最快的 Google Fonts

Harry Roberts 前端之巅 2020-06-05

作者 | Harry Roberts
译者 | 王强
策划 | 李俊辰

> 一般来说，今天的 Web 字体速度比以往任何时候都更快。浏览器厂商在进一步规范 FOUT/FOIT 行为，新一代 font-display 规范也开始普及，看来性能（和用户体验）终于成为了人们关注的焦点。

本文最初发布于 csswizardry.com 网站，经原作者授权由 InfoQ 中文站翻译并分享。

FOUT：无样式文本闪现，Flash of Unstyled Text

FOIT：不可见文本闪现，Flash of Invisibale Text

FOFT：伪文本闪现，Flash of Faux Text

人们一般认为自托管字体是最快的选项：相同来源意味着较少的网络协商，可预测的 URL 意味着我们可以 preload（预加载），自托管意味着我们可以设置自己的 cache-control（缓存控制）指令，而完整所有权可以减少将静态资产留在第三方源上的风险。

然而，像谷歌字体（Google Fonts）这种服务的便利性也是不可小觑的。他们能够为特定用户代理和平台量身打造体积最小巧的字体文件，再加上由谷歌 CDN 网络分发的规模庞大的免费库等优势，难怪越来越多的用户在转向谷歌字体了。

本网站（csswizardry）极度重视性能表现，所以我完全放弃了 Web 字体，而选择使用访客的系统字体。这种方案速度飞快，可以适应各种各样的设备平台，并且工程开销几乎为零。但在 harry.is 这个网站上，我很想抛弃理性，随心所欲一次。面对恐惧吧！这个网站上我要用 Web 字体！

然而，我还是不能牺牲性能。

在网站的原型设计阶段，我转向了谷歌字体。它们最近添加了通过一个 URL 参数（&display=swap）支持 font-display 的功能，所以速度应该是很快的。然后我有了一个主意。

首先看这条推文：

虽然 font-display: swap 效果出众，但还有很多事情都不能让我满意。我能再做些什么来进一步提升谷歌字体的性能呢？

结果，我给自己挖了个大坑……

测试

我针对 harry.is 和 csswizardry.com 主页运行了相同的测试套件。首先测试的是 harry.is，因为我在这个网站上使用了谷歌字体，但它的页面太简单了，没什么代表性。因此我把 CSS Wizardry 的主页复制了几份，分别实现了几种不同的方法。本文的每个小节中我都会列出两个站点的测试结果。我的方法有：

- 旧版：使用谷歌字体，不启用 font-display。
- font-display: swap;：使用谷歌字体的新默认设置。
- 异步 CSS：异步加载谷歌字体文件。
- preload：预加载 CSS 文件以提升其优先级。
- preconnect：使用我自己准备的 fonts.gstatic.com 源。

此外，每种方法都是上一种方法的扩展——也就是在上一种方法的基础上加入新内容。我不会只尝试 preload 或异步，因为这样做毫无意义——我们知道，各种选项结合起来使用才会有最好的效果。

每次测试时，我会捕获以下指标：

- **First Paint（FP）**：第一次绘制完成所需的时间，关键路径受到多大程度的影响？
- **First Contentful Paint（FCP）**：Web 字体也好还是其他东西也罢，我们要多久才能开始看到网页内容？
- **First Web Font（FWF）**：第一个 Web 字体何时加载完毕？
- **Visually Complete（VC）**：何时完成全部工作（是 Last Web Font 的一种代理指标，但它们并不等价）？
- **Lighthouse 分数**：指标怎么能缺了 Lighthouse 呢？

注意： 所有测试都是使用一个私有 WebPageTest 实例进行的（WebPageTest 现在下线了，因此我无法使用公共实例，也就是说我无法共享任何 URL，抱歉）。具体的配置文件是 Samsung Galaxy S4 over 3G。

为了让代码段更易读，我将用 $CSS 替换所有的实例：

https://fonts.googleapis.com/css2?family=Roboto:ital,wght@0,400;0,700;1,400;1,700

默认 / 旧版

谷歌字体在过去一年间有了明显的进步。默认情况下，新创建的谷歌字体代码段均带有 &display=swap 参数，该参数将 font-display: swap; 注入所有 @font-face @规则。参数的值可以是 swap、optional、fallback 或 block。MDN 上有更多信息：

https://developer.mozilla.org/en-US/docs/Web/CSS/@font-face/font-display

但我的基线会关闭 font-display。许多站点可能还在使用这种旧格式，所以用它做基线比较合适。

代码段：

```
<link rel="stylesheet"
      href="$CSS" />
```

这里有两个关键问题：

1. 一个第三方源上同步的，进而阻塞渲染的 CSS 文件。
2. 一个包含 @font-face @规则的文件，没有 font-display 描述符。

这是同步中的同步，不是什么好事。

Results (s) – harry.is:

FP	FCP	FWF	VC	Lighthouse
3.4	4.6	4.9	5.0	98

Results (s) – CSS Wizardry:

FP	FCP	FWF	VC	Lighthouse
3.4	4.3	4.4	4.4	96

上面就是我们的基线成绩。谷歌字体文件是两个测试中唯一阻塞渲染的外部资源，并且我们可以看到两个测试的 first paint 结果完全一致。在本文这些实验的过程中，让我很惊讶的一件事就是谷歌字体的一致性表现。

这里，Lighthouse 给出了一个错误和一个警告：

- （错误）确保在 Webfont 加载期间文本仍然可见
- （警告）移除阻塞渲染的资源

前者是因为没有 Web 字体加载方案（例如 font-display）；后者是因为同步的谷歌字体 CSS 文件。

现在我们一点点加入改动，希望能获得一些改进。

font-display: swap;

这一节中我加回来了 &display=swap。它的效果是让字体文件本身变成异步的——浏览器会先显示我们的回退（fallback）文本，等 Web 字体可用时再切换过去。也就是说我们不会让用户看到任何不可见的文本（FOIT），从而带来更快、更舒适的体验。

注意：如果你能精心定义一个在过渡期间显示的合适回退，效果会更好——如果 Open Sans 字体就绪之前，用户看到满页面都是 Times New Roman 字体，这种体验就会很糟糕了。还好我们有 Monica 这个工具，让这里的工作变得轻松愉快。她的她的字体样式匹配器非常好用：

https://meowni.ca/font-style-matcher/

代码段：

```
<link rel="stylesheet"
      href="$CSS&display=swap" />
```

Results – harry.is:

	FP	FCP	FWF	VC	Lighthouse
	3.4	3.4	4.5	5.2	99
Change from Baseline:	0	−1.2	−0.4	+0.2	+1

Results – CSS Wizardry:

	FP	FCP	FWF	VC	Lighthouse
	3.6	3.6	4.6	4.6	95
Change from Baseline:	+0.2	−0.7	+0.2	+0.2	−1

我们没有从关键路径中移除任何阻塞渲染的资源，所以 first paint 应该不会有任何改善。实际上，虽然 harry.is 的成绩没变，但 CSS Wizardry 的速度慢了 200 毫秒。但与此同时，我们看到 **first contentful paint 成绩显著提升**，在 harry.is 上提升了超过 1 秒！**在 harry.is 上 first web font 的成绩进步了**，但在 csswizardry.com 上却没有。Visually complete 速度慢了 200 毫秒。

我很高兴能看到最关键的几个指标快了 700–1200 毫秒。

尽管这种方法确实可以大大改善渲染 Web 字体的速度，但它仍然是在一个同步 CSS 文件中定义的——这一步能带来的改进也就这么多了。

预料之中，Lighthouse 现在只发出一个警告：

- （警告）移除阻塞渲染的资源

那么下一步就是解决同步 CSS 文件的问题。

引用一下我自己的话："如果你要在谷歌字体中使用 font-display，那么就应该异步加载整个请求链。"如果我能让 CSS 文件的内容异步处理，那么还让 CSS 文件本身完全同步就显得很傻了。

font-display: swap; 是个好主意。

异步 CSS

引入 Critical CSS 时，一项关键技术就是让 CSS 异步处理。要做到这一点可用的方法很多，但我敢说，最简单，最常用的就是 Filament Group 的 print media type 技巧：

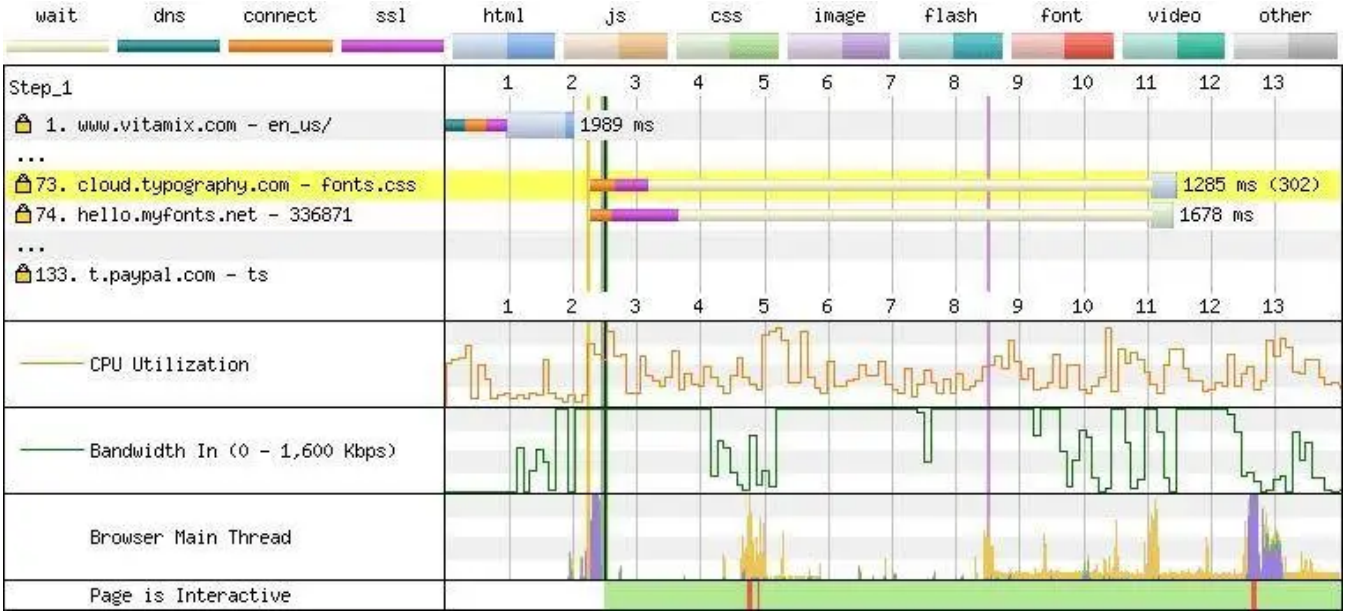
https://www.filamentgroup.com/lab/load-css-simpler/

这种方法将隐式告诉浏览器以非阻塞方式加载 CSS 文件，仅将样式应用于 print 上下文。但当文件传输过来时，我们会告诉浏览器将其应用于 all 上下文，为页面的剩余部分应用样式。

代码段：

```
<link rel="stylesheet"
      href="$CSS&display=swap"
      media="print" onload="this.media='all'" />
```

尽管这个技巧非常简单，**但我长期以来一直对它持保留态度。**可以看到，常规的同步样式表会阻塞渲染，因此浏览器会为其分配“最高”优先级。但 print 样式表（或任何与当前上下文不匹配的样式表）分配的优先级就完全相反了：Idle。也就是说，在浏览器开始发出请求时，异步 CSS 文件往往会以非常低的优先级来处理（或者说，它的优先级是正确的，但比你想的低很多）。以我的一个客户 Vitamix 为例，我为他们实现了异步 CSS，字体提供商是他们自己的：



虽然 Chrome 可以执行异步 DNS/TCP/TLS，但它会在较慢的连接上序列化并停止非关键请求。

浏览器完全按照我们的指示做事：要求这些 CSS 文件具有 print 样式表该有的优先级。因此在 3G 网络连接上，要花费超过 9 秒才能下载完各个文件。浏览器将其他几乎所有内容（包括 body 内的资源）都放在了 print 样式表前面。也就是说页面在 3G 网络上要等足足 12.8 秒才能渲染好第一个自定义字体。

还好在使用 Web 字体时，这个问题并不是无解的：

- 不管怎样 Web 字体应该是一种增强技术，因此我们需要一种基础方案；
- 我们可以并且应当设计适当的回退，在 Web 字体不可用时展示给用户；另外
- 如果我们预计会有如此严重的延迟，则应使用 font-display: optional;。

但对于 fold CSS 中的内容来说，将近 10 秒的延迟是不可接受的——用户肯定不会等这么久，早就去看其他内容了。

如果我们异步加载谷歌字体，会发生什么情况呢？

Results – harry.is:

	FP	FCP	FWF	VC	Lighthouse
	1.8	1.8	4.5	5.1	100
Change from Baseline:	−1.6	−2.8	−0.4	+0.1	+2
Change from Previous:	−1.6	−1.6	0	−0.1	+1

Results – CSS Wizardry:

	FP	FCP	FWF	VC	Lighthouse
	1.7	2.2	4.9	5.0	99
Change from Baseline:	−1.7	−2.1	+0.5	+0.6	+3
Change from Previous:	−1.9	−1.4	+0.3	+0.4	+4

结果非常出色。

我对这些结果感到非常满意。与我们的基线相比，**first paint 大幅改善了 1.6-1.7 秒**，而 CSS Wizardry 上则 **比上一节中提升了 1.9 秒。First contentful paint 改进了 2.8 秒之多**，并且 **Lighthouse 分数首次达到 100。**

就关键路径而言，这是一个巨大的胜利。

但是——这个"但是"不可忽视——由于 CSS 文件的优先级降低了，CSS Wizarddry 的 **first web font 比我们的基线慢了 500 毫秒之多。**这就是代价所在。

异步加载谷歌字体是一个不错的主意，但是 CSS 文件的优先级下降实际上减慢了自定义字体的渲染速度。

我要说的是，异步 CSS 总体来说是个好主意，但我需要以某种方式缓解优先级问题。

preload

如果 print CSS 的优先级太低，我们需要的就是高优先级的异步提取（fetch）。该轮到 preload 上场了。

在上一节方法的基础上加入 preload 后，我们就能两全其美了：

1. 在几乎所有现代浏览器中都有高优先级的异步提取，并且；
2. 是一种获得广泛支持的重新应用异步加载 CSS 的方法。

注意：我们不能完全依赖 preload，因为它还没有获得足够的支持。实际上在撰写本文时，大约有 20%的网站用户无法使用它。可以考虑将 print 样式表作为回退。

代码段：

```
<link rel="preload"
      as="style"
      href="$CSS&display=swap" />
<link rel="stylesheet"
      href="$CSS&display=swap"
      media="print" onload="this.media='all'" />
```

注意：将来，我们应该可以使用优先级提示（Priority Hints）来解决这个问题。

Results – harry.is:

	FP	FCP	FWF	VC	Lighthouse
	1.8	1 .8	4.5	5.3	100
Change from Baseline:	−1.6	−2.8	−0.4	+0.3	+2
Change from Previous:	0	0	0	+0.2	0

Results – CSS Wizardry:

	FP	FCP	FWF	VC	Lighthouse
	2.0	2.0	4.3	4.3	98
Change from Baseline	−1.4	−2.3	−0.1	−0.1	+2
Change from Previous	+0.3	−0.2	−0.6	−0.7	−1

虽然 first paint 的成绩没变或变慢了，但 **first contentful paint 的成绩则变快了或至少没变**；在 CSS Wizardry 的测试中，**first web font 比上一个迭代的速度快了 600 毫秒**。

在 harry.is 的测试中各项指标和上一节相比没什么变化。**Visually complete 成绩提高了 200 毫秒**，但所有 first- 指标均未受影响。由于 harry.is 的页面非常小巧，因此 print 样式表没什么负面影响——其优先级的变化并不会有什么效果可言。

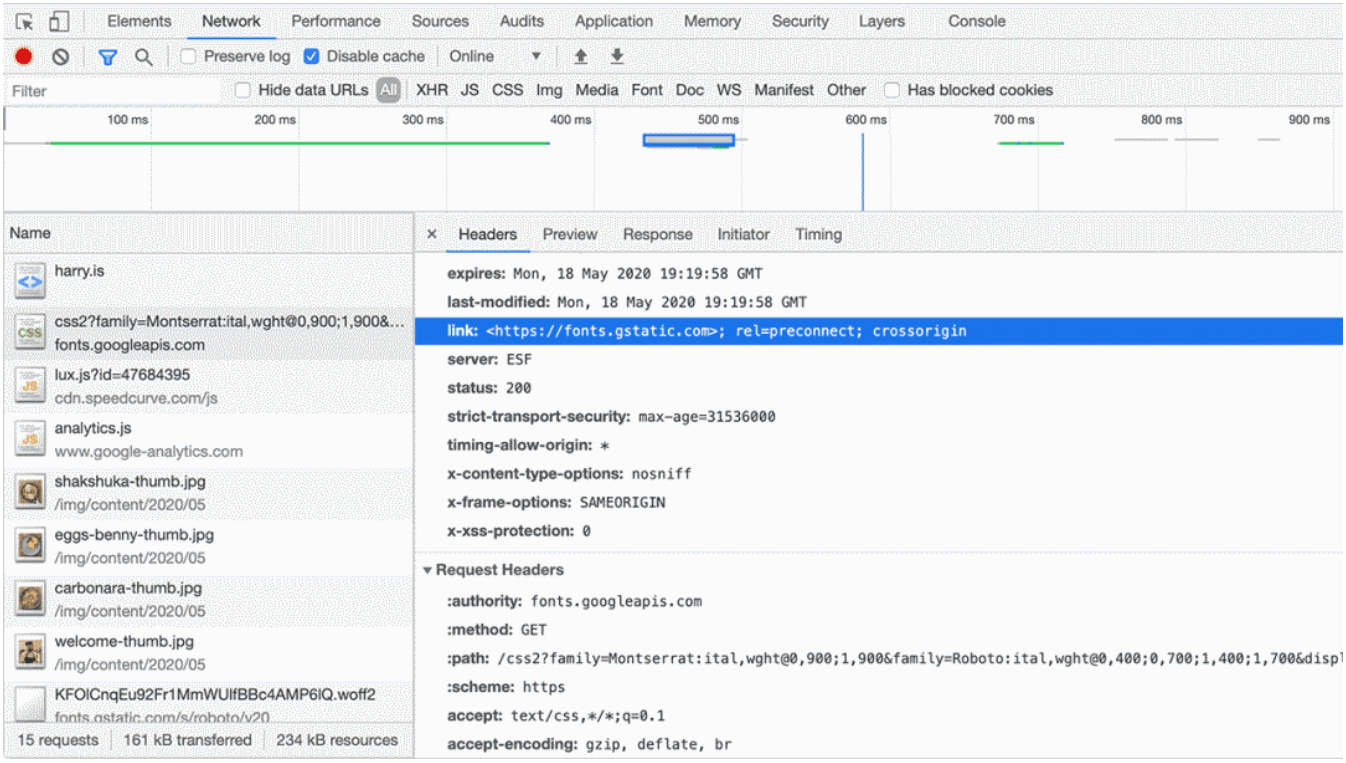
在 CSS Wizardry 的测试中，我们意外地看到 first paint 要慢 300 毫秒，不过这也无关紧要（这里没有阻塞渲染的 CSS，因此更改异步 CSS 文件的优先级可能没有任何影响——我就当它是测试中的异常了）。让人高兴的是，**first contentful paint 的成绩提升了 200 毫秒，first web font 的速度提高了 600 毫秒，而 visually complete 的速度提高了 700 毫秒**。

preload 谷歌字体是一个好主意。

preconnect

接下来是最后一个问题。当我们为 CSS 链接到 fonts.googleapis.com 时，字体文件本身却托管在 fonts.gstatic.com 上。遇到高延迟连接时这就是大问题了。

谷歌字体很贴心——它们会通过附在 fonts.googleapis.com 响应上的一个 HTTP 标头抢先 preconnect fonts.gstatic.com 这个源：



尽管在这些演示中效果不尽如人意，但我希望更多的第三方提供商可以做到这一点。

但是，这个标头的执行受到响应的 TTFB（Time to First Byte）时间约束，在高延迟网络中 TTFB 可能会非常高。在所有测试中，谷歌字体 CSS 文件的 TTFB（包括请求队列、DNS、TCP、TLS 和服务器时间）的中位数为 1406 毫秒。相比之下，CSS 文件的平均下载时间仅为 9.5 毫秒——到达文件标头所花费的时间是下载文件本身所花费时长的 148 倍。

换句话说，即使谷歌为我们 preconnect 了 fonts.gstatic.com 源，他们也只能获得 10 毫秒的优势。也就是说这个文件遇到了延迟瓶颈，而不是带宽瓶颈：

https://csswizardry.com/2019/01/bandwidth-or-latency-when-to-optimise-which/

如果我们实现了第一方的 preconnect，那么我们应该能获得显著的收益。下面就看看实测表现。

代码段：

```
<link rel="preconnect"
      href="https://fonts.gstatic.com"
      crossorigin />
<link rel="preload"
```

```
      as="style"
      href="$CSS&display=swap" />
<link rel="stylesheet"
      href="$CSS&display=swap"
      media="print" onload="this.media='all'" />
```

我们可以在 WebPageTest 中直观地看到收益：

看看我们通过 preconnect 前移了多少连接开销。

Results – harry.is:

	FP	FCP	FWF	VC	Lighthouse
	1.8	1.8	3.8	4.4	100
Change from Baseline	−1.6	−2.8	−1.1	−0.6	+2
Change from Previous	0	0	−0.7	−0.9	0

Results – CSS Wizardry:

	FP	FCP	FWF	VC	Lighthouse
	1.9	1.9	3.5	3.6	99
Change from Baseline	−1.5	−2.4	−0.9	−0.8	+3
Change from Previous	−0.1	−0.1	−0.8	−0.7	+1

First（contentful）paint 的成绩其实没受影响。此处的任何更改都与我们的 preconnect 无关，因为 preconnect 仅影响关键路径之后的资源。我们关注的是 first web font 和 visually complete 的成绩，这两个指标都显示出巨大的进步。与我们之前的版本相比，**first web font 的改进为 700–1,200 毫秒，visually complete 的改进为 700–900 毫秒**；与我们的基线相比则 **分别为 600–900 毫秒和 600–800 毫秒。Lighthouse 分数分别是 100 和 99。**

preconnect fonts.gstatic.com 是一个好主意。

font-display: optional;

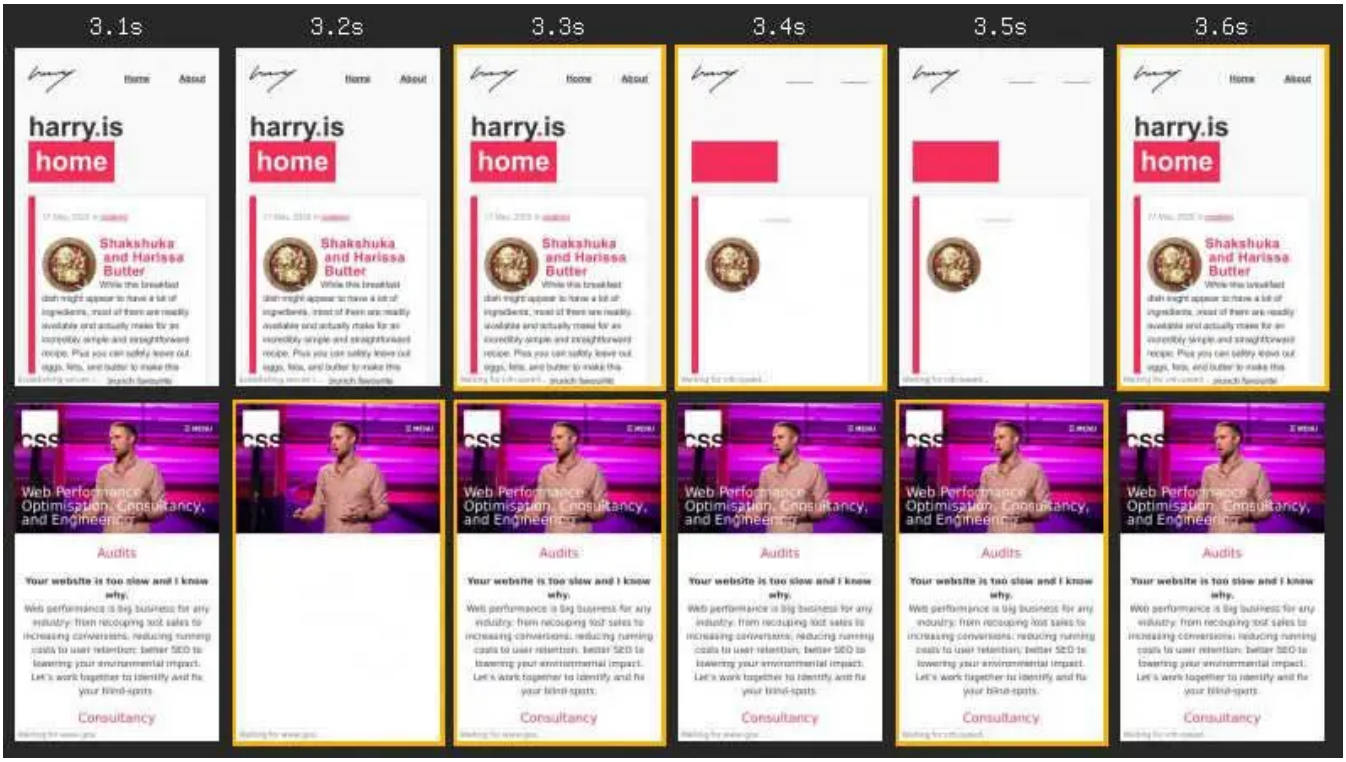
使用异步 CSS 和 font-display 会使我们容易受到 FOUT 的影响（或者如果我们正确设计了回退，就会是 FOFT）。为了缓解这种问题，我决定使用 font-display: optional; 做一次测试。

这种方法会告诉浏览器：Web 字体被认为是可选的，如果在"极短的阻塞时间"内我们无法获取字体文件，则我们"不提供切换期"。实际的效果是，如果 Web 字体加载时间太长，那么页面视图就完全不会用它。这样就能避免 FOUT，从而为用户带来更稳定的体验——他们

不会在页面浏览的途中看到文本样式出现变化——以及更好的累积布局偏移（Cumulative Layout Shift）分数。

但事实证明，在使用异步 CSS 时这个方法总是很麻烦。当 print 样式表变成 all 样式表时，浏览器将更新 CSSOM，然后将其应用于 DOM。此刻，页面被告知它需要一些 Web 字体，进入"极短的阻塞时间"，在页面加载生命周期中显示一个 FOIT。更糟糕的是，浏览器将用和一开始时一样的回退来替代 FOIT，因此用户甚至无法体验新字体的好处。它看起来就像是一个错误。

一图胜千言，以下是整个过程的屏幕截图：



注意 3.4-3.5 秒和 3.2 秒时的 FOIT

DevTools 中关于这个问题的视频：

https://csswizardry.com/wp-content/uploads/2020/05/video-devtools-foit.mp4

我不建议同时使用 font-disply: optional; 和异步 CSS。总的来说，FOUT 加上非阻塞 CSS 总比没必要的 FOIT 要强。

对比与视频

在下面这些慢动作视频中，你可以清楚地看到不同方法之间的差异。

harry.is

https://csswizardry.com/wp-content/uploads/2020/05/video-comparison-harry.is.mp4

- 异步、preload 和 preconnect 都在 1.8 秒开始渲染。
  - 这也代表了它们的 first contentful paint——first render 中的有用信息。
- 旧版和 swap 都在 3.4 秒开始渲染。
  - 但旧版没有任何文字——FOIT。
- preconnect 的 Web font 在 3.8 秒加载。
  - visually complete 为 4.4 秒。
- Legacy 在 4.5 秒钟内完成了 first contentful 和 first web font paint。
  - 所有都是同步的，所以同时完成。
- 旧版的 visually complete 为 5 秒。
- 异步 CSS 的 visually complete 为 5.1 秒。
- swap 是 5.2 秒。
- preload 在 5.3 秒 visually complete。

CSS Wizardry

https://csswizardry.com/wp-content/uploads/2020/05/video-comparison-csswizardry.com.mp4

- 异步 CSS 在 1.7 秒开始渲染。
- preconnect 在 1.9 秒开始渲染。

    - 它的 first contentful paint 也是 1.9 秒。
- preload 在 2 秒开始渲染。
    - 它的 first contentful paint 也是 2 秒。
- 异步 CSS 在 2.2 秒完成渲染。
- 旧版在 3.4 秒开始渲染。
- swap 在 3.6 秒完成 first contentful paint。
- preconnect 的 visually complete 是 3.6 秒。
- 旧版在 4.3 秒完成了 first contentful paint。
- preload 在 4.3 秒 visually complete。
- 旧版在 4.4 秒 visually complete。
- swap 完成于 4.6 秒。
- 异步 CSS 排在最后，是 5 秒。
- preconnect 的所有指标都是最快的。

结论

尽管自托管 Web 字体可能是解决性能和可用性问题的最佳方法，但我们可以设计一些相当灵活的措施来缓解使用谷歌字体时出现的许多相关问题。

异步加载 CSS、异步加载字体文件、选择 FOFT、快速提取异步 CSS 文件以及预热外部域等方法的组合，能获得比基线快几秒钟的成绩。

如果你是谷歌字体用户，强烈建议你采用这套方法。

如果谷歌字体不是你唯一的渲染阻塞资源，并且你违反了其他快速 CSS 原则（例如 @import 导入谷歌字体 CSS 文件），结果还会有差异。如果谷歌字体是你最大的性能瓶

颈，那么这些优化措施的效果就会非常明显了。

谷歌字体异步代码段

本文介绍了很多技巧，但是它们结合起来生成的代码仍然很轻巧且容易维护，不会带来问题。这一段代码无需拆分，可以全部保存在文档的中。

以下是用于高性能谷歌字体的最佳代码段：

```
<!--
  - 1. Preemptively warm up the fonts' origin.
  -
  - 2. Initiate a high-priority, asynchronous fetch for the CSS file. Works in
  - most modern browsers.
  -
  - 3. Initiate a low-priority, asynchronous fetch that gets applied to the page
  - only after it's arrived. Works in all browsers with JavaScript enabled.
  -
  - 4. In the unlikely event that a visitor has intentionally disabled
  - JavaScript, fall back to the original method. The good news is that,
  - although this is a render-blocking request, it can still make use of the
  - preconnect which makes it marginally faster than the default.
  -->
<!-- [1] -->
<link rel="preconnect"
      href="https://fonts.gstatic.com"
      crossorigin />
<!-- [2] -->
<link rel="preload"
      as="style"
      href="$CSS&display=swap" />
<!-- [3] -->
<link rel="stylesheet"
      href="$CSS&display=swap"
      media="print" onload="this.media='all'" />
<!-- [4] -->
<noscript>
  <link rel="stylesheet"
        href="$CSS&display=swap" />
</noscript>
```

延伸阅读

https://csswizardry.com/2020/05/the-fastest-google-fonts/

喜欢此内容的人还喜欢

闲鱼正在悄悄放弃 Flutter 吗?

前端之巅

这绯闻，就离谱!

新浪娱乐

“妈妈都是为了你好”，这句话真的毁了我的童年

费加罗夫人