



Brief Introduction to HTTP - 3h+ session

Xiaoxing Ye, Sun Never Sets Studio

Table of Content

- 互联网的基础
 - TCP/IP; TCP; UDP; DNS; URI
- HTTP 协议到底是什么？
 - 请求 & 响应 -
 - HTTP Status Code
 - HTTP 首部信息
- HTTPS (Hypertext Transfer Protocol Secure)
 - SSL / TLS?
- 无状态的 HTTP 协议 - Cookie 和 会话管理
- WebSocket、HTTP/2、HTTP/3



互联网的基础

- 你知道当我们在浏览器的地址栏中输入 URL 时, Web 页面是如何呈现的吗?
- 和我们说话一样, 你对服务器吼一声“我要打开知乎”, 就有人去知乎服务器拿页面给你看?
- 还是要变成 0101 这样的二进制代码?

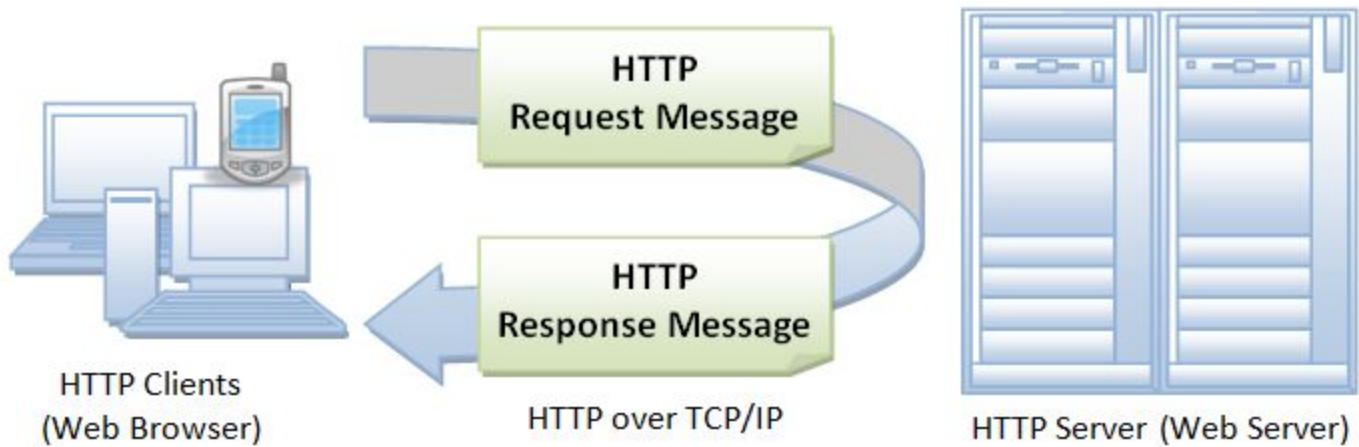


互联网的基础

页面当然不能凭空显示出来。根据用户所指定的 URL，浏览器从服务器端获取文件资源等信息，从而显示出页面。

像这种通过发送请求获取服务器资源的 Web 浏览器等，都可称为客户端 (Client)。

互联网使用一种名为 HTTP (HyperText Transfer Protocol, 超文本传输协议) 的协议作为规范，完成从客户端到服务器端等一系列运作流程。而协议是指规则的约定。可以说，Web 是建立在 HTTP 协议上通信的。





HTTP 为何而诞生？

1989 年 3 月 12 日，欧洲粒子物理研究所(CERN)的计算机科学家蒂姆·伯纳斯·李在其一份提案《Information Management: A Proposal》中提出了一个构想：创建一个以超文本系统为基础的项目，允许在不同计算机之间分享信息，其目的是方便研究人员分享及更新信息。这个构想最终成了WWW(**W**orld **W**ide **W**eb)万维网的基础，彻底改变了人类社会的沟通交流方式。



HTTP 为何而诞生？

不过这个提案在当时并没有引起人们的兴趣。

1993 年 4 月 30 日，欧洲粒子物理研究所(CERN)将万维网软件开源，发布了一个开放式许可证，使得万维网得到最大化的传播。但直到 20 世纪 90 年代中期吉姆·克拉克和马克·安德森在 Netscape(网景通信公司)推广商业网页浏览这一概念后，万维网的应用才开始真正爆发。



HTTP 的历史

HTTP 的最早版本诞生在 1991 年，这个最早版本和现在比起来极其简单，没有 HTTP 头，没有状态码，甚至版本号也没有，后来它的版本号才被定为 0.9 来和其他版本的 HTTP 区分。HTTP/0.9 只支持一种方法——Get，请求只有一行。响应也是非常简单的，只包含 HTML 文档本身。

当 TCP 建立连接之后，服务器向客户端返回 HTML 格式的字符串。发送完毕后，就关闭 TCP 连接。由于没有状态码和错误代码，如果服务器处理的时候发生错误，只会传回一个特殊的包含问题描述信息的 HTML 文件。这就是最早的 HTTP/0.9 版本。



HTTP 的历史

1996 年, HTTP/1.0 版本发布, 大大丰富了 HTTP 的传输内容, 除了文字, 还可以发送图片、视频等, 这为互联网的发展奠定了基础。相比 HTTP/0.9, HTTP/1.0 主要有如下特性:

- 请求与响应支持 HTTP 头, 增加了状态码, 响应对象的一开始是一个响应状态行
- 协议版本信息需要随着请求一起发送, 支持 HEAD, POST 方法
- 支持传输 HTML 文件以外其他类型的内容



HTTP 的历史

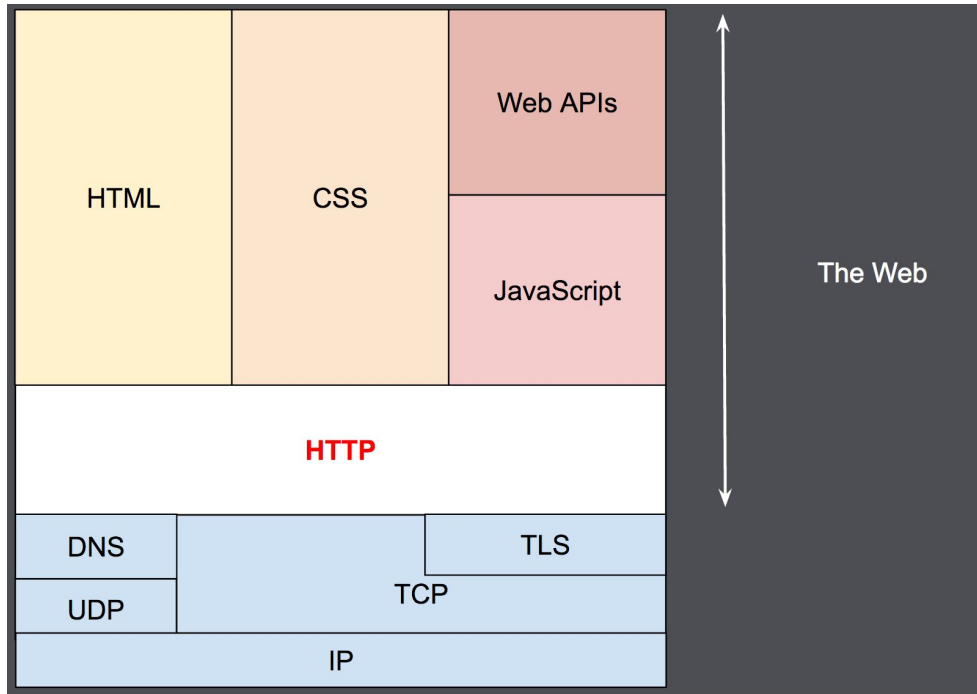
在 HTTP/1.0 发布几个月后, HTTP/1.1 就发布了。HTTP/1.1 更多的是作为对 HTTP/1.0 的完善, 在 HTTP1.1 中, 主要具有如下改进:

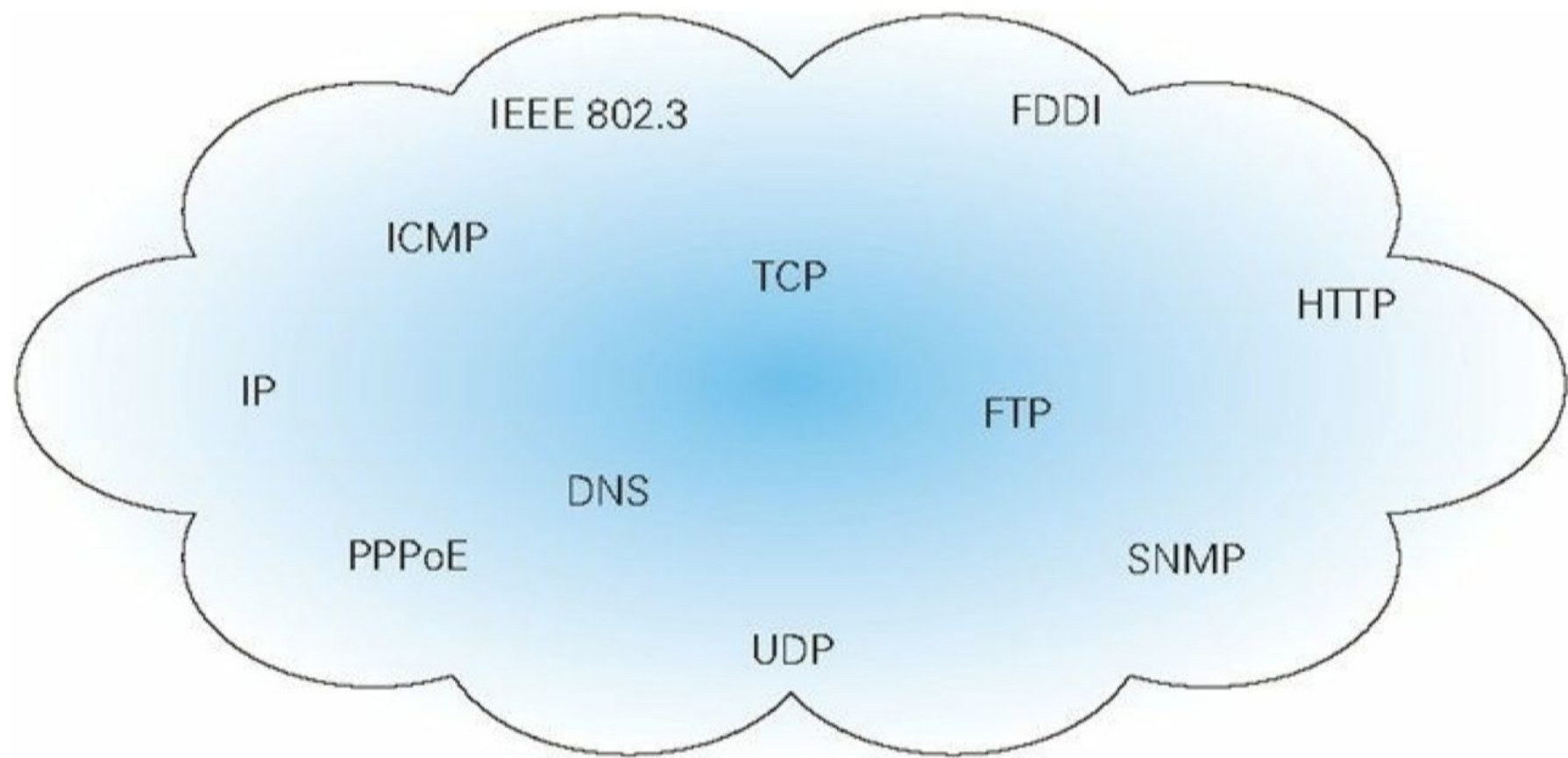
- 可以复用连接
- 引入内容协商机制, 包括语言, 编码, 类型等, 并允许客户端和服务端之间约定以最合适的内容进行交换
-
-

TCP/IP 协议族

为了理解 HTTP, 我们有必要事先了解一下 TCP/IP 协议族。

通常使用的网络(包括互联网)是在 TCP/IP 协议族的基础上运作的。而 HTTP 属于它内部的一个子集。







TCP/IP

计算机与网络设备要相互通信，双方就必须基于相同的方法。

比如，如何探测到通信目标、由哪一边先发起通信、使用哪种语言进行通信、怎样结束通信等规则都需要事先确定。

不同的硬件、操作系统之间的通信，所有的这一切都需要一种规则。而我们就把这种规则称为协议(Protocol)。



TCP/IP

协议中存在各式各样的内容。从电缆的规格到 IP 地址的选定方法、寻找异地用户的方法、双方建立通信的顺序，以及网页显示需要处理的步骤，等等。

像这样把与互联网相关联的协议集合起来总称为 TCP/IP。

也有说法认为，TCP/IP 是指 TCP 和 IP 这两种协议。

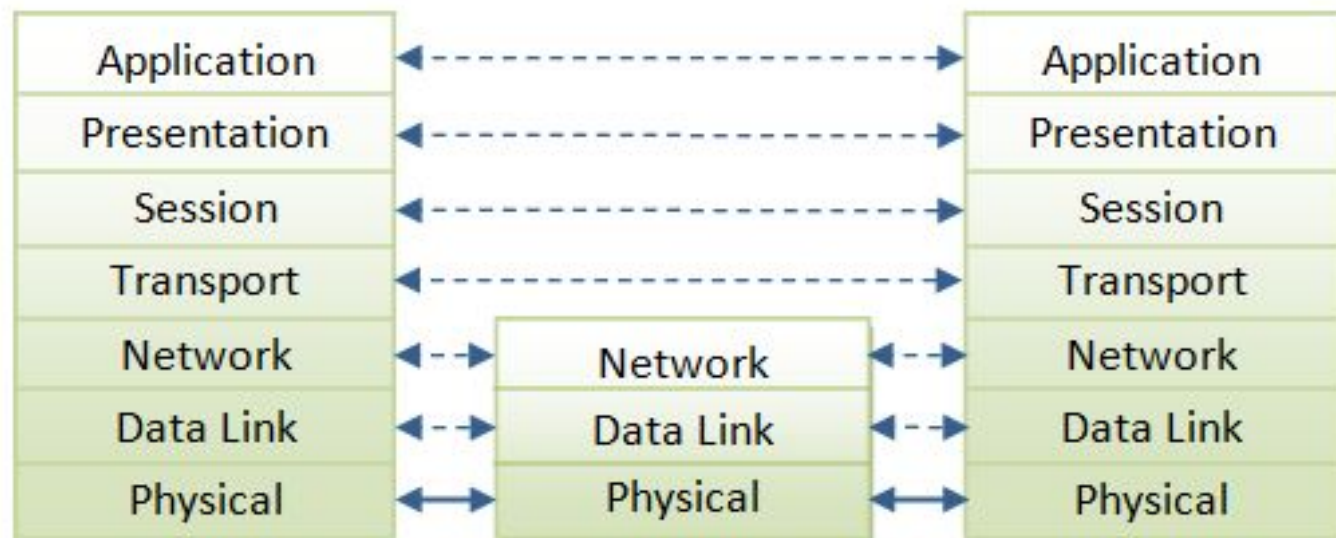
还有一种说法认为，TCP/IP 是在 IP 协议的通信过程中，使用到的协议族的统称。



参考模型

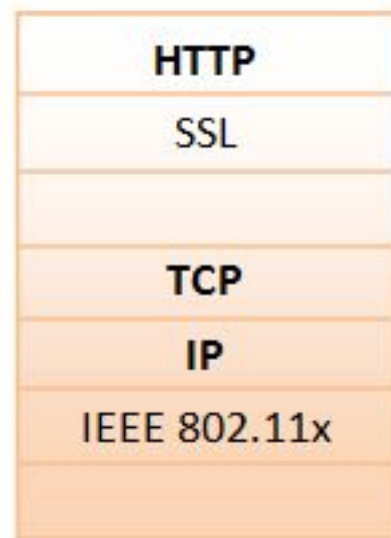
ISO/OSI 参考模型和 TCP/IP 参考模型都采用了层次结构的概念；

ISO/OSI 参考模型是理论上的模型；而 TCP/IP 参考模型已经成为“实际上的国际标准”。



Immediate Nodes (routers)

ISO OSI 7-layer network



HTTP over TCP/IP



TCP/IP 分层

TCP/IP 协议族按层次分别分为以下 4 层:应用层、传输层、网络层和数据链路层。



TCP/IP 分层 - 应用层

应用层决定了向用户提供应用服务时通信的活动。

TCP/IP 协议族内包括了各类通用的应用服务。比如, FTP (File Transfer Protocol, 文件传输协议) 和 DNS (Domain Name System, 域名系统) 服务就是其中两类。

HTTP 协议也处于该层。



TCP/IP 分层 - 传输层

传输层对上层应用层, 提供处于网络连接中的两台计算机之间的数据传输。

在传输层有两个性质不同的协议: TCP (Transmission Control Protocol, 传输控制协议) 和 UDP (User Data Protocol, 用户数据报协议)。

HTTP 0.9 ~ 2 都是基于 TCP 的应用层协议; 而到了 HTTP 3, 则变成了 UDP。
为什么?



Side Note: TCP v.s. UDP

TCP 提供**可靠**的通信传输, 而 UDP 则常被用于让广播和细节控制交给应用的通信传输。TCP 与 UDP 基本区别:

1. 基于连接与无连接
2. TCP 要求系统资源较多, UDP 较少
3. UDP 程序结构较简单
4. 流模式(TCP)与数据报模式(UDP)
5. TCP 保证数据正确性, UDP 可能丢包
6. TCP 保证数据顺序, UDP 不保证



TCP/IP 分层 - 网络层(又名网络互连层)

网络层用来处理在网络上流动的数据包。数据包是网络传输的最小数据单位。该层规定了通过怎样的路径(所谓的传输路线)到达对方计算机, 并把数据包传送给对方。

与对方计算机之间通过多台计算机或网络设备进行传输时, 网络层所起的作用就是在众多的选项内选择一条传输路线。



TCP/IP 分层 - 链路层

链路层(又名数据链路层, 网络接口层)

用来处理连接网络的硬件部分。包括控制操作系统、硬件的设备驱动、NIC (Network Interface Card, 网络适配器, 即网卡), 及光纤等物理可见部分 (还包括连接器等一切传输媒介)。硬件上的范畴均在链路层的作用范围之内。

客户端

服务器

应用层

HTTP
客户端

HTTP
服务器

传输层

TCP

TCP

网络层

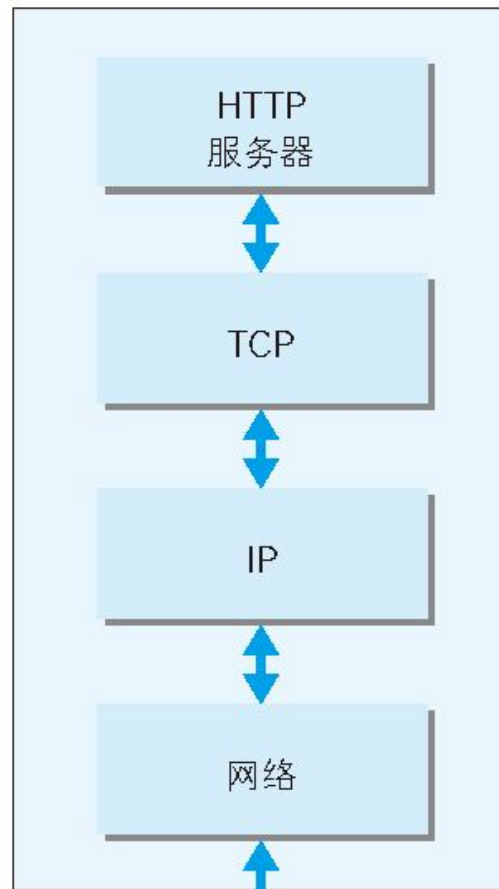
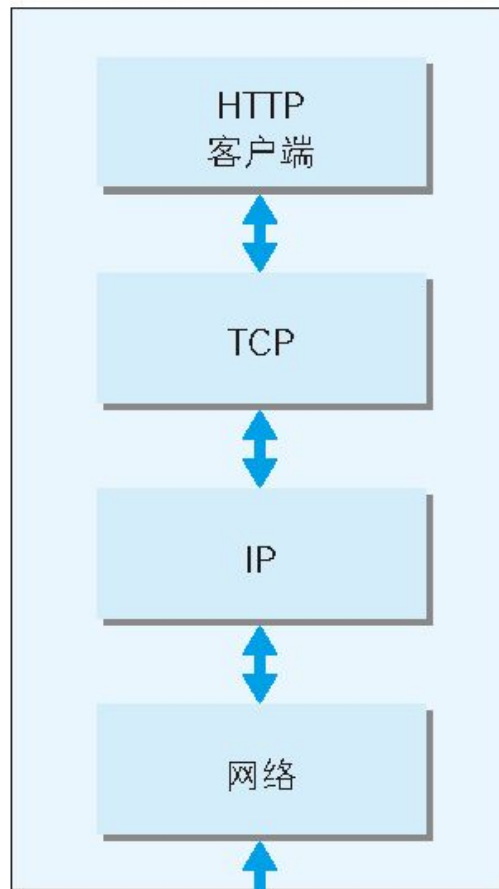
IP

IP

链路层

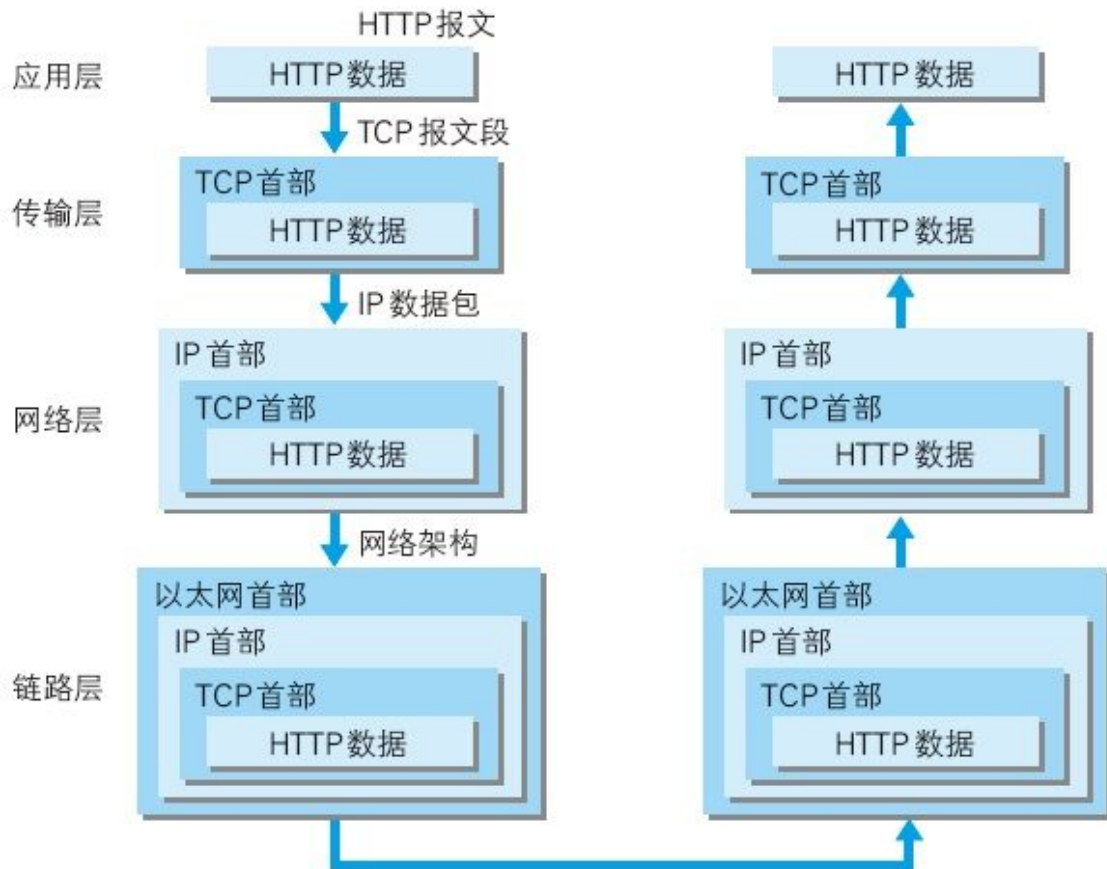
网络

网络



发送端
每通过一层则
增加首部

接收端
每通过一层则
删除首部





确保可靠性的 TCP 协议

还记得刚刚讲过的 TCP v.s. UDP？让我们再回来了解一下 TCP。

按层次分，TCP 位于传输层，提供可靠的字节流服务。

所谓的字节流服务(Byte Stream Service)是指，为了方便传输，将大块数据分割成以报文段(segment)为单位的数据包进行管理。而可靠的传输服务是指，能够把数据准确可靠地传给对方。

一言以蔽之，TCP 协议为了更容易传送大数据才把数据分割，而且 TCP 协议能够确认数据最终是否送达到对方。



TCP

确保数据能到达目标

为了准确无误地将数据送达目标处, TCP 协议采用了三次握手 (three-way handshaking) 策略。用 TCP 协议把数据包送出去后, TCP 不会对传送后的情况置之不理, 它一定会向对方确认是否成功送达。

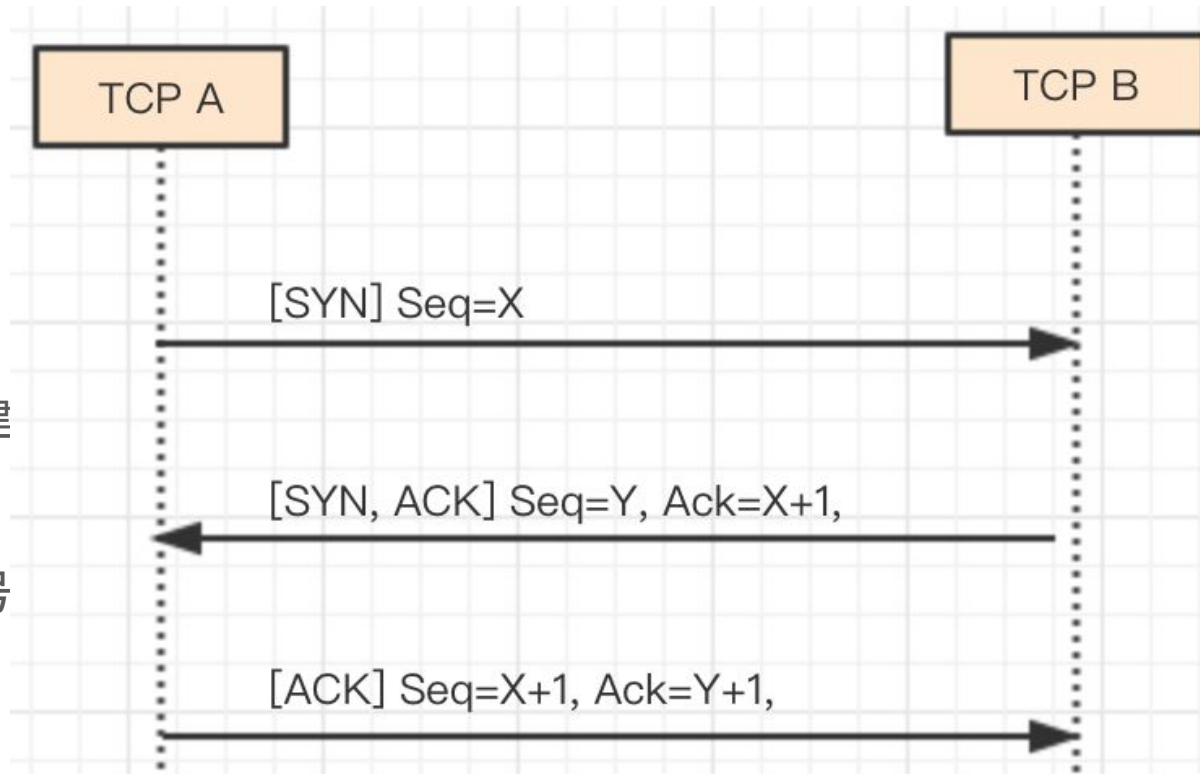
握手过程中使用了 TCP 的标志(flag) —— SYN(synchronize) 和 ACK(acknowledgement)。

TCP - 三次握手

一般情况下，握手流程如右图所示，主要做了两件事情：

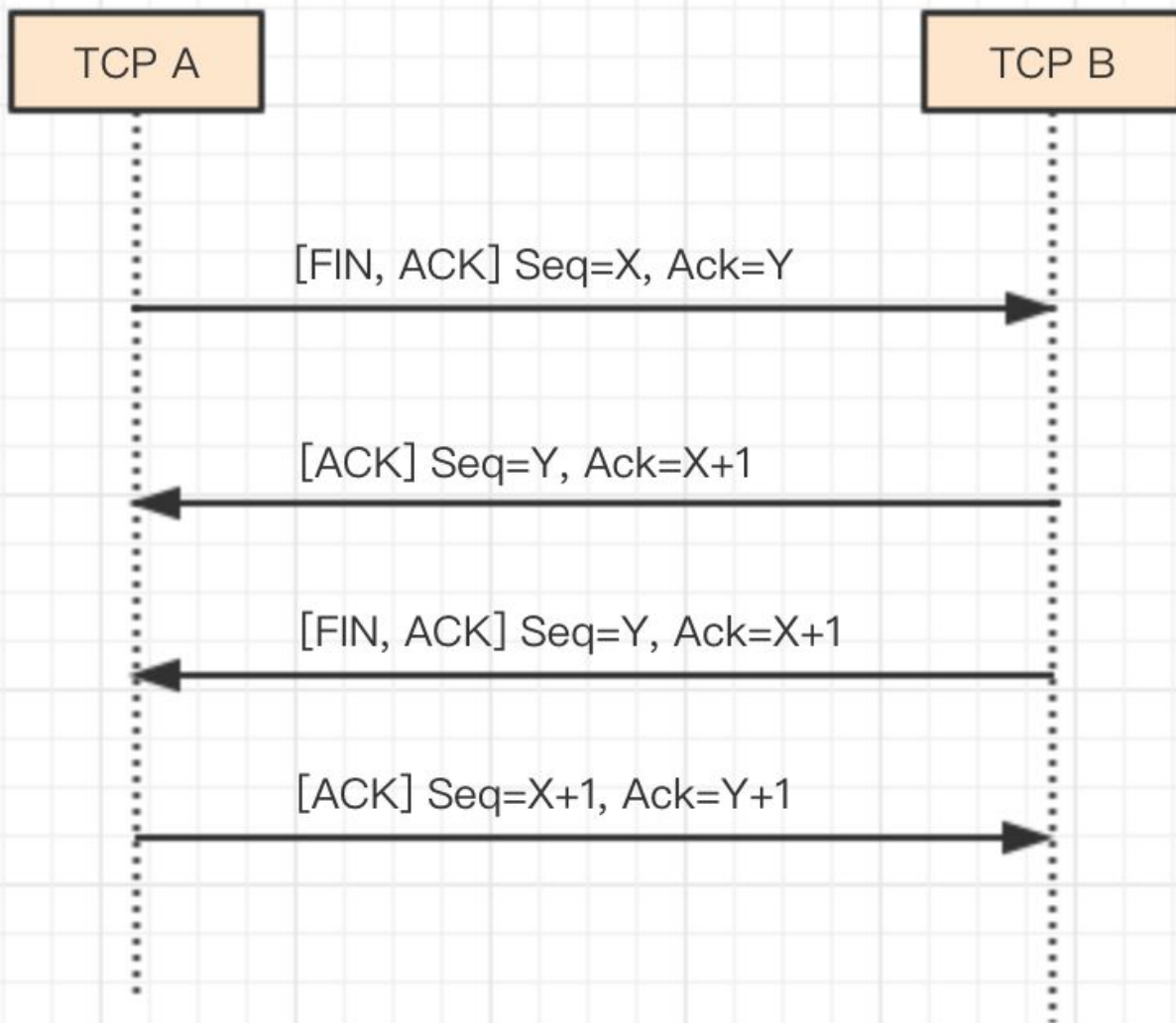
互相确认对方当前可以建立连接

互相交换确认初始序列号 (ISN)



TCP - 四次挥手

一般情况下，TCP断开连接需要4次挥手。假设TCP A 主动断开连接，流程如下。主要就是告知对方，自己准备断开连接了，并且等待对方的确认。





Side Note: 为什么要三次握手？

- 为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误
- 为了解决“网络中存在延迟的重复分组”的问题
- “已失效的连接请求报文段”的产生在这样一种情况下：
- C 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达 S。本来这是一个早已失效的报文段。但 S 收到此失效的连接请求报文段后，就误认为是 C 再次发出的一个新的连接请求，于是就向 C 发出确认报文段，同意建立连接。



Side Note: 为什么要三次握手？

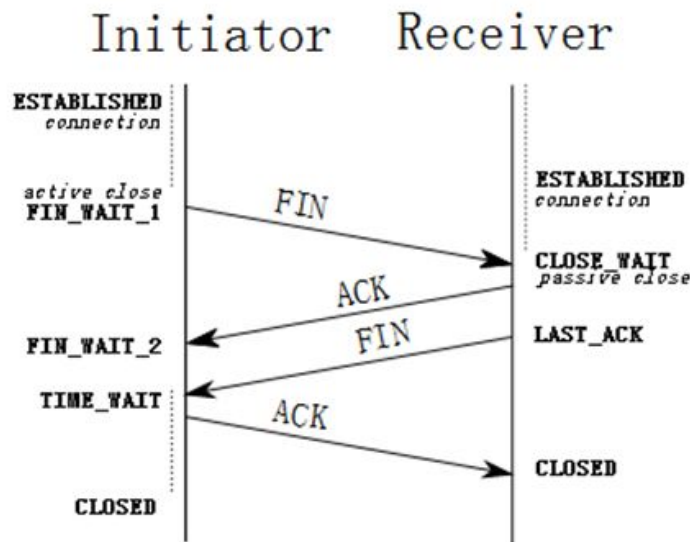
假设不采用“三次握手”，那么只要 S 发出确认，新的连接就建立了。由于现在 C 并没有发出建立连接的请求，因此不会理睬 S 的确认，也不会向 S 发送数据。但 S 却以为新的运输连接已经建立，并一直等待 C 发来数据。这样，S 的很多资源就白白浪费掉了。

采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，C 不会向 S 的确认发出确认。S 由于收不到确认，就知道 C 并没有要求建立连接。

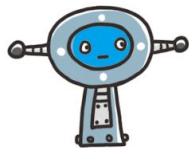
Side Note: 为什么要四次挥手？

在 TCP 连接握手时为何 ACK 是和 SYN 一起发送，而挥手时 ACK 却没有和 FIN 一起发送呢。

原因是 TCP 是全双工模式，接收到 FIN 时意味将没有数据再发来，但是还是可以继续发送数据。



我想浏览
http://hackr.jp/xss/ Web 页面



客户端

告诉我 hackr.jp 的 IP 地址吧



DNS

hackr.jp 对应的 IP 地址是
20X.189.105.112

HTTP 协议的职责

生成针对目标 Web 服务器的 HTTP 请求报文

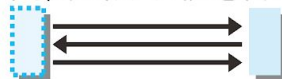
请给我 http://hackr.jp/xss
页面的资源

TCP 协议的职责

为了方便通信，将 HTTP 请求报文分割成报文段



把每个报文段可靠地传给对方



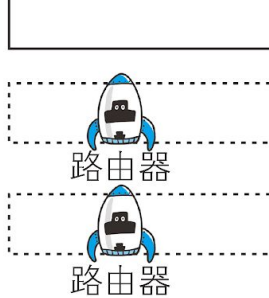
IP 协议的职责

搜索对方的地址，一边中转一边传送



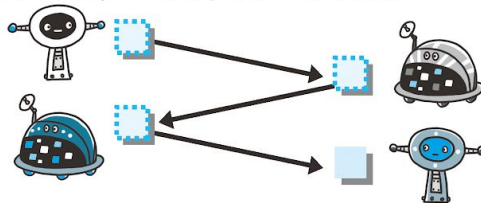
路由器





IP 协议的职责

搜索对方的地址，一边中转一边传送

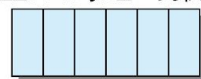


TCP 协议的职责

从对方那里接收到的报文段



重组到达的报文段



按序号以原来的顺序
重组请求报文

HTTP 协议的职责

对 Web 服务器请求的内容的处理

原来是想要这台计算机上的 /xss/ 资源啊

IP 地址
20X.189.105.112



hackr.jp
服务器

请求的处理结果也同样利用 TCP/IP
通信协议向用户进行回传

终于讲完 TCP/IP 的一小部分了.....
喝口水先。



URL & URI

与 URI (Uniform Resource Identifier, 统一资源标识符) 相比, 我们更熟悉 URL (Uniform Resource Locator, 统一资源定位符)。URL 正是使用浏览器等访问页面时需要输入的网页地址。

URI 用字符串标识某一互联网资源, 而 URL 表示资源的地点 (互联网上所处的位置)。可见 URL 是 URI 的子集。



URI

URI 是 Uniform Resource Identifier 的缩写。RFC2396 分别进行了如下定义：

- Uniform: 规定统一的格式可方便处理多种不同类型的资源, 而不用根据上下文环境来识别资源指定的访问方式。另外, 加入新增的协议方案(如 http: 或 ftp:) 也更容易。
- Resource: 资源的定义是“可标识的任何东西”。不仅是文档文件、图像或服务(例如当天的天气预报)等能够区别于其他类型的, 全都可作为资源。另外, 资源不仅可以是单一的, 也可以是多数的集合体。
- Identifier: 表示可标识的对象。也称为标识符。

综上所述, URI 就是由某个协议方案表示的资源的定位标识符。协议方案是指访问资源所使用的协议类型名称。

URI

http://user:pass@www.example.jp:80/dir/index.htm?uid=1#ch1

协议
方案名

登录信息
(认证)

服务器地址

服务器端口号

带层次的文件路径

查询字符串

片段标识符

好了，终于要开始讲 HTTP 了.....



HTTP

超文本传输协议(HTTP)是应用层的一个协议,是万维网生态系统的核心,在 OSI 七层模型中在最上层。

它并不涉及数据包传输,主要规定了客户端和服务端之间的通信格式,默认使用 80 端口(HTTPS 默认使用 443),但实际上你可以使用你喜欢的端口,比如 23333,但是在访问时就需要显式带上端口了。

HTTP 协议采用 B/S 架构,也就是浏览器到服务器的架构,客户端通过浏览器发送 HTTP 请求给服务器,服务器经过解析响应客户端的请求。



RFC says...

“HTTP is an application-level protocol with the lightness and speed necessary for distributed, hypermedia information systems.”



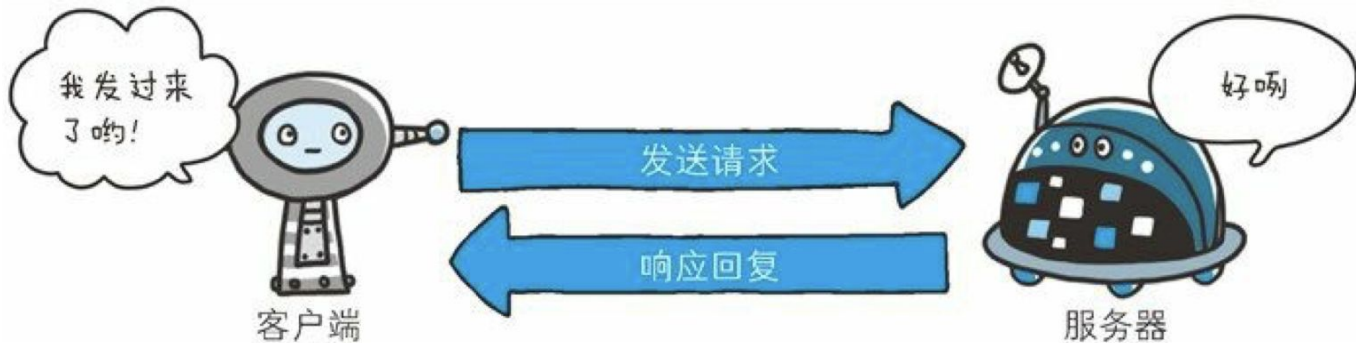
HTTP 1.1

接下来将使用 HTTP/1.1 版本，针对 HTTP 协议结构进行讲解。

之后会有关于 HTTP/2 和 HTTP/3 的介绍。

HTTP - 建立请求

HTTP 协议规定，请求从客户端发出，最后服务器端响应该请求并返回。换句话说，肯定是先从客户端开始建立通信的，服务器端在没有接收到请求之前不会发送响应。





HTTP - 建立请求

从客户端发送给某个 HTTP 服务器端的请求报文中的内容：

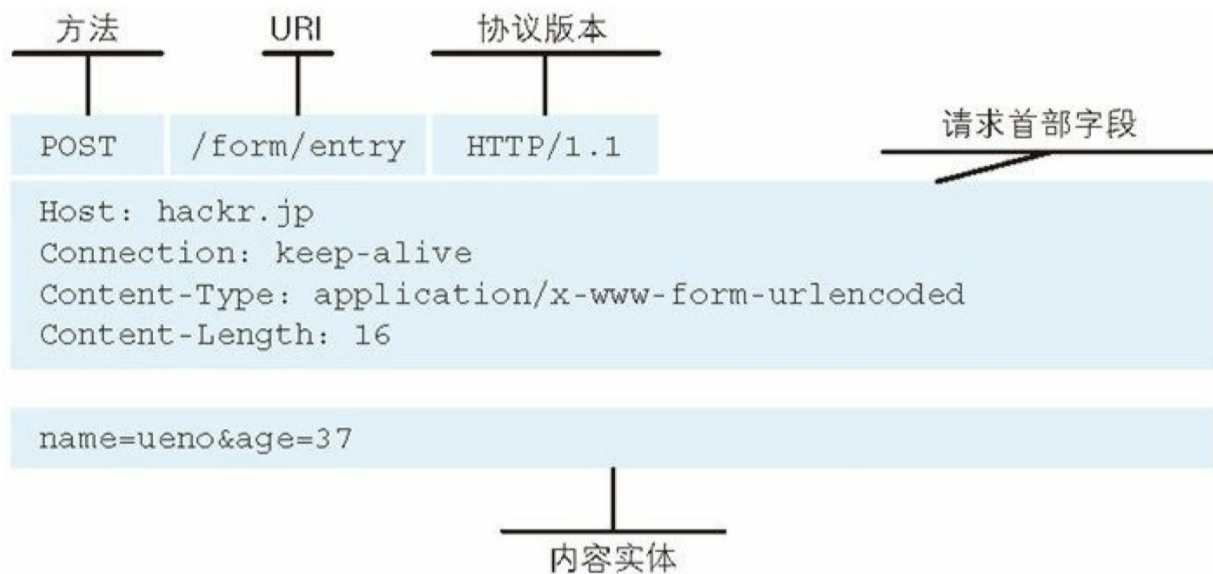
GET /index.htm HTTP/1.1

Host: www.cuhk.edu.cn

起始行开头的 GET 表示请求访问服务器的类型，称为方法 (Method)。随后的字符串 /index.htm 指明了请求访问的资源对象，也叫做请求 URI (Request-URI)。最后的 HTTP/1.1，即 HTTP 的版本号，用来提示客户端使用的 HTTP 协议功能。

HTTP - 建立请求

一个请求报文由以下四个部分组成: 请求行 (Request Line)、消息头部 (Header)、空行、请求正文



HTTP - 返回响应



接收到请求的服务器，会将请求内容的处理结果以响应(Response)的形式返回。

响应报文基本上由协议版本、状态码(表示请求成功或失败的数字代码)、用以解释状态码的原因短语、可选的响应首部字段以及实体主体构成。



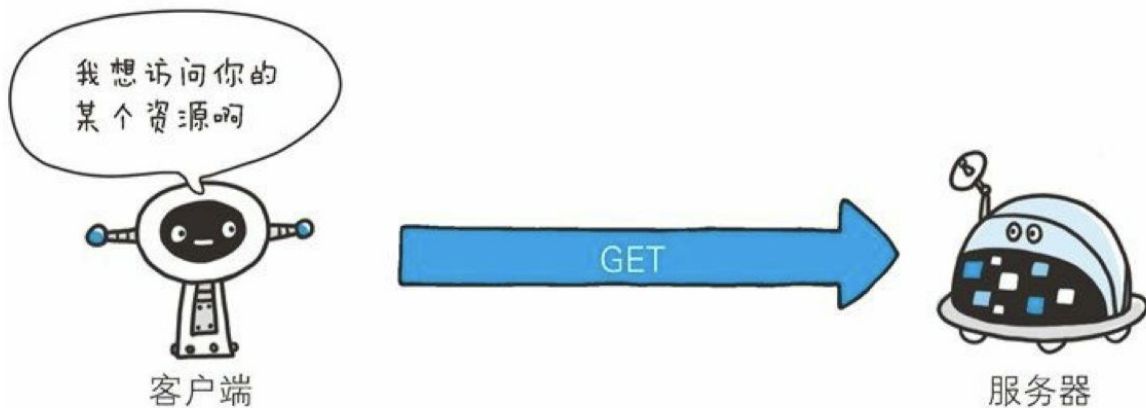
HTTP - 可用的方法(Methods)

下面, 我们介绍 HTTP/1.1 中常用的方法。

- GET : 获取资源
- POST: 传输实体主体
- PUT: 传输文件
- HEAD: 获得报文首部
- DELETE: 删除文件
- OPTIONS: 询问支持的方法
- CONNECT: 要求用隧道协议连接代理

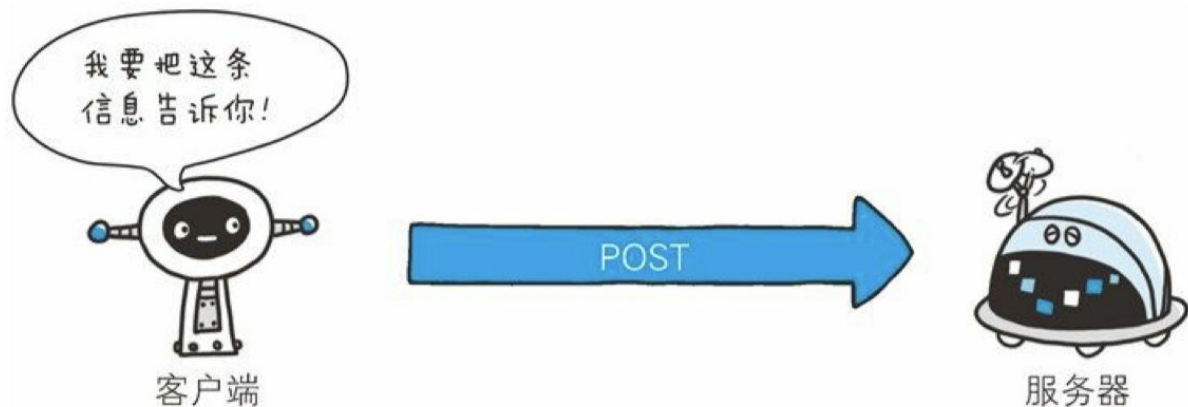
HTTP Methods - GET 获取资源

GET 方法用来请求访问已被 URI 识别的资源。指定的资源经服务器端解析后返回响应内容。也就是说，如果请求的资源是文本，那就保持原样返回；如果请求的是某个脚本/程序的结果，则返回经过执行后的输出结果。



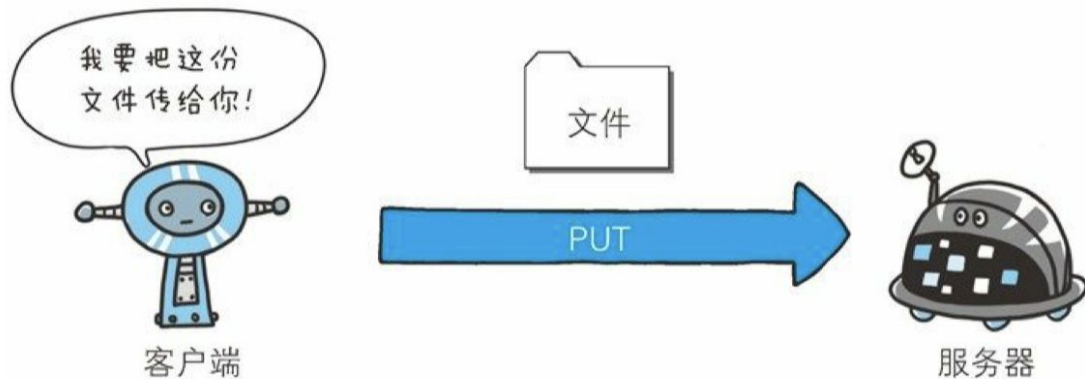
HTTP Methods - POST 传输实体主体

POST 方法用来传输实体的主体。虽然用 GET 方法也可以传输实体的主体，但一般不用 GET 方法进行传输，而是用 POST 方法。虽说 POST 的功能与 GET 很相似，但 POST 的主要目的并不是获取响应的主体内容。



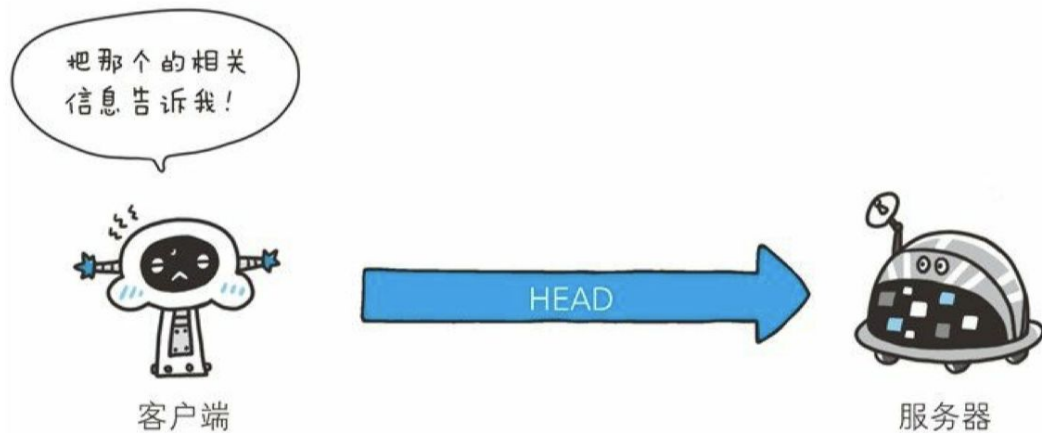
HTTP Methods - PUT 传输文件

PUT 方法用来传输文件。就像 FTP 协议的文件上传一样，要求在请求报文的主体中包含文件内容，然后保存到请求 URI 指定的位置。注意：HTTP/1.1 的 PUT 方法自身不带验证机制，任何人都可以上传文件，存在安全性问题，需要配合 Web 应用程序的验证机制使用。



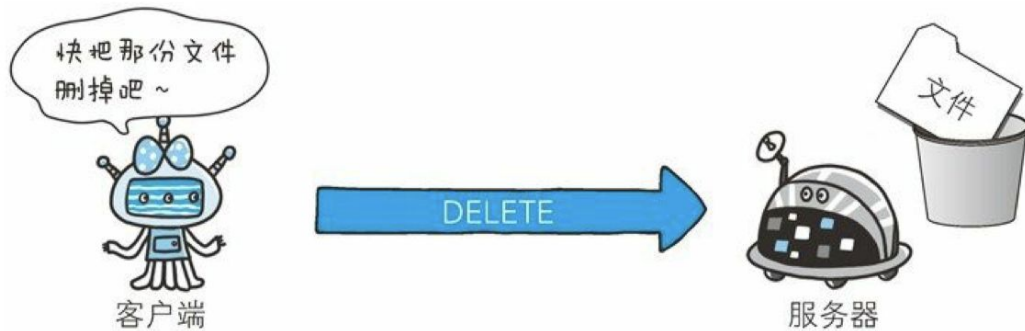
HTTP Methods - HEAD 获得报文首部

HEAD 方法和 GET 方法一样，只是不返回报文主体部分。用于确认 URI 的有效性 & 资源更新的日期时间等。



HTTP Methods - DELETE 删除文件

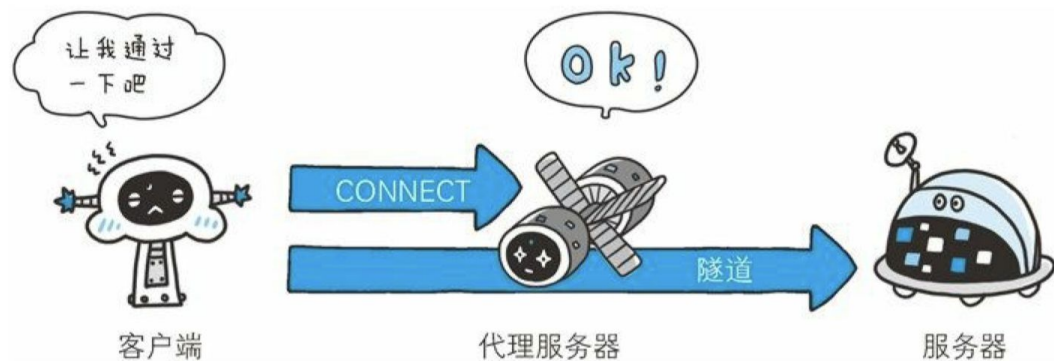
DELETE 方法用来删除文件，是与 PUT 相反的方法。DELETE 方法按 请求 URI 删除指定的资源。但是，HTTP/1.1 的 DELETE 方法本身和 PUT 方法一样不带验证机制，所以一般的 Web 网站也不使用 DELETE 方法。当配合 Web 应用程序的验证机制，或遵守 REST 标准时还是有可能会开放使用的。



HTTP Methods - CONNECT

要求用隧道协议连接代理

CONNECT 方法要求在与代理服务器通信时建立隧道，实现用隧道协议进行 TCP 通信。主要使用 SSL (Secure Sockets Layer, 安全套接层) 和 TLS (Transport Layer Security, 传输层安全) 协议把通信内容加密后经网络隧道传输。





Practice: HTTP Basics

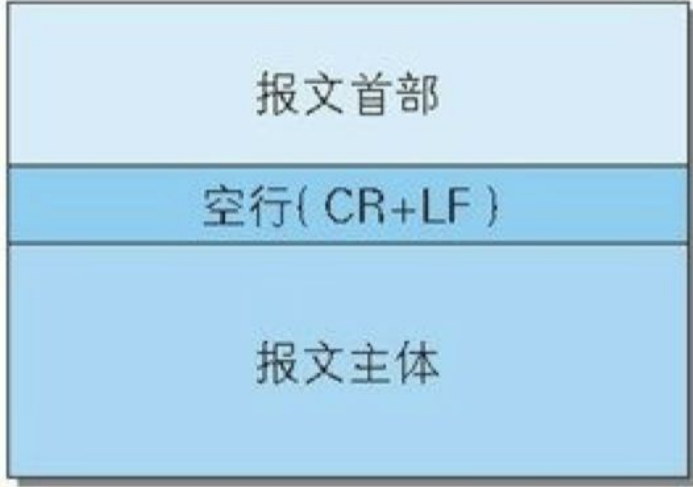
1. 拿出你的电脑, Linux 和 macOS 用户请打开终端, Windows 用户.....看屏幕, 我也不知道(摊手)
2. 试试最简单的 GET 请求?
 - a. `curl -X GET "http://httpbin.org/get" -v`
 - b. 解释:-X 指定方法;-v 使用话痨模式
3. 再试试 POST?
 - a. `curl -d "param1=value1¶m2=value2" -X POST http://httpbin.org/post -v`



HTTP 报文

用于 HTTP 协议交互的信息被称为 HTTP 报文。请求端(客户端)的 HTTP 报文叫做请求报文，响应端(服务器端)的叫做响应报文。HTTP 报文本身是由多行(用 CR+LF 作换行符)数据构成的字符串文本。

HTTP 报文大致可分为报文首部和报文主体两块。两者由最初出现的 空行 (CR+LF) 来划分。报文主体不一定存在(如大多数 GET 请求)。



报文首部

空行(CR+LF)

报文主体

【 报文首部 】

服务器端或客户端需处理的请求或响应的内容及属性

【 CR + LF 】

CR(Carriage Return, 回车符: 16进制 0x0d)和
LF(Line Feed, 换行符: 16进制 0x0a)

【 报文主体 】

应被发送的数据

报文首部

空行(CR + LF)

报文主体

请求行

请求首部字段

通用首部字段

实体首部字段

其他

报文首部

空行(CR + LF)

报文主体

状态行

响应首部字段

通用首部字段

实体首部字段

其他

GET / HTTP/1.1

请求行

Host: hackr.jp

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/20100101 Firefox/13.0.1

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: ja,en-us;q=0.7,en;q=0.3

Accept-Encoding: gzip, deflate

DNT: 1

Connection: keep-alive

Pragma: no-cache

Cache-Control: no-cache

各种首部字段

空行(CR + LF)

HTTP/1.1 200 OK

状态行

Date: Fri, 13 Jul 2012 02:45:26 GMT
Server: Apache
Last-Modified: Fri, 31 Aug 2007 02:02:20 GMT
ETag: "45bae1-16a-46d776ac"
Accept-Ranges: bytes
Content-Length: 362
Connection: close
Content-Type: text/html

各种首部字段

空行(CR + LF)

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>hackr.jp</title>
</head>
<body>

</body>
</html>
```

报文主体



HTTP 报文 - 首部内容

请求报文和响应报文的首部内容由以下数据组成。

请求行: 包含用于请求的方法, 请求 URI 和 HTTP 版本。

状态行: 包含表明响应结果的状态码, 原因短语和 HTTP 版本。

首部字段: 包含表示请求和响应的各种条件和属性的各类首部。一般有 4 种首部, 分别是: 通用首部、请求首部、响应首部和实体首部。



如何在一份报文中发送多个实体？

发送邮件时，我们可以在邮件里写入文字并添加多份附件。这是因为 采用了 MIME (Multipurpose Internet Mail Extensions) 机制，它允许邮件处理文本、图片、视频等多个不同类型的数据。例如，图片等二进制数据以 ASCII 码字符串编码的方式指明，就是利用 MIME 来描述标记数据类型。而在 MIME 扩展中会使用一种称为多部分对象集合 (Multipart) 的方法，来容纳多份不同类型的数据。

相应地，HTTP 协议中也采纳了多部分对象集合，发送的一份报文主体内可含有多类型实体。通常是在图片或文本文件等上传时使用。



如何在一份报文中发送多个实体？

多部分对象集合包含的对象如下。

- multipart/form-data: 在 Web 表单文件上传时使用。
- multipart/byteranges: 状态码 206 (Partial Content, 部分内容) 响应报文包含了多个范围的内容时使用。

multipart/form-data

```
Content-Type: multipart/form-data; boundary=AaB03x
```

```
--AaB03x
```

```
Content-Disposition: form-data; name="field1"
```

```
Joe Blow
```

```
--AaB03x
```

```
Content-Disposition: form-data; name="pics"; filename="file1.txt"
```

```
Content-Type: text/plain
```

```
... (file1.txt的数据) ...
```

```
--AaB03x--
```



如何在一份报文中发送多个实体？

在 HTTP 报文中使用多部分对象集合时，需要在首部字段里加上 Content-type。使用 boundary 字符串来划分多部分对象集合指明的各类实体。在 boundary 字符串指定的各个实体的起始行之前插入“--”标记（例如-AaB03x、--THIS_STRING_SEPARATES），而在多部分对象集合对应的字符串的最后插入“--”标记（例如：--AaB03x--、-THIS_STRING_SEPARATES--）作为结束。



HTTP - 内容协商

同一个网站有可能存在着多份相同内容的页面。比如英语版和中文版的 Web 页面，它们内容上虽相同，但使用的语言却不同。

当浏览器的默认语言为英语或中文，访问相同 URI 的页面时，则会显示对应的英语版或中文版的 Web 页面。这样的机制称为内容协商 (Content Negotiation)。

内容协商机制是指客户端和服务端就响应的资源内容进行交涉，然后提供给客户端最为适合的资源。内容协商会以响应资源的语言、字符集、编码方式等作为判断的基准。



HTTP - 内容协商

包含在请求报文中的某些首部字段(如下)就是判断的基准。

- Accept
- Accept-Charset
- Accept-Encoding
- Accept-Language
- Content-Language



返回结果的 HTTP 状态码

HTTP 状态码负责表示客户端 HTTP 请求的返回结果、标记服务器端的处理是否正常、通知出现的错误等工作。

状态码的职责是当客户端向服务器端发送请求时，描述返回的请求结果。借助状态码，用户可以知道服务器端是正常处理了请求，还是出现了错误。




HTTP Status Code - Intro

状态码如 200 OK, 以 3 位数字和原因短语组成。

数字中的第一位指定了响应类别, 后两位无分类。响应类别有以下 5 种:

1XX	Informational(信息性状态码)	接收的请求正在处理
2XX	Success(成功状态码)	请求正常处理完毕
3XX	Redirection(重定向状态码)	需要进行附加操作以完成请求
4XX	Client Error(客户端错误状态码)	服务器无法处理请求
5XX	Server Error(服务器错误状态码)	服务器处理请求出错



HTTP Status Code - Frequently Used

仅记录在 RFC2616 上的 HTTP 状态码就达 40 种，若再加上 WebDAV (Web-based Distributed Authoring and Versioning) (RFC4918、5842) 和附加 HTTP 状态码 (RFC6585) 等扩展，数量就达 60 余种。

虽然种类繁多，实际上经常使用的大概只有 14 种。接下来，我们就介绍一下这些具有代表性的 14 个状态码。



HTTP Status Code - 2xx Success

- 200 OK: 表示从客户端发来的请求在服务器端被正常处理了。
 - 在响应报文内, 随状态码一起返回的信息会因方法的不同而发生改变。比如, 使用 GET 方法时, 对应请求资源的实体会作为响应返回; 而使用 HEAD 方法时, 只返回首部, 不会返回实体的主体部分。
- 204 No Content: 该状态码代表服务器接收的请求已成功处理, 但在返回的响应报文中不含实体的主体部分。另外, 也不允许返回任何其他实体的主体。比如, 当从浏览器发出请求处理后, 返回 204 响应, 那么浏览器显示的页面不需要发生更新。一般在只需要从客户端往服务器发送信息, 而对客户端不需要发送新信息内容的情况下使用。



HTTP Status Code - 2xx Success

- 206 Partial Content: 该状态码表示客户端进行了范围请求, 而服务器成功执行了这部分的 GET 请求。响应报文中包含由 Content-Range 指定范围的实体内容。



HTTP Status Code - 3xx Redirection

3XX 响应结果表明浏览器需要执行某些特殊的处理以正确处理请求。

- 301 Moved Permanently: 永久性重定向。该状态码表示请求的资源已被分配了新的 URI, 以后应使用资源现在所指的 URI。一般来说, 浏览器会缓存结果, 下次请求同一个 URI 时会直接按新的 URI 发送请求。
- 302 Found: 临时性重定向。该状态码表示请求的资源已被分配了新的 URI, 希望用户(本次)能使用新的 URI 访问。和 301 Moved Permanently 状态码相似, 但 302 状态码代表的资源不是被永久移动, 只是临时性质的。换句话说, 已移动的资源对应的 URI 将来还有可能发生改变。



HTTP Status Code - 3xx Redirection

3XX 响应结果表明浏览器需要执行某些特殊的处理以正确处理请求。

- 303 See Other: 该状态码表示由于请求对应的资源存在着另一个 URI, 应使用 GET 方法定向获取请求的资源。303 状态码和 302 Found 状态码有着相同的功能, 但 303 状态码明确表示客户端应当采用 GET 方法获取资源, 这点与 302 状态码有区别。
- 当 301、302、303 响应状态码返回时, 几乎所有的浏览器都会把 POST 改成 GET, 并删除请求报文内的主体, 之后请求会自动再次发送。301、302 标准是禁止将 POST 方法改变成 GET 方法的, 但实际使用时大家都会这么做。



HTTP Status Code - 3xx Redirection

3XX 响应结果表明浏览器需要执行某些特殊的处理以正确处理请求。

- 304 Not Modified: 该状态码表示客户端发送附带条件的请求时, 服务器端允许请求访问资源, 但未满足条件的情况。304 状态码返回时, 不包含任何响应的主体部分。
 - 304 虽然被划分在 3XX 类别中, 但是和重定向没有关系。
 - 附带条件的请求是指采用 GET 方法的请求报文中包含 If-Match, If-Modified-Since, If-None-Match, If-Range, If-Unmodified-Since 中任一首部。



HTTP Status Code - 3xx Redirection

3XX 响应结果表明浏览器需要执行某些特殊的处理以正确处理请求。

- 307 Temporary Redirect: 临时重定向。该状态码与 302 Found 有着相同的含义。尽管 302 标准禁止 POST 变换成 GET, 但实际使用时大家并不遵守。
- 一般浏览器遇到 307 会遵照标准, 不会从 POST 变成 GET。但是, 对于处理响应时的行为, 每种浏览器有可能出现不同的情况。



HTTP Status Code - 4xx Client Error

~~客户怎么会犯错呢，客户永远是上帝。~~4XX 的响应结果表明客户端是发生错误的原因所在。

- 400 Bad Request: 该状态码表示请求报文中存在语法错误。当错误发生时，需修改请求的内容后再次发送请求。另外，浏览器会像 200 OK 一样对待该状态码。



HTTP Status Code - 4xx Client Error

4XX 的响应结果表明客户端是发生错误的原因所在。

- 401 Unauthorized: 该状态码表示发送的请求需要有通过 HTTP 认证 (BASIC 认证、DIGEST 认证) 的认证信息。另外若之前已进行过 1 次请求, 则表示用户认证失败。
 - 返回含有 401 的响应必须包含一个适用于被请求资源的 WWW-Authenticate 首部用以质询(challenge) 用户信息。当浏览器初次接收到 401 响应, 会弹出认证用的对话框。



HTTP Status Code - 4xx Client Error

4XX 的响应结果表明客户端是发生错误的原因所在。

- 403 Forbidden: 该状态码表明对请求资源的访问被服务器拒绝了。服务器端没有必要给出拒绝的详细理由, 但如果想作说明的话, 可以在实体的主体部分对原因进行描述, 这样就能让用户看到了。
 - 未获得文件系统的访问授权, 访问权限出现某些问题(从未授权的发送源 IP 地址试图访问)等列举的情况都可能是发生 403 的原因。



HTTP Status Code - 4xx Client Error

4XX 的响应结果表明客户端是发生错误的原因所在。

- 404 Not Found: 该状态码表明服务器上无法找到请求的资源。除此之外，也可以在服务器端拒绝请求且不想说明理由时使用。



HTTP Status Code - 5xx Server Error

5XX 的响应结果表明服务器本身发生错误。

- 500 Internal Server Error: 该状态码表明服务器端在执行请求时发生了错误。也有可能是 Web 应用存在的 bug 或某些临时的故障。
- 503 Service Unavailable: 该状态码表明服务器暂时处于超负载或正在进行停机维护, 现在无法处理请求。如果事先得知解除以上状况需要的时间, 最好写入 Retry-After 首部字段再返回给客户端。



Wrong HTTP Status Code?

状态码和状况的不一致

不少返回的状态码响应都是错误的，但是用户可能察觉不到这点。比如 Web 应用程序内部发生错误，状态码依然返回 200 OK，这种情况也经常遇到。

比如腾讯，他们的 API 都不会返回非 200 的状态码，而是在响应主体中告诉你发生了什么.....

Take some rest... And listen to
music. <https://www.youtube.com/watch?v=nSKp2S>





HTTP 首部

HTTP 协议的请求和响应报文中必定包含 HTTP 首部, 只是我们平时在使用 Web 的过程中感受不到它。

首部内容为客户端和服务端分别处理请求和响应提供所需要的信息。对于客户端用户来说, 这些信息中的大部分内容都无须亲自查看。

因 HTTP 版本或扩展规范的变化, 首部字段可支持的字段内容略有不同。



HTTP 首部字段

HTTP 首部字段是构成 HTTP 报文的要素之一。在客户端与服务器之间以 HTTP 协议进行通信的过程中, 无论是请求还是响应都会使用首部字段, 它能起到传递额外重要信息的作用。

使用首部字段是为了给浏览器和服务器提供报文主体大小、所使用的语言、认证信息等内容。



HTTP 首部字段

- HTTP 首部字段是由首部字段名和字段值构成的, 中间用英文半角冒号“:”分隔。
- 格式:首部字段名: 字段值
- 例如, 在 HTTP 首部中以 Content-Type 这个字段来表示报文主体的对象类型:Content-Type: text/html
- 另外, 字段值对应单个 HTTP 首部字段可以有多个值, 如下所示。
- Keep-Alive: timeout=15, max=100



HTTP 首部字段 - Oops, 字段重复了？

当 HTTP 报文首部中出现了两个或两个以上具有相同首部字段名时会怎么样？这种情况在规范内尚未明确，根据浏览器内部处理逻辑的不同，结果可能并不一致。有些浏览器会优先处理第一次出现的首部字段，而有些则会优先处理最后出现的首部字段。



HTTP 首部字段类型

HTTP 首部字段根据实际用途被分为以下 4 种类型：

1. 通用首部字段 (General Header Fields) : 请求报文和响应报文两方都会使用的首部。
2. 请求首部字段 (Request Header Fields) : 从客户端向服务器端发送请求报文时使用的首部。补充了请求的附加内容、客户端信息、响应内容相关优先级等信息。
3. 响应首部字段 (Response Header Fields) : 从服务器端向客户端返回响应报文时使用的首部。补充了响应的附加内容, 也会要求客户端附加额外的内容信息。
4. 实体首部字段 (Entity Header Fields) : 针对请求报文和响应报文的实体部分使用的首部。补充了资源内容更新时间等与实体有关的信息。

HTTP 首部字段 - 通用首部字段

首部字段名	说明
Cache-Control	控制缓存的行为
Connection	逐跳首部、连接的管理
Date	创建报文的日期时间
Pragma	报文指令
Trailer	报文末端的首部一览
Transfer-Encoding	指定报文主体的传输编码方式
Upgrade	升级为其他协议
Via	代理服务器的相关信息
Warning	错误通知

HTTP 首部字段 - 请求首部字段

首部字段名	说明
Accept	用户代理可处理的媒体类型
Accept-Charset	优先的字符集
Accept-Encoding	优先的内容编码
Accept-Language	优先的语言(自然语言)
Authorization	Web认证信息
Expect	期待服务器的特定行为
From	用户的电子邮箱地址
Host	请求资源所在服务器
If-Match	比较实体标记(ETag)

HTTP 首部字段 - 请求首部字段

首部字段名	说明
If-Modified-Since	比较资源的更新时间
If-None-Match	比较实体标记(与 If-Match 相反)
If-Range	资源未更新时发送实体 Byte 的范围请求
If-Unmodified-Since	比较资源的更新时间(与 If-Modified-Since相反)
Max-Forwards	最大传输逐跳数
Proxy-Authorization	代理服务器要求客户端的认证信息
Range	实体的字节范围请求
Referer	对请求中 URI 的原始获取方
User-Agent	HTTP 客户端程序的信息

HTTP 首部字段 - 响应首部字段

首部字段名	说明
Accept-Ranges	是否接受字节范围请求
Age	推算资源创建经过时间
ETag	资源的匹配信息
Location	令客户端重定向至指定 URI
Proxy-Authenticate	代理服务器对客户端的认证信息
Retry-After	对再次发起请求的时机要求
Server	HTTP服务器的安装信息
Vary	代理服务器缓存的管理信息
WWW-Authenticate	服务器对客户端的认证信息

HTTP 首部字段 - 实体首部字段

Allow	资源可支持的HTTP方法
Content-Encoding	实体主体适用的 编码方式
Content-Language	实体主体的自然 语言
Content-Length	实体主体的大小(单位: 字节)
Content-Location	替代对应资源的URI
Content-MD5	实体主体的 报文摘要
Content-Range	实体主体的位置范 围
Content-Type	实体主体的媒体 类型
Expires	实体主体过期的日期时间
Last-Modified	资源的最后修改日期 时间



HTTP 首部字段 - 其他

在 HTTP 协议通信交互中使用到的首部字段, 不限于 RFC2616 中定义的 47 种首部字段。还有 Cookie、Set-Cookie 和 Content-Disposition 等在其他 RFC 中定义的首部字段, 它们的使用频率也很高。




HTTP 首部字段 - Accept

Accept 首部字段可通知服务器，用户代理能够处理的媒体类型及媒体类型的相对优先级。可使用 type/subtype 这种形式来一次指定多种媒体类型。

- 文本文件: text/html, text/plain, text/css, application/xhtml+xml, application/xml ...
- 图片文件: image/jpeg, image/gif, image/png ...
- 视频文件: video/mpeg, video/quicktime ...
- 应用程序使用的二进制文件: application/octet-stream, application/zip ...

若想要给显示的媒体类型增加优先级，则使用 q= 来额外表示权重值 1，用分号(;)进行分隔。权重值 q 的范围是 0~1(可精确到小数点后 3 位)，且 1 为最大值。不指定权重 q 值时，默认权重为 q=1.0。



HTTP 首部字段 - Accept-Charset

Accept-Charset 首部字段可用来通知服务器用户代理支持的字符集及字符集的相对优先顺序。另外，可一次性指定多种字符集。与首部字段 Accept 相同的是可用权重 q 值来表示相对优先级。

该首部字段应用于内容协商机制的服务器驱动协商。

Accept-Charset: iso-8859-5, unicode-1-1;q=0.8



HTTP 首部字段 - Accept-Encoding

Accept-Encoding 首部字段用来告知服务器用户代理支持的内容编码及内容编码的优先级顺序。可一次性指定多种内容编码。

- gzip: 由文件压缩程序 gzip (GNU zip) 生成的编码格式 (RFC1952), 采用 Lempel-Ziv 算法 (LZ77) 及 32 位循环冗余校验 (Cyclic Redundancy Check, 通称 CRC)。
- compress: 由 UNIX 文件压缩程序 compress 生成的编码格式, 采用 LempelZiv-Welch 算法 (LZW)。
- deflate: 组合使用 zlib 格式 (RFC1950) 及由 deflate 压缩算法 (RFC1951) 生成的编码格式。
- Identity: 不执行压缩或不会变化的默认编码格式

采用权重 q 值来表示相对优先级, 这点与首部字段 Accept 相同。另外, 也可使用星号 (*) 作为通配符, 指定任意的编码格式。



HTTP 首部字段 - Accept-Language

Accept-Language: zh-cn,zh;q=0.7,en-us,en;q=0.3

首部字段 Accept-Language 用来告知服务器用户代理能够处理的自然语言集(指中文或英文等), 以及自然语言集的相对优先级。可一次指定多种自然语言集。

和 Accept 首部字段一样, 按权重值 q 来表示相对优先级。在上述例子中, 客户端在服务器有中文版资源的情况下, 会请求其返回中文版对应的响应, 没有中文版时, 则请求返回英文版响应。



HTTP 首部字段 - Authorization

Authorization: Basic dWVub3NIbjpwYXNzd29yZA==

首部字段 Authorization 是用来告知服务器, 用户代理的认证信息(证书值)。通常, 想要通过服务器认证的用户代理会在接收到返回的 401 状态码响应后, 把首部字段 Authorization 加入请求中。



HTTP 首部字段 - Host

首部字段 Host 会告知服务器，请求的资源所处的互联网主机名和端口号。Host 首部字段在 HTTP/1.1 规范内是唯一一个必须被包含在请求内的首部字段。

首部字段 Host 和以单台服务器分配多个域名的虚拟主机的工作机制有很密切的关联，这是首部字段 Host 必须存在的意义。

请求被发送至服务器时，请求中的主机名会用 IP 地址直接替换解决。但如果这时，相同的 IP 地址下部署运行着多个域名，那么服务器就会无法理解究竟是哪个域名对应的请求。因此，就需要使用首部字段 Host 来明确指出请求的主机名。

若服务器未设定主机名，那直接发送一个空值即可。如下所示。

Host:



HTTP 首部字段 - Range

Range: bytes=5001-10000

对于只需获取部分资源的范围请求，包含首部字段 Range 即可告知服务器资源的指定范围。上面的示例表示请求获取从第 5001 字节至第 10000 字节的资源。

接收到附带 Range 首部字段请求的服务器，会在处理请求之后返回状态码为 206 Partial Content 的响应。无法处理该范围请求时，则会返回状态码 200 OK 的响应及全部资源。



HTTP 首部字段 - Referer

Referer: <http://i.cuhk.edu.cn/login.aspx>

首部字段 Referer 会告知服务器请求的原始资源的 URI。

客户端一般都会发送 Referer 首部字段给服务器。但当直接在浏览器的地址栏输入 URI, 或出于安全性的考虑时, 也可以不发送该首部字段。因为原始资源的 URI 中的查询字符串可能含有 ID 和密码等保密信息, 要是写进 Referer 转发给其他服务器, 则有可能导致保密信息的泄露。

另外, Referer 的正确的拼写应该是 Referrer, 但不知为何, 大家一直沿用这个错误的拼写。



HTTP 首部字段 - User-Agent

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:13.0) Gecko/20100101

首部字段 User-Agent 会将创建请求的浏览器和用户代理名称等信息传达给服务器。

由网络爬虫发起请求时，有可能会在字段内添加爬虫作者的电子邮件地址。此外，如果请求经过代理，那么中间也很可能被添加上代理服务器的名称。



HTTP 首部字段 - Location

使用首部字段 Location 可以将响应接收方引导至某个与请求 URI 位置不同的资源。

基本上, 该字段会配合 3xx : Redirection 的响应, 提供重定向的 URI。

几乎所有的浏览器在接收到包含首部字段 Location 的响应后, 都会强制性地尝试对已提示的重定向资源的访问。



HTTP 首部字段 - Server

Server: Apache/2.2.17 (Unix)

首部字段 Server 告知客户端当前服务器上安装的 HTTP 服务器应用程序的信息。不单单会标出服务器上的软件应用名称, 还有可能包括版本号和安装时启用的可选项。

Server: Apache/2.2.6 (Unix) PHP/5.2.5



HTTP 首部字段 - Content-Encoding

Content-Encoding: gzip

首部字段 Content-Encoding 会告知客户端服务器对实体的主体部分选用的内容编码方式。内容编码是指在不丢失实体信息的前提下所进行的压缩。

编码的方式请参考此前 Accept-Encoding 首部字段。



HTTP 首部字段 - Content-Type

Content-Type: text/html; charset=UTF-8

首部字段 Content-Type 说明了实体主体内对象的媒体类型。和首部字段 Accept 一样，字段值用 type/subtype 形式赋值。

参数 charset 使用 iso-8859-1 或 euc-jp 等字符集进行赋值。



HTTP 首部字段 - DNT

DNT: 1

首部字段 DNT 属于 HTTP 请求首部, 其中 DNT 是 Do Not Track 的简称, 意为拒绝个人信息被收集, 是表示拒绝被精准广告追踪的一种方法。首部字段 DNT 可指定的字段值如下。

- 0 : 同意被追踪
- 1 : 拒绝被追踪

由于首部字段 DNT 的功能具备有效性, 所以 Web 服务器需要对 DNT 做对应的支持。



Practice: Try to get some headers?

- `curl --head -v http://www.cuhk.edu.cn`



HTTP 持久化连接

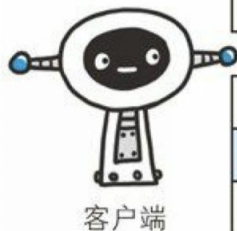
HTTP 协议的初始版本中，每进行一次 HTTP 通信就要断开一次 TCP 连接。

以当年的通信情况来说，因为都是些容量很小的文本传输，所以即使这样也没有多大问题。可随着 HTTP 的普及，文档中包含大量图片的情况多了起来。

比如，使用浏览器浏览一个包含多张图片的 HTML 页面时，在发送请求访问 HTML 页面资源的同时，也会请求该 HTML 页面里包含的其他资源。因此，每次的请求都会造成无谓的 TCP 连接建立和断开，增加通信量的开销。



发送请求一份包含多张图片的HTML文档对应的Web页面，会产生大量的通信开销。



建立TCP连接
HTTP 请求/响应
断开TCP连接

获取HTML文档

建立TCP连接
HTTP 请求/响应
断开TCP连接

获取图片

...

建立TCP连接
HTTP 请求/响应
断开TCP连接

获取图片



服务器



HTTP 持久化连接

为解决上述 TCP 连接的问题, HTTP/1.1 和一部分的 HTTP/1.0 想出了持久连接 (HTTP Persistent Connections, 也称为 HTTP keep-alive 或 HTTP connection reuse) 的方法。持久连接的特点是, 只要任意一端没有明确提出断开连接, 则保持 TCP 连接状态。

持久连接的好处在于减少了 TCP 连接的重复建立和断开所造成的额外开销 (Round-trip Time RTT), 减轻了服务器端的负载。另外, 减少开销的那部分时间, 使 HTTP 请求和响应能够更早地结束, 这样 Web 页面的显示速度也就相应提高了。



HTTP 持久化连接 - 相关的 HTTP 头部

Connection 首部字段具备如下两个作用。

- 控制不再转发给代理的首部字段
 - Connection: 不再转发的首部字段名
- 管理持久连接
 - Connection: close
 - HTTP/1.1 版本的默认连接都是持久连接。为此, 客户端会在持久连接上连续发送请求。当服务器端想明确断开连接时, 则指定 Connection 首部字段的值为 Close。



Stateless HTTP

HTTP 是一种不保存状态，即无状态(Stateless)协议。HTTP 协议自身不对请求和响应之间的通信状态进行保存。也就是说在 HTTP 这个级别，协议对于发送过的请求或响应都不做持久化处理。

使用 HTTP 协议，每当有新的请求发送时，就会有对应的新响应产生。协议本身并不保留之前一切的请求或响应报文的信息。这是为了更快地处理大量事务，确保协议的可伸缩性，而特意把 HTTP 协议设计成如此简单的。



Stateless HTTP - Cookies

HTTP/1.1 虽然是无状态协议，但为了实现期望的保持状态功能，于是引入了 Cookie 技术。有了 Cookie 再用 HTTP 协议通信，就可以管理状态了。

Cookie 会根据从服务器端发送的响应报文内的一个叫做 Set-Cookie 的首部字段信息，通知客户端保存 Cookie。当下次客户端再往该服务器 发送请求时，客户端会自动在请求报文中加入 Cookie 值后发送出去。

服务器端发现客户端发送过来的 Cookie 后，会去检查究竟是从哪一个客户端发来的连接请求，然后对比服务器上的记录，最后得到之前的状态信息。

1. 请求报文（没有 **Cookie** 信息的状态）

```
GET /reader/ HTTP/1.1  
Host: hackr.jp  
*首部字段内没有Cookie的相关信息
```

2. 响应报文（服务器端生成 **Cookie** 信息）

```
HTTP/1.1 200 OK  
Date: Thu, 12 Jul 2012 07:12:20 GMT  
Server: Apache  
<Set-Cookie: sid=1342077140226724; path=/; expires=Wed,  
10-Oct-12 07:12:20 GMT>  
Content-Type: text/plain; charset=UTF-8
```

3. 请求报文（自动发送保存着的 **Cookie** 信息）

```
GET /image/ HTTP/1.1  
Host: hackr.jp  
Cookie: sid=1342077140226724
```



Stateless HTTP - Cookies

调用 Cookie 时, 由于可校验 Cookie 的有效期, 以及发送方的域、路径、协议等信息, 所以正规发布的 Cookie 内的数据不会因来自其他 Web 站点和攻击者的攻击而泄露。

首部字段名	说明	首部类型
Set-Cookie	开始状态管理所使用的 Cookie 信息	响应首部字段
Cookie	服务器接收到的 Cookie 信息	请求首部字段



Stateless HTTP - Cookies 的属性

属性	说明
NAME=VALUE	赋予 Cookie 的名称和其值(必需项)
expires=DATE	Cookie 的有效期(若不明确指定 则默认为浏览器关闭前为止)
path=PATH	将服务器上的文件目录作为Cookie的适用对象(若不指定则默 认为文档所在的文件目录)
domain=域名	作为 Cookie 适用对象的域名(若不指定则默认为创建 Cookie 的服务器的域名)
Secure	仅在 HTTPS 安全通信时才会发送 Cookie
HttpOnly	加以限制, 使 Cookie 不能被 JavaScript 脚本访问



Stateless HTTP - Cookies 的属性

expires 属性

Cookie 的 expires 属性指定浏览器可发送 Cookie 的有效期。当省略 expires 属性时，其有效期仅限于维持浏览器会话 (Session) 时间段内。这通常限于浏览器应用程序被关闭之前。

另外，一旦 Cookie 从服务器端发送至客户端，服务器端就不存在可以显式删除 Cookie 的方法。但可通过覆盖已过期的 Cookie，实现对客户端 Cookie 的实质性删除操作。



Stateless HTTP - Cookies 的属性

path 属性

Cookie 的 path 属性可用于限制指定 Cookie 的发送范围的文件目录。不过另有办法可避开这项限制，看来对其作为安全机制的效果不能抱有期待。



Stateless HTTP - Cookies 的属性

domain 属性

通过 Cookie 的 domain 属性指定的域名可做到与结尾匹配一致。比如, 当指定 example.com 后, 除 example.com 以外, www.example.com 或 www2.example.com 等都可以发送 Cookie。

因此, 除了针对具体指定的多个域名发送 Cookie 之外, 不指定 domain 属性显得更安全。



Stateless HTTP - Cookies 的属性

secure 属性

Cookie 的 secure 属性用于限制 Web 页面仅在 HTTPS 安全连接时, 才可以发送 Cookie。



Stateless HTTP - Cookies 的属性

HttpOnly 属性

Cookie 的 HttpOnly 属性是 Cookie 的扩展功能, 它使 JavaScript 脚本无法获得 Cookie。其主要目的为防止跨站脚本攻击 (Cross-site scripting, XSS) 对 Cookie 的信息窃取。发送指定 HttpOnly 属性的 Cookie 的方法如下所示。

```
Set-Cookie: name=value; HttpOnly
```

通过上述设置, 通常从 Web 页面内还可以对 Cookie 进行读取操作。但使用 JavaScript 的 `document.cookie` 就无法读取附加 HttpOnly 属性后的 Cookie 的内容了。因此, 也就无法在 XSS 中利用 JavaScript 劫持 Cookie 了。



Stateless HTTP - Cookies Same-origin Policy

只要 Domain 跟 Path 一致的 Cookie，就會被視為同源。若是 Cookie 有加上一些特別的設定，便需要判斷 Scheme 是 HTTPS 或是 HTTP 才會送出 Cookie。

經過設定，子網域與母網域的 Cookie 可以共用。

當 Cookie 的 domain 被設定為：`Set-Cookie: name=value; domain=game.com`

此時的 Cookie 不會與子網域(例如：`login.game.com`)共用。但若是設定為：

`Set-Cookie: name=value; domain=.game.com`

則該 Cookie 可以在瀏覽器連上子網域時，也會一並回傳到網站主機。



Stateless HTTP - Sessions

Session 是在无状态的 HTTP 协议下，服务端记录用户状态时用于标识具体用户的机制。它是在服务端保存的用来跟踪用户的状态的数据结构，可以保存在文件、数据库或者集群中。在浏览器关闭后这次的 Session 就消失了，下次打开就不再拥有这个 Session。其实并不是 Session 消失了，而是 Session ID 变了，服务器端可能还是存着你上次的 Session ID 及其 Session 信息，只是他们是无主状态，也许一段时间后会被删除。

目前大多数的应用都是用 Cookie 实现 Session 跟踪的。第一次创建 Session 时，服务端会通过 HTTP 协议中反馈到客户端，需要在 Cookie 中记录一个 Session ID，以便今后每次请求时都可分辨你是谁。



缓存

缓存是指代理服务器或客户端本地磁盘内保存的资源副本。利用缓存可减少对源服务器的访问，因此也就节省了通信流量和通信时间。

缓存服务器是代理服务器的一种，并归类在缓存代理类型中。换句话说，当代理转发从服务器返回的响应时，代理服务器将会保存一份资源的副本。

缓存服务器的优势在于利用缓存可避免多次从源服务器转发资源。因此客户端可就近从缓存服务器上获取资源，而源服务器也不必多次处理相同的请求了。



缓存的有效期限

即便缓存服务器内有缓存，也不能保证每次都会返回对同资源的请求。因为这关系到被缓存资源的有效性问题。

当遇上源服务器上的资源更新时，如果还是使用不变的缓存，那就会演变成返回更新前的“旧”资源了。

即使存在缓存，也会因为客户端的要求、缓存的有效期等因素，向源服务器确认资源的有效性。若判断缓存失效，缓存服务器将会再次从源服务器上获取“新”资源。



客户端的缓存

缓存不仅可以存在于缓存服务器内，还可以存在于客户端浏览器中。浏览器缓存如果有效，就不必再向服务器请求相同的资源了，可以直接从本地磁盘内读取。

另外，和缓存服务器相同的一点是，当判定缓存过期后，会向源服务器确认资源的有效性。若判断浏览器缓存失效，浏览器会再次请求新资源。



HTTP 与缓存相关的首部字段

- Cache-Control: 通过指定首部字段 Cache-Control 的指令, 就能操作缓存的工作机制。
- Expires: 资源失效的日期。源服务器不希望缓存服务器对资源缓存时, 最好在 Expires 字段内写入与首部字段 Date 相同的时间值。
- Last-Modified: 指明资源最终修改的时间。

Emmm 关于头部我们就讲到这里.....
接下来, 讲点更硬核的 HTTPS



HTTP Is Not Secure

到现在为止，我们已了解到 HTTP 具有相当优秀和方便的一面，然而 HTTP 并非只有好的一面，事物皆具两面性，它也是有不足之处的。HTTP 主要有这些不足，例举如下。

- 通信使用明文(不加密)，内容可能会被窃听
- 不验证通信方的身份，因此有可能遭遇伪装
- 无法证明报文的完整性，所以有可能已遭篡改

这些问题不仅在 HTTP 上出现，其他未加密的协议中也会存在这类问题。



HTTP Is Not Secure - 通信使用明文可能会被窃听

由于 HTTP 本身不具备加密的功能, 所以也无法做到对通信整体(使用 HTTP 协议通信的请求和响应的内容)进行加密。即, HTTP 报文使用明文(指未经过加密的报文)方式发送。

TCP/IP 是可能被窃听的网络

如果要问为什么通信时不加密是一个缺点, 这是因为, 按 TCP/IP 协议族的工作机制, 通信内容在所有的通信线路上都有可能遭到窥视。



HTTP Is Not Secure - 通信使用明文可能会被窃听

加密处理防止被窃听

- 通信的加密: HTTP 协议中没有加密机制, 但可以通过和 SSL (Secure Socket Layer, 安全套接层) 或 TLS (Transport Layer Security, 安全层传输协议) 的组合使用, 加密 HTTP 的通信内容。用 SSL 建立安全通信线路之后, 就可以在这条线路上进行 HTTP 通信了。与 SSL 组合使用的 HTTP 被称为 HTTPS (HTTP Secure, 超文本传输安全协议) 或 HTTP over SSL。
- 内容的加密: 把 HTTP 报文里所含的内容进行加密处理。在这种情况下, 客户端需要对 HTTP 报文进行加密处理后再发送请求。由于该方式不同于 SSL 或 TLS 将整个通信线路加密处理, 所以内容仍有被篡改的风险。



HTTP - 不验证通信方的身份就可能遭遇伪装

HTTP 协议中的请求和响应不会对通信方进行确认。也就是说存在“服务器是否就是发送请求中 URI 真正指定的主机, 返回的响应是否真的 返回到实际提出请求的客户端”等类似问题。

任何人都可发起请求

在 HTTP 协议通信时, 由于不存在确认通信方的处理步骤, 任何人都可以发起请求。另外, 服务器只要接收到请求, 不管对方是谁都会返回一个响应(但也仅限于发送端的 IP 地址和端口号没有被 Web 服务器设定限制访问的前提下)。



HTTP - 不验证通信方的身份就可能遭遇伪装

查明对手的证书

虽然使用 HTTP 协议无法确定通信方，但如果使用 SSL 则可以。SSL 不仅提供加密处理，而且还使用了一种被称为证书的手段，可用于确定方。

证书由值得信任的第三方机构颁发，用以证明服务器和客户端是实际存在的。另外，伪造证书从技术角度来说是非常困难的一件事。所以只要能够确认通信方（服务器或客户端）持有的证书，即可判断通信方的真实意图。

通过使用证书，以证明通信方就是意料中的服务器。这对使用者个人来讲，也减少了个人信息泄露的危险性。另外，客户端持有证书即可完成个人身份的确认，也可用于对 Web 网站的认证环节。



HTTP - 无法证明报文完整性，可能已遭篡改

所谓完整性是指信息的准确度。若无法证明其完整性，通常也就意味着无法判断信息是否准确。

接收到的内容可能有误

由于 HTTP 协议无法证明通信的报文完整性，因此，在请求或响应送出之后直到对方接收之前的这段时间内，即使请求或响应的内容遭到篡改，也没有办法获悉。换句话说，没有任何办法确认，发出的请求 / 响应和接收到的请求 / 响应是前后相同的。

请求或响应在传输途中，遭攻击者拦截并篡改内容的攻击称为中间人攻击 (Man-in-the-Middle attack, MITM)。



HTTP - 无法证明报文完整性，可能已遭篡改

SSL 提供认证和加密处理及摘要功能。

仅靠 HTTP 确保完整性是非常困难的，因此通过和其他协议组合使用来实现这个目标。

HTTP + 加密 + 认证 + 完整性保护 = HTTPS

HTTPS 是身披 SSL 外壳的 HTTP

HTTPS 并非是应用层的一种新协议。只是 HTTP 通信接口部分用 SSL (Secure Socket Layer) 和 TLS (Transport Layer Security) 协议代替而已。

通常, HTTP 直接和 TCP 通信。当使用 SSL 时, 则演变成先和 SSL 通信, 再由 SSL 和 TCP 通信了。简言之, 所谓 HTTPS, 其实就是身披 SSL 协议这层外壳的 HTTP。

在采用 SSL 后, HTTP 就拥有了 HTTPS 的加密、证书和完整性保护这些功能。



HTTP



HTTPS



Cryptography

在对 SSL 进行讲解之前，我们先来了解一下加密方法。SSL 采用一种叫做公开密钥加密 (Public-key cryptography) 的加密处理方式。

近代的加密方法中加密算法是公开的，而密钥却是保密的。通过这种方式得以保持加密方法的安全性。

加密和解密都会用到密钥。没有密钥就无法对密码解密，反过来说，任何人只要持有密钥就能解密了。如果密钥被攻击者获得，那加密也就失去了意义。



Cryptography

共享密钥加密的困境

加密和解密同用一个密钥的方式称为共享密钥加密(Common key crypto system), 也被叫做对称密钥加密。

以共享密钥方式加密时必须将密钥也发给对方。可究竟怎样才能安全地转交? 在互联网上转发密钥时, 如果通信被监听那么密钥 就可会落入攻击者之手, 同时也就失去了加密的意义。另外还得设法安全地保管接收到的密钥。



Cryptography

使用两把密钥的公开密钥加密

公开密钥加密方式很好地解决了共享密钥加密的困难。

公开密钥加密使用一对非对称的密钥。一把叫做私有密钥 (Private key)，另一把叫做公开密钥 (Public key)。私钥不能让其他任何人知道，而公钥则可以随意发布，任何人都可以获得。

使用公开密钥加密方式，发送密文的一方使用对方的公开密钥进行加密处理，对方收到被加密的信息后，再使用自己的私有密钥进行解密。利用这种方式，不需要发送用来解密的私有密钥，也不必担心密钥被攻击者窃听而盗走。



Cryptography

使用两把密钥的公开密钥加密

.....

另外，要想根据密文和公开密钥，恢复到信息原文是异常困难的，因为解密过程就是在对离散对数进行求值，这并非轻而易举就能办到。退一步讲，如果能对一个非常大的整数做到快速地因式分解，那么密码破解还是存在希望的。但就目前的技术来看是不太现实的。



Cryptography

HTTPS 采用混合加密机制

HTTPS 采用共享密钥加密和公开密钥加密两者并用的混合加密机制。若密钥能够实现安全交换，那么有可能会考虑仅使用公开密钥加密来通信。但是公开密钥加密与共享密钥加密相比，其处理速度要慢。

所以应充分利用两者各自的优势，将多种方法组合起来用于通信。在交换密钥环节使用公开密钥加密方式，之后的建立通信交换报文阶段则使用共享密钥加密方式。

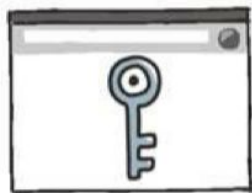


SSL - 证明公开密钥正确性的证书

遗憾的是，公开密钥加密方式还是存在一些问题的。那就是无法证明公开密钥本身就是货真价实的公开密钥。比如，正准备和某台服务器建立公开密钥加密方式下的通信时，如何证明收到的公开密钥就是原本预想的那台服务器发行的公开密钥。或许在公开密钥传输途中，真正的公开密钥已经被攻击者替换掉了。

为了解决上述问题，可以使用由数字证书认证机构(CA, Certificate Authority)和其相关机关颁发的公开密钥证书。

数字证书认证机构处于客户端与服务器双方都可信赖的第三方机构的立场上。



数字证书认证机构的
公开密钥已事先
植入到浏览器里了



数字证书认证机构



数字证书认证机构的私有密钥

②数字证书认证机构用自己的私有密钥向服务器的公开密码署数字签名并颁发公钥证书

①服务器把自己的公开密钥
登录至数字证书认证机构



服务器的
公开密钥

③客户端拿到服务器的公
钥证书后，使用数字证
书认证机构的公开密
钥，向数字证书认证机
构验证公钥证书上的数
字签名，以确认服务器
的公开密钥的真实性



服务器的
公开密钥
+
数字证书认证
机构的数字签名
公钥证书



客户端

④使用服务器的公开密钥对报
文加密后发送

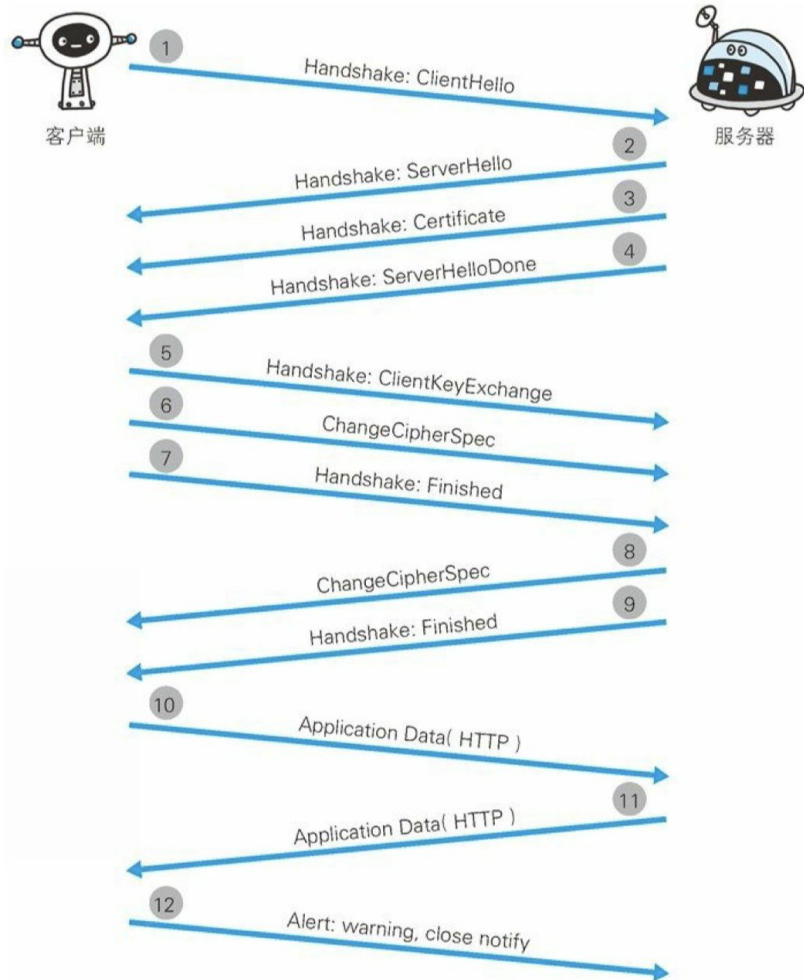


服务器



服务器的
私有密钥

⑤服务器用私
有密钥对报
文解密





HTTPS 的安全通信机制

步骤 1: 客户端通过发送 Client Hello 报文开始 SSL 通信。报文中包含客户端支持的 SSL 的指定版本、加密组件 (Cipher Suite) 列表 (所使用的加密算法及密钥长度等)。

步骤 2: 服务器可进行 SSL 通信时, 会以 Server Hello 报文作为应答。和客户端一样, 在报文中包含 SSL 版本以及加密组件。服务器的加密组件内容是从接收到的客户端加密组件内筛选出来的。

步骤 3: 之后服务器发送 Certificate 报文。报文中包含公开密钥证书。

步骤 4: 最后服务器发送 Server Hello Done 报文通知客户端, 最初阶段的 SSL 握手协商部分结束。

步骤 5: SSL 第一次握手结束之后, 客户端以 Client Key Exchange 报文作为回应。报文中包含通信加密中使用的一种被称为 Pre-master secret 的随机密码串。该报文已用步骤 3 中的公开密钥进行加密。



HTTPS 的安全通信机制

步骤 6: 接着客户端继续发送 Change Cipher Spec 报文。该报文会提示服务器, 在此报文之后的通信会采用 Pre-master secret 密钥加密。

步骤 7: 客户端发送 Finished 报文。该报文包含连接至今全部报文的 整体校验值。这次握手协商是否能够成功, 要以服务器是否能够正确解密该报文作为判定标准。

步骤 8: 服务器同样发送 Change Cipher Spec 报文。

步骤 9: 服务器同样发送 Finished 报文。

步骤 10: 服务器和客户端的 Finished 报文交换完毕之后, SSL 连接 就算建立完成。当然, 通信会受到SSL 的保护。从此处开始进行应用 层协议的通信, 即发送 HTTP 请求。

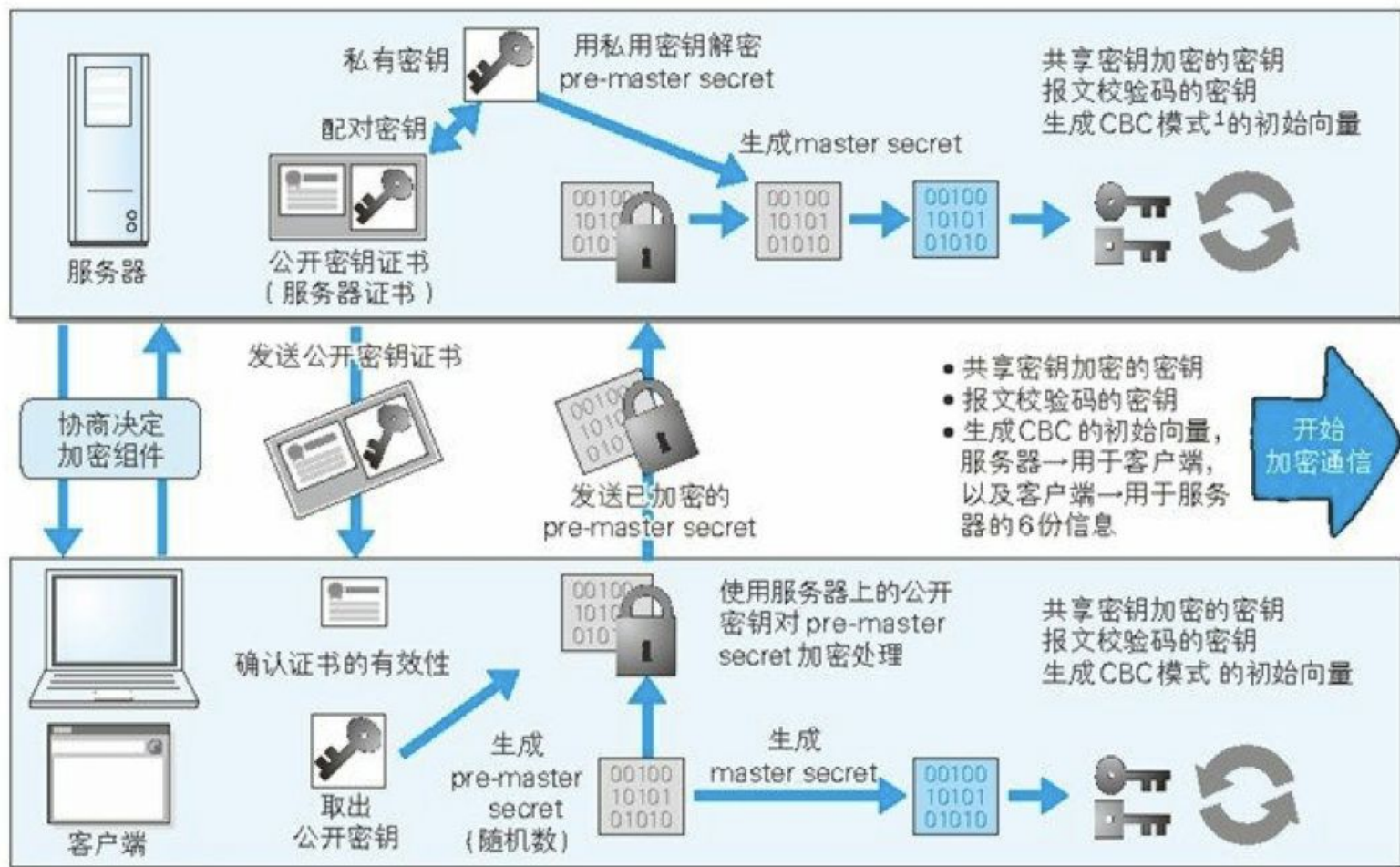


HTTPS 的安全通信机制

步骤 11: 应用层协议通信, 即发送 HTTP 响应。

步骤 12: 最后由客户端断开连接。断开连接时, 发送 close_notify 报文。上图做了一些省略, 这步之后再发送 TCP FIN 报文来关闭与 TCP 的通信。

在以上流程中, 应用层发送数据时会附加一种叫做 MAC (Message Authentication Code) 的报文摘要。MAC 能够查知报文是否遭到篡改, 从而保护报文的完整性。





SSL v.s. TLS

HTTPS 使用 SSL (Secure Socket Layer) 和 TLS (Transport Layer Security) 这两个协议。

SSL 技术最初是由浏览器开发商网景通信公司率先倡导的, 开发过 SSL 3.0 之前的版本。目前主导权已转移到 IETF (Internet Engineering Task Force, Internet 工程任务组) 的手中。

IETF 以 SSL 3.0 为基准, 后又制定了 TLS 1.0、TLS 1.1、TLS 1.2 和 TLS 1.3。TLS 是以 SSL 为原型开发的协议, 有时会统一称该协议为 SSL。当前主流的版本是 TLS 1.1 和 TLS 1.2。

由于 SSL 1.0 协议在设计之初被发现出了问题, 就没有实际投入使用。SSL 2.0 也被发现存在问题, 所以很多浏览器直接废除了该协议版本。



WebSocket

HTTP 是单向通讯的, 即只能由客户端主动发起请求。WebSocket 的出现, 使得浏览器具备了实时双向通信的能力。

WebSocket 是 HTML 5 开始提供的一种浏览器与服务器进行全双工通讯的网络技术, 属于应用层协议。它基于 TCP 传输协议, 并复用 HTTP 的握手通道 - 不是 HTTP 协议的一部分。



WebSocket 优势

- 支持双向通信，实时性更强。
- 更好的二进制支持。
- 较少的控制开销。连接创建后，ws客户端、服务端进行数据交换时，协议控制的数据包头部较小。在不包含头部的情况下，服务端到客户端的包头只有2~10字节(取决于数据包长度)，客户端到服务端的的话，需要加上额外的4字节的掩码。而HTTP协议每次通信都需要携带完整的头部。
- 支持扩展。ws协议定义了扩展，用户可以扩展协议，或者实现自定义的子协议。(比如支持自定义压缩算法等)



HTTP/2

HTTP/1.1 标准自 1999 年发布的 RFC2616 之后再未进行过改订。SPDY 和 WebSocket 等技术纷纷出现, 很难断言 HTTP/1.1 仍是适用于当下的 Web 的协议。

负责互联网技术标准的 IETF (Internet Engineering Task Force, 互联网工程任务组) 创立 httpbis (Hypertext Transfer Protocol Bis, <http://datatracker.ietf.org/wg/httpbis/>) 工作组, 其目标是推进下一代 HTTP——HTTP/2.0 在 2014 年 11 月实现标准化。



Why not HTTP/1.1?

1、TCP 连接数限制

对于同一个域名，浏览器最多只能同时创建 6~8 个 TCP 连接（不同浏览器不一样）。为了解决数量限制，出现了域名分片技术，其实就是资源分域，将资源放在不同域名下（比如二级子域名下），这样就可以针对不同域名创建连接并请求，以一种讨巧的方式突破限制，但是滥用此技术也会造成很多问题，比如每个 TCP 连接本身需要经过 DNS 查询、三步握手、慢启动等，还占用额外的 CPU 和内存，对于服务器来说过多连接也容易造成网络拥挤、交通阻塞等，对于移动端来说问题更明显。



Why not HTTP/1.1?

2、线头阻塞 (Head Of Line Blocking) 问题

每个 TCP 连接同时只能处理一个请求 - 响应, 浏览器按 FIFO 原则处理请求, 如果上一个响应没返回, 后续请求 - 响应都会受阻。为了解决此问题, 出现了 管线化 技术, 但是管线化存在诸多问题, 比如第一个响应慢还是会阻塞后续响应、服务器为了按序返回相应需要缓存多个响应占用更多资源、浏览器中途断连重试服务器可能得重新处理多个请求、还有必须客户端、代理、服务器都支持管线化。



Why not HTTP/1.1?

3、Header 内容多，而且每次请求 Header 不会变化太多，没有相应的压缩传输优化方案。

4、为了尽可能减少请求数，需要做合并文件、雪碧图、资源内联等优化工作，但是这无疑造成了单个请求内容变大延迟变高的问题，且内嵌的资源不能有效地使用缓存机制。

5、明文传输不安全。



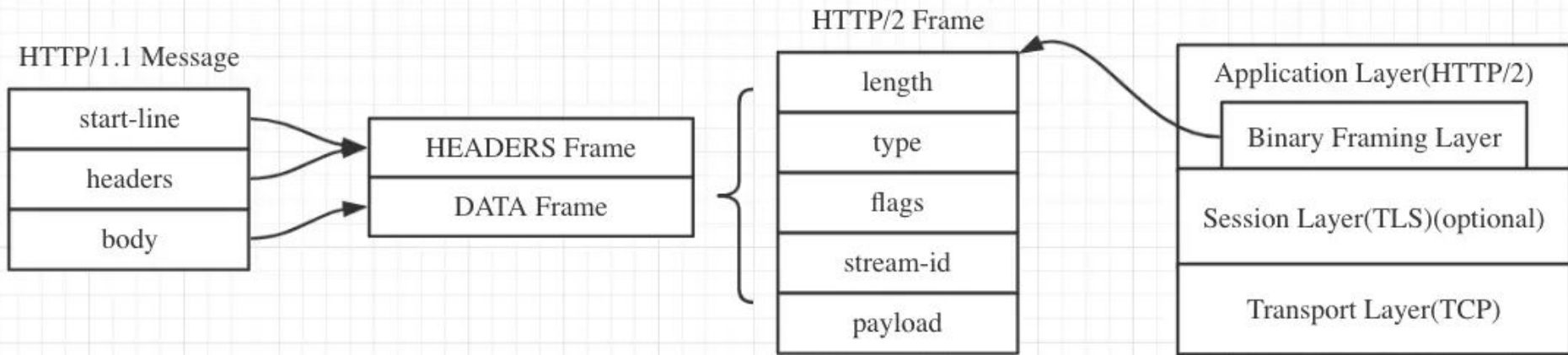
HTTP/2

HTTP/2 的主要目标是通过支持完整的请求与响应复用来减少延迟，通过有效压缩 HTTP 标头字段将协议开销降至最低，同时增加对请求优先级和服务器推送的支持。为达成这些目标，HTTP/2 还给我们带来了大量其他协议层面的辅助实现，例如新的流控制、错误处理和升级机制。上述几种机制虽然不是全部，但却是最重要的，每一位网络开发者都应该理解并在自己的应用中加以利用。

HTTP/2 没有改动 HTTP 的应用语义。HTTP 方法、状态代码、URI 和标头字段等核心概念一如往常。不过，HTTP/2 修改了数据格式化（分帧）以及在客户端与服务器间传输的方式。这两点统帅全局，通过新的分帧层向我们的应用隐藏了所有复杂性。因此，所有现有的应用都可以不必修改而在新协议下运行。

HTTP/2 - 二进制分帧层 (Binary Framing Layer)

帧是数据传输的最小单位，以二进制传输代替原本的明文传输，原本的报文消息被划分为更小的数据帧：





HTTP/2 - 多路复用 (MultiPlexing)

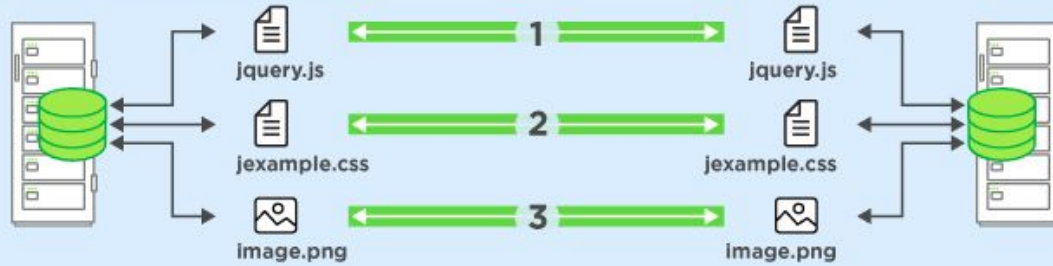
在一个 TCP 连接上, 我们可以向对方不断发送帧, 每帧的 Stream Identifier 的标明这一帧属于哪个流, 然后在对方接收时, 根据 Stream Identifier 拼接每个流的所有帧组成一整块数据。

把 HTTP/1.1 每个请求都当作一个流, 那么多个请求变成多个流, 请求响应数据分成多个帧, 不同流中的帧交错地发送给对方, 这就是 HTTP/2 中的多路复用。

流的概念实现了单连接上多请求 - 响应并行, 解决了线头阻塞的问题, 减少了 TCP 连接数量和 TCP 连接慢启动造成的问题。

所以 http2 对于同一域名只需要创建一个连接, 而不是像 http/1.1 那样创建 6~8 个连接。

3 TCP CONNECTIONS



1 TCP CONNECTION





HTTP/2 - 服务端推送 (Server Push)

浏览器发送一个请求，服务器主动向浏览器推送与这个请求相关的资源，这样浏览器就不用发起后续请求。

Server-Push 主要是针对资源内联做出的优化，相较于 http/1.1 资源内联的优势：

- 客户端可以缓存推送的资源
- 客户端可以拒收推送过来的资源
- 推送资源可以由不同页面共享
- 服务器可以按照优先级推送资源



HTTP/2 - Header 压缩

4、Header 压缩 (HPACK): 使用 HPACK 算法来压缩首部内容

每个 HTTP 传输都承载一组标头, 这些标头说明了传输的资源及其属性。在 HTTP/1.x 中, 此元数据始终以纯文本形式, 通常会给每个传输增加 500-800 字节的开销。如果使用 Cookie, 增加的开销有时能达到上千字节。为了减少此开销和提升性能, HTTP/2 使用 HPACK 压缩格式压缩请求和响应标头元数据, 这种格式采用两种简单但是强大的技术:

1. 这种格式支持通过静态霍夫曼代码对传输的标头字段进行编码, 从而减小了各个传输的大小。
2. 这种格式要求客户端和服务端同时维护和更新一个包含之前见过的标头字段的索引列表(换句话说, 它可以建立一个共享的压缩上下文), 此列表随后会用作参考, 对之前传输的值进行有效编码。



HTTP/2 - Header 压缩

利用霍夫曼编码，可以在传输时对各个值进行压缩，而利用之前传输值的索引列表，我们可以通过传输索引值的方式对重复值进行编码，索引值可用于有效查询和重构完整的标头键值对。

作为一种进一步优化方式，HPACK 压缩上下文包含一个静态表和一个动态表：静态表在规范中定义，并提供了一个包含所有连接都可能使用的常用 HTTP 标头字段（例如，有效标头名称）的列表；动态表最初为空，将根据在特定连接内交换的值进行更新。因此，为之前未见过的值采用静态 Huffman 编码，并替换每一侧静态表或动态表中已存在值的索引，可以减小每个请求的大小。

注：在 HTTP/2 中，请求和响应头部字段的定义保持不变，仅有微小差异：所有标头字段名称均为小写，请求行拆分成各个 `:method`、`:scheme`、`:authority` 和 `:path` 伪标头字段。

Request #1

:method	GET
:scheme	https
:host	example.com
:path	/resource
accept	image/jpeg
user-agent	Mozilla/5.0 ...

implicit

implicit

implicit

implicit

implicit

Request #2


:method	GET
:scheme	https
:host	example.com
:path	/new_resource
accept	image/jpeg
user-agent	Mozilla/5.0 ...

HEADERS frame (Stream 1)

:method: GET
:scheme: https
:host: example.com
:path: /resource
accept: image/jpeg
user-agent: Mozilla/5.0 ...

HEADERS frame (Stream 3)

:path: /new_resource



HTTP/2 - Others

5、应用层的重置连接:对于 HTTP/1 来说, 是通过设置 tcp segment 里的 reset flag 来通知对端关闭连接的。这种方式会直接断开连接, 下次再发请求就必须重新建立连接。HTTP/2 引入 RST_STREAM 类型的 frame, 可以在不断开连接的前提下取消某个 request 的 stream, 表现更好。

6、请求优先级设置:HTTP/2 里的每个 stream 都可以设置依赖 (Dependency) 和权重, 可以按依赖树分配优先级, 解决了关键请求被阻塞的问题。

7、流量控制:每个 http2 流都拥有自己的公示的流量窗口, 它可以限制另一端发送数据。对于每个流来说, 两端都必须告诉对方自己还有足够的空间来处理新的数据, 而在该窗口被扩大前, 另一端只被允许发送这么多数据。



HTTP/3

Let's talk about history again...



图1-1 HTTP历史



HTTP/3

HTTP-over-QUIC -> HTTP/3

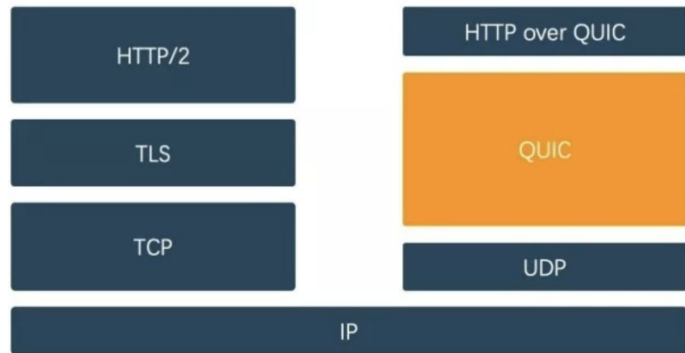
HTTP/3 is the coming new HTTP version that uses QU
for transport!



QUIC

QUIC

TCP 一直是传输层中举足轻重的协议，而 UDP 则默默无闻，在面试中问到 TCP 和 UDP 的区别时，有关 UDP 的回答常常寥寥几语，长期以来 UDP 给人的印象就是一个很快但不可靠的传输层协议。但有时候从另一个角度看，缺点可能也是优点。QUIC(Quick UDP Internet Connections, 快速 UDP 网络连接) 基于 UDP，正是看中了 UDP 的速度与效率。同时 QUIC 也整合了 TCP、TLS 和 HTTP/2 的优点，并加以优化。用一张图可以清晰地表示他们之间的关系。





零 RTT 建立连接

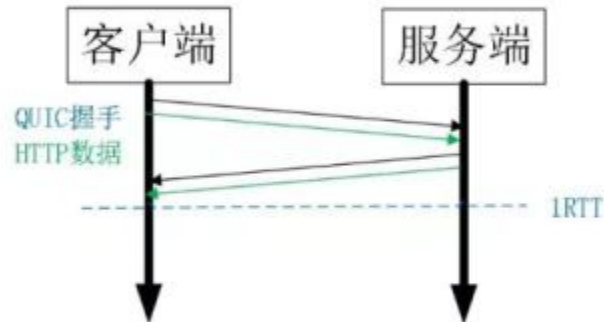
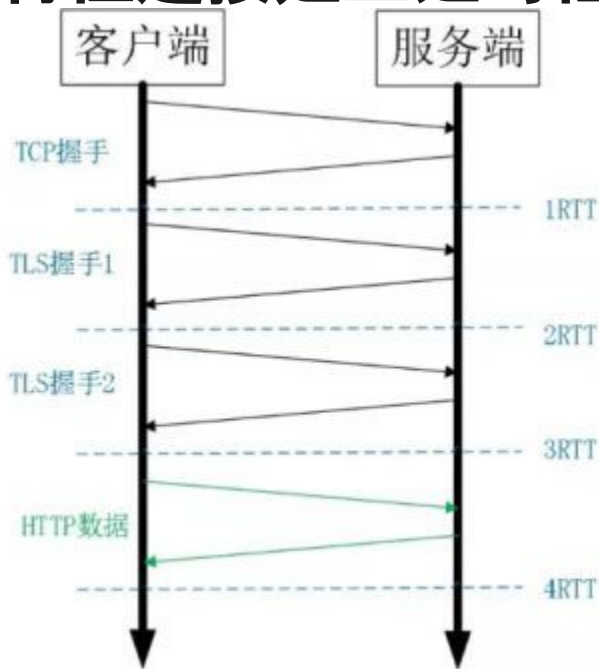
HTTP/2 的连接需要 3 RTT，如果考虑会话复用，即把第一次握手算出来的对称密钥缓存起来，那么也需要 2 RTT，更进一步的，如果 TLS 升级到 1.3，那么 HTTP/2 连接需要 2 RTT，考虑会话复用则需要 1 RTT。有人会说 HTTP/2 不一定需要 HTTPS，握手过程还可以简化。这没毛病，HTTP/2 的标准的确不需要基于 HTTPS，但实际上所有浏览器的实现都要求 HTTP/2 必须基于 HTTPS，所以 HTTP/2 的加密连接必不可少。

而 HTTP/3 首次连接只需要 1 RTT，后面的连接更是只需 0 RTT，意味着客户端发给服务端的第一个包就带有请求数据，这一点 HTTP/2 难以望其项背。那这背后是什么原理呢？

QUIC 核心特性连接建立延时低 (oRTT)

传输层 ORTT 就能建立连接。

加密层 ORTT 就能建立加密连接。



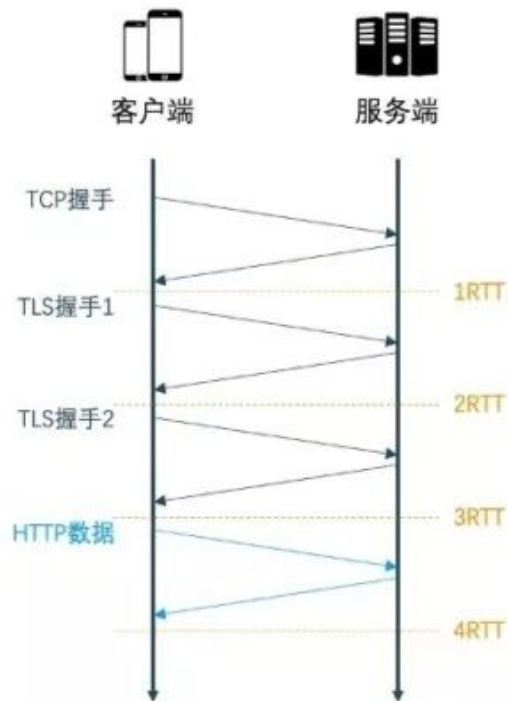


图2-2 TCP建立连接

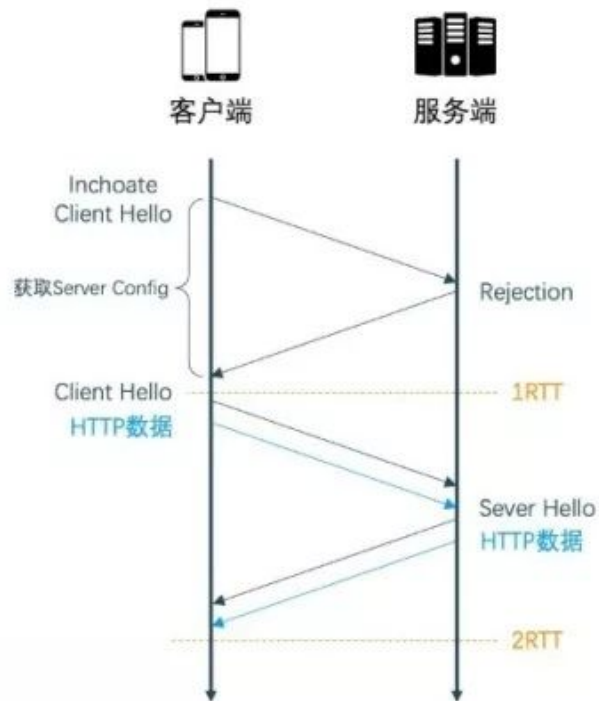


图2-3 QUIC建立连接



QUIC - 0RTT

Step1: 首次连接时, 客户端发送 Inchoate Client Hello 给服务端, 用于请求连接;

Step2: 服务端生成 g 、 p 、 a , 根据 g 、 p 和 a 算出 A , 然后将 g 、 p 、 A 放到 Server Config 中再发送 Rejection 消息给客户端;

Step3: 客户端接收到 g 、 p 、 A 后, 自己再生成 b , 根据 g 、 p 、 b 算出 B , 根据 A 、 p 、 b 算出初始密钥 K 。 B 和 K 算好后, 客户端会用 K 加密 HTTP 数据, 连同 B 一起发送给服务端;

Step4: 服务端接收到 B 后, 根据 a 、 p 、 B 生成与客户端同样的密钥, 再用这密钥解密收到的 HTTP 数据。为了进一步的安全(前向安全性), 服务端会更新自己的随机数 a 和公钥, 再生成新的密钥 S , 然后把公钥通过 Server Hello 发送给客户端。连同 Server Hello 消息, 还有 HTTP 返回数据;

Step5: 客户端收到 Server Hello 后, 生成与服务端一致的新密钥 S , 后面的传输都使用 S 加密。



QUIC - 0RTT

QUIC 从请求连接到正式接发 HTTP 数据一共花了 1 RTT, 这 1 个 RTT 主要是为了获取 Server Config, 后面的连接如果客户端缓存了 Server Config, 那么就可以直接发送 HTTP 数据, 实现 0 RTT 建立连接。



QUIC - 连接迁移

TCP 连接基于四元组(源 IP、源端口、目的 IP、目的端口), 切换网络时至少会有一个因素发生变化, 导致连接发生变化。当连接发生变化时, 如果还使用原来的 TCP 连接, 则会导致连接失败, 就得等原来的连接超时后重新建立连接, 所以我们有时候发现切换到一个新网络时, 即使新网络状况良好, 但内容还是需要加载很久。如果实现得好, 当检测到网络变化时立刻建立新的 TCP 连接, 即使这样, 建立新的连接还是需要几百毫秒的时间。

QUIC 的连接不受四元组的影响, 当这四个元素发生变化时, 原连接依然维持。那这是怎么做到的呢? 道理很简单, QUIC 连接不以四元组作为标识, 而是使用一个 64 位的随机数, 这个随机数被称为 Connection ID, 即使 IP 或者端口发生变化, 只要 Connection ID 没有变化, 那么连接依然可以维持。



HTTP 1.1/2 - 队头阻塞

HTTP/1.1 和 HTTP/2 都存在队头阻塞问题(Head of line blocking), 那什么是队头阻塞呢?

TCP 是个面向连接的协议, 即发送请求后需要收到 ACK 消息, 以确认对方已接收到数据。如果每次请求都要在收到上次请求的 ACK 消息后再请求, 那么效率无疑很低, 之后 HTTP/1.1 提出了 Pipelining 技术, 允许一个 TCP 连接同时发送多个请求, 这样就大大提升了传输效率。在这个背景下, 下面就来谈 HTTP/1.1 的队头阻塞。一个 TCP 连接同时传输 10 个请求, 其中第 1、2、3 个请求已被客户端接收, 但第 4 个请求丢失, 那么后面第 5 - 10 个请求都被阻塞, 需要等第 4 个请求处理完毕才能被处理, 这样就浪费了带宽资源。因此, HTTP 一般又允许每个主机建立 6 个 TCP 连接, 这样可以更加充分地利用带宽资源, 但每个连接中队头阻塞的问题还是存在。



HTTP 1.1/2 - 队头阻塞

HTTP/2 的多路复用解决了上述的队头阻塞问题。不像 HTTP/1.1 中只有上一个请求的所有数据包被传输完毕下一个请求的数据包才可以被传输，HTTP/2 中每个请求都被拆分成多个 Frame 通过一条 TCP 连接同时被传输，这样即使一个请求被阻塞，也不会影响其他的请求。

事情还没完，HTTP/2 虽然可以解决“请求”这个粒度的阻塞，但 HTTP/2 的基础 TCP 协议本身却也存在着队头阻塞的问题。HTTP/2 的每个请求都会被拆分成多个 Frame，不同请求的 Frame 组合成 Stream，Stream 是 TCP 上的逻辑传输单元，这样 HTTP/2 就达到了一条连接同时发送多条请求的目标，这就是多路复用的原理。

队头阻塞主要是 TCP 协议的可靠性机制引入的。TCP 使用序列号来标识数据的顺序，数据必须按照顺序处理，如果前面的数据丢失，后面的数据就算到达了也不会通知应用层来处理。另外 TLS 协议层面也有一个队头阻塞，因为 TLS 协议都是按照 record 来处理数据的，如果一个 record 中丢失了数据，也会导致整个 record 无法正确处理。

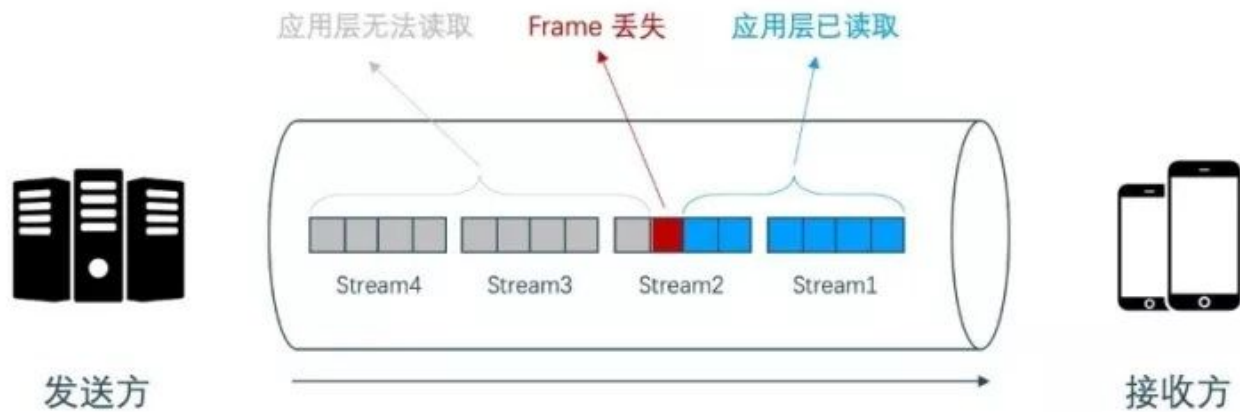


图2-9 TCP队头阻塞



QUIC - Solve?

QUIC 最基本的传输单元是 Packet, 不会超过 MTU 的大小, 整个加密和认证过程都是基于 Packet 的, 不会跨越多个 Packet。这样就能避免 TLS 协议存在的队头阻塞。

Stream 之间相互独立, 比如 Stream2 丢了一个 Packet, 不会影响 Stream3 和 Stream4。不存在 TCP 队头阻塞。

当然, 并不是所有的 QUIC 数据都不会受到队头阻塞的影响, 比如 QUIC 当前也是使用 Hpack 压缩算法 [10], 由于算法的限制, 丢失一个头部数据时, 可能遇到队头阻塞。

总体来说, QUIC 在传输大量数据时, 比如视频, 受到队头阻塞的影响很小。



HTTP/3 <-> QUIC

QUIC 丢掉了 TCP、TLS 的包袱, 基于 UDP, 并对 TCP、TLS、HTTP/2 的经验加以借鉴、改进, 实现了一个安全高效可靠的 HTTP 通信协议。凭借着零 RTT 建立连接、平滑的连接迁移、基本消除了队头阻塞、改进的拥塞控制和流量控制等优秀的特性, QUIC 在绝大多数场景下获得了比 HTTP/2 更好的效果, HTTP/3 未来可期。



More to go...



Reference

书名:图解HTTP

作者:上野 宣

译者:于均良

ISBN:978-7-115-35153-1

出版社:人民邮电出版社



Reference

- <https://www.jishuwen.com/d/2MZN/zh-hk>
- <https://juejin.im/post/5be935f2e51d4570813b8cf0>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>
- <https://medium.com/@jaydenlin/same-origin-policy-%E5%90%8C%E6%BA%90%E6%94%BF%E7%AD%96-%E4%B8%80%E5%88%87%E5%AE%89%E5%85%A8%E7%9A%84%E5%9F%BA%E7%A4%8E-36432565a226>
- <https://medium.com/@factoryhr/http-2-the-difference-between-http-1-1-benefits-and-how-to-use-it-38094fa0e95b>
- <https://developers.google.com/web/fundamentals/performance/http2/?hl=zh-cn>
- <https://juejin.im/post/5b88a4f56fb9a01a0b31a67e>
- <https://zhuanlan.zhihu.com/p/58668946>