



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

计算机网络实验报告

---

Lab2: 配置 Web 服务器, 编写简单页面, 分析交互过程

---

冯思程 2112213

年级: 2021 级

专业: 计算机科学与技术

指导教师: 吴英、文静静

2023 年 11 月 3 日

## 摘要

本次实验，根据要求利用 docker+nginx 搭建了 web 服务器，然后自己编写了 web 页面（包括图片文字音频和 css 文件），然后用 wireshark 工具进行抓包分析整个 TCP 连接的交互过程（三次握手等），并编写报告。

**关键字：**docker、nginx、TCP 连接、web 服务器、三次握手

## 目录

<b>一、 预备工作及实验环境</b>	<b>1</b>
(一) 实验要求与功能 . . . . .	1
(二) 实验环境与说明 . . . . .	1
<b>二、 实验过程</b>	<b>1</b>
(一) web 服务器搭建 . . . . .	1
1. docker 安装配置与 nginx 配置 . . . . .	2
2. web 页面设计 . . . . .	3
(二) wireshark 捕获分析 . . . . .	7
1. 三次握手 . . . . .	9
2. 通过 TCP 传输 HTTP 数据传输全过程 . . . . .	11
3. 长连接维持 . . . . .	15
4. TCP 连接断开 . . . . .	16
5. 浏览器的“并行”机制 . . . . .	19
6. http1.0 VS http1.1 . . . . .	19
<b>三、 总结与思考</b>	<b>19</b>

## 一、 预备工作及实验环境

### (一) 实验要求与功能

实验要求的基础功能与要求:

1. 搭建 Web 服务器 (自由选择系统), 并制作简单的 Web 页面, 包含简单文本信息 (至少包含专业、学号、姓名)、自己的 LOGO、自我介绍的音频信息。页面不要太复杂, 包含要求的基本信息即可。
2. 通过浏览器获取自己编写的 Web 页面, 使用 Wireshark 捕获浏览器与 Web 服务器的交互过程, 并进行简单的分析说明。
3. 使用 HTTP, 不要使用 HTTPS。
4. 提交实验报告。
5. Wireshark 捕获交互过程, 使用 Wireshark 过滤器使其**仅显示 HTTP 协议**, 提交捕获文件。

合理的自行扩展功能:

1. 为 web 页面编写了一个 css 文件, 美化外观, 同时可以进行捕获分析。

### (二) 实验环境与说明

具体的实验环境配置如下:

Windows 版本	vs code 版本	docker 版本	nginx 版本 (AMD64 架构)
windows11	1.82.0	24.0.6	1.25.3

表 1: 实验环境说明表

感谢老师与助教的审查批阅与指正, 辛苦!

## 二、 实验过程

### (一) web 服务器搭建

本次实验采用 docker+nginx 来完成这次实验, 下面简单介绍一下这两个工具:

1. **docker**: Docker 是一个开源的应用容器引擎, 允许开发者将应用及其依赖打包到一个可移植的容器中, 然后发布到任何流行的 Linux 机器或 Windows 机器上, 也可以实现虚拟化。在这次实验中我安装了 windows 版本的 docker 桌面工具来辅助我完成实验。
2. **nginx**: Nginx 是一个开源的高性能、高并发的 Web 服务器、反向代理服务器, 同时也提供了 IMAP/POP3/SMTP 服务。nginx 的工作原理是一个主线程什么也不做, 然后通过 fork 多个子进程来进行工作。在这次实验中, 我在 docker 中 pull 了一个 nginx 的 image, 并以此建立本次实验的容器。

## 1. docker 安装配置与 nginx 配置

这里首先需要去 Docker 官网安装 **Docker Desktop** 工具并注册账号，然后为其换一个镜像源，让其可以更快的 pull 下来一个 image。

然后在命令行中，使用命令 **docker pull nginx** 直接 pull 下来一个有最新版本的 nginx 的容器，然后查看 Docker Desktop 中的 image 界面结果如下：

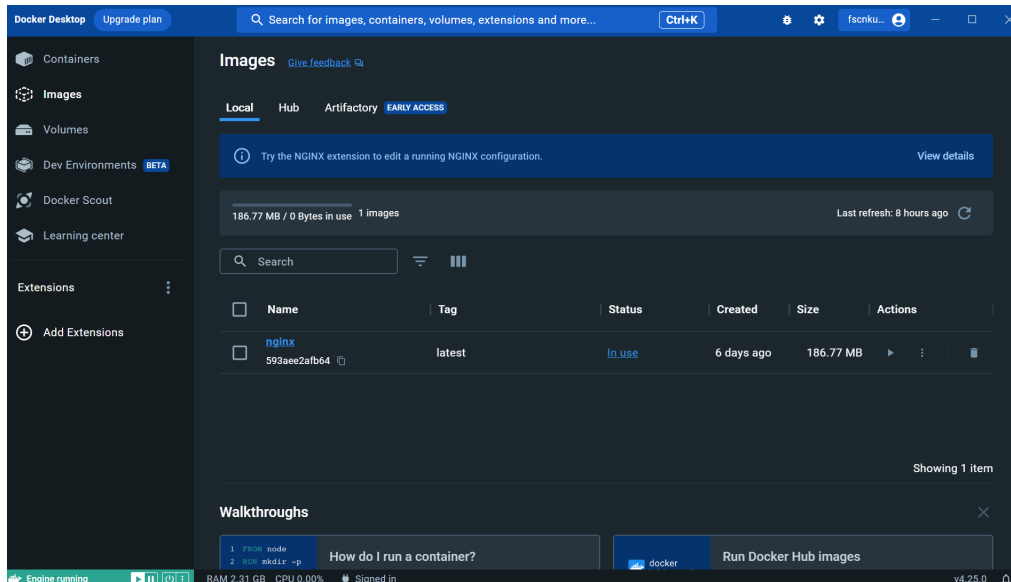
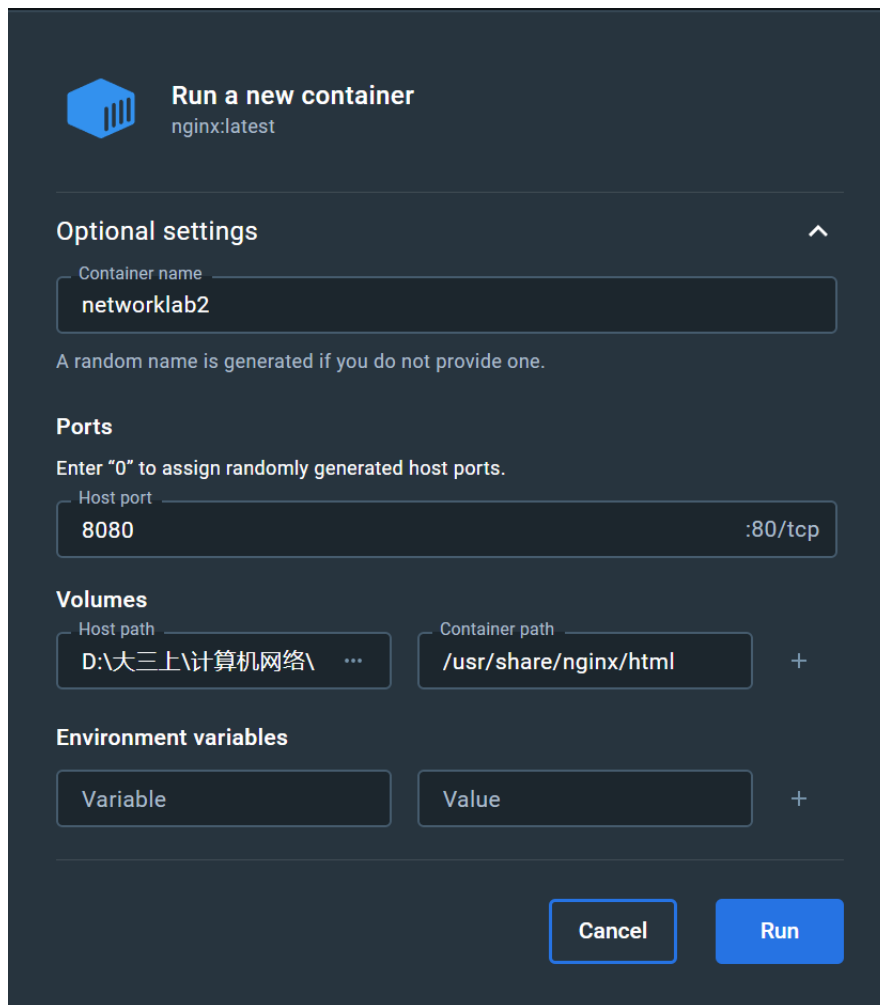


图 1: image 界面

现在我已经成功配置了一个可以运行的 image，而且其中的 nginx 可以帮助我构建 web 服务器，接下需要运行，这里需要设置一些来将本机的页面文件映射到容器中，从而让其可以运行我们自己设计的 web 页面，这里注意也需要将一些必要的文件也映射进去，例如 css 文件、mp3 文件等等，这些文件需要放在本机中的一个文件夹下，映射到容器的这个路径 (**/usr/share/nginx/html**) 下。同时需要进行端口映射，这里容器默认的是 80 端口，需要设置将本机的某个端口映射过去，这里我设置的是将本机的 8080 端口映射过去，整体设置如下：



**Run a new container**  
nginx:latest

**Optional settings**

Container name  
networklab2

A random name is generated if you do not provide one.

**Ports**  
Enter "0" to assign randomly generated host ports.  
Host port: 8080 :80/tcp

**Volumes**  
Host path: D:\大三上\计算机网络\ ... Container path: /usr/share/nginx/html +

**Environment variables**  
Variable Value +

Cancel Run

图 2: run setting

## 2. web 页面设计

这里这次的页面中有文字有音频有照片，同时我用 css 文件对页面进行了一定程度的美化，下面先展示我的 html 文件的代码：

页面 html

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>原神启动!!</title>
8   <link rel="stylesheet" href="styles.css">
9 </head>
10
11 <body>
12   <header>
13     <h1>Welcome To My Personal Homepage</h1>
```

```

14     <p>Here, you can learn more about me and my professional background.<
      /p>
15 </header>
16
17 <section id="profile">
18     <h2>PROFILE</h2>
19     
20     <h3>Major: Computer Science</h3>
21     <h4>Student ID: 2112213</h4>
22     <h5>Name: 冯思程</h5>
23     <p>    I am currently an undergraduate student at Nankai University ,
      majoring in CS and minoring in Actuarial Science</p>
24     <p>    I ' m looking to collaborate on ML & AI & Quant</p>
25     <audio controls>
26         <source src="intro.mp3" type="audio/mpeg">
27         Your browser does not support the audio element.
28     </audio>
29 </section>
30 <section id="contact">
31     <h2>CONTACT ME</h2>
32     <p>If you want to contact me, you can do so through the following
      methods:</p>
33     <ul>
34         <li>Email: fscdyx888@163.com</li>
35         <li>Tel: 18822623404</li>
36         <li>Wechat: stark888</li>
37     </ul>
38 </section>
39 <footer>
40     <p>版权所有 &copy; 2112213 冯思程 NKU CS</p>
41 </footer>
42 </body>
43
44 </html>

```

音频是 mp3 格式，是我用 python 第三方库自动合成的，用到的 python 代码如下：

#### 音频合成

```

1 from gtts import gTTS
2
3 text = "欢迎你来到我的主页，我叫冯思程，外号是来自NKU的无敌龙帝，我喜欢AI与数
      据结合，对数字比较敏感，我未来想从事的领域是量化"
4 tts = gTTS(text=text, lang="zh-CN")
5 tts.save("intro.mp3")

```

然后图片是截图下来的，将其格式保存为 jpg 图片。

最后编写一个 css 文件对页面进行美化，代码如下：

CSS

```
1  /* 全局样式 */
2  body {
3      font-family: Arial, sans-serif;
4      background-color: #f4f4f4;
5      margin: 0;
6      padding: 0;
7  }
8
9  header, section, footer {
10     width: 80%;
11     margin: 20px auto;
12     padding: 20px;
13     background-color: #fff;
14     border-radius: 5px;
15     box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
16 }
17
18 h1, h2, h3, h4, h5 {
19     color: #333;
20 }
21
22 p {
23     line-height: 1.6;
24     color: #666;
25 }
26
27 /* 导航样式 */
28 nav ul {
29     padding: 0;
30     list-style: none;
31 }
32
33 nav ul li {
34     display: inline;
35     margin-right: 10px;
36 }
37
38 nav ul li a {
39     text-decoration: none;
40     color: #333;
41     font-weight: bold;
42     padding: 5px 10px;
43     border: 1px solid #333;
44     border-radius: 5px;
45     transition: background-color 0.3s;
46 }
47
```

```
48 nav ul li a:hover {
49     background-color: #333;
50     color: #fff;
51 }
52
53 /* 图片样式 */
54 .profile-pic {
55     display: block;
56     width: 150px;
57     height: 150px;
58     margin: 20px auto;
59     border-radius: 50%;
60     box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
61 }
62
63 /* 视频和音频样式 */
64 video, audio {
65     display: block;
66     max-width: 100%;
67     margin: 20px 0;
68 }
69
70 /* 脚注样式 */
71 footer {
72     text-align: center;
73 }
```

将上述文件都存在我们之前提到的本机映射文件夹下，结果如下：



图 3: 映射文件夹内容

运行容器后，在本机访问网址 <http://localhost:8080/index.html>，结果如下：



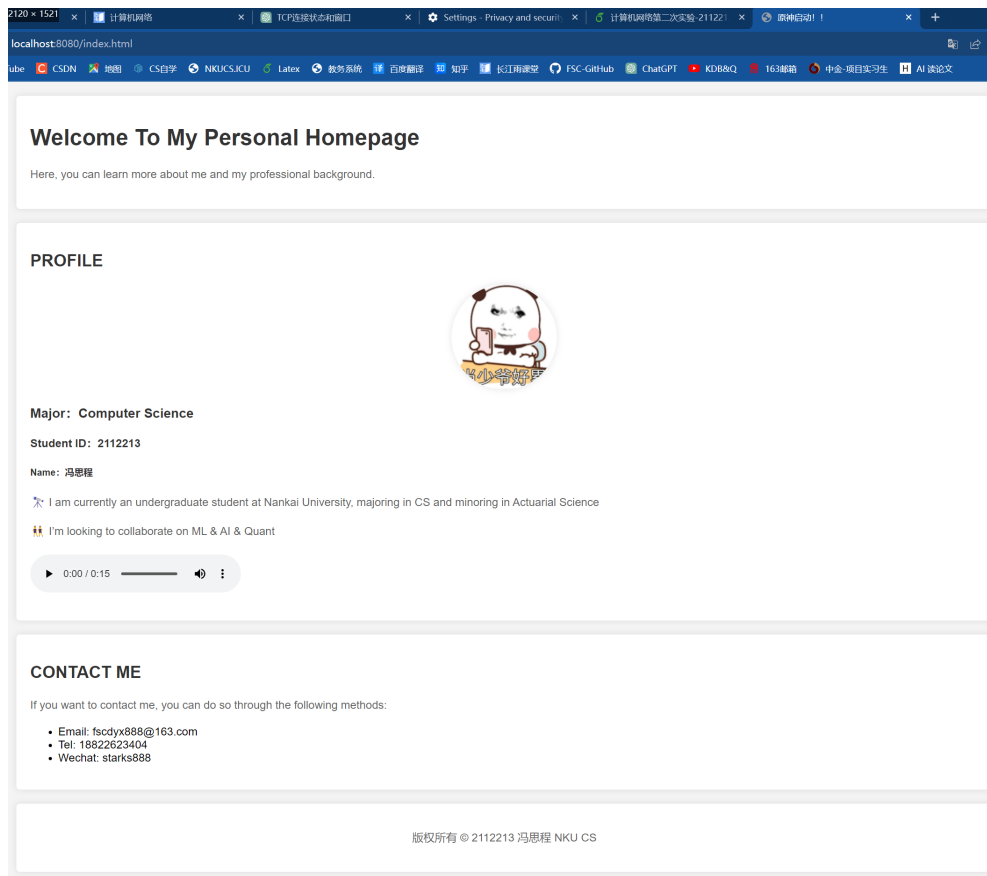


图 4: web 页面展示

经过点击测试，音频可以正确播放，页面实现正确无误。

下面进行 wireshark 捕获分析，具体的分析 tcp 连接建立到断开的全过程。

## (二) wireshark 捕获分析

这里捕获需要注意的一点就是，如果需要多次捕获的话，需要每次提前删掉缓存，或者直接在开发者工具中禁用缓存（如果不这样的话，会发现捕获的是 not modified，也就是浏览器会自动缓存加载页面所需要的内容，影响实验效果），如下：（下图展示我本次实验使用的 chrome 浏览器的缓存清除位置，只需要在 setting 中搜索 cache 即可找到。）

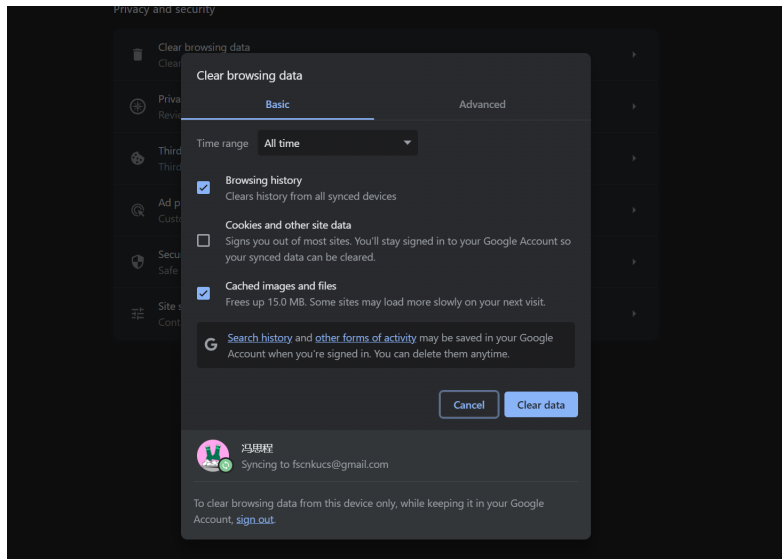


图 5: 缓存清除

然后我再来先展示一下，tcp 协议的整体交互过程流程图，但是我的实验中在断开的过程和标准的 tcp 协议流程有所区别，在后文会具体分析。流程图如下：

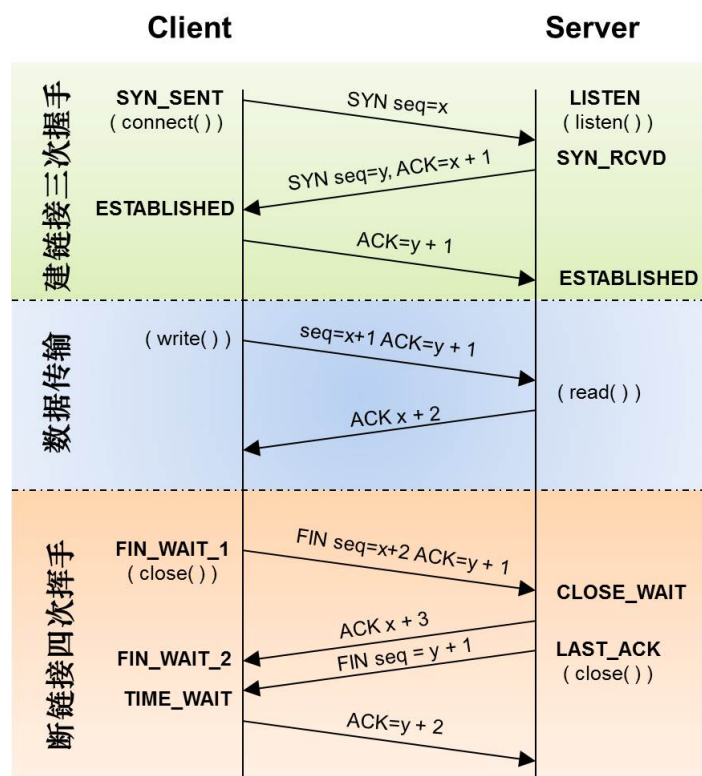


图 6: 交互流程

接下来我会利用 wireshark 工具对整个交互过程进行分析，这里选择 wireshark 中的对回环网卡进行监听捕获，访问网址为：<http://localhost:8080/index.html>。同时由于回环网卡中还有别的连接，所以这里需要设置过滤器以只查看我们想要查看的信息，这里筛选的条件我设置为：`tcp.port == 8080`，然后正式开始监听分析，首先开启服务器（即 run 设置的容器），然

后开启监听，然后打开浏览器访问网址进行监听捕获。下面先解释一下 TCP 连接状态标识：

### TCP 连接状态标识

1. SYN 表示建立连接
2. FIN 表示关闭连接
3. ACK 表示响应
4. PSH 表示有 DATA 数据传输
5. RST 表示连接重置

### 1. 三次握手

三次握手是建立一个 TCP 连接时，需要客户端和服务端总共发送 3 个包。三次握手的目的是连接服务器指定端口，建立连接，并同步连接双方的序列号和确认号，交换窗口大小信息。

609	32.124327	::1	::1	TCP	76	55133 → 8080	[SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
610	32.124389	::1	::1	TCP	76	8080 → 55133	[SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SA
611	32.124427	::1	::1	TCP	64	55133 → 8080	[ACK] Seq=1 Ack=1 Win=327168 Len=0
612	32.124648	::1	::1	HTTP	753	GET /index.html HTTP/1.1	
613	32.124680	::1	::1	TCP	64	8080 → 55133	[ACK] Seq=1 Ack=690 Win=2159872 Len=0
614	32.127065	::1	::1	TCP	76	55134 → 8080	[SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
615	32.127119	::1	::1	TCP	76	8080 → 55134	[SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SA
616	32.127150	::1	::1	TCP	64	55134 → 8080	[ACK] Seq=1 Ack=1 Win=327168 Len=0
617	32.132464	::1	::1	TCP	303	8080 → 55133	[PSH, ACK] Seq=1 Ack=690 Win=2159872 Len=239 [TCP segment
618	32.132511	::1	::1	TCP	64	55133 → 8080	[ACK] Seq=690 Ack=240 Win=326912 Len=0
619	32.132554	::1	::1	HTTP	1517	HTTP/1.1 200 OK (text/html)	

图 7: 三次握手

1. **第一次握手：**在输入了网址之后，客户端要主动和服务端建立连接，因此第一次握手是客户端向服务器发送了 TCP 请求，由于是回环测试，所以 IP 地址均为 127.0.0.1，但是端口号不同，客户端端口号为 55133，服务器端口号为我之前设置的 8080。

客户端将 TCP 报文标志位 SYN 置为 1（表示这条报文为建立连接的报文），随机产生一个序号值 seq=x（如图 x=0），保存在 TCP 首部的序列号字段里，指明客户端打算连接的服务器的端口，并将该数据包发送给服务器端，发送完毕后，客户端进入 SYN\_SENT（同步已发送状态），等待服务器端确认。TCP 规定，SYN 报文段（SYN=1 的报文段）不能携带数据，因此报文的数据段长度 Len 为 0。报文段的最大长度 MSS 为 65475，表示所能够接收到的报文段数据最大为 65475 个字节。WIN 字段表示接收端还能够接收的字节数为 65535，下一次发送的数据大小不能够超过这个数值。WS 表示窗口的大小为 8\*256。SACK\_PERM 表示允许选择确认重传机制。这里消耗了一个序列号。

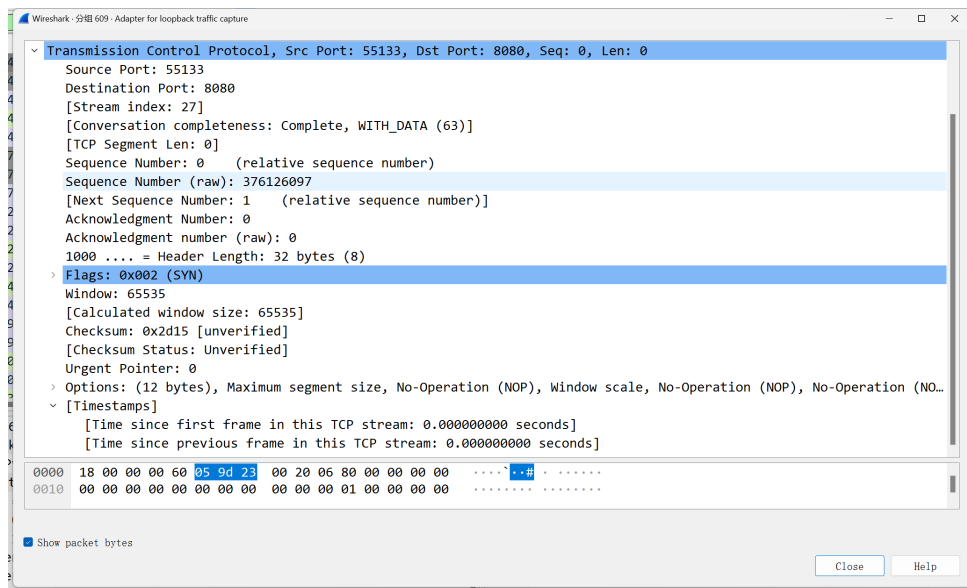


图 8: 第一次握手

2. **第二次握手**: 第二次握手是服务器接收到了客户端发起的连接请求后, 给浏览器的应答, 并在之后为该 TCP 分配缓存和变量。可见这条 TCP 报文是从 8080 端口发向 55133 端口的。服务器收到客户端的 SYN 报文段, 如图中同意连接, 则发出确认报文, 确认报文中 ACK=1, SYN=1, 确认号 ACKnum=x+1 (客户端的 seq+1), 同时, 自身的相对序号 Seq 也会随机出一个值, 这里为 0。服务器端将上述所有信息放到一个报文段 (即 SYN+ACK 报文段) 中, 一并发送给客户端, 此时, TCP 服务器进程进入 SYN-RCVD (同步收到) 状态。这个报文也不能携带数据, len 为 0。其余的数据段和第一次握手分析一致, 不再赘述。这里也消耗了一个序列号。

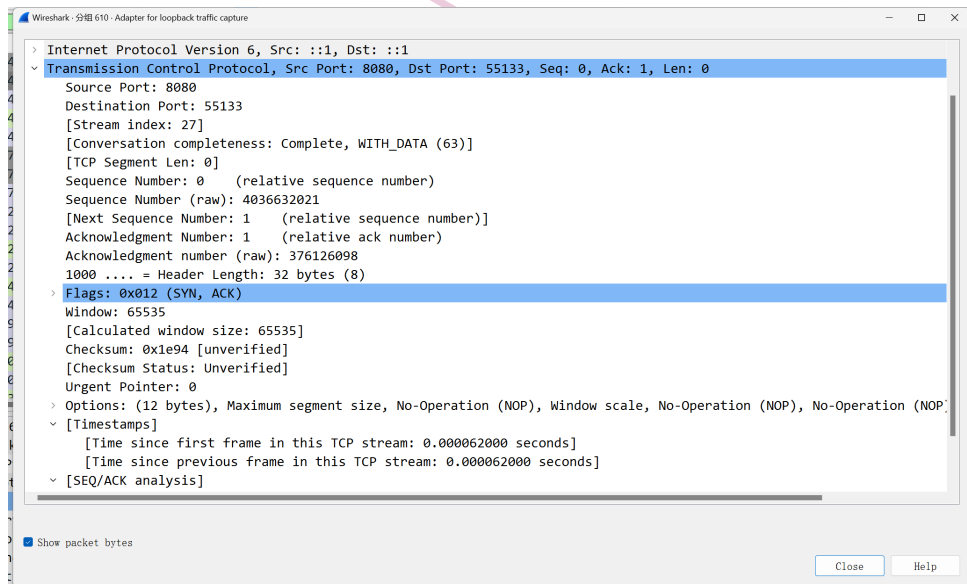


图 9: 第二次握手

3. **第三次握手**: 第三次握手是客户端收到服务器回应的确认报文后, 给服务器的应答, 并为 TCP 连接分配缓存和变量。这里的 flag 中只有 ACK 一位有效, 表示这只是一条确认报

文。由于之前已经消耗了一个序号，因此这条报文的序号 Seq 为 1（第一次的 seq+1），确认号 ACK 为第二次握手的 Seq+1，值为 1。注意到此时的窗口大小已经发生了调整，变成了  $1278 \times 256 = 327168$ 。经过三次握手之后，客户端和服务端都进入了 ESTABLISHED 状态，完成 TCP 三次握手。双方确认连接完成，可以进行数据的传输交互。

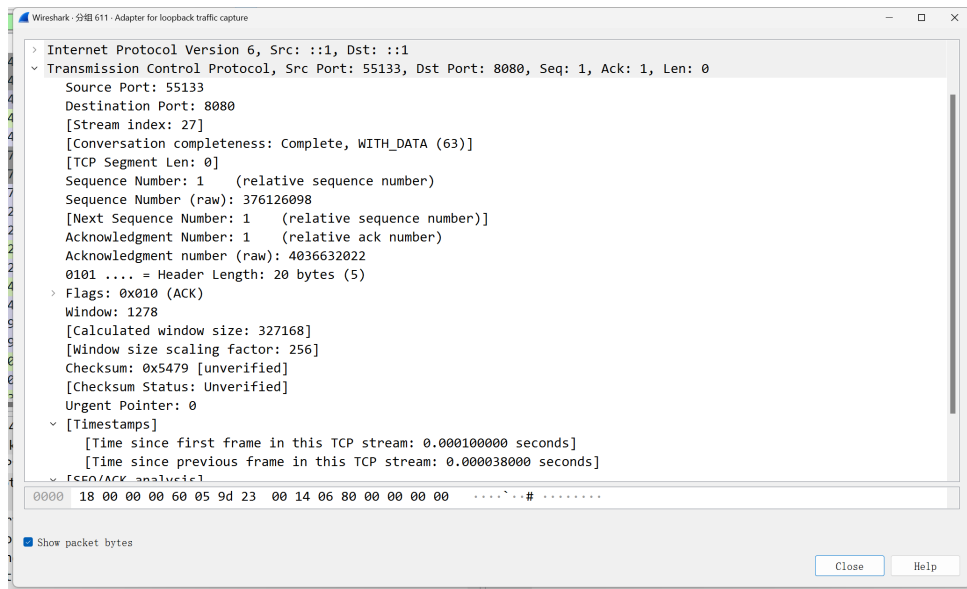


图 10: 第三次握手

**注意：**在后续过程中，seq 和 ACK 值计算还是比较多，但是理解其基本概念，可以辅助理解其计算更新过程：

1. seq 指的是本次发送数据起始的偏移量，需要注意，客户端和服务端端的偏移量是不共通的，因此需要两边维护两个 seq。
2. ACK 指的是期望对方发送数据的偏移量。

## 2. 通过 TCP 传输 HTTP 数据传输全过程

建立连接后，客户端可以从浏览器获取数据，这里首先需要获取的数据就是我上文编写的 html 文件，所以我这里以 html 文件的整个获取过程为例进行讲解：

**1. 客户端向服务器发送 GET 请求** GET 请求是 HTTP 协议支持的，也是 HTTP 协议中最常用的请求资源的方法。这里我们可以发现请求获取资源的目标是 index.html 文件，这是 nginx 架构中用来展示页面的文件。注意这里使用的协议是 HTTP1.1 版本。

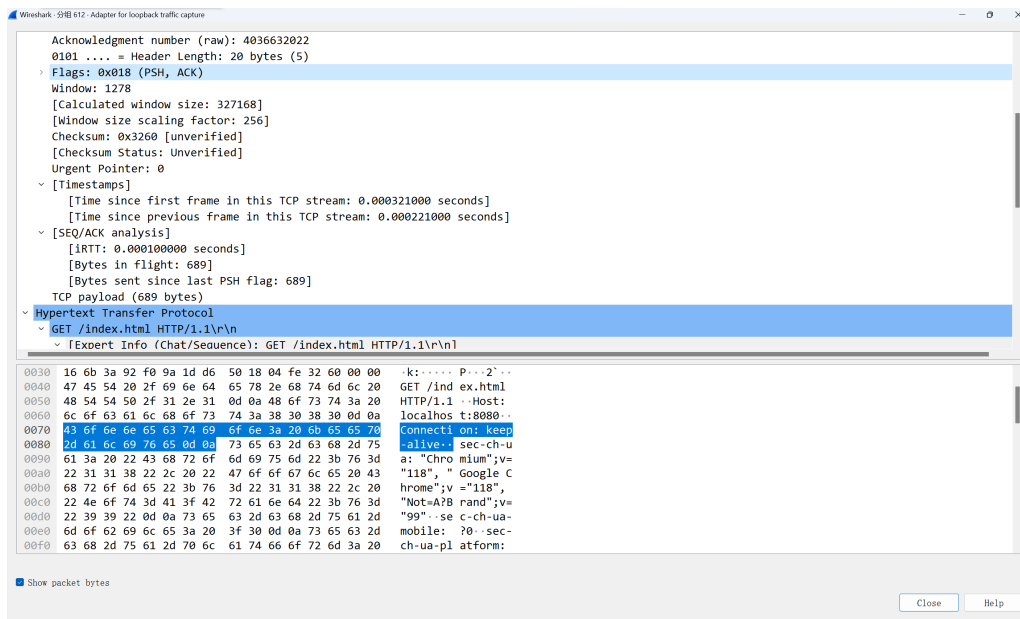


图 11: GET 请求报文

端口号是从 55133 到 8080 的，说明是客户端向服务器发送，此时的 PSH 和 ACK 均为 1，由于在第三次握手时，消耗了一个序号，因此本次的 Seq 为 1。由于 TCP 报文段封装了 HTTP 报文数据，因此 TCP 载荷也就是数据段的值不再为 0，可以看到数据段长度为 689 字节，因此下一个 Seq 的值就到了 690 (1+689)。

然后分析一下 GET 报文的内容，GET 请求通常没有请求体，所以这里分析其他部分（请求行、请求头、空行），其中请求头会包含很多字段，这里我以请求 html 文件的 GET 请求报文为例：

1. 请求行：**GET /index.html HTTP/1.1**，由三部分组成：HTTP 方法、请求的资源 and 使用的 HTTP 版本。
2. 请求头：**host: localhost:8080**，这是请求头中的 host 字段，声明连接的目标主机，这里 8080 是我之前设置的端口号。
3. 请求头：**Connection: keep-alive**，对于 HTTP 连接的处理，keep-alive 表示保持连接，代表这是一个长连接，这也是 HTTP1.1 协议默认的。在对应的 HTTP 响应报文中，如果是一个长连接，这个字段会继续保持 keep-alive，如果是一个短连接的话，则会是 close。
4. 请求头：**sec-ch-ua**，这是“User-Agent Client Hints”的一个字段，用于提供关于浏览器的信息。其中“Chromium”;v="118"，"Google Chrome";v="118"，"Not A Brand";v="99" 表明用户使用的浏览器是基于 Chromium 118 的，正是我的电脑的 118 版本的 Google Chrome。
5. 请求头：**sec-ch-ua-mobile**，该字段指示用户是否使用移动设备。?0 表示桌面设备。
6. 请求头：**sec-ch-ua-platform**，这个字段提供了关于用户操作系统的信息。在这里，“Windows”表示正在使用 Windows 操作系统。
7. 请求头：**Upgrade-Insecure-Requests**，这个字段指示浏览器想要升级为安全连接。1 表示浏览器希望升级不安全的请求。



8. 请求头: **User-Agent**, 这个字段提供了关于发起请求的浏览器和操作系统的信息, Mozilla/5.0 是浏览器的标识; Windows NT 10.0; Win64; x64 表示操作系统是 Windows 10 64 位; AppleWebKit/537.36 是呈现引擎的标识; KHTML, like Gecko 是呈现引擎的描述; Chrome/118.0.0.0 表示这是 Chrome 浏览器版本 118; 最后的 Safari/537.36 是另一个呈现引擎的标识。
9. 请求头: **Accept**, 这表示客户端可以识别的响应内容。text/html: HTML 文档; application/xhtml+xml: XHTML 文档; image/avif: AVIF 格式的图片; image/webp: WebP 格式的图片; image/apng: APNG 格式的图片; \*/\*;q=0.8: 其他任何类型的内容, 但质量系数是 0.8。application/signed-exchange;v=b3;q=0.7: Signed exchanges 的一个版本具有权重 0.7。参数 q 是权重因子, 范围从 0.0 到 1.0, 表示用户代理对特定类型的相对偏好。当服务器有多种资源可以提供时, 这可以帮助服务器确定应发送哪种资源。
10. 请求头: **Sec-Fetch-Site**: 这个字段描述了请求的来源关系。none 表示请求没有与任何网站关联, 是由用户直接在地址栏中输入 URL 发起的。  
**Sec-Fetch-Mode**, 指定请求的模式。navigate 表示该请求是由用户导航到所请求的地址触发的。  
**Sec-Fetch-User**, 这个字段表示用户是否明确触发了请求。?1 表示是由用户行为触发的请求。  
**Sec-Fetch-Dest**, 表示请求的目标或用途。document 表示请求是用于导航到文档的, 如 HTML 文档。
11. 请求头: **Accept-Encoding**, 表示客户端接受的编码格式。
12. 请求头: **Accept-Language**, 表示客户端接受的语言集。
13. 空行: \r\n 表示换行, 有一个单独的空行, 用来分隔请求头和请求体。

## 2. 确认和响应头发送 首先展示一下图片, 其中的 1, 2, 3 分别依次对应下面的三段分析:

o.	Time	Source	Destination	Protocol	Length	Info
609	32.124327	::1	::1	TCP	76	55133 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
610	32.124389	::1	::1	TCP	76	8080 → 55133 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
611	32.124427	::1	::1	TCP	64	55133 → 8080 [ACK] Seq=1 Ack=1 Win=327168 Len=0
612	32.124648	::1	::1	HTTP	753	GET /index.html HTTP/1.1
613	32.124680	::1	::1	TCP	64	8080 → 55133 [ACK] Seq=1 Ack=690 Win=2159872 Len=0
614	32.127065	::1	::1	TCP	76	55134 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
615	32.127119	::1	::1	TCP	76	8080 → 55134 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
616	32.127150	::1	::1	TCP	64	55134 → 8080 [ACK] Seq=1 Ack=1 Win=327168 Len=0
617	32.132464	::1	::1	TCP	303	8080 → 55133 [PSH, ACK] Seq=1 Ack=690 Win=2159872 Len=239 [TCP segment of a reassembled PDU]
618	32.132511	::1	::1	TCP	64	55133 → 8080 [ACK] Seq=690 Ack=240 Win=326912 Len=0
619	32.132554	::1	::1	HTTP	15	HTTP/1.1 200 OK (text/html)
620	32.132568	::1	::1	TCP	64	55133 → 8080 [ACK] Seq=690 Ack=1693 Win=325632 Len=0
621	32.154829	::1	::1	HTTP	632	GET /styles.css HTTP/1.1
622	32.154883	::1	::1	TCP	64	8080 → 55133 [ACK] Seq=1693 Ack=1258 Win=2159360 Len=0
623	32.159440	::1	::1	TCP	302	8080 → 55133 [PSH, ACK] Seq=1693 Ack=1258 Win=2159360 Len=238 [TCP segment of a reassembled PD...
624	32.159482	::1	::1	TCP	64	55133 → 8080 [ACK] Seq=1258 Ack=1931 Win=325376 Len=0
625	32.160257	::1	::1	HTTP	12	HTTP/1.1 200 OK (text/css)

图 12: 确认和响应头发送

1. 根据 TCP 协议, 服务器收到来自客户端的 GET 请求后, 首先会发送一个 ack 给客户端, 表示服务器已经收到了来自客户端的 GET 请求, len 是 0, 说明这是一个确认报文, 并没有具体内容。seq 是 1, 是由于在握手的时候消耗了一个序号。ACK 的值是 690 是 GET 请求报文的 seq+len+1=0+689+1。对于 WIN 来说, 会根据 256 取整进行递减, 例如这里到达红圈三, WIN 字段从 327168 减小到了 326912, 少了一个 256。后面都是类似的运算规则。
2. 然后开始发送消息, 这里的消息是指服务器向客户端发送 HTTP 响应头, 状态为 200 OK。len 是 239, 说明数据段的长度是 239 个字节。而且发现这条报文带有带有 [TCP segment

of a reassembled PDU] 标签, 说明完整的 TCP 报文被拆分成了不同的段发送。此时 seq 值仍为 1, ACK 仍为 690。

3. 客户端在收到数据后, 根据 TCP 协议又会向服务器发送一个确认报文, 即一个 ack, seq 是 690, ACK 值等 240, 这表示这是上条数据传输报文的确认报文, 表示客户端已经收到了服务器发送的消息。

注意: 这里可能已经开始发送数据, 例如像请求数据量比较大的视频或者音频等。多次传输的话就是, 只需要多次的 (psh, ack), 然后结合后续的确认报文。

**3. 数据发送** 然后服务器向客户端发送了 HTML 页面内容的文本, 这即是 HTTP 响应报文。下图中可以看到其将我编写的 html 文档内容已经包含到报文中。

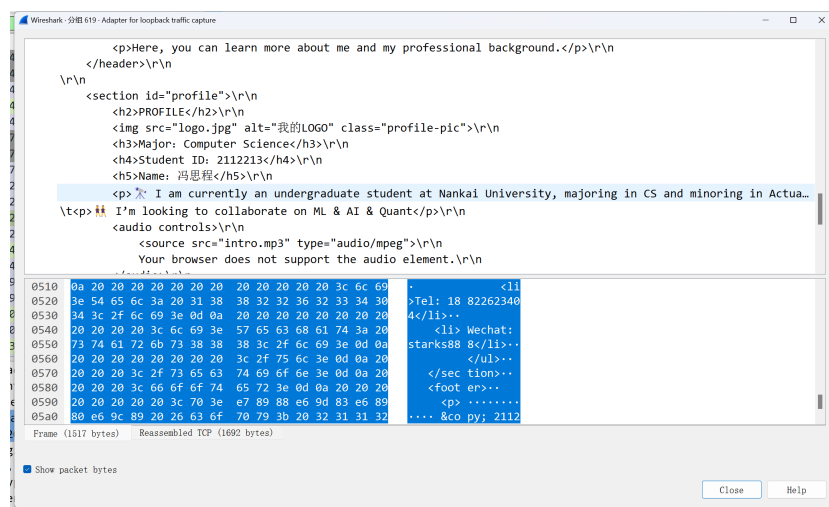


图 13: 数据发送

其 PSH 和 ACK 有效, 说明是传输数据。seq 和 ACK 值分别为 240 和 690, 计算规则不变。TCP 数据段承载了 HTTP 报文, 在 HTTP 报文的数据段中包含了 HTML 页面的具体内容。下面分析一下 HTTP 响应报文的内容。

HTTP 响应报文内容主要有状态行、响应头、消息体。

#### 1. 状态行:

- HTTP/1.1 表示 HTTP 的版本。
- 200 OK 表示请求成功。

#### 2. 响应头:

- Server: nginx/1.25.3 说明该响应是由 nginx 服务器版本 1.25.3 生成的。
- Date: Wed, 01 Nov 2023 02:42:33 GMT 表示响应的生成日期和时间。
- Content-Type: text/html 表示返回的消息体的数据类型为 HTML。
- Content-Length: 1453 表示响应消息体的长度为 1453 字节。
- Last-Modified: Tue, 31 Oct 2023 15:30:02 GMT 表示资源上次修改的时间。
- Connection: keep-alive 表示长连接。这里可以发现和前文的 GET 请求头的 connection 字段一致, 都是 keep-alive, 说明这是一个长连接。长连接是一种优化, 避免每次 get 数据都需要重新建立连接。



- ETag: "65411d7a-5ad" 是资源的版本标识, 用于缓存验证。
- Accept-Ranges: bytes 表示服务器支持字节范围请求。

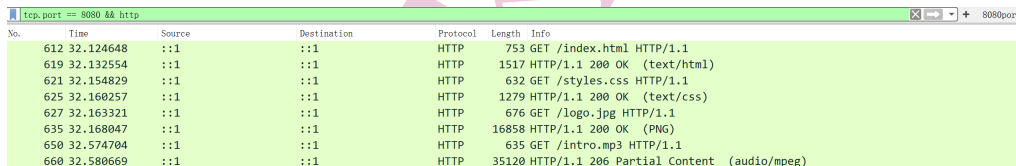
### 3. 消息体:

- File Data: 1453 bytes 表示数据部分的长度。
- Line-based text data: text/html (44 lines) 表示数据部分是基于行的文本数据, 具有 44 行。即我编写的 html 文档

客户端在接收到服务器响应的数据后, 会再发送一次确认报文, 其中 seq 和 ACK 计算方法和上面一致, 不再赘述。

**补充说明** http 协议可以传输任意类型的数据, 例如像本次实验上面没有具体分析的音频、图片、css 文件等, 都是通过类似上文的过程实现的。

**HTTP 过滤结果展示** 将过滤条件变成: `tcp.port == 8080 && http`, 可以看到这是一个 HTTP1.1 协议的体现, 并没有流水线的体现, 说明就是 HTTP1.1 标准协议。看到分别一次获取了 index.html、styles.css、logo.jpg、intro.mp3, 这里用到的都是 HTTP1.1 协议支持的 GET 请求, 然后对于 html 文件、css 文件、照片都是返回的状态是 200 OK。但是对于像音频这种范围请求, 返回的 http 响应状态是 206 Partial Content, 并只发送请求的资源部分, 同时在响应头中包含 Content-Range 来表示返回的是哪部分内容, 我查看到 Content-Range 字段的内容是 0-60863/60864, 说明其实这里是获取全了的, 但是对于范围请求, HTTP 协议一般规定的响应状态就是 206 Partial Content。

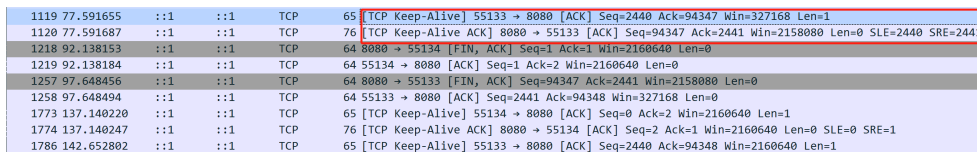


No.	Time	Source	Destination	Protocol	Length	Info
612	32.124648	:::1	:::1	HTTP	753	GET /index.html HTTP/1.1
619	32.132554	:::1	:::1	HTTP	1517	HTTP/1.1 200 OK (text/html)
621	32.154829	:::1	:::1	HTTP	632	GET /styles.css HTTP/1.1
625	32.160257	:::1	:::1	HTTP	1279	HTTP/1.1 200 OK (text/css)
627	32.163321	:::1	:::1	HTTP	676	GET /logo.jpg HTTP/1.1
635	32.168047	:::1	:::1	HTTP	16858	HTTP/1.1 200 OK (PNG)
650	32.574704	:::1	:::1	HTTP	635	GET /intro.mp3 HTTP/1.1
660	32.580669	:::1	:::1	HTTP	35120	HTTP/1.1 206 Partial Content (audio/mpeg)

图 14: HTTP 过滤

### 3. 长连接维持

这里的两次 tcp 连接都是长连接的, 这也是 HTTP1.1 协议支持的。这里维持长连接的方法是通过从客户端向服务器发送 **keep-alive** 试探报文, 来试探是否仍与服务器处于连接状态, 如果连接仍保持, 则服务器会向客户端发送一个 **keep-alive ack** 确认报文。这里展示一下 55133 端口与 8080 的长连接保持试探过程:



1119	77.591655	:::1	:::1	TCP	65	[TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94347 Win=327168 Len=1
1120	77.591687	:::1	:::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94347 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
1218	92.138153	:::1	:::1	TCP	64	8080 → 55134 [FIN, ACK] Seq=1 Ack=1 Win=2160640 Len=0
1219	92.138184	:::1	:::1	TCP	64	55134 → 8080 [ACK] Seq=1 Ack=2 Win=2160640 Len=0
1257	97.648456	:::1	:::1	TCP	64	8080 → 55133 [FIN, ACK] Seq=94347 Ack=2441 Win=2158080 Len=0
1258	97.648494	:::1	:::1	TCP	64	55133 → 8080 [ACK] Seq=2441 Ack=94348 Win=327168 Len=0
1773	137.140220	:::1	:::1	TCP	65	[TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
1774	137.140247	:::1	:::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
1786	142.652802	:::1	:::1	TCP	65	[TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1

图 15: 保持过程

这里可以看到试探报告的 len 是 1, 这是一个用于试探的数据没有实际含义。你可以看到这里的 seq 是之前已经确认的  $seq-1=2441-1=2440$ , ACK 值保持不变, 仍是 94347。

然后再来看这个服务器给客户端发送的 ack 报文，看到这里的 seq 是试探报文的 ACK 值，ACK 值是试探报文的 seq+len=2441。len=0 表示这是一个不会传输数据的报文，这里需要注意的是这里有一个 SLE 的值是 2440，SRE 的值是 2441，SLE 代表已被接收的数据块的第一个字节的序列号，SRE 代表已被接收的数据块的最后一个字节的下一个序列号。这两个字段值本意是被设计来完善重传机制的，这里用于确认服务器可以收到客户端发来的试探报文，这两个的具体值也说明我们服务器接收到了试探报文发送的数据。

注意：这里如果是 HTTP1.0 协议的话，则会是短连接，对于短连接的话，GET 请求的 connection 字段是 keep-alive，而响应的时候 connection 字段直接会消失。

#### 4. TCP 连接断开

首先我们知道标准的 tcp 协议规定的是四次挥手的过程，过程如下：在 TCP 四次挥手的过程中，不论是服务器端还是客户端，都有可能首先发起断开连接请求。以服务器端为例，当它决定关闭连接时，它会发送一个 FIN 报文段表示希望终止连接，进入 FIN\_WAIT\_1 状态。客户端收到这个请求后，进入 CLOSED\_WAIT 状态，回应一个 ACK 报文段，服务器收到后进入 FIN\_WAIT\_2 状态。之后，如果客户端也表示希望关闭连接，它会发送另一个 FIN 报文段并进入 LAST\_ACK 状态。最后，服务器端在收到这个 FIN 后，回复一个 ACK 报文段并进入 TIME\_WAIT 状态，客户端收到 ACK 报文后就会关闭达到 CLOSED 状态。经过一定时间后（这个时间是 2MSL，即两倍的报文生存时间），服务器端确认客户端已关闭连接，于是它也正式关闭连接，达到 CLOSED 状态。这个过程确保了双方都达成了关闭连接的共识，并且保证所有数据的完整传输。

但是在本次实验中和标准的 tcp 协议规定的断开过程有所出入，这里我先来展示一下我在 wireshark 中抓包的结果，如下：（注意：这里我即使在访问后直接关闭浏览器的页面，也不会捕获到四次挥手，这里是浏览器的问题，并不会在你关闭后立刻断开连接，因为觉得你可能在短时间内还会访问相同网址，这是浏览器做出的一个优化。）

1119	77.591655	::1	::1	TCP	65 [TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94347 Win=327168 Len=1
1120	77.591687	::1	::1	TCP	76 [TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94347 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
1218	92.138153	::1	::1	TCP	64 8080 → 55134 [FIN, ACK] Seq=1 Ack=1 Win=2160640 Len=0
1219	92.138184	::1	::1	TCP	64 55134 → 8080 [ACK] Seq=1 Ack=2 Win=2160640 Len=0
1257	97.648456	::1	::1	TCP	64 8080 → 55133 [FIN, ACK] Seq=94347 Ack=2441 Win=2158080 Len=0
1258	97.648494	::1	::1	TCP	64 55133 → 8080 [ACK] Seq=2441 Ack=94348 Win=327168 Len=0
1773	137.140220	::1	::1	TCP	65 [TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
1774	137.140247	::1	::1	TCP	76 [TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
1786	142.652802	::1	::1	TCP	65 [TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1
1787	142.652831	::1	::1	TCP	76 [TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94348 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
3134	182.154253	::1	::1	TCP	65 [TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
3135	182.154278	::1	::1	TCP	76 [TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
3156	187.663370	::1	::1	TCP	65 [TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1
3157	187.663394	::1	::1	TCP	76 [TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94348 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
3403	212.150137	::1	::1	TCP	64 8080 → 55134 [RST, ACK] Seq=2 Ack=1 Win=0 Len=0
3445	217.654578	::1	::1	TCP	64 8080 → 55133 [RST, ACK] Seq=94348 Ack=2441 Win=0 Len=0

图 16: 断开过程

**第一次挥手** 以端口号 55134（第二个 tcp 连接的端口号，一直是空闲的）为例，可以看到首先由服务器发送了一个 fin 报文给客户端。然后我们来观察一下上下文如下：

65	[TCP Keep-Alive]	55134 → 8080	[ACK]	Seq=0 Ack=1 Win=2160640 Len=1
76	[TCP Window Update]	8080 → 55134	[ACK]	Seq=1 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
65	[TCP Keep-Alive]	55133 → 8080	[ACK]	Seq=2440 Ack=94347 Win=327168 Len=1
76	[TCP Keep-Alive ACK]	8080 → 55133	[ACK]	Seq=94347 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
64	8080 → 55134	[FIN, ACK]	Seq=1 Ack=1 Win=2160640 Len=0	
64	55134 → 8080	[ACK]	Seq=1 Ack=2 Win=2160640 Len=0	
64	8080 → 55133	[FIN, ACK]	Seq=94347 Ack=2441 Win=2158080 Len=0	
64	55133 → 8080	[ACK]	Seq=2441 Ack=94348 Win=327168 Len=0	

图 17: 第一次挥手的由来

上面的试探分别是端口 55133 和 55134 的第一次试探，然后可以发现，服务器对试探报文的 ack 确认和后面的 fin 报文是连接起来的，所以这里 nginx 的机制就是：如果客户端开始给服务器发试探报文了，也就是说现在这个连接里面并没有请求了，事情都干完了，所以服务器想关掉它。

下面可以看到具体的一些内容，可以看到与我们预料的无出入。

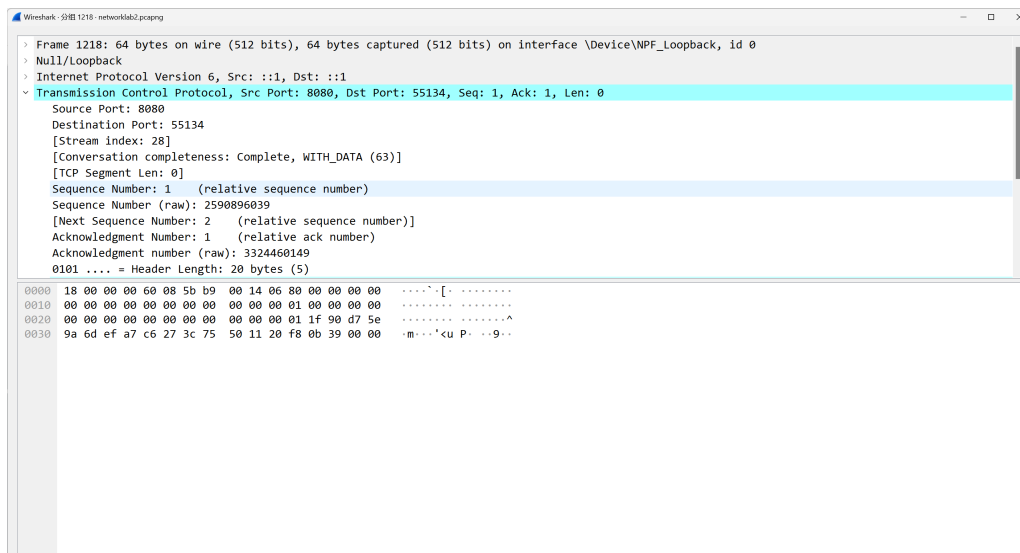


图 18: 第一次挥手

**第二次挥手** 还是以端口号 55134 为例，可以看到客户端对服务器发送的 fin 报文进行了确认，ACK 有效。seq 为上次的 ACK 值 =1，ACK 值是上次的 seq+1=2。

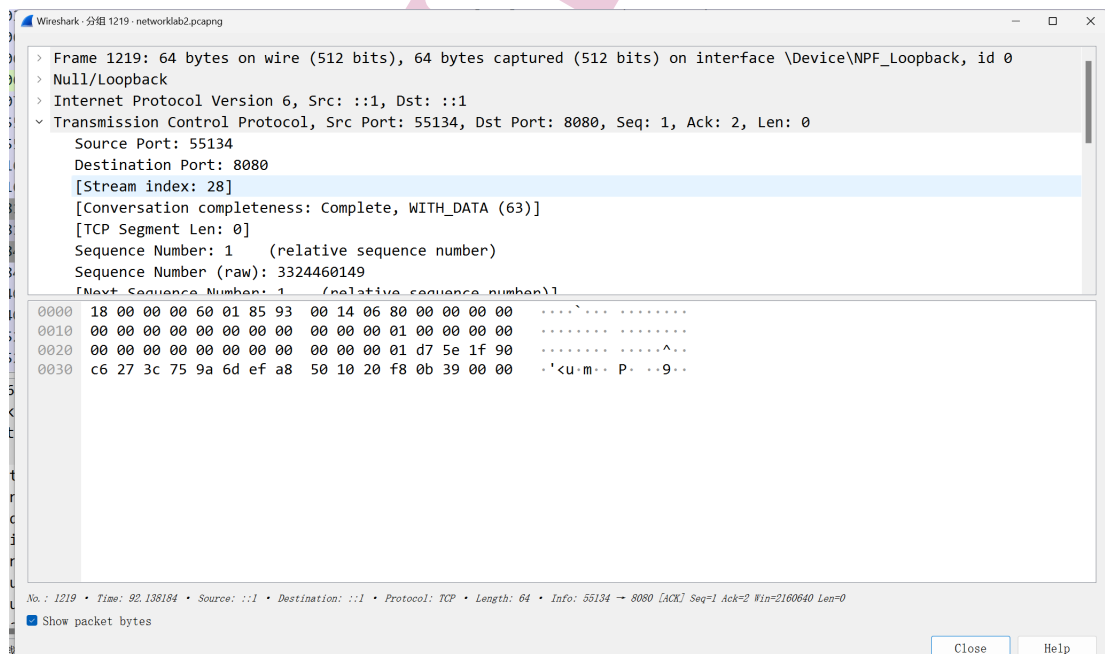


图 19: 第二次挥手

但是本应该出现的第三次挥手和第四次挥手，也就是期望的客户端也向服务器发送 fin 报文，然后服务器再对这个报文进行确认。实际上这个步骤并没有发生，经过我仔细查证，这个可以解

释为 HTTP1.1 的长连接机制，简单来说就是，现在是服务器想关掉连接，但是客户端不想关，于是可以看到在这两个 tcp 连接完成前两次挥手后，客户端继续发送试探报文来保证连接维持，而不是我们所期望的 fin 报文。如下：

1218	92.138153	::1	::1	TCP	64	8080 → 55134 [FIN, ACK] Seq=1 Ack=1 Win=2160640 Len=0
1219	92.138184	::1	::1	TCP	64	55134 → 8080 [ACK] Seq=1 Ack=2 Win=2160640 Len=0
1257	97.648456	::1	::1	TCP	64	8080 → 55133 [FIN, ACK] Seq=94347 Ack=2441 Win=2158080 Len=0
1258	97.648494	::1	::1	TCP	64	55133 → 8080 [ACK] Seq=2441 Ack=94348 Win=327168 Len=0
1773	137.140220	::1	::1	TCP	65	[TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
1774	137.140247	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
1786	142.652802	::1	::1	TCP	65	[TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1
1787	142.652831	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94348 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
3134	182.154253	::1	::1	TCP	65	[TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
3135	182.154278	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
3156	187.663370	::1	::1	TCP	65	[TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1
3157	187.663394	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94348 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441

图 20: 客户端继续发送试探报文

**服务器强制关闭连接** 可以看到在经过上文的几次试探报文后，服务器主动强制关闭了两个 tcp 连接，如下：

1218	92.138153	::1	::1	TCP	64	8080 → 55134 [FIN, ACK] Seq=1 Ack=1 Win=2160640 Len=0
1219	92.138184	::1	::1	TCP	64	55134 → 8080 [ACK] Seq=1 Ack=2 Win=2160640 Len=0
1257	97.648456	::1	::1	TCP	64	8080 → 55133 [FIN, ACK] Seq=94347 Ack=2441 Win=2158080 Len=0
1258	97.648494	::1	::1	TCP	64	55133 → 8080 [ACK] Seq=2441 Ack=94348 Win=327168 Len=0
1773	137.140220	::1	::1	TCP	65	[TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
1774	137.140247	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
1786	142.652802	::1	::1	TCP	65	[TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1
1787	142.652831	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94348 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
3134	182.154253	::1	::1	TCP	65	[TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
3135	182.154278	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
3156	187.663370	::1	::1	TCP	65	[TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1
3157	187.663394	::1	::1	TCP	76	[TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94348 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
3403	212.150137	::1	::1	TCP	64	8080 → 55134 [RST, ACK] Seq=2 Ack=1 Win=0 Len=0
3445	217.654578	::1	::1	TCP	64	8080 → 55133 [RST, ACK] Seq=94348 Ack=2441 Win=0 Len=0

图 21: 强制关闭连接

想要解释这个的合理性，需要先看下来一下 nginx 这个框架的配置文件，这里我是从 docker 中找到 nginx.conf 文件，请观察红圈中的内容：

```

Name ↑      Note      Size      Last modified      Mode
├─ mke2fs.conf      782 Bytes      8 months ago      -rw-r--r--
├─ motd      286 Bytes      1 month ago      -rw-r--r--
├─ mtab -> /proc/mounts      12 Bytes      11 hours ago      Lrwxrwxrwx
└─ /etc/nginx/nginx.conf      Nginx
19      $status $body_bytes_sent $http_referer
20      " $http_user_agent" $http_x_forwarded_for";
21
22      access_log /var/log/nginx/access.log main;
23
24      sendfile on;
25      #tcp_nopush on;
26      keepalive_timeout 65;
27
28      #gzip on;
29
30      include /etc/nginx/conf.d/*.conf;
31
32
33

```

图 22: 强制关闭连接

这里 nginx 设置了长连接的强制关闭时间，表示一个持久连接在进入空闲状态后可以保持打开状态的时间，也就是说一旦客户端和服务端之间的交互结束，并且没有新的请求在该连接上发送，这个计时器就开始运行。如果在 keepalive\_timeout 指定的时间段内，没有新的请求到达，连接将被服务器强制关闭。这也解释了为什么服务器会先强制关闭 55134 端口，因为其空闲时间更长。这里 nginx 的默认设置是 65 秒。

这里的 seq 和 ACK 值和上一次服务器传回来的确认报文一致，说明这也是连接起来的，也就是说明这里到达时限后，服务器会主动强制关闭掉 tcp 连接。RST 是一个控制标志位，用于指示接收端应立即终止当前连接。

3134	182.154253	:::1	:::1	TCP	65 [TCP Keep-Alive] 55134 → 8080 [ACK] Seq=0 Ack=2 Win=2160640 Len=1
3135	182.154278	:::1	:::1	TCP	76 [TCP Keep-Alive ACK] 8080 → 55134 [ACK] Seq=2 Ack=1 Win=2160640 Len=0 SLE=0 SRE=1
3156	187.663370	:::1	:::1	TCP	65 [TCP Keep-Alive] 55133 → 8080 [ACK] Seq=2440 Ack=94348 Win=2160640 Len=1
3157	187.663394	:::1	:::1	TCP	76 [TCP Keep-Alive ACK] 8080 → 55133 [ACK] Seq=94348 Ack=2441 Win=2158080 Len=0 SLE=2440 SRE=2441
3403	212.150137	:::1	:::1	TCP	64 8080 → 55134 [RST, ACK] Seq=2 Ack=1 Win=0 Len=0
3445	217.654578	:::1	:::1	TCP	64 8080 → 55133 [RST, ACK] Seq=94348 Ack=2441 Win=0 Len=0

图 23: 强制关闭

## 5. 浏览器的“并行”机制

浏览器的 http1.1 协议会预先多开几个 tcp 连接用来备用（防止只开一个 tcp 连接产生阻塞或者意外情况影响传输效率），然后是多余的那些 tcp 连接都是长连接就先空闲着，如果之前用的 tcp 阻塞或者意外情况，则会采用这些备用的 tcp 连接进行传输。如果没用到备用的 tcp 连接，则在后面这里会像上文说的一样一起在最后强制断开连接。

在我的这次 wireshark 捕获中，可以看到在最开始很快就通过三次握手建立了第二个 tcp 连接，端口号是第一个端口 +1。但是可以发现在后续这个连接并没有用于数据传输，可以理解为一个空闲 tcp 长连接，在最后和第一个 tcp 长连接一起被 nginx 强制断开。下图蓝圈就是第二个 tcp 连接，其的 seq、ACK 值等与第一个 tcp 连接是隔离的。

609	32.124327	:::1	:::1	TCP	76 55133 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
610	32.124389	:::1	:::1	TCP	76 8080 → 55133 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
611	32.124427	:::1	:::1	TCP	64 55133 → 8080 [ACK] Seq=1 Ack=1 Win=327168 Len=0
612	32.124648	:::1	:::1	HTTP	753 GET /index.html HTTP/1.1
613	32.124680	:::1	:::1	TCP	64 8080 → 55133 [ACK] Seq=1 Ack=690 Win=2159872 Len=0
614	32.127065	:::1	:::1	TCP	76 55134 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
615	32.127119	:::1	:::1	TCP	76 8080 → 55134 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
616	32.127150	:::1	:::1	TCP	64 55134 → 8080 [ACK] Seq=1 Ack=1 Win=327168 Len=0
617	32.132464	:::1	:::1	TCP	303 8080 → 55133 [PSH, ACK] Seq=1 Ack=690 Win=2159872 Len=239 [TCP segment of a reassembled PDU]
618	32.132511	:::1	:::1	TCP	64 55133 → 8080 [ACK] Seq=690 Ack=240 Win=326912 Len=0
619	32.132554	:::1	:::1	HTTP	15.. HTTP/1.1 200 OK (text/html)
620	32.132568	:::1	:::1	TCP	64 55133 → 8080 [ACK] Seq=690 Ack=1693 Win=325632 Len=0
621	32.154829	:::1	:::1	HTTP	632 GET /styles.css HTTP/1.1
622	32.154883	:::1	:::1	TCP	64 8080 → 55133 [ACK] Seq=1693 Ack=1258 Win=2159360 Len=0
623	32.159440	:::1	:::1	TCP	302 8080 → 55133 [PSH, ACK] Seq=1693 Ack=1258 Win=2159360 Len=238 [TCP segment of a reassembled PD..]
624	32.159482	:::1	:::1	TCP	64 55133 → 8080 [ACK] Seq=1258 Ack=1931 Win=325376 Len=0
625	32.160257	:::1	:::1	HTTP	12.. HTTP/1.1 200 OK (text/css)
626	32.160300	:::1	:::1	TCP	64 55133 → 8080 [ACK] Seq=1258 Ack=3146 Win=324096 Len=0

图 24: 第二次 tcp 连接

## 6. http1.0 VS http1.1

这里我还额外对 http1.0 协议进行了观察，这里我们知道 http1.0 协议是短连接，短连接的含义就是在一个 tcp 连接在处理完一个请求后直接断开连接，这里可以将上文中的 timeout 设定改为 0，即可模拟短连接的实现，但是很明显这种会拉低性能，于是就有了这里上文的 http1.1 协议的长连接。

当然后续还有效率更好的 http2.0 协议等等，采用流机制等更加先进的机制来提高性能。

## 三、 总结与思考

这次实验在编码方面难度较低，没有遇到什么困难，但是我主要遇到了两个难点。第一个难点就是 docker 的配置和对应 nginx 的 image 的拉取和容器 run 的设置选项。这些我花费了不少的时间和精力来熟悉。但是我也通过这次实验对 docker 这个工具的使用有了一个初步的了解，同时也对 nginx 框架有了较为深刻的理解。

第二大难点就是 wireshark 的抓包分析过程，遇到了许多奇奇怪怪的情况，而且结果会由于浏览器类别、版本、设置而不同，像上文中没有出现正常的四次挥手，而是只有两次挥手然后继

续试探报文最后强制关闭，这个过程在最开始令我十分的困惑，而且这种情况似乎十分的少见，最终在我的仔细查证下找到了一个合理的解释。

在这个过程，我对 wireshark 这个工具更加的熟悉，使用起来也更加的得心应手。同时对于 TCP 协议和 HTTP 协议有了更加深刻的认识 and 了解。而且在这个过程，我也了解了一些优化技术，例如我曾试过将上文提到的 nginx 的 `keepalive_timeout` 设置为 0，即变成短连接，发现还出现了 3 次挥手即断开连接的情况，后来也了解了这是一种 TCP 捎带的技术。

整个过程锻炼了我的分析能力也磨练了我的耐心，可以说是非常不错的一次练习 tcp 协议和 http 协议的实验。

NIJUB