



南開大學
Nankai University

网络空间安全学院
计算机网络课程实验报告

实验二：配置 **WEB** 服务器，分析交互
过程

姓名：魏伯繁

学号：2011395

专业：信息安全

2022 年 10 月 28 日

目录

1 实验要求	2
2 网页搭建	2
2.1 服务器搭建	2
2.2 网页渲染	2
3 抓包分析	3
3.1 概述	3
3.2 http/1.1	3
3.3 三次握手	4
3.3.1 第一次握手	5
3.3.2 第二次握手	6
3.3.3 第三次握手	7
3.4 http 报文分析	7
3.4.1 页面请求	7
3.4.2 四次挥手	9
3.4.3 第一次挥手	10
3.4.4 第二次挥手	10
3.4.5 第三次挥手	10
3.4.6 第四次挥手	11
4 问题解决及分析	12
4.1 浏览器缓存	12
4.2 304 状态码	13
4.3 多端口	13
4.4 大量 PSH 的出现	13
4.5 favicon	14
4.6 json	14
4.7 Lkeep-alive	14
5 总结	15

1 实验要求

(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）和自己的 LOGO。

(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。

需要实现的功能包括：

Web 服务器搭建、编写 Web 页面

使用 Wireshark 捕获交互过程

2 网页搭建

2.1 服务器搭建

在本实验中使用 springboot 为框架搭建 maven 项目作为创建本地服务器的方式。具体工具包括：IDEA2021、maven3.5.4、springboot2.5.3

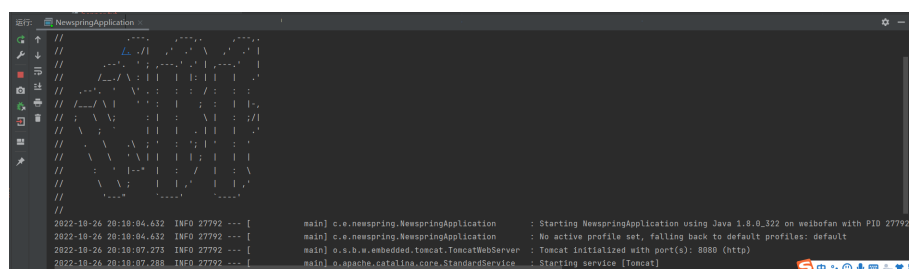


图 2.1: 服务器启动视图

在访问时可以通过 127.0.0.1:8080/myvieww 进行访问

2.2 网页渲染

由于本次实验重点在于使用 wireshark 进行抓包分析，所以未使用 CSS+JS 对页面进行过渡渲染，而是只是用 html 进行页面展示。



图 2.2: 服务器网页展示

3 抓包分析

3.1 概述

抓包分析是本实验的重点，在本次实验中，我们将使用 Wireshark 最为抓包工具进行网页数据的抓包分析。

首先，我们需要选择想要进行监听的网络，由于服务器的搭建是在本地，所以我们选择 127.0.0.1

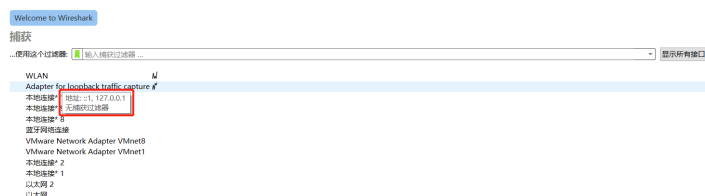


图 3.3: 选择监听网络

最后，我们的分析包括但不限于三次握手、四次挥手、以及对应 http 版本下客户与服务器交互时请求相应数据抓包分析

3.2 http/1.1

HTTP 是一种允许获取资源的协议，例如 HTML 文档。它是 Web 上任何数据交换和客户端 - 服务器协议的基础，这意味着请求由接收者（通常是 Web 浏览器）发起。从所获取的不同子文档重建完整文档，例如文本，布局描述，图像，视频，脚本等。

客户端和服务端通过交换单个消息（而不是数据流）进行通信。客户端（通常是 Web 浏览器）发送的消息称为请求，服务器发送的消息称为响应。

HTTP 是客户端 - 服务器协议：请求由一个实体（用户代理（或代表它的代理））发送。大多数情况下，用户代理是 Web 浏览器，但它可以是任何东西。

每个单独请求都被发送到服务器，服务器将处理它并提供一个称为响应的答案。在该请求和响应之间，存在许多实体，统称为代理，其执行不同的操作并且例如充当网关或高速缓存。



图 3.4: http 简单模型

http1.1 缺省为持久性连接，也就是说在第一次使用三次握手建立连接后，以后再进行数据请求就不需要再进行建立连接的操作，而是默认为始终保持连接。

同样，为了加速数据加载速度，http/1.1 进入了流水线机制，但也需要按顺序进行响应，经历较少的慢启动过程，减少往返时间。

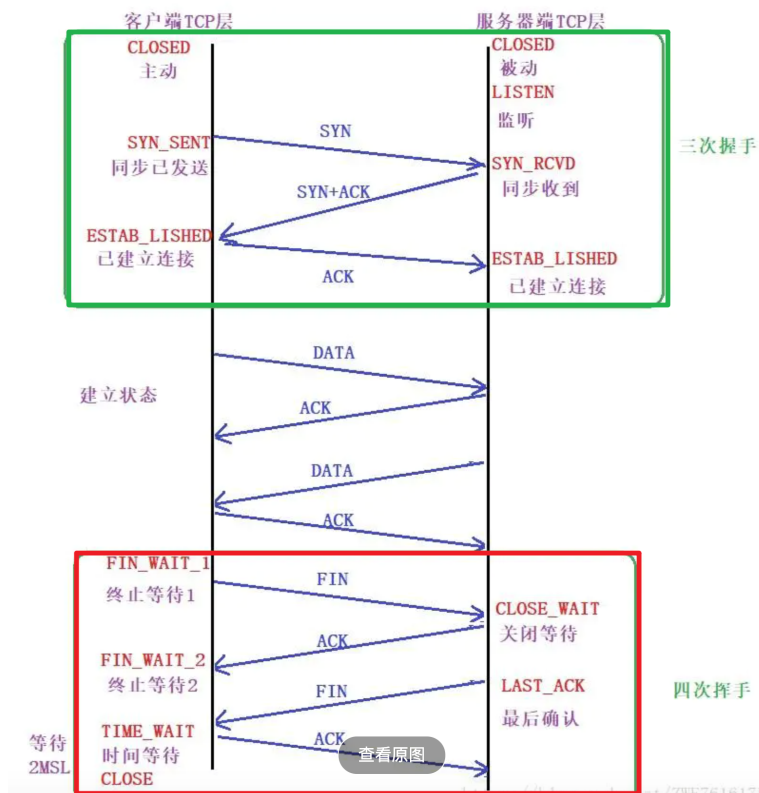


图 3.5: http1.1 模型

3.3 三次握手

想要了解三次握手，需要首先了解 TCP 的头部结构。

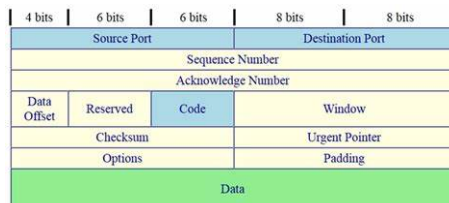


图 3.6: TCP 头部结构

TCP 传递给 IP 层的信息单位称为报文段或段，下面都用段做单位。

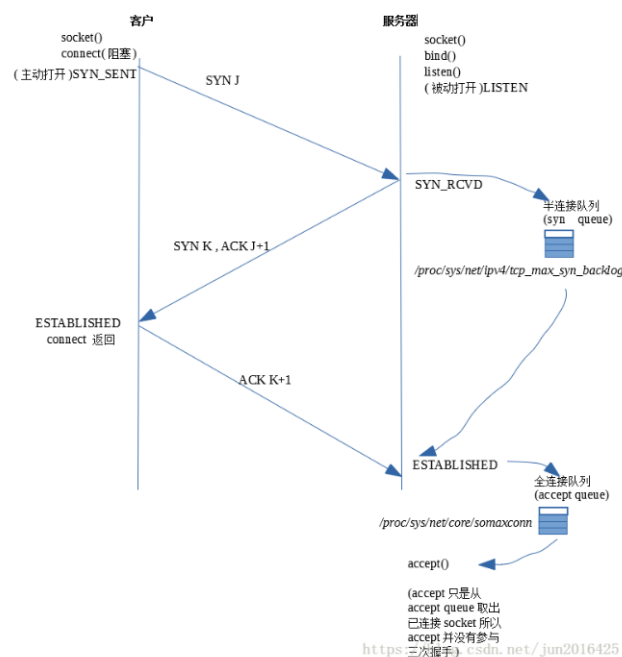


图 3.7: TCP 头部结构

最初两端的 TCP 进程 (客户端和服务端) 都处于关闭状态

一开始, TCP 服务器进程首先创建传输控制块, 用来存储 TCP 连接中的一些重要信息。例如 TCP 连接表、指向发送和接收缓存的指针、指向重传队列的指针, 当前的发送和接收序号等。之后就准备接受 TCP 客户进程的连接请求, 此时 TCP 服务器进程就要进入监听状态等待 TCP 客户进程的连接请求。

TCP 客户进程也是首先创建传输控制块, 然后再打算建立。TCP 服务器进程是被动等待来自 TCP 客户端进程的连接请求, 因此称为被动打开连接。

3.3.1 第一次握手

TCP 连接时向 TCP 服务器进程发送 TCP 连接请求报文段, 并进入同步已发送状态。

由于 TCP 连接建立是由 TCP 客户进程主动发起的, 因此称为主动打开连接。注意 TCP 规定 SYN 被设置为 1 的报文段不能携带数据但要消耗掉一个序号。

所以我们可以看下面这张图片, 他的相对序列号是 0, 但是下一次相对序列号就会变成 1, 证明第一次握手确实会消耗序列号。

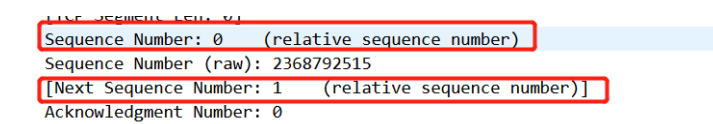


图 3.8: 第一次握手 SEQ

客户端给服务器发送一个 SYN 段 (在 TCP 标头中 SYN 位字段为 1 的 TCP/IP 数据包), 该段中也包含客户端的初始序列号 (Sequence number = J)

SYN 是同步的缩写, SYN 段是发送到另一台计算机的 TCP 数据包, 请求在它们之间建立连接

```

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
  ....0... = Syn: Set
  > [Expert Info (Chat/Sequence): Connection establish request (SYN): server port 8080]
  ....0... = Fin: Not set

```

图 3.9: 第一次握手 SYN

在 Transmission control protocol 中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 状态, 等待服务器端确认。

```

✓ Transmission Control Protocol, Src Port: 61573, Dst Port: 8080, Seq: 0, Len: 0
  Source Port: 61573
  Destination Port: 8080
  [Stream index: 7]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 0]
  Sequence Number: 0 (relative sequence number)

```

图 3.10: 第一次握手 port

3.3.2 第二次握手

第二次握手:

服务器端收到数据包后由标志位 SYN=1 知道客户端请求建立连接, 服务器端将 TCP 报文标志位 SYN 和 ACK 都置为 1, $ack=J+1$, 随机产生一个序号值 $seq=K$, 并将该数据包发送给客户端以确认连接请求, 服务器端进入 SYN_RCVD 状态。

```

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

```

图 3.11: 第二次握手 SYN 与 ACK

在这里需要明确一个概念就是:

上面写的 ack 和 ACK, 不是同一个概念:

小写的 ack 代表的是头部的确认号 Acknowledge number, 缩写 ack, 是对上一个包的序号进行确认的号, $ack=seq+1$ 。大写的 ACK, 则是我们上面说的 TCP 首部的标志位, 用于标志的 TCP 包是否对上一个包进行了确认操作, 如果确认了, 则把 ACK 标志位设置成 1。

可以看到由于第二次握手的信息是由服务端发出的, 所以 relative sequence 被重置了, 下一个 ACK 以及 sequence 也是一样的。

```

Sequence Number: 0 (relative sequence number)
Sequence Number (raw): 1926731891
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 2368792516

```

图 3.12: 第二次握手 sequence

3.3.3 第三次握手

第三次握手:

客户端收到确认后, 检查 ack 是否为 J+1, ACK 是否为 1, 如果正确则将标志位 ACK 置为 1, ack=K+1, 并将该数据包发送给服务器端, 服务器端检查 ack 是否为 K+1, ACK 是否为 1, 如果正确则连接建立成功, 客户端和服务端进入 ESTABLISHED 状态, 完成三次握手, 随后客户端与服务端之间可以开始传输数据了。

```

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

```

图 3.13: 第三次握手的 flag 位

并且我们可以与第一次握手作比较, 可以发现 sequence 确实被成功加上了:

```

Sequence Number: 1 (relative sequence number)
Sequence Number (raw): 2368792516
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 1926731892
1000 .... = Header Length: 32 bytes (8)

```

图 3.14: 第三次握手的 flag 位

3.4 http 报文分析

3.4.1 页面请求

首先, 如下图所示, 客户端向服务端发送了一个请求。并得到了服务端的相应

1188	12.271553	127.0.0.1	127.0.0.1	HTTP	781 GET /myvieww HTTP/1.1
1189	12.271623	127.0.0.1	127.0.0.1	TCP	56 8080 → 61573 [ACK] Seq=1 Ack=726 Win=326656 Len=0 TSval=229464770 TSecr=229464770
1250	12.793116	127.0.0.1	127.0.0.1	TCP	2715 8080 → 61573 [PSH, ACK] Seq=1 Ack=726 Win=326656 Len=2659 TSval=229465291 TSecr=229464770 [TCP segment]
1251	12.793200	127.0.0.1	127.0.0.1	TCP	56 61573 → 8080 [ACK] Seq=726 Ack=2660 Win=2616576 Len=0 TSval=229465291 TSecr=229465291

图 3.15: 请求页面

点开查看详细的抓包信息, 可以看到详细的请求头, 其中第一行说明了请求方法为 GET, 以及请求的 url 的相对路径以及使用的 http 的版本, 也就是之前介绍过的 http1.1, 随后就是对个以键值对形式出现的请求头, 在请求头最后还有一个独立的

r

n 来表示请求头部分已经结束。

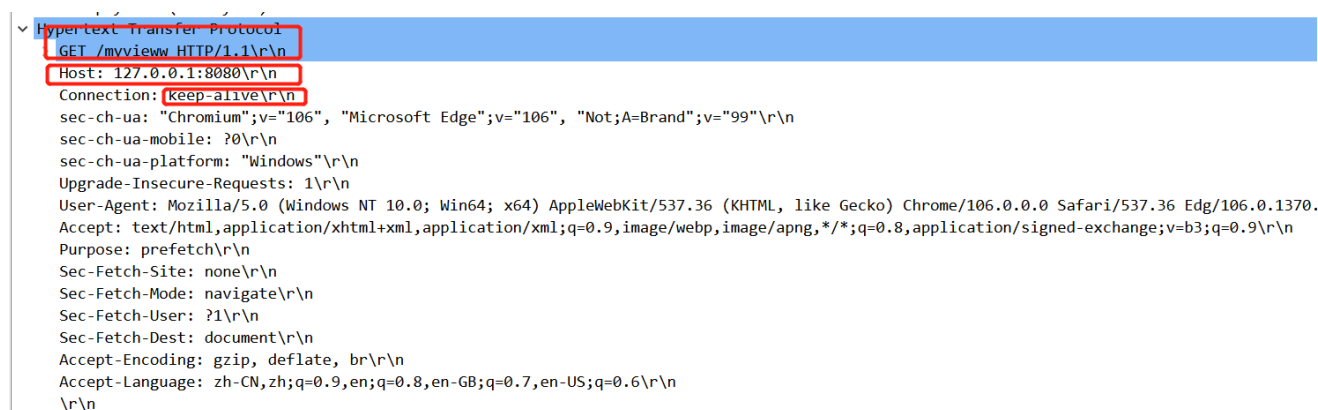


图 3.16: 请求头

而在随后的交互中我们看到了服务端将网页的数据交还给了客户端，可以看到服务端把要展示的 html 文档的内容以 asc 码的形式返回给了客户端

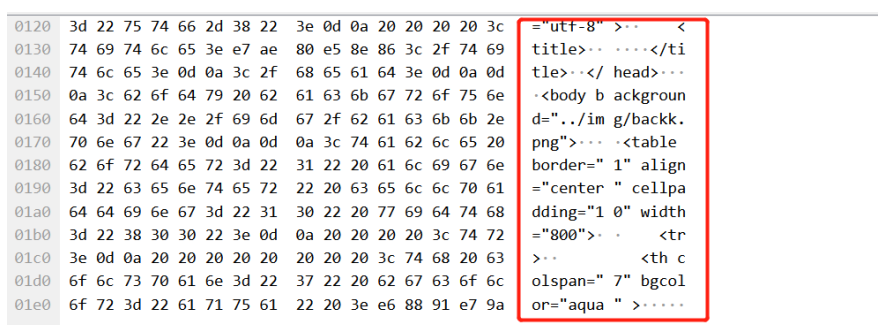


图 3.17: 数据传回

并且在这里，服务端设置了一个非常重要的 flag 就是 PSH，其代表的英文含义就是 PUSH。

如果使用 PSH 标志，上面这件事就确认下来了：

对于发送端来说，由 TCP 模块自行决定，何时将接收缓冲区中的数据打包成 TCP 报文，并加上 PSH 标志（在图 1 中，为了演示，我们假设人为的干涉了 PSH 标志位）。一般来说，每一次 write，都会将这一次的数据打包成一个或多个 TCP 报文段（如果数据量大于 MSS 的话，就会被打包成多个 TCP 段），并将最后一个 TCP 报文段标记为 PSH。

当然上面说的只是一般的情况，如果发送缓冲区满了，TCP 同样会将发送缓冲区中的所有数据打包发送。

对于接收端来说，如果接收方接收到了某个 TCP 报文段包含了 PSH 标志，则立即将缓冲区中的所有数据推送给应用进程（read 函数返回）。当然有时候接收缓冲区满了，也会推送。

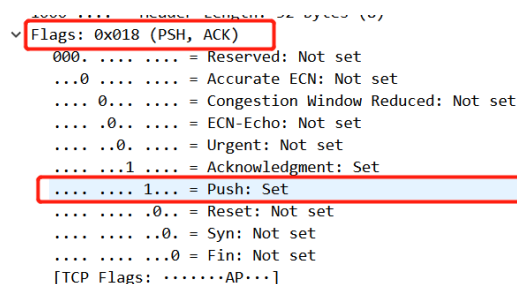


图 3.18: PSH 标志

在页面内容交互后，服务端将向客户端返回一个响应报文。

1252	12.794327	127.0.0.1	127.0.0.1	HTTP	61 HTTP/1.1 200 (text/html)
1253	12.794366	127.0.0.1	127.0.0.1	TCP	56 61573 → 8080 [ACK] Seq=726 Ack=2665 Win=2616576 Len=0 TSval=229465293 TSecr=229465293
1264	12.815942	127.0.0.1	127.0.0.1	HTTP	702 GET /img/wbf.png HTTP/1.1
1265	12.816053	127.0.0.1	127.0.0.1	TCP	56 8080 → 61573 [ACK] Seq=2665 Ack=1372 Win=2619136 Len=0 TSval=229465314 TSecr=229465314
1266	12.839565	127.0.0.1	127.0.0.1	TCP	8248 8080 → 61573 [PSH, ACK] Seq=2665 Ack=1372 Win=2619136 Len=8192 TSval=229465338 TSecr=229465314 [TCP s
1267	12.839669	127.0.0.1	127.0.0.1	TCP	56 61573 → 8080 [ACK] Seq=1372 Ack=10857 Win=2608384 Len=0 TSval=229465338 TSecr=229465338

图 3.19: 响应报文

响应报文的内容与请求报文基本类似，主要包括使用的 http 版本，状态码（200 表示正常返回了）也包括内容的类型、使用的语言、更改的日期以及连接的类型等等。

```

Hypertext Transfer Protocol
> HTTP/1.1 200 \r\n
Content-Type: text/html;charset=UTF-8\r\n
Content-Language: zh-CN\r\n
Transfer-Encoding: chunked\r\n
Date: Thu, 27 Oct 2022 07:03:26 GMT\r\n
Keep-Alive: timeout=60\r\n
Connection: keep-alive\r\n
\r\n
[HTTP response 1/4]
[Time since request: 0.522774000 seconds]
[Request in frame: 1188]
[Next request in frame: 1264]
[Next response in frame: 1268]
[Request URI: http://127.0.0.1:8080/myview]
> HTTP chunked response
File Data: 2458 bytes
> Line-based text data: text/html (80 lines)

```

图 3.20: 响应报文

后面的过程与前面基本一致，也就是客户端请求包含请求头，服务端相应，并且在返回后返回回头。后面对网页中使用到的图片进行了请求

1252	12.794327	127.0.0.1	127.0.0.1	HTTP	61 HTTP/1.1 200 (text/html)
1253	12.794366	127.0.0.1	127.0.0.1	TCP	56 61573 → 8080 [ACK] Seq=726 Ack=2665 Win=2616576 Len=0 TSval=229465293 TSecr=229465293
1264	12.815942	127.0.0.1	127.0.0.1	HTTP	702 GET /img/wbf.png HTTP/1.1
1265	12.816053	127.0.0.1	127.0.0.1	TCP	56 8080 → 61573 [ACK] Seq=2665 Ack=1372 Win=2619136 Len=0 TSval=229465314 TSecr=229465314
1266	12.839565	127.0.0.1	127.0.0.1	TCP	8248 8080 → 61573 [PSH, ACK] Seq=2665 Ack=1372 Win=2619136 Len=8192 TSval=229465338 TSecr=229465314 [TCP s
1267	12.839669	127.0.0.1	127.0.0.1	TCP	56 61573 → 8080 [ACK] Seq=1372 Ack=10857 Win=2608384 Len=0 TSval=229465338 TSecr=229465338
1268	12.839793	127.0.0.1	127.0.0.1	HTTP	195 HTTP/1.1 200 (PNG)
1269	12.839820	127.0.0.1	127.0.0.1	TCP	56 61573 → 8080 [ACK] Seq=1372 Ack=10996 Win=2608128 Len=0 TSval=229465338 TSecr=229465338
1374	13.535892	127.0.0.1	127.0.0.1	TCP	64 61584 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=229466034 TSecr=0
1375	13.535990	127.0.0.1	127.0.0.1	TCP	64 8080 → 61584 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=229466034 TSecr=0
1376	13.536095	127.0.0.1	127.0.0.1	TCP	56 61584 → 8080 [ACK] Seq=1 Ack=1 Win=2619136 Len=0 TSval=229466034 TSecr=229466034
1395	13.638675	127.0.0.1	127.0.0.1	HTTP	690 GET /img/backk.png HTTP/1.1
1396	13.638788	127.0.0.1	127.0.0.1	TCP	56 8080 → 61573 [ACK] Seq=10996 Ack=2006 Win=2618624 Len=0 TSval=229466137 TSecr=229466137
1397	13.648219	127.0.0.1	127.0.0.1	TCP	8248 8080 → 61573 [PSH, ACK] Seq=10996 Ack=2006 Win=2618624 Len=8192 TSval=229466146 TSecr=229466137 [TCP s

图 3.21: 后续流程

3.4.2 四次挥手

顾名思义，就是在关闭连接的时候双方一共要操作四次，以客户端主动请求断开连接为例：下面首先展示四次挥手的示意图：

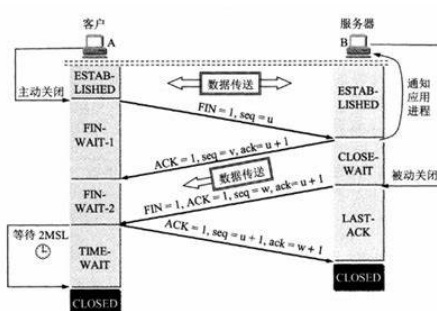


图 3.22: 四次挥手流程图

而由于浏览器开启了多线程，我们通过把握分析仪展示其中一个线程的关闭流程：

3596	33.576465	2001:250:401:6571:d...	2001:250:401:6571:d...	TCP	64	61625 → 8080	[FIN, ACK] Seq=1 Ack=1 Win=2618880 Len=0
3597	33.576484	2001:250:401:6571:d...	2001:250:401:6571:d...	TCP	64	8080 → 61625	[ACK] Seq=1 Ack=2 Win=2618880 Len=0
3598	33.576596	2001:250:401:6571:d...	2001:250:401:6571:d...	TCP	64	8080 → 61625	[FIN, ACK] Seq=1 Ack=2 Win=2618880 Len=0
3599	33.576635	2001:250:401:6571:d...	2001:250:401:6571:d...	TCP	64	61625 → 8080	[ACK] Seq=2 Ack=2 Win=2618880 Len=0

图 3.23: wireshark 观测结果

3.4.3 第一次挥手

第一次挥手：A 数据传输完毕需要断开连接，A 的应用进程向其 TCP 发出连接释放报文段（FIN = 1, 序号 seq = u），并停止再发送数据，主动关闭 TCP 连接，进入 FIN-WAIT-1 状态，等待 B 的确认。

下图展示了客户端主动请求关闭，并且将 FIN 位设置为 1

```

Flags: 0x011 (FIN, 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
.... .... ...1 = Fin: Set

```

图 3.24: 第一次挥手

3.4.4 第二次挥手

第二次挥手：B 收到连接释放报文段后即发出确认报文段（ACK=1, 确认号 ack=u+1, 序号 seq=v），B 进入 CLOSE-WAIT 关闭等待状态，此时的 TCP 处于半关闭状态，A 到 B 的连接释放。而 A 收到 B 的确认后，进入 FIN-WAIT-2 状态，等待 B 发出的连接释放报文段。

下图展示了服务端接受到了客户端的关闭连接请求并发送确认

```

0101 .... = Header Length: 20 bytes (5)
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....]

```

图 3.25: 第二次挥手

3.4.5 第三次挥手

第三次挥手：当 B 数据传输完毕后，B 发出连接释放报文段（FIN = 1, ACK = 1, 序号 seq = w, 确认号 ack=u+1），B 进入 LAST-ACK（最后确认）状态，等待 A 的最后确认。

服务端发送连接释放报文段并等待客户端最后的确认

```

0101 .... = Header Length: 20 bytes (5)
v Flags: 0x011 (FIN, 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
  .... .... ...1 = Fin: Set
> [TCP Flags: .....A...F]

```

图 3.26: 第三次挥手

3.4.6 第四次挥手

第四次挥手: A 收到 B 的连接释放报文段后, 对此发出确认报文段 ($ACK = 1, seq=u+1, ack=w+1$), A 进入 TIME-WAIT (时间等待) 状态。此时 TCP 未释放掉, 需要经过时间等待计时器设置的时间 $2MSL$ 后, A 才进入 CLOSE 状态。

客户端确认释放, 并传送给服务端

```

v 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.....]

```

图 3.27: 第四次挥手

之所以要等待两个时间等待是因为要保证 A 发送的最后一个 ACK 报文段能够到达 B, 保证 A、B 正常进入 CLOSED 状态。这个 ACK 报文段有可能丢失, 使得处于 LAST-ACK 状态的 B 收不到对已发送的 FIN+ACK 报文段的确认, B 超时重传 FIN+ACK 报文段, A 能 $2MSL$ 时间内收到这个重传的 FIN+ACK 报文段, 接着 A 重传一次确认, 同时重启 $2MSL$ 计数器, $2MSL$ 时间后 A 和 B 进入 CLOSE 状态, 如果 A 在 TIME-WAIT 状态时接收到 B 的 FIN+ACK 报文段之后向 B 发出确认报文段, 而不再确认 B 是否收到立即进入 CLOSED 状态, 如若 B 并没有正常收到 A 的确认报文段, 则 B 无法正常进入到 CLOSED 状态。

其实, 不一定只有客户端才能发送释放连接请求, 所以说下面这个模型会更加常见

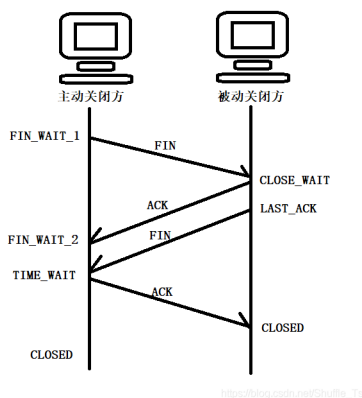


图 3.28: 四次挥手模型

4 问题解决及分析

在本次实验中，由于网络情况比较复杂，可能会出现比较多状况外的一些数据包，我在查看 ppt 以及进一步查阅资料后对下面的问题做出了如下回答：

4.1 浏览器缓存

我发现，当我第一次请求访问我的页面时发现本地的端口访问 8080 端口以请求图像，可是关闭浏览器再次进行实验并抓包分析的时候却看不到了，更奇怪的是，没请求但是图像却完好无损的呈现在了浏览器上。一开始我以为是图像放在了 maven 工程中的 static 里，可是我发现当我换了个路径还是一样的效果。

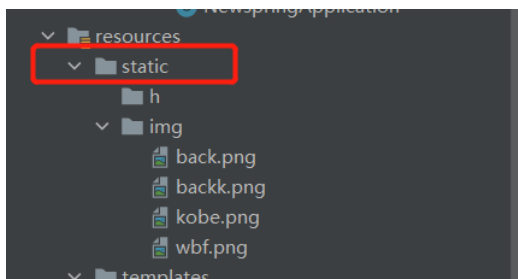


图 4.29: maven 项目目录

于是我翻阅课程 ppt，看到了 ppt 上讲缓存，于是我想到可能是浏览器缓存的原因，于是我更改了浏览器的对应设置，让每次浏览器关闭时都要清理缓存，于是每次加载时对于图片的请求就有恢复了



图 4.30: 更改浏览器缓存

4.2 304 状态码

在某一次实验过程中，我的抓包数据出现了 304，一般都是 200 或者 404 的，但这次出现了 304，比较罕见，我也是翻看了 ppt，其中有一页就是讲述状态码的，304 的含义其实就是：客户端在请求一个文件的时候，发现自己缓存的文件有 Last Modified，那么在请求中会包含 If Modified Since，这个时间就是缓存文件的 Last Modified。因此，如果请求中包含 If Modified Since，就说明已经有缓存存在客户端。服务端只要判断这个时间和当前请求的文件的修改时间就可以确定是返回 304 还是 200。

所以说 HTTP 304，有时也称为“304 Not Modified”，是一种与浏览器通信的代码：“自上次访问以来，请求的资源未被修改”

4.3 多端口

从下图我们可以看到，其实在建联过程中总共有两个端口都对 8080 进行了三次握手，那么为什么需要两个端口呢，根据上课老师讲授的内容，其实最多可以允许 6 个端口进行建联的，这是为了保证数据传输的效率，后面我们也会分析到，如果数据很多的话，多个端口并行传输可以一定程度上解决引用层的头阻塞问题。

188	22.905673	127.0.0.1	127.0.0.1	TCP	64 49949 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=332400714 TSecr=0
189	22.905759	127.0.0.1	127.0.0.1	TCP	64 8080 → 49949 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=332400714 TSecr=0
190	22.905905	127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1 Ack=1 Win=2619136 Len=0 TSval=332400714 TSecr=332400714
191	22.907134	127.0.0.1	127.0.0.1	TCP	64 49950 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=332400715 TSecr=0
192	22.907217	127.0.0.1	127.0.0.1	TCP	64 8080 → 49950 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM TSval=332400716 TSecr=0
193	22.907324	127.0.0.1	127.0.0.1	TCP	56 49950 → 8080 [ACK] Seq=1 Ack=1 Win=261888 Len=0 TSval=332400716 TSecr=332400716

图 4.31: 多端口建联

4.4 大量 PSH 的出现

在抓包过程中，我观察到了很多 PSH 状态的出现，仔细观察发现是发生在一个图片请求之后的，并且我们可以观察到，客户端和服务端的交互状态是一个两极分化的过程，客户端的 seq 不增长但是 ack 增长的非常快，服务端的 ack 不增长但是 seq 增长的非常快，根据前面我们对三次握手和四次挥手的分析，我们不难得出这个结论：

seq 号不变，ack 号一直在变大，说明 A 一直在收 B 的数据，一直在给 B 应答

seq 号一直变大，ack 号一直没变，说明 A 一直在向 B 发数据，不用给 B 应答，而是在等 B 的应答

所以说我的猜测是由于图片的大小太过于庞大，所以需要多次分批传输，才造成了这样的结果。

127.0.0.1	127.0.0.1	HTTP	690 GET /img/backup.png HTTP/1.1
127.0.0.1	127.0.0.1	TCP	56 8080 → 49949 [ACK] Seq=10996 Ack=1973 Win=2617856 Len=0 TSval=332401332 TSecr=332401332
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=10996 Ack=1973 Win=2617856 Len=8192 TSval=332401338 TSecr=332401332 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=19188 Win=2599936 Len=0 TSval=332401338 TSecr=332401338
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=19188 Ack=1973 Win=2617856 Len=8192 TSval=332401338 TSecr=332401338 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=27380 Win=2591744 Len=0 TSval=332401338 TSecr=332401338
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=27380 Ack=1973 Win=2617856 Len=8192 TSval=332401338 TSecr=332401338 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=35572 Win=2583552 Len=0 TSval=332401338 TSecr=332401338
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=35572 Ack=1973 Win=2617856 Len=8192 TSval=332401338 TSecr=332401338 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=43764 Win=2575360 Len=0 TSval=332401338 TSecr=332401338
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=43764 Ack=1973 Win=2617856 Len=8192 TSval=332401338 TSecr=332401338 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=51956 Win=2567168 Len=0 TSval=332401338 TSecr=332401338
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=51956 Ack=1973 Win=2617856 Len=8192 TSval=332401339 TSecr=332401338 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=60148 Win=2558976 Len=0 TSval=332401339 TSecr=332401339
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=60148 Ack=1973 Win=2617856 Len=8192 TSval=332401339 TSecr=332401339 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=68340 Win=2619136 Len=0 TSval=332401339 TSecr=332401339
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=68340 Ack=1973 Win=2617856 Len=8192 TSval=332401339 TSecr=332401339 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=76532 Win=2610944 Len=0 TSval=332401339 TSecr=332401339
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=76532 Ack=1973 Win=2617856 Len=8192 TSval=332401339 TSecr=332401339 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=84724 Win=2602752 Len=0 TSval=332401339 TSecr=332401339
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=84724 Ack=1973 Win=2617856 Len=8192 TSval=332401339 TSecr=332401339 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=92916 Win=2594560 Len=0 TSval=332401339 TSecr=332401339
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=92916 Ack=1973 Win=2617856 Len=8192 TSval=332401339 TSecr=332401339 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=101108 Win=2586368 Len=0 TSval=332401340 TSecr=332401340
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=101108 Ack=1973 Win=2617856 Len=8192 TSval=332401340 TSecr=332401340 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=109300 Win=2578176 Len=0 TSval=332401340 TSecr=332401340
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=109300 Ack=1973 Win=2617856 Len=8192 TSval=332401340 TSecr=332401340
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=117492 Win=2569984 Len=0 TSval=332401340 TSecr=332401340
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=117492 Ack=1973 Win=2617856 Len=8192 TSval=332401340 TSecr=332401340 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=125684 Win=2561792 Len=0 TSval=332401340 TSecr=332401340
127.0.0.1	127.0.0.1	TCP	8248 8080 → 49949 [PSH, ACK] Seq=125684 Ack=1973 Win=2617856 Len=8192 TSval=332401340 TSecr=332401340 [TCP
127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=1973 Ack=133876 Win=2619136 Len=0 TSval=332401341 TSecr=332401340

图 4.32: 图片分次传输

4.5 favicon

我发现在抓包数据中还包括 favicon 的内容，但是这个之前一直没有听说过，于是我搜索了互联网得到了一下结果：所谓 favicon，即 Favorites Icon 的缩写，是指显示在浏览器收藏夹、地址栏和标签标题前面的个性化图标。以图标的方式区别不同的网站

410 23.610796	127.0.0.1	127.0.0.1	HTTP	688 GET /favicon.ico HTTP/1.1
411 23.610892	127.0.0.1	127.0.0.1	TCP	56 8080 → 49949 [ACK] Seq=726437 Ack=2605 Win=2617344 Len=0 TSval=332401419 TSecr=332401419
412 23.725688	127.0.0.1	127.0.0.1	TCP	427 8080 → 49949 [PSH, ACK] Seq=726437 Ack=2605 Win=2617344 Len=371 TSval=332401534 TSecr=332401419 [TCP
413 23.725782	127.0.0.1	127.0.0.1	TCP	56 49949 → 8080 [ACK] Seq=2605 Ack=726808 Win=2616064 Len=0 TSval=332401534 TSecr=332401534

图 4.33: favicon

4.6 json

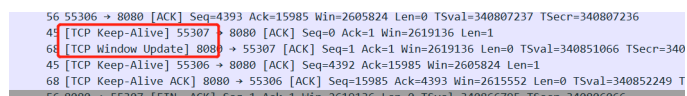
我还发现，每次抓包都会找到 json 文件，而且每次都是 404 失败，我的理解是，其实就是在网页阶段他会抓很多东西下来不止一个简单的 html，可能还会有 css、js 等等，但是由于本次实验只是用一个简单的 html，所以并没有抓到 json

2160 27.880165	127.0.0.1	127.0.0.1	TCP	56 65231 → 8080 [ACK] Seq=1 Ack=1 Win=261888 Len=0 TSval=340570299 TSecr=340570299
2167 27.883080	127.0.0.1	127.0.0.1	HTTP/1.1 404	61 HTTP/1.1 404 [JavaScript Object Notation (application/json)]
2168 27.883160	127.0.0.1	127.0.0.1	TCP	56 65231 → 8080 [ACK] Seq=2605 Ack=726813 Win=2616064 Len=0 TSval=340570302 TSecr=340570302
2171 27.823310	127.0.0.1	127.0.0.1	TCP	56 65233 → 8080 [FIN, ACK] Seq=1 Ack=1 Win=261888 Len=0 TSval=340570391 TSecr=340569567

图 4.34: json

4.7 Lkeep-alive

有的时候如果我的页面保持了很长时间，他会出现 keep-alive，这个东西还是一个交互的过程，就是说：在 TCP 中有一个 Keep-alive 的机制可以检测死连接，原理很简单，TCP 会在空闲了一定时间后发送数据给对方：如果主机可达，对方就会响应 ACK 应答，就认为是存活的。



```
56 55306 → 8080 [ACK] Seq=4393 Ack=15985 Win=2605824 Len=0 TSval=340807237 TSecr=340807236
45 [TCP Keep-Alive] 55307 → 8080 [ACK] Seq=0 Ack=1 Win=2619136 Len=1
68 [TCP Window Update] 8080 → 55307 [ACK] Seq=1 Ack=1 Win=2619136 Len=0 TSval=340851066 TSecr=340
45 [TCP Keep-Alive] 55306 → 8080 [ACK] Seq=4392 Ack=15985 Win=2605824 Len=1
68 [TCP Keep-Alive ACK] 8080 → 55306 [ACK] Seq=15985 Ack=4393 Win=2615552 Len=0 TSval=340852249 T
56 8080 → 55307 [FIN, RST] Seq=1 Ack=1 Win=2619136 Len=0 TSval=340866795 TSecr=340806066
```

图 4.35: “生命包”

5 总结

通过本次实验，我对应用层传输的传输有了一个比较清晰的认识，对 http 协议以及其演化有了深刻的了解，同时也通过亲自动手实验进行了查漏补缺，并根据实际实验中遇到的一些问题加深了课上知识的理解，也获取了很多新的知识。