



HTTP/2 在行动

巴里波拉德

M 曼宁

HTTP/2 实战

HTTP/2 实战

巴里波拉德_



曼宁
庇护岛

有关本书和其他 Manning 书籍的在线信息和订购,请访问www.manning.com。出版商在订购大量本书时提供折扣。

获取更多资讯,请联系

特别销售部 Manning Publications
Co.
鲍德温路 20 号
PO Box 761
Shelter Island, NY 11964 电
子邮箱:orders@manning.com

©2019 Manning Publications Co. 保留所有权利。

未经出版商事先书面许可,不得以任何形式或通过电子、机械、影印或其他方式复制、存储在检索系统中或传播本出版物的任何部分。

制造商和销售商用来区分其产品的许多名称都被声明为商标。如果这些名称出现在书中,并且 Manning Publications 知道商标声明,则这些名称已印在首字母大写或全部大写中。

认识到保存所写内容的重要性,Manning 的政策是将我们出版的书籍印刷在无酸纸上,我们为此尽最大努力。

还认识到我们有责任保护地球资源,Manning 书籍印刷的纸张至少有 15% 被回收利用,并且在不使用元素氯的情况下进行加工。



曼宁出版公司
鲍德温路 20 号
邮政信箱 761
纽约州庇护岛 11964

开发编辑:Kevin Harrel
技术开发编辑:Thomas McKearney
评论编辑:Ivan Martinovic
项目编辑:Vincent Nordhaus
文案编辑:凯西·辛普森
校对:Alyson Brener
技术校对:Lokeshwar Vangala
排字员:Dennis Dalinnik
封面设计师:Marija Tudor

书号:9781617295164

美国印制

1 2 3 4 5 6 7 8 9 10 - SP - 24 23 22 21 20 19

纪念 Ronan Rafferty (1977-2018) ,网络开
发者和朋友

简要内容

第1部分转向HTTP/2	1
1 ■ Web 技术和 HTTP 3	2
2 ■ HTTP/2 之 路 35	
3 ■ 升级到 HTTP/2	69
第2部分使用HTTP/2.....	91
4 ■ HTTP/2 协议基础知 识 93	5
5 ■ 实施 HTTP/2 推送 142	
6 ■ 针对 HTTP/2 进行优化 182	
第3部分高级HTTP/2	223
7 ■ 高级 HTTP/2 概念 225	8
8 ■ HPACK 标头 压缩 249	
第4部分HTTP的未来.....	279
9 ■ TCP、QUIC 和 HTTP/3 281	10
10 ■ HTTP 从何而来 317	

内容

前言 xv 致谢
xvii 关于本书 xx

关于作者 xxiii
关于封面插图 xxiv

第1部分转向HTTP/2 1

1 Web 技术和 HTTP 3

1.1 网络如何运作 3

互联网与万维网 4 ■ 当您浏览网页时会发生什么? 4个

1.2 什么是HTTP? 9

1.3 HTTP 的语法和历史 15 HTTP/0.9 15 ■
HTTP/1.0 16 ■ HTTP/1.1 22

1.4 HTTPS简介 28

1.5 用于查看、发送和接收 HTTP 消息的工具 31 在 Web 浏览器中使用开发人员工具 31 ■ 发送 HTTP 请求 33 ■
用于查看和发送 HTTP 请求的其他工具 34

2 HTTP/2 之路 35

2.1 HTTP/1.1 和当前的万维网 36

HTTP/1.1 的基本性能问题 38 HTTP/1.1 的流水线 40 ■用
于 Web 性能测量的瀑布图 41 2.2 HTTP/1.1 性能问题的解决方法
43

使用多个 HTTP 连接 44 ■减少请求 46 HTTP/1 性能优化总结 48 2.3
HTTP/1.1 的其他问题 48 2.4 实际示例 49 示例网站 1:amazon.com 49
■示例网站 2:imgur.com 54 ■如何这真的是一个很大的问题吗? 55 2.5 从 HTTP/
1.1 迁移到 HTTP/2 56

SPDY 56 ■ HTTP/2 58

2.6 HTTP/2 对 Web 性能意味着什么 59 HTTP/2 强大功能的极端示例
59 ■设定对 HTTP/2 性能提升的预期 62 ■ HTTP/1.1 的性能
变通方案作为潜在的反模式 67

3 升级到 HTTP/2 69

3.1 HTTP/2 支持 69 浏览器端的

HTTP/2 支持 70 ■ HTTP/2 对服务器的支持 75 ■不支持 HTTP/2 时的
回退 77

3.2 为您的网站启用 HTTP/2 的方法 78

Web 服务器上的 HTTP/2 78 ■带有反向代理的 HTTP/2 80 ■通
过 CDN 的 HTTP/2 84 ■实施
HTTP/2 总结 85

3.3 HTTP/2 设置故障排除 85

第2部分 使用HTTP/2.....91

4 HTTP/2 协议基础 93

4.1 为什么用 HTTP/2 而不是 HTTP/1.2? 94

二进制而不是文本 95 ■多路复用而不是同步 96 ■流优先级和流
量控制 99
标头压缩 100 ■服务器推送 101

内容

习

4.2 如何建立 HTTP/2 连接 101

使用 HTTPS 协商 102 ■ 使用 HTTP 升级标头 109 ■ 使用先验知识
 112 ■ HTTP 替代方案
 服务 112 ■ HTTP/2 前言消息 113

4.3 HTTP/2 帧 114

查看 HTTP/2 帧 114 ■ HTTP/2 帧格式 121
 通过示例检查 HTTP/2 消息流 122 ■ 其他框架 137

5 实施 HTTP/2 推送 142

5.1 什么是 HTTP/2 服务器推送? 142 5.2 如何推动

146

使用 HTTP 链接标头推送 146 ■ 查看 HTTP/2 推送 148 ■ 使用链接标
 头从下游系统推送 151 ■ 更早推送 155 ■ 以其他方式推送 161 5.3 HTTP/
 2 推送如何在浏览器中工作 163

查看推送缓存如何工作 163 ■ 拒绝推送
 RST_流 166

5.4 如何有条件地推送 167 在服务器端跟踪推送 167

■ 使用 HTTP 条件请求 168 ■ 使用基于 cookie 的推送 168 使用
 缓存摘要 169 5.5 推送什么 170

你能推什么? 170 ■ 你应该推什么? 171 自动推送 172

5.6 HTTP/2 推送故障排除 173 5.7 HTTP/2 推送的性能

影响 175 5.8 推送与预加载 176 5.9 HTTP/2 推送的其他用例 179

6 针对 HTTP/2 进行优化 182

6.1 HTTP/2 对 Web 开发人员意味着什么 183 6.2 一些 HTTP/1.1 优化现
 在是反模式吗? 184 HTTP/2 请求仍然有成本 184 ■ HTTP/2 不是无限
 的 187 ■ 压缩对于更大的资源更有效 189 ■ 带宽限制和资源争
 用 191 ■ 分片 192 ■ 内联 193 结论 193

6.3 Web 性能技术在 HTTP/2 下仍然相关 194	最小化传输的数据量 194	■ 使用缓存来防止重新发送数据 202	■ Service workers 可以进一步减少网络负载 206	■ 不要发送不需要的东西 206	HTTP 资源提示 207	■ 减少最后一英里延迟 209	优化 HTTPS 209	■ 与 HTTP 无关的网络性能技术 212
6.4 同时针对 HTTP/1.1 和 HTTP/2 进行优化 212								
	测量 HTTP/2 流量 213	■ 检测服务器端的 HTTP/2 支持 214	■ 检测客户端的 HTTP/2 支持 217	■ 连接合并 218	■ 为 HTTP/1.1 用户优化多长时间 220			
第3部分 高级 HTTP/2	223							

7 高级 HTTP/2 概念 225

7.1 流状态 226		
7.2 流量控制 229		
	流量控制示例 230	■ 在服务器上设置流量控制 234
7.3 流优先级 234	流依赖性 235	■
	流加权 238	为什么优先级需要如此复杂？ 241
	网络服务器和浏览器的优先级排序 241	
7.4 HTTP/2 一致性测试 245	服务器一致性测试 245	
	客户端一致性测试 247	

8 HPACK 头压缩 249

8.1 为什么需要头部压缩？ 249	8.2 压缩如何工作 251
找表 252	■ 更高效的编码技术 252
	回溯压缩 254

8.3 HTTP 主体压缩 255	8.4 HTTP/2 的	
HPACK 标头压缩 257		
	HPACK 静态表 258	■ HPACK 动态表 259
	HPACK 头类型 259	■ 霍夫曼编码表 265
	霍夫曼编码脚本 266	■ 为什么霍夫曼编码并不总是最优的 268

8.5 HPACK 压缩的真实示例 268 8.6 客户端和服务器实现中的 HPACK 275

8.7 HPACK 的价值 277

第4部分 HTTP的未来.....279

9 TCP、QUIC 和 HTTP/3 281

9.1 TCP 低效和 HTTP 282

创建 HTTP 连接时的设置延迟 283 ■ TCP 中的拥塞控制效率低下

284 ■ TCP 效率低下对网络的影响

HTTP/2 293 ■ 优化 TCP 297 ■ TCP 的未来和
HTTP 302

9.2 QUIC 303 QUIC 的

性能优势 304 ■ QUIC 和互联网堆栈 305 ■ 什么是 UDP 以及为
什么在其上构建 QUIC 306 QUIC 标准化 309 ■ HTTP/2 和 QUIC
之间的差异 311 ■ QUIC 工具 313 ■ QUIC 实现 314 你应该使用奎
克？ 315

10 HTTP 从何而来 317

10.1 HTTP/2 的争议及其未解决的问题 318 针对 SPDY 的争论 318 ■ HTTP

中的隐私问题和状态 320 ■ HTTP 和加密 324 ■ 传输协议问题

327 ■ HTTP/2 过于复杂 331 ■ HTTP/2 是权宜之计 332 10.2 现实世
界中的 HTTP/2 333 10.3 HTTP/2 的未来版本以及 HTTP/3 或 HTTP/
4 可能带来什么 334 QUIC HTTP/3 是什么？ 334 ■ 进一步发展 HTTP

二进制协议 335 ■ 在传输层之上发展 HTTP 335 什么需要新的 HTTP 版本？ 338

■ 如何引入 HTTP 的未来版本 339 10.4 HTTP 作为更通用的传输协议 339

使用 HTTP 语义和消息传递非 Web 流量 339

使用 HTTP/2 二进制框架层 341 ■ 使用 HTTP 启动另一个协议 341

附录 将常见的 Web 服务器升级到 HTTP/2 346

索引 375

前言

我很早就对 HTTP/2 产生了兴趣。一项新技术的出现,承诺几乎免费的性能提升,同时可能消除 Web 开发人员必须使用的一些混乱的变通办法的需要,这绝对是有趣的。

然而,实际情况要复杂一些,在花了一些时间弄清楚如何将它部署到我的 Apache 服务器上,然后努力解释我所看到的对性能的影响之后,我对缺乏文档感到沮丧。我写了几篇关于如何设置它的博客文章,这些文章被证明很受欢迎。与此同时,我开始参与 GitHub 上的一些 HTTP/2 项目,并潜伏在 Stack Overflow 上,帮助那些与我有类似问题的人。当 Manning 来找我写一本关于 HTTP/2 的书时,我欣然接受了这个机会。我没有参与它的起源,但我觉得我可以和许多像我一样苦苦挣扎的网络开发人员交谈,他们听说过这项技术但缺乏部署它的知识。

在撰写本书的一年半时间里,HTTP/2 已经成为主流,并被越来越多的网站所使用。随着软件的更新,一些部署问题变得更容易了,我希望本书中描述的一些问题很快就会成为过去,但我怀疑至少再过几年,HTTP/2 将需要一些努力来启用它。

当您能够打开它时,HTTP/2 应该可以立即提升性能,而无需进行任何配置或了解。然而,此生没有免费的东西,协议及其部署中的微妙之处和细微差别

意味着更深入的理解将为网站所有者提供良好的服务。Web 性能优化是一个蓬勃发展的行业,而 HTTP/2 是另一种工具,应该会在现在和未来带来有趣的技术和机会。

网络上提供了大量信息,对于那些有时间和意愿去寻找、过滤和理解的人来说,阅读所有不同的意见甚至直接与协议设计者交流是非常令人满意的和实施者。然而,对于像 HTTP/2 这样大的主题,一本书的范围和深度让我有机会全面解释这项技术,同时触及相关主题并为您提供参考,以便在我激起您对某事的兴趣时跟进。我希望我已经通过这本书实现了这个目标。

致谢

首先,我要感谢我非常善解人意的妻子 Aine,她在过去一年半的时间里照顾了我们两个年幼的孩子(在本书写作期间变成了“三个年幼的孩子”)!当我被锁在外面时,我疯狂地敲击着我的键盘。看到这本书终于出版,她可能是世界上唯一比我高兴的人了!还需要特别感谢我的姻亲(Buckles),他们帮助Aine在远离我家庭书房的地方招待我们的孩子,这样我就可以集中注意力。

Manning 团队在整个过程中给予了极大的支持。我要特别感谢 Brian Sawyer,他首先与我取得联系并为我提供了编写本书的机会。他在指导我完成提案过程中的帮助确保了这本书被出版商选中。Kevin Harreld 作为开发编辑做得很好,在耐心地回答我的许多问题的同时,轻轻地把我推向正确的方向。Thomas McKearney 作为技术开发编辑提供了出色的技术监督,对所有章节都提供了详细的反馈。由 Ivan Martinovic 组织的三轮评论提供了宝贵的反馈和指导,说明在本书的进展过程中什么是有效的,什么是需要改进的。同样,Manning Early Access Program (MEAP) 是从真实读者那里获得反馈的绝佳方式,而 Matko Hrvatin 在组织这方面做得很好。

我还要感谢整个 Manning 营销团队,他们从一开始就帮助宣传了这本书,但要特别感谢克里斯托弗·考夫曼(Christopher Kaufman),他容忍了我似乎没完没了地编辑宣传材料的要求。

让这本书准备好投入生产是一项艰巨的任务,所以感谢文森特

Nordhaus 在此过程中引导我的宝贵成果。凯西·辛普森 (Kathy Simpson) 和艾莉森·布雷纳 (Alyson Brener) 在文字编辑和校对期间使这本书的可读性大大提高，并且两人都承担了一项艰巨的任务，即在太多场合处理我对他们（好多了！）的措辞和语法的质疑。还要感谢其他校对人员、图形、排版和排版团队，他们帮助本书完成了最后阶段。我的名字可能出现在封面上，但所有这些人，以及其他人都帮助我将漫无目的的想法写成一本专业出版物。任何仍然存在的错误无疑都是我自己的错，而不是他们的错。

我收到了 Manning 以外的许多人的反馈，从提案审稿人到手稿审稿人（特别感谢你们中那些通过了所有三个审阅的人！），再到 MEAP 读者。我要特别感谢 Lucas Par due 和 Robin Marx，他们煞费苦心地审阅了整篇手稿，并在整个过程中提供了宝贵的 HTTP/2 专业知识。其他评论家包括 Alain Courniot、Anto Aravindh、Art Bergquist、Camal Cakar、Debmalya Jash、Edwin Kwok、Ethan Rivett、Evan Wallace、Florin-Gabriel Barbuceanu、John Matthews、Jonathan Thoms、Joshua Horwitz、Justin Coulston、Matt Deimel、Matthew Farwell、Matthew Halverson、Morteza Kiadi、Ronald Cranston、Ryan Burrows、Sandeep Khurana、Simeon Leyzerzon、Tyler Kowallis 和 Wesley Beary。谢谢大家。

在技术方面，我要感谢 Tim Berners-Lee 爵士多年前启动了整个网络，感谢 Mike Belshe 和 Robert Peon 发明了 SPDY，然后在 Martin Thomp 之子，代理编辑。标准化之所以成为可能，要归功于互联网工程任务组 (IETF) 的辛勤工作，尤其是 HTTP 工作组，由 Mark Nottingham 和 Patrick McManus 担任主席。没有他们，也没有他们的雇主允许花时间在这项工作上，就不会有 HTTP/2，因此也不需要这本书。

我一直对技术社区投入志愿者工作的时间和精力感到惊讶。从开源项目到 Stack Overflow、GitHub 和 Twitter 等社区网站，再到博客和演示文稿，许多人付出了如此多的时间，除了帮助他人和扩展自己的知识外，没有任何明显的物质回报。我很感激也很自豪能成为这个社区的一员。如果没有学习 Web 性能专家 Steve Sounders、Yoav Weiss、Ilya Grigorik、Pat Meehan、Jake Archibald、Hooman Beheshti 和 Daniel Stenberg 的教诲，这本书就不可能完成，所有这些人都在本书中被引用。特别要感谢 Stefan Eissing，他在 Apache HTTP/2 实现方面做了大量工作，这首先引起了我的兴趣，还有 Tatsuhiko Tsujikawa，他创建了它使用的底层 nghttp2 库（以及许多其他 HTTP/2 实现）。同样，Web Pagetest、The HTTP Archive、W3Techs、Draw.io、TinyPng、nghttp2、curl、Apache、nginx 和 Let's Encrypt 等免费工具是本书得以出版的重要原因。我要特别感谢那些允许在本书中使用其工具图像的公司。

最后,我要感谢您,读者,感谢您对本书的关注。
尽管许多人以某种方式帮助制作它,但他们这样做只是因为像您这样的人帮助使书籍保持活力并使它们值得出版。我希望您能从本书中获得宝贵的见解和理解。

关于这本书

编写HTTP/2 in Action是为了使用真实世界的示例以易于理解、实用的方式解释协议。协议规范可能枯燥难懂,因此本书旨在通过与互联网所有用户相关的易于理解的示例来阐述详细信息。

谁应该读这本书?

本书是为 Web 开发人员、网站管理员以及那些对了解 Internet 技术的工作原理感兴趣的人而写的。本书旨在全面介绍 HTTP/2 及其涉及的所有细节。

尽管存在大量关于该主题的博客文章,但大多数都是针对特定主题的高级或详细级别。本书旨在涵盖整个协议以及其中涉及的许多复杂性,如果读者希望进一步阅读,这应该让读者做好阅读和理解规范和特定博客文章的准备。HTTP/2 的创建主要是为了提高性能,因此任何对 Web 性能优化感兴趣的人都一定会获得有用的理解和见解。此外,本书还包含许多可供进一步阅读的参考资料。

本书的结构本书共 10 章,分为 4 个部分。

第 1 部分解释了升级到 HTTP/2 的背景、需求和方法: ■第 1 章提供了理解本书所需的背景知识。即使那些对互联网只有基本了解的人也应该能够跟上。

■第2章着眼于HTTP/1.1的问题以及需要HTTP/2的原因。■第3章讨论了为您的网站启用HTTP/2的升级选项以及与此过程相关的一些复杂问题。本章由附录进行补充,附录提供了流行的Web服务器Apache.nginx和IIS的安装说明。

在第2部分中加快步伐,我将教授协议及其对Web开发实践的意义:

- 第4章介绍了HTTP/2协议的基础知识,HTTP/2如何配置连接建立,HTTP/2帧的基本格式。
- 第5章介绍HTTP/2推送,这是该协议的全新部分,允许网站所有者主动发送浏览器尚未请求的资源。
- 第6章着眼于HTTP/2对Web开发实践的意义。

第3部分进入协议的高级部分,Web开发人员甚至Web服务器管理员目前可能没有太多影响力:

- 第7章介绍了HTTP/2规范的其余部分 包括状态、流量控制和优先级 并考察了实现中HTTP/2一致性的差异。
- 第8章深入探讨了HPACK协议,该协议用于HTTP/2中的HTTP标头压缩。

第4部分着眼于HTTP的未来:

- 第9章着眼于TCP、QUIC和HTTP/3。技术永不眠,现在HTTP/2可用,开发人员已经在寻找改进它的方法。
- 本章讨论HTTP/2没有解决的低效问题,以及如何在其后继者HTTP/3中改进这些问题。■第10章超越了HTTP/3,探讨了HTTP可以改进的其他方式,包括对HTTP/2标准化期间提出的问题的反思,以及这些问题是否已被证明是现实世界中的问题。

阅读本书后,读者应该对HTTP/2及相关技术有很好的理解,对Web性能优化也应该有更深入的了解。当QUIC和HTTP/3将来问世时,它们也将为它做好准备。

关于代码

与大多数技术书籍不同,HTTP/2 in Action没有大量代码,因为这本书是关于协议而不是编程语言的。它试图向您传授适用于任何Web服务器或用于为Web页面提供服务的编程语言的高级概念。然而,本书有一些NodeJS和Perl示例,以及网络服务器配置片段。

源代码和配置片段采用固定宽度的字体格式,这样可以将它们与普通文本分开。有时,代码也会以粗体显示,以突出显示与本章之前的步骤相比发生变化的代码,例如当新功能添加到现有代码行时。

源代码可从出版商的网站<https://www.manning.com/books/http2-in-action>或从 GitHub 的<https://github.com/bazzadp/http2-in-action> 下载。

liveBook 论坛

购买HTTP/2 in Action可免费访问由 Manning Publications 运营的私人网络论坛,您可以在其中发表对本书的评论、提出技术问题,并获得作者和其他用户的帮助。要访问论坛,请访问<https://livebook.manning.com/#!/book/http2-in-action/discussion>,您还可以在<https://livebook.manning.com/#!/discussion>上了解有关 Manning 论坛和行为规则的更多信息。

Manning 对我们的读者的承诺是提供一个场所,让各个读者之间以及读者与作者之间可以进行有意义的对话。这不是对作者参与任何特定数量的承诺,他们对论坛的贡献仍然是自愿的(并且是无偿的)。我们建议您尝试向作者提出一些具有挑战性的问题,以免他的兴趣发生偏差!只要本书还在印刷,就可以从出版商的网站访问论坛和以前讨论的档案。

在线资源

需要额外的帮助吗?

- HTTP/2 主页位于<https://http2.github.io/>。该页面包含指向 HTTP/2 和 HPACK 规范、HTTP/2 实现和常见问题解答的链接。
- HTTP 工作组主页位于<https://httpwg.org/>。该小组的大部分工作都在 GitHub 页面<https://github.com/httpwg/>和该小组的邮件列表(<https://lists.w3.org/Archives/Public/ietf-http-wg/>)上公开。
- Stack Overflow 也有一个 HTTP/2 标签(<https://stackoverflow.com/questions/tagged/http2>),作者经常在那里回答问题。

关于作者



BARRY POLLARD是一位专业的软件开发人员,拥有近二十年的软件和基础设施开发和支持行业经验。他对 Web 技术、性能调优、安全性和技术的实际使用有着浓厚的兴趣。您可以在<https://www.tunetheweb.com>或在 Twitter 上以@tunetheweb 的身份找到他的博客。

关于封面插画

HTTP/2 in Action封面上的图片标题为“1768年俄罗斯女商人的习惯”。插图取自 Thomas Jefferys 的 A Collection of the Dresses of Different Nations, Ancient and Modern (四卷) , 伦敦,出版于 1757 年至 1772 年间。扉页指出这些是手工上色的铜版画,用阿拉伯树胶增色.托马斯·杰弗里斯 (Thomas Jefferys, 1719-1771 年) 被称为“乔治三世国王的地理学家”。他是一位英国制图师,是当时领先的地图供应商。他为政府和其他官方机构雕刻和印刷地图,并制作了范围广泛的商业地图和地图集,尤其是北美地图。

作为一名地图制作者,他的工作激发了人们对他所调查和绘制的土地的当地服饰习俗的兴趣,这些习俗在这本合集中得到了精彩的展示。对遥远国度的迷恋和休闲旅行在 18 世纪后期是相对较新的现象,而像这样的收藏品很受欢迎,将游客和扶手椅旅行者介绍给其他国家的居民。潜水员

杰弗里斯书中的图画生动地讲述了大约 200 年前世界各国的独特性和个性。从那以后,着装规范发生了变化,当时如此丰富的地区和国家的多样性已经消失。现在通常很难区分一个大陆和另一个大陆的居民。也许,试图乐观地看待它,我们已经用文化和视觉多样性换取了更多样化的个人生活 或者更多样化和有趣的智力和技术生活。在很难区分一本计算机书籍和另一本计算机书籍的时代,曼宁用基于两个世纪前地区生活丰富多样性的书籍封面来庆祝计算机行业的创造力和主动性,杰弗里斯的照片使之重现生机。

第1部分

转向 HTTP/2

理解为什么 HTTP/2 在网络性能方面引起如此大的轰动

行业,您首先需要了解为什么需要它以及它希望解决什么问题。因此,本书的第一部分向那些不熟悉 HTTP/1 到底是什么或它如何工作的读者介绍;然后它解释了为什么需要版本 2。我在较高层次上谈论 HTTP/2 的工作原理,但将低层次的细节留到本书后面。相反,我通过讨论可用于将 HTTP/2 部署到站点的各种方法来结束这一部分。

1 Web 技术和 HTTP

本章涵盖浏览器如何加载网页什
么是 HTTP 以及它如何发展到 HTTP/1.1

HTTPS 基础知识
基本的 HTTP 工具

本章向您介绍了当今网络的工作原理，并解释了理解本书其余部分所必需的一些关键概念；然后介绍 HTTP 和以前版本的历史。我希望本书的许多读者至少对本文中讨论的许多内容有所了解。

章，但我鼓励你不要跳过它；以本章为契机重温自己的基础知识。

1.1 网络如何运作

互联网已经成为日常生活中不可或缺的一部分。购物、银行业务、通信和娱乐都依赖于互联网，随着物联网 (IoT) 的发展，越来越多的设备被放到网上，可以远程访问它们。这种访问是通过多种技术实现的，包括超文本传输协议 (HTTP)，这是请求的关键方法。

第1章 Web 技术和 HTTP

访问远程 Web 应用程序和资源。尽管大多数人都知道如何使用 Web 浏览器上网,但很少有人真正了解这项技术的工作原理,为什么 HTTP 是 Web 的核心部分,或者为什么下一版本 (HTTP/2) 在网络社区。

1.1.1 互联网与万维网

对于许多人来说,互联网和万维网是同义词,但区分这两个术语很重要。

互联网是通过共享使用互联网协议 (IP) 来路由消息而链接起来的公共计算机的集合。它由许多服务组成,包括万维网、电子邮件、文件共享和互联网电话。因此,万维网 (或网络)只是互联网的一部分,尽管它是最明显的一部分,而且人们经常通过网络邮件前端 (例如 Gmail、Hotmail 和 Yahoo!) ,他们中的一些人可以将 Web 与互联网互换使用。

HTTP 是网络浏览器请求网页的方式。它是 Tim Berners-Lee 在发明网络时定义的三大技术之一,另外还有资源的唯一标识符 (这是统一资源定位器或 URL 的来源) 和超文本标记语言 (HTML)。Internet 的其他部分有自己的协议和标准来定义它们的工作方式以及它们的基础消息如何通过 Internet 路由 (例如使用 SMTP、IMAP 和 POP 的电子邮件)。在检查 HTTP 时,您主要是在与万维网打交道。然而,这条界线越来越模糊,因为构建在 HTTP 之上的服务,即使没有传统的 Web 前端,也意味着定义 Web 本身越来越棘手!这些服务 (缩写为 REST 或 SOAP) 可由网页和非网页 (如移动应用程序) 使用。IoT 仅表示公开服务的设备,其他设备 (计算机、移动应用程序,甚至其他 IoT 设备) 通常可以通过 HTTP 调用与之交互。因此,例如,您可以使用 HTTP 向灯发送消息以从手机应用程序打开或关闭它。

尽管互联网由无数服务组成,但随着网络使用量的持续增长,其中许多服务的使用比例越来越低。我们这些最早回忆起互联网的人都会想起 BBS 和 IRC 等首字母缩略词,如今它们几乎已经消失,取而代之的是网络论坛、社交媒体网站和聊天应用程序。

所有这一切都意味着,尽管万维网一词经常被错误地与互联网互换使用,但网络的持续兴起 或者至少是为它创建的 HTTP 可能意味着很快,这种理解可能不会那么遥远从它曾经的真相。

1.1.2 浏览网页时会发生什么?

现在,我回到 HTTP: 请求网页的主要和第一次使用。当您在您最喜欢的浏览器中打开网站时,无论该浏览器是在台式机还是笔记本电脑、平板电脑、手机或任何其他可以访问的设备上

网络是如何运作的

5/5

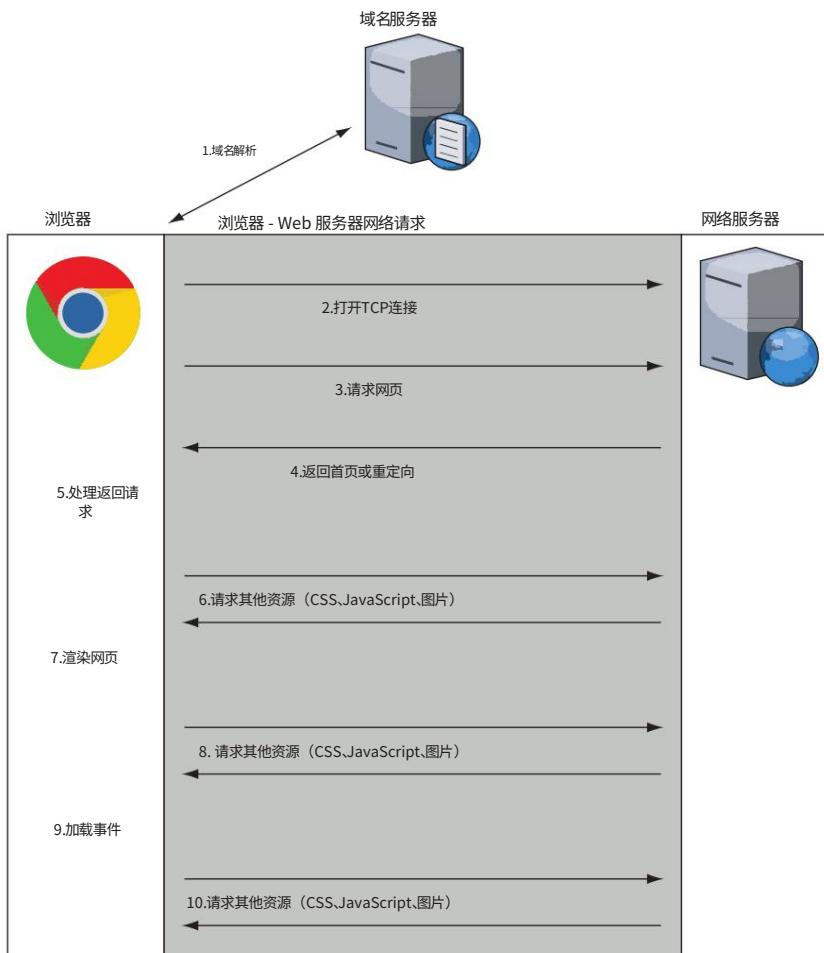


图 1.1 浏览网页时的典型交互

允许互联网访问,发生了很多事情。要充分利用本书,您需要准确了解浏览网页的方式。

假设您启动浏览器并访问www.google.com。在几秒钟内,将发生以下情况,如图 1.1 所示:

- 1 浏览器从域名系统 (DNS) 服务器请求www.google.com的真实地址,该服务器将人类友好的名称 www.google.com 转换为机器友好的 IP 地址。

如果您将 IP 地址视为电话号码,那么 DNS 就是电话簿。此 IP 地址可以是旧格式的 IPv4 地址 (例如 216.58.192.4, 这几乎是人类可用的) 或新格式的 IPv6 地址 (例如 2607:f8b0:4005:801:0:0:2004, 这肯定会进入“仅限机器”)

第1章 Web 技术和 HTTP

领土）。就像当一个城市的电话号码开始用完时偶尔会重新指定电话区号一样，需要 IPv6 来应对现在和将来连接到互联网的设备的爆炸式增长。

请注意，由于互联网的全球性，大公司通常在全球拥有多台服务器。当您向 DNS 询问 IP 地址时，它通常会提供最近的服务器的 IP 地址，以使您的互联网浏览速度更快。例如，美国的人获得的 www.google.com 的 IP 地址与欧洲的人不同，所以如果您获得的 www.google.com 的 IP 地址值与我提供的不同，请不要担心这里。

IPv5 到底发生了什么？

如果互联网协议第 4 版 (IPv4) 被第 6 版 (IPv6) 取代，那么第 5 版发生了什么变化？为什么您从未听说过 IPv1 到 IPv3？

IP 数据包的前 4 位给出版本，理论上将其限制为 15 个版本。在广泛使用的 IPv4 之前，有四个实验版本，从 0 开始到 3。这些版本都没有正式标准化，直到第 4 版才正式标准化。^a之后，第 5 版被指定用于互联网流协议，这是旨在用于实时音频和视频流，类似于后来的 IP 语音 (VoIP)。然而，该版本从未流行起来，尤其是因为它遇到了与版本 4 相同的地址限制，并且当版本 6 出现时，对它的工作停止了，留下版本 6 作为 IPv4 的继承者。显然，它最初被称为版本 7 是因为错误地假设版本 6 已经分配。^b版本 7、8 和 9 也已分配，但同样不再使用。如果有 IPv6 的后继者，它很可能是 IPv10 或更高版本，这无疑会导致类似于打开此边栏的问题！

^a 请参阅<https://tools.ietf.org/html/rfc760>。该协议后来被更新和替换(<https://tools.ietf.org/html/rfc791>)。

^b 请参阅<https://archive.is/QqU73#selection-417.1-417.15>。

2 Web 浏览器要求您的计算机在标准 Web 端口（端口 80）²或标准安全 Web 端口（端口 443）上通过 IP 打开到此地址的传输控制协议 (TCP) 连接¹。

IP 用于通过互联网引导流量（因此，名称为互联网协议！），但 TCP 增加了稳定性和重新传输以使连接可靠（“你好，你明白了吗？”“不，你能重复最后一点吗，请？”）。

¹ 谷歌已经开始试验 QUIC，所以如果你从 Chrome 连接到谷歌网站，你可以使用它。我在第 9 章讨论 QUIC。

² 包括谷歌在内的一些网站使用一种称为 HSTS 的技术来自动使用安全 HTTP 连接 (HTTPS)，该连接在端口 443 上运行，因此即使您尝试通过 HTTP 连接，连接也会在发送请求之前自动升级为 HTTPS。

由于这两种技术经常一起使用,它们通常缩写为 TCP/IP,并且它们一起构成了大部分互联网的骨干。

一台服务器可用于多种服务（例如电子邮件、FTP、HTTP 和 HTTPS [HTTP 安全] 网络服务）,并且端口允许不同的服务一起位于一个 IP 地址下,就像企业可能有电话分机一样对于每个员工。

3当浏览器连接到网络服务器时,它可以开始请求网站。这一步是 HTTP 发挥作用的地方,我将在下一节中研究它的工作原理。现在,请注意网络浏览器使用 HTTP 向谷歌服务器请求谷歌主页。

注意此时,您的浏览器会自动将简写网址(www.google.com)更正为语法更正确的 URL 地址 <http://www.google.com>。实际的完整 URL 包括端口,应该是<http://www.google.com:80>,但如果使用标准端口（80 用于 HTTP,443 用于 HTTPS）,浏览器会隐藏端口。如果正在使用非标准端口,则会显示该端口。例如,某些系统,特别是在开发环境中,使用端口 8080 进行 HTTP 或使用 8443 进行 HTTPS。

如果正在使用 HTTPS（我在 1.4 节中更详细地介绍了 HTTPS）,则需要额外的步骤来设置保护连接的加密。

4 Google 服务器以您请求的任何 URL 进行响应。通常,从初始页面发回的是构成 HTML 格式网页的文本。HTML 是一种标准化的、结构化的、基于文本的格式,它构成了页面的文本内容。它通常分为由 HTML 标记定义的各个部分,并引用制作您习惯看到的富媒体网页所需的其他信息（级联样式表 [CSS]、JavaScript 代码、图像、字体等）。

然而,响应可能不是 HTML 页面,而是转到不同位置的指令。例如,Google 仅在 HTTPS 上运行,因此如果您访问<http://www.google.com>,响应是一条特殊的 HTTP 指令（通常是301或302响应代码）,它会重定向到[https](https://www.google.com)上的新位置//www.google.com。

此响应会再次启动前面的部分或全部步骤,具体取决于重定向地址是不同的服务器/端口组合、同一位置的不同端口（例如重定向到 HTTPS）,甚至是网站上的不同页面相同的服务器和端口。

同样,如果出现问题,您会返回一个 HTTP 响应代码,其中最著名的是404 Not Found响应代码。

5 Web 浏览器处理返回的请求。假设返回的响应是 HTML,浏览器开始解析 HTML 代码并在内存中构建文档对象模型 (DOM),它是页面的内部表示。在此处理过程中,浏览器可能会看到正确显示页面所需的其他资源（例如 CSS、JavaScript 和图像）。

第1章 Web 技术和 HTTP

6 Web 浏览器请求它需要的任何其他资源。谷歌保持其网页相当精简;在撰写本文时,只需要 16 个其他资源。这些资源中的每一个都以类似的方式请求,遵循步骤 1-6,是的,这包括此步骤,因为这些资源可能反过来请求其他资源。普通网站不像 Google 那样精简,需要 75 种资源,³通常来自许多域,因此必须对所有这些资源重复步骤 1-6。这种情况是导致 Web 浏览变慢的关键因素之一,也是 HTTP/2 的关键原因之一,其主要目的是使请求这些额外资源的效率更高,您将在以后的章节中看到。

7 当浏览器拥有足够的关键资源时,它开始在屏幕上呈现页面。选择何时开始呈现页面是一项具有挑战性的任务,并不像听起来那么简单。如果 Web 浏览器等到所有资源都下载完毕,则需要很长时间才能显示网页,并且 Web 会变得更慢、更令人沮丧。但是,如果 Web 浏览器过早地开始呈现页面,您最终会随着更多内容的下载而页面跳来跳去,如果您正在阅读一篇文章时页面跳下来,这会很烦人。深入了解构成网络的技术 尤其是 HTTP 和 HTML/CSS/JavaScript 可以帮助网站所有者在加载页面时减少这些烦人的跳转,但是太多的网站没有有效地优化他们的页面以防止这些跳跃。

8 页面初始显示后,Web 浏览器在后台继续下载页面需要的其他资源,并在处理这些资源时更新页面。这些资源包括非关键项目,例如图像和广告跟踪脚本。因此,您经常会看到最初显示的网页没有图像(尤其是在连接速度较慢的情况下),随着下载的图像越来越多,图像被填充。

9 当页面完全加载时,浏览器停止加载图标(大多数浏览器地址栏上或附近的旋转图标)并触发 onLoad JavaScript 事件,JavaScript 代码可将其用作页面已准备好执行的标志某些动作。

10 此时,页面加载完毕,但浏览器并没有停止发送请求。网页是静态信息页面的时代已经过去很久了。许多网页现在都是功能丰富的应用程序,它们不断地与 Internet 上的各种服务器通信以发送或加载其他内容。

此内容可能是用户发起的操作,例如当您在 Google 主页的搜索栏中键入请求并立即看到搜索建议而无需单击搜索按钮时,也可能是应用程序驱动的操作,例如您的 Facebook 或 Twitter 提要会自动更新,您无需单击刷新按钮。这些动作通常发生在后台并且是

³ <https://httparchive.org/reports/page-weight#reqTotal>

您不可见,尤其是跟踪您在网站上的行为以向网站所有者和/或广告网络报告分析的广告和分析脚本。

如您所见,当您键入 URL 时会发生很多事情,而且通常发生在眨眼之间。这些步骤中的每一个都可以构成整本书的基础,在某些情况下会有所不同。然而,本书专注于(并更深入地研究)步骤 3-8(通过 HTTP 加载网站)。后面的一些章节(尤其是第 9 章)也涉及到步骤 2(HTTP 使用的底层网络连接)。

1.2 什么是HTTP?

上一节特意详细介绍了 HTTP 的工作原理,以便您了解 HTTP 如何适应更广泛的互联网。在本节中,我将简要描述 HTTP 的工作原理和使用方法。

正如我之前提到的,HTTP 代表超文本传输协议。顾名思义,HTTP 最初旨在传输超文本文档(包含指向其他文档的链接的文档),第一个版本只支持这些文档。很快,开发人员意识到该协议可用于传输其他文件类型(如图像),因此现在 HTTP 首字母缩略词的超文本部分不再太相关,但考虑到 HTTP 的广泛使用,重命名为时已晚它。

HTTP 依赖于可靠的网络连接,通常由 TCP/IP 提供,它本身建立在某种类型的物理连接(以太网、Wi-Fi 等)之上。

因为通信协议是分层的,所以每一层都可以专注于它擅长的事情。HTTP 不关心自己的底层细节

该网络连接是如何建立的。尽管 HTTP 应用程序应该注意如何处理网络故障或断开连接,但协议本身不允许执行这些任务。

开放系统互连(OSI)模型是一个概念模型,通常用于描述这种分层方法。该模型由七层组成,尽管这些层并不完全映射到所有网络,尤其是互联网流量。TCP 至少跨越两个(也可能是三个)层,具体取决于您如何定义这些层。图 1.2 粗略地显示了这个模型如何映射到网络流量以及 HTTP 在这个模型中的位置。

关于每一层的确切定义存在一些争论。在像互联网这样的复杂系统中,并不是所有的东西都可以像开发人员所希望的那样容易地分类和分离。事实上,互联网工程任务组(IETF)警告不要过于关注分层。⁴但从较高层次了解 HTTP 在该模型中的适用范围以及它如何依赖较低层次的协议来工作可能会有所帮助。

许多 Web 应用程序构建在 HTTP 之上,因此应用层更多地指网络层而不是 JavaScript 应用程序。

HTTP 本质上是一种请求-响应协议。Web 浏览器使用 HTTP 语法向 Web 服务器发出请求,Web 服务器使用包含所请求资源的消息进行响应。HTTP 成功的关键在于它的简单性。作为

⁴ <https://tools.ietf.org/html/rfc3439#section-3>

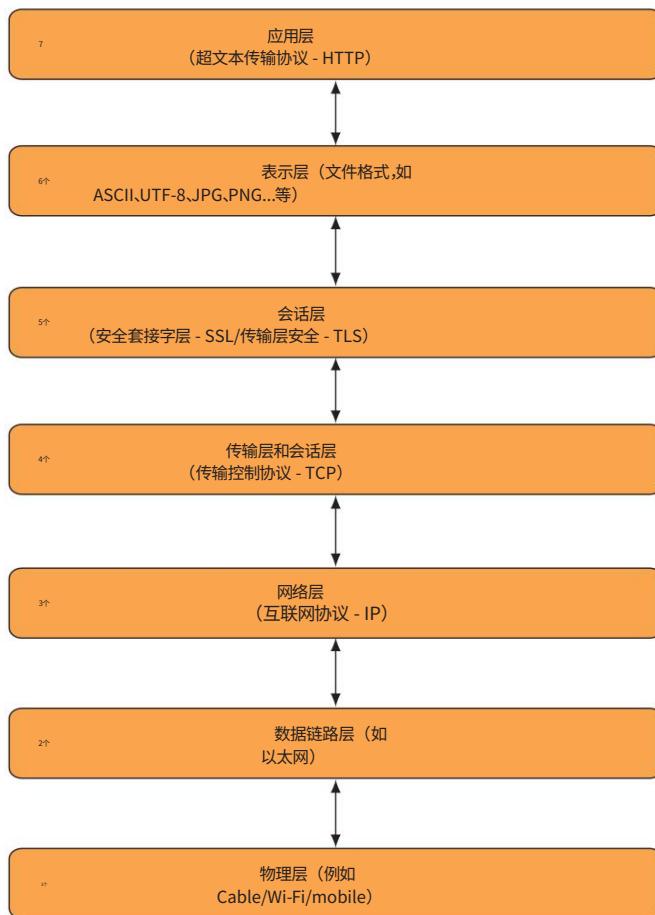


图 1.2 互联网流量的传输层

然而,您将在后面的章节中看到,这种简单性可能是 HTTP/2 的一个问题,它为了效率而牺牲了一些简单性。

打开连接后,HTTP 请求的基本语法如下:

获取/page.html

符号表示回车/换行 (Enter 或 Return 键)。在其基本形式中,HTTP 就是这么简单!您提供几种 HTTP 方法之一 (在本例中为 GET),然后提供您想要的资源 (/page.html)。请记住,此时您已经使用 TCP/IP 等技术连接到适当的服务器,因此您只需从该服务器请求您想要的资源,而无需关心该连接如何进行发生或被管理。

HTTP 的第一个版本 (0.9) 只允许这种简单的语法并且只有 GET 方法。在这种情况下,您可能会问为什么需要为 HTTP/0.9 声明 GET

request,因为它是多余的,但是 HTTP 的未来版本引入了其他方法,所以感谢 HTTP 的发明者有先见之明,看到更多的方法会出现。在下一节中,我将讨论 HTTP 的各种版本,但这种语法仍可识别为 HTTP GET 请求的格式。

考虑一个现实生活中的例子。因为 Web 服务器只需要 TCP/IP 连接来接收 HTTP 请求,您可以使用 Telnet 等程序模拟浏览器。Telnet 是一个简单的程序,它打开到服务器的 TCP/IP 连接,并允许您键入文本命令和查看文本响应。这个程序正是 HTTP 所需要的,尽管我在本章末尾介绍了用于查看 HTTP 的更好的工具。不幸的是,有些技术正变得不那么流行,Telnet 就是其中之一;默认情况下,许多操作系统不再包含 Telnet 客户端。您可能需要安装 Telnet 客户端来尝试一些简单的 HTTP 命令,或者您可以使用类似nc命令的等效命令。

此命令是netcat的缩写,安装在大多数类似 Linux 的环境中,包括 macOS,对于我在这里展示的简单示例,它几乎与 Telnet 相同。

对于 Windows,我建议使用 PuTTY 软件¹⁰而不是与 Windows 捆绑的默认客户端(通常不会安装,必须手动添加),因为默认客户端经常有显示问题,例如不显示您正在键入的内容或覆盖终端上已有的内容。安装并启动 PuTTY 时,您会看到配置窗口,您可以在其中输入主机(www.google.com)、端口(80)和连接类型(Telnet)。确保单击“从不”选项以在退出时关闭窗口;否则,您将看不到结果。

所有这些设置如图 1.3 所示。另请注意,如果您在输入以下任何命令时遇到问题并收到有关格式错误请求的消息,您可能需要将连接 > Telnet > Telnet 协商模式更改为

被动的。

如果您使用的是 Apple Macintosh 或 Linux 机器,您可以发出
如果已安装 Telnet,则直接从 shell 提示符输入 Telnet 命令:

```
$ 远程登录 www.google.com 80
```

或者,正如我之前提到的,以相同的方式使用nc命令:

```
$ nc www.google.com 80
```

当您有一个 Telnet 会话并建立连接时,您会看到一个空白屏幕,或者,根据您的 Telnet 应用程序,会看到类似以下的一些说明:

```
正在尝试 216.58.193.68...
连接到 www.google.com。
转义符是 ^] 。
```

¹⁰ <https://www.putty.org/>

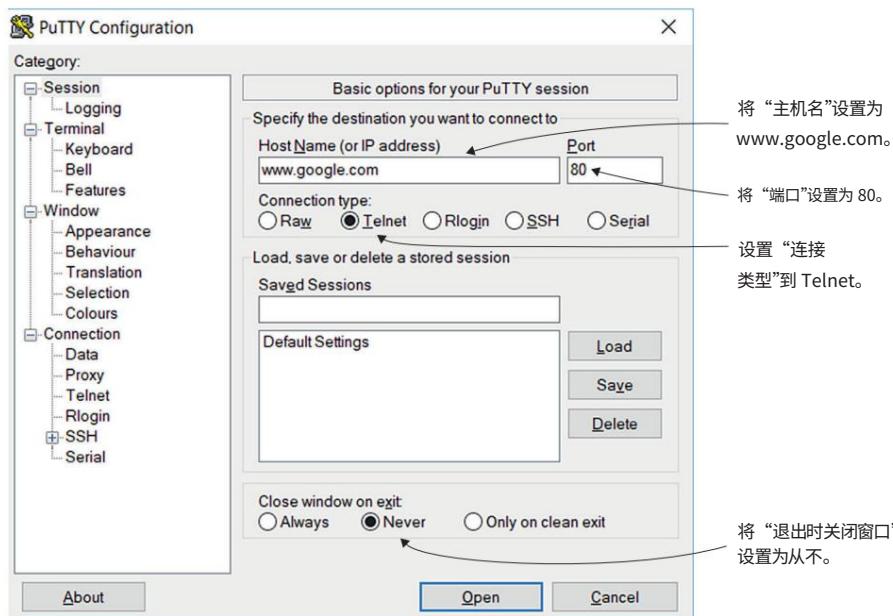


图 1.3 用于连接到 Google 的 PuTTY 详细信息

无论是否显示此消息,您都应该能够键入 HTTP 命令,因此键入 GET /然后按 Return 键,这会告诉 Google 服务器您正在寻找默认页面(/)和 (因为你还没有指定 HTTP 版本)你想使用默认的 HTTP/0.9。请注意,某些 Telnet 客户端在默认情况下不会回显您正在输入的内容 (尤其是与 Windows 捆绑在一起的默认 Telnet 客户端,正如我之前提到的),因此可能很难准确地看到您正在输入的内容。但是您仍然应该发送命令。

在公司代理后面使用 Telnet如果您的计算机不能直接访问互联网,您将无法使用 Telnet 直接连接到 Google。在使用代理来限制直接访问的公司环境中,这种情况经常发生。(我在第 3 章介绍了代理。)在这种情况下,您可以使用您的内部 Web 服务器之一(例如您的内部网站)作为示例,而不是使用 Google。在 1.5.3 节中,我讨论了可以与代理一起工作的其他工具,但是现在,您可以不按照说明继续阅读。

尽管您发送了默认的 HTTP/0.9 请求 (没有服务器再使用 HTTP/0.9),Google 服务器将响应,很可能使用 HTTP/1.0。响应是 HTTP 响应代码 200 (表示命令成功)或 302 (表示

声明服务器希望您重定向到另一个页面),然后关闭连接。我将在下一节中详细介绍这个过程,所以现在不要太在意这些细节。

以下是来自 Linux 服务器上命令行提示符的此类响应,响应行以粗体显示。请注意,为简洁起见,返回的 HTML 内容并未完整显示:

```
$ telnet www.google.com 80 正在尝试
172.217.3.196...
连接到 www.google.com。
转义符是 ^] 。
GET /
HTTP/1.0 200 OK 日期:2017
年 9 月 10 日星期日 16:20:09 GMT 到期: -1 缓存控制:私有,最大年
龄 = 0 内容类型:文本/html; charset=ISO-8859-1 P3P: CP= 这
不是 P3P 策略!看

https://www.google.com/support/accounts/answer/151657?hl=en 获取更多信息。”
服务器:gws X-XSS-
保护:1; mode=block X-Frame-Options: SAMEORIGIN
Set-Cookie:

NID=111=QIMb1TZHhHGXEPUXqbHChZGCcVLFQOvmqjNcUlejUXqbHChZKtrF4Hf4x4DVjTb01R 8DWShPlu6_aQ-
AnPXgONzEoGOpapm_VOTW0Y8TWVpNap_1234567890-p2g; expires=Mon, 12-Mar-2018 16:20:09 GMT;路径=/;域名
=.google.com;仅限HTTP
接受范围:无
变化:接受编码

<!doctype html><html itemscope="" itemtype=" http://schema.org/WebPage"
lang=" zh-cn "><head><meta content=" 搜索全球信息,包括网页、图片、视频等。Google 有许多特殊功能可帮助您准确找到所需内容。" name="描述"

...ETC。

</script></div></body></html>外部主机关闭了连接。
```

如果您不在美国,您可能会看到重定向到本地 Google 服务器。例如,如果您位于爱尔兰,Google 会发送 302 响应并建议浏览器转至 Google Ireland (<http://www.google.ie>) ,如下所示:

```
GET /
HTTP/1.0 302 找到位置: http://
www.google.ie/?gws_rd=cr&dcr=0&ei=BWe1WYrf123456qpibwDg 缓存控制:私有 内容类型:文本/html; charset=UTF-8 P3P:
CP= 这不是 P3P 策略!看

https://www.google.com/support/accounts/answer/151657?hl=en 获取更多信息。”
日期:2017 年 9 月 10 日,星期日 16:23:33 GMT
服务器:gws
内容长度:268
```

```
X-XSS-保护:1;模式=块
X-框架选项:SAMEORIGIN
设置 Cookie:NID=111=ff1KAwiMjt3X4MEg_KzqR_9eAG78CWNGEFIDG0Xlf7dLZsQeLerX
P8uSnXYCWNGEFIDG0dsM-8V8X8ny4nbu2w96GRTZtzXWOHvWS123456dhd0LpD_123456789; expires=Mon, 12-Mar-2018 16:23:33
GMT;路径=/;域名=.google.com;仅限HTTP
```

```
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8" >
<TITLE>302 移动</TITLE></HEAD><BODY>
<H1>302移动</H1>
文档已移动
<-->
此处
</BODY></HTML>外部主机关闭了连接。
```

如每个示例末尾所示,连接已关闭;要发送另一个 HTTP 命令,您需要重新打开连接。为避免此步骤,您可以通过在请求的资源后输入HTTP/1.1来使用 HTTP/1.1 (默认情况下保持连接打开,我将在后面讨论) :

获取/HTTP/1.1

请注意,如果您使用的是 HTTP/1.0 或 HTTP/1.1,则必须按两次 Return 以告知 Web 服务器您已完成发送 HTTP 请求。在下一节中,我将讨论为什么 HTTP/1.0 和 HTTP/1.1 需要这个双回车/空行

连接。

服务器响应后,您可以重新发出GET命令以再次获取页面。

实际上,Web 浏览器通常使用此打开的连接来获取其他资源,而不是再次获取相同的资源,但概念是相同的。

从技术上讲,为了遵守 HTTP/1.1 规范,HTTP/1.1 请求还要求您指定主机标头,原因我 (再次)稍后讨论。不过,对于这些简单的例子,不必太担心这个要求,因为 Google 似乎并不坚持这一点 (尽管如果您使用的是[www.google.com 以外的网站](http://www.google.com),您可能会看到意想不到的结果)。

如您所见,基本的 HTTP 语法很简单。它是一种基于文本的请求和响应格式,尽管这种格式在 HTTP/2 下转变为二进制格式时发生了变化。

如果您要请求非文本数据 (例如图像),则像 Telnet 这样的程序是不够的。当 Telnet 尝试将二进制图像格式转换为有意义的文本但失败时,Gobbledygook 将出现在终端会话中,如本例所示:

```
$ telnet www.google.com 80 正在尝试
172.217.3.164...
连接到 www.google.com。
转义符是 ^] 。
获取/images/branding/googlelogo/2x/googlelogo_color_120x44dp.png
```

```

I   ¥&   J   S   y   W I   7E^T   ~n   EG   I   ^
+.`x\w   CR\n   U3V   O>6b   y8   S   cHj^.   F   4=xw
(   F   Bc   ]Zu   ~Hj   i   R   G   mH   .   <   xH
      ;   fH   5   %WH   7   s/   w
b   @   4   BL   {:$   .(O   y   =!   \   D   M   9
      .,$I,$I,$I,$I,~T,LC
END B`连接被外部主机关闭。

```

我不再使用 Telnet,因为可以使用更好的工具来查看 HTTP 请求的详细信息,但是这个练习对于解释 HTTP 消息的格式和展示协议的初始版本有多么简单很有用。

正如我前面提到的,HTTP 成功的关键在于它的简单性,这使得它在服务级别上相对容易实现。因此,几乎任何具有网络能力的计算机,从复杂的服务器到物联网世界中的灯泡,都可以实现 HTTP 并立即通过网络提供有用的命令。实现一个完全 HTTP 兼容的 Web 服务器是一项艰巨得多的任务。类似地,Web 浏览器非常复杂,并且在通过 HTTP (包括用于显示已获取页面的 HTML、CSS 和 JavaScript) 获取网页后,还有无数其他协议需要应对。但是创建一个简单的服务来侦听 HTTP GET 请求并响应数据并不困难。HTTP 的简单性也导致了微服务架构风格的繁荣,其中一个应用程序被分解成许多独立的 Web 服务,通常基于更轻的应用程序服务器,例如 Node.js (Node)。

1.3 HTTP的语法和历史

HTTP 由 Tim Berners-Lee 和他在 CERN 研究组织的团队于 1989 年发起。它旨在成为一种实现互连计算机网络的方式,以提供对研究的访问并将它们链接起来,以便它们可以轻松地相互引用即时的;单击链接将打开相关文档。这种系统的想法由来已久,超文本一词是在 1960 年代创造的。随着 20 世纪 80 年代互联网的发展,实现这个想法成为可能。在 1989 年和 1990 年期间,伯纳斯-李发表了建立这样一个系统的提案⁶;他继续构建第一个基于 HTTP 的 Web 服务器和第一个请求 HTML 文档并显示它们的 Web 浏览器。

1.3.1 HTTP/0.9

第一个发布的 HTTP 规范是 0.9 版,发布于 1991 年。规范⁷很小,不到 700 字。它指定通过 TCP/IP (或类似的面向连接的服务)与服务器和可选端口 (如果未指定端口则使用 80) 建立连接。应发送单行 ASCII 文本,包括

⁶ <https://www.w3.org/History/1989/proposal.html>

⁷ <https://www.w3.org/Protocols/HTTP/AsImplemented.html>

GET,文档地址（没有空格）,回车和换行（回车是可选的）。服务器将以 HTML 格式的消息进行响应,它定义为“ASCII 字符的字节流”。它还指出,“消息因服务器关闭连接而终止”,这就是前面示例中每次请求后连接都关闭的原因。关于处理错误,规范指出:“错误响应以 HTML 语法的人类可读文本形式提供。除了文本内容之外,无法区分错误响应和满意响应。”它以这段文字结尾:“请求是幂等的。断开连接后,服务器无需存储有关请求的任何信息。”该规范为我们提供了 HTTP 的无状态部分,这既是一种祝福（就其简单性而言）也是一种诅咒（由于必须附加 HTTP cookie 等技术以允许状态跟踪的方式,这对于复杂的应用程序是必需的）。

以下是 HTTP/0.9 中唯一可能的命令：

获取/section/page.html

当然,请求的资源(/section/page.html)可以更改,但语法的其余部分是固定的。

没有 HTTP 标头字段（这里称为 HTTP 标头）或任何其他媒体（例如图像）的概念。令人惊奇的是,从这个简单的请求/响应协议（旨在为研究机构提供对信息的轻松访问）迅速催生了当今世界根深蒂固的富媒体万维网。即使在早期阶段,Berners-Lee 就将他的发明称为万维网（没有我们今天使用的空间）,再次显示了他对该项目范围的远见和计划,使其成为一个全球系统。

1.3.2 HTTP/1.0

万维网几乎是立竿见影的成功。根据 NetCraft,⁸到 1995 年 9 月,网络上有 19,705 个主机名。一个月后,这个数字跃升至 31,568,此后一直以惊人的速度增长。在撰写本文时,我们正在接近 20 亿个网站。到 1995 年,简单的 HTTP/0.9 协议的局限性很明显,大多数 Web 服务器已经实现了远远超出 0.9 规范的扩展。由 Dave Raggett 领导的 HTTP 工作组 (HTTP WG) 开始研究 HTTP/1.0,试图记录“协议的通用用法”。该文档于 1996 年 5 月作为 RFC 1945.⁹发布。它可以作为正式标准被接受,也可以作为非正式文档保留。¹⁰HTTP/1.0 RFC 是后者,不是正式规范。它将自己描述为顶部的“备忘录”,

⁸ <https://news.netcraft.com/archives/category/web-server-survey/>

⁹ <https://tools.ietf.org/html/rfc1945> 关于阅读和理解 RFC 的优秀文章位于

¹⁰ https://www.mnot.net/blog/2018/07/31/read_rfc。

声明，“这份备忘录为互联网社区提供信息。本备忘录未指定任何类型的互联网标准。”

不管 RFC 的正式地位如何,HTTP/1.0 添加了一些关键特性,包括

更多请求方式:在之前定义的基础上增加了HEAD和POST
得到。

为所有消息添加一个可选的 HTTP 版本号。默认情况下假定 HTTP/0.9 有助于向后兼容。 HTTP
标头,可以与请求和响应一起发送,以提供有关正在请求的资源和正在发送的响应的更多信息。

一个三位数的响应代码,表示 (例如)响应是否成功。此代码还启用了重定向请求、条件请求和错误状态
(404 未找到是最著名的之一)。

这些急需的协议增强是通过使用有机地发生的,HTTP/1.0 旨在记录现实世界中许多 Web 服务器已经发生的事情,而不是定义新的选项。这些附加选项为 Web 带来了大量新机会,包括首次通过使用响应 HTTP 标头
定义正文数据的内容类型来向网页添加媒体的能力。

HTTP/1.0方法GET方

法与 HTTP/0.9 下的方法保持一致,尽管添加的标头允许有条件的GET (仅当此资源自上次客户端获取后
发生更改时才获取的指令;否则,告诉客户端资源未更改并继续使用该旧副本)。此外,正如我之前提到的,用
户可以获得的不仅仅是超文本文档,还可以使用 HTTP 下载图像、视频或任何类型的媒体。

HEAD方法允许客户端获取资源的所有元信息 (例如 HTTP 标头),而无需下载资源本身。由于许多原
因,此方法很有用。例如,像 Google 这样的网络爬虫可以检查资源是否已被修改,只有在有修改时才下载它,
从而为它和网络服务器节省资源。

POST方法更有趣,允许客户端将数据发送到 Web 服务器。用户可以使用 HTTP POST文件,而不是使
用标准文件传输方法将新的 HTML 文件直接放在服务器上,前提是 Web 服务器设置为接收数据并对其进行
处理。 POST不限于整个文件;它可以用于更小的数据部分。网站上的表单通常使用POST,表单的内容作
为 HTTP 请求正文中的字段/值对发送。因此, POST方法允许将内容作为 HTTP 请求的一部分从客户端
发送到服务器,这是第一次 HTTP 请求可以像 HTTP 响应一样具有主体。

事实上，GET 允许在 URL 末尾指定的查询参数中发送数据，在 ? 特点。例如，<https://www.google.com/?q=search+string> 告诉 Google 您正在搜索术语搜索字符串。查询参数出现在最早的统一资源标识符 (URI) 规范中，¹¹ 但它们旨在提供额外的参数来阐明 URI，而不是用作将数据上传到 Web 服务器的一种方式。URL 在长度和内容方面也有限制（例如，这里不能发送二进制数据），一些机密数据（密码、信用卡数据等）不应存储在 URL 中，因为它在屏幕上和浏览器历史记录中很容易看到，或者如果共享 URL 则可能包含在内。

因此，POST 通常是一种更好的发送数据的方式，而且数据不那么可见（尽管在通过纯 HTTP 而不是安全 HTTPS 发送时仍应小心处理此数据，正如我稍后讨论的那样）。另一个区别是 GET 请求是幂等的，而 POST 请求不是，这意味着对同一个 URL 的多个 GET 请求应该总是返回相同的结果，而对同一个 URL 请求的多个 POST 请求可能不会。例如，如果您刷新网站上的标准页面，它应该显示相同的内容。如果您从电子商务网站刷新确认页面，您的浏览器可能会询问您是否确定要重新提交数据，这可能会导致您进行额外购买（尽管电子商务网站应将其应用程序写入确保这种情况不会发生！）。

HTTP REQUEST HEADERS

HTTP/0.9 只有一行来获取资源，而 HTTP/1.0 引入了 HTTP 标头。这些标头允许向服务器提供附加信息的请求，它可以使用户这些信息来决定如何处理请求。HTTP 标头在原始请求行之后的单独行中提供。一个 HTTP GET 请求从此改变

获取/page.html

对此

```
GET /page.html HTTP/1.0  Header1:  
Value1  
标头 2:值 2
```

或（没有标题）到

获取/page.html HTTP/1.0

也就是说，一个可选的版本部分被添加到初始行（如果未指定，默认为 HTTP/0.9），并且一个可选的 HTTP 标头部分后跟两个回车符/换行符（为简洁起见，此后称为返回字符）

¹¹ <https://tools.ietf.org/html/rfc1630>

HTTP 的语法和历史

而不是一个。第二个换行符是发送一个空行所必需的，它用于指示（可选的）请求标头部分已完成。

HTTP 标头由标头名称、冒号和标头内容指定。根据规范，标头名称（尽管不是内容）不区分大小写。当您以空格或制表符开始每一行时，标题可以跨越多行，但不推荐这种做法；很少有客户端或服务器使用这种格式并且可能无法正确处理它们。可能会发送多个相同类型的标头；它们在语义上与发送以逗号分隔的版本相同。因此

```
GET /page.html HTTP/1.0 Header1: Value1
```

标头 1: 值 2

应以同样的方式对待

```
GET /page.html HTTP/1.0 Header1:  
Value1, Value2
```

尽管 HTTP/1.0 定义了一些标准标头，但此示例还演示了 HTTP/1.0 允许提供自定义标头（在本示例中为 Header1），而无需更新协议版本。该协议被设计为可扩展的。然而，该规范明确指出“这些字段不能假定为接收者可识别的”并且可以被忽略，而标准标头应由符合 HTTP/1.0 的服务器处理。

典型的 HTTP/1.0 GET 请求是

```
GET /page.html HTTP/1.0 接受:text/  
html,application/xhtml+xml,image/jxr/*/* 接受编码:gzip, deflate, br  
接受语言:en-GB,en-US;q=0.8,en;q=0.6  
连接:keep-alive 主机:www.example.com 用户代理:MyAwesomeWebBrowser  
1.1
```

此示例告诉服务器您可以接受哪些格式的响应（HTML、XHTML、XML 等），您可以接受各种编码（例如 gzip、deflate 和 brotli，它们是用于压缩发送的数据的压缩算法通过 HTTP），以及您喜欢什么语言（英国英语，然后是美国英语，然后是任何其他形式的英语），以及您使用的是什么浏览器（MyAwesomeWebBrowser 1.1）。它还告诉服务器保持连接打开（我稍后会回到这个主题）。整个请求是用两个返回字符完成的。从这里开始，出于可读性原因，我排除了返回字符。您可以假设请求中的最后一行后跟两个返回字符。

HTTP 响应代码

来自 HTTP/1.0 服务器的典型响应是

```
HTTP/1.0 200 正常  
日期:2017 年 6 月 25 日星期日 13:30:24 GMT
```

内容类型:文本/html

服务器:阿帕奇

```
<!doctype html> <html>
```

```
<头>
```

```
...ETC。
```

提供的其余 HTML 如下。如您所见,响应的第一行是响应消息的 HTTP 版本(HTTP/1.0)、一个三位数的 HTTP 状态代码(200)和该状态代码的文本描述(OK)。状态码和描述是HTTP/1.0下的新概念;在 HTTP/0.9 下,没有响应码这样的概念;错误只能在返回的 HTML 本身中给出。表 1.1 显示了 HTTP/1.0 规范中定义的 HTTP 响应代码。

表 1.1 HTTP/1.0 响应码

类别	价值	描述	细节
1xx (信息)	不适用	不适用	HTTP/1.0 没有定义任何 1xx 状态代码,但定义了类别。
2xx (成功)	200	好的	此代码是成功请求的标准响应代码。
	201	已创建	应为 POST 请求返回此代码。
	202	公认	正在处理请求,但尚未完成处理。
	204	无内容	请求已被接受并处理,但没有要发回的 BODY 响应。
3xx (重定向)	300	多项选择	不直接使用此代码。它解释说 3xx 类别意味着该资源在一个 (或多个)位置可用,并且确切的响应提供了有关资源位置的更多详细信息。
	301	永久移动 Location HTTP 响应	标头应提供资源的新 URL。
	302	临时移动 Location HTTP 响应	标头应提供资源的新 URL。
	304	未修改	此代码用于不需要再次发送 BODY 的条件响应。
4xx (客户端错误)	400	错误的请求	该请求无法理解,应在重新发送前更改。
	401	未经授权	此代码通常表示您未通过身份验证。
	403	禁止	此代码通常表示您已通过身份验证,但您的凭据无权访问。
	404	未找到	此代码可能是最著名的 HTTP 状态代码,因为它经常出现在错误页面上。

表 1.1 HTTP/1.0 响应代码（续）

类别	价值	描述	细节
5xx (服务器错误)	500	内部服务器错误由于服务器原因无法完成请求	边错误。
	501	未实现	服务器无法识别请求（例如尚未实现的 HTTP 方法）。
	502	错误的网关	服务器充当网关或代理并从下游服务器收到错误。
	503	服务不可用 服务器无法满足请求,也许因为服务器过载或停机维护。	

敏锐的读者可能会注意到 HTTP/1.0 RFC 的早期草案中缺少一些代码（203、303、402）。一些额外的代码被排除在最终发布的 RFC 之外。

其中一些代码在 HTTP/1.1 中返回,但通常具有不同的描述和含义。互联网编号分配机构 (IANA) 维护着所有 HTTP 版本的 HTTP 状态代码的完整列表,但表 1.1 中的状态代码首先在 HTTP/1.0,¹² 中定义,代表最常用的状态代码。

很明显,某些响应可能会重叠。例如,您可能想知道无法识别的请求是 400（错误请求）还是 501（未实现）。响应代码被设计成广泛的类别,每个应用程序都可以使用最适合的状态代码。该规范还指出响应代码是可扩展的,因此可以根据需要添加新代码而无需更改协议。这是对响应代码进行分类的另一个原因。现有的 HTTP/1.0 客户端可能无法理解新的响应代码（例如 504）,但它会知道请求由于某种原因在服务器端失败,并且可以像处理其他 5xx 响应代码一样处理它。

HTTP 响应标头

在第一个返回行之后是零个或多个 HTTP/1 标头响应行。请求标头和响应标头遵循相同的格式。它们后面跟着两个返回字符,然后是正文内容,如粗体所示：

```
GET /
HTTP/1.0 302 找到位置: http://
www.google.ie/?gws_rd=cr&dcr=0&ei=BWe1WYrf123456qpibwDg 缓存控制:私有;内容类型:文本/html; charset=UTF-8 日期:2017
年 9 月 10 日星期日 16:23:33 GMT 服务器:gws 内容长度:268 X-XSS-保护:1; mode=block X-Frame-选项: SAMEORIGIN
```

¹² <https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>

```
<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8">
<TITLE>302 移动</TITLE></HEAD><BODY>
<H1>302移动</H1>
文档已移动
<A href="http://www.google.ie/?gws_rd=cr&dcr=0&ei=BWe1WYrflojUgAbqpl bwDg" >此处</A></BODY></HTML>外部主机已关闭连接。
```

随着 HTTP/1.0 的发布,HTTP 语法得到了极大的扩展,使其能够创建动态的、功能丰富的应用程序,而不仅仅是最初发布的 HTTP/0.9 版本所允许的简单的文档存储库获取。HTTP 也变得越来越复杂,从大约 700 字的 HTTP/0.9 规范扩展到近 20,000 字的 HTTP/1.0 RFC。然而,即使该规范已发布,HTTP 工作组仍将其视为记录当前使用情况的权宜之计,并且已经在致力于 HTTP/1.1。正如我之前提到的,HTTP/1.0 的发布主要是为了给 HTTP 带来一些标准和文档,因为它在野外使用,而不是为客户端和服务器定义任何新的语法来实现。除了新的响应代码之外,当时使用的其他方法(如PUT、DELETE、LINK和UNLINK)和其他 HTTP 标头都列在 RFC 的附录中,其中一些将在 HTTP/1.1 中标准化。

HTTP 的成功使得工作组在向世界推出短短五年后就努力跟上实现的步伐。

1.3.3 HTTP/1.1

如您所见,HTTP 作为获取基于文本的文档的基本方式作为 0.9 版发布。该版本从文本扩展到更成熟的协议 1.0,并在 1.1 中进一步标准化和完善。正如版本控制所暗示的那样,HTTP/1.1 更像是 HTTP/1.0 的调整,不包含对协议的根本更改。从 0.9 到 1.0 是一个更大的变化,增加了 HTTP 标头。HTTP/1.1 做了一些进一步的改进,以允许优化使用 HTTP 协议(例如持久连接、强制服务器标头、更好的缓存选项和分块编码)。也许更重要的是,它提供了一个正式的标准,未来的万维网将以此为基础。尽管 HTTP 的基础很容易理解,但有许多复杂的地方可以用稍微不同的方式实现,而且缺乏正式的标准使其难以扩展。

第一个 HTTP/1.1 规范于 199713 年 1 月发布(仅在 HTTP/1.0 规范发布后九个月)。它在 199914 年 6 月被更新规范所取代,然后在 2014 年 6 月进行了第三次增强。¹⁵每个版本都使以前的版本过时。HTTP 规范现在跨越 305

¹³ <https://tools.ietf.org/html/rfc2068>

¹⁴ <https://tools.ietf.org/html/rfc2616>

¹⁵ <https://tools.ietf.org/html/rfc7230> 和 <https://tools.ietf.org/html/rfc7235>

页面和将近 100,000 个单词,这表明这个简单的协议增长了多少,以及澄清 HTTP 应该如何使用的复杂性是多么重要。事实上,在撰写本文时,该规范正在再次更新,¹⁶并且此更新预计将于 2019 年初发布。从根本上说,HTTP/1.1 与 HTTP/1.0 并没有太大区别,但网络的爆炸式增长首先它存在的二十年产生了额外的功能和需要的文档来展示如何使用它。

描述 HTTP/1.1 本身就需要一本书,但我试图在这里讨论要点,为本书后面的一些 HTTP/2 讨论提供背景和上下文。HTTP/1.1 的许多附加功能是通过 HTTP 标头引入的,这些标头本身是在 HTTP/1.0 中引入的,这意味着 HTTP 的基本结构在两个版本之间没有改变,尽管主机标头是强制性的并添加了持久连接。HTTP/1.0 的语法有两个显着变化。

强制主机头

HTTP 请求行 (例如GET命令)提供的 URL 不是绝对 URL (例如`http://www.example.com/section/page.html`)而是相对 URL (例如`/section/页.html`)。创建 HTTP 时,假定 Web 服务器将仅托管一个网站 (尽管该网站上可能有许多部分和页面)。因此,URL 的主机部分是显而易见的,因为用户必须在发出 HTTP 请求之前连接到该 Web 服务器。如今,许多 Web 服务器在同一台服务器上托管多个站点 (这种情况称为虚拟托管),因此告诉服务器您想要哪个站点以及该站点上的哪个相对 URL 非常重要。可以通过将 HTTP 请求中的 URL 更改为完整的绝对 URL 来实现此功能,但人们认为此更改会破坏许多现有的 Web 服务器和客户端。相反,该功能是通过在请求中添加主机标头来实现的:

```
GET / HTTP/1.1
Host: www.google.com
```

此标头在 HTTP/1.0 中是可选的,但 HTTP/1.1 使其成为强制性的。以下请求在技术上的格式很糟糕,因为它指定了 HTTP/1.1 但没有提供主机标头:

```
GET / HTTP/1.1
```

根据 HTTP/1.1 规范,¹⁷该请求应该被服务器拒绝 (使用 400 响应代码),尽管大多数 Web 服务器比它们应该的更宽容,并且有一个为此类请求返回的默认主机。

强制使用 Host 标头是 HTTP/1.1 中的重要一步,允许服务器更多地使用虚拟主机,因此允许巨大的

¹⁶ <https://github.com/httpwg/http-core>

¹⁷ <https://tools.ietf.org/html/rfc7230#section-5.4>

网络的增长,而无需为每个站点添加单独的网络服务器的复杂性。此外,如果没有此更改,IPv4 IP 地址的相对较低限制会更快达到。另一方面,如果没有实施该限制,也许它会有助于更早地推动向 IPv6 的迁移;相反,尽管它已经存在了 20 多年,但在撰写本文时仍处于推出过程中!

指定一个强制性的Host标头字段而不是将相对 URL 更改为绝对 URL 会引起一些争论。¹⁸ 随 HTTP/1.1 引入的 HTTP 代理允许通过中间 HTTP 服务器连接到 HTTP 服务器。代理的语法已经设置为要求所有请求的完整绝对 URL,但实际的 Web 服务器(也称为源服务器)被强制使用Hosts标头。正如我之前提到的,此更改对于避免破坏现有服务器是必要的,但强制执行它毫无疑问地表明 HTTP/1.1 客户端和服务器必须使用虚拟托管样式请求才能完全兼容 HTTP/1.1 实现。人们认为在未来的某个 HTTP 版本中,这种情况会得到更好的处理。HTTP/1.1 规范指出,“为了允许在某些未来版本的 HTTP 中将所有请求转换为绝对形式,服务器必须接受请求中的绝对形式,即使 HTTP/1.1 客户端只会在请求中将它们发送到代理人。”然而,正如您稍后将看到的,HTTP/2 并没有彻底解决这个问题,而是用:authority伪标头字段替换了主机标头(请参阅第 4 章)。

持久连接 (AKA KEEP-ALIVES)

另一个在 HTTP 中引入并被许多 HTTP/1.0 服务器支持的重要变化是持久连接的引入,即使它没有包含在 HTTP/1.0 规范中。最初,HTTP 是一个单一的请求和响应协议。客户端打开连接,请求资源,获取响应,然后关闭连接。随着网络变得更加丰富,关闭连接被证明是一种浪费。显示单个页面需要多个 HTTP 资源,因此关闭连接只是为了重新打开它会造成不必要的延迟。

此问题已通过可与 HTTP/1.0 请求一起发送的新连接HTTP 标头解决。通过在此标头中指定值Keep-Alive ,客户端要求服务器保持连接打开以允许发送其他请求:

```
GET /page.html HTTP/1.0连接:保持活动
```

服务器会像往常一样响应,但如果它支持持久连接,它会在响应中包含一个Connection: Keep-Alive 标头:

¹⁸ 有关此主题的一些讨论,请参阅<https://lists.w3.org/Archives/Public/ietf-http-wg-old/1999SepDec/0014.html>。

HTTP 的语法和历史

HTTP/1.0 200 OK 日期:2017
 年 6 月 25 日星期日 13:30:24 GMT 连接:保持活动内容类型:文本/html
 内容长度:12345 服务器:Apache

```
<!doctype html> <html>
<头>
···ETC··
```

此响应告诉客户端,一旦响应完成,它就可以在同一连接上发送另一个请求,因此服务器不必关闭与客户端的连接,只需重新打开它即可。使用持久连接时,知道响应何时完成可能会更复杂;连接关闭是一个很好的信号,表明服务器已完成非持久连接的发送!

相反,必须使用Content-Length HTTP 标头来定义响应主体的长度,并且在接收到整个主体后,客户端可以自由发送另一个请求。

客户端或服务器可以随时关闭 HTTP 连接。
 关闭可能是意外发生的(可能是由于网络连接错误),也可能是故意的(例如,如果某个连接有一段时间未使用,服务器决定关闭该连接以为其他连接重新获得一些资源)。因此,即使是持久连接,客户端和服务端都应该监控连接并能够处理意外关闭的连接。某些请求的情况变得更加复杂。例如,如果您正在电子商务网站上结帐,您可能不想在不检查服务器是否处理了初始请求之前重新发送请求。

HTTP/1.1 不仅将这个持久连接过程添加到文档标准中,而且更进一步将其更改为默认值。即使在响应中不存在Connection: Keep-Alive标头,也可以假定任何 HTTP/1.1 连接都使用持久连接。如果服务器确实想要关闭连接,无论出于何种原因,它都必须在响应中显式包含Connection: close HTTP 标头:

HTTP/1.1 200 OK 日期:2017
 年 6 月 25 日星期日 13:30:24 GMT 连接:关闭内容类型:文本/html;
 charset=UTF-8 服务器:Apache

```
<!doctype html> <html>
<头>
···ETC··
外部主机关闭连接。
```

我在本章前面的 Telnet 示例中谈到了这个主题。现在您可以再次使用 Telnet 发送以下内容：

没有 Connection: Keep-Alive 标头的 HTTP/1.0 请求。你应该看到连接在响应后被服务器自动关闭

已发送。

相同的 HTTP/1.0 请求，但带有 Connection: Keep-Alive 标头。你应该看到连接保持打开状态。

一个 HTTP/1.1 请求，有或没有 Connection: Keep-Alive 标头。你应该看到连接默认保持打开状态。

看到 HTTP/1.1 客户端在 HTTP/1.1 请求中包含这个 Connection: Keep-Alive 标头并不罕见，尽管它是默认设置并且应该假定。

同样，服务器有时会在 HTTP/1.1 响应中包含标头，尽管这是不必要的。

在类似的主题上，HTTP/1.1 添加了流水线的概念，因此应该可以通过同一个持久连接发送多个请求并按顺序返回响应。例如，如果 Web 浏览器正在处理 HTML 文档，并且发现它需要一个 CSS 文件和一个 JavaScript 文件，它应该能够同时发送对这些文件的请求并按顺序返回响应，而不是等待在发送第二个请求之前获得第一个响应。这是一个例子：

```
GET /style.css HTTP/1.1 主机:  
www.example.com
```

```
GET /script.js HTTP/1.1 主机:  
www.example.com
```

```
HTTP/1.1 200 OK 日期:  
2017 年 6 月 25 日 星期日 13:30:24 GMT 内容类型: text/css; 字符  
集=UTF-8 内容长度:1234 服务器:Apache
```

```
.风格 {  
…ETC。
```

```
HTTP/1.1 200 OK 日期:  
2017 年 6 月 25 日 星期日 13:30:25 GMT 内容类型: application/x-  
javascript; 字符集=UTF-8 内容长度:5678 服务器:Apache
```

功能（……等
等。

由于多种原因（我将在第 2 章中进行介绍），流水线从未起飞，并且在客户端（浏览器）和服务器中对流水线的支持都很差。因此，虽然持久连接允许 TCP 连接被重复用于多个请求，这是一个

良好的性能改进,HTTP/1.1 从根本上讲仍然是大多数实现的请求和响应协议。在处理该请求时,将阻止 HTTP 连接用于其他请求。

其他新功能

HTTP/1.1 引入了许多其他特性,包括

除了HTTP/1.0 中定义的GET、 POST和HEAD方法之外的新方法。这些方法是PUT、 OPTIONS以及较少使用的CONNECT、 TRACE和DELETE。

更好的缓存方法。这些方法允许服务器指示客户端将资源 (例如 CSS 文件) 存储在浏览器的缓存中,以便以后需要时可以重用。 HTTP/1.1 中引入的Cache-Control HTTP 标头比 HTTP/1.0 中的Expires标头具有更多选项。

HTTP cookie 使 HTTP 不再是无状态协议。 HTTP 响应中的字符集 (如本章的一些示例所示) 和语言的介绍。

代理支持。 认证。

新的状态代码。

尾随标头 (在第 4 章 4.3.3 节中讨论)。

HTTP 不断添加新的 HTTP 标头以进一步扩展功能,其中许多是出于性能或安全原因。 HTTP/1.1 规范并不声称是 HTTP/1.1 的最终终结,而是积极鼓励新的标头,甚至专门用一节 19 来讨论如何定义和记录标头。正如我之前提到的,其中一些标头是出于安全原因添加的,用于允许网站告诉 Web 浏览器打开某些可选的安全保护,因此它们不需要在服务器端实现 (除了能够发送标题)。曾经有一种约定,在这些标头中包含一个X-,以表明它们没有正式标准化 (X-Content-Type、 X-Frame-Options、 X-XSS-Protection),但这种约定已被弃用的 20 和新的实验标头很难与 HTTP/1.1 规范中的标头区分开来。

通常,这些标头在它们自己的 RFC (Content-Security-Policy、 22 等) 中被标准化。²¹
严格的运输安全,

¹⁹ <https://tools.ietf.org/html/rfc7231#section-8.3.1>

²⁰ <https://tools.ietf.org/html/rfc6648>

²¹ <https://tools.ietf.org/html/rfc7762>

²² <https://tools.ietf.org/html/rfc6797>

1.4 HTTPS简介

HTTP 最初是纯文本协议。 HTTP 消息未加密地通过 Internet 发送,因此在将消息路由到其目的地时,任何一方都可以读取该消息。顾名思义,互联网是计算机网络,而不是点对点系统。互联网无法控制消息的路由方式,作为互联网用户,您不知道有多少其他方会看到您的消息,因为它们是通过互联网从您的互联网服务提供商 (ISP) 发送到电信公司和其他方。因为 HTTP 是纯文本,所以消息可以在途中被拦截、读取甚至更改。

HTTPS 是 HTTP 的安全版本,它使用传输层安全性 (TLS) 协议对传输中的消息进行加密,尽管它的前身通常称为安全套接字层 (SSL),如下面的侧边栏所述。

HTTPS 为 HTTP 消息添加了三个重要概念: 加密 消息在传输过程中无法被第三方读取。完整性 消息在传输过程中未被更改,因为整个加密消息都经过数字签名,并且该签名在解密之前经过密码验证。身份验证 服务器是您打算与之交谈的服务器。

SSL、TLS、HTTPS 和 HTTP

HTTPS 使
用 SSL 或 TLS 提供加密。SSL (安全套接字层)是由 Netscape 发明的。SSLv1 从未在 Netscape 之外发布,因此第一个生产版本是 SSLv2,于 1995 年发布。SSLv3,于 1996 年发布,解决了一些不安全问题。

由于 SSL 归 Netscape 所有,因此它不是正式的互联网标准,尽管它随后被 IETF 作为历史文件发布。
SSL 被标准化为 TLS (传输层安全)。TLSv1.0b 类似于 SSLv3,但不兼容。TLSv1.1c 和 TLSv1.2d 分别于 2006 年和 2008 年紧随其后,并且更加安全。TLSv1.3 于 2018 年被批准为标准;虽然它更安全且性能更高,但需要时间才能普及。

尽管有这些更新、更安全、标准化的版本可用,但许多人认为 SSLv3 已经足够好了,因此它在很长一段时间内都是事实上的标准,尽管许多客户端也支持 TLSv1.0。然而在 2014 年,SSLv3,发现重大漏洞,不得再使用,浏览器不再支持。这种情况开始了向 TLS 的重大转变。在 TLSv1.0 中发现类似漏洞后,安全指南坚持使用 TLSv1.1 或更高版本。

所有这些历史的最终结果是人们以不同的方式使用这些缩写词。

许多人仍然将加密称为 SSL,因为它是很长时间以来的标准;其他人使用 SSL/TLS 或 TLS。有些人试图通过将其称为 HTTPS 来避免争论,尽管这个术语并不严格正确。

通常在本书中,我将加密称为 HTTPS (而不是 SSL 或 SSL/TLS) ,除非我专门谈论 TLS 协议的特定部分。同样,我将 HTTP 的核心语义称为 HTTP,无论它是用于未加密的 HTTP 连接还是加密的 HTTPS 连接。

- a <https://tools.ietf.org/html/rfc6101>
- b <https://tools.ietf.org/html/rfc2246>
- c <https://tools.ietf.org/html/rfc4346>
- d <https://tools.ietf.org/html/rfc5246>
- e <https://tools.ietf.org/html/rfc8446>
- F <https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/>
- <https://www.us-cert.gov/ncas/alerts/TA14-290A>
- H <https://tools.ietf.org/html/rfc7568>
- [www.pcisecuritystandards.org/documents/Migrating-from-SSL-Early-TLS-Info-Supp v1_1.pdf](http://www.pcisecuritystandards.org/documents/Migrating-from-SSL-Early-TLS-Info-Supp_v1_1.pdf)

HTTPS 的工作原理是使用公钥加密,它允许服务器在用户首次连接时以数字证书的形式提供公钥。您的浏览器使用此公钥加密消息,只有服务器可以解密,因为只有它具有相应的私钥。该系统允许您与网站安全通信,而无需提前知道共享密钥,这对于像互联网这样的系统至关重要,每天每一秒都有新的网站和用户来来去去。

数字证书由浏览器信任的各种证书颁发机构 (CA) 颁发并进行数字签名,这就是为什么可以验证公钥是用于您要连接的服务器的原因。HTTPS 的一个大问题是它仅表示您正在连接到该服务器,而不表示该服务器值得信赖。可以使用 HTTPS 为不同但相似的域 (examplebank.com 而不是 examplebank.com) 轻松设置假钓鱼站点。HTTPS 站点通常在 Web 浏览器中显示为绿色挂锁,许多用户认为这是安全的意思,但它仅仅意味着安全加密。

一些 CA 在颁发证书并提供扩展验证证书 (称为 EV 证书) 时会对网站进行一些额外的审查,该证书以与普通证书相同的方式加密 HTTP 流量,但也会在大多数 Web 浏览器中显示公司名称,如图1.4所示。

许多人质疑 EV 证书的好处,²³主要是因为绝大多数用户没有注意到公司名称,并且在使用 EV 或标准域验证 (DV) 证书的网站上没有任何不同的行为。中间地带的组织验证 (OV) 证书会进行一些检查,但不会在浏览器中提供额外的通知,这使得它们在技术层面上基本上毫无意义 (尽管 CA 可能在购买它们时包含额外的支持承诺) 。

²³ <https://www.tunetheweb.com/blog/what-does-the-green-padlock-really-mean/>



图 1.4 HTTPS Web 浏览器指示器

在撰写本文时,Google Chrome 团队正在研究和试验这些安全指标,²⁴试图删除它认为不必要的信息,包括方案 (http 和 https)、任何 www 前缀,甚至可能是挂锁本身 (而不是假设 HTTPS 是规范并且 HTTP 应该明确标记为不安全)。该团队也在考虑是否移除 EV。²⁵ HTTPS 是围绕 HTTP 构建的,几乎与 HTTP 协议本身无缝衔接。

它默认托管在不同的端口 (端口 443 与标准 HTTP 的端口 80 相对),并且它具有不同的 URL 方案 (<https://> 与 <http://> 相对),但它并没有从根本上改变除了加密和解密本身之外,HTTP 在语法或消息方面的使用方式。

当客户端连接到 HTTPS 服务器时,它会经历协商阶段 (或 TLS 握手)。在这个过程中,服务器提供公钥,客户端和服务器商定使用的加密方法,然后客户端和服务器协商一个共享的加密密钥,以备将来使用。(公钥加密速度慢,因此公钥仅用于协商共享密钥,用于加密未来消息,性能更好)。我在第 4 章 (第 4.2.1 节) 中详细讨论了 TLS 握手。

建立 HTTPS 会话后,将交换标准 HTTP 消息。客户端和服务器在发送和接收解密之前加密这些消息,但对于一般的 Web 开发人员或服务器管理员来说,HTTPS 和配置后的 HTTP 没有区别。除非您正在查看通过网络发送的原始消息,否则一切都会透明地发生。HTTPS 包装了标准的 HTTP 请求和响应,而不是用其他协议替换它们。

HTTPS 是一个很大的话题,远远超出了本书的范围。我会在以后的章节中再次简要介绍它,因为 HTTP/2 确实带来了一些变化。但就目前而言,重要的只是知道 HTTPS 的存在以及它工作在比 HTTP 更低的级别 (介于 TCP 和 HTTP 之间)。除非您查看加密消息本身,否则您不会发现 HTTP 和 HTTPS 之间有任何真正的区别。

²⁴ <https://blog.chromium.org/2018/05/evolving-chromes-security-indicators.html>

²⁵ <https://groups.google.com/forum/#topic.mozilla.dev.security.policy/szD2KBHfwl8%5B1 -25%5D>

对于使用 HTTPS 的 Web 服务器,您需要一个能够理解 HTTPS 并进行加密和解密的客户端,这样您就不能再使用 Telnet 向这些服务器发送示例 HTTP 请求。 OpenSSL 程序提供了一个 s_client 命令,您可以使用它向 HTTPS 服务器发送 HTTP 命令,类似于 Telnet 的使用方式:

```
openssl s_client -crlf -connect www.google.com:443 -quiet GET / HTTP/1.1 主机:  
www.google.com
```

HTTP/1.1 200 OK等
等。

然而,我们即将结束使用命令行工具检查 HTTP 请求的有用性。在下一节中,我将简要介绍浏览器工具,它们提供了一种更好的方式来查看 HTTP 请求和响应。

1.5 用于查看、发送和接收 HTTP 消息的工具 大多数网页。有几种工具可以让您以比 Telnet 更好的方式查看和发送 HTTP 请求。许多这些工具都可以从您用 来与网络交互的主要工具中使用:您的网络浏览器。

1.5.1 在网络浏览器中使用开发者工具

所有网络浏览器都带有所谓的开发者工具,可以让您看到网站背后的许多细节,包括 HTTP 请求和响应。

您可以通过按键盘快捷键 (对于大多数浏览器在 Windows 上为 F12,或在 Apple 计算机上为 Option+Command+I) 或通过右键单击一段 HTML 并从上下文菜单中选择检查来启动开发人员工具。开发人员工具有各种选项卡显示页面背后的技术细节,但出于本次讨论的目的,您最感兴趣的选项卡是网络选项卡。如果您打开开发人员工具然后加载页面,“网络”选项卡会显示所有 HTTP 请求,单击其中一个会生成更多详细信息,包括请求和响应标头。图 1.5 显示了您在加载 <https://www.google.com> 时获得的 Chrome 开发者工具。

URL 像往常一样在地址栏 (1) 的顶部输入。请注意挂锁和 https:// 方案,表明 Google 正在成功使用 HTTPS (尽管如前所述,Chrome 可能正在改变这一点)。网页再次像往常一样返回到地址栏下方。但是,如果您在打开开发者工具的情况下加载此页面,您会看到一个包含各种选项卡的新部分。单击网络选项卡 (2) 显示 HTTP 请求 (3),包括 HTTP 方法(GET)、响应状态(200)、协议(http/1.1)和方案(https)等信息。您可以通过右键单击列标题来更改显示的列。议定书,

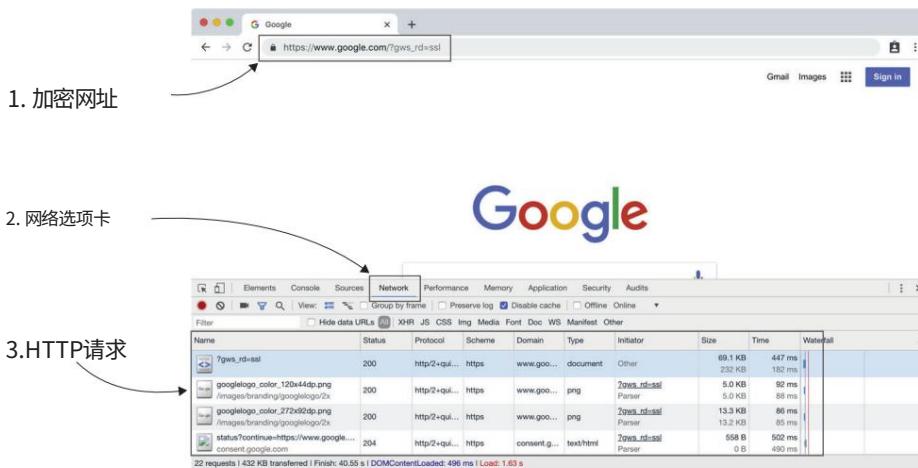


图 1.5 开发者工具 Chrome 中的网络选项卡

Scheme 和 Domain 列在默认情况下不显示,例如,对于某些站点 (例如 Twitter) ,您会在该列中看到h2 表示 HTTP/2 甚至可能是http/2+quic (Google) 表示更新的我在第 9 章讨论的协议。

图 1.6 显示了单击第一个请求 (1) 时发生的情况。右侧部分被选项卡式视图取代,您可以在其中查看响应头 (2) 和请求头 (3)。我在本章中讨论了很多但不是所有这些标题。

HTTPS 由浏览器处理,因此开发者工具仅显示加密前的 HTTP 请求消息和加密后的响应消息

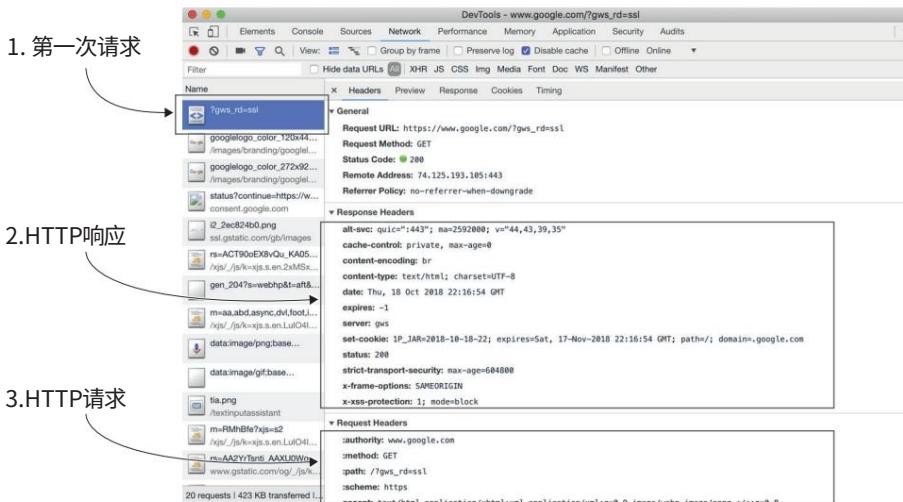


图 1.6 在 Chrome 的开发者工具中查看 HTTP 标头

用于查看、发送和接收 HTTP 消息的工具

解密。在大多数情况下，HTTPS 在设置后可以忽略，前提是您有合适的工具来为您处理加密和解密。此外，大多数浏览器的开发者工具可以正确显示媒体，因此图像可以正确显示，代码（HTML、CSS 和 JavaScript）通常可以格式化以便于阅读。

在整本书中，我都会回到开发人员工具。如果您不熟悉您的网站或您使用的热门网站，您应该熟悉浏览器的开发者人员工具。

1.5.2 发送HTTP请求

尽管 Web 浏览器的开发人员工具是查看原始 HTTP 请求和响应的最佳方式，但它们在允许您发送原始 HTTP 请求方面出奇地差。除了只能用于发送简单 GET 请求的地址栏以及网站构建的任何功能（例如，通过 HTML 表单进行 POST）之外，这些工具很少为您提供发送任何其他原始 HTTP 消息的能力。

高级 REST 客户端应用程序²⁶为您提供了一种发送原始 HTTP 消息并查看响应的方法。向 URL <https://www.google.com> (2)发送 GET 请求 (1)，然后单击发送 (3) 以获取响应 (4)，如图 1.7 所示。请注意，该应用程序会为您处理 HTTPS。

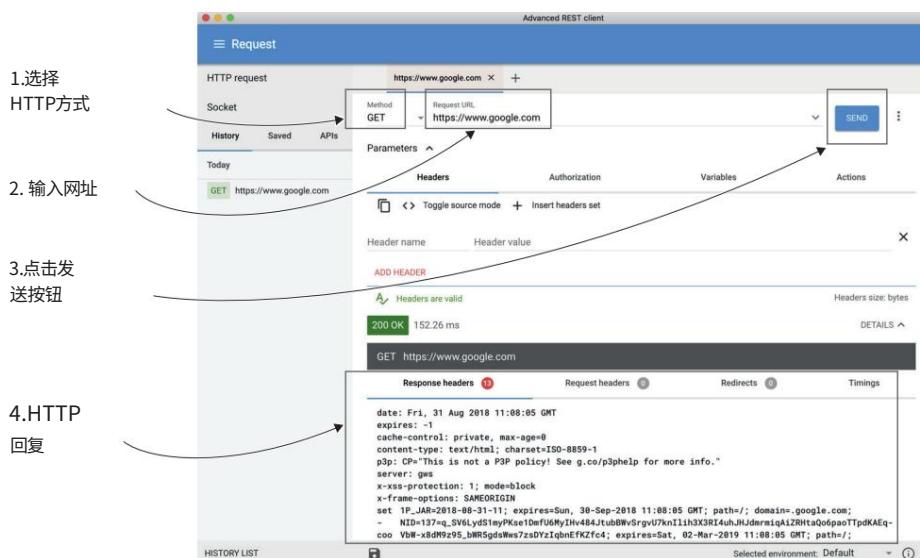


图 1.7 高级 REST 客户端应用程序

²⁶ <https://install.advancedrestclient.com> (注意：必须在 Chrome 中打开。)

使用此应用程序与使用浏览器没有什么不同,但高级 REST 客户端还允许您发送其他类型的 HTTP 请求 (例如POST和PUT)并设置要发送的标头或正文数据。 Advanced REST Client 最初是作为 Chrome 浏览器扩展程序²⁷使用的,但后来被转移到一个单独的应用程序中。类似的浏览器扩展工具就像 Advanced REST Client,包括 Postman (Chrome)、 Rested²⁸ RESTClient²⁹ (Firefox) 和 RESTMan³⁰ (Opera),所有这些工具都具有类似的功能。

1.5.3 其他查看和发送HTTP请求的工具

您可以使用许多其他工具在浏览器外部发送或查看 HTTP 请求。

其中一些从命令行 (例如 curl³¹ wget³²和 httpie³³)工作,而一些与桌面客户端 (例如 SOAP-UI³⁴)一起工作。

如果您希望查看较低级别的流量,您可能需要考虑 Chrome 的网络内部页面或网络嗅探器程序,例如 Fiddler³⁵和 Wire shark。³⁶我会在后面的章节中查看其中一些工具,当我查看 HTTP/2 的详细信息,但目前本节中提到的工具应该足够了。

概括

HTTP 是网络的核心技术之一。 浏览器发出多个 HTTP 请求来加载网页。 HTTP 协议最初是一个简单的基于文本的协议。 HTTP 变得越来越复杂,但基本的基于文本的格式没有改变

在过去的20年里。

HTTPS 加密标准 HTTP 消息。 有多种工具可用于查看和发送 HTTP 消息。

²⁷ <https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo>

²⁸ <https://addons.mozilla.org/en-US/firefox/addon/rested/>

²⁹ <https://addons.opera.com/en/extensions/details/restman/>

³⁰ <https://curl.haxx.se/>

³¹ <https://www.gnu.org/software/wget/>

³² <https://www.wireshark.org/>

³³

³⁴

³⁵

³⁶

通往 HTTP/2 的道路

本章涵盖检查 HTTP/1.1 中
的性能问题了解 HTTP/1.1 性能问题的解决方法

调查 HTTP/1 问题的真实示例 SPDY 及其如何改进
HTTP/1 SPDY 如何标准化为 HTTP/2 Web 性
能在 HTTP/2 下如何变化

为什么我们需要 HTTP/2? Web 在 HTTP/1 下工作正常,不是吗?到底什么是 HTTP/2?在本章中,我将用真实世界的例子来回答这些问题,并说明为什么 HTTP/2 不仅是必要的,而且早就该出现了。

HTTP/1.1 是大多数 Internet 的基础,对于一项已有 20 年历史的技术而言,它一直运行良好。然而,在那段时间里,网络使用呈爆炸式增长,我们已经从简单的静态网站转变为涵盖网上银行、购物、预订假期、观看媒体、社交以及我们生活的几乎所有其他方面的完全交互式页面。

随着办公室和家庭的宽带和光纤等技术的发展,互联网的可用性和速度正在提高,这意味着速度比互联网推出时用户必须处理的旧拨号速度快很多倍。

甚至移动技术也见证了 3G 和 4G 等技术以合理的消费级价格在移动中带来宽带级速度。

尽管下载速度的增长令人印象深刻,但对更快速度的需求已经超过了这种增长。宽带速度可能会继续增加一段时间,但其他限制无法轻易解决。正如您将看到的,延迟是浏览网页的一个关键因素,而延迟从根本上受到光速的限制 物理学认为光速不能增加的普遍常数。

2.1 HTTP/1.1 和当前的万维网

在第 1 章中,您了解到 HTTP 是一种请求-响应协议,最初设计用于请求单项纯文本内容,并在完成后结束连接。HTTP/1.0 引入了其他媒体类型,例如允许在网页上显示图像,而 HTTP/1.1 确保连接不会默认关闭(假设网页需要更多请求)。

这些改进很好,但是自 HTTP 的最后一次修订 (1997 年的 HTTP/1.1,尽管正式规范已经澄清了几次,并且在撰写本文时正在再次澄清,如第 3 章中所述,互联网已经发生了很大变化)。HTTP Archive 的趋势站点<https://httparchive.org/reports/state-of-the-web>可以让您看到过去八年来网站的增长情况,如图 2.1 所示。忽略 2017 年 5 月左右的小幅下降,这是由于 HTTP Archive.1 的测量问题造成的

如您所见,一般网站请求 80 到 90 个资源并下载近 1.8 MB 的数据 通过网络传输的数据量,包括使用 gzip 或类似应用程序压缩的文本资源。未压缩的网站现在超过 3 MB,这会导致移动设备等受限设备出现其他问题。

不过,该平均值存在很大差异。查看 Alexa Top 10 网站以美国的 sites2 为例,你看到的结果如表 2.1 所示。

该表显示,一些网站 (如维基百科和谷歌) 经过大量优化并且需要很少的资源,但其他网站加载数百个资源和数兆字节的数据。因此,查看平均网站甚至这些平均统计数据的价值之前一直受到质疑。3无论如何,很明显,趋势是在越来越多的资源中获取越来越多的数据。网站的发展主要是由于变得更加媒体丰富,图像和

¹ <https://github.com/HTTPArchive/legacy.httparchive.org/issues/98#issuecomment-301641938>

² www.alexa.com/topsites/countries/US <https://speedcurve.com/blog/web-性能页面膨胀/>

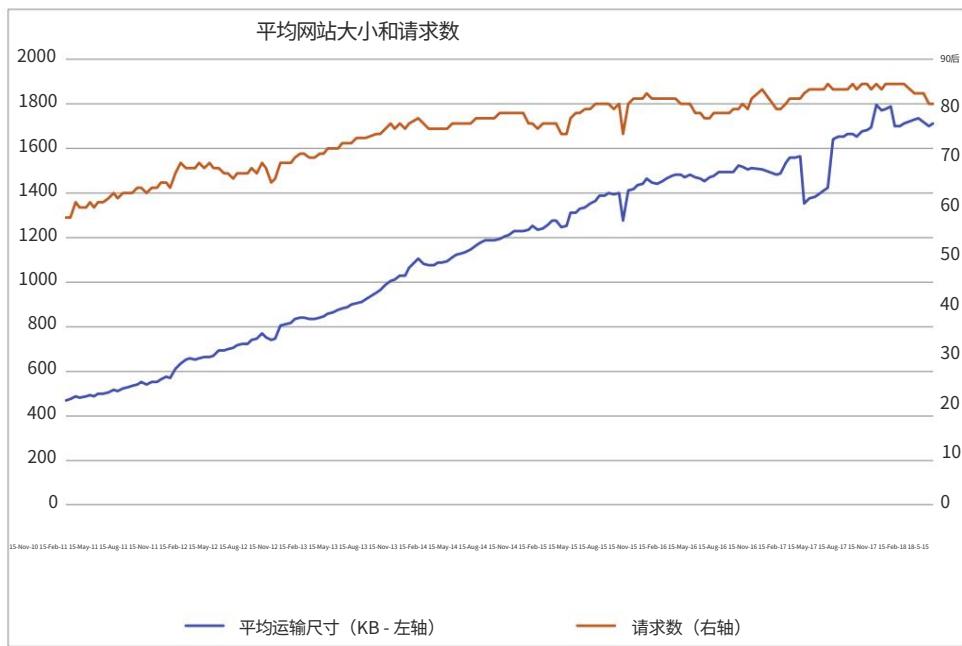


表 2.1 美国排名前 10 的网站人气排序

人气	地点	请求数	尺寸
1 [†]	https://www.google.com	17	0.4MB
2 [†]	https://www.youtube.com	75	1.6MB
3 [†]	https://www.facebook.com	172	2.2MB
4 [†]	https://www.reddit.com	102	1.0 MB
5 [†]	https://www.amazon.com	136	4.46 MB
6 [†]	https://www.yahoo.com	240	3.8MB
7	https://www.wikipedia.org	7	0.06MB
8 [†]	https://www.twitter.com	117	4.2MB
9	https://www.ebay.com	160	1.5MB
10	https://www.netflix.com	44	1.1MB

[†] <https://httparchive.org/reports/state-of-the-web>

视频是大多数网站的常态。此外，网站变得越来越复杂，需要多个框架和依赖项才能正确显示其内容。

网页最初是静态页面，但随着 Web 变得更具交互性，网页开始在服务器端动态生成，例如通用网关接口 (CGI) 或 Java Servlet/Java 服务器页面 (JSP)。下一阶段从服务器端生成的完整页面转移到基本的 HTML 页面，辅以客户端 JavaScript 调用的 AJAX（异步 JavaScript 和 XML）。这些 AJAX 调用向 Web 服务器发出额外请求，以允许更改网页内容，而无需重新加载整个页面或要求在服务器端动态生成基本图像。理解这一点的最简单方法是查看网络搜索的变化。在 web 的早期，在搜索引擎出现之前，网站目录和页面是在 web 上查找信息的主要方式，它们是静态的，只是偶尔更新。然后第一个搜索引擎出现了，允许用户提交搜索表单并从服务器获取结果（动态页面在服务器端生成）。如今，大多数搜索网站都会在您键入时在下拉菜单中提供建议，甚至在您单击“搜索”之前。谷歌更进一步，在用户输入时显示结果（尽管它在 2017 年夏天取消了这一功能，因为更多的搜索转移到移动设备上，这种功能在移动设备上的意义不大）。

除了搜索引擎之外的各种网页也大量使用 AJAX 请求，从加载新帖子的社交媒体网站到在新闻进来时更新主页的新闻网站。所有这些额外的媒体和 AJAX 请求允许网站更有趣的网络应用程序。然而，HTTP 协议在设计时并没有考虑到资源的巨大增加，而且该协议在其简单的设计中存在一些基本的性能问题。

2.1.1 HTTP/1.1 的根本性能问题

想象一个简单的网页，其中包含一些文本和两个图像。假设一个请求需要 50 毫秒 (ms) 才能通过 Internet 传输到 Web 服务器，并且该网站是静态的，因此 Web 服务器从文件服务器获取文件并将其发回。比如说，在 10 毫秒内。同样，Web 浏览器需要 10 毫秒来处理图像并发送下一个请求。这些数字是假设的；如果您有一个动态创建页面的内容管理系统 (CMS)（例如，WordPress 运行 PHP 来处理页面），则 10 毫秒服务器时间可能不准确，具体取决于服务器上正在进行的处理和/或数据库。此外，与 HTML 页面相比，图像可能很大，发送时间也更长。我们将在本章后面查看实际示例，但对于这个简单的示例，HTTP 下的流程如图 2.2 所示。

方框代表客户端或服务器端的处理，箭头代表网络流量。在这个假设的示例中，显而易见的是来回发送消息花费了多少时间。在绘制所需的 360 毫秒中

HTTP/1.1 和当前的万维网

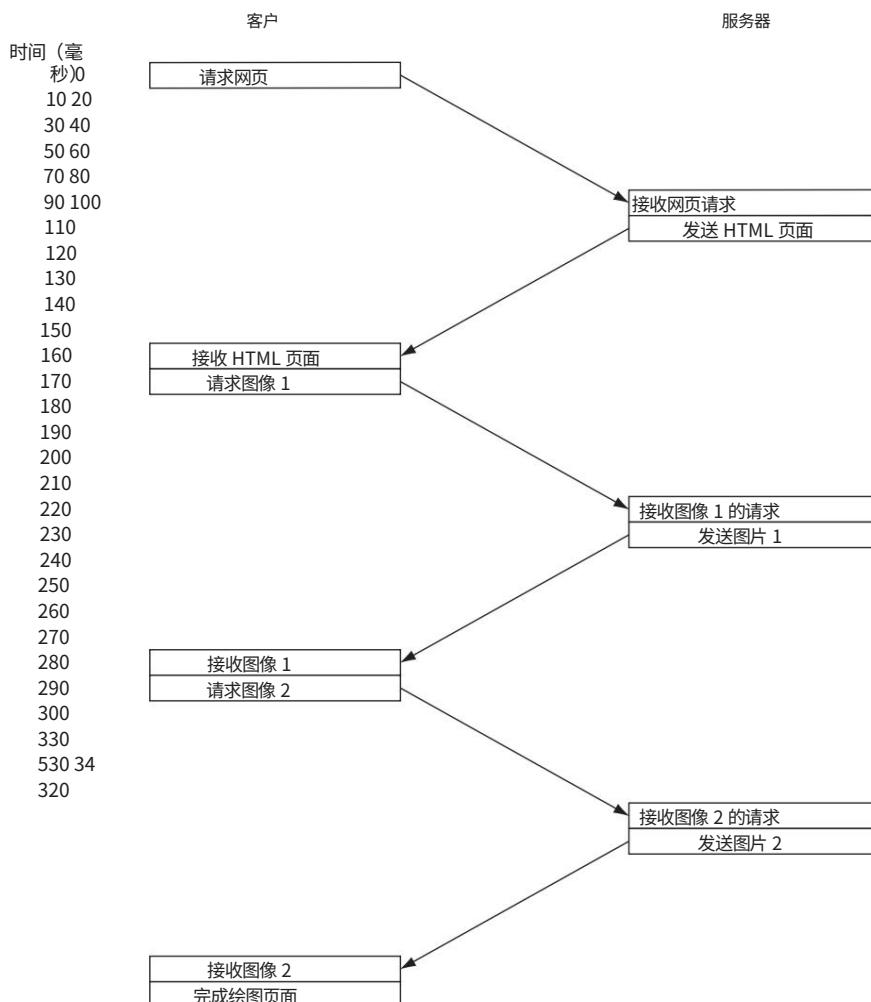


图 2.2 基本示例网站通过 HTTP 的请求-响应流

完整的页面,只有 60 毫秒用于处理客户端或浏览器端的请求。总共有 300 毫秒,或超过 80% 的时间,用于等待消息在互联网上传输。在此期间,网络浏览器和网络服务器在这个例子中都没有做太多事情;这个时间被浪费了,是 HTTP 协议的一个主要问题。在 120 毫秒标记处,在浏览器请求图像 1 之后,它知道它需要图像 2,但在发送请求之前等待连接空闲,直到 240 毫秒标记才会发生。这个过程效率低下,但有很多方法可以解决,稍后您将看到。例如,大多数浏览器会打开多个连接。关键是基本的 HTTP 协议效率很低。

大多数网站并非仅由两张图片组成,图 2.2 中的性能问题随着需要下载的资产数量的增加而增加尤其是对于相对于网络请求在两侧进行少量处理的较小资产而言和响应时间。

现代互联网最大的问题之一是延迟而不是带宽。延迟衡量将单个消息发送到服务器所需的时间,而带宽衡量用户可以下载这些消息的量。新技术一直在增加带宽(这有助于解决网站规模的增加),但延迟并没有改善(这阻止了请求数量的增加)。延迟受到物理学(光速)的限制。通过光纤电缆传输的数据已经非常接近光速了。无论技术改进多少,在这里都只能获得一点点。

谷歌的 Mike Belshe 做了一些实验⁵表明我们正在达到增加带宽的收益递减点。我们现在可以流式传输高清电视,但我们的网上冲浪并没有以同样的速度变得更快,即使在快速的互联网连接上,网站加载也需要几秒钟。这

如果没有针对基础的解决方案,互联网就无法继续以现有的速度增长
HTTP/1.1 的心理性能问题:发送和接收即使是很小的 HTTP 消息也会浪费太多时间。

2.1.2 HTTP/1.1 流水线

如第 1 章所述,HTTP/1.1 试图引入流水线,它允许在收到响应之前发送并发请求,以便可以并行发送请求。最开始的 HTML 还是需要单独请求,但是当浏览器看到需要两张图片的时候,就可以一个接一个地请求了。如图 2.3 所示,流水线减少了 100 毫秒,或者在这个简单的假设示例中减少了三分之一的时间。

流水线技术本应为 HTTP 性能带来巨大的改进,但由于多种原因,它难以实现,容易被破坏,并且没有得到 Web 浏览器或 Web 服务器的良好支持。⁶因此,它很少被使用。例如,主要的 Web 浏览器都不使用流水线。⁷

即使更好地支持流水线,它仍然需要按照请求的顺序返回响应。如果图像 2 可用,但图像 1 必须从另一台服务器获取,则图像 2 响应等待,即使应该可以立即发送该文件。此问题称为线头(HOL)阻塞,在其他网络协议和 HTTP 中很常见。我在第 9 章讨论了 TCP HOL 阻塞问题。

⁵ <https://docs.google.com/a/chromium.org/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDoxMzcyOWI1N2l4Yzl3NzE2> <https://tools.ietf.org/html/draft-nottingham-http-pipeline-01#section-3> https://en.wikipedia.org/wiki/HTTP_pipelining#Implementation_in_web_browsers

⁷

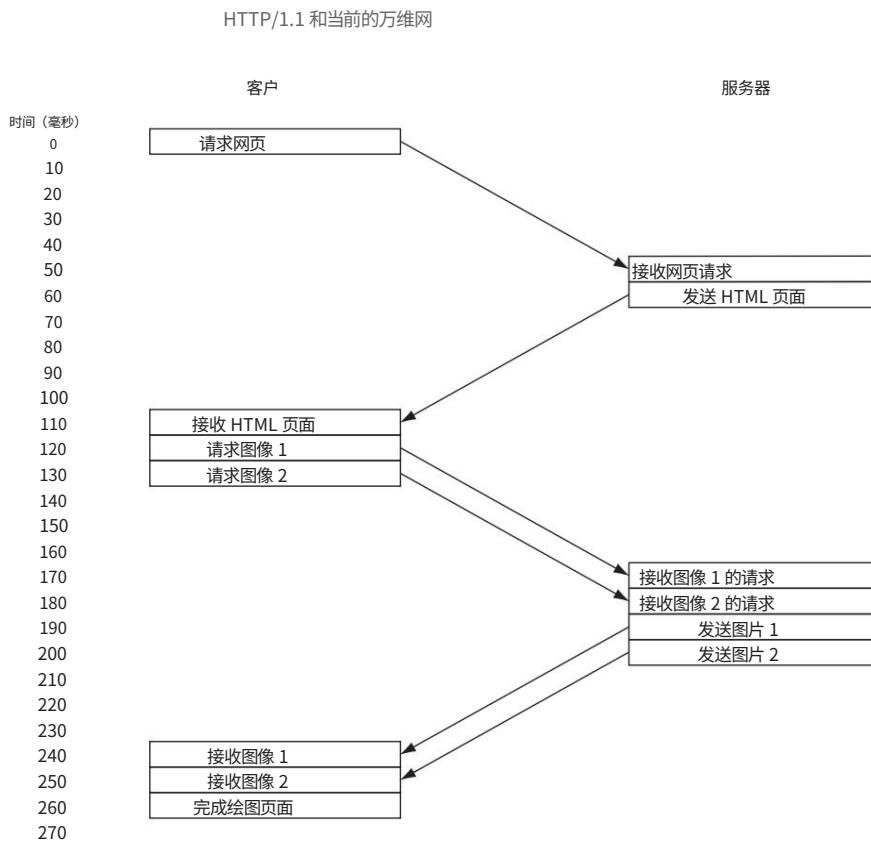


图 2.3 用于基本示例网站的带流水线的 HTTP

2.1.3 Web 性能测量的瀑布图

图 2.2 和 2.3 中显示的请求和响应流通常显示为瀑布图,左边是资产,右边是递增的时间。对于大量资源,这些图表比图 2.2 和 2.3 中使用的流程图更易于阅读。图 2.4 显示了我们假设的示例站点的瀑布图,图 2.5 显示了使用流水线时的同一点站。



图 2.4 示例网站瀑布图

在这两个示例中,第一条垂直线表示可以绘制初始页面的时间(称为首次绘制时间或开始渲染),第二条垂直线表示页面完成的时间。浏览器经常尝试在图像被渲染之前绘制页面



图 2.5 流水线示例网站的瀑布图

下载,然后再填充图像,因此图像通常介于这两个时间之间。这些示例很简单,但也可能变得复杂,正如我在本章后面的一些真实示例中向您展示的那样。

各种工具,包括 WebPagetest⁸和 Web 浏览器开发工具(在第 1 章末尾简要介绍),生成瀑布图,在审查 Web 性能时理解这些是很重要的。大多数这些工具将每个资产的总时间分解为域名服务(DNS)查找和 TCP 连接时间等组件,如图 2.6 所示。

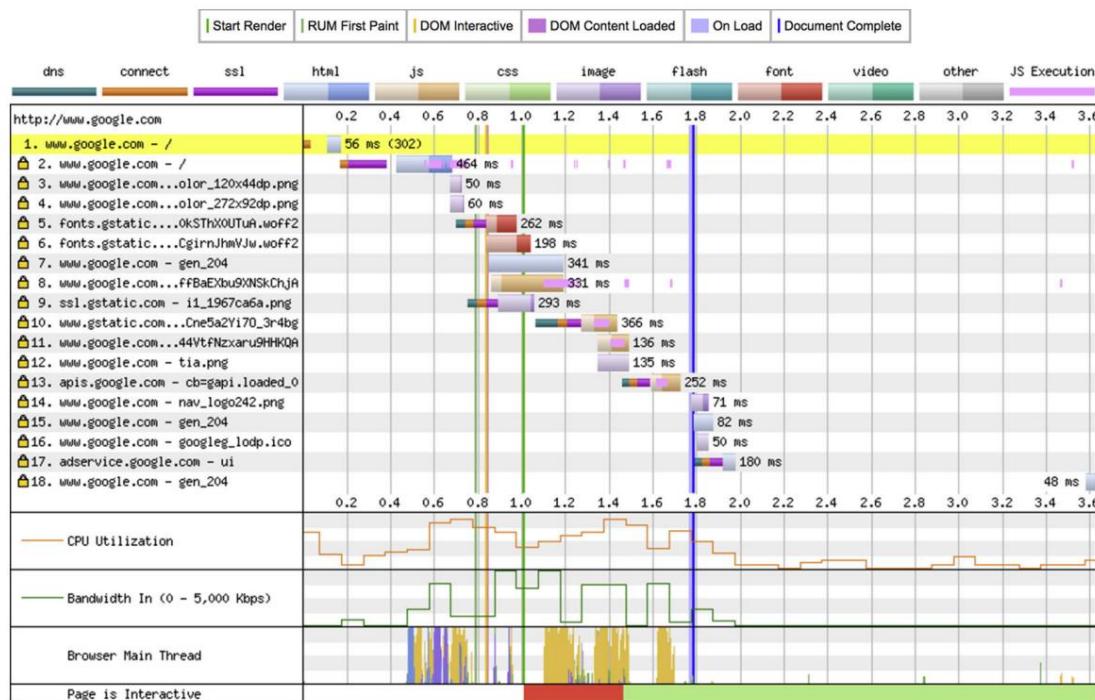


图 2.6 来自 webpagetest.org 的瀑布图

⁸ <https://www.webpagetest.org>

该图提供的信息比简单的瀑布图多得多。它将每个请求分成几个部分，包括 DNS 查找网络连接时间

HTTPS (或SSL)协商时间请求的资源 (并且还将资源负载分成两部分,请求的颜色较浅,响应下载的颜色较深)

加载页面不同阶段的各种垂直线显示 CPU 使用、网络带宽和浏览器的其他图表
主线程正在处理

所有这些信息都有助于分析网站的性能。我在整本书中大量使用瀑布图来解释这些概念。

2.2 HTTP/1.1 性能问题的变通办法

如前所述,HTTP/1.1 不是一个有效的协议

议,因为它会阻塞发送并等待响应。实际上,它是同步的;在当前请求完成之前,您不能继续进行另一个 HTTP 请求。如果网络或服务器很慢,HTTP 的性能会更差。由于 HTTP 的主要目的是从通常远离客户端的服务器请求资源,因此网络缓慢是 HTTP 的一个现实问题。对于 HTTP 的初始用例 (单个 HTML 文档),这种缓慢并不是什么大问题。但是随着网页变得越来越复杂,正确呈现它们需要越来越多的资源,缓慢成为一个问题。

针对慢速网站的解决方案催生了整个网络性能优化行业,出版了许多关于如何提高网络性能的书籍和教程。

尽管克服 HTTP/1.1 的问题并不是唯一的性能优化,但它是这个行业的很大一部分。随着时间的推移,已经创建了各种技巧、窍门和 hack 来克服 HTTP/1.1 的性能限制,它们分为以下两类:

使用多个 HTTP 连接。
发出更少但可能更大的 HTTP 请求。

其他与 HTTP 协议关系不大的性能技术包括确保用户以最佳方式请求资源 (例如首先请求关键 CSS)、减少下载量 (压缩和响应图像) 以及减少在浏览器上工作 (更高效的 CSS 或 JavaScript)。这些技术大多超出了本书的范围,尽管我会在第 6 章中回顾其中的一些技术。Manning 的书 Web Performance in Action,作者 Jeremy Wagner,⁹ 是学习更多这些技术的极好资源。

⁹ <https://www.manning.com/books/web-performance-in-action>

2.2.1 使用多个HTTP连接

解决 HTTP/1.1 阻塞问题的最简单方法之一是打开更多连接,允许并行处理多个 HTTP 请求。此外,与流水线不同,不会发生 HOL 阻塞,因为每个 HTTP 连接都独立于其他连接工作。由于这个原因,大多数浏览器为每个域打开六个连接。

为了进一步增加这个六个限制,许多网站提供来自子域 (例如static.example.com) 的静态资产,例如图像、CSS 和 JavaScript,允许 Web 浏览器为每个新域打开另外六个连接。这种技术被称为域分片 (对于那些来自非 Web 背景的读者来说,不要将其与数据库分片混淆,尽管性能原因是相似的)。除了增加并行化之外,域分片也很方便,例如减少 cookie 等 HTTP 标头 (请参阅第 2.3 节)。通常,这些共享域托管在同一台服务器上。共享相同的资源但使用不同的域名会使浏览器误认为服务器是独立的。图 2.7 显示 stackoverflow.com 使用多个域:从 Google 域加载 JQuery,从 cdn.sstatic.net 加载脚本和样式表,以及从 i.stack.imgur.com 加载图像。

Name	Status	Domain	Type
stackoverflow.com	200	stackoverflow.com	document
jquery.min.js ajax.googleapis.com/ajax/libs/jquery/1.12.4	200	ajax.googleapis.com	script
stub.en.js?v=1ec6f067df10 cdn.sstatic.net/Jss	200	cdn.sstatic.net	script
stacks.css?v=4fe27c331a7b cdn.sstatic.net/Shared	200	cdn.sstatic.net	stylesheet
primary-unified.css?v=92dbb274d371 cdn.sstatic.net/Sites/stackoverflow	200	cdn.sstatic.net	stylesheet
yQoqq.png?s=48&g=1 i.stack.imgur.com	200	i.stack.imgur.com	png
6HFc3.png i.stack.imgur.com	200	i.stack.imgur.com	png
5d55j.png i.stack.imgur.com	200	i.stack.imgur.com	png
vobok.png i.stack.imgur.com	200	i.stack.imgur.com	png

图 2.7 stackoverflow.com 的多个域

虽然使用多个 HTTP 连接听起来像是一种提高性能的简单修复方法,但它并非没有缺点。当使用多个 HTTP 连接时,客户端和服务器都会有额外的开销:启动 TCP 连接需要时间,维护连接需要额外的内存和处理。

然而,多个 HTTP 连接的主要问题是底层 TCP 协议效率低下。TCP 是一种有保证的协议,它发送具有唯一序列号的数据包,并通过检查丢失的序列号来重新请求在途中丢失的任何数据包。TCP 需要三次握手来建立,如图2.8所示。

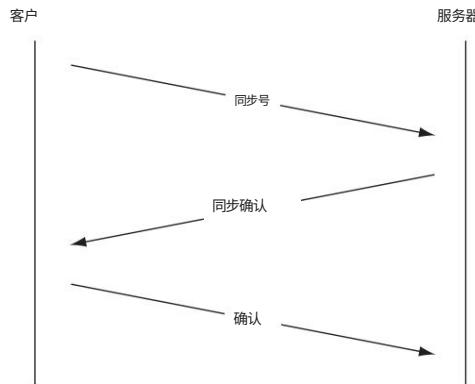


图2.8 TCP三次握手

我将详细解释这些步骤:

1客户端发送一条同步 (SYN) 消息,告诉服务器它应该期望来自该请求的所有未来 TCP 数据包所基于的序列号。

2服务器确认来自客户端的序列号并发送它自己的同步请求,告诉客户端它将为它的消息使用什么序列号。两条消息合并为一条 SYN-ACK 消息。

3最后,客户端用 ACK 确认服务器序列号信息。

在发送单个 HTTP 请求之前,此过程会增加 3 次网络传输 (或 1.5 次往返) !

此外,TCP 谨慎启动,在确认之前发送少量数据包。拥塞窗口 (CWND) 随着时间的推移逐渐增加,因为连接显示能够处理更大的大小而不会丢失数据包。TCP 拥塞窗口的大小由 TCP 慢启动算法控制。由于 TCP 是一种不想使网络过载的有保证的协议,因此在发送更多数据包之前,必须确认 CWND 中的 TCP 数据包,使用在三向握手中设置的序列号的增量。因此,对于较小的 CWND,可能需要多次 TCP 确认才能发送完整的 HTTP 请求消息。通常比 HTTP 请求大得多的 HTTP 响应也受到相同的拥塞窗口限制。随着 TCP 连接使用的越来越多,它增加了 CWND 并变得更加高效,但它总是人为地启动

即使在最快、带宽最高的网络上也是如此。我将在第 9 章回到 TCP，但现在，即使是这个快速介绍也应该表明多个 TCP 连接是有成本的。

最后，即使没有任何 TCP 设置和慢启动问题，使用多个独立连接也会导致带宽问题。例如，如果使用了所有带宽，结果可能是 TCP 超时并在其他连接上重新传输。

这些独立连接上的流量之间没有优先级概念，无法以最有效的方式使用可用带宽。

建立 TCP 连接后，安全网站需要设置 HTTPS。通过重用主连接中使用的许多参数而不是从头开始，可以在后续连接上最小化此设置，但该过程仍需要更多的网络行程，因此需要时间。我现在不会详细讨论 HTTPS 握手，但我们在第 4 章中更详细地研究它。

因此，在 TCP 和 HTTPS 级别打开多个连接是低效的，即使这样做在 HTTP 级别是一个很好的优化。HTTP/1.1 延迟问题的解决方案需要多次额外的请求和响应；因此，该解决方案很容易出现本应解决的延迟问题！

此外，当这些额外的 TCP 连接达到最佳 TCP 效率时，很可能已经加载了大部分网页，不再需要额外的连接。如果缓存了公共元素，即使浏览到后续页面也可能不需要很多资源。Mozilla 的 Patrick McManus 指出，在 Mozilla 对 HTTP/1 的监控中，“我们 74% 的活动连接只承载一个事务。”我将在本章后面介绍一些现实生活中的例子。

因此，多个 TCP 连接并不是解决 HTTP/1 问题的好方法，尽管在没有更好的解决方案可用时它们可以提高性能。

顺便说一句，这解释了为什么浏览器将连接数限制为每个域六个。尽管可以增加此数字（某些浏览器允许您这样做），但考虑到每个连接所需的开销，收益会递减。

2.2.2 减少请求

另一种常见的优化技术是减少请求，这涉及减少不必要的请求（例如通过在浏览器中缓存资产）或通过更少的 HTTP 请求请求相同数量的数据。前一种方法涉及使用 HTTP 缓存标头，在第 1 章中进行了简要讨论，并在第 6 章中进行了更详细的讨论。后一种方法涉及将资产捆绑到组合文件中。

对于图像，这种捆绑技术称为spriting。例如，如果您的网站上有很多社交媒体图标，您可以为每个图标使用一个文件。但是这种方法会导致很多低效的 HTTP 排队，因为图片会很小，所以获取它们所需的时间中有相当一部分会花在下载它们的开销上。相反，您可以将它们捆绑成一个大的

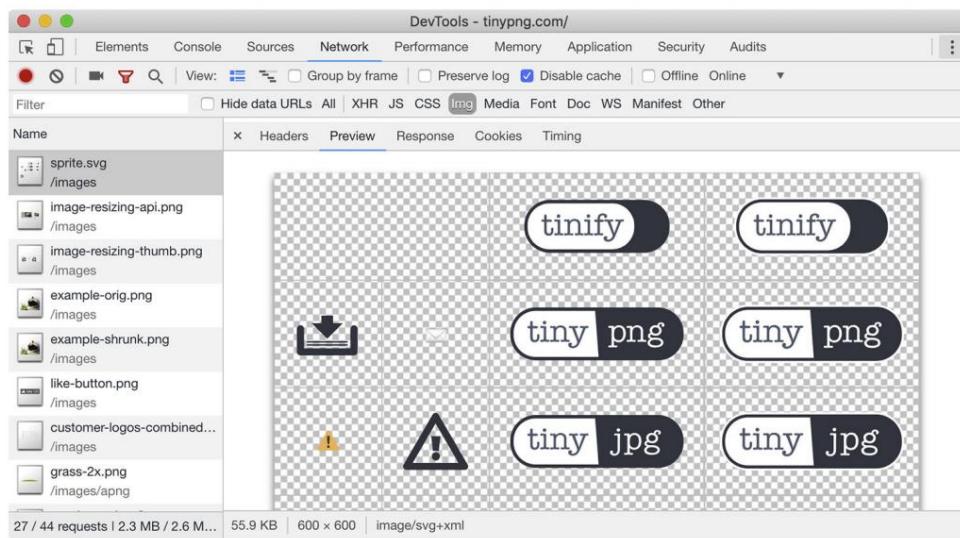


图 2.9 TinyPNG 的 Sprite 图片

图像文件,然后使用 CSS 提取图像的各个部分以有效地重新创建单个图像。图 2.9 显示了 TinyPNG 使用的一个这样的精灵图像,它在一个文件中有常见的图标。

对于 CSS 和 JavaScript,许多网站将多个文件连接起来,以便生成更少的文件,尽管组合文件中的代码量相同。这种连接通常是在最小化 CSS 或 JavaScript 以删除空格、注释和其他不必要的元素的同时完成的。这两种方法都会产生性能优势,但需要努力设置。

其他技术涉及将资源内联到其他文件中。例如,关键 CSS 通常通过<style>标签直接包含在 HTML 中。或者图像可以作为内联可缩放矢量图形 (SVG) 指令或基本 64 位编码的二进制文件包含在 CSS 中,从而节省额外的 HTTP 请求。

该解决方案的主要缺点是它引入的复杂性。创建图像精灵需要付出努力;将图像作为单独的文件提供更容易。并非所有网站都使用可以自动执行优化 (例如连接 CSS 文件) 的构建步骤。如果您的网站使用内容管理系统 (CMS),它可能不会自动连接 JavaScript 或精灵图像。

另一个缺点是这些文件中的浪费。一些页面可能正在下载大的 sprite 图像文件并且只使用其中的一两个图像。跟踪您的 sprite 文件中有多少仍在使用以及何时修剪它是很复杂的。您还必须重写所有 CSS 以从新 sprite 文件中的正确位置正确加载图像。同样,如果你连接太多并下载一个巨大的文件,即使你只需要

使用少量它。这种技术在网络层（特别是在开始时，由于 TCP 启动缓慢）和处理（因为 Web 浏览器需要处理它不会使用的数据）方面都是低效的。

最后一个问题是缓存。如果你缓存你的 sprite 图像很长时间（这样网站访问者就不会经常下载它）但是之后需要添加图像，你必须让浏览器重新下载整个文件，即使访问者可能不需要这个图片。您可以使用各种技术，例如向文件名添加版本号或使用查询参数。¹⁰但这些技术仍然很浪费。同样，在 CSS 或 JavaScript 方面，单个代码更改需要重新下载整个连接的文件。

2.2.3 HTTP/1 性能优化总结

归根结底，HTTP/1 性能优化是为了绕过 HTTP 协议中的一个基本缺陷。最好在协议级别修复此缺陷以节省每个人的时间和精力，而这正是 HTTP/2 的目标。

2.3 HTTP/1.1 的其他问题

HTTP/1.1 是一个简单的基于文本的协议。这种简单性带来了问题。

虽然 HTTP 消息的主体可以包含二进制数据（例如客户端和服务器可以同意的任何格式的图像），但请求和标头本身仍然必须是文本。文本格式对人类来说很好，但对机器来说并不是最佳的。处理 HTTP 文本消息可能很复杂且容易出错，这会带来安全问题。例如，一些针对 HTTP 的攻击基于在 HTTP 标头中注入换行符。¹¹ HTTP 是一种文本格式的另一个问题是 HTTP 消息比它们需要的更大，因为没有有效地编码数据（例如将 Date 标头表示为数字与完整的人类可读文本）和重复标头。

同样，对于具有单个请求的网络的初始用例，这种情况并不是什么大问题，但是不断增加的请求数量使这种情况变得非常低效。HTTP 标头的使用越来越多，这导致了很多重复。

例如，即使只有主页请求需要 cookie，cookie 也会随每个 HTTP 请求一起发送到域。通常，图像、CSS 和 JavaScript 等静态资源不需要 cookie。如本章前面所述，引入域分片是为了允许额外的连接，但它也用于创建所谓的无 cookie 域，出于性能和安全原因，这些域不需要向它们发送 cookie。HTTP 响应也在增长，随着诸如 Content-Security-Policy 之类的安全 HTTP 标头产生极其庞大的 HTTP 标头，基于文本的协议的不足也越来越明显。由于许多网站由 100 种或更多资源组成，大型 HTTP 标头可能会增加数十或数百 KB 的传输数据。

¹⁰ <https://css-tricks.com/strategies-for-cache-busting-css/>

¹¹ [https://www.owasp.org/index.php/Testing_for_HTTP_Splitting/Smuggling_\(OTG-INPVAL-016\)](https://www.owasp.org/index.php/Testing_for_HTTP_Splitting/Smuggling_(OTG-INPVAL-016))

性能限制只是 HTTP/1.1 可以改进的一方面。其他问题包括纯文本协议的安全和隐私问题（通过围绕它包装 HTTPS 非常成功地解决）和缺乏状态（通过添加 cookie 不太成功地解决）。在第 10 章中，我将更多地探讨这些问题。然而，对于许多人来说，如果不实施引入自身问题的解决方法，性能问题就不容易解决。

2.4 真实世界的例子我已经展示了

HTTP/1.1 对于多个请求是低效的，但是这种情况有多糟糕呢？很明显吗？让我们看几个现实世界的例子。

真实世界的网站和 HTTP/2

当我最初编写本章时，我使用的两个示例网站都不支持 HTTP/2。这两个站点后来都启用了它，但是这里显示的教训仍然与受 HTTP/1.1 影响的复杂网站的示例相关，并且这里讨论的许多细节可能与其他网站的细节相似。

HTTP/2 越来越受欢迎，任何被选为示例的站点都可能在某个时候升级。我更喜欢使用真实的、知名的网站来演示 HTTP/2 希望解决的问题，而不是使用纯粹为了证明一个观点而创建的人工示例网站，所以我保留了这两个示例网站，尽管它们是现在在 HTTP/2 上。这些站点不如它们显示的概念重要。

要在 [webpagetest.org](http://www.webpagetest.org) 重复这些测试，您可以通过指定 --disable-http2（高级设置 > Chrome > 命令行选项）来禁用 HTTP/2。如果您使用 Firefox 作为浏览器，则有类似的选项。^a 这些也是在您使用 HTTP/2 后测试您自己的 HTTP/2 性能变化的有用方法。

<https://www.webpagetest.org/forums/showthread.php?tid=14162>

2.4.1 示例网站1:amazon.com

到目前为止，我都是从理论上讲的，但现在我看的是现实世界的例子。如果你访问 www.amazon.com 并通过 www.webpagetest.org 运行它，你会得到如图 2.10 所示的瀑布图。此图演示了 HTTP/1.1 的许多问题：

第一个请求是主页，我在

图 2.11。

在发送单个请求之前，需要时间进行 DNS 查找、连接和 SSL/TLS HTTPS 协商。时间很小（在图 2.11 中略多于 0.1 秒），但它加起来了。对于第一个请求，我们无能为力。正如第 1 章中所讨论的那样，这个结果是 Web 工作方式的重要组成部分，尽管对 HTTPS 密码和协议的改进可能会减少 SSL 时间，但第一个请求是

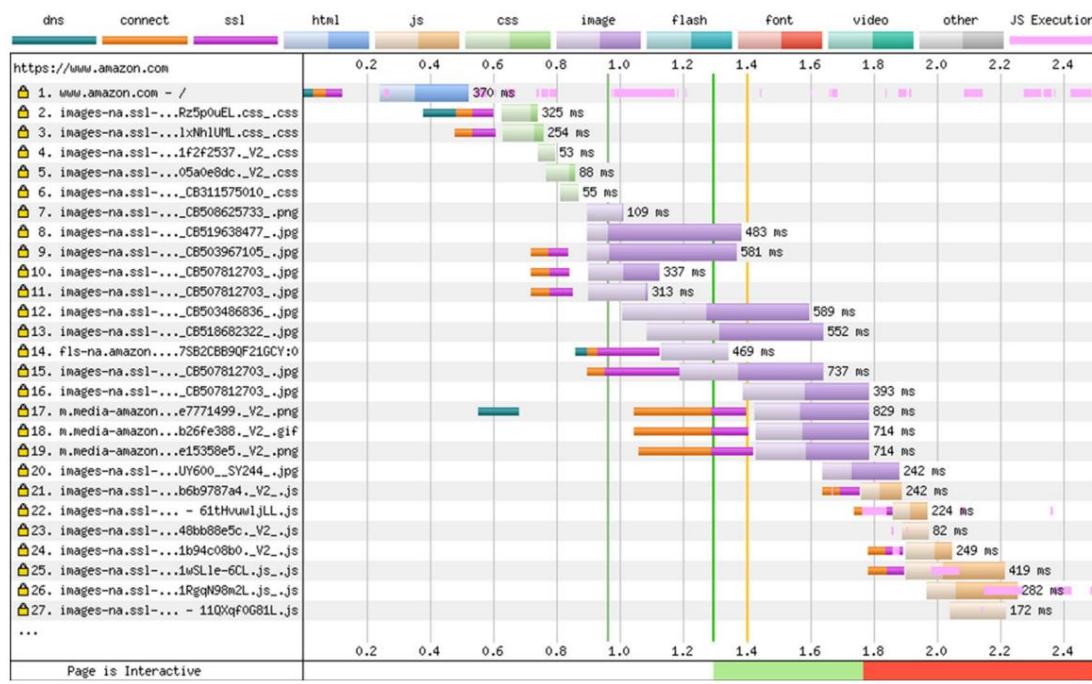


图 2.10 www.amazon.com的部分结果

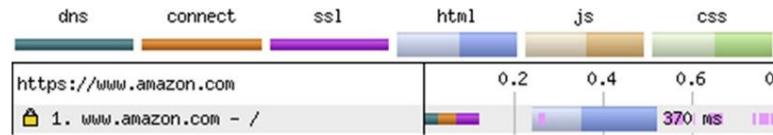


图2.11 第一次请求首页

将受到这些延误。您能做的最好的事情就是确保您的服务器响应迅速，并且理想情况下，靠近用户以尽可能缩短往返时间。在第 3 章中，我讨论了内容分发网络 (CDN)，它可以帮助解决这个问题。

在这个初始设置之后，会出现轻微的停顿。我无法解释这种暂停，这可能是由于时间稍微不准确或 Chrome 浏览器出现问题。当用 Firefox 重复测试时，我没有看到同样的差距。然后发出第一个 HTTP 请求（浅色），然后下载 HTML（浅色），由 Web 浏览器解析和处理。

HTML 引用了几个 CSS 文件，这些文件也已下载，如图 2.12 所示。

真实世界的例子

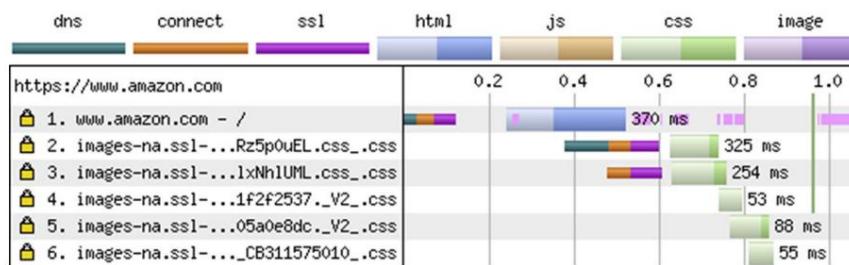


图 2.12 对 CSS 文件的五个请求

这些 CSS 文件托管在另一个域(images-na.ssl-images

amazon.com) ,如前所述,出于性能原因,它已从主域中分离出来。由于此域是独立的,您需要从头开始第二次请求,并在使用此域下载 CSS 之前进行另一次 DNS 查找、另一次网络连接和另一次 HTTPS 协商。虽然请求 1 的设置时间有些不可避免,但是这第二次设置时间被浪费了;域名分片是为了解决 HTTP/1.1 性能问题。另请注意,此 CSS 文件出现在请求 1 中 HTML 页面处理的早期,导致请求 2 在 0.4 秒标记之前开始,尽管 HTML 页面直到 0.5 秒后才完成下载。浏览器没有等待完整的 HTML 页面被下载和处理;相反,它一看到引用的域就请求额外的 HTTP 连接 (即使在本例中由于连接设置延迟,资源本身直到 HTML 完全接收后才开始下载)。第三个请求是针对同一分片上的另一个 CSS 文件。由于 HTTP/1.1 只允许同时发送一个请求,因此浏览器会创建另一个连接。这次您不需要 DNS 查找 (因为您从请求 2 中知道该域的 IP 地址) ,但在请求此 CSS 之前,您确实需要昂贵的 TCP/IP 连接设置和 HTTPS 协商时间。同样,此额外连接的唯一原因是解决 HTTP/1.1 性能问题。接下来,浏览器请求另外三个 CSS 文件,这些文件通过已经建立的两个连接加载。图中未显示的是浏览器没有立即请求这些其他 CSS 文件的原因,这将需要创建更多连接以及与之相关的成本。我查看了 Amazon 源代码,在这些 CSS 文件请求之前有一个 <script> 标记会阻止后面的需求,直到脚本被处理,这就解释了为什么请求 4.5 和 6 不会同时被请求请求 2 和 3。

这一点很重要,我稍后会提到:虽然 HTTP/1.1 的低效率是网络的一个问题,可以通过改进来解决

HTTP（如 HTTP/2 中的那些），它们远不是 Web 性能低下的唯一原因。请求 2 到 6 处理完 CSS 后，浏览器判断接下来是图片，于是开始下载图片，如图 2.13 所示。



图 2.13 图片下载

第一个 .png 文件在请求 7 中，它是一个包含多个图像的 sprite 文件（图 2.13 中未显示），这是亚马逊实施的另一个性能调整。接下来，从请求 8 开始下载一些 .jpg 文件。当其中两个图像请求正在运行时，浏览器需要建立更昂贵的连接以允许其他文件在请求 9、10、11 和 15 中并行加载，然后在请求 14、17、18 中再次加载新域，和 19。在某些情况下（请求 9、10 和 11），浏览器猜测可能需要更多连接并提前设置连接，这就是连接和 SSL 部分发生得更早的原因，也是为什么它可以在请求 7 和 8 的同时请求图像。

亚马逊添加了一项性能优化，以便在 m.media.amazon.com 需要它之前为它执行 DNS 预取¹²，但奇怪的是， fls-na.amazon.com 却没有。

这就是为什么请求 17 的 DNS 查找发生在 0.6 秒标记处，远在需要它之前。我会在第 6 章回到这个话题。

加载会在这些请求之后继续，但即使仅查看前几个请求也能识别 HTTP/1.1 的问题，因此我不会通过继续加载整个站点来强调这一点。

需要许多连接来防止任何排队，而且通常，建立此连接所花费的时间是下载资产所需时间的两倍。Web Page Test 有一个方便的连接视图¹³（如图 2.14 所示）。

¹² <https://css-tricks.com/prefetching-preloading-prebrowsing/>

¹³ https://www.webpagetest.org/result/170820_NR_53c5bf9cae67301a933947d80a32a53/1/details/#connectionView_fv_1

真实世界的例子

53

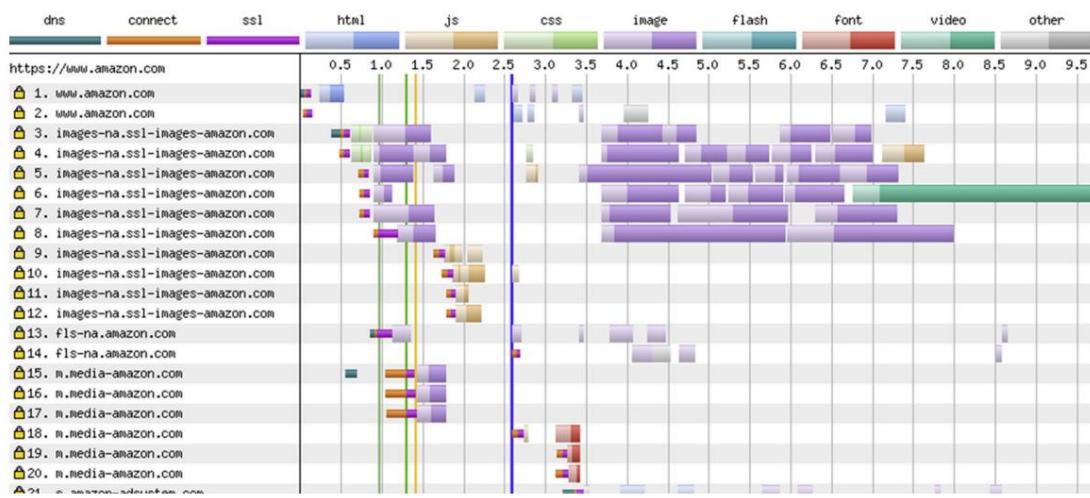


图 2.14 加载amazon.com的连接视图

您可以看到加载amazon.com需要 20 个连接到主站点,忽略广告资源,这又增加了 28 个连接 (图 2.14 中未显示)。尽管前六个images-na.ssl-images-amazon.com连接使用得很好 (连接 3-8) ,但该域的其他四个连接 (连接 9-12) 使用得较少;与许多其他连接 (例如 15、16、17、18、19 和 20)一样,它们仅用于加载一个或两个资源,从而浪费了创建该连接所需的时间。

为images-na.ssl images-amazon.com打开这四个额外连接的原因 (以及为什么 Chrome 似乎打破了每个域六个连接的限制) 很有趣,并进行了一些调查。可以使用凭据 (通常是 cookie)发送请求,但也可以在没有凭据的情况下发送请求,并由 Chrome 通过单独的连接处理。出于安全原因,由于跨域请求在浏览器中的处理方式,¹⁴ Amazon 在其中一些 JavaScript 文件请求中使用了 setAttribute (crossorigin , anonymous),没有凭据,这意味着现有连接是 用过。相反,会创建更多连接。对于在 HTML 中使用<script>标记的直接 JavaScript 请求,则不需要这样做。托管在正在加载的主域上的资源也不需要解决方法,这再次表明分片在 HTTP 级别可能效率低下。

Amazon 的示例表明,即使站点已使用必要的变通办法优化以提高 HTTP/1.1 下的性能,使用这些性能变通办法仍然会造成性能损失。这些性能解决方法的设置也很复杂。并非每个站点都想要管理多个

¹⁴ <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

域或 sprite 图像或将所有 JavaScript (或 CSS) 合并到一个文件中,并不是每个站点都拥有亚马逊的资源来创建这些优化,甚至都不知道它们。较小的站点通常没有经过优化,因此更容易受到 HTTP/1 的限制。

2.4.2 示例网站2:imgur.com

如果不进行这些优化会怎样?例如,查看因为它是一个图像共享网站, [imgur.com](#)在主页上加载了大量图像,但不会将它们拼接成单个文件。WebPagetest 瀑布图的一个子部分如图 2.15 所示。

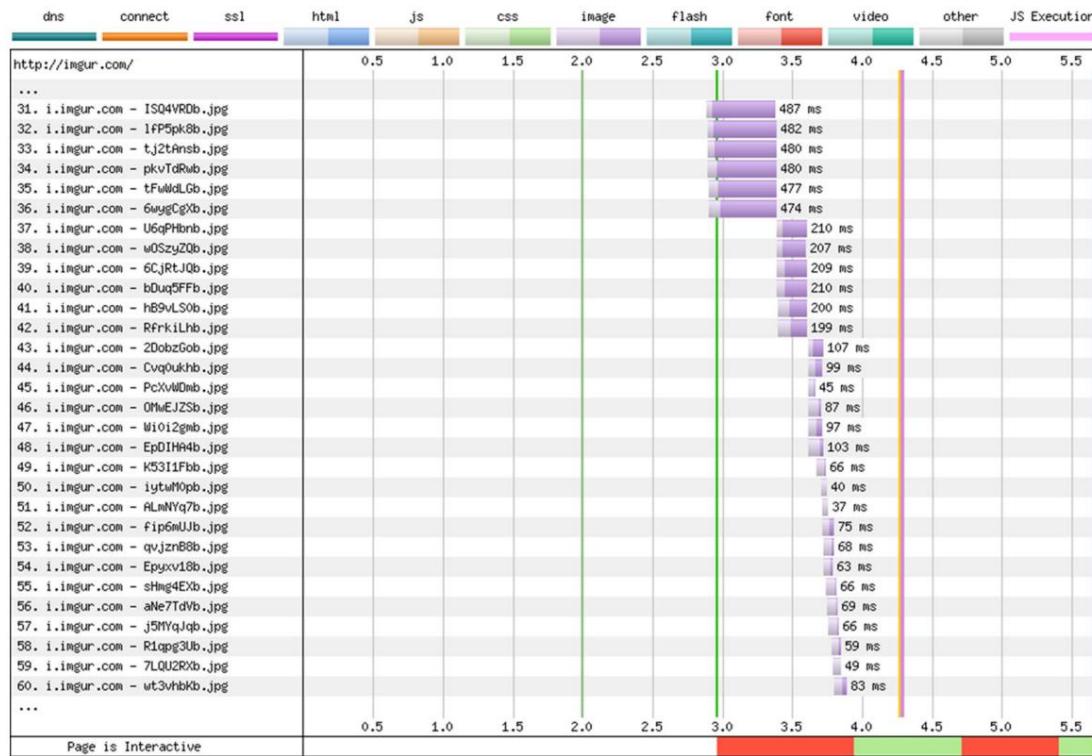


图 2.15 [imgur.com](#)的瀑布视图

我跳过了页面加载的第一部分 (在请求 31 之前), 它重复了很多[amazon.com](#)示例。您在这里看到的是最多六个连接用于加载请求 31–36;其余的都在排队。当这六个请求中的每一个都完成时,可以触发另外六个请求,然后再触发另外六个请求,这导致了这些图表名称的标志性瀑布形状。请注意,这六个资源是不相关的,并且可能在不同的时间完成 (因为它们在瀑布的更深处完成)

图表),但如果它们是类似大小的资源,则它们大约同时完成并不罕见。这一事实给人一种资源相关的错觉,但在 HTTP 级别上,它们不是(尽管它们共享网络带宽并转到同一台服务器)。

Chrome 的瀑布图,如图 2.16 所示,使问题更加明显,因为它还测量了资源可能被请求时的延迟。如您所见,对于以后的请求,在请求图像(由矩形突出显示)之前会发生较长的延迟,然后是相对较短的下载时间。

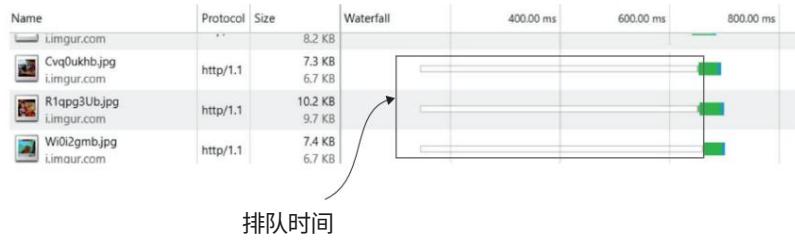


图 2.16 imgur.com 的 Chrome 开发者工具瀑布视图

2.4.3 这到底有多大问题?

尽管本章指出了 HTTP 中的低效之处,但还是有变通办法的。

然而,这些变通办法需要时间、金钱和理解来实施和保持前进,并且它们增加了它们自己的性能问题。开发人员并不便宜,让他们花时间解决低效协议是必要的,但代价高昂(更不用说许多没有意识到性能不佳对其流量的影响的站点)。¹⁵多项研究表明,速度较慢的网站会导致放弃以及访客和销售量的流失。¹⁶

您还必须考虑此问题相对于其他性能问题的严重程度。网站运行缓慢的原因有很多,从互联网连接的质量到网站的大小,再到某些网站使用的 JavaScript 数量惊人,再到性能不佳的广告和跟踪网络的激增。虽然能够快速有效地下载资源肯定是问题的一部分,但许多网站仍然很慢。显然,许多网站都担心这方面的性能,这就是他们实施 HTTP/1.1 变通办法的原因,但许多其他网站并没有这样做,因为这些变通办法需要复杂性和理解。

另一个问题是这些解决方法的局限性。这些变通办法会产生自己的低效率,但随着网站规模和规模的不断增长

¹⁵ <https://developers.google.com/web/fundamentals/performance/why-performance-matters/>

¹⁶ <https://developer.akamai.com/blog/2016/09/14/mobile-load-time-user-abandonment>

复杂性,在某些时候,即使是变通办法也不再有效。尽管浏览器为每个域打开六个连接并可能增加这个数字,但这样做的开销与收益相比已经达到收益递减点,这就是为什么浏览器首先将连接数限制为六个,即使网站所有者已尝试通过域分片解决此限制。

归根结底,每个网站都是不同的,每个网站所有者或网站开发人员都需要花时间分析网站自身的资源瓶颈,使用瀑布图等工具,看看网站是否受到 HTTP/1.1 性能问题的严重影响。

2.5 从 HTTP/1.1 迁移到 HTTP/2

自 1999 年 HTTP/1.1 出现以来,HTTP 并没有真正改变。该规范在 2014 年发布的新征求意见稿 (RFC) 中得到了澄清,但该规范更像是一种文档练习,而不是协议中的任何实际更改。更新版本 (HTTP-NG) 的工作已经开始,它本来是对 HTTP 工作方式的完全重新设计,但它在 1999 年被放弃了。一般的感觉是这个变化过于复杂,没有办法将它引入现实世界。

2.5.1 SPDY

2009 年,Google 的 Mike Belshe 和 Robert Peon 宣布他们正在研究一种名为 SPDY (发音为 “speedy” 而非首字母缩写词)的新协议。他们一直在实验室条件下对该协议进行试验,并看到了出色的结果,页面加载时间缩短了 64%。实验是在前 25 个网站的副本上运行的,而不是可能无法代表现实世界的假设网站。

SPDY 建立在 HTTP 之上,但并未从根本上改变协议,这与 HTTPS 围绕 HTTP 而未改变其底层用途的方式大致相同。HTTP 方法 (GET、POST 等) 和 HTTP 标头的概念仍然存在于 SPDY 中。SPDY 在较低级别工作,对于 Web 开发人员、服务器所有者和 (至关重要的) 用户而言,SPDY 的使用几乎是透明的。任何 HTTP 请求都被简单地转换为 SPDY 请求,发送到服务器,然后再转换回来。这个请求看起来像任何其他对更高级别应用程序的 HTTP 请求 (例如客户端的 JavaScript 应用程序和那些配置 Web 服务器的应用程序)。此外,SPDY 仅通过安全 HTTP (HTTPS) 实现,这允许消息的结构和格式对在客户端和服务器之间传递消息的所有 Internet 管道隐藏。因此,所有现有网络、路由器、交换机和其他基础设施都可以处理 SPDY 消息而无需任何更改,甚至不知道它们正在处理 SPDY 消息而不是 HTTP/1 消息。SPDY 本质上是向后兼容的,可以以最小的变化和风险引入,这无疑是它成功而 HTTP-NG 失败的一个重要原因。

HTTP-NG 试图解决 HTTP/1 的多个问题,而 SPDY 的主要目标是解决 HTTP/1.1 的性能限制。它引入了几个重要的概念来处理 HTTP/1.1 的局限性:

多路复用流 请求和响应使用单个 TCP 连接,并被分成交织的数据包,分组到单独的流中。 **请求优先级** 为了避免同时发送所有请求而引入新的性能问题,引入了请求优先级的概念。

HTTP 标头压缩 HTTP 主体早已被压缩,但现在标头也可以被压缩。

不可能使用 HTTP 在此之前的基于文本的请求和响应协议来引入这些功能,因此 SPDY 成为了二进制协议。此更改允许单个连接处理小消息,这些消息一起形成较大的 HTTP 消息,这与 TCP 本身将较大的 HTTP 消息分解为许多对大多数 HTTP 实现透明的较小 TCP 数据包的方式非常相似。 SPDY 在 HTTP 层实现了 TCP 的概念,因此多个 HTTP 消息可以同时传输。

服务器推送等其他高级功能允许服务器标记额外的资源。如果您请求主页,服务器推送可以提供显示它所需的 CSS 文件,以响应该请求。这个过程省去了请求该 CSS 文件的往返过程中的性能延迟以及内联关键 CSS 的复杂性和工作量。

谷歌处于控制主要浏览器 (Chrome) 和一些最受欢迎的网站 (如www.google.com) 的独特地位,因此它可以通过新协议进行更大规模的现实生活实验在连接的两端实现它。 SPDY 于 2010 年 9 月发布到 Chrome,到 2011 年 1 月,所有 Google 服务都支持 SPDY¹⁷ 无论以何种标准衡量,这都是一个令人难以置信的快速推出。

SPDY 几乎是立竿见影的成功,其他浏览器和服务器也迅速添加了支持。 Firefox 和 Opera 在 2012 年增加了支持。在服务器端,Jetty 增加了支持,其次是其他人,包括 Apache 和 nginx。绝大多数支持 SPDY 的网站都在后两个 Web 服务器上。引入 SPDY 的网站 (包括 Twitter、Facebook 和 WordPress) 获得了与 Google 相同的性能提升,除了初始设置外几乎没有缺点。根据 w3techs.com 的数据,SPDY 达到了所有网站的 9.1%,¹⁸ 尽管现在 HTTP/2 已经出现,浏览器已经开始取消对它的支持。自 2018 年初以来,SPDY 的使用率直线下降,如图 2.17 所示。

¹⁷ <https://groups.google.com/d/msg/spdy-dev/TCOW7Lw2scQ/T2kM5aPDydwJ>

¹⁸ <https://w3techs.com/technologies/details/ce-spdy/all/all>

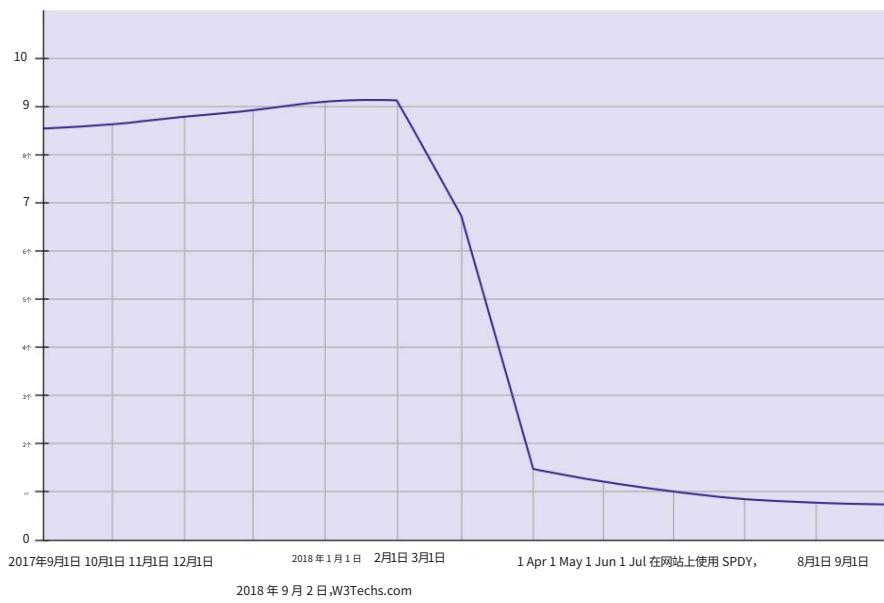


图 2.17 自 HTTP/2 推出以来,网站对 SPDY 的支持有所下降。

2.5.2 HTTP/2

SPDY 证明了 HTTP/1.1 可以改进,不是以理论上的方式,而是通过它在现实世界中的主要站点上运行的例子。2012 年,互联网工程任务组 (IETF) 的 HTTP 工作组注意到 SPDY 的成功,并征求了有关下一版本 HTTP 的建议。¹⁹ SPDY 是下一版本的自然基础,因为它已在 wild,尽管工作组明确避免这样说,更愿意对任何提案持开放态度 (尽管有些人对这一立场提出异议,如第 10 章所述)。

在考虑了其他提案的短暂时之后,SPDY 在 2012 年 11 月发布的初稿中形成了 HTTP/2 的基础。²⁰该草案在接下来的两年中略有修改以改进它 (特别是它对流和压缩的使用).我在第 4、5、7 和 8 章中详细介绍了该协议的技术细节,因此我在这里简单地介绍了这个主题。

到 2014 年底,HTTP/2 规范作为互联网的建议标准提交,并于 2015 年 5 月正式获得批准,成为 RFC 7450.²¹支持端口紧随其后,特别是因为该规范在很大程度上基于 SPDY,这许多浏览器和服务器已经实现。火狐支持

¹⁹ <https://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html>

²⁰ <https://tools.ietf.org/html/draft-ietf-httpsbis-http2-00> <https://tools.ietf.org/>

²¹ <https://rfc7540>

HTTP/2 从 2015 年 2 月开始,Chrome 和 Opera 从 2015 年 3 月开始。Internet Explorer 11、Edge 和 Safari 紧随其后。

Web 服务器迅速增加了支持,许多服务器在标准化过程中实施了各种版本。LiteSpeed²²和 H2O²³是首批支持的 Web 服务器。到 2015 年底,绝大多数互联网用户使用的主要三个 Web 服务器 (Apache、IIS 和 nginx) 都有实现,尽管它们最初被标记为实验性的并且默认情况下没有打开。

根据 w3tech.com 的数据,截至 2018 年 9 月,所有网站中有 30.1% 提供 HTTP/2。²⁴这一覆盖范围在很大程度上是由于支持 HTTP/2 的内容交付网络和大型网站,但对于三个岁的技术。正如您将在第 3 章中看到的,目前在服务器端启用 HTTP/2 需要付出相当大的努力;否则,使用率可能会更高。

要点是 HTTP/2 在这里并且可用。它已在现实生活中得到证明,并被证明可以显着提高性能,因为它解决了本章中提出的 HTTP/1.1 问题。

2.6 HTTP/2 对网络性能的意义

您已经看到了 HTTP/1 固有的性能问题,现在有了 HTTP/2 的解决方案,但是 HTTP/2 是否解决了所有 Web 性能问题,如果所有者升级到 HTTP/2?

2.6.1 HTTP/2 强大的极端例子

许多示例显示了 HTTP/2 的性能改进。我的网站上有一个,网址为<https://www.tunetheweb.com/performance-test-360/>。此页面可通过 HTTP 1.1、通过 HTTPS 的 HTTP 1.1 和通过 HTTPS 的 HTTP/2 访问。如第 3 章所述,浏览器仅通过 HTTPS 支持 HTTP/2;因此,没有 HTTPS 测试就没有 HTTP/2。此测试基于 <https://www.httpvshttps.com/> 上的类似测试,该测试排除了 HTTPS-without-HTTP/2 测试并通过三种技术加载包含 360 度图像的网页,使用一些 JavaScript 来计时负载。结果如图 2.18 所示。

该测试表明 HTTP 版本加载页面和所有图像需要 10.471 秒。HTTPS 版本花费了大致相同的时间,为 10.533 秒,这表明 HTTPS 不会像以前那样导致性能下降,而且与纯文本 HTTP 相比几乎看不出差异。事实上,多次重新运行此测试通常显示 HTTPS 比 HTTP 快一点,这没有任何意义(因为 HTTPS 涉及额外的处理),但这种额外的处理在该测试站点的误差范围内。

²² <https://blog.litespeedtech.com/2015/04/17/lsws-5-0-is-out-support-for-http2-esi-litemage-cache/>

²³ <https://h2o.example.net/> <https://w3techs.com/technologies/details/ce-http2/all/all>

²⁴

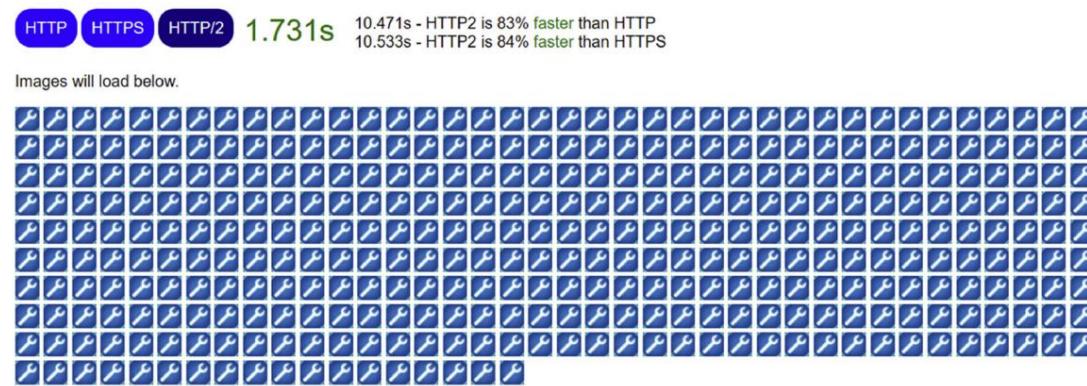


图 2.18 HTTP vs HTTPS vs HTTP/2 性能测试

真正令人惊讶的是 HTTP/2,它在 1.731 秒内加载了网站 比其他两种技术快 83%!查看瀑布图可以说明原因。比较图 2.19 和 2.20 中的 HTTPS 和 HTTP/2 图。

在 HTTPS 下,您会看到熟悉的设置多个连接的延迟,然后以六个为一组加载图像。然而,在 HTTP/2 下,图像是一起请求的,因此没有延迟。为简洁起见,我只显示了前 21 个请求,而不是所有 360 个请求,但该图说明了 HTTP/2 在加载此类站点时的巨大性能优势。另请注意,在图 2.19 中,在使用了页面的最大连接数后,浏览器选择在请求 10 中加载 Google Analytics。该请求针对不同的域

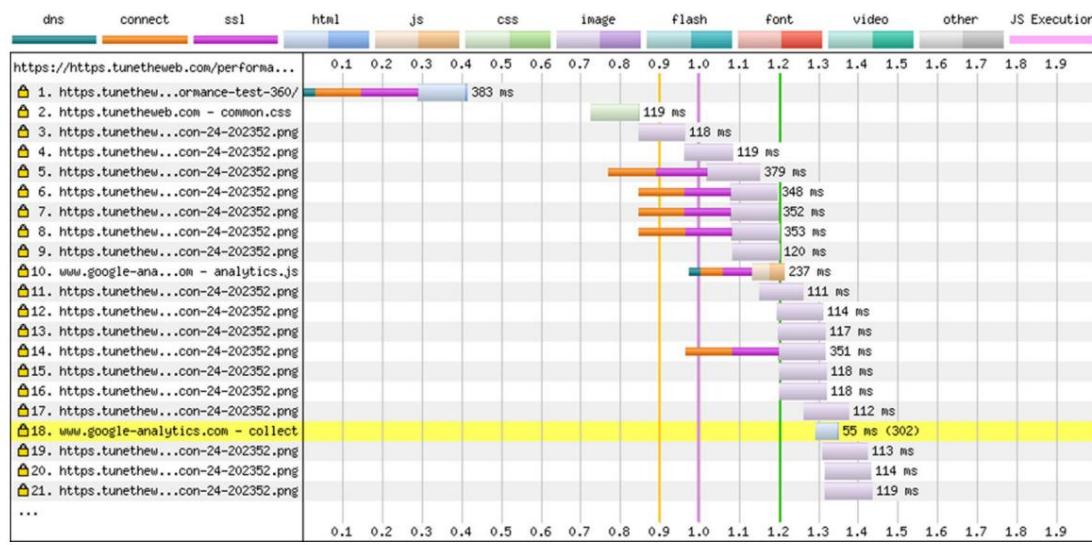


图 2.19 HTTPS 测试瀑布图。由于 302 响应,请忽略第 18 行的突出显示。

HTTP/2 对 Web 性能意味着什么

61

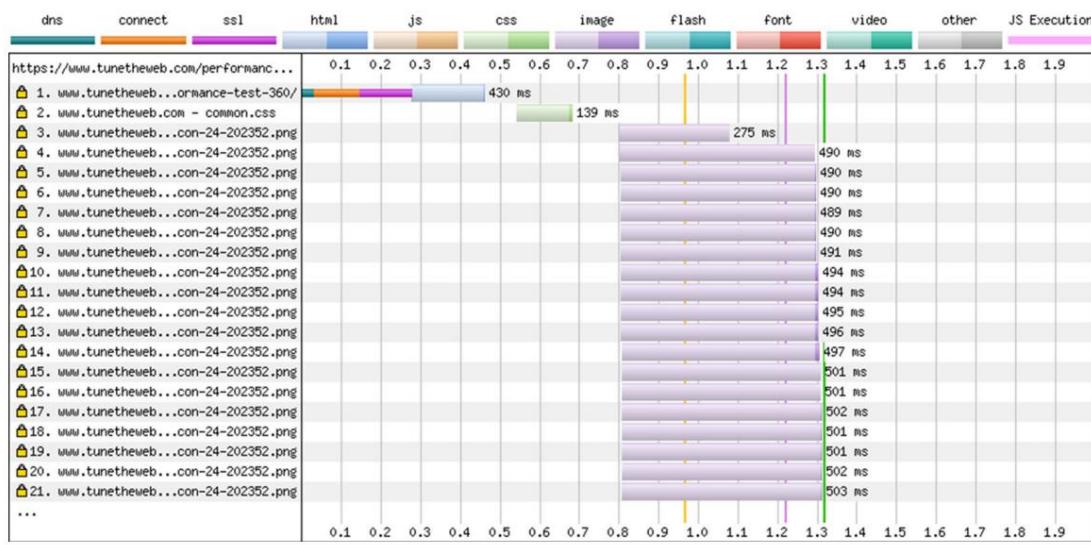


图2.20 HTTP/2测试瀑布图

尚未达到其最大连接限制。图 2.20 显示了一个更高的同时请求限制,因此在开始时请求了更多的图像,并且 Google Analytics 请求没有显示在前 21 个请求中,而是显示在这个瀑布图的更下方。

精明的读者可能已经注意到,在 HTTP/2 下下载图像需要更长的时间:大约 490 毫秒,而在 HTTP/1.1 下大约需要 115 毫秒(忽略前六个,如果包括连接设置时间,这需要更长的时间)。由于测量资产的方式不同,在 HTTP/2 下下载资产似乎需要更长的时间。瀑布图通常衡量从发送请求到收到响应的时间,可能不会衡量排队时间。以请求 16 为例,在 HTTP/1 下,该资源在大约 1.2 秒标记时被请求,并在 118 毫秒后收到,大约为 1.318 毫秒。然而,浏览器在处理 HTML 并在 0.75 秒发出第一个请求后知道它需要图像,这正是 HTTP/2 示例请求它的时间(不是巧合!)。因此,0.45 秒的延迟并没有准确反映在 HTTP/1 瀑布图中,可以说,时钟应该从 0.75 秒标记开始。如 2.4.2 节所述,Chrome 的瀑布图包括等待时间;因此,它显示了真实的整体下载时间,比 HTTP/2 更长。

然而,由于带宽、客户端或服务器限制,请求在 HTTP/2 下可能需要更长的时间。在 HTTP/1 下使用多个连接的需要创建了同时六个请求的自然排队机制。HTTP/2 使用与流的单一连接,理论上消除了该限制,尽管实现可以自由添加它们自己的限制。例如,我用来托管此页面的 Apache 默认限制每个连接 100 个并发请求。发送许多

同时请求会导致请求共享可用资源并花费更长的时间来下载。图 2.20 中的图像逐渐变长（从请求行 4 中的 282 毫秒到请求行 25 中的 301 毫秒）。图 2.21 在第 88-120 行显示了相同的结果。您可以看到图像请求最多需要 720 毫秒（是 HTTP/1 下的六倍）。此外，当达到 100 个请求的限制时，会出现暂停，直到第一个请求被下载，然后请求剩余的资源。

由于 HTTP/1 连接限制，这种效果与瀑布效应相同，但由于限制大大增加，发生的次数越来越少。还要注意，在此暂停期间，Google Analytics 被请求为请求 104。类似的事情发生在图 2.19 中 HTTP/1.1 的暂停期间，但它发生得更早，在请求 10 处。

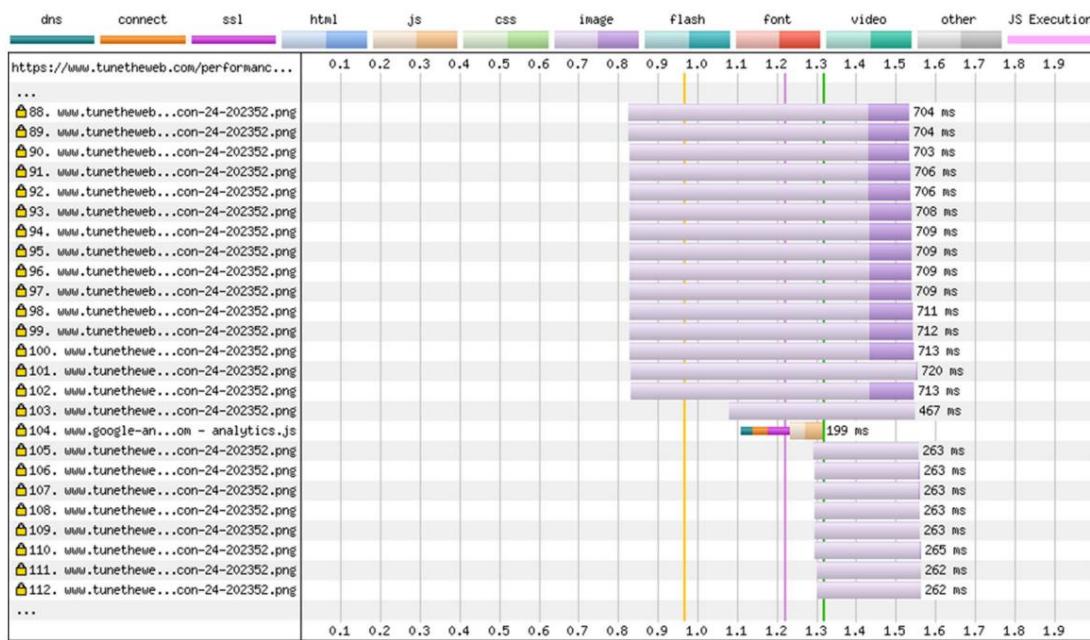


图 2.21 HTTP/2 下的延迟和瀑布

当您查看 HTTP/2 下的瀑布图时，很容易忽略的重要一点是它们本质上衡量的是不同的东西。相反，您应该查看整体时间，在这里，HTTP/2 显然在整体页面加载时间中获胜。

2.6.2 设定对 HTTP/2 性能提升的期望

2.6.1 节中的示例显示了 HTTP/2 可以给网站带来的巨大收益；83% 的性能提升非常可观。然而，这个例子是不现实的，大多数网站都很难看到接近这个结果的任何东西。这个例子说明了 HTTP/2 性能最佳的条件（这也是为什么我更喜欢在本书的例子中使用真实的、知名的网站的另一个原因）。一些

如果网站有其他性能问题,则切换到 HTTP/2 可能看到任何性能改进,这意味着 HTTP/1 的缺陷不是什么大问题。

HTTP/2 对某些现有网站影响不大的原因有两个。第一个原因是网站可能会使用 2.2 节中讨论的变通方法进行调整,以至于由于 HTTP/1 中固有的问题,它们几乎看不到缓慢。但即使是经过良好调整的站点仍然会因这些技术而遭受一些性能缺陷,更不用说使用和维护这些性能技术所需的大量工作了。从理论上讲,只要服务器支持 HTTP/2,HTTP/2 允许每个网站甚至比域分片、串联的网站更好地使用 sprite 和内联 CSS、JavaScript 和图像,只要服务器支持 HTTP/2 就可以零努力!

HTTP/2 可能无法改善网站的另一个原因是其他性能问题远远超过 HTTP/1 造成的问题。许多网站都有大量打印质量的图像,需要很长时间才能下载。其他网站加载了过多的 JavaScript,这需要时间来下载 (HTTP/2 可能有帮助) 和处理 (HTTP/2 无济于事)。即使在加载后速度缓慢或出现卡顿 (当浏览器努力跟上用户在网站上滚动时) 的网站也无法通过 HTTP/2 得到改善,它只关注性能的网络方面。其他主要是边缘情况也会使 HTTP/2 在某些高丢包情况下变慢,我将在第 9 章中讨论。

话虽如此,我坚信 HTTP/2 将带来性能更高的网站,并减少使用网站所有者迄今为止不得不使用的一些复杂变通方法的需要。与此同时,HTTP/2 的拥护者必须对 HTTP/2 可以解决什么问题和不能解决什么问题设定期望;否则,人们只会在将网站迁移到 HTTP/2 而不会立即看到巨大的性能提升时感到失望。在撰写本文时,我们可能正处于膨胀期望的顶峰 (对于那些熟悉 Gartner 炒作周期的人来说),²⁵ 在现实出现之前,人们普遍期望新技术将解决所有问题。网站需要了解他们自己的性能问题,而 HTTP/1 瓶颈只是这些性能问题的一部分。然而,根据我的经验,典型的网站从迁移到 HTTP/2 后会看到很好的改进,并且 HTTP/2 比 HTTP/1 慢的情况极为罕见 (尽管并非不可能)。

一个示例是带宽受限的网站 (例如具有许多打印质量图像的网站),如果自然排序 (由 HTTP/1.1 中的有限连接数强制执行) 导致关键资源,则在 HTTP/2 下可能会变慢下载速度更快。一家平面设计公司发布了一个有趣的示例,²⁶ 但如第 7 章所述,即使是这个示例也可以通过正确的调整在 HTTP/2 下变得更快。

²⁵ <https://www.gartner.com/technology/research/methodologies/hype-cycle.jsp>

²⁶ <https://99designs.ie/tech-blog/blog/2016/07/14/real-world-http-2-400gb-图像每天/>

回到现实世界的例子,我拿了一份亚马逊网站的副本,修改了所有引用以使其成为本地引用,通过 HTTP/1 和 HTTP/2 (均通过 HTTPS)加载副本,并测量了不同的加载时间典型结果如表2.2所示。

表 2.2 HTTP/2 可能给亚马逊带来的改进

协议	加载时间	第一个字节	开始渲染	视觉上完整	速度指数
HTTP/1	2.616	0.409秒	1.492s	2.900s	1692
HTTP/2	2.337	0.421秒	1.275s	2.600s	1317
差异 11%		-3%	15%	10%	22%

这张表介绍了网络性能圈中常见的几个术语:

加载时间是页面发送 onload 事件所花费的时间 通常,

在加载所有 CSS 和阻塞 JavaScript 之后。

第一个字节是从网站获得第一个响应所需的时间。通常,

此响应是第一个真正的响应,忽略任何重定向。

Start render 是页面开始绘制的时候。该指标是一个关键性能指标,因为如果网站不提供正在加载网站的视觉更新,用户可能会放弃该网站。

视觉上完整是指页面停止变化时,通常在加载时间很久之后,如果异步 JavaScript 在初始加载时间后仍在更改页面。速度指数是一种 WebPagetest 计算,表示加载网页每个部分的平均时间,以毫秒为单位。²⁷

这些指标中的大多数都显示了 HTTP/2 的良好改进。第一字节时间略有恶化,但重复测试表明某些测试的情况恰恰相反,因此该结果看起来在误差范围内。

然而,我承认这些改进有些人为,因为我没有像亚马逊那样完全实施该网站。我只使用了一个域(因此没有发生域分片)并将每个资产保存为静态文件而不是动态生成的内容,就像亚马逊所做的那样,这可能会受到其他延迟的影响。尽管如此,这些限制出现在测试的 HTTP/1 和 HTTP/2 版本中,因此在这些限制内,您可以看到 HTTP/2 的明显改进。

比较图 2.22 和 2.23 中两个负载之间的瀑布图显示了 HTTP/2 下的预期改进:没有额外的连接,并且开始时需要很多资源时阶梯式瀑布负载更少。

²⁷ <https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index>

HTTP/2 对 Web 性能意味着什么

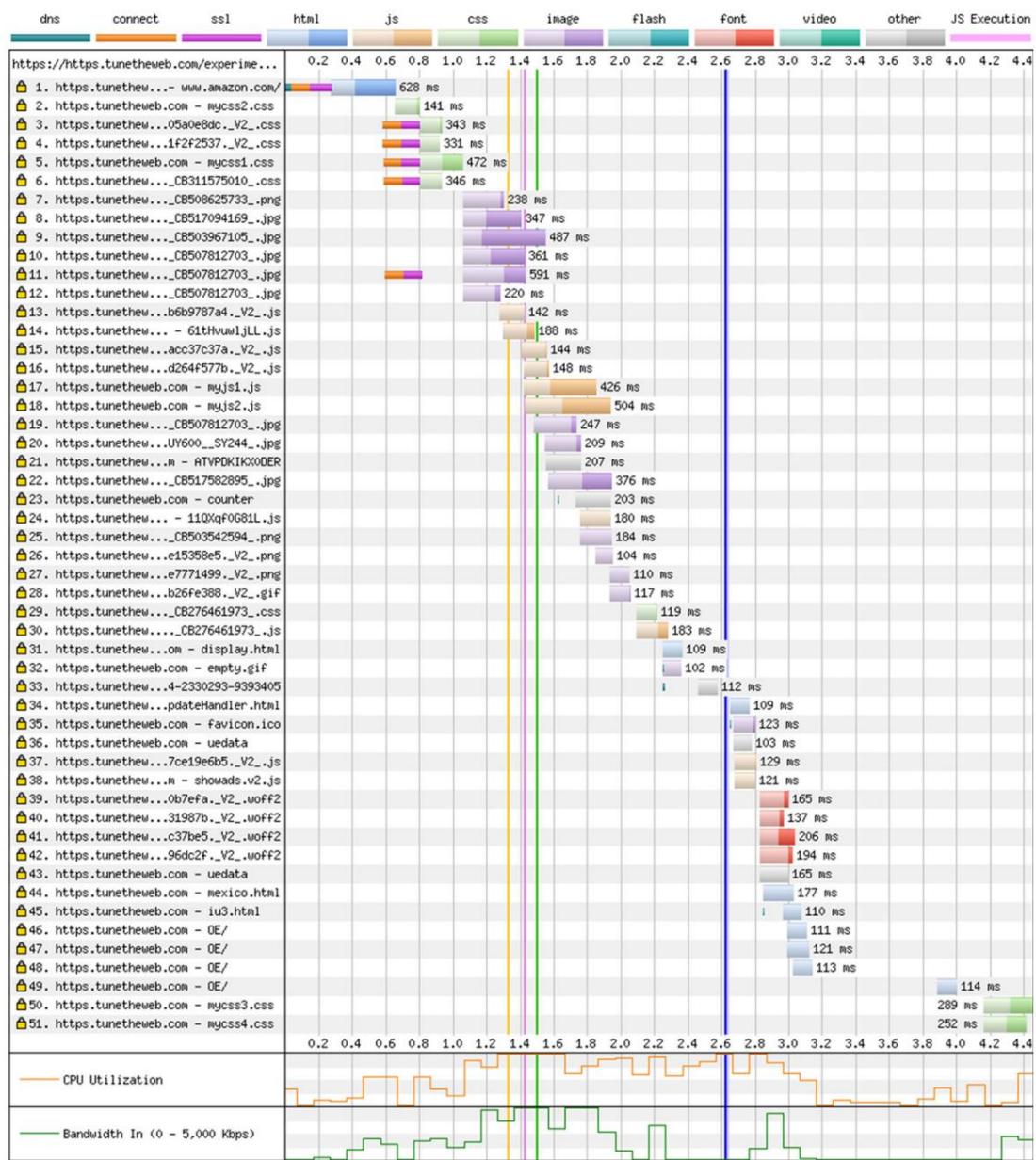


图 2.22 通过 HTTP/1 加载亚马逊主页的副本

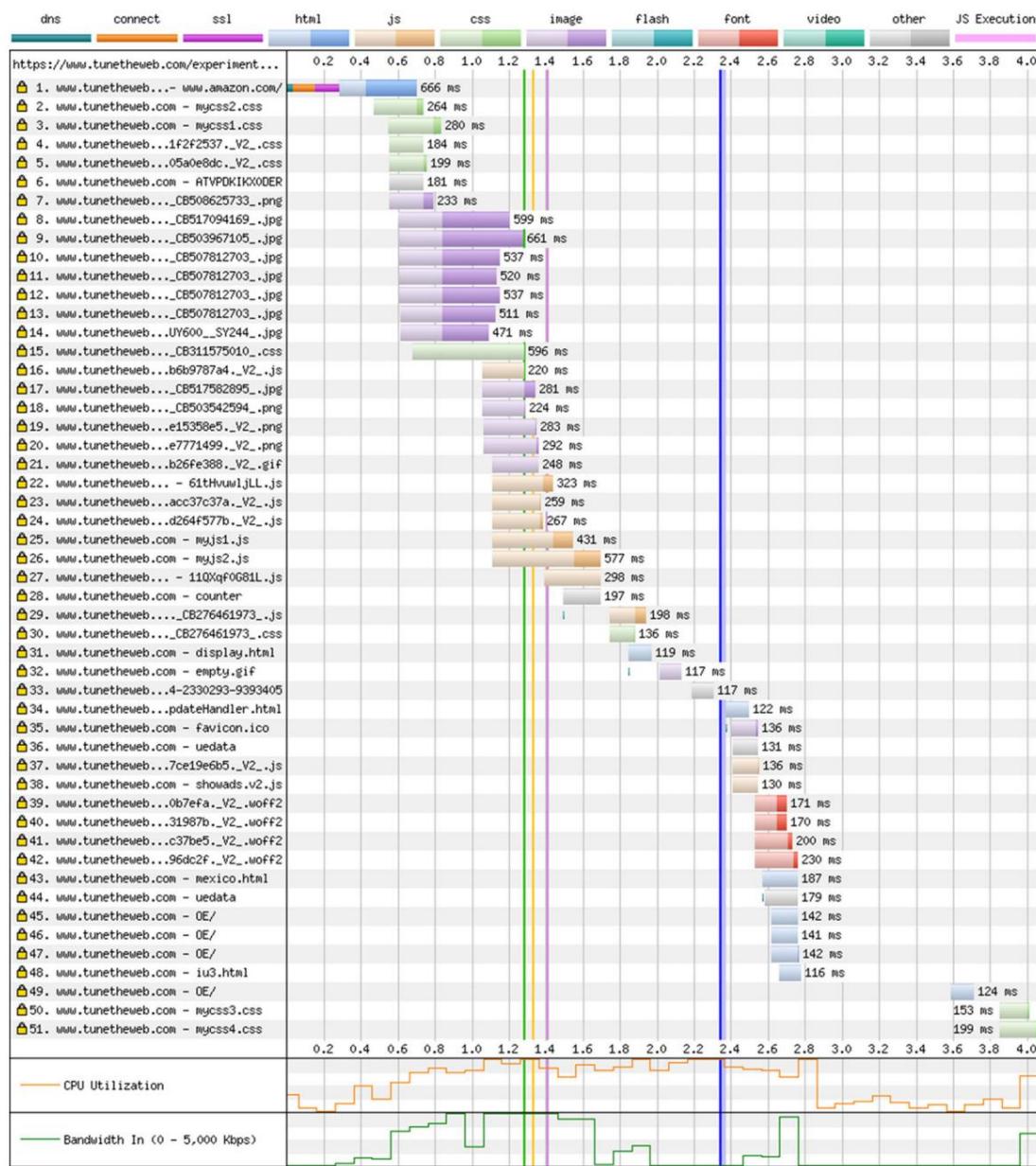


图 2.23 通过 HTTP/2 加载 amazon.com 主页的副本

网站上两种请求类型之间没有更改代码;这个结果只是 HTTP/2 带来的改进。由于网络技术的依赖性,在 HTTP/2 下加载站点仍然存在瀑布方面:例如,网页加载 CSS,它加载图像。但是设置连接和队列所花费的时间更少,因此 HTTP 约束导致的瀑布效应消失了。

这些数字可能看起来很小,但 22% 的改进是一个巨大的收获,特别是考虑到这种改进不需要对 Web 服务器本身进行任何更改。

真正针对 HTTP/2 优化并使用 HTTP/2 中可用的一些新功能(我将在本书后面介绍)的网站应该会看到更大的改进。目前,我们有 20 年在 HTTP/1 下优化网站的经验,但几乎没有针对 HTTP/2 优化的经验。

我使用 Amazon 作为一个知名网站的例子,该网站(在我撰写本章时)尚未迁移到 HTTP/2,并且针对 HTTP/1 进行了高度(尽管不完美!)优化。需要明确的是,我并不是说亚马逊编码不当或性能不佳;我正在展示 HTTP/2 可能立即给网站带来的性能改进,也许更重要的是,通过不必执行 HTTP/1.1 变通办法来获得出色的性能可以节省的工作量。

自从我最初编写本章以来,Amazon 已转向 HTTP/2,它显示了一些类似的结果。然而,重点是将亚马逊视为现实世界中复杂网站的示例,该网站已经实施了一些 HTTP/1 性能优化,但仍可以通过 HTTP/2 显着改进。

2.6.3 HTTP/1.1 作为潜在反模式的性能解决方法

因为 HTTP/2 修复了 HTTP/1.1 中的性能问题,理论上应该不需要再部署本章讨论的性能变通方案。事实上,许多人认为这些变通办法正在成为 HTTP/2 世界中的反模式,因为它们可能会阻止您获得 HTTP/2 的全部好处。例如,如果网站所有者使用域名分片并因此强制建立多个连接(尽管我在第 6 章中讨论了旨在解决此问题的连接合并),则加载网站的单个 TCP 连接的好处就会被抵消。默认情况下,HTTP/2 使创建高性能网站变得更加简单。

然而,现实从来没有那么简单,正如我在后续章节(特别是第 6 章)中所说的那样,在 HTTP/2 得到更多应用之前完全放弃这些技术可能还为时过早。在客户端,一些用户会尽管有强大的浏览器支持,但仍在使用 HTTP/1.1。他们可能正在使用较旧的浏览器或通过尚不支持 HTTP/2 的代理(包括防病毒扫描程序和公司代理)进行连接。

此外,在客户端和服务器端,实现仍在不断变化,同时人们也在学习如何最好地使用这个新协议。在 HTTP/1.1 推出后的 20 年里,一个蓬勃发展的 Web 性能优化行业成长起来,教会了开发人员

如何最好地针对 HTTP 协议优化他们的网站。尽管我希望 HTTP/2 不需要像 HTTP/1.1 那样需要那么多的 Web 优化，并且它应该可以毫不费力地提供 HTTP/1.1 下的那些优化所带来的大部分相同性能优势，但开发人员仍在习惯这个新的协议，毫无疑问，一些最佳实践和技术需要学习。

到目前为止，您可能急于启动并运行 HTTP/2。在第 3 章中，我将向您展示如何做到这一点。我稍后会回到性能优化，以展示您如何衡量改进并最好地利用 HTTP/2 来发挥您的优势。本章让您领略了 HTTP/2 可以为网络带来什么，我希望它能让您渴望了解如何在您的网站上部署它。

总结 HTTP/

1.1 有一些基本的性能问题，特别是在获取多个资源时。这些性能问题（多连接、分片、spriting 等）存在解决方法，但它们有其自身的缺点。在可以生成的瀑布图中很容易看到性能问题

通过 WebPagetest 等工具。

SPDY 旨在解决这些性能问题。HTTP/2 是 SPDY 的标准化版本。并不是所有的性能问题都可以通过 HTTP/2 来解决。

升级到 HTTP/2

3

本章涵盖浏览器和服务器对
HTTP/2 的支持为您的应用程序启用 HTTP/2 的不同
选项
网站
反向代理和 CDN 及其影响
HTTP/2
解决未使用 HTTP/2 的问题

在前两章中,我介绍了 HTTP 并展示了它在当今网络上的适用范围;然后我解释了为什么 HTTP/2 是一个必要的升级,对于大多数网站来说应该比 HTTP/1 更快。现在是让 HTTP/2 在您的网站上运行的时候了,这样您就可以看到它会给您带来多大的好处。

3.1 HTTP/2 支持 HTTP/2 于 2015 年

5 月被正式批准为互联网标准,因此它仍然是一项相对较新的技术。与所有新技术一样,实施者必须决定何时是采用的最佳时机。实施得太早,该技术将被认为是前沿技术并且实施起来有风险,因为该技术可能会发生很大变化,如果事实证明它是可行的,甚至可能会被放弃

不成功。此外，使用该技术的能力将受到不支持它的其他各方的阻碍，这意味着成为先行者之一可能获益甚微。另一方面，先行者证明了这项技术并为其成为主流铺平了道路。

幸运的是，在大多数情况下，HTTP/2 不适合通常的技术周期，因为它在现实生活中已经在其早期的非标准化化身 SPDY 中得到了证明（如第 2 章所述）。根据 w3techs.com¹，在撰写本文时，超过 30% 的网站已经使用 HTTP/2。到您阅读本书时，这个数字可能会进一步增加。HTTP/2 已经过验证并已在许多站点上使用。您是否可以在您的网站上使用新的网络技术归结为三个考虑因素：

Web 浏览器是否支持该技术？您的基础设施是否支持它？如果技术不受支持，是否可以使用可靠的后备方案？

总的来说，HTTP/2 在所有类别中都表现出色。它在几乎所有浏览器和服务器软件中都有很好的支持，如果不支持 HTTP/2，则可以无缝回退到 HTTP/1.1。然而，一些微妙之处和细微差别使这种看似强大的支持变得不那么清晰。

3.1.1 浏览器端的 HTTP/2 支持

浏览器端的 HTTP/2 支持很强。几乎每个现代浏览器都支持它，如 HTTP/2 的 caniuse.com 页面所示（图 3.1）。

Android 是最后一个为其原生浏览器添加对西方世界支持的主要平台，但 UC 浏览器（在中国、印度、印度尼西亚和其他亚洲国家流行）在撰写本文时仍不支持 HTTP/2。Opera Mini 浏览器呈现页面服务器端，因此页面由 Opera 的服务器提供，在本次讨论中可以忽略大部分。

切换到 Usage Relative 视图，如图 3.2 所示，会根据该版本的用户百分比更改每个框的大小。此图说明了 UC 浏览器（撰写本文时为 11.8 版）的强大使用，这是目前 HTTP/2 上的主要支持者。

支持在浏览器中还不普遍，但很强大，在撰写本文时占全球浏览器使用率的 83.21%，因此它只会及时改进。此外，如果您主要为一个国家/地区的用户提供服务，caniuse.com 允许您查看每个国家/地区的统计数据，以便为您提供更准确的用户群统计数据（可能不会经常使用 UC 浏览器）。然而，这些统计数据有一些重要的微妙之处。

¹ <https://w3techs.com/technologies/details/ce-http2/all/all>
² <https://caniuse.com/#feat=http2>

HTTP/2 支持

71

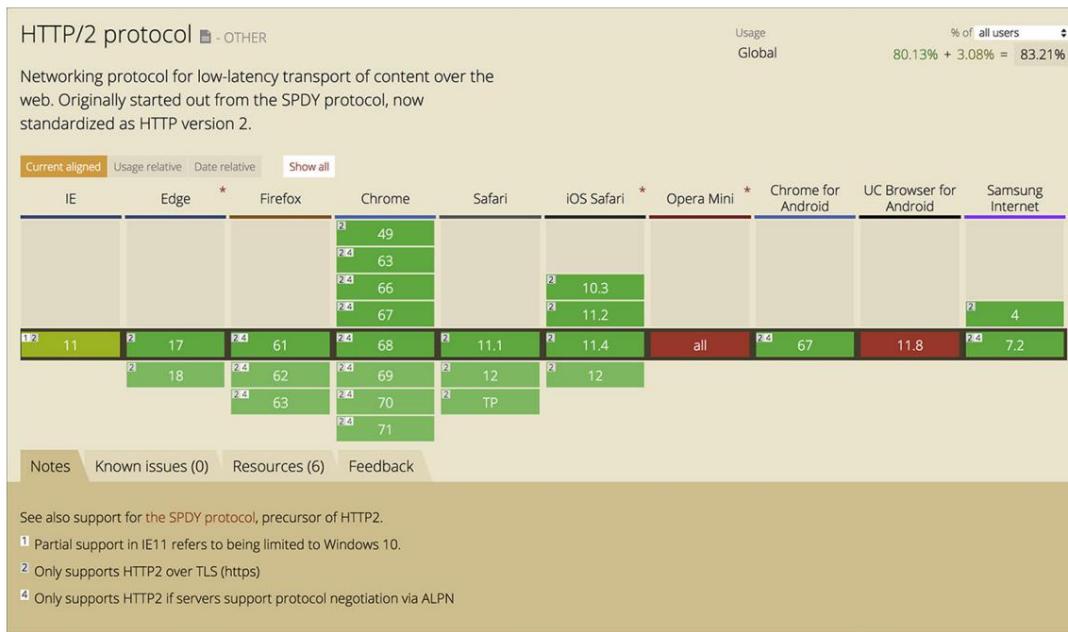


图 3.1 HTTP/2 的 caniuse.com 页面



图 3.2 caniuse.com HTTP/2 使用相关页面

浏览器的HTTP/2和HTTPS上图中每个

支持 HTTP/2 的浏览器的小 2 在底部解释为“Only supports HTTP2 over TLS (https)”,因此不使用 HTTPS 的网站无法访问受益于 HTTP/2。SPDY 也有类似的限制,这个限制在 HTTP/2 标准化时进行了详细讨论,许多方面都在推动将 HTTPS 作为规范的一部分。最终,这个要求被排除在 HTTP/2 的正式规范之外,但所有浏览器供应商都表示他们将仅支持 HTTP/2 over HTTPS,使其成为事实上的标准。要求 HTTPS 无疑会让 HTTP 网站的所有者感到不安,但这种要求有两个很好的理由。

第一个原因纯粹是实用的。仅通过 HTTPS 使用 HTTP/2 意味着不太可能出现兼容性问题。互联网上许多支持 HTTP 的基础设施在升级之前不知道如何处理 HTTP/2 消息。通过将消息包装在 HTTPS 中,您可以隐藏 HTTP 消息本身,从而防止出现兼容性问题。(HTTPS 消息只能由接收者读取,尽管我将在下一节讨论拦截代理的特殊情况。)

第二个原因本质上更具意识形态。许多浏览器供应商(以及其他人,包括我在内)坚信应该摆脱未加密的 HTTP,并进一步认为所有网站都应该转向 HTTPS。因此,较新的功能通常仅限于 HTTPS,以此作为鼓励这一举措的一种形式。

HTTPS 增加了与网站通信的安全性、隐私性和完整性。

这些功能不再只对需要保护支付细节的电子商务网站很重要,而是对所有网站都很重要。³搜索词和您正在查看的页面包含可能敏感的个人数据。要求您提供电子邮件地址以注册兴趣的注册表单正在收集私人信息,因此应该对其进行保护。尽管拦截和更改数据对于博客来说听起来不太可能,例如,移动运营商和飞机 Wi-Fi 运营商等数据提供商定期插入广告,如果用户通过它们浏览互联网并且网站上未使用 HTTPS 来防止这种情况发生情况。更多恶意方可以注入更多危险内容,例如加密货币挖掘 JavaScript 代码或恶意软件。

网站所有者将更难避免使用 HTTPS,而 HTTP/2 要求它是转向它的另一个原因。

即使您的站点上有 HTTPS,您仍然可能会遇到问题。HTTP/2 需要强大的 HTTPS。一些浏览器(Chrome, Firefox 和 Opera)的小 4 表示“如果服务器支持通过 ALPN 进行协议协商,则仅支持 HTTP2。”我在第 3.1.2 节中讨论了这个主题,但现在请注意,只有更新的 HTTPS 服务器支持应用层协议协商 (ALPN),并且如果 ALPN 不可用,某些浏览器将不会使用 HTTP/2。此外,许多浏览器

³ <https://tools.ietf.org/html/rfc7258>

在他们使用 HTTP/2.4 之前需要某些更新、更安全的密码套件同样，我将在 3.1.2 节中讨论这个主题。

拦截代理

为了能够使用 HTTP/2，浏览器和服务器都必须支持 HTTP/2。但是，如果用户使用有效地将 HTTP 连接一分为二的代理，那么如果代理不支持 HTTP/2，则可能会阻止使用 HTTP/2。

在许多企业环境中，使用代理并限制直接互联网访问是很常见的。这允许扫描威胁，并阻止访问某些站点（例如个人电子邮件帐户）。同样，对于家庭用户，许多防病毒产品会创建一个代理，网络流量通过该代理进行扫描。

对于 HTTPS 流量（例如 HTTP/2 对所有 Web 浏览器的要求），这种情况是个问题，因为这些代理无法读取此加密流量。因此，当您使用需要读取 HTTPS 流量的代理时，您的浏览器被配置为创建一个到代理的 HTTPS 连接，而代理创建一个单独的 HTTPS 连接到真实网站。因此，您的 Web 浏览器仅与代理建立 HTTPS 连接，并向浏览器发送伪造的 HTTPS 证书，假装是真实站点。通常，这种情况在浏览器中是一个很大的警告信号，因为 HTTPS 的一部分是验证 HTTPS 证书颁发者的真实性。然而，安装这些代理涉及到将代理软件设置为该计算机上公认的证书颁发者，因此 Web 浏览器将接受这些假证书。

将流量分成两部分允许代理读取流量，但不幸的是，您的浏览器不再直接连接到网站，因此您能否使用 HTTP/2 取决于代理是否支持 HTTP/2。如果代理不支持 HTTP/2，它会有效地将您的连接降级为 HTTP/1.1。除了不能从 HTTP/2 中获益之外，您可能对当浏览器和服务器似乎都支持 HTTP/2 时为什么会发生降级感到困惑（请参阅第 3.3 节）。

很多安全行业的人认为拦截代理造成的问题比他们解决的要多，因为浏览器制造商通常在确保良好的 HTTPS 连接方面非常强大和主动，将这种连接一分为二意味着浏览器无法再验证最终连接。无论如何，在尝试理解 HTTP/2 支持时需要考虑使用代理。研究表明，有 4% 到 9% 的互联网流量以这种方式被拦截，其中 58% 的流量被防病毒软件拦截，35% 被公司代理拦截。⁵最简单的方法来查看您的计算机是否正在使用拦截代理是查看 HTTPS 证书，看它是由真正的证书颁发机构颁发的（有很多，所以这可能不是很明显）或本地软件。图 3.3 显示了 Avast 病毒扫描程序创建证书时 Internet Explorer 中的差异。

⁵ <https://tools.ietf.org/html/rfc7540#appendix-A>

⁶ <https://jhalderm.com/pub/papers/interception-ndss17.pdf>



图 3.3 查看与 Google 直接连接的 HTTPS 证书以及被防病毒产品拦截的证书

从积极的一面来看,拦截代理通常用于家庭或公司环境,在这些环境中连接通常很快,而 HTTP/2 的好处不大。

以这种方式拦截移动流量的情况要少得多,低延迟网络(例如移动网络)是 HTTP/2 改进的一些主要受益者。

浏览器对 HTTP/2 的支持总结

如本节所示,浏览器对 HTTP/2 的支持通常很强,自动更新的常青浏览器的出现(请参阅下面的边栏)意味着 HTTP/2 的推出非常顺利在浏览器端无缝。

然而,可能不使用 HTTP/2 的原因有很多,包括需要在服务器端设置强大的 HTTPS 和使用拦截代理。

对 HTTPS 的要求,尤其是 HTTPS 类型的严格性质,是启用 HTTP/2 的额外复杂因素,并且确实会引起混淆。

然而,正如我在 3.1.2 节中讨论的那样,这种复杂性主要需要正确设置服务器。网络正在转向 HTTPS,对未加密的 HTTP 站点的处罚将继续增加,可见的警告越来越多,可用的功能越来越少。在撰写本文时,超过 75% 的互联网流量是通过 HTTPS 提供的。⁶虽然这个数字无疑因几个大型网站的高使用率而有所偏差,但现实情况是,如果您运行一个不使用 HTTPS 的网站,您应该计划尽快迁移到它。

长青浏览器 Chrome 和 Firefox 等浏览器在后台静默更新而不提示用户,被称为长青浏览器。因此,这些浏览器的用户可能会运行这些浏览器的最新版本,这些浏览器支持 HTTP/2。

这种情况并不总是常见的。Web 开发历史充满了沮丧的开发人员,如果某些用户仍在使用 Internet Explorer 5 或类似软件,他们不得不检测浏览器版本并实施黑客攻击。

⁶ <https://letsencrypt.org/stats/>

然而,情况并不像看起来那么乐观。尽管桌面上的 Chrome、Firefox 和 Opera 在保持常青方面做得很好,但其他浏览器和平台的自动升级并没有那么无缝。Safari 升级通常与底层操作系统相关联,尤其是在移动设备上,虽然最新版本 iOS 的采用速度总是很快,但主要升级每年才发布,HTTP/2 等功能通常是一部分的重大升级。Android 从 Android 5 (Lollipop) 迁移到了常青树 Webview Chromium,但通常仍需要用户选择通过 Play Store 安装升级。Edge 是另一种明显的常青浏览器,但由于与操作系统升级相关,因此名不副实,尽管微软最近承诺改进这一点。

最后,有些人关闭了自动升级。公司环境喜欢控制何时推出更新,所以他们中的许多人关闭了自动更新,然后没有腾出足够的时间来手动推出升级。

-
- ⁷ <https://www.scirra.com/blog/173/just-how-evergreen-is-microsoft-edge>
 - ⁸ <https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/>

3.1.2 服务器的 HTTP/2 支持

服务器对 HTTP/2 的支持一直落后于浏览器的支持,但现在几乎所有的服务器都增加了支持。HTTP/2 GitHub 站点有一个页面跟踪客户端和服务器端的 HTTP/2 实现,⁷可以快速检查哪些服务器支持 HTTP/2。⁸根据 Netcraft 8 ,占活跃互联网网站 80% 以上的四个最流行的 Web 服务器使用 Apache、nginx、Google 或 Microsoft IIS,它们都支持 HTTP/2。

服务器端的问题不在于最新版本的服务器软件是否支持 HTTP/2,而在于网站运行的版本是否支持。大多数实现不会自动更新或像浏览器那样容易更新,因此服务器通常运行添加 HTTP/2 支持之前的旧版本。

通常,该版本与操作系统相关(例如,仅在 IIS 10.0 和 Windows Server 2016 中添加了 Microsoft IIS 支持)或该操作系统的包管理器(例如Red Hat/CentOS/Fedora的yum,它没有在撰写本文时,我没有安装支持 HTTP/2 的 Apache 或 nginx 版本)。

尽管通常可以升级服务器软件的版本,但过程可能很复杂。在基于 Linux 的系统上,升级可能涉及下载源代码并编译它,这需要一定程度的技能和理解,以及持续致力于使软件保持最新或至少应用安全补丁。让操作系统或包管理器处理这个过程的好处是,保持对安全问题的关注成为定期(甚至自动)运行更新的简单问题。通过走出那个过程,

⁷ <https://github.com/http2/http2-spec/wiki/Implementations>

⁸ <https://news.netcraft.com/archives/category/web-server-survey/>

您承担更多工作或引入更多风险,或两者兼而有之。根据您的操作系统,可以使用提供软件的 HTTP/2 版本的第三方存储库 (repo),但是您信任的是第三方而不是官方存储库。

HTTPS库和支持

服务器端 (尤其是 Linux 端)的最大问题之一是我在本章前面提到的严格的 HTTPS 要求。大多数 Web 服务器将 HTTPS 所需的 SSL/TLS 复杂性委托给一个单独的库 通常是 OpenSSL,尽管存在变体,包括 LibreSSL 和 BoringSSL。这个加密库通常是操作系统的一部分,虽然升级 Web 服务器可能很棘手,但升级 SSL/TLS 库通常更加困难,因为它可能会影响服务器上的所有其他软件。

在 3.1.1 节中,您看到 Chrome 和 Opera 仅通过 SSL/TLS 的 ALPN 扩展支持 HTTP/2,而不是旧的 NPN (下一个协议协商)扩展。ALPN 和之前的 NPN 一样,允许 Web 服务器声明服务器支持哪些应用协议作为 HTTPS 协商的一部分;我在第 4 章中更详细地讨论了这个主题。问题是 ALPN 支持仅包含在最新版本的 OpenSSL (1.0.2 和更高版本)中,并且在许多平台上作为标准构建的一部分不可用。RedHat 和 CentOS 仅在 2017 年 8 月和 2017 年 9 月才分别添加了对 OpenSSL 1.0.2 的支持,但打包版本的 Web 服务器软件 (如 Apache)通常是针对不支持 ALPN 的旧 1.0.1 版本编译的,因此 Chrome 和 Opera 不允许使用 HTTP/2。同样,Ubuntu 在版本 16 中添加了 OpenSSL 支持,而不是在广泛使用的版本 14 中,而 Debian 直到版本 9 (Stretch) 才添加支持 ALPN 的 OpenSSL。即使有一个现代的 OpenSSL 并且 Web 服务器是针对它编译的,HTTP/2 代码也可能不包含在默认情况下,如表 3.1 中总结的那样。

表 3.1 各种 Linux 操作系统对 ALPN 的支持

操作系统和版本	默认为 ALPN 打开SSL	默认为 ALPN Apache/nginx	默认为 HTTP/2 Apache/nginx
RHEL/CentOS < 7.4	否 (1.0.1)	否	否
RHEL/CentOS 7.4 & 7.5	Y (1.0.2)	否/是	否/是
Ubuntu 14.04 长期支持版	否 (1.0.1)	否	否
Ubuntu 16.04 长期支持版	Y (1.0.2)	是	否/是
Ubuntu 18.04 长期支持版	是 (1.1.0)	是	是
Debian 7 (“喘息”)	否 (1.0.1)	否	否
Debian 8 (“杰西”)	否 (1.0.1)	否	否
Debian 9 (“延伸”)	是 (1.1.0)	是	是

正如您在表 3.1 中看到的,在常见的 Linux 发行版中,只有 Ubuntu 18.04 和 Debian 9 为 Apache 提供开箱即用的 HTTP/2 (尽管它需要在 Web 服务器配置期间打开)。对于 RHEL/CentOS,如果您想使用 HTTP/2,则需要从源代码或另一个非默认存储库安装 Apache。

对于 nginx,可以通过 nginx 存储库安装 HTTP/2,⁹ 因此只要您使用的是最新版本,通常会为 nginx 配置 HTTP/2,但它仍然取决于底层 OpenSSL 版本。

服务器支持摘要

尽管 HTTP/2 的服务器端支持在理论上与浏览器端支持一样好,但现实情况是,一段时间以来,大多数人将运行不支持 HTTP/2 的旧版本服务器软件。这些版本需要升级才能支持 HTTP/2,升级可能简单也可能复杂。随着更新版本的操作系统成为常态,这种情况将会改变,但这对 HTTP/2 的采用来说是一个挑战。好消息是升级应该在网站所有者的控制之下,他们可以在服务器端承担将软件升级到 HTTP/2 的麻烦,当他们这样做时,他们可以假设大多数客户端软件都支持它。如果服务器端的支持很强,而客户端的支持相对较弱,那么网站所有者只能等待用户升级。

如果你不想或不能升级你的网络服务器软件,其他的实现也是可能的,正如我在 3.2 节中讨论的那样。

3.1.3 不支持 HTTP/2 时的回退

另一个好消息是,当不支持 HTTP/2 时,网站仍然可以正常工作,因为它们会回退到使用 HTTP/1.1。HTTP/1.1 离被禁用还有很长的路要走 (如果它曾经被禁用的话)。从理论上讲,就可支持性而言,如果可以的话,启用 HTTP/2 并没有真正的缺点。

然而,当您想开始利用 HTTP/2 功能并更改您的网站时,情况会变得很有趣,这可能会对 HTTP/1.1 用户不利。该站点仍然可以工作,但如果它不分片、组合和内联资产,它可能会变慢。这种情况对您来说有多大问题取决于您拥有的 HTTP/1.1 流量。我会在第 6 章回到这个话题。

更难衡量的是客户端或服务器端的实施问题。

HTTP/2 还相对年轻,尽管在现实生活中被积极使用,但仍处于采用的早期阶段。毫无疑问,在可能会影响通过 HTTP/2 加载您的网站的实现中会发现错误。根据我目前的经验,这些错误通常会导致 HTTP/2 没有您预期的那么快,而不是造成任何真正的伤害。但是在为您的生产网站启用之前,您应该彻底测试任何主要升级 (例如 HTTP/2)。

⁹ <http://nginx.org/en/download.html>

3.2 为您的网站启用 HTTP/2 的方法 迁移到 HTTP/2 的最明

显方法是在您的 Web 服务器上启用 HTTP/2,但此过程可能需要升级。但是,在 Web 服务器上启用 HTTP/2 并不是启用它的唯一方法,您可能需要考虑其他选项。这些选项涉及在您的 Web 服务器前面添加一个基础设施:另一个软件或服务,例如内容分发网络 (CDN) 来处理连接的 HTTP/2 部分。哪种方法适合您取决于几个因素,包括您的 Web 服务器是否支持 HTTP/2、启用 HTTP/2 支持的难度以及您是否希望通过实施其他一些选项来使您的环境复杂化。

启用 HTTP/2 后,您可能会注意到您的流量仍在使用 HTTP/1.1,因此在 3.3 节中,我将讨论故障排除。如果您已经实施了 HTTP/2 但正在努力让 HTTP/2 在您的环境中工作,请跳至该部分。

3.2.1 Web 服务器上的 HTTP/2

在 Web 服务器上启用 HTTP/2 允许支持 HTTP/2 的客户端使用此新协议。图 3.4 显示了这个简单的设置。

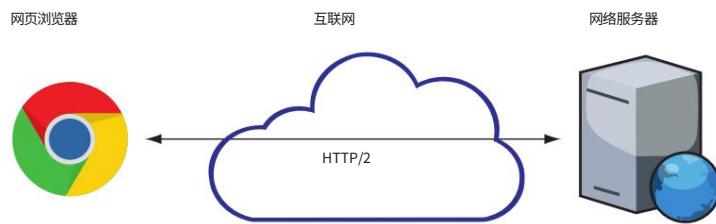


图 3.4 Web 服务器上的 HTTP/2

此选项的主要问题是您可能无法随时使用它。如第 3.1.2 节所述,您可能必须将 Web 服务器升级到新版本,这可能需要升级运行 Web 服务器的服务器的操作系统。或者您的 Web 服务器软件即使在最新版本中也不支持 HTTP/2。表 3.2 显示了一些添加了 HTTP/2 支持的常见 Web 和应用程序服务器的版本。

表 3.2 当 HTTP/2 支持被添加到流行的 Web 服务器时

网络服务器	添加了 HTTP/2 版本
HTTPD	2.4.17 (尽管在 2.4.26 之前标记为实验性)
信息系统	10.0
码头	9.3
网络	4.1

为您的网站启用 HTTP/2 的方法

表 3.2 当 HTTP/2 支持添加到流行的 Web 服务器时（续）

网络服务器	添加了 HTTP/2 版本
nginx	1.9.5
节点JS	8.4.0（虽然默认情况下直到 9.0 才启用并且在 10.10 之前仍标记为实验性）
雄猫	8.5

Linux 软件通常是通过包管理器（例如 yum 和 apt get）和一组官方软件仓库安装的，这使得安装和打补丁变得容易。其中许多环境将稳定性置于新功能之上，并且由于 HTTP/2 相对较新（至少在服务器发布方面），Web 服务器的默认版本通常不包括 HTTP/2 支持。如果您的操作系统没有提供在您首选的 Web 服务器上启用 HTTP/2 的直接方法（许多没有），并且您想在您的服务器上启用 HTTP/2 等新功能，那么您别无选择但要从其他位置安装应用程序。这个决定并非没有风险，你应该在冒险走这条路之前了解后果（见下面的侧边栏）。

从其他位置安装应用程序的风险 从其他位置安装意味着直接从另一个站点下载 Web 服务器的预打包版本、添加包管理器可以从中下载包的存储库，或者从源代码安装。

从第三方下载预构建的软件包意味着您信任该软件提供商提供基础设施的关键部分，并且以根用户身份运行（网络服务器通常这样做）。此外，这些软件包中有许多针对某个版本的 OpenSSL 静态编译，因此如果在 OpenSSL 中发现漏洞，您需要更新您的 Web 服务器以包含任何修复。许多公司对这些限制中的任何一个都感到不舒服。如果您习惯于使用第三方的预构建包，CodeDelta 是一个提供 Apache 和 nginx 预构建包的 repo，并提供了有关如何安装它们的良好说明。

另一种选择是在 Docker 等容器中运行 Web 服务器。提供了具有通用 Web 服务器合规版本的容器图像。你有同样的问题信任那个容器的提供者，但是你在它自己的容器中运行应用程序，这可能限制了对服务器其余部分的访问。由于容器软件本身就是一个话题，因此我不会在本书中进一步讨论它。

不想信任第三方的人可以从原始源代码安装软件。源代码应从信誉良好的来源（最好是原始供应商）下载，并在下载后进行验证，方法是根据签名检查下载内容或计算并检查下载内容的哈希值。

即使您是从供应商的官方网站安装的，您也是在包管理器工具之外管理该软件，因此您不会受益于安全补丁。

从其他位置安装应用程序的风险（续）包管理器使安装变得容易；您需要自己承担手动升级的责任。例如，RHEL 7 和 CentOS 7 在标准存储库中提供 Apache 2.4.6。但是，这个版本不是原来的 Apache 2.4.6；Red Hat 不断对其进行修补，以包含自该版本以来的所有相关安全更新。通过在包管理器之外运行一个版本，您将不会获得这些安全补丁，并且需要自己升级软件才能获得任何安全修复；否则，您将面临运行易受攻击的不安全软件的风险。

另一种选择是使用替代的半官方回购。许多操作系统供应商提供替代的软件集合存储库（例如 Red Hat Software Collections），或者供应商可以提供官方存储库（如 nginx 所做的）。优点是易于安装和继续易于打补丁。

最终，在服务器上安装第三方软件时，您需要决定最适合哪种方法。

→ https://codeit.guru/en_US/

附录提供了有关安装和升级一些常见 Web 服务器和平台以启用 HTTP/2 支持的说明。根据您的操作系统和 Web 服务器用户，此过程可能会非常复杂。随着时间的推移，随着默认安装升级到支持 HTTP/2 的版本，此过程将变得更加容易；那么启用 HTTP/2 支持应该涉及一个简单的配置更改，或者协议可能默认启用。然而，在接下来的几年里，许多人将努力在他们的网络服务器中启用 HTTP/2 支持

软件。

但是，您将在接下来的两节中看到其他选项可用。例如，在某些设置中，如果负载均衡器本身不支持 HTTP/2，则在您的 Web 服务器前面使用负载均衡器时，如果负载均衡器本身不支持 HTTP/2，则在您的 Web 服务器上启用 HTTP/2 可能不会向您的用户提供 HTTP/2。

如果您希望设置一个简单的 Web 服务器来试验 HTTP/2，并且可能要遵循本书中的一些示例，我建议您选择您最熟悉的 Web 服务器。如果您没有特别的偏好，Apache 是流行的 Web 服务器中功能最齐全的，因为它在许多平台上都可用，而且它支持 HTTP/2 推送和 HTTP/2 代理（我将在本书后面介绍这两者）。

3.2.2 带反向代理的 HTTP/2

实现 HTTP/2 的另一种选择是在支持 HTTP/2 的主 Web 服务器之前放置一个反向代理服务器；然后它可以翻译 HTTP/1.1 中的请求并将它们代理到您现有的 Web 服务器，如图 3.5 所示。

顾名思义，反向代理与标准拦截代理相反。标准代理将网络与外界隔离开来，并提供一个

为您的网站启用 HTTP/2 的方法

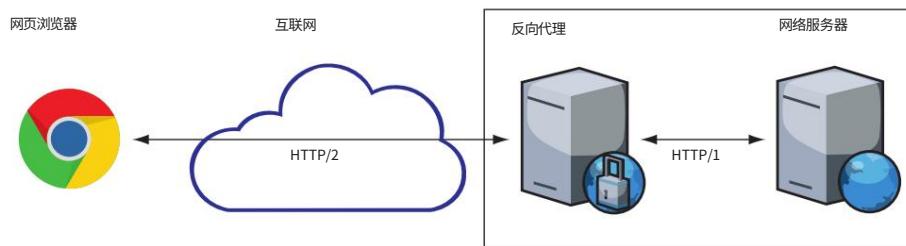


图 3.5 使用反向代理实现 HTTP/2

需要与 Internet 通信的出站流量路径。反向代理处理来自互联网的传入流量,允许访问外部世界无法直接使用的服务器。反向代理已经很普遍,主要出于以下两个原因之一使用:

作为负载均衡器

卸载 HTTPS 或 HTTP/2 等功能

负载均衡器位于至少两个 Web 服务器之前,并将流量发送到任一 Web

服务器,具体取决于它的配置方式 (实时-实时或实时-待机)。实时负载均衡器使用算法来决定如何拆分流量 (例如,基于源 IP 地址或循环法)。此设置如图 3.6 所示。

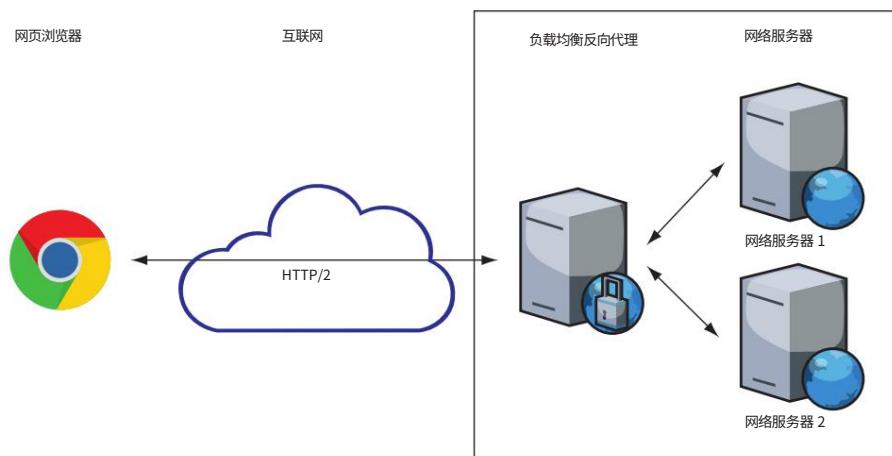


图 3.6 负载均衡反向代理

如果您已经进行了此设置,则可以在负载均衡器上启用 HTTP/2,而不必在您的 Web 服务器上启用它。事实上,正如我已经提到的,如果所有流量都将负载均衡器作为第一个联系点,那么在 Web 服务器上启用 HTTP/2 可能没有任何效果。一些负载均衡器产品 (例如

如 F5、Citrix Netscaler 和 HAProxy)已经支持 HTTP/2,而那些不支持的有望很快添加支持。

您需要一直使用 HTTP/2 吗?

当您在反向代理上实现 HTTP/2 时,HTTP/2 连接在反向代理处终止,并且从那时起,使用单独的连接(可能使用 HTTP/1.1)。此过程类似于在反向代理处终止 HTTPS,然后将 HTTP 与基础架构的其余部分通信,这是简化 HTTPS 配置的常见用例(证书仅需要在入口点设置和管理)和因为过去 HTTPS 所需的资源(尽管现在随着计算能力的提高,这些要求可以忽略不计)。

那么,您是否需要一直使用 HTTP/2,通过 HTTP/1.1 建立后端连接会损失什么?

HTTP/2 的主要好处是提高了高延迟、低带宽连接的速度,例如最终用户到您的边缘服务器(在本例中为反向代理)。从你的反向代理到你的网络基础设施的其余部分的流量可能必须通过低延迟、高带宽的网络链接短距离传输(通常到同一个数据中心,如果不是同一台机器),所以性能问题 HTTP/1.1 通常不是什么大问题。

对 HTTP/2 流量使用单个连接也没有从反向代理到真实服务器的好处,因为它们不限于浏览器设置的六个连接。甚至有人担心使用单个连接可能会导致性能问题,具体取决于此连接在反向代理和目标服务器上的实现方式。nginx 已经声明它不会为代理传递连接实现 HTTP/2,部分原因是这个原因。

因此,与 HTTPS 一样,可能无需在您的基础架构中一直使用 HTTP/2 来获得基本的 HTTP/2 支持。正如我在第 5 章中讨论的那样,即使是仅 HTTP/2 的功能(如 HTTP/2 推送)仍然可以通过此设置实现。

→ <http://mailman.nginx.org/pipermail/nginx/2015-December/049445.html>

即使不为负载均衡器使用反向代理,在后端应用程序服务器(如 Tomcat 或 Node.js)和 Web 之前使用 Apache 或 nginx 等 Web 服务器也是很常见的(并且在我看来,推荐) server to proxy 将部分(或全部)请求传递给后端服务器,如图3.7所示。

这种技术有几个优点,主要的一个是您可以卸载功能并加载到 Web 服务器,例如从 Web 服务器提供静态资产(图像、CSS 文件、JavaScript 库等)、卸载 HTTPS,以及是卸载 HTTP/2。通过减轻应用程序服务器的负载,您可以让它专注于它最擅长的事情:提供需要一些处理和可能的数据库查找来决定返回什么动态资产。

此选项还具有安全优势,因为第一个接触点是 Web 服务器,它可以防止错误请求到达更脆弱的应用程序。

为您的网站启用 HTTP/2 的方法

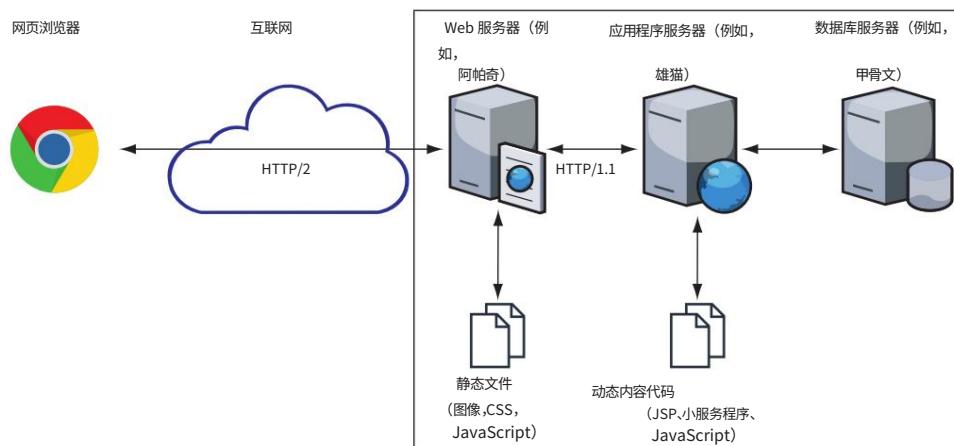


图 3.7 应用服务器/数据库服务器前的Web服务器

服务器,当然还有任何后端数据库。因此,如果您正在使用难以启用 HTTP/2 的应用程序服务器,则可能 (甚至推荐)通过在其前面放置另一个支持 HTTP/2 的 Web 服务器来获得 HTTP/2 支持。

反向代理也是测试 HTTP/2 及其对您网站影响的有效方式。只需使用不同的服务器名称 (`http2.example.com` 或 `test.example.com`) 在您的服务器附近托管一个反向代理,并通过快速本地连接使用 HTTP/1.1 将请求代理回主 Web 服务器,如图 3.8 所示。

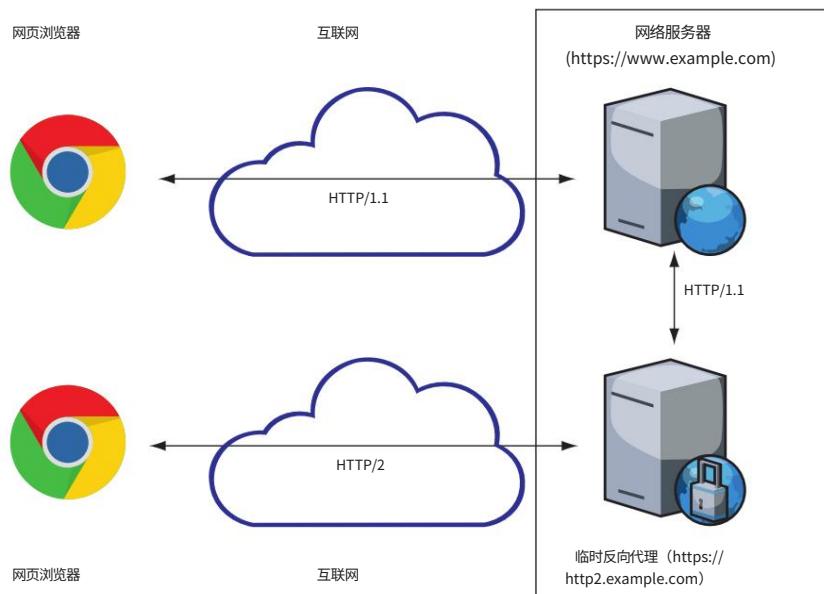


图 3.8 添加临时反向代理测试 HTTP/2

如果您的 Web 服务器已经支持 HTTP/2 但您尚未启用它，则您可能不需要单独的代理服务器。通过设置启用 HTTP/2 和单独主机名的虚拟主机，您可以同时运行 HTTP/1.1 和 HTTP/2 站点；因此，您可以先测试 HTTP/2，然后再在访问者使用的主要虚拟主机上启用它。

3.2.3 通过 CDN 的 HTTP/2

CDN 是分布在世界各地的一系列服务器，充当您网站的本地联系点。您网站的访问者通过在全球范围内拥有不同的 DNS 条目来连接到最近的 CDN 服务器。请求被路由回您的 Web 服务器（称为源服务器），并且可能会在 CDN 中缓存一个副本，以便在下次收到相同请求时更快地提供服务。大多数 CDN 已经支持 HTTP/2，因此您可以切换在 HTTP/2 上使用 CDN 并将源服务器留在 HTTP/1.1 上。此方法类似于反向代理方法，只是 CDN 有许多反向代理并为您管理此基础设施。此设置如图 3.9 所示。

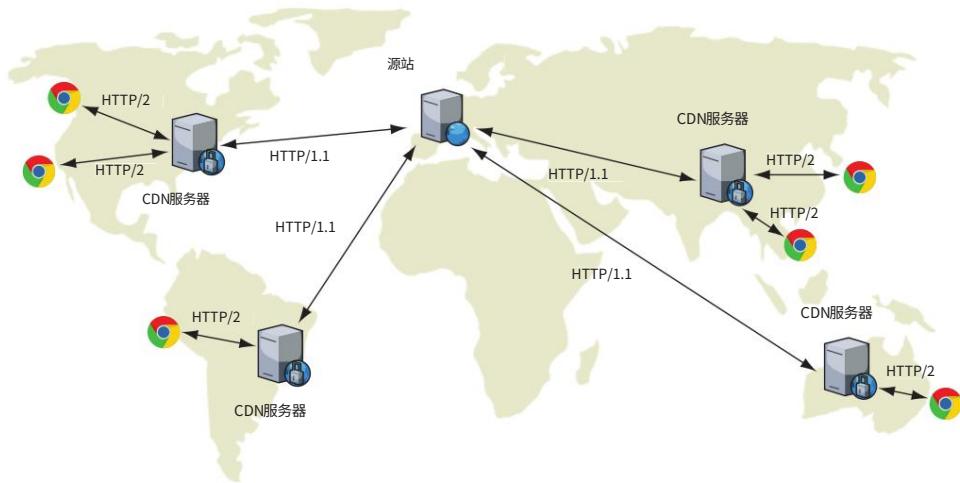


图 3.9 通过 CDN 启用 HTTP/2

尽管在您的设置中添加了额外的基础设施，但使用 CDN 可能比直接连接要快得多，因为本地服务器可以处理客户端的一些连接设置请求（例如初始 TCP 连接和 HTTPS 协商）。让更靠近用户的服务器处理这些客户端设置请求的好处超过了请求中涉及额外服务器跳转的负面影响。此外，响应可以缓存在每个 CDN 服务器上，以便本地服务器直接为其他请求提供服务，而不是由源服务器提供服务，从而为用户节省时间，并为源服务器节省负载和带宽。

CDN 是增压反向代理。在 HTTP/2 出现之前,CDN 主要用于性能原因,但现在它们也为 HTTP/2 提供了一个简单的升级选项。

CDN 可以处理 HTTP/2 所需的 HTTPS 要求。但是,如果源服务器未实施 HTTPS,则网络流量将仅针对其在互联网上的某些旅程进行加密。尽管使用 HTTPS 的许多原因是为了减轻客户端的风险(例如,连接到未知的 Wi-Fi 网络涉及只有 HTTPS 才能减轻的风险),但仍然首选使用 HTTPS 来实现完整的端到端连接。许多人说,在 CDN 上卸载 HTTPS 而在其余的互联网旅程中使用 HTTP 是不诚实的,因为您的网站访问者不会知道他们的密码可能会在不安全的情况下穿越网站。但是如果在您的服务器上获得 HTTP/2 所需的严格 HTTPS 设置并不容易(比如 ALPN 支持),至少你可以让 CDN 处理这个过程并通过 HTTP/1.1 恢复到旧的 HTTPS 配置与原始服务器的连接。

CDN 提供了许多好处,并且易于支持 HTTP/2 是考虑使用它们的另一个原因。CDN 可以快速实施 HTTP/2,有些甚至提供免费套餐,为较小的站点提供轻松访问 HTTP/2 的途径。但是,CDN 将能够对流量进行解密,因此您必须对此类可以访问您的流量的第三方感到满意。

3.2.4 实现HTTP/2总结

您可以通过多种方式为您的网站启用 HTTP/2,具体取决于您用于服务网络流量的基础设施结构。不幸的是,目前最明显的在 Web 服务器上实现 HTTP/2 的方法很困难,并且对于旧版本可能需要大量手动操作。随着技术的普及和服务器软件的升级,这种情况会有所改善,但就目前而言,这个过程比应有的痛苦得多。

存在用于添加 HTTP/2 支持的其他选项,希望实施 HTTP/2 的网站所有者应该了解它们。反向代理可以在现场或作为服务拥有和管理,例如通过使用 CDN。这些方法可以是实现 HTTP/2 的简单方法,直到服务器端支持在常见的服务器发行版中得到广泛传播。

此时,您应该能够为您的网站选择最佳的 HTTP/2 升级选项,并开始尝试看看它对您的网站有何不同。

如果你有一个支持 HTTP/2 的服务器来尝试这些例子,本书的其余部分会更有意义,尽管有些项目可以在公共网站上展示。

3.3 HTTP/2 设置故障排除

查看是否正在使用 HTTP/2 的最简单方法是查看浏览器中的开发人员工具。尽管 HTTP/2 似乎在 Web 服务器上启用,但由于提到的许多微妙之处,许多人都在努力使 HTTP/2 正常工作

贯穿本章。以下是我接触 HTTP/2 以来看到的一些常见原因：

您的 Web 服务器不支持 HTTP/2。显然，您的服务器需要支持 HTTP/2。正如本章所讨论的，目前服务器上的大多数默认安装都不支持 HTTP/2。检查您正在运行的服务器软件版本以及添加 HTTP/2 支持的时间。请注意，单独安装最新更新（例如使用 yum update 或 apt-get）不一定会将您的 Web 服务器更新到支持 HTTP/2 的版本。您的 Web 服务器上未启用 HTTP/2。即使您的 Web 服务器支持 HTTP/2，它也可能未启用。某些服务器（例如 IIS）默认启用 HTTP/2 支持。在其他服务器（例如 Apache）上，HTTP 支持取决于您使用的配置或构建。例如，ApacheHaus Windows 构建默认启用 HTTP/2，但从源代码安装默认不启用它。

此外，从 2.4.27 版本开始，Apache 在使用 prefork mpm¹⁰ 时不再支持 HTTP/2

此外，需要一些编译选项（例如 Apache 的 --enable-http2 和 nginx 的 --with http_v2_module）以允许打开 HTTP/2，但不要默认打开它。如果 HTTP/2 不工作，请查看您的 Web 服务器的文档以了解如何打开它。

您的 Web 服务器未启用 HTTPS。如第 3.1.1 节所述，Web 浏览器仅通过 HTTPS 支持 HTTP/2，因此如果您的网站仅支持 HTTP 而不是 HTTPS，您将无法在浏览器中使用 HTTP/2，直到您切换您的站点到 HTTPS。

您的网络服务器未启用 ALPN 支持。ALPN 是用于创建 HTTPS 会话的 TLS 协议的扩展，它允许服务器通告它支持 HTTP/2。一些 Web 浏览器（在撰写本文时，Safari、Edge 和 Internet Explorer）允许您在旧的 NPN 和新的 ALPN 方法上使用 HTTP/2。其他浏览器（例如 Chrome、Firefox 和 Opera）仅使用较新的 ALPN 方法。

测试 ALPN 支持的最简单方法是使用在线工具，例如 SSL Labs¹¹（它对 HTTPS 设置运行综合测试，但需要几分钟才能运行）或 KeyCDN HTTP/2 Test¹²（速度更快，因为它只测试用于 HTTP/2 和 ALPN 支持）。如果您的 Web 服务器不可公开访问，则您不能使用 Web 工具来测试此支持，而必须使用命令行工具，例如 OpenSSL 的 s_client（假设您的 OpenSSL 版本支持 ALPN）：

```
openssl s_client -alpn h2 -connect www.example.com:443 -status
```

¹⁰ https://github.com/icing/mod_h2/releases/tag/v1.10.7

¹¹ <https://www.ssllabs.com/ssltest/> <https://tools.keycdn.com/>

¹² [http2-test](#)

或者,下载 testssl 工具,¹³它执行大部分与 SSLLabs 相同的测试。但它需要支持 ALPN 的 OpenSSL 版本才能全面测试 HTTP/2 支持。

与网络浏览器类似,一些网络服务器 (如 Apache)仅使用 ALPN;其他 (如 nginx)使用 ALPN 或 NPN。您的服务器是否支持 ALPN 取决于您使用的 TLS 库的版本。表 3.3 显示了公共库中的 ALPN 支持。如果您不确定自己使用的是什么 TLS 库,很可能您使用的是适用于 Linux 的 OpenSSL、适用于 macOS 的 LibreSSL 或适用于 Windows 的 SChannel。

表 3.3 常见 TLS 库中的 ALPN 支持

TLS 库	添加 ALPN 支持的版本
打开SSL	1.0.2
自由SSL	2.5.0
SChannel (由 Microsoft 应用程序使用)	8.1 / 2012 R2
GnuTLS	3.2.0

即使您的 TLS 库支持 ALPN,您的服务器软件也可能不是使用该版本的 TLS 库构建的。例如,RHEL/CentOS 7.4 添加了 OpenSSL 1.0.2,但默认安装的 Apache 和 nginx 版本仍使用 OpenSSL 1.0.1 构建,因此它们不支持 ALPN。

Apache 通常会在重启时的错误日志中添加一行,详细说明它运行的 OpenSSL 版本:

```
[mpm_worker:notice] [pid 19678:tid 140217081968512] AH00292: Apache/2.4.27 (Unix) OpenSSL/1.0.2k-fips配置 恢复正常操作
```

或者,您可以针对 mod_ssl 模块运行 ldd 并查看它链接的版本:

```
$ ldd /usr/local/apache2/modules/mod_ssl.so | grep libssl
libssl.so.10 => /lib64/libssl.so.10 (0x00007f185b829000)
$ ls -la /lib64/libssl.so.10 lrwxrwxrwx。 1 root
root 16 Oct 15 16:07 /lib64/libssl.so.10 -> libssl.so.1.0.2k
```

对于 nginx,您可以使用 -V 选项来显示构建:

```
$ nginx -V nginx
版本:nginx/1.13.6 由 gcc 4.8.5 20150623 (Red
Hat 4.8.5-16) (GCC) 构建
```

¹³ <https://testssl.sh/>

使用OpenSSL 1.0.2k-fips构建2017年1月26日TLS SNI支持启用配置
参数：--with-http_ssl_module --with-http_v2_module

对于其他服务器,请参阅您的支持文档。

您的 Web 服务器上未启用强 HTTPS 密码。HTTP/2 规范列出了客户端不得用于 HTTP/2 连接的几种密码。¹⁴某些浏览器（例如 Chrome）不使用这些密码的 HTTP/2，因此您的服务器必须设置更好的密码如果您想使用 HTTP/2（在撰写本文时意味着使用 ECDHE GCM 或 POLY 密码）。大多数默认安装包括更强的密码，通常优先于较弱的密码使用，但如果您从以前的安装移植旧的密码配置，这些密码可能不会启用。

要检查您的 HTTPS 密码设置，请使用 SSL Labs 在线测试工具。

该工具最初可能难以理解，但可以为您提供有关 HTTPS 设置的完整信息，包括普通客户端是否支持 HTTP/2。

Mozilla SSL Configuration Generator¹⁵也是一个有用的工具，用于提供常见 Web 浏览器所需的 HTTPS 配置。大多数站点应使用现代设置，但如果需要支持较旧的客户端，则可能需要使用中级设置。正在使用拦截代理并将您降级到 HTTP/1.1。如果您使用代理（例如，在公司环境中）或防病毒软件，这些元素可能会将您的连接降级为 HTTP/1.1，因为它们会拦截 HTTPS 连接。我在 3.1.1 节中讨论了这个主题。查看网站的 HTTPS 证书是否由真正的证书颁发机构颁发。

如果您的网站是面向公众的，您可以使用 SSL Labs 或 KeyCDN HTTP/2 Test¹⁶ 等在线工具来查看您的网站是否支持 HTTP/2。如果是这样，问题可能是您的本地问题，可能是由拦截代理引起的。

对于病毒软件，通常可以关闭 HTTPS 拦截或
将某些网站列入白名单。

错误转发了升级标头。后端服务器（例如 Apache）可能会使用 Upgrade: h2 标头来建议切换到 HTTP/2。如果该标头由反向代理盲目转发，即使它不理解 HTTP/2，该标头也会导致问题。浏览器尝试升级到 HTTP/2（正如标头所建议的那样）但失败了，因为反向代理不理解 HTTP/2。反向代理不应该转发

¹⁴ <https://httpwg.org/specs/rfc7540.html#BadCipherSuites>

¹⁵ <https://mozilla.github.io/server-side-tls/ssl-config-generator/>

¹⁶ <https://tools.keycdn.com/http2-test>

在这种情况下升级标头。我将在第 4 章进一步讨论这个主题。Safari 处理这种情况的能力特别差，通常会出现 nsposixerrordomain:100 错误。 HTTPS 标头无效。 Chrome 会为无效的 HTTP 标头（例如标头名称中的空格或双冒号）返回 ERR_SPDY_PROTOCOL_ERROR 消息¹⁷，即使它对 HTTP/1.1 中的相同错误更为宽容。

出于同样的原因， Safari 可以返回 nsposixerrordomain:100 错误。 缓存项目报告原始下载协议。 如果您将服务器升级到 HTTP/2 并尝试对其进行测试，如果不先清除缓存，您可能仍在使用缓存的资源。 缓存项显示最初用于下载请求的 HTTP 版本（如果在升级前下载，则可能是 HTTP/1.1）。 同样，如果服务器发送 304 Not Modified 响应，浏览器将使用缓存的资源并显示用于下载它的任何协议。

概括

客户端对 HTTP/2 的支持很强，几乎所有主流浏览器都支持移植它。

服务器端的 HTTP/2 支持在较新的版本中可用，但通常情况下，如果没有完整的服务器升级和/或手动安装，这些版本不容易安装。

启用 HTTP/2 的各种选项都可用，包括使用 CDN 等第三方基础设施来提供此支持。 有几个原因导致 HTTP/2 即使已经被使用也可能不被使用

启用。

¹⁷ https://www.michalspacek.com/chrome-err_spdy_protocol_error-and-an-invalid-http-header

第2部分

使用 HTTP/2

在 本书的第一部分介绍了新版本 HTTP 的需求和动机,介绍了 HTTP/2,并描述了为您的网站设置 HTTP/2 的方法。对于很多人来说,这些信息就足够了,大多数网站应该开始看到升级到 HTTP/2 的好处,即使没有进行任何其他更改。HTTP/2 被设计为易于迁移到大多数站点并为大多数站点提供立竿见影的好处,而无需进行任何更改。

然而,要真正受益于 HTTP/2 必须提供的所有功能,深入了解该协议及其工作原理是很有用的。本书的这一部分描述了该协议的核心方面。第 4 章和第 5 章涵盖了允许网站所有者和开发人员充分利用 HTTP/2 的协议的技术细节。第 6 章从协议本身开始讨论它对 Web 性能意味着什么,以及开发人员应该改变哪些实践来优化这个新的 HTTP/2 世界。

HTTP/2 协议基础

本章涵盖 HTTP/2 的基础知识:它是什么以及它与 HTTP/1.1 的区别

客户端和服务器如何同意使用 HTTP/2 而不是 HTTP/1.1

HTTP/2 帧以及如何调试它们

本章涵盖了 HTTP/2 的基础知识（帧、流和多路复用）。我在第 7 章和第 8 章讨论了协议的更高级部分（特别是流优先级和流量控制）。HTTP/2 规范¹是协议的最终参考点，可以在本章之后或与本章一起参考，但是本章中添加的细节和示例（我希望）将使学习该协议更加容易。

¹ <https://tools.ietf.org/html/rfc7540>

4.1 为什么用 HTTP/2 而不是 HTTP/1.2?

我在第 2 章中谈到了 HTTP/1 和 HTTP/2 之间的区别。HTTP/2 是专门为解决 HTTP/1 中的性能问题而创建的,新版本的协议通过添加以下概念而有所不同: 二进制而非二进制文本协议多路复用而非同步流量控制

流优先级头压缩服务器推
送

这些概念 (在本章中有更详细的描述)是基础性的,破坏了协议的变化,因为它们不向后兼容;尽管 HTTP/1.0 网络服务器可以理解 HTTP/1.1 消息并忽略更高版本引入的额外功能,但对于具有不同结构和格式的 HTTP/2 消息而言并非如此。因此,HTTP/2 被视为主要版本升级。

大多数差异涉及 HTTP/2 在客户端和服务器之间的传输方式。在比大多数 Web 开发人员处理的更高级别 (HTTP 语义)2, HTTP/2 的行为很像 HTTP/1。它具有相同的方法 (GET、POST、PUT 等);它使用相同的 URL、相同的响应状态代码 (200、404、301、302) 和相同的 HTTP 标头 (大部分)。HTTP/2 是发出相同 HTTP 请求的更有效方式。

在许多方面,HTTP/2 与 HTTPS 相似,因为它在发送前以特殊格式有效地包装标准 HTTP 消息,并在接收后解包。因此,尽管客户端 (Web 浏览器) 和服务器 (Web 服务器) 需要知道来回发送消息的确切细节,但更高级别的应用程序不需要对这些版本有太大区别,因为 HTTP 的基本概念他们使用的是相似的。然而,与 HTTPS 不同的是,HTTP/2 应该会导致不同的网站开发方式。就像对 HTTP/1 的深刻理解导致第 2 章中讨论的 Web 优化一样,对 HTTP/2 的深刻理解导致不同的优化,这些优化可以使 Web 开发人员变得更好,并帮助使网站更快。因此,了解这些核心差异非常重要。

HTTP/2.0 还是 HTTP/2?

HTTP/2 最初称为 HTTP/2.0,但 HTTP 工作组决定删除次要版本号 (.0),改用 HTTP/2。正如我之前提到的,HTTP/2 定义了这个新版本 HTTP 的主要部分 (二进制、多路复用等)

[“] <https://tools.ietf.org/html/draft-ietf-httpbis-semantics>

on),并且任何未来的实现或更改(例如HTTP/2.1)都应该是兼容的。同样的事情也发生在HTTP/1(一个从未流行过的术语,但我在整本书中用来表示HTTP/1.0和HTTP/1.1),它是一种基于文本的协议,带有标头和主体。

此外,与HTTP/1消息不同,版本号未在请求中明确说明。例如,在HTTP/2中没有GET /index.html HTTP/1.1风格的请求。但是,许多实现在日志文件中使用次要版本。例如,在Apache日志文件中,版本号显示为HTTP/2.0,文件甚至伪造了一个HTTP/1风格的请求:

```
78.1.23.123 -- [14/Jan/2018:15:04:45 +0000] 2 GET /index.html 200 1797 - Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36
```

因此,您会在日志中看到HTTP/1类型的消息,尽管我之前的声明与此相反。然而,这个请求并不是真正的请求,它是由网络服务器模拟出来的,以便于解析日志消息,而不是协议直接发送的消息。事实上,规范中只有一个地方引用了HTTP/2.0前言消息(在4.2.5节中讨论)。

4.1.1 二进制而不是文本

HTTP/1和HTTP/2之间的主要区别之一是HTTP/2是二进制的、基于数据包的协议,而HTTP/1是完全基于文本的。基于文本的协议对人类来说很容易理解,但对计算机来说更难解析。这种情况对于HTTP最初作为简单的请求和响应协议来说是可以接受的,但越来越多地限制了该协议在现代互联网中的使用。

使用基于文本的协议,需要先发送请求并接收完整的响应,然后才能处理另一个请求。总的来说,HTTP在过去20年中一直以这种方式工作,尽管进行了一些小的改进。HTTP/1.0引入了二进制HTTP主体,例如,可以在其中发送图像和其他媒体作为响应,而HTTP/1.1引入了流水线技术(参见第2章)和分块编码。分块编码允许首先发送消息正文的一部分,然后在可用时发送其余部分。HTTP主体被分成块,接收分块响应的客户端(或接收分块请求的服务器)可以在消息完全接收之前开始处理消息。当事先不知道动态生成的数据长度时,通常会使用此技术。分块编码和流水线都存在队头(HOL)阻塞问题,队列顶部的消息会阻止发送后续响应,更不用说流水线在现实世界中得不到很好支持的事实。

HTTP/2转向完整的二进制协议,其中HTTP消息被拆分并以明确定义的帧发送。所有HTTP/2消息都有效地使用分块编码

作为标准,不需要明确设置。事实上,HTTP/2 规范指出

RFC7230 第 4.1 节中定义的分块传输编码不得在 HTTP/2 中使用。

这些帧类似于构成大多数 HTTP 连接基础的 TCP 数据包。

收到所有帧后,即可重建完整的 HTTP 消息。

尽管在很多方面都像 TCP,HTTP/2 仍然通常位于 TCP 之上而不是取代它 (尽管谷歌正在试验用 QUIC 取代 TCP 并在其之上实现更轻量级的 HTTP/2,正如我在第 1 章中讨论的那样) 9).诸如 TCP 之类的底层协议用于保证消息按顺序到达,而无需将此类处理添加到 HTTP/2 协议中。

HTTP/2 的二进制表示用于消息的发送和接收,但消息本身类似于旧的 HTTP/1 消息。二进制框架通常由较低级别的客户端或库 (Web 浏览器或 Web 服务器) 处理。如前所述,更高级别的应用程序 (例如 JavaScript 应用程序) 不需要关心消息的发送方式,并且在大多数情况下可以将 HTTP/2 连接完全视为 HTTP/1.1 连接。然而,了解甚至查看 HTTP/2 帧以调试意外错误可能会有所帮助 尤其是在采用的早期阶段,当您可能需要调试某些 (我希望很少见!) 场景中的实现问题时。

4.1.2 多路复用而非同步

HTTP/1 是一个同步的、单一的请求和响应协议。客户端发送 HTTP/1 消息,服务器收到 HTTP/1 响应。第 2 章讨论了为什么该协议效率低下,尤其是考虑到现代万维网,一个网站通常由数百种资源组成。HTTP/1 中的主要变通方法是打开多个连接或发送较少的大请求而不是许多小请求,但这两种变通方法都引入了它们自己的问题和低效。图 4.1 显示了如何使用三个 TCP 连接来并行发送和接收三个 HTTP/1 请求。请注意,初始页面的请求 1 未显示,因为只有在此初始请求之后,才需要在请求 2-4 中并行请求多个资源。

HTTP/2 允许在单个连接上同时处理多个请求,对每个 HTTP 请求或响应使用不同的流。通过移动到二进制帧层,每个帧都有一个流标识符,使同时发生多个独立请求的概念成为可能。当接收到该流的所有帧时,接收方可以重建完整的消息。

帧是允许同时发送多条消息的关键。每个帧都被标记以指示它属于哪个消息 (流),这允许在同一时间同时发送或接收两个、三个或一百个消息。

为什么使用 HTTP/2 而不是 HTTP/1.2?

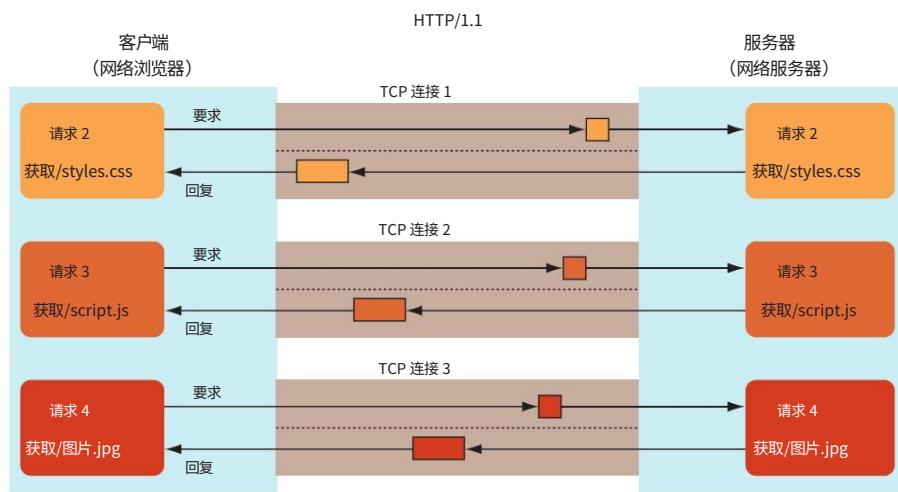


图 4.1 并行的多个 HTTP/1 请求需要多个 TCP 连接。

多路复用连接,而不是大多数浏览器允许的六个并行 HTTP/1 连接。图 4.2 显示了与图 4.1 相同的三个请求,但是请求在同一连接上一个接一个地发送 (类似于 HTTP/1.1 流水线),并且响应被混合发回 (这在 HTTP/1.1 中是不可能的) 1.1 流水线)。

这个例子表明请求并不是在完全相同的时间发送的,因为最终,每个帧都需要在同一个 HTTP/TCP 连接上一个接一个地发送。HTTP/1.1 也是如此,因为即使请求看起来是

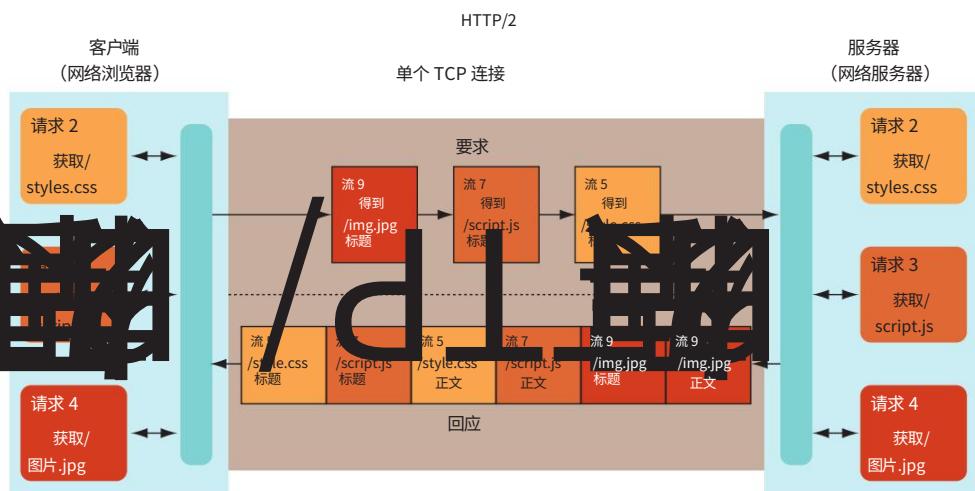


图 4.2 通过多路复用 HTTP/2 连接请求三个资源

在多个连接上并行,只有(至少通常!)只有一个网络连接,因此每个请求都将在网络级别排队等待发送。要点是 HTTP/2 连接在发送请求后直到收到响应后才被阻塞,就像在 HTTP/1.1 中一样(如第 2 章所述)。

同样,响应可以混合发回(图 4.2 中的流 5 和 7),或按顺序发回(图 4.2 中的流 9)。服务器发回响应的顺序完全取决于服务器,尽管客户端可以指定优先级。如果可以发送多个响应,服务器可以优先考虑重要资源(如 CSS 和 JavaScript)而不是不太重要的资源(如图像)。我在第 7 章中介绍了这个主题。

每个请求都被赋予一个新的、递增的流 ID(图 4.2 中的 5、7 和 9),并且响应以相同的流 ID 发回,因此流是双向的,就像 HTTP 连接一样。响应完成后,流将关闭。HTTP/2 流并不直接类似于 HTTP/1.1 连接,因为流被丢弃且未被重用,而 HTTP/1.1 保持连接打开并且它可以被重用以发送另一个请求。

为了防止流 ID 冲突,客户端发起的请求被赋予奇数流 ID(这就是我在前面的图中使用流 ID 5、7 和 9 的原因,假设流 1 和 3 已在此连接上使用),和服务器发起的请求甚至被赋予流 ID。请注意,在撰写本文时,服务器在技术上无法启动流,除非在特定用例中,这些用例最终仍会响应客户端流,正如我在第 5 章中讨论的那样。对请求的响应标记有相同的流 ID,如前面提到的。流 ID 0(图中未显示)是客户端和服务器用来管理连接的控制流。

理解图 4.2 是理解 HTTP/2 的关键。如果您理解了这个概念并了解它与 HTTP/1 的不同之处,那么您就已经在理解 HTTP/2 的道路上走了很长一段路。虽然存在错综复杂和细微之处,但该图展示了 HTTP/2 的两个基本原理:

HTTP/2 使用多个二进制帧通过单个 TCP 连接发送 HTTP 请求和响应,使用多路复用流。

HTTP/2 在消息发送层面上大不相同,甚至在更高的层面上,HTTP 的核心概念保持不变。请求有一个方法(比如 GET)、一个你想获取的资源(比如 /styles.css)、headers、body、状态码(比如 200、404)、缓存、cookies 等等。相同的。

第一点意味着 HTTP/2 可以如图 4.3 所示,其中每个流在 HTTP/1 世界中就像一个单独的连接。但是图 4.3 可能会被熟悉 HTTP/1 的人误解,因为 HTTP/2 流没有被重用(因为连接在 HTTP/1 中)并且 HTTP/2 流不是完全独立的。这是有充分理由的,正如您将在第 7 章中看到的那样。

第二点是 HTTP/2 能够取得如此大进展的原因。只要 Web 浏览器和 Web 服务器知道 HTTP/2 并且可以处理

为什么使用 HTTP/2 而不是 HTTP/1.2?

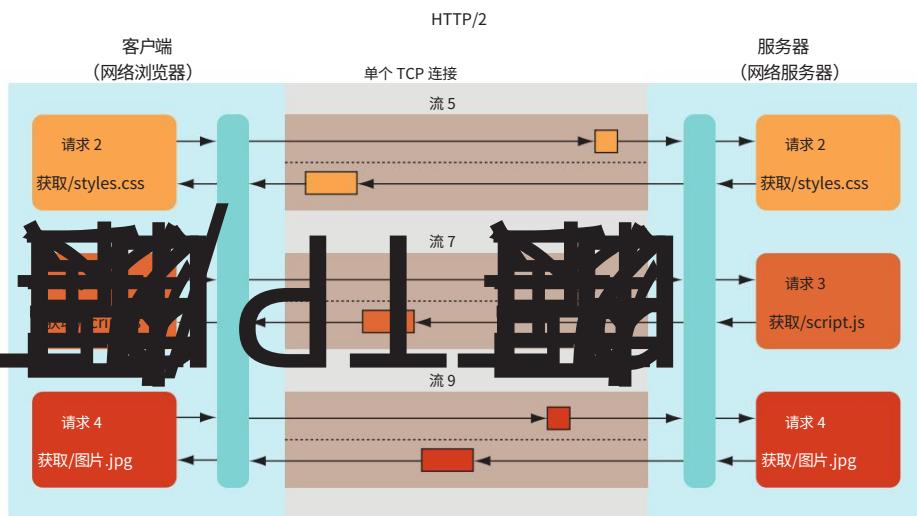


图 4.3 HTTP/2 流类似于 HTTP/1 连接。

低级别的细节,用户 (最终用户和网络开发人员)不需要了解 HTTP/2 或以与对待任何其他网站的方式不同的方式对待它。

HTTP/2 甚至不需要新方案 (URL 开头的 https:// 位),正如我在 4.2 节中讨论的那样。事实上,大多数人在不知不觉中使用 HTTP/2 一段时间了。Google、Twitter 和 Facebook 都使用 HTTP/2,因此如果您使用过这些网站,即使您没有意识到,您也可能已经使用过 HTTP/2。

但是,正如对 HTTP/1 的深入了解使 Web 开发人员能够创建更好、性能更高的网站一样,那些花时间了解 HTTP/2 工作原理的开发人员将更好地让您的网站从该协议中受益。

4.1.3 流优先级和流量控制

在 HTTP/2 之前,HTTP 是一个单一的请求和响应协议,因此不需要在协议内划分优先级。客户端 (通常是 Web 浏览器)通过使用有限数量的 HTTP/1 连接 (通常是六个)决定发送消息的顺序来决定 HTTP 之外的优先级。这种优先级排序通常需要首先请求关键资源 (HTML、渲染阻塞 CSS 和关键 JavaScript),然后再请求非阻塞项目 (例如图像和异步 JavaScript)。请求排队,等待免费的 HTTP/1 连接,由浏览器管理的排队决定优先级。

HTTP/2 现在对任何时候正在运行的请求数量都有更高的限制 (在许多实现中通常默认为 100 个活动流),因此许多请求不再需要浏览器排队,可以立即发送。

这一事实可能导致带宽被浪费在优先级较低的资源上 (例如

图像),因此导致页面在 HTTP/2 上看起来加载速度较慢。需要流优先级,以便可以以更高的优先级发送最关键的资源。流优先级是通过在帧队列等待发送时服务器为高优先级请求发送比低优先级请求更多的帧来实现的。流优先级还允许比 HTTP/1 下更好的控制,在 HTTP/1 下,单独的连接是独立的。在 HTTP/1 中,除了不使用连接外,无法确定某些连接的优先级。如果你有五个关键资源和第六个非关键资源,在 HTTP/1 下,所有资源都可以在六个单独的连接上以相同的优先级发送,或者可以发送前五个而第六个被阻止。在 HTTP/2 下,所有六个请求都可以以适当的优先级发送,这个优先级用于决定分配多少资源来发送每个响应。

流量控制是在同一连接上使用多个流的另一个必然结果。如果接收方无法像发送方发送的那样快速处理传入消息,则存在积压,必须对其进行缓冲,最终导致数据包被丢弃并需要重新发送。在这种情况下,TCP 允许在连接级别限制连接,但 HTTP/2 要求它发生在流级别。以带有实时视频的网页为例。如果视频被用户暂停,谨慎的做法是只暂停该 HTTP/2 流的下载,但允许网站使用的任何其他资源通过其他流继续满负荷下载。

在第 7 章中,我将返回到流优先级和流量控制,并提供有关它们如何工作的更多细节。这些进程通常由浏览器和服务器控制,因此在撰写本文时用户和 Web 开发人员几乎无法控制它们,这就是我在本书后面讨论它们的原因。

4.1.4 头部压缩

HTTP 标头用于从客户端向服务器发送有关请求和响应的附加信息,反之亦然。这些标头中有很多重项,因为它们通常针对每个资源以相同方式发送。考虑以下标头,它们随每个请求一起发送,并且经常重复之前发送的值:

Cookie Cookie 随每个请求一起发送到该域 (Amazon 使用的未经授权的请求除外 [参见第 2 章],但这些请求有些特殊且不规范)。Cookie 标头可能会变得特别大,通常只有 HTML 文档资源才需要,但每次请求都会发送。

User-Agent 这个头部通常说明正在使用的网络浏览器。它从来没有在会话期间更改,但它仍会随每个请求一起发送。

Host 这个头用于完全限定请求 URL,并且对于同一主机的每个请求总是相同的。接受 这个头定义了它期望的响应格式 (浏览器知道如何显示的可接受的图像格式,等等)。因为

浏览器支持的格式通常不会在不升级浏览器的情况下发生变化,此标头随请求类型（图像、文档、字体等）而变化,但对于这些类型的每个实例都是相同的。`Accept-Encoding` 这个头文件定义了压缩格式（通常是`gzip`、`deflate`,并且越来越多的`brotli`用于接受较新的`brotli`压缩的浏览器）。与`Accept`标头类似,此标头在整个会话期间不会更改。

这些响应标头可以重复并且很浪费。一些专门的响应标头（例如内容安全策略标头）可能很大并且同样重复对于较小的请求尤其不利,其中 HTTP 标头将按比例占下载的较大部分。

HTTP/1 允许压缩 HTTP 主体（因此,前面提到的`Accept-Encoding`标头）,但不允许压缩 HTTP 标头。HTTP/2 引入了标头压缩的概念,但正如我在第 8 章中讨论的那样,它使用一种不同于主体压缩的技术来允许交叉请求压缩并防止用于 HTTP 主体压缩的算法出现一些安全问题。

4.1.5 服务器推送

HTTP/1 和 HTTP/2 的另一个重要区别是 HTTP/2 增加了服务器推送的概念,它允许服务器以多个响应响应一个请求。在 HTTP/1 下,当返回主页时,浏览器必须读取它,然后在开始渲染页面之前请求其他资源（例如 CSS 和 JavaScript）。使用 HTTP/2 服务器推送,这些资源可以与初始响应一起发送,并且在浏览器希望使用它们时应该可用。

HTTP/2 服务器推送是 HTTP 中的一个新概念,但如果注意,当以任何方式推送浏览器不需要的资源时,它很容易导致带宽浪费,特别是如果被推送的资源是用先前的请求并且已经在浏览器缓存中可用。决定何时以及如何推送是充分利用此功能的关键。出于这个原因,HTTP/2 服务器推送在本书中有自己的一章（第 5 章）。

4.2 如何建立 HTTP/2 连接

鉴于 HTTP/2 在连接级别与 HTTP/1 如此不同,客户端 Web 浏览器和服务器都需要能够交谈和理解 HTTP/2 才能使用它。由于涉及两个独立的参与方,因此需要有一个过程让每一方都可以说它愿意并且能够使用 HTTP/2。

迁移到 HTTPS 是最后一个类似的变化,这是通过新的 URL 方案 (`https://`) 实现的,并通过不同的默认端口（HTTPS 为 443,HTTP 为 80）提供服务。此更改允许清楚地分离协议,因此清楚地指示使用哪种协议进行通信。

然而,转向新方案、端口或两者都有一些缺点,包括

在接近普遍采用之前,默认值需要保留现有的 `http://` (或 `https://`,如果它成为默认值,正如许多人所希望的那样)。因此,像 HTTPS 那样添加一个新方案将需要重定向以使用 HTTP/2,这会导致速度变慢。这正是 HTTP/2 应该解决的问题! 网站必须更改新方案的链接。尽管内部链接可以通过网站本身的相关链接来固定(例如, `/images/image.png` 而不是`https://example.com/images/image.png`) ,但外部链接需要包含完整的 URL,包括计划。采用 HTTPS 通常很复杂,部分原因是网站需要将每个 URL 更改为新方案。

现有网络基础设施出现兼容性问题(例如火
墙阻挡任何新的非标准端口)。

出于这些原因以及为了更加无缝地过渡到 HTTP/2,HTTP/2 (和它所基于的 SPDY)决定不使用新方案,而是考虑使用替代的本机方法来建立 HTTP/2 连接。HTTP/2 规范3 提供了三种创建 HTTP/2 连接的方法(尽管后来添加了第四种方法,如第 4.2.4 节所述) :

使用 HTTPS 协商。 使用 HTTP
升级标头。 使用先验知识。

理论上,HTTP/2 可通过未加密的 HTTP (称为 h2c) 和加密的 HTTPS (称为 h2) 使用。实际上,所有 Web 浏览器仅通过 HTTPS (h2) 支持 HTTP/2,因此选项 1 用于由 Web 浏览器协商 HTTP/2。服务器到服务器的 HTTP/2 通信可以通过未加密的 HTTP (h2c) 或 HTTPS (h2) 进行,因此它可以使用所有这些方法,具体取决于所使用的方案。

4.2.1 使用 HTTPS 协商

HTTPS 连接通过协议协商阶段来建立连接,因为它们需要在建立连接和交换 HTTP 消息之前就 SSL/TLS 协议、密码和要使用的各种其他设置达成一致。这个阶段是灵活的,允许引入新的 HTTPS 协议和密码,只有在客户端和服务器都同意使用它们时才使用它们。HTTP/2 支持可以成为 HTTPS 握手的一部分,从而保存在连接建立时和发送第一个 HTTP 消息之前需要完成的任何升级重定向。

¹⁰ <https://tools.ietf.org/html/rfc7540#section-3>

HTTPS握手

使用 HTTPS 意味着使用 SSL/TLS 来加密标准的 HTTP 连接,无论是 HTTP/1 还是 HTTP/2。请参阅第 1 章的“SSL、TLS、HTTPS 和 HTTP”边栏,了解这些首字母缩略词与本书中使用的命名约定之间的区别。

公私密钥加密称为非对称加密,因为它使用不同的密钥来加密和解密消息。需要这种类型的加密来允许与您以前从未连接过的服务器进行安全通信,但速度很慢,因此它用于就用于加密其余连接的对称加密密钥达成一致。此协议发生在 TLS 握手期间,该握手发生在连接开始时。在目前使用的主要版本 TLSv1.2 下,它使用图 4.4 所示的握手来建立加密连接。如第 9 章所述,这种握手在新标准化的 TLSv1.3 中略有变化。

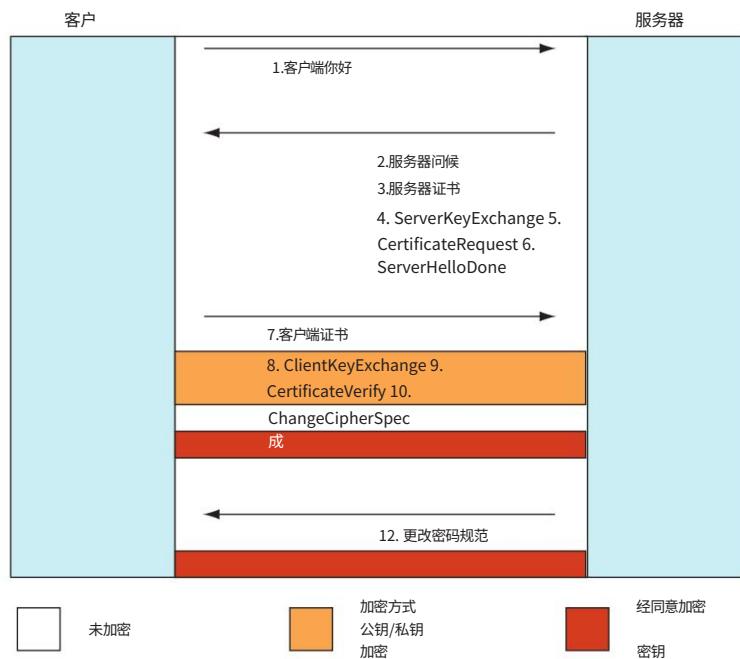


图 4.4 HTTPS 握手

握手涉及四组消息：

客户端发送一条ClientHello消息,详细说明其加密功能。
此消息未加密发送,因为尚未就加密方法达成一致。

服务器发回类似的ServerHello消息,根据它知道客户端支持的内容选择 HTTPS 协议 (例如 TLSv1.2)。它还会发送将用于此连接的密码 (例如ECDHE-RSA-AES128-GCM-SHA256),再次根据Client Hello消息中公布的那些以及它自身支持的内容选择一个。然后它提供服务器HTTPS证书 (ServerCertificate)。密钥详细信息取决于所选的密码(ServerKeyExchange)以及是否需要客户端 HTTPS 证书 (CertificateRequest,大多数网站不需要)。最后,服务器说它完成了 (ServerHelloDone)。客户端验证服务器证书并在请求时发送客户端证书 (ClientCertificate,大多数站点不需要)。然后它发送其秘密密钥详细信息(ClientKeyExchange)。这些详细信息由服务器证书中的公钥加密发送,因此只有服务器才能使用私钥解密消息。如果正在使用客户端证书,则发送CertificateVerify消息,用私钥签名,以证明客户端证书的所有权。客户端使用ServerKeyExchange 和ClientKeyExchange详细信息定义加密的对称密钥,并发送ChangeCipherSpec消息通知客户端加密开始;然后它发送加密的Finished消息。服务器也切换到加密连接 (ChangeCipherSpec)并发送加密的Finished消息。

除了用于就要使用的对称加密密钥达成一致外,公有私钥加密还用于确认身份,因为如果您正在与错误的一方进行安全对话,那么使用强加密毫无意义!身份得到确认,因为消息由服务器的隐藏私钥签名,可以使用证书中的公钥解锁。每个 SSL/TLS 证书还由计算机信任的公证书颁发机构进行加密签名。如果正在使用客户端证书,则类似的过程会以相反的方式进行。关于身份,所有可以确认的是它所属的服务器域签署了 SSL/TLS 证书。

如果服务器域错误 (www.amaz0n.com而不是www.amazon.com) ,那么您正在与错误的服务器进行安全对话,并且可能与您认为正在与之交谈的一方不同。正如我在第 1 章介绍 HTTPS 时提到的,这种情况会引起很多混乱。绿色挂锁并不意味着网站是合法或安全的 只是与它的通信是安全加密的。

在所有这些步骤之后,HTTPS 会话就建立起来了,所有未来的通信都使用约定的密钥进行保护。在您可以发送单个请求之前,此设置至少添加两次往返。HTTPS 传统上被认为很慢,但尽管计算机的进步使得消息的加密和解密并不真正引人注目,但初始连接延迟是显而易见的,如第 2 章中的瀑布图所示。完成初始 HTTPS 设置后,未来的 HTTP 同一连接上的消息不需要经过此协商。同样,未来的联系

(无论它们是并行的额外连接还是稍后重新连接)如果它们在称为 TLS会话恢复的过程中重用上次使用的密钥,则可以跳过其中一些步骤。

除了尝试限制创建新连接(如 HTTP/2 所做的)之外,您对这种初始缓慢几乎无能为力。大多数人都认为 HTTPS 的好处超过了初始连接的性能成本。在撰写本文时完成的TLSv1.3,4引入了额外的效率来将此协商减少到一次往返(或者甚至是之前协商中拾取时的零次往返),但是它需要一些时间才能被采用和在许多情况下仍然需要往返一次。

应用层协议协商ALPN5向ClientHello消

息添加了一个额外的扩展,客户端可以在其中通告应用程序协议支持(“嘿,我支持 h2 和 http/1。如果你愿意,请使用它们中的任何一个。”),并且还可以 ServerHello 消息,其中服务器可以在 HTTPS 协商后确认使用哪个应用程序协议(“好的,让我们使用 h2”)。我在图 4.5 中展示了这个过程。

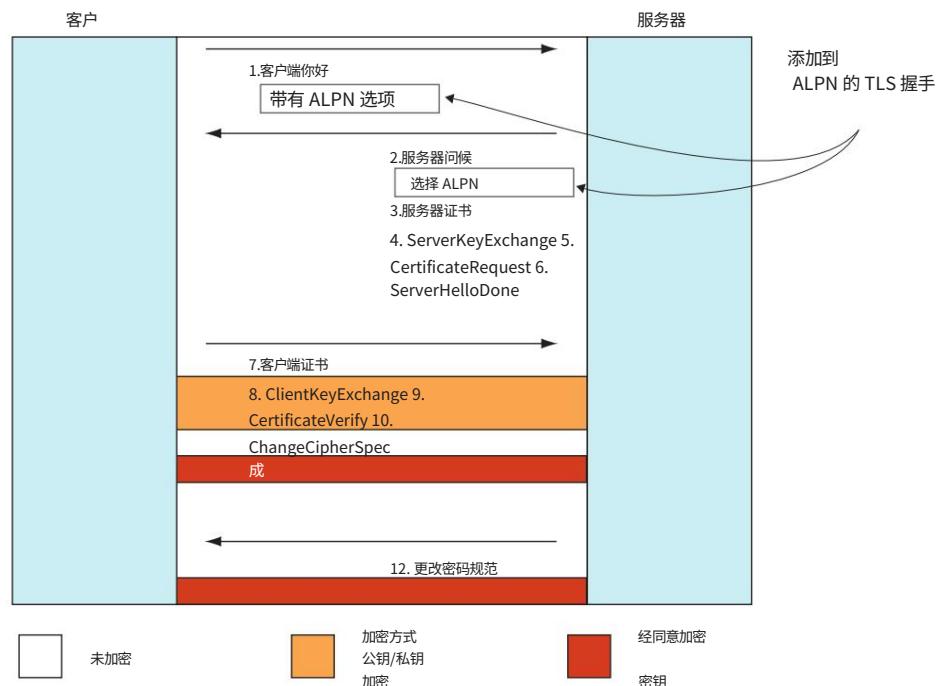


图 4.5 与 ALPN 的 HTTPS 握手

¹¹ <https://tools.ietf.org/html/draft-ietf-tls-tls13>

¹² <https://tools.ietf.org/html/rfc7301>

ALPN 很简单,可用于就是否对现有 HTTPS 协商消息使用 HTTP/2 达成一致,而无需添加任何进一步的往返、重定向或其他升级延迟。 ALPN 的唯一问题在于它相对较新,支持并不普遍,特别是在服务器端,旧版本的 TLS 库很常见 (请参阅第 3 章)。如果不支持 ALPN,服务器通常会假定客户端不支持 HTTP/2 并使用 HTTP/1.1。

ALPN 可用于除 HTTP/2 之外的其他协议,但在撰写本文时它仅用于 HTTP/2 及其所基于的 SPDY 协议,尽管其他 ALPN 应用程序已经注册,包括 HTTP 的原始三个版本:HTTP/0.9、HTTP/1.0 和 HTTP/1.1.6实际上,ALPN 在 HTTP/2 出现之前于 2014 年 7 月完成,并且 ALPN7 的 RFC 仅定义了 HTTP/1.1 和 SPDY 的扩展。 HTTP/2 ALPN 扩展 (h2) 后来注册,作为最终确定 HTTP/2 规范的一部分。⁸

下一个协议谈判

NPN,ALPN 的前身,以类似的方式工作。尽管被许多浏览器和 Web 服务器使用,但它从未正式成为互联网标准 (尽管制定了规范草案)⁹。正式化的 ALPN 主要基于 NPN,就像 HTTP/2 SPDY 的正式版本。

主要区别在于,对于 NPN,客户端决定使用的协议,而对于 ALPN,服务器决定 (决定其余 TLS 参数的方式)。使用 NPN, ClientHello 消息声明客户端很高兴使用 NPN, ServerHello 消息包含服务器支持的所有 NPN 协议,启用加密后,客户端选择 NPN 协议 (例如 h2) 并发送另一条消息有了这个选择。图 4.6 说明了这个过程,在步骤 1、2 和 11 中突出显示了三个额外的部分。

NPN 是一个三步过程,而 ALPN 是一个两步过程,尽管这两个过程都重用现有的 HTTPS 步骤并且不添加往返 (尽管 NPN 确实添加了一条消息来确认要使用的协议)。此外,对于 NPN,所选的应用程序协议是加密的 (图 4.6 中的步骤 11),而在 ALPN 中,它是在未加密的 ServerHello 消息中发送的。由于服务器支持的协议在 NPN 中的 ServerHello 消息中未加密发送,并且由于某些网络解决方案可能想知道将要使用的应用程序,因此 TLS 工作组决定在 ALPN 中更改此过程,以便服务器选择应用程序协议 (与其他 HTTPS 参数一样)。

NPN 已被弃用,取而代之的是 ALPN,并且如第 3 章所述,许多 Web 浏览器已停止支持 NPN 用于 HTTP/2 连接;一些网络服务器 (如 Apache) 从一开始就不支持它。其他实现是

⁶ <https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids> <https://tools.ietf.org/html/rfc7301> <https://tools.ietf.org/html/rfc7540#section-11.1> <https://tools.ietf.org/html/draft-agl-tls-nextprotoneg-04>

⁹

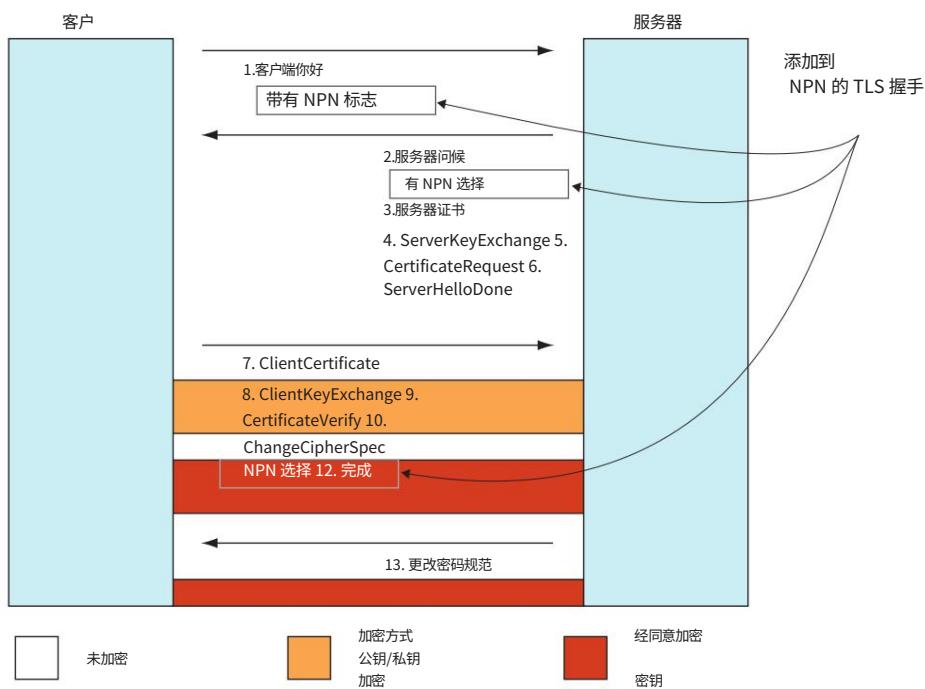


图 4.6 使用 NPN 的 HTTPS 握手

随着时间的推移,预计会弃用它。HTTP/2 规范声明应该使用 ALPN¹⁰而没有提及 NPN,因此从技术上讲,仍然使用 NPN 的实现不符合规范。

弃用 NPN 会导致尚不支持 ALPN 的服务器出现问题。

NPN 较旧,因此得到更多服务器(或至少是他们使用的 TLS 库)的支持。尽管较新的版本支持 ALPN,但即使在撰写本文时也很常见的旧版本(例如 OpenSSL 1.0.1)仅支持较旧的 NPN 扩展。这种情况是即使为服务器配置了 HTTP/2 也可能无法使用的原因之一(请参阅第 3 章)。

与ALPN的HTTPS握手示例

您可以使用一些工具来查看 HTTPS 握手,但 curl¹¹是最简单的工具之一,并且可用于许多环境,尽管它可能是不支持 ALPN 的版本。例如,对于那些使用 Git Bash 的人来说,它包含一个支持 ALPN 的版本。以下输出显示了当您使用 curl 通过 HTTP/2 连接到 Facebook 时发生的情况,其中 ALPN 和 HTTP/2 特定部分以粗体显示:

¹⁰ <https://tools.ietf.org/html/rfc7540#section-3.3>

¹¹ <https://curl.haxx.se/>

```
$ curl -vso /dev/null --http2 https://www.facebook.com * 重建 URL 为:https://
www.facebook.com/ * 尝试 31.13.76.68...

* TCP_NODELAY 设置 * 连接
到 www.facebook.com (31.13.76.68) 端口 443 (#0)
* ALPN,提供 h2 * ALPN,提供
http/1.1 * 成功设置证书验证位置:* CAfile: /
etc/pki/tls/certs/ca-bundle.crt CPath:无} [5 字节数据]

* TLSv1.2 (OUT), TLS 握手, Client hello (1): } [214 字节数据]

* TLSv1.2 (IN), TLS 握手, 服务器问候 (2): { [102 字节数据]

* TLSv1.2 (IN), TLS 握手, 证书 (11): { [3242 字节数据]

* TLSv1.2 (IN), TLS 握手, 服务器密钥交换 (12): { [148 字节数据]

* TLSv1.2 (IN), TLS 握手, 服务器完成 (14): { [4 字节数据]

* TLSv1.2 (OUT), TLS 握手, 客户端密钥交换 (16): } [70 字节数据]

* TLSv1.2 (OUT), TLS change cipher, Client hello (1): } [1 字节数据]

* TLSv1.2 (OUT), TLS 握手, 完成 (20): } [16 字节数据]

* TLSv1.2 (IN), TLS 握手, 完成 (20): { [16 字节数据]

* 使用 TLSv1.2 / ECDHE-ECDSA-AES128-GCM-SHA256 的 SSL 连接* ALPN,服务器接受使用 h2 * 服务器证
书:
* 主题:C=美国; ST=加利福尼亚; L=门洛帕克; O=脸书公司;
CN=*.facebook.com
* 开始日期:格林威治标准时间 2016 年 12 月 9 日 00:00:00
* 过期日期:格林威治标准时间 2018 年 1 月 25 日 12:00:00 *
subjectAltName:主机 “www.facebook.com”匹配证书的 “*.facebook.com”* 发行人:C=US; O=数字证书公司;
OU=www.digicert.com; CN=DigiCert SHA2 高
保证服务器 CA
* SSL 证书验证正常。
* 使用 HTTP2,服务器支持多用途* 连接状态改变 (HTTP/2 确认)
```

在这里,您可以看到客户端声明它将使用 ALPN 来声明对 HTTP/2 (h2) 和 HTTP/1.1 (http/1.1) 的支持。然后它会经历各种握手步骤 (并未显示所有细节,但足以让您了解正在发生的事情),最后,与 TLSv1.2、ECDHE-ECDSA-AES128-GCM 建立连接-SHA256 密码套件和 h2 ALPN 设置。接下来显示服务器证书, curl 切换到 HTTP/2。使用 curl 是在服务器上测试 ALPN 支持的好方法 (当然前提是您的 curl 版本支持 ALPN)。如果你很好奇,你可以使用--no-alpn 标志测试 NPN,但是这个测试没有显示那么多的信息和

排除前面示例中显示的所有 ALPN 行而不替换任何 NPN 等价物,尽管最后两行是相同的:

```
$ curl -vso /dev/null --http2 https://www.facebook.com --no-alpn
...
* SSL 证书验证正常。
* 使用 HTTP2_服务器支持多用途 * 连接状态改变 (HTTP/2 确认)
```

4.2.2 使用 HTTP 升级头

客户端可以通过发送Upgrade HTTP 标头请求将现有的 HTTP/1.1 HTTP 连接升级到 HTTP/2。此标头应仅用于未加密的 HTTP 连接 (h2c)。加密的 HTTPS HTTP/2 连接 (h2) 不应使用此方法协商 HTTP/2,并且必须使用 ALPN 作为 HTTPS 协商的一部分。正如我多次声明的那样,Web 浏览器仅在加密连接上支持 HTTP/2,因此它们不会使用此方法。那些使用外部浏览器 (例如,在 API 上)的人可能有兴趣了解有关此过程如何工作的更多详细信息。

客户端何时发送升级标头完全取决于客户端。标头可以随每个请求一起发送,仅随初始请求一起发送,或者仅当服务器已通过HTTP 响应中的升级标头通告 HTTP/2 支持时才发送。以下示例描述了Upgrade 标头的工作原理。

示例1:不成功的升级请求HTTP/1.1 请求带有升级标头,因为此客户端支持 HTTP/2,因此更喜欢使用它:

```
GET / HTTP/1.1 Host:
www.example.com Upgrade: h2c
HTTP2 -Settings: <稍后讨论>
```

这样的请求必须包含一个HTTP-Settings标头,它是 HTTP/2 设置消息的 base-64 编码,我稍后将对此进行讨论。

不理解 HTTP/2 的服务器可以正常响应 HTTP/1.1 消息,就好像没有发送升级标头一样:

```
HTTP/1.1 200 OK 日期:2017
年 6 月 25 日星期日 13:30:24 GMT 连接:保持活动内容类型:text/html
服务器:Apache
```

```
<!doctype html> <html>
<头>
··ETC。
```

示例2:成功的升级请求 不是忽略升级请求并发回

HTTP/1.1 200响应,理解 HTTP/2 的服务器可以响应HTTP/1.1 101响应,表示它将切换协议:

HTTP/1.1 101 切换协议连接:升级升级:h2c

然后服务器立即切换到 HTTP/2,发送一个SETTINGS帧 (见 4.3.3 节),然后以 HTTP/2 格式发送对原始消息的响应。

示例3:服务器建议的升级

发出 HTTP/1.1 请求,但客户端假定服务器不支持 HTTP/2,因此它不发送升级标头:

```
GET / HTTP/1.1
Host: www.example.com
```

理解 HTTP/2 的服务器可以使用200响应代码进行响应,但通过在响应的升级HTTP 标头中通告支持来告诉客户端它也支持 HTTP/2。在这种情况下,这是一个升级建议而不是升级请求,因为所有升级请求都必须从客户端发起。以下是服务器通告 h2 (HTTP/2 over HTTPS)和 h2c (HTTP/2 over HTTP)支持的示例:

```
HTTP/1.1 200 OK
Date: 2017
Content-Type: text/html
Server: Apache
Upgrade: h2c,h2
```

```
<!doctype html> <html>
<head>
...ETC.
```

客户端可以使用此信息通过在下一个请求中发送Upgrade标头来启动升级,如前两个示例所示:

```
GET /styles.css HTTP/1.1
Host: www.example.com
Upgrade: h2c
HTTP2-
Settings: <稍后讨论>
```

服务器以101响应响应并如前所述升级连接。请注意,升级标头和协商方法不能用于 h2 连接 只能用于 h2c 连接。在这里,服务器已经通告了 h2 和

h2c 连接,但如果客户端要使用 h2,则应切换到 HTTPS 并使用 ALPN 来协商此连接。

发送升级标头的问题

因为在撰写本文时所有 Web 浏览器都仅通过 HTTPS 支持 HTTP/2,所以升级选项可能永远不会被浏览器使用,这可能会导致问题。

考虑这种情况。您在连接的一侧有一个支持 HTTP/2 的 Web 浏览器,而在另一侧,您在应用程序服务器 (例如如 Tomcat) 支持 HTTP/2。在这种情况下,Web 服务器充当反向代理,并可能在客户端 (Web 浏览器) 和最终服务器 (Tomcat 应用程序服务器) 之间发送所有请求,两者都使用 HTTP/2。应用程序服务器可能会尝试提供帮助并发送升级标头以建议迁移到更好的 HTTP/2 协议。Web 服务器可能会盲目转发此标头。客户将看到此升级建议并决定升级是个好主意。但是客户端连接到的 Web 服务器不支持 HTTP/2。

在类似的场景中,Web 服务器已经在与 Web 浏览器通信 HTTP/2,但使用 HTTP/1.1 代理请求到后端应用程序服务器。应用服务器可能会发送升级建议,如果这个建议被转发给浏览器,它可能会感到困惑,因为建议是将 HTTP/2 连接升级到它已经在使用的 h2。

这些问题不是理论上的;在推出 HTTP/2 时,它们造成了真正的问题。当 Safari 在 HTTP/2 连接上看到 h2 升级标头时,它通常会返回错误 (请参阅第 3 章)。

在撰写本文时,nginx 团队已被要求停止盲目地传递 Upgrade 标头,例如,当它位于通过此标头宣传 HTTP/2 支持的 Apache 服务器前面时。可以通过一些配置 (`proxy_hide_header Upgrade`) 来移除这个 header,但是很少有人知道添加它直到遇到问题。此外,某些客户端或服务器可能无法正确实现升级标头。在尝试使用 HTTP/2 时,我注意到在 Apache 开始发布升级标头后 NodeJS 断开连接的问题。¹²这个问题已得到修复,但仍然存在于仍在使用的旧版本的 NodeJS 中。

虽然这些问题很可能在本书出版时得到解决,但毫无疑问,类似的问题还会出现。总而言之,我更喜欢服务器实现不宣传升级选项 (至少,默认情况下)。在我看来,这个选项不会用得太多,而且会导致比它解决的问题更多的问题。对于大多数实现 (以及所有浏览器),更有可能使用 HTTPS 协商。如果不支持或不需要 HTTPS,则可以将先验假设方法用于后端服务器。Apache 是主要违规者之一,已被要求停止

¹² <https://trac.nginx.org/nginx/ticket/915>

¹³ <https://github.com/nodejs/node/issues/4334>

默认情况下包括Upgrade标头¹⁴,但与此同时,您可以使用以下mod_headers Apache 配置来关闭此标头的发送,我建议您在运行支持 HTTP/2 的 Apache 时这样做 (尽管此解决方案可能会导致其他需要使用升级标头的协议出现问题,例如 WebSockets,因此不能在这些场景中使用) :

标头未设置升级

4.2.3 使用先验知识

HTTP/2 规范声明客户端可以使用 HTTP/2 的第三种也是最后一种方式是它是否已经知道服务器理解 HTTP/2。在这种情况下,它可以立即开始使用 HTTP/2,避免任何升级请求。

客户端如何预先知道服务器能够理解 HTTP/2 可以通过不同的方法。如果您正在运行反向代理来卸载 HTTPS,您可能希望通过 HTTP (h2c) 将 HTTP/2 与您的后端服务器通信,因为您知道它们使用 HTTP/2。或者,可以根据Alt-Svc标头 (HTTP/1.1) 或ALTSVC框架 (参见第 4.2.4 节)公布的替代支持来假设先验知识。

这个选项是风险最大的一个,因为它做出了服务器可以使用 HTTP/2 的某些假设。使用先验知识的客户必须注意适当地处理任何拒绝消息,以防先验知识被证明是不正确的。

服务器对 HTTP/2 前言消息的响应 (我将在本章后面讨论)在这种选择使用 HTTP/2 的方式中非常重要。仅当您同时控制客户端和服务器时才应使用此方法。

4.2.4 HTTP 替代服务

第四种方式未包含在原始 HTTP/2 规范中,是使用 HTTP Alter native Services,¹⁵在 HTTP/2 发布后作为单独的标准添加。该标准允许服务器使用 HTTP/1.1 (通过Alt-Svc HTTP 标头)通知客户端所请求的资源在使用不同协议的另一个位置 (例如另一个服务器或端口)可用。该协议可用于在具有先验知识的情况下启动 HTTP/2。

替代服务不仅适用于 HTTP/1,还可以通过现有的 HTTP/2 连接进行通信 (通过新的ALTSVC框架,在本章后面介绍),以防客户端想要切换到不同的连接 (一个位于附近例如,给客户,或者不那么忙的客户)。该标准相当新,并未广泛使用。它仍然会导致在一个连接上启动然后切换,这比通过 ALPN 或先验知识启动 HTTP/2 慢。它介绍了一些超出本书范围的有趣可能性,但至少有一个内容分发网络似乎打算充分利用它。¹⁶

¹⁴ https://bz.apache.org/bugzilla/show_bug.cgi?id=59311

¹⁵ <https://tools.ietf.org/html/rfc7838>

¹⁶ blog.cloudflare.com/cloudflare-onion-service/

4.2.5 HTTP/2 前言消息

必须在 HTTP/2 连接上发送的第一条消息（无论使用哪种方法来建立 HTTP/2 支持）是 HTTP/2 连接前言或“魔术”字符串。此消息由客户端作为 HTTP/2 连接上的第一条消息发送。此消息是 24 个八位字节的序列，在十六进制表示法中如下所示：

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

此序列转换为 ASCII 格式的以下消息：

```
PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
```

发送此消息似乎很奇怪，但并非巧合，它几乎是一个 HTTP/1 风格的消息：

```
PRI * HTTP/2.0
```

```
SM
```

即 HTTP 方法为 PRI（而不是 GET 或 POST），资源为 *，HTTP 版本号为 HTTP/2.0。接下来是双返回（所以没有请求头），然后是 SM 的请求主体。

这个无意义的、类似于 HTTP/1 的消息的目的是针对客户端试图与不理解 HTTP/2 的服务器通信 HTTP/2 的情况。这样的服务器尝试像解析任何其他 HTTP 消息一样解析此消息，但失败了，因为它无法识别废话方法(PRI)或 HTTP 版本(HTTP/2.0)，并且应该拒绝该消息。请注意，此前言消息是官方规范中唯一仍然使用次版本号引用 HTTP/2.0 的部分；正如“HTTP/2.0 还是 HTTP/2？”中所讨论的，其他地方都是 HTTP/2。4.1 节中的边栏。服务器知道客户端根据传入消息使用 HTTP/2，因此不会发送此神奇消息；它必须发送一个 SETTINGS 帧作为它的第一条消息（可以是空的）。

为什么是 PRI 和 SM？

HTTP/2 规范早期草案中的原始 HTTP/2 前言使用 FOO 和 BARa 或 BAb，这是编程中众所周知的占位符名称。但在规范草案的第 4 版中，此占位符更改为 PRI 和 SM，规范中未说明原因。

这一变化显然是为了回应爱德华斯诺登在这段时间揭露的关于用于收集来自不同公司的互联网流量的 PRISM 程序的揭露。这些揭露令自由互联网的支持者感到不安（其中一些人还帮助决定互联网的标准），他们认为在开始每个 HTTP/2 连接时都带有一点提醒会很幽默。

为什么是 PRI 和 SM? (继续)

这条消息的内容并不重要;该消息原本是无意义的消息,不应被识别为有效的 HTTP。其他建议包括 START,但最终,PRI SM 在标记为“行使关于魔法的编辑自由裁量权”的变更提交中将其纳入最终规范。

- ^a <https://tools.ietf.org/html/draft-ietf-httpbis-http2-02#section-3.2>
- ^b <https://tools.ietf.org/html/draft-ietf-httpbis-http2-03#section-3.2>
- ^c <https://tools.ietf.org/html/draft-ietf-httpbis-http2-04#section-3.5>
- ^d <http://blog.jgc.org/2015/11/the-secret-message-hidden-in-every.html>
- ^e <https://github.com/http2/http2-spec/commit/ac468f3fab9f7092a430eedfd69ee1fb2e23c944>

4.3 HTTP/2 帧

设置好 HTTP/2 连接后,就可以开始发送 HTTP/2 消息了。如您所见,HTTP/2 消息由在单个多路复用连接上的流上发送的数据帧组成。框架是许多 Web 开发人员不需要了解的低级概念,但了解技术的构建块总是值得的。通过在帧级别查看 HTTP/2,第 3 章末尾的许多错误更容易调试,因此查看帧级别具有实际和理论用途。我通过一个真实世界的例子来解释 HTTP/2 的主要部分。

在本节中,我查看并解释了框架类型,这些框架类型起初看起来有点令人生畏和困惑,但需要吸收很多内容。我鼓励读者不要过多地关注第一次阅读,而是要理解框架的整体概念 HTTP/2 框架,并对每种框架类型有一个高层次的理解。关于各个框架和每个框架的设置的部分可以作为以后的参考,HTTP/2 规范本身也可以,但是你不需要记住它们来理解本书的其余部分或真正的 HTTP/2 世界。

4.3.1 查看 HTTP/2 帧

有一些工具可用于查看 HTTP/2 帧,包括 Chrome 的网络导出页面、nghttp 和 Wireshark。您的 Web 服务器也可以增加日志记录以显示单个帧,但是对于潜在的大量用户,该技术很快就会变得混乱,因此前面的工具更容易使用,除非您试图调试您的潜在问题网络服务器。

铬净出口

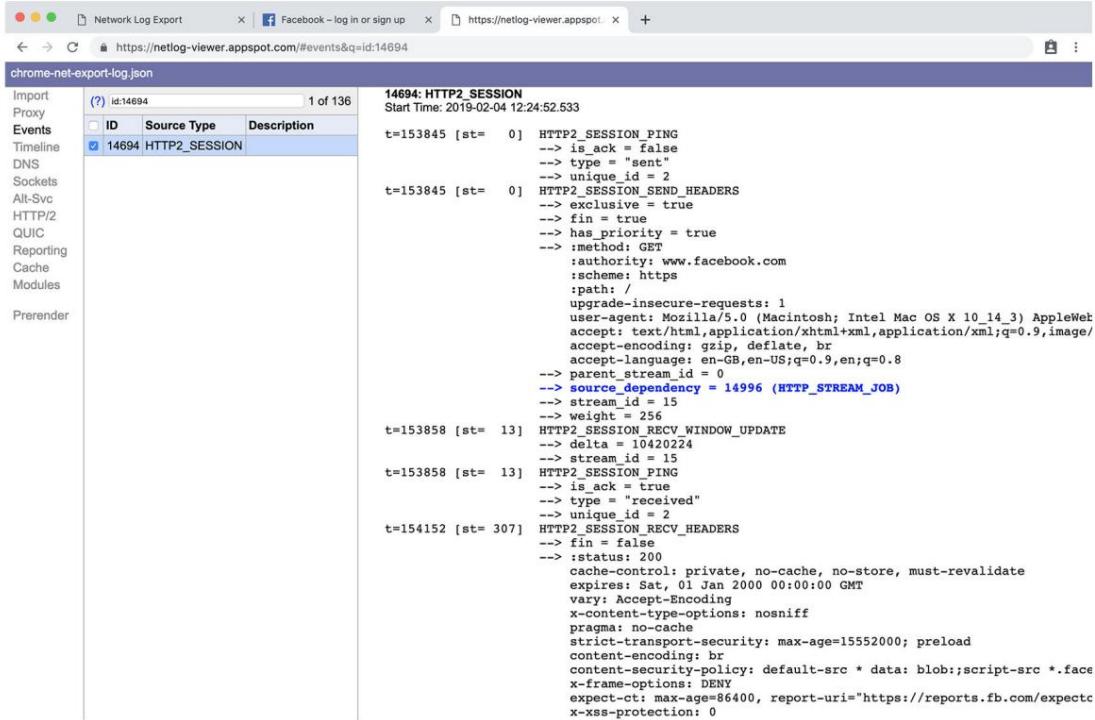
在不安装额外软件的情况下查看 HTTP/2 帧的最简单方法是使用 Chrome 的网络导出页面。这曾经在 net-internals 页面中可用,但从 Chrome 71 开始,由于多种原因,它移到了 net-exernals¹⁷,并且

¹⁷ <https://docs.google.com/document/d/1Ll7T5cguj5m2DqkUTad5DWRCqtbQ3L1q9FRvTN5-Y28/>

需要更多的努力才能查看。打开 Chrome 浏览器,然后在 URL 栏中键入以下内容:

铭: //网络出口/

单击“开始记录到磁盘”并为日志文件选择一个文件位置。在另一个选项卡中,打开一个 HTTP/2 站点 (例如<https://www.facebook.com>) ,加载后单击“停止记录”。此时您可以使用 NetLog 查看器 (<https://netlog-viewer.appspot.com>) 打开并检查创建的日志文件 (注意:此工具仅在本地查看文件,不会将其上传到服务器)。单击左侧的 HTTP/2 选项,然后单击站点 (例如 www.facebook.com) ,您应该会看到如图 4.7 所示的底层 HTTP/2 消息。



The screenshot shows the Network Log Export interface in Chrome DevTools. On the left, there's a sidebar with options like Import, Proxy, Events, Timeline, DNS, Sockets, Alt-Svc, HTTP/2, QUIC, Reporting, Cache, Modules, and Prerender. The main area has a table titled "chrome-net-export-log.json" with columns: ID, Source Type, and Description. One row is selected, showing "ID: 14694" and "Source Type: HTTP2_SESSION". The "Description" column contains detailed log entries for an HTTP/2 session, including timestamps, stream IDs, and various HTTP headers and settings. The log entries show the initial session setup, header exchange, window update, and a SETTINGS frame being sent.

```

14694: HTTP2_SESSION
Start Time: 2019-02-04 12:24:52.533

t=153845 [st= 0] HTTP2_SESSION_PING
--> is_ack = false
--> type = "sent"
--> unique_id = 2

t=153845 [st= 0] HTTP2_SESSION_SEND_HEADERS
--> exclusive = true
--> fin = true
--> has_priority = true
--> :method: GET
--> :authority: www.facebook.com
--> :scheme: https
--> :path: /
--> upgrade-insecure-requests: 1
--> user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36
--> accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
--> accept-encoding: gzip, deflate, br
--> accept-language: en-GB,en-US;q=0.9,en;q=0.8
--> parent_stream_id = 0
--> source_dependency = 14996 (HTTP_STREAM_JOB)
--> stream_id = 15
--> weight = 256

t=153858 [st= 13] HTTP2_SESSION_RECV_WINDOW_UPDATE
--> delta = 1042024
--> stream_id = 15

t=153858 [st= 13] HTTP2_SESSION_PING
--> is_ack = true
--> type = "received"
--> unique_id = 2

t=154152 [st= 307] HTTP2_SESSION_RECV_HEADERS
--> fin = false
--> status: 200
--> cache-control: private, no-cache, no-store, must-revalidate
--> expires: Sat, 01 Jan 2000 00:00:00 GMT
--> vary: Accept-Encoding
--> x-content-type-options: nosniff
--> pragma: no-cache
--> strict-transport-security: max-age=15552000; preload
--> content-encoding: br
--> content-security-policy: default-src * data: blob:;script-src *.facebook.net
--> expect-ct: max-age=86400, report-uri="https://reports.fb.com/expectct"
--> x-xss-protection: 0

```

图 4.7 在 Chrome 中查看 HTTP/2 帧

Chrome 向这个屏幕添加了很多自己的细节,并且经常将帧拆分成多行。以下输出来自一个 SETTINGS 框架:

```
t= 1646 [st= t= 1647 [st=
1]      HTTP2_SESSION_RECV_SETTINGS
2]      HTTP2_SESSION_RECV_SETTING --> id =
1 (SETTINGS_HEADER_TABLE_SIZE) --> 值 = 4096
```

```
t= 1647 [st=          2]      HTTP2_SESSION_RECV_SETTING --> id =
                                5 (SETTINGS_MAX_FRAME_SIZE) --> 值 = 16384

t= 1647 [st=          2]      HTTP2_SESSION_RECV_SETTING --> id =
                                6 (SETTINGS_MAX_HEADER_LIST_SIZE) --> value = 131072

t= 1647 [st=          2]      HTTP2_SESSION_RECV_SETTING --> id =
                                3 (SETTINGS_MAX_CONCURRENT_STREAMS) --> 值 = 100

t= 1647 [st=          2]      HTTP2_SESSION_RECV_SETTING --> id =
                                4 (SETTINGS_INITIAL_WINDOW_SIZE) --> 值 = 65536
```

与使用其他两个工具相比,使用此 Chrome 屏幕时阅读各个帧可能会有点困难,但该屏幕包含大部分相同的信息。另一方面,无需安装其他工具就可以在浏览器中获得这种级别的详细信息,而且您可以使用各种工具更好地格式化输出。¹⁸在撰写本文时,我不知道其他浏览器显示这种细节程度,尽管与 Chrome 具有相同代码库的 Opera 具有相似的功能。

使用 NGHTTP

nghttp 是在 nghttp2 C 库之上开发的命令行工具,许多 Web 服务器和客户端使用它来处理底层的 HTTP/2 复杂性。如果您为您的服务器安装了 nghttp2 库(例如,Apache 需要 nghttp2 库),您可能已经安装了这个工具。您可以使用它以类似于 Chrome 网络导出工具的方式查看 HTTP/2 消息,但我发现输出更清晰:

```
$ nghttp -v https://www.facebook.com [0.042] Connected 协
商协议:h2 [0.109] 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] [0.109] recv
WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0> (window_size_increment=10420225) [0.109] 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]
…ETC.
```

使用WIRESHARK

Wireshark¹⁹允许您嗅探计算机发送和接收的所有流量。当您看到发送和接收的原始消息时,这个工具可以方便地进行一些核心低级调试。不幸的是,它使用起来也相当复杂!

¹⁸ <https://github.com/rmurphrey/chrome-http2-log-parser>

¹⁹ <https://www.wireshark.org/>

复杂性之一是 Wireshark 不是客户端这一事实；它嗅探从您的浏览器发送到服务器的流量。但是，所有浏览器都在 HTTPS 上使用 HTTP/2，因此除非您知道用于加密和解密这些消息的 SSL/TLS 密钥，否则您将无法读取流量，而这正是 HTTPS 的意义所在。Chrome 和 Firefox 开发人员想到了这个用例，这些浏览器允许您将 HTTPS 密钥保存到单独的文件中，以便您可以使用 Wireshark 等工具进行调试。显然，当您使用它进行调试时，您应该关闭该工具。您所要做的就是通过设置 SSLKEYLOGFILE 环境文件或通过在命令行中传递以下代码来启动 Chrome 来告诉 Chrome 或 Firefox 用于保存密钥的文件：

```
C:\Program Files (x86)\Google\Chrome\Application\chrome.exe --ssl-key-log file=%USERPROFILE%\sslkey.log
```

注意确保使用正确的连字符。许多应用程序，例如 Microsoft Office 中的应用程序，喜欢自动将短连字符 (-) 更改为短破折号 (-) 或长破折号 ()，这是三个单独的字符，命令行不会将其识别为传递参数。结果是一个空文件，从未向其中添加 SSL 密钥，并且充满了混乱和沮丧。

对于 macOS，设置 SSLKEYLOGFILE 环境变量：

```
$ export SSLKEYLOGFILE=~/sslkey.log $ /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome
```

或直接将其作为命令行参数提供：

```
$ /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome--ssl-key-log-file=/Users/barry/sslkey.log
```

接下来，启动 Wireshark 并通过选择 Edit > Preferences > Protocols > SSL 并设置 (Pre)-Master-Secret 日志文件名来设置相同的文件位置，如图 4.8 所示。

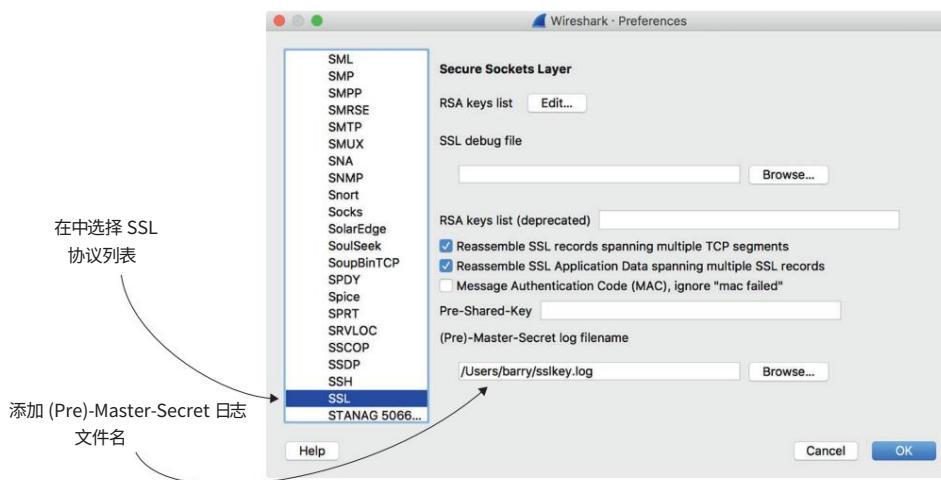


图 4.8 设置 Wireshark HTTPS 密钥文件

此时,您应该能够读取 Chrome 使用的所有 HTTPS 数据,因此如果您访问<https://www.facebook.com>并在 http2 上的 Wireshark 中进行过滤,您应该能够看到消息,包括前言 4.1.5 节讨论的消息(在 Wireshark 中可用),如图 4.9 所示。

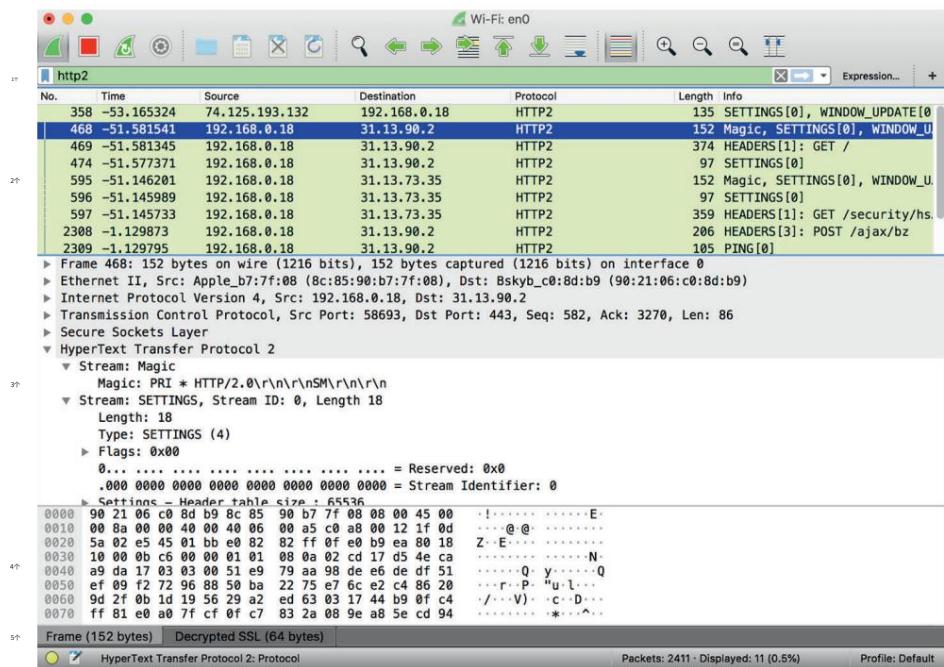


图 4.9 Wireshark 中“神奇”的 HTTP/2 前言消息

图 4.9 中发生了很多事情,所以如果您不习惯使用 Wireshark,它可能会让人望而生畏。以下列表描述了标有 1-5 的各个部分:

1 过滤器视图允许您键入各种过滤器选项。在这里,我过滤了 http2 消息。如果您打开了许多 HTTP/2 连接,您可能需要使用更具体的过滤器,包括您所连接的服务器的 IP 地址。

以下过滤器仅显示发送到 IP 地址 31.13.90.2 和从 IP 地址 31.13.90.2(我在本例中连接到的 Facebook 服务器;从浏览器的开发人员工具获取)发送的 HTTP/2 消息:

```
http2 && (ip.dst==31.13.90.2 || ip.src==31.13.90.2)
```

2 接下来是符合您的过滤器的邮件列表。如果单击这些消息,您将获得更多详细信息。

3 此部分显示消息的完整详细信息。如果 Wireshark 识别该协议(我还没有找到它不能识别的协议!),它会以一种易于阅读的格式显示消息适用的每个协议。

这个例子,从上到下阅读,从 Wireshark 自己的基本格式开始,它称之为框架(不要与 HTTP/2 框架混淆)。基本帧作为以太网消息发送,这些消息被封装到 IPv4 消息中,通过 TCP 发送,通过 SSL/TLS 发送,最后,您会看到 HTTP/2 消息。Wireshark 允许您查看任何这些级别的消息。在屏幕截图中,我展开了 HTTP/2 部分,然后进一步展开了“神奇的”HTTP/2 序言消息,但如果您对查看以太网、IP、TCP 上的消息更感兴趣,您可以类似地展开其他级别,或 SSL/TLS 级别。

4靠近底部的部分显示原始数据,通常以十六进制和 ASCII 格式。

5原始数据底部的选项卡允许您决定显示数据的原始程度。您几乎肯定只会对 mat 的解压缩标头(或魔术消息的解密 SSL,它没有任何压缩标头)感兴趣,而不是原始帧格式。

您甚至可以使用 Wireshark 查看 HTTPS 协商消息(包括 ClientHello 消息中的 ALPN 扩展请求和 ServerHello 消息中的响应)。在图 4.10 中,箭头表示客户端首选 h2,然后是 http/1.1。

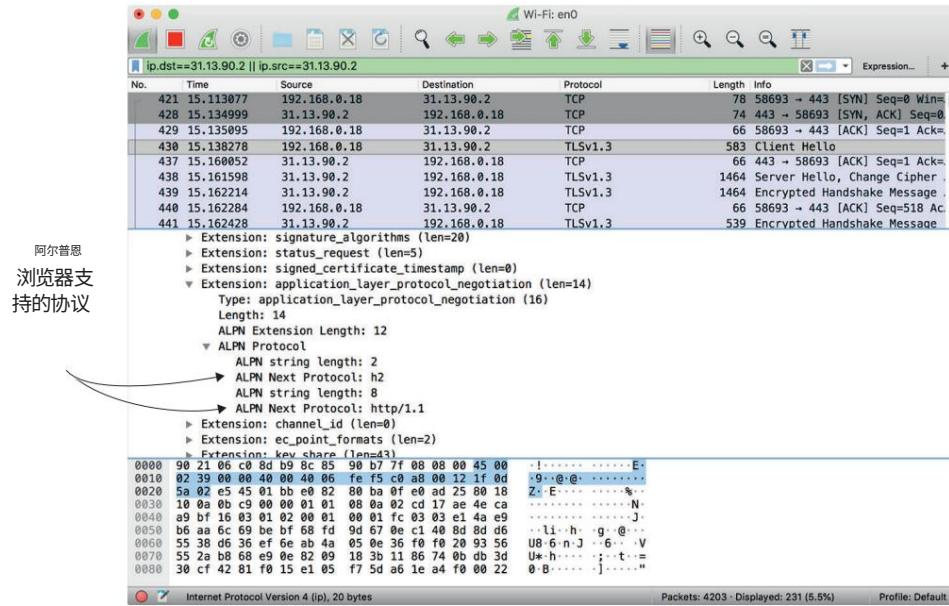


图 4.10 ALPN 扩展作为 Wireshark 中 ClientHello 消息的一部分

让 Wireshark 解密流量时遇到问题？

不幸的是，至少可以说，使用 Wireshark 解密 HTTPS 流量可能有点不稳定，因此您可能需要多次尝试才能正确处理。

原因之一是此过程仅适用于具有完整 TLS 握手的新 HTTPS 会话。问题是，如果您再次连接到同一个站点，该站点可能会重用一些旧的加密设置，并且可能只完成了部分握手不足以让 Wireshark 解密流量。要查看 Wireshark 中是否正在使用 HTTPS 会话恢复，请过滤ssl，而不是http2。查看第一条 ClientHello消息以查看 Session ID或SessionTicket TLS是否为非零。如果是这样，您将重新加入旧会话，并且 Wireshark 将无法解密消息（除非它在原始会话建立后运行）。

更糟糕的是，没有一个浏览器提供可靠的方法来删除 SSL/TLS 会话密钥/会话票证并强制执行完整的握手。已提出一些将此功能添加到 Chrome 和 Firefox 的请求，但它们已经开放了一段时间。

我发现 Wireshark 通常更难在 macOS 中工作。例如，最新版本的 Firefox 似乎不再一直记录 SSL 密钥信息。

我能提供的最佳建议是确保您运行的是最新版本的 Wireshark。

休息一下并在会话过期时返回似乎也有帮助，但并不令人满意。或者，您可以尝试使用其他浏览器，该浏览器不应存储以前的会话。

-
- a <https://bugs.chromium.org/p/chromium/issues/detail?id=90454>
 - b https://bugzilla.mozilla.org/show_bug.cgi?id=285440

使用哪个工具

使用您觉得最舒服的工具或您喜欢的替代工具。

Wireshark 提供了最详细的信息，因此如果您想很好地了解消息结构和格式，它是无与伦比的。但在许多情况下，这种详细程度可能过于详细。

此外，设置它很复杂。除非您熟悉 Wireshark，否则最好使用其他两种工具之一。

对于本章的其余部分，我在示例中使用 nghttp，因为对于本书而言，它是最容易捕获和格式化的工具。无论您使用什么工具，消息都应该是相似的，尽管它们显示数据的顺序或某些设置可能有所不同。

这些工具可用于低级调试或查看协议的详细信息，但大多数人不需要每天使用它们。当然，大多数开发人员将从浏览器中的标准开发人员工具中获得足够的帮助，他们不需要深入研究较低级别的网络导出或 Wireshark 的详细级别。然而，这三者都是您随身携带的好工具，因为它们可以帮助您巩固对协议的理解。

4.3.2 HTTP/2 帧格式

在开始查看一些示例帧之前,了解 HTTP/2 帧的构成可能会有所帮助。每个 HTTP/2 帧都由固定长度的标头 (详见表 4.1)和后跟有效负载组成。

表 4.1 HTTP/2 帧头格式

场地	长度	描述
长度	24位	帧的长度,不包括此表中详述的所有标头字段,最大大小为224 – 1 个八位字节;受限于 SETTINGS_MAX_FRAME_SIZE,默认为较小的214个八位字节
类型	8位	目前,已经定义了 14 种帧类型: a 数据 (0x0) 标题 (0x1) 优先级 (0x2) RST_STREAM (0x3) 设置 (0x4) PUSH_PROMISE (0x5) PING (0x6) GOAWAY (0x7) WINDOW_UPDATE (0x8) CONTINUATION (0x9) ALTSVC (0xa),通过RFC 7838b添加 (0xb),目前未使用,但在pastc中使用 ORIGIN (0xc),通过RFC 8336d添加 CACHE_DIGEST,proposede
旗帜	8位	帧特定标志
保留位	1位	当前未使用,必须设置为 0
溪流 标识符	31位	标识帧的无符号 31 字节整数

^a <https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml>

^b <https://tools.ietf.org/html/rfc7838> <https://github.com/httplib/httplib/pull/323>

^c <https://tools.ietf.org/html/rfc8336> https://datatracker.ietf.org/doc/draft-ietf-httpbis-cache-digest/?include_text=1

^d *

框架被如此明确定义的事实使 HTTP/2 成为二进制协议。HTTP/2 帧不同于可变长度的 HTTP/1 文本消息,后者必须通过扫描换行符和空格来解析。这是一个低效且容易出错的过程。更严格、定义明确的 HTTP/2 帧格式允许更容易的解析和更小的消息,因为可以使用特定的代码 (例如0x01用于 HEADERS帧类型而不是完整的措辞)。

八位字节与字节与许多协

议定义一样,HTTP/2 规范使用八位字节而不是模糊字节。一个八位字节恰好是 8 位,而一个字节通常被理解为 8 位,这取决于所使用的系统架构。

长度字段 (我希望)是不言自明的。以下部分将详细介绍每种消息类型。 Flags字段是特定于帧的,并用每种帧类型进行描述。不使用保留位字段电流。流标识符字段也应该是不言自明的。显然,将此字段限制为 31 位的原因之一是为了 Java 互操作性,因为它没有 32 位无符号整数。²⁰

标志的含义和有效载荷的构成取决于帧类型。 HTTP/2 被编写为可扩展的。最初的 HTTP/2 规范²¹只定义了帧类型 0-9,但现在又增加了三种,而且将来无疑会增加更多。

4.3.3 通过示例检查 HTTP/2 消息流

了解框架的最简单方法是查看它们在现实世界中的使用。例如,使用 nghttp 连接到www.facebook.com (支持 HTTP/2 的众多网站之一)。

输出如下:

```
$ nghttp -va https://www.facebook.com | more [ 0.043] Connected 协商协议:h2
[ 0.107] 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] [0.107] recv WINDOW_UPDATE
frame <length=4, flags=0x00, stream_id=0> (window_size_increment=10420225) [0.107] 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]
[0.107]发送设置帧<长度=0,标志=0x01,stream_id=0>
;确认
(niv=0)
[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=3>
(dep_stream_id=0, weight=201, exclusive=0)
[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=5>
(dep_stream_id=0, weight=101, exclusive=0)
```

²⁰ <https://stackoverflow.com/questions/39309442/why-is-the-stream-identifier-31-bit-in-http-2-and-why-is-it-preceded-with-a-rese> <https://tools.ietf.org/html/rfc7540>

```
[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=7> (dep_stream_id=0,权重=1,独占=0)[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=9> (dep_stream_id=7, weight=1, exclusive=0) [0.107] 发送优先帧 <length=5, flags=0x00, stream_id=11> (dep_stream_id=3, weight=1, exclusive=0) [ 0.107] send标头帧 <length=43, flags=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: nghttp2/1.28.0 [ 0.138]
recv SETTINGS frame <length=0,标志=0x01, stream_id=0>

;确认
(niv=0)
[0.138]recv WINDOW_UPDATE帧<长度=4,标志=0x00,stream_id=13> (window_size_increment=10420224)[0.257]recv
(stream_id=13) :状态:200[0.257]recv (stream_id=13)x-xss-保护:0 [0.257] recv (stream_id=13) pragma: no-cache [0.257] recv (stream_id=13) cache-control: private, no-cache, no-store, must-revalidate
```

```
[ 0.257] recv (stream_id=13) x-frame-options: DENY [ 0.257] recv (stream_id=13) strict-
transport-security: max-age=15552000; preload [0.257] recv (stream_id=13) x-content-type-options: nosniff [0.257] recv (stream_id=13)
到期时间:星期六,2000年1月1日00:00:00GMT [0.257] recv (stream_id=13)设置 cookie:fr=0m7urZrTka6WQuSGa..BaQ42y.61.A
AA.0.0.BaQ42y.AWXRqgzE;过期=2018年3月27日星期二12:10:26GMT;最大年龄=7776000;路径=/;域名=.facebook.com;安全的;httponly
[0.257] recv (stream_id=13) set-cookie: sb=so1DWrDge9flkTZ7e-iSS2To; expi res=2019年12月27日星期五12:10:26GMT;最大年龄
=63072000;路径=/;域名=.facebook.com;安全的; httponly [ 0.257] recv (stream_id=13) vary: Accept-Encoding [ 0.257] recv (stream_id=13)
content-encoding: gzip [ 0.257] recv (stream_id=13) content-type: text/html; CHARSET = UTF-8 [0.257] RECV (stream_id = 13)X-FB-DEBUG:
YRE7EQV05DKXF8R1+i4VLIZMUNINV1+AP DYG7HCW6T7NCETGKIIRQJADQJADLWJ87HMHMHK6Z/n3o2121212222.212 ===== 26
GMT [0.257] recv HEADERS frame <length=517, flags=0x04, stream_id=13>; END_HEADERS (padlen=0)
```

```
;第一个响应头
<!DOCTYPE html>
<html lang=" zh-cn " id=" facebook " class=" no_js " ><head><meta
charset=" utf-8 " />
...ETC.
[0.243]接收数据帧<长度=1122,标志=0x00,stream_id=13>
....
```

```
[0.243]接收数据帧<长度=2589,标志=0x00,stream_id=13>
...
[0.264]接收数据帧<长度=13707,标志=0x00,stream_id=13>
...
[0.267]发送WINDOW_UPDATE帧<长度=4,标志=0x00,stream_id=0> (window_size_increment=33706)[0.267]发送
WINDOW_UPDATE帧<长度=4,flags=0x00,stream_id=13> (window_size_increment=33706)

...
[416.688] 接收数据帧 <length=8920, flags=0x01, stream_id=13>; END_STREAM [417.226] 发送 GOAWAY 帧
<length=8, flags=0x00, stream_id=0>

(last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

我展示了一些DATA帧,但删除了大部分,用……等替换了它们。文本。

您还可以传递-n标志来隐藏数据并仅显示帧头：

```
$ nghttp -nv https://www.facebook.com |更多的
```

即使数据被截断,这段代码看起来也很复杂,所以我将逐步引导您完成它。

首先,您通过 HTTPS (h2) 连接并协商 HTTP/2。 nghttp 不输出 HTTPS 设置或 HTTP/2 前言/魔术消息,因此您收到
SETTINGS 框架优先:

```
$ nghttp -v https://www.facebook.com | more [ 0.043] Connected 协
商协议:h2 [ 0.107] 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]
```

设置框

SETTINGS 帧(0x4)是必须由服务器和客户端发送的第一个帧 (在 HTTP/2 前言/魔术消息之后)。帧由一个空的有效载荷或几个字段/值对组成,如表 4.2 所示。

SETTINGS 帧只定义了一个可以在公共帧头中设置的标志: ACK (0x1)。如果 HTTP/2 连接的这一端正在通告设置,则将标志设置为0 ;将其设置为1以确认已由 HTTP/2 连接的另一端发送的设置。如果是确认 (标志设置为1) ,则不应在有效负载中设置其他设置。

表 4.2 HEADERS帧格式

场地	长度	描述								
标识符 16 位		<p>规范中定义了六个设置,最近又添加了两个 (将来可能会添加更多)。提议的设置没有正式标准化:</p> <ul style="list-style-type: none"> SETTINGS_HEADER_TABLE_SIZE (0x1) SETTINGS_ENABLE_PUSH (0x2) SETTINGS_MAX_CONCURRENT_STREAMS (0x3) SETTINGS_INITIAL_WINDOW_SIZE (0x4) SETTINGS_MAX_FRAME_SIZE (0x5) SETTINGS_MAX_HEADER_LIST_SIZE (0x6) SETTINGS_ACCEPT_CACHE_DIGEST (0x7)a SETTINGS_ENABLE_CONNECT_PROTOCOL (0x8)b <p>注意:SETTINGS_ACCEPT_CACHE_DIGEST 是提议的设置,尚未正式标准化,可能会发生变化。</p>								
价值	32位	<p>该字段是设置的值。请注意,如果未定义设置,则使用默认值。建议的设置尚未正式标准化:</p> <table border="0"> <tr> <td>4096 个八位字节</td> </tr> <tr> <td>1</td> </tr> <tr> <td>无限制</td> </tr> <tr> <td>65,535 个八位字节</td> </tr> <tr> <td>16,384 个八位字节</td> </tr> <tr> <td>没有限制</td> </tr> <tr> <td>0 - 否</td> </tr> <tr> <td>0 - 否</td> </tr> </table> <p>注意:SETTINGS_ACCEPT_CACHE_DIGEST 是一个提议的设置,尚未正式标准化并且可能会更改,包括其默认值。</p>	4096 个八位字节	1	无限制	65,535 个八位字节	16,384 个八位字节	没有限制	0 - 否	0 - 否
4096 个八位字节										
1										
无限制										
65,535 个八位字节										
16,384 个八位字节										
没有限制										
0 - 否										
0 - 否										

a <https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest> https://

b tools.ietf.org/html/rfc8441

记住这些知识,再看看第一条消息:

```
[0.107]接收设置帧<长度=30,标志=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]
```

接收到的SETTINGS帧有一个 30 个八位字节长的有效载荷,没有设置标志 (因此不是确认帧),并使用流 ID 0。流 ID 0 保留用于控制消息 (SETTINGS 和 WINDOW_UPDATE 帧),因此它是正确的服务器使用流 0 发送此设置帧。

接下来你得到设置本身，在这个例子中有5个（niv=5），每个都是16位（标识符）+ 32位（值）。此示例总共为 48 位或 6 个八位字节，构成了标头中给出的 30 个八位字节长度（5 个标头 x 6 个八位字节 = 30 个八位字节）。到目前为止，一切都很好。现在查看发送的各个设置：

- 1 Facebook 使用 4,096 个八位字节的 SETTINGS_HEADER_TABLE_SIZE。此设置用于我在第 8 章中讨论的 HPACK HTTP 标头压缩，因此暂时忽略它。
- 2 Facebook 还使用 16,384 个八位字节的 SETTINGS_MAX_FRAME_SIZE，因此您的客户端（nghttp）不得在此连接上发送任何更大的有效负载。
- 3 接下来 Facebook 将 SETTINGS_MAX_HEADER_LIST_SIZE 设置为 131,072 个八位字节，因此您不能发送任何大于该值的未压缩标头。
- 4 Facebook 将 SETTINGS_MAX_CONCURRENT_STREAMS 设置为 100 个流。第 2 章展示了其中一个示例，其中尝试从具有 100 个流限制的服务器加载超过 100 个图像。在那种情况下，请求被排队等待空闲流，类似于 HTTP/1 中请求排队的方式，但连接限制低于大多数浏览器使用的六个连接。HTTP/2 显着增加了可以执行的并行请求的数量，但数量通常受服务器限制（通常为 100 或 128 个流）而不是无限（默认）。

- 5 最后，Facebook 将 SETTINGS_INITIAL_WINDOW_SIZE 设置为 65,536 个八位字节。此设置用于流量控制，我将在第 7 章中介绍。

这个看似简单的框架中有几件事值得注意。首先，设置可以按任何顺序排列，例如 SETTINGS_MAX_CONCURRENT_STREAMS，它在规范中定义为设置 3 (0x03)，但在 SETTINGS_MAX_HEADER_LIST_SIZE 之后给出，即设置 6 (0x06)。此外，许多设置都使用默认的初始值，因此服务器可以发送这个简化的 SETTINGS 帧以获得相同的效果，只需要三个设置：

```
[0.107]接收设置帧<长度=18,标志=0x00,stream_id=0>
(niv=3)
[SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
[SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
[SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
```

但是，更明确地说明您要使用的值并没有什么害处。

此示例显示 Facebook 使用的 SETTINGS_INITIAL_WINDOW_SIZE 比默认值（65,535 个八位字节）大 1 个八位字节，这似乎很奇怪，因为几乎不值得更改默认值。

最后，请注意 Facebook 服务器未设置 SETTINGS_ENABLE_PUSH。此设置旨在让服务器推送（如果规范作者决定将其用于此目的）到客户端，因此供客户端使用。服务器设置此设置没有意义，尽管我猜它可以用来宣传此服务器上是否可能支持推送。

如果客户端不支持HTTP/2 推送或不想启用它,则关闭此设置。

回到示例,我将暂时跳过WINDOW_UPDATE框架,转而查看接下来的三个SETTINGS框架:

```
[0.107] recv设置帧<长度=30,标志=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]
[0.107]发送设置帧<长度=12,标志=0x00,stream_id=0> (niv=2)

[SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
[SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[0.107]发送设置帧<长度=0,标志=0x01, stream_id=0>
;确认
(niv=0)
...
[0.138] recv设置帧<长度=0,标志=0x01, stream_id=0>
;确认
(niv=0)
```

nghhttp 接收初始服务器SETTINGS帧 (已经讨论过) ,然后客户端发送带有几个设置的SETTINGS帧。接下来,客户端确认服务器的SETTINGS帧。确认设置帧是一个简单的帧,设置了ACK (0x01)标志,长度为0 ,因此设置为0 (niv=0)。再往下一点是服务器以相同的简单格式确认客户端的SETTINGS帧。

此示例显示存在一段时间,在此期间一方已发送SETTINGS帧但未收到任何确认。在此期间,无法使用这些非默认设置。但是因为所有 HTTP/2 实现都必须能够处理默认值,并且因为必须先发送SETTINGS帧,所以这种情况应该不会造成问题。

WINDOW_UPDATE框架

服务器还发送了一个WINDOW_UPDATE帧:

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

WINDOW_UPDATE帧(0x8)用于流量控制,例如限制可以发送的数据量以避免接收方不堪重负。在 HTTP/1 下,一次只能发送一个请求。如果客户端开始被数据淹没,它就会停止处理 TCP 数据包;然后 TCP 流量控制 (类似于 HTTP/2 流量控制)启动并减慢数据发送速度,直到接收方准备好处理更多数据。在 HTTP/2 中,同一个连接上有多个流,所以你

不能依赖于 TCP 流量控制，并且必须实现您自己的每流减速方法。

可以发送的数据的初始窗口大小在SETTINGS框架中设置，WINDOW_UPDATE框架用于增加此数量。因此，WINDOW_UPDATE框架是一个简单的框架，没有任何标志，只有一个值（和一个保留位），如表 4.3 所示。

表 4.3 WINDOW_UPDATE帧格式

场地	长度	描述
保留位	1位	不曾用过
窗口大小增量	31位	之前可以发送的八位字节数 必须接收下一个 WINDOW_UPDATE 帧

WINDOW_UPDATE框架没有定义任何标志并适用于给定的流，或者，如果为流0 设置，则适用于整个HTTP/2 连接。因此，发件人必须在流级别和总级别进行跟踪。

HTTP/2 流量控制仅适用于数据帧。即使流量控制窗口已用完，所有其他帧类型（或至少目前已定义的帧类型）仍可继续发送。此功能可防止重要的控制消息（例如WINDOW_UPDATE消息本身）被大数据帧阻塞。此外，DATA帧应该是任何大小的唯一帧。

我在第 7 章中研究了 HTTP/2 流量控制机制。

优先帧

接下来的帧是几个优先级帧（0x2）：

```
[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=3>
  (dep_stream_id=0, weight=201, exclusive=0)
[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=5>
  (dep_stream_id=0, weight=101, exclusive=0)
[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=7> (dep_stream_id=0,权重=1,独占=0)[0.107]发送优先级帧<长度=5,标志=0x00,stream_id=9> (dep_stream_id=7, weight=1, exclusive=0) [0.107] 发送优先帧 <length=5,
flags=0x00, stream_id=11> (dep_stream_id=3, weight=1, exclusive=0)
```

此代码创建多个具有不同优先级的流供 ngnhttp 使用。事实上，ngnhttp 并不直接使用流 3-11；它使用 dep_stream_id 挂起它在开始时设置的流中的其他流。这种预先创建的优先级流的使用允许对请求进行适当的优先级排序，而无需为每个后续新流明确设置优先级。并非所有 HTTP/2 客户端都预定义流，并且 ngnhttp 基于其在 Firefox 模型上的实现，²²因此如果您使用其他工具并且看不到这些 PRIORITY 帧，请不要担心。

²² <https://ngnhttp2.org/documentation/ngnhttp.1.html#dependency-based-priority>

HTTP/2 下的流优先级可能会变得复杂,所以我推迟到第 7 章才开始研究它们。现在,请注意一些请求(例如初始 HTML、关键 CSS 和关键 JavaScript)可以优先于不太重要的请求(例如图像或非关键的异步 JavaScript)。帧格式如表 4.4 所示,但在您学习第 7 章之前,您不会理解这种格式。

表 4.4 PRIORITY 帧格式

场地	长度	描述
E (独家)	1位	指示流是否独占(仅当为此帧设置了优先级标志)
流依赖	31位	此标头依赖于哪个流的指示符(仅当为此帧设置了优先级标志时才设置)
重量	8位	此流的权重(仅当为此帧设置了优先级标志时才设置)

PRIORITY 帧(0x2)是固定长度的,不定义任何标志。

标题框架

最后,完成所有这些设置后,您就可以了解协议的实质并可以发出 HTTP/2 请求。HTTP/2 请求在 HEADERS 帧(0x1)中发送:

```
[0.107]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 :方法:GET

:path: / :scheme:
https :authority: www.facebook.com
accept: */* accept-encoding: gzip, deflate
user-agent: nghttp2/1.28.0
```

如果忽略 HTTP/2 帧头的前几行,其余部分看起来应该与 HTTP/1 请求有些相似。你可能还记得第 1 章,一个 HTTP/1 请求由第一行和强制主机头(以及任何其他 HTTP 头)的组合组成:

```
GET / HTTP/1.1 主机:
www.facebook.com
```

在 HTTP/2 中,不是在HEADERS帧中使用特定的请求帧类型或不同的第一行,而是将所有内容作为标头发送,并且创建了新的伪标头(以冒号开头)来定义请求的各个部分HTTP 请求行:

```
:方法:获取

:path: / :scheme:
https :authority: www.facebook.com
```

请注意，`:authority`伪标头已替换 HTTP/1.1 主机标头。

HTTP/2 伪标头是严格定义的,²³与标准 HTTP 标头不同,如果不更改 HTTP/2 就无法添加它们,因此您不能像这样创建新的伪标头:

`:bari:价值`

对于任何特定于应用程序的标头,您必须坚持使用没有初始冒号的普通 HTTP 标头:

`巴里:价值`

但是,您可以使用新规范创建伪标头,在撰写本文时已经发生过一次: `:protocol`伪标头已添加到使用 HTTP/2 RFC 引导 WebSockets 中。²⁴使用新的伪标头可能需要新的设置指示客户端和服务器支持的参数。

这些伪标头也可以显示在客户端工具中,例如 Chrome 的开发人员工具 (图 4.11),因此它们也表示 HTTP/2 请求 (尽管其他浏览器,如 Firefox,在撰写本文时不显示伪标头)。

Chrome 开发者工具网络选项卡中的 HTTP/2 伪标头

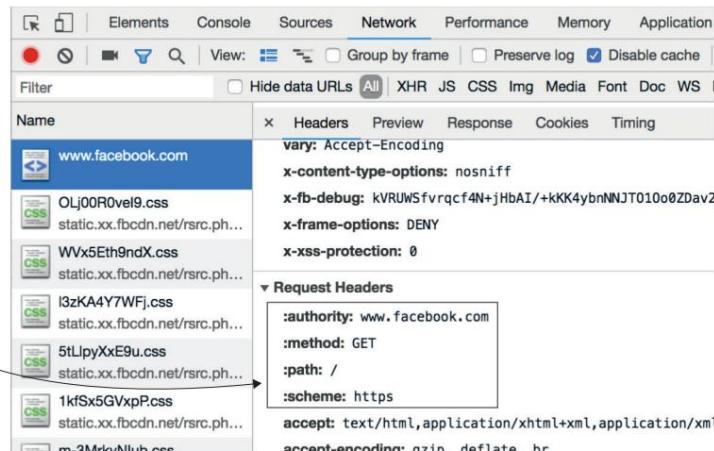


图 4.11 Chrome 开发者工具中的 HTTP/2 伪标头

另请注意,HTTP/2 强制使用小写 HTTP 标头名称。HTTP/1 官方对标头名称不区分大小写,尽管某些实现并未严格遵守此规范。HTTP 标头值可以包含不同的大小写,但标头名称本身不能。HTTP/2 对格式错误的 HTTP 也更加严格

²³ <https://tools.ietf.org/html/rfc7540#section-8.1.2>

²⁴ <https://tools.ietf.org/html/rfc8441#section-3>

标题。前导空格、双冒号和换行符可能会导致 HTTP/2 出现问题，即使大多数 HTTP/1 实现都会处理它们。此示例是在帧级别检查 HTTP/2 消息的一个重要用例，正如您在此处所做的那样。这些错误通常在这个低级别突出显示，而客户端在发现无效的 HTTP 标头时会向用户返回更多神秘的错误消息（例如 Chrome 中的ERR_SPDY_PROTOCOL_ERROR），从而阻止您的网站运行。HEADERS 帧格式如表 4.5 所示。

表 4.5 HEADERS 帧格式

字段	长度	描述
焊盘长度	8 位（可选）	一个可选字段，指示填充字段的长度（仅当为此帧设置了填充标志）
E（独家）	1位	指示流是否排除 sive（仅当为该帧设置了优先级标志时才设置）
流依赖	31位	指示此流所在的流 标头取决于（仅当为此帧设置了优先级标志）
重量	8位	此流的权重（仅当为此帧设置了优先级标志时才设置）
Header Block Fragment 帧减去的长度	该表中的其他字段	请求标头（包括伪标头）
填充	由焊盘指示 长度字段（可选）	为每个填充字节设置为 0（仅当为该帧设置了 Padded 标志时才设置）

我在第 7 章中讨论了 E、Stream Dependency 和 Weight 字段。添加 Pad Length 和 Padding 字段是出于安全原因，可以选择允许隐藏消息的真实长度。标头块片段字段是发送所有标头（包括伪标头）的地方。该字段不是明文，正如 nghttp 输出可能暗示的那样。我在第 8 章研究了 HPACK 标头压缩格式，所以现在不用担心，特别是因为 nghttp 等工具会自动为您解压缩 HTTP 标头。

HEADERS 帧定义了四个可以在公共帧头中设置的标志：

如果没有其他帧跟随着此 HEADERS 帧（例如 POST 请求的 DATA 帧），则设置 END_STREAM (0x1)。

有点违反直觉的是，CONTINUATION 框架（在本章后面讨论）不受此限制；它们被认为是 HEADERS 框架的延续，而不是额外的

帧并由 END_HEADERS 标志控制。END_HEADERS

(0x4) 表示所有 HTTP 标头都包含在该帧中，并且后面没有带有附加标头的 CONTINUATION 帧。

使用填充时设置PADDED (0x8)。该标志表示 DATA 帧的前 8 位表示在DATA帧的末尾添加了多少填充

标题框架。

PRIORITY (0x20)表示E、 Stream Dependency、 Weight字段是设置在这个框架中。

如果 HTTP 标头大于单个帧,则使用CONTINUATION帧（紧接在HEADERS帧之后继续）,而不是额外的HEADERS帧。

与 HTTP 主体相比,这个过程可能看起来过于复杂,HTTP 主体根据需要使用尽可能多的数据帧。但是表 4.5 中讨论的其他字段只能使用一次,并且在相同请求的后续HEADERS帧中以不同方式设置它们会导致问题。让CONTINUATION帧紧跟在 HEADERS帧之后而不是允许交错的要求也限制了 HTTP/2 的多路复用性质,并且考虑了替代方案。²⁵ 现实情况是CONTINUATION帧很少被使用并且大多数请求将适合单个HEADERS帧.

通过这些知识查看输出,您可以理解第一部分
现在消息好一点:

```
[0.107]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:
nghttp2/1.28.0
```

每个新请求都被赋予一个唯一的流 ID,从上次使用的流 ID 递增（在本例中为 11,从 nghttp 创建的最后一个PRIORITY帧开始,所以这个帧是用流 ID 13 创建的,因为偶数头被保留用于服务器）。设置了各种标志,这些标志组合在一起使十六进制值成为0x25 ,并且 nghttp2 有助于显示在下面的行中。设置END_STREAM (0x1)和END_HEADERS (0x4)标志以指示此帧包含完整请求并且没有DATA帧（因为POST请求可能有）。设置优先级标志(0x20)以指示在该帧中使用了优先级。将这些十六进制值相加 (0x1 + 0x4 + 0x20),您将得到帧头中显示的0x25 。此流依赖于流 ID 11,因此它被赋予适当的优先级和该优先级内的权重 16。同样,现在不要太担心这个话题;我在第 7 章解释了优先级.nghttp 指出这个流是新的（打开新流）,然后列出了各种 HTTP pseudoheaders 和 HTTP 请求标头。

²⁵ <https://github.com/http2/http2-spec/wiki/ContinuationProposals>

HTTP 响应也与同一流上的HEADERS帧一起发送,如您在本例中所见：

```
[ 0.257] recv (stream_id=13) :status: 200 [ 0.257] recv (stream_id=13)
x-xss-protection: 0 [ 0.257] recv (stream_id=13) pragma: no-cache [ 0.257] recv
(stream_id=13) 缓存控制:私有、无缓存、无存储、必须重新验证

[ 0.257] recv (stream_id=13) x-frame-options: DENY [ 0.257] recv (stream_id=13)
strict-transport-security: max-age=15552000; preload [ 0.257] recv (stream_id=13) x-content-type-options: nosniff
[ 0.257] recv (stream_id=13) 到期时间:星期六,2000年1月1日00:00:00 GMT [ 0.257] recv (stream_id=13) 设置 cookie:
fr=0m7urZrTka6WQuSGa..BaQ4Ay.61.A AA.0.0.BaQ42y.12345678;过期 = 2018年3月27日星期二12:10:26 GMT;最
大年龄=7776 000;路径=/;域名=.facebook.com;安全的; httponly [ 0.257] recv (stream_id=13) set-cookie:
sb=so11234567890TZ7e-i5S2To; expi res=2019年12月27日星期五12:10:26 GMT;最大年龄=63072000;路径=/;域名
=.facebook.com;安全的; httponly [ 0.257] recv (stream_id=13) vary: Accept-Encoding [ 0.257] recv (stream_id=13)
content-encoding: gzip [ 0.257] recv (stream_id=13) content-type: text/html; charset = utf-8 [ 0.257] recv (stream_id=13)
x-fb-debug:yre7eqv05dkxf8r1+1234567890nvi+ap dyg7hbw6t7hbw6t7ncetgkiirqjadlwj87hmhk6z/n3on3oecnept
= 10 deckt = 10.2121212 = 3.212112.12 = 3. 26 GMT [ 0.257] recv HEADERS frame <length=517, flags=0x04,
stream_id=13>; END_HEADERS (padlen=0)
```

;第一个响应头

在这里,您首先看到状态伪标头(: status: 200),它与 HTTP/1.1 不同,它只给出三位数的 HTTP 代码(200),而不是该状态代码的文本表示 (例如200 OK)。这个伪标头之后是各种 HTTP 标头,不过,这不是它们在线路上发送的方式,正如您在第 8 章查看 HPACK 时会看到的那样。然后 ngnhttp 列出了HEADERS帧的详细信息。令人困惑的是 (至少对我而言) ,ngnhttp 在帧负载之后而不是之前给出了帧详细信息,正如我希望的那样。²⁶这些详细信息包括END_HEADERS标志 (0x04) ,表示整个 HTTP 响应标头适合此单个框架。

尾随标头HTTP/1.1

引入了尾随标头的概念,可以在正文之后发送。这些标头允许无法预先计算的元数据。例如,对于流数据,可以计算内容的校验和或数字签名并将其作为尾随的 HTTP 标头包含在内。

²⁶ <https://github.com/ngnhttp2/ngnhttp2/issues/1163>

尾随标题（续）

实际上，尾随标头的支持很差并且很少使用。但是 HTTP/2 决定继续支持它们，因此一个HEADERS帧（或一个HEADERS帧后跟一个或多个CONTINUATION帧）出现在该流的所有DATA帧之前或之后（可选）。

数据帧

在HEADERS帧之后是DATA帧（0x0），用于发送消息体。在 HTTP/1 中，消息正文在 HTTP 标头之后的响应中发送，在双换行符（表示 HTTP 标头结束）之后。在 HTTP/2 中，数据是一种单独的消息类型。您可以发送标头，然后发送一些正文、不同流的一部分、更多的正文，等等。通过将 HTTP/2 响应分成一个或多个帧，您可以在同一连接上拥有多路复用流。

HTTP/2数据帧很简单，包含所需的任何数据：UTF-8 编码、gzip 压缩、HTML 代码、构成 JPEG 图片的字节，或其他任何内容。主帧头包含长度，因此帧本身的数据帧中不需要长度。与HEADERS帧一样，出于安全原因， DATA帧允许使用填充来掩盖消息的大小，因此可以在开头使用Pad Length字段来说明长度。因此， DATA帧格式很简单，如表 4.6 所示。

表 4.6 数据帧格式

场地	长度	描述
填充长度 8 位（可选）		指示填充字段长度的可选字段（仅当设置了 PADDED 标志时才包含）
数据	帧的长度减去任何填充字段	数据
填充	由焊盘指示 长度字段（可选）	为每个填充字节设置为 0（仅当已设置 PADDED 标志）

DATA帧定义了两个可以在公共帧头中设置的标志：**如果**此帧是流中的最后一个，则设置 END_STREAM（0x1）。 使用填充时设置PADDED（0x8）。这意味着DATA帧的前 8 位用于指示帧末尾添加了多少填充。

在示例中，出于空间原因，我删除了大部分内容，但是……等等。行通常会填充适当的数据：

```

<!DOCTYPE html>
<html lang= zh-cn id= facebook class= no_js >
<head><meta charset= utf-8 />
…ETC。
[0.243]接收数据帧<长度=1122,标志=0x00,stream_id=13>…等等。
[0.243]接收数据帧<长度=2589,标志=0x00,stream_id=13>…等等。
[0.264]接收数据帧<长度=13707,标志=0x00,stream_id=13>…等等。
[0.267]发送WINDOW_UPDATE帧<长度=4,标志=0x00,stream_id=0> (window_size_increment=33706)[0.267]发送WINDOW_UPDATE
帧<长度=4,flags=0x00,stream_id=13> (window_size_increment=33706)

…ETC。
[416.688] recv 数据帧 <length=8920, flags=0x00, stream_id=13>

```

在这里,您看到 HTML 代码在各种数据帧中发送 (nghttp 有助于为您解压缩数据) ,并且当客户端处理这些帧时,它会发回WINDOW_UPDATE帧,从而允许服务器继续发送更多数据。有趣的是,Facebook 选择最初发送相对较小的数据帧 (1,122 个八位字节、2,589 个八位字节等) ,尽管客户端愿意处理更大的帧 (最多 65,535 个八位字节) 。我不确定这种选择是否是有意的 (尽可能快地向客户端获取尽可能多的数据) ,是由于最初较小的 TCP 拥塞窗口,还是出于其他原因。

由于默认情况下 HTTP/2 数据帧可以拆分成多个部分,因此不需要分块编码 (在 4.1.1 节中讨论) 。HTTP/2 规范甚至说“分块传输编码 … 不得在 HTTP/2 中使用。”

离开框架

GOAWAY 帧(0x7)是下一条消息:

```
[417.226] 发送 GOAWAY 帧 <length=8, flags=0x00, stream_id=0>
(last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

这种听起来有点粗鲁的帧类型用于关闭连接,要么是因为没有更多的消息要发送,要么是因为发生了严重的错误。 GOAWAY 帧格式如表 4.7 所示。

表 4.7 GOAWAY 帧格式

场地	长度	描述
保留位	1位	不曾用过
最后一个流 ID	31位	处理的最后一个传入流 ID,以允许客户端知道是否错过了最近启动的流

表 4.7 GOAWAY 帧格式（续）

场地	长度	描述
错误代码	32位	错误代码,以防因错误而发送 GOAWAY 帧: NO_ERROR (0x0) PROTOCOL_ERROR (0x1) INTERNAL_ERROR (0x2) FLOW_CONTROL_ERROR (0x3) SETTINGS_TIMEOUT (0x4) STREAM_CLOSED (0x5) FRAME_SIZE_ERROR (0x6) REFUSED_STREAM (0x7) CANCEL (0x8) COMPRESSION_ERROR (0x9) CONNECT_ERROR (0xa) ENHANCE_YOUR_CALM (0xb) INADEQUATE_SECURITY (0xc) HTTP_1_1_REQUIRED (0xd)
附加调试 数据	剩余帧长（可选）	未定义的、特定于实现的格式

GOAWAY 框架没有定义任何标志。

查看前面 nghttp 输出中的最终消息,您会看到一个 GOAWAY 帧的示例:

```
[417.226] 发送 GOAWAY 帧 <length=8, flags=0x00, stream_id=0>
  (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

nghttp 客户端发送了 GOAWAY 帧,而不是从服务器接收它。在此示例中,nghttp 获取了主页 HTML,但并未请求普通浏览器会请求的所有依赖资源 (CSS、JavaScript 等)。当处理完响应并且客户端不再等待任何数据时,它会发送此帧以关闭 HTTP/2 连接。如果发出后续请求,Web 浏览器可能会保持连接打开,但 nghttp 在收到此响应后已完成,因此决定在退出前关闭连接。当您退出时,浏览器可能会对任何打开的连接做同样的事情。

发送的帧长度最小为 8 个八位字节 (1 位 + 31 位 + 32 位);没有设置标志;并且帧在流 0 上发送。从服务器收到的最后一个流 ID 是 0,因此没有服务器启动的流。没有错误代码(NO_ERROR [0x00]),也没有额外的调试数据。总而言之,这个例子是一种在不再需要连接时干净地关闭连接的标准方法。

4.3.4 其他框架

nghhttp Facebook 示例涵盖了许多 HTTP/2 帧类型,但在这个简单的流程中没有使用更多类型。此外,编写 HTTP/2 以允许扩展帧类型。添加了三种新的帧类型 ALTSVC、ORIGIN 和 CACHE_DIGEST 并将在本节末尾进行讨论。在撰写本文时,只有前两个已正式标准化,但最后一个甚至更多可能会在本书出版时标准化。每个新的 HTTP/2 帧类型、HTTP/2 设置和 HTTP/2 错误代码都必须在互联网号码分配机构 (IANA) 注册。²⁷

延续框架

CONTINUATION帧(0x9)用于大型 HTTP 标头,紧跟在HEADERS帧或PUSH_PROMISE帧之后。因为在处理请求之前需要整个 HTTP 标头,并且要检查 HPACK 字典 (请参阅第 8 章),所以CONTINUATION帧必须紧跟在它继续的HEADERS 帧之后。正如我在讨论HEADERS框架时提到的那样,此要求限制了 HTTP/2 的多路复用性质,并且关于是否需要 CONTINUATION框架或是否应允许更大的HEADERS框架存在很多争论。目前,框架仍然存在,但预计不会被大量使用。

CONTINUATION框架比HEADER或PUSH_PROMISE框架更简单
继续。它包含额外的标题数据。帧格式如表 4.8 所示。

表 4.8 CONTINUATION帧格式

场地	长度	描述
标头块片段	帧的长度减去这个字段	数据

CONTINUATION帧只定义了一个标志,可以在公共帧头中设置。END_HEADERS (0x4)设置后,表示所有 HTTP 标头都在此帧中完成,并且后面没有另一个带有附加标头的CONTINUATION帧。

CONTINUATION帧不使用END_STREAM标志来指示有
没有正文,因为此框架是从原始的HEADERS框架中删除的。

Ping帧

PING帧(0x6)用于测量来自发送方的往返行程,也可用于保持未使用的连接处于活动状态。当它收到此帧时,接收方应立即以类似的PING帧进行响应。两个PING帧都应仅在控制流 (流 ID 0) 上发送。PING帧格式如表4.9所示。

²⁷ <https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml>

表 4.9 PING帧格式

场地	长度	描述
不透明数据	64 位 (8 个八位字节)	要在返回的 PING 请求中发送的数据

PING帧定义了一个可以在公共帧头中设置的标志。不应在初始PING帧中设置ACK (0x1) ,但应在返回的PING帧中设置它。

PUSH_PROMISE FRAME

服务器使用PUSH_PROMISE帧(0x5)告诉客户端服务器将推送客户端未明确请求的资产。

PUSH_PROMISE帧需要向客户端提供有关将要推送的资产的信息,因此它包含通常包含在 HEADERS 帧请求中的所有 HTTP 标头 (并且类似地可以后跟带有标头的推送请求的CONTINUATION帧大于可以容纳在单个框架中)。 PUSH_PROMISE帧格式如表 4.10 所示。

表 4.10 PUSH_PROMISE帧格式

场地	长度	描述
焊盘长度	8 位 (可选)	一个可选字段,指示填充字段的长度
保留位	1位	不曾用过
承诺流 ID	31位	指示在其上的流 这个推送承诺将被发送
帧的标头块片段长度减去	该表中的其他字段	推送的 HTTP 标头 资源
填充	由 Pad Length 字段指示 (可选)	为每个填充字节设置为 0

PUSH_PROMISE帧定义了两个可以在公共帧头中设置的标志：

END_HEADERS (0x4)表示所有 HTTP 标头都包含在该帧中,并且后面没有带有附加标头的 CONTINUATION帧。 使用填充时设置PADDED (0x8)。这意味着DATA帧的前 8 位用于指示在PUSH_PROMISE帧的末尾添加了多少填充。

我在第 5 章讨论了 HTTP/2 服务器推送。

RST_STREAM FRAME

原始 HTTP/2 规范中定义的最终帧是RST_STREAM帧 (0x3) ,用于立即取消 (或重置)流。这种取消可以

由于错误或不再需要请求。也许客户端已经导航离开、取消加载或不需要服务器推送的资源。

HTTP/1.1 不提供此功能。如果您开始在页面上下载大量资源，除非您终止连接，否则即使您离开该页面，您也无法下载资源。您无法在飞行中取消请求。此功能是 HTTP/2 改进 HTTP/1.1 的另一种方式。 RST_STREAM 帧格式如表 4.11 所示。

表 4.11 RST_STREAM 帧格式

场地	长度	描述
错误代码	32位	<p>错误代码，用于解释流被终止的原因：</p> <ul style="list-style-type: none"> NO_ERROR (0x0) PROTOCOL_ERROR (0x1) INTERNAL_ERROR (0x2) FLOW_CONTROL_ERROR (0x3) SETTINGS_TIMEOUT (0x4) STREAM_CLOSED (0x5) FRAME_SIZE_ERROR (0x6) REFUSED_STREAM (0x7) CANCEL (0x8) COMPRESSION_ERROR (0x9) CONNECT_ERROR (0xa) ENHANCE_YOUR_CALM (0xb) INADEQUATE_SECURITY (0xc) HTTP_1_1_REQUIRED (0xd)

RST_STREAM 帧没有定义任何标志。

该规范对这些错误代码的含义几乎没有提供任何指导，即使有，有时也不太清楚。例如，它说明了以下内容，以表明可以使用两个错误代码之一来取消推送的响应：

如果客户端出于任何原因确定它不希望从服务器接收推送的响应，或者如果服务器花了太长时间才开始发送承诺的响应，则客户端可以使用 CANCEL 或 REFUSED_STREAM 发送 RST_STREAM 帧代码并引用推送流的标识符。

最终，由实施者决定使用哪些错误代码以及何时使用。

实现可能并不总是一致。

ALTSVC 帧

自 HTTP/2 规范获得批准以来，ALTSVC 帧(0xa)是第一个添加到 HTTP/2 的帧。它在单独的规范 28 中有详细说明，并允许服务器

²⁸ <https://tools.ietf.org/html/rfc7838>

通告可用于获取此资源的替代服务,如第 4.2.4 节所述。此帧可用于升级 (例如从 h2 到 h2c 连接) 或将流量定向到另一个版本。见表 4.12。

表 4.12 ALTSVC 帧格式

场地	长度	描述
原点长度	16位	Origin 字段的长度
起源	由 Origin-Len 字段指示 (可选)	备用网址
Alt-Svc-Field-Value 帧的长度减去其他	此表中的字段	替代服务类型

ALTSVC 框架没有定义任何标志。

原帧

ORIGIN 框架(0xc)是一个新框架,于 2018 年 3 月标准化,²⁹允许服务器指示该服务器将响应哪些来源 (例如域名)。

这个框架对于客户端决定是否合并连接很有用
HTTP/2 连接。 ORIGIN 帧格式如表 4.13 所示。

表 4.13 ORIGIN 帧格式

场地	长度	描述
原点长度	16位	Origin 字段的长度
起源	由 Origin-Len 字段指示 (可选)	备用网址

可以包含多个 Origin-Len/Origin 对,直至帧的长度。

ORIGIN 框架没有定义任何标志。

在第 6 章讨论连接合并时,我会回到 ORIGIN 框架。

CACHE_DIGEST FRAME

CACHE_DIGEST 框架(0xd)是一个新的框架提案。³⁰该框架允许客户端指示它缓存了哪些资产。它表示服务器不应推送任何这些资源,例如,因为客户端已经拥有它们。表 4.14 中显示了撰写本文时的 CACHE_DIGEST 帧格式 (可能会发生变化)。

²⁹ <https://tools.ietf.org/html/rfc8336>

³⁰ <https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest>

表 4.14 CACHE_DIGEST帧格式

场地	长度	描述
原点长度	16位	Origin字段的长度
起源	由 Origin-Len 字段指示 (可选)此摘要引用的来源	
帧的摘要值长度减去此表中的其他字段 (可选)		Cache-Digest (在第 5 章中讨论)

CACHE_DIGEST框架定义了以下标志：

RESET (0x1)允许客户端告诉服务器重置任何当前持有的
CACHE_DIGEST信息。

COMPLETE (0x2)表示包含的缓存摘要是缓存的完整表示,而不是缓存的子集。

在第 5 章讨论 HTTP/2 服务器推送时,我会回到CACHE_DIGEST框架。

概括

HTTP/2 是一种二进制协议,具有特定的、详细的格式和结构,用于
它的消息。

为此,客户端和服务器必须在发送前同意使用 HTTP/2
任何 HTTP 消息。

对于网络浏览器,这个协议主要是在 HTTPS 连接中制定的
协商,使用称为 ALPN 的新扩展。在 HTTP/2 中,请求和响应
在 HTTP/2 帧中发送和接收。例如, HTTP/2 GET 请求通常作为HEADERS帧发送,响应通常作为
HEADERS帧后跟DATA帧接收。大多数 web 开发人员和 web 服务器管理员不需要关心 HTTP/2 框架,尽
管可以使用工具来查看它们。

存在多个 HTTP/2 框架,并且可以添加新的框架。

实施 HTTP/2 推送

本章涵盖什么是 HTTP/2 推送？请求 HTTP/2 推送的各种方式 HTTP/2 推送如何从服务器端和客户端工作

什么该推,什么不该推 HTTP/2 推送故障排除 HTTP/2 推送的一些风险

5.1 什么是 HTTP/2 服务器推送？

HTTP/2 服务器推送（以下称为HTTP/2 推送）允许服务器发回客户端未请求的额外资源。在引入 HTTP/2 之前，HTTP 是一种简单的请求和响应协议；浏览器请求资源，服务器响应该资源。如果页面需要显示额外资源（例如 CSS、JavaScript、字体、图像等），浏览器必须下载初始页面，查看是否引用了额外资源，然后请求它们。对于图像，提出这些额外的请求可能不会有太大问题；图像通常不会占用初始绘制时间，并且页面会

从图像应该所在的空白区域开始渲染。然而,有些资源对页面呈现至关重要（例如 CSS 和 JavaScript）,在下载这些资源之前,浏览器甚至不会尝试呈现页面。此过程至少增加了一个额外的往返行程,因此会减慢网页浏览速度。HTTP/2 多路复用允许在同一个连接上并行请求所有资源,因此它比 HTTP/1 更好,因为应该有更少的排队。但如果没有 HTTP/2 推送,浏览器将不得不在下载初始页面后发出这些额外请求。因此,在最好的情况下,大多数网页请求至少需要两次往返,而且显示时间可能是您希望的两倍。图 5.1 显示了在第二组请求中同时下载一个 CSS 文件和一个 JavaScript 文件。



图 5.1 关键资源需要额外的请求往返。

图 5.2 显示执行初始绘制大约需要两次往返请求。请注意,由于网络或处理限制,styles.css 和 script.js 资源到达的时间略有不同,而不是同时到达;它们不是并行运行的。

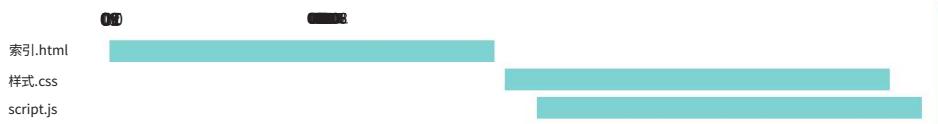


图 5.2 关键资源往返滞后瀑布图

这种往返延迟导致了性能优化,例如使用<style>标签将样式表直接内联到 HTML 页面上,并使用<script>标签在 Java Script 中做类似的事情。通过内联关键资源,浏览器可以在原始页面下载和解析后立即开始首次渲染,而不是等待其他关键资源。

但是,内联资源有几个缺点。对于 CSS,通常只包含关键样式（初始绘制所需的样式）;完整的样式表稍后加载以最小化内联代码量并避免使页面太长。

大的。从 CSS 资源中提取所需的关键样式并将它们嵌入到 HTML 文件中是很复杂的,尽管确实存在可以帮助完成此任务的工具。除了复杂之外,这个过程很浪费;关键 CSS 在网站的每个页面上都被复制,而不是存储在一个可以缓存并在其他页面上重用的 CSS 文件中。更糟糕的是,内联的关键 CSS 通常仍包含在稍后加载的主 CSS 文件中;它不仅跨页面重复,而且在每个页面内重复!其他缺点包括需要使用 JavaScript 来加载任何非关键 CSS 文件,因为仅使用标准<link rel=stylesheet type=text/css href=...>会导致渲染暂停,直到文件加载完成已加载,因为CSS链接标签没有异步属性。此外,如果您想要更改任何此关键 CSS (例如重新设计站点),则需要更改每个页面而不是更新一个通用 CSS 文件。总而言之,内联为首次访问提供了良好的性能优势,但它有点 hack。最好用其他方式解决这个问题,这就是 HTTP/2 推送的目的。

HTTP/2 推送打破了 HTTP 一直遵循的“一个请求 = 一个响应”范式。它允许服务器以多个响应来响应一个请求。“我可以得到这个页面吗?”可以回答“当然,这里有一些额外的资源,你会想要加载该页面。”图 5.3 只显示了获取页面和开始呈现该页面所需的关键资源的一次往返。



图 5.3 使用 HTTP/2 推送可以消除关键的往返延迟
资源。

这个过程也可以用瀑布图来描述,如图 5.4 所示。同样,这三个资源不会同时返回,如它们之间的短暂间隔所示,但所需时间略多于一次往返而不是两次往返。

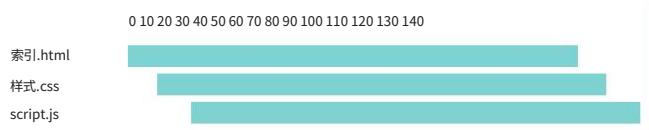


图 5.4 使用 HTTP/2 推送在同一往返中接收所有请求的瀑布图

时间节省也可以通过我在第 2 章中介绍的类型的请求和响应图来说明。图 5.5 显示了通过将所有关键资源与初始页面一起发回而节省的大量时间。

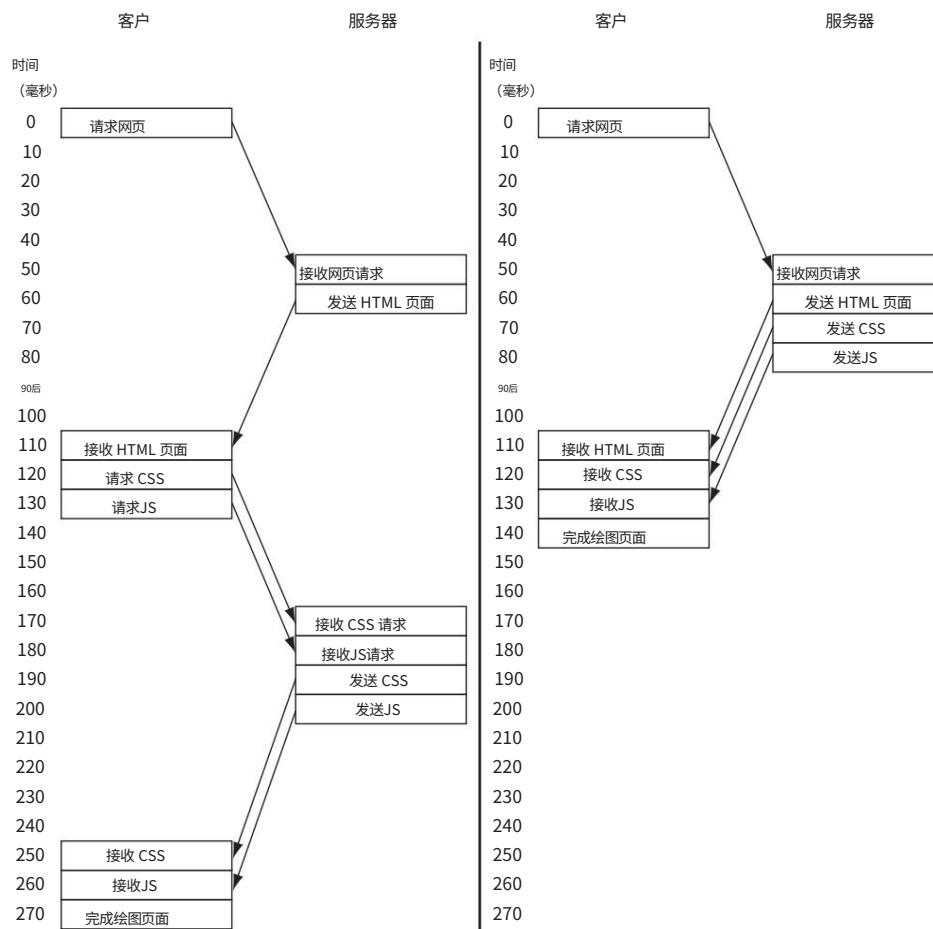


图 5.5 没有 HTTP/2 推送 (左)和有 HTTP/2 推送 (右)的基本网页的请求流

如果正确使用 HTTP/2 推送可以缩短加载时间,但如果过度推送客户端不会使用或缓存中已有的资源,它也可能会阻碍加载时间。你会浪费带宽,你可以更好地使用下载你确实需要的资源。正如我在本章中讨论的那样,应谨慎使用 HTTP/2 推送并进行一些思考。

HTTP/2 推送是 WebSockets 或 SSE 的替代品吗？

需要注意的一个关键点是推送资源仍然仅在响应初始请求时发送。不可能纯粹根据服务器决定客户端可能想要或需要资源来使用 HTTP/2 推送资源。WebSockets 和服务器发送事件 (SSE) 等技术确实允许双向流，但 HTTP/2 并不是真正的双向；一切仍然是从客户端请求发起的。推送资源是为响应初始请求而做出的额外响应。当该初始请求完成时，流将关闭，除非发出另一个客户端请求，否则无法推送其他资源。因此，HTTP/2 推送并不能替代当前指定的 WebSockets 或 SSE，尽管如果进一步扩展它可能会替代（请参阅第 5.9 节）。

5.2 如何推送

如何推送取决于您的 Web 服务器，因为在撰写本文时并非每个服务器都支持 HTTP/2 推送。某些 Web 服务器可以使用 HTTP 链接标头或配置进行推送。其他（如 IIS）需要编写代码，因此它们只能从动态生成的页面而不是静态 HTML 文件推送。请查阅您的 Web 服务器文档，了解您的服务器是否支持 HTTP/2 推送以及如何使用它。

对于本章的其余部分，我主要在示例中使用 Apache、nginx 和 Node.js。这些概念适用于大多数 HTTP/2 Web 服务器，即使实现细节略有不同。如果您的 Web 服务器不支持 HTTP/2 推送，您仍然可以在您的服务器周围构建一个内容分发网络（请参阅第 3 章）并使用该 CDN 来提供推送功能。

5.2.1 使用 HTTP 链接头进行推送

许多网络服务器（如 Apache、nginx 和 H2O）和一些 CDN（如 Cloudflare 和 Fastly）使用 HTTP 链接头来通知网络服务器进行推送。如果 Web 服务器看到这些 HTTP 标头，它会推送标头中引用的资源。在 Apache 中，您可以使用这样的配置来添加这样的链接标头：

标题添加链接 “</assets/css/common.css>;as=style;rel=preload”

如果您使用的是 nginx，则语法类似：

```
add_header Link </assets/css/common.css>;as=style;rel=preload
```

推送链接标头通常包含在条件语句中，以仅对特定路径或文件类型应用推送。例如，在 Apache 中，要使用 index.html 文件而不是所有资源来推送 CSS 样式表，请使用如下语法：

```
<文件匹配 index.html >
    标题添加链接 “</assets/css/common.css>;as=style;rel=preload”
</文件匹配>
```

其他 Web 服务器也有类似的方法添加 HTTP 标头,但并非所有 Web 服务器都使用 HTTP 链接标头方法来推送资源。对于那些这样做的人,当请求被发送回客户端时,Web 服务器读取这些标头,请求该资源,并发送它。需要设置 `rel=preload` 属性以向 Web 服务器指示要推送此资源,但 `as=style` 部分(指示资源类型)可能是可选的。例如,此 `as` 属性可用于决定优先级,尽管其他 Web 服务器可能不需要它:Apache 使用内容类型而不是 `as` 属性来确定优先级。

预加载 HTTP 标头和 HTTP/2 推送

早于 HTTP/2,最初是作为客户端提示(在第 6 章中讨论)。此标头将允许浏览器立即获取这些资源,而无需在决定是否需要下载资产之前等待下载、读取和解析整个页面。预加载标头允许网站所有者说,“肯定需要这个资源,所以如果你的缓存中还没有它,我建议你尽快请求它。”

预加载链接标头已被许多 HTTP/2 实现重新用于实现服务器推送,以进一步采取这一提示并主动发送资源。如果你想使用原来的预加载目的而不是推送资源,你通常可以使用 `nopush` 属性:

标题添加链接 “`</assets/css/common.css>;as=style;rel=preload;nopush`”

目前,你没有标准的方法来做相反的事情(说链接头应该被推送但不被视为预加载头),尽管 H2O 网络服务器(和 CDN Fastly,它使用这个网络服务器 a)已经添加了 `x-http2-push-only` 属性来处理这种情况:

链接:`</assets/jquery.js>;as=script;rel=preload;x-http2-push-only`

Preload 也可以在 HTML 本身的 HEAD 标签中设置,代码如下:

```
<link rel="preload" href="/assets/css/common.css" as="style">
```

但是,只有 HTTP 标头版本通常适用于 HTTP/2 推送。出于 HTTP/2 推送目的,HTML 版本被忽略,因为与读取 HTTP 标头相比,服务器解析 HTML 以提取这些标头更加复杂和耗时。Web 浏览器无论如何都需要解析 HTML,因此它们接受预加载提示的任何一种方法。

对于客户端提示,必须指定 `as` 属性,但对于 HTTP/2 推送,情况可能并非如此。为了避免混淆,我建议始终设置此属性。完整的 `as` 属性集在 w3.org 网站上列出,包括脚本、样式、字体、图像和提取。请注意,其中一些属性(尤其是字体)也需要 `crossorigin` 属性。

预加载 HTTP 标头和 HTTP/2 推送（续）

有些人发现重用 HTTP/2 推送的预加载标头令人困惑。

他们说为新目的重用现有功能不是一个好主意，并建议更改此功能。尽管存在这种担忧，但使用似乎仍在增长。

对客户端提示和服务器推送都使用预加载标头的一个额外好处是，不支持 HTTP/2 推送的客户端/服务器组合仍然可以使用标头以高优先级预加载资源，因此您仍然可以获得一些性能曼斯增益。我将返回到 5.8 节中的预加载指令，讨论它们和 HTTP/2 推送的不同用例。我想在这里为那些认识到与客户端提示重叠的读者提供一些信息。

-
- ^a <https://www.fastly.com/blog/optimizing-http2-server-push-fastly>
 - ^b <https://www.w3.org/TR/preload/#as-attribute>
 - ^c [#font-fetching-requirements](https://github.com/w3c/preload/issues/99)
 - ^d

在 Apache 中进行测试时，您应该关闭 PushDiary，它会尝试防止在同一连接上对同一资源进行两次推送
(更多内容请参见第 5.4.4 节)：

```
H2PushDiarySize 0
```

浏览器中的显式刷新请求 (F5) 会导致 Apache 忽略 PushDiary，但在测试时关闭 PushDiary 更容易；否则，您可能会看到不一致的结果。其他服务器可能有类似的推送跟踪需要关闭。您还可以使用两个链接标头推送多个标头：

```
标题添加链接 "</assets/css/commoncss>;rel=preload;as=style"
标题添加链接 "</assets/js/common.js>;rel=preload;as=script"
```

或者通过将标头组合成一个以逗号分隔的标头：

```
标题添加链接 "</assets/css/common.css>;rel=preload;as=style, </assets/js/
common.js>;rel=preload;as=script"
```

在第 1 章中，我指出这两种方法在 HTTP 协议中在语法上是相同的，因此可以使用任何一种。

5.2.2 查看 HTTP/2 推送

推送的资源在 Chrome 开发者工具的 Initiator 栏中显示，如图 5.6 所示。

在这里你可以看到第二个资源 (common.css) 被服务器推送了。
您还看到资源立即开始下载，在该请求的瀑布图中没有绿色等待 (TTFB)，正如您在后续请求中看到的那样。

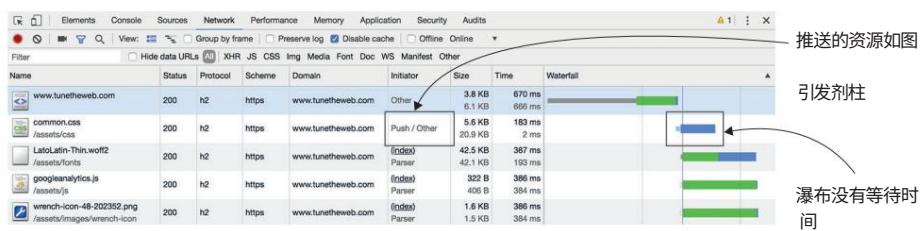


图 5.6 Chrome 开发者工具网络选项卡上的 HTTP/2 推送资源

图 5.7 显示了没有推送的相同页面加载（其中 common.css 请求已从第二个请求资源移动到第三个请求资源并且未被推送）。

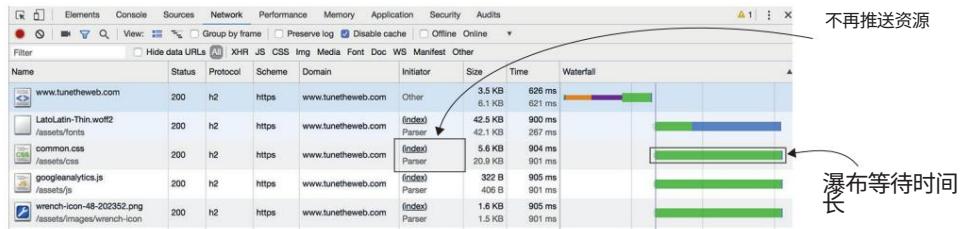


图 5.7 与图 5.6 相同的页面加载,没有 HTTP/2 推送

webpagetest.org 生成的瀑布图不会以任何不同的方式指示推送的资源。但是点击资源在详情部分显示 SERVER PUSHED ,如图5.8所示。

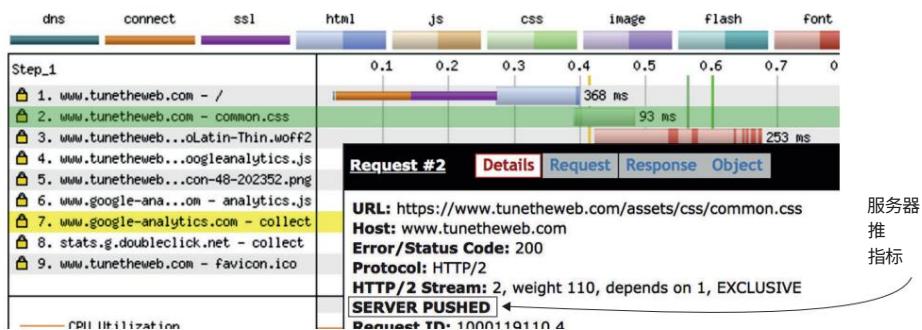


图 5.8 WebPagetest 中的推送资源

您还可以使用 nghttp 发出 Web 请求,以便您可以看到第 4 章中讨论的帧。使用以下命令（适当更改 URL）：

```
$ nghttp -anv https://www.tunetheweb.com/performance/
```

此命令请求页面所需的资源和任何资产（-a标志），不在屏幕上显示下载的数据（-n标志），并打开详细输出以显示 HTTP/2 帧（-v标志）。

连接并使用SETTINGS和PRIORITY设置连接后
frames, nghttp2 使用HEADERS frame 请求页面：

```
[0.013]发送头帧<长度=53,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
方法:获取:路径: /性
能/ :方案:https:权威:
www.tuntheweb.com接受:/*/*接
受编码: gzip, deflate [0.017] recv (stream_id=13) 用户代理:nghttp2/1.28.0
```

在收到返回的页面之前，您会看到它收到了一个用于推送资源的PUSH_PROMISE帧，如下所示。请记住，nghttp 首先显示接收到的帧的帧内容，然后显示帧详细信息：

```
[ 0.017] recv (stream_id=13) :scheme: https [ 0.017] recv
(stream_id=13) :authority: www.tuntheweb.com [ 0.017] recv (stream_id=13) :path: /assets/css/
common.css [ 0.017] recv (stream_id=13) :method: GET [ 0.017] recv (stream_id=13) accept: /* [ 0.017]
recv (stream_id=13) accept-encoding: gzip, deflate [ 0.017] recv (stream_id=13) 用户代理:nghttp2/1.28.0
[ 0.017] recv (stream_id=13) host: www.tuntheweb.com [ 0.017] recv PUSH_PROMISE frame <length=73,
flags=0x04, stream_id=13>
```

```
; END_HEADERS
(padlen=0, promised_stream_id=2)
```

PUSH_PROMISE帧类似于浏览器为获取原始资源而发送的HEADERS帧，但有两个重要区别：

帧是由服务器发送给浏览器的，而不是相反。

这是服务器的提示，告诉客户端，“我将向您发送此资源，就好像您这样请求一样。”

它包括promised_stream_id，这是推送资源的流 ID，如最后一行所示，说明推送资源将在流 ID 2 上发送。服务器启动的流（目前仅用于推送流）是偶数。

在此之后，服务器通过使用后跟数据帧的标头帧在请求流（13）上返回最初请求的资源。然后它在承诺的流（2）上发送推送的资源，再次使用HEADERS帧后跟DATA

框架：

```
[0.017]recv (stream_id=13) :状态:200[0.017]recv (stream_id=13)
日期:星期日,2018年2月4日12:28:07GMT[0.017]recv (stream_id=13)服务器:Apache[0.017]recv
(stream_id=13)last-modified:Thu,18Jan201821:52:14GMT[0.017]recv (stream_id=13)accept-ranges:
bytes[0.017]recv (stream_id=13)缓存控制:max-age=10800,public[0.017]recv (stream_id=13)expires:Sun,04Feb
201815:28:07GMT[0.017]recv (stream_id=13)vary:Accept-Encoding,User-Agent[0.017]recv (stream_id=13)内容编
码:gzip[0.017]recv (stream_id=13)链接:</assets/css/common.css>;rel=preload[0.017]recv (stream_id=13)内容长度:
6755[0.017]recv (stream_id=13)内容类型:文本/html; charset=utf-8[0.017]recv (stream_id=13)push-policy:default
[0.017]recv HEADERS frame<length=2035,flags=0x04,stream_id=13>;END_HEADERS(padlen=0)
```

```
;第一响应头[0.017]recv DATA frame
<length=1291,flags=0x00,stream_id=13>[0.017]recv DATA frame<length=1291,flags=0x00,
stream_id=13>[0.018]recv DATA frame<length=1291,标志=0x00,stream_id=13>[0.018]接收数据帧<长度
=1291,标志=0x00,stream_id=13>[0.018]接收数据帧<长度=1291,标志=0x00,stream_id=13>[0.018]接收数据
帧<length=300,flags=0x01,stream_id=13>;END_STREAM[0.018]recv (stream_id=2):status:200
[0.018]recv (stream_id=2):date:Sun,04Feb201812:28:07GMT[0.018]recv (stream_id=2):服务器:
Apache[0.018]recv (stream_id=2):last-modified:Sun,07Jan201814:57:44GMT[0.018]recv
(stream_id=2):accept-ranges:bytes[0.018]recv (stream_id=2):缓存控制:最大-age=10800,
public[0.018]recv (stream_id=2):expires:Sun,04Feb201815:28:07GMT[0.018]recv (stream_id=2)
vary:Accept-Encoding,User-Agent[0.018]recv (stream_id=2):内容编码:gzip[0.018]recv (stream_id=2):内
容长度:5723[0.018]recv (stream_id=2):内容类型:文本/css; charset=utf-8[0.018]recv HEADERS frame
<length=63,flags=0x04,stream_id=2>;END_HEADERS(padlen=0)
```

```
;第一个推送响应头
[0.018]接收数据帧<长度=1291,标志=0x00,stream_id=2>[0.018]接收数据帧<长度=1291,标志=0x00,
stream_id=2>[0.018]接收数据帧<长度=1291,标志=0x00,stream_id=2>[0.018]recv DATA frame
<length=1291,flags=0x00,stream_id=2>[0.018]recv DATA frame<length=559,flags=0x01,
stream_id=2>;结束_流
```

5.2.3 使用链接头从下游系统推送

如果您使用 HTTP 链接标头来指示要推送的资源，则不需要在 Web 服务器配置中设置这些标头。正如第 3 章中所讨论的，出于性能和安全原因，在下游应用程序代码（可能是 Tomcat、NodeJS 或某些 PHP 处理程序等应用程序服务器）之前使用 Apache 等 Web 服务器是很常见的。如果这些应用程序服务器通过使用 HTTP 链接标头（如 Apache 和 nginx）支持 HTTP/2 推送的 Web 服务器进行代理

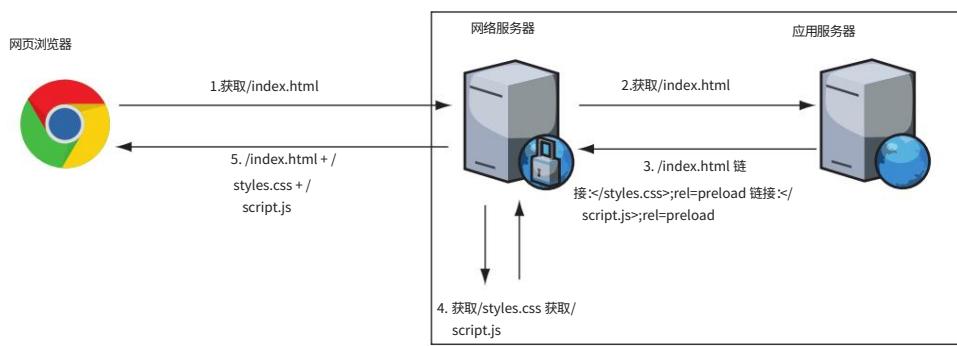


图 5.9 HTTP/2 从下游应用服务器推送链接头

do),只要你有设置HTTP响应头的能力,应用服务器就可以请求web服务器推送资源,如图5.9所示。

使用链接 HTTP 标头允许应用程序告诉 Web 服务器要推送什么,因此所有逻辑都可以在一个地方,而不必每次都更改 Web 服务器配置和应用程序代码。即使这些后端连接是 HTTP/1 连接,此过程也能正常工作。即使你想从那里推送,你也不需要后端服务器上的 HTTP/2 考虑到这个过程可能涉及的复杂性(在第 3 章中讨论),这是一个真正的祝福!图 5.10 显示了此流程在请求-响应图中的样子。

要查看此流程的示例,请使用 HTTP/1.1 创建一个简单的节点服务,如下所示
清单如下所示。

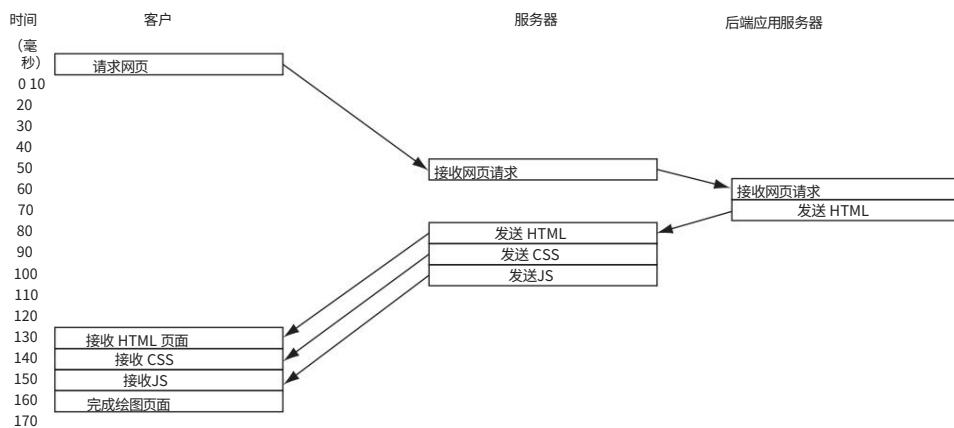


图 5.10 使用链接头从后端应用服务器推送资源

清单 5.1 带有 HTTP 链接头的 HTTP/1.1 节点服务

```
var http = require('http') // 常量端口 = 3000

const requestHandler = (request, response) => {
  console.log(request.url)
  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='text/css' href='/assets/css/common.css'>\n`)
  response.write(`</head>\n`); response.write(`<body>\n`); response.write(`<h1>测试</h1>\n`); response.write(`</body>\n`)
  response.write(`</html>\n`); response.end();
}

var server = http.createServer(requestHandler); server.listen(port);
console.log(`服务器正在监听 + 端口 ${port}`)
```

将这段代码放在一个名为 app.js 的文件中,然后使用以下命令运行它:

节点应用程序.js

你应该看到这样一行:

服务器正在侦听 3000

此代码侦听端口 3000 并返回一个简单的硬编码网页,该网页引用HEAD标记中的样式表并将该样式表作为链接标头包含在内。您可以使用 curl 在另一个窗口中检查此结果:

```
$ curl -v http://localhost:3000
* 将 URL 重建为: http://localhost:3000/* 正在尝试 ::1...
* TCP_NODELAY 设置 * 连接
到本地主机 (::1) 端口 3000 (#0)
> GET / HTTP/1.1 > 主机:本
地主机:3000 > 用户代理:curl/7.56.1 >
接受:/* >
< HTTP/1.1 200 OK < Link: </assets/css/common.css>;rel=preload;as=style < Content-Type: text/html < Date: Sun, 04 Feb 2018 15:46:12 GMT <连接:保持活动状态
```

```
<传输编码:分块
<
<!DOCTYPE html>
<html>
<head>
<link rel= stylesheet type= text/css media= all href= /assets/css/
common.css ></head><body><h1>测试</h1></body></html> * 到主机
localhost 的连接 #0 完好无损
```

要允许通过 Apache 调用此节点服务器,请将以下行添加到 Apache 配置中。您需要在 Apache 中启用 mod_proxy 和 mod_proxy_http :

```
ProxyPass /testnodeservice/ http://localhost:3000/
```

然后代码通过 Apache 调用此服务,在端口 443 上通过 HTTPS 侦听。

此代码允许您在浏览器中通过 HTTP/2 调用此服务,即使 Node 应用程序未设置为 HTTP/2 或 HTTPS;Apache 会为您处理该过程。您会看到 Apache 已经推送了样式表,如图 5.11 所示。

Name	Status	Protocol	Scheme	Domain	Initiator	Size
testnodeservice/ www.tunetheweb.com	200	h2	https	www.tunetheweb.com	Other	473 157
common.css /assets/css	200	h2	https	www.tunetheweb.com	Push / Other	7.2 K 20.9 K

图 5.11 链接标头中引用的资源可以允许来自下游系统的推送。

在此示例中,推送的资源 (common.css) 由 Apache 提供。链接的资源可以由 Apache 本身、下游应用程序 (在本例中为 NodeJS) 或另一个下游系统提供服务。只要 Apache 能够发出资源请求,它就可以推送任何此类资源。

Web 服务器不能为另一个域推送资源。例如,如果您从 example.com 加载一个页面,而该页面从 google.com 加载图像,则您无法推送该图像;只有 google.com 可以推送它。有关此主题的更多讨论,请参阅第 5.5.1 节。

前面的示例 (其中始终推送样式表) 很简单,但您可以使用您通过使用 HTTP 链接标头的 Web 服务器 (或 CDN) 代理的任何下游服务器语言创建更复杂的示例。应用程序

可以根据它对请求或该用户会话的了解来决定推送什么以及何时推送,但仍将实际推送卸载到 Web 服务器。

5.2.4 提前推送

在 Web 服务器配置中设置 HTTP 链接标头并不是推送资源的唯一方法。如何执行此操作取决于您的 Web 服务器,因为该过程是特定于实现的。例如,Apache 使用H2PushResource指令:

H2PushResource 添加/assets/css/common.css

nginx 提供类似的语法:

http2_push /assets/css/common.css;

未使用的推送资源不会出现在 Chrome 开发者工具中 如果页面使用了推送的资源,它们只会显示在 Chrome 开发者工具的网络选项卡上。对于preload hints,preloader算作一次使用,所以所有用HTTP链接头方式推送的资源都会出现(前提是包含as属性)。但是,如果您使用其他方式推送,并且页面没有使用该资源,则该资源将在后台推送,但不会显示在 Chrome 开发者工具中。

如果您无法在 Chrome 开发者工具中看到您的推送资源,请检查该网页是否确实需要推送的资源。如果页面不需要推送资源,那么推送就浪费了。

与 HTTP 链接标头方法相比,直接推送的优势在于服务器不必等待资源返回来检查链接标头然后推送资源。相反,可以在服务器处理原始请求时推送依赖资源。这对于 Web 服务器直接从磁盘为自己提供服务并因此生成速度很快的简单静态资源可能无关紧要,但它会对生成速度较慢的资源产生相当大的影响。图 5.12 显示了一个请求和响应图与本章前面的图 5.5 类似,但这一次,网页生成需要 100 毫秒,可能是因为它需要数据库查找或其他一些动态处理。

该图显示了一个很大的差距,其中没有通过 HTTP/2 连接发送或接收任何东西,这是一种浪费,让人想起 HTTP/2 试图解决的一些队头阻塞问题。CSS 和 JavaScript 可能会更快地生成,因为它们可以是静态的并由 Web 服务器从本地磁盘获取(或者甚至缓存在 Web 服务器中)。该时间可用于推送一些资源,以便在页面本身到达时依赖资源已经存在,如图 5.13 所示。

¹ https://icing.github.io/mod_h2/earlier.html

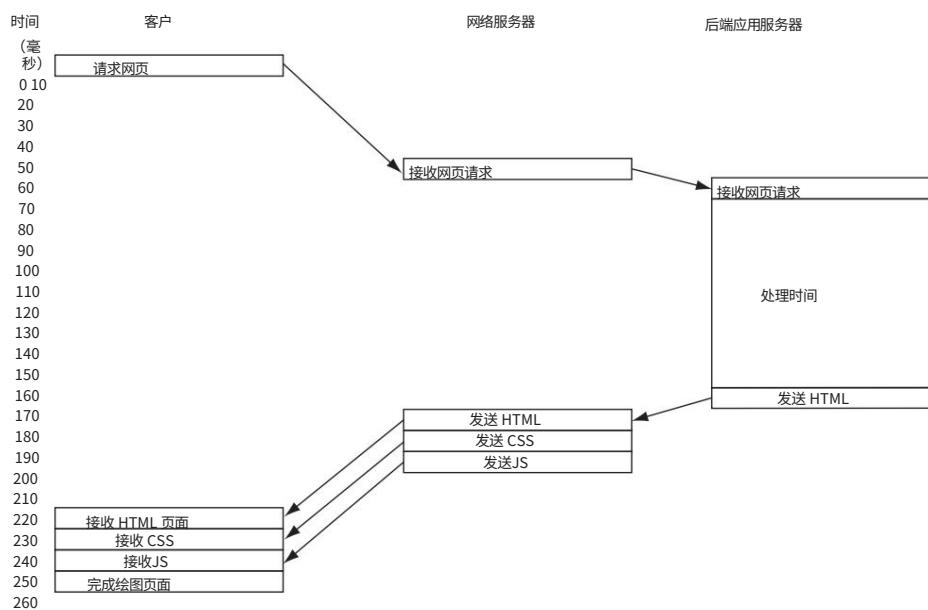


图 5.12 加载带有后端处理时间的网页

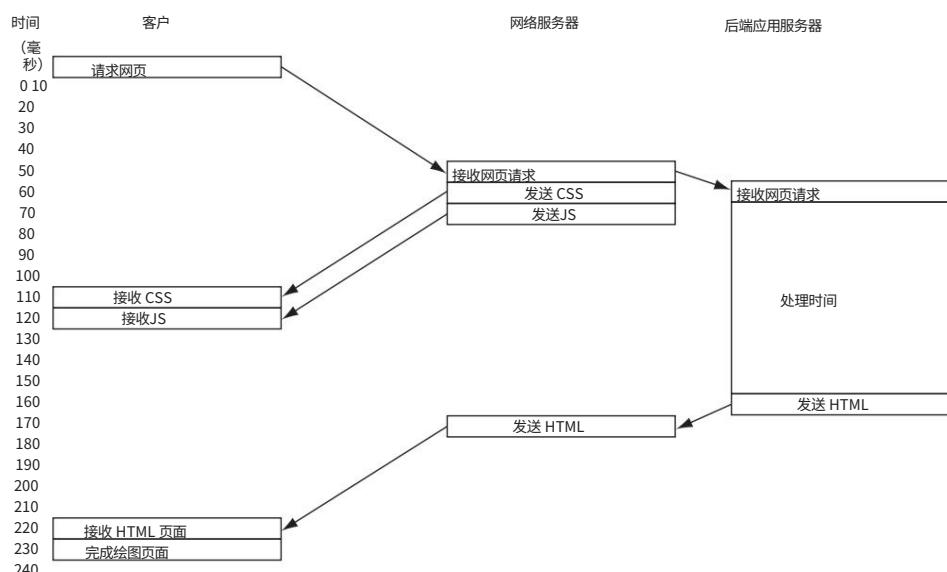


图 5.13 使用早期推送来充分利用原本浪费的时间

为了演示,更改简单的 NodeJS 服务以模拟延迟,如以下清单所示。请注意,异步/等待代码需要 NodeJS 7.10 或更高版本。

清单 5.2 具有 HTTP 链接标头和 10 毫秒延迟的节点服务

```

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

异步函数 requestHandler (请求,响应){

    console.log(request.url)

    //开始获取响应就绪 response.setHeader( Link , </
    assets/css/common.css>;rel=preload )

    //这里暂停10秒,模拟一个慢资源 await sleep(10000)

    //现在返回资源 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= text/css

    media= all href= /assets/css/common.css >\n ) response.write( <
    head>\n ) response.write( <body>\n ) response.write( <h1>测试</
    h1>\n ) response.write( </body>\n ) response.write( </html>\n )
    response.end();

}

function sleep(ms){ return new
    Promise(resolve=>{ setTimeout(resolve,ms)

    })
}

var server = http.createServer(requestHandler) server.listen(port)
console.log( 服务器正在监听
            + 端口)

```

如果您使用此代码重复 nghttp 调用并将其通过管道传输到 grep 以仅显示recv 帧行,您会在建立连接后看到 10 秒的延迟,直到
发送PUSH_PROMISE帧 (对应前面代码中的10秒休眠) :

```

$ nghttp -anv https://www.tunetheweb.com/testnodeservice/ | grep recv.*frame [0.209] recv SETTINGS frame <length=6,
flags=0x00, stream_id=0> [0.209] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0> [0.213] recv 设置frame
<length=0, flags=0x01, stream_id=0> [10.225] recv PUSH_PROMISE frame <length=73, flags=0x04, stream_id=13> [10.225] recv
HEADERS frame <length=647, flags=0x04, stream_id= 13> [10.225]接收数据帧<长度=139,标志=0x01,stream_id=13>[10.226]接收数据
帧<长度=108,标志=0x04,stream_id=2>[10.226]接收数据帧<长度=1291, 标志=0x00, stream_id=2>

```

```
[10.226]接收数据帧<长度=1291,标志=0x00,stream_id=2>[10.226]接收数据帧<长度=1291,标志=0x00,stream_id=2>[10.226]
接收数据帧<长度=1291,标志=0x00, stream_id=2> [ 10.226] recv DATA frame <length=559, flags=0x01, stream_id=2>
```

如果您将 Apache 配置更改为通过H2PushResource推送而不是等待链接标头，则推送会在 10 秒延迟之前立即发生，因为推送的资源不再被主资源占用：

```
$ nghttp -anv https://www.tunetheweb.com/testnodeservice/ | grep recv.*frame
[0.248] recv SETTINGS frame <length=6, flags=0x00,
stream_id=0> [0.248] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0> [0.253] recv 设置frame <length=0, flags=0x01,
stream_id=0> [0.253] recv PUSH_PROMISE frame <length=73, flags=0x04, stream_id=13> [0.253] recv HEADERS frame <length=675, flags=0x04,
stream_id=2> [0.253]接收数据帧<长度=1291,标志=0x00,stream_id=2>[0.253]接收数据帧<长度=1291,标志=0x00,stream_id=2>[0.253]接收数据帧<
长度=1291 ,标志=0x00,stream_id=2> [0.253]接收数据帧<长度=1291,标志=0x00,stream_id=2>[0.253]接收数据帧<长度=559,标志=0x01,stream_id=2>
[10.262] recv HEADERS frame <length=60, flags=0x04, stream_id=13> [10.262] recv DATA frame <length=139, flags=0x01, stream_id=13>
```

这个改进很有用。尽管大多数资源在理想情况下没有 10 秒的延迟（此处夸大了效果），但您推送的越早，您使用可用带宽的效率就越高，而不必在准备好稍后发送时与主请求争用。

使用 Web 服务器的早期推送命令（例如H2Push Resource）的缺点是您不再能够让应用程序启动这些推送，并且应用程序很可能处于决定是否推送的最佳位置。为解决这种情况，新的 HTTP 状态代码 103 早期提示² 允许通过预加载 HTTP 链接标头更早地指示资源的要求。与100范围内的所有状态代码一样，它是信息性的，可以忽略，但它允许仅发送标头（包括 HTTP/2 推送所需的链接标头）的早期响应，然后是标准的200响应代码。在 HTTP/1.1 世界中，这段代码看起来像是一个接一个的响应：

```
HTTP/1.1 103 早期提示链接:</assets/css/
common.css>;rel=preload;as=style

HTTP/1.1 200 OK 内容类型:
text/html 链接:</assets/css/
common.css>;rel=preload;as=style

<!DOCTYPE html>
<html>
...ETC.
```

² <https://tools.ietf.org/html/rfc8297>

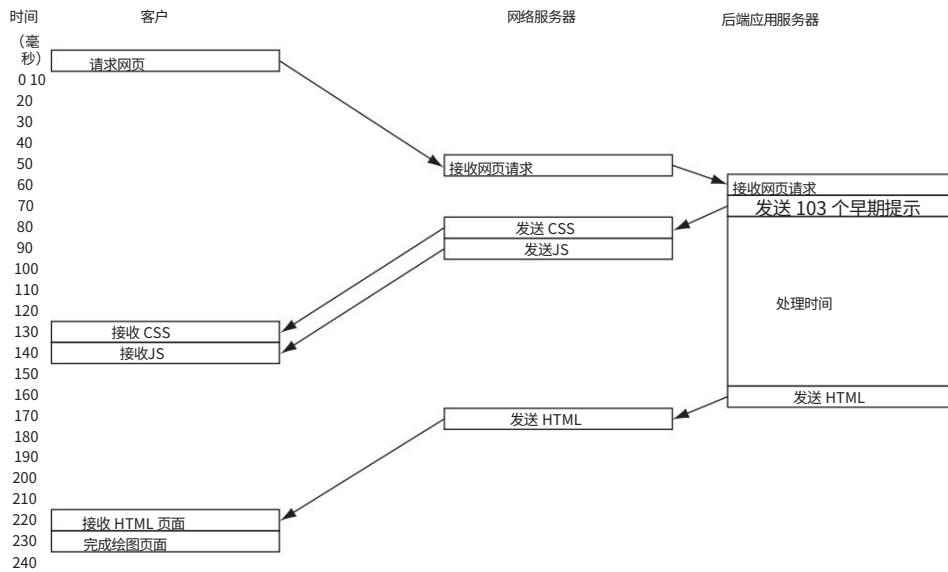


图 5.14 使用 status 103 Early Hints 告诉 web 服务器提前推送资源

图 5.14 显示了请求和响应图。

在这里，后端服务器已经发送了一个早期的 103 响应，表示页面需要 CSS 和 JavaScript。Web 服务器在等待页面本身生成时使用 HTTP/2 推送来推送这两个静态资源。之后，当页面生成并转发时，客户端可以立即使用这些推送的资源。您可以在网页到达后立即呈现页面，但无需在后端服务器和 Web 服务器之间拆分逻辑（也无需使用内联 hack）。

这个过程可能比将 Web 服务器配置为知道要推送哪些资源要慢。在图 5.14 中，推送可以在 60 毫秒标记处开始，如果在 Web 服务器中配置而不是通过链接标头，虽然它直到 80 毫秒才开始，但是让后端服务器控制推送的优势使得在许多情况下值得短暂的延迟。

在 ngnhttp 下，这样的场景是这样的：

```
$ ngnhttp -anv https://www.tunetheweb.com/testnodeservice/ | grep "recv.*frame"
[0.307]接收设置帧<长度=6,标志=0x00,stream_id=0>[0.307]接收WINDOW_UPDATE帧<长度=4,标志=0x00,
stream_id=0>[0.307]接收设置帧<长度=0,标志=0x01,stream_id=0>[0.308] recv HEADERS frame <length=60,flags=0x04,
stream_id=13> [0.308] recv PUSH_PROMISE frame <length=73,flags=0x04, stream_id=13> [0.309] recv HEADERS frame
<length=675,flags=0x04,stream_id=2> [0.309] recv DATA frame <length=1291,flags=0x00,stream_id=2> [0.310] recv DATA
frame <length=1291,flags=0x00,stream_id=2>
```

```
[0.310]接收数据帧<长度=1291,标志=0x00, stream_id=2> [0.310]接收数据帧<长度=1291,标志=0x00,
stream_id=2> [0.310]接收数据帧<长度=559,标志=0x01, stream_id=2> [10.317] recv HEADERS frame
<length=60, flags=0x04, stream_id=13> [10.317] recv DATA frame <length=1291, flags=0x01,
stream_id=13> [10.317] recv DATA frame <length=1291, flags=0x00, stream_id=13> [10.318] recv DATA
frame <length=1291, flags=0x00, stream_id=13> [10.318] recv DATA frame <length=1291, flags=0x00,
stream_id= 13> [10.318]接收数据帧<长度=1291,标志=0x00, stream_id=13> [10.318]接收数据帧<长度=300,
标志=0x01, stream_id=13>
```

初始设置后,您会看到以下内容： 103响应首先作为流

13 上的HEADERS帧在 0.308-
第二个标记。

PUSH_PROMISE帧（也在流 13 上）警告客户端推送即将到来。 资源被推送到一个HEADERS帧和几个
DATA帧中,全部发送到
流 2 在 0.309 和 0.310 秒标记处。

在人为的 10 秒延迟后处理完真正的响应后,它会作为一个HEADERS帧发回,随后在 10.317 秒处发送一个或
多个DATA帧。

在撰写本文时,对103 Early Hints（新的）的支持是有限的。例如,Node 本身并不支持它³,但您可以通过使用第三
方库⁴或将原始 HTTP 写入流来添加此支持,这正是第三方库所做的。

Apache 支持处理103响应,并会处理其中的任何链接头以推送资源,但它有意不将103响应转发给浏览器,因为某些
浏览器不支持这些响应并感到困惑。可以使用H2EarlyHints⁵指令启用转发。

支持也很有限,因为它涉及对一个请求发送多个响应。尽管此行为对于100范围内的响应是有效的 HTTP ,但与
其他 HTTP 响应相比它是不寻常的,因为它是最终响应之外的额外响应。并非每个 HTTP 实现都能很好地处理这个
额外的响应,并且可能错误地期望只有一个响应返回到 HTTP 请求。 100范围内的其他状态码（例如100
Continue、 101 Switching Protocols和102 Processing）仅用于特定场景,例如切换到 WebSockets。

许多工具和库允许您设置不同的状态代码,甚至是那些在创建这些工具时不知道的状态代码,但很少有人能够正确
处理两个请求以手动设置响应代码所需的103响应代码。当支持不可避免地出现时,这个响应代码将被证明是有用
的,我希望它的使用会迅速增长。

³ 已添加状态代码但未添加使用它的选项：<https://github.com/nodejs/node/pull/16644>。<https://www.npmjs.com/package/early-hints>

⁴

⁵

5.2.5 其他方式推送

Web 服务器并不是启用推送的唯一方式;一些后端应用程序服务器还允许开发人员以编程方式推送。以下清单显示了如何使用推送创建一个简单的 NodeJS 服务器。此清单需要http2模块,这需要NodeJS v9 或更高版本。

清单 5.3 带服务器推送的节点服务

使用严格

```
const fs = require('fs') const http2 =
require('http2')

常数端口=8443

//使用 HTTPS 证书和密钥创建 HTTP/2 服务器 const server = http2.createSecureServer({ cert:
fs.readFileSync('server.crt'), key: fs.readFileSync('server.key') })

//处理任何传入的流 server.on('stream', (stream,
headers) => {

//检查传入流是否支持连接级别的推送 if (stream.session.remoteSettings.enablePush) {

//如果支持推送,则推送CSS文件 console.log('Push enabled. Pushing CSS
file')

//打开文件读取 const cssFile = fs.openSync(
'www/htdocs/assets/css/common.css', 'r')

//获取 HTTP 响应头文件的一些统计信息 const cssStat = fs.fstatSync(cssFile) const cssRespHeaders = {

content-length : cssStat.size, last-modified :
cssStat.mtime.toUTCString(), content-type : 'text/css'

}

//为文件发送一个 Push Promise 流 stream.pushStream({
:path : '/assets/css/common.css'}, (err, pushStream, headers) => {

//在新创建的 pushStream 中推送文件 pushStream.respondWithFD(cssFile,
cssRespHeaders) }) } else { //如果推送被禁用,记录 console.log('Push disabled.') }

})
```

```
//响应原始请求 stream.respond({ content-
type : text/html , :status :200

})

stream.write( <DOCTYPE html><html><head> )
stream.write( <link rel= stylesheet type= text/css media= all
ref= /assets/css/common.css > )
stream.write( </head><body><h1>测试</h1></body></html> )})

//启动服务器监听给定端口上的请求 server.listen(PORT) console.log(` Server listening on $ {PORT}`)

})
```

此代码允许 NodeJS 将资产推送到您的浏览器。这个简单的示例仅通过 HTTPS 支持 HTTP/2。对于真实的服务器，您可能还应该启用 HTTP/1.1 和非加密的 HTTP 访问（这说明了为什么在 Node 等应用程序服务器前面使用 Web 服务器通常更容易）。其他编程语言（如 ASP.NET 和 Java）也有类似地推送资源的方法。

一路过关斩将？

正如我之前提到的，通常将负载均衡器或 Web 服务器作为系统的入口点（通常称为边缘服务器），然后将请求代理到后端应用程序服务器或服务。事实上，我推荐这种设置，因为 Web 服务器通常比动态应用程序服务器性能更高、更安全。谈到 HTTP/2 推送时，您可能认为最好在整个基础架构中使用 HTTP/2，这样您就可以，例如，通过 Web 服务器将资源从应用程序基础架构推送到浏览器。然而，当涉及中间人时，这个过程通常会导致额外的并发症。

如果应用服务器和边缘服务器支持推送，但客户端不支持推送，反之亦然？您如何处理三个（或更多）玩家之间推送资源的跟踪？

HTTP/2 规范声明 a

中介可以从服务器接收推送并选择不将它们转发给客户端。换句话说，如何使用推送的信息取决于中介。同样，中介可能会选择向客户端进行额外的推送，而服务器不采取任何操作。

事实上，让边缘服务器处理推送并使用 HTTP 链接标头（有或没有 103 早期提示）来做到这一点要容易得多。有时，应用服务器会告诉 Web 服务器推送一个资源，然后它必须从应用服务器获取资源，这似乎是一个迂回的过程，但它更简单，并且允许应用服务器推送它们不需要的资源 t 控制（例如存储在 Web 服务器层的静态文件和媒体）。

在撰写本文时,我不知道有任何 Web 服务器允许 HTTP/2 一直推送。Apache HTTP/2 代理模块 (mod_proxy_http2)是在撰写本文时存在的后端 HTTP/2 连接的少数实现之一;它明确地关闭了后端连接的推送,使用设置框架来防止并发症。^a

回顾第 3 章中讨论的 HTTP/2 基础设施设置选项,在 Web 和代理服务器上缺乏后端 HTTP/2 支持是整个基础设施支持 HTTP/2 可能没有必要甚至有益的另一个原因 至少在 HTTP/2 支持无处不在之前,没有理由不支持它。

^a <https://httpwg.org/specs/rfc7540.html#PushResources>

^b https://github.com/icing/mod_h2/issues/154

5.3 HTTP/2 推送如何在浏览器中工作 无论您如何在服务器

端推送资源,浏览器处理此过程的方式都与您预期的不同。资源不是直接推送到网页,而是推送到缓存。网页按正常方式处理。

当页面看到它需要的资源时,它会检查缓存,在那里找到它,然后从缓存中加载它,而不是从服务器请求它。

细节因浏览器而异,在 HTTP/2 规范中没有详细说明,但大多数浏览器似乎都实现了 HTTP/2 推送,并带有特殊的 HTTP/2 推送缓存,这与大多数 Web 开发人员熟悉的普通 HTTP 缓存不同。^c撰写本文时最好的文档是 Jake Archibald^d (来自 Google Chrome 团队)的一篇博客文章,内容是关于尝试使用 HTTP/2 推送以查看每个浏览器如何处理它。这篇文章详细介绍了 HTTP/2 在理论上应该如何工作以及它在实践中通常如何以意想不到的方式工作。由于他的工作,出现了一些错误,有些在撰写本文时已修复,有些尚未修复。

HTTP/2 推送是一个新概念,必须做一些工作来解决浏览器端(也可能是服务器端)的所有实施问题。我会尽力强调一些主要错误,但可能会引入新错误。

5.3.1 查看推送缓存的工作原理

推送的资源保存在一个单独的内存位 (HTTP/2 推送缓存)中等待浏览器请求它们,此时它们被加载到页面中;如果设置了缓存标头,它们也会像往常一样保存在浏览器的 HTTP 缓存中以供以后重用。一个值得注意的例外是基于 Chromium 的浏览器 (Chrome 和 Opera)不会为不受信任的证书 (例如使用红色挂锁的自签名证书)缓存资源。即使点击了证书错误,缓存仍然没有被使用

^c <https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/>

through.⁷要使用 HTTP/2 推送,您必须通过使用真实证书或在浏览器的信任库中接受您的自签名证书来获得完整的绿色挂锁;否则,推送的资源将被忽略。⁸推送缓存不是浏览器查找资源的第一个地方。

虽然这个过程是特定于浏览器的,但实验表明,如果资源在通常的HTTP 缓存中可用,浏览器将不会使用推送的资源。即使推送的资源比缓存的资源更新,只要浏览器认为缓存的资源可用(基于缓存控制标头),浏览器仍然倾向于使用其较旧的缓存内容。在使用它们的网站的推送缓存之前,还会检查服务工作者。您可以很容易地浪费资源来推动不会被使用的资源。图 5.15 显示了加载资源时发生的情况,以及在 Chrome 浏览器中为页面所需的每个资源检查哪些缓存。

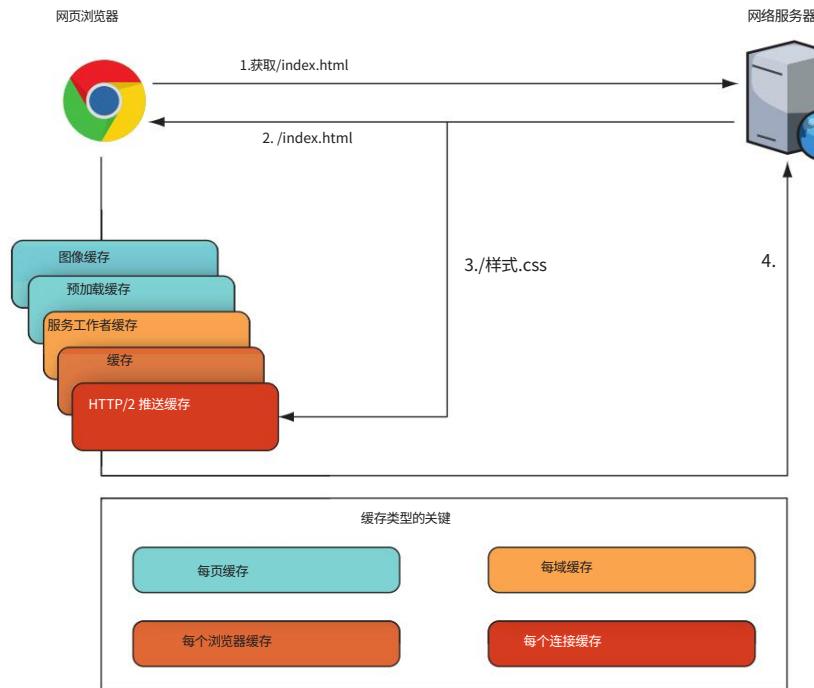


图 5.15 与 HTTP/2 推送的浏览器交互

⁷ <https://bugs.chromium.org/p/chromium/issues/detail?id=103875#c8>
⁸ <https://bugs.chromium.org/p/chromium/issues/detail?id=824988>

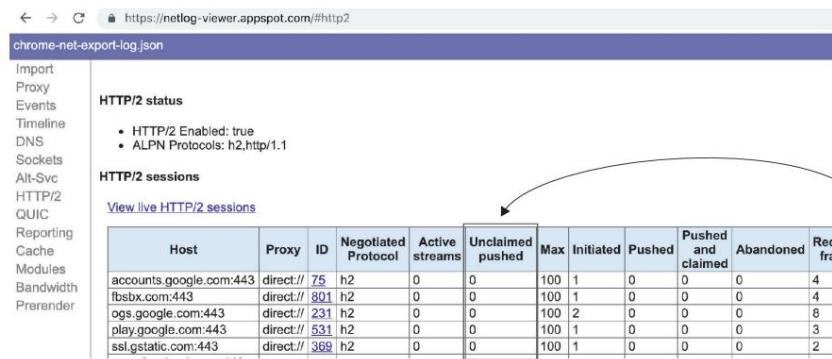
当页面被请求 (1) 并返回 (2) 时,任何推送的资源都会放入 HTTP/2 推送缓存 (3),并在向 Web 服务器发出请求 (4) 之前按顺序检查缓存。以下是每个缓存的简要说明:

图像缓存是该页面的短期内存缓存,例如,如果图像在页面上被引用两次,它可以防止浏览器获取两次图像。当用户离开页面时,缓存被销毁。预加载缓存是另一种短暂的内存缓存,用于保存预加载的资源(参见第 6 章)。同样,此缓存是特定于页面的。不要为另一个页面预加载一些东西,因为它不会被使用。服务工作者是相当新的后台应用程序,独立于网页运行,充当网页和网站的中间人。例如,即使您失去网络连接,它们也能让网站更像本地应用程序。他们有自己的缓存链接到域。HTTP 缓存是大多数开发人员都知道的主要缓存,是跨浏览器共享的基于磁盘的持久缓存,其大小有限,可用于所有域。

HTTP/2 推送缓存是一个短暂的内存缓存,绑定到连接并最后检查。

当服务器推送 styles.css 时,它会被推送到 HTTP/2 推送缓存中。当 Web 浏览器决定它需要 styles.css 时,它不知道(或关心)服务器已经推送了该资源,并且它在向源发出网络请求之前按顺序检查所有缓存。如果主 HTTP 缓存中存在有效的 styles.css,浏览器会从该缓存中获取它,即使更新的副本位于 HTTP/2 推送缓存中。

使用 4.3.1 节中讨论的 chrome://net-export 工具,您会看到所有当前活动页面的未认领推送资源的摘要,如图 5.16 所示。



The screenshot shows the chrome://net-export-log.json netlog viewer interface. On the left, there's a sidebar with various network-related tabs like Import, Proxy, Events, Timeline, DNS, Sockets, Alt-Svc, HTTP/2, QUIC, Reporting, Cache, Modules, Bandwidth, and Prerender. The main area has two sections: 'HTTP/2 status' and 'HTTP/2 sessions'. The 'HTTP/2 sessions' section contains a table with columns: Host, Proxy, ID, Negotiated Protocol, Active streams, Unclaimed pushed, Max, Initiated, Pushed, Pushed and claimed, Abandoned, and Rec fra. There are five rows of data corresponding to different hosts. A callout arrow points from the text '域未认领的推送资源(推送但未被页面使用的资源)将显示在该列中。' to the 'Unclaimed pushed' column.

Host	Proxy	ID	Negotiated Protocol	Active streams	Unclaimed pushed	Max	Initiated	Pushed	Pushed and claimed	Abandoned	Rec fra
accounts.google.com:443	direct://	75	h2	0	0	100	1	0	0	0	4
fbbsx.com:443	direct://	801	h2	0	0	100	1	0	0	0	4
ogs.google.com:443	direct://	231	h2	0	0	100	2	0	0	0	8
play.google.com:443	direct://	531	h2	0	0	100	1	0	0	0	3
ssl.gstatic.com:443	direct://	369	h2	0	0	100	1	0	0	0	2

图 5.16 Chrome 中无人认领的推送资源跟踪

HTTP/2 推送缓存绑定到连接的事实也意味着如果不使用连接,则不会使用推送的资源。此过程不同于大多数开发人员习惯使用的 HTTP 缓存,并导致一些

有趣的考虑。首先,如果连接丢失,推送缓存和任何未使用的推送资源也会丢失(因此您在推送它们时浪费了资源)。如果使用另一个连接,则可能无法使用推送的资源。对于 HTTP/2,应该只有一个连接,因此您可能认为这种情况不是什么大问题,但浏览器之间的情况可能会有所不同,浏览器可能会决定以不同的方式实现功能。早些时候,我谈到了非认证请求,大多数浏览器使用单独的连接处理这些请求。但是,Web 超文本应用程序技术工作组 (WHATWG) 正在讨论进行更改以允许同一连接用于有凭证和无凭证的连接

requests.⁹一个结果是你不能推送跨源字体(那些从另一个域加载的字体,包括分片域),因为它们必须是未经授权的请求。此外,单独的选项卡或浏览器进程可能会启动单独的连接,具体取决于浏览器:Chrome 和 Firefox 跨选项卡共享连接;边缘没有; Safari 似乎甚至在同一个选项卡中打开多个连接。¹⁰由于 HTTP/2 推送缓存处于连接级别而不是页面级别,因此可以为未来的页面导航推送资产,但此缓存的短暂性,再加上连接可能会断开这一事实,使这个想法变得糟糕。

最后,当从连接的推送缓存中“认领”资产时,它会被删除并且不能再次从推送缓存中加载,尽管如果设置了 HTTP 缓存控制标头,它可以从浏览器的 HTTP 缓存中使用。推送缓存与 HTTP 缓存的不同之处还在于,不可缓存的资源(那些设置有no-cache和no-store HTTP缓存控制标头的资源)可以从推送缓存中推送和读取。它不是传统意义上的缓存,而是请求的存放区。Web 性能架构师 Yoav Weiss 将其称为“无人认领的推送流容器”¹¹,但他承认这个术语不如推送缓存那么吸引人。

5.3.2 使用 RST_STREAM 拒绝推送

客户端可以通过在推送流上发送带有CANCEL或REFUSED_STREAM代码的RST_STREAM帧来拒绝推送的资源。可以使用此框架,因为浏览器已经有正在推送的项目或出于某些其他原因(例如用户在页面仍在加载时浏览离开页面,因此浏览器不再需要该项目)。

这个过程听起来像是一种防止过度推送浏览器不需要的资源的好方法。问题是将这个RST_STREAM帧发送回服务器需要时间,同时,服务器继续发送HEADERS和DATA帧,浏览器将丢弃这些帧。RST_STREAM帧是一个控制信号,不像断开连接那样激进,在 HTTP/2 中不能在不中断该连接上的其他流的情况下完成。整个

⁹ <https://github.com/whatwg/fetch/issues/341>

¹⁰ https://bugs.webkit.org/show_bug.cgi?id=172639

¹¹ <https://blog.yoav.ws/tale-of-four-caches/>

资源可能在RST_STREAM帧被接收并被服务器停止发送资源之前发送。

只有当浏览器意识到它不需要被推送的资源时,RST_STREAM 框架才有用。如果浏览器的 HTTP 缓存中已有资源,则很明显可以使用RST_STREAM帧来停止推送。但是,如果推送了一张巨大的图像但页面上从未引用过怎么办?页面可能已更新为不再使用该图像,但推送指令可能尚未更新。

浏览器不会知道不需要图像,并且会很高兴地接收整个资源,但永远不会使用它。根据浏览器工具的不同,您甚至可能不知道自己在不必要地发送此图像,因为它可能不会出现在开发人员工具的网络选项卡上。

总而言之, RST_STREAM帧是停止流的有用方法,尤其是推送的资源流,但您不应该依赖它作为控制不正确推送的资源的方法。过度使用资源是一种资源浪费,但即使您的服务器可以处理它,请记住带宽不是免费的。尤其是移动连接通常会限制带宽,因此如果您发送不必要的资源,访问者将花费更多,更不用说浪费的带宽是本可以更好地用于发送页面确实需要的资源的带宽。

5.4 如何有条件地推送

使用 HTTP/2 推送的一大风险是不必要地推送资源。一些风险是由于本章前面讨论的实现问题引起的（例如,如果连接未被重用）,但主要是问题在于推送浏览器已有的资源。

例如,如果您认为推送样式表是个好主意,您可以缩短第一个请求的页面加载时间。但是,如果您在访问者浏览您的站点时继续为每个页面请求推送样式表,那么您就不需要推送用户已经拥有的资源（假设您使用良好的缓存控制标头来确保资源被缓存）。

由于RST_STREAM框架在停止推送资产而不浪费资源方面效率低下,您可以使用哪些其他方法来确保您不会推送客户端不会使用的资源?

5.4.1 服务器端跟踪推送

服务器可以跟踪它在特定客户端连接上推送的资产。

该技术将取决于服务器,但可以基于连接或会话 ID。例如,每次推送资源时,服务器都会标记此连接/会话不应再次推送该资源,即使被要求也是如此。这个进程就是 Apache 使用的,也是为什么你必须在测试时关闭H2PushDiarySize设置的原因。此功能可以在 Web 应用程序中实现,而不是在 Web 服务器软件中实现,以便为 Web 开发人员提供更多控制权。

缺点是服务器正在对是否应该推送进行有根据的猜测。例如,如果浏览器缓存已被清除,资源将不可用,但服务器仍然不会推送。此外,繁忙的服务器在跟踪推送的资源方面可能有资源限制,如果同一台服务器并不总是为客户端提供服务,则负载平衡的服务器可能无法了解已推送的内容的完整信息。

最终,这个过程很复杂,这种向无状态 HTTP 协议添加状态的粗略尝试可能不是最好的方法。HTTP/2 将状态的概念添加到协议的其他部分 (HPACK 标头压缩和流状态,如第 7 章和第 8 章中所讨论),因此也许这个问题可以而且应该在协议级别解决以获得更好的实现。我在 5.4.4 节讨论了一个这样的建议 (缓存摘要),但首先,我看看现在可以使用的其他方法。

5.4.2 使用 HTTP 条件请求

如果客户端发送 if-modified-since 或 etag HTTP 标头,则此页面已在浏览器的缓存中但已过期。如果你通常推送 CSS 资产,当你在请求中看到这样的标头时,你可以选择不推送它,因为样式表也可能被缓存 (可能比引用它的页面更长,这是经常发生的情况)。此过程比跟踪此服务器端更简单,但有许多相同的缺点,例如服务器正在对客户端的内容进行有根据的猜测以及导航到使用已缓存样式表的另一个页面的情况。

5.4.3 使用基于 cookie 的推送

下一个选项是记录资产已被推送到客户端的事实。

Cookie 可能是用于此目的的天然载体,也可以使用LocalStorage 或 Session Storage。

这个想法是,当你推送一个资源时,你设置了一个 cookie,该 cookie 对该会话 (短缓存资源) 有效,或者与被推送的资源 (长缓存资源) 同时有效。随着每个页面请求的到来,您检查 cookie 是否存在。如果 cookie 不存在,则资源可能不在浏览器缓存中,因此推送它并设置 cookie。如果存在 cookie,请不要推送资源。此功能可以在任何客户端应用程序甚至服务器配置中实现。这是 Apache 中的示例:

```
#检查是否有 cookie 表示 css 已经加载
#如果是,设置一个环境变量,以备后用
SetEnvIf Cookie _cssloaded=1 _cssloaded

#如果没有cookie,并且它是一个html文件,那么推送css文件#并设置一个会话级别的cookie,这样下次就不会被推送了:

<文件匹配 index.html >
    标题添加链接 "</assets/css/common.css>;as=style;rel=preload"
    env=_cssloaded
    标头添加 Set-Cookie _cssloaded=1; Path=/; Secure; HttpOnly
    env=_cssloaded </文件匹配>
```

类似的逻辑可以用任何服务器端语言实现。

该方法是对前两种策略的进一步改进。不需要在服务器端跟踪任何内容，因此逻辑不那么复杂，并且您要根据浏览器状态进行更多跟踪。但是，Cookie 与 HTTP 缓存不同。虽然您可以将过期时间设置为相同，但 cookie 可以独立重置（例如，在浏览器中关闭或以隐身模式运行），尽管对于任何服务器端跟踪方法来说都是如此。

在撰写本文时，cookie 可能是跟踪资产是否具有已被推送并且很可能在缓存中，但它们仍然存在问题。

5.4.4 使用缓存摘要缓存摘要是一项

允许浏览器通知服务器其缓存内容的提议¹²。

建立连接后，浏览器会发送一个新的CACHE_DIGEST帧，其中列出了该域（或该连接在其上具有权威性的其他域；参见第 6 章）的 HTTP 缓存中当前保存的所有资源。服务器将缓存的内容作为 URL 以及etag标头值获取，以获取 URL 的某种版本控制。这种方法比使用以前的猜测缓存内容的迂回方法要好得多，因为浏览器已经明确地告诉服务器它的缓存中有什么。服务器可以记住客户端缓存的连接内容，甚至可以在发送更多资源时更新它。

CACHE_DIGEST帧应该在连接开始时发送一次（最好在第一个请求发出后）。

缓存的内容可能很大，因此与其发送完整的 URL 和 etags，不如让客户端将它们编码在基于布谷鸟过滤器的摘要中。

我不会详细介绍布谷鸟过滤器，¹³但足以说明这些过滤器是发送缓存内容的有效方式，冲突风险很低（例如错误地暗示资源在缓存中，而实际上它不在缓存中，或相反亦然）。

在撰写本文时，缓存摘要不是批准的标准，也不是由任何浏览器生成的。有趣的是，一些服务器（例如 Apache、http2server 和 H2O）已经添加了对当前标准草案（来自 H2O 实现）的支持。因为浏览器目前不发送CACHE_DIGEST帧来初始化任何服务器端缓存，所以这些实现仅用于跟踪服务器推送的请求。这些服务器会跟踪它们发送的资源，因此它们不应将资源推送两次（即使有指示）。这个特性很有用，但如果状态也可以用浏览器缓存状态初始化，它会更有用，这需要CACHE_DIGEST框架。正如我在本节开头提到的，您在测试 HTTP/2 推送时使用以下配置关闭了此 Apache 功能：

```
H2PushDiarySize 0
```

¹² <https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest>

¹³ <https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf>

此行将最大推送日志大小设置为0,表示您没有推送日志,因此即使资源已发送也允许推送。在不将此大小设置为0的情况下,如果您多次测试一个页面,您会发现资源有时会被推送,有时不会,这在您测试 HTTP/2 推送时可能会造成混淆。完成测试后,删除此配置并使用默认的H2PushDiarySize (每个连接 256 个条目)或适当地设置它。其他服务器可能具有与 Apache 类似的实现,因此请查看您的服务器文档。

如果您在 Web 应用程序中使用服务工作者,现在可以在浏览器端使用缓存摘要的实现,因为服务工作者允许您拦截和更改您的 HTTP 请求。为此存在一些实现。¹⁴由于CACHE_DIGEST框架未在任何浏览器或服务器中获得批准或实现,因此无法使用;相反,这些实现通常在 HTTP 标头或 cookie 中发送缓存摘要。您的服务器可能会使用此 cookie 或 HTTP 标头来初始化缓存摘要。您的 Web 应用程序需要手动发送此标头 (使用服务工作者),然后使用它来初始化服务器端。虽然测试此功能可能很有趣,但如果将其标准化并由浏览器发送会更好。然而,在 2019 年 1 月,HTTP 工作组表示他们此时不会继续标准化缓存摘要。¹⁵关于缓存摘要的最后一个问题是安全性。浏览器缓存可能包含敏感信息,例如之前访问过哪些 URL,或者可能允许在不使用 cookie 等的情况下对用户进行指纹识别。服务器可能无论如何都可以访问其中的一些数据 (必须在某个时候向服务器发出请求),但安全性仍然是一个问题。当前的草案建议浏览器在隐私模式下或在未使用或清除 cookie 时不发送缓存摘要。安全和隐私问题是此时停止缓存摘要标准化的另一个原因。

5.5 推送什么

到目前为止,您应该对 HTTP/2 推送及其在服务器端和客户端的工作方式有了很好的了解。但是你需要仔细考虑你推送的资产。

5.5.1 你能推什么?

该规范规定了 HTTP/2 推送的一些基本规则:¹⁶客户端可以通过在 SETTINGS

框架中将SETTINGS_ENABLE_PUSH选项设置为0来禁用推送。此后,服务器不得使用 PUSH_PROMISE 帧。推送的请求必须是可缓存的方法 (GET、HEAD 和一些 POST 请求)。推送的请求必须是安全的 (通常是 GET 或 HEAD)。

¹⁴ <https://www.npmjs.com/package/cache-digest-immutable>

¹⁵ <https://lists.w3.org/Archives/Public/ietf-http-wg/2019JanMar/0033.html>

¹⁶ <https://httpwg.org/规格/rfc7540.html#PushResources>

推送请求不得包含请求主体（尽管它们通常包含响应主体）。推送的请求必须只发送到服务器所在的域

迭代的。

客户无法推送；只有服务器可以推送。只能在响应当前请求时推送资源。这是不可能的
如果没有请求在处理中，服务器启动推送。

实际上，由于这些规则，只会推送GET请求。前面的规则是关于你可以推送什么的，但是你需要更多地考虑你应该推送什么。

权限限制限制您只能推送Web服务器服务的资源（直接或间接）。如果该网站使用从getbootstrap.com加载的Bootstrap（或者如果它使用托管在jquery.com或类似网站上的jQuery），则您的服务器无法推送。如果你想通过你的服务器代理这些请求，你可以，但你需要更新所有引用以期望来自你的服务器的请求，在这一点上，为什么不在本地托管页面并消除代理它的复杂性？

一个名为 Signed HTTP Exchanges¹⁷（正式名称为 Web Packaging）的有趣提案将允许您从您的域中提供签名资源，就好像它们直接来自原始域一样，从而使您能够有效地推送其他域的资源。然而，该提案仍在定义中，并且在撰写本文时尚未在任何浏览器或服务器中可用，但它确实值得关注。

5.5.2 你应该推送什么？

想要使用HTTP/2推送的网站所有者必须回答的一个关键问题是推送哪些资产。也许更重要的是，哪些不推送。HTTP/2推送旨在进行性能优化，但如果推送过多并浪费重要带宽推送客户端不会使用的资产，而不是使用可用资源下载页面资产，它可能会成为性能拖累将使用。

理想情况下，您应该只推送页面需要的关键资产。推送不会被使用的资源是一种资源浪费。不会使用的资源包括资产

未使用的资产（例如未引用的资产）、客户端无法使用的资产（例如该客户端不支持的图像格式）以及根据客户端可能不使用的资产（例如图像仅针对特定屏幕尺寸显示）。我之前说过，只应推送关键资源。尽管推送页面所需的所有内容可能很诱人，但您可能会减慢关键资源的交付速度，具体取决于推送的资源与其他客户端发起的请求相比如何确定优先级。

¹⁷ <https://tools.ietf.org/html/draft-yasskin-http-origin-signed-responses>

此外,您需要考虑客户端是否已经在其缓存之一中拥有资产(请参阅第5.3节)。仅当推送的资产很可能尚未缓存时,才应推送。

您应该使用HTTP/2推送来充分利用空闲网络时间。因此,推送页面所需的所有资源不太可能提高性能,因为您正在覆盖浏览器可能制定的任何加载优先级。Chrome团队撰写了一篇关于推送内容的深入论文,¹⁸其中主要建议之一是推送“填补空闲网络时间,仅此而已”所需的最低限度。

其他研究¹⁹表明类似的保守策略应该与推送一起使用。因此,早期推送和103状态代码是对基本推送策略的重要改进。

简而言之,少压总比多压好。如果不推送资源,可能发生的最坏情况是无论如何都需要请求它,并且页面可能不会像在最佳条件下那样得到改善。另一方面,过度推送可能发生的最坏情况是发送不必要的资产,从而浪费客户端、网络和服务器上的资源,并使页面加载速度变慢。但是HTTP/2推送,即使过度推送,也不应该破坏页面。该页面不会像它应该的那样高效,并且可能会浪费资源(这并非没有成本),但页面本身会加载 - 最终。

5.5.3 自动推送

您还需要制定有关推送内容的策略。网站所有者或开发人员是否必须决定推送什么(可能是每个页面),然后在服务器中配置此决定?或者该过程应该更加自动化?Jetty²⁰是一个Java servlet引擎,它选择第二个选项并尝试自动推送。²¹它使用该请求的Referer标头监视请求和后续请求。然后它使用它所看到的内容来为来自其他客户端的类似未来请求构建建议的推送资源。这个引擎当然消除了决定推送什么的很多复杂性,但是你取决于你是否同意该实施以及它是否适合你的网站。决定推送什么很复杂,每个站点的自动推送也同样复杂。Jetty的实现很有趣,加上某种形式的缓存摘要以防止过度推送,这可能就足够了。或者,网站所有者可能想要更直接的控制,因为他们应该更好地了解他们的网站和访问者,并且应该能够更好地了解要推送的内容。

¹⁸ <https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0/>也可在<https://goo.gl/89RLGQ>上获得<https://calendar.perfplanet.com/2016/http2-push-the-details/>

¹⁹ <https://www.eclipse.org/jetty/documentation/current/http2-configuring-push.html>

²⁰

²¹

5.6 HTTP/2 推送故障排除

HTTP/2 推送最容易在 Chrome 开发者工具（或类似的基于 Chromium 的浏览器，如 Opera）的网络选项卡的启动器列中看到。

但是，如果您在此列中没有看到推送的资源怎么办？这里有一些常见原因：

你在使用 HTTP/2 吗？如果没有，推送将不起作用。添加协议列以确保您使用的是 HTTP/2。请参阅第 3 章，了解有关何时未使用 HTTP/2 的故障排除提示，即使您希望它使用。

您的服务器是否支持 HTTP/2 推送？在撰写本文时，某些服务器和 CDN 不支持 HTTP/2 推送。不幸的是，客户端声明它是否支持在 SETTINGS 框架中推送，而不是服务器，因此无法通过使用 nghttp 或 Chrome 的 net-export 页面查看 SETTINGS 框架来查看服务器是否支持它。

您的服务器是否落后于其他基础设施？如果您的服务器位于负载均衡器或其他终止 HTTP/2 连接的基础设施之后，它可能不支持 HTTP/2 推送，即使您的服务器支持。即使它确实支持 HTTP/2 推送，它也可能不会传递需要由该边缘基础设施推送的推送资源。

如果你在后端应用服务器（比如 Node 或者 Jetty）前面有 Apache，Apache 不会让后端服务器自己推送资源，它必须使用 HTTP 链接头来让 Apache 推送资源。

资产是否被服务器推送？您可以使用 nghttp 检查实际帧以调查是否正在发送 PUSH_PROMISE 帧和资产本身，以确定问题是否是浏览器问题。

页面是否需要资产？如果页面不需要资产，浏览器将不会使用它们，Chrome 甚至不会在“网络”选项卡上显示它们。您可以使用 chrome net-export 工具查看所有当前活动页面的未认领推送资源的摘要，如图 5.16 所示并在 5.3.1 节中讨论。

但是，如果您使用带有 rel=preload 和 as 属性的 HTTP 链接标头进行推送，Chrome 会认为页面需要这些资源（通过预加载提示）并将它们显示在“网络”选项卡上。这种情况既有用又令人困惑。

一种调试方法是从链接标头中删除 as 属性（例如 as=style）。Chrome 不会将资产用作预加载提示，但您的 Web 服务器仍应推送它们（因为 as 属性对于推送而言不是强制性的，具体取决于实现）。如果资源没有出现在 Network 选项卡上，但在它具有 as 属性时出现，则您知道您正在推送页面不需要的资源。您是否使用正确的方式为您的服务器推送？您如何推送取决于服务器。许多服务器使用 HTTP 链接标头，但并非所有服务器都使用，因此您

不能假设他们会使用这种方法来推送。查看您的服务器文档或在线指南，了解有关如何在您的服务器上推送的详细信息。服务器是否明确决定不推送资源？如果你已经实现了缓存感知推送（参见 5.4 节）或其他一些仅在特定情况下推送的方法，该实现可以解释为什么推送没有发生。如果刷新页面或重新启动浏览器（甚至服务器）导致资产有时显示为被推送，请检查推送资源何时设置为发送。对于 Apache，可以将 H2PushDiarySize 设置为 0 以关闭推送日记功能，该功能试图阻止推送服务器认为客户端已经拥有的资源。当您希望服务器多次推送同一资源时，此功能对于调试很有用。推送的资产是否存在？指定要推送的资源时很容易出错，如果资源不存在，则无法推送！推送不存在的资源会导致 Web 服务器日志中出现 404（未找到）状态代码。同样，如果您使用的是 nginx，此 404 状态代码会出现在返回的框架中。如果您使用带有 rel=preload 和 as 属性的 HTTP 链接标头，它会在“网络”选项卡上显示为 404 或已取消的 22 请求。

您是否正在推送与浏览器期望的连接不同的连接？如 5.2.1 节所述，HTTP/2 推送链接到连接。如果您在一个连接上推送，但浏览器希望资源在另一个连接上发送，则浏览器不会使用推送的资源。字体是最明显的问题，因为它们必须在未经授权的连接上加载，但一些浏览器问题和怪癖可能会导致使用不同的连接。WebPageTest 中的连接视图可能是查看这种情况的最佳方式。

您使用的是自签名证书还是不受信任的证书？Chrome 会忽略对不受信任的 HTTPS 证书（包括为本地主机创建的自签名虚拟证书）的推送请求。²² 您必须将证书添加到计算机的信任库中，以便为要使用的推送资源启用绿色挂锁。Chrome 还坚持要求证书具有有效的主题备用名称（SAN）。

许多关于创建自签名证书的教程只包含较旧的主题字段，因此即使将它们添加到信任库中，这些证书也不会被识别；它们必须被同时具有主题和 SAN 的证书替换才能推送工作。

²² <https://bugs.chromium.org/p/chromium/issues/detail?id=811077>

²³ <https://bugs.chromium.org/p/chromium/issues/detail?id=824988>

HTTP/2 推送的性能影响

5.7 HTTP/2 推送的性能影响 HTTP/2 推送的影响因网站而异,取决于往返时间 服务资源所需的时间 以及网站的优化程度。

目前,很少有站点使用 HTTP/2 推送,因此关于其性能影响的有意义的信息很少见。

有效使用 HTTP/2 推送的关键是在未使用连接时利用带宽间隙。对于需要很长时间生成服务器端的页面,收益会很大。对于静态页面,收益不太明显。尽管存在潜在的单程节省,但由于带宽和处理限制,推送的资源可能会以任何方式排队在更高优先级的主要资源后面,从而减少请求延迟(半个往返)的收益。图 5.17 显示了这种效果。对于前四个资源,这两个瀑布看起来相同,因此使用推送几乎没有好处。

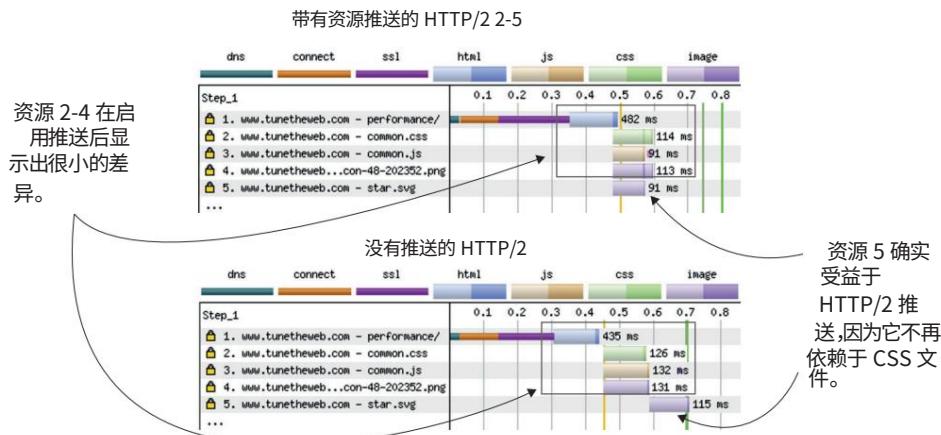


图 5.17 HTTP/2 推送的资源与请求的资源不同时到达。

第五个star.svg资源确实受益于推送;它不再需要等待样式表被下载才能发现它是必要的。(在本章后面我将讨论预加载,它提供了类似的好处)。在这个例子中,push 似乎并没有解决它的原始用例,消除了对内联资源的需求,但如果使用得当,它仍然可以提供巨大的好处。

在 2018 年 7 月于蒙特利尔举行的第 102 届 IETF 会议上,Akamai²⁴和 Chrome 团队²⁵发表了他们对 HTTP/2 推送效果的观察。Akamai 在先发制人地推送被视为关键的资源时显示出一些统计改进。Chrome 也有一些小的改进,但尝试了

²⁴ <https://github.com/httpwg/wg-materials/blob/gh-pages/ietf102/akamai-server-push.pdf>

²⁵ https://github.com/httpwg/wg-materials/blob/gh-pages/ietf102/chrome_push.pdf

禁用推送并测量差异。这些略有不同的方法提出了一些问题。Akamai 客户是否总体上更能代表 Web,因为 Chrome 只关注已经启用推送的网站（这些网站很少,而且可能由 HTTP/2 拥护者运营）？Akamai 是否决定推送正确的资产,它的决定比决定自己推送什么的网站更好或更差,就像他们在 Chrome 实验中所做的那样？

突出了两个大问题,尤其是 Chrome 团队:很少有网站使用 HTTP/2 推送（据 Chrome 称,占 HTTP/2 会话的 0.04%）,推送可能使性能变差是一个真正的问题。Chrome 团队甚至质疑如果关闭推送,是否有人会注意到。低使用率本身就说明了这一点,我在本章中展示了推送很复杂,所以很多人质疑它的用处并提出替代方案。

5.8 推送与预加载HTTP/2 推送有很多

细微差别,即使它按预期工作（情况并非总是如此）。使用 HTTP/2 推送存在明显的风险,例如浪费带宽和减慢您的网站速度而不是加速它。正如我之前提到的,风险不是网站所有者会破坏他们的页面,而是他们会浪费资源,而这些资源可能在其他地方使用得更好。主要问题之一是服务器不知道浏览器的 HTTP 缓存中有什么,如果缓存摘要成为标准化,也许缓存摘要可以解决该问题。在那之前,有些人会问 HTTP/2 推送是否已准备好用于主流用途,或者我们是否应该对预加载感到满意。

Preload²⁶是一种向浏览器指示页面需要资源的方法,而不是等待浏览器发现这一事实。如 5.1.1 节所述,您可以使用 HTTP 链接标头进行预加载（可能使用nopush属性来防止推送）：

链接：“</assets/css/common.css>;rel=preload;as=style;nopush”

在 HTML 中

```
<link rel= preload href= /assets/css/common.css as= style >
```

无论使用哪种方式,浏览器都应该以这一行为标志,去获取优先级高的资源。正如我之前提到的,与 HTTP/2 推送的一个很大区别是as属性对于预加载资源更为重要;排除它会导致预加载提示被忽略或资源被下载两次。

在请求资源之前,预加载不如 HTTP/2 推送快
它,但它是一个浏览器发起的请求,它有几个优点:

²⁶ <https://w3c.github.io/preload/>

浏览器知道它的缓存中有什么，并适当地使用这些知识来决定是否发出请求。与 HTTP/2 推送不同，预加载提示不会导致客户端已经拥有的资源的新下载。如果浏览器已经拥有该资源，它会忽略预加载提示。因为许多 HTTP/2 服务器使用 HTTP 链接头来推送资源，但是如果你不想推送资源，你应该添加一个nopush属性。使用预加载提示时，您对推送缓存的担忧和复杂性会减少，预加载提示应该被下载并拉入 HTTP 缓存。如果未使用预加载资源，您仍然在浪费时间下载它，但无论您使用预加载还是 HTTP/2 推送，该规则都适用。您还可以使用预加载从其他域加载资源，而您

只能对您自己域中的资源使用 HTTP/2 推送。

Chrome 开发者工具会显示所有预加载的请求，无论它们是否被使用，但它只显示已使用的推送请求（尽管解决方法是为每个推送资源发送预加载 HTTP 链接标头）。

这些优势目前可能已经足够了，有些人建议暂时坚持使用风险较小的预加载方法。Fastly 的 Hooman Beheshti 的分析表明，在 2018²⁷ 年 2 月（HTTP/2 正式批准后近三年），只有 0.02% 的网站使用 HTTP/2 推送，这与 Chrome 团队的分析类似，在 5.7 节中有介绍。有些人对使用这项技术犹豫不决，并且有充分的理由，特别是如果预加载资源提示提供大部分相同的好处而风险要小得多。

将预加载与新的 103 HTTP 状态代码一起使用会使预加载和 HTTP/2 推送之间的性能差距更加拉近，因为需要一段时间才能加载的资源可以更早地发送 103 响应以及预加载 HTTP 链接标头，并告诉浏览器开始请求他们。资源在需要时可能已经存在，具体取决于页面和生成所需的时间。在本章前面的 5.2.4 节中，我讨论了 103 响应与 HTTP/2 推送的结合如何让您使用处理时间（当网络空闲时）主动推送资源。为了避免您返回，我将在图 5.18 中重复之前的图表。

图中显示后端应用服务器可以使用 103 Early Hints 响应告诉 Web 服务器在处理过程中推送资源以生成请求的网页。在这种情况下，由于 103 响应仅用于告诉 Web 服务器推送一些响应，它可能会被 Web 服务器吞下，因为它发送到客户端并没有真正的好处（而且，如前所述，有些浏览器不能很好地处理 103 响应）。

如果这个推送选项不是你想要实现的，出于本章讨论的所有原因，风险较小的选项可能不使用 HTTP/2 推送，而是将 103 响应发送回浏览器（当浏览器支持时到达）。然后

²⁷ <https://www.youtube.com/watch?v=wR1gF5Lhcq0>

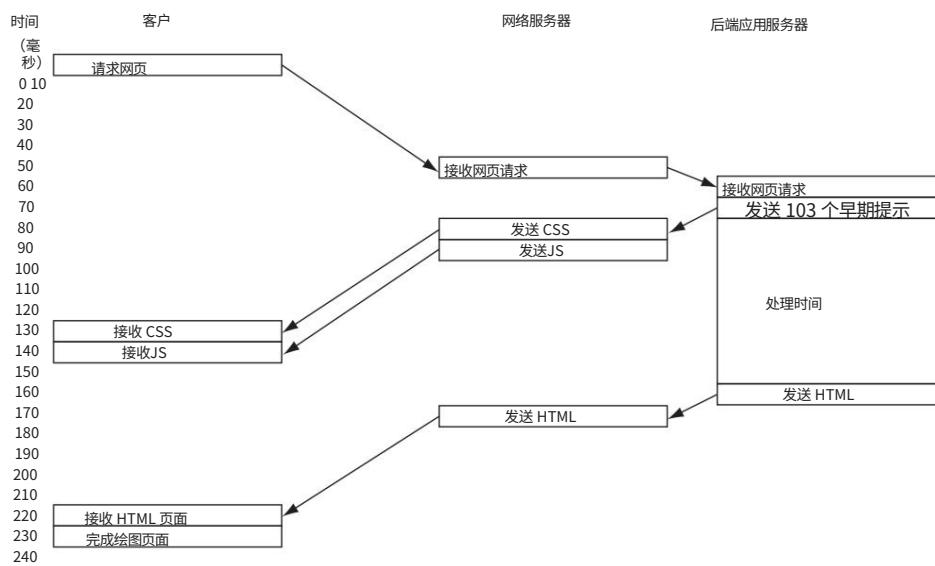


图 5.18 使用 status 103 Early Hints告诉 web 服务器提前推送资源

浏览器可以使用 preload HTTP 链接头来预加载资源,如图 5.19 所示。

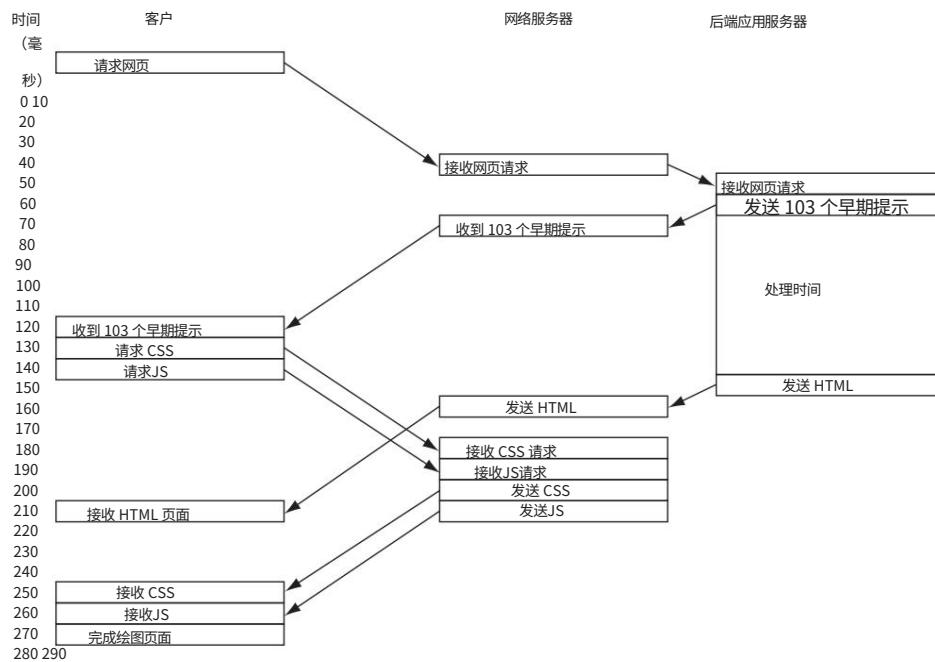


图 5.19 使用带有预加载标头的103而不是 HTTP/2 推送

这个过程没有推送那么快,因为浏览器仍然需要请求这些资源,但是比等待浏览器意识到网页需要这些资源要快。在这个例子中,请求附加响应和发送主页也有一些重叠,这使得这个流程有点复杂,但这个流程可能代表了现实生活。然而,与推送相比,此过程的优势在于减少了对浪费带宽的担忧。如果浏览器已经拥有资源,则不会请求它们。根据时间的不同,预加载的资源可能会在需要它们的页面之前被完全接收,这将使该过程与使用 HTTP/2 推送一样快。即使它不像图 5.19 那样快,预加载也可能被证明是一个更安全的中间地带,直到开发人员找到更好的方法来安全地使用 HTTP/2 推送而不浪费资源。

在撰写本文时,没有浏览器支持103 Early Hints处理 (尽管 Chrome28和 Firefox29正在处理它) ,并且并非所有浏览器都支持预加载链接标头。³⁰也许到103 Early Hints标头得到更好的支持时,过度推送问题也会解决已通过缓存摘要或类似技术解决;那么开发人员将有两个选择。

5.9 HTTP/2 推送的其他用例目前,HTTP/2 推送针对特

定用例:尽早推送关键资源以加速页面加载,而无需内联资源。然而,从一开始,一些人就一直在问这个用例是否可以扩展。³¹正在讨论的用例包括以下内容: 如果要求仅使用推送来响应具体要求放宽了吗?这些技术允许客户端和服务器之间进行双向通信 (例如在服务器上有新信息可用时更新网页) 。还是将 HTTP/2 与 WebSockets 或 SSE 结合使用就足够了?³²目前,正如本章开头所述,

HTTP/2 推送并不是一个好的替代品,但通过一些更改可以使通信真正成为双向通信 方式,这里有潜力 (尽管 HTTP 的开销可能意味着与 WebSockets 的原始格式相比,这种解决方案并不理想) 。在相关说明中,BBC 研发部门在一篇有趣的论文中研究了使用 HTTP/2 推送作为广播方法。³³

HTTP/2 推送能否用于在资源变化时更新浏览器缓存?目前,缓存和缓存破坏³⁴技术很复杂。但是如果 HTTP/2

²⁸ <https://crbug.com/671310>

²⁹ https://bugzilla.mozilla.org/show_bug.cgi?id=1407355

³⁰ <https://caniuse.com/#feat=link-rel-preload>

³¹ <http://www.igvita.com/2013/06/12/innovating-with-http-2-0-server-push/>

³² <http://www.infoq.com/articles/websocket-and-http2-coexist>

³³ <http://www.bbc.co.uk/rd/publications/whitepaper336>

³⁴ <https://css-tricks.com/strategies-for-cache-busting-css/>

`push` 允许您将资源直接推送到 HTTP 缓存中（目前还不能），将有新的机会来处理网站上的更改。

它可以用 来改进渐进式 JPEG 吗？ 渐进式 JPEG 开始显示模糊图像，随着文件下载次数的增加变得越来越清晰，这将受益于能够并行下载许多图像。 如果您可以在发送初始视图后更改优先级，那么使用 HTTP/2 推送会变得更有趣。³⁵ 这样，服务器可以发送具有高优先级的初始视图，然后退回以发送低优先级图像的其余部分优先。 Shimmer cat 是一个使用这种技术的 Web 服务器，将在第 7 章中讨论。 它可以用于 API 吗？ 至少有一位 API 开发人员建议，HTTP/2 推送可用于推送额外信息，但仍保持资源分离，这可能会导致许多有趣的用例，或许不受浏览器推送缓存的限制。 该协议允许使用推送来回复任何请求。 但是在很多方面，浏览器通过不允许使用推送的资源甚至通知页面来限制推送的使用，除非页面随后请求资源。 非基于浏览器的 HTTP/2 客户端可以消除此类限制。

添加通知会导致其他用例吗？ 向浏览器添加 HTTP/2 推送通知事件或 API 可能会导致其他有趣的用例。 例如，新闻或社交媒体网站可以用简短的“是否有任何更新？”来轮询其服务器。 要求。 如果发生更新（例如突发新闻），任何需要的资源都可以作为标准 HTTP 资源（HTML、基于 JSON 的数据、图像等）推送； 然后可以将一个事件发送到 Web 应用程序，以通知客户端在这些资源到达时获取并显示这些资源。 这种技术可以允许即时加载这些网页。 这种技术相对于 WebSockets 或 SSE 的优势是 HTTP 的所有优势（例如缓存、文件格式和简单性）。

总之，HTTP/2 推送的用例可能比改进首次绘制时间更好。 我认为 HTTP/2 推送在其原始用例中未得到充分利用 出于本章讨论的充分理由 但它具有很大的潜力，也许对于本章中提到的一些想法以及其他想法。³⁶ 互联网工程任务组（IETF）已经启动了一个跟踪 HTTP/2 服务器推送用例的信息 RFC。³⁷ 或者，我们是否在试图维持 HTTP 是一种请求和响应协议的旧概念时过多地限制了自己？ WebSockets 和 SSE 显示出对通过 HTTP 传输的双向协议的需求和胃口，也许我们

³⁵ <https://calendar.perfplanet.com/2016/even-faster-images-using-http2-and-progressive-jpegs/>

³⁶ <https://groups.google.com/a/chromium.org/forum/#msg/net-dev/yfkW4mkWIPU/5RckmfktJgAJ>； 也可在 <https://goo.gl/gTJrwC> <https://tools.ietf.org/html/draft-bishop-httpbis-push-cases>

³⁷

应该在协议中允许它。至少已经为此编写了一个提案³⁸，并且 HTTP/2 引入的二进制框架层适用于这些类型的实现。我会在第 10 章回到这个话题。

HTTP/2 推送是新事物，网站所有者应谨慎对待它。这是一个有趣的功能，实验将显示 HTTP/2 推送承诺的性能提升是否实现，或者 HTTP/2 推送是否使一切都过于复杂而获得的额外收益很少。

我希望本章表明，尽管 HTTP/2 推送具有巨大的潜力，但您可能不想在没有仔细考虑的情况下就仓促行事。事实上，在更好地定义更多最佳实践和缓存感知技术之前，您可能不想考虑它。

概括

HTTP/2 推送是 HTTP/2 中的一个新概念，它允许将多个响应发送回单个 HTTP 请求。HTTP/2 推送被提议作为内联关键资源的替代方案。许多服务器和 CDN 是通过使用 HTTP 链接头来实现 HTTP/2 推送的。新的 103 Early Hints 状态代码可用于更早地提供链接标头。HTTP/2 推送在浏览器中的实现方式可能并不明显。过度推送资源很容易，并且会对网站产生不利影响

表现。

HTTP/2 推送的性能优势可能不是很大，风险是高的。

使用预加载提示可能会更好，也许有 103 个早期提示，而不是比推。

HTTP/2 推送可能有其他用例，尽管有些需要更改在协议中。

³⁸ <https://tools.ietf.org/html/draft-benfield-http2-p2p>