

## 16 | HTTP/2是怎样提升性能的？

2020-06-10 陶辉

系统性能调优必知必会

[进入课程 >](#)



讲述：陶辉

时长 13:48 大小 12.65M



你好，我是陶辉。

上一讲我们从多个角度优化 HTTP/1 的性能，但获得的收益都较为有限，而直接将其升级到兼容 HTTP/1 的 HTTP/2 协议，性能会获得非常大的提升。

HTTP/2 协议既降低了传输时延也提升了并发性，已经被主流站点广泛使用。多数 HTTP 头部都可以被压缩 90% 以上的体积，这节约了带宽也提升了用户体验，像 Google 的高性能协议 gRPC 也是基于 HTTP/2 协议实现的。



目前常用的 Web 中间件都已支持 HTTP/2 协议，然而如果你不清楚它的原理，对于 Nginx、Tomcat 等中间件新增的流、推送、消息优先级等 HTTP/2 配置项，你就不知是否需要调整。

同时，许多新协议都会参考 HTTP/2 优秀的设计，如果你不清楚 HTTP/2 的性能究竟高在哪里，也就很难对当下其他应用层协议触类旁通。而且，HTTP/2 协议也并不是毫无缺点，到 2020 年 3 月时它的替代协议 [HTTP/3](#) 已经经历了 [27 个草案](#)，推出在即。HTTP/3 的目标是优化传输层协议，它会保留 HTTP/2 协议在应用层上的优秀设计。如果你不懂 HTTP/2，也就很难学会未来的 HTTP/3 协议。


所以，这一讲我们就将介绍 HTTP/2 对 HTTP/1.1 协议都做了哪些改进，从消息的编码、传输等角度说清楚性能提升点，这样，你就能理解支持 HTTP/2 的中间件为什么会提供那些参数，以及如何权衡 HTTP/2 带来的收益与付出的升级成本。

## 静态表编码能节约多少带宽？

HTTP/1.1 协议最为人诟病的是 ASCII 头部编码效率太低，浪费了大量带宽。HTTP/2 使用了静态表、动态表两种编码技术（合称为 HPACK），极大地降低了 HTTP 头部的体积，搞清楚编码流程，你自然就会清楚服务器提供的 `http2_max_requests` 等配置参数的意义。

我们以一个具体的例子来观察编码流程。每一个 HTTP/1.1 请求都会有 Host 头部，它指示了站点的域名，比如：

```
1 Host: test.taohui.tech\r\n
```

 复制代码

算上冒号空格以及结尾的 `\r\n`，它占用了 24 字节。**使用静态表及 Huffman 编码，可以将它压缩为 13 字节，也就是节约了 46% 的带宽！** 这是如何做到的呢？

我用 Chrome 访问站点 `test.taohui.tech`，并用 Wireshark 工具抓包（关于如何用 Wireshark 抓 HTTP/2 协议的报文，如果你还不太清楚，可参见 [《Web 协议详解与抓包实战》第 51 课](#)）后，下图高亮的头部就是第 1 个请求的 Host 头部，其中每 8 个蓝色的二进制位是 1 个字节，报文中用了 13 个字节表示 Host 头部。

9	0.077...	192.168.0.101	129.28.62.166	HTTP2	567	HEADERS[1]: GET /
10	0.115...	129.28.62.166	192.168.0.101	TCP	54 443 → 14058	[ACK] Seq=
11	0.115...	129.28.62.166	192.168.0.101	HTTP2	132	SETTINGS[0], WINDOW UP

[Header Count: 16]	
> Header: :method: GET	
> Header: :authority: test.taohui.tech	Host头部（静态表）
> Header: :scheme: https	
> Header: :path: /	test.taohui.tech
> Header: pragma: no-cache	(11个字节的Huffman编码)

0008	00000001	10000000	00000000	00000000	00000000	11111111	10000010	01000001
0010	10001011	01001001	01010000	10010101	11010010	00110011	11001111	01101001
0018	10010111	01001001	01001001	00111111	10000111	10000100	01000000	10000101

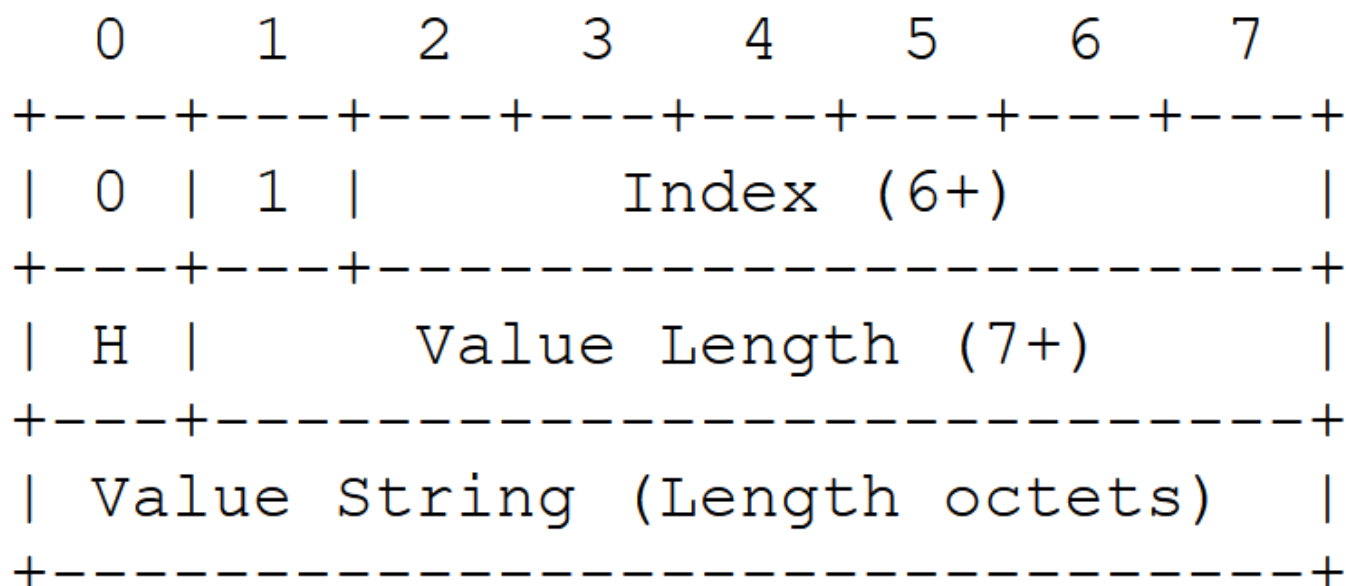
HTTP/2 能够用 13 个字节编码原先的 24 个字节，是依赖下面这 3 个技术。

首先基于二进制编码，就不需要冒号、空格和\r\n 作为分隔符，转而是用表示长度的 1 个字节来分隔即可。比如，上图中的 01000001 就表示 Host，而 10001011 及随后的 11 个字节表示域名。

其次，使用静态表来描述 Host 头部。什么是静态表呢？HTTP/2 将 61 个高频出现的头部，比如描述浏览器的 User-Agent、GET 或 POST 方法、返回的 200 SUCCESS 响应等，分别对应 1 个数字再构造出 1 张表，并写入 HTTP/2 客户端与服务器的代码中。由于它不会变化，所以也称为静态表。

Index	Header Name	Header Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
...	...	...
54	server	
55	set-cookie	
56	strict-transport-security	
57	transfer-encoding	
58	user-agent	
59	vary	
60	via	
61	www-authenticate	

这样收到 01000001 时，根据 [RFC7541](#) 规范，前 2 位为 01 时，表示这是不包含 Value 的静态表头部：



再根据索引 000001 查到 authority 头部（Host 头部在 HTTP/2 协议中被改名为 authority）。紧跟的字节表示域名，其中首个比特位表示域名是否经过 Huffman 编码，而后 7 位表示了域名的长度。在本例中，10001011 表示域名共有 11 个字节（ $8+2+1=11$ ），且使用了 Huffman 编码。

最后，使用静态 Huffman 编码，可以将 16 个字节的 test.taohui.tech 压缩为 11 个字节，这是怎么做到的呢？根据信息论，高频出现的信息用较短的编码表示后，可以压缩体积。因此，在统计互联网上传输的大量 HTTP 头部后，HTTP/2 依据统计频率将 ASCII 码重新编码为一张表，参见 [这里](#)。test.taohui.tech 域名用到了 10 个字符，我把这 10 个字符的编码列在下表中。

原字符	Huffman编码
t	01001
e	00101
s	01000
.	010111
a	00011
o	00111
h	100111
u	101101
i	00110
c	00100

这样，接收端在收到下面这串比特位（最后 3 位填 1 补位）后，通过查表（请注意每个字符的颜色与比特位是一一对应的）就可以快速解码为：

test.taohui.tech

01001001 01010000 10010101 11010010 00110011 11001111 01101001 10010111 01001001  
01001001 00111111

由于 8 位的 ASCII 码最小压缩为 5 位，所以静态 Huffman 的最大压缩比只有 5/8。关于 Huffman 编码是如何构造的，你可以参见 [每日一课《HTTP/2 能带来哪些性能提升？》](#)。

**动态表编码能节约多少带宽？**

虽然静态表已经将 24 字节的 Host 头部压缩到 13 字节，**但动态表可以将它压缩到仅 1 字节，这就能节省 96% 的带宽！**那动态表是怎么做到的呢？

你可能注意到，当下许多页面含有上百个对象，而 REST 架构的无状态特性，要求下载每个对象时都得携带完整的 HTTP 头部。如果 HTTP/2 能在一个连接上传输所有对象，那么只要客户端与服务器按照同样的规则，对首次出现的 HTTP 头部用一个数字标识，随后再传输它时只传递数字即可，这就可以实现几十倍的压缩率。所有被缓存的头部及其标识数字会构成一张表，它与已经传输过的请求有关，是动态变化的，因此被称为动态表。

静态表有 61 项，所以动态表的索引会从 62 起步。比如下图中的报文中，访问 test.taohui.tech 的第 1 个请求有 13 个头部需要加入动态表。其中，Host: test.taohui.tech 被分配到的动态表索引是 74（索引号是倒着分配的）。



8 0.077...	192.168.0.101	129.28.62.166	HTTP2	153 Magic, SETTINGS[0],
9 0.077...	192.168.0.101	129.28.62.166	HTTP2	567 HEADERS[1]: GET /
10 0.115...	129.28.62.166	192.168.0.101	TCP	54 443 -> 44050 [ACK] S
<				
▼ Stream: HEADERS, Stream ID: 1, Length 475, GET /				
Length: 475				
Type: HEADERS (1)				
> Flags: 0x25				
0... .. = Reserved: 0x0				
.000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1				
[Pad Length: 0]				
1... .. = Exclusive: True				
.000 0000 0000 0000 0000 0000 0000 0000 = Stream Dependency: 0				
Weight: 255				
[Weight real: 256]				
Header Block Fragment: 82418b495095d233cf699749493f87844085aec1cd48ff8e				
[Header Length: 813]				
[Header Count: 16]				
> Header: :method: GET				
表项74	> Header: :authority: test.taohui.tech			
	> Header: :scheme: https			
	> Header: :path: /			
表项73	> Header: pragma: no-cache			
表项72	> Header: cache-control: no-cache			
表项71	> Header: upgrade-insecure-requests: 1			
表项70	> Header: user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit			
表项69	> Header: sec-fetch-dest: document			
表项68	> Header: accept: text/html,application/xhtml+xml,application/xml;q=0.9,			
表项67	> Header: sec-fetch-site: none			
表项66	> Header: sec-fetch-mode: navigate			
表项65	> Header: sec-fetch-user: ?1			
表项64	> Header: accept-encoding: gzip, deflate, br			
表项63	> Header: accept-language: en,en-US;q=0.9,zh-CN;q=0.8,zh;q=0.7			
表项62	> Header: cookie: wp_xh_session_8dfd67ff6538645df475bd7404e3f7a7=3df82f85			

这样，后续请求使用到 Host 头部时，只需传输 1 个字节 11001010 即可。其中，首位 1 表示它在动态表中，而后 7 位 1001010 值为  $64+8+2=74$ ，指向服务器缓存的动态表第 74 项：



40	0.228...	192.168.0.101	129.28.62.166	HTTP2	170	HEADERS[3]: GET /dlib.js
41	0.228...	192.168.0.101	129.28.62.166	HTTP2	146	HEADERS[5]: GET /dlib.css
42	0.231...	192.168.0.101	129.28.62.166	HTTP2	166	HEADERS[7]: GET /dlib-logo.png
43	0.232...	192.168.0.101	129.28.62.166	HTTP2	125	HEADERS[9]: GET /plus.gif

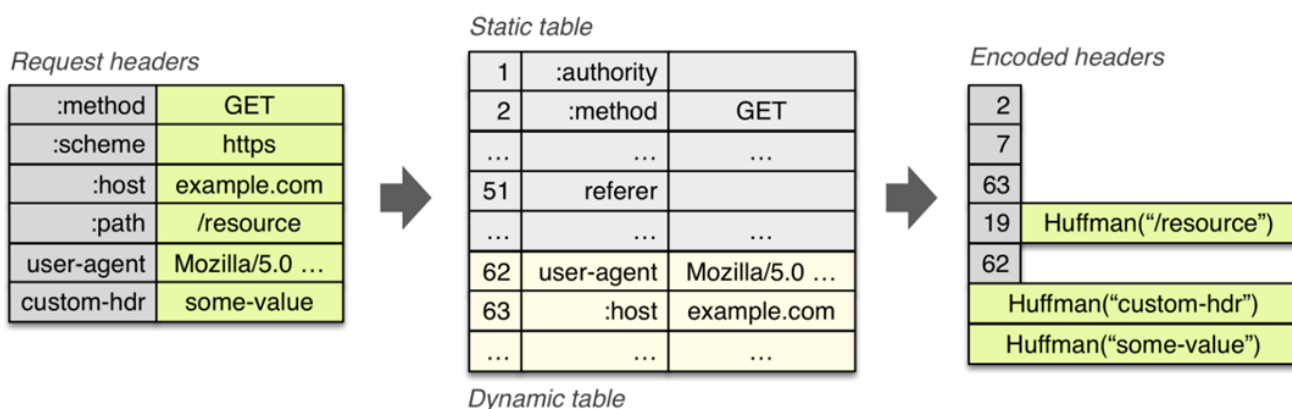
[Header Count: 15]

- > Header: :method: GET
- > Header: :authority: test.taohui.tech
- > Header: :scheme: https
- > Header: :path: /dlib.js

0000	00000000	00000000	01001110	00000001	00100101	00000000	00000000	00000000	...	N.%...
0008	00000011	10000000	00000000	00000000	00000000	11011011	10000010	11001010	...	.....
0010	10000111	00000000	10000100	10111001	01011000	11010011	00111111	10000110	...	...X...

静态表、Huffman 编码、动态表共同完成了 HTTP/2 头部的编码，其中，前两者可以将体积压缩近一半，而后者可以将反复传输的头部压缩 95% 以上的体积！



那么，是否要让一条连接传输尽量多的请求呢？并不是这样。动态表会占用很多内存，影响进程的并发能力，所以服务器都会提供类似 `http2_max_requests` 这样的配置，限制一个连接上能够传输的请求数量，通过关闭 HTTP/2 连接来释放内存。因此，**`http2_max_requests` 并不是越大越好，通常我们应当根据用户浏览页面时访问的对象数量来设定这个值。**

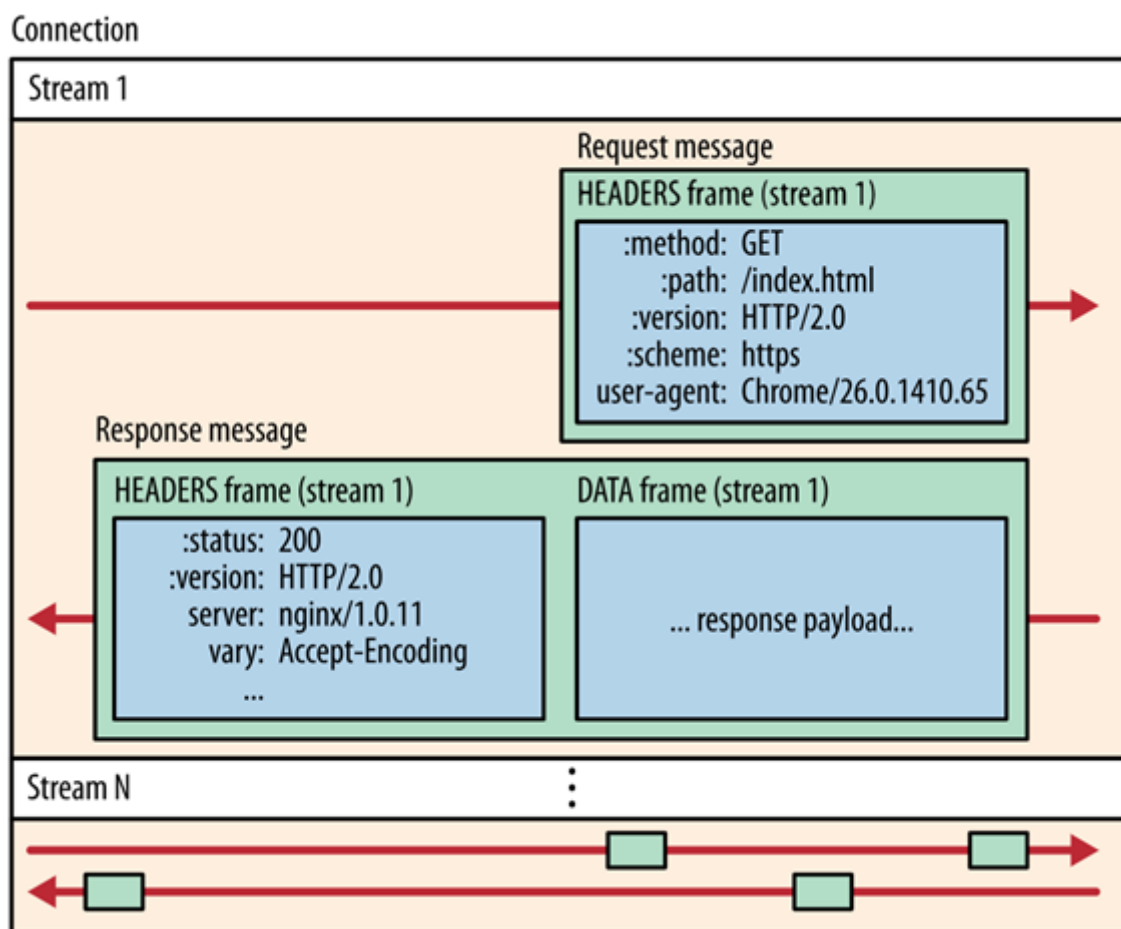
## 如何并发传输请求？

HTTP/1.1 中的 KeepAlive 长连接虽然可以传输很多请求，但它的吞吐量很低，因为在发出请求等待响应的那段时间里，这个长连接不能做任何事！而 HTTP/2 通过 Stream 这一设计，允许请求并发传输。因此，HTTP/1.1 时代 Chrome 通过 6 个连接访问页面的速度，远远比不上 HTTP/2 单连接的速度，具体测试结果你可以参考这个 [页面](#)。

为了理解 HTTP/2 的并发是怎样实现的，你需要了解 Stream、Message、Frame 这 3 个概念。HTTP 请求和响应都被称为 Message 消息，它由 HTTP 头部和包体构成，承载这二

者的叫做 Frame 帧，它是 HTTP/2 中的最小实体。Frame 的长度是受限的，比如 Nginx 中默认限制为 8K (http2\_chunk\_size 配置)，因此我们可以得出 2 个结论：HTTP 消息可以由多个 Frame 构成，以及 1 个 Frame 可以由多个 TCP 报文构成 (TCP MSS 通常小于 1.5K)。

再来看 Stream 流，它与 HTTP/1.1 中的 TCP 连接非常相似，当 Stream 作为短连接时，传输完一个请求和响应后就会关闭；当它作为长连接存在时，多个请求之间必须串行传输。在 HTTP/2 连接上，理论上可以同时运行无数个 Stream，这就是 HTTP/2 的多路复用能力，它通过 Stream 实现了请求的并发传输。

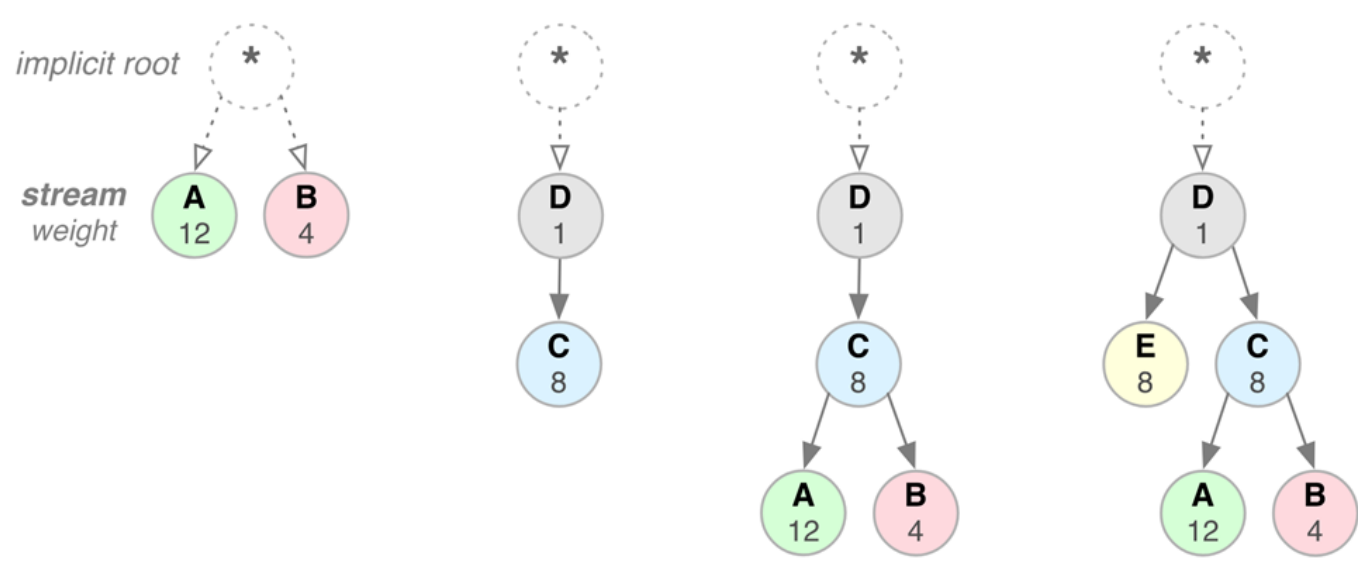


图片来源: <https://developers.google.com/web/fundamentals/performance/http2>

虽然 RFC 规范并没有限制并发 Stream 的数量，但服务器通常都会作出限制，比如 Nginx 就默认限制并发 Stream 为 128 个 (http2\_max\_concurrent\_streams 配置)，以防止并发 Stream 消耗过多的内存，影响了服务器处理其他连接的能力。

HTTP/2 的并发性能比 HTTP/1.1 通过 TCP 连接实现并发要高。这是因为，**当 HTTP/2 实现 100 个并发 Stream 时，只经历 1 次 TCP 握手、1 次 TCP 慢启动以及 1 次 TLS 握手，但 100 个 TCP 连接会把上述 3 个过程都放大 100 倍！**

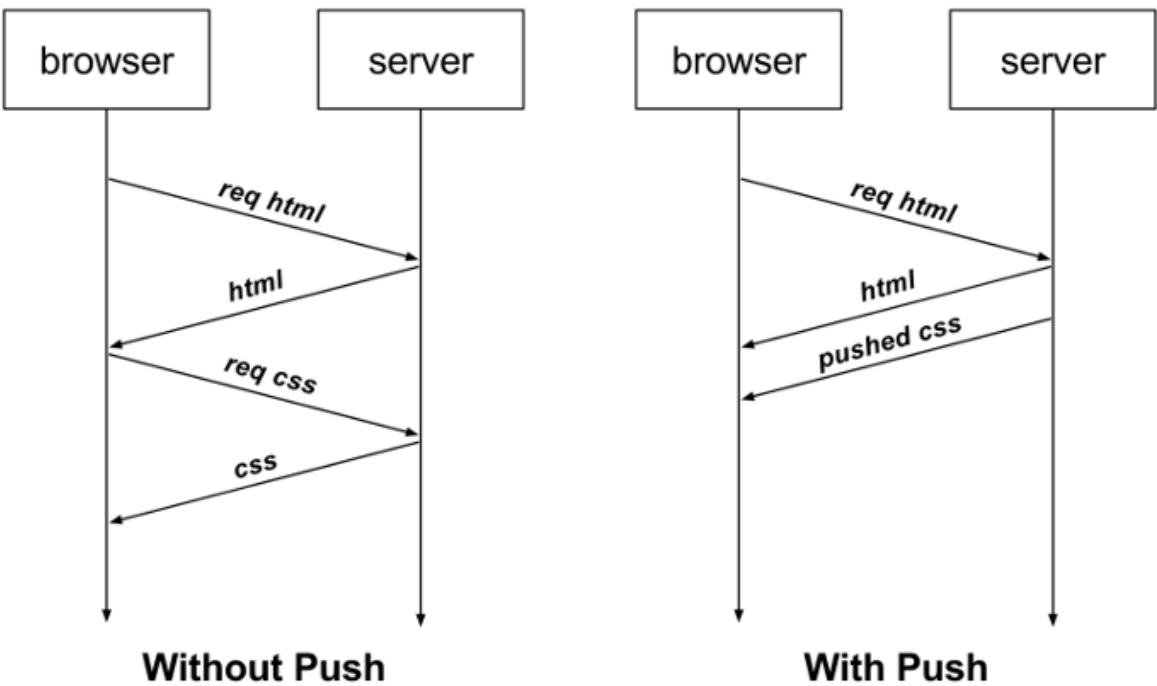
HTTP/2 还可以为每个 Stream 配置 1 到 256 的权重，权重越高服务器就会为 Stream 分配更多的内存、流量，这样按照资源渲染的优先级为并发 Stream 设置权重后，就可以让用户获得更好的体验。而且，Stream 间还可以有依赖关系，比如若资源 A、B 依赖资源 C，那么设置传输 A、B 的 Stream 依赖传输 C 的 Stream 即可，如下图所示：



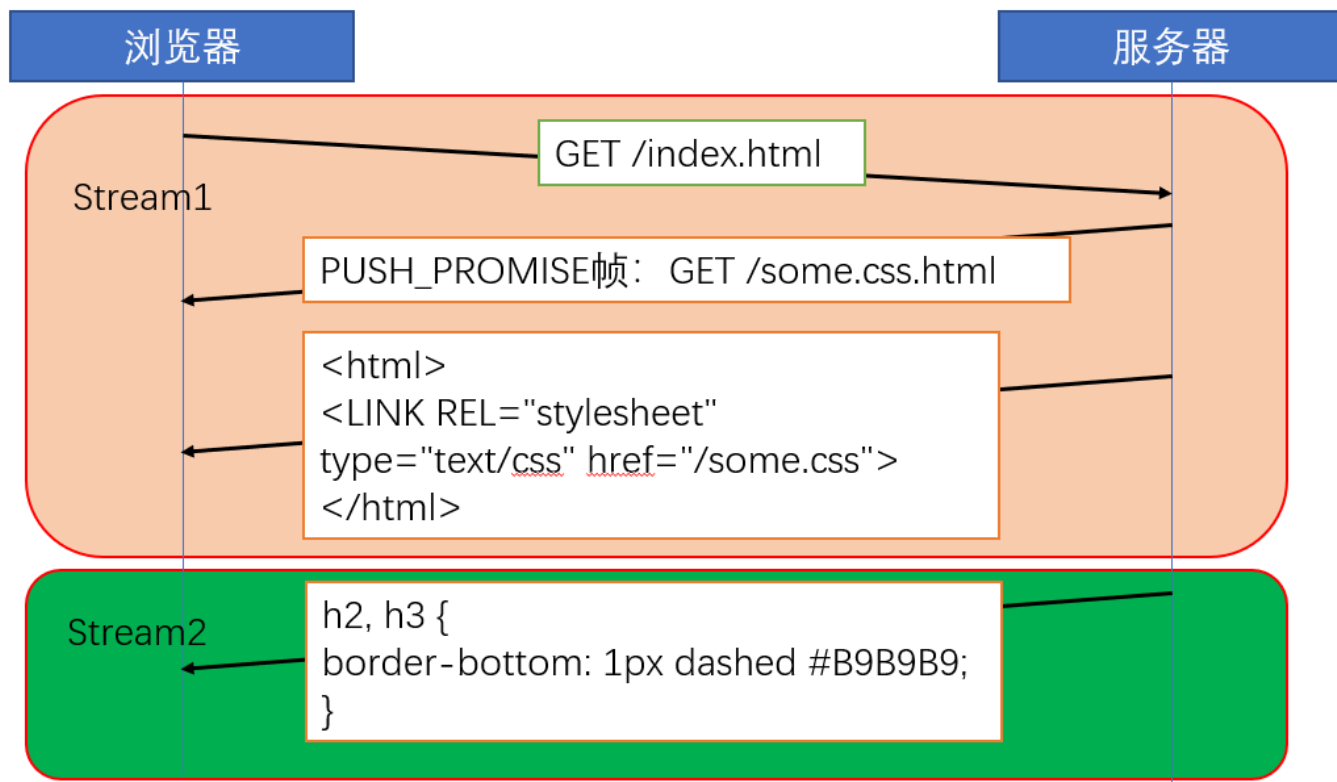
图片来源：<https://developers.google.com/web/fundamentals/performance/http2>

服务器如何主动推送资源？

HTTP/1.1 不支持服务器主动推送消息，因此当客户端需要获取通知时，只能通过定时器不断地拉取消息。HTTP/2 的消息推送结束了无效率的定时拉取，节约了大量带宽和服务器资源。




HTTP/2 的推送是这么实现的。首先，所有客户端发起的请求，必须使用单号 Stream 承载；其次，所有服务器进行的推送，必须使用双号 Stream 承载；最后，服务器推送消息时，会通过 PUSH\_PROMISE 帧传输 HTTP 头部，并通过 Promised Stream ID 告知客户端，接下来会在哪个双号 Stream 中发送包体。



在 SDK 中调用相应的 API 即可推送消息，而在 Web 资源服务器中可以通过配置文件做简单的资源推送。比如在 Nginx 中，如果你希望客户端访问 /a.js 时，服务器直接推送 /b.js，那么可以这么配置：

```
1 location /a.js {
2     http2_push /b.js;
3 }
```

 复制代码

服务器同样也会控制并发推送的 Stream 数量（如 `http2_max_concurrent_pushes` 配置），以减少动态表对内存的占用。

## 小结

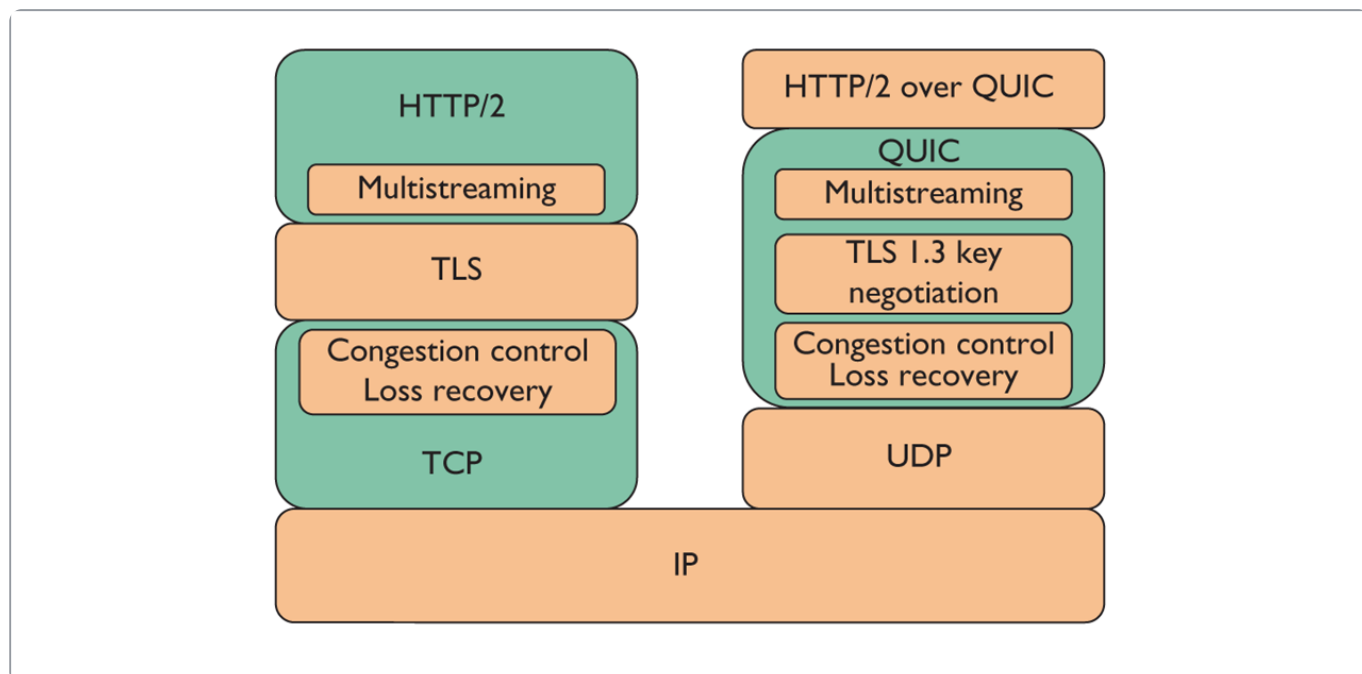
这一讲我们介绍了 HTTP/2 的高性能是如何实现的。

静态表和 Huffman 编码可以将 HTTP 头部压缩近一半的体积，但这只是连接上第 1 个请求的压缩比。后续请求头部通过动态表可以压缩 90% 以上，这大大提升了编码效率。当然，动态表也会导致内存占用过大，影响服务器的总体并发能力，因此服务器会限制 HTTP/2 连接的使用时长。

HTTP/2 的另一个优势是实现了 Stream 并发，这节约了 TCP 和 TLS 协议的握手时间，并减少了 TCP 的慢启动阶段对流量的影响。同时，Stream 之间可以用 Weight 权重调节优先级，还可以直接设置 Stream 间的依赖关系，这样接收端就可以获得更优秀的体验。

HTTP/2 支持消息推送，从 HTTP/1.1 的拉模式到推模式，信息传输效率有了巨大的提升。HTTP/2 推消息时，会使用 PUSH\_PROMISE 帧传输头部，并用双号的 Stream 来传递包体，了解这一点对定位复杂的网络问题很有帮助。

HTTP/2 的最大问题来自于它下层的 TCP 协议。由于 TCP 是字符流协议，在前 1 字符未到达时，后接收到的字符只能存放在内核的缓冲区里，即使它们是并发的 Stream，应用层的 HTTP/2 协议也无法收到失序的报文，这就叫做队头阻塞问题。解决方案是放弃 TCP 协议，转而使用 UDP 协议作为传输层协议，这就是 HTTP/3 协议的由来。



## 思考题

最后，留给你一道思考题。为什么 HTTP/2 要用静态 Huffman 查表法对字符串编码，基于连接上的历史数据统计信息做动态 Huffman 编码不是更有效率吗？欢迎你在留言区与我一起探讨。

感谢阅读，如果你觉得这节课对你有一些启发，也欢迎把它分享给你的朋友。

## 更多课程推荐

# MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇  
前阿里资深技术专家



涨价倒计时 🕒

今日秒杀 **¥79**，6月13日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 如何提升HTTP/1.1性能？

下一篇 17 | Protobuf是如何进一步提高编码效率的？

## 精选留言 (9)

写留言



安排

2020-06-10

静态表保存了最常用的一些头部，这些不变的头部可以全局保存一份，节约内存，不用每个连接重新构建，也节省构建表的时间。

展开 ▾



2



test

2020-06-12

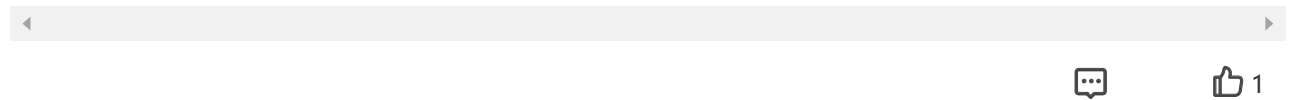
protobuf是对请求body进行了压缩，http2是对请求的header进行压缩。http2还可以使



用stream方式传输，这些都是protobuf没有解决的。

展开 ∨

作者回复: 是的，h2是应用层协议，而pb只是纯粹的消息编码工具



**罐头瓶子**

2020-06-10

静态 Huffman 编码可以在第一次传输时就降低头部大小，这部分编码是所有链接都可公用的，服务端客户端可降低动态编码的内存消耗。

展开 ∨



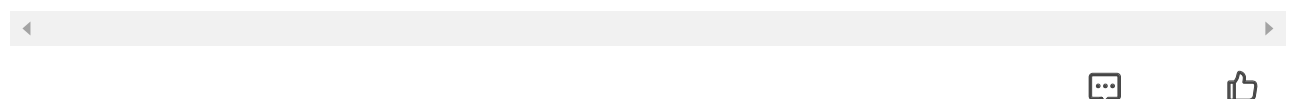
**鹤鸣**

2020-06-14

静态表和动态表本身也是占了一定的空间的，在发送报文时，静态表本身不需要随着报文被发送，因为双方已经达成了共识。但是对端不知道动态表是如何编码的，所以动态表则需要随着报文一起发送。也就是说，动态表本身也占了一部分发送的数据量，增大了待发送报文的长度。

展开 ∨

作者回复: 你好鹤鸣，前面理解都对，但动态表不需要发送，只要双方对于首次出现的HTTP头部，用同样的规则去构建动态表即可，你可以结合wireshark抓包重读下我在文中的例子



**有铭**

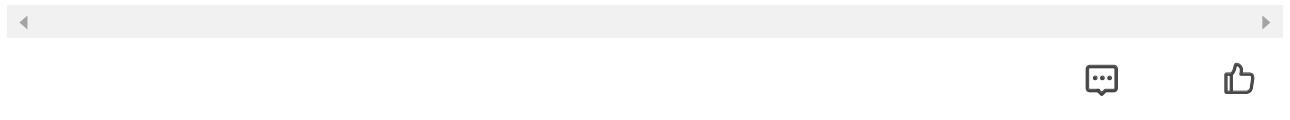
2020-06-11

我查了很多资料，都说h2的推送是针对资源的，单次请求，主动传输多个关联资源。目前没发现有暴露api给应用层使用者主动推送消息的案例。所以似乎h2的推送和我们传统意义上的推送不是一个概念，至少它和websocket这种真正的应用层可控双向传输不是一回事。我也希望老师能有案例告诉我h2真能达到websocket那种效果

展开 ∨

作者回复: 你好有铭，h2确实是针对资源的推送，websocket只不过没有定义应用层协议，它只是允许双向传输，至少传输的消息如何编码，它是不管的。而h2要求，推送时仍然要使用http作为推送协议，这是差别。

至少你说的案例，Java最基本的Servlet编程，其中HttpServletRequest有一个方法叫newPushBuilder，就可以推送消息。



 **Geek\_007**

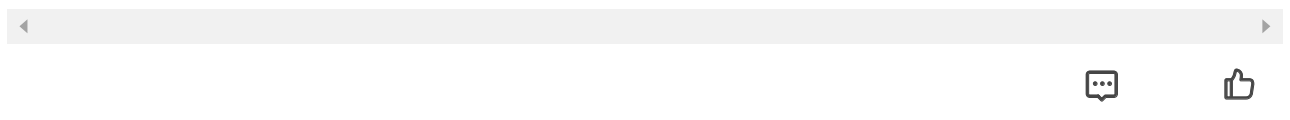
2020-06-10

虽然h2做了很多性能上的提升，但还是得结合实际来看，比如上传大文件使用h2的话，性能会不及h1。因为上传的文件大小是一样的，但是h2的流控会限制住每条connection,h2的默认控制窗口在65535字节，尽管tcp没有拥塞窗口的控制，但也要受限于h2自己的流控。也就是一路发送65535字节后就得等服务端ack。所以链接复用只对小请求有效。除非客户端和服务端避免特地避免了上述问题。陶老师，觉得呢？

展开 ▾

作者回复: 你好Geek\_007，我认为h2针对的应用场景是并发传输场景，如果只有1个传输的请求，那么讨论h2是没有意义的，它在tcp之上加了那么多约束，肯定没有直接跑在h1上快，特别是你提到的大文件，那么http header的压缩意义也不大了。因此，当客户端并发请求数量为1，且文件很大时，h2不会提升性能，甚至会降低性能。

然而，一旦有上百个HTTP请求并发传输时，h2的意义仍然非常大，即使有1个大文件在传输，h2连接仍然允许小文件的并发传输（比如你提到的stream流控）



**那一刻**

2020-06-10

请教一个在HTTP/2里Stream权重的问题，A（权重12）、B（权重8）的Stream依赖传输C（权重3）Stream, 此时这个Stream的权重还是 3么？如果权重是3的话，那之前A，B所在Stream被丢弃了么？



**myrfy**

2020-06-10

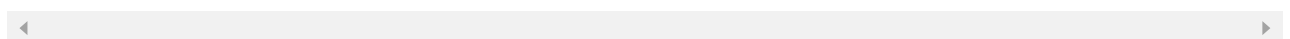
请问老师，h2的推送和websocket有什么区别呢？

展开 ▾

作者回复: 你好myrfy，主要是消息格式的不同。

websocket的推送是纯粹的二进制流。

h2的推送是HTTP消息，必须含有HTTP头部、包体。



**东郭**

2020-06-10

请问老师，不知道我的这些理解是不是对的：

- 1、多个stream并发是共用的同一个tcp连接，所以只需要1 次 TCP 握手、1 次 TCP 慢启动以及 1 次 TLS 握手；
- 2、因为多个stream共用了同一个tcp连接，tcp报文是有序的，所以也会有队头阻塞问题； ...

展开 ▾

作者回复: 你好东郭，你的理解前3点都是对的，第4点需要做一下补充：

首次出现的HTTP头部（不是首个请求）使用静态表和Huffman，后续请求头部使用动态表编码有一个前提，就是动态表没有超出限制。

关于host头部，可以这么理解：HTTP头部包括name和value，其中host头部仅有name在静态表中，所以首次出现时使用了静态表，但动态表可以存放name/value，所以第2次出现时就使用了动态表。

