Lecture 13
Performance

Play

What happened?

- Service is too busy

Why Website Performance Matters?

- 40% Abandon a site after waiting 3 seconds for a page to load.
- 80% of the people will NOT return
- Almost half will tell others about their negative experience.
- Amazon: 1% Revenue increase for every 100MS of improvement

How Website Performance Affects Shopping Behavior

Website Speed and SEO

- Website speed is important to your SEO efforts because faster websites are:
  - Easier to crawl
  - Easier to access
  - More likely to rank (although this is marginal)
  - Most importantly, more likely to keep visitors around!

Web 性能优化
web performance optimization (WPO)

- 网站越快,用户的黏性越高; 参与度
- 网站越快,用户忠诚度更高; 留存率
- 网站越快,用户转化率越高。

延迟与带宽

- 延迟
  - 分组从信息源发送到目的地所需的时间。
- 带宽
  - 逻辑或物理通信路径最大的吞吐量。
-

延迟的构成要素

- 传播延迟 消息从发送端到接收端需要的时间,是信号传播距离和速度的函数
- 传输延迟 把消息中的所有比特转移到链路中需要的时间,是消息长度和链路速率的函数
- 处理延迟 处理分组首部、检查位错误及确定分组目标所需的时间
- 排队延迟 到来的分组排队等待处理的时间

Web性能要点

- 延迟和带宽对Web性能的影响;
- 传输协议(TCP)对HTTP的限制;
- HTTP协议自身的功能和缺陷;
- Web应用的发展趋势及性能需求;
- 浏览器局限性和优化思路。

- 不同层之间总是相互依赖， 但优化方式却有很多可能性。任何优化建议和最佳做法都不是一成不变的，涉及的每个要素都是动态发展的:
  - 浏览器越来越快
  - 用户上网条件不断改善
  - Web应用的功能和复杂度也与日俱增

浏览器页面渲染机制

https://blog.logrocket.com/how-browser-rendering-works-behind-the-scenes-6782b0e8fb10

html
Node
body
p
p
Hello,
World

Bytes => Characters => Tokens => Node => DOM

从 CSS 的原始字节到 CSSOM

Bytes => Characters => Tokens => Node => DOM

渲染树

渲染树

布局

- 回流（reflow）

渲染阻塞资源

- 渲染阻塞（render-blocking）
  - “有东西阻止了屏幕上节点的实际绘制”
- 应该尽快将 HTML 和 CSS 提供给客户端，以优化应用程序的首次渲染时间。

JavaScript 如何执行?

- 每当浏览器遇到脚本标签时，DOM 构造就会暂停！整个 DOM 构建过程都将停止，直到脚本执行完成。
  - JavaScript 可以同时修改 DOM 和 CSSOM
- 在默认情况下，每个脚本都是一个解析器阻断器！
  - async

关键渲染路径

- 周密的关键渲染路径（CRP）优化策略使浏览器能够通过确定优先加载的资源以及资源加载的顺序来尽可能快地加载页面。
-

回流

- 当渲染树中部分或者全部元素的尺寸、结构或者属性发生变化时，浏览器会重新渲染部分或者全部文档的过程就称为回流。
- 下面这些操作会导致回流：
  - 页面的首次渲染
  - 浏览器的窗口大小发生变化
  - 元素的内容发生变化
  - 元素的尺寸或者位置发生变化
  - 元素的字体大小发生变化
  - 激活CSS伪类
  - 查询某些属性或者调用某些方法
  - 添加或者删除可见的DOM元素

- 在触发回流（重排）的时候，由于浏览器渲染页面是基于流式布局的，所以当触发回流时，会导致周围的DOM元素重新排列，它的影响范围有两种：
  - 全局范围：从根节点开始，对整个渲染树进行重新布局
  - 局部范围：对渲染树的某部分或者一个渲染对象进行重新布局

重绘

- 当页面中某些元素的样式发生变化，但是不会影响其在文档流中的位置时，浏览器就会对元素进行重新绘制，这个过程就是重绘。
- 下面这些操作会导致回流：
  - color、background 相关属性：background-color、background-image 等
  - outline 相关属性：outline-color、outline-width 、text-decoration
  - border-radius、visibility、box-shadow
- 注意： 当触发回流时，一定会触发重绘，但是重绘不一定会引发回流。

减少回流与重绘的措施

- 操作DOM时，尽量在低层级的DOM节点进行操作
- 不要使用table布局， 一个小的改动可能会使整个table进行重新布局
- 使用CSS的表达式
- 不要频繁操作元素的样式，对于静态页面，可以修改类名，而不是样式。
- 使用absolute或者fixed，使元素脱离文档流，这样他们发生变化就不会影响其他元素
- 避免频繁操作DOM，可以创建一个文档片段documentFragment，在它上面应用所有DOM操作，最后再把它添加到文档中
- 将元素先设置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。
- 将DOM的多个读操作（或者写操作）放在一起，而不是读写操作穿插着写。这得益于浏览器的渲染队列机制。

Anatomy of a Modern Web Application

- 2013 年初，一个普通的 Web 应用由下列内容构成。
  - 90 个请求，发送到 15 个主机，总下载量 1311 KB
    - HTML:10 个请求，52 KB
    - 图片:55 个请求，812 KB
    - JavaScript:15 个请求，216 KB
    - CSS:5 个请求，36 KB
    - 其他资源:5 个请求，195 KB

http://httparchive.org

https://httparchive.org/reports/page-weight#bytesTotal

Including

HTML Transfer Size & HTML Requests
70
60
50
40
30
20
10
0
52kB 52kB 67kB 67kB 67kB 65kB 64kB 58kB 65kB 54kB 53kB 53kB
8.8 9.1 9.3 9.3 13.7 11.8 11.7 12.2 9.8 9.2 9.7 10.1
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
HTML Transfer Size (kB)
HTML Requests

CSS Transfer Size & CSS Requests
80
70
60
10
0
73kB 74kB 76kB 77kB 71kB 71kB 70kB 76kB 77kB 75kB 74kB 76kB
7.5 7.5 7.7 7.7 6.9 6.9 6.8 7.1 7.2 7.2 7.1 7.2
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
CSS Transfer Size (kB)
CSS Requests

JS Transfer Size & JS Requests
500
400
300
200
40
30
20
10
358kB 368kB 357kB 362kB 405kB 398kB 387kB 402kB 390kB 406kB 401kB 410kB
22 22 22 22 22 21 21 22 22 22 23 23
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
JS Transfer Size (kB)
JS Requests

Image Transfer Size & Image Requests
2000
1500
1000
90
80
70
60
50
40
1408kB 1421kB 1426kB 1431kB 1413kB 1424kB 1425kB 1522kB 1535kB 1562kB 1577kB 1650kB
54 55 54 54 59 57 57 60 55 55 56 57
11/15 12/15 1/16 2/16 3/16 4/16 5/16 6/16 7/16 8/16 9/16 10/16
Image Transfer Size (kB)
Image Requests

Time and user perception

Delay	User perception
0–100 ms	Instant
100–300 ms	Small perceptible delay
300–1000 ms	Machine is working
1,000+ ms	Likely mental context switch
10,000+ ms	Task is abandoned

Analyzing the Resource Waterfall

http://www.webpagetest.org/

Connection View

Performance Pillars: Computing, Rendering, Networking

- The execution of a web program primarily involves three tasks: fetching resources, page layout and rendering, and JavaScript execution.
  - More Bandwidth Doesn't Matter (Much)
  - Latency as a Performance Bottleneck

Benchmarking Utilities

- Apache Bench：apache自带的一款功能强大的测试工具。
- Siege一款开源的压力测试工具，可以根据配置对一个WEB站点进行多用户的并发访问，记录每个用户所有请求过程的相应时间，并在一定数量的并发访问下重复进行。
  - http://www.joedog.org/JoeDog/Siege
- http_load (Excellent for latency tests)
  - 程序非常小，解压后也不到100K
  - http_load以并行复用的方式运行，用以测试web服务器的吞吐量与负载。
  - http://www.acme.com/software/http_load/
- 阿里云性能测试（Performance Testing）：一个SaaS性能测试平台，具有强大的分布式压测能力，可模拟海量用户真实的业务场景，
- Tsung：一个开源的支持多协议的分布式压力测试工具
  - 目前支持HTTP分布式压力测试、WebDAV分布式压力测试、SOAP分布式压力测试、PostgreSQL分布式压力测试、MySQL分布式压力测试、LDAP分布式压力测试、MQTT分布式压力测试、Jabber/XMPP servers分布式压力测试
- JMeter：
  - 作为一款广为流传的开源分布式压测产品，能自动生成图形报告。最初被设计用于Web应用测试，如今JMeter可以用于测试静态和动态资源，例如静态文件、Java 小服务程序、CGI 脚本、Java 对象、数据库、FTP服务器等等，还能对服务器、网络或对象模拟巨大的负载，通过不同压力类别测试它们的强度和分析整体性能。另外，JMeter能够对应用程序做功能测试和回归测试。

Affecting Your Benchmark Figures

- Geographical location
- Network issues
- Response size
- Code processing
- Browser behavior
- Web server configuration

问题

- 人造测试不能发现所有性能瓶颈。
  - 场景及页面选择:很难重复真实用户的导航模式;
  - 浏览器缓存:用户缓存不同，性能差别很大;
  - 中介设施:中间代理和缓存对性能影响很大;
  - 硬件多样化:不同的 CPU、GPU 和内存比比皆是;
  - 浏览器多样化:各种浏览器版本，有新有旧;
  - 上网方式:真实连接的带宽和延迟可能不断变化。

性能监控指标

- FP（全称“First Paint”，“首次绘制”）：对于应用页面，首次出现视觉上不同于跳转之前内容的时间点，或者说是页面发生第一次绘制的时间点。
- FCP（全称“First Contentful Paint”，“首次内容绘制”）：指浏览器完成渲染DOM中第一部分内容（可能是文本、图像或其他任何元素）的时间点，此时用户应该在视觉上有直观的感受。
  - 注意：只有首次绘制文本、图片（包含背景图）、非白色的canvas或SVG时才被算作FCP。
- FP与FCP这两个指标之间的主要区别是：
  - FP是当浏览器开始绘制内容到屏幕上的时候，只要在视觉上开始发生变化，无论是什么内容触发的视觉变化，在这一刻，这个时间点，叫做FP。
  - 相比之下，FCP指的是浏览器首次绘制来自DOM的内容。例如：文本，图片，SVG，canvas元素等，这个时间点叫FCP。
  - FP和FCP可能是相同的时间，也可能是先FP后FCP。
- FMP（全称“First Meaningful Paint”，翻译为“首次有意义绘制”）：指页面关键元素的渲染时间。
  - 没有标准化定义，因为关键元素可以由开发者自行定义。
  - FMP本质上是通过一个算法来猜测某个时间点可能是FMP，所以有时候不准。

性能监控指标

- 首屏时间：进入页面之后，应用渲染完成整个手机屏幕（未滚动之前）内容的时间。
  - 业界对于这个指标没有确切定论，比如是否包含屏幕内图片的渲染完成时间。
- 用户可交互时间：用户可以与应用进行交互的时间。
  - 一般来说，是DOMReady的时间，因为通常会在这时绑定事件操作。
  - 如果页面中涉及交互的脚步没有下载完成，那么当然没有到达所谓的用户可交互时间。

- 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的异步渲染，那么可以将异步渲染全部完成的时间做为总下载时间。
- 自定义指标：由于应用特点不同，可以根据需求自定义时间。
  - 比如，一个类似Instagram的页面由图片瀑布流组成，那么可能非常关心屏幕中第一排图片渲染完成的时间。

Navigation Timing 2

W3C Web Performance Working Group

- DNS 查询耗时 = domainLookupEnd - domainLookupStart
- TCP 连接耗时 = connectEnd - connectStart
- 请求响应耗时 = responseEnd - requestStart
- 白屏时间 = domLoading - navigationStart
- 首屏时间 = loadEventStart - navigationStart
- DOM 树解析耗时 = domInteractive - domLoading
- Domready 时间 = domComplete - navigationStart
- onload 时间 = loadEventEnd - navigationStart

- Performance 接口可以获取到当前页面中与性能相关的信息。它是 High Resolution Time API 的一部分，同时也融合了 Performance Timeline API、Navigation Timing API、 User Timing API 和 Resource Timing API。
- 该类型的对象可以通过调用只读属性 Window.performance 来获得。
- 功能强大，但并不适用于所有场景。
  - 比如，如果在单页应用中改变URL但不刷新页面（单页应用的典型路由方案），那么使用window.performance.timing所获取的数据是不会更新的，还需要开发者重新设计统计方案。
  - window.performance.timing可能无法满足一些自定义的数据。

分析工具

- 基于网页分析工具：
  - Web PageTest, http://www.webpagetest.org/
  - PingDom Tools
  - GTmetrix, https://gtmetrix.com/
  - Google PageSpeed, https://developers.google.com/speed/
  - whatsmydns, https://www.whatsmydns.net
- 基于浏览器分析工具：
  - Chrome自带工具F12
  - Firefox插件：YSlow（Yahoo工具）
  - Page Speed（google）

浏览器优化

- 可行的优化手段会因浏览器而异，但从核心优化策略来说，可以宽泛地分为两类：
  - 基于文档的优化：熟悉网络协议，了解文档、CSS 和 JavaScript 解析管道，发现和优先安排关键网络资源，尽早分派请求并取得页面，使其尽快达到可交互的状态。主要方法是优先获取资源、提前解析等。
  - 推测性优化：浏览器可以学习用户的导航模式，执行推测性优化，尝试预测用户的下一次操作。然后，预先解析DNS、预先连接可能的目标。

大多数浏览器利用的四种技术

- 所有这些优化都由浏览器替我们自动完成，经常可以节省几百 ms 的网络延迟。既然如此，那理解这些优化背后的原理就至关重要了，这样才能利用浏览器的这些特性，提升应用性能。大多数浏览器都利用了如下四种技术。
  - 资源预取和排定优先次序：文档、CSS 和 JavaScript 解析器可以与网络协议层沟通，声明每种资源的优先级:初始渲染必需的阻塞资源具有最高优先级，而低优先级的请求可能会被临时保存在队列中。
  - DNS预解析：对可能的域名进行提前解析，避免将来HTTP请求时的DNS延迟。预解析可以通过学习导航历史、用户的鼠标悬停，或其他页面信号来触发。
  - TCP预连接：DNS 解析之后，浏览器可以根据预测的HTTP请求，推测性地打开 TCP 连接。 如果猜对的话，则可以节省一次完整的往返(TCP 握手)时间。
  - 页面预渲染：某些浏览器可以让我们提示下一个可能的目标，从而在隐藏的标签页中预先渲染整个页面。这样，当用户真的触发导航时，就能立即切换过来。

speculative optimization

- 预解析特定的域名
  - <link rel="dns-prefetch" href="//hostname_to_resolve.com">
- 预取得页面后面要用到的关键性资源
  - <link rel=“subresource" href="/javascript/myapp.js">
- 预取得将来导航要用的资源
  - <link rel=“prefetch" href="/images/big.jpeg">
- 根据对用户下一个目标的预测，预渲染特定页面
  - <link rel="prerender"href="//example.org/next_page.html">

Optimizing Application Delivery

最佳实践

- 两个准则
  - 消除或减少不必要的网络延迟
  - 将需要传输的数据压缩至最少。

性能规则

- 减少DNS查找
  - 每一次主机名解析都需要一次网络往返,从而增加请求的延迟时间,同时还会阻塞后续请求。
- 重用TCP连接
  - 尽可能使用持久连接,以消除 TCP 握手和慢启动延迟。
- 减少HTTP重定向
  - HTTP 重定向极费时间,特别是不同域名之间的重定向,更加费时;这里面既有额外的 DNS 查询、TCP 握手,还有其他延迟。最佳的重定向次数为零。
- 使用CDN(内容分发网络)
  - 把数据放到离用户地理位置更近的地方,可以显著减少每次 TCP 连接的网络延 迟,增大吞吐量。这一条既适用于静态内容,也适用于动态内容。
- 去掉不必要的资源
  - 任何请求都不如没有请求快。

更多规则

- 在客户端缓存资源
  - 应该缓存应用资源, 从而避免每次请求都发送相同的内容。
- 传输压缩过的内容
  - 传输前应该压缩应用资源, 把要传输的字节减至最少:确保对每种要传输的资源采用最好的压缩手段。
- 消除不必要的请求开销
  - 减少请求的 HTTP 首部数据(比如 HTTP cookie),节省的时间相当于几次往返的延迟时间。
- 并行处理请求和响应
  - 请求和响应的排队都会导致延迟,无论是客户端还是服务器端。这一点经常被忽视,但却会无谓地导致很长延迟。
- 针对协议版本采取优化措施
  - HTTP 1.x 支持有限的并行机制, 要求打包资源、跨域分散资源等等。相对而言, HTTP 2.0只要建立一个连接就能实现最优性能, 同时无需针对HTTP 1.x 的那些优化方法。

客户端缓存

- Cache-Control 首部用于指定缓存时间（max-age)。
- Last-Modified 和 ETag 首部提供验证机制。

压缩传输的数据

- HTML、CSS 和 JavaScript 等文本资源的大小经过 gzip 压缩平均可以减少 60%~80%。
- 图片则需要仔细考量:
  - 图片一般会占到一个网页需要传输的总字节数的一半;
  - 通过去掉不必要的元数据可以把图片文件变小;
  - 要调整大小就在服务器上调整,避免传输不必要的字节;
  - 应该根据图像选择最优的图片格式;
  - 尽可能使用有损压缩。

消除不必要的请求字节

- 浏览器会在每个请求中自动附加关联的 cookie 数据;
- 在 HTTP 1.x 中,包括 cookie 在内的所有 HTTP 首部都会在不压缩的状态下传输;
- 在 HTTP 2.0 中,这些元数据经过压缩了,但开销依然不小;
- 最坏的情况下,过大的 HTTP cookie 会超过初始的 TCP 拥塞窗口,从而导致多余的网络往返。

并行处理请求和响应

- 为了让应用响应速度达到最快,应该尽可能第一时间就分派所有资源请求。
- 要是想实现最佳性能,就要记住以下几点:
  - 使用持久连接,从 HTTP 1.0 升级到 HTTP 1.1;
  - 利用多个 HTTP 1.1 连接实现并行下载;
  - 可能的情况下利用 HTTP 1.1 管道;
  - 考虑升级到 HTTP 2.0 以提升性能;
  - 确保服务器有足够的资源并行处理请求。
- 识别造成客户端和服务器延迟的不必要资源既是艺术也是技术: 要仔细检查客户端资源瀑布, 以及服务器日志。常见问题如下:
  - 服务器资源不足,造成不必要的资源处理延迟;
  - 代理及负载均衡器容量不足,造成向应用服务器交付请求的延迟(请求排队);
  - 客户端资源阻塞导致页面构建延迟。

针对HTTP 1.x的优化

- 针对 HTTP 1.x 的优化次序很重要：首先要配置服务器以最大限度地保证 TCP 和 TLS 的性能最优，然后再谨慎地选择和采用移动及经典的应用最佳实践,之后再度量，迭代。
- 采用了经典的应用优化措施和适当的性能度量手段,还要进一步评估是否有必要为应用采取特定于 HTTP 1.x 的优化措施(其实是权宜之计)。
  - 利用HTTP管道
    - 如果你的应用可以控制客户端和服务器这两端,那么使用管道可以显著减少网络延迟。
  - 采用域名分区
    - 如果你的应用性能受限于默认的每来源 6 个连接,可以考虑将资源分散到多个来源。
  - 打包资源以减少HTTP请求
    - 拼接和精灵图等技巧有助于降低协议开销，又能达成类似管道的性能提升。
  - 嵌入小资源
    - 考虑直接在父文档中嵌入小资源，从而减少请求数量。
- 管道缺乏支持,而其他优化手段又各有各的利弊。事实上,这些优化措施如果过于激进或使用不当,反倒会伤害性能。总之,要有务实的态度,通过度量来评估各种措施对性能的影响,在此基础上再迭代改进。

针对HTTP 2.0的优化

- HTTP 2.0 的主要目标就是提升传输性能,实现客户端与服务器间较低的延迟和较高的吞吐量。显然,在 TCP 和 TLS 之上实现最佳性能,同时消除不必要的网络延迟, 从来没有如此重要过。
  - 服务器应该通过 ALPN(针对 SPDY 则为 NPN)协商支持 TLS;
    - ALPN（Application Layer Protocol Negotiation，应用层协议协商），ALPN 是客户端发送所支持的 HTTP 协议列表，由服务端选择；
    - NPN（Next Protocol Negotiation，下一代协议协商） 是服务端发送所支持的 HTTP 协议列表，由客户端选择。
  - 服务器应该支持 TLS 恢复以最小化握手延迟。
- 要通过 HTTP 2.0 获得最佳性能,特别是从每个来源仅用一个连接的角度说,的确需要各层协议的紧密配合。 采用移动及其他经典的最佳做法: 少发数据、削减请求,根据无线网络情况调整资源供给。不管使用什么版本的协议,减少传输的数据量和消除不必要的网络延迟,对任何应用都是最有效的优化手段。
- 最后,杜绝和忘记域名分区、文件拼接、图片精灵等不良的习惯,这些做法在 HTTP 2.0 之上完全没有必要。事实上,继续使用这些手段反而有害!可以利用 HTTP 2.0 内置的多路分发以及服务器推送等新功能。

去掉对1.x的优化

- 每个来源使用一个连接
  - HTTP 2.0 通过将一个 TCP 连接的吞吐量最大化来提升性能。事实上,在 HTTP 2.0 之下再使用多个连接(比如域名分区)反倒成了一种反模式,因为多个连接会抵消新协议中首部压缩和请求优先级的效用。
- 去掉不必要的文件合并和图片拼接
  - 打包资源的缺点很多,比如缓存失效、占用内存、延缓执行,以及增加应用复杂性。有了 HTTP 2.0,很多小资源都可以并行发送,导致打包资源的效率反而更低。
- 利用服务器推送
  - 之前针对 HTTP 1.x 而嵌入的大多数资源,都可以而且应该通过服务器推送来交付。这样一来,客户端就可以分别缓存每个资源,并在页面间实现重用,而不必 把它们放到每个页面里了。
- 要获得最佳性能,应该尽可能把所有资源都集中在一个域名之下。域名分区在 HTTP 2.0 之下属于反模式,对发挥协议的性能有害:分区是开始,之后影响会逐渐扩散。打包资源不会影响 HTTP 2.0 协议本身,但对缓存性能和执行速度有负面影响。

双协议应用策略

- 相同的应用代码,双协议部署
  - 相同的应用代码可能通过 HTTP 1.x 也可能通过 HTTP 2.0 交付。可能任何一种协议之下都达不到最佳性能,但可以追求性能足够好。所谓足够好,需要通过针对每一种应用单独度量来保证。这种情况下,第一步可以先撤销域名分区以实现 HTTP 2.0 交付。然后,随着更多用户迁移到 HTTP 2.0,可以继续撤销资源打包并尽可能利用服务器推送。
- 分离应用代码,双协议部署
  - 根据协议不同分别交付不同版本的应用。这样会增加运维的复杂性,但实践中对很多应用倒是十分可行。比如,一台负责完成连接的边界服务器可以根据协商后的协议版本,把客户端请求引导至适当的服务器。
- 动态HTTP 1.x和HTTP 2.0优化
  - 某些自动化的 Web 优化框架,以及开源及商业产品,都可以在响应请求时动态重写交付的应用代码(包括连接、拼合、分区等等)。此时,服务器也可以考虑协商的协议版本,并动态采用适当的优化策略。
- HTTP 2.0,单协议部署
  - 如果应用可以控制服务器和客户端,那没理由不只使用 HTTP 2.0。事实上,如果真有这种可能,那就应该专一使用 HTTP 2.0。

- 谷歌的 PageSpeed Optimization Libraries(PSOL)提供了40 多种“Web 优化过滤 器”的开源实现,可以集成到任何服务器运行时,动态应用各种优化策略。

Translating 1.x to 2.0 and Back

WebpageTest

结果

Web page performance test result for
https://www.nju.edu.cn

From: London, UK - EC2 - **Chrome** - **Cable**
2021/12/20 下午3:15:55

E	A	A	A
Security score	First Byte Time	Keep-alive Enabled	Compress Transfer

F	F	X
Compress Images	Cache static content	Effective use of CDN

Summary Details Web Vitals Performance Content Domains Processing Screenshot Image Analysis Request Map

First View only
Test runs: **3**

Re-run the test

Export Files

Performance Results

MEDIAN RUN BASED ON SPEEDINDEX

					Web Vitals			Document Complete			Fully Loaded			
	First Byte	Start Render	First Contentful Paint	Speed Index	Largest Contentful Paint	Cumulative Layout Shift	Total Blocking Time	Time	Requests	Bytes In	Time	Requests	Bytes In	Cost
First View (Run 3)	1.842s	3.900s	3.874s	6.604s	16.718s	0.037	0.097s	68.244s	63	29,242 KB	68.549s	64	29,243 KB	$$$$$

指标

- 1. 网页级指标(Page-level Metrics)
  - 这些是为整个页面捕获并显示的顶级度量值。
- 2. 整页加载时间(Load Time)
  - 测量的时间是从初始化请求，到开始执行window.onload事件。
- 3. 页面所有元素加载时间(Fully Loaded)
  - 从初始化请求，到Document Complete后，2秒内（中间几百毫秒轮询）没有网络活动的时间，但这2秒是不包括在测量中的，所以会出现两个差值大于或小于2秒。
- 4. 第一个字节加载时间(First Byte)
  - 第一个字节时间（通常缩写为TTFB）被测量为从初始化请求，到服务器响应的第一个字节，被浏览器接收的时间（不包括DNS查询、TCP连接的时间）。
- 5. 页面渲染时间(Start Render)
  - 测量的时间是从初始化请求，到第一个内容被绘制到浏览器显示的时间。在瀑布图中有两个参数指标Start Render和msFirstPaint。
    - Start Render是通过捕获页面加载的视频，并在浏览器第一次显示除空白页之外的其他内容时查看每个帧来衡量的。它只能在实验室测量，通常是最准确的测量。
    - msFirstPaint（IE专用属性）是由浏览器本身报告的一个测量，它认为绘制的第一个内容。通常是相当准确，但有时它报告的时候，浏览器只画一个空白屏幕。
- 6. 首屏展现平均值(Speed Index)
  - 表示页面呈现用户可见内容的速度（越低越好）。有关如何计算的更多信息，请参见：Speed Index。
- 7. DOM元素数量(DOM Elements)
  - 在测试结束时测试页面上的DOM元素的计数。
- 8. 请求级度量标准(Request-level Metrics)
  - 这些是为每个请求捕获和显示的度量。

Real User Data from CrUX

▼ This test, First View

INDIVIDUAL RUNS

Run 1:

	Waterfall	Screenshot	Video
First View (71.248s) **Timeline (view)** **Processing Breakdown** **Trace (view)**			Filmstrip View · Watch Video

Content Breakdown

Filmstrip View

Tested From: London, UK - EC2 - **Chrome** - **Cable**

Compare all runs

Filmstrip key:

- Visual change
- Visual change + Layout Shift
- Largest Contentful Paint
- Largest Contentful Paint + Layout Shift

Adjust Filmstrip Settings

CPU Utilization
Bandwidth In (0 - 5,000 Kbps)
Browser Main Thread
Long Tasks

Visual Progress (%)
100
80
60
40
20
0
0.00
2.60
5.20
7.80
10.40
13.00
15.60
18.20
20.80
23.40
26.00
28.60
31.20
33.80
36.40
39.00
41.60
44.20
46.80
49.40
52.00
54.60
Time (seconds)
www.nju.edu.cn

A Netflix Web Performance Case Study

- Improving Time-To-Interactive for Netflix.com on Desktop
- 通过改进 Netflix.com 注册过程中所使用的 JavaScript 及预加载技术，开发团队能够为移动用户和桌面用户提供更好的用户体验，主要改进如下。
  - 加载和交互时间减少了 50%（Netflix.com 桌面登录主页）；
  - 在从 React 和其他客户端库切换到普通的 JavaScript 之后，JavaScript 包大小减少了 200KB。服务器端仍然使用 React。
  - HTML、CSS、JavaScript（React）预加载使后续页面的浏览交互时间减少了 30%。

https://medium.com/dev-channel/a-netflix-web-performance-case-study-c0bcde26a9d9

减少 JavaScript 传输，缩短交互时间

- Netflix 针对其登录主页的性能进行了优化
  - 这个页面最初包含 300KB 的 JavaScript，其中一些是 React 和其他客户端代码（比如像 Lodash 这样的实用程序库），还有一些是补充 React 状态所需的上下文数据。
- 使用 Chrome 的开发工具和 Lighthouse 模拟在 3G 连接上加载登录主页，结果显示，登录主页需要 7 秒的加载时间，对于一个简单的登录页面来说太长了，因此需要研究改进的可能。通过一些性能审计，Netflix 发现他们客户端的 JS 开销很高。
- 登录主页是否真得需要 React?
  - 用Vanilla JavaScript 替换！！！

优化

- 移植到原生 JavaScript 的组件列表：
  - 基础交互（主页选项卡）
  - 语言切换器
  - “Cookie 横幅（Cookie banner）”（针对非美国用户）
  - 客户端日志分析
  - 性能度量和记录
  - 广告归属检测引导代码（出于安全考虑，放在沙箱式 iFrame 中）

结果

- 尽管 React 最初占用的空间仅为 45KB，但将 React、几个库和相应的应用程序代码从客户端移除后，JavaScript 的总量减少了 200KB 以上，这使得 Netflix 在登录主页的交互时间减少了 50% 以上。

后续页面的 React 预加载

- 预加载
  - 通过浏览器内置的 API 和 XHR 预加载
  - 交互时间减少了 30%

NETFLIX

COMPARISON OF PREFETCHING TECHNIQUES

	Browser API (`<link />`)	XHR
Browser Support	Partial	Full
Success Rate	30%-90%, depending on browser	> 95%
Ease of Implementation	1 line of code	3 lines of code
Can Prefetch HTML Document	Yes	No

总结

- 密切关注 JavaScript 的开销
- Netflix 的折中方案是，使用 React 在服务器端渲染登录页面，但同时也为注册过程的其他部分预取 React 代码。这不仅优化了首次加载性能，还优化了注册过程其余部分的加载时间，因为它是单页应用，所以有更大的 JS 包需要下载。
- 补充：
  - Netflix 考虑过 Preact，但是，对于一个交互性比较低的简单页面流，使用普通的 JavaScript 是一个更简单的选择。
  - Netflix 尝试使用 Service Workers 进行静态资源缓存。当时，Safari 不支持这个 API（现在支持了），但他们现在又在探索这个 API。Netflix 的注册过程需要比客户体验更多的遗留浏览器支持。许多用户会在旧的浏览器上注册，但会在他们本地的移动应用程序或电视设备上观看 Netflix。
  - Netflix 的登录页面极为动态。这是他们的注册过程中进行 A/B 测试最多的页面，机器学习模型用于根据位置、设备类型和许多其他因素定制消息和图像。支持近 200 个国家，每个派生页面都面对着不同的本地化、法律和价值信息挑战。

References

- https://developers.google.com/web/fundamentals/performance/why-performance-matters/
- https://www.webpagetest.org
- https://developers.google.com/web/fundamentals/performance/speed-tools/
- https://github.com/pwstrick/WebPagetest-Docs
- High performance browser networks

- 高性能网站建设指南
- 高性能网站建设进阶指南
- 大型网站技术架构－核心原理与案例分析
-

Thanks!!!