# 前端性能优化全解析

## 1. 前端开发中性能优化的核心操作

前端性能优化围绕「加载性能」「渲染性能」「运行性能」三大维度展开，核心操作如下：

- **加载阶段优化**：资源压缩（JS/CSS/图片）、代码分割、懒加载、CDN 接入、DNS 预解析、HTTP 缓存配置、减少 HTTP 请求（雪碧图、资源合并）、使用现代资源格式（WebP、Avif）、预加载关键资源
- **渲染阶段优化**：避免重绘与回流、优化 CSS 选择器、使用 Flex/Grid 布局（性能优于传统布局）、异步加载非关键 CSS、首屏渲染优化（骨架屏、关键 CSS 内联）
- **运行阶段优化**：JS 代码性能优化、避免内存泄漏、使用虚拟 DOM、防抖节流、优化事件绑定（事件委托）、减少 DOM 操作、合理使用 Web Worker（处理耗时任务）

## 2. 代码分割(Code Splitting) 及性能优化实践

### 什么是代码分割

代码分割是指将原本打包在一起的整份 JS/CSS 代码，按一定规则拆分成多个小的代码块（chunk），而非一次性加载所有资源的前端优化技术。它解决了传统打包模式下「打包体积过大、首屏加载时间过长」的问题。

### 开发中如何利用代码分割优化性能

1. **按路由分割（最常用，框架友好支持）**
   - Vue：结合 `vue-router` 的 `import()` 动态导入实现路由懒加载

     Code block

     ```
     const Home = () => import(/* webpackChunkName: "home" */ './views/Home.vue')
     const router = new VueRouter({ routes: [{ path: '/', component: Home }] })
     ```

   - React：结合 `react-router` 与 `import()` 实现

     Code block

     ```
     const About = React.lazy(() => import(/* webpackChunkName: "about" */ './About'))
     // 配合 Suspense 使用
     ```

```
3   <Route path="/about" element={<Suspense fallback={<div>加载中...</div>}>
    <About /></Suspense>} />
```

2. **按组件分割（非核心组件按需加载）**

对弹窗、图表等非首屏必需的组件，使用 `import()` 动态导入，触发特定操作（如点击按钮）时再加载组件代码。

3. **按功能模块分割（第三方库分割）**

对体积较大的第三方库（如 Lodash、ECharts），单独拆分成独立 chunk，避免与业务代码打包在一起，利用浏览器缓存缓存第三方库资源。

- Webpack 配置示例：通过 `splitChunks` 拆分第三方依赖

Code block

```
module.exports = {
  optimization: {
    splitChunks: {
      chunks: 'all',
      cacheGroups: {
        vendor: {
          test: /[\\/]node_modules[\\/]/,
          name: 'vendors',
          priority: -10
        }
      }
    }
  }
}
```

## 3. 懒加载(Lazy Loading) 与预加载(Preloading)

### 核心定义

- **懒加载（延迟加载）**：指资源（图片、组件、路由等）并非页面初始化时加载，而是在其即将进入可视区域或触发特定操作时，才开始加载的技术。
- **预加载（提前加载）**：指在页面核心资源加载完成后、浏览器空闲时，主动提前加载后续可能需要用到的资源（如下一个路由的组件、一张非首屏图片）的技术。

### 核心作用

- **懒加载的作用**：

a. 减小页面初始化时的资源请求体积，缩短首屏加载时间，提升首屏渲染性能

b. 减少无效资源加载（如用户未滚动到的图片无需加载），节省用户带宽和服务器资源

- **预加载的作用**：
  a. 提前缓存后续需要的资源，当用户触发相关操作（如跳转路由、点击按钮）时，资源已缓存，提升交互响应速度，减少等待时间
  b. 合理利用浏览器空闲时间，不阻塞核心资源的加载和渲染

# 4. CDN 定义及核心作用

## 什么是 CDN

CDN（Content Delivery Network，内容分发网络）是一套分布式的边缘节点服务器集群，它会将源站的静态资源（JS、CSS、图片、视频等）缓存到靠近用户的边缘节点上，当用户请求资源时，无需访问源站，直接从最近的边缘节点获取资源。

## CDN 的核心作用

1. **提升资源加载速度**：用户从就近的边缘节点获取资源，减少网络传输距离和延迟，解决「跨地域访问慢」的问题
2. **减轻源站压力**：大部分静态资源请求由边缘节点承接和响应，大幅减少源站的访问量和带宽消耗
3. **提高网站可用性和容灾能力**：CDN 节点多为分布式部署，某一个节点故障时，会自动切换到其他可用节点，避免源站单点故障导致的服务不可用
4. **实现静态资源缓存**：边缘节点会缓存静态资源，重复请求可直接返回缓存内容，进一步提升响应速度

# 5. CDN 原理、使用场景及开发实践

## CDN 的核心原理

1. **资源缓存**：CDN 首次接收到用户请求时，会先向源站请求资源，获取后将资源缓存到该边缘节点，并设置缓存过期时间
2. **调度系统（核心）**：通过 GSLB（全局负载均衡）系统，根据用户的地理位置、网络质量、节点负载情况等因素，智能分配最近、最优的边缘节点给用户
3. **请求转发与响应**：用户请求资源时，先发送到 GSLB 进行节点调度，再由调度后的边缘节点提供资源服务；若节点无缓存或缓存过期，会先回源更新缓存，再响应用户
4. **缓存失效机制**：当源站资源更新时，可通过主动推送、手动刷新或等待缓存过期的方式，让 CDN 节点更新缓存，保证用户获取最新资源

## CDN 的使用场景

1. **静态资源分发**（最核心场景）：JS、CSS、图片（Logo、轮播图）、视频、音频、静态 HTML、字体文件等
2. **大文件下载分发**：安装包、压缩包、大型视频（直播回放、影视资源）等
3. **跨地域网站加速**：面向全国/全球用户的网站，解决不同地域网络延迟差异问题
4. **抗 DDoS 攻击**：部分高端 CDN 服务提供抗 DDoS 能力，可抵御常见的流量型攻击，保护源站安全

### 开发中的 CDN 使用实践

在实际开发中，CDN 的使用非常普遍，常见场景有：

1. **第三方库引入**：直接通过 CDN 引入 Vue、React、jQuery 等第三方库，无需本地打包，减少打包体积

```
Code block
1   <!-- 引入 Vue 的 CDN 资源 -->
2   <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.min.js">
    </script>
```

2. **静态资源部署**：将项目打包后的 JS、CSS、图片等静态资源，上传到 CDN 平台（如阿里云 OSS+CDN、腾讯云 COS+CDN），通过 CDN 域名访问资源

```
Code block
1   <!-- 引用 CDN 上的静态资源 -->
2   <link rel="stylesheet" href="https://cdn.xxx.com/css/app.min.css">
3   <script src="https://cdn.xxx.com/js/app.min.js"></script>
```

3. **图片/视频加速**：将产品图片、宣传视频等上传到 CDN，提升用户浏览和播放体验

## 6. 浏览器缓存 & HTTP 缓存策略

浏览器缓存是指浏览器将已请求过的资源存储在本地，当再次请求该资源时，优先从本地读取，无需重新向服务器请求，从而提升资源加载速度。HTTP 缓存策略是浏览器缓存的核心实现，分为「强缓存」和「协商缓存」两个层级，优先级：强缓存 > 协商缓存。

### 1. 强缓存

- 核心：浏览器直接从本地缓存读取资源，无需和服务器建立连接，响应状态码为 `200 OK (from memory cache)` 或 `200 OK (from disk cache)`
- 控制字段（HTTP 1.0/1.1）：
  - `Expires`：HTTP 1.0 字段，是一个绝对时间戳（如 `Wed, 31 Dec 2025 23:59:59 GMT`），存在本地时间不一致的问题

- `Cache-Control`：HTTP 1.1 字段，优先级高于 `Expires`，支持相对时间配置（如 `Cache-Control: max-age=3600` 表示缓存1小时），常用值：
  - `public`：所有节点（浏览器、CDN 等）均可缓存
  - `private`：仅浏览器可缓存（默认值）
  - `no-cache`：不使用强缓存，直接进入协商缓存
  - `no-store`：不缓存任何资源
  - `max-age`：缓存有效时间（秒）

### 2. 协商缓存

- 核心：浏览器先向服务器发送请求，携带本地缓存的资源标识，服务器根据标识判断资源是否更新，再决定返回「新资源」或「304 Not Modified」（告知浏览器使用本地缓存）
- 资源标识（两组核心字段）：
  - `Last-Modified` & `If-Modified-Since`：基于资源修改时间的标识
    - 服务器返回资源时，通过 `Last-Modified` 告知浏览器资源最后修改时间
    - 浏览器再次请求时，通过 `If-Modified-Since` 携带该时间戳，服务器对比后，若未修改则返回 304，否则返回新资源和新的 `Last-Modified`
    - 缺陷：无法识别资源内容微调（如修改后又还原）、无法识别秒内多次修改
  - `ETag` & `If-None-Match`：基于资源内容的唯一标识（哈希值），优先级高于 `Last-Modified`
    - 服务器返回资源时，通过 `ETag` 生成资源内容的哈希值并返回
    - 浏览器再次请求时，通过 `If-None-Match` 携带该哈希值，服务器对比后，若哈希值不变则返回 304，否则返回新资源和新的 `ETag`
    - 优势：能精准识别资源内容是否变化，不受修改时间影响

### 缓存流程总结

1. 浏览器再次请求资源时，先判断是否命中强缓存（通过 `Cache-Control` / `Expires`）
2. 命中强缓存：直接使用本地缓存，无需请求服务器
3. 未命中强缓存：发送请求到服务器，携带协商缓存标识（`If-Modified-Since` / `If-None-Match`）
4. 服务器判断资源是否更新：未更新返回 304，浏览器使用本地缓存；已更新返回 200 和新资源，同时更新本地缓存

## 7. 如何避免重绘(Repaint)和回流(Reflow)

## 先明确核心概念

- **回流（重排，Reflow）**：当 DOM 元素的**布局属性**（宽高、位置、尺寸、节点增减等）发生变化时，浏览器需要重新计算元素布局并渲染页面，这个过程消耗性能极大
- **重绘（Repaint）**：当 DOM 元素的**样式属性**（颜色、背景、字体颜色等，不影响布局）发生变化时，浏览器无需重新计算布局，仅重新绘制元素外观，性能消耗低于回流，但频繁触发仍会影响性能
- 核心关系：**回流一定会触发重绘，重绘不一定触发回流**

## 避免/减少重绘和回流的核心方案

1. **减少直接 DOM 操作，批量处理 DOM 变更**
   - 方案1：使用 DocumentFragment 作为容器，批量添加/修改节点，最后一次性插入 DOM

     Code block

     ```
     const fragment = document.createDocumentFragment();
     for (let i = 0; i < 100; i++) {
       const div = document.createElement('div');
       div.textContent = `item ${i}`;
       fragment.appendChild(div);
     }
     document.body.appendChild(fragment); // 仅触发1次回流
     ```

   - 方案2：先将元素脱离文档流（隐藏 `display: none` / 移出 DOM），修改完成后再恢复，仅触发2次回流
2. **避免频繁读取布局属性，缓存结果**

频繁读取 `offsetTop`、`clientWidth`、`scrollHeight` 等属性，会强制浏览器刷新布局（因为浏览器需要保证数据最新），导致回流频繁触发。

Code block

```
// 不良实践：循环中频繁读取布局属性，触发多次回流
for (let i = 0; i < 100; i++) {
  div.style.top = div.offsetTop + 10 + 'px';
}

// 优化实践：先缓存属性值，批量修改，仅触发1次回流
const top = div.offsetTop;
for (let i = 0; i < 100; i++) {
  div.style.top = top + 10 * i + 'px';
}
```

3. **使用 CSS 类批量修改样式，而非单个样式逐一修改**

```
// 不良实践：多次修改样式，触发多次回流/重绘
div.style.width = '100px';
div.style.height = '100px';
div.style.backgroundColor = 'red';

// 优化实践：添加单个类，仅触发1次回流/重绘
div.classList.add('box-style');
```

4. **使用绝对定位/固定定位，脱离文档流**

绝对定位（`position: absolute`）、固定定位（`position: fixed`）的元素，其布局变更不会影响其他元素的布局，回流范围更小，性能消耗更低。

5. **避免使用触发回流的 CSS 属性，优先使用 transform/opacity**
   - 避免频繁修改 `width`、`height`、`margin`、`top` 等布局属性
   - 实现元素移动、缩放、透明度变化时，使用 `transform` 和 `opacity`，这两个属性由 GPU 加速，不会触发回流，仅触发合成层更新（性能最优）

```
// 最优实践：无回流，仅合成层更新
div.style.transform = 'translateX(100px)';
div.style.opacity = '0.5';
```

6. **减少 CSS 选择器复杂度，避免复杂选择器匹配**

复杂的 CSS 选择器会增加浏览器的样式计算时间，间接增加回流的性能消耗（后续问题9会详细说明）

7. **避免频繁操作样式表，减少样式重计算**

尽量不要动态创建样式表、频繁添加/删除样式规则，避免浏览器频繁重计算样式。

# 8. Vue、React 中的虚拟DOM如何增强性能

## 核心前提：真实DOM操作的性能瓶颈

真实 DOM 不仅包含元素的布局、样式等数据，还绑定了大量浏览器内置方法，直接频繁操作真实 DOM 会触发大量的回流和重绘，这是前端性能瓶颈的核心来源之一。

## 虚拟DOM（Virtual DOM）的本质

虚拟 DOM 是一个**轻量级的 JavaScript 对象**，它是对真实 DOM 的抽象映射，仅保留真实 DOM 的核心属性（标签名、属性、子节点等），没有真实 DOM 的冗余方法和属性，示例：

Code block

```
// 虚拟DOM对象
const vnode = {
  tag: 'div',
  props: { className: 'box' },
  children: [{ tag: 'span', children: 'Hello Virtual DOM' }],
  key: 1
};
```

### 虚拟DOM增强性能的核心原理

虚拟 DOM 并非「比真实 DOM 更快」，而是通过「**减少不必要的真实 DOM 操作**」来提升性能，核心流程分为3步：

1. **生成虚拟DOM树**：组件渲染时，先根据组件状态（data/props）生成对应的虚拟 DOM 树，这个过程是 JS 内部计算，无任何 DOM 操作，性能消耗极低
2. **对比虚拟DOM差异（Diff算法）**：当组件状态发生变化时，会生成新的虚拟 DOM 树，框架（Vue/React）会通过 Diff 算法，**精准对比新旧虚拟 DOM 树的差异**，只找出需要更新的节点（而非更新整个 DOM 树），这个差异对比过程同样是 JS 内部操作
    - Diff 算法优化策略：同层对比（不跨层级对比，降低复杂度）、key 标识（精准复用节点，避免不必要的节点创建和删除）、类型判断（不同类型节点直接销毁重建，不深入对比）
3. **批量更新真实DOM**：将 Diff 算法找到的「差异节点」批量转换为真实 DOM 操作，一次性更新到页面中，从而**最小化真实 DOM 操作的次数和范围**，大幅减少回流和重绘的触发，最终提升性能

### 补充：框架层面的优化

- Vue：通过响应式系统精准监听状态变化，结合虚拟 DOM 的 Diff 算法，实现按需更新；Vue 3 还引入了 PatchFlags（补丁标记），进一步缩小 Diff 对比范围
- React：通过 Fiber 架构实现可中断的 Diff 过程，优先处理高优先级任务，避免长时间阻塞主线程，提升页面流畅度

## 9. CSS选择器会影响浏览器的渲染性能吗

会影响，但影响程度分场景：**在现代浏览器中，简单的 CSS 选择器对性能的影响几乎可以忽略，但复杂、低效的 CSS 选择器，会增加浏览器的样式计算时间，间接影响渲染性能（尤其是在大型应用或高频 DOM 更新场景下）**。

### 浏览器解析CSS选择器的核心规则

浏览器解析 CSS 选择器的顺序是「**从右到左**」（反向匹配），而非我们直观的从左到右。例如选择器 `div .box p`，浏览器的匹配顺序是：

1. 先找到所有 `<p>` 元素
2. 再向上查找父级，是否存在 class 为 `box` 的元素
3. 最后再向上查找，是否存在 `<div>` 元素
4. 全部匹配成功，才会应用样式

这种反向匹配的目的是：快速过滤掉不匹配的元素，减少后续匹配次数，但也导致「右侧的选择器（关键选择器）」对性能影响最大。

## 低效CSS选择器（应避免）

1. **通配符选择器 `*`**：匹配页面所有元素，性能消耗极大，尤其页面元素较多时

```
Code block
1   * { margin: 0; padding: 0; } // 不推荐，可针对性重置样式
```

2. **深层嵌套选择器**：嵌套层级过深（如 `div > .a > .b > .c > p`），会增加浏览器的匹配层级和计算时间，同时降低样式可维护性
3. **低效关键选择器**：将通配符、标签选择器作为关键选择器（右侧选择器），会匹配大量无关元素

```
Code block
1   .box * { color: red; } // 关键选择器是*，匹配.box下所有元素，低效
2   div .box p { color: blue; } // 关键选择器是p，匹配所有p元素后再向上验证，低效
```

4. **属性选择器（无明确匹配规则）**：如 `[class^="box"]`（匹配class以box开头的元素），需要遍历元素属性进行字符串匹配，性能低于类选择器
5. **伪类选择器（高频触发重计算）**：如 `:hover` 在大量元素上使用，会高频触发样式重计算，影响性能

## 优化CSS选择器的实践

1. 选择器尽量扁平化，嵌套层级不超过3层
2. 优先使用类选择器（`.box`）、ID选择器（`#box`，优先级最高，匹配最快），避免使用标签选择器和通配符作为关键选择器
3. 避免使用不必要的复杂选择器，保持选择器简洁
4. 利用 CSS 继承特性，减少重复选择器书写（如文本颜色、字体可继承到子元素，无需单独给子元素设置）

## 10. JavaScript代码性能优化操作

JS 代码性能优化的核心是「**减少主线程阻塞、降低计算复杂度、减少不必要的操作**」，核心操作如下：

1. **减少主线程阻塞，避免长任务**
   - 使用 Web Worker：将耗时计算（如大数据处理、复杂算法）放到 Web Worker 中执行，Web Worker 运行在后台线程，不阻塞主线程（UI 渲染和交互）
   - 拆分长任务：将超过 50ms 的长任务拆分为多个小任务，通过 `requestIdleCallback` 或 `setTimeout` 分批执行，避免主线程卡顿
2. **优化数据操作，降低计算复杂度**
   - 数据查询优化：使用 Map/Set 替代 Array 进行高频查询（Map 的查找时间复杂度为 O(1)，Array 的 `indexOf` / `find` 为 O(n)）

     Code block

     ```
     // 低效：Array 查找
     const arr = [{ id: 1, name: 'a' }, { id: 2, name: 'b' }];
     const target = arr.find(item => item.id === 2); // O(n)

     // 高效：Map 查找
     const map = new Map([[1, { name: 'a' }], [2, { name: 'b' }]]);
     const target = map.get(2); // O(1)
     ```

   - 避免不必要的循环嵌套：循环嵌套的时间复杂度通常为 $O(n^2)$ 甚至更高，尽量优化为单层循环
   - 缓存计算结果：对重复计算的结果进行缓存（如斐波那契数列的递归优化、组件渲染中的计算属性缓存）
3. **优化DOM操作（前文补充，此处核心强调）**
   - 减少直接 DOM 操作，优先使用虚拟 DOM 或框架提供的 API
   - 使用事件委托（事件冒泡机制），减少事件绑定数量

     Code block

     ```
     // 事件委托：仅给父元素绑定1个事件，处理所有子元素的点击事件
     document.getElementById('parent').addEventListener('click', (e) => {
       if (e.target.tagName === 'LI') {
         console.log(e.target.textContent);
       }
     });
     ```

4. **优化函数调用，减少不必要的开销**

- 避免频繁创建匿名函数：匿名函数无法被缓存，频繁创建会增加内存开销（如在 `render` 方法中创建匿名回调）
- 函数柯里化/防抖节流：减少不必要的函数执行（后文详细说明）
- 避免递归深度过深：递归深度过深会导致调用栈溢出，可改为迭代实现

5. **优化资源加载，减少JS执行时间**
   - 使用 `defer` / `async` 加载非关键 JS 脚本：
     - `async`：异步加载脚本，加载完成后立即执行（不保证执行顺序）
     - `defer`：异步加载脚本，页面解析完成后按顺序执行（推荐用于需要保证执行顺序的非关键脚本）

     Code block
     ```
     <script src="app.js" defer></script>
     <script src="vendor.js" async></script>
     ```
   - 压缩JS代码：使用 Terser、UglifyJS 等工具压缩 JS，移除注释、空格、未使用代码，减小文件体积，提升加载和执行速度
6. **其他优化点**
   - 避免频繁操作数组的 `shift` / `unshift`（会移动数组所有元素，时间复杂度 O(n)），优先使用 `push` / `pop`（时间复杂度 O(1)）
   - 使用 `const` / `let` 替代 `var`，提升代码可读性的同时，避免变量提升带来的不必要开销
   - 减少不必要的属性访问：优先缓存对象属性，避免频繁深层访问（如 `obj.a.b.c` 可缓存为 `const c = obj.a.b.c`）

## 11. JavaScript开发中如何避免内存泄漏

内存泄漏是指：**程序中已分配的内存，由于某种原因无法被垃圾回收机制回收，导致内存占用持续升高，最终可能引发页面卡顿、崩溃等问题**。JS 中避免内存泄漏的核心是「**及时释放无用资源，避免无用引用的存在**」，核心方案如下：

1. **及时清除DOM元素的引用**
   - 当 DOM 元素被移除（如删除节点、组件卸载）时，要清除对该 DOM 元素的所有手动引用（如全局变量、对象属性中的引用）

     Code block
     ```
     // 不良实践：全局变量保留了DOM引用，DOM移除后内存无法释放
     let box = document.getElementById('box');
     document.body.removeChild(box); // DOM已移除，但box仍引用该元素，导致内存泄漏
     ```

```
4
5   // 优化实践：移除DOM后，手动清空引用
6   let box = document.getElementById('box');
7   document.body.removeChild(box);
8   box = null; // 清空引用，垃圾回收机制可回收该内存
```

2. **及时清除事件监听**
   - 当组件卸载或元素移除时，要手动移除对应的事件监听，避免事件监听残留导致的内存泄漏

Code block

```
1   // 组件挂载时绑定事件
2   const handleClick = () => console.log('click');
3   document.addEventListener('click', handleClick);
4
5   // 组件卸载时移除事件监听
6   document.removeEventListener('click', handleClick);
```

   - Vue/React 中：在组件销毁生命周期（Vue `beforeDestroy` / `unmounted`、React `componentWillUnmount` / `useEffect` 清除函数）中清除事件监听

3. **避免使用全局变量，合理管理变量作用域**
   - 避免无意识创建全局变量（如未声明的变量、函数内部 `this` 指向全局）
   - 使用块级作用域（`let` / `const`）替代 `var`，变量超出作用域后会自动释放引用

4. **及时清除定时器（setTimeout/setInterval）**
   - 定时器未清除会一直保留引用，即使组件已卸载，定时器仍会执行，导致内存泄漏

Code block

```
1   // 组件挂载时创建定时器
2   const timer = setInterval(() => {
3     console.log('timer running');
4   }, 1000);
5
6   // 组件卸载时清除定时器
7   clearInterval(timer);
```

5. **避免闭包滥用，及时释放闭包引用**
   - 闭包会保留对外部函数变量的引用，若闭包被长期持有，外部函数变量无法被回收
   - 当闭包不再使用时，手动清空闭包引用

6. **优化第三方库使用**

- 部分第三方库（如ECharts、jQuery）在使用后需要手动销毁实例，否则会导致内存泄漏

```
Code block
1  // ECharts实例创建
2  const myChart = echarts.init(document.getElementById('chart'));
3
4  // 组件卸载时销毁ECharts实例
5  myChart.dispose();
```

7. **避免循环引用**
   - 避免对象之间形成循环引用（如 `a.b = b; b.a = a`），虽然现代浏览器的垃圾回收机制（标记清除法）已能处理循环引用，但在低版本浏览器或特殊场景下仍可能导致内存泄漏，尽量避免不必要的循环引用

## 12. 防抖和节流提高性能的场景

防抖（Debounce）和节流（Throttle）是两种用于「**限制函数执行频率**」的技术，核心作用是：避免函数因高频触发（如滚动、输入、点击）而频繁执行，减少不必要的计算和 DOM 操作，从而提升页面性能和流畅度。

### 防抖（Debounce）

- 核心逻辑：**触发事件后，延迟n秒再执行函数；若在n秒内再次触发事件，则重新计时**，最终只执行一次函数
- 提高性能的场景（需要「最后一次触发」执行的场景）：
  a. **搜索框输入联想**：用户在搜索框输入时，无需每输入一个字符就发送请求，而是等用户输入停止后（如500ms内无输入），再发送联想请求，减少接口请求次数，减轻服务器压力
  b. **窗口大小调整（resize）**：用户调整浏览器窗口大小时，会高频触发 `resize` 事件，使用防抖可在窗口调整完成后，只执行一次布局调整逻辑，减少回流次数
  c. **文本编辑器自动保存**：等用户停止输入后，再执行自动保存操作，避免频繁保存导致的性能开销
  d. **按钮防重复点击**：避免用户快速点击按钮（如提交表单），导致多次触发接口请求，使用防抖可保证只执行一次提交逻辑

### 节流（Throttle）

- 核心逻辑：**触发事件后，每隔n秒只执行一次函数，无论事件触发多频繁**，保证函数在固定时间间隔内只执行一次
- 提高性能的场景（需要「间隔执行」的场景，保证执行频率稳定）：

a. **页面滚动（scroll）**：用户滚动页面时，会高频触发 `scroll` 事件（如滚动一次可能触发几十次），使用节流可每隔100ms执行一次滚动相关逻辑（如懒加载、导航栏吸顶），避免频繁计算和 DOM 操作

b. **鼠标移动（mousemove）**：如拖拽元素、鼠标悬浮跟随效果，高频触发 `mousemove` 事件，使用节流可稳定执行跟随逻辑，提升流畅度

c. **高频点击按钮**：如点赞按钮、刷新按钮，用户快速点击时，使用节流可限制点击频率（如1秒内只能点击一次），避免频繁触发接口请求

d. **视频播放进度条拖拽**：拖拽进度条时，高频触发事件，使用节流可稳定更新播放时间，减少性能开销

## 13. 常见图片优化操作及性能提升方案

图片是前端静态资源中体积占比最高的部分，图片优化的核心是「**在保证视觉效果的前提下，最小化图片体积，减少加载时间**」，常见优化操作如下：

1. **选择合适的图片格式**
   - 静态图片：优先使用 WebP/Avif 格式（压缩率远高于 JPG/PNG，相同视觉效果下体积更小），兼容低版本浏览器时可提供 JPG/PNG 降级方案
     - WebP：支持有损压缩和无损压缩，静态图片体积比 JPG 小 25%-35%，比 PNG 小 50% 左右
     - Avif：压缩率比 WebP 更高，体积更小，但兼容性略差于 WebP
     - JPG：适合色彩丰富的图片（如摄影图、宣传图），有损压缩，体积较小
     - PNG：适合透明图片、图标、线条图（无损压缩，支持透明通道）
     - SVG：适合矢量图（图标、logo、简单图形），无限放大不失真，体积小
   - 动态图片：优先使用 WebM 格式（体积比 GIF 小 50% 以上），兼容低版本浏览器时使用 GIF
2. **图片压缩（无损/有损压缩）**
   - 有损压缩：在可接受的视觉损耗范围内，降低图片质量，减小体积（如 JPG 质量设为 70%-80%，视觉效果无明显差异，体积大幅减小）
   - 无损压缩：不损失图片质量，通过优化图片编码方式减小体积（如 PNG 图片压缩）
   - 工具推荐：在线压缩（TinyPNG、CompressJPG）、构建时压缩（webpack 的 image-webpack-loader）
3. **使用合适的图片尺寸，避免大图小用**
   - 避免加载远大于显示尺寸的图片（如显示尺寸为 200x200px，却加载 1000x1000px 的图片），造成不必要的带宽浪费和加载时间
   - 使用响应式图片：根据设备屏幕尺寸和分辨率，加载不同尺寸的图片

```
<!-- 加载对应srcset + sizes：根据屏幕密度/宽度加载对应图片 -->
<img
  src="image_400w.jpg"
  srcset="image_400w.jpg 400w, image_800w.jpg 800w"
  sizes="(max-width: 768px) 400px, 800px"
  alt="响应式图片"
>
```

4. **图片懒加载（核心优化）**
   - 对非首屏图片（如下方内容图片、列表图片）使用懒加载，只有当图片即将进入可视区域时才加载，减少首屏加载时间
   - 实现方式：原生 `loading="lazy"`（简单易用，现代浏览器支持）、Intersection Observer API（兼容性更好，可自定义逻辑）

Code block

```
<!-- 原生懒加载 -->
<img src="image.jpg" alt="懒加载图片" loading="lazy">
```

5. **使用雪碧图（CSS Sprite）**
   - 将多个小图标（如导航图标、按钮图标）合并为一张大图，通过 CSS `background-position` 定位显示单个图标
   - 优势：减少 HTTP 请求次数（多个图标只需加载一张图片），提升加载速度
6. **图片预加载（针对性使用）**
   - 对后续即将用到的图片（如下一个轮播图、跳转后的首屏图片）使用预加载，提前缓存，提升交互流畅度
7. **使用 CDN 加速图片加载**
   - 将图片上传到 CDN 平台，利用 CDN 的边缘节点加速图片加载，减少网络延迟
8. **避免图片内联（大图片）**
   - 小图标可使用 Base64 内联（减少 HTTP 请求），但大图片不建议内联（会增加 HTML 文件体积，影响首屏加载）

## 14. 性能监控工具及页面性能监控方法

### 常用性能监控工具

#### 1. 浏览器内置工具（核心，无需额外安装）

- **Chrome DevTools（最常用）**

- Performance 面板：录制页面加载和运行过程，分析帧率、主线程阻塞、回流重绘、JS 执行时间等，定位性能瓶颈（如长任务、频繁回流）
- Network 面板：监控所有资源的加载时间、大小、请求状态，分析资源加载瓶颈（如大文件加载慢、请求阻塞）
- Layers 面板：查看页面的合成层，分析 GPU 加速情况
- Console 面板：通过 `console.time()` / `console.timeEnd()` 手动测试代码执行时间
- Performance Insights 面板：更简洁的性能分析面板，适合快速定位首屏加载、交互卡顿问题

- **Chrome Lighthouse**
  - 内置在 Chrome DevTools 中（Audits 面板），可生成全面的性能报告，包含核心 Web 指标（LCP、FID、CLS）、SEO、可访问性等评分，并给出优化建议
  - 支持手动运行，也可集成到 CI/CD 流程中，实现自动化性能监控

### 2. 第三方工具/库

- **Web Vitals**：Google 推出的前端性能监控库，专门用于监控核心 Web 指标（LCP、FID、CLS、INP），可将数据上报到自定义服务端
- **Sentry**：不仅支持错误监控，还支持性能监控，可跟踪页面加载、API 请求、函数执行的耗时，定位线上性能问题
- **New Relic**：企业级应用性能监控工具，支持前端、后端全链路性能监控
- **Fiddler**：网络抓包工具，监控 HTTP/HTTPS 请求，分析资源加载和接口性能问题

## 页面性能监控方法

页面性能监控分为「**手动监控（开发阶段）**」和「**自动监控（线上阶段）**」两类：

### 1. 开发阶段：手动监控（定位性能瓶颈）

1. **使用 Chrome DevTools Performance 面板**
   - 步骤：打开 DevTools → 切换到 Performance → 点击 Record 按钮 → 操作页面 → 停止录制 → 分析报告
   - 关注指标：FPS（帧率，越高越好，60fps 为流畅）、长任务（超过 50ms 的任务，会导致卡顿）、Call Stack（函数调用栈，定位耗时函数）、Layout（回流）、Paint（重绘）
2. **使用 Chrome Lighthouse 生成性能报告**
   - 步骤：打开 DevTools → 切换到 Lighthouse → 勾选 Performance → 点击 Generate report → 查看评分和优化建议
   - 关注核心 Web 指标：
     - LCP（最大内容绘制）：衡量首屏加载性能，目标 < 2.5s

- FID（首次输入延迟）：衡量交互响应性能，目标 < 100ms（已逐步被 INP 替代）
- CLS（累积布局偏移）：衡量页面稳定性，目标 < 0.1
- INP（交互到下一步绘制）：衡量交互流畅度，目标 < 200ms

3. **手动埋点测试代码性能**
   - 使用 `console.time()` 测试单个函数或代码块的执行时间

```
Code block
1  console.time('arraySort');
2  const arr = Array(100000).fill(0).map(() => Math.random());
3  arr.sort();
4  console.timeEnd('arraySort'); // 输出: arraySort: 12.345ms
```

   - 使用 `performance` API 更精细地监控性能

```
Code block
1  // 记录性能标记
2  performance.mark('start');
3  // 执行耗时操作
4  doHeavyTask();
5  performance.mark('end');
6  // 计算标记之间的时间差
7  performance.measure('taskTime', 'start', 'end');
8  const measure = performance.getEntriesByName('taskTime')[0];
9  console.log('任务耗时: ', measure.duration);
10 // 清除性能标记
11 performance.clearMarks();
12 performance.clearMeasures();
```

## 2. 线上阶段：自动监控（监控线上用户体验）

1. **监控核心 Web 指标（基于 Web Vitals/Performance API）**
   - 通过 `Performance API` 获取 LCP、CLS 等指标，通过 `EventTarget` 获取 FID/INP 指标
   - 将监控数据上报到服务端，进行统计分析，当指标异常时（如 LCP > 4s），触发告警
2. **监控资源加载性能**
   - 通过 `performance.getEntriesByType('resource')` 获取所有资源的加载时间、大小等信息，监控慢加载资源（如加载时间 > 3s 的 JS/CSS/图片）
3. **监控 JS 错误和长任务**

- 监听 `window.onerror` 和 `window.unhandledrejection` 捕获 JS 错误，上报错误信息和上下文
- 通过 `PerformanceObserver` 监听长任务，上报长任务信息，定位线上卡顿问题

Code block

```
// 监控长任务
const observer = new PerformanceObserver((list) => {
  const entries = list.getEntries();
  entries.forEach((entry) => {
    // 上报长任务数据
    fetch('/api/report/longtask', {
      method: 'POST',
      body: JSON.stringify({
        duration: entry.duration,
        startTime: entry.startTime,
        type: 'longtask'
      })
    });
  });
});
observer.observe({ entryTypes: ['longtask'] });
```

4. **集成第三方监控平台**
   - 直接集成 Sentry、New Relic 等第三方平台，无需手动开发监控逻辑，快速实现线上性能监控和告警

# 15. DNS 预解析（DNS Prefetching）配置及应用

## 核心定义

DNS 预解析（DNS Prefetching）是指：**浏览器在页面加载过程中，提前解析页面中可能需要访问的域名的 DNS 解析记录，并缓存起来**。当用户后续需要访问该域名的资源时，无需再进行 DNS 解析（或从缓存中读取），减少 DNS 解析时间，提升资源加载速度。

## DNS 解析的性能瓶颈

DNS 解析是资源加载的第一步，通常需要 20-100ms（甚至更长，取决于网络环境和域名复杂度），若页面中存在多个外部域名（如 CDN 域名、接口域名），DNS 解析时间会累积，影响页面加载速度。DNS 预解析的核心就是提前消耗少量资源，换取后续资源加载的时间节省。

## 配置 DNS 预解析的方法

1. **HTML 标签配置（最常用，推荐）**

使用 `<link rel="dns-prefetch" href="https://xxx.com">` 标签，在 HTML `<head>` 中配置需要预解析的域名，浏览器会自动在空闲时进行 DNS 解析。

Code block

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>DNS 预解析示例</title>
  <!-- 预解析 CDN 域名 -->
  <link rel="dns-prefetch" href="https://cdn.xxx.com">
  <!-- 预解析 接口域名 -->
  <link rel="dns-prefetch" href="https://api.xxx.com">
  <!-- 预解析 第三方资源域名 -->
  <link rel="dns-prefetch" href="https://fonts.googleapis.com">
</head>
<body>
  <!-- 页面内容 -->
</body>
</html>
```

- 注意：`href` 属性只需填写域名（无需带具体路径），如 `https://cdn.xxx.com` 而非 `https://cdn.xxx.com/css/app.css`
- 优先级：可配合 `preconnect` 使用（`preconnect` 不仅预解析 DNS，还会建立 TCP 连接，优先级更高，消耗也更大）

Code block

```
<!-- preconnect：预解析 DNS + 建立 TCP 连接（适合高频访问的域名） -->
<link rel="preconnect" href="https://cdn.xxx.com">
<!-- dns-prefetch：仅预解析 DNS（适合低频访问的域名） -->
<link rel="dns-prefetch" href="https://api.xxx.com">
```

### 2. HTTP 响应头配置

服务器通过设置 `X-DNS-Prefetch-Control` 响应头，控制浏览器是否开启 DNS 预解析（默认情况下，浏览器会自动开启 DNS 预解析）

- `X-DNS-Prefetch-Control: on`：开启 DNS 预解析（默认值）
- `X-DNS-Prefetch-Control: off`：关闭 DNS 预解析（不推荐，除非有特殊需求）
- 示例（Nginx 配置）：

Code block

```
add_header X-DNS-Prefetch-Control on;
```

3. **禁用不必要的 DNS 预解析**

若页面中无需预解析任何域名，或需要手动控制解析时机，可通过以下方式禁用：

- HTML 标签：`<meta http-equiv="X-DNS-Prefetch-Control" content="off">`
- HTTP 响应头：`X-DNS-Prefetch-Control: off`

## DNS 预解析的应用场景

1. **页面中包含外部域名资源**

当页面中使用了 CDN 域名、第三方资源域名（如字体、地图、统计脚本）、接口域名等外部域名时，提前预解析这些域名，减少后续资源加载的 DNS 解析时间

2. **预解析下一个路由的域名**

对于单页应用，可根据用户行为（如鼠标悬浮在导航链接上），预解析下一个路由对应的接口域名或静态资源域名，提升路由跳转后的加载速度

3. **优化首屏加载性能**

将首屏需要加载的外部资源域名（如 CDN 域名）优先进行 DNS 预解析，缩短首屏资源的加载时间，提升首屏渲染性能

## 注意事项

1. 不要过度预解析域名：过多的 DNS 预解析会消耗浏览器资源和用户带宽，建议只预解析确实需要访问的核心域名
2. 优先使用 `preconnect` 处理高频访问域名：对于首屏加载必须用到的域名（如 CDN 域名），使用 `preconnect`（预解析 DNS + 建立 TCP 连接），进一步减少资源加载时间；对于非核心域名，使用 `dns-prefetch` 即可
3. 兼容低版本浏览器：`dns-prefetch` 兼容所有现代浏览器，以及 IE 9+ 浏览器，无需担心兼容性问题

## 总结

1. 前端性能优化覆盖加载、渲染、运行三大阶段，核心是减少资源体积、减少 DOM 操作、减少不必要的执行频率
2. 代码分割、懒加载、CDN、HTTP 缓存是提升加载性能的核心手段；避免回流重绘、优化 CSS 选择器是提升渲染性能的关键
3. 虚拟 DOM 通过最小化真实 DOM 操作提升性能，防抖节流通过限制函数执行频率减少性能开销

4. 性能监控分为开发阶段（Chrome DevTools/Lighthouse）和线上阶段（Web Vitals/第三方平台），需针对性使用
5. DNS 预解析、图片优化、JS 代码优化等细节操作，是性能优化的重要补充，能进一步提升用户体验

（注：文档部分内容可能由 AI 生成）