0619 - 一起来聊聊性能优化这件事~

聊到性能的关键词

一. 先看看有哪些性能指标

常规指标

FP - first print： 首次绘制时间

FCP - first contented print： 首次有内容绘制时间

FMP - first meaningful print：首次有意义绘制时间

TTI - Time to Interactive：用户可交互时间

TTFB - time for first byte：网络请求耗时

DCL：DomContentLoaded

L：DOM onload

总下载时间

chrome 最新指标 web-vitals

LCP（Largest Contentful Paint）good - 2.5sec - middle - 4.0 sec - poor

FID（First Input Delay）首次输入延迟 good - 100ms - middle - 300ms - poor

CLS（Cumulative Layout Shift）good - 0.1- middle - 0.25 - poor

二. 如何获取这些指标

相关参数计算

代码工程方法获取

1. 性能面板，火箭图

lighthouse

2. window.performance

web-vitals

三. 从什么维度来剖析性能?

计算机的维度： I/O维度 - network

Memory 维度。

前端如何有效地扩容

CPU 维度。

如何有效减少密集计算

说一说你理解的性能优化

说一说你在项目做过哪些性能优化

雪碧图，

说一说，react项目，如何进行性能优化

你在项目实施过程中，是如何考虑性能的

你说一说，为啥 vue 性能比 react 好

你说一说，为啥VDOM快

以一般人的努力程度之低，还轮不到拼天赋的时候。

以一般人的代码质量之差，还轮不到拼框架的时候。

聊到性能的关键词

- **指标；**
- **平衡，取舍；ROI。**
- **维度；**

一. 先看看有哪些性能指标

常规指标

FP – first print： 首次绘制时间

首次绘制包括了任何用户自定义的背景绘制，它是将**第一个像素点绘制到屏幕**的时刻，

对于应用页面，用户在视觉上首次出现不同于跳转之前的内容时间点，或者说是页面发生第一次绘制的时间点。

FCP – first contented print： 首次有内容绘制时间

指浏览器完成渲染 DOM 中第一个内容的时间点，可能是文本、图像、SVG或者其他任何元素，此时用户应该在视觉上有直观的感受。

FMP – first meaningful print：首次有意义绘制时间

指页面关键元素渲染时间。这个概念并没有标准化定义，因为关键元素可以由开发者自行定义——究竟什么是“有意义”的内容，只有开发者或者产品经理自己了解。

TTI – Time to Interactive：用户可交互时间

顾名思义，也就是用户可以与应用进行交互的时间。一般来讲，我们认为是 domready 的时间，因为我们通常会在这时候绑定事件操作。如果页面中涉及交互的脚本没有下载完成，那么当然没有到达所谓的用户可交互时间。那么如何定义 domready 时间呢?

TTFB – time for first byte：网络请求耗时

TTFB是发出页面请求到接收到应答数据第一个字节所花费的毫秒数

DCL：DomContentLoaded

L：DOM onload

总下载时间

页面所有资源加载完成所需要的时间。一般可以统计 window.onload 时间，这样可以统计出同步加载的资源全部加载完的耗时。如果页面中存在较多异步渲染，也可以将异步渲染全部完成的时间作为总下载时间。

DOMContentLoaded 与 load 事件的区别

DOMContentLoaded 指的是文档中 DOM 内容加载完毕的时间，也就是说 HTML 结构已经完整。但是我们知道，很多页面包含图片、特殊字体、视频、音频等其他资源，这些资源由网络请求获取，DOM 内容加载完毕时，由于这些资源往往需要额外的网络请求，还没有请求或者渲染完成。**而当页面上所有资源加载完成后，load 事件才会被触发**。因此，在时间线上，load 事件往往会落后于 DOMContentLoaded 事件。

关于 DOMContentLoaded 和 domReady？

我们简单说一下，浏览器是从上到下，从左到右，一个个字符串读入，大致可以认为两个同名的开标签与闭标签就是一个DOM(有的是没有闭签)，这时就忽略掉它的两个标签间的内容。页面上有许多标签, 但标签会生成同样多的DOM，因为有的标签下只允许存在特定的子标签，比如tr下面一定是td,th, select下面一定是opgroup,option,而option下面，就算你写了，它都会忽略掉，option下面只存在文本，这就是我们需要自定义下拉框的缘故。

我们说过, 这顺序是从上到下, 有的元素很简单,会构建得很快,但标签存在src, href属性,它会引用外部资源,这就要区别对待了.比如说, script标签,它一定会等src指定的脚本文件加载下来,然后全部执行了里面的脚本,才会分析下一个标签.这种现象叫做堵塞.

堵塞是一种非常致命的现象,因为浏览器渲染引擎是单线程的,如果头部脚本过多过大会导致白屏,影响用户体验,因此雅虎的20军规就有一条提到 ,将所有script标签放到body之后.

此外, style标签与link标签,它们在加载样式文件时是不会堵塞,但它们一旦异步加载好,就立即开始渲染已经构建好的元素节点们, 这可能会引起reflow, 这也影响速度.

另一个影响DOM树构建的因此是iframe,它也会加载资源, 虽然不会堵塞DOM构建,但它由于是发出HTTP请求,而HTTP请求是有限,它会与父标签的其他需要加载外部资源的标签产生竞争。我们经常看到一些新闻网,上面会挂许多iframe广告, 这些页面一开始加载时就很卡,也是这缘故.

此外还有object元素, 用来加载flash

<script>

document.getElementById();

</script>

等等，这些东西都会影响到DOM树的构建过程.因此在这时候,当我们贸贸然,使用**getElementById**, getElementsByTagName获取元素,然后操作它们, 就会有很大机率碰到 元素为null的 异常. 这时, 目标元素还可以没有转换为DOM节点, 还只是一个普通的字符串呢!

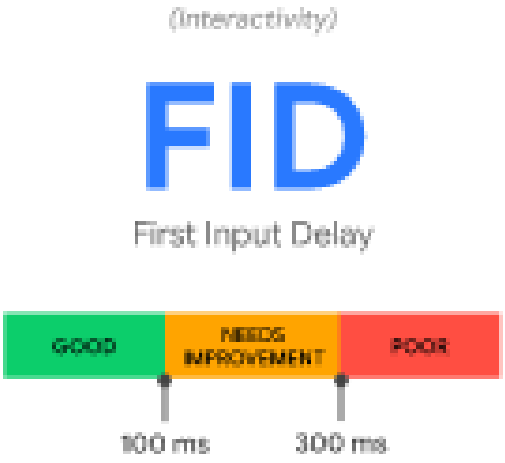
很早期, 浏览器提供了一个**window.onload**方法,但这东西是等到所有标签变成DOM,并且外部资源,图片,背景音乐什么都加载好才触发, 时间上有点晚.

幸好,浏览器提供了一个document.readyState属性,当它变成complete时,说明这时机到了

但这是一个属性,不是一个事件,需要使用不太精确的setInterval轮询

在标签浏览器, W3C终于绅士地提供了一个**DOMContentLoaded**事件把这件事解决了。

chrome 最新指标 web-vitals



LCP（Largest Contentful Paint）good – 2.5sec – middle – 4.0 sec – poor

衡量页面的加载体验，它表示视口内可见的最大内容元素的渲染时间。相比 FCP，这个指标可以更加真实地反映具体内容加载速度。比如，如果页面渲染前有一个 loading 动画，那么 FCP 可能会以 loading 动画出现的时间为准，而 LCP 定义了 loading 动画加载后，真实渲染出内容的时间。

FID（First Input Delay）首次输入延迟 good – 100ms – middle – 300ms – poor

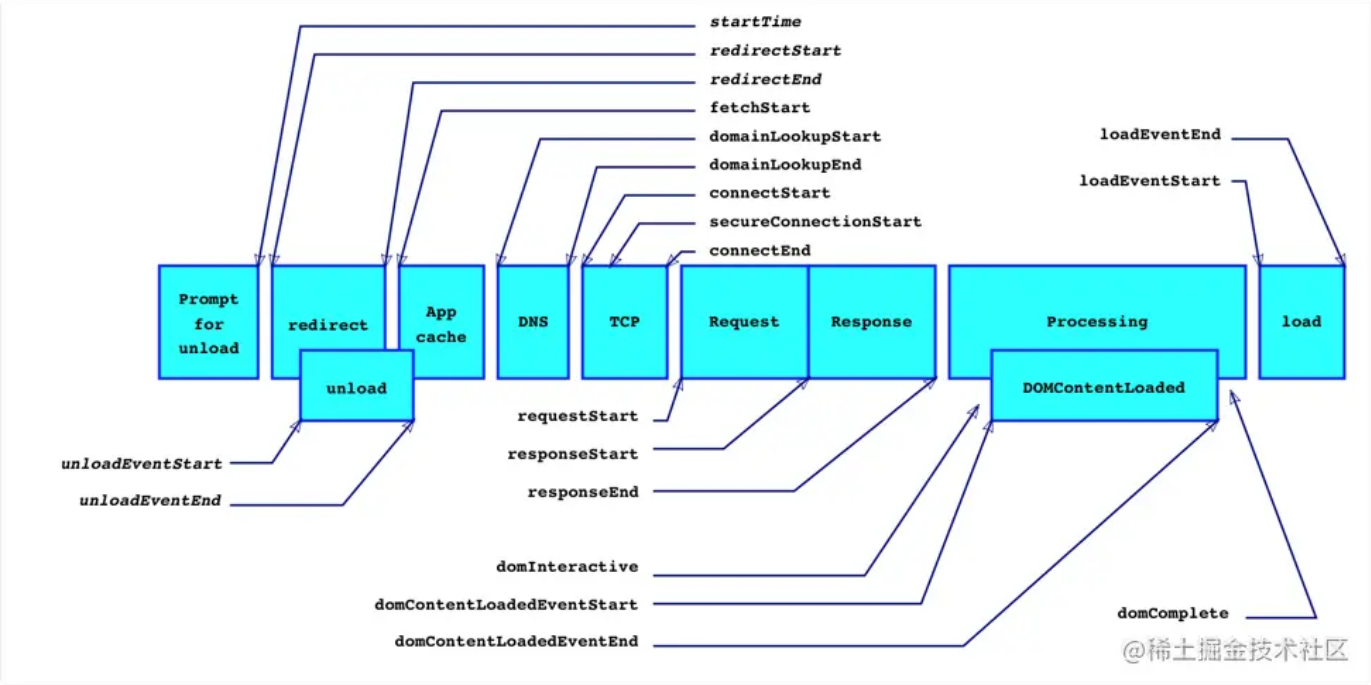
react 18 – 调度器。

衡量可交互性，它表示用户和页面进行首次交互操作所花费的时间。它比 TTI（Time to Interact）更加提前，这个阶段虽然页面已经显示出部分内容，但并不能完全具备可交互性，对于用户的响应可能会有较大的延迟。

CLS（Cumulative Layout Shift）good – 0.1– middle – 0.25 – poor

衡量视觉稳定性，表示页面的整个生命周期中，发生的每个意外的样式移动的所有单独布局更改得分的总和。所以这个分数当然越小越好。

二. 如何获取这些指标



字段	含义
navigationStart	加载起始时间，如果没有前一个页面的unload,则与fetchStart值相等
redirectStart	重定向开始时间（如果发生了HTTP重定向，每次重定向都和当前文档同域的话，就返回开始重定向的fetchStart的值。其他情况，则返回0）
redirectEnd	重定向结束时间（如果发生了HTTP重定向，每次重定向都和当前文档

	同域的话，就返回最后一次重定向接受完数据的时间。其他情况则返回0）
fetchStart	fetchStart 浏览器发起资源请求时，如果有缓存，则返回读取缓存的开始时间
domainLookupStart	DNS域名开始查询的时间,如果有本地的缓存或keep-alive等，则返回fetchStart
domainLookupEnd	domainLookupEnd 查询DNS的结束时间。如果没有发起DNS请求，同上
connectStart	TCP开始建立连接的时间,如果有本地的缓存或keep-alive等,则与fetchStart值相等
secureConnectionStart	https 连接开始的时间,如果不是安全连接则为0
connectEnd	TCP完成握手的时间，如果有本地的缓存或keep-alive等，则与connectStart 值相等
requestStart	HTTP请求读取真实文档开始的时间,包括从本地缓存读取
requestEnd	HTTP请求读取真实文档结束的时间,包括从本地缓存读取
responseStart	返回浏览器从服务器收到（或从本地缓存读取）第一个字节时的Unix毫秒时间戳
responseEnd	返回浏览器从服务器收到（或从本地缓存读取，或从本地资源读取）最后一个字节时的Unix毫秒时间戳
unloadEventStart	前一个页面的unload的时间戳 如果没有则为0
unloadEventEnd	与unloadEventStart相对应，返回的是unload函数执行完成的时间戳
domLoading	这是当前网页DOM结构开始解析时的时间戳，是整个过程的起始时间戳，浏览器即将开始解析第一批收到的 HTML 文档字节，此时document.readyState变成loading,并将抛出readyStateChange事件
domInteractive	返回当前网页DOM结构结束解析、开始加载内嵌资源时时间戳,document.readyState 变成interactive，并将抛出readyStateChange事件(注意只是DOM树解析完成,这时候并没有开始加载网页内的资源）
domContentLoaded	网页domContentLoaded事件发生的时间

EventStart	
domContentLoadedEventEnd	网页domContentLoaded事件脚本执行完毕的时间,domReady的时间
domComplete	DOM树解析完成,且资源也准备就绪的时间,`document.readyState`变成`complete`.并将抛出`readystatechange`事件
loadEventStart	load 事件发送给文档，也即load回调函数开始执行的时间
loadEventEnd	load回调函数执行完成的时间

相关参数计算

字段	描述	计算方式	意义
unload	前一个页面卸载耗时	unloadEventEnd — unloadEventStart	-
redirect	重定向耗时	redirectEnd — redirectStart	重定向的时间
appCache	缓存耗时	domainLookupStart — fetchStart	读取缓存的时间
dns	DNS 解析耗时	domainLookupEnd — domainLookupStart	可观察域名解析服务是否正常
tcp	TCP 连接耗时	connectEnd — connectStart	建立连接的耗时
ssl	SSL 安全连接耗时	connectEnd — secureConnectionStart	反映数据安全连接建立耗时
ttfb	Time to First Byte(TTFB)网络请求耗时	responseStart — requestStart	TTFB是发出页面请求到接收到应答数据第一个字节所花费的毫秒数
response	响应数据传输耗时	responseEnd —	观察网络是否正常

		responseStart	
dom	DOM解析耗时	domInteractive — responseEnd	观察DOM结构是否合理，是否有JS阻塞页面解析
dcl	DOMContentLoaded 事件耗时	domContentLoadedEventEnd — domContentLoadedEventStart	当 HTML 文档被完全加载和解析完成之后，DOMContentLoaded事件被触发，无需等待样式表、图像和子框架的完成加载
resources	资源加载耗时	domComplete — domContentLoadedEventEnd	可观察文档流是否过大
domReady	DOM阶段渲染耗时	domContentLoadedEventEnd — fetchStart	DOM树和页面资源加载完成时间，会触发domContentLoaded事件
首次渲染耗时	首次渲染耗时	responseEnd–fetchStart	加载文档到看到第一帧非空图像的时间，也叫白屏时间
首次可交互时间	首次可交互时间	domInteractive–fetchStart	DOM树解析完成时间，此时document.readyState为interactive
首包时间耗时	首包时间	responseStart–domainLookupStart	DNS解析到响应返回给浏览器第一个字节的时间
页面完全加载时间	页面完全加载时间	loadEventStart – fetchStart	–
onLoad	onLoad事件耗时	loadEventEnd — loadEventStart	

代码工程方法获取

1. 性能面板，火箭图

lighthouse

写一个简单的性能测试 node 服务

JavaScript | 复制代码

```
const fs = require('fs');
const lighthouse = require('lighthouse');
const chromeLaucher = require('chrome-launcher');

(
    async () => {
        // 启动一个 Chrome
        const chrome = await chromeLaucher.launch();
        const options = {
            logLevel: 'info',
            output: 'html',
            onlyCategories:['performance'],
            port: chrome.port
        }
        // 使用 Lighthouse 对页面进行一个计算
        const res = await lighthouse('https://www.baidu.com/', options);
        const { report } = res;
        // 对报告写入文件
        fs.writeFileSync('repost.html', report);
        console.log('Report is done for ', res.lhr.finalUrl);
        await chrome.kill();
    }
)()
```

2. window.performance

JavaScript | 复制代码

```javascript
console.log("Welcome luyi");

// LCP,  FP,  FCP
new PerformanceObserver((entryList, observer) => {
    let entries = entryList.getEntries();
    for(let i = 0; i < entries.length; i++) {
        if(entries[i].name === 'first-paint') {
            console.log(`FP: ${entries[i].startTime}ms`)
        }
        if(entries[i].name === 'first-contentful-paint') {
            console.log(`FCP: ${entries[i].startTime}ms`)
        }
    }
    observer.disconnect()

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

new PerformanceObserver((entryList, observer) => {
    let entries = entryList.getEntries();
    const lastEntry = entries[entries.length -1];
    console.log(`LCP: ${lastEntry.startTime}`);
    observer.disconnect()
})
.observe({ entryTypes:['largest-contentful-paint']});

setTimeout(() => {
    const {
        fetchStart,
        connectEnd,
        connectStart,
        requestStart,
        responseStart,
        responseEnd,
        domLoading,
        domInteractive,
        domContentLoadedEventStart,
        domContentLoadedEventEnd,
        loadEventStart
    } = performance.timing;

    console.log(`
    connectTime(TCP连接耗时): ${connectEnd - connectStart}
    ttfbTime: ${responseStart - requestStart}
    responseTime(Response响应耗时): ${responseEnd - responseStart}
```

```
        parseDOMTime(DOM 解析耗时): ${loadEventStart - domLoading}
        DCL: ${domContentLoadedEventEnd - domContentLoadedEventStart}
        TTI: ${domInteractive - fetchStart}
        loadTime: ${loadEventStart - fetchStart}
        `)
    },2000)
```

web-vitals

https://web.dev/vitals/

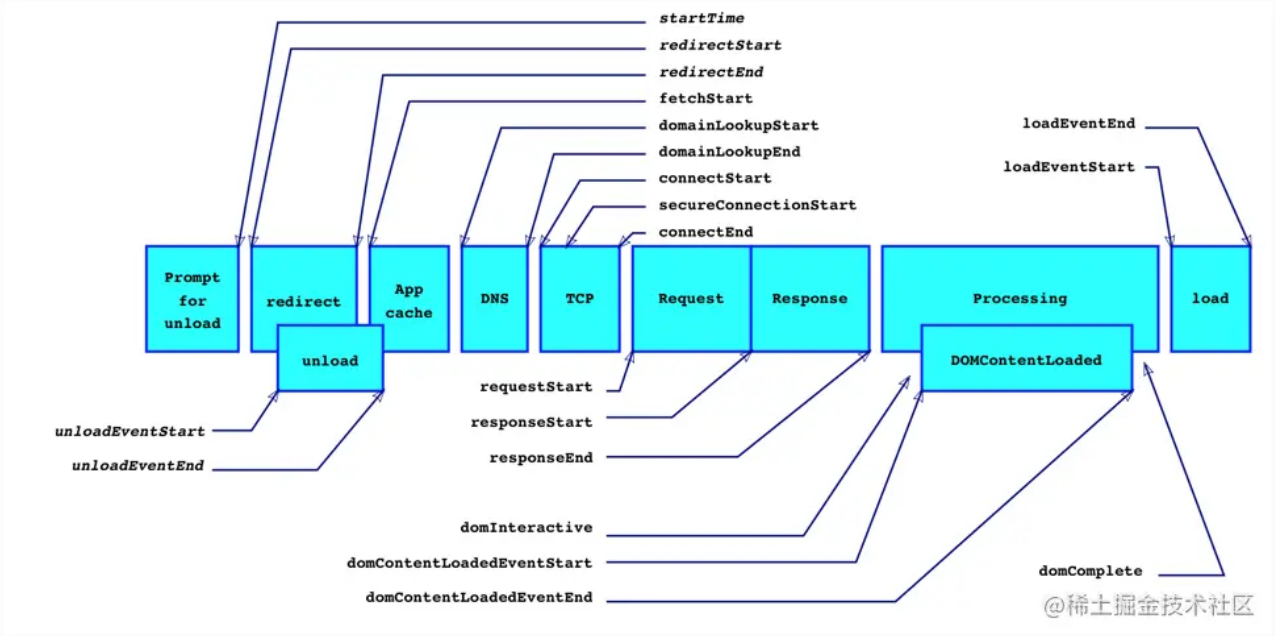
JavaScript | 复制代码

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

function reportToAnalytics(data) {
  const body = JSON.stringify(data);
  (navigator.sendBeacon && navigator,sendBeacon('/analytics', body)) ||
    fetch('/analytics', { body, method:'POST', keepalive: true });
}

getCLS(metric => reportToAnalytics({cls: metric.value}))
getFID(metric => reportToAnalytics({fid: metric.value}))
getLCP(metric => reportToAnalytics({lcp: metric.value}))
```

三. 从什么维度来剖析性能?

分类和正交。

计算机的维度： I/O维度 – network



App Cache

如何缓存，如何有效地缓存。

http 缓存

- 强缓存
  - expries
  - cache-control
    - no-cache
- 协商缓存
  - last-modified -- if-modified-since
  - etag -- if-none-match

缓存中有哪些细节要注意?

- 直接在浏览器输入 http:xxx/xxx/xx.html 该文件是不会被缓存的；
- webpack 的hash 指纹，让该缓存的缓存
  - hash
  - content hash
  - chunk hash
- modified 是根据时间判断，1s的时间，可能引发很多问题。
- 在 cdn 下，hash 缓存是否有比较好的效果。
- 没有了强缓存的必要字段值，浏览器还会走缓存吗？答案是肯定的（启发式缓存）。

缓存是一个很大的概念。

关键是：**数据在业务中的生命周期是什么样子?**

deps: {'a_b_c': 10};

DNS阶段

TCP阶段

- http 1.0
- http 1.1
- http 2
- http 3

RES REQ 阶段

1. 包体积，如何缩减到极致？？？
    a. ugfily; minify;
    b. runtime; 按需引入 polyfill -- @babel/preset-env {useBuiltIn:'entry'}
    c. tree shaking;
2. 首屏加载的内容，如何进行分解?
    a. code splitting
3. 如何在tcp 数量和请求之间做 tradeoff
    a. chrome 最大6个并发
4. processing阶段前

浏览器一般是怎么加载文件的?

js : async / defer -- 并行下载。

顺序。。。

Memory 维度。

Vue2 -- Vue3

闭包、

监听器记得删除。

前端如何有效地扩容

- iframe 无限 localStorage
- indexedDB
- 享元模式
- http2 key – value.

CPU 维度。

如何有效减少密集计算

跟后端撕逼

BFF

密集计算必须要发生

WASM

worker

需要处理JS执行的长任务

React 18

RAF , RIdelC

GPU/渲染 维度

transform + 复杂动画

CLS 指标

fragment 插入。

四. 一些比较好的文章

节流和防抖相关

- Debouncing and Throttling Explained Through Examples
- 谈谈 JS 中的函数节流
- JavaScript 函数节流和函数防抖之间的区别
- 高性能滚动 scroll 及页面渲染优化
- 从 lodash 源码学习节流与防抖
- 理解并优化函数节流 Throttle

浏览器引擎渲染性能相关

- Inside look at modern web browser
- How Browsers Work: Behind the scenes of modern web browsers
- How browsers work
- How browser rendering works—behind the scenes
- What Every Frontend Developer Should Know About Webpage Rendering
- 前端文摘：深入解析浏览器的幕后工作原理
- 从 Chrome 源码看浏览器如何加载资源
- 浏览器内核渲染：重建引擎
- 体现工匠精神的 Resource Hints
- 浏览器页面渲染机制，你真的弄懂了吗
- 前端不止：Web 性能优化 — 关键渲染路径以及优化策略
- 浏览器前端优化
- 浅析前端页面渲染机制
- 浅析渲染引擎与前端优化
- 渲染性能
- Repaint 、Reflow 的基本认识和优化 (2)

动画性能相关

- Timing control for script-based animations
- Gain Motion Superpowers with requestAnimationFrame
- CSS Animation 性能优化

- GSAP的动画快于 jQuery 吗？为何?
- Javascript 高性能动画与页面渲染
- 也许你不知道，JS animation 比 CSS 更快！
- 渐进式动画解决方案
- 你应该知道的 requestIdleCallback
- 无线性能优化：Composite
- 优化动画卡顿：卡顿原因分析及优化方案
- 一篇文章说清浏览器解析和 CSS（GPU）动画优化

实战案例相关

- Building the Google Photos Web UI
- A Netflix Web Performance Case Study
- The Cost Of JavaScript In 2018
- How we reduced our initial JS/CSS size by 67%
- Front-End Performance Checklist 2019
- 网站性能优化实战——从 12.67s 到 1.06s 的故事
- 前端黑科技：美团网页首帧优化实践
- Web 字体图标-自动化方案
- JS 加载慢？谷歌大神带你飞!
- 前端性能优化（三） 移动端浏览器前端优化策略
- CSS @font-face 性能优化
- 移动 Web 性能优化从入门到进阶
- 记一次惊心动魄的前端性能优化之旅