HTTP

发展史 http0.9 -->http1.0 -->http1.1 -->http2.0

> http请求和http响应

```
GET /index.html HTTP/1.1 ❶
Host: website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ❷
Server: nginx/1.0.11
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Via: HTTP/1.1 GWA
Date: Wed, 25 Jul 2012 20:23:35 GMT
Expires: Wed, 25 Jul 2012 20:23:35 GMT
Cache-Control: max-age=0, no-cache
Transfer-Encoding: chunked

100 ❸
<!doctype html>
(snip)

100
(snip)

0 ❹

GET /favicon.ico HTTP/1.1 ❺
Host: www.website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: */*
Referer: http://website.org/
Connection: close ❻
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ❼
Server: nginx/1.0.11

 Content-Type: image/x-icon
 Content-Length: 3638
 Connection: close
 Last-Modified: Thu, 19 Jul 2012 17:51:44 GMT
 Cache-Control: max-age=315360000
 Accept-Ranges: bytes
 Via: HTTP/1.1 GWA
 Date: Sat, 21 Jul 2012 21:35:22 GMT
 Expires: Thu, 31 Dec 2037 23:55:55 GMT
 Etag: W/PSA-GAu26oXbDi
```

主要步骤包括：

❶ 请求 HTML 文件，及其编码、字符集和元数据

❷ 对原始 HTML 请求的分块响应

❸ 以 ASCII 十六进制数字表示的分块数据的字节数（ 256 字节）

❹ 分块数据流响应结束

❺ 在同一个 TCP 连接上请求图标文件

❻ 通知服务器不再使用连接了

❼ 图标响应，随后关闭连接

啊，这一次可复杂多了。首先，最明显的差别是这里发送了两次对象请求，一次请求 HTML 页面，一次请求图片，这两次请求都是通过一个连接完成的。这个连接是持久的，因而可以重用 TCP 连接对同一主机发送多次请求，从而实现更快的用户体验。

为终止持久连接，客户端的第二次请求通过 Connection 首部，向服务器明确发送了关闭令牌。类似地，服务器也可以在响应完成后，通知客户端自己想要关闭当前TCP 连接。从技术角度讲，不发送这个令牌，任何一端也可以终止 TCP 连接。但为确保更好地重用连接，客户端和服务器都应该尽可能提供这个信息。

以上就是我们最熟悉的HTTP.1，HTTP 1.1 改变了 HTTP 协议的语义，默认使用持久连接。换句话说，除非明确告知（通过 `Connection: close` 首部），否则服务器默认会保持连接打开。

不过，这个功能也反向移植到了 HTTP 1.0，可以通过`Connection: KeepAlive` 首部来启用。实际上，如果你使用的是 HTTP 1.1，从技术上说不需要 `Connection: Keep-Alive` 首部，但很多客户端还是选择加上它。

优化方向

在了解更宏观的 Web 性能优化之前，先看看下面这些优化方向：

• 浏览器解析和优化方向；

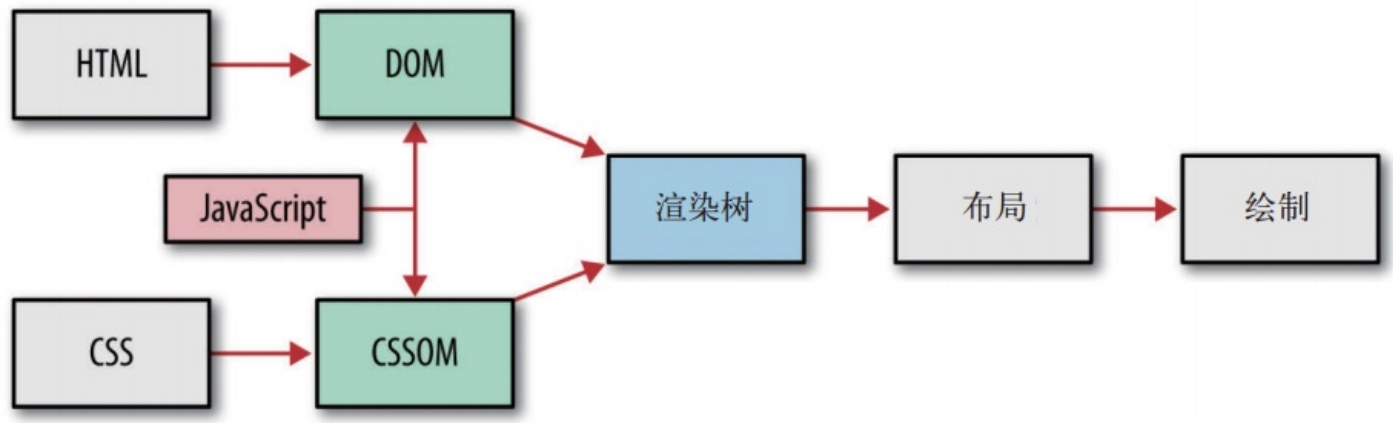
• 延迟和带宽对 Web 性能的影响；

• 传输协议（ TCP）对 HTTP 的限制；

• HTTP 协议自身的功能和缺陷；

浏览器解析过程

我们得先回顾一下浏览器架构，了解一下解析、布局和脚本如何相互配合在屏幕上绘制出像素来。



浏览器在解析 HTML 文档的基础上构建 DOM（ Document Object Model，文档对象模型）。与此同时，还有一个常常被忽略的模型——CSSOM（ CSS Object Model， CSS 对象模型），也会基于特定的样式表规则和资源构建而成。这两个模型共同创建“渲染树”，之后浏览器就有了足够的信息去进行布局，并在屏幕上绘 制图形。到目前为止，一切都很好理解。

然而，此时不得不提到我们最大的朋友和祸害： JavaScript。脚本执行过程中可能遇到一个同步的 document.write，从而阻塞 DOM 的解析和构建。类似地，脚本也可能查询任何对象的计算样式，从而阻塞 CSS 处理。结果， DOM 及 CSSOM的构建频繁地交织在一起： DOM 构建在 JavaScript 执行完毕前无法进行，而JavaScript 在 CSSOM 构建完成前也无法进行。

应用的性能，特别是首次加载时的“渲染前时间”，直接取决于标记、样式表和JavaScript 这三者之间的依赖关系。顺便说一句，还记得流行的“样式在上，脚本在下”的最佳实践吗？现在你该知道为什么了。渲染和脚本执行都会受样式表的阻塞，因此必须让 CSS 以最快的速度下载完。

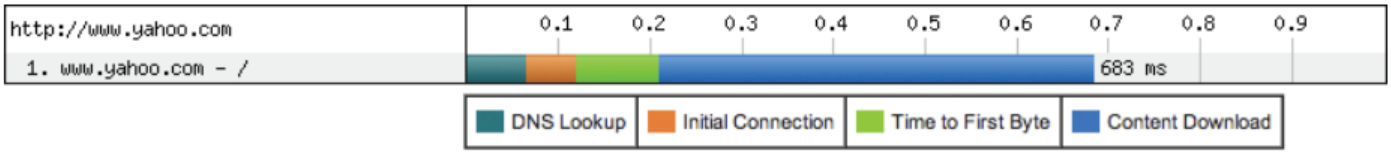
与桌面应用相比， Web 应用不需要单独安装，只要输入 URL，按下回车键，就可以正常运行。可是，桌面应用只需要安装一次，而 Web 应用每次访问都需要走一遍“安装过程”——下载资源、构建 DOM 和 CSSOM、运行 JavaScript。正因为如此，Web 性能研究迅速发展，成为人们热议的话题也就不足为怪了。上百个资源、成兆 字节的数据、数十个不同的主机，所有这些都必须在短短几百 ms 内亲密接触一次，才能带来即刻呈现的 Web 体验。

速度、 性能与用户期望

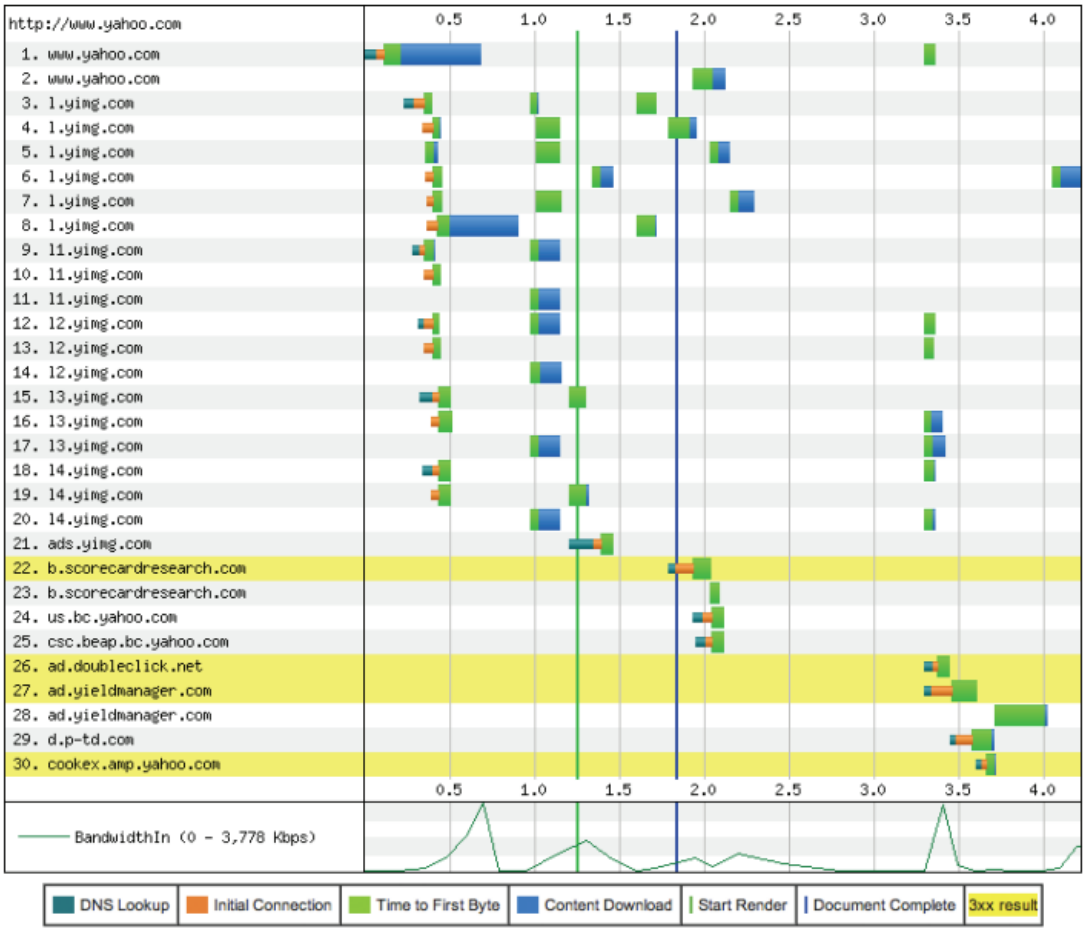
时间	感觉
0 ~100 ms	很快
100~300 ms	有一点点慢
300~1000 ms	机器在工作呢
> 1000 ms	先干点别的吧
> 10000 ms	不能用了

现在，把 DNS 查询，随后的 TCP 握手，以及请求网页所需的几次往返时间都算上，光网络上的延迟就能轻易突破 100~1000 ms 的预算。难怪有那么多用户，特别是那些移动或无线用户，抱怨上网速度慢了！

分析资源瀑布 谈到 Web 性能，必然要谈资源瀑布。WebPageTest HTTP 请求的构成（ WebPageTest）



资源瀑布图记录的是 HTTP 请求，而连接视图展示了每个 TCP 连接（这里共 30个）的生命期，这些连接用于获取 Yahoo! 主页的资源。哪里比较突出呢？注意蓝色的下载时间，很短，在每个连接的总延迟里几乎微不足道。这里总共发生了 15次 DNS 查询， 30 次 TCP 握手，还有很多等待接收每个响应第一个字节的网络延迟（绿色）。



最早渲染时间、文档完成时间和最后资源获取时间，这三个时间说明我们讨论 Web性能时有三个不同测量指标。

Web 应用的执行主要涉及三个任务：**取得资源、页面布局和渲染、 JavaScript 执行**。

其中，渲染和脚本执行在一个线程上交错进行，不可能并发修改生成的 DOM。实际上，优化运行时的渲染和脚本执行是至关重要的，可是，就算优化了 JavaScript 执行和渲染管道，如果浏览器因网络阻塞而等待资源到来，那结果也好不到哪里去。对运行在浏览器中的应用来说，**迅速而有效地获取网络资源是第一要义。**

针对浏览器的优化建议

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。

每个页面的结构和交付：

• CSS 和 JavaScript 等重要资源应该尽早在文档中出现；

• 应该尽早交付 CSS，从而解除渲染阻塞并让 JavaScript 执行；

• 非关键性 JavaScript 应该推迟，以避免阻塞 DOM 和 CSSOM 构建；

• HTML 文档由解析器递增解析，从而保证文档可以间隙性发送，以求得最佳性能。

除了优化页面结构，还可以在文档中嵌入提示，以触发浏览器为我们采用其他优化机制：

```
<link rel="dns-prefetch" href="//hostname_to_resolve.com"> ❶
<link rel="subresource" href="/javascript/myapp.js"> ❷
<link rel="prefetch" href="/images/big.jpeg"> ❸
<link rel="prerender" href="//example.org/next_page.html"> ❹
```

❶ 预解析特定的域名

❷ 预取得页面后面要用到的关键性资源

❸ 预取得将来导航要用的资源

❹ 根据对用户下一个目标的预测，预渲染特定页面

经典的性能优化最佳实践

无论什么网络，也不管所用网络协议是什么版本，所有应用都应该致力于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。这两条标准是经典的性能优化最佳实践，是其他数十条性能准则的出发点。

• 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；参见 2.2.2 节“慢启动”。

• 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。说到这，所有建议都无需解释。延迟是瓶颈，最快的速度莫过于什么也不传输。然 而， HTTP 也提供了很多额外的机制，比如缓存和压缩，还有与其版本对应的一些性能技巧。

• 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。要说最快的网络请求，那就是不用发送请求就能获取资源。要保证首部包含适当的缓存字段：

```
• Cache-Control 首部用于指定缓存时间；
• Last-Modified 和 ETag 首部提供验证机制。
```

• 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。 图片一般会占到一个网页需要传输的总字节数的一半，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%。

连接与拼合

最快的请求是不用请求。不管使用什么协议，也不管是什么类型的应用，减少请求次数总是最好的性能优化手段。可是，如果你无论如何也无法减少请求，那么对HTTP 1.x 而言，可以考虑把多个资源捆绑打包到一块，通过一次网络请求获取：

• 连接

把多个 JavaScript 或 CSS 文件组合为一个文件。

• 拼合

把多张图片组合为一个更大的复合的图片。

对 JavaScript 和 CSS 来说，只要保持一定的顺序，就可以做到把多个文件连接起来而不影响代码的行为和执行。类似地，多张图片可以组合为一个“图片精灵”，然后使用 CSS 选择这张大图中的适当部分，显示在浏览器中。

可是，**牺牲了模块化和缓存粒度**。实现这些技术也要求额外的处理、部署和编码（比如选择图片精灵中子图的 CSS 代码），因而也会给应用带来额外的复杂性。此外，把多个资源打包到一块，也可能给缓存带来负担，影响页面的执行速度。

• 相同类型的资源都位于一个 URL（缓存键）下面。资源包中可能包含当前页面不需要的内容。

• 对资源包中任何文件的更新，都要求重新下载整个资源包，导致较高的字节开销。

• JavaScript 和 CSS 只有在传输完成后才能被解析和执行，因而会拖慢应用的执行速度。

实践中，大多数 Web 应用都不是只有一个页面，而是由多个视图构成。每个视图都有自己的资源，同时资源之间还有部分重叠：公用的 CSS、JavaScript 和图片。实际上，把所有资源都组合到一个文件经常会导致处理和加载不必要的字节。虽然可以把它看成一种预获取，但代价则是降低了初始启动的速度。

内存占用也会成为问题。对图片精灵来说，浏览器必须分析整个图片，即便实际上只显示了其中的一小块，也要始终把整个图片都保存在内存中。

在资源受限的设备，比如手机上，内存占用很快就会成为瓶颈。对于游戏等严重依赖图片的应用来说，这个问题就会更明显。

> 平衡的艺术

• 有选择地组合一些请求对你的应用有没有好处？

• 放弃缓存粒度对用户有没有负面影响？

• 组合图片是否会占用过多内存？

• 首次渲染时是否会遭遇延迟执行？

把首次绘制所需的 CSS 单独拿出来，优先于其他 CSS 文件发送；

嵌入资源

嵌入资源是另一种非常流行的优化方法，把资源嵌入文档可以减少请求的次数。比如， JavaScript 和 CSS 代码，通过适当的 script 和 style 块可以直接放在页面中，而图片甚至音频或 PDF 文件，都可以通过数据 `URI（ data:[mediatype][;base64],data）` 的方式嵌入到页面中：

```
<img src="
AAAAACH5BAAAAAAALAAAAAABAAEAAAICTAEAOw=="
alt="1x1 transparent (GIF) pixel" />
```

数据 URI 适合特别小的，理想情况下，最好是只用一次的资源。嵌入资源也不是完美的方法。如果你的应用 要使用很小的、个别的文件，在考虑是否嵌入时，可以参照如下建议：

• 如果文件很小，而且只有个别页面使用，可以考虑嵌入；

• 如果文件很小，但需要在多个页面中重用，应该考虑集中打包；

• 如果小文件经常需要更新，就不要嵌入了；