React 快速入门（中文讲义版）

从 0 到 1：理解 React 核心概念 + 用 Vite 实作 Todo App

这份讲义会带你完成什么？

在这一份笔记里，你会学到 React 的基础概念，并且用 React 实作一个简单的 **Todo 待办清单**：

- 新增待办事项
- 完成/取消完成
- 编辑待办事项
- 删除待办事项

通过这个小项目，你会对 React 的运作方式更有整体感，而不是只会「跟着打代码」。

什么是 React？为什么要学？

React 是什么？（一句话）

React 是一个用来建构 **使用者介面（UI）** 的 JavaScript **函式库（library）**。

React：Library 还是 Framework？

严格来说 React 是 **library**，不是完整的 **framework**。但因为 React 生态非常庞大（路由、状态管理、表单、UI 组件库…都找得到），所以很多人会把它当作「前端框架」来称呼。

谁在用 React？

大量知名网站与产品都使用 React（例如社群平台、影音平台、电商网站等）。如果你想走前端开发，React 几乎是必备技能之一。

为什么不用纯 HTML/CSS/JS 就好？

技术上可以，但当网站变复杂时：

- 代码会变得难维护
- UI 会重复、逻辑分散、改一个地方牵一发动全身

React 的核心价值是：**把网页拆成组件（component）**，让复杂度大幅下降。

React 与 SPA（单页式应用）

传统网站：切换页面常常需要向服务器重新请求新的 HTML。
React 常见模式：**初次载入拿到一个 HTML**，之后由 React 在浏览器里动态更新内容，体感更流畅。

开始前准备：你需要什么？

建议先具备的基础

- JavaScript / HTML / CSS 基础概念
- 基本 npm 指令概念（安装套件、启动开发服务器）

不会也没关系：遇到不懂的地方，再查资料补起来即可。

开发环境必备

- 安装 **Node.js**
- 安装编辑器：建议 **VS Code**

建立 React 专案（推荐：Vite）

两种常见建专案方式

- Create React App（CRA）：经典但较旧
- **Vite**：较新、速度快、现在更主流

这份课程使用 Vite。

用 Vite 建立 React 专案

```
npm create vite@latest
# 或指定版本（避免未来版本不相容）
npm create vite@5.2.2
```

建立后启动专案的三行指令

- `cd <project-name>`：进入专案目录
- `npm install`：安装依赖
- `npm run dev`：启动开发服务器（例如 `localhost:5173`）

React 专案结构：哪些要看？哪些不用管？

重要文件与资料夹（先记这几个就够）

- `node_modules/`：套件库（不要改）
- `src/`：**你 99% 会写代码的地方**
- `public/`：静态资源（图、字体、txt…）
- `index.html`：SPA 的唯一 HTML（通常不用改）
- `package.json`：专案资讯 + scripts（例如 `dev`）

新手最容易误会的点

- `node_modules` 不要动
- 很多配置档（eslint、gitignore、vite config）先不用管
- 专注在 `src/` 就好

核心概念 1：Component（组件）

Component 是什么？

组件就像一个「容器」：把多个 UI 元素包在一起。
React 会把很多组件组合起来，形成最终网页（像树状结构）。

函数组件（Function Component）是主流

React 早期有 Class Component，但现在主流是 **Function Component**。
规则：**只要函数回传 JSX（像 HTML 的东西），它就是一个组件**。

最简单组件：回传 JSX

```
function MyComponent() {
  return <h1>你好</h1>;
}
```

命名规则（必背）

- 组件名称 **开头必须大写**（例如 `MyComponent`）
- 常用命名：**PascalCase**（每个单字首字大写）

不要把组件「定义」写在另一个组件里面

组件定义写在组件内部通常会让 React 变慢、甚至出现奇怪 bug。
组件可以被使用（render）在组件里面，但定义请写在外层。

Parent / Child（父组件 / 子组件）

- A 组件里面使用 B：A 是父组件，B 是子组件
- 组件最终会形成树状结构（类似 HTML DOM 结构）

核心概念 2：JSX（为什么 JS 里可以写 HTML ?）

JSX 是什么？

JSX = JavaScript + 类 HTML 语法。
浏览器不懂 JSX，但 **Vite 会编译 JSX → JavaScript**，所以能运行。

JSX 关键规则 1：组件只能回传「一个」根元素

下面这样不行（两个并列根元素）：

- `return <h1/> <p/>;`

解决：包一层容器，或使用 Fragment（空标签）。

Fragment：避免多余 div

```
return (
  <>
    <h1>标题</h1>
    <p>内容</p>
  </>
);
```

return 换行陷阱（JavaScript 语法）

`return` 如果单独一行，会被视为函数结束，下面 JSX 不会执行。
请写成 `return ( ... )` 或 `return <div/>;` 同一行。

在 JSX 里使用 JavaScript：{ } 大括号

{ } 的意义

在 JSX 里面用 { } 包起来，代表「这里要执行 JavaScript 表达式」。

显示变量

```
const text = "hello world";
return <h1>{text}</h1>;
```

在属性里用 JavaScript

```
<input placeholder={text} />
```

JSX 与 HTML 属性差异（常见两条）

- `class` → `className`（因为 class 是 JS 关键字）
- `for` → `htmlFor`

多单字属性用 **驼峰式**（camelCase）。

双大括号 {{ }}：不是新语法

{{ }} 常见在 style

外层 { }：表示 JSX 中执行 JS。
内层 { }：表示 JavaScript 物件。

Inline style（物件写法）

```
<h1 style={{ backgroundColor: "red" }}>Hello</h1>
```

注意：CSS 写法与 JS 物件写法不同

- CSS：`background-color`
- JSX style 物件：`backgroundColor`

核心概念 3：事件处理（onClick / onChange）

事件处理写法

在 JSX 标签上写事件属性（例如 `onClick`），值放入函数（不要马上执行）。

正确：传函数本体

```
<button onClick={handleClick}>点我</button>
```

错误：写成 handleClick() 会立刻执行

```
<button onClick={handleClick()}>点我</button>
```

事件物件（event）

事件处理函数会收到事件物件 e，可用 `e.target.value` 取得输入内容。

核心概念 4：阵列渲染（map / filter / key）

为什么需要 map？

`map` 可以把资料阵列转换成「UI 阵列」（一堆 JSX 元素）。

map：把资料变成 div 列表

```
items.map((item) => (
  <div key={item.id}>{item.content}</div>
));
```

filter：过滤资料

`filter` 回传一个新阵列，只保留回调函数回传 `true` 的资料。

filter：过滤掉某一项

```
const newItems = items.filter((item) => item.content !== "李四");
```

key 是什么？（必背）

当你渲染阵列时，每个元素都要有稳定的 `key`，React 才能正确追踪并提升效率。
不要用 `Math.random()` 当 key（每次 render 都会变，反而更慢）。

核心概念 5：条件渲染（if / 三元 / &&）

if/else：回传不同 UI

`if` 条件成立就回传 A，不成立就回传 B。

三元运算子（最常见）

```
condition ? A : B
```

在 JSX 中做条件渲染

```
return (
  <div>
    {isLoggedIn ? <h1>会员内容</h1> : <h1>访客内容</h1>}
  </div>
);
```

AND（&&）简写

当你只想在条件成立时显示某元素：`condition && <Component />`

&& 条件渲染

```
{isAdmin && <p>管理员专区</p>}
```

条件 className（切换 CSS）

你可以用三元 + 模板字串切换 class：

切换 className

```
<div className={`todo ${done ? "completed" : ""}`}>
  ...
</div>
```

核心概念 6：Props（父传子）

Props 是什么？

Props = 父组件传给子组件的资料（像函数参数）。
父组件用「属性」传入，子组件用参数接收。

传入 props

```
// Parent
<MyComponent a="hello" b="你好" />

// Child
function MyComponent(props) {
  return <p>{props.a} - {props.b}</p>;
}
```

常见写法：解构 props

```
function MyComponent({ a, b }) {
  return <p>{a} - {b}</p>;
}
```

重要限制（必背）

Props 只能由上往下传（父 → 子）。
子组件不能直接把 props “传回” 父组件（需要透过函数回调或状态提升）。

children：把组件当作内容传进去

children 是什么？

当你把某个组件写在另一个组件的「标签内容」里，React 会把它放进 `children` 这个 props。

children 示例

```
function Wrapper({ children }) {
  return <div className="box">{children}</div>;
}

<Wrapper>
  <h1>Hello</h1>
</Wrapper>
```

这叫 Composition（组合）

用 children/组件嵌套的方式，可以让组件更灵活地组合 UI（进阶概念，先有印象即可）。

核心概念：Props（小学生版，多层传递）

一句话给小学生听

Props 就像「纸条」。
爸爸（父组件）把纸条交给小孩（子组件），
小孩可以再交给孙子（更下面的组件）。

最重要的三件事

- Props 只能 **由上往下传**（父 → 子）
- 子组件 **不能直接改 props**
- 父组件可以把「资料」或「函数」当作 props 传下去

例子 1：最简单的 Props（传一个字串）

Example Code：爸爸给孩子一张纸条

```
// App.jsx
function ChildGreeting({ message }) {
  return <p>孩子看到纸条：{message}</p>;
}

export default function App() {
  return <ChildGreeting message="你好，我是爸爸给你的纸条！" />;
}
```

运行结果 画面会显示：
孩子看到纸条：你好，我是爸爸给你的纸条！

例子 2：多层 Props（爸爸 → 孩子 → 孙子）

小学生版说明

纸条可以一路往下传，
爸爸不知道孙子怎么显示，但纸条一定会到。

Example Code：多层传递

```
function GrandChild({ title }) {
  return <p>孙子最后显示：{title}</p>;
}

function Child({ title }) {
  return <GrandChild title={title} />;
}

export default function App() {
  return <Child title="这是一张一路传下来的纸条" />;
}
```

运行结果 画面会显示：
孙子最后显示：这是一张一路传下来的纸条

例子 3：Props 传「函数」（孩子叫爸爸做事）

小学生版说明

孩子不能命令爸爸，
但可以「按按钮通知爸爸」。

Example Code：传函数当 props

```
function Child({ name, onHello }) {
  return (
    <button onClick={() => onHello(name)}>
      {name} 跟爸爸说你好
    </button>
  );
}

export default function App() {
  function handleHello(childName) {
    alert(childName + " 跟你说你好！");
  }

  return (
    <>
      <Child name="小明" onHello={handleHello} />
      <Child name="小美" onHello={handleHello} />
    </>
  );
}
```

运行结果

- 点「小明」按钮 → 跳出：小明跟你说你好！
- 点「小美」按钮 → 跳出：小美跟你说你好！

例子 4：Props 传「物件」（资料打包传下去）

小学生版说明

如果资料很多，
可以装进一个盒子（物件）再传。

Example Code：传学生资料

```
function Student({ student }) {
  return (
    <p>
      名字：{student.name},
      年级：{student.grade}
    </p>
  );
}

export default function App() {
  const student = {
    name: "小明",
    grade: 3,
  };

  return <Student student={student} />;
}
```

运行结果 画面显示：
名字：小明，年级：3

例子 5：状态提升（爸爸管 State，孩子用 Props）

小学生版说明（超重要）

- 分数放在爸爸那里（state）
- 孩子只能「请求爸爸改」

这叫：**状态提升（Lifting State Up）**

Example Code：孩子加分

```
import { useState } from "react";

function Child({ score, setScore }) {
  return (
    <button onClick={() => setScore(score + 1)}>
      加 1 分
    </button>
  );
}

export default function App() {
  const [score, setScore] = useState(0);

  return (
    <>
      <p>分数：{score}</p>
      <Child score={score} setScore={setScore} />
    </>
  );
}
```

运行结果

- 画面显示目前分数
- 每按一次按钮，分数 +1

例子 6：children（把组件当内容塞进去）

小学生版说明

组件就像盒子，
你可以把任何东西放进盒子里。

Example Code：children

```
function Box({ children }) {
  return <div className="box">{children}</div>;
}

export default function App() {
  return (
    <Box>
      <h1>我是标题</h1>
      <p>我是内容</p>
    </Box>
  );
}
```

运行结果 标题和段落会被显示在 Box 组件里面。

Props 小学生必背口诀

- 爸爸给资料，孩子只能用
- 改资料一定要请爸爸
- 要让孩子影响爸爸：传函数

核心概念 7：State（状态）与 useState

为什么一般变量不行？

普通变量改变时，React **不会自动更新 UI**。
要让 UI 跟着资料变化，就要用 **State**。

useState 的回传值（阵列两格）

`useState(initialValue)` 会回传：

- state 目前的值
- 更新 state 的函数（惯例命名：`setXxx`）

计数器：state 改变会更新画面

```
import { useState } from "react";

function Counter() {
  const [clicks, setClicks] = useState(0);

  return (
    <button onClick={() => setClicks(clicks + 1)}>
      点击次数：{clicks}
    </button>
  );
}
```

阵列解构 vs 物件解构（本课最常用）

- 物件解构：**名字重要，顺序不重要**
- 阵列解构：**顺序重要，名字不重要**

所以 `useState` 的第一格永远是值，第二格永远是 setter。

不要直接改 state 变量

像 `clicks++` 不会让 React 正确更新 UI。
必须使用 setter（例如 `setClicks`），React 才知道要 re-render。

Render（渲染）小抄

- Initial rendering：第一次把 JSX 变成真实 DOM
- Re-rendering：state 改变后，React 重新计算 UI 并更新 DOM

State 放在哪里？（状态提升的观念）

State 应该放在 **需要使用它的组件的共同父层**，再用 props 往下传。
原因：props 只能父传子（上 → 下）。

Props drilling（概念先认识）

如果 state 放太高，会导致中间层组件拿到一堆用不到的 props，还要一层层往下传。
这是 props drilling。进阶解法：**Context**（本课先不展开）。

核心概念：State / useState / useEffect（小学生版）

一句话给小学生听

State 就像「会自动更新画面的记分板」。
你改记分板的数字，画面就会自动跟着变。
useState 是帮你做一个记分板的工具。
useEffect 是「发生某件事时自动执行的任务」(像闹钟/机器人)。

先记住 3 个超重要规则

- 普通变量变了，画面**不会**自动更新；State 变了，画面**会**自动更新
- 你**不能直接改** state 变量，必须用 setter：`setXxx(...)`
- `useEffect(函数, [依赖])`：依赖变化时，自动做事；`[]` 代表只做一次

Part A：为什么普通变量不行？(反例)

反例：普通变量变了，画面不会变

下面这段代码，按钮按了数字会加，但画面不会更新（只会 console 变）。

Example Code（反例）：普通变量不会触发更新

```
export default function App() {
  let score = 0;

  function addOne() {
    score = score + 1;
    console.log("score =", score);
  }

  return (
    <>
      <p>分数：{score}</p>
      <button onClick={addOne}>加 1 分</button>
    </>
  );
}
```

运行结果（你会看到什么）

- 你按按钮：console 会变（score=1,2,3...）

- 但画面一直显示：**分数：0**

Part B：useState 入门：计分板（最简单）

useState 是什么？

useState(初始值) 会给你两样东西（用阵列装起来）：

- 第一格：现在的值（例如 score）
- 第二格：改值的按钮（函数）（例如 setScore）

Example Code：计分器（按一下 +1）

```
import { useState } from "react";

export default function App() {
  const [score, setScore] = useState(0);

  return (
    <>
      <p>分数：{score}</p>
      <button onClick={() => setScore(score + 1)}>
        加 1 分
      </button>
    </>
  );
}
```

运行结果

- 一开始显示：分数 0
- 每按一次按钮：分数会变 1、2、3…（画面自动更新）

Part C：useState 例子：输入框（双向绑定）

小学生版解释：双向绑定

- 你在输入框打字 → state 跟着变
- state 变了 → 画面显示的文字跟着变

Example Code：你打什么，它就显示什么

```
import { useState } from "react";

export default function App() {
  const [name, setName] = useState("");

  return (
    <>
      <input
        placeholder="请输入名字"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <p>你好，{name}！</p>
    </>
  );
}
```

运行结果

- 输入框打「小明」→ 画面显示：你好，小明！
- 删除文字 → 画面也会跟着变空

Part D：useState 例子：布林开关（显示/隐藏）

小学生版解释：开关

`true` = 打开，`false` = 关掉。
我们用按钮来切换它。

Example Code：显示/隐藏秘密句子

```
import { useState } from "react";

export default function App() {
  const [show, setShow] = useState(false);

  return (
    <>
      <button onClick={() => setShow(!show)}>
        {show ? "隐藏" : "显示"}秘密
      </button>

      {show && <p>秘密：我今天吃了冰淇淋！</p>}
    </>
  );
}
```

运行结果

- 按按钮：秘密文字出现
- 再按一次：秘密文字消失

Part E：useState 例子：阵列（加入清单）

小学生版解释：把东西放进书包（阵列）

阵列就像书包：里面可以放很多东西。
注意：不能直接改原本书包，要做一个**新的书包**再放进去（用展开运算子 ...）。

Example Code：把水果加入清单

```
import { useState } from "react";

export default function App() {
  const [fruit, setFruit] = useState("");
  const [fruits, setFruits] = useState([]);

  function addFruit() {
    if (fruit.trim() === "") return;
    setFruits([...fruits, fruit]);
    setFruit("");
  }

  return (
    <>
      <input
        placeholder="输入水果"
        value={fruit}
        onChange={(e) => setFruit(e.target.value)}
      />
      <button onClick={addFruit}>加入</button>

      <ul>
        {fruits.map((f, idx) => (
          <li key={idx}>{f}</li>
        ))}
      </ul>
    </>
  );
}
```

运行结果

- 输入「苹果」按加入 → 清单出现苹果
- 再输入「香蕉」按加入 → 清单变成：苹果、香蕉

Part F：useState 进阶：物件（改一个属性）

小学生版解释：人物卡（物件）

物件像人物卡：有名字、有年龄。
要改年龄：要先复制旧卡，再换掉其中一栏。

Example Code：生日 +1 岁

```
import { useState } from "react";

export default function App() {
  const [student, setStudent] = useState({
    name: "小明",
    age: 7,
  });

  function birthday() {
    setStudent({ ...student, age: student.age + 1 });
  }

  return (
    <>
      <p>
        名字：{student.name}，年龄：{student.age}
      </p>
      <button onClick={birthday}>过生日（+1岁）</button>
    </>
  );
}
```

运行结果 每按一次按钮，年龄会变：7 → 8 → 9 …

useEffect（小学生版：自动任务/闹钟）

useEffect 是什么？

useEffect 就像「自动机器人」：
当页面刚打开或**当某个 state 改变**时，它会自动做事。

useEffect 两个参数

- 第一个：要做的事（函数）
- 第二个：什么时候做（依赖阵列）

例子 1：只做一次（页面打开时说嗨）

[] 的意思

依赖阵列是空的 []：代表**只在页面第一次打开时做一次**。

Example Code：页面加载时 alert

```
import { useEffect } from "react";

export default function App() {
  useEffect(() => {
    alert("页面打开了！你好！");
  }, []);

  return <p>看！我在页面上。</p>;
}
```

运行结果 你一打开网页就会跳出对话框：**页面打开了！你好！**

例子 2：跟着 state 变化（分数改变就记录）

依赖阵列写 [score]

代表：**每次 score 改变，就做一次事情**。

Example Code：分数改变就 console.log

```
import { useEffect, useState } from "react";

export default function App() {
  const [score, setScore] = useState(0);

  useEffect(() => {
    console.log("分数改变了！现在是：", score);
  }, [score]);

  return (
    <>
      <p>分数：{score}</p>
      <button onClick={() => setScore(score + 1)}>加 1 分</button>
    </>
  );
}
```

运行结果

- 画面分数会更新
- console 会一直印：分数改变了！现在是：1、2、3…

例子 3：定时器（每秒 +1）+ 清理 cleanup

小学生版解释：机器人每秒加分

我们用 `setInterval` 做一个「每秒做一次」的机器。
但离开页面时要把机器关掉（cleanup），不然会越来越乱。

Example Code：每秒自动加 1 分

```
import { useEffect, useState } from "react";

export default function App() {
  const [sec, setSec] = useState(0);

  useEffect(() => {
    const timerId = setInterval(() => {
      setSec((s) => s + 1);
    }, 1000);

    // cleanup：组件离开时，把计时器关掉
    return () => clearInterval(timerId);
  }, []);

  return <p>已经过了 {sec} 秒</p>;
}
```

运行结果

- 画面会自动跳：已经过了 0 秒、1 秒、2 秒…
- 关闭页面时，计时器会被正确关掉

例子 4：假装在「抓资料」（加载中 Loading）

小学生版解释：等一下再给你答案

有时候要等资料回来（像从服务器拿数据），
我们会先显示 Loading，拿到后再显示内容。

Example Code：用 setTimeout 假装 API

```
import { useEffect, useState } from "react";

export default function App() {
  const [loading, setLoading] = useState(true);
  const [data, setData] = useState("");

  useEffect(() => {
    // 假装在抓资料：2 秒后拿到结果
    const id = setTimeout(() => {
      setData("资料来了：今天的午餐是咖喱饭！");
      setLoading(false);
    }, 2000);

    return () => clearTimeout(id);
  }, []);

  return <p>{loading ? "Loading..." : data}</p>;
}
```

运行结果

- 前 2 秒显示：Loading...
- 2 秒后显示：资料来了：今天的午餐是咖喱饭！

小学生必背口诀（State + Effect）

- **State**：记分板，改了画面会更新
- **useState**：做记分板（给你值 + setter）
- **useEffect**：自动机器人（页面打开/某个值变了就去做事）
- **cleanup**：离开时关掉计时器/任务

实作项目：Todo App（Vite + React）

成品功能列表

- 输入 todo 并新增
- 点击切换完成/未完成（样式变化）
- 点击编辑进入编辑模式并更新内容
- 点击删除移除 todo

步骤 1：建立项目与清理模板

建立专案（示范版本号）

```
npm create vite@5.2.1
cd my-react-todo
npm install
npm run dev
```

清理模板（教学常用做法）

删除不需要的：

- `assets/`
- 默认 CSS
- 预设示例代码

先让页面只显示 `Hello World`，再从 0 写起。

步骤 2：建立组件结构（推荐做法）

为什么要拆组件？

把所有 UI 写进 App 会变巨大、难维护，也失去 React “组件化” 的意义。
拆组件：每个功能一块，最后组合起来。

建议组件规划（本项目）

- `TodoWrapper`：整体容器 + todo 状态管理（todos state）
- `CreateForm`：输入与新增 todo
- `Todo`：每一条 todo 的显示（内容、完成、编辑、删除按钮）
- `EditForm`：编辑模式的表单

步骤 3：样式（App.css）与基本布局

你会用到的 CSS 技巧

- 全域 reset（`* { margin:0; padding:0; box-sizing:border-box;}`）
- 置中布局（`display:flex; justify-content:center;`）
- 容器卡片样式（背景色、padding、圆角）

CSS 引入方式

```
import "./App.css";
```

为什么可以 import CSS？

纯 JavaScript 原本不能 import CSS。
这是 Vite 的能力：它会把 CSS 插入最终 HTML，因此才能这样写。

步骤 4：新增 Todo（表单 + state + two-way binding）

CreateForm 的关键：two-way binding

- `value={content}`：让 input 显示 state
- `onChange`：每次输入更新 state

这样 UI 和 state 会互相同步。

表单提交要 preventDefault

表单 submit 预设会刷新页面，所以要：

- `e.preventDefault();`

步骤 5：渲染 Todo 列表（map + key）

todos state 建议结构

每个 todo 用物件表示，而不是单纯字串：

- `id`：稳定的 key
- `content`：文字内容
- `isCompleted`：是否完成
- `isEditing`：是否编辑中

key 的正确来源

- 最佳：来自资料库的 id
- 教学/小项目：可临时用 `Math.random()` 产生，但要在建立 todo 时生成一次，不要在 render 里生成

步骤 6：删除 Todo（filter）

删除逻辑

用 `filter` 过滤掉指定 id 的 todo：

- 保留 `todo.id != id` 的资料

步骤 7：完成/取消完成（toggle + className）

完成状态切换（toggle）

点击 todo 时，把 `isCompleted` 从 false ↔ true 切换。
常见做法：`map` 回圈 + 找到目标 id 并回传新物件。

样式切换

- 完成：`opacity: 0.4`、`text-decoration: line-through`
- 用 `className` 条件加入 `completed`

步骤 8：编辑 Todo（isEditing + EditForm）

编辑模式的核心

- 每条 todo 有 `isEditing`
- `isEditing ? <EditForm/> : <TodoUI/>`
- 点编辑按钮切换 `isEditing`

EditForm 的关键

- 进入编辑时，input 初始值应为 `todo.content`
- 提交时呼叫 `editTodo(id, newContent)`
- 同时把 `isEditing` 改回 false（结束编辑）

本集总结（你应该带走什么）

React 基础必背清单

- React 用 **组件**组织 UI（组件树）
- JSX 让你在 JS 里写类 HTML，并可用 `{ }` 插入 JS
- 列表渲染用 `map`，过滤用 `filter`，列表元素要有稳定 `key`
- 条件渲染：`if`、三元、`&&`
- Props 父传子，不能子传父（需要函数回调/状态提升）
- State 用 `useState`，state 变了会触发 re-render 更新 UI

接下来可以学什么（进阶主题）

- Hook 生命周期与 `useEffect`
- Context（解决 props drilling）
- Routing（多页面/路由）
- 更完整的状态管理与资料持久化（LocalStorage / API / DB）

提示：你不需要一次把 React 全背完。把 Todo 项目做顺、做熟，你会自然理解组件、props、state、render 的意义。