Machine Translated by Google

6 针对 HTTP/2 进行优化

本章涵盖 HTTP/2 对 Web 开发人员的改变 HTTP/1.1 Web 性能技术是否是 HTTP/2 下的反模式

其他性能技术以及它们在 HTTP/2 下是否仍然相关如何针对 HTTP/1 和 HTTP/2 进行优化连接合并

您已经对 HTTP/2 有了很好的理解:它旨在解决什么问题、它是如何工作的,以及它带来的一些新特性和机会。在本书的第三部分中,我还有一些更高级的主题要涵盖,但是您已经有了足够的信息来了解 HTTP/2 对您的网站意味着什么,以及如何针对它进行优化。你应该如何改变你的开发实践?你能放弃一些表演技巧吗?您可以使用哪些新技术?对于那些不能使用 HTTP/2 的用户,你会怎么做?本章旨在回答这些问题。

Machine Translated by Google

6.1 HTTP/2 对 Web 开发人员意味着什么

您已经看到 HTTP/2 从根本上改变了 HTTP 消息发送到服务器的方式,并因此带来性能优势。但是开发人员是否需要改变他们的开发语言和实践?您应该使用特定的 JavaScript 框架来利用 HTTP/2 的优势吗?总的来说,这些问题的答案是不需要改变,尽管有些改变可能是有益的。

HTTP/2 旨在向后兼容。如果您的服务器支持 HTTP/2,您应该能够打开它,并且在大多数情况下,无需更改一行代码即可立即看到性能优势。一些新功能（例如HTTP/2 推送)需要更改才能使用它们,并且通过本章旨在提供的对 HTTP/2 的更深入理解,可以实现性能增强改进,但 HTTP/2 不应该要求您进行切换后这些更改。

在大多数情况下,这些更改是可进一步提高性能的可选附加功能。

然而,启用 HTTP/2 可能并不像您希望的那么简单（在第 3 章和附录中介绍）。您可能需要升级您的基础架构,甚至考虑新的基础架构,例如在您的 Web 服务器前面安装反向代理或内容分发网络。此决定可能会导致您在升级的同时或在未来可以利用的其他决定和机会。不过,这些主题有些不同。在本章中,我旨在解释 HTTP/2 对 Web 开发人员意味着什么,假设他们可以访问它。

如何从浏览器进行 HTTP/2 调用 HTTP/2

最好的事情之一是在服务器端支持它之后,您不需要在客户端进行任何更改;浏览器会为您完成这项任务。您无需升级 JQuery 版本、使用不同的 AJAX 语法或从 Angular 切换到 React（反之亦然）。从 Web 开发人员的角度来看,每个前端 HTTP 请求和响应都像以前一样工作 除了（理想情况下)更快,因为没有排队发生。这些库和工具让浏览器处理进行网络调用的低级细节,因此只有浏览器需要了解 HTTP/2。

目前,前端开发人员无法指定应该使用 HTTP/1.1 还是 HTTP/2,就像他们无法指定是使用 HTTP/1.1 还是 HTTP/1.0（甚至 HTTP/0.9)一样。这种情况将来可能会改变,因为能够在 HTML 或 AJAX 中提供调用的优先级,例如 a、 b、 c ,或者为 HTTP/2 推送注册回调,d将非常有用。

---

[a] https://bugzilla.mozilla.org/show_bug.cgi?id=559092 https://

[b] bugs.chromium.org/p/chromium/issues/detail?id=41501 https://github.com/

[c] WICG/priority-提示 https://github.com/whatwg/fetch/issues/65

[d]

Machine Translated by Google

6.2 一些 HTTP/1.1 优化现在是反模式吗?

HTTP/2 旨在解决 HTTP/1.1 中的一些基本性能问题。这些性能问题使得在 HTTP/1.1 下请求单独的资源变得非常昂贵,这导致出现了各种技术来增加 HTTP 连接数或最小化请求的资源数。增加 HTTP 连接的数量需要浏览器打开多个连接,甚至在多个域上托管资源(分片)。减少请求数量意味着使用连接将多个 CSS 和 JavaScript 文件合并为一个大文件,或者为网站使用的所有小图像(例如社交媒体图标或其他小图标)创建图像精灵,这些图像可以再次提取巧妙地使用CSS。两种类型的优化都涉及使用更少的 HTTP 请求传输相同的数据(或至少相似的数据)。

这些解决方法解决了 HTTP/1.1 的一些低效问题,但也引入了新问题,如第 2 章所述。HTTP/2 试图在协议级别修复其中的许多问题。由于二进制框架层,请求现在在协议层几乎是免费的;因此,是否不再需要解决方法?事实上,关于这些 HTTP/1.1 性能技术在 HTTP/2 下成为反模式的讨论很多。好吧,没那么快;我说几乎是免费的,而且只是在协议层。

6.2.1 HTTP/2 请求仍然有成本

当网页引用资源时,会发生很多事情,其中一些通过 HTTP/2 进行了改进,而另一些则没有。图 6.1 详细说明了当网页要求包含资源时浏览器需要做出的许多决定和过程中的一些。

在较高层次上,浏览器需要检查各种缓存中是否存在有效副本(在第 5 章中讨论),如果不为它发出 HTTP 请求。发出 HTTP 请求可能涉及使用现有连接或启动新连接,具体取决于使用的域和请求类型。下载资源后,客户端需要查看缓存标头以决定是否将资源保存到缓存中以供潜在重用。即使完成所有这些步骤,浏览器仍必须处理资源(解析 CSS 或 JavaScript、处理 JPEG 图像等)。

在您了解 HTTP 发送和接收请求的实际细节之前,已经发生了很多事情,不要介意在拥有资源时如何处理它。所有这些过程都需要时间 通常是很短的时间。但是,如果您去除 HTTP/1.1 典型的六个连接限制并拥有数百个资源,您就会开始看到一些新的、有趣的问题。

2016 年,随着 HTTP/2 使用的增加,Chrome 开发人员注意到在对大量资源使用 HTTP/2 时出现明显的延迟。[1]一些网站冻结了

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=655585

Machine Translated by Google

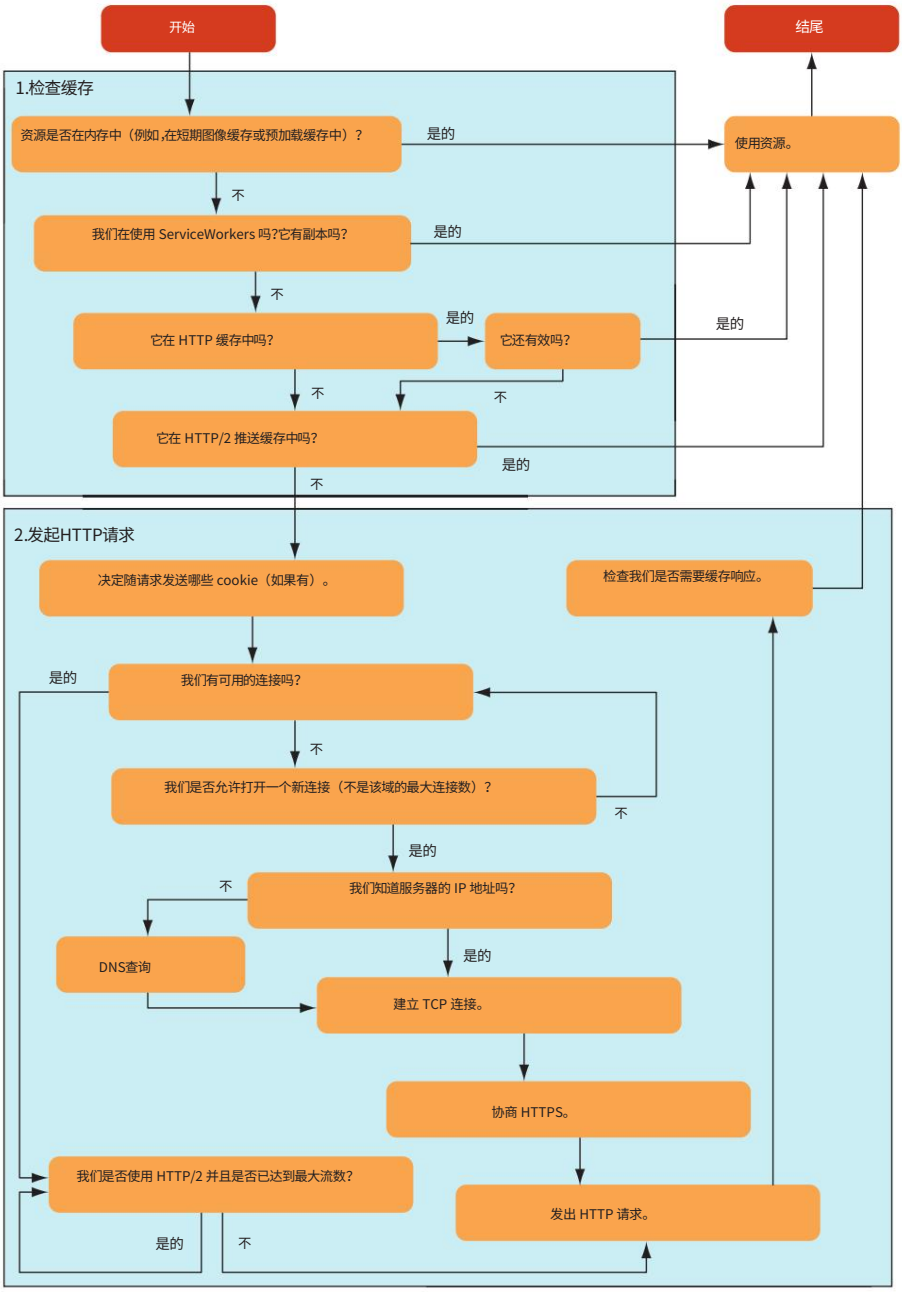


图 6.1 需要 HTTP 资源时的决策和流程

Machine Translated by Google

例如,当没有发送单个请求时为 400 到 500 毫秒。这个问题与 HTTP/2 协议本身无关。您将在第 7 章中看到流优先级应该允许这些并发请求在协议级别适当地确定优先级。问题纯粹是发送资源的瓶颈。

在此期间,浏览器忙于完成上一节中提到的所有任务。 HTTP/2 可能允许许多资源一起运行,而不会因等待空闲连接而导致性能下降,但这种能力将瓶颈转移到其他地方。

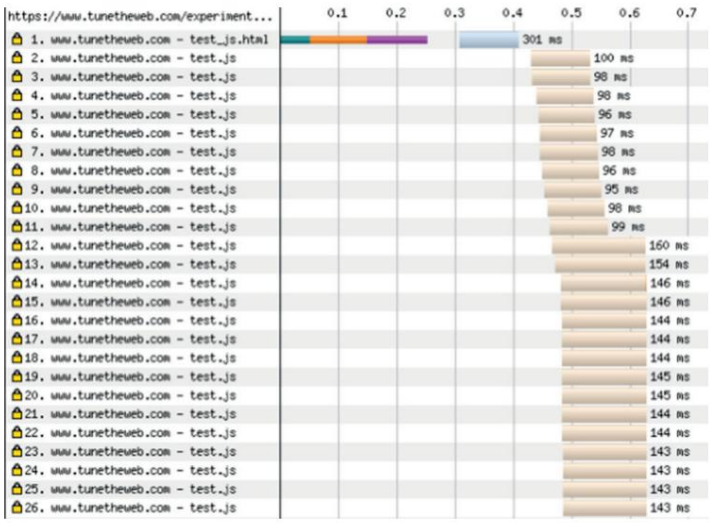
为了防止这种延迟,Chrome 团队不得不限制即使在 HTTP/2 下也可以排队的资源数量,回到 HTTP/1 的六个连接限制,直到团队可以优化代码。此限制仅针对非必要资源实施,因此从理论上讲,HTTP/2 并不比 HTTP/1.1 差,对网站来说应该无关紧要。但是出现了一个有趣的场景:具有async或defer属性（通常出于性能原因添加以防止阻塞这些资源)的 Java Script 资源受到限制,而没有这些属性的 JavaScript 资源则没有。使用性能最佳实践（例如使 JavaScript 非阻塞)的网站受到人为限制,加载速度比不遵循这些最佳实践的网站慢。使用async或defer的网站最终加载与 HTTP/1.1 请求完全一样,如图 6.2 所示。

这种情况是否重要还有待商榷； JavaScript 被标记为async或defer ,因此对于加载来说并不重要。但这让一些网站所有者感到惊讶,他们在实施 HTTP/2.2 时看到了这种类似 HTTP/1 的行为。此限制已从更高版本的 Chrome 中删除,但确实表明在删除瓶颈时需要小心,例如中的六个连接限制浏览器,因为瓶颈可能掩盖了其他性能问题。

[20] https://stackoverflow.com/questions/45384243/google-chrome-does-not-do-multiplexing-with-http2/45775288#45775288

Machine Translated by Google

HTTP/2 - 不使用异步/延迟



HTTP/2 - 使用异步/延迟

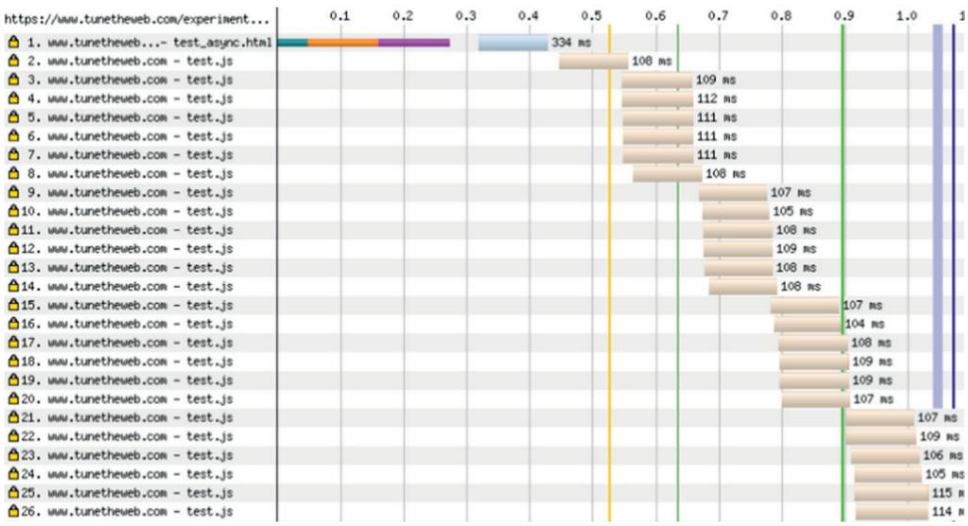


图 6.2 Nonrender blocking JavaScript 在 Chrome 中被人为减慢

6.2.2 HTTP/2 不是无限的

另一点需要注意的是,HTTP/2 并未完全取消限制。您在第 5 章中看到,尽管 SETTINGS_MAX_CONCURRENT_STREAMS默认为无限制,但许多实现添加了限制,如表 6.1 和 6.2 所示。

Machine Translated by Google

表 6.1 流行的 HTTP/2 服务器端实现的并发流限制

软件	类型	默认并发流
Apache HTTPD (v2.4.35)	网络服务器	100a
nginx (v1.14.0)	网络服务器	128b
水 (2.3.0)	网络服务器	100摄氏度
信息系统 (v10)	网络服务器	100
码头 (9.4.12)	Web 和 Java servlet 容器	128d
阿帕奇汤姆猫 (9.0)	Web 和 Java servlet 容器	200e
节点 (10.11.0)	JavaScript 运行时环境	100f
阿卡迈	内容分发网络	100
亚马逊 CloudFront 和 S3	内容分发网络	128
云焰	内容分发网络	128
最大CDN	内容分发网络	128

a https://httpd.apache.org/docs/2.4/mod/mod_http2.html#h2maxsessionstreams http://
b nginx.org/en/docs/http/ngx_http_v2_module.html#http2_max_concurrent_streams https://
c h2o.examp1e.net/配置/http2_directives.html#http2-max-concurrent-requests-per-connection https://www.eclipse.org/
d jetty/documentation/9.4.x/http2-configuring.html https://tomcat.apache.org/ tomcat-9.0-doc/config/http2.html
e https://nodejs.org/api/http2.html
f

表 6.2 流行的 HTTP/2 网络浏览器的并发流限制

软件	默认并发流
铬 (v69)	1000
火狐 (v62)	未设置（使用 HTTP/2 默认值无限制）
野生动物园 (v12)	1000
歌剧 (v56)	1000
边缘 (v17)	1024
浏览器 11	1024

其中一些设置取自文档,其他设置取自实验。对于 CDN,我使用 Wireshark 拦截对主页或该 CDN 上托管的其他页面的请求,并查看第 4 章（第 4.3.3 节)中所述的SETTINGS框架。对于浏览器,我设置了一个带有详细日志记录的 nghttp 服务器,并检查了各种浏览器在建立 HTTP/2 连接时发送的设置帧。

Machine Translated by Google

显而易见的是,服务器端设置远低于 Web 浏览器的限制。事实上,Firefox 没有设置限制并使用默认的 unlimited 值,这是有道理的,因为最终浏览器可以控制发送的请求,因此它可以在协议之外添加限制（就像 Chrome 最初在前一节）。

此外,浏览器服务的请求数量可能比服务器要少得多,服务器可能同时响应多个用户。然而,考虑到 Chrome 在请求大量资源时遇到的性能问题,如果浏览器制造商进一步限制默认连接数,直到 HTTP/2 完全适应,可能会更好。拥有超过 200 个资源的站点很少见,但可能当 HTTP/2 完全嵌入并且网站停止连接和拼接资源时,这种情况就会变得普遍。如果你有一个需要 500 个资源的网站,并且你认为在 HTTP/2 下你不需要连接这些资源,那么很可能最初只会获取 100 到 128 个资源,其余的将像在 HTTP/2 下一样排队1.1.当达到 100 个流服务器限制时,此行为与第 2 章（第 2.6.1 节)中描述的完全相同。在一些示例中,网站试图删除连接并遇到这些新限制。 3

6.2.3 资源越大压缩越高效

所有网络资源在通过网络发送之前都应该被压缩。对于某些格式（例如用于图像的 JPEG 和 PNG 以及用于字体的 WOFF 和 WOFF2）,压缩是内置在格式中的,Web 服务器不应在此压缩之上尝试它。对于主要基于文本的格式,如 HTML、CSS 和 Java Script,使用 gzip（或较新的 brotli 格式4)等压缩,通常由 Web 服务器即时进行。

几乎所有这些压缩格式的一个共同点是,它们可以比小文件更有效地压缩大文件。第 7.3.4 节介绍了每种压缩算法的工作原理,但只需说明这些算法中的大多数都是通过查找重复的数据系列并将它们替换为对该数据的单个版本的引用来工作的。对于更大的文件,可以找到更多的重复项,并且压缩比可以更大。最终结果是,压缩一个 100 KB 的大文件总是比分别压缩 10 个 10 KB 的文件更好,即使未压缩的数据总量完全相同。通过不再捆绑资产（就像您在 HTTP/1.1 下所做的那样）,您可能会失去一些压缩优势并最终在 HTTP/2 下通过网络发送更多数据,即使未压缩的数量相同。

这种情况究竟有多大影响取决于您的文件。至
检查一些真实世界的例子,下载一些流行的 jQuery 文件：

```
卷曲-OL https://code.jquery.com/jquery-3.3.1.min.js 卷曲-OL https://
code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min .js curl -OL https://code.jquery.com/ui/1.12.1/jquery-
ui.min.js
```

---

[3] http://engineering.khanacademy.org/posts/js-packaging-http2.htm#http-2-0-has-service-issues
[4] https://tools.ietf.org/html/rfc7932

Machine Translated by Google

然后使用 Linux cat命令合并这些文件：

```
cat jquery-3.3.1.min.js jquery.mobile-1.4.5.min.js jquery-ui.min.js > jquery_combined.js
```

如果列出文件,您应该会看到文件大小总和：

```
-rw-r--r-- 1 barry p 86927 5 月 19 日 19:31 jquery-3.3.1.min.js -rw-r--r-- 1 barry p 253668 5 月 19 日 19:31
jquery-ui.min .js -rw-r--r-- 1 barry p 200143 5 月 19 日 19:31 jquery.mobile-1.4.5.min.js -rw-r--r-- 1 barry
p 540738 5 月 19 日 19:31 jquery_combined .js
```

86,927 + 200,143 + 253,668 = 540,738 KB 合并文件的大小,符合预期。

然后使用标准设置的 gzip 压缩所有文件：

```
gzip jquery*
```

现在看看新的文件大小：

```
-rw-r--r-- 1 barry p 30371 5 月 19 日 19:31 jquery-3.3.1.min.js.gz -rw-r--r-- 1 barry p 68058 5 月 19 日 19:31
jquery-ui .min.js.gz -rw-r--r-- 1 barry p 55649 5 月 19 日 19:31 jquery.mobile-1.4.5.min.js.gz -rw-r--r-- 1 barry
p 152652 5 月 19 日 19:31 jquery_combined.js.gz
```

执行相同的加法 (30,371 KB + 68,058 KB + 55,649 KB) 产生 154,078 KB,但组合文件压缩为 152,652 KB。它只小了 1%,但还是有区别的,对于非缩小文件（我在 6.3.1 节中讨论）,差异可能更大。

另一方面,不再捆绑文件可能会让开发人员更加关注他们在页面上包含的内容。例如,某些开发人员可能已将一个大型串联 JavaScript 文件包含在网站的每个页面中。有些页面需要此 JavaScript 的某些部分,而其他页面需要其他部分,但由于 HTTP/1.1 处理多个文件时效率低下,因此通常为每个页面发送一个大文件比为每个页面创建自定义包更有效。

使用 HTTP/2,多次下载不再是问题,因此每个页面都可以更改为只包含它需要的 JavaScript,从而减少每个页面要传输的数据量。

传输数据的减少是否抵消了压缩率的损失?答案取决于您的站点、您进行拆分的程度,以及您之前发送了多少现在不需要发送的不必要数据。一些示例案例研究 5 表明,即使删除不必要的代码,在线传输的数据也会略有增加,尽管在这些案例中交付的 JavaScript 数量可能是一个更大的问题。即使资源稍大一些,由于在某些页面上处理较少的数据而带来的性能改进也可能超过

[97] http://engineering.khanacademy.org/posts/js-packaging-http2.htm#bundling-improves-compression

Machine Translated by Google

抵消使用分拆时发送更多原始字节带来的性能损失。总而言之,HTTP/2 可能不需要像 HTTP/1.1 那样多的捆绑,但在解除捆绑之前要考虑压缩率的任何损失。

6.2.4 带宽限制和资源争用

HTTP/1.1 对随时可能发生的 HTTP/1.1 请求创建了自然节流。因此,Web 浏览器努力将最重要的资源优先发送。是的,Web 开发人员试图通过连接和分片来解决这种情况,但这些解决方案是有限的。现在限制已经显着增加（如果没有删除）,您可以更容易地陷入某些请求使其他请求带宽不足的情况。

在第 2 章的示例站点中,通过 HTTP/2 下载 360 度图像导致并行下载 100 个图像,而不是单独的 HTTP/1.1 连接下的六个资源限制。因为100次下载是同时发生的,在HTTP/2下每个资源的下载时间都比较长,如图6.3所示。

总体而言,通过 HTTP/2 下载图像的速度要快得多,但最好先完全下载一些资源（在 HTTP/1.1 下自然会发生）,而不是每个图像都下载一点（在 HTTP/1.1 下自然会发生） 2,除非使用明确的优先级来防止这种行为）。

网页设计网站 99design 在第一次切换到 HTTP/2.6 时遇到了类似的问题网页所需的所有资源现在都可以并行下载,包括一些大的、高质量的图像（正如你对设计的期望网站）。但是页面顶部的一些图像在 HTTP/2 下加载速度较慢,因为屏幕外图像同时下载并占用带宽。在 HTTP/1.1 下,浏览器必须优先下载屏幕上的图像,六个连接的限制确保这些图像不会被许多低优先级的屏幕外图像延迟。

这两种情况都可以通过适当地对请求进行优先级排序来改善。到目前为止,我已经简要地谈到了 HTTP/2 下的优先级,但在第 7 章中深入介绍了它。优先级在很大程度上不在网站所有者的手中（这就是为什么我将它留到后面的章节）,但它可能证明成为浏览器和服务器性能的关键区别。如果浏览器可以建议适当的优先级（例如以比页面下方图像更高的优先级下载视口中的图像）,并且如果服务器可以响应这些建议,则优先级可以带来更高性能的页面。或者,图像可以完全按照请求的顺序下载,这就是 Chrome 目前所做的（见第 7 章）。

因此,这个问题与其说是 HTTP/2 协议本身的问题,不如说是一个早期实施问题。然而,鉴于开发人员在 HTTP/1 方面拥有 20 年的经验,这不太可能成为向 HTTP/2 世界迁移过程中的最后一个此类问题。

---

https://99designs.com/tech-blog/blog/2016/07/14/real-world-http-2-400gb-of-images-per-day/

Machine Translated by Google

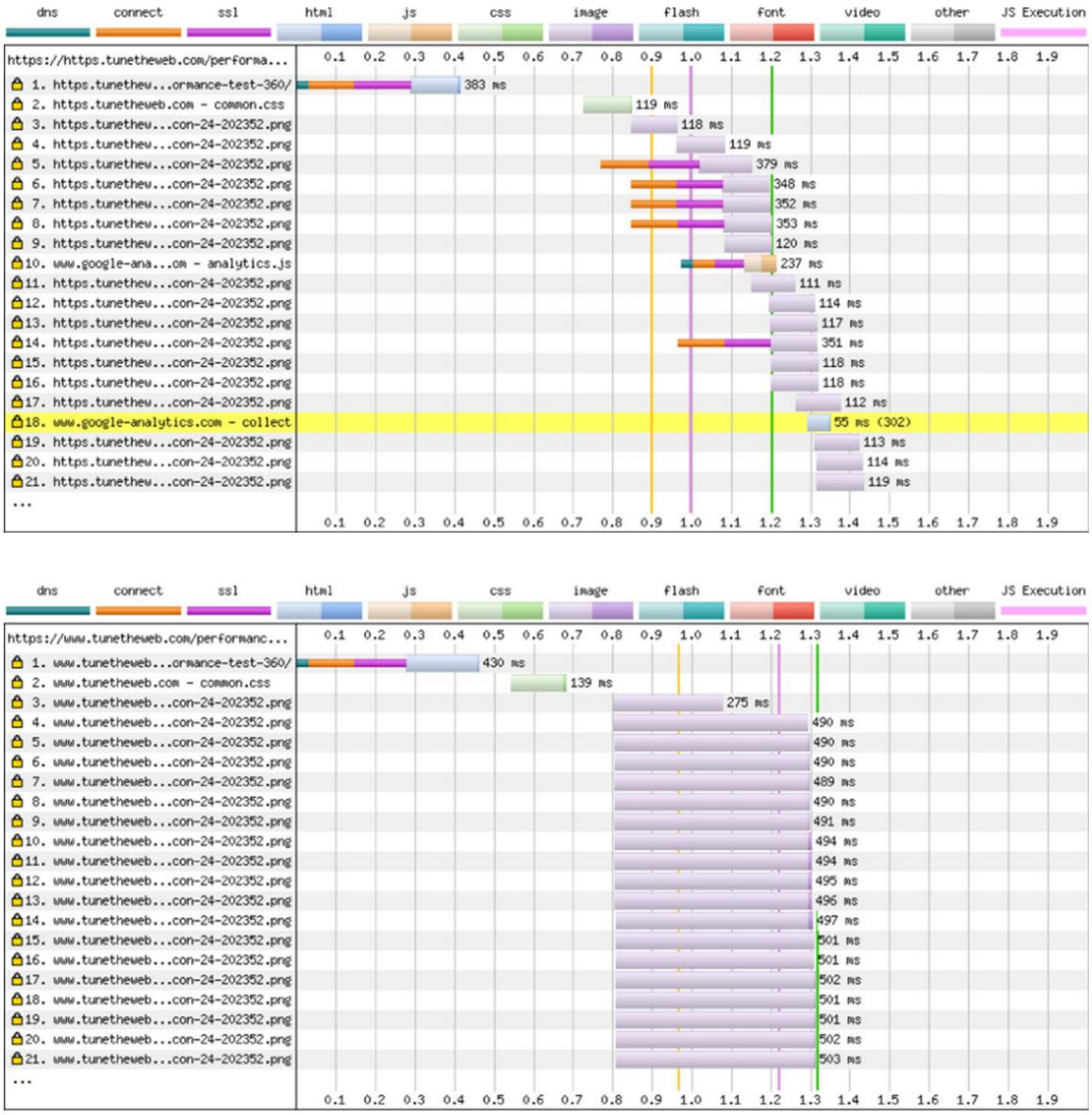


图 6.3 在 HTTP/1.1（顶部)下下载单个资源比在 HTTP/2（底部)下花费更少的时间

6.2.5 分片

分片用于打破浏览器通常强加给域的六个连接限制。通过在子域或单独的域上托管媒体,网站可以并行下载更多内容。在我看来,除了大量资源外,分片的影响可能被夸大了,在这种情况下,连接或拼接可能是更好的解决方案。研究表明,许多额外的连接通常只用于一两个资源,因此设置这些连接所需的时间

Machine Translated by Google

可能会超过有效使用它们的任何性能提升。一如既往,网站应该衡量分片等技术的影响,而不是盲目地将它们放在适当的位置,假设它们总是会提高性能。

在 HTTP/2 世界中,分片没有那么大的意义,建立和管理独立基础设施的努力产生的收益有限。此外,HTTP/2 的某些部分（例如HTTP/2 推送和 HPACK 标头压缩)在单个连接上工作得更好,因此分片会导致性能下降。因此,我希望随着 HTTP/2 变得越来越普遍,分片的使用会越来越少。正如您将在第 6.4.4 节中看到的那样,HTTP/2 有一种方法来反向分片,允许两全其美。在某些特定情况下,分片可能仍然比有损连接有价值（请参阅第 9 章）,但一般来说,除非在那些特定情况下,否则不再需要它。即使在这些情况下,浏览器最好决定在需要时打开多个连接,而不是让站点为所有连接设置分片,无论它们是否有损。

6.2.6 内联 关键

CSS 或脚本的内联,对我来说一直是一个 hack[7] 一个强大的 hack,但仍然是一个 hack。将 CSS 代码放在页面的头部可以改善第一个页面的加载,但随后该代码要么在加载真正的 CSS 时复制内容,要么不允许在同一站点上为后续页面加载进行缓存。此外,仅内联关键 CSS 可能会很复杂,因为可以覆盖 CSS 样式表的默认加载方式以防止它们呈现阻塞。

HTTP/2 推送本应消除对内联的需要,但正如第 5 章所展示的那样,事实证明这项技术很难有效地使用,因此它并没有像预期的那样流行起来。我希望内联在一段时间内仍然是一种常见的性能优势,对于那些愿意使用它并希望从第一页加载中挤出最后一盎司的网站来说。

6.2.7 结论

HTTP/2 的主要目标之一是解决 HTTP/1.1 中请求成本高昂的问题。 HTTP/2 有了相当大的改进,但 HTTP 请求仍然有成本。通常,这个问题是由于协议之外的问题（例如向浏览器发出多个请求的成本)造成的,但其他问题至少部分是由于 HTTP/2 的工作方式造成的。例如,如果关键请求资源匮乏,太多并发请求可能会导致某些指标（例如首次绘制)变慢。

您还必须考虑 HTTP/2 实现尚未成熟的事实;一些次优的实施无疑会随着时间的推移而改善。可能会发现错误,或者新协议的使用效率可能很低（例如没有适当地确定请求的优先级）。我强烈建议你在这个早期阶段让你的 HTTP/2 软件（服务器和浏览器)保持最新。

---

[7] https://www.tunetheweb.com/blog/inlining-css-is-not-for-me/

Machine Translated by Google

重复前面的一点,经过 20 年的发展,开发人员对 HTTP/1 和成熟的技术栈有了深刻的理解,但 HTTP/2 仍处于起步阶段。该技术似乎并没有太多令人震惊的问题,但存在许多意想不到的瓶颈和低效实例,其中 HTTP/2 可能没有达到应有的效率。

HTTP/2 还显着增加了并行下载限制（从之前的六个连接）,但并没有删除它们。我的建议是您暂时将每个域的请求保持在 100 个以下资源。如果这样做会导致数百个文件,请不要删除连接,而是使用增加的限制来更适当地捆绑。

在 HTTP/1.1 下流行的性能优化要求较少,但在开发人员习惯这个新协议及其含义时删除它们还为时过早。相反,将它们缩小到适当的水平（例如,使用较少的连接而不是没有连接）。开发人员或许应该将他们的代码分组到可能一起使用的资产组中,而不是像过去那样将文件串联成一个大文件。减少 HTTP/1.1 技术的使用而不是根除它们似乎是研究人员和网站所有者达成的普遍共识。 8 当 HTTP/2 首次出现时,有很多人谈论许多 HTTP/1.1 性能技术是现在是反模式,但这并不是严格意义上的。

关键要点是不要假设 HTTP/2 是灵丹妙药。大多数网站都会看到改进,但少数网站可能会受到影响。解释任何预期的行为都需要仔细衡量和理解协议。

6.3 Web 性能技术在 HTTP/2 下仍然相关

HTTP/2 改进了 HTTP 协议早期版本的一些低效,如本章所讨论的,这使得一些性能技术在 HTTP/2 下的相关性有所降低。 HTTP 协议优化并不是您可以用来提高网站性能的唯一技术,但根据 Web 的性质（通常涉及客户端和服务器在一定距离内进行交互）,Web 性能的很大一部分是优化网络级别的使用。值得回顾影响数据传输的其他相关性能最佳实践,并解释为什么它们在 HTTP/2 下仍然相关,并指出 HTTP/2 下的任何新机会。

6.3.1 最小化传输的数据量

无论 HTTP/2 为处理请求和响应提供了哪些改进,发送较少的数据总是更好。 HTTP/2 并没有神奇地使互联网连接变大;它使他们更有效率。网站加载速度似乎更快,但

[8] https://uhdspace.uhasselt.be/dspace/bitstream/1942/23909/1/h2bestpractices_RobinMarx_WEBIST2017.pdf

Machine Translated by Google

它们通常加载相同数量的数据（如果压缩效率不高,可能会稍微多一些,如第 6.2.3 节所述）。因此，尽可能减少您发送的数据量仍然是值得的。您用于实现此减少的所有技术在 HTTP/2 下同样相关。

使用适当的文件格式和大小

富媒体网页可能很有趣,但下载和显示媒体需要时间。根据 HTTP Archive,将近 80% 的网页大小来自图像和视频9 （见图 6.4）,因此发送适当的媒体格式和大小仍然很重要。

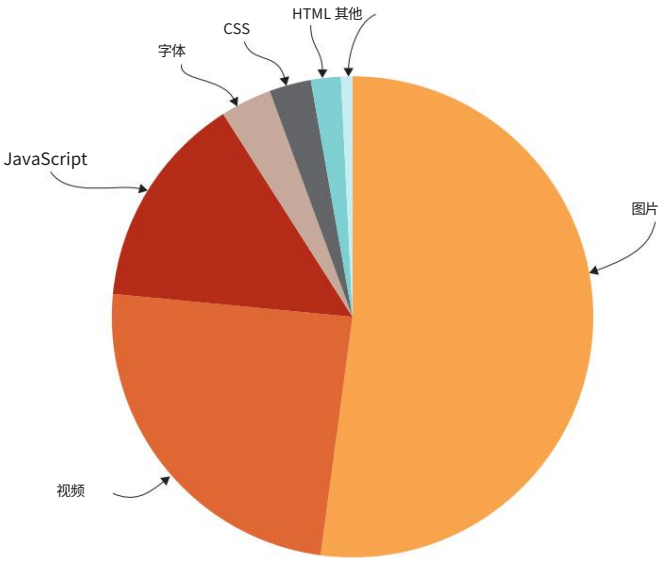


图 6.4 按内容类型分类的每页平均字节数

视频和音频有点专业,所以我们不会在这里介绍它们,但我们会稍微讨论一下图像。10通常,您对照片使用 JPEG（也称为 JPG)格式,对其他图形使用 PNG。 WebP 由 Google 推动,但似乎并没有流行起来,尽管它已经推出好几年了。11 SVG 的使用正在增长,但仍有一段路要走。几乎所有网站都使用 JPEG 和 PNG 格式,12这是有充分理由的;这些垫子得到普遍支持,并在尺寸和质量之间取得了良好的平衡。

JPEG 是一种有损格式;您会损失一些图像质量,但可以设置压缩级别与质量。降低图像质量并生成更小的图像很容易,而肉眼可察觉的质量不会有任何明显下降。图片可以

---

9 https://httparchive.org/reports/page-weight
10 https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats https://
11 caniuse.com/#feat=webp https:// w3techs.com/technologies/overview/image_format/all
12

Machine Translated by Google

还包含大量元数据（拍摄照片的时间、拍摄的相机、使用的 ISO 设置等）。这些元数据中的大部分与浏览您网页的人无关，应该被删除，但请注意您不拥有的图像，以防许可条款限制更改。

各种工具和图像编辑软件可以更改质量和元数据以减小文件大小，但为了易于使用，我推荐 tinypng.com，[13]它可以快速且毫不费力地压缩 JPEG 和 PNG。它也可以在 tinyjpg.com 上获得；两个站点都使用相同的工具，可以处理两种格式。图 6.5 显示了如何显着减小大多数图像的大小。

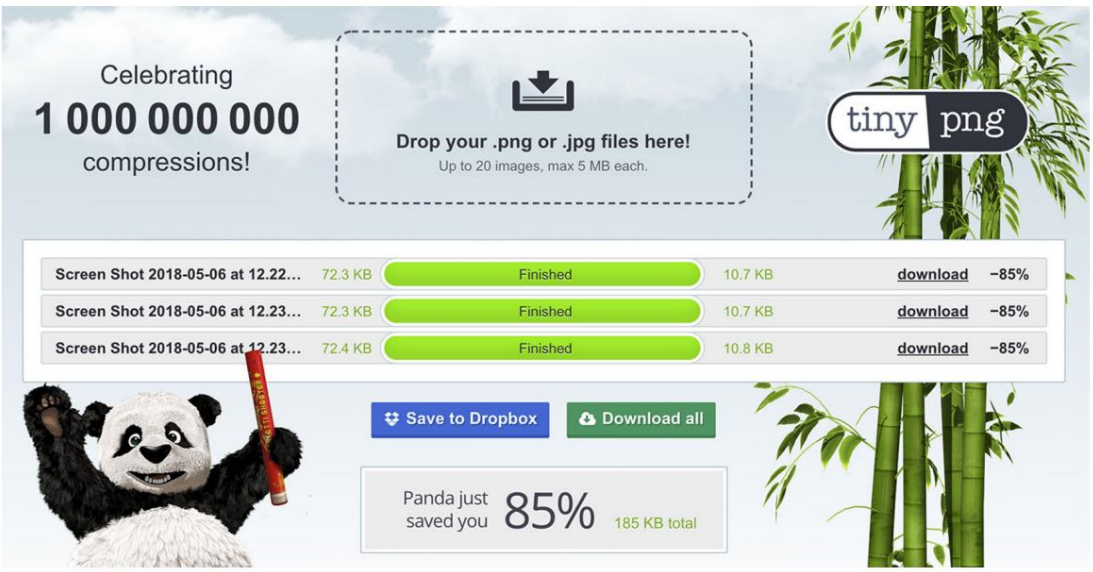


图 6.5 在 tinypng 中拖放文件可以显着减小文件大小。

除了看质量水平，你还应该看尺寸。发送 5120 × 2880 像素的图像以 100 像素宽度显示它会浪费下载时间和浏览器的处理时间。绝不能将大打印质量、打印尺寸的图像放在网页上。如果您需要这些类型的图像可用，请将它们添加为单独的下载链接。

发送适当大小的图像通常也意味着为移动和桌面网站使用不同的图像，并且可能还为介于两者之间的各种屏幕尺寸使用不同的图像。
虽然幸运地在大屏幕上访问您的网站的访问者可能会欣赏高质量的图像，但如果向他们发送相同的图像，使用移动设备的访问者可能会诅咒您的缓慢网站。图 6.6 显示桌面和移动站点的大小正在趋同，这可能是因为更多的站点正在迁移到

[13] https://tinypng.com/

Machine Translated by Google

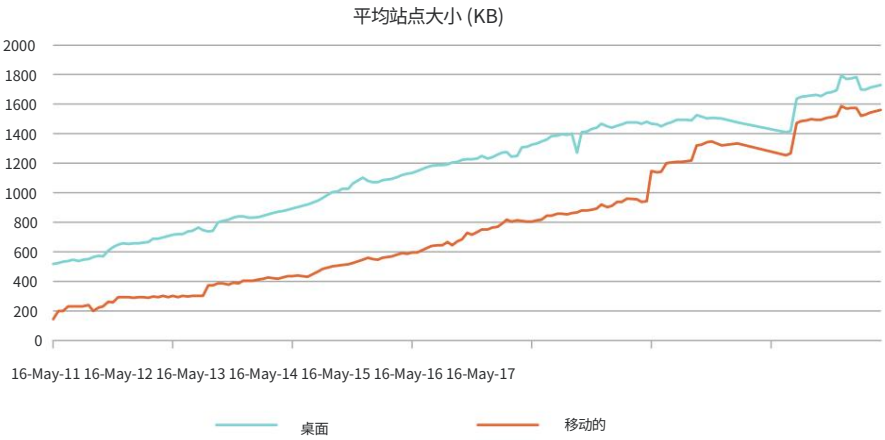


图 6.6 移动端和桌面端站点规模正在趋同。

响应式设计,两者都使用一个网站,但未能在该网站上使用技术将不同的图像大小传递到两个不同的平台。

总之,HTTP/2 不会改变可以发送的文件格式,也不会改变传输到客户端或在客户端使用的数据。继续使用最合适的文件格式并优化您的媒体,然后再将其放到您的网站上。

压缩文本数据

尽管媒体压缩通常由图像格式处理,但 Web 的基本技术（HTML、CSS 和 JavaScript)是基于文本的,您也应该尽可能减少这些资源的大小。在 HTTP/1.1 下,您使用 gzip 或类似工具压缩 HTTP 主体以减少发送的数据量,在 HTTP/2 下您应该继续这样做。 HTTP/2 总体上不会改变发送什么数据,只会改变发送方式。因此,压缩文本响应主体在 HTTP/2 下仍然与在 HTTP/1.1 下一样重要。文本压缩效果非常好,您可以轻松实现高达 90% 的压缩率。表 6.3 显示了常见 JavaScript 和 CSS 库的压缩率。 14

表 6.3 使用 gzip 的常用库的压缩率

图书馆	尺寸	压缩尺寸	压缩率
jquery-1.11.0.js	276 KB	82 KB	70%
angular-1.2.15.js	729 KB	182 KB	75%
bootstrap-3.1.1.css	118 KB	18 KB	85%
foundation-5.css	186 KB	22 KB	88%

[14] https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize encoding-and-transfer?hl=en#text-compression-with-gzip

Machine Translated by Google

在发送数据之前压缩数据的唯一缺点是在服务器端压缩和在浏览器端解压缩所需的时间和计算能力,但这种能力在现代硬件上几乎可以忽略不计。发送更少字节的网络优势几乎总是远高于压缩的时间和计算成本。

Gzip 仍然是最流行的压缩技术,[15]尽管 Brotli[16] 等压缩算法正变得越来越普遍。 Brotli 提供更好的压缩（取决于设置[17]） ,因此节省更多。对于尚不支持 brotli 的浏览器,除了 gzip 之外还应该使用这些工具。[18] HTTP/2 以与 HTTP/1.1 相同的方式处理不同的内容编码,因此更改为 HTTP/2 对是否压缩没有区别您的响应主体（请这样做！)或使用什么格式。唯一轻微的相关性是一些较新的格式（如 brotli)需要 HTTPS,并且可能需要更高版本的 Web 服务器。如果您同时迁移到 HTTPS 和 HTTP/2 或者必须为 HTTP/2 升级您的网络服务器软件,您现在可以选择使用这些新格式,尽管也许最好通过您的 HTTP/2在将这些格式添加到组合之前进行迁移。

压缩的最好之处在于,除了在服务器上进行一些设置外,它是无缝的。设置好后,Web 服务器会即时压缩资源,Web 浏览器会自动解压缩资源,大多数网站所有者在启用压缩后就可以忘记它了。某些 Web 服务器允许您提供预压缩的内容以减少 Web 服务器上的负载。每次您想要添加新内容（在上传之前对其进行预压缩)时,此技术都需要付出一些额外的努力,但这种努力在 HTTP/2 下不会改变。

无论您使用什么版本的 HTTP,您都应该继续压缩您的 HTTP 主体。内容编码将在 HTTP/1.1 下的内容编码HTTP 标头中传达给浏览器。

压缩 HTTP 标头当我谈到压缩主题
时,在 HTTP/2 下确实发生了变化并且我还没有详细讨论的一件事是标头压缩。 HTTP/1 只允许压缩请求和响应主体,而 HTTP/2 还使用称为 HPACK 的格式压缩 HTTP 标头。这种格式是 HTTP/2 减少多个请求的性能开销的另一种方式。如果没有 HPACK,通过两个请求发送相同的内容将导致发送两倍的需要发送的标头数据。随着 HTTP 标头的使用不断增长,这种改进尤为重要,因为对于许多小请求,标头可以构成请求和响应的很大一部分。

[15] https://w3techs.com/technologies/details/ce-compression/all/all
[16] https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html
[17] https://blogs.akamai。 com/2016/02/understanding-brotlis-potential.html https://
[18] caniuse.com/#feat=brotli

Machine Translated by Google

因为标头压缩由底层浏览器处理,网站所有者和开发人员几乎无事可做,所以我现在不进一步讨论这个话题,但我会在第 8 章中解释这个过程是如何工作的。

精简代码

另一种数据缩减方法是缩小代码,无论该代码是 HTML、CSS 还是 JavaScript。此方法涉及去除空格和注释,并经常重写代码以减小局部变量名称的大小并删除不必要的分隔符。与压缩一样,此方法在 HTTP/2 下不会改变,因此如果您之前最小化,您可能应该在迁移到 HTTP/2 后继续这样做。

但是 HTTP/2 可能允许在没有太多（或至少没有那么多)串联的情况下部署代码,因此可能不再需要构建步骤。您可能希望利用这个机会使您部署的代码与您的源代码更加一致,因为不再需要这样的差异。在这种情况下,您可能不想再缩小。不缩小可能会增加轻微的性能成本,但在我看来,在压缩之上进行缩小只会增加一点额外的成本,因为压缩已经相当好地去除了重复的字符串（例如空格）。不利的一面是,如果您尝试调试生产中的问题,则阅读最小化代码会更加困难。不幸的是,并非所有错误都可以在开发环境中轻松重现!您可以添加源映射,允许您稍微“取消最小化”您的生产代码,但该过程可能很复杂。19考虑一个真实世界的代码示例以了解缩小的空间优势:流行的 Bootstrap v4.0.0 库。这个库提供了它的 CSS 和 JavaScript 代码的缩小版本和原始版本,所以比较这些版本,从 Bootstrap CSS 开始。在Chrome中,打开开发者工具,确保出现Size栏,从https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css加载Bootstrap CSS文件,如图6.7.

如果“大小”列未显示这两个数字,请单击“使用大请求行”按钮。在此示例中,大小列同时显示压缩大小 (21.0 KB) 和未压缩值 (138 KB)。然而,从技术上讲,这些数字是不可比较的:最高值是传输的大小,因此它包括 HTTP 标头（请求和响应）,而最低值只是返回的原始正文大小。我分别测量了标头,它们在未压缩时大约为 0.5 KB。所以忽略这些差异,因为它们的影响可以忽略不计。也可以单独下载和 gzip 文件（如第 6.2.3 节中所述)以获得更准确的值,但该技术引入了有关命令行和 Web 服务器使用的 gzip 设置的问题。

[19] https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/

Machine Translated by Google

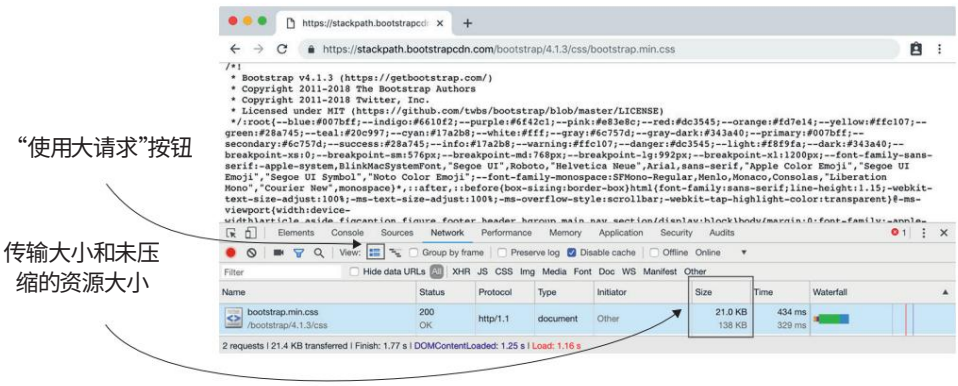


图 6.7 查看请求的压缩和未压缩大小

对 unminified 版本重复此过程,然后将它们放在一起产生表 6.4 中的值。

表 6.4 gzip 和压缩对 Bootstrap CSS (v4.1.3) 的影响

压缩型	Bootstrap.css	Bootstrap.min.css
原始文件	170 KB	138 KB（原始大小的 81%）
压缩包	22.8 KB（原始大小的 13%）	21.0 KB（原始大小的 12%）

如您所见,未缩小版本缩小到大小的 13%,而缩小版本缩小到大小的 12%。缩小和 gzip 压缩更好,但你说的只是 1% 或 1.8 KB 的差异。是的,每个字节都很重要,对于像 Bootstrap 这样的提供程序,提供库的缩小版本是有意义的。但是对于您自己的代码,节省的空间通常并不大。

保存取决于个人代码。尽管 Bootstrap CSS 文件似乎没有太大的好处,但是当您查看 Bootstrap JavaScript 文件时,您会发现更大的好处（表 6.5）。

表 6.5 gzip 和压缩对 Bootstrap JavaScript (v4.1.3) 的影响

压缩型	Bootstrap.js	Bootstrap.min.js
原始文件	121 KB	49.8 KB（原始大小的 41%）
压缩包	20.9 KB（原始大小的 17%）	14.2 KB（原始大小的 12%）

在这里,您会看到更大的压缩增益（仅 gzipped 17%,gzipped 和 mini fied 12%）,从而节省了 5%,即 6.7 KB。大多数人对 JavaScript 代码的注释多于对 CSS 代码的注释,而 JavaScript 通常使用更多

Machine Translated by Google

空白比 CSS 多,并且包含更多的变量名,所以这个结果并不出人意料。事实上,您可以看到单独缩小,没有 gzip 压缩,将 JavaScript 文件减小到 41% 的大小,而 CSS 仍然是原始大小的 81%。

顺便说一下,两个仅缩小后的文件大小都比仅压缩后的版本大得多,这支持了我的观点,即 gzip（或类似的压缩)是使用的主要优化;如果你想使用,缩小会给你一些额外的改进

它也。

然而,对于拥有必要专业知识的大型网站和网站所有者来说,缩小可能是值得的。表 6.6 对表 6.5 进行了扩展,显示了一些常见 Web 开发库的压缩大小。您会在其中一些库的缩小版本中看到明显的差异,尤其是 jQuery 和 Angular。

表 6.6 常用库的压缩率

图书馆	尺寸	压缩尺寸	压缩率
jquery-3.3.1.js	265 KB	78.9 KB	30%
jquery-3.3.1.min.js	84.9 KB	30.0 KB	35%
角度-1.7.2.js	960 KB	297 KB	31%
angular-1.7.2.min.js	168 KB	56.6 KB	34%
bootstrap-4.1.3.css	121 KB	20.9 KB	17%
bootstrap-4.1.3.min.css	49.8 KB	14.2 KB	29%
foundation-6.4.3.css	158 KB	18.8 KB	12%
foundation-6.4.3.min.css	118 KB	14.7 KB	12%

最小化代码的另一个原因是混淆。然而,尝试使用混淆来隐藏任何逻辑的用途有限,因为取消代码是微不足道的。尽管如此,去除评论可能会阻止内部开发人员的任何尴尬想法泄露到现实世界中（例如“必须在某个时候解决这个糟糕的代码”）。最后,最小化的代码在理论上可以稍微好一点,因为浏览器要解析的代码更少,尽管任何解析的第一步都会有效地最小化代码,所以可能不会有太大的好处。

总而言之,HTTP/2 应该不会直接影响您缩小代码的决定。但是,如果转向 HTTP/2 改变了您的开发实践（例如减少连接）,您可能需要重新审视缩小的好处,看看它们是否仍然适用。缩小比让 Web 服务器压缩文件更复杂,并且缩小在压缩之上的改进要少得多。

Machine Translated by Google

6.3.2 使用缓存防止重发数据

关于 Web 性能的一个经常重复的引用（尽管似乎没有人记得是谁第一个说的20)是：“最快的 HTTP 请求是未发出的请求。” HTTP/2 试图提高 HTTP 请求的性能,但它永远不会比通过尽可能缓存根本不发出请求更快。如果访问者再次需要该资源,他们可以从 HTTP 缓存中获取它,这总是比发出完整的网络请求更快,无论协议是 HTTP/1、HTTP/2 还是

一些未来的版本。

尽可能多地缓存是 HTTP/1 的一个极好的性能提示,它仍然是 HTTP/2 中的一个。相同的cache-control和expires HTTP 标头存在于 HTTP/2 中并且仍然应该被使用 尽管有些人认为不再需要expires标头并不是没有道理的,因为所有常见的客户端都理解 HTTP/1.1 和 HTTP/1.0在现实生活中很少使用,尤其是需要注意缓存的应用程序。21缓存可能很复杂。如何最大化缓存而不导致在更新后显示陈旧内容是一个棘手的话题,需要大量思考和缓存破坏技术。22 缓存常见资产（样式表、Java 脚本、徽标等)可以是快速响应的网站与使用起来缓慢且痛苦的网站之间的最大区别之一。因为缓存是一个如此重要的主题,所以有必要了解 HTTP 缓存和304 （未修改)响应代码,我在第 1 章中简要提到过。

当收到 HTTP 响应时,它可以包含缓存控制HTTP 标头（或较旧的expires HTTP 标头）,指示资源应被视为有效的时间长度。考虑一个真实世界的例子。如果您从新浏览器加载维基百科,您应该会看到图 6.8 中的标题。

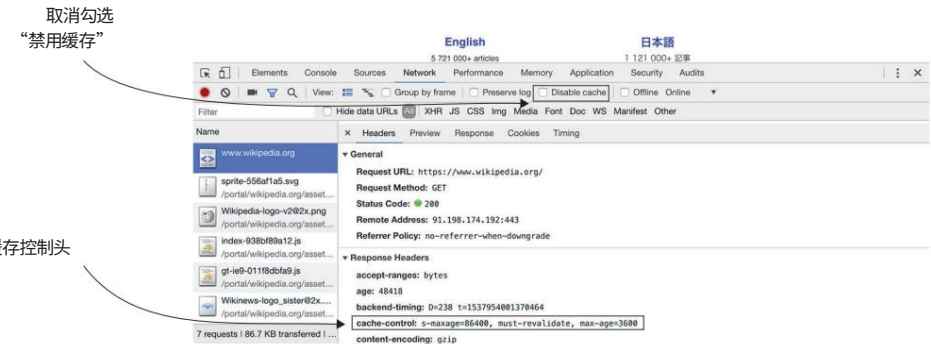


图 6.8 维基百科缓存控制标头

---

20 最有可能的候选人 Steve Souders 否认是他本人,尽管这句话经常被归因于他： https ://www.stevesouders.com/blog/2012/03/22/cache-them-if-you-can /. https://www.fastly.com/blog/headers-we-dont-want https://css-tricks.com/

21 strategies-for-cache-busting-css/

22

Machine Translated by Google

此图显示维基百科主页可以缓存 3600 秒（max-age=3600），即 1 小时，之后必须重新验证才能使用（必须重新验证）。该图还指定了其他中间缓存，例如代理，可以将页面缓存 86,400 秒，或 1 天（s-maxage=86400）。但是这些中介通常使用其他技术来保持最新（一个超出本书范围的主题），因此您可以忽略该设置。

确保未选中图中显示的禁用缓存复选框。
然后向后浏览，然后向前浏览一个页面，您应该会看到类似于图 6.9 的内容。

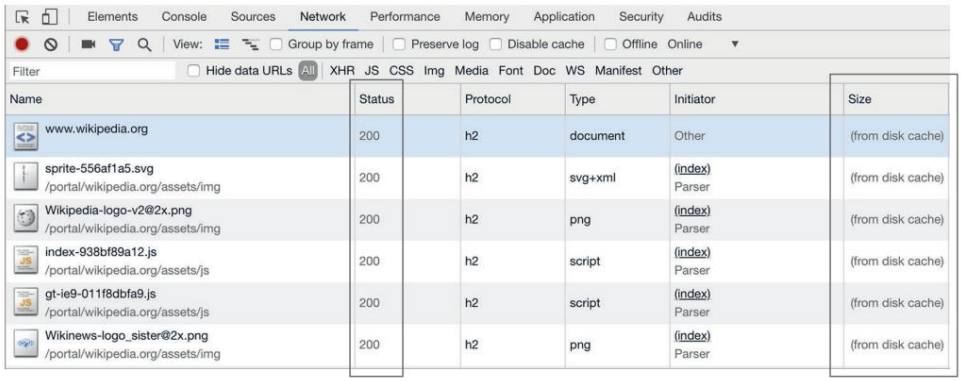


图 6.9 从磁盘缓存中加载维基百科

正如预期的那样，该网站已从磁盘缓存中加载，如“大小”列中所示。如果你看到这个（来自内存缓存）而不是（来自磁盘缓存），很可能你来自另一个维基百科页面而不是来自不同的站点，所以这些资源在更近的内存缓存中，但原理是一样的.旧的缓存响应如图 6.9 所示，包括200状态代码（稍微变暗以表明它是缓存响应，在图中可能很难看到）。

为了让事情变得更有趣，等待一个小时让缓存过期，然后再次尝试实验。为了节省时间，重新加载页面（效果相同）。
您应该会看到类似于图 6.10 的内容。
在这里，您会看到304响应代码而不是通常的200。顺便说一下，如果您重新加载同一个窗口，您还会看到 Chrome 使用内存中的图像缓存来存储图像，而不是通常的基于磁盘的 HTTP 缓存，根据图 6.9 的讨论。出现304响应是因为浏览器发出了条件GET请求，如图 6.11 所示。

浏览器在缓存中找到了主页，发现它已过时，并发送了对该页面的请求 但它说“如果它比我根据上次修改得到的页面更新，请将页面发送给我上次随页面发送的日期(if-modified-since)或eTag值(if-none-match)。” eTag值允许更多

Machine Translated by Google

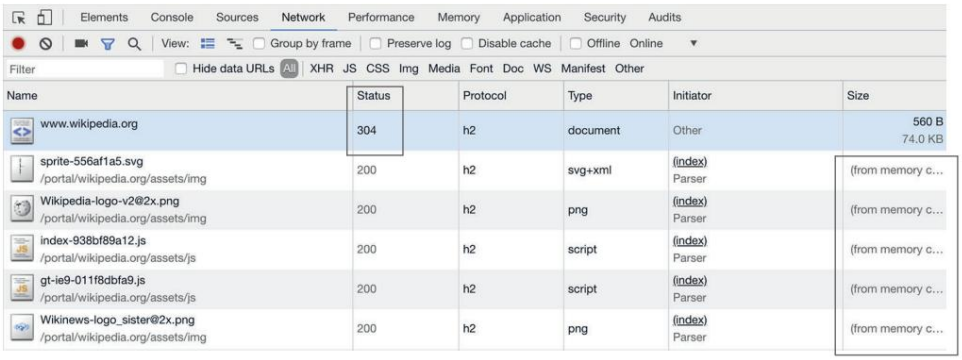


图 6.10 从过时的缓存中重新加载维基百科

比用来表示新鲜度的日期。该值是特定于实现的,但可以是内容的散列,例如。如果给出了两个值（如图 6.11 所示）,则if-none-match标头中给出的eTag值优先。服务器检查,发现页面没有变化,并发送一个304响应,表示 Web 浏览器的副本仍然完好。 304响应没有HTTP 正文,因此下载速度比完整资源快。

304响应还是慢;他们需要一直到服务器并返回的网络调用。在 HTTP/2 下网络旅行更便宜（但不是免费的！）。然而,在 HTTP/1 下,由于 HTTP 请求相对昂贵,发送304响应的成本几乎与发送完整的200响应一样高,因此304响应可能没有得到应有的使用。例如,许多网站根本不缓存它们的 HTML 页面。每次您转到主页时,浏览器都会再次下载该页面,然后仅将缓存用于它所访问的资源

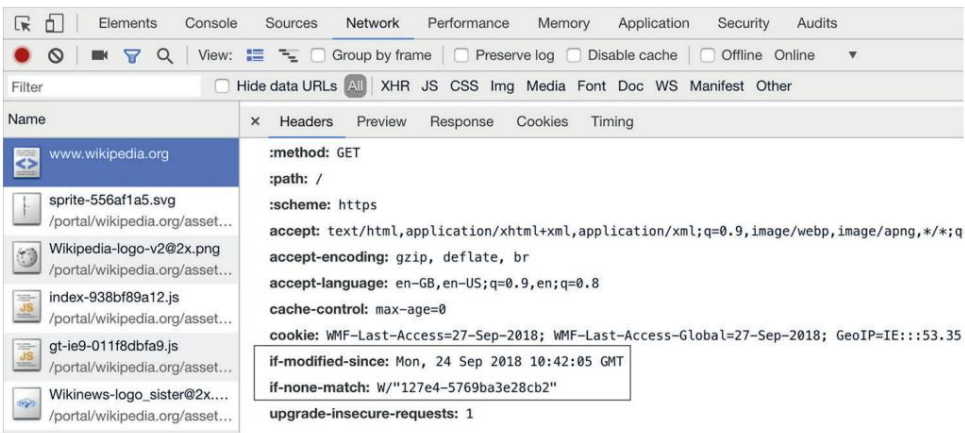


图 6.11 条件GET请求

Machine Translated by Google

需要。不缓存网页会使浏览网站的速度变慢。如果您在主页上,单击另一个页面然后想返回主页,重新加载应该是即时的,但浏览器重新加载主页时通常会有一小段延迟。向网页添加缓存控制指令,即使是短暂的指令,也可以使站点感觉响应更快（它在缓存时间内立即从缓存加载)并且还可以节省带宽（即使在缓存过期后也使用304响应）.

然而,重要的是要认识到304响应仍然会产生成本,因此我并不是建议您将它们用作缓存的替代品,而是可能用作您迄今为止尚未缓存的某些资源的替代品。当我为前面的图拍摄屏幕截图时,我努力寻找一个像维基百科那样缓存网页本身的网站。一些新闻和社交媒体网站依赖于页面是最新的,但还有其他更好的方法可以将内容加载到页面中,而不是提供不应缓存的预生成 HTML,例如使用 JavaScript AJAX 请求。我认为网站应该在短时间内缓存页面,至少在使用 HTTP/2 时,因为304响应的成本远低于200响应。

许多关于 HTTP 缓存的建议建议使用长期缓存时间,以便将来的访问从缓存中受益,但这种技术可能不如使用缓存在该会话中浏览网站重要,这样网站在用户导航时感觉响应迅速。将缓存用于会话中浏览改进同样适用于短期缓存时序,而短期缓存可能会降低复杂的缓存清除技术的必要性。同样,在服务器端使用缓存来防止边缘服务器查询后端服务器可以带来显着的改进,即使缓存时间很短。 23 缓存时间较短的另一个缺点是资源可能会被清除缓存,因为浏览器认为它们是无效的。最好的选择是“长时间缓存但在较短时间后重新验证”选项,但这样的选项不存在。 HTTP 工作组联合主席 Mark Nottingham 24建议使用Stale-While- Revalidate 选项 25允许这种情况,但还没有浏览器采纳该建议。

总而言之,HTTP/2 并没有直接改变缓存选项,但网络请求成本的降低可能会导致对缓存策略的重新评估。此外,有人建议可以更改 HTTP/2 推送以允许更新缓存（请参阅第 5 章）,但目前无法进行此更改。

---

[23] https://www.nginx.com/blog/benefits-of-microcaching-nginx/ https://
[24] httpwg.org/ https://www.mnot.net/blog/2014/06/01/
[25] chrome_and_stale-while-重新验证

Machine Translated by Google

6.3.3 Service workers 可以进一步降低网络负载

Service workers26是除 Internet Explorer 11.27 之外的所有现代浏览器都提供的相对较新的功能,它们提供了一种启动位于网页和网络之间的 JavaScript 代理的方法,如图 6.12 所示。

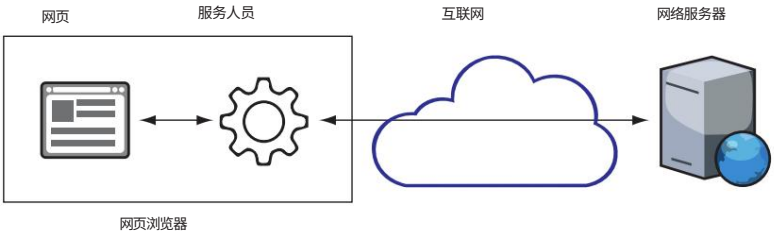


图 6.12 服务工作者

Service Worker 可以查看、回答或更改 HTTP 请求。它们可用于提供与本机移动应用程序类似的体验,尤其是在离线时。即使网页本身被缓存,页面重新加载也会尝试连接到网站以检查缓存版本是否仍然有效,如果没有网络连接,重新加载将失败。同样的体验通常不会发生在移动应用程序中,它不允许您离线刷新。当在网站上使用 service worker 时,service worker 可以中断请求,并在离线时返回先前缓存的资源版本。此方法允许缓存站点加载甚至

离线,就像移动应用程序一样。

Service Workers 为优化 Web 开发的 HTTP 端创造了各种有趣的机会。您可以使用较短的缓存期,但即使在过期后也不会从 Service Worker 缓存中清除项目。这种技术可以让您更频繁地使用304响应,而不会出现长期缓存的危险。由于 Service Worker 的使用在 HTTP/2 下没有改变,并且 Service Worker 本身就是一整本书的主题,因此我不再进一步介绍 Service Worker。但我预计 service workers 的使用在未来几年内会大幅增加,因为 service workers 提供了强大的选项来处理 HTTP 请求。

6.3.4 不要发送你不需要的东西

继续不发出不必要的 HTTP 请求的主题,另一个性能改进是确保您只发送实际需要的数据。

尽管这一点似乎很明显,但有很多原因可能导致您发送未实际使用的数据。

6.2 节中提到的技术 连接文件和使用图像精灵 通常会导致发送比减少

---

[26] https://developers.google.com/web/fundamentals/primers/service-workers/
[27] https://caniuse.com/#feat=serviceworkers

Machine Translated by Google

HTTP 请求数。由于这种请求减少在 HTTP/2 下不太必要,您可能希望从数据角度重新审视这些技术。如果您已将它们集成到网站发布的构建过程中,那么您可能会很乐意继续使用它们,因此没有迫切需要删除它们,但您还应该考虑是否由于这些原因发送了不必要的数据技巧。

还有其他方法可以加载不需要的资源。例如,您可能包括隐藏在移动视图中但仍被下载的图像。

HTTP/2 不做任何更改来阻止您发送不会被使用的资源。
事实上,它添加了一种需要格外小心的新方法（HTTP/2 推送）,如第 5 章所述。

HTTP/2 可能使下载资源看起来更快,但它对提高下载的数据量没有任何作用（撇开 HTTP 标头压缩不谈）。网站应继续确保只下载必要的数据。

6.3.5 HTTP 资源提示

我在第 5 章介绍了预加载资源提示。这个提示是一套资源提示28 的一部分,可用于进一步优化 HTTP 使用,这些选项在 HTTP/2 下与在 HTTP/1 下一样相关。每个提示都作为链接HTTP 标头或 HTML 中的<link>元素实现。 HTTP 资源提示有

已经存在了一段时间,但最近获得了牵引力和支持。它们提供了补充 HTTP/2 的其他方法。

DNS 预取

您在第 2 章中看到亚马逊使用了这个提示。这段代码包含在主页的HEAD部分:

```
<link rel= dns-prefetch href= //m.media-amazon.com >
```

顾名思义,这个提示使 DNS 查找在需要连接之前就发生了,从而节省了时间（参见图 6.13 的第 17 行）。

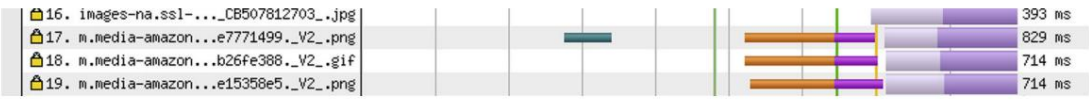


图 6.13 使用中的 DNS 预取

这个提示可能只节省一点点时间,但只需要一点点代码就可以实现!支持很强,所有主要浏览器都支持它一段时间。29 DNS 查找确实有生存时间 (TTL),因此网站不应过早查找域（例如仅用于下一次导航的域）, 因为查找可能

[28] https://w3c.github.io/resource-hints/
[29] https://caniuse.com/#feat=link-rel-dns-prefetch

Machine Translated by Google

如果 TTL 到期,则需要重复。但 300 秒或更长的 TTL 是常见的,理想情况下,您的页面加载时间不会超过 5 分钟,因此使用当前页面上的资源应该是安全的。但是,此提示仅对较晚发现的资源有用。在引用资源之前进行dns-prefetch是没有意义的,因为该引用无论如何都会进行 DNS 查找。此技术对于依赖资源所需的连接最有用,该连接从单独解析 HTML 时看不出来。大多数网站从任何其他域加载内容,因此使用此标头可以获得很好的收益。

预连接

Preconnect 将概念更进一步。它不是只进行 DNS 查找,而是继续建立连接,这可以节省与建立新连接相关的昂贵的 TCP 和 HTTPS 设置成本。浏览器支持很强大,30大多数现代浏览器都支持它。不要过早预连接； TCP 慢启动算法将在一段时间不使用后启动,或者更糟的是,连接将被断开（参见第 9 章）。

与 DNS 预取一样,当需要来自其他域的关键资源时,预连接很有用。

预取

预取获取具有低优先级的资源。与试图使当前页面加载更快的预加载（接下来讨论)不同,预取通常用于将来的导航;因为它是以如此低的优先级获取的,所以在当前页面完成加载之前不会使用它。它下载的资源存储在缓存中,以备后用。大多数现代浏览器31 都支持它,但 Safari（在撰写本文时)除外。由于 HTTP/2,我没有看到预取的使用发生变化。

预载

预加载告诉浏览器以高优先级加载此页面的资源。它是预连接的下一个本地步骤,但与预取不同,它用于此页面上的资源。 Web 浏览器非常擅长在 HTML 中提前扫描并加载所有必要的资源,但预加载允许提前获取未直接包含在页面上的资源（例如 CSS 文件中引用的字体）。支持需要一些时间,但现在许多现代浏览器都支持预加载。 32

预加载与 HTTP/2 推送（在第 5 章中讨论)有一定的关联性,但更多的是因为它被劫持是为了这个目的,而不是作为一个预期的用例。考虑到 HTTP/2 推送的复杂性,预加载资源提示（没有 HTTP/2 推送)可能被证明是一个更简单的选择。目前很多人推荐使用 preload 而不是 HTTP/2 push,在这种情况下一定要在使用链接头时使用nopush属性（HTML 版本不需要使用它,因为 web 服务器不使用它作为推送该资源的指示）。

---

[30] https://caniuse.com/#feat=link-rel-preconnect
[31] https://caniuse.com/#feat=link-rel-prefetch
[32] https://caniuse.com/#feat=link-rel-preload

Machine Translated by Google

当103 Early Hints HTTP 响应代码（在第 5 章中讨论)变得更加普遍时,预加载可能会变得更加有用,因为它可以包含 HTTP 预加载链接标头（即使在 HTTP/1.1 下）。

预渲染器

Prerender 是最昂贵的资源提示。它允许下载和预呈现完整页面（包括页面需要的任何资源）。这个想法是,如果可以可靠地估计下一页导航,则可以立即加载该页面。在撰写本文时,支持端口仅限于 Chrome 和 IE 11,33尽管 Chrome 正在寻求弃用它并且可能不会支持它更长的时间。34 过度使用预渲染的风险相当大,会浪费客户端的带宽和处理时间。我没有看到预呈现的使用因 HTTP/2 而改变,并且不希望它成为其他浏览器实现的优先事项。

6.3.6 减少最后一英里延迟

HTTP/2 旨在通过确保单个 TCP 连接可在请求运行期间用于其他 HTTP 请求来减少延迟的影响。然而,HTTP/2 并未解决延迟问题,仍应尽一切努力减少延迟。网络服务器通常通过高速、高带宽、永远在线的基础设施连接到互联网的其余部分,但浏览网络的用户通常通过不太可靠的连接进行连接,包括宽带和移动网络。最后一英里指的是到最终用户的最后一跳,通常是延迟特别影响连接的地方。

解决此问题的最简单方法是尽可能靠近浏览器,对于全球网站而言,这通常涉及在您的用户群附近建立本地服务器网络。该网络可以是私人管理的网络或（越来越普遍的)CDN。大多数 CDN 支持 HTTP/2.35鉴于升级 Web 服务器以支持它的复杂性和更高的 HTTPS 要求,在 Web 服务器前面的 CDN 是迁移到 HTTP/2 的简单选择（如第 3 章所述)并改善延迟。

6.3.7 优化HTTPS

世界正在转向 HTTPS。出于技术和意识形态的原因,HTTP/2 等新功能需要 HTTPS,因为运行互联网关键组件的人（浏览器供应商、HTTP 工作组等)认为世界应该加密。尽管 HTTPS 最初仅由某些网站或这些网站中的某些网页使用,但网站正朝着对所有页面使用 HTTPS 的方向发展。

---

[33] https://caniuse.com/#feat=link-rel-prerender
[34] https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/0nSxuuv9bBw http://
[35] cdncomparison.com /

Machine Translated by Google

随着 Let s Encrypt36 等免费证书提供商的推出以及 Chrome37和 Mozilla Firefox 38等浏览器供应商的推动,HTTPS 在过去几年中取得了相当大的发展,如图 6.14.39 HTTP/2所示只会提供更多转向它的理由。

图 6.14 过去几年 HTTPS 的增长,由 Let s Encrypt 跟踪（基于 Firefox 遥测统计）

HTTPS 向网页加载添加额外的延迟以建立 HTTPS 会话。此后,延迟最小,因为即使对于移动设备,加密和解密流量所需的计算也可以忽略不计,但初始连接时间会受到影响。您可以使用预连接（在 6.3.5 节中讨论)尝试提前连接以减少对任何依赖域的影响,但这种技术对初始连接没有帮助。

确保您的 HTTPS 设置经过优化以减少建立此 HTTPS 连接所需的时间并提供高级别的安全性（为您的访问者并防止任何浏览器警告)非常重要。优化 HTTPS 设置对于所有使用 HTTPS 的站点（以及所有使用 HTTP/2 的站点)都很重要,因为这意味着您是出于浏览器要求而使用 HTTPS。

以下是确保最佳 HTTPS 使用（使用 HTTP/1 或 HTTP/2)的建议：

[36] https://letsencrypt.org/
[37] https://blog.chromium.org/2018/02/a-secure-web-is-here-to-stay.html
[38] https://blog.mozilla.org/security/2017 /01/20/communicating-the-dangers-of-non-secure-http/
[39] https://letsencrypt.org/stats/#percent-pageloads

Machine Translated by Google

确保只加载 HTTPS 资源以避免混合内容警告。考虑在内容安全策略中使用upgrade-insecure-requests40来实施这种做法,因为它具有强大的浏览器支持。41 确保您的 HTTPS 证书按时更新。过期的证书会阻止访问您的站点。续订过程过去是手动的,但在 Let s Encrypt 的推动下,自动解决方案变得越来越普遍。 Let s Encrypt 只允许 90 天的证书,实际上强制自动化,因为手动更改短期证书比自动化过程更工作。 定期检查您的 HTTPS 设置。 HTTPS 协议、密码和配置经常更改,随着计算能力的增加,新选项被添加而旧选项变得不那么安全。在线 SSLLabs 服务器测试工具42提供了 HTTPS 设置的综合测试,以及已知的 HTTPS 漏洞能力和最佳实践。 A 等级确保您没有问题,并且定期（至少每季度一次)扫描以保持该等级应该确保您没有意外。 实施传输层安全 (TLS) 会话恢复。43 TLS 握手需要大量时间和精力,因此尽量减少它至关重要。最好的方法之一是允许 TLS会话恢复,这样就不需要为每个新连接进行完整的握手。在 HTTP/2 下,应该使用更少的连接,但是以后的或额外的连接（例如有凭证和无凭证的连接)可以提高性能。 TLS 会话恢复引入了一些安全问题,44因为 HTTPS 连接在后续重新连接时可能会变弱（尽管 TLSv1.3 解决了大部分这些问题）,但大多数网站仍希望启用 TLS 恢复以获得可观的性能提升。 不要过分强调安全。安全性很重要,但它始终与性能保持平衡。如果您只允许最新的 TLS 协议和具有最强设置的密码,您将创建一个许多人无法访问的缓慢网站。在撰写本文时,2048 RSA 密钥、TLSv1.2 和 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 密码就足够了,并且得到了很好的支持。这种情况会随着时间的推移而改变,但关键是选择正确的安全级别。

您还可以使用 Mozilla Configuration Generator45为常见的 Web 服务器生成适当的配置,并使用 SSLLabs 工具扫描其他站点以查看它们使用的设置以及您如何进行比较。我经常自己扫描ssllabs.com站点,假设 SSLLabs 最了解如何配置它。

---

[40] https://www.w3.org/TR/upgrade-insecure-requests/
[41] https://caniuse.com/#feat=upgradeinsecurerequests
[42] https://www.ssllabs.com/ssltest/ https://
[43] calendar.perfplanet。 com/2014/speeding-up-https-with-session-resumption/ https://
[44] timtaubert.de/blog/2014/11/the-sad-state-of-server-side-tls-session-resumption-implementations/ https://
[45] mozilla.github.io/server-side-tls/ssl-config-generator/

Machine Translated by Google

考虑您是否最适合处理 HTTPS 设置。 HTTPS 很复杂。
将此功能外包给专家或使用云提供商或 CDN 比自己管理更容易,并且可以确保您的用户始终看到强大的 HTTPS 设置。但是,当您真正将密钥交给 HTTPS 设置时,您需要确保您信任对方;否则,你就失去了安全感!

在可用时启用 TLSv1.3。该协议于 2018 年 8 月标准化,46但最初可能无法供许多读者使用。与以前的版本相比,它包含许多性能（和安全性)改进。 47

HTTPS 将继续存在。许多站点已经在 HTTPS 上,并且前面提到的要点无论如何都适用于它们。需要 HTTPS 的 HTTP/2 为其他网站提供了另一个转向 HTTPS 的理由。然而,HTTPS 需要一些管理,并且由网站所有者决定如何最好地做到这一点。现在,无论是否使用 HTTP/2,每个运营网站的人都应该认真考虑这里提到的几点。

6.3.8 非 HTTP 相关的网络性能技术

在本节中,我将重点介绍与通过 HTTP 传输资源相关的技巧和技术。许多其他网络性能改进与数据下载方式无关。性能不佳的 JavaScript 尤其容易使网站变慢。加载大量广告网络和跟踪器可以使用您的网站可能需要的资源。低规格的服务器可能难以应对网站收到的大量请求。这些主题远远超出了本书的主题,但是认为只要您通过使用此处的一些技巧和技术来优化您对 HTTP 的使用,您就永远不会遇到任何性能问题,这是不正确的。

与 HTTP 相关的性能技术很重要,任何 Web 性能资源（书籍、博客或讲座)都非常关注它们,但它们并不是 Web 性能的最终目的和全部。因此,如果您可以在查看网站或 Web 应用程序的其他区域时获得更大的收益,请确保您没有花费太多时间来优化 HTTP 使用。

6.4 针对 HTTP/1.1 和 HTTP/2 进行优化

鉴于浏览器对它的强大支持,

HTTP/2 应该对大多数访问者可用。48然而,一些用户由于使用旧浏览器或浏览器可能不容易更新的旧设备（例如手机）。或者代理（包括防病毒软件)可能会降低浏览器和服务器之间的连接。如果你在本章中使用了任何 HTTP/2 特定的技术（比如连接更少）,那些用户会发生什么?好消息是,即使您开始针对 HTTP/2 进行优化,您的网站仍应为他们工作。

---

46 https://tools.ietf.org/html/rfc8446
47 https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/
48 https://caniuse.com/#feat=http2

Machine Translated by Google

在最坏的情况下,如果您删除 HTTP/1 优化,站点可能会变慢,但它不应该中断。然而,在可预见的未来,直到您网站上的 HTTP/1 使用变得更小,您可能希望同时针对 HTTP/1 和 HTTP/2 进行优化。

6.4.1 测量 HTTP/2 流量

您应该做的第一件事是测量使用每种协议的流量。

此时,我假设您已经升级到 HTTP/2 但尚未更改您的站点,因此您仍然拥有 HTTP/1.1 优化。如果您的绝大部分流量已经在 HTTP/2 上,那么担心 HTTP/1 流量可能没有什么意义。该网站仍然可以运行,但会比理想情况慢一点。

测量 HTTP/2 流量的最简单方法是将其记录在您的 Web 服务器日志中。在 Apache 中,您可以在 LogFormat指令中添加此数据,通常在主 httpd.conf 文件（或某些发行版上的 apache2.conf)中设置,方法是向日志中添加%H

格式:

```
日志格式“%h %l %u %t %{ms}T %H \    %r\    %>s %b \    %{Referer}i\
   %{User-Agent}i\    %{SSL_PROTOCOL}x %{SSL_CIPHER}x
%{Content-Encoding}o %{H2_PUSHED}e    合并
CustomLog /usr/local/apache2/log/ssl_access_log 合并
```

访问日志中出现以下内容:

```
78.17.12.1234 - - [11/Mar/2018:22:04:47 +0000] 3 HTTP/2.0 “GET / HTTP/2.0”
200 1847    -       Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3)
          AppleWebKit/537.36（KHTML,如 Gecko)Chrome/64.0.3282.186 Safari/537.36    TLSv1.2 ECDHE-
          RSA-AES128-GCM-SHA256 br
```

在这里,您会看到协议(HTTP/2.0)在请求(GET / HTTP/2.0) 之前被记录。由于 Apache 在日志中以 HTTP/1 样式格式打印请求,因此可以在请求行(%r) 中获取协议,但使用%H在日志文件中单独列出它可能更容易,以便您可以更容易地解析它。nginx 允许使用类似的方法记录协议,使用$server_protocol

环境变量:

```
log_format my_log_format    $remote_addr - $remote_user [$time_local]       $server_protocol    $request
                                $status $body_bytes_sent          $http_referer       $http_user_agent       ; access_log /
                                usr/local/nginx/nginx-access.log my_log_format;
```

如果您使用的是其他网络服务器,请查阅您的网络服务器文档。

边缘服务器的重要性如果您在多个 Web
服务器之前使用负载均衡器,您可能需要确定在负载均衡器上测量的协议,具体取决于您拥有的负载均衡器类型。

Machine Translated by Google

边缘服务器的重要性（续）

HTTP 负载平衡器（也称为第 7 层负载平衡器,遵循第 1 章中提到的 OSI 模型)终止负载平衡器上的 HTTP 连接,并建立从负载平衡器到实际 Web 服务器的另一个 HTTP 连接。

因此,如果您在 Web 服务器上测量此协议,Web 服务器日志会显示用于该负载均衡器 - Web 服务器连接的协议,但此协议可能与主客户端 - 负载均衡器连接不同,后者您可能更感兴趣。在这种情况下,您应该测量负载平衡器而不是 Web 服务器上的协议使用情况。

TCP 负载平衡器（也称为第 4 层负载平衡器)在 TCP 级别工作,并将 TCP 数据包（HTTP 消息)的有效负载转发到下游 Web 服务器。因此,HTTP 消息是原始消息,协议可以在 Web 服务器级别进行测量。

所谓的边缘服务器是您感兴趣的协议（在本例中为 HTTP)用户的入口点,因此当您尝试测量流量时,了解您的基础设施和适当的地方进行测量。

当您记录使用的协议时,您可以分析日志以查看使用每种协议的流量百分比。如果您使用的是基于 Linux 或 UNIX 的系统,则可以结合使用grep、 sort和uniq来轻松完成此操作：

```
$ grep -oh    HTTP\/[0-9]\.[0-9]*    ssl_access_log |排序 | uniq -c 196 HTTP/1.0 1182 HTTP/1.1
        5977 HTTP/2.0
```

在这里,您看到您仍然获得少量 HTTP/1.0 流量（这往往是机器人而不是真实流量）。其余的,16% 是 HTTP/1.1,81% 是 HTTP/2。

6.4.2 在服务器端检测 HTTP/2 支持

假设 HTTP/1 访问者仍然占您网络流量的相当大一部分,您可能想要检测当前连接是通过 HTTP/1.1 还是 HTTP/2 并为每种类型提供不同的响应。 HTTP/1 用户可以获得完全串联的资源,甚至可以从分片域加载资产,而 HTTP/2 用户可以获得更少的串联资源,并从主域加载所有内容。

能够以不同方式处理这两种协议需要了解传入连接使用的协议。在衡量协议使用时,您需要衡量边缘服务器正在使用的协议（请参阅上面的边栏“边缘服务器的重要性”）,这取决于该边缘服务器的功能。向下游发送此信息可能很重要。

大多数 Web 服务器设置各种环境变量,可用于做出决定和更改配置。 CGI 和 PHP 脚本可以访问 SERVER_PROTOCOL环境变量,该变量应根据需要设置为 HTTP/1.1 或 HTTP/2.0。

Machine Translated by Google

一些网络服务器设置了额外的变量。例如,Apache 服务器设置可以使用的环境变量49 如表 6.7 所示。

表 6.7 Apache HTTP/2 环境变量

变量的名称	值类型	描述
HTTP2	旗帜	正在使用 HTTP/2。
H2PUSH	旗帜	此连接启用 HTTP/2 服务器推送,客户端也支持。
H2_PUSH	旗帜	此名称是 H2PUSH 的替代名称。
H2_PUSHED	细绳	对于服务器推送的请求,此变量为空或 PUSHED。
H2_PUSHED_ON	数字	此变量标识触发此请求推送的 HTTP/2 流编号。
H2_STREAM_ID	数字	此变量是此请求的 HTTP/2 流编号。
H2_STREAM_TAG 字符串		该变量是 HTTP/2 进程的唯一流标识符,由连字符分隔的连接 ID 和流 ID 组成。

这些环境变量可用于 Apache 配置以及 CGI 和 PHP 脚本。它们也可以用在LogFormat指令中,因此值得向自定义日志格式添加%{H2_ PUSHED}e部分,就像我之前所做的那样,以跟踪日志文件中推送的资源。

nginx 网络服务器有一个$http2变量50,当通过 HTTPS 使用 HTTP/2.0 时设置为h2 （因为所有浏览器连接都是）,或者当使用未加密的 HTTP/2 连接时设置为h2c 。在撰写本文时,nginx 不再提供任何环境变量,例如是否启用推送。

许多人还使用 Apache 和 nginx 作为反向代理,将请求代理到下游应用程序（例如 Node 或基于 Java 的应用程序服务器,例如 Tom cat）。在 Apache 中,您使用ProxyPass指令：

```
ProxyPass /webapplication/ http://localhost:3000/
```

在这种情况下,因为这些系统位于下游,所以它们无权访问 Apache 环境变量。但是,可以使用RequestHeader指令设置额外的 HTTP 标头以通知下游系统：

```
#设置一个 HTTP_VERSION 变量,因为 Apache 没有这个变量 #（SERVER_PROTOCOL 和 Request_Protocol 不是完整的环境变量）

SetEnvIf Request_Protocol  (.*)  HTTP_VERSION=$1
```

[49] https://httpd.apache.org/docs/2.4/mod/mod_http2.html#envvars
[50] http://nginx.org/en/docs/http/ngx_http_v2_module.html#variables

Machine Translated by Google


```
#然后用这个变量设置一个HTTP Header给下游系统看
RequestHeader 设置协议“%{HTTP_VERSION}e”
#添加一些其他预定义的 HTTP2 变量
RequestHeader 设置 http2 %{HTTP2}e
RequestHeader 设置 h2push %{H2PUSH}e
ProxyPass /webapplication/ http://localhost:3000/
```

下游应用程序可以像读取任何其他 HTTP 标头一样读取这些 HTTP 标头。

nginx 的类似语法是

```
位置/webapplication/{
        proxy_set_header 协议 $server_protocol; proxy_set_header http2
        $http2; proxy_pass http://localhost:3000;

}
```

以节点为例,您可以返回到第 5 章的简单服务器并添加两行额外的行来记录 HTTP/2 支持,如以下清单所示。

清单 6.1 带有 HTTP 标头检查的节点

```
使用严格

var http = require(  http  ) 常量端口 = 3000


const requestHandler = (request, response) => { const { headers } = 请求;
    console.log(  HTTP 版本:  + headers[  protocol  ]) console.log(  HTTP2
    支持:  + headers[  http2  ]) console.log(  HTTP2推送支持:  + headers[  h2push  ])
    response.setHeader(  Link  ,  </assets/css/common.css;rel=preload response.writeHead(200,
    {  Content-Type  :  text/html  }) response.write (  <!DOCTYPE html>\n response.write(  ) <html>\n  )
    response.write(  <head>\n  ) response.write(  <link rel=  stylesheet  type=  文字/CSS”
    CSS;rel=preload;as=style
```

Machine Translated by Google

通过正确配置的 Apache 服务器从浏览器访问/webapplication/ ,在节点日志中打印以下内容：

```
HTTP 版本:HTTP/2.0 HTTP2 支持:开
启HTTP2 推送支持:开启
```

HTTP/1 浏览器打印：

```
HTTP 版本:HTTP/1.1 HTTP2 支持:
（空）
HTTP2 推送支持:（空）
```

nginx 打印的内容略有不同：

```
HTTP 版本:HTTP/2.0 HTTP2 支持:h2
HTTP2 推送支持:未定义
```

通过 nginx 的 HTTP/1.1 请求如下所示：

```
HTTP 版本:HTTP/1.1 HTTP2 支持:
未定义HTTP2 推送支持:未定义
```

另一种方法是将详细信息作为查询参数而不是 HTTP 标头传递。但我发现 HTTP 标头更清晰,更容易为GET和POST添加

要求。无论如何,您可以看到可以知道使用的是什么协议,并让您的应用程序根据协议做出不同的反应。

6.4.3 在客户端检测 HTTP/2 支持

客户端应用程序可能还想知道您使用的是 HTTP/1 还是 HTTP/2。目前,没有获取此信息的标准方法,尽管 Resource Timing Level 2 API 包含提供此信息的nextHopProtocol[51]属性。

然而,由于中间代理的存在,衡量客户认为可用的东西是很棘手的。可能是 Web 浏览器仅限于 HTTP/1.1,但代理通过 HTTP/2 连接到服务器（尽管在现实中,情况可能恰恰相反,浏览器支持 HTTP/2 但被阻止)由于代理到 HTTP/1.1）。出于这个原因,我认为检测服务器端使用的协议然后将结果发送回客户端总是更好。您可以分几次将结果发回

方式,包括 HTTP 标头或 JavaScript 变量。唯一需要注意的是要考虑缓存指示此信息的资源可能会如何影响未来的连接。如果您从 HTTP/2 连接开始并缓存客户端用于指示连接的资源,然后客户端切换到 HTTP/1 连接,则您的优化可能不正确。

[51] https://www.w3.org/TR/resource-timing-2/#dom-performanceresourcetiming-nexthopprotocol

Machine Translated by Google

6.4.4 连接合并

HTTP/2 规范允许多个域重复使用相同的 HTTP/2 连接,前提是它们对每个域都是权威的[52] 也就是说,如果域解析到相同的 IP 地址和 HTTPS 证书涵盖两个域。意向是最大化单个连接并允许自动拆分托管在同一服务器上的域（也称为连接合并）。

假设您位于 www.example.com 的站点使用 images.example.com 域来托管图像。如果这两个域托管在同一台服务器上,并且只是该服务器上的单独虚拟主机,在 HTTP/2 下,如果您设置适当的:authority伪标头,它们可以由同一连接提供服务。如果您创建这些分片域纯粹是由于 HTTP/1 效率低下并且没有为它们使用单独的服务器,则可能会发生这种情况,如图 6.15 所示。

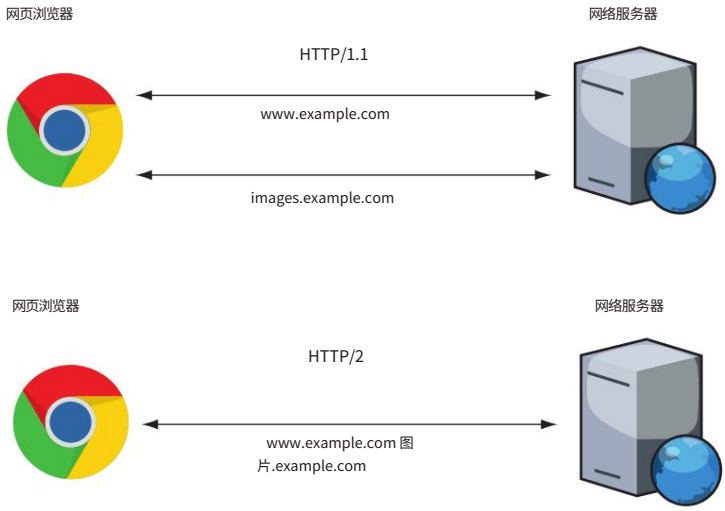


图 6.15 HTTP/2 下的连接合并

在更高级别（例如在浏览器的开发人员工具中）,HTTP 请求看起来与它们被分片完全相同;只有在连接级别,客户端才能决定合并。网站可以继续拥有 HTTP/1.1 连接将自动使用的分片域,并且 HTTP/2 连接将自动合并,就好像它们是非分片域一样,所有内容都由一个连接提供。这种情况听起来很理想,不需要额外的努力来继续支持 HTTP/1 用户,或者针对 HTTP/2 用户进行优化。与往常一样,现实有点棘手。

[52] https://httpwg.org/specs/rfc7540.html#reuse

Machine Translated by Google

首先,此过程仅在您将这些域托管在同一台服务器上时才有效。如果服务器是分开的,则它们需要有单独的连接。另一个警告是浏览器需要实现连接重用,但并非所有浏览器都这样做。53规范说连接可以重用,而不是必须重用。 Safari 和 Edge 在撰写本文时并未合并,而 Chrome 和 Firefox 合并。

此外,此功能可能会导致问题,具体取决于浏览器如何实现它。对于域的多个 IP 地址（其中一些可能与其他域共享）,浏览器可能认为它可以重用（或合并)一个连接,而实际上它不能。假设您拥有表 6.8 中所示的 IP 地址。

表 6.8 连接合并示例

领域	IP地址
www.example.com	1.2.3.4 1.2.3.5
images.example.com	1.2.3.4 1.2.3.6

在这种情况下,IP 地址 1.2.3.4 上的任何连接都可以服务于 HTTP/2 下的 www.example.com 和 images.example.com 请求,如果客户端想要的话,但其他 IP 地址（1.2. 3.5 和 1.2.3.6) 不能。 Firefox 已经实施了积极的连接合并,并且会尝试为两个域使用任何连接,无论三个 IP 地址中的哪个用于该实际连接,如果它知道这种重叠。这种情况会导致错误,正如 BBC 在最初迁移到 HTTP/2 时所注意到的那样。54已创建新的 HTTP 状态代码421以允许服务器礼貌地告诉浏览器它使用了错误的连接并再次查看它应该在哪里发送这些请求。但是,正如 BBC 发现的那样,对该状态代码的支持仍然有限。作为替代解决方案， ORIGIN frame55允许服务器通知客户端它对哪些域具有权威性,而不是让客户端

猜测。该框架在撰写本文时是新的,但一些服务器已经支持它,56,其他服务器已公开请求跟踪它。 58、59在浏览器方面,Firefox60已经支持框架[57]岁和框架,并且其他浏览器现在有望

53 https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing/
54 https://medium.com/bbc-design-engineering/http-2-is-easy-just-turn- it-on-34baad2d1fb1
55 https://tools.ietf.org/html/rfc8336 https://github.com/nghttp2/nghttp2/pull/901 https://
56 github.com/h2o/h2o/pull/1199 https: //github.com/icing/mod_h2/issues/96 https://
57 trac.nginx.org/nginx/ticket/1530 https://bugzilla.mozilla.org/show_bug.cgi?id=1337791
58
59
60

Machine Translated by Google

已被批准为提议的标准。61此帧可能会在连接开始时发送（理想情况下,因为 HTTP/2 SETTINGS 帧已发送）,这也应防止发送错误的请求。与此相关的是,HTTP/262 中的二级证书身份验证是另一个提议,即使在使用不同证书但 IP 地址相同时也允许连接合并。

简而言之,连接合并很复杂,所以我建议不要依赖它。
相反,看看是否需要分片以及它是否会给您带来性能优势。如果要保留分片,最好将其托管在单独的服务器上以防止合并并发症。

6.4.5 为 HTTP/1.1 用户优化多长时间

为两组用户进行优化的最终考虑是继续做多长时间。优化需要付出额外的努力,因此请考虑是否要执行这些额外的工作,如果是,需要多长时间。替代方案是可用的,包括暂时不撤销 HTTP/1.1 变通办法,因为它们在 HTTP/2 下应该不会更糟,但不再是必要的。或者,如果您的大部分访问者已经在 HTTP/2 上,那么以牺牲少数人为代价为大多数人进行优化可能会更好,并让 HTTP/1.1 连接上的访问者承受比绝对必要的更慢的负载。每个网站所有者都应根据自己的基础和任何更改的预期效果做出此决定。由于浏览器对 HTTP/2 的支持很强,不能使用 HTTP/2 的用户分为几类:旧版本）

公司代理背后的人（可能连接速度更快） 防病毒代理背后的人（可能是桌面用户,主要是广泛的
乐队）
那些使用模糊浏览器的人（可能有其他渲染问题
你的网站）
机器人和爬虫（你可能对保持快乐不太感兴趣）

最终,值得看看您的哪些访问者仍在使用 HTTP/1,然后决定是否值得为这两种协议进行优化。对于大型站点,答案可能是肯定的,但对于较小的操作,这项工作可能不值得。如果您不再针对 HTTP/1 进行优化,该站点仍然可以正常工作（尽管速度更慢)这一事实使这个选项值得考虑。以类似的方式,在网站设计中使用优雅降级,为不支持最佳体验所需功能的浏览器提供工作但未优化的网站。

---

61 https://tools.ietf.org/html/rfc8336
62 https://tools.ietf.org/html/draft-ietf-httpbis-http2-secondary-certs

Machine Translated by Google

概括

HTTP/2 旨在解决 HTTP/1.1 性能低下的问题。 希望不再需要 HTTP/1.1 性能优化,这需要付出努力并有缺点。这个希望只是部分实现了。虽然您应该不太需要使用这些技术,但删除它们可能还为时过早。 其他网络性能技术在 HTTP/2 下仍然很重要,但在迁移到 HTTP/2 时,可能值得为您的站点重新访问它们。

可以针对 HTTP/1.1 连接和 HTTP/2 进行优化。 连接合并允许浏览器自动取消对域的分片,但是

这个过程很复杂。

Machine Translated by Google

Machine Translated by Google

第 3 部分

高级 HTTP/2

本书的第一部分介绍了 HTTP/2。第二部分看了 HTTP/2 的细节以及它应该如何使用。第 4 章介绍了 HTTP/2 的基础知识及其工作原理;第 5 章专门介绍 HTTP/2 推送,这是 HTTP 的一个新概念;第 6 章探讨了 HTTP/2 应该和不应该如何改变开发实践。

在这一部分中,我会进行更深入的研究,涵盖一些很少有人真正理解的高级主题。第 7 章完成了之前未涵盖的规范部分,第 8 章介绍了单独的 HPACK HTTP 标头压缩规范。这两章将帮助您将坚实的基础转变为对 HTTP/2 协议的专家级理解。

您将能够解决任何 HTTP/2 问题,甚至可能有助于推动协议的发展!

Machine Translated by Google

Machine Translated by Google

7 先进的 HTTP/2 概念

本章涵盖 HTTP/2 流状态
HTTP/2 中的流量控制
HTTP/2 中的优先级 HTTP/2 一致性测试

本章涵盖了 HTTP/2 协议的其余部分,大致按照它们在规范中出现的顺序排列。[1] 其中许多部分不受 Web 开发人员的直接控制,甚至可能不受服务器控制管理员（除非他们自己编写 HTTP/2 服务器）,所以这些顶级 ics 肯定更高级。但是,如果您希望实现自己的 HTTP/2 服务器,了解它们将使您深入了解协议的工作原理并帮助调试。此外,在未来,开发人员或至少 Web 服务器管理员可能会获得更多控制权。

第 8 章着眼于 HPACK 协议,它是一个独立于 HTTP/2 的规范。

[1] https://httpwg.org/specs/rfc7540.html

Machine Translated by Google

7.1 流状态

为单次下载创建 HTTP/2 流,然后丢弃。这就是为什么 HTTP/2 流不是 HTTP/1.1 连接的精确模拟的原因之一,尽管这可能是第一次教授 HTTP/2 时解释它们的最简单方法。许多图表将 HTTP/2 流和 HTTP/1 连接进行了比较（就像我在第 2 章中使用并在图 7.1 中重复的那样），但这种约定并不严格正确,因为流没有被重用。

流完成其资源的传送后,流将关闭。当请求新资源时,将启动一个新流。流是一个虚拟的概念并且是

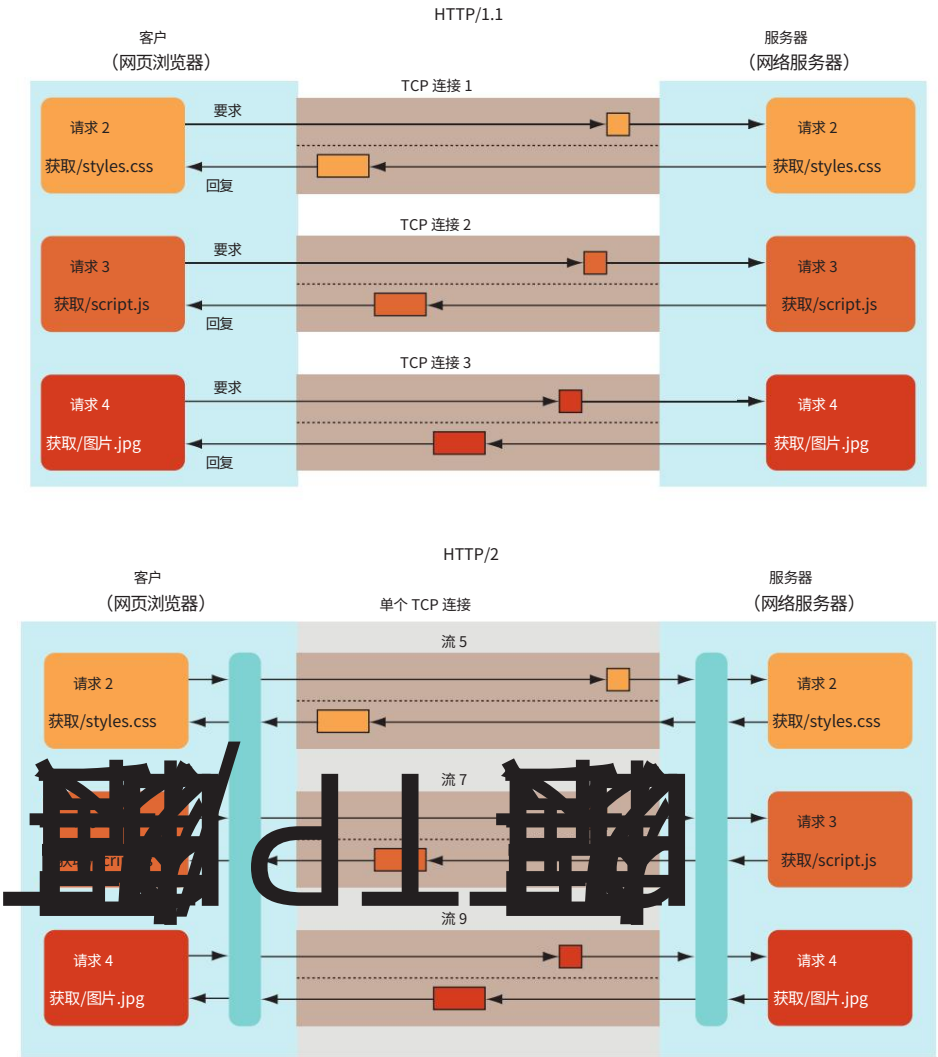


图 7.1 HTTP/1.1 连接和 HTTP/2 流可以表示为相似,即使它们不同。

/dl上删上删

Machine Translated by Google

只不过是每个帧标记的数字,称为流标识符。
因此,关闭流或打开新流的成本远低于打开 HTTP/1.1 连接的成本（后者涉及 TCP 三向握手和发送请求之前的可选 HTTPS 协议协商）。

事实上,HTTP/2 连接比 HTTP/1 连接的成本更高,因为它们额外需要 HTTP/2 “神奇”的前言消息和至少一个SETTINGS帧才能发出单个请求。 HTTP/2 流要便宜得多。

HTTP/2 流经历一个状态生命周期。客户端发送的HEADERS帧启动 HTTP 请求（例如GET请求）,服务器响应该请求,然后关闭流。此过程经历以下状态：

空闲　当流被创建或被引用时。实际上,大多数流不会长时间保持这种状态,因为除非您打算使用它,否则很少引用流,因此大多数空闲流会立即使用,然后立即进入下一阶段:打开。

Open　当流已被用于发送请求的HEADERS帧时,该流被认为是开放的并且可用于双向通信。

当客户端仍在发送数据时,流保持状态。因为大多数 HTTP/2 请求可以在单个HEADERS帧中发送,所以当该帧已发送时,流很可能进入下一阶段（半关闭）。 半封闭　当客户端用END_STREAM标志表明请求HEADERS帧包含了它想要从这个请求中得到的所有内容时,流被认为是半封闭的,应该只用于将响应发送回客户端;它不应该用于从客户端发送更多数据（除了控制帧,例如WINDOW_UPDATE）。 关闭当服务器完成发送并在最后一帧使用END_STREAM标志时,流被认为已关闭,不应使用

了。

尽管此列表解释了简单的客户端发起的 HTTP 请求的状态转换,但同样的情况也可能发生在服务器发起的请求中。目前,只有HTTP/2 推送响应是服务器发起的（尽管没有什么可说的,未来的一些新框架将不会是服务器发起的）。在这种情况下,一个流启动另一个流（承诺的流标识符）,并且新的承诺流经历类似的状态转换：

空闲　当承诺的流首次创建或被另一个流上发送的PUSH_PROMISE帧引用时。

Reserved　当推送的流立即进入reserved状态,直到服务器准备好推送资源。你知道流存在（所以它至少是空闲的）;您知道它将用于特定资源（因此它不仅仅是空闲的,因此是保留状态）;但是您不知道该资源是什么的全部详细信息,就像在HEADERS框架被删除后的第一个示例中一样

Machine Translated by Google

已收到。但是,因为它用于推送资源,所以流永远不应该处于打开状态,因为您永远不会期望客户端在此流上发送数据。

它应该被保留,然后在发送HEADERS帧时进入半关闭状态（在原始流上发送PUSH_PROMISE帧之后）。

这个状态,并非巧合,是下一个状态。

半关闭 当服务器开始推送响应时,承诺的流进入半关闭状态,应该只用于发送该推送资源的数据。 关闭 当服务器完成发送并在最后一个数据帧上使用了END_STREAM标志时,流被认为是关闭的,不应该被使用

了。

完整的 HTTP/2 状态图如图 7.2 所示,包括前面列表中指示的两个流和其他几种可能性（例如当RST_STREAM帧用于过早结束连接时）。

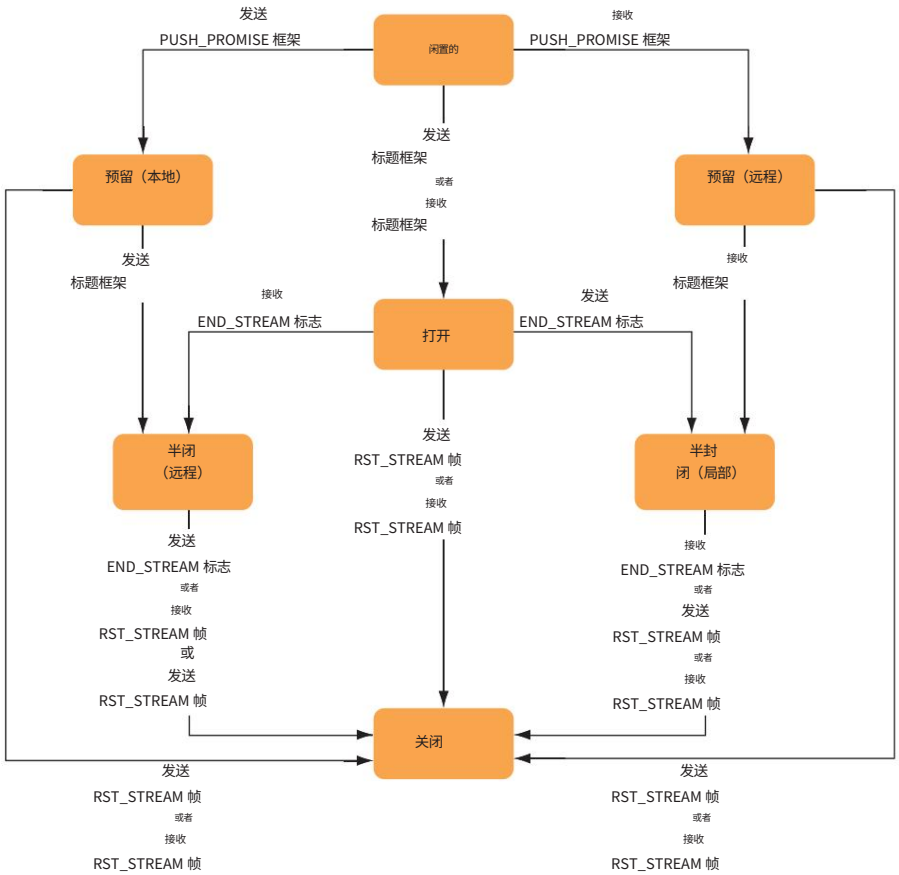


图 7.2 HTTP/2 流状态图

Machine Translated by Google

在这些流中的每一个中,客户端和服务器对流状态的看法略有不同,这取决于它是启动该状态还是根据来自另一端的消息移动到该状态。因此,其中一些状态具有本地或远程指示符（分别取决于您是流的发起者还是接收者）,并且每个状态都有发送和接收转换。

回到GET请求的第一个示例,您知道它会经历以下状态:空闲、打开、半关闭和关闭。然而,半关闭状态是有歧义的:它对客户端关闭（因此它不能发送数据,只能接收数据）,但它不是对服务器半关闭。客户端将流视为半关闭（本地）,而服务器将其视为半关闭（远程）。因此,对于客户端和服务器,请求不会以相同的方式沿着状态图向下流动;它同时流向图表的左侧或右侧,具体取决于您查看的是客户端视图还是服务器视图。

此外,状态图仅显示状态转换。某些帧不会导致状态转换。例如， CONTINUATION帧被认为是前面的HEADERS帧的扩展,因此在图中它们被认为是HEADERS的一部分。同样,其他帧（例如PRIORITY、 SETTINGS、 PING和WINDOW_UPDATE)永远不会导致状态转换,因此未在此图中捕获。

老实说,HTTP/2 状态图对于大多数 HTTP/2 用户来说并不重要,低级 HTTP/2 库的实现者更关心的是了解什么帧可以发送,什么帧不能发送状态。该图在 HTTP/2 规范中,2但是,该规范中引用了很多不同的状态,因此理解它会有所帮助。 HTTP/2 不允许的任何状态转换尝试都应导致PROTOCOL_ERROR消息。同样,理解状态图可以帮助您理解为什么会出现这样的错误（尽管这个错误通常是由于底层 HTTP/2 实现中的错误造成的,并且超出了大多数 Web 开发人员可以自行修复的范围）。

HTTP/2 状态图起初可能令人生畏,而且与我目前介绍的一些概念不同,它不是您可以在浏览器的开发人员工具中看到的,甚至无法通过使用本书中的其他一些工具看到（例如 nghttp 和 Wire shark）。它更像是 HTTP/2 实现需要维护和跟踪的内部状态。鉴于这一事实,理解起来可能会很复杂。回到主要用例（请求 HTTP 资源）,然而,如上图所述,通常会揭开一些谜团。

7.2 流量控制

流量控制是网络协议的重要组成部分。它允许接收方在发送方尚未准备好处理时阻止发送方向其发送数据,这可能是因为它太忙而无法处理更多传入数据。流量控制是必要的,因为不同的客户端可以以不同的速度消费数据。高速服务器可能能够发送数据

---

[2] https://httpwg.org/specs/rfc7540.html#StreamStates

Machine Translated by Google

很快,但是如果一个低速客户端（比如手机)跟不上,它就会开始在内存中缓冲数据,当缓冲区被填满时,它就会开始丢弃数据包,要求它们被发送再次。结果,服务器端、网络和客户端都浪费了资源。

HTTP/1.1 中不需要流量控制,因为任何时候只有一条消息在传输。因此,可以在连接级别使用TCP 流量控制。如果接收方停止使用 TCP 数据包,它就不再确认这些数据包,并且发送方停止发送它们,因为它的 TCP 拥塞窗口（CWND)将用完（见第 2 章）。

在 HTTP/2 中,您拥有独立流的多路复用连接,因此连接级别的流量控制不再足够。控制需要处于连接级别和流级别。可能您很高兴在一个流上接收到更多数据而不是另一个。第 4 章提供了一个带有用户暂停视频的网站示例。在这种情况下,您可能不希望视频在暂停时继续下载,但您希望继续使用 HTTP/2 连接上的其他流。

流量控制在 HTTP/2 中的处理方式与 TCP 类似。在连接开始时（使用SETTINGS帧）,流量控制窗口的大小是确定的（如果未指定大小,则使用默认的 65,535 个八位字节）。然后从总数中减去每条发送的数据,并加回确认的每一位数据（通过WINDOW_UPDATE帧）。有一个连接级流量控制窗口,它是 TCP 流量控制窗口的一种镜像,每个流也有一个。发送方最多只能发送最小流量控制窗口（连接级别或该流)的最大大小,并且当流量控制窗口达到零时,发送方必须停止发送数据,直到收到确认,从而导致非零流量控制窗口。如果您实施 HTTP/2 客户端或服务器而忘记实施WINDOW_UPDATE帧,您很快就会注意到对方停止与您交谈!

流量控制用于数据帧（尽管未来的 HTTP/2 帧类型也可能属于流量控制）。当客户端停止确认帧时,仍然可以发送控制帧（特别是控制流控制所需的WINDOW_UPDATE帧）。

7.2.1 流量控制示例

对于流量控制的示例,我将返回使用 nghttp 工具。在本节中,我们向 Facebook 发出主页和所有必需资源的请求,然后将其通过管道传输到grep以仅显示重要部分:

```
$ nghttp -anv https://www.facebook.com | grep -E     frame <|SETTINGS|
window_size_increment     [0.110] recv SETTINGS 帧<length=30, flags=0x00,
 stream_id=0>
                    [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
                    [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
                    [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
```

Machine Translated by Google

[SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
[SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536] [0.110] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0> (window_size_increment=10420225) [0.110]发送 SETTINGS frame <length=12, flags=0x00, stream_id=0 >

[SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
[SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]

在这里,您看到 Facebook 服务器已决定使用 65,536 个八位字节的流量控制窗口大小（ recv SETTINGS 框架中的SETTINGS_INITAL_WINDOW_SIZE值）,而 nghttp 使用 65,535 个八位字节（发送方 SETTINGS框架中的SETTINGS_INITAL_WINDOW_SIZE值）。顺便说一句,65,535 也是默认大小,因此 nghttp 根本不需要发送它。代码也说明了两边可以有不同的

流控制窗口大小（即使它们在这里足够接近相同,仅相差 1 个八位字节）。

在这两个SETTINGS框架的中间,您会看到第一个WINDOW_UPDATE框架（在代码中突出显示）：

[0.110] recv WINDOW_UPDATE帧<长度=4,标志=0x00， stream_id=0> （window_size_increment=10420225）

此帧声明 Facebook 准备接收最多 10,420,225 个八位字节,并且由于该帧是在流 0 上发送的,因此此限制是要在所有流中使用的连接级别限制,以及它们的流级别限制。 Stream 0 永远不应该用于DATA帧并且不需要它自己的流量控制,这就是它可以用于连接级流量控制的原因。这 10,420,225 个允许的八位字节在初始窗口大小的 65,535 个八位字节之上,因此 Facebook 也可以将初始大小设置为两者的总和 (10,485,761)，但也允许以这种方式实现。

接下来,nghttp 确认服务器的设置,然后是 nghttp 设置优先级的更多帧（顺便说一句,可以在空闲状态下创建帧并保持在那里直到使用的少数实例之一）：

[0.110]发送设置帧<长度=0,标志=0x01,stream_id=0>[0.110]发送优先级帧<长度=5,标志=0x00，stream_id=3>[0.110]发送优先级帧<长度=5,标志=0x00, stream_id=5> [0.110] 发送优先级帧 <length=5, flags=0x00, stream_id=7> [0.110] 发送优先级帧 <length=5, flags=0x00, stream_id=9> [0.110] 发送优先级帧 <length=5, flags=0x00, stream_id=11>

接下来我将讨论PRIORITY帧,所以暂时忽略它们。

接下来,您会看到第一个请求是通过流13上的HEADERS帧发送的：

[0.110]发送标题帧<长度=43,标志=0x25， stream_id=13>

回想一下,客户端发起的流必须具有奇数流标识符。

流 13 是下一个空闲流,因为 11 已被最后一个PRIORITY帧使用。

Machine Translated by Google

然后服务器确认SETTINGS帧和另一个WINDOW_UPDATE帧,将流 13 的窗口大小增加到 10,420,224 个八位字节（奇怪的是,比连接级大小小 1 个八位字节,但没有说大小必须相同）：

```
[0.134]接收设置帧<长度=0,标志=0x01,stream_id=0>[0.134]接收WINDOW_UPDATE帧<长度=4,标志=0x00，
stream_id=13> （window_size_increment=10420224）
```

接下来,nghttp 开始接收资源的HEADERS和DATA帧：

```
[0.348]接收头帧<长度=293,标志=0x04,stream_id=13>[0.349]接收数据帧<长度=1353,标志=0x00,流ID=13>[0.350]
接收数据帧<长度=2571,标志=0x00,stream_id=13> [0.351]接收数据帧<长度=8144,标志=0x00,stream_id=13>[0.374]
接收数据帧<长度=5563,标志=0x00,stream_id=13>[0.375]接收数据frame <length=2572, flags=0x00,
stream_id=13> [0.376] recv DATA frame <length=1491, flags=0x00, stream_id=13> [0.377] recv DATA frame
<length=2581, flags=0x00, stream_id= 13> [0.378]接收数据帧<长度=4072,标志=0x00,stream_id=13> [0.379]
接收数据帧<长度=5572,标志=0x00,stream_id=13>
```

在收到几个数据帧后,nghttp 决定让服务器知道它已经消耗了那么多数据。仅将数据帧相加（1353 + 2571 + 8144 + 5563 + 2572 + 1491+ 2581 + 4072 + 5572)得到 33,919,这就是 nghttp 告诉服务器它在连接级别（流 0)和流 13 上消耗的内容:

```
[0.379]发送WINDOW_UPDATE帧<长度=4,标志=0x00， stream_id=0> （window_size_increment=33919） [0.379]
                    发送WINDOW_UPDATE帧<长度=4,flags=0x00， stream_id=13> （window_size_increment=33919）
```

重要的是要注意只有DATA帧有效负载的长度（由长度字段给出)被包括在流控制计算中,九个八位字节的帧头被排除在流控制之外。

连接以类似的方式继续,直到所有资源都已交付，
并且客户端发送非常礼貌的GOAWAY帧关闭连接：

```
[0.381]接收数据帧<长度=2563,标志=0x00,stream_id=13>[0.382]接收数据帧<长度=1491,标志=0x00，
stream_id=13>[0.384]接收数据帧<长度=2581,标志=0x00,stream_id=13> [0.398]接收数据帧<长度=4072,标
志=0x00,stream_id=13>[0.400]接收数据帧<长度=2332,标志=0x00,stream_id=13>[0.402]接收数据frame
<length=1491, flags=0x00, stream_id=13> [0.403] recv DATA frame <length=1500, flags=0x00,
stream_id=13> [0.405] recv DATA frame <length=1500, flags=0x00, stream_id= 13> [0.406]接收数据帧<长
度=3644,标志=0x00,stream_id=13>[0.416]发送报头帧<长度=250,标志=0x25,stream_id=15>[0.417]接收数据
帧<长度=9635 , flags=0x00, stream_id=13> [ 0.417] recv DATA frame <length=807, flags=0x00,
stream_id=13> [ 0.419]发送 WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
(window_size_increment= 33107)
```

Machine Translated by Google

[0.419]发送WINDOW_UPDATE帧<长度=4,标志=0x00， stream_id=13> （window_size_increment=33107） [0.420]接收数据帧<长度=16384,标志=0x00,stream_id=13>[0.420]接收数据帧<长度=369,标志=0x00， stream_id=13> [0.424]接收数据帧<长度=16209,标志=0x01,stream_id=13>[0.444]接收WINDOW_UPDATE帧<长度=4,标志=0x00， stream_id=15> (window_size_increment=10420224) [0.546] recv (stream_id=15) x-frame-options: DENY [0.546] recv HEADERS frame <length=255, flags=0x04, stream_id=15> [0.546] recv DATA frame <length=1293 ,标志=0x00,stream_id=15> [0.546]接收数据帧<长度=2618,标志=0x00,stream_id=15>[0.547]接收数据帧<长度=3135,标志=0x00,stream_id=15>[0.547]发送 WINDOW_UPDATE 帧<length=4, flags=0x00, stream_id=0> (window_size_increment=34255) [0.547] recv DATA frame <length=10, flags=0x01, stream_id=15> [0.547] 发送 GOAWAY frame <length=8 , 标志=0x00, stream_id=0>

没有看到WINDOW_UPDATE帧?

如果您使用的是 Facebook 以外的示例（可能是您自己的站点），您可能会惊讶地没有看到任何WINDOW_UPDATE帧。也许您使用的站点太小，可以在发送单个WINDOW_UPDATE帧之前完整下载。

即使在 Facebook 示例中，nghttp 仅在 9 帧和 33,919 个八位字节后发送了一个WINDOW_UPDATE帧 远早于我们之前声明我们可以处理的 65,535 个限制。如果此时 nghttp 没有发送这个WINDOW_UPDATE帧，服务器会很高兴地继续发送数据更长的时间。

确切地发送WINDOW_UPDATE帧的时间（在消耗每个数据帧之后?当您接近限制时?定期？）取决于客户端。当 nghttp 消耗了一半以上的流量控制窗口 a （本例中为 32,768 个八位字节）时，它决定发送它们。这就是它在上面的 5572 DATA帧之后发送它的原因，在该帧之前，总数为 28,347 个八位字节（低于此限制），而在该帧之后，总数为 33,919 个八位字节（高于此限制）。

如果我们以 Twitter 为例，发送回 nghttp 的响应小于 32 KB（至少对于非登录请求），因此 nghttpd 不需要使用任何WINDOW_UPDATE帧，这不会使举个有趣的例子。读者可以通过使用-w和-W标志来使用 nghttp 在自己的站点上进行实验，以使用不同的初始窗口大小。 b

---

[a] 在 https://github.com/nghttp2/nghttp2/blob/master/lib/nghttp2_helper.c 中搜索 nghttp2_should_send_window_update函数https://nghttp2.org/documentation/

[b] nghttp.1.html#cmdoption-nghttp-w

Machine Translated by Google

7.2.2 设置服务器流控

Apache 允许您使用H2WindowSize指令设置流控制窗口大小:3

```
H2WindowSize 65535
```

其他服务器也可能允许设置此指令。例如,NodeJS 允许使用 initialWindowSize 设置来设置此指令， 4而Jetty servlet 引擎允许使用jetty.http2.initialStreamRecvWindow设置来设置它。5许多其他服务器（例如 nginx 和 IIS)不支持在撰写本文时授予您对该指令的任何控制权。然而,现实情况是您不太可能需要更改默认设置的指令,除非您想要对服务器进行详细控制。

7.3 流优先级接下来我们看一下

流优先级。 HTTP/2 引入了优先级的概念,允许客户端建议请求的相对重要性。浏览器下载页面后,它会请求查看该页面所需的资源。关键的、阻塞渲染的资源（想想 CSS 和任何阻塞的 JavaScript)是高优先级的,任何图像或异步JavaScript 都可以以低得多的优先级请求。服务器可以使用优先级来决定发送帧的顺序;可以首先发送更重要的帧,因此它们可以更早到达并且不会由于任何流量控制或带宽问题而被延迟。

流优先级:提示或说明?

流优先级由请求者（例如客户端)发送,但最终决定发送什么帧的是发送者（例如服务器）。因此,优先级是建议或提示,发送方完全可以忽略优先级并按照发送方认为最合适的顺序发送数据。该规范明确说明了这一事实,称“表示优先级是 . . .只是一个建议。”a

浏览器或服务器最好决定优先级吗?浏览器传统上承担了这个角色,因为它们只有有限数量的 HTTP/1.1 连接并且必须决定如何最好地使用它们,但是 HTTP/2 颠覆了这种情况并表示服务器负责。如果网站管理员将 Web 服务器调整到他或她最了解的特定站点,这种情况可能是有道理的,但如果没有这种高级调整（大多数网站所有者不太可能愿意进行）,Web 浏览器可能会比网络服务器更好地理解优先级。

我怀疑大多数 Web 服务器使用客户端提供的优先级提示来决定优先级,因此最终,客户端（例如 Web 浏览器)可能会保留

---

[3] https://httpd.apache.org/docs/2.4/mod/mod_http2.html#h2windowsize

[4] https://nodejs.org/api/http2.html#http2_settings_object https://github.com/

[5] eclipse/jetty.project/ blob/master/jetty-documentation/src/main/asciidoc/administration/http2/configuring-http2.adoc

Machine Translated by Google

决定优先级。可能有机会在服务器端覆盖这些设置（请参阅第 7.3.4 节），但大多数情况下，我希望遵循客户端优先级排序请求。

一些网络服务器可能决定根本不使用优先级，因为它的实现可能非常复杂，但我怀疑那些实现它的人会比那些没有实现它的人看到性能改进，所以选择你的网络服务器（和网络浏览器）明智地！

https://httpwg.org/specs/rfc7540.html#StreamPriority

HTTP/2 定义了两种不同的设置优先级的方法：

流依赖流加权

这些优先级可以通过HEADERS帧中的请求设置，或者可以随时通过单独的PRIORITY帧重新设置优先级。

7.3.1 流依赖

一个流可以依赖于另一个流，所以只有当依赖流不需要使用连接来发送任何东西时，它才应该被用来发送资源。图 7.3 显示了这样一个例子。

默认情况下，一切都依赖于流 0（图 7.3 中未显示），它是控制流，表示无依赖性。在此示例中，main.css 是对 index.html 的第一个依赖项，应以最高优先级发送，其次是 main.js，最后是 image.jpg。通常，首先获取 index.html，然后是依赖项，因此可能不需要像此处所示那样对 HTML 文档文件添加依赖项。

但是一个大的 index.html 可能仍在下载其他请求，所以让所有请求都依赖于它不一定是错误的。

这种依赖层次结构并不意味着依赖流会阻塞它们的父流。如果 main.css 不是立即可用于 Web 服务器并且必须从后端获取

服务器，例如，服务器可以同时发送 main.js，假设它可用。如果两个文件都不可用，则可以在服务器等待这些文件时发送 image.jpg。目标是

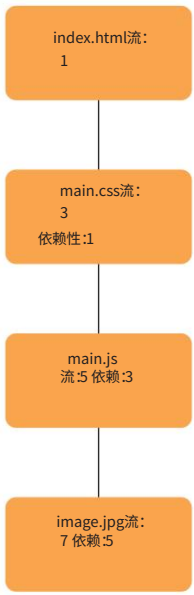


图 7.3 HTTP/2 流依赖示例

流优先级是为了最有效地使用连接而不是充当阻塞机制。

服务器可能会在获取 main.css 和 main.js 时开始发送 image.jpg，当这些文件可用于发送时，服务器可能会暂停发送 image.jpg 和

Machine Translated by Google

在取消暂停 image.jpg 并发送该文件的其余部分之前,先发送 main.css,然后发送 main.js。或者,服务器可以有一个更简单的模型,并在它启动时完成发送 image.jpg,而其他服务器在准备就绪时排队。

选择取决于服务器。

流也可以有多个依赖项,如图 7.4 所示,因为每个流都可以指定它所依赖的流。

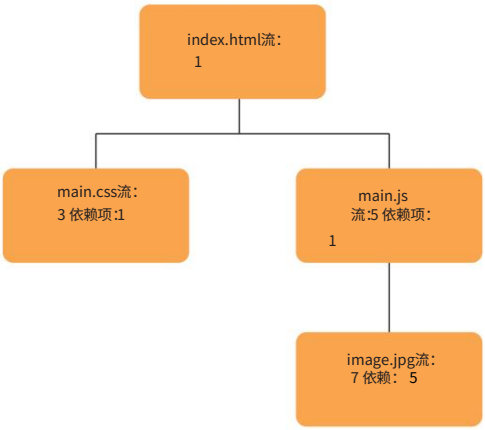


图 7.4 多个流可以依赖于同一个父流。

本例中main.css和main.js都依赖stream 1,图片依赖main.js stream。如果图像文件的优先级低于这两个关键资源,理想情况下它将依赖于 CSS 和 JS 流,如图 7.5 所示,但不支持多重依赖的概念。

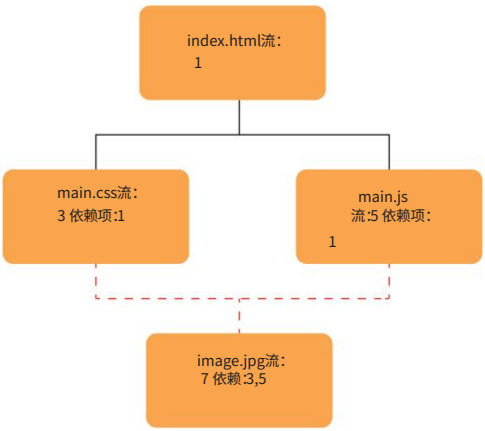


图 7.5 HTTP/2 不支持多个父依赖项。

如果支持多依赖,只有当main.css和main.js都不需要连接时才下载文件,但多依赖

Machine Translated by Google

不受 HTTP/2 依赖模型支持（尽管它们可以通过使用权重来近似）。

随着可用于服务器发送的资源变得可用,流优先级的管理可能会变得复杂。通过添加新请求或完成正在进行的请求,它可能会变得更加复杂。通常,服务器必须在请求的生命周期内多次重新评估依赖关系以获得最佳性能。 Apache 在其 HTTP/2 实施的早期就发现,不进行这种重新确定优先级会导致效率低下。[6]

流也可以作为独占依赖项添加。流应该获得对其依赖项的独占访问权,并且任何现有的依赖项都应该依赖于这个新的独占流。图 7.6 显示了将 critical.css 添加到混合中并使其依赖于流 0,没有（左)和（右）独占标志集。

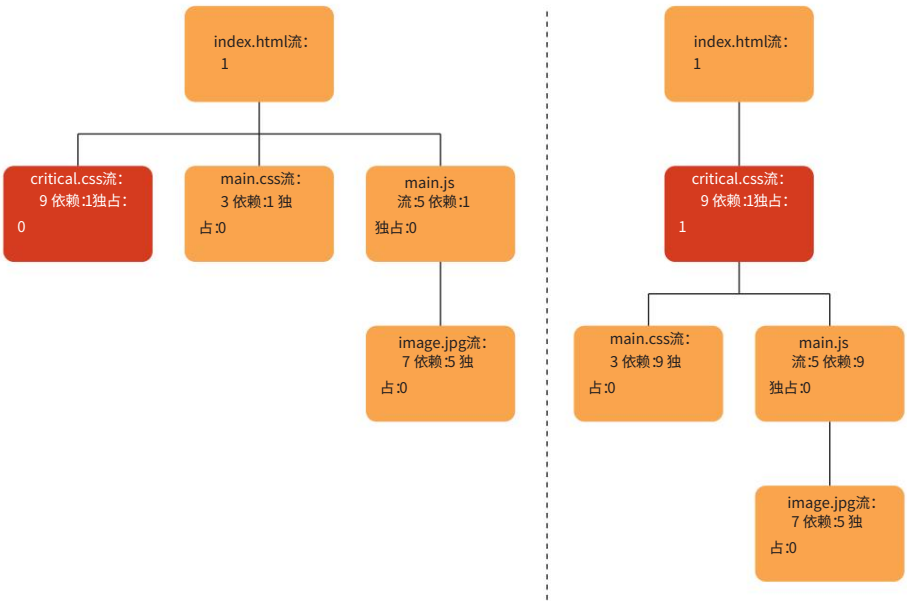


图 7.6 添加新的 critical.css 依赖项,设置和不设置独占标志

如您所见,如果没有独占标志,critical.css 与 main.css 和 main.js 处于相同的依赖级别,但是当独占标志设置为1 时,它具有优先权并将所有内容都依赖于它,这根据名称 (critical.css),很可能正是本例中所需要的。

[6] http://icing.github.io/mod_h2/nimble.html

Machine Translated by Google

7.3.2 流加权

另一个有助于定义流优先级的概念是加权,它用于确定依赖于同一父资源的两个请求的优先级。与假设相同依赖级别的资源平均加权相比,流加权允许更复杂的场景。例如,critical.css 场景可以使用权重以(几乎)类似的方式实现,如图 7.7 所示。

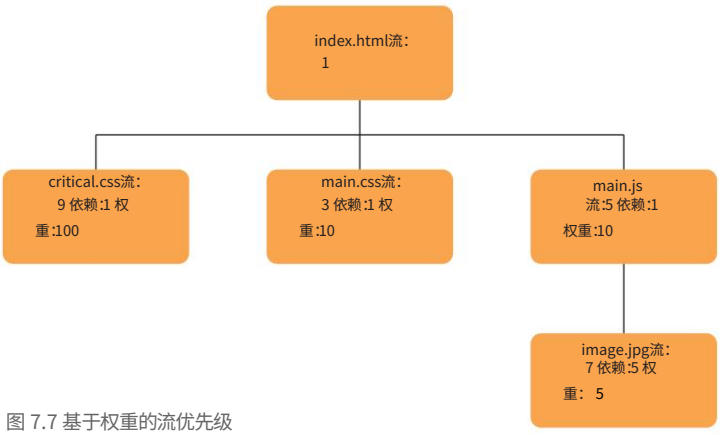


图 7.7 基于权重的流优先级

在这里,critical.css(权重 100)的资源分配应该是 main.css(权重 10)和 main.js(权重 10)的 10 倍。使用权重与使它们依赖(如排他标志)不同,但很接近。交付 critical.css 时,main.css 和 main.js 获得 50% 的资源,因为它们的权重是均匀的。

场景中未使用 image.jpg 的 5 权重。如果 main.js 在 main.css 之前完成发送(或者如果 main.js 还不能发送),image.jpg 获得 50% 的资源,因为它获得了 main.js 的份额。因此,为了防止这种情况发生并赋予 CSS 和 JS 文件比图像更高的权重,更好的依赖图可能是图 7.8 中所示的更平坦的依赖图。

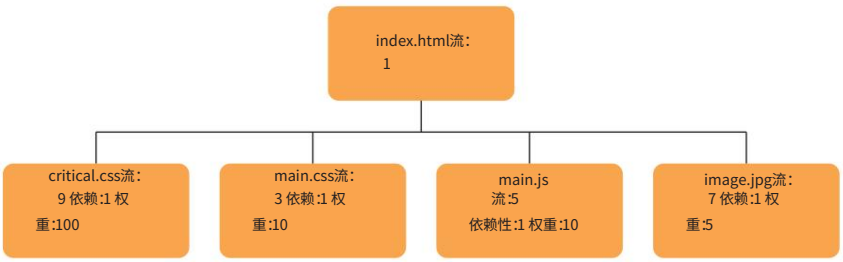


图 7.8 基于权重的依赖关系

Machine Translated by Google

为了更容易确定优先级,一些客户端使用PRIORITY框架预先设置具有适当优先级的虚拟流,并挂起它们的请求。允许虚拟PRIORITY帧的概念是在 HTTP/27 的批准后期添加的,但提供了很大的灵活性并允许轻量级优先级模型。

,

它允许依赖树,例如,如图 7.9 所示。

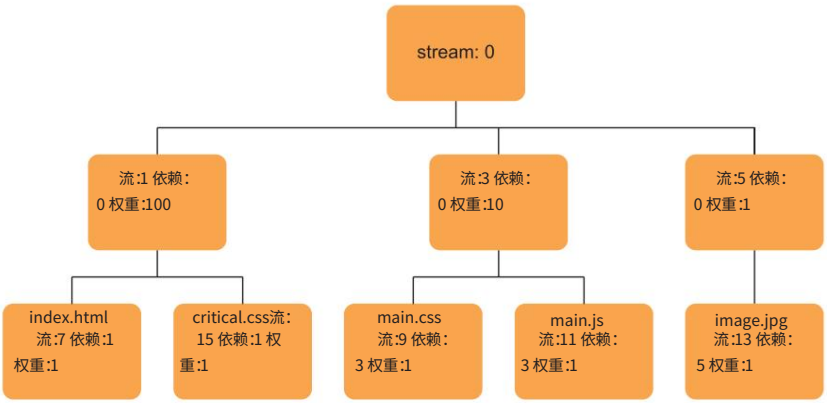


图 7.9 使用虚拟流挂起请求以适当地设置依赖关系

这些虚拟流仅用于确定优先级,从不直接发送请求。当 nghttp 出于优先级原因在连接开始时设置流 3、5、7、9 和 11 时,您会看到这种情况:

```
$ nghttp -nva https://www.facebook.com:443 [ 0.041] 已连接协商协
 议:h2 [ 0.093] recv SETTINGS frame <length=30, flags=0x00,
stream_id=0> (niv=5)

                    [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
                    [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
                    [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
                    [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
                    [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536] [0.093] recv
WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0> (window_size_increment=10420225) [0.093] 发送
                    SETTINGS frame <length=12, flags=0x00, stream_id=0 > (niv=2)

                    [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
                    [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[0.093]发送设置帧<长度=0,标志=0x01,stream_id=0>
                    ; ACK
                    (niv=0)
[ 0.093] 发送优先帧 <length=5, flags=0x00, stream_id=3>
                    (dep_stream_id=0, weight=201, exclusive=0)
```

[7] https://lists.w3.org/Archives/Public/ietf-http-wg/2014OctDec/0467.html

Machine Translated by Google


```
[0.093]发送优先级帧<长度=5,标志=0x00,stream_id=5>
                    (dep_stream_id=0, weight=101, exclusive=0)
[0.093]发送优先级帧<长度=5,标志=0x00,stream_id=7>
                    (dep_stream_id=0, weight=1, exclusive=0)
[0.093]发送优先级帧<长度=5,标志=0x00,stream_id=9>
                    (dep_stream_id=7, weight=1, exclusive=0) [0.093] 发送优先帧
<length=5, flags=0x00, stream_id=11> (dep_stream_id=3, weight=1, exclusive=0)
```

此代码导致图 7.10 中所示的依赖树,它具有高优先级流 3（具有依赖流 11）、低优先级流 7（具有依赖流 9)和中等优先级流 5。

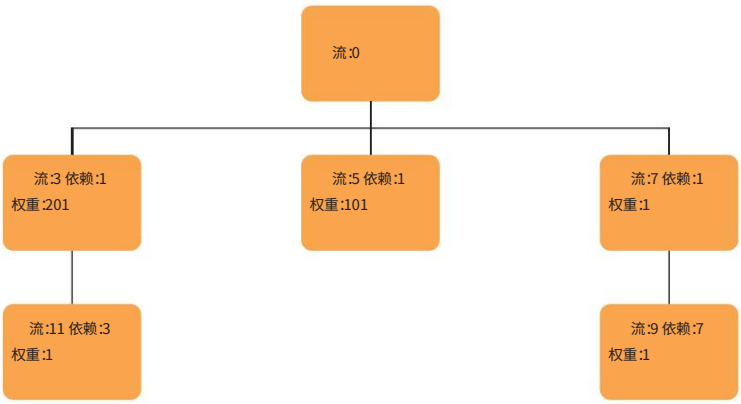


图 7.10 nghttp 流优先级

任何请求都依赖于这些流之一：

```
[0.093]发送报头帧<长度=43,标志=0x25， stream_id=13>
                    ;结束流 | END_HEADERS |优先级（padlen=0，
                    dep_stream_id=11， weight=16,exclusive=0）
                    ;打开新流 :method:
                    GET :path: / :scheme:
                    https :authority:
                    www.facebook.com accept:
                    */* accept-encoding: gzip, deflate user-agent:
                    nghttp2/1.31.0
```

此设置基于原始的 Firefox 依赖树。关键的 CSS 和 Java Script 依赖于流 3,非关键的 JavaScript 依赖于流 5,其他一切都依赖于流 11。请注意,流 7 和 9 在撰写本文时没有使用。8通过始终能够要挂起来自相同流的资源,您可以轻松地创建一个相当有效的依赖模型。

[8] https://nghttp2.org/documentation/nghttp.1.html#dependency-based-priority

Machine Translated by Google

7.3.3 为什么优先排序需要如此复杂?

为什么需要流依赖和加权?当 HTTP/2 被标准化时,这个问题争论不休,而 HTTP/2 所基于的 SPDY 最初只有基于权重的优先级。事实上,优先级排序是复杂的,并且允许依赖性和权重,或两者的混合,可以为优先级排序提供最大的灵活性。纯粹为了优先级目的而创建流的附加功能导致了更多的实施选项。

然而,不需要支持流优先级排序,客户端和服务器端的许多实现都选择不支持,我将在下一节中讨论。正如我在第 6.2.4 节中所述,高效处理 HTTP/2 流优先化的能力可能成为浏览器和服务器实现之间的另一个关键区别,尽管大多数 Web 用户和开发人员可能不了解这些技术细节。

自从 HTTP/2 标准化以来,HTTP/2 优先级在现实世界中的使用,以及尚未找到完美的优先级方案的实施这一事实（正如我们接下来将要讨论的那样)导致了更多要求简化这一点的呼声。[9]这是否会导致HTTP/2 中的任何更改或考虑用于未来版本 (HTTP/3) 的任何更改仍有待观察。

7.3.4 网络服务器和浏览器的优先级

HTTP/2 优先级是一个潜在的强大选项,它允许有效地使用单个 HTTP/2 连接。当除了不使用其中一个连接之外没有相对优先级的概念时,此选项优于六个单独的 HTTP/1.1 连接。但是,确定优先级很复杂,目前支持有限。尽管服务器端和客户端的许多实现都支持优先级排序,但很少有网站所有者能有更多的控制权。

为网络服务器设置优先级

在撰写本文时,服务器对优先级的支持是一个混合包。有些服务器通过配置选项支持它,有些服务器没有配置选项支持它,有些服务器不支持它。表 7.1 总结了流行的 HTTP/2 Web 服务器中的优先级支持。

表 7.1 流行的 HTTP/2 Web 服务器中的优先级支持

服务器（和版本）	HTTP/2 优先级支持
支持 Apache HTTPD (v2.4.35)	优先级,但只能显式配置推送优先级。a
IIS (v10.0)	不支持优先级排序。b
nginx (v1.14)	支持优先级排序,c但没有可用的配置选项。d

[9] https://lists.w3.org/Archives/Public/ietf-http-wg/2019JanMar/0073.html

Machine Translated by Google

表 7.1 流行的 HTTP/2 Web 服务器中的优先级支持（续）

服务器（和版本）	HTTP/2 优先级支持
节点 (v10)	支持优先级排序,并且可以明确设置。e
nghttpd (1.34)	完全支持优先排序。f

a https://httpd.apache.org/docs/2.4/mod/mod_http2.html#h2pushpriority https://
b forums.iis.net/t/1233780.aspx https://www.nginx.com/blog/http2- module-nginx/
c #prioritization http://nginx.org/en/docs/http/ngx_http_v2_module.html https://
d nodejs.org/api/http2.html#http2_http2stream_priority_options https://nghttp2.org/
e blog/2014/ 04/27/如何基于依赖的优先级排序/
f

大多数其他 Web 服务器很少提及 HTTP/2 优先级,这表明它可能不受支持,如果支持,当然也不可配置。在支持它的服务器中,普遍的想法似乎是允许客户端在请求中指定优先级而不是允许服务器端优先级配置。

Shimmercat 是一个相当新的 Web 服务器,它采用了一种有趣的方法,允许以初始高优先级发送图像请求,然后回拨到较低优先级。这种方法允许发送前几个字节,这样浏览器就可以知道图像的大小和尽早布置页面所需的其他元数据,并降低图像文件其余部分的优先级。

也许更多的网络服务器将允许这种类型的创新或更多的控制。但目前,大多数服务器使用客户端建议的优先级或不支持它们。

为网络浏览器设置优先级

Web 浏览器支持也有点碰运气。查找有关此主题的文档很棘手,但可以查看流优先级并推断它在做什么。为此,您可以按照第 4 章中的讨论设置 Wireshark,但这种技术只允许您拦截导出 HTTPS 密钥设置的浏览器（例如桌面上的 Chrome、Opera 和 Firefox）。更好的方法是以详细模式运行 nghttpd 服务器并查看传入的消息。您还可以将输出通过管道传输到grep以仅过滤重要消息。如果没有 Linux 终端的 Windows 用户使用 PowerShell,则可以使用findstr或select-string执行等效操作：

```
nghttpd -v 443 server.key server.crt | grep -E 优先级|路径|权重
```

然后在同一文件夹中创建一个虚拟 index.html 文件,其中包含对各种媒体类型的引用,以了解每个浏览器如何发送每种媒体类型：

```
<html>
<head>
<title>这是一个测试</title> <link rel= stylesheet
type= text/css media= all href= head_styles.css > <script src= head_script.js ></script> </head> <body>
```

Machine Translated by Google


```
<h1>这是一个测试</h1> <img
src=  image.jpg  /> <script
src=  body_script.js  /></script> </body> </html>
```

对于这个简单的测试,引用的样式表、JavaScript 或图像文件是否存在并不重要。如果这些项目不存在,测试会稍微容易一些,事实上,因为您只会看到404 HEADERS帧响应,而不是HEADERS帧和DATA帧响应,这只会增加噪音。

接下来,您连接到服务器（例如 https://localhost)并查看发送的帧。 Firefox (v62) 以类似于 nghttp 客户端的方式发送帧,这不足为奇,因为 nghttp 是基于 Firefox 实现的：

```
[id=1] [3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=3>
                (dep_stream_id=0, weight=201, exclusive=0)
[id=1] [3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=5>
                (dep_stream_id=0, weight=101, exclusive=0)
[id=1] [3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=7>
                (dep_stream_id=0, weight=1, exclusive=0)
[id=1] [3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=9>
                (dep_stream_id=7, weight=1, exclusive=0)
[id=1] [3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=11>
                (dep_stream_id=3, weight=1, exclusive=0)
[id=1] [3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=13>
                (dep_stream_id=0, weight=241, exclusive=0) [id=1] [ 3.010] recv
(stream_id=15) :path: / ;结束流 | END_HEADERS |优先级（padlen=0，
                dep_stream_id=13,weight=42， exclusive=0）

[id=1] [3.033] recv (stream_id=17) :path: /head_styles.css
                ;结束流 | END_HEADERS |优先级（padlen=0，
                dep_stream_id=3,weight=22， exclusive=0）
[id=1] [3.034] recv (stream_id=19) :path: /head_script.js ;结束流 | END_HEADERS |优先级
                （padlen=0， dep_stream_id=3,weight=22， exclusive=0）

[id=1] [3.035] recv (stream_id=21) :path: /image.jpg
                ;结束流 | END_HEADERS |优先级（padlen=0，
                dep_stream_id=11,weight=12， exclusive=0）
[id=1] [3.035] recv (stream_id=23) :path: /body_script.js ;结束流 | END_HEADERS |优先级
                （padlen=0， dep_stream_id=5,weight=22， exclusive=0）
```

输出显示 Firefox 添加了一个额外的流 13,权重为 241（超紧急流？)用于原始请求,使其比任何 CSS 请求具有更高的优先级。

Chrome (v69) 不使用前端优先级框架,如 nghttp 或 Firefox,但它在发送请求时设置请求的优先级，并添加对先前流的依赖性。
它还喜欢独占依赖,创建一个高大的依赖图：

```
[id=3] [112.082] recv (stream_id=1) :path: / ;结束流 | END_HEADERS |
                优先级（padlen=0， dep_stream_id=0,weight=256，
                exclusive=1）
```

Machine Translated by Google


```
[id=3] [112.101] recv (stream_id=3) :path: /head_styles.css ;结束流 | END_HEADERS |优先
                 级（padlen=0， dep_stream_id=0,weight=256,exclusive=1）

[id=3] [112.101] recv (stream_id=5) :path: /head_script.js ;结束流 | END_HEADERS |优先
                 级（padlen=0， dep_stream_id=3,weight=220,exclusive=1）

[id=3] [112.101] recv (stream_id=7) :path: /image.jpg
                 ;结束流 | END_HEADERS |优先级（padlen=0，
                 dep_stream_id=5,weight=147,exclusive=1）
[id=3] [112.107] recv (stream_id=9) :path: /body_script.js ;结束流 | END_HEADERS |优先
                 级（padlen=0， dep_stream_id=0,weight=183,exclusive=1）
```

这种使用独占位的好处仍在争论中。10 Chromium 团队的主要论点似乎是大多数请求在收到完整资源之前是不可用的（HTML 和渐进式 JPEG 是主要的例外），所以它通常不会 通过同时发送多个资源来稀释连接是没有意义的。

Opera (v59) 与 Chrome（是另一个基于 Chromium 的浏览器)做同样的事情，但 Safari (v12.0) 似乎根据优先级进行加权并且不使用流依赖性（与 Chrome 相反！）：

```
[id=9] [213.347] recv (stream_id=1) :path: / ;结束流 | END_HEADERS
                 |优先级（padlen=0， dep_stream_id=0,weight=255，
                 exclusive=0）
[id=9] [213.705] recv (stream_id=3) :path: /head_styles.css ;结束流 | END_HEADERS |优先
                 级（padlen=0， dep_stream_id=0,weight=24， exclusive=0）

[id=9] [213.705] recv (stream_id=5) :path: /head_script.js ;结束流 | END_HEADERS |优先
                 级（padlen=0， dep_stream_id=0,weight=24， exclusive=0）

[id=9] [213.706] recv (stream_id=7) :path: /image.jpg
                 ;结束流 | END_HEADERS |优先级（padlen=0，
                 dep_stream_id=0,weight=8， exclusive=0）
[id=9] [213.706] recv (stream_id=9) :path: /body_script.js ;结束流 | END_HEADERS |优先
                 级（padlen=0， dep_stream_id=0,weight=24， exclusive=0）
```

Edge (v41) 的实现最差，在撰写本文时选择不使用流优先级，因此每个资源的默认优先级权重为 16：

```
[id=4] [64.393] recv (stream_id=1) :path: / [id=4] [64.616] recv
(stream_id=3) :path: /head_styles.css [id=4] [64.641] recv (stream_id =5) :path: /
head_script.js [id=4] [64.642] recv (stream_id=7) :path: /image.jpg [id=4] [64.642] recv
(stream_id=9) :path: /body_script .js
```

如您所见，浏览器之间存在很大差异，这导致同一站点的性能不同。一些研究人员进行了更广泛的

[10] https://bugs.chromium.org/p/chromium/issues/detail?id=651538

Machine Translated by Google

测试浏览器之间的差异。[11]在这方面可能会有更多的研究和改进。 HTTP/2 提供了用于特定优先级的工具,但我还没有找到使用它们的最佳方法。

7.4 HTTP/2 一致性测试

现在您了解了 HTTP/2 的所有细节,您可以比较客户端和服务器端的各种实现。

7.4.1 服务器一致性测试

H2spec[12]是一个 HTTP/2 一致性测试器,它向 HTTP/2 服务器发送各种消息并检查它是否准确地遵循规范。下载适用于您的计算机类型的版本,[13]并将其指向 HTTP/2 服务器:

```
h2spec -t -S -h localhost -p 443
```

> 注意如果您使用的是不受信任的证书（例如本地主机的自签名证书）,您可能需要传入-k 选项以忽略证书错误:
>
> ```
> h2spec -t -S -h -k localhost -p 443
> ```

此代码应针对您的服务器运行多个测试,并向您显示每个测试是通过还是失败:

```
$ ./h2spec -t -S -h localhost -p 443 HTTP/2 服务器的通用测试 1. 启
动 HTTP/2

    ✓ 1:发送客户端连接前言

  2. 流和多路复用
    ✓ 1:在空闲流上发送优先级帧✓ 2:在半关闭（远程)流上发送 WINDOW_UPDATE 帧✓
    3:在半关闭（远程)流上发送优先级帧✓ 4:在半关闭（远程)流上发送 RST_STREAM 帧关闭（远程)流✓ 5:在关闭流上发送优先级
    帧



  3. 框架定义
    3.1.数据
      ✓ 1:发送一个数据帧✓ 2:发送多个数据帧✓ 3:发
      送一个带填充的数据帧


    3.2.标头
      ✓ 1:发送一个 HEADERS 帧✓ 2:发送一个带填充的
      HEADERS 帧✓ 3:发送一个带优先级的 HEADERS 帧

…ETC
```

---

[11] https://speakerdeck.com/summerwind/2-prioritization和https://www.researchgate.net/publication/324514529_HTTP2_Prioritization_and_its_Impact_on_Web_Performance https://github.com/summerwind/h2spec

[12] https://github.com/summerwind/h2spec/发布

[13]

Machine Translated by Google

我已经针对一些流行的 Web 服务器运行了该工具,结果如表 7.2 所示。

表 7.2 流行 Web 服务器的 HTTP/2 规范一致性

服务器（和版本）	测试通过
阿帕奇 (v2.4.33)	146/146 (100%)
nghttpd (v1.13.0)	145/146 (99%)
Apache 流量服务器 (v7.1.3)	140/146 (96%)
CaddyServer (v0.10.14)	137/146 (94%)
代理服务器 (v1.8.8)	136/146 (93%)
信息系统 (v10)	119/146 (82%)
AWS弹性负载均衡器	115/146 (79%)
nginx (v1.13.9)	112/146 (77%)

我在一些常见的内容分发网络的主页上做了类似的测试,假设主页运行在 CDN 基础设施上,诚然这可能不是一个有效的假设。表 7.3 显示了结果。

表 7.3 流行 CDN 的 HTTP/2 规范一致性

CDN（和站点测试）	测试通过
快速(www.fastly.com)	137/146 (94%)
谷歌(www.google.com)	135/146 (92%)
Cloudflare (www.cloudflare.com)	113/146 (77%)
MaxCDN (www.maxcdn.com)	113/146 (77%);请注意,测试 6.3.2 已挂起
阿卡迈(www.akamai.com)	107/146 (73%)

感谢 Apache 取得了唯一的满分。但是,某些实现不符合规范,这有关系吗?可以说不是,因为它们在尝试处理不应该首先发送的错误消息时经常会失败。许多流行的服务器/CDN 成功地处理了 HTTP/2 流量并且没有问题,尽管没有获得 100% 的一致性。

如果您以 nginx 的结果为例,您会看到以下内容是服务器失败的测试之一：

```
4.2.车架尺寸
    ✓1:发送一个长度为 2^14 个八位字节的 DATA 帧    2:发送一个大尺寸的 DATA 帧,超过

  SETTINGS_MAX_FRAME_SIZE -> 端点必须
        发送错误代码 FRAME_SIZE_ERROR。
```

Machine Translated by Google


```
预期:GOAWAY 帧（错误代码:FRAME_SIZE_ERROR）
                RST_STREAM 帧（错误代码:FRAME_SIZE_ERROR）
                连接关闭
  实际:WINDOW_UPDATE 帧（长度:4,标志:0x00,stream_id:1）
```

nginx 没有像它应该的那样处理大数据帧,但与此同时,没有客户端应该发送这样的帧。继续下一个错误,您会看到一些状态错误:

```
5. 流和多路复用
  5.1.流状态
      1: idle: 发送一个DATA帧
       -> 端点必须将此视为 PROTOCOL_ERROR 类型的连接错误。

          预期:GOAWAY 帧（错误代码:PROTOCOL_ERROR）
                连接关闭
            实际:超时
     2: 空闲:发送 RST_STREAM 帧 -> 端点必须将此视为
       PROTOCOL_ERROR 类型的连接错误。

          预期:GOAWAY 帧（错误代码:PROTOCOL_ERROR）
                连接关闭
```

同样,当流处于空闲状态时,nginx 无法正确处理错误发送的帧,但同样,客户端不应发送这些帧。大多数其他错误也随之而来。

如果您正在编写 HTTP/2 服务器,h2 规范工具可用于检查您的服务器是否正确实现了规范,但现实情况是许多主要 Web 服务器的实现并不完美。 Web 在技术方面一直是一个宽容的地方,并且（与许多编程语言不同)经常忽略轻微的错误。尽管如此,这些错误可能会在以后导致更多意外错误,因此了解您的服务器的行为方式很有趣。当我在 Twitter 上发布上述统计数据时,[14]一些服务器实施注意到了这一点,并试图提高它们的合规性。

7.4.2 客户端一致性测试

该工具的客户端等价物（例如用于测试浏览器)确实存在,[15]尽管未提供构建版本,因此该工具必须从源代码编译。我把这个任务留给读者作为练习。

概括

HTTP/2 有几个很少讨论的高级概念,因为很多
人们专注于更高层次的概念。
本章中的大部分底层细节不受服务器管理员或网站开发人员的控制。

---

[14] https://twitter.com/tunetheweb/status/988196156697169920
[15] https://github.com/summerwind/h2spec/pull/74

Machine Translated by Google

HTTP/2 具有流状态和显示有效转换的状态图
状态之间。

HTTP/2 允许在流级别进行细粒度的流量控制,而不是让 TCP 在连接级别进行管理（如 HTTP/1.1 所做的那样）。 HTTP/2 引入了流优先级,它允许客户端建议服务器在返回请求时使用的优先级。 HTTP/2 流优先级系统基于依赖性和权重,可以使用其中之一（或两者）。

不同的浏览器和服务器使用不同的流优先级。 许多 HTTP/2 实现并不完全符合规范。

Machine Translated by Google

8 HPACK 头压缩

本章涵盖数据压缩的背景
为什么 HTTP/2 需要自己的 HTTP 标头
压缩技术

HPACK 压缩格式解压缩 HPACK 编
码的标头客户端和服务器实现中的 HPACK

下一个主题是头压缩。 HTTP/1 一直允许压缩 HTTP 主体,但只有在 HTTP/2 之后才有可能压缩 HTTP 标头。

8.1 为什么需要头部压缩?

确实,一般来说,HTTP 标头与 HTTP 正文相比相对较小,但它们仍然很冗长且重复。来自 Chrome 的典型 HTTP/2 GET请求如下所示:

```
:authority: www.example.com :method:
GET :path: /url
```

Machine Translated by Google

:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9, image/webp,image/apng,*/
*;q=0.8 accept-encoding: gzip, deflate, br accept-语言:en-GB,en-US;q=0.9,en;q=0.8
升级不安全请求:1 用户代理:Mozilla/5.0（Macintosh;Intel Mac OS X 10_13_4)AppleWebKit/537 .36
（KHTML,像壁虎)Chrome/66.0.3359.139 Safari/537.36

对于发送到此服务器的下一个请求,只有以粗体突出显示的部分可能会发生变化:请求的方法(GET)和路径(/url)。在这段代码的403个字符中,只有7个字符下次不会重复。事实上,对于大多数网页请求,即使是该方法也可能设置为GET ,尽管 Web 服务可能使用其他方法。因此,只有:path或 URL 可能会发生变化,因此每次 Chrome 请求的 399 个字符都是重复的 这是一种巨大的浪费。

由于其中一些标头很长,例如前面代码中显示的accept和user-agent标头,并且cookie标头可能会变得更长,因此问题变得更加严重。以下是 Twitter 的请求标头（混淆了 cookie 值）：

:authority: twitter.com :method:
GET :path: / :scheme: https accept:
text/html,application/
xhtml+xml,application/xml;q=0.9,
image/webp,image/apng,*/*; q=0.8 接受编码:gzip、deflate、br 接受语言:en-GB、en-US;q=0.9、en;
q=0.8 cookie：_ga=GA1.2.123432087.1234567890； eu_cn=1； dnt=1； kdt=rmnAfbecvko4123
4oRYSzztq7n12345abcdABCD12； remember_checked_on=1; personalization_id= v1_k
0123451/EKaVeysDnuhKg== ; guest_id=v1%3A152383314680123456； ads_prefs= HBES
AAA= ; twid= u=3374717733 ; auth_token=12791876dfc0e57eae12345897b7940f55ac 7dfd； tfw_exp=1;
csrf_same_site_set=1; csrf_same_site=1; lang=en; _twitter_
sess=BAh7CSIKZ12345678zonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNo%250ASGFza
HsABjoKQHVzZWR7ADoPY3JlYXRlZF9hdGwrCPdpPx12345HaWQiJWY4%250AZGUwOGM3ZjRiYzJ
mYjRiAbCdEfGwNjIyZTk1Ogxjc3JmX2lkIiVkYjg3%2501234kZTVkMDdlMTAxMGI2YTgyZDFhN TA0MmZiNQ%253D%253D--
fd52ba1537f8fb9bf35dbd6080a6cd413edc6cd2; ct0=713653a0 6266b507960945523226bcc4；
_gid=GA1.2.1893258168.1525002762； external_referer=1234567890w%3D|0|
S381234567896Dak8Eqj76tqsc12345Lq4vYdCl5zxIvK6Q123vRkA%3 D%3D； app_shell_visited=1 referer: https://
twitter.com/upgrade-insecure-requests: 1 user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/
537 .36 (KHTML, like Gecko) Chrome/66.0 .3359.139 野生动物园/537.36

这段代码现在是 1,278 个字符,而且,很可能只有:path伪标头会随着下一个请求而改变,因此每个请求几乎浪费了 1,277 个字符。反应也好不到哪里去。图 8.1 显示了对 Twitter 的响应。

Machine Translated by Google

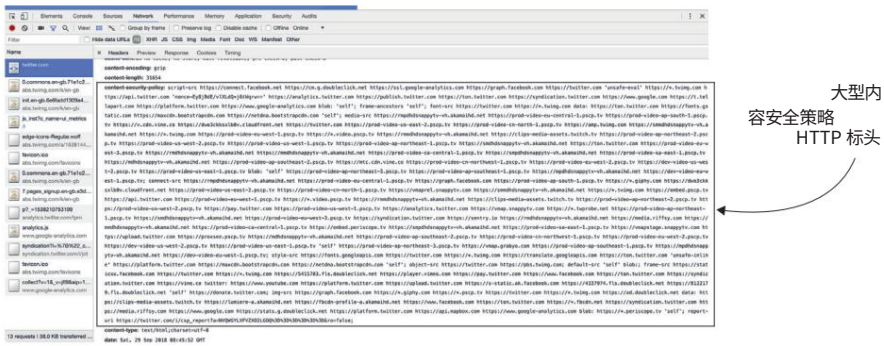


图 8.1 Twitter 响应头

这个响应是一个非常荒谬的 5,804 个字符,主要由一个详细的（和大的)内容安全策略标头组成,这是一种安全功能,允许网站告诉浏览器允许在该网站上加载什么类型的内容。 1个

计算机科学家厌恶重复,所有这些重复是 HTTP 标头压缩成为 HTTP/2 关键部分的原因之一,也是它从一开始就内置于其前身 (SPDY) 中的原因之一。压缩和解压缩数据需要时间和处理能力,但与发送网络请求所需的时间相比,这些时间相对较少,因此在将数据发送到网络之前压缩数据几乎总是值得的。此外,HTTPS 需要加密,这在计算上比压缩更昂贵。最好先压缩,然后加密较小的数据量。

8.2 压缩如何工作要理解本节其余部分关于

HTTP 压缩的内容,您需要了解一些数据压缩的背景知识。这个主题相当高级,我有意避免复杂的数学,但是本节应该给你足够的细节来理解 HTTP/2 如何以及为什么以它的方式实现标头压缩。

有些压缩是有损压缩;一些细节可以丢弃,因为它不需要。这种类型的压缩通常用于媒体:音乐文件、图像和视频可以在不丢失数据的整体含义的情况下进行大量压缩。如果压缩太多,就会丢失细节,例如,无法再放大图像。有损压缩是减小尺寸和保持质量之间的谨慎平衡。

HTTP 标头是重要的数据,即使它们重复了很多次,所以有损压缩不是一种选择,尽管它通常会提供更好的压缩。无损压缩通过删除重复数据来工作,这些数据可以在以后需要解压缩数据时轻松添加回来。您可以通过三种方式执行此操作:

---

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

Machine Translated by Google

查找表更高效的
编码技术回溯压缩

我将在以下部分中描述这些方法中的每一种。

8.2.1 查找表

第一种方法涉及获取长的、重复的数据位并用引用替换它们。解压缩涉及用查找表中的原始文本替换引用。此过程可以是动态的，但特别适用于以相同方式构建的数据。看一个简单的GET请求：

```
:authority: www.example.com :method:
GET :path: /url :scheme: https
```

这个请求是 64 个字符。如果你有这样的查找表

```
$1=:authority:
$2=:method: $3=:path:
$4=:scheme:
```

你可以像这样编码这个文本：

```
$1 www.example.com $2 获
取 $3 /url $4 https
```

这个请求是 39 个字符　改进了 40%!然而，查找表可能需要包含在压缩版本中，具体取决于它是格式已知的标准查找表还是由特定文本生成的动态查找表。如果包含表格，整体大小可能与原文大小相同或更大，这会破坏这一点。仅当值经常重复时，查找表才会对您有益。

在这个简单的示例中，查找是常用的 HTTP 标头名称，因此可以为它们使用预先商定的静态查找表，并且不需要每次都发送。该表可以由一个动态查找表来补充，以便在该预先商定的列表之上使用额外的值。例如，域 www.example.com 可能会在后续请求中重复使用，因此可以将其添加到动态列表中以供后续引用。

8.2.2 更高效的编码技术

有几种技术属于这一类，但它们都认识到这样一个事实，即如果使用更具体的方式来表示数据，则可以用更小的尺寸来表示被压缩的数据。

Machine Translated by Google

例如,基于像素的图像可以基于 1 位像素范围（黑色和白色）。如果您的图片只有红色和黄色,您可以使用 8 位调色板并仅使用其中两种颜色。或者,您可以使用 1 位调色板,但在开头声明 0 是黄色，1 是红色。同样,对于文本,您可以使用 ASCII（7 位）、UTF-8（8 位用于 ASCII 字符,16 位用于常用的西方字符,24 位用于大多数其他常用字符,最多 32 位用于其他字符）, 或 UTF-16（大多数字符为 16 位,其他字符最多为 32 位）。如果你用英语写作,UTF-8 最有意义,但如果你用非西方语言写作，UTF-16 可能更有意义,因为 UTF-8 通常对这些语言使用 24 位,与UTF-16 中的 16 位。选择适当的格式可以提高编码效率。

为了节省更多,请查看可变长度编码。大多数编码技术都涉及固定大小的编码。例如,ASCII 使用 7 位字符,如表 8.1 所示。

表 8.1 ASCII 码的子集

二进制	特点
01000001	一个
01000010	乙
01000011	C
01000100	丁
01000101	乙
01000110	F
01000111	G
等等	

这种安排很好,也很简单,但效率不高,因为无论使用频率如何,每个字母都会占用相同的空间。在英语中,E 是最常见的字母,其次是 T 和 A,一直到最不常见的字母（分别是 X、J、Q 和 Z）。鉴于它们不会被平等使用,为什么要平等对待它们呢?

与其对所有 ASCII 字符都使用 7 位,不如对英文文本使用变长字符更有效。该技术将使用小于 7 位的二进制值表示常用字符,使用大于 7 位的二进制值表示不常用的字母。 Unicode（UTF-8 和 UTF-16)在某种程度上通过不同的字符块（1-4 个八位字节)使用此方法,具体取决于字符的常用程度。主要的困难是识别字符之间的边界（因为在这种格式中它们不再都是 7 位长）。

霍夫曼编码将可变长度编码提升到一个更极端的水平。它的工作原理是根据每个值的使用频率为每个值分配一个唯一的代码，

Machine Translated by Google

并且它确保没有代码是另一个代码的前缀。唯一代码是如何计算的超出了本书的范围,但它可能会导致像表 8.2 这样的表格（基于基于英语字母频率分布的霍夫曼代码）。

表 8.2 霍夫曼编码的 ASCII 码子集

二进制	霍夫曼编码	特点
01000001	1111	一个
01000010	101000	乙
01000011	01010	C
01000100	11011	丁
01000101	100	乙
01000110	01011	F
01000111	00001	G

在霍夫曼编码中,没有代码被完全表示为另一个代码的开始。代码0101不用于表示字母,因为该代码可能会被混淆为该字母、C 的开头或 F 的开头。通过仔细选择编码字符串,可以可靠地解码数据。只要你从文本的开头开始,你就可以通过阅读直到匹配一个字母来解码它;然后你开始下一个字母并继续直到整个文本未编码。

以这些代码为例,如果你想对单词face 进行编码,你可以使用常规 ASCII（4 x 7 位 = 28 位)或霍夫曼编码（5 + 4 + 5 + 3 = 17 位）。
即使在这个简单的例子中,霍夫曼编码也更小。使用更长的文本,收益将是可观的。

霍夫曼编码压缩是查找表的扩展。与本章前面讨论的更常规的查找表一样,可以根据数据可能与之相似的已知结构（例如英语文本)预先定义霍夫曼表;它们可以根据被加密的数据动态生成;或者它们可以是两者的组合。

8.2.3 回溯压缩

回溯压缩涉及根据当前放置引用重复的文本。这种类型的压缩在 HTML 文本中表现得特别好：

```
<html>
<body>
<h1>这是一个标题</h1> <p>这是一些文字
</p> </body> </html>
```

Machine Translated by Google

该文本可以压缩如下：

```
<html>
<body>
<h1>这是一个标题</(-20,3) <p>(-24,6)一些文字</
(-19,2) </ (-58,4) </(-73 ,4)
```

每个重复的文本位都被一个引用所取代，该引用说明解压器可以找到多远的重复文本以及要提取多少。引用(-20,3)表示返回 20 个字符并获取接下来的三个字符，这应该替换此引用。如您所见，此过程适用于 HTML 文本，其中结束标记重复开始标记（尽管此示例中的标记很少，查找表可能更适合 HTML）。您还可以对 HTTP 标头使用这种类型的压缩，例如accept标头：

```
接受： text/html,application/xhtml+xml,application/xml;q=0.9, image/webp,image/apng,*/*;q=0.8
```

如您所见， html在此标头中重复了两次（应用程序、 xml和图像也是如此），因此可以使用回溯函数对其进行编码。计算机不关心只查找整个单词；我这样做只是为了让解释更清楚。例如， ml用于html和xml，因此使用它可能更好。

8.3 HTTP body 压缩

HTTP body 压缩一般用于文本数据。媒体通常根据特定格式进行预压缩，并且通常被排除在压缩之外。

例如，JPEG 是一种特定的图像压缩格式，网络服务器不应进一步压缩这些图像；图像不会压缩更多（甚至可能最终变得更大），因此会浪费处理。文本适用于所有上述压缩技术。 Web 服务器和浏览器使用的技术（deflate、gzip 和 brotli）非常相似；它们是基于 Deflate 的算法的变体，并使用多种技术的组合来实现良好的压缩率。发出请求时，浏览器通过使用accept-encoding HTTP 标头通知服务器它支持的压缩算法：

```
接受编码：gzip、deflate、br
```

服务器选择其中一种算法并压缩标头，并在响应中通知浏览器它使用了哪种算法来压缩资源：

```
内容编码：gzip
```

该技术允许引入新的压缩算法（例如 brotli），并且只有在客户端和服务器都支持时才会使用这些算法。

Machine Translated by Google

基于 Deflate 的压缩有一个主要缺陷:它已被证明是不安全的。问题是您可以使用长度来猜测内容,特别是如果您可以影响其中的某些内容。尽管 HTTP 正文可以包含一些敏感数据（例如您的姓名或帐号是否显示在页面上）,但大多数安全问题都与 HTTP 标头有关,因为它们包含用于提供身份验证的 cookie 和其他令牌。假设您有以下请求:

```
:authority: www.example.com :method:
GET :path: /secretpage :scheme: https

cookie:令牌=秘密
```

如果您能获得该令牌值（秘密）,您就能够冒充该用户。当然,您不能那样做,因为消息是加密的 理想情况下,cookie 被标记为HttpOnly,将其注入页面。[29] 所以它不能被任何 JavaScript 看到,即使你可以

但是,如果您可以访问该页面,则可以发送以下内容
使用略有不同的 URL 请求并测量发送的消息的长度:

```
https://www.example.com/secretpage?testtoken=a https://www.example.com/
secretpage?testtoken=b https://www.example.com/secretpage?testtoken=c …等等。
```

由于基于 Deflate 的压缩技术通过识别和替换重复模式来工作,您最终可能会注意到一个测试(testtoken=s)比其他测试短,因为它重复了真实 cookie 的第一部分(token=secret)。

现在你知道令牌的第一个字母了!您可以重复此过程,直到获得完整的令牌:

```
https://www.example.com/secretpage?testtoken=sa https://www.example.com/
secretpage?testtoken=sb https://www.example.com/secretpage?testtoken=sc https://
www .example.com/secretpage?testtoken=sd https://www.example.com/secretpage?
testtoken=se – 这个更短！ https://www.example.com/secretpage?testtoken=sea
https://www.example.com/secretpage?testtoken=seb https://www.example.com/secretpage?testtoken=sec – 这个更
短！ https://www.example.com/secretpage?testtoken=seca https://www.example.com/secretpage?testtoken=secb
https://www.example.com/secretpage?testtoken=secc …等等。
```

[29] https://www.owasp.org/index.php/HttpOnly

Machine Translated by Google

这个过程可能听起来很长,但它很容易编写脚本,并且是一种真正实用的攻击,称为 CRIME（压缩比信息泄漏变得容易）。3该攻击针对 SPDY 进行了演示,它使用 gzip 进行 HTTP 标头压缩。

8.4 HTTP/2 的 HPACK 头压缩

由于 CRIME 表现出的不安全性,HTTP/2 需要使用一种不易受到此类攻击的不同压缩方法。 HTTP 工作组创建了一个名为 HPACK（不是首字母缩写词)的新规范,它基于查找表和霍夫曼编码,但（至关重要的)不是基于回溯的压缩。

HPACK4是一个独立于 HTTP/2 的规范。曾一度讨论过合并这两个规范,但最终,工作组决定将它们分开。 HTTP/2 规范对 HPACK 的详细信息进行了阐述,将大部分细节推迟到单独的 HPACK 规范,5但它确实声明标头压缩是 HTTP/2 的一部分并且它是有状态的（更多信息在第 8.4.2 节）。

关于 HPACK 的一个有趣事实是,与许多 HTTP 规范不同,它不是灵活或设计用于扩展。事实上,HPACK 规范明确表示:6

> HPACK 格式有意简单且不灵活。这两个特性都降低了由于实施错误而导致的互操作性或安全问题的风险。没有定义可扩展性机制;只有通过定义一个完整的替换才可能对格式进行更改。

尽管我不确定我是否同意将 HPACK 称为简单,但我同意它对于 Internet 规范来说是异常严格的,它（如引用所解释的那样)是出于安全原因而完成的。最终,毫无疑问将引入新版本的 HPACK（可能是作为 QUIC 的一部分的 QPACK;参见第 9 章）。这个版本将如何实现需要定义（可能是关于连接建立的新设置）,但目前,HPACK 的定义相当严格。

---

[3] https://blog.qualys.com/ssllabs/2012/09/14/crime-information-leakage-attack-against-ssltls
[4] https://httpwg.org/specs/rfc7541.html https://httpwg.org/ specs/rfc7540.html#HeaderBlock
[5] https://httpwg.org/specs/rfc7541.html#rfc.section.1
[6]

Machine Translated by Google

8.4.1 HPACK静态表

HPACK 有一个包含 61 个常见 HTTP 标头名称（在某些情况下是值)的静态表，其中的一部分显示在表 8.3 中。如需完整表格，请参阅 HPACK 规范。 7

表 8.3 部分 HPACK 静态表

指数	标头名称	标头值
1个	:权威	
2个	:方法	得到
3个	:方法	邮政
4个	:小路	/
5个	:小路	/index.html
6个	:方案	网址
7	:方案	https
8个	:地位	200
9	:地位	204
10	:地位	206
11	:地位	304
12	:地位	400
13	:地位	404
14	:地位	500
15	接受字符集	
16	接受编码	gzip,放气
17	接受语言	
18	接受范围	
19	接受	
.....	.....	.....
60	通过	
61	www-认证	

该表用于请求和响应，并允许 HTTP 消息有效地压缩常用的名称，以及一些常用的名称和值对。结果，标题

:方法:获取

7 https://httpwg.org/specs/rfc7541.html#static.table.definition

Machine Translated by Google

可以压缩为对索引 2 的引用。

再举个例子,标题

```
:方法:删除
```

表中不存在,但可以使用标头名称和DELETE 的编码值对索引 2 的引用进行压缩。也就是说,即使是名称/值对表条目（例如:method: GET)也可以仅用于提供名称部分（例如:method）。然而,事实并非如此。没有工具可以查找与另一个标头关联的值。例如， header1: GET标头不能使用索引 2 中的GET值。

将此:method: DELETE标头编码为索引 3 和DELETE的编码值而不是索引 2 同样正确。这两种方法都引用:method标头名称,因此两者都是有效的。我将在 8.6 节回到这个主题。

8.4.2 HPACK动态表

除了静态表之外,HPACK 还有一个连接级动态表,从位置 62（静态表之后)开始,最大表大小由SETTINGS框架中的SETTINGS_HEADER_TABLE_SIZE值定义。如果未明确定义,默认值为 4,096 个八位字节。当达到最大表大小时,最旧的条目将被逐出。为了使这个过程更容易,每个条目在写入表时都会递增。如果请求包含两个自定义标头

```
标头 1:值 1
标头 2:值 2
```

Header1最初将被赋予表条目 62。当看到Header2时， Header1将被移动到表条目 63,并且Header2将被添加为表条目 62。也就是说,表头的表条目位置不是静态的;随着新标头添加到此请求和未来请求中的表中,它会不断增加。为此,必须接收HEADER和CONTINUATION帧以维护动态表的完整性。 TCP 保证此顺序,因此在 HTTP/2 中,动态表对于每个 TCP 连接都是唯一的。

这个过程很复杂,最好用一个真实的例子来说明,我在第 8.5 节中提供了这个例子。首先,您需要更多地了解如何在静态和动态表中引用这些标头。

8.4.3 HPACK 报头类型

可以设置标题是否添加到动态表中。有四种类型的 HPACK 标头,将在以下部分中进行描述。

文字头字段表示

文字头字段表示（以1 开头)是从表（静态或动态)中直接查找的,因此当头名称和值都已经存在时使用它

Machine Translated by Google

存在于表中。此标头由填充到最少 7 位的表索引值组成。图 8.2 显示了格式。

0	1个	2个	3个	4个	5个	6个	7
[97]	索引 (7+)						

图 8.2 文字头字段表示格式

对于需要 7 位或更多位的更大数字,存在一些额外的逻辑。8我在这里不介绍该逻辑,但足以说明它涉及用 1 填充所有这 7 位并使用额外的八位字节来指定较大的指标值。

例如,要对:method: GET进行编码,请使用索引 2（二进制10,或用前导零填充时的000 0010 ）。当您为文字头字段压缩添加前导 1 时,您最终得到1000 0010,或十六进制的82 ,如果您使用 Wireshark 查看流量,您会看到这一点（图 8.3）。 8.5 节解释了如何在 Wireshark 中查看这些帧,因此以这些屏幕截图为例。

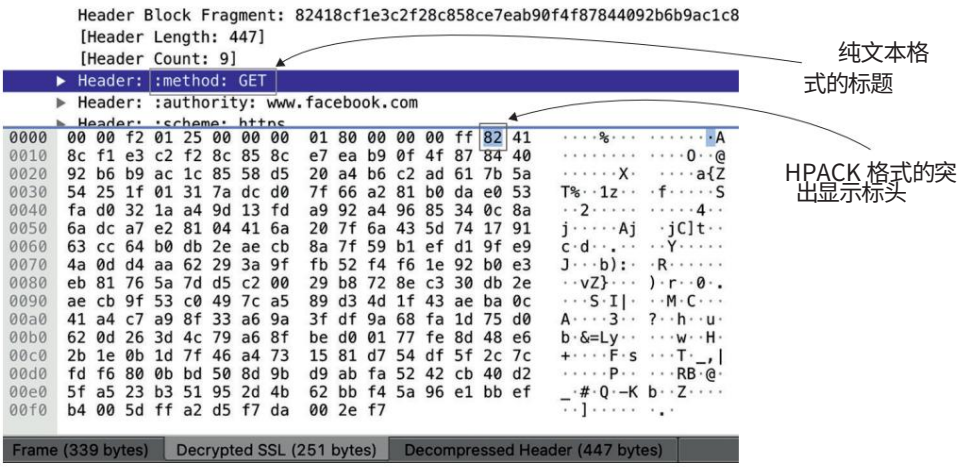


图 8.3 :method: GET header的 HPACK 压缩

具有增量索引的文字标题字段

当标头值在表中不可用但应添加到动态表以供以后使用时,使用具有增量索引类型（以01 开头)的文字标头字段。

此类型包含标头名称（可能是表中已存在的标头名称的索引引用,或者表中不存在的实际标头名称)和标头值。

---

[97] https://httpwg.org/specs/rfc7541.html#integer.representation

Machine Translated by Google

如果使用索引标头名称（标头名称已存在于表中），则01之后的位（填充到最少 6 位)定义索引值,然后是标头值本身,如图 8.4 所示。

0	1个	2个	3个	4个	5个	6个	7
0	1个	索引 (6+)					
H	值长度 (7+)						
值字符串（长度八位字节）							

图 8.4 增量索引格式 1 的文字头字段

标头值字符串可以是霍夫曼编码的,也可以不是霍夫曼编码的（取决于此过程是否会使它更短），如果使用霍夫曼编码,则 1 位 H 值设置为1 ,如果是ASCII 值,则设置为0 。理想情况下,应使用最短的编码,因此一些标头可能是霍夫曼编码的,而其他标头可能是纯 ASCII 文本。

图 8.5 显示了一个真实世界的例子。 : authority:标头的索引为1,因此它应编码为01000001,或十六进制的41 ,后跟霍夫曼编码值（我将在 8.4.4 节中讨论）。

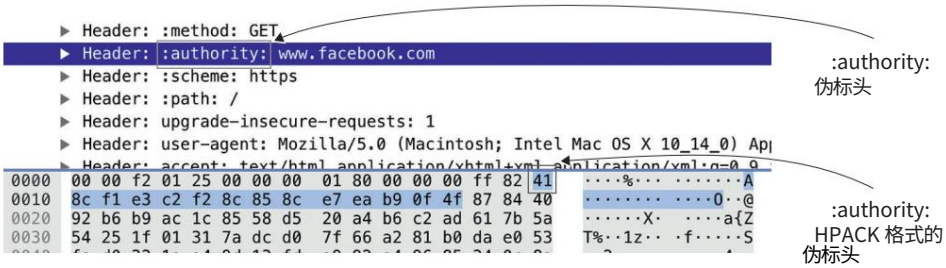


图 8.5 :authority: pseudoheader 新值的 HPACK 编码

否则,对于不在查找表中的新头部名称,将初始01之后的 6 位设置为0,然后将头部名称和值作为长度/值对给出,如图 8.6 所示。

0	1个	2个	3个	4个	5个	6个	7
0	1个	0					
H	姓名长度 (7+)						
名称字符串（长度八位字节）							
H	数值长度 (7+)						
值字符串（长度八位字节）							

图 8.6 增量索引格式 2 的文字头域

Machine Translated by Google

因此,前 8 位是01000000或40。图 8.7 显示了一个以40 开头的真实示例。

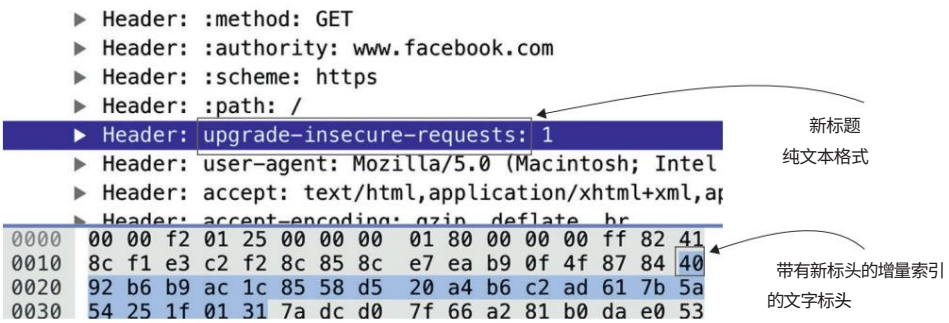


图 8.7 要索引的新 HPACK 标头

没有索引的文字标题字段

不带索引类型的文字头字段（以0000 开头)用于每次请求中可能会发生变化的项目,这会使添加到动态表中造成浪费（例如路径）。此标头类型包含标头名称（可能是对表中字段名称的索引引用或实际字段名称）,但标头名称和值不会保存为动态表中的条目。图 8.8 和 8.9 显示了两个 for mats（取决于标题名称是用表索引引用还是完整给出）。

0	1个	2个	3个	4个	5个	6个	7
0	0	0	0	索引 (4+)			
H	数值长度 (7+)						
值字符串（长度八位字节）							

图 8.8 没有索引格式 1 的文字头字段

0	1个	2个	3个	4个	5个	6个	7
0	0	0	0	0			
H	姓名长度 (7+)						
名称字符串（长度八位字节）							
H	数值长度 (7+)						
值字符串（长度八位字节）							

图 8.9 没有索引格式 2 的文字头域

Machine Translated by Google


```
▶ Header: :method: GET
▶ Header: :authority: facebook.com
▶ Header: :scheme: https
▶ Header: :path: /security/hsts-pixel.gif
▶ Header: user-agent: Mozilla/5.0 (Macintosh; Intel
▶ Header: accept: image/webp,image/apng,image/*,*/*
▶ Header: referer: https://www.facebook.com/

0000  00 00 62 01 25 00 00 00  03 80 00 00 00 92 82 41
0010  89 94 64 2c 67 3f 55 c8  7a 7f 87 00 84 b9 58 d3
0020  3f 91 61 05 25 b6 19 3e  98 9d 09 42 d5 9b c9 68
0030  5e 63 4b c2 53 9e 35 23  98 ac 78 2c 75 fd 1a 91
0040  cc 56 07 5d 53 7d 1a 91  cc 56 3e 7e be 58 f9 fb
0050  ed 00 17 7b 73 93 9d 29  ad 17 18 63 c7 8f 0b ca
0060  32 16 33 9f aa e4 3d 2c  7f c2 c1
```

纯文本格式的标题

没有索引的文字标题

图 8.10 不带索引头的 HPACK 示例

图 8.10 显示了一个示例。由于编码标头以十六进制形式的00 （或二进制形式的0000 0000)开头,因此您知道此标头是第二种类型,而不是从表中引用标头名称（:路径） 。

: path名称被霍夫曼编码为84 b9 58 d3 3f,标头值(/security/hsts-pixel.gif)被霍夫曼编码为91 61 05…等,正如我在第 8.4.4 节中讨论的那样,给出总压缩头为00 84 b9 58 d3 3f 91 61 05等等。

这种编码似乎有点奇怪和浪费,因为:path标头已经存在于表中（在位置 4 和 5）。标头可以使用格式 1 进行编码,并且会缩短 5 个八位字节：

格式 1（索引 5）： 05 91 61 05等等格式 2（无索引）： 00 84 b9 58 d3 3f 91 61 05等等

如果你用 Firefox 尝试同样的事情,它使用索引 5,但 Chrome 似乎更喜欢发送标头名称,原因不明。这个例子告诉你客户端可能不会按照你一直期望的方式编码！

字面标题字段从未索引

文字头字段从不索引类型（以0001 开头)类似于前面的值,除了该值不能在任何后续重新编码中添加到动态表中（例如当服务器充当两个代理之间的代理时） HTTP/2 实现）。此标头类型用于您不希望在共享 HTTP 标头索引中实现的敏感信息（例如用户名、密码或两者）。标头是关于如何处理重新编码以及在传输标头时现有编码的指令。

Machine Translated by Google

cookie 应该存储在 HPACK 表中吗?

Cookie 是敏感数据,似乎正是最后一种类型的设计目的。缺点是对后续请求的 cookie 的压缩减少。

Cookie 可能很大且重复,因此理想情况下,它们应该被压缩。

与部分回溯压缩不同,使用 HPACK 需要在看到效果之前猜测整个 cookie（也许请求大小更小）。一些实现（例如 Firefox 和 nghttp)a仅使用带有小 cookie（小于 20 字节)的 Never Index 类型,理论上较大的 cookie 更难猜测,因此压缩收益是值得的。对于较大的 cookie,这些实现会索引该值,以便后续请求可以引用它。

Chrome 似乎不管 cookie 的长度都使用 Index 类型,所以它似乎没有使用这种 Never Index 压缩类型。

[a] https://github.com/nghttp2/nghttp2/blob/master/lib/nghttp2_hd.c

如果发送方选择使用这种类型,则设置类似于前面的“无索引”格式（图 8.11 和 8.12）,具体取决于标头是使用当前表的索引引用还是完整给出。

0	1个	2个	3个	4个	5个	6个	7
0	0	0	1个	索引 (4+)			
H	数值长度 (7+)						
值字符串（长度八位字节）							

图 8.11 没有索引格式 1 的文字头字段

0	1个	2个	3个	4个	5个	6个	7
0	0	0	1个	0			
H	姓名长度 (7+)						
名称字符串（长度八位字节）							
H	数值长度 (7+)						
值字符串（长度八位字节）							

图 8.12 没有索引格式 2 的文字头字段

Machine Translated by Google

8.4.4 霍夫曼编码表

霍夫曼编码取决于定义用于文本中每个字符的代码表。对于 HPACK,此表在规范中定义,因此客户端和服务器都知道用于编码和解码标头名称和值的值。表 8.4 显示

HPACK 霍夫曼编码的一部分。如需完整表格,请参阅 HPACK 规范。 9

表 8.4 部分 HPACK 哈夫曼编码值

象征	ASCII码	霍夫曼码（二进制）	长度（位）
	(32)	|010100	[6]
!	(33)	|11111110|00	[10]
	(34)	|11111110|01	[10]
#	(35)	|11111111|1010	[12]
$	(36)	|11111111|11001	[13]
.....	.....	.....	.....
0	(48)	|00000	[5]
1	(49)	|00001	[5]
.....	.....	.....	.....
一个	(65)	|100001	[6]
B	(66)	|1011101	[7]
C	(67)	|1011110	[7]
D	(68)	|1011111	[7]
E	(69)	|1100000	[7]
.....	.....	.....	.....
L	(76)	|1100111	[7]
.....	.....	.....	.....
T	(84)	|1101111	[7]
.....	.....	.....	.....

要返回到前面的示例,请查看标题

:方法:删除

9 https://httpwg.org/specs/rfc7541.html#huffman.code

Machine Translated by Google

可以使用对索引 2 和编码值的引用来压缩此标头
删除：

信	丁	乙	戊	乙	吨	乙
霍夫曼码	1011111 1100000 1100111 1100000 1101111 1100000					

当分组为八位字节时，标头将是1011 1111 1000 0011 0011 1110 0000 1101 1111 1000 00，最后一个八位字节用 1 填充到0011。此标头转换为十六进制的bf 83 3e 0d f8 3f ，这需要前面是霍夫曼标志和长度。在这种情况下，长度为 6 个八位字节，二进制为110或填充为 7 位时为000 0110 。在长度八位位组（1000 0110或86)的开头添加霍夫曼编码位为1 ，最终得到完全编码的标头为86 bf 83

3e 0d f8 3f。

8.4.5 霍夫曼编码脚本

HPACK 霍夫曼编码和解码可以轻松实现自动化。下面的列表显示了 Perl 中的一个这样的实现。请注意，由于篇幅原因，仅显示了部分霍夫曼表。完整清单可在本书的 GitHub 页面上找到。 10

清单 8.1 一个简单的 HPACK 霍夫曼编码器

```
#!/usr/bin/perl

使用严格;使用警告;


#从命令行读取要转换的字符串。我的（$input_string)=@ARGV;


if (not defined $input_string) { die    需要输入字符串\n   ;

}

#设置并使用所有霍夫曼查找值填充散列变量。
#注意在这个简单的例子中只使用了可打印的值。我的 %hpack_huffman_table;


$hpack_huffman_table{      }=   010100   ;
$hpack_huffman_table{  !  }=   1111111000   ;
$hpack_huffman_table{  \     }=   1111111001   ;
$hpack_huffman_table{  #  }=   111111111010   ;
…ETC。
$hpack_huffman_table{  }  }=   11111111111101   ;
$hpack_huffman_table{  ~  }=   1111111111101   ;
```

[10] https://github.com/bazzadp/http2-in-action

Machine Translated by Google


```
#设置一个二进制字符串变量 my $binary_string=       ;


#按字符拆分输入字符串 my @input_array = split(//, $input_string);


#对于每个输入字符,在哈夫曼哈希表中查找字符串 #并将其添加到 binary_string 变量中。 foreach (@input_array) { $binary_string
= $binary_string . $hpack_huffman_table{$_};


}

#填充二进制字符串以确保它能被 8 整除 while (length($binary_string) % 8 != 0) {

                    $binary_string = $binary_string 。 “1”;
};

#除以8计算长度 my $string_length = length($binary_string)/8;


#This simple implementation doesn    t handle large strings #（留给读者作为练习）。如果
（$string_length > 127){

                    die     Error string length > 127 which isn    t handled by this program\n    ;

}

#Set the most significant bit (128) to indicate that Huffman encoding is used #and include the length #（同样,这个简单的版本天真地假
设长度为7位）。 printf(    霍夫曼编码标志 + 长度: %x\n    ,128+$string_length);



#遍历每个 4 位值并转换为十六进制 printf(    Huffman 编码值 for(my $count=0;$count<length($binary_string);
$count = $count + 4) { printf(    %x     ,oct(    0b     .substr($binary_string,$count,4)));



} printf(    \n    );
```

此代码将字符串编码为十六进制 HPACK 霍夫曼编码字符串。下面是一个例子：

```
$ ./hpack_huffman_encoding.pl DELETE Huffman 编码标志 +
长度:86 Huffman 编码值:bf833e0df83f
```

您可以使用类似的脚本来解码 HPACK Huffman 编码的字符串。可以增强此脚本以获取标题列表并处理动态表状态。我将这两项任务留作练习,供您使用您选择的语言完成。

虽然哈夫曼编码对人来说处理起来比较复杂,但是它很容易用代码实现,并且对计算机来说编码和解码效率很高。因此,您不太可能像我在本章中所做的那样手动编码或解码。

Machine Translated by Google

8.4.6 为什么霍夫曼编码并不总是最优的

对于某些值,霍夫曼编码可能会导致比使用纯 ASCII 时更大的值。例如,如果您决定使用 ASCII对delete进行编码,您可以直接跳转到十六进制（因为每个 ASCII 代码都是 1 个八位字节长）：

信	丁	乙	[illegible]	乙	吨	乙
ASCII 十六进制代码	44	45	4C	45	54	45

您可以看到 ASCII 编码版本的长度也是 6,并且将 Huffman 编码标志设置为0,在标头之前加上06并获得06 44 45 4C 45 54 45作为完全编码的标头。

在这种情况下,使用霍夫曼编码和不使用它之间没有区别　都是 7 个八位字节长。尽管此特定示例中使用的所有霍夫曼代码都是 7 位（与 8 位 ASCII 代码相比）,但需要填充以四舍五入为完整的八位字节,因此代码最终大小相同。对于其他一些标头,ASCII 编码可能更小,如果标头中使用了霍夫曼表中的一些不常见字符（长于 8 位）,就会出现这种情况。因此,HPACK 规范允许根据客户端的需要使用或不使用 Huffman 编码,并且编码可以随每个标头而变化。应使用能够以最少的八位字节表示值的任何编码。

然而,一般来说,霍夫曼编码通常比 ASCII 更有效。这种效率部分是由于 ASCII 只需要 7 位,但使用了完整的 8 位八位字节,因此每个 ASCII 编码值都浪费了 1 位。霍夫曼编码允许使用可变长度编码,因此理论上不会浪费位。然而,在这种可变长度编码中,不太常用的字符使用超过 8 位,因此这些值可以更有效地编码为 ASCII。根据定义,这些值应该是罕见的（假设 HPACK Huffman 编码表反映了现实生活中的使用）。最后,从静态或动态表中查找总是比以 Huffman 或 ASCII 格式编码更有效。

8.5 HPACK 压缩的真实例子

我在前面的部分介绍了很多理论。在本节中,我将向您展示一些真实的示例,以帮助您理解所有这些理论。大多数支持 HTTP/2 的工具会为您处理 HPACK,并且不会暴露它在这方面必须为您做的所有额外的艰苦工作,因此请返回 Wireshark 查看在线路上发送的原始数据（以及该数据的解码版本）。我在第 4 章介绍了 Wireshark,因此请参阅该章以使其正常工作。

假设您的 HTTP/2 流量嗅探工作正常,请开始查看 HTTP/2 标头。图 8.13 显示了一个请求 Facebook 的示例,其中选择了第一个HEADERS框架（窗口中的第二行）。

Machine Translated by Google

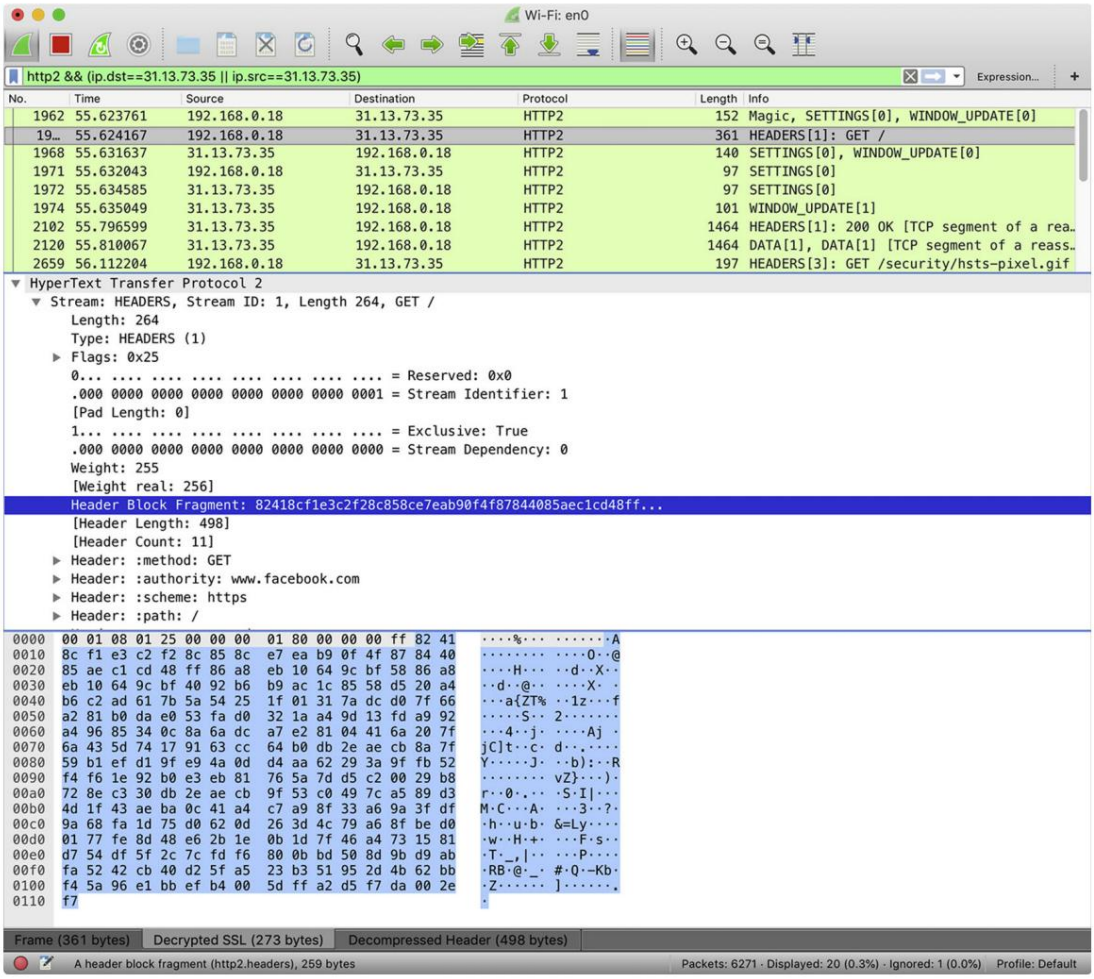


图 8.13 Wireshark 中的 HTTP/2 头帧

在图的底部,您会看到 Wireshark 帧（数据包的通用术语,不要与 HTTP/2 帧混淆)为 361 字节。该帧解密后变为 273 字节,解压后变为 498 字节。即使对于第一帧,标头压缩也很棒,节省了 45% (273/498 = 55%),尽管其中一些节省在数据加密时再次用完。

当你去掉 HTTP/2 帧的细节（比如头部类型、标志、重量等）,您将获得十六进制格式的编码标头：

82 41 8c f1 e3等等

Machine Translated by Google

用你新学到的知识解码这个标题。因为您正在处理可变长度的霍夫曼编码,所以将标头视为八位字节没有用,因此将其转换为二进制:

82418cf1e3 ... = 1000 0010 0100 0001 1000 1100 1111 0001 1110 0011 ...

此时,您可以开始阅读标题。你知道有四种类型的标题:

文字头字段表示（从1 开始） 带有增量索引的文字头字段
（从01 开始） 没有索引的文字头字段（从0000 开始） 从未索引的文字头
字段（从0001 开始）

这个头块以1 开头,所以它是第一种类型。接下来的 7 位给出了表索引,即2,相当于:method: GET 你的第一个未压缩的标头!
这个标头存储在 1 个八位字节(82)中,而不是用 ASCII 编码对这 12 个字符中的每一个字符进行编码所需的 12 个八位字节 这是一个巨大的节省。所以前 8 位被解码:

~~1000 0010 0100~~ 0001 1000 1100 1111 0001 1110 0011 ···

下一部分从01 开始,所以你知道它是一个带有增量索引的文字头字段。因为它后面没有跟 6 个零,所以您知道索引名称是由这 6 个字节引用的。图 8.14 重复了填充了二进制值的增量格式。

位								实际值
0	1个	2个	3个	4个	5个	6个	7	
0	1个	索引 (6+)						0100 0001
H	数值长度 (7+)							1000 1100
值字符串（长度八位字节）								1111 0001 等等

图 8.14 增量索引格式 1 的文字头字段

现在解码下一个标头的第一个八位字节(0100 0001)。
去掉01 后,索引号为00 0001或1,即表中的:authority标头。解码前 16 位:

~~1000 0010 0100 0001 1000 1100~~ 1111 0001 1110 0011 1100 ...

现在您需要找到该:authority标头的值。下一个八位位组的第一个字符是 1,因此您知道即将到来的值是霍夫曼编码值,长度是该八位位组的余数,或000 1100 = 8 + 4 = 12 个八位位组:

1000 1100 长度为 12 的霍夫曼编码字符串。

Machine Translated by Google

解码前 24 位：

~~1000 0010 0100 0001 1000 1100 1111 0001 1110 001~~1 1100 ...

当您查看接下来的 12 个八位字节时，您会看到以下内容：

1111 0001 1110 0011 1100 0010 1111 0010 1000 1100 1000 0101
1000 1100 1110 0111 1110 1010 1011 1001 0000 1111 0100 1111

读取每一位，直到找到唯一的霍夫曼值。我将执行 Huffman 表查找以节省您的时间（易于编程但手动执行起来要困难得多）：

1111000被唯一标识为w。 1111000被唯一标识为w。 1111000被唯一标识为w。 010111被唯一标识为..

等等

00100被唯一标识为c。 00111被唯一标识为o。 101001被唯一标识为m。 111 是填充以填充最后一个八位位组。

完整的值是www.facebook.com，当它与标头名称(:authority) 结合时，您将获得完整的标头：

:权威:www.facebook.com

此标头添加到索引为 62 的动态表中。静态索引以 61 个索引值结束，因此 62 是下一个空闲值。

以类似的方式继续HEADERS框架的其余部分，您最终动态表如表 8.5 所示。

表 8.5 收到第一个HEADERS帧后的动态报头

指数	标头	价值
62	接受语言	en-GB,en-US;q=0.9,en;q=0.8
63	接受编码	gzip、放气、br
64	接受	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
65	用户代理	Mozilla/5.0（Macintosh；Intel Mac OS X 10_14_0)AppleWebKit/537.36（KHTML，如 Gecko)Chrome/69.0.3497.100 Safari/537.36

Machine Translated by Google

表 8.5 收到第一个HEADERS帧后的动态报头（续）

指数	标头	价值
66	升级不安全请求 1	
67	缓存控制	无缓存
68	语用	无缓存
69	:权威	www.facebook.com

如您所见， :authority:索引标头已被一直向下推到索引 69。因此,标头的顺序很重要,可能与开发人员工具中显示的标头不匹配:Chrome 在 devel 中按字母顺序排列标头例如,操作工具但不按字母顺序发送它们。另请注意,并非所有标头都已添加。例如， :scheme: https和:path: / pseudoheaders 已经在静态表中（分别在索引 7 和 4 处）,因此它们没有添加到动态表中。

第一个请求可能涉及填充动态表,因为这些值不会在表中,因此您不会获得最佳压缩。后续请求可以使用这些值并获得更好的压缩,如图 8.15 所示。

我将跳过下一个接收到的HEADERS帧,而是转到下一个发送的HEADERS帧。您可以根据源 IP 地址和目标 IP 地址查看每个请求的传输方向。已发送和已接收的标头以相同的方式单独处理。在这里,我专注于客户端发送端;一切都同样适用于服务器发送端,但有一个单独管理的动态头表。

对于第二次发送的请求,你从82开始,和之前一样,你知道它是未压缩/未编码为:method: GET,这里不再赘述。第二个标题更有趣。 41是:authority标头,与之前的请求相同,后面是霍夫曼编码的权限(facebook.com)。有趣的是,这个域与第一个请求（facebook.com与www.facebook.com 相对)不同,因此不能重复使用之前存储的动态值,而必须发送该值。此示例还显示了连接合并操作,因为即使域不同,此请求也不需要单独的 HTTP/2 连接。有关连接合并的更多信息,请参阅第 6 章。

Machine Translated by Google

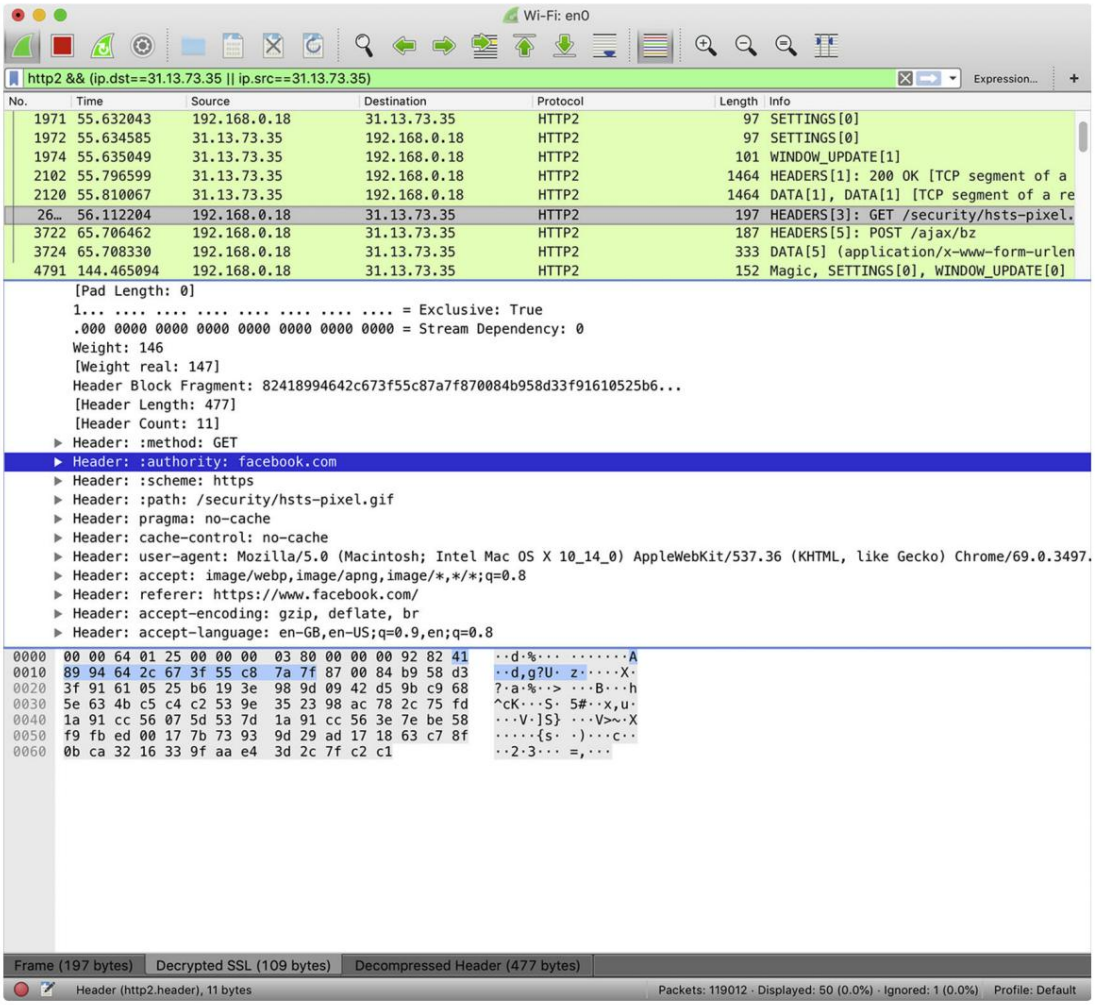


图 8.15 重用 HTTP/2 索引标头是允许发送大标头的有效方法。

接下来的几个标头很有趣,我用user-agent标头来演示它们,如图 8.16 所示。

如您所见,第一个请求需要完整的长用户代理标头,[11]使用 94 个八位字节来发送它。在第二个请求中,user-agent标头没有改变,因此可以用两个八位字节(c2)发送它以节省大量空间!与纯文本 HTTP/1.1 ASCII 标头所需的 131 个八位字节相比,这种节省甚至更大。

[11] 如果您对为什么这个标头这么长感到好奇,请查看https://webaim.org/blog/user-agent-string-history/ 。

Machine Translated by Google

第一个请求中的用户代理标头

```
▶ Header: :method: GET
▶ Header: :authority: www.facebook.com
▶ Header: :scheme: https
▶ Header: :path: /
▶ Header: pragma: no-cache
▶ Header: cache-control: no-cache
▶ Header: upgrade-insecure-requests: 1
▶ Header: user-agent: Mozilla/5.0 (Macintosh; Intel
▶ Header: accept: text/html,application/xhtml+xml,app
▶ Header: accept-encoding: gzip, deflate, br
▶ Header: accept-language: en-GB,en-US;q=0.9,en;q=0.
0000  00 01 08 01 25 00 00 00  01 80 00 00 00 ff 82 41
0010  8c f1 e3 c2 f2 8c 85 8c  e7 ea b9 0f 4f 87 84 40
0020  85 ae c1 cd 48 ff 86 a8  eb 10 64 9c bf 58 86 a8
0030  eb 10 64 9c bf 40 92 b6  b9 ac 1c 85 58 d5 20 a4
0040  b6 c2 ad 61 7b 5a 54 25  1f 01 31 7a dc d0 7f 66
0050  a2 81 b0 da e0 53 fa d0  32 1a a4 9d 13 fd a9 92
0060  a4 96 85 34 0c 8a 6a dc  a7 e2 81 04 41 6a 20 7f
0070  6a 43 5d 74 17 91 63 cc  64 b0 db 2e ae cb 8a 7f
0080  59 b1 ef d1 9f e9 4a 0d  d4 aa 62 29 3a 9f fb 52
0090  f4 f6 1e 92 b0 e3 eb 81  76 5a 7d d5 c2 00 29 b8
00a0  72 8e c3 30 db 2e ae cb  9f 53 c0 49 7c a5 89 d3
00b0  4d 1f 43 ae ba 0c 41 a4  c7 a9 8f 33 a6 9a 3f df
00c0  9a 68 fa 1d 75 d0 62 0d  26 3d 4c 79 a6 8f be d0
```

第二个请求中的用户代理标头

```
▶ Header: :method: GET
▶ Header: :authority: facebook.com
▶ Header: :scheme: https
▶ Header: :path: /security/hsts-pixel.gif
▶ Header: pragma: no-cache
▶ Header: cache-control: no-cache
▶ Header: user-agent: Mozilla/5.0 (Macintosh; Intel
▶ Header: accept: image/webp,image/apng,image/*,*/*;
▶ Header: referer: https://www.facebook.com/
▶ Header: accept-encoding: gzip, deflate, br
▶ Header: accept-language: en-GB,en-US;q=0.9,en;q=0.
0000  00 00 64 01 25 00 00 00  03 80 00 00 00 92 82 41
0010  89 94 64 2c 67 3f 55 c8  7a 7f 87 00 84 b9 58 d3
0020  3f 91 61 05 25 b6 19 3e  98 9d 09 42 d5 9b c9 68
0030  5e 63 4b c5 c4 c2 53 9e  35 23 98 ac 78 2c 75 fd
```

图 8.16 第一次和第二次请求中的用户代理头

要查看c2如何转换为用户代理标头,首先您需要意识到添加了一个新标头(:authority: facebook.com),并且它将存储在动态表中,将所有内容向上移动,如表中所示8.6.

表 8.6 收到第一个HEADERS帧后的动态报头

指数	标头	价值
62	:权威	脸书网
63	接受语言	en-GB,en-US;q=0.9,en;q=0.8
64	接受编码	gzip、放气、br
65	接受	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
66	用户代理	Mozilla/5.0（Macintosh;Intel Mac OS X 10_14_0）AppleWebKit/537.36（KHTML,如 Gecko)Chrome/69.0.3497.100 Safari/537.36
67	升级不安全请求 1	
68	缓存控制	无缓存
69	语用	无缓存
70	:权威	www.facebook.com

然后你可以解压用户代理头：

c5 = 1100 0010

Machine Translated by Google

第一个1是标头名称和值的文字查找， 100 0010转换为 66,它（如您在表 8.6 中所见)是先前存储的用户代理标头。

这只有在发送方和接收方都保持其动态 HPACK 表同步时才有意义,但是,只有在保留HEADERS帧的顺序的情况下才能做到这一点 这要归功于 TCP 的保证交付。这个过程可能看起来很复杂,特别是如果您手动进行的话,但它可以很容易地通过计算机自动执行。使用Wireshark 等工具的痛苦和出错率要低得多,但现在您知道如何手动解码,如果需要手动解码的话。

这种易于自动化的复杂性带来的收益令人印象深刻。在图 8.15 中,您可能已经注意到解密后的 SSL 大小为 109 字节,而图 8.13 中第一个请求的大小为 273 字节。即使对于这个简单的站点,通过此连接加载三个资源（其余资源从分片域加载)而没有任何大 cookie 或复杂的标头,您可以节省 68% 的字节,如在HEADERS帧中发送的字节,如图所示在表 8.7 中。

表 8.7 HPACK 标头节省

要求	解密的 SSL	解压头	保存
1个	273	498	45%
2个	109	477	77%
3个	99	547	82%
全部的	481	1522	68%

由于第一个请求是最少压缩的,因此使用的连接越多,平均收益就会增加。

8.6 客户端和服务器实现中的 HPACK

在结束本章之前,请考虑几个更重要的要点。首先,静态表和动态表中有几个重复的标题,其中一些显示在表 8.8 中。

表 8.8 HPACK 静态和动态表中的重复标头示例

指数	标头名称	标头值
1个	:权威	
2个	:方法	得到
3个	:方法	邮政
4个	:小路	/
5个	:小路	/index.html

Machine Translated by Google

表 8.8 HPACK 静态和动态表中的重复标头示例（续）

指数	标头名称	标头值
.....	.....	.....
19	接受	
.....	.....	.....
64	接受	文本/html,应用程序/xhtml+xml,应用程序/xml； q=0.9,图片/webp,图片/apng,*/*;q=0.8

我在本章前面提到过， DELETE的:method可以用索引 2 或 3 引用,后跟值DELETE,因为索引 2 和 3 都引用:method标头。类似地， :path在表中有两个条目用于公共:path值,并且在示例中收到第一个请求后，也接受。 HPACK 规范没有给出任何关于在这种情况下应该使用哪个索引的指导。发件人可以决定使用第一个实例、最后一个实例或中间的一个（如果存在的话）,或者他们可以选择不引用先前定义的标头名称索引号并添加另一个（如 Chrome 所做的:路径标头名称）。

作为浏览器如何以不同方式处理编码的另一个示例,Chrome 和 Firefox 对同一请求中的多个标头使用略有不同的方法。12如果您发送两个cookie标头,例如,在第一个标头编码后,您有两个对表中的cookie标头名称:静态表中的原始引用和动态表中对您编码的值的另一个引用。为了对第二个cookie标头进行编码,Chrome 使用索引 32 处的静态表引用,而 Firefox 使用它添加到动态表中的cookie标头（在大于 62 的索引位置）。

需要明确的是,所有这些方法都是有效的。只要引用最终解析为正确的标头名称,发送方就可以自由使用它喜欢的任何索引查找。
到目前为止,我给出的所有示例都是针对具有不同值的重复标头名称。但是规范明确指出,如果客户愿意,可以准确地复制整个名称/值对。客户端可以发送表中已有的相同标头和值,并附上索引它们的指令,而不是从先前索引的值中引用它们:13

> 动态表可以包含重复条目（具有相同名称和相同值的条目）。因此,解码器不得将重复条目视为错误。

第一次使用重复标头时,此过程会导致该标头的压缩减少,但从技术上讲,这是允许的。

[12] https://stackoverflow.com/questions/49437846/weird-http-2-hpack-encoding-in-firefox
[13] https://httpwg.org/specs/rfc7541.html#dynamic.table

Machine Translated by Google

最后,发件人不需要使用动态表。 nginx 网络服务器仅使用静态表,例如,14大概是因为它更易于实施和管理。有一个补丁可以实现完整的 HPACK 编码。15根据补丁作者的说法,这个补丁将压缩率提高了 40% 到 95%,但在撰写本文时它没有包含在基本的 nginx 代码中。 16

8.7 HPACK的价值

HPACK 乍一看可能很复杂且令人生畏（它的 RFC 几乎是整个 HTTP/2 规范本身的大小）,但我已将其压缩到一个单独的章节中。我希望本章已经揭开了 HPACK 的一些神秘面纱,并且如果您需要的话,将使 RFC 本身不那么令人生畏。大多数 HTTP/2 用户和 Web 开发人员可以忽略复杂的细节并接受 HTTP/2 标头以有效方式压缩,从而节省大量空间,尤其是在请求端,其中标头占数据的大部分。 Cloudflare 是最大的 CDN 之一,在启用 HPACK 后请求数据减少了 53%17。在响应端,尽管 HTTP 标头可能很大,但它们通常与 HTTP 正文相比相形见绌,因此节省的空间似乎不那么显着。 （Cloudflare 平均只节省了 1.4%。)但与下载相比,大多数消费者网络连接都限制了上传带宽,并且请求资源是第一阶段,因此请求端可以说是收益更重要的地方。

概括

有不同的压缩数据的方法。 HTTP 标头包含cookie 等敏感数据,因此它们不能使用与HTTP 正文相同的压缩技术,因为它们可以通过各种攻击泄露数据。

HPACK 是一种专门为 HTTP 头压缩而编写的压缩格式 HTTP/2 中的 sion。

HPACK 格式有一个特定的二进制格式,它使用一个预定义的公共头名称静态表（在某些情况下是值)和一个在会话期间创建的动态表。

不是表引用的值可以用 ASCII 或 Huffman 传输编码格式。

霍夫曼编码格式通常会产生较小的值。 在 HPACK 中有多种发送 HTTP 标头的方式,浏览器可能对 HTTP 标头进行不同的编码。

---

[14] https://trac.nginx.org/nginx/changeset/12cadc4669a7/nginx 和
[15] https://github.com/cloudflare/sslconfig/blob/master/patches/nginx_1.13.1_http2_hpack.patch
[16] https://twitter.com/ igrigorik/status/1029827634815856640 https://blog.cloudflare.com/hpack-the-
[17] silent-killer-feature-of-http-2/

Machine Translated by Google

Machine Translated by Google

第 4 部分

HTTP 的未来

到目前为止,您应该对 HTTP/2 协议有深入的了解,并且能够完全理解规范。在这最后一部分,我展望了 HTTP 的未来。 HTTP/2 就在此时此地,被越来越多的网站使用,但那些帮助定义互联网协议的人并没有固步自封。在某些方面,HTTP/2 已是老生常谈,人们已经在关注该协议的下一步进展。

第 9 章介绍 QUIC。 QUIC 是一种新协议,旨在继续从 HTTP/2 开始的工作并解决 TCP 层中较低的问题。它即将进行标准化（并且可能在本书出版时已经标准化）,但我怀疑它可能需要更长的时间才能广泛使用。 QUIC 使用了 HTTP/2 的许多概念,因此已经做到这一点的读者应该能够开始学习该协议并可能有助于其采用。

回到协议层,在第 10 章中,我还研究了 HTTP 及其可能在何处（以及如何)发展。 HTTP 一直是健壮和可扩展的,而 HTTP/2 延续了这一概念,因此有许多选项可以使协议更进一步。

Machine Translated by Google

Machine Translated by Google

9 TCP、QUIC 和 HTTP/3

本章涵盖

- TCP 效率低下
- TCP 优化
- QUIC 简介
- QUIC 和 HTTP/2 的区别

HTTP/2 的目标是改善 HTTP 协议固有的低效率,主要是通过允许单一的多路复用连接。在 HTTP/1.1 下,连接未被充分利用,因为它一次只能用于一种资源。

如果在响应请求时有任何延迟（例如因为服务器正忙于生成该资源）,连接将被阻止并且不会被使用。 HTTP/2 允许连接用于多个资源,因此其他资源在这种情况下仍然可以使用该连接。

除了防止浪费连接外,HTTP/2 还提供了改进的性能,因为 HTTP 连接本身效率低下。创建 HTTP 连接是有代价的;否则,多路复用就没有真正的好处。成本不是由于 HTTP 本身,而是由于用于创建此连接的两种基础技术:用于提供 HTTPS 的 TCP 和 TLS。

Machine Translated by Google

在本章中,我研究了这些低效问题,并表明尽管 HTTP/2 能够更好地处理大多数这些低效问题,但在某些情况下,由于这些低效问题,HTTP/2 可能比 HTTP/1.1 更慢。然后我讨论 QUIC,它做了一些改进。

9.1 TCP 低效和 HTTP

HTTP 依赖于保证数据可靠且有序地传送的网络连接。直到最近,这种有保证的连接是通过使用 TCP（传输控制协议)实现的。 TCP 允许在两个端点（通常是浏览器和 Web 服务器)之间建立连接,并负责传递消息,确保它们到达,如果消息没有到达则处理任何重新传输,并确保消息在之前正确排序被传递到任何应用层（HTTP）。 HTTP 不需要实现这些复杂功能中的任何一个;相反,它假定已满足这些标准。该协议是建立在该假设之上的。

TCP 通过为每个 TCP 数据包分配一个序列号,然后在到达时重新排列数据包（如果接收到的数据乱序)或重新请求任何丢失的序列号数据包（如果检测到数据包丢失)来强制执行此保证的完整性。 TCP 在 CWND 基础上工作（这也是 HTTP/2 流量控制工作原理的基础;参见第 7 章）,由此决定可以发送的最大数据量（CWND 大小）,并且发送的消息减少了这个窗口和确认的数据包再次增加它。窗口开始时很小,但会随着时间的推移而增长,因为事实证明网络容量能够处理增加的负载。如果客户端似乎跟不上,窗口也可以缩小。这个过程运行得相当好,TCP/IP 也因此成为互联网的支柱。然而,TCP 的基本工作方式也导致该协议存在五个主要问题,至少在 HTTP 方面是这样：

存在设置延迟。发送方和接收方要使用的序列号必须在连接开始时达成一致。 TCP 慢启动算法限制了TCP的性能,因为它对发送多少数据非常谨慎,尽可能避免重传。 连接使用不足会导致 TCP 节流。 TCP 缩放 CWND

如果连接未完全使用,则退回,因为无法确定自上次最佳 CWND 建立以来网络参数是否未更改。 丢失的数据包导致 TCP 节流。 TCP 假定所有数据包丢失都是由于

拥堵,但情况可能并非总是如此。

数据包可能会排队。乱序接收的数据包可能会被阻止以确保顺序得到保证。

这些问题在 HTTP/2 下没有改变,其中一些是在 HTTP/2 下使用单个 TCP 连接更好的原因。然而,最后两个问题可能导致 HTTP/2 在某些有损条件下比 HTTP/1.1 慢。

Machine Translated by Google

9.1.1 创建 HTTP 连接时的设置延迟

我在第 2 章中讨论了 TCP 三向握手。这种握手,加上 HTTP（以及所有浏览器对 HTTP/2 连接)越来越需要的 HTTPS 设置,导致在发送第一个 HTTP 消息之前出现明显的延迟,因为如图 9.1 所示。

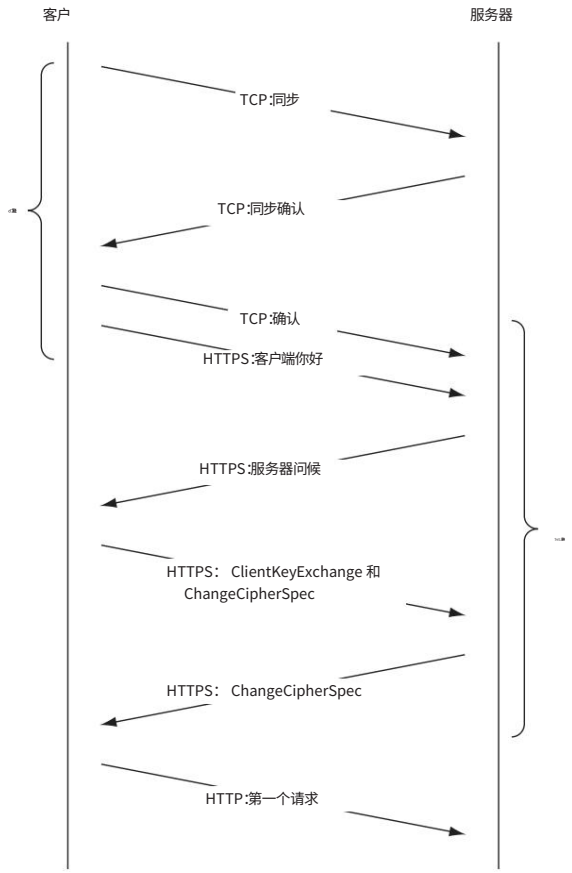


图 9.1 HTTPS 连接所需的 TCP 和 HTTPS 设置流量

根据 HTTPS 握手消息的大小,在您发送第一个请求之前,至少需要三次往返才能建立与服务器的连接（TCP 1.5,HTTPS 2,重叠 0.5）。此图不包括任何 DNS 查找,这可能会增加另一个延迟。

这些连接设置步骤在现实生活中会导致明显的延迟,尤其是在 HTTP/1.1 下,但在 HTTP/2 下也是如此。图 9.2 显示了第 2 章中亚马逊的瀑布图,其中突出显示了所有连接延迟。

在 HTTP/2 下,使用单个连接要好得多,但每个连接仍会出现初始延迟。此外,任何无法合并的单独域

Machine Translated by Google

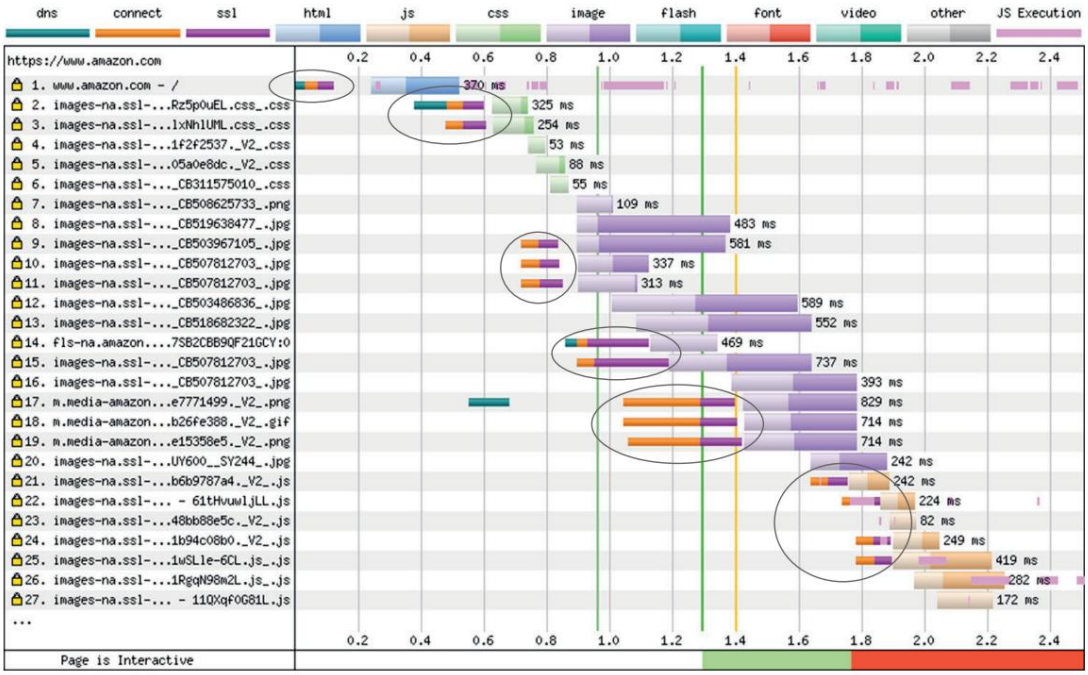


图 9.2 Amazon 在 HTTP/1.1 下的连接设置延迟

（见第 6 章)受到这些延误的影响。自图 9.2 以来,Amazon 已升级到 HTTP/2,但我仍然看到初始连接和任何无法使用相同 HTTP/2 连接的后续连接（因为它未解析到同一服务器或因为它是经过身份验证与未经过身份验证的）,如图 9.3 所示。

HTTP/2 大大减少了连接数,因此显着减少了图 9.3 中显示的 15 次左右的连接延迟,但最好也解决这三个剩余的延迟。

9.1.2 TCP 中的拥塞控制效率低下

即使在建立连接之后,TCP 的低效率也会导致其他性能问题,这主要是由于 TCP 的保证性质:所有 TCP 数据包都保证按顺序到达。这个看似简单的语句需要在协议中考虑几个因素,特别是拥塞控制。

拥塞控制旨在防止网络崩溃,当网络花费更多时间重新传输丢失的数据包而不是发送新数据包时。这个概念在 20 世纪 80 年代中期几乎成为现实,当时互联网开始腾飞。 1

[1] https://tools.ietf.org/html/rfc2914#section-3.1

Machine Translated by Google

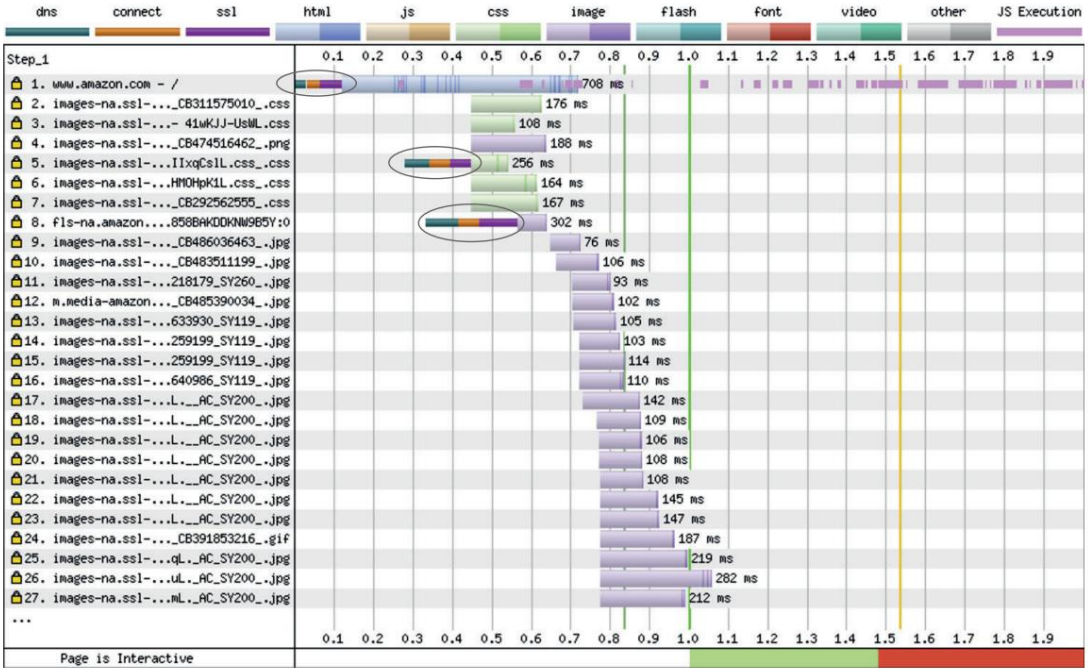


图 9.3 连接延迟大大减少,但仍低于 HTTP/2。

为防止这些问题,TCP 在 1980 年代后期得到了增强,具有各种拥塞控制功能,这些功能至今仍在不断调整和更改。这些拥塞控制算法和概念带来了稳定性,但也带来了低效,尤其是对于 HTTP。

TCP慢启动

TCP 慢启动机制可在网络上找到 TCP的最佳吞吐量,而不会淹没并可能危及网络。 TCP 是一种谨慎的算法,它以低速率开始并逐渐增加到最大容量,在此期间它会仔细监控它认为可以处理的连接和容量。

TCP 连接可以发送的数据量基于拥塞窗口大小。这个拥塞窗口保守地开始,对于现代 PC 和服务器,有 10 个 1460 字节最大段大小 (MSS) 的段,或大约 14 KB（一个相对较新的变化,2因为许多服务器仍然使用以前使用的四段大小） .在慢启动期间,拥塞窗口的大小随着每次往返而加倍,如表 9.1 所示。

---

[2] https://tools.ietf.org/html/rfc6928

Machine Translated by Google

表 9.1 典型的 TCP 缓慢启动增长

往返	移动安全系统	CWND 大小（段）	CWND 大小 (KB)
1个	1460	10	14
2个	1460	20	28
3个	1460	40	57
4个	1460	80	114
5个	1460	160	228

这种大小的翻倍会产生指数增长，并且在几次往返之后达到接收方表示愿意接受的最大容量（参见图 9.4 的第一部分）。

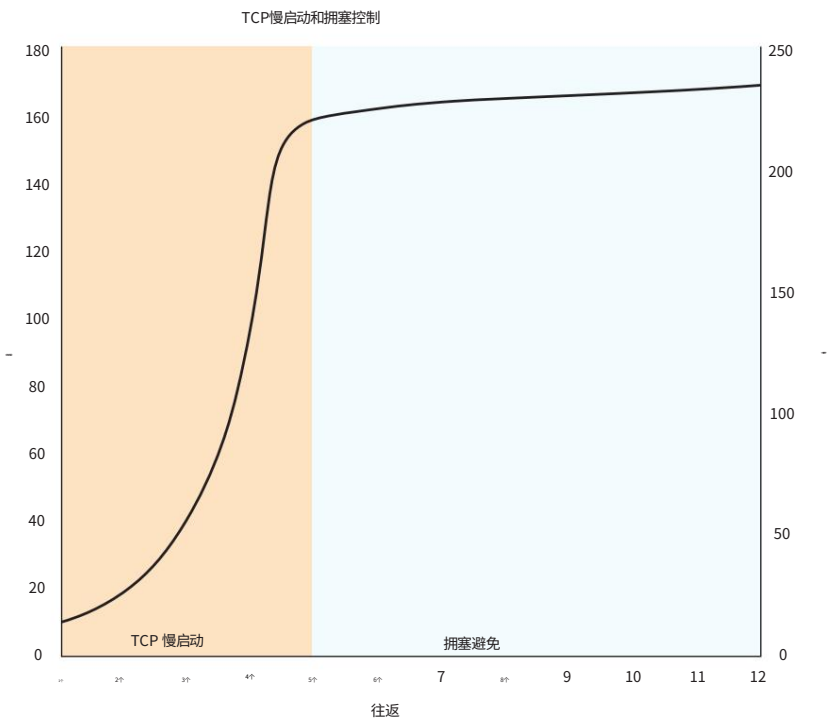


图 9.4 TCP 慢启动逐渐增加到最佳容量。

上限通常也比图 9.4 中显示的要低得多，65 KB 是常见的，因为这是 TCP 下的初始限制（请参阅第 9.1.4 节中有关窗口缩放的讨论）。达到最大容量后，假设没有发生丢包，TCP 拥塞控制进入拥塞避免阶段并继续增加，但线性速率要慢得多（相对于指数

Machine Translated by Google

TCP慢启动有那么慢吗?

由于 TCP 慢启动的指数性质,大多数定义并不慢。事实上,TCP 的拥塞避免阶段是一个非常慢的增长速度。慢启动是指最初的规模很小,这就是为什么它被称为慢启动而不是慢增长。

它肯定比从服务器可以发送的最大值开始并按比例缩小要慢。每个 TCP 连接都会经历这种增长,因此该协议最初很慢 故意、有意地变慢,但仍然很慢。

慢启动期间的速率),直到它开始看到丢弃的数据包并假设达到容量,如图 9.4 的第二部分所示。

不幸的是,对于 HTTP,初始阶段是您可能需要全部容量的地方。例如,在 Facebook 会话中,仅主页就有 125 KB,至少要到第四次往返才能达到。在初始下载网页及其所有资产后,下载数据的需求通常会减少,这通常发生在 TCP 达到其最佳容量时。

尽可能挤入前 14 KB一个经常吹捧的 Web 性能技巧是将所有关键资源放入 HTML 的前 14 KB。理论上,前 14 KB 将在前 10 个 TCP 数据包中下载,从而防止任何 TCP 确认延迟。例如,任何关键的内联 CSS 都应该包含在最初的 14 KB 中(假设浏览器乐于开始处理部分 HTML 页面,就像许多浏览器所做的那样)。

这种情况在 HTTPS 连接下发生变化,特别是在 HTTP/2 连接下,其中一些初始 10 个 TCP 数据包将被以下(至少)使用:

两个 HTTPS 响应(服务器问候和更改规范) 两个 HTTP/2设置帧(服务器发送一个并确认客户的设置框架)
一个HEADERS帧响应第一个请求

在最好的情况下,该连接会留下 5 个数据包,或大约 7 KB。实际上,这些消息中的任何一个都可能大于一个 TCP 数据包,使用超过五个数据包。此外,在您添加任何WINDOW_UPDATE或PUSH_PROMISE帧之后,这个数字可能会更小。

然而,幸运的是,客户端在发送这些 TCP 数据包时对其进行了确认,这增加了 CWND 的大小。在某些方面,由于 HTTPS 导致的初始延迟可能意味着在您使用 HTTP 时 CWND 大小已经更大,尽管这一事实被 HTTPS 本身的初始设置成本所抵消。

将关键资源放在 HTML 中较高位置的主要观点仍然存在,但在我看来,没有必要在 HTTPS 或 HTTP/2 下挂在 14 KB 上。

Machine Translated by Google

我在第 2 章中指出,具有单一连接的 HTTP/2 比 HTTP/1.1 有优势,但当您深入了解细节时,这种说法可能不是 100% 准确。
一方面,由于多个连接（通常每个域六个,如果使用分片,则更多）,HTTP/1.1 最初下载更多,因此与 HTTP/2 相比,它有效地获得多个初始 CWND（假设所有连接都打开)同时）,如表 9.2 所示。

表 9.2 具有六个连接的典型 TCP 缓慢启动增长

往返	MSS	CWND 大小（分段）	CWND 大小 (KB)	六个连接的 CWND 大小 (KB)
1个	1460	10	14	85
2个	1460	20	28	171
3个	1460	40	57	342
4个	1460	80	114	684
5个	1460	160	228	1368

另一方面,如果最初只使用一个连接（下载 HTML 网页时经常出现这种情况）,则 HTTP/1.1 下的任何其他新连接都以比单个 HTTP/2 连接更慢、更低的限制开始,这可能已经满负荷运转。 TCP 慢启动影响协议的两个版本

在一定程度上。

空闲连接会降低性能

TCP 慢启动算法会在连接开始时以及连接空闲时造成延迟。 TCP 是谨慎的,在闲置一段时间后,网络状况可能已经发生变化,因此 TCP 将其 CWND 大小调低并重新启动慢启动机制以再次找到最佳 CWND 大小。

不幸的是,Web 浏览本质上是由突发流量（当您导航到新页面时)和随后的空闲时间（当您阅读网页时)组成的。然后循环重复,因此在空闲期间重置回开始会对网页浏览产生很大影响。

在 Amazon HTTP/1.1 示例中,页面是从 Amazon 主域加载的,但使用的大部分资产都来自子域,导致与主域的初始连接出现长时间不活动,如图 9.5 中突出显示的那样。

虽然这种不活动对于突出显示的第一个连接特别不利（因为它可能会在任何后续页面导航中再次使用）,但您可以在图 9.5 中看到许多其他连接未得到充分利用。这些差距表明此连接的使用效率低下,如第 2 章中强调的那样,但从 TCP 的角度来看,它比您可能意识到的更糟糕,因为 TCP 在这些不活动期间会限制连接。当需要再次使用这些连接时（例如在页面导航上）,该过程几乎从头开始,尽管至少 TCP

Machine Translated by Google

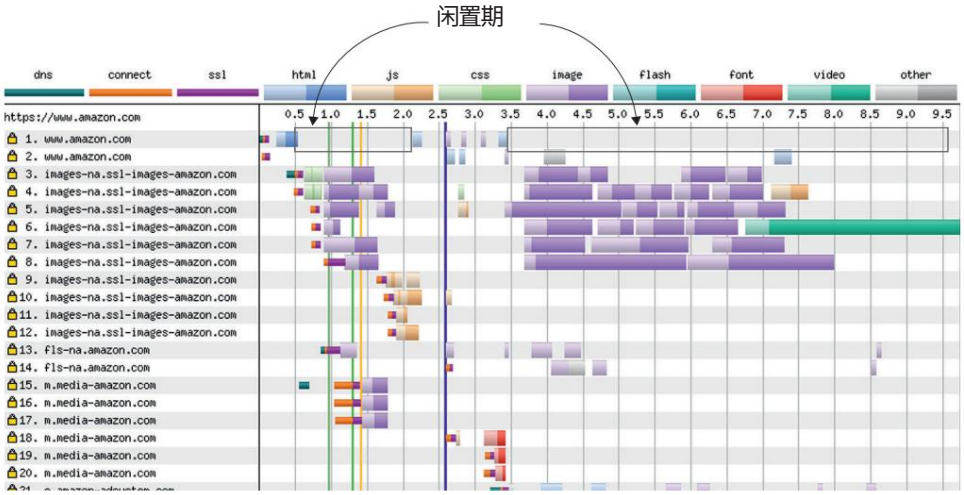


图 9.5 Amazon 在 HTTP/1 下使用的连接

如果连接保持打开状态,则无需重复握手和 HTTPS 握手。

HTTP/2,每个域使用一个连接,在这种情况下表现得更好。每个资源都有助于保持单个 TCP 连接处于活动状态,因此它不太可能处于空闲状态。如果任何连接通过 XHR 轮询、服务器发送的事件或类似技术定期与服务器通信,这种情况尤其重要。此类活动使连接保持预热状态并为下一页导航做好准备。

数据包丢失会降低TCP性能

除了在开始时和连接闲置一段时间后需要一段时间才能达到容量之外,TCP 将数据包丢失作为极端事件处理。它假设此事件是由于容量限制引起的并且反应迅速,将 CWND 减半,从而将容量减半（取决于使用的 TCP 拥塞算法）。3然后 TCP 使用拥塞避免算法来建立容量并继续拥塞避免阶段（同样取决于 TCP 算法）,如图 9.6 所示。

CWND 减半会导致特定问题。丢包的原因有很多,网络拥塞只是其中之一。例如,移动网络可能不如有线连接可靠,并且可能会随机丢失数据包,无论网络容量有多大。因此,假设数据包丢失纯粹是由于拥塞造成的可能是错误的,因此应该会导致容量急剧下降。

---

[3] https://ieeeexplore.ieee.org/document/7796870

Machine Translated by Google

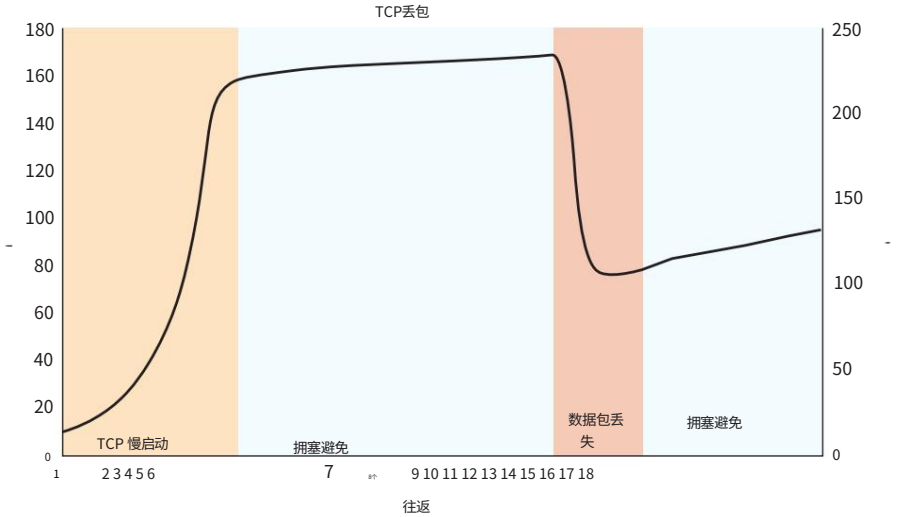


图 9.6 TCP CWND 大小受丢包影响。

较新的 TCP 拥塞控制算法对这些网络丢弃的处理比前面的示例要温和:它们可能会将 CWND 减少不到一半,在丢包后增长更快（类似于慢启动）,或者使用丢包的替代方法来决定容量（例如将平均往返时间视为比丢包更好的指标）。但在高层次上,这些概念大致相似。数据包丢失会导致网络速度变慢。

这个问题对于单连接的 HTTP/2 尤为严重。在 HTTP/2 世界中丢失单个数据包会导致所有正在下载的资源受到影响。
将此协议与 HTTP/1.1 进行比较,后者可能有六个独立的连接。单个数据包丢失会减慢其中一个连接的速度,但其他五个连接将继续满负荷运行。

在连接恢复之前,第二次数据包丢失可能会在 HTTP/2 下产生更可怕的后果,它再次减半至 25% 的容量（再次假设使用基本的 TCP 拥塞控制）。在 HTTP/1.1 下,第二次数据包丢失可能发生在已经遇到问题的连接上（也将容量减少到 25%)或单独的连接（减少到 50%）,但其他 TCP 连接不受影响。表 9.3 显示了六个资源的结果

通过 HTTP/2 和两种 HTTP/1.1 方案下载。
正如你在表 9.3 中看到的,在这个例子中,HTTP/2 下的平均容量下降到 25%,因为整个连接（所有六个资源正在通过它下载)都受到影响,而对六个独立连接的影响是减少到 83% 到 88%,具体取决于受影响的连接。

Machine Translated by Google

表 9.3 HTTP/2 与 HTTP/1.1 连接的第二次丢包结果

资源	HTTP/2	HTTP/1.1:相同的连接	HTTP/1.1:不同的连接
资源 1	25%	25%	50%
资源 2	25%	100%	50%
资源 3	25%	100%	100%
资源 4	25%	100%	100%
资源 5	25%	100%	100%
资源 6	25%	100%	100%
平均	25%	88%	83%

如果此连接特别糟糕,或者存在真正的容量瓶颈,则 HTTP/1.1 和 HTTP/2 都会受到影响。然而,在 HTTP/2 下效果总是更好;它的单一连接总是承受任何数据包丢失的全部冲击。

数据包丢失导致项目排队HTTP/2 多路
复用允许多个请求流在同一 TCP 连接上并行运行。在 HTTP/1.1 下做同样的事情需要多个 TCP 连接。当您有数据包丢失以及容量减少时,就会出现 HTTP/2 特定问题。假设您有三个正在运行的资产,它们正在下载,如图 9.7 所示。

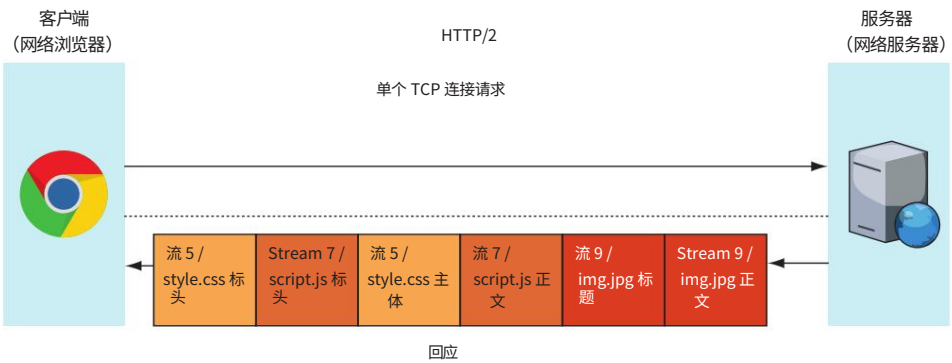


图 9.7 几个正在运行的响应

现在假设来自第一个响应的 TCP 数据包 在流 5 上发送的 style.css 标头响应 由于某种原因丢失了。在这种情况下,客户端不会确认该数据包,一段时间后,服务器会重新发送它。这个重传被添加到队列的末尾,如图 9.8 所示。请注意,为简单起见,该图模糊了 HTTP/2 帧和 TCP 数据包之间的界限。

Machine Translated by Google

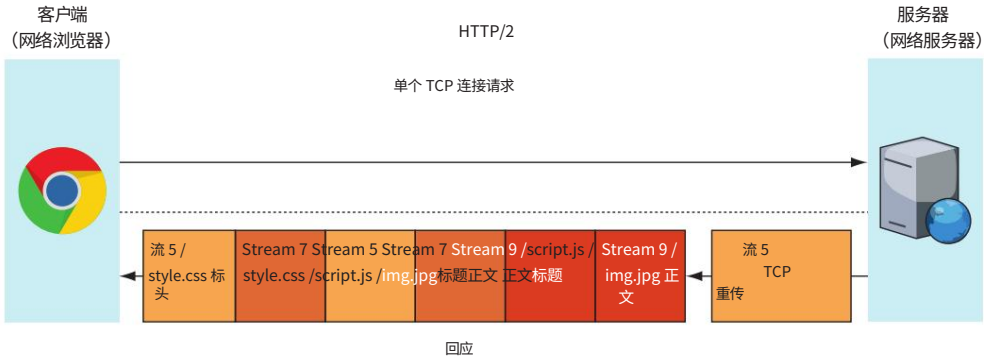


图 9.8 部分 HTTP/2 帧的 TCP 重传

如果没有其他数据包丢失发生，则在重传到达之前将完整接收流 7 和 9。但是，这些响应必须排队，因为 TCP 保证了顺序，因此即使已完全下载，也无法使用 script.js 和 image.jpg。在 HTTP/1.1 下，这个过程将在三个独立的 TCP 连接下进行，如图 9.9 所示。

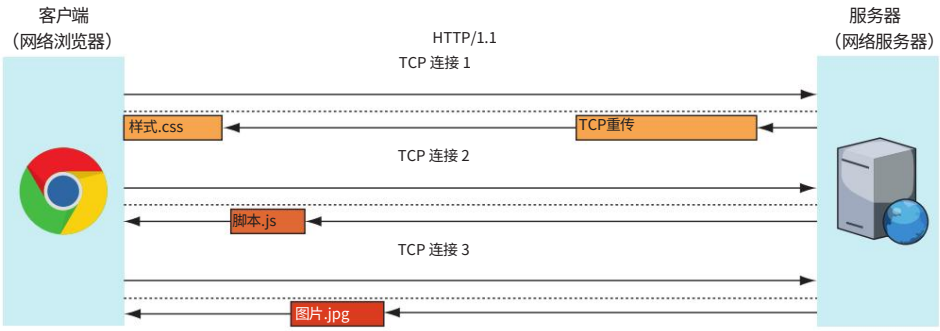


图 9.9 HTTP/1.1 下的 TCP 重传只影响需要重传的连接。

因此，浏览器可以在 script.js 和 image.jpg 到达时立即处理它们；只有 style.css 被延迟。在这个例子中，浏览器可能会等到 style.css 可用，这取决于它是否认为这个资源是关键资源（CSS 通常是这样）。重点仍然是 HTTP/2 在这里添加了一个约束，而在具有多个连接的 HTTP/1.1 下不存在。更糟糕的是，如果连接由于 TCP 缓冲区大小有限而无法对所有乱序数据包进行排队，它可能会丢弃一些数据包，从而也需要重新传输它们！

HTTP/2 解决了 HTTP 级别的线头 (HOL) 阻塞问题，因为通过多路复用，单个延迟响应不会阻止 HTTP 连接用于其他资源。然而，HOL 阻塞仍然存在于 TCP 级别。

Machine Translated by Google

来自一个流的单个丢弃数据包有效地阻塞了所有其他流,即使它们可能不需要被阻止。

9.1.3 TCP 低效对 HTTP/2 的影响

我已经展示了 TCP 低效率会导致 HTTP 出现问题,但真正的影响是什么,它在 HTTP/1 和 HTTP/2 下有什么不同吗?

我之前指出 HTTP/2 通常优于 HTTP/1.1。此外,谷歌对 SPDY 的实验表明,在实验室实验和现实世界中都获得了相当大的速度提升。

然而,性能损失的影响不可低估。 Fastly 的 Hooman Beheshti使用 WebPagetest 工具进行了一些实验4,结果表明,当持续出现 2% 的数据包丢失时,HTTP/2 的性能始终比 HTTP/1.1 差。诚然,持续 2% 的数据包丢失表明网络非常糟糕;大多数网络偶尔会丢失一个数据包,而不是经历这种一致的丢失程度。但实验表明,HTTP/2 可能不是适用于所有场景的灵丹妙药。更深入的研究5同样显示了数据包丢失的影响,甚至建议在 HTTP/2 下使用有限分片,这似乎有悖常理。

我能够在一些热门网站上重复 Beheshti 的发现,但在其他网站上却不行。如果您想重复测试,请访问 https://www.webpagetest.org/。在 Test Settings 选项卡上,选择自定义设置并设置丢包,如图 9.10 所示。

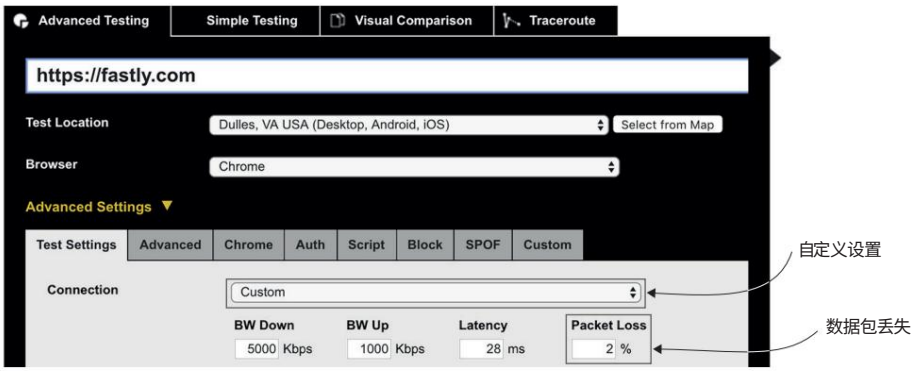


图 9.10 在 WebPagetest 中测试丢包

---

[4] https://www.youtube.com/watch?v=CkFEoZwWbGQ和https://www.youtube.com/watch?v=wR1gF5Lhcq0
[5] https://arxiv.org/abs/1707.05836

Machine Translated by Google

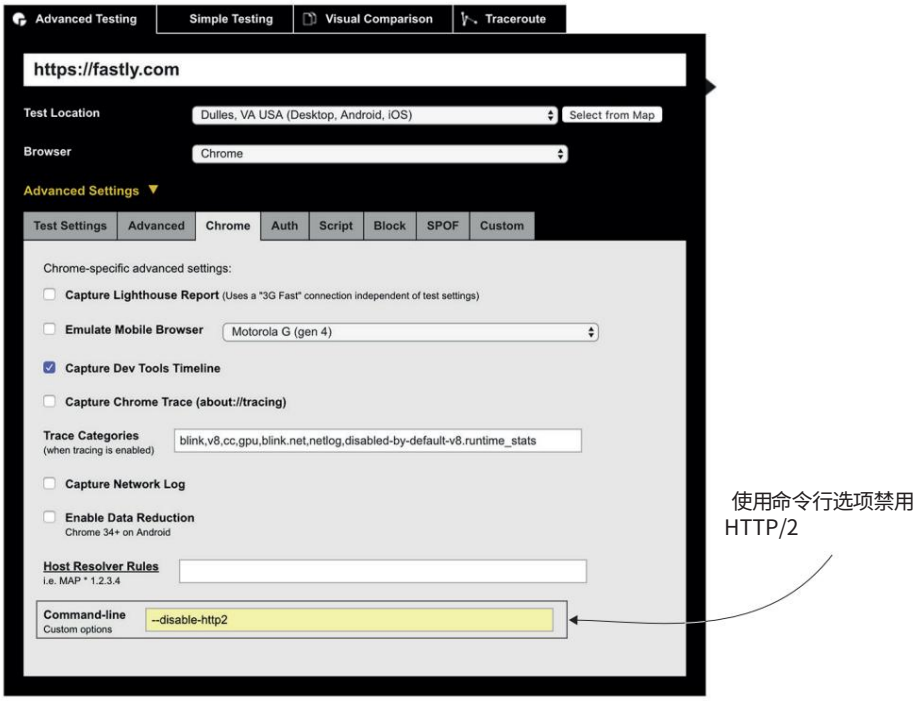


图 9.11 为 Chrome 禁用 HTTP/2

要测试 HTTP/2 与 HTTP/1.1,您可以使用 Chrome 并添加--disable http2命令行选项,如图 9.11 所示。

对于 Firefox,您必须使用稍微不同的方法。上输入以下内容
脚本选项卡,将最后的导航行替换为您要测试的站点:

firefoxPref network.http.spdy.enabled firefoxPref 错误的
network.http.spdy.enabled.http2 false firefoxPref network.http.spdy.enabled.v3-1 false导航

https://www.fastly.com/

请注意,在这些设置的各部分之间必须使用制表符,而不是空格,如图 9.12 所示。

为防止任何偏差或防止个别结果影响整体结果,请在不同的网络条件下和不同的位置多次运行测试。图 9.13 显示了 ebay.com 的一项测试的结果。

如图 9.13 所示,HTTP/2（顶部)慢了将近半秒。重复
零数据包丢失的相同测试表明 HTTP/2 更快,正如预期的那样。

也可以通过单击右侧的原始页面数据来导出原始数据
页面,如图9.14所示。

Machine Translated by Google

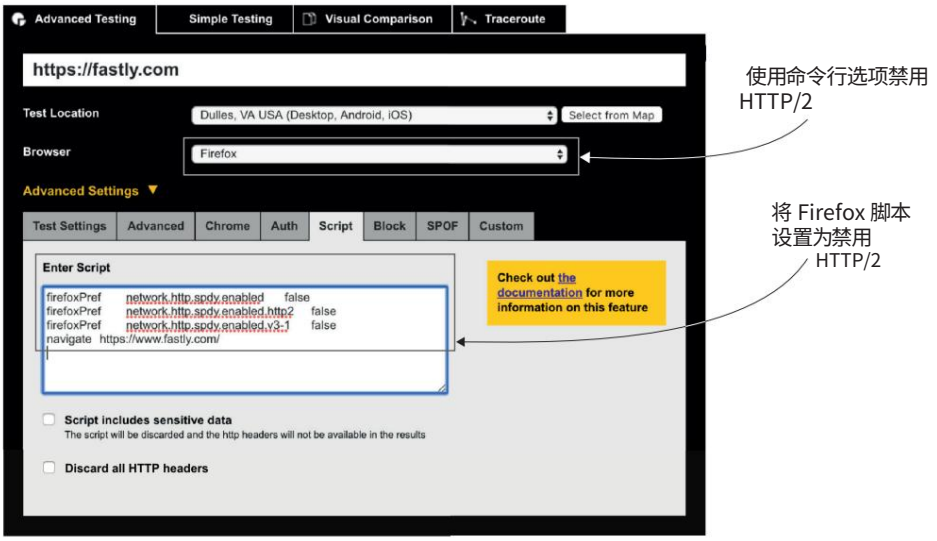


图 9.12 在 WebPagetest 中禁用 Firefox 上的 HTTP/2

丢包率为 2% 的 HTTP/2

性能结果（中值运行）

	Load Time	First Byte	Start Render	Speed Index	First Interactive (beta)	Document Complete Time	Document Complete Requests	Document Complete Bytes In	Fully Loaded Time	Fully Loaded Requests	Fully Loaded Bytes In	Fully Loaded Cost
First View (Run 7)	3.101s	0.299s	0.800s	3.386s	> 5.485s	3.101s	38	843 KB	7.310s	135	2,265 KB	$$$$$

HTTP/1.1 有 2% 数据包丢失

性能结果（中值运行）

	Load Time	First Byte	Start Render	Speed Index	First Interactive (beta)	Document Complete Time	Document Complete Requests	Document Complete Bytes In	Fully Loaded Time	Fully Loaded Requests	Fully Loaded Bytes In	Fully Loaded Cost
First View (Run 4)	2.672s	0.374s	1.000s	4.790s	6.637s	2.672s	38	826 KB	11.998s	137	2,487 KB	$$$$$

图 9.13 通过 HTTP/2 和 HTTP/1.1 加载 Ebay 主页,丢包率 2%

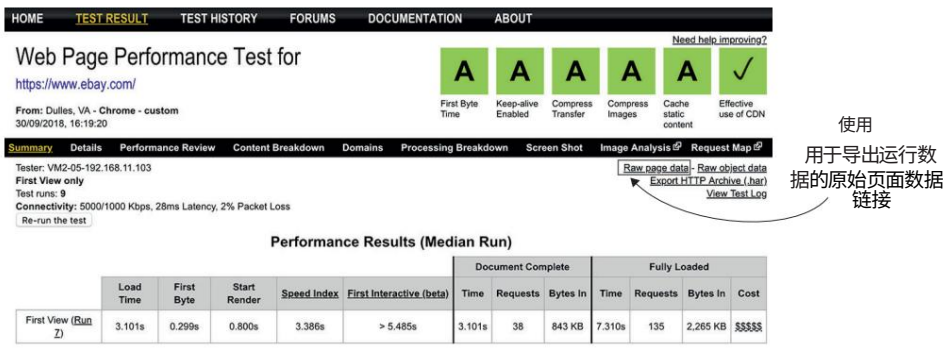


图 9.14 将 WebPagetest 原始数据导出到 CSV 文件

Machine Translated by Google

当您进行多次运行并希望将它们绘制在图表中时,此功能会很方便。另外,您可以单击“测试历史记录”并选择两个图像以查看效果的快速比较,如图 9.15 所示。

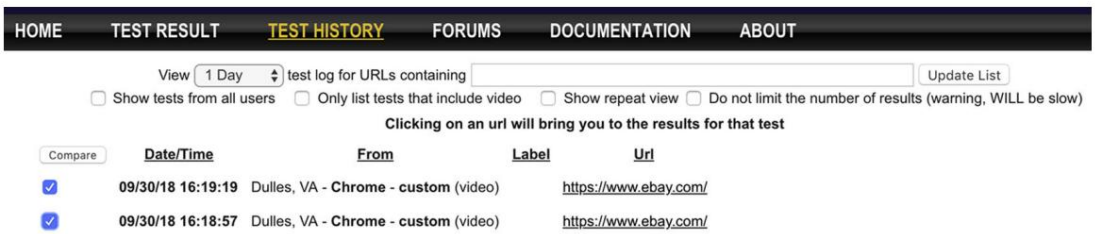


图 9.15 选择两个结果进行比较

此功能使您可以访问大量视图和数据,包括按时间轴和缩略图显示的大小,如图 9.16 所示。

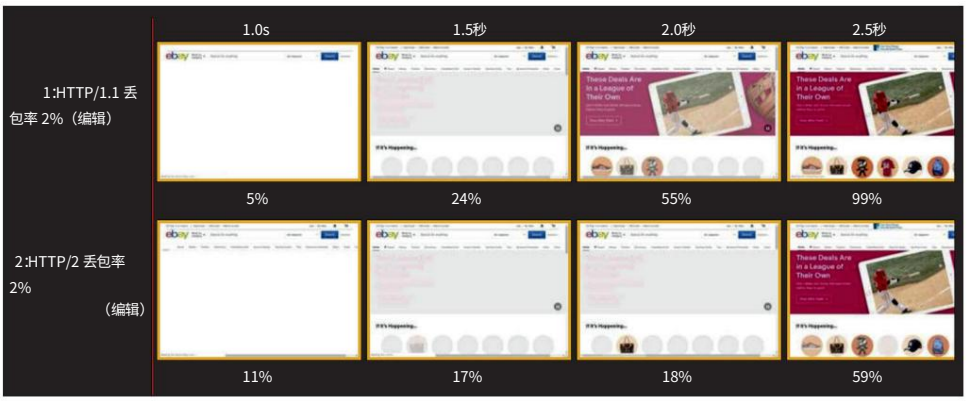


图 9.16 比较两个 WebPagetest 运行

WebPagetest 是运行此类性能比较的绝佳工具。
您还可以托管自己的私有实例,如果您计划执行大量检查或想要测试无法用于 Web 版本的开发服务器,这非常值得研究。

由于 TCP 效率低下,您是否应该推迟迁移到 HTTP/2?

鉴于 HTTP/2 性能在严重丢包的情况下可能会变差,您是否应该推迟迁移到 HTTP/2?拖延可能会矫枉过正。
请记住,在大多数情况下,HTTP/2 比 HTTP/1.1 更快。你是否应该坚持让用户受益,因为在某些情况下（理想情况下很少见）,它的性能比 HTTP/1.1 差?

Machine Translated by Google

本章讨论 HTTP 和 TCP 的一些实际问题,但不讨论这些问题发生的可能性。最好的衡量标准是现实生活中的指标,而不是像本章中那样的人为场景。正如我之前所说,经历持续 2% 数据包丢失的真实网络很可能是一个糟糕的网络。不幸的是,在现实生活中测量数据包丢失更加困难。统计数据不太容易获得。然而,一些基于更现实的数据包丢失场景的科学研究表明,正如预期的那样,一般来说 HTTP/2 优于 HTTP/1.1。

HTTP/2 实现仍然相对较新,并且会随着时间的推移而改进。网站也是如此,可能会针对 HTTP/2 优化得更好。最后,TCP 本身仍在改进并且可以优化（参见第 9.1.4 节）。有些人建议对 HTTP/2 使用多个 TCP 连接来解决其中的一些问题,但这种解决方法否定了在 HTTP/2 下移动到单个连接的原因。 HTTP/2 主要表现更好,因为它使用单个连接。

在大多数用户的网络连接较差且无法改善的特定情况下,明智的做法是继续使用 HTTP/1.1 或对不会合并的 HTTP/2 连接进行分片。最终,最好的建议（一如既往)是衡量和测试任何变化。

---

https://www.semanticscholar.org/paper/HTTP%2F2-Performance-in-Cellular-Networks-Goel Steiner/63fa6b3310a7c4d799d5b0b5bf37f0620dd3fc5d?tab=abstract

9.1.4 优化TCP

您已经看到 TCP 可以极大地影响 HTTP 的性能。在许多方面,HTTP 协议中的低效问题已通过 HTTP/2 解决,但其他地方存在的性能瓶颈现在更加明显。由于多路复用， HTTP HOL 阻塞不再是 HTTP/2 中的问题,但 TCP HOL 阻塞已成为一个问题,尤其是在有损环境中。

这些问题的唯一解决方案是改进 TCP 或摆脱它。以下部分将介绍第一种解决方案； 9.2 节着眼于第二个。这些年来,TCP 已经有了一些改进,其中一些可能已经是

在某些环境中使用。

升级您的操作系统

最大的作用是升级你的操作系统。尽管 TCP 历史悠久,可追溯到 1974 年,但随着新研究的完成和计算机使用的变化,仍在进行改进和创新。不幸的是,TCP 通常由低级操作系统控制,您很少有机会在操作系统之外更改它。因此,确保最佳 TCP 使用的最佳方法是确保您运行的是最新版本的操作系统。

在本节中,我以 Linux 为例,但这些设置同样适用于其他操作系统,包括 Windows 和 macOS，即使设置不在同一个位置或不容易更改。在适当的情况下,我提供

Machine Translated by Google

引入更改的 Linux 版本。表 9.4 显示了一些最流行的 Linux 发行版中包含的 Linux 内核版本。

表 9.4 流行发行版的 Linux 内核版本

分配	内核版本
瑞尔/森托斯 6	2.6.32
瑞尔/森托斯 7	3.10.0
Ubuntu/Debian 14.04	3.13
Ubuntu/Debian 16.04	4.4
Ubuntu/Debian 18.04	4.15
Debian 8 杰西	3.16
Debian 9 延伸版	4.9

查找和更改 TCP 连接设置

Linux 中的大多数 TCP 设置都可以在以下目录中查看：

```
/proc/sys/net/ipv4/
```

尽管有目录名称，但大多数这些设置也适用于 IPv6 TCP 连接。您可以使用cat 查看值：

```
$ cat /proc/sys/net/ipv4/tcp_slow_start_after_idle
1
```

您可以使用sysctl命令设置值：

```
sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

但是，在更改任何这些设置时要小心，因为 TCP 是系统的关键部分。我建议大多数读者不要更改设置。相反，我建议读者使用这些知识来确保这些设置是适当的，并将它们用作整个操作系统升级的参数，应该将这些值设置为内核发布时的最佳实践值。

增加初始CWND大小

TCP 慢启动需要往返以增加 CWND 的大小，它从 1 个 TCP 数据包的大小开始，但随着时间的推移增加到 2，然后是 4；在 Linux 内核 2.6.39 中，默认情况下该设置从 4 增加到 10。此设置通常被硬编码到内核代码中，因此除非升级操作系统，否则不建议更改它。

Machine Translated by Google

允许窗口缩放

传统上,TCP 允许最大 CWND 窗口大小为 65,535 字节,但后来的版本允许将比例因子应用于该值,理论上允许 CWND 大小最大为 1 GB。这个设置在 Linux 内核 2.6.8 中被设置为默认设置,所以对于大多数读者来说它应该是打开的,但是为了确保,你可以这样检查它:

```
$ cat /proc/sys/net/ipv4/tcp_window_scaling
1
```

使用选择性确认

选择性确认 (SACK)允许 TCP 确认收到乱序的数据包,以避免在另一个数据包丢失时重新发送它们。如果数据包 1-10 已发送,但数据包 4 被丢弃,您可以确认 1-3 和 5-10。这样,只有数据包 4 必须重新发送。如果没有此功能,则本例中的数据包 4–10 需要重新发送。使用以下命令确认此功能已设置为 1（开启）:

```
$ cat /proc/sys/net/ipv4/tcp_sack
1
```

禁用慢启动重启

此设置仍然默认为可能错误的值,至少对于 Web 服务器而言,因此您可能需要考虑更改它。在假设网络条件可能已经改变的情况下,TCP 连接在空闲期后会节流,因此之前的假设可能不正确。然而,默认情况下,网络服务器有点间歇性,当用户浏览网站时会出现流量爆发,暂停阅读网页,并可能浏览到其他页面,因此启用此设置可能不是网络服务器的最佳选择。

该设置通常默认启用:

```
$ cat /proc/sys/net/ipv4/tcp_slow_start_after_idle
1
```

要禁用它,请使用以下命令:

```
sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

如前所述,您不应该轻易更改系统 TCP 设置。但是根据您的服务器的用途（例如专用 Web 服务器）,更改此设置可能值得考虑。

使用TCP快速打开

TCP 快速打开允许使用TCP 三向握手的初始SYN部分发送初始流量包。此方法可防止与 TCP 相关的某些设置延迟（请参阅第 9.1.1 节）。出于安全原因,此数据包只能在 TCP 重新连接时发送,而不是在初始连接时发送,并且需要客户端和服务器支持。 TCP 快速打开有效地允许在握手中更早地发送 HTTP（或 HTTPS)消息,如图 9.17 所示。

Machine Translated by Google

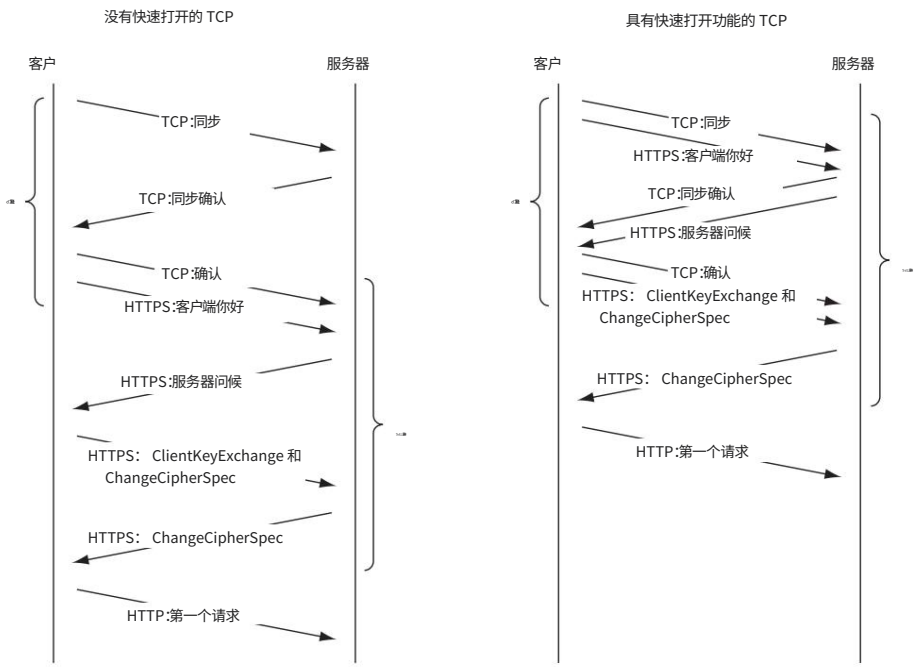


图 9.17 使用和不使用快速打开的 TCP 和 HTTPS 重新连接握手

您可以按如下方式查看 Linux 对该功能的支持：

```
$ cat /proc/sys/net/ipv4/tcp_fastopen 0
```

该设置通常被禁用（设置为0）。表 9.5 列出了此设置的一些选项。

表 9.5 TCP 快开设置

价值	意义
0	残疾人
1个	为传出连接启用
2个	为传入连接启用
3个	为传出和传入连接启用

您可以使用以下命令更改此设置：

```
echo 3 > /proc/sys/net/ipv4/tcp_fastopen
```

Linux 3.7 中添加了对该功能的支持，并在 3.13 版中默认启用，但 IPv6 支持直到 Linux 3.16 才添加。

Machine Translated by Google

除了在操作系统级别设置此设置外,您还必须配置服务器软件才能使用它。在 Web 服务器端,nginx 允许此设置,但需要编译选项和配置,因此默认情况下未启用。 Windows IIS7 支持它,但是 Apache 在文档中没有提到它,所以推测它不支持它。其他不太常见的服务器可能不支持此功能。在客户端,可以在 Edge、8和 Android 上的 Chrome 中启用该设置,但在撰写本文时,它在 Chrome for Windows 或 macOS9 中不受支持,并且在 Firefox.10 中关闭

TCP Fast Open 带来的收益确实令人印象深刻。谷歌表示 11 “基于流量分析和网络仿真,我们表明 TCP Fast Open 将 HTTP 事务网络延迟减少 15%,整个页面加载时间平均减少 10% 以上,在某些情况下甚至高达 40%。”然而,对这个相对较新的 TCP 新增功能（RFC 于 2014 年发布)12的支持进展缓慢。鉴于这些复杂性,TCP Fast Open 可能是未来值得关注的一个,而不是现在改变。

使用拥塞控制算法、 PRR和BBR

TCP 有多种拥塞控制算法,可以控制 TCP 在丢包时如何反应。大多数 TCP 实现使用CUBIC算法13 （自 Linux 内核 2.6.19 以来的默认算法）。该算法通过比例速率降低 (PRR)14拥塞避免（自 3.2 以来的默认设置)得到增强,它减少了丢包时拥塞控制窗口的一半。 15 差异的详细描述超出了本书的范围,但是可以说更好的算法可以显着提高性能就够了。使用此命令查看当前使用的算法：

```
$ cat /proc/sys/net/ipv4/tcp_congestion_control cubic
```

可用的拥塞控制算法可在此处获得：

```
$ cat /proc/sys/net/ipv4/tcp_available_congestion_control reno cubic
```

---

https://nginx.org/en/docs/http/ngx_http_core_module.html#listen

7 https://blogs.technet.microsoft.com/networking/2016/07/18/announcing-new-transport-advancements in-the- anniversary-update-for-windows-10-and-windows-server-2016/ https://www.windowscentral.com/ enable-tcp-fast-open-microsoft-edge-faster-page-load-times https:/ /bugs.chromium.org/p/chromium/

9 issues/detail?id=635080 https://bugzilla.mozilla.org/show_bug.cgi?id=1398201 https://ai.google/research/

10 pubs/pub37517 https: //tools.ietf.org/html/rfc7413 https://tools.ietf.org/html/rfc831 https://tools.ietf.org/

11 html/rfc6937 https://ai.google/research/pubs/pub37486

12

13

14

15

Machine Translated by Google

一种更新的算法,瓶颈带宽和往返传播时间 (BBR),已被证明可以进一步提高性能,16特别是对于 HTTP/2 连接。17 BBR 由谷歌创建,可在 Linux 内核 4.9 中使用;它不需要客户端更改。要在具有它的 Linux 内核（版本 4.9 或更高版本)中启用它,请使用以下命令：

```
#动态加载 tcp_bbr 模块,如果尚未加载 sudo modprobe tcp_bbr #添加公平队列流量监管,BBR
与 sudo echo    net.core.default_qdisc=fq    > /etc/sysctl.conf #将 TCP 拥塞算法更改为
BBR sudo echo    net.ipv4.tcp_congestion_control=bbr    > /etc/sysctl.conf #重新加载设置
sudo sysctl -p
```

然而,一些研究人员 18声称 BBR 可能不是网络上的好玩家,尤其是在与其他非 BBR 流量一起运行时,并且可能占用不公平的网络资源比例。

9.1.5 TCP 和 HTTP 的未来

我已经向您展示了 TCP 的一些复杂性　一种看似简单的协议,但实际上比大多数人意识到的要复杂得多。与 HTTP/1.1 一样,TCP 具有一些内在的低效性,用户可能现在才开始体验到,因为解决了 HTTP 等更高级别协议的低效性,以及对网络的需求

继续增加。

该协议仍在不断发展,尽管速度相当缓慢。尽管一直在创建新的选项和拥塞控制算法,并且浏览器也在升级以利用这些功能（如果可用）,但它们需要一些时间才能进入服务器的网络堆栈。新功能通常与操作系统的基本功能相关,因此需要对操作系统进行全面升级。如果您运行的操作系统版本已经引入了这些功能但没有默认设置,则可以手动打开其中一些设置,但通常最好升级操作系统。这个领域是一个专门的领域,虽然我在这里提到了一些可能对未来几年有益的最近的创新,但通常最好让操作系统的维护者拥有必要的技能和知识,决定这些设置应该是什么。

此外,我没有涉及通常位于用户的 Web 浏览器和 Web 服务器之间的所有网络管道和管道。即使双方都支持这些相对较新的 TCP 特性中的一些,如果它们之间的任何东西都不支持,那么也有可能

---

[16] https://cloudplatform.googleblog.com/2017/07/TCP-BBR-congestion-control-comes-to-GCP-your-Internet-just-got-faster.html https://blog.cloudflare.com/http- 2-prioritization-with-nginx/ https://doc.tm.uka.de/2017-kit-icnp-bbr-authors-copy.pdf

[17]

[18]

Machine Translated by Google

一个问题。就像 HTTP 代理将连接降级到 HTTP/1.1 一样,即使两端都支持 HTTP/2,这一领域的创新也会被所谓的 mid dleboxes 阻碍。因为 TCP 是一种旧算法,所以这些中间盒中的一些对 TCP 的使用方式有一定的期望并且反应不佳,或者不允许以新的、意想不到的方式使用它。

由于这些原因以及更多原因,一些人质疑 TCP 是否是 HTTP 的正确基础协议,以及新协议是否是可行的方法 一种为 HTTP 当前（和未来)需求而设计的协议,没有包袱过去或对操作系统的依赖。一种这样的协议是 QUIC。

9.2 快速

QUIC（发音为quick)是 Google（又是 Google！)发明的一种基于 UDP 的新协议,旨在取代 TCP 和传统 HTTP 堆栈的其他部分,以解决本章中提到的许多低效问题。 HTTP/2 引入了一些类似 TCP 的概念（例如数据包和流量控制）,但 QUIC 将这些概念提升到了一个新的水平并取代了 TCP。

QUIC 代表什么?

QUIC 最初是 Quick UDP Internet Connections 的首字母缩写词,正如引入该协议时在大多数 Google Chromium 文档中所显示的那样。a、 b、c在正式化过程中,QUIC 工作组决定放弃这个首字母缩写词,d和 QUIC 规范明确注释,“QUIC 是一个名称,而不是首字母缩略词。”e

但是,许多来源仍然使用首字母缩略词。工作组的一名成员有趣地表示,“QUIC 不是首字母缩写词。你应该大声喊出来 ;)。”f

---

a https://www.chromium.org/quic

b https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNS DHLq9D5Bty4FSU/ https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-

c saqsQx7rFV- ev2jRFUoVD34/ https://github.com/quicwg/base-drafts/pull/1282 https://tools.ietf.org/html/

d draft-ietf-quic-transport#section-2 https://www.ietf。组织/邮件存档/web/quic/current/msg03844.html

e

f

QUIC 的创建考虑了以下特点:19

显着减少连接建立时间改进的拥塞控制无 HOL 线路阻塞的多路复用前向纠错

连接迁移

---

19 https://www.chromium.org/quic

Machine Translated by Google

从本章讨论的 TCP（和 HTTPS)缺点来看,前三个原因应该是显而易见的。最后两个原因是有趣的补充,进一步解决了这些问题。

前向纠错(FEC) 旨在通过在相邻数据包中包含 QUIC 数据包的一部分来减少数据包重传的需要。这个想法是,如果只有一个数据包被丢弃,应该可以从成功传送的数据包中重新组合该数据包。该过程已与“网络级别的 RAID 5”进行了比较。20我之前说过,数据包可能会随机丢失,但这不一定是连接限制的标志,FEC 旨在纠正这个问题。 QUIC 增加了冗余和开销成本,但考虑到 HTTP 需要保证交付这一事实（与视频流协议不同,在视频流协议中,数据包可能会被丢弃而不会产生任何影响）,获得的收益可能是值得的。在撰写本文时,QUIC 的这一功能仍处于实验阶段21,不会出现在 QUIC 的初始版本中,因为它在 QUIC-WG 章程中被明确称为超出范围。 22

连接迁移旨在通过允许连接在网络之间移动来减少连接设置开销。在 TCP 下,连接链接到两端的 IP 地址和端口。更改 IP 地址需要建立新的 TCP 连接。当 TCP 被发明时,这个要求是可以接受的,因为 IP 地址被认为在会话的生命周期内不太可能改变。

现在,有了多种网络（有线、无线和移动）,这种情况不再是理所当然的。因此,QUIC 允许您在家中通过 Wi-Fi 开始会话,然后无需重新启动会话即可转移到移动网络。

您甚至应该能够通过一种称为多路径的技术同时使用您的 Wi-Fi 和移动网络进行一个 QUIC 连接,该技术允许增加带宽。同样,此多路径功能在第一个版本中不可用,但连接迁移应该可用。

9.2.1 QUIC 的性能优势

2015 年 4 月,谷歌发布了一篇关于 QUIC 性能优势的博文23,包括以下内容：

- 75% 的连接利用了零往返连接时间。
- Google 搜索的平均页面加载时间缩短了 3%,并且在最慢的网络上将页面加载时间减少了一秒。这些数字可能看起来并不多,但请记住,Google 搜索是一个经过大规模优化的网站,任何改进都是特别的。
- YouTube 用户报告说,使用 QUIC 时,重新缓冲减少了 30%。

---

[20] https://ma.ttias.be/googles-quic-protocol-moving-web-tcp-udp/
[21] https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk
[22] https://datatracker.ietf.org /wg/quic/about/
[23] https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html

Machine Translated by Google

这些测量结果可能与 HTTP/2 和 SPDY 进行了比较。当时,50% 的 Chrome 流向谷歌的流量使用的是 QUIC;自那时以来,这一百分比可能已经大幅增长。因为 QUIC 直到最近才被 Chrome 和谷歌支持（见第 9.2.6 节）,所以它的使用受到限制。例如,W3Tech 表示,在撰写本文时,略多于 1% 的网站使用 QUIC,24尽管其他衡量标准表明这一数字相当于 7.8% 的流量,25其中 98% 是谷歌。

9.2.2 QUIC 和互联网栈

QUIC 取代的不仅仅是 TCP。图 9.18 显示了 QUIC 在传统 HTTP 技术堆栈中的位置。

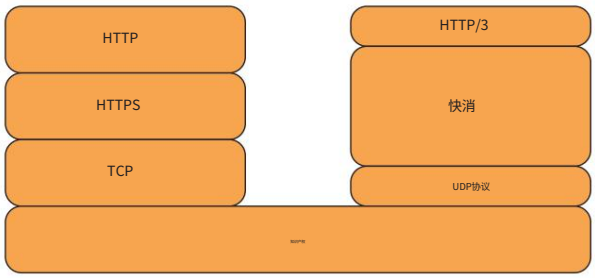


图 9.18 QUIC 在 HTTP 技术栈中的位置

如图 9.18 所示,QUIC 取代了 TCP 传统上提供的大部分功能（设置、可靠性和拥塞控制部分）、所有 HTTPS（以改善设置延迟）,甚至部分 HTTP/2（流量控制和标头压缩部分）。

QUIC 旨在通过同时执行连接层（传统世界中的 TCP)和加密层（传统世界中的 TLS)来建立单程连接。为此,它使用了许多已添加到 TCP（例如 Fast Open)和 TLS（例如 TLSv1.3)的概念和创新。

在更高的层次上,QUIC 并没有取代 HTTP/2,但它接管了一些传输层部分,让更轻量级的 HTTP/2 实现在上面运行。随着从 HTTP/1.1 到 HTTP/2 的转变,大多数高层开发人员需要关心的 HTTP 核心语法在 QUIC 下保持不变,而 HTTP/2 中引入的概念（例如多路复用流、标头压缩、和服务器推送)仍然以几乎相同的方式存在； QUIC 负责一些较低级别的细节。

从 HTTP/1.1 到 HTTP/2 的转变为开发人员带来了更大的变化,但所有概念在 QUIC 下都保持不变,因此您在本书中阅读和学习的一切都没有浪费!该协议仍然是一个多路复用的、基于流的二进制协议,用于在较低级别实现此更改的一些细节现在属于 QUIC 而不是 HTTP/2。反映 HTTP/2 的变化,以区分

[24] https://w3techs.com/technologies/details/ce-quic/all/all
[25] https://blog.apnic.net/2018/05/15/how-much-of-the-internet-is-using-quic /

Machine Translated by Google

它来自 QUIC 本身,并且为了表明这是 HTTP 的最佳版本,已经同意基于 QUIC 的 HTTP 将被称为 HTTP/3（在第 10 章的 10.3 节中有更多讨论）。[26]

9.2.3 什么是 UDP 以及为什么 QUIC 建立在它之上

QUIC 基于用户数据报协议 (UPD),与 TCP 相比,它是一种轻型协议,但同样建立在互联网协议 (IP) 之上。 TCP在IP中为网络连接实现了可靠性,包括重传、拥塞和流量控制。这些功能通常是好的和必要的,但在 HTTP/2 中,它们会导致效率低下。在 HTTP/2 下,网络级别不一定需要这些功能;因此,它们会产生不必要的 TCP HOL 阻塞问题。

与 TCP 相比,UDP 是基本的。它有端口的概念,类似于TCP,因此可以在同一台计算机上运行多个基于UDP 的服务。它还具有可选的校验和,以便检查 UDP 数据包的完整性。除了这两个功能,该协议没有太多内容。可靠性、排序和拥塞控制不存在,如果你想要它们,它们必须由应用程序构建。

如果 UDP 数据包丢失,它不会自动重发。如果 UDP 数据包乱序到达,它仍然可以被更高级别的应用程序看到。 UDP 最初用于不需要交付保证的应用程序（例如视频,其中可以丢弃一些帧而不会造成太多服务损失）。 UDP 也非常适合 HTTP/2 等多路复用协议,如果更高级别的协议想要对这些问题实施比 TCP 中可用的更好的解决方案。

为什么不改进TCP?

最明显的问题是为什么不改进 TCP? TCP 仍在创新,问题可以通过进一步改进来解决。主要缺点是实施任何此类改进的速度。 TCP 是一个核心协议,它几乎总是嵌入到操作系统中,虽然可以对其进行一些更改来配置它或可以在服务器端进行一些改进,但需要操作系统升级才能从大多数 TCP 改进中受益。问题不在于操作系统不能创新;这是广泛部署这些创新所需的时间长度。 TCP Fast Open 就是一个典型的例子;它提供了巨大的好处,但还没有被绝大多数互联网浏览器或服务器使用。

互联网基础设施加剧了这种创新缓慢,互联网基础设施对 TCP 等协议做出某些假设,并在这些假设被打破时做出糟糕的反应。这个问题被称为协议僵化,创新因这些假设而被扼杀。通过摆脱 TCP,QUIC 希望拥有更大的自由度和更少的约束。

---

[26] https://lists.w3.org/Archives/Public/ietf-http-wg/2018OctDec/0065.html

Machine Translated by Google

为什么不使用SCTP?

QUIC 可以使用Stream Control Transmis 27 ,而不是在 UDP 之上构建新的传输协议或等待 TCP 的创新变得更加广泛,该协议与 QUIC 具有许多相同的特性,例如sion Protocol (SCTP)。基于流的可靠消息传递,但它已经存在并且自 2007 年以来一直是互联网标准。

不幸的是,现有的标准不足以确保使用,SCTP 的采用率很低,主要是因为 TCP 到目前为止已经足够好了。因此,迁移到 SCTP 可能需要与升级 TCP 一样长的时间。即使采取这样的举措,协议的创新也可能停滞不前。 QUIC 旨在改善流级拥塞控制和其他影响互联网的问题,例如 HTTPS 握手、有限的数据包丢失和连接迁移。

为什么不直接使用IP ?

QUIC 设计人员可以使用的另一个选择是在 IP 上构建,因为传输层的要求很低。 IP 只不过是源 IP 地址和目标 IP 地址;其他一切都可以建立在它之上。

但是直接使用 IP 与使用 SCTP 存在同样的问题。该协议必须在操作系统级别实施,因为很少有应用程序可以直接访问 IP 数据包。此外,QUIC 应该针对特定的应用程序,因此它需要 UDP 具有的端口。许多客户端可以通过 QUIC 打开单独的 HTTP 连接,例如同时运行 Chrome 和 Firefox,也许还有一个使用 HTTP 的无关程序。如果没有这个特性,一些 QUIC 控制应用程序将需要读取所有 QUIC 数据包并将它们路由到适当的每个应用程序。

UDP的优点

UDP 是一种基本协议,也在内核中实现。任何建立在它之上的东西都需要建立在应用层,即用户空间。处于内核之外可以通过在任一侧部署应用程序来实现快速创新。当您使用 Chrome 时,Google 在其所有服务中使用 QUIC,因此打开开发人员工具并导航到 Google 站点会显示您当前使用的 QUIC 版本（撰写本文时为 43 版）,如图 9.19 所示。

在 QUIC 面世的短短几年内,Google 已经创建了 43 个版本。28就像在部署 SPDY 时所做的那样,Google 能够将更改部署到用于浏览网络 (Chrome) 的主要客户端和一些最常用的客户端。流行的服务器很容易,然后在用户不注意的情况下进行创新。截至 2017 年,估计有 7% 的互联网使用 QUIC,29尽管这个数字很可能主要代表谷歌网站。

---

[27] https://tools.ietf.org/html/rfc4960版本

[28] 历史详见源码： https://chromium.googlesource.com/chromium/src/+/master/net/third_party/quic/core/quic_versions 。H。 https://ai.google/research/pubs/pub46403

[29]

Machine Translated by Google

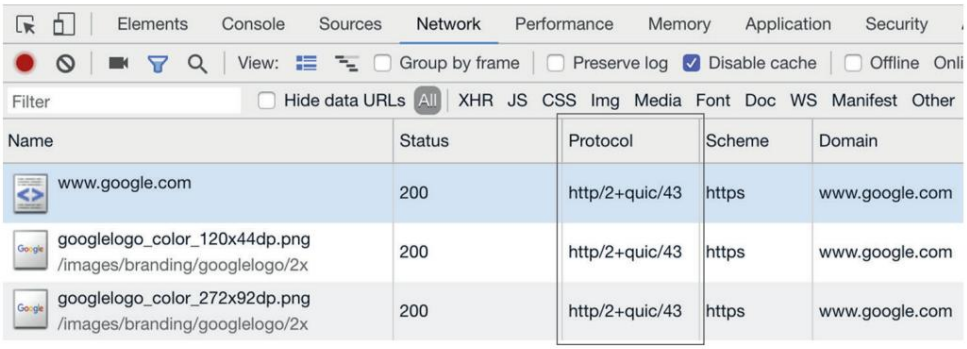


图 9.19 查看在www.google.com上部署的 QUIC 版本

只有使用 UDP 才能如此迅速地推出 QUIC,而不是试图强制采用或更改现有协议,这需要时间,而且很可能会被当前互联网的大部分基础设施所阻止。使用轻量级和有限的 UDP 允许谷歌构建和创新它认为合适的协议,因为它可以控制连接的双方。

UDP 并非没有问题。它是一种通用协议,但不如 TCP 通用。
例如,DNS 在 UDP 上工作,因为它是一个简单的协议,不需要复杂或缓慢的 TCP（尽管有一些措施允许 DNS 在 HTTPS 上工作,如第 10 章所述）。其他应用程序（例如实时视频流和在线视频游戏)也使用 UDP,因此它通常得到网络基础设施的支持。然而,TCP 更为常见,而 UDP 通常在默认情况下被防火墙和中间件流量阻止。在这种情况下,Chrome 优雅地回退到 HTTP/2 over TCP。这种担忧在一开始是一个很大的问题,但谷歌的实验表明 93% 的 UDP 流量通过了它,并且这个百分比随着时间的推移而有所提高。尽管一些基础设施会阻止 HTTP 的 UDP 流量（其中也使用端口 443）,但绝大多数都不会。如果 UDP 变得普遍（至少对于 Google 服务而言）,UDP 也很容易启用。

UDP 的另一个问题是用户空间并不总是像高度优化的内核空间那样高效。 QUIC 的早期测量表明,服务器使用的 CPU 是基于 TLS/TCP 的等效服务器的 3.5 倍。 30尽管该使用已被优化为仅两倍,但结果仍然表明 UPD 是一种更昂贵的协议,并且当它存在于内核之外时,它可能会保持这种状态。

[30] https://dl.acm.org/citation.cfm?id=3098842

Machine Translated by Google

QUIC 会一直使用 UDP 吗?

谷歌在推出 QUIC 时发布的原始常见问题解答中表示,“我们希望 QUIC 功能在证明有效的情况下能够迁移到 TCP 和 TLS。”

所以也许UDP会被用于实验,TCP会以较慢的速度随之演进。 QUIC 会在某个时候恢复为 TCP 吗?这个问题很难回答,但我的看法是很难放弃进化的自由。互联网似乎正处于传输层的创新时期,协议开发人员似乎不太可能达到他们乐于停止创新并安于固定的、难以升级的协议的地步。

此外,与 TCP 相比,QUIC 正在实现根本性的变化,即使存在这样做的动力,这些变化也不会轻易被 TCP 采纳。

更有可能的是,HTTP 将继续在 TCP (HTTP/2) 和 UDP（QUIC 和 HTTP/3)上可用,但 TCP 实现在功能和性能方面将落后于 UDP。

https://docs.google.com/document/d/1lmL9EF6qKrk7gbazY8bIdvq3Pno2Xj_l_YShP40GLQE/

9.2.4 标准化 QUIC

QUIC 最初是谷歌协议,并于 2013 年 6 月公开宣布。31 谷歌在接下来的两年内对该协议进行了改进,并于 2015 年 6 月将其作为提议标准提交给了互联网工程任务组 (IETF)。 32这提交发生在谷歌的最后一个标准（SPDY)被正式采用为 HTTP/2 之后,所以时机很好;许多与该标准化相关的人可以自由地从事 QUIC 的工作。几个月后,IETF QUIC 工作组成立,致力于协议的标准化。 33

两个QUICS： GQUIC和IQUIC与 SPDY 一样,QUIC 在 Google 的管理下发展,而标准化过程贯穿其中。在撰写本文时,这种演变导致了两种实现:gQUIC（用于 Google QUIC)和 iQUIC（用于 IETF QUIC）。谷歌继续在 gQUIC 中运行其生产环境,并在其认为合适的情况下继续发展和改进该协议,而无需为每个更改获得正式批准。与 SPDY 一样,gQUIC 预计会在 iQUIC 正式标准化后消亡（预计在 2019 年初发生）,但就目前而言,gQUIC 是该协议在生产环境中唯一可用的版本。

只有 Chrome 和基于 Chromium 的浏览器（例如 Opera)实现了 QUIC（它使用 gQUIC）,并且 gQUIC 随着 Google 团队的发展而频繁变化

31 https://blog.chromium.org/2013/06/experimenting-with-quic.html https://
32 datatracker.ietf.org/doc/draft-tsvwg-quic-protocol/00/ https://
33 datatracker.ietf .org/wg/quic/about/

Machine Translated by Google

改变它。34在服务器端,所有的谷歌服务都支持 gQUIC。在撰写本文时,其他网络服务器实现包括 Caddy35和 LiteSpeed,36但由于它们基于不断发展的、非标准化的 gQUIC,因此它们需要跟上 Google 的变化,可能会落后并停止与 Chrome 一起工作。 37

GQUIC和IQUIC之间的差异这个主题随着每个协议的进步而发展,但在撰写本文时,gQUIC 和 iQUIC 之间的主要区别之一在于加密层。谷歌使用自定义密码设计,而 iQUIC 使用 TLSv1.3.38之所以做出此选择,只是因为在发明 QUIC 时 TLSv1.3 不可用。谷歌表示,当 TLSv1.3 获得正式批准后,它将用 TLSv1.3 替换其自定义密码设计,39现在已经发生了,因此 gQUIC 和 iQUIC 很可能会融合。这两个协议之间存在一些不兼容的其他变化,但在概念层面上,除了使用 TLSv1.3 外,它们是相似的。

QUIC标准在撰写本文时,没有一个 QUIC 标准,只有六个!与由两个标准（HTTP/2 和 HPACK)组成的 HTTP/2 一样,QUIC 对其主要部分有单独的标准： 核心传输协议 QUIC Recovery42丢失检测和拥塞控制 QUIC TLS43QUIC 中如何使用 TLS 加密 HTTP/344主要基于 HTTP/2 并进行了一些更改 QUIC QPACK45QUIC One 中用于 HTTP 的标头压缩已经提出了更多实验性文件： QUIC Spinbit46将添加一个位,用于对加密的 QUIC 连接进行基本监控。有两个关于使用 QUIC 的附加信息文档,用于应用程序开发人员 47和用于在网络上管理 QUIC。 48

34 请参阅https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/ 的“按版本划分的最新更改”部分。 https://github.com/mholt/caddy/wiki/QUIC https://
35 blog.litespeedtech.com/2017/07/11/litespeed-announces-quic-support/ https://
36 github.com/mholt/caddy /issues/2194 https://tools.ietf.org/html/rfc8446 https://
37 docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g
38 https://tools.ietf.org/html/draft-ietf- quic 不变量 https://tools.ietf.org/html/draft-ietf-quic-
39 transport https://tools.ietf.org/html/draft-ietf-quic-recovery https://tools.ietf.org /html/draft-ietf-
40 quic-tls https://tools.ietf.org/html/draft-ietf-quic-http https://tools.ietf.org/html/draft-ietf-quic-
41 qpack https: //tools.ietf.org/html/draft-ietf-quic-spin-exp https://tools.ietf.org/html/draft-ietf-quic-
42 applicability https://tools.ietf.org/html/ draft-ietf-quic-可管理性
43
44
45
46
47
48

Machine Translated by Google

在撰写本文时,IEFT 工作组正在处理这些文档。由于该标准仍在制定中,因此这些规范（甚至它们的数量)可能会发生变化。

需要注意的重要一点是,QUIC 旨在成为一种通用协议； HTTP 只是它的一种用途。尽管 HTTP 目前是 QUIC 的主要用例,也是工作组目前关注的重点,但该协议在设计时考虑了其他潜在用例。

9.2.5 HTTP/2 和 QUIC 的区别

QUIC 建立在 HTTP/2 之上,因此当 QUIC 成为标准并且使用范围超出 Google 服务器和浏览器时,您在本书中学到的许多核心概念将对您大有裨益。但是,存在一些关键差异,包括底层 UDP 协议。以下部分讨论其他差异。

QUIC和HTTPS

HTTPS 内置于 QUIC 中,与 HTTP/2 不同,它不会使 QUIC 可用于未加密的 HTTP 连接。出于与 HTTP/2 只能通过 HTTPS 进行 Web 浏览（参见第 3 章)相同的实际和意识形态原因,做出了这种选择。

在实践方面,加密数据可确保不熟悉协议的各方不会无意中干扰协议或对协议做出假设。虽然这种情况现在看起来可能不是问题（没有基础设施应该期望通过 UDP 传输 HTTP 流量）,但它已经给 QUIC 带来了问题,随着 QUIC 的发展,中间件供应商做出的假设不再成立。 49随着协议的发展, 防止在 TCP 下经历僵化将变得更加重要,假设由检查 TCP 流量的中间盒做出。 QUIC 旨在尽可能多地加密。一项允许单个未加密位以允许中间盒监控流量的提议 50 引起了极大的震惊,51并且在撰写本文时,尚未得出确定的结论（尽管该提议作为工作草案包含在内,如本章前面所述） .

建立快速连接HTTP/2 建立了几种协

商 HTTP/2 协议的方法,包括 ALPN、升级标头、先验知识和 Alt-Svc HTTP 标头或 HTTP/2 帧。然而,所有这些方法最初都假定使用 TCP。因为 QUIC 基于 UDP,所以连接到 Web 服务器的 Web 浏览器必须在 TCP 上启动连接并升级到 QUIC。52这个过程引入了对 HTTP over TCP 的依赖,因此否定了 QUIC 的一个关键优势（显着减少连接化成立时间）。备选方案包括同时尝试 TCP 和 UDP 或接受

---

[49] https://www.youtube.com/watch?v=BazWPeUGS8M&feature=youtu.be&t=2216
[50] https://datatracker.ietf.org/doc/draft-ietf-quic-spin-exp/ https://
[51] news.ycombinator .com/item?id=16695816 https://tools.ietf.org/html/draft-ietf-
[52] quic-http-12#section-2.1

Machine Translated by Google

初始性能受到影响,也许下次服务器使用 QUIC 时会记得。
无论如何,ALPN 和 Alt-Svc 标识符h3将被注册用于 HTTP/3（注意:在 HTTP/3 名称达成一致之前,这最初是用于 HTTP over QUIC 的hq ）。
此标识符在标准化后应仅用于官方 iQUIC;当前的 gQUIC 实现不应使用此保留值。 53

QPACK

HPACK,用于报头压缩,依赖于TCP的保证性质,保证HTTP报头帧按顺序接收,这样就可以在两端正确维护动态表,如图9.20所示。

HPACK静态表

指标值	标头 姓名	标头值
1个	:权威	
2个	:方法	得到
3个	:方法	邮政
4个	:小路	/
.....	...	...
58	用户代理	
.....	...	...
61	www-认证	

请求 1

标头	标头值
:方法	得到
:权威	www.example.com
:小路	/
用户代理	铭69

压缩请求 1

标头	标头值
索引 2	
带索引的文字索引 24	www.example.com
索引 4	
带索引的文字索引 56	铭69

请求 1 后的动态表

指标值	标头 姓名	标头值
62	用户代理	铭62
63	:权威	www.example.com

请求 2

标头	标头值
:方法	得到
:权威	www.example.com
:小路	/样式.css
用户代理	铭69

压缩请求 2

标头	标头值
索引 2	
索引 63	
没有索引的文字索引 4	/样式.css
索引 62	

请求 2 后的动态表

指标值	标头 姓名	标头值
62	用户代理	铭62
63	:权威	www.example.com

图 9.20 HPACK 压缩示例

[53] https://github.com/w3c/navigation-timing/issues/71

Machine Translated by Google

请求 2 使用请求 1 中定义的标头索引（62 和 63）。如果请求 1 的部分丢失，以至于无法完整读取头部，则无法知道动态表的状态，因此在接收到丢失的数据包之前无法处理请求 2，as, oth否则，可能会使用不正确的引用。 QUIC 旨在消除在连接级别保证有序数据包传输的需要，以允许独立处理流，但 HPACK 仍然需要这种保证（至少对于标头帧），重新引入 HOL 阻塞，这正是它正在尝试的问题解决。

因此，HTTP/3 需要 HPACK 的变体，称为 QPACK（出于显而易见的原因）。这种变体很复杂，在撰写本文时仍在定义中，但它似乎引入了公认标头的概念。如果发送方需要使用未确认的标头，它可以使用它（并冒着在该流上被阻塞的风险)或者可以发送带有文字的标头（以降低该标头值的压缩效率为代价防止阻塞）。

QPACK 引入了一些其他的变化。一位定义是使用静态表还是动态表（而不是根据 HPACK 明确地从 61 开始计数）。此外，可以更轻松有效地复制标头，以允许关键标头（例如:authority和user-agent)保持在动态表的顶部附近，并以更少的位传输。

其他差异QUIC 使用

的帧和流还有一些其他变化。54一些传输层协议的帧从 HTTP/3 层（例如PING和WINDOW_UPDATE帧）中删除并移至核心 QUIC-Transport层，它不是特定于 HTTP 的（这是有道理的，因为这些帧很可能用于 QUIC 上的非 HTTP 协议）。此外，在 HTTP/2 中很少使用的CONTINUATION框架已从 HTTP/3 中删除。matting 的变化也有一些框架，但由于协议在撰写本文时仍在不断发展，我不会在这里讨论它们。从概念上讲，几乎所有的 HTTP/2 都保留了一种或另一种格式，当 QUIC 和 HTTP/3 正式标准化并可用于客户端和服务器实现时，已经做到这一点的读者将在 QUIC 和 HTTP/3 方面打下良好的基础。

9.2.6 QUIC 工具

由于 QUIC 尚未标准化，因此只有 gQUIC 在野外可用，尽管许多开发人员正在研究 iQUIC 实现。55通常，用于查看 QUIC 的最佳工具是连接到 Google 服务器的 Chrome。可以使用类似于 HTTP/2（请参阅第 4.3.1 节)的网络导出页面。单击 QUIC 会话时，您会看到如图 9.21 所示的屏幕。

---

[54] https://github.com/quicwg/wg-materials/blob/master/interim-18-06/HTTP.pdf https://
[55] github.com/quicwg/base-drafts/wiki/Implementations

Machine Translated by Google

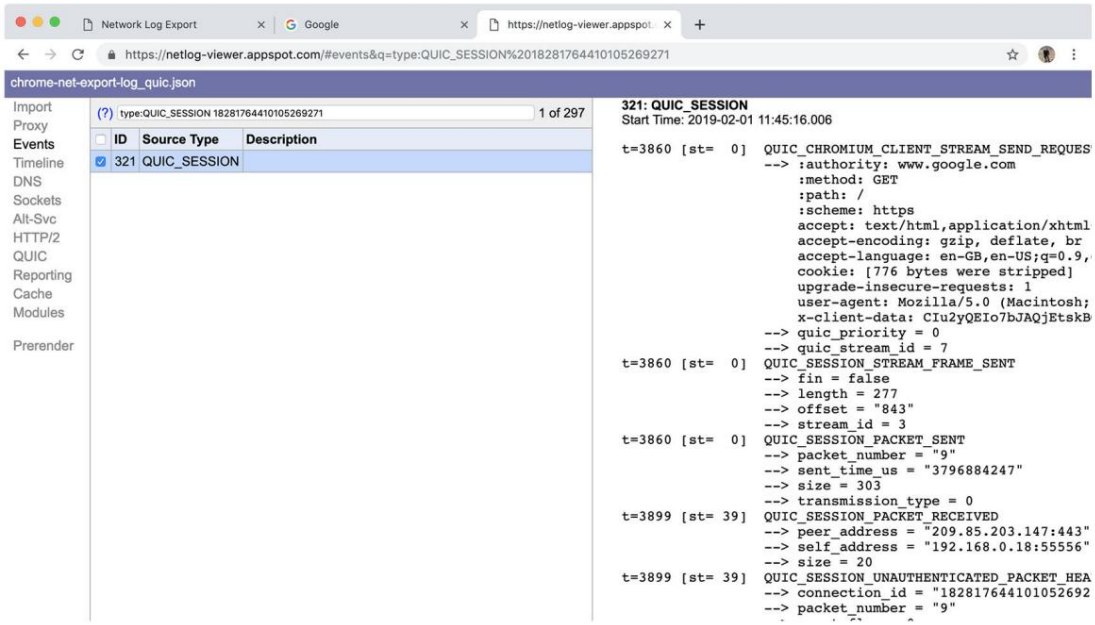


图 9.21 从 Chrome 查看 QUIC 数据

Wireshark 等其他工具对 gQUIC 有一定的支持,如图 9.22 所示。

因为 gQUIC 不是标准化的并且仍在被谷歌改变,所以它需要跟上谷歌所做的任何改变。根据我的经验,由于这个原因,您可能会发现非 Google 工具无法读取格式错误的数据包或加密的有效负载。

9.2.7 QUIC 实现

如果你想实现一个 QUIC 服务器,情况是相似的。 Caddy 有一个基于 Go 编程语言中 QUIC 实现的 gQUIC 实现,但在撰写本文时,该实现在当前版本中已被关闭。 56它可以通过从源代码编译 Caddy 获得,并且应该使它成为下一个版本。 Go 版本 57通常保持最新,因此如果您下载最新版本,Chrome 应该能够使用 gQUIC 对其进行通信。类似地,LiteSpeed 从 201758 年 6 月开始就有了 QUIC 实现,并且一直保持最新的实现,但是开源版本还不支持它,所以它不是一个很好的试验 QUIC 的工具,除非你已经使用 LiteSpeed。

LiteSpeed 还开源了一个可能有用的 QUIC 客户端59。最近,

[56] https://github.com/mholt/caddy/issues/2190
[57] https://github.com/lucas-clemente/quic-go
[58] https://blog.litespeedtech.com/2017/06/26/litespeed-is-由 quic 提供支持/
[59] https://github.com/litespeedtech/lsquic-client

Machine Translated by Google

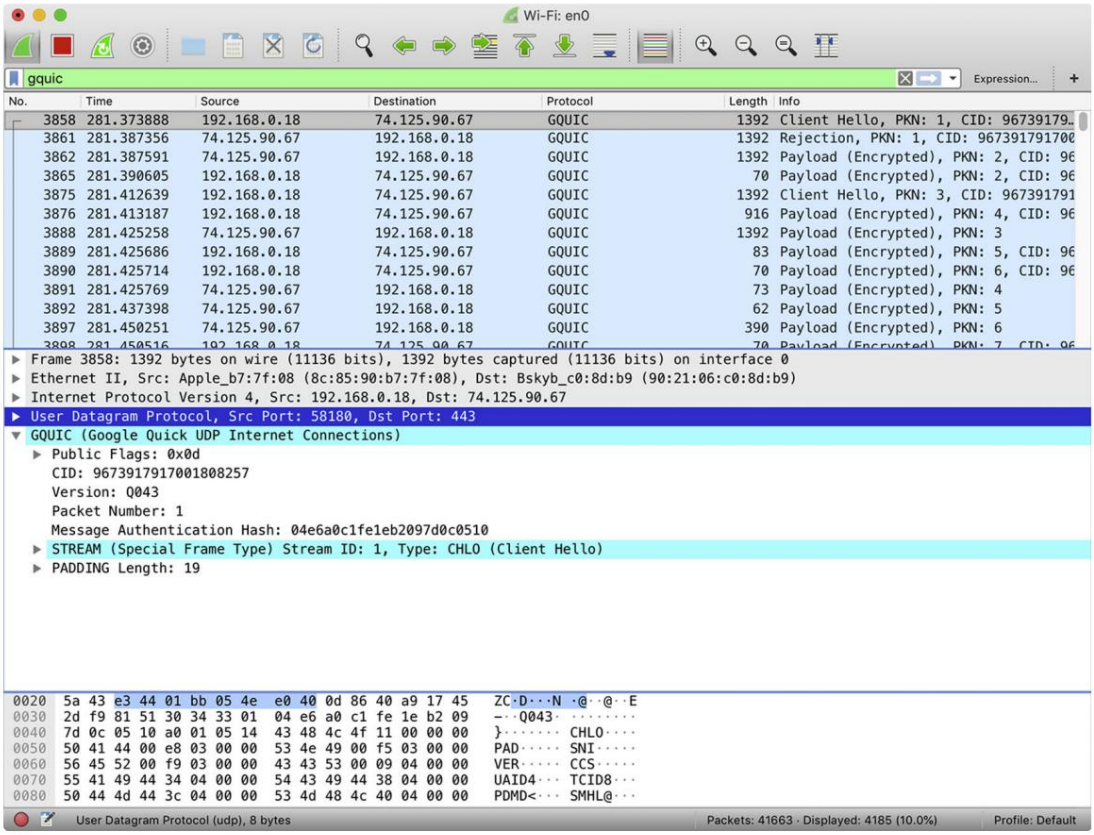


图 9.22 Wireshark 中的 gQUIC

Akamai 于 2018 年 5 月宣布在其内容交付网络平台上支持 gQUIC ,[60]并且在 2018 年 6 月,谷歌宣布为其谷歌云平台负载平衡器提供 gQUIC 支持,[61]因此使用该平台的人可以直接从马口中获得 gQUIC,可以这么说.

9.2.8 你应该使用 QUIC 吗?

与 SPDY 不同,gQUIC 还没有被更广泛的社区所接受,现在 iQUIC 正在标准化,这似乎不太可能发生。在这一点上,除非使用谷歌云平台,否则很难推荐 QUIC。对于任何想要试验 QUIC 的人来说,Go 可能是最好的选择,但它可能不应该用于浏览器的生产环境。 Chrome 中的浏览器实现

---

[60] https://community.akamai.com/customers/s/article/FAQ-QUIC-Native-Platform-Support-for-Media-Delivery Products?language=en_US https://cloudplatform.googleblog.com/2018/06/介绍-QUIC-support-for-HTTPS-load-balancing.html

[61]

Machine Translated by Google

可能会发生很大变化,并且 Chrome 在推出新版本后会迅速关闭浏览器中旧版本的 gQUIC。

在 iQUIC 标准化之后,我预计会出现更多的实现。与 SPDY 相比,现阶段标准化的生产实施较少。

我怀疑 QUIC 和 HTTP/3 的推出将比 HTTP/2 的推出花费更长的时间,因为它是一个更大的变化并且因为它使用 UDP 而不是 TCP。 QUIC 是未来值得关注的协议,在标准化几年后,我希望开发人员能够像现在使用 HTTP/2 一样,使用量迅速增加,甚至最终成为网络领域的主要参与者。 QUIC 对大量网络流量的采用可能会很快发生,少数玩家（如谷歌)和 CDN 服务于大部分流量,但较小的公司和服务器的长尾可能会保留在较旧的 TCP 和 HTTP/2 上（甚至 HTTP/1.1) 堆栈一段时间。

概括

当前的 HTTP 网络堆栈在 TCP 和 HTTPS 层。

由于TCP连接建立和谨慎的拥塞控制,TCP连接达到最大容量需要时间,而HTTPS握手增加了更多时间。 存在解决这些低效率问题的创新,但特别是在 TCP 方面 ular,他们推出的速度很慢。

QUIC 是建立在 UDP 之上的新协议。 通过使用 UDP,QUIC 旨在比 TCP 更快地进行创新。 QUIC 建立在 HTTP/2 之上,并使用许多相同的概念和额外的创新。

QUIC 不仅仅适用于 HTTP;它也可以用于其他协议未来。

基于 QUIC 的 HTTP 将被称为 HTTP/3。 QUIC 有两个版本:Google Quic (gQUIC) 以有限的方式提供但尚未标准化,以及 IETF QUIC (iQUIC),目前正在标准化。

gQUIC 有望在获得批准后被 iQUIC 取代,就像 SPDY 取代 HTTP/2 一样。

Machine Translated by Google

10 HTTP 从何而来

本章涵盖 HTTP/2 的争议
HTTP/2 自推出以来的使用将
HTTP 扩展到 HTTP/2 之外
HTTP 作为更通用的传输层

HTTP/2 规范于 2015 年 5 月正式获得批准,距 HTTP/1.0 推出并迅速被 HTTP/1.1 取代近 20 年。在这段时间里,互联网已经成为每个人生活中不可或缺的一部分,而 HTTP/1.1 的长盛不衰足以说明该协议。然而,很长一段时间以来,该协议停滞不前,推动它向前发展的尝试都失败了,1仅限于更准确地记录 HTTP/1.1,或通过 HTTP 标头添加有限的新功能。

既然 HTTP/2 已问世并迅速在互联网上推广,2那么 HTTP 将何去何从? HTTP/2 在现实世界中的表现如何? HTTP 的主要问题现在解决了吗?下一场大满贯赛会是 20 年后吗?

[1] https://www.w3.org/Protocols/HTTP-NG/Activity.html
[2] https://w3techs.com/technologies/details/ce-http2/all/all

Machine Translated by Google

协议的创新,还是互联网创新的新阶段导致了更大的变化?本章试图回答这些问题并对 HTTP 将如何发展做出一些有根据的猜测。

10.1 HTTP/2 的争议以及它没有解决的问题

HTTP/2 并非没有争议,在整个标准化过程中有不少人表达了担忧,尤其是在它接近结论和批准时。人们提出了许多争论,其中一些争论比其他争论更激烈。3争论认为 SPDY 不应用作 HTTP/2 的基础,并且 HTTP 的隐私问题没有得到解决。此外,有人支持和反对在协议中强制执行加密。我将在以下各节中讨论这些以及许多其他争议点。

许多批评在 Internet 工程任务组的HTTP 工作组 (HTTP-WG)邮件列表中进行了讨论,4 并且由更广泛的 Internet 社区在诸如 HackerNews、 5、6、7 SlashDot、8和 The Register.9 提出了许多反驳意见。

既然该协议已经在现实世界中出现了一段时间,并且正在讨论 HTTP 的未来,那么有必要重新审视这些论点,看看哪些论点仍然正确,以及开发人员在考虑 HTTP 的下一次迭代时可以从中学到什么。

10.1.1 反对 SPDY 的论据

HTTP/2 很大程度上基于 SPDY 协议,谷歌在现实世界中证明了这一点。
SPDY 是一个实用的升级,可以在互联网上实施和部署。 SPDY 的成功促使 IETF 着眼于升级 HTTP,10尽管 IETF 没有单独致力于 SPDY,但该协议很可能是 HTTP/2 的基础。许多人抱怨没有适当考虑 HTTP/2 应该是什么,而是只关注 SPDY 如何成为新版本。

SPDY是HTTP/2的唯一实际选择吗?

HTTP 工作组的章程11对 HTTP/2 作了如下规定:

> 在保留 HTTP 语义但没有遗留 HTTP/1.x 消息框架和语法的协议中出现了新的实现经验和兴趣,这已被确定为阻碍性能并鼓励滥用底层传输。

---

[3] https://lists.w3.org/Archives/Public/ietf-http-wg/2015JanMar/0043.html
[4] https://lists.w3.org/Archives/Public/ietf-http-wg/ https://
[5] news .ycombinator.com/item?id=8850059 https://news.ycombinator.com/
[6] item?id=9022470 https://news.ycombinator.com/item?id=9066379 https://
[7] tech.slashdot.org/故事/15/01/09/0118226/http2---the-ietf-is-phoning-it-in
[8] https://www.theregister.co.uk/2015/02/18/http2_specification_approved/ https://
[9] lists .w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html https://
[10] datatracker.ietf.org/wg/httpbis/charter/
[11]

Machine Translated by Google

工作组将制定 HTTP 当前语义在有序双向流中的新表达规范。与 HTTP/1.x 一样,主要目标传输是 TCP,但应该可以使用其他传输。

工作将开始使用 draft-mbelshe-httpbis-spdy-00 作为起点。

该声明毫无疑问地表明 SPDY 将构成 HTTP/2 的基础。尽管许多人对 SPDY 的成功印象深刻,而且很少有人怀疑它是对 HTTP/1 的一个很好的改进,但许多人认为 IETF 应该更广泛地了解如何整体改进 HTTP,而不是对这种设计进行橡皮图章。提出了一个任意的两年时间表,12一些人认为满足这样一个时间表强制执行 SPDY 是唯一可行的选择。

最初考虑了另外两个提案:Microsoft 的 HTTP 速度和移动性提案13（基于 SPDY 和 WebSockets14）和网络友好 HTTP 升级。15这两个提案在很多方面都与 SPDY 相似（考虑到 HTTP/2 所依据的参数,这不足为奇被定义)并专注于添加二进制框架层和 HTTP 标头改进。

然而,SPDY 被证明在野外有效,而且不仅在谷歌。许多常见的 Web 服务器和 Web 浏览器都支持 SPDY,并且许多站点已经迁移到 SPDY 或正在迁移到它。 Facebook对支持 SPDY 并拒绝其他两个备选方案的所有三项提案 16 进行了早期分析。尽管 HTTP/2 工作组在标准化过程中对其进行了更改和改进,但 SPDY 成为基础是有意义的。 HTTP/2 不是 SPDY,也不与其有线兼容,尽管它有很多共同点。

主要的担忧是错过了超越 SPDY 的机会。
SPDY 旨在解决 HTTP/1 的一个主要问题　性能　尽管它很好地完成了这项任务,但它没有解决有关 HTTP 的其他问题,例如 cookie。鉴于之前的升级（例如 HTTP-NG)都失败了,这在很大程度上是因为范围变得太大,并且没有实用的方法将其引入野外,因此推进一个实用的实现并不是一件坏事。 SPDY 是升级 HTTP 的最初动力,如果协议的下一个版本没有大量基于它,我们今天可能仍然使用 HTTP/1.1。

间谍和谷歌

另一个令人担忧的领域是 SPDY 主要是一家公司的结果:谷歌。尽管其他大大小小的网站也使用 SPDY（包括 Yahoo!、Twitter 和 Facebook）,但它由 Google 拥有和定义。一些人表示担心谷歌已经在网络上拥有强大的影响力,正在推动自己的

---

[12] https://lists.w3.org/Archives/Public/ietf-http-wg/2012OctDec/0003.html
[13] https://tools.ietf.org/html/draft-montenegro-httpbis-speed-mobility https://
[14] blogs.msdn.microsoft.com/interoperability/2012/03/25/speed-and-mobility-an-approach-for-http 2-0-to-make-mobile-apps-and-the-web-faster/ https //tools.ietf.org/html/draft-taarreau-httpbis-network-friendly https://
[15] lists.w3.org/Archives/Public/ietf-http-wg/2012JulSep/0251.html
[16]

Machine Translated by Google

协议,因此,它在更大的网络社区上的议程。对谷歌的一些不信任是由于它在互联网上的主导地位,因为人们不信任所有在其领域拥有大量影响力的公司。这种情况对谷歌来说尤其如此,因为它通过网络广告赚取了大量收入,这对隐私和跟踪有影响。 HTTP/2 因未尝试解决许多人认为是该协议更大问题的隐私问题而著名。然而,我不认为忽视 HTTP 的隐私问题是邪恶的,并且理解谷歌将性能视为优先解决的问题。

然而,这种论点忽略了将 SPDY 标准化为 HTTP/2 的关键价值:消除对单一公司的依赖,同时允许整个网络社区和主要互联网标准社区 (IEFT) 审查和改进协议。谷歌仍然是互联网上的主要创新者之一,并在网络标准方面创造了许多其他进步,其中一些（如 QUIC;见第 9 章)将在本书中讨论。如果这些创新是有益的,那么忽视谷歌的创新或试图绕过它们似乎是无礼的。

10.1.2 HTTP 中的隐私问题和状态

HTTP/2 的另一个争议是隐私,尤其是关于 HTTP cookie。
Cookie 通常被认为是 HTTP 的最大问题之一,因为它们涉及安全和隐私问题。 HTTP 被设计为无状态协议,在大多数情况下它仍然是无状态协议,即使在 HTTP/2 下也是如此。您向服务器发出的任何请求在理论上与任何先前或未来的请求无关。

然而,现实是现代应用程序和网站需要状态。当您在亚马逊购物时将商品添加到您的购物篮中时,您不想一直添加它。当您登录网上银行时,您不希望每次后续操作都必须重新登录（至少在短时间内）。因此,由于 HTTP 无法将请求与其他请求相关联,因此需要向协议添加状态,并且由于连接并不总是持久存在或重用,因此无法在连接级别完成。

可以使用各种方法来解决 HTTP 中的这种状态困境。网址可以是

添加了会话 ID 参数（例如http://www.example.com?SESSIONID= 12345）,但由于使用会话 ID 共享或标记 URL,这些参数很难看、令人困惑并且充满安全风险。 HTTP cookie17被认为是答案。 Cookie 是存储在浏览器上的小块信息,由浏览器随每个后续请求发送。有了 cookie,就有可能拥有一个会话标识符或其他适用于 HTTP 的首选项或设置。 HTTP cookie 最近名声不好,但 HTTP cookie 不好也不坏,替代品（如 URL 参数)往往会带来更大的问题。

HTTP cookie 被视为不良的原因有很多,其中包括:

[17] https://tools.ietf.org/html/rfc6265

Machine Translated by Google

它们允许出于广告目的（或更糟！）进行跟踪。 默认情况下，它们是不安全的。 它们随每个请求一起发送。

Cookie 和第三方追踪

Cookie 不仅可以由正在加载的网站使用，还可以由网络浏览器为该网站加载的任何资源使用。当网站（例如 www.example.com ）从广告网站（例如adwords.google.com)加载内容并设置可在其他网站（例如 www.example2.com）。该 cookie 还参考了第三方广告网站 (adwords.google.com)，如图 10.1 所示。

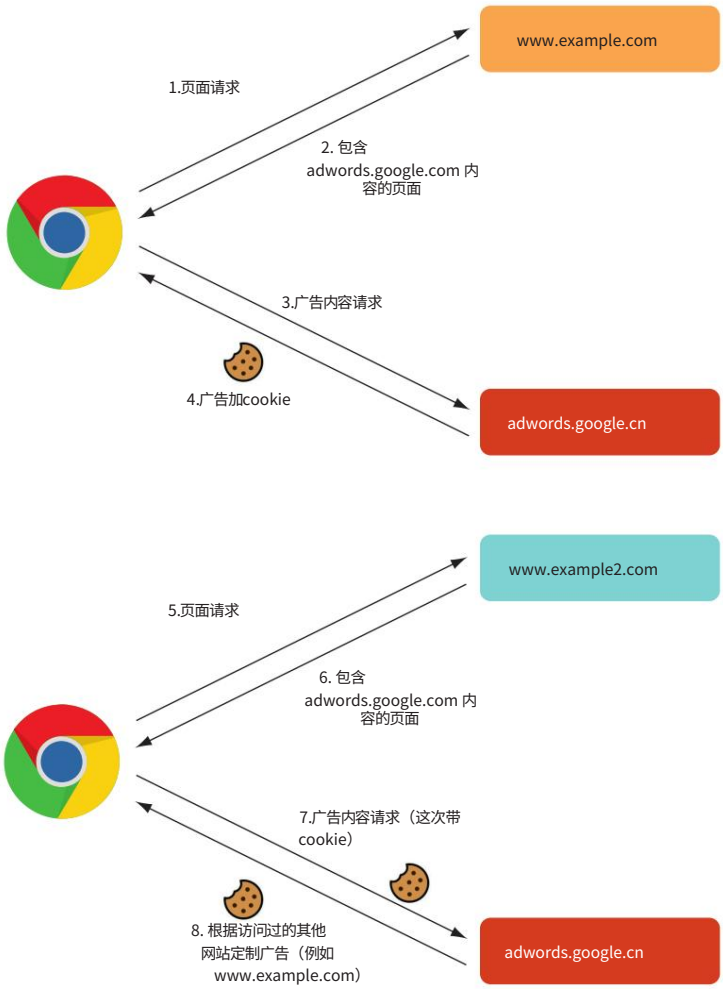


图 10.1 第三方 cookie 如何允许跨站点跟踪

Machine Translated by Google

可以根据用户的浏览历史在后续网站上显示相关广告,但通常是在用户不知情的情况下。因此,欧盟 (EU) 实施了所谓的 cookie 法,根据该法,网站必须告知用户他们正在使用 cookie,如图 10.2 所示。

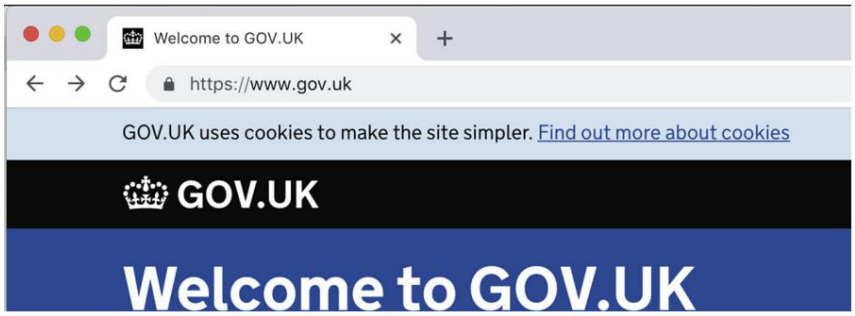


图 10.2 英国政府网站上的 Cookie 横幅

Web 用户,尤其是在欧盟,不得不在他们访问的每个网站上单击以隐藏相当无意义的“此网站正在使用 cookie”消息,因为大多数网站都在使用 cookie。通用数据保护条例 (GDPR) 中的一项更严格的要求于 2018 年生效,并导致屏幕上出现了关于 cookie 的更大警告。许多用户将此警告视为另一个弹出窗口,可以点击进入他们想要的内容,但往往没有意识到 GDPR 是一项旨在让消费者控制其数据的亲消费者立法。

HTTP Cookie 和安全

另一个问题是默认情况下 HTTP cookie 是不安全的。尽管添加了选项以使其更安全（例如Secure[18]和HttpOnly[19]标志）,但这些选项要求程序员在创建 cookie 时明确设置它们。即便如此,程序员也不能总是阻止这些 cookie 被不安全的 cookie 覆盖,所以这个解决方案只是部分解决方案。在撰写本文时的研究表明,大约 8% 的 cookie 设置了这些属性,[20]表明绝大多数 cookie 使用不安全的默认值。

这种情况很糟糕,因为存储会话标识符的 cookie 提供了完全访问权限

到一个帐户,所以得到一个 cookie 几乎等同于拥有用户名和密码。出于这个原因,您会认为 cookie 会受到限制（并且它们可以被限制）,但默认情况下,它们不是。

除非在创建 cookie 时使用Secure属性,否则站点的 cookie 会在 HTTP 和 HTTPS 请求上发送。由于性能成本

[18] https://tools.ietf.org/html/rfc6265#section-5.2.5
[19] https://tools.ietf.org/html/rfc6265#section-5.2.6
[20] https://github.com/mikewest/http-状态令牌#a-problem

Machine Translated by Google

HTTPS（这在大多数情况下不再是问题），只有一些登录或签出页面常用 HTTPS,站点的其余部分仅在 HTTP 中可用,但这种安排会暴露未明确设置的 cookie通过不安全的 HTTP 流量发送（并可能读取)的安全属性。类似地,页面上加载的任何 JavaScript 片段（例如使页面看起来不错的便捷脚本、添加评论系统或在页面上放置一个很酷的小部件)都可以访问 cookie,除非在cookie被创建。

即使您使用了所有这些保护措施,安全的HttpOnly cookie 也有可能被可以伪造 HTTP 请求的恶意用户覆盖。已提出Cookie 前缀21来防止此问题,例如,要求名称以__Secure开头的 cookie具有Secure属性。同样,此解决方案需要网站所有者（使用)和浏览器（执行)的支持。

HTTP COOKIES 与每个请求一起发送 Cookie 与

每个后续请求一起发送　在简单性方面是一种祝福,但在许多其他方面却是一种诅咒。虽然向您的网上银行服务请求检查您的余额或移动资金应该需要发送会话信息,但对徽标或其他静态资源的请求不应该需要此信息,但浏览器还是会愉快地发送它们。这种情况具有安全隐患,尤其是对于跨站点请求伪造攻击,22每次请求都会发送不必要的数据,从而存在泄露信息的风险。已提议使用SameSite属性23 来防止包含 cookie 的跨站点请求,但此属性不是默认属性,需要理解才能实现。

其他类型的隐私跟踪方法通常称为 cookie,即使它们不是 HTTP cookie。在 Flash 而不是 HTTP 或超级 cookie中实现的 Flash cookie使用其他指纹识别方法24来跟踪用户,也可用于实现僵尸 cookie,即使在 cookie 被清除后也会返回。所有这些变化进一步损害了 cookie 的声誉,尤其是当它们规避了浏览器提供的允许用户管理 cookie 的任何控件时。

HTTP/2应该解决HTTP Cookie 的问题吗?

尽管 HTTP cookie 因其隐私和安全隐患而被视为消极的,但尚未提出可行的替代方案。 Cookie 并非天生邪恶或危险;只有使用它们才可以。允许将状态添加到 HTTP 的替代方案（例如 URL 参数和本地存储）25具有类似的缺陷,并且给用户的控制较少,并且它们没有获得 cookie 所具有的隐私和安全监督。

---

[21] https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis#section-4.1.3
[22] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
[23] https://tools .ietf.org/html/draft-ietf-httpbis-rfc6265bis#section-5.3.7
[24] https://nakedsecurity.sophos.com/2018/03/20/apple-burns-the-hsts-super-cookie/
[25] https: //dev.to/rdegges/please-stop-using-local-storage-1i04

Machine Translated by Google

有人说 HTTP/2 应该用状态来解决这个问题,并实现一个更安全、隐私侵犯更少的解决方案。尽管经常尝试解决这个问题,但没有人提出比 cookie 更好的解决方案。26任何此类解决方案都需要支持传统的 HTTP cookie,否则就有可能无法获得任何支持。 HTTP 是无状态的,虽然 HTTP/2 在网络层面增加了一些状态的概念（比如流状态和 HPACK 动态表状态）,但在应用层面它仍然是无状态的。在一个 HTTP/2 连接上发送的 HTTP 消息可能与在二进制级别的另一个 HTTP/2 连接上发送的消息略有不同（例如,由于标头压缩）,但它仍然是相同的状态级应用。

正如本章前面所述,cookie 已经有了一些创新,也许 HTTP/2 可以强制执行它们,而不是像 HTTP/1 那样使它们成为可选的。但强制执行会给采用造成障碍,这就是为什么 HTTP/2 章程表达了保留 HTTP/1 语义并仅更改传输层的愿望。

将来,开发人员可能希望为 HTTP 实现比 cookie 更好的状态管理系统。但没有理由阻止下一个版本

HTTP,因为当前版本的所有问题都无法解决。

10.1.3 HTTP 和加密

与状态一样,加密最初并不是 HTTP 的设计原则;它是在事后使用 HTTPS 添加的。 HTTPS 在发送之前以加密形式包装常规 HTTP 消息,并在它们到达后解包,并且在大多数情况下,它是有效的。以前,人们担心 SSL/TLS 证书的成本以及加密和解密的性能,但这些成本问题大多已通过廉价（甚至免费)27证书设计解决,并且随着计算能力的提高,28尽管如第 9 章所述,初始连接仍然会导致性能下降。

应该加密所有HTTP吗?

对 HTTPS 的主要担忧是初始设置和管理的复杂性、对第三方证书颁发机构 (CA) 提供认可证书的依赖,29以及它并非无处不在的事实。最后一点是 HTTP/2 标准化时争论的焦点。许多人认为下一版本的 HTTP 应该只提供安全版本 (HTTPS),不安全的 HTTP 世界应该留在过去。其他人强烈认为在许多情况下加密是不必要的,并且不应在协议中强制执行。具有讽刺意味的是,这些人通常是抱怨 HTTP/2 没有通过处理 cookie 来充分改善隐私的人。

---

[26] 例如： https: //github.com/mikewest/http-state-tokens https://
[27] letsencrypt.org/ https://istlsfastyet.com/ https://
[28] www.howtogeek.com/182425/5-serious- https-and-ssl-security-
[29] on-the-web 问题/

Machine Translated by Google

机会加密

完全 HTTPS 加密需要加密和身份验证（通过使用第三方 CA)的替代方法是HTTP URL 的机会加密。这种技术对传输中的数据进行加密,但不像 HTTPS 提供的那样对身份验证提供任何保证 用户正在与实际网站交谈,而不是与假装是该网站的人交谈。机会加密比 HTTP 更进一步,但不如 HTTPS 强大;但是,它可以在协议级别部署,无需使用第三方 CA 或网站所有者的任何努力。

HTTP/2和加密最后,经过

多次争论,没有达成共识,HTTP/2 发布了同时使用加密 (h2) 和未加密 (h2c) 连接的能力。不包括使用机会加密的中间立场。

现实有点片面。如第 3 章所述,所有 Web 浏览器（HTTP 的主要客户端)决定仅通过 HTTPS (h2) 实现 HTTP/2。这样做是出于意识形态原因（主要浏览器供应商表示他们打算转向加密 Web） 30也出于技术原因;通过使用加密会话,可以引入新协议,而无需任何中间件互联网工作基础设施需要了解这种格式。微软是唯一一家表示有兴趣允许使用未加密的 HTTP/2 的浏览器供应商,但最终,它只提供了加密版本,显然是在看到不通过 HTTPS 使用 HTTP/2 时出现互操作性问题之后。

Firefox 允许使用Alt-Svc标头通过替代服务获取网站

这导致一些供应商为 HTTP站点提供 HTTP/2。31 此方法适用于具有 HTTPS 版本但尚未准备好切换到它的站点（例如避免混合内容警告）。 Firefox 可以使用Alt-Svc标头通过 HTTPS（和 HTTP/2)获取站点,但将其显示为 HTTP 站点,这是一种带有扭曲的机会主义加密。

要求 HTTP/2 仅通过 HTTPS 可用可能还为时过早。
HTTPS 的使用在过去几年中大幅增加,但不幸的是,它还远未普及。第 6 章指出,Firefox 超过 70% 的 Web 流量是通过 HTTPS 提供的,但该统计数据因较大的参与者而有所偏差。当你放眼大局时,数字就不那么乐观了。如图 10.3 所示,前 1000 万个网站的 HTTPS 使用率仍略高于 40%32 ,尽管该数字也呈上升趋势。 2015 年,当 HTTP/2 获得批准时,只有大约 5% 的网站默认使用 HTTPS。

可以说,安全性（和性能,就此而言)对于大型网站更为重要,但在 HTTPS 成为运行网站的要求之前还有一段路要走。

---

30 https://blog.mozilla.org/security/2015/04/30/deprecating-non-secure-http/和https://blog.cloudflare.com/opportunistic-encryption-bringing-http-2-to-the -未加密网络/ https://blog.cloudflare.com/opportunistic-encryption-
31 bringing-http-2-to-the-unencrypted-web/ https://w3techs.com/technologies/details/ce-httpsdefault/all /全部
32

Machine Translated by Google

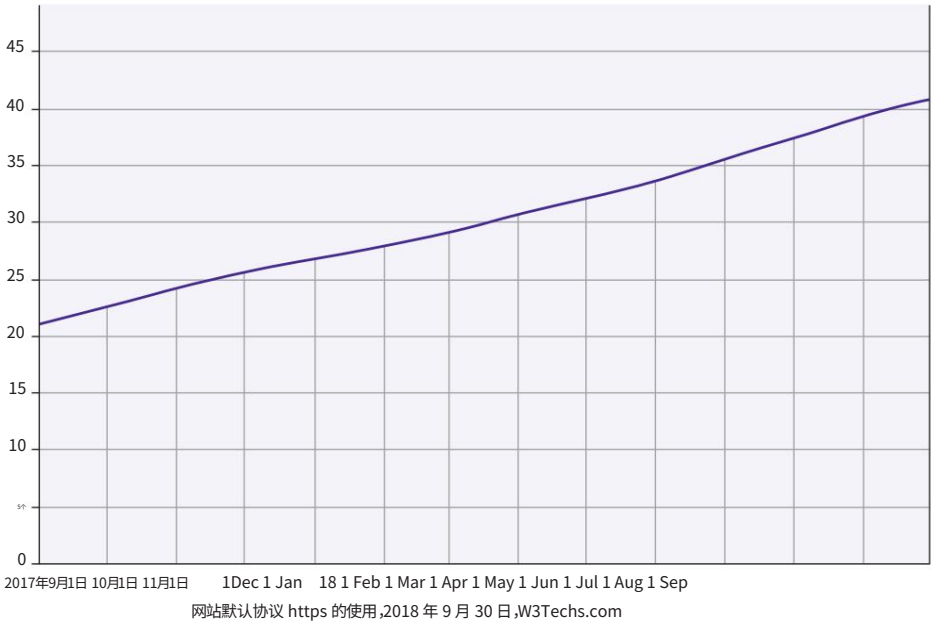


图 10.3 HTTPS 在前 1000 万个网站中的使用情况

仅出于这个原因,在标准化时将 HTTP/2 限制为 HTTPS 可能有点过时了。在使用激励措施鼓励采用良好标准和增加采用的另一个障碍之间需要谨慎平衡。

另一个需要记住的用例是内部的、非面向互联网的流量 内部网站和后端应用程序服务器都以 Web 服务器为前端,其中 HTTPS 被卸载。 Intranet 网站应该像外部网站一样使用 HTTPS 进行保护,但这种做法需要一定程度的承诺和基础设施,而这并不总是存在。互联网现在才转向加密,部分原因是它所依赖的开放网络存在风险。封闭的内部网络通常被认为风险要小得多,因此加密内部网站通常优先级较低。

在 Intranet 站点上使用 HTTPS 可能会滞后 Internet HTTPS 的采用一段时间,特别是对于后端服务器,当前端 Web 服务器用于卸载 HTTPS 要求时。通常认为没有必要在 Web 服务器和后端应用程序服务器之间进行加密,尤其是因为在某些应用程序服务器上管理 HTTPS 证书需要付出额外的努力。

仅限内部的站点通常不能使用商业 CA（尤其是像 Let s Encrypt 这样的自动免费 CA）,除非它们将自己暴露在公共互联网上或使用通配符证书,后者更昂贵且不易自动化。运行内部 CA 通常是答案,但内部 CA 通常不具备允许自动颁发和更新证书的自动化级别。最后,一些 Web 服务器（特别是基于 Java 的具有 Java 密钥存储和 IIS 的服务器)需要额外的步骤

Machine Translated by Google

安装证书,所以即使您可以自动颁发证书,您也需要了解如何在证书即将到期时自动安装和使用证书。

当然,HTTP/2 不是网站的要求;您可以将其视为那些想要提高速度的人的一个选择,而这类高级用户无论如何都可能使用 HTTPS。在 Intranet 站点上,延迟不应该像在公共 Internet 上那样成为一个大问题,因此迁移到 HTTP/2 的好处更少。我在前面的章节中展示了目前不需要在后端服务器上支持 HTTP/2,只要访问者访问的边缘服务器可以通过 HTTP/2 访问即可。无论如何,当很大一部分用户无法使用它时,尝试发明新版本的网络核心协议毫无意义 在撰写本文时,公共互联网高达 60%,在 HTTP 时,这一比例高达 95% /2 已标准化。在私人内部网上,年龄百分比可能要高得多。

还有 HTTP 的增长超出网站和物联网 (IoT)领域的小问题,其中证书和 HTTPS 管理肯定更加复杂。即使在撰写本文时,这个领域也没有得到充分解决,尽管已经提出了一些建议,例如本地的 HTTPS 网络。[33] 尽管物联网设备可能不需要 HTTP/2 的好处,但通过坚持加密来强制它们保持在 HTTP/1 上似乎是落后的,直到这个问题得到解决。

最终,考虑到 HTTP/2 获得批准时的情况,它没有通过仅 HTTPS 的要求进行标准化也就不足为奇了。尽管有充分的理由推动 HTTPS 成为规范,但应谨慎行事,在正确的时间以正确的方式推动它,对于 HTTP/2 这样做还为时过早。 QUIC 决定只推动 HTTPS,这可能更有意义,因为这两种协议之间的格局在三四年间发生了变化,但是许多关于非面向公众的 HTTP 实现的论点仍然站得住脚,而且还没有被接受妥善解决。

10.1.4 传输协议问题

其他投诉是关于传输级别的协议本身。以前,HTTP 并没有过多涉及传输层,只是说它是一个数据流,前面有请求（或响应)行和 HTTP 标头。 HTTP/2 通过引入二进制框架层改变了这种情况。

分层违规

正如在第 1 章中讨论的那样,网络协议通常构建在不同的层中。
HTTP/2 跳出了它的传统层,并具有 TCP 的许多特征。
图 10.4 显示了 Web 堆栈以及它如何（粗略地)映射到一些 OSI 网络工作模型层,尽管有一些重叠。例如,第 6 层不直接映射到任何这些技术,尽管 HTTP 允许文件格式的通信。

[33] https://www.w3.org/community/httpslocal/

Machine Translated by Google

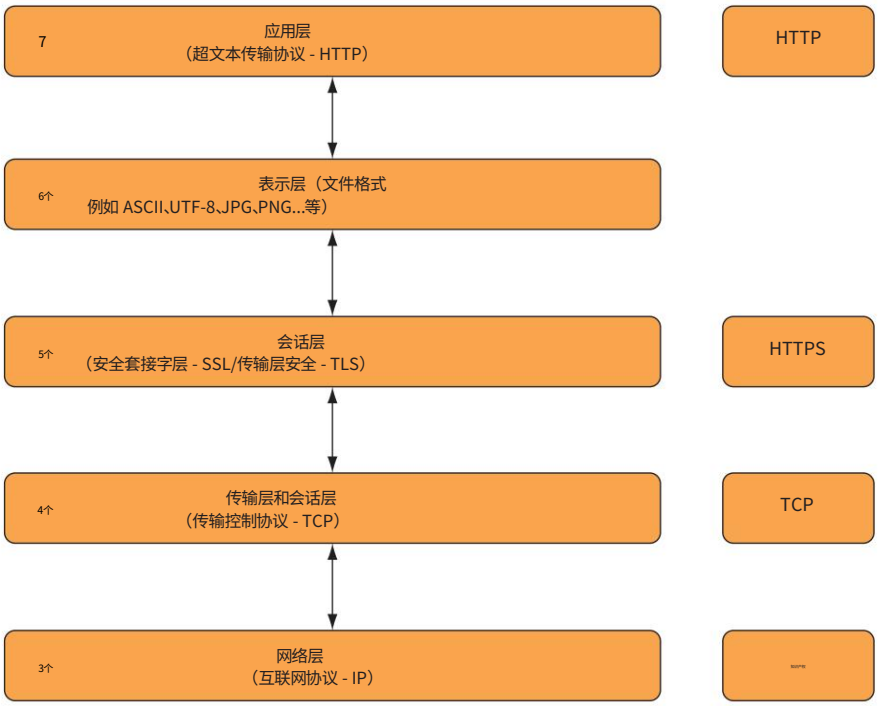


图 10.4 OSI 模型和 Web 网络堆栈

HTTP/2 不再像 HTTP/1 那样只映射到顶层应用程序层;它管理许多传统上被认为是传输层职责的项目,例如多路复用和流量控制。因为 HTTP/2 重复而不是取代 TCP 中对这些概念的处理,它更像是 HTTP/2,补充了传统上被认为是 HTTP 的内容。因此,新的二进制框架层跨越多个层,如图 10.5 所示。

许多人认为这种“分层违规”是个坏主意。协议分层允许在每个级别上进行更简单的实现,并且通常被视为一件好事。尽管分层在现实世界中从来没有明确定义,而且人们可能会过于执着于此（这是不可取的）,[34]但该论点确实有其道理。特别是,TCP 概念的重复虽然仍然受到 TCP 限制的约束,但会导致问题,如第 9 章所述。

QUIC 试图在应用程序传输协议 (HTTP) 和网络传输协议（TCP 和 UDP)之间返回一个更明确的分层,如图 10.6 所示。在此模型中,HTTP/3 映射 HTTP 帧类型并显示它们应如何用于 HTTP 流量,但保留传输层信息

[34] https://tools.ietf.org/html/rfc3439#section-3

Machine Translated by Google

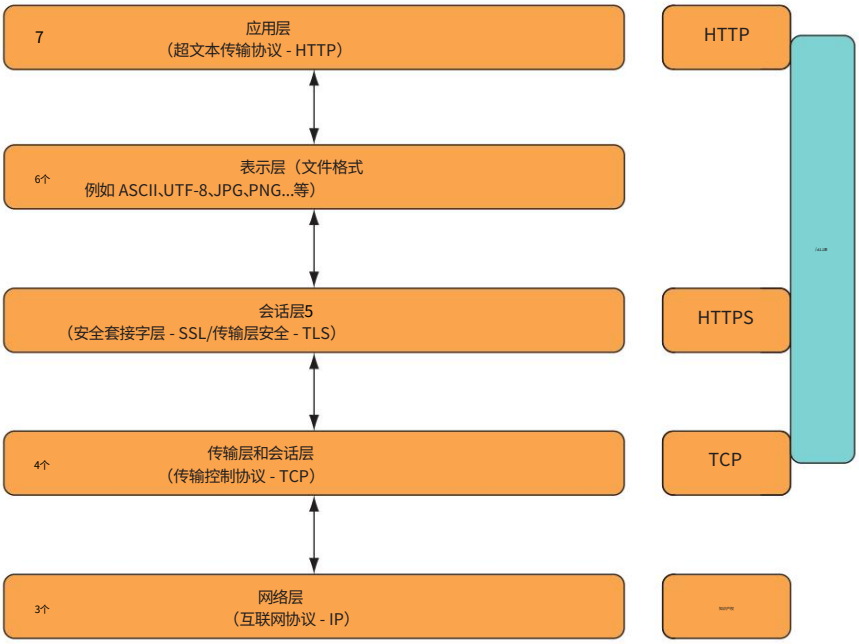


图 10.5 OSI 模型和使用 HTTP/2 的 Web 网络堆栈

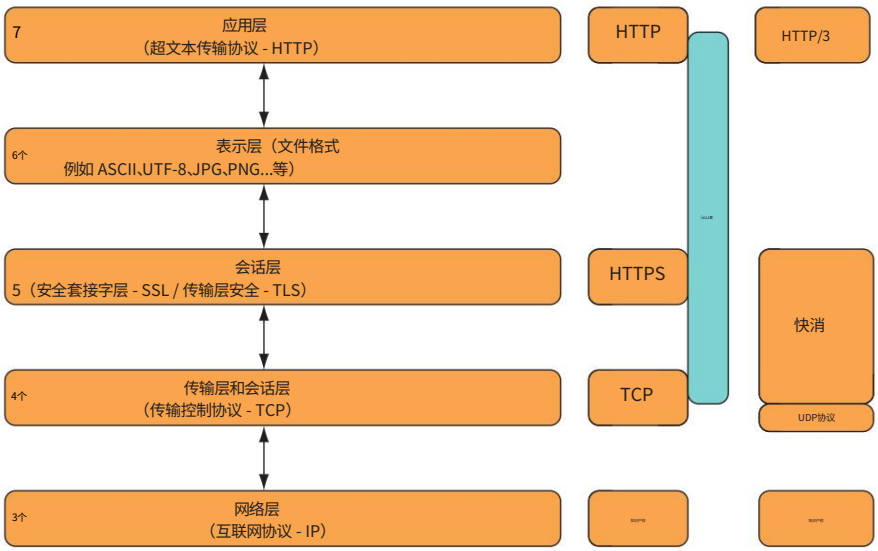


图 10.6 OSI 模型和具有 QUIC 和 HTTP/3 的 Web 网络堆栈

Machine Translated by Google

（例如管理多路复用流)到 QUIC 本身。在其他方面,QUIC 可以通过合并部分 TCP 和 HTTPS 层来进一步模糊层,但这种安排可能是有道理的。 HTTPS 总是大部分与 HTTP 分开,因此它可能更属于会话和传输层而不是应用程序层。

TCP 霍尔阻塞

第 9 章解释了 HTTP/2 多路复用如何解决 HTTP 队头 (HOL) 阻塞问题,并将其推到 TCP 级别。因此,在某些有损条件下,HTTP/2 可能比 HTTP/1 慢。这个问题从一开始就很明显,并且在第一个 HTTP/2 草案35 中被提及（但从后来的草案中删除）,它解释了单个 TCP 连接的好处超过了负面影响:

> 然而,使用多个连接并非没有好处。因为 SPDY 将多个独立的流多路复用到单个流上,所以它可能会在传输级别产生队头阻塞问题。在目前的测试中,线头阻塞的负面影响（尤其是在存在数据包丢失的情况下)被压缩和优先化的好处所抵消。
>
> HTTP/2 规范版本 00

正如您在第 9 章中看到的,QUIC 旨在解决这个问题,但对于大多数连接而言,HTTP/2 已经足够好了。坚持完美的解决方案不需要延迟有用的升级,更不用说从 HTTP/1.1 到 QUIC 和 HTTP/3 的巨大升级了!

HTTP不再是无状态的

另一种说法是,该协议首次将状态引入核心协议 HTTP。本章前面讨论的HTTP cookie允许通过 HTTP 传输状态,但不将状态添加到核心协议本身。协议和流 ID 的多路复用特性（第 4 章）、HTTP/2 状态机（第 7 章)和 HPACK 标头压缩（第 8 章)都将状态的概念添加到以前的无状态协议中,而没有添加一个定义HTTP 用户实际需要的状态:会话状态,如 10.1.2 节所述。虽然这个选择是经过深思熟虑的,但并不像看起来那么矛盾。虽然状态已经在连接级别添加到 HTTP/2,但在整个 HTTP 级别,这无关紧要。相同的 HTTP 请求仍然可以通过单独的 HTTP/2 连接发出,无论是在并行连接中还是在以后的连接中,并且仍然在 HTTP 级别传达相同的语义。添加状态以处理多路复用协议和标头压缩,但不会超出这些目的使用。如果状态以任何方式变得混乱,则可以断开连接并重新建立连接,并且可以通过新连接尝试 HTTP 请求,但存在与 HTTP 中完全相同的问题和限制（例如何时可以安全地重新发送[idempotent]).36 HTTP/2 的有状态部分不多不少

---

[35] https://tools.ietf.org/html/draft-ietf-httpbis-http2-00#section-4.3
[36] https://tools.ietf.org/html/rfc7231#section-4.2.2

Machine Translated by Google

比始终存在的 TCP 有状态部分重要。事实上,添加状态只是为了让 HTTP/2 的流表现得更像 TCP。

HTTP /2的工具 二进

制分层和从纯文本的转移也让一些人感到恼火,因为不能再使用像 Telnet 这样的简单工具;我在第 1 章中使用了它们来展示它们在 HTTP 中的用法。然而,这种抱怨没有什么价值,因为 HTTPS 加密连接也遇到同样的问题,而且工具集已经发展到支持 HTTPS。或许可以提出一个更好的论据,即一些支持 HTTP 的网络设备（例如像 Varnish 这样的 HTTP 缓存)现在需要实现更多的 HTTP 才能完全理解 HTTP 消息。但该论点几乎不支持不改进协议或让基础设施在不完全理解 HTTP 消息的细节的情况下尝试路由 HTTP 流量。

HTTP /2是关于改进HTTP 的吗?

最后,HTTP/2 中的大部分更改都在传输级别,而不是传统上认为的 HTTP 级别。是否在 HTTP 级别进行了足够的改进以保证 HTTP/2 的名称,或者该协议是否应该命名为 HTTP/1.2、TCP/2 或 HTTPS/2? QUIC 在这一领域的进一步工作似乎证实了这种分层或许不应被视为 HTTP 的一部分。

最终,这样的争论是没有成效的。新协议包含重大更改,因此根据任何合理的定义,它都应该与之前的版本区分开来。此外,版本号很便宜并且在许多技术实现中使用得过于谨慎。 20 年后移动到版本 2,因为它包含许多重大更改并且不向后兼容,这并非不合理,实际上是推荐的路径。

10.1.5 HTTP/2 太复杂了

另一个经常提出的论点是 HTTP/2 很复杂。毫无疑问,情况确实如此,尤其是与看似简单的 HTTP/1.1 相比。
与 HTTP/1 相比,框架层、其二进制性质和有状态协议是难以理解的概念,更不用说复杂的优先级排序和标头压缩了。你需要一整本书来理解这些概念!尽管这种情况为技术作者创造了有趣的工作,但它确实是协议的一个问题。 HTTP 成功的主要原因之一是它的简单性。 HTTP的基本细节在HTTP/0.9的一页规范（如果页面可以称为规范)中,虽然加入了HTTP/1.0和1.1,但从概念上讲,HTTP还是很容易理解的。

然而,这还不是全部。尽管 HTTP/1.1 对人类来说很容易理解,但很难用软件实现。需要各种边缘情况和额外处理来确定这种未格式化、非结构化、基于文本的协议的含义。 HTTP/2 似乎要复杂得多,但所有这些复杂性都可以很容易地自动化。这并不是说在实施过程中不会出现网站开发人员不需要的实施错误或细微差别

Machine Translated by Google

虽然工作（这本书的另一个原因），但这种情况对于任何新技术都是如此，对于 HTTP/1 尤其如此。HTTP/2 的复杂性与 HTTP/1 的复杂性一样，主要影响低级实现者，例如网络浏览器和 Web 服务器开发人员，而不是高级 Web 开发人员，并且大多数低级开发人员认为 HTTP/ 2 比 HTTP/1 更易于实现。

也许衡量复杂性是否是一个问题的主要衡量标准是实现的绝对数量。 HTTP/2 主页列出了 80 多个独立的、活跃的实现。37几乎所有常见的 Web 服务器和 Web 浏览器都支持 HTTP/2，并且许多在它上线后不久就支持了。尽管这些实施存在错误和实施问题，这些问题可能会持续一段时间，但采用的速度表明这些问题并不严重。但是请注意，由于所涉及的复杂性，规范的某些部分（例如 HTTP/2 推送和优先级排序）并未出现在每个实现中。

很难不说 HTTP/2 在概念上很复杂，这并不像您想象的那么大。在大多数情况下，简单胜过复杂（Keep It Sim ple Stupid [KISS] 原则），但 HTTP/1 并不像看起来那么简单。尽管有所有关于复杂性无关紧要的争论，但那些发现一些晦涩难懂的 HTTP/2 实现问题并需要几天时间才能解决的人无疑会诅咒这种复杂性，而我们这些经历过并做过这些的人会表示同情！

10.1.6 HTTP/2 是权宜之计

HTTP/2 并未尝试解决 HTTP 的所有问题。经过多年的停滞，以及经过验证的 SPDY 用例，IETF HTTP-WG 急于避免陷入兔子洞，因此许多问题没有完全解决，并且 HTTP/2 在某些问题上没有达成共识就获得批准。这种情况无疑让那些认为某些问题本应在 HTTP/2 中得到解决的人感到沮丧，但这种做法似乎是务实的。

HTTP/2 改进了协议的性能，消除了一些基本的瓶颈，因此很适合在现实世界中推出和使用。没有什么能阻止 HTTP 的未来版本进一步改进此修订版无法实现的功能。事实上，正在提出通过新设置和帧类型改进协议的其他方法。

QUIC 旨在解决一些 HTTP/2 没有解决的问题，例如 TCP HOL 阻塞、更完整的加密、改进的连接建立和连接迁移。将这些更改合并到 HTTP/2 中可能还需要四年时间，但似乎不需要这样的等待。

当 QUIC 被标准化时，它可能需要更长的时间才能实现，因为它甚至比 HTTP/2 更复杂。在标准化过程的相似点上，QUIC 的实现比 HTTP/2 少，这可能是因为将现有的 SPDY 实现迁移到 HTTP/2 实现很容易。但是 gQUIC

[37] https://github.com/http2/http2-spec/wiki/Implementations

Machine Translated by Google

没有以同样的方式被接受。转向 UDP 也将为传统上几乎完全基于 TCP 的互联网带来挑战。考虑到这种复杂性,HTTP/2 是一个权宜之计并且正在取得一些进展这一事实是积极的,而不是消极的。

10.2 现实世界中的 HTTP/2

所有这些争论都是在 HTTP/2 正式标准化之前提出的,但没有一个被认为严重到足以阻止或延迟最终标准化。从那时起,HTTP/2 得到迅速采用,在撰写本文时,超过 30% 的前 1000 万网站支持它38（图 10.7）。

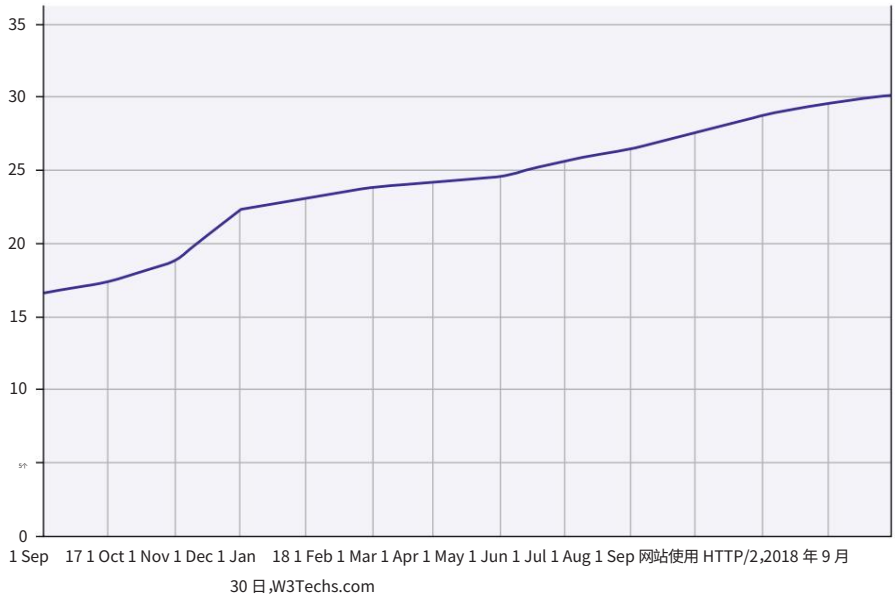


图 10.7 2017 年 9 月至 2018 年 9 月 HTTP/2 增长情况

更好的是,超过 55% 的网络流量使用 HTTP/2,39 ,因为谷歌、YouTube 和 Facebook 等大型网站支持 HTTP/2,并且产生的流量按比例高于小型网站。基于这些统计数据,HTTP/2 已经被证明是一个巨大的成功。

此外,HTTP/2 已经扩展,具有用于增强协议的新设置和帧类型,例如替代服务,40 ORIGIN

[38] https://w3techs.com/technologies/details/ce-http2/all/all
[39] https://telemetry.mozilla.org/new-pipeline/dist.html#!cumulative=0&measure=HTTP_RESPONSE_VERSION
[40] https://tools.ietf .org/html/rfc7838

Machine Translated by Google

frame,41和基于 HTTP/2 的 WebSockets。42其他提案,例如缓存摘要,进一步扩展了 HTTP/2,还有[43]寻求更多的建议正在酝酿之中。

总而言之,HTTP/2 已成为互联网的广泛组成部分,并且在使用和功能方面都在继续增长。本章所讨论的所有问题,虽然并非完全没有道理,但在现实生活中并没有被证明是大问题。

然而,并非一切都取得了成功。到目前为止,尤其是 HTTP/2 推送未能产生任何特别的影响,主要是由于正确使用它的复杂性（在第 5 章中讨论）,尽管缺乏服务器端支持也无济于事。也许事后看来,HTTP/2 推送并不是某些人认为的提高性能的灵丹妙药。或许它应该被排除在最初的 HTTP/2 提案之外,并在必要时作为可选框架添加。有些人甚至呼吁将其从规范中删除,尽管这种行为似乎还为时过早。

此外,实现 HTTP/2 的复杂性表明 HTTP/2 并不像您希望的那样容易开启（对于一个主要的新协议来说不足为奇）;它也不是有保证的性能提升。大多数网站在切换到 HTTP/2 后都看到了性能提升,但有些网站的性能有所下降,或者没有像他们想象的那样受益。充分利用任何新技术都需要很好的理解。我预计 HTTP/2 会继续增长,至少在 QUIC 发布并变得无处不在之前是这样,但两者都不会很快在许多实现中取代 HTTP/1,如果有的话。

10.3 HTTP/2 的未来版本以及 HTTP/3 或什么

HTTP/4 可能带来既然 HTTP/

2 已在现实世界中出现并且使用量正在增长,您对 HTTP 的下一次迭代有何期待?是 HTTP/2.1 还是 HTTP/3?

10.3.1 是 QUIC HTTP/3 吗?

QUIC 将 HTTP/2 的概念提升到一个新的水平,因此被视为 HTTP/2 的继承者,按照定义使其成为 HTTP/3。 2018 年 7 月,HTTP 工作组的两位主席之一表示,44 “我在大多数方面都将 QUIC 视为 [HTTP/3]。 . . hq 是 h2 的逻辑继承者”,2018 年 11 月,HTTP 工作组同意 QUIC 的 HTTP 部分应称为 HTTP/3,并将其（和 QPACK)从 QUIC 工作组移至 HTTP 工作组publication.45在 QUIC 获得批准之前,HTTP/3 的名称不会被正式注册,因此不能保证 HTTP/3 会是什么,但在撰写本文时看起来很有可能。

QUIC 被定位为一种更大、更通用的传输层协议,但它的用途远不止 HTTP,而 HTTP/3 只是一个用例

---

[41] https://tools.ietf.org/html/rfc8336
[42] https://tools.ietf.org/html/rfc8441
[43] https://datatracker.ietf.org/doc/draft-ietf-httpbis-cache-digest/?include_text =1
[44] Patrick McManus（HTTP-WG 联合主席）,IETF 102 HTTPBIS 会议 II (https://youtu.be/tQAfDmW0qII?t=588)
[45] HTTP 工作组 IETF 103 (https://youtu.be/uVf_yyMfIPQ?t=4956)

Machine Translated by Google

快速。在许多方面,QUIC 是 TCP 而不是 HTTP/2 的继承者,尽管有些人对名称 TCP/2.46有异议。这是名称 HTTP/3 将用于区分较大的 QUIC 协议和 HTTP 部分的另一个原因这个的。这并不意味着 HTTP/2 已经完成并且不会进一步发展,就像 HTTP/2 出现后 HTTP/1.1 停止使用一样 没有计划淘汰 HTTP/1.1 或 HTTP/2,因为许多 HTTP 实现可能会保留在 TCP 之下,使用 HTTP/2 或 HTTP/1.1。然而,一旦 QUIC 变得广泛可用（这需要一些时间）,那么 HTTP/3 将代表 HTTP 的最佳版本,应尽可能使用 因此得名。

10.3.2 进一步发展HTTP二进制协议

回避 QUIC (h3) 和 TCP 版本 (h2/h2c) 的差异和未来方向,未来应该如何扩展这种新的二进制和多路复用协议?

过去,HTTP 标头支持核心 HTTP 协议的扩展,具体取决于客户端和服务器对这些标头的使用。HTTP/2 添加了具有新设置值和新帧类型的新功能。设置 values47特别允许在建立连接时预先发现新功能,这比发送 HTTP 标头希望对方能理解它要好得多。

该协议已通过ALTSVC、 ORIGIN和（提议的)进行了扩展 CACHE_DIGEST帧。还提出了其他建议,例如辅助证书,48因此存在一种通过新帧类型推动协议向前发展的稳健方法。 49目前没有明确需要像 HTTP/2.1 这样的版本编号,这也是为什么 HTTP工作组删除了次要版本号并将协议称为 HTTP/2 而不是 HTTP/2.0。

10.3.3 在传输层之上演化 HTTP

尽管 HTTP/2 和 QUIC 着眼于 HTTP 的较低级别的传输层,但在 HTTP 的较高级别做了什么?一直有源源不断的新 HTTP 标头来控制客户端和服务器的行为,但 HTTP 语义自 HTTP/1.1 以来并没有太大变化,而 HTTP/2 也没有给这个更高的级别带来任何变化。

正如我之前提到的,不断出现的一个关注领域是 HTTP cookie 的替代方案。不过,到目前为止,似乎没有任何取代它们的提议获得通过
牵引力。

否则,HTTP 已被证明在更高级别上出人意料地健壮,并且需要澄清和调整而不是扩展。 IETF 工作组正在对 HTTP 进行大量扩展,50不太正式的 Web 平台

---

46 https://github.com/HTTPWorkshop/workshop2016/blob/master/talks/quic.pdf https://
47 www.iana.org/assignments/http2-parameters/http2-parameters.xhtml#settings https://工
48 具。 ietf.org/html/draft-ietf-httpbis-http2-secondary-certs https://www.iana.org/assignments/
49 http2-parameters/http2-parameters.xhtml#frame-type https://github.com/ httpwg/http-扩展
50

Machine Translated by Google

孵化器社区团体 (WICG),51或作为私人外部利益 如 SPDY 和 QUIC 开始与谷歌一样。大多数这些扩展都可以在不从根本上改变 HTTP 的情况下实现,而是使用已经可用的扩展方法（HTTP 标头、HTTP/2 设置或新的 HTTP/2 帧类型）。

新的HTTP方法

也许最令人惊讶的是,自 HTTP/1.1 以来,主要的 HTTP 方法（GET、 POST、 PUT、 DELETE等)没有得到扩展。 Web Distributed Author 52引入了一些新方法（包括 PROPFIND、 ing 和 Versioning (WebDAV) COPY和LOCK）,还有一些 RFC 介绍了其他的,53但最后一个是在 2010 年注册的（BIND）。总的来说,HTTP 主要依靠 20 年前引入的四种核心方法（主要是GET和POST,也有一些使用PUT和DELETE)存活下来。

HTTP 方法相对容易添加,但没有必要,而且大多数需求都可以通过POST 代理。以下示例中的操作变量传达了必要的特定于应用程序的方法（订购此商品）,因此无需在 HTTP 中执行相同的操作：

```
:方法:POST
:路径：/ api / doaction

{
    “行动”：“订单”， “项
    目”:12345，。“数量”:1 }
```

其他 HTTP 实现使用 HTTP 标头提供附加信息,包括要采取的操作：

```
:方法:POST
:path: /api/doaction动作:顺序


{
    “项目”:12345，“数
    量”:1
}
```

20 年后,我没有看到立即需要新的 HTTP 方法。尽管未来几年无疑会引入一些新方法,但我预计它们不会对引入它们的应用程序产生重大影响。

---

[51] https://www.w3.org/blog/2015/07/wicg/
[52] https://tools.ietf.org/html/rfc4918 https://
[53] www.iana.org/assignments/http-methods/http-methods .xhtml

Machine Translated by Google

新的HTTP标头

新的 HTTP 标头的使用随着时间的推移而增长,我预计它会继续增长。 HTTP/2 明确禁止使用以冒号开头的新伪头字段54 （:method、 :scheme、 :authority、 :path和:status）,尽管这些字段可以添加新规范（例如:protocol伪头)通过 HTTP/2 RFC 添加了 Bootstrapping Websockets）。55

其他标头可以并且已经被添加为应用程序认为合适的。 HTTP 规范就新标头字段的注意事项提供了建议,56表明它们旨在扩展。有一个消息头的官方注册表57 （包括 HTTP 使用的那些）,但应用程序使用许多头而无需注册。

HTTP 标头允许轻松添加新功能,无论该功能是双方之间的附加信息（例如“此响应使用 XXX 格式”）、提示（例如“仅供参考,我支持以下格式”）、身份验证信息（如 cookie）、路由信息（“这是从这个 IP 地址转发的”)等等。即使中间的 HTTP 基础设施（例如 HTTP 服务器或 Web 浏览器)不理解标头并盲目地传递它,任何一端的应用程序都可以对标头进行操作。

在过去几年中,特别是使用 HTTP 标头来提高安全性。
这些标头通常作为启用内容安全策略 (CSP)58和 HTTP 严格传输安全 (HSTS)等安全功能的说明作为响应从网站发送到浏览器。 59 反过来,有建议向服务器提供有关客户端（Web 浏览器或其他)的更多信息,例如客户端提示规范,60应允许根据客户端支持的内容提供不同的内容。

随着 HTTP 使用的持续增长以及在客户端和服务器端添加新功能,毫无疑问会添加许多新的标头。新的 HTTP 标头还没有

需要且不应需要新版本的 HTTP。

新格式

自从HTTP/1.0引入HTTP头之后,HTTP就支持了不同的文件格式,并允许HTTP协议使用不同的内容编码（如gzip、br等压缩方式）。同样,HTTP 在这方面很容易扩展,而不需要新的版本号。

---

54 https://httpwg.org/specs/rfc7540.html#PseudoHeaderFields
55 https://tools.ietf.org/html/rfc8441 https://tools.ietf.org/html/
56 rfc7231#section-8.3 https://www .iana.org/assignments/
57 message-headers/message-headers.xhtml https://w3c.github.io/webappsec-
58 csp/ https://tools.ietf.org/html/rfc6797 https://tools.ietf .org/html/draft-ietf-
59 httpbis-client-hints
60

Machine Translated by Google

新状态代码HTTP

状态代码61是扩展 HTTP 功能的其他方式。它们不需要新版本,至少对于核心 HTTP 规范定义的粗略分组内的新状态代码,62如表 10.1 所示。

表 10.1 HTTP 状态代码分组

代码	类型	描述
1xx	信息性的	收到请求;该进程仍在继续。
2xx	成功的	请求已成功接收、理解和接受。
3xx	重定向	需要采取进一步的行动来完成请求。
4xx	客户端错误	该请求包含错误的语法或无法实现。
5xx	服务器错误	服务器未能完成明显有效的请求。

版本。然而,当103首次使用时,63它揭示了许多7年底推出,没有例如Web浏览器和CDN需要这些实现预计不会收到多个 HTTP 响应,因为1XX信息响应没有被使用太多,除非在以下情况下具体情况。从技术上讲,此更改是一个不间断的更改,不需要新版本。但由于它与现有状态代码的工作方式略有不同,因此许多客户将其视为重大更改,直到他们更正其实施为止。

还可以添加新类别（6XX、7XX、8XX、9XX)而无需扩展传统的三位数响应状态代码类别。在 HTTP 出现的 20 年里,没有要求扩展状态码类别,但将来可能会扩展;范围在那里。

10.3.4 什么需要新的 HTTP 版本?

考虑到扩展 HTTP 的所有方式,HTTP 的版本在过去 20 年中没有改变也许不足为奇。需要更改什么才能需要新版本?答案是目前没有人知道！ HTTP 工作组集思广益,64但 HTTP/2 的某些领域尚未解决（其中大部分在 QUIC 和 HTTP/3 下得到解决）。但目前,对于 HTTP/3 之后的下一个 HTTP 版本可能带来什么,还没有确切的想法。

有一件事是肯定的:与 HTTP/2 和 HTTP/3 一样,下一个主要版本需要带来一些不向后兼容的重大更改以保证主要版本号升级。这些是否是有线格式的变化（如 HTTP/2）,

61 https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml#http-status-
62 codes-1 https://tools.ietf.org/html/rfc7231#section-6 https://tools.ietf.org/html/rfc8297 https://
63 github.com/HTTPWorkshop/workshop2016/wiki/Future-of-HTTP#http3
64

Machine Translated by Google

传输的基本期望发生变化（如 HTTP/3 通过 UDP 转向 QUIC），或者其他一些事情仍有待观察。

10.3.5 HTTP 的未来版本可能会如何引入

HTTP/2 的引入提供了更多扩展协议的机会，也引入了使用 ALPN 或第 4 章中详述的其他升级方法引入新版本的途径。如果在某个时候需要进行重大更改，则应该更容易引入，理想情况下，这将带来比过去 20 年更多的创新。

HTTP/3 还试图摆脱 TCP，如果成功，它将为未来转向全新的底层技术打开新的大门。开发人员将来会使用 TCP 和 UDP 以外的东西来传递 HTTP 消息吗？ IP会变吗?答案尚不清楚，但这是互联网协议进步的新时代，开发人员终于有了引入它们的方法。

10.4 HTTP 作为更通用的传输协议

HTTP 使用方面的变化如何，它们在未来可能将协议引向何方？ HTTP 的最初用例是网页，但人们普遍希望在这个用例之外使用流行的 HTTP。 HTTP 是一种相当容易理解并得到广泛支持的协议，因此存在许多实现和库。至少对于 HTTP/1.1，使用任何允许读取或写入 TCP 通道的软件很容易获得一个简单的 HTTP 服务器或客户端（尽管如前所述，HTTP/1.1 的文本性质可能导致许多问题，这些问题是最初被这种看似简单的东西所掩盖）。

HTTP 是一种成熟的协议，可以超越其初衷。在这个永远连接的世界中，HTTP 允许以一种标准化的、易于理解的方式在不同的系统之间进行简单的通信。从使用 REST API 或类似技术的复杂应用程序到 IoT 设备，HTTP 在 Web 应用程序中使用并直接在非 Web 应用程序中使用。

想要使用 HTTP 的应用程序有几种选择： 使用 HTTP 语义和消息

来传递非 Web 流量。 使用 HTTP/2 二进制框架层。 使用HTTP 启动另一个协议。

我将在本章的其余部分探讨这些选项。

10.4.1 使用 HTTP 语义和消息传递非 Web 流量

这种方法是在网页外部使用 HTTP 最常见的方法。 API 消息可以通过 HTTP 以客户端和服务器知道使用的任何格式（XML、JSON 或某些专有格式)发送。通常，这些基于 HTTP 的 API 使用常见的 HTTP 方法（GET、 POST、 PUT和DELETE)跨服务执行操作。

例如，微服务架构使用小型独立服务，用于

Machine Translated by Google

which HTTP 是一个很好的选择。其他协议可以很容易地在 HTTP 中建模（请参阅下面的侧边栏）。

HTTPS 上的 DNS (DoH)

域名系统 (DNS) 是一种相对简单的协议,通常涉及从中央目录获取一种记录类型。尽管 DNS 传统上是在单独的端口（端口 53)上拥有自己的专有格式,但它也可以通过简单的GET请求来实现。

切换到 HTTP 的原因之一是使用 HTTPS 提供的加密。过去,DNS 一直是一个未加密的系统,尝试添加加密导致复杂的协议,如 DNSSEC 和 DANE,它们都有自己的缺陷和弱点,这超出了本书的范围。 HTTPS 经过验证、安全（如果配置正确)并且得到很好的支持。通过在 DNS 上放置 HTTPS 接口,开发人员可以解决他们争论了几十年的 DNS 加密问题。该系统被称为 DNS over HTTPS,a或 DoH。 （尽量不要想起荷马·辛普森。）

开发人员可以在现有 DNS 消息传递上使用 TLS,而不是转向 HTTPS,并且为此目的也存在单独的规范,b但使用 HTTPS 似乎更易于实施,并允许使用 HTTP 的其他优势。

这些好处包括在多路复用请求中使用 HTTP/2 或 QUIC 的广泛支持和代理友好使用,以及使用 HTTP/2 推送发回多个响应。

在撰写本文时,Googlec和 Cloudflared在服务器端支持 DoH； Firefox 已经添加了支持并开始试验它,e并对其进行了很好的解释。f结果令人鼓舞,g这显示了将 HTTP 用于应用程序的强大功能 即使是已经拥有自己的传输协议的应用程序！

---

a https://tools.ietf.org/html/draft-ietf-doh-dns-over-https
b https://tools.ietf.org/html/rfc7858 https://tools.ietf.org/html/
c rfc8484 https //developers.cloudflare.com/1.1.1.1/dns-over-
d https/ https://blog.nightly.mozilla.org/2018/06/01/improving-dns-
e privacy-in-firefox/ https:/ /hacks.mozilla.org/2018/05/a-cartoon-intro-to-dns-over-https/
f

g https://blog.nightly.mozilla.org/2018/08/28/firefox-nightly-secure-dns-experimental-results/

IETF 发布了一个规范， On the Use of HTTP as a Substrate,一组建议和最佳实[65]其中列出践,以通过这种方式充分利用 HTTP。一些应用程序偏离了核心 HTTP 规范并且只使用了协议的一部分,本文档并不试图禁止这种分歧,但它指出了为什么这样做可能会失去使用 HTTP 的一些好处和经验教训。

HTTP 的发展方式有很多原因,其中一些可能不会立即显现出来,本文档试图为该协议的其他用户阐明这些原因。

---

[65] https://tools.ietf.org/html/rfc3205但目前正在升级,因此应该很快被https://tools.ietf.org/html/draft-ietf-httpbis-bcp56bis取代

Machine Translated by Google

10.4.2 使用 HTTP/2 二进制框架层

多路复用二进制框架层引入了新的选项和使用 HTTP 的原因。尽管 HTTP 可能由于支持直接 TCP 连接或 WebSockets 的低效率而被拒绝,但 HTTP/2 解决了许多这些问题并成为竞争者。例如,Google 的新 gRPC 协议（gRPC Remote Procedure Calls 的缩写)[66]使用 HTTP 语义和 HTTP/2 二进制框架层[67]来实现基于协议缓冲区的更高效的 API,[68]而不是 JSON 等效率较低的格式。如果没有 HTTP/2 提供的改进,Google 不太可能为此目的选择 HTTP。

有些人可能希望将 HTTP/2 二进制框架层用于非 HTTP 流量作为全双工协议,而不是 HTTP/2 的客户端启动协议,只有受限的 HTTP/2 推送用于服务器到客户端的请求.目前不支持此功能。也许 QUIC 是从头开始构建的,考虑到用于其他协议,更适合这个目的（或者任何此类实现都将通过 TCP 反向移植到 HTTP/2 中）。同时,开发人员可以使用 HTTP 作为启动另一个协议的方式,例如真正的双向协议。

10.4.3 使用 HTTP 启动另一个协议

将 HTTP 用作更通用的协议的最后一个选项是在开始时使用 HTTP,然后切换到另一个协议。 HTTP 是一种广泛支持的协议,可用于屏蔽其他协议以使其看起来像 HTTP 以获得更广泛的网络支持,尤其是通过代理和其他中介。选项包括使用 HTTP CONNECT方法和升级来自 HTTP 的连接（例如 Web Sockets）。这两个选项都以 HTTP 开头,然后迅速转向另一个协议。然而,对于旁观者来说,它们可能看起来像是 HTTP 流量。

HTTP连接方法

从 HTTP/1.1 开始,HTTP 就有了CONNECT方法,它允许将 HTTP 连接用作代理以隧道连接到备用服务器和端口。此方法通常用于允许 HTTP 代理使用如下语法建立 HTTPS 连接隧道：

```
连接 example.com:443 HTTP/1.1
```

HTTP/2 格式的语法类似：

```
:方法 :连接 :权限：
example.com:443
```

---

[66] https://grpc.io/faq/
[67] https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md https://
[68] developers.google.com/protocol-buffers/docs/overview

Machine Translated by Google

这段代码创建了一个从 HTTP 服务器到 example.com 的新连接,并允许消息从客户端传递到该服务器,如图 10.8 所示。

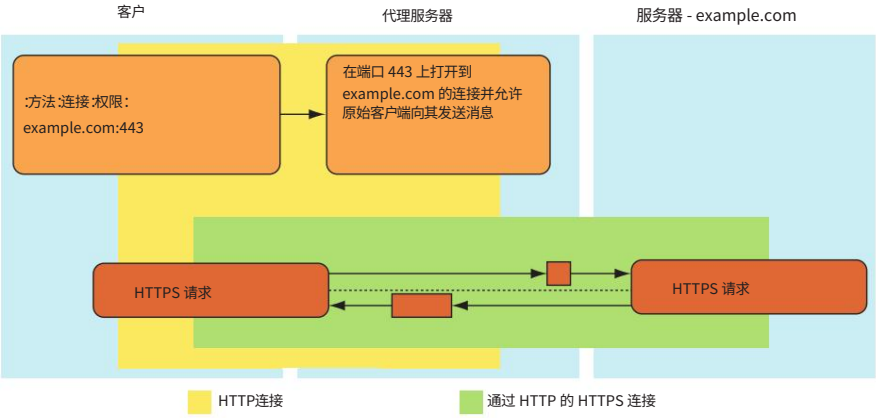


图 10.8 使用CONNECT通过 HTTP 代理连接建立 HTTPS 连接隧道

请注意,代理的这种使用与中间人代理是分开的,中间人代理创建两个单独的 HTTP 连接。在这种情况下,一直有一个 HTTPS 连接,但通过两个 TCP 连接,因此在初始设置之后,就好像客户端直接连接到终端服务器一样。

代理无法读取 HTTPS 连接,因为存在端到端 HTTPS 连接,因此如果客户端和服务器端的任一端都支持 HTTP/2,则此设置甚至允许通过 HTTP/1.1 代理进行 HTTP/2 连接。见图 10.9。

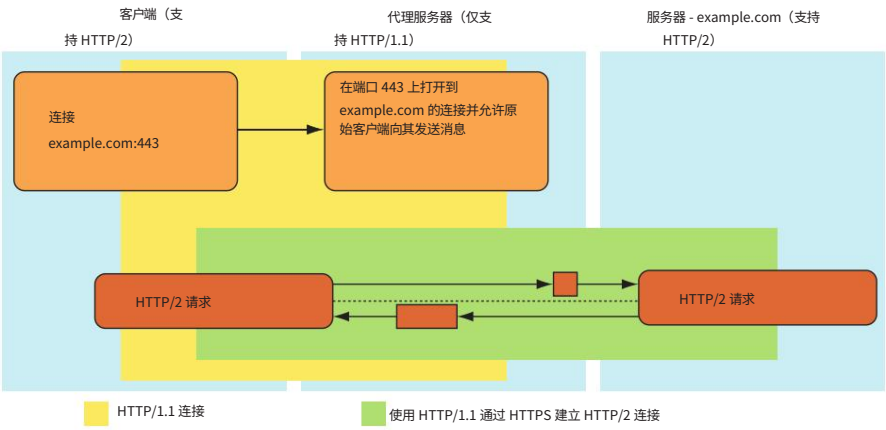


图 10.9 使用CONNECT通过 HTTP/1.1 代理建立 HTTP/2 连接隧道

Machine Translated by Google

隧道连接不需要携带除初始消息之外的HTTP消息,并且可以使用任何协议。图 10.10 使用 HTTP 连接到端口 22 上的 example.com（由 SSH 使用）。此后,在此连接上发送的任何消息都应该是 SSH 指令,而不是 HTTP。

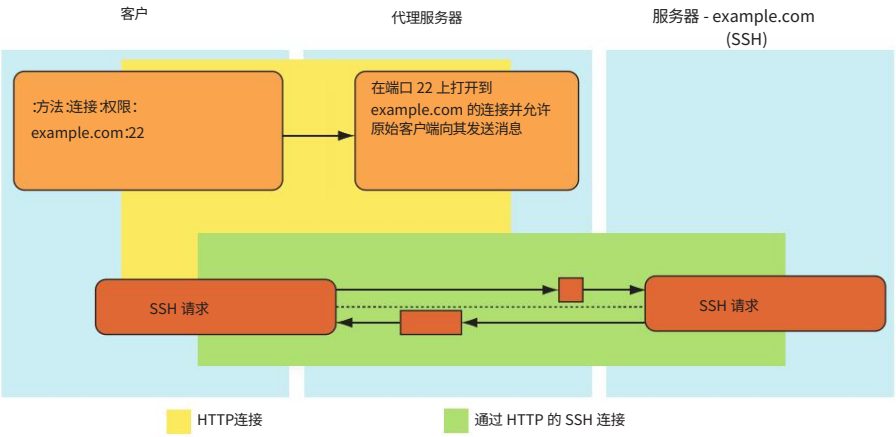


图 10.10 使用CONNECT通过 HTTP 连接建立 SSH 连接隧道

同样,此设置可用于允许可能无法直接从系统中使用的替代协议（假设它们在代理服务器上可用）。所有这些示例都表明 HTTP 可用于允许将其他协议引入环境。因此,HTTP 可用于引入新协议,而不用担心网络基础设施会处理它们,尤其是当 HTTPS 用于对代理服务器本身隐藏详细信息时。在从 TCP 转移到更轻量级的 UDP 协议的过程中,这个过程可能比 QUIC 更有趣。目前,HTTP/3 不支持不恢复为 TCP 的 CONNECT方法,但已经有人为此提出了建议。69此设置需要代理服务器,并且客户端必须配置为通过代理服务器进行连接。

升级来自HTTP的连接（例如 WebSockets）

另一种选择是使用 HTTP 启动连接,然后将该连接升级到替代协议。 WebSockets 以这种方式工作。握手如下图所示,使用 HTTP/1.1 语法。

客户端请求（为简单起见,未包含其他一些 WebSockets 标头字段）

```
GET /application HTTP/1.1 主机:
example.com 升级:websocket连接:升级
```

[69] https://tools.ietf.org/html/draft-pardue-httpbis-http-network-tunnelling

Machine Translated by Google

服务器响应

```
HTTP/1.1 101 切换协议升级:websocket连接:升级
```

然后,仍然通过 TCP 端口 80 或 443 传送的 HTTP 连接被转换为 WebSockets 连接。对于更小的应用程序,WebSockets 比 HTTP 更高效
消息,因为没有 HTTP 标头的开销（由于 HPACK,HTTP/2 中的开销大大减少但仍然有开销）,并且它还允许全双工通信。因此,使用 WebSockets 可用于实时更新应用程序,例如聊天、金融价格应用程序或体育更新。

HTTP/2 不支持此方法,因为它需要升级整个 HTTP 连接,这对于只应升级流的多路复用连接没有意义。因此,基于 HTTP/2 的 WebSockets 应该使用CONNECT方法70 （略有不同以允许指定协议）。

客户端请求（为简单起见,未包含某些 WebSockets 标头字段）

```
:方法=连接
:protocol = websocket :scheme
= https :path = /
application :authority =
example.com
```

服务器响应

```
:状态= 200
```

此后,WebSockets 数据可以在该流的任一方向上交换。
该连接上的其他 HTTP/2 流可以继续讨论 HTTP、其他 WebSockets 通信,甚至可以以类似方式定义的一些其他协议。 WebSockets 开启了许多可能性,而 HTTP 的请求和响应特性使之变得困难。可以使用相同的方法切换到任何其他协议,所以我以另一种使用 HTTP 的方式结束本章!

在这最后一章中,我试图展望未来,看看 HTTP 将何去何从。诚然,关于下一版本的 HTTP 可能是什么样子,有很多猜测,但几乎没有确定的信息,因为在这个阶段,没有人知道。您可以确定的两件事是 HTTP 仍将是 Internet 的重要组成部分,并且它的使用可能会增长。在第 1 章的开头,我说“万维网经常被错误地与互联网互换使用,[但是]网络的持续兴起,或者至少是为它创建的 HTTP,可能意味着很快它可能不像以前那样远离事实。”本章解释了允许 HTTP 使用增长并将继续允许增长的各种选项。

---

70 https://tools.ietf.org/html/draft-ietf-httpbis-h2-websockets

Machine Translated by Google

到目前为止,您应该对 HTTP/2 和相关技术有了很好的理解。您还发现,尽管 HTTP/2 提供了许多机会和潜在的速度改进,但并不能保证提高性能。

那些寻求快速修复的人可能会有些不满意,但 HTTP/2 对于大多数网站来说仍然更快,而且它仍处于早期使用阶段。QUIC 和 HTTP/3 将使用与 HTTP/2 相同的概念并将它们提升到一个新的水平。有了从本书中获得的知识,您应该能够很好地利用 QUIC。即使在 QUIC 出现之前,关于 HTTP/2 以及如何最好地使用它还有很多东西需要学习。但我希望这本书能让你有能力和兴趣进一步试验并观察这个有趣的、广泛使用的协议的演变。

快乐的实验!

概括

HTTP/2 在标准化过程中有一些争议和相当多的反对者。这些问题并没有阻碍 HTTP/2 在现实世界中的采用,并且在大多数情况下并没有被证明像人们担心的那样存在问题。 HTTP/2 已在现实世界中得到广泛采用,并且已经可以考虑

因为这个原因取得了成功。

HTTP 现在由 HTTP/2 二进制框架层和 HTTP 语义层组成
通过这些帧发送的抽动。

QUIC 被视为 HTTP/2 的自然继承者,当它发布时,
QUIC 规范的 HTTP 部分将称为 HTTP/3。 HTTP 有很多扩展其用途的方法,HTTP/2
增加了更多。 HTTP 可用于传输其他协议。 HTTP 的未来一片光明!

Machine Translated by Google

附录
将常见的 Web 服务器升级到 HTTP/2

本附录涵盖升级一些常见的 Web 服务器以在直接模式 (A1) 或作为反向代理 (A2) 下启用 HTTP/2 支持。附录绝不是所有支持 HTTP/2 的 Web 服务器的权威参考;您应该参考您自己的网络浏览器的文档。

A.1 升级您的 Web 服务器以支持 HTTP/2

在接下来的部分中,我将讨论如何安装 HTTP/2 兼容版本的常见 Web 服务器以及如何配置它们以支持 HTTP/2。请注意,此列表远非详尽无遗,并且会随着时间的推移而变化;它旨在反映一些设置 HTTP/2 兼容 Web 服务器的示例。每个部分重复许多步骤,因此请跳至您最感兴趣的服务器部分。

自签名 HTTPS 证书和证书错误本附录中的大多数示例都使用自签名 SSL/TLS 证书（又名 HTTPS 证书）。这些证书由 Web 服务器提供或使用openssl命令等创建。要让浏览器识别和信任证书,它必须由公认的证书颁发机构 (CA) 颁发。

每个浏览器或操作系统都保留这些 CA 的列表,它会根据这些 CA 检查任何证书。 Web 服务器附带的虚拟证书（以及您在本附录中创建的证书)不是由 CA 颁发的,它们被称为自签名证书。这些证书允许站点通过 HTTPS 运行,但由于证书,您会收到来自浏览器的警告。通常可以跳过这些错误并访问该站点,但是即使您忽略可怕的警告和红色挂锁,使用自签名证书也会出现一些问题。

Machine Translated by Google

但是,当点击证书错误时,Chrome 和 Opera 不使用 HTTP 缓存作为安全功能,因此无法使用 HTTP/2 推送。

可以将自签名证书添加到计算机的信任库中,以便浏览器识别证书、显示绿色挂锁并解决这些错误。建议将此技术用于本地开发（通过本地主机）。

具体操作方式取决于您的浏览器和操作系统,但通常双击证书会提供执行此操作的选项。

如果您运行的是具有真实域名的服务器,则最好获得真实证书,因为所有浏览器都可以识别它而无需额外的步骤来注册它。此选项不适用于本地主机或任何非私有 IP 地址（例如 127.0.0.1 或 ::0）。

https://bugs.chromium.org/p/chromium/issues/detail?id=103875#c8

A.1.1 Apache

Apache HTTP Server (AKA httpd) 在 2.4.17 版中通过mod_http2模块引入了 HTTP/2 支持（当它作为一个单独的模块在核心 Apache 外部管理时称为mod_h2 ）。1在 2.4 版之前它被标记为实验性的.26. 我建议运行最新版本的 Apache（可从http://httpd.apache.org/ 获得）,因为此模块在这些版本和更高版本之间得到了积极改进。

Apache 使用 ALPN 支持基于 HTTPS 的 HTTP/2,并且从未实现协商 SPDY/HTTP/2 的旧 NPN 方法。因此,Apache 至少需要 OpenSSL 1.0.2（或等效版本)才能启用 HTTP/2 支持,即使对于仍然支持 NPN 的浏览器也是如此。 Apache 确实允许通过纯文本 HTTP 连接（称为 h2c)的 HTTP/2,尽管此功能的用途有限,因为没有浏览器支持它。

Apache 使用 nghttp2 HTTP/2 库作为其 HTTP/2 功能的基础,并且在撰写本文时需要 1.5.0 或更高版本才能获得完整功能。

Apache 支持 HTTP/2 推送（在第 5 章中介绍）。 Apache 也有mod_proxy_http2模块,它允许通过 HTTP/2 连接到后端系统,尽管这个模块在撰写本文时仍被标记为实验性的。作为“你需要一直说 HTTP/2 吗？”第 3 章的侧边栏讨论了,通常没有必要一直说 HTTP/2 到后端连接。

WINDOWS上的 APACHE

尽管在 Windows 上的生产系统上使用 Apache 可能不像在 Linux 上那样普遍,但 Windows 版本通常用于开发目的。从源代码为 Windows 编译 Apache 超出了本书的范围。但是,如果您希望在本地启动并运行 Apache 以试验 HTTP/2,您可以使用来自各种来源的预构建 Windows 版本 不幸的是,不是来自 Apache。 Windows 版本的热门选项包括 Apache Haus、2 Apache

https://github.com/icing/mod_h2
https://www.apachehaus.com/cgi-bin/download.plx

Machine Translated by Google

Lounge,3和各种 XAMPP 安装。这些选项列在 Apache 网站http://httpd.apache.org/docs/current/platform/windows.html#down。您通常需要选择要使用的 Apache 类型。您的选择取决于以下因素：

- 您要使用哪个版本的Visual C++ 您可能必须安装Visual C++ Distributable（可从Microsoft 获得）。请注意,Microsoft 喜欢在某些文档中按年份引用这些可分发文件,在其他文档中按版本引用。为了便于参考,我列出了两种类型： – Visual Studio 2008:VC++ 9 – Visual Studio 2010:VC++ 10 – Visual Studio 2012:VC++ 11 – Visual Studio 2013:VC++ 12 – Visual Studio 2015:VC++ 14 – Visual Studio 2017:VC++ 15您可以在 Windows 系统的“添加或删除程序”控制面板中查看已安装的版本。安装更高版本很容易,我建议选择最新版本。 ■体系结构 在64 位和32 位（又名x86)之间进行选择。如果您运行的是 64 位操作系统,我推荐 64 位 Apache。如果您运行的不是 64 位操作系统,问问自己为什么不呢,在这个时代!您可以通过右键单击“我的电脑”（或某些版本中的“我的 PC”)并从上下文菜单中选择“属性”来查看架构。在出现的窗口中,您应该会看到系统类型下方列出的体系结构。

- OpenSSL 版本 您需要 1.0.2 或更高版本,但一些站点提供针对 1.1.0 或更高版本的构建,您最好尽可能使用最新版本。

下面介绍如何安装支持 HTTP/2 的 Apache Haus 包。安装其他类似：

1. 根据以上三个选择,下载合适的版本,并将文件夹解压到你喜欢的位置（例如C:\Program Files\Apache\Apache24）。

2. 编辑 conf\httpd.conf 文件以更改此行

   定义 SRVROOT /Apache24

   到您的服务器位置,例如

   定义 SRVROOT C:\Program Files\Apache\Apache24

   确保您没有以斜线结束此行(C:\Program Files\Apache\Apache24\),如果路径包含空格（例如Program Files）,请记住将路径用引号引起来。

---

[7] https://www.apachelounge.com/download/

Machine Translated by Google

3确保mod_http2模块已激活且未被注释掉（前面有一个哈希）：

```
LoadModule http2_module 模块/mod_http2.so
```

4保存对 httpd.conf 的更改,这可能需要管理员权限,如果
它位于 Program Files 文件夹中。

5启动 Apache,最好从命令提示符启动,这样您就可以看到任何错误：

```
cd c:\Program Files\Apache\Apache24\bin httpd.exe
```

以下是可能阻止此命令运行的一些常见问题： – 一条错误消息抱怨缺少 VCRUNTIME140.dll 或类似文件,这表明您尚未安装所需的 Visual C++ 可再发行组件,如本附录前面所述.

– 日志文件设置为只读,因此您会看到一条关于无法打开错误日志的错误消息。右键单击 C:\Program Files\Apache\Apache24\Logs 文件夹,从上下文菜单中选择“属性”,然后取消选中“只读”选项（如果已选中）。

– Windows 防火墙弹出窗口要求您授予它对此 Web 服务器的访问权限。你需要回答是。

– 一条错误消息说您没有端口 80 或端口 443 的权限。很可能,另一个程序正在使用此端口。常见的罪魁祸首包括服务应用程序中的万维网发布服务 (AKA IIS)。 （如果您不使用 IIS、Skype 或其他 Web 服务器,请停止此服务并将其设置为手动而不是自动。）

6当 Apache 正在运行时,检查http://localhost,您应该会在其中看到 Apache Haus 欢迎页面。然后尝试 https://locahost,如果您使用默认的虚拟证书,您会在其中收到证书错误。跳过证书错误,如果您打开开发操作工具并添加协议列,您应该会看到您的页面通过 HTTP/2 交付,如图 A.1 所示。

7当您手动正确运行 Apache 时,您可以将其安装为一项服务,以便于停止和启动并在您的机器重新启动时自动启动它。以管理员身份启动命令提示符,然后运行以下命令：

```
cd c:\Program Files\Apache\Apache24\bin httpd.exe -k 安装
```

您应该会在“服务”屏幕上看到该服务。您可以使用以下代码删除该服务：

```
cd c:\Program Files\Apache\Apache24\bin httpd.exe -k 卸载
```

Machine Translated by Google

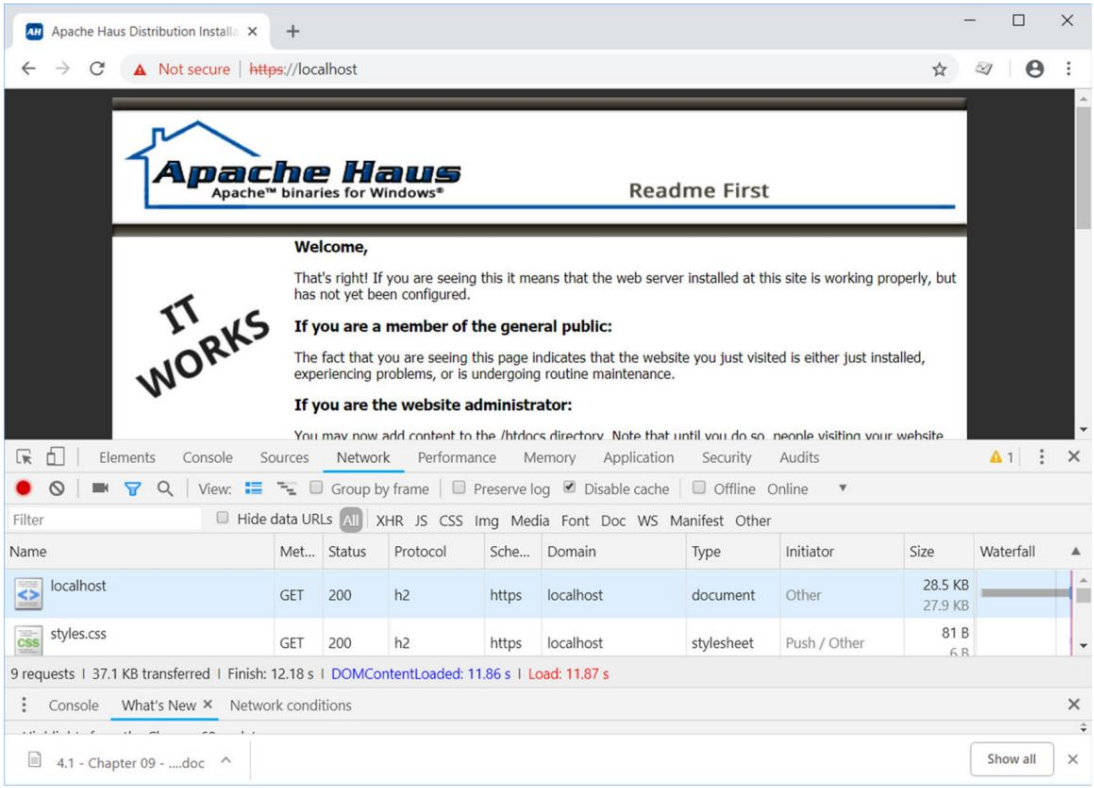


图 A.1 运行 HTTP/2 的 Apache

LINUX上的APACHE

在 Linux 服务器上的 Apache 上使用 HTTP/2 更加复杂,因为 Apache 和所需软件的版本较旧（在第 3 章中讨论）。您通常必须从源代码安装。我在本节中讨论了一种用于 Red Hat/CentOS 的方法,但如果您运行的是不同风格的 Linux,则需要调整此方法。

它所基于的 Red Hat Enterprise Linux (RHEL) 和 CentOS 在 7.4 版中添加了 OpenSSL 1.0.2 支持,解决了一个问题。但在撰写本文时,它们不包含支持 HTTP/2 的 Apache 版本;相反,它们默认包含旧的 2.4.6 版本。此外,当 Apache 使用 prefork MPM（多处理模块)时,HTTP/2 不受支持,4通常出于兼容性原因默认安装该模块,并且如果不重新编译就无法更改。

Red Hat /CentOS 有两个半官方的附加存储库:

---

[4] https://serverfault.com/questions/383526/how-do-i-select-which-apache-mpm-to-use
[5] https://fedoraproject.org/wiki/EPEL

Machine Translated by Google

红帽/中央操作系统。不幸的是,它不包括 Apache,尽管它确实包括在 Apache 中启用 HTTP/2 支持所需的更新版本的 nghttp2。

- Red Hat Software Collections (RHSCL) repo[6]是官方维护的Red Hat repo,其中包括较新版本的通用软件。版本 3.1 包括 Apache 2.4.27,其中包括 HTTP/2 支持,前提是您使用的是 RHEL/CentOS 7.4 或更高版本。此 repo 在单独的位置 (/opt/rh/httpd24/) 安装了一个附加版本的 Apache (apache24),其配置文件和位置与正常情况不同,因此可能需要一些时间来适应。此外,由于 HTTP/2 正在积极开发中,因此自版本 2.4.27 以来已经发生了其他开发。这些集合为您提供半受支持的版本,但可能并不理想,因为它们可能运行的是旧版本。

要获得最新版本,您可能需要从非官方位置安装或从源代码安装。这两种方法都不是理想的,也不会在没有仔细考虑的情况下被选择

考虑（见第 3 章）。

消除这些警告后,假设您仍想从源代码下载,我将向您展示如何操作。具体步骤取决于您使用的 RHEL/CentOS 版本。对于 7.4 之前的版本,您还必须从源代码安装 OpenSSL,因为支持 ALPN 的 1.0.2 版不可用。对于 7.4 及更高版本,一些步骤更容易,因为您可以使用默认版本的 OpenSSL：

1. 安装所需的所有依赖项：

```
sudo yum -y 安装 wget sudo yum -y 安装 perl
sudo yum -y 安装 zlib-devel sudo yum -y 安装
gcc sudo yum -y 安装 pcre-devel sudo yum -y 安装 expat-
devel sudo yum -y 安装 epel-release
```

2. 为软件设置一个目录：

```
cd /tmp
mkdir 来源
CD 来源
```

对于 OpenSSL,您有两种选择。

如果您使用的是 RHEL/CentOS 7.4 或更高版本,则可以使用 OpenSSL 的打包版本,其中包括 ALPN 支持：

```
sudo yum -y 安装 openssl-devel
```

[6] https://developers.redhat.com/products/softwarecollections/overview/

Machine Translated by Google

如果您使用的是旧版本或想要最新的 OpenSSL,则必须从源代码安装它：

```
#从http://openssl.org/source/获取 #例如:wget https://www.openssl.org/
source/openssl-1.1.1a.tar.gz wget https://www.openssl.org /source/
openssl-1.1.1a.tar.gz.asc #下载后验证包 gpg --verify openssl-1.1.1a.tar.gz.asc #If you get a  Can  t
check signature: No public key  error #then 获取相应的公钥并再次验证。
```

```
#例如:gpg --recv-keys
0E604491 gpg --verify openssl-1.1.1a.tar.gz.asc
#Note 你会看到一条警告,表明密钥未经过#with 可信签名认证。这是预料之
中的,此处不作介绍。

#解压文件并编译:tar -zxvf openssl-1.1.1a.tar.gz cd
openssl-1.1.1a ./config shared zlib-dynamic --prefix=/usr/
local/ssl

制作
须藤安装
光盘..
```

3安装nghttp2。同样,您可以使用打包版本：

```
sudo yum install -y libnghttp2-devel
```

此版本适用于 RHEL/CentOS 6 和 7。如果您想获得最新版本,因为它具有您想要使用的功能,请从源安装（但请注意,不包含 PGP 密钥）：

```
#下载并安装 nghttp2（mod_http2 需要）。
#从https://nghttp2.org/获取 #最新版本在这里:https://
github.com/nghttp2/nghttp2/releases/ #例如:wget https://github.com/nghttp2/nghttp2/releases/下载/v1.34.0/
nghttp2-1.34.0.tar.gz tar -zxvf nghttp2-1.34.0.tar.gz cd nghttp2-1.34.0 ./configure make
```

```
须藤安装
光盘..
```

4接下来,获取验证 Apache 下载所需的签名密钥：

```
#下载并安装 Apache 使用的 PGP 密钥 wget https://www.apache.org/dist/httpd/
KEYS gpg --import KEYS wget https://people.apache.org/keys/group/apr.asc gpg
- -导入apr.asc
```

Machine Translated by Google

5现在您需要安装 apr 和 apr-util 开发库。如果您使用的是 RHEL/CentOS 7.4 和打包的 OpenSSL (1.0.2),则可以使用打包的版本：

sudo yum -y 安装 apr-devel sudo yum -y 安装 apr-util-
devel

如果你想使用 Openssl 1.1.0 或更高版本或者正在使用 7.4 之前的 RHEL/CentOS 版本，你应该从源安装：

#下载并安装最新的apr #注意如果使用openssl 1.1.0或以上则
需要在APR 1.6或以上#从http://apr.apache.org/获取 #例如:wget http://mirrors. whoishostingthis.com/apache/apr/apr-1.6.5.tar.gz
wget https://www.apache.org/dist/apr/apr-1.6.5.tar.gz.asc #下载gpg后验证包--verify apr-1.6.5.tar.gz.asc #Note 你会看到一个警告,表明密钥没有经过#with a trusted signature的认证。这是意料之中的。

#安装包:tar -zxvf apr-1.6.5.tar.gz cd
apr-1.6.5 ./configure make

须藤安装
光盘..

#下载并安装最新的apr-util #注意如果使用openssl 1.1.0则需要APR-
UTIL 1.6或以上版本 #从http://apr.apache.org/获取 #例如:wget http:// mirrors.whoishostingthis.com/apache/apr/apr-
util-1.6.1.tar.gz wget https://www.apache.org/dist/apr/apr-util-1.6.1.tar.gz.asc #下载后验证包:gpg --verify apr-
util-1.6.1.tar.gz.asc #Note 你会看到一条警告,指出密钥未经过#with 可信签名认证。这是意料之中的。 tar -zxvf apr-util-1.6.1.tar.gz
cd apr-util-1.6.1 ./configure --with-apr=/usr/local/apr

制作
须藤安装
光盘..

6最后,安装 Apache：

#下载并安装apache #例如:wget http://
mirrors.whoishostingthis.com/apache/httpd/httpd
2.4.37.tar.gz wget https://www.apache.org/dist/httpd/httpd-2.4 .37.tar.gz.asc #下载后校验包：

Machine Translated by Google


```
gpg --verify httpd-2.4.37.tar.gz.asc #Note 你会看到一条警告,指出密
钥未经过#with 可信签名认证。这是意料之中的。

#提取源码
tar -zxvf httpd-2.4.37.tar.gz cd httpd-2.4.37
```

7 该步骤取决于您在前面的步骤中是否编译了openssl、 nghttp、 apr和apr-util 。如果您使用的是所有这些库的系统安装版本,则可以按如下方式编译 Apache：

```
./configure --enable-ssl --enable-so --enable-http2
制作
须藤安装
光盘..
```

如果您使用本地apr、 apr-util和openssl,则需要执行以下操作才能使用这些版本：

```
./configure --with-ssl=/usr/local/ssl \
    --with-apr=/usr/local/apr/bin/apr-1-config \ --with-apr-util=/usr/local/apr/bin/
    apu-1-config \ --enable-ssl - -enable-so --enable-http2make

须藤安装
光盘..
```

此代码应将 Apache 安装到 /usr/local/apache2。

8 启动 Apache 以查看它是否适用于基本 HTTP：

```
sudo /usr/local/apache2/bin/apachectl -k 优雅
```

如果一切顺利,您应该能够通过 HTTP 访问您的站点并看到默认的“It works”页面。

让 HTTP/2 工作需要更多的步骤。

9 如果编译了 OpenSSL 或 nghttp2,则需要编辑 bin 目录中的 envvars 文件以加载本地安装的路径。如果因为使用 7.4 而使用标准安装,请跳至下一步：

```
如果测试    x$LD_LIBRARY_PATH    !=    x    ;然后
        LD_LIBRARY_PATH=    /usr/local/apache2/lib:/usr/local/lib/: /usr/local/ssl/lib:
        $LD_LIBRARY_PATH
别的
        LD_LIBRARY_PATH=    /usr/local/apache2/lib:/usr/local/lib/:/usr/local/ssl/lib

菲
```

10 从 httpd.conf 文件中取消注释以下模块以加载 SSL 和 HTTP 模块以及 SSL 所需的 socache 模块：

```
LoadModule socache_shmcb_module 模块/mod_socache_shmcb.so LoadModule
ssl_module 模块/mod_ssl.so LoadModule http2_module 模块/mod_http2.so
```

Machine Translated by Google

11 添加此行以包括 SSL 配置：

```
包括 conf/extra/httpd-ssl.conf
```

12 添加此行以表明服务器优先使用 HTTP/2 (h2),然后再使用 HTTP/1 并启用日志记录：

```
<IfModule http2_module>
    协议 h2 http/1.1 LogLevel
    http2:info </IfModule>
```

如果您想通过 HTTP 启用 HTTP/2（并且不需要 HTTPS）,您也可以将h2c添加到Protocols行，但此功能不受任何浏览器支持且用途有限。

13 安装 HTTPS 证书。获取证书超出了本书的范围,因此我将向您展示如何使用 OpenSSL 生成基本的自签名证书。
浏览器不会识别此证书,但它可以用于一些基本测试：

```
#Note 我将使用的 openssl 命令需要以 root 身份运行 su - cd /usr/local/apache2/conf openssl req
\ -newkey rsa:2048 \ -x509 \ -nodes \ -keyout server.key \ -new \ -out server.crt \ -subj /
CN=server.domain.tld \ -reqexts SAN \ -extensions SAN \ -config <(cat /etc/pki/tls/openssl.cnf
\
<(printf [SAN]\nsubjectAltName=DNS:server.domain.tld )) \ -sha256 \ -days 3650
```

您应该为您的服务器输入一个正确的主题和 SAN（在前面代码的两个地方显示为 server.domain.tld）,但是由于证书无论如何都不会被识别,所以这个信息不是太重要。

默认的 Apache 配置（在 conf/extra/httpd-ssl.conf 中)期望证书被称为 server.key 和 server.crt,但如果您使用的是非默认配置,请根据需要更改名称。

14 停止并重新启动 Apache 以获取新配置。如果您在第 9 步中更改了 envvars 文件,那么正常重启是不够的：

```
/usr/local/apache2/bin/apachectl -k 停止
```

运行以下命令以确保所有 httpd 进程已停止：

```
ps-ef | grep httpd
```

Machine Translated by Google

使用以下命令重新启动：

```
/usr/local/apache2/bin/apachectl -k 优雅
```

15通过 HTTPS 访问站点,如果您使用的是第 13 步中的自签名测试证书而不是真实证书,请忽略证书错误。您应该看到通过 HTTP/2 (h2) 加载的页面,类似于图 A.1,方法是打开 devel oper 工具并添加 Protocol 列。

MACOS 上的 APACHE

在 macOS 上安装 Apache 类似于在 Linux 上安装它。首先,检查安装的版本。示例命令适用于 macOS Mojave 10.14：

```
$ httpd -V服务器版
本:Apache/2.4.34 (Unix)
服务器构建:2018 年 8 月 17 日 16:29:43 服务器的模块幻数:20120211:79
服务器加载:APR 1.5.2,APR-UTIL 1.5.4 使用编译:APR 1.5.2,APR-UTIL 1.5.4 架构：
64 -少量

服务器 MPM：          预叉
    螺纹：            不
        分叉：        是（可变进程数）
```

这最初看起来很有希望;您正在使用具有 HTTP/2 支持的最新版本的 Apache。但是,您可以看到 Apache 是使用不支持 HTTP/2 的 Apache 的预分叉版本编译的。您有两个选择： ■从另一个包管理器安装,例如 Homebrew.7

■从源安装。

这两种选择都不是理想的（见第 3 章）。

通过 Homebrew 安装很简单,涉及两个命令：

```
/usr/bin/ruby -e    $(curl -fsSL https://raw.githubusercontent.com/Homebrew/
        安装/掌握/安装）”

酿造安装httpd
```

要检查它,请启动 Apache：

```
brew 服务重启 httpd
```

Homebrew 使 Apache 通过端口 8080 和 8443 而不是通常的 80 和 443 可用,因此请访问 http://localhost:8080/ 以查看您的新 Web 服务器。

---

7 https://brew.sh/

Machine Translated by Google

配置 HTTP/2 需要编辑主配置文件

/usr/local/etc/httpd/httpd.conf

并进行以下更改

1从 prefork MPM 切换到事件 MPM：

```
LoadModule mpm_event_module lib/httpd/modules/mod_mpm_event.so #LoadModule
mpm_prefork_module lib/httpd/modules/mod_mpm_prefork.so
```

2取消注释以下模块：

```
LoadModule socache_shmcb_module lib/httpd/modules/mod_socache_shmcb.so LoadModule ssl_module lib/
httpd/modules/mod_ssl.so LoadModule http2_module lib/httpd/modules/mod_http2.so
```

3提供服务器名称（例如 localhost）：

```
#ServerName www.example.com:8080 ServerName
localhost
```

4在底部添加如下配置：

```
<IfModule http2_module>
        协议 h2 http/1.1 LogLevel http2:info
        </IfModule>
```

5设置 HTTPS 证书：

```
#Note 我将使用的 openssl 命令需要以 root 身份运行 su – cd /usr/local/etc/httpd cat /System/
Library/OpenSSL/openssl.cnf > /tmp/openssl.cnf echo    [SAN] \nsubjectAltName=DNS:localhost
>> /tmp/openssl.cnf openssl req \ -newkey rsa:2048 \ -x509 \ -nodes \ -keyout server.key \ -new
\ -out server.crt \ -subj /CN= localhost \ -reqexts SAN \ -extensions SAN \ -config /tmp/openssl.cnf
\ -sha256 \ -days 3650
```

6你应该很好地重新启动 Apache：

```
brew 服务重启 httpd
```

检查 https://localhost:8443/,您应该会看到通过 HTTP/2 提供的页面。

Machine Translated by Google

从源代码下载有点复杂,类似于 Linux 设置：

1为软件设置一个目录：

```
cd /tmp
mkdir 来源
CD 来源
```

2安装最新的 OpenSSL（或 LibreSSL,如果您愿意）：

```
#从http://openssl.org/source/获取 #例如:curl -O https://www.openssl.org/
source/openssl-1.1.1a.tar.gz curl -O https://www .openssl.org/source/
openssl-1.1.1a.tar.gz.sha256 #通过比较这两个值来验证下载后的包:openssl dgst -sha256 openssl-1.1.1a.tar.gz cat
openssl-1.1.1a。 tar.gz.sha256 #解压文件并编译:tar -zxvf openssl-1.1.1a.tar.gz cd openssl-1.1.1a ./config shared zlib-dynamic --
prefix=/usr/local/ssl
```

```
制作
须藤安装
光盘..
```

3安装nghttp2模块：

```
#下载并安装 nghttp2（mod_http2 需要）。
#从 https://nghttp2.org/ 获取 #最新版本在这里:https://
github.com/nghttp2/nghttp2/releases/ #例如:curl -O -L https://github.com/nghttp2/ nghttp2/releases/download/v1.33.0/
nghttp2-1.33.0.tar.gz tar -zxvf nghttp2-1.33.0.tar.gz cd nghttp2-1.33.0 ./configure 制作
```

```
须藤安装
光盘..
```

4获取apr、 apr-util和PCRE：

```
#下载并安装最新的apr #注意如果使用openssl 1.1.0则需要在APR
1.6或以上版本 #从http://apr.apache.org/获取 #例如:curl -O http://mirrors. whoishostingthis.com/apache/apr/apr-1.6.5.tar.gz
curl -O https://www.apache.org/dist/apr/apr-1.6.5.tar.gz.sha256 #验证包后下载 cat apr-1.6.5.tar.gz.sha256 openssl dgst
-sha256 apr-1.6.5.tar.gz #安装包:tar -zxvf apr-1.6.5.tar.gz cd apr-1.6.5 。 /配置
```

Machine Translated by Google

制作
须藤安装
光盘..

#下载并安装最新的apr-util #注意如果使用openssl 1.1.0则需要APR-UTIL
1.6或以上版本 #从http://apr.apache.org/获取 #例如:curl -O http: //mirrors.whoishostingthis.com/apache/apr/apr-util-1.6.1.tar.gz
curl -O https://www.apache.org/dist/apr/apr-util-1.6.1.tar。 gz.sha256 #下载后验证包:cat apr-util-1.6.1.tar.gz.sha256 openssl
dgst -sha256 apr-util-1.6.1.tar.gz #安装包:tar -zxvf apr-util -1.6.1.tar.gz cd apr-util-1.6.1 ./configure --with-apr=/usr/local/apr

制作
须藤安装
光盘..

#从版本 8 分支下载并安装最新的 PCRE
#note apache 仅适用于 PCRE 8 分支,不适用于 PCRE 10 #从 http://www.pcre.org/ 获取 #例如:curl -O https://
ftp.pcre.org/pub/pcre/pcre- 8.42.tar.gz #安装包:tar -zxvf pcre-8.42.tar.gz cd pcre-8.42 ./configure make

须藤安装
光盘..

5安装阿帕奇：

#下载并安装apache #例如:curl -O http://
mirrors.whoishostingthis.com/apache/httpd/httpd
2.4.37.tar.gz curl -O https://www.apache.org/dist/httpd /httpd-2.4.37.tar.gz.sha256 #下载后验证包： cat
httpd-2.4.37.tar.gz.sha256 openssl dgst -sha256 httpd-2.4.37.tar.gz #解压源码tar -zxvf httpd-2.4.37.tar.gz cd
httpd-2.4.37 ./configure --with-ssl=/usr/local/ssl --with-pcre=/usr/local/bin/pcre config --enable -ssl --enable-so --with-apr=/
usr/local/apr/bin/apr-1-config --with-apr-util=/usr/local/apr/bin/apu-1-config - -with-nghttp2=/usr/local/opt/nghttp2 --
enable-http2 make

须藤安装
光盘..

Machine Translated by Google

此代码应将 Apache 安装到 /usr/local/apache2。

6启动 Apache 以查看它是否适用于基本 HTTP：

```
sudo /usr/local/apache2/bin/apachectl -k 优雅
```

如果一切顺利，您应该能够通过 HTTP 访问您的站点并在 http://localhost 上看到默认的“It works”页面，如图 A.2 所示。

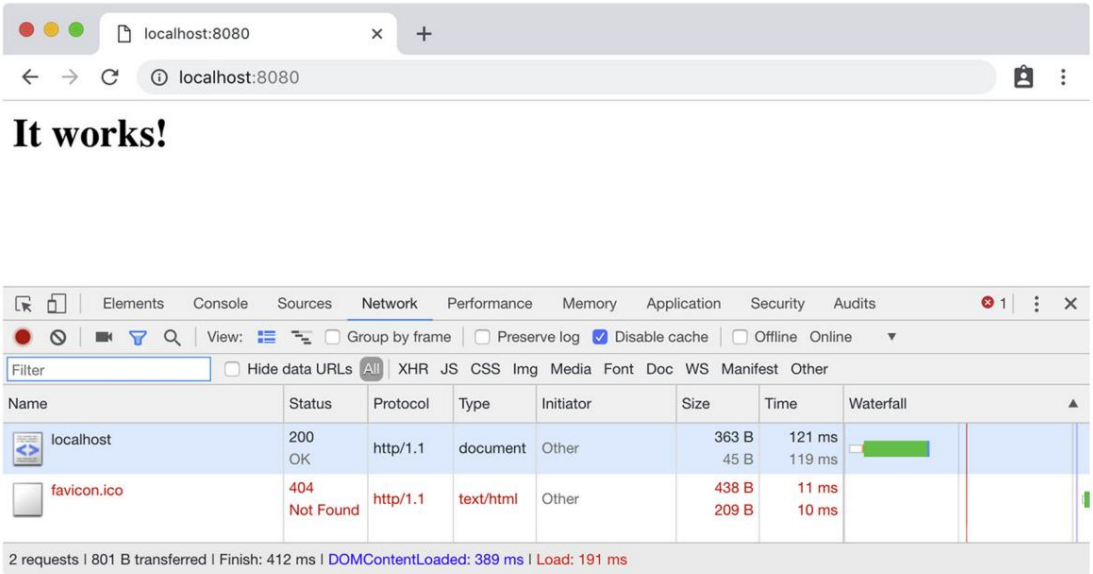


图 A.2 默认 Apache macOS 上的页面

让 HTTP/2 工作需要更多的步骤。

7从 httpd.conf 文件中取消注释以下模块以加载 SSL 和 HTTP 模块以及 SSL 所需的 socache 模块：

```
LoadModule socache_shmcb_module 模块/mod_socache_shmcb.so LoadModule ssl_module 模
块/mod_ssl.so LoadModule http2_module 模块/mod_http2.so
```

8添加此行以包含 SSL 配置：

```
包括 conf/extra/httpd-ssl.conf
```

9添加此行以表明此服务器优先使用 HTTP/2 (h2)，然后再使用 HTTP/1 并启用日志记录：

```
<IfModule http2_module>
    协议 h2 http/1.1 LogLevel
    http2:info </IfModule>
```

Machine Translated by Google

如果您想通过 HTTP 启用 HTTP/2（并且不需要 HTTPS），您可以将h2c添加到 Protocols 行，但此功能不受任何浏览器支持且用途有限。

10 接下来，安装 HTTPS 证书。获取证书超出了本书的范围，因此我将向您展示如何使用 OpenSSL 生成基本的自签名证书。浏览器不会识别此证书，但它可以用于一些基本测试：

```
#Note 我将使用的 openssl 命令需要以 root 身份运行 sudo su – cd /usr/local/apache2/conf cat /
System/Library/OpenSSL/openssl.cnf > /tmp/openssl.cnf echo  [SAN]
\nsubjectAltName=DNS:localhost  >> /tmp/openssl.cnf openssl req \ -newkey rsa:2048 \
-x509 \ -nodes \ -keyout server.key \ -new \ -out server.crt \ -subj /CN= localhost \ -reqexts
SAN \ -extensions SAN \ -config /tmp/openssl.cnf \ -sha256 \ -days 3650
```

您应该为您的服务器提供一个正确的主题和 SAN（在代码中的两个地方显示为 localhost），但是证书无论如何都不会被识别，所以这个信息不是太重要。

默认的 Apache 配置（在 conf/extra/httpd-ssl.conf 中）期望证书被称为 server.key 和 server.crt，但如果您使用的是非默认配置，请根据需要更改名称。

11 停止并重新启动 Apache 以使用以下 com 获取新配置
要求：

```
/usr/local/apache2/bin/apachectl -k 优雅
```

12 通过 HTTPS 访问站点，如果您使用的是步骤 10 中的自签名测试证书而不是真实证书，请忽略证书错误。您应该看到通过 HTTP/2 (h2) 加载的页面，类似于图 A.1，方法是打开 devel oper 工具并添加 Protocol 列。

A.1.2 nginx

nginx（读作 Eengine-X）在 1.9.5 版本中通过ngx_http_v2_module引入了 HTTP/2 支持，取代了ngx_http_spdy_module。我建议您运行最新稳定版本的 nginx（可在https://nginx.org/en/download.html 获得），

Machine Translated by Google

因为这个模块仍然相当新,并且在 1.9.5 和撰写本文时的当前版本 (1.14.2) 之间得到了积极改进。

nginx 不支持通过 HTTP/2 的后端连接（nginx 作为反向代理）,在撰写本文时,没有计划添加此支持。8 nginx 支持使用 NPN 和 ALPN 通过 HTTPS 的 HTTP/2。然而,由于 Chrome 仅支持 ALPN,因此最好使用至少 OpenSSL 1.0.2（或等效版本)构建 nginx,以便为所有浏览器启用 HTTP/2 支持。 nginx 允许通过纯文本 HTTP 连接（称为 h2c)的 HTTP/2,尽管此功能的用途有限,因为没有浏览器支持它。

WINDOWS上的NGINX

与 Apache 不同,nginx 在其下载页面上提供 Windows 版本。9要获得在 Windows 上运行的 HTTP/2 版本的 nginx,请执行以下操作：

1从下载页面下载最新的稳定版本。10 2将软件解压缩到您要安装的位置（例如 C:\Program Files\nginx）。

3以管理员身份启动命令提示符,然后启动 nginx.exe 可执行文件：

```
cd C:\Program Files\nginx 启动nginx.exe
```

如果您收到类似这样的任何错误,则另一台 Web 服务器已经在侦听端口 80：

```
nginx:[emerg] bind() 到 0.0.0.0:80 失败（10013:尝试以其访问权限禁止的方式访问套接字）
```

在 Windows 上,此服务器通常是万维网发布服务/IIS,因此进入服务并停止它。 Skype 可能会导致类似的问题。

此时,您应该能够通过 HTTP（但不是 HTTPS)查看默认的 nginx 页面,并且当您打开开发人员工具并添加协议列时,它应该通过 HTTP/1.1 提供服务。

4接下来,创建一些 HTTPS 证书。不幸的是,这个过程在 Windows 上有点棘手,因为与 Apache 不同,nginx 在默认的 Windows 下载中不包含任何虚拟证书。最简单的选择是使用 Linux 上的 Apache 部分中显示的命令在 Linux 服务器上生成证书。如果无法使用 Linux 服务器,请使用http://www.selfsignedcertificate.com/等在线服务来生成证书。这些证书只能用于测试服务器,不能用于生产服务器。将密钥保存在 conf 目录中名为 cert.key 的文件中,并将证书保存在名为 cert.pem 的文件中。

---

[8] https://trac.nginx.org/nginx/ticket/923
[9] https://nginx.org/en/download.html
[10] https://nginx.org/en/download.html

Machine Translated by Google

5通过编辑主配置文件（例如 conf/nginx.conf)来配置 nginx,取消注释 SSL 主机部分,并将http2添加到listen命令：

```
#HTTPS 服务器
#
服务器 {
    听                    443 SSL HTTP2;
```

6通过在命令中运行以下命令重新加载 nginx 配置线：

```
nginx -s 重新加载
```

7通过 HTTP/2 访问默认网站。如果您使用的是浏览器无法识别的自签名证书,您可能必须忽略证书错误。

NGINX ON LINUX FROM NGINX REPOS

nginx.org 为主要操作系统提供官方 nginx 存储库。[11]从版本1.12.2 开始,稳定的存储库是针对 OpenSSL 1.0.2 构建的，在操作系统支持的情况下具有 ALPN 支持,因此您可以按照以下步骤在 RHEL/CentOS 7.4 机器上安装 repo：

1通过使用 vi 等编辑器创建 repo 文件来安装 nginx repo：

```
sudo vi /etc/yum.repos.d/nginx.repo
```

2将配置添加到 nginx.repo 文件。

对于 RHEL 7（最低 7.4）,使用

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/rhel/7/$basearch/gpgcheck=0 enabled=1
```

对于 CentOS 7（最低 7.4）,使用相同的代码但更改baseurl如下：

```
baseurl=http://nginx.org/packages/centos/7/$basearch/
```

nginx 也有一个包含最新修复的主线版本,但我不建议在生产系统上安装它。

3安装 nginx：

```
sudo yum 安装 nginx
```

4启动 nginx：

```
sudo nginx
```

[11] https://nginx.org/en/linux_packages.html

Machine Translated by Google

5通过打开开发人员工具并添加协议列来检查通过 HTTP 加载的默认页面。

6要打开 HTTP/2,请添加 default_ssl.conf 文件：

```
vi /etc/nginx/conf.d/default_ssl.conf
```

7将以下内容添加到 default_ssl.conf 文件：

```
#HTTPS 服务器

# 服务器 {
        听                          443 SSL HTTP2;服务器名
        称本地主机；

        ssl_certificate cert.pem；  ssl_certificate_key 证书密
        钥；

        ssl_session_cache                       共享:SSL:1m；
        ssl_session_timeout 5m；

        ssl_ciphers HIGH:!aNULL:!MD5; ssl_prefer_server_ciphers
        开启；

        位置 / { root /usr/
                share/nginx/html; index index.html index.htm;

        }

}
```

8创建 HTTPS 证书：

```
#Note 我将使用的 openssl 命令需要以 root 身份运行 su - cd /etc/nginx/conf openssl req \ -newkey rsa:2048
\ -x509 \ -nodes \ -keyout cert.key \ -new \ - out cert.pem \ -subj /CN=server.domain.tld \ -reqexts SAN \
-extensions SAN \ -config <(cat /etc/pki/tls/openssl.cnf \
<(printf    [SAN]\nsubjectAltName=DNS:server.domain.tld    )) \ -sha256 \ -days 3650
```

用您的域名替换两个地方的server.domain.tld 。无论如何你都会收到一个证书错误，所以不要太担心这部分。

Machine Translated by Google

9重新加载 nginx 配置：

```
nginx -s 重新加载
```

10通过 HTTPS 加载页面，并通过打开开发人员工具并添加协议列确保它显示 HTTP/2（参见图 A.1）。

NGINX ON LINUX FROM SOURCE CODE

由于 nginx 的官方打包版本是针对支持它的平台（例如 RHEL/CentOS 7.4)的 OpenSSL 1.0.2 构建的，因此您不需要从源代码构建。如果您使用的是旧平台，请按照以下步骤操作：

1安装依赖项：

```
sudo yum -y 安装 wget sudo yum -y 安装 perl
sudo yum -y 安装 zlib-devel sudo yum -y 安装
gcc sudo yum -y 安装 pcre-devel sudo yum -y 安装 expat-
devel sudo yum -y 安装 epel-release sudo yum -y 安装
libnghttp2-devel sudo yum -y 安装 openssl-devel
```

```
cd /tmp
mkdir 来源
CD 来源
```

2对于7.4之前的RHEL/CentOS，同样从源码安装OpenSSL：

```
#安装Openssl http://openssl.org/source/ #例如:wget https://www.openssl.org/
source/openssl-1.1.1a.tar.gz wget https://www.openssl.org/ source/
openssl-1.1.1a.tar.gz.asc #下载后验证包 gpg --verify openssl-1.1.1a.tar.gz.asc #If you get a “Can t check
signature: No public key” error #then 获取相应的公钥并再次验证。 gpg --recv-keys 0E604491 gpg --verify
openssl-1.1.1a.tar.gz.asc #解压文件并编译:tar -zxvf openssl-1.1.1a.tar.gz cd openssl-1.1.1a ./配置共享 zlib-dynamic --
prefix=/usr/local/ssl make
```

```
须藤安装
光盘..
```

3下载并解压最新的稳定版nginx：

```
cd /tmp
mkdir 来源
CD 来源
#下载最新稳定版
```

Machine Translated by Google


```
#从https://nginx.org/en/download.html获取 #例如:wget https://nginx.org/download/
nginx-1.14.2.tar.gz wget https://nginx.org/下载/nginx-1.14.2.tar.gz.asc #验证下载：
#nginx keys are here: https://nginx.org/en/pgp_keys.html
```

```
#像这样全部安装:wget https://nginx.org/keys/
mdounin.key gpg –import mdounin.key #然后验证包:gpg --verify
nginx-1.14.2.tar.gz.asc #Note您将看到一条警告,指出该密钥未经过#with 可
信签名认证。这是意料之中的。
```

```
#解压文件并编译:tar -xvf nginx-1.14.2.tar.gz cd nginx-1.14.2
```

4配置制作脚本。

对于使用系统openssl的 RHEL/CentOS 7.4,请使用以下命令：

```
#配置并编译：./configure --with-
 http_ssl_module --with-http_v2_module
```

对于7.4之前的RHEL/CentOS,需要使用你安装的自定义OpenSSL：

```
#配置编译：./configure --with-
 http_ssl_module --with-http_v2_module \ --with-openssl=/tmp/sources/openssl-1.1.1a
```

5制作并安装构建：

```
制作
须藤安装
光盘..
```

6启动 nginx：

```
sudo /usr/local/nginx/sbin/nginx
```

7通过打开开发人员工具并添加协议列来测试基本 HTTP 站点（但不是 HTTPS）。

8通过编辑主配置文件（例如 /usr/local/nginx/conf/nginx.conf)为 HTTPS 和 HTTP/2 配置 nginx，确保取消注释 HTTPS 部分,并添加以粗体显示的http2选项：

```
#HTTPS 服务器

# 服务器 {
    听                    443 SSL HTTP2;
```

Machine Translated by Google

9创建 HTTPS 证书：

```
#openssl 命令需要以 root 身份运行,所以对那个 sudo su - cd /usr/local/nginx/conf openssl req \ -newkey
rsa:2048 \ -x509 \ -nodes \ -keyout cert.key \ -new \ -out cert.pem \ -subj /CN=server.domain.tld \
-reqexts SAN \ -extensions SAN \ -config <(cat /etc/pki/tls/openssl.cnf \
<(printf [SAN]\nsubjectAltName=DNS:server.domain.tld )) \ -sha256 \ -days 3650
```

您应该为您的服务器提供正确的主题和 SAN（在代码中的两个位置显示为server.domain.tld ）。但是无论如何证书都不会被识别,所以这部分不是太重要。

默认的 nginx 配置（在 conf/nginx.conf 中)期望证书被称为 cert.key 和 cert.pem,但如果您使用的是非默认配置,请根据需要更改名称。

10重新加载 nginx 配置：

```
sudo /usr/local/nginx/sbin/nginx -s reload
```

11通过 HTTP/2 查看默认的“欢迎使用 nginx”页面。

MACOS 上的 NGINX

在 macOS 上安装 nginx 有两种方法：

- 从另一个包管理器安装,例如Homebrew。
- 从源安装。

这两种选择都不是理想的（见第 3 章）。

通过 Homebrew 安装很简单,涉及两个命令：

```
/usr/bin/ruby -e $(curl -fsSL https://raw.githubusercontent.com/Homebrew/
安装/掌握/安装）”

酿造安装nginx
```

要检查是否有效,请启动 nginx：

```
brew 服务启动 nginx
```

Machine Translated by Google

Homebrew 使 nginx 通过端口 8080 和 8443 而不是通常的 80 和 443 可用,因此请访问 http://localhost:8080/ 以查看您的新 Web 服务器。

配置 HTTP/2 需要编辑主配置文件

/usr/local/etc/nginx/nginx.conf

并进行以下更改

1取消注释以下行以添加 HTTPS 服务器,并添加http2
监听线的指令:

```
#HTTPS 服务器#

服务器 {
        听                          8443 SSL HTTP2;服务器名
        称本地主机;

        ssl_certificate cert.pem;  ssl_certificate_key 证书密
        钥;

        ssl_session_cache                    共享:SSL:1m;
        ssl_session_timeout 5m;

        ssl_ciphers HIGH:!aNULL:!MD5; ssl_prefer_server_ciphers
        开启;

        地点 / {
                根 HTML;  index
                index.html index.htm;
        }

}
```

2设置 HTTPS 证书:

```
#Note 我将使用的 openssl 命令需要以 root 身份运行 sudo su – cd /usr/local/etc/nginx cat /System/Library/
OpenSSL/openssl.cnf > /tmp/openssl.cnf echo    [SAN] \nsubjectAltName=DNS:localhost    >> /tmp/
openssl.cnf openssl req \ -newkey rsa:2048 \ -x509 \ -nodes \ -keyout cert.key \ -new \ -out cert.pem \
-subj /CN= localhost \ -reqexts SAN \ -extensions SAN \ -config /tmp/openssl.cnf \ -sha256 \ -days 3650
```

Machine Translated by Google

3重启 nginx：

```
brew 服务重启 nginx
```

4在 https://localhost:8443/ 检查站点是否通过 HTTP/2 加载。

如果您不想使用 Homebrew，从源代码下载会更复杂一些，并且类似于 Linux 设置：

1为软件设置一个目录：

```
cd /tmp
mkdir 来源
CD 来源
```

2安装最新的 OpenSSL（或 LibreSSL，如果您愿意）：

```
#从http://openssl.org/source/获取 #例如：curl -O https://www.openssl.org/
source/openssl-1.1.1a.tar.gz curl -o https://www .openssl.org/source/
openssl-1.1.1a.tar.gz.sha256 #通过比较这两个值来验证下载后的包：openssl dgst -sha256 openssl-1.1.1a.tar.gz cat
openssl-1.1.1a。 tar.gz.sha256 #解压文件并编译：tar -zxvf openssl-1.1.1a.tar.gz cd openssl-1.1.1a ./config shared zlib-dynamic --
prefix=/usr/local/ssl
```

```
制作
须藤安装
光盘..
```

3安装 nginx：

```
#下载安装nginx #例如：curl -O https://nginx.org/
download/nginx-1.14.2.tar.gz #解压源码 tar -zxvf
nginx-1.14.2.tar.gz cd nginx -1.14.2 ./configure --with-http_ssl_module --with-http_v2_module \ --with-
cc-opt=   -I/usr/local/ssl/include    \ --with-ld-opt=   -L /usr/local/ssl/lib
```

```
制作
须藤安装
光盘..
```

4以 root 身份启动 nginx。请注意，从源安装时，您使用默认端口 80 和 443，这需要 root 访问权限才能使用：

```
sudo /usr/local/nginx/sbin/nginx
```

5通过打开开发人员工具并添加协议列来测试基本 HTTP 站点（但不是 HTTPS）。

Machine Translated by Google

6通过使用文本编辑器编辑主配置文件来配置 HTTP/2
比如vi：

```
sudo vi /usr/local/nginx/conf/nginx.conf
```

还要取消注释以下行以添加 HTTPS 服务器，并将http2指令添加到侦听行：

```
#HTTPS 服务器

# 服务器 {
        听                          443 SSL HTTP2;服务器名
        称本地主机；

        ssl_certificate cert.pem； ssl_certificate_key 证书密
        钥；

        ssl_session_cache                    共享:SSL:1m；
        ssl_session_timeout 5m；

        ssl_ciphers HIGH:!aNULL:!MD5; ssl_prefer_server_ciphers
        开启；

        地点 / {
                根 HTML； index
                index.html index.htm;
        }

}
```

7设置 HTTPS 证书：

```
#Note 我将使用的 openssl 命令需要以 root 身份运行 sudo su – cd /usr/local/etc/nginx cat /System/Library/
OpenSSL/openssl.cnf > /tmp/openssl.cnf echo    [SAN] \nsubjectAltName=DNS:localhost    >> /tmp/
openssl.cnf openssl req \ -newkey rsa:2048 \ -x509 \ -nodes \ -keyout cert.key \ -new \ -out cert.pem \
-subj /CN= localhost \ -reqexts SAN \ -extensions SAN \ -config /tmp/openssl.cnf \ -sha256 \ -days 3650
```

8重新启动 nginx 以获取新配置：

```
sudo /usr/local/nginx/sbin/nginx -s reload
```

9在 https://localhost/ 检查站点是否通过 HTTP/2 加载。

Machine Translated by Google

A.1.3 Microsoft Internet 信息服务 (IIS)

在 Windows Server 2016 和 Windows 10 中引入的 IIS 10 中添加了 HTTP/2 支持。IIS 仅通过 HTTPS 支持 HTTP/2。 IIS 10 默认启用 HTTP/2,因此如果您运行的是带有 HTTPS 的 Windows Server 2016 或更高版本,并且您有一个 HTTPS 站点,则可能已经在使用 HTTP/2。如果您的服务器早于 Windows Server 2016,唯一的选择是升级整个服务器。无法在较旧的 Windows 计算机上安装支持 HTTP/2 的 IIS 10。

对于 Windows 10 台式机,您可能需要在“打开或关闭 Windows 功能”屏幕（可从“开始”选项搜索 Windows 功能获得)上启用 IIS 管理控制台,如图 A.3 所示。

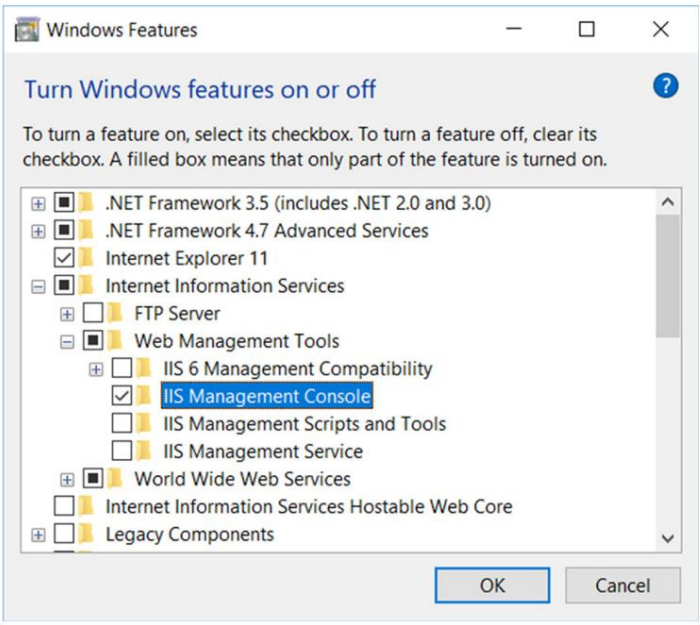


图 A.3 启用 IIS 管理控制台

启用此控制台后,您应该能够像往常一样在以下位置配置站点:控制面板\所有控制面板项目\管理工具\Internet 信息服务 (IIS) 管理器。

安装 HTTPS 证书（此处未介绍)后,您应该能够
通过 HTTP/2 加载默认网站。

Machine Translated by Google

A.1.4 其他服务器

可以以类似的方式安装其他服务器。重要的一点是检查 HTTP/2（以及它被添加到哪个版本），检查它是用哪个版本的 OpenSSL（或类似的)构建的,并启用 HTTP/2 和 HTTPS。 HTTP/2 实现页面 12在撰写本文时列出了 50 多个支持 HTTP/2 的服务器。

A.2 通过反向代理服务器设置 HTTP/2

如果无法升级您的主 Web 服务器以支持 HTTP/2,或者如果您想在不对当前设置进行任何更改的情况下测试 HTTP/2,您可能需要在您的 Web 服务器前面设置一个反向代理以启用 HTTP/2 支持。在接下来的部分中,我提供了 Apache 和 nginx 的基本说明。

在这两个示例中,反向代理都使用 HTTP/1.1 与后端服务器通信。 Apache 允许通过 mod_proxy_http2模块使用 HTTP/2。在撰写本文时,该模块仍被标记为实验性的,13并且此时通过 HTTP/2 代理后端连接几乎没有什么好处（请参阅“您需要一直说 HTTP/2 吗？”章节中的侧边栏3),因此这里不涉及该用例。以下示例允许 Apache 或 nginx 处理 HTTP/2 通信并让后端服务器使用 HTTP/1.1。

A.2.1 阿帕奇

要让 Apache 充当代理服务器,您需要启用 HTTP/2,如前一节所述,并在主配置文件中启用以下模块：

```
proxy_module
proxy_http_module
```

您可以通过在主配置文件（httpd.conf 或 apache.conf)中取消注释或添加适当的Load Module行或通过对基于 Ubuntu 的系统使用a2enmod来启用这些模块。

然后添加代理配置（假设您要代理的 Web 服务器在本地主机端口 8080 上）：

```
ProxyPreserveHost 开启

# 将所有请求代理到 localhost 端口 8080 ProxyPass / http://127.0.0.1:8080/
ProxyPassReverse / http://127.0.0.1:8080/
```

此代码使用 Ipv4 环回地址 (127.0.0.1) 将请求直接传递到后端服务器。如果您愿意,也可以使用 Ipv6 环回地址 (::1)。这两个选项都优于本地主机名,后者需要进行不必要的 DNS 查找。

---

[12] https://github.com/http2/http2-spec/wiki/Implementations
[13] https://httpd.apache.org/docs/trunk/mod/mod_proxy_http2.html

Machine Translated by Google

该应用程序可能需要配置为与前面的代理服务器一起使用。
例如,它生成的任何链接都应该引用代理端口 (80/443),而不是它运行的实际端口（本例中为 8080）。由于反向代理应用程序服务器很常见,因此许多应用程序可以使用基本 URL 或类似选项轻松配置此服务器。如果后端服务器没有提供这个选项,Apache 允许你使用proxy_html_module动态重写 HTML 来自动替换链接（比如把http://www.example.com:8080替换成https://www.example. com）。

A.2.2 nginx

nginx 的工作方式与 Apache 类似,配置如下：

```
位置 /
        { proxy_pass http://127.0.0.1:8080/;
}
```

类似于 Apache 部分的评论,应用程序服务器可能需要配置以告诉它它在反向代理后面运行。如果此配置不可行,则ngx_http_sub_module可以动态重写任何 URL,类似于 Apache 的proxy_html_module 。

Machine Translated by Google

Machine Translated by Google

指数

符号

?特点。请参见查询参数

数理

103 早期提示状态159–160、162、172、178–179、209
200 响应代码110
304（未修改)响应代码89、202
404（未找到)状态码7、174

一个

接受标头100、250接受编码标头101、255
高级 REST 客户端应用程序33
AJAX（异步 JavaScript 和 XML） 38种算法,拥塞控制301–302
ALPN（应用层协议协商）105-106的107-109概述的HTTPS 握手

替代服务333
ALTSVC 帧112、139–140
Alt-Svc 标头112、325反模式67–68
阿帕奇之家348
阿帕奇休息室348
Apache Web 服务器347-361
在 Linux 上350-356在macOS 上356-361在Windows 上347-349通过反向代理服务器设置 HTTP/2 372-373

应用层协议协商。将ALPN 视为属性147、173
非对称加密103异步属性186异步/等待代码156

异步 JavaScript 和 XML (AJAX) 38自动化服务器推送172

乙

带宽191
BBR（瓶颈带宽和往返支柱agation time) 301–302二进制框架层341二进制协议335阻塞，HOL 330主体压缩255–257瓶颈带宽和往返传播时间。请参阅BBR
Brotli 压缩算法198浏览器开发人员工具31-33 HTTP/2 72-73 HTTPS 72-73服务器推送163-167推送缓存163-166拒绝使用 RST_STREAM 推送166-167
流优先级241-245支持对于70–74

C

缓存摘要169–170、334缓存控制标头27、164、166–167、202–205 CACHE_DIGEST 帧140–141、169缓存。另请参见推送缓存27、48、202–205
Caddy 310、314 CA（证书颁发机构） 29、324、346

Machine Translated by Google

CDN（内容分发网络） 50、78、84–85、146、183、246 CentOS 350
证书颁发机构。请参见CA CertificateRequest SSL/TLS 消息103–108 CGI（通用网关接口） 38 ChangeCipherSpec SSL/TLS 消息103–108 Chrome 开发人员工具155 ClientCertificate SSL/TLS 消息103–108 ClientHello SSL/TLS 消息103–108、119 客户端检测 HTTP /2 客户端支持217 HPACK 标头压缩275–277 Cloudflare 277、340 CMS（内容管理系统） 38合并连接218–220 CodeIt 79通用网关接口 (CGI)。请参阅CGI 压缩标头，另请参阅 HPACK 100–101、249–277更大的资源189–191回溯压缩254–255 概述251–255编码技术252–254查找表252
文本数据197–198
压缩比信息泄漏变得容易
（犯罪） 257一致
性测试245-247客户端247服务器245-247
拥塞控制算法301-302 TCP 中的低效率
284-293空闲连接288-289数据包丢失
289-291数据包丢失导致项目排队291-293
拥塞窗口（CWND) 45, 230

CONNECT 方法341–344 个连接

合并218-220建立101-113
HTTP 替代服务112 HTTP/2
前言消息113使用 HTTP 升级标头109-112
使用 HTTPS 协商102-109使用先验知识112
空闲288-289持续24-27 QUIC 311-312从 HTTP 升级
343–345竞争资源191内容交付网络 (CDN)。请参阅CDN
内容管理系统 (CMS)。请参阅CMS 内容编码 HTTP 标头
198
内容长度 HTTP 标头25内容安全策略 (CSP)
27、48、251、337 CONTINUATION 帧132、137、229、313 cookie
264、320–323 cookie 标头100、250、276推送基于168–169 已发
送每个请求323 COPY 方法336 CRIME。请参阅压缩比信息泄漏
Easy
crossorigin 属性147密码库76
CSP。参见Content-Security-
Policy CSS（层叠样式表） 7 CUBIC 算法
301 CWND。查看拥塞窗口

丁

数据
最小化传输量194-201压缩文本数据197-198压缩代码
199-201重新发送202-205只发送必要的数据206-207
数据帧134-135, 150, 160, 232, 247延迟属性186基于
Deflate 的算法255删除方法22、259依赖项 235–237摘要。
请参阅缓存摘要 DNS（域名系统） 5、42、340 dns-
prefetch 提示207–208 DoH （基于 HTTPS 的 DNS） 340
DOM（文档对象模型） 7域名系统 (DNS)。请参见DNS 域分
片44 个域验证 (DV) 证书29个下游系统151–155个虚拟流239
个重复条目276个DV（域验证)证书。请参见域验证 (DV) 证书

乙

边缘服务器162、213编码。另
见霍夫曼编码252–254加密28 HTTP 和324–327 HTTP/2 和325–
327机会主义325加密密钥30 END_HEADERS 标志131
END_STREAM 标志137、227

Machine Translated by Google

EPEL（企业 Linux 的额外软件包） 350
ERR_SPDY_PROTOCOL_ERROR 消息89，
131
etag HTTP 标头168–169 eTag 值
203 EV（扩展验证)证书29常青浏览器
74–75过期标头27扩展验证 (EV) 证书。请参阅企业
Linux 的 EV（扩展验证)证书额外包。见EPEL

F

FEC（前向纠错） 304获取属性147首次绘制时间
41流量控制229-234, 328 230-232概述的示例
99-100服务器上的设置234字体属性147格式337
前向纠错。请参阅FEC 帧114–141检查消息流122–
136 121–122的格式查看114–120选择要使用
的工具120 chrome net-internals 114–116
使用 nghttp 116使用 Wireshark 116–119

G

Gartner 炒作周期63
GDPR（通用数据保护条例） 322通用传输协议，HTTP 为339–345
使用 HTTP 语义和消息传递非 Web 流量339–340
HTTP/2 二进制框架层341以 HTTP 341-345
开始协议
GET方法10、14、17、249
GOAWAY 框架135–136
Google 和 SPDY 319–320 gQUIC
309–310 gRPC 协议（gRPC 远程过
程
来电） 341
压缩198, 200

H

H2PushDiarySize 指令167、170
H2PushResource 指令 155、158半关闭流状态
227
握手，HTTPS
103-105与 ALPN 107-109
的概述
头部方法17
头部标签153
Header Block Fragment field 131, 137
headers 337 compressing 100–101
HPACK头压缩249-277 249-251的优点268-275的
例子
HPACK 动态表259
HPACK 静态表258–259
HTTP 主体压缩255-257在客户端实现中
275-277在服务器实现中275-277压缩概述
251-255 277的值
HPACK 头类型259–264文字头字段从未
索引263–264文字头字段表示259–260带有增量索引的文字
头字段260–262没有索引的文字头字段262–263
HTTP 链接标头146–148
HTTP 升级标头109-112发送问题111-112服
务器建议的升级110-111成功的升级请求
110不成功的升级请求109链接标头151-155强制
主机标头23-24请求标头18-19响应标头21-22
标题帧129、132–133、150、160、227、271
HOL（线头)阻塞40、95、292、297、313、330主机头23–24、100
HPACK动
态表259头压缩249-277
优点249-251实例268-275
霍夫曼编码265-268在客户端实现中
275-277在服务器实现中275-277概述251-255
277头类型的值259-264
文字头字段从不索引263–264文字头字段表示259–260带有
增量索引的文字头字段260–262没有索引的文字头字段262–
263静态表258–259

Machine Translated by Google

HSTS（HTTP 严格传输安全） 27、337
HTTP（超文本传输协议）
替代服务112作为通用传输
协议339–345主体压缩255–257条件请求168
CONNECT 方法341–343加密和
324–327在传输层之上演进335–
338格式337未来302–303标头337历史15–27
如何引入未来版本339链接标头146–148消息
339–340方法336多重连接44–46不再无状态
330–331概述9–15隐私问题320–324 cookie
321–322
HTTP cookies 322-323第
三方追踪321-322资源提示
207-209
dns 预取提示207–208预连接
提示208预取提示208预加载提
示208–209预呈现提示209
语义339-340创建连接
时的设置延迟283-284启动协议341-345状态代码
338升级标头109-112
发送问题111–112服务器建议的
升级110–111成功的升级请求110不成功
的升级请求109从343–345升级连接
HTTP 存档36 HTTP
严格传输安全。请参阅HSTS HTTP WG（HTTP
工作组） 16、318 HTTP/0.9 15–16 HTTP/1.0 16–
22 HTTP/1.1 22–27问题38–40、48–49强制主机标
头23–24移至 HTTP/2 56–59优化连接合并218–220
在客户端检测 HTTP/2 支持217在服务器端检测
HTTP/2 支持214–217
需要多长时间220
测量 HTTP/2 流量213–214持久连接24–
27流水线处理40性能问题的解决方法43–48
减少请求46–48多个 HTTP 连接44–46万维
网和36–43 HTTP/2 优势通过 HTTP/1.1 94–101二
进制而非文本95–96流量控制99–100标头压缩
100–101多路复用而非同步96–99服务器推送
101流优先级99–100作为权宜之计332–333 331–
332差异的复杂性在 QUIC 和311-313之间支持网站
78-85加密和325-327浏览器72-73限制187-189对
Web 开发人员意味着183从 HTTP/1.1 迁移
56-59在 Web 服务器上78-80性能提升62 –67个
真实示例333–334通过反向代理服务器设置372–
373 Apache 372–373 nginx 373通过 CDN 84–
85用于331故障排除设置的工具85–89升级
到69–89将 Web 服务器升级到346–373
阿帕奇347–361
Microsoft IIS（因特网信息
服务） 371
nginx 361–370带反
向代理80–84
HTTP/2 推送缓存165 HTTP/
3。另见QUIC 305–316、328–330、334–335、338–
339、343、345
http2 选项 118、161、366 $
http2 变量215
HttpOnly 标志256、322–323
用于浏览器的 HTTPS（安全超文本传输协议） 72–
73握手概述103–105与 ALPN 107–109库76–77

Machine Translated by Google

HTTPS（安全超文本传输协议） （续)协商102–109 ALPN
（应用层协议协商） 105–106 NPN（下一个协议协
商） 106–107优化209–212概述28–31 QUIC 和311 HTTP
设置标头109霍夫曼编码253、261、263 268脚本的缺点
266–267表265–266超文本15超文本标记语言
(HTML) 4超文本传输协议。请参阅安全的 HTTP 超文本
传输协议。请参见HTTPS

I

IANA（互联网号码分配机构） 21，
137
ID 参数320空闲流状态227
IETF（互联网工程任务组）
9、28、58、180、309、320 if-none-match 标头204 IIS（互联网信息服
务） 371图像属性147图像缓存165索引增量260– 262文字标头
字段从未编入索引263–264没有索引的文字标头字段262–263
initialWindowSize 设置234内联193–194拦截代理73–74互联网号码
分配机构。请参阅IANA 互联网工程任务组。请参阅IETF 互联网信息服
务。参见IIS

物联网 (IoT) 3、327互联网协议。请参阅IP 互
联网堆栈305–306互联网与万维网4物联网。
请参阅物联网 IP（互联网协议） 4、306–307
iQUIC 309–310 ISP（互联网服务提供商）
28

杰

Java Servlet/Java 服务器页面 (JSP) 38
Jetty（Java servlet 引擎） 172

钾

保活24–27
KISS（保持简单愚蠢） 332

大号

延迟36、40、46、209
第 4 层负载均衡器214分层违规327–
330
Let s Encrypt 210–211、326
个库76–77个链接标头

HTTP 146-148从下游
系统推送151-155

LINK 方法22 <link> 元
素207
操作系统
Apache on 350–356
nginx on from nginx
repos 363–365 from source
code 365–367 listen
command 363 literal header
fields never indexed 263–264
representation 259–260 with
incremental indexing 260–262
without indexing 262–263

LiteSpeed 59, 310, 314
live-live 负载均衡器81 live-
standby 负载均衡器81负载均衡器
214
本地存储168
锁定方法336
LogFormat 指令213、215回溯压缩254–
255、264查找表252有损压缩251

米

操作系统
Apache on 356–361 nginx
on 367–370强制主机头23–
24最大段大小 (MSS) 285消息流122–136

数据框134–135
GOAWAY 框架135–136
标题帧129–133
优先帧128–129
设置帧124–127
WINDOW_UPDATE 帧127–128

Machine Translated by Google

消息
HTTP 339–340接
收31–34发送31–
34查看31–34

另见前言消息
微软 IIS（互联网信息服务）。请参阅互联网信息服务
(IIS) 缩小代码199–201 mod_http2 模块347
mod_proxy_http2 模块347、349、372 Mozilla SSL 配
置生成器88、211 MPM（多处理模块） 350 MSS。查看最
大分段大小多路复用流57多路复用96–99、328

优化182-220带宽限制
191压缩更大的资源189-191
HTTP/2 请求的成本184-186 HTTP/1.1 和
HTTP/2 连接合并218-220检测客户端的 HTTP/
2 支持217检测 HTTP/2服务器端支持214-217
为 HTTP/1.1 用户优化多长时间220测量
HTTP/2 流量213-214 HTTPS 209-212内联193-194
HTTP/2 的限制187-189资源争用191分片192-193
TCP 允许窗口缩放299 BBR 301–302拥塞控制
算法301–302禁用慢启动重启299增加初始 CWND 大
小298 PPR 301–302 SACK 299 TCP 快速打开299–
301升级操作系统297–298

否

协商,HTTPS 102–109
ALPN（应用层协议
谈判） 105–106
NPN（下一个协议协商） 106-107 net-internals/
net-export (Chrome) 114-116
下一个协议协商 (NPN) 76、106–107 nextHopProtocol
属性217 nghttp 库116 nghttp2 模块358 nginx 服务
器361–370在 Linux 上

来自 nginx repos 363–365来
自源代码365–367 on macOS
367–370 on Windows 362–363
通过反向代理服务器设置 HTTP/
2 373 ngx_http_spdy_module 361
ngx_http_sub_module 373
ngx_http_v2_module 361非网络流量

使用 HTTP 消息传递339-340使用 HTTP 语义
传递339-340 nopush 属性176-177, 209 NPN。
见下一个协议协商

网络性能技术194–212
缓存以防止重新发送数据202-205 HTTP 资源提
示207-209最小化传输的数据量194-201与 HTTP
无关212减少最后一英里延迟209仅发送必要的数
据206-207服务工作者206 ORIGIN 帧140,
219来源服务器24、84 OSI。请参阅开放系统互连模
型,否则不受信任的证书174 OV（组织验证)证书
29

欧

八位字节
122 onload 事件,JavaScript 8、64
不透明数据字段138打开流
状态227
开放系统互连 (OSI) 模型9 openssl 命令346操作系
统,升级297–298机会加密325

P

包管理器79数据包丢失
292导致项目排队291-293
289-291概述

填充长度字段134父流
236部分回溯压缩264持
久连接24–27

PING 帧137–138流水线
40
POST 方法17预连接
提示208

Machine Translated by Google

前言消息113预取提示208预
加载缓存165预加载提示208–
209预加载链接头147预加载
与服务器推送176–179预呈
现提示209 PRI 方法113先验知
识112优先级帧128–129、231、239、243隐
私问题320– 324 cookies 321–322 HTTP
cookies 322–323第三方跟踪321–322
PROPFIND 方法336按比例降低费率。参见
PRR PROTOCOL_ERROR 消息229协议93–141建立连
接101–113 HTTP 替代服务112 HTTP/2 前言消息113
使用 HTTP 升级标头109–112使用 HTTPS 协商
102–109使用先验知识112帧114–141检查消息流
122– 121-122的136格式查看114-120从 HTTP
341-345代理开始。另请参见反向代理服务器24、73–74
proxy_html_module 373 PRR（按比例降低速率）
301–302伪标头130、132、250公钥密码学30、103推送缓
存163–166推送。查看服务器推送 PushDiary 148
PUSH_PROMISE 框架 137–138、150、157、160、
173、287 PUT方法22 PuTTY 11

问

QPACK 312–313查询
参数18, 48排队项291–293

QUIC 303-316作为
HTTP/3 334-335选择使用
315-316 HTTP/2 和311-313建
立连接的区别311-312

HTTPS 和311实现
314-315互联网堆栈和305-306
304–305标准化的性能优势309–311 gQUIC
309–310 iQUIC 309–310 310–311工具的标
准313–314 UPD 和306–308 UDP 的优
势307–308 IP 与 UPD 307 SCTP 307
TCP 306

R

原始 HTTP 请求33重新缓冲
304接收消息31–34 Red Hat
Enterprise Linux。请参阅RHEL
Red Hat 软件集合。请参见RHSCL 引用标头172
REFUSED_STREAM 代码166 rel=preload 属性147
repos 363–365征求意见。查看RFC 请求标头18-19请
求46-48条件168 cookie 发送323成本184-186发送
33-34成功升级请求110不成功升级请求109查看34重
新发送数据202-205保留流状态227资源,竞争191响应
代码19-21响应标头21-22返回字符（作为 HTTP 行结
尾） 10、18个反向代理服务器183 HTTP/2 80-84通过
372-373 Apache设置 HTTP/2 372-373 nginx 373
RFC（请求评论) 16, 56 RHEL (Red Hat Enterprise
Linux) 350 RHSCL (Red Hat Software
Collections) 351 RST_STREAM frame 228
overview of 138–139 refusing pushes with 166–
167

小号

SACK（选择性确认） 299
SameSite 属性323

Machine Translated by Google

SAN（主题替代名称） 174可缩放矢量图。请参阅SVG 缩放299 s_client 命令31脚本属性147 <script> 标记51、143 SCTP（流控制传输协议） 307安全标志322–323安全套接字层。查看SSL 安全加密29安全性211、251自签名证书174、346语义339–340发送 HTTP 升级标头111–112消息31–34仅必要数据206–207请求33–34服务器推送101、142–146、162 –181自动化172选择要推送的资产170–172在浏览器中163–167概述142–145性能影响175–176与176–179有条件地推送167–170在服务器端跟踪推送167–168使用缓存摘要169 –170使用基于 cookie 的推送168–169使用 HTTP 条件请求168提前推送155–160使用链接头从下游系统推送151–155 使用RST_STREAM拒绝166–167故障排除173–174使用案例179–181使用HTTP 链接头146–148查看148–150ServerCertificate SSL/TLS 消息103–108 ServerHello SSL/TLS 消息103–108、119 ServerHelloDone SSL/TLS 消息103–108 ServerKeyExchange SSL/TLS 消息103–108 $server_protocol 变量213会议恢复105会话存储168设置帧110、124–127、227、259SETTINGS_ENABLE_PUSH 选项126、170SETTINGS_HEADER_TABLE_SIZE 值126，259SETTINGS_INITIAL_WINDOW_SIZE 126SETTINGS_MAX_CONCURRENT_STREAMS 126，187 SETTINGS_MAX_FRAME_SIZE 126SETTINGS_MAX_HEADER_LIST_SIZE 126 sharding 192–193 Shimmercat 242 Signed HTTP Exchanges 171 slow start restart 299 slow-start algorithm 45，287 SPDY 56–57 as option for HTTP/2 318–319disadvantages of 318–320 Google和319–320速度指数64 spriting 46 SSE（服务器发送的事件） 146、179SSL（安全套接字层）。另请参阅HTTPS 28SSLKEYLOGFILE 环境文件117 SSLLabs 86 Stale-While- Revalidate 205开始呈现41状态代码20–21、338流控制传输协议。请参阅SCTP 流标识符227

服务器

在214–217上检测 HTTP/2 支持HPACK header compression 275-277setting flow control on 234 support for 75-77 upgrades provided by 110-111

另请参阅Web 服务器服务器发送的事件 (SSE) 146、179service workers 165、206

溪流

依赖项235-237 99-100的优先级， 234-245在浏览器中 241-245在Web 服务器中241-245状态226-229权重238-240 Strict-Transport-Security 标头。请参见HSTS (HTTP Strict-Transport-Security) 样式属性147 <style> 标签47、143 Subject Alternative Name。查看SAN 支持69–77不支持时回退77用于浏览器70–74 HTTP/2 用于浏览器72–73HTTPS 用于浏览器72–73拦截代理73–74用于服务器75–77在客户端217在服务器端214–217 SVG (Scalable Vector Graphic) 47同步 (SYN) 消息45

Machine Translated by Google

吨

桌子
HPACK 动态表259
HPACK 静态表258–259
霍夫曼编码265–266查找表252

TCP（传输控制协议） 6、281–316拥塞控制效率低下284–293空闲连接288–289数据包丢失289–291数据包丢失导致项目排队291–293效率低下的影响293–296
快速开放299-301未来302-303 HOL（队首)阻塞330效率低下282-303优化允许窗口缩放299 BBR 301-302拥塞控制算法301-302禁用慢启动重启299增加初始 CWND 大小298
PRR 301–302 SACK 299升级操作系统297–298 TCP Fast Open 301测试一致性245–247客户端247服务器245–247文本数据压缩197–198基于文本的协议95第三方 cookie 321第三方跟踪321 –322 生存时间。请参见TTL TinyPNG 47 TLS（传输层安全)协议。另请参阅HTTPS 28、30、211跟踪推送167–168流量
测量213–214非网络流量
使用 HTTP 消息传递339-340使用 HTTP 语义传递339-340尾随标头133传输数据194-201压缩文本数据197-198缩小代码199-201传输控制协议。请参阅TCP 传输层安全协议。参见TLS 传输层,在 HTTP 之上演进335–338新格式337新 HTTP 标头337新 HTTP 方法336新状态代码338
传输协议,问题327–331
HTTP 不再无状态330–331分层违规327–330
TCP（传输控制协议)HOL（线头)阻塞330用于 HTTP/2的工具331
另请参阅通用传输协议,HTTP 作为故障排除
HTTP/2 设置85–89服务器推送173–174
TTL（生存时间） 207

ü

统一资源标识符。请参见URI UNLINK 方法22
UPD（用户数据报协议） 307-308 IP 与307
QUIC 和306-308 SCTP 307 TCP 306的优势
升级标头,HTTP 109-112发送问题111-112
服务器建议的升级110-111成功的升级请求110不成功的升级请求109
upgrade-insecure-requests 211从HTTP 343-345操作系统升级连接297-298成功请求110服务器建议110-111
到 HTTP/2 69-89不成功请求109 web 服务器到 HTTP/2 346-373 Apache 347 –361
Microsoft IIS 371 nginx 361–370通过反向代理服务器设置 HTTP/2 372–373 URI（统一资源标识符） 18使用相关视图70用户数据报协议。请参阅UPD 用户空间307用户代理标头100、250、273

V

可变长度编码253
清漆331观察框
114–120
选择使用哪个工具120 chrome net-internals 114–116

Machine Translated by Google

使用 nghttp查看（续）
116使用 Wireshark
116–119消息31–34请求34服务
器推送148–150虚拟主机23

W

瀑布图41-43网络和 HTTP/1.1
36-43互联网与4 3-9网络浏览器
的概述。请参见浏览器 Web
超文本应用技术工作组。请参
阅WHATWG Web 性能59–
68、194–212缓存以防止重新发
送数据202–205 HTTP 资源提示207–209 dns 预取提示207–
208预连接提示208预取提示208预加载提示208–209预
呈现提示209最小化数据量传输194-201压缩文本数据
197-198压缩代码199-201非 HTTP 相关技术212使用
HTTP/2 的性能提升62-67减少最后一英里延迟209只发送
必要的数据206-207服务工作者206瀑布图测量41–43
Web 平台孵化器社区组。看
Web 服务器
HTTP/2 on 78-80流优
先级241-245升级到 HTTP/2
346-373 Apache 347-361 Microsoft
IIS（互联网信息服务） 371 nginx
361-370通过反向代理服务器设置 HTTP/2
372- 373 WebDAV（Web 分布式创作
和版本控制） 336 WebPagetest 42、293个
网站，为78-85个WebSockets启用 HTTP/2
145、179、341加权流238-240 WHATWG
（Web 超文本应用技术工作组） 166 WICG（Web
平台孵化器社区组） 335窗口缩放299
Windows 操作系统 Apache 347–349 nginx 362–
363 WINDOW_UPDATE 框架125、127–128、230–
231、233、287、313 Wireshark 分析器116–119
Wireshark 框架269万维网。见网页

X

X-Content-Type 标头27
X-Frame-Options 标头27
XHR 轮询289 x-
http2-push-only 属性147
X-XSS-保护标头27

Machine Translated by Google

WEB开发

HTTP/2 正在行动

巴里波拉德

HTTP 超文本传输协议，是标准用于在网站之间交换消息和浏览器。 20 年后，它得到了急需的升级。通过对流、服务器推送、标头压缩和优先级的支持， HTTP/2在速度、安全性和效率方面提供了巨大的改进。

HTTP/2 in Action教您有效使用HTTP/2所需的一切知识。您将了解如何使用框架、多路复用和推送等新功能优化 Web 性能。您还将探索有关流控制和依赖关系等高级主题的真实示例。凭借现成的实施技巧和最佳实践，这本实用指南一定会让您和您的网站快速上手！

里面有什么

- 面向 Web 开发人员的HTTP/2
- 升级和故障排除
- 真实示例和案例研究
- QUIC和HTTP/3

为 Web 开发人员和网站管理员编写。

Barry Pollard是一名专业开发人员，拥有二十年的软件和基础设施开发、支持和调优经验。

要下载 PDF、ePub 和 Kindle 格式的免费电子书，本书的所有者应访问
www.manning.com/books/http2-in-action

Free eBook
见第一页

“对新的HTTP/2标准的精彩介绍，有深刻的解释和非常好的例子。”

Alain Couniot， STIB-MIVB

“该协议在 20 年内发生的最重大变化，详尽描述。批判性阅读适用于网络操作人员和网络开发人员。”

Ronald Cranston，英国天空电视台

“学习HTTP/2的所有细微差别以及它将如何影响未来发展的重要资源。”

Tom McKearney 应用信息科学

“迄今为止我看到的对HTTP/2最清晰的解释。强烈推荐。”

Edwin Kwok，红兵

ISBN-13: 978-1-61729-516-4
ISBN-10: 1-61729-516-7
54999
9 781617 295164

曼宁

49.99 美元/罐 65.99 美元[包括电子书]