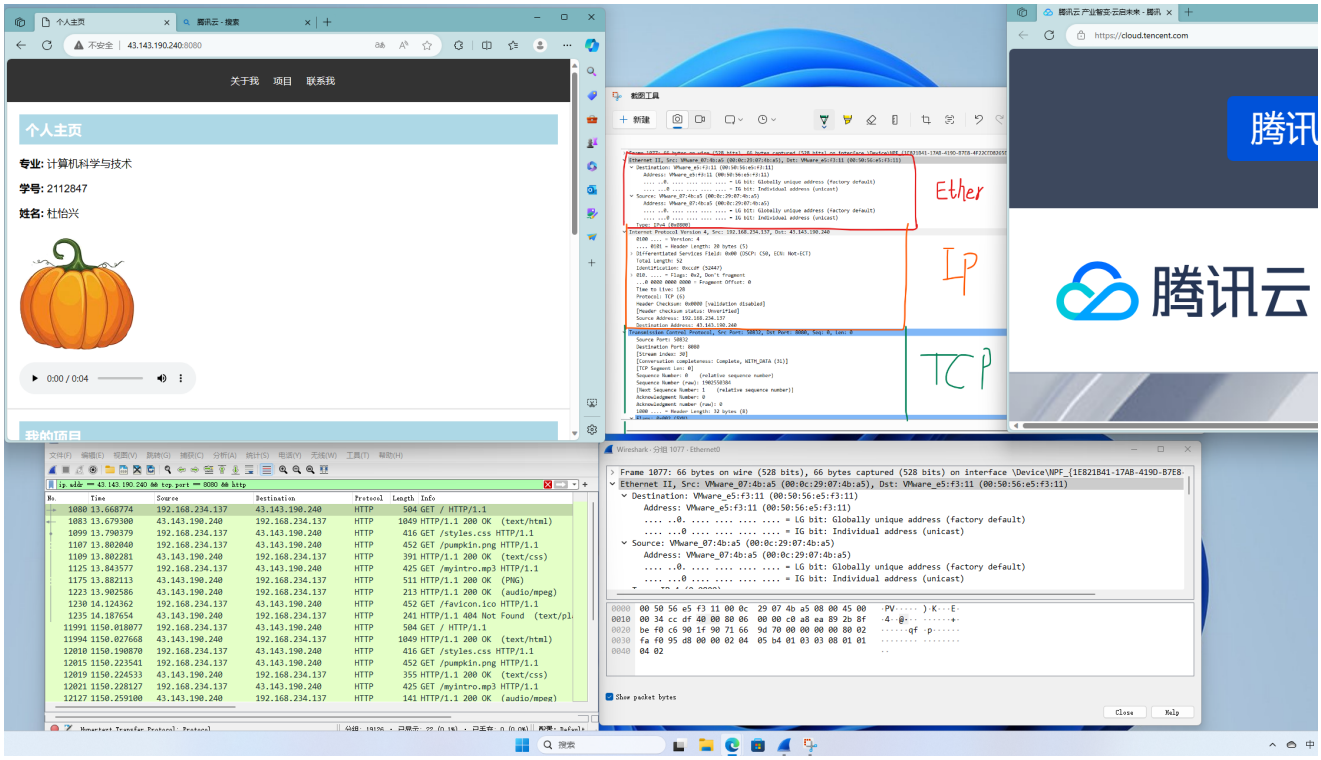
计网实验二:wire shark抓包分析

基于云服务器上的个人网站

2112847 杜怡兴



实验报告(22页)

1. 实验准备

1.1 实验要求

本次实验的主要要求包括：

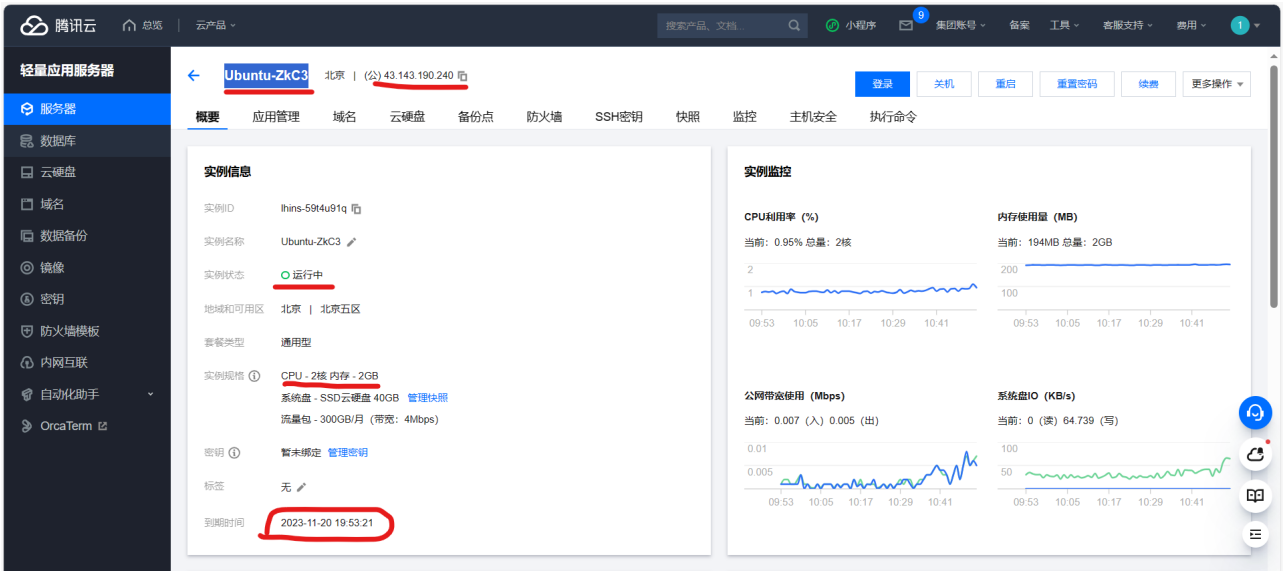
1. 搭建Web服务器，制作简单的Web页面，包含文本信息、LOGO、音频自我介绍。
2. 使用Wireshark捕获浏览器与Web服务器的交互过程，并进行简单的分析。
3. 使用HTTP协议，不使用HTTPS。

1.2 租用Web服务器

在本次实验中，我租用了腾讯云服务器，具体配置如下：

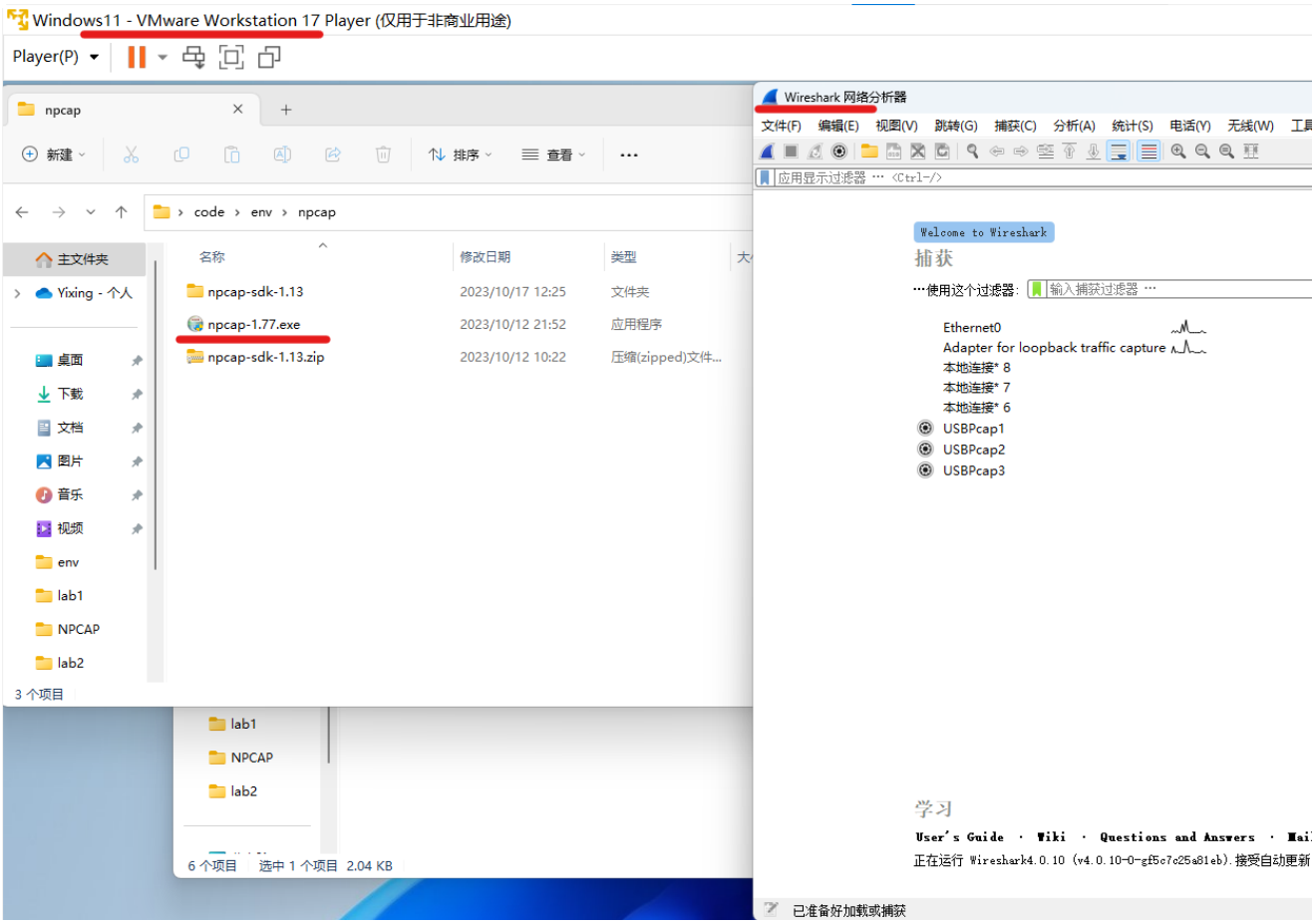
- 公网IP：http://43.143.190.240:8080/【在到期前，近几天内都可以访问】
- CPU & 内存：2核 2GiB
- 操作系统：Ubuntu-ZkC3

- 服务器软件：Node.js



1.3 安装Wireshark抓包工具

Wireshark是一款用于网络封包分析的软件。我使用的npcap版本是npcap 1.77，运行在Windows 11操作系统上。



2. 实验流程

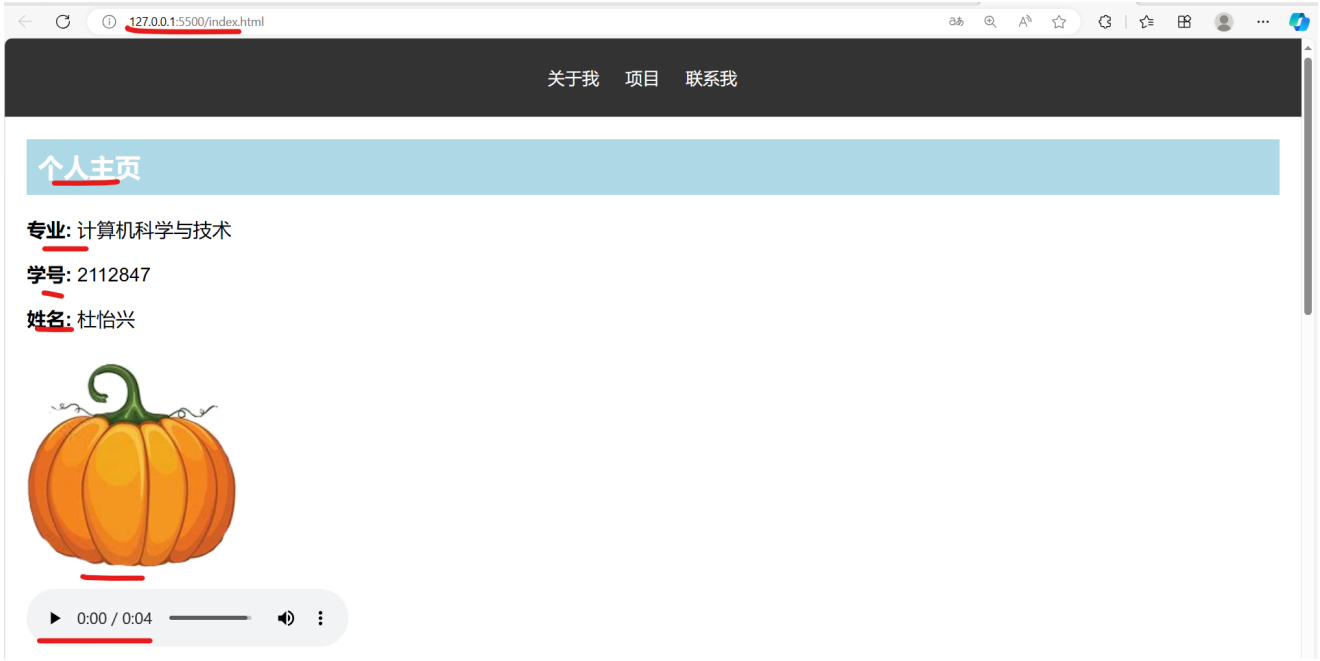
2.1 制作网页

我首先编写了一个基本的HTML文档，并使用CSS对其进行修饰。HTML文档中包含了文本信息、图片资源以及音频资源。
其中index.html如下

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>个人主页</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <nav>
            <ul>
                <li><a href="#about">关于我</a></li>
                <li><a href="#projects">项目</a></li>
                <li><a href="#contact">联系我</a></li>
            </ul>
        </nav>
    </header>
    <section id="about">
        <h1>个人主页</h1>
        <p><b>专业:</b> 计算机科学与技术</p>
        <p><b>学号:</b> 2112847</p>
        <p><b>姓名: </b>杜怡兴</p>
        <img src="./pumpkin.png" alt="我的LOGO">
        <br>
        <audio controls>
            <source src="./myintro.mp3" type="audio/mpeg">
            您的浏览器不支持音频播放。
        </audio>
    </section>
    <!-- 其他页面内容省略 -->
    <footer>
        2112847dyx的作业
    </footer>
</body>
</html>
```

具体的css和图片、音频文件见附件

用vscode的插件liveserver可以运行在本机的5500端口:

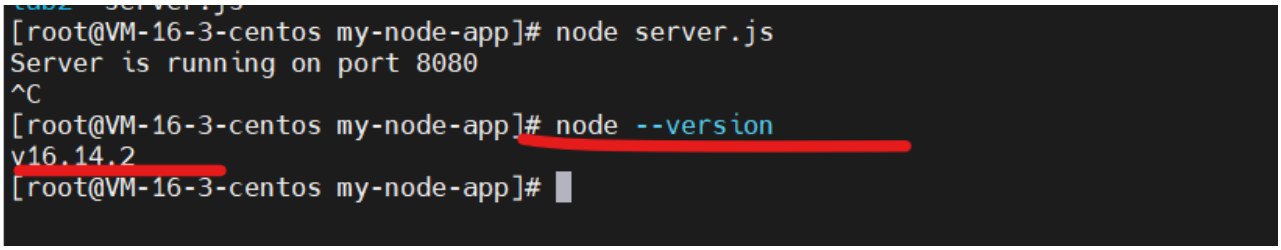


2.2搭建服务器

安装nodejs环境:

1. **安装Node.js之后查看版本**

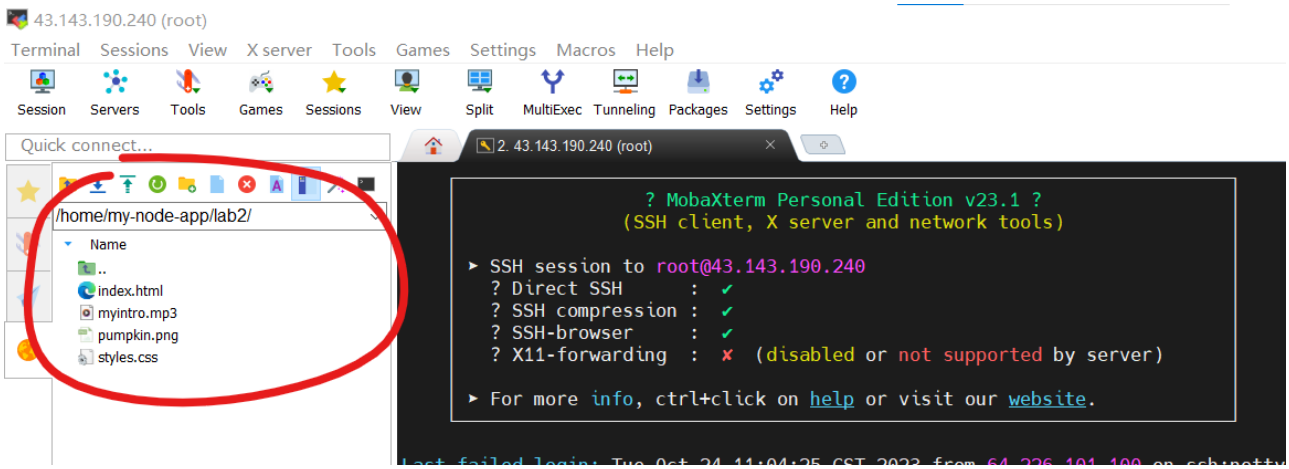
```
node --version
```



2. **创建项目目录**：在服务器上创建一个新的目录

```
mkdir my-node-app
cd my-node-app
```

3. **创建lab2/index.html文件**：将本机上的文件夹上传服务器。



4. **创建Node.js应用**：在项目目录中创建一个Node.js应用，用于启动HTTP服务器并提供`index.html`作为响应。创建一个名为`server.js`的JavaScript文件，内容如下：

```
const http = require('http');
const fs = require('fs');
const path = require('path');

const server = http.createServer((req, res) => {
  // 解析请求的URL，如果请求根路径("/")，则默认为请求index.html文件
  const url = req.url === '/' ? '/lab2/index.html' :'/lab2'+ req.url;
  const filePath = path.join(__dirname, url);

  // 检查文件是否存在
  fs.access(filePath, fs.constants.F_OK, (err) => {
    if (err) {
      // 如果文件不存在，返回404错误
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      res.end('404 Not Found');
      return;
    }

    // 获取文件的扩展名，用于设置正确的Content-Type
    const fileExtension = path.extname(filePath);
    let contentType = 'text/html'; // 默认为HTML

    // 根据扩展名设置Content-Type
    switch (fileExtension) {
      case '.html':
        contentType = 'text/html';
        break;
      case '.css':
        contentType = 'text/css';
        break;
      case '.js':
        contentType = 'text/javascript';
        break;
      case '.png':
        contentType = 'image/png';
        break;
      case '.jpg':
      case '.jpeg':
        contentType = 'image/jpeg';
        break;
      case '.mp3':
        contentType = 'audio/mpeg';
        break;
      // 添加其他可能的文件类型
    }


// 根据文件的扩展名决定是否使用'utf8'字符编码
const isBinaryFile = ['.jpg', '.jpeg', '.png', '.mp3'].includes(fileExtension);

// 读取文件并发送响应
fs.readFile(filePath, isBinaryFile ? null : 'utf8', (err, data) => {
  if (err) {
    // 如果读取文件出错，返回500错误
    res.writeHead(500, { 'Content-Type': 'text/plain' });
```

```
    res.end('Internal Server Error');
    return;
  }

  // 设置响应的Content-Type
  res.writeHead(200, { 'Content-Type': contentType });
  res.end(data);
});
  });
});

const port = 8080;
server.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

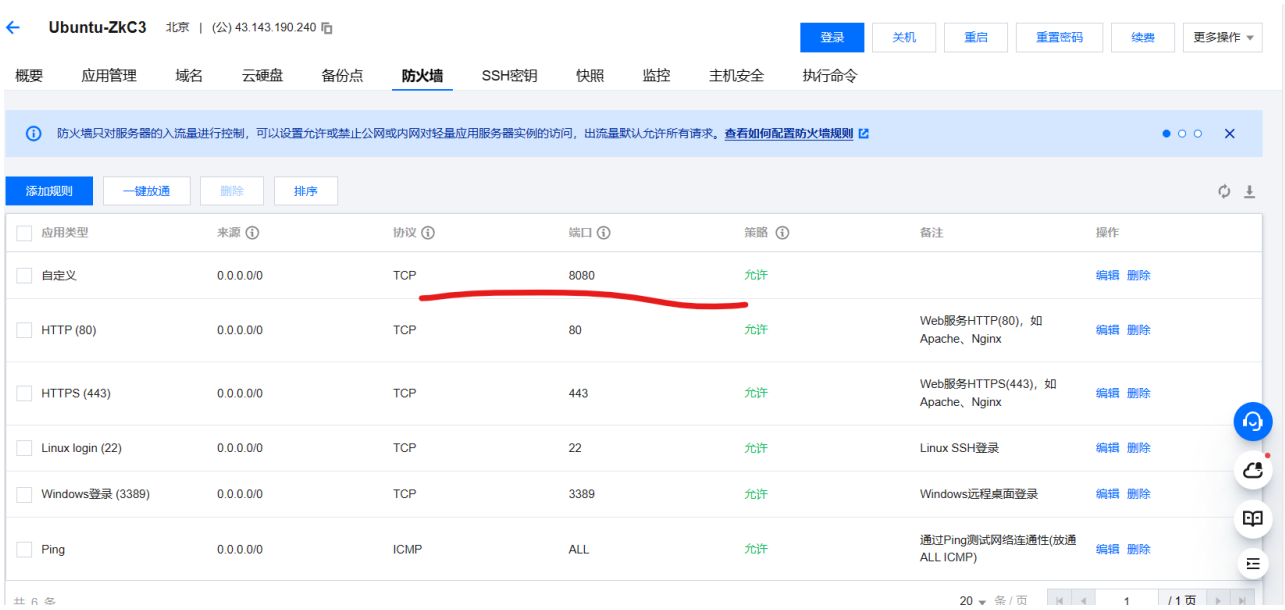
这个简单的Node.js应用创建了一个HTTP服务器，它会解析URL并根据URL的路径发送相应的文件作为响应。

5. **启动Node.js应用**：在项目目录中运行以下命令来启动Node.js应用：

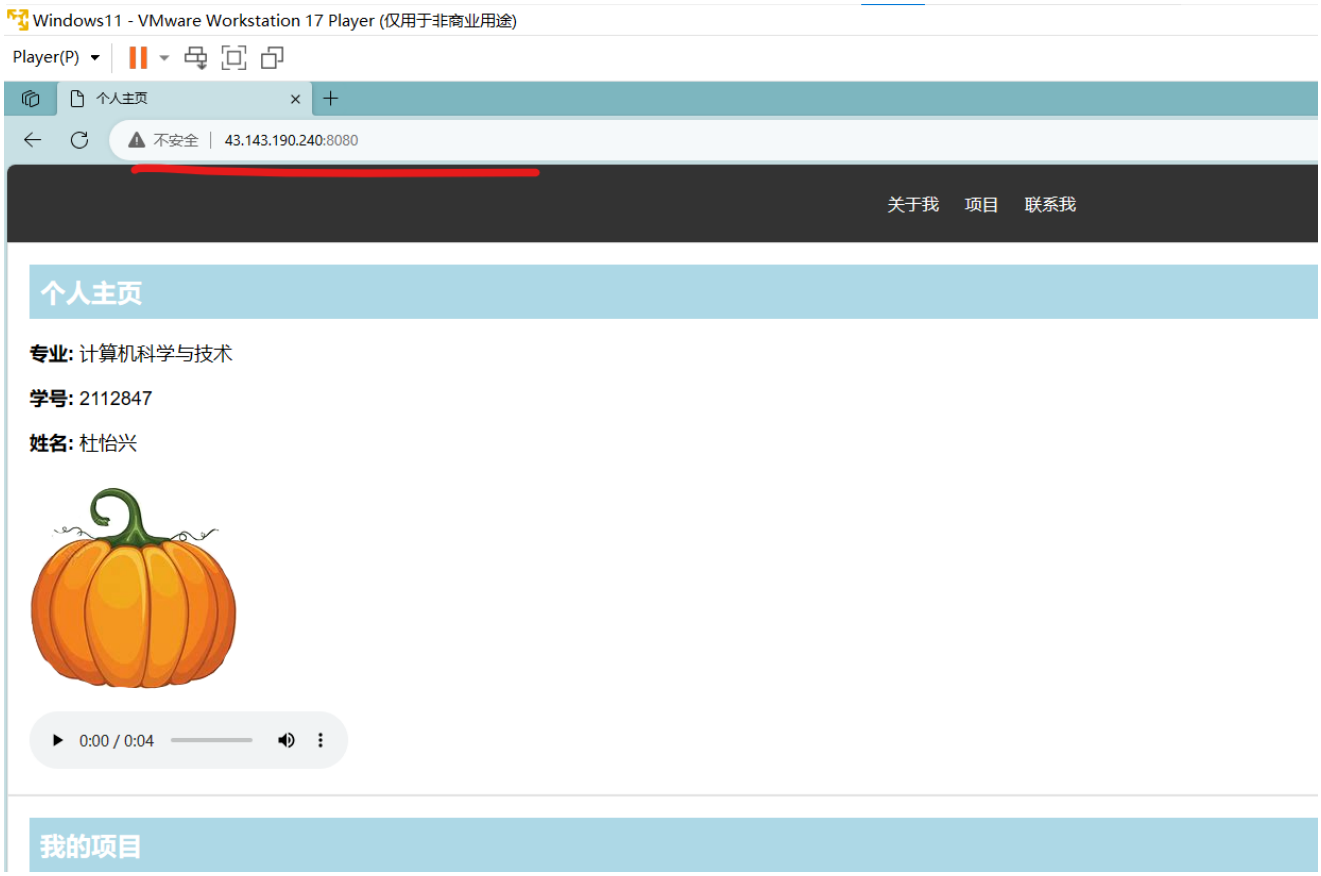
```
node server.js
```

```
[root@VM-16-3-centos my-node-app]# node server.js
Server is running on port 8080
```

6. **配置云服务器防火墙规则**：在腾讯云配置安全组，里面加入8080端口



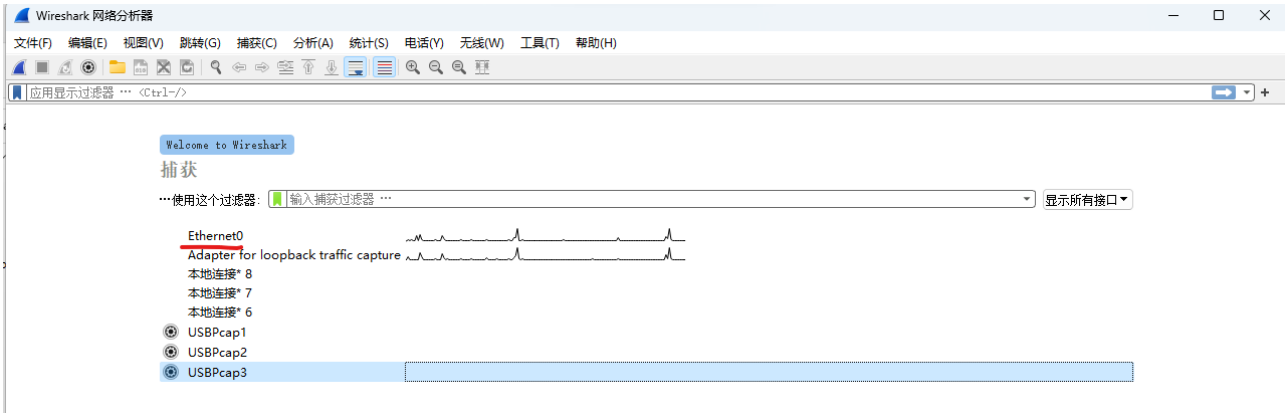
7. **访问网站**：现在，可以使用云服务器的公网IP地址，在浏览器中访问`http://公网IP地址:8080`，得到如下响应



2.3 抓包

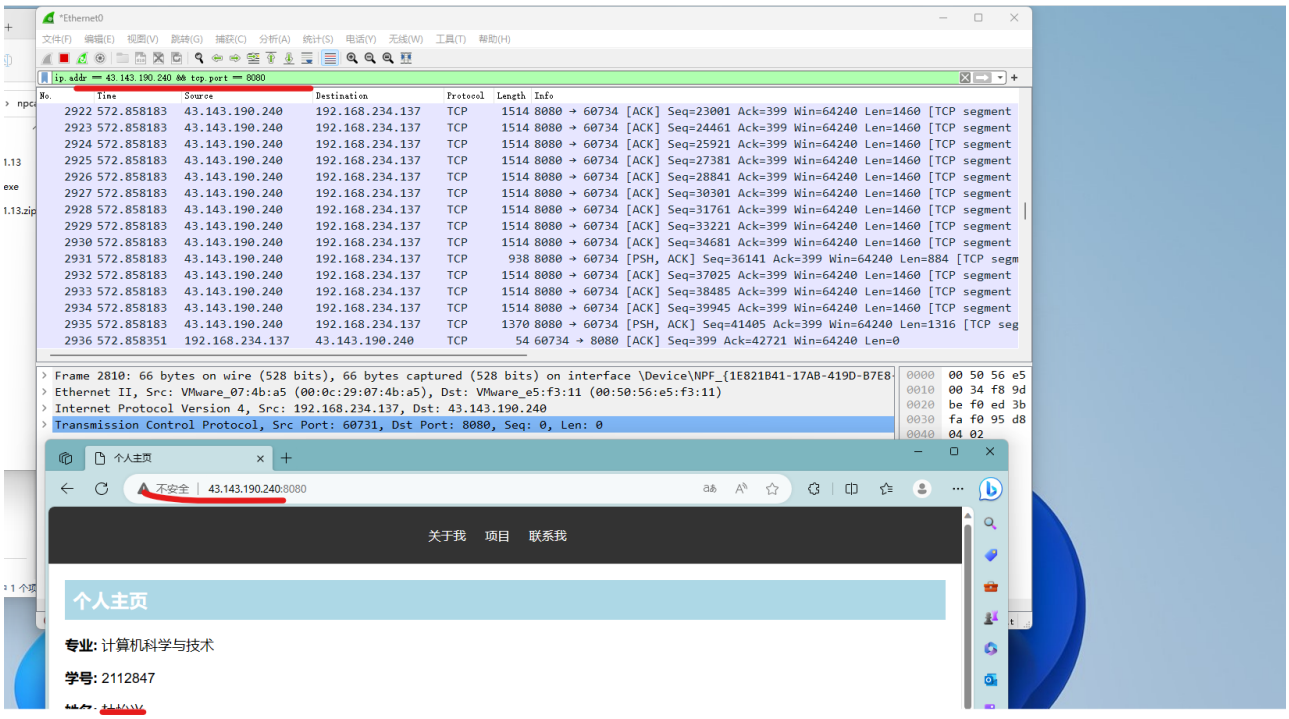
在本次实验中，使用Wireshark对浏览器与Web服务器的交互过程进行抓包和分析。以下是抓包的流程：

1. 选择接口：由于Web服务器有公网ip且不是运行在本机上的，因此选择Ethernet0接口。



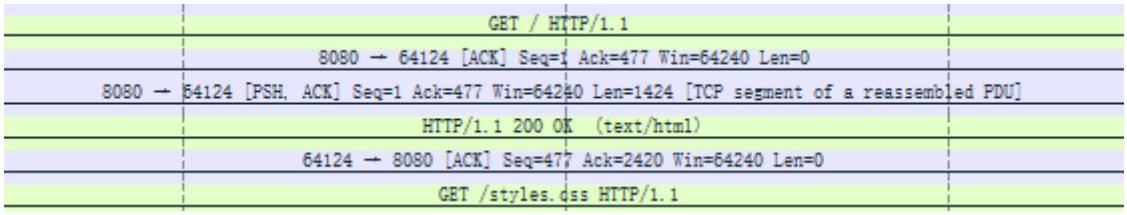
2. 过滤数据包：通过添加关键字段值进行过滤，输入`ip.addr == 43.143.190.240 && tcp.port == 8080`以捕获与目标服务器有关的数据包。

在edge浏览器打开个人主页前，wire shark没有捕获到数据包，打开后wires hark获取到大量数据包。

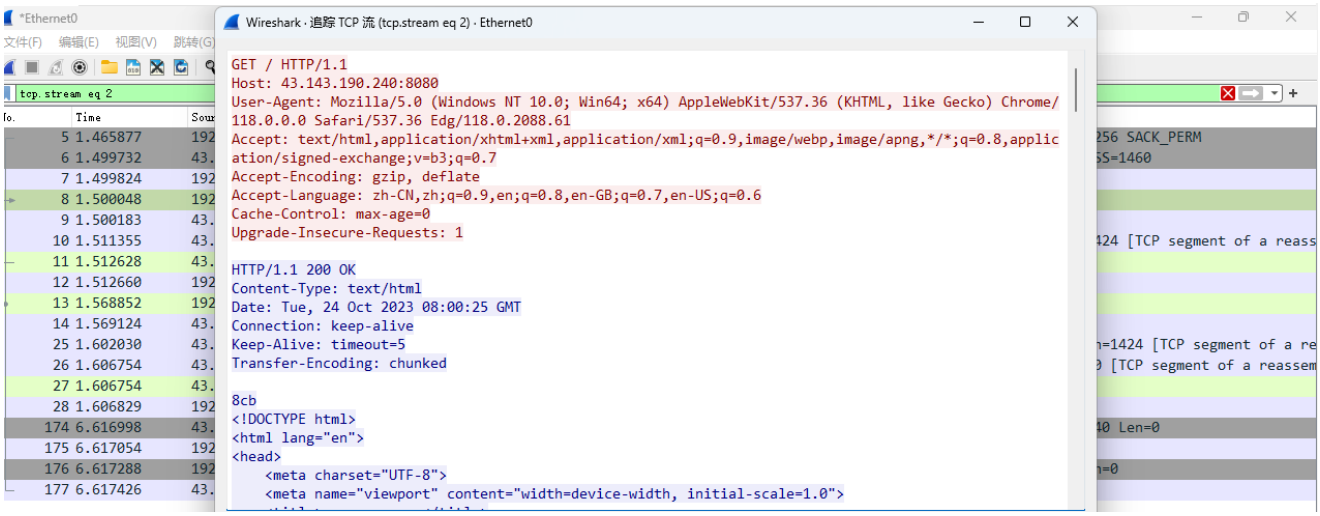


于此同时，为了以防万一，也可以在本地开启5500端口作为Web服务器，然后wires hark选择本地回环，然后选择5500端口进行监听。不过本实验为了普遍性，还是采取监听公网ip。

3. 可以通过点击统计>流量图 打开流量图工具。流量图分析：使用Wireshark的流量图工具，可以直观地查看每个TCP流中各帧的流向和时序。



4. 还可以右键>追踪流>TCP流，以把此次建联的TCP流完整的显示出来



3.实验结果

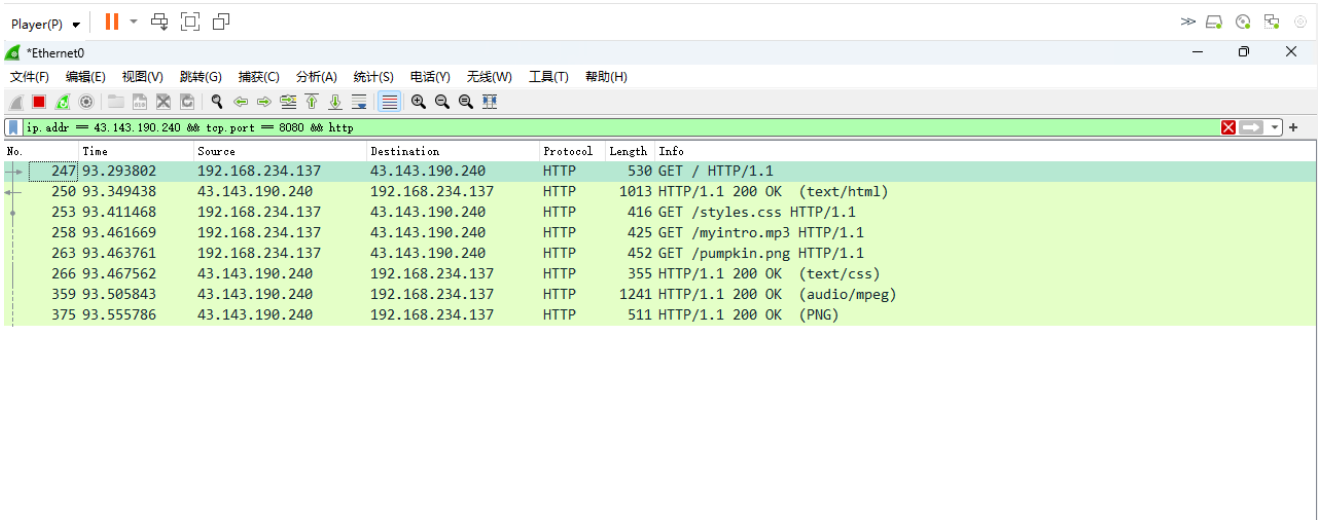
3.1导出数据包

由于实验要求当中提到，使用Wireshark过滤器使其仅显示HTTP协议，提交捕获文件（20分）

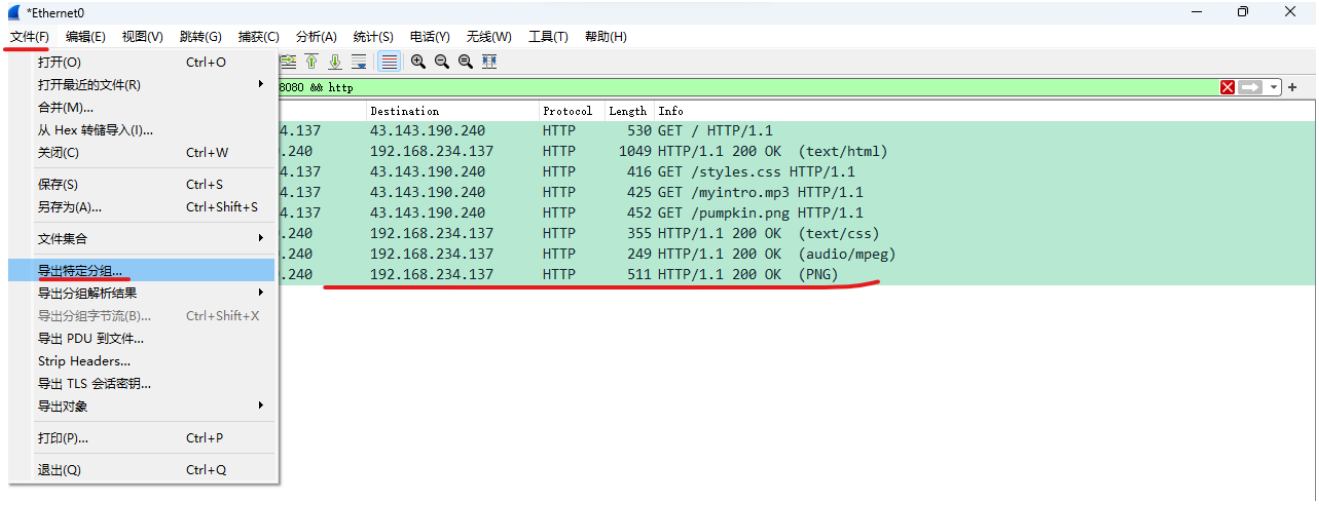
因此我设置捕获条件为ip.addr == 43.143.190.240 && tcp.port == 8080 && http

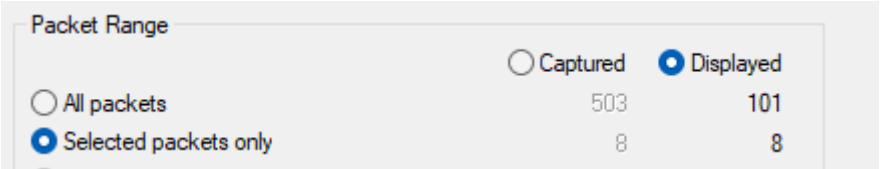
```
ip.addr == 43.143.190.240 && tcp.port == 8080 && http
```

此时得到了和个人网站交互的http数据包

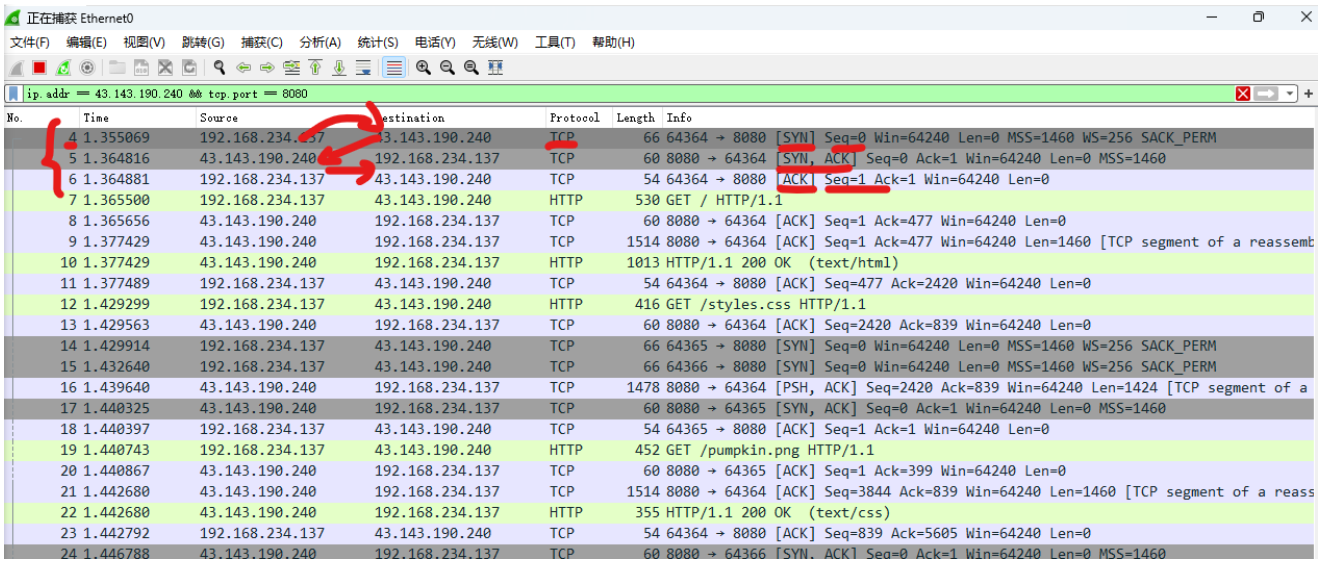


导出的时候，鼠标右键+ctrl全部依次选中，然后暂停捕获，点击文件>导出特定分组>只导出选中的数据包





3.2分析数据包:三次握手



1. **数据包4**: `192.168.234.137`（也就是我）向Web服务器`43.143.190.240`发送了一个TCP **`SYN`**（同步）数据包。这是建立连接的第一步。
2. **数据包5**: Web服务器`43.143.190.240`回复了一个**`SYN`，`ACK`**（同步，确认）数据包。这表示服务器已经接受了连接请求，并且已经准备好开始传输数据。
3. **数据包6**: `192.168.234.137`发送了一个**`ACK`**（确认）数据包，这标志着三次握手过程的完成，现在两台计算机之间的连接已经建立。

三次握手之后，建立TCP连接，我正式发起HTTP请求(类型是GET),版本是HTTP1.1，此时服务器会把index.html返回。我可以看到随后的HTTP请求和响应，以及其他的TCP数据包。

我的主机`192.168.234.137`首先发送了一个GET请求获取Web服务器上的网页（数据包7），服务器随后返回了请求的网页内容（数据包10）。在之后的数据包中，我可以看到其他的GET请求以及服务器的响应，如对`/styles.css`和`/pumpkin.png`,`myintro.mp3`的请求和响应等。

并且有意思的地方是，接收消息的顺序是：

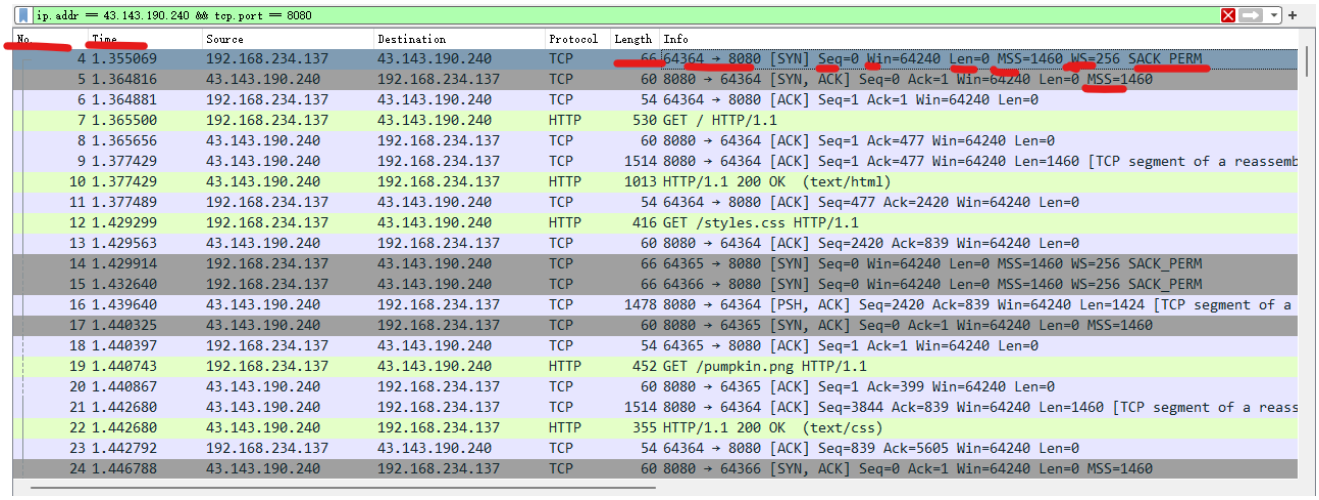
我GET请求，服务端发回index.html,我得知index.html 还需要引用css和图片，音频文件

1. 我GET请求css文件
2. 我GET请求图片
3. 我收到css文件
4. 我GET请求音频
5. 我收到图片
6. 我收到音频

并且收到音频和图片中间耗时很长，主要是因为音频和图片包含更多数据。

3.3分析数据包：其他信息的意义

在数据包当中，还有其他如下信息：



1.No. (序号)

- **取值含义**：表示该数据包在捕获的整个数据包序列中的顺序。

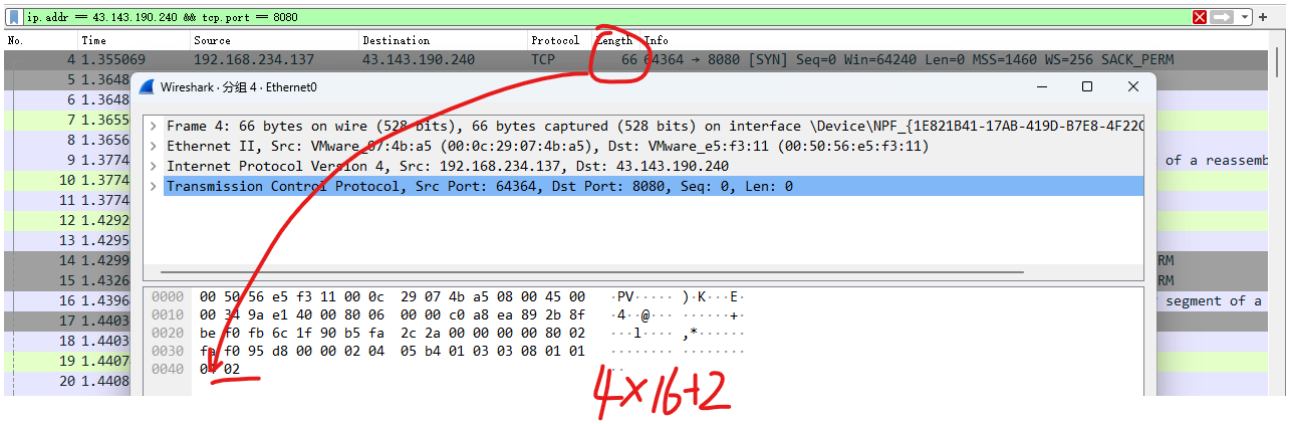
例如一开始监听就访问Web服务器，此时第一个数据包序号就很小。过了很久才访问，序号就很大。

2.Time (时间)

- **取值含义**：以秒为单位表示，是从数据包捕获开始计算的相对时间。
  这一系列数据包都是开始监听后大约2s内捕获的。

3.Length (长度)

- **取值含义**：表示该数据包的总字节大小。



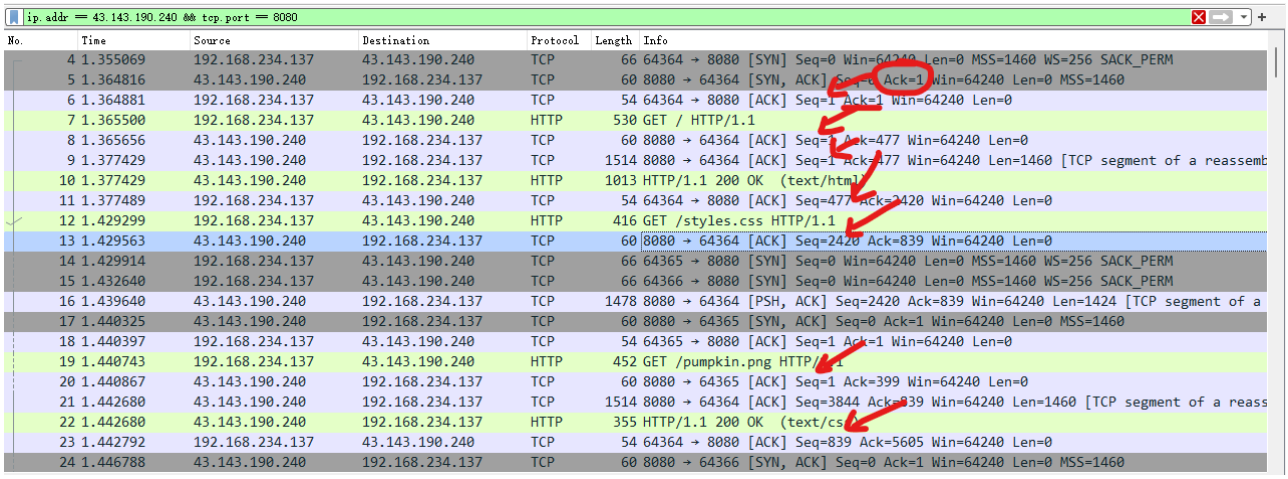
例如数据包总共有4*16+2=66个字节，对应长度就是66

4.Info (信息)

- **取值含义**：包含如下字段：
  - **8080->64364,**8080是Web服务器端口号，64364是客户端的临时端口号。
  - 当实际的Web内容（如GET请求、POST请求、HTTP响应等）被传输时，会看到**HTTP**。

    当建立连接、终止连接、或进行纯粹的数据传输时，通常会看到**TCP**。
  - **Ack** (Acknowledgment Number，确认号)：表示期望对方下一个报文的Sequence number。它用来确认已接收的数据。

    在图中，经常见到一方发送的Ack(例如477)对应另一方发回的Seq(477)



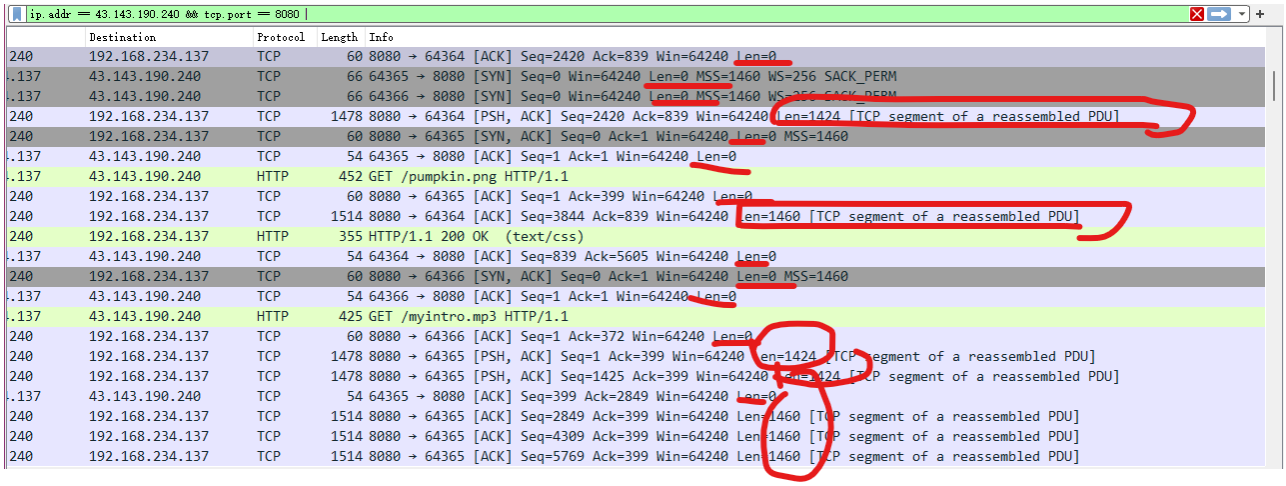
  - **Seq** (Sequence Number，序列号)：每个TCP数据包都有一个序列号，它表示该数据包中的第一个字节的编号。当开始一个新的TCP会话时，序列号是随机的。

假设A向B发送一个TCP段，其SEQ为100，并且携带10个字节的数据。

当B接收到这个段时，B会知道A发送的数据是从序列号100开始，总共有10个字节，所以最后一个字节的序列号是109。

B在回复确认时，会设置ACK为110，表示“I've received up to byte 109, so I'm expecting the next byte you send to be 110”。

  - **Win** (Window Size，窗口大小)：用于流量控制。表示接收方当前可用于接收数据的缓冲区大小。
  - **WS** (Window Scale，窗口扩大因子)：由于16bit限制，窗口最大值受限，因此加入窗口因子。它是一个可选的TCP头部选项，用于增加窗口大小，从而允许发送方发送更多的数据。实际窗口大小 = 窗口因子 * 窗口尺寸。
  - **MSS** (Maximum Segment Size，最大段大小)：表示TCP在一个段中可以发送的最大数据量。在Ethernet网络中，MSS常见是1460字节。因为1500字节的MTU (Maximum Transmission Unit) 中，去掉IP头部20字节和TCP头部20字节后，就剩下了1460字节，这就是为什么MSS的大小通常是1460字节。
  - **Len** (Length，长度)：TCP数据部分的长度，不包括TCP头部。



如图可以看到三次握手过程并没有数据包传输，Len=0，而之后一旦连接建立就需要发送数据了，Len=1424，这是因为该数据包包含了真正的数据内容。

这里还有一个提示：显示"**TCP segment of a reassembled PDU**"时，它意味着这个TCP段是一个分片的数据包的一部分，并且已经被重新组合。这种情况经常发生在传输大数据包时，尤其是当数据包的大小超过了网络的MTU时。

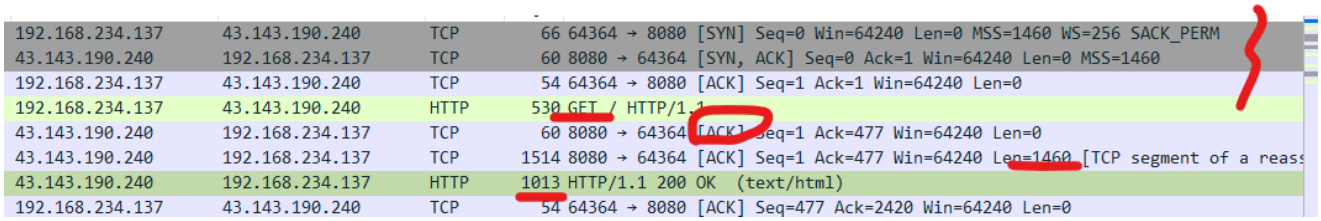
- **SACK_PERM** (Selective ACK Permitted，选择性确认允许)：这是一个TCP选项，表示接收方支持选择性确认（SACK）。选择性确认允许接收方只确认收到的非连续块的数据。

  在标准TCP确认中，接收方只发送一个确认号，告诉发送方它已经接收到的所有数据中的最后一个字节。但这种方法在出现丢包时并不是最高效的。例如，假设我发送了10个包，只有第4个包丢失了，那么接收方将持续地确认前3个包，即使它已经接收到了后面的包。这会导致发送方重新发送第4个包及其后面的所有包，即使大多数都已成功传输。

  使用SACK，接收方可以通知发送方它已接收到的不连续的数据段。继续上面的例子，接收方可以使用SACK来告诉发送方，尽管第4个包丢失了，但它已经接收到了后面的包。这样，发送方只需要重新发送第4个包。

3.4分析数据包：文本文件是如何传输的

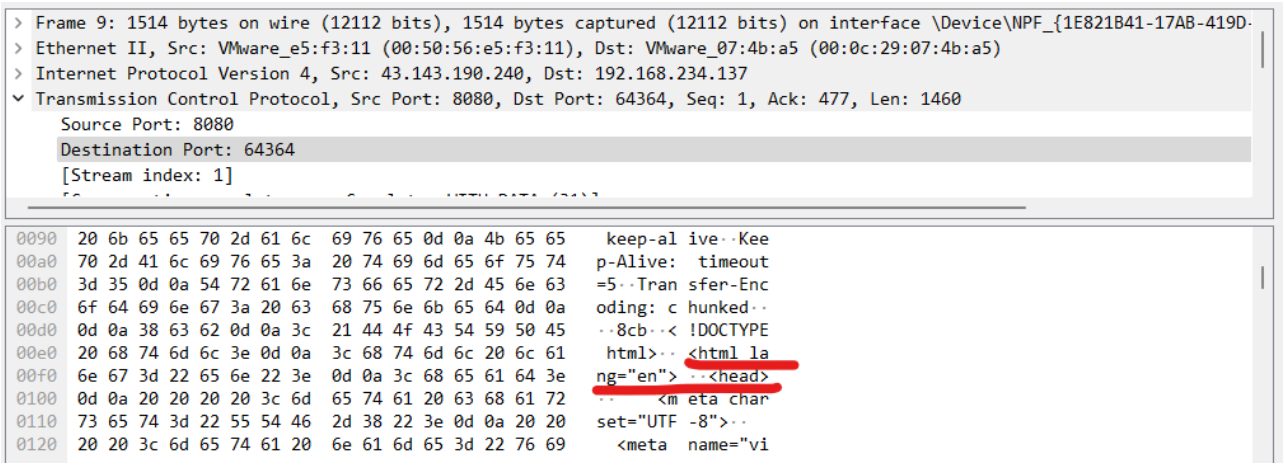
index.html是如何传输的



在三次握手之后，连续传了5个数据包

1. **192.168.234.137** (我)向 **43.143.190.240** （Web服务器）发送HTTP `GET` 请求:
   - 协议: HTTP
   - 长度: 530
   - 信息: `GET / HTTP/1.1`
     表示**请求**服务器上的某个资源（这里是**网站的首页**）。
2. **43.143.190.240** 向 **192.168.234.137** 发送ACK确认:
   - 协议: TCP
   - 长度: 60
   - 信息: `[ACK] Seq=1 Ack=477 Win=64240 Len=0`
     这是服务器**对我的HTTP GET请求的第一个确认**，它还没有发送实际的数据。`Len=0` 表示这个包中没有携带数据。
3. **43.143.190.240** 向 **192.168.234.137** 发送数据:
   - 协议: TCP
   - 长度: 1514
   - 信息: `[ACK] Seq=1 Ack=477 Win=64240 Len=1460 [TCP segment of a reassembled PDU]`
     这是服务器开始发送实际的HTTP数据给我的机器。这里，服务器**发送了1460字节的数据**，这个长度与MSS（最大段大小）相符。"[TCP segment of a reassembled PDU]" 的意思是这是一个被分段的TCP数据，需要更多的分段才能完全重组原始的PDU。

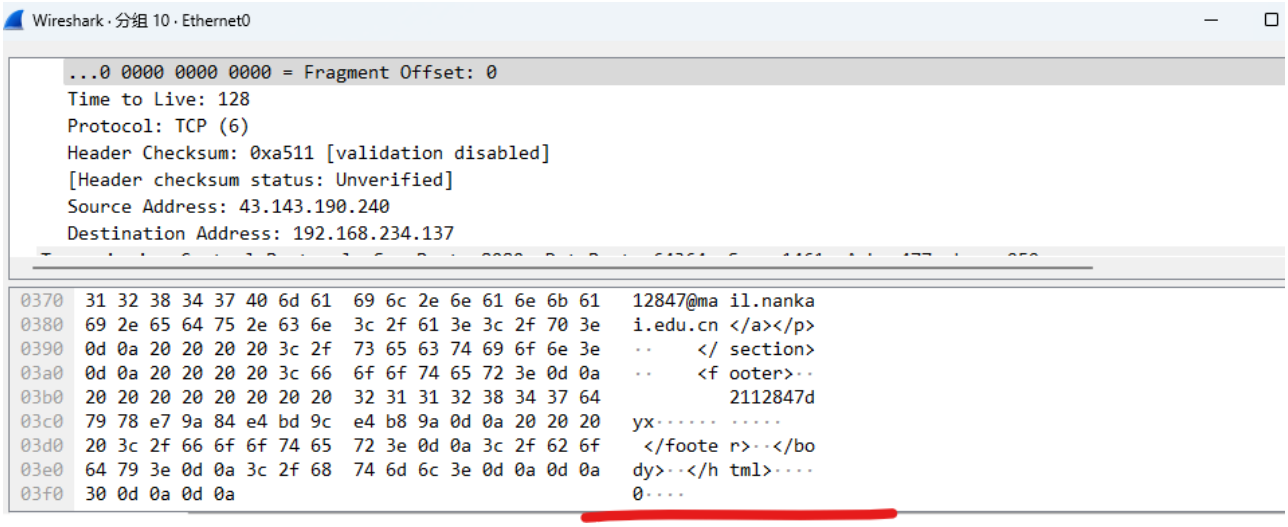
可以看到数据包当中包含index.html的前半段代码:



4. **43.143.190.240** 向 **192.168.234.137** 发送HTTP响应:

- 协议: HTTP
- 长度: 1013
- 信息: `HTTP/1.1 200 OK (text/html)`
  这是服务器的HTTP响应，表示**成功找到并返回了所请求的资源。**状态码为`200 OK`，表示请求成功，并且响应的内容类型是`text/html`，即一个HTML页面。

可以看到数据包当中包含index.html的后半段代码:



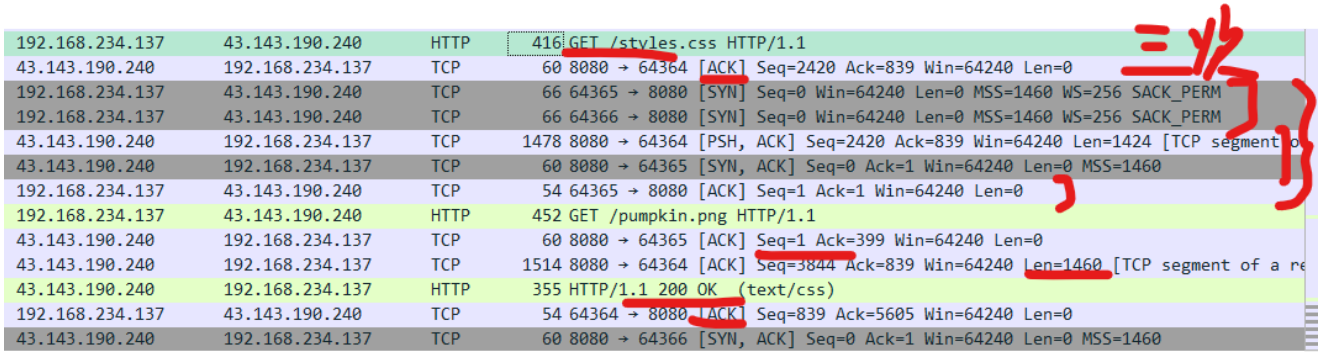
5. **192.168.234.137** 向 **43.143.190.240** 发送TCP回复:

- 协议:TCP
- 长度: 54
- 信息: `[ACK] Seq=477 Ack=2420 Win=64240 Len=0`
  这是我的响应，相当于告诉服务器我已经收到了index.html

综上，**index.html发给我的时候，数据是分两次传输的。**第一次传输了1460字节，第二次传输了一个完整的HTTP响应，其中包含了1013字节的数据。**状态码是200**，表示请求成功。

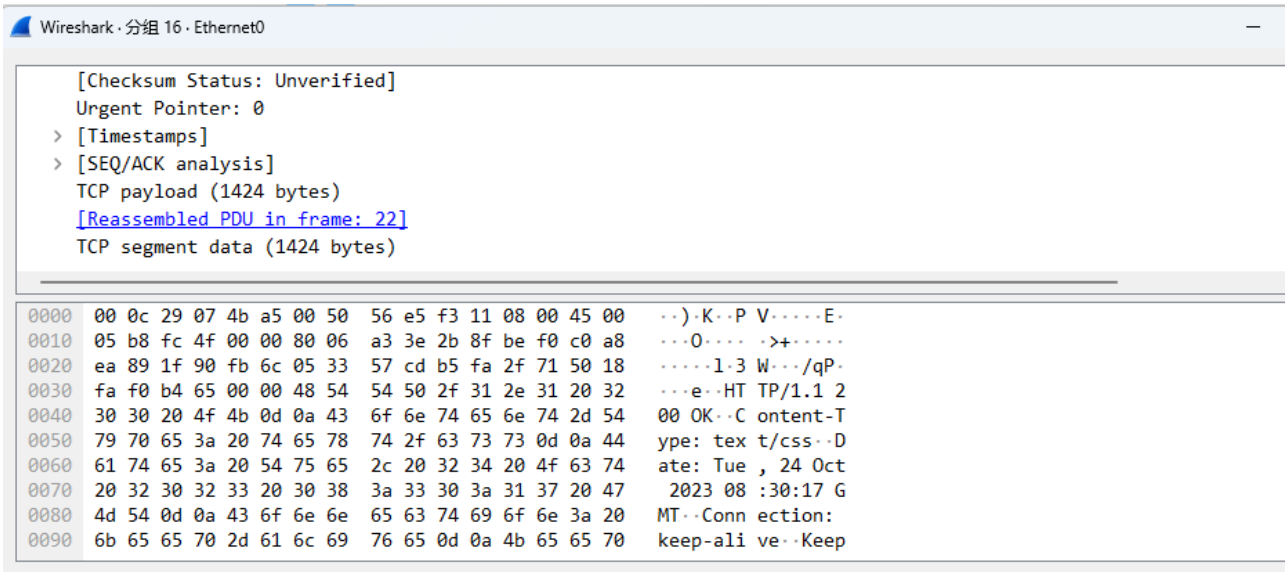
css是如何传输的

首先是我完全接受了index.html之后，才知道需要接收css,之后:

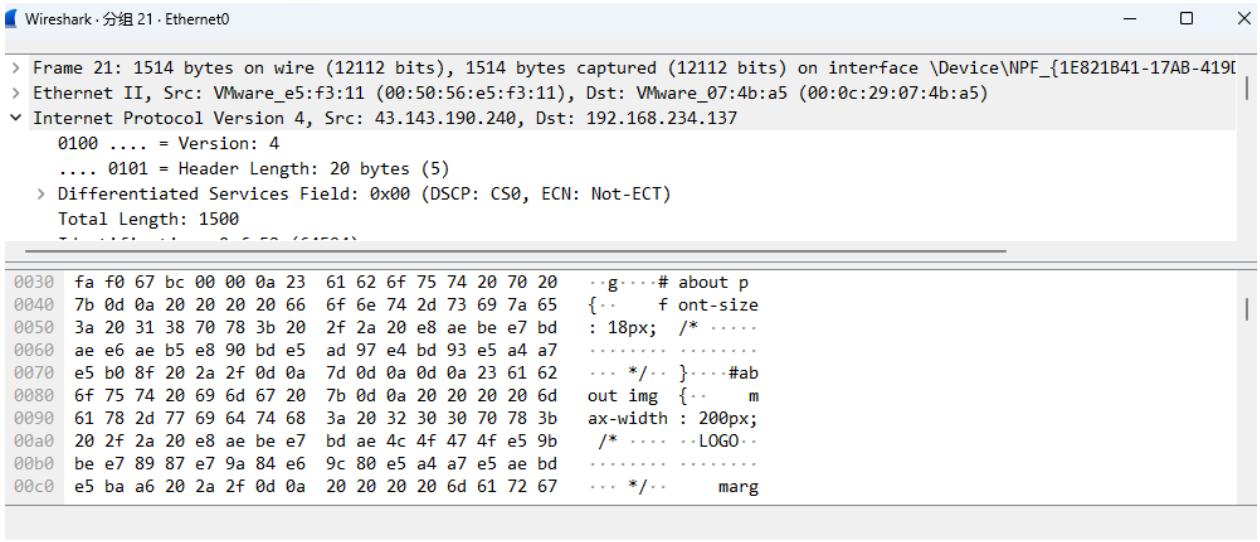


1. **请求CSS文件**:
   - 我的设备 `192.168.234.137` 发起一个GET请求 `GET /styles.css HTTP/1.1`，请求 `43.143.190.240` 服务器上的 `/styles.css` 文件。
2. **服务器的确认回应**:
   - 服务器 `43.143.190.240` 发送了一个ACK报文给源设备，确认它收到了GET请求。这里没有携带应用层数据，因此 `Len=0`。
3. **新的TCP连接请求【之前已经建立过，现在是第二次建立TCP连接】**:
   - `192.168.234.137` 向 `43.143.190.240` 发送一个SYN报文，请求建立一个新的TCP连接。这里的 `SYN` 标志表示TCP三次握手中的第一次握手，也即同步序列号用于初始化连接两端的序列号。
   - 这里还有一个有意思的地方，我的设备希望建立新的TCP连接的时候，SYN请求发了两次，这一个点接下来会分析。
   - [在这个中间，服务器开始向源设备发送 `/styles.css` 文件的数据。这里的 `PSH` 标志表示"push"，意味着这个TCP段包含了应用层数据，并且应该立即发送给应用，不用等待其他报文。数据长度为 `1424` 字节。]

   此时传输的是css文件的上半段:



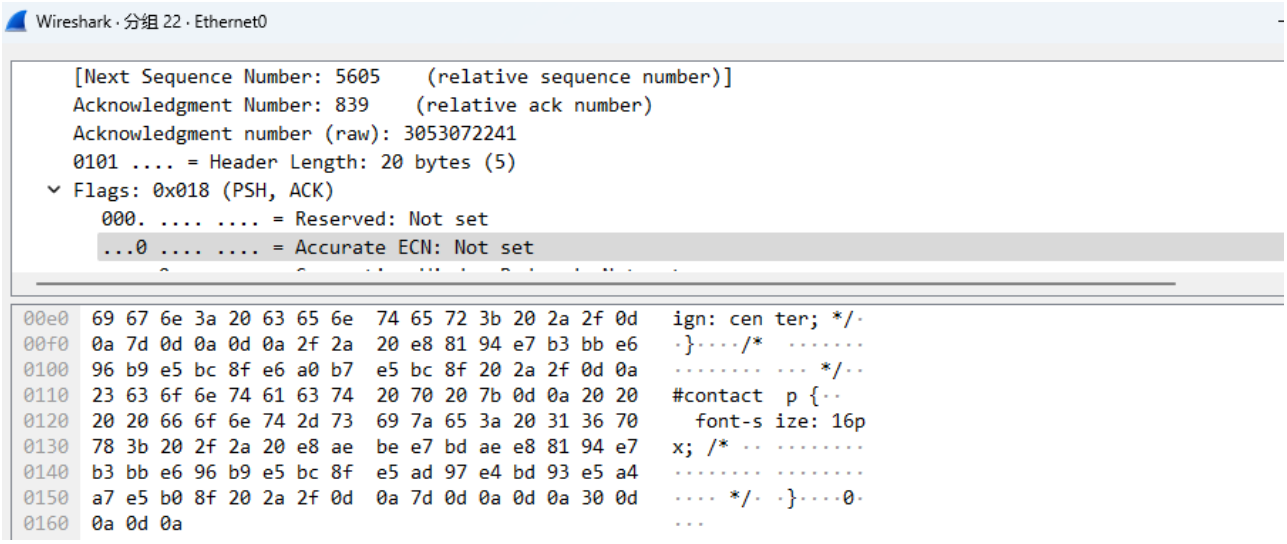
   - 接下来服务器回应一个 `SYN`，`ACK` 报文，表示同意建立连接，并且确认了源设备的SYN。这是TCP三次握手中的第二步。
   - 我的设备再次回应一个ACK报文，这样新的TCP连接就建立了。这三次握手过程中，数据长度都是0，因为这些报文仅用于连接管理，不携带应用层数据。
4. **数据传输开始**:
   - 源设备向服务器请求了一个其他资源 `GET /pumpkin.png HTTP/1.1`，服务器回复了一个ACK，但我的关注点是CSS文件，所以跳过这部分。
   - 接着服务器再次发送数据，数据长度为 `1460` 字节，这是一个完整的数据段，表示这是文件的其他部分。



5. **数据传输结束与确认**:

   - 服务器发送了 `HTTP/1.1 200 OK (text/css)` 响应给源设备，表示 `/styles.css` 文件成功地发送给了源设备，并且确认这是一个CSS文件。

     这一次服务器把css的最后一段传了过来。接着我的设备表示收到。



分析数据包:HTTP1.1的特性

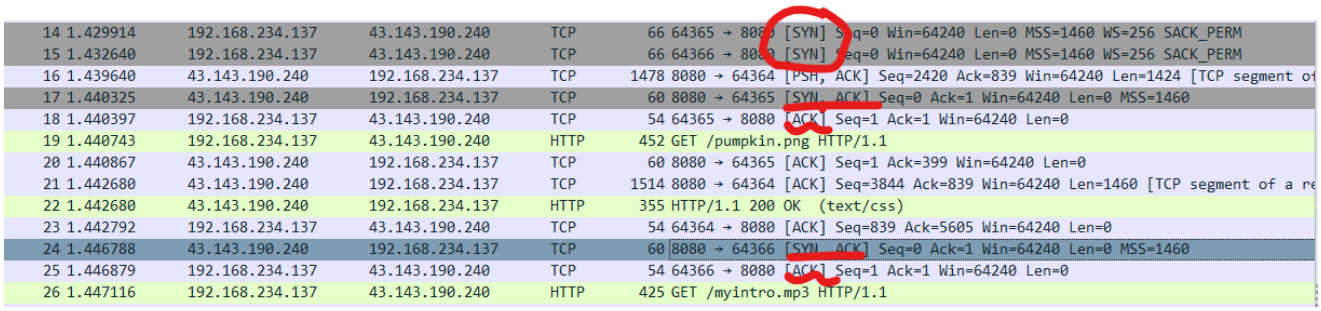
如之前的分析，在传输css、音频、图片文件的时候，可以请求一下，传输一下，还没传完，继续请求

这是由于HTTP/1.1支持请求的**管线化**，这意味着客户端可以在收到先前请求的响应之前发送多个请求。但这并不意味着响应可以交叉到达；响应仍然是按请求的顺序到达的。

如之前的分析，总共进行了两次TCP连接，在请求获取html的时候有一次，获取完html，请求获取css、图片、音频的时候有一次。

这是由于HTTP/1.1支持**持久连接（Keep-Alive）**：HTTP/1.1引入了持久连接的概念，这意味着一个TCP连接可以被用来发送和接收多个HTTP请求和响应。

分析数据包:超时重传

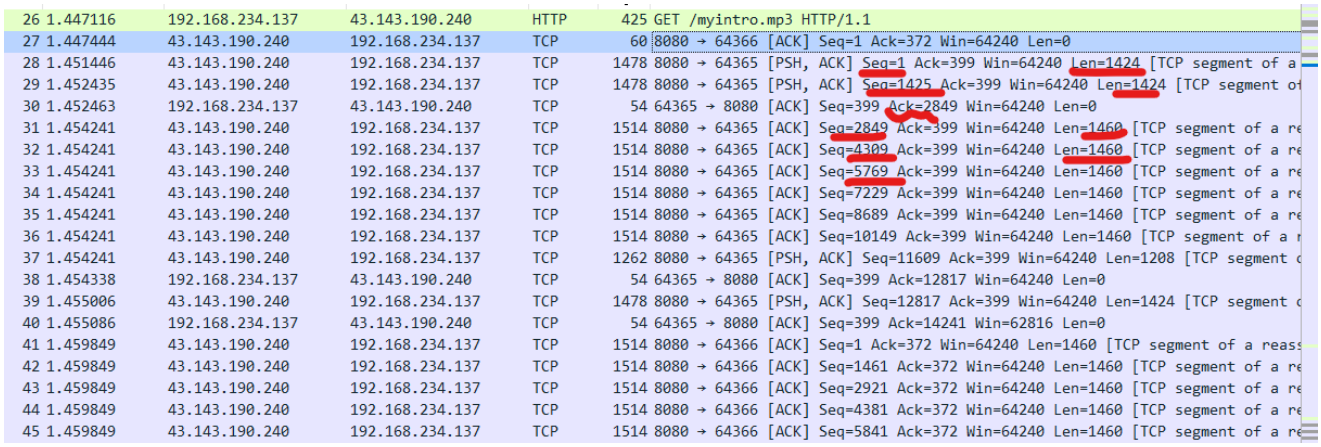


14	1.429914	192.168.234.137	43.143.190.240	TCP	66 64365 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
15	1.432640	192.168.234.137	43.143.190.240	TCP	66 64366 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
16	1.439640	43.143.190.240	192.168.234.137	TCP	1478 8080 → 64364 [PSH, ACK] Seq=2420 Ack=839 Win=64240 Len=1424 [TCP segment of
17	1.440325	43.143.190.240	192.168.234.137	TCP	60 8080 → 64365 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
18	1.440397	192.168.234.137	43.143.190.240	TCP	54 64365 → 8080 [ACK] Seq=1 Ack=1 Win=64240 Len=0
19	1.440743	192.168.234.137	43.143.190.240	HTTP	452 GET /pumpkin.png HTTP/1.1
20	1.440867	43.143.190.240	192.168.234.137	TCP	60 8080 → 64365 [ACK] Seq=1 Ack=399 Win=64240 Len=0
21	1.442680	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64364 [ACK] Seq=3844 Ack=839 Win=64240 Len=1460 [TCP segment of a re
22	1.442680	43.143.190.240	192.168.234.137	HTTP	355 HTTP/1.1 200 OK (text/css)
23	1.442792	192.168.234.137	43.143.190.240	TCP	54 64364 → 8080 [ACK] Seq=839 Ack=5605 Win=64240 Len=0
24	1.446788	43.143.190.240	192.168.234.137	TCP	60 8080 → 64366 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
25	1.446879	192.168.234.137	43.143.190.240	TCP	54 64366 → 8080 [ACK] Seq=1 Ack=1 Win=64240 Len=0
26	1.447116	192.168.234.137	43.143.190.240	HTTP	425 GET /myintro.mp3 HTTP/1.1

在第二次建立TCP连接的时候，我可以清晰地看到TCP三次握手的过程，即SYN、SYN-ACK和ACK。并且可以发现，连续发了两个SYN，间隔的收到了两个SYN-ACK。

1. **SYN**：在图中，数据包 `No. 14` 和`No. 15`都包含SYN标志，这意味着第一次的SYN可能没有被服务器正确接收，导致客户端重新发送了一个SYN数据包。
2. **SYN-ACK**：在接收到客户端的SYN数据包后，服务器会回复一个SYN-ACK（同步应答）数据包。在图中，数据包 `No. 17` 和 `No. 24` 都是SYN-ACK数据包。由于之前有两个SYN数据包，所以收到了两个SYN-ACK。
3. **ACK**：客户端在收到服务器的SYN-ACK数据包后，会发送一个ACK（应答）数据包来确认连接的建立。在图中，数据包 `No. 19` 和 `No. 25` 都是ACK数据包。由于之前有两次SYN ACK，所以各自回复了一个ACK。

分析数据包：多媒体文件传输是如何传输的



26	1.447116	192.168.234.137	43.143.190.240	HTTP	425 GET /myintro.mp3 HTTP/1.1
27	1.447444	43.143.190.240	192.168.234.137	TCP	60 8080 → 64366 [ACK] Seq=1 Ack=372 Win=64240 Len=0
28	1.451446	43.143.190.240	192.168.234.137	TCP	1478 8080 → 64365 [PSH, ACK] Seq=1 Ack=399 Win=64240 Len=1424 [TCP segment of a
29	1.452435	43.143.190.240	192.168.234.137	TCP	1478 8080 → 64365 [PSH, ACK] Seq=1425 Ack=399 Win=64240 Len=1424 [TCP segment of
30	1.452463	192.168.234.137	43.143.190.240	TCP	54 64365 → 8080 [ACK] Seq=399 Ack=2849 Win=64240 Len=0
31	1.454241	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64365 [ACK] Seq=2849 Ack=399 Win=64240 Len=1460 [TCP segment of a re
32	1.454241	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64365 [ACK] Seq=4309 Ack=399 Win=64240 Len=1460 [TCP segment of a re
33	1.454241	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64365 [ACK] Seq=5769 Ack=399 Win=64240 Len=1460 [TCP segment of a re
34	1.454241	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64365 [ACK] Seq=7229 Ack=399 Win=64240 Len=1460 [TCP segment of a re
35	1.454241	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64365 [ACK] Seq=8689 Ack=399 Win=64240 Len=1460 [TCP segment of a re
36	1.454241	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64365 [ACK] Seq=10149 Ack=399 Win=64240 Len=1460 [TCP segment of a r
37	1.454241	43.143.190.240	192.168.234.137	TCP	1262 8080 → 64365 [PSH, ACK] Seq=11609 Ack=399 Win=64240 Len=1208 [TCP segment c
38	1.454338	192.168.234.137	43.143.190.240	TCP	54 64365 → 8080 [ACK] Seq=399 Ack=12817 Win=64240 Len=0
39	1.455006	43.143.190.240	192.168.234.137	TCP	1478 8080 → 64365 [PSH, ACK] Seq=12817 Ack=399 Win=64240 Len=1424 [TCP segment c
40	1.455086	192.168.234.137	43.143.190.240	TCP	54 64365 → 8080 [ACK] Seq=399 Ack=14241 Win=62816 Len=0
41	1.459849	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64366 [ACK] Seq=1 Ack=372 Win=64240 Len=1460 [TCP segment of a reass
42	1.459849	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64366 [ACK] Seq=1461 Ack=372 Win=64240 Len=1460 [TCP segment of a re
43	1.459849	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64366 [ACK] Seq=2921 Ack=372 Win=64240 Len=1460 [TCP segment of a re
44	1.459849	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64366 [ACK] Seq=4381 Ack=372 Win=64240 Len=1460 [TCP segment of a re
45	1.459849	43.143.190.240	192.168.234.137	TCP	1514 8080 → 64366 [ACK] Seq=5841 Ack=372 Win=64240 Len=1460 [TCP segment of a re

当传输大文件时，TCP会将其分割为多个小数据段进行发送，而不是一次性发送整个文件。这确保了网络的有效利用和数据的完整性。

1. **服务端的Seq和Len关联**：

- 当服务端发送数据时，它会使用一个序列号(Seq)，这是数据段的首字节的编号。
- `Len`指示了数据段的长度。
- 下一个数据段的`Seq`将是当前数据段的`Seq`加上当前数据段的`Len`。
  例如，数据包`No. 29`的`Seq`是`1425`，`Len`是`1424`。那么下一个数据段的`Seq`是`1425 + 1424 = 2849`，与数据包`No. 30`中的`Seq`一致。

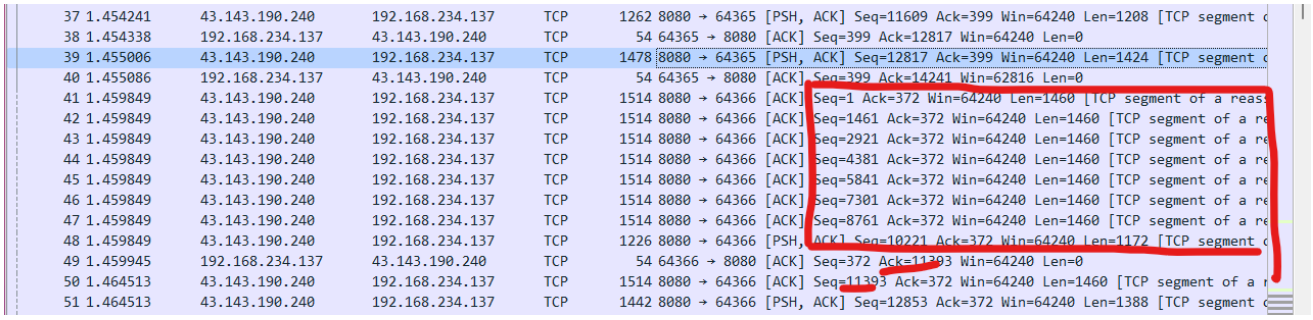
2. **客户端的ACK**：

- 当客户端成功接收到一个数据段，它会发送一个ACK来确认接收。ACK的值等于它期望接收的下一个数据段的Seq。
- 数据包`No. 30`中的ACK值为2849，这意味着客户端已经接收到了Seq为0~2849的数据段，并期望接收Seq为2849的下一个数据段。这与服务端在数据包`No. 31`中发送的数据段的Seq一致。

3. **分段传输**：

- 数据包No．28、No. 29、No. 37、No. 39等都标有PSH，这表示推送标志。当应用程序要求TCP立即发送数据时，就会使用这个标志。
- 每个这样的数据包都有其长度（如1460、1424等），这意味着大文件被分成了多个小段进行传输。

4.**连续发送，偶尔回复**：可以看到服务器连续多次发送了长1460的多媒体数据包，中间不时地收到客户端的响应，客户端指明它收到了哪里，以及希望继续从哪里开始发送。例如下面这一段，服务端连续发了1~11393这一段，客户端回复已经收到了，请继续从11393发送，然后服务端继续从11393发送。



分析数据包：四次挥手

Uploaded image

1. **第一次挥手**:

- **FIN, PSH, ACK**：数据包No．172中，地址43.143.190.240发送了一个FIN(finish)标志【因为服务器的数据已经发完了】，表示它已完成数据发送并希望关闭连接。同时，它还发送了PSH和ACK标志。这个数据包的Seq是5605，Ack是839。

2. **第二次挥手**:

- **ACK**：在数据包No．173中，目标地址192.168.234.137发送了一个ACK标志，表示它已收到对方的FIN请求，并确认了这个FIN请求。Seq为839，Ack为5606（即前一个数据包的Seq+1）。

3. **第三次挥手**:

- **FIN, ACK**：在数据包No．174中，源地址192.168.234.137也表示它已完成数据发送并希望关闭连接。所以，它发送了一个FIN标志，并带有ACK确认。Seq为839，Ack为5606。

4. **第四次挥手**:

- **ACK**：最后，在数据包No．175中，目标地址43.143.190.240发送了一个ACK标志，确认了对方的FIN请求。Seq为5606，Ack为840（即前一个数据包的Seq+1）。

通过这四次挥手的过程，TCP断开。

并且有意思的是，**之前建立了三个TCP连接，即SYN了三次，所以有三次TCP关闭，即四次挥手了三次。**

因为每个TCP连接都是独立的，因此每个连接的建立（三次握手）和关闭（四次挥手）都是独立进行的。如果在同一个通讯过程中建立了三个TCP连接，那么当这些连接不再需要时，都需要进行单独的关闭流程。

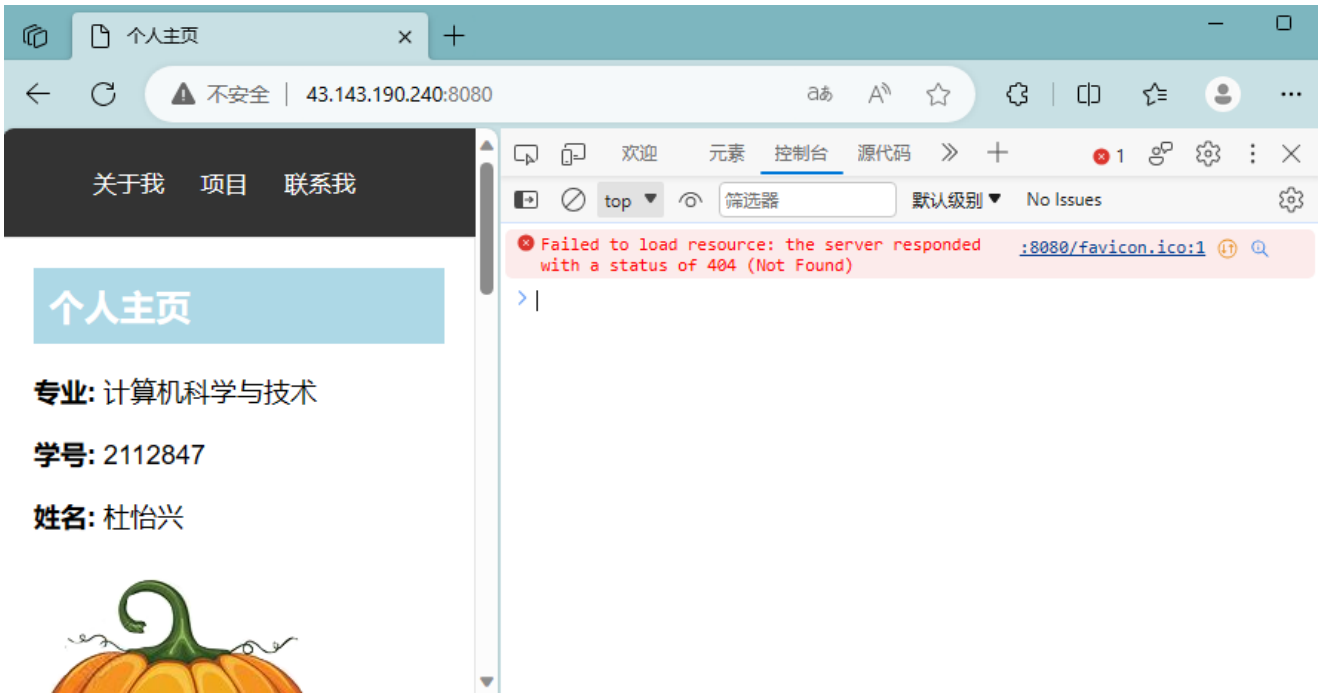
而之前说过，之前TCP连接请求超时重传了，一般来讲，只会建立两个TCP连接，最后只会有两个四次挥手。就像下面这样。

1223	13.902586	43.143.190.240	192.168.234.137	HTTP	213 HTTP/1.1 200 OK (audio/mpeg)
1224	13.902615	192.168.234.137	43.143.190.240	TCP	54 50832 → 8080 [ACK] Seq=1184 Ack=74152 Win=64240 Len=0
1225	13.940258	192.168.234.137	43.143.190.240	TCP	54 50833 → 8080 [ACK] Seq=399 Ack=60266 Win=63783 Len=0
1230	14.124362	192.168.234.137	43.143.190.240	HTTP	452 GET /favicon.ico HTTP/1.1
1231	14.124781	43.143.190.240	192.168.234.137	TCP	60 8080 → 50832 [ACK] Seq=74152 Ack=1582 Win=64240 Len=0
1235	14.187654	43.143.190.240	192.168.234.137	HTTP	241 HTTP/1.1 404 Not Found (text/plain)
1236	14.238908	192.168.234.137	43.143.190.240	TCP	54 50832 → 8080 [ACK] Seq=1582 Ack=74339 Win=64053 Len=0
1310	18.908014	43.143.190.240	192.168.234.137	TCP	60 8080 → 50833 [FIN, PSH, ACK] Seq=60266 Ack=399 Win=64240 Len=0
1311	18.908112	192.168.234.137	43.143.190.240	TCP	54 50833 → 8080 [ACK] Seq=399 Ack=60267 Win=63783 Len=0
1312	18.908281	192.168.234.137	43.143.190.240	TCP	54 50833 → 8080 [FIN, ACK] Seq=399 Ack=60267 Win=63783 Len=0
1313	18.908492	43.143.190.240	192.168.234.137	TCP	60 8080 → 50833 [ACK] Seq=60267 Ack=400 Win=64239 Len=0
1323	19.215001	43.143.190.240	192.168.234.137	TCP	60 8080 → 50832 [FIN, PSH, ACK] Seq=74339 Ack=1582 Win=64240 Len=0
1325	19.215220	192.168.234.137	43.143.190.240	TCP	54 50832 → 8080 [ACK] Seq=1582 Ack=74340 Win=64053 Len=0
1326	19.215960	192.168.234.137	43.143.190.240	TCP	54 50832 → 8080 [FIN, ACK] Seq=1582 Ack=74340 Win=64053 Len=0
1327	19.216467	43.143.190.240	192.168.234.137	TCP	60 8080 → 50832 [ACK] Seq=74340 Ack=1583 Win=64239 Len=0

小插曲：favicon

1230	14.124362	192.168.234.137	43.143.190.240	HTTP	452 GET /favicon.ico HTTP/1.1
1231	14.124781	43.143.190.240	192.168.234.137	TCP	60 8080 → 50832 [ACK] Seq=74152 Ack=1582 Win=64240 Len=0
1235	14.187654	43.143.190.240	192.168.234.137	HTTP	241 HTTP/1.1 404 Not Found (text/plain)
1236	14.238908	192.168.234.137	43.143.190.240	TCP	54 50832 → 8080 [ACK] Seq=1582 Ack=74339 Win=64053 Len=0

经常在数据包里面发现这样一段话，get 一个favicon.icon文件,ACK,然后显示404 not found,回复ACK
然后f12说Failed to load resource: the server responded with a status of 404 (Not Found)



这是由于，浏览器会自动发送一个请求到网站，试图获取 `/favicon.ico` 文件，以展示对应的网站图标。

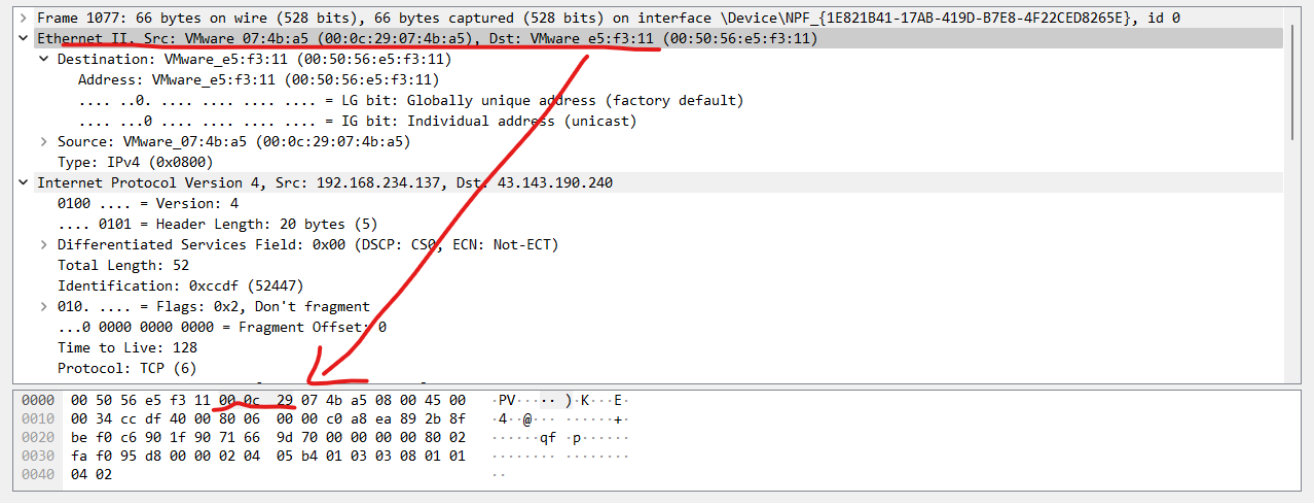
如果服务器上没有这个文件，服务器就会返回一个 `404 Not Found` 错误。

为了解决这个问题，可以**在 HTML 中指定图标**：可以在HTML的`<head>`部分使用`<link>`标签来指定一个不同名称的图标文件，例如：

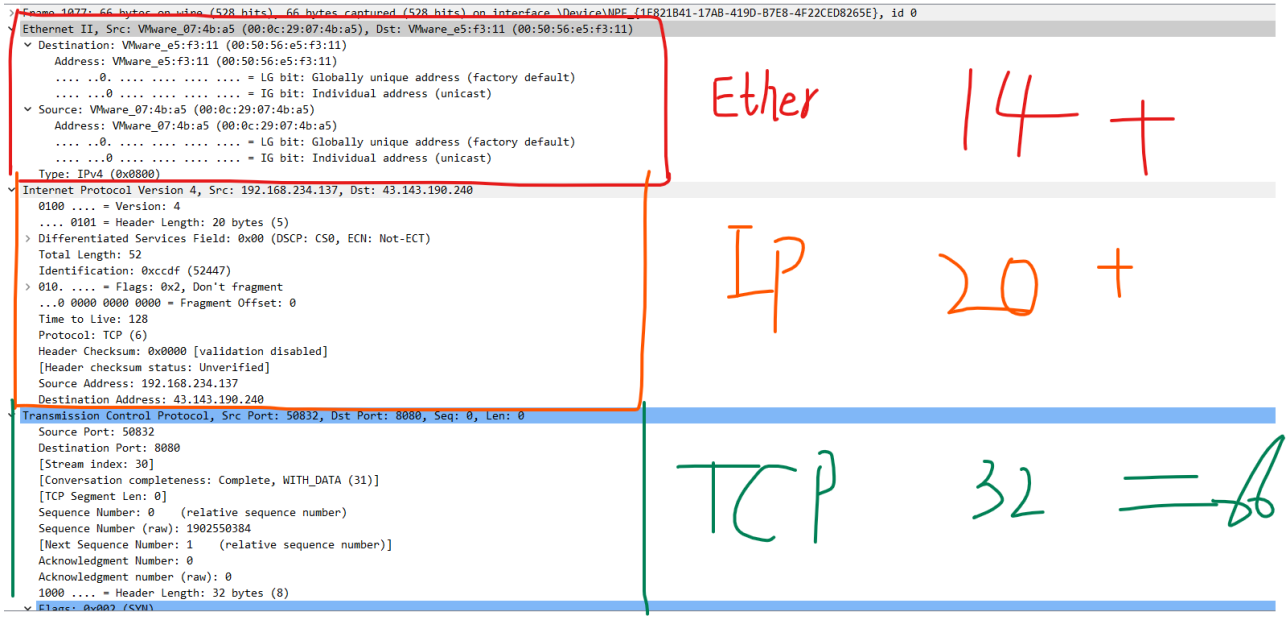
```
<link rel="icon" href="/path/to/icon.png" type="image/png">
```

分析数据包：数据报文详解

数据包当中除了数据(长度为Len)之外还有Ethernet头，IP头和TCP头的长度，并且每一个二进制都对应一段信息。



1. **Frame 1077**:
   - `66 bytes on wire (528 bits)`: 这表示这个数据帧包含 66 字节，或者 528 位,这和一开始的66对应。
   - `66 bytes captured (528 bits) on interface Device\NPF_{1E812B41-17AB-419D-B7E8-...}`: 这表示在特定的网络接口上捕获了66字节的数据。
2. **Ethernet II**:
   - `Src: VMware_07:4b:a5 (00:0c:29:07:4b:a5)`: 发送此帧的设备【我的电脑】的物理地址（MAC地址）。
   - `Dst: VMware_e5:f3:11 (00:50:56:e5:f3:11)`: 目的设备【服务器】的物理地址（MAC地址）。
   - `Type: IPv4 (0x0800)`: 这指示该帧的内容是IPv4数据。



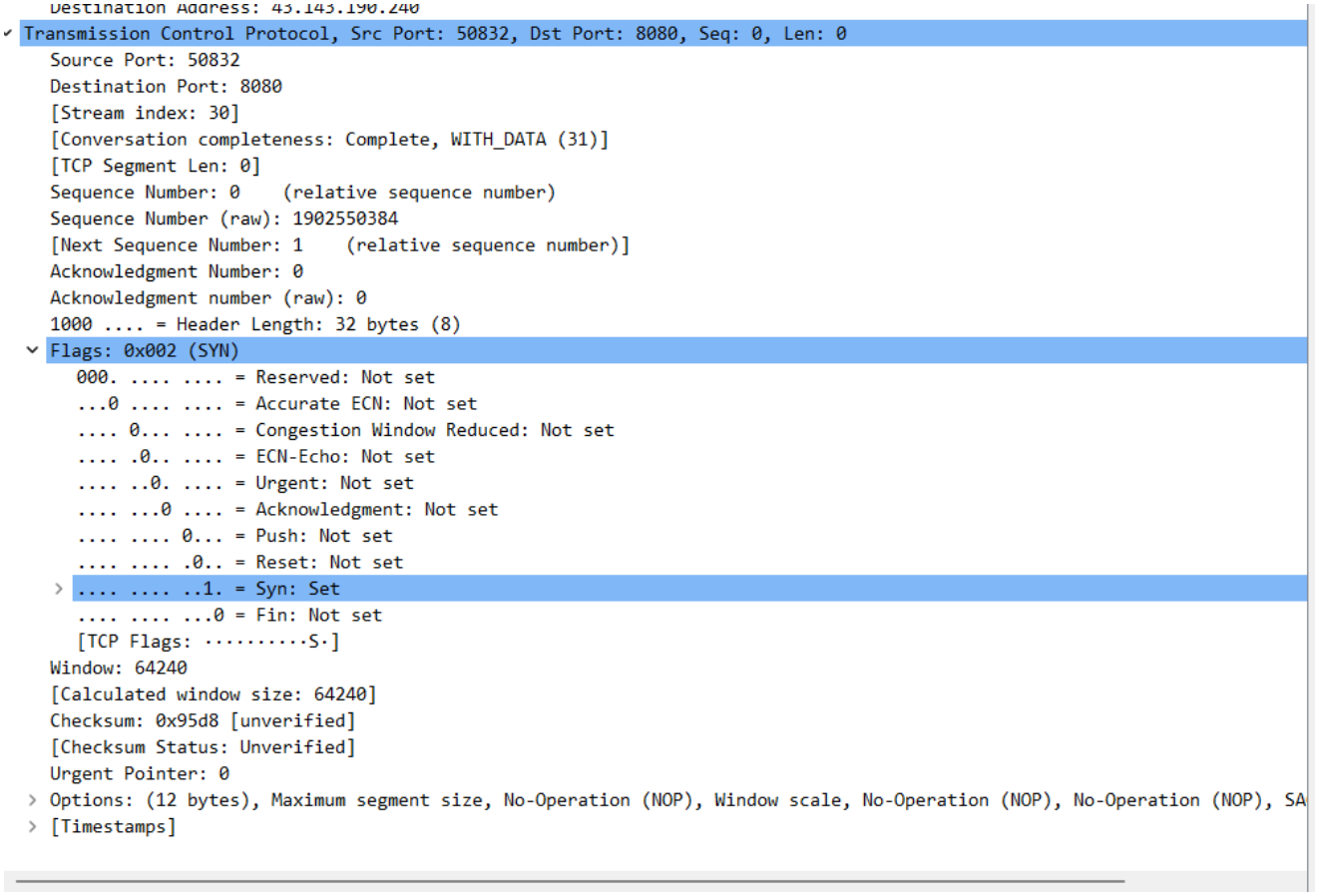
IP头详解:

1. **Internet Protocol Version 4**:
   - `Version: 4`: 这表示使用的是IP协议的第4版本，即IPv4。
   - `Header Length: 20 bytes (5)`: 这表示IP头的长度是20字节【通常就是20】。
   - `Total Length: 52`: 这表示整个IP数据包的长度为52字节。
   - `Identification: 0xccdf (52447)`: 数据包的标识号。
   - `Time to Live (TTL): 128`: 这表示数据包在网络中的生存时间，超过这个时间数据包会被丢弃。
   - `Protocol: TCP (6)`: 这指示数据包的内容是TCP数据。
   - `Header Checksum: 0x0000`: 用来检查头部数据的正确性的校验和。

- `Source Address: 192.168.234.137`: 发送数据包的设备的IP地址【我】。
- `Destination Address: 43.143.190.240`: 目的设备的IP地址【服务器】。

2. **Transmission Control Protocol, Src Port: 50832, Dst Port: 8080, Seq: 0, Len: 0**:
   - TCP层的信息。来源端口是50832，目的端口是8080。序列号为0，数据长度为0。

TCP头详解



- **Source Port**: 50832 - 发起连接请求的客户端所使用的源端口（我）。
- **Destination Port**: 8080 - 目标服务器监听的端口（服务器），通常用于HTTP代理和HTTP备用端口。
- **TCP Segment Len**: 0 - 这表示TCP数据段的长度为0，即此数据包不包含任何应用层数据。
- **Sequence Number (raw)**: 19255038 - TCP数据段的序列号。当这个连接开始时，这个序列号会被随机选择，随后的数据包会增加相应的值，基于发送的数据量。
- **Next Sequence Number**: 1 - 下一个期望的序列号，意味着此数据包后的下一个数据包的序列号应该是这个值。
- **Acknowledgment Number**: 0 - 因为一个SYN数据包，所以没有任何数据被确认。
- **Header Length**: 32 bytes (8) - 这表示TCP头部的长度是32字节。包括了选项字段的长度。

这里很有意思，为什么之前IP数据包是52B，因为TCP头是32B+IP头20B+14Ethernet头通常为14字节=66B

- **Flags**: TCP头中的标志位。在这里，我看到SYN标志被设置，表示一个连接请求。
- **Window**: 64240 - 接收窗口大小，表示发送端愿意接收的字节数量，不需要确认。这可以用于流量控制。
- **Checksum**: 0x95d8 - 为了检查数据包在传输过程中是否发生错误的检查和值。
- **Urgent Pointer**: 0 - 这指示了数据包中的紧急数据。当它为0时，表示没有紧急数据。
- **Options**:
  - **Maximum segment size**: TCP连接的两端可以接收的最大段大小。
  - **Window scale**: 这用于定义滑动窗口的大小。它可以让TCP使用更大的窗口，以便更有效地传输数据。
  - **No-Operation (NOP)**: 这实际上是一个填充选项，常用于对齐其他选项。

实验总结

计网实验二的总结：

在这次实验中，我深入探讨了使用Wireshark抓包工具在云服务器上的个人网站上抓取和分析网络数据包。首先，我完成了实验的准备工作，包括明确实验要求、租用Web服务器和安装Wireshark工具。随后，我按照实验流程，先制作了网页，然后搭建服务器并开始抓包。

在得到的实验结果中，我成功导出了数据包并进行了深入分析。我详细探讨了三次握手的数据包过程，理解了每个数据包中的信息如序号、时间、长度和其他信息的意义。我进一步研究了文本文件如index.html和css是如何在网络上传输的，并探索了HTTP1.1的特性。此外，我还研究了超时重传、多媒体文件的传输方法以及TCP连接终止时的四次挥手。

在实验过程中，我还遇到了一个小插曲，即favicon的分析。最后，我对数据报文进行了详尽的解析，包括IP头和TCP头的每个字段。

总的来说，这次实验不仅增强了我对网络传输原理的理解，还让我熟悉了Wireshark这一强大的网络分析工具，为我今后的学习和研究打下了坚实的基础。