前端性能优化的几个大招（理论+实践，看完就是Leader水平）

前言

性能优化，一个掣肘用户体验的关键模块。它没有固定的标准定义或唯一的解决方案。但是我们从整个项目的开发-部署-用户体验的整个过程中，总能摸到很多有普适性的规范或者优化理念。

我们的目的是什么？优化啥？

应当是：**「更快的加载和响应速度、更稳定的功能表现、更简洁的代码与架构设计、更好用更人性化」**。

>
>
> 说人话是：性能优化应当是让用户能感觉到爽的，并且产生用户粘性的所有方式的总称
>

好吧，也没有那么人话...

那知道了我们性能优化的目的，我们要如何搞，才可以达到这个目的呢？

在我的脑图中，我一直将性能优化分为2大模块：

- 针对网络层面的优化，
- 针对渲染层面的优化。

这是从浏览器的渲染流程来进行分类的（老生常谈了，知道的可以跳过）

>
>
> 当用户在浏览器地址栏输入网址并按下回车后，浏览器首先通过DNS解析获取域名对应的IP地址，然后建立TCP连接（或HTTPS需额外进行TLS握手）。接着，浏览器向服务器发送HTTP请求，服务器处理请求并返回HTML等响应内容。浏览器接收响应后开始解析HTML，构建DOM树，同时解析CSS构建CSSOM树，并通过JavaScript处理动态逻辑。随后，浏览器结合DOM树和CSSOM树生成渲染树，进行布局计算确定元素的大小和位置，最终将内容绘制到屏幕上。整个过程中，外部资源如CSS、JS、图片等会并行加载，并在加载完成后动态更新页面。渲染完成后，浏览器持续加载异步资源并处理用户的交互操作，实现页面的完整性和交互性。
>

在具体实践中，可以从以下几个方面着手：

1. **「前端加载性能」**：减少首屏加载时间和资源体积，优化用户体验。
2. **「运行时性能」**：提高页面渲染和交互的流畅性，降低资源占用。

3. 「稳定性和可靠性」：确保在高并发或复杂场景下的性能表现一致。
4. 「代码维护性」：通过简洁、高效的代码实现功能，降低技术债务和后续开发成本。

无论是哪种优化策略，目标都是在满足实际业务需求的基础上，提供最佳的用户体验和技术可持续性。

实施起来总体就是这几个词：「缓存」、「压缩」、「懒加载」，做好这几个词，基本就做好了大半的性能优化。

接下来我们娓娓道来。

1. 加载时性能优化

我们要让资源加载的快一点，无非就是：资源小一点，少请求一点，当前渲染不要的东西先别加载，请求的网速快一点。我们挨个讲。

1.1 资源小一点

怎么让资源可以尽可能的小呢？基本逃离不了4个方法：

- 删除冗余代码
- 按需加载（包含懒加载）
- 细颗粒度的代码分割（其实是利用缓存策略）
- 开启gzip压缩
- 图片体积优化

接下来我们展开讲一讲

1.1.1 资源小一点：删除冗余代码

冗余代码可能来源于未使用的模块、组件、函数、样式或不再需要的第三方库。以下是常用的方法：

「方法一：Tree Shaking」

「什么是 Tree Shaking？」

Tree Shaking 是一种 **「通过静态分析移除未使用代码」** 的技术，通常用来优化前端项目中的 JavaScript 和 CSS 代码。

「说人话：静态分析就是依赖esm的语法，知道哪个模块引用了，哪个模块没引用，没引用的就删掉。」

其名称来源于“摇树”，模拟将用不到的代码从代码树中“摇掉”，使最终的打包体积更小，加载速度更快。

「Tree Shaking 的原理」

Tree Shaking 的核心依赖于 ES Module 规范 (`import/export`)，因为它是 **「静态导入」** ，可以在构建时确定哪些代码被使用。- CommonJS (`require/module.exports`) 是动态导入的，难以在编译时进行静态分析，因此不支持 Tree Shaking。

它是 Dead Code Elimination 的一个子集。它首先标记代码中哪些导入的模块未被使用，然后通过代码压缩器（如 Terser）来移除这些死代码。

「前端项目如何做 Tree Shaking? 」

1. **「使用 ESM 格式的模块」** ：
2. 确保代码使用 `import/export` ，避免 `require/module.exports` 。
3. **「配置打包工具支持」** ：
4. 不管是vite还是wepack,启用生产模式，结合构建工具和适当的配置就可以开启Tree Shaking对未使用代码进行优化。
5. **「静态导入」** ：
6. 避免动态导入或在运行时条件下选择模块，因为这些情况无法被静态分析。
7. **「避免副作用 (Side Effects)」** ：
8. 如果模块在导入时有副作用（如修改全局变量），打包工具会保留它。通过 `sideEffects` 配置声明哪些文件安全删除。

「代码示例」

「文件： `math.js` 」

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
export const multiply = (a, b) => a * b;
export const divide = (a, b) => a / b;
```

这里我们定义了四个函数 `add` 、 `subtract` 、 `multiply` 、 `divide` ，并通过 `export` 导出。

「文件： `index.js` 」

```
// index.jsimport { add } from './math.js';
console.log(add(2, 3)); // 输出: 5
```

在 `index.js` 中，我们只使用了 `add` 函数，其他三个函数（ `subtract` 、 `multiply` 、 `divide` ）没有被引用。

经过打包后的输出:

如果启用了 Tree Shaking（以 Webpack 或 Vite 为例），构建工具会分析模块的依赖和引用，移除未使用的代码。

```
// 打包后代码const add = (a, b) => a + b;
console.log(add(2, 3)); // 输出: 5
```

未使用的 `subtract` 、 `multiply` 、 `divide` 函数已经被移除。

「模块级别的冗余代码删除，tree-shaking比较得心应手，但是更细颗粒度的删除，还需要依赖其他手段。」

「方法二：压缩插件优化删除」

- JavaScript 的压缩和优化
- 通过工具（如 Terser、esbuild 等），可以删除未被引用的代码片段、简化表达式、内联变量等。举个例子：Webpack 配置（Terser）

```
const TerserPlugin = require('terser-webpack-plugin');
module.exports = {  mode: 'production', // 开启生产模式，默认启用代码压缩
optimization: {    minimize: true,    minimizer: [new TerserPlugin()],  },};
```

```
// 原始代码const unused = () => console.log('I am not used!');const multiply =
(a, b) => a * b;console.log(multiply(2, 3));
//压缩删除后console.log(2 * 3);
```

- 未使用的 `unused` 函数被移除，表达式 `a * b` 被直接计算为结果。
- CSS 的压缩和优化
- 未使用的 CSS 选择器可以通过工具（如 PurgeCSS、UnCSS 等）删除。举个例子：PurgeCSS 配置
- PurgeCSS 检查 HTML、JS 中使用的类名，仅保留相关样式。

```
// 安装 PurgeCSSnpm install @fullhuman/postcss-purgecss --save-dev
```

- `postcss.config.js`

```
1   module.exports = {  plugins: [    require('@fullhuman/postcss-purgecss')({
      content: ['./src/**/*.html', './src/**/*.js'], // 检测的文件路径
    defaultExtractor: content => content.match(/[\w-/:]+(?<!:)/g) || [], // 提取类
    名    }),  ],};
```

- **「示例：」**

```
1
2   /* 原始 CSS 文件 */.button {  color: red;}
3   .unused-class {  background: yellow;}
4   /* 优化后 CSS 文件 */.button {  color: red;}
```

- 未使用的 `.unused-class` 样式被移除。

1.1.2 资源小一点：按需加载

「按需加载（On-demand Loading）」 是前端优化的一种核心手段，指在应用运行过程中，仅在用户需要时动态加载特定资源（如JavaScript 代码、CSS 文件、组件、图片等），而非一次性加载所有资源。它的核心思想是 **「「延迟加载非必要内容」」** ，类似于分批处理任务，通过减少初始加载量来提升性能。

简单来说，传统的 Web 页面在加载时，会一次性加载所有的 JavaScript 和 CSS 文件，即使其中的很多资源在当前页面并未使用。而按需加载则可以 **「分批加载」** 这些资源，减少不必要的加载，提高网页的加载速度和运行效率。具体体现在：

- **「减少首屏加载时间」** ：避免一次性加载所有资源，加快网页首次渲染速度，提高用户体验。
- **「优化性能」** ：减少浏览器的解析和执行负担，避免加载无用的代码，提高运行效率。
- **「节省带宽」** ：只加载需要的资源，减少不必要的网络请求，降低服务器和用户的流量消耗。
- **「提升交互体验」** ：在用户实际需要时才加载资源，比如滚动到某个部分才加载图片，提高页面的响应速度。

「那按需加载的如何实现？按需加载的理论基础是什么？」

按需加载其实是基于模块化的理论基础。要不是代码可以分模块写，那就根据需求加载特定模块就无从谈起。

那我们在前端的项目中，一般是依靠 动态导入（Dynamic Import）语法，也就是 `import(...)` 。这种写法在没有成为规范之前是依赖webpack等打包工具的支持，但是2023年之后也成为了ECMA Script 标准的一部分，百分之九十以上的浏览器都支持。

不使用动态导入的写法： `import APage from '../pages/APage'`

使用动态导入的写法： `const APage = import('../pages/APage')`

使用动态导入语法，那么将会得到一个 `promise` ,加载成功是 `fufiled` ,加载失败是 `rejected` 。而在构建的时候，对应的模块会被拆分成对应的区块，也就是chunk文件，是独立的。在代码运行时会动态添加script标签，触发加载和执行。

「具体如何实施按需加载呢？」

在前端项目中，可以通过多种方式实现按需加载，常见的方法包括：

- 代码层面的动态导入
- 组件级的懒加载
- 图片和静态资源的懒加载
- 路由级的懒加载
- 服务端返回数据的按需加载

我们展开讲讲：

「代码按需加载」

- 通过 **「JavaScript 动态导入（import()）」** ，在代码执行到某个逻辑时再加载对应的模块，而不是在页面加载时就全部加载。
- 主要应用于 **「大型前端项目，避免一次性加载全部 JavaScript 代码」** 。

「示例（使用 Webpack/Vite 实现代码分割）：」

```
// 传统方式（非按需加载）import HeavyComponent from './HeavyComponent';
// 按需加载方式（动态导入）const loadComponent = async () => {  const { default: HeavyComponent } = await import('./HeavyComponent');  return HeavyComponent;};
```

⚡ **「适用场景」**： **「第三方库按需加载、大型组件加载、页面内嵌功能的动态加载」** 。

「组件级的按需加载」

在 **「React、Vue」** 这些前端框架中，可以使用 `lazy` 和 `Suspense` （React）或 `defineAsyncComponent` （Vue）实现组件的按需加载。

「还有第三方组件库（antd)的按需加载（放到后面的vue项目优化中讲）」

「React 组件按需加载示例：」

```
import React, { lazy, Suspense } from 'react';
// 按需加载组件const LazyComponent = lazy(() => import('./LazyComponent'));
const App = () => (  <Suspense fallback={<div>Loading...</div>}>  <LazyComponent />  </Suspense>);
```

⚡ 「适用场景」：「大组件、弹窗、富文本编辑器等复杂组件的按需加载」。

「Vue 组件按需加载示例（Vue 3）：」

```
import { defineAsyncComponent } from 'vue';
export default {  components: {    LazyComponent: defineAsyncComponent(() => import('./LazyComponent.vue')),  },};
```

⚡ 「适用场景」：「Vue 组件按需加载，减少初始 bundle 体积」。

「图片按需加载」

对于图片等静态资源，可以使用「懒加载（Lazy Load）」技术，在用户「滚动到图片可见时才加载」。

「HTML 原生懒加载（现代浏览器支持）：」

```
<img src="placeholder.jpg" data-src="real-image.jpg" loading="lazy" alt="Lazy Image">
```

「JavaScript 手动实现图片懒加载（适用于所有浏览器）：」

```
const images = document.querySelectorAll('img[data-src]');const observer = new IntersectionObserver((entries) => {  entries.forEach(entry => {    if (entry.isIntersecting) {      const img = entry.target;      img.src = img.dataset.src; // 赋值真实图片路径      observer.unobserve(img);    }  });});
images.forEach(img => observer.observe(img));
```

⚡ 「适用场景」：「长列表的图片加载、博客文章、新闻网站等」。

「路由按需加载」

前端路由可以使用「懒加载」，在用户访问某个页面时才加载该页面对应的 JS 代码，而不是一次性加载所有页面。

「Vue Router 懒加载示例：」

```
const routes = [  {    path: '/about',    component: () => import('./views/About.vue'), // 按需加载  },];
```

「React Router 懒加载示例：」

```
1  import { lazy, Suspense } from 'react';import { BrowserRouter as Router,
   Route, Routes } from 'react-router-dom';
2  const Home = lazy(() => import('./pages/Home'));const About = lazy(() =>
   import('./pages/About'));
3  function App() {  return (    <Router>      <Suspense fallback=
   {<div>Loading...</div>}>        <Routes>          <Route path="/" element=
   {<Home />} />          <Route path="/about" element={<About />} />
   </Routes>      </Suspense>    </Router>  );}
```

⚡「适用场景」：「大型单页应用（SPA）优化首屏加载速度」。

「数据按需加载」

在请求接口时，只加载当前需要的数据，减少不必要的请求和数据传输量。

「示例（前端分页请求数据）：」

```
1  const loadMoreData = async (page) => {  const response = await
   fetch(`/api/data?page=${page}`);  const data = await response.json();
   renderData(data);};
```

⚡「适用场景」：「表格分页、无限滚动、搜索结果分页」。

「那落实到实际我们要怎么去执行上面的方案来达到项目整体的按需加载呢？」

- 「分析项目需求」，找出哪些资源可以按需加载（JS 代码、组件、图片、路由、数据等）一般可以通过一些插件来分析打包的产物，例如 `webpack-bundle-analyzer-plugin`，看看哪些产物比较大且又是初始化的时候不需要加载的，可以单独抽出来按需加载。
- 「使用 Webpack/Vite 代码分割」，通过 `import()` 进行动态加载。
- 「在 React/Vue 中使用懒加载组件」，使用 `React.lazy()` 或 `defineAsyncComponent()`。
- 「优化图片加载」，使用 `loading="lazy"` 或 `IntersectionObserver` 进行懒加载。
- 「使用路由懒加载」，按需加载路由页面，提高首屏加载速度。
- 「合理进行数据加载优化」，使用分页请求、按需请求 API，减少不必要的网络请求。

1.1.3 资源小一点：细颗粒度的代码分割

「什么是细颗粒度的代码分割方案？」

代码分割是利用现代前端构建工具的功能，将原本的单一构建文件拆分成多个小文件，从而提高缓存的命中率，进而优化用户体验。

「为什么拆得小了，拆得多了就会提高缓存命中率？」

因为我们常见的缓存策略是：html文件不缓存，每次都去请求最新的html文件。静态资源文件是通过构建工具打了hash值的tag的，只要资源文件发生变化，就会生成新hash，从而命中不了缓存，达到获取新资源的目的。（这个原理不懂可以先问下AI）

那由此可知，拆分的更细了，代码文件之间的影响就更小了。例如模块a和模块b不拆分时打包到一个文件ab-chunk中， 那么a或者b模块变了，都要完整再加载一次ab-chunk资源，但是如果模块a和模块b分开了，就互不影响了，能更最大化的命中缓存。

具体的实践方案网上有很多了，这里不展开。

1.1.4 资源小一点：Gzip压缩

「什么是Gzip压缩？」

Gzip是一种常用的数据压缩格式，它可以通过对HTTP响应内容进行压缩，减小文件的体积，从而加快网页加载速度，提升前端性能。在前端开发中，Gzip压缩主要应用于HTTP响应中传输的文本文件（如HTML、CSS、JavaScript等）。

「说点人话」：gzip就是一个我们常见的压缩资源的一种方式，你只需要在请求头写上：`accept-encoding:gzip` 服务器就能知道要开启压缩，从而压缩资源并返回，浏览器接受到压缩资源进行解压。本质就是牺牲服务器的开销和浏览器的开销来换取资源的最小化，从而提升加载速度。

gzip压缩一般能帮我们压缩到原本资源的70%大小，但也不是所有情况下的压缩效率都有这么高。gzip压缩背后的原理是，在一个资源文件中找重复，并临时替换这些重复，从而就缩小整个文件，所以资源文件中的代码重复率越高，压缩效率也越高。

「那问题又来了，用服务器的压缩时间和浏览器的解压时间来换取资源缩小，真的值得吗？」

是的这个问题也不能说绝对，但是对于绝大部分情况来说，都是值得的，压缩和解压的时间相对于资源缩小而带来的传输速度的提升是微不足道的，除非你的文件超级小1k、2k。

那作为前端开发人员，如何利用Gzip呢?

「webpack Gzip + 服务器 Gzip 的最佳实践」

使用构建工具开启Gzip压缩：

Webpack 中的 Gzip 压缩操作本质上是 **「在构建阶段」** 提前对静态资源（如 JS、CSS、HTML）进行 Gzip 预压缩，并将压缩后的 `.gz` 文件作为构建产物输出。这样，在部署时，服务器无需动态压缩这些文件，而是直接提供预压缩的 `.gz` 文件，减少了服务器的 CPU 负担，提高了响应速度。

在 Webpack 中，我们使用 `compression-webpack-plugin` 这个插件来实现 Gzip 预压缩。这个插件在 Webpack 打包时，针对符合条件的静态文件（通常是 `.js` 、 `.css` 、 `.html` ），使用 Gzip 算法生成 `.gz` 文件，并将其与普通未压缩的文件一起输出到 `dist` 目录。

「Webpack 在打包时执行 `compression-webpack-plugin` 」

- 遍历所有构建生成的文件，筛选出符合压缩条件的文件（如 `.js` 、 `.css` ）。
- 使用 `zlib` 进行 Gzip 压缩，生成 `.gz` 文件（如 `bundle.js.gz` ）。

- 这些 `.gz` 文件被保留在构建产物中，等待部署。

在 Webpack 配置 `compression-webpack-plugin` ，生成 `.gz` 版本的 JS、CSS、HTML 资源：

```
1    const CompressionWebpackPlugin = require('compression-webpack-plugin');
2    module.exports = {    plugins: [      new CompressionWebpackPlugin({
algorithm: 'gzip',        test: /.(js|css|html)$/, // 需要压缩的文件类型
threshold: 10240, // 只有大于 10KB 的文件才进行压缩        minRatio: 0.8, // 只有
压缩比低于 0.8 才会被压缩        deleteOriginalAssets: false, // 是否删除原始文件，
通常保留原始文件，方便回退      })    ]  };
```

「服务器配置」

服务器（如 Nginx 或 Apache）需要配置规则，当请求的文件存在 `.gz` 版本时，直接返回压缩后的文件，并添加 `Content-Encoding: gzip` 响应头。 **「服务器优先返回 Webpack 预压缩的 `.gz` 文件」**

「Nginx 配置」

```
1    server {    gzip on;    gzip_disable "msie6"; # 禁止 IE6 使用 gzip
gzip_types text/plain text/css application/json application/javascript
text/xml application/xml;    gzip_vary on;            # 优先返回 gzip 版本的文
件    location / {      try_files $uri $uri.gz $uri/ =404;      add_header
Content-Encoding gzip;    }  }
```

Webpack Gzip 预压缩和服务器 Gzip 各有优势， **「它们并不是互相替代的，而是可以互相配合，以优化性能」** ：

- **「Webpack 预压缩静态资源」** ，减轻服务器 CPU 负担，提高文件加载速度。
- **「服务器 Gzip 处理动态内容」** ，如 API 响应的数据压缩，确保整体性能优化。
- **「正确配置服务器」** ，优先提供 `.gz` 版本的文件，保证浏览器能够正确加载 Gzip 资源。

这样可以做到 **「资源加载快，服务器压力小，整体性能最佳」**

1.1.5 资源小一点：图片体积优化

图片的优化对前端性能至关重要，原因有几个方面。首先，图片是现代网页中最常见且最占用带宽的资源之一。在大多数网站中，图片的体积通常占据了整个页面资源的很大一部分，尤其是对于内容丰富的页面（如电商网站、博客、新闻网站等）而言。如果不对图片进行优化，加载这些图片会大大拖慢页面的渲染速度，影响用户体验，甚至增加CDN或服务器的负担。

「为什么图片优化对前端性能优化至关重要？」

1. **「大大降低页面加载时间」**：图片通常是网页上最大且最重的资源之一。未经优化的图片会导致网络请求时间过长，尤其是在带宽有限或移动网络环境下，图片加载可能会成为瓶颈。如果使用压缩或更高效的格式（如WebP、AVIF等），可以显著减少图片文件的大小，进而减少页面加载时间。
2. **「提高缓存命中率」**：对图片进行优化后，它们的体积会变小，缓存策略能够更有效地工作。较小的图片资源可以减少浏览器缓存中占用的空间，提高缓存的命中率，从而加快后续访问同一页面时的加载速度。
3. **「降低带宽消耗」**：优化图片可以减少传输的数据量，进而减少带宽消耗。对于带宽有限的用户，图片的快速加载和小文件体积是非常重要的，尤其是在移动设备或慢速网络条件下。
4. **「提升用户体验」**：网页加载速度直接影响到用户体验。页面加载时间过长可能会导致用户流失。图片优化有助于提升页面的加载速度和响应时间，从而提升用户的浏览体验。

常见的图片类型主要可以分为以下几类：

1. **「JPG（JPEG）」**：
   - **「特点」**：一种有损压缩的图片格式，广泛应用于照片类图片。体积较小，支持多种色彩（适用于复杂色彩的图片，如风景照、人物照等）。
   - **「优化方式」**：通过调整压缩率来减小图片体积。对于需要处理大量照片的场景，使用适当的压缩率可以减小体积，但不牺牲过多的视觉质量。
2. **「PNG」**：
   - **「特点」**：一种无损压缩的格式，支持透明通道，适用于图标、网页元素、截图等。
   - **「优化方式」**：对于没有透明通道的PNG图片，建议使用 `pngcrush` 、 `optipng` 等工具进行优化，减小无损压缩后的体积。对于透明图片，使用WebP或者AVIF等格式可以替代PNG，进一步降低体积。
3. **「GIF」**：
   - **「特点」**：常用于制作动画图像。支持多帧动画，但颜色深度有限（最多256种颜色）。文件体积较大。
   - **「优化方式」**：尽量避免使用GIF动画，尤其是大尺寸的GIF，可以考虑使用WebP或APNG（Animated PNG）作为替代，提供更好的质量和更小的体积。
4. **「SVG」**：
   - **「特点」**：矢量图格式，适用于简单的图形、图标和标志。可缩放，且不失真。适用于动态、交互式图形。
   - **「优化方式」**：对SVG文件进行清理，去掉多余的元数据、注释和空格，进一步减小文件体积。可以通过在线工具（如 `SVGO` ）来优化SVG文件。
5. **「WebP」**：
   - **「特点」**：支持有损压缩和无损压缩，适用于Web。比JPG、PNG等格式有更高的压缩率，体积更小，支持透明通道。

- 「优化方式」：直接将图片转换为WebP格式，利用其高效压缩算法来减少文件体积，尤其适用于大批量图片的优化。

6. 「AVIF」：
   - 「特点」：一种较新的图片格式，支持高效的有损和无损压缩，支持透明通道，图像质量较高，体积较小。
   - 「优化方式」：将图片转换为AVIF格式，能够提供更小的体积和更好的图像质量，尤其适用于图像内容较复杂的场景。

格式	简介与特性	体积示例（基于图片CDN计算）	发明年份	浏览器兼容性
JPG	- 最常见且应用最广泛的图片格式 - 体积适中，通常小于PNG、GIF等格式	158 KB（100%）	1992	几乎所有浏览器支持
PNG	- 支持透明通道，可以做部分透明图片 - 体积较大	819 KB（518%）	1996	几乎所有浏览器支持
GIF	- 支持动态效果的图片 - 体积较大	423 KB（267%）	1989	几乎所有浏览器支持
SVG	- 矢量图，不会因缩放失真 - 本质是标记语言，浏览器可解析渲染 - 体积视内容而定	--	2001	Chrome 4（2010年发布）及以上版本支持 参考资料：caniuse.com/svg
WebP	- 支持动态图片 - 压缩效率高，支持有损和无损压缩 - 专为Web平台优化 - 体积较小	136 KB（86%）	2010	Chrome 32（2014年发布）及以上版本支持 参考资料：caniuse.com/webp
AVIF	- 支持动态图片 - 压缩率高 - 体积较小	96 KB（60%）	2019	Chrome 85（2020年发布）及以上版本支持 参考资料：caniuse.com/avif

通过上表可以看出，图片格式大致可分为两类：

- 「传统图片格式」：如JPG、PNG、GIF、SVG等，这些格式出现已有二十多年。

- 「现代图片格式」：如WebP、AVIF等，这些格式是近十年内新出现的。

从功能和性能上来看：

- 「体积」：传统格式的图片文件普遍较大。与JPG格式相比，WebP格式通常可以将文件体积减少约10%，而AVIF格式甚至能减少超过40%的体积。
- 「特性」：现代格式支持更多特性，如动态图片和无损压缩等，而传统格式的特性相对单一。
- 「浏览器兼容性」：现代格式的浏览器兼容性稍逊一筹，支持它们的浏览器数量相对较少。

如果能够使用WebP、AVIF等现代图片格式，可以显著解决前端应用中图片文件体积大、加载慢、CDN开销高等问题，从而提升性能。

然而，由于浏览器兼容性问题，我们不敢完全依赖这些现代格式，无法大规模应用。

「如何进行图片性能优化？」

- 「选择合适的图片格式」：
- 根据图片的用途选择合适的格式。例如，照片类图片优先使用JPG或WebP，图标和UI元素则使用SVG或WebP。对于需要透明度的图片，优先使用WebP或AVIF。
- 「使用现代格式」：
- 尽可能使用WebP和AVIF等现代格式，这些格式提供更高的压缩比和更小的体积，能够显著提升页面加载速度，尤其是在图片数量较多的页面上。结合 `<picture>` 元素去做，例如：
  -
  -
  -
  -
  -

```
1       <picture>       <source type="image/avif"
    srcset="https://cdn.com/image.avif" />      <source type="image/webp"
    srcset="https://cdn.com/image.webp" />      <img
    src="https://cdn.com/image.jpg" alt="Example" />    </picture>
```

- 就是从上到下哪个兼容用哪个。
- 「调整图片质量与尺寸」：
- 在不显著影响视觉效果的前提下，降低图片的质量和尺寸。例如，通过调整JPG的压缩率，或者使用更高效的PNG优化工具，如 `pngcrush` 或 `optipng` 。
- 使用合适的图片尺寸：确保图片的尺寸与实际显示需求相匹配，不要上传过大的图片。例如，在响应式设计中，根据不同设备分辨率和屏幕尺寸加载不同大小的图片。

- **「图片懒加载」**：
- 这个前面有聊过，图片懒加载是延迟加载图片的技术，只有当图片即将进入视口时才会加载，从而减少初始页面加载的资源消耗，提高页面响应速度。
- **「使用CDN」**：
- 将图片托管到CDN（内容分发网络）上，使得图片能够从离用户最近的服务器加载，减少网络延迟，提高加载速度。各大云服务供应商都提供了图片CDN服务，除了基本的资源存储功能外，还附加了多种强大功能。
- **「自动化优化工具」**：
- 使用构建工具和图片处理工具自动化优化图片。例如，使用 `image-webpack-loader` 、`sharp` 等工具，在Webpack构建过程中自动压缩和转换图片格式。
- **「图像精灵」**：
- 将多个小图片（如图标）合并成一张大的图片，使用CSS定位来显示不同的部分，减少HTTP请求次数（减少资源加载时间）。

图片优化对于前端性能的提升具有非常重要的意义，合理选择图片格式、进行图片压缩、尺寸调整、懒加载和使用CDN等技术，都能显著提高网站加载速度，改善用户体验。随着Web的不断发展，现代图片格式（如WebP、AVIF）为我们提供了更高效的图片压缩方式，而根据实际需求选择合适的图片格式和优化手段，能使得网站在加载时事半功倍。

1.2 少请求一点

1.2.1 少请求一点：缓存

我们前面说过，性能优化无非就是三个词：**「缓存」**、**「压缩」**、**「懒加载」**。这一节我们来讲如何通过 **「缓存」** 来优化性能。

在前端性能优化中，**「缓存」** 是一个非常重要的手段，能够显著提高网页的加载速度，减少服务器请求，减轻网络压力，从而提升用户体验。通过合理使用缓存，我们可以在不同场景下存储数据和资源，避免重复加载和计算，提升响应速度。

「什么是缓存？」

缓存是指将某些数据存储在一个临时的存储介质（如内存、硬盘或浏览器等）中，以便在以后需要时能够更快速地获取。缓存的目的是避免每次请求都从头开始计算或加载，而是直接从缓存中获取数据或资源，从而提升效率和减少不必要的延迟。

「缓存的类型有哪些？」

缓存可以分为多种类型，每种缓存都有不同的应用场景和作用。常见的缓存类型包括：

1. **「浏览器缓存（Client-side Cache）」**

2. 浏览器缓存是在用户浏览器中存储资源（如图片、CSS、JS文件等），以便在下一次访问同一页面时无需重新下载这些资源。主要通过HTTP头部控制，如 `Cache-Control` 、 `ETag` 、 `Last-Modified` 等。
3. **「DNS缓存」**
4. DNS缓存是指在本地设备（如浏览器、操作系统、路由器等）中缓存DNS解析结果的机制。当你访问一个网站时，浏览器需要通过DNS（域名系统）将域名（如 `www.example.com` ）转换为对应的IP地址。这个过程称为DNS解析。
5. 在首次访问某个域名时，DNS解析器会向域名的DNS服务器发起请求来获取域名的IP地址。为了避免每次都需要重新解析相同的域名，DNS结果会被缓存一段时间，这段时间被称为 **「TTL」**（Time To Live）。TTL过期之前，设备会直接使用缓存的IP地址，而不必再次进行DNS解析。
6. **「HTTP缓存（Server-side Cache）」**
7. HTTP缓存是指在服务器与客户端之间通过HTTP协议进行的缓存处理。服务器会将请求的数据缓存下来，若下次相同请求到来，直接返回缓存的内容，而不再进行计算或查询。常见的缓存策略有：**「客户端缓存」** 、 **「代理缓存」** （如CDN缓存）、 **「服务器缓存」** 等。
8. **「CDN缓存（Content Delivery Network Cache）」**
9. CDN缓存是通过将静态资源（如图片、JS、CSS等）分发到全球各地的CDN节点，减少用户请求的响应时间。CDN缓存提高了静态资源加载的速度，并减轻了源服务器的压力。
10. **「内存缓存（In-memory Cache）」**
11. 内存缓存是将常用数据存储在内存中，减少磁盘I/O操作，提高访问速度。常见的内存缓存技术有： **「Redis」** 、 **「Memcached」** 。
12. **「本地存储（Local Storage / Session Storage）」**
13. LocalStorage：浏览器提供的一种持久化存储方式，数据不会过期，适合存储不频繁变化的数据，如用户信息、设置等。
14. SessionStorage：与LocalStorage类似，但数据仅在当前会话期间有效。适用于存储会话级别的临时数据。
15. **「Service Worker缓存」**
16. Service Worker是一个可以在后台线程运行的JavaScript，它能够拦截网络请求并将响应存储到缓存中。Service Worker缓存主要用于支持离线功能，使得应用能够在没有网络的情况下继续运行。

「如何通过缓存来提升性能？」

缓存优化可以有效减少重复加载和计算，提升页面的加载速度和响应能力。我们接着上面的分类挨个讲：

「1. 浏览器缓存」

在浏览器缓存机制中，**「强缓存」** 和 **「协商缓存」** 是两个核心概念，它们帮助浏览器决定资源是否需要重新请求服务器，进而优化页面加载速度。两者有不同的工作原理和应用场景。下面，我将详细解释这两种缓存的内容，以及它们是如何变化的。

- **「强缓存（Cache-Control、Expires）」**
- **「强缓存」** 是一种最常见的缓存方式，它会直接判断资源是否在缓存有效期内，如果有效，就会直接从缓存中加载资源，而不需要与服务器进行任何交互。
- 强缓存通过设置 `Cache-Control` 或 `Expires` 来控制资源的缓存策略。它的关键点是：在缓存的有效期内，浏览器会直接使用缓存中的资源，而不会向服务器发起任何请求。
- **「Cache-Control」**：这是一个现代的、灵活的缓存控制头部，可以用来指定资源的缓存策略。例如：
- **「Expires」**：这是一个过时的缓存头部，用来指定资源的过期时间。比如 `Expires: Wed, 21 Oct 2025 07:28:00 GMT` 。不过， `Expires` 已经被 `Cache-Control` 替代，`Cache-Control` 提供了更多的控制选项。
- **「强缓存的变化过程」**
- **「举例」**：
  - `Cache-Control: max-age=3600` ： 资源缓存1小时。在这1小时内，浏览器不会发起任何请求，直接使用缓存。如果超过1小时，缓存失效，浏览器会重新请求资源。
  - `Cache-Control: no-cache` ： 表示每次都需要与服务器确认缓存是否有效，即使资源存在缓存，浏览器也会发送请求进行验证。
  - **「首次请求」**： 浏览器请求资源时，服务器会返回带有缓存控制头部的响应，浏览器根据 `Cache-Control` 或 `Expires` 判断是否缓存资源。
  - **「有效缓存」**： 如果缓存仍然有效，浏览器就会直接从缓存中加载资源，而不会向服务器发起请求。
  - **「过期缓存」**： 如果缓存已经过期，浏览器会再次向服务器发送请求，获取最新的资源。
  - `max-age`：指定资源在缓存中的最大存活时间（单位为秒）。比如， `Cache-Control: max-age=3600` 表示资源会被缓存1小时。
  - `public`：表示资源可以被任何缓存服务器缓存（即使是代理服务器也可以缓存）。
  - `private`：表示资源只能被用户浏览器缓存，不能被代理服务器缓存。
  - `no-cache` 和 `no-store`：强制不缓存资源，其中 `no-store` 是最严格的缓存控制，表示不允许缓存。
- **「协商缓存（ETag、Last-Modified）」**
- **「协商缓存」** 是一种浏览器与服务器之间的缓存机制，它依赖于浏览器和服务器的通信来确定资源是否有变化。如果资源没有变化，服务器会返回 304（Not Modified）响应，告诉浏览器继续使用缓存中的资源。

- 协商缓存的核心是 `ETag` 和 `Last-Modified` 这两个HTTP头部。它们用于帮助浏览器和服务器之间判断资源是否发生变化，具体步骤如下：
- 当浏览器使用缓存的资源时，会在请求头中带上这些标识符，服务器会根据这些标识符判断资源是否有变化：
- **「协商缓存的变化过程」**：
- **「首次请求」**：服务器返回资源并附带 `ETag` 或 `Last-Modified` ，浏览器将其存储起来。
- **「后续请求」**：浏览器向服务器发送带有 `If-None-Match` 或 `If-Modified-Since` 的请求头，询问服务器资源是否有变化。
- **「服务器响应」**：如果资源没有变化，服务器返回 304 状态码，浏览器继续使用本地缓存。如果资源变化，服务器返回新的资源和新的 `ETag` 或 `Last-Modified` 。
- **「举例」**：
- **「ETag」**：
- **「Last-Modified」**：
  - 服务器返回：`Last-Modified: Tue, 20 Apr 2021 12:00:00 GMT`
  - 浏览器请求时发送：`If-Modified-Since: Tue, 20 Apr 2021 12:00:00 GMT`
  - 如果资源未修改，服务器返回 304 状态。
  - 服务器返回：`ETag: "12345"`
  - 浏览器请求时发送：`If-None-Match: "12345"`
  - 如果资源未修改，服务器返回 304 状态，告诉浏览器继续使用缓存。
  - **「If-None-Match」**：浏览器在请求时带上上次请求的 `ETag` 值，服务器比较 `ETag` 是否相同，如果相同则返回 304（Not Modified）。
  - **「If-Modified-Since」**：浏览器会发送 `Last-Modified` 值，服务器检查文件自上次修改以来是否被更新，如果未更新，则返回 304。
  - **「ETag」**：服务器会在响应头中返回一个唯一标识符（如哈希值），该标识符代表资源的内容。如果资源内容没有发生变化， `ETag` 就不会改变。
  - **「Last-Modified」**：服务器会在响应头中返回资源最后修改的时间。如果资源没有被修改，服务器返回的时间就不会变化。

「强缓存」：通过设置缓存过期时间（ `Cache-Control` 或 `Expires` ）来控制资源的存储期，缓存有效期内直接使用缓存，不与服务器交互。适用于静态且更新不频繁的资源。

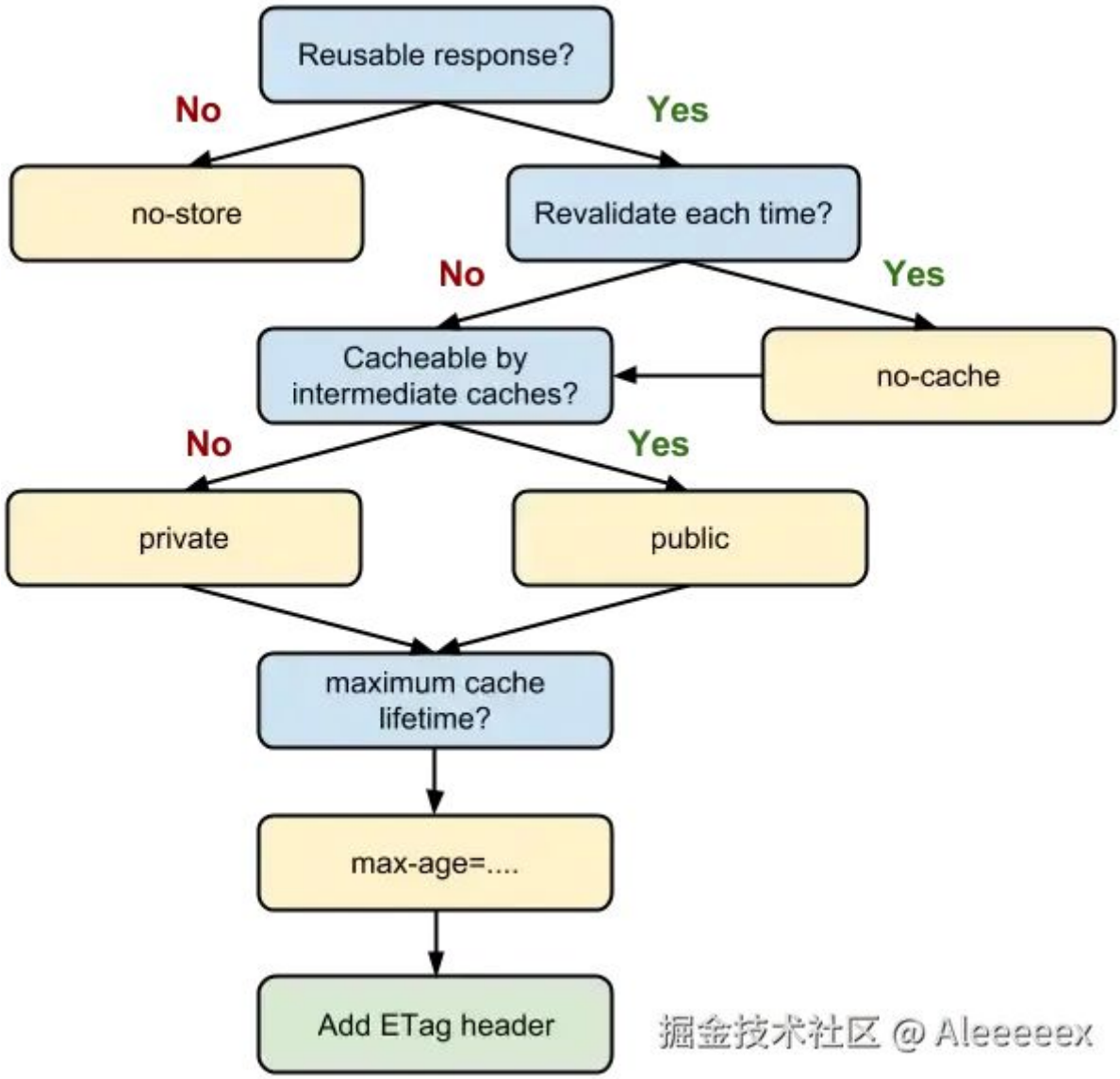
「协商缓存」：依赖 `ETag` 和 `Last-Modified` ，通过与服务器的通信来验证缓存是否有效。适用于需要频繁更新且服务器内容不可预知的资源。

这两种缓存方式可以组合使用，先使用强缓存，如果强缓存失效，再通过协商缓存向服务器验证资源的有效性，从而提供最优的缓存策略。

特性	强缓存	协商缓存
工作方式	完全依赖缓存头，直接使用缓存，不与服务器交互	每次请求都需要与服务器交互，验证缓存是否有效
主要控制头	`Cache-Control`，`Expires`	`ETag`，`Last-Modified`
请求发送	如果缓存有效，浏览器不发送请求	浏览器每次请求都会带上 `If-None-Match` 或 `If-Modified-Since`
响应状态	直接使用缓存，若过期才会重新请求	服务器根据缓存标识符判断是否修改，未修改返回 304 状态
适用场景	适用于不需要频繁更新的静态资源，如图片、CSS、JS	适用于需要频繁更新并且更新间隔不可预知的资源

「举例」：对于图片、JS、CSS等静态资源，可以设置较长的缓存过期时间，而对于频繁更新的内容（如动态HTML页面），可以设置较短的缓存时间或不缓存。

谷歌给了一张图，更好的说明了应该怎么去指定资源缓存的策略



总结一下上面的图。首先，如果资源内容完全不可复用，那就直接把 `Cache-Control` 设置为 `no-store` ，拒绝任何形式的缓存。否则，问自己一个问题：每次都要去服务器确认缓存是否有效吗？如果是，那就设为 `no-cache` 。接下来，思考下这个资源是否可以被代理服务器缓存，基于这个决定，将其设置为 `private` 还是 `public` 。然后，进一步考虑资源的过期时间，合理配置 `max-age` 和 `s-maxage` 。最后，别忘了配置协商缓存所需要的 `ETag` 、 `Last-Modified` 等参数。

「2. DNS缓存」

「DNS缓存」 是指在本地设备（如浏览器、操作系统、路由器等）中缓存DNS解析结果的机制。当你访问一个网站时，浏览器需要通过DNS（域名系统）将域名（如 `www.example.com` ）转换为对应的IP地址。这个过程称为DNS解析。

在首次访问某个域名时，DNS解析器会向域名的DNS服务器发起请求来获取域名的IP地址。为了避免每次都需要重新解析相同的域名，DNS结果会被缓存一段时间，这段时间被称为 **「TTL」** （Time To Live）。TTL过期之前，设备会直接使用缓存的IP地址，而不必再次进行DNS解析。

DNS缓存的工作原理：

1. 「首次请求」：浏览器向DNS服务器发起查询请求，服务器返回域名对应的IP地址，并设置TTL。
2. 「缓存存储」：浏览器将返回的IP地址存储在DNS缓存中，并在TTL有效期内使用该IP地址。
3. 「过期查询」：TTL到期后，浏览器会再次发起DNS查询，获取最新的IP地址。

DNS缓存不仅存在于浏览器中，还可以在操作系统和网络设备（如路由器）中缓存DNS结果。

「那DNS缓存如何提升性能？」

DNS解析的过程是必须的，但每次请求都进行DNS解析会增加额外的时间延迟。利用DNS缓存可以避免重复的解析请求，从而减少加载时间，提升网站的访问速度。

具体来说，利用DNS缓存可以带来以下优化：

「减少DNS查询延迟」：当域名的IP地址已经在缓存中，浏览器可以直接读取缓存，避免再次向DNS服务器发起请求。如果缓存命中，DNS解析就可以在几毫秒内完成，而如果不命中，则需要通过网络查询DNS服务器，耗时会较长。

「减少网络请求」：通过避免重复的DNS查询，减少了客户端与DNS服务器之间的通信，减少了网络延迟。特别是在用户访问同一网站的多个页面时，DNS缓存可以显著降低每次页面加载的时间。

「加速多域名资源加载」：如果一个页面需要加载来自多个不同域名的资源（例如，CDN上的图片、JavaScript文件、API请求等），DNS缓存可以避免重复解析这些域名，从而加速资源加载。

「作为前端开发人员，虽然我们不能直接控制DNS解析过程，但可以采取一些措施，利用DNS缓存来优化性能：」

「使用持久的域名和合理的TTL设置」

- 选择稳定的域名：选择长期存在的、稳定的域名，避免频繁更换域名。因为每次更换域名都会导致DNS缓存失效。
- 合理配置TTL值：对于CDN、静态资源等不频繁变化的资源，可以配置较长的TTL（如几小时或几天），这样浏览器在一段时间内会缓存DNS解析结果，避免频繁解析。对于动态内容，可以设置较短的TTL，确保DNS解析结果保持最新。

「减少跨域DNS查询」

- 避免过多的第三方域名：尽量减少页面中跨多个域名的请求（比如加载不同域名下的图片、字体、广告等）。每个新的域名都需要进行一次DNS解析。如果这些域名都缓存得不好，会增加DNS查询的次数和延迟。
- 合并请求：尽量将多个静态资源合并到一个域名下，这样浏览器只需要解析一次DNS，减少额外的延迟。

「利用DNS预解析（DNS Prefetch）」

「DNS预解析」是HTML中的一个技术，可以让浏览器提前解析将要请求的域名。通过 `<link rel="dns-prefetch">` 标签，浏览器可以提前解析特定域名，从而加快后续请求的响应速度。

举例：如果你知道网站会加载外部CDN资源，可以在页面中提前指定DNS预解析。

-

```
<link rel="dns-prefetch" href="https://cdn.example.com">
```

这样，在浏览器加载页面时，会提前解析 `cdn.example.com` 的IP地址，减少后续对该域名的DNS解析时间。

「使用HTTP/2（多路复用）」

「HTTP/2协议」 允许多个请求共享一个TCP连接并进行多路复用，减少因多个域名引发的DNS查询延迟。通过使用HTTP/2，可以提高多个域名下资源加载的并行性，并且通过较少的TCP连接减少DNS解析的次数。

例如，当你使用多个域名加载资源时，浏览器必须分别为每个域名进行DNS解析。HTTP/2可以优化这一点，降低域名解析和建立连接的时间。

「合理使用CDN」

「CDN缓存」 ：使用CDN可以将静态资源分发到离用户更近的服务器，并且CDN会缓存DNS解析结果。这样，当用户请求相同资源时，不仅可以加速资源加载，DNS解析也会被缓存，减少了DNS查询的次数和延迟。

「设置和管理子域名」

- 使用子域名：如果有多个独立的子系统或服务，可以通过子域名来拆分资源，缓存不同域名的DNS结果。比如，你可以为静态资源、API服务、图片等分别设置不同的子域名，如 `assets.example.com` 、 `api.example.com` 、 `images.example.com` 等。
- 避免频繁修改DNS记录：如果频繁更换子域名的DNS记录会导致DNS缓存被清空，从而增加解析延迟。因此，要谨慎管理DNS记录和TTL设置。

通过合理利用 **「DNS缓存」** ，我们可以有效减少DNS解析时间，提高页面加载速度，提升用户体验。具体优化方法包括：

- **「选择长期稳定的域名」** ，并合理设置TTL值；
- **「减少跨域DNS查询」** ，通过合并资源减少DNS解析次数；
- **「利用DNS预解析」** 加速外部资源加载；
- **「使用HTTP/2协议」** 优化多个资源加载；
- **「使用CDN」** 加速静态资源加载，同时缓存DNS结果。

通过这些方式，前端开发人员能够有效利用DNS缓存提升网站的性能，减少延迟，提高用户体验。

「3. CDN缓存」

「利用CDN缓存加速资源加载」
CDN通过将资源缓存到离用户更近的节点，使得静态资源可以从最近的服务器获取，从而加快加载速度并减轻源服务器的负担。使用CDN缓存可以显著提高网站的响应速度，尤其是对于跨地域的访问。

「举例」：对于全球访问的网站，部署CDN缓存，可以确保资源的快速加载，避免重复请求源服务器。

「4. Service Worker缓存」

「实现离线缓存」

使用Service Worker可以缓存页面的资源，甚至实现离线访问功能。当用户没有网络连接时，Service Worker可以从缓存中获取资源，使得应用仍然可以正常显示。

「举例」：通过Service Worker缓存首页、图标、样式文件等资源，确保即使在没有网络的情况下，用户也能访问应用的一部分内容。

示例代码：

```
1    self.addEventListener('install', event => {   event.waitUntil(
    caches.open('my-cache').then(cache => {       return cache.addAll([
```

```
    '/index.html',         '/style.css',         '/app.js',         '/logo.png'
      ]);     })   ); });
2   self.addEventListener('fetch', event => {   event.respondWith(
    caches.match(event.request).then(response => {       return response ||
    fetch(event.request);     })   ); });
```

「5. 本地存储缓存」

「利用LocalStorage和SessionStorage缓存数据」

使用浏览器的 `localStorage` 或 `sessionStorage` 来存储页面中的非敏感数据，如用户设置、浏览历史等，可以避免每次加载页面时重新请求相同的数据，减少了请求时间。

「举例」：存储用户的主题色、登录状态、购物车数据等，以便在页面刷新后恢复。

-
-
-
-
-

```
1   // 保存数据 localStorage.setItem('theme', 'dark');
2   // 获取数据 const theme = localStorage.getItem('theme');
```

「5. 内存缓存（In-memory Cache）」

- **「使用内存缓存提高访问速度」**
  内存缓存（如 `Redis` 、 `Memcached` ）存储的是内存中的数据，访问速度极快。通常用于缓存数据库查询结果、API响应等高频次请求的数据，避免重复计算或重复查询数据库。
- **「举例」**：缓存API请求的响应，避免每次都查询数据库。例如，一个产品详情页的API可以缓存该页面的响应结果，在一定时间内返回缓存数据，而不是每次都访问数据库。

「7. 利用版本控制和哈希」

「版本控制资源和缓存失效」

通过为静态资源文件（如JS、CSS、图片等）添加版本号或哈希值，可以确保文件更新时，浏览器能自动检测并加载新版本资源，而不被缓存旧版本。

「举例」：当资源文件更改时，通过修改文件名（如 `app.123456.js` ）来确保浏览器加载最新版本的文件，而不是使用缓存的旧文件。

总之缓存是前端性能优化中的核心技术之一，能够显著减少网络请求，提升加载速度，减少带宽消耗，最终优化用户体验。通过合理使用浏览器缓存、CDN缓存、内存缓存、本地存储、Service Worker缓存等技术，我们可以大幅提高网页的性能，确保网站在不同场景下的快速响应。

1.3 请求快一点

1.3.1 请求快一点：CDN的优化

我们前面的缓存一节中，已经简单介绍过cdn缓存，这一节我们来聊一聊，如何对cdn进行优化从而使得请求得快一点。

「什么是cdn?」

简单说就是把一些静态资源放到不同地理位置的服务器中，用户访问的时候，就访问最近的那个服务器的资源，速度就会快很多。

「那什么是静态资源？」

就是如js、css、html 、图片等不需要服务器即时计算的资源就是计算资源。举个例子，如果你请求一个接口，该接口返回最新的数据如：位置信息，实时的画面等，这种就不是静态资源。

「那我们应该实施什么行为来优化cdn呢？」

- **「根据用户的地理位置分布选择cdn的地理位置」**
- 大多数的cdn云服务都有负载服务器遍布大部分地区，但是需要手动切换，所以在选择cdn服务的时候，要看用户的地理位置分布，基于用户来选择cdn服务器。
- **「cdn服务器域名和业务域名不同的好与不好」**
- cdn服务器的域名和业务域名不一样会导致跨域的问题，对于开发人员来说，解决跨域问题是一件比较繁琐的事情，但是对用户没有直接的不良体验。但由此产生的好处也是大大的。相同的域名下，请求静态资源都会携带cookie,不同的域名反而不需要，否则请求每个静态资源都要携带cookie这对带宽也是一种浪费。
- **「开启压缩算法」**
- cdn服务器也是一个标准服务器，那我们自然也要开启压缩，例如 `gzip` `br` 等压缩功能，能节省资源体积，提升加载速率。
- **「使用最新版本的http协议」**
- 这个在下一节会详细讲，不赘述，字面意思。

1.3.2 请求快一点：http2

HTTP协议经历了多次迭代，每个版本都在前一个版本的基础上进行了优化，解决了性能瓶颈、提高了用户体验，并适应了互联网不断发展的需求。下面是HTTP各个版本的详细发展过程和其特性：

「1. HTTP/0.9 (1991年) — 初代协议」

- **「核心特性」**：
  - **「请求方式」**：只有 **「GET」** 方法，用于请求文本资源（HTML）。
  - **「响应格式」**：只支持 **「纯文本」**（没有MIME类型）。
  - **「没有请求头」**：没有请求头和响应头，所有的请求和响应都是简单的文本内容。

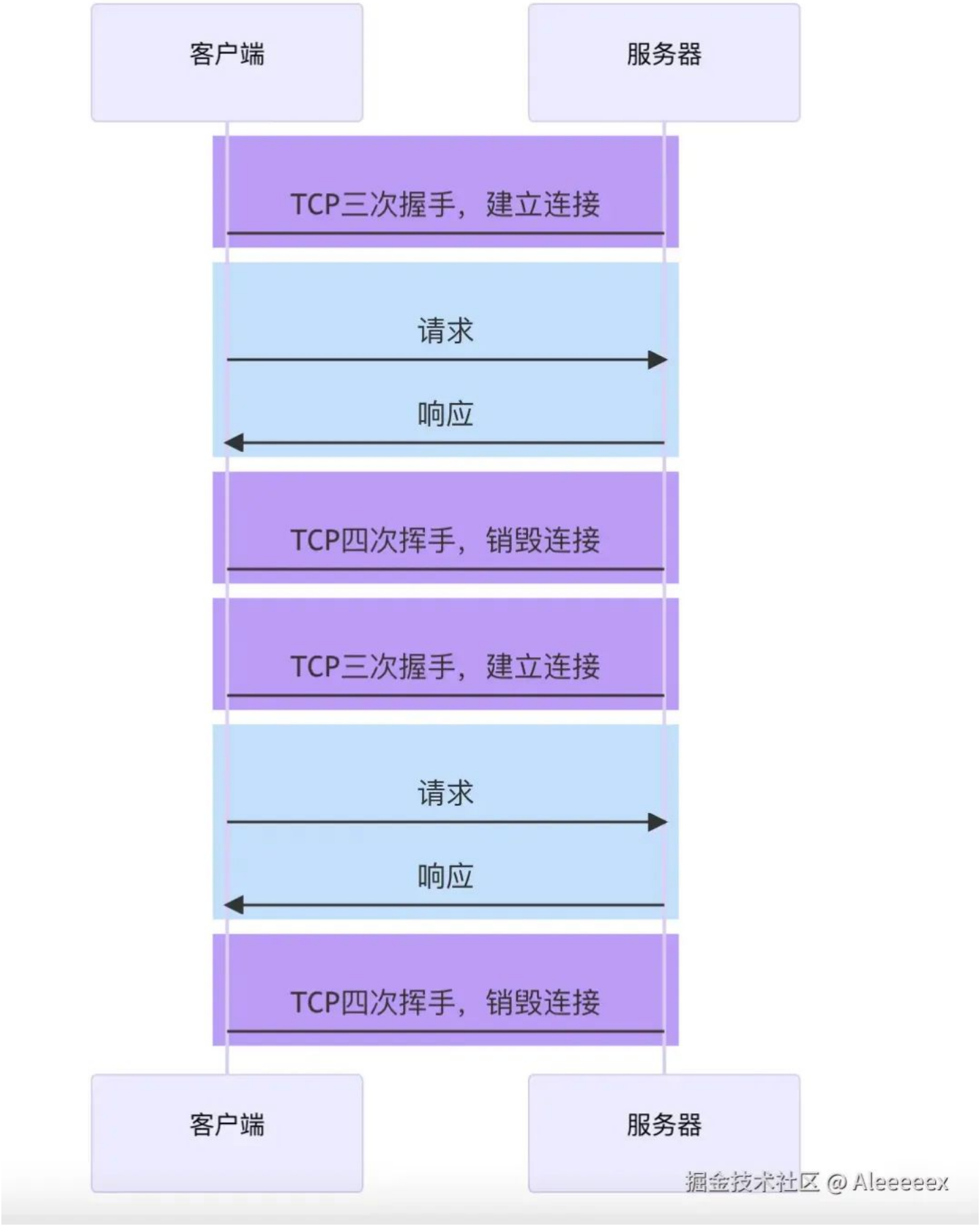
    - 「简单连接」：每个请求和响应结束后，都会关闭连接。
- 「适用场景」：用于最初的网页浏览，功能非常简单，只适用于文本文件。

「2. HTTP/1.0 (1996年) — 标准化初步完善」

- 「核心特性」：
    - 「请求方式」：除了GET，还支持 「POST」 方法，可以发送数据。
    - 「MIME类型」：支持传输多种格式的数据（如HTML、图片、音频、视频等），并在响应头中标明数据类型（如 `Content-Type` ）。
    - 「请求头和响应头」：引入了请求头和响应头，可以携带更多的信息，如用户代理信息、服务器信息、缓存控制等。
    - 「短连接」：每次请求都需要建立新的TCP连接，完成数据传输后关闭连接。
- 「问题」：尽管引入了更丰富的功能，但由于每个请求都需要单独建立TCP连接，造成了大量的开销。

「无法复用连接是一个1.0版本比较大的问题」

HTTP1.0为每个请求单独新开一个TCP连接



由于每个请求都是独立的连接，因此会带来下面的问题：

1. 连接的建立和销毁都会占用服务器和客户端的资源，造成内存资源的浪费

2. 连接的建立和销毁都会消耗时间，造成响应时间的浪费
3. 无法充分利用带宽，造成带宽资源的浪费

>
>
> TCP协议的特点是「慢启动」，即一开始传输的数据量少，一段时间之后达到传输的峰值。而上面这种做法，会导致大量的请求在TCP达到传输峰值前就被销毁了
>
>

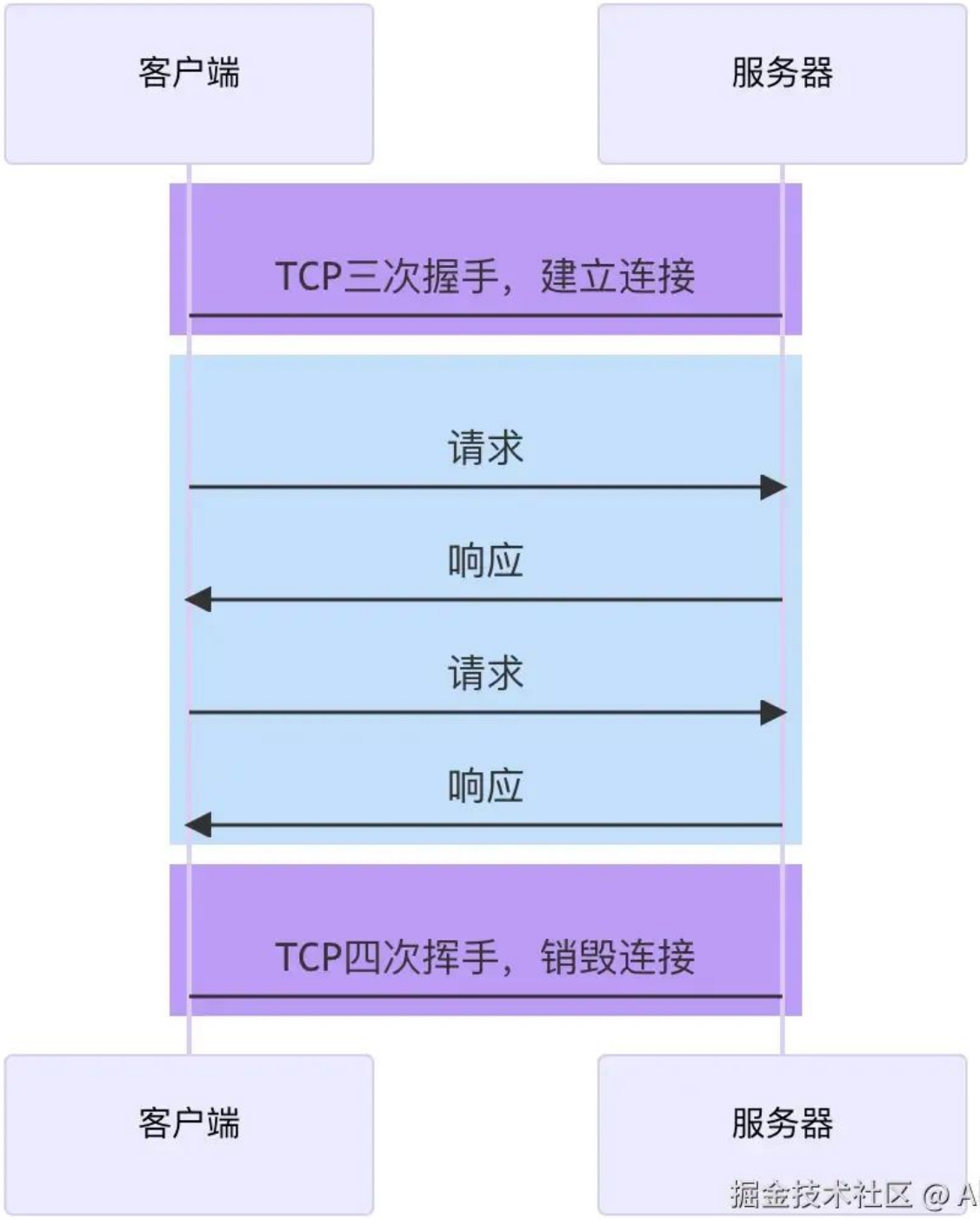
为了解决1.0的问题于是1.1版本出现了

「3. HTTP/1.1 (1997年) — 大规模应用」

HTTP/1.1是最广泛使用的版本，进行了大量改进：

- **「核心特性」**：
  - **「持久连接（Keep-Alive）」**：默认开启长连接，可以复用TCP连接进行多个请求和响应，避免了频繁建立和关闭TCP连接的开销。
  - **「管道化（Pipelining）」**：客户端可以在等待响应的同时，发送多个请求，服务器按顺序响应这些请求（尽管实际上，很多浏览器直到较晚才支持这一功能）。
  - **「分块传输（Chunked Transfer Encoding）」**：允许不确定文件大小的数据分块传输，支持实时传输内容（例如流媒体）。
  - **「缓存控制」**：新增了更强大的缓存机制，支持如 `Cache-Control`、`Etag`、`Last-Modified` 等头部，有效提高了资源的复用性。
  - **「虚拟主机」**：HTTP/1.1支持通过 **「Host」** 头部区分不同的虚拟主机，实现了同一IP地址下多个网站的托管。
  - **「更多的请求方法」**：除了GET和POST，还支持了如PUT、DELETE等HTTP方法，丰富了资源的操作方式。
- **「问题」**：
  - **「队头阻塞（Head-of-Line Blocking）」**：尽管支持管道化，但多个请求仍然会在同一连接中按顺序处理。如果某个请求延迟，后续的请求也会受到影响。
  - **「多个TCP连接」**：虽然开启了长连接，但由于TCP连接的并发限制，浏览器仍然会为同一个域名创建多个连接，导致开销。

1.1版本最主要是改进了默认开启长链接为了解决HTTP1.0的问题，**「HTTP1.1默认开启长连接」**，即让同一个TCP连接服务于多个请求-响应。



在这种情况下，多次请求响应可以共享同一个TCP连接，这不仅减少了TCP的握手和挥手时间，同时可以充分利用TCP「慢启动」的特点，有效的利用带宽。

实际上，在HTTP1.0后期，虽然没有官方标准，但开发者们慢慢形成了一个共识：

「只要请求头中包含Connection:keep-alive，就表示客户端希望开启长连接，希望服务器响应后不要关闭TCP连接。如果服务器认可这一行为，即可保持TCP连接。」

当需要的时候，任何一方都可以关闭TCP连接

>
>
> 扩展知识

> 连接关闭的情况主要有三种：

1. 客户端在某一次请求中设置了 `Connection:close` ，服务器收到此请求后，响应结束立即关闭TCP
2. 在没有请求时，客户端会不断对服务器进行心跳检测（一般每隔1秒）。一旦心跳检测停止，服务器立即关闭TCP
3. 当客户端长时间没有新的请求到达服务器，服务器会主动关闭TCP。运维人员可以设置该时间。

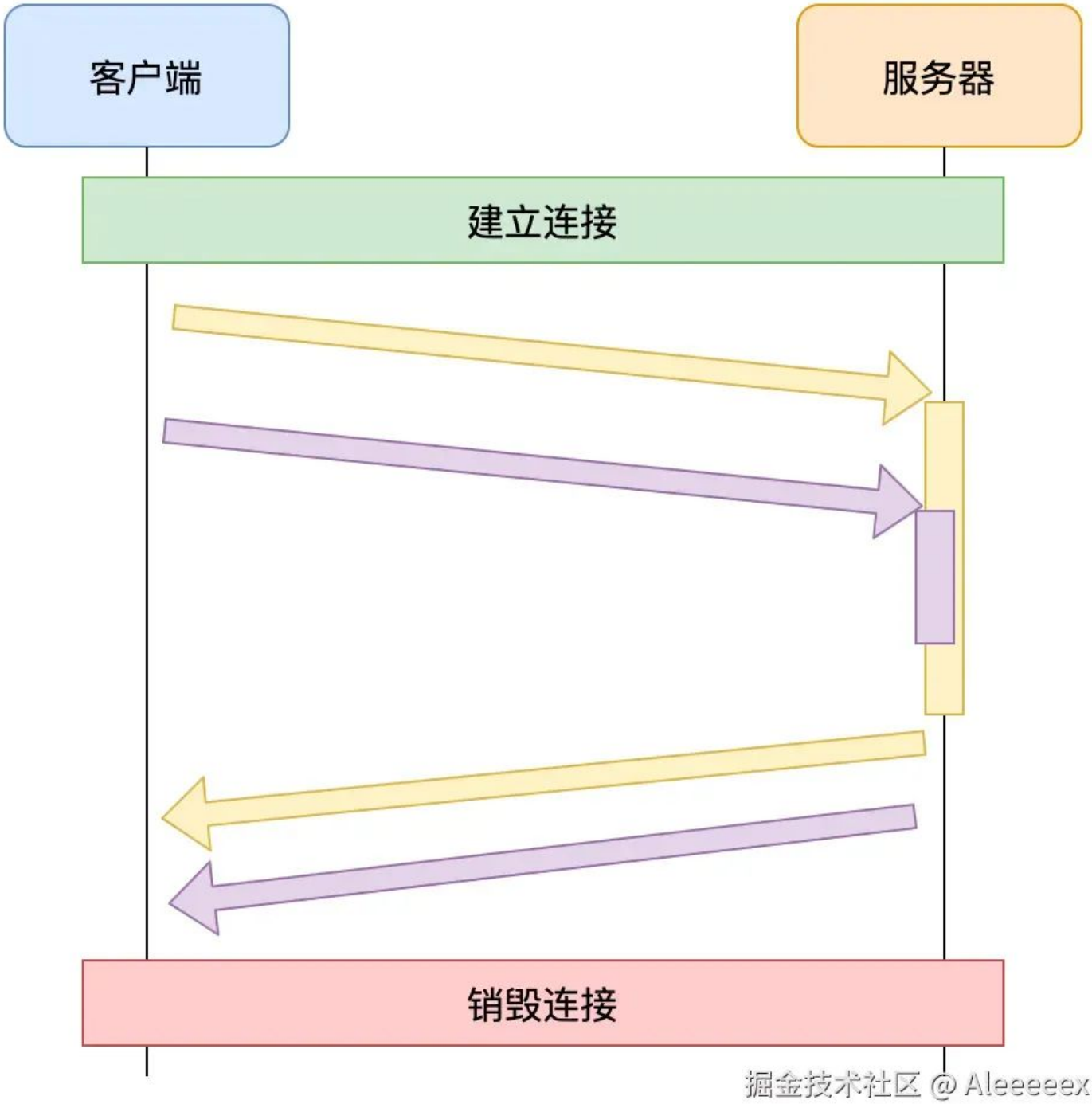
> ☒

由于一个TCP连接可以承载多次请求响应，并在一段时间内不会断开，因此这种连接称之为长连接。

「管道化和队头阻塞」

HTTP1.1允许在响应到达之前发送下一个请求，这样可以大幅缩减带宽限制时间

「但这样做会存在队头阻塞的问题」



由于多个请求使用的是同一个TCP连接，**「服务器必须按照请求到达的顺序进行响应」**

于是，导致了一些后发出的请求，无法在处理完成后响应，产生了等待的时间，而这段时间的带宽可能是空闲的，这就造成了带宽的浪费

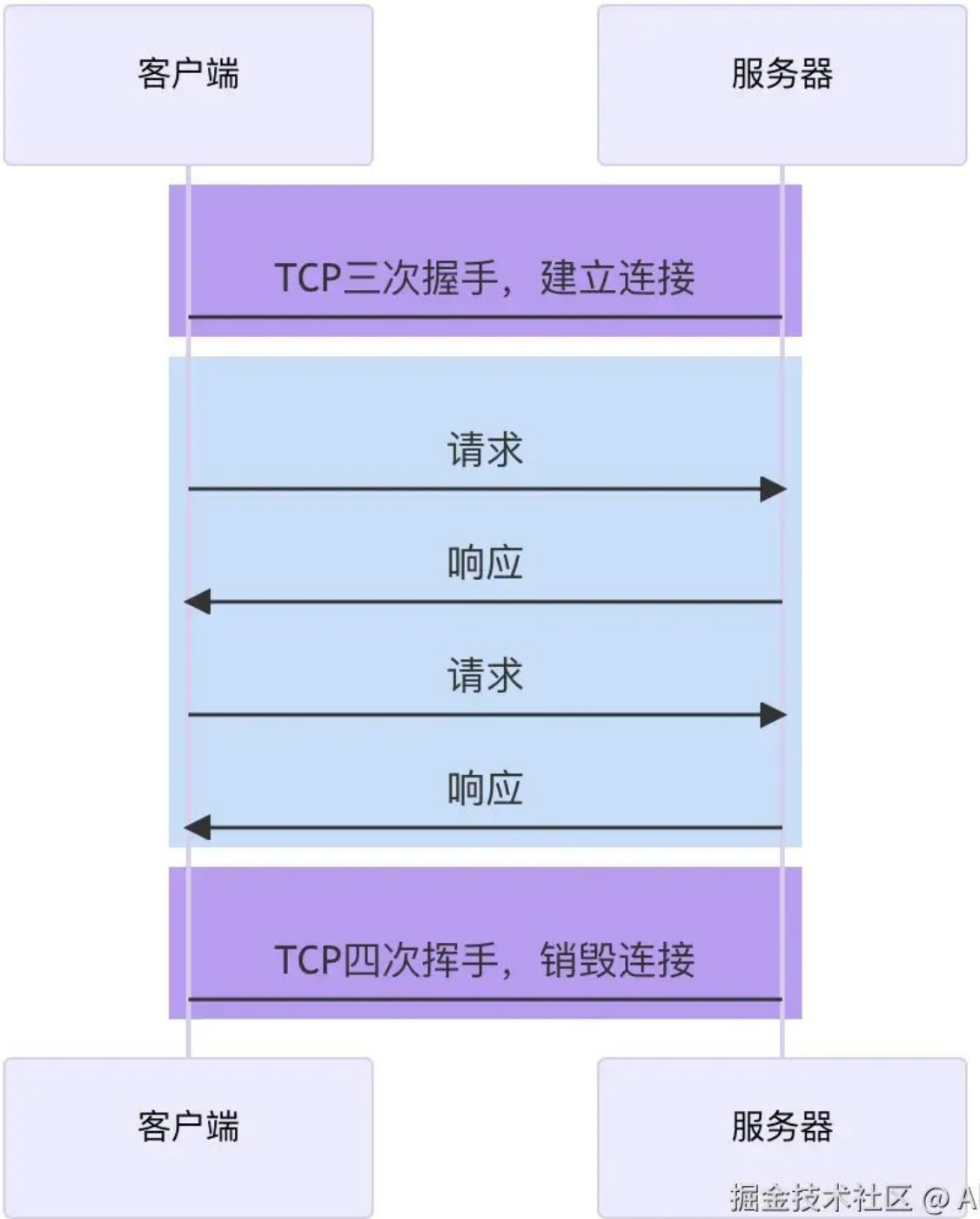
队头阻塞虽然**「发生在服务器」**，但这个问题的根源是客户端无法知晓服务器的响应是针对哪个请求的。

正是由于存在队头阻塞，我们常常使用下面的手段进行优化：

- 通过减少文件数量，从而减少队头阻塞的几率
- 通过开辟多个TCP连接，实现真正的、有缺陷的并行传输

- 浏览器会根据情况，为打开的页面自动开启TCP连接，对于同一个域名的连接最多6个
- 如果要突破这个限制，就需要把资源放到不同的域中

「然而，管道化并非一个成功的模型，它带来的队头阻塞造成非常多的问题，所以现代浏览器默认是关闭这种模式的」



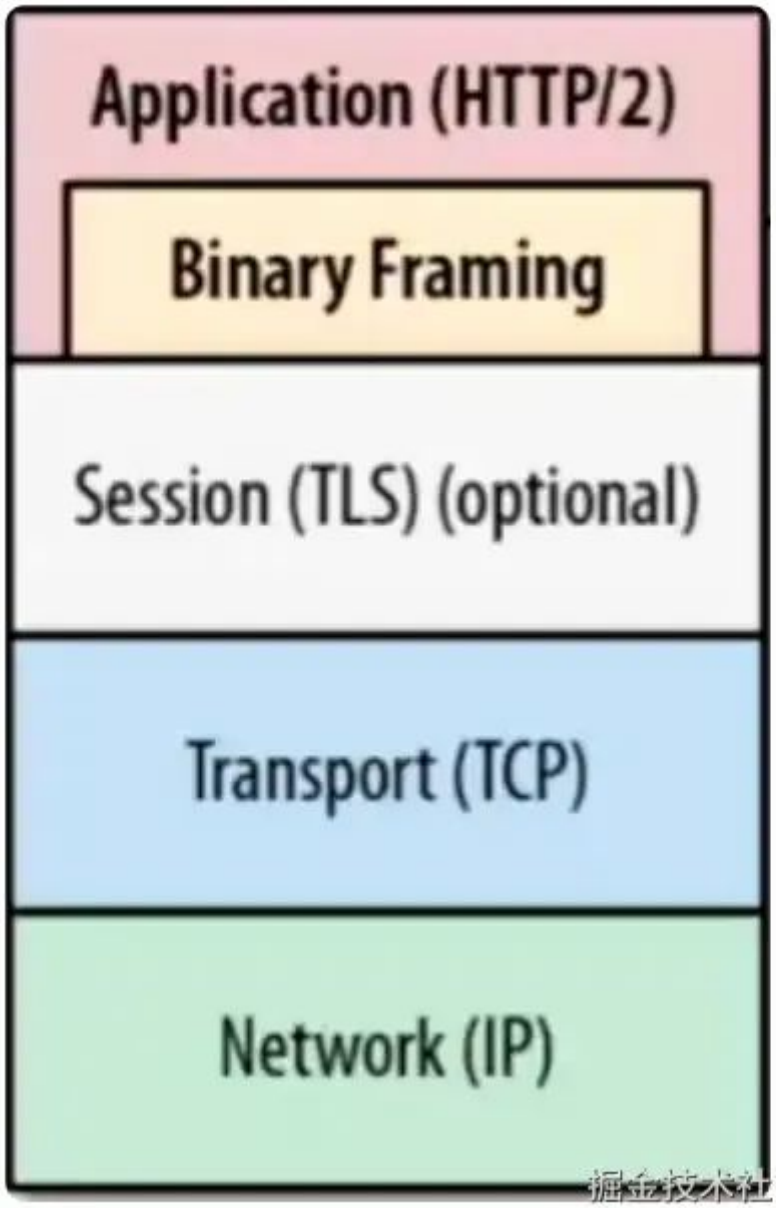
「4. HTTP/2 (2015年) — 性能革命」

HTTP/2是对HTTP协议的根本性改进，旨在解决HTTP/1.x中的性能瓶颈，尤其是资源加载的延迟问题。其核心优势在于支持多路复用、头部压缩等技术，大大提高了性能。

- **「核心特性」**：
  - **「二进制协议」**：HTTP/2采用 **「二进制帧」**（而不是HTTP/1.x的文本协议），提高了数据的传输效率，减少了解析开销。
  - **「多路复用（Multiplexing）」**：通过在一个TCP连接上同时并发多个请求和响应，避免了HTTP/1.x中的队头阻塞问题。多个请求可以同时在同一连接中传输，不再按照顺序依次等待。
  - **「头部压缩（HPACK）」**：HTTP/2对请求和响应的头部进行压缩，减少了冗余信息，提高了带宽利用率。
  - **「服务器推送（Server Push）」**：服务器可以主动推送资源给客户端，无需等待客户端请求。这有助于提前加载所需的资源（如CSS、JS文件）。
  - **「优先级控制」**：允许客户端指定不同资源的优先级，帮助服务器在有限的带宽下优先传输重要资源。
  - **「减少连接数」**：避免了HTTP/1.x中同一域名下创建多个TCP连接的情况，减少了连接数，从而减少了TCP握手和关闭连接的延迟。
- **「性能提升」**：通过多路复用和头部压缩，HTTP/2显著降低了延迟，提高了页面加载速度。
- **「局限性」**：HTTP/2仍然基于TCP协议，存在 **「TCP队头阻塞问题」**（如果TCP连接出现问题，所有的请求都会受到影响）。另外，由于HTTP/2是二进制协议，某些代理和中间件可能需要更新以支持该协议。

「二进制分帧」

HTTP2.0可以允许以更小的单元传输数据，每个传输单元称之为 **「帧」**，而每一个请求或响应的完整数据称之为 **「流」**，每个流有自己的编号，每个帧会记录所属的流。



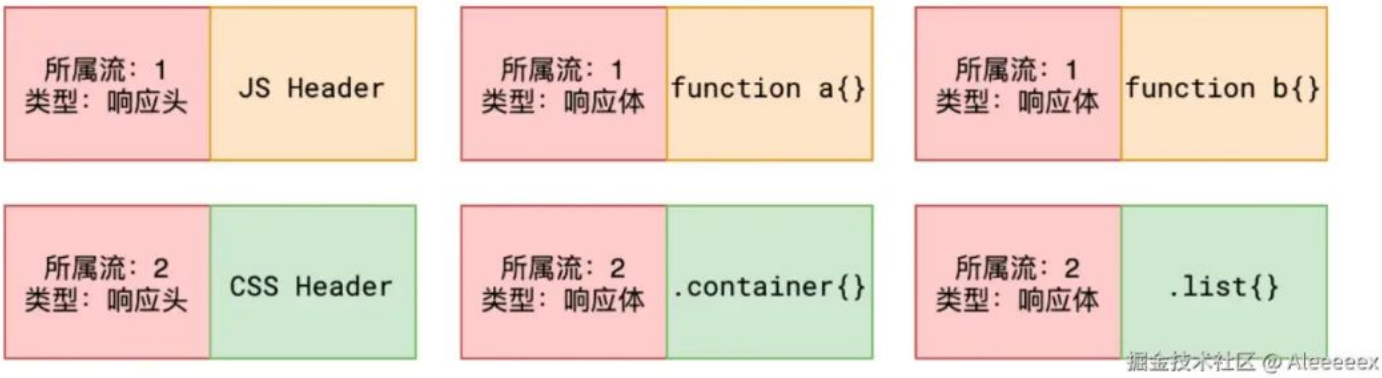
比如，服务器连续接到了客户端的两个请求，一个请求JS、一个请求CSS，两个文件如下：

-
-
-
-
-
-
-

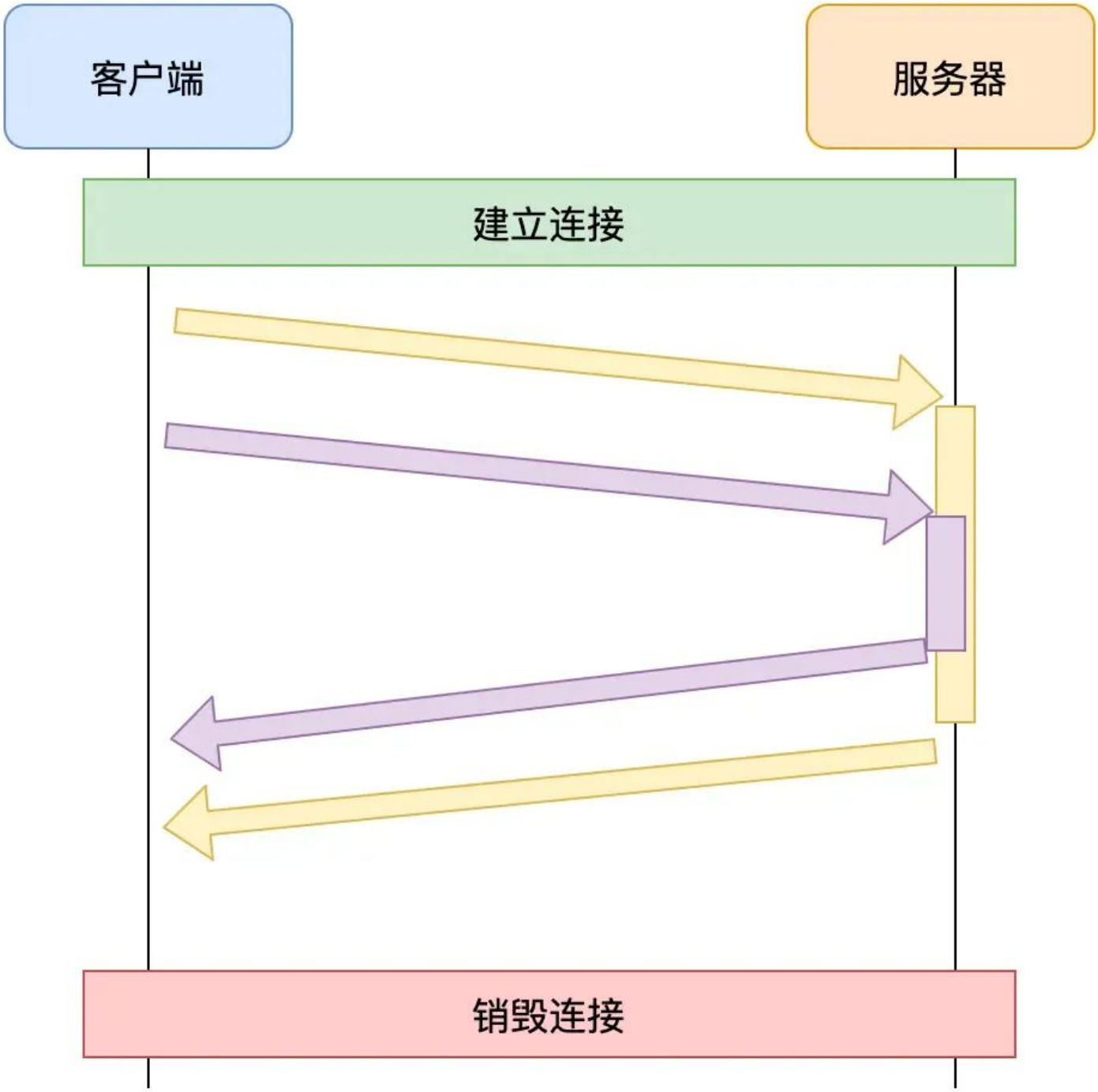
```
1   function a(){}
2   function b(){}
```

```
3   .container{}
4   .list{}
```

最终形成的帧可能如下



可以看出，每个帧都带了一个头部，记录了流的ID，这样做就能够准确的知道这一帧数据是属于哪个流的。



这样就真正的解决了共享TCP连接时的队头阻塞问题，实现了真正的 **「多路复用」**

不仅如此，由于传输时是以帧为单元传输的，无论是响应还是请求，都可以实现并发处理，即不同的传输可以交替进行。

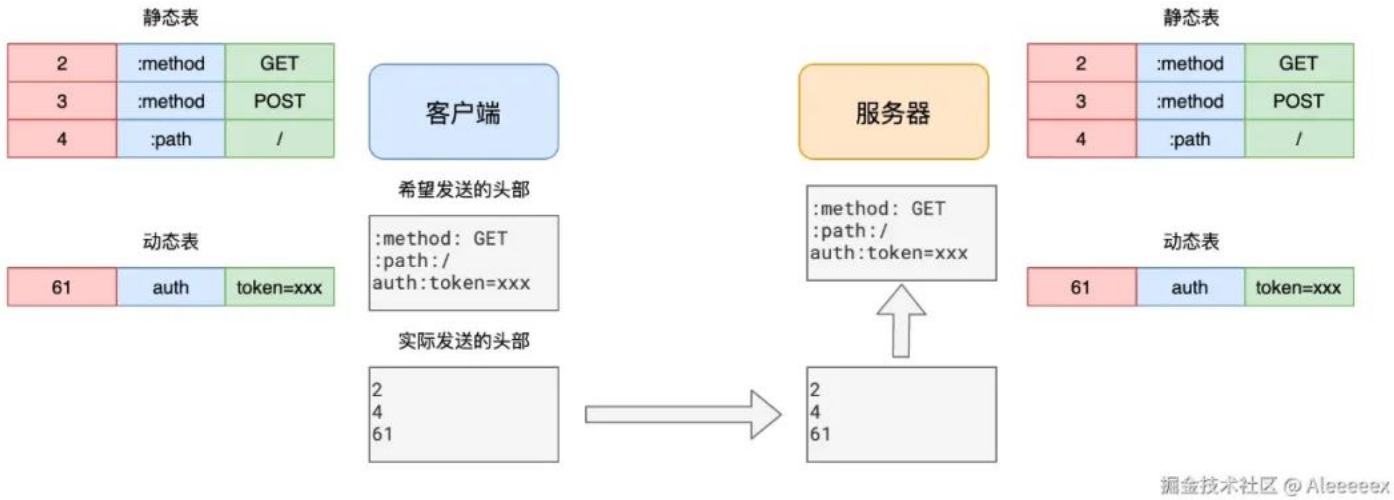
由于进行了分帧，还可以设置传输优先级。

「头部压缩」

HTTP2.0之前，所有的消息头都是以字符的形式完整传输的

可实际上，大部分头部信息都有很多的重复

为了解决这一问题，HTTP2.0使用头部压缩来减少消息头的体积



对于两张表都没有的头部，则使用Huffman编码压缩后进行传输，同时添加到动态表中

「服务器推送」

HTTP2.0允许在客户端没有主动请求的情况下，服务器预先把资源推送给客户端

当客户端后续需要请求该资源时，则自动从之前推送的资源中寻找

「5. HTTP/3 (2022年正式标准化) — 未来趋势」

HTTP/3是HTTP协议的最新版本，基于Google的QUIC协议（Quick UDP Internet Connections）。QUIC协议最初是为了提高移动网络下的传输性能而设计的，HTTP/3将QUIC引入到HTTP协议中，彻底改变了数据传输的方式。

- **「核心特性」**：
  - **「基于QUIC协议」**：HTTP/3不再依赖TCP协议，而是使用QUIC（基于UDP），解决了TCP中的队头阻塞问题。
  - **「快速连接建立」**：QUIC协议的最大优势是 **「零往返时间连接建立（0-RTT）」**，意味着连接建立几乎是瞬时的，从而减少了延迟。
  - **「内置TLS加密」**：HTTP/3强制使用TLS 1.3加密，提升了安全性，减少了握手延迟。
  - **「无队头阻塞」**：QUIC协议解决了TCP的队头阻塞问题，使得HTTP/3能够同时处理多个请求和响应，即使其中一个请求出现延迟，其他请求也不会受到影响。
  - **「移动网络优化」**：QUIC对于网络切换（如Wi-Fi到4G）的支持更加友好，可以保持连接不中断。
- **「性能提升」**：HTTP/3通过减少连接建立时间和无队头阻塞大大降低了延迟，尤其在 **「不稳定的网络环境下」** 表现尤为出色。

- 「挑战与前景」：
  - 「兼容性问题」：HTTP/3需要支持QUIC协议的服务器和客户端，目前大部分现代浏览器（如Chrome、Firefox、Edge）已经支持，但某些旧版本仍不兼容。
  - 「网络设备更新」：由于QUIC使用UDP协议，因此需要网络设备（如防火墙、代理服务器）支持QUIC，可能会带来一些部署上的挑战。

总之

1. 「HTTP/0.9」：最早的单一文本传输协议。
2. 「HTTP/1.0」：引入了请求头、响应头和多种数据类型，但仍然是短连接，性能较低。
3. 「HTTP/1.1」：广泛采用的版本，引入持久连接、管道化、缓存机制等功能，极大改善了性能。
4. 「HTTP/2」：通过二进制协议、多路复用、头部压缩等方式，解决了性能瓶颈，显著提高了数据传输效率。
5. 「HTTP/3」：基于QUIC协议，彻底解决了TCP的队头阻塞问题，提升了移动网络下的性能，并提供了更快的连接建立和更高的安全性。

「讲完了http的发展，我们现阶段投入产出比最高的优化其实是使用http2.0,这个给我们带来的提升不是一星半点的，现在不管是阿里云还是腾讯云都支持2.0版本了，需要后端和运维同学去配合升级。」

1.3.3 请求快一点：预加载和预链接

资源优先级提示（Resource Priority Hints）是一组浏览器提供的优化手段，可以帮助开发者更精确地控制资源的加载顺序和时机，以减少关键资源的阻塞时间，提升页面的加载速度和用户体验。这些机制包括 **「Prefetch、Preload、Preconnect、DNS-Prefetch」**，它们各自针对不同类型的资源加载场景进行优化。下面我们详细讲解每个 API 的概念、用法和最佳实践。

「1. 预取回（Prefetch）」

「概念」：

- **「Prefetch」** 适用于 **「即将需要但当前页面不急需」** 的资源（如下一页的脚本、样式表、图片等）。
- 浏览器会 **「在空闲时间」** 低优先级下载这些资源并缓存，以便用户稍后访问时可以更快加载。

「用法」：

-
-

```
<link rel="prefetch" href="next-page.js" as="script"><link rel="prefetch"
href="next-page.css" as="style">
```

「适用场景」：

- 用户点击某个链接后，下一页的 JS、CSS 已经被预加载，访问下一页时会更快。
- SPA（单页应用）可以预取未来可能访问的页面资源。

「最佳实践」：✅ 适用于 「下一步可能会用到但当前不影响渲染」 的资源，例如：

- 预测用户行为，如新闻网站、电子商务网站的下一页资源预取。
- 「避免滥用」：Prefetch 会占用带宽资源，影响当前页面的加载，因此不适合对大量资源进行预取。

「2. 预加载（Preload）」

「概念」：

- 「Preload」 用于显式告诉浏览器 「高优先级」 加载某个资源（如 JS、CSS、字体、图片等），以便在关键渲染路径上避免阻塞。
- 「与 Prefetch 的区别」：
  - 「Prefetch」 是低优先级加载（用于未来页面）。
  - 「Preload」 是高优先级加载（用于当前页面）。

「用法」：

-
-
-

```
<link rel="preload" href="critical.js" as="script"><link rel="preload" href="styles.css" as="style"><link rel="preload" href="font.woff2" as="font" type="font/woff2" crossorigin="anonymous">
```

「适用场景」：

- 关键资源（如 Web 字体、首屏所需 JS 和 CSS）可以 「提前加载」，避免渲染阻塞。
- 视频文件的预加载，减少白屏时间。

「最佳实践」：✅ 适用于 「当前页面必须用到的关键资源」，例如：

- 「Web 字体」：
  -

```
<link rel="preload" href="/fonts/myfont.woff2" as="font" type="font/woff2" crossorigin="anonymous">
```

- 「关键 CSS 或 JS」：
  -
  -

```
<link rel="preload" href="critical.css" as="style"><link rel="preload" href="important.js" as="script">
```

⚠ 「注意」：

- 「不要滥用」：滥用 Preload 可能会影响其他重要资源的加载。
- 「搭配 `as` 属性」：正确指定资源类型（如 `as="script"` ），否则浏览器可能不会正确处理。

「3. 预连接（Preconnect）」

「概念」：

- 「Preconnect」 用于提前建立到第三方服务器的连接，包括 「DNS 解析、TCP 握手和 TLS 连接」，从而减少请求的延迟。
- 适用于 「跨域资源（如 CDN、API、广告、第三方分析工具）」。

「用法」：

-

```
<link rel="preconnect" href="https://cdn.example.com">
```

如果第三方服务器需要 CORS 访问，建议加上 `crossorigin`：

-

```
<link rel="preconnect" href="https://cdn.example.com" crossorigin>
```

「适用场景」：

- 「CDN 资源」（图片、CSS、JS等）加载优化。
- 「Google Fonts、第三方 API、分析工具」（如 Google Analytics）。

「最佳实践」：✅ 适用于 「需要跨域加载资源的情况」，例如：

- 「CDN 加载的字体文件」
  -

-

```
1   <link rel="preconnect" href="https://fonts.googleapis.com"><link
    rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
```

- 「加载第三方 API」
  -

```
1   <link rel="preconnect" href="https://api.example.com">
```

⚠ 「注意」：

- 仅在 「确实需要的情况下使用」，例如重要的域名，数量也不要超过，否则会浪费 TCP 连接。因为会与目标域名保持10秒的链接，会阻碍其他资源加载

「4. DNS 预取（DNS-Prefetch）」

「概念」：其实跟上面的preconnect有重合，大部分情况下用上面那个就好了

- 「DNS-Prefetch」仅用于提前解析域名的 「DNS 解析」，但不会建立完整的连接。
- 适用于 「低优先级」的跨域资源预解析，提升首次请求的速度。

「用法」：

-

```
1   <link rel="dns-prefetch" href="//cdn.example.com">
```

「适用场景」：

- 适用于 「CDN、第三方 API、广告资源、字体资源等」，加快 DNS 解析时间。

「最佳实践」：✅ 适用于 「第三方资源加载但优先级较低的情况」，例如：

- 「广告、分析工具、CDN」：
  -

```
1   <link rel="dns-prefetch" href="//analytics.google.com">
```

⚠ 「注意」：

- `dns-prefetch` 仅进行 **「DNS 解析」** ，并不会预加载资源，如果资源非常重要，建议改用 `preconnect` 。

「对比总结」

API	作用	优先级	适用场景	示例
「Prefetch」	预取未来可能需要的资源	低	预测下一步操作（如预加载下一页资源）	`<link rel="prefetch" href="next-page.js">`
「Preload」	预加载当前页面的关键资源	高	关键 JS、CSS、字体、视频等	`<link rel="preload" href="critical.js" as="script">`
「Preconnect」	提前建立 TCP 连接	高	重要的第三方资源，如 CDN、API	`<link rel="preconnect" href="https://cdn.example.com">`
「DNS-Prefetch」	提前解析域名	低	第三方资源但优先级较低	`<link rel="dns-prefetch" href="//cdn.example.com">`

「最佳实践示例:」

「1.优化字体加载」

-
-
-
-

```
1
2   <link rel="preconnect" href="https://fonts.googleapis.com"><link
    rel="preconnect" href="https://fonts.gstatic.com" crossorigin><link
```

```
rel="preload" href="/fonts/myfont.woff2" as="font" type="font/woff2"
crossorigin="anonymous">
```

「2.优化 CDN 资源」

-
-

```
<link rel="preconnect" href="https://cdn.example.com"><link rel="dns-
prefetch" href="//analytics.example.com">
```

「3.优化首屏渲染」

-
-

```
<link rel="preload" href="critical.css" as="style"><link rel="preload"
href="important.js" as="script">
```

- 「合理使用 Preload，可以显著提高关键资源的加载速度」，加快首屏渲染。
- 「使用 Preconnect 和 DNS-Prefetch，可以减少第三方资源的加载延迟」，优化网络性能。
- 「Prefetch 适用于预测未来的资源需求」，避免页面跳转时的加载等待。

通过综合使用这些 API，可以有效减少首屏加载时间，提高用户体验，使 Web 应用更加流畅快速。我们还有现成的帮我们添加这些资源提示词的工具。可以根据构建产物，自动生成资源优先级提示代码。这个你们自己找吧，不赘述。

2. 渲染时的性能优化

SSR服务端渲染

「什么是 SSR（服务端渲染）？」

「SSR（Server-Side Rendering）」是一种网页内容渲染技术，其中网页的 HTML 内容由服务器在用户请求时预先渲染生成，而不是由浏览器端的 JavaScript 渲染。也就是说，服务端渲染的网页会在用户首次请求时返回完整的 HTML 文件，浏览器接收到 HTML 后，直接呈现页面，而不需要等待 JavaScript 完全加载和执行才能显示内容。

与传统的客户端渲染（CSR）不同，在客户端渲染中，页面内容通常会先加载一个空的 HTML 页面，随后 JavaScript 会接管并在客户端动态渲染页面。相比之下，SSR 使得页面的内容能够在服务器端完

全渲染好，从而减少了浏览器端的渲染负担。

「说人话：就是vue和react等框架原本就是客户端渲染的，但是SSR就是在服务器渲染好了，所以在SEO上和首页加载的体验上会很好，但是也有缺点，那就是牺牲服务器的算力嘛。」

「SSR 的本质原理：」

1. **「请求到达服务器」**：用户通过浏览器访问网站时，发起 HTTP 请求到服务器。
2. **「服务器渲染页面」**：服务器接收到请求后，生成页面的 HTML 内容。对于动态页面，服务器会执行 JavaScript 代码，获取数据并将其与模板结合，生成完整的 HTML 页面。
3. **「返回完整的 HTML 页面」**：服务器将渲染好的 HTML 页面返回给浏览器，浏览器接收到后直接渲染页面，展示给用户。
4. **「客户端接管」**：一旦 HTML 渲染完毕，客户端的 JavaScript 会接管页面，启用页面中的交互功能，比如动态加载的内容、事件绑定等。此时，浏览器变为一个 SPA（单页应用）。

常用的库和框架有 **「Next.js」**（React）、**「Nuxt.js」**（Vue）等，它们通过服务器端渲染来提高首屏加载速度和 SEO 性能。

不管是 **「Next.js」** 还是 **「Nuxt.js」**，主流的服务端渲染框架主要都是基于2个东西：

- renderToString(element)：**「把我们写的组件渲染成HTML字符串返回给浏览器直接渲染」**
- hydrate(element, container):**「在浏览器端激活，使得我们的事件等交互激活，变成一个SPA应用」**

主要是这两件事，后面我们会展开讲。

「SSR 用来做什么？」

- **「提高 SEO（搜索引擎优化）」**：
- 搜索引擎爬虫无法有效解析和索引使用客户端渲染（CSR）技术的单页应用，因为它们通常无法执行 JavaScript。使用 SSR 时，页面在服务器上生成时已经包含了完整的 HTML 内容，爬虫可以直接抓取这些内容，从而提高 SEO 排名。
- **「提高首屏渲染速度」**：
- SSR 可以将渲染结果直接返回给浏览器，减少了等待 JavaScript 加载和执行的时间。因此，用户在请求页面时可以更快地看到内容，提升用户体验。
- **「提高性能」**：
- 由于在服务器端生成页面，浏览器端无需渲染整个应用，减轻了客户端的负担，尤其对于低性能设备（如手机）非常有效。

具体的实践很多文章都有些，可以是使用框架next、nuxt,也可以使用react或者vue提供的api,不再赘述。

css的优化

每一个网页都离不开 `css` ，但是很多人又认为， `css` 主要是用来完成页面布局的，像一些细节或者优化，就不需要怎么考虑，实际上这种想法是不正确的

作为页面渲染和内容展现的重要环节， `css` 影响着用户对整个网站的第一体验

因此，在整个产品研发过程中， `css` 性能优化同样需要贯穿全程

实现方式有很多种，主要有如下：

- 内联首屏关键CSS
- 异步加载CSS
- 资源压缩
- 合理使用选择器
- 减少使用昂贵的属性
- 不要使用@import

这些方式会和上面的加载性能优化的一节有重合的部分，但是还是完整讲一下

「内联首屏关键CSS」

在打开一个页面，页面首要内容出现在屏幕的时间影响着用户的体验，而通过内联 `css` 关键代码能够使浏览器在下载完 `html` 后就能立刻渲染

而如果外部引用 `css` 代码，在解析 `html` 结构过程中遇到外部 `css` 文件，才会开始下载 `css` 代码，再渲染

所以， `CSS` 内联使用使渲染时间提前

注意：但是较大的 `css` 代码并不合适内联（初始拥塞窗口、没有缓存），而其余代码则采取外部引用方式

「异步加载CSS」

在 `CSS` 文件请求、下载、解析完成之前， `CSS` 会阻塞渲染，浏览器将不会渲染任何已处理的内容

前面加载内联代码后，后面的外部引用 `css` 则没必要阻塞浏览器渲染。这时候就可以采取异步加载的方案，主要有如下：

- 使用javascript将link标签插到head标签最后
-
-
-
-
-
-

```
// 创建link标签const myCSS = document.createElement( "link" );myCSS.rel =
"stylesheet";myCSS.href = "mystyles.css";// 插入到header的最后位置
document.head.insertBefore( myCSS, document.head.childNodes[
document.head.childNodes.length - 1 ].nextSibling );
```

「资源压缩」

利用 `webpack` 、 `gulp/grunt` 、 `rollup` 等模块化工具，将 `css` 代码进行压缩，使文件变小，大大降低了浏览器的加载时间

大型的 CSS 文件不仅增加了页面的下载时间，也增加了浏览器的解析和渲染时间。

- **「使用工具压缩 CSS 文件」**：像 **「CSSNano」** 、 **「PostCSS」** 等工具可以帮助你压缩和优化 CSS 文件，去除无用的空格、注释和不必要的规则。
- **「使用 CSS 代码分割」**：通过工具（如 **「Webpack」** 或 **「Parcel」** ）将 CSS 按需分割，只加载当前页面所需的样式，而不是加载所有样式。

「合理使用选择器」

`css` 匹配的规则是从右往左开始匹配，例如 `#markdown .content h3` 匹配规则如下：

- 先找到h3标签元素
- 然后去除祖先不是.content的元素
- 最后去除祖先不是#markdown的元素

如果嵌套的层级更多，页面中的元素更多，那么匹配所要花费的时间代价自然更高

所以我们在编写选择器的时候，可以遵循以下规则：

- 不要嵌套使用过多复杂选择器，最好不要三层以上
- 使用id选择器就没必要再进行嵌套
- 通配符和属性选择器效率最低，避免使用

「减少使用昂贵的属性」

某些 CSS 属性的计算会比较昂贵，尤其是在复杂布局的情况下。避免不必要的属性，或者使用更高效的替代方案。

在页面发生重绘的时候，昂贵属性如 `box-shadow` / `border-radius` / `filter` /透明度/ `:nth-child` 等，会降低浏览器的渲染性能

- **「避免使用 `box-shadow` 和 `text-shadow` 等过于复杂的属性」**，尤其是在多层嵌套的元素上。这些属性会增加页面渲染的复杂性。
- **「使用 `will-change` 来优化动画」**：当你知道某个元素即将进行动画变化时，可以使用 `will-change` 来告知浏览器优化该元素。
-

-
-
-

```
1        .box {     will-change: transform;   }
```

「不要使用@import」

css样式文件有两种引入方式，一种是 `link` 元素，另一种是 `@import`

`@import` 会影响浏览器的并行下载，使得页面在加载时增加额外的延迟，增添了额外的往返耗时

而且多个 `@import` 可能会导致下载顺序紊乱

比如一个css文件 `index.css` 包含了以下内容： `@import url("reset.css")`

那么浏览器就必须先把 `index.css` 下载、解析和执行后，才下载、解析和执行第二个文件 `reset.css`

「减少不必要的 CSS 重绘和重排」

每当页面中的元素样式发生变化时，浏览器会进行重绘或重排操作，这会影响性能，尤其是在大量 DOM 元素的情况下。

- **「减少 DOM 元素的更改」**：尽量避免频繁地修改元素的布局或样式。例如，不要在动画过程中频繁改变 `width` 、 `height` 、 `margin` 等会导致重排的属性。
- **「使用 `transform` 和 `opacity` 代替 `top` 、 `left` 等属性」**：这些 CSS 属性不会触发重排，浏览器可以通过 GPU 加速动画。
  -
  -
  -
  -
  -
  -
  -
  -
  -
  -
  -
  -

```
1    //推荐.element {  transform: translateX(100px);}
2   //不推荐.element {  position: absolute;  top: 100px;  left: 100px;}
```

「其他」

- 减少重排操作，以及减少不必要的重绘
- 了解哪些属性可以继承而来，避免对这些属性重复编写
- cssSprite，合成所有icon图片，用宽高加上backgroud-position的背景图方式显现出我们要的icon图，减少了http请求
- 把小的icon图片转成base64编码
- CSS3动画或者过渡尽量使用transform和opacity来实现动画，不要使用left和top属性
- 使用gpu来渲染，例如transform willchange等
- 不要多次跟改样式，应该使用类名一次性更改
- 使用 CSS 变量来减少冗余代码

优化 CSS 不是一蹴而就的，但通过一些常规的性能优化手段，可以有效地提高网页加载速度和用户体验。

JS的优化

JavaScript 性能优化是前端开发中重要的部分，直接影响页面的速度、交互响应速度以及整体用户体验。下面我将从 **「代码优化、渲染优化、执行优化、加载优化、内存管理」** 等多个方面提供详细的 JS 性能优化建议，并结合最佳实践进行说明。

「代码优化」

- **「避免使用全局变量：」** 全局变量存储在全局作用域中，会增加变量查找的时间，并可能导致变量污染。尽量使用 `let` 、 `const` 而不是 `var` ，避免变量挂载到 `window` 对象上。使用立即执行函数 (IIFE) 或模块化来封装作用域。
- **「避免不必要的计算：」** 重复计算相同的表达式会增加 CPU 计算开销。应当缓存计算结果，避免重复计算。

「渲染优化」

- **「避免重排和重绘 (Reflow & Repaint)」** ：有几件事会导致重排和重绘：一是修改 DOM 树的结构，移动增删等，二是修改dom元素的几何属性，宽高之类的，三是获取offsetTop、offsetLeft、offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight这样的样式也会触发重排重绘；
- 那我们应该如何避免呢?
  -
  -

```
// 不推荐element.style.width = "100px";element.style.height = "50px";element.style.backgroundColor = "red";
// 推荐element.classList.add("new-style");
```

- 「**缓存位置的值**」 如：offsetTop，不要老访问
- 「**使用 `DocumentFragment` 」** 进行批量 DOM 操作，减少回流次数。
- 「**避免逐个修改样式**」，而是使用 `classList` 统一修改。
- 「**使用 Virtual DOM**」：直接操作 DOM 可能会触发大量的重排和重绘。使用 React / Vue 这样的框架，利用 Virtual DOM 进行高效的 DOM 更新。

```
// React 使用 Virtual DOM 进行高效更新function App() {  const [count, setCount] = React.useState(0);  return (    <button onClick={() => setCount(count + 1)}>      Clicked {count} times    </button>  );}
```

「加载优化」

- 「**延迟加载 JavaScript**」：JavaScript 会阻塞 HTML 解析。应当使用 `async` 或 `defer` 加载脚本。

```
<!-- 不推荐 --><script src="app.js"></script>
<!-- 推荐 --><script src="app.js" async></script>
```

「内存管理」

- **「避免内存泄漏」**:未释放的变量、定时器等会导致内存泄漏，影响性能。
  - **「及时清除定时器」**：

```
let timer = setInterval(() => {  console.log("running");}, 1000);
// 清除定时器clearInterval(timer);
```

  - **「避免 DOM 引用未释放」**：

```
let div = document.getElementById("box");div = null; // 释放 DOM 引用
```

- **「使用 `IndexedDB` 进行本地存储」**：`localStorage` 仅支持字符串存储，性能较差。使用 `IndexedDB` 存储大量结构化数据，提高访问效率。

「使用 Web Worker 进行异步计算」

复杂计算任务会阻塞 UI 渲染，导致页面卡顿。**「使用 Web Worker 进行多线程计算」**，避免阻塞主线程。

-
-
-
-
-
-
-
-
-
-
-
-

```
1
2   // worker.jsself.onmessage = function (event) {  let result = event.data.num
    * 2;  self.postMessage(result);};
3   // 主线程const worker = new Worker("worker.js");worker.postMessage({ num: 10
    });worker.onmessage = function (event) {  console.log("计算结果:",
    event.data);};
```

「其他」

- **「使用事件委托，将事件绑定在父元素上」**：利用事件冒泡机制处理子元素事件。
- **「避免数组、对象的深拷贝」**：使用 JSON 进行深拷贝，或者 `lodash.cloneDeep()` ，会导致大量的对象克隆，影响性能。应当使用结构共享方法，如 `Object.assign()` 或 `...` 。
- **「使用 `Map` 和 `Set` 替代 `Object` 和 `Array` 」**:普通对象和数组在大规模数据操作时，`Map` 和 `Set` 有更高的性能。当键值对存储较多时，使用 `Map` ，当去重查找时，使用 `Set` 。

以上，如果你的页面需要加载大量 JavaScript 资源，可以这样优化：

1. **「使用 Webpack 代码分割」**（ `import()` 进行动态加载)。
2. **「使用 `async` / `defer` 加载外部脚本」** 。
3. **「使用 `IndexedDB` 存储大量数据，而非 `localStorage` 」** 。
4. **「使用 Web Worker 处理复杂计算，避免主线程阻塞」** 。
5. **「使用 `Service Worker` 进行离线缓存，提高页面可用性」** 。

通过这些 **「优化策略」** ，可以显著提升 JavaScript 的执行效率，减少页面加载时间，提高交互流畅度

vue项目优化

Vue 项目的性能优化涉及多个层面，从 **「数据管理、组件优化、事件处理、网络优化、渲染优化」** 等多个维度进行全面优化。可能和前面的加载时的性能优化有重复，但是还是完整阐述。

「数据管理优化」

数据管理是 Vue 性能优化的关键，因为 Vue 依赖响应式数据系统进行 DOM 更新， **「避免不必要的响应式追踪」** 可以提高性能。

- **「避免不必要的响应式数据：」** Vue 3 使用 `reactive()` 和 `ref()` 创建响应式数据，Vue 2 使用 `data()` 进行响应式管理。如果某些数据 **「不会变更」** ，可以使用 `shallowRef()` 或 `shallowReactive()` 降低 Vue 的响应式追踪成本。
  -
  -
  -
  -
  -
  -

```
<script setup>import { ref, shallowRef } from 'vue';
const reactiveData = ref({ count: 0 }); // 响应式（开销较大）const
nonReactiveData = shallowRef({ count: 0 }); // 非响应式（开销较小）</script>
```

- 在数据不会变更但需要在模板中显示或者避免 Vue 进行深层次的依赖收集的时候使用 `shallowRef`
- **「避免 `watch` 监听深层对象：」** 在 Vue 3 中， `watch` 默认 **「不会深度监听」** 对象，而 Vue 2 需要手动设置 `deep: true` 。 **「深度监听会增加性能开销」** ，应尽量避免。
  -
  -
  -
  -
  -
  -
  -

```
<script setup>import { ref, watch } from 'vue';
const user = ref({ name: 'Alice', age: 25 });
// ❌ 不推荐：深度监听整个对象watch(user, (newVal, oldVal) => {
console.log('User changed:', newVal);}, { deep: true });
// ✅ 推荐：仅监听必要的属性watch(() => user.value.name, (newVal, oldVal) => {
console.log('User name changed:', newVal);});</script>
```

- **「避免 `computed` 修改数据:」** `computed` 计算属性应该是 **「纯函数」** ，不应修改任何状态，否则会触发 Vue 的 **「无限依赖追踪」** ，导致性能问题。

```
<script setup>import { computed, ref } from 'vue';
const price = ref(100);const discount = ref(0.9);
// ❌ 不推荐：修改原始数据const finalPrice = computed(() => {  price.value *= discount.value; // 会引起不必要的响应式追踪  return price.value;});
// ✅ 推荐：返回新计算值const finalPriceBetter = computed(() => price.value * discount.value);</script>
```

「组件优化」

Vue 组件的合理拆分和优化可以减少 **「不必要的渲染」** ，提高页面流畅度。

- **「使用 `v-once` 进行静态渲染:」** 如果一个元素的内容不会变更，可以使用 `v-once` ，让 Vue **「只渲染一次」** ，避免额外的 DOM 计算。
  -

```
<h1 v-once>这个标题不会变</h1>
```

- **「使用key 提高 v-for 效率」**
- `v-for` 生成的列表 **「必须使用唯一 `key` 」** ，否则 Vue 可能错误地复用组件，导致渲染效率降低。
  -
  -
  -

```
<ul>  <li v-for="item in list" :key="item.id">{{ item.name }}</li></ul>
```

🚨 **「不要使用 `index` 作为 `key` 」** ，否则在数据变更时，Vue 可能会错误地复用列表项。

- **「组件懒加载」** ：使用 `defineAsyncComponent` 实现组件 **「按需加载」** ，减少首屏加载时间。
  -
  -
  -
  -
  -

```
<script setup>import { defineAsyncComponent } from 'vue';
const LazyComponent = defineAsyncComponent(() =>
import('./LazyComponent.vue'));</script>
<LazyComponent v-if="showComponent" />
```

- 「避免 `v-if` 和 `v-for` 同时使用:」 `v-if` + `v-for` 会导致 「每次渲染都执行判断」，影响性能。应 「先用 `computed` 过滤数据」。一般在外层套个template在套vif或者如下面代码所示

```
<!-- ❌ 不推荐 --><li v-for="item in list" v-if="item.show">{{ item.name }}
</li>
<!-- ✅ 推荐 --><li v-for="item in filteredList">{{ item.name }}</li>
<script setup>import { computed } from 'vue';
const list = ref([{ name: 'A', show: true }, { name: 'B', show: false
}]);const filteredList = computed(() => list.value.filter((item) =>
item.show));</script>
```

「事件优化」

- 「使用防抖（debounce）和节流（throttle）」
- 防抖适用于搜索框等场景，减少 「输入时的频繁请求」。

```
<script setup>import _ from 'lodash';
const fetchResults = _.debounce(() => {  console.log('Fetching search results...');}, 300);</script>
<input @input="fetchResults" />
```

- 节流适用于 `scroll` 、 `resize` 事件，减少 **「高频触发」** 。

```
<script setup>import _ from 'lodash';
const onScroll = _.throttle(() => {  console.log('Handling scroll event');}, 500);</script>
<div @scroll="onScroll">可滚动区域</div>
```

「网络优化」

- **「代码分割:」** 使用 `import()` 进行 **「懒加载」** 。

```
<script setup>const LazyComponent = defineAsyncComponent(() =>
import('./LazyComponent.vue'));</script>
```

- **「开启gzip压缩」**
- Vue CLI 可开启 `gzip` 压缩，减少资源体积。

```
npm install compression-webpack-plugin -D
```

- 在 `vue.config.js` 中配置：

```
const CompressionWebpackPlugin = require('compression-webpack-plugin');
module.exports = {  configureWebpack: {    plugins: [      new
CompressionWebpackPlugin({        test: /.(js|css|html)$/,        threshold:
10240, // 仅压缩大于 10KB 的文件      }),    ],  },};
```

「渲染优化」

- **「使用 `keep-alive` 缓存组件:」** `keep-alive` 避免重复销毁和创建，提高性能。
  -
  -
  -

```
1   <keep-alive>  <router-view /></keep-alive>
```

- **「延迟图片加载」** 使用 `LazyLoad` 使 **「图片懒加载」** 。
  -
  -

```
1   复制编辑<img v-lazy="'image.webp'" />
```

「函数式组件」

在 Vue 中， **「函数式组件」** （Functional Components）是一种没有实例、没有生命周期钩子、没有响应式数据的组件，它们通常被用于呈现简单的 UI，而不需要 Vue 的响应式系统的支持。函数式组件因其简单性和效率，通常能够带来性能优化，尤其在渲染大量静态组件时。所以合理使用函数式组件能带来性能的提升

Vue 中的函数式组件与普通组件最大的区别是它不包含 Vue 的实例对象、生命周期钩子、响应式数据等。它通过一个渲染函数直接返回虚拟 DOM。

函数式组件的特点：

- **「无状态」** ：函数式组件没有实例（没有 `this` ），它们是纯渲染的。
- **「无生命周期」** ：没有如 `mounted` 、 `created` 等生命周期钩子。
- **「更快的渲染性能」** ：由于没有实例和响应式机制，函数式组件的渲染开销较小，尤其在渲染大量静态元素时非常高效。

它能带来：

1. **「更少的内存开销」** 函数式组件不需要创建 Vue 实例，它不依赖 Vue 的响应式系统，因此避免了 Vue 对组件实例进行的一系列性能开销（如依赖追踪、数据更新等）。
2. **「更少的渲染开销」** 由于没有实例化的过程，函数式组件可以更直接地返回渲染结果，这意味着 Vue 在渲染时不需要处理组件的内部状态，也不需要响应式更新，从而减少了虚拟 DOM 的生成和更新开销。

3. **「避免不必要的渲染」** 函数式组件通常是纯渲染的，它们接收的只是 **「输入属性」** ，并根据这些属性渲染输出。因此，它们本身不依赖 Vue 的响应式数据，更新时不会导致父组件重新渲染，从而避免了不必要的渲染。

在 Vue 2 中，函数式组件的实现比较简单，只需要将 `functional: true` 设置为组件的选项即可。

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

```
1
2   <template functional>  <div>    <h1>{{ props.title }}</h1>    <p>{{
    props.content }}</p>  </div></template>
3   <script>export default {  functional: true, // 声明该组件为函数式组件  props: {
      title: String,    content: String  }};</script>
```

在 Vue 3 中， **「函数式组件的创建方式更简洁」** ，直接通过渲染函数（ `render()` ）来定义组件。这里的 `setup()` 语法糖与传统的函数式组件结合使用时，可以更加清晰和高效。

-
-

```
1
2   <script setup>defineProps({  title: String,  content: String});</script>
3   <template>  <div>    <h1>{{ title }}</h1>    <p>{{ content }}</p>  </div>
    </template>
```

对于更高级的渲染函数，你也可以手动使用 `h()` 来返回虚拟 DOM：

```
1   import { h } from 'vue';
```

```
2  export default {  functional: true,  render() {    return h('div', [
   h('h1', this.title),      h('p', this.content)    ]);  }};
```

「何时使用函数式组件？」

1. **「静态内容渲染」**：函数式组件最适合用于 **「静态内容」**，即内容不依赖于组件的内部状态的情况。比如一个展示某些数据的展示组件，或者一个无状态的布局组件。
2. **「高效的列表渲染」**：如果你需要渲染一个包含大量简单静态内容的列表（如卡片、条目等），可以使用函数式组件，这样可以避免每个列表项都拥有自己的 Vue 实例，从而减少内存和性能开销。
3. **「减少不必要的复杂性」**：对于非常简单、无状态的组件，使用函数式组件可以避免使用 Vue 的响应式系统和生命周期钩子，这样组件会更简单、更易于维护。

「第三方组件的按需引入」

在 Vue 项目中，使用第三方 UI 组件库（如 Element Plus、Ant Design Vue 等）可以加速开发，提高 UI 质量。然而，直接全量引入组件库会导致项目体积增大、加载时间变长、性能下降。因此，**「按需引入」** 组件库成为提升 Vue 项目性能的一个关键优化策略。

「那如何实现按需引入？」

- **「以 Element Plus 为例」**
- Element Plus 是 Vue 3 生态中常用的 UI 组件库，默认情况下如果直接全量引入：
  -
  -
  -
  -

```
1  // ❌ 全量引入（不推荐） import ElementPlus from 'element-plus';import 'element-
   plus/dist/index.css';app.use(ElementPlus);
```

- 这样会导致所有组件都被打包，即使只用到了少数几个组件。
- **「✅ 按需引入（推荐）」** 通过 `unplugin-vue-components` 插件实现按需引入：
  -

```
1  npm install unplugin-vue-components unplugin-auto-import -D
```

- 然后在 `vite.config.js` 或 `webpack.config.js` 中配置：
  -

```
1  import Components from 'unplugin-vue-components/vite';import {
   ElementPlusResolver } from 'unplugin-vue-components/resolvers';
2  export default defineConfig({  plugins: [    Components({      resolvers:
   [ElementPlusResolver()], // 自动按需引入    }),  ],});
```

- 这样，Vue 组件在使用时会 **「自动引入对应的组件」** ，无需手动导入，提高开发效率，同时减少打包体积。
- **「以 Ant Design Vue 为例」**
- **「默认全量引入（不推荐）」**

```
1  import Antd from 'ant-design-vue';import 'ant-design-
   vue/dist/antd.css';app.use(Antd);
```

- 这种方式会让整个 `ant-design-vue` 组件库打包到项目中，导致体积过大。
- **「✅ 按需引入」** 可以使用 `unplugin-vue-components` 进行按需加载：

```
1   import { AntDesignVueResolver } from 'unplugin-vue-components/resolvers';
2   export default defineConfig({  plugins: [    Components({      resolvers:
    [AntDesignVueResolver()], // 按需加载 Ant Design Vue    }),  ],});
```

- 或者手动按需引入：

```
1   import { Button, Input } from 'ant-design-vue';app.use(Button);app.use(Input);
```

- 这样只会加载 `Button` 和 `Input` 组件，减少不必要的资源加载。
- **「以 Vant 为例」**
- Vant 是移动端 UI 组件库，按需引入可以通过 `babel-plugin-import` 来实现：

```
1   npm install babel-plugin-import -D
```

- 然后在 `babel.config.js` 中配置：

-
-
-

```
1  module.exports = {  plugins: [    ['import', {      libraryName: 'vant',
   libraryDirectory: 'es',      style: true    }, 'vant']  ]};
```

- 这样在使用 Vant 组件时：
  -
  -
  -

```
1  import { Button, Dialog } from 'vant';app.use(Button);app.use(Dialog);
```

- 仅会打包 `Button` 和 `Dialog` 组件，避免加载整个 Vant 组件库。

1. **「优先使用插件自动按需引入」**（如 `unplugin-vue-components`），减少手动管理的负担
2. **「如果插件不适用，使用手动按需引入」**，只加载需要的组件。
3. **「避免全量引入 UI 组件库」**，除非有明确需求（如动态引入所有组件）。
4. **「结合动态导入（`import()`）进行异步加载」**，减少首屏加载压力。

「优化方式」	「性能提升点」
按需引入 UI 组件库	避免全量加载，减少 JavaScript 体积
自动按需引入插件	自动识别使用的组件，提高开发效率
手动按需引入	更细粒度的控制，适用于特定需求
结合 Tree Shaking	让 Webpack/Vite 去掉未使用的组件
异步加载组件	减少首屏渲染压力，加快页面加载速度

通过按需引入第三方组件库，Vue 项目可以大幅减少打包体积、加快加载速度，提高整体性能和用户体验

react项目优化

React 提供了强大的声明式 UI 开发能力，但如果不注意优化，应用可能会遇到 **「渲染性能瓶颈」** 、 **「状态管理问题」** 、 **「资源加载缓慢」** 等问题。以下是 React 项目中的 **「性能优化策略」** ，涵盖组件渲染优化、状态管理优化、资源优化等多个方面，并结合实际代码示例。

「1. 避免不必要的组件渲染」

React 组件的重复渲染是影响性能的关键因素，减少不必要的渲染可以显著提升性能。

「1.1 使用 `React.memo` 避免不必要的函数组件渲染」

`React.memo` 是一个高阶组件（HOC），用于缓存组件的渲染结果，避免因 **「父组件重新渲染」** 导致子组件重复渲染。

「示例」

-
-
-
-
-
-
-
-
-
-
-
-

```
1   import React from 'react';
2   // 普通组件，每次父组件更新都会重新渲染const Button = ({ label, onClick }) => {
    console.log('Button Rendered');  return <button onClick={onClick}>{label}
    </button>;};
3   // 使用 React.memo 缓存组件，只有 props 变化时才重新渲染const MemoizedButton =
    React.memo(Button);
4   export default MemoizedButton;
```

「✅ 适用场景」

- 组件渲染成本较高，且 `props` 不常变化的情况下。

「⚠ 注意」

- `React.memo` 只对 **「纯组件」** 有效，如果 `props` 是对象或函数，每次渲染都会创建新的引用，可能导致 `React.memo` 失效。

「1.2 使用 `PureComponent` 优化类组件」

对于类组件，可以使用 `React.PureComponent` 代替 `React.Component` ，它会 **「自动进行浅层比较」** ，避免不必要的更新。

「示例」

```
import React, { PureComponent } from 'react';
class Counter extends PureComponent {  render() {    console.log('Counter Rendered');    return <h1>Count: {this.props.count}</h1>;  }}
export default Counter;
```

「✅ 适用场景」

- 适用于 **「类组件」** ，并且 `props` 和 `state` 只包含 **「基本类型」** 。

「⚠ 注意」

- `PureComponent` 只进行 **「浅比较」** ，如果 `props` 传递的是 **「引用类型」** （对象、数组），需要配合 `useMemo` 或 `useCallback` 。

「1.3 shouldComponentUpdate：控制类组件是否重新渲染。」

不赘述

「2. 使用 `useCallback` 和 `useMemo` 缓存计算和函数」

`useCallback` 和 `useMemo` 主要用于缓存 **「不变的函数」** 和 **「计算结果」** ，避免在子组件中因 **「函数重新创建」** 而触发不必要的渲染。

「2.1 使用 `useCallback` 缓存回调函数」

「示例」

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

```
1  import React, { useState, useCallback } from 'react';import MemoizedButton
   from './MemoizedButton';
2  const App = () => {  const [count, setCount] = useState(0);
3    // ✅ 使用 useCallback 缓存函数，避免每次渲染都创建新函数  const handleClick =
   useCallback(() => {    setCount(prevCount => prevCount + 1);  }, []);
4    return (    <div>      <h1>Count: {count}</h1>      <MemoizedButton
   label="Increment" onClick={handleClick} />    </div>  );};
5  export default App;
```

「✅ 适用场景」

- `useCallback` 适用于将函数作为 `props` 传递给子组件的情况。

「2.2 使用 `useMemo` 缓存计算结果」

「示例」

-
-
-
-
-
-
-
-
-
-

```
1   import React, { useState, useMemo } from 'react';
2   const ExpensiveCalculation = ({ num }) => {  const result = useMemo(() => {
     console.log('Computing...');    return num * 2;  }, [num]); // 只有 num 变化时
    才重新计算
3     return <h2>Result: {result}</h2>;};
```

「✅ 适用场景」

- 适用于 **「高计算量」** 的场景，如 **「过滤数据、计算列表项」** 等。

「3. 渲染列表优化」

「3.1 使用 `key` 提高列表渲染性能」

React 依赖 `key` 识别列表项，优化 **「增删修改」** 操作。

「示例」

-
-
-
-
-

```
const List = ({ items }) => (  <ul>    {items.map(item => (      <li key={item.id}>{item.name}</li> // ✅ 使用唯一的 key    ))}  </ul>);
```

「⚠ 避免」

```
{items.map((item, index) => (  <li key={index}>{item.name}</li> // ❌ 避免使用 index 作为 key（不稳定）))}
```

「4. 代码分割 & 资源优化」

「4.1 使用 `React.lazy` 和 `Suspense` 进行懒加载」

React 支持动态加载组件，减少 「初始加载体积」 ，提高首屏渲染速度。

「示例」

```
import React, { Suspense, lazy } from 'react';
// 动态导入组件const LazyComponent = lazy(() => import('./HeavyComponent'));
```

```
3   const App = () => (  <Suspense fallback={<div>Loading...</div>}>
    <LazyComponent />  </Suspense>);
```

「✅ 适用场景」

- **「大组件」** 或 **「非首屏组件」** 按需加载，提高性能。

「5. 状态管理优化」

「5.1 使用 `useReducer` 代替 `useState` 」

当 `useState` 状态复杂时，使用 `useReducer` 进行优化。

「示例」

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

```
import React, { useReducer } from 'react';
const reducer = (state, action) => {  switch (action.type) {    case 'increment': return { count: state.count + 1 };    case 'decrement': return { count: state.count - 1 };    default: return state;  }};
const Counter = () => {  const [state, dispatch] = useReducer(reducer, { count: 0 });
  return (    <div>      <h1>Count: {state.count}</h1>      <button onClick={() => dispatch({ type: 'increment' })}>+</button>      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>    </div>  );};
```

「6. 使用 Immutable 不可变数据」

在 React 项目中，**「数据的可变性（mutability）」** 可能会导致性能问题，影响组件的渲染效率。为了提升性能，推荐使用 **「不可变数据（Immutable Data）」** ，这样可以避免对原始数据的直接修改，减少不必要的重新渲染，提升应用的可维护性。

「6.1 什么是不可变数据（Immutable Data）？」

「不可变数据」 指的是 **「数据一旦创建，就不能被修改」** 。如果想要更新数据，必须 **「创建一个新的数据副本」** ，而不是直接修改原始数据。

在 JavaScript 中，**「基本数据类型」** （如 `string` 、 `number` 、 `boolean` ）是 **「不可变的」** ，但 **「引用数据类型」** （如 `Array` 、 `Object` ）是 **「可变的」** ：

-
-

```
let obj = { name: "Alice" };obj.name = "Bob"; // 直接修改了原始对象
```

这会导致 React 无法正确判断数据变化，进而导致 **「错误的渲染行为或性能损耗」** 。

「6.2 为什么使用 Immutable 数据？」

「1. 降低数据可变性带来的复杂度」

在状态管理中，可变数据会导致难以追踪的 bug，因为对象可以在不同地方被修改，导致不可预测的 UI 变化。使用 Immutable 数据后，每次更新都会创建新的对象，保证数据的可预测性。

「2. 提高 React 性能，减少不必要的渲染」

React 依赖 `shouldComponentUpdate` 、 `React.memo` 或 `PureComponent` 进行性能优化， **「但如果数据是可变的，React 无法正确判断是否需要重新渲染」** 。

「错误示例：直接修改 state，导致 React 不知道数据变化」

-

```
1   const [user, setUser] = useState({ name: "Alice" });
2   const updateUser = () => {  user.name = "Bob";  // 直接修改 state，不会触发组件重新渲染!   setUser(user);};
```

React 发现 `user` 的 **「引用没有变」**，所以不会重新渲染，导致 UI **「不会更新」**。

「正确示例：使用不可变数据」

```
1   const updateUser = () => {  setUser(prevUser => ({ ...prevUser, name: "Bob" })); // 创建新对象};
```

这样 `setUser` 传入的是 **「新的对象引用」**，React 才能检测到变化，触发重新渲染。

「3. 更好的时间旅行（Time Travel）支持」

- 在 Redux 中，撤销/重做（Undo/Redo）功能需要保存状态历史。
- **「Immutable 数据只存储引用，而不是复制整个对象，减少内存占用」**。

「示例：Redux 时间旅行」

```
const history = [];let state = { count: 0 };
const newState = { ...state, count: 1 };history.push(state); // 保存旧状态
state = newState;history.push(state); // 保存新状态
```

这样可以高效地实现 **「时间旅行（Time Travel Debugging）」** 。

「4. 避免并发问题，提高可预测性」

React 18 引入了 **「并发模式（Concurrent Mode）」** ，可能会在 **「不同的渲染阶段修改 state」** 。

「Immutable 数据保证数据只读」 ，避免了 race condition（竞态条件）导致的错误。

「6.3 如何在 React 项目中使用 Immutable？」

「纯 JavaScript 实现 Immutable」

在不使用第三方库的情况下，我们可以 **「使用 `Object.assign()` 或 展开运算符 `...` 」** 来创建新对象：

-
-
-
-
-
-
-

```
const user = { name: "Alice", age: 25 };
// 创建一个新的对象，而不是修改原始对象const newUser = { ...user, age: 26 };
console.log(user.age); // 25console.log(newUser.age); // 26
```

对于数组：

-
-
-
-
-
-

-

```
const list = [1, 2, 3];
// 添加新元素，返回新数组const newList = [...list, 4];
console.log(list); // [1, 2, 3]console.log(newList); // [1, 2, 3, 4]
```

「使用 Immutable.js」

Immutable.js 提供 `List` 、 `Map` 、 `Set` 等不可变数据结构，避免手动深拷贝。

-
-
-
-
-
-
-

```
import { fromJS } from "immutable";
const user = fromJS({ name: "Alice", age: 25 });const newUser =
user.set("age", 26);
console.log(user.get("age")); // 25console.log(newUser.get("age")); // 26
```

「Immutable.js 的优势」 ：

- **「支持嵌套结构」** （无需手动拷贝深层数据）。
- **「优化性能（结构共享）」** ：相同的数据不会重复存储，而是共享引用。

「6.4 总结」

使用 Immutable 数据优化 React 项目，带来的 **「核心收益」** 包括：

1. **「减少不必要的渲染」** （避免对象引用变化导致的 `re-render` ）。
2. **「提升内存管理」** （结构共享，避免深拷贝）。
3. **「更好的时间旅行（Undo/Redo）」** （Redux 状态管理的最佳实践）。
4. **「避免并发问题」** （React 18 并发模式安全）。
5. **「提升代码可维护性」** （防止意外修改 `state` ）。

「7. 避免阻塞 UI 线程」

「7.1 使用 Web Workers」

对于计算量大的任务，可以使用 Web Workers 进行 **「异步计算」** ，避免阻塞 UI。

「示例」

-
-
-
-

```
1   const worker = new Worker('./worker.js');
2   worker.postMessage(1000000); // 发送数据到 Web Workerworker.onmessage = e =>
    console.log('Result:', e.data);
```

「总结」

以上是 React 性能优化的核心策略，包括：

1. 避免不必要的渲染（ `React.memo` 、 `PureComponent` ）。
2. 缓存计算结果和函数（ `useMemo` 、 `useCallback` ）。
3. 代码分割（ `React.lazy` 、 `Suspense` ）。
4. 优化状态管理（ `useReducer` ）。
5. 避免主线程阻塞（Web Workers）。
6. 渲染列表优化（ `key` ）。
7. 资源优化（按需加载）。

合理应用这些优化方法，可以极大提高 React 应用的性能！

3. 开发阶段的优化

开发阶段的优化对于我们开发人员也非常重要，也应该归纳到性能优化的范畴去聊。在 React、Vue 或其他前端框架的开发阶段，我们主要关注 **「构建速度、代码热更新、调试效率、开发环境增强」** 等。优化构建和开发体验，可以显著提升开发效率。

如果是一句话概括优化开发阶段的性能: **「那就是用vite等更新的打包工具。」**

但是一些老旧的项目，使用webpack的我们肯定还是要去知道怎么去优化构建速度，热更新等等来提升我们的开发效率。

如何减少打包耗时?

传统的方案如下：

- **「利用多线程并行编译」**：可以使用 Happypack 或 thread-loader 来提升构建速度，使 Webpack 能够并行处理任务，从而减少单线程的编译瓶颈。
- **「拆分模块，减少重复编译」**：借助 DLL Plugin，可以将不经常变动的依赖库预编译成动态链接库，提高构建效率并减少打包时间。
- **「缓存编译结果，加快增量构建」**：Webpack 提供了内置的 cache 功能，可以存储打包结果，避免重复编译，提升构建性能。
- **「优化项目构建逻辑」**：通过调整项目结构，使打包过程只针对改动的部分代码进行增量编译，减少整体构建的工作量，从而有效降低编译时间。

「1. 利用多线程并行编译」

在 Webpack 默认情况下，整个构建过程是单线程的，而 Webpack 在处理大量模块时会遇到性能瓶颈。为了提高编译速度，可以借助 **「Happypack」** 或 **「thread-loader」** 来进行多线程并行处理。

- **「Happypack」**：可以将 Webpack 任务分发到多个子进程（Worker）并行执行，从而充分利用多核 CPU 资源，提高构建效率。（比较老了 不推荐）
- **「thread-loader」**：适用于 Webpack 4 及以上版本，能够为某些处理繁重任务（如 Babel、TS 转译）的 loader 开启 Worker 线程，实现并行编译，提高速度。

「示例：使用 thread-loader 提升 Babel 编译性能」

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

```
module.exports = {  module: {    rules: [      {        test: /.js$/,
use: [          {            loader: 'thread-loader', // 开启多线程
options: {            workers: 4, // 设定线程数          },          },
      'babel-loader',        ],      },    ],  },};
```

「适用场景」：

- 适用于大规模项目，减少 Webpack 在转换代码（如 Babel 转译）时的 CPU 开销，提高构建速度。
- 但线程开销较大，小型项目可能得不偿失，需权衡是否启用。

「2. 拆分模块，减少重复编译」

Webpack 的 **「DLL Plugin（动态链接库）」** 可以将不常变动的第三方依赖（如 `react` 、 `lodash` 、 `moment` ）单独编译，并在构建时直接引用，避免每次重新编译这些库。

「示例：使用 DLL Plugin 预编译依赖库」

- **「创建 `webpack.dll.js` 配置文件」** （打包第三方库）

```
const path = require('path');const webpack = require('webpack');
module.exports = {  entry: {    vendor: ['react', 'react-dom', 'lodash'], //
需要预编译的库  },  output: {    path: path.resolve(__dirname, 'dll'),
filename: '[name].dll.js',    library: '[name]_library',  },  plugins: [
new webpack.DllPlugin({      name: '[name]_library',      path:
path.resolve(__dirname, 'dll', '[name]-manifest.json'),    }),  ],};
```

- 「在 Webpack 主配置文件中引入 DLL」（引用预编译好的库）

```
const webpack = require('webpack');const path = require('path');
module.exports = {  plugins: [    new webpack.DllReferencePlugin({
manifest: require(path.resolve(__dirname, 'dll', 'vendor-manifest.json')),
}),  ],};
```

「适用场景」：

- 适用于包含大量稳定第三方依赖的项目，如 `react` 、 `vue` 、 `lodash` 、 `moment` 等。
- 避免在每次构建时重复编译不变的依赖，大幅减少编译时间。

「3. 缓存编译结果，加快增量构建」

Webpack 通过 **「cache」** 机制存储编译结果，在后续构建中直接复用，从而提高性能，减少不必要的重复编译。常见的缓存优化策略包括：

- **「babel-loader 缓存」**：减少 Babel 转译的重复计算。
- **「terser-webpack-plugin 缓存」**：加速代码压缩过程。
- **「持久化缓存」**（`cache: { type: 'filesystem' }`）：将缓存存储到磁盘，加快二次构建速度。

「示例：启用 Webpack 缓存」

```
module.exports = {  cache: {    type: 'filesystem', // 持久化缓存（存储在磁盘上）  },  module: {    rules: [      {        test: /.js$/,        use: [          {            loader: 'babel-loader',            options: {              cacheDirectory: true, // 启用 Babel 缓存            },          },        ],      },    ],  },  optimization: {    minimizer: [      new TerserPlugin({        cache: true, // 开启压缩缓存      }),    ],  },};
```

「适用场景」：

- 适用于大型项目，能够有效减少每次构建的时间。
- 开发模式下推荐使用 `cacheDirectory: true` 来提升 Babel 解析速度。

「4. 优化项目构建逻辑（增量编译）」

Webpack 默认情况下，每次编译都会重新解析所有文件，而 **「增量编译」** 只针对变更文件进行重新打包，从而提高构建效率。常见的方法包括：

- **「`webpack --watch` 监听文件变更」**：仅编译改动的文件，提高热更新速度。
- **「`Hot Module Replacement (HMR)` 热更新」**：仅更新变更部分，避免整个应用重新加载。
- **「`module.exports.performance` 设定性能提示」**：帮助识别大文件，避免性能问题。

「示例：启用 Webpack 监听模式（增量编译）」

```
module.exports = {  watch: true, // 监听模式  devServer: {    hot: true, // 启用热更新  },};
```

「适用场景」：

- 适用于开发模式，提高代码改动后的反馈速度。

- 可以结合 `webpack-dev-server` 进行实时更新，无需刷新页面。

其他

- **「externals 提取项目依赖」**
- 构建产物中最大的几个文件都是一些公共依赖包，那么只要把这些依赖提取出来，就可以解决主包过大的问题
- 可以使用 `externals` 来提取这些依赖包，告诉 webpack 这些依赖是外部环境提供的，在打包时可以忽略它们，就不会再打到主包中
- vue.config.js 中配置：
  -
  -
  -
  -
  -
  -
  -
  -
  -

```
1  module.exports = {  configureWebpack: {    externals: {      vue: 'Vue',
   'vue-router': 'VueRouter',      axios: 'axios',      echarts: 'echarts'    }}
```

- 在 index.html 中使用 CDN 引入依赖
  -
  -
  -
  -
  -
  -

```
1  <body>  <script src="http://lib.baomitu.com/vue/2.6.14/vue.min.js"></script>
   <script src="http://lib.baomitu.com/vue-router/3.5.1/vue-router.min.js">
   </script>  <script src="http://lib.baomitu.com/axios/1.2.1/axios.min.js">
```

```
    </script>  <script src="http://lib.baomitu.com/echarts/5.3.2/echarts.min.js">
    </script></body>
```

「总结」

方法	作用	适用场景
「多线程并行编译」	提升编译速度，减少单线程瓶颈	适用于大项目的 Babel 解析、TS 转译
「拆分模块（DLL Plugin）」	预编译不常变动的依赖库，减少重复编译	适用于包含大量稳定依赖的项目
「缓存编译结果」	存储编译结果，加快二次构建	适用于大型项目，提高增量编译速度
「优化构建逻辑（增量编译）」	仅编译变更部分，减少整体构建时间	适用于开发模式，结合热更新提升体验

vite等esbuild内核的打包工具出现

直接改用vite就好了，内在原理在我的其他文章中有讲，不赘述。

4. 分析评估

在前端项目中，性能优化不是盲目的，而是基于量化的指标和监控数据来判断 **「哪些地方需要优化」** 以及 **「如何优化」** 。我们可以通过前端性能监控、分析工具和指标数据，找出性能瓶颈，并针对性地进行优化。

「1. 什么是前端性能监控量化？」

「前端性能监控量化」 是指使用可量化的指标（如页面加载时间、交互响应速度、资源大小等）来评估网页或应用的性能，并通过监测这些指标的数据趋势，找出可能的性能瓶颈。

常见的性能优化需要关注以下几方面：

- **「页面加载性能」** （首屏渲染时间、白屏时间、资源加载速度）
- **「交互响应性能」** （用户操作的延迟、动画流畅度）
- **「代码执行性能」** （JavaScript 运行速度、计算密集型任务优化）
- **「网络请求优化」** （HTTP 请求数、请求大小、CDN 加速）
- **「错误和异常监控」** （JS 报错、网络请求失败）

「2. 前端性能监控的核心指标」

前端性能监控指标可以分为 **「页面加载性能指标」** 和 **「交互体验性能指标」** 。

「2.1 页面加载性能指标」

指标	说明	重要性
「TTFB（Time to First Byte）」	从用户发起请求到服务器返回第一字节的时间	🔥🔥🔥
「FCP（First Contentful Paint）」	页面首次渲染内容出现的时间	🔥🔥
「LCP（Largest Contentful Paint）」	加载最大可视内容（如大图片或大段文本）的时间	🔥🔥🔥
「TTI（Time to Interactive）」	页面可以交互的时间	🔥🔥🔥
「DOMContentLoaded（DCL）」	DOM 解析完成的时间	🔥
「Load Time」	页面所有资源加载完成的时间	🔥

- **「示例：使用 Performance API 监控 LCP」**
-
-
-
-
-

```
1  const observer = new PerformanceObserver((entryList) => {  const entries = entryList.getEntries();  console.log('LCP:', entries[0].startTime);});observer.observe({ type: 'largest-contentful-paint', buffered: true });
```

「2.2 交互体验性能指标」

指标	说明	重要性

「FID（First Input Delay）」	用户首次交互（点击、输入等）与浏览器响应之间的延迟	🔥🔥🔥
「CLS（Cumulative Layout Shift）」	页面布局的视觉稳定性	🔥🔥
「FPS（Frames Per Second）」	页面帧率，影响动画流畅度	🔥🔥
「TBT（Total Blocking Time）」	JavaScript 阻塞主线程的时间	🔥🔥🔥

- 🔹 「示例：使用 Performance API 监控 FID」
-
-
-
-
-
-

```
1   const observer = new PerformanceObserver((entryList) => {
    entryList.getEntries().forEach((entry) => {    console.log('FID:',
    entry.processingStart - entry.startTime);  });});observer.observe({ type:
    'first-input', buffered: true });
```

「3. 如何进行前端性能监测？」

我们可以通过以下方式对前端项目进行监控：

「3.1 使用 Chrome DevTools」

「Chrome DevTools」 提供了一整套分析工具：

1. 「Network 面板」：查看网络请求、资源加载情况。
2. 「Performance 面板」：分析 CPU、JavaScript 执行、帧率等。
3. 「Coverage 面板」：检测未使用的 CSS 和 JavaScript 代码。

- 🔹 「示例：使用 Performance 进行分析」

1. 打开 DevTools（`F12` 或 `Cmd + Option + I`）。
2. 选择 「Performance」 选项卡。
3. 点击 「Start Profiling and Reload Page」 进行录制。

4. 查看 CPU、网络请求、渲染时间等数据。

「3.2 使用 Lighthouse 进行性能评分」

Lighthouse 是 Google 提供的开源工具，能够分析前端性能、SEO、可访问性等。

- **「如何使用 Lighthouse」**

1. **「在 Chrome DevTools 中运行」**
   - 打开 DevTools (`F12`)
   - 进入 **「Lighthouse」** 选项卡
   - 点击 **「"Generate report"」**
   - 查看 **「Performance」** 分数和优化建议
2. **「使用 CLI 运行」**

```
sh
复制编辑
npx lighthouse https://example.com --view
```

1. **「使用 PageSpeed Insights」**
   - 访问 PageSpeed Insights
   - 输入网站 URL，分析性能

「3.3 使用 Web Vitals 监测」

Google 提供的 Web Vitals 可以帮助监测 LCP、FID、CLS 等关键指标。

- **「示例：集成 Web Vitals 进行监控」**

```
js
复制编辑
import { getCLS, getFID, getLCP } from 'web-vitals';

getCLS(console.log);
getFID(console.log);
getLCP(console.log);
```

「3.4 使用 Performance API」

Performance API 允许我们在 JavaScript 代码中监控关键性能数据。

- **「示例：获取页面加载时间」**

```
window.addEventListener('load', () => {  const { loadEventEnd, navigationStart } = performance.timing;  console.log('页面加载时间:', loadEventEnd - navigationStart, 'ms');});
```

「3.5 使用第三方监控平台」

为了监控线上环境的性能，可以使用以下服务：

工具	主要功能
「Google Analytics」	监测页面性能、用户交互
「Sentry」	监测错误和异常
「New Relic」	监测应用运行状态
「Datadog」	监测前端和后端性能
「腾讯云 CLS」	监测 Web 应用日志

- 「示例：使用 Sentry 监控前端错误」

```
import * as Sentry from '@sentry/browser';
Sentry.init({  dsn: 'https://your-dsn@sentry.io/your-project-id',});
try {  throw new Error('测试异常');} catch (error) {
Sentry.captureException(error);}
```

「4. 总结」

「如何找到性能问题？」

1. **「使用 Chrome DevTools 分析页面加载情况」**
2. **「使用 Lighthouse 进行自动化评分」**
3. **「使用 Performance API 监测关键指标」**
4. **「使用 Web Vitals 监测 LCP、FID、CLS」**
5. **「使用第三方监控工具（Sentry、New Relic）」**

通过这些方法，我们可以准确定位性能瓶颈，并进行针对性优化。