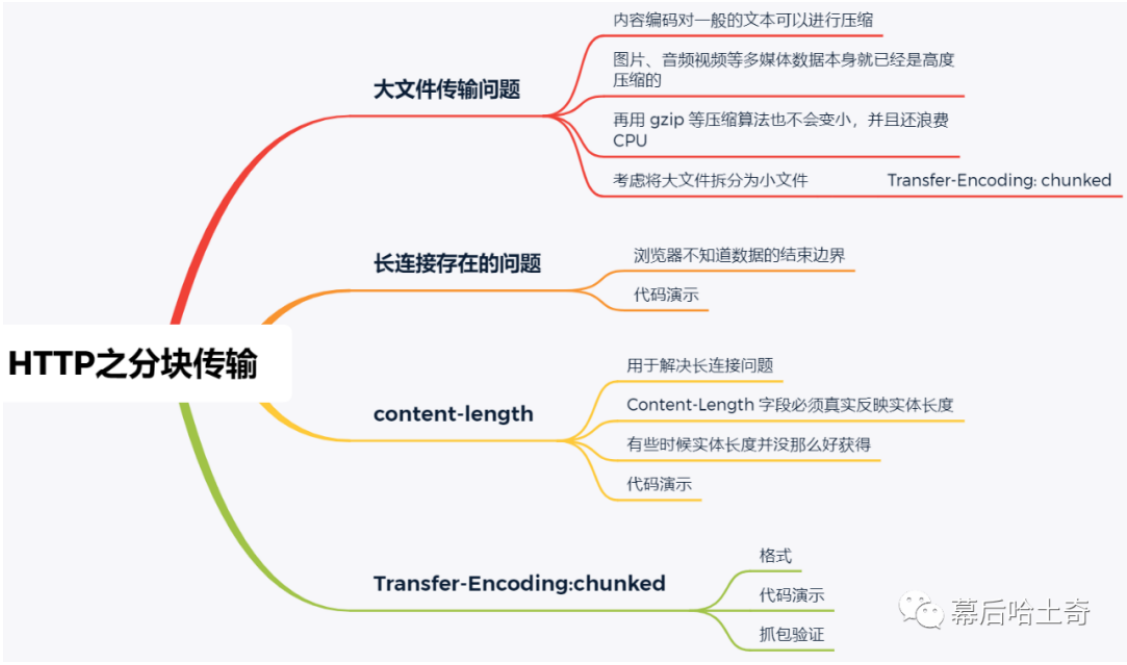
本篇文章探索的是Transfer-Encoding字段，我们来看看这个字段为什么会产生、有何妙用？跟着我走。本文大纲为：



一、大文件传输问题

我们已经学习过Encoding type首部字段，即内容编码，一般浏览器发送请求时都会带上"Accept-Encoding"字段，里面是浏览器支持的压缩格式列表，例如gzip、deflate、br 等，这样服务器就可以从中选择一种压缩算法，放进“Content-Encoding”响应头里，再把原数据压缩后发给浏览器。

对于比如HTML文件，进行压缩是很有必要的，在相同带宽的条件下，本来100K的文件被压缩为20K传输，将大大提高传输效率。

但是，内容编码通常是选择性的，图片、音频视频等多媒体数据本身就已经是高度压缩的，再用 gzip 等压缩算法也不会变小，并且还浪费CPU。

那在传输大文件比如视频文件时，该如何解决呢?

我们可以考虑将大文件拆分为小文件，这样浏览器和服务器都不用在内存里保存文件的全部，每次只收发一小部分，网络也不会被大文件长时间占用，内存、带宽等资源也就节省下来了。

这种拆分思想是从网络链路层一层传承到应用层的，我们尽可能不要一次性传输大的数据，抛开其他考虑，当出错的时候，补偿的代价也会小很多。

该如何进行拆分呢？在 HTTP 协议里就是“chunked”分块传输编码。

响应报文里用头字段“Transfer-Encoding: chunked”来表示，意思是报文里的 body 部分不是一次性发过来的，而是分成了许多的块（chunk）逐个发送。

听起来很理所当然，但是实现起来又会遇到问题。

二、长连接存在的问题

问题出在哪里呢？问题就出在优秀的长连接上。

我们简单复习下长连接，在HTTP/1.0中引入了长连接机制，通过 Connection: keep-alive 这个头部来实现，服务端和客户端都可以使用它告诉对方在发送完数据之后不需要断开 TCP 连接，以备后用。

HTTP/1.1 则规定所有连接都必须是持久的，除非显式地在头部加上 Connection: close。实际上，HTTP/1.1 中 Connection 这个头部字段已经没有 keep-alive 这个取值了，但由于历史原因，很多 Web Server 和浏览器，还是保留着给 HTTP/1.1 长连接发送 Connection: keep-alive 的习惯。

好了，复习完了长连接，下面来看下长连接的问题。

接着《61 | 应用层篇：实战之实现一个简易的web服务器》文章，修改我们自己实现的web服务器代码：

```
public static void main(String[] args) throws IOException {
    ServerSocket server = new ServerSocket(8888);
    System.out.println("服务器已经启动...正在监听8888端口，随时等待客户端连接");
    //服务端创建一个线程来处理客户端请求
    while (!Thread.interrupted()){
        //接收用户请求
        Socket client = server.accept();
        //获取输入输出流
        InputStream ins = client.getInputStream();
        OutputStream out = client.getOutputStream();

        String c = "hello world!";
        BufferedReader br = new BufferedReader(new InputStreamReader(ins));
        System.out.println(br.readLine());

        //给用户响应
        PrintWriter pw = new PrintWriter(out);
        pw.println("HTTP/1.1 200 OK");
        pw.println("");
        pw.flush();
        pw.println(c);
        pw.flush();
        System.out.println("处理结束");
    }
```

```
}
```

我发现一直pending不返回hello world：

```
[C:\~]$ curl  http://localhost:8888
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
 Left  Speed
100    14    0    14    0     0      0      0 --:--:--  0:00:30 --
:--:--     0
```

是什么原因呢?

这是因为，对于短连接，浏览器可以通过连接是否关闭来界定请求或响应实体的边界；

而对于长连接，这种方法显然不奏效。上例中，尽管我已经发送完所有数据，但浏览器并不知道这一点，它无法得知这个打开的连接上是否还会有新数据进来，只能傻傻地等了。

三、content-length

要解决上面这个问题，最容易想到的办法就是计算实体长度，并通过头部告诉对方。这就要用到 Content-Length 了，改造一下上面的例子：

```
public static void main(String[] args) throws IOException {
    ServerSocket server = new ServerSocket(8888);
    System.out.println("服务器已经启动...正在监听8888端口，随时等待客户端连
接");
    //服务端创建一个线程来处理客户端请求
    while (!Thread.interrupted()){
        //接收用户请求
        Socket client = server.accept();
        //获取输入输出流
        InputStream ins = client.getInputStream();
        OutputStream out = client.getOutputStream();

        String c = "hello world!";
        BufferedReader br = new BufferedReader(new
InputStreamReader(ins));
        System.out.println(br.readLine());

        //给用户响应
        PrintWriter pw = new PrintWriter(out);
```

```
        pw.println("HTTP/1.1 200 OK");
        //****注意这一行****
        pw.println("Content-Length: " + (c.length()));
        pw.println("");
        pw.flush();
        pw.println(c);
        pw.flush();
        System.out.println("处理结束");
    }
}
```

注意到，我们加了一行代码：

```
//****注意这一行****
pw.println("Content-Length: " + (c.length()));
```

执行curl命令顺利拿到了结果：

```
[C:\~]$ curl  http://localhost:8888
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
 Left  Speed
100    12  100    12    0     0     12      0  0:00:01 --:--:--
 0:00:01   151
hello world!
```

继续调整，修改下这一行代码：

```
pw.println("Content-Length: " + (c.length()-5));
```

结果是：

```
[C:\~]$ curl  http://localhost:8888
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
 Left  Speed
100     7  100     7    0     0      7      0  0:00:01 --:--:--
 0:00:01   148
hello w
```

我再次修改：

```
pw.println("Content-Length: " + (c.length()+100));
```

发现又pending不返回hello world了：

```
[C:\~]$ curl  http://localhost:8888
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
 Left  Speed
 12   112   12    14    0     0      2      0  0:00:56  0:00:06
 0:00:50     0
```

可以看到，在长连接中，响应头中不仅需要Content-Length字段，并且还必须准确，如果Content-Length小于实际的文本长度，返回内容会被截断；若Content-Length大于实际的文本长度，会造成 pending。

由于 Content-Length 字段必须真实反映实体长度，但实际应用中，有些时候实体长度并没那么好获得，例如实体来自于网络文件，或者由动态语言生成。这时候要想准确获取长度，只能开一个足够大的 buffer，等内容全部生成好再计算。但这样做一方面需要更大的内存开销，另一方面也会让客户端等更久。

为此我们需要一个新的机制：不依赖头部的长度信息，也能知道实体的边界。

四、Transfer-Encoding:chunked

好了，为了解决如上问题，本文的主角Transfer-Encoding隆重登场。

当响应报文中返回了“Transfer-Encoding: chunked”，则代表这个报文采用了分块编码。

其格式为：报文中的实体需要改为用一系列分块来传输，每个分块包含十六进制的长度值和数据，长度值独占一行，长度不包括它结尾的 CRLF（\r\n），也不包括分块数据结尾的 CRLF。最后一个分块长度值必须为 0，对应的分块数据没有内容，表示实体结束。

下面按照这种格式改造代码：

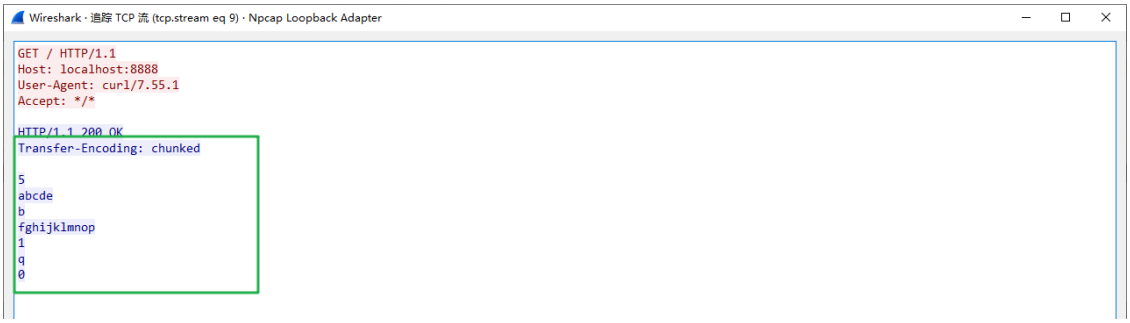
```
public static void main(String[] args) throws IOException {
    ServerSocket server = new ServerSocket(8888);
    System.out.println("服务器已经启动...正在监听8888端口，随时等待客户端连
接");
    //服务端创建一个线程来处理客户端请求
    while (!Thread.interrupted()){
        //接收用户请求
```

```
        Socket client = server.accept();
        //获取输入输出流
        InputStream ins = client.getInputStream();
        OutputStream out = client.getOutputStream();

        BufferedReader br = new BufferedReader(new
InputStreamReader(ins));
        System.out.println(br.readLine());

        //给用户响应
        PrintWriter pw = new PrintWriter(out);
        pw.println("HTTP/1.1 200 OK");
        pw.println("Transfer-Encoding: chunked");
        pw.println("");
        pw.flush();

        pw.println("5");
        pw.write("abcde");
        pw.println("");
        pw.println("b");
        pw.write("fghijklmnop");
        pw.println("");
        pw.write("1");
        pw.println("");
        pw.write("q");
        pw.println("");
        pw.write("0");
        pw.println("");
        pw.println("");
        pw.flush();

        System.out.println("处理结束");
    }
}
```

使用curl命令测试：

```
[C:\~]$ curl  http://localhost:8888
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
 Left  Speed
100    17    0    17    0     0     17      0 --:--:-- --:--:-- --
:--:--   269
abcdefghijklmnopq
```

通过抓包可以看到符合预期的报文：



可以看出来，每一行十六进制的长度值也十分重要，我们调整下其中一行：

```
//指定分块的长度是1
pw.write("1");
pw.println("");
//修改点**********，显然不等于1，看会发生什么
pw.write("987q");
```

```
[C:\~]$ curl  http://localhost:8888
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
 Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --
:--:--     0
curl: (56) Malformed encoding found in chunked-encoding
abcdefghijklmnop9
```

错误提示是：在分块编码中发现格式错误的编码，很容易看出来，明明长度设置的是1，但是实际数据有4个字符，显然是有冲突的。

当然了，这个值一般不会错的，毕竟分块是服务端来做的，他心里是有数的。

Content-Encoding 和 Transfer-Encoding 二者经常会结合来用，其实就是针对进行了内容编码（压缩）的内容再进行传输编码（分块）。

另外值得注意的是，“Transfer-Encoding: chunked”和“Content-Length”这两个字段是互斥的，也就是说响应报文里这两个字段不能同时出现，一个响应报文的传输要么是长度已知，要么是长度未知（chunked）。

好了，《60 | 应用层篇：HTTP协议报文整体长什么样》这篇文章最后提的问题得到了答复。