

计算机网络实验报告

Lab2 Web服务器交互抓包分析

2112338 李威远 计算机学院 计算机卓越班

一、报告说明：

在本次实验中，我采取了不同方式进行Web服务器搭建，实际wireshark中抓包的结果，在不经过具体跟踪TCP流或详细清洗前是存在很多不同的（但每个TCP流内部是正确的三次握手、四次挥手的流程）。

起初我只使用了flask，但由于flask的某一特性，使得结果只有短连接，这一过程让我思考了很久，在咨询老师后了解到了原因，这让我对不同方式下的具体处理产生了浓厚的兴趣。

因此，在完成我们实验所需的抓包分析的基础之上，我展开调研了基于**flask**、**django**、**node.js**和**phpnow**等web开发框架进行搭建和直接应用web服务器套件**IIS**和**NGINX**进行搭建等六种方式，对**edge**、**Chrome**、**Firefox**、**IE**多个浏览器进行测试，使用wireshark过滤并捕获数据包传递过程，分析其异同及其原因，深入挖掘不同框架、服务器和不同浏览器下TCP连接的特殊处理。

事实上，这些交互过程的异同是由于不同的服务器配置和浏览器优化造成的。

其中，利用node.js框架搭建服务器，在edge、Chrome浏览器访问进行实验，在我的电脑环境下能够完美实现仅存在一次TCP连接、结果最为清晰明了，虽然由于其存在流水线优化访问的现象，会多出现三次握手，等待一段时间后会多出现四次挥手（原因稍后会解释），但结果易于清洗和观察。

因此我们以node.js、为主要分析对象，按照以下顺序进行实验的汇报

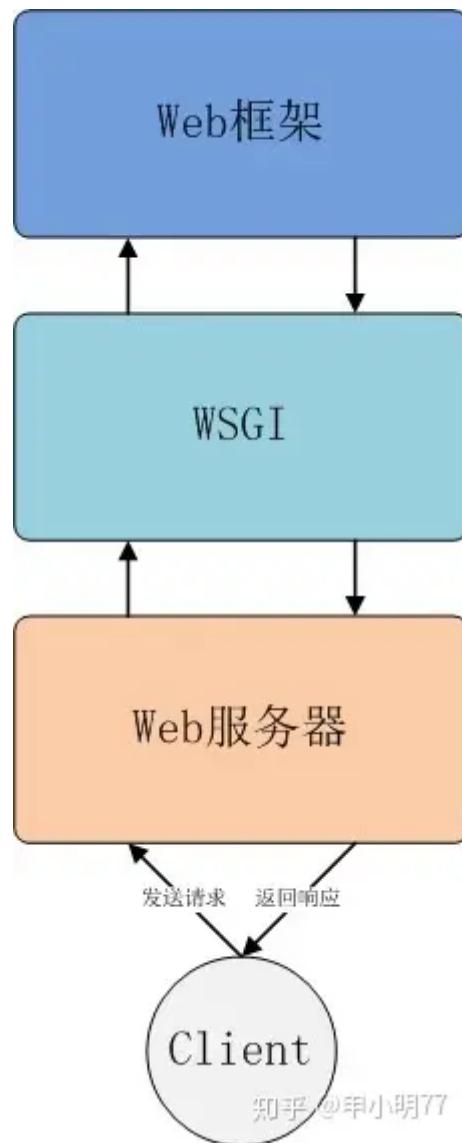
1. 在报告第二部分介绍其Web服务器搭建、编写Web页面的过程。
2. 在报告第三部分展开我们浏览器与服务器交互过程的wireshark的详细分析。
3. 在报告第四部分与其他框架和套件的TCP连接过程作对比分析，了解不同框架和套件的不同特性。

此外，由于我最初使用了Flask框架，且IIS套件的环境配置不同于开发框架，在第二部分同样展开详细介绍其环境配置过程。其余框架和套件仅在第四部分作对比时候介绍和展示。

二、Web服务器搭建、编写Web页面：

当然，在展开交互过程分析前，我们先梳理我进行的六种建站方式的服务器搭建过程和我们编写的Web界面，其中有四种是Web框架、两种为Web服务器套件

这里，还需要认识一下这些Web框架和Web服务器的关系，下图为我们所用到的flask等Web框架和NGINX等WEB服务器套件之间的关系（以flask为例），其中，WSGI是一种接口，它只适用于 Python 语言，其全称为Web Server Gateway Interface，定义了web服务器和web应用之间的接口规范，而Flask框架基于的Werkzeug正是一个WSGI工具包。



因此，当我们使用Flask等开发框架的时候，实际上是其中调用的封装方法，如Flask的app.run()方法，在本地部署搭建了服务器项目，这是一个简单的内部开发服务器，当然，它只能建立在本地上，无法部署到远程服务器上，实际开发需要结合IIS、NGINX等web服务器套件进行部署。

而我们使用IIS、NGINX等web服务器套件的时候，可以在本地和远程服务器部署网站，这些网站被称为生产级Web服务器。但是，单一使用这两个服务器套件的时候，是无法实现动态页面的管理的，即只能实现仅含前端的静态页面，实际开发需要结合Flask等框架使用。

回到我们实验本身，我们只需要建立一个Web服务器，且仅有前端上静态网页的需求，不需要实现后端，因此采用两种方法都是可以的。接下来，我来介绍一下如何配置这些环境

(一) Flask:

Flask是我最常使用的Web开发框架，使用python进行后端的开发，相对于Django十分轻量简便。也是我进行本次实验最初使用的框架，但是它的交互过程出现了一些问题，也正是它让我对不同框架的特性产生了兴趣，扩展了本次实验的方向。

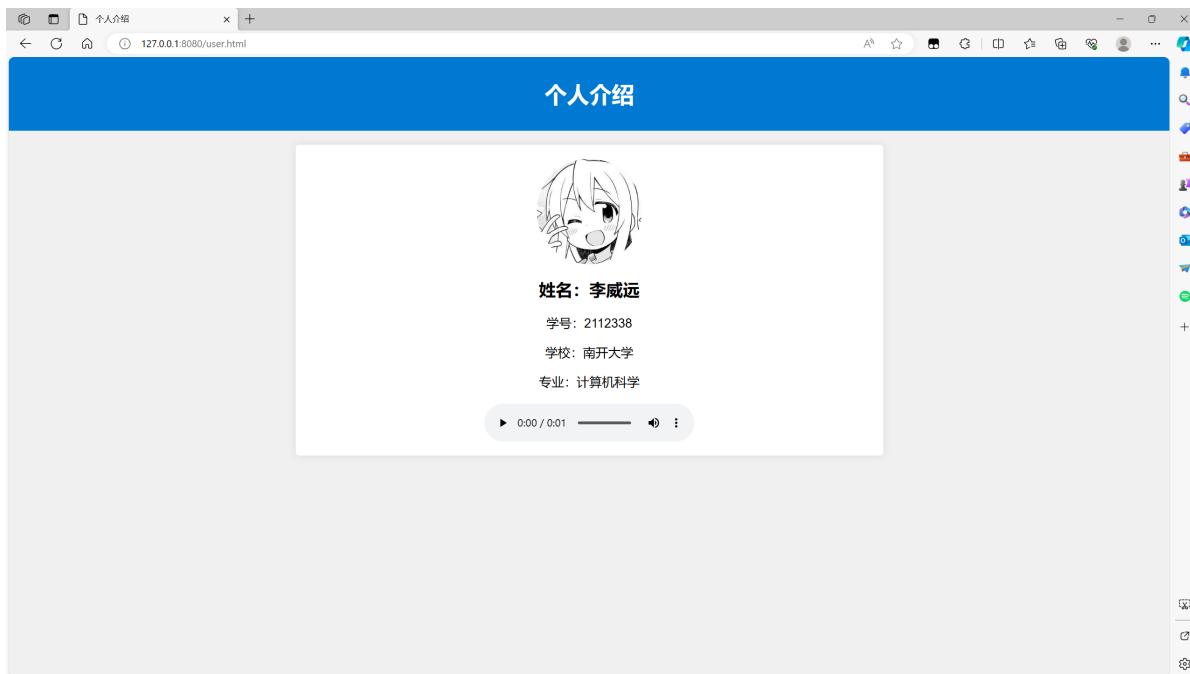
利用pycharm新建Flask项目，项目结构如下所示：

```
+-- Flask/
|   +-- static/
|   |   +-- BGM.wav
|   |   +-- head.jpg
|   |   +-- style1.css
|   +-- templates/
|   +-- main.py
```

在pycharm中直接运行main.py文件即可，这里将Flask下使用的端口设置为8080，使用本地环回地址127.0.0.1作为测试开发服务器环境。

具体的代码已经上传到我的[github](#)中，这里不作过多赘述，其中，为了保证不同框架相同，之后的所有框架和套件中，我使用的都是如下html代码，通过style1.css进行简单的渲染，包含图片和音频文件。

打开浏览器，输入127.0.0.1:8080/user.html，具体效果在浏览器中显示如下：



(二) Node.js:

Node.js利用了JavaScript实现前后端的整合合一，同样也是较为主流的Web开发框架。这里我们首先配置html文件、app.js启动文件和静态文件，项目结构如下所示。

```
+-- Node.js/
|   +-- static/
|   |   +-- BGM.wav
|   |   +-- head.jpg
|   |   +-- style1.css
|   +-- main.html
|   +-- app.js
```

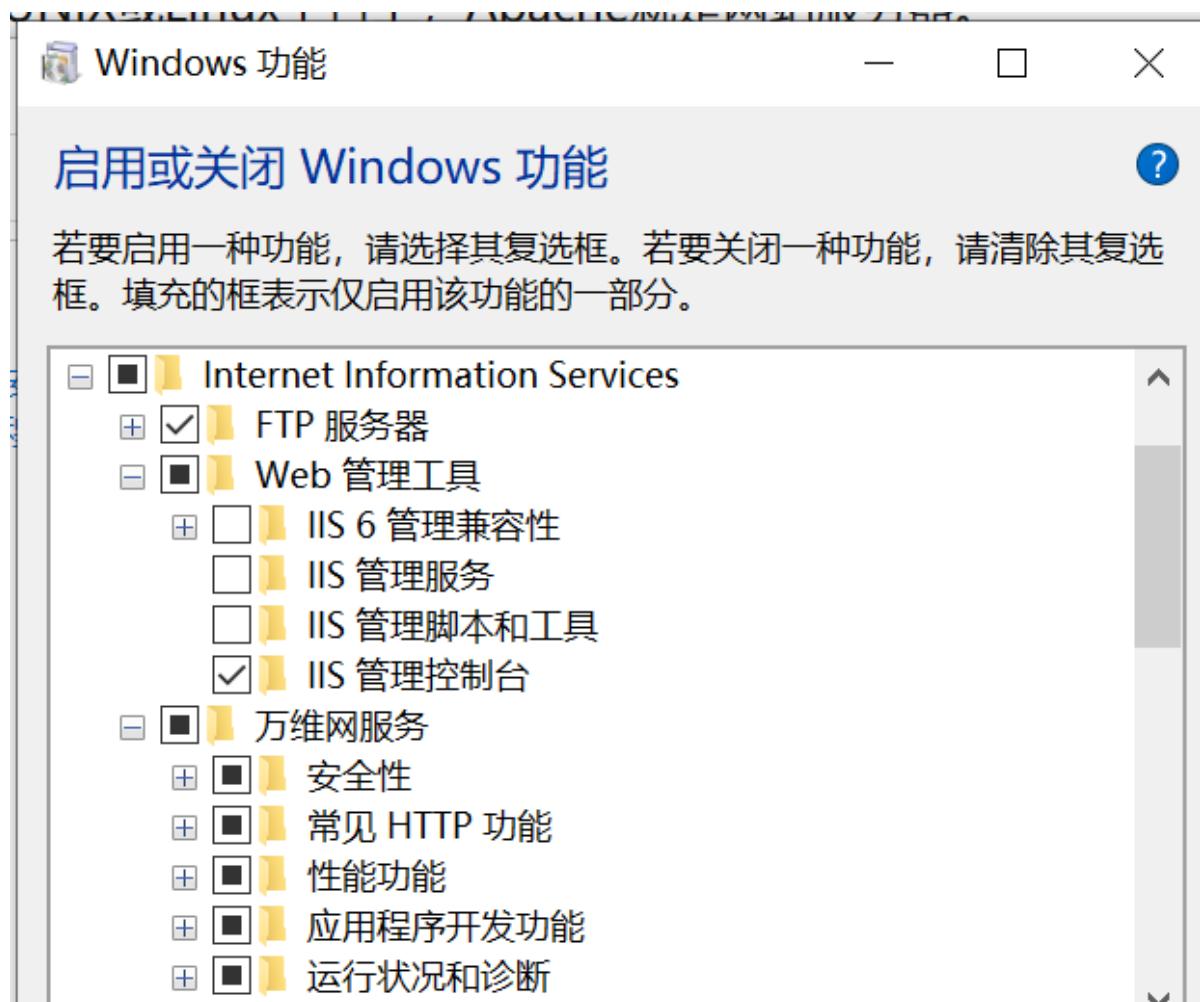
随后，利用 `npm i` 指令初始化node.js的配置文件，然后使用 `node app.js` 指令初始化开发服务器环境，页面打开效果与Flask中相同，代码也在[github](#)中给出。

(三) IIS:

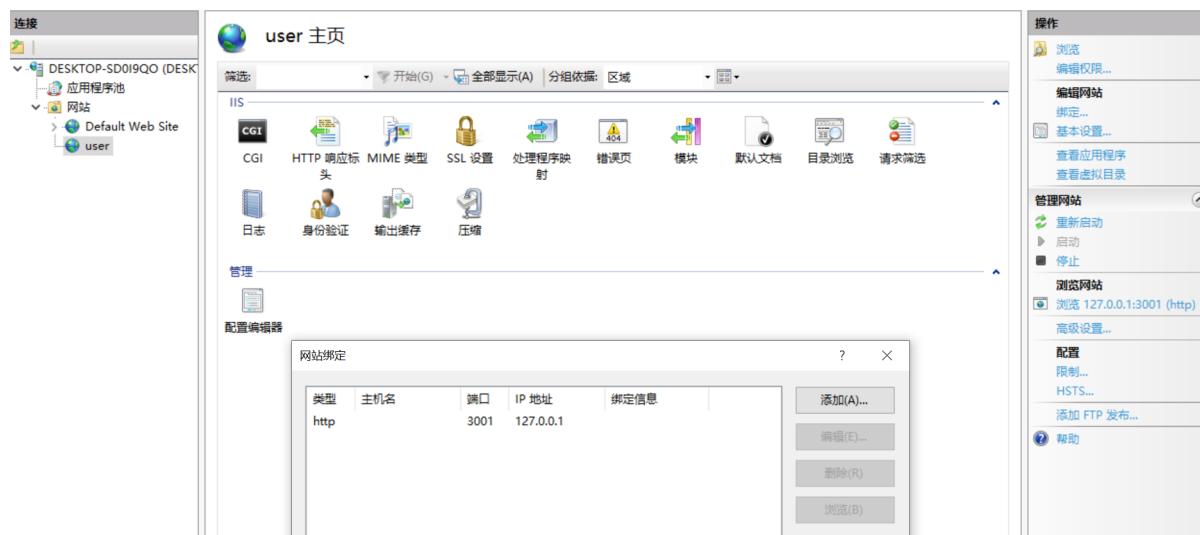
IIS是指World Wide Web server服务，IIS是一种Web（网页）服务组件，IIS可以赋予一部主机电脑一组以上的IP地址，而且还可以有一个以上的域名作为Web网站。

IIS是一种服务，是Windows 2000 Server系列的一个组件。不同于一般的应用程序，它就像驱动程序一样是操作系统的一部分。

因此，我们首先需要启用IIS相关服务，打开控制面板，在程序设置中选择“启用或关闭Windows功能”，在其中勾选IIS相关服务，点击确认启动IIS服务。



随后，我们运行IIS，在其中新建网址，并将其绑定到ip地址为127.0.0.1的3001端口处，如下图所示：



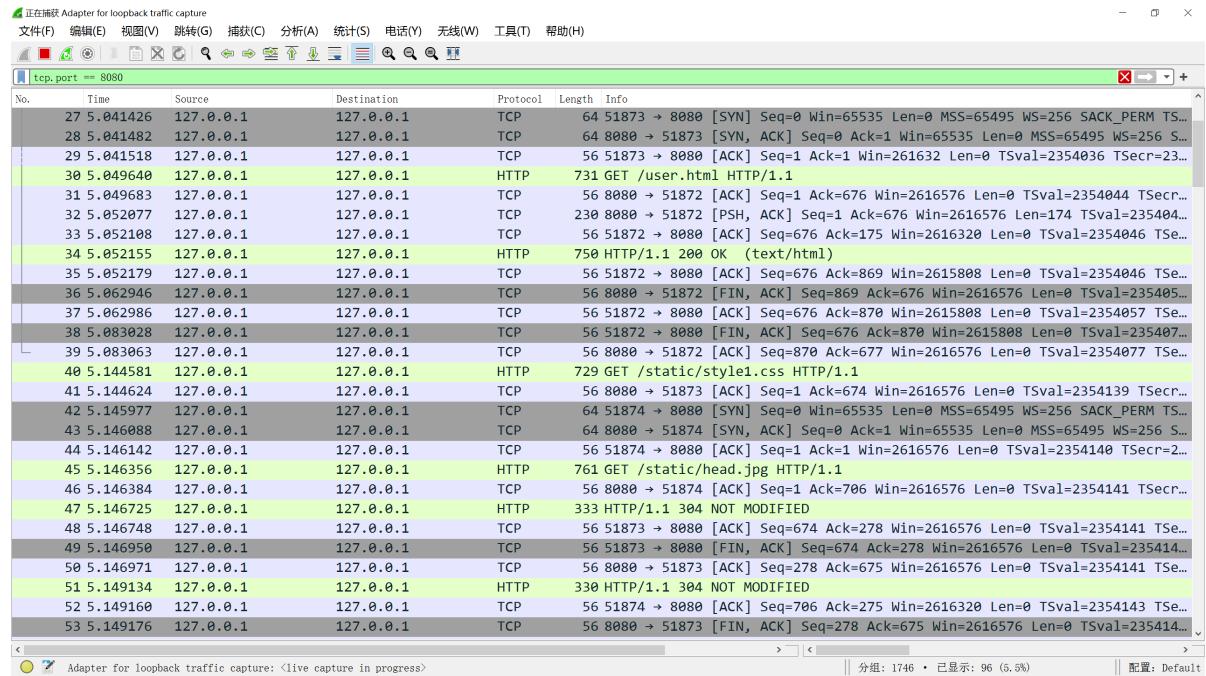
再将其映射到本机的某一静态目录下，就可以在浏览器中访问静态网页了，其页面效果和先前也是一样的。

(四) 总结

其他几个框架和服务器套件的使用不再赘述，基本上是类似的。

综上，我们成功运用开发框架或服务器套件，在本机ip地址的不同端口上，部署了浏览器可访问的服务器。

但是，这几种方式部署的服务器，在浏览器中进行访问的效果是一致的吗？我们通过初步使用wireshark进行抓包分析，发现了Flask部署服务器的抓包结果如下图所示，即出现了短连接的现象：



当然，即使是短连接也是满足http协议的，但是明明是http1.1，为什么会变成短连接呢？我们放到后面第四部分的对比分析再说。接下来，我们以仍然为长连接的Node.js的抓包结果作为分析对象，考察整个浏览器访问服务器的交互过程。

三、交互过程分析(以Node.js为例):

(一) 抓包操作流程及结果展示:

利用firefox浏览器对Node.js的本地服务器端127.0.0.1:3000处进行访问，并利用wireshark进行过滤，过程如下：

1. 启动wireshark抓包软件，选择Adapter for loopback traffic capture 接口，并利用过滤指令
`tcp.port == 3000` 监听3000端口处的数据流
2. 打开浏览器，访问127.0.0.1:3000地址，一边观察wireshark抓包结果一边等待，一段时间后关闭wireshark
3. 保存抓包结果，如下图所示。

Adapter for loopback traffic capture

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

tcp_port == 3000

No.	Time	Source	Destination	Protocol	Length	Info
419 8.368245	127.0.0.1	127.0.0.1	TCP	64	35201 → 3000 [SYN] Seq=0 Win=65535 MSS=65495 WS=256 SACK_PERM TSval=6697706 TSecr=0	
420 8.368204	127.0.0.1	127.0.0.1	TCP	64	3000 → 35201 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=6697707 TSecr=669...	
421 8.368235	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=1 Ack=1 Win=0 TSval=6697707 TSecr=6697707	
422 8.368235	127.0.0.1	127.0.0.1	HTTP	599	GET / HTTP/1.1	
423 8.368433	127.0.0.1	127.0.0.1	TCP	1054	HTTP/1.1 200 OK (text/html)	
428 8.369711	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=535 Ack=999 Win=2616576 Len=0 TSval=6697708 TSecr=6697708	
466 8.479647	127.0.0.1	127.0.0.1	HTTP	526	GET /static/style.css HTTP/1.1	
467 8.479657	127.0.0.1	127.0.0.1	TCP	56	3000 → 35201 [ACK] Seq=999 Ack=1005 Win=2616064 Len=0 TSval=6697818 TSecr=6697818	
476 8.488454	127.0.0.1	127.0.0.1	HTTP	1037	HTTP/1.1 200 OK (text/css)	
477 8.488470	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=1005 Ack=1980 Win=2615552 Len=0 TSval=6697819 TSecr=6697819	
486 8.488669	127.0.0.1	127.0.0.1	HTTP	531	GET /static/head.ing HTTP/1.1	
487 8.489669	127.0.0.1	127.0.0.1	TCP	56	3000 → 35201 [ACK] Seq=1980 Ack=1480 Win=2615552 Len=0 TSval=6697819 TSecr=6697819	
508 8.489998	127.0.0.1	127.0.0.1	TCP	64	35202 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=6697819 TSecr=0	
501 8.481845	127.0.0.1	127.0.0.1	TCP	64	3000 → 35201 [SYN, ACK] Seq=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=6697819 TSecr=669...	
502 8.481867	127.0.0.1	127.0.0.1	TCP	56	35202 → 3000 [ACK] Seq=1 Ack=1 Win=0 TSval=6697819 TSecr=6697819	
506 8.481177	127.0.0.1	127.0.0.1	HTTP	29297	HTTP/1.1 200 OK (JPEG JFIF image)	
508 8.481198	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=1480 Ack=31221 Win=2586368 Len=0 TSval=6697820 TSecr=6697820	
553 8.493156	127.0.0.1	127.0.0.1	HTTP	603	GET /static/BGM.wav HTTP/1.1	
554 8.493269	127.0.0.1	127.0.0.1	TCP	56	3000 → 35201 [ACK] Seq=31221 Ack=2027 Win=2615040 Len=0 TSval=6697832 TSecr=6697832	
555 8.494177	127.0.0.1	127.0.0.1	TCP	65473	3000 → 35201 [ACK] Seq=1480 Ack=2027 Win=2615040 Len=55417 TSval=6697833 TSecr=6697833 [TCP segment o...	
556 8.494194	127.0.0.1	127.0.0.1	TCP	532	3000 → 35201 [PSH, ACK] Seq=96538 Ack=2027 Win=2615040 Len=476 TSval=6697833 TSecr=6697832 [TCP segment o...	
557 8.494222	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=96538 Ack=2027 Win=2615040 Len=476 TSval=6697833 TSecr=6697833	
558 8.494358	127.0.0.1	127.0.0.1	TCP	65473	3000 → 35201 [ACK] Seq=97114 Ack=2027 Win=2615040 Len=55417 TSval=6697833 TSecr=6697833 [TCP segment o...	
559 8.494358	127.0.0.1	127.0.0.1	TCP	175	3000 → 35201 [PSH, ACK] Seq=162543 Ack=2027 Win=2615040 Len=119 TSval=6697833 TSecr=6697833 [TCP segment o...	
568 8.494377	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=2027 Ack=162650 Win=2615040 Len=0 TSval=6697833 TSecr=6697833	
561 8.494458	127.0.0.1	127.0.0.1	HTTP	11114	HTTP/1.1 206 Partial Content (audio/x-wav)	
562 8.494473	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=2027 Ack=173708 Win=2605568 Len=0 TSval=6697833 TSecr=6697833	
579 8.510887	127.0.0.1	127.0.0.1	HTTP	527	GET /favicon.ico HTTP/1.1	
588 8.510982	127.0.0.1	127.0.0.1	TCP	56	3000 → 35201 [ACK] Seq=173708 Ack=2499 Win=2614528 Len=0 TSval=6697849 TSecr=6697849	
581 8.511515	127.0.0.1	127.0.0.1	HTTP	478	HTTP/1.1 404 Not Found (text/html)	
582 8.511534	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=2498 Ack=174136 Win=2605956 Len=0 TSval=6697850 TSecr=6697850	
867 13.524606	127.0.0.1	127.0.0.1	TCP	56	3000 → 35201 [FIN, ACK] Seq=174136 Ack=2498 Win=2614528 Len=0 TSval=6702863 TSecr=6697850	
868 13.524629	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [ACK] Seq=2498 Ack=174131 Win=2605956 Len=0 TSval=6702863 TSecr=6702863	
869 13.524659	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [FIN, ACK] Seq=2498 Ack=174131 Win=2605956 Len=0 TSval=6702863 TSecr=6702863	
879 13.524725	127.0.0.1	127.0.0.1	TCP	56	3000 → 35201 [ACK] Seq=174131 Ack=2499 Win=2614528 Len=0 TSval=6702863 TSecr=6702863	
879 13.763198	127.0.0.1	127.0.0.1	TCP	56	35201 → 3000 [FIN, ACK] Seq=1 Ack=1 Win=2616576 Len=0 TSval=6703101 TSecr=6697819	
880 13.763121	127.0.0.1	127.0.0.1	TCP	56	3000 → 35202 [ACK] Seq=1 Ack=2 Win=0 TSval=6703102 TSecr=6703101	

Frame 472: 598 bytes on wire (4792 bits), 598 bytes captured (4792 bits) on interface \Device\NPF_Linnenhack_id_0

Internet Protocol Version 4 (ip), 20 byte(s)

分组数: 2452 • 已显示: 40 (1.6%) 配置: Default

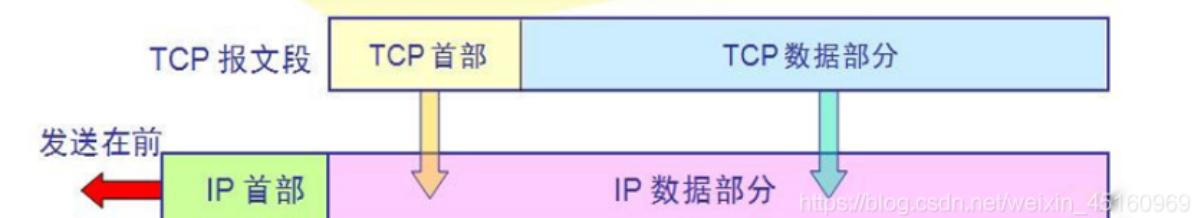
但是，这里可以看到，我们进行了六次握手和八次挥手，即有了两次tcp连接，实际上，这里只有一次tcp连接是存在数据传输的，另一个tcp连接不进行任何操作。这是为什么呢？这属于浏览器的优化范畴，我们还是放到第四部分再说。

接下来，我们进一步追踪TCP流，只对其中存在数据交互的TCP连接进行交互过程分析，考察其建立中的TCP连接的三次握手、四次挥手和Get请求发送获取数据的过程。

(二) 报文结构梳理:

在我们的TCP连接中，我们用到TCP报文、HTTP请求报文和HTTP响应报文，在开始整个的分析之前，有必要梳理我们用到的这三种报文的格式，帮助我们更好的认识这整个过程。

1、TCP报文



TCP报文段是TCP连接中传输的基本单元，其既可以用来运载数据，也可以用于建立连接、释放连接和应答。一个 TCP 报文段分为首部和数据两部分，整个TCP 文段作为IP数据报的数据部分封装在 IP 数报报中，如上图所示。

其中，我们重点来考察我们实验中会用到的一些标记字段和重要信息位，包含如下六个标记位（抓包过程未涉及URG）：

1. **确认位ACK**。仅当ACK=1 时确认号字段才有效。当ACK=0时，确认号无效（确认号用于划分数据块传输保证无数据丢失）。TCP 规定，在连接建立后所有传送的报文段都必须把ACK 置1。
需要注意的是，在wireshark中的Ack**并非确认位**，而是确认号，确认号的定义在后面给出。
2. **推送位 PSH(Push)**。接收方 TCP 收到 PSH=1 的报文段，就尽快地交付给接收应用进程而不再等到整个缓存都填满了后，再向上交付。
3. **复位位 RST(Reset)**。当 RST=1时，表明 TCP 连接中出现严重差错(如主机崩溃或其他原因)，必须释放连接，然后再重新建立运输连接。
4. **同步位 SYN**。当SYN=1时表示这是一个连接请求或连接接受报文。当SYN=1， ACK=0 时，表明这是一个连接请求报文，对方若同意建立连接，则应在响应报文中使用SYN=1， ACK=1。
5. **终止位 FIN(Finish)**。用来释放一个TCP连接。当FIN=1时，表明此报文段的发送方的数据已发送完毕，并要求释放运输连接。
6. **紧急位 URG**。当URG=1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送（相当于高优先级的数据）。URG 需要和首部中紧急指针字段配合使用，即数据从第一个字节到紧急指针所指字节就是紧急数据。

此外，还包含以下几个重要信息：

1. **源端口和目的端口**。端口是传输层与应用层的服务接口，传输层的复用和分用功能都要通过端口实现。
2. **窗口 Win**。它指出现在允许对方发送的数据量，接收方的数据缓存空间是有限的，因此用窗口值作为接收方让发送方设置其发送窗口的依据。
3. **序号Seq**。TCP 是面向字节流的（即TCP 传送时是逐个字节传送的），所以TCP 连接传送的字节流中的每个字节都按顺序编号。序号字段的值指的是本报文段所发送的数据的第一个字节的序号。
4. **确认号 Ack**。注意与ACK的区分，确认号是期望收到对方下一个报文段的第一个数据字节的序号。若确认号为N，则表明到序号N-1为止的所有数据都已正确收到。
5. **其他信息**。其他的一些信息可能是TCP段的Option涵盖、wireshark得到的额外信息、和存在IPv4头部当中的信息，如TStamp、TSecur时间戳、Len这种TCP段的数据长度等等。

2. HTTP请求报文和HTTP响应报文

首先需要明确的概念是，**HTTP报文本质上也是TCP报文**，它可以分为HTTP请求报文和HTTP响应报文。在Wireshark中看到的HTTP报文内容是来自TCP报文段的数据部分，而不是TCP报文段的首部。

(1) HTTP请求报文



如上图所示，其为HTTP请求报文的内容示例，可以看到，其由三部分组成：

- **请求行**：包含请求方法、URL地址（和请求头中的Host属性共同组成完整的请求URL）、协议名称及版本
- **请求头**：即HTTP的报文头，其中包含若干属性，格式为“属性名：属性值”，服务端据此获取客户端信息。
- **请求体**：按照键值对的与的格式编码形成的多个请求参数的数据。

其中，我们重点考察请求头中的一些属性，这些属性大多与浏览器自己的设置相关，也就是造成了不同浏览器效果的不同的根本原因。

1. **Host**：请求报头域主要用于指定被请求资源的Internet域名和端口号，它通常从HTTP URL中提取出来的。
2. **Referer**：先前网页的地址，当前请求网页紧随其后。
3. **User-Agent**：该属性标识着操作系统和浏览器的版本等信息。
4. **Accept**：指定客户端能够接收的内容类型，如文本、图片等等。
5. **Accept-Charset**：指定客户端能够接收的各类东西类型，如Accept-Charset指浏览器能够接收的字符编码集等等。
6. **Connection**：表示是否需要持久连接，为close则是关闭。
7. **Cache-Control**：指定客户端遵循的缓存机制。
8. **If-None-Match等**：为辅助匹配的属性，如缓存内容未修改则返回304代码，通过这些属性设置比较的时间等等。
9. **Cookie**：本次实验未出现，该属性控制我们很熟悉的cookie机制下的所有cookie值。

(2) HTTP响应报文



如上图所示，其为HTTP响应报文的内容示例，可以看到，其同样由三部分组成：

- **响应行**：包含报文协议及版本、状态码及其描述
- **响应头**：即HTTP的报文头，其中同样也包含若干属性，格式为“属性名：属性值”，客户端据此获取服务端信息。
- **响应体**：按照键值对的与的格式编码形成的多个响应参数的数据，即我们请求想要得到的数据。

其中，我们还是重点考察响应头中的一些属性，这些属性大多与服务器自己的设置相关，也就是造成了不同服务器效果的不同的根本原因。

1. **与请求头相同的属性**：其中，有一些属性是二者都可能有的，如Connection、Cache-Control
2. **Content-Type**：响应内容的类型。
3. **Keep-alive**：设置长连接的最大时限。

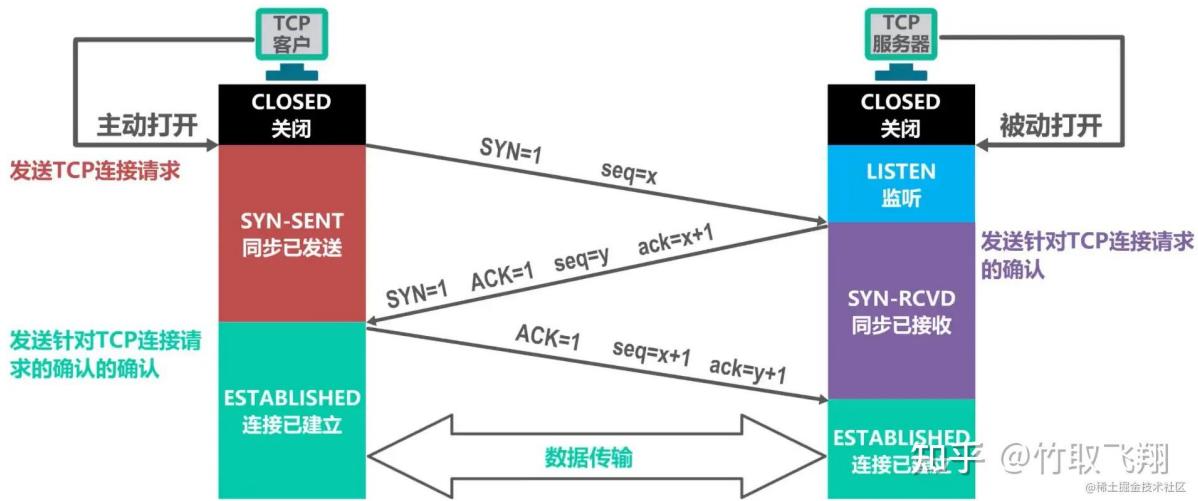
其他还有一些特殊的属性，因为和实验关联不大就没有过多了解，这里贴一下我参考的一个博客，更具体的解释可以在里面找到：[HTTP请求头、响应头的属性](#)。

此外，响应状态码的含义也需要我们大致的了解，其中大致划分如下：

- **1xx 消息**：一般是告诉客户端，请求已收到，正在处理。
- **2xx 处理成功**：一般表示请求已受理、已经处理完成等信息。
- **3xx 重定向到其它地方**：它让客户端再发起一个请求以完成整个处理。
- **4xx 处理发生错误**：责任在客户端，如客户端的请求一个不存在的资源，客户端未被授权，禁止访问等。
- **5xx 处理发生错误**：责任在服务端，如服务端抛出异常，路由出错，HTTP版本不支持等。

(三) 抓包结果分析：

1、“三次握手”的建立连接过程



如上图所示，TCP 通过三次交互来实现 TCP 连接的建立，即“三次握手”，在 wireshark 的抓包结果中，我们可以清晰的看到这三次握手：

Protocol	Length	Info
TCP	64	35201 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256
TCP	64	3000 → 35201 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65
TCP	56	35201 → 3000 [ACK] Seq=1 Ack=1 Win=261632 Len=0 TSval=6697

在连接建立之前，服务器进程（3000 端口）会处于“LISTEN”收听阶段，等待客户进行连接的请求。

（1）第一次握手：

第一次握手是指，首先客户端（35201）向服务器（3000）发送一个 SYN 包，并等待服务器确认，完成如下这些功能：

- 标志位为 SYN，根据我们先前汇总，表示请求建立连接；
- 客户端选择一个初始序号为 $Seq = x$ ；
- 客户端进入 SYN-SENT（同步已发送）阶段，等待服务器的响应。

考察第一次请求中 TCP 段的信息，在 wireshark 中发现如下图所示：

```

Transmission Control Protocol, Src Port: 35201, Dst Port: 3000, Seq: 0, Len: 0
  Source Port: 35201
  Destination Port: 3000
  [Stream index: 6]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 0]
  Sequence Number: 0 (relative sequence number)
  Sequence Number (raw): 2618481987
  [Next Sequence Number: 1 (relative sequence number)]
  Acknowledgment Number: 0
  Acknowledgment number (raw): 0
  1010 .... = Header Length: 40 bytes (10)
  Flags: 0x002 (SYN)
    000. .... = Reserved: Not set
    ...0 .... = Accurate ECN: Not set
    .... 0.... = Congestion Window Reduced: Not set
    .... .0.. = ECN-Echo: Not set
    .... ..0. = Urgent: Not set
    .... ...0 .. = Acknowledgment: Not set
    .... ....0 .. = Push: Not set
    .... ...0 .. = Reset: Not set
    > .... ...1. = Syn: Set
    .... ...0 .. = Fin: Not set
    [TCP Flags: .....S..]
  Window: 65535
  [calculated window size: 65535]
  Checksum: 0x0e49 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  Options: (20 bytes) Maximum segment size, No-Operation (NOP), Window scale, SACK permitted, Timestamps
    > TCP Option - Maximum segment size: 65495 bytes
    > TCP Option - No-Operation (NOP)
    > TCP Option - Window scale: 8 (multiply by 256)
    > TCP Option - SACK permitted
    > TCP Option - Timestamps

```

这里，我们能够看到许多TCP段的重要信息，可以看到，这时候wireshark栏目中有：

1. Seq其实是相对序列号，客户端的真实序列号为2618481907。
2. 而客户端的Ack确认号目前还没有分配，设置为0。
3. Ack相应的ACK标记位这时候也为0，而SYN请求位为1，表示请求建立连接。
4. Window为滑动窗口大小，初始定义为65535。
5. 同时，在Option端中，我们看到了MSS（最大段长度）的值为65495，这和后面的可靠字节流的分段处理相关。

在实际意义上，我们可以认识到，若第一次握手成功，服务端能确定客户端的发送功能和服务器端的接收功能正常，但此时客户端还不知道这个事实。

(2) 第二次握手：

第二次握手是指服务器接收到客户端发来的 SYN 包后，对该包进行确认后结束 LISTEN 阶段，并返回一段 TCP 报文，其中：

- 标志位为 SYN 和 ACK，表示确认客户端的报文 Seq 序号有效，服务器能正常接收客户端发送的数据，并同意创建新连接；
- 分配服务器端的序号为 Seq = y；
- 服务器端的确认号为 Ack = x + 1，表示收到客户端的序号 Seq 并将其值加 1 作为自己确认号 Ack 的值，随后服务器端进入 SYN-RECV 阶段。

这里需要认识到的是，TCP 规定，**SYN 报文段不能携带数据，但要消耗掉一个序号**。因此需要给 Ack+1。此外，**服务器端将在第二次握手时分配资源，因此，第三次握手中Win等信息可能会改变**。

考察第二次握手的TCP段内容，如下所示：

```
Transmission Control Protocol, Src Port: 3000, Dst Port: 35201, Seq: 0, Ack: 1, Len: 0
Source Port: 3000
Destination Port: 35201
[Stream index: 6]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 0 (relative sequence number)
Sequence Number (raw): 976454346
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 2618481908
1010 .... = Header Length: 40 bytes (10)
Flags: 0x012 (SYN, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0.... .... = Congestion Window Reduced: Not set
    .... .0.... .... = ECN-Echo: Not set
    .... ..0.... .... = Urgent: Not set
    .... ..1.... .... = Acknowledgment: Set
    .... .0.... .... = Push: Not set
    .... .... 0... = Reset: Not set
    > .... .... 1.. = Syn: Set
    .... .... 0.. = Fin: Not set
    [TCP Flags: ....A..S..]
Window: 65535
```

其中，Window等信息与第一次握手相同，这里不做展示，可以看到wireshark栏目中有：

1. 服务器端的Seq此时完成分配，初始值为976454346。
2. 服务器端的Ack是基于客户端传入的Seq+1，即期望下一次收到的消息的数据的开始序列号，保证可靠字节流的传输。
3. 这里，SYN和ACK标记位都被启用，表示收到了客户端的请求信息，同时向客户端发送请求信息。

在实际意义上，我们可以认识到，若第二次握手成功，客户端能够得知服务端确实收到了消息，且成功给自己发送了消息，即客户端确定服务端、客户端的收发功能都正常；而服务端只能确定客户端的发送功能和服务器端的接收功能正常，**不能确认自己的发送功能和客户端的接收功能是否正常**。

因此，第三次握手是必须的。

(3) 第三次握手：

第三次握手是指客户端接收到发送的 SYN + ACK 包后，明确了从客户端到服务器的数据传输是正常的，从而结束 SYN-SENT 阶段。并返回最后一段报文。其中：

- 标志位为 ACK，表示确认收到服务器端同意连接的信号；
- 客户端序号为 Seq = x + 1，表示收到服务器端的确认号 Ack，并将其值作为自己的序号值；
- 客户端确认号为 Ack= y + 1，表示收到服务器端序号 seq，并将其值加 1 作为自己的确认号 Ack 的值。
- 随后服务端、客户端进入 ESTABLISHED。
- 第二次握手中，服务器端完成资源分配，因此实际分配的Win值可能会改变。

这里需要认识到的是，MSS等信息往往只有在消息建立连接的几个请求中会指出，之后不会再在 Option 段中指出。在消息建立连接后，**ACK标记位都会默认为1，使能Ack的值为可靠。**

考察第三次握手的TCP段内容，如下所示：

```
Transmission Control Protocol, Src Port: 35201, Dst Port: 3000, Seq: 1, Ack: 1, Len: 0
Source Port: 35201
Destination Port: 3000
[Stream index: 6]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 1 (relative sequence number)
Sequence Number (raw): 2618481908
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 976454347
1000 .... = Header Length: 32 bytes (8)
Flags: 0x010 (ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    ....0.... .... = Congestion Window Reduced: Not set
    .... .0. .... = ECN-Echo: Not set
    .... ..0.... = Urgent: Not set
    .... ..1.... = Acknowledgment: Set
    .... ..0.... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
    .... .... ..0 = Fin: Not set
    [TCP Flags: .....A....]
Window: 1022
[Calculated window size: 261632]
[Window size scaling factor: 256]
Checksum: 0x42db [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
```

可以在Wireshark栏目中清晰地看到：

1. 客户端端的Seq此时加1，记录连接请求，发现与服务器端的Ack相同，标志传输正常。
2. 客户端的Ack是基于服务端端传入的Seq+1，同样为期望下一次收到的消息的数据的开始序列号，保证可靠字节流的传输。
3. 之后，ACK标志位会一直启用，原因在前面已说明。
4. 可以看到，此时Win的值发生改变，第三次握手中，客户端也完成自身资源的分配。

在实际意义上，我们可以认识到，若第三次握手成功，客户端和服务端都能确认双方的收发功能正常，因此可以建立稳定的连接，确保了TCP连接的正确性。

2、基于HTTP请求的数据收发过程

可以看到，基于状态码返回的不同，我们的抓包结果中有三类Get请求和接收的过程，我们以第一轮请求和响应过程为主要过程，在最后补充说明特殊的情况：

HTTP	590 GET / HTTP/1.1
TCP	56 3000 → 35201 [ACK] Seq=1 Ack=535 Win=2616576 Len=0 TSval=6697707 TSecr=6697707
HTTP	1054 HTTP/1.1 200 OK (text/html)
TCP	56 35201 → 3000 [ACK] Seq=535 Ack=999 Win=2616576 Len=0 TSval=6697708 TSecr=6697708
HTTP	526 GET /static/style1.css HTTP/1.1
TCP	56 3000 → 35201 [ACK] Seq=999 Ack=1005 Win=2616064 Len=0 TSval=6697818 TSecr=6697818
HTTP	1037 HTTP/1.1 200 OK (text/css)
TCP	56 35201 → 3000 [ACK] Seq=1005 Ack=1980 Win=2615552 Len=0 TSval=6697819 TSecr=6697819
HTTP	531 GET /static/head.jpg HTTP/1.1
TCP	56 3000 → 35201 [ACK] Seq=1980 Ack=1480 Win=2615552 Len=0 TSval=6697819 TSecr=6697819
HTTP	29297 HTTP/1.1 200 OK (JPEG JFIF image)
TCP	56 35201 → 3000 [ACK] Seq=1480 Ack=31221 Win=2586368 Len=0 TSval=6697820 TSecr=6697820
HTTP	603 GET /static/BGM.wav HTTP/1.1
TCP	56 3000 → 35201 [ACK] Seq=31221 Ack=2027 Win=2615040 Len=0 TSval=6697832 TSecr=6697832
TCP	65473 3000 → 35201 [ACK] Seq=31221 Ack=2027 Win=2615040 Len=65417 TSval=6697833 TSecr=6697833
TCP	532 3000 → 35201 [PSH, ACK] Seq=96638 Ack=2027 Win=2615040 Len=476 TSval=6697833 TSecr=6697833
TCP	56 35201 → 3000 [ACK] Seq=2027 Ack=97114 Win=2616576 Len=0 TSval=6697833 TSecr=6697833
TCP	65473 3000 → 35201 [ACK] Seq=97114 Ack=2027 Win=2615040 Len=65417 TSval=6697833 TSecr=6697833
TCP	175 3000 → 35201 [PSH, ACK] Seq=162531 Ack=2027 Win=2615040 Len=119 TSval=6697833 TSecr=6697833
TCP	56 35201 → 3000 [ACK] Seq=2027 Ack=162650 Win=2616576 Len=0 TSval=6697833 TSecr=6697833
HTTP	11114 HTTP/1.1 206 Partial Content (audio/wav)
TCP	56 35201 → 3000 [ACK] Seq=2027 Ack=173708 Win=2605568 Len=0 TSval=6697833 TSecr=6697833
HTTP	527 GET /favicon.ico HTTP/1.1
TCP	56 3000 → 35201 [ACK] Seq=173708 Ack=2498 Win=2614528 Len=0 TSval=6697849 TSecr=6697849
HTTP	478 HTTP/1.1 404 Not Found (text/html)

(1) Get请求的发送和接收:

我们首先以第一轮请求和响应过程为例，看看最常规情况下的请求响应过程是如何进行的：

590 GET / HTTP/1.1
56 3000 → 35201 [ACK] Seq=1 Ack=535 Win=2616576 Len=0 TSval=6697707 TSecr=6697707
1054 HTTP/1.1 200 OK (text/html)
56 35201 → 3000 [ACK] Seq=535 Ack=999 Win=2616576 Len=0 TSval=6697708 TSecr=6697708

可以看到，最一开始，客户端向服务器端发送了一个get请求，这里的请求头中的URL为/，在我们Node.js的重定向下即为我们的html文件。

考察请求报文内容，可以看到如下信息：

```
Hypertext Transfer Protocol
> GET / HTTP/1.1\r\n
Host: 127.0.0.1:3000\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/119.0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8\r\n
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2\r\n
Accept-Encoding: gzip, deflate, br\r\n
Connection: keep-alive\r\n
Upgrade-Insecure-Requests: 1\r\n
Sec-Fetch-Dest: document\r\n
Sec-Fetch-Mode: navigate\r\n
Sec-Fetch-Site: none\r\n
Sec-Fetch-User: ?1\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache\r\n
```

可以看到许多有用的信息，包含User-Agent中的操作系统和浏览器的版本、Accept中规定的返回响应信息的类型等等，大部分属性在先前已经解释过，这里我们主要看影响第四部分TCP连接效果的两个属性。

- **Connection:** 这里，浏览器申请了一个长连接的TCP连接，若服务端也回应长连接，则会进行长连接操作。
- **Cache-Control:** 这个属性是控制缓存是否启用，由于不禁用缓存会造成304响应（即本地已有缓存），结果不理想，因此启用了浏览器的开发者模式，在其中禁用缓存，导致这个结果为no-cache。

随后，服务器端成功接收到了该请求信息，并向客户端发送一个很小的TCP传输，标记位为ACK，表示自己已经收到了该请求。

同时，我们还需要注意TCP报文段内的信息，可以看到，这里GET请求启用了PSH位和ACK位，表明不等待缓存填满就发送GET请求，以加快访问的速度。这时候，TCP段长度为534，也就是客户端发送了534个字节的数据，则序列号需要增加534，在下一次客户端发送信息中即可以看到

```
Transmission Control Protocol, Src Port: 35201, Dst Port: 3000, Seq: 1, Ack: 1, Len: 534
  Source Port: 35201
  Destination Port: 3000
  [Stream index: 6]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 534]
  Sequence Number: 1      (relative sequence number)
  Sequence Number (raw): 2618481908
  [Next Sequence Number: 535      (relative sequence number)]
  Acknowledgment Number: 1      (relative ack number)
  Acknowledgment number (raw): 976454347
    1000 .... = Header Length: 32 bytes (8)
  Flags: 0x018 (PSH, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0.... .... = Congestion Window Reduced: Not set
    .... .0... .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    ..... 1... = Push: Set
    ..... .0.. = Reset: Not set
    ..... ..0. = Syn: Not set
    ..... .... .0 = Fin: Not set
  [TCP Flags: .....AP....]
```

考察服务器端收到GET请求的确认接收信息，可以看到此时服务器端的确认号为535，即对应了我们先前的推断，下一次客户端信息发送的序列号应当从535开始。

```
Sequence Number: 1      (relative sequence number)
Sequence Number (raw): 976454347
[Next Sequence Number: 1      (relative sequence number)]
Acknowledgment Number: 535      (relative ack number)
```

(2) 响应信息的发送和接收:

由于html文件不大，响应报文本身的数据段就可以装载下该文件，就不会再用更多的TCP传输处理。这里，服务器端通过简单的一次HTTP响应完成了html文件的传输，考察其HTTP响应头如下：

```
Hypertext Transfer Protocol
> HTTP/1.1 200 OK\r\n
X-Powered-By: Express\r\n
Accept-Ranges: bytes\r\n
Cache-Control: public, max-age=0\r\n
Last-Modified: Tue, 31 Oct 2023 15:31:16 GMT\r\n
ETag: W/"2aa-18b865c4709"\r\n
Content-Type: text/html; charset=UTF-8\r\n
> Content-Length: 682\r\n
Date: Thu, 02 Nov 2023 12:34:55 GMT\r\n
Connection: keep-alive\r\n
Keep-Alive: timeout=5\r\n
\r\n
[HTTP response 1/5]
[Time since request: 0.001290000 seconds]
[Request in frame: 422]
[Next request in frame: 466]
[Next response in frame: 476]
[Request URI: http://127.0.0.1:3000/]
File Data: 682 bytes
Line-based text data: text/html (27 lines)
\n
<html>\n
```

可以看到，这里响应头返回了一些服务器端的信息，因为我们这是在node.js开发环境下进行，包含了其技术支持Express的信息；还能够看到ETag等用于缓存处理的标记、Content相关的标记文本类型为text/html等等属性。

这里，我们还是只看影响我们的TCP连接的一些属性：

- **Connection:** 可以看到，服务器也会返回自己支持的连接方式，只有二者都支持长连接，才会进行长连接操作。
- **Cache-Control:** 类似的，服务器也会返回自己是否支持缓存，这里，服务器是支持缓存的，但是我们在客户端关闭了缓存的启用，因此不会启用缓存服务
- **Keep-Alive:** 这是服务器端对长连接的约束，用来加速服务器端的处理，当长连接时间过长的时候，会基于这个属性来关闭长连接。
- **File Data:** 获取到的文件的大小，可以看到这个大小远远小于我们的WSS，所以不会进行分段传输。

随后，客户端成功接收到了该请求信息，并向服务器端发送一个很小的TCP传输，标记位为ACK，表示自己已经收到了该请求。

我们再次考察响应信息的TCP段中属性，可以看到此时服务器发送了998字节的信息，Seq需要增大998个字。在下一次服务区发送信息时可以看到。此时响应报文同样是PSH、ACK同时启用。

```
Transmission Control Protocol, Src Port: 3000, Dst Port: 35201, Seq: 1, Ack: 535, Len: 998
  Source Port: 3000
  Destination Port: 35201
  [Stream index: 6]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 998]
  Sequence Number: 1      (relative sequence number)
  Sequence Number (raw): 976454347
  [Next Sequence Number: 999      (relative sequence number)]
  Acknowledgment Number: 535      (relative ack number)
  Acknowledgment number (raw): 2618482442
  1000 .... = Header Length: 32 bytes (8)
> Flags: 0x018 (PSH, ACK)
```

我们考察客户端回应服务器端表明自己已经收到html的信息的TCP段，可以看到Seq的值为先前的535，而Ack的值变为服务器端理应增大的Seq的值，即999，应证了可靠字节流的传输。

(3)大文件的传输

前面我们提到，当文件过大的时候，http响应报文无法放下这么多的数据（即超过了MSS），这时候就需要进行分段传输，而我们抓包结果中音频段的数据就需要进行分段传输，我们来看看这部分是怎么处理的。

这里，我们不再重复分析TCP段的变化和Get及响应的过程，其处理和先前是相同的，我们看看有哪些地方发生了不同

首先先看看GET请求中有什么不同，我们注意到，这里新增了一个从未出现的属性，即Range，它的作用是获取一定范围的数据，这里规定是0-，也就是全部获取。

```
> GET /static/BGM.wav HTTP/1.1\r\n
  Host: 127.0.0.1:3000\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/109.0
  Accept: audio/webm, audio/ogg, audio/wav, audio/*;q=0.9, application/ogg; q=0.7, video/*
  Accept-Language: zh-CN, zh;q=0.8, zh-TW;q=0.7, zh-HK;q=0.5, en-US;q=0.3, en;q=0.2\r\n
  Range: bytes=0-\r\n
  Connection: keep-alive\r\n
```

随后，我们可以看到，在GET请求发送后，有：

1. 首先服务器回复了确认的信息。
2. 然后服务器开始向客户端分段传输数据，每次传输结束时，会通过一次标记PSH的立即传输结束这次传输，然后客户端会输出确认信息。

3. 中间部分的大量数据都由若干个小段分开，其中segment of a reassembled PDU信息表示这是整个数据的一段信息。

4. 最后由HTTP响应返回最后的一段信息，结束传输。

```
603 GET /static/BGM.wav HTTP/1.1
56 3000 → 35201 [ACK] Seq=31221 Ack=2027 Win=2615040 Len=0 TSval=6697832 TSecr=6697832
65473 3000 → 35201 [ACK] Seq=31221 Ack=2027 Win=2615040 Len=65417 TSval=6697833 TSecr=6697832 [TCP segment of a reassembled PDU]
532 3000 → 35201 [PSH, ACK] Seq=96638 Ack=2027 Win=2615040 Len=476 TSval=6697833 TSecr=6697832 [TCP segment of a reassembled PDU]
56 35201 → 3000 [ACK] Seq=2027 Ack=97114 Win=2616576 Len=0 TSval=6697833 TSecr=6697833
65473 3000 → 35201 [ACK] Seq=97114 Ack=2027 Win=2615040 Len=65417 TSval=6697833 TSecr=6697833 [TCP segment of a reassembled PDU]
175 3000 → 35201 [PSH, ACK] Seq=162531 Ack=2027 Win=2615040 Len=119 TSval=6697833 TSecr=6697833 [TCP segment of a reassembled PDU]
56 35201 → 3000 [ACK] Seq=2027 Ack=162650 Win=2616576 Len=0 TSval=6697833 TSecr=6697833
11114 HTTP/1.1 206 Partial Content (audio/wav)
```

对于HTTP响应报文，我们首先可以看到它是一个206的响应类型，表明它收到了部分数据（尽管它可能确实是全部的数据了），这是由于先前GET请求的range属性造成的。

随后，在响应报文的属性中，我们可以看到Content-length，这是指明我们获取到的数据范围，若没有获取够（如超大的视频文件），很可能在后续的加载过程中继续获取。

Content-Type: audio/wav\r\n

Content-Range: bytes 0-142129/142130\r\n

Content-Length: 142130\r\n

综上，即为整个GET请求获取数据的过程，一些特殊情况我们放到第四部分的思考再说。

3、“四次挥手”的断开连接过程

四次挥手的过程本质是与三次握手类似的，不同的是，四次挥手需要保证数据传输的稳定性，即要完成数据的传输才能结束连接，因此，需要在第二次挥手的时候等待对方的数据传输完，即增加了一次挥手。

在我们的实验当中，由于响应头中设置的长连接时限，超出这个时限服务器端就会主动断开连接，这里，实际上是服务器主动发出断开连接的过程，但是我没有找到服务器主动发出的图，于是还是以客户端主动断开的图为例

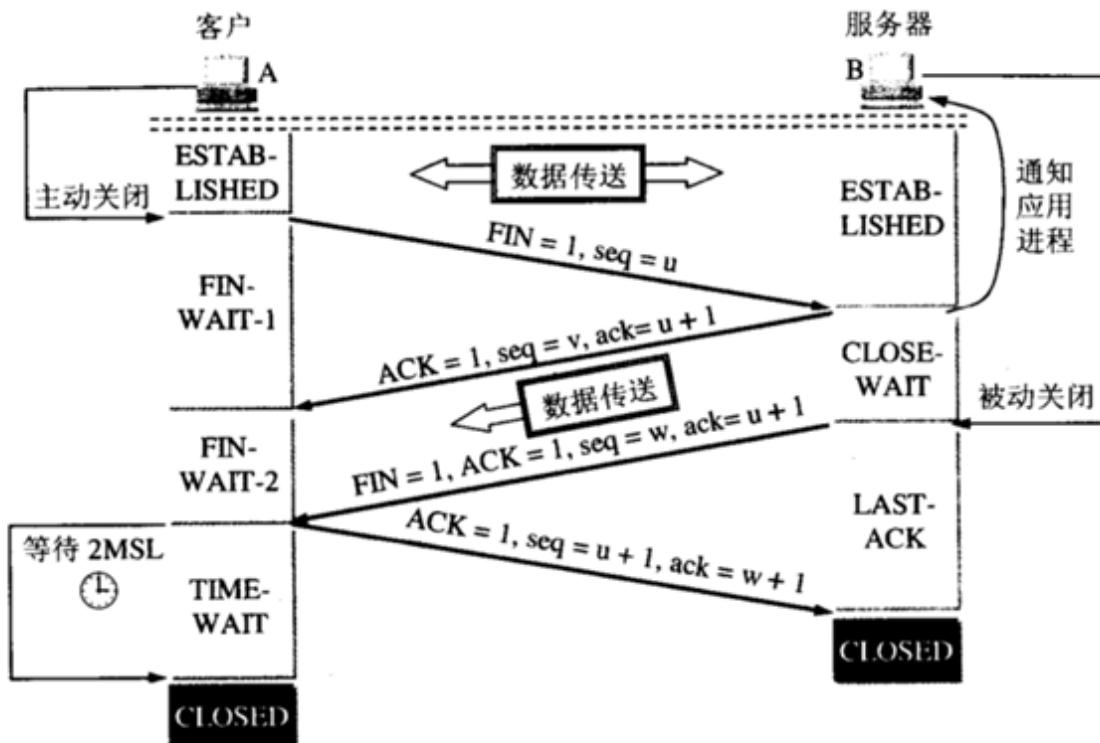


图 5-32 TCP 连接释放的过程

如下图所示，为我们实验中的四次挥手的抓包结果：

```
56 3000 → 35201 [FIN, ACK] Seq=174130 Ack=2498 Win=2614528 Len=0 TSval=6702863 TSecr=6697850
56 35201 → 3000 [ACK] Seq=2498 Ack=174131 Win=2605056 Len=0 TSval=6702863 TSecr=6702863
56 35201 → 3000 [FIN, ACK] Seq=2498 Ack=174131 Win=2605056 Len=0 TSval=6702863 TSecr=6702863
56 3000 → 35201 [ACK] Seq=174131 Ack=2499 Win=2614528 Len=0 TSval=6702863 TSecr=6702863
```

(1) 第一次挥手:

第一次挥手是指，服务端打算关闭连接时，向其 TCP 发送连接释放报文段，并终止发送数据，主动关闭TCP 连接，该报文段有：

1. 终止位 FIN 置1，ACK位仍然置1。
2. 序号 seq=u，它等于前面已传送过的数据的最后一个字节的序号加1 FIN 报文段即使不携带数据，也消耗掉一个序号。
3. 这时，TCP 客户进程进入 FIN-WAIT-1(终止等待1)状态。

TCP 是全双工的，即可以想象为一条TCP 连接上有两条数据通路发送 FIN 的一端不能再发送数据，即关闭了其中一条数据通路，但对方还可以发送数据。

考察我们的第一次挥手的报文，可以看到此时的报文内容确实如上述所示。

```
Transmission Control Protocol, Src Port: 3000, Dst Port: 35201, Seq: 174130, Ack: 2498, Len: 0
Source Port: 3000
Destination Port: 35201
[Stream index: 6]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 174130      (relative sequence number)
Sequence Number (raw): 976628476
[Next Sequence Number: 174131      (relative sequence number)]
Acknowledgment Number: 2498      (relative ack number)
Acknowledgment number (raw): 2618484405
1000 .... = Header Length: 32 bytes (8)
> Flags: 0x011 (FIN, ACK)
Window: 10213
```

通过这一次挥手，客户端了解到服务器想要关闭连接。

(2) 第二次挥手:

客户端收到连接释放报文段后即发出确认：

1. 确认号 ack=u+1，序号 seq=v，等于它前面已传送过的数据的最后一个字节的序号加1。
2. WAIT (关闭等待)然后服务器进入 CLOSE状态。
3. 此时，从服务器到客户端这方向的连接就释放了，TCP 连接处于半关闭状态。但客户端若发送数据，服务端仍要接收，即从客户端到服务端这个方向的连接并未关闭。

这里，在wireshark中只能够看到客户端回复的确认信息，因此不做过多的分析。通过这一次挥手，服务器了解到客户端收到关闭连接的信息，但还需要等待客户端传递完自身的数据。

(3) 第三次挥手:

第三次挥手指的是，若客户端已经没有要向服务器发送的数据，就通知TCP 释放连接，此时，有：

1. 发出FIN为1的连接释放报文段。
2. 设该报文段的序号为 w(在半关闭状态客户端可能又发送了一些数据)，还须重复上次已发送的确认号 ack=u+1。
3. 这时客户端进入 LASTACK (最后确认)状态。

考察wireshark中捕获的结果，发现与分析一致。

```

Transmission Control Protocol, Src Port: 35201, Dst Port: 3000, Seq: 2498, Ack: 174131, Len: 0
Source Port: 35201
Destination Port: 3000
[Stream index: 6]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 2498      (relative sequence number)
Sequence Number (raw): 2618484405
[Next Sequence Number: 2499      (relative sequence number)]
Acknowledgment Number: 174131      (relative ack number)
Acknowledgment number (raw): 976628477
1000 .... = Header Length: 32 bytes (8)
> Flags: 0x011 (FIN, ACK)

```

通过这一次挥手，服务器了解到客户端做好了关闭连接的准备，即客户端可以进入关闭状态，只需要等待服务器传递一个ACK即可。

(4) 第四次挥手：

第四次挥手指的是，服务器收到连接释放报文段后，必须发出确认。完成以下操作：

1. 把确认报文段中的确认位 ACK置1，确认号 ack=w+1，序号 seq=u+1。
2. 此时 TCP 连接还未释放，必须经过时间等待计时器设置的时间 2MSL（最长报文段寿命）后，客户机才进入 CLOSED(连接关闭)状态。

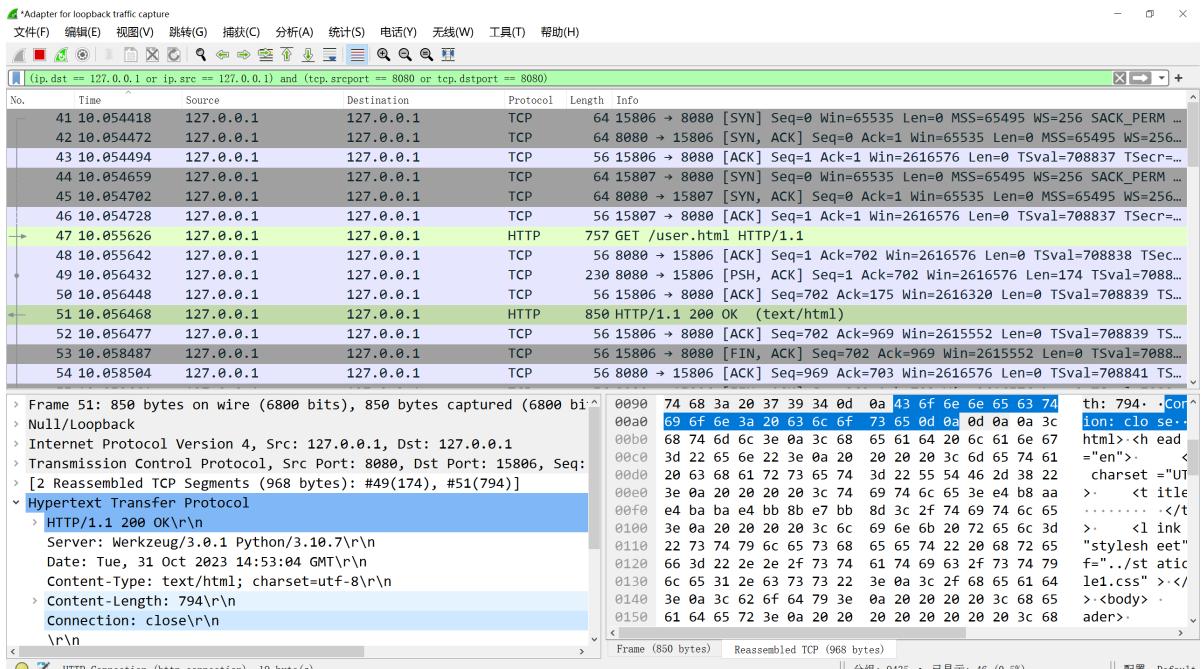
综上所述的四次挥手，完成了我们整个TCP连接释放的过程。

四、思考：不同服务器、浏览器下的优化：

(一) Flask的短连接现象：

事实上，在最初进行抓包实验时，我就使用了Flask进行实验，但正如先前所说，Flask部署的开发服务器会进行短连接，而协议版本确实是HTTP1.1，这是为什么呢？

我们考察Flask响应头，发现在Flask开发服务器中，默认的设置会把Connection属性设置为关闭，这是由于Flask使用的WSGI包中更改了默认设置，关闭了长连接。



Django中可以更改响应头的属性信息，Flask中可能也提供了相应修改的接口，我们可以通过重新设置长连接实现修复。

(二) 浏览器的缓存优化:

若我们用Node.js、Django等框架，能够正常进行长连接。

但当我们使用除了IE之外的浏览器的时候，会发生如下图所示的问题，也就是不再出现正常的数据传输过程，每个响应头的响应码都是304 Not Modified。

No.	Time	Source	Destination	Protocol	Length	Info
5.049571	127.0.0.1	127.0.0.1	TCP	64	13486 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSeq=4336951 TSecr=0	
5.049624	127.0.0.1	127.0.0.1	TCP	64	3000 → 13486 [SYN, ACK] Seq=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSeq=4336952 TSecr=4336951	
5.049647	127.0.0.1	127.0.0.1	TCP	56	13486 → 3000 [ACK] Seq=1 Ack=1 Win=2616532 Len=0 TSeq=4336952 TSecr=4336952	
5.049777	127.0.0.1	127.0.0.1	TCP	64	13487 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSeq=4336952 TSecr=0	
5.049886	127.0.0.1	127.0.0.1	TCP	64	3000 → 13487 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSeq=4336952 TSecr=4336952	
5.049819	127.0.0.1	127.0.0.1	TCP	56	13487 → 3000 [ACK] Seq=1 Ack=1 Win=2616576 Len=0 TSeq=4336952 TSecr=4336952	
5.044929	127.0.0.1	127.0.0.1	HTTP	808	GET / HTTP/1.1	
5.044955	127.0.0.1	127.0.0.1	TCP	56	3000 → 13486 [ACK] Seq=1 Ack=753 Win=2616576 Len=0 TSeq=4336956 TSecr=4336956	
5.046412	127.0.0.1	127.0.0.1	HTTP	321	HTTP/1.1 304 Not Modified	
5.046435	127.0.0.1	127.0.0.1	TCP	56	13486 → 3000 [ACK] Seq=753 Ack=266 Win=2616576 Len=0 TSeq=4336957 TSecr=4336957	
5.079301	127.0.0.1	127.0.0.1	HTTP	705	GET /static/style1.css HTTP/1.1	
5.079349	127.0.0.1	127.0.0.1	TCP	56	3000 → 13486 [ACK] Seq=266 Ack=1482 Win=2615888 Len=0 TSeq=4336990 TSecr=4336990	
5.088649	127.0.0.1	127.0.0.1	HTTP	321	HTTP/1.1 304 Not Modified	
5.088688	127.0.0.1	127.0.0.1	TCP	56	13486 → 3000 [ACK] Seq=1482 Ack=513 Win=2616320 Len=0 TSeq=4336992 TSecr=4336992	
5.081424	127.0.0.1	127.0.0.1	HTTP	739	GET /static/head.jpg HTTP/1.1	
5.081453	127.0.0.1	127.0.0.1	TCP	56	3000 → 13486 [ACK] Seq=513 Ack=2085 Win=2615296 Len=0 TSeq=4336992 TSecr=4336992	
5.082284	127.0.0.1	127.0.0.1	HTTP	322	HTTP/1.1 304 Not Modified	
5.082382	127.0.0.1	127.0.0.1	TCP	56	13486 → 3000 [ACK] Seq=2085 Ack=797 Win=2616064 Len=0 TSeq=4336993 TSecr=4336993	
5.172668	127.0.0.1	127.0.0.1	HTTP	714	GET /static/BGM.wav HTTP/1.1	
5.172665	127.0.0.1	127.0.0.1	TCP	56	3000 → 13486 [ACK] Seq=797 Ack=2743 Win=2614528 Len=0 TSeq=4337084 TSecr=4337084	
5.174177	127.0.0.1	127.0.0.1	HTTP	323	HTTP/1.1 304 Not Modified	
5.174246	127.0.0.1	127.0.0.1	TCP	56	13486 → 3000 [ACK] Seq=2743 Ack=1064 Win=2615808 Len=0 TSeq=4337085 TSecr=4337085	
16.185433	127.0.0.1	127.0.0.1	TCP	56	3000 → 13486 [FIN, ACK] Seq=1064 Ack=743 Win=2614528 Len=0 TSeq=4342096 TSecr=4342096	
16.185433	127.0.0.1	127.0.0.1	TCP	56	13486 → 3000 [ACK] Seq=2743 Ack=1065 Win=2615808 Len=0 TSeq=4342096 TSecr=4342096	
49.498651	127.0.0.1	127.0.0.1	TCP	56	13487 → 3000 [FIN, ACK] Seq=1 Ack=1 Win=2616576 Len=0 TSeq=4372409 TSecr=4336952	
49.498671	127.0.0.1	127.0.0.1	TCP	56	3000 → 13487 [ACK] Seq=1 Ack=2 Win=2616576 Len=0 TSeq=4372410 TSecr=4372409	
49.498699	127.0.0.1	127.0.0.1	TCP	56	13486 → 3000 [FIN, ACK] Seq=2743 Ack=1065 Win=2615888 Len=0 TSeq=4372410 TSecr=4342096	
49.498914	127.0.0.1	127.0.0.1	TCP	56	3000 → 13486 [ACK] Seq=1065 Ack=2744 Win=2614528 Len=0 TSeq=4372410 TSecr=4372410	
49.498914	127.0.0.1	127.0.0.1	TCP	56	3000 → 13487 [FIN, ACK] Seq=1 Ack=2 Win=2616576 Len=0 TSeq=4372410 TSecr=4372409	
49.498929	127.0.0.1	127.0.0.1	TCP	56	13487 → 3000 [ACK] Seq=2 Ack=2 Win=2616576 Len=0 TSeq=4372410 TSecr=4372410	

这是事实上是浏览器对访问作出的优化，在获取到服务器端资源后，会存储到本地的高速缓存中，当需要再次使用的时候，只需要向服务器端确认这个值是否为最新就可以实现，这个确认方式利用了Etag属性来实现。

因此，我们只需要F12打开开发者模式，禁用缓存，就可以避免这种情况的发生。

(三) HTTP1.1的并发TCP连接现象：

此外，在这时候还会出现存在两个TCP连接的情况，可以看到，在下图中，获取html文件、css文件是通过14252端口建立的TCP连接实现，而获取BGM、图片等信息，是通过14273端口建立的TCP连接实现，这是为什么呢？

No.	Time	Source	Destination	Protocol	Length	Info
3.359444	127.0.0.1	127.0.0.1	TCP	64	14273 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSeq=7535730 TSecr=0	
3.359534	127.0.0.1	127.0.0.1	TCP	64	3000 → 14273 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSeq=7535730 TSecr=7535730	
3.359534	127.0.0.1	127.0.0.1	HTTP	765	GET / HTTP/1.1	
3.351465	127.0.0.1	127.0.0.1	TCP	56	14273 → 3000 [ACK] Seq=1 Ack=1 Win=10221 Len=0 TSeq=7535731 TSecr=7535731	
3.352884	127.0.0.1	127.0.0.1	HTTP	1854	HTTP/1.1 200 OK (text/html)	
3.352982	127.0.0.1	127.0.0.1	TCP	56	14252 → 3000 [ACK] Seq=718 Ack=999 Win=10217 Len=0 TSeq=7535733 TSecr=7535733	
3.385944	127.0.0.1	127.0.0.1	HTTP	662	GET /static/style1.css HTTP/1.1	
3.385944	127.0.0.1	127.0.0.1	TCP	56	3000 → 14252 [ACK] Seq=1316 Ack=10219 Win=10219 Len=0 TSeq=7535766 TSecr=7535766	
3.386095	127.0.0.1	127.0.0.1	HTTP	695	GET /static/head.jpg HTTP/1.1	
3.386112	127.0.0.1	127.0.0.1	TCP	56	3000 → 14273 [ACK] Seq=1 Ack=640 Win=326400 Len=0 TSeq=7535766 TSecr=7535766	
3.388215	127.0.0.1	127.0.0.1	HTTP	1837	HTTP/1.1 200 OK (text/css)	
3.388263	127.0.0.1	127.0.0.1	TCP	56	14252 → 3000 [ACK] Seq=1316 Ack=1988 Win=10213 Len=0 TSeq=7535768 TSecr=7535768	
3.388731	127.0.0.1	127.0.0.1	HTTP	29297	HTTP/1.1 200 OK (JPEG/JIF image)	
3.388762	127.0.0.1	127.0.0.1	TCP	56	14273 → 3000 [ACK] Seq=640 Ack=29242 Win=2616576 Len=0 TSeq=7535769 TSecr=7535768	
3.412238	127.0.0.1	127.0.0.1	HTTP	663	GET /static/BGM.wav HTTP/1.1	
3.412288	127.0.0.1	127.0.0.1	TCP	56	3000 → 14273 [ACK] Seq=29242 Ack=1247 Win=325632 Len=0 TSeq=7535792 TSecr=7535792	
3.414542	127.0.0.1	127.0.0.1	TCP	65473	3000 → 14273 [ACK] Seq=29242 Ack=1247 Win=325632 Len=65417 TSeq=7535794 TSecr=7535792 [TCP segment of a reassembled PDU]	
3.414545	127.0.0.1	127.0.0.1	TCP	532	3000 → 14273 [PSH, ACK] Seq=946599 Ack=1247 Win=325632 Len=476 TSeq=7535794 TSecr=7535792 [TCP segment of a reassembled PDU]	
3.414594	127.0.0.1	127.0.0.1	TCP	56	14273 → 3000 [ACK] Seq=1247 Win=10213 Len=0 TSeq=7535794 TSecr=7535794	
3.414863	127.0.0.1	127.0.0.1	TCP	65473	3000 → 14273 [PSH, ACK] Seq=95135 Ack=1247 Win=325632 Len=65417 TSeq=7535795 TSecr=7535795 [TCP segment of a reassembled PDU]	
3.414872	127.0.0.1	127.0.0.1	TCP	175	3000 → 14273 [PSH, ACK] Seq=160552 Ack=1247 Win=325632 Len=119 TSeq=7535795 TSecr=7535795 [TCP segment of a reassembled PDU]	
3.414968	127.0.0.1	127.0.0.1	TCP	56	14273 → 3000 [ACK] Seq=1606712 Ack=2616320 Win=0 TSeq=7535795 TSecr=7535795	
3.415128	127.0.0.1	127.0.0.1	HTTP	11114	HTTP/1.1 200 Partial Content (audio/wav)	
3.415146	127.0.0.1	127.0.0.1	TCP	56	14273 → 3000 [ACK] Seq=1247 Ack=171729 Win=2605312 Len=0 TSeq=7535795 TSecr=7535795	
8.401582	127.0.0.1	127.0.0.1	TCP	56	3000 → 14252 [FIN, ACK] Seq=1316 Ack=10213 Len=0 TSeq=7540782 TSecr=7535768	
8.417668	127.0.0.1	127.0.0.1	TCP	56	14252 → 3000 [ACK] Seq=171729 Ack=1247 Win=325632 Len=0 TSeq=7540798 TSecr=7535795	
8.417783	127.0.0.1	127.0.0.1	TCP	56	14273 → 3000 [ACK] Seq=1247 Ack=171730 Win=2605312 Len=0 TSeq=7540798 TSecr=7540798	
9.204763	127.0.0.1	127.0.0.1	TCP	56	14252 → 3000 [FIN, ACK] Seq=1316 Ack=10821 Len=0 TSeq=7541585 TSecr=7540782	
9.204812	127.0.0.1	127.0.0.1	TCP	56	3000 → 14252 [ACK] Seq=1317 Ack=10219 Len=0 TSeq=7541585 TSecr=7541585	
9.204914	127.0.0.1	127.0.0.1	TCP	56	14273 → 3000 [FIN, ACK] Seq=1247 Ack=171730 Win=2605312 Len=0 TSeq=7541585 TSecr=7540798	

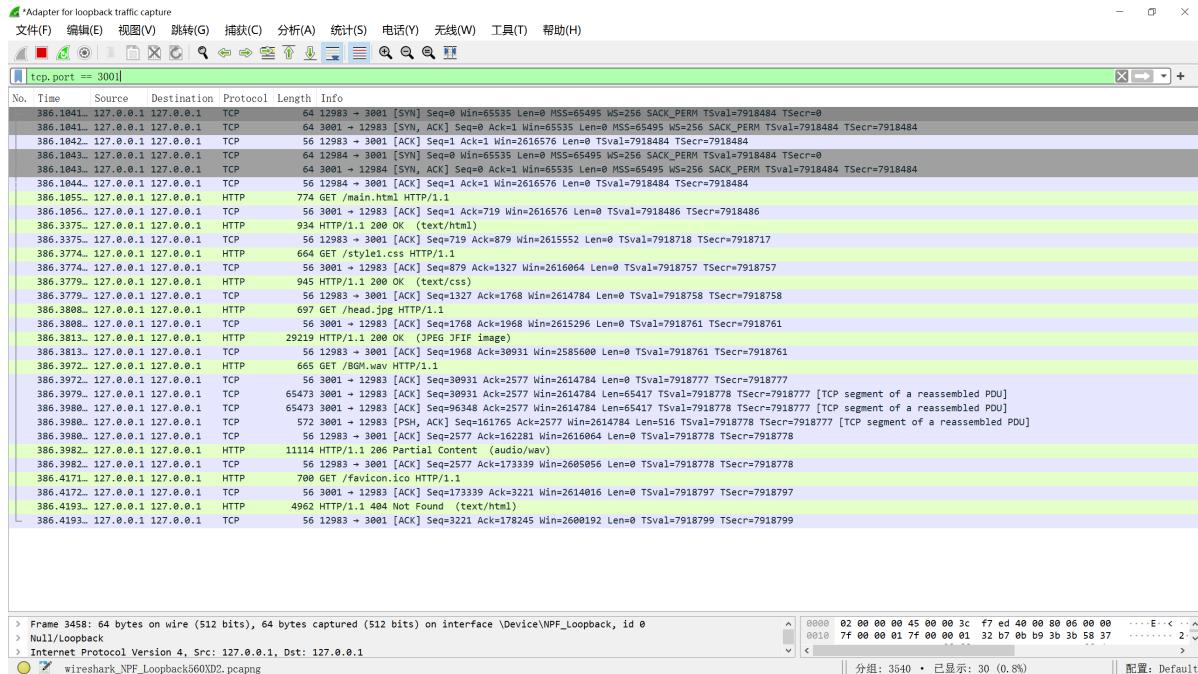
事实上，在HTTP1.1中，由于必须基于文本协议，进行问答有序的数据传输模式，先请求的必须先响应。

这造成了一种名为队头阻塞的问题，即前面的数据传递可能会卡死，因此，浏览器会选择建立多个TCP连接，通过不同TCP连接传送的请求没有响应顺序的要求。

而我们实验中遇到的这种情况，事实上就是多个TCP连接并发的一个现象，它是HTTP1.1解决队头阻塞的一种策略。

(四) IIS、NGINX的长连接挂起现象：

此外，我们在运用IIS、NGINX进行实验的时候，会出现迟迟不出现四次挥手，直到关闭浏览器后才出现挥手的现象，这是为什么呢？



我们看看关闭浏览器后的结果（这里也存在TCP连接的并发处理，有两个TCP连接），可以看到，此时TCP连接事实上由浏览器首先进行释放，但是出现RST这个严重报错，可能是强行关闭浏览器造成的。

```
56 12984 → 3001 [FIN, ACK] Seq=1 Ack=1 Win=2616576 Len=0 TSval=7977019 TSecr=7963491
56 3001 → 12984 [ACK] Seq=1 Ack=2 Win=2616576 Len=0 TSval=7977019 TSecr=7977019
56 12983 → 3001 [FIN, ACK] Seq=3221 Ack=178245 Win=2600192 Len=0 TSval=7977019 TSecr=7963802
56 3001 → 12983 [ACK] Seq=178245 Ack=3222 Win=2614016 Len=0 TSval=7977019 TSecr=7977019
44 3001 → 12984 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
56 3001 → 12983 [FIN, ACK] Seq=178245 Ack=3222 Win=2614016 Len=0 TSval=7977020 TSecr=7977019
56 12983 → 3001 [ACK] Seq=3222 Ack=178246 Win=2600192 Len=0 TSval=7977020 TSecr=7977020
```

考察服务器端的响应头，我们发现了其与Node.js灯开发服务器的不同，即服务器的响应头并没有Connect属性，也没有Keep-alive属性来限制长连接时间，因此，长连接事实上会一直挂起，直至客户端关闭连接。

```
Hypertext Transfer Protocol
> HTTP/1.1 200 OK\r\n
Content-Type: image/jpeg\r\n
Last-Modified: Tue, 31 Oct 2023 09:39:48 GMT\r\n
Accept-Ranges: bytes\r\n
ETag: "b0f17130debda1:0"\r\n
Server: Microsoft-IIS/10.0\r\n
Date: Fri, 03 Nov 2023 17:17:07 GMT\r\n
> Content-Length: 28936\r\n
\r\n
[HTTP response 3/5]
[Time since request: 0.000521000 seconds]
[Prev request in frame: 3477]
[Prev response in frame: 3479]
[Request in frame: 3481]
[Next request in frame: 3485]
[Next response in frame: 3491]
[Request URI: http://127.0.0.1:3001/BGM.wav]
File Data: 28936 bytes
```

我们在IIS中尝试为服务器端增加相应的响应头，解决了这个问题。

 **HTTP 响应标头**

使用此功能配置从 Web 服务器添加到响应中的 HTTP 标头。

名称	值	条目类型
本地		
Connection	keep-alive	本地
Keep-Alive	timeout =5	本地