15 | Nginx的499状态码是怎么回事？

2022-02-23 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉
时长 21:45 大小 19.93M

你好，我是胜辉。

“实战一：TCP 真实案例解密篇”刚刚结束。在过去的十几讲里，我们全面回顾了 TCP 的各种技术细节，从握手到挥手，从重传等容错机制，到传输速度等效率机制，应该说也是对我们的 TCP 知识做了一个全面的“体检”。如果你发现自己对 TCP 的掌握还有不少漏洞，也别着急，可以回头复习一下相应部分的内容，或者在留言区提问，我会给你解答。

从这节课开始，我们要进入网络排查的“实战二：应用层真实案例解密篇”了。今天要给你讲解的是一个关于 Nginx 的排查案例。

Nginx 的 499 状态码是怎么回事？

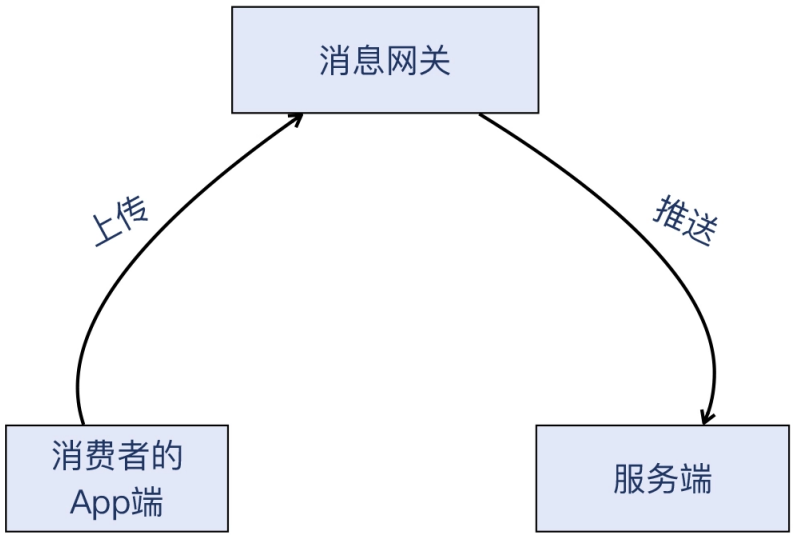
你肯定听说过 Nginx，或者经常用到它。作为一个高性能的 HTTP 和反向代理服务器，Nginx 不管是用来搭建 Web Server，还是用作负载均衡都很合适，并且它可供配置的日志字段也很丰富，从各类 HTTP 头部到内部的性能数据都有。

不过，你在日常维护 Nginx 时有没有遇到过这种情况：**在 Nginx 的访问日志中，存在 499 状态码的日志。**但是常见的 4xx 家族的状态码只有 400、401、403、404 等，这个 499 并未在 HTTP 的 RFC 文档中定义，是不是很奇怪？

这个 499 错误日志，在流量较大的场景下，特别是面向 Internet 的 Web 站点场景下还是很常见的 。但如果你遇到过，第一感觉可能会是一头雾水，不知道 499 这个状态码具体是用来干啥的，因为确实跟其他的 400 系列状态码太不同了。

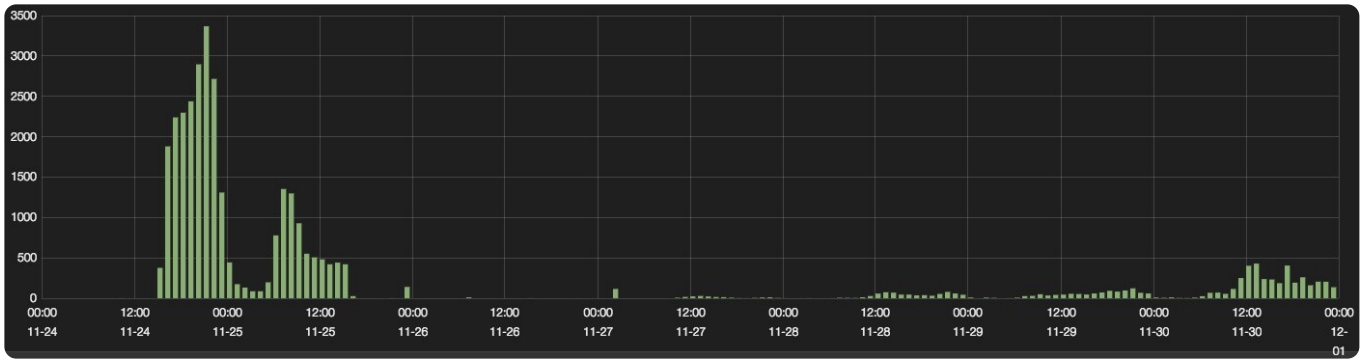
我在公有云的时候，做过的一个案例正好是关于 Nginx 的 499 日志。当时一位客户向我反馈：他们的 Nginx 服务器会连续几天记录较多的 499 错误日志，之后几天可能趋零，然后再回升，整体状况起伏不定。

这个客户经营的是 To C 的电子产品，跟手机端 App 协同工作。这个 App 会定时把消息上传到微信消息网关，后者再把这些消息推送到该客户的服务端（在公有云上）做业务处理，整体的消息量约每日三十万条。那么，对消息网关来说，这个服务端就是一个 Web 回调接口。下面是架构简图：



极客时间

他们给我提供了 499 日志趋势图：



由于大量 499 日志的存在，客户非常担心业务已经受到影响，比如他们的终端消费者是否经常上传数据失败？是否已经严重影响了消费者的体验？所以，我们需要搞清楚 499 错误日志的含义。

那么，499 这个状态码本身能帮到我们什么呢？我们可以查一下它在 Nginx 里的官方定义：

> NGX_HTTP_CLIENT_CLOSED_REQUEST | 499

可是，什么叫 client closed request（客户端关闭了请求）呢？好像说了跟没说也没太大区别。我们知道 499 是客户端关闭请求引起的，那又是什么原因，引起了“客户端关闭了请求”呢？关于这个问题，Nginx 的文档并没有提及。

有一句话叫做“解决问题的办法，可能不在问题自身所处的这个层面”。**应用层日志，其实记录的依然是表象。**更深层次的原因，很可能在更底层，比如在传输层或者网络层。

所以，搞清楚 499 这个状态码是什么意思，对于我们来说，不仅是理解这个 499 码的底层含义，而且通过这种排查，我们还能掌握一套**对 HTTP 返回码进行网络分析的方法**。这种方法，对于维护好 Nginx 以及其他 Web 服务，都是很有帮助的。

那么接下来，我们就根据这个案例，一起探讨下如何用抓包分析，来拆解 HTTP 返回码的真正含义。

锚定到网络层

如前面所说，我是选择用**抓包分析**这个方法来展开排查的。之所以采用这个方法，是因为我前面也说过，从软件文档已经无法查清楚问题根因了，所以需要下沉到网络层排查。如

果你在处理应用层故障，比如 HTTP 异常返回码（4xx 和 5xx 系列）场景中，也遇到了在应用层找不到答案的情况，你就可以考虑采用抓包分析的方法。

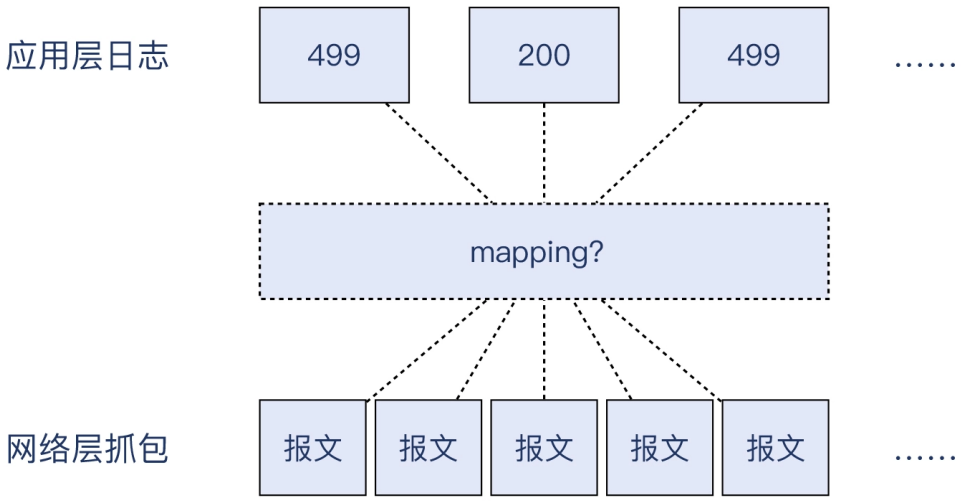
> 补充：下文中的“客户端”都指微信消息网关，“服务端”指这个客户在公有云的服务器。

这样，我在**服务端**使用 tcpdump 工具做了抓包，然后用 Wireshark 打开抓包文件展开分析。从抓包文件中，我一般会寻找一些比较可疑的报文。正好，这次抓包里有不少 RST 报文，于是我过滤出了一个典型的带 RST 报文的 TCP 流，请看下图：

No.	Time	SrcPort	DstPort	TTL	TCP Seglen	Info
1	0.000000	14492	80	47	0	14492 → 80 [SYN] Seq=0 Win=5760 Len=0[Packet size limited during capture]
2	0.000014	80	14492	64	0	80 → 14492 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0[Packet size limited during capture]
3	0.030001	14492	80	47	0	14492 → 80 [ACK] Seq=1 Ack=1 Win=45 Len=0
4	2.997629	14492	80	47	308	14492 → 80 [PSH, ACK] Seq=1 Ack=1 Win=45 Len=308
5	0.000081	80	14492	64	0	80 → 14492 [ACK] Seq=1 Ack=309 Win=123 Len=0
6	2.000016	14492	80	47	0	[TCP Previous segment not captured] 14492 → 80 [FIN, ACK] Seq=777 Ack=1 Win=45 Len=0
7	0.000012	80	14492	64	0	[TCP Dup ACK 5#1] 80 → 14492 [ACK] Seq=1 Ack=309 Win=123 Len=0[Packet size limited during capture]
8	16.028791	14492	80	47	468	[TCP Retransmission] 14492 → 80 [PSH, ACK] Seq=309 Ack=1 Win=45 Len=468
9	0.000010	80	14492	64	0	80 → 14492 [ACK] Seq=1 Ack=778 Win=131 Len=0
10	0.000538	80	14492	64	68	80 → 14492 [FIN, ACK] Seq=1 Ack=778 Win=131 Len=68
11	0.030135	14492	80	47	0	14492 → 80 [RST] Seq=778 Win=0 Len=0

相信你也一眼就看到了那个结尾处的 RST。但问题是，**这个 TCP 流一定跟 499 日志有关系吗？**

得益于 TCP/IP 的精妙的分层设计，应用层只需要通过系统调用，就可以像使用文件 IO 那样使用网络 IO，具体的网络细节都由内核处理了。可是由此也带来了一个问题：**以应用层的视角，是无法“看到”具体的网络报文的。**



极客时间

我们需要根据一些关键信息来确定应用层日志跟网络报文的对应关系。比如在这里，我可以确认上面这个带有 RST 的 TCP 流，就是日志中记录的一条 499 日志记录。这是如何做到的呢？

就是因为以下三点。

客户端 IP：日志中的 remote IP 跟抓包文件里面的 IP 符合。

时间戳：日志的时间戳也跟这个 TCP 流的时间吻合。

应用层请求：日志里的 HTTP URL 路径和这个 TCP 流里的 URL 相同。

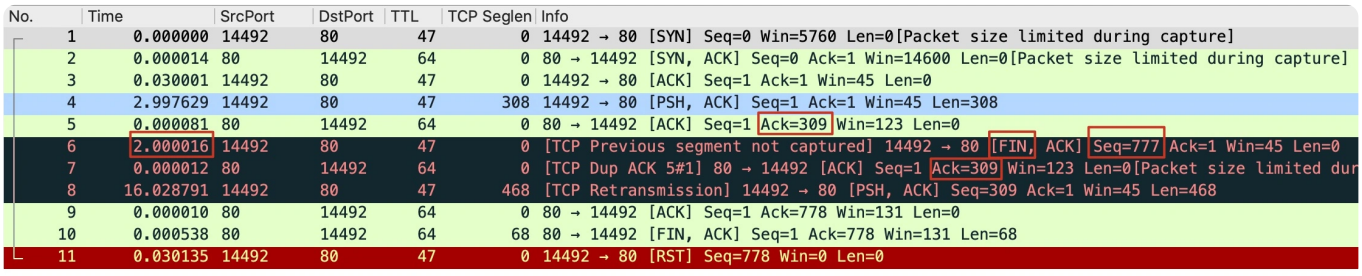
> 补充：如果你对第 4 讲有印象，应该记得当时也是用类似的方式找到了应用日志跟报文的对应关系。

实际上，在真实的抓包分析场景中，“如何把应用层问题跟网络层抓包关联起来”，始终是一个关键环节。同时，这也是比较令人困扰的关键技术障碍，很多人就是在这一关前败下阵来，导致没有办法真正彻底地查到根因。所以这里的方法可以作为给你的参考，当你以后再处理这种关键环节的时候，也可以根据上面提到的三个维度的信息，即 IP、时间戳、应用层请求（包括 URL 和 header），来达到“把应用层问题锚定到网络层数据包”的目的。

好，既然确定这个流就是代表了一次 499 事件，那么我们就需要好好分析一下这些报文里面的文章了。

TCP 流的解读

这里，你可以先注意一下我在下图的这个 TCP 流示意图中，标记出的红框部分，在后续的分析过程中，我会重点分析这几个部分。



No.	Time	SrcPort	DstPort	TTL	TCP Seglen	Info
1	0.000000	14492	80	47	0	14492 → 80 [SYN] Seq=0 Win=5760 Len=0[Packet size limited during capture]
2	0.000014	80	14492	64	0	80 → 14492 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0[Packet size limited during capture]
3	0.030001	14492	80	47	0	14492 → 80 [ACK] Seq=1 Ack=1 Win=45 Len=0
4	2.997629	14492	80	47	308	14492 → 80 [PSH, ACK] Seq=1 Ack=1 Win=45 Len=308
5	0.000081	80	14492	64	0	80 → 14492 [ACK] Seq=1 Ack=309 Win=123 Len=0
6	2.000016	14492	80	47	0	[TCP Previous segment not captured] 14492 → 80 [FIN, ACK] Seq=777 Ack=1 Win=45 Len=0
7	0.000012	80	14492	64	0	[TCP Dup ACK 5#1] 80 → 14492 [ACK] Seq=1 Ack=309 Win=123 Len=0[Packet size limited dur
8	16.028791	14492	80	47	468	[TCP Retransmission] 14492 → 80 [PSH, ACK] Seq=309 Ack=1 Win=45 Len=468
9	0.000010	80	14492	64	0	80 → 14492 [ACK] Seq=1 Ack=778 Win=131 Len=0
10	0.000538	80	14492	64	68	80 → 14492 [FIN, ACK] Seq=1 Ack=778 Win=131 Len=68
11	0.030135	14492	80	47	0	14492 → 80 [RST] Seq=778 Win=0 Len=0

首先是报文 1~3，表示 TCP 握手成功。

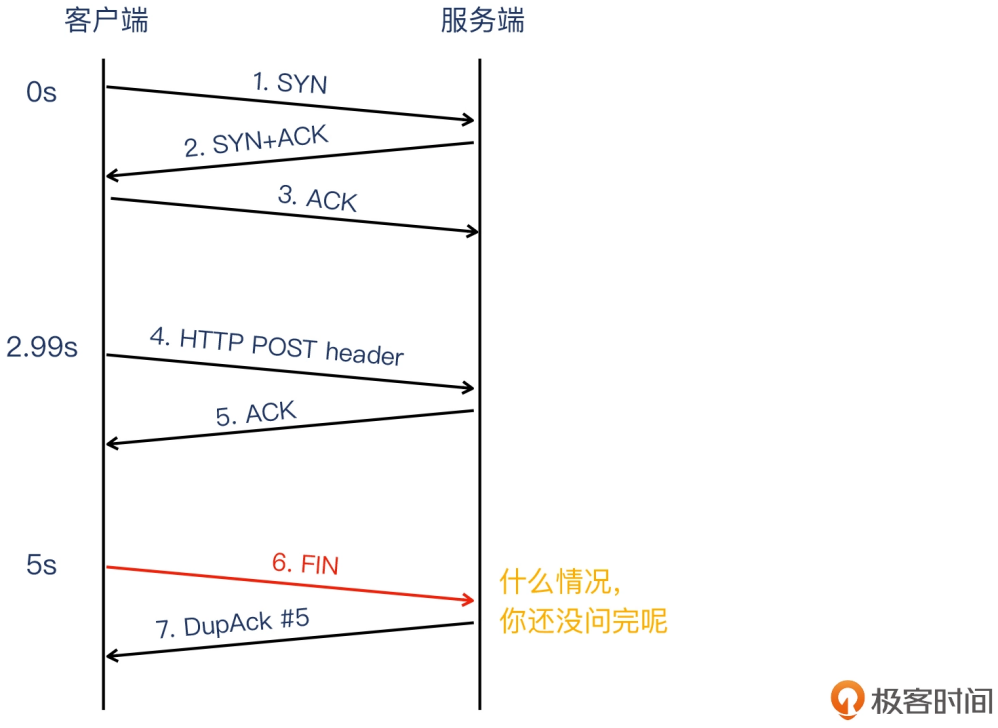
然后是报文 4（客户端发出），表示客户端（消息网关）向服务器发送报文，这个报文里只包含 HTTP header，其声明该请求为 POST 方法，但不含 POST body。这其实是正常的，因为 HTTP 协议就是这样规定的，数据的先后顺序是：先 header（包含 method、URL、headers），后 body。所以，既然方法（method）和 URL 单独位于一个报文里面了，那么按顺序来说，body 就是在后续的报文里面。

接下来是报文 5（服务端发出），它是一个确认报文。它的意思是：我（服务端）确认收到了你（客户端）发过来的报文 4。

紧接着是报文 6（客户端发出），此时距离上一个报文的时间是 2 秒。这个报文被 Wireshark 标记为了红色，注释为 TCP Previous segment not captured，意思是它之前的 TCP 报文段没有被抓到。

什么叫做“之前的 TCP 报文”呢？其实就是按 TCP 序列号顺序，排在当前报文之前的报文。我对这个 6 号报文标注了 3 处红框，它们都有很重要的含义。这里我们先关注下右边一个红框圈出来的 FIN 标志位，这说明，**这是一个客户端主动关闭连接的报文。**

我们可以把到目前为止的报文情况，用下面这个示意图来表示：



你看这里是不是很奇怪？明明 HTTP POST 请求的 body（也称为 HTTP 载荷）部分还没发过来，这个客户端就嚷嚷着要关闭连接了？这就好比有个朋友跟你说：“我有个事情要

你帮忙，嗯，拜拜~”，你刚听到上半句他的求助意向，还没听到这个忙具体是什么，他就跟你说再见了。惊不惊喜，意不意外？你可能暂时看不出这里究竟出了什么问题，不过没关系，先放一放。

我们继续看报文 7（服务端发出）。服务器收到了 FIN+ACK 报文（6 号报文），但发现序列号并不是它期待的 309，而是 777，于是服务器 TCP 协议栈判断：有一个长度为 777-309=468 的 TCP 段（TCP segment）丢失了。

按 TCP 的约定，这时候服务端只可以确认其收到的字节的最后位置，在这里，就是上一次（报文 5）的 ACK 位置。形式上，报文 7 就成了一个 DupAck（重复确认）。

当客户端收到 DupAck 的时候，它就需要长一个心眼了：“情况有点微妙，如果凑满 3 个 DupAck 可能有丢包啊”。

> 补充：如果凑满 3 个 DupAck 就重传的机制，被称为快速重传机制，我们在第 12 讲我们有深入学习过。

为了帮助理解，这里我再展示下报文 4 的 TCP 信息：

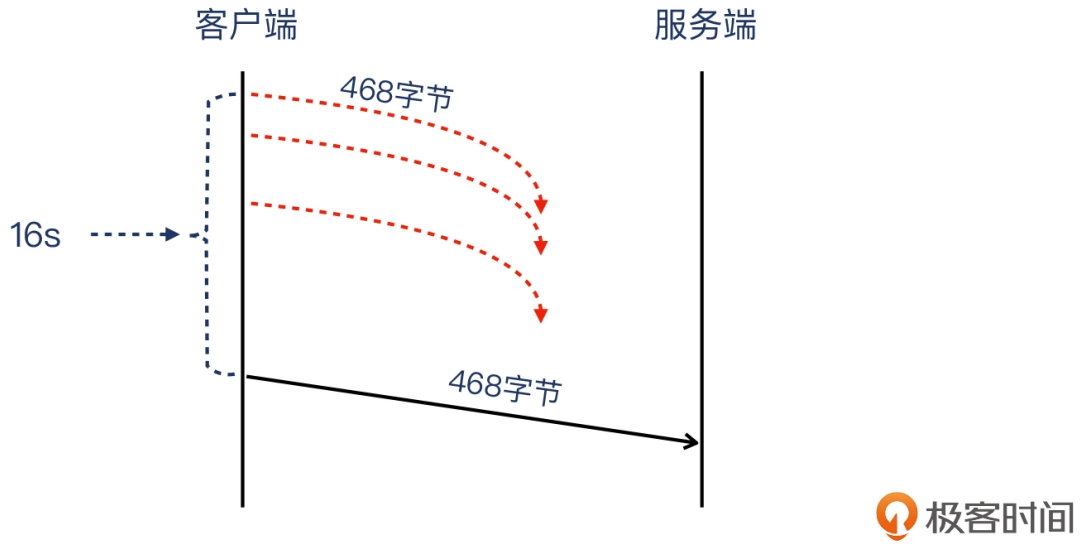
```
▼ Transmission Control Protocol, Src Port: 14492, Dst Port: 80, Seq: 1, Ack: 1, Len: 308
    Source Port: 14492
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 308]
    Sequence number: 1    (relative sequence number)
    Sequence number (raw): 122545270
    [Next sequence number: 309    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Acknowledgment number (raw): 974992479
    0101 .... = Header Length: 20 bytes (5)
```

那么，按 TCP 的设计，客户端将要发送的下一个报文的序列号（309）= 本次序列号（1）+ 本次数据长度（308），也就是图中的 Next sequence number。

我们再来看报文 8（客户端发出），过了 16 秒之久，客户端**重传**了这个报文，包含 POST body 的数据，长度为 **468** 字节。你看，这是不是就跟前面说的 777-309=468 对应起来了。

可能你在这里又有点困惑了，明明这个 468 字节的报文是第一次出现，怎么就算重传了呢？

其实是因为，这个抓包文件是在服务端生成的，所以它的视角，是无法看到多次传送同样这个报文的现象的。但我判断，在客户端抓包的话，一定可以看到这个 468 字节的报文被试图传送了多次。



我们就以服务端视角来判断，一开始这个报文应该是走丢了，没有达到服务端，所以没有在这个服务端抓包文件里现身。又因为过了 16 秒之久才到达，很可能不是单纯一次重传，而是多次重传后才最终到达的。因此从这一点上讲，确实属于重传。

我们继续分析。接下来就是报文 9，服务端对这个 POST body 的数据包回复了确认报文。

最后是报文 10，服务端发送了 HTTP 400 的响应报文给消息网关。这个信息并没有被 Wireshark 直接按 HTTP 格式进行展示，但是因为 HTTP 是文本编码的，所以我们可以鼠标选中 Transmission Control Protcol 部分，在底下的文本栏直接看到 HTTP 400 这段文本：

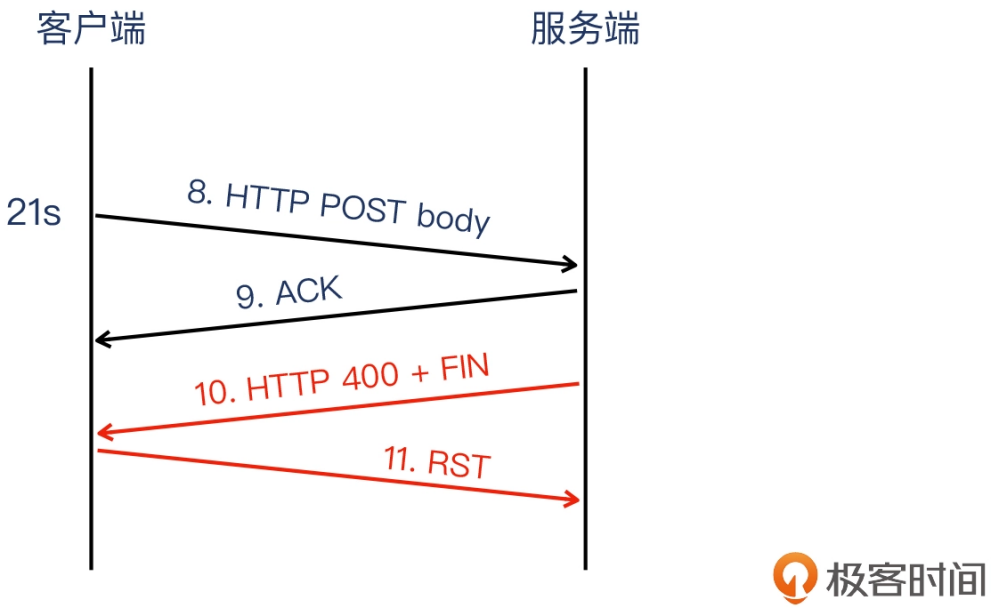
```
    9   0.000010  80     14492  64   0  80 → 14492 [ACK] Seq=1 Ack=778 Win=16768 Len=0
   10   0.000538  80     14492  64   68 Continuation[Packet size limited during capture]
   11   0.030135  14492  80     47   0  14492 → 80 [RST] Seq=778 Win=0 Len=0
```

```
▶ Frame 10: 122 bytes on wire (976 bits), 80 bytes captured (640 bits) on interface unknown, id 0
▶ Ethernet II, Src: 52:60:00:10:ff:49, Dst: 20:0b:c7:3b:c6:0c
▶ Internet Protocol Version 4, Src: 123.59.70.99 (123.59.70.99), Dst: 101.226.62.77 (101.226.62.77)
▶ Transmission Control Protocol, Src Port: 80, Dst Port: 14492, Seq: 1, Ack: 778, Len: 68
▶ Hypertext Transfer Protocol
  [Packet size limited during capture: HTTP truncated]
```

```
0000  20 0b c7 3b c6 0c 52 60  00 10 ff 49 08 00 45 00   ··;··R` ···I··E·
0010  00 6c 64 df 40 00 40 06  6f df 7b 3b 46 63 65 e2   ·ld·@·@· o·{;Fce·
0020  3e 4d 00 50 38 9c 3a 1d  34 5f 07 4d e7 7f 50 11   >M·P8·:· 4_·M··P·
0030  00 83 66 2c 00 00 48 54  54 50 2f 31 2e 30 20 34   ··f,··HT TP/1.0 4
0040  30 30 20 62 61 64 20 72  65 71 75 65 73 74 20 68   00 bad r equest h
```

有趣的是，这个 **HTTP 400 报文也是带 FIN 标志位的**，也就是服务端操作系统“图省事”了，把应用层的应答数据（HTTP 400），跟操作系统对 TCP 连接关闭的控制报文（这个 FIN），合并在同一个报文里面了。也就是我们在第 3 讲提到的搭顺风车（Piggybacking），提升了网络利用效率。

这个阶段的报文图示如下：



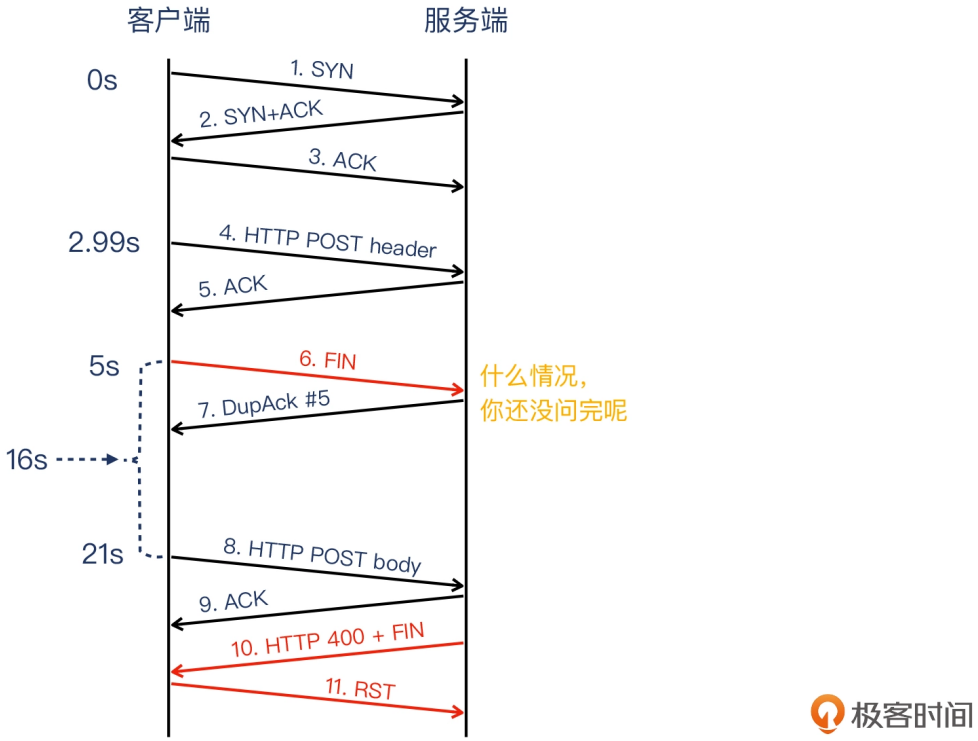
那么，从这些报文的顺序来看，我们会发现它确实是有问题的。特别是有以下几个疑点：

服务端先收到了 HTTP header 报文，随后并没有收到期望的 HTTP body 报文，而是收到了 FIN 报文，即客户端试图关闭连接。这个行为十分古怪，要知道 HTTP 请求还没发送到服务端，服务端回复 HTTP 响应更是无从谈起，这个时候客户端发送 FIN 就不符合常理了（即前面说的朋友求帮忙的类比）。

服务端回复了 HTTP 400，并且也发送 FIN 关闭了这个连接。

客户端回复 RST 彻底关闭这个连接。

而把上面这几条信息综合起来看，你有没有发现一个重要的线索？**客户端先发送了 FIN，之后才发送 POST body。**现在让我们把全部过程拼接起来，看一下全景图：



这么古怪的行为，可以描述为“**服务端还没回复数据而客户端已经要关闭连接**”。按照 499 的官方定义，这种行为就被 Nginx 判定为了 499 状态。对内表现为记录 499 日志，对外表现为回复 HTTP 400 给消息网关。

所以，在服务端的 Nginx 日志中，就留下了大量的 499 日志条目；而在消息网关那头，如果它也做 Web 日志的话，相信就不是 499 日志，而是 400 的报错了。

那么到这里，问题是水落石出了吗？其实不是。

从现象到本质

我们还需要搞清楚最底层的疑问：为什么客户端先发送 FIN，然后才发送 POST body？

我们回到 Wireshark 窗口，再次关注下 6 号报文：

```
5      0.000081 80        14492     64        0 80 → 14492 [ACK] Seq=1 Ack=309 Win=15744 Len=0
6      2.000016 14492     80        47        0 [TCP Previous segment not captured] 14492 → 80 [FIN, ACK]
```

它离上一个报文相差了 2 秒，而我们知道这个信息，是因为 Wireshark 很友好地显示了报文之间的间隔时长。

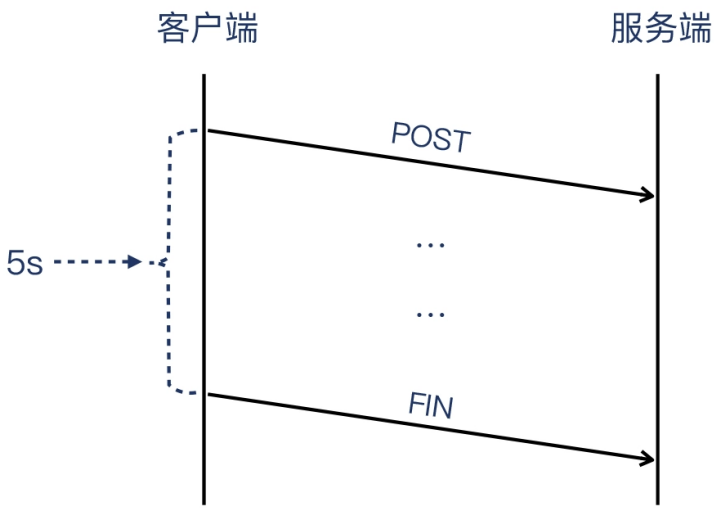
我们再往前看 4 号报文：

```
3      0.030001 14492     80        47        0 14492 → 80 [ACK] Seq=1 Ack=1 Win=5760 Len=0
4      2.997629 14492     80        47      308 Continuation[Packet size limited during capture]
```

离 3 号报文相差了 2.997 秒，几乎就是 3 秒整了。那么加起来，6 号报文离 TCP 握手完成，正好隔了 **5 秒整**。

一般出现这种整数，就越发可疑了，因为如果是系统或者网络的错乱导致的行为，其时间分布上应该是**随机的**，不可能卡在整数时间上。就我的经验来看，**这往往跟某种人为的设置有关系**。

所以，经过我的提醒，客户自己仔细查看了微信网关的使用文档，果然发现了它确实有 5 秒超时的设置。也就是说，如果一个 HTTP 事务（在这个例子里是 HTTP POST 事务）无法在 5 秒内完成，就关闭这个连接。



这个“无法完成”，在这个抓包里面体现为：HTTP header 报文发过去了，但 HTTP body 报文没有一起过去（网络原因导致）。而由于初始阶段报文少，**无法凑齐 3 个 DupAck**，所以快速重传没有被启动，只好依赖超时重传（关于超时重传的知识在第 12 讲也有详细的介绍），而且这多次超时重传也失败了，服务端只好持续等待这个丢失的报文。5 秒钟过后，客户端（微信消息网关）没有收到服务端的响应，就主动关闭了这次连接（可以下次再试，这次就不继续干等了）。

也就是说，这个场景里的 Nginx 499 错误日志的产生，主要是由于两个因素造成的：

> **“消息网关—> 服务器”方向上的一个 TCP 包丢失（案例里是 HTTP POST body 报文），引起服务端空闲等待；**
>
> **消息网关有一个 5 秒超时的设置，即连接达到 5 秒时，消息网关就发送 FIN 关闭连接。**

所以到这里，想必你也明白了这里的逻辑链条，也就是：

> 要解决 499 报错的问题，就需要解决 5 秒超时的问题；
>
> 要解决 5 秒超时的问题，就需要解决丢包问题；
>
> 要解决丢包的问题，就需要改善网络链路质量。

最根本的解决方案，就是如何确保客户端到服务端的**网络连接**可靠稳定，使得类似的报文延迟的现象降到最低。只要不丢包不延迟，HTTP 事务就能在 5 秒内完成，消息网关就不会启动 5 秒超时断开连接的机制。

这样，我们跟客户还有网关的工程师一起配合，确实发现网关到我们公有云的一条链路有问题。更换为另外一条链路后，丢包率大幅降低，问题得到了极大改善。虽然还是有极小比例的错误日志（大约万分之一），但是这对于客户来说，完全在可接受范围之内了。

另外，因为丢包的存在，客户端的 FIN 报文跟 HTTP POST body 报文一样，也可能会丢失。不过，无论这个 FIN 是否被服务端及时收到，这次 HTTP 事务本身也已经在客户端被记为失败了，也就是不改变这件事的结果。

你可能会问了：链路丢包这种问题应该挺明显的，为什么没有在第一时间发现呢？

这其实是多种因素导致的：

> 我们虽然对主要链路的整体状况有细致的监控，但这里的网关到客户的公有云服务属于“点到点”的链接，本身也属于客户自身的业务，公有云难以对这种情况做监控，理想情况是客户自己来实现监控。
>
> 客户的消息量很大，哪怕整体失败比例不高，但乘以绝对的消息量，产生的错误的绝对数也就比较可观了。

至于 Nginx 为什么要“创造”499 这个独有的状态码的原因，其实在 Nginx 源码的注释部分里，已经写得非常清楚了。它并非标新立异，而确实是为了弥补标准 HTTP 协议的不足。相关代码如下：

复制代码

```
/*
 * HTTP does not define the code for the case when a client closed
 * the connection while we are processing its request so we introduce
 * own code to log such situation when a client has closed the connection
 * before we even try to send the HTTP header to it
 */
#define NGX_HTTP_CLIENT_CLOSED_REQUEST     499
```

翻译过来就是：HTTP 并没有对服务端还在处理请求的时候客户端就关闭连接的情况，做一个状态码的定义。所以我们定义了自己的状态码（499），以记录这种“还没来得及发送返回，客户端就关闭了连接”的情形。

小结

现在，我们就清楚在这个例子里，造成 499 状态码的根因了。不过基于普适性的应用需求，我想把这个案例再延伸拓展一下，希望可以帮助你了解到更多的知识，并且在理解了这些知识点之后，你能够有效应用在类似的 HTTP 异常码的故障排查里。

首先，我们要知道，**Nginx 499 是 Nginx 自身定义的状态码，并非任何 RFC 中定义的 HTTP 状态码**。它表示的是“Nginx 收到完整的 HTTP request 前（或者已经接收到完整的 request 但还没来得及发送 HTTP response 前），客户端试图关闭 TCP 连接”这种反常情况。

第二，**超时时间跟 499 报错数量也有直接关系。**如果我们有办法延长消息网关的超时时间，比如从 5 秒改为 50 秒，那么客户端就有比较充足的时间去等待丢失的报文被成功重传，从而在 50 秒内完成 HTTP 事务，499 日志也会少很多。

第三，**我们要关注网络延迟对通信的影响。**比如客户端发出的两个报文（报文 3 和报文 4）间隔了 3 秒钟，这在网络通信中是个非常大的延迟。而造成这么大延迟的原因，会有两种可能：一是消息网关端本身是在握手后隔了 3 秒才发送了这个报文，属于**应用层问题**；二是消息网关在握手后立刻发送了这个报文，但在公网上丢失了，微信消息网关就根据“超时重传”的机制重新发了这个报文，并于 3 秒后到达。这属于**网络链路问题。**

由于上面的抓包是在服务端做的，所以未到达服务器的包自然也不可能抓到，也就是无法确定是具体哪一种原因（客户端应用层问题或网络链路问题）导致，但这并不影响结论。

最后一点，就是我们要清楚，**公网上丢包现象不可能完全消失。**千分之一左右的公网丢包率属于正常范围。由于客户发送量比较大（这是主要原因），加上微信消息网关设置的 5 秒超时相对比较短（这是次要原因），这两个因素一结合，问题就会在这个案例中被集中暴露出来。

那么，像上面第二点说的那样，设置更长的超时阈值（比如 50 秒）能解决问题吗？相信出错率会降低不少，但是这样新的问题也来了：

消息网关会有更多的资源消耗（内存、TCP 源端口、计算能力等）；

消息网关处理事务的平均耗时会增加。

所以，选择 5 秒应该是一个做过权衡后的适中的方案。

而从排查的方法论上来说，对于更广泛的应用层报错日志的排查，我的推荐是这样的：

首先查看应用文档，初步确定问题性质，大体确定排查方向。

通过对比应用日志和抓取的报文，在传输层和网络层寻找可疑报文。在这一步，可以采用以下的比对策略来找到可疑报文：

- 日志中的 IP 跟报文中的 IP 对应；
- 日志和报文的时间戳对应；

- 应用层请求信息和报文信息对应。

结合协议规范和报文现象，推导出根因。

思考题

给你留两个思考题，欢迎在留言区分享你的答案和思考，我们一起交流讨论。

第 7 个报文是 DupAck，为什么没有触发快速重传呢？

消息网关那头的应用日志应该不是 499，那会是什么样的日志呢？

欢迎你把今天的内容分享给更多的朋友，我们一起成长。

上一篇 春节特别放送（四）| 测一测你的网络排查能力

精选留言 (6)

bingo

2022-02-23

报文的时间有点乱，看得有点懵逼

作者回复: 我看看示意图如何优化一下，把时间线展示的更加清楚些~
就顺序来说，示意图里对每个报文表上了数字序号，跟抓包文件里的序号是一致的~

taochao_zs
2022-02-23

1 触发快速重传条件是再丢失报文段之后有3次以上的新报文段，4号报文段后只有6号报文段未达到重传条件。
2 消息网关应该是出现timeout之类的消息日志（由于丢包+网关超时设置）。

作者回复: 嗯 第一个问题回答正确，因为没有凑满3次DupAck，所以不会触发快速重传。
第二个问题，有两种可能性，一种就是你说的timeout，一种可能直接是原始的HTTP 400。具体记录了哪一种，要看客户端（消息网关）写日志的逻辑了：）


那时刻
2022-02-23

问题一，应该是需要客户端收到三个dupack，才能触发快速重传
问题二，消息网管那里因为5秒超时，状态码可能是504 Gateway timeout

作者回复: 问题一解答正确：）
问题二，那边虽然叫"网关"，但本身就属于独立业务端，并不是消费者app的http proxy，所以从语义上看，消息网关那里用504来结束不合适，我认为比较有可能的是两种：
1. 就是原始的HTTP 400
2. 记录应用层超时timeout（但不是504）


那时刻
2022-02-23

请问老师客户端在报文6发送了fin，而服务器在报文8收到http body，这个报文8是由于客户端内核超时重传报文导致的吧？
客户端对于服务器报文10的fin，回复rst，是不是因为报文10的seq number不合法导致的呢？


Realm
2022-02-23

试着回答下

1）7号报文之上的6号报文，是网关发出的，带有fin标志位，按老师之前分享的，发出fin

后，表示我可接受数据，即使收到服务端重传的请求，但不会发数据了，所以网关不会重传；...

展开

作者回复: 嗯可能答案比你想的要简单些：）只是因为只有一个DupAck，没有凑满3个，所以不会触发快速重传~

网关（作为客户端）应该是收到了HTTP 400错误日志，你这个回答很好。我在另外一个回复里也提了另一种可能，就是记录应用层自己的超时报错，但这属于"二次包装"。如果答案只能选一个，我也会选HTTP 400，因为这就是HTTP原始报文里的信息，最为真实。

潘政宇

2022-02-23

第4个报文，发送http header，为什么不标注为http请求，而是psh ack呢

作者回复: 示例文件已经上传到https://gitee.com/steelvictor/network-analysis/tree/master/15

报文4没有被标注为HTTP（确切说是标注为POST）的原因是，这个报文并不是完整的HTTP请求，只是前一半（headers）。

标记为PSH的话，是因为这个报文就是当时客户端（微信网关）的发送队列里最后一个报文，内核会对此加上PSH标志位，这个行为不是应用程序控制的。