谁动了我的网络

作为中国网民，我们享有学习网络知识的天然优势，这是很多老外一辈子都不敢奢望的。还记得刚学会上网的时候，某知名搜索网站突然就连不上了，有位学长说这是域名被封，直接连 IP 就可以了，还帮我修改了 hosts 文件。于是我沿着这个方向研究，很快就理解了 DNS 协议。在实践中学到的本领，比捧着课本背诵的不知道高到哪里去。

又过了一阵，竟然连 IP 都连不上了。我在探索过程中，又学会了 HTTP 代理和 VPN 等科学上网技术。就这样，十几年下来身经百战，不知不觉中掌握了很多网络技术，每天都能到外网和同行们谈笑风生。现在回忆起来，我的知识真没多少是刻意去学的，而是在和网络问题斗争时被动学会的。被虐久了还得了斯德哥尔摩综合症，去年到国外出差了一个月，便觉得食不知味，因为根本找不到学习的机会。回到国内赶紧打开浏览器，Duang~~立即弹出运营商推送的广告。还是那个熟悉的味道，回家的温馨顿时涌上心头。

本文要讲述的也是一个颇有中国特色的网络技术，其实很多人都遇到过，但没有去深究。最早向我反馈的是一位细心的网友，他在打开 www.17g.com 这个游戏网站时，有一定概率会加载出其他网站的游戏，比如 xunlei 的。他觉得很好奇，便采取了一些措施来排查。

1. 一开始怀疑是电脑中毒，于是在同网络下的其他电脑上测试，症状还是一样。

2. 其他地区的网友（包括我）打开这个网站时没有发现相同问题。

3. 他怀疑是当地运营商（哪家运营商我就不说了）搞的鬼，于是换了个宽带，果然就没问题了。

这位网友很生气，想知道运营商究竟对他的网络做了什么手脚，所以抓了个出问题时的包来找我。**我刚开始以为很简单，肯定是运营商的 DNS 劫持，即故意在收到 DNS 查询时回应一个假的 IP 地址，从而导致客户端加载错误的广告页面。**于是我打开网络包，用 dns 作了过滤，发现 www.17g.com 被解析到了 IP 地址 123.125.29.243（它还有一个别名叫 w3.dpool.sina.com.cn，见图 1）。可是经过进一步测试验证，发现这个 IP 地址竟然是对的，并没有被劫持。

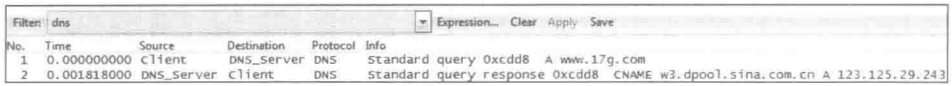

图 1

既然不是 DNS 劫持，那又是什么原因导致的呢？可惜这位网友抓包的时候电脑上开了太多应用，所以干扰包很多，无法采用暴力方式来分析（就是指不过滤，用肉眼把所有包都一一看过的分析方式）。如果用“ip.addr eq 123.125.29.243”过滤则得到图 2 的结果，似乎平淡无奇，只是显示有些包乱序了，但不知道这意味着什么。后来我才知道这就是线索之一，具体原因后面会讲到。

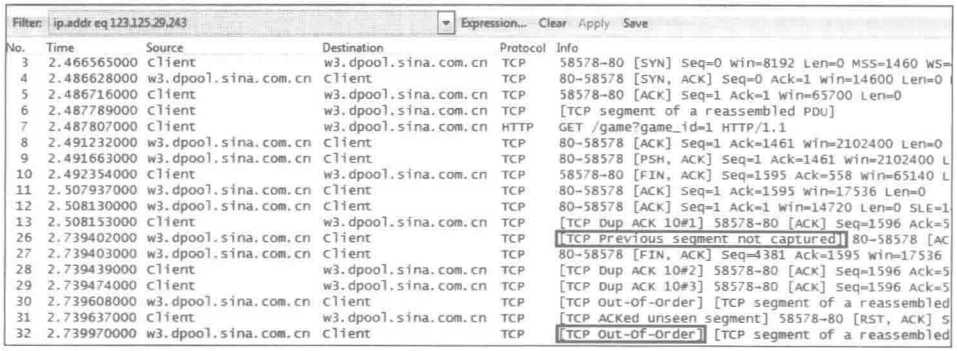

图 2

由于这是我第一次分析劫持包，所以不得要领，当晚分析到凌晨都没有弄明白。第二天早上只好到技术群求援了，一位在运营商工作的朋友给我科普了 HTTP 劫持的几种方式。其中有一种引起了我的注意，其大概工作方式如图 3 所示，实线箭头表示正常的网络包，虚线箭头表示运营商做的手脚。

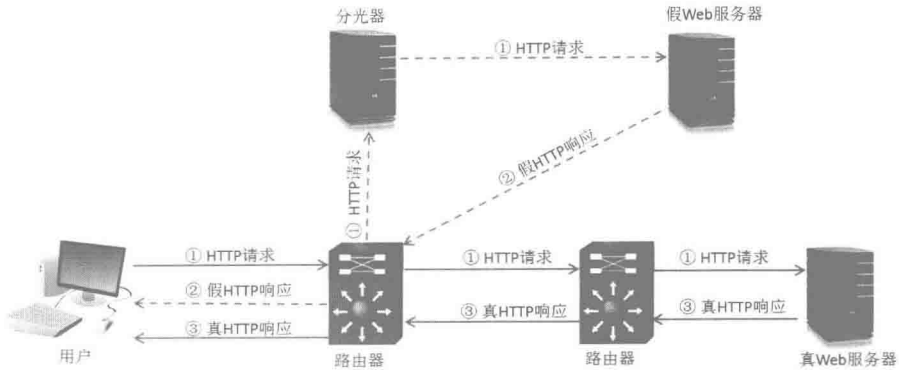

图 3

注意：这只是简单的示意图，不完全等同于真实过程。

在正常情况下，用户发出的 HTTP 请求（即图中的①）经过层层路由才能到达真实的 Web 服务器，然后真实的 HTTP 响应（即图中的③）又经过层层路由才能回到用户端。而**在做了手脚的网络中，运营商可以在路由器上复制 HTTP 请求，再交给假的 Web 服务器。然后赶在真实的 HTTP 响应之前，把假的 HTTP 响应（即图中的②）送达用户。**这个抢先应答会导致用户在收到真实的 HTTP 响应时，以为是无效包而丢弃。

根据这个工作原理，我们能否推测出假的 HTTP 响应有什么特征呢？如果能，那就能据此过滤出关键包了。我首先考虑到的是网络层的特征：**因为假 Web 服务器是抢先应答的，所以它发出的包到达用户时，TTL（Time to Live）可能和真实的包不一样。**那要怎么知道真实的 TTL 应该是多少呢？考虑到 3 次握手发生在 HTTP 劫持之前，所以我们可以假定参与 3 次握手的那台服务器是真的，从图 4 可见其 TTL 为 54。

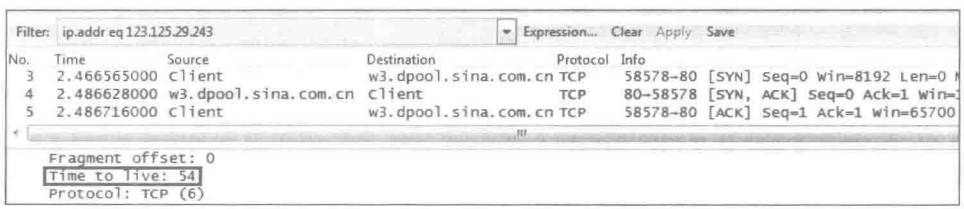

图 4

接下来就要动手过滤出假的包了。根据其源地址同样为 123.125.29.243，但

TTL 不等于 54 的特征，我用“(ip.srceq 123.125.29.243) && !(ip.ttl == 54)”过滤，得到图 5 的两个包，即 8 号包和 9 号包。看看右下角显示了什么信息？这不正是我要寻找的假页面“src=http://jump.niu.xunlei.com:8080/6zma2a”吗？

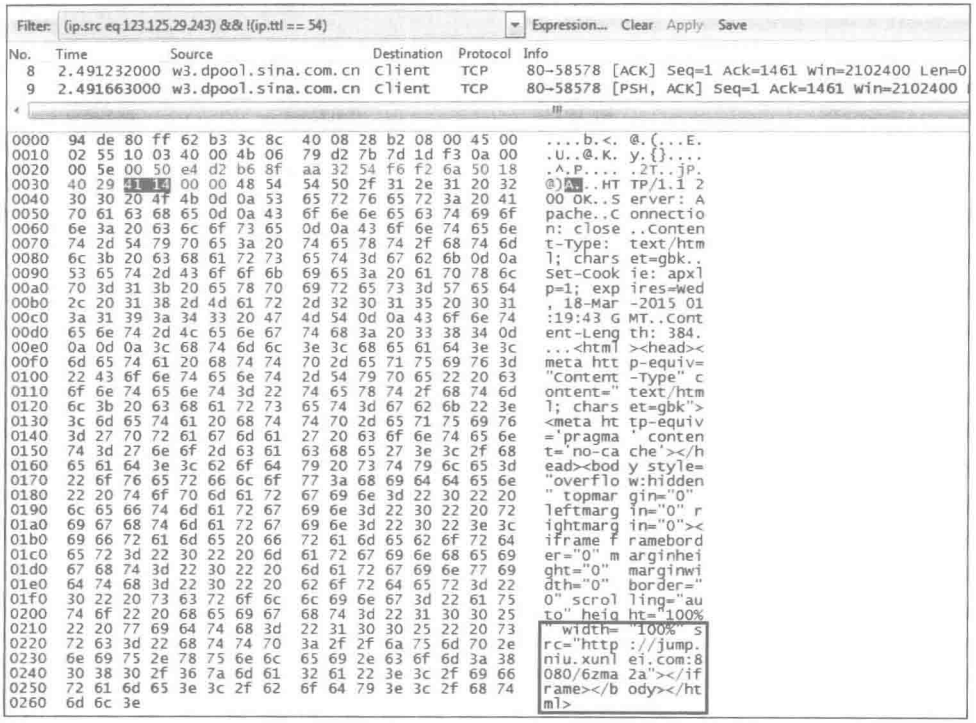

图 5

这个发现令我信心大增，有种拨云见日的感觉。再往下看几个包，果然发现了和 jump.niu.xunlei.com 的新连接。接着这个连接又把页面跳转到了“http://niu.xunlei.com/actives/welcome1426……”上（见图 6）。跳来跳去地非常难以追寻。

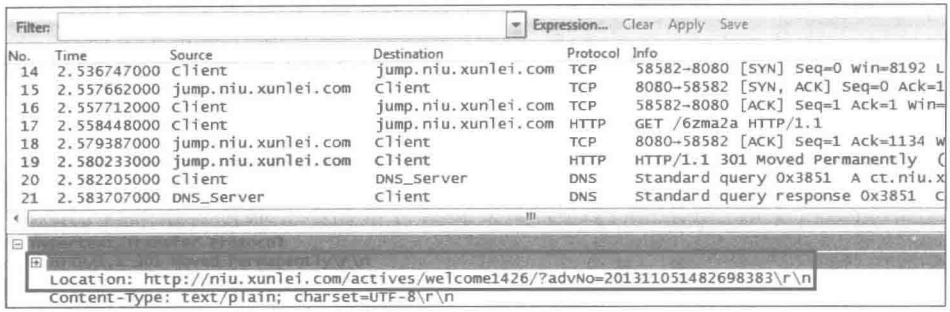

图 6

再后面的包就没必要分析了，以上证据已经足以向工信部投诉。据说投诉后运营商解决起问题来还挺爽快的，百度曾经上诉某运营商的劫持案件也获赔了。商场上的黑暗故事，就不在本书里展开讨论了，我们还是继续关注技术问题吧。

在这个案例中，万一真假网络包的 TTL 恰好一样，还有什么办法可以找出假的包吗？仔细想想还是有的。比如服务器每发送一个包，就会对其网络层的 Identification 作加 1 递增。由于 4 号包的 Identification 为 4078（见图 7），那它的下一个包，也就是 8 号包的 Identification 就大概是 4079 了（或者略大一些）。可是从图 8 可见，它的 Identification 一下子跳到了 55872，这也是一个被劫持的明显的特征。

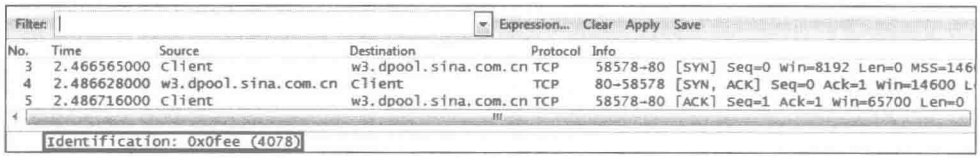

图 7

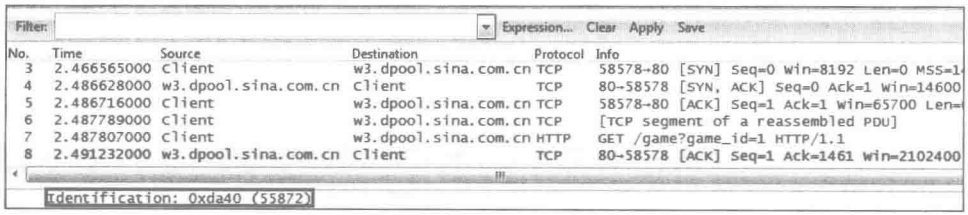

图 8

那万一运营商技术高超，把 TTL 和 Identification 都给对上号了，我们还有什么特征可以找吗？还是有的！刚刚介绍的两个特征都在网络层，接下来我们可以

到 TCP 层找找。在图 5 可以看到 8 号和 9 号这两个假冒的包都声明了“win=2102400”，表示服务器的接收窗口是 2102400 字节。对比一下其他网络包，你会发现这个数字大得出奇。为什么会这样呢？这是因为真正的 Web 服务器在和客户端建立 3 次握手时，约好了它所声明的接收窗口要乘以 128（见图 9）才是真正的窗口大小。假的那台服务器不知道这个约定，所以直接把真正的窗口值（win=16425）发出来，被这么一乘就变成了 16425×128=2102400 字节，大得夸张。

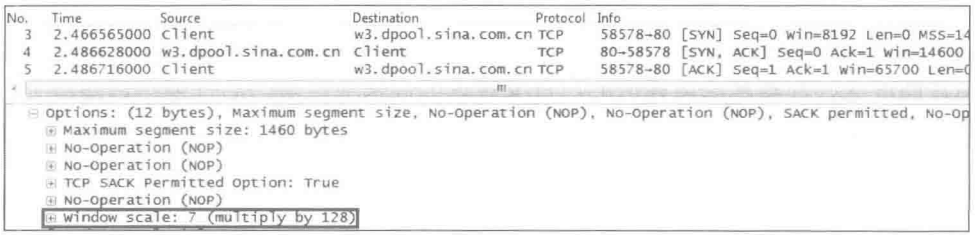

图 9

这个特征在本案例中非常明显，但不是每个 TCP 连接被劫持后都会表现出来的。假如 3 次握手时没有声明图 9 所示的 Window Scale 值，那就无此特征了。

其实我在一开始还提到了另一个现象，即图 2 中 Wireshark 提示的［TCP Previous segment not captured］和［TCP Out-of-Order］，意味着存在乱序。为什么会有这些提示呢？这是因为假服务器伪造的包抢先到达，增加了 Seq 号，因此等到真服务器发出的包到达时，Seq 号已经对不上了。Wireshark 还没有智能到能判断真假包的程度，只能根据 Seq 号的大小提示乱序了。

总而言之，在理解了劫持原理之后，我们便能推理出假包的特征，然后再根据这些特征过滤出关键包。但不是所有特征都能在每次劫持中体现出来的，比如接收窗口的大小就很可能是正常的，所以一定要逐层认真分析。这还只是众多劫持方式中的一种，如果采用了其他方式，那么在包里看到的现象又会有所不同。等我下次遇到了，再写一篇跟大家分享。

一个协议的进化

互联网行业日新月异，几年前估计连马云都预想不到今天的网络规模。从打车、订餐、抢购手机到付款理财，几乎无孔不入。与之不相称的是，互联网所依赖的基础协议——HTTP 却一直没有更新。知道现在最通用的 HTTP 1.1 是什么时候出现的吗？20 世纪末！那时候我还是林家庄跑得最快的少年，现在下个楼梯都能感觉肚子上的脂肪在跳跃。

那是什么使得 HTTP 1.1 青春永驻呢？是因为它的设计特别有前瞻性吗？可惜答案是否定的。当今网络的两个特征，导致 HTTP 1.1 已经成为性能瓶颈[①]：

- 现在网络的带宽比 20 世纪大得多，家庭带宽普遍在 10 兆以上，有些运营商甚至提供 200 兆的家庭套餐。
- 每个页面的内容远比 20 世纪的丰富，比如包含了更多小图片。类似 www.qq.com 这样还不算炫目的网站，光打开首页就能触发一百多个 GET 请求，但每个 GET 的数据量都不大。

这两个特征和 HTTP 1.1 有什么冲突呢？我们先从一个简单的例子开始说起。图 1 显示的是一个典型的网页打开过程，客户端只和服务器建立了一个 TCP 连接，然后从服务器上依次 GET 了三个资源，每个的数据量都很小，3 个包就能完成，即 1-3，4-6，7-9。

---

① 本文所说的性能，指的是浏览网页或者刷微博之类的小流量场景，不包括下载电影这样的大流量场景。

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	Client	Server	HTTP	GET /interface/getsub?callback=customOrder&para=%7B%22busi_id%2
3	0.078748000	Server	Client	HTTP	HTTP/1.1 200 OK (text/html)
4	0.107910000	Client	Server	HTTP	GET /qhome/uinterest?num=4&callback=contentInit&random=0.260833
6	0.151834000	Server	Client	HTTP	HTTP/1.1 200 OK (text/html)
7	0.179456000	Client	Server	HTTP	GET /newalgorithm/groupnews?callback=entCallback&channel=ent&ra
9	0.225205000	Server	Client	HTTP	HTTP/1.1 200 OK (text/html)

图 1

从这个包里面可以看出不少问题。

1. 客户端不是多个 GET 请求一起发出的，而是先发出一个请求，等收到响应之后才发出下一个请求。这样假如前一个操作发生了丢包，就会直接影响到后续的操作，成为“线头阻塞”（Head of Line［HOL］Blocking）。

2. 即使没有丢包，每个 GET 至少也要耗费一个 RTT（往返时间）。因此采用这种**非并发**的方式时，上百个 GET 所耗费的时间总量就非常可观。

3. 这种工作方式导致同时发出的包数太少，所以 TCP 窗口再大也派不上用场。这就相当于带宽被浪费了，家里办个 200M 带宽和 10M 带宽的上网体验差不多。想象一下六车道马路上总共跑着 3 辆车，你就能理解这种浪费了。

4. 还有一个副作用，就是包数太少会凑不起触发快速重传所必需的 3 个 Dup Ack，因此一丢包就只能等待超时重传，效率大打折扣。

图 1 演示的还只是明文传输的情况，如果要加密传输还会出现更严重的延迟。图 2 是一个 HTTP 1.1 加密传输过程，由于 HTTP 协议本身是明文传输的，所以用到了 TLS 来加密。

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	Client	Server	TCP	57422-443 [SYN] Seq=2414288222 Win=8192 Len=0 MSS=1460 WS=4 SACK_PER
2	0.088498000	Server	Client	TCP	443-57422 [SYN, ACK] Seq=2346460516 Ack=2414288223 Win=65535 Len=0 M
3	0.088617000	Client	Server	TCP	57422-443 [ACK] Seq=2414288223 Ack=2346460517 Win=66364 Len=0
4	0.091404000	Client	Server	TLSv1.2	Client Hello
5	0.178971000	Server	Client	TLSv1.2	Server Hello
6	0.179328000	Server	Client	TCP	443-57422 [ACK] Seq=2346460517 Ack=2414288447 Win=6592 Len=0
7	0.179360000	Client	Server	TCP	57422-443 [ACK] Seq=2414288447 Ack=2346461885 Win=64996 Len=0
8	0.179674000	Server	Client	TLSv1.2	Continuation Data
9	0.179712000	Client	Server	TCP	57422-443 [ACK] Seq=2414288447 Ack=2346461885 Win=64996 Len=0 SLE=23
10	0.180481000	Server	Client	TLSv1.2	Continuation Data
11	0.180531000	Client	Server	TCP	57422-443 [ACK] Seq=2414288447 Ack=2346464621 Win=66364 Len=0
12	0.180678000	Server	Client	TLSv1.2	Continuation Data
13	0.180789000	Server	Client	TLSv1.2	Continuation Data
14	0.180822000	Client	Server	TCP	57422-443 [ACK] Seq=2414288447 Ack=2346466709 Win=66364 Len=0
15	0.183432000	Client	Server	TLSv1.2	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
16	0.272740000	Server	Client	TLSv1.2	Change Cipher Spec, Encrypted Handshake Message
17	0.473554000	Client	Server	TCP	57422-443 [ACK] Seq=2414288789 Ack=2346466784 Win=66288 Len=0
18	0.520483000	Server	Client	TLSv1.2	Change Cipher Spec, Encrypted Handshake Message
19	0.520554000	Client	Server	TCP	57422-443 [ACK] Seq=2414288789 Ack=2346466784 Win=66288 Len=0 SLE=23
20	0.644898000	Client	Server	TLSv1.2	Application Data
21	0.734305000	Server	Client	TLSv1.2	Application Data
22	0.734499000	Server	Client	TLSv1.2	Application Data

图 2

这个过程可以分解成下面三步。

1．前 3 个包是三次握手过程，完成时刻是 0.0886 秒。

2．接下来的 4～19 号包是 TLS 握手过程，完成时刻是 0.5206 秒。

3．20～22 号包是真正的 HTTP 传输过程，完成时刻是 0.7345 秒。

不难看出，真正传输有效数据的是第三步，而它所耗费的时间在整个连接中的比例却并不高，大多时间是被前两步用掉了。最近有人在倡导所有网站都加密，恐怕没有意识到这样做会给网速带来多少影响。

既然单个连接不能并行发送 HTTP 请求，那能不能同时建立很多个连接呢？也不可以的，定义了 HTTP 1.1 的 RFC 2616 明确把最大连接数限制为 2 个，原文如下：

Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD NOT maintain more than 2 connections with any server.（使用长连接的客户端应当限制和某一台服务器的同时连接数。单用户客户端不能和任意一台服务器同时保持两个以上的连接。）

综合以上分析，我们可以得到一个结论，即 HTTP **协议所导致的网络延迟才是影响上网体验的主要因素，而不是带宽**。那有没有改进的办法呢？的确有一些

优化措施，比如大多数网站会让客户端与**多台服务器**建立并发的 TCP 连接。图 3 是我在打开国外某购物网站时的 HTTP 包，看上去似乎高效了很多，至少可以向多台服务器并行发送 GET 了。**不过这个方案也不完美，因为每个新建的 TCP 连接都会处于慢启动状态中，传输效率很低。而过了慢启动阶段，速度终于变快了，数据却已经传完了。**再说也不是每个网站都愿意承担多台服务器的成本。

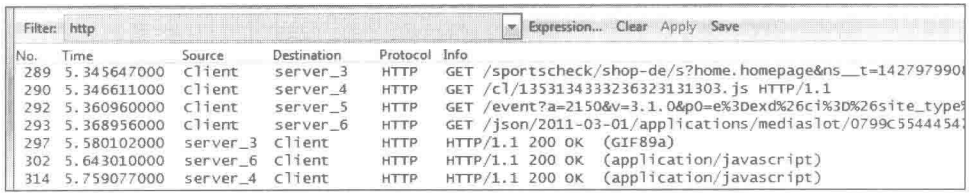

图 3

还有一个优化技术叫 Pipelining，可惜它也受到一些限制，比如代理服务器不支持等。那有没有办法可以彻底地解决这些问题呢？我脑洞大开地想象一下，也许符合以下需求的协议才可以：

- 它不需要三次握手和加密握手，能够节省多个往返时间；
- 它没有慢启动过程，所以不会一开始就传得很慢；
- 它能并行发送请求和响应，即支持“多路复用”（Multiplexing）。

也就是说，当前的一个 HTTP 连接和理想中的差距大概如图 4 所示。同样是 3 个操作，理想中的模型处理起来会快得多。

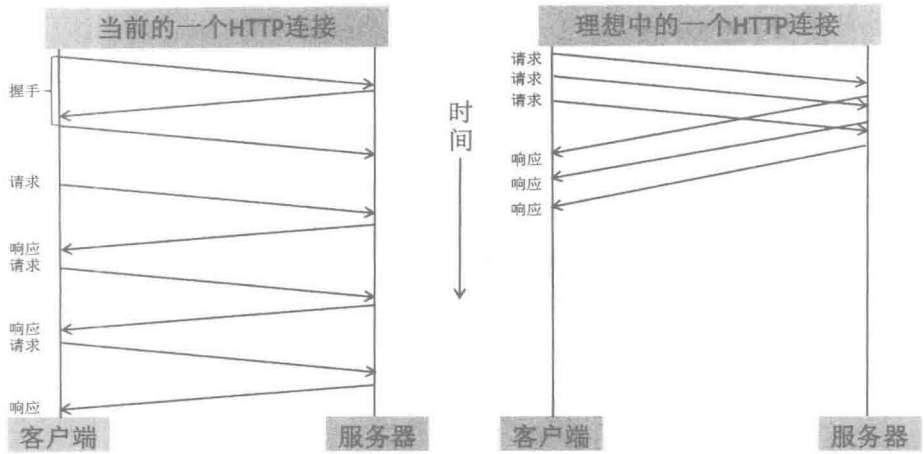

图 4

要完美实现这些需求，恐怕现有的 HTTP 和 TCP 机制都要被抛弃，得重新设计一套全新的协议才行。这在技术上也不是没有可能，说不定 Google 和 Microsoft 之类的公司就有这样的实力，但是在商业上完全不可行——在当前如此庞大的网络规模面前，谁也没有实力去推动所有网站、运营商和客户端做出改变。不要说 TCP 了，就连 HTTP 层的升级都会遇到不少阻力，因此只能采用向下兼容、逐步改进的办法。

近几年就有一些业内先锋尝试了不同的解决方案。其中最出色的是 Google 推出的 SPDY 协议，它只是在 HTTP 和 TCP 之间增加了一层，从而支持多路复用等功能（即在一个 TCP 连接里并行处理多个 HTTP 请求），很容易得到现有网站和客户端的支持。目前几乎所有主流浏览器都支持 SPDY，比如在 Chrome 上可以通过“chrome://flags”启用它，如图 5 底部所示。国外的主流网站也都支持 SPDY，比如 Facebook、Wordpress、YouTube 和 Twitter 等，可惜这些网站我们都没有条件测试。SPDY 的多路复用解决了本文开头提到的不少问题，比如带宽施展不开、丢包时难以触发快速重传等。经过几年的实验，SPDY 终于在 2015 年“进化”到 HTTP 2.0。

图 5

现在（2015 年 5 月份）HTTP 2.0 的 RFC 还没有出来。不过在其草稿中，已经明确表示“An HTTP/2.0 connection is an application level protocol running on top of a TCP connection”。只要它还是基于 TCP 的，就还有改进的空间，因为 TCP 三次握手和慢启动的负面影响仍然存在。怎么改进呢？如果你观察足够仔细，还会在图 5 中看到一个“实验性 QUIC 协议”，它或许会在以后实现真正的零延迟通信，而且是用在 HTTP 上。

QUIC 是 Quick UDP Internet Connections 的简称，旨在消除网页应用的延迟。由于它本质上是 UDP，所以不需要握手也没有慢启动过程，技术上的确有优势。目前只有 Google 的网站支持 QUIC，因为某些原因，中国技术人员还没有条件抓包来学习（用 VPN 也不行）。我委托一位印度同行抓了一个很简单的包，从图 6 的 Seq 号大致可以看到它是并发传输的。

No.	Time	Source	Destination	Protocol	Info
18	5.062888000	Client	Server	QUIC	CID: 11989733321912874687, Seq: 1
19	5.068569000	Client	Server	QUIC	CID: 11989733321912874687, Seq: 2
20	5.070567000	Server	Client	QUIC	CID: 11989733321912874687, Seq: 1
21	5.079090000	Client	Server	QUIC	CID: 11989733321912874687, Seq: 3
22	5.083972000	Server	Client	QUIC	CID: 11989733321912874687, Seq: 2
23	5.103229000	Server	Client	QUIC	CID: 11989733321912874687, Seq: 3
24	5.109300000	Client	Server	QUIC	CID: 11989733321912874687, Seq: 4
25	5.140611000	Server	Client	QUIC	CID: 11989733321912874687, Seq: 4
26	5.211370000	Client	Server	QUIC	CID: 11989733321912874687, Seq: 5
32	5.460549000	Client	Server	QUIC	CID: 11989733321912874687, Seq: 6
33	5.496031000	Server	Client	QUIC	CID: 11989733321912874687, Seq: 5
34	5.555704000	Server	Client	QUIC	CID: 11989733321912874687, Seq: 6

图 6

目前 QUIC 还没有流行开来，但 Google 已经发布了不少文档。说来有趣，我下载该文档时，发现其推荐语是“如果你需要一些材料来帮助睡眠，可以看看这

些文档。”让人哭笑不得。打开来的第一句话又是“我为这篇文章的长度而抱歉，如果我有足够多的时间，一定会把它写得短一点。”再次被作者逗乐了，浏览了一下发现篇幅果然很长。还是等 QUIC 哪天真正流行了，再单独为它写一篇吧。

假装产品经理

由于我最近经常评论手机 App 的设计细节，所以被一位新认识的网友问，“你是产品经理吧？连这个都知道。”

被误认为产品经理可不算好事，因为很多“程序猿”眼中的“产品狗”就是技术渣渣（虽然我不是这样认为的，各有所长嘛）。不过这一问倒是提醒了我，互联网行业的产品经理们也可以学学 Wireshark 的。如果需要研究对手的产品，用不着派间谍去偷文档，抓个包仔细分析就能得到不少信息，《寻找 HttpDNS》中提到的 IP 缓存便是极好的例子。如果只是想改进自己的产品，抓个包看看可能也有意外收获。就像 Windows 上自带的 FTP 客户端有个存在多年的 bug，测试时很难发现，但用 Wireshark 一打开就一目了然，详情可见我上一本书中的《一个古老的协议——FTP》。

今天我就假装一下产品经理，**用 Wireshark 分析一下微博 APP 是怎样上传和下载图片的，这对一个社交 App 来说至关重要**。实验过程很简单，启动抓包后执行以下步骤。

1．新建微博并选择一张 3.9MB 左右的图片，然后点“下一步”。

2．随便输入些字符后点击发送按钮。

3．点击这条已发微博的小图，从而打开大图。

4．在大图上点击“原图”，然后停止抓包。

每做完一个步骤都从电脑上 ping 一次手机的 IP 作为分隔标记，这是一个良好的习惯，有助于分析过程中区分每一步。接下来开始分析，先用“http||icmp”过滤一下抓到的网络包，实验过程的每个步骤就都显示出来了。从图 1 可见，四

次 ping 的标记都赫然在目（Protocol 栏显示为 ICMP），因此很容易判断哪些包对应着哪个步骤。我把上传和下载图片相关的 HTTP 请求都用方框标记出来，这样更加一目了然。

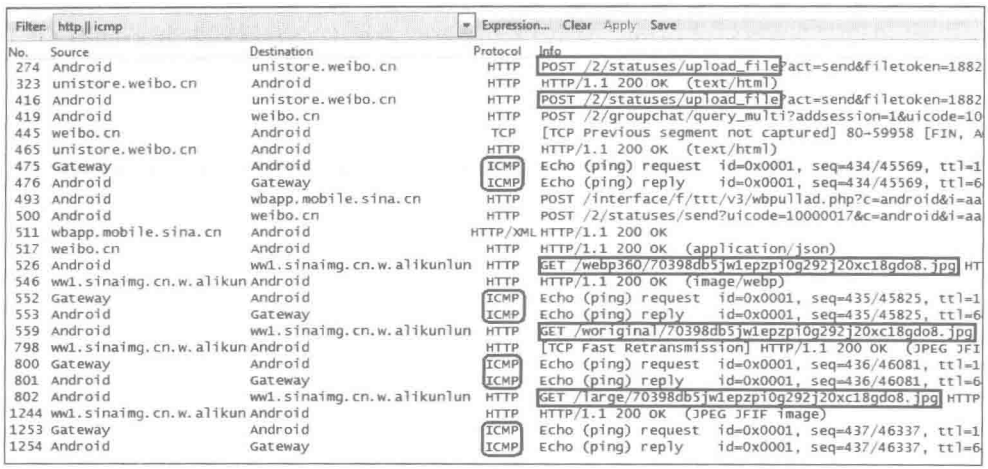

图 1

接下来再看看每一步都发生了什么。在第一次 ping 之前，我的操作是在微博上选择手机里一张 3.9MB 的图片，然后点击“下一步”。本以为这个操作只发生在手机本身，所以不会有网络流量产生。没想到微博 App 在这一步就已经上传图片了，从图 1 可见它用了两个 POST 来上传（274 和 416 两个包）。如果点开网络包的话，还可以从详情中看到总共传输了 320KB。这一步至少透露出微博的产品经理作了如下考量。

- **图片被选定之后就开始上传，而不是等到用户点击发送按钮之后。这样可以让用户感觉更流畅，好像点一下按钮就瞬间传完了。**当然提前上传也有负面作用：假如用户选定了多张图片并点击“下一步”，但是在发送前又改变主意了，于是点了“取消”，这样用户以为自己没有发过任何图片，但其实多张图片的流量都浪费了。

- 3.9MB 的图片只用了 320KB 的流量，**说明微博 APP 在上传图片之前会先大幅度压缩，**这就是为什么美女们好不容易 PS 完照片发出去，看到的效果却很糟糕。用网页版上传就不会压缩得这么严重，这也许是因为产品经理

考虑到手机用户是按流量计费的，而网页版用户一般都用包月宽带。

接着往下看。在第二次 ping 之前，我的操作是点击发送按钮，所以看到两个 POST（包号 493 和 500）是情理之中的。只有 526 号包“GET /webp360/70398db5jw1epzpi0g292j20xc18 gdo8.jpg”比较令人疑惑，为什么点发送的时候还会有 GET 图片的操作？其实这时已经发送完毕，开始下载小图并显示出来了。如果你观察足够仔细，会发现图片被上传到了 unistore.weibo.cn（见 274、416 等包），而下载时却是走 alikunlun（见 526 等包）。放 Google 一搜，原来阿里昆仑是阿里云 CDN 的内部名字。好吧，本来只是想分析一下产品设计，没想到连商业上的信息也不小心看到了，**新浪一定是把微博的 CDN 委托给阿里云了。**

插播一个读者疑问，为什么在这本书的截图中，Wireshark 会把 IP 地址显示成域名呢？其实只要勾上 View→Name Resolution→Enable for Network Layer 就行了，步骤如图 2 所示。

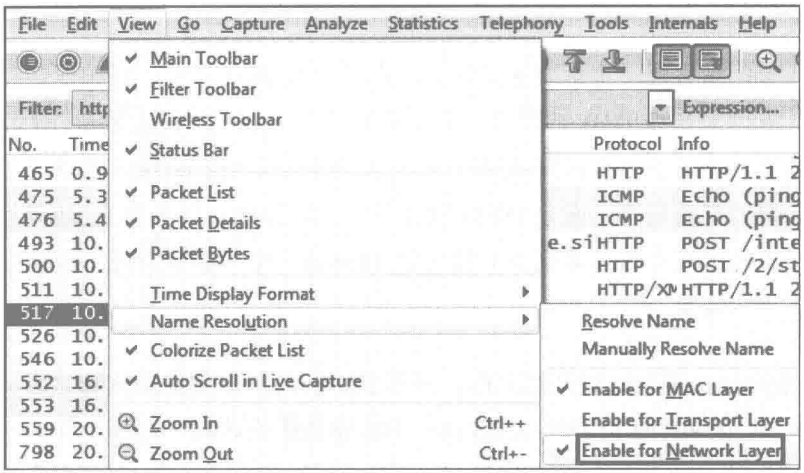

图 2

接下来再看看第三次 ping 之前的那个操作，即点开微博大图时的包。如图 3 所示，客户端通过 GET 下载了一张图片，“Content-Length: 155971”说明下载的所谓大图比上传时的还小，只剩下 156KB 左右了，又压缩掉一半。这 APP 真会给用户省流量。

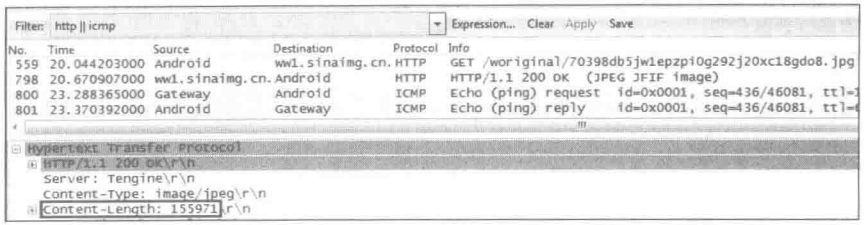

图 3

微博大图上还有个“原图”按钮，我一点击又产生了图 4 的流量，这次下载的图是 320KB 左右。可见微博认为的原图是 APP 上传前压缩过的那个，比起真正的原图（3.9 MB）还是小很多。

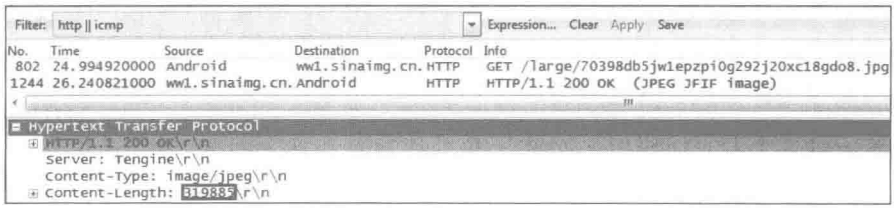

图 4

在图片处理这一点上，本山寨产品狗只能看出这么多了。正牌的产品经理如果有心去钻研，相信还能找出更多来。要是想知道微博的其他底层细节，比如负载均衡或者文本加密等，也完全可以设计一些实验，然后抓包来研究。我个人的下一个研究对象则是某款流行的手机游戏，相信和社交应用会大有不同。

自学的窍门

有家公司找我分析了几个网络包，事后很感激地说，“林工在网络行业做了很多年吧？”我只好如实相告，“其实我是存储行业的，看包只是业余爱好。”这回答听上去像是老林爱吹牛的毛病又犯了，但的确是实话。我身边还有很多“不务正业”的朋友，比如读化学出身的冬瓜头，年纪轻轻便写了本书叫《大话存储》，把存储技术的方方面面都覆盖到位了，而我这个在存储行业摸爬滚打了十来年的老人却只懂文件系统和卷这两层。另一位朋友 @馒头家的花卷 也是如此，这几年翻译了好多本 IT 方面的书，从操作系统到密码学都有涉及，更神奇的是他还是果壳科普达人，还有个三产是做音响服务的。可见**在这个信息爆炸的时代，很多行业的门槛已经被网络填平了，有志者皆可跨界入门，经过努力甚至能达到专业水平**。本文要分享的，就是我的一些自学窍门。

第一步，从浏览权威的百科网站开始。

当我们下定决心学习某项技术时，到维基百科阅读相关词条是极好的开始。几乎所有的技术都可以在上面找到，如果真的找不到，就要考虑如此冷门的东西是否值得投入时间学习了。大多数词条里讲到的概念都能链接到相应的新词条，比如 TCP 词条里会说到 handshake 这个概念，想多了解它就可以点进去看看。用这种方式认真地阅读完一个词条，实际上已经把相关的概念也弄懂了，相当于读完一本简略的入门书。不光技术方面，历史、政治等学科也可以用这个方式来入门，因为**词条之间的关联性非常有助于形成初步的知识体系，而不是没有关联的孤立知识点**。每次使用维基百科，我都会不知不觉地打开很多相关页面。比如本来只是想了解一下曹操的生平，一不小心就把曹操的子孙、对手和谋臣的词条也读了，一下子觉得人物关系清楚了很多。

百科网站那么多，我为什么推荐维基而不是其他？这是因为它比较权威而且全面，引用和注释也很规范。对一个初学者来说，信息的准确性是最重要的，否

则误解了一个入门知识点就可能毁了学习热情。维基百科唯一的不足是中文词条的数量和质量都远不如英文的，不过也不用担心，都是很好懂的 Plain English。技术研究到一定深度都是要读英文资料的，连中国学者写的顶级论文也是用英文的，我们何不从入门时就开始适应呢？

第二步，善用搜索引擎。

如果你求知若渴，一定不会满足于百科网站，因为脑子里产生的无数疑问会驱使你四处寻找答案。这时候身边有个大牛来指点是最好的，但是大牛回答三个以上的小白问题就会失去耐心，除非他一直在暗恋你。怎么办呢？自己搜索呗。几乎所有技术问题的答案都在网上，就算没有正面答案也会有侧面的，就看你的搜索技能了。我个人的技巧有以下三点。

- 技术方面的搜索要用 Google，因为它返回的头几条结果往往就是我想要的。一个典型的例子就是在 Google 和某国内著名网站搜“三点透视”，出来的结果完全属于两个不同的领域。假如你的研究已经到了领域尖端，需要读学术论文，那 Google 的优势就更加明显了。

- 把关键词翻译成英文再搜。世界上的技术高手很多，其中一些人也乐意回答网友的提问，而这些回答大多是用英文的。这就导致了英文资料比其他语种的资料丰富得多，假如你只用中文搜索就会错过这些答案了。不要怕英语不够用，开头也许是有点难，但是慢慢就能适应了。以我为例，至今美国同事讲的笑话我还是完全不知道笑点在哪，英文算很弱吧？但是技术方面的交流则毫无障碍，因为英文的技术文档看得太多了。

- 不要忽视图片搜索的价值。网络技术讲解得好的文章，往往是有图片的，而不是纯文本。所以当网页搜索得不到满意的结果时，尝试图片搜索，然后再从喜欢的图片链接到原网页。我就用这个方法找到过不少优秀的技术博客。

第三步，啃一本大部头。

有些人买书很大方，比如网络教程就买了很多本相似的，看到快递员扛来的

一大叠书把自己都吓到了，完全符合叶公好龙的定义。其实没有必要买那么多，**大部头的买一本足矣，关键是要真的去读**。像我这种铁公鸡类型的就不会犯这种错误，买书的时候精挑细选，买来之后读不完还觉得亏了。我现在还很怀念当年啃网络书的时光，每天都觉得很赚。现在还有很多书是可以免费在线阅读的，比如《The TCP/IP Guide》，觉得对胃口的话再点击 Donate 按钮给作者付点钱表示感谢，我惊奇地发现**付钱之后会读得更加认真，付得越多效果越好**。

第四步，动手操作。

“纸上得来终觉浅，绝知此事要躬行。”陆放翁诚不我欺。只有自己动手操作过了，才能理解得深刻，甚至纠正阅读时产生的误解。比如你可能已经把教材上的 TCP 流控理论都背得滚瓜烂熟了，但是遇到网络性能问题还是会手足无措，完全应用不上书里学过的知识。这就需要在读书的同时辅以动手训练，如果你在 Wireshark 里看过了拥塞重传，看过了 TCP zero window，甚至动手解决了它，从此流控技术就会像游泳、骑车一样成为你的自带属性，经年不忘。

也许有人会问，我到哪里找网络包来训练呢？其实机会就在身边，比如妹子寝室的网络不好啦，下载小电影变慢啦，都是抓包分析的好机会。实在没有机会也可以自己创造。十八岁以后学钢琴已经太晚了（因为你妈已经打不过你），但学网络却正是时候，自己在家搭个网络实验室都没人管。用虚拟的网络设备练习路由器命令，或者在个人电脑上搭建 Windows Domain 等，都非常有用。

能做好以上几点，我觉得已经很不容易了，进步也应该会很快。还有几点是我已经意识到了但自己也没有做好的，也列出来分享一下。

- 不要收藏了文章而不去读它，那样是在浪费时间。很多人看到技术分享就说句 mark，但**实际上从来不会回头去读**（中枪了没？）。我最近采取的措施就是强迫自己不去收藏，改成当场读完，能记得多少比例都比纯收藏强。

- 多给新人做培训。在准备培训的过程中相当于把知识点梳理了一遍。为了确保内容无误，你可能还需要做实验验证，这也是很好的练习。最重要的是，**能把一个技术讲到新手能听懂，比起自己懂就高了一层境界**。有的时候觉得自己很懂了，但是想把它讲出来或者写出来却很别扭，那就说明不

是真的懂。也不要怕分享了之后被别人抢饭碗，实际上无论你讲得多精彩，大多数听众过几天就忘了。

- 兴趣主导。很多领域牛人都是完全由兴趣主导的，一心钻研自己喜欢的技术，连领导交代的工作都放在第二位。越痴迷，越专注，水平也就越高。
- 多参加技术圈的交流。有些极客很宅，拒绝任何社交，并以此为荣。我觉得这是把缺点当作优点了，其实技术交流是非常有利于进步的，很多同行也是相当有趣的人。比如我曾经被一个难题困住了好久，没想到跟淘宝技术保障的朋友一聊，他立即就指了一条明路。虽然他也不是业内的大人物，但是技术背景互补，合作起来相当高效。当然了，交友从来都不只是为了互助，聊得投机才是最重要的。

以上都是我的个人经验，不一定会适合你，但希望有些参考价值。