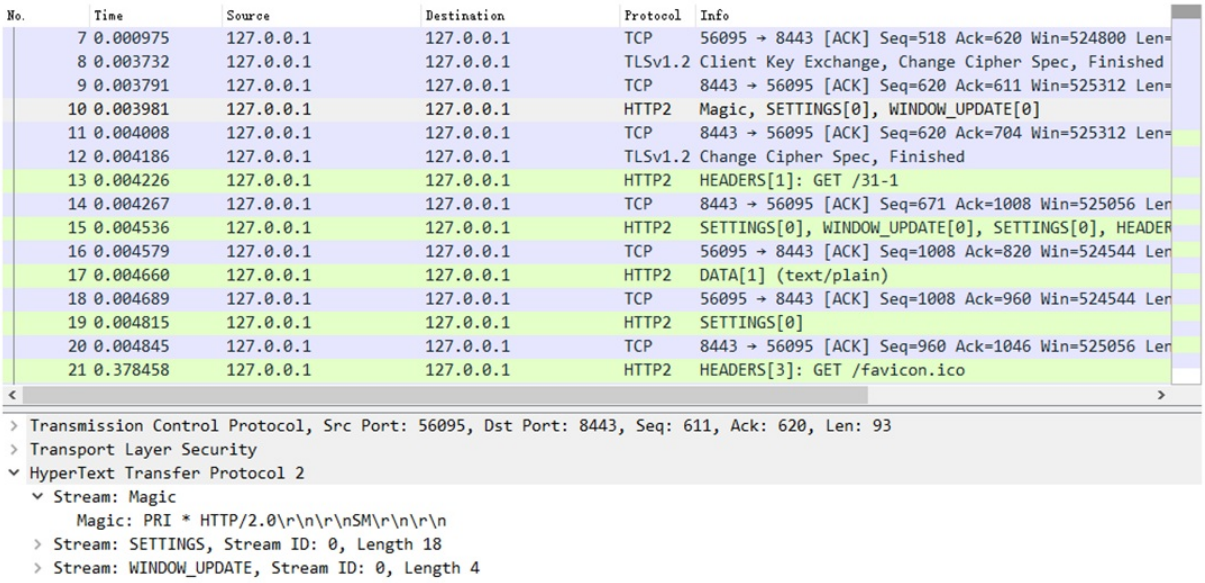
31-时代之风（下）：HTTP2内核剖析

今天我们继续上一讲的话题，深入HTTP/2协议的内部，看看它的实现细节。



这次实验环境的URI是“/31-1”，我用Wireshark把请求响应的过程抓包存了下来，文件放在GitHub的“wireshark”目录。今天我们就对照着抓包来实地讲解HTTP/2的头部压缩、二进制帧等特性。

连接前言

由于HTTP/2“事实上”是基于TLS，所以在正式收发数据之前，会有TCP握手和TLS握手，这两个步骤相信你一定已经很熟悉了，所以这里就略过去不再细说。

TLS握手成功之后，客户端必须要发送一个“**连接前言**”（connection preface），用来确认建立HTTP/2连接。

这个“连接前言”是标准的HTTP/1请求报文，使用纯文本的ASCII码格式，请求方法是特别注册的一个关键字“PRI”，全文只有24个字节：

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

在Wireshark里，HTTP/2的“连接前言”被称为“**Magic**”，意思就是“不可知的魔法”。

所以，就不要问“为什么会是这样”了，只要服务器收到这个“有魔力的字符串”，就知道客户端在TLS上想要的是HTTP/2协议，而不是其他别的协议，后面就会都使用HTTP/2的数据格式。

头部压缩

确立了连接之后，HTTP/2就开始准备请求报文。

因为语义上它与HTTP/1兼容，所以报文还是由“Header+Body”构成的，但在请求发送前，必须要

用“**HPACK**”算法来压缩头部数据。

“HPACK”算法是专门为压缩HTTP头部定制的算法，与gzip、zlib等压缩算法不同，它是一个“有状态”的算法，需要客户端和服务器各自维护一份“索引表”，也可以说是“字典”（这有点类似brotli），压缩和解压缩就是查表和更新表的操作。

为了方便管理和压缩，HTTP/2废除了原有的起始行概念，把起始行里面的请求方法、URI、状态码等统一转换成了头字段的形式，并且给这些“不是头字段的头字段”起了个特别的名字——“**伪头字段**”（pseudo-header fields）。而起始行里的版本号和错误原因短语因为没什么大用，顺便也给废除了。

为了与“真头字段”区分开来，这些“伪头字段”会在名字前加一个“:”，比如“:authority”“:method”“:status”，分别表示的是域名、请求方法和状态码。

现在HTTP报文头就简单了，全都是“Key-Value”形式的字段，于是HTTP/2就为一些最常用的头字段定义了一个只读的“**静态表**”（Static Table）。

下面的这个表格列出了“静态表”的一部分，这样只要查表就可以知道字段名和对应的值，比如数字“2”代表“GET”，数字“8”代表状态码200。

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
...	....	...
59	vary	
60	via	
61	www-authenticate	

但如果表里只有Key没有Value，或者是自定义字段根本找不到该怎么办呢？

这就要用到“**动态表**”（Dynamic Table），它添加在静态表后面，结构相同，但会在编码解码的时候随时更新。

比如说，第一次发送请求时的“user-agent”字段长是一百多个字节，用哈夫曼压缩编码发送之后，客户端和服务器都更新自己的动态表，添加一个新的索引号“65”。那么下一次发送的时候就不用再重复发那么多字节了，只要用一个字节发送编号就好。

```
| 59    | vary                          |                |
| 60    | via                           |                |
| 61    | www-authenticate              |                |
| 62    | ...                           |                |
| 63    | ...                           |                |
| 64    | ...                           |                |
| 65    | user-agent                    | xxxxx          |
+-------+-------------------------------+----------------+
```

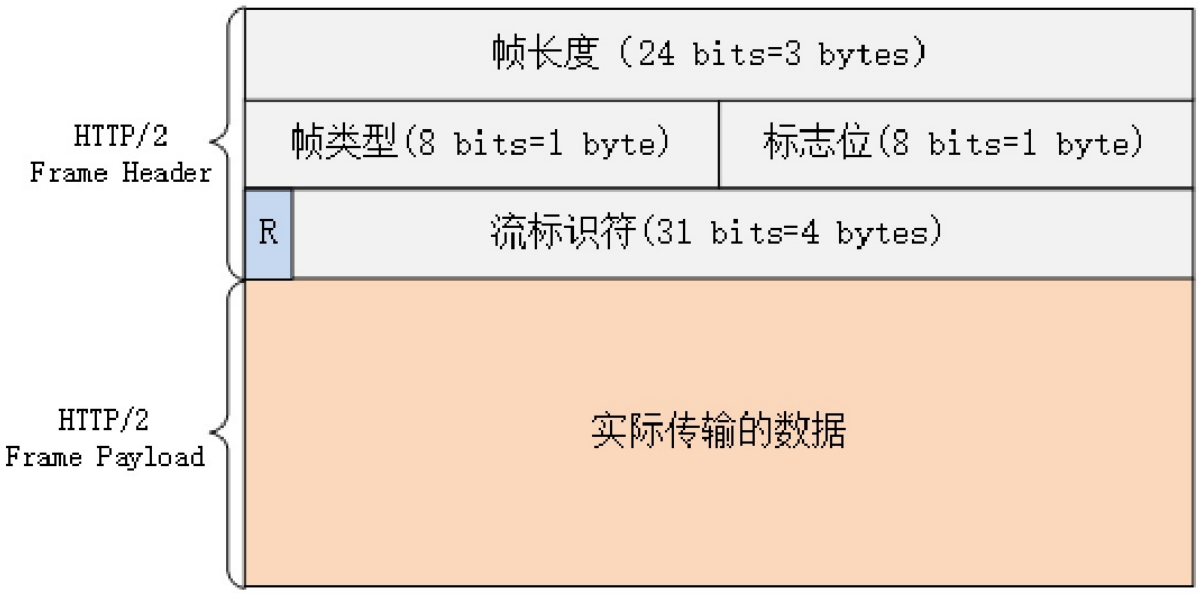
你可以想象得出来，随着在HTTP/2连接上发送的报文越来越多，两边的“字典”也会越来越丰富，最终每次的头部字段都会变成一两个字节的代码，原来上千字节的头用几十个字节就可以表示了，压缩效果比gzip要好得多。

二进制帧

头部数据压缩之后，HTTP/2就要把报文拆成二进制的帧准备发送。

HTTP/2的帧结构有点类似TCP的段或者TLS里的记录，但报头很小，只有9字节，非常地节省（可以对比一下TCP头，它最少是20个字节）。

二进制的格式也保证了不会有歧义，而且使用位运算能够非常简单高效地解析。



帧开头是3个字节的**长度**（但不包括头的9个字节），默认上限是2^14，最大是2^24，也就是说HTTP/2的帧通常不超过16K，最大是16M。

长度后面的一个字节是**帧类型**，大致可以分成**数据帧**和**控制帧**两类，HEADERS帧和DATA帧属于数据帧，存放的是HTTP报文，而SETTINGS、PING、PRIORITY等则是用来管理流的控制帧。

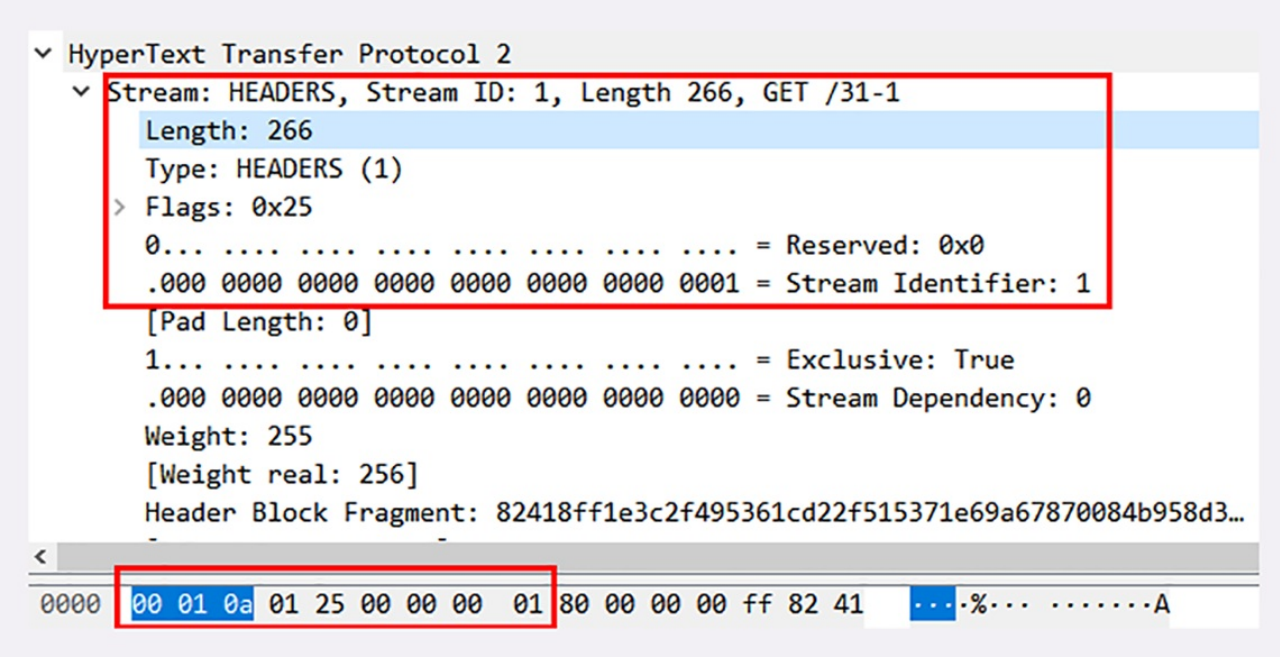
HTTP/2总共定义了10种类型的帧，但一个字节可以表示最多256种，所以也允许在标准之外定义其他类型实现功能扩展。这就有点像TLS里扩展协议的意思了，比如Google的gRPC就利用了这个特点，定义了几种自用的新帧类型。

第5个字节是非常重要的**帧标志**信息，可以保存8个标志位，携带简单的控制信息。常用的标志位有**END_HEADERS**表示头数据结束，相当于HTTP/1里头后的空行（“\r\n”），**END_STREAM**表示单方向数据发送结束（即EOS，End of Stream），相当于HTTP/1里Chunked分块结束标志（“0\r\n\r\n”）。

报文头里最后4个字节是**流标识符**，也就是帧所属的“流”，接收方使用它就可以从乱序的帧里识别出具有相同流ID的帧序列，按顺序组装起来就实现了虚拟的“流”。

流标识符虽然有4个字节，但最高位被保留不用，所以只有31位可以使用，也就是说，流标识符的上限是2^31，大约是21亿。

好了，把二进制头理清楚后，我们来看一下Wireshark抓包的帧实例：



在这个帧里，开头的三个字节是“00010a”，表示数据长度是266字节。

帧类型是1，表示HEADERS帧，负载（payload）里面存放的是被HPACK算法压缩的头部信息。

标志位是0x25，转换成二进制有3个位被置1。PRIORITY表示设置了流的优先级，END_HEADERS表示这一个帧就是完整的头数据，END_STREAM表示单方向数据发送结束，后续再不会有数据帧（即请求报文完毕，不会再有DATA帧/Body数据）。

最后4个字节的流标识符是整数1，表示这是客户端发起的第一个流，后面的响应数据帧也会是这个ID，也就是说在stream[1]里完成这个请求响应。

流与多路复用

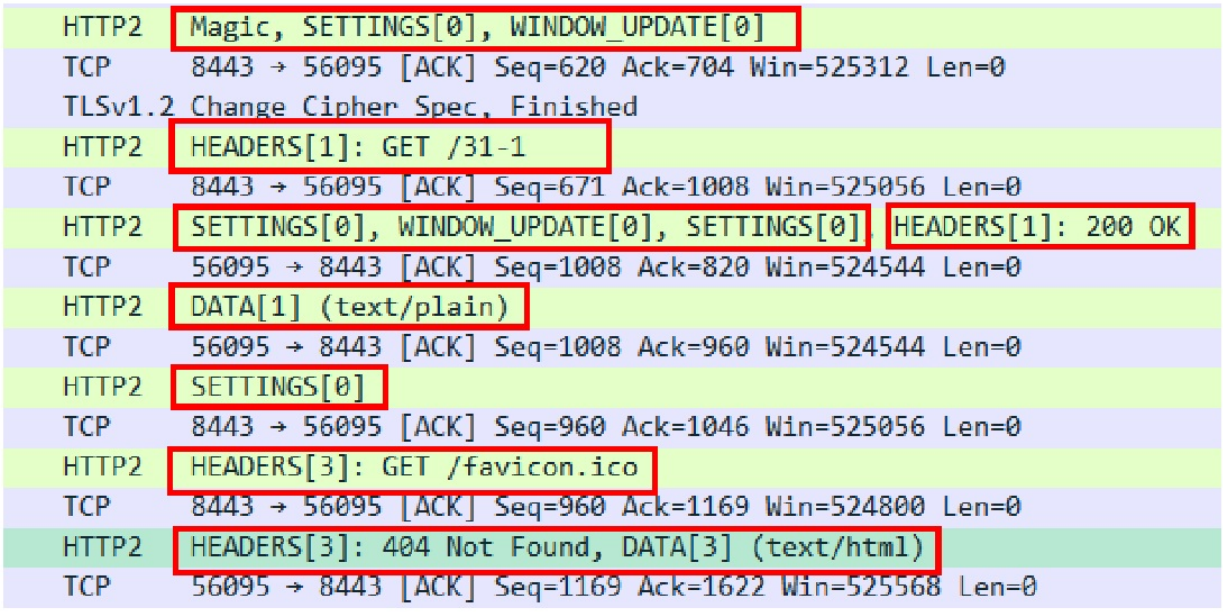
弄清楚了帧结构后我们就来看HTTP/2的流与多路复用，它是HTTP/2最核心的部分。

在上一讲里我简单介绍了流的概念，不知道你“悟”得怎么样了？这里我再重复一遍：**流是二进制帧的双向传输序列**。

要搞明白流，关键是要理解帧头里的流ID。

在HTTP/2连接上，虽然帧是乱序收发的，但只要它们都拥有相同的流ID，就都属于一个流，而且在这个流里帧不是无序的，而是有着严格的先后顺序。

比如在这次的Wireshark抓包里，就有“0、1、3”一共三个流，实际上就是分配了三个流ID号，把这些帧按编号分组，再排一下队，就成了流。



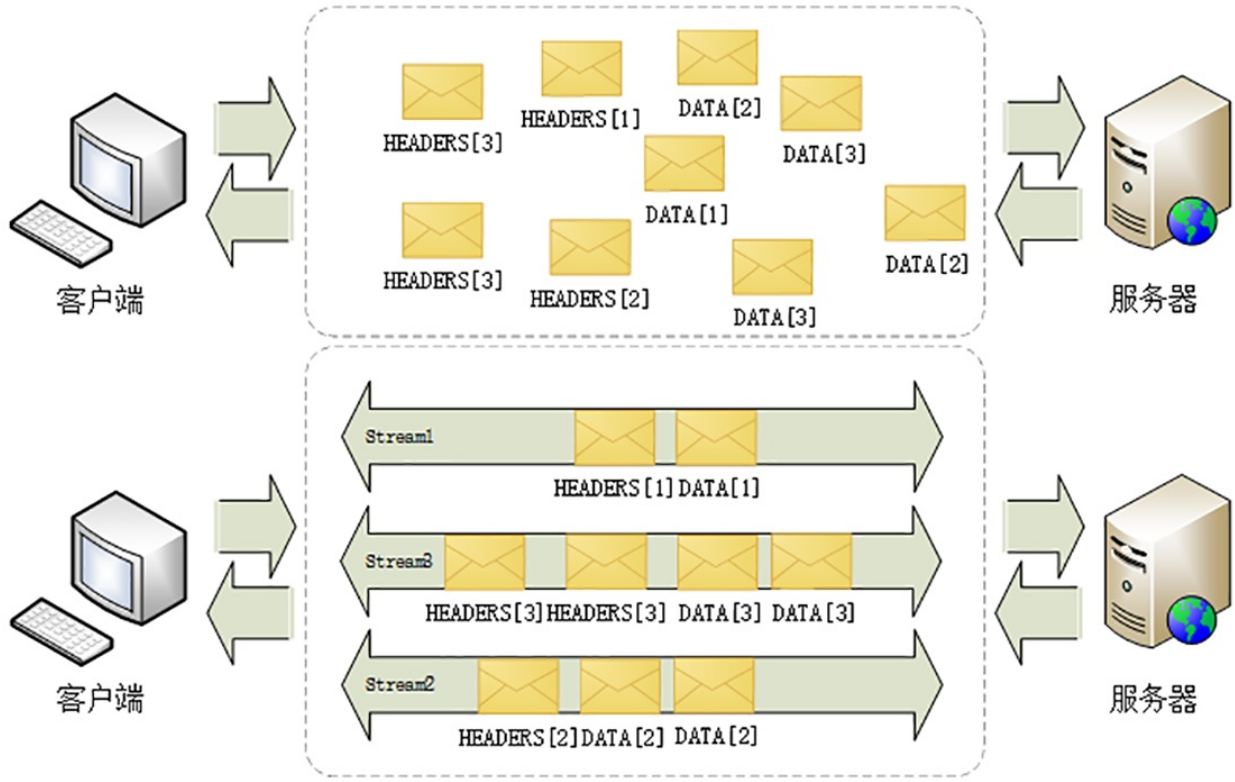
在概念上，一个HTTP/2的流就等同于一个HTTP/1里的“请求-应答”。在HTTP/1里一个“请求-响应”报文来回是一次HTTP通信，在HTTP/2里一个流也承载了相同的功能。

你还可以对照着TCP来理解。TCP运行在IP之上，其实从MAC层、IP层的角度来看，TCP的“连接”概念也是“虚拟”的。但从功能上看，无论是HTTP/2的流，还是TCP的连接，都是实际存在的，所以你以后大可不必再纠结于流的“虚拟”性，把它当做是一个真实存在的实体来理解就好。

HTTP/2的流有哪些特点呢？我给你简单列了一下：

1. 流是可并发的，一个HTTP/2连接上可以同时发出多个流传输数据，也就是并发多请求，实现“多路复用”；
2. 客户端和服务器都可以创建流，双方互不干扰；
3. 流是双向的，一个流里面客户端和服务器都可以发送或接收数据帧，也就是一个“请求-应答”来回；
4. 流之间没有固定关系，彼此独立，但流内部的帧是有严格顺序的；
5. 流可以设置优先级，让服务器优先处理，比如先传HTML/CSS，后传图片，优化用户体验；
6. 流ID不能重用，只能顺序递增，客户端发起的ID是奇数，服务器端发起的ID是偶数；
7. 在流上发送“RST_STREAM”帧可以随时终止流，取消接收或发送；
8. 第0号流比较特殊，不能关闭，也不能发送数据帧，只能发送控制帧，用于流量控制。

这里我又画了一张图，把上次的图略改了一下，显示了连接中无序的帧是如何依据流ID重组成流的。



从这些特性中，我们还可以推理出一些深层次的知识点。

比如说，HTTP/2在一个连接上使用多个流收发数据，那么它本身默认就会是长连接，所以永远不需要“Connection”头字段（keepalive或close）。

你可以再看一下Wireshark的抓包，里面发送了两个请求“/31-1”和“/favicon.ico”，始终用的是“56095<->8443”这个连接，对比一下第8讲，你就能够看出差异了。

又比如，下载大文件的时候想取消接收，在HTTP/1里只能断开TCP连接重新“三次握手”，成本很高，而在HTTP/2里就可以简单地发送一个“RST_STREAM”中断流，而长连接会继续保持。

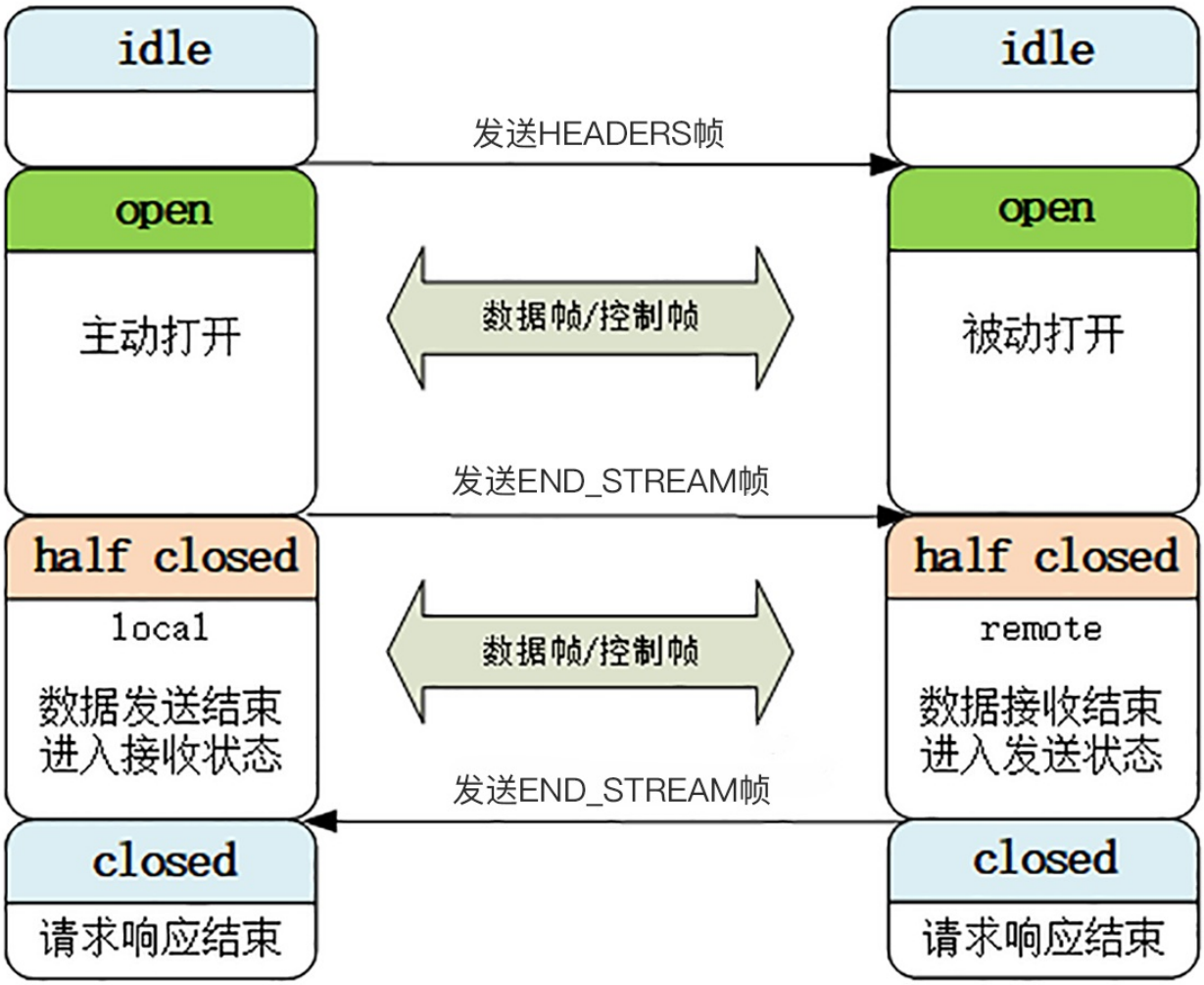
再比如，因为客户端和服务器两端都可以创建流，而流ID有奇数偶数和上限的区分，所以大多数的流ID都会是奇数，而且客户端在一个连接里最多只能发出2^30，也就是10亿个请求。

所以就要问了：ID用完了该怎么办呢？这个时候可以再发一个控制帧“GOAWAY”，真正关闭TCP连接。

流状态转换

流很重要，也很复杂。为了更好地描述运行机制，HTTP/2借鉴了TCP，根据帧的标志位实现流状态转换。当然，这些状态也是虚拟的，只是为了辅助理解。

HTTP/2的流也有一个状态转换图，虽然比TCP要简单一点，但也不那么好懂，所以今天我只画了一个简化的图，对应到一个标准的HTTP“请求-应答”。



最开始的时候流都是“**空闲**”（idle）状态，也就是“不存在”，可以理解成是待分配的“号段资源”。

当客户端发送HEADERS帧后，有了流ID，流就进入了“**打开**”状态，两端都可以收发数据，然后客户端发送一个带“END_STREAM”标志位的帧，流就进入了“**半关闭**”状态。

这个“半关闭”状态很重要，意味着客户端的请求数据已经发送完了，需要接受响应数据，而服务器端也知道请求数据接收完毕，之后就要内部处理，再发送响应数据。

响应数据发完了之后，也要带上“END_STREAM”标志位，表示数据发送完毕，这样流两端就都进入了“**关闭**”状态，流就结束了。

刚才也说过，流ID不能重用，所以流的生命周期就是HTTP/1里的一次完整的“请求-应答”，流关闭就是一次通信结束。

下一次再发请求就要开一个新流（而不是新连接），流ID不断增加，直到到达上限，发送“GOAWAY”帧开一个新的TCP连接，流ID就又可以重头计数。

你再看看这张图，是不是和HTTP/1里的标准“请求-应答”过程很像，只不过这是发生在虚拟的“流”上，而不是实际的TCP连接，又因为流可以并发，所以HTTP/2就可以实现无阻塞的多路复用。

小结

HTTP/2的内容实在是太多了，为了方便学习，我砍掉了一些特性，比如流的优先级、依赖关系、流量控制

等。

但只要你掌握了今天的这些内容，以后再看RFC文档都不会有难度了。

1. HTTP/2必须先发送一个“连接前言”字符串，然后才能建立正式连接；
2. HTTP/2废除了起始行，统一使用头字段，在两端维护字段“Key-Value”的索引表，使用“HPACK”算法压缩头部；
3. HTTP/2把报文切分为多种类型的二进制帧，报头里最重要的字段是流标识符，标记帧属于哪个流；
4. 流是HTTP/2虚拟的概念，是帧的双向传输序列，相当于HTTP/1里的一次“请求-应答”；
5. 在一个HTTP/2连接上可以并发多个流，也就是多个“请求-响应”报文，这就是“多路复用”。

课下作业

1. HTTP/2的动态表维护、流状态转换很复杂，你认为HTTP/2还是“无状态”的吗？
2. HTTP/2的帧最大可以达到16M，你觉得大帧好还是小帧好？
3. 结合这两讲，谈谈HTTP/2是如何解决“队头阻塞”问题的。

欢迎你把自己的学习体会写在留言区，与我和其他同学一起讨论。如果你觉得有所收获，也欢迎把文章分享给你的朋友。

课外小贴士

01 你一定很好奇 HTTP/2“ 连接前言 ”的来历吧，其实把里面的字符串连起来就是“PRISM”，也就是 2013 年斯诺登爆出的“ 棱镜计划 ”。

02 在 HTTP/1 里头字段是不区分大小写的，这在实践中造成了一些混乱，写法很随意，所以 HTTP/2 做出了明确的规定，要求所有的头字段必须全小写，大写会认为是格式错误。

03 HPACK 的编码规则比较复杂，使用了一些特殊的标志位，所以在 Wireshark 抓包里不会直

接看到字段的索引号，需要按照规则解码。

04 HEADERS 帧后还可以接特殊的“CONTINUATION”帧，发送特别大的头，最后一个“CONTINUATION”需要设置标志位 END_HEADERS 表示头结束。

05 服务器端发起推送流需要使用“PUSH_PROMISE”帧，状态转换与客户端流基本类似，只是方向不同。

06 在“RST_STREAM”和“GOAWAY”帧里可以携带 32 位的错误代码，表示终止流的原因，它是真正的“错误”，与状态码的含义是不同的。

精选留言：

- 何用 2019-08-07 20:48:42
  HTTP/2 底层还是依赖 TCP 传输，没有解决队头阻塞的问题啊，这就是为何 HTTP/3 要基于 UDP 来传输 [2赞]

  作者回复2019-08-07 21:44:39
  对，虽然是部分解决，但对于http/1来说已经是一个很大的进步了。

- 何用 2019-08-07 20:31:04
  服务端是不是要为每一个客户端都单独维护一份索引表？连接的客户端多了的话内存不就OOM了嘛 [2赞]

  作者回复2019-08-07 20:56:06
  是的，不过动态表也有淘汰机制，服务器可以自己定制策略，不会过度占用内存。

- 想个昵称好难 2019-08-07 14:58:55
  还有一个问题想请教下老师,您之前在《HTTP的前世今生》上有一段回复是说,只要是HTTP/1.1，就都是文本格式，虽然里面的数据可能是二进制，但分隔符还是文本，这些都会 在“进阶篇”里讲, 不过我看到现在还是有点迷惑,所二进制协议和文本协议的区别是什么呢?可以按照stackoverflow中https://stackoverflow.com/questions/2645009/binary-protocols-v-text-protocols 的回答来理解吗? [2赞]

  作者回复2019-08-07 18:28:07
  指的是协议本身的数据格式，而不是负载（payload）的格式。

  你看http/1，请求行、头、body里的分隔符，都是ASCII码。

  而http/2，是二进制帧，用字节、位来表示信息，没有ASCII码。

  你可以把自己想象成协议的解析器，你看到的协议头是什么格式，文本还是二进制。

- 想个昵称好难 2019-08-07 14:53:44
  老师您好,打扰您实在是抱歉,想请教您一个问题,您在文中说HTTP/2会在两端维护“Key-Value”的索引表,静态表应该是一摸一样的,那动态表俩边一样吗?如果一样的话,同步是比较难做的事情吧,我看RFC文档中是这么写的,”When used for bidirectional communication, such as in HTTP, the encoding and decoding dynamic tables maintained by an endpoint are completely independent, i.e., the request and response dynamic tables are separate.“, 所以我的理解是,动态表在客户端和服务器各自都有俩个表,一个是用来保存客户端发送的message的header,另外一个是保存服务器发送的header, 我看stackoverflow中也是这么写的,https://stackoverflow.com/questions/53003333/how-does-headers-keep-sync-in-both-client-and-server-side-in-http-2, 如果我有哪个地方理解错了,麻烦下老师指点一下

  [2赞]

  作者回复2019-08-07 18:44:40
  是的，客户端和服务器维护各自的动态表，收发各一张表，但字典里的内容必须是一致的，否则索引号就对不上了。

- 许童童 2019-08-07 14:00:50
  HTTP/2 的动态表维护、流状态转换很复杂，你认为 HTTP/2 还是“无状态”的吗？
  还是无状态的，对上层应用来说，动态表维护、流状态转换这些操作对它不可见。

  HTTP/2 的帧最大可以达到 16M，你觉得大帧好还是小帧好？

大帧好，应该小帧需要很多额外的头信息，有数据冗余。小帧可以当出差错时，只转输出错的帧，细粒度控制。

结合这两讲，谈谈 HTTP/2 是如何解决“队头阻塞”问题的。
因为流可以并发，一个流被阻塞了，并不影响其它的流。 [2赞]

作者回复2019-08-07 18:38:24
我个人认为小帧比较好，当然如果在某些特定场景里，比如下载大文件，可以适当加大。

- Geek_54edc1 2019-08-07 11:38:50
3、首先要明确造成“队头阻塞”的原因，因为http1里的请求和应答是没有序号标识的，导致了无法将乱序的请求和应答关联起来，也就是必须等待起始请求的应答先返回，则后续请求的应答都会延迟，这就是“队头阻塞”，而http2采用了虚拟的“流”，每次的请求应答都会分配同一个流id，而同一个流id里的帧又都是有序的，这样根据流id就可以标识出同一次的请求应答，不用再等待起始请求的应答先返回了，解决了“队头阻塞” [2赞]

作者回复2019-08-07 18:31:22
http/1里的请求都是排队处理的，所以有队头阻塞。

http/2的请求是乱序的，彼此不依赖，所以没有队头阻塞。

- sunxu 2019-08-07 08:54:33
想问一下，nginx前端采用http2, 反向代理到应用服务使用的http1.1, 这种方式对请求响应有提升吗？ [2赞]

作者回复2019-08-07 18:34:46
Nginx作为代理，实际上就把传输链路拆分成了两个部分，下游因为使用了http/2，所以肯定会有性能提升。

而上游还是http/1，所以瓶颈就在这里，但因为后台系统的服务能力都很强，网络也好，所以不会有太大影响。

当然，如果全用http/2就更好了。

- -W.LI- 2019-08-07 08:42:08
1.还是无状态,流状态只是表示流是否建立，单次请求响应的状态。并非会话级的状态保持
2.小帧好，少量多次，万一拥堵重复的的少。假设大帧好，只要分流不用分帧了。
3.每一个请求响应都是一个流，流和流之间可以并行，流内的帧还是有序串行。 [2赞]

作者回复2019-08-07 18:35:32
good。

- 许童童 2019-08-07 12:02:40
老师你好，nginx反向代理与服务端应用间有必要使用HTTP2吗？对性能提升大吗？ [1赞]

作者回复2019-08-07 18:36:33
如果有可能就尽量用http/2，HPack、流都可以提升性能，具体能提升多少就要看应用场景了，需要做测试。

- Geek_54edc1 2019-08-07 11:18:02
1、http的“无状态”是指对事务处理没有记忆，每个请求之间都是独立的，这与HPACK算法里的动态表

、流状态转换是两回事。HPACK算法里维护动态表是用于头部压缩，而流状态转换只是表示一次请求应答里流的状态，都不会记录之前事务的信息 [1赞]

作者回复2019-08-07 18:31:55
说的很好，也可以这么表述“语法上有状态，语义上无状态”。

- -W.LI- 2019-08-07 08:44:08
老师好!TCP网络不好的时候会降速，http2的话是一个帧没收到就会导致TCP降速么? [1赞]

作者回复2019-08-07 18:35:19
是的，会发生tcp层的队头阻塞，下一次http/3会讲。

- 信信 2019-08-08 20:20:13
上次留言说：课程里的所有链接都返回200，和访问http://www.chrono.com/一个效果，是一个由众多js，css组成的网站。。。
作者: 建议打开开发者工具，看看uri是如何处理的。

比如http://www.chrono.com/18-1?dst=/15-1?name=a.json，应该是跳转到15-1。

有用开发者工具，显示的状态码就是200，并未跳转到15-1。。。。

- sunözil 2019-08-08 12:52:21
老师 ，想请教一下，作为一个软件开发商，只交付系统不涉及硬件。需要在哪些方面保障系统安全性?

- Geek_54edc1 2019-08-07 11:44:16
2、小帧好，如果多个流的帧可以在同一个tcp数据段发送的话，就可以提高网络利用率

作者回复2019-08-07 18:32:08
great。

- 我行我素 2019-08-07 10:51:29
请求复用。拆分传输，一个请求头被阻塞，不会影响其他请求

作者回复2019-08-07 18:32:38
√