Table of Contents

过去几年，我一直在利用HTTP知识解决我的产品和技术的问题。在这个过程中，我一点一滴的感到它的价值。我发现我的身边的人也是如此，并且我也看到了现今以HTTP为基础网络协议的应用越来越多。年初的一个机缘，我开始系统的投入研究HTTP。我拿到了HTTP的rfc，也买了一系列和HTTP有关的书。其中的最厚的砖头书，我买了4本，放到车上、办公室、家、另一个办公室各一本。我反复的啃这些资料，正反的对照，并且使用node.js写代码做一些技术验证。

接触的多了，眼光和以往作为纯粹的用户相比开始有些变化。于是，很自然的有一天，我听到一个声音说：“为它写点什么，这准是个有趣的事儿“。因为：

1. 它被使用的多，所以非常有用
2. 已有的资料，过于纠缠细节

HTTP 本该简单，这正是它的优势。可要搞得复杂也不难，只要按着协议文本的严格和滴水不漏，然后堆积细节即可，这正是目前的资料所处的状态。所以，我可以：

1. 直接了当。说清楚场景和价值，然后给出案例和验证代码
2. 高屋建瓴。有了架构，读者自己弄的明白细节

我会用第一章通过案例引出HTTP的架构、概念、主线条。随后的章节，我把大卸八块的HTTP的每个块给出场景，然后是交互过程。有些特别难区别的概念和交互过程，还会来些命令行和少量代码做进一步的解释。所以，这正是一本HTTP的书，是程序员写就的，也写给程序员的一本书。我希望它看起来像是好的代码风格一样的直接了当、详略得当。

针对读者，它准能帮你省下不少时间，因为你本来需要啃砖头大的几块资料的，并且是反复啃才行——因为我就是。

现在，它就摆在你的面前。

## 关于作者

作者：刘传君

创建过产品，创过业。不好动，读书机器。倾慕unix哲学，以此书略表致敬。

可以通过 1000copy#gmail.com 联系到我。

# 介绍

假设这样的一个场景：一个站点example.com上有一个hello.htm 的页面，位于站点的根目录。那么我可以打开一个浏览器，在地址栏内输入URL（http://example.com/hello.htm），确认回车后等待一些时间，就可以看到一个html页面呈现在浏览器内。

看到这样漂亮的表象后，一般用户就拿到了他需要的资源，并且就此止步。而具有好奇心的程序员则希望知道浏览器和服务器的交互对话的具体内容，那么这时候就需要学习HTTP协议了。正是这个协议规定了如何把客户端的请求打包为HTTP请求消息并发送给服务器，也是它规定了把一个响应打包成HTTP响应消息，然后送回客户端。

还是以hello.htm资源的获取过程为例，具体过程是这样的：

客户端软件打开到服务器的连接，发送文本如下：

GET /hello.htm HTTP/1.1  
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)  
Host: example.com  
Accept-Language: en-us  
Accept-Encoding: gzip, deflate

服务器软件接到这个请求消息，通过解析首行GET /hello.htm HTTP/1.1可以知道客户端发了一个GET请求，想要根目录下的hello.htm 资源，HTTP协议版本为1.1。服务器还可以根据第二行到空行之间的被称为首部字段区内得到更多客户端信息。比如看到Accept-Language: en-us，表明客户端接受美国英语的内容。

服务器软件根据资源定位符在服务器上定位并找到此资源，打包给出如下响应到客户端：

HTTP/1.1 200 OK  
Date: Mon, 27 Jul 2009 12:28:53 GMT  
Server: Apache/2.2.14 (Win32)  
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT  
Content-Length: 88  
Content-Type: text/html  
Connection: Closed  
  
<html>  
 <body>  
  
 <h1>Hello, World!</h1>  
  
 </body>  
</html>

客户端的浏览器接收到此响应消息完毕后开始解析。首先解析首行HTTP/1.1 200 OK，从此可以知道协议版本为1.1，状态码为200，由状态码可以知道这个请求在服务器已经成功处理。然后从第二行解析到空行，得到更多的消息首部信息。比如Content-Type: text/html 指示在本消息的主体内承载的是一个html文件；比如Content-Length: 88只是消息主体内承载的内容长度。然后在这些首部头字段的帮助下，就解析出实体内的html文件内容，并呈现html给用户。

考察这个过程，我们已经涉及到了不少新概念，包括首部字段、协议版本、资源、状态码等等。其中最重要的一个概念是消息。无论是请求消息还是响应消息都有4个部分构成：首行+首部字段行区+空行+消息主体。在HTTP协议中，总是由客户端发送请求消息给服务器，服务器则返回响应消息给客户端。接下来我们分别详细的考察请求消息和响应消息。

## 请求消息

通用的目的下，一个请求消息是由一个请求行、0到多个首部字段行、一个空行、随后的消息主体构成的。

### 请求行

请求消息的第一行就是请求行。它指明使用的请求方法、资源标示符、和HTTP 版本。

请求方法可以是 GET、POST、HEAD、PUT、DELETE、CONNECT、OPTIONS、TRACE的一个。本案例中使用的是 GET 。

* GET方法 表示我要请求一个指定名称的资源。
* PUT方法 表示如果指定URL不存在就创建它，否则就修改它。资源数据由消息主体提供。
* POST方法 表示要创建一个新的子资源，或者更新一个存在的资源。资源数据由消息主体提供。
* DELETE方法 表示我要删除一个指定名称的资源。
* CONNECT方法、OPTIONS方法、TRACE方法会在后面单独讲解。

请求方法是最关键的请求消息字段。因此按照请求方法对请求消息做出归类是非常方便的。这样的话，使用GET方法的消息，以后我们会直接简化它为GET请求；相应的，自然还有POST请求，PUT请求，DELETE请求等等。

PUT 和 POST 都可以创建和更新资源，如何选择？假设我们正在设计电子订单系统，那么：

* PUT /orders/1 创建订单号1的资源；如果此订单已经存在，那么就更新它。订单号是由客户端指定的。
* POST /orders 创建一个订单，新订单号由服务器指定。
* POST /orders/1 如果订单1存在就更新它。如果1不存在，应该抛出“资源未找到”错误。

特别请留意，重复执行 PUT 请求是不会影响服务器状态的。在HTTP 协议中，这个特性被称为幂等性。所以，如果可能优先使用PUT创建资源。在我们的订单案例中，如果重复一次提交POST创建子资源的请求，会导致创建两个订单。因此POST是不具备幂等性的。

### 首部字段

零行或者多行头字段行。可以用来传递客户端的更多信息，以及传递解析消息主体的必要信息。案例中的:

User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)  
Host: example.com  
Accept-Language: en-us  
Accept-Encoding: gzip, deflate

都被称为首部字段（或者成为头字段）。首部字段行由冒号分隔为两部分，左边的被称为头字段名，右边的是头字段值。比如Host: example.com，说明头字段Host的值为example.com。头字段的可选值是一个超长的列表，对应它们的值也各有不同。不过我们暂时先不关注这些细节。

### 空行（CRLF)

指示头字段区完成，消息主体开始（如果有消息主体的话）。

### 消息主体

消息主体是请求消息的承载数据。比如在提交POST表单，并且表单方法不是GET时，表单数据就是打包在消息主体内的。消息主体是可选的，本案例是没有主体的。

## 响应消息

响应消息由一个状态行、一个或者多个首部字段行、一个空行、消息主体 构成。

### 状态行

由http版本、状态码、状态描述文字构成。如HTTP/1.1 200 OK。状态码200表示成功。

状态码共有5组，分别是 100-199，200-299，300-399，400-499，500-599的范围。

* 200-299 成功。 指明客户端请求是正确的，并被成功执行。
* 300-399 重定向。指明客户端请求是正确的，不过当前请求资源的位置在别处，请再次定向你的资源位置，发起新的请求。
* 400-499 客户端错误。 指明客户端的请求是不正确的，可能是格式无法识别，或者URL太长等等。
* 500-599 服务器端错误。 指明客户端的请求正确，但是服务器因为自身原因无法完成请求。
* 100-199 信息提示。 这个系列的状态码只有2个，但是比较费解，会专门单独的做出解释。

状态码是最关键的响应消息字段，选择不同的状态码常常意味着不同的首部字段和主体。直接按照状态码对响应消息做出归类是非常方便的。这样的话，状态码为100系列的响应消息就可以简化为100型响应。相应的还有200型响应、300型响应、400型响应、500型响应。

### 首部字段

和请求消息类似，首部字段会包括服务器本身的一些信息指示、以及响应消息本身的元数据。本案例中这些行都是头字段：

Date: Mon, 27 Jul 2009 12:28:53 GMT  
Server: Apache/2.2.14 (Win32)  
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT  
Content-Length: 88  
Content-Type: text/html  
Connection: Closed

比如Server: Apache/2.2.14 (Win32)指示服务器使用的是Apache Server。而Content-Type: text/html 指明消息主体是html格式的资源。

### 一个空行（CRLF)

指示头字段完成。

### 可选的消息主体

案例中就是一个hello.htm文件的内容。

## 实验：表象下的细节

搭建一个简单的HTTP 服务器，并使用nc命令行发送请求，接受响应消息。

### 1. 建立一个http 服务器。

服务器代码（文件名为app.js)：

var express = require('express');  
var app = express();  
app.get('/', function (req, res) {  
 res.send('<a href="/test204">204</a> <a href="/test205">205</a> <a href="/test300">300</a>');  
});  
var server = app.listen(3000, function () {  
 var host = server.address().address;  
 var port = server.address().port;  
  
 console.log('Example app listening at http://%s:%s', host, port);  
});

### 2. 执行服务器

执行如下命令：

$node app.js

要是报错的话，可能因为你还没有搭建好环境。请移步到实验环境一章，首先搭建好环境，然后来运行此案例。

如果看到打印消息：

Example app listening at http//localhost:3000

就说明服务器准备完毕并且在3000端口等待客户端的连接。

### 3. 发送请求，查看响应

执行 nc（netcat），随后在控制台输入GET / ，两次回车，即可发出请求。随后nc会打印响应消息如下：

$ nc localhost 3000  
GET /  
  
HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: text/html; charset=utf-8  
Content-Length: 80  
ETag: W/"50-41mmSLl6PW+Zt5VLKLE2/Q"  
Date: Thu, 03 Dec 2015 08:54:23 GMT  
Connection: close

### 4. 降低版本试试

我们有提到了HTTP的版本。目前的HTTP版本有0.9、1.0 、1.1、2.0 之分。目前的主流使用版本是1.1版。使用nc和Node.js，不但可以查看1.1 的HTTP响应，还可以查看协议版本为 1.0 的http响应。可以对比查看两者的差别。

$ nc localhost 3000  
GET / HTTP/1.0  
  
HTTP/1.0 200 OK  
X-Powered-By: Express  
Content-Type: text/html; charset=utf-8  
Content-Length: 80   
ETag: W/"50-41mmSLl6PW+Zt5VLKLE2/Q"  
Date: Thu, 03 Dec 2015 08:52:45 GMT  
  
<a href="/test204">204</a> <a href="/test205">205</a> <a href="/test300">300</a>

# 术语

为了更好的讨论问题，我们引入一系列的相关概念：

## 资源（resource）

Web 资源是使用URL指向的Web内容。 1. 内容可以是静态的，如：文本文件，HTML文件，JPEG文件。可以查看RFC 2045文档，了解这个很长的清单。 2. 或者是动态的内容。如：摄像头的实时采集软件生成的动态影像，用户填写的电子网站订单。

## 资源类型

Web服务器会为所有HTTP资源赋予一个类型，以便于HTTP软件处理消息主体。如，用text/html标记html。可以再看两个案例：

text/plain :ASCII文本文档  
image/jpeg :JPEG版本的图片

每个条目对应的文本格式由表示一种主要的对象类型和一个特定的子类型，中间由一条斜杠来分隔。

非常多的资源类型和文本标记的对应关系，一起构成了一个超长的清单，并且由RFC 2045标准化。此标准被称为MIME。MIME是Multipurpose Internet Mail Extension的缩写。虽然名称很长，但是含义简单，就是用来指定消息内的实体类型的。之所以有Mail字样，是因为最初设计是为了Mail的异构系统交换文档的。

## 资源标示符

URL是一种资源位置标示方法。URL描述了一个资源在服务器上的位置。这就是一个合法的URL：http://example.com/part/index.htm

1. 第一部分：方案(scheme)。指明了访问资源所使用的协议类型。这部分通常是HTTP协议(http://)。
2. 第二部分：服务器地址(比如，example.com）。
3. 其余部分指定了Web服务器上的某个资源(比如，/part/index.htm)。

URL 是 Uniform Resource Locator （统一资源定位符），用来指向互联网的一个资源。 一个典型的URL ：http://www.example.com/index.html, 指示了协议 (http), 一个主机名(www.example.com), 和一个资源名 (index.html)。

当在地址栏输入此资源名并回车后，用户代理会把URL解析，把必要的信息以HTTP协议的要求，打入请求消息内。以http://www.example.com/index.html，变成

GET index.html HTTP/1.1  
host:www.example.com  
CRLF

打开到www.example.com的tcp连接，并发送此请求消息给服务器，然后等待服务器响应并解析显示给用户。

## HTTP事务

一个HTTP事务由一条请求消息和一个响应消息构成。

## HTTP方法

HTTP支持几种不同的请求命令，这些命令被称为HTTP方法(HTTP method)。每条HTTP请求报文都包含一个方法。

## 状态码

每条HTTP响应消息返回时都会携带一个状态码。状态码是一个三位数字的代码，告知客户端请求是否成功，或者是需要采取其他行动。

## 消息

从Web客户端发往Web服务器的HTTP报文称为请求消息。从服务器发往客户端的消息称为响应消息。HTTP报文包括三部分：

起始行 首部字段 主体

如发送一个hello.htm 的资源给客户端，请求消息是：

GET /hello.html HTTP/1.1

请求消息只有起始行，指明使用的HTTP方法，资源的URL，以及协议的版本。没有首部字段和主体。

响应消息为

HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: text/html; charset=utf-8  
Content-Length: 22  
ETag: W/"16-FmHX0hamHjYkHeAP/7PfzA"  
Date: Thu, 03 Dec 2015 09:54:01 GMT  
Connection: close  
  
<h1>Hello, World!</h1>

这个消息第一行为起始行，指明协议版本 ，状态码（200表示成功）和状态说明（OK）。接下来一直到空行之间都是首部字段，用来说明服务器，资源类型，内容长度，生成文档时间等。空行后就是主体。这里就是一个html文件的内容。实际上，主体可以承载任何内容，而不限于文本。

# 工具

我常常发现，对于同一个概念，文字描述和代码验证是各有擅长的。有些文字难以表达的概念，要是换用代码或者命令行来表达却常常可以干净利索。所以我会适时的使用代码来补充说明一些描述性文字，协同帮助读者建构概念图景。

我选择的编码工具是Node.js。因为本书研究的是特定的HTTP领域，我使用的命令行最关键的工具是netcat 。

## Node.js

选择Node.js 因为它小巧，跨平台，并且内置有我们需要的HTTP标准库。我远程办公使用的是一台Mac Book Air 笔记本，在公司时我使用的是一台 PC，Node.js 的跨平台特性，可以让我自由的切换环境而不会被一些细节感到懊恼。

安装Node在不同平台是不一样的，不过基本上雷同。无论Windows还是Mac，都有安装程序，一步步来就是，并没有什么特别需要讲的。

在Node.js之内，我们还需要它的一个框架，以便编写Http应用可以更加简易。这个框架叫做Express.js 。安装完 Node.js后，可以通过Node.js 内置的包管理工具（npm）来安装和初始化express，以便运行我给出的代码案例。只要：

$npm i express

就可以构建和准备好环境了。

## netcat

同样能够解决问题的情况下，我更喜欢命令行而非图形界面的风格。这样做对读者有利，因为可以降低眼球识别负担——看命令输入和执行结果常常是最小化高相关性的信息输出。而使用图形化的浏览器的话，就不可避免的需要大量的截图，以及操作过程的说明（点击、点击、下拉、拖放），并且这些截图几乎必然会包含了很多和问题无关的信息。

我经常使用的netcat命令就是一个命令行工具。它专用于 http 协议测试验证，堪称此领域的万能瑞士军刀。 它可以在命令行中发送请求消息，接受并打印响应消息。关于netcat命令，可以在这里找到更多的信息： http://nc110.sourceforge.net。netcat命令行文件名为nc（Windows 操作系统上的文件名为nc.exe)，可以输入nc执行此命令。此命令很简单，所以你未必需要专门学习，而可以跟着教程走，自然的学习它的用法。

## 代码

本书涉及的代码其中有些简单的会直接在书内贴出。有些复杂点的，我把它放到一个包内，这个文件包可以在此处下载：https://github.com/1000copy/httpbookcode

# 历史

时间回到1989年。

一个物理学家想要做一个可以在不同地点的科学家之间分享知识的超文本文档网络。他称之为 World Wide Web （WWW就是世界范围的网络——多么宏大的名字）。随后他编写了让世界网络成为现实的服务器和客户端。

这样一个简单的想法，逐步的吸引了更多的人投入进来，构成了如今互联网的最异彩纷呈的部分。Web，就被启动了。

这个物理学家就是Tim Berners-Lee。

超文本本身被称为 HTML。它的"超"字，来源于一个最显著的特点：在一个 HTML 文档内，除了本文档的内容外，还可以通过URL指向其他文档、图片、视频等不同的媒体格式。

为了传递 HTML 文档，Berners-Lee设计了一套应用层协议，随后被标准化，正式协议文本的第一版为HTTP/1.0(1996年）。之前的协议，虽然没有被正式规范，为了沿袭兼容，被称为 HTTP/0.9。

有趣的是，当年Berners-Lee采用的NeXTStep环境经过多年的小众状态，如今以iOS的新包装浴火重生，而HTTP则因为互联网的潮流快速成为主流的应用层协议。

WWW技术包括了HTTP协议、HTML标准，以及作为实现的众多的服务器、客户端、代理软件。WWW位于应用层，和它处于同一层面的还有很多包括Email、FTP、Telnet等。

在这里，我们关心HTTP（超文本传输协议）。

在Berners-Lee的年代，已经存在了一些应用层协议，包括Email、FTP、News，但是他提出了现有协议的局限和新协议的更多需求。

* Email 用来由作者发起，发送一些暂态的消息给少量的接收者。
* FTP 可以传递文档，但是在响应者一端只能做很少的处理。
* News 可以广播暂态消息给大量听众。

基于这样的认识，HTTP 应该能够：

1. 只要一部分的文件传输功能。HTTP 只要客户端单向发起，服务端单向响应。
2. 自动在客户端服务器之间协商格式。
3. 引导客户端到别的服务器的能力。
4. 可以由作者放置长效文档，访问者可以输入URL，访问得到这个文档。

这些看来极为基础的思想成为当前HTTP的种子。

# 请求

HTTP协议只允许客户端发起请求，也只允许服务器针对请求返回响应。我们已经看到了请求的样例，现在我们具体了解HTTP请求消息的构成。请求消息包括一个首行、首部、请求主体。

Request = Request-Line CRLF  
 \*Header CRLF  
 CRLF  
 [ message-body ]

## 首行

请求行由请求方法、请求URL、一个HTTP版本构成。

Request-Line = Method SPACE Request-URI SPACE HTTP-Version CRLF

## 请求方法

HTTP/1.1支持方法包括：

Method = "OPTIONS"   
 | "GET"   
 | "HEAD"   
 | "POST"   
 | "PUT"   
 | "DELETE"   
 | "TRACE"   
 | "CONNECT"   
 | extension-method  
 extension-method = token

HTTP 0.9协议版本中，GET方法是唯一被支持的HTTP请求方法，用来向服务器请求一个指定URL关联的资源。随后，为了优化的目的，HEAD方法被加入进来；为了支持向服务器提交数据，引入了POST、PUT、DELETE方法。分别在语义上表达更新、创建和删除指定的资源。对于任何一个通用目的的HTTP服务器，GET、HEAD方法是必须实现的，其他的方法都是可选实现的。可以采用OPTIONS方法去查询一个服务器资源所支持的请求方法清单。

## 首部

首部包括零到多个首部行，每个首部行由“：”分隔，左边的文本被称为首部字段，右边的文本被称为首部值。采用文本的首部让HTTP变得容易扩展。只要客户端和服务器做好约定，就可添加自己需要的新的请求方法。不过，对于如何扩展的内容，不在本书需要讨论的范围。

## 消息主体 message-body

消息主体就是消息的承载内容。单独分章节随同请求方法来做说明

## 总结

* GET 表示我要请求一个由URI指定的在服务器上的资源。
* PUT方法 表示如果指定URI资源不存在就创建它，否则就修改它。
* POST方法 表示要创建一个新的子资源，或者更新一个存在的资源。
* DELETE表示我要删除一个由URI指定的资源。
* HEAD 和GET一样，但是仅仅返回指定资源响应的头部分，而不必返回响应主体
* OPTIONS 查询目标资源支持method的清单。
* TRACE 查询到目标资源经过的中间节点。用于测试。
* CONNECT 建立一个到URI指定的服务器的隧道。

## GET方法

GET方法用来获取URL指定的资源。这个URL指向可以是一个静态文件，也可以是一个数据生成软件产生的动态内容。

如果GET请求在首部区包含了条件获取字段，那么GET 请求就具体化为条件获取(conditional GET)。条件字段包括： If-Modified-Since、 If-Unmodified-Since、 If-Match、 If-None-Match、 If-Range 。条件获取请求下，只有满足了条件的资源才会传递响应主体到客户端。这样的首部字段搭配使用就可以达成缓存的目的。

如果GET 请求包括了范围条件，那么GET请求就被具体化为局部获取(partial GET)。使用局部获取，对于大文件可以分块传递从而提高传输效率。要是你在做一个视频播放应用，那么可以只传递用户跳播的视频片段，提供更好的用户体验。

以访问hello.txt获取其局部为例。使用GET方法，并通过Range头字段指定指定获取文件的开始字节索引和结束字节索引发出如下局部请求：

GET /hello.txt HTTP/1.1  
Range: bytes=0-2

服务器响应:

HTTP/1.1 206 Partial Content  
Content-Type: text/html  
Content-Range: bytes 0-2/12  
Content-Length: 3  
  
hel

本案例中的Content-Range的值是一个内容为 bytes 0-2/12 的字符串，这里需要对它稍作解释：分隔符“/”的前面一组数字表明本次返回位置范围，“/”后的数字指明资源的总大小。

### 实验

#### 环境准备

在code代码目录内找到hello.js ，代码如下：

并且通过node执行：

node hello.js

#### 验证GET和HEAD的差别

然后我们使用nc发起get：

GET /hello.htm HTTP/1.1

可以看到响应如下：

HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: text/html; charset=utf-8  
Content-Length: 22  
ETag: W/"16-FmHX0hamHjYkHeAP/7PfzA"  
Date: Thu, 03 Dec 2015 09:54:01 GMT  
Connection: close  
  
<h1>Hello, World!</h1>

如果使用HEAD方法：

HEAD /hello.htm HTTP/1.1

响应就只会发送响应消息的头，而不会发送响应消息的主体了。如下：

HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: text/html; charset=utf-8  
Content-Length: 22  
ETag: W/"16-FmHX0hamHjYkHeAP/7PfzA"  
Date: Thu, 03 Dec 2015 09:54:01 GMT  
Connection: close

#### 验证：条件获取

本实验中的GET /hello.html的响应首部字段内，有了一个ETag的字段，它是文档内容变化的标识。如果文档改变了，这个标识就会改变。服务器发送它的目的是为了支持客户端的条件获取。客户端可以发起一个GET ,并使用条件获取首部字段，以便指示服务器如果文档改变就发送，否则就会使用本地的缓冲。这样可以省下一些并非必要的网络流量。

继续这个案例，我们可以搭配ETag和If-None-Match头字段，达到使用缓存的效果。客户端可以通过 If-None-Match 头字段指明，如果文档标识不匹配，就发送新文档来；否则，服务器就会发送 304 Not Modified：

GET /sample.html HTTP/1.1  
Host: example.com  
If-None-Match: W/"16-FmHX0hamHjYkHeAP/7PfzA"

如果服务器发现指定的标识是匹配的，那么服务器响应：

HTTP/1.1 304 Not Modified  
X-Powered-By: Express  
ETag: W/"16-FmHX0hamHjYkHeAP/7PfzA"  
Date: Thu, 03 Dec 2015 09:54:01 GMT  
Connection: close

还可以搭配 Last-Modified(响应首部字段），If-Modified-Since（请求首部字段）,If-UnModified-Since（请求首部字段） 字段达到按照修改时间来让服务器决定发送文档或者客户端使用缓存文件。比如：

GET /sample.html HTTP/1.1  
Host: example.com  
If-Modified-Since: Thu, 03 Dec 2015 09:54:01 GMT

指示服务器如果文件并没有在指定时间前修改过的话，那么服务器响应一个无消息主体的消息即可：

HTTP/1.1 304 Not Modified  
X-Powered-By: Express  
Last-Modified: Thu, 04 Dec 2015 09:54:01 GMT  
Connection: close

### 验证：局部获取(partial GET)

录入以下命令，查看结果。

cd code  
node partial-get.js  
  
$nc localhost 8000  
GET /hello.txt HTTP/1.1  
Range: bytes=0-2

如果你有兴趣不妨阅读代码来了解它的具体实现。

## POST方法

POST 方法常常用来提交表单数据。假设有一个表单，其html如下：

<form enctype="application/x-www-form-urlencoded" />  
 <input type="text" name="user" action="/example"/>  
 <input type="password" name="password" />  
 <input type="submit"/>  
</form>

当在两个文本框内分别填写1，2，然后点击提交的时候，我们需要传递形如：

username :1  
password :2

的内容给服务器。

本案例中，我们需要注意到Form的属性enctype指定了一个看起来有些复杂的值： application/x-www-form-urlencoded，这个值指示Form经由HTTP提交到服务器的消息主体采用的编码方式。最后的提交请求数据就是这样的：

POST /example HTTP/1.1  
  
username=1&password=2

此编码方式把提交字段和值用“=”分隔，多个提交项目之间用“&”分隔，空格会被使用“+“替代，其他不是字母和数字的字符会用url encoding 来编码，替换为%HH，比如%21 表示 “！”。看到案例后，就知道看起来复杂的enctype其实还比较简单的。

只不过使用这个编码方式的话，一个非字母数字的二进制值会需要3个字符来表达。对于较大的二进制数据来说这样的作法实在有些浪费。因此HTTP标准也引入了新的封装格式：multipart/form-data。我们依然可以通过Form 的enctype属性指定此新的打包格式：

<form enctype="multipart/form-data" />

这样指定的封装类型，请求消息会变成：

POST /example HTTP/1.1  
Host: example.com  
Content-Type: multipart/form-data; boundary=---------------------------9051914041544843365972754266  
Content-Length: 554  
  
-----------------------------9051914041544843365972754266  
Content-Disposition: form-data; name="username"  
  
1  
-----------------------------9051914041544843365972754266  
Content-Disposition: form-data; name="password"  
Content-Type: text/plain  
  
2  
-----------------------------9051914041544843365972754266--

我们要留意的是，在请求消息内的第三行，也就是Content-Type字段的值内，指定请求主体内的格式为multipart/form-data，同时在“；“后还有一个boundary子字段，它的值是一个字符串。此字符串的目的就是指定每个表单字段的开始和结束。在两个boundary之间，就是一个字段的内容。每个字段内可以指定字段的名称和类型，然后加上一个空行后内容开始。直到遇到一个新的boundary结束。

boundary字符串常常由若干个连字符加上一个随机字符串构成，由客户端软件生成，算法可以自己决定，只要不会在内容中出现就可以。如果对冲突感到担心，还可以在生成后由软件在请求消息体内搜索此字符串，如果发现有相同的话，就在此基础上继续添加一个随机字符，再执行此过程直到不再出现即可。

## OPTIONS方法

请求方法OPTIONS用来查询URL指定的资源所支持的方法列表。

请求案例：

OPTIONS /example HTTP/1.1

响应：

HTTP/1.1 OK  
Allow:GET,POST,PUT,OPTIONS

本请求案例中，请求的就是/example 指定的资源所支持的方法。响应案例给出的是此资源支持的请求方法，列表为GET、POST、PUT、OPTIONS。

服务器应该返回405 (Method Not Allowed)响应，如果使用的请求方法是为服务器所知的、但是并不被允许的话。 服务器应该返回 501 (Not Implemented) ，如果请求方法没有被实现的话。

## PUT方法和DELETE方法

PUT方法的意图，是对URL指定的资源进行创建,如果资源存在就修改它；相应的，DELETE方法的意图是对URL指定的资源进行删除。

可是要做到这两件事，只是POST就够了。那么为何在HTTP标准内还有PUT和DELETE呢？我将会举个业务案例，给出只是使用POST、和全面使用HTTP方法的效果，以此对比来说明问题。

假设我们手里有一个电子商务网站，那么我们必然需要提供订单的维护，包括创建、修改、删除、查询。

对于查询，我们可以采用GET 方法，并提供订单编号为参数：

GET /order/1

使用GET 方法在这里是恰如其分的，因为GET隐含着只是查询，而并不会影响服务器的状态。

接下来，我们更新一个订单，假设编号为2：

POST /order/2  
Content-Type:text/json  
  
{  
 "date":"2015-12-07"，  
 “guest":"frodo",  
 [{  
 "item":"The King of Ring",  
 "count":"2",  
 "price":"100"  
 }]  
}

这里使用POST也是合适的。因为POST的语义中包括对资源进行更新。

但是对于创建和删除，我们就有不同的做法了。

首先看创建。我们常见的方法依然是使用POST，但是在URL内（或者请求消息主体内的参数）要和更新操作不同，以便区别两者：

POST /order/2/create  
Content-Type:text/json  
  
{  
 "date":"2015-12-07"，  
 “guest":"frodo",  
 [{  
 "item":"The King of Ring",  
 "count":"1",  
 "price":"100"  
 }]  
}

这样做是可以达成业务需求的。然而，我们可以有更好的选择。这个选择不但能够完成功能的需求，还能够满足Restful App的规范化需求。我们可以在创建资源是选择PUT:

PUT /order/2  
Content-Type:text/json  
  
...

删除的时候也是一样。典型的请求消息：

POST /order/2/delete  
或者有人这样  
  
POST /order/2/remove  
  
或者还有人这样  
  
POST /order/remove\_order/2

而如果我想要满足Restful app规范，选择就是一个样：

DELETE /order/2

稍微做过总结，Restful App 的方案的好处是看得到的：

1. 把操作意图表达在请求方法内
2. 把操作意图从URL中分离出来

相对于使用POST做全部的提交数据的做法而言，这样的做法经过一个著名框架（Ruby on Rails)的首倡，目前得到了很多框架的附和，堪称一时风气之先。这样做可以有语义上的一致性，避免不同程序员选择的不同方案导致的不必要的混乱。

尽管Web Form的Action字段只能指定为GET和POST，本身没有提供PUT、DELETE方法，但是可以通过Form隐含字段来细分POST为 PUT、POST、DELETE，比如约定一个叫做\_method的字段，其值可以在PUT、POST、DELETE、POST之间选择一个：

<form method="post" ...>  
 <input type="hidden" name="\_method" value="PUT | POST | DELETE " />  
 ...

这样就可以由框架实现完整的对资源操作的不同类型。在使用框架的基础上，应用可以直接享受到完整的GET、PUT 、POST、 DELETE语义。

## 连接方法

在当前已经建立HTTP连接的情况下，CONNECT 方法用来告知代理服务器，客户端想要和服务器之间建立SSL连接。

要是没有HTTP代理服务器，客户端可以使用Connection头字段来表达客户端要升级到SSL的请求：

GET http://example.bank.com/acct\_stat.html?749394889300 HTTP/1.1  
Host: example.bank.com  
Upgrade: TLS/1.0  
Connection: Upgrade

这样服务器接收到此消息即可发送：

HTTP/1.1 101 Switching Protocols  
Upgrade: TLS/1.0, HTTP/1.1  
Connection: Upgrade

表示确认。一次握手后，双方认可，这个http连接之后就可以发送SSL流量了。

如果中间有HTTP代理服务器的话，情况就不同了。因为我们使用的Connection头字段是hop-by-hop（逐跳）的，这个头字段会被代理服务器认为是在客户端到代理服务器之间的协议升级。于是，代理升级连接协议，解析并删除此字段后继续转到服务器。这样服务器是收不到这个首部的，本来希望客户端和服务器直接达成SSL 升级，实际上却变成了客户端和代理服务器之间的SSL升级，这是违背Connetion字段的本意的。

为了解决此问题，HTTP 引入了 Connect 方法。客户端使用如下消息，通知代理服务器，去做一个连接到指定的服务器地址和端口:

CONNECT example.com 443 HTTP/1.1

代理服务器随后提取CONNECT 方法指定的地址和端口（这里是 example.com 443 ），建立和此服务器的SSL连接，成功后随后通知客户端，需要的连接建立完毕：

HTTP/1.1 OK

之后，代理服务器简单的转发客户端的消息到服务器，以及转发服务器来的消息给客户端。因为它只是转发，它就变成了一个透明代理。透明代理和一般代理是不同的，一般的http 代理不是仅仅转发，还需要解析头字段、考虑是否缓存、添加Via头字段等工作，而透明代理只管转发，不管内容和格式。

升级到 SSL 只能有客户端发起。如果服务器希望升级，可以通过状态码426 upgrade required 告知客户端。

因此，客户端和服务器之间要升级到SSL，就必须区分两种情况，一种是两者之间存在代理服务器，就需要用Connect 方法；否则使用第一种方法（使用Connection 头字段的方法）即可。

有很多资料提到 Connect 方法建立的是一种隧道，叫做SSL隧道。我觉得这个说法不妥，因为和隧道的定义是不符的。

隧道被用来在一个协议上承载一个系统本来并不支持的外部协议。一个协议内嵌套另一个协议，就像一个管道嵌入另一个管道，因此取名为隧道。 这就意味着，完全可能在TCP上承载IP、IPV6协议，或者在TCP上承载NetBIOS协议。

HTTP隧道使用普通的请求方法POST、GET、PUT 和DELETE来实现对被承载协议的包装。HTTP隧道服务器运行在被限制的网络外，执行一个特别的HTTP服务器角色。HTTP隧道客户端运行在被限制的网络内。当任何网络流量传递到客户端，它就把流量作为HTTP 实体，然后加上HTTP首部，传递到外部的服务器；后者解开包，并执行这个原始的网络请求。对这个请求的响应获得后，也会被加上HTTP响应首部，传递会给客户端。对此实现感兴趣的话，可以可以参看node-http-tunnel，或者gnu 的htc、hts，他们都是开放源代码的，并且有还不错的文档帮助你。

我们再进一步查看CONNECTION连接和HTTP隧道在防火墙面前的差异。在使用CONNECT 方法时，一旦连接建立成功后，传递的内容如加密流量是在RAW SOCKET上，对于可以识别HTTP包格式的防火墙，可以知道这个流量尽管可能是80端口，但是并非HTTP流量，因为它的格式根本不遵循HTTP标准内的请求和响应包格式。而HTTP隧道的流量到来时，即使可以识别包格式的防火墙也会认为它传递的就是HTTP流量，因为非HTTP流量本来就是包装HTTP消息内的。因此，CONNECTION 方法建立的SSL通道并不是隧道。把Connect建立的通道叫做隧道会导致认知的混乱。

所以理解 Connect 方法，需要对比和区分以下内容：

1. Http 可以通过Connection：upgrade的方法升级到TLS。仅仅使用于无代理服务器的情况。
2. Connect 方法是 http upgrade tls 的一个替代。针对有代理的情况。
3. Connect 方法成功返回后，中间的http代理变成了透明的代理：不再使用HTTP协议解析数据包和修改数据包，而是简单的转发流量。

这样就清晰了。

## 参考资料：

http tunneling  
  
https://blog.udemy.com/http-tunneling/  
  
HTTP tunnel  
  
https://en.wikipedia.org/wiki/HTTP\_tunnel  
  
Tunneling protocol  
  
https://en.wikipedia.org/wiki/Tunneling\_protocol  
  
Networking 101: Understanding Tunneling  
  
http://www.enterprisenetworkingplanet.com/netsp/article.php/3624566/Networking-101-Understanding-Tunneling.htm  
  
windows tcp tunnel  
http://www.codeproject.com/Articles/14617/Windows-TCP-Tunnel  
  
  
L2TP layer 2 tunneling protocols  
https://en.wikipedia.org/wiki/Layer\_2\_Tunneling\_Protocol  
  
Tunneling TCP based protocols through Web proxy   
https://tools.ietf.org/html/draft-luotonen-web-proxy-tunneling-01

# 响应

服务器执行了客户端请求后，需要给客户端以响应，告知它请求的执行状态、以及放置在消息主体内的执行结果。

响应消息构成：

Response = Status-Line   
\*(header CRLF)   
CRLF  
[ message-body ]

### 状态行（ Status-Line)

由HTTP版本、状态码、状态说明共三个字段构成：

Status-Line = HTTP-Version SPACE Status-Code SPACE Reason-Phrase CRLF

状态说明（Reason-Phrase）就是一个给人类阅读的内容，对于处理逻辑并不重要。 HTTP版本、状态码则直接影响客户端如何处理此响应消息。状态码是一个三位的数字，用来告知客户端请求的处理结果。三位数字的第一位是一个分类，指明状态码的类型，以此数字把状态码分为5类：

* 1xx: 信息类 。
* 2xx: 成功。请求被成功的接受或者理解，或者执行。我们常见的200 OK就是这个分类内的。
* 3xx: 重定向 - 为完成请求，需要进一步的行动。我们常见的301 Redirect就是这个分类内的。
* 4xx: 客户端错误。客户端提交的数据错误，不能被理解或者接受等等。我们常见的404 Not Found 就是这个分类内的。
* 5xx: 服务器错误。错误发生了，是服务器的问题，和客户端无关。

本书内，我们把响应消息按照状态码类别的不同分为6节来阐释。2xx-5xx 的各有一节。而其中1xx内的状态码目前只有两个，但是因为它们俩都相对比较复杂，因此拆分为两节，以便说明和理解。

### 首部字段

和请求消息的首部字段类似，只是因为请求和响应的差异，可以选择的首部字段各有不同。这些字段会在随后的章节，随同状态码分类来做介绍。

### 消息主体 （message-body）

消息主体承载响应请求消息的具体内容。会在随后的章节，随同状态码分类来做介绍。

## 200系列

200 系列的状态码都可以表示请求已经成功处理。可是作为成功的语义，细节可以各不相同。并且对 HTTP 客户端有不同的动作指示，比如是否刷新当前页面内容等。

首先看下本类型响应下的具体状态码列表：

200 OK  
201 Created   
202 Accepted   
204 No Content  
205 Reset Content  
206 Partial Content

### 200 OK

可能是最常用的状态。它指明请求已经成功完成。

### 201 Created

201 Created 会比200 OK 有更加具体的语义。201指明请求成功且创建了一个资源，因此201常常配合PUT方法使用，因为PUT方法的语义上就是创建一个资源。

### 202 Accepted

202 Accepted 会比200 OK 有更加具体的语义。202表明请求成功被接受，但不一定已经完成资源创建或者修改，而只是被接受，可能还有服务器的后续的处理。

### 204 No Content

表明请求处理成功，但是作为服务器并不想要提供消息在消息主体内，或者并没有什么消息主体需要提供。 比如使用DELETE请求情况下，如果服务成功完成，可以返回204 No Content。此场景下就是告诉客户端：“你的DELETE请求已经完成，但是因为这个资源已经被删除，所以，也就没有什么需要返回的消息”。

204 对用户代理（浏览器）也是有意义的。用户代理收到204，就不应该引发请求的文档的当前视图（就是不要去刷新当前文档，也不要导航到别的URL）。这就意味着，如果有一个 HTML Form然后提交，如果服务器返回的状态是204状态，那么浏览器不可以刷新窗口或者到其他的页面。所有的 Form 输入的内容都不要改变。当然这样的做法在用户代理中很少有人如此实践。因为用户点击发生了，和服务器的交互也发生了，而用户界面却对此毫无响应，那么这样的做法显然会让用户感到迷惑。

可要是在ajax的应用上下文，这样做就比较有价值了。ajax应用获得204状态返回就可以提示用户操作已经成功。并且如同204状态码的意图，不需修改当前 Form 的任何值。因为服务本来不需要返回任何具体数据，它只需要告诉客户端请求已经成功处理。

### 205 Reset Content

此状态码告诉客户端请求已经成功执行。不同于204，它的意图是要告诉客户应该清除Form的内容或者刷新用户界面。具体说，我可以填写Form，提交后，如果接到了205响应，就应该重设Form，然后初始化一个新的输入。

事实上，并没有什么浏览器支持这样的意图：浏览器要么把205当成204，要么当成200。然而，对于ajax应用就可以实现这样的意图：ajax应用接到205码，用户界面应该把数据设置到默认值。如果是 Restful App ，在数据输入场景下，204响应适合对一条记录做一系列的编辑；205更适合输入一系列的记录；故而，Restful App的建议更加尊重http的设计本意。

### 206 Partical Content

它的存在目的是为了支持大文件的分段下载。当客户端发起资源范围请求，服务器就可以返回206型响应，告知客户端操作成功并且返回部分内容。见如下案例：

HEAD /large.jpg HTTP/1.1  
Host: example.com  
   
HTTP/1.1 200 OK  
Accept-Ranges: bytes  
Content-Length: 3980  
  
GET /large.jpg HTTP/1.1  
Host: example.com  
Range: bytes=0-999  
   
HTTP/1.1 206 Partial Content  
Accept-Ranges: bytes  
Content-Length: 1000  
Content-Range: bytes 0-999/3980  
   
{binary data}  
  
GET /large.jpg HTTP/1.1  
Host: example.com  
Range: bytes=1000-  
   
HTTP/1.1 206 Partial Content  
Accept-Ranges: bytes  
Content-Length: 2980  
Content-Range: bytes 1000-3979/3980  
   
{binary data}

上面的案例中，首先通过HEAD 方法查询资源的大小、以及查询是否支持分段下载。服务器如果支持分段下载就通过Accept-Ranges: bytes 的首部字段指示它是支持的。接下来，客户端就可以通过首部字段 Range 来指定要获取资源的范围。而服务器通过206 Partial Content指示获取成功、本次获取为部分内容、和本次获取部分资源的范围。

## 300系列

3xx系列的响应涉及的状态码除了304 Not Modified 之外都是用于重定向的。我们首先查看3xx系列的重定向状态码和描述：

* 300 Multiple Choices 客户端请求了实际指向多个资源的URL。
* 301 Moved Permanently 请求的 URL 已移走。
* 302 Found 请求的URL临时移走
* 303 See Other 客户端应该使用指定URL
* 307 Temporary Redirect 客户端应该临时定位到指定URL

于是，看起来并不复杂的重定向，稍微对比就会感到很混淆的。特别是301/302 ，303/307 一组，好像根本就是重复的。我们会在下文解释。

### 300 multiple choices

含义在于——同样的一个URL，可以对应多个实际的资源。比如同样的软件下载可以有多个平台的版本，或者多种打包压缩格式。又比如，同样的文档可以有不同的文档格式 。客户端可以在这些结果中，根据自己的情况作出自动的选择（比如中文用户就自动选择中文文档），或者给出列表，提交给最终用户选择。

可是，标准内并没有给出具体的多个选择项的格式。因此，该状态码很少被标准的web服务器和用户代理使用。我看到的两本和http有关的书，讲到状态码 300 的时候，一本语焉不详，一本干脆略过，想来也是这样的原因。

但是程序员完全可以在300状态码的基本含义情况下，具体化多资源构造的格式，在自己的应用的客户端和服务器之间遵守，然后实现应用的特定目的。比如，在实体主体内自定义如下的格式：

HTTP/1.1 300 Multiple Choices  
Date: Tue, 11 Jun 1996 20:02:21 GMT  
Content-Type: text/html  
Content-Length: 130  
  
<h2>Multiple Choices:</h2>  
<ul>  
<li><a href=paper.1>HTML</a>  
<li><a href=paper.2>Kindle</a>  
<li><a href=paper.3>Doc</a>  
</ul>

要么提供选择界面 ，由用户选择其中一个、要么由用户代理自动选择，然后重定向到这个资源去。

### 301 Moved Permanently

说的是客户端请求的 URL 对应的资源已经被挪到其他位置，这个新位置已经在响应消息的LOCATION 头字段内指定。如果你的书签使用了这个URL，那么应该由用户代理自动更新到新的位置。下次访问也希望使用新的URL。

客户端请求:

GET /abc HTTP/1.1  
Host: www.example.org

服务器响应:

HTTP/1.1 301 Moved Permanently  
Location: http://www.example.org/def

## 302 Found

说的是你当前访问的URL对应的资源暂时被移动到一个新位置，这个新位置在Location头内指定。和301不同的是，302并不影响你的书签，你也不必下次访问新的URL，因为这个变化是暂时的。实际上，这个状态码在HTTP 1.0引入，本来命名就是 302 Moved Temporarily，以便和301对照使用。

客户端请求：

GET /abc HTTP/1.1  
Host: www.example.org

服务器响应：

HTTP/1.1 302 Found  
Location: http://www.example.org/def

然而，对于这个状态码，标准本来希望的是保持两次请求的请求方法一致的。就是说，原来用POST重定向就用POST；原来用GET方法请求的，重定向后也继续用 GET 方法。而实际上，众多的用户代理都做了和标准不一致的实现：不管原来引发请求的是POST，还是GET，在重定向后都改成了GET方法。这是不恰当的实现，但是因为大家都这么实现，故而它反而成为了事实上的标准。为此，在修订 HTTP 1.1版本时就又引入了303 See Other 和307 Temporary Redirect 状态码，以便解决标准和实现不一致引发的语义问题。请继续阅读，以便理解这两个新的、看起来有些混淆的状态码。

## 303 See Other

此状态码也是重定向。但是它不管之前的请求方法是什么，都强制要求转换请求方法为GET

## 307 Temporary Redirect

此状态码也是重定向响应。但是它和303不同，它要保持新的请求方法和之前发起请求的请求方法一致。就是说，如果之前是 GET 方法，那么这次重定向也需继续使用GET 方法；之前使用POST方法，这次依然需要保持使用POST 方法。

## 304 Not Modified

此状态码其实和重定向无关。但是总不至于单独为它一个而增加一个分类，所以就放到了300系列内。当用户代理发起GET请求并设置了修改时间的前条件，而服务器发现被请求的资源并没有在给出的时间后被修改，就会返回这个状态码。这个状态码的存在是为了性能上的考量。不必传递用户代理有的、服务器也没有修改的资源。案例：

客户端请求：

GET /sample.html HTTP/1.1  
Host: example.com  
If-Modified-Since: Wed, 01 Sep 2004 13:24:52 GMT

服务器响应：

HTTP/1.1 304 Not Modified  
Date: Tue, 27 Dec 2005 05:25:19 GMT

## 400系列

400 系列响应消息都是用来由服务器告诉客户端，收到的请求是它无法处理的。比如最常用的404 Not Found可以用来指示客户端请求的资源在服务器上根本没有。

400系列数量众多，但是大部分都比较简单。所以我只会把 412、417、403 单列出来做特别说明，因为前两个错误相对而言和协议的其他特性耦合比较多，因此显得复杂，并且对于设计良好的Restful app来说也是比较实用的。第三个状态码比较常见，但是标准定义比较抽象，我希望把它可以具体化一点。

### 412 Precondition Failed

客户端发起了条件请求，服务器发现这个请求中的其中一个条件并不成立，那么服务器就会用此错误码作为响应消息的状态码返回给客户端。

请求头上可以使用如下字段对请求做出条件限定：

If-Match  
If-Modified-Since  
If-None-Match  
If-Range  
If-Unmodified-Since

这些请求头被称为前条件。通过它们可以告诉服务器只有条件满足才去完成请求的执行。

举个例子。假设我们GET了一个资源，在客户端由用户修改后提交更新。那么我们肯定希望在更新资源之前首先询问服务器此资源在 GET 后到更新之前是否有改动。如果有了改动，就说明另一个用户已经修改它了，我们当前的更新就不应该执行，否则就可能会导致不一致的事务了。具体的做法就是使用 POST 方法请求去更新资源的状态，并加上 If-Unmodified-Since 限定只有你最近GET资源之后此资源没有被修改的这个条件满足才去执行更新。而服务器可以校验此条件，如果这个条件没有满足，那么返回状态码412 (Precondition Failed) 即可。

### 417 Expectation Failed

客户端在请求头内加入了Expect字段，这个字段要求的期望如果并不能被服务器支持，那么服务器会返回417 状态码给客户端。

案例：

GET /todo/1 HTTP/1.1  
Host: example.org  
Content-Type:json  
Expect: 100-continue  
  
HTTP/1.1 417 Expectation Failed

### 403 Fobidden

服务器禁止提供资源来响应它，尽管客户端的请求是有效的。相对而言，401 Unauthorized响应只要客户端提供授权就可以得到资源；403 fobidden 是即使客户端给了用户授权也是不行的。那么到底是什么原因禁止呢？标准在此处是沉默的。看到这个状态码，作为客户除了离开或者重试外，其实并不知道到底出了什么错，也不知道如何解决。为了希望用户不太困惑，有人为此错误给出了更加详细的子错误码——这是微软的IIS服务器的做法。这里给出了子错误码的局部列表：

403.1 - Execute access forbidden   
403.2 - Read access forbidden   
403.3 - Write access forbidden   
403.4 - SSL required.   
403.5 - SSL 128 required.   
403.6 - IP address rejected   
403.7 - Client certificate required.   
403.8 - Site access denied   
403.9 - Too many users

对程序员来说，这些状态子码，大部分可以一眼看明白它的含义，因此这套错误码可以帮助对403 Fobidden 的本意做出更加清晰的了解。所以尽管它们并不是标准的一部分，也依然是值得去学习和了解的。 对于用户而言，仅仅看到403 fobidden 而不去了解细节或许是更好的选择。

更多的400系列的错误码是比较简单的，其中大部分都是可以望文生义的了解到它的错误场景。比如411 Lengh Required 就是告诉客户端你的请求必须有Content-Length首部。401 Unauthorize 则是告诉客户端当前访问的资源需要认证，请提供用户名和密码过来。这里就不详述了。

## 500系列

这个系列响应消息的状态码，对用户而言表现的更加含糊。看到了这个错可以确认的就是：这不是客户端的错，也不是用户的错。它就是服务器的错。服务器也不想让用户或用户代理知道更多的细节。

### 500 Internal Server Error

就是一个这样模糊的错。语义上就是服务器遇到了一个妨碍它提供服务的错误，就使用此状态码。作为服务器的开发者，应该需要在发出这个错误时，内部记录具体的，可以有助于解决问题的错误消息。

### 503 Service Unavailable

说明服务器现在无法提供服务，但是将来可以。如果服务器知道何时资源可用，应该在响应中包含Retry-After的首部，提示客户端可以重试服务的时间。

这个状态码略显诡异的是，既然服务已经不可用，那么这条消息是谁给出的？

状态码由Web Server给出，指示为当前网页服务的模块被关闭了。

以IIS为例。IIS是有应用池(application pool)的概念，它是一个比Web Server更小的模块,每个网页都由应用池提供具体服务。要是应用池被关闭了，那么，我们就会遇到这个错误。原因可能是：

1. 你的应用崩溃了
2. 或者你的应用常常崩溃，因此IIS决定关闭你的应用池。（常常崩溃的标准是5分钟内5次）

## 100响应

当客户端发送 Expect:100-Continue时, 服务端可以响应 100 Continue 为允许，或者不许可（ (417 Expectation Failed) 。100 Continue 状态码通知客户端可以继续发送请求。

在发送大文件之前，客户端可以首先发出询问，要是在服务器不接受大文件的话，服务器就可以直接拒绝继续。否则，服务器只能从请求头内提取内容大小，当发现不符合条件的时候，实体内容这时候可能已经在传递了。这可是在浪费带宽了。

### 案例

在此案例中，客户端试图 POST 一个 MP3 视频到服务器，这个文件大小为101MB。客户端并没有在请求主体内发送这个文件，而是添加一个 Expect: 100-continue 的请求首部字段。如果可以接受这样大小的文件，服务器就返回100 Continue ,否则返回417 状态码。

### 编写此服务器代码(app.js)

var http = require("http");  
function onRequest(request, response) {  
response.end("hello world\n")  
}  
http.createServer(onRequest).listen(8181);

### ，并使用Node来执行它

$node app.js

### 发起请求

使用nc 做客户端，直接在console内贴入请求消息文本，并回车两次就可以发出请求到服务器

$ nc localhost 8181  
POST /content/videos HTTP/1.1  
Host: media.example.org  
Content-Type: video/mp4  
Content-Length: 105910000  
Authorization: Basic bWFkZTp5b3VfbG9vaw==  
Expect: 100-continue  
  
HTTP/1.1 100 Continue  
  
HTTP/1.1 200 OK  
Date: Mon, 23 Nov 2015 06:27:53 GMT  
Connection: keep-alive  
Transfer-Encoding: chunked  
  
c  
hello world  
  
0

Node.js的http 服务器默认接受任何大小的文件。因此，当它发现 Expect: 100-continue时，会返回 HTTP/1.1 100 Continue。

希望覆盖默认行为的方法，根据条件（比较具体文件大小）决定是否接受的话，可以参考 Node.js 手册 https://nodejs.org/api/http.html#http\_event\_continue

## 101响应

HTTP 协议提供一个机制，允许在已经建立的连接上把HTTP协议切换到一个新的、不兼容的协议上。

客户端可以发起这个协议切换请求，而服务器可以选择拒绝并关闭连接，或者选择接受。如果服务器选择了接受，接下来就可以在此连接上传递新的协议内容。这样做的好处在于不必重建连接即可做协议升级或者调整为新的协议。服务器可以发送101型响应消息给客户端表示接受协议切换。比如本来是HTTP/1.1协议可以经过101 Switch Protocols就改变为h2c 、WebSocket、TLS。握手完成后，传递的协议就此改变。

案例：如何利用101 Switch Protocols 把HTTP协议切换为WebSocket？

客户端通过http协议的 GET 方法的首部字段，向服务器发起请求：

GET ws://echo.websocket.org/?encoding=text HTTP/1.1  
Origin: http://websocket.org  
Cookie: \_\_utma=99as  
Connection: Upgrade  
Host: echo.websocket.org  
Sec-WebSocket-Key: uRovscZjNol/umbTt5uKmw==  
Upgrade: websocket  
Sec-WebSocket-Version: 13

特别留意的是首部的这两行：

Connection: Upgrade  
Upgrade: websocket

这两个首部字段，就是指明客户端向服务器发起请求，希望把连接升级到websocket。

如果服务器端理解这个请求 ，就会返回一个响应：

HTTP/1.1 101 WebSocket Protocol Handshake  
Date: Fri, 10 Feb 2012 17:38:18 GMT  
Connection: Upgrade  
Upgrade: WebSocket

在响应首行，只是状态码为101，就是协议切换被认可。之后再这个连接上就可以传递websocket协议了。

案例： ADDONE 协议

为了演示升级的过程，我们可以自己实现一个叫做 ADDONE的新协议。这个协议希望客户端发送一个整数过来，然后把这个数字加1后返回给客户端。在正常的http 连接内发送单一的一个整数并不符合HTTP协议的请求包规定，因此会被识别为无效数据而报错，或者被HTTP协议置之不理。但是切换协议后，同样的连接来的数据将不再被HTTP 代码解析，因此不会被HTTP协议识别为非法。ADDONE协议并不具备实用性，但是可以演示HTTP 的升级过程。

实际的代码和测试用例，都在本书附属代码的code/addone.js内。这里仅仅提到其中实现的一些要点：

具体而言，Node 在HTTP 模块会有一个upgrade事件：

function (request, socket, head) { }

它在收到客户端升级请求后发射。如果没有应用代码侦听这个事件，node会关闭此连接。

如果在此事件内确认可以升级，就把 101 Switch Protocols响应发给客户端。然后，HTTP 实现将不再侦听socket的data事件，而升级后的协议实现应该监听data事件并根据data内容做出响应。想要怎么解析请求和发送响应就是新代码的工作了。

## 消息体

无论是请求消息还是响应消息，都有一个可选的消息主体(message-body)。如果客户端在提交表单，那么请求消息主体内就可以放置表单的数据；如果客户端请求下载一个gif，那么响应消息主体内就可以放置gif文件的二进制字节集合。所以，消息主体内可以放置任何内容。它的定义也是如此：

message-body = \*OCTET

OCTET 就是字节。而 \*OCTET则标示为多个字节。 仅仅看\*OCTET是无法知道其中到底是什么内容。这就需要在消息头字段内用一组头字段来标示它。比如Content-Type就会指示内容的类型。下图列出可作为消息主体限定的首部字段清单:

entity-header = Content-Encoding   
| Content-Language   
| Content-Length   
| Content-Location   
| Content-MD5   
| Content-Range   
| Content-Type   
| Allow   
| Expires   
| Last-Modified   
| extension-header  
extension-header = message-header#response(OK)

我们看看每个字段的含义：

* Content-Type 实体中所承载对象的类型。
* Content-Length 所传送实体主体的长度或大小。
* Content-Language 与所传送对象最相配的人类语言。
* Content-Encoding 对象数据所做的压缩格式。
* Content-Location 一个备用位置，请求时可通过它获得对象。
* Content-Range 说明它是整体的哪个部分。
* Content-MD5 实体主体内容的校验和。
* Last-Modified 所传输内容在服务器上创建或最后修改的日期时间。
* Expires 实体数据将要失效的日期时间。
* Allow 该资源所允许的各种请求方法，例如，GET 和 HEAD。
* ETag 这份文档的唯一验证码。

消息主体可以放置静态文件内容、动态生成内容、还可以放置压缩后的动态和静态内容。整个内容可以一次传输完毕，或者分成多块传输。如果有必要，在内容后，还可以放置拖挂——一些只有内容传递完毕才能够知道的首部字段值。

我们重新查看下 HTTP引入一章已经提到的响应消息案例：

HTTP/1.1 200 OK  
Date: Mon, 27 Jul 2009 12:28:53 GMT  
Server: express  
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT  
Content-Length: 88  
Content-Type: text/html  
Connection: Closed  
  
<html>  
 <body>  
  
 <h1>Hello, World!</h1>  
  
 </body>  
</html>

它正是在主体内放置静态Html文件，然后通过Content-Type指明承载内容类型、通过Content-Length 指明内容长度的。

### 内容类型

Content-Type可以是丰富多彩的静态文件，也可以是一些在文件系统内看不到的但是非常实用的格式，比如表达表单数据的multipart/form-data。此格式可以在 POST请求 一节内看到案例。再比如 multipart/x-byterangs ,用来传递文件局部。此格式可以在GET请求 一节内查看案例。

### 内容编码

消息主体可以压缩后再传递从而节省网络传递流量。Content-Encoding就是内容的压缩编码格式。目前支持的压缩方式有：

* gzip ： GNU zip 编码
* compress ： Unix 的文件压缩程序
* deflate ： zlib 的格式压缩
* identity： 没有进行压缩。

### 传输编码 Transfer-Encoding

传输编码可以把消息主体分为若干块大小已知的块来传输。Transfer-Encoding 字段目前的取值只能是chunked，表示分块传输。有了分块传输，可以边生成边传输给客户端，从而提升良好的客户体验。

假设我们要分块传递一个 hello world 的字符串，先传5个字节再传后面的7个字节。那么响应消息如下：

HTTP/1.1 200 OK  
Transfer-Encoding: chunked  
Server: express  
  
5  
hello  
7  
 world  
0

分块主体结构比较简单，首选发送一个数字（16进制）指明本块大小，随后回车标示本块开始。接下来第二个块、以及更多的块也是以块大小的数字开始，随后回车标示本块开始。如此等等。直到块结束就跟着一个数字0。整个内容传递完毕。

### 拖挂

可以在内容传递完毕后，接着加入一些首部值作为拖挂。之所以这些首部值不放到真正的首部区域，是因为这些首部值只有传递内容都完成了才知道。比如，我希望为分块传输添加一个内容校验首部的话，那么此值必须全部内容都传递了才能计算完成。

如果要放置拖挂，需要首先在首部使用Trailer头字段声明拖挂字段名。在传递内容完毕后，填写字段和字段值，以“：”分隔。依然以hello world 字符串为例：

HTTP/1.1 200 OK  
Transfer-Encoding: chunked  
Server: express  
Trailer: checksum  
  
5  
hello  
7  
 world  
0  
checksum:5eb63bbbe01eeed093cb22bb8f5acdc3

# 连接

HTTP是一个应用层协议，基于TCP/IP。HTTP的连接其实就是TCP连接，从头到尾都是。只不过在其中传递的数据包不是任意的二进制，而是HTTP规定好的数据包。数据的发送也不是任意的，而是一个请求一个响应（请求和响应一对构成一个事务）。

我们关心连接的使用方式，是因为它事关多事务情况下的性能。而在http中，处理多事务的连接有4种方式：

* 并行连接
* Keep-Alive
* 持久连接
* 管线(pipeline)。

我们拿一个简单的文件作为多事务的案例（文件名为1.html）：

<html>  
<body>  
 <h1>image/<h1>  
 <img src="2.png"/>  
 <img src="3.png"/>  
</body>  
</html>

即使如此简单的HTML文件，完全获取资源并呈现给客户也需要三个事务。这三个事务分别用来获取1.html、2.png、3.png文件。我们来看不同的事务处理模型应对此组请求有何不同。

## 并行连接

我们首先访问1.HTML，那么客户端会发送1个请求给服务器。首先获取1.html，是“打开连接、发送请求、获取响应、关闭”。

GET /1.HTML

然后获取2.png，依然是“打开连接、发送请求、获取响应、关闭”。

GET /2.png

然后获取3.png，依然是“打开连接、发送请求、获取响应、关闭”。

GET /3.png

这样，为了获取3个文件，我们做了三次重复的连接打开和关闭过程。因为获取了1.html ,解析知道1.png和 2.png文件，之后可以不必分次序，而是同时打开两个连接，分别同时获取2个png资源，然后关闭。

然而，既然三个资源都在同一个服务器，请求也都来自一个客户端，是否可以重用既有的已经打开的连接呢？这样做是允许的，接下来的两种连接模型都是出于这样的优化目的而设计出来的。

## keep-alive连接

keep-alive方法允许客户端和服务器暂时不关闭连接而继续用于接下来的事务。HTTP协议引入了Connection:keep-alive的首部字段，让双方都可以表达保持连接打开的意图。这样，上面的3个事务就变成了：

打开连接、发送请求、获取响应、但是不关闭：

GET /1.HTML   
Connection:keep-alive  
  
---------  
  
HTTP/1.1 200 OK  
Connection:keep-alive  
  
内容...

然后获取2.png，“使用现有连接、发送请求、获取响应、也不关闭”。

GET /2.png  
Connection:keep-alive  
  
---------  
HTTP/1.1 200 OK  
Connection:keep-alive  
  
内容...

然后获取3.png，依然是“使用现有连接、发送请求、获取响应、关闭”。

GET /3.png  
---------  
  
HTTP/1.1 200 OK  
  
内容...

最后一条事务没有发送Connection:keep-alive，因此连接关闭。

使用keep-alive之后，效果就是后面的两个事务可以重用第一个事务建立的连接，从而省下两次打开和关闭连接的开销。

## 持久连接

持久连接是对 keep-alive 的改进。持久连接通过头字段值Connection:close 来通知连接关闭，如果没有发送，就表示保持打开。和 keep-alive 的差别在于默认值的不同，持续连接默认保持，而Keep-Alive默认关闭。这就是两者的不同。再看同样的案例，在持久连接下的不同表现：

打开连接、发送请求、获取响应、但是不关闭：

GET /1.HTML   
---------  
  
HTTP/1.1 200 OK  
  
内容...

然后获取2.png，“使用现有连接、发送请求、获取响应、也不关闭”：

GET /2.png  
  
---------  
HTTP/1.1 200 OK  
  
内容...

然后获取3.png，依然是“使用现有连接、发送请求、获取响应、关闭”：

GET /3.png  
Connection:close  
---------  
  
HTTP/1.1 200 OK  
Connection:close  
内容...

持久连接和Keep-alive在重用连接方面是一致的。但是使用持久连接可以少发送两次Connection头字段。

## 管线

管线是在持久连接的基础上的又一次优化。持久连接内的事务还是逐个方式的。就是说，客户端发起一个请求，然后等待响应，响应收完了再发新的请求。而管线的做法是不同的，在这个模型下，客户端可以一次发出全部请求，然后按照发出的次序，逐个的收对应的响应。依然看案例：

首先获取1.html，打开连接、发送请求、获取响应、但是不关闭：

GET /1.HTML   
---------  
  
HTTP/1.1 200 OK  
  
内容...

然后解析完成1.html，浏览器发现要完整呈现还需要两个资源：2.png和3.png。这时就和前面的事务模型有体现出差别了：客户端会同时发出两个GET请求，而服务器会按照请求的次序，发送两个响应回来。

GET /2.png  
  
GET /3.png  
Connection:close  
---------  
HTTP/1.1 200 OK  
  
内容...  
  
HTTP/1.1 200 OK  
Connection:close  
  
内容...

这就是管线模式和持久连接的不同。在高时延网络条件下，这样做可以降低网络时间。

嗯，这就是4种处理多事务的连接模型的差别。

## 实验：管线连接能力的验证

使用nc和echo的命令组合，可以同时发送两个资源请求到Node Http服务器，如果服务器返回我们请求的两个资源，就说明服务器是支持管线(pipeline)能力的。

cd code   
node pipeline.js

然后执行命令：

$(echo -en "GET /1 HTTP/1.1\n\nGET /2 HTTP/1.1\n\n"; sleep 10) | nc localhost 3000

# 客户识别

客户识别一点也不神奇，不但HTTP要用，日常生活也常常遇到。那我自己来说吧，我喜欢到一家老字号吃饭，那里的店家就得用客户识别。我去那个店家，点上韩包子、钟水饺，交钱，然后店家会给我一个牌，上有一个数字，让我放到桌上（最近一次给我的是25号）。有了这个的标识手段，店里的几个服务员就知道把我的（25号）小吃给我，别人的（比如我后面的26号）不能给我。

在软件开发过程中，即使只是实现购物车功能，服务器也需要区别客户的。HTTP协议以Cookie技术来应对此需求。

Cookie就是用于标识的一个字符串。它会对每个首次来访的客户发一个不同的字符串，客户端会存下此字符串，以便在接下来的同一会话访问过程中提交此字符串给服务器来亮明身份。这样服务器就知道此次访问时之前的是哪个客户端了。

我们还可以再具体点，假设两个客户端来访，它们分别是 A 和 B 。

A首次访问，服务器响应：

HTTP/1.1 200 OK  
Set-Cookie:id=25  
  
这是你的首访

B首次访问，服务器响应：

HTTP/1.1 200 OK  
Set-Cookie:id=26  
  
这是你的首访

A再次访问，发送之前的Cookie给服务器：

GET /A HTTP/1.1  
Cookie: id=25

服务器解析Cookie头字段，知道这个客户是A，可以做出个性响应：

HTTP/1.1 200 OK  
  
欢迎你第二次访问，你的号牌为25

B再次访问，带上Cookie：

GET /A HTTP/1.1  
Cookie: id=26

服务器响应：

HTTP/1.1 200 OK  
Set-Cookie:id=26  
  
欢迎你第二次访问，你的号牌为26

有了这个客户端识别ID，我们就可以完成客户识别的目标了。

### Cookie 格式定义

Cookie并不是只能存储ID，你可以使用键值对，像这样：

Cookie: name="1000copy"; charactor="strong";height="175cm"

还可以指定特殊的键值对，以便指示Cookie的使用范围（DOMAIN）、失效日期（Expires）、安全（SECURE）:

Cookie: name="1000copy";Expires=Wed, 09 Jun 2021 10:18:14 GMT

它的格式设计是面向通用目的的:

Set-Cookie:  
 name=value  
 [;EXPIRES=dateValue]  
 [;DOMAIN=domainName]  
 [;PATH=pathName]  
 [;SECURE]

此 Cookie 格式定义是网景公司首次提出和采用的，其他的浏览器也支持此格式。之后，Cookie 定义格式也有经过RFC的规范过程，并由此基础上，提出了Set-Cookie2 和 Cookies 的格式。但是这组格式定义显得稍微复杂，且用户代理实现不一，并没有广泛且完整的被业界采用。对它们有兴趣的读者，可以参看rfc6265。