

# 万字长文：分享前端性能优化知识体系

技术漫谈 2022-02-21 18:45

## 编者荐语：

大家好，我是Levi，今天分享一篇掘金上的前端性能优化的文章。

以下文章来源于稀土掘金技术社区，作者雨亭



稀土掘金技术社区

掘金，一个帮助开发者成长的技术社区



## 前言

最近在学习前端性能优化方面的知识，看了很多大佬的文章，感觉文章多了比较零散，学习效率不高，所以就整合了一下大佬们写的性能优化的东西，[从页面的渲染过程来建立自己的一个前端性能优化的体系](#)，本来打算自己留着复习，毕竟是整合的也不是自己写的，看现在马上金三银四了，发出来给准备面试的兄弟们看一下吧，也希望节约一下兄弟们的时间，文章大部分都是整合的，如果大佬们介意可以联系我删除。

我们要优化一个网站的性能，首先需要学会如何衡量一个网站的性能。

## RAIL 模型



image.png

## 以用户为中心的性能指标

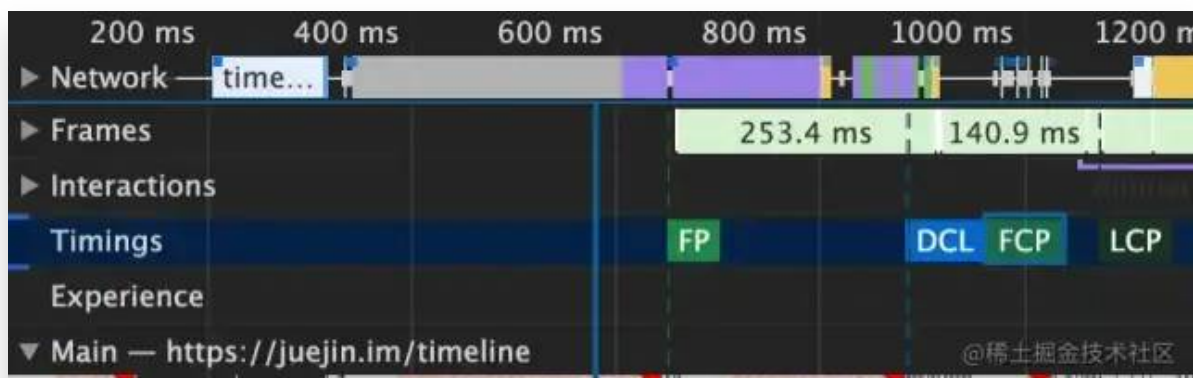
- First Paint 首次绘制 (FP)
- First contentful paint 首次内容绘制 (FCP)
- Largest contentful paint 最大内容绘制 (LCP)
- First input delay 首次输入延迟 (FID)
- Time to Interactive 可交互时间 (TTI)
- Total blocking time 总阻塞时间 (TBT)
- Cumulative layout shift 累积布局偏移 (CLS)

## FP & FCP

首次绘制，FP (First Paint)，这个指标用于记录页面第一次绘制像素的时间。

首次内容绘制，FCP (First Contentful Paint)，这个指标用于记录页面首次绘制文本、图片、非空白 Canvas 或 SVG 的时间。

这两个指标看起来大同小异，但是 FP 发生的时间一定小于等于 FCP，如下图是掘金指标：



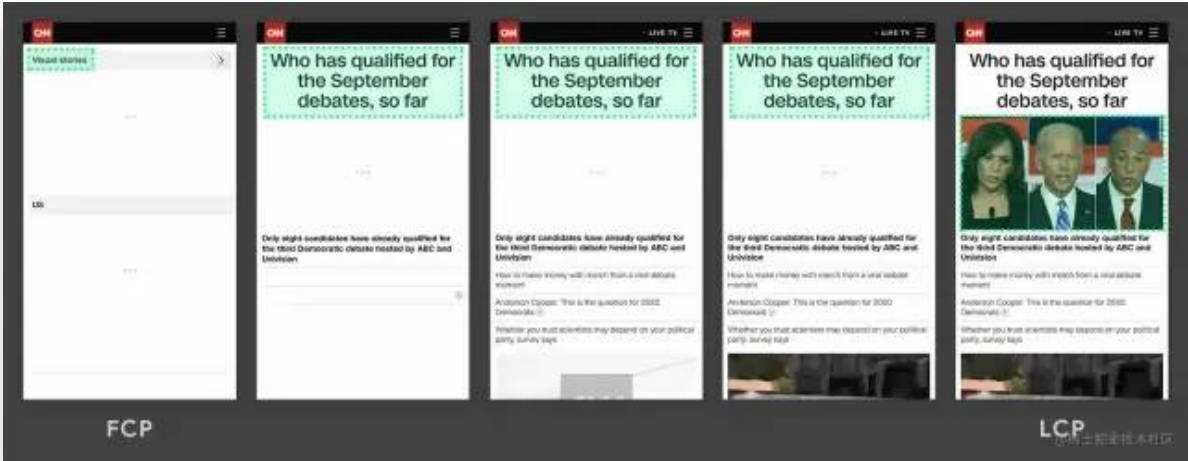
FP 指的是绘制像素，比如说页面的背景色是灰色的，那么在显示灰色背景时就记录下了 FP 指标。但是此时 DOM 内容还没开始绘制，可能需要文件下载、解析等过程，只有当 DOM 内容发生变化才会触发，比如说渲染出了一段文字，此时就会记录下 FCP 指标。因此说我们可以把这两个指标认为是和白屏时间相关的指标，所以肯定是最快越好。

FCP time (in seconds)	Color-coding	FCP score (HTTP Archive percentile)
0-2	Green (fast)	75-100
2-4	Orange (moderate)	50-74
Over 4	Red (slow)	0-49

上图是官方推荐的时间区间，也就是说如果 FP 及 FCP 两指标在 2 秒内完成的话我们的页面就算体验优秀。

## LCP

最大内容绘制，LCP（Largest Contentful Paint），用于记录视窗内最大的元素绘制的时间，该时间会随着页面渲染变化而变化，因为页面中的最大元素在渲染过程中可能会发生改变，另外该指标会在用户第一次交互后停止记录。指标变化如下图：



LCP 其实能比前两个指标更能体现一个页面的性能好坏程度，因为这个指标会持续更新。举个例子：当页面出现骨架屏或者 Loading 动画时 FCP 其实已经被记录下来了，但是此时用户希望看到的内容其实并未呈现，我们更想知道的是页面主要的内容是何时呈现出来的。

此时 LCP 指标是能够帮助我们实现想要的需求的。

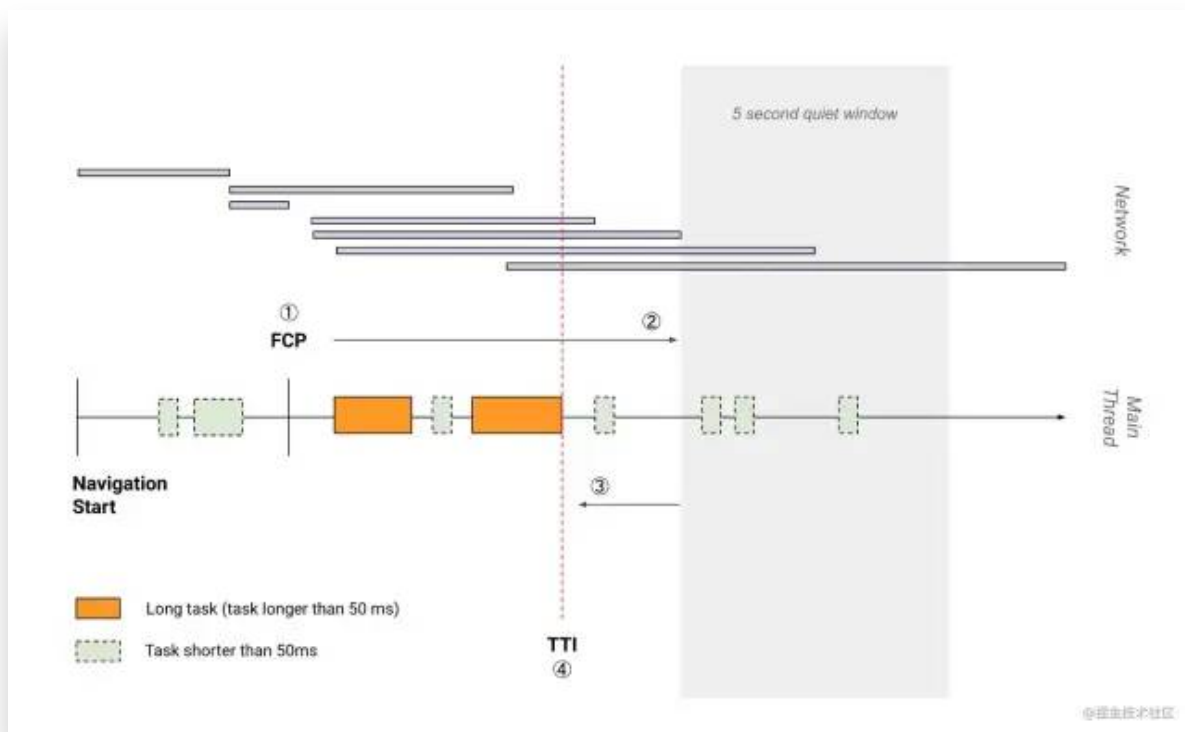


上图是官方推荐的时间区间，在 2.5 秒内表示体验优秀。

## TTI

首次可交互时间，TTI（Time to Interactive）。这个指标计算过程略微复杂，它需要满足以下几个条件

1. 从 FCP 指标后开始计算
2. 持续 5 秒内无长任务（执行时间超过 50 ms）且无两个以上正在进行中的 GET 请求
3. 往前回溯至 5 秒前的最后一个长任务结束的时间



这里你可能会疑问为什么长任务需要定义为 50ms 以外？

Google 提出了一个 RAIL 模型：



对于用户交互（比如点击事件），推荐的响应时间是 100ms 以内。那么为了达成这个目标，推荐在空闲时间里执行任务不超过 50ms（[W3C<sup>\[1\]</sup>](#) 也有这样的标准规定），这样能在用户无感知的情况下响应用户的交互，否则就会造成延迟感。

长任务也会在 FID 及 TBT 指标中使用到。

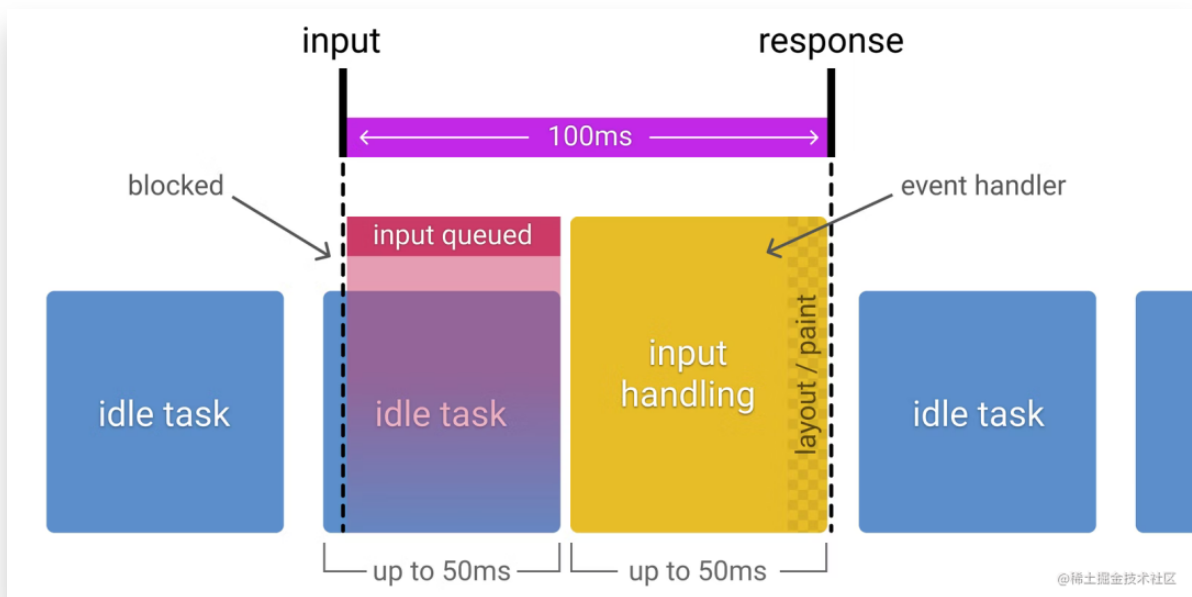


image.png

因此这是一个很重要的用户体验指标，代表着页面何时真正进入可用的状态。毕竟光内容渲染的快也不够，还要能迅速响应用户的交互。想必大家应该体验过某些网站，虽然内容渲染出来了，但是响应交互很卡顿，只能过一会才能流畅交互的情况。

## FID

首次输入延迟，FID（First Input Delay），记录在 FCP 和 TTI 之间用户首次与页面交互时响应的延迟。

这个指标其实挺好理解，就是看用户交互事件触发到页面响应中间耗时多少，如果其中有长任务发生的话那么势必会造成响应时间变长。

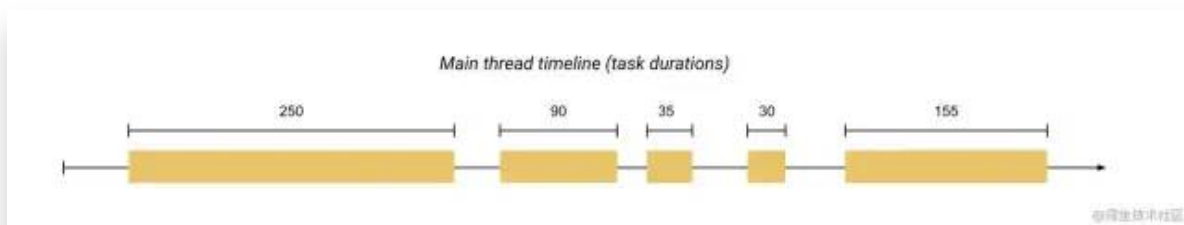
其实在上文我们就讲过 Google 推荐响应用户交互在 100ms 以内：



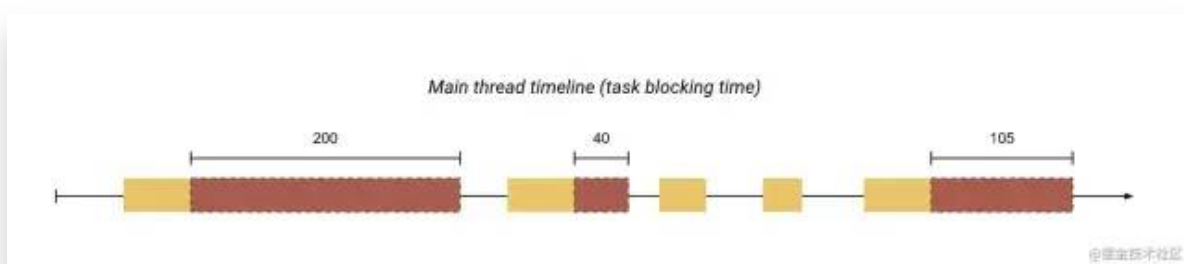
## TBT

阻塞总时间，TBT（Total Blocking Time），记录在 FCP 到 TTI 之间所有长任务的阻塞时间总和。

假如说在 FCP 到 TTI 之间页面总共执行了以下长任务（执行时间大于 50ms）及短任务（执行时间低于 50ms）



那么每个长任务的阻塞时间就等于它所执行的总时间减去 50ms



所以对于上图的情况来说，TBT 总共等于 345ms。

这个指标的高低其实也影响了 TTI 的高低，或者说和长任务相关的几个指标都有关联性。

## CLS

累计位移偏移，CLS（Cumulative Layout Shift），记录了页面上非预期的位移波动。

大家想必遇到过这类情况：页面渲染过程中突然插入一张巨大的图片或者说点击了某个按钮突然动态插入了一块内容等等相当影响用户体验的网站。这个指标就是为这种情况而生的，计算方式为：位移影响的面积 \* 位移距离。





以上图为例，文本移动了 25% 的屏幕高度距离（位移距离），位移前后影响了 75% 的屏幕高度面积（位移影响的面积），那么 CLS 为  $0.25 * 0.75 = 0.1875$ 。



CLS 推荐值为低于 0.1，越低说明页面跳来跳去的情况就越少，用户体验越好。毕竟很少有人喜欢阅读或者交互过程中网页突然动态插入 DOM 的情况，比如说插入广告~

介绍完了所有的指标，接下来我们来了解哪些是用户体验三大核心指标、如何获取相应的指标数据及如何优化。

## 三大核心指标

Google 在今年五月提出了网站用户体验的三大核心指标，分别为：

- LCP
- FID
- CLS

LCP 代表了页面的速度指标，虽然还存在其他的一些体现速度的指标，但是上文也说过 LCP 能体现的东西更多一些。一是指标实时更新，数据更精确，二是代表着页面最大元素的渲染时间，通常来说页面中最大元素的快速载入能让用户感觉性能还挺好。



FID 代表了页面的交互体验指标，毕竟没有一个用户希望触发交互以后页面的反馈很迟缓，交互响应的快会让用户觉得网页挺流畅。

CLS 代表了页面的稳定指标，尤其在手机上这个指标更为重要。因为手机屏幕挺小，CLS 值一大的话会让用户觉得页面体验做的很差。

## 如何获取指标

### Lighthouse

你可以通过安装 [Lighthouse](#)<sup>[2]</sup> 插件来获取如下指标

Metrics

First Contentful Paint

4.0 s

Time to Interactive

11.6 s

Speed Index

10.7 s

Total Blocking Time

980 ms

Largest Contentful Paint

19.1 s

Cumulative Layout Shift

0.201

Values are estimated and may vary. The [performance score is calculated](#) directly from these metrics. [See calculator.](#)

### web-vitals-extension

官方出品，你可以通过安装 [web-vitals-extension](#)<sup>[3]</sup> 插件来获取三大核心指标

https://github.com - 22:45:40

Metrics

▲

Largest Contentful Paint

2.53 s

●

First Input Delay

5.24 ms

●

Cumulative Layout Shift (might change)

0.000

@稀土掘金技术社区

### web-vitals 库

官方出品，你可以通过安装 [web-vitals](#)<sup>[4]</sup> 包来获取如下指标





## CORE WEB VITALS

- Cumulative Layout Shift (CLS)
- First Input Delay (FID)
- Largest Contentful Paint (LCP)

## Other Web Vitals

- First Contentful Paint (FCP)
- Time to First Byte (TTFB)

@掘金技术社区

代码使用方式也挺简单：

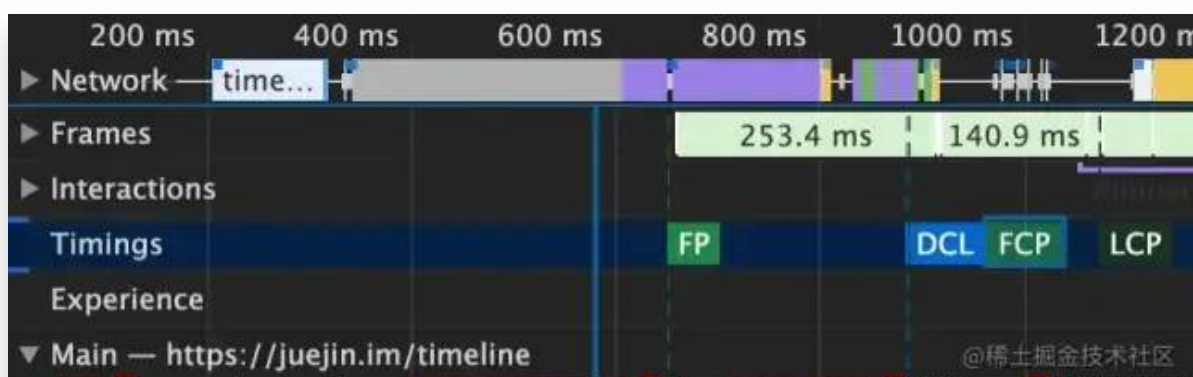
```
import {getCLS, getFID, getLCP} from 'web-vitals';

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

复制代码

## Chrome DevTools

打开 Performance 即可快速获取如下指标



我们对网站性能进行优化前首先需要了解一个页面是如何渲染的，知道可以在页面渲染的哪些过程中进行优化。

## 页面的渲染过程

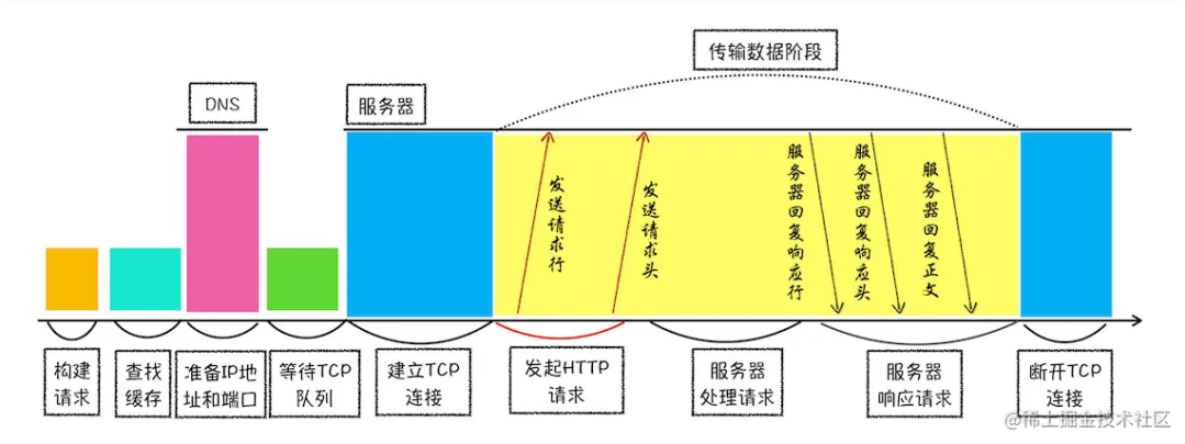


image.png

```
graph TD
    A[输入网址] --> B[解析URL]
    B --> C[检查浏览器缓存]
    C --> D[DNS解析]
    D --> E[TCP/IP连接]
    E --> F[http请求]
    F --> G[服务器请求并返回http报文]
    G --> H[浏览器渲染页面]
    H --> I[断开连接]
```

## 输入网址

当用户在地址栏中输入一个查询关键字时，地址栏会判断输入的关键字是搜索内容，还是请求的 URL。

- 如果是搜索内容，地址栏会使用浏览器默认的搜索引擎，来合成新的带搜索关键字的 URL。
- 如果判断输入内容符合 URL 规则，比如输入的是 time.geekbang.org，那么地址栏会根据规则，把这段内容加上协议，合成为完整的 URL，如 https://time.geekbang.org。当用户输入关键字并键入回车之后，这意味着当前页面即将要被替换成新的页面，不过在这个流程继续之前，浏览器还给了当前页面一次执行 beforeunload 事件的机会，beforeunload 事件允许页面在退出之前执行一些数据清理操作，还可以询问用户是否要离开当前页面，比如当前页面可能有未提交完成的表单等情况，因此用户可以通过 beforeunload 事件来取消导航，让浏览器不再执行任何后续工作。

## 解析URL

URL 主要由 **协议**、**主机**、**端口**、**路径**、**查询参数**、**锚点** 6部分组成！输入URL后，浏览器会解析出协议、主机、端口、路径等信息，并构造一个HTTP请求。

## 检查浏览器缓存

首先，网络进程会查找本地缓存是否缓存了该资源。如果有缓存资源，那么直接返回资源给浏览器进程；如果在缓存中没有查找到资源，那么直接进入网络请求流程。

优化：

### 开启浏览器缓存

1. 浏览器发送请求前，根据请求头的 **expires** 和 **cache-control** 判断是否命中（包括是否过期）强缓存策略，如果命中，直接从缓存获取资源，并不会发送请求。如果没有命中，则进入下一步。
2. 没有命中强缓存规则，浏览器会发送请求，根据请求头的 **If-Modified-Since** 和 **If-None-Match** 判断是否命中协商缓存，如果命中，直接从缓存获取资源。如果没有命中，则进入下一步。
3. 如果前两步都没有命中，则直接从服务端获取资源。

### 第三方库公共模块抽取

通常情况下我们的 WebApp 是有我们的自身代码和第三方库组成的,我们自身的代码是会常常变动的,而第三方库除非有较大的版本升级,不然是不会变的,所以第三方库和我们的代码需要分开打包,我们可以给第三方库设置一个较长的强缓存时间,这样就不会频繁请求第三方库的代码了。

那么如何提取第三方库呢?在 webpack4.x 中, SplitChunksPlugin 插件取代了 CommonsChunkPlugin 插件来进行公共模块抽取,我们可以对 SplitChunksPlugin 进行配置进行 **拆包** 操作。详情见前端性能优化三部曲(加载篇)

## DNS解析

在发起http请求之前，浏览器首先要去做去获得我们想访问网页的IP地址，浏览器会发送一个UDP的包给DNS域名解析服务器。

# DNS域名解析过程

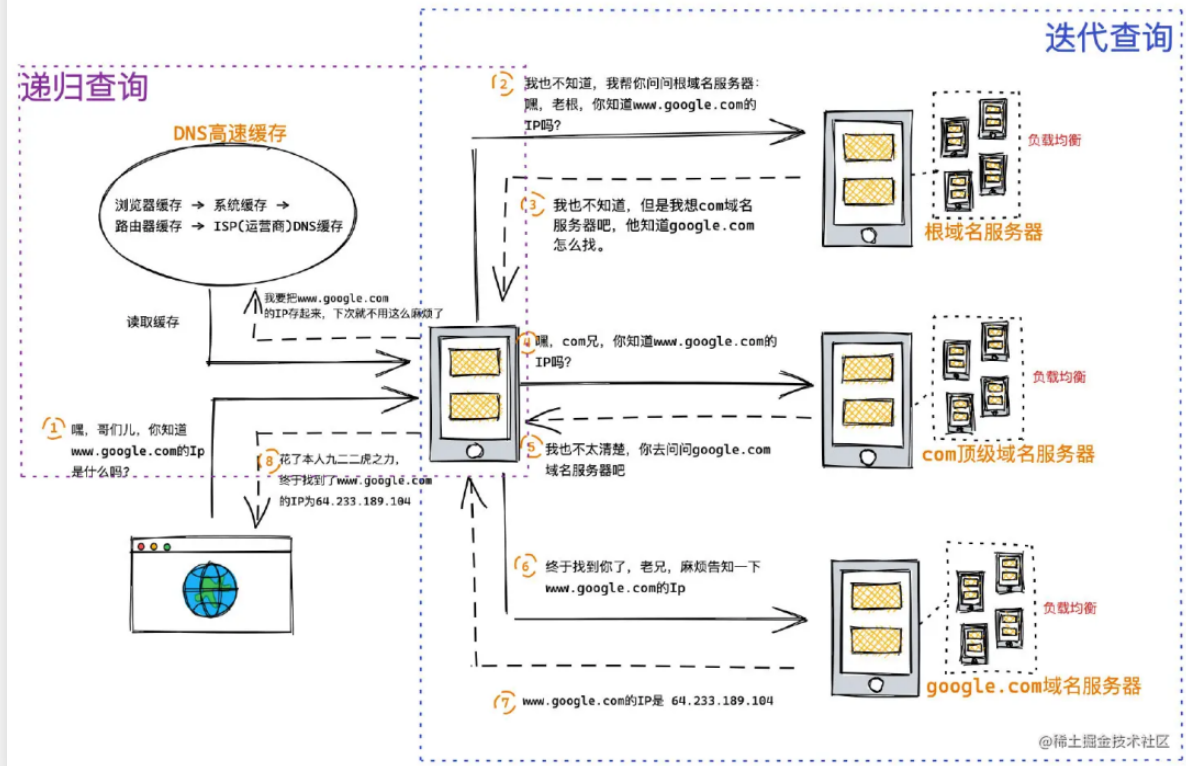
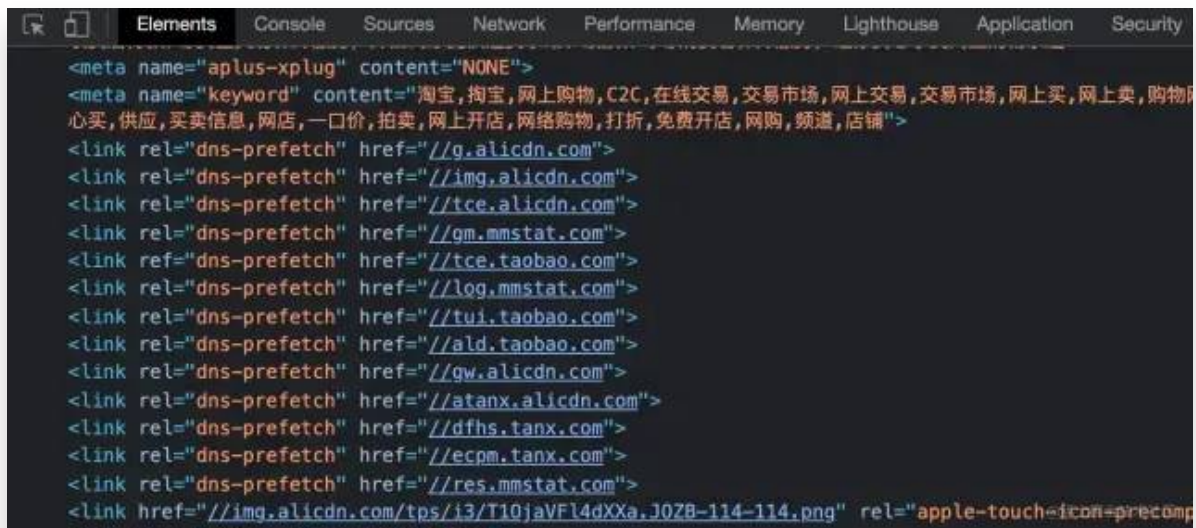


image.png

优化:

## DNS预解析

大型网站, 有多个不同服务器资源的情况下, 都可采取DNS预解析, 提前解析, 减少页面卡顿。

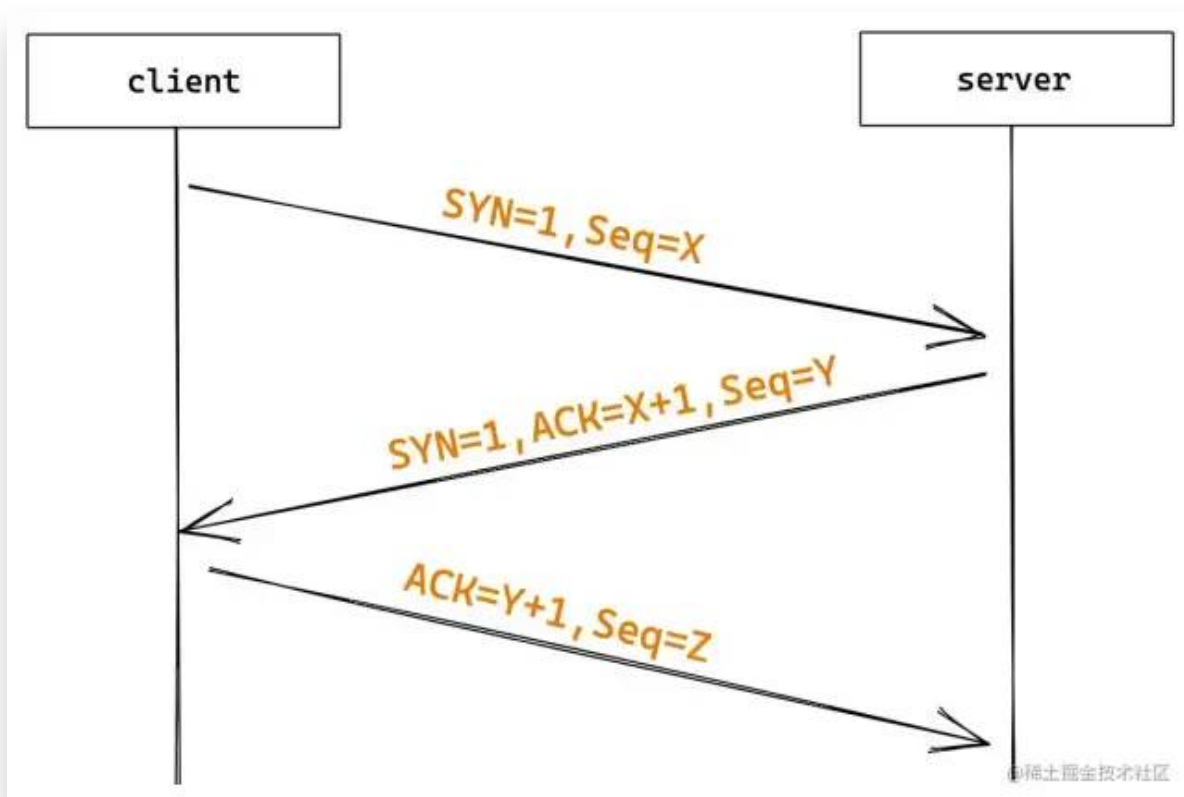


## DNS负载均衡

DNS还有负载均衡的作用，现在很多网站都有多个服务器，当一个网站访问量过大的时候，如果所有请求都请求在同一个服务器上，可能服务器就会崩掉，这时候就用到了DNS负载均衡技术，当一个网站有多个服务器地址时，在应答DNS查询的时候，DNS服务器会对每个查询返回不同的解析结果，也就是返回不同的IP地址，从而把访问引导到不同的服务器上去，来达到负载均衡的目的。例如可以根据每台机器的负载量，或者该机器距离用户的地理位置距离等等条件。

## TCP/IP连接

### 三次握手



位码即tcp标志位，有6种标示：

- SYN(synchronous建立联机)
- ACK(acknowledgement 确认)
- PSH(push传送)
- FIN(finish结束)
- RST(reset重置)
- URG(urgent紧急)



第一次握手：主机A发送位码为 `SYN=1`，随机产生 `Seq number=1234567` 的数据包到服务器，主机B由 `SYN=1` 知道，A要求建立联机；（第一次握手，由浏览器发起，告诉服务器我要发送请求了）

第二次握手：主机B收到请求后要确认联机信息，向A发送 `ack number=(主机A的seq+1)`，`SYN=1`，`ACK=1234567 + 1`，随机产生 `Seq=7654321` 的包；（第二次握手，由服务器发起，告诉浏览器我准备接受了，你赶紧发送吧）

第三次握手：主机A收到后检查 `ack number` 是否正确，即第一次发送的 `seq number+1`，以及位码 `SYN` 是否为1，若正确，主机A会再发送 `ack number=(主机B的seq+1)`，`ack=7654321 + 1`，主机B收到后确认 `Seq` 值与 `ACK=7654321+ 1` 则连接建立成功；（第三次握手，由浏览器发送，告诉服务器，我马上就发了，准备接受吧）

## http请求

连接建立之后，浏览器端会构建请求行、请求头等信息，并把和该域名相关的 Cookie 等数据附加到请求头中，然后向服务器发送构建的请求信息。

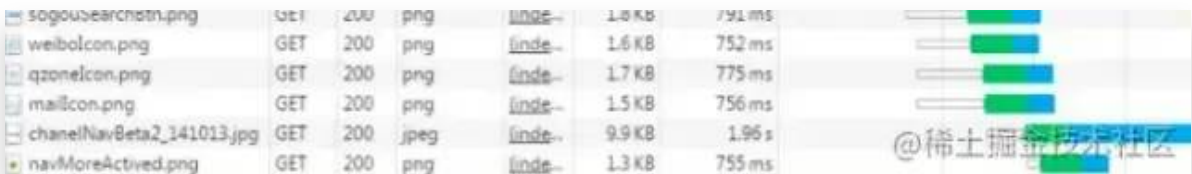
优化：

开启HTTP2

我们看到在获取 html 之后我们需要自上而下解析,在解析到 `script` 相关标签的时候才能请求相关资源,而且由于浏览器并发限制,我们最多一次性请求 6 次,那么有没有办法破解这些困境呢?

http2 是非常好的解决办法,http2 本身的机制就足够快:

1. http2采用二进制分帧的方式进行通信,而 http1.x 是用文本,http2 的效率更高
2. http2 可以进行多路复用,即跟同一个域名通信,仅需要一个 TCP 建立请求通道,请求与响应可以同时基于此通道进行双向通信,而 http1.x 每次请求需要建立 TCP,多次请求需要多次连接,还有并发限制,十分耗时



sogousearchbtn.png	GET	200	png	linde-	1.8 KB	791 ms
weiboicon.png	GET	200	png	linde-	1.6 KB	752 ms
qzoneicon.png	GET	200	png	linde-	1.7 KB	775 ms
mailicon.png	GET	200	png	linde-	1.5 KB	756 ms
chanelNavBeta2_141013.jpg	GET	200	jpeg	linde-	9.9 KB	1.96 s
navMoreActived.png	GET	200	png	linde-	1.3 KB	755 ms

image.png

3. http2 可以头部压缩,能够节省消息头占用的网络的流量,而HTTP/1.x每次请求, 都会携带大量冗余头信息, 浪费了很多带宽资源

例如：下图中的两个请求， 请求一发送了所有的头部字段， 第二个请求则只需要发送差异数据， 这样可以减少冗余数据， 降低开销

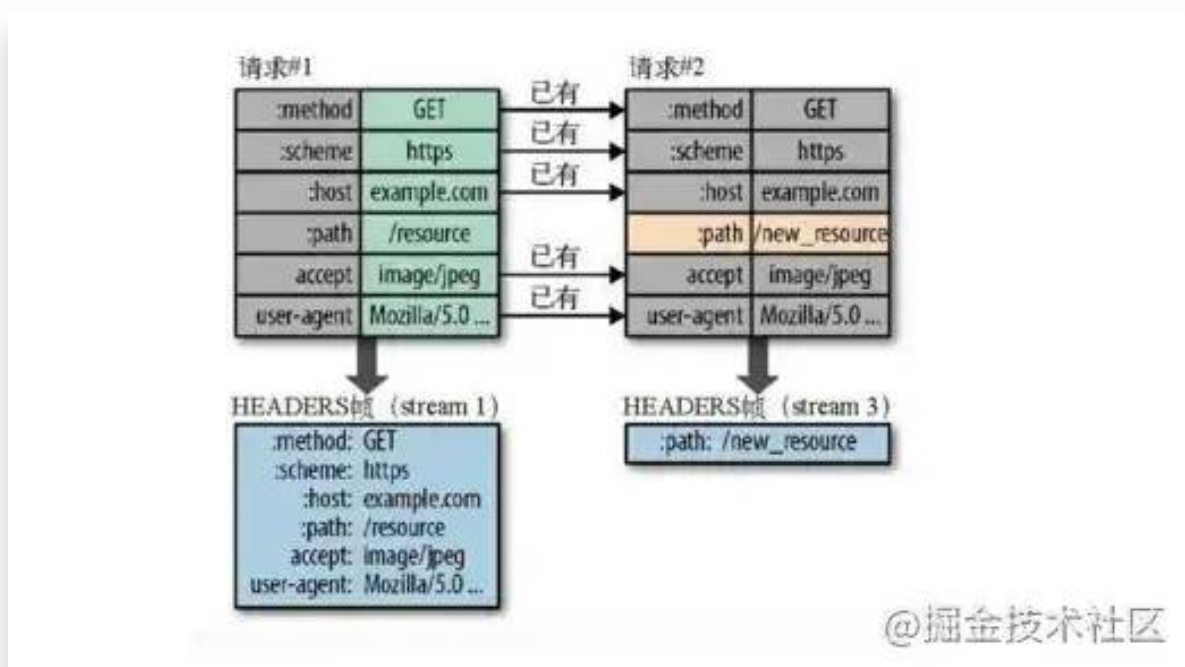


image.png

4. http2可以进行服务端推送,我们平时解析 HTML 后碰到相关标签才会进而请求 css 和 js 资源,而 http2 可以直接将相关资源直接推送,无需请求,这大大减少了多次请求的耗时

我们可以点击[此网站\[5\]](#) 进行 http2 的测试

http2 在网络通畅+高性能设备下的表现没有比 http1.1有明显的优势,但是网络越差,设备越差的情况下 http2 对加载的影响是质的,可以说 http2 是为移动 web 而生的,反而在光纤加持的高性能PC 上优势不太明显.

## 服务器请求并返回http报文

服务器接收到请求信息后, 会根据请求信息生成响应数据 (包括响应行、响应头和响应体等信息), 并发给网络进程。等网络进程接收了响应行和响应头之后, 就开始解析响应头的内容了。

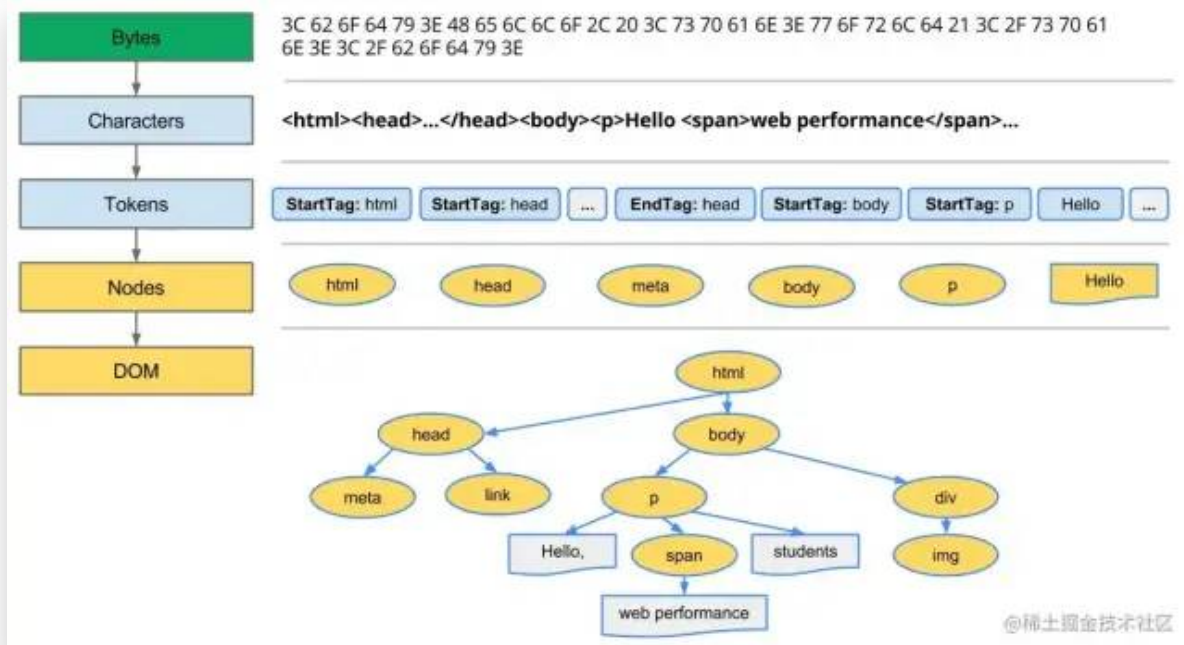
## 浏览器渲染页面



## DOM树

字节 → 字符 → 令牌 → 节点 → 对象模型。

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div></div>
  </body>
</html>
```

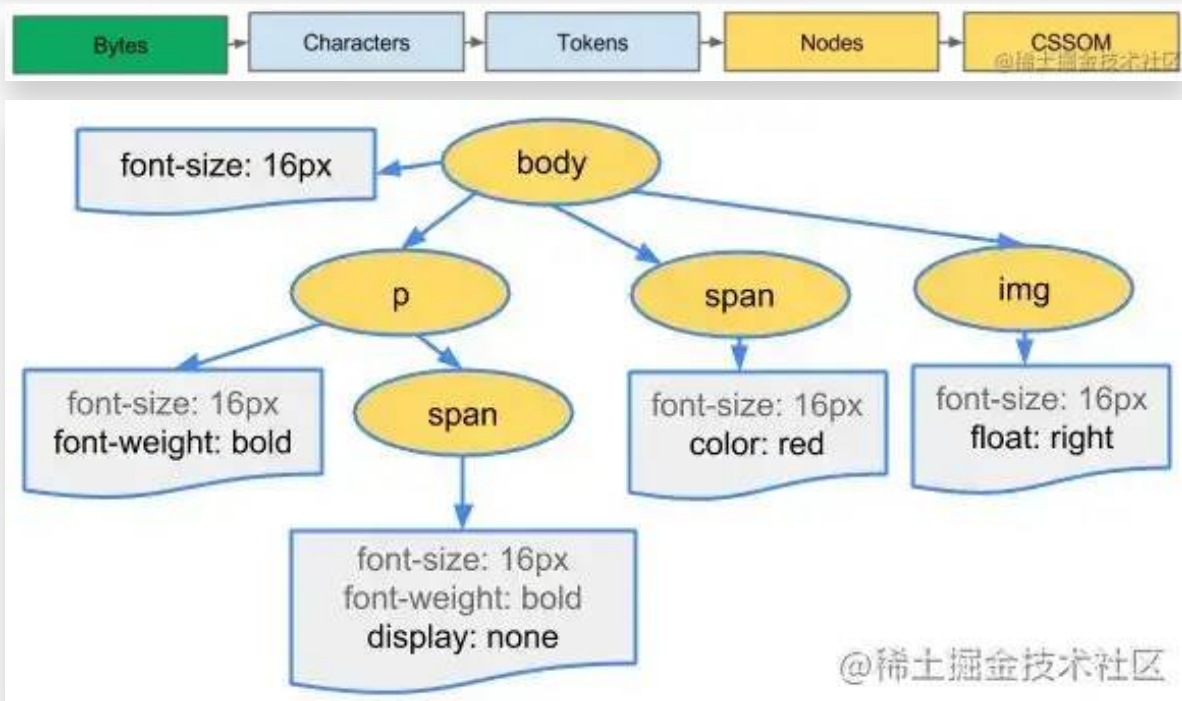


- 转换: 浏览器从磁盘或网络读取 HTML 的原始字节, 并根据文件的指定编码 (例如 UTF-8) 将它们转换成各个字符。
- 令牌化: 浏览器将字符串转换成 W3C HTML5 标准规定的各种令牌, 例如, “<”、“>”, 以及其他尖括号内的字符串。每个令牌都具有特殊含义和一组规则。
- 词法分析: 发出的令牌转换成定义其属性和规则的“对象”。
- DOM 构建: 最后, 由于 HTML 标记定义不同标记之间的关系 (一些标记包含在其他标记内), 创建的对象链接在一个树数据结构内, 此结构也会捕获原始标记中定义的父亲-子

项关系: HTML 对象是 body 对象的父项, body 是 paragraph 对象的父项, 依此类推。

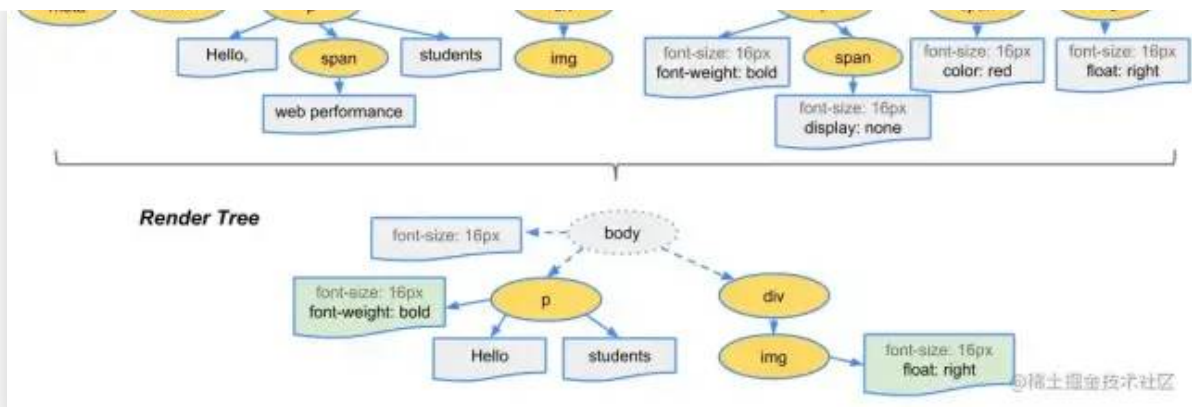
- CSS 对象模型 (CSSOM)

```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```



- 布局树Layout Tree
- DOM 树与 CSSOM 树合并后形成渲染树。
- 渲染树只包含渲染网页所需的节点。
- 布局计算每个对象的精确位置和大小。
- 最后一步是绘制, 使用最终渲染树将像素渲染到屏幕上。





## 渲染

渲染流程：

1. 获取DOM后分割为多个图层
2. 对每个图层的节点计算样式结果（Recalculate style--样式重计算）
3. 为每个节点生成图形和位置（Layout--重排,回流）
4. 将每个节点绘制填充到图层位图中（Paint--重绘）
5. 图层作为纹理上传至GPU
6. 组合多个图层到页面上生成最终屏幕图像（Composite Layers--图层重组）

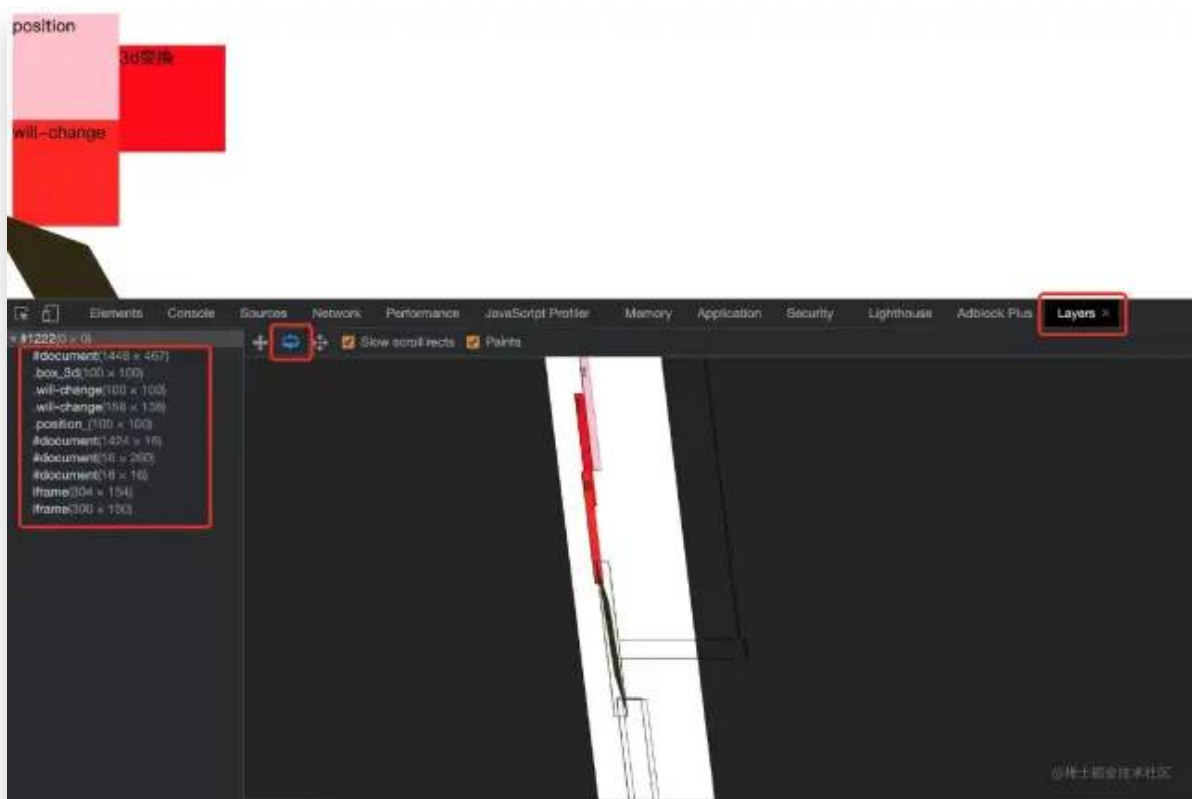
## 创建图层



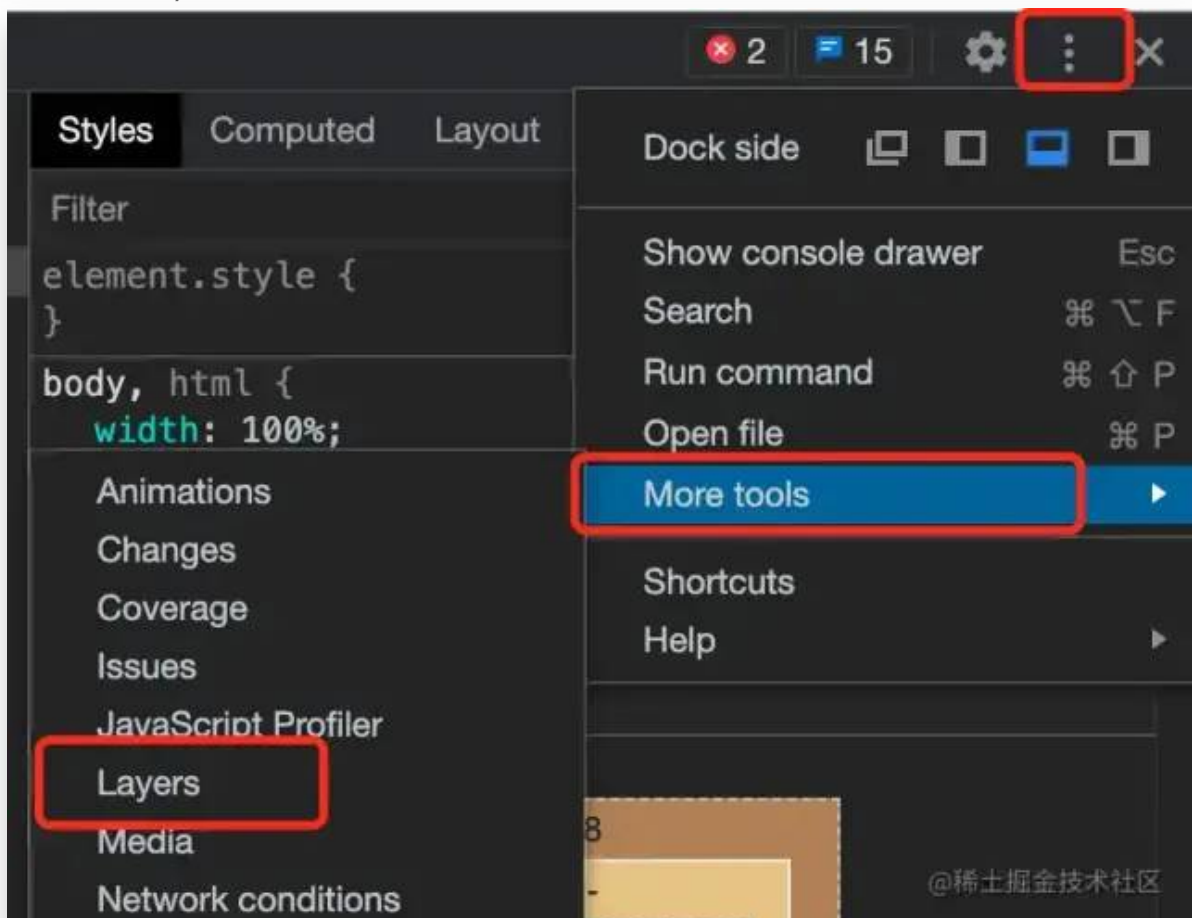
```
<div class="position_">position</div>
<div class="box_3d">3d变换</div>
<div class="will-change">will-change</div>
<div class="transform"></div>
<iframe src="https://www.baidu.com"></iframe>

div {width: 100px;height: 100px;}
.position_ {background: pink;position: fixed;z-index: 20;}
.box_3d {background: red;transform: translate3d(100px,30px,10px);}
.will-change {background: #f12312;will-change: transform;}
.transform {background: #302912;transform: skew(30deg, 20deg);}
```

在 chrome 上查看 Layers.



如果没有打开Layers,按下图打开:



知道图层的存在，我们可以手动打开一个图层，通过添加 `transform: translateZ(0)` 这样回流和重绘的代价就小了，效率就会大大提高。但是不要滥用这个属性，否则会大大增加内存消耗。—— 开启GPU加速。

- 重绘

当页面中元素样式的改变并不影响它在文档流中的位置时（例如：color、background-color、visibility等），浏览器会将新样式赋予给元素并重新绘制它，这个过程称为重绘。

- 回流

当Render Tree中部分或全部元素的尺寸、结构、或某些属性发生改变时，浏览器重新渲染部分或全部文档的过程称为回流。

- 回流必将引起重绘，而重绘不一定会引起回流。

引起回流：

1. 页面首次渲染
2. 浏览器窗口大小发生改变
3. 元素尺寸或位置发生改变
4. 元素内容变化（文字数量或图片大小等等）
5. 元素字体大小变化
6. 添加或者删除可见的DOM元素
7. 激活CSS伪类（例如：:hover）
8. 查询某些属性或调用某些方法

引起回流的属性和方法：

- clientWidth、clientHeight、clientTop、clientLeft
- offsetWidth、offsetHeight、offsetTop、offsetLeft
- scrollWidth、scrollHeight、scrollTop、scrollLeft
- scrollIntoView()、scrollIntoViewIfNeeded()
- getComputedStyle()
- getBoundingClientRect()
- scrollTo()

优化：

## 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。

## 防止脚本阻塞

将 CSS 放在文件头部，JavaScript 文件放在底部

1. CSS 执行会阻塞渲染，阻止 JS 执行
2. JS 加载和执行会阻塞 HTML 解析，阻止 CSSOM 构建

## 图片优化

1. 压缩图片
2. 图片延迟加载
3. 响应式图片
4. 使用 webp 格式的图片

## 压缩文件

在 webpack 可以使用如下插件进行压缩：

- JavaScript：UglifyPlugin
- CSS：MiniCssExtractPlugin
- HTML：HtmlWebpackPlugin

使用 gzip 压缩。可以通过向 HTTP 请求头中的 Accept-Encoding 头添加 gzip 标识来开启这一功能。服务器也得支持这一功能。

## 减少重绘重排

- CSS
1. 避免使用table布局;

2. 尽可能在DOM树的最末端改变class;
  3. 避免设置多层内联样式;
  4. 将动画效果应用到position属性为absolute或fixed的元素上;
  5. 避免使用CSS表达式 (例如: calc()) 。
- JS
    1. 避免频繁操作样式, 最好一次性重写style属性, 或者将样式列表定义为class并一次性更改class属性。
    2. 避免频繁操作DOM, 创建一个documentFragment, 在它上面应用所有DOM操作, 最后再把它添加到文档中。
    3. 也可以先为元素设置display: none, 操作结束后再把它显示出来。因为在display属性为none的元素上进行的DOM操作不会引发回流和重绘。
    4. 避免频繁读取会引发回流/重绘的属性, 如果确实需要多次使用, 就用一个变量缓存起来。
    5. 对具有复杂动画的元素使用绝对定位, 使它脱离文档流, 否则会引起父元素及后续元素频繁回流。

## 代码分割 (Code Splitting)

进行路由分割,甚至可以进行组件级别的代码分割,当然是用方式也是大同小异,组件的级别的分割带来的好处是我们可以页面的加载中只渲染部分必须的组件,而其余的组件可以按需加载.

## Tree Shaking

Tree Shaking的作用就是,通过程序流分析找出你代码中无用的代码并剔除,如果不用Tree Shaking那么很多代码虽然定义了但是永远都不会用到,也会进入用户的客户端执行,这无疑是性能的杀手,Tree Shaking依赖es6的module模块的静态特性,通过分析剔除无用代码.

## Skeleton (骨架屏)

在白屏结束之后,页面开始渲染,但是此时的页面还只是出现个别无意义的元素,比如下拉菜单按钮、或者乱序的元素、导航等等, 这些元素虽然是页面的组成部分但是没有意义. 什么是有意义?

- 对于搜索引擎用户是完整搜索结果
- 对于微博用户是时间线上的微博内容



- 对于淘宝用户是商品页面的展示

那么在FCP 和 FMP(首次有意义绘制)之间虽然开始绘制页面,但是整个页面是没有意义的,用户依然在焦虑等待,而且这个时候可能出现乱序的元素或者闪烁的元素,很影响体验,此时我们可能需要进行用户体验上的一些优化。

Skeleton是一个好方法,Skeleton现在已经很开始被广泛应用了,它的意义在于事先撑开即将渲染的元素,避免闪屏,同时提示用户这要渲染东西了,较少用户焦虑

## 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: 首屏渲染快, SEO 好。

缺点: 配置麻烦, 增加了服务器的计算压力。

## (伪)服务端渲染

那么既然在 HTML 加载到 js 执行期间会有时间等待,那么为什么不直接服务端渲染呢?直接返回的 HTML 就是带完整 DOM 结构的,省得还得调用 js 执行各种创建 dom 的工作,不仅如此还对 SEO 友好。

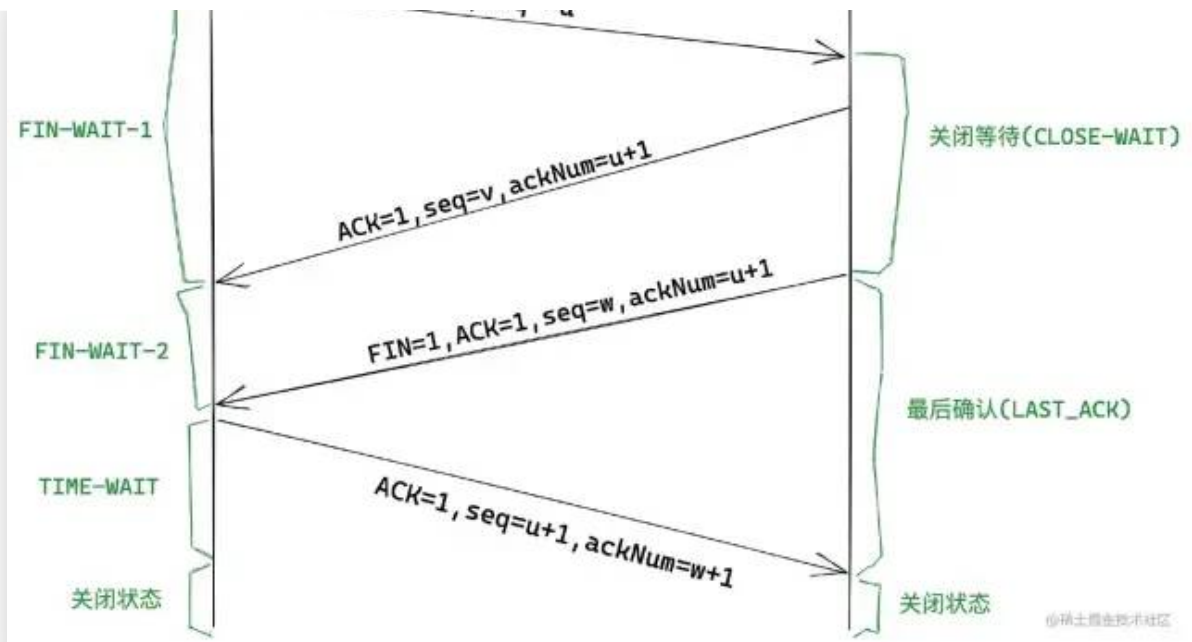
正是有这种需求 vue 和 react 都支持服务端渲染,而相关的框架Nuxt.js、Next.js也大行其道,当然对于已经采用客户端渲染的应用这个成本太高了。

于是有人想到了办法,谷歌开源了一个库Puppeteer,这个库其实是一个无头浏览器,通过这个无头浏览器我们能用代码模拟各种浏览器的操作,比如我们就可以用 node 将 html 保存为 pdf,可以在后端进行模拟点击、提交表单等操作,自然也可以模拟浏览器获取首屏的 HTML 结构。

[prerender-spa-plugin<sup>\[6\]</sup>](#)就是基于以上原理的插件,此插件在本地模拟浏览器环境,预先执行我们的打包文件,这样通过解析就可以获取首屏的 HTML,在正常环境中,我们就可以返回预先解析好的 HTML 了。

## 断开连接





1. 刚开始双方都处于established状态，假如是客户端先发起关闭请求
2. 第一次挥手：客户端发送一个FIN报文，报文中会指定一个序列号。此时客户端处于FIN\_WAIT1状态
3. 第二次挥手：服务端收到FIN之后，会发送ACK报文，且把客户端的序列号值+1作为ACK报文的序列号值，表明已经收到客户端的报文了，此时服务端处于CLOSE\_WAIT状态
4. 第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发送FIN报文，且指定一个序列号。此时服务端处于LAST\_ACK的状态
5. 需要过一阵子以确保服务端收到自己的ACK报文之后才会进入CLOSED状态，服务端收到ACK报文之后，就处于关闭连接了，处于CLOSED状态。

参考链接:

1. 前端性能优化三部曲(加载篇)
2. 前端性能优化 24 条建议 (2020)
3. 从输入URL开始建立前端知识体系
4. 全链路前端性能优化(欢迎收藏)
5. 还在看那些老掉牙的性能优化文章么？这些最新性能指标了解下
6. 浏览器工作原理与实践

## 参考资料

- [1] <https://www.w3.org/TR/2017/WD-longtasks-1-20170907/>: <https://link.juejin.cn?target=https%3A%2F%2Fwww.w3.org%2FTR%2F2017%2FWD-longtasks-1-20170907%2F>
- [2] <https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk>: <https://link.juejin.cn?target=https%3A%2F%2Fchrome.google.com%2Fwebstore%2Fdetail%2Flighthouse%2Fblipmdconlkpinefehnmjammfjpmpbjk>

- [3] <https://github.com/GoogleChrome/web-vitals-extension>: <https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2FGoogleChrome%2Fweb-vitals-extension>
- [4] <https://github.com/GoogleChrome/web-vitals>: <https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2FGoogleChrome%2Fweb-vitals>
- [5] <https://http2.akamai.com/demo>: <https://link.juejin.cn?target=https%3A%2F%2Fhttp2.akamai.com%2Fdemo>
- [6] <https://github.com/chrisvfritz/prerender-spa-plugin>: <https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fchrisvfritz%2Fprerender-spa-plugin>

[阅读原文](#)

喜欢此内容的人还喜欢

新来个技术总监，给团队引入了这款开发神器，同事直呼哇塞！

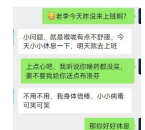
捡田螺的小男孩



开发神器

小小病毒，可笑可笑

yes的练级攻略



土木转行，历经7个月我的大数据上岸之路

涂生大数据

