

1. Serve resources from a consistent URL: 使用一个确定的 URL 来引用资源

这个主要是避免不必要的资源重复下载和 DNS 查询。这种情况主要是当同一个资源（尤其是图片）在几个页面或者几个站点之间重复使用的时候，特别是公司内的各个站点，比如：a.example.com 和 b.example.com 中同时引用了 example.png 这个图片，所以现在就要把这个图片放在 a 或者 b 域名下，而不能既放在 a 又放在 b，这样会造成不必要的重复下载同一个图片，而且也增加了 DNS 的查询时间，造成性能问题。

同时还有一点需要指出的是：相对 URL 和绝对 URL 的问题。如果是在同一个站点内，/images/example.gif 和 www.example.com/images/example.gif 是一样的，站内页面之间可以使用相对地址来引用图片，从而减少页面的大小。但是在站点之间的时候，就会有一个问题：图片所在的站点使用相对 URL/images/example.gif 来引用资源，而另外站点通过 www.example.com/images/example.gif 来引用的时候，IE 和 FF 下都能够很好的处理，无论在哪一个站点访问了这个图片，在另外一个站点就可以在浏览器的缓存里读取图片，可是在 Chrome 下却不会，尽管是一个相同的 URL 页面，不同的 Tab 下浏览都会重新下载资源。测试页面：

<http://www.ilovejs.net/lab/domain.html> <http://www.webairness.com/apps/domain.html>

因此：将想要共享使用的资源（图片、CSS、JS 等等）存放到一个唯一的站点里，其他站点都引用这个唯一的 URL。

2. Serve scaled images: 衡量图片的大小缩放

我们都知道页面中的图片最好能明确的通过 HTML 标签属性或者 CSS 来声明它实际的尺寸，而不要强制声明的尺寸跟图片实际尺寸不符合。但是有种情况是：在一个页面中需要显示一个图片的好几个不同的尺寸，而不想下载几个尺寸不同的图片。比如一个 250x250 的图片，需要在页面中显示一个缩略图是 10x10 尺寸的，这种场景经常在一个 Tab 组件中见到。Google Page Speed 提到，如果图片的实际尺寸跟页面中几个尺寸中至少一个尺寸相符，并且是最大的那个，比如这里的是 250x250，那么这对性能会有所提高。

对于上面提到的情况：到底是使用一个图片并在页面中强制声明缩放图片尺寸，还是加载几个尺寸不同的图片，从而在页面中显示不同尺寸但是内容相同的图片？我写了两个测试页面：

<http://www.ilovejs.net/lab/scale-img/test1.html>

<http://www.ilovejs.net/lab/scale-img/test2.html>

从对上面两个页面测试中发现：当时第一次浏览空缓存的时候，两个页面的渲染速度没什么差别，而总的加载时间是有缩放图片的页面快；当有缓存的时候，这就排除了 HTTP 请求数的影响，此时页面的渲染速度明显是几个尺寸不同的图片快。

因此：这就需要有一个衡量的事情。

附：在 Google Page Speed 中关于这个有一句英文不明白意思 “However, if you serve an image that is larger than the dimensions used in all of the markup instances, you are sending unnecessary bytes over the wire.” 希望懂的人指明一二~

3. Optimize images: 优化图片

优化图片方面主要是利用好三种图片格式：PNG（又分为 PNG8、PNG24）、GIF、JPG/JPEG。这三种格式各有用途，而总得来说：PNG 用在 Sprites 背景图片、修饰页面的 icon 以及渐变图片中；GIF 虽然也可以用在一些背景图片，视情况而定，主要就是因为 GIF 图片格式相对其他两种来说文件大小会小很多，适合用于一些清晰度要求不高的图片中；JPG/JPEG 格式的文件大小一般都是最大的，因为它的色素高，适合展示相片之类对清晰度要求高的图片。

除了选择好了图片格式之外，还得利用工具来压缩图片，对于 JPG/JPEG 格式的图片，可以使用 [jgegran](#) 和 [jpegoptim](#) 这两个工具；对于 PNG，则可以使用 [OptiPNG](#) 和 [PNGOUT](#)。

同时，在 HTML 编写方面，需要明确的指定图片的尺寸（显式声明 width、height 属性），避免浏览器不必要的渲染性能消耗。

4. 拆分初始化负载

随页面一起加载的 Javascript 中，有些函数是用于在加载页面的同时就起作用的，但是还有一部分代码是在页面加载的过程中不会执行或者用不到的，如果让这部分代码连同需要执行的函数代码随页面一起加载的话，将会阻塞后续资源的加载，造成更多的时间延迟。所以我们需要将需要立即执行的代码和在页面 onload 之后才有可能出发的代码拆分开来，而这个拆分的平衡点就是：那些需要在页面的 onload 事件触发前需要执行的以及哪些是在 onload 之后才有可能执行的。将那部分不会立即执行的代码通过 onload 事件来动态的加载，减小页面加载的总时间。

5. Minify HTML

Minify HTML 文档的内容（包括行内脚本和行内样式），这样就可以减小页面的大小，更快的加载 HTML 页面、渲染和 Layout 文档。

Minify HTML 文档跟缩小 JS、CSS 一样会得到非常多的好处：减小网络延迟、增进压缩、更快的加载和渲染。而且 HTML 经常会包含行内脚本和行内样式，压缩它们也都非常有必要。举个简单的例子：行内脚本使用 `<script>` 标签来引入，页面内只需要写 `<script>`，而不需要写 `language` 和 `type` 属性，`<style>` 也一样，不需要写明 `type` 属性，浏览器发现这些没写明的话，会采取默认的执行方式。

同时对于 HTML 标签的写法，也有几点建议：

- 第一、 对于属性用单引号还是双引号的问题。建议统一使用单引号或者双引号，或者在 DOCTYPE 允许的情况下，省略掉。
- 第二、 对于属性的顺序问题。比如 a 标签，应该把 href 属性放在最前面来声明，其他属性根据字符顺序排序来声明：`link`。
- 第三、 在标签于标签之间，去掉不必要的空格和 `\t\r\n` 等等，例如可以看看 <http://www.ilovejs.net> 站点的 HTML 源码。
- 第四、 对于 CSS 的 key-value 的编写方式建议按照字母顺序 a-z 来组织。

6. Minify CSS

Minify CSS 代码（移除不必要的空格、注释、换行符等等）可以节省字节数、加快下载、渲染和执行时间。

有几个工具可以用来实现压缩 CSS：[YUI Compressor](#)、[cssmin.js](#)。

同时，对于 Minify CSS 在编写代码方面尽量保持简写和归类的方式，这样一来容易维护，二来在性能上也会有所提高。比如：

简写：`background:url(example.png) no-repeat 0 0;`

归类：`#test1,#test2,...#testN{background:url(example.png) no-repeat;}`

7. Minify JavaScript

Minify Javascript 代码（移除不必要的空格、注释、换行符等等）可以节省字节数、加快下载、渲染和执行时间。

Minify Javascript 之后有三个好处：

- 第一、 如果有些行内脚本和外部脚本不想要缓存的时候，文件越小网络延迟加载的时间就越短。
- 第二、 Minify Javascript 之后可以更加增进外部脚本代码的压缩以及 HTML 内的行内脚本。
- 第三、 更小的文件可以更快的被浏览器加载和执行。

有几个工具可以用来压缩 Javascript 代码：[JSMIn](#)，[YUI Compressor](#)，[Closure Compiler](#)，[Dojo ShrinkSafe](#)

8. Remove unused CSS: 删除无用的 CSS

删除或者推迟加载一些当前文档用不到的样式规则，可以避免不必要的下载字节数以至于让浏览器更快的

开始渲染页面。

在浏览器开始渲染页面之前，它必须要下载和解析全部的用于布局页面的样式表。即使一个外部的样式表已经存在于缓存中，页面渲染也都会被阻塞直到浏览器从缓存中加载了全部的样式表。另外，一旦样式表被加载下来，浏览器的 CSS 引擎就会执行每一条样式规则，从而确定每一条声明的样式规则对当前文档是否得到应用。但现实中的情况是很多网站从始至终在网站的每一个页面中都使用同一个样式表，而不管样式表中有些规则是否会被当前文档用到。

所以，最好的方式就是最小化由样式表加载和解析所造成的延迟时间，而解决方案就是移除或者延迟加载那些不被当前文档使用到的样式规则。

对此，Google Page Speed 做如下建议：

第一、 在行内样式中删除掉任何不为当前页面使用的样式规则。

第二、 如果你的站点中在几个页面中都引用了同一个样式文件，考虑下将样式文件拆分为更小的文件来为特定的页面特定的样式。

第三、 如果一些样式规则在页面加载的时候不需要使用，最好将他们放置到另外一个文件里，在页面的 onload 事件里加载它。

第四、 如果使用 Javascript 来动态生成样式规则时，确保那些会生成当前页面不需要的的样式规则的 Javascript 函数不会被执行。

上面提到的四点对于清除无用的 CSS 是比较良好的解决方案和建议。口碑网现在加载页面的样式表同样使用了 Minify 的方式，因为头部和底部是全部页面都通用的，所以页面统一都加载了 base.css 和 header.css。但是口碑网站中包括了许多中不同样式的头和底部，base 库里也会有非常多的样式不能在全部的页面中都使用上，所以就造成了在当前页面非常多的无用的 CSS 规则，这一来影响了 CSS 文件的加载，二来影响了 CSS 引擎的解析。

所以，更理想的方式是将 Minify 的方式更加的颗粒化，将 base.css 和 header.css 更加的细分，真正做到按需加载。

9. Enable compression: 允许压缩

用 Gzip 或者 deflate 来压缩资源可以减少大部分的网络传输数据量，这点是毋庸置疑的。可以压缩 HTML 文档内容、CSS 文件、Javascript 文件等等来使得页面加载速度更快。

当前主流浏览器都支持使用 gzip/deflate 来压缩 HTML 文档、CSS 和 Javascript 文件，从而使得从服务端发送回来的数据量比原始大小减少了许多（通常在 50% 上下），这是非常可观的。Gzip 压缩的原理非常简单，就是将指定的文件类型在服务器端打包压缩，在发送回浏览器解压，这个过程是服务器和浏览器会自动处理的，我们只需要正常的使用普通的方式链入 CSS、Javascript 文件即可。

但是我们也知道，这个压缩和解压的过程，是需要消耗 CPU 和内存的，所以 gzip 压缩一般只在压缩大文件的时候才能显示出它的优势。Google Page Speed 经过测试发现：当使用 gzip 来压缩一个大小在 150 到 1000 bytes 的文件时，实际上会使他们更大，体现不出 gzip 的优势了。所以尽量保持在只对 1000 bytes 大小以上的资源进行 gzip 压缩。

同时还有一点需要指出的是：不要对图片或者其他的二进制的文件（比如：PDF, video, ppt 等等）进行 gzip 压缩，因为它们已经进行过压缩了。如果对它们使用了 gzip 压缩之后，不仅不会带来应有的好处，还会实际上使它们更大。如果要压缩图片，可以查看上面的：**Optimize images: 优化图片。**

10. Minimize request size: 最小化 request 请求头信息的大小

尽量保持 request 请求头信息的短小能确保 HTTP 的 request 请求可以通过发送一个数据包就可以完成请求信息的传输。

理想的情况下，一个 HTTP 请求尽量不要超过一个数据包来发送到服务器。当前大部分的在使用的网络对一个数据包大小的限制是在 1500 bytes 左右，所以当你限制每个请求的头信息的大小在 1500 bytes 以内，

你就可以减少发送请求头的时间总开销。而 HTTP 请求头信息一般包括：

- 第一、 Cookies。如果请求一些资源必须要发送 cookie 的时候，尽量保持 cookie 的大小尽量的小，来确保 HTTP 请求头信息的总大小在限制的范围内。因此，发送的每一个 cookie 尽量保持在 1000 bytes 以内。建议是每个域名下发送的全部的 Cookies 的平均值在 400 bytes 以下，将没用的或者是重复的 Cookies 给删除掉。
- 第二、 浏览器自动设置的信息。许多头信息都是由用户代理自动地设置的，这部分是无法控制的。
- 第三、 请求的 URL (GET 或者 Host 头)。如果 URL 设置了很多参数的话，这会无疑的增加了 HTTP 请求头总体的大小。所以尽量保持 URL 的大小越小越好，建议是至少是在几百 bytes 之内。
- 第四、 Referer URL。有时候请求头中会包含 referer 头指定的 URL，这个 URL 跟第三点就有相同的要求了。

因此，尽量保持 HTTP 请求头信息总大小在限制的范围内，对提高页面加载速度，还是有效果的。

11. Serve static content from a cookieless domain: 减小静态资源的 cookie 的大小

通常如果在一个域名下 (比如 www.example.com) 的 HTML 页面在发送请求的时候设置了 cookie，那么在页面中的一些静态资源的请求头中也会发送这些 cookie，而服务器对于静态资源发送的 cookie 是毫无作用的，反而是徒增了 HTTP 请求头的大小。所以，确保静态资源的 HTTP 请求头中不发送这些 cookie 来减小 HTTP 请求头信息的大小。

解决这个问题的方案是：创建一个子域名或者购买一个新域名来映射静态文件，跟当前域名隔离开来。

比如：你的域名是 www.example.com，你就可以创建一个子域名 static.example.com 来映射静态文件。但是，如果你是将 cookie 设置到最高级的 example.com 域名下，那么子域名下的静态资源也都会发送 cookie，这个时候，你就不得不花钱购买一个新的域名来映射这些静态资源。

同时，移除静态资源的 cookie 的另外一个好处是避免一些代理对静态资源拒绝缓存的情况，因为一些代理会拒绝缓存一些 HTTP 请求头中包含 cookie 信息的静态资源。所以从这方面出发，避免静态资源的 HTTP 请求头中包含 cookie 信息，益处是颇大的。

还有一点需要说明的是，对于 Javascript 文件，如果它一定要放置在 head 里并需要在页面中即时加载，最好将这个 Javascript 文件放置到跟 HTML 文档同一个域下，因为反正 Javascript 文件的加载都会阻塞页面其他资源的加载和阻塞页面的渲染，所以就没有必要将这个 Javascript 文件放置到其他域下，增加 DNS 查找的延迟时间。

12. Optimize the order of styles and scripts: 优化 js 和 css 文件的加载顺序

正确对外部的样式表文件和脚本文件或者行内脚本进行排列加载，会得到更好的并行下载数和加快浏览器渲染页面。

这个主要是因为 Javascript 代码可能会改变页面的内容和布局，所以浏览器在遇到 script 标签的时候，会延迟渲染页面任何内容，直到脚本下载并执行完毕。而还有一点就是加载脚本会阻塞后续任何资源的加载，这个是最致命的。但是另一方面，在页面的加载队列里，如果在脚本文件的前面已经存在有另外的文件需要加载，那么脚本文件将会跟这些文件并行加载。

所以这就有几种情况：

- 第一、 当 js 文件穿插在 css 文件中来加载。这样的后果就是每到加载 js 文件的时候，就会阻塞后续的 js 或者 css 文件的加载，这无疑就没有利用好浏览器的并行加载的机制。
- 第二、 当 js 文件都放在 css 文件的最末尾来加载。因为 css 文件的加载不会阻塞后续资源的加载，而且还可以跟其他资源并行加载，所以前面的样式表和样式表后的第一个 js 文件可以并行加载。这比第一点就快速多了，具体图示可以浏览《[Optimize the order of styles and scripts](#)》。

这里还有一种情况需要说明的是：当行内脚本紧跟在样式表后面的时候，将会阻塞后续资源的加载（这里我提一点：是当 script 标签紧跟在样式表 link 标签后面的时候，也将会阻塞后续资源的加载，测试页面：

<http://www.ilovejs.net/lab/script-after-stylesheet.html>)。当行内脚本紧跟在样式表后面的时候，浏览器要在样式表完全下载之后才开始执行行内脚本，如果行内脚本又需要执行比较长的时间，这可能还会更糟。这是因为行内脚本有可能含有依赖于样式表中样式的代码，比如 HTML5 中的 `getElementsByClassName` 方法。浏览器按顺序下载样式表和执行行内脚本是为了保证一致的结果。