# Frontend Developer Aptitude Test

Demo Link:

**Advanced Theoretical Knowledge**

**React Advanced**

**. Explain how the virtual DOM works in React and how it improves performance.**

虚拟DOM（Virtual DOM）是实际DOM的轻量级JavaScript副本，它在内存中维护一个树形结构，表示页面的结构和内容。

虚拟 DOM 提升性能

1.减少直接操作真实 DOM 的次数：

真实 DOM 操作（如增删改查元素）通常比较慢，频繁操作会造成页面卡顿。虚拟 DOM 在 JS 层先做 diff 计算，把多次状态变更合并为最小的、批量的真实 DOM 操作，从而极大减少页面重绘和回流次数。

2.高效的 diff 算法：

React 的 diff 算法能在 O(n) 复杂度下快速找到最优更新路径，只渲染变动节点，避免全量重绘。

3.跨平台能力：

虚拟 DOM 让 React 能够适配多种渲染环境，不仅可以渲染到浏览器 DOM，也可以用在 React Native 等原生应用上。

**. Describe what error boundaries are in React and how to implement an error boundary component.**

1. 错误边界（Error Boundary） 是 React 16 及以后版本引入的一种特殊组件，用来捕获其子组件树中的 JavaScript 运行时错误，并优雅地进行处理，而不会导致整个 React 应用崩溃。
2. 错误边界只能捕获“渲染期间”、“生命周期方法”以及“构造函数”中的错误。

3.无法捕获事件处理函数、异步代码（如 setTimeout）、服务端渲染或自身的错误。

**Next.js Advanced**

**. How does Incremental Static Regeneration work in Next.js? Describe its mechanism and use cases.**

ISR（增量静态生成） 是 Next.js 提供的一种页面预渲染机制，它允许你在构建后，不用重新部署整个网站，就能在后台自动更新静态页面，实现“静态页面的增量刷新”。

工作机制

1.页面首次请求：静态生成

在 Next.js 中，如果你的页面使用了 getStaticProps，首次构建时会静态生成 HTML 文件并缓存到 CDN/服务器。用户访问该页面时，会直接返回这个静态 HTML，速度极快，SEO 友好。

2.设置 revalidate 参数：后台自动再生成

你可以在 getStaticProps 里返回一个 revalidate 字段，指定页面多久“后台重新生成”一次（比如 60 秒）。

当第一个用户在 60 秒后访问该页面时：Next.js 会先返回“老的静态页面”给用户（保证秒开、不阻塞）。然后在后台自动触发 getStaticProps，用最新数据生成新的静态页面并缓存。后续用户访问时，就会直接看到最新的页面内容。

3.页面始终有静态缓存，更新无感知

ISR 结合了“静态页面（极致性能）”和“动态数据（无需全量重新部署）”的优点。

**. Explain the working mechanism of API Routes in Next.js and provide an example of how to use them in a project.**

API Routes 是 Next.js 提供的一种在项目里直接写“后端接口”的能力。你可以在 app/api/（App Router）或 pages/api/（Pages Router）目录下新建 JS/TS 文件，每个文件就是一个独立的 API 接口，可以处理 HTTP 请求（如 GET、POST、PUT、DELETE 等）。

这样，无需额外后端服务，就能在 Next.js 项目里轻松实现数据处理、表单提交、鉴权等后端逻辑。

详见demo link

**Tailwind CSS Advanced**

**. How can you achieve deep customization in Tailwind CSS, such as adding custom plugins or extending existing functionalities?**

在 Tailwind CSS v4 中实现深度自定义（deep customization），主要有三大方向：

1.扩展配置（extend theme/自定义变量）

2.编写/引入自定义插件（custom plugins）

3.重写或扩展核心功能（variant、utility、预处理等）

**. What is the JIT mode in Tailwind CSS? What are its advantages?**

JIT 模式（Just-In-Time，即时编译模式） 是 Tailwind CSS 在 v2.1 及以后版本引入的编译机制。它会在你保存/修改代码时，实时（即时）只生成你页面实际用到的 CSS 类，而不是像传统模式那样预先把所有可能的类都生成一遍。

JIT 模式的主要优势：

1.极致的编译速度和开发体验：

开发阶段 CSS 构建几乎“秒级”，你每加一个新 class，浏览器就立即能看到效果，省去反复重启构建的烦恼。

2.更小的 CSS 文件体积：

只打包你项目中用到的类，不会像传统“全量编译”把所有可能的 Tailwind 类都生成，生产环境 CSS 文件极小，页面加载更快。

3.支持任意新组合/动态 class：

你可以直接写 bg-[#123456]、w-[555px]、text-[2.5rem] 这种自定义色值/长度/字体大小，JIT 都会动态生成，不用再提前在 config 里配置。

4.极大提升开发自由度：

动态 class 名、条件渲染（如 dark:bg-black、hover:scale-110）、第三方插件等都能即时生效，开发迭代极快。

5.更好的 Purge 体验：

传统 Tailwind 需要手动配置“Purge CSS”来清除无用类，JIT 模式天然只生成你需要的，不怕“漏删”或“误删”。

**Advanced Practical Skills （ 详见demo link）**

**Complex Component Development**

**. Create a reusable React form component that supports dynamic fields and validation.**

**. Implement a simple shopping cart system using React Hooks and Context API, supporting add, remove, and checkout functionalities.**

**Next.js Complex Application**

**Frontend Developer Aptitude Test**

**Build a Next.js blog application that supports Markdown file parsing and SEO optimization.**

**Implement a data fetching scheme using GraphQL and display these data in Next.js.**

**Tailwind CSS Advanced Design**

**. Design a responsive dashboard interface using Tailwind CSS, requiring different layouts and interactions on various devices.**

**. Implement a component using Tailwind CSS animations, such as a loading indicator or transition effects.**

**Advanced Problem Solving**

**Performance Optimization**

**. How would you use React.memo and useMemo to optimize**

**performance in a React application? Provide real-world scenarios.**

1.React.memo 是一个高阶组件（HOC），用来缓存函数组件的渲染结果。

如果父组件重新渲染，但传给子组件的 props 没变，React.memo 包裹的子组件会跳过重新渲染，直接复用上一次的结果。  
适用场景：

父组件频繁渲染，但子组件 props 很少变化

例子：一个大页面中有个列表，每次输入表单父组件都会更新，但列表数据没有变化，可以用 React.memo 避免不必要的渲染。

2.useMemo 是一个 React Hook，用来缓存复杂计算的结果，只有依赖项发生变化时才重新计算，避免重复计算浪费性能。

适用场景：

组件中有复杂、耗时的计算或数据处理，每次渲染都要重复执行时。

需要传递引用稳定的对象/函数给子组件（防止触发 React.memo 或 useEffect 误判变化）。

**. Discuss the benefits and implementation methods of using CDN and image optimization features in Next.js.**

使用 CDN 的好处：

1.加速全球访问：

静态资源（图片、JS、CSS等）通过 CDN 节点就近分发，全球用户都能更快加载页面。

2.减轻服务器压力：

静态文件由 CDN 缓存和分发，源站（你的服务器）压力大大降低，带宽成本降低。

3.提升稳定性和可用性：

多节点分布，主站宕机时用户依然能访问已缓存的内容，提高可用性。

4.提升 SEO：

页面加载越快，SEO 越友好，Google/Bing 都推荐使用 CDN 加速。

Next.js 图片优化（Image Optimization）优势：

1.自动格式转换：

Next.js 的 <Image /> 组件会自动优先选择体积更小的 WebP/AVIF 等现代格式，提高加载速度。

2：自动响应式尺寸：

根据不同设备和屏幕尺寸，自动生成并分发合适大小的图片，减少流量浪费。

3.懒加载（Lazy loading）：

默认开启，首屏只加载必要图片，其余等用户滑动到可视区时再加载，提升页面响应速度。

4.支持 CDN 优化：

可以结合 Vercel Image CDN、第三方 CDN（如阿里云OSS、Cloudflare、七牛等）进一步加速图片分发。

5.缓存和再利用：

生成后的图片会自动缓存，下次访问时无需重复计算和压缩。

Next.js 下的 CDN 和图片优化实现方法

1. 使用 CDN 托管静态资源
2. 在next.config.ts中配置资源路

3.使用 Next.js 图片优化功能（Image Optimization）,官方 <Image /> 组件

**Code Refactoring and Design Patterns**

**. Refactor the following code to improve maintainability and optimize structure using design patterns:**

策略模式  
// 1. 定义策略接口

class ParseStrategy {

parse(response) {

throw new Error('parse() must be implemented');

}

}

class JsonParseStrategy extends ParseStrategy {

async parse(response) {

return response.json();

}

}

class TextParseStrategy extends ParseStrategy {

async parse(response) {

return response.text();

}

}

// 2. 定义成功/失败处理策略

class SuccessStrategy {

handle(data) {

// 默认处理

console.log('数据：', data);

}

}

class ErrorStrategy {

handle(error) {

// 默认处理

console.error('请求出错：', error);

}

}

// 3. Context：统一调度

class FetchContext {

constructor({

parseStrategy = new JsonParseStrategy(),

successStrategy = new SuccessStrategy(),

errorStrategy = new ErrorStrategy(),

} = {}) {

this.parseStrategy = parseStrategy;

this.successStrategy = successStrategy;

this.errorStrategy = errorStrategy;

}

async fetchData(url) {

try {

const response = await fetch(url);

const data = await this.parseStrategy.parse(response);

this.successStrategy.handle(data);

return data;

} catch (error) {

this.errorStrategy.handle(error);

throw error;

}

}

}

//================================

// 使用示例

const fetchJsonData = new FetchContext({

parseStrategy: new JsonParseStrategy(),

successStrategy: {

handle: (data) => {

console.log('自定义处理成功：', data);

},

},

errorStrategy: {

handle: (err) => {

alert('自定义失败处理：' + err.message);

},

},

});

fetchJsonData.fetchData('https://api.example.com/data');

**System Design and Architecture**

**1. System Design**

**. Describe how you would design the frontend architecture of a large- scale React application, including state management, component**

**layering, and routing strategy.**

整体架构设计目标：

1. 高可维护性、高扩展性、高性能、易于团队协作和长期演进
2. 明确层级分工，做到“职责单一、解耦复用”
3. 支持灵活的业务模块拆分与动态加载

状态管理方案：

1. 本地局部状态（UI/组件内部）
2. 使用 React 原生 useState、useReducer、useRef 等

3.适合只影响当前组件的简单状态（如输入值、hover/展开状态）

全局共享状态：

1. 选择主流库：Redux Toolkit、Zustand、Jotai、Recoil，根据团队技术栈和业务复杂度决定
2. Redux Toolkit 适合数据流复杂/团队协作严谨场景
3. Zustand/Jotai 适合轻量、业务弹性场景
4. 配合 React Context 实现权限、主题、用户等全局信息
5. 服务端/异步数据（如 API、缓存、订阅等）

6.持久化/同步：

对于需要持久化（localStorage、IndexedDB）或多端同步的状态，结合相应的中间件（如 redux-persist、zustand/middleware）

推荐用React Query（TanStack Query）/ SWR来管理服务器数据（异步请求、缓存、自动刷新、乐观更新等）可与 Redux 等集成，分离本地和远端状态

组件分层设计：

通常可分为以下几个层级：

1.页面级（Page）组件

1.1路由的承载容器（如 /pages/xxx.tsx、/app/xxx/page.tsx）

1.2负责页面布局、拉取主数据、权限/布局控制

1.3不直接处理业务逻辑，调度子组件/模块

2.容器（Container/Smart）组件

2.1负责数据获取、状态管理、逻辑处理，少量渲染

2.2常见如：表单容器、业务列表容器，可以用 hooks 提炼逻辑（如 useUserList）

2.3展示型（Presentational/Dumb）组件，只负责 UI 展示，通过 props 接收数据和回调

无副作用，无依赖全局状态，易于复用/单测，典型如 Card、Button、List、Table、Modal

2.4原子/基础组件（UI Library/Design System）

2.5团队自研 UI 组件库，或三方库（如 shadcn/ui、Ant Design、Material UI、Chakra）

所有展示组件应统一风格和行为

路由策略设计

1.路由库选择

1.1.Next.js 直接用文件路由（推荐大型项目，自动支持分包、动态路由、嵌套路由等）

1.2 传统 SPA 用 React Router，按需分包、嵌套路由、权限控制等

2.嵌套路由与模块拆分

2.1顶层路由对应主页面/业务线，如 /dashboard/\*、/admin/\*

2.2支持多级嵌套，如 /user/profile/settings

2.3动态路由与参数：如 /article/[id]

1. 路由懒加载与权限控制

2.1大型项目务必按需懒加载（React.lazy、Suspense、Next.js 自动分包）

2.2权限控制统一做“路由守卫”（如高阶组件 withAuth、middleware、布局包裹）

2.3支持“白名单路由”“多角色路由”管理

项目结构建议

app/ // 页面路由与布局层，底层组件等

components/ // 展示型、原子组件

content/ // provide层

config/ // 项目的一些基础定义配置

lib/ // 数据库等相关配置

web3/ // web3相关

containers/ // 容器/业务逻辑组件

hooks/ // 自定义 hooks，复用业务逻辑

store/ // 状态管理（Redux/Zustand/Recoil 等）

services/ // 接口请求（API 封装、React Query）

utils/ // 工具函数

publish/assets/ // 静态资源

layouts/ // 页面布局

middleware/ // 权限/路由守卫等

constants/ // 常量

types/ // 类型声明（TypeScript）

**. What factors would you consider when designing a multi-language Next.js application? Describe your solution.**

设计多语言 Next.js 应用时要考虑的核心因素

1.语言切换的体验与持久化

2.如何让用户自由切换语言，切换后是否记忆/持久化

3.URL 结构设计:

路径型（如 /en/about）、子域名型（如 en.site.com）还是参数型（如 /about?lang=en）

4.内容维护与可扩展性

5.多语言文案的存储、维护、动态导入与复用

6.服务端渲染（SSR）/静态生成（SSG）兼容性

7.多语言页面如何在 SSR/SSG 下 SEO 友好，且便于爬虫抓取

8.本地化（Locale）数据支持

9.日期、货币、数字、时区、单位等本地化适配

10.第三方依赖的 i18n 支持

11.UI 库、表单校验等三方组件如何适配多语言

12.SEO 和 hreflang 处理

13.搜索引擎正确识别不同语言页面，防止内容重复

14.代码组织与可维护性

15.文案/翻译与业务代码解耦，便于团队协作和持续迭代

主流解决方案：next-i18next

优点：社区成熟、基于 i18next，支持静态/服务端渲染、自动路由、切换体验佳

**Architecture Optimization**

**. Discuss how to effectively use code splitting and lazy loading in a large project to improve application performance.**

Next会自动将每个页面打包成单独的chunk，无需额外配置。

1.代码分割（Code Splitting）

2.路由级分割：每个页面/路由生成独立的 chunk

3.组件级分割：大体积/低频组件单独打包

4.第三方库分割：将大型第三方库单独分离

4.懒加载（Lazy Loading）

只有在用户真正需要时，才动态加载某个组件或资源（如滚动到可见区域才加载图片、点击按钮才加载弹窗组件等）。

进阶优化

1.按需分割第三方库:

如 moment、chart.js、monaco-editor 这类大包，遇到用到时再按需 import()

2.业务模块按路由/角色拆分

如 /admin、/user 分别打包，减少主业务代码干扰

3.骨架屏与渐进式加载体验

Suspense fallback 配合骨架屏，避免“白屏”现象

4.监控/分析 Bundle 体积

用 webpack-bundle-analyzer 或 Next.js 内置分析器查找“大 chunk”，重点拆分

. Describe how to keep the CSS file size minimal in a Tailwind CSS project.

5.使用 JIT（Just-In-Time）模式（Tailwind v3/v4 默认）

JIT 会只生成你项目实际用到的 CSS 类，写多少就生成多少，极大减少无用代码，自动瘦身。不用再手动维护 PurgeCSS 配置，省心且体积最小。

1. 配置 content 字段，精确扫描文件

7.避免滥用动态 class 拼接:

对于形如 className={'bg-' + color} 这种动态拼接，JIT 可能检测不到，要么用 safelist、要么提前写好所有变体。

8.精简/自定义 Tailwind 主题:

只扩展和覆盖实际要用的 colors、spacing、fontSize 等，不要全量引入自定义变量。

精简第三方/自定义插件使用。

9:插件只装需要的，避免体积过大的三方组件库或自定义全局样式。

1. 组件库（如 daisyUI、shadcn/ui）也建议只用到的组件导入。

**Team Collaboration and Best Practices**

**Code Quality and Style**

**. How would you promote and implement a code style guide within a team? Provide specific measures.**

1.接入代码格式化工具

1.1Prettier 自动格式化所有代码（保存时 or commit 时自动修正）

1.2在 .prettierrc 配置项目规则

2接入静态检查工具

2.1 ESLint 针对 JS/TS 代码风格和潜在错误

2.2 Stylelint 检查 CSS/SCSS/Sass 等样式规范

2.3 配合团队风格自定义 .eslintrc.js、.stylelintrc 配置

3使用 git 钩子强制校验

3.1lint-staged + husky 在每次 commit 前自动格式化+静态检查，未通过禁止提交

4集成到 CI 流程

4.1在 GitHub Actions、GitLab CI、Jenkins 等 CI 流程中加入自动检查环节，PR/合并必须通过所有规范检查

5完善工作流文档，严格执行

**Describe how to use tools like ESLint and Prettier to maintain code quality in a project.**

1. ESLint用于检测和自动修复 JavaScript/TypeScript 代码中的语法错误、风格问题和潜在 bug。支持自定义规则、插件和多种风格（如 Airbnb、Standard）。

2. Prettier用于自动格式化代码，让团队所有人的代码风格保持高度一致。专注于代码排版（如缩进、分号、引号、行宽、括号等），无须争论格式细节。

3. lint-staged 和 husky，检查代码规范

4.使用

4.1 安装依赖

4.2 根据团队风格配置ESLint与Prettier，lint-staged 和 husky

4.3 结合编辑器自动保存

4.4 在git提交前进行自动修复与代码检查

4.5 持续集成（CI）中集成代码检查

**Collaboration and Communication**

**. How do you ensure effective collaboration between frontend**

**development and other teams (e.g., backend, design) in a cross- functional team?**

1.流程与规范建设

1.1需求澄清与同步

# 1.1.1每个新需求前，前端、后端、设计、产品一起评审（需求评审会/PRD讲解），确保大家对目标、边界、交互逻辑理解一致。

1.1.2.输出“需求文档/接口文档/设计稿/交互说明”。

1.2制定接口/组件协议

1.2.1 推行 OpenAPI/Swagger 或类似工具，统一接口文档，避免因接口变更引发沟通障碍。

1.2.2 UI 组件、页面结构、交互逻辑用 Figma/Sketch 等工具明确标注。

1.3 明确交付与验收标准

1.3.1前后端定义“接口完成标准”（含字段、错误码、分页、mock 数据等），设计团队明确“还原度/自适应标准”。

2.工具驱动协作

2.1接口文档自动同步

2.1.1用 Postman、Swagger、YAPI、Apifox、Rap2 等工具统一维护接口，支持自动 mock、接口联调。

2.1.2后端接口变更自动通知前端（Webhook/飞书机器人等）。

2.2设计稿在线协作

2.2.1Figma/Sketch/Adobe XD 提供在线标注和交互演示，前端可自助提取尺寸/颜色/切图，减少低效对接。

3.组件库/设计系统共建

3.1搭建 Storybook、Styleguidist 等可视化组件库，设计/前端共同维护 UI 资产和风格基线。

3.开发/联调阶段措施

3.1Mock/分支独立开发

3.1.1接口未完成时前端用 Mock 工具造数据或用 Swagger/YAPI 自动生成接口 mock 服务，实现前后端并行。

3.1.2后端本地搭建 swagger 或线上接口联调环境。

3.2灰度/测试环境分离

3.2.1提供稳定的测试/预发布环境，前端可随时对接并提报问题，避免线上直连。

4.沟通机制与文化建设

4.1日常同步（Scrum/每日例会）

4.1.1定期（如每日 standup）同步各自进度与难点，及时发现依赖阻塞。

4.2及时反馈与答疑

4.2.1建立高效的在线沟通群（如飞书、钉钉、Slack），问题不过夜，减少“等接口/等设计”的时间。

4.3评审与复盘

4.3.1每个迭代结束有前后端联测/回归测试，设计与产品共同验收 UI/交互一致性。

4.3.2发现问题及时复盘，流程持续优化。

. Provide examples of how to give constructive feedback during code review.

1.建设性反馈

1.1以尊重、鼓励为前提，明确指出问题，并提出具体改进建议

1.2关注代码本身，不针对个人

1.3说明理由，让被评审者易于理解和接受

1.4 正例

指出问题时，语气委婉且具体

“这部分的逻辑有点难理解，建议拆分成多个函数，提升可读性。比如可以把数据处理和渲染分开。”

“这里如果输入为空会报错，建议加一个非空判断。”

“代码格式好像和团队规范不一致，可以用 Prettier 自动格式化一下。”

提出改进建议而不是只说问题

“建议把这个变量名改得更有语义，比如 userList 更容易理解。”

“你可以考虑用 React.memo 优化这个组件，避免无谓的重复渲染。”

“建议这里使用 async/await，会比 then/catch 链式写法更清晰。”

肯定优点，再建议优化

“接口抽象得很清晰，后续如果要支持分页，可以在这里加一个参数。”

“整体实现思路不错，代码结构清晰。如果能把表单校验单独提取到 utils，会更易复用。”

表达个人观点，邀请讨论

“我觉得这里用 map 可能比 forEach 更合适，这样能直接返回新数组。你怎么看？”

“我有个小建议，如果用解构赋值可以让代码更简洁，不过看你的想法～”

补充文档/注释建议

“建议给这个复杂函数加点注释，方便后面维护。”

“API 文档很详细，如果能补充下返回数据示例就更好了。”

多用鼓励语气和表情

“思路很棒，这里稍微优化一下性能就更完美了 ”

“整体实现很高效，有几点小细节可以优化，见评论～”