React18 核心面试题详细解答

一、React18 核心源码精读

React18 的核心升级围绕**并发渲染（Concurrent Rendering）**展开，核心源码可聚焦于调度层、渲染层的关键模块和特性：

1. 核心架构分层（源码核心目录）

React 源码采用分层架构，核心分为三层，对应源码目录（`packages` 下）：

- **调度层（Scheduler）**：核心包 `scheduler`，负责任务优先级排序、时间切片（Time Slicing），是并发渲染的基础。
- **协调层（Reconciler）**：核心包 `react-reconciler`，负责虚拟 DOM 对比（Diff 算法）、Fiber 节点构建与更新调度，React18 中重构为「并发协调器」。
- **渲染层（Renderer）**：如 `react-dom`（浏览器端）、`react-native`（原生端），负责将协调层产出的 Fiber 树转化为真实 DOM 或原生组件。

2. 核心新特性源码关键点

（1）并发渲染（Concurrent Rendering）

- 核心概念：不再是同步阻塞式渲染，而是可以**中断、暂停、恢复或放弃**的渲染流程，源码中通过 `isConcurrentMode` 标识开启（React18 中通过 `createRoot` 默认启用）。
- 关键实现：基于 Fiber 架构的「可中断遍历」，Fiber 节点作为最小调度单元，每个 Fiber 节点遍历后都会检查是否有更高优先级任务（如用户输入），若有则暂停当前渲染，保存上下文后切换任务。

（2）自动批处理（Automatic Batching）

- 核心逻辑：源码中通过 `unstable_batchedUpdates` 方法封装批处理逻辑，React18 中取消了「仅合成事件内批处理」的限制，将 Promise、setTimeout、原生事件等异步场景的状态更新统一纳入批处理。
- 关键细节：批处理的本质是延迟执行 `flushSync`（刷新状态更新），当所有批量更新收集完成后，一次性执行协调和渲染，减少 DOM 操作次数。

（3）Transitions（过渡更新）

- 核心 API：`useTransition`、`startTransition`，源码中通过标记更新的优先级（`TransitionPriority` 低于普通更新）实现。

- 关键实现：标记为 Transition 的更新会被归类为「非阻塞更新」，当有高优先级任务（如输入框输入）时，该更新会被暂停，等高优先级任务执行完成后再恢复，避免页面卡顿。

（4）createRoot 与 legacyRoot

- 源码差异：`createRoot`（React18 推荐）创建的是「并发根节点」，启用所有新特性；`ReactDOM.render`（旧版）创建的是 `legacyRoot`（遗留根节点），兼容旧版同步渲染逻辑。
- 核心区别：根节点初始化时，`createRoot` 会初始化并发调度器，而 `legacyRoot` 仅初始化同步调度器。

（5）Suspense 升级（数据加载支持）

- 源码实现：通过 `SuspenseComponent` Fiber 节点管理加载状态，当子组件抛出 `Promise` 时，React 会捕获该 Promise，暂停当前组件渲染，展示 fallback 内容，待 Promise 解析后再恢复渲染。
- 关键改进：React18 中 Suspense 支持服务端渲染（SSR）和并发渲染场景，源码中新增了 `SuspenseList` 组件用于控制多个 Suspense 组件的渲染顺序。

二、Redux 核心源码精读

Redux 是一款轻量级状态管理库，核心源码简洁（核心逻辑仅几百行），围绕「单向数据流」和「不可变状态」展开：

1. 核心模块与入口

Redux 核心包暴露 4 个关键 API：`createStore`（创建仓库）、`combineReducers`（合并 reducer）、`applyMiddleware`（应用中间件）、`compose`（组合函数），核心逻辑集中在 `createStore` 中。

2. createStore 核心实现

- 初始化：接收 `reducer`、`preloadedState`（初始状态）、`enhancer`（增强器，如中间件），内部维护 `currentState`（当前状态）、`currentReducer`（当前 reducer）、`currentListeners`（监听函数数组）。
- 核心方法：
  a. `getState()`：直接返回 `currentState`，获取当前状态（无副作用）。
  b. `dispatch(action)`：唯一修改状态的入口，逻辑为：`currentState = currentReducer(currentState, action)`，执行 reducer 计算新状态，随后遍历执行所有监听函数。
  c. `subscribe(listener)`：将监听函数加入 `currentListeners`，返回一个取消订阅的函数（用于移除监听）。

d. `replaceReducer(nextReducer)`：替换当前 reducer，用于代码分割或动态切换 reducer。

3. combineReducers 核心实现

- 核心作用：将多个子 reducer 合并为一个根 reducer，解决复杂应用状态拆分的问题。
- 实现逻辑：
  a. 校验传入的 reducer 对象，过滤无效 reducer（非函数类型）。
  b. 返回一个新的根 reducer，该 reducer 执行时，会遍历所有子 reducer，分别传入对应子状态和 action，计算出各子状态的新值，最终合并为完整的根状态。
  c. 关键细节：只有子 reducer 返回新状态时，根状态对应的属性才会更新，保证状态不可变。

4. applyMiddleware 核心实现

- 核心作用：扩展 dispatch 功能，支持异步操作（如网络请求）、日志打印等副作用处理（Redux 原生 dispatch 仅支持同步 action）。
- 实现逻辑：
  a. 接收多个中间件作为参数，返回一个「仓库增强器」（函数）。
  b. 该增强器接收 `createStore` 作为参数，返回一个新的 `createStore` 函数，新函数创建仓库时，会对 `dispatch` 进行包装。
  c. 通过 `compose` 函数组合所有中间件，形成一个中间件链，最终返回一个增强后的 dispatch，使得 action 会依次经过中间件处理后，再传入原生 dispatch。
  d. 经典中间件：`redux-thunk`（支持函数类型 action）、`redux-saga`（异步流程管理）、`redux-logger`（打印 action 和状态日志）。

5. compose 核心实现

- 核心作用：从右到左组合多个函数，形成一个复合函数，用于中间件组合、仓库增强器组合等场景。
- 实现逻辑：若传入空参数，返回一个恒等函数（`x => x`）；若传入单个函数，直接返回该函数；若传入多个函数，通过 `reduce` 方法从右到左依次组合，前一个函数的返回值作为后一个函数的参数。

三、React 设计思路与理念，及与 Vue 的异同

1. React 核心设计思路与理念

React 的设计围绕「组件化」和「声明式编程」展开，核心理念有 4 点：

- **声明式编程（Declarative）**：开发者只需描述「UI 应该是什么样子」（基于状态），无需关心「如何一步步修改 DOM」，React 内部自动完成从状态到 DOM 的映射，相比命令式编程更简洁、易维护。
- **组件化思想（Componentization）**：将复杂 UI 拆分为独立、可复用的组件，组件拥有自己的状态和生命周期，支持组件嵌套组合，形成组件树，实现「高内聚、低耦合」的代码结构。
- **单向数据流（Unidirectional Data Flow）**：状态（State）只能从父组件传递给子组件（通过 Props），子组件不能直接修改父组件传递的 Props，若需修改状态，需通过父组件传递的回调函数触发，保证状态流向清晰、可追溯。
- **虚拟 DOM（Virtual DOM）与不可变状态（Immutability）**：
  a. 虚拟 DOM 是对真实 DOM 的轻量级描述，状态变化时，React 先对比虚拟 DOM 的差异（Diff 算法），再批量更新真实 DOM，减少 DOM 操作开销。
  b. 不可变状态：状态一旦创建就不能直接修改，需通过 `setState`（类组件）或 `useState`（函数组件）返回的更新函数创建新状态，保证状态变化可追踪、便于实现时间旅行等功能。

2. React 与 Vue 的异同

（1） 相同点

1. **核心目标一致**：都是用于构建用户界面的前端框架，解决原生 JavaScript 操作 DOM 繁琐、效率低下的问题。
2. **支持组件化**：均以组件为核心构建单元，支持组件嵌套、复用，提高代码可维护性和复用性。
3. **声明式编程**：均采用声明式语法，开发者只需关注状态与 UI 的映射关系，无需手动操作 DOM。
4. **虚拟 DOM**：内部均实现了虚拟 DOM 机制，通过 Diff 算法减少真实 DOM 操作，提升渲染性能。
5. **支持状态管理**：均有对应的状态管理方案（React + Redux/MobX；Vue + Vuex/Pinia），解决复杂应用的状态共享问题。

（2） 不同点

对比维度	React	Vue
核心语法	基于 JSX 语法，将 HTML 嵌入 JavaScript 中（JS 优先）	基于 SFC（单文件组件），将 HTML、CSS、JS 分块编写（HTML 优先），也支持 JSX
组件类型	函数组件（React16.8 后推荐）、类组件	选项式组件（Vue2 推荐、Vue3 兼容）、组合式组件（Vue3 推荐，基于 `<script setup>`）
状态管理		

	类组件：`setState`（异步更新，批量处理）；函数组件：`useState` / `useReducer`	Vue2：`data` 选项（响应式，直接修改）；Vue3：`ref` / `reactive`（响应式，可直接修改）
响应式原理	无原生响应式，依赖状态更新函数（`setState` / `useState`）触发重新渲染，React18 支持并发渲染	Vue2：基于 `Object.defineProperty`（无法监听数组下标、对象新增属性）；Vue3：基于 `Proxy`（全方位监听对象/数组变化，性能更优）
Diff 算法	基于 Fiber 架构的可中断 Diff 算法，按组件层级遍历，优先级高的任务可中断低优先级任务	传统同步 Diff 算法（Vue2/Vue3），Vue3 优化了静态节点标记，减少对比开销，无并发中断能力
生态体系	生态庞大且灵活，配套工具丰富（React Router、Redux、Next.js 等），更偏向「库」，需要开发者自行整合方案	生态完善且一体化，Vue 官方提供全套解决方案（Vue Router、Vuex/Pinia、Vite、Nuxt.js 等），更偏向「框架」，开箱即用
适用场景	更适合大型、复杂的前端应用（如电商后台、企业级系统），在 React Native 移动端开发中优势明显	入门门槛低，更适合中小型应用、快速迭代项目，在国内中小企业中普及率更高

四、React 事件机制（与普通 DOM 事件的区别及工作原理）

1. 什么是 React 事件机制

React 事件机制是**自定义事件系统（合成事件 SyntheticEvent）**，并非原生 DOM 事件的直接封装，而是 React 基于原生 DOM 事件封装的跨浏览器、统一规范的事件系统，解决了不同浏览器之间的事件兼容问题，同时集成了 React 的核心特性（如批量更新、 Fiber 调度）。

2. React 合成事件与普通 DOM 事件的区别

对比维度	React 合成事件	普通 DOM 事件
事件绑定对象	绑定在 React 组件的虚拟 DOM 上，最终会委托到 `document` 上（React17 后改为绑定到根节点 `root`）	直接绑定在真实 DOM 元素上

事件类型命名	采用驼峰式命名（如 `onClick`、`onChange`）	采用小写字母命名（如 `onclick`、`onchange`）
事件对象	合成事件对象（SyntheticEvent），兼容所有浏览器，提供与原生事件对象一致的 API（如 `target`、`preventDefault`）	原生 DOM 事件对象，不同浏览器存在兼容性差异
事件冒泡	合成事件冒泡：在 React 组件树中冒泡，与原生 DOM 冒泡独立，可通过 `e.stopPropagation()` 阻止合成事件冒泡，但无法阻止原生事件冒泡（反之亦然）	原生 DOM 事件冒泡：在真实 DOM 树中冒泡
事件解绑	无需手动解绑（函数组件卸载后自动失效，类组件通过 `this` 绑定的事件在组件卸载时需手动清理，或使用箭头函数避免内存泄漏）	需手动调用 `removeEventListener` 解绑，否则可能导致内存泄漏
批量更新支持	合成事件回调中触发的 `setState` 会进行批量更新，优化性能	原生 DOM 事件回调中触发的 `setState`（React17 前）不支持批量更新，React18 后通过自动批处理支持

3. React 事件机制的工作原理（三步流程）

（1） 事件注册（初始化阶段）

- React 在组件挂载时，会遍历组件的事件属性（如 `onClick`），将其注册为**事件委托**（React16 及以前委托到 `document`，React17 及以后委托到组件挂载的根节点 `root`），而不是直接绑定到组件对应的真实 DOM 元素上。
- 注册时，React 会为每个合成事件分配一个唯一的标识，并将事件回调函数与标识关联，存储在内部的回调函数映射表中。

（2） 事件触发（运行时阶段）

1. 用户操作触发真实 DOM 事件，该事件会按照原生 DOM 冒泡机制冒泡到委托节点（`document` 或 `root`）。
2. 委托节点上的统一事件监听器捕获到该原生事件，React 会根据原生事件信息，创建对应的**合成事件对象（SyntheticEvent）**，并对合成事件对象进行封装（兼容不同浏览器的差异）。

3. React 根据原生事件的类型和目标元素，在回调函数映射表中查找对应的组件事件回调函数。

（3） 事件分发与执行（运行时阶段）

1. React 会在**合成事件冒泡阶段**，按照 React 组件树的层级，从触发事件的组件开始，向上遍历组件树，依次执行组件上注册的合成事件回调函数（即合成事件冒泡）。
2. 事件回调函数执行完成后，React 会立即回收合成事件对象（复用合成事件对象，提升性能），因此在异步回调中无法访问合成事件对象（如需访问，需调用 `e.persist()` 方法保留事件对象）。
3. 若回调函数中触发了 `setState`，React 会进行批量更新，待所有合成事件回调执行完成后，统一执行协调和渲染流程。

五、React.Component 与 React.PureComponent 的区别

React.Component（普通类组件）和 React.PureComponent（纯类组件）均是 React 类组件的基类，核心区别在于**是否自动实现了浅比较（Shallow Comparison）的 `shouldComponentUpdate` 方法**，具体差异如下：

1. 核心区别：shouldComponentUpdate 方法

- **React.Component**：默认没有实现 `shouldComponentUpdate` 方法，该方法默认返回 `true`。这意味着，无论组件的 `props` 或 `state` 是否发生变化，只要父组件重新渲染，或者组件自身调用 `setState`（即使传入相同的状态值），该组件都会触发重新渲染（包括自身 `render` 方法执行、子组件重新渲染），可能导致不必要的性能开销。
  - 若需优化性能，开发者需手动实现 `shouldComponentUpdate` 方法，手动对比 `nextProps` 与 `this.props`、`nextState` 与 `this.state` 的差异，返回 `true`（需要渲染）或 `false`（不需要渲染）。
- **React.PureComponent**：自动实现了 `shouldComponentUpdate` 方法，该方法内部会对 `props` 和 `state` 进行**浅比较（Shallow Compare）**：
  a. 浅比较规则：先比较引用类型的引用地址（若 `props` / `state` 是对象/数组，仅比较是否指向同一个内存地址，不深入对比对象内部属性），再比较基本类型的值（如数字、字符串）。
  b. 只有当浅比较发现 `props` 或 `state` 发生变化时，`shouldComponentUpdate` 才返回 `true`，触发组件重新渲染；若浅比较发现无变化，则返回 `false`，阻止组件不必要的重新渲染，优化性能。

2. 其他补充差异

1. **使用场景**：
   - React.Component：适用于所有类组件场景，尤其是组件的 `props` 或 `state` 为复杂嵌套对象/数组（浅比较无法检测到内部属性变化），或需要自定义渲染逻辑的场景。

- React.PureComponent：适用于组件的 `props` 和 `state` 均为基本类型，或为简单对象/数组（无嵌套结构），且组件渲染仅依赖 `props` 和 `state` 的场景，用于减少不必要的渲染，提升性能。

2. **注意事项**：
   - PureComponent 的浅比较存在「局限性」：若 `props` 或 `state` 是复杂嵌套对象/数组，即使内部属性发生变化，由于引用地址未变，浅比较会认为无变化，导致组件无法触发重新渲染（此时需通过创建新对象/数组的方式更新状态，如使用 `Object.assign`、扩展运算符 `...`）。
   - 不要在 PureComponent 中使用不可变数据以外的状态更新方式，否则会导致组件渲染异常。
3. **对应函数组件方案**：
   - React.Component 对应普通函数组件（无性能优化，父组件重新渲染则自身重新渲染）。
   - React.PureComponent 对应函数组件中的 `React.memo`（`React.memo` 是高阶组件，对函数组件的 `props` 进行浅比较，实现与 PureComponent 类似的性能优化效果，若需对比 `state`，可结合 `useMemo` / `useCallback`）。

六、React 的高阶组件（HOC）：定义、与普通组件的区别及使用场景

1. 什么是高阶组件（HOC）

高阶组件（Higher-Order Component，简称 HOC）是 React 中用于**复用组件逻辑**的高级技巧，本质是一个**函数**，而非组件本身。该函数接收一个或多个组件作为参数，经过逻辑处理后返回一个新的增强型组件。

核心公式：`HOC(WrapperComponent) = EnhancedComponent`，即通过高阶函数包装原始组件，生成具备额外能力的新组件，不修改原始组件的代码，符合「开闭原则」。

常见示例（如 React-Redux 的 `connect` 方法）：`const EnhancedComponent = connect(mapStateToProps, mapDispatchToProps)(OriginalComponent)`

2. HOC 与普通组件的区别

对比维度	高阶组件（HOC）	普通组件
本质	函数（接收组件，返回新组件）	React 元素的构建单元（类或函数）
核心作用	复用组件逻辑、增强组件能力（无 UI 渲染能力）	描述 UI 结构和交互（核心是渲染 UI）
输入输出	输入：组件 + 配置；输出：新组件	输入：Props；输出：JSX（UI）

生命周期	自身无生命周期，可在返回的新组件中添加生命周期逻辑增强原始组件	拥有完整的生命周期（类组件）或依赖 Hooks 模拟生命周期（函数组件）
使用方式	通过函数调用包装原始组件，生成新组件后使用	直接在 JSX 中渲染（如 <OriginalComponent />）

3. HOC 的常见使用场景

- **状态复用**：提取多个组件共有的状态逻辑，封装到 HOC 中，让多个组件无需重复编写即可拥有该状态。例如：用户登录状态管理（判断用户是否登录，未登录则跳转登录页）、表单数据管理（输入值绑定、表单验证逻辑）。
- **属性增强**：为多个组件统一注入固定的 Props 或动态计算的 Props。例如：为列表组件注入分页数据、为组件注入全局配置信息。
- **权限控制**：封装权限校验逻辑，根据用户权限决定是否渲染原始组件，或渲染降级组件。例如：管理员权限组件（仅管理员可查看）、游客权限组件（隐藏敏感操作按钮）。
- **日志埋点/性能监控**：在组件的生命周期（或 Hooks）中添加日志记录、性能统计逻辑，无需侵入组件内部代码。例如：记录组件的挂载/卸载时间、统计组件渲染耗时。
- **组件包装/样式统一**：为多个组件统一添加外层容器、样式或动画效果。例如：为所有页面组件添加统一的头部导航、为卡片组件添加统一的边框和阴影样式。

七、React 组件重新渲染的触发条件及 render 函数内部逻辑

1. 触发组件重新渲染的方法/操作

组件重新渲染的核心触发条件是「组件的状态或Props发生变化」，具体场景分为以下几类：

- **自身状态变化**：
- 类组件：调用 `setState` 方法（无论传入的新状态与旧状态是否相同，默认都会触发重新渲染，除非重写 `shouldComponentUpdate` 阻止）。
- 函数组件：调用 `useState` 或 `useReducer` 返回的更新函数（传入新状态时触发渲染，若新状态与旧状态完全相同，React 会跳过渲染优化）。

父组件传递的 Props 变化：

- 父组件重新渲染时，若传递给子组件的 Props 发生变化（浅比较差异），子组件会触发重新渲染。
- 即使 Props 未变化，若父组件重新渲染，子组件默认也会跟随重新渲染（除非通过 `PureComponent` 或 `React.memo` 优化）。

父组件强制触发子组件渲染：父组件通过 `forceUpdate` 方法强制自身重新渲染，此时所有子组件都会跟随重新渲染（`forceUpdate` 会跳过 `shouldComponentUpdate` 的校验）。

使用 context 时的变化：组件使用 `useContext`（函数组件）或 `static contextType`（类组件）订阅了 Context，当 Context 中的值发生变化时，所有订阅该 Context 的组件都会触发重新渲染。

使用 Hooks 时的依赖变化：函数组件中，`useEffect`、`useMemo`、`useCallback` 等 Hooks 的依赖数组发生变化时，会触发对应的逻辑执行，若逻辑中涉及状态更新，会进一步触发组件渲染。

2. 组件重新渲染时，render 函数内部发生的事情

render 函数（类组件）或函数组件的返回值部分，是组件渲染 UI 的核心逻辑，重新渲染时会执行以下操作：

- **执行 render 函数/函数组件主体**：
- 类组件：执行 `render` 方法，执行过程中会解析组件内部的 JSX 语法，同时计算组件内的变量、表达式（如 `{count + 1}`）。
- 函数组件：直接执行函数组件的主体代码，从函数开始到 return 语句结束，包括内部变量声明、Hooks 调用（如 `useState` 会获取最新状态）、逻辑计算等。

生成虚拟 DOM（VNode）树：render 函数/函数组件的返回值（JSX）会被 Babel 编译为 `React.createElement` 调用，执行该调用后生成虚拟 DOM 树。虚拟 DOM 是对真实 DOM 的轻量级描述，包含组件类型、属性（Props）、子节点等信息。

执行 Diff 算法对比新旧虚拟 DOM：React 会将本次生成的新虚拟 DOM 树与上一次渲染保存的旧虚拟 DOM 树进行对比（Diff 算法），找出两者的差异（如节点增删、属性变化、文本内容变化等）。

生成更新补丁并应用到真实 DOM：根据 Diff 算法的对比结果，React 会生成最小化的 DOM 更新补丁（只更新变化的部分），然后通过渲染层（如 react-dom）将这些补丁应用到真实 DOM 中，完成页面的更新。

注意事项：

- render 函数是「纯函数」（理想情况下），不应包含副作用操作（如修改状态、调用接口、操作 DOM），否则会导致渲染逻辑混乱或性能问题。
- 重新渲染时，若 Diff 算法发现新旧虚拟 DOM 无差异，React 会跳过真实 DOM 的更新操作，这是 React 的核心性能优化点之一。

八、避免 React 中不必要渲染的性能优化方案

不必要的渲染（如父组件渲染导致子组件无意义跟随渲染、状态未变却触发渲染）会浪费性能，尤其是复杂应用中，需通过以下方案优化：

1. 类组件优化方案

- **使用 React.PureComponent**：纯组件自动实现了 `shouldComponentUpdate` 方法，对 `props` 和 `state` 进行浅比较，只有当两者发生变化时才触发渲染。适用于 props 和 state 为基本类型或简单对象的组件（避免复杂嵌套对象，浅比较无法检测内部变化）。

- **手动重写 shouldComponentUpdate**：对于复杂场景（如嵌套对象的 props），可手动实现 `shouldComponentUpdate(nextProps, nextState)`，自定义对比逻辑（如深比较关键属性），返回 `false` 阻止不必要的渲染。
- **避免在 render 中创建新对象/新函数**：render 中创建的新对象（如 `{ name: 'xxx' }`）、新函数（如 `onClick={() => this.handleClick()}`）会导致每次渲染时 props 引用变化，即使内容相同，浅比较也会认为 props 变化，触发子组件渲染。解决方案：将对象/函数提升到组件实例属性（如 `this.obj = { name: 'xxx' }`）或使用箭头函数绑定在构造函数中。

2. 函数组件优化方案

- **使用 React.memo 包装组件**：`React.memo` 是高阶组件，用于函数组件的性能优化，类似类组件的 `PureComponent`，会对组件的 props 进行浅比较，只有 props 变化时才触发渲染。用法：`const MemoComponent = React.memo(FunctionComponent)`；若需自定义对比逻辑，可传入第二个参数（对比函数）。
- **使用 useMemo 缓存计算结果**：对于组件内的复杂计算逻辑（如大数据过滤、排序），每次渲染都会重新执行，浪费性能。`useMemo` 可缓存计算结果，仅当依赖数组中的值变化时才重新计算。用法：`const result = useMemo(() => computeFn(data), [data])`。
- **使用 useCallback 缓存函数引用**：函数组件中，每次渲染都会重新创建函数，导致传递给子组件的函数 props 引用变化，触发子组件不必要渲染。`useCallback` 可缓存函数引用，仅当依赖数组变化时才创建新函数。用法：`const handleClick = useCallback(() => { ... }, [deps])`。

3. 通用优化方案

- **拆分组件为更小的粒度**：将复杂组件拆分为多个独立的子组件，使得状态变化时，只有受影响的子组件重新渲染，而非整个复杂组件。例如：将列表项拆分为独立的 `ListItem` 组件，单个列表项状态变化时，仅该组件渲染。
- **合理使用 Context 或状态管理库**：避免将过多无关状态放入 Context 中，否则 Context 状态变化时，所有订阅该 Context 的组件都会渲染。可将 Context 拆分为多个细分 Context，或使用 Redux、Zustand 等状态管理库，实现状态的精准订阅（仅依赖该状态的组件才渲染）。
- **避免不必要的状态更新**：调用 `setState` 或 `useState` 更新函数时，确保传入的新状态与旧状态确实不同。例如：类组件中可先判断再更新（`if (newState !== this.state.count) this.setState({ count: newState })`）；函数组件中，`useState` 会自动忽略与旧状态相同的新状态（基本类型）。
- **使用懒加载和代码分割**：通过 `React.lazy` 和 `Suspense` 实现组件的懒加载，避免初始渲染时加载过多无关组件，减少初始渲染时间。用法：`const LazyComponent = React.lazy(() => import('./LazyComponent'))`，配合 `Suspense` 展示加载占位符。

九、setState 的调用原理及 React 内部执行流程

1. setState 的调用原理

setState 是 React 类组件中更新状态的核心方法，其核心原理是「触发组件的重新渲染」，但并非直接修改 `this.state`（直接修改不会触发渲染），而是通过 React 内部的状态更新机制，异步且批量地处理状态更新，最终完成组件渲染。

关键特性：setState 是「异步更新」（多数场景下）、「批量更新」、「可能被合并」，其原理依赖于 React 的调度系统和协调系统。

2. 调用 setState 后，React 内部发生的事情（详细流程）

1. **接收状态更新请求，加入更新队列：**
2. 调用 `setState(partialState, callback)` 时，React 首先会接收传入的 `partialState`（可以是对象或函数），并将其加入到当前组件的「更新队列」中（更新队列是存储组件所有待处理状态更新的队列）。
3. 若 `partialState` 是函数，React 会在后续处理时调用该函数，传入当前的 `state` 和 `props`，获取最终的新状态（便于依赖前一个状态计算新状态，如 `this.setState(prevState => ({ count: prevState.count + 1 }))`）。
4. **判断是否需要批量更新：**
5. React 会检查当前执行环境是否处于「批量更新上下文」中（如合成事件回调、生命周期方法、React 钩子函数内部）。若是，则将当前更新加入批量更新队列，等待所有批量更新收集完成后统一处理；若否（如原生事件、setTimeout、Promise 回调中），则直接触发后续的更新流程（React18 后通过自动批处理，大部分场景都支持批量更新）。
6. **触发调度器（Scheduler）调度更新：**
7. 批量更新收集完成后，React 会调用调度器（Scheduler）的 `scheduleCallback` 方法，将状态更新任务加入调度队列，并根据任务优先级排序（如用户输入是高优先级，普通状态更新是低优先级）。
8. 调度器负责时间切片（Time Slicing），确保高优先级任务先执行，避免长时间占用主线程导致页面卡顿。
9. **协调器（Reconciler）执行 Diff 算法，生成 Fiber 树：**
10. 调度器触发更新任务后，协调器（Reconciler）会启动工作，从组件的根节点开始，基于 Fiber 架构遍历组件树（可中断遍历）。
11. 在遍历过程中，协调器会合并更新队列中的所有状态更新，计算出组件的最终新状态，并对比新旧虚拟 DOM（Diff 算法），找出组件树中需要更新的部分，生成新的 Fiber 树（Fiber 树是 React16+ 用于描述组件更新的链表结构，记录组件的类型、状态、子节点等信息）。
12. **渲染器（Renderer）应用更新，更新真实 DOM：**

13. 协调器完成 Fiber 树的构建后，会将 Fiber 树交给渲染器（如 react-dom），渲染器会遍历新的 Fiber 树，根据 Fiber 节点的更新信息，生成最小化的 DOM 操作指令（如创建节点、修改属性、删除节点）。
14. 渲染器执行这些 DOM 操作指令，将更新应用到真实 DOM 中，完成页面的更新。
15. **执行 setState 的回调函数**：
16. 真实 DOM 更新完成后，React 会调用 `setState` 的第二个参数（回调函数），在该回调函数中可以获取到更新后的 `this.state`（因为此时状态和 DOM 都已完成更新）。

十、setState 调用是同步还是异步的?

setState 的调用「既不是绝对同步，也不是绝对异步」，而是「默认异步，特定场景下同步」，核心取决于调用 `setState` 时的「执行环境」，即是否处于 React 自身的批量更新上下文控制中。

1. 默认异步场景（大部分日常开发场景）

当 `setState` 调用处于 React 控制的执行环境中时，表现为异步更新，此时无法在 `setState` 调用后立即获取到新状态。

- **常见场景**：
- React 合成事件回调（如 `onClick`、`onChange`、`onSubmit` 等）。
- React 生命周期方法（如 `componentDidMount`、`componentDidUpdate`、`shouldComponentUpdate` 等，注意 `componentWillMount` 已废弃）。
- React 钩子函数内部（如 `useEffect`、`useLayoutEffect` 等）。

示例：

Code block

```
class Example extends React.Component {
  state = { count: 0 };
  handleClick = () => {
    this.setState({ count: this.state.count + 1 });
    console.log(this.state.count); // 输出 0（异步更新，此时状态未更新）
  };
  render() {
    return <button onClick={this.handleClick}>点击</button>;
  }
}
```

原因：React 在此类场景下会开启批量更新，将多个 `setState` 调用合并为一次更新，减少 DOM 操作次数，提升性能。因此，`setState` 不会立即修改 `this.state`，而是等待批量更新收集完成后统一处理。

2. 同步场景（非 React 控制的执行环境）

当 `setState` 调用处于非 React 控制的执行环境中时，表现为同步更新，此时可以在 `setState` 调用后立即获取到新状态。

- **常见场景**：
- 原生 DOM 事件回调（如 `addEventListener` 绑定的事件）。
- 定时器回调（如 `setTimeout`、`setInterval`）。
- Promise 回调（如 `Promise.then`、`async/await` 后续逻辑）。
- 手动调用 `setState` 时的同步代码块（如直接在控制台调用）。

示例：

Code block

```
class Example extends React.Component {
  state = { count: 0 };
  componentDidMount() {
    // 原生 DOM 事件
    document.getElementById('btn').addEventListener('click', () => {
      this.setState({ count: this.state.count + 1 });
      console.log(this.state.count); // 输出 1（同步更新，立即获取新状态）
    });
    // setTimeout 回调
    setTimeout(() => {
      this.setState({ count: this.state.count + 1 });
      console.log(this.state.count); // 输出 2（同步更新）
    }, 0);
  }
  render() {
    return <button id="btn">点击</button>;
  }
}
```

原因：此类场景下，代码执行脱离了 React 的批量更新上下文，React 无法将更新加入批量队列，因此会立即触发状态更新流程，`setState` 调用后状态和 DOM 会同步更新。

3. React18 中的变化：自动批处理（Automatic Batching）

React18 之前，只有 React 控制的场景（合成事件、生命周期）支持批量更新；React18 后通过「自动批处理」特性，将批量更新的范围扩展到了原生事件、setTimeout、Promise 等所有场景，即大部分原本同步的场景现在也支持异步批量更新，进一步优化性能。

若需在 React18 中强制同步获取新状态，可使用 `ReactDOM.flushSync` 包裹 `setState` 调用，示例：

Code block

```
import ReactDOM from 'react-dom';

class Example extends React.Component {
  state = { count: 0 };
  handleClick = () => {
    ReactDOM.flushSync(() => {
      this.setState({ count: this.state.count + 1 });
    });
    console.log(this.state.count); // 输出 1（强制同步更新）
  };
  render() {
    return <button onClick={this.handleClick}>点击</button>;
  }
}
```

总结

setState 的同步/异步取决于执行环境：

- React 控制的环境（合成事件、生命周期）：默认异步，批量更新。
- 非 React 控制的环境（原生事件、定时器、Promise）：React18 前同步，React18 后默认异步（自动批处理），可通过 `flushSync` 强制同步。
- 无论同步还是异步，都不建议依赖 `setState` 后的 `this.state` 做后续逻辑，如需使用更新后的状态，优先使用 `setState` 的回调函数（第二个参数）或 `componentDidUpdate` 生命周期方法。

十一、React 中 setState 的批量更新过程具体实现

setState 的批量更新是 React 优化性能的核心机制之一，其核心思想是「延迟执行状态更新，合并多个更新请求，最终一次性触发渲染」，具体实现依赖于 React 内部的「批量更新上下文」和「更新队列」管理，不同 React 版本（尤其是 React18）的实现范围有差异，具体过程如下：

1. 核心前提：批量更新上下文标识

React 内部通过一个全局标识（如 `isBatchingUpdates`）来标记当前是否处于「批量更新上下文」中。该标识由 React 控制，在特定场景下自动切换状态：

- 进入批量更新上下文时（如合成事件回调、生命周期方法执行前），将 `isBatchingUpdates` 设为 `true`。
- 退出批量更新上下文时（如合成事件回调、生命周期方法执行完成后），将 `isBatchingUpdates` 设为 `false`，并触发后续的更新合并与渲染流程。

2. 批量更新的具体执行步骤

1. **收集更新请求：加入组件更新队列**：
2. 当调用 `setState(partialState)` 时，React 首先检查 `isBatchingUpdates` 标识：若为 `true`（处于批量上下文），则不立即处理该更新，而是将 `partialState`（状态更新描述）加入当前组件的「更新队列」（每个组件实例都有一个独立的更新队列，存储待处理的状态更新）。
3. 若 `partialState` 是函数（如 `this.setState(prev => ({ count: prev.count + 1 }))`），则直接将函数存入更新队列，后续合并时再执行函数计算最终状态（确保依赖前一个状态的更新能正确叠加）。
4. **合并更新请求：去重与状态合并**：
5. 同一组件在批量更新上下文中多次调用 `setState` 时，更新队列会收集所有的 `partialState`，并在后续合并阶段进行「去重与合并」。
6. 合并规则：
7. 若多次更新的是同一个状态字段（如连续两次 `setState({ count: 1 })`），则仅保留最后一次更新（去重）。
8. 若更新的是不同状态字段（如 `setState({ count: 1 })` 后调用 `setState({ name: 'xxx' })`），则合并为一个完整的状态对象（如 `{ count: 1, name: 'xxx' }`）。
9. 若存在函数类型的 `partialState`，则按顺序执行函数，将前一个函数的返回值作为下一个函数的参数（确保状态更新的依赖链正确），最终得到合并后的新状态。
10. **触发批量更新：统一执行渲染流程**：
11. 当批量更新上下文结束（`isBatchingUpdates` 设为 `false`）时，React 会触发「批量刷新」逻辑（通过 `flushBatchedUpdates` 方法），遍历所有存在待更新队列的组件。
12. 对于每个组件，取出其更新队列中的合并后的新状态，执行后续的协调（Reconciler）和渲染（Renderer）流程：对比虚拟 DOM、生成 Diff 补丁、更新真实 DOM，完成一次完整的渲染。

3. React18 自动批处理（Automatic Batching）的扩展实现

React18 之前，批量更新仅支持 React 控制的场景（合成事件、生命周期）；React18 通过「自动批处理」扩展了批量更新的范围，其核心实现变化是：

- 将「批量更新上下文」从「全局标识」改为「基于根节点的上下文管理」（通过 `createRoot` 创建的根节点会维护自己的批量更新状态）。
- 在所有异步场景（原生事件、`setTimeout`、`Promise` 回调等）中，自动包裹批量更新逻辑，即进入这些场景时自动开启批量上下文，退出时关闭并刷新更新。
- 若需强制取消批量更新（立即执行更新），可使用 `ReactDOM.flushSync` 方法，该方法会强制将当前组件的更新队列立即刷新，触发同步渲染。

4. 批量更新的核心价值与注意事项

- 核心价值：减少 DOM 操作次数。若没有批量更新，多次 `setState` 会触发多次渲染，而批量更新将多次更新合并为一次渲染，大幅提升性能。
- 注意事项：在批量更新上下文中，无法立即获取 `this.state` 的最新值（因为状态尚未合并更新），如需获取最新状态，需使用 `setState` 的第二个回调参数（如 `this.setState({ count: 1 }, () => console.log(this.state.count))`）。

十二、React 的生命周期及各阶段方法

React 生命周期是组件从「创建」到「销毁」的完整过程，分为三个核心阶段：**挂载阶段（Mounting）**、**更新阶段（Updating）**、**卸载阶段（Unmounting）**。React16.3 后废弃了部分生命周期方法（如 `componentWillMount`），推荐使用更安全的替代方法，具体阶段与方法如下：

1. 挂载阶段（组件从无到有，首次渲染到 DOM）

核心目的：初始化组件状态、绑定事件、请求数据、创建 DOM 节点，执行顺序固定：

1. `constructor(props)`：组件构造函数，最先执行。用于初始化 `this.state`、绑定事件处理函数（如 `this.handleClick = this.handleClick.bind(this)`），注意不能在此处调用 `setState`（会触发不必要的渲染），也不能调用 `this.props`（需通过参数 `props` 访问）。
2. `static getDerivedStateFromProps(nextProps, prevState)`（React16.3+ 新增，替代 `componentWillReceiveProps`）：静态方法，无 `this` 指向。用于根据 `props` 变化推导状态变化，返回新状态对象（如 `return { count: nextProps.count }`）或 `null`（不更新状态），适用于「状态完全依赖 props」的场景。
3. `render()`：核心渲染方法，纯函数（无副作用）。返回 JSX 描述 UI 结构，被调用时会生成虚拟 DOM，不直接操作 DOM，可多次调用（如父组件重新渲染时）。
4. `componentDidMount()`：组件挂载完成（DOM 已生成并插入页面）后执行，仅执行一次。常用于：发送网络请求（如获取初始化数据）、绑定原生 DOM 事件（如 `addEventListener`）、初始化第三方库（如图表库、地图库），此处可安全调用 `setState`（会触发一次额外渲染，但不会影响首次挂载的性能）。

2. 更新阶段（组件状态/Props 变化，触发重新渲染）

核心目的：处理状态/Props 变化、优化渲染逻辑，执行顺序固定：

1. `static getDerivedStateFromProps(nextProps, prevState)`：与挂载阶段相同，在更新阶段也会执行，用于根据新 Props 推导状态变化。

2. `shouldComponentUpdate(nextProps, nextState)`：布尔值返回函数，用于控制组件是否需要重新渲染。默认返回 `true`（允许渲染），可手动重写对比 `nextProps` 与 `this.props`、`nextState` 与 `this.state`，返回 `false` 阻止渲染（优化性能）。`PureComponent` 会自动实现该方法（浅比较 Props/State）。
3. `render()`：重新执行，生成新的虚拟 DOM，与旧虚拟 DOM 对比（Diff 算法）。
4. `getSnapshotBeforeUpdate(prevProps, prevState)`（React16.3+ 新增，替代 `componentWillUpdate`）：在 DOM 更新前执行，用于捕获 DOM 更新前的快照信息（如滚动位置、DOM 尺寸），返回的快照值会作为 `componentDidUpdate` 的第三个参数。适用于需要保存 DOM 状态的场景（如滚动列表更新时保持滚动位置）。
5. `componentDidUpdate(prevProps, prevState, snapshot)`：组件更新完成（DOM 已更新）后执行。常用于：根据 Props/State 变化更新 DOM（如调整第三方库配置）、发送网络请求（仅当 Props 变化时），此处可调用 `setState`（但需加条件判断，避免无限循环更新）。

3. 卸载阶段（组件从 DOM 中移除，销毁）

核心目的：清理资源，避免内存泄漏，仅一个方法：

- `componentWillUnmount()`：组件卸载前执行，仅执行一次。常用于：解绑原生 DOM 事件（如 `removeEventListener`）、取消网络请求（如中止 `fetch` 或 `axios` 请求）、清理定时器（如 `clearTimeout`）、销毁第三方库实例，此处 **`setState` 不能调用**（组件已进入卸载流程，不会再渲染）。

4. 特殊生命周期方法（错误处理）

React16+ 新增错误边界相关方法，用于捕获子组件渲染错误，避免整个应用崩溃：

- `static getDerivedStateFromError(error)`：静态方法，捕获子组件错误后执行，返回新状态对象，用于渲染错误提示 UI（如 `return { hasError: true }`）。
- `componentDidCatch(error, errorInfo)`：捕获子组件错误后执行，可用于日志上报（如将错误信息发送到监控平台），`errorInfo` 包含错误堆栈信息。

十三、React 生命周期中的性能优化阶段及原理

React 性能优化的核心方向是「减少不必要的重新渲染」，最关键的优化阶段是**更新阶段**，通过拦截渲染触发流程实现优化；此外，挂载阶段也可通过提前准备数据减少冗余渲染，具体优化点及原理如下：

1. 核心优化阶段：更新阶段（拦截渲染触发）

更新阶段是组件最常执行的阶段，也是性能优化的重点，核心优化方法是重写 `shouldComponentUpdate` 或使用 `PureComponent`：

- **优化方法 1：使用 React.PureComponent**：

- 适用阶段：更新阶段（自动触发 `shouldComponentUpdate` 校验）。
- 优化原理：`PureComponent` 自动实现了 `shouldComponentUpdate` 方法，内部对 `nextProps` 与 `this.props`、`nextState` 与 `this.state` 进行「浅比较」：
- 基本类型（数字、字符串、布尔值）：直接比较值是否相同。
- 引用类型（对象、数组）：比较引用地址是否相同（不深入对比内部属性）。
- 若浅比较无变化，`shouldComponentUpdate` 返回 `false`，阻止组件重新渲染；若有变化则返回 `true`，触发渲染。
- 适用场景：组件的 Props/State 为基本类型或简单引用类型（无嵌套结构），且渲染仅依赖 Props/State。
- **优化方法 2：手动重写 shouldComponentUpdate**：
- 适用阶段：更新阶段。
- 优化原理：开发者根据组件的实际渲染依赖，自定义对比逻辑，仅在关键属性变化时返回 `true`，其他情况返回 `false`。例如，组件仅依赖 `props.id`，则仅对比 `nextProps.id` 与 `this.props.id`，忽略其他 Props 变化。
- 优势：适用于复杂场景（如嵌套对象的 Props），可通过「深比较关键属性」实现精准优化（注意：避免全量深比较，否则会抵消优化效果）。

2. 辅助优化阶段：挂载阶段与卸载阶段

- **挂载阶段优化：componentDidMount 中请求数据**：
- 优化原理：避免在 `render` 或 `constructor` 中请求数据（会导致多次请求或阻塞渲染），在 `componentDidMount` 中请求数据，此时组件已挂载完成，请求到数据后调用 `setState` 触发一次更新，仅渲染一次数据加载后的 UI，减少冗余渲染。
- **卸载阶段优化：componentWillUnmount 中清理资源**：
- 优化原理：若组件卸载前未清理资源（如定时器、原生事件监听、未完成的网络请求），这些资源会持续占用内存，导致内存泄漏，间接影响应用性能。在 `componentWillUnmount` 中清理这些资源，释放内存，保证应用运行流畅。

3. 其他生命周期相关优化：避免在 render 中创建新对象

适用阶段：更新阶段（`render` 方法执行时）。

- 优化原理：`render` 方法每次执行都会重新创建新对象/新函数（如 `onClick={() => this.handleClick()}`、`props.data = { name: 'xxx' }`），导致 `PureComponent` 或 `shouldComponentUpdate` 的浅比较认为 Props/State 变化，触发不必要的渲染。

- 解决方案：将对象/函数提升到组件实例属性（如 `this.handleClick = this.handleClick.bind(this)`、`this.data = { name: 'xxx' }`），避免在 `render` 中重复创建。

十四、React 的严格模式（Strict Mode）及作用

1. 什么是 React 严格模式

React 严格模式是一种「开发环境下的辅助工具」，通过 `<React.StrictMode>` 组件包裹子组件启用，仅在开发环境生效（生产环境自动禁用，无性能开销）。它不是一种语法或功能，而是用于检测组件中的潜在问题，帮助开发者编写更规范、更可靠的 React 代码。

使用方式（包裹需要检测的组件树）：

Code block

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App /> {/* 所有子组件都会被严格模式检测 */}
  </React.StrictMode>
);
```

2. 严格模式的核心作用

- **检测不安全的生命周期方法**：
- React16.3 后废弃了部分生命周期方法（如 `componentWillMount`、`componentWillReceiveProps`、`componentWillUpdate`），这些方法存在潜在风险（如重复渲染、状态不可预测）。
- 严格模式下，若组件使用了这些废弃方法，会在控制台输出警告，并提示替代方法（如用 `getDerivedStateFromProps` 替代 `componentWillReceiveProps`）。
- **检测意外的副作用**：
- React 期望 `render` 方法是纯函数（无副作用，如修改状态、操作 DOM、请求数据），但开发者可能会在 `render` 或其他生命周期方法中添加意外副作用。
- 严格模式下，会在开发环境中「双重调用」以下方法，以检测副作用：`render` 方法、`getDerivedStateFromProps` 静态方法、`setState` 的更新函数、`useState` / `useReducer` / `useMemo` 的回调函数。

- 双重调用的目的：若方法存在副作用（如修改全局变量），则会被重复执行，从而在开发环境中暴露问题（生产环境不会双重调用）。
- **检测过时的 API 使用**：
- 严格模式会检测组件中是否使用了 React 过时的 API（如 `ReactDOM.findDOMNode`、旧版上下文 API `childContextTypes`、`UNSAFE_` 前缀的生命周期方法）。
- 若使用了这些 API，会在控制台输出警告，提示使用最新的替代方案（如用 `ref` 替代 `findDOMNode`，用新版 `createContext` 替代旧版上下文 API）。
- **检测非严格模式下的并发渲染问题**：
- React18 默认启用并发渲染，部分非严格模式下正常的代码（如依赖同步渲染顺序的逻辑）在并发渲染下可能出现问题。
- 严格模式会模拟并发渲染的环境，检测这些潜在问题（如状态更新顺序错误、依赖渲染时机的副作用），帮助开发者提前适配并发渲染。
- **检测不安全的 ref 使用**：
- 严格模式会检测是否使用了过时的 `ref` 用法（如字符串 ref，如 `<div ref="myDiv" />`），并提示使用回调 ref 或 `useRef`（如 `<div ref={this.myDivRef} />`）。

3. 关键注意事项

- 严格模式仅作用于开发环境，生产环境不会执行任何检测逻辑，也不会影响应用性能。
- 严格模式不直接修复问题，仅通过警告提示开发者问题所在，需要开发者手动修改代码解决。
- 严格模式会包裹子组件树，所有子组件（包括嵌套组件）都会被检测，若需排除部分组件，可将其移出 `<React.StrictMode>` 包裹范围。

十五、React 中组件间通信的方式及适用场景

React 组件间通信根据「组件关系」（父子、兄弟、跨层级）的不同，有多种实现方式，核心思路是「数据传递」或「状态共享」，具体方式及适用场景如下：

1. 父子组件通信（最常见场景）

- **方式 1：父传子：Props 传递**：
- 实现逻辑：父组件通过在子组件标签上添加属性（Props）传递数据/方法，子组件通过 `this.props`（类组件）或参数（函数组件）接收。
- 适用场景：父组件向子组件传递静态数据、动态状态，或传递回调函数（让子组件触发父组件状态更新）。
- 示例：
- **方式 2：子传父：回调函数（Props 传递函数）**：

- 实现逻辑：父组件定义回调函数，通过 Props 传递给子组件；子组件触发事件时，调用该回调函数，并将需要传递的数据作为参数传入。
- 适用场景：子组件向父组件传递用户操作数据（如表单输入值、按钮点击状态）。
- **方式 3：父组件获取子组件实例/DOM：Refs 传递**：
- 实现逻辑：父组件通过 `useRef`（函数组件）或 `createRef`（类组件）创建 Ref 对象，传递给子组件；子组件通过 `ref` 属性接收，父组件可通过 Ref 对象直接访问子组件的实例（类组件）或 DOM 元素（函数组件需用 `forwardRef` 转发）。
- 适用场景：父组件需要直接操作子组件的 DOM（如获取输入框焦点）或调用子组件的方法（如类组件的实例方法）。
- 示例（函数组件转发 Ref）：

2. 兄弟组件通信

- **方式：通过共同父组件中转**：
- 实现逻辑：兄弟组件的通信需借助它们的共同父组件：兄组件通过回调函数将数据传递给父组件，父组件将数据通过 Props 传递给弟组件。
- 适用场景：兄弟组件之间需要共享数据或交互（如兄组件点击按钮，弟组件展示对应数据），且组件层级较浅。
- 示例：

3. 跨层级组件通信（祖孙、远亲组件）

- **方式 1：Context API（React 内置）**：
- 实现逻辑：通过 `React.createContext` 创建上下文对象，父组件通过 `Provider` 组件提供数据，所有子组件（无论层级多深）都可通过 `useContext`（函数组件）或 `static contextType`（类组件）订阅并获取上下文数据。
- 适用场景：跨层级组件共享少量全局数据（如用户登录状态、主题配置、语言设置），避免 Props 层层传递（Props Drilling）。
- 示例：
- **方式 2：状态管理库（复杂应用）**：
- 实现逻辑：使用专门的状态管理库（如 Redux、Zustand、Jotai）管理全局状态，所有组件可通过库提供的 API（如 Redux 的 `useSelector` / `useDispatch`）获取或修改全局状态，实现跨层级通信。
- 适用场景：大型应用、多个跨层级组件需要共享复杂状态（如电商购物车、用户权限），或需要状态回溯、异步状态管理的场景。

4. 其他通信方式（特殊场景）

- **方式 1：事件总线（Event Bus）：**
- 实现逻辑：基于发布-订阅模式（如使用 `events` 库或自定义事件总线），组件通过「发布事件」发送数据，其他组件通过「订阅事件」接收数据，无需依赖组件关系。
- 适用场景：非父子、非兄弟的零散组件之间的通信，且不适合使用 Context 或状态管理库的简单场景。
- 注意事项：需手动管理事件订阅与取消（避免内存泄漏），大型应用中易导致代码逻辑混乱，不推荐优先使用。
- **方式 2：LocalStorage/SessionStorage（持久化通信）：**
- 实现逻辑：组件通过 `localStorage.setItem` 存储数据，其他组件通过 `localStorage.getItem` 获取数据，利用浏览器本地存储实现跨组件通信。
- 适用场景：需要持久化存储数据的场景（如用户登录信息持久化、页面刷新后保留状态），但注意数据同步问题（需监听 `storage` 事件感知数据变化）。

5. 组件通信方式选择总结

组件关系	推荐通信方式	适用场景
父子组件	Props + 回调函数、Refs	简单数据传递、父操作子 DOM/方法
兄弟组件	共同父组件中转	层级较浅的兄弟组件交互
跨层级组件	Context API（简单场景）、状态管理库（复杂场景）	全局数据共享、避免 Props 层层传递
零散组件（无明确关系）	事件总线、LocalStorage	简单数据传递、持久化数据

十六、React-Router 的实现原理是什么？它是如何在 React 应用中管理路由的？

1. React-Router 核心实现原理

React-Router 是基于 React 生态的路由管理库，核心实现原理依赖于 **浏览器内置的路由 API** 和 **React 的组件化思想**，核心目标是实现“URL 与 UI 组件的映射关系”，让不同 URL 对应不同的页面组件，同时支持无刷新页面跳转。其核心原理可拆解为 3 点：

- **基于浏览器路由机制（两种模式）**：React-Router 提供两种路由模式，适配不同场景，底层依赖浏览器不同的 API 实现：

- **Hash 模式（默认模式）**：依赖浏览器 URL 中的 Hash （# 后面的部分），通过监听 `hashchange` 事件感知 Hash 变化（如 #/home → #/about）。Hash 变化不会触发浏览器向服务器发送请求，仅在客户端层面生效，适合无后端配置的静态页面。
- **History 模式**：依赖 HTML5 新增的 History API（`pushState`、`replaceState` 方法和 `popstate` 事件）。通过 `pushState` / `replaceState` 可以修改 URL 且不触发页面刷新，通过监听 `popstate` 事件感知浏览器前进/后退按钮的操作。该模式 URL 中无 #，更美观，但需要后端配置（所有路由都指向 index.html，避免刷新 404）。
- **组件化封装路由逻辑**：React-Router 将路由相关逻辑封装为可复用的 React 组件（如 `<BrowserRouter>`、`<Route>`、`<Link>`），开发者无需直接操作浏览器路由 API，通过组件组合即可实现路由管理。
- **上下文（Context）传递路由状态**：React-Router 内部通过 `createContext` 创建路由上下文（RouterContext），由最外层的 `<BrowserRouter>` 或 `<HashRouter>` 提供路由核心状态（如当前 URL、路由匹配规则、跳转方法等），内部组件（如 `<Route>`、`<Link>`、`useNavigate`）通过消费上下文获取路由状态和方法，实现跨组件路由通信。

2. React-Router 在应用中管理路由的核心流程

React-Router 通过“组件组合 + 路由匹配 + 状态同步”三个核心步骤管理路由，具体流程如下：

1. **1. 初始化路由根容器**：在应用入口处，用 `<BrowserRouter>`（History 模式）或 `<HashRouter>`（Hash 模式）包裹整个应用组件，该根组件的核心作用是：
2. 初始化路由模式（绑定对应的浏览器 API 事件，如 `hashchange` 或 `popstate`）；
3. 创建并提供路由上下文（RouterContext.Provider），将当前 URL、跳转方法（如 push、replace）等核心状态传递给内部组件。
4. **2. 定义路由规则（URL 与组件映射）**：通过 `<Route>` 组件定义路由规则，核心属性是 `path`（URL 路径）和 `element`（对应渲染的组件），示例：`<Route path="/home" element={<Home />} />`。
5. 同时可使用 `<Routes>`（React-Router v6+ 新增，替代旧版 `<Switch>`）包裹多个 `<Route>`，实现“排他性匹配”（即只渲染第一个匹配成功的 `<Route>`，避免多个路由同时匹配）。
6. **3. 实现页面跳转（无刷新导航）**：通过 `<Link>` 组件或编程式导航实现页面跳转，避免传统 `<a>` 标签跳转导致的页面刷新：
7. `<Link to="/about">关于我们</Link>`：内部通过调用路由上下文提供的 `push` 方法，底层调用 `pushState`（History 模式）或修改 `window.location.hash`（Hash 模式），实现无刷新 URL 变更。

8. 编程式导航：通过 `useNavigate` 钩子（函数组件）或 `withRouter` 高阶组件（类组件）获取 `navigate` 方法，手动触发跳转，如 `navigate('/detail', { state: { id: 1 } })`（可传递参数）。
9. **4. 路由匹配与 UI 渲染**：当 URL 发生变化时（如点击 `<Link>`、浏览器前进/后退），React-Router 会：
10. 监听 URL 变化事件（`hashchange` 或 `popstate`），更新路由上下文的“当前 URL”状态；
11. `<Routes>` 组件遍历内部所有 `<Route>`，对比 `Route.path` 与当前 URL，找到匹配成功的 `<Route>`；
12. 渲染匹配成功的 `<Route>.element` 组件，完成“URL 变化 → UI 更新”的映射。
13. **5. 路由参数与嵌套路由处理**：支持动态路由参数（如 `path="/detail/:id"`），通过 `useParams` 钩子获取参数（如 `const { id } = useParams()`）；支持嵌套路由（如 `<Route path="/user" element={<User>} ><Route path="profile" element={<Profile>} /></Route>`），通过 `<Outlet>` 组件渲染子路由对应的组件。

十七、对 Redux 的理解是什么？它主要解决了哪些问题？

1. 对 Redux 的核心理解

Redux 是一款基于“单向数据流”理念的**状态管理库**（非 React 专属，可配合任意框架使用，常与 React-Redux 结合用于 React 应用），核心作用是集中管理应用中多个组件共享的复杂状态，让状态变化可预测、可追踪。其核心设计遵循三大原则：

- **单一数据源**：整个应用的状态集中存储在一个“状态树”（store）中，避免多个组件分散存储状态导致的同步问题。
- **状态不可变**：状态（state）是只读的，不能直接修改，只能通过触发“动作”（action）生成新的状态，确保状态变化可追溯。
- **使用纯函数修改状态**：通过“减速器”（reducer）函数处理状态更新，reducer 是纯函数（无副作用、输入相同则输出相同），接收旧状态和 action，返回新状态，避免状态修改逻辑混乱。

简单来说，Redux 的核心思想是“将状态管理从组件内部抽离，交给独立的全局 store 管理，通过规范的流程修改状态”，让组件专注于 UI 渲染，而非状态管理逻辑。

2. Redux 主要解决的问题

Redux 主要解决 React 应用中“状态管理混乱”的问题，尤其是在中大型应用中，具体可分为以下 4 类：

- **解决跨组件状态共享问题**：在 React 中，父子组件可通过 Props 传递状态，但跨层级组件（如祖孙组件）、无直接关系的组件（如兄弟组件）共享状态时，需要通过“Props 层层传递”（即 Props Drilling），代码冗余且维护成本高。Redux 将共享状态集中在全局 store 中，任何组件都可直接获取或修改，无需依赖组件关系。

- **解决状态变化不可预测问题**：组件内部通过 `setState` 直接修改状态时，状态变化逻辑分散在各个组件中，尤其是多人协作时，容易出现“状态修改后不知道哪里触发的”“状态同步不及时”等问题。Redux 规定状态修改必须通过“action → reducer → newState”的规范流程，每一次状态变化都有明确的动作描述，且可通过 Redux DevTools 追踪状态变化历史，让状态变化可预测、可调试。
- **解决异步状态管理复杂问题**：React 组件中处理异步逻辑（如网络请求）时，异步结果修改状态的逻辑分散在组件的生命周期或事件回调中，容易导致“异步流程混乱”“多次请求竞态”等问题。Redux 配合中间件（如 redux-thunk、redux-saga）可将异步逻辑抽离到组件外部，通过规范的方式处理异步动作，让异步状态管理更清晰、可维护。
- **解决组件复用状态逻辑困难问题**：多个组件需要复用相同的状态管理逻辑（如用户登录状态、表单提交逻辑）时，传统方式需通过高阶组件（HOC）或 Render Props 封装，容易导致组件嵌套过深（回调地狱）。Redux 将状态逻辑抽离到 reducer 和 action 中，多个组件可通过订阅 store 或调用 action 复用相同逻辑，无需嵌套组件。

注意：Redux 并非所有应用都必需，小型应用（状态简单、组件通信少）使用 React 内置的 `useState`、`useContext` 即可满足需求；中大型应用（状态复杂、多组件共享状态多）使用 Redux 才能体现其价值。

十八、Redux 的工作原理及其流程是怎样的?

1. Redux 核心工作原理

Redux 的工作原理基于“单向数据流”和“发布-订阅模式”，核心是通过“store（状态容器）、action（动作描述）、reducer（状态处理器）”三个核心模块的协同工作，实现状态的集中管理和可预测更新：

- **store**：全局唯一的状态容器，存储整个应用的状态树（state），提供 `getState()`（获取状态）、`dispatch(action)`（触发状态更新）、`subscribe(listener)`（订阅状态变化）三个核心方法。
- **action**：状态变化的“动作描述”，是修改状态的唯一触发源，必须是纯对象，且必须包含 `type` 属性（描述动作类型，如 `type: 'USER_LOGIN'`），可选包含 `payload` 属性（传递状态更新所需的数据，如 `payload: { username: 'admin' }`）。
- **reducer**：纯函数，负责处理状态更新逻辑，接收两个参数：`prevState`（旧状态）和 `action`（动作描述），根据 action.type 判断需要执行的逻辑，返回新的状态（不能直接修改旧状态）。

核心逻辑：组件通过 `dispatch` 触发 action，action 被 reducer 接收并处理，reducer 返回新状态更新 store，store 通知所有订阅状态变化的组件，组件重新获取新状态并渲染 UI。

2. Redux 完整工作流程（同步场景）

以“用户点击登录按钮，更新登录状态”为例，同步场景下 Redux 工作流程如下：

1. **1. 初始化 store**：通过 Redux 的 `createStore(reducer)` 方法创建全局 store，传入 reducer 用于处理状态更新，store 初始化时会调用 reducer 一次，获取初始状态（reducer 接收 `undefined` 旧状态时，返回初始 state）。
2. **2. 组件订阅 store 状态**：组件通过 `store.subscribe(() => { this.setState(store.getState()) })`（类组件）或 React-Redux 的 `useSelector` 钩子（函数组件）订阅 store 状态，当 store 状态变化时，组件会自动获取新状态并重新渲染。
3. **3. 组件触发 action**：用户点击登录按钮，组件内部调用 `store.dispatch(action)` 触发状态更新，其中 action 是描述“登录动作”的对象，示例：`const loginAction = { type: 'USER_LOGIN', payload: { username: 'admin' } }`。
4. **4. reducer 处理 action 并返回新状态**：store 接收到 action 后，自动调用 reducer 函数，传入当前旧状态（`prevState`）和 action：
5. **5. store 更新状态并通知组件**：reducer 返回新状态后，store 会用新状态替换旧状态（`currentState = newState`），然后遍历执行所有通过 `subscribe` 订阅的监听函数，通知组件“状态已更新”。
6. **6. 组件重新渲染**：组件接收到 store 的状态更新通知后，通过 `store.getState()` 或 `useSelector` 获取新状态，触发组件重新渲染，展示更新后的 UI（如显示用户名、隐藏登录按钮）。

3. 异步场景补充（需中间件）

上述流程是同步场景，若需要处理异步逻辑（如登录前先发送网络请求验证账号密码），则需要在“组件触发 action”和“reducer 处理 action”之间加入 **Redux 中间件**（如 redux-thunk）。中间件的作用是“拦截 action”，先执行异步逻辑（如网络请求），待异步完成后，再触发真正的同步 action 交给 reducer 处理，流程变为：组件 dispatch 异步 action → 中间件拦截并执行异步逻辑 → 异步完成后 dispatch 同步 action → reducer 处理同步 action → 更新 store → 组件渲染。

十九、在 Redux 中，异步请求应如何处理？Redux 中间件是什么？如何编写 Redux 的中间件？

1. Redux 中处理异步请求的方式

Redux 原生（reducer 是纯函数）不支持异步逻辑，处理异步请求（如网络请求）必须借助 **Redux 中间件**，核心思路是“通过中间件拦截 action，在 action 到达 reducer 之前执行异步逻辑，异步完成后再触发同步 action 更新状态”。常见的 3 种处理方式：

- **方式 1：使用 redux-thunk（最常用，简单场景）**：
- 核心原理：允许 dispatch 的 action 是一个“函数”（而非纯对象），这个函数接收 `dispatch` 和 `getState` 作为参数，在函数内部执行异步逻辑，异步完成后通过 `dispatch` 触发同步 action。

- 示例（登录异步请求）：

方式 2：使用 redux-saga（复杂场景，支持并发控制）：

- 核心原理：通过“生成器函数（generator）”编写异步逻辑，将异步流程与组件解耦，支持更复杂的异步场景（如取消请求、请求竞态处理、定时任务）。
- 核心思路：监听特定的 action，当 action 被 dispatch 时，执行对应的生成器函数处理异步逻辑，异步完成后 dispatch 同步 action。

方式 3：使用 redux-observable（基于 RxJS，响应式编程）：

- 核心原理：基于 RxJS 库，将 action 视为“数据流”，通过响应式操作符（如 `switchMap`、`catchError`）处理异步逻辑，适合需要复杂数据流控制的场景（如实时数据推送）。

2. Redux 中间件的定义与作用

Redux 中间件是 Redux 生态中“扩展 dispatch 功能”的核心机制，本质是一个“函数”，位于“组件 dispatch action”和“reducer 处理 action”之间，对 action 进行拦截、处理或转换，实现额外的功能。

（1）核心作用

- 处理异步逻辑：如网络请求、定时器等（这是最常用的作用）；
- 日志打印：记录每一次 action 的触发、状态的变化，便于调试（如 redux-logger）；
- 状态校验：拦截 action，校验状态更新是否符合规范，避免非法状态修改；
- action 转换：将收到的 action 转换为其他 action，再传递给 reducer。

（2）工作位置

Redux 原始流程：组件 → dispatch(action) → reducer → store 更新 → 组件渲染；

加入中间件后流程：组件 → dispatch(action) → 中间件（拦截、处理 action） → reducer → store 更新 → 组件渲染。

3. 编写 Redux 中间件的步骤（自定义中间件）

Redux 中间件遵循“柯里化函数”的规范，结构为“三层函数嵌套”：

Code block

```
// 中间件核心结构（三层函数）
const customMiddleware = store => next => action => {
  // 中间件逻辑
  // 执行完成后，调用 next(action) 将 action 传递给下一个中间件或 reducer
  next(action);
};
```

各层函数的作用：

- 第一层函数（store）：接收 Redux 的 store 对象，可通过 `store.getState()` 获取当前状态，`store.dispatch()` 触发新的 action；
- 第二层函数（next）：接收“下一个中间件的 dispatch 方法”（若没有下一个中间件，则是原始的 reducer 处理函数），调用 `next(action)` 可将 action 传递下去；
- 第三层函数（action）：接收组件 dispatch 的 action，在此层编写核心处理逻辑（如异步、日志、校验等）。

（1）自定义中间件示例 1：日志中间件（记录 action 和状态变化）

Code block

```
const loggerMiddleware = store => next => action => {
  // 1. 记录触发的 action
  console.log('触发 action: ', action);
  // 2. 记录当前状态（action 处理前）
  console.log('处理前状态: ', store.getState());
  // 3. 将 action 传递给下一个中间件或 reducer
  const result = next(action);
  // 4. 记录处理后的状态
  console.log('处理后状态: ', store.getState());
  // 5. 返回结果（可选，便于后续中间件使用）
  return result;
};
```

（2）自定义中间件示例 2：异步中间件（简化版 redux-thunk）

Code block

```
const asyncMiddleware = store => next => action => {
  // 核心逻辑：如果 action 是函数，则执行函数（传入 dispatch 和 getState）；否则传递给下一个中间件
  if (typeof action === 'function') {
    return action(store.dispatch, store.getState());
  }
  // 非函数 action，正常传递
  return next(action);
};
```

（3）应用自定义中间件

通过 Redux 的 `applyMiddleware` 方法将中间件应用到 store，示例：

```
import { createStore, applyMiddleware } from 'redux';
import rootReducer from './reducers';
import loggerMiddleware from './middleware/loggerMiddleware';
import asyncMiddleware from './middleware/asyncMiddleware';

// 创建 store 时应用中间件
const store = createStore(
  rootReducer,
  applyMiddleware(asyncMiddleware, loggerMiddleware) // 中间件执行顺序：从左到右
);

export default store;
```

二十、Redux 状态管理器与全局对象来保存数据有什么不同?

Redux 状态管理器和“全局对象保存数据”（如 `window.globalState = { ... }`）都能实现“数据全局共享”，但两者在“状态管理规范性、可维护性、调试能力、生态支持”等方面差异巨大，Redux 本质是“规范化的全局状态管理方案”，而全局对象是“简单的全局数据存储”，具体差异如下：

对比维度	Redux 状态管理器	全局对象（如 window.globalState）
状态修改规范	有严格的修改规范：必须通过“dispatch(action) → reducer → 新状态”的流程修改，不允许直接修改状态（状态不可变），确保状态变化可预测。	无任何规范：可在任意位置直接修改（如 `window.globalState.user = 'admin'`），状态修改逻辑分散，容易出现“不知道哪里修改了状态”的问题，维护成本高。
状态变化追踪	支持状态变化追踪：配合 Redux DevTools 可查看每一次状态变化的历史（包括触发的 action、旧状态、新状态），支持“时间旅行”（回退到任意历史状态），调试效率极高。	无状态追踪能力：修改状态后无法追溯修改来源、修改时间，调试时需要手动打印日志，排查问题困难。
组件联动更新	自动联动更新：组件通过 `subscribe` 或 `useSelector` 订阅状态变化，当 store 状态更新时，所	无自动联动更新：修改全局对象后，依赖该数据的组件不会自动重新渲染，需要手动调用组件的更新方法（如 `setState`），容易出

	有订阅的组件会自动重新渲染，确保 UI 与状态同步。	现“状态更新但 UI 未同步”的问题。
异步逻辑支持	原生支持异步逻辑：通过中间件（redux-thunk、redux-saga 等）将异步逻辑抽离到组件外部，规范异步流程，避免异步逻辑分散在组件中导致的混乱。	无异步逻辑支持：异步逻辑（如网络请求）需要在组件内部处理，处理完成后手动修改全局对象，异步流程分散，容易出现竞态问题（如多次请求返回顺序不一致）。
生态与工具支持	生态完善：有配套的工具（Redux DevTools）、中间件（处理异步、日志）、React 适配库（React-Redux），支持代码分割、状态持久化（redux-persist）等高级功能。	无任何生态支持：所有功能（如状态持久化、异步处理）都需要手动实现，代码冗余且易出错。
适用场景	中大型应用：状态复杂、多组件共享状态多、需要调试和维护的场景。	小型简单场景：仅需存储少量静态数据（如应用版本号）、无频繁修改、无需组件联动更新的场景。
数据安全性	安全性高：状态不可直接修改，只能通过规范流程修改，避免非法修改导致的状态异常。	安全性低：全局对象可在任意位置被修改，容易被误操作或恶意修改，导致应用崩溃。

总结

全局对象是“简单粗暴的全局数据存储方案”，仅适用于极简单的场景；而 Redux 是“规范化、可维护、可调试的全局状态管理方案”，通过严格的流程和丰富的生态，解决了中大型应用中状态管理的混乱问题。两者的核心差异在于“是否有规范的状态修改流程”和“是否支持组件联动更新与调试追踪”。

二十一、Redux 与 MobX 有什么区别？如何选择它们？

1. Redux 与 MobX 的核心区别

Redux 和 MobX 均是前端常用的状态管理库，但设计理念截然不同：Redux 基于“单向数据流”和“不可变数据”，强调状态变化的可预测性；MobX 基于“响应式编程”和“可变数据”，强调开发效率和直观性，具体差异如下：

对比维度	Redux	MobX

核心设计理念	单向数据流，状态不可变（Immutable），通过纯函数修改状态	响应式编程，状态可变（Mutable），直接修改状态触发响应
状态存储	单一全局状态树（Single Store），所有状态集中管理	支持多Store，可按模块拆分状态，分散管理
状态修改方式	必须通过 dispatch(action) → reducer → 新状态的规范流程，不可直接修改	可直接修改状态（如 this.state.count++），MobX 自动追踪依赖并更新UI
依赖追踪	无自动依赖追踪，组件需手动订阅状态（如 useSelector），状态变化时可能触发多余渲染（需手动优化）	自动依赖追踪，仅渲染依赖状态变化的组件，无需手动优化，性能更优（开箱即用）
调试能力	强，配合 Redux DevTools 可实现状态变化历史追踪、时间旅行（回退历史状态），调试逻辑清晰	较弱，状态变化是隐式的，难以追溯修改来源，调试依赖 MobX DevTools，体验不如 Redux
代码量与开发效率	模板代码多（action、reducer 等），开发门槛高，初期效率低，但长期维护性好	代码简洁，无需模板代码，开发门槛低，直观高效，适合快速迭代项目
生态与适配性	生态庞大完善，适配 React 为主，也可用于其他框架，中间件丰富（处理异步、日志等）	生态较轻，适配 React、Vue 等多框架，核心依赖较少，集成简单
学习成本	高，需理解单向数据流、不可变数据、纯函数、中间件等概念	低，贴近原生 JavaScript 开发思维，核心是响应式监听，易上手

2. Redux 与 MobX 的选择依据

选择核心取决于“项目规模、团队熟悉度、开发效率需求、调试维护需求”，具体建议如下：

- **优先选择 Redux 的场景**：
- 中大型复杂应用：状态逻辑复杂、多组件共享状态多，需要严格的状态管理规范保证可维护性；
- 团队协作场景：多人协作开发，需要统一的状态修改流程和清晰的调试能力，避免逻辑混乱；
- 需要状态追溯需求：如电商订单流程、表单多步骤提交等，需记录状态变化历史，支持回退操作；
- 团队熟悉函数式编程：Redux 基于函数式思想，团队对纯函数、不可变数据等概念接受度高。

优先选择 MobX 的场景：

- 中小型快速迭代项目：需要快速开发上线，追求开发效率，无需复杂的状态管理规范；
- 状态逻辑简单直观：状态修改场景少，无需严格的流程约束，希望贴近原生开发体验；
- 性能优化需求高且不想手动处理：MobX 自动依赖追踪，可精准更新组件，减少不必要渲染；
- 团队不熟悉函数式编程：MobX 思维更贴近命令式编程，学习成本低，团队上手快。

特殊说明：

- 小型应用（状态简单）：无需使用 Redux 或 MobX，React 内置的 useState、useContext 即可满足需求；
- 混合场景：部分复杂模块可用 Redux 管理，简单模块可用 MobX 或 React 内置状态，灵活搭配（需注意项目一致性）。

二十二、Redux 与 Vuex 有何区别？它们共有的设计思想是什么？

1. Redux 与 Vuex 的核心区别

Redux 和 Vuex 均是基于“单向数据流”的状态管理库，核心目标是解决复杂应用的状态共享问题，但两者因适配框架（Redux 适配 React，Vuex 适配 Vue）的差异，在 API 设计、响应式实现、使用体验上有明显区别：

对比维度	Redux	Vuex
适配框架	通用（以 React 为主），需通过 React-Redux 适配 React 组件	专属 Vue 框架，深度集成 Vue 响应式系统，无需额外适配
核心 API 设计	核心模块：store（通过 createStore 创建）、action、reducer；API 简洁（getState、dispatch、subscribe）	核心模块：state、mutations、actions、getters、modules；API 更细分（commit、dispatch、mapState 等）
状态修改流程	组件 → dispatch(action) → reducer（纯函数） → 新状态 → store 更新 → 组件渲染	组件 → dispatch(action)（异步逻辑） → commit(mutation)（同步修改） → state 更新 → 组件渲染
状态修改限制	状态不可变（Immutable），reducer 必须是纯函数，不可直接修改旧状态，需返回新状态	状态可变（Mutable），但必须通过 mutation 同步修改（强制约束），action 仅处理异步逻辑
响应式实现		

	无原生响应式，组件需手动订阅状态变化（如 useSelector），状态更新后触发组件重新渲染	基于 Vue 响应式系统（Vue2：Object.defineProperty；Vue3：Proxy），state 是响应式数据，修改后自动触发组件更新
模块化支持	原生不支持模块化，需通过 combineReducers 合并多个 reducer 实现模块化拆分	原生支持 modules 模块，可按业务拆分状态（如 user 模块、cart 模块），支持命名空间（namespaced）避免冲突
异步逻辑处理	原生不支持异步，必须借助中间件（redux-thunk、redux-saga 等）处理异步逻辑	原生支持异步，通过 action 处理异步逻辑（如网络请求），异步完成后通过 commit 触发 mutation
组件集成方式	React 组件通过 useSelector（函数组件）或 connect 高阶组件获取状态，通过 dispatch 触发 action	Vue 组件通过 mapState、mapGetters 等辅助函数映射状态到组件，通过 mapActions 映射 action 到组件方法

2. Redux 与 Vuex 共有的设计思想

尽管 Redux 和 Vuex 存在诸多差异，但核心设计思想高度一致，均源于“单向数据流”和“集中式状态管理”的理念，具体共有思想如下：

- **集中式状态管理**：将应用中所有组件共享的状态集中存储在一个全局容器（store）中，避免状态分散在各个组件中导致的同步问题，实现状态的统一管理和维护。
- **单向数据流**：状态流转遵循“单向”规则，确保状态变化可预测。核心流程均为“组件触发操作 → 状态容器处理逻辑 → 状态更新 → 组件渲染”，禁止组件直接修改全局状态，避免状态变化混乱。
- **分离状态修改逻辑**：将状态修改逻辑与组件 UI 逻辑分离，组件仅负责触发操作（如点击按钮触发 action）和渲染 UI，状态修改逻辑（如异步处理、状态计算）由 store 中的专门模块（reducer/mutation、action）处理，降低组件复杂度。
- **状态可追踪**：通过规范的状态修改流程，使得每一次状态变化都有明确的触发源头（如 action 描述），便于调试和问题排查。两者均支持调试工具（Redux DevTools、Vue DevTools），可查看状态变化历史。
- **组件解耦**：通过全局 store 实现组件间的间接通信，无需依赖组件层级关系（如父子、跨层级），减少组件间的直接耦合，提升组件的复用性和独立性。

二十三、什么是 React Hooks？为什么 React 团队引入了 Hooks？

1. 什么是 React Hooks？

React Hooks 是 React 16.8 版本引入的**新特性**，本质是“可在函数组件中使用的特殊函数”，用于让函数组件拥有类组件的核心能力（如状态管理、生命周期、上下文访问等），同时避免类组件的复杂问题（如 this 指向、生命周期嵌套、HOC 嵌套地狱等）。

核心特点：

- 只能在函数组件的顶层调用（不可在条件判断、循环、嵌套函数中调用），确保 Hooks 执行顺序稳定；
- 只能在 React 函数组件或自定义 Hooks 中使用，不能在普通 JavaScript 函数中使用；
- 通过“钩子”的方式将 React 内部能力“注入”到函数组件中，无需编写类组件即可实现复杂功能。

常用内置 Hooks：

- useState：用于在函数组件中添加状态（state）；
- useEffect：用于处理函数组件的副作用（如网络请求、DOM 操作、定时器），替代类组件的生命周期；
- useContext：用于在函数组件中访问 React Context，实现跨组件通信；
- useReducer：用于复杂状态管理，替代 useState，类似 Redux 的 reducer 逻辑；
- useRef：用于获取 DOM 元素或存储持久化数据（组件重新渲染时数据不丢失）；
- useMemo/useCallback：用于性能优化，缓存计算结果或函数引用。

2. React 团队引入 Hooks 的原因

React 团队引入 Hooks 的核心目标是“解决类组件的固有问题，优化函数组件的开发体验和代码质量”，具体原因可分为以下 4 点：

- **解决类组件 this 指向混乱问题**：类组件中 this 指向动态变化（如事件回调中 this 可能为 undefined），需要通过 bind 绑定或箭头函数等方式规避，增加了代码冗余和学习成本。Hooks 基于函数组件，无需关注 this 指向，代码更简洁直观。
- **解决生命周期逻辑分散与冗余问题**：类组件的生命周期方法（如 componentDidMount、componentDidUpdate、componentWillUnmount）常常包含多个不相关的逻辑（如网络请求、定时器绑定、事件监听），导致代码臃肿且难以维护；同时，相关逻辑可能分散在不同生命周期中（如定时器绑定在 componentDidMount，清除在 componentWillUnmount），增加了逻辑关联的理解成本。Hooks 中的 useEffect 可将相关逻辑聚合在一起，通过依赖数组控制执行时机，替代多个生命周期方法。

- **解决组件复用逻辑复杂问题**：类组件中复用逻辑的方案（如高阶组件 HOC、Render Props）存在明显缺陷：HOC 会导致组件嵌套过深（地狱嵌套），增加调试难度；Render Props 会导致代码嵌套复杂，可读性差。Hooks 允许将复用逻辑抽取为“自定义 Hooks”，函数组件可直接调用自定义 Hooks 复用逻辑，无需组件嵌套，代码更简洁、复用性更强。
- **简化状态管理与逻辑组织**：类组件的状态管理需要通过 state 和 setState 实现，复杂状态逻辑（如依赖多个子状态的计算）难以拆分；而 Hooks 中的 useState 可实现状态的细粒度拆分，useReducer 可处理复杂状态逻辑，让状态管理更灵活。同时，Hooks 让函数组件成为 React 开发的主流选择，函数式编程的思想（纯函数、无副作用）更符合 React 的设计理念，便于代码的测试和维护。

补充：Hooks 并非要完全替代类组件，而是提供了更优的函数组件开发方案。React 至今仍支持类组件，但官方推荐新项目优先使用 Hooks 开发。

二十四、useState 和类组件的 state 有什么区别?

useState 是 Hooks 中用于函数组件的状态管理方案，类组件的 state 是传统的状态管理方案，两者核心目标都是“管理组件状态”，但在**使用方式、状态更新逻辑、数据结构、生命周期关联、性能优化**等方面存在显著差异：

对比维度	useState（函数组件）	类组件的 state
使用方式	通过数组解构获取状态和更新函数：`const [count, setCount] = useState(0)`；每个状态独立声明，可拆分多个 useState。	通过构造函数初始化或类属性声明：`state = { count: 0, name: 'xxx' }`；所有状态集中在一个对象中。
状态更新方式	调用更新函数（如 setCount）更新状态，支持两种方式：传入新值（`setCount(1)`）或传入函数（`setCount(prev => prev + 1)`）；更新函数是异步的。	调用 setState 方法更新状态，支持两种方式：传入对象（`this.setState({ count: 1 })`）或传入函数（`this.setState(prev => ({ count: prev.count + 1 }))`）；setState 是异步的（批量更新）。
状态合并规则	无自动合并：useState 是独立状态，更新一个状态不会影响其他状态；若需合并多个相关状态，需手动处理（或使用 useReducer）。	自动浅合并：setState 传入的对象会与原有 state 进行浅合并，仅更新指定的状态字段，其他字段保留原值（如 `this.setState({`

		`count: 1 })` 不会影响 state.name）。
状态数据结构	支持任意数据类型（基本类型、对象、数组等），但推荐将复杂状态拆分为多个独立的 useState（如 count、name 分别声明），便于管理。	状态是一个对象，所有状态字段都包含在该对象中，复杂状态需嵌套在对象中（如 `state = { user: { name: 'xxx', age: 18 } }`）。
生命周期关联	无直接生命周期方法，通过 useEffect 模拟生命周期逻辑（如 componentDidMount：`useEffect(() => { ... }, [])`；componentDidUpdate：`useEffect(() => { ... }, [count])`）。	拥有明确的生命周期方法（componentDidMount、componentDidUpdate、componentWillUnmount 等），状态更新逻辑可直接写在对应生命周期中。
状态更新后获取新状态	更新函数是异步的，无法在调用后立即获取新状态；需通过更新函数的函数形式（`setCount(prev => { ... })`）获取前一个状态，或在 useEffect 中监听状态变化获取新状态。	setState 是异步的，无法在调用后立即获取新状态；可通过 setState 的第二个参数（回调函数）获取新状态（`this.setState({ count: 1 }, () => console.log(this.state.count))`），或在 componentDidUpdate 中获取。
性能优化方式	通过 useMemo 缓存计算结果，useCallback 缓存函数引用，避免不必要的重新渲染；复杂状态可使用 useReducer 优化状态更新逻辑。	通过 PureComponent 或重写 shouldComponentUpdate 方法优化渲染；避免在 render 中创建新对象/函数，防止 props 引用变化导致的多余渲染。
this 指向问题	无 this 指向问题，函数组件中无需关注 this，代码更简洁。	存在 this 指向问题，事件回调中 this 可能为 undefined，需通过 bind 绑定或箭头函数规避（如 `this.handleClick = this.handleClick.bind(`

		`this)` 或 `handleClick = () => { ... }`）。

二十五、自定义 Hook 是什么，如何创建一个自定义 Hook?

1. 什么是自定义 Hook?

自定义 Hook 是“基于 React 内置 Hooks 封装的、可复用的函数”，命名以 `use` 开头（如 useUser、useRequest），用于抽取多个组件共有的逻辑（如状态管理、副作用处理、数据请求等），让函数组件能够轻松复用这些逻辑，同时避免类组件中 HOC 或 Render Props 导致的嵌套问题。

核心特点：

- 命名必须以 `use` 开头，这是 React 识别 Hook 的约定，确保 ESLint 等工具能够检测 Hook 的使用规范；
- 内部可调用 React 内置 Hooks（如 useState、useEffect、useContext 等），但需遵循 Hook 的使用规则（如只能在顶层调用）；
- 可接收参数，根据参数动态调整逻辑（如请求 Hook 可接收请求地址、请求参数）；
- 可返回任意数据（状态、函数、对象等），供调用的组件使用；
- 没有组件结构，仅封装逻辑，调用时与普通函数一样，无需嵌套组件，代码简洁。

核心价值：实现逻辑复用，减少代码冗余，提升组件的简洁性和可维护性。

2. 如何创建一个自定义 Hook?

创建自定义 Hook 的核心步骤是“抽取复用逻辑 → 用函数封装 → 内部调用内置 Hooks → 返回组件所需数据/函数”，具体步骤及示例如下：

（1）创建步骤

1. **识别复用逻辑**：分析多个组件中重复的逻辑（如数据请求、状态管理、副作用处理），确定需要抽取的代码块；
2. **定义 Hook 函数**：创建一个以 `use` 开头的函数，作为自定义 Hook 的载体；
3. **封装逻辑与内置 Hooks**：在函数内部调用 React 内置 Hooks（如 useState 管理状态、useEffect 处理副作用），实现复用逻辑；
4. **设计参数（可选）**：根据逻辑的灵活性需求，为 Hook 设计参数（如请求地址、配置项），让 Hook 适配不同场景；
5. **返回组件所需数据**：将组件需要的状态、操作函数等作为返回值（可返回单个值、数组、对象），供组件使用；
6. **使用自定义 Hook**：在函数组件中直接调用自定义 Hook，获取返回值并使用。

（2）示例 1：封装数据请求的自定义 Hook（useRequest）

场景：多个组件需要发送网络请求，且需处理“加载中、成功、失败”状态，抽取该逻辑为自定义 Hook。

Code block

```

// 1. 定义自定义 Hook：useRequest.js
import { useState, useEffect } from 'react';

// 接收请求函数和依赖数组作为参数
function useRequest(requestFn, dependencies = []) {
  // 2. 用内置 Hooks 管理状态
  const [data, setData] = useState(null); // 存储请求结果
  const [loading, setLoading] = useState(false); // 加载状态
  const [error, setError] = useState(null); // 错误状态

  // 3. 用 useEffect 处理副作用（发送请求）
  const fetchData = async () => {
    setLoading(true);
    setError(null); // 重置错误状态
    try {
      const result = await requestFn(); // 执行传入的请求函数
      setData(result);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false); // 无论成功失败，结束加载
    }
  };

  // 组件挂载时执行一次，依赖变化时重新执行
  useEffect(() => {
    fetchData();
  }, dependencies);

  // 4. 返回组件所需的数据和函数
  return { data, loading, error, refetch: fetchData };
}

export default useRequest;
```

使用自定义 Hook 的组件：

Code block

```
// 2. 在函数组件中使用 useRequest
import React from 'react';
import useRequest from './useRequest';

// 定义请求函数（获取用户列表）
const fetchUserList = async () => {
  const response = await fetch('https://api.example.com/users');
  return response.json();
};

function UserList() {
  // 调用自定义 Hook，传入请求函数和依赖数组
  const { data: userList, loading, error, refetch } =
useRequest(fetchUserList, []);

  if (loading) return <div>加载中...</div>;
  if (error) return <div>请求失败：{error}</div>;

  return (
    <div>
      <h3>用户列表</h3>
      <ul>
        {userList?.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
      <button onClick={refetch}>刷新列表</button>
    </div>
  );
}

export default UserList;
```

（3）示例 2：封装表单状态管理的自定义 Hook（useForm）

场景：多个表单组件需要处理“输入值绑定、表单重置、表单验证”逻辑，抽取为自定义 Hook。

Code block

```
// 1. 定义自定义 Hook：useForm.js
import { useState } from 'react';

// 接收初始表单值作为参数
function useForm(initialValues) {
  // 管理表单状态
```

```
  const [values, setValues] = useState(initialValues);
  // 管理错误信息
  const [errors, setErrors] = useState({});

  // 处理输入变化（绑定到 input 的 onChange）
  const handleChange = (e) => {
    const { name, value } = e.target;
    setValues(prev => ({ ...prev, [name]: value }));
  };

  // 表单验证逻辑
  const validate = () => {
    const newErrors = {};
    // 示例：验证用户名和密码不为空
    if (!values.username) newErrors.username = '用户名不能为空';
    if (!values.password) newErrors.password = '密码不能为空';
    setErrors(newErrors);
    // 返回是否验证通过
    return Object.keys(newErrors).length === 0;
  };

  // 表单提交处理
  const handleSubmit = (onSubmit) => (e) => {
    e.preventDefault();
    const isValid = validate();
    if (isValid) {
      onSubmit(values); // 执行组件传入的提交逻辑
    }
  };

  // 表单重置
  const resetForm = () => {
    setValues(initialValues);
    setErrors({});
  };

  // 返回表单相关的状态和方法
  return { values, errors, handleChange, handleSubmit, resetForm };
}

export default useForm;
```

使用自定义 Hook 的表单组件：

Code block

```
```

```jsx
// 2. 在函数组件中使用 useForm
import React from 'react';
import useForm from './useForm';

function LoginForm() {
  // 调用自定义 Hook，传入初始表单值
  const { values, errors, handleChange, handleSubmit, resetForm } = useForm({
    username: '',
    password: ''
  });

  // 表单提交逻辑
  const onLogin = (formValues) => {
    console.log('提交登录信息：', formValues);
    // 此处可添加发送登录请求的逻辑
  };

  return (
    <form onSubmit={handleSubmit(onLogin)}>
      <div>
        <label>用户名：</label>
        <input
          type="text"
          name="username"
          value={values.username}
          onChange={handleChange}
        />
        {errors.username && <span style={{ color: 'red' }}>{errors.username}</span>}
      </div>
      <div>
        <label>密码：</label>
        <input
          type="password"
          name="password"
          value={values.password}
          onChange={handleChange}
        />
        {errors.password && <span style={{ color: 'red' }}>{errors.password}</span>}
      </div>
      <button type="submit">登录</button>
      <button type="button" onClick={resetForm} style={{ marginLeft: '10px' }}>重置</button>
    </form>
  );
}
```

```
46
47  export default LoginForm;
```

（4）创建自定义 Hook 的注意事项

- 严格遵循 Hook 使用规则：只能在函数组件顶层或其他自定义 Hook 中调用，不可在条件判断、循环、嵌套函数中调用；
- 命名必须以 `use` 开头，否则 React 无法识别为 Hook，ESLint 会报错；
- 自定义 Hook 是逻辑复用，不是组件复用：每个组件调用自定义 Hook 时，内部的内置 Hooks 都是独立的（如多个组件调用 useRequest，每个组件的 loading、data 状态互不影响）；
- 避免过度封装：仅抽取真正复用的逻辑，简单逻辑无需封装为自定义 Hook，否则会增加代码复杂度；
- 可组合多个自定义 Hook：一个自定义 Hook 内部可调用其他自定义 Hook，实现更复杂的逻辑（如 useRequest 内部可调用 useLocalStorage 实现请求结果缓存）。

二十六、useCallback 和 useMemo 的区别及其使用场景

1. 核心定义与本质区别

useCallback 和 useMemo 均是 React Hooks 中用于**性能优化**的工具，核心作用是“缓存”，避免组件重新渲染时重复创建不必要的数据/函数，从而减少子组件不必要的渲染。但两者缓存的目标不同，这是它们的核心区别：

- **useCallback**：缓存的是「函数引用」，返回一个记忆化的函数。当组件重新渲染时，若依赖项未变化，useCallback 返回的函数引用始终不变；若依赖项变化，才会创建新的函数并返回新引用。
- **useMemo**：缓存的是「函数执行的结果」，返回一个记忆化的值。当组件重新渲染时，若依赖项未变化，useMemo 直接返回缓存的结果，不会重新执行传入的函数；若依赖项变化，才会重新执行函数并缓存新结果。

补充：两者的语法结构相似，均接收两个参数（回调函数、依赖数组），但用途完全不同，需根据优化目标选择。

2. 语法对比

Code block

```
1
2  // useCallback：缓存函数引用
3  const memoizedCallback = useCallback(
4    () => {
5      // 函数逻辑
6      doSomething(a, b);
7    },
```

```
    [a, b] // 依赖数组：仅当 a 或 b 变化时，才会创建新的函数
  );

  // useMemo：缓存函数执行结果
  const memoizedValue = useMemo(
    () => {
      // 需缓存结果的函数逻辑（通常是复杂计算）
      return computeSomething(a, b);
    },
    [a, b] // 依赖数组：仅当 a 或 b 变化时，才会重新计算结果并缓存
  );
```

3. 关键区别总结

对比维度	useCallback	useMemo
缓存目标	函数引用（避免函数重复创建）	函数执行结果（避免重复计算）
返回值	记忆化的函数	记忆化的函数执行结果
适用场景	优化子组件渲染（避免因函数引用变化导致子组件不必要渲染）	优化复杂计算（避免组件重新渲染时重复执行耗时计算）
副作用注意	内部可包含副作用（但不推荐，副作用应放在 useEffect 中）	必须是纯函数（无副作用），因为执行时机不确定（可能被 React 暂停/恢复）

4. 使用场景详解

（1）useCallback 的使用场景

核心场景：当函数组件需要将函数作为 props 传递给子组件，且子组件通过 React.memo 优化（浅比较 props）时，用 useCallback 缓存函数引用，避免因父组件重新渲染导致函数引用变化，进而触发子组件不必要的重新渲染。

示例：

Code block

```

  // 子组件：通过 React.memo 优化，浅比较 props 变化才渲染
  const Child = React.memo(({ onClick, name }) => {
    console.log('子组件渲染');
    return <button onClick={onClick}>{name}</button>;
```

```
});

// 父组件
function Parent() {
  const [count, setCount] = useState(0);

  // 未使用 useCallback：父组件每次渲染都会创建新的 handleClick 函数，导致 Child 重新渲染
  // const handleClick = () => { console.log('点击'); };

  // 使用 useCallback：依赖项为空数组，仅初始化时创建一次函数，后续父组件渲染不会变化
  const handleClick = useCallback(() => {
    console.log('点击');
  }, []); // 依赖项：无依赖，函数引用永久不变

  return (
    <div>
      <p>count: {count}</p>
      <button onClick={() => setCount(count + 1)}>修改 count</button>
      <Child onClick={handleClick} name="测试按钮" />
    </div>
  );
}
```

说明：父组件中 count 变化时，父组件会重新渲染。若未使用 useCallback，handleClick 会重新创建，Child 组件的 onClick props 引用变化，即使其他 props 未变，也会触发 Child 重新渲染；使用 useCallback 后，handleClick 引用不变，Child 不会因该函数 props 变化而不必要渲染。

（2）useMemo 的使用场景

核心场景：组件内存在复杂计算（如大数据排序、过滤、多维数据处理），且计算结果仅依赖特定变量，用 useMemo 缓存计算结果，避免组件每次重新渲染时都重复执行耗时计算，提升性能。

示例：

Code block

```

function DataList() {
  const [list, setList] = useState([1, 3, 2, 5, 4]);
  const [search, setSearch] = useState('');

  // 复杂计算：过滤并排序列表（假设 list 数据量极大，计算耗时）
  // 未使用 useMemo：组件每次渲染（如 search 变化、父组件渲染）都会重新执行过滤排序
  // const filteredList = list.filter(item => item.toString().includes(search)).sort((a, b) => a - b);

```

```
    // 使用 useMemo：仅当 list 或 search 变化时，才重新执行过滤排序并缓存结果
    const filteredList = useMemo(() => {
      console.log('执行过滤排序');
      return list.filter(item => item.toString().includes(search)).sort((a, b) => a - b);
    }, [list, search]); // 依赖项：仅 list 或 search 变化时重新计算

    return (
      <div>
        <input
          type="text"
          value={search}
          onChange={(e) => setSearch(e.target.value)}
          placeholder="搜索..."
        />
        <ul>
          {filteredList.map(item => <li key={item}>{item}</li>)}
        </ul>
      </div>
    );
  }
```

说明：未使用 useMemo 时，每次输入框变化（search 变化）或父组件渲染，都会重新执行过滤排序；使用 useMemo 后，仅当 list 或 search 变化时才执行一次计算，后续渲染直接复用缓存结果，减少性能开销。

5. 注意事项

- 不要过度使用：useCallback 和 useMemo 本身也有性能开销（需要存储缓存、对比依赖项），对于简单函数/计算，直接创建/执行的成本更低，无需缓存。
- 依赖数组必须完整：若缓存的函数/计算依赖组件内的变量（如 state、props），必须将这些变量加入依赖数组，否则可能导致使用过期的值（闭包陷阱）。
- useMemo 不可用于副作用：useMemo 的回调函数必须是纯函数（无 DOM 操作、网络请求等副作用），因为 React 可能会根据性能需求，在不渲染组件的情况下执行该函数。

二十七、useEffect 与 useLayoutEffect 在 React 中的不同及其使用场景

1. 核心区别：执行时机

useEffect 和 useLayoutEffect 均是 React Hooks 中用于处理「副作用」的工具（如 DOM 操作、网络请求、定时器、事件监听等），核心差异在于**执行时机不同**，这直接决定了它们的适用场景和性能影响。

- **useEffect**：「异步执行」，在浏览器完成页面渲染（绘制 DOM）之后执行。React 会先更新 DOM 并绘制到页面，再异步调用 useEffect 的回调函数，不会阻塞浏览器的渲染流程。
- **useLayoutEffect**：「同步执行」，在 React 完成 DOM 更新但尚未将 DOM 绘制到页面之前执行。React 会先执行 useLayoutEffect 的回调函数，待回调完成后，再将更新后的 DOM 绘制到页面，会阻塞浏览器的渲染流程。

补充：两者的语法完全一致，均接收两个参数（副作用回调函数、依赖数组），但执行时机的差异导致了使用场景的严格区分。

2. 执行流程对比

（1）useEffect 的执行流程

1. 组件触发重新渲染（如状态变化、props 变化）；
2. React 计算并更新虚拟 DOM，生成真实 DOM 操作指令；
3. 浏览器执行 DOM 操作，将更新后的页面绘制到屏幕上；
4. 异步执行 useEffect 的回调函数（若依赖数组满足条件）；
5. 回调函数执行完成（若有清理函数，会在下次回调执行前执行）。

（2）useLayoutEffect 的执行流程

1. 组件触发重新渲染；
2. React 计算并更新虚拟 DOM，生成真实 DOM 操作指令；
3. 浏览器执行 DOM 操作（更新 DOM 树，但未绘制到屏幕）；
4. 同步执行 useLayoutEffect 的回调函数（若依赖数组满足条件）；
5. 回调函数执行完成（若有清理函数，优先执行）；
6. 浏览器将更新后的 DOM 绘制到屏幕上。

3. 关键区别总结

对比维度	useEffect	useLayoutEffect
执行时机	浏览器渲染（绘制）之后，异步执行	浏览器渲染（绘制）之前，同步执行
是否阻塞渲染	不阻塞，不会影响页面加载速度	阻塞，回调执行时间过长会导致页面卡顿
DOM 操作影响	若在回调中修改 DOM，可能导致页面闪烁（先绘制旧 DOM，再更新为新 DOM）	在绘制前修改 DOM，可避免页面闪烁（绘制的是最终 DOM 状态）

适用场景	绝大多数副作用场景（网络请求、定时器、事件监听、数据处理等）	需要在绘制前修改 DOM、获取 DOM 布局信息（如宽高、位置）并同步更新状态的场景
执行顺序	同一组件中，晚于 useLayoutEffect 执行	同一组件中，早于 useEffect 执行

4. 使用场景详解

（1）useEffect 的使用场景

useEffect 是处理副作用的「默认选择」，适用于不需要阻塞渲染、不依赖 DOM 布局信息的场景，常见案例：

- 网络请求（如组件挂载时获取数据）；
- 定时器/间隔器（如组件挂载时启动定时器，卸载时清除）；
- 事件监听（如监听窗口大小变化、滚动事件）；
- 数据持久化（如将状态同步到 localStorage）；
- 非紧急的 DOM 操作（如添加动画效果，允许轻微延迟）。

示例：组件挂载时获取用户数据

Code block

```
function UserProfile() {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // 网络请求（异步，不阻塞渲染）
    const fetchUser = async () => {
      try {
        const res = await fetch('https://api.example.com/user');
        const data = await res.json();
        setUser(data);
      } catch (err) {
        console.error('获取用户失败', err);
      } finally {
        setLoading(false);
      }
    };

    fetchUser();
  }, []); // 依赖数组为空，仅组件挂载时执行一次

```

```
    if (loading) return <div>加载中...</div>;
    return <div><h3>{user.name}</h3><p>{user.email}</p></div>;
  }
```

（2）useLayoutEffect 的使用场景

useLayoutEffect 仅适用于「必须在绘制前完成」的场景，核心是避免页面闪烁或获取准确的 DOM 布局信息，常见案例：

- 获取 DOM 布局信息（如宽高、位置）并同步更新状态（如根据 DOM 宽度调整组件样式）；
- 需要在绘制前修正 DOM 状态（如避免元素位置偏移、文字闪烁）；
- 与第三方 DOM 库集成（如某些图表库需要在 DOM 绘制前初始化）。

示例：根据元素宽度调整文字大小（避免闪烁）

Code block

```

  function ResizableText() {
    const [fontSize, setFontSize] = useState(16);
    const textRef = useRef(null);

    useLayoutEffect(() => {
      // 绘制前获取 DOM 宽度（准确，因为 DOM 已更新但未绘制）
      const width = textRef.current.offsetWidth;

      // 根据宽度调整字体大小（同步更新状态，绘制时直接使用新字体大小，无闪烁）
      if (width > 500) {
        setFontSize(24);
      } else if (width < 200) {
        setFontSize(12);
      } else {
        setFontSize(16);
      }
    }, []); // 组件挂载时执行一次

    return (
      <div ref={textRef} style={{ fontSize: `${fontSize}px` }}>
        这是一段根据容器宽度自适应字体大小的文字
      </div>
    );
  }
```

说明：若使用 useEffect，会先绘制默认字体大小的文字，再获取宽度调整字体大小，导致页面出现“字体闪烁”；使用 useLayoutEffect 可在绘制前完成调整，直接渲染最终字体大小，避免闪烁。

5. 注意事项

- 优先使用 useEffect：useLayoutEffect 会阻塞渲染，可能导致页面卡顿，非必要不使用；仅当 useEffect 无法满足需求（如出现闪烁、需要准确布局信息）时，才考虑 useLayoutEffect。
- 避免长时间任务：useLayoutEffect 的回调函数执行时间过长会严重阻塞页面渲染，应尽量简化逻辑，复杂计算需移至 useEffect 或 Web Worker 中。
- 服务器端渲染（SSR）注意：useLayoutEffect 在服务器端不会执行（因为服务器端无 DOM 绘制流程），若组件在 SSR 环境中使用，可能导致客户端与服务器端渲染结果不一致；此时应改用 useEffect，或添加客户端判断。

二十八、使用 React Hooks 时需要遵守哪些限制和规则?

React Hooks 的设计遵循严格的规则，这些规则是为了确保 Hooks 能够正确追踪组件状态、维护执行顺序的稳定性，避免出现不可预测的 bug。违反规则可能导致组件状态错乱、生命周期逻辑异常等问题，核心规则由 React 官方定义，同时包含一些实践层面的最佳规则。

1. 官方核心规则（必须严格遵守）

（1）只能在函数组件的顶层调用 Hooks

核心要求：Hooks 必须在函数组件的「最顶层作用域」调用，不能在条件判断（if/else）、循环（for/while）、嵌套函数（如函数内部定义的函数）中调用。

原因：React 是通过「Hooks 的调用顺序」来追踪状态的（如第一个 useState 对应第一个状态，第二个 useState 对应第二个状态）。若在条件/循环中调用 Hooks，会导致组件每次渲染时 Hooks 的调用顺序不一致，React 无法正确关联状态与 Hooks，从而引发状态错乱。

错误示例：

Code block

```

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

  // 错误：在条件判断中调用 useEffect
  if (count > 0) {
    useEffect(() => {
      console.log('count 大于 0');
    }, [count]);
  }

  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

正确示例：

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

  // 正确：顶层调用 useEffect，在内部判断条件
  useEffect(() => {
    if (count > 0) {
      console.log('count 大于 0');
    }
  }, [count]);

  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

（2）只能在 React 函数组件或自定义 Hooks 中调用 Hooks

核心要求：Hooks 不能在普通 JavaScript 函数中调用，只能在两种场景下使用：

- React 函数组件（定义为函数，返回 JSX 的组件）；
- 自定义 Hooks（命名以 use 开头，内部可调用其他 Hooks 的函数）。

原因：Hooks 依赖 React 的内部机制（如 Fiber 架构、状态追踪系统）来工作，普通 JavaScript 函数不具备这些机制，调用 Hooks 会导致 React 无法追踪状态，进而引发错误。

错误示例：

```
// 错误：在普通 JavaScript 函数中调用 useState
function normalJsFunction() {
  const [name, setName] = useState(''); // React 无法识别，报错
  return name;
}
```

正确示例：

```
// 正确：在自定义 Hooks 中调用 Hooks
function useName() {
  const [name, setName] = useState('');
```

```
  return [name, setName];
}

// 正确：在函数组件中调用 Hooks
function GoodComponent() {
  const [name, setName] = useName();
  return <input value={name} onChange={(e) => setName(e.target.value)} />;
}
```

2. 实践层面的最佳规则（推荐遵守）

（1）自定义 Hooks 必须以 use 开头命名

规则：自定义 Hooks 的名称必须以「use」作为前缀（如 useRequest、useForm、useUser）。

原因：这是 React 官方的约定，目的是让开发者快速识别 Hooks，同时让 ESLint 等工具（如 eslint-plugin-react-hooks）能够检测到 Hooks 的使用场景，避免违反核心规则（如在普通函数中调用）。

错误示例：const requestHook = () => { ... }（未以 use 开头，无法被识别为 Hooks）；

正确示例：const useRequest = () => { ... }。

（2）依赖数组必须完整且准确

规则：对于 useEffect、useCallback、useMemo 等需要依赖数组的 Hooks，必须将回调函数中使用的所有组件内变量（state、props、函数等）都加入依赖数组。

原因：依赖数组的作用是告诉 React“当哪些变量变化时，需要重新执行回调函数/重新创建函数/重新计算结果”。若依赖数组不完整，会导致回调函数使用过期的变量（闭包陷阱），出现状态错乱、逻辑异常等问题。

错误示例：

Code block

```

function BadComponent({ id }) {
  const [data, setData] = useState(null);

  // 错误：回调中使用了 id，但未加入依赖数组，id 变化时不会重新请求数据
  useEffect(() => {
    const fetchData = async () => {
      const res = await fetch(`https://api.example.com/data/${id}`);
      const data = await res.json();
      setData(data);
    };
    fetchData();
  }, []); // 遗漏依赖 id

```

```
    return <div>{data?.name}</div>;
}
```

正确示例：

Code block

```
function GoodComponent({ id }) {
  const [data, setData] = useState(null);

  // 正确：将 id 加入依赖数组，id 变化时重新请求数据
  useEffect(() => {
    const fetchData = async () => {
      const res = await fetch(`https://api.example.com/data/${id}`);
      const data = await res.json();
      setData(data);
    };
    fetchData();
  }, [id]); // 依赖数组包含所有用到的变量

  return <div>{data?.name}</div>;
}
```

（3）避免在 Hooks 回调中执行副作用（useMemo/useCallback）

规则：useMemo 和 useCallback 的回调函数应是「纯函数」，无副作用（如 DOM 操作、网络请求、修改状态等）；副作用应统一放在 useEffect 中处理。

原因：useMemo 和 useCallback 的核心作用是缓存，其回调函数的执行时机不确定（React 可能为了性能暂停/恢复执行），若包含副作用，可能导致副作用执行次数异常、逻辑错乱。

错误示例：

Code block

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

  // 错误：useMemo 回调中包含副作用（修改状态）
  useMemo(() => {
    setCount(count + 1); // 副作用，执行时机不确定
    return count * 2;
  }, [count]);

  return <div>{count}</div>;
```

```
}
```

正确示例：

Code block

```
function GoodComponent() {
  const [count, setCount] = useState(0);
  const doubledCount = useMemo(() => {
    return count * 2; // 纯函数，无副作用
  }, [count]);

  // 正确：副作用放在 useEffect 中
  useEffect(() => {
    setCount(prev => prev + 1);
  }, []);

  return <div>{doubledCount}</div>;
}
```

（4）避免过度使用 Hooks

规则：不追求“ Hooks 数量最少”，但也避免不必要的 Hooks 拆分或滥用（如将简单状态拆分为多个 useState，或过度使用 useMemo/useCallback 缓存）。

原因：合理的 Hooks 拆分能提升代码可读性（如将不同功能的状态分开管理），但过度拆分会增加代码复杂度；过度使用缓存类 Hooks（useMemo/useCallback）会增加 React 的性能开销（缓存存储、依赖对比），反而降低性能。

（5）清理 useEffect 中的副作用

规则：对于有“持续副作用”的 useEffect（如定时器、事件监听、网络请求），必须在回调函数中返回清理函数，用于组件卸载或依赖变化时清理副作用，避免内存泄漏。

示例：清理定时器

Code block

```
function GoodComponent() {
  useEffect(() => {
    const timer = setInterval(() => {
      console.log('定时器执行');
    }, 1000);

    // 清理函数：组件卸载或依赖变化时清除定时器
```

```
    return () => {
      clearInterval(timer);
    };
  }, []);

  return <div>定时器示例</div>;
}
```

3. 规则检测工具

为了避免违反 Hooks 规则，推荐使用 ESLint 插件 **eslint-plugin-react-hooks**，该插件会自动检测代码中 Hooks 的使用情况，对违反规则的代码给出警告/错误提示。

配置方式（在 .eslintrc.js 中）：

Code block

```
module.exports = {
  plugins: ['react-hooks'],
  rules: {
    // 检查 Hooks 的调用规则（顶层调用、仅在 React 函数中调用）
    'react-hooks/rules-of-hooks': 'error',
    // 检查依赖数组的完整性
    'react-hooks/exhaustive-deps': 'warn'
  }
};
```

二十九、请解释 React 如何追踪 Hook 的状态？Hooks 的原理是什么？

React Hooks 的核心原理是「通过链表结构追踪 Hooks 状态，结合 Fiber 架构维护组件状态与 Hooks 的关联」。React 并非通过“命名”或“变量名”追踪状态，而是通过「Hooks 的调用顺序」和「组件的 Fiber 节点」来精准关联每个 Hook 及其对应的状态，确保组件重新渲染时 Hooks 能正确读取/更新状态。

1. React 如何追踪 Hook 的状态?

React 追踪 Hook 状态的核心依赖两个关键机制：「Fiber 节点的 Hooks 链表」和「Hooks 调用顺序的稳定性」。

（1）Fiber 节点：存储组件状态的容器

React 16+ 采用 Fiber 架构，每个组件对应一个「Fiber 节点」（虚拟 DOM 的升级版本，用于描述组件的类型、状态、子节点等信息）。Fiber 节点中包含一个关键属性：`memoizedState`，用于存储组件的 Hooks 相关状态。

对于函数组件：`memoizedState` 指向一个「Hooks 链表」的头节点，每个 Hook（如 useState、useEffect）对应链表中的一个节点，节点中存储该 Hook 的状态数据、依赖项、清理函数等信息。

（2）Hooks 链表：按调用顺序串联状态

当函数组件首次渲染时，React 会按 Hooks 的调用顺序，创建对应的 Hook 节点，并将这些节点串联成一个单向链表，最终将链表头节点赋值给组件 Fiber 节点的 `memoizedState`。

每个 Hook 节点包含以下核心信息（以 useState 为例）：

- `memoizedState`：当前 Hook 的状态值（如 useState 的 count 值）；
- `queue`：状态更新队列，存储该 Hook 的待处理更新（如 setState 触发的更新）；
- `next`：指向链表中的下一个 Hook 节点（串联所有 Hooks）。

示例：首次渲染时的 Hooks 链表创建过程

Code block

```
// 函数组件
function MyComponent() {
  const [count, setCount] = useState(0); // Hook 1
  const [name, setName] = useState(''); // Hook 2
  useEffect(() => { ... }, [count]); // Hook 3
  return <div>...</div>;
}

// 首次渲染时，React 执行流程：
1. 初始化 Fiber 节点的 memoizedState 为 null；
2. 执行 useState(0)：创建 Hook1 节点（memoizedState=0），Fiber.memoizedState 指向 Hook1；
3. 执行 useState('')：创建 Hook2 节点（memoizedState=''），Hook1.next 指向 Hook2；
4. 执行 useEffect(...)：创建 Hook3 节点（存储依赖 [count]、回调函数等），Hook2.next 指向 Hook3；
5. 最终形成链表：Fiber.memoizedState → Hook1 → Hook2 → Hook3。
```

（3）重新渲染时：按顺序遍历链表读取状态

当组件因状态变化、props 变化等触发重新渲染时，React 会：

1. 获取组件 Fiber 节点的 `memoizedState`（即 Hooks 链表头节点）；
2. 按与首次渲染完全相同的顺序，遍历 Hooks 链表：
3. 执行第一个 useState 时，读取链表头节点（Hook1）的 `memoizedState`（count 值），并返回 [count, setCount]；

4. 执行第二个 useState 时，通过 Hook1.next 找到 Hook2 节点，读取其 `memoizedState`（name 值），返回 [name, setName]；
5. 执行 useEffect 时，找到 Hook3 节点，对比依赖项变化，决定是否执行回调函数；
6. 遍历完成后，确保所有 Hooks 都能正确读取到对应的状态。

关键结论：React 是通过「Hooks 的调用顺序」来匹配链表中的 Hook 节点，进而追踪状态的。这也是为什么 Hooks 必须在组件顶层调用（保证调用顺序稳定）的核心原因——若顺序变化，遍历链表时会匹配到错误的 Hook 节点，导致状态错乱。

2. Hooks 的核心原理（以 useState 和 useEffect 为例）

（1）useState 的原理

useState 的核心作用是“创建状态并返回更新状态的函数”，其原理可拆解为 3 个步骤：

1. **首次渲染：创建 Hook 节点并初始化状态**：
2. 当调用 `useState(initialState)` 时，React 会创建一个 Hook 节点，将 `initialState` 作为初始状态存入节点的 `memoizedState`；
3. 创建状态更新队列（queue），用于存储后续通过 setCount 触发的更新；
4. 将该 Hook 节点加入组件的 Hooks 链表，并返回 [memoizedState, dispatch]（dispatch 是更新状态的函数，即 setCount）。

1. **触发更新：将更新加入队列并调度渲染**：
2. 当调用 setCount(newState) 时，实际上是调用 dispatch 函数；
3. dispatch 函数会将 newState 包装成一个“更新对象”，加入到对应 Hook 节点的 queue 中；
4. 调用 React 的调度器（Scheduler），触发组件重新渲染。

1. **重新渲染：合并更新并更新 Hook 节点状态**：
2. 组件重新渲染时，遍历 Hooks 链表找到当前 useState 对应的 Hook 节点；
3. 合并该节点 queue 中的所有更新（如多次调用 setCount，会合并为最终的新状态）；
4. 将合并后的新状态更新到 Hook 节点的 `memoizedState`；
5. 清空更新队列，返回 [新状态, dispatch]。

补充：setCount 是异步的，因为 React 会将更新加入队列后批量处理，避免频繁渲染提升性能（React18 后支持自动批处理）。

（2）useEffect 的原理

useEffect 的核心作用是“处理副作用并在合适时机清理”，其原理可拆解为 4 个步骤：

1. **首次渲染：创建 Effect Hook 节点并存储信息**：
2. 调用 `useEffect(callback, deps)` 时，React 会创建一个 Effect Hook 节点，存储：

3. 副作用回调函数（callback）；
4. 依赖数组（deps）；
5. 清理函数（callback 返回的函数，若有）；
6. 将该节点加入 Hooks 链表。

1. **首次渲染后：执行副作用回调**：
2. 组件首次渲染完成（DOM 更新并绘制后，useEffect 异步执行），React 遍历 Effect Hook 节点，执行每个节点的 callback 函数；
3. 若 callback 返回清理函数，将清理函数存储在 Hook 节点中，供后续使用。

1. **重新渲染时：对比依赖并执行清理/回调**：
2. 组件重新渲染时，找到对应的 Effect Hook 节点；
3. 对比新旧依赖数组（deps）：
4. 若依赖数组变化（浅比较）：先执行上一次存储的清理函数，再执行新的 callback 函数，更新节点中的依赖数组和清理函数；
5. 若依赖数组未变化：不执行任何操作；
6. 若依赖数组为空（[]）：仅在首次渲染时执行 callback，重新渲染时不执行。

1. **组件卸载时：执行清理函数**：
2. 当组件被卸载时，React 遍历所有 Effect Hook 节点，执行每个节点存储的清理函数，清理副作用（如清除定时器、移除事件监听），避免内存泄漏。

3. Hooks 原理的关键总结

- 核心载体：组件的 Fiber 节点，通过 `memoizedState` 存储 Hooks 链表；
- 关联方式：Hooks 的调用顺序与链表中的节点顺序一一对应，确保重新渲染时能正确匹配状态；
- 状态存储：每个 Hook 对应链表中的一个节点，节点存储状态值、更新队列、依赖项等信息；
- 更新机制：通过调用更新函数（如 setCount）将更新加入队列，触发组件重新渲染，遍历链表合并更新并更新状态；
- 规则根源：Hooks 必须在顶层调用，是为了保证重新渲染时 Hooks 的调用顺序与首次渲染一致，避免匹配错误的 Hook 节点。

三十、什么是 React Fiber？它如何改善了 React 应用的性能？

React Fiber 是 React 16 版本引入的「核心架构重构」，本质是一套“重新实现的协调引擎”（Reconciliation Engine）。它并非新增的 API，而是 React 内部用于处理组件渲染、更新的底层机制，核心目标是解决 React 15 及之前版本中“同步渲染导致的页面卡顿问题”，通过“可中断、可恢复、优先级调度”的特性，显著提升应用性能。

1. 先了解 React 15 的问题：同步协调（Stack Reconciler）

React 15 及之前使用的协调引擎是「Stack Reconciler」（栈协调器），其核心问题是「同步且不可中断的渲染流程」：

（1）同步渲染流程

当组件触发更新（如 setState、props 变化）时，React 会执行以下步骤：

1. 「协调（Reconciliation）」：递归遍历新旧虚拟 DOM 树，通过 Diff 算法找出差异（这是一个递归过程，依赖调用栈）；
2. 「提交（Commit）」：根据差异生成 DOM 操作指令，同步执行 DOM 更新。

关键问题：整个协调过程是「同步且不可中断」的。若应用存在复杂的组件树（如嵌套层级深、节点数量多），协调过程会占用主线程较长时间（如几百毫秒）。

（2）主线程阻塞导致卡顿

浏览器的主线程是“单线程”，负责执行 JavaScript、DOM 操作、布局、绘制等任务。若 React 的同步协调过程占用主线程过久，会导致浏览器无法及时响应用户交互（如点击、输入、滚动）、无法及时绘制页面，从而出现页面卡顿、操作延迟等问题。

示例：当用户在复杂组件树中输入文字时，输入事件被 React 的同步协调过程阻塞，导致文字输入延迟，严重影响用户体验。

2. React Fiber 的核心设计：可中断的协调引擎

React Fiber 的核心思路是「将同步的递归协调，拆分为异步的、可中断的单元任务」，并引入「优先级调度」机制，让高优先级任务（如用户交互）能打断低优先级任务（如普通状态更新），待高优先级任务执行完成后，再恢复低优先级任务。

Fiber 的名称来源于“光纤”，寓意“将大任务拆分为细小的单元，像光纤一样并行传输”。从技术角度，Fiber 既是“任务单元”，也是“组件的虚拟节点表示”（替代了 React 15 中的虚拟 DOM 节点）。

（1）Fiber 节点：任务单元与组件状态的载体

每个 Fiber 节点对应一个组件，包含以下核心信息：

- `type`：组件类型（如函数组件、类组件、原生 DOM 组件）；
- `memoizedState`：组件的状态（如 Hooks 链表、类组件的 state）；
- `pendingProps/memoizedProps`：待处理的 props 和已缓存的 props；
- `child/sibling/return`：指向子 Fiber 节点、兄弟 Fiber 节点、父 Fiber 节点（形成链表结构，替代递归调用栈）；
- `effectTag`：标记该组件需要执行的 DOM 操作（如插入、更新、删除）；

- `priority`：该 Fiber 任务的优先级。

关键变化：React 15 中用「递归调用栈」遍历虚拟 DOM 树，React Fiber 中用「链表遍历」替代递归，每个 Fiber 节点是一个独立的任务单元，便于中断和恢复。

（2）核心特性：可中断、可恢复、优先级调度

1. **可中断的协调过程**：
2. Fiber 将协调过程（Diff 算法）拆分为多个独立的任务单元（每个 Fiber 节点的处理是一个任务单元）；
3. React 执行任务单元时，会定期检查是否有更高优先级任务需要执行，或是否超出浏览器分配的时间切片（通常是 16ms，对应 60fps 的刷新率）；
4. 若存在高优先级任务或时间切片用尽，React 会暂停当前任务，保存当前进度（如已处理的 Fiber 节点、任务状态），释放主线程；
5. 待高优先级任务执行完成后，React 会恢复之前暂停的任务，从上次中断的位置继续处理，无需重新开始整个协调过程。
6. **优先级调度机制**：
1. Fiber 为不同类型的更新任务划分了优先级（如用户输入、动画是高优先级，普通状态更新是低优先级）；
2. 调度器（Scheduler）会根据任务优先级排序，高优先级任务优先执行，确保用户交互、动画等关键操作的响应速度；
3. 低优先级任务可能被高优先级任务多次打断，直到没有高优先级任务时才完成执行。
4. **分离协调与提交阶段**：
1. 协调阶段（Reconciliation）：异步、可中断，负责遍历 Fiber 树、执行 Diff 算法、标记需要更新的 Fiber 节点（effectTag）；
2. 提交阶段（Commit）：同步、不可中断，负责根据 Fiber 节点的 effectTag 执行 DOM 操作（插入、更新、删除），并执行 useEffect 回调、组件生命周期方法等；
3. 分离后，只有协调阶段可被中断，提交阶段因涉及真实 DOM 操作，必须同步执行，避免 DOM 状态不一致。

3. Fiber 如何改善 React 应用的性能？

Fiber 通过解决 React 15 同步渲染的核心问题，从三个关键维度提升性能：

- **避免主线程阻塞，提升用户体验**：
- 将不可中断的同步协调拆分为可中断的异步任务，确保主线程不会被长时间占用；
- 浏览器有足够时间响应用户交互（如点击、输入、滚动）、执行动画渲染，避免页面卡顿、操作延迟，尤其是在复杂组件树中效果显著。

- **优先级调度，保障关键任务优先执行**：
- 为不同更新任务分配优先级，高优先级任务（如用户输入、动画）可打断低优先级任务，优先完成执行；
- 例如：用户在输入框打字时，输入事件的优先级高于列表数据的普通更新，Fiber 会优先处理输入更新，确保打字响应流畅，待输入完成后再继续处理列表更新。
- **减少不必要的计算，提升渲染效率**：
- 可中断的协调过程避免了因任务阻塞导致的重复计算（如 React 15 中若同步协调被打断，重新触发更新时需完全重新递归）；
- Fiber 节点的链表结构便于精准定位中断位置，恢复后继续处理，减少无效计算；同时，Fiber 对 Diff 算法的优化（如静态节点标记）也进一步提升了协调效率。

三十一、在 React Fiber 中，什么是 Fiber 节点？它如何与传统的 React 虚拟 DOM 元素相比？

1. 什么是 Fiber 节点？

在 React Fiber 架构中，Fiber 节点是 React 内部的「核心数据结构」，既是「组件状态与属性的载体」，也是「任务调度的基本单元」。它并非对外暴露的 API，而是 React 用于替代传统虚拟 DOM 元素、实现可中断协调与优先级调度的底层基础。

每个 Fiber 节点对应一个组件（或原生 DOM 元素），包含以下核心信息（简化版）：

- **组件相关信息**：`type`（组件类型，如函数组件、类组件、div 等原生 DOM 类型）、`pendingProps`（待处理的 props）、`memoizedProps`（已缓存的当前 props）；
- **状态相关信息**：`memoizedState`（组件的状态数据，如类组件的 state、函数组件的 Hooks 链表）；
- **链表结构信息**：`child`（指向第一个子 Fiber 节点）、`sibling`（指向兄弟 Fiber 节点）、`return`（指向父 Fiber 节点），通过这三个属性形成完整的 Fiber 树（链表结构）；
- **任务调度信息**：`priority`（当前 Fiber 节点对应的任务优先级）、`effectTag`（标记需要执行的 DOM 操作，如插入、更新、删除）、`nextEffect`（指向需要执行副作用的下一个 Fiber 节点）；
- **其他辅助信息**：`alternate`（指向当前 Fiber 节点的备用节点，用于存储更新过程中的临时状态，实现双缓存机制）。

2. Fiber 节点与传统虚拟 DOM 元素的对比

传统的 React 虚拟 DOM 元素（React 15 及之前）是对真实 DOM 的「轻量级描述」，核心作用是用于 Diff 算法对比新旧 DOM 差异；而 Fiber 节点在继承虚拟 DOM 核心功能的基础上，新增了任务调度、状态管理等能力，两者的核心差异可总结为以下维度：

对比维度	Fiber 节点	传统虚拟 DOM 元素
核心定位	组件状态载体 + 任务调度单元	真实 DOM 的轻量级描述
数据结构	链表结构（child/sibling/return），支持遍历中断与恢复	树状结构（递归嵌套），遍历依赖调用栈，不可中断
核心功能	1. 描述组件/ DOM 信息；2. 执行 Diff 算法对比差异；3. 存储组件状态（state/Hooks）；4. 标记任务优先级与 DOM 操作；5. 支持任务中断与恢复	1. 描述组件/ DOM 信息；2. 执行 Diff 算法对比差异
调度能力	具备优先级调度能力，可作为独立任务单元被调度器管理（暂停、恢复、插队）	无调度能力，仅作为 Diff 对比的数据源，遍历过程同步不可中断
状态存储	内置状态存储（memoizedState），可存储类组件 state、函数组件 Hooks 链表等	不存储状态，仅存储组件的 props、type 等描述信息，状态需额外维护
底层作用	支撑 React 可中断协调、优先级调度、双缓存渲染等核心机制	仅用于 Diff 算法对比，为 DOM 更新提供差异依据

3. 关键总结

Fiber 节点并非完全替代虚拟 DOM，而是在传统虚拟 DOM 的基础上进行了「功能增强与架构升级」：传统虚拟 DOM 是“静态的描述性数据”，仅用于 Diff 对比；而 Fiber 节点是“动态的任务与状态载体”，既保留了虚拟 DOM 的 Diff 核心功能，又新增了链表结构、优先级调度、状态存储等能力，最终支撑 React 实现可中断渲染，解决了传统同步渲染的卡顿问题。

三十二、React Fiber 的协调算法（Reconciliation）与之前的版本有何不同?

React 的协调算法（Reconciliation）核心作用是「对比新旧组件树的差异，确定需要更新的部分」，是 DOM 更新的前置步骤。React Fiber 架构对协调算法进行了彻底重构，与 React 15 及之前的栈协调器（Stack Reconciler）相比，核心差异在于「从同步不可中断的递归遍历，改为异步可中断的链表遍历」，并新增了优先级调度机制，具体不同可分为以下维度：

1. 遍历方式：递归遍历 vs 链表遍历

- **React 15 协调算法（Stack Reconciler）**：

- 采用「递归深度优先遍历」方式遍历虚拟 DOM 树，依赖 JavaScript 调用栈实现；
- 遍历过程是「同步且不可中断」的：一旦开始遍历，必须完整遍历整个组件树，直到找出所有差异，期间无法释放主线程；
- 问题：若组件树层级深、节点多，遍历耗时过长（超过 16ms），会阻塞主线程，导致页面卡顿、用户交互延迟。
- **React Fiber 协调算法**：
- 采用「链表遍历」方式遍历 Fiber 树，通过 Fiber 节点的 `child`（子）、`sibling`（兄弟）、`return`（父）三个属性实现遍历，替代了递归调用栈；
- 遍历过程是「异步且可中断」的：将整个遍历过程拆分为多个独立的任务单元（每个 Fiber 节点的处理为一个任务单元），执行每个任务单元后，会检查是否有更高优先级任务或时间切片用尽，若有则暂停当前任务，保存进度后释放主线程；
- 优势：避免主线程长时间被占用，保障用户交互、动画等关键操作的流畅性。

2. 调度机制：无优先级 vs 优先级调度

- **React 15 协调算法**：
- 无优先级概念，所有更新任务“一视同仁”，按触发顺序同步执行；
- 问题：高优先级任务（如用户输入、动画）可能被低优先级任务（如普通数据更新）阻塞，导致关键操作响应延迟。
- **React Fiber 协调算法**：
- 引入「优先级调度机制」，为不同类型的更新任务划分优先级（如 React 内部定义的 ImmediatePriority、UserBlockingPriority、NormalPriority 等）；
- 调度器（Scheduler）会根据任务优先级排序，高优先级任务可打断正在执行的低优先级任务，优先完成；低优先级任务被打断后，会在高优先级任务执行完成后，从上次中断的位置恢复执行；
- 优势：确保用户交互、动画等关键任务优先响应，提升应用的交互体验。

3. 协调与提交阶段：未分离 vs 完全分离

- **React 15 协调算法**：
- 协调阶段（Diff 对比）与提交阶段（DOM 更新）未完全分离，递归遍历过程中可能穿插 DOM 操作；
- 问题：进一步加剧主线程阻塞，且若遍历中断（如抛出异常），可能导致 DOM 状态不一致。
- **React Fiber 协调算法**：
- 将「协调阶段」与「提交阶段」完全分离：
- 协调阶段（Reconciliation）：异步、可中断，负责遍历 Fiber 树、执行 Diff 对比、标记需要执行的 DOM 操作（effectTag），不执行任何真实 DOM 操作；

- 提交阶段（Commit）：同步、不可中断，负责根据协调阶段标记的 effectTag，批量执行真实 DOM 操作（插入、更新、删除），并执行组件生命周期方法、useEffect 回调等；
- 优势：协调阶段的异步可中断特性不会影响真实 DOM 状态，提交阶段的同步执行确保 DOM 操作的原子性，避免 DOM 状态混乱。

4. 性能优化：无缓存 vs 双缓存机制

- **React 15 协调算法**：
- 无缓存机制，每次更新都需要重新创建完整的虚拟 DOM 树，再与旧树进行 Diff 对比；
- 问题：重复创建虚拟 DOM 树增加内存开销，Diff 对比效率较低。
- **React Fiber 协调算法**：
- 引入「双缓存机制」，通过 Fiber 节点的 `alternate` 属性维护两棵 Fiber 树：
- current 树：当前渲染在页面上的 Fiber 树，与真实 DOM 一一对应；
- workInProgress 树：正在进行更新的 Fiber 树，用于执行 Diff 对比、标记 effectTag，更新过程中不会影响 current 树；
- 更新完成后，通过切换 current 树与 workInProgress 树的引用，完成 DOM 更新，旧的 current 树变为下一次更新的 workInProgress 树；
- 优势：避免重复创建 Fiber 节点，减少内存开销；同时，workInProgress 树的异步构建不会影响当前页面的渲染状态，提升稳定性。

5. 关键总结

React Fiber 的协调算法是对传统栈协调器的「颠覆性重构」，核心目标是「解决同步渲染的卡顿问题」。通过“链表遍历替代递归遍历”“引入优先级调度”“分离协调与提交阶段”“双缓存机制”四大核心优化，实现了异步可中断的协调过程，既保障了关键操作的响应速度，又提升了渲染效率与稳定性。

三十三、什么是虚拟 DOM？以及它如何帮助提升 React 应用的性能？

1. 什么是虚拟 DOM？

虚拟 DOM（Virtual DOM，简称 VDOM）是 React 等前端框架中用于「描述真实 DOM 结构的轻量级 JavaScript 对象」，是对真实 DOM 的抽象映射。它不直接操作浏览器的真实 DOM，而是通过对象的形式记录真实 DOM 的关键信息（如节点类型、属性、子节点等），本质是“在内存中模拟 DOM 树的结构”。

例如，一个真实的 div 节点：

Code block

```
1   <div className="container" id="root">
```

```
  <p>Hello React</p>
</div>
```

对应的虚拟 DOM 对象（简化版）可能是：

Code block

```
{
  type: 'div', // 节点类型（原生 DOM 类型/组件类型）
  props: { // 节点属性
    className: 'container',
    id: 'root'
  },
  children: [ // 子节点（同样是虚拟 DOM 对象）
    {
      type: 'p',
      props: {},
      children: ['Hello React'] // 文本节点
    }
  ],
  key: null // 用于 Diff 算法的唯一标识（列表节点常用）
}
```

核心特点：

- 轻量级：仅包含真实 DOM 的必要信息，不包含浏览器 DOM 节点的复杂属性与方法，操作成本远低于真实 DOM；
- 内存中存在：虚拟 DOM 是 JavaScript 对象，存储在内存中，操作时不会触发浏览器的重排（Reflow）与重绘（Repaint）；
- 可描述性：完整映射真实 DOM 的层级结构与属性信息，能够精准描述页面的 UI 状态。

2. 虚拟 DOM 如何提升 React 应用的性能?

浏览器操作真实 DOM 的成本极高（尤其是重排/重绘），而虚拟 DOM 通过「减少真实 DOM 的操作次数」和「最小化 DOM 更新范围」，显著提升 React 应用的性能，具体优化逻辑可分为三个核心步骤：

（1）避免频繁操作真实 DOM，减少重排/重绘

在传统原生 JavaScript 开发中，若需要频繁更新页面（如列表数据刷新、表单输入联动），会直接多次操作真实 DOM，每次操作都会触发浏览器的重排/重绘，导致性能开销巨大。

虚拟 DOM 的优化逻辑：

- React 中所有的 UI 更新都会先操作虚拟 DOM（内存中的 JavaScript 对象），而非直接操作真实 DOM；

- 多次虚拟 DOM 操作会被合并为一次真实 DOM 更新，避免了频繁操作真实 DOM 导致的多次重排/重绘；
- 例如：连续修改一个元素的 className 和文本内容，React 会先在虚拟 DOM 中完成两次修改，再通过一次真实 DOM 操作同步到页面，而非两次独立的 DOM 操作。

（2）通过 Diff 算法，最小化 DOM 更新范围

当应用状态变化时，React 会创建一棵新的虚拟 DOM 树，然后通过「Diff 算法」对比新旧两棵虚拟 DOM 树的差异，找出“需要更新的最小部分”，仅将差异部分同步到真实 DOM 中，而非整体替换整个 DOM 树。

核心优势：

- 传统 DOM 更新可能需要替换整个父节点及其所有子节点，即使只有一个子节点变化；
- 虚拟 DOM + Diff 算法能精准定位变化的节点，仅更新变化的部分，避免了不必要的 DOM 操作，极大降低了性能开销；
- 例如：列表中某一个列表项的文本变化，React 会通过 Diff 算法找到该列表项对应的虚拟 DOM 节点，仅更新该节点的文本内容，而不影响列表中的其他节点。

（3）抽象 DOM 操作，提升开发效率与跨平台能力

虚拟 DOM 对 DOM 操作进行了抽象封装，开发者无需直接编写 DOM 操作代码，只需关注应用状态与 UI 的映射关系（如编写 JSX），React 会自动完成虚拟 DOM 到真实 DOM 的同步。

间接性能优化：

- 减少开发者手动编写低效 DOM 操作代码的可能性（如冗余的 DOM 查询、重复的 DOM 修改）；
- 虚拟 DOM 是平台无关的抽象层，除了映射浏览器 DOM，还可映射到其他平台（如 React Native 中的原生组件），实现“一次编写，多平台运行”，避免了为不同平台单独优化 DOM 操作的成本。

3. 注意事项：虚拟 DOM 并非“更快”，而是“更高效”

虚拟 DOM 本身存在一定的性能开销（如创建虚拟 DOM 对象、执行 Diff 算法），因此对于简单的、单次的 DOM 操作，虚拟 DOM 的性能可能不如直接操作真实 DOM。但在复杂应用中，虚拟 DOM 通过“合并操作”和“最小化更新”带来的性能优势，远大于其自身的开销，整体提升应用的性能与可维护性。

三十四、React 的 Diff 算法是如何工作的（也称为 Reconciliation 算法）？请解释它如何比较两个虚拟 DOM 树的。

React 的 Diff 算法（又称协调算法/Reconciliation 算法）是 React 核心优化机制之一，核心目标是「高效对比新旧两棵虚拟 DOM 树的差异，找出需要更新的最小部分」，从而最小化真实 DOM 的操作，提升应用性能。其设计基于两个关键假设（ React 官方提出），并通过“分层对比+同层优化”的策略实现高效对比。

1. Diff 算法的核心假设（前提条件）

React Diff 算法的高效性依赖于两个合理的假设，若开发者遵循这些假设，能进一步提升 Diff 效率：

- **假设 1：同类型的组件产生相似的 DOM 结构，不同类型的组件产生不同的 DOM 结构**：
- 若两个组件类型相同（如都是 或都是自定义组件 <UserList>），则认为它们的 DOM 结构相似，继续深入对比其属性和子节点；
- 若两个组件类型不同，则直接销毁旧组件对应的 DOM 结构，创建新组件对应的 DOM 结构，无需深入对比子节点（大幅减少对比成本）。
- **假设 2：列表节点通过唯一的 key 属性，可以确定节点在列表中的位置是否变化**：
- 对于列表类子节点（如 <ul> 下的多个 ），若未设置 key，React 会通过“索引”对比，可能导致错误的 DOM 复用；
- 若设置了唯一的 key，React 可通过 key 快速定位节点的新增、删除、移动操作，提升列表对比效率（详细逻辑见第 35 题）。

2. Diff 算法的核心工作流程：分层对比（深度优先）

React Diff 算法采用「分层对比」策略，而非传统的“全量递归对比”，即从根节点开始，按层级遍历对比新旧虚拟 DOM 树的节点，同一层级的节点逐一对比，不跨层级对比（基于“UI 页面的 DOM 结构很少跨层级变化”的现实情况，大幅减少对比成本）。

具体工作流程可分为三个核心步骤：

（1）对比根节点：根据组件类型判断是否复用 DOM

首先对比新旧虚拟 DOM 树的根节点，根据节点类型（type）分为两种情况：

- **节点类型不同（如旧节点是 ，新节点是 <p>）**：
- 直接销毁旧节点对应的真实 DOM 及其所有子节点；
- 创建新节点对应的真实 DOM 及其所有子节点，替换旧节点的位置；
- 无需继续对比该节点的子节点（因为类型不同，DOM 结构完全不同）。
- **节点类型相同（如新旧节点都是 ）**：
- 复用旧节点对应的真实 DOM（避免重新创建 DOM 的开销）；
- 对比节点的属性（props）：若属性有变化（如 className、id、事件绑定等），则更新真实 DOM 的对应属性；若属性无变化，则不操作；
- 继续递归对比该节点的子节点（进入下一层级对比）。

（2）对比子节点：分“非列表子节点”和“列表子节点”优化

当父节点类型相同，需要对比其子节点时，React 会根据子节点的类型（是否为列表）采用不同的对比策略：

- **非列表子节点（如 下的 、<img> 等无序子节点）：**
- 按“索引顺序”逐一对比新旧子节点列表中的节点；
- 若旧子节点列表长度大于新列表：删除多余的旧子节点对应的 DOM；
- 若新子节点列表长度大于旧列表：新增多余的新子节点对应的 DOM；
- 若索引对应的节点类型相同：复用 DOM 并更新属性，继续对比其子节点；
- 若索引对应的节点类型不同：销毁旧节点，创建新节点，替换位置。
- **列表子节点（如 <ul> 下的多个 ，或通过 map 生成的子节点）：**
- 无 key 时：按索引对比（存在缺陷，如列表插入/删除元素时导致错误复用 DOM）；
- 有 key 时：通过 key 建立新旧节点的映射关系，快速判断节点的“新增”“删除”“移动”操作：
- 步骤 1：遍历新子节点列表，用 key 构建“key - 新节点”的映射表；
- 步骤 2：遍历旧子节点列表，通过 key 查找新节点列表中是否存在对应节点：
- - 若不存在：标记该旧节点为“删除”，后续销毁对应的 DOM；
- - 若存在：对比节点类型和属性，若有变化则更新，同时记录节点的位置变化；
- 步骤 3：遍历新子节点列表，查找旧节点列表中不存在的 key，标记为“新增”，后续创建对应的 DOM 并插入；
- 步骤 4：根据节点位置变化记录，调整真实 DOM 的顺序（避免销毁再创建，仅移动位置，提升效率）。

（3）标记差异并提交更新

在对比过程中，React 会为所有需要更新的虚拟 DOM 节点标记“差异类型”（如属性更新、节点新增、节点删除、节点移动等）；

对比完成后，React 会一次性将所有标记的差异同步到真实 DOM 中（批量更新），避免频繁操作真实 DOM 导致的多次重排/重绘。

3. 示例：简单 Diff 对比过程

假设旧虚拟 DOM 树：

Code block

```
<div className="old">
  <p>旧文本</p>
  <ul>
    <li key="1">列表项 1</li>
    <li key="2">列表项 2</li>
  </ul>
</div>
```

新虚拟 DOM 树：

```
<div className="new">
  <p>新文本</p>
  <ul>
    <li key="1">列表项 1</li>
    <li key="3">列表项 3</li>
  </ul>
</div>
```

Diff 对比过程：

1. 对比根节点（都是 ）：复用 DOM，更新 className 为 "new"；
2. 对比 的子节点（<p> 和 ）：
3. - 对比 <p> 节点（类型相同）：复用 DOM，更新文本内容为 "新文本"；
4. - 对比 <ul> 节点（类型相同）：复用 DOM，继续对比其子节点；
5. 对比 <ul> 的子节点（列表项）：
6. - 通过 key 对比：key="1" 的节点存在且无变化，复用；key="2" 的节点在新列表中不存在，标记删除；key="3" 的节点在旧列表中不存在，标记新增；
7. 提交更新：同步 className 变化、文本变化，删除 key="2" 的 ，新增 key="3" 的 。

4. 关键总结

React Diff 算法的核心优势是「基于合理假设的分层对比策略」：通过“同类型组件深入对比、不同类型组件直接替换”减少对比深度，通过“key 优化列表对比”减少列表操作成本，最终实现高效的差异查找；对比完成后批量更新真实 DOM，最小化 DOM 操作开销，提升应用性能。

三十五、React 在处理列表时为什么推荐使用唯一的 key 属性？这与 Diff 算法有什么关系？

1. 核心原因：帮助 Diff 算法精准识别列表节点的变化

React 处理列表时推荐使用唯一的 key 属性，核心目的是「为列表中的每个节点提供唯一标识」，帮助 Diff 算法快速、准确地判断节点的“新增”“删除”“移动”操作，避免错误的 DOM 复用，同时提升列表对比的效率。

若不设置 key 或 key 不唯一，React 会默认使用「索引」作为 key 对比节点，这种方式在列表发生“插入”“删除”“重新排序”等操作时，会导致 Diff 算法误判，引发 UI 异常或性能问题。

2. 无 key（索引作为 key）的问题：错误的 DOM 复用与性能损耗

当列表节点使用索引作为 key 时，Diff 算法会通过“索引匹配”来判断节点是否可复用，这种方式在列表结构变化时会出现两个核心问题：

（1）问题 1：UI 展示异常（错误复用带状态的节点）

若列表节点是带状态的组件（如输入框、复选框），使用索引作为 key 会导致 React 错误复用旧节点的 DOM，从而保留旧节点的状态，引发 UI 异常。

示例：

Code block

```
// 初始列表（索引 0、1 作为 key）
<ul>
  <li key={0}><input placeholder="输入内容" /> 列表项 1</li>
  <li key={1}><input placeholder="输入内容" /> 列表项 2</li>
</ul>
```

若在列表头部插入一个新节点，新列表变为：

Code block

```
<ul>
  <li key={0}><input placeholder="输入内容" /> 新列表项</li>
  <li key={1}><input placeholder="输入内容" /> 列表项 1</li>
  <li key={2}><input placeholder="输入内容" /> 列表项 2</li>
</ul>
```

Diff 算法对比过程：

- 索引 0：旧节点是“列表项 1”，新节点是“新列表项”，类型相同，React 会复用旧节点的 DOM（包括输入框的状态），仅更新文本内容；
- 索引 1：旧节点是“列表项 2”，新节点是“列表项 1”，同样复用旧节点的 DOM；
- 索引 2：新节点，创建新 DOM；
- 结果：若原列表项 1 的输入框有内容，插入新节点后，新列表项的输入框会保留原列表项 1 的内容，出现 UI 异常。

（2）问题 2：性能损耗（不必要的 DOM 销毁与创建）

当列表发生重新排序时，使用索引作为 key 会导致 Diff 算法误判节点为“删除+新增”，而非“移动”，从而销毁旧节点并创建新节点，产生不必要的性能开销。

示例：列表 [A, B, C]（索引 0、1、2）重新排序为 [B, A, C]，使用索引作为 key 时：

- Diff 算法认为索引 0 的节点从 A 变为 B（销毁 A，创建 B），索引 1 的节点从 B 变为 A（销毁 B，创建 A），索引 2 的节点无变化；

- 实际只需移动 A 和 B 的位置即可，但因索引 key 导致额外的 DOM 销毁与创建，浪费性能。

3. 有唯一 key 的优势：精准匹配与高效更新

当列表节点设置唯一的 key（如后端返回的 id、唯一标识符）时，Diff 算法会通过 key 建立新旧节点的映射关系，精准判断节点的变化类型，解决上述问题：

（1）优势 1：避免错误复用带状态节点

key 是节点的唯一标识，与索引无关，即使列表结构变化，React 也能通过 key 找到对应的节点，不会错误复用其他节点的 DOM 状态。

延续上述插入示例，设置 key 为节点唯一 id：

Code block

```
// 初始列表（key 为 101、102）
<ul>
  <li key={101}><input placeholder="输入内容" /> 列表项 1</li>
  <li key={102}><input placeholder="输入内容" /> 列表项 2</li>
</ul>

// 插入新节点后（新节点 key 为 100）
<ul>
  <li key={100}><input placeholder="输入内容" /> 新列表项</li>
  <li key={101}><input placeholder="输入内容" /> 列表项 1</li>
  <li key={102}><input placeholder="输入内容" /> 列表项 2</li>
</ul>
```

Diff 算法对比过程：

- key=100：新节点，创建新 DOM（输入框为空，无状态复用）；
- key=101、102：旧节点存在，复用 DOM（保留原输入框状态）；
- 结果：UI 状态正常，无异常复用。

（2）优势 2：提升列表更新性能（识别节点移动）

通过 key 映射，Diff 算法能快速判断节点是否仅发生“位置移动”，无需销毁再创建 DOM，只需调整 DOM 顺序，大幅提升性能。

延续排序示例，列表 [A(key=101), B(key=102), C(key=103)] 重新排序为 [B(key=102), A(key=101), C(key=103)]：

- Diff 算法通过 key 发现：B、A、C 都是旧节点，仅位置变化；
- 直接移动 A 和 B 的 DOM 位置，无需销毁创建，性能开销极低。

4. key 的使用规范（避免踩坑）

- **key 必须唯一**：同一列表中不能有重复的 key，否则 React 会报警告，且 Diff 算法无法正确匹配节点；
- **key 必须稳定**：key 不应随渲染次数变化（如不推荐用 Math.random() 生成 key），否则每次渲染都会被认为是新节点，销毁旧节点创建新节点，严重影响性能；
- **优先使用后端返回的唯一 id**：如数据库主键 id，确保 key 的唯一性与稳定性；
- **避免用索引作为 key**：仅当列表是“静态列表”（无插入、删除、排序操作）且无状态时，可临时使用，否则必须用唯一 id。

5. 与 Diff 算法的关系总结

key 是 React Diff 算法处理列表节点的「核心优化手段」：Diff 算法的设计基于“通过 key 确定节点位置变化”的假设，key 为 Diff 算法提供了节点的唯一标识，使其能：

- 精准区分节点的“新增”“删除”“移动”操作，避免错误复用 DOM 导致的 UI 异常；
- 减少不必要的 DOM 销毁与创建，通过移动 DOM 位置提升列表更新性能；
- 降低 Diff 算法对比列表节点的复杂度，提升整体对比效率。

因此，使用唯一的 key 属性，是确保 React 列表渲染正确且高效的关键前提。

三十六、React 与 Vue 的 diff 算法在实现上有什么区别?

React 和 Vue 的 diff 算法核心目标一致：均为「高效对比新旧虚拟 DOM 树差异，最小化真实 DOM 操作」，且都基于“分层对比”（不跨层级对比，降低复杂度）的核心策略。但两者在具体实现细节、优化方向、适用场景适配等方面存在显著差异，核心区别集中在“对比粒度”“列表优化细节”“静态节点处理”等维度。

1. 核心设计思路差异：“整体协调” vs “精准靶向”

- **React Diff 算法**：
- 核心思路是「整体协调（Reconciliation）」，更侧重“任务调度的兼容性”。由于 React 引入了 Fiber 架构支持可中断渲染，其 diff 算法需要适配优先级调度机制，因此对比过程被设计为“可中断的链表遍历”，而非一次性完成全量对比。
- 设计前提是「组件类型决定 DOM 结构相似性」：若组件类型不同，直接销毁旧组件 DOM 树并重建新组件 DOM 树，不深入子节点对比；若类型相同，才继续对比属性和子节点。这种“粗粒度预判”能快速减少对比范围，适配 Fiber 的中断调度需求。
- **Vue Diff 算法**：
- 核心思路是「精准靶向更新」，更侧重“DOM 操作的最小化”。Vue 没有 Fiber 那样的可中断调度机制（Vue3 虽引入调度器，但核心仍为同步对比），因此 diff 算法追求“一次性精准定位差异”，减少不必要的遍历和判断。

- 设计前提是「模板编译优化」：Vue 基于模板语法开发，编译阶段会对模板进行静态分析（如标记静态节点、静态根节点），运行时 diff 时可直接跳过静态节点对比，大幅提升效率。而 React 基于 JSX 动态生成虚拟 DOM，无编译阶段的静态优化（需依赖开发者手动优化，如 memo、useMemo）。

2. 子节点对比策略差异（核心差异点）

子节点对比是 diff 算法的核心，两者在列表子节点对比和非列表子节点对比上均有差异，其中列表对比差异最为显著：

（1）列表子节点对比：key 作用与移动优化逻辑

- **React**：
- key 的核心作用是「建立新旧节点的映射关系」，用于判断节点的“新增/删除/移动”。但 React 对列表节点移动的优化较为基础：
- 1. 遍历新列表，用 key 构建“key-新节点”映射表；
- 2. 遍历旧列表，通过 key 查找新列表中的对应节点：不存在则标记删除，存在则记录位置；
- 3. 遍历新列表，查找旧列表中不存在的节点标记新增；
- 4. 对于需要移动的节点，通过“最长递增子序列”计算最小移动步数（React16+ 引入，优化移动效率），但整体逻辑仍服务于 Fiber 的可中断遍历。
- 局限：当列表节点数量庞大且移动频繁时，对比效率受 Fiber 调度开销影响，略低于 Vue。
- **Vue**：
- Vue2 和 Vue3 的列表 diff 均对“移动优化”做了深度适配，尤其是 Vue3 的“最长递增子序列”优化更精准：
- 1. Vue2：采用“双指针法”遍历新旧列表，通过 key 对比节点，同时记录节点移动、新增、删除状态，对比过程一次性完成，无调度开销；
- 2. Vue3：在 Vue2 基础上优化，通过“最长递增子序列”确定无需移动的节点，仅对其他节点进行移动操作，进一步减少 DOM 操作次数；
- 优势：同步对比+精准移动优化，在列表更新场景下（如长列表滚动加载、频繁排序）效率更高。

（2）非列表子节点对比：静态节点处理差异

- **React**：无编译阶段的静态节点标记，即使节点属性和内容完全不变（如静态文本、固定样式节点），每次 diff 仍会对比其属性和子节点（除非通过 React.memo 或 PureComponent 手动优化），存在冗余对比开销。
- **Vue**：编译阶段会标记“静态节点”（如 静态文本）和“静态根节点”（包含多个静态节点的父节点），运行时 diff 时会直接跳过这些节点的对比（因为其内容和属性不会变化），无需任何冗余判断，大幅提升对比效率。

3. 对比粒度与中断机制差异

- **React**：
- 对比粒度是「Fiber 任务单元」：将整个 diff 过程拆分为多个独立的任务单元（每个 Fiber 节点对应一个任务单元），执行过程中可被高优先级任务中断（如用户输入、动画），待高优先级任务完成后恢复对比。
- 优势：避免长时间占用主线程，保障用户交互流畅；劣势：中断与恢复过程需要保存/读取任务状态，存在少量调度开销。
- **Vue**：
- 对比粒度是「节点层级」：diff 过程为同步一次性完成，无中断机制（Vue3 虽支持调度器，但仅用于异步组件、nextTick 等场景，不影响 diff 核心流程）。
- 优势：无调度开销，对比过程更高效；劣势：若组件树层级过深、节点过多，可能阻塞主线程，导致页面卡顿（需通过拆分组件、懒加载等手动优化）。

4. 适用场景差异总结

- **React Diff**：更适合“复杂组件树+频繁用户交互”场景（如企业级后台、移动端应用），通过可中断调度保障交互流畅，牺牲少量 diff 效率换取更好的用户体验。
- **Vue Diff**：更适合“中等复杂度组件树+频繁列表更新”场景（如电商列表、数据可视化页面），通过编译优化和精准移动优化，实现更高的 diff 效率和 DOM 更新性能。

三十七、React18 引入了哪些重要的新特性？简单介绍一下

React18 的核心升级方向是「并发渲染（Concurrent Rendering）」，通过重构底层架构（如调度器、协调器），引入一系列新特性和 API，旨在提升应用的响应速度和用户体验，同时保持对旧版本的向下兼容。核心新特性可分为“底层架构升级”“核心 API 新增/优化”“开发体验优化”三大类：

1. 底层架构核心升级：并发渲染（Concurrent Rendering）

这是 React18 最核心的升级，并非新增 API，而是底层协调引擎的能力增强。并发渲染允许 React 中断、暂停、恢复或放弃渲染任务，优先处理高优先级任务（如用户输入、动画），避免低优先级任务（如数据加载、列表渲染）阻塞主线程，从而保障页面交互流畅。

关键说明：并发渲染默认不开启，需通过新的根节点渲染 API（createRoot）启用（旧版 render API 仍兼容，但无法使用并发特性）。

2. 核心 API 新增与优化

（1）新的根节点渲染 API：createRoot

替代旧版 ReactDOM.render，用于启用并发渲染模式，同时简化根节点渲染逻辑：

```
· · · ·
// 旧版（React17 及之前）
ReactDOM.render(<App />, document.getElementById('root'));

// 新版（React18）
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

优势：启用并发渲染、支持自动批处理、简化服务端渲染逻辑。

（2）自动批处理（Automatic Batching）

升级了旧版的“批量更新”机制，将批量更新的范围扩展到所有场景（包括原生事件、setTimeout、Promise 回调等），默认合并多个 setState 调用为一次渲染，减少 DOM 操作次数，提升性能（旧版仅在 React 合成事件、生命周期中支持批量更新）。

示例：

Code block

```

// React18 前：setTimeout 中不支持批量更新，会触发 2 次渲染
setTimeout(() => {
  setState({ count: 1 });
  setState({ name: 'xxx' });
}, 0);

// React18 后：自动批处理，仅触发 1 次渲染
setTimeout(() => {
  setState({ count: 1 });
  setState({ name: 'xxx' });
}, 0);
```

若需强制同步更新，可使用 ReactDOM.flushSync 包裹。

（3）过渡更新（Transitions）：startTransition

用于标记“低优先级更新”（如列表筛选、搜索结果渲染），告知 React 此类更新可被高优先级任务（如用户输入）中断，避免因低优先级更新导致页面卡顿。

核心价值：区分“紧急更新”（如输入框输入）和“非紧急更新”（如输入后的搜索结果渲染），保障紧急更新的响应速度。

示例：

Code block

```
```

```
import { startTransition } from 'react';

function handleInputChange(e) {
  // 紧急更新：更新输入框内容（优先执行，不被中断）
  setInputValue(e.target.value);

  // 非紧急更新：筛选列表（可被输入操作中断）
  startTransition(() => {
    setFilteredList(filterData(e.target.value));
  });
}
```

（4） suspense 与流式服务端渲染（Streaming SSR）

增强了 suspense 的能力，使其支持服务端渲染场景：

- 流式 SSR：服务端可分块发送 HTML 内容，先发送静态部分，再发送动态数据加载完成的部分，减少页面“白屏时间”；
- suspense fallback：在服务端渲染时，若组件数据未加载完成，可先渲染 fallback 占位符（如加载中），数据加载完成后替换为真实内容。

3. 开发体验与稳定性优化

（1）严格模式（StrictMode）增强

新增“组件自动卸载与重新挂载”检测，用于发现因未清理副作用（如定时器、事件监听）导致的内存泄漏问题。在严格模式下，React 会模拟组件的挂载-卸载-重新挂载过程，若存在未清理的副作用，会触发警告。

（2）废弃与兼容处理

废弃了部分旧 API（如 ReactDOM.unmountComponentAtNode，替代为 root.unmount()），同时提供兼容层，确保旧版代码可正常运行；优化了错误边界（Error Boundary）的能力，使其能捕获更多场景的错误（如并发渲染中的错误）。

4. 其他重要特性

- useId：生成跨服务端和客户端的唯一 ID，解决服务端渲染时 ID 不匹配的问题（如表单标签关联、组件唯一标识）；
- useTransition：Hook 版本的 startTransition，用于在函数组件中标记过渡更新；
- useDeferredValue：延迟更新非紧急状态，类似 useTransition，但更侧重“值的延迟传递”（如延迟传递筛选后的列表数据）。

三十八、在 React 17 中，事件处理机制经历了哪些变化？特别是与之前版本相比，事件委托的处理有何不同？

React 17 对事件处理机制的修改核心是「调整事件委托的目标节点」，同时保持合成事件的 API 完全兼容（开发者无需修改代码）。此外，还优化了事件系统的兼容性和可维护性，解决了旧版本事件委托的部分痛点。

1. 核心变化一：事件委托目标从 document 改为根节点（root）

这是 React 17 事件机制最核心的变化，直接影响事件冒泡和委托的底层逻辑：

（1）React 16 及之前的事件委托逻辑

- 所有 React 合成事件（如 onClick、onChange）都委托到 `document` 节点上；
- 当用户触发真实 DOM 事件时，事件冒泡到 document 节点，被 React 统一的事件监听器捕获，再分发到对应的组件。

痛点：

- 1. 若第三方库或原生代码修改了 document 的事件监听器（如阻止冒泡），会干扰 React 事件的正常触发；
- 2. 服务端渲染（SSR）时，document 可能不存在，需额外处理事件绑定逻辑；
- 3. 多个 React 应用共存时（如同一页面嵌入多个独立 React 应用），所有应用的事件都会委托到 document，可能出现冲突。

（2）React 17 的事件委托逻辑

- 将事件委托的目标从 `document` 改为「应用的根节点」（即 ReactDOM.render 挂载的节点，如 ）；
- 每个 React 应用的事件监听器都绑定在自己的根节点上，事件冒泡到根节点后被 React 捕获并分发。

优势：

- 1. 避免与第三方库/原生代码的事件冲突，提升事件系统的稳定性；
- 2. 简化服务端渲染的事件处理逻辑；
- 3. 支持同一页面多个 React 应用独立运行，互不干扰。

2. 核心变化二：原生事件与合成事件的冒泡关系调整

React 17 之前，合成事件的冒泡是在 document 节点完成的，因此原生事件若在 document 之前阻止冒泡，会导致合成事件无法触发；React 17 中，合成事件的冒泡在根节点完成，原生事件与合成事件的冒泡边界更清晰：

- 若在原生事件中阻止冒泡（e.stopPropagation()），且该事件的冒泡范围未到达根节点，则不会影响 React 合成事件；
- 若原生事件冒泡到根节点后被阻止，则合成事件的冒泡会被中断（但合成事件本身仍会触发）。

示例：

Code block

```
// 原生事件绑定在根节点的子节点上
document.getElementById('child').addEventListener('click', (e) => {
  e.stopPropagation(); // 阻止冒泡到根节点
});

// React 组件的合成事件
function App() {
  const handleClick = () => console.log('合成事件触发');
  return <div id="child" onClick={handleClick}>点击</div>;
}
```

React 17 中，点击后合成事件仍会触发（因为事件委托在根节点，原生事件阻止冒泡未到达根节点）；React 16 及之前，合成事件不会触发（因为原生事件阻止冒泡后，事件无法到达 document 节点）。

3. 其他变化：事件系统的底层重构与兼容性优化

- （1）废弃了旧版的事件池（SyntheticEvent Pool）：React 17 之前，合成事件对象会被复用，需调用 e.persist() 才能在异步回调中访问；React 17 中，合成事件对象不再复用，可直接在异步回调中访问，无需调用 e.persist()；
- （2）优化了对浏览器原生事件的兼容性处理：支持更多边缘浏览器的事件类型，提升事件系统的鲁棒性；
- （3）简化了事件绑定的底层代码：React 17 重写了事件绑定的核心逻辑，减少了冗余代码，提升了事件系统的可维护性。

4. 关键总结：API 兼容，底层优化

React 17 的事件机制变化均为底层实现调整，未修改任何合成事件的 API（如 onClick、e.target、e.preventDefault 等），因此开发者无需修改现有代码即可升级。核心价值在于提升事件系统的稳定性、兼容性和可扩展性，解决了旧版本事件委托的多个痛点。

三十九、React 18 引入了自动批处理。请解释什么是批处理，以及自动批处理如何改善了 React 应用的性能？

1. 什么是批处理（Batching）？

批处理是 React 用于优化性能的核心机制之一，核心定义是：「将多个状态更新（setState/useState 更新）合并为一次渲染，减少真实 DOM 的操作次数」。

原理：React 中每次状态更新都会触发组件重新渲染（包括虚拟 DOM 对比、真实 DOM 更新），若短时间内多次调用状态更新，会导致多次渲染，产生大量冗余的 DOM 操作（浏览器重排/重绘），严重影响性能。批处理通过收集短时间内的多个状态更新，合并为一次状态计算和一次渲染，从而减少 DOM 操作，提升应用性能。

示例（未批处理 vs 批处理）：

Code block

```
1
2   // 多次状态更新
3   setState({ count: 1 });
4   setState({ name: 'xxx' });
5   setState({ isShow: true });
```

- 未批处理：触发 3 次渲染，3 次虚拟 DOM 对比，3 次 DOM 更新；
- 批处理：触发 1 次渲染，1 次虚拟 DOM 对比，1 次 DOM 更新（合并 3 个状态更新为一个完整状态）。

2. 旧版 React 的批处理局限（React17 及之前）

React17 及之前的批处理机制存在「场景限制」，仅在 React 自身控制的执行环境中支持批处理，在非 React 控制的环境中不支持，导致部分场景下仍会出现冗余渲染：

（1）支持批处理的场景

- React 合成事件回调（如 onClick、onChange、onSubmit 等）；
- React 生命周期方法（如 componentDidMount、componentDidUpdate 等）；
- React Hooks 回调（如 useEffect 等）。

（2）不支持批处理的场景

- 原生 DOM 事件回调（如 addEventListener 绑定的事件）；
- 定时器回调（如 setTimeout、setInterval）；
- Promise 回调（如 Promise.then、async/await 后续逻辑）；
- 手动调用的同步代码块。

示例（React17 中不支持批处理的场景）：

```
Code block
function App() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('');

  useEffect(() => {
    // setTimeout 中不支持批处理，触发 2 次渲染
    setTimeout(() => {
      setCount(1);
      setName('xxx');
    }, 0);
  }, []);

  console.log('组件渲染'); // 输出 2 次
  return <div>{count} - {name}</div>;
}
```

3. 自动批处理（Automatic Batching）的优化：突破场景限制

React 18 引入的自动批处理，核心优化是「将批处理的范围扩展到所有场景」，无论状态更新发生在 React 控制的环境还是非 React 控制的环境，默认都进行批处理，从而彻底解决旧版的冗余渲染问题。

（1）自动批处理支持的场景（全覆盖）

- 保留旧版支持的场景（合成事件、生命周期、Hooks 回调）；
- 新增支持场景：原生事件、定时器、Promise 回调、async/await 等所有异步/同步场景。

示例（React18 中自动批处理）：

```
Code block

function App() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('');

  useEffect(() => {
    // setTimeout 中自动批处理，仅触发 1 次渲染
    setTimeout(() => {
      setCount(1);
      setName('xxx');
    }, 0);
  }, []);

  console.log('组件渲染'); // 输出 1 次
  return <div>{count} - {name}</div>;
```

```
16  }
```

（2）如何启用自动批处理？

自动批处理默认启用，但需使用 React18 新增的根节点渲染 API（createRoot）：

Code block

```
1
2   // 启用自动批处理（推荐）
3   const root = ReactDOM.createRoot(document.getElementById('root'));
4   root.render(<App />);
5
6   // 旧版 render API（不启用自动批处理，兼容旧代码）
7   ReactDOM.render(<App />, document.getElementById('root'));
```

（3）特殊需求：强制同步更新

若需在某些场景下避免批处理（强制同步更新），可使用 ReactDOM.flushSync 包裹状态更新：

Code block

```
1
2   import { flushSync } from 'react-dom';
3
4   setTimeout(() => {
5     flushSync(() => {
6       setCount(1); // 强制同步更新，触发 1 次渲染
7     });
8     setName('xxx'); // 单独触发 1 次渲染
9   }, 0);
```

4. 自动批处理如何改善性能？

自动批处理通过以下两个核心方式提升 React 应用性能：

- （1）减少渲染次数：将分散在不同场景的多个状态更新合并为一次渲染，避免了旧版非 React 控制场景下的多次冗余渲染，减少了虚拟 DOM 对比和真实 DOM 更新的次数；
- （2）降低主线程开销：真实 DOM 操作（重排/重绘）是浏览器主线程的高开销任务，自动批处理减少了 DOM 操作次数，降低了主线程的负担，避免了因频繁渲染导致的页面卡顿，尤其在复杂组件树和频繁状态更新场景下（如列表筛选、表单联动）效果显著。

总结：自动批处理是 React18 对批处理机制的“全场景升级”，无需开发者手动优化，即可默认获得性能提升，是 React18 提升应用响应速度的重要特性之一。

四十、React 18 引入了新的协调器（Concurrent Mode）。请从源码角度解释 Concurrent Mode 的工作原理是什么？

注：React 18 中，“Concurrent Mode”（并发模式）已不再是明确的“模式”，而是升级为「并发渲染能力」（Concurrent Rendering），通过底层协调器（Reconciler）的重构实现，核心依赖「Scheduler（调度器）」「Fiber 架构」「Lane 优先级模型」三大核心模块。从源码角度，其工作原理可拆解为“优先级调度-可中断协调-提交确认”三个核心阶段，以下结合源码核心逻辑展开说明：

核心前提：并发渲染能力默认不开启，需通过 createRoot 启用，本质是让协调器具备“中断、暂停、恢复、放弃渲染”的能力，优先处理高优先级任务。

1. 源码核心依赖模块

并发渲染的实现依赖 React 源码中三个核心模块的协同工作：

- （1）Scheduler（调度器）：源码位置 `react-scheduler`，负责任务的优先级排序、时间切片管理（默认 16ms），判断任务是否需要中断；
- （2）Reconciler（协调器）：源码位置 `react-reconciler`，基于 Fiber 架构，负责组件树的遍历（可中断）、Diff 算法执行、标记更新类型（effectTag）；
- （3）Lane 模型（优先级模型）：源码中通过二进制位表示任务优先级（如 ImmediateLane 最高，IdleLane 最低），用于标记每个更新任务的优先级，决定任务的执行顺序。

2. 工作原理核心流程（源码视角）

（1）阶段一：任务调度与优先级标记（Scheduler + Lane 模型）

当触发状态更新（如 setState、useState 更新）时，React 源码会执行以下步骤：

1. 创建更新对象（Update）：封装更新的状态值、回调函数等信息；
2. 通过 Lane 模型为更新任务分配优先级：根据更新类型（如用户输入为高优先级，数据加载为低优先级），从 Lane 优先级列表中分配对应的 Lane（二进制位）；
3. 将更新对象加入对应 Fiber 节点的更新队列（updateQueue）；
4. 调用 Scheduler 的 `scheduleCallback` 方法，将协调任务（Reconciliation）加入调度队列，等待执行。

源码关键逻辑：Lane 模型通过二进制位运算高效判断任务优先级，例如高优先级任务的 Lane 会覆盖低优先级任务，确保高优先级任务优先执行。

（2）阶段二：可中断协调（Reconciler + Fiber 架构）

调度器触发协调任务后，协调器基于 Fiber 架构开始遍历组件树（深度优先），此过程可被中断，核心源码逻辑是 `workLoop` 函数（可中断的循环遍历）：

Code block

```
// 简化版 workLoop 源码逻辑（react-reconciler 中）
function workLoop(hasTimeRemaining, initialLane) {
  let current = workInProgress; // 当前正在处理的 Fiber 节点
  while (current !== null && !shouldYield()) { // shouldYield() 由 Scheduler 提供，判断是否需要中断
    // 处理当前 Fiber 节点（执行 Diff、标记 effectTag）
    performUnitOfWork(current);
    // 移动到下一个 Fiber 节点（基于 Fiber 链表：child > sibling > return）
    current = getNextUnitOfWork(current);
  }
  // 若任务未完成（current !== null），则再次调度，等待恢复执行
  if (current !== null) {
    scheduleCallback(performConcurrentWorkOnRoot.bind(null, root, lanes));
  }
}
```

关键细节：

- 可中断判断：`shouldYield()` 函数由 Scheduler 提供，判断当前时间切片是否用尽（默认 16ms）或是否有更高优先级任务插入；若需要中断，立即停止遍历，保存当前 Fiber 节点（workInProgress）的状态；
- 任务恢复：当高优先级任务执行完成或时间切片重新分配后，调度器会再次调用 `workLoop`，从上次中断的 Fiber 节点继续遍历；
- 任务放弃：若在中断期间，有更高优先级的更新任务覆盖了当前任务（通过 Lane 模型判断），则直接放弃当前未完成的协调任务，重新开始高优先级任务的协调。

（3）阶段三：提交阶段（不可中断，Renderer 参与）

当协调阶段完成（所有 Fiber 节点遍历完成，Diff 对比结束，标记完所有 effectTag），React 会进入提交阶段（Commit），此阶段不可中断（避免 DOM 状态不一致），核心源码逻辑是 `commitRoot` 函数：

1. 执行前置清理逻辑：如调用组件的 `getSnapshotBeforeUpdate` 生命周期；
2. 执行 DOM 操作：根据 Fiber 节点的 effectTag（如插入、更新、删除），批量执行真实 DOM 操作；
3. 执行后置逻辑：如调用组件的 `componentDidMount` / `componentDidUpdate` 生命周期、useEffect 回调、更新任务的回调函数；
4. 切换 Fiber 树：将 workInProgress 树（协调阶段构建的新树）切换为 current 树（当前渲染的树），完成渲染更新。

3. 核心优化点：时间切片与优先级抢占

从源码角度，并发渲染的核心优化源于两个关键机制：

- （1）时间切片（Time Slicing）：Scheduler 基于 `requestIdleCallback` 模拟实现时间切片（兼容不支持该 API 的浏览器），确保每次协调任务的执行时间不超过 16ms（对应 60fps 的刷新率），避免长时间占用主线程；
- （2）优先级抢占：通过 Lane 模型标记任务优先级，高优先级任务可在低优先级任务执行过程中“插队”（Scheduler 触发中断），低优先级任务需等待高优先级任务完成后才能恢复执行，确保用户输入、动画等关键操作的响应速度。

4. 源码层面的关键设计：双缓存 Fiber 树

并发渲染依赖 Fiber 架构的“双缓存”设计（源码中通过 `current` 和 `workInProgress` 两个 Fiber 树实现）：

- current 树：对应当前页面渲染的 DOM 结构，不可修改；
- workInProgress 树：协调阶段构建的新树，可中断修改，包含最新的组件状态和 DOM 变化。

优势：协调阶段的中断/修改不会影响 current 树，确保页面渲染状态稳定；提交阶段仅需切换两棵树的引用，即可完成 DOM 更新，高效且安全。

5. 总结（源码视角）

React 18 的并发渲染（原 Concurrent Mode）本质是通过「Scheduler 调度优先级任务」「Reconciler 执行可中断协调」「Lane 模型标记优先级」「双缓存 Fiber 树保障稳定性」的协同工作，实现“高优先级任务优先执行，低优先级任务可中断等待”的渲染能力。从源码来看，核心是对协调器（Reconciler）的重构，通过 `workLoop` 可中断循环、`shouldYield` 中断判断、Lane 优先级标记等逻辑，让 React 能够灵活控制渲染流程，避免主线程阻塞，最终提升应用的响应速度和用户体验。

（注：文档部分内容可能由 AI 生成）