React 前端框架和组件化

> 适用于想要入门 React 前端开发的同学，期望通过本文章快速上手开发，以及掌握一些 React 底层的工作原理。

React 的由来

在 React 诞生之前，传统的前端开发方式使用 **jQuery + 模板引擎** 命令式编程，操作 DOM 繁琐且容易出错，页面性能和可维护性都越来越差。

开发者面临的典型问题包括：

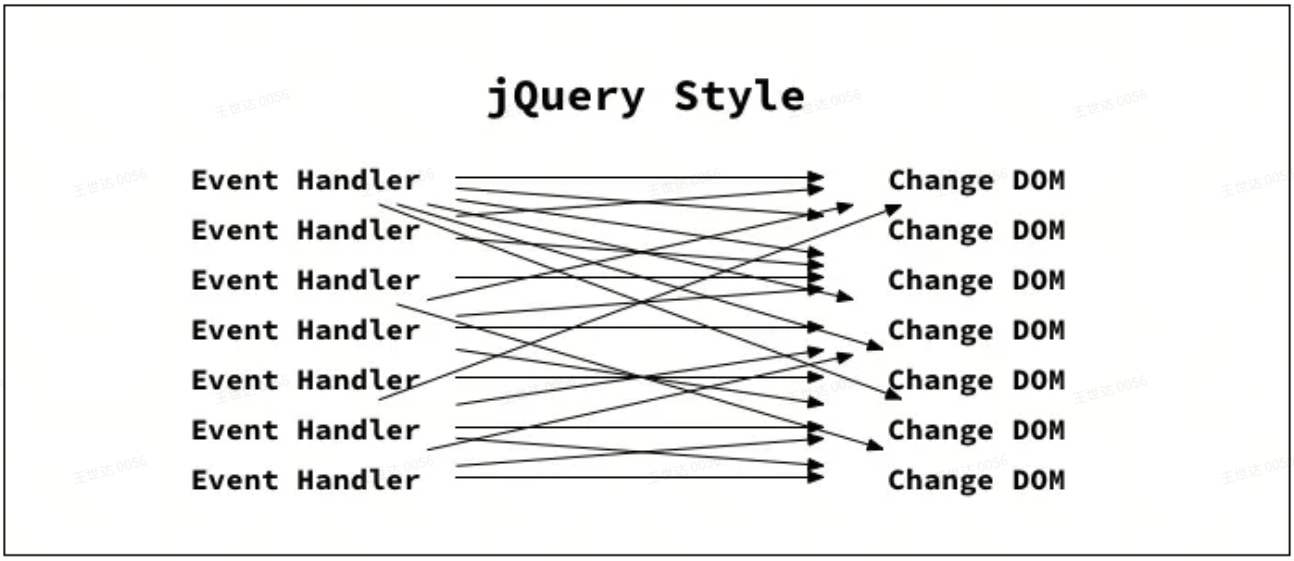
- **频繁的 DOM 更新** 导致性能瓶颈；
- **数据与视图不同步**（UI 状态管理困难）；
- **代码复用性差**。

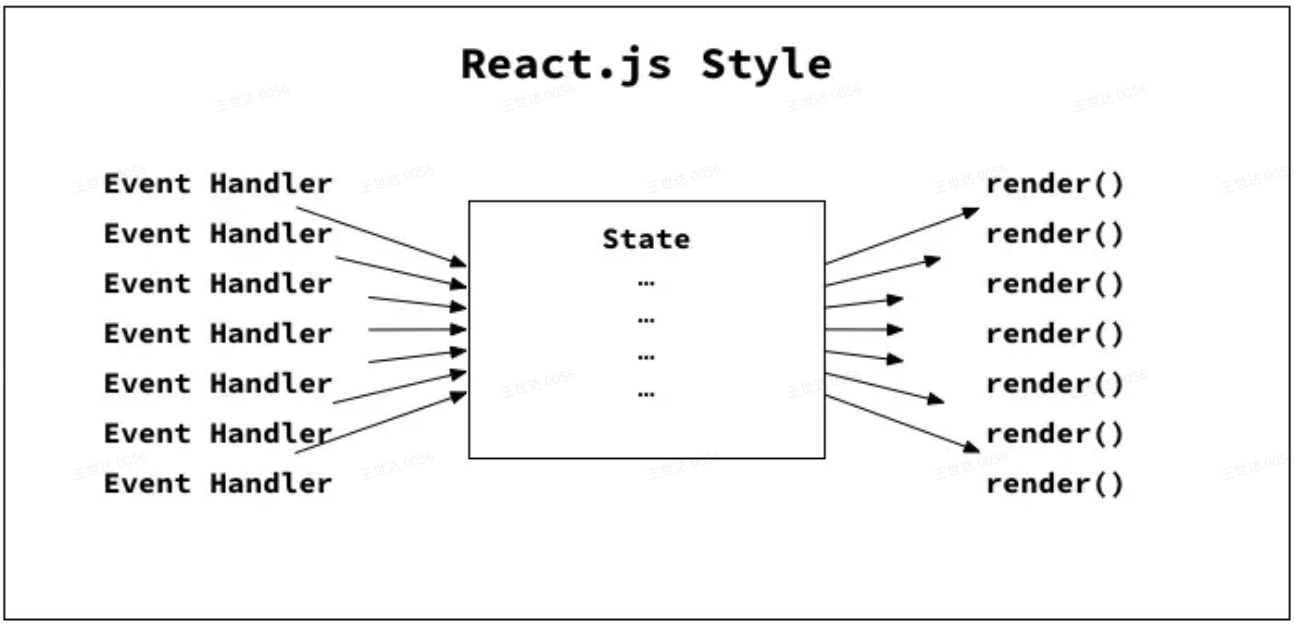
React 的诞生的初衷：

- 把 UI 拆分为可复用的组件；
- 让组件根据数据变化自动更新。

React 框架的 **声明式UI** + **组件化** + **虚拟DOM** + **单向数据流** 等特性，让开发者能够更直观的 UI 表达，极大降低复杂项目的开发和维护成本。

相比于传统前端开发，React 带来的不只是一个工具，而是一种前端工程化思维的转变。从 “一步步告诉计算机怎么做”，抽象到“**告诉 React 我想要什么结果**”，React 通过 Virtual DOM 和状态驱动，让 UI 成为状态的映射函数，从而实现了声明式、可预测的界面更新。





React 基础

> 📌 强烈建议沉浸式看完 React 官方教程，建议从 React 17/18 版本上手学习，下面会简单总结一些要点。

组件

在 React 中，组件本质上是一个**函数（或类）**，它根据输入的 props（属性） 和自身的 state（状态），**描述要渲染的 UI 结构**（React 元素树）。函数式可表达为 $UI = f(props, state)$。

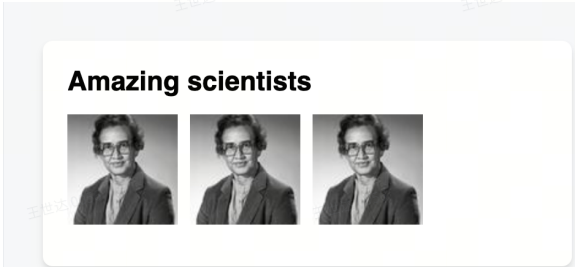
一个组件示例

代码块

```
import React from 'react';

function Profile({ style }) {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
      style={style}
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1 style={{ color: "gray" }}>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile style={{ width: 40, height: 40 }}/>
    </section>
  );
}
```



- **import / export** 分别为导入和导出语法，这样组件作为一个 ES6 module 可以在文件之间互相引用。
- 组件的输入：用于使用方定制属性。
- 组件的输出：使用 JSX 表达的声明式 UI。React 框架会将其转换为虚拟 DOM 节点，进一步转换为真实的 DOM 片段，插入到页面里。

两种不同的组件写法

函数组件（推荐）

以函数的方式声明组件

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

Class 组件

以 class 的形式声明组件，继承 React.Component，通过 render 方法返回 UI 视图

代码块

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

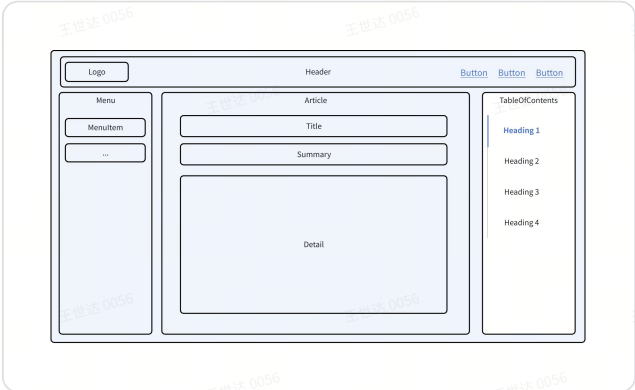
对比点	类组件	函数组件
写法	复杂（class、this）	简洁（纯函数）
状态管理	this.state / setState	useState
生命周期	componentDidMount 等	useEffect
逻辑复用	HOC / Render Props	自定义 Hook
性能	较低	更优（Hooks + Fiber）
未来趋势	被弱化	推荐标准

重要：推荐使用函数组件，结合 react hooks，可以做到更极致的代码复用。React 团队也表示未来会弱化类组件支持（不会立刻废弃，但已停止扩展）。

从组件到页面

通过不断复用和组合不同的组件，来形成最终的页面。

一个典型页面与其对应的抽象的 React 组件：



认识 JSX

在表达 UI 视图时，React 提供了 createElement 方法来创建一个视图元素 Element。但是 UI 视图本身会有比较复杂的嵌套和组合，直接使用 createElement 会导致代码可读性很差。

React 引入 JSX 来扩展 JS 与法，让 UI 视图的表达形式与原生 HTML 近似，极大提升代码可读性。

JSX 语法速览

语法	说明	示例
基本用法	JSX 是在 JavaScript 中编写类似 HTML 的语法。	代码块 `const element = <h1>Hello, world!</h1>;`
表达式插值	使用 {} 在 JSX 中嵌入任意 JS 表达式。	代码块 `const user = {name: "abc"};` `const element = <h1>{user.name}</h1>;`
属性传值（Props）	属性值为字符串时用引号，表达式用 {}。	代码块 `const element = <img src={user.avatar} alt="头像" />;`
条件渲染	使用三元表达式或逻辑与（&&）进行条件渲染。	代码块 `const element = (` `  <div>`

		代码块 4 {isLogin ? <User /> : <Login /> } 5 {count > 0 && {count} } 6 7)
列表渲染	使用 map() 遍历数组并返回元素，需加唯一 key。【参考】	代码块 1 const elements = (2 3 {items.map(item => 4 <li key={item.id}>{item.name} 5)} 6 7)
Fragment（片段）	用 <></> 包裹多个子元素而不引入额外 DOM。	代码块 1 const element = (2 <> 3 标题 4 内容 5 </> 6)

组件 Props & State

Props 和 State 都能够让组件呈现出不同的形态。但从设计意图上，两者存在较大差异。

理解组件 Props

Props 可以理解为组件的属性，组件向外暴露这些参数用于**定制**。

这个概念可以与 HTML 标签的属性对应上，组件可以视为自定义的 HTML 标签，组件的属性用法也类似于 HTML 标签的属性。

Props 示例：

代码块

```jsx
import Avatar from './Avatar.js';

function Card({ children }) {
  return (
    <div className="card">
      {children}
    </div>
  );
}

export default function Profile(props) {
  return (
    <Card>
      <Avatar size={100} {...props} />
    </Card>
  );
}
```

- 函数的第一个参数为函数组件的 Props，可以直接通过 props 获取，或者解构获取内部属性
- JSX 嵌套作为 children 传入
- 可以使用 spread 语法传入 Props
- 函数 Props 由父组件传入，不能在组件内部进行修改
- 默认情况下，当 Props 发生变化时，组件会重新渲染

理解组件 State

State 表示组件内部的状态，可以视为组件的 "记忆"。比如 Todo List 的列表数据，表单提交中的状态，输入框中的文本，表格的过滤条件等。

在一些用户交互后，内部状态会随着更新，更新状态会自动触发组件重新渲染。

代码块

```jsx
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>Clicked {count} times</button>
  );
}
```

- 在函数组件里使用 useState hook 来声明一个组件状态
- 在 class component 里使用 state 和 setState 来管理组件状态

Props vs State

- Props 代表这个组件对外暴露的可定制接口。在设计组件时，应当提前考虑组件如何被调用，从而来设计组件的 Props。
- State 是组件内部的记忆。保存用户在与组件交互过程中，产生的一些中间状态。

State 管理

初始化状态

代码块

```
const [state, setState] = useState(initialState)
```

更新状态

使用 useState 返回结果里的第二个参数 setState 来更新状态。

代码块

```
setState(newState); // 直接更新 state
setState(state => newState); // 根据之前的 state 更新 state
```

上述两种模式的差异：

- **直接更新 state**

代码块

```
function handleClick() {
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
}
```

- **根据之前的 state 更新 state**

代码块

```
function handleClick() {
  setAge(a => a + 1); // setAge(42 => 43)
  setAge(a => a + 1); // setAge(43 => 44)
  setAge(a => a + 1); // setAge(44 => 45)
}
```

📌 **React 状态更新队列**

- **legacy 模式**

在 React 中，状态更新（setState 或 useState 的 setter）不是立即生效的，它会被放入一个**更新队列（Update Queue）**中，等到合适的时机（如事件结束或下一次渲染周期）再统一执行。

如果每次调用 setState() 都立即更新组件和重新渲染，那在一次点击中多次调用 setState() 会导致 重复渲染，性能非常差。

React 会将这些更新**暂存（enqueue）**，然后在合适的时机 **批量处理（flush）**，只渲染一次组件。

- **concurrent 模式**

setState 不再是立即同步更新，而是触发一个带有「优先级」的更新任务，React 的调度器会根据优先级决定 何时、是否中断、是否恢复 该更新。

在哪里更新状态

事件处理函数	代码块 1 `function Counter() {` 2 `  const [count, setCount] = useState(0);` 3 `  const handleClick = () => setCount(count + 1);` 4 `  return (` 5 `  <button onClick={handleClick}>Clicked {count} times</button>` 6 `  );` 7 `}`
【不推荐，参考】useEffect 里	代码块 1 `function List({ items }) {` 2 `  const [isReverse, setIsReverse] = useState(false);` 3 `  const [selection, setSelection] = useState(null);` 4 5 `  useEffect(() => {` 6 `  setSelection(null);` 7 `  }, [items]);` 8 `  // ...` 9 `}`

【不推荐】函数组件渲染期间直接调整 state

> 相比 useEffect 较优，但仍不推荐。
> 示例里应当移除掉 selection 状态，直接在渲染期间计算。

代码块

```
function List({ items }) {
  const [isReverse, setIsReverse] = useState(false);
  const [selection, setSelection] = useState(null);

  const [prevItems, setPrevItems] = useState(items);
  if (items !== prevItems) {
    setPrevItems(items);
    setSelection(null);
  }
}
```

更新复杂对象状态

- 直接更新

正确更新复杂对象状态，**原则是需要更新对象地址（构建新的对象）**。

代码块

```
const [person, setPerson] = useState({
  name: 'Niki de Saint Phalle',
  artwork: {
    title: 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgUOm.jpg',
  }
});

// ❌ 无法生效
person.artwork.city = 'New Delhi';
setPerson(person);

// ✅ 可以生效
setPerson({
  ...person,
  artwork: {
    ...person.artwork,
    city: 'New Delhi'
  }
});
```

- 使用 Immer 更新

本质上也是依赖 immutable 库，构建出新的对象

代码块

```
import { useImmer } from 'use-immer';
const [person, setPerson] = useImmer({
  name: 'Niki de Saint Phalle',
  artwork: {
    title: 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgUOm.jpg',
  }
});

// ✅ 可以生效
setPerson(draft => {
  draft.artwork.city = 'Lagos';
});
```

外部如何更新状态

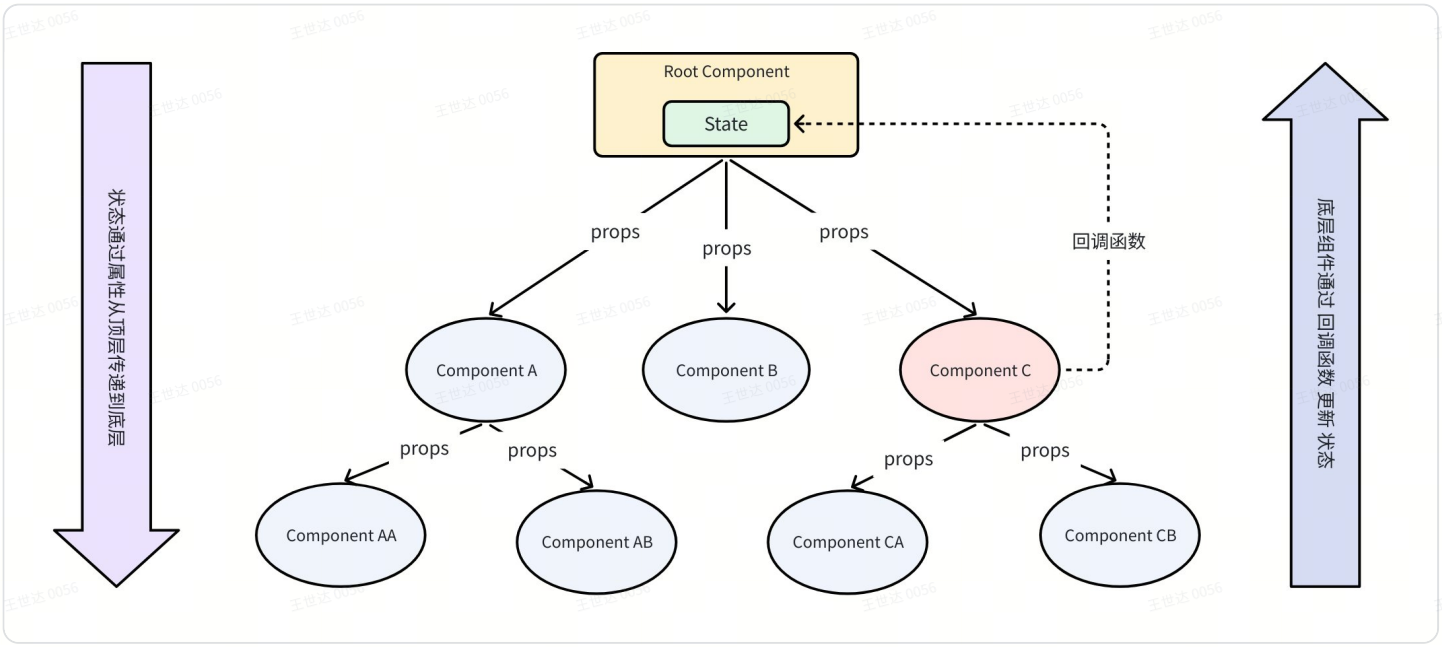
原则上讲，不能在组件外部直接更新组件内的状态。如果想要在外部更新组件内的一个状态，可以有以下方式：

- ✅ **提升子组件的状态到共同的父组件中，将状态作为属性传递给子组件**
- ✅ **使用全局状态 Context / Redux / Zustand 等**
- 通过 ref 暴露内部更新方法（forwardRef + useImperativeHandle）【合法但最佳实践不推荐】
- 全局发布订阅机制
- ❌ 直接暴露 setState 【强烈不推荐】

单向数据流

在 React 中，数据只能由上而下（从父组件到子组件）流动，组件不能直接修改父组件或兄弟组件的状态。

- 父组件通过 **props** 把数据传给子组件；
- 子组件通过 **回调函数** 通知父组件改变；
- React 的 UI 渲染永远是由状态（state）→ UI 的单向映射。
- 组件的状态管理变得可预测、可复现、可调试。



> 与单向数据流相对应的，即双向数据流。双向数据流在一些其他前端框架里有使用到，比如 Angular/ Vue。
>
> - 单向数据流：数据只能从上层往下层流动，状态更可控，适用于大型复杂项目
> - 双向数据流：视图与数据双向绑定，用起来会比较方便，但在大型项目里维护难度较大。

Hooks

Hooks 是一组可以让你在函数组件中使用状态（state）和生命周期（life cycle）等 React 特性的函数。

在 Hooks 出现之前：

- 逻辑复用困难（需要用 HOC 或 render props）；
- 组件状态逻辑分散（生命周期函数里混杂逻辑）；
- 类组件 this 指向问题频繁；
- 代码臃肿、难测试。

React 团队提出 Hooks 的目的：

✅ 更好的逻辑复用方式（通过自定义 Hook）；

✅ 让函数组件更强大（能使用 state、effect 等）；

✅ 更纯粹的组件模型（UI = f(state)）。

常用 Hooks

useState	管理组件内部状态	代码块 1 function Counter() { 2 const [count, setCount] = useState(0); 3 return <button onClick={() => setCount(count + 1)}>点击 {count}</button>; 4 }
useEffect	副作用（生命周期）处理	代码块 1 useEffect(() => { 2 // deps 发生变化时执行 3 }, [deps]) 4 5 useEffect(() => { 6 console.log('组件挂载'); 7 return () => console.log('组件卸载'); 8 }, []); // 依赖数组为空，只在挂载/卸载时执行
useMemo	缓存计算结果	代码块 1 const value = useMemo(() => expensiveCalc(), [deps]) 2 // 根据 deps 变化自动计算出新的值
useCallback	缓存函数引用 避免函数重新创建，触发组件属性变更，导致组件重新渲染吗	代码块 1 const handleClick = useCallback(() => { 2 // ... 3 }, [deps]) 4 5 return <Button onClick={handleClick}>+</Button>
useContext	使用上下文（全局数据）	代码块

		`` const ThemeContext = createContext('light'); function App() { return (<ThemeContext.Provider value="dark"> <Child /> </ThemeContext.Provider>); } function Child() { const theme = useContext(ThemeContext); return 当前主题: {theme} ; } ```
useRef	保存可变引用或访问 DOM	代码块 ``` const listRef = useRef<HTMLDivElement>(null); useEffect(() => { listRef.current?.scrollTo({ top: listRef.current.scrollHeight, behavior: "smooth", }); }, [listRef.current?.scrollHeight]); ``` 代码块 ``` const wsRef = useRef(null); useEffect(() => { wsRef.current = new WebSocket("xxx"); // ... return () => { wsRef.current && wsRef.current.close(); } }, []); ```

Hooks 使用规则

> React 用“Hook 调用顺序”作为内部状态匹配机制，如果***多次渲染之间顺序变化***会导致状态错乱（React 会直接报错）。

使用时需遵循一下规则：

- 只能在函数组件或自定义 Hook 中使用：不能在普通函数或 if/for 语句中用
- 每次渲染 Hook 调用顺序必须一致：React 依靠调用顺序来关联状态
- 自定义 Hook 必须以 use 开头：如 useFetch, useUser 等

自定义 Hook

赋予 React 代码逻辑复用的能力。把复杂逻辑提炼为 Hook，让 UI 组件更纯粹。

代码块

```
function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    const onResize = () => setWidth(window.innerWidth);

    window.addEventListener('resize', onResize);
    return () => window.removeEventListener('resize', onResize);
  }, []);
  return width;
}

// 使用
function App() {
  const width = useWindowWidth();
  return <p>窗口宽度：{width}</p>;
}
```

代码块

```
function useDocumentVisibility(): VisibilityState {
  const [documentVisibility, setDocumentVisibility] = useState(() => getVisibility());

  useEventListener(
    'visibilitychange',
    () => {
      setDocumentVisibility(getVisibility());
    },
    {
      target: () => document,
    },
  );

  return documentVisibility;
}
```

Hooks 原理浅析

React 会在渲染时：

- 为每个组件维护一个 Hook 链表；

- 按照 Hook 调用的顺序保存或读取相应的状态；
- 在下一次渲染时，根据这个顺序取出上一次的状态进行更新。

Hook 链表及其类型定义：

代码块

```
const [name, setName] = useState('John');
const [age, setAge] = useState(18);
```

代码块

```
{
  memoizedState: 'John',
  baseState: 'John',
  baseQueue: null,
  queue: null,
  next: {
    memoizedState: 18,
    baseState: 18,
    baseQueue: null,
    queue: null,
  },
};
```

代码块

```
// react-reconciler/src/ReactFiberHooks.old.js
export type Hook = {|
  memoizedState: any,
  baseState: any,
  baseQueue: Update<any, any> | null,
  queue: any,
  next: Hook | null,
|};

export type Effect = {|
  tag: HookFlags,
  create: () => (() => void) | void,
  destroy: (() => void) | void,
  deps: Array<mixed> | null,
  next: Effect,
|};
```

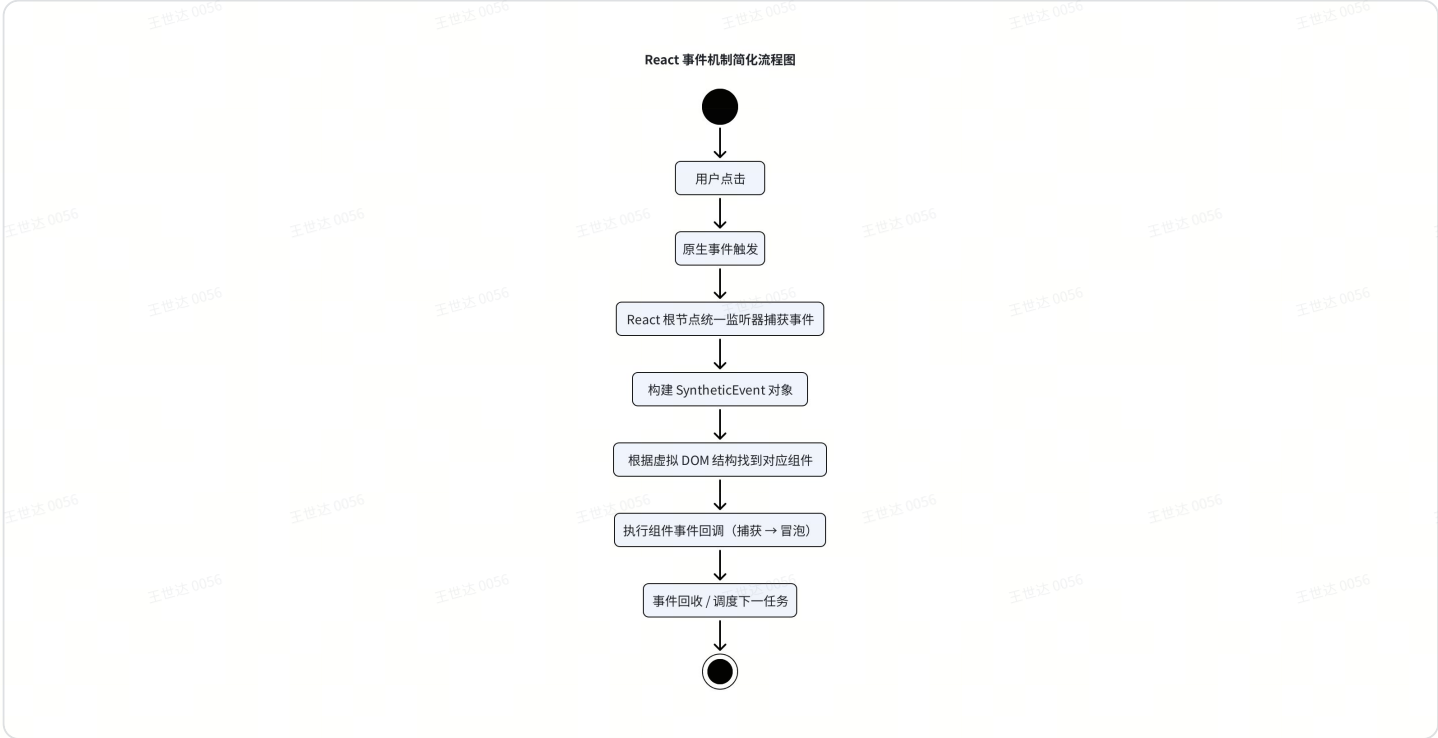
React 渲染函数组件：

- 在 renderWithHooks 里执行组件函数
- 组件首次渲染时，调用每个 Hook（如 useState/useEffect），创建初始的 Hook 数据；
- 调用 mountWorkInProgressHook 将函数组件的 Hook 转换成链表（如上所示）；
- 组件更新时，会按 Hook 调用顺序，从存储在 Fiber 的 state 里获取 Hook 节点，更新对应的值。

这也是为什么组件多次渲染期间 **Hook 的调用顺序必须保持一致**。

事件机制

React 并没有使用原生的浏览器事件机制，而是封装了一套合成事件（**SyntheticEvent**）机制。



即我们在 React 代码里绑定的 onClick，onChange 等事件回调，并不会直接绑定到目标元素上，而是绑定到 React 根节点上。

当事件触发时找到触发元素，依次向上冒泡事件，执行回调函数。

这么做的好处是：

- 屏蔽浏览器差异
- 事件委托机制，减少内存消耗 (**根结点统一监听事件**)

虚拟 DOM

React 的 **虚拟 DOM（Virtual DOM）** 是 React 性能优化的核心机制之一。它的主要作用是提高 UI 更新效率，使页面在频繁更新时依然保持高性能。

虚拟 DOM 是一种 **以 JavaScript 对象形式** 表示真实 DOM 结构的轻量级副本。

虚拟 DOM	• 保存在内存中； • 可以快速比较前后版本的变化； • 最终只把“必要的最小改动”同步到真实 DOM。	代码块 `const vdom = {` `  type: 'div',` `  props: {` `  id: 'app',` `  children: {` `  type: 'h1',`

		`` 7 props: { 8 children: 'Hello React' 9 } 10 } 11 } 12 }; ```
直接操作真实 DOM	• 每次修改都会引发**重绘（repaint）和回流（reflow）**； • 浏览器需要频繁更新渲染树，性能消耗大。	代码块 ``` 1 <div id="app"> 2 <h1>Hello React</h1> 3 </div> ```

React 的更新流程一般是这样的：

1. **初次渲染**

- React 组件生成虚拟 DOM。
- React 将虚拟 DOM 转换为真实 DOM，并插入页面。

2. **状态更新**

当组件的 state 或 props 变化时，React 生成一个新的虚拟 DOM（新的快照）。

3. **Diff 算法**

对比新旧虚拟 DOM，找出变化的部分（比如哪个节点、属性或文本变了）。

4. **批量更新真实 DOM**

只更新需要变化的部分，而不是重新渲染整个页面。

> 📌 **虚拟 DOM Diff 算法**
>
> 比较两棵树的所有节点（完全递归对比）是 $O(n^3)$ 的复杂度。 React 通过一系列假设和优化，将原本 复杂度降到了 **O(n)**。
>
> - 同层比较：只比较相同层级的节点；
> - 相同组件只更新 props，不同类型节点直接替换；
> - 列表 Diff: 根据 key 来判断哪些子节点移动、修改或删除。

状态管理进阶

复杂应用下的问题

随着应用的体量变大，简单的使用 useState 来管理状态会出现以下问题：

状态分散、难以维护	每个组件都维护自己的局部状态，数据需要一层层传递，一旦结构复杂或数据依赖改变，就很容易失控。 如果只用 state： • 每个组件都维护自己的局部状态； • 数据需要在父子组件之间一层层传递（props drilling）； • 当状态被多个页面或模块依赖时，数据同步变得困难。
状态同步困难、逻辑重复	不同模块可能依赖同一份状态（例如：用户信息、权限、购物车等）。 仅用 state 时： • 每个模块都需要单独拉取数据或维护逻辑； • 状态变更需要多处同步更新； • 逻辑重复、容易出现数据不一致。
性能问题（频繁重新渲染）	当顶层组件管理过多状态时： • 状态变更会触发整个子树重新渲染； • 即使某个子组件不依赖该状态，也会被迫更新。

可以使用 React Context 或者全局状态管理库来解决这些问题。

React Context

React 内部的跨组件共享状态机制，能够跨组件共享状态

用法实例

代码块

```
// 状态提供方
const ThemeContext = createContext();

function App() {
  const [theme, setTheme] = useState("light");
```

代码块

```
// 状态消费者
function DetailPage() {
  const { theme } = useContext(ThemeContext);
  return <div className={theme}>Click</div>;
}
```

```
    return (
      <ThemeContext.Provider value={{ theme, setTheme }}>
        <Layout />
      </ThemeContext.Provider>
    );
  }
```

```

function ThemeToggle() {
  const { theme, setTheme } = useContext(ThemeContext);
  const toggleTheme = () => {
    setTheme(t => t === 'light' ? 'dark' : 'light')
  }
  return <Button onClick={toggleTheme}>theme</Button>;
}
```

局限性

- **Context 更新会导致所有消费者重渲染**

React Context 是“广播式”的：当 Provider 的 value 变化时，所有 useContext() 的组件都会重新渲染 —— 即使它们只用到 value 的一部分。

- **缺乏状态逻辑管理能力**

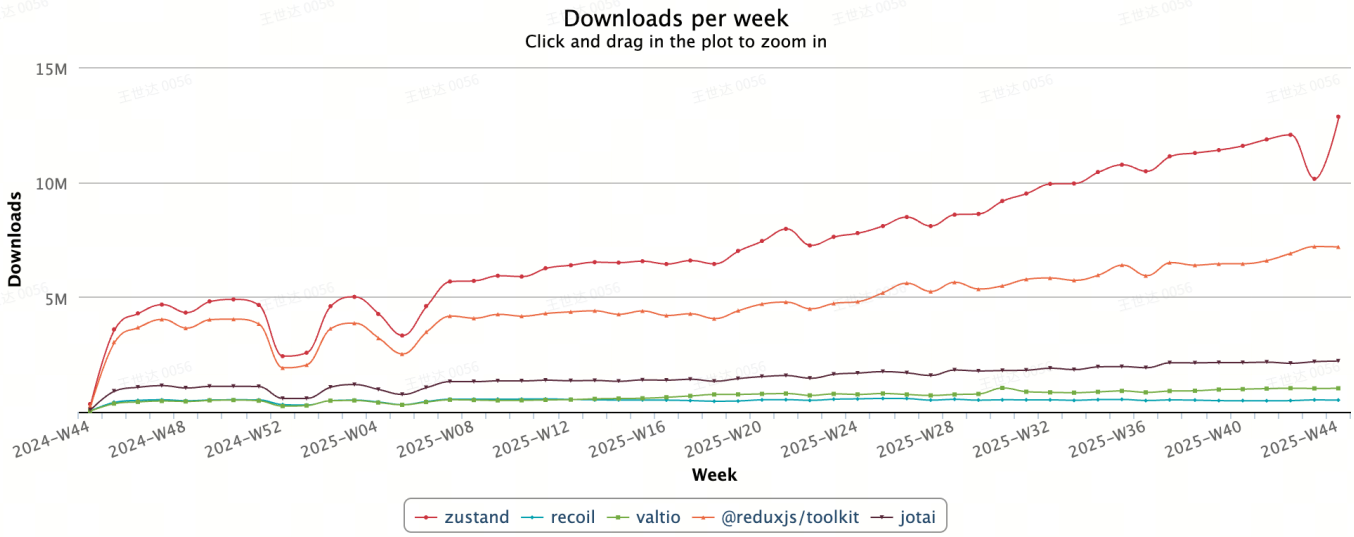
状态管理库一般能够处理异步数据流，支持中间件机制，管理状态变更历史等，这些都是 React Context 无法做到的。

适用场景

轻量全局状态（theme、locale、userInfo）/ 简单只读状态

状态管理库

大型 React 应用，一般会采用状态管理库来统一管理状态。社区里也存在各式各样的开源方案可用，从原理上大概可以分为三类范式。

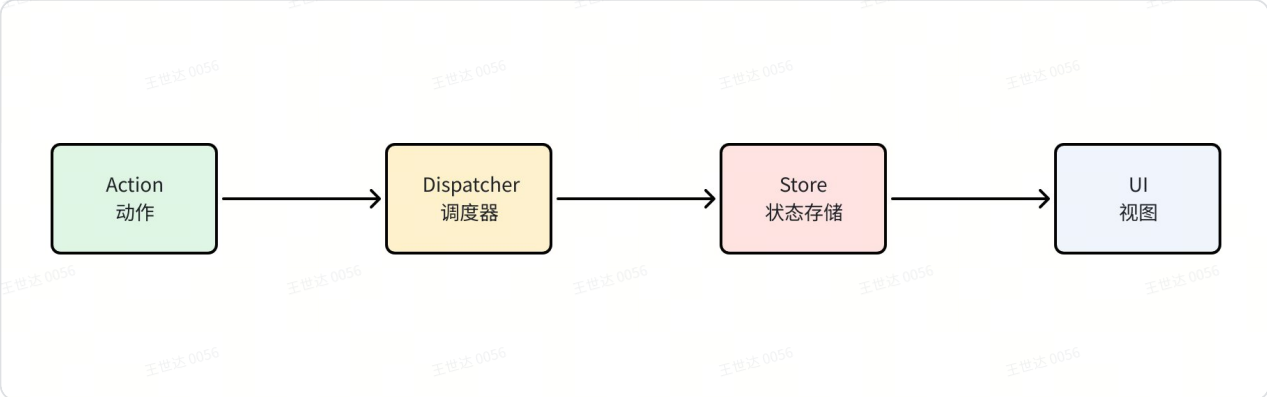


状态管理范式

1. Flux

代表库如 Redux、Zustand。

- Flux 的核心原则是**数据应该沿单一方向流动，**这使得应用程序的逻辑更易于预测和理解。



- Action 携带 payload 数据，经由 Dispatcher 更新到 Store。UI 层订阅了 Store 的变化，会随着 Store 的变化自动更新视图。

2. Atomic State

代表库如 Recoil、Jotai。

- 核心思想：将状态切成很多小的 “原子” （atoms）——每个 atom 是状态的最小单元，可被读／写。然后还有 Selectors（派生状态）——基于 atom 或其它 selector 计算得来。
- 保留了一定的单向数据流，但更强调 “细粒度” 状态管理、组件订阅特定状态、以及 derived state。

3. Proxy State

代表库如 Valtio、MobX。

- 利用了 js 的 Proxy 或类似机制，实现对状态对象的“代理”监控，从而实现响应式（reactivity）——状态更改后，自动触发 UI 更新。
- 相对比 Flux 那种强结构，Proxy 模式更「直觉」，更少样板代码。在大型应用复杂度上来后，管理会比较乱。

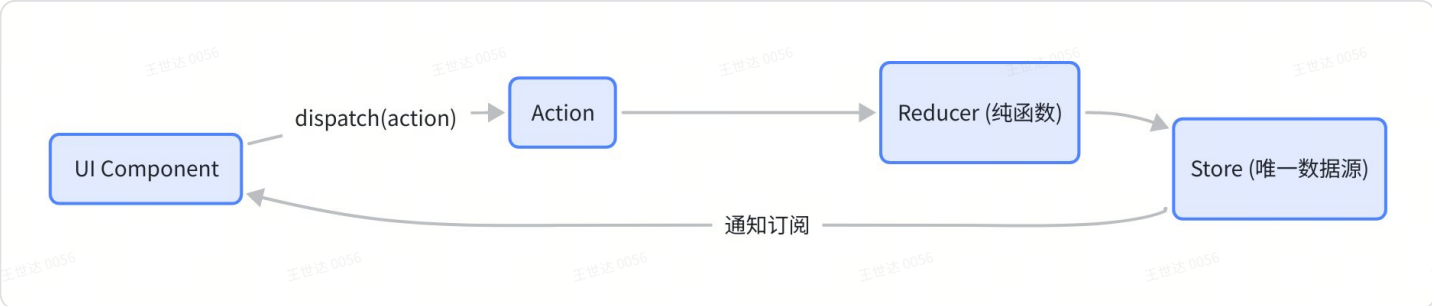
Redux

Redux 是一个用于可预测和可维护的全局状态管理的 JS 库。

核心思想

与 Flux 一致，在其基础上做了一定的简化。

- **单一数据源**：应用程序的全局状态存储在单个 Store 中。
- **状态只读：**改变状态只能通过发出一个 Action 来触发。
- **通过纯函数变更**：即 Reducer，本质上是纯函数，它接收前一个状态和一个 action，并返回下一个状态。



最小化示例(Redux Toolkit)

Redux 本身设计非常精简。导致手动初始化 Redux 配置，直到能够正常处理前端状态及副作用的过程是比较复杂的。

推荐使用 Redux Toolkit 来帮助配置，以及使用 React Redux 来与 React 视图层进行连接。

> 📌 相比传统 Redux，Redux Toolkit 做了什么
>
> https://redux-toolkit.js.org/introduction/why-rtk-is-redux-today#how-redux-toolkit-is-different-from-the-redux-core

- 初始化 Store

代码块

```
import { combineSlices, configureStore } from "@reduxjs/toolkit"
import { counterSlice } from "../features/counter/counterSlice"
import { quotesApiSlice } from "../features/quotes/quotesApiSlice"
```

```
4
5  const rootReducer = combineSlices(counterSlice, quotesApiSlice)
6
7  export const store = configureStore({
8    reducer: rootReducer,
9  })
```

- 包裹 Provider

代码块

```
1  import { Provider } from "react-redux"
2  import { store } from "./store"
3
4  const root = createRoot(container)
5
6  root.render(
7      <StrictMode>
8        <Provider store={store}>
9          <App />
10       </Provider>
11     </StrictMode>,
12 )
```

- 定义状态分片 Slice

代码块

```
1  import { asyncThunkCreator, buildCreateSlice } from "@reduxjs/toolkit"
2
3  export const createAppSlice = buildCreateSlice({
4    creators: { asyncThunk: asyncThunkCreator },
5  })
6
7  const initialState = {
8    value: 0,
9    status: "idle",
10 }
11
12 export const counterSlice = createAppSlice({
13   name: "counter",
14   initialState,
15   reducers: create => ({
16     incrementByAmount: create.reducer(
17       (state, action) => {
18         state.value += action.payload
```

```
      },
    ),
    incrementAsync: create.asyncThunk(
      async (amount: number) => {
        const response = await fetchCount(amount)
        return response.data
      },
      {
        pending: state => {
          state.status = "loading"
        },
        fulfilled: (state, action) => {
          state.status = "idle"
          state.value += action.payload
        },
        rejected: state => {
          state.status = "failed"
        },
      },
    ),
  }),
})
```

- 连接视图

代码块

```
import { useDispatch, useSelector } from "react-redux"
import { incrementAsync, incrementByAmount } from "./counterSlice"

export const Counter = (): JSX.Element => {
  const dispatch = useDispatch()
  const count = useSelector(state => state.count)

  return (
    <button
      className={styles.button}
      onClick={() => dispatch(incrementByAmount(3))}
    >
      Add Amount
    </button>
  )
}
```

异步请求处理（RTK Query）

RTK Query 从“管理状态”转向“管理缓存数据”。开发者定义数据来源与失效策略，框架完成请求生命周期、缓存、挂载/卸载时机管理与 hooks 生成。

- 使用 RTK Query 请求 API

代码块

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Pokemon } from './types'

export const pokemonApi = createApi({
  reducerPath: 'pokemonApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  endpoints: (build) => ({
    getPokemonByName: build.query<Pokemon, string>({
      query: (name) => `pokemon/${name}`,
    }),
  }),
})

export const { useGetPokemonByNameQuery } = pokemonApi
```

- 更新 store config

代码块

```
import { pokemonApi } from './services/pokemon'
export const store = configureStore({
  reducer: {
    [pokemonApi.reducerPath]: pokemonApi.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(pokemonApi.middleware),
})
```

- 调用 API

代码块

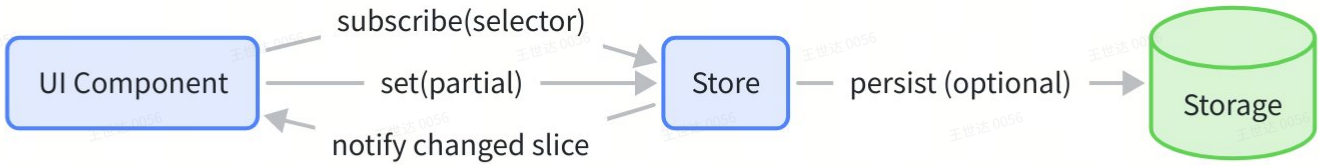
```
export const App = () => {
    const { data, error, isLoading } = useGetPokemonByNameQuery('bulbasaur')
}
```

Zustand

Zustand 是一个“更近组件、订阅更细、结构不强制”的状态管理方案。它把“选择器 + 订阅”放在一等公民位置，让你只在真正相关的状态变化时重渲染，减少无谓开销。

核心思想

沿袭了 Flux 的核心：单向数据流。组件触发动作（调用 set 或封装的 action），更新 store；选择器提取片段；订阅者按需重渲染。它去掉了繁琐的“必须是 action + reducer”的显式结构，把“怎么组织”交给开发者。



最小化示例

代码块

```
import { create } from 'zustand'

// 类型定义
type CounterState = {
  count: number
  increment: () => void
  reset: () => void
}

// 创建 store（作为 hook 使用）
export const useCounterStore = create<CounterState>(((set) => ({
  count: 0,
  otherState: "test",
  increment: () => set((s) => ({ count: s.count + 1 })),
  reset: () => set({ count: 0 }),
})))

// 组件按需订阅
function Counter() {
  const count = useCounterStore((s) => s.count)
  const increment = useCounterStore((s) => s.increment)
  return (
    <div>
      <span>Count: {count}</span>
      <button onClick={increment}>+1</button>
```

```
      </div>
    )
  }
```

异步请求处理

不同于 Redux，Zustand 里处理异步请求，无需引入单独的异步请求框架或者范式。直接在 store 里定义 async function 进行处理。

代码块

```
export const useCounterStore = create<CounterState>(((set) => ({
  count: 0,
  loading: false,
  error: null,
  increment: () => set((s) => ({ count: s.count + 1 })),
  reset: () => set({ count: 0 }),
  fetchRemoteCount: async () => {
      set({ loading: true })
      try {
        const res = await fetch(`xxxx`)
        const data = await res.json()
        set({ user: data, loading: false })
      } catch (err) {
        set({ error: err.message, loading: false })
      }
  }
})))
```

最佳实践

- 按领域划分 Store，如 `Theme`、`Notification`、`WizardForm`
- 复杂对象需要构建新的结构体来更新，或者使用 Immer
- 使用 useShallow 来阻止不必要的 rerender

路由管理

传统应用切换页面时，都需要重新加载 html，整体体验比较差。

React 应用里一般会采用**路由管理工具**来在 SPA 应用里实现多页面切换的能力。

SPA 的本质都是监听路由变化（hashchange/popchange），通过 JS 来动态将对应 Page 的 HTML 片段插入到当前页面的某个节点（Container）下。

目前最主流的 React 路由库是 React Router。

用法概括如下：

路由接入	代码块 1 import { BrowserRouter, Routes, Route } from "react-router-dom"; 2 import Home from "./pages/Home"; 3 import About from "./pages/About"; 4 5 function App() { 6 return (7 <BrowserRouter> 8 <Routes> 9 <Route path="/" element={<Home />} /> 10 <Route path="/about" element={<About />} /> 11 <Route path="/user/:id" element={<User />} /> 12 </Routes> 13 </BrowserRouter> 14); 15 } 16 17 export default App;
路由跳转	代码块 1 import { Link } from "react-router-dom"; 2 3 function Nav() { 4 return (5 <nav> 6 <Link to="/">首页</Link> 7 <Link to="/about">关于</Link> 8 </nav> 9); 10 } 11 12 function Login() { 13 const navigate = useNavigate(); 14 15 function handleLogin() { 16 // 登录逻辑...

```
      navigate("/"); // 跳转到首页
    }

    return <button onClick={handleLogin}>登录</button>;
  }
```

路由参数

代码块

```
// 定义路由参数，正则规范
<Route path="/user/:id" element={<User />} />

// 获取路由参数
import { useParams } from "react-router-dom";
function User() {
  const { id } = useParams();
  return <h1>用户 ID: {id}</h1>;
}
```

路由嵌套

代码块

```
<Route path="/dashboard" element={<Dashboard />}>
  <Route path="profile" element={<Profile />} />
  <Route path="settings" element={<Settings />} />
</Route>

import { Outlet } from "react-router-dom";

function Dashboard() {
  return (
    <div>
      <Sidebar />
      <Outlet /> {/* 子路由组件会渲染在这里 */}
    </div>
  );
}
```

一般情况下使用 BrowserRouter，基于 HTML5 History API。路由更新后，触发 popupstate 事件，React-Router 获取 path 后进行 match，然后执行渲染逻辑。

样式管理

在 React 应用里，引入 CSS 存在多种方式。

方案对比

• 直接引用 CSS 文件	代码块 1 // 直接引用 CSS 文件 2 import "./App.css" 3 4 function App() { 5 return Hello ; 6 }	优点 • 简单容易理解 缺点 • 类名容易冲突 • 样式是全局作用的 • 难以做组件化封装
• CSS Modules	代码块 1 // CSS Modules: 以模块形式引入 2 import styles from "./styles.module.css" 3 4 function App() { 5 return Hello ; 6 }	优点 • 组件级样式隔离 • 结合 classnames 库动态控制 • 无类名冲突 缺点： • 本地开发通过类名定位代码比较麻烦，已有插件解决 • 组件被引用后，如果需要样式定制无法直接使用 css 修改。**推荐通过组件属性进行定制**。
• CSS In JS	代码块 1 import styled from "styled-components"; 2 3 const Button = styled.button` 4 background: ${(props) => (props.primary ? "blue" : "gray")}; 5 color: white; 6 border-radius: 5px; 7 padding: 8px 12px;	优点 • 能够在 JS 里直接编写样式 • 样式与组件强绑定； • 无类名冲突； 缺点： • 编译时性能略低

	`` `; function App() { return (<> <Button>普通按钮</Button> <Button primary>主要按钮</Button> </>); } ```	• 大量组件时运行时样式注入开销略大 • 同时存在有 CSSModules 存在的缺点
• 原子化 CSS 框架	代码块 ``` function App() { return (<div className="p-4 bg-blue-500 text-white rounded-lg shadow-md"> Hello Tailwind </div>); } ```	优点 • 不再写样式文件； • 原子化、组件化； • 构建时优化性能； 缺点 • 上手难度高 • 类名多时可读性较差； • 自定义复杂样式需要配置。

推荐的方式

在开发 React 应用时，一般推荐结合 UI 组件库使用，**设计上遵循组件库的 UI 风格，尽可能避免自定义样式**。

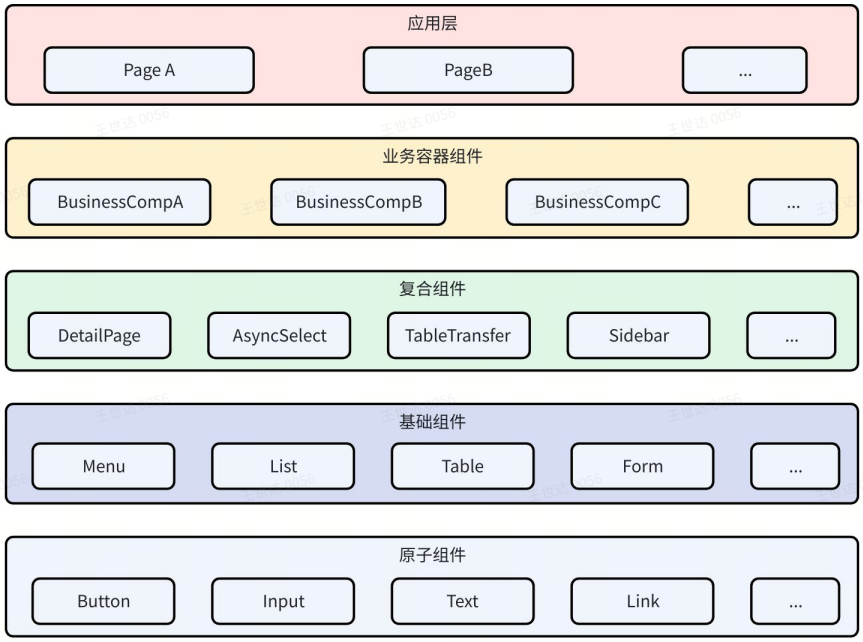
无可避免自定义样式时，**通过 CSS Modules 的方式进行定制**。

常见 React 应用设计原则

这个章节将介绍几种常用的设计原则， 来让我们写出模块化、低耦合、可维护、可扩展、性能良好的 React 应用。

遵循原子设计模式	原子设计的五个不同层次：原子 > 分子 > 有机体 > 模板 > 页面，与 React 的组件化架构完美契合。

在设计和组织 React 组件时，也可以类似的维度进行划分：原子组件 > 基础组件 > 复合组件 > 业务容器组件 > 应用层。



业务逻辑与UI渲染分开

将“获取/计算数据 + 业务逻辑”与“UI 渲染”分开，一个容器组件负责 fetch API、管理 loading/error 状态，然后传给一个展示组件渲染。

代码块

```jsx
// UserList.jsx — 展示组件 (Presentational Component)
import React from "react";

export const UserList = ({ users, loading, error }) => {
  if (loading) return <p>Loading users...</p>;
  if (error) return <p style={{ color: "red" }}>❌ {error}</p>;

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>
          👤 {user.name} ({user.email})
        </li>
      ))}
    </ul>
  );
};
```

代码块

```
// UserListContainer.jsx — 容器组件 (Container Component)
import React, { useEffect, useState } from "react";
import { UserList } from "./UserList";

export const UserListContainer = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchUsers = async () => {
      // ...
    };

    fetchUsers();
  }, []);

  // 把数据和状态交给展示组件
  return <UserList users={users} loading={loading} error={error} />;
};
```

使用自定义 Hook 复用业务逻辑

利用自定义 hook 将可复用的“状态逻辑 / 副作用逻辑”抽离出来

代码块

```
// useUsers.js
import { useEffect, useState } from "react";

export function useUsers() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchUsers = async () => {
      // ...
    };

    fetchUsers();
  }, []);

  // ✅ 返回数据与状态，让外部组件使用
  return { users, loading, error };
```

```
}
```

代码块

```jsx
// UserListContainer.jsx — 容器组件 (Container Component)
import React, { useEffect, useState } from "react";
import { UserList } from "./UserList";

export const UserListContainer = () => {
  const { users, loading, error } = useUsers();

  // 把数据和状态交给展示组件
  return <UserList users={users} loading={loading} error={error} />;
};
```

使用 Provider 管理全局状态	使用 Context + Provider 管理全局状态，避免“prop drilling”（层层传 prop）的问题
引入状态管理库管理复杂状态	-
按路由延迟加载页面	React.lazy() + Suspense 延迟加载，按页面划分

代码块

```jsx
import { lazy, Suspense } from 'react';
import { BrowserRouter, Routes, Route } from "react-router-dom";

const Home = lazy(() => import('@/pages/Home'));
const About = lazy(() => import('@/pages/About'));

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Suspense fallback={<Loading />}>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/user/:id" element={<User />} />
        </Suspense>
```

```
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

避免重复渲染或重复计算

- 使用 React.memo、useMemo、useCallback 来避免不必要的重新渲染或重复计算

代码块

```
import React, { useState, useMemo, useCallback, memo } from "react";

// 👇 子组件：UserList —— 使用 React.memo 避免不必要渲染
const UserList = memo(({ users, onSelect }) => {
  console.log("👀 UserList rendered");

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id} onClick={() => onSelect(user)}>
          👤 {user.name}
        </li>
      ))}
    </ul>
  );
});

// 👇 父组件：UserSearch
export const UserSearch = () => {
  const [search, setSearch] = useState("");
  const [selected, setSelected] = useState(null);

  const allUsers = [
    { id: 1, name: "Alice" },
    { id: 2, name: "Bob" },
    { id: 3, name: "Charlie" },
  ];

  // ✅ useMemo：只有当 search 改变时才重新计算过滤结果
  const filteredUsers = useMemo(() => {
    console.log("🔍 Filtering users...");
```

```
    return allUsers.filter((u) =>
u.name.toLowerCase().includes(search.toLowerCase())
    );
  }, [search]);

  // ✅ useCallback: 保持函数引用稳定，避免触发子组件重复渲染
  const handleSelect = useCallback((user) => {
    setSelected(user);
  }, []);

  return (
    <div style={{ fontFamily: "sans-serif" }}>
      <h2>🔎 User Search</h2>
      <input
        type="text"
        value={search}
        placeholder="Search user..."
        onChange={(e) => setSearch(e.target.value)}
      />

      <UserList users={filteredUsers} onSelect={handleSelect} />

      {selected && <p>✅ Selected: {selected.name}</p>}
    </div>
  );
};
```

或者

使用 UI 库

这一章节将讲述如何使用成熟的 UI 组件库，来加速我们的开发。课程采用 ArcoDesign 组件库进行演示。

使用示例

https://codesandbox.io/embed/reverent-voice-v2yzx?fontsize=14&hidenavigation=1&theme=dark

定制主题

ArcoDesign 使用了 Less 作为预编译语言，通过 Less 的 modifyVars 功能，可以很方便的对样式粒子变量进行定制。

1. 从 less 文件引用样式

代码块

```less
// global.less
import '@arco-design/web-react/dist/css/index.less';
```

2. 参考 ArcoDesign `components/style/theme/global.less` 定制 less 变量。或者查看组件粒度变量进行覆盖 `components/Button/style/token.less`

代码块

```less
// theme.less
@color-text-1: #232323;

@btn-size-mini-radius: 4px;
```

3. 或者在打包工具的 less-loader 里修改环境变量

代码块

```
// webpack.config.js
module.exports = {
  rules: [{
    test: /\.less$/,
    use: [{
      loader: 'style-loader',
    }, {
      loader: 'css-loader',
    }, {
      loader: 'less-loader',
+     options: {
+       modifyVars: {  // 在less-loader@6 modifyVars 配置被移到 lessOptions 中
+         'arcoblue-6': '#f85959',
+       },
+       javascriptEnabled: true
+     },
    }],
    ...
  }],
  ...
```

```
21   }
```

常用组件

接下来分析一些高频组件及其用法

布局组件 Layout

页面的基础布局框架，常与组件嵌套使用，构建页面整体布局。

https://codesandbox.io/embed/dyk53j?view=editor+%2B+preview&module=%2Fdemo.js

表格组件 Table

用于数据收集展示、分析整理、操作处理。

https://codesandbox.io/embed/5l94wy?view=editor+%2B+preview&module=%2Fdemo.tsx

表单组件 Form

具有数据收集、校验和提交功能的表单，包含复选框、单选框、输入框、下拉选择框等元素。

https://codesandbox.io/embed/mvfvbl?view=editor+%2B+preview&module=%2Fdemo.js

对话框 Modal

在当前页面打开一个浮层，承载相关操作。

https://codesandbox.io/embed/57ftfq?view=editor+%2B+preview&module=%2Fdemo.js

课后作业

电商商品列表页与详情页开发

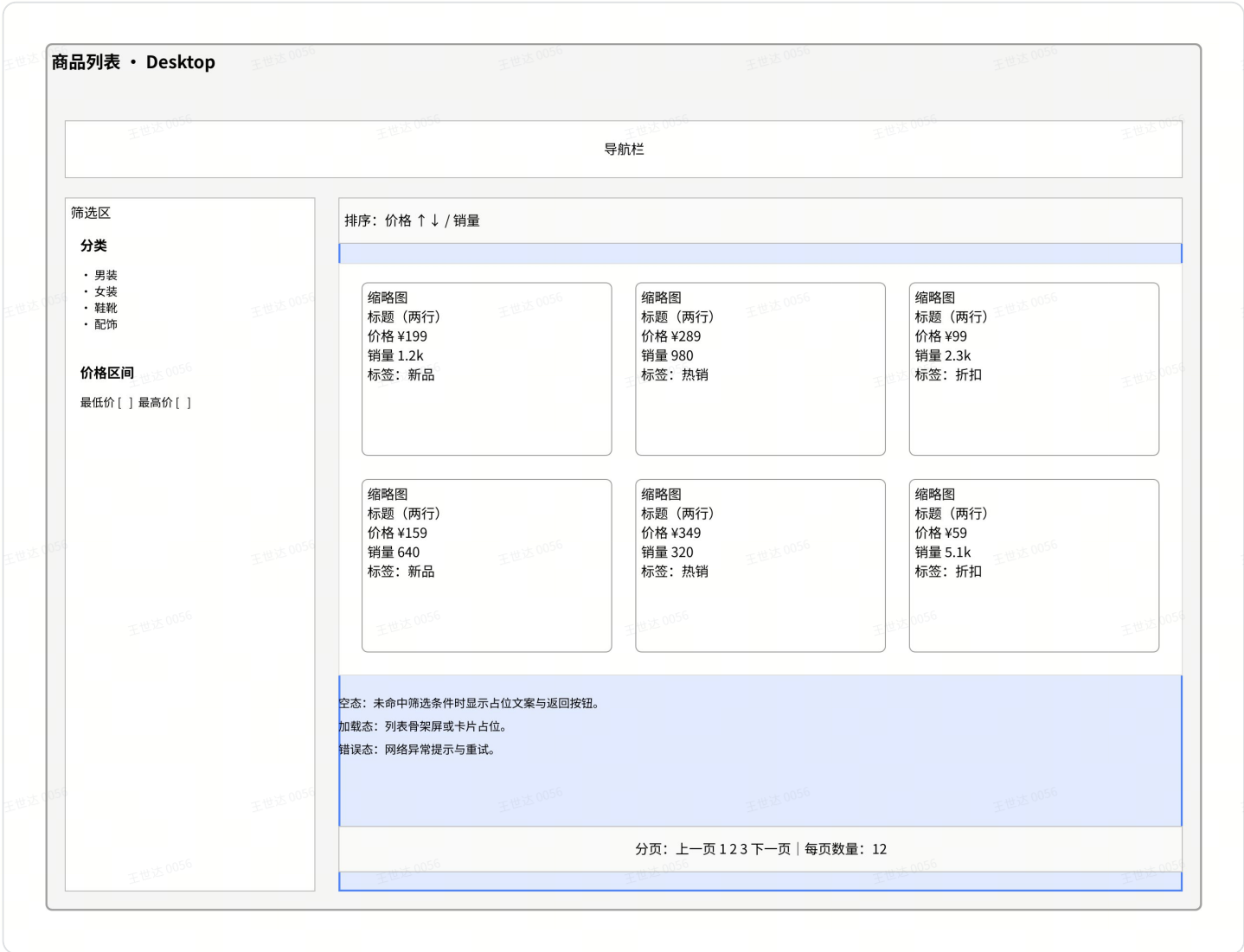
1. 项目需求：实现电商平台的商品列表页（支持商品筛选、排序、分页）与商品详情页（展示商品信息、规格选择、加入购物车），要求组件化拆分合理、状态管理清晰、支持响应式
2. 实战步骤

- 第一步：项目初始化（使用 Vite/Rspack 创建项目，配置 UI 库、状态管理库（如 Redux Toolkit / Zustand））
- 第二步：组件拆分，将页面拆分为通用组件（导航栏、分页器、筛选组件）、业务组件（商品卡片、规格选择器、购物车弹窗）、页面组件（商品列表页、商品详情页）
- 第三步：实现组件通信与状态管理，通过 Props 实现父子组件数据传递，使用状态管理库存储全局数据（商品列表、筛选条件、购物车数据）

- 第四步：开发响应式 UI，利用 UI 库与自定义样式，确保页面在电脑、平板、手机端均有良好展示效果
- 第五步：模拟数据交互（使用 Mock.js 生成商品数据），实现商品筛选（价格、分类）、排序（价格高低、销量）、分页加载、加入购物车功能
- 第六步：代码优化与调试，检查组件复用性、状态管理合理性，解决响应式适配问题与交互 bug

原型设计

- 列表页（Desktop / Tablet / Mobile）原型：



- 详情页（Desktop / Tablet / Mobile）原型：

商品详情 · Desktop

导航栏

主图轮播区

信息与规格选择区

商品标题（两行）
价格：¥299

尺码

S M L XL

颜色

黑 白 蓝

库存：根据所选尺码/颜色展示可用数量；无库存时禁用加入购物车。

加入购物车：数量选择器 + 按钮

购物车抽屉：加入后从右侧滑出，展示购物车项列表与结算按钮。

缩略图 缩略图 缩略图 缩略图 缩略图

推荐/相似商品

卡片（4-6 个）