中文

http2讲解

这是一篇详细讲解HTTP/2（RFC 7540）的文档，主要内容包括该协议的背景、思想、协议本身的内容、对一些现有实现的探讨与对协议未来的展望。

若想访问该文档官方主页，请至https://daniel.haxx.se/http2/

若想获取本书源码，请至https://github.com/bagder/http2-explained

贡献

我鼓励大家帮助我们改进文档。我们接受pull requests，但你也可以提issues或者将你的建议直接发送到daniel-http2@haxx.se

/ Daniel Stenberg

背景

这篇文档会从技术和协议层面来介绍http2。文档起源于2014年4月我在斯德哥尔摩做了一次相关的演讲，在那之后我对演讲内容的细节进行了一些解释和补充，从而写出了这篇文档。

正式版http2规格标准叫做RFC 7540，发布于2015年5月15日：https://www.rfc-editor.org/rfc/rfc7540.txt

如果你有在这篇文章中发现任何我的失误造成的错误或疏漏，请帮我指正。我会在后续版本中修改。

为了让阅读体验更流畅，在这篇文章中我会使用“http2”来指代这一新协议，但请记住该协议的正式名字是HTTP/2。

1.1 关于作者

我的名字叫做Daniel Stenberg，在Mozilla工作。在过去20年，我一直致力于开源事业，参与了多个网络方面的项目。可能我最广为人知的身份是curl和libcurl的首席开发者。同时，我也参与了IETF HTTPbis工作组多年，工作在HTTP 1.1和http2标准化的一线.

Email: daniel@haxx.se

Twitter: @bagder

Web: daniel.haxx.se

Blog: daniel.haxx.se/blog

1.2 帮助我！

如果你在该文档里面发现任何错误、疏漏，请发送给我一份相关段落更改后的版本，我会进行修正并且注明所有对文档有贡献的人！希望能将这份文档变得越来越好。

这篇文档可以在https://daniel.haxx.se/http2下载。

1.3 许可证

1.4 文档历史

该文档的第一版发布于2014年4月25日。下面是最近主要改动的更新历史。

Version 1.13

- Converted the master version of this document to Markdown syntax
- 13: Mention more resources, updated links and descriptions
- 12: Updated the QUIC description with reference to draft
- 8.5: Refreshed with current numbers
- 3.4: The average is now 40 TCP connections
- 6.4: Updated to reflect what the spec says

Version 1.12

- 1.1: HTTP/2 is now in an official RFC
- 6.5.1: Link to the HPACK RFC
- 9.1: Mention the Firefox 36+ config switch for http2
- 12.1: Added section about QUIC

Version 1.11

- Lots of language improvements mostly pointed out by friendly contributors
- 8.3.1: Mention nginx and Apache httpd specific acitivities

Version 1.10

- 1: The protocol has been “okayed”
- 4.1: Refreshed the wording since 2014 is last year
- Front: Added image and call it “http2 explained” there, fixed link
- 1.4: Added document history section
- Many spelling and grammar mistakes corrected
- 14: Added thanks to bug reporters
- 2.4: Better labels for the HTTP growth graph
- 6.3: Corrected the wagon order in the multiplexed train
- 6.5.1: HPACK draft-12

Version 1.9

- Updated to HTTP/2 draft-17 and HPACK draft-11
- Added section "10. http2 in Chromium" (== one page longer now)
- Lots of spell fixes
- At 30 implementations now
- 8.5: Added some current usage numbers
- 8.3: Mention internet explorer too
- 8.3.1 Added "missing implementations"
- 8.4.3: Mention that TLS also increases success rate

HTTP的现状

几乎所有互联网上的内容都采用了HTTP 1.1作为通信协议。人们在该协议上投入了大量精力，所以基于它的基础架构也得以日臻完善。而得益于此，在现有的HTTP协议之上构建新的方案会比从底层建立新的协议要容易得多。

2.1 HTTP 1.1过于庞大

HTTP刚诞生的时候只被当作是一个相对简单直观的协议，但时间证明了早期的设计并不尽人意。于1996年发布的、描述HTTP 1.0规范的RFC 1945只有60页，但仅仅3年之后、描述HTTP 1.1规范的RFC 2616就一下增长到了176页。而当我们在IETF小组对该规范进行更新时，它更是被拆分成了总页数更多的六个文档（这就是RFC 7230及其文件族的由来与诞生）。总而言之，HTTP 1.1包含了太多细节和可选的部分，这让它变得过于庞大。

2.2 过多的可选项

HTTP 1.1不仅包含了非常多的细枝末节，同时也为未来的扩展预留了很多选项。这种事无巨细的风格导致在现有的软件生态中，几乎没有任何实现真正实现了协议中提及的所有细节，甚至要弄清楚“所有细节”到底包括哪些细节都非常困难。而这也导致了很多最初不常用的功能在后来的实现中很少会被支持，而有些最初实现了的功能，却又很少被使用。

随着时间推移，这些当初看似被边缘化的功能逐渐被用上，客户端和服务器的互用性（interoperability）问题就被暴露了出来。HTTP管线化（HTTP pipelining）就是一个非常好的例子。

2.3 未能被充分利用的TCP

HTTP 1.1很难榨干TCP协议所能提供的所有性能。HTTP客户端和浏览器必须要另辟蹊径的去找到新的解决方案来降低页面载入时间。

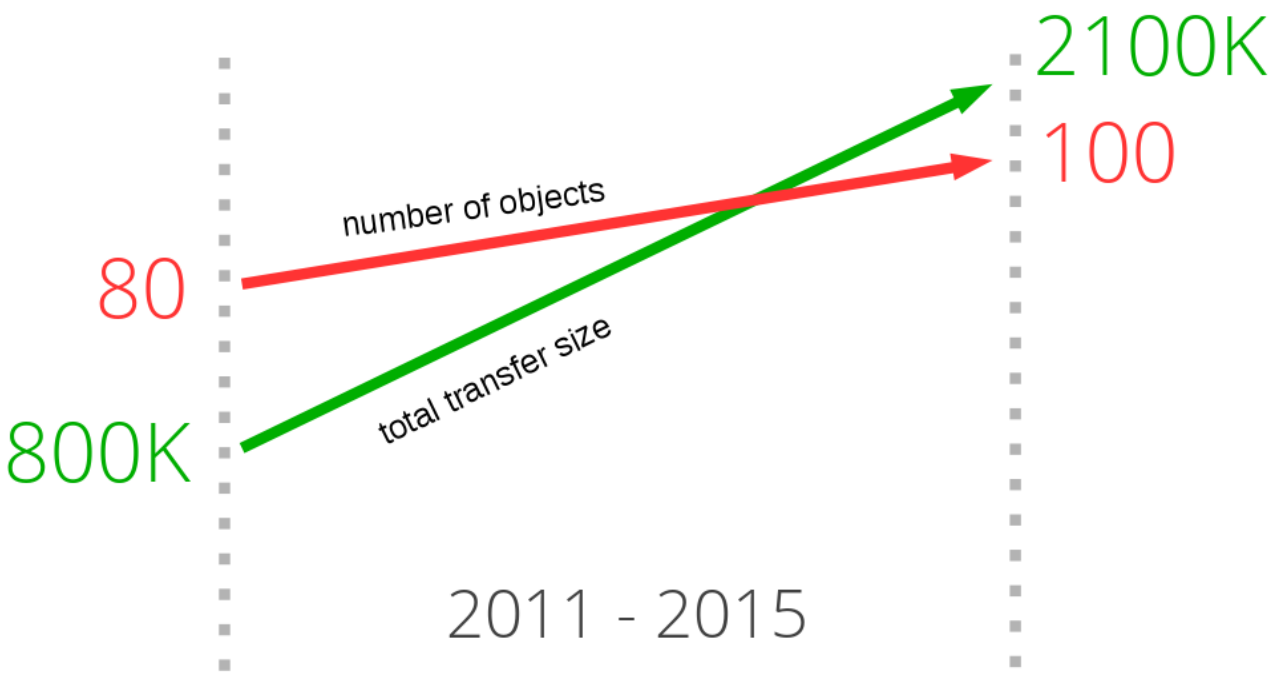
与此同时，人们也尝试去用新的协议来替代TCP，但结果证明这也非常困难。无奈之下，我们只能尝试同时改进TCP协议本身和基于TCP的上层协议。

简单来说，我们可以通过更好的利用TCP来减少传输过程中的暂停，并充分挖掘利用那些本可以用于发送/接受更多数据的时间。下面几段我们将会着重讨论这些问题。

2.4 传输大小和资源数量

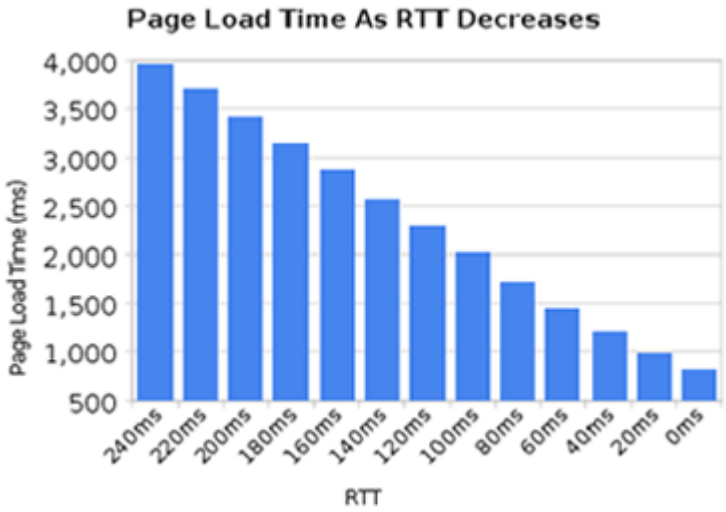
如果仔细观察打开那些最流行的网站首页所需要下载的资源的话，会发现一个非常明显的趋势。 近年来加载网站首页需要的下载的数据量在逐渐增加，并已经超过了1.9MB。但在这里我们更应该关心的是：平均每个页面为了完成显示与渲染所需要下载的资源数已经超过了100个。

正如下图所示，这种趋势已经持续了很长一段时间，并且没有减缓的迹象。该图表中绿色直线展示了传输数据大小的增长，红色直线展示了平均请求资源数量的增长。



transfer size growth

2.5 恼人的延迟



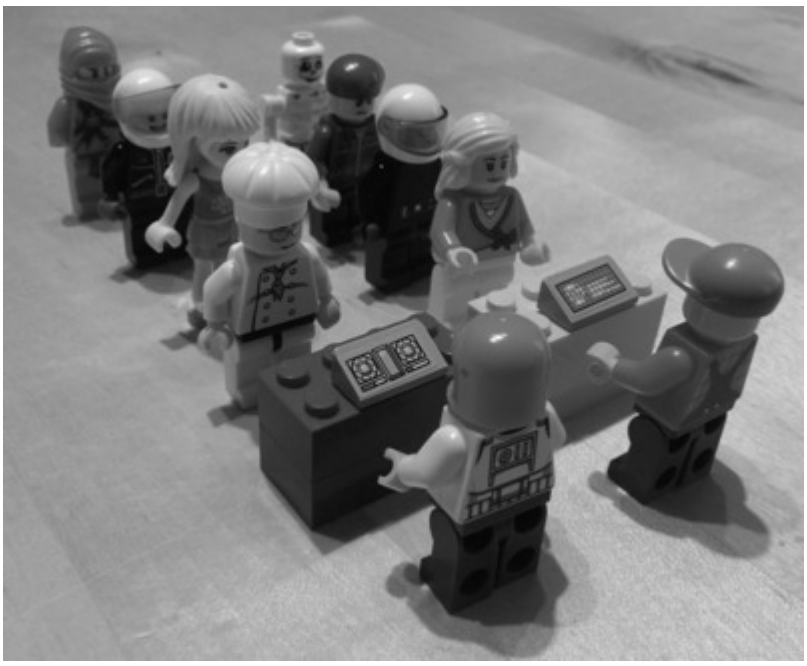
HTTP 1.1对网络延迟非常敏感。部分原因是HTTP pipelining还存有很多问题，所以对大部分用户来说这项技术是被默认关闭的。

虽然近几年来网络带宽增长非常快，然而我们却并没有看到网络延迟有对应程度的降低。在高延迟的网络上（比如移动设备），即使拥有高连接速率，也很难获得优质快速的网络体验。

另外一个需要低延迟的场景是某些视频服务，如视频会议、游戏和一些类似无法预生成待发送数据流的服务。

2.6 线头阻塞（Head-of-line blocking）

HTTP pipelining是这样一种技术：在等待上一个请求响应的同时，发送下一个请求。(译者注：作者这个解释并不完全正确，HTTP pipelining其实是把多个HTTP请求放到一个TCP连接中一一发送，而在发送过程中不需要等待服务器对前一个请求的响应；只不过，客户端还是要按照发送请求的顺序来接收响应。)但就像在超市收银台或者银行柜台排队时一样，你并不知道前面的顾客是干脆利索的还是会跟收银员/柜员磨蹭到世界末日（译者注：不管怎么说，服务器（即收银员/柜员）是要按照顺序处理请求的，如果前一个请求非常耗时（顾客磨蹭），那么后续请求都会受到影响），这就是所谓的线头阻塞（head-of-line blocking）。



当然，你可以在选择队伍时候就做好功课，去排一个你认为最快的队伍，或者甚至另起一个新的队伍（译者注：即新建一个TCP连接）。但不管怎么样，你总归得先选择一个队伍，而且一旦选定之后，就不能更换队伍。

但是，另起新队伍会导致资源耗费和性能损失（译者注：新建 TCP 连接的开销非常大）。这种另起新队伍的方式只在新队伍数量很少的情况下有作用，因此它并不具备可扩展性。（译者注：这段话意思是说，靠大量新建连接是不能有效解决延迟问题的，即HTTP pipelining并不能彻底解决head-of-line blocking问题。）所以针对此问题并没有完美的解决方案。

这就是为什么即使到了今天，大部分桌面浏览器仍然会选择默认关闭HTTP pipelining这一功能的原因。

而关于这个问题的更多细节，可以参阅Firefox的 bugzilla #264354。

那些年，克服延迟之道

再困难的问题也有解决的方案，但这些方案却良莠不齐。

3.1 Spriting

Spriting是一种将很多较小的图片合并成一张大图，再用JavaScript或者CSS将小图重新“切割”出来的技术。

网站可以利用这一技巧来达到提速的目的——在HTTP 1.1里，下载一张大图比下载100张小图快得多。

但是当某些页面只需要显示其中一两张小图时，这种缓存整张大图的方案就显得过于臃肿。同时，当缓存被清楚的时候的时候，Spriting会导致所有小图片被同时删除，而不能选择保留其中最常用的几个。

3.2 内联（Inlining）

Inlining是另外一种防止发送很多小图请求的技巧，它将图片的原始数据嵌入在CSS文件里面的URL里。而这种方案的优缺点跟Spriting很类似。

```
.icon1 {
    background: url(<data>) no-repeat;
  }
.icon2 {
    background: url(<data>) no-repeat;
  }
```

3.3 拼接（Concatenation）

大型网站往往会包含大量的JavaScript文件。开发人员可以利用一些前端工具将这些文件合并为一个大的文件，从而让浏览器能只花费一个请求就将其下载完，而不是发无数请求去分别下载那些琐碎的JavaScript文件。但凡事往往有利有弊，如果某页面只需要其中一小部分代码，它也必须下载完整的那份；而文件中一个小小的改动也会造成大量数据的被重新下载。

这种方案也给开发者造成了很大的不便。

3.4 分片（Sharding）

最后一个我要说的性能优化技术叫做“Sharding”。顾名思义，Sharding就是把你的服务分散在尽可能多的主机上。这种方案乍一听比较奇怪，但是实际上在这背后却蕴藏了它独辟蹊径的道理！

最初的HTTP 1.1规范提到一个客户端最多只能对同一主机建立两个TCP连接。因此，为了不和规范冲突，一些聪明的网站使用了新的主机名，这样的话，用户就能和网站建立更多的连接，从而降低载入时间。

后来，两个连接的限制被取消了，现在的客户端可以轻松地和每个主机建立6-8个连接。但由于连接的上限依然存在，所以网站还是会用这种技术来提升连接的数量。而随着资源个数的提升（上面章节的图例），网站会需要更多的连接来保证HTTP协议的效率，从而提升载入速度。在现今的网站上，使用50甚至100个连接来打开一个页面已经并不罕见。根据httparchive.org的最新记录显示，在Top 30万个URL中平均使用40（！）个TCP连接来显示页面，而且这个数字仍然在缓慢的增长中。

另外一个将图片或者其他资源分发到不同主机的理由是可以不使用cookies，毕竟现今cookies的大小已经非常可观了。无cookies的图片服务器往往意味着更小的HTTP请求以及更好的性能！

下面的图片展示了访问一个瑞典著名网站的时产生的数据包，请注意这些请求是如何被分发到不同主机的。

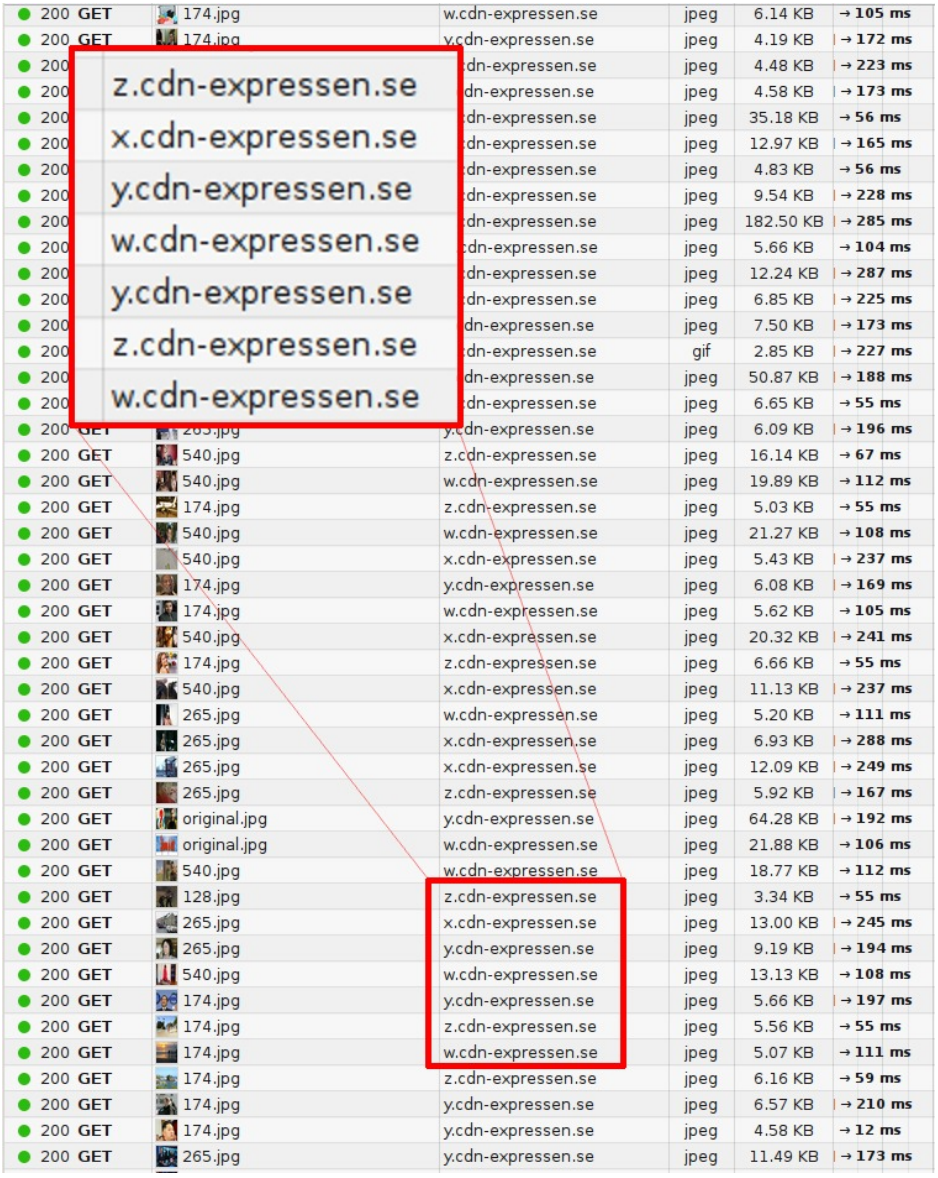


image sharding at expressen.se

升级HTTP

花点功夫去改善HTTP协议显然是极好的事情。我们可以着手于以下几个方面：

1. 降低协议对延迟的敏感
2. 修复pipelining和head of line blocking的问题
3. 防止主机需求更高的连接数量
4. 保留所有现有的接口，内容，URI格式和结构
5. 由IETF的HTTPbis工作组来制定

4.1. IETF和HTTPbis工作组

The Internet Engineering Task Force (IETF)是一个开发和推广互联网标准的组织。他们的重心是在协议层面。他们最出名的工作是制定了TCP、DNS、FTP和它们最佳实践的RFC规范，但HTTP和许多其他协议却进展缓慢。

IETF成立了独立的“工作小组”以便完成某些特定领域内的目标，他们建立了一个“章程”用以制定达到目标的指导方针和规范。在这里，任何人都可以参与讨论和开发，并且每个人有同等的话语权，没人关心你来自哪个公司或组织。

HTTPbis工作组（我们待会儿再解释这个名字）在2007年夏天成立之后就着手于HTTP1.1标准的更新。在组内，关于下一版本HTTP协议的讨论实际上在2012年后期才开始。而HTTP1.1的更新工作在2014年初完成，并被整理成RFC 7320系列。

2014年6月初，HTTPbis工作组名义上的最终版文档会议在纽约召开。而剩下的讨论以及等IETF走完流程通过官方的RFC版本预计在来年完成。

一些HTTP领域的权威缺席了工作组的讨论和会议。我并不想在此提及任何公司和产品。但藉此，现在互联网上也有一些参与者因此获得了更多信心——不需要这些公司参与IETF也能做得很好。。。

4.1.1. 名字中的“bis”

工作组名字中的“bis”来自拉丁语中表示“二”的副词，Bis通常被IETF用作名字的后缀来以表示标准的升级或者一些二次工作，比如这里是针对HTTP1.1。

4.2. 起源于SPDY的http2

SPDY是由Google牵头开发的协议。他们将其开源，使得每个人都可以参与开发。但很明显，他们通过控制浏览器的实现和享用着优质服务的大量用户来获益。

当HTTPbis小组决定开始制定http2的时候，SPDY已经充分证实了它是一个非常好用的方案。当时已经有人在互联网上成功部署SPDY，并且也有一些文章讨论他的性能。因此，http2便基于SPDY/3草案进行一些修改之后发布了http2的draft-00。

http2的观念

http2到底做了些什么呢？而HTTPbis小组究竟又应该把它制定到什么样的程度呢？

事实上，http2有着非常严格的边界，这也给小组成员的创新带来了些许限制。

- http2必须维持HTTP的范式。毕竟它只是一个让客户端发送请求到服务器的基于TCP的协议。
- 不能改变 http:// 和 https:// 这样的URL，也不能对其添加新的结构。使用这类URL的网站太多了，没法指望他们全部改变。
- HTTP1的服务器和客户端依然会存在很久，所以我们必须提供HTTP1到http2服务器的代理。
- 随后，我们也要让这种代理能够将http2的功能一对一的映射到HTTP 1.1的客户端。
- 删除或者减少协议里面那些可选的部分。虽然这并不算的上是一个需求，但是SPDY和Google的团队都非常喜欢这点。通过让协议里所有的内容都成为了强制性要求，可以防止人们在实现的时候偷懒，从而规避一些将来可能会发生的问题。
- 不再使用小版本号。服务器和客户端都必须确定自己是否完整兼容http2或者彻底不兼容。如果将来该协议需要被扩充或者变更，那么新的协议将会是http3，而不是http 2.x。

5.1. http2和现有的URI结构

如上所述，现有的URI结构正在被HTTP 1.x使用而不能被更换，所以http2也必须沿用该结构。因此不得不找到一种方式将使用的协议升级至http2，比如可以要求服务器让它作响应时使用http2来替代旧的协议。

HTTP 1.1本身就制定过“升级”的方案：提供一个首部字段，表示允许服务器在收到旧协议请求的同时，可以向客户端发送新协议的响应。但这一方案往往需要花费一次额外的往返通信来作为升级的代价。

而这一代价是SPDY团队不想接受的。因为他们只实现了基于TLS的SPDY，所以他们开发了一个TLS的扩展去简化协议的协商。这个扩展被称作NPN（Next Protocol Negotiation），借助于此，服务器会通知客户端所有它支持的协议，让客户端从中选择一个合适的来进行通讯。

5.2. 为 https:// 所准备的http2

有相当多的人关注到了http2可以在TLS上正常的运作，而SPDY依赖于TLS，所以按理说TLS也应成为http2 必需的组件，不过出乎大家意料的是http2将TLS标记成了可选。然而，全球两大浏览器领导者 —— Firefox和Chrome都明确地表示，他们只会实现基于TLS的http2.

选择TLS的原因的其中之一是希望保护以及尊重用户的隐私，而早期的评估结果也表明，在TLS上建立新的协议更有可能获得成功。而这其中部分原因是人们普遍认为任何来自80端口的流量都是基于HTTP 1.1亦或者是其某个变种的，而不是另外一种全新的协议。

关于是否应该强制使用TLS的主题在邮件组内和会议上引起了不小的争议 —— 这到底是好是坏呢？不管怎么样，对于这种备受争议的话题还是请谨慎讨论，尤其是当你面对一个HTTPbis小组成员的时候。

诸如此类，还有一个激烈而长期的讨论，即：如果选择了使用TLS，那http2是否应该强制规定密码列表，也许应该建立起一个黑名单，又或者它根本就不需要从TLS层得到任何东西。不过这个问题还是留给TLS工作组去解决吧，最后的规范中指定了TLS最低版本为1.2，并且会有加密组的限制。

5.3 基于TLS之上的http2协商

Next Protocol Negotiation (NPN)是一个用来在TLS服务器上协商SPDY的协议。IETF将这个非正式标准进行规范化，从而演变成了ALPN（Application Layer Protocol Negotiation）。ALPN会随着http2的应用被推广，而SPDY的客户端与服务器则会继续使用NPN。

由于NPN先于ALPN诞生，而ALPN又经历了一些标准化过程，所以许多早期的http2客户端和服务器在协商http2时会将这两者同时实现。与此同时，考虑到SPDY会使用NPN，而许多服务器又会同时提供SPDY以及http2，所以在这些服务器上同时支持ALPN以及NPN显然会成为最理所当然的选择。

ALPN和NPN的主要区别在于：谁来决定通信协议。在ALPN的描述中，是让客户端先发送一个协议优先级列表给服务器，由服务器最终选择一个合适的。而NPN则正好相反，客户端有着最终的决定权。

5.4 为 http:// 所准备的http2

正如我们之前所提到的，对于纯文本的HTTP1.1来说，协商http2的方法就是通过给服务器发送一个带**升级**头部的报文。如果服务器支持http2，它将以“101 Switching”作为回复的状态码，并从此开始在该连接上使用http2。也许你很容易就发现这样一个升级的流程会需要消耗掉一整个的往返时延，但好处是http2连接相比HTTP1可以被更大限度地重用和保持。

虽然有些浏览器厂商的发言人宣称他们不会实现这样的http2会话方式，但IE团队已公开表示他们会实现，与此同时，curl也已经支持了这种方式。

直到今天，没有任何主流浏览器支持非TLS的http2.

http2协议

背景介绍就到此为止了，历史的脚步已经将我们推到了今天。现在让我们深入看看该协议的规范，看看那些细节和概念。

6.1. 二进制

http2是一个二进制协议。

仔细想想，如果你是一个曾经跟互联网协议打过交道，那你很可能会本能反对二进制协议，你甚至准备好了一大堆理由来证明基于文本/ascii的协议是多么的有用，正如你曾无数次地通过telnet等应用手工地输入HTTP来发起请求。

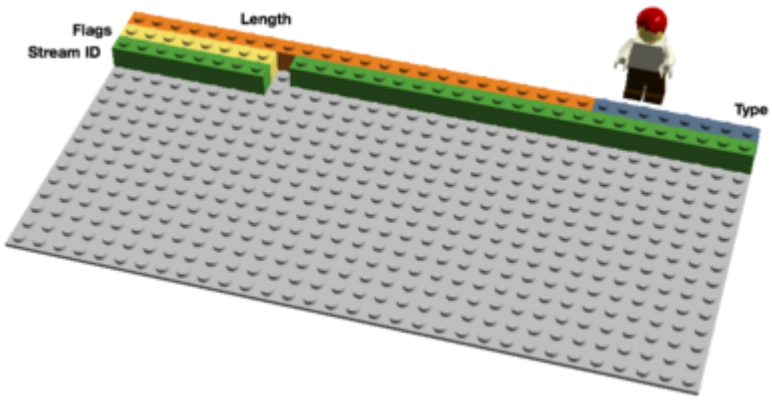
基于二进制的http2可以使成帧的使用变得更为便捷。在HTTP1.1和其他基于文本的协议中，对帧的起始和结束识别起来相当复杂。而通过移除掉可选的空白符以及其他冗余后，再来实现这些会变得更容易。

而另一方面，这项决议同样使得我们可以更加便捷的从帧结构中分离出那部分协议本身的内容。而在HTTP1中，各个部分相互交织，犹如一团乱麻。

事实上，由于协议提供了压缩这一特性，而其经常运行在TLS之上的事实又再次降低了基于纯文本实现的价值，反正也没办法直接从数据流上看到文本。因此通常情况下，我们必须习惯使用类似Wireshark这样的工具对http2的协议层一探究竟。

我们可以使用curl这样的工具来调试协议，而如果要进一步地分析网络数据流则需要诸如Wireshark这样的http2解析器。

6.2. 二进制格式



http2会发送有着不同类型的二进制帧，但他们都有如下的公共字段：Type, Length, Flags, Stream Identifier和frame payload

规范中一共定义了10种不同的帧，其中最基础的两种分别对应于HTTP 1.1的DATA和HEADERS。之后我会更详细的介绍它们其中的一部分。

6.3. 多路复用的流

上一节提到的Stream Identifier将http2连接上传输的每个帧都关联到一个“流”。流是一个独立的，双向的帧序列可以通过一个http2的连接在服务端与客户端之间不断的交换数据。

每个单独的http2连接都可以包含多个并发的流，这些流中交错的包含着来自两端的帧。流既可以被客户端/服务器端单方面的建立和使用，也可以被双方共享，或者被任意一边关闭。在流里面，每一帧发送的顺序非常关键。接收方会按照收到帧的顺序来进行处理。

流的多路复用意味着在同一连接中来自各个流的数据包会被混合在一起。就好像两个（或者更多）独立的“数据列车”被拼凑到了一辆列车上，但它们最终会在终点站被分开。下图就是两列“数据火车”的示例

它们就是这样通过多路复用的方式被组装到了同一列火车上。

multiplexed train

6.4. 优先级和依赖性

每个流都包含一个优先级（也就是“权重”），它被用来告诉对端哪个流更重要。当资源有限的时候，服务器会根据优先级来选择应该先发送哪些流。

借助于PRIORITY帧，客户端同样可以告知服务器当前的流依赖于其他哪个流。该功能让客户端能建立一个优先级“树”，所有“子流”会依赖于“父流”的传输完成情况。

优先级和依赖关系可以在传输过程中被动态的改变。这样当用户滚动一个全是图片的页面的时候，浏览器就能够指定哪个图片拥有更高的优先级。或者是在你切换标签页的时候，浏览器可以提升新切换到页面所包含流的优先级。

6.5. 头压缩

HTTP是一种无状态的协议。简而言之，这意味着每个请求必须要携带服务器需要的所有细节，而不是让服务器保存住之前请求的元数据。因为http2并没有改变这个范式，所以它也以同样原理工作。

这也保证了HTTP可重复性。当一个客户端从同一服务器请求了大量资源（例如页面的图片）的时候，所有这些请求看起来几乎都是一致的，而这些大量一致的东西则正好值得被压缩。

每个页面请求的资源数量在增多（如前所述），同时 cookies 的使用和请求的大小也在日渐增长。cookies需要被包含在所有请求中，且他们在多个请求中经常是一模一样的。

HTTP 1.1请求的大小正变得越来越大，有时甚至会大于TCP窗口的初始大小，这会严重拖累发送请求的速度。因为它们需要等待带着ACK的响应回来以后，才能继续被发送。这也是另一个需要压缩的理由。

6.5.1. 压缩是非常棘手的课题

HTTPS和SPDY的压缩机制被发现有受BREACH和CRIME攻击的隐患。通过向流中注入一些已知的文本来观察输出的变化，攻击者可以从加密的载荷中推导出原始发送的数据。

为协议的动态内容进行压缩并使其免于被攻击，需要仔细且全面的考虑，而这也正是HTTPbis小组尝试去做的。

HPACK，HTTP/2头部压缩，顾名思义它是一个专为http2头部设计的压缩格式。确切的讲，它甚至被制定写入在另外一个单独的草案里。新的格式同时引入了一些其他对策让破解压缩变得困难，例如采用帧的可选填充和用一个bit作为标记，来让中间人不压缩指定的头部。

用Roberto Peon（HPACK的设计者之一）的话说

> “HPACK旨在提供一个一致性的实现使信息量的损失尽可能少，使编解码快速而方便，使接收方能控制压缩文本的大小，允许代理重新建立索引（如，通过代理在前后端共享状态），以及对哈夫曼编码串的更快速比较”

6.6. 重置 - 后悔药

HTTP 1.1的有一个缺点是：当一个含有确切值的Content-Length的HTTP消息被送出之后，你就很难中断它了。当然，通常你可以断开整个TCP链接（但也不总是可以这样），但这样导致的代价就是需要通过三次握手来重新建立一个新的TCP连接。

一个更好的方案是只终止当前传输的消息并重新发送一个新的。在http2里面，我们可以通过发送RST_STREAM帧来实现这种需求，从而避免浪费带宽和中断已有的连接。

6.7. 服务器推送

这个功能通常被称作“缓存推送”。主要的思想是：当一个客户端请求资源X，而服务器知道它很可能也需要资源Z的情况下，服务器可以在客户端发送请求前，主动将资源Z推送给客户端。这个功能帮助客户端将Z放进缓存以备将来之需。

服务器推送需要客户端显式的允许服务器提供该功能。但即使如此，客户端依然能自主选择是否需要中断该推送的流。如果不需要的话，客户端可以通过发送一个RST_STREAM帧来中止。

6.8. 流量控制

每个http2流都拥有自己的公示的流量窗口，它可以限制另一端发送数据。如果你正好知道SSH的工作原理的话，这两者非常相似。

对于每个流来说，两端都必须告诉对方自己还有足够的空间来处理新的数据，而在该窗口被扩大前，另一端只被允许发送这么多数据。

而只有数据帧会受到流量控制。

扩展

http2协议强制规定了接收方必须读取并忽略掉所有未知帧（即未知帧类型的帧）。双方可以在逐跳原则（hop-by-hop basis）基础上协商使用新的帧，但这些帧的状态无法被改变，也不受流控制。

是否应该允许添加扩展的这个话题在制定http2协议的时候被反复讨论了很久，但在draft-12之后，最终尘埃落定确定了允许添加扩展。

但扩展不再是协议本身的一部分，它被记录在核心协议规范之外。现在已经有两种类型的帧被工作组记录在案，它们很可能率先被纳入协议的扩展部分，而这两个曾被当作“原生”的帧非常流行，所以接下来我会详细讨论它们。

7.1. 备选服务（Alternative Services）

随着http2逐渐被接受，我们有理由相信，相对于HTTP 1.x，TCP连接会更长并被保持的更久。对客户端来讲，最好是到每个主机/站点的每一条连接都可以做尽可能多的事情，而这也需要每个连接可以保持更长的时间。

但这会影响到HTTP负载均衡器的正常工作，比如在一个网站会出于性能的考虑，当然也可能是正常的维护或者一些类似的原因，想建议客户端连接到另外一个主机的时候。

服务器将会通过发送Alt-Svc头（或者http2的ALTSVC帧）来告知客户端另一个备选服务。即另外一条指向不同的服务源、主机或端口，但却能获取同样内容的路由。

客户端应该尝试异步的去连接到该服务，如果连接成功的话，即可以使用该备选服务。

7.1.1. 机会型TLS（Opportunistic TLS）

Alt-Svc头部意味着允许服务器基于 `http://` 提供内容，与此同时，这个头部也意味着告知客户端：同样的内容也可以通过TLS连接来获取。

这是个还在讨论中的功能。因为这样的连接会产生一个未认证的、在任何地方也不会被标示为“安全”的TLS连接，也不会在客户端界面上出现任何锁标识，所以没法让用户知道这其实

不是常规的HTTP连接。这就是很多人强烈反对机会型TLS的原因。

7.2. 阻塞（Blocked）

这个类型的帧意味着：当服务端存在需要发送的内容，但流控制却禁止发送任何数据时，那么此类型的帧将会被发送且**仅**发送一次。这种帧设计的目的在于，如果你接收到了此帧，那么连接中必然有错误发生或者是得到了低于期望的传输速度。

在此帧被放到协议扩展部分之前，draft-12中的一段话：

> ”阻塞帧被包含在草案版本中作为实验性的特性，如果它无法获得良好的反馈，那么该特性最后会被移除。”

http2的世界

那么当http2被广泛采用的时候，世界将会成什么样呢？或者说，它会被真正的采用吗？

8.1. http2会如何影响普通人？

到目前为止，http2还没被大范围部署使用，我们也无法确定到底会发生什么变化，但至少可以参考SPDY的例子和曾经做过的实验来进行大概的估计。

http2减少了网络往返传输的数量，并且用多路复用和快速丢弃不需要的流的办法来完全避免了head of line blocking(线头阻塞)的困扰。

它也支持大量并行流，所以即使网站的数据分发在各处也不是问题。

合理利用流的优先级，可以让客户端尽可能优先收到更重要的数据。

所有这些加起来，我认为页面载入时间和站点的响应速度都会更快。简而言之，它们都代表着更好的web体验。

但到底能变得多快，到底提升有多大呢？我认为目前很难说清楚。毕竟这些技术依然在早期阶段，我们还无法看见客户端和服务器实现这些并真正受益于它所提供的强大功能。

8.2. http2会如何影响web开发？

近年来，web开发者、web开发环境为HTTP 1.1存在的一些问题提供了一部分临时的解决方案。其中的一部分我已在上文中简单的介绍了，不妨简单的回忆一下。

很多工具和开发者可能会默认使用这些方案，但它们其中的一部分也许会损害到http2的性能，或者至少让我们无法真正利用到http2新提供的强大威力。Spriting和内联应该是http2里面最不需要的了。因为http2更倾向于使用更少的连接，所以Sharding甚至会伤害到http2的性能。

这里的问题在于：对于网站的开发者而言，在短期内开发和部署同一套前端来支持HTTP 1.1和http2的客户端访问并获得最大性能将会是一个挑战。

考虑到这些问题，我认为彻底发掘http2的潜力还有很长一段路要走。

8.3. http2的各种实现

在这样一篇文章中详细说明每个实现细节注定乏味且毫无意义，我将用更通用的术语来解释实际的场景，并在此给大家提供一个http2的实现列表作为参考。

在http2的早期就已经有大量的实现。并且在http2标准化工作期间，这个数量还持续增长。截至我写这篇文档的时候，共有40种实现已记录在案，他们中的大多数都实现了最新的草案。

8.3.1. 浏览器

Firefox一直紧跟最新的协议，Twitter也紧追不舍提供了基于http2的服务。2014年4月期间，Google在少数测试服务器上提供http2支持。从同年5月开始，开发版的Chrome支持http2。Microsoft也在他们的产品预发布会上展示了支持http2的下一代浏览器。Safari (iOS 9 以及 Mac OS X El Capitan) 和 Opera也都表态它们将会支持http2。

8.3.2 服务器

事实上，已经有不少的服务器实现了http2。

时下最流行的Nginx自1.9.5(发布于2015年9月22号)版本后提供了对http2的支持并且取缔了原来的SPDY模块(因此SPDY和http2无法同时运行在同一个Nginx服务器实例中)。

而Apache HTTPD服务器也实现了一个名为mod_http2的http2模块，并与2015年10月9号在2.4.17的版本中发布。

此外，H2O, Apache Traffic Server, nghttp2, Caddy 以及 LiteSpeed 也都发布了可以工作于http2下的服务器。

8.3.3 其他

curl和libcurl支持未加密的http2并借助某些TLS库支持了TLS版本。

Wireshark同样支持了http2, 所以用它来分析http2网络数据流着实是再好不过的了。

8.4. 对http2的常见批评

在制定协议的讨论过程中往往存在许多争议，甚至会有不少人认为这样的协议最终会以失败告终。这里我想提一些常见的对协议的批评以及我的解释：

8.4.1. “这个协议是Google设计制定的”

江湖上有太多传言暗示着这个世界越来越被Google所控制，但事实显然并非如此。这个协议是IETF制定的，就跟过去30年间很多其他协议一样。但不得不承认，SPDY是Google非常出色的成果。它不仅仅证明了开发一个新协议的可行性，还充分展现了新协议所能带来的好处。

而Google也公开声明了他们会在2016年移除Chrome里对SPDY和NPN的支持，并且极力推动服务器迁移至HTTP/2。2016年2月他们声明了SPDY和NPN会在Chrome 51被移除.

8.4.2. “这个协议只在浏览器上有用”

在一定意义上，这是对的。开发http2的其中一个主要原因就是修复HTTP pipelining。如果在你的应用场景里本来就不需要pipelining，那么确实很有可能http2对你没有太大帮助。虽然这并不是唯一的提升，但显然这是非常重要的一个。

一旦当某些服务意识到在一个连接上建立多路复用流的强大威力时，我认为会有越来越多的程序采用http2。

小规模的REST API和采用HTTP 1.x的简单程序可能并不会从迁移到http2中获得多大的收益。但至少，迁移至http2对绝大部分用户来讲几乎是没有坏处的。

8.4.3. “这个协议只对大型网站有用”

完全不是这样。因为缺乏内容分发网络，小网站的网络延迟往往较高，而多路复用的能力可以极大的改善在高网络延迟下的体验。大型网站往往已经将内容分发到各处，所以速度其实已经非常快了。

8.4.4. “TLS让速度变得更慢”

这个评价在某种程度上是对的。虽然TLS的握手确实增加了额外的开销，但也有越来越多的方案提出来减少TLS往返的时间。使用TLS而不是纯文本带来的开销是显著的，有可观证据表明，和传输同样的流量相比，TLS会消耗更多的CPU和其他资源。具体影响有多大以及怎么影响是一个和具体测量有关的课题。更多的例子可以参看istlsfastyet.com。

Telecom和一些其他网络服务商，例如ATIS开放网络联盟，表示为了为卫星、飞机等提供的快速网络体验，他们需要一些不加密的流量来提供caching，压缩和其他技术。

由于http2并不强制要求使用TLS，所以我们不应该为此担心。

如今，很多互联网使用者都希望TLS能被更广泛的使用来保护用户隐私。

实验也证明了通过使用TLS能比用在80端口实现一个新的基于文本的协议更容易成功。因为当前已经有太多中间商使用该方案，所以凡是基于80端口的协议，都很可能被理所当然的当作HTTP 1.1。

最后，得益于http2可以在单一连接上提供多路复用的流，正常使用普通浏览器也可以减少TLS握手的次数，所以使用HTTPS仍然会比HTTP 1.1更快。

8.4.5. “不基于ASCII是没法忍受的”

是的，如果我们可以直接读出协议内容，那么调试和追踪都会变得更为简单。但是基于文本的协议更容易产生错误，造成更多解析的问题。

假如你真的无法接受二进制协议，那么你也很难在HTTP 1.x中处理TLS和压缩。因为其实这些技术已经被使用了很久了。

8.4.6. “它根本没有比HTTP/1.1快”

当然，到底该如何定义和衡量“快”就是另外一个话题了，但在SPDY的时代，已经有很多实验证明了该协议会让浏览器载入页面变得更快（例如华盛顿大学的“SPDY有多快？”和Hervé Servy的“评估启用SPDY后的Web服务器的性能”），同样这些实验也可以被用来证明http2。我期待能有越来越多的诸如此类的测试实验结果发布。而这篇文章httpwatch.com的一个简单测试亦能证明HTTP/2名副其实。

8.4.7. “它违反了网络分层”

你确定这也是反对的理由么？网络分层并不是不可侵犯的。如果我们在制定http2的时候已经踏入了灰色地带，那我们当然可以尝试在限制内制定出更好更高效的协议。

8.4.8. “它并没有修复很多HTTP/1.1的短板”

确实是这样。兼容HTTP/1.1的范式是我们的目标之一，所以一些老的HTTP功能仍然被保留。例如一些常用的协议头、可怕的cookies、验证头等等。但保留这些范式的好处就是我们在升级到新协议的时候少掉很多工作，也不需要重写很多底层的东西。Http2其实只是一个新的帧层。

8.5. http2会被广泛部署吗？

现在讨论这个议题还言之尚早，但我仍然要在这里做出我的预估。

很多怀疑论者会以“看看IPv6现在的德性”为让我们回想起这个经历了10多年才开始慢慢被采用的协议。但http2毕竟不是IPv6。它是一个建立在TCP之上的使用基于原有HTTP协议升级过后的机制、端口号和TLS等的协议。大部分路由器或者防火墙不需要为此而进行更改。

Google向世界展示了他们的SPDY，证明了像这样的新协议也能在足够短的时间内拥有多种实现，并且能被浏览器和服务所采用。虽然如今支持SPDY服务器端数量在1%以内，但通过这些服务器所交换的数据却要大很多。很多非常流行的网站现在也有提供SPDY支持。

我认为建立在SPDY的基本范式之上的http2会被更广泛的部署，其中一个主要的原因是：它是一个IETF制定的协议。而SPDY则因为背负了“它是Google的协议”这个恶名，导致它的发展总是畏首畏脚。

在它首次发布的幕后有很多大型浏览器支持。来自Firefox，Chrome，Safari，Internet Explorer和Opera的代表宣布了他们会发布支持http2特性的浏览器，并且他们已经演示了一些能正常运作的实现。

也有很多像Google，Twitter和Facebook这样的服务器运营者希望尽快支持http2，也同样希望可以快点在主流服务器实现中出现对http2的支持（例如Apache HTTP Server和nginx）。而H2o作为一个极具潜力的新生HTTP服务器，已经支持了http2。

那些大型代理程序开发者，例如HAProxy、Squid和Varnish也表示出了他们对支持http2的兴趣。

纵观2015年，http2的流量正在逐步上升。9月初，Firefox 40中http2流量占据了所有HTTP流量中的13%，HTTPS中的27%。与此同时，Google表示约有18%的流量来自HTTP/2。值得注意的是，Google同时也在实验其他协议（参见12.1中的QUIC），这也使得http2的使用量暂时比正常值低一些。

Firefox里的http2

Firefox紧跟着草案，并且很早之前就实现了http2的测试实现。在http2协议开发的时候，客户端和服务器需要采用同一的协议草案版本，进行测试也变得比较繁琐。所以请一定注意你的客户端和服务器支持的是一样的版本。

9.1. 首先，确保它已被启用

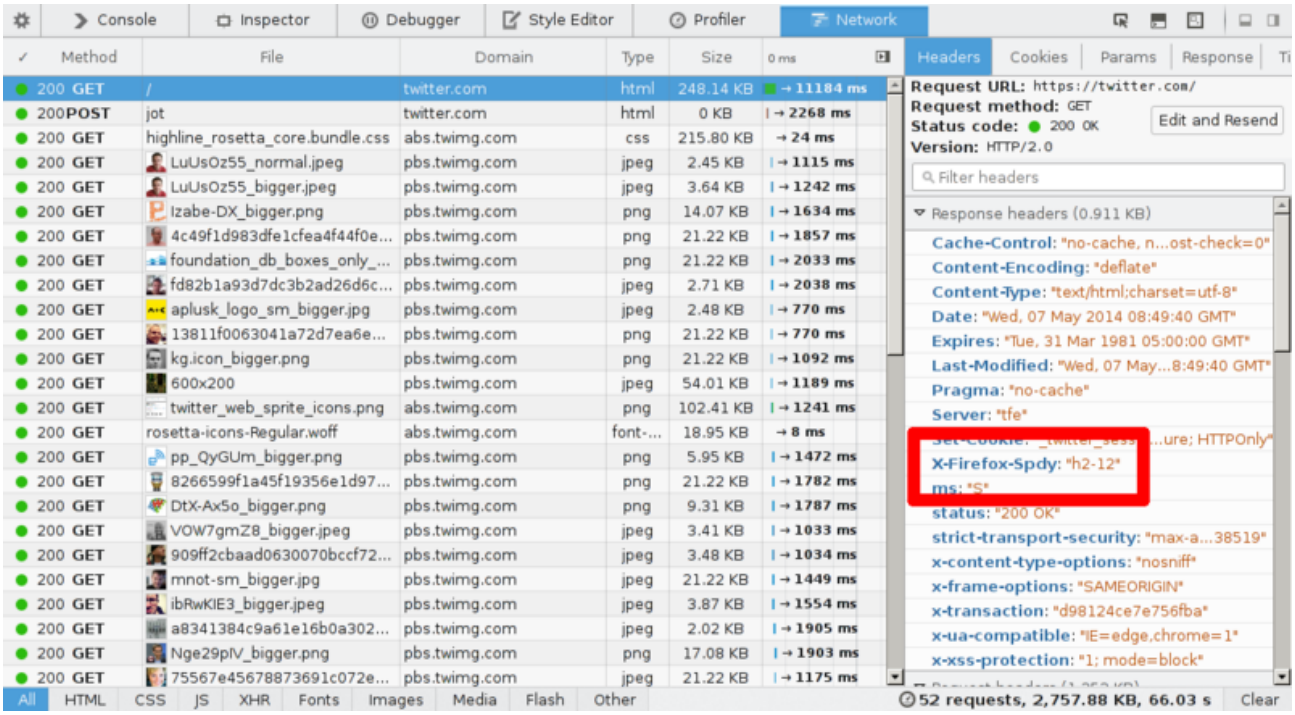
从发布于2015年1月13日的Firefox 35之后，http2支持是默认开启的。

在地址栏里进入'about:config'，再搜索一个名为“network.http.spdy.enabled.http2draft”的选项，确保它被设置为 `true` 。Firefox 36添加了一个“network.http.spdy.enabled.http2”的配置项，并默认设置为*true*。后者控制的是“纯”http2版本，而前者控制了启用／禁用通过http2草案版本进行协商。从Firefox 36之后，这两者都默认为true。

9.2. 仅限TLS

请记住Firefox只在TLS上实现了http2。你只会看到http2只在 `https://` 的网站里得到支持。

9.3. 透明！



transparent http2 use

在UI上，没有任何元素标明你正在使用http2。但想确认也并不复杂，一种方法是启用“Web developer->Network”，再查看响应头里面服务器发回来的内容。这个响应是“HTTP/2.0”，并且Firefox也插入了一个自己头“X-Firefox-Spdy:”，如上面截图所示。

你在这里看到的头文件是网络工具把二进制的http2格式转换成类似HTTP 1.x显示方式的文本格式。

9.4. 图形化HTTP/2

有一些Firefox的插件可以图形化HTTP/2，比如“HTTP/2 and SPDY Indicator”。

Chromium里的http2

Chromium团队并且很早之前就已经在dev和beta分支里面实现并支持了HTTP/2。从2015年1月27日发布的Chrome 40起，http2已经默认为一些用户启用该功能。虽然刚开始用户的数量会很少，但会慢慢增加。

Chrome 51移除了SPDY的支持来为http2铺路。在2016年2月的一篇博客里面有如下一段话：

> “在Chrome里有超过25%的资源是通过HTTP/2来传输的，而SPDY只有不到5%。考虑到如此大范围的采用，自5月15日，也就是HTTP/2 RFC的周年纪念日起，Chrome将不再支持SPDY。”

10.1. 首先，确保它已被启用

在地址栏里进入 `chrome://flags/#enable-spdy4` ，如果没有被enable的话，点击"enable"启用它。

10.2. TLS-only

请记住Chrome只在TLS上实现了http2。你只会在以 `https://` 做前缀的网站里得到http2的支持。

10.3. 图形化HTTP/2

有一些Chrome的插件可以图形化HTTP/2，比如“HTTP/2 and SPDY Indicator”。

10.4. QUIC

Chrome正在试验QUIC（详情请看12.1），所以或多或少稀释了HTTP/2的份额。

Curl里的http2

curl项目从2013年9月就开始对http2提供实验性的支持。

为了遵从curl的要旨，我们尽可能全方位地支持http2。curl通常被用作一个网站连接测试工具，希望这项使命也能在http2上被得以延续。

curl使用一个叫做nghttp2的库来提供http2帧层的支持。curl依赖于nghttp2 1.0以上版本。

请注意当前linux curl和libcurl并没有默认启用对HTTP/2协议的支持。

11.1. 跟HTTP 1.x非常相似

curl会在内部把收到的http2头部转换为HTTP1.x风格的头部再呈现给用户，这样一来，它们就和目前的HTTP非常类似。这也使得无论是用curl还是HTTP，转换都非常容易。类似地，curl会用相同的方式对发出的HTTP头部做转换，即发给curl的HTTP 1.x风格头部会在被发送到http2服务器之前完成转换。这使得户无需关心底层到底使用的是哪个版本的HTTP协议。

11.2. 不安全的纯文本

curl通过升级头部支持基于标准TCP的http2. 当发起一个使用http2的HTTP请求，如果可能，curl会请求服务器把连接升级到http2.

11.3. TLS和相关库

curl可以使用许多不同TLS的底层库来提供TLS支持，http2也得这样。TLS兼容http2的挑战来自于对ALPN以及一些NPN扩展的支持。

基于最新版本的OpenSSL或NSS编译curl可以同时获得ALPN和NPN支持。而使用GnuTLS或PolarSSL只能得到ALPN。

11.4. 命令行中使用

无论是用纯文本还是通过TLS，必须使用 `--http2` 参数来让curl使用http2。在未使用该参数的默认情况下，curl会使用HTTP/1.1。

11.5. libcurl参数

11.5.1 启用HTTP/2

应用程序和从前一样使用 `https://` 或者 `http://` 风格的URL，但你可以通过将 `curl_easy_setopt` 的 `SURLOPT_HTTP_VERSION` 参数设置为 `CURL_HTTP_VERSION_2` 来使libcurl尝试使用http2。它将优先尽可能地使用http2，如果不行的话，会继续使用HTTP 1.1。

11.5.2 多路复用

正如libcurl想尽可能量维持以前的用法，你需要通过CURLMOPT_PIPELINING参数为你的程序启用HTTP/2多路复用功能。不然的话，它会保持一个连接只发送一个请求。

另一个需要注意的小细节是，当你通过libcurl同时请求多个传输的时候，请使用多接口模式。这样能使应用程序能同时启用任意数量的传输。如果你宁愿让libcurl等待也要把它们放到同一个连接来传输的话，请使用CURLOPT_PIPEWAIT参数。

11.5.3 服务器推送

libcurl 7.44.0及其后续版本开始支持HTTP/2服务器推送功能。你可以通过在CURLMOPT_PUSHFUNCTION参数中设定一个推送回调来激活该功能。如果应用程序接受了该推送，它将为CURL建立一个新的传输，以便接受内容。

后http2时代

http2做了许多艰难的折衷和妥协。随着http2逐渐部署，将会带来一个健全的协议升级方式，而这为将来更多的协议升级奠定了基础。同时，它也引入了一套概念和基础架构来并行处理多个不同版本协议。也许我们并不需要在引入新协议时就完全将旧的淘汰掉。

http2仍然背负了许多HTTP1的历史包袱，主要是为了保证数据流量能够在HTTP 1和http2之间无碍转发。这些包袱会阻碍进一步的的开发和创造，期待http3能丢掉其中一部分。

亲爱的读者，你认为http还缺少什么？

12.1. QUIC

Google的QUIC （快速UDP互联网连接）协议是一个非常有趣的试验，它在很大程度上继承了SPDY的衣钵。QUIC是一个UDP版的TCP + TLS + HTTP/2替代实现。

QUIC可以创建更低延迟的连接，并且也像HTTP/2一样，通过仅仅阻塞部分流解决了包裹丢失这个问题，让连接在不同网络上建立变得更简单 – 这其实正是MPTCP想去解决的问题。

QUIC现在还只有Google的Chrome和它后台服务器上的实现，虽然有第三方库libquic，但这些代码仍然很难在其他地方被复用。该协议也被IETF通信工作组引入了草案。

扩展阅读

如果对读者你来说，这份文档的内容或技术细节稍显浅尝辄止，下面的资源也许能满足你的好奇：

- HTTPBis小组邮件列表和归档：https://lists.w3.org/Archives/Public/ietf-http-wg/
- HTML版本的http2协议规范本体：https://httpwg.github.io/specs/rfc7540.html
- Firefox http2网络细节：https://wiki.mozilla.org/Networking/http2
- curl http2实现细节：https://curl.haxx.se/docs/http2.html
- http2网站：https://http2.github.io/ ，以及关于http2的FAQ：https://http2.github.io/faq/
- Ilya Grigorik的“High Performance Browser Networking”一书中关于HTTP/2的章节: https://hpbn.co/http2/

致谢

介绍数据包格式的乐高图片来自于Mark Nottingham，感谢他给我提供的灵感。

HTTP趋势数据来自https://httparchive.org/。

RTT图来自Mike Belshe的演讲。

感谢我的孩子Agnes和Rex，他们把乐高玩具借给我作为head of line的图片。

感谢以下为我提供校对和反馈的朋友：Kjell Ericson, Bjorn Reese, Linux Swälas和Anthony Bryan。是你们让这份文档变得更好！

在完成文档的过程中，以下朋友反馈了bug帮助改进文档：Mikael Olsson, Remi Gacogne, Benjamin Kircher, saivlis, florin-andrei-tp, Brett Anthoine, Nick Parlante, Matthew King, Nicolas Peels, Jon Forrest, sbrickey, Marcin Olak, Gary Rowe, Ben Frain, Mats Linander, Raul Siles, Alex Lee, Richard Moore