前端进阶面试知识体系（React / 算法编程 / TS / Vue / 性能 / 工程化）

> 本文档针对前端进阶面试题，按 **6 大主题**（React、算法编程题、TypeScript、Vue、性能优化、工程化）系统梳理，每题给出**核心原理 + 解决方案 + 代码示例**。适合 3-8 年经验面试复习与日常知识沉淀。 。

目录

一、React（23 题）

Q1：Fiber 架构的工作原理？

核心要点：Fiber 是 React 16 引入的**协调引擎（Reconciliation Engine）**，将渲染工作拆分成可中断、可恢复的最小工作单元。

工作原理：

1. **数据结构**：每个 Fiber 节点对应一个 React 元素，包含 `type`、`key`、`child`、`sibling`、`return`、`pendingProps`、`memoizedProps`、`memoizedState`、`alternate`（双缓冲）等字段，形成一棵 Fiber 树。
2. **双缓冲机制**：维护两棵 Fiber 树（`current` 和 `workInProgress`），更新时构建 `workInProgress`，完成后整体切换，减少页面闪烁。
3. **两阶段渲染**：
   - **Render 阶段（可中断）**：调用 `beginWork` 向下遍历，构建新 Fiber 树，标记副作用（Placement/Update/Deletion）。
   - **Commit 阶段（不可中断）**：将副作用一次性提交到真实 DOM，调用 `useEffect`/`useLayoutEffect` 回调。
4. **时间切片（Time Slicing）**：通过 `MessageChannel` 或 `scheduler` 包将渲染任务切成 5ms 小片，每帧 `requestIdleCallback` 思想交还主线程。

简化流程图：

```
scheduleUpdate → scheduleWork → workLoop (yield when no time)
    → beginWork (down) → completeWork (up) → commitWork (sync)
```

Q2：ReactReconciler 为何要采用 Fiber 架构?

原因总结：

1. **解决同步阻塞问题**：React 15 之前采用 Stack Reconciler，递归调用栈一旦开始无法中断，长任务（>16ms）会阻塞主线程，导致卡顿、掉帧。
2. **支持优先级调度**：Fiber 节点可携带 `lanes`（优先级位），实现高优先级更新（用户输入）打断低优先级更新（数据加载）。
3. **支持并发特性**：为 Concurrent Mode、`useTransition`、`useDeferredValue` 等 React 18 特性提供底层支撑。
4. **更好的错误恢复**：可中断后从断点恢复，结合 Error Boundary 提升健壮性。

Q3：useState 是如何实现的?

实现原理（简化源码）：

```
// 挂载阶段
function mountState(initialState) {
  const hook = mountWorkInProgressHook(); // 创建 hook 节点挂到 fiber.memoizedState
链表
  hook.memoizedState = typeof initialState === 'function' ? initialState() :
initialState;
  hook.queue = { pending: null, dispatch: null };
  const dispatch = dispatchAction.bind(null, currentlyRenderingFiber, hook.queue);
  hook.queue.dispatch = dispatch;
  return [hook.memoizedState, dispatch];
}

// 更新阶段
function dispatchAction(fiber, queue, action) {
  const update = { action, next: null, lane: requestUpdateLane() };
  // 链表追加 update
  enqueueRenderPhaseUpdate(queue, update);
  // 调度更新
  scheduleUpdateOnFiber(fiber, update.lane);
}
```

关键点：

- Hook 以**链表**形式存储在 Fiber 节点的 `memoizedState` 上；
- `dispatch` 通过调度器触发更新；
- 多次 `setState` 在同一事件中会**批处理**（React 18 自动批处理，Promise/setTimeout 中也合并）。

Q4：React Fiber 是什么?

定义：Fiber 是 React 内部的**协调算法 + 数据结构**，是对"虚拟 DOM 节点"的升级版。

两个层面的含义：

1. **架构层面**：新的协调引擎（Reconciler）。
2. **数据结构层面**：每个 Fiber 节点是一个 JS 对象，包含组件信息、DOM 信息、工作状态、副作用链表等。

核心属性：

属性	说明
`type` / `elementType`	组件类型（'div' / FunctionComponent / ClassComponent）
`key`	Diff 时用于识别节点
`stateNode`	真实 DOM / 类实例
`return` / `child` / `sibling`	树结构指针
`pendingProps` / `memoizedProps`	新旧 props
`memoizedState`	Hook 链表 / 类组件 state
`flags` / `subtreeFlags`	副作用标记（Placement/Update/ChildDeletion）
`lanes` / `childLanes`	优先级
`alternate`	指向另一棵树的对应节点（双缓冲）

Q5：简单介绍 React 中的 diff 算法

三大策略（将 O(n^3) 降为 O(n)）：

1. **Tree Diff（树级）**：同层比较，不跨层级移动节点。若发现节点跨层移动，**直接删除重建**。
2. **Component Diff（组件级）**：
   - 同类型组件：继续往下 diff；
   - 不同类型组件：直接替换整棵子树。
3. **Element Diff（元素级）**：通过 `key` 标识同级元素的唯一性。
   - 有 `key`：用 key 匹配节点，移动位置而非重建；
   - 无 `key`：按顺序逐个对比，效率低且易出错。

最佳实践：列表渲染务必提供稳定且唯一的 `key`（建议用业务 ID，不要用 index）。

Q6：如何让 useEffect 支持 async/await?

直接写法（错误）：

```
useEffect(async () => { // ✕ 返回 Promise, React 不会等待
  await fetchData();
}, []);
```

正确写法：

```
// 方案1：内部定义 async 函数
useEffect(() => {
  const load = async () => {
    const data = await fetchData();
    setData(data);
  };
  load();
}, []);

// 方案2：抽成函数
const fetchData = async () => { /* ... */ };
useEffect(() => { fetchData(); }, []);
```

原因：`useEffect` 的回调要么返回 `undefined`，要么返回**清理函数**（cleanup）。`async` 函数返回 Promise，会被当作"清理函数"忽略，导致清理逻辑失效。

可选优化：使用 `AbortController` 取消请求：

```
useEffect(() => {
  const ctrl = new AbortController();
  fetch(url, { signal: ctrl.signal }).then(...);
  return () => ctrl.abort();
}, [url]);
```

Q7：React Fiber 是如何实现更新过程可控？

可控性的三大支柱：

1. **可中断**：每次执行一个 Fiber 单元后检查 `shouldYield()`，时间片用完就 `return`，把控制权交回调度器。
2. **可恢复**：通过 `workInProgress` 记住当前进度，下次从断点 Fiber 继续 `beginWork`。
3. **可优先级**：每个更新携带 `lane`（优先级），高优先级插入队列时可**插队**抢占低优先级。

实现关键：

```
function workLoopConcurrent() {
  while (workInProgress !== null && !shouldYield()) {
    workInProgress = performUnitOfWork(workInProgress);
  }
}

function shouldYield() {
  return performance.now() >= deadline; // 5ms 时间片
}
```

调度器基于 `MessageChannel` 实现，比 `setTimeout(fn, 0)` 更稳定，比 `requestIdleCallback` 兼容性更好。

Q8：React 中懒加载的实现原理是什么？

核心 API：`React.lazy` + `Suspense`。

原理：

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function App() {
  return (
    <Suspense fallback={<Loading />}>
      <OtherComponent />
    </Suspense>
  );
}
```

1. `React.lazy` 接收返回 **Promise** 的动态 `import()`；
2. 首次渲染时 React 抛出 Promise，进入 Suspense 的 fallback；
3. Promise resolve 后重新渲染，替换为真实组件；
4. 底层通过 **Throw Promise** 模式让渲染可中断，等模块加载完成再恢复。

路由级懒加载（React Router v6）：

```
const Home = lazy(() => import('./pages/Home'));
<Route path="/" element={<Suspense fallback={<Loading />}><Home /></Suspense>} />
```

Q9：React 中怎么实现状态自动保存（KeepAlive）？

场景：列表页 → 详情页 → 返回时保留列表滚动位置和筛选条件。

方案对比：

方案	实现方式	适用场景
缓存组件状态	外部 Store（Zustand/Redux）保存状态	中小型项目
手动缓存	`useRef` + `useImperativeHandle` 暴露 reset	单组件
第三方库	`react-activation` / `react-keep-alive-router`	完整方案
DOM 节点缓存	隐藏的 `display: none` 容器保留 DOM	简单场景

react-activation 示例：

```
import { KeepAlive } from 'react-activation';

<KeepAlive name="list-page" saveScrollPosition>
  <ListPage />
</KeepAlive>
```

底层通过 Portal 将组件挂载到隐藏容器，路由切换时不卸载。

Q10：React 有哪些性能优化的方法?

按层级:

1. 减少渲染次数

- `React.memo` / `PureComponent` 浅比较 props;
- `useMemo` / `useCallback` 缓存引用;
- 状态下推（拆分组件，缩小影响范围）;
- Context 拆分（细粒度 Context）。

2. 减少渲染计算量

- 虚拟列表（`react-window`、`react-virtuoso`）;
- 防抖/节流高频事件;
- Web Worker 处理 CPU 密集任务。

3. 减少包体积

- 路由懒加载 + 组件懒加载;
- Tree Shaking;
- 按需引入第三方库;
- 压缩图片（WebP/AVIF）。

4. 网络优化

- CDN、HTTP 缓存、Service Worker;
- 接口合并、GraphQL;
- 预加载（`<link rel="preload">`）。

5. 渲染层

- SSR/SSG（Next.js）;
- Suspense 流式渲染。

Q11：不同版本的 React 都做过哪些优化?

版本	关键优化
React 15	Stack Reconciler，diff 算法奠基

版本	关键优化
React 16	Fiber 架构、Error Boundaries、Fragment、Portal、createRef
React 16.3	new Context API、`React.createRef`、`React.forwardRef`
React 16.6	`React.lazy` + `Suspense`、`memo`、`forwardRef`
React 16.8	Hooks 革命（useState/useEffect/useContext 等）
React 17	渐进式升级、新 JSX 运行时（`react/jsx-runtime`）、事件委托到根容器
React 18	Concurrent Renderer、自动批处理、Transitions（useTransition / useDeferredValue）、Suspense SSR 流式、useId、useSyncExternalStore
React 19（实验）	Actions、useOptimistic、`use()` API、`ref` 作为 prop、`<form>` Action、`<title>` 文档元数据

Q12：React 18 新特性

1. **Concurrent Features（并发渲染）**
   - `createRoot` API 启用并发模式；
   - `useTransition`：标记非紧急更新；
   - `useDeferredValue`：延迟更新值。
2. **自动批处理**：所有更新（包括 Promise、setTimeout、原生事件处理器）合并渲染。
3. **Suspense 改进**：
   - 服务端 Suspense 流式 HTML（Streaming SSR）；
   - `<SuspenseList>` 协调多个 Suspense。
4. **新 Hook**：`useId`、`useSyncExternalStore`、`useInsertionEffect`。
5. **严格模式增强**：开发模式下组件二次挂载/卸载，复现副作用问题。
6. **Server Components（RSC）**：服务端组件，零客户端 JS。

Q13：React Hook 的闭包陷阱是什么？解决方案？

陷阱示例：

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

  useEffect(() => {
    const id = setInterval(() => {
      console.log(count); // 永远打印 0，闭包陷阱
      setCount(count + 1); // 永远 +1（基于 0）
    }, 1000);
    return () => clearInterval(id);
  }, []); // 空依赖，count 永远是最初的 0
}
```

解决方案：

1. **函数式更新（推荐）**：

```
setCount(c => c + 1); // 总是基于最新值
```

2. **使用 ref 保存最新值**：

```
const countRef = useRef(count);
useEffect(() => { countRef.current = count; }, [count]);
// 定时器内使用 countRef.current
```

3. **正确声明依赖**：`useEffect(..., [count])`，但可能造成定时器反复创建/销毁。

根因：每次渲染都会创建一个新的 effect 函数，effect 捕获的是创建时的 props/state。

Q14：React 中，怎么给 children 添加额外的属性?

方案 1：React.cloneElement（最常见）

```
function MyContainer({ children }) {
  return React.Children.map(children, child =>
    React.cloneElement(child, { extraProp: 'value' })
  );
}
```

方案 2：Context 透传（更优雅）

```
const ExtraCtx = createContext({});
function Parent({ children }) {
  const extra = { theme: 'dark' };
  return <ExtraCtx.Provider value={extra}>{children}</ExtraCtx.Provider>;
}
```

方案 3：HOC / 自定义 Hook 包装

方案 4：React 19 直接 `ref` 作为 prop，未来属性传递将更自然。

Q15：Fiber 为什么是 React 性能的一个飞跃?

飞跃点：

1. **从同步到异步**：Stack Reconciler 必须一次性跑完，Fiber 可中断，让出主线程给高优先级任务。

2. **从单优先级到多优先级**：基于 `lane` 模型实现 5+ 种优先级，用户输入、动画、数据加载可差异化调度。
3. **从一次性提交到分阶段**：Render/Commit 分离，DOM 变更可控，避免半成品状态。
4. **为并发特性铺路**：`useTransition`、`Suspense`、`Server Components` 全部依赖 Fiber。
5. **降低主线程压力**：长列表、大状态树更新不再卡顿，60fps 成为可能。

Q16：React 是否支持给标签设置自定义的属性?

支持，但**不推荐**。

两种方式：

1. **HTML 自定义属性**：原生支持，`data-*` 推荐使用，可通过 `dataset` 访问。
2. **非标准属性**：React 会**原样传递**到 DOM 上（除了 `key`/`ref`/`dangerouslySetInnerHTML` 等特殊属性）。

```
<div custom-attr="hello" data-id="123" />
```

注意事项：

- 会触发 React 警告 `Unknown prop "custom-attr" on <a> tag`（如需消除，加 `// @ts-ignore` 或扩展 JSX 类型）；
- 推荐优先用 `data-*`，符合 HTML5 规范且语义清晰；
- 复杂场景建议用 Context 或全局状态管理。

Q17：说说 React render 阶段的执行过程

核心流程：

1. **入口**：`scheduleUpdateOnFiber` → 调度器根据 `lane` 决定立即/延后执行；
2. **构建 workInProgress 树**：
   - 复用 `current` 节点（通过 `alternate`）；
   - 调用 `beginWork(current, workInProgress, renderLanes)` 处理当前 Fiber；
3. **beginWork 内部**：
   - 函数组件：执行函数体，收集 Hook；
   - 类组件：调用 `render` 方法；
   - 调和子节点，标记 Diff（flags）；
4. **completeWork**：向上回溯，完成 Fiber 节点信息收集，构建 effect 链表；
5. **循环**：在 `workLoopConcurrent` 中通过 `shouldYield()` 检查时间片，用完则暂停。

特点：纯 JS 计算，不操作 DOM，可随时中断。

Q18：React 中，Fiber 是如何实现时间切片的?

实现机制：

1. **最小工作单元**：单个 Fiber 节点（`performUnitOfWork`）；

2. **时间片控制**：
   - 调度器设置 `deadline = performance.now() + 5ms`；
   - 每完成一个 Fiber 调用 `shouldYield()` 检查是否超时；
   - 超时则退出 workLoop，保留 `workInProgress` 指针；
3. **恢复机制**：浏览器空闲后（通过 `MessageChannel.postMessage`）重新进入 workLoop，从上次断点继续；
4. **底层调度器**：`scheduler` 包实现了基于优先级队列和最小堆的调度算法。

为什么不直接用 `requestIdleCallback`：

- 兼容性差（仅 Chrome 支持）；
- 最小延迟 50ms 太长；
- 无法精细控制优先级。

Q19：说说 React commit 阶段的执行过程

Commit 阶段三个子阶段：

1. **BeforeMutation**（DOM 变更前）
   - 调用类组件 `getSnapshotBeforeUpdate`；
   - 调度 `useEffect` 销毁函数。
2. **Mutation**（DOM 变更）
   - 执行所有 DOM 插入/更新/删除；
   - 绑定/解绑 ref；
   - 调用类组件 `componentDidMount`/`componentDidUpdate`。
3. **Layout**（DOM 变更后、浏览器绘制前）
   - 调用 `useLayoutEffect` 回调；
   - 调度 `useEffect` 创建函数（异步，避免阻塞绘制）。

关键特性：

- 同步执行，**不可中断**（保证 DOM 一致性）；
- 整个过程通过 `commitRoot` 一次性提交 effect 链表。

Q20：React 中的路由懒加载是什么？原理？

概念：按路由拆分代码块，访问对应路由时才加载对应组件的 JS。

实现：

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

const Home = lazy(() => import(/* webpackChunkName: "home" */ './pages/Home'));
const About = lazy(() => import(/* webpackChunkName: "about" */ './pages/About'));
```

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

原理：

1. `import()` 触发 webpack/vite 拆包（自动生成单独 chunk）；
2. `React.lazy` 把模块包装成懒加载组件；
3. 首次渲染时 React 抛 Promise，进入 Suspense fallback；
4. 模块加载完成 resolve 后重新渲染。

优势：首屏 JS 体积显著减少，加快 FCP/TTI。

Q21：React 中为什么不直接使用 requestIdleCallback?

原因：

1. **兼容性问题**：仅 Chrome/Firefox 支持，Safari/Edge 早期不支持；
2. **最大延迟 50ms**：对于需要快速响应的场景（动画、输入），延迟过高；
3. **无法精细优先级**：`requestIdleCallback` 只区分 idle 与否，无法满足多优先级调度；
4. **无法可靠触发**：浏览器在标签页非激活时可能**完全停止**调用，影响数据一致性。

React 的替代方案：

- 使用 `MessageChannel` 宏任务；
- `scheduler` 包实现自定义优先级队列；
- 通过 `shouldYield()` + `performance.now()` 控制时间片。

Q22：React 组件间怎么进行通信?

通信方式汇总：

场景	方式
父子	`props` / 回调函数
子父	回调函数 / `ref` 暴露实例
兄弟	状态提升到共同父组件 / Context

场景	方式
跨层级	Context API / Redux / Zustand
任意组件	全局状态（Redux / MobX / Zustand / Jotai）
深层组件	Composition（children / render props）
命令式	`useImperativeHandle` + `forwardRef`
外部系统	`useSyncExternalStore` 订阅外部 Store

最佳实践：

- 优先用 props / composition；
- 跨多层级才考虑 Context；
- 复杂应用用 Redux Toolkit / Zustand；
- 避免过度使用 Context（导致不必要的 re-render）。

Q23：React 和 Vue 在技术层面有哪些区别？

维度	React	Vue 3
范式	函数式 + JSX，强调"UI = f(state)"	选项式 + 组合式，渐进式框架
响应式	不可变数据 + 显式 `setState`	Proxy 代理 + 自动依赖收集
Diff	单端比对	双端对比 + 最长递增子序列
编译优化	React Compiler（实验）	模板编译 + 静态提升 + 补丁标志
Fiber/调度	必备	不需要（响应式粒度更细）
生态	庞大、社区驱动	官方全家桶（Router/Pinia/Vite）
学习曲线	较陡（Hooks、心智模型）	较平缓（模板直观）
TypeScript	一流支持	良好支持（3.x 改进）
包体积	较小核心（~44KB gzip）	较小核心（~34KB gzip）
服务端	Server Components / Next.js	Nuxt 3

本质差异：

- React 假设"一切皆组件"，状态更新需要手动 setState；
- Vue 通过 Proxy 自动追踪依赖，更新粒度更细，不需要 Fiber 也能高效更新。

二、算法编程题（27 题）

P1：使用 Promise 实现红绿灯交替重复

```
function trafficLight(times: number, order: string[] = ['red', 'yellow', 'green'])
{
  const delay = (ms: number) => new Promise(res => setTimeout(res, ms));
  const lights: Record<string, number> = { red: 3000, yellow: 1000, green: 2000 };

  return new Promise<void>(async resolve => {
    for (let i = 0; i < times; i++) {
      for (const color of order) {
        console.log(color);
        await delay(lights[color]);
      }
    }
    resolve();
  });
}

// 递归写法（更直观）
function loop(times: number) {
  const tick = (count: number): Promise<void> => {
    if (count >= times) return Promise.resolve();
    return new Promise(res => setTimeout(res, 1000))
      .then(() => { console.log('🟢'); return new Promise(r => setTimeout(r,
1000)); })
      .then(() => { console.log('🟡'); return new Promise(r => setTimeout(r,
1000)); })
      .then(() => { console.log('🔴'); })
      .then(() => tick(count + 1));
  };
  return tick(0);
}
```

关键点：`setTimeout` 包成 Promise，用 `async/await` 或 Promise 链按顺序触发。

P2：bind、call、apply 有什么区别？如何手写 bind？

区别：

方法	参数形式	调用时机	返回值
`call(obj, arg1, arg2, ...)`	逐个参数	立即调用	函数返回值
`apply(obj, [args])`	数组	立即调用	函数返回值
`bind(obj, arg1, ...)`	逐个参数	返回新函数	绑定 this 的新函数

手写 call：

```
Function.prototype.myCall = function(thisArg: any, ...args: any[]) {
  const fn = Symbol('fn'); // 唯一 key 避免覆盖
  thisArg = thisArg ?? globalThis;
```

```
  thisArg[fn] = this;
  const result = thisArg[fn](...args);
  delete thisArg[fn];
  return result;
};
```

手写 apply：

```
Function.prototype.myApply = function(thisArg: any, args: any[] = []) {
  const fn = Symbol('fn');
  thisArg = thisArg ?? globalThis;
  thisArg[fn] = this;
  const result = thisArg[fn](...args);
  delete thisArg[fn];
  return result;
};
```

手写 bind（考点，需考虑参数合并 + new 调用）：

```
Function.prototype.myBind = function(thisArg: any, ...presetArgs: any[]) {
  const self = this;
  return function Fn(...laterArgs: any[]) {
    // 用 new 调用时 this 应指向新对象
    const isNew = this instanceof Fn;
    return self.apply(isNew ? this : thisArg, [...presetArgs, ...laterArgs]);
  };
};
```

P3：字符串压缩（利用字符重复次数）

```
function compress(str: string): string {
  if (!str) return '';
  let result = '';
  let count = 1;
  for (let i = 0; i < str.length; i++) {
    if (str[i] === str[i + 1]) {
      count++;
    } else {
      result += str[i] + count;
      count = 1;
    }
  }
  return result.length < str.length ? result : str;
}
```

示例：`'aabcccccaaa'` → `'a2b1c5a3'`

P4：new 操作符具体干了什么？

四步执行：

1. 创建空对象 `obj = {}`；
2. 设置原型：`obj.__proto__ = Constructor.prototype`；
3. 执行构造函数（`this` 指向 `obj`），为 `obj` 添加属性；
4. **判断返回值**：
   - 显式返回对象 → 返回该对象；
   - 其他情况 → 返回 `obj`。

手写实现：

```
function myNew(Constructor: Function, ...args: any[]) {
  const obj = Object.create(Constructor.prototype);
  const result = Constructor.apply(obj, args);
  return result instanceof Object ? result : obj;
}
```

P5：如何实现上拉加载、下拉刷新？

核心原理：

```
// 上拉加载：监听 scroll，判断距离底部
function onScroll() {
  const { scrollTop, clientHeight, scrollHeight } = document.documentElement;
  if (scrollTop + clientHeight >= scrollHeight - 50) {
    loadMore(); // 触发加载
  }
}

// 下拉刷新：touch 事件计算下拉距离
let startY = 0, pulling = false;
el.addEventListener('touchstart', e => {
  if (window.scrollY === 0) {
    startY = e.touches[0].clientY;
    pulling = true;
  }
});
el.addEventListener('touchmove', e => {
  if (!pulling) return;
  const dy = e.touches[0].clientY - startY;
  if (dy > 0) {
    e.preventDefault(); // 阻止默认滚动
    el.style.transform = `translateY(${dy}px)`;
  }
```

```
  });
  el.addEventListener('touchend', e => {
    if (pulling) {
      pulling = false;
      el.style.transform = '';
      // 触发刷新
      refresh().finally(() => {});
    }
  });
```

推荐方案：直接用第三方库（如 `better-scroll`、`vue-virtual-scroller`、React 的 `react-pull-to-refresh`）。

P6：大文件怎么实现断点续传？

核心流程：

1. **文件分片**：使用 `Blob.slice()` 将文件切成固定大小（5MB）的 chunks；
2. **计算唯一标识**：`spark-md5` 计算文件 hash（避免全量上传）；
3. **秒传校验**：上传前查询服务端是否已有该 hash 的文件，有则秒传；
4. **分片上传**：每个 chunk 携带 `index`、`hash`、文件 `hash`；
5. **断点续传**：上传前查询服务端已有分片，跳过已上传部分；
6. **合并请求**：所有分片上传完成后通知服务端合并。

关键代码骨架：

```
async function upload(file: File) {
  const CHUNK_SIZE = 5 * 1024 * 1024;
  const chunks = [];
  for (let i = 0; i < file.size; i += CHUNK_SIZE) {
    chunks.push(file.slice(i, i + CHUNK_SIZE));
  }
  const fileHash = await calcHash(chunks);
  const { uploaded = [] } = await api.checkFile(fileHash);

  const tasks = chunks
    .map((chunk, idx) => ({ idx, chunk }))
    .filter(({ idx }) => !uploaded.includes(idx))
    .map(({ idx, chunk }) =>
      api.uploadChunk(chunk, fileHash, idx).retry(3)
    );

  // 限制并发 6
  await pLimit(6, tasks);
  await api.mergeChunks(fileHash);
}
```

P7：防抖与节流的编码实现

防抖（debounce）：触发后等待 `wait` ms，若再次触发则重置。

```
function debounce<T extends (...args: any[]) => any>(fn: T, wait: number) {
  let timer: any = null;
  return function (this: any, ...args: Parameters<T>) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), wait);
  };
}
```

节流（throttle）：固定时间内只执行一次（首触发或尾触发）。

```
// 时间戳版（首触发立即执行）
function throttle<T extends (...args: any[]) => any>(fn: T, wait: number) {
  let last = 0;
  return function (this: any, ...args: Parameters<T>) {
    const now = Date.now();
    if (now - last >= wait) {
      last = now;
      fn.apply(this, args);
    }
  };
}

// 定时器版（尾触发）
function throttle2(fn: Function, wait: number) {
  let timer: any = null;
  return function (this: any, ...args: any[]) {
    if (!timer) {
      timer = setTimeout(() => {
        timer = null;
        fn.apply(this, args);
      }, wait);
    }
  };
}
```

使用场景：防抖 → 搜索输入、resize；节流 → 滚动、mousemove、点击。

P8：AJAX 的原理与实现

原理：通过 `XMLHttpRequest` 或 `Fetch` 异步与服务器交换数据，局部刷新页面。

手写 XHR：

```
function ajax({ url, method = 'GET', data, headers = {} }: any) {
  return new Promise((resolve, reject) => {
```

```
    const xhr = new XMLHttpRequest();
    xhr.open(method, url, true);
    Object.entries(headers).forEach(([k, v]) => xhr.setRequestHeader(k, v as
string));
    xhr.onload = () => {
      if (xhr.status >= 200 && xhr.status < 300) {
        resolve(JSON.parse(xhr.responseText));
      } else {
        reject(new Error(xhr.statusText));
      }
    };
    xhr.onerror = () => reject(new Error('Network Error'));
    xhr.send(data ? JSON.stringify(data) : null);
  });
}
```

Fetch 版本：

```
async function fetchJSON(url: string, options: RequestInit = {}) {
  const res = await fetch(url, options);
  if (!res.ok) throw new Error(`${res.status} ${res.statusText}`);
  return res.json();
}
```

P9：深拷贝浅拷贝区别与实现

区别：

- **浅拷贝**：只复制对象的第一层属性，引用类型仍共享同一地址（`Object.assign`、`{...obj}`、`slice`）。
- **深拷贝**：递归复制所有层级，新对象与原对象完全隔离。

深拷贝实现：

```
function deepClone<T>(obj: T, map = new WeakMap()): T {
  if (obj === null || typeof obj !== 'object') return obj;
  if (obj instanceof Date) return new Date(obj) as any;
  if (obj instanceof RegExp) return new RegExp(obj.source, obj.flags) as any;
  if (map.has(obj as any)) return map.get(obj as any);

  const cloneObj: any = Array.isArray(obj) ? [] : {};
  map.set(obj as any, cloneObj);
  for (const key in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, key)) {
      cloneObj[key] = deepClone((obj as any)[key], map);
    }
  }
```

```
  return cloneObj;
}
```

生产推荐：`structuredClone(obj)`（原生）、`lodash.cloneDeep`（处理更多类型）。

注意：函数、Symbol、原型链、循环引用、特殊对象（Map/Set）需特别处理。

P10：JS 实现二叉树与基本操作

```
class TreeNode {
  val: number;
  left: TreeNode | null = null;
  right: TreeNode | null = null;
  constructor(val: number) { this.val = val; }
}

// 遍历
const preorder = (root: TreeNode | null): number[] => {
  if (!root) return [];
  return [root.val, ...preorder(root.left), ...preorder(root.right)];
};
const inorder = (root: TreeNode | null): number[] =>
  !root ? [] : [...inorder(root.left), root.val, ...inorder(root.right)];
const postorder = (root: TreeNode | null): number[] =>
  !root ? [] : [...postorder(root.left), ...postorder(root.right)];

// 层序（BFS）
const levelOrder = (root: TreeNode | null): number[][] => {
  if (!root) return [];
  const result: number[][] = [];
  const queue: TreeNode[] = [root];
  while (queue.length) {
    const level: number[] = [];
    const size = queue.length;
    for (let i = 0; i < size; i++) {
      const node = queue.shift()!;
      level.push(node.val);
      if (node.left) queue.push(node.left);
      if (node.right) queue.push(node.right);
    }
    result.push(level);
  }
  return result;
};
```

P11：实现轮播图组件

核心要点：

```
<!-- Vue 版 -->
<template>
  <div class="carousel" @mouseenter="pause" @mouseleave="play">
    <div class="track" :style="{ transform: `translateX(-${index * 100}%)` }">
      <div v-for="(img, i) in images" :key="i" class="slide">
        <img :src="img" />
      </div>
    </div>
    <button @click="prev">‹</button>
    <button @click="next">›</button>
    <div class="dots">
      <span v-for="(_, i) in images" :key="i"
        :class="{ active: i === index }" @click="index = i" />
    </div>
  </div>
</template>

<script setup lang="ts">
import { ref, onMounted, onBeforeUnmount } from 'vue';
const props = defineProps<{ images: string[]; interval?: number }>();
const index = ref(0);
let timer: any = null;

const next = () => index.value = (index.value + 1) % props.images.length;
const prev = () => index.value = (index.value - 1 + props.images.length) %
props.images.length;
const play = () => { timer = setInterval(next, props.interval ?? 3000); };
const pause = () => clearInterval(timer);

onMounted(play);
onBeforeUnmount(pause);
</script>
```

关键点:

- `transform` + `transition` 实现平滑滚动;
- 自动播放 + 鼠标悬停暂停;
- 无限循环（首尾克隆 + 索引修正）;
- 触摸滑动支持（移动端）。

P12：将数字转换为汉语输出

```
function trans(num: number): string {
  if (num === 0) return '零';
  const digits = '零一二三四五六七八九';
  const units = ['', '十', '百', '千'];
  const bigUnits = ['', '万', '亿', '万亿'];

  const sectionToCN = (n: number): string => {
```

```
    let result = '';
    let zeroFlag = false;
    let i = 0;
    while (n > 0) {
      const d = n % 10;
      if (d === 0) {
        if (!zeroFlag && result) zeroFlag = true;
      } else {
        if (zeroFlag) { result = '零' + result; zeroFlag = false; }
        result = digits[d] + units[i] + result;
      }
      n = Math.floor(n / 10);
      i++;
    }
    return result;
  };

  let result = '';
  let bigIdx = 0;
  let needZero = false;
  while (num > 0) {
    const section = num % 10000;
    if (section === 0) {
      if (result && !result.startsWith('零')) needZero = true;
    } else {
      const cn = sectionToCN(section);
      if (needZero) result = '零' + result;
      result = cn + bigUnits[bigIdx] + result;
      needZero = false;
    }
    num = Math.floor(num / 10000);
    bigIdx++;
  }
  return result;
}
```

P13：编写 Vue 组件，使用插槽接收外部内容

```
<!-- Card.vue -->
<template>
  <div class="card">
    <header v-if="$slots.header">
      <slot name="header" :title="title" />
    </header>
    <main>
      <slot>默认内容</slot>
    </main>
    <footer v-if="$slots.footer">
      <slot name="footer" />
    </footer>
```

```
  </div>
</template>

<script setup lang="ts">
defineProps<{ title?: string }>();
</script>

<!-- 使用 -->
<Card title="卡片">
  <template #header="{ title }">
    <h2>{{ title }}</h2>
  </template>
  <p>主体内容</p>
  <template #footer>
    <button>确定</button>
  </template>
</Card>
```

P14：去除字符串中出现次数最少的字符，不改变原顺序

```
function removeMinChars(str: string): string {
  const counts = new Map<string, number>();
  for (const ch of str) counts.set(ch, (counts.get(ch) || 0) + 1);
  const minCount = Math.min(...counts.values());
  return [...str].filter(ch => counts.get(ch)! > minCount).join('');
}
```

P18：树转数组（扁平化）

```
function treeToArray(tree: any[]): any[] {
  const result: any[] = [];
  const dfs = (nodes: any[]) => {
    nodes.forEach(node => {
      result.push({ id: node.id, name: node.name, pid: node.pid });
      if (node.children?.length) dfs(node.children);
    });
  };
  dfs(tree);
  return result;
}
```

P19：数组转树

```ts
function arrayToTree(arr: Array<{ id: number; pid: number; name: string }>): any[]
{
  const map = new Map<number, any>();
  const roots: any[] = [];
  arr.forEach(item => map.set(item.id, { ...item, children: [] }));
  arr.forEach(item => {
    const node = map.get(item.id)!;
    if (item.pid === 0) {
      roots.push(node);
    } else {
      map.get(item.pid)?.children.push(node);
    }
  });
  return roots;
}
```

P20：删除链表的一个节点

已知要删除节点的指针（无头节点引用）：

```ts
function deleteNode(node: ListNode): void {
  // 把下一个节点的值复制到当前节点，然后删除下一个节点
  node.val = node.next!.val;
  node.next = node.next!.next;
}
```

已知链表头 + 值：

```ts
function deleteNodeByVal(head: ListNode | null, val: number): ListNode | null {
  if (!head) return null;
  if (head.val === val) return head.next;
  let cur = head;
  while (cur.next && cur.next.val !== val) cur = cur.next;
  if (cur.next) cur.next = cur.next.next;
  return head;
}
```

P21：实现请求并发控制（最多 N 个并发）

```ts
async function pLimit<T>(max: number, tasks: (() => Promise<T>)[]): Promise<T[]> {
  const results: T[] = [];
  let idx = 0;
  const workers = Array.from({ length: max }, async () => {
    while (idx < tasks.length) {
```

```
      const cur = idx++;
      results[cur] = await tasks[cur]();
    }
  });
  await Promise.all(workers);
  return results;
}
```

进阶：失败重试 + 优先级 → 推荐使用 `p-limit` + `p-retry` npm 包。

P22：实现 fetchWithRetry（带重试机制）

```
async function fetchWithRetry(
  url: string,
  options: RequestInit = {},
  retries = 3,
  delay = 1000
): Promise<Response> {
  try {
    const res = await fetch(url, options);
    if (!res.ok) throw new Error(`HTTP ${res.status}`);
    return res;
  } catch (err) {
    if (retries <= 0) throw err;
    await new Promise(r => setTimeout(r, delay));
    // 指数退避
    return fetchWithRetry(url, options, retries - 1, delay * 2);
  }
}
```

P23：链表环的入口节点

快慢指针法：

```
function detectCycle(head: ListNode | null): ListNode | null {
  if (!head || !head.next) return null;
  let slow = head, fast = head;
  while (fast && fast.next) {
    slow = slow.next!;
    fast = fast.next.next!;
    if (slow === fast) break;
  }
  if (slow !== fast) return null;
  // 相遇后，让 slow 从头出发，再次相遇即入口
  slow = head;
  while (slow !== fast) {
    slow = slow.next!;
```

```
    fast = fast.next!;
  }
  return slow;
}
```

原理：快指针走 2k 步，慢指针走 k 步，相遇时让慢指针回到头，再次相遇即为环入口。

P24：多叉树指定层节点的个数

```
function countAtLevel(root: any, targetLevel: number, curLevel = 1): number {
  if (!root) return 0;
  if (curLevel === targetLevel) return 1;
  let count = 0;
  for (const child of (root.children || [])) {
    count += countAtLevel(child, targetLevel, curLevel + 1);
  }
  return count;
}

// BFS 版
function countAtLevelBFS(root: any, targetLevel: number): number {
  if (!root || targetLevel < 1) return 0;
  let level = 1, queue = [root];
  while (queue.length) {
    if (level === targetLevel) return queue.length;
    const next: any[] = [];
    for (const node of queue) next.push(...(node.children || []));
    queue = next;
    level++;
  }
  return 0;
}
```

P25：手写快速排序

```
function quickSort(arr: number[]): number[] {
  if (arr.length <= 1) return arr;
  const pivot = arr[0];
  const left = arr.slice(1).filter(x => x <= pivot);
  const right = arr.slice(1).filter(x => x > pivot);
  return [...quickSort(left), pivot, ...quickSort(right)];
}

// 原地版（性能更好）
function quickSortInPlace(arr: number[], lo = 0, hi = arr.length - 1): number[] {
  if (lo < hi) {
    const pivotIdx = partition(arr, lo, hi);
```

```
    quickSortInPlace(arr, lo, pivotIdx - 1);
    quickSortInPlace(arr, pivotIdx + 1, hi);
  }
  return arr;
}
function partition(arr: number[], lo: number, hi: number): number {
  const pivot = arr[hi];
  let i = lo - 1;
  for (let j = lo; j < hi; j++) {
    if (arr[j] <= pivot) {
      i++;
      [arr[i], arr[j]] = [arr[j], arr[i]];
    }
  }
  [arr[i + 1], arr[hi]] = [arr[hi], arr[i + 1]];
  return i + 1;
}
```

复杂度：平均 O(n log n)，最坏 $O(n^2)$。

P26：有序数组原地去重

```
function uniqueSorted(arr: number[]): number[] {
  if (arr.length <= 1) return arr;
  let slow = 0;
  for (let fast = 1; fast < arr.length; fast++) {
    if (arr[fast] !== arr[slow]) {
      slow++;
      arr[slow] = arr[fast];
    }
  }
  return arr.slice(0, slow + 1);
}
```

复杂度：O(n) 时间，O(1) 空间。

P27：计算数组中的平均时间

题目：给定形如 `[{ start: '10:00', end: '12:00' }, ...]` 的数组，计算所有时间段的平均时长（小时）。

```
function averageTime(intervals: Array<{ start: string; end: string }>): number {
  const toMinutes = (t: string) => {
    const [h, m] = t.split(':').map(Number);
    return h * 60 + m;
  };
  const total = intervals.reduce((sum, { start, end }) => {
```

```
    return sum + (toMinutes(end) - toMinutes(start));
  }, 0);
  return total / intervals.length / 60; // 转小时
}
```

三、TypeScript（27 题）

T1：TypeScript 中命名空间与模块的理解？区别？

模块（Module）：

- ES6 标准，基于文件的模块系统（`import` / `export`）；
- 可以是 `export` 任何顶级声明（变量、函数、类、接口）；
- 由模块加载器（ESM、CommonJS、AMD）负责依赖管理。

命名空间（Namespace）：

- TypeScript 早期为解决全局变量污染引入的**内部模块**机制；
- 使用 `namespace X { ... }` 声明，内部 `export` 导出成员；
- 最终被编译为 IIFE，依赖全局对象；
- **不推荐在新项目使用**，仅适用于声明文件（`.d.ts`）和全局类型扩展。

```
// 命名空间（不推荐）
namespace Utils {
  export function format(date: Date) { /* ... */ }
}

// 模块（推荐）
// utils.ts
export function format(date: Date) { /* ... */ }
```

区别对比：

维度	模块	命名空间
范围	文件级	块级
加载	模块系统	全局对象
推荐场景	所有现代项目	仅 .d.ts 中使用

T2：TypeScript 是什么？与 JavaScript 的区别？

TypeScript 是 JavaScript 的**超集**，添加了**静态类型系统**和现代语言特性，最终编译为纯 JavaScript。

核心区别：

维度	JavaScript	TypeScript

维度	JavaScript	TypeScript
类型系统	动态类型，运行时检查	静态类型，编译时检查
错误检测	运行时崩溃	编译时报警
开发体验	需手动测试	智能提示 + 重构
适用场景	小型脚本、快速迭代	中大型项目、团队协作
学习曲线	低	中（需理解类型）
运行时性能	无额外开销	编译后无运行时类型检查

关键优势：

1. 静态类型提前发现问题；
2. 强大的 IDE 支持（VS Code 智能补全、跳转、重构）；
3. 现代特性提前使用（装饰器、可选链、空值合并等，编译到目标 ES 版本）；
4. 更适合大型项目维护。

T3：TypeScript 中泛型是什么？

泛型（Generics）：在定义函数、接口、类时不预先指定类型，使用时再指定类型的特性。

```
// 泛型函数
function identity<T>(value: T): T {
  return value;
}
identity<string>('hello'); // T = string
identity(42);              // T 自动推断为 number

// 泛型接口
interface ApiResponse<T> {
  code: number;
  data: T;
  message: string;
}
const res: ApiResponse<User> = { code: 0, data: { name: 'Tom' }, message: 'ok' };

// 泛型类
class Stack<T> {
  private items: T[] = [];
  push(item: T) { this.items.push(item); }
  pop(): T | undefined { return this.items.pop(); }
}

// 泛型约束
interface Lengthwise { length: number; }
function logLength<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
```

```
  return arg;
}
```

作用：编写**可复用、类型安全**的代码，避免使用 any 丢失类型。

T4：TypeScript 中有哪些声明变量的方式?

```
// 1. var（不推荐，函数作用域）
var a = 1;

// 2. let（块作用域，可重新赋值）
let b = 2;

// 3. const（块作用域，不可重新赋值）
const c = 3;

// 4. 解构声明
const { x, y } = { x: 1, y: 2 };
const [first, ...rest] = [1, 2, 3];

// 5. 类型断言声明
const el = document.querySelector('#app') as HTMLDivElement;
const len = (<string>someValue).length;

// 6. 函数声明
function foo() {}
const bar = function() {};

// 7. 接口/类型别名
interface User { name: string; }
type Pair = [number, number];

// 8. 类声明
class Animal {}

// 9. 枚举声明
enum Color { Red, Green, Blue }

// 10. namespace / module 声明
namespace Utils { export const PI = 3.14; }
```

T5：TypeScript 的方法重载是什么?

概念：同一个函数名定义多个类型签名，编译器按调用实参匹配最合适的一个。

```
// 签名列表（无函数体）
function add(a: number, b: number): number;
```

```ts
function add(a: string, a: string): string;
// 实现签名（必须兼容所有重载）
function add(a: any, b: any): any {
  return a + b;
}

add(1, 2);          // 3
add('a', 'b');      // 'ab'
// add(true, false); // 报错
```

注意：

- 重载是**编译时**的概念，运行时仍是单个函数；
- 重载列表必须由上到下从**最具体到最宽泛**排列；
- 实现签名对外不可见。

T6：实现 sleep 方法

```ts
// Promise 版（推荐）
function sleep(ms: number): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, ms));
}

// 使用
async function run() {
  console.log('start');
  await sleep(1000);
  console.log('after 1s');
}
```

T7：TypeScript 中 is 关键字有什么用?

类型谓词（Type Predicate）：用于在自定义类型守卫函数中**收窄类型**。

```ts
interface Fish { swim(): void; }
interface Bird { fly(): void; }

function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}

function move(pet: Fish | Bird) {
  if (isFish(pet)) {
    pet.swim();  // 此处 pet 被收窄为 Fish
  } else {
    pet.fly();   // 此处 pet 被收窄为 Bird
```

```
  }
}
```

作用：告诉编译器某个条件返回 true 时参数的具体类型，提升类型推断准确性。

T8：TypeScript 支持的访问修饰符有哪些?

```
class User {
  public name: string;        // 默认，任何位置可访问
  private age: number;        // 仅本类内部
  protected role: string;     // 本类及子类
  readonly id: number;        // 只读，构造后不可改

  constructor(name: string, age: number, role: string, id: number) {
    this.name = name;
    this.age = age;
    this.role = role;
    this.id = id;
  }
}

// 参数属性简化写法
class User2 {
  constructor(
    public name: string,
    private age: number,
    protected role: string,
    readonly id: number
  ) {}
}

// private 在编译期生效，运行时仍是公共字段
// 如需运行时保护，使用 # 私有字段（ES2022）
class A {
  #secret = 123; // 真正的私有
}
```

T9：实现 myMap 方法

```
// 类似 Array.prototype.map
function myMap<T, U>(arr: T[], callback: (item: T, index: number, array: T[]) =>
U): U[] {
  const result: U[] = [];
  for (let i = 0; i < arr.length; i++) {
    result.push(callback(arr[i], i, arr));
  }
```

```
  return result;
}
```

T10：实现 treePath 方法

题目：给定树形结构（带 `id` / `children`），返回从根到目标节点的路径。

```
interface TreeNode { id: number; children?: TreeNode[]; }

function treePath(root: TreeNode, target: number, path: number[] = []): number[] |
null {
  path.push(root.id);
  if (root.id === target) return path;
  for (const child of root.children || []) {
    const found = treePath(child, target, path);
    if (found) return found;
  }
  path.pop();
  return null;
}
```

T11：实现 product 方法（笛卡尔积 / 多数组组合）

```
function product<T>(...arrays: T[][]): T[][] {
  if (arrays.length === 0) return [[]];
  return arrays.reduce((acc, arr) =>
    acc.flatMap(a => arr.map(b => [...a, b])), [[]] as T[][]
  );
}

product([1, 2], ['a', 'b'], [true, false]);
// [
//   [1, 'a', true], [1, 'a', false],
//   [1, 'b', true], [1, 'b', false],
//   [2, 'a', true], [2, 'a', false],
//   [2, 'b', true], [2, 'b', false]
// ]
```

T12：实现 myAll 方法（Promise.all 类型安全版）

```
function myAll<T>(promises: Promise<T>[]): Promise<T[]> {
  return new Promise((resolve, reject) => {
    const result: T[] = [];
    let count = 0;
```

```
    if (promises.length === 0) return resolve([]);
    promises.forEach((p, i) => {
      Promise.resolve(p).then(
        val => { result[i] = val; if (++count === promises.length)
resolve(result); },
        reject
      );
    });
  });
}
```

T13：实现 sum 方法（柯里化求和）

```
function sum(...args: number[]): number {
  return args.reduce((a, b) => a + b, 0);
}

// 柯里化版 sum(1)(2)(3)(4)() = 10
function curriedSum(...args: number[]): any {
  const inner = (...rest: number[]) => curriedSum(...args, ...rest);
  inner.valueOf = () => args.reduce((a, b) => a + b, 0);
  return inner;
}
```

T14：实现 mergeArray 方法（数组合并去重）

```
function mergeArray<T>(...arrays: T[][]): T[] {
  return [...new Set(arrays.flat())];
}

// 复杂对象去重（指定 key）
function mergeByKey<T>(arrays: T[][], key: keyof T): T[] {
  const map = new Map<T[keyof T], T>();
  arrays.flat().forEach(item => map.set(item[key], item));
  return [...map.values()];
}
```

T15：实现 firstSingleChar 方法

题目：找到字符串中第一个只出现一次的字符。

```
function firstSingleChar(str: string): string | null {
  const counts = new Map<string, number>();
  for (const ch of str) counts.set(ch, (counts.get(ch) || 0) + 1);
```

```
  for (const ch of str) {
    if (counts.get(ch) === 1) return ch;
  }
  return null;
}
```

T16：实现 reverseWord 方法（按单词反转字符串）

```
function reverseWord(str: string): string {
  return str.trim().split(/\s+/).reverse().join(' ');
}
```

T17：定义混合类型数组（元素可能是 string 或 number）

```
// 联合类型
const arr1: (string | number)[] = [1, 'a', 2, 'b'];

// 元组（位置固定）
const arr2: [string, number, string] = ['a', 1, 'b'];

// 更复杂的场景：每项是不同结构
type Item = { type: 'text'; value: string } | { type: 'num'; value: number };
const arr3: Item[] = [
  { type: 'text', value: 'hello' },
  { type: 'num', value: 42 },
];

// 使用时类型守卫收窄
arr3.forEach(item => {
  if (item.type === 'text') console.log(item.value.toUpperCase());
  else console.log(item.value.toFixed(2));
});
```

T18：补充 objToArray 函数

题目：`{ a: 1, b: 2 }` → `[['a', 1], ['b', 2]]`

```
function objToArray<T>(obj: Record<string, T>): [string, T][] {
  return Object.entries(obj);
}

// 或自定义转换
function objToArray2<T>(obj: Record<string, T>, keyName = 'key', valueName =
'value') {
```

```
  return Object.entries(obj).map(([k, v]) => ({ [keyName]: k, [valueName]: v }));
}
```

T19：使用 TS 判断参数是否是数组

```
// 方法1：Array.isArray（推荐）
function isArray<T>(arg: unknown): arg is T[] {
  return Array.isArray(arg);
}

// 方法2：instanceof
function isArray2<T>(arg: unknown): arg is T[] {
  return arg instanceof Array;
}
```

T20：TypeScript 的内置数据类型有哪些?

```
// 原始类型
let s: string = 'hello';
let n: number = 42;
let b: boolean = true;
let nu: null = null;
let un: undefined = undefined;
let sy: symbol = Symbol('id');
let bi: bigint = 10n;

// 对象类型
let obj: object = {};
let fn: Function = () => {};

// 特殊类型
let anyT: any = 'anything';       // 绕过类型检查
let unk: unknown = 'safe';          // 安全的 any，使用前需收窄
let nev: never = (() => { throw new Error(); })();  // 不可能存在的值
let voi: void = undefined;          // 无返回值

// 复合类型
let uni: string | number = 'a';
let inter: { name: string } & { age: number } = { name: 'Tom', age: 18 };

// 字面量类型
let lit: 'small' | 'medium' | 'large' = 'medium';

// 集合类型
let arr: number[] = [1, 2];
let tup: [string, number] = ['a', 1];
```

```
// 包装类型
let strObj: String = new String('a');  // 不推荐
```

T21：any 和 unknown 有什么区别?

维度	any	unknown
类型检查	绕过所有检查	使用前必须收窄
可赋值给	任何类型	仅 any / unknown
可调用	直接调用	必须先收窄才能调用
安全性	不安全	类型安全

最佳实践：**默认用 unknown**，必要时用类型守卫收窄。

```
let a: any = 'hello';
a.toFixed(); // 编译通过，运行可能报错

let u: unknown = 'hello';
// u.toFixed(); // 编译报错
if (typeof u === 'string') {
  u.toUpperCase(); // 收窄为 string 后安全使用
}
```

T22：如何将 unknown 类型指定为更具体的类型?

三种方式:

1. **类型断言**（强制，不做检查）：

```
const v: unknown = 'hello';
const s = v as string;
```

2. **类型守卫**（推荐）：

```
if (typeof v === 'string') { /* v 为 string */ }
if (v instanceof Date) { /* v 为 Date */ }
if (Array.isArray(v)) { /* v 为数组 */ }
```

3. **自定义类型谓词**:

```ts
interface User { name: string; }
function isUser(x: unknown): x is User {
  return typeof x === 'object' && x !== null && 'name' in x;
}
```

T23：使用 TS 实现入参是否是数组的判断

```ts
// 类型谓词版
function isArray<T>(arg: unknown): arg is T[] {
  return Array.isArray(arg);
}

// 更严格的判断（只接受非空数组）
function isNonEmptyArray<T>(arg: unknown): arg is [T, ...T[]] {
  return Array.isArray(arg) && arg.length > 0;
}
```

T24：tsconfig.json 文件有什么用?

作用：TypeScript 编译器的配置文件，控制编译行为。

常用配置：

```jsonc
{
  "compilerOptions": {
    "target": "ES2020",              // 编译目标 JS 版本
    "module": "ESNext",              // 模块系统
    "moduleResolution": "bundler",   // 模块解析策略
    "lib": ["DOM", "ES2020"],        // 类型库
    "jsx": "react-jsx",              // JSX 处理方式

    "strict": true,                  // 严格模式总开关
    "noImplicitAny": true,           // 禁止隐式 any
    "strictNullChecks": true,        // 严格 null 检查
    "strictFunctionTypes": true,

    "esModuleInterop": true,         // 允许 default import 语法
    "allowSyntheticDefaultImports": true,
    "skipLibCheck": true,            // 跳过 .d.ts 类型检查
    "forceConsistentCasingInFileNames": true,

    "baseUrl": "./",
    "paths": {                       // 路径别名
      "@/*": ["src/*"]
    },
```

```
    "outDir": "./dist",
    "rootDir": "./src",
    "sourceMap": true,
    "declaration": true            // 生成 .d.ts
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

T25：TypeScript 中 Declare 关键字有什么用?

作用：告诉编译器"该变量/模块已存在"，避免类型错误。

```
// 1. 声明全局变量
declare const VERSION: string;
declare const __DEV__: boolean;

// 2. 声明全局函数
declare function greet(name: string): void;

// 3. 声明全局类
declare class MyLib { constructor(opts?: any); }

// 4. 声明模块（最常见：补充没有类型的 npm 包）
declare module 'my-lib' {
  export function init(config: object): void;
  export const version: string;
}

// 5. 声明命名空间扩展
declare global {
  interface Window {
    myApp: { foo(): void };
  }
}

// 6. 声明模块扩展（UMD）
declare module '*.css' {
  const content: string;
  export default content;
}
```

T26：解释 TypeScript 中的枚举

枚举（Enum）：为一组数值赋予有意义的名字。

```ts
// 数字枚举（自动递增）
enum Direction {
  Up,    // 0
  Down,  // 1
  Left,  // 2
  Right  // 3
}

// 字符串枚举（推荐）
enum Status {
  Pending = 'PENDING',
  Success = 'SUCCESS',
  Error = 'ERROR'
}

// 常量枚举（编译时内联）
const enum Color {
  Red = '#FF0000',
  Green = '#00FF00'
}

// 反向映射（仅数字枚举）
enum Role { Admin, User }
const name: string = Role[0]; // 'Admin'

// 使用
function setStatus(s: Status) { /* ... */ }
setStatus(Status.Success);
```

注意：

- 普通枚举会产生额外 JS 代码；
- 字符串枚举不会反向映射；
- 推荐使用 **字面量联合类型** 代替枚举（更轻量、Tree-shakable）：

```ts
type Status = 'PENDING' | 'SUCCESS' | 'ERROR';
```

T27：TypeScript 的主要特点是什么？

1. **静态类型系统**：编译期类型检查，提前发现错误。
2. **类型推断**：自动推导变量类型，减少冗余标注。
3. **现代 JS 特性**：支持 ES6+、装饰器、可选链、空值合并等，编译到旧版本。
4. **强大的泛型**：编写可复用、类型安全的代码。
5. **结构化类型（鸭子类型）**：只要形状兼容即兼容，灵活。
6. **渐进式采用**：可以与 JS 共存，逐步迁移。
7. **丰富工具链**：VS Code 一流支持，智能提示、重构、跳转。
8. **高级类型**：联合类型、交叉类型、条件类型、映射类型、模板字面量类型。

9. **装饰器**：支持类与方法元编程（Stage 3）。
10. **生态丰富**：几乎所有主流库都有 .d.ts 类型定义。

四、Vue（24 题）

V1：Vue 有了数据响应式，为何还要 diff?

原因：

1. **响应式只能解决“是否要更新”问题**：当响应式数据变化时，Vue 知道“该重新渲染了”，但**不知道具体要改哪里。**
2. **Vue 的粒度不是单一变量**：一个组件模板中可能有多个数据依赖，状态变化后需要重新执行 render 函数生成新的 VNode。
3. **diff 负责高效比较新旧 VNode 树**：找出最小 DOM 操作，应用到真实 DOM。
4. **性能优化关键**：diff 使用**双端对比 + 最长递增子序列**算法，复杂度从 $O(n^3)$ 降到 O(n)。

总结：响应式负责**调度**，diff 负责**精准更新。**

V2：Vue 3 为什么不需要时间分片?

原因：

1. **响应式粒度更细**：Vue 通过 Proxy 精确知道哪个响应式数据变化，且只触发**使用了该数据的组件**重新渲染。
2. **静态提升 + 补丁标记**：编译期优化避免了大量 VNode 比较。
3. **组件级调度**：默认一个组件一个更新任务，粒度适合，不容易产生超长任务。
4. **框架定位不同**：Vue 设计为“渐进式”，默认不启用并发模式，代码量与心智负担更低。

React 为什么需要：不可变数据 + setState 模型下，一次更新可能涉及大量组件（任何依赖该状态的组件都重渲染），所以需要时间切片中断与优先级调度。

V3：Vue 3 为什么要引入 Composition API?

三大动机：

1. **逻辑复用**：Options API 中逻辑被拆到不同选项（data / methods / computed / mounted），复用难（mixin 冲突、来源不清）。Composition API 把逻辑聚合为函数，可随意复用。
2. **更好的 TS 支持**：Composition API 主要用函数与变量，与 TS 推断结合更好。
3. **逻辑组织灵活性**：Options API 按选项分类（横向），Composition API 按逻辑关注点分类（纵向），复杂组件更清晰。

示例对比：

```
// Options API
export default {
  data() { return { count: 0 }; },
  computed: { double() { return this.count * 2; } },
  mounted() { console.log(this.count); }
```

```
};

// Composition API
import { ref, computed, onMounted } from 'vue';
export default {
  setup() {
    const count = ref(0);
    const double = computed(() => count.value * 2);
    onMounted(() => console.log(count.value));
    return { count, double };
  }
};
```

V4：谈 Vue 事件机制，手写 $on、$off、$emit、$once

Vue 2 原型事件机制：在 `Vue.prototype` 上维护一个事件中心。

```
// 简化实现
class EventBus {
  private events: Record<string, Function[]> = {};

  $on(event: string, fn: Function) {
    (this.events[event] ||= []).push(fn);
  }

  $off(event: string, fn?: Function) {
    if (!this.events[event]) return;
    if (!fn) { delete this.events[event]; return; }
    this.events[event] = this.events[event].filter(f => f !== fn);
  }

  $emit(event: string, ...args: any[]) {
    (this.events[event] || []).forEach(fn => fn(...args));
  }

  $once(event: string, fn: Function) {
    const wrapper = (...args: any[]) => {
      this.$off(event, wrapper);
      fn(...args);
    };
    this.$on(event, wrapper);
  }
}
```

Vue 3 移除原因：官方推荐使用外部库（mitt / tiny-emitter）或 props/emit，理由是 Vue 实例本身已足够庞大。

V5：computed 计算值为什么可以依赖另一个 computed?

```
const a = ref(1);
const b = computed(() => a.value * 2);
const c = computed(() => b.value + 1); // 依赖 b，b 依赖 a

// a.value = 2; → b 更新 → c 自动更新
```

原理：

- computed 是基于**响应式依赖**实现的特殊 effect；
- 当 computed 被访问时会进行依赖收集，依赖列表中包含**所有被访问的响应式源**（包括其他 computed）；
- 被依赖的 computed 内部使用 `dirty` 标记控制是否重新计算。

优势：自动依赖追踪，调用者可当作普通 ref 使用，无需手动订阅。

V6：说一下 vm.$set 原理

Vue 2 问题：Vue 2 通过 `Object.defineProperty` 劫持已有属性，**无法检测新增属性**和**直接通过索引修改数组元素**。

$set 实现原理：

```
function set(target: any, key: string | number, val: any) {
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    target.length = Math.max(target.length, key as number);
    target.splice(key as number, 1, val);
    return val;
  }
  if (key in target && !(key in Object.prototype)) {
    target[key] = val;
    return val;
  }
  const ob = target.__ob__;
  if (!ob) { target[key] = val; return val; }
  ob.defineReactive(target, key, val); // 关键：手动变为响应式
  ob.dep.notify(); // 手动触发更新
  return val;
}
```

Vue 3 已解决：基于 Proxy，无需 $set，直接赋值即可响应式。

V7：怎么在 Vue 中定义全局方法?

Vue 2：

```
// 1. 挂到 Vue 原型
Vue.prototype.$http = axios;

// 2. 插件形式（推荐）
export default {
  install(Vue) {
    Vue.prototype.$utils = utils;
  }
};
Vue.use(MyPlugin);
```

Vue 3：

```
// 1. app.config.globalProperties
app.config.globalProperties.$http = axios;

// 2. provide / inject（推荐）
app.provide('http', axios);

// 在组件中
const http = inject('http');
```

V8：Vue 中父组件怎么监听到子组件的生命周期？

Vue 2：通过 `@hook:生命周期` 事件。

```
<Child @hook:mounted="onChildMounted" />
```

Vue 3：需要在子组件显式 emit：

```
<!-- Child.vue -->
<script setup>
const emit = defineEmits(['mounted']);
onMounted(() => emit('mounted'));
</script>

<!-- Parent.vue -->
<Child @mounted="onChildMounted" />
```

更优方案：父组件可通过 `ref` 获取子组件实例，调用其暴露的方法（`defineExpose`）。

V9：vue 组件里写的原生 addEventListener 监听事件，要手动销毁吗？

必须手动销毁！

```
onMounted(() => {
  window.addEventListener('resize', onResize);
});
onBeforeUnmount(() => {
  window.removeEventListener('resize', onResize); // 必须，否则内存泄露
});
```

原因：Vue 只会在组件卸载时清理**它自己绑定的事件**（通过模板 `@event`），不会清理通过原生 API 绑定的事件。

Vue 3 setup 自动 cleanup：使用 `useEventListener` 库或自实现 hook。

V10：Vue 3 中的响应式设计原理

核心：Proxy + Reflect

```
function reactive<T extends object>(target: T): T {
  return new Proxy(target, {
    get(target, key, receiver) {
      track(target, key); // 依赖收集
      return Reflect.get(target, key, receiver);
    },
    set(target, key, value, receiver) {
      const oldValue = target[key];
      const result = Reflect.set(target, key, value, receiver);
      if (oldValue !== value) {
        trigger(target, key); // 派发更新
      }
      return result;
    },
    deleteProperty(target, key) {
      const result = Reflect.deleteProperty(target, key);
      trigger(target, key);
      return result;
    }
  });
}
```

依赖收集：

- 每个响应式对象关联一个 `dep`（依赖容器）；
- 访问属性时记录当前激活的 effect；
- 收集为 `dep.subs = Set<effect>`。

派发更新：

- 属性变化时找到 `dep`，遍历 `subs` 依次执行 effect。

对比 Vue 2：

- Vue 2：Object.defineProperty 劫持已有属性，初始化递归遍历；
- Vue 3：Proxy 懒加载（访问才代理）、数组下标 / length / Map / Set 全部支持。

V11：Vue 中，created 和 mounted 钩子之间的时间受什么影响？

时间差 = created 钩子执行 → render 函数生成 VNode → 首次 patch → DOM 插入 → mounted 钩子执行

影响因素：

1. **模板复杂度**：模板越复杂、节点越多，渲染耗时越长；
2. **组件嵌套深度**：嵌套越多，patch 次数越多；
3. **同步子组件状态**：子组件 init / mounted 也会占用时间；
4. **首次渲染数据计算**：复杂的 computed、watch 初始化；
5. **业务逻辑**：created 中同步耗时操作（同步请求、复杂计算）会卡在这一阶段。

优化建议：

- created 只做轻量初始化（state 准备、事件绑定）；
- 耗时逻辑放 `onMounted` 或异步加载。

V12：Vue 中，推荐在哪个生命周期发起请求？

推荐：`created` 或 `setup` 中

原因：

1. **提前请求**：在 DOM 渲染前发起，能提前获取数据，减少首屏加载时间；
2. **SSR 友好**：服务端渲染时 created 会执行，而 mounted 不会（服务器无 DOM）；
3. **依赖收集**：created 阶段访问数据也能被响应式系统追踪。

需要等待 DOM 的场景才用 mounted（如获取元素尺寸、绑定 DOM 事件）。

```
// 推荐写法
export default {
  async created() {
    const res = await fetchData();
    this.list = res.data;
  }
};
```

V13：为什么 React 需要 Fiber 架构，而 Vue 却不需要？

React 面临的问题：

- 数据不可变，状态变化触发**整个组件子树重渲染**；
- 需要调度器决定优先级、可以中断与恢复。

Vue 天然优势：

- **响应式粒度细**：Proxy 精确追踪到哪个组件、哪个属性变化，只重新渲染**实际依赖该数据的组件**；
- **编译期优化**：静态节点提升、补丁标记，减少 VNode 比较；
- **架构简单**：不需要时间切片、不需要并发模式也能保持良好性能。

总结：React 是"**重调度 + 轻响应式**"，Vue 是"**重响应式 + 轻调度**"。

V14：SPA 首屏加载速度慢怎么解决？

八种优化手段：

1. **包体积优化**：路由懒加载、按需引入第三方库、Tree Shaking、压缩代码（terser / esbuild）。
2. **资源优化**：CDN、图片压缩（WebP/AVIF）、雪碧图、内联关键 CSS。
3. **缓存策略**：HTTP 强缓存 + 协商缓存、Service Worker、IndexedDB。
4. **网络优化**：HTTP/2 多路复用、域名分片、预解析 DNS、预连接（preconnect）、预加载（preload）。
5. **SSR / SSG**：Nuxt / Next.js、服务端预渲染、静态化。
6. **骨架屏 / Loading**：优化感知体验。
7. **首屏数据预取**：路由进入前预取数据。
8. **监控 & 调优**：Lighthouse、WebPageTest、PerformanceObserver。

V15：说下 Vite 的原理

核心：浏览器原生 ESM + esbuild 预构建

冷启动：

1. **预构建依赖**（基于 esbuild）：用 `esbuild` 将 `node_modules` 中的 CommonJS / UMD 模块转换为 ESM（快 10-100 倍）。
2. **按需编译**：浏览器请求哪个模块，Vite 才用 `esbuild` 转换该模块（转换与请求并行），返回后浏览器继续请求依赖模块。
3. **缓存**：强缓存（HTTP 头）+ 协商缓存（304）。

热更新 HMR：

1. 通过 WebSocket 监听文件变化；
2. 变动的模块被 `esbuild` 重新编译；
3. 框架插件接收更新（vue-loader / react-refresh），热替换运行中的模块。

生产构建：使用 Rollup（生态成熟、产物优化好）。

为什么快：

- **不走打包**：无需全量构建；
- **按需编译**：用哪个编哪个；
- **原生 ESM**：浏览器原生支持，无需 polyfill。

V16：Vue 2 为什么不能检测数组变化？怎么解决？

原因：Vue 2 通过 `Object.defineProperty` 劫持，**数组的以下操作不会被检测**：

- 直接通过索引赋值：`arr[0] = 1`；
- 修改 length：`arr.length = 0`；
- 部分方法（Vue 2 已重写了 7 个变更方法：`push/pop/shift/unshift/splice/sort/reverse`）。

解决方案：

```
// 1. Vue.set / this.$set
this.$set(this.arr, index, value);

// 2. 替换数组（推荐）
this.arr.splice(index, 1, value);

// 3. 整体重新赋值
this.arr = [...this.arr.slice(0, index), value, ...this.arr.slice(index + 1)];
```

Vue 3 已根治：基于 Proxy，所有数组操作天然可追踪。

V17：说说 Vue 页面渲染流程

Vue 3 渲染流程：

1. **编译阶段**：模板 → AST → 渲染函数（带静态提升、补丁标记优化）。
2. **挂载阶段**（首次渲染）：
   - 创建组件实例（setup）；
   - 执行 render 函数生成 VNode 树；
   - 调用 `patch` 函数；
   - **首次 patch**走挂载逻辑：创建真实 DOM、插入到容器、绑定事件；
   - 触发 `onMounted` 钩子。
3. **更新阶段**：
   - 响应式数据变化触发 effect；
   - 调度器（scheduler）将更新加入微任务队列（nextTick）；
   - 重新执行 render 函数生成新 VNode；
   - 调用 `patch` 函数进行 **diff 比较**；
   - 将差异（DOM 操作）应用到真实 DOM；
   - 触发 `onUpdated` 钩子。
4. **卸载阶段**：触发 `onBeforeUnmount` / `onUnmounted`，移除 DOM、解绑事件、清理 effect。

V18：Vue 中 computed 和 watch 区别

维度	computed	watch
用途	派生新值	监听数据变化执行副作用

维度	computed	watch
缓存	有缓存，依赖不变不重算	无缓存
返回	必须 return	不需要
异步	不支持异步	支持异步
调用时机	访问时惰性求值	数据变化立即/深度监听
适用	模板渲染、数据转换	路由变化、数据持久化、接口请求

```
// computed
const fullName = computed(() => `${firstName.value} ${lastName.value}`);

// watch
watch(searchKey, async (newVal) => {
  const res = await api.search(newVal);
  list.value = res.data;
}, { debounce: 300, immediate: true });
```

V19：Vuex 中的辅助函数怎么使用?

辅助函数：mapState、mapGetters、mapMutations、mapActions、createNamespacedHelpers。

```
// Options API 中
import { mapState, mapMutations } from 'vuex';

export default {
  computed: {
    ...mapState(['count']),
    ...mapState('user', ['name'])
  },
  methods: {
    ...mapMutations(['increment']),
    ...mapActions(['fetchData'])
  }
};

// Composition API 中（推荐）
import { useStore } from 'vuex';
import { computed } from 'vue';

export default {
  setup() {
    const store = useStore();
    return {
      count: computed(() => store.state.count),
      increment: () => store.commit('increment')
    };
```

```
  }
};
```

Vue 3 推荐 Pinia：更轻量、Composition API 友好、TypeScript 友好。

V20：用 Vue 3 实现一个 Modal，怎么设计？

设计要点：

1. **Teleport 渲染到 body**：避免被父元素裁剪 / 滚动；
2. **受控 / 非受控**：`v-model` 双向绑定显隐；
3. **事件**：`open` / `close` / `confirm` / `cancel`；
4. **插槽**：title、default、footer；
5. **可配置**：尺寸、遮罩、点击遮罩关闭、ESC 关闭、滚动锁定。

```
<template>
  <Teleport to="body">
    <Transition name="modal">
      <div v-if="modelValue" class="modal-mask" @click.self="close">
        <div class="modal" :style="{ width: width }">
          <header v-if="$slots.title">
            <slot name="title" />
            <button @click="close">×</button>
          </header>
          <main><slot /></main>
          <footer v-if="$slots.footer">
            <slot name="footer" :confirm="confirm" :cancel="cancel" />
          </footer>
        </div>
      </div>
    </Transition>
  </Teleport>
</template>

<script setup lang="ts">
const props = defineProps<{ modelValue: boolean; width?: string }>();
const emit = defineEmits(['update:modelValue', 'confirm', 'cancel']);

const close = () => emit('update:modelValue', false);
const confirm = () => emit('confirm');
const cancel = () => emit('cancel');

const onKey = (e: KeyboardEvent) => e.key === 'Escape' && close();
watch(() => props.modelValue, (v) => {
  document.body.style.overflow = v ? 'hidden' : '';
  if (v) document.addEventListener('keydown', onKey);
  else document.removeEventListener('keydown', onKey);
});
</script>
```

V21：Vue 3 Tree Shaking 特性是什么？举例说明

原理：ES Module 静态结构使得打包工具可以分析哪些导出未被使用，删除未引用的代码。

Vue 3 的优势：

- 全部 API 采用 ESM 导出，按需引入；
- 编译器、运行时、服务端渲染分离为独立包；
- `v-model` / 自定义指令等按功能分包。

```
// 只引入需要的 API（Tree-shakable）
import { ref, computed, onMounted } from 'vue';

// 未被引入的 API（如 v-show / Transition）不会出现在最终 bundle 中
```

对比 Vue 2：Vue 2 将所有 API 挂载到 Vue 单例上（`Vue.nextTick`、`Vue.set`），打包工具无法静态分析，默认全部打包进去。

V22：Vue 3 Composition API vs Vue 2 Options API

维度	Options API	Composition API
组织方式	按选项分类（data/methods/...）	按逻辑关注点聚合
代码复用	mixin（覆盖/来源不清晰）	自定义 Hook 函数
TS 推断	一般	优秀
学习曲线	低，上手快	中，需要理解响应式原理
适用场景	小型组件、简单逻辑	复杂组件、大型企业应用
响应式原理透明性	隐藏	需手动管理

推荐策略：复杂组件用 Composition API，简单展示组件用 Options API（Vue 3 同时支持）。

V23：Vue 3 性能提升主要体现在哪几方面？

1. **响应式系统升级**：Proxy 替代 defineProperty，支持数组下标 / Map / Set，新增属性自动响应式。
2. **编译期优化**：
   - 静态节点提升（PatchFlag）；
   - 事件监听缓存；
   - 树摇友好（按需引入）。
3. **VNode 优化**：单个 VNode 体积减少，patch 更快。
4. **SSR 优化**：`@vue/server-renderer` 流式输出。
5. **体积更小**：核心 ~34KB（Vue 2 ~50KB），Tree Shaking 进一步减少实际包体积。
6. **Composition API**：逻辑复用更高效，减少 mixin 带来的不必要渲染。

V24：Vue 3 的设计目标是什么？做了哪些优化？

设计目标：

1. **更好的 TS 支持**：大型项目类型安全；
2. **更好的逻辑复用**：解决 mixin 缺陷；
3. **更好的性能**：编译期 + 运行时全面优化；
4. **更小的体积**：Tree-shakable；
5. **更灵活的 API**：同时支持 Options 和 Composition；
6. **为未来铺路**：为 Vapor 模式（无 VNode）和自定义渲染器打基础。

主要优化：

维度	优化手段
响应式	Proxy 替代 defineProperty
编译	PatchFlag、静态提升、缓存事件
Diff	最长递增子序列算法减少 DOM 移动
SSR	流式渲染、组件级缓存
包体积	Tree Shaking、按需打包

五、性能优化（23 题）

P-OPT1：`<script>` 放在 header 和 body 底部的区别？

位置	区别
header 中	浏览器遇到 `<script>` 会**同步加载并执行**，阻塞后面 DOM 解析（脚本可能 document.write 改变 DOM），白屏时间长。
body 底部	DOM 解析完成后才加载脚本，页面元素先显示出来，用户感知更快。

进阶方案：

```
<!-- defer：并行下载，DOMContentLoaded 前执行 -->
<script src="app.js" defer></script>

<!-- async：并行下载，下载完立即执行（不保证顺序） -->
<script src="analytics.js" async></script>

<!-- type="module"：默认 defer 行为 -->
<script type="module" src="app.js"></script>
```

推荐：

- 第三方库用 `defer`（依赖顺序可控）；

- 独立统计脚本用 `async`（执行越早越好）；
- 应用入口用 `defer`。

P-OPT2：前端性能优化指标有哪些？怎么检测？

核心指标（Core Web Vitals）：

指标	说明	优秀阈值
FCP（First Contentful Paint）	首次内容渲染	< 1.8s
LCP（Largest Contentful Paint）	最大内容渲染	< 2.5s
FID（First Input Delay）	首次输入延迟	< 100ms
INP（Interaction to Next Paint）	交互到下一帧	< 200ms
CLS（Cumulative Layout Shift）	累积布局偏移	< 0.1
TTI（Time to Interactive）	可交互时间	-
TBT（Total Blocking Time）	总阻塞时间	< 200ms

检测工具：

- **Lighthouse**：综合性能评分；
- **WebPageTest**：多地点、多浏览器、多网络环境测试；
- **Chrome DevTools Performance**：录制运行时性能；
- **PerformanceObserver API**：业务中上报性能数据。

```
// 上报 LCP / FID / CLS
new PerformanceObserver(list => {
  for (const entry of list.getEntries()) {
    console.log(entry.name, entry.startTime);
    sendMetric({ name: entry.name, value: entry.startTime });
  }
}).observe({ type: 'largest-contentful-paint', buffered: true });
```

P-OPT3：SPA 首屏加载速度慢怎么解决？

与 V14 互补，从性能指标角度补充：

1. **骨架屏 / Loading**：转移注意力，减少感知等待时间；
2. **路由级代码分割**：按路由懒加载，初始包体积减少 50%+；
3. **CDN 加速**：静态资源走 CDN，启用 HTTP/2；
4. **资源预加载**：`<link rel="preload" href="critical.css">`；
5. **预获取数据**：路由切换前预取下一页数据；
6. **SSR / SSG**：首屏 HTML 服务端生成，TTI 提升明显；
7. **开启 gzip / brotli 压缩**：减少 70%+ 体积；

8. **Web Vitals 监控**：上线后持续优化。

P-OPT4：用 CSS 提升页面性能

1. **减少 CSS 体积**：
   - 压缩合并（CSSNano）；
   - 移除未使用样式（PurgeCSS / UnCSS）；
   - CSS-in-JS 会增加运行时开销，需谨慎。
2. **关键 CSS 内联**：首屏所需 CSS 放在 `<style>` 中，其余异步加载。
3. **避免昂贵选择器**：避免深层后代选择器（如 `.a .b .c .d`），浏览器从右到左匹配。
4. **避免 @import**：串行加载。
5. **CSS Containment**：`contain: layout / paint / size` 限制重绘范围。
6. **will-change**：提示浏览器提前优化，但**不能滥用**（会消耗 GPU 内存）。
7. **GPU 加速属性**：`transform` / `opacity` 触发 GPU 合成，跳过重绘重排。

P-OPT5：站点内的图片性能优化

1. **选择合适格式**：
   - **WebP**：体积比 JPEG/PNG 小 25-35%；
   - **AVIF**：体积更小，压缩比优于 WebP；
   - **SVG**：适合图标、插画。
2. **按需加载**：
   - 原生 `loading="lazy"`；
   - 滚动到视口附近才加载；
   - 占位图 + 模糊到清晰过渡。
3. **响应式图片**：

```
<img srcset="small.jpg 480w, medium.jpg 800w, large.jpg 1200w"
     sizes="(max-width: 600px) 480px, 800px"
     src="medium.jpg" />
```

4. **CDN + 图片处理**：使用 `?x-oss-process=image/resize,w_800` 等 CDN 参数。
5. **压缩与裁剪**：去除 EXIF、合理质量（75-85% 肉眼难辨）。
6. **雪碧图**：HTTP/2 下作用减弱。
7. **避免 Layout Shift**：永远设置 `width` / `height` 属性。

P-OPT6：虚拟 DOM 一定更快吗?

不一定!

虚拟 DOM 适用场景：DOM 操作复杂、组件数量中等（几十到几百）。

更快的场景：

- **直接 DOM 操作**：极简单页面，框架开销大于收益；
- **细粒度响应式**（Vue / SolidJS）：精确更新，零 VNode 开销；
- **WebGL / Canvas**：不走 DOM。

虚拟 DOM 的优势不是“更快”，而是“提供合理性能 + 开发体验”：

- 避免手动管理 DOM 状态；
- 提供跨平台能力（React Native、SSR）；
- 批量更新减少 DOM 操作次数。

本质：是用 JS 计算换 DOM 操作，开发效率提升远大于性能损失。

P-OPT7：有些框架不用虚拟 DOM，性能也不错是为什么？

原因：

1. **编译期优化**：Svelte / SolidJS 在编译阶段将组件编译为命令式 DOM 操作代码，无需运行时 VNode 比较。
2. **细粒度响应式**：Vue / SolidJS 的响应式系统能精确追踪到**组件内具体属性**变化，只更新必要的 DOM 节点。
3. **跳过 VNode 层**：直接调用 DOM API，省去 diff 计算与对象创建开销。
4. **举例**：
   - **SolidJS**：类似 React 的 JSX，但编译产物是“订阅-更新”函数；
   - **Svelte**：编译为高度优化的 vanilla JS；
   - **Vue 3 + Vapor**：Vue 3 的无 VNode 模式。

总结：**没有 VNode 也能做到高效**，关键是“精准更新”。

P-OPT8：几百个函数需要执行，怎么优化页面？

方案：

1. **Web Worker**：把 CPU 密集任务放到后台线程，避免主线程卡顿。

```
const worker = new Worker('worker.js');
worker.postMessage(data);
worker.onmessage = e => console.log(e.data);
```

2. **任务切片 + 调度器**：

```
function chunk(tasks: Function[], chunkSize = 50) {
  let i = 0;
  function next() {
```

```
      const slice = tasks.slice(i, i + chunkSize);
      i += chunkSize;
      slice.forEach(fn => fn());
      if (i < tasks.length) requestIdleCallback(next);
    }
    next();
  }
```

3. **requestIdleCallback / requestAnimationFrame**：在浏览器空闲时执行。
4. **防抖节流**：高频触发合并处理。
5. **WebAssembly**：计算密集型（图形、加密）场景。

P-OPT9：png8、png16、png32 的区别，png 的特点

类型	颜色数	透明度	适用
PNG8	256 色	不支持	简单图标、表情
PNG24	约 1600 万色	不支持	复杂图片、照片
PNG32	约 1600 万色	8 位 alpha	需透明背景的复杂图片

PNG 特点：

- 无损压缩，质量高；
- 支持 alpha 透明；
- 文件体积通常比 JPEG 大，不适合照片（应选 JPEG / WebP）；
- 适合 UI 切图、Logo、图标。

P-OPT10：页面加载过程中，JS 文件一定会阻塞 DOM 解析吗?

不一定!

加载方式	阻塞情况
普通 `<script src="...">`	**阻塞**：加载并同步执行
`<script defer>`	不阻塞（并行下载，DOMContentLoaded 前执行）
`<script async>`	不阻塞（并行下载，下载完立即执行）
`<script type="module">`	默认 defer 行为
动态创建 `<script>`	不阻塞 DOMContentLoaded

注意：

- `defer` / `async` 脚本**仍会阻塞 `load` 事件**（执行完成后才触发）；
- `defer` 脚本按顺序执行，`async` 不保证顺序。

P-OPT11：React.memo 和 useMemo 的用法与区别

维度	React.memo	useMemo
类型	组件级	Hook（值级）
作用	浅比较 props，相同则跳过渲染	缓存计算结果
返回	记忆化组件	记忆化值
场景	子组件不需随父组件重渲染	复杂计算 / 引用稳定性

```
// React.memo
const Child = React.memo(function Child({ value }) {
  console.log('Child render');
  return <div>{value}</div>;
});

// useMemo
const sorted = useMemo(() => list.sort((a, b) => a.id - b.id), [list]);

// useCallback（缓存函数）
const onClick = useCallback(() => doSomething(id), [id]);
```

注意：React.memo 浅比较对象 props 失效时需自定义比较函数（第二个参数）。

P-OPT12：页面加载白屏时间长的原因与优化

原因排查：

1. **HTML 体积过大**：服务器响应慢；
2. **关键 CSS / JS 阻塞渲染**：未使用 defer / async；
3. **第三方脚本拖累**：广告 SDK、统计代码；
4. **首屏 JS 体积大**：未做代码分割；
5. **API 响应慢**：数据返回后再渲染页面；
6. **大量同步计算**：主线程阻塞。

优化方案：

- 启用 CDN + gzip；
- 内联关键 CSS；
- 路由懒加载；
- SSR / 骨架屏；
- 异步数据预取；
- 移除未使用的第三方依赖。

P-OPT13：100000 条数据的列表如何展示?

核心：虚拟列表（Virtual List）

原理：只渲染视口内可见的列表项（约 10-30 个），滚动时动态替换 DOM。

主流库：

- React：`react-window`、`react-virtuoso`、`react-virtualized`；
- Vue：`vue-virtual-scroller`；
- 通用：自实现基于 `IntersectionObserver`。

简化实现思路：

```
function VirtualList({ items, itemHeight, height }) {
  const [scrollTop, setScrollTop] = useState(0);
  const startIdx = Math.floor(scrollTop / itemHeight);
  const visibleCount = Math.ceil(height / itemHeight);
  const visibleItems = items.slice(startIdx, startIdx + visibleCount);
  const offsetY = startIdx * itemHeight;

  return (
    <div style={{ height, overflow: 'auto' }} onScroll={e =>
setScrollTop(e.target.scrollTop)}>
      <div style={{ height: items.length * itemHeight, position: 'relative' }}>
        <div style={{ transform: `translateY(${offsetY}px)` }}>
          {visibleItems.map(item => <Row key={item.id} {...item} />)}
        </div>
      </div>
    </div>
  );
}
```

进阶：动态高度、分片加载、滚动到顶部触发加载更多。

P-OPT14：DNS 预解析是什么？怎么实现？

作用：在用户点击链接前，浏览器提前解析域名到 IP，减少 DNS 查询延迟（50-200ms）。

```
<!-- 开启 DNS 预解析 -->
<link rel="dns-prefetch" href="//cdn.example.com">

<!-- 预连接（DNS + TCP + TLS，比预解析更彻底） -->
<link rel="preconnect" href="https://api.example.com">

<!-- 预加载（提前加载关键资源） -->
<link rel="preload" href="critical.css" as="style">

<!-- 预获取（下一页资源，优先级低） -->
<link rel="prefetch" href="/next-page.js">
```

Chrome 默认开启：TLS 握手中的 hostname 也会被预解析。

P-OPT15：React 中可以做哪些性能优化？

详见 Q10，补充：

1. **生产构建**：生产环境 React 会移除 PropTypes / DevTools；
2. **使用 Profiler**：定位瓶颈组件；
3. **不可变数据**：避免 `setState` 误判；
4. **键列表正确使用 key**：避免 DOM 重建；
5. **React DevTools**：可视化分析组件渲染次数；
6. **React.lazy + Suspense**：按需加载；
7. **避免 inline 对象 / 函数**：每次渲染都是新引用，会破坏 memo。

P-OPT16：浏览器为什么限制并发数？

原因：

1. **保护服务器**：避免恶意页面同时打开上百个连接压垮源站；
2. **保护客户端**：每个 TCP 连接占用内存（接收/发送缓冲区）、端口；
3. **HTTP/1.1 队头阻塞**：同域 6 个并发（Chrome / Firefox），其他浏览器 2-8 个不等；
4. **历史包袱**：RFC 2616 建议客户端不超过 2 个并发连接。

不同浏览器并发数（HTTP/1.1 同域）：

浏览器	并发数
Chrome	6
Firefox	6
Safari	6
IE 11	8
IE 8-10	6-8

HTTP/2 解决：多路复用，单连接可承载无数请求。

P-OPT17：Performance API 是什么？怎么确定页面可用性时间？

Performance API：浏览器提供的性能监控接口。

关键时间点：

```
const t = performance.timing; // 已废弃，改用 PerformanceNavigationTiming
const nav = performance.getEntriesByType('navigation')[0];

const metrics = {
  // DNS 查询
  dns: nav.domainLookupEnd - nav.domainLookupStart,
```

```
    // TCP 连接
    tcp: nav.connectEnd - nav.connectStart,
    // 首字节时间
    ttfb: nav.responseStart - nav.requestStart,
    // 响应下载
    download: nav.responseEnd - nav.responseStart,
    // DOM 解析
    domParse: nav.domInteractive - nav.responseEnd,
    // 资源加载
    resourceLoad: nav.loadEventStart - nav.domContentLoadedEventEnd,
    // 首屏可交互
    fmp: nav.domContentLoadedEventEnd - nav.startTime
  };
```

可用性时间：`domInteractive`（DOM 可交互）或 `domContentLoaded`（DOM + 同步脚本完成）。

P-OPT18：window.requestAnimationFrame 是什么?

概念：浏览器在下一次重绘前调用指定回调函数，通常以 60fps 频率（16.7ms）。

```
function animate() {
  el.style.transform = `translateX(${x++}px)`;
  requestAnimationFrame(animate);
}
requestAnimationFrame(animate);
```

优势：

- 与显示器刷新率同步；
- 标签页隐藏时**自动暂停**，节省资源；
- 比 `setTimeout(fn, 16)` 更精准；

典型场景：

- 动画 / 滚动联动；
- Canvas 绘制；
- 复杂 UI 渐进展示。

配套 API：

- `requestIdleCallback`：浏览器空闲时调用，适合非关键任务；
- `cancelAnimationFrame`：取消回调。

P-OPT19：CSS 加载会造成阻塞吗?

会，但有条件：

加载方式	阻塞

加载方式	阻塞
`<link rel="stylesheet">`	**阻塞渲染**（但不阻塞 DOM 解析）
内联 `<style>`	阻塞渲染（取决于大小）
`media="print"` / `media="(max-width: 0)"`	**不阻塞**（媒体查询不匹配）
`rel="preload" as="style"`	不阻塞（需配合 `onload` 应用）

阻塞原因：CSS 可能改变布局 / 样式，浏览器需等待 CSSOM 构建完成才能渲染。

优化：

```
<!-- 关键 CSS 内联 -->
<style>/* critical CSS */</style>

<!-- 非关键 CSS 异步加载 -->
<link rel="preload" href="main.css" as="style" onload="this.rel='stylesheet'">
```

P-OPT20：什么是内存泄漏？什么原因导致？

定义：程序中不再使用的内存未被释放，长期累积导致应用卡顿甚至崩溃。

常见原因：

1. **未清理的定时器 / 事件监听**：`setInterval` / `addEventListener` 未在 unmount 时清理；
2. **闭包持有 DOM 引用**：变量无意中保存了已移除 DOM；
3. **意外的全局变量**：未用 var/let/const 直接赋值；
4. **游离的 DOM 引用**：Map / WeakMap 中 key 是 DOM 节点，被移除后未清理；
5. **循环引用**：老 IE 问题，现代浏览器 GC 优化后较少；
6. **未释放的 WebSocket / EventSource。**

排查工具：

- Chrome DevTools Memory 面板；
- `performance.memory`（Chrome 扩展 API）；
- 堆快照对比。

```
// 正确写法
onMounted(() => {
  const timer = setInterval(tick, 1000);
  const handler = () => console.log('resize');
  window.addEventListener('resize', handler);

  onBeforeUnmount(() => {
    clearInterval(timer);
    window.removeEventListener('resize', handler);
```

```
  });
});
```

P-OPT21：用 Webpack 优化前端性能

1. **缩小打包体积**：
   - Tree Shaking（`mode: 'production'` 默认开启）；
   - Scope Hoisting（`optimization.concatenateModules`）；
   - 按需引入（`babel-plugin-import`）；
   - 图片压缩（`image-webpack-loader`）；
   - Gzip / Brotli 压缩。
2. **拆包策略**：

```
splitChunks: {
  chunks: 'all',
  cacheGroups: {
    vendor: { test: /[\\/]node_modules[\\/]/, name: 'vendor' },
    common: { minChunks: 2, name: 'common' }
  }
}
```

3. **代码分割**：`import()` 动态导入；
4. **持久化缓存**：`cache: { type: 'filesystem' }`；
5. **多线程**：`thread-loader`、`terser-webpack-plugin`（默认多核）；
6. **DLL / Module Federation**：共享依赖；
7. **SourceMap 区分环境**：开发用 eval-cheap-module，生产去掉；
8. **Bundle 分析**：`webpack-bundle-analyzer`。

P-OPT22：常规的前端性能优化手段

系统化总结（八层优化）：

层级	手段
网络层	CDN、HTTP/2、DNS 预解析、预连接、强缓存
资源层	压缩、合并、雪碧图、WebP、字体子集化
加载层	懒加载、预加载、代码分割、Tree Shaking
渲染层	SSR、SSG、骨架屏、避免重排重排

层级	手段
运行时	Web Worker、防抖节流、虚拟列表
代码层	算法优化、避免全局查找、缓存计算结果
框架层	React.memo、Vue v-once、组件细分
监控层	Lighthouse、Web Vitals、APM

原则：先测量、再优化、最后监控。

P-OPT23：什么是 CSS Sprites?

概念：将多个小图标合并为一张大图，通过 `background-position` 定位显示。

```
.icon { background: url('sprites.png') no-repeat; }
.icon-home { background-position: 0 0; width: 16px; height: 16px; }
.icon-user { background-position: -16px 0; width: 16px; height: 16px; }
```

优点：

- 减少 HTTP 请求数（一次下载多图）；
- 利于缓存（合并图一次缓存，多次复用）。

缺点：

- 维护成本高（新增图标需重新拼图）；
- 不支持 Retina 高清屏（需提供 2x 图）；
- HTTP/2 下意义不大（多路复用已减少请求成本）。

现代方案：

- **SVG Sprite**：多个 SVG 合并为一个 `<symbol>`，`<use href="#icon-home" />` 引用；
- **字体图标**：Iconfont / Iconify；
- **CDN 图标服务**：按需加载 SVG。

六、工程化（23 题）

E1：package.json 中 devDependencies 和 dependencies 区别?

字段	说明	打包是否包含
dependencies	**运行时**依赖，项目运行必需	是
devDependencies	**开发时**依赖，仅本地开发、测试、构建需要	否
peerDependencies	同伴依赖，期望使用者提供	由宿主安装
optionalDependencies	可选依赖，安装失败不报错	由宿主选择

示例：

```
{
  "dependencies": {
    "vue": "^3.4.0",          // 运行必需
    "axios": "^1.6.0"
  },
  "devDependencies": {
    "vite": "^5.0.0",         // 构建工具
    "typescript": "^5.3.0",
    "@types/node": "^20.0.0"
  }
}
```

最佳实践：

- 发布 npm 包时，`peerDependencies` 声明框架版本，避免重复安装；
- 发布组件库时，业务依赖放 `dependencies`，开发工具放 `devDependencies`；
- 使用 `npm i --save-dev xxx` / `npm i xxx` 区分。

E2：Webpack 5 的主要升级点

1. **持久化缓存**：`cache: { type: 'filesystem' }`，二次构建提速 90%+；
2. **资源模块（Asset Modules）**：取代 `file-loader` / `url-loader` / `raw-loader`，默认内置；
3. **Module Federation**：原生支持微前端、跨应用共享模块；
4. **Tree Shaking 增强**：能追踪到嵌套模块导出、支持 `sideEffects: false`；
5. **Web Worker 支持**：`new Worker(new URL('./worker.js', import.meta.url))`；
6. **Node Polyfills 自动移除**：仅对前端需要的 polyfill 提供；
7. **新生成器（experiments）**：`output.module: true` 输出 ESM；
8. **更好的 SourceMap**：`devtool: 'eval-cheap-module-source-map'` 更精准；
9. **Chunk 拆分算法优化**：默认 `splitChunks` 策略更智能。

E3：Vite 的原理

（与 V15 互补，更详细）

冷启动快的原因：

1. **不走打包**：开发模式不做全量 bundling；
2. **预构建依赖**：用 **esbuild**（Go 编写，比 JS 快 10-100 倍）预编译 `node_modules` 为 ESM；
3. **按需编译**：浏览器请求哪个模块才编译哪个，请求与编译并行；
4. **原生 ESM**：浏览器原生加载 ESM，无需 IIFE / AMD 包装。

热更新 HMR：

```
// vite 内部
const watcher = chokidar.watch(root, { ignored: ['**/.git/**',
```

```
  '**/node_modules/**'] });
  watcher.on('change', async file => {
    await invalidateModule(file);
    ws.send({ type: 'update', path: file, timestamp: Date.now() });
  });
```

- WebSocket 推送更新；
- 浏览器仅请求变更的模块；
- 框架插件（vue / react-refresh）接收更新，热替换组件。

生产构建：使用 **Rollup**（生态丰富、产物优化好）。

E4：与 Webpack 类似的工具及区别

工具	定位	特点
Vite	新一代构建工具	开发用 esbuild + 原生 ESM，生产用 Rollup
Rollup	库打包首选	输出更干净，Tree Shaking 优秀
Parcel	零配置	自动处理依赖，但生态较小
esbuild	Go 写的极速编译器	仅做转换，不做打包
swc	Rust 写的 Babel 替代	比 Babel 快 20 倍
Turbopack	Webpack 作者新项目	Rust 编写，Webpack 继任者（Next.js 团队）
Bun	全能运行时	自带打包器，性能优异
Rspack	Rust 版的 Webpack	兼容 Webpack 生态，速度提升 5-10 倍

选型建议：

- **应用开发**：Vite（首选）、Rspack（大型项目）；
- **组件库开发**：Rollup；
- **极致构建速度**：esbuild / swc；
- **存量 Webpack 项目**：短期不动，长期考虑迁移。

E5：如何借助 Webpack 优化前端性能

（与 P-OPT21 互补，补充更多细节）

1. 体积优化

```
// webpack.config.js
module.exports = {
  mode: 'production', // 启用 production 内置优化
  optimization: {
    minimize: true,
    minimizer: [new TerserPlugin({ parallel: true })],
```

```
    splitChunks: {
      chunks: 'all',
      cacheGroups: {
        vendor: {
          test: /[\\/]node_modules[\\/]/,
          priority: 10,
          name: 'vendor'
        }
      }
    }
  },
  performance: {
    hints: 'warning',
    maxAssetSize: 250000,
    maxEntrypointSize: 250000
  }
};
```

2. 加载优化

```
// 动态导入
const LazyComp = () => import('./LazyComp.vue');

// Prefetch / Preload
import(/* webpackPrefetch: true */ './LoginModal');
```

3. 持久化缓存

```
cache: {
  type: 'filesystem',
  buildDependencies: { config: [__filename] }
}
```

E6：Webpack Proxy 工作原理与跨域解决

跨域原因：浏览器同源策略，协议 / 域名 / 端口不同即为跨域。

Proxy 解决方案：

```
// vue.config.js / webpack.config.js
module.exports = {
  devServer: {
    proxy: {
      '/api': {
        target: 'http://backend.com',
        changeOrigin: true,
```

```
        pathRewrite: { '^/api': '' }
      }
    }
  }
};
```

工作原理：

1. 开发服务器（webpack-dev-server / vite）启动 HTTP 服务；
2. 浏览器请求 `/api/users` 实际请求本地开发服务器（同源）；
3. 开发服务器将请求转发到真实后端（代理突破浏览器同源限制）；
4. 后端响应返回给开发服务器，再返回给浏览器。

为什么能解决跨域：

- 浏览器只关心**协议 + 域名 + 端口**；
- 跨域是浏览器行为，服务器之间无跨域限制；
- Proxy 是**服务器端代理**，绕过了浏览器同源策略。

注意：仅开发环境有效，生产需用 Nginx / 后端 CORS 配置。

E7：Webpack 热更新原理

HMR（Hot Module Replacement）：运行时替换、添加、删除模块，**无需刷新页面**。

工作流程：

1. **监听文件变化**：webpack 使用 `webpack-dev-middleware` 调用 `chokidar` 监听；
2. **重新编译**：文件变化后，webpack 增量编译（仅编译变更模块）；
3. **推送更新**：webpack-dev-server 通过 **WebSocket** 向浏览器推送 `{ type: 'hash', data: hash }`；
4. **拉取更新**：浏览器收到通知，通过 `HotModuleReplacementPlugin` 运行时请求 `hash.hot-update.json` 和 `chunk.hash.hot-update.js`；
5. **模块替换**：在浏览器端按模块 ID 替换旧模块，触发 `module.hot.accept` 回调。

核心代码：

```
// 业务模块
if (module.hot) {
  module.hot.accept('./library.js', () => {
    // 重新执行使用 library 的代码
    doSomething();
  });
}
```

E8：Loader 和 Plugin 的区别？如何编写？

维度	Loader	Plugin

维度	Loader	Plugin
作用	**转换**模块源码	扩展 webpack **生命周期**
配置	`module.rules`	`plugins` 数组
本质	函数，接收源码返回新源码	类，apply 方法挂载钩子
数量	多对一处理	全局干预

编写 Loader（示例：自定义 markdown-loader）

```
module.exports = function(source: string) {
  // this 是 webpack 提供的上下文
  this.cacheable && this.cacheable();
  const html = marked(source);
  return `module.exports = ${JSON.stringify(html)}`;
};
// module.exports.raw = true; // 如果要接收二进制
```

编写 Plugin（示例：自定义 WriteFilePlugin）

```
class WriteFilePlugin {
  apply(compiler: any) {
    compiler.hooks.emit.tapAsync('WriteFilePlugin', (compilation: any, cb:
Function) => {
      for (const [name, asset] of Object.entries(compilation.assets)) {
        require('fs').writeFileSync(`./dist/${name}`, asset.source());
      }
      cb();
    });
  }
}
```

E9：常见 Webpack Plugin 及解决的问题

Plugin	作用
HtmlWebpackPlugin	自动生成 HTML 并注入 bundle
MiniCssExtractPlugin	提取 CSS 到单独文件
TerserPlugin	JS 压缩
CssMinimizerPlugin	CSS 压缩
DefinePlugin	定义编译时全局常量
CopyWebpackPlugin	拷贝静态资源

Plugin	作用
CleanWebpackPlugin	清理输出目录
BundleAnalyzerPlugin	可视化分析包体积
HotModuleReplacementPlugin	启用 HMR
ProvidePlugin	自动加载模块（如 $）
IgnorePlugin	忽略指定模块
CompressionPlugin	预生成 gzip 文件
ForkTsCheckerWebpackPlugin	TS 类型检查
ModuleFederationPlugin	微前端 / 模块联邦

E10：常见 Webpack Loader 及解决的问题

Loader	作用
babel-loader	ES6+ → ES5
ts-loader / **esbuild-loader** / **swc-loader**	TS / JSX 转译
css-loader	解析 CSS 文件中的 `@import` / `url()`
style-loader	将 CSS 注入 `<style>`
sass-loader / **less-loader** / **stylus-loader**	CSS 预处理器
postcss-loader	PostCSS 处理（autoprefixer 等）
file-loader（已废） / **Asset Modules**	处理文件资源
url-loader（已废）	小图转 base64
vue-loader	Vue 单文件组件
eslint-loader / **stylelint-loader**	代码检查
raw-loader	以字符串形式导入文件
thread-loader	多线程加速

E11：Webpack 的构建流程

六大阶段：

1. **初始化（Initialization）**：
   - 读取配置、加载 Plugin、注册 Compiler 钩子；
2. **编译（Compilation）**：
   - 从入口文件出发，调用 Loader 转换模块；
   - 构建模块依赖图（Module Graph）；
3. **完成编译（Finish Make）**：

    - 收集所有模块、依赖关系；
4. **封装（Sealing）**：
    - 生成 Chunk、合并 SplitChunks、优化产物；
    - 触发 Plugin 钩子；
5. **输出（Emit）**：
    - 将 Chunk 写入磁盘；
    - 触发 `emit` 钩子；
6. **完成（Done）**：
    - 触发 `done` 钩子。

简化流程图：

```
entry → loader处理模块 → 构建依赖图 → splitChunks → 优化 → 写入dist
```

E12：对 Webpack 的理解？解决了什么问题？

Webpack 本质：静态模块打包器（Static Module Bundler）。

核心能力：

1. **模块化**：支持 ESM / CommonJS / AMD / CSS / 图片等一切资源作为模块；
2. **依赖管理**：自动解析模块依赖，构建依赖图；
3. **代码分割**：按需加载、提取公共代码；
4. **Loader / Plugin 生态**：转换和扩展能力强；
5. **开发体验**：HMR、Source Map、devServer。

解决的问题：

- 浏览器不识别 `import` / `require`，需打包成单文件；
- 多文件加载导致 HTTP 请求过多；
- 高级语法（TS / JSX / Sass）浏览器不识别；
- 代码组织混乱，无法模块化开发。

生态地位：曾经的事实标准（Vue 2 / React 主流选择），现在被 Vite 挑战。

E13：Loader 和 Plugin 的实现原理

Loader 实现原理：

- 本质是一个函数，接收模块源码，返回转换后源码；
- webpack 调用 Loader 时通过 `this` 注入上下文（query / cacheable / addDependency 等）；
- Loader 默认接收字符串，可通过 `module.exports.raw = true` 接收 Buffer；
- 支持链式调用（从右到左、从下到上）。

Plugin 实现原理：

- 本质是一个带 `apply` 方法的类；

- `apply` 接收 `compiler` 对象，通过 `compiler.hooks.<name>.tap(...)` 注册钩子回调；
- webpack 在不同生命周期触发钩子，Plugin 在特定时机介入。

常用钩子：

```
compiler.hooks.beforeRun.tap('Plugin', () => {});
compiler.hooks.compile.tap('Plugin', () => {});
compilation.hooks.optimize.tap('Plugin', () => {});
compilation.hooks.emit.tapAsync('Plugin', (compilation, cb) => cb());
```

E14：如何提高 Webpack 构建速度

1. **升级到 Webpack 5** + 开启 `cache: { type: 'filesystem' }`；
2. **多线程**：`thread-loader`、`terser-webpack-plugin`（默认多核）；
3. **缩小构建范围**：

```
module: {
  rules: [
    { test: /\.js$/, include: /src/, exclude: /node_modules/ }
  ]
}
```

4. **DLLPlugin**：预打包不常变动的依赖；
5. **减少 Loader 处理**：少用昂贵的 Loader（babel-loader → swc-loader / esbuild-loader）；
6. **优化 resolve 配置**：

```
resolve: {
  modules: [path.resolve('node_modules')],
  extensions: ['.js', '.json'],
  alias: { '@': path.resolve('src') }
}
```

7. **HappyPack / thread-loader**：多进程并行；
8. **动态 polyfill**（按需加载）；
9. **可视化分析**：`speed-measure-webpack-plugin` 找瓶颈；
10. **升级 Node 版本**（新 Node 启动更快）。

E15：webpack-dev-server 原理

核心职责：启动 HTTP 服务 + 内存编译 + 监听变化 + 推送 HMR。

工作流程：

1. **启动**：读取 webpack 配置，调用 webpack 编译；

2. **内存编译**：编译产物存入**内存文件系统**（memfs），不写磁盘；
3. **HTTP 服务**：express / http-proxy-middleware 启动服务器；
4. **路由转发**：请求 `bundle.js` 时从内存文件系统读取；
5. **监听文件**：chokidar 监听源文件变化，重新编译；
6. **HMR 推送**：WebSocket 向浏览器发送更新事件；
7. **代理配置**：通过 `http-proxy-middleware` 转发 API 请求。

对比 Vite：

- webpack-dev-server 每次修改都需重新构建依赖图（即使改动很小）；
- Vite 利用浏览器原生 ESM + 按需编译，几乎瞬时响应。

E16：Babel 的 stage 代表什么意思？

TC39 阶段：JavaScript 提案进入标准要经过 5 个阶段（Stage 0 ~ 4），Babel 通过 `@babel/preset-env` 的 `stage` 配置控制启用哪些提案。

Stage	名称	含义
Stage 0	Strawman	任意想法，可能进入规范
Stage 1	Proposal	正式提案，值得讨论
Stage 2	Draft	草案，初步规范
Stage 3	Candidate	候选，预计纳入规范
Stage 4	Finished	已纳入规范，下个版本发布

注意：

- Babel 7 之后**不再推荐**按 stage 配置，建议用 `@babel/preset-env` + 浏览器目标自动注入 polyfill；
- Stage 4 已纳入 ES 标准，浏览器原生支持，无需 Babel 转换。

E17：Webpack 的 module、bundle、chunk 区别？

概念	定义
module	源码模块（JS / CSS / 图片），webpack 处理的基本单位
chunk	打包过程中的代码块（一个或多个 module 组成），由 webpack 内部管理
bundle	输出的最终文件（通常是 chunk 的产物），运行在浏览器
chunk group	一组 chunk 的集合

关系示例：

```
src/index.js  →  module
src/utils.js  →  module
src/style.css →  module
```

```
webpack 打包后:
  entry chunk (main.js)    → 由 entry 引用的 module 组成
  vendor chunk (vendors.js) → 由 node_modules 中的 module 组成
  async chunk (login.js)   → 由动态 import 的 module 组成

这些 chunk 输出到 dist/ 后称为 bundle。
```

E18：什么是 CI/CD?

CI（Continuous Integration，持续集成）：

- 频繁地将代码集成到主干（每天多次）；
- 每次集成通过自动化构建、测试、代码检查快速验证；
- 目标：尽早发现集成错误。

CD（Continuous Deployment / Delivery，持续部署 / 交付）：

- **持续交付**：代码通过 CI 后自动打包成可发布版本，需**人工确认**发布；
- **持续部署**：代码通过 CI 后**自动发布到生产环境**，无需人工。

典型流程：

```
push → CI触发
  → 代码拉取 → 安装依赖 → 单元测试 → 代码检查
  → 构建 → 集成测试 → 打包 → 上传产物
→ CD触发
  → 灰度发布 → 全量发布 → 监控告警
```

常用工具：

- **CI**：Jenkins、GitLab CI、GitHub Actions、CircleCI、Travis CI；
- **CD**：Argo CD、Spinnaker、Jenkins X；
- **代码检查**：ESLint、SonarQube。

E19：前端工程化的理解?

定义：使用一系列工具、规范、流程将前端开发过程标准化、自动化，提升开发效率与代码质量。

核心组成：

维度	工具 / 规范
开发规范	ESLint、Prettier、EditorConfig、commitlint
模块化	ESM、CommonJS、UMD
组件化	React/Vue 组件库、Storybook

维度	工具 / 规范
构建工具	Webpack、Vite、Rollup
包管理	npm、yarn、pnpm（monorepo）
测试	Jest、Vitest、Cypress、Playwright
版本控制	Git、GitFlow
CI/CD	GitHub Actions、Jenkins
部署	Docker、Nginx、Vercel
监控	Sentry、Lighthouse、Web Vitals

目标：

- 提升团队协作效率；
- 降低 bug 率；
- 缩短发布周期；
- 提升代码可维护性。

E20：对 SSG 的理解？

SSG（Static Site Generation，静态站点生成）：构建时生成完整 HTML 静态文件，部署到 CDN 直接返回。

与 SSR / CSR 对比：

模式	渲染时机	适用
CSR（客户端渲染）	浏览器加载 JS 后渲染	后台应用、SPA
SSR（服务端渲染）	每次请求服务端渲染	内容频繁更新
SSG（静态生成）	**构建时**预渲染	博客、文档、营销页

优点：

- 首屏极快（直接返回 HTML）；
- SEO 友好；
- CDN 缓存成本低；
- 安全性高（无运行时服务端逻辑）。

缺点：

- 构建时间长（页面多时）；
- 不适合实时数据；
- 数据更新需重新构建。

框架支持：

- **Next.js**（`getStaticProps`）；
- **Nuxt 3**（`nuxt generate`）；

- **Astro**（默认 SSG）；
- **VitePress**（文档站首选）。

E21：聊聊 Vite 和 Webpack 的区别

维度	Webpack	Vite
打包策略	全量打包后启动	按需编译 + 原生 ESM
冷启动	慢（10s+）	极快（<1s）
HMR	较慢（重新编译 + 推送）	极快（仅更新变更模块）
生产构建	Webpack 5 / Terser	Rollup
生态	极其丰富（Loader / Plugin 1000+）	较小（但增长快）
配置复杂度	较复杂	极简（约定大于配置）
TypeScript	ts-loader / babel	内置 esbuild
JSX	babel-loader	内置 esbuild
CSS	css-loader + style-loader	内置
大项目	经过考验	逐渐成熟
适用规模	任意规模	中小型项目更优，大型项目渐可

选择建议：

- **新项目**：优先 Vite（开发体验好）；
- **大型复杂项目**：Webpack 5 + 优化（生态成熟）；
- **存量项目**：短期不动，长期评估迁移成本。

E22：Webpack Tree Shaking 原理

原理：基于 ES Module 静态分析，删除未被使用的导出代码。

工作条件：

1. **使用 ESM 语法**（`import` / `export`），不能是 CommonJS；
2. **设置** `mode: 'production'`（默认开启）；
3. **标记无副作用**：在 `package.json` 中添加 `"sideEffects": false` 或数组形式 `["*.css"]`；
4. **避免副作用代码**：模块顶层不能有立即执行的副作用（修改全局变量、polyfill）。

原理流程：

1. **入口分析**：从 entry 开始标记所有使用的导出；
2. **依赖标记**：递归遍历 import 语句，标记所有引用的导出；
3. **未引用清除**：未被标记的导出在压缩阶段删除；
4. **作用域提升（Scope Hoisting）**：合并模块，减少函数声明。

```
// package.json
{
  "sideEffects": false  // 或 ["*.css"]
}

// a.js
export const used = 1;
export const unused = 2; // 被 Tree Shaking 删除

// index.js
import { used } from './a';
console.log(used);
```

E23：介绍下 Tree Shaking 及其工作原理

概念回顾：Tree Shaking 是移除 JavaScript 上下文中**未被引用代码**（dead-code elimination）的优化手段，源自 ES Module 静态结构特性。

工作流程（webpack 视角）：

1. **收集 exports**：从入口出发，分析所有 ES Module 的 export；
2. **标记 usedExports**：从 entry 开始递归，跟踪哪些 export 被实际使用；
3. **生成 usedExports 标记**：webpack 内部标记哪些 export 保留、哪些删除；
4. **Terser 压缩**：根据标记删除未使用代码；
5. **合并模块**（Scope Hoisting）：多个模块合并为一个函数，减少闭包与函数声明。

前提条件：

- ES Module 语法（`import` / `export`）；
- `mode: 'production'`；
- `package.json` 设置 `"sideEffects": false`；
- 代码无运行时副作用。

失效场景：

- 第三方库使用 CommonJS；
- 模块顶层有副作用（`window.xxx = ...`）；
- 动态导入的命名空间（`import * as`）；
- Babel 转译 CommonJS（需配置 `modules: false`）。

附录：面试答题模板与复习建议

一、万能答题结构（STAR-R 法则）

```
S - Situation：背景 / 面试官为什么问
T - Task：要解决的问题
A - Action：你的解决方案 / 原理
```

```
R - Result：效果 / 适用场景
R - Reflection：你的深入思考 / 最佳实践
```

二、各模块复习优先级

阶段	重点
3-5 年	React Hooks、Vue 响应式、TS 基础、Webpack 基础
5-8 年	Fiber、Concurrent、SSR/SSG、微前端、性能优化体系
8 年+	框架原理、编译原理、跨端架构、工程体系设计

三、高频加分项

1. 能讲清 **React Fiber 调度原理**；
2. 能手写 **响应式核心 5 行**（track / trigger）；
3. 能说出 **3 种以上首屏优化方案并量化收益**；
4. 能解释 **Vite 为什么比 Webpack 快**；
5. 能画出 **webpack 构建流程图**；
6. 能列举 **5 种以上 TypeScript 高级类型。**

四、推荐学习资源

- **React 官方文档**（新版含并发模式讲解）；
- **Vue 3 官方文档 + 源码阅读**；
- **TypeScript 官方 Handbook**；
- **Webpack 官方文档 + 源码**；
- **《JavaScript 高级程序设计》《你不知道的 JavaScript》**；
- **《深入浅出 React 与 Redux》《深入浅出 Vue.js》**；
- **极客时间 / 慕课网 / 掘金小册。**

五、文档维护

本文档按主题分类整理，每题给出**原理 + 示例 + 实践建议。**建议：

- 结合自身项目经验**举一反三**；
- 每周抽出 2-3 小时**口述练习**（讲给同伴听）；
- 持续补充实际工作中遇到的细节问题；
- 关注官方博客 / GitHub Issues，追踪最新特性。

> 本文档结束。如有补充建议或勘误，请在项目中提 Issue / PR。