HTTP 协议中的 **Transfer-Encoding**

May 04, 2015

18 Comments

Jerry Qu

专注 WEB 端开发

本文作为我的博客「HTTP 相关」专题新的一篇，主要讨论 HTTP 协议中的 **Transfer-Encoding**。这个专题我会根据自己的理解，以尽量通俗的讲述，结合代码示例和实际场景来说明问题，欢迎大家关注和留言交流。

文章目录

- Persistent Connection
- Content-Length
- **Transfer-Encoding**: chunked

Transfer-Encoding，是一个 HTTP 头部字段，字面意思是「传输编码」。实际上，HTTP 协议中还有另外一个头部与编码有关：Content-Encoding（内容编码）。Content-Encoding 通常用于对实体内容进行压缩编码，目的是优化传输，例如用 gzip 压缩文本文件，能大幅减小体积。内容编码通常是选择性的，例如 jpg / png 这类文件一般不开启，因为图片格式已经是高度压缩过的，再压一遍没什么效果不说还浪费 CPU。

而 **Transfer-Encoding** 则是用来改变报文格式，它不但不会减少实体内容传输大小，甚至还会使传输变大，那它的作用是什么呢？本文接下来主要就是讲这个。我们先记住一点，Content-Encoding 和 **Transfer-Encoding** 二者是相辅相成的，对于一个 HTTP 报文，很可能同时进行了内容编码和传输编码。

Persistent Connection

暂时把 **Transfer-Encoding** 放一边，我们来看 HTTP 协议中另外一个重要概念：Persistent Connection（持久连接，通俗说法长连接）。我们知道 HTTP 运行在 TCP 连接之上，自然也有着跟 TCP 一样的三次握手、慢启动等特性，为了尽可能的提高 HTTP 性能，使用持久连接就显得尤为重要了。为此，HTTP 协议引入了相应的机制。

HTTP/1.0 的持久连接机制是后来才引入的，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` 。所以实际上，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。

浏览器重用已经打开的空闲持久连接，可以避开缓慢的三次握手，还可

以避免遇上 TCP 慢启动的拥塞适应阶段，听起来十分美妙。为了深入研究持久连接的特性，我决定用 Node 写一个最简单的 Web Server 用于测试，Node 提供了 `http` 模块用于快速创建 HTTP Web Server，但我需要更多的控制，所以用 `net` 模块创建了一个 TCP Server：

```
require('net').createServer(function(sock) {
    sock.on('data', function(data) {
        sock.write('HTTP/1.1 200 OK\r\n');
        sock.write('\r\n');
        sock.write('hello world!');
        sock.destroy();
    });
}).listen(9090, '127.0.0.1');
```

启动服务后，在浏览器里访问 127.0.0.1:9090，正确输出了指定内容，一切正常。去掉 `sock.destroy()` 这一行，让它变成持久连接，重启服务后再访问一下。这次的结果就有点奇怪了：迟迟看不到输出，通过 Network 查看请求状态，一直是 pending。

这是因为，对于非持久连接，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；而对于持久连接，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。

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：

```
require('net').createServer(function(sock) {
    sock.on('data', function(data) {
        sock.write('HTTP/1.1 200 OK\r\n');
        sock.write('Content-Length: 12\r\n');
        sock.write('\r\n');
        sock.write('hello world!');
    });
}).listen(9090, '127.0.0.1');
```

可以看到，这次发送完数据并没有关闭 TCP 连接，但浏览器能正常输出内容并结束请求，因为浏览器可以通过 `Content-Length` 的长度信息，判断出响应实体已结束。那如果 Content-Length 和实体实际长度不一致会怎样？有兴趣的同学可以自己试试，通常如果 `Content-Length` 比实际长度短，会造成内容被截断；如果比实体内容长，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。

我们在做 WEB 性能优化时，有一个重要的指标叫 TTFB（Time To First Byte），它代表的是从客户端发出请求到收到响应的第一个字节所花费

的时间。大部分浏览器自带的 Network 面板都可以看到这个指标，越短的 TTFB 意味着用户可以越早看到页面内容，体验越好。可想而知，服务端为了计算响应实体长度而缓存所有内容，跟更短的 TTFB 理念背道而驰。但在 HTTP 报文中，实体一定要在头部之后，顺序不能颠倒，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。

Transfer-Encoding: chunked

本文主角终于再次出现了， `Transfer-Encoding` 正是用来解决上面这个问题的。历史上 `Transfer-Encoding` 可以有多种取值，为此还引入了一个名为 `TE` 的头部用来协商采用何种传输编码。但是最新的 HTTP 规范里，只定义了一种传输编码：分块编码（chunked）。

分块编码相当简单，在头部加入 `Transfer-Encoding: chunked` 之后，就代表这个报文采用了分块编码。这时，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。按照这个格式改造下之前的代码：

```
require('net').createServer(function(sock) {
    sock.on('data', function(data) {
        sock.write('HTTP/1.1 200 OK\r\n');
        sock.write('Transfer-Encoding: chunked\r\n');
        sock.write('\r\n');

        sock.write('b\r\n');
        sock.write('01234567890\r\n');

        sock.write('5\r\n');
        sock.write('12345\r\n');

        sock.write('0\r\n');
        sock.write('\r\n');
    });
}).listen(9090, '127.0.0.1');
```

上面这个例子中，我在响应头中表明接下来的实体会采用分块编码，然后输出了 11 字节的分块，接着又输出了 5 字节的分块，最后用一个 0 长度的分块表明数据已经传完了。用浏览器访问这个服务，可以得到正确结果。可以看到，通过这种简单的分块策略，很好的解决了前面提出的问题。

前面说过 Content-Encoding 和 **Transfer-Encoding** 二者经常会结合来用，其实就是针对 **Transfer-Encoding** 的分块再进行 Content-Encoding。下面是我用 telnet 请求测试页面得到的响应，就对分块内容进行了 gzip 编码：

```
> telnet 106.187.88.156 80

GET /test.php HTTP/1.1
```

```
Host: qgy18.qgy18.com
Accept-Encoding: gzip

HTTP/1.1 200 OK
Server: nginx
Date: Sun, 03 May 2015 17:25:23 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Content-Encoding: gzip

1f
�1�7H�1�7�1�7�1�7W(�1�7/�1�7I�1�7J

0
```

用 HTTP 抓包神器 Fiddler 也可以看到类似结果，有兴趣的同学可以自己试一下。

本文链接：https://imququ.com/post/**transfer-encoding**-header-in-http.html，参与评论 �0�3

--EOF--

发表于 2015-05-04 09:12:33，并被添加「HTTP、Keep-Alive」标签，最后修改于 2015-05-04 14:49:03。查看本文 Markdown 版本 �0�3

> 提醒：本文最后更新于 556 天前，文中所描述的信息可能已发生改变，请谨慎使用。

专题「HTTP 相关」的其他文章 �0�3

- HTTP Alternative Services 介绍 (Aug 21, 2016)
- 关于启用 HTTPS 的一些经验分享（三） (May 05, 2016)
- 如何压缩 HTTP 请求正文 (Apr 18, 2016)
- HTTP 协议中的 Content-Encoding (Apr 17, 2016)
- 三种解密 HTTPS 流量的方法介绍 (Mar 28, 2016)
- HTTP Public Key Pinning 介绍 (Mar 05, 2016)
- 关于启用 HTTPS 的一些经验分享（二） (Dec 22, 2015)
- 关于启用 HTTPS 的一些经验分享（一） (Dec 04, 2015)
- HTTP 代理原理及实现（二） (Nov 20, 2015)
- HTTP 代理原理及实现（一） (Nov 20, 2015)

« HTTP 请求头中的 X-Forwarded-For

Referrer Policy 介绍 »

Comments