# 01 | 网络模型和工具：网络为什么要分层？

2022-01-12 杨胜辉

《网络排查案例课》 课程介绍 >

**讲述：杨胜辉**
时长 27:09 大小 24.88M

你好，我是胜辉。

今天是咱们的第一节正课，就像我在开篇词里介绍的，在预习篇这里，我们的目标是搞清楚网络分层的概念，还有初步学习抓包分析。所以接下来，我会先从一些基础的网络知识说起，为你重点讲解网络分层模型以及各层之间的区别和联系。

因为咱们是以案例实战为导向的课程，所以我除了会在网络的每一层，给你介绍相关的技术细节以外，还会带你认识相应的排查工具。学完这节课，哪怕你原本是网络方面的小白，你也可以在网络排查方面“一试身手”了，是不是有点期待了呢？好，让我们开始吧。

## 网络是七层、五层还是四层？

学习网络排查，可能首先要搞清楚的，就是网络的分层模型了。工作中，我们也时常会听到这些术语，比如三层交互机、七层规则等等。网络分层的概念，可谓深入人心。

可是你有没有想过，网络为什么要分层呢？难道是非分不可吗？回答这个问题之前，我们先做个有趣的假设：这会儿是在网络诞生的前夜，什么 IP 协议、TCP 协议都还不存在，而你是网络的缔造者，面临设计网络这个伟大的任务。面对这么好的机会，你会选择做怎样的设计呢？

你大体上有这么两种选择：

**应用程序包办一切。**程序把应用层的数据，按某种编码转化为二进制数据，然后程序去操控网卡，把二进制数据发送到网络上。这期间，通信的连接方式、传输的可靠性、速度和效率的保证等等，都需要这个程序去实现。然后下次开发另外一个应用的时候，就把上面这些活，再干一遍。

**应用程序、操作系统、网络设备等环节各自分工。**应用程序只负责实现应用层的业务逻辑，操作系统负责连接的建立、处理网络拥塞和丢包乱序、优化网络读写速度等等，然后把数据交给网卡，后者和交换机等设备做好联动，负责二进制数据在物理线路上的传送和接收。

那么显然，第一种大包大揽的方式，实现难度太大、耦合度太高，怎么看都是一个“反面典型”。所以，我们应该选择第二种，也就是分层的方式去实现。

你有没有发现，其实这个思路，跟编程的思想是类似的。在编程中，我们需要把一些逻辑抽象为函数或者对象，以实现更好的解耦和复用。在网络世界里也是如此，每一层干好自己的分内事，那么所有的层次配合起来工作的时候，就显得有条不紊了。

说到具体的分层模型，你应该会想到两种比较有名的方案。对，它们就是 **OSI 的七层模型**，和 **TCP/IP 的四层 / 五层模型**。这两种模型的最大区别，就是前者在传输层和应用层之间，还有会话层和表示层，而后者没有。

我们来看一下示意图：

| OSI模型 七层模型 | TCP/IP模型 五层模型 | TCP/IP模型 四层模型 |
| --- | --- | --- |
| 应用层 | 应用层 | 应用层 |
| 表示层 | / | / |
| 会话层 | / | / |
| 传输层 | 传输层 | 传输层 |
| 网络层 | 网络层 | 网络层 |
| 数据链路层 | 数据链路层 | 网络访问层 |
| 物理层 | 物理层 | |

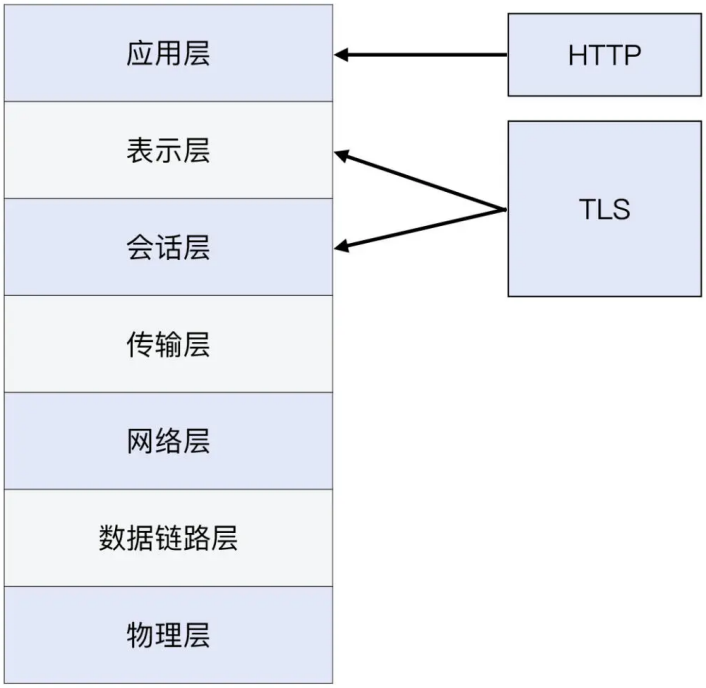


那在这里，你可能还会想：这两种模型哪种用得最多，或者说，哪种更合理呢？

其实我觉得倒不用过于纠结在“谁比谁更好”这个点上，如果我们理解了每一层的作用，那么就不会被表象上的层级所束缚了。事实上，两种分法都有可取之处。

一般来说，七层模型在我们工作当中谈论得更多些。比如，我的同事会找过来说“你帮我建一个七层规则吧”。这里的七层，就是指应用层，他说的“七层规则”呢，可能是 HTTP 路由规则，比如把符合某种条件的 HTTP 请求，分流到某个特定的后端集群。

还有一些场景，也是比较适合用七层模型来解释的。比如，TLS 虽然在 TCP 之上，按 TCP/IP 模型就要被归入应用层。但事实上，在 HTTPS 的场景下，HTTP 协议就是运行在 TLS 协议之上的，那么是不是把 HTTP 和 TLS 分到不同的层次更合适呢？正好在七层模型里，第五层和第六层，可以分别代表 TLS 的会话保持功能和数据加解密这种表示层的功能。






不过，会话层和表示层的协议确实比较少。从控制模型复杂度的角度来看，如果把这两层都合并到应用层，那么模型倒是比较简单，也适合入门学习的。所以从这一点上看，TCP/IP 模型也有可取之处。

这里你可能稍有疑问，为什么 TCP/IP 还有四层和五层模型这两种说法呢？其实五层模型就是 OSI 的前四层，加上一个应用层。这样的话，这个五层模型跟 OSI 七层模型，差异就比四层模型又缩小了一点。

所以，你现在应该明白了，**两种分层模型的最大差异，其实还是在会话层和表示层上面。**第一到第四层，已经基本统一了。而它们的最高层，虽然一个叫第七层，一个叫第四层或者第五层，表面上虽然并不一致，但实际上都可以用“应用层”来代替。这样既避免了可能的误解，也更加准确地表示了这一层的具体用途。

## 什么是 TCP 流？

在一些技术文档，特别是 Wireshark 相关的文档中，“TCP 流”是一个很常见的词汇。它是什么意思呢？为什么叫“流”，难道跟水有关吗？

其实，这里的 TCP 流，就是英文的 TCP Stream。Stream 这个词有“流”的意思，也有“连续的事件”这样一个含义，所以它是有前后、有顺序的，这也正对应了 TCP 的特性。

跟 Stream 相对的一个词是 Datagram，它是指没有前后关系的数据单元，比如 UDP 和 IP 都属于 Datagram。在 Linux 网络编程里面，TCP 对应的 socket 类型是 SOCK_STREAM，而 UDP 对应的，就是 SOCK_DGRAM 了。显然，DGRAM 就是 Datagram 的简写。

在具体的网络报文层面，一个 TCP 流，对应的就是一个五元组：**传输协议类型、源 IP、源端口、目的 IP、目的端口**。比如，今天你访问了极客时间网站，那么你这次的 TCP 流就可能就是这样一个五元组：

```
1 (TCP, your_ip, your_port, geekbang_ip, 443)
```

复制代码

一个 IP 报文，包含了所有这五个元素，所以 Wireshark 在解析抓包文件时，自然就能通过五元组知道每个报文所属的 TCP 流了。这也是为什么我们可以在 Wireshark 里，用 Follow TCP Stream 的方法，找到报文所在的 TCP 流。

不过有时候，也会有四元组的说法。其实它跟五元组大体上是一致的，只是四元组没有区分传输层协议类型（TCP 或者 UDP）。但是如果我们都清楚地知道应用类型，比如知道应用是 HTTP 协议的，那它的传输层协议默认就是 TCP，这一元是否算在里面，已经不重要了。

## 报文、帧、分组、段、数据包，这些术语是同一个东西吗？

**报文（packet）**，是一种相对宽泛和通用的说法，基本上每一层都可以用。比如，在应用层，你可以说“HTTP 报文”；在传输层，你可以说“TCP 报文”；同样的，在网络层，当然就是“IP 报文”了。事实上，网络层也是“报文”一词被使用最多的场景了。**数据包**也是类似的，可以在很多场景下通用。

我们再稍微考究一下语法。packet 这个词的后缀是 et。而在英文中，以 et 结尾的很多词表示某一个小小的东西。比如功能完备的一小段代码，叫 code snippet，一小段内嵌在 HTML 中的 Java 前端代码，叫 applet。自然的，packet 就是一个小的 pack（包裹）。

然而，另外几个术语在用的时候，就需要讲究一点了，因为它们并不是通用词，而是特定层的专有词汇。

**帧（frame）**是二层也就是数据链路层的概念，代表了二层报文，它包含帧头、载荷、帧尾。注意，帧是有尾部的，而其他像 IP、TCP、HTTP 等层级的报文，都没有尾部。我们不可以说“TCP 帧”或者“IP 帧”，虽然也许对方也明白你的意思，但我们都想做得专业一点，不是嘛。这里还有个小知识点：HTTP/2 实现了多路复用，其中也有帧的概念，不过那个帧，跟这里网络二层的帧，除了名称相同以外，就没有别的联系了。

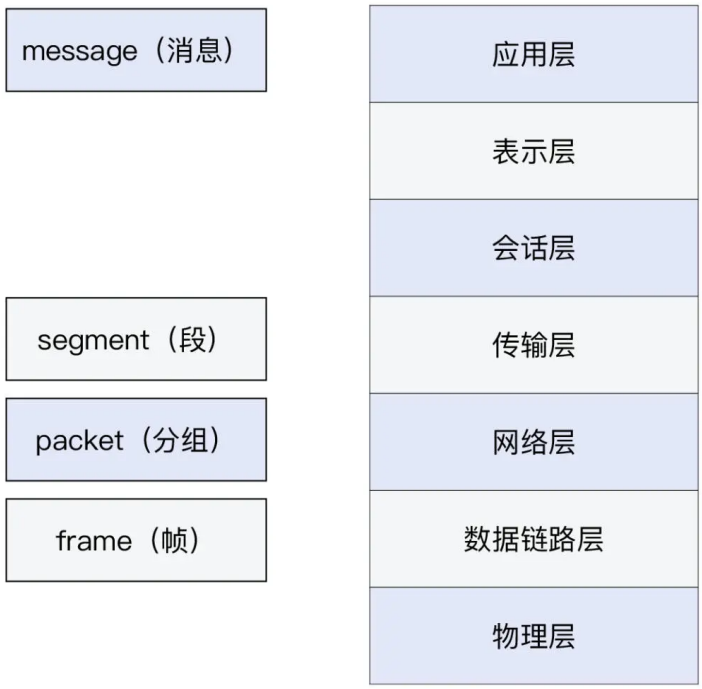
**分组**是 IP 层报文，也就是狭义的 packet。

**段特指 TCP segment**，也就是 TCP 报文。既然 segment 是“部分”的意思，那这个“整体”又是什么呢？它就是在应用层交付给传输层的消息（message）。当 message 被交付给传输层时，如果这个 message 的原始尺寸，超出了传输层数据单元的限制（比如超出了 TCP 的 MSS），它就会被划分为多个 segment。这个过程就是**分段**（segmentation），也是 TCP 层的一个很重要的职责。

说到 segmentation，你可能也会想到 fragmentation（分片）。这俩是同一个东西吗？这方面的知识点也不少，我在这里就不具体展开了。不过别着急，我会在第 8 讲里，帮你把这两个东西梳理清楚。

另外，这里还要提一下，Datagram 的中文叫**“数据报”**，但不是“数据包”。读音类似，但意思并不完全相同。前面说过，“数据包”是一个通用词，所以用“UDP 数据包”指代“UDP 数据报”并没有问题。但反过来，非 UDP 协议的数据包，比如 TCP 段，就不能叫“TCP 数据报”了，因为 TCP 不是 Datagram。

最后，你可以再来看下这张层级和术语对应关系的示意图：




极客时间

## 网络各层都有哪些排查工具呢？

通过上面的内容，你应该对于网络为什么要做分层、为什么那样做分层，已经有了比较清晰地认识了，我也带你探讨了每个层级的名词概念。所谓“名不正则言不顺”，咱们把这些术语搞清楚了，是不是感觉自己的技术“格调”也有那么点提升了呢？

接下来，我们进入干货部分，也就是每个层级的排查工具，用大白话说就是：“这可是我们吃饭的家伙儿”。

## 应用层

应用层的排查工具就太多了，相信做应用的同学，对自己的应用排查，应该是比我要更加熟悉。那我这里呢，就选一个主要的应用来展开吧，我们来谈谈 **HTTP 应用的排查工具**。

现在主流的浏览器是 Google 的 Chrome，它本身就**内置了一个开发者工具**。在 Chrome 界面里按下 F12，或者你是苹果系统的话，还可以按下组合键 option + command + I，启动开发者工具。

其实在其他的浏览器上，都有类似这样的工具，比如 **Firefox 和 Edge**。而且因为 Edge 基于 Chromium 浏览器内核，它的开发者工具跟 Chrome 的开发者工具很相似。

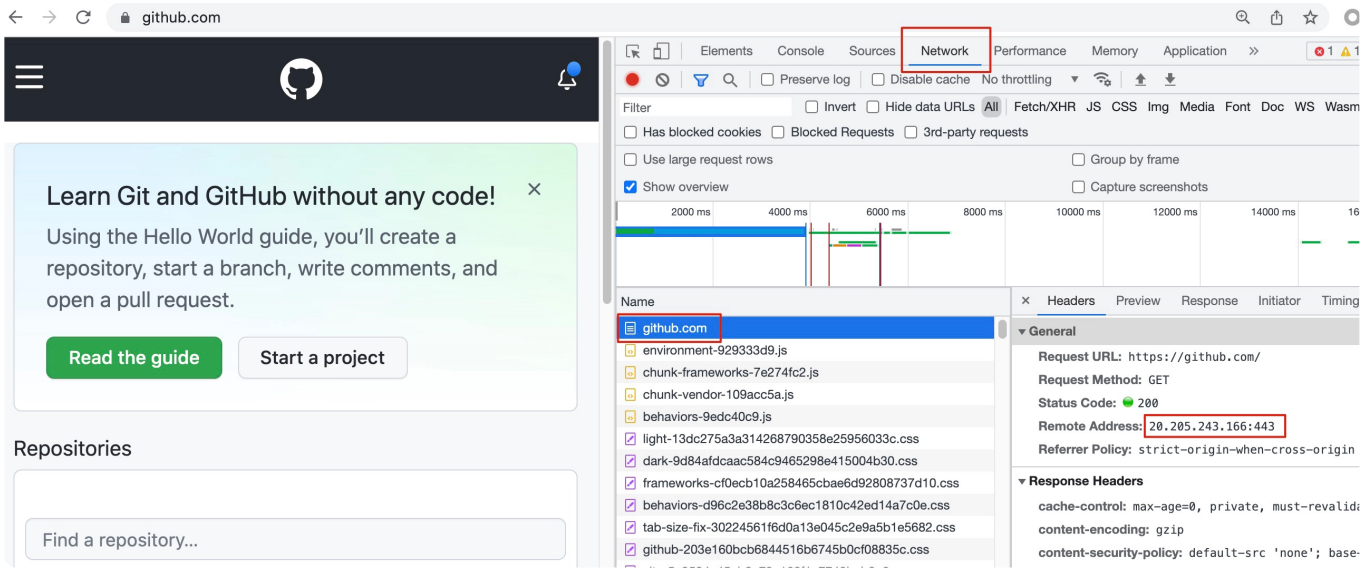
在更老的 IE 浏览器时代，并没有原生的开发者工具。当时有一个叫 **HttpWatch** 的工具，可以在 IE 上实现类似的功能，但需要另外安装。

借助开发者工具，我们可以非常方便地做很多事，比如以下这些。

### 找到有问题的服务端 IP

比如有用户报告死活访问不了你的网站，但是你很清楚这个网站的域名对应了很多 IP 地址，你怎么知道用户连的是哪个 IP 呢？

你可以这样做：让客户启用开发者工具，在 Network 页找到主页对象，在它的 Headers 部分，就能看到 Remote address，这里的 IP 就是当前连接的 IP，比如下面这样：

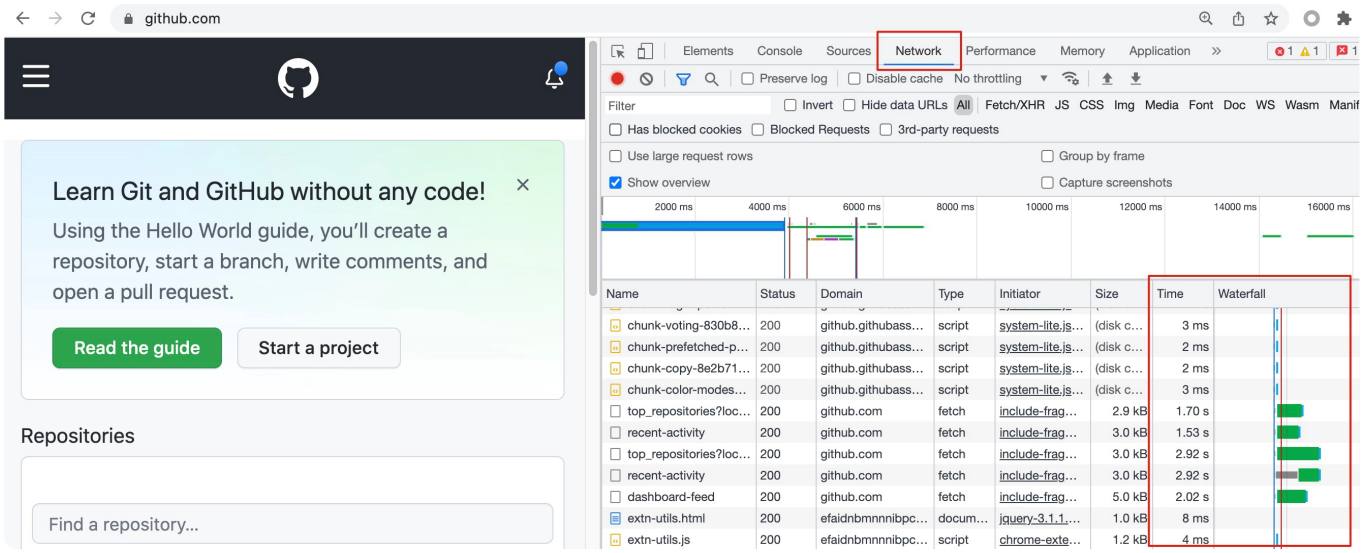



不过有句成语叫“刻舟求剑”，因为 DNS 解析的关系，你很可能下次重连就不是这个 IP 了，所以每次都应该重新确认一下这个信息。

这个技巧，在**排查公网的访问问题**的时候特别有用。要知道，现在流量大一点的网站都已经上了 CDN，那就必然在全国乃至全球各地，有少则数十个、多则数百个 CDN 终端节点，在给访问者提供就近的服务。如果有人说他访问不了某个站点了，那么请一定让他用开发者工具，找到他连的远程 IP，然后你再根据这个信息展开排查工作。

### 辅助排查网页慢的问题

访问页面感觉很慢，那么可以借助开发者工具的**时间统计功能**，找到耗时较高的 HTTP 资源对象，再针对性排查。比如我觉得访问 https://github.com 很慢，那么可以先打开开发者工具，然后访问站点，等全部加载完成后，到 Network 页查看这些 HTTP 对象的加载时间。




不过，这个办法只能排查到是哪个资源对象耗时比较长，但更进一步的排查，比如“为什么这个对象的加载时间比别的对象长”这个问题，开发者工具就难以回答了。关于这个问题，我会在后续的课程里深入展开，我们会用到抓包分析这把“手术刀”，来根本性地排查这类问题。

### 解决失效 Cookie 带来的问题

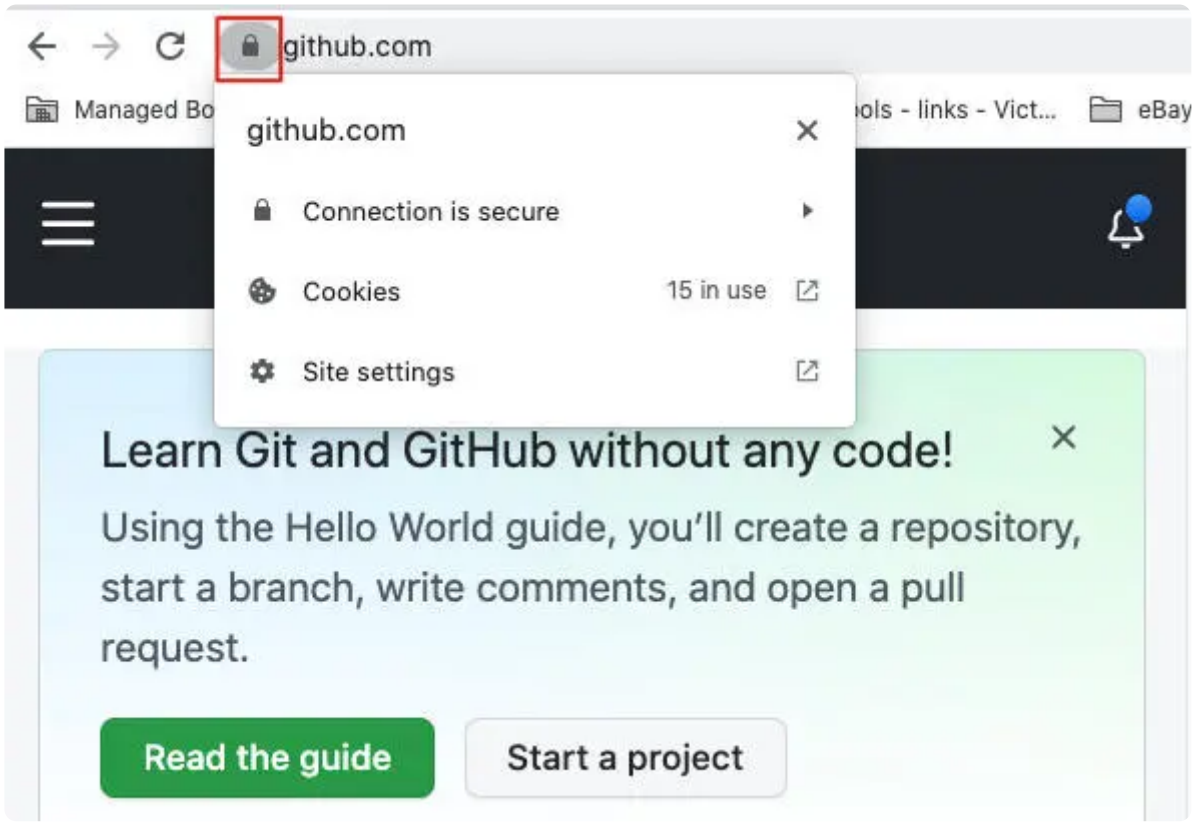
有时候我们的 Cookie 过期了，导致无法正常登录站点，那么可以打开开发者工具，到 Application 页，找到 Storage -> Cookie，把对应的条目清除。这样下次你再访问这个站点，就已经“洗心革面”了。对站点来说，你就是一次新的访问，可以生成一次新的 Cookie 了。

当然，你通过删除浏览器缓存的方式，也是可以做到这一点的。但开发者工具的优点是，可以**细粒度**到这个网站级别，而删除缓存的方式，删除的就是所有站点的 Cookie 了，这未必是你想要的。

## 表示层和会话层

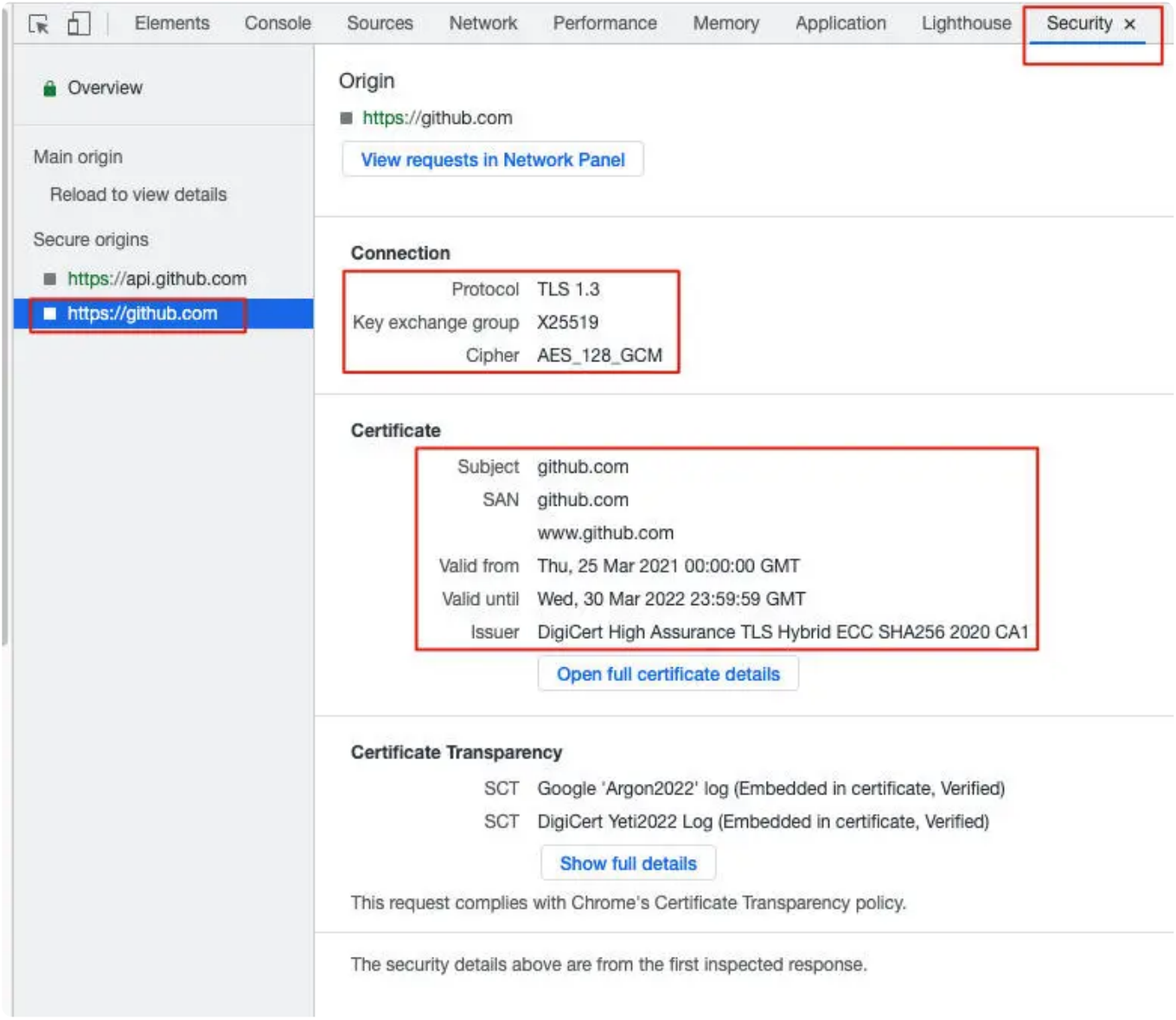
在前面的网络分层部分，我提到过，其实表示层和会话层的协议并不多，TLS 可以归入这两个层级。为了对 TLS 的问题进行排查，我推荐你两种工具。

**第一种，还是基于浏览器做初步的检查，主要是围绕证书本身做检查。**在浏览器的地址栏那里，有一个按钮，点开后就可以查看 TLS 证书等信息：



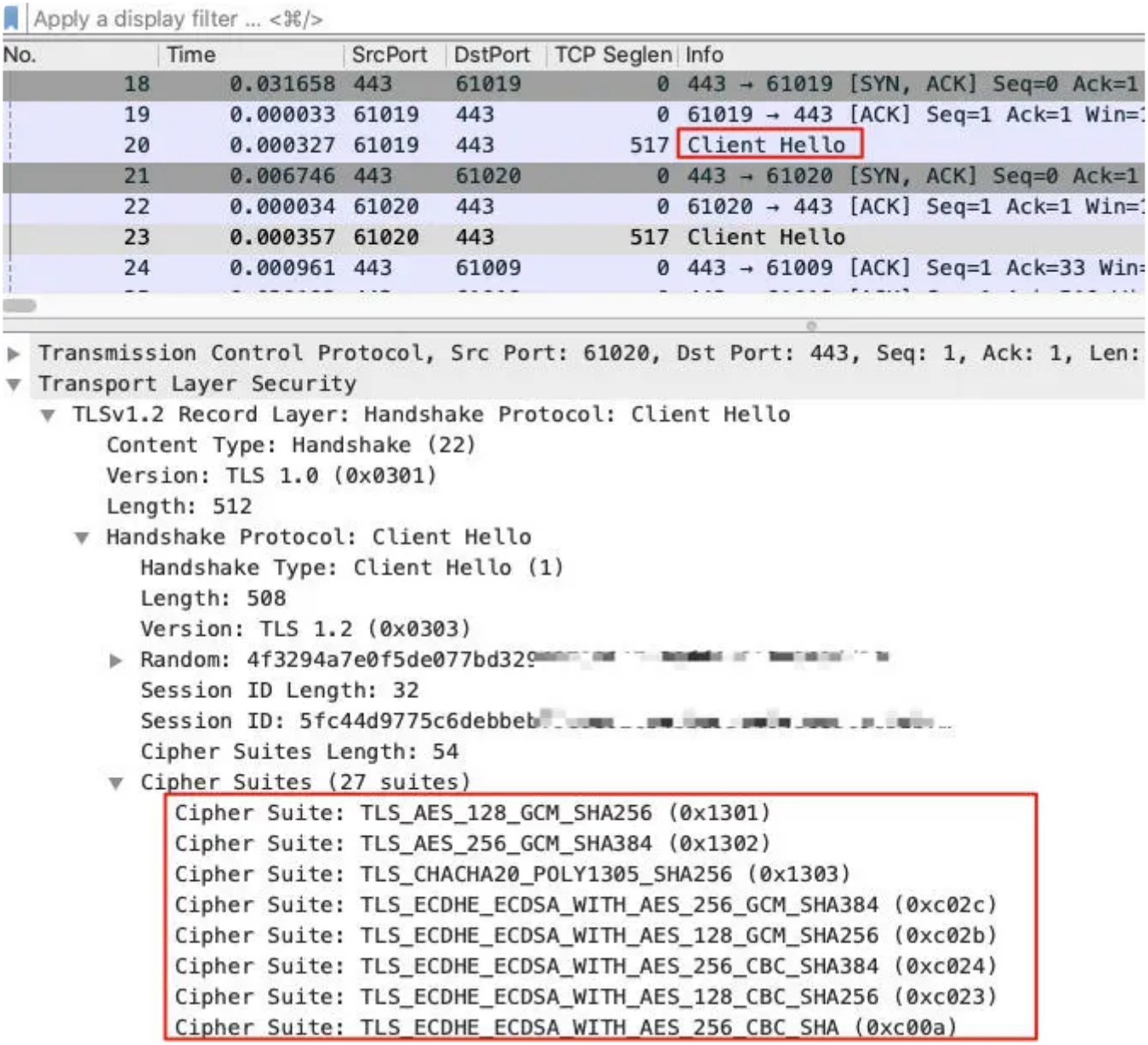

在上面的菜单中，继续点开 Connection is secure 按钮，进而点击 Certificate is valid 按钮，就能查看证书了。

另外，使用开发者工具的 Security 菜单，还可以查看更为详细的 TLS 信息，包括协议版本、密钥交换算法、证书有效期等等。




**第二种，关于 TLS 握手、密钥交换、密文传输等方面的排查，还是需要用 tcpdump 和 Wireshark 来做。**在 Wireshark 中，可以更加全面地查看 TLS 细节。

比如，我们可以直接看到 TLS 握手阶段里，双方协商**过程中**各自展示的 Cipher suite，而在开发者工具里，我们只能看到协商**完成后**的选择。




## 传输层

传输层毫无疑问是重中之重，工具也很多。我们就按排查场景来介绍工具。

### 路径可达性测试

如果我们要测试 TCP 握手，我们有 **telnet、nc** 这两个常规工具。比如 telnet：

复制代码

```
$ telnet www.baidu.com 443
Trying 180.101.49.12...
Connected to www.a.shifen.com.
Escape character is '^]'.

```

用 nc 呢，可以这样：

```
$ nc -w 2 -zv www.baidu.com 443
Connection to www.baidu.com 443 port [tcp/https] succeeded!
```

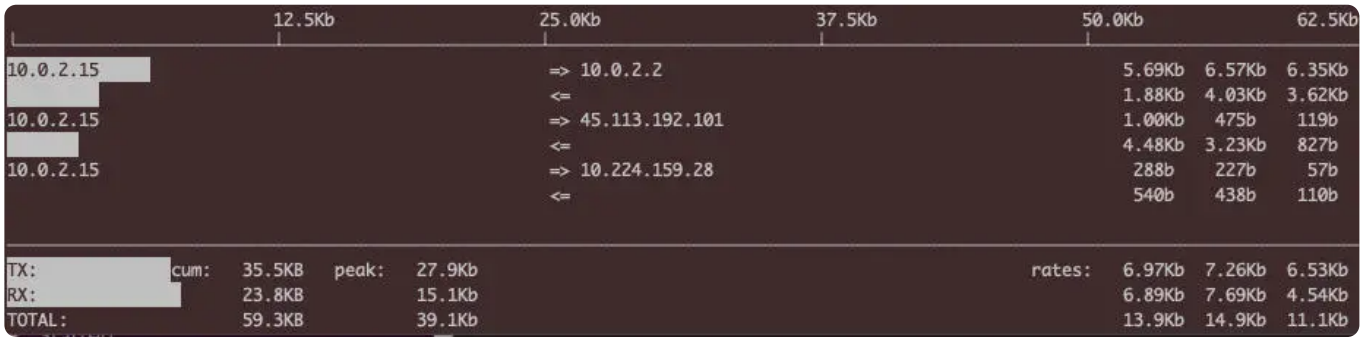
### 查看当前连接状况

**netstat** 命令是一个经典命令了，很多同学都会使用它来获取当前的 TCP、UDP 等的连接信息，比如：

```
$ netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0    280 10.0.2.15:22            10.0.2.2:56669          ESTABLISHE
tcp6       0      0 :::22                   :::*                    LISTEN
```

### 查看当前连接的传输速率

有时候，你的网络跑得挺繁忙的，但你却不知道哪个连接占用了大量的带宽？你可以用 **iftop**。这个工具不是系统默认自带的，需要你安装一下，然后执行 iftop 就好了。对了，你需要有 sudo 权限，也就是执行 sudo iftop，然后就能看到不同连接的传输速率，把祸害你带宽的连接给找到。比如下面这样：




### 查看丢包和乱序等的统计

其实，用 netstat 除了可以获取实时连接状况，还可以获取历史统计信息。比如，你怀疑一台机器的网络很不稳定，除了用 ping 做简单的测试，你还可以用 **netstat -s** 来获取更加详细的统计信息。比如，其中的 TCP 丢包和乱序计数值，就能帮助你判断传输层的状况。下面是我截取了一次 netstat -s 命令的输出：

复制代码

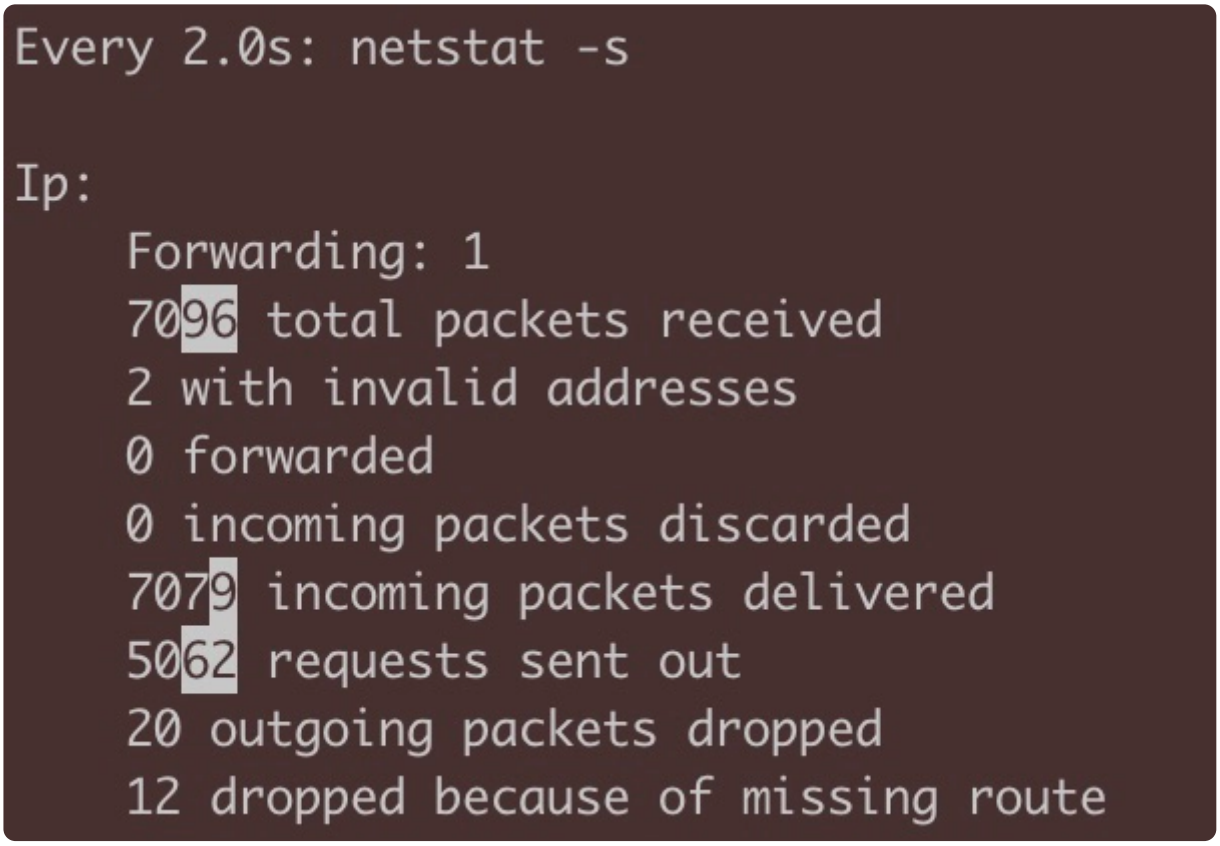
```
$ netstat -s
......
Tcp:
    16 active connection openings
    1 passive connection openings
    8 failed connection attempts
    1 connection resets received
    1 connections established
    6254 segments received
    4035 segments sent out
    1 segments retransmitted
    0 bad segments received
    3 resets sent
......
TcpExt:
    1 ICMP packets dropped because socket was locked
    3 TCP sockets finished time wait in fast timer
    8 delayed acks sent
    4674 packet headers predicted
    10 acknowledgments not containing data payload received
    1008 predicted acknowledgments
    TCPTimeouts: 1
    TCPBacklogCoalesce: 140
    1 connections reset due to early user close
    TCPRcvCoalesce: 2187
    TCPAutoCorking: 110
    TCPSynRetrans: 1
    TCPOrigDataSent: 1041
    TCPDelivered: 1049
```

你可能会问：这些不是静态值吗，我想知道当前情况啊？这个也很好解决，你可以这样做：

复制代码

```
watch --diff netstat -s
```

这个命令会把发生变化的数值进行高亮，方便我们查看：




当然，上面这个算运维“青铜”版。你也可以写一个简单的脚本，在两次 netstat -s 命令之间执行 sleep，然后计算两个读数之间的差值，并除以 sleep 的时间，得到大致的变化速度。这样就又升级了一点。

如果你想做得再到位一点，你可以把 netstat -s 的输出值写入到 TSDB，然后用 Grafana 之类的 Dashboard 展示，这样不仅有视图，也有历史值，可以算运维“王者”了。

### 还有 ss？

**ss** 命令是 Iproute2 包里的命令，也是 netstat 的“取代者”。它提供了对 socket 的丰富的统计信息。比如下面这条命令我也经常用，可以查看到当前连接的统计信息：

复制代码

```
$ ss -s
Total: 164
TCP:   5 (estab 1, closed 0, orphaned 0, timewait 0)

Transport Total     IP        IPv6
RAW    1         0         1
UDP    2         2         0
TCP    5         4         1
INET    8         6         2
FRAG    0         0         0
```

当然，也不能完全说“ss 等于 netstat”，因为事实上 netstat 命令的功能，被拆分到了 ss 和 ip 这两个命令里，并分别得到了丰富和加强。具体的细节，我们在课程中还会陆续提到。

## 网络层

在这一层，除了可以直接用 ping 这个非常简便的工具以外，你还应该掌握另外两个命令，它们能提供更为强大的排查能力，它们就是 **traceroute 和 mtr**。

### 查看网络路径状况

下面这个，是我用自己的 Mac 笔记本做一个简单的 traceroute 的典型输出：

复制代码

```
$ traceroute  www.baidu.com
traceroute to www.a.shifen.com (180.101.49.12), 64 hops max
  1   10.0.2.2  0.133ms  0.131ms  0.087ms
  2   192.168.1.1  3.048ms  1.466ms  1.574ms
  3   100.65.0.1  8.975ms  3.067ms  6.472ms
  4   61.152.53.149  5.644ms  3.691ms  4.624ms
  5   61.152.24.226  5.357ms  4.393ms  4.244ms
  6   202.97.29.122  10.171ms  10.403ms  8.755ms
  7   58.213.94.118  10.707ms  11.880ms  11.441ms
  8   58.213.94.90  9.644ms  *  *
  9   58.213.96.110  12.758ms  12.095ms  11.842ms
 10   *  *  *
 11   *  *  *
 12   *  *  *
 13   *  *  *
 14   *  *  *
 15   *  *  *
 16   *  *  *
 17   *  *  *
 18   *  *  *
 19   *  *  *
 20   *  *  *
```

哦，等等，为什么从第 10 跳开始就没有 IP，只有星号了？你是不是也遇到过这种情况呢？其实，你稍微改一下命令，也就是加上 **-I** 参数（I 代表 ICMP），就可以正常跑到底了：

复制代码

```
$ traceroute  www.baidu.com -I
traceroute to www.a.shifen.com (180.101.49.12), 64 hops max
  1   10.0.2.2  0.099ms  2.363ms  0.078ms
  2   192.168.1.1  3.320ms  1.220ms  1.204ms
  3   100.65.0.1  8.737ms  4.872ms  6.403ms
  4   61.152.54.125  5.035ms  3.397ms  4.288ms
  5   *  61.152.25.110  4.176ms  *
  6   202.97.101.30  7.447ms  6.399ms  5.936ms
  7   58.213.95.110  10.488ms  *  9.014ms
  8   *  58.213.95.134  11.064ms  *
  9   58.213.96.74  10.997ms  10.042ms  10.592ms
 10   *  *  *
 11   *  *  *
 12   *  *  *
 13   180.101.49.12  11.269ms  9.518ms  8.779ms
```

背后的原理，就是 traceroute 默认是用 UDP 作为探测协议的，但是很多网络设备并不会对 UDP 作出回应。所以我们改成 ICMP 协议做探测后，网络设备就有回应了。其实，Windows 上的 tracert，就是默认用 ICMP，这一点跟 Linux 正好是反过来的。两个操作系统，真是“相爱相杀”啊。

但是，traceroute 也有一个明显的不足：**它不能对这个路径做连续多次的探测。**

于是，mtr 出现了，它可以说是 traceroute 的超集，除了 traceroute 的功能，还能实现丰富的探测报告。尤其是它对每一跳的丢包率的百分比，是用来定位路径中节点问题的重要指标。所以，当你在遇到**“连接状况时好时坏的问题”**的时候，单纯用一次性的 traceroute 恐怕难以看清楚，那就可以用 mtr，来获取更加全面和动态的链路状态信息了。

复制代码

```
$ mtr www.baidu.com -r -c 10
Start: 2022-01-07T04:05:02+0000
HOST: victorebpf                  Loss%   Snt   Last   Avg  Best  Wrst StDev
  1.|-- _gateway                   0.0%    10    0.3   0.4   0.2   1.2   0.3
  2.|-- 192.168.1.1                0.0%    10    1.6   1.8   1.4   3.2   0.5
  3.|-- 100.65.0.1                 0.0%    10    3.8   7.0   3.8  10.3   2.0
  4.|-- 61.152.54.125              0.0%    10    4.0   4.3   3.6   5.1   0.5
  5.|-- 61.152.25.110             30.0%    10    5.0   6.8   4.4  18.9   5.4
  6.|-- 202.97.101.30             20.0%    10    7.8   6.6   5.4   7.8   0.8
  7.|-- 58.213.95.110             80.0%    10   10.0   9.8   9.6  10.0   0.3
  8.|-- ???                      100.0     10    0.0   0.0   0.0   0.0   0.0
```

```
 9.|-- 58.213.96.74              0.0%    10   10.5  12.7   9.9  24.7   4.9
10.|-- ???                      100.0    10    0.0   0.0   0.0   0.0   0.0
11.|-- ???                      100.0    10    0.0   0.0   0.0   0.0   0.0
12.|-- ???                      100.0    10    0.0   0.0   0.0   0.0   0.0
13.|-- 180.101.49.12            0.0%    10    9.4   9.1   8.3   9.7   0.5
```

### 查看路由

命令 **route** 可以查看路由表，不过这个命令比较老一点：

复制代码

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.2        0.0.0.0         UG    100    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 enp0s3
10.0.2.2        0.0.0.0         255.255.255.255 UH    100    0        0 enp0s3
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker
```

传输层工具里介绍的 **netstat**，其实也能帮我们查看路由，只要加上 **-r** 参数：

复制代码

```
$ netstat -r
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
default         _gateway        0.0.0.0         UG        0 0          0 enp0s
10.0.2.0        0.0.0.0         255.255.255.0   U         0 0          0 enp0s
_gateway        0.0.0.0         255.255.255.255 UH        0 0          0 enp0s
172.17.0.0      0.0.0.0         255.255.0.0     U         0 0          0 docke
```

我前面说过，netstat 是被 ss 和 ip 这两个命令替代了。所以我们同样可以用 **ip 命令**查看路由。比如这样：

复制代码

```
$ ip route
default via 10.0.2.2 dev enp0s3 proto dhcp src 10.0.2.15 metric 100
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.15
10.0.2.2 dev enp0s3 proto dhcp scope link src 10.0.2.15 metric 100
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
```

## 数据链路层和物理层

这一层离应用层已经很远了，一般来说是专职的网络团队在负责。如果这一层有问题，就会直接体现在网络层表现上面，比如 IP 会有丢包和延迟等现象，然后会引发传输层异常（如丢包、乱序、重传等）。所以，**一个稳定的数据链路层乃至物理层，是网络可靠性的基石。**

你可能会奇怪：既然底下这两层的稳定性如此重要，那上层的 TCP 不是号称还有传输可靠性的保障吗？难道这种保障形同虚设？

其实，这两点并不矛盾。TCP 的传输可靠性是通过序列号、确认号、重传机制等来保证的，通过这种机制，TCP 可以在**一定程度**的网络不稳定场景下，依然保证传输可靠，但不等于 TCP 可以无限容忍底层的不稳定，因为各种 TCP 拥塞控制算法都会因为这种问题，而极大地降低传输性能。

如果你想查看这两层的状况，可以用 **ethtool** 这个工具。比如这样：

复制代码

```
# ethtool -S enp0s3
NIC statistics:
     rx_packets: 45897
     tx_packets: 9457
     rx_bytes: 59125524
     tx_bytes: 834625
     rx_broadcast: 0
     tx_broadcast: 17
     rx_multicast: 0
     tx_multicast: 59
     rx_errors: 0
     tx_errors: 0
     tx_dropped: 0
```

它的原理，是网卡驱动会到内核中注册 ethtool 回调函数，然后我们用 ethtool 命令就可以查看这些信息了。由于信息是由网卡驱动提供的，所以十分“接地气”。

如果你在传输层和网络层的排查工具上，已经看到明确的链路不稳定的信息，那就直接找网络团队去处理吧。

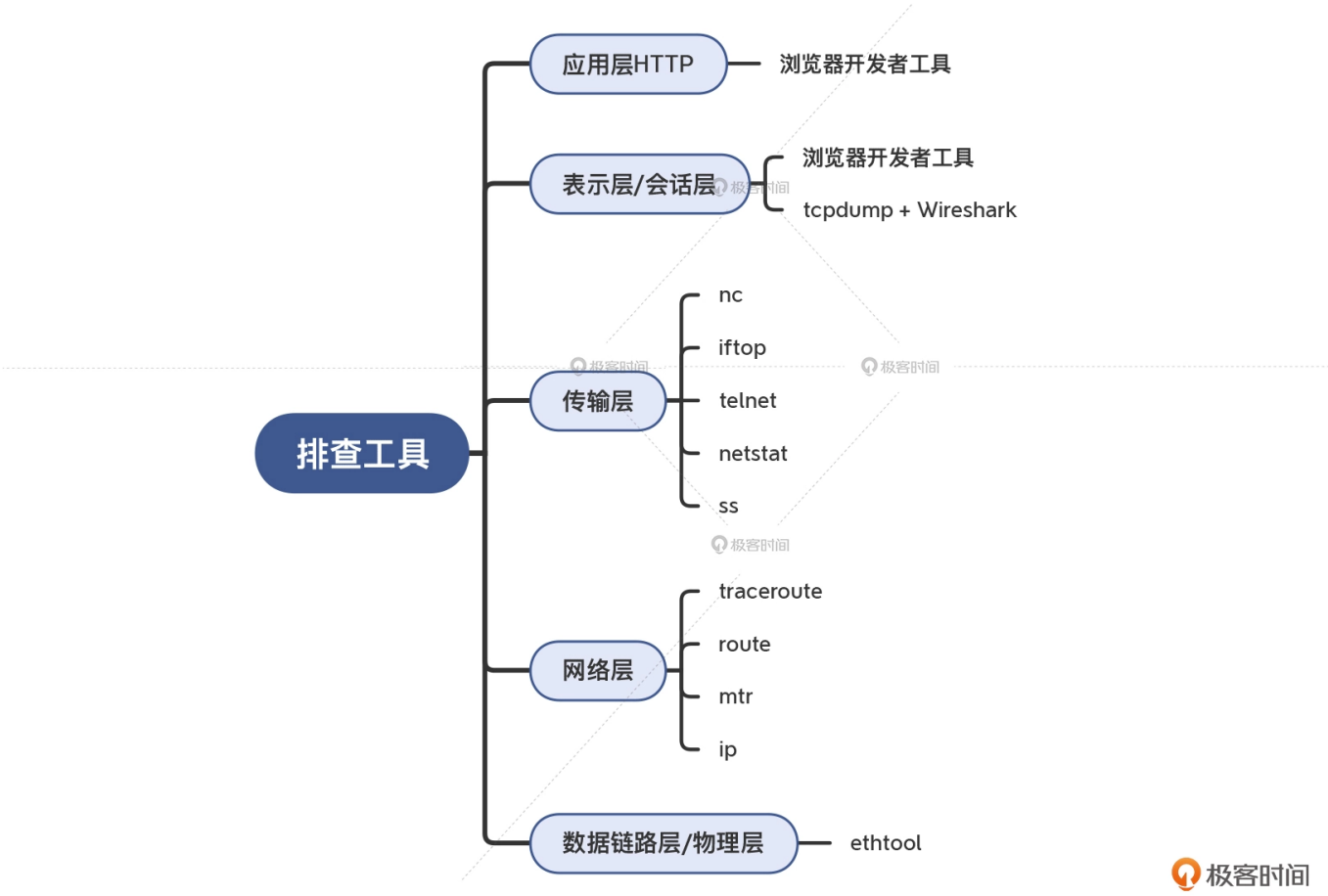
## 小结

这节课，我们回顾了网络分层模型，也了解了 OSI 模型和 TCP/IP 模型的区别和联系。通过“抠字眼”的方式，我们把每层的术语搞清楚，由此对分层模型有了更加深入的理解，这个对我们开展网络排查工作，有很强的指导性意义。

然后，我们逐一学习了各层的常用排查工具。我来给你再梳理一下：

1. 应用层以 HTTP 为例，可以用**浏览器开发者工具**，实现远程 IP 识别、耗时分析、Cookie 删除等需求。
2. 会话层和表示层以 TLS 为主，我们还是用**浏览器开发者工具**，可以查看证书细节、协商后使用的 Cipher suite 等信息，属于静态信息。然后学习了**用 tcpdump 和 Wireshark** 查看更详细的 TLS 握手细节的方法。这些信息是动态的，也只有用抓包分析的手段才能做到。
3. 在传输层，我们学到了 **telnet、nc、netstat、ss** 等命令，通过它们，我们可以测试连通性，也可以获取连接状况和统计信息，对于传输问题的排查都很有帮助。
4. 在网络层及以下的部分，我们学习了 **traceroute、mtr、ip** 等工具，可以检测网络路径状况。
5. 在数据链路层和物理层，我们可以做得不多，主要依靠网络层观察到的链路质量来推断这两次的情况。当然，也可以用 **ethtool** 这个工具查看这两层的详情。

最后，为了方便你复习，我也给你画了一张思维导图，让你能一目了然：




如果对这些命令的更多细节或者原理很感兴趣，在实战三模块里，我也会专门讨论这些工具相关的案例和使用技巧，相信会让你的网络排查技能变得更加丰富多元。

## 思考题

感谢你认真学完了这节课的内容，不过在结束之前，给你留几道思考题：

1. traceroute 默认是用 UDP 来做探测的，那这个又是基于什么原理呢？通和不通，我们会收到怎样的回复？
2. 有时候运行 telnet 后命令就挂起，没有响应了，这说明了什么问题呢？

欢迎你把答案写到留言区，我们一起交流讨论。也欢迎你把今天的内容分享给更多的朋友，一同成长和进步。

上一篇 开篇词 | 网络排查是工程师的必备能力

下一篇 02 | 抓包分析技术初探：你会用tcpdump和Wireshark吗？

## 精选留言 (8)



**webmin**

2022-01-12

1. traceroute 使用UDP探测时 初始时把TTL设置为1，经过路由器时TTL会被减1，当TTL变为0时，包被丢弃，路由器向源地址发回一个ICMP超时通知（ICMP Time Exceeded Message），源收到这个通知就会把下一次发送的包的TTL在原来的基础加1，这样就可以多前进一步，探测时使用了一个大于30000的端口号去连接，随着TTL的增加端口也会加1，目地服务器在收到这个数据包的时候会返回一个端口不可达的ICMP错误信息（ICMP Port Unr...

展开

作者回复: 1. 是的，TTL的操控是traceroute之类工具的核心，通过它实现了"逐跳"的探测。你这里提到了TTL为0时，路由器回复ICMP超时消息，这个补充非常好。

另外，文稿里的例子用默认UDP的方式遇到了"黑洞"，也就是对端并没有对这个UDP探测包回复ICMP port unreachable消息，但是它会对ICMP echo request探测包回复IMCP port unreachable消息，所以通过traceroute -I就可以看到完整的路径了。一般来说，对端要么对UDP模式有响应，要么对ICMP模式有响应，所以我们可以两个分别试一下。

2. 是的，telnet的握手SYN包发出去，但对端一直没回复SYN+ACK，导致telnet的connect()无法成功返回，造成了挂起。

8

**Alex_Shen**

2022-01-13

一. 原理 程序是利用增长存活时间（TTL）值来实现其功能的。每当数据包通过一个路由器，其存活时间就会减1。当其存活时间是0时，主机便取消数据包，并发送一个ICMP TTL数据包给原数据包的发出者。程序发出的首3个数据包TTL值是1，以后3个是2，如此类推，它便获得一连串数据包路径。注意IP不保证每一个数据包走的路径都同样。

1. Linux和Mac OS等系统使用UDP包进行探测，目标端口号默认为33434，每次探测目标...

展开

作者回复: 回复100分： )

对于问题一，稍作补充：如果udp端口号不是递增，会有个问题是无法判断返回的time-to-live exceeded报文是代表了哪一跳。由于这个ttl exceeded报文携带了这次探测的udp端口号信息，比如是34440，减去base number33434，就是6，那么就是第6跳回复的。

问题二，抱歉可能大家对"挂起"的理解不完全一致，我原意是指telnet没有失败退出，也没有成功到提示符，而是处在尝试连接状态中。
telnet成功后，表示握手成功，这时，我们可以继续发送文本，比如GET / HTTP/1.1
Host: www.baidu.com

这就是一次模拟的HTTP请求。
可以参考我这边的实验结果：

$ telnet www.baidu.com 80
Trying 180.101.49.11...
Connected to www.a.shifen.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.baidu.com

HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 9508
Content-Type: text/html
Date: Thu, 13 Jan 2022 13:19:58 GMT
P3p: CP=" OTI DSP COR IVA OUR IND COM "
P3p: CP=" OTI DSP COR IVA OUR IND COM "

1

**webmin**

2022-01-13

看到有同学问："为什么udp port需要增加？"
我的理解是traceroute是探测，有可能会遇到有的路由器在TTL减为0时不做回应，traceroute在等超时以后还会努力去再去尝试探测下一跳，traceroute每次用的端口是Base Port + T

LL，这样如果有回应包，traceroute就可以通过回应包中的目的地Port - BasePort就可以得知TTL是多少。...

展开

作者回复: 很赞~
是的，我观察到traceroute不是“挨个”探测的，而是一把发出多个不同TTL（和不同UDP端口）的探测报文，然后根据返回的情况，再判断的

1

**includestdio.h**
2022-01-12

1.traceroute UDP探测时，使用一个大于30000的端口号，服务器在收到这个数据包的时候会返回一个端口不可达的ICMP错误信息，客户端通过判断收到的错误信息是TTL超时，还是端口不可达来判断数据包是否到达目标主机。如收到超时则表示未得到对端主机应答，属于不通，收到端口不可达，则得到了对端主机的错误应答，属于通过
...

展开

作者回复: 回答很不错，给你点赞
“挂起”就是程序暂时没有响应，既不前进（成功）也不后退（报错失败）。telnet如果握手成功，是会进入正常提示符的，而挂起则不会，也意味着连接不成功（持续等待服务端的回应）。

共 3 条评论 >

1

**罗辑思维**
2022-01-14

telnet的握手SYN包发出去，但对端一直没回复SYN+ACK，导致telnet的connect()无法成功返回，造成了挂起。

这个一般会是什么原因呢？
是对端繁忙，还是防火墙呢？

展开

作者回复: 嗯，这个要结合客户端和服务端的抓包，对比来分析：
1. 客户端抓包中有发出SYN，但在服务端抓包中没收到SYN，一般是客户端去往服务端的问题，检查这个方向上的网络设备；虚拟化情况下，也可能是客户端或者服务端VM/pod所在的hypervisor上的virtual switch等环节做了“手脚”，这些也在广义的网络设备范畴内，也要查；
2. 在服务端抓包中，发现SYN包进来了，但服务端没回SYN+ACK，那就检查本地是否有拦截（比如iptables）、端口监听是否正常、系统资源是否正常、应用的网络部分的代码是否正常、应用层

的network syscall的返回是否正常；
3. 在服务端抓包中，发现SYN包进来了，自己也回了SYN+ACK，但客户端抓包中没收到SYN+ACK，推进方法跟1类似。

在1的服务端的入方向，tcpdump抓包发生在iptables规则起作用之前，所以如果SYN/SYN+ACK已经来了，tcpdump就能看到SYN，但iptables接着做拦截的话，这个连接也还是无法建立（跟网络上丢失的效果一致），这时候虽然见着了SYN但协议栈没回复SYN+ACK，却并不是协议栈的问题（而是iptables拦截导致）。这个分析也适用于3的客户端入方向。

可能没有涵盖所有情况，先想到这些：）防火墙的话，会引起1和3，当然防火墙也会引起别的问题。关于防火墙的内容，下周上第5和第6课就可以学习到了。


**Middleware** 酥
2022-01-13

三层交互机是个什么意思？

作者回复: 三层交换机是有路由功能的交换机。一般来说交换机是工作在二层的，所以这里强调“三层”交换机，就是突出了它带有三层（路由）功能。


**Dexter** 酥
2022-01-12

为什么udp port需要增加？没什么意义吧

作者回复: 看明白你的问题了，是指traceroute udp模式（也就是默认模式）下，为什么每次发送的udp探测包，其目的端口是递增的对吧？这样可以通过返回的ICMP消息里面携带的payload（这是一个“内嵌”的udp数据包），识别到这个ICMP是呼应了哪个udp探测包。如果每次端口号都相同，那所有的ICMP回复，就没有区别了，这并不是我们想要的。

共 3 条评论 >


**Christopher** 酥
2022-01-12

这种课真是运维同学最爱

作者回复: 谢谢支持： )

共 5 条评论 >