# 前端性能优化

## 简介

性能优化是把双刃剑，有好的一面也有坏的一面。好的一面是能提升网站性能，坏的一面就是配置过于麻烦，或要遵守的规则太多。并且某些性能优化规则并不适用所有场景，需要谨慎使用。

## 参考来源

* [前端性能优化 24 条建议（2020）](https://segmentfault.com/a/1190000022205291?utm_source=weekly&utm_medium=email&utm_campaign=SegmentFault%20%E7%B2%BE%E9%80%89%E6%AF%8F%E5%91%A8%E7%B2%BE%E9%80%89%E4%B8%A8%E5%B0%8F%E7%A8%8B%E5%BA%8F%E6%A1%86%E6%9E%B6%E8%BF%90%E8%A1%8C%E6%97%B6%E6%80%A7%E8%83%BD%E5%A4%A7%E6%B5%8B%E8%AF%84%E4%B8%A8%E5%89%8D%E7%AB%AF%E6%80%A7%E8%83%BD%E4%BC%98%E5%8C%96%2024%20%E6%9D%A1%E5%BB%BA%E8%AE%AE)
* [性能为何至关重要](https://developers.google.com/web/fundamentals/performance/why-performance-matters?hl=zh-cn)
* [高性能网站建设指南](https://github.com/woai3c/recommended-books/blob/master/%E5%89%8D%E7%AB%AF/%E9%AB%98%E6%80%A7%E8%83%BD%E7%BD%91%E7%AB%99%E5%BB%BA%E8%AE%BE%E6%8C%87%E5%8D%97.pdf)
* [Web 性能权威指南](https://github.com/woai3c/recommended-books/blob/master/%E5%89%8D%E7%AB%AF/Web%E6%80%A7%E8%83%BD%E6%9D%83%E5%A8%81%E6%8C%87%E5%8D%97.pdf)
* [高性能 JavaScript](https://github.com/woai3c/recommended-books/blob/master/%E5%89%8D%E7%AB%AF/%E9%AB%98%E6%80%A7%E8%83%BDJavaScript.pdf)
* [高效前端：Web 高效编程与优化实践](https://book.douban.com/subject/30170670/)

## 1. 减少 http 请求

### 参考资料

* [understanding-resource-timing](https://developers.google.com/web/tools/chrome-devtools/network/understanding-resource-timing)

### 将多个小文件合并为一个大文件，从而减少 http 请求次数

一个 http 请求中真正下载数据的时间占比 A，文件越小，这个比例越小，可能不足 10%，文件越大，比例就越高。

## 2. 使用 http2

### 参考资料

* [HTTP2 简介](https://developers.google.com/web/fundamentals/performance/http2/?hl=zh-cn)
* [HTTP2 与 web 性能优化](https://imququ.com/post/http2-and-wpo-3.html#toc-3)
* [基于 Node.js 的 HTTP/2 Server 实践](https://juejin.im/post/5b0e9ff4518825153515aade)

### http2 所有性能增强的核心在于新的的二进制分帧层，它定义了如何封装 HTTP 消息并在客户端与服务器之间传输。

## 3. 使用服务端渲染

* 客户端渲染 获取 HTML 文件，根据需要下载 JavaScript 文件，运行文件，生成 DOM，再渲染；
* 服务端渲染 服务端返回 HTML 文件，客户端只需解析 HTML。
* 优点：首屏渲染好，SEO 好；
* 缺点：配置麻烦；

## 4. 静态资源使用 CDN

内容分发网络（CDN）是一组分布在多个不同地理位置的 Web 服务器。我们都知道，当服务器离用户越远时，延迟越高。CDN 就是为了解决这一问题，在多个位置部署服务器，让用户离服务器更近，从而缩短请求时间。

### 参考资料

* [CDN 是什么？使用 CDN 有什么优势？](https://www.zhihu.com/question/36514327/answer/193768864)

## 5. 将 CSS 放在文件头部，JavaScript 文件放在底部

## 6. 使用字体图标 iconfont 代替图片图标

### 那么现在 antd 中的 icon 组件怎么样？

### icon 中的图标组件呢？

### [Iconfont-阿里巴巴矢量图标库](https://www.iconfont.cn/)

## 7. 善用缓存，不重复加载相同的资源

### 参考资料

* [数据摘要要算法](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E6%91%98%E8%A6%81%E7%AE%97%E6%B3%95/3286770?fromtitle=%E6%91%98%E8%A6%81%E7%AE%97%E6%B3%95&fromid=12011257)
* [张云龙--大公司里怎样开发和部署前端代码？](https://www.zhihu.com/question/20790576/answer/32602154)

## 8. 压缩文件

### webpack 使用以下插件进行压缩

* JavaScript：UglifyPlugin
* CSS: MiniCssExtractPlugin
* HTML: HtmlWebpackPlugin

### gzip 压缩

* 可以使用 webpack 和 node 配置 gzip 的方式

## 9. 图片优化（重点）

### 1. 图片延迟加载

在页面中，先不给图片设置路径，只有当图片出现在浏览器的可视区域时，才去加载真正的图片，这就是延迟加载。对于图片很多的网站来说，一次性加载全部图片，会对用户体验造成很大的影响，所以需要使用图片延迟加载。

首先可以将图片这样设置，在页面不可见时图片不会加载：

<img data-src="https://avatars0.githubusercontent.com/u/22117876?s=460&u=7bd8f32788df6988833da6bd155c3cfbebc68006&v=4">

等页面可见时，使用 JS 加载图片：

const img = document.querySelector('img')  
img.src = img.dataset.src

这样图片就加载出来了，完整的代码可以看一下参考资料。

#### 参考资料：

* [web 前端图片懒加载实现原理](https://juejin.im/entry/594a483061ff4b006c12cea1)

### 2. 响应式图片

响应式图片的优点是浏览器能够根据屏幕大小自动加载合适的图片。

* 通过 picture 实现

<picture>  
 <source srcset="banner\_w1000.jpg" media="(min-width: 801px)">  
 <source srcset="banner\_w800.jpg" media="(max-width: 800px)">  
 <img src="banner\_w800.jpg" alt="">  
</picture>

* 通过 @media 实现

@media (min-width: 769px) {  
 .bg {  
 background-image: url(bg1080.jpg);  
 }  
}  
@media (max-width: 768px) {  
 .bg {  
 background-image: url(bg768.jpg);  
 }  
}

### 3. 调整图片大小

例如，你有一个 1920 \* 1080 大小的图片，用缩略图的方式展示给用户，并且当用户鼠标悬停在上面时才展示全图。如果用户从未真正将鼠标悬停在缩略图上，则浪费了下载图片的时间。

所以，我们可以用两张图片来实行优化。一开始，只加载缩略图，当用户悬停在图片上时，才加载大图。还有一种办法，即对大图进行延迟加载，在所有元素都加载完成后手动更改大图的 src 进行下载。

### 4. 降低图片质量

例如 JPG 格式的图片，100% 的质量和 90% 质量的通常看不出来区别，尤其是用来当背景图的时候。我经常用 PS 切背景图时， 将图片切成 JPG 格式，并且将它压缩到 60% 的质量，基本上看不出来区别。

### 5. 尽可能利用 css3 效果代替图片

有很多图片使用 CSS 效果（渐变、阴影等）就能画出来，这种情况选择 CSS3 效果更好。因为代码大小通常是图片大小的几分之一甚至几十分之一。

## 10. 通过 webpack 按需加载 JavaScript 代码

### 参考资料

* [懒加载](http://webpack.docschina.org/guides/lazy-loading/)

## 11. 减少重绘重排

### 说明

不是所有的动作都会导致重排，例如改变字体颜色，置灰导致重绘。

**记住，重排会导致重绘，重绘不会导致重排！**

重排和重绘这两个操作都是非常昂贵的，因为 JavaScript 引擎线程和 GUI 渲染线程是互斥的，它们同事只能一个在工作。

### 什么操作会导致重排？

* 添加或删除可见的 DOM 元素
* 元素位置改变
* 元素尺寸改变
* 内容改变
* 浏览器窗口尺寸改变

### 如何减少重排重绘？

* 用 JavaScript 修改样式时，最好不要直接写样式，二十替换class来改变样式
* 如果要对 DOM 元素执行一系列操作，可以将 DOM 元素脱离文档流，修改完成后，再将它待会文档。推荐使用隐藏元素（display:none）或文档碎片（DocumentFragement）,都能很好的实现这个方案。

### 浏览器渲染功能

* 1. 解析 HTML 生成 DOM 树
  2. 解析 CSS 生成 CSSOM 规则树
     + ？相关名词：规则树、上下文树、呈现树
  3. 将 DOM 树与 CSSOM 规则树合并在一起生成渲染树
  4. 遍历渲染树开始布局，计算每个节点的位置大小信息
  5. 将渲染树每个节点绘制到屏幕

### 重排

* 解释：当改变 DOM 元素位置或大小时，会导致浏览器重新生成渲染树，这个过程叫重排。

### 重绘

* 解释：当重新生成渲染树后，就要将渲染树每个节点绘制到屏幕，这个过程叫重绘。

## 12. 使用事件委托

### React 事件处理机制自动实现了事件委托，对所有绑定事件进行统一处理，没有绑定到对应的 DOM 元素上，所以直接在 jsx 中绑定到 li 上并不会影响对性能。？//todo 去查 react 的事件处理机制

### 使用事件委托可以节省内存。

## 13. 注意程序的局部性

### 参考资料

* [深入理解计算机系统](https://book.douban.com/subject/26912767/)

### 1. 时间局部性

时间局部性：在一个具有良好时间局部性的程序中，被引用过一次的内存位置很可能在不远的将来被多次引用。

### 2. 空间局部性

空间局部性 ：在一个具有良好空间局部性的程序中，如果一个内存位置被引用了一次，那么程序很可能在不远的将来引用附近的一个内存位置。

## 14. if-else 对比 switch

### 结论：当判断条件数量越来越多时，越倾向使用switch而不是if-else

* 从可读性来说，switch 语句也更好。
* 从使用时机来说，当条件值大于两个的时候，使用 switch 更好。
* 不过 switch 只能用于 case 值为常量的分支结构，而 if-else 更加灵活。

## 15. 查找表

### 当条件语句特别多时，使用 switch 和 id-else 不是最佳的选择，这时不放试一下查找表。查找表可以使用数组和对象来构建。

普通的 switch 语句可以转化为：e.g.：

const results = [  
 result0,  
 result1,  
 result2,  
 result3,  
 result4,  
 result5,  
 result6,  
 result7,  
 result8,  
 result9,  
 result10,  
 result11,  
];  
  
return results[index];

如果条件语句不是数值而是字符串，可以用对象来建立查找表:

const map = {  
 red: result0,  
 green: result1,  
};  
  
return map[color];

## 16. 避免页面卡顿

### 60fps 与设备刷新率

#### 参考资料

* [高性能 JavaScript](https://github.com/woai3c/recommended-books/blob/master/%E5%89%8D%E7%AB%AF/%E9%AB%98%E6%80%A7%E8%83%BDJavaScript.pdf)
* [高效前端：Web 高效编程与优化实践](https://book.douban.com/subject/30170670/)
* [渲染性能](https://developers.google.com/web/fundamentals/performance/rendering)

## 17. 使用 requestAnimationFrame 来实现视觉变化

### 说明

从第 16 点我们可以知道，大多数设备屏幕刷新率为 60 次/秒，也就是说每一帧的平均时间为 16.66 毫秒。在使用 JavaScript 实现动画效果的时候，最好的情况就是每次代码都是在帧的开头开始执行。而保证 JavaScript 在帧开始时运行的唯一方式是使用 requestAnimationFrame。

如果采取 setTimeout 或 setInterval 来实现动画的话，回调函数将在帧中的某个时点运行，可能刚好在末尾，而这可能经常会使我们丢失帧，导致卡顿。

### 参考资料

\*[优化 JavaScript 执行](https://developers.google.com/web/fundamentals/performance/rendering/optimize-javascript-execution?hl=zh-cn)

## 18. 使用 Web Workers

### 说明

Web Worker 使用其他工作线程从而独立于主线程之外，它可以执行任务而不干扰用户界面。一个 worker 可以将消息发送到创建它的 JavaScript 代码, 通过将消息发送到该代码指定的事件处理程序（反之亦然）。

Web Worker 适用于那些处理纯数据，或者与浏览器 UI 无关的长时间运行脚本。

### 参考资料

* [Web Workers](https://developer.mozilla.org/zh-CN/docs/Web/API/Web_Workers_API/Using_web_workers)

## 19. 使用位操作

### 说明

JavaScript 中的数字都是用 IEEE-754 标准以 64 位格式存储。 但是在位操作中，数字被转换为有符号的 32 位格式。即使需要转换，位操作也比其他数字运算和布尔操作快得多。

### 取模

由于偶数的最低位为 0，奇数为 1，所以取模运算可以用位操作来代替。

if (value % 2) {  
 // 奇数  
} else {  
 // 偶数  
}  
// 位操作  
if (value & 1) {  
 // 奇数  
} else {  
 // 偶数  
}

### 取反

~~10.12; // 10  
~~10; // 10  
~~"1.5"; // 1  
~~undefined; // 0  
~~null; // 0

### 位掩码

const a = 1;  
const b = 2;  
const c = 4;  
const options = a | b | c;

通过定义这些选项，可以用按位与操作来判断 a/b/c 是否在 options 中。

// 选项 b 是否在选项中  
if (b & options) {  
 ...  
}

## 20. 不要覆盖原生方法

### 说明

无论你的 JavaScript 代码如何优化，都比不上原生方法。因为原生方法使用低级语言写的（C/C++），并且被编译成机器码，成为浏览器的一部分。当原生方法可用时，尽量使用它们，特别是数学运算和 DOM 操作。

### 原生方法都有什么？

## 21. 降低 CSS 选择器的复杂性

### 1. 浏览器读取选择器，遵循的原则是从选择器的右边到左边读取。

#### 说明

看个示例

#block .text p {  
 color: red;  
}

* 1. 查找所有 P 元素。
  2. 查找结果 1 中的元素是否有类名为 text 的父元素
  3. 查找结果 2 中的元素是否有 id 为 block 的父元素
* 浏览器如何读取 css 选择器？

### 2. css 选择器优先级

内联 > ID选择器 > 类选择器 > 标签选择器

#### 参考资料

* [CSS selector performance](https://ecss.io/appendix1.html)
* [Optimizing CSS: ID Selectors and Other Myths](https://www.sitepoint.com/optimizing-css-id-selectors-and-other-myths/)
  1. 选择器越短越好
  2. 尽量使用高优先级的选择器，例如 id 和类选择器
  3. 避免使用通配符\*
* 总结：css 选择器没有优化的必要，因为最慢和最快的选择器性能差别非常小。

## 22. 使用 flexbox 布局而不是较早的布局模型

### 参考

* [使用 flexbox 而不是较早的布局模型](https://developers.google.com/web/fundamentals/performance/rendering/avoid-large-complex-layouts-and-layout-thrashing?hl=zh-cn)

## 23. 使用 transform 和 opacity 属性更改来实现动画

### 说明

在 CSS 中，transforms 和 opacity 这两个属性更改不会触发重排与重绘，它们是可以由合成器（composite）单独处理的属性。

### 参考资料

* [使用 transform 和 opacity 属性更改来实现动画](https://developers.google.com/web/fundamentals/performance/rendering/stick-to-compositor-only-properties-and-manage-layer-count?hl=zh-cn)

## 24. 合理使用规则，避免过度优化

### 说明

性能优化主要分为两类：

* 1. 加载时优化
  2. 运行时优化

上述 23 条建议中，属于加载时优化的是前面 10 条建议，属于运行时优化的是后面 13 条建议。通常来说，没有必要 23 条性能优化规则都用上，根据网站用户群体来做针对性的调整是最好的，节省精力，节省时间。

在解决问题之前，得先找出问题，否则无从下手。所以在做性能优化之前，最好先调查一下网站的加载性能和运行性能。

### 1. 检查加载性能

### 2. 检查运行性能

#### 参考资料

* [performance.timing.navigationStart](https://developer.mozilla.org/zh-CN/docs/Web/API/PerformanceTiming/navigationStart)

*XMind: ZEN - Trial Version*