# 性能优化

**前端需要加载一个大体积的文件时， 一般有哪些优化思路【热度: 594】**

**一、文件压缩**

**服务器端压缩：**

在服务器上配置文件压缩功能，如使用 Gzip 或 Brotli 压缩算法对文件进行压缩后再传输。这样可以显著减少文件的大小，降低传输时间。

例如，在 Nginx 服务器中，可以通过配置开启 Gzip 压缩：

gzip on;

gzip\_comp\_level 6;

gzip\_types text/plain text/css application/javascript application/json image/svg+xml;

**客户端解压缩：**

现代浏览器通常支持对 Gzip 和 Brotli 压缩的文件进行自动解压缩。当浏览器接收到压缩后的文件时，会自动解压缩并使用。

无需额外的客户端代码，浏览器会自动处理压缩文件的解压缩过程，提高文件加载速度。

**二、文件分割与懒加载**

**文件分割：**

将大体积文件分割成多个较小的文件。例如，对于一个大型的 JavaScript 库，可以将其拆分成多个模块，根据需要逐步加载。

这样可以避免一次性加载整个大文件，减少初始加载时间。

例如，使用 Webpack 等构建工具可以将代码分割成多个 chunk，根据路由或特定条件进行加载。

**懒加载：**

对于不是立即需要的文件或资源，可以采用懒加载的方式。当用户实际需要使用该资源时，再进行加载。

例如，对于图片、视频等资源，可以在用户滚动到可视区域时再进行加载，避免在页面初始加载时加载所有资源。

对于 JavaScript 模块，可以使用动态导入（dynamic import）的方式实现懒加载：

const loadModule = async () => {

const module = await import("./largeModule.js");

// 使用加载的模块

};

**三、缓存策略**

**浏览器缓存：**

设置合理的缓存策略，让浏览器缓存已经加载过的文件。这样，当用户再次访问时，可以直接从缓存中读取文件，而无需再次从服务器下载。

可以通过设置 HTTP 响应头来控制缓存，例如：

location / {

add\_header Cache-Control "max-age=3600";

}

上述配置将设置文件的缓存时间为 1 小时。

**缓存更新机制：**

当文件内容发生变化时，需要确保浏览器能够获取到最新的版本。可以通过在文件名中添加版本号或哈希值来实现缓存更新。

例如，将文件名改为largeFile\_v1.2.js或largeFile\_abc123.js，当文件内容变化时，更新版本号或哈希值，浏览器会认为这是一个新的文件并进行下载。

**四、优化加载顺序**

**关键资源优先加载：**

确定哪些资源是页面加载的关键资源，优先加载这些资源。对于大体积文件，如果不是关键资源，可以延迟加载。

例如，对于一个图片库应用，先加载页面的基本结构和导航部分，图片可以在用户交互时再进行加载。

**异步加载：**

使用异步加载的方式加载大体积文件。例如，对于 JavaScript 文件，可以使用<script async>标签或动态创建<script>标签并插入到页面中进行异步加载。

<script async src="largeScript.js"></script>

这样可以避免阻塞页面的渲染，提高用户体验。

**五、CDN 加速**

**使用内容分发网络（CDN）：**

将大体积文件托管在 CDN 上，利用 CDN 的分布式节点，可以让用户从离自己最近的节点获取文件，减少网络延迟，提高加载速度。

例如，将图片、视频、静态文件等托管在 CDN 上，通过 CDN 的 URL 进行访问。

**CDN 缓存：**

CDN 通常会对文件进行缓存，进一步提高文件的加载速度。当文件内容发生变化时，需要及时更新 CDN 上的缓存。

可以通过设置 CDN 的缓存策略或使用版本号等方式来管理 CDN 缓存。

## web 系统里面， 如何对图片进行优化？【热度: 789】

关键词：图片优化

图片作为网页和移动应用中不可或缺的元素之一，对于用户体验和网站性能都有着重要的影响。

加载速度是用户体验的关键因素之一，而大尺寸的图片会增加网页加载时间，导致用户等待时间过长，从而影响用户的满意度和留存率。通过优化图片，我们可以显著减少页面加载时间，提供更快速流畅的使用体验。

图片优化是提升用户体验、提高网站性能、减少流量消耗和增加搜索引擎曝光度的关键因素。为了提供更出色的用户体验，同时也提升网站的性能。总结了一下通用的图片优化首手段。

### 1. 选择合适的图片格式

以下是对常用的图片格式jpg、png和webp进行深度对比的表格：

| 特性 | JPG | PNG | WebP |
| --- | --- | --- | --- |
| 压缩算法 | 有损压缩 | 无损压缩 | 有损压缩 |
| 透明度 | 不支持透明度 | 支持透明度 | 支持透明度 |
| 图片质量 | 可调整质量 | 无法调整质量 | 可调整质量 |
| 文件大小 | 相对较小 | 相对较大 | 相对较小 |
| 浏览器支持 | 支持在所有主流浏览器上显示 | 支持在所有主流浏览器上显示 | 部分浏览器支持 |
| 动画支持 | 不支持动画 | 不支持动画 | 支持动画 |
| 兼容性 | 兼容性较好 | 兼容性较好 | 兼容性较差 |

请注意，这个表格只是对这些格式的一般特征进行了总结，并不代表所有情况。实际情况可能因图像内容、压缩设置和浏览器支持等因素而有所不同。因此，在选择图像格式时，您应根据具体要求和应用场景进行评估和选择。

### 2. 图片压缩

主要介绍 webpack 对图片进行压缩，可以使用以下步骤：

1.安装依赖：首先，确保你已经在项目中安装了webpack和相关的loader。可以使用以下命令安装所需的loader：

npm install --save-dev file-loader image-webpack-loader

2.配置Webpack：在Webpack的配置文件中进行相关配置。以下是一个简单的示例：

const path = require('path');

module.exports = {

entry: 'src/index.js',

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist')

},

module: {

rules: [

{

test: /\.(png|jpe?g|gif)$/i,

use: [

{

loader: 'file-loader',

options: {

name: '[name].[ext]',

outputPath: 'images/'

}

},

{

loader: 'image-webpack-loader',

options: {

mozjpeg: {

progressive: true,

quality: 65

},

// optipng.enabled: false will disable optipng

optipng: {

enabled: false,

},

pngquant: {

quality: [0.65, 0.90],

speed: 4

},

gifsicle: {

interlaced: false,

},

// the webp option will enable WEBP

webp: {

quality: 75

}

}

}

]

}

]

}

};

上述配置中，我们使用file-loader将图片复制到输出目录，并使用image-webpack-loader对图片进行压缩和优化。

3.运行Webpack：现在，当你运行Webpack时，它将自动使用image-webpack-loader对匹配到的图片进行压缩和优化。压缩后的图片将被复制到输出目录中。

### 3. 雪碧图

Web图片优化的雪碧图（CSS Sprites）是一种将多个小图片合并为一个大图片的技术。通过将多个小图片合并成一张大图片，可以减少浏览器发送的请求次数，从而提高页面加载速度。

雪碧图的原理是通过CSS的background-image和background-position属性，将所需的小图片显示在指定的位置上。这样，只需加载一张大图，就可以显示多个小图片，减少了网络请求的数量，提高了页面加载速度。

听上去好像很麻烦，**实际上可以使用 webpack 插件 webpack-spritesmith 完成自动化处理雪碧图合成**，我们在使用过程中正常使用即可。

以下是使用webpack-spritesmith插件来自动处理雪碧图的步骤：

1.安装插件：使用npm或yarn安装webpack-spritesmith插件。

npm install webpack-spritesmith --save-dev

2.配置Webpack：在Webpack配置文件中，引入webpack-spritesmith插件，并配置相应的选项。

const SpritesmithPlugin = require('webpack-spritesmith');

module.exports = {

// ...其他配置

plugins: [

new SpritesmithPlugin({

src: {

cwd: path.resolve(\_\_dirname, 'path/to/sprites'), // 需要合并的小图片所在的目录

glob: '\*.png' // 小图片的文件名格式

},

target: {

image: path.resolve(\_\_dirname, 'path/to/output/sprite.png'), // 生成的雪碧图的路径和文件名

css: path.resolve(\_\_dirname, 'path/to/output/sprite.css') // 生成的CSS样式表的路径和文件名

},

apiOptions: {

cssImageRef: 'path/to/output/sprite.png' // CSS样式表中引用雪碧图的路径

}

})

]

}

3.使用雪碧图：在HTML中，使用生成的CSS样式类来显示相应的小图片。Webpack会自动处理雪碧图的合并和CSS样式的生成。

然后，你可以按照以下方法在CSS中引用雪碧图：

CSS方式：

div {

background: url(path/to/output/sprite.png) no-repeat;

}

.icon-facebook {

/\* 设置小图标在雪碧图中的位置和大小 \*/

width: 32px;

height: 32px;

background-position: 0 0; /\* 该小图标在雪碧图中的位置 \*/

}

.icon-twitter {

width: 32px;

height: 32px;

background-position: -32px 0; /\* 该小图标在雪碧图中的位置 \*/

}

.icon-instagram {

width: 32px;

height: 32px;

background-position: -64px 0; /\* 该小图标在雪碧图中的位置 \*/

}

在HTML中，你可以像下面这样使用对应的CSS类来显示相应的小图标：

<div class="icon icon-facebook"></div>

<div class="icon icon-twitter"></div>

<div class="icon icon-instagram"></div>

这样，Webpack会根据配置自动处理雪碧图，并生成对应的雪碧图和CSS样式表。CSS中的background属性会引用生成的雪碧图，并通过background-position来指定显示的小图标在雪碧图中的位置。

确保在CSS中指定了每个小图标在雪碧图中的位置和大小，以便正确显示。

使用Webpack自动处理雪碧图可以简化开发流程，并且可以根据需要自定义配置。webpack-spritesmith是一个常用的Webpack插件，可以帮助自动处理雪碧图。

### 4. 图标类型资源推荐使用 iconfont

如果你有很多图标类型的图片资源，并且想使用iconfont来处理这些资源，可以按照以下步骤进行处理：

* 获取图标资源：首先，你需要获取你想要的图标资源。你可以从iconfont网站或其他图标库中选择和下载符合需求的图标。这个没有啥好说的， 直接推荐: https://www.iconfont.cn/
* 生成字体文件：接下来，你需要将这些图标转换成字体文件。你可以使用iconfont提供的在线转换工具，将图标文件上传并生成字体文件（包括.ttf、.eot、.woff和.svg格式）。
* 引入字体文件：将生成的字体文件下载到本地，并在你的项目中引入。通常，你需要在CSS文件中通过@font-face规则引入字体文件，并为字体定义一个唯一的名称。
* 使用图标：一旦字体文件引入成功，你可以在CSS中通过设置content属性来使用图标。每个图标都会有一个对应的Unicode代码，你可以在iconfont提供的网站或字体文件中找到对应图标的Unicode代码，并通过设置content属性的值为该Unicode代码来使用图标。

### 5. 使用 base64 格式

实际开发过程中， 为何会考虑 base64 ？

使用Base64图片的优势有以下几点：

* 减少HTTP请求数量：通常情况下，每个网页都需要加载多张图片，因此会发送多个HTTP请求来获取这些图片文件。使用Base64图片可以将图片数据嵌入到CSS或HTML文件中，减少了对服务器的请求次数，从而提高网页加载速度。
* 减少图片文件的大小：Base64是一种编码方式，可以将二进制数据转换成文本字符串。通过使用Base64，可以将图片文件转换成文本字符串，并将其嵌入到CSS或HTML文件中。相比于直接引用图片文件，Base64编码的字符串通常会更小，因此可以减少图片文件的大小，从而减少了网页的总体积，加快了网页加载速度。
* 简化部署和维护：将图片数据嵌入到CSS或HTML文件中，可以减少文件的数量和复杂性，使得部署和维护变得更加简单和方便。此外，也不需要处理图片文件的路径和引用相关的问题。
* 实现一些特殊效果：通过Base64图片，可以实现一些特殊的效果，例如页面背景渐变、图标的使用等。这样可以避免使用额外的图片文件，简化了开发过程。

上面虽然说列出了挺多优势，但是**劣势**也是很明显：

* 增加了文本文件的体积：因为Base64编码将二进制数据转换成文本字符串，所以会增加CSS或HTML文件的体积。在图片较大或数量较多时，这可能会导致文件变得庞大，从而导致网页加载速度变慢。
* 缓存问题：由于Base64图片被嵌入到了CSS或HTML文件中，如果图片内容有更新，那么整个文件都需要重新加载，而无法使用缓存。相比于独立的图片文件，Base64图片对缓存的利用效率较低。

使用Base64图片在一些特定的场景下可以提供一些优势，但也需要权衡其带来的一些缺点。在实际开发中，可以根据具体的需求和情况，选择是否使用Base64图片。所以**建议复用性很强, 变更率较低，且小于10KB的图片文件，可以考虑 base64**

#### 如何使用？

有要介绍一下 webpack 插件了： url-loader 或 file-loader

要使用Webpack将图片自动转换为Base64编码，您需要执行以下步骤：

1.安装依赖：首先，确保您已经安装了url-loader或file-loader，它们是Webpack的两个常用的加载器。

npm install url-loader --save-dev

2.配置Webpack：在Webpack的配置文件中，添加对图片文件的处理规则。您可以在module.rules数组中添加一个新的规则，以匹配图片文件的后缀。

module.exports = {

// ...

module: {

rules: [

// ...

{

test: /\.(png|jpe?g|gif)$/i,

use: [

{

loader: 'url-loader',

options: {

limit: 8192, // 设置图片大小的阈值，小于该值的图片会被转为Base64

outputPath: 'images', // 输出路径

publicPath: 'images', // 资源路径

},

},

],

},

],

},

// ...

};

在上面的示例中，配置了一个处理png、jpeg、jpg和gif格式图片的规则。使用url-loader加载器，并设置了一些选项，例如limit限制了图片大小的阈值，小于该值的图片将会被转换为Base64编码。

3.在代码中引用图片：在您的代码中，可以像引用普通图片一样引用图片文件，Webpack会根据配置自动将其转换为Base64编码。

import imgSrc from './path/to/image.png';

const imgElement = document.createElement('img');

imgElement.src = imgSrc;

document.body.appendChild(imgElement);

4.构建项目：最后，使用Webpack构建项目，它会根据配置自动将符合规则的图片文件转换为Base64编码，并将其嵌入到生成的输出文件中。

npx webpack

这样，Webpack就会自动将图片转换为Base64编码，并将其嵌入到生成的输出文件中。请注意，在使用Base64图片时，需要权衡文件大小和性能，适度使用Base64编码，避免过大的文件导致网页加载变慢。

### 6. 使用 CDN 加载图片

CND 加载图片优势非常明显：

* 加速网页加载速度：CDN通过将图片资源分布在全球的多个节点上，使用户能从离自己最近的节点获取资源，从而大大减少了网络延迟和加载时间。这可以提高网页的加载速度和用户体验。
* 减轻服务器负载：CDN充当了一个缓冲层，当用户请求图片资源时，CDN会将图片资源从源服务器获取并缓存在节点中，下次再有用户请求同一资源时，CDN会直接从节点返回，减少了对源服务器的请求，分担了服务器的负载。
* 提高并发性能：CDN节点分布在不同地区，用户请求图片资源时可以从离他们最近的节点获取，这可以减少网络拥塞和并发请求，提高了并发性能。
* 节省带宽成本：CDN的节点之间会自动选择最优路径，有效利用了带宽资源，减少了数据传输的成本，尤其在大量图片资源请求时，能够带来显著的成本节省。
* 提供高可用性：CDN通过分布式存储和负载均衡技术，提供了高可用性和容错能力。即使某个节点或源服务器发生故障，CDN会自动切换到其他可用节点，确保用户能够正常访问图片资源。

总之，使用CDN加载图片可以提高网页加载速度、降低服务器负载、提高并发性能、节省带宽成本，并提供高可用性，从而改善用户体验和网站性能。

### 7. 图片懒加载

图片懒加载是一种在网站或应用中延迟加载图片的技术。它的主要目的是减少页面的初始加载时间，并提高用户的浏览体验。

* 原理：图片懒加载的原理是只在用户需要时加载图片，而不是在页面初始加载时全部加载。这通常通过将图片的真实地址存储在自定义属性（例如data-src）中，而不是在src属性中。然后，在图片进入浏览器视图时，通过JavaScript动态将data-src的值赋给src属性，触发图片的加载。
* 优势：图片懒加载可以显著减少初始页面的加载时间，特别是当页面中有大量图片时。它使页面加载变得更快，提高了用户的浏览体验。此外，懒加载还可以节省带宽和减轻服务器负载，因为只有当图片进入视图时才会加载。
* 实现方法：图片懒加载可以通过纯JavaScript实现，也可以使用现成的JavaScript库，如LazyLoad.js、Intersection Observer API等。这些库提供了方便的API和配置选项，可以自定义懒加载的行为和效果。
* 最佳实践：在使用图片懒加载时，可以考虑一些最佳实践。例如，设置一个占位符或加载中的动画，以提供更好的用户体验。另外，确保在不支持JavaScript的情况下仍然可用，并为可访问性提供替代文本（alt属性）。此外，对于移动设备，可以考虑使用响应式图片来适应不同的屏幕分辨率。

#### 实现方法

##### 1.Intersection Observer API

Intersection Observer API 是一种用于异步检查文档中元素与视口叠加程度的API。可以将其用于检测图片是否已经进入视口，并根据需要进行相应的处理。

let observer = new IntersectionObserver(function (entries) {

entries.forEach(function (entry) {

if (entry.isIntersecting) {

const lazyImage = entry.target;

lazyImage.src = lazyImage.dataset.src;

observer.unobserve(lazyImage);

}

});

});

const lazyImages = [...document.querySelectorAll(".lazy")];

lazyImages.forEach(function (image) {

observer.observe(image);

});

##### 2.自定义监听器

或者，可以通过自定义监听器来实现懒加载。其中，应该避免在滚动事件处理程序中频繁进行图片加载，因为这可能会影响性能。相反，使用自定义监听器只会在滚动停止时进行图片加载。

function lazyLoad() {

const images = document.querySelectorAll(".lazy");

const scrollTop = window.pageYOffset;

images.forEach((img) => {

if (img.offsetTop < window.innerHeight + scrollTop) {

img.src = img.dataset.src;

img.classList.remove("lazy");

}

});

}

let lazyLoadThrottleTimeout;

document.addEventListener("scroll", function () {

if (lazyLoadThrottleTimeout) {

clearTimeout(lazyLoadThrottleTimeout);

}

lazyLoadThrottleTimeout = setTimeout(lazyLoad, 20);

});

在这个例子中，我们使用了 setTimeout() 函数来延迟图片的加载，以避免在滚动事件的频繁触发中对性能的影响。

无论使用哪种方法，都需要为需要懒加载的图片设置占位符，并将未加载的图片路径保存在 data 属性中，以便在需要时进行加载。这些占位符可以是简单的 div 或样式类，用于预留图片的空间，避免页面布局的混乱。

<!-- 占位符示例 -->

<div class="lazy-placeholder" style="background-color: #ddd;height: 500px;"></div>

<!-- 图片示例 -->

<img class="lazy" data-src="path/to/image.jpg" alt="预览图" />

### 8. 图片预加载

图片预加载是一种在网站或应用中提前加载图片资源的技术。它的主要目的是在用户实际需要加载图片之前，将其提前下载到浏览器缓存中。

图片预加载通常是在页面加载过程中或在特定事件触发前异步加载图片资源。 通过使用 JavaScript，可以在网页DOM元素中创建一个新的Image对象，并将要预加载的图片的URL赋值给该对象的src属性。 浏览器在加载过程中会提前下载这些图片，并将其缓存起来，以备将来使用。

图片预加载可以使用原生JavaScript实现，也可以使用现成的JavaScript库，如Preload.js、LazyLoad.js等。这些库提供了方便的API和配置选项，可以灵活地控制预加载的行为和效果。

实现图片预加载可以使用原生JavaScript或使用专门的JavaScript库。**下面分别介绍两种方式的实现方法：**

#### 1.使用原生JavaScript实现图片预加载：

function preloadImage(url) {

return new Promise(function(resolve, reject) {

var img = new Image();

img.onload = resolve;

img.onerror = reject;

img.src = url;

});

}

// 调用预加载函数

preloadImage('image.jpg')

.then(function() {

console.log('图片加载成功');

// 在此处可以执行加载成功后的操作，例如显示图片等

})

.catch(function() {

console.error('图片加载失败');

// 在此处可以执行加载失败后的操作，例如显示错误信息等

});

在上述代码中，我们定义了一个preloadImage函数，它使用Image对象来加载图片资源。通过onload事件和onerror事件来监听图片加载完成和加载错误的情况，并使用Promise对象进行异步处理。

#### 2.使用JavaScript库实现图片预加载：

使用JavaScript库可以更简便地实现图片预加载，并提供更多的配置选项和功能。以下以Preload.js库为例进行说明：

首先，在HTML文件中引入Preload.js库：

<script src="preload.js"></script>

然后，在JavaScript代码中使用Preload.js库来进行图片预加载：

var preload = new createjs.LoadQueue();

preload.on("complete", handleComplete);

preload.on("error", handleError);

preload.loadFile('image.jpg');

function handleComplete() {

console.log('图片加载成功');

// 在此处可以执行加载成功后的操作，例如显示图片等

}

function handleError() {

console.error('图片加载失败');

// 在此处可以执行加载失败后的操作，例如显示错误信息等

}

在上述代码中，我们首先创建一个LoadQueue对象，并使用on方法来监听加载完成和加载错误的事件。然后使用loadFile方法来指定要预加载的图片资源的URL。

当图片加载完成时，handleComplete函数会被调用，我们可以在此处执行加载成功后的操作。当图片加载错误时，handleError函数会被调用，我们可以在此处执行加载失败后的操作。

以上是两种常用的实现图片预加载的方法，根据具体需求和项目情况选择合适的方式来实现图片预加载。

### 9. 响应式加载图片

要在不同分辨率的设备上显示不同尺寸的图片，你可以使用<picture>元素和<source>元素来实现响应式图片。以下是一个示例：

<picture>

<source media="(min-width: 1200px)" srcset="large-image.jpg">

<source media="(min-width: 768px)" srcset="medium-image.jpg">

<source srcset="small-image.jpg">

<img src="fallback-image.jpg" alt="Fallback Image">

</picture>

在上面的示例中，<picture>元素内部有多个<source>元素，每个<source>元素通过srcset属性指定了对应分辨率下的图片链接。media属性可以用来指定在哪个分辨率下应用对应的图片。如果没有任何<source>元素匹配当前设备的分辨率，那么就会使用<img>元素的src属性指定的图片链接。

可以根据不同分辨率的设备，提供不同尺寸和质量的图片，以优化用户的视觉体验和页面加载性能。

### 10. 渐进式加载图片

实现渐进式加载的主要思想是先加载一张较低分辨率的模糊图片，然后逐步加载更高分辨率的图片。

下面是实现渐进式加载图片的一般步骤：

1. 创建一张模糊的低分辨率图片。可以使用图片处理工具将原始图片进行模糊处理，或者使用低分辨率的缩略图作为初始图片。
2. 使用<img>标签将低分辨率的图片设置为src属性。这将立即加载并显示这张低分辨率的图片。
3. 在加载低分辨率图片时，同时加载高分辨率的原始图片。可以将高分辨率图片的URL设置为data-src等自定义属性，或者使用JavaScript动态加载高清图片。
4. 使用JavaScript监听图片的加载事件，在高分辨率图片加载完成后，将其替换低分辨率图片的src属性，以实现渐进式加载的效果。

下面是一个示例代码，演示了如何实现渐进式加载图片：

<!-- HTML -->

<img src="blur-image.jpg" data-src="high-res-image.jpg" alt="Image">

<script>

// JavaScript

const image = document.querySelector('img');

// 监听高分辨率图片加载完成事件

image.addEventListener('load', () => {

// 替换低分辨率图片的src属性

image.src = image.dataset.src;

});

</script>

在上面的示例中，一开始会显示一张模糊的低分辨率图片，然后在高分辨率图片加载完成后，将其替换为高分辨率图片，实现了渐进式加载的效果。

渐进式加载图片可以减少用户等待时间，提供更好的用户体验。然而，需要注意的是，为了实现渐进式加载，需要额外加载高分辨率的图片，这可能会增加页面加载时间和网络带宽消耗。因此，开发者需要在性能和用户体验之间进行权衡，并根据实际情况进行选择和优化。

### 页面加载速度提升（性能优化）应该从哪些方向来思考？【热度: 1,099】

关键词：性能提升、加载优化

页面加载优化

* 页面启动
* service worker 缓存重要的静态资源
* 页面保活
* 资源加载
* 网络连接
* NDS
* 减少 NDS 解析
* NDA 预解析
* HTTP
* 开启 HTTP2 多路复用
* 优化核心请求链路
* HTML 加载
* 内容优化
* 代码压缩
* 代码精简(tailwindcss)
* 服务端渲染 SSG
* 流程优化
* 服务端渲染 SSR
* 流式渲染
* 预渲染
* 静态资源加载
* 内容优化
* JS、CSS 代码压缩
* 均衡资源包体积：复用代码抽离为一份资源打包、同时开启
* 精简代码
* 雪碧图
* 动态图片降质量
* 动态 polyfill (根据浏览器的支持情况，动态加载需要的 polyfill（填充）脚本)
* 不常变的资源单独打包
* 根据浏览器版本打包， 高版本浏览器， 直接使用 es6 作为输出文件
* 流程优化
* 配置前端缓存: 资源、请求
* 使用 CDN
* CDN 优化
* 协调资源加载优先级
* 动态资源转静态 CDN 链接加载(例如大图片等)
* 静态资源使用 service worker 离线存储
* 非首屏资源懒加载
* 资源和业务请求预加载
* 微前端加载应用
* 微组件加载核心模块资源
* 代码执行
* 减少执行
* 减少重复渲染
* 大体量计算场景， 尽量使用缓存函数
* 使用防抖节流
* 速度提升
* 使用 worker 多线程加速
* 充分利用异步请求的线下之间来进行核心代码的加载或者执行
* wasm 处理大量计算场景
* 渲染高耗时场景， 迁移到 canvas 、虚拟 dom 等
* 动态渲染：动态渲染可视区内容， 例如图片懒加载等；
* 流程优化
* 非首屏模块， 延迟加载与渲染
* longtask 任务拆分执行
* 利用请求闲暇时间， 请求后续页面资源
* 数据获取
* 内容优化
* 减少请求、合并请求、BFF
* 首屏数据使用模板注入到前端应用
* 流程优化
* 数据预请求
* 常量数据缓存
* 非首屏请求，延迟到首屏加载完成之后请求
* 请求并行
* 渲染相关
* 虚拟列表
* 延迟渲染
* 减少重绘重排
* 图片预加载到内存

## SPA首屏加载速度慢的怎么解决【热度: 868】

**关键词：SPA首屏、加快首屏加载**

统计首屏时间

可以参考下面的文档：

**[性能] 衡量页面性能的指标有哪些？【热度: 1,045】 #515**

**[性能] 常见性能指标获取方式？【热度: 954】 #516**

### 影响首屏可能得因素

* 网络延时问题
* 资源文件体积是否过大
* 资源是否重复发送请求去加载了
* 加载脚本的时候，渲染内容堵塞了

### 解决方案

* 减小入口文件积
* 静态资源本地缓存
* UI框架按需加载
* 图片资源的压缩
* 组件重复打包
* 开启 GZip 压缩
* 使用 SSR
* 启用 CDN 加速

## [性能] 常见性能指标获取方式？【热度: 954】

关键词：web性能指标获取

### 常见性能指标获取方式

相关性能指标问题， 可以看这个文章：#515

### 指标所反映的用户体验

下表概述了我们的性能指标如何对应到我们的问题之上：

**开始了吗？**

首次绘制、首次内容绘制 First Paint (FP) / First Contentful Paint (FCP)

**有用吗？**

首次有效绘制、主要元素时间点 First Meaningful Paint (FMP) / Hero Element Timing

**能用吗？**

可交互时间点 Time to Interactive (TTI)

**好用吗？**

慢会话 Long Tasks (从技术上来讲应该是：没有慢会话)

### 页面何时开始渲染 - FP & FCP

这两个指标，我们可以通过 performance.getEntry、performance.getEntriesByName、performanceObserver 来获取。

performance.getEntries().filter(item => item.name === 'first-paint')[0]; // 获取 FP 时间

performance.getEntries().filter(item => item.name === 'first-contentful-paint')[0]; // 获取 FCP 时间

performance.getEntriesByName('first-paint'); // 获取 FP 时间

performance.getEntriesByName('first-contentful-paint'); // 获取 FCP 时间

// 也可以通过 performanceObserver 的方式获取

var observer = new PerformanceObserver(function(list, obj) {

var entries = list.getEntries();

entries.forEach(item => {

if (item.name === 'first-paint') {

...

}

if (item.name === 'first-contentful-paint') {

...

}

})

});

observer.observe({type: 'paint'});

### 页面何时渲染主要内容 - FMP & SI & LCP

FMP, 是一个已经废弃的性能指标。在实践过程中，由于 FMP 对页面加载的微小差异过于敏感，经常会出现结果不一致的情况。此外，该指标的定义依赖于特定于浏览器的实现细节，这意味着它不能标准化，也不能在所有 Web 浏览器中实现。目前，官方并没有提供有效的获取 FMP 的接口，因此性能分析的时候不再使用这个指标。

SI 和 FMP 一样，官方也没有提供有效的获取接口，只能通过 lighthouse 面板来查看，不作为 Sentry 等工具做性能分析的指标。

LCP，和 FMP 类似，但只聚焦页面首次加载时最大元素的绘制时间点，计算相对简单一些。通过 performanceObserver 这个接口，我们可以获取到 LCP 指标数据。

new PerformanceObserver((entryList) => {

for (const entry of entryList.getEntries()) {

console.log('LCP candidate:', entry.startTime, entry);

}

}).observe({type: 'largest-contentful-paint', buffered: true});

### 何时可以交互 - TTI & TBT

TTI, time to ineractive, 可交互时间， lighthouse 面板中的六大指标之一, 用于测量页面从开始加载到主要资源完成渲染，并能够快速、可靠地响应用户输入所需的时间, 值越小约好。

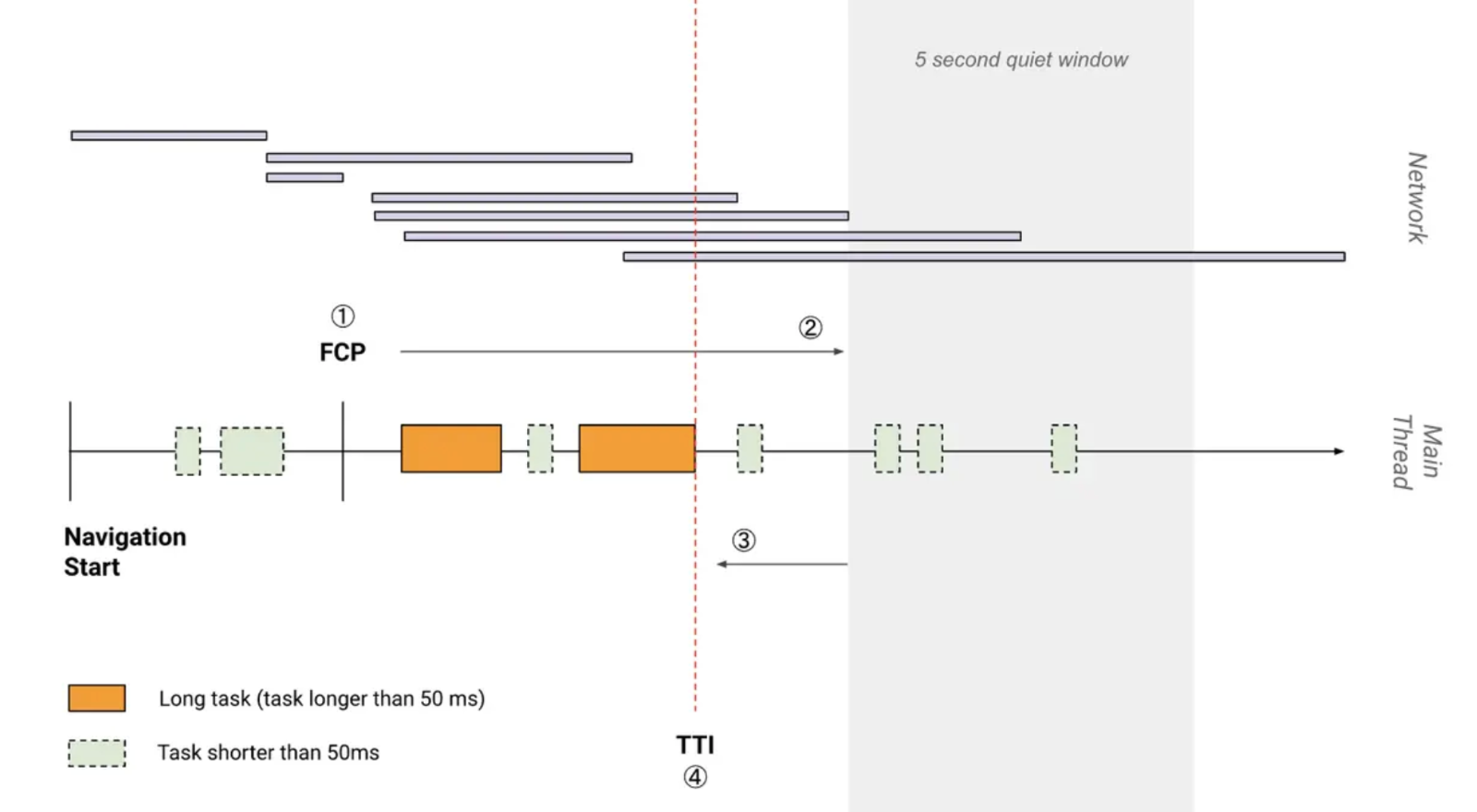
官方资料: [TTI](https://web.dev/i18n/zh/tti/) 。

和 FMP、SI 一样，官方并没有提供获取 TTI 的有效接口，只能通过 lighthouse 面板来查看。

#### 计算方式人如下:

1. 先进行 First Contentful Paint 首次内容绘制；
2. 沿时间轴正向搜索时长至少为 5 秒的安静窗口，其中，安静窗口的定义为：没有长任务且不超过 2 个正在处理的网络请求;
3. 沿时间轴反向搜索安静窗口之前的最后一个长任务，如果没有找到长任务，则在 FCP 步骤停止执行。
4. TTI 是安静窗口之前最后一个长任务的结束时间（如果没有找到长任务，则与 FCP 值相同）。

理解计算过程如下图：



TTI 表示的是完全可交互的时间， 每个 web 系统， 对 TTI 时间定义可能并不一定相同， 上面只是提供一个计算较为通用的 TTI 的一个方式。

### 交互是否有延迟 - FID & MPFID & Long Task

FID 和 MPFID 可用来衡量用户首次交互延迟的情况，Long Task 用来衡量用户在使用应用的过程中遇到的延迟、阻塞情况。

FID，first input delay, 首次输入延迟，测量从用户第一次与页面交互（例如当他们单击链接、点按按钮或使用由 JavaScript 驱动的自定义控件）直到浏览器对交互作出响应，并实际能够开始处理事件处理程序所经过的时间。官方资料: FID。

FID 指标的值越小约好。通过 performanceObserver，我们可以获取到 FID 指标数据。

new PerformanceObserver((entryList) => {

for (const entry of entryList.getEntries()) {

const delay = entry.processingStart - entry.startTime;

console.log('FID candidate:', delay, entry);

}

}).observe({type: 'first-input', buffered: true});

MPFID, Max Potential First Input Delay，最大潜在首次输入延迟，用于测量用户可能遇到的最坏情况的首次输入延迟。和 FMP 一样，这个指标已经被废弃不再使用。

Long Task，衡量用户在使用过程中遇到的交互延迟、阻塞情况。这个指标，可以告诉我们哪些任务执行耗费了 50ms 或更多时间。

new PerformanceObserver(function(list) {

var perfEntries = list.getEntries();

for (var i = 0; i < perfEntries.length; i++) {

...

}

}).observe({ type: 'longtask'});

#### 页面视觉是否有稳定 - CLS

CLS, Cumulative Layout Shift, 累积布局偏移，用于测量整个页面生命周期内发生的所有意外布局偏移中最大一连串的布局偏移情况。官方资料: CLS。

CLS, 值越小，表示页面视觉越稳定。通过 performanceObserver，我们可以获取到 CLS 指标数据。

new PerformanceObserver(function(list) {

var perfEntries = list.getEntries();

for (var i = 0; i < perfEntries.length; i++) {

...

}

}).observe({type: 'layout-shift', buffered: true});

## [性能] 衡量页面性能的指标有哪些？【热度: 1,045】

**关键词：web性能指标**

### 性能的核心问题

* 什么样的性能指标最能度量人的感觉？
* 怎样才能从我们的真实用户中获取这些指标？
* 如何用我们所获取的指标来确定一个页面表现得是否「快」？
* 当我们得知用户所感知的真实性能表现后，我们应该如何做才能避免重蹈覆辙，并在未来提高性能表现？

### 以用户为中心的性能指标

**开始了吗？**

页面开始加载了吗？得到了服务端的回应吗？

**有用吗？**

有足够用户期望看到的内容被渲染出来了吗？

**能用吗？**

用户能够与我们的页面交互了吗？还是依然在加载？

**好用吗？**

交互是否流畅、自然、没有延迟与其他的干扰？

### 首次绘制（First Paint）和首次内容绘制（First Contentful Paint）

首次绘制（FP）和首次内容绘制（FCP）。在浏览器导航并渲染出像素点后，这些指标点立即被标记。 这些点对于用户而言十分重要，因为它回答了我们的第一个问题问题：开始了吗？

FP与FCP的主要区别在于，FP标记浏览器所渲染的任何与导航前内容不同的点，而FCP所标记的是来自于DOM中的内容，可能是文本、图片、SVG，甚至是canvas元素。

### 首次有效绘制（First Meaningful Pain）和主要元素时间点（Hero Element Timing）

首次有效绘制（FMP）回答了我们的问题：有用吗？对于现存的所有网页而言，我们不能去清晰地界定哪些元素的加载是「有用」的（因此目前尚无规范），

但是对于开发者他们自己而言，他们很容易知道页面的哪些部分对于用户而言是最为有用的。

### 可交互时间（TTI）

可交互时间（TTI）标记了你的页面已经呈现了画面，并且能够响应用户输入的时间点。页面不能响应用户输入有以下常见的原因：

将被JavaScript所操作的元素还未被加载出来；

* 一些慢会话阻塞了浏览器的主线程（如我们在上一部分所描述的）
* TTI所记录实际上是页面的JavaScript完成了初始化，主线程处于空闲的时间点。

### long tasks

浏览器像是单线程的。 某些情况下，一些任务将可能会花费很长的时间来执行，如果这种情况发生了，主线程阻塞，剩下的任务只能在队列中等待。

用户所感知到的可能是输入的延迟，或者是哐当一下全部出现。这些是当今网页糟糕体验的主要来源。

Long Tasks API认为任何超过50毫秒的任务都可能存在潜在的问题，并将这些任务揭露给开发者。既然能够满足50毫秒内完成任务，也就能够符合RAIL在100毫秒内相应用户输入的要求。

### 指标所反映的用户体验

下表概述了我们的性能指标如何对应到我们的问题之上：

**开始了吗？**

首次绘制、首次内容绘制 First Paint (FP) / First Contentful Paint (FCP)

**有用吗？**

首次有效绘制、主要元素时间点 First Meaningful Paint (FMP) / Hero Element Timing

**能用吗？**

可交互时间点 Time to Interactive (TTI)

**好用吗？**

慢会话 Long Tasks (从技术上来讲应该是：没有慢会话)

### 获取指标

主要依赖浏览器提供的 api

* [PerformanceObserver](https://developer.mozilla.org/zh-CN/docs/Web/API/PerformanceObserver)
* [PerformanceEntry](https://developer.mozilla.org/zh-CN/docs/Web/API/PerformanceEntry)
* [DOMHighResTimeStamp](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMHighResTimeStamp)

#### PerformanceObserver 使用示范

要使用 PerformanceObserver，首先需要创建一个 PerformanceObserver 实例，并指定一个回调函数作为参数。回调函数将在性能事件触发时被调用。然后，通过 PerformanceObserver 的 observe() 方法去监听所关注的性能事件类型。

以下是一个使用 PerformanceObserver 的示例代码：

// 创建回调函数

function performanceCallback(list, observer) {

list.getEntries().forEach(entry => {

console.log(entry.name + "：" + entry.startTime);

});

}

// 创建 PerformanceObserver 实例

const observer = new PerformanceObserver(performanceCallback);

// 监听性能事件类型

observer.observe({ entryTypes: ["measure", "paint"] });

在上面的代码中，定义了一个名为 performanceCallback 的回调函数，它接收两个参数：list 和 observer。list 参数是一个 PerformanceEntryList 对象，包含了所有触发的性能事件，可以通过 getEntries() 方法获取详细信息。observer 参数表示 PerformanceObserver 实例本身。

然后，通过 new PerformanceObserver(performanceCallback) 创建了一个 PerformanceObserver 实例，并将 performanceCallback 作为回调函数传递进去。

最后，通过 observer.observe({ entryTypes: ["measure", "paint"] }) 方法，指定了要监听的性能事件类型，这里监听了 "measure" 和 "paint" 两种类型的事件。

当指定的性能事件类型发生时，回调函数将被调用，并传递触发事件的 PerformanceEntry 对象作为参数。开发者可以在回调函数中进一步处理和分析这些对象，以获取性能数据并进行优化。

需要注意的是，观察者模式是异步的，因此回调函数可能不会立即执行。另外，一旦创建了 PerformanceObserver 实例，需要调用其 disconnect() 方法来停止监听性能事件，避免内存泄漏。

以上是 PerformanceObserver 的基本用法，开发者可以根据实际需求和业务场景来灵活运用。

#### PerformanceObserver 如何统计FP、FCP

要使用 PerformanceObserver 统计 FP（First Paint）和 FCP（First Contentful Paint），可以按照以下步骤进行：

1.创建 PerformanceObserver 实例，并指定一个回调函数作为参数。

const observer = new PerformanceObserver((list) => {

const entries = list.getEntries();

entries.forEach((entry) => {

if (entry.name === 'first-paint') {

console.log('FP:', entry.startTime);

} else if (entry.name === 'first-contentful-paint') {

console.log('FCP:', entry.startTime);

}

});

});

2.使用 PerformanceObserver 的 observe() 方法监听 'paint' 类型的性能事件。

observer.observe({ entryTypes: ['paint'] });

3.在回调函数中，通过遍历 PerformanceEntryList 对象的 getEntries() 方法获取所有触发的性能事件，然后根据 entry.name 来判断是否是 FP 或 FCP。

4.如果是 FP，可以通过 entry.startTime 获取其开始的时间戳，进行相应的处理。同样，如果是 FCP，也可以通过 entry.startTime 获取其开始的时间戳。

完整的示例代码如下：

const observer = new PerformanceObserver((list) => {

const entries = list.getEntries();

entries.forEach((entry) => {

if (entry.name === 'first-paint') {

console.log('FP:', entry.startTime);

} else if (entry.name === 'first-contentful-paint') {

console.log('FCP:', entry.startTime);

}

});

});

observer.observe({ entryTypes: ['paint'] });

在上述代码中，创建了一个 PerformanceObserver 实例，并指定一个回调函数。在回调函数中，根据 entry.name 的值判断是否是 FP 或 FCP，并打印出对应的开始时间戳。然后通过调用 observer.observe() 方法监听 'paint' 类型的性能事件。

通过以上步骤，就可以使用 PerformanceObserver 统计 FP 和 FCP，并对这些性能指标进行进一步的处理和分析。

#### PerformanceObserver 如何统计 long tasks

要使用 PerformanceObserver 统计长任务（Long Tasks），可以按照以下步骤进行：

1.创建 PerformanceObserver 实例，并指定一个回调函数作为参数。

const observer = new PerformanceObserver((list) => {

const entries = list.getEntries();

entries.forEach((entry) => {

console.log('Long Task:', entry);

});

});

2.使用 PerformanceObserver 的 observe() 方法监听 'longtask' 类型的性能事件。

observer.observe({ entryTypes: ['longtask'] });

3.在回调函数中，通过遍历 PerformanceEntryList 对象的 getEntries() 方法获取所有触发的长任务事件。

4.可以通过遍历获得的长任务事件数据，并进行进一步的处理和分析。

完整的示例代码如下：

const observer = new PerformanceObserver((list) => {

const entries = list.getEntries();

entries.forEach((entry) => {

console.log('Long Task:', entry);

});

});

observer.observe({ entryTypes: ['longtask'] });

在上述代码中，创建了一个 PerformanceObserver 实例，并指定一个回调函数。在回调函数中，遍历 PerformanceEntryList 对象的 getEntries() 方法获取所有长任务事件，并打印出相关的长任务数据。

通过以上步骤，就可以使用 PerformanceObserver 统计长任务，并对这些长任务进行进一步的处理和分析。

### 补充： 页面性能指标有哪些？

以下是常见的页面性能指标，按照阶段顺序进行表述：

| 阶段 | 指标名称 | 描述 |
| --- | --- | --- |
| 导航阶段 | DNS 解析时间 | 浏览器解析域名并获取目标服务器IP地址所花费的时间 |
| 导航阶段 | TCP 连接时间 | 浏览器与服务器建立 TCP 连接所花费的时间 |
| 导航阶段 | SSL/TLS 握手时间 | 如果网站启用了 HTTPS，浏览器与服务器进行 SSL/TLS 握手所花费的时间 |
| 导航阶段 | 请求时间 | 浏览器发送 HTTP 请求并等待服务器响应的时间 |
| 导航阶段 | 首字节时间（TTFB） | 浏览器收到服务器响应的第一个字节所花费的时间 |
| 渲染阶段 | DOM 解析时间 | 浏览器将 HTML 文档解析为 DOM 树的时间 |
| 渲染阶段 | CSS 解析时间 | 浏览器将 CSS 样式表解析为 CSSOM 树的时间 |
| 渲染阶段 | 首次渲染时间（FP） | 浏览器将 DOM 树和 CSSOM 树合并，开始绘制页面的时间 |
| 渲染阶段 | 首次内容绘制时间（FCP） | 页面首次有可见内容被绘制的时间 |
| 渲染阶段 | 首次有意义绘制时间（FMP） | 页面首次有有意义的内容被绘制的时间 |
| 交互阶段 | 首次输入延迟时间（FID） | 用户首次与页面进行交互（点击按钮、输入框等）后，页面响应交互的时间 |
| 交互阶段 | 首次可交互时间（TTI） | 页面变得完全可交互（用户可以进行大部分常规操作）所花费的时间 |
| 交互阶段 | 页面完全加载时间（Page Load） | 页面所有资源（包括图片、CSS、JavaScript等）加载完成、渲染完毕并且可交互的时间 |
| 用户体验阶段 | 页面响应时间 | 用户发起请求后，页面完成响应所花费的时间 |
| 用户体验阶段 | 页面加载时间 | 用户打开页面到页面加载完成所花费的时间 |
| 用户体验阶段 | 页面交互性能 | 页面响应用户交互（点击、滚动等）的流畅程度 |

## 如果用户说 web 应用感觉很反应慢或者卡顿，该如何排查？

如果用户觉得 web 应用反应卡顿， 主要从哪几个方面来排查？

* 加载慢
* 资源下载慢
* 首屏并发请求资源过多
* 首屏接口慢
* 首屏对应的 JS 执行慢
* 首屏渲染慢
* 首屏加载静态资源过大
* .......
* 执行过程慢
* 接口慢
* long tasks 太多, 阻塞 JS 执行
* 内存泄漏
* 重绘重排 过多
* 关键节点没有加 节流防抖
* .......

### 主要排查手段有哪些

* 通过建立性能监控指标: 通过真实用户数据反馈， 来判断用户是否卡顿， 包含网络监控、运行时性能监控
* Chrome devtools: NetWork 主要排查网络问题
* Chrome devtools: Performance 主要细查性能运行时性能，包含了 long tasks、render 次数、重排重绘、执行时间线、阻塞场景
* Chrome devtools: Performance monitor 主要监控用户运行时性能，看看是否有内存泄露
* React Developer Tools: 可以用于追踪 react 应用性能、渲染次数、重排重绘
* Lighthouse: 全面分析网页性能的一个工具、支持浏览器插件
* webpack-bundle-analyzer: 进行产物依赖分析、包大小分析
* 抓包: 通过抓包的方式， 看看线上请求分析、请求模拟、网络劫持之后仅仅看 JS 执行时间
* E2E测试: 通过 E2E 进行性能预检， 每次上线前进行一系列系统操作， 看看时间耗时和线上耗时波动

### 主要解决办法和思路

#### 首屏加载慢的方向

* 资源加载方向
* 使用 tree shaking 减少包体积
* 代码压缩和混淆
* 对于高版本浏览器， 直接使用 ES6 语法，低版本浏览器再使用 ES5（es6 语法代码量会比编译成 es5 代码量小很多， 且执行速度也快）
* 使用 split chunks 进行大包拆分、小包复用
* 使用 gzip
* 使用 图片压缩
* 使用 雪碧图
* 图标使用 iconfont 加载
* 懒加载， 仅加载首屏必要资源
* 使用 tailwindcss 等技术， 复用 css
* 使用 微前端 技术，首屏仅加载当前子应用页面，可以做到只加载整站很少的一部分代码
* 首屏非必要依赖尽量延后到 FMP 或者 TTI 之后再加载
* 组件微前端化
* 渲染方向
* 尽量减少重排重绘
* 减少重复渲染（useMemo、useCallback、memo 等）
* 减少 setState 次数（多次 setState 可以合并为一次）
* 尽量减少 dom 节点深度
* 网络方向
* 使用流式服务端渲染， 可以查看文档：https://juejin.cn/post/6953819275941380109
* 使用服务端渲染， 减少首屏请求
* 使用 SSG 静态站点生成
* 首屏必要数据， 不作客户端请求， 用后端模板注入
* 使用 BFF 进行请求聚合
* 使用 CDN 进行网络请求分发
* DNS Prefetch
* 资源预加载（在闲暇时间加载后续页面所需要的资源和接口，例如：link rel preload）
* 启用 HTTP2 多路复用
* 在业务逻辑上， 首屏必要接口提前（例如在 html 加载的那一瞬间，利用一个非常小的 js 文件将首屏需要的请求发送出去, 然后缓存下来， 到业务使用的时候直接就使用即可）
* 使用缓存技术缓存资源与请求：强缓存、协商缓存、离线缓存、Service Worker 缓存、后端业务缓存

#### 运行时卡顿方向

* 查看是否存在有有 long tasks， 有计划的拆解 long tasks
* 解决项目中复杂度问题： https://www.jianshu.com/p/ffbb25380904
* 排查项目是否有内存泄露
* 排查特定业务流程是否有慢接口
* 高复杂计算逻辑放在 service worker 处理

## 【性能】以用户为中心的前端性能指标有哪些？

[以用户为中心的前端性能指标「译」](https://www.jianshu.com/p/456e6eff59c8)