習了第 4 章的判斷：**事件能在元素上接完，就不需要 effect**。

8.5　每秒上百次 setPosition：誰在重新渲染？

高刷新率螢幕加上高回報率滑鼠，`pointermove` 每秒可達一兩百次，每次都 `setPosition`，也就是每次都重新渲染 `FloatingWindow`。該慌嗎？

第 3 章的答案：渲染的成本＝函式重跑＋子樹 diff，貴不貴要看**子樹多大**。視窗的「殼」（標題列、外框、縮放把手）只有幾個節點，重算一百次也無感。真正的問號是**內容**——如果 `children` 裝的是幾百個節點的平面圖呢？

好消息：**內容根本不會重算**。推理只用第 3 章的既有規則：

- `<FloorPlan />` 這個元素物件是**父層**（`ReportPage`）渲染時建立的，經 `children` prop 交給視窗。
- 拖曳觸發的是 `FloatingWindow` **自己的** setState；父層沒有重新渲染，`children` 的引用自然原封不動。
- React 協調時發現「這個位置的元素跟上次是**同一個物件**」，直接沿用整棵子樹——連 `FloorPlan` 函式都不會被呼叫。可以理解成免費版的 `memo`：引用相同這件事本身就是「沒變」的證明。

於是 8.1 節「內容做成插槽」的決定，在這裡兌現了第二次紅利：第 5 章買到 API 的開放性，第 3 章買到「拖曳只付殼的錢」。反過來說，若把內容寫死在 `FloatingWindow` 內部，每秒上百次的整棵重算就會如實發生——**元件邊界就是渲染邊界**，拆元件的第三個理由（狀態隔離，第 5 章）在這裡具象化了。

驗證方式一如第 3 章：打開 Profiler 錄一段拖曳，內容元件不應該出現在任何一次 commit 的名單裡。先量測，再談優化。

終極逃生口：拖曳期間繞過 React

如果量測顯示連「殼」的重算都有感（殼上掛了很重的東西），還有最後一招：拖曳期間不進 React——`pointermove` 裡直接 `el.style.transform = ...`（用 ref 拿 DOM），`pointerup` 時才 `setPosition` 一次，把最終位置提交回狀態。中間位置沒有任何其他元件需要知道，不放進 state 並不違反單一事實來源；但它放棄了「畫面＝ f(state)」的即時一致性，屬於量測到問題才動用的逃生口，不是預設寫法。React 官方對 DOM 逃生口的態度一貫如此：門開著，但走之前先確認非走不可。

8.6　疊層：點一下，浮到最上面

多視窗需求：點到哪個，哪個到最前面。「誰的 `z-index` 最大」是**全頁共同的事實**，但為了它動用 context 或狀態提升，殺雞用牛刀。浮動層的務實慣例是一個**模組層計數器**：

```
let topZ = 100;   // 模組層的「目前最高樓層」，同頁所有視窗共用

function FloatingWindow(...) {
  const [zIndex, setZIndex] = useState(topZ);

  useEffect(() => { setZIndex(++topZ); }, []);   // 掛載：新視窗直接置頂

  function bringToFront() {
    if (zIndex !== topZ) setZIndex(++topZ);      // 已在頂端就不再疊高
  }

  return createPortal(
    <div onPointerDown={bringToFront} style={{ /* ... */ zIndex }}>
      ...
```

這一小段藏了三個值得對帳的決策：

- **計數器只在事件與 effect 裡遞增，渲染中絕不碰。**渲染必須純（第 1 章）：同一份狀態渲染兩次必須得到同一個畫面。寫成 `style={{ zIndex: ++topZ }}` 的話每次渲染都在改變世界，StrictMode 的雙跑立刻讓它穿幫。事件與 effect 才是雜質的合法住所——掛載 effect 在 StrictMode 下會多跳一號，但 `z-index` 只在乎相對大小，無妨。
- `if (zIndex !== topZ)` 是道小守門：點擊已置頂的視窗時，不徒增樓層、不多渲染一次（第 3 章：沒必要的 setState 就別發）。注意不能寫成 `setZIndex((z) => z === topZ ? z : ++topZ)`——updater 必須是純函式，React 可能重跑它；雜質留在事件本體。
- `onPointerDown` 掛在**根節點**：點內容、點標題列都會冒泡上來，整扇視窗任何地方按下去都置頂。

元件庫等級的做法會再進一步：一個視窗管理員 context 用**有序陣列**管所有視窗，`z-index` 就是陣列索引，能實作「關掉最上面那個」「全部收合」這類群體行為，z 值也不會無限長高。代價是

使用端要包 Provider。計數器版本是九十分的解：夠用、零依賴；等真的出現群體行為的需求，再把狀態升級到管理員手上（第 2 章：擁有者跟著需求走）。

8.7　縮放與 usePointerDrag：同一支舞，只跳一次

右下角的縮放把手，跳的是跟拖曳**完全同一支舞**：按下記錨點、capture、移動算差值、放開收尾。差別只有一個——差值**套用到誰身上**：拖曳改 `position`，縮放改 `size`。兩份幾乎一樣的「ref ＋三個處理器」，正是第 4 章說的訊號：重複的組合，抽自訂 hook。

設計 hook 介面時只問一件事：它**不需要知道**被拖的是位置還是尺寸——它只負責手勢，把「這次移動了多少」回報出去：

```
// usePointerDrag.ts ——— 把「pointer 拖曳手勢」封裝成可復用的能力
import { useRef } from 'react';

export function usePointerDrag(onDelta: (dx: number, dy: number) => void) {
  const last = useRef<{ x: number; y: number } | null>(null);  // null = 沒在拖

  const stop = () => { last.current = null; };

  return {
    onPointerDown(e: React.PointerEvent<HTMLElement>) {
      last.current = { x: e.clientX, y: e.clientY };
      e.currentTarget.setPointerCapture(e.pointerId);
      e.preventDefault();                // 拖曳途中不要順手選取文字
    },
    onPointerMove(e: React.PointerEvent<HTMLElement>) {
      if (!last.current) return;         // 只是滑過，沒有抓住
      onDelta(e.clientX - last.current.x, e.clientY - last.current.y);
      last.current = { x: e.clientX, y: e.clientY };
    },
    onPointerUp: stop,
    onPointerCancel: stop,               // 來電、系統手勢打斷時也要收尾
  };
}
```

使用端把兩個互動各接一句話，然後展開到對應的元素上：

```
const dragHandlers = usePointerDrag((dx, dy) =>
  setPosition((p) => keepReachable({ x: p.x + dx, y: p.y + dy }, size)),
);
const resizeHandlers = usePointerDrag((dx, dy) =>
  setSize((s) => ({
    width:  Math.max(minSize.width,  s.width  + dx),
    height: Math.max(minSize.height, s.height + dy),
  })),
```

```
);

// <header {...dragHandlers}>...</header>
// <div className="fw-resize" {...resizeHandlers} />
```

幾個值得內化的觀察：

- **增量回報＋函數式更新是一對**。hook 回報 dx / dy，呼叫端用 `set((prev) => ...)` 把增量疊上去——hook 完全不讀呼叫端的狀態，自然沒有依賴問題、沒有過期閉包，呼叫端也不需要 `useCallback`。第 3、4 章的函數式更新在這裡變成慣性。
- **回傳物件的欄位名就是 DOM 的事件 props**，`{...dragHandlers}` 展開即完成接線，型別也逐欄對得上。順手把 `onPointerCancel` 收進去——抽象的另一個紅利：邊角案例修一處，拖曳與縮放同時受惠。
- **抽象不是免費的**。增量式疊加配上 `keepReachable` 之後，視窗被夾在邊界時「指標離抓住點多遠」這個資訊丟了——指標一回頭視窗立刻跟動，與桌面系統「回到抓住點才動」的手感不同。絕對式（8.4 節）保真，但 hook 介面就得多傳東西。取捨要自覺；想改回絕對式，自我檢測第 5 題請你動手。

8.8 鍵盤、觸控與可及性

浮動視窗不該只對滑鼠友善。三件基本禮儀：

```
const rootRef = useRef<HTMLDivElement>(null);
useEffect(() => { rootRef.current?.focus(); }, []);  // 開窗即聚焦：ESC 立即可用

// 根節點：
<div
  ref={rootRef}
  role="dialog"                 // 讓輔助技術知道這是一扇對話視窗
  aria-label={title}
  tabIndex={-1}                 // 容器可聚焦，才接得到鍵盤事件
  onKeyDown={(e) => { if (e.key === 'Escape') onClose(); }}
>
```

- **聚焦是 DOM 指令，不是狀態**——ref ＋掛載 effect，第 4 章列過的正當 DOM 逃生口。
- **ESC 掛在視窗上而不是 `window` 上**：鍵盤事件從焦點所在元素冒泡上來，焦點在哪扇視窗，ESC 就關哪扇——多視窗時這正是你要的語意；掛全域反而得自己判斷「誰該收」。
- **誠實的邊界**：這是「非模態」的浮動視窗。模態對話框——遮罩、焦點圈禁、`aria-modal`、關閉時把焦點還給觸發元素——是規格嚴得多的另一個元件；原生 `<dialog>` 與成熟元件庫已經把它做對，不要拿浮動視窗冒充模態。

觸控與平台細節清單

- 標題列與縮放把手**必須**加 `touch-action: none`：否則觸控裝置會把手勢解讀成捲動頁面，`pointermove` 根本不派送——「手機上拖不動」十之八九是漏了它。
- 把手加 `user-select: none`（搭配 `pointerdown` 的 `preventDefault`），拖快時不會順手反白半個頁面。
- `cursor: move` / `nwse-resize` 給滑鼠使用者「這裡可以拖」的暗示。
- 關閉鍵在 `pointerdown` 停止冒泡，免得「按關閉」被標題列當成「開始拖曳」。
- 雙擊標題列收合 / 展開是桌面使用者的肌肉記憶，`onDoubleClick` 一行接上（附錄有這題練習）。

8.9　完整組裝與決策對帳

把前面各節組裝起來。`usePointerDrag` 沿用 8.7 節，這裡是視窗本體的完整程式碼：

```
// FloatingWindow.tsx ——— 完整組裝。外觀交給 CSS，這裡只保留機制必需的樣式
import { useEffect, useRef, useState } from 'react';
import { createPortal } from 'react-dom';
import { usePointerDrag } from './usePointerDrag';

export interface Point { x: number; y: number }
export interface Size  { width: number; height: number }

export interface FloatingWindowProps {
  title: string;
  onClose: () => void;
  defaultPosition?: Point;
  defaultSize?: Size;
  minSize?: Size;
  children: React.ReactNode;
}

const clamp = (v: number, lo: number, hi: number) =>
  Math.min(Math.max(v, lo), hi);

const EDGE = 48;                        // 拖出視口時，至少留一角抓得回來
function keepReachable(pos: Point, size: Size): Point {
  return {
    x: clamp(pos.x, EDGE - size.width, window.innerWidth - EDGE),
    y: clamp(pos.y, 0, window.innerHeight - EDGE),
  };
}

let topZ = 100;                         // 模組層計數器：只在事件與 effect 中遞增
```

```tsx
export function FloatingWindow({
  title,
  onClose,
  defaultPosition = { x: 80, y: 80 },
  defaultSize = { width: 360, height: 260 },
  minSize = { width: 220, height: 120 },
  children,
}: FloatingWindowProps) {
  const [position, setPosition] = useState(defaultPosition);
  const [size, setSize] = useState(defaultSize);
  const [zIndex, setZIndex] = useState(topZ);
  const rootRef = useRef<HTMLDivElement>(null);

  useEffect(() => {                    // 掛載：置頂並聚焦，ESC 立即可用
    setZIndex(++topZ);
    rootRef.current?.focus();
  }, []);

  const bringToFront = () => { if (zIndex !== topZ) setZIndex(++topZ); };

  const dragHandlers = usePointerDrag((dx, dy) =>
    setPosition((p) => keepReachable({ x: p.x + dx, y: p.y + dy }, size)));

  const resizeHandlers = usePointerDrag((dx, dy) =>
    setSize((s) => ({
      width:  Math.max(minSize.width,  s.width  + dx),
      height: Math.max(minSize.height, s.height + dy),
    })));

  return createPortal(
    <div
      ref={rootRef}
      className="floating-window"
      role="dialog"
      aria-label={title}
      tabIndex={-1}
      onPointerDown={bringToFront}
      onKeyDown={(e) => { if (e.key === 'Escape') onClose(); }}
      style={{
        position: 'fixed', left: 0, top: 0,
        transform: `translate(${position.x}px, ${position.y}px)`,
        width: size.width, height: size.height, zIndex,
        display: 'flex', flexDirection: 'column',
      }}
    >
      <header
        className="fw-titlebar"
```

```
        {...dragHandlers}
        style={{ touchAction: 'none', cursor: 'move', userSelect: 'none' }}
      >
        <span>{title}</span>
        <button
          aria-label="關閉"
          onClick={onClose}
          onPointerDown={(e) => e.stopPropagation()}  // 按關閉不是開始拖曳
        >×</button>
      </header>

      <main className="fw-content" style={{ flex: 1, overflow: 'auto' }}>
        {children}   {/* 拖曳時這棵子樹不會重算（8.5 節） */}
      </main>

      <div
        className="fw-resize"
        {...resizeHandlers}
        style={{
          position: 'absolute', right: 0, bottom: 0, width: 16, height: 16,
          touchAction: 'none', cursor: 'nwse-resize',
        }}
      />
    </div>,
    document.body,
  );
}
```

最後，把整章的設計決策與它們的依據對帳一次——這張表就是「讀懂一個互動元件」時該還原出來的推理：

程式碼裡的決策	它在回答什麼	依據
`createPortal` 到 `document.body`	裁切與疊層發生在 DOM；搬 DOM、不搬資料流	8.2 節
內容經 `children` 插槽傳入	API 對內容開放；拖曳時內容子樹不重算	第 5 章、8.5 節
`position` / `size` / `zIndex` 是 state；抓點與錨點是 ref	「值變了畫面要不要跟著變」逐一判定	第 4 章
增量回報＋函數式更新	手勢 hook 不讀呼叫端狀態：無依賴、無過期閉包	第 3、4 章
`topZ` 只在事件與 effect 中遞增	渲染保持純函數，StrictMode 雙跑安全	第 1 章
`if (zIndex !== topZ)` 守門	點擊已置頂視窗不多渲染一次	第 3 章
可選 props ＋解構預設值	最常見的用法寫起來最短；介面即文件	第 5、7 章

8.10　距離元件庫還有多遠

這個 `FloatingWindow` 已經能用在真實產品裡，但誠實列出它與元件庫成品的距離——每一項都能沿著本章的骨架長出來，差別只是需求，不是新原理：

- **受控的位置與尺寸**：補上 `position`＋`onPositionChange` 與 `defaultPosition` 並存的雙介面（第 5 章慣例）。做了它，「記住上次位置」就只是父層把位置存進 `localStorage` 的事。
- **視口變化的重新夾取**：使用者縮小瀏覽器視窗後，浮動視窗可能整個留在視口外。「位置」與「視口尺寸」的同步——監聽 `resize` 的 effect，第 4 章的模型再用一次。
- 開窗時的層疊偏移（cascade）、吸附邊緣、最大化與收合動畫、視窗管理員（8.6 節提過的群體行為）、SSR 掛載點（8.2 節）。

對照讀物：**react-rnd**（拖曳與縮放）與 **Floating UI**（錨定式浮動層：tooltip、下拉選單的定位引擎）。讀完本章再看它們的文件，你應該能說出每個選項為什麼存在——這正是這份講義想給你的能力。

本章重點回顧

- 浮動層的第一個決策是 `createPortal`：DOM 搬到 body 逃離裁切與 stacking context；狀態、context、事件冒泡仍走 React 樹。
- 拖曳三段式：按下記抓點（ref）、移動算新位置（state）、放開清空。事件能用

`setPointerCapture` 在元素上接完就不需要 effect；接不完（全域、跨元素）才用「狀態驅動監聽器」的 effect 同步。

- 高頻 setState 不可怕，子樹大才可怕：內容走 children 插槽，拖曳只重算視窗殼；量測到熱點才動用 ref 直寫 DOM 的逃生口。
- 疊層用模組計數器，但只在事件與 effect 中遞增，渲染保持純；置頂前先檢查是否已在頂端，省掉無效渲染。
- 拖曳與縮放是同一個手勢：抽成 `usePointerDrag`，增量回報＋函數式更新，hook 與呼叫端互不讀取彼此的狀態。
- `role="dialog"`、`tabIndex`、ESC、`touch-action: none` 是浮動視窗的基本禮儀；模態對話框是規格更嚴的另一個元件。

自我檢測

1. **視窗被 `createPortal` 掛到 `document.body` 之後，還讀得到原本位置上方的 context 嗎？視窗內的點擊還會冒泡給原本的父元件嗎？**
   參考答案：都會。portal 只改變 DOM 節點的落點，元件在 React 樹上的位置不變——context 沿 React 樹解析，事件也沿 React 樹冒泡。改變的只有瀏覽器的排版、裁切與疊層。

2. **拖曳時 `FloatingWindow` 每秒重新渲染上百次，為什麼經 children 傳入的內容不會跟著重算？什麼情況下會開始重算？**
   參考答案： children 是父層渲染時建立的元素物件；拖曳只觸發視窗自己的 setState，父層沒有渲染，children 引用不變，協調時遇到同一個元素物件就整棵沿用。若把內容直接寫死在 FloatingWindow 的 JSX 裡（每次渲染新建元素），或父層自己也在頻繁重新渲染，內容就會跟著重算。

3. **抓點偏移 `grabOffset` 為什麼放 `useRef` 而不是 `useState`？**
   參考答案：它是拖曳換算用的機制資料，畫面上沒有任何東西讀它——判準「值變了畫面要不要跟著變」答案是否，所以住 ref。放 state 只會讓每次按下標題列多一次無意義的渲染，而且在 move 的閉包裡讀 state 還得面對快照過期的問題。

4. **effect ＋ window 監聽版裡，`onMove` 若寫成 `setPosition({ x: position.x + e.movementX, ... })` 會發生什麼事？怎麼修？**
   參考答案： effect 只在 dragging 變 true 時建立一次，onMove 捕捉的是那次渲染的 position 快照；之後每次 move 都在同一個舊值上疊加，視窗動一步就卡住或跳回原地。修法：函數式更新 setPosition((p) => ...)，或改用「指標絕對座標減抓點偏移」的算法，根本不讀 position。

5. **增量式 `usePointerDrag` 在視窗被夾在視口邊界後，指標一反向移動視窗就立刻跟動。想要「指標回到抓住點才動」的手感，該怎麼改？**
   參考答案：改成絕對式：pointerdown 時記下指標錨點，onDelta 改為回報「自按下以來的總位移」（目前座標減錨點），呼叫端在按下時另記狀態初始值，每次以「初始值＋總位移」算出目標再夾取。位置不再從被夾過的值累加，與抓住點的關係就保住了。

第 9 章

綜合實戰：把觀念組裝成一個完整功能

本章目標——讀完後你應該能回答

- 一個「使用者管理頁」從型別到頁面的完整分層長什麼樣？
- 每一層的每個設計決策，分別對應前面各章的哪個原則？
- 拿到一段陌生的公司程式碼時，該按什麼順序提問？

第 1 到 7 章的觀念是零件，第 8 章用它們打造了一個元件，這一章做最後的總裝。需求選一個幾乎每家公司都有的頁面：**使用者管理**——搜尋框（防抖）、結果表格、分頁、載入與錯誤狀態。程式碼會完整走一遍，但重點放在每段旁邊的「為什麼」。

9.1 分層與檔案結構

```
features/users/
├── types.ts        // 領域型別：單一事實來源
├── api.ts          // 請求層：泛型 fetcher + 端點函式
├── useUsers.ts     // 邏輯層：狀態、防抖、抓取、競態防護
├── UserTable.tsx   // 呈現層：純顯示元件
└── UserPage.tsx    // 組裝層：接線
```

這個結構本身就是第 5 章「邏輯與畫面分離」的落地：**依賴方向永遠往下**（頁面依賴 hook，hook 依賴 api，api 依賴 types），每一層都能獨立閱讀與測試。公司專案的資料夾名稱可能不同，但分層幾乎一定認得出來。

9.2 types.ts：先把「什麼是合法的」說清楚

```ts
export interface User {
  id: string;
  name: string;
  email: string;
  role: 'admin' | 'member';        // 字面量聯集：非法角色無法表達
}

export interface Paginated<T> {    // 泛型：分頁結構與內容型別解耦
  items: T[];
```

```
  total: number;
  page: number;
  pageSize: number;
}

export type FetchState<T> =        // 標記聯集：狀態機，見第 5、6 章
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success'; data: T }
  | { status: 'error'; error: string };
```

為什麼先寫這個檔：型別是這個功能的「合約」。`Paginated<T>` 讓分頁形狀全公司只定義一次；`FetchState<T>` 讓「載入中卻有資料」這類非法狀態從根本上無法出現。之後每一層都 import 這一份，後端改欄位時，紅線會自動標出所有需要跟著改的地方。

9.3　api.ts：請求邏輯寫一次

```
import type { Paginated, User } from './types';

async function api<T>(path: string): Promise<T> {
  const res = await fetch(`/api${path}`);
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  return res.json() as Promise<T>;   // 邊界宣告：格式關鍵時改用 zod 驗證（7.8 節）
}

export function searchUsers(keyword: string, page: number) {
  const q = new URLSearchParams({ q: keyword, page: String(page) });
  return api<Paginated<User>>(`/users?${q}`);
}
```

為什麼：錯誤處理、base URL、JSON 解析這些「請求的固定成本」集中一處；端點函式退化成一行，型別 `Paginated<User>` 從這裡出發，流過 hook、流進元件，全程不需要再標註——泛型保留關係的實際效益。

9.4　useUsers.ts：這一頁「怎麼運作」的全部

```
export function useUsers() {
  const [keyword, setKeyword] = useState('');
  const [page, setPage] = useState(1);
  const debouncedKeyword = useDebounce(keyword, 300);   // 第 4 章的自訂 hook
  const [state, setState] = useState<FetchState<Paginated<User>>>({
    status: 'idle',
  });
```

```
  // 改關鍵字時回到第一頁：這是「事件的後果」，寫在事件裡，不用 effect 連鎖
  function changeKeyword(next: string) {
    setKeyword(next);
    setPage(1);          // 兩次 setState 自動批次，只渲染一次（第 3 章）
  }

  // 同步：讓「畫面上的資料」跟上「防抖後的查詢條件」
  useEffect(() => {
    let ignore = false;                     // 競態防護（第 4 章）
    setState({ status: 'loading' });
    searchUsers(debouncedKeyword, page)
      .then((data) => { if (!ignore) setState({ status: 'success', data }); })
      .catch((e)   => { if (!ignore) setState({ status: 'error', error: String(e)
    }); });
    return () => { ignore = true; };        // 條件又變了：宣告上一輪過期
  }, [debouncedKeyword, page]);             // 誠實申報：同步內容取決於這兩個值

  return { keyword, page, state, changeKeyword, changePage: setPage };
}
```

這個 hook 濃縮了最多決策，逐條對帳：

- **狀態的選擇**（第 2 章）：`keyword` 與 `page` 是真正的事實；`debouncedKeyword` 由 hook 推導；結果資料是伺服器狀態的快取，裝在 `FetchState` 裡。沒有一個多餘的 state。（實務中結果快取常直接交給 React Query；這裡手寫是為了讓你看見它替你做了什麼。）
- **「換關鍵字要回第一頁」寫在事件處理器**（第 4 章）：若寫成「effect 監聽 keyword 去 setPage」，因果就斷了，而且會多一輪「舊頁碼＋新關鍵字」的錯誤請求。
- **effect 是純粹的同步**（第 4 章）：依賴誠實、cleanup 防競態，快速輸入時慢回應的舊請求不會蓋掉新結果；StrictMode 雙跑也安然無恙。

9.5　呈現層：UserTable 與搜尋框

```
interface UserTableProps {
  users: User[];
  onSelect: (userId: string) => void;   // 對外只說「選了誰」（第 5 章）
}

// memo：表格大、且 props 在翻頁打字之外不常變 —— 值得設檢查哨（第 3 章）
export const UserTable = React.memo(function UserTable(
  { users, onSelect }: UserTableProps,
) {
  return (
    <table>
```

```
      <tbody>
        {users.map((u) => (
          <tr key={u.id} onClick={() => onSelect(u.id)}>  {/* id 當 key，不用 index
    */}
            <td>{u.name}</td><td>{u.email}</td><td>{u.role}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
});
```

9.6　UserPage.tsx：接線，然後追蹤一次資料流

```
export function UserPage() {
  const { keyword, page, state, changeKeyword, changePage } = useUsers();
  const navigate = useNavigate();
  const handleSelect = useCallback(
    (id: string) => navigate(`/users/${id}`),
    [navigate],
  );   // 穩定引用，讓 UserTable 的 memo 真正生效（第 3 章）

  return (
    <>
      <SearchInput value={keyword} onChange={changeKeyword} />
      {state.status === 'loading' && <Spinner />}
      {state.status === 'error' && <Alert message={state.error} />}
      {state.status === 'success' && (
        <>
          <UserTable users={state.data.items} onSelect={handleSelect} />
          <Pagination
            page={page}
            total={state.data.total}
            onChange={changePage}
          />
        </>
      )}
    </>
  );
}
```

注意 JSX 裡的 `state.status === 'success'` 分支：`state.data` 只在這個分支裡拿得到（標記聯集的收窄），「忘記處理載入中就取資料」在這個設計下**寫不出來**。

最後，把第 3 章的渲染分析拿來走一遍「使用者打了一個字」：

時刻	發生的事
按鍵當下	`changeKeyword`：`setKeyword`＋`setPage` 批次成一次渲染；`SearchInput` 顯示新字。`UserTable` 的 props（`users`、`onSelect`）引用皆未變，memo 淺比較通過，**大表格不重算**。
300ms 後	`useDebounce` 內的計時器到期，`debouncedKeyword` 更新→ effect 的依賴變了→ cleanup 將上一輪標記為過期→發新請求，`state` 進入 `loading`（再一次渲染，顯示 Spinner）。
回應到達	`state` 進入 `success` →渲染→ `UserTable` 收到新的 `users` 引用，memo 比較不通過，表格以新資料重算——這次的重算正是我們要的。

能把這張表自己推出來，就代表前面各章的模型已經連成一體了。

9.7　閱讀公司程式碼的提問清單

拿到陌生程式碼時，按順序問這十二個問題，多數設計意圖會自己浮現：

看到 state 時（第 2 章）：

1. 這個狀態的**擁有者**是誰？為什麼放這一層——有誰在共用它？
2. 它能不能從其他 state / props **推導**出來？（能的話為什麼還存了一份？）
3. 它是客戶端狀態，還是**伺服器資料的快取**？

看到 useEffect 時（第 4 章）：

4. 它在跟哪個**外部系統**同步？（答不出來→可能是誤用，找衍生資料或事件邏輯的影子）
5. 依賴陣列**誠實**嗎？有沒有被刻意留空或刪減？
6. cleanup 在防什麼？（訂閱洩漏？競態？沒有 cleanup 合理嗎？）

看到 memo / useMemo / useCallback 時（第 3 章）：

7. 下游有誰在**消費**這個穩定性？（memo 元件？依賴陣列？都沒有→可能是貨物崇拜）
8. 被 memo 的元件，props **全部**穩定了嗎？

看到型別時（第 6、7 章）：

9. 這個型別讓哪些**錯誤用法寫不出來**？（標記聯集？never 欄位？字面量？）
10. 泛型參數把哪幾個位置**綁定**在一起？
11. `as` 斷言出現在哪裡？那個邊界有沒有執行期驗證？

12. 這份型別的**事實來源**在哪？（手寫的？`z.infer`？後端 schema 產生的？）

9.8 常見 code smell 對照表

日常 code review 高頻出現的模式，以及它們對應的章節與重構方向：

Smell	重構方向	章節
`useState`＋`useEffect` 同步一個算得出來的值	改為渲染時計算（必要時 `useMemo`）	2
一叢互相牽制的布林狀態	合併成標記聯集狀態機	5, 6
effect 監聽狀態變化來執行「使用者操作的後果」	邏輯搬回事件處理器	4
依賴陣列被留空 / 刪減並加了 eslint-disable	誠實申報依賴，用函數式更新或 ref 解衝突	4
會增刪重排的列表用 index 當 key	改用穩定 id	3
元件函式內部定義另一個元件	搬到模組頂層，用 props 傳參	3
無差別的 `useCallback` / `useMemo` 全覆蓋	先量測，確認下游有消費者再加	3
十幾顆布林 / 設定 props 的「萬能元件」	variant 聯集＋ children / 插槽組合	5
context 裝高頻變動值，整棵樹跟著抖	狀態下移，或拆 context、memo value	2, 3
`as User` 直接斷言 API 回應	邊界改用 zod 驗證，內部消滅斷言	6, 7

9.9 接下來的路

這份講義刻意停在「理解既有設計」的層次。往下走的順序建議：

1. **官方文件的兩篇必讀**：react.dev 的 *Thinking in React* 與 *You Might Not Need an Effect*——讀完本講義再看，會有完全不同的吸收率。TypeScript 端則是官方 *Handbook* 的 Generics 與 Narrowing 兩章。
2. **資料層**：把手寫的 `FetchState` 換成 React Query，對照它文件裡的 staleTime、invalidation 概念，你會清楚看到第 2 章「伺服器狀態」那張表的完整版。
3. **測試**：React Testing Library。會發現「邏輯抽進 hook、介面誠實、受控元件」的設計讓測試變得順理成章——可測試性是好設計的免費副產品。
4. **往框架層看**：Next.js 與 React Server Components 改變的是「哪些元件在哪裡執行」，但本講義的資料流、型別、設計原則全部沿用。

最重要的練習只有一個：**回到公司的程式碼庫，用 9.7 節的十二個問題掃一個你常改的模組**，把答不出來的問題拿去問資深同事——這時你們已經說同一種語言了。

本章重點回顧

- 分層：types（合約）→ api（請求一次寫好）→ hook（運作邏輯）→元件（長相）→頁面（接線），依賴單向往下。
- 每個決策都有出處：批次更新、事件內處理連動、effect 同步＋競態防護、memo ＋穩定引用、標記聯集收窄。
- 讀碼用提問清單：狀態三問、effect 三問、memo 二問、型別四問。
- smell 對照表是 code review 的起手式；下一步是官方文件、資料層、測試。

自我檢測

1. **為什麼「換關鍵字要重設頁碼」寫在 `changeKeyword` 裡，而不是用 effect 監聽 keyword？**
   參考答案：它是使用者事件的直接後果，屬於事件處理器；用 effect 監聽會拆散因果、多一輪渲染，且在「舊頁碼＋新關鍵字」的中間狀態下發出一次錯誤的請求。

2. **在 UserPage 裡，為什麼 `handleSelect` 要包 `useCallback`？拿掉會怎樣？**
   參考答案： UserTable 被 React.memo 包住，它的 props 需要引用穩定；拿掉 useCallback 後每次渲染都是新函式，memo 淺比較永遠失敗，表格在每次打字時都會白白重算——memo 形同虛設。

3. **使用者在 300ms 內連打五個字，會發出幾次請求？為什麼？**
   參考答案：一次。useDebounce 的 effect 在每次值改變時 cleanup 掉上一個計時器，只有最後一次輸入的計時器活到期滿，debouncedKeyword 只更新一次，抓資料的 effect 也就只跑一次。

4. **如果後端把 `email` 欄位改名成 `emailAddress`，這個專案裡會發生什麼事？**
   參考答案：編譯期：所有讀取 `user.email` 的地方在更新 types.ts 後全部亮紅線，改完才能編譯。但若沒有執行期驗證，「後端已改、前端型別還沒改」的窗口期裡資料會默默變 undefined——這正是關鍵路徑需要 zod 這類邊界驗證的原因。

附錄：速查表、練習與資源

A 「該用哪個 hook」決策速查

你的需求	工具與依據
值改變時畫面要跟著變	`useState`（第 4 章）
值可以從其他 state / props 算出來	不要開 state，渲染時用 `const` 算（第 2 章）
計算昂貴，或需要穩定引用給 memo / 依賴	`useMemo` / `useCallback`（第 3 章）
值改變但畫面不需要反應（計時器 id、連線實例、最新值鏡像）	`useRef`（第 4 章）
需要與外部系統同步（訂閱、連線、DOM API、資料抓取）	`useEffect` + cleanup（第 4 章）
多個狀態總是一起變、轉移規則複雜	`useReducer` + 標記聯集 action（第 4、6 章）
整棵樹要讀同一份低頻資料	`createContext` + 自訂 hook 封裝（第 2 章）
兩個元件出現重複的 state + effect 組合	抽自訂 hook（第 4 章）

B 高頻型別速查

寫法	意思與典型用途
`React.ReactNode`	任何可渲染內容；children 與插槽 props 的預設選擇
`ChangeEvent<HTMLInputElement>`	輸入框 onChange 事件；`e.target.value` 由此而來
`ComponentPropsWithoutRef<'button'>`	原生元素的全部 props；包裝元件時做交集擴充
`useState<User | null>(null)`	初始值代表不了完整型別時手動標註
`Partial<T>` / `Pick<T, K>` / `Omit<T, K>`	從完整模型演算出 patch / 子集型別
`Record<string, T>`	鍵值查找表
`keyof typeof OBJ`、`(typeof ARR)[number]`	從 `as const` 的值推導字面量聯集（單一事實來源）
`x is User`（type guard）	自訂收窄函式；搭配 `unknown` 處理邊界資料
`never`	空集合：窮盡檢查哨兵、禁止欄位出現（`href?: never`）

C 動手練習：把公司程式碼當教材

以下練習全部以**你正在維護的專案**為對象，每題 30–60 分鐘，按章節順序做完，觀念就真正落地了：

1. **（第 1 章）**找一個元件，把它的渲染輸出手畫成「狀態→畫面」的對應表：列出所有 state 與 props，以及每種組合下畫面的差異。畫不出來的元件，通常正是難維護的元件。
2. **（第 2 章）**在專案裡找出一個「用 `useEffect` 同步衍生資料」的例子（搜尋 `setXxx` 出現在 effect 裡的地方），改成渲染時計算，確認行為不變。
3. **（第 2 章）**挑一個 context，回答：它的 value 多久變一次？有多少 consumer？變動時哪些元件會重新渲染？用 React DevTools 驗證你的答案。
4. **（第 3 章）**打開 Profiler，錄一次你們產品裡「打字」或「勾選」的互動，找出渲染範圍最大的一次更新，解釋每個被渲染的元件「為什麼在名單裡」。
5. **（第 3 章）**找一個被 `React.memo` 包住的元件，逐一檢查它的每個 props 的引用是否穩定，判斷這個 memo 是否真的在生效。
6. **（第 4 章）**找一個依賴陣列被 `eslint-disable` 壓掉警告的 effect，還原「當初為什麼要壓」，然後用函數式更新、`useCallback` 或搬移邏輯把 disable 拿掉。
7. **（第 5 章）**找出專案裡 props 最多的元件，把它的 props 分成「資料 / 行為 / 外觀設定」三

類，提出一個用 variant 與插槽收斂的重構草案（不用真的改，寫下來就好）。

8. **（第 6 章）**找一組「多顆布林互相牽制」的狀態，設計出對應的標記聯集，並在 switch 裡加上 `never` 窮盡檢查。
9. **（第 7 章）**追蹤一個 API 回應的型別旅程：從宣告處出發，經過幾層到達元件？中間有沒有 `as`？邊界有沒有驗證？畫出這條鏈。
10. **（第 8 章）**找出專案（或你們用的元件庫）裡的 Modal / Drawer / Tooltip：它用 portal 嗎？ESC 與外部點擊怎麼處理？開啟時焦點去哪、關閉時回到哪？對照第 8 章的零件清單，找出它多做或少做了什麼。
11. **（第 9 章）**用 9.7 節的十二個問題完整掃一個你常改的模組，把答案寫成半頁筆記——這份筆記就是你理解該模組設計的證明。

D 延伸資源

- **React 官方文件**（`https://react.dev/learn`）——本講義多次呼應的 *Thinking in React*、*You Might Not Need an Effect*、*Preserving and Resetting State* 是最值得精讀的三篇。
- **TypeScript Handbook**（`https://www.typescriptlang.org/docs/handbook/`）——搭配第 6 章讀 *Narrowing*、*Generics*、*Mapped Types*。
- **TypeScript Playground**（`https://www.typescriptlang.org/play`）——驗證型別行為的最快方式，hover 即見推斷結果。
- **React DevTools Profiler**——第 3 章所有效能主張的量測工具；「先量測再優化」的原則靠它落實。
- **TanStack Query 文件**（`https://tanstack.com/query`）——伺服器狀態管理的概念章節寫得極好，是第 2 章觀念的工業級延伸。
- **zod**（`https://zod.dev`）——邊界驗證與型別單一來源的實作。

—— 全書完。願你下次打開公司的程式碼時，看到的是一連串可以被理解的決策。 ——