HTTP 超全汇总

HTTP 超全汇总

- 认识 HTTP
  - 什么是超文本
  - 什么是传输
  - 什么是协议
- 与 HTTP 有关的组件
  - 网络模型
    - 应用层
    - 运输层
    - 网络层
    - 链路层
    - 物理层
  - OSI 模型
  - 浏览器
  - Web 服务器
  - CDN
  - WAF
  - WebService
  - HTML
  - Web 页面构成
- 与 HTTP 有关的协议
  - TCP/IP
  - DNS
  - URI / URL
  - HTTPS
- HTTP 请求响应过程
- HTTP 请求特征
- 详解 HTTP 报文
  - HTTP 请求方法
  - HTTP 请求 URL
  - HTTP 版本
- HTTP 标头
  - 通用标头
    - Cache-Control
    - Connection
    - Date
    - Pragma
    - Trailer
    - Transfer-Encoding
    - Upgrade
    - Via
    - Warning

- 请求标头
  - Accept
  - Accept-Charset
  - Accept-Encoding
  - Accept-Language
  - Authorization
  - Expect
  - From
  - Host
  - If-Match
  - If-Modified-Since
  - If-None-Match
  - If-Range
  - If-Unmodified-Since
  - Max-Forwards
  - Proxy-Authorization
  - Range
  - Referer
  - TE
  - User-Agent
- 响应标头
  - Accept-Ranges
  - Age
  - ETag
  - Location
  - Proxy-Authenticate
  - Retry-After
  - Server
  - Vary
  - www-Authenticate
  - Access-Control-Allow-Origin
- 实体标头
  - Allow
  - Content-Encoding
  - Content-Language
  - Content-Length
  - Content-Location
  - Content-MD5
  - Content-Range
  - Content-Type
  - Expires
  - Last-Modified

HTTP 内容协商

- 什么是内容协商
- 内容协商的种类
- 为什么需要内容协商
- 内容协商标头

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

HTTP 认证

- 通用 HTTP 认证框架
- 代理认证
  - Proxy-Authenticate
  - Proxy-Authorization
- 禁止访问
  - WWW-Authenticate 和 Proxy-Authenticate 头
  - Authorization 和 Proxy-Authorization 标头

HTTP 缓存

- 不同类型的缓存
  - 不缓存过期资源
  - 私有缓存
  - 共享缓存
- 缓存控制
  - 不缓存
  - 缓存但需要验证
  - 私有和共享缓存
  - 缓存过期
  - 缓存验证
- 什么是新鲜的数据
- 缓存验证
  - Etag
  - 避免碰撞
  - 缓存未占用资源

HTTP CROS 跨域

- Origin
- 跨域的特点
- 同源策略
- 跨域请求
- 跨域功能概述
- 访问控制
  - 简单请求
  - 预检请求
  - 带凭证的请求
- HTTP 响应标头
  - Access-Control-Allow-Origin
  - Access-Control-Allow-Credentials
  - Access-Control-Allow-Headers
  - Access-Control-Allow-Methods
  - Access-Control-Expose-Headers

- Access-Control-Max-Age
- Access-Control-Request-Headers
- Access-Control-Request-Method
- Origin

HTTP 条件请求

- 原则
- 验证
  - 强验证
  - 弱验证
- 条件请求
  - If-Match
  - If-None-Match
  - If-Modified-Since
  - If-Range
  - If-Unmodified-Since
- 条件请求示例
  - 缓存更新
  - 断点续传
  - 通过乐观锁避免丢失更新

HTTP Cookies

- 创建 Cookie
  - Set-Cookie 和 Cookie 标头
  - 会话 Cookies
  - 永久性 Cookies
  - Cookie的 Secure 和 HttpOnly 标记
- Cookie 的作用域

HTTP 的优点和缺点

- HTTP 的优点
  - 简单灵活易扩展
  - 应用广泛、环境成熟
  - 无状态
- HTTP 的缺点
  - 无状态
  - 明文
  - 性能

HTTPS 为什么会出现

- HTTPS 解决了什么问题
- 什么是 HTTPS
- HTTPS 做了什么

什么是 SSL/TLS

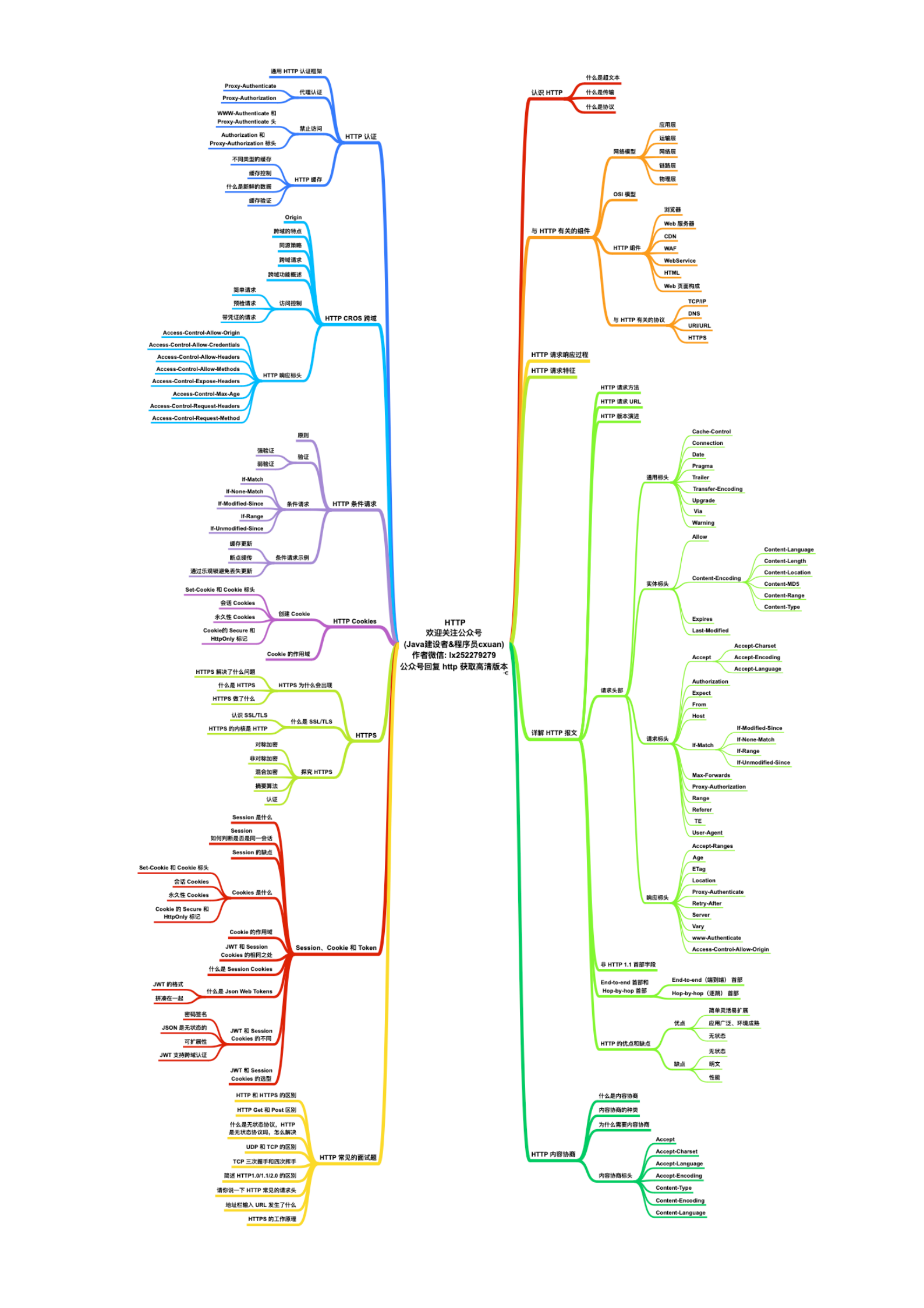
- 认识 SSL/TLS
- HTTPS 的内核是 HTTP

探究 HTTPS

- 对称加密
  - 加密分组
- 非对称加密
- 混合加密

- 摘要算法
- 认证
- Cookie 和 Session
  - Session 是什么
  - Session 如何判断是否是同一会话
  - Session 的缺点
  - Cookies 是什么
  - 创建 Cookie
    - Set-Cookie 和 Cookie 标头
    - 会话 Cookies
    - 永久性 Cookies
    - Cookie 的 Secure 和 HttpOnly 标记
  - Cookie 的作用域
- JSON Web Token 和 Session Cookies 的对比
  - JWT 和 Session Cookies 的相同之处
  - 什么是 Session Cookies
  - 什么是 Json Web Tokens
    - JWT 的格式
    - 拼凑在一起
  - JWT 和 Session Cookies 的不同
    - 密码签名
    - JSON 是无状态的
    - 可扩展性
    - JWT 支持跨域认证
  - JWT 和 Session Cookies 的选型
- 后记
- HTTP 和 HTTPS 的区别
- HTTP Get 和 Post 区别
- 什么是无状态协议，HTTP 是无状态协议吗，怎么解决
- UDP 和 TCP 的区别
  - UDP 是什么
  - TCP 是什么
  - TCP 和 UDP 的不同
- TCP 三次握手和四次挥手
  - TCP 三次握手
  - TCP 四次挥手
- 请你说一下 HTTP 常见的请求头
  - 通用标头
  - 实体标头
  - 请求标头
  - 响应标头
- 地址栏输入 URL 发生了什么
- HTTPS 的工作原理

HTTP 超全思维导图，涉及 HTTP 基本认识、HTTP 请求过程、HTTP 响应标头、HTTPS 的出现原因、解决什么问题、总结了 Session、Cookie 和 Token，最后再为你整理了 HTTP 的核心面试题。


HTTP
欢迎关注公众号
(Java建设者&程序员cxuan)
作者微信: lx252279279
公众号回复 http 获取高清版本
认识 HTTP
什么是超文本
什么是传输
什么是协议
与 HTTP 有关的组件
网络模型
应用层
运输层
网络层
链路层
物理层
OSI 模型
HTTP 组件
浏览器
Web 服务器
CDN
WAF
WebService
HTML
Web 页面构成
与 HTTP 有关的协议
TCP/IP
DNS
URI/URL
HTTPS
HTTP 请求响应过程
HTTP 请求特征
详解 HTTP 报文
请求头部
HTTP 请求方法
HTTP 请求 URL
HTTP 版本演进
通用标头
Cache-Control
Connection
Date
Pragma
Trailer
Transfer-Encoding
Upgrade
Via
Warning
实体标头
Allow
Content-Encoding
Content-Language
Content-Length
Content-Location
Content-MD5
Content-Range
Content-Type
Expires
Last-Modified
请求标头
Accept
Accept-Charset
Accept-Encoding
Accept-Language
Authorization
Expect
From
Host
If-Match
If-Modified-Since
If-None-Match
If-Range
If-Unmodified-Since
Max-Forwards
Proxy-Authorization
Range
Referer
TE
User-Agent
响应标头
Accept-Ranges
Age
ETag
Location
Proxy-Authenticate
Retry-After
Server
Vary
www-Authenticate
Access-Control-Allow-Origin
非 HTTP 1.1 首部字段
End-to-end 首部和 Hop-by-hop 首部
End-to-end（端到端） 首部
Hop-by-hop（逐跳） 首部
HTTP 的优点和缺点
优点
简单灵活易扩展
应用广泛、环境成熟
无状态
缺点
无状态
明文
性能
HTTP 内容协商
什么是内容协商
内容协商的种类
为什么需要内容协商
内容协商标头
Accept
Accept-Charset
Accept-Language
Accept-Encoding
Content-Type
Content-Encoding
Content-Language
HTTP 认证
通用 HTTP 认证框架
代理认证
Proxy-Authenticate
Proxy-Authorization
禁止访问
WWW-Authenticate 和 Proxy-Authenticate 头
Authorization 和 Proxy-Authorization 标头
HTTP 缓存
不同类型的缓存
缓存控制
什么是新鲜的数据
缓存验证
HTTP CROS 跨域
Origin
跨域的特点
同源策略
跨域请求
跨域功能概述
访问控制
简单请求
预检请求
带凭证的请求
HTTP 响应标头
Access-Control-Allow-Origin
Access-Control-Allow-Credentials
Access-Control-Allow-Headers
Access-Control-Allow-Methods
Access-Control-Expose-Headers
Access-Control-Max-Age
Access-Control-Request-Headers
Access-Control-Request-Method
HTTP 条件请求
原则
验证
强验证
弱验证
条件请求
If-Match
If-None-Match
If-Modified-Since
If-Range
If-Unmodified-Since
条件请求示例
缓存更新
断点续传
通过乐观锁避免丢失更新
HTTP Cookies
创建 Cookie
Set-Cookie 和 Cookie 标头
会话 Cookies
永久性 Cookies
Cookie的 Secure 和 HttpOnly 标记
Cookie 的作用域
HTTPS
HTTPS 为什么会出现
HTTPS 解决了什么问题
什么是 HTTPS
HTTPS 做了什么
什么是 SSL/TLS
认识 SSL/TLS
HTTPS 的内核是 HTTP
探究 HTTPS
对称加密
非对称加密
混合加密
摘要算法
认证
Session、Cookie 和 Token
Session 是什么
Session
如何判断是否是同一会话
Session 的缺点
Cookies 是什么
Set-Cookie 和 Cookie 标头
会话 Cookies
永久性 Cookies
Cookie 的 Secure 和 HttpOnly 标记
Cookie 的作用域
JWT 和 Session Cookies 的相同之处
什么是 Session Cookies
什么是 Json Web Tokens
JWT 的格式
拼凑在一起
JWT 和 Session Cookies 的不同
密码签名
JSON 是无状态的
可扩展性
JWT 支持跨域认证
JWT 和 Session Cookies 的选型
HTTP 常见的面试题
HTTP 和 HTTPS 的区别
HTTP Get 和 Post 区别
什么是无状态协议，HTTP 是无状态协议吗，怎么解决
UDP 和 TCP 的区别
TCP 三次握手和四次挥手
简述 HTTP1.0/1.1/2.0 的区别
请你说一下 HTTP 常见的请求头
地址栏输入 URL 发生了什么
HTTPS 的工作原理

认识 HTTP

首先你听的最多的应该就是 HTTP 是一种 `超文本传输协议(Hypertext Transfer Protocol)`，这你一定能说出来，但是这样还不够，假如你是大厂面试官，这不可能是他想要的最终结果，我们在面试的时候往往把自己知道的尽可能多的说出来，才有和面试官谈价钱的资本。那么什么是超文本传输协议?

超文本传输协议可以进行文字分割：**超文本（Hypertext）**、**传输（Transfer）**、**协议（Protocol）**，它们之间的关系如下

按照范围的大小 协议 > 传输 > 超文本。下面就分别对这三个名次做一个解释。

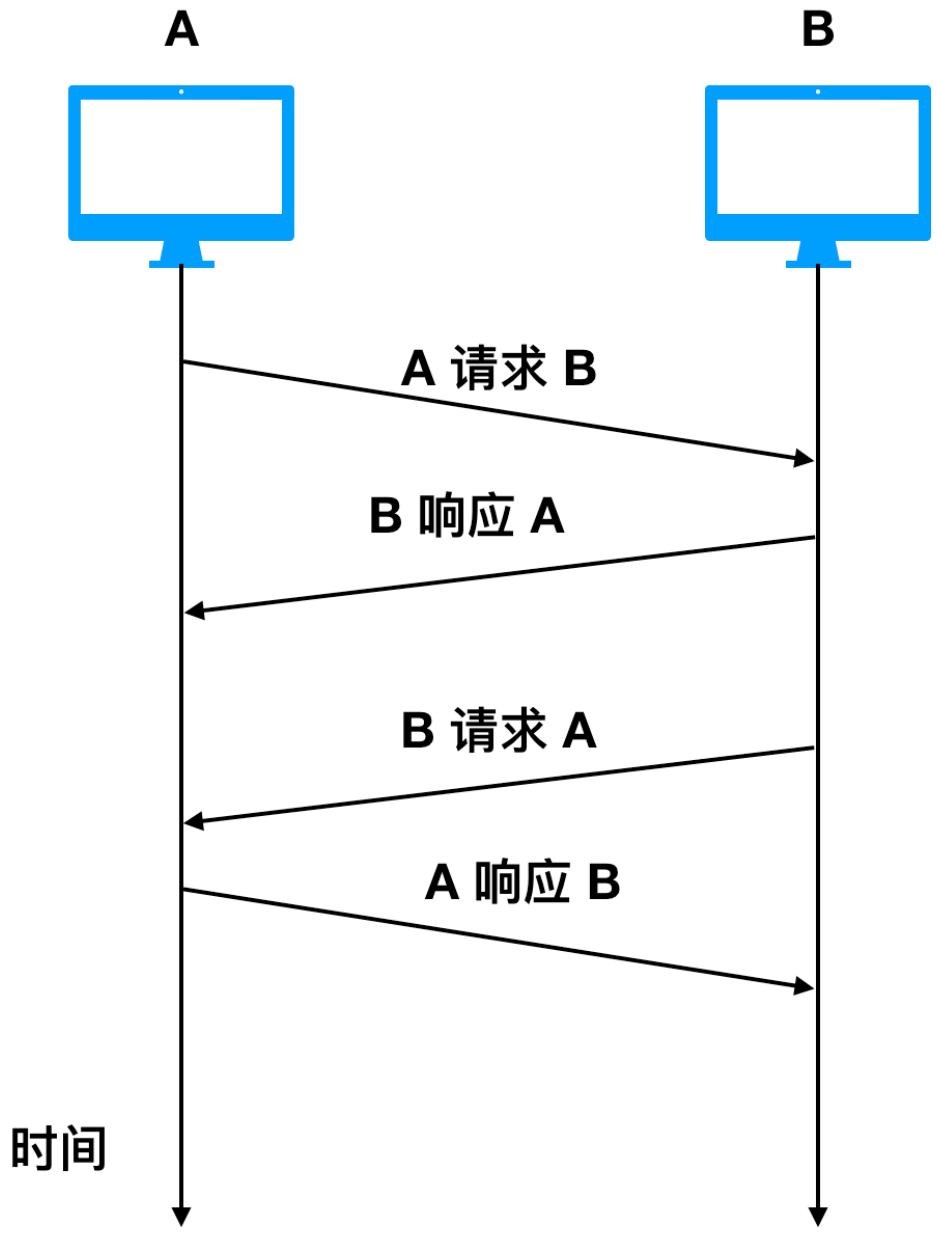
什么是超文本

在互联网早期的时候，我们输入的信息只能保存在本地，无法和其他电脑进行交互。我们保存的信息通常都以 `文本` 即简单字符的形式存在，文本是一种能够被计算机解析的有意义的二进制数据包。而随着互联网的高速发展，两台电脑之间能够进行数据的传输后，人们不满足只能在两台电脑之间传输文字，还想要传输图片、音频、视频，甚至点击文字或图片能够进行 `超链接` 的跳转，那么文本的语义就被扩大了，这种语义扩大后的文本就被称为 `超文本(Hypertext)` 。

什么是传输

那么我们上面说到，两台计算机之间会形成互联关系进行通信，我们存储的超文本会被解析成为二进制数据包，由传输载体（例如同轴电缆，电话线，光缆）负责把二进制数据包由计算机终端传输到另一个终端的过程（对终端的详细解释可以参考 **你说你懂互联网，那这些你知道么？** 这篇文章）称为 `传输(transfer)` 。

通常我们把传输数据包的一方称为 `请求方` ，把接到二进制数据包的一方称为 `应答方` 。请求方和应答方可以进行互换，请求方也可以作为应答方接受数据，应答方也可以作为请求方请求数据，它们之间的关系如下



如图所示，A 和 B 是两个不同的端系统，它们之间可以作为信息交换的载体存在，刚开始的时候是 A 作为请求方请求与 B 交换信息，B 作为响应的一方提供信息；随着时间的推移，B 也可以作为请求方请求 A 交换信息，那么 A 也可以作为响应方响应 B 请求的信息。

什么是协议

协议这个名词不仅局限于互联网范畴，也体现在日常生活中，比如情侣双方约定好在哪个地点吃饭，这个约定也是一种 协议 ，比如你应聘成功了，企业会和你签订劳动合同，这种双方的雇佣关系也是一种 协议 。注意自己一个人对自己的约定不能成为协议，协议的前提条件必须是多人约定。

那么网络协议是什么呢?

网络协议就是网络中(包括互联网)传递、管理信息的一些规范。如同人与人之间相互交流是需要遵循一定的规矩一样，计算机之间的相互通信需要共同遵守一定的规则，这些规则就称为网络协议。

没有网络协议的互联网是混乱的，就和人类社会一样，人不能想怎么样就怎么样，你的行为约束是受到法律的约束的；那么互联网中的端系统也不能自己想发什么发什么，也是需要受到通信协议约束的。

那么我们就可以总结一下，什么是 HTTP？可以用下面这个经典的总结回答一下： **HTTP 是一个在计算机世界里专门在两点之间传输文字、图片、音频、视频等超文本数据的约定和规范**

与 HTTP 有关的组件

随着网络世界演进，HTTP 协议已经几乎成为不可替代的一种协议，在了解了 HTTP 的基本组成后，下面再来带你进一步认识一下 HTTP 协议。

网络模型

网络是一个复杂的系统，不仅包括大量的应用程序、端系统、通信链路、分组交换机等，还有各种各样的协议组成，那么现在我们就来聊一下网络中的协议层次。

为了给网络协议的设计提供一个结构，网络设计者以 `分层(layer)` 的方式组织协议，每个协议属于层次模型之一。每一层都是向它的上一层提供 `服务(service)` ，即所谓的 `服务模型(service model)` 。每个分层中所有的协议称为 `协议栈(protocol stack)` 。因特网的协议栈由五个部分组成：物理层、链路层、网络层、运输层和应用层。我们采用自上而下的方法研究其原理，也就是应用层 -> 物理层的方式。

应用层

应用层是网络应用程序和网络协议存放的分层，因特网的应用层包括许多协议，例如我们学 web 离不开的 `HTTP` ，电子邮件传送协议 `SMTP` 、端系统文件上传协议 `FTP` 、还有为我们进行域名解析的 `DNS` 协议。应用层协议分布在多个端系统上，一个端系统应用程序与另外一个端系统应用程序交换信息分组，我们把位于应用层的信息分组称为 `报文(message)` 。

运输层

因特网的运输层在应用程序断点之间传送应用程序报文，在这一层主要有两种传输协议 `TCP` 和 `UDP` ，利用这两者中的任何一个都能够传输报文，不过这两种协议有巨大的不同。

TCP 向它的应用程序提供了面向连接的服务，它能够控制并确认报文是否到达，并提供了拥塞机制来控制网络传输，因此当网络拥塞时，会抑制其传输速率。

UDP 协议向它的应用程序提供了无连接服务。它不具备可靠性的特征，没有流量控制，也没有拥塞控制。我们把运输层的分组称为 `报文段(segment)`

网络层

因特网的网络层负责将称为 `数据报(datagram)` 的网络分层从一台主机移动到另一台主机。网络层一个非常重要的协议是 `IP` 协议，所有具有网络层的因特网组件都必须运行 IP 协议，IP 协议是一种网际协议，除了 IP 协议外，网络层还包括一些其他网际协议和路由选择协议，一般把网络层就称为 IP 层，由此可知 IP 协议的重要性。

链路层

现在我们有应用程序通信的协议，有了给应用程序提供运输的协议，还有了用于约定发送位置的 IP 协议，那么如何才能真正的发送数据呢？为了将分组从一个节点（主机或路由器）运输到另一个节点，网络层必须依靠链路层提供服务。链路层的例子包括以太网、WiFi 和电缆接入的 `DOCSIS` 协议，因为数据从源目的地传送通常需要经过几条链路，一个数据包可能被沿途不同的链路层协议处理，我们把链路层的分组称为 `帧(frame)`

物理层

虽然链路层的作用是将帧从一个端系统运输到另一个端系统，而物理层的作用是将帧中的一个个 `比特` 从一个节点运输到另一个节点，物理层的协议仍然使用链路层协议，这些协议与实际的物理传输介质有关，例如，以太网有很多物理层协议：关于双绞铜线、关于同轴电缆、关于光纤等等。

五层网络协议的示意图如下

OSI 模型

我们上面讨论的计算网络协议模型不是唯一的 `协议栈`，ISO（国际标准化组织）提出来计算机网络应该按照7层来组织，那么7层网络协议栈与5层的区别在哪里？

从图中可以一眼看出，OSI 要比上面的网络模型多了 `表示层` 和 `会话层`，其他层基本一致。表示层主要包括数据压缩和数据加密以及数据描述，数据描述使得应用程序不必担心计算机内部存储格式的问题，而会话层提供了数据交换的定界和同步功能，包括建立检查点和恢复方案。

浏览器

就如同各大邮箱使用电子邮件传送协议 `SMTP` 一样，浏览器是使用 HTTP 协议的主要载体，说到浏览器，你能想起来几种？是的，随着网景大战结束后，浏览器迅速发展，至今已经出现过的浏览器主要有

浏览器正式的名字叫做 `Web Broser`，顾名思义，就是检索、查看互联网上网页资源的应用程序，名字里的 Web，实际上指的就是 `World Wide Web`，也就是万维网。

我们在地址栏输入URL（即网址），浏览器会向DNS（域名服务器，后面会说）提供网址，由它来完成 URL 到 IP 地址的映射。然后将请求你的请求提交给具体的服务器，在由服务器返回我们要的结果（以HTML编码格式返回给浏览器），浏览器执行HTML编码，将结果显示在浏览器的正文。这就是一个浏览器发起请求和接受响应的过程。

Web 服务器

Web 服务器的正式名称叫做 `Web Server`，Web 服务器一般指的是网站服务器，上面说到浏览器是 HTTP 请求的发起方，那么 Web 服务器就是 HTTP 请求的应答方，Web 服务器可以向浏览器等 Web 客户端提供文档，也可以放置网站文件，让全世界浏览；可以放置数据文件，让全世界下载。目前最主流的三个Web服务器是Apache、 Nginx 、IIS。

CDN

CDN的全称是 `Content Delivery Network`，即 `内容分发网络`，它应用了 HTTP 协议里的缓存和代理技术，代替源站响应客户端的请求。CDN 是构建在现有网络基础之上的网络，它依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户 `就近` 获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。CDN的关键技术主要有 `内容存储` 和 `分发技术` 。

打比方说你要去亚马逊上买书，之前你只能通过购物网站购买后从美国发货过海关等重重关卡送到你的家里，现在在中国建立一个亚马逊分基地，你就不用通过美国进行邮寄，从中国就能把书尽快给你送到。

WAF

WAF 是一种 Web 应用程序防护系统（Web Application Firewall，简称 WAF），它是一种通过执行一系列针对HTTP / HTTPS的 `安全策略` 来专门为Web应用提供保护的一款产品，它是应用层面的 `防火墙`，专门检测 HTTP 流量，是防护 Web 应用的安全技术。

WAF 通常位于 Web 服务器之前，可以阻止如 SQL 注入、跨站脚本等攻击，目前应用较多的一个开源项目是 ModSecurity，它能够完全集成进 Apache 或 Nginx。

WebService

WebService 是一种 Web 应用程序，**WebService是一种跨编程语言和跨操作系统平台的远程调用技术**。

Web Service 是一种由 W3C 定义的应用服务开发规范，使用 client-server 主从架构，通常使用 WSDL 定义服务接口，使用 HTTP 协议传输 XML 或 SOAP 消息，它是**一个基于 Web（HTTP）的服务架构技术**，既可以运行在内网，也可以在适当保护后运行在外网。

HTML

HTML 称为超文本标记语言，是一种标识性的语言。它包括一系列标签．通过这些标签可以将网络上的文档格式统一，使分散的 Internet 资源连接为一个逻辑整体。HTML 文本是由 HTML 命令组成的描述性文本，HTML 命令可以说明文字，图形、动画、声音、表格、链接等。

Web 页面构成

Web 页面（Web page）也叫做文档，是由一个个对象组成的。一个 `对象(Objecy)` 只是一个文件，比如一个 HTML 文件、一个 JPEG 图形、一个 Java 小程序或一个视频片段，它们在网络中可以通过 `URL` 地址寻址。多数的 Web 页面含有一个 `HTML 基本文件` 以及几个引用对象。

举个例子，如果一个 Web 页面包含 HTML 文件和5个 JPEG 图形，那么这个 Web 页面就有6个对象：一个 HTML 文件和5个 JPEG 图形。HTML 基本文件通过 URL 地址引用页面中的其他对象。

与 HTTP 有关的协议

在互联网中，任何协议都不会单独的完成信息交换，HTTP 也一样。虽然 HTTP 属于应用层的协议，但是它仍然需要其他层次协议的配合完成信息的交换，那么在完成一次 HTTP 请求和响应的过程中，需要哪些协议的配合呢？一起来看一下

TCP/IP

`TCP/IP` 协议你一定听过，TCP/IP 我们一般称之为 `协议簇`，什么意思呢？就是 TCP/IP 协议簇中不仅仅只有 TCP 协议和 IP 协议，它是一系列网络通信协议的统称。而其中最核心的两个协议就是 TCP / IP 协议，其他的还有 UDP、ICMP、ARP 等等，共同构成了一个复杂但有层次的协议栈。

TCP 协议的全称是 `Transmission Control Protocol` 的缩写，意思是 `传输控制协议`，HTTP 使用 TCP 作为通信协议，这是因为 TCP 是一种可靠的协议，而 `可靠` 能保证数据不丢失。

IP 协议的全称是 `Internet Protocol` 的缩写，它主要解决的是通信双方寻址的问题。IP 协议使用 `IP 地址` 来标识互联网上的每一台计算机，可以把 IP 地址想象成为你手机的电话号码，你要与他人通话必须先要知道他人的手机号码，计算机网络中信息交换必须先要知道对方的 IP 地址。（关于 TCP 和 IP 更多的讨论我们会在后面详解）

DNS

你有没有想过为什么你可以通过键入 `www.google.com` 就能够获取你想要的网站？我们上面说到，计算机网络中的每个端系统都有一个 IP 地址存在，而把 IP 地址转换为便于人类记忆的协议就是 `DNS 协议` 。

DNS 的全称是 `域名系统（Domain Name System，缩写：DNS）` ，它作为将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便地访问互联网。

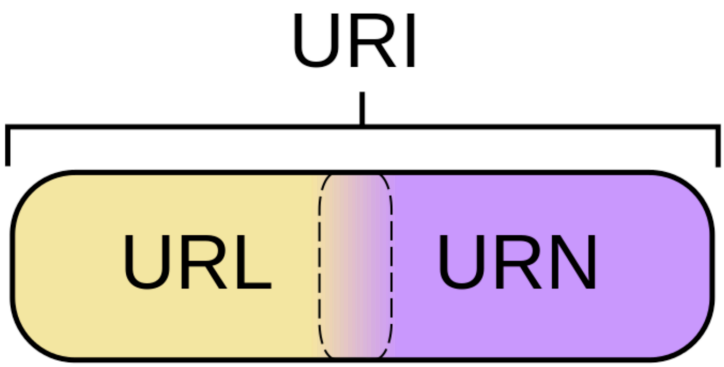
URI / URL

我们上面提到，你可以通过输入 `www.google.com` 地址来访问谷歌的官网，那么这个地址有什么规定吗？我怎么输都可以？AAA.BBB.CCC 是不是也行？当然不是的，你输入的地址格式必须要满足 `URI` 的规范。

`URI` 的全称是（Uniform Resource Identifier），中文名称是统一资源标识符，使用它就能够唯一地标记互联网上资源。

`URL` 的全称是（Uniform Resource Locator），中文名称是统一资源定位符，也就是我们俗称的 `网址` ，它实际上是 URI 的一个子集。

URI 不仅包括 URL，还包括 URN（统一资源名称），它们之间的关系如下



HTTPS

HTTP 一般是明文传输，很容易被攻击者窃取重要信息，鉴于此，HTTPS 应运而生。HTTPS 的全称为（Hyper Text Transfer Protocol over SecureSocket Layer），全称有点长，HTTPS 和 HTTP 有很大的不同在于 HTTPS 是以安全为目标的 HTTP 通道，在 HTTP 的基础上通过传输加密和身份认证保证了传输过程的安全性。HTTPS 在 HTTP 的基础上增加了 `SSL` 层，也就是说 **HTTPS = HTTP + SSL**。（这块我们后面也会详谈 HTTPS）

HTTP 请求响应过程

你是不是很好奇，当你在浏览器中输入网址后，到底发生了什么事情？你想要的内容是如何展现出来的？让我们通过一个例子来探讨一下，我们假设访问的 URL 地址为 `http://www.someSchool.edu/someDepartment/home.index`，当我们输入网址并点击回车时，浏览器内部会进行如下操作

- DNS服务器会首先进行域名的映射，找到访问 `www.someSchool.edu` 所在的地址，然后HTTP 客户端进程在 80 端口发起一个到服务器 `www.someSchool.edu` 的 TCP 连接（80 端口是 HTTP 的默认端口）。在客户和服务器进程中都会有一个 `套接字` 与其相连。
- HTTP 客户端通过它的套接字向服务器发送一个 HTTP 请求报文。该报文中包含了路径 `someDepartment/home.index` 的资源，我们后面会详细讨论 HTTP 请求报文。
- HTTP 服务器通过它的套接字接受该报文，进行请求的解析工作，并从其 `存储器(RAM 或磁盘)` 中检索出对象 **www.someSchool.edu/someDepartment/home.index**，然后把检索出来的对象进行封装，封装到 HTTP 响应报文中，并通过套接字向客户进行发送。
- HTTP 服务器随即通知 TCP 断开 TCP 连接，实际上是需要等到客户接受完响应报文后才会断开 TCP 连接。
- HTTP 客户端接受完响应报文后，TCP 连接会关闭。HTTP 客户端从响应中提取出报文中是一个 HTML 响应文件，并检查该 HTML 文件，然后循环检查报文中其他内部对象。
- 检查完成后，HTTP 客户端会把对应的资源通过显示器呈现给用户。

至此，键入网址再按下回车的全过程就结束了。上述过程描述的是一种简单的 `请求-响应` 全过程，真实的请求-响应情况可能要比上面描述的过程复杂很多。

HTTP 请求特征

从上面整个过程中我们可以总结出 HTTP 进行分组传输是具有以下特征

- 支持客户-服务器模式
- 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。
- 灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。
- 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。
- 无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

详解 HTTP 报文

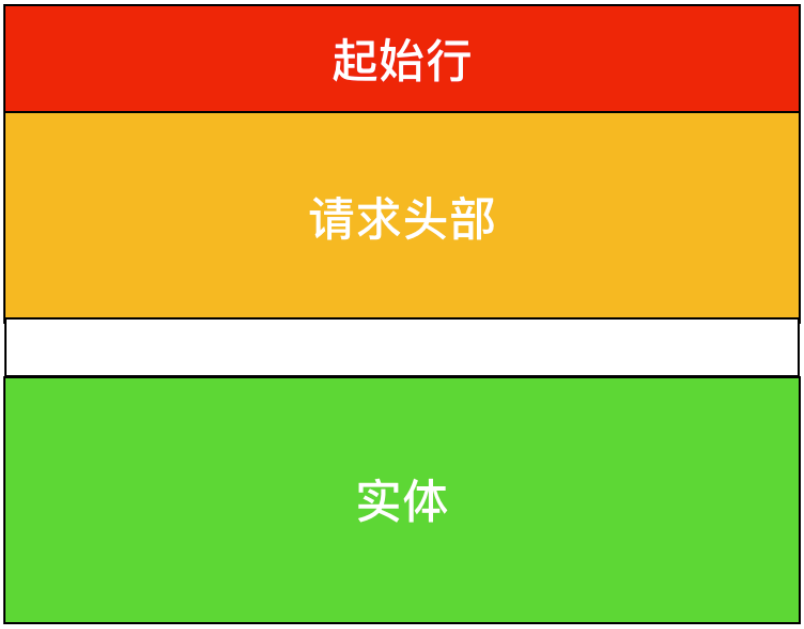
我们上面描述了一下 HTTP 的请求响应过程，流程比较简单，但是凡事就怕认真，你这一认真，就能拓展出很多东西，比如 **HTTP 报文是什么样的，它的组成格式是什么？** 下面就来探讨一下

HTTP 协议主要由三大部分组成：

- `起始行（start line）`：描述请求或响应的基本信息；
- `头部字段（header）`：使用 key-value 形式更详细地说明报文；

- `消息正文（entity）`：实际传输的数据，它不一定是纯文本，可以是图片、视频等二进制数据。

其中起始行和头部字段并成为 `请求头` 或者 `响应头`，统称为 `Header`；消息正文也叫做实体，称为 `body`。HTTP 协议规定每次发送的报文必须要有 Header，但是可以没有 body，也就是说头信息是必须的，实体信息可以没有。而且在 header 和 body 之间必须要有一个空行（CRLF），如果用一幅图来表示一下的话，我觉得应该是下面这样



我们使用上面的那个例子来看一下 http 的请求报文

如图，这是 `http://www.someSchool.edu/someDepartment/home.index` 请求的请求头，通过观察这个 HTTP 报文我们就能够学到很多东西，首先，我们看到报文是用普通 `ASCII` 文本书写的，这样保证人能够可以看懂。然后，我们可以看到每一行和下一行之间都会有换行，而且最后一行（请求头部后）再加上一个回车换行符。

每个报文的起始行都是由三个字段组成：**方法、URL 字段和 HTTP 版本字段**。

HTTP 请求方法

HTTP 请求方法一般分为 8 种，它们分别是

- `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；
- `POST 传输实体`，虽然 GET 方法也可以传输主体信息，但是便于区分，我们一般不用 GET 传输实体信息，反而使用 POST 传输实体信息，
- 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。

  但是，鉴于 HTTP 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，因此一般的 W eb 网站不使用该方法。若配合 W eb 应用程序的验证机制，或架构设计采用 `REST（REpresentational State Transfer，表征状态转移）` 标准的同类 Web 网站，就可能会开放使用 PUT 方法。
- HEAD 获得响应首部，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 的有效性及资源更新的日期时间等。
- 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。
- OPTIONS 询问支持的方法，OPTIONS 方法用来查询针对请求 URI 指定的资源支持的方法。
- TRACE 追踪路径，TRACE 方法是让 Web 服务器端将之前的请求通信环回给客户端的方法。
- 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。

我们一般最常用的方法也就是 GET 方法和 POST 方法，其他方法暂时了解即可。下面是 HTTP1.0 和 HTTP1.1 支持的方法清单

方法	说明	支持的 HTTP 协议版本
GET	获取资源	1.0、1.1
POST	传输实体主体	1.0、1.1
PUT	传输文件	1.0、1.1
HEAD	获得报文首部	1.0、1.1
DELETE	删除文件	1.0、1.1
OPTIONS	询问支持的方法	1.1
TRACE	追踪路径	1.1
CONNECT	要求用隧道协议连接代理	1.1
LINK	建立和资源之间的联系	1.0
UNLINE	断开连接关系	1.0

HTTP 请求 URL

HTTP 协议使用 URI 定位互联网上的资源。正是因为 URI 的特定功能，在互联网上任意位置的资源都能访问到。URL 带有请求对象的标识符。在上面的例子中，浏览器正在请求对象 `/somedir/page.html` 的资源。

我们再通过一个完整的域名解析一下 URL

比如 `http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2#SomewhereInTheDocument` 这个 URL 比较繁琐了吧，你把这个 URL 搞懂了其他的 URL 也就不成问题了。

首先出场的是 `http`

`http://` 告诉浏览器使用何种协议。对于大部分 Web 资源，通常使用 HTTP 协议或其安全版本，HTTPS 协议。另外，浏览器也知道如何处理其他协议。例如， `mailto:` 协议指示浏览器打开邮件客户端； `ftp:` 协议指示浏览器处理文件传输。

第二个出场的是 `主机`

`www.example.com` 既是一个域名，也代表管理该域名的机构。它指示了需要向网络上的哪一台主机发起请求。当然，也可以直接向主机的 **IP address** 地址发起请求。但直接使用 IP 地址的场景并不常见。

第三个出场的是 `端口`

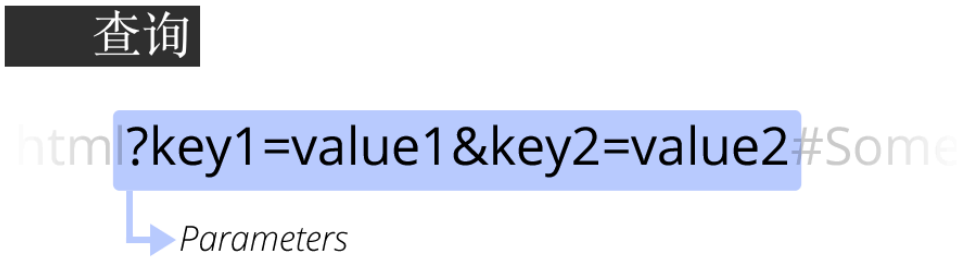
我们前面说到，两个主机之间要发起 TCP 连接需要两个条件，主机 + 端口。它表示用于访问 Web 服务器上资源的入口。如果访问的该 Web 服务器使用HTTP协议的标准端口（HTTP为80，HTTPS为443）授予对其资源的访问权限，则通常省略此部分。否则端口就是 URI 必须的部分。

上面是请求 URL 所必须包含的部分，下面就是 URL 具体请求资源路径

第四个出场的是 `路径`

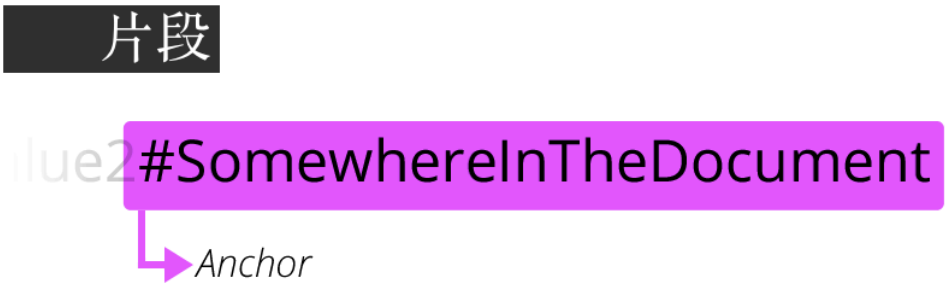
`/path/to/myfile.html` 是 Web 服务器上资源的路径。以端口后面的第一个 `/` 开始，到 `?` 号之前结束，中间的 每一个 `/` 都代表了层级（上下级）关系。这个 URL 的请求资源是一个 html 页面。

紧跟着路径后面的是 `查询参数`



`?key1=value1&key2=value2` 是提供给 Web 服务器的额外参数。如果是 GET 请求，一般带有请求 URL 参数，如果是 POST 请求，则不会在路径后面直接加参数。这些参数是用 & 符号分隔的 `键/值对` 列表。key1 = value1 是第一对，key2 = value2 是第二对参数

紧跟着参数的是 `锚点`



`#SomewhereInTheDocument` 是资源本身的某一部分的一个锚点。锚点代表资源内的一种“书签”，它给予浏览器显示位于该“加书签”点的内容的指示。 例如，在HTML文档上，浏览器将滚动到定义锚点的那个点上；在视频或音频文档上，浏览器将转到锚点代表的那个时间。值得注意的是 # 号后面的部分，也称为片段标识符，永远不会与请求一起发送到服务器。

HTTP 版本

HTTP 1.0

HTTP 1.0 是在 1996 年引入的，从那时开始，它的普及率就达到了惊人的效果。

- HTTP 1.0 仅仅提供了最基本的认证，这时候用户名和密码还未经加密，因此很容易收到窥探。
- HTTP 1.0 被设计用来使用短链接，即每次发送数据都会经过 TCP 的三次握手和四次挥手，效率比较低。
- HTTP 1.0 只使用 header 中的 If-Modified-Since 和 Expires 作为缓存失效的标准。
- HTTP 1.0 不支持断点续传，也就是说，每次都会传送全部的页面和数据。
- HTTP 1.0 认为每台计算机只能绑定一个 IP，所以请求消息中的 URL 并没有传递主机名（hostname）。

HTTP 1.1

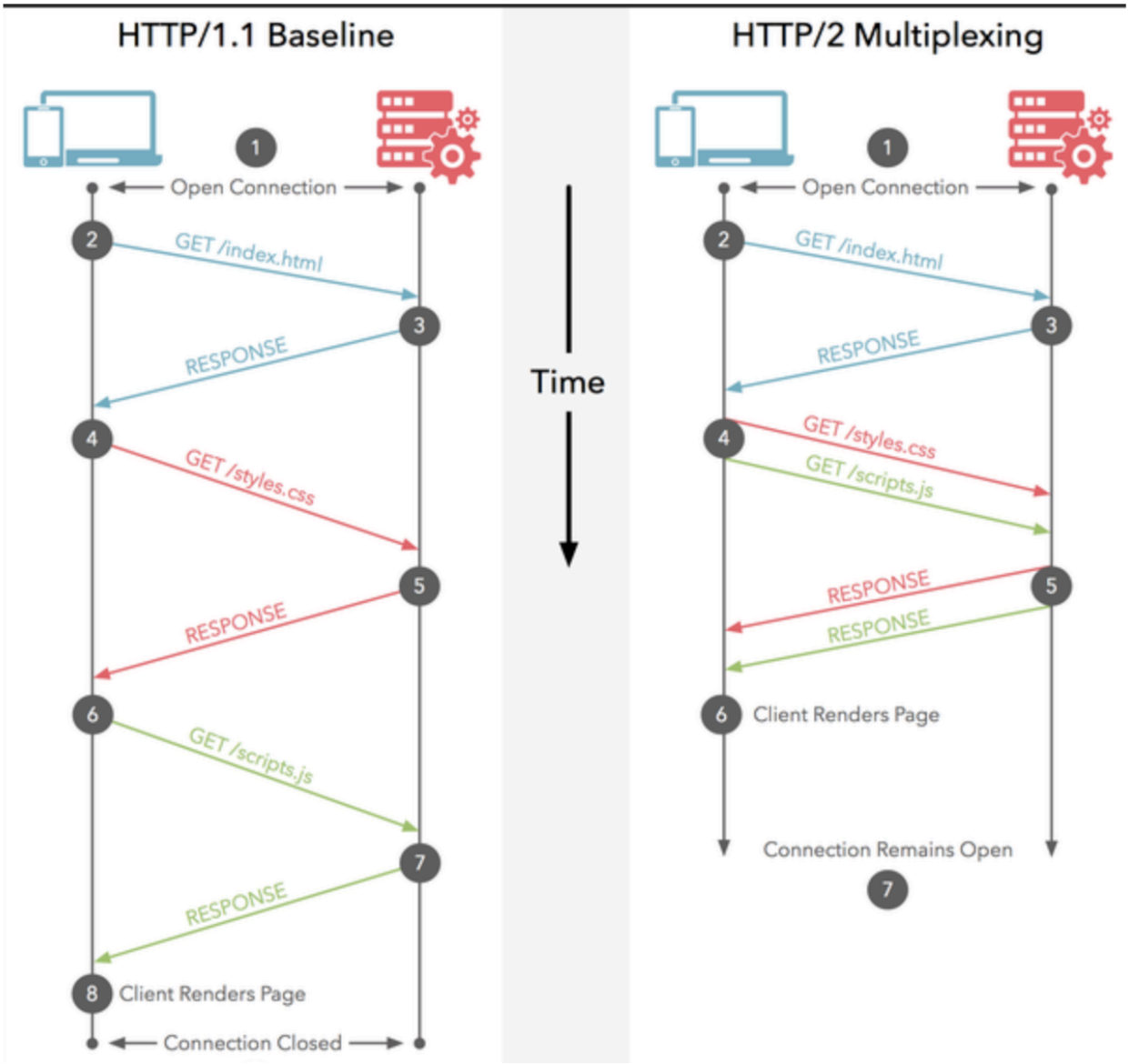
HTTP 1.1 是 HTTP 1.0 开发三年后出现的，也就是 1999 年，它做出了以下方面的变化

- HTTP 1.1 使用了摘要算法来进行身份验证
- HTTP 1.1 默认使用长连接，长连接就是只需一次建立就可以传输多次数据，传输完成后，只需要一次切断连接即可。长连接的连接时长可以通过请求头中的 `keep-alive` 来设置
- HTTP 1.1 中新增加了 E-tag，If-Unmodified-Since, If-Match, If-None-Match 等缓存控制标头来控制缓存失效。
- HTTP 1.1 支持断点续传，通过使用请求头中的 `Range` 来实现。
- HTTP 1.1 使用了虚拟网络，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。

HTTP 2.0

HTTP 2.0 是 2015 年开发出来的标准，它主要做的改变如下

- `头部压缩`，由于 HTTP 1.1 经常会出现 **User-Agent**、**Cookie**、**Accept**、**Server**、**Range** 等字段可能会占用几百甚至几千字节，而 Body 却经常只有几十字节，所以导致头部偏重。HTTP 2.0 使用 `HPACK` 算法进行压缩。
- `二进制格式`，HTTP 2.0 使用了更加靠近 TCP/IP 的二进制格式，而抛弃了 ASCII 码，提升了解析效率
- `强化安全`，由于安全已经成为重中之重，所以 HTTP2.0 一般都跑在 HTTPS 上。
- `多路复用`，即每一个请求都是是用作连接共享。一个请求对应一个id，这样一个连接上可以有多个请求。



HTTP 标头

HTTP 1.1 的标头主要分为四种， 通用标头 、 实体标头 、 请求标头 、 响应标头 ，现在我们来对这几种标头进行介绍

通用标头

HTTP 通用标头之所以这样命名，是因为与其他三个类别不同，它们不是限定于特定种类的消息或者消息组件（请求，响应或消息实体）的。HTTP 通用标头主要用于传达有关消息本身的信息，而不是它所携带的内容。它们提供一般信息并控制如何处理和处理消息。

尽管通用标头不会限定于是请求还是响应报文，但是某些通用标头大部分或全部用于一种特定类型的请求中。也就是说，如果某个通用标头出现在请求报文中，那么大部分通用标头都会显示在该请求报文中。响应报文也是一样的。

先列出来一个清单，讲明我们都需要介绍哪些通用标头

- Cache-Control

- Connection
- Date
- Pragma
- Trailer
- Transfer-Encoding
- Upgrade
- Via
- Warning

Cache-Control

`缓存（Cache）`是计算机领域里的一个重要概念，是优化系统性能的利器。不仅计算机中的 CPU 为了提高指令执行效率从而选择使用寄存器作为辅助，计算机网络同样存在缓存，下面我们就来介绍一下计算机网络中的缓存。

`Cache-Control` 是通用标头的指令，它能够管理如何对 HTTP 的请求或者响应使用缓存。

因为计算机网络中是可以有 `第三者` 出现的，也就是 `缓存服务器` ，这个指令通过影响 `请求/响应` 中的缓存服务器从而达到控制缓存的目的；不仅有缓存服务器，还有浏览器内部缓存也会影响链路的缓存。

这个标头中可以出现许多单独的指令，其详细信息可以在 RFC 2616 中找到，**即使这是常规标头，某些指令也只能出现在请求或响应中**。下表提供了一个 Cache-Control 选项的总结并告诉你如何去使用

> 请注意，在 Cache-Control 标头中只能出现一个指令，但是在消息中可以出现多个这样的标头。

缓存控制指令	HTTP 消息类型	描述	
no-cache	请求或响应	Cache-Control: no-cache	强制缓存设备将对同一内容的所有后续请求转发到服务器以进行重新验证。
public	响应	Cache-Control: public	表示响应可以被任何缓存所缓存。
private	响应	Cache-Control: private	当指定 private 指令后，响应只能以特定的用户作为对象，并且不应该放入共享缓存中。
no-strore	请求或响应	Cache-Control: no-store	指定整个请求和响应不应该被存储在缓存中。
max-age	请求或响应	Cache-Control: max-age=[秒]	表示如果服务器/客户端发送该指令的数值要比真实时间数值小，则会返回缓存资源。
s-maxage	响应	Cache-Control: s-maxage=[秒]	表示指令指定接收响应的共享缓存的最长期限专用缓存（只为一个客户端提供服务）则使用max-age。
min-fresh	请求	Cache-Control: min-fresh=[秒]	表示客户端收到请求后至少指定时间内缓存不过期
max-stale	请求	Cache-Control: max-stale=[秒]	如果不指定时间，那么不论多了多久客户端都会接受 如果指定时间，那么客户端将接受指定时间内的缓存
only-if cached	请求	Cache-Control: only-if-cached	仅在特殊情况下使用，这个属性表示缓存服务器不会再重新加载指令，强制请求来自缓存。
must-revalidate	响应	Cache-Control: must-revalidate	代理会向源服务器再次验证返回的响应缓存目前是否仍然有效
proxy-revalidate	响应	Cache-Control: proxy-revalidate	和 must-revalidate 很相似，但仅适用于为许多用户提供服务的代理，私有缓存不受影响。
no-transform	请求或响应	Cache-Control: no-transform	指令规定无论是在请求还是响应中，缓存都不能改变实体主体的媒体类型。

上面这个表格其实会有四种分类

- 可缓存性： 它们分别是 no-cache 、 no-store 、 private 和 public
- 缓存有效性时间： 它们分别是 max-age 、 s-maxage 、 max-stale 、 min-fresh
- 重新验证并重新加载： 它们分别是 must-revalidate 和 proxy-revalidate
- 其他： 它们分别是 only-if-cached 和 no-transform

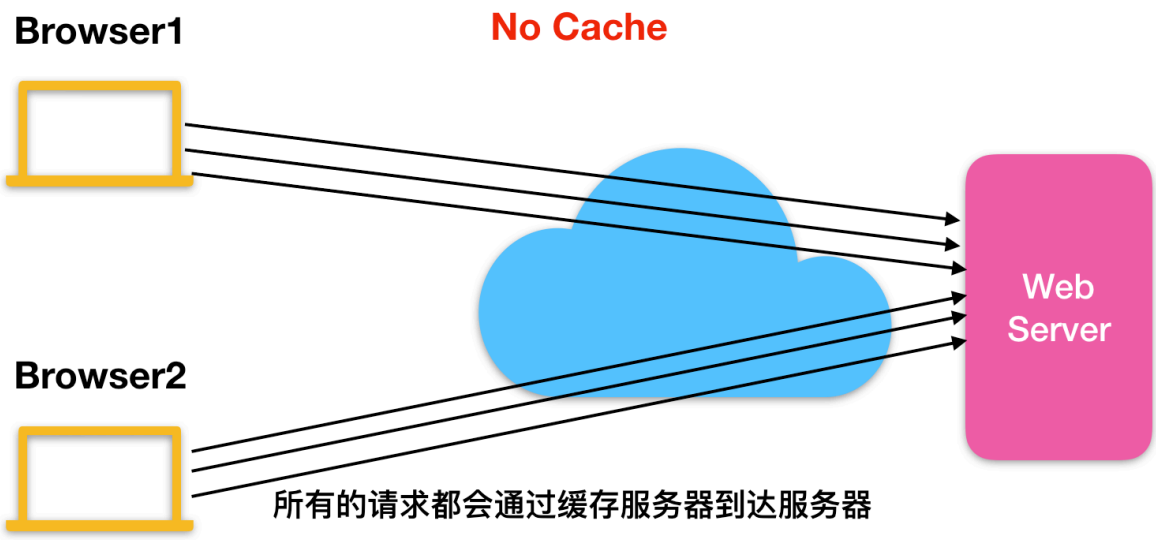
分别对表格中的内容进行一下详细介绍

no-cache

no-cache 很容易和 no-store 混淆，一般都会把 no-cache 认为是不缓存，其实不是这样。

使用 no-cache 指令的目的是**为了防止从缓存中返回过期的资源**，例如下图所示

```
Cache-Control: no-cache
```



no-cache 示意图

举个例子你就明白了，No-Cache 就相当于是 吃着碗里的，占着锅里的 ，如果锅里还有新的肉片，就先吃锅里的，如果锅里没有新的，再吃自己的，这里 锅里的 就相当于是源服务器产生的， 碗里的 就相当于是缓存的。

no-store

no-store 才是真正意义上的 不缓存 ，每次服务器接受到客户端的请求后，都会返回最新的资源给客户端。

```
Cache-Control: no-store
```

max-age

max-age 可以用在请求或者响应中，当客户端发送带有 max-age 的指令时，缓存服务器会判断自己缓存时间的数值和 max-age 的大小，如果比 max-age 小，那么缓存有效，可以继续给客户端返回缓存的数据，如果比 max-age 大，那么缓存服务器将不能返回给客户端缓存的数据。

```
Cache-Control: max-age=60
```

如果 max-age = 0 ，那么缓存服务器将会直接把请求转发到服务器

```
Cache-Control: max-age=0
```

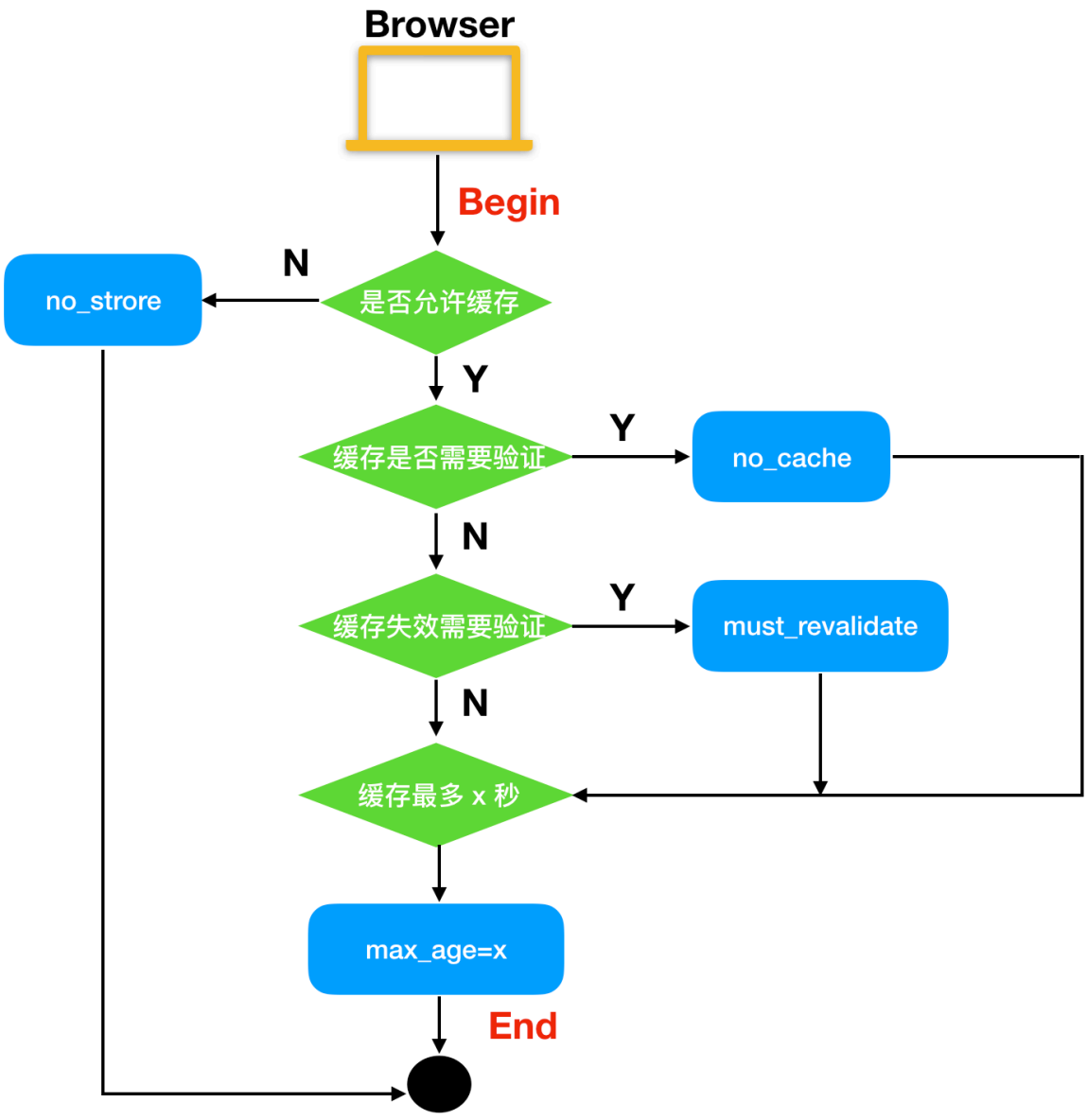
> 注意：这个 max-age 的值是相对于请求时间的

must-revalidate

表示一旦资源过期，缓存就必须在原始服务器上没有成功验证的情况下才使用其过期的数据。

```
Cache-Control: must-revalidate
```

`no-store` 、 `no_cache` 、 `must-revalidate` 和 `max-age` 可以一起看，下面是一个这四个标头的流程图

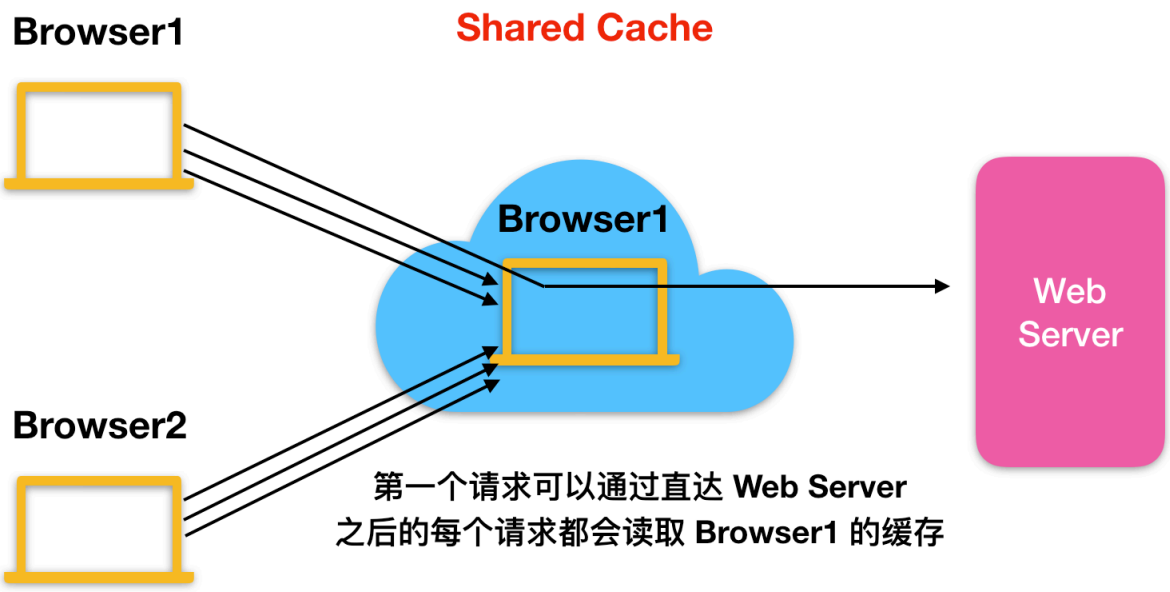


缓存控制流程图

public

`public` 属性只出现在客户端响应中，表示响应可以被任何缓存所缓存。在计算机网络中，分为两种缓存，共享缓存和私有缓存，如下所示

```
Cache-Control: public
```

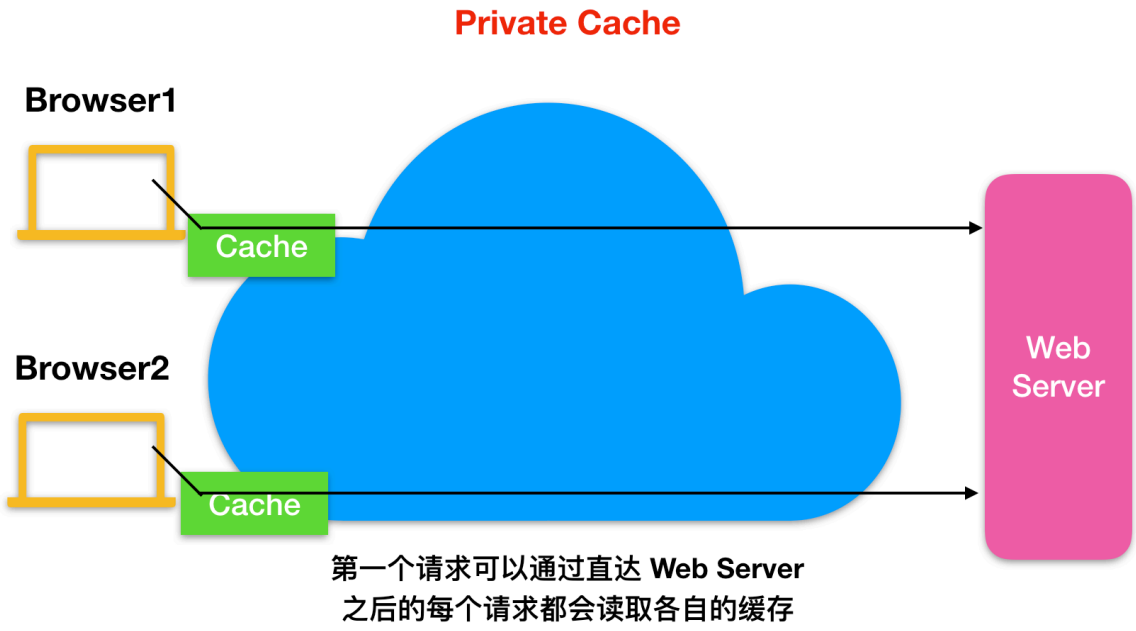


共享缓存示意图

private

当指定 `private` 指令后，响应只以特定的用户作为对象，这与 `public` 的用法相反，缓存服务器只对特定的客户端进行缓存，其他客户端发送过来的请求，缓存服务器则不会返回缓存。

```
Cache-Control: private
```



私有缓存示意图

s-maxage

`s-maxage` 指令的功能和 `max-age` 指令的功能相同，不同点之处在于 s-maxage 不能用于私有缓存，只能用于多用户使用的公共服务器，对于同一用户的重复请求和响应来说，这个指令没有任何作用。

```
Cache-Control: s-maxage=60
```

min-fresh

`min-fresh` 只能出现在请求中， `min-fresh` 要求缓存服务器返回 min-fresh 时间内的缓存数据。例如 `Cache-Control:min-fresh=60` ，这就要求缓存服务器发送60秒内的数据。

```
Cache-Control: min-fresh=60
```

max-stable

`max-stable` 只能出现在请求中，表示客户端会接受缓存数据，即使过期也照常接收。

```
Cache-Control: max-stable=60
```

only-if-cached

这个标头只能出现在请求中，使用 `only-if-cached` 指令表示客户端仅在缓存服务器本地缓存目标资源的情况下才会要求其返回。

```
Cache-Control: only-if-cached
```

proxy-revalidate

`proxy-revalidate` 指令要求所有的缓存服务器在接收到客户端带有该指令的请求返回响应之前，必须再次验证缓存的有效性。

```
Cache-Control: proxy-revalidate
```

no-transform

使用 `no-transform` 指令规定无论是在请求还是响应中，缓存都不能改变实体主体的媒体类型。

```
Cache-Control: no-transform
```

Connection

HTTP 协议使用 TCP 来管理连接方式，主要有两种连接方式， `持久性连接` 和 `非持久性连接` 。

持久性连接

持久性连接指的是一次会话完成后，TCP 连接并未关闭，第二次再次发送请求后，就不再需要建立 TCP 连接，而是可以直接进行请求和响应。它的一般表示形式如下

```
Connection: keep-alive
```

从 HTTP 1.1 开始，默认使用持久性连接。

`keep-alive` 也是一个通用标头，一般 Connection 都会和 keep-alive 一起使用，keep-alive 有两个参数，一个是 `timeout` ；另一个是 `max` ，它们的主要表现形式如下

```
Connection: Keep-Alive
Keep-Alive: timeout=5, max=1000
```

- timeout: 指的是空闲连接必须打开的最短时间，也就是说这次请求的连接时间不能少于5秒，
- max: 指的是在连接关闭之前服务器所能够收到的最大请求数。

非持久性连接

非持久性连接表示一次会话请求/响应后关闭连接的方式。HTTP 1.1 之前使用的连接都是非持久连接，也就是

```
Connection: close
```

Date

Date 是一个通用标头，它可以出现在请求标头和响应标头中，它的基本表示如下

```
Date: Wed, 21 Oct 2015 07:28:00 GMT
```

表示的是格林威治标准时间，这个时间要比北京时间慢八个小时

Pragma

`Pragma` 是 http 1.1 之前版本的历史遗留字段，仅作为与 http 的向后兼容而定义。它的一般形式如下

```
Pragma: no-cache
```

只用于客户端发送的请求中。客户端会要求所有的中间服务器不返回缓存的资源。

如果所有的中间服务器都以实现 HTTP /1.1为标准，那么直接使用 Cache-Control: no-cache 即可，如果不是的话，就要包含两个字段，如下

```
Cache-Control: no-cache
Pragma: no-cache
```

Trailer

首部字段 Trailer 会事先说明在报文主体后记录了哪些首部字段。该首部字段可应用在 HTTP/1.1 版本分块传输编码时。一般用法如下

```
Transfer-Encoding: chunked
Trailer: Expires
```

以上用例中，指定首部字段 Trailer 的值为 Expires，在报文主体之后（分块长度 0 之后）出现了首部字段 Expires。

Transfer-Encoding

Transfer-Encoding 属于内容协商的范畴，下面会具体介绍一下内容协商，现在先做个预告：`Transfer-Encoding` 规定了传输报文所采用的编码方式

> 注意：HTTP 1.1 的传输编码方式仅对分块传输有效，但是 HTTP 2.0 就不再支持分块传输，而提供了自己更有效的数据传输机制。

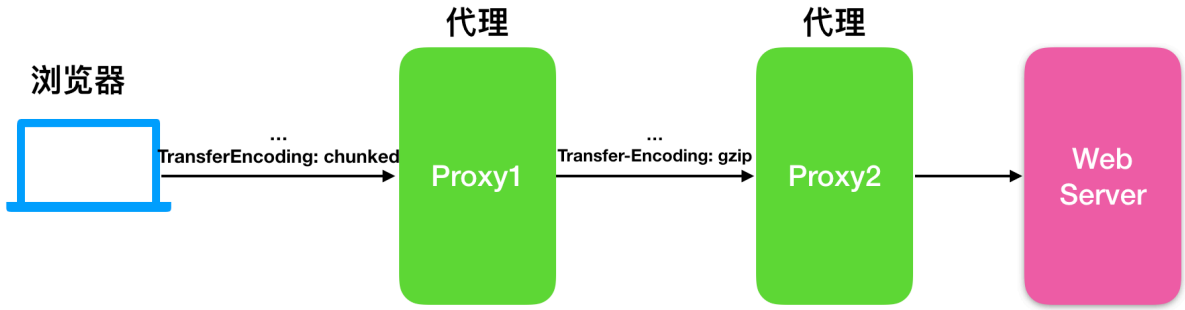
```
Transfer-Encoding: chunked
```

Transfer-Encoding 也属于 `Hop-by-hop（逐跳） 首部` ，下面来回顾一下，HTTP 报文标头除了可以根据属性所在的位置分为 `通用标头` 、 `请求标头` 、 `响应标头` 和 `实体标头` ；还可以按照是否被缓存分为 `端到端首部(End-to-End)` 和 `逐跳首部(Top-to-Top)` 。

除了下面八种属于逐跳首部外，其余都属于端到端首部

Connection、Keep-Alive、Proxy-Authenticate、Proxy-Authorization、Trailer、TE、Transfer-Encoding、Upgrade

下面回到讨论中来，Transfer-Encoding 用于两个节点之间传输消息，而不是资源本身。在多个节点传输消息的过程中，每一段消息的传输都可以使用不同的 `Transfer-Encoding` 。如图所示



Transfer-Encoding 支持文件压缩，如果你想要以文件压缩后的形式发送的话。Transfer-Encoding 所有可选类型如下

- `chunked` ： 数据按照一系列块发送，在这种情况下，将省略 `Content-Length` 标头，并在每个块的开头，需要以十六进制填充当前块的长度，后跟 `'\r\n'` ，然后是块本身，然后是另一个 `'\r\n'` 。当将大量数据发送到客户端并且在请求已被完全处理之前，可能无法知道响应的总大小时，分块编码很有用。 例如，在生成由数据库查询产生的大型 HTML 表时或在传输大型图像时。 分块的响应看起来像这样

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

7\r\n
Mozilla\r\n
9\r\n
Developer\r\n
7\r\n
Network\r\n
0\r\n
\r\n
```

终止块通常是0。紧随 `Transfer-Encoding` 后面的是 `Trailer` 标头， Trailer 可能为空。

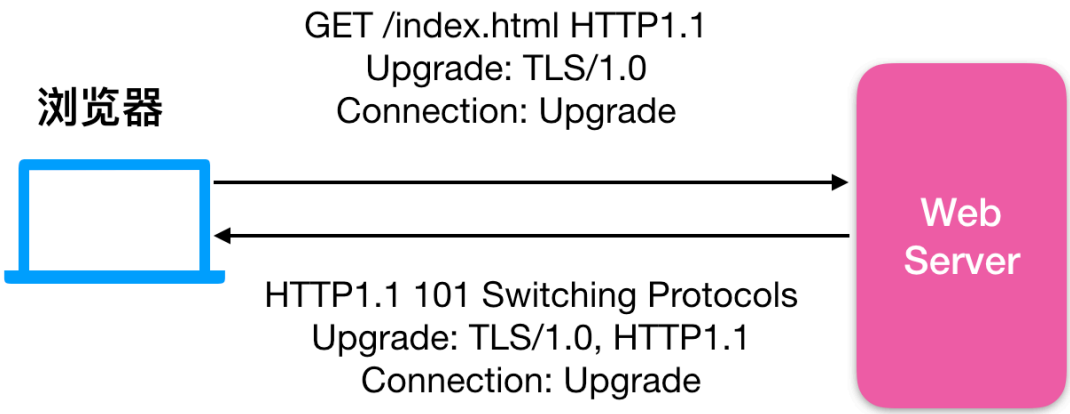
- `compress` ： 使用 `Lempel-Ziv-Welch(LZW)` 算法的格式。值名称取自 `UNIX` 压缩程序，该程序实现了该算法。现在几乎没有浏览器使用这种内容编码了，因为这个专利在 2003 年就停掉了。
- `deflate` ： 使用 `zlib(在 RFC 1950 定义)` 结构和 deflate 压缩算法
- `gzip` ： 使用 `Lempel-Ziv编码 (LZ77)` 和32位 `CRC` 的格式。这最初是 `UNIX gzip` 程序的格式。 `HTTP / 1.1` 标准还建议出于兼容性目的，支持此内容编码的服务器应将 x-gzip 识别为别名。
- `identity` ： 使用身份功能（即无压缩或修改）。

也可以列出多个值，以逗号分隔，类似一个集合列表

```
Transfer-Encoding: gzip, chunked
```

Upgrade

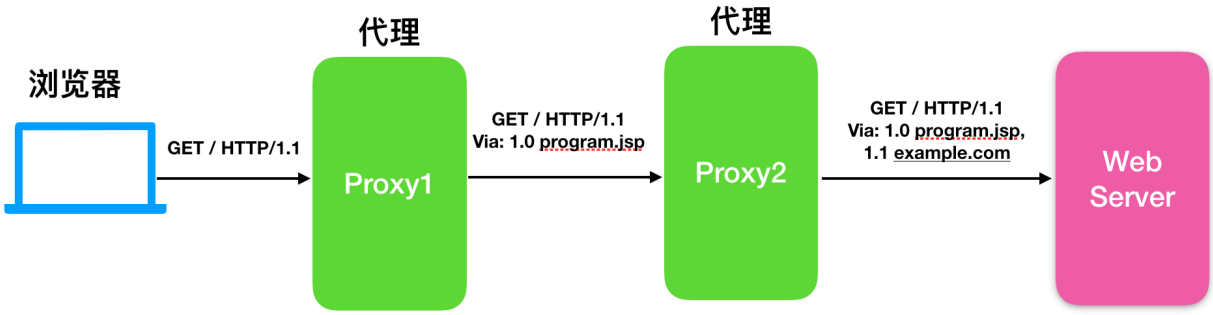
首部字段 Upgrade 用于检测 HTTP 协议及其他协议是否可使用更高的版本进行通信，其参数值可以用来指定一个完全不同的通信协议。



上图用例中，首部字段 `Upgrade` 指定的值为 `TLS/1.0` 。请注意此处两个字段首部字段的对应关系，Connection 的值被指定为 Upgrade。 Upgrade 首部字段产生作用的对象仅限于客户端和临近服务器之间。因此，使用首部字段 Upgrade 时，还需要额外指定 `Connection: Upgrade` 。 对于附有首部字段 Upgrade 的请求，服务器可用 `101 Switching Protocols` 状态码作为响应返回。

Via

使用 Via 是为了跟踪客户端和服务器之间的请求/响应路径，避免请求循环以及能够识别 `请求/响应` 链中发送者协议的功能。Via 字段由代理服务器添加，不论是正向代理还是反向代理，并且可以出现在请求标头和响应标头中。它用于跟踪消息转发。例如下图所示



Via 后面的的 `1.1, 1.0` 表示接收服务器上的 HTTP 版本，Via 首部是为了跟踪路径，经常和 `TRACE` 方法一起使用。

Warning

> 注意：Warning 字段即将被弃用
>
> 查阅 **Warning (https://github.com/httpwg/http-core/issues/139)** and **Warning: header & stale-while-revalidate (https://github.com/whatwg/fetch/issues/913)** 获取更多细节

Warning 通用 HTTP 标头通常会告知用户一些与缓存相关的问题的警告

HTTP/1.1 中定义了 7 种警告。它们分别如下

警告码	警告内容	说明
110	Response is stale（响应已过期）	代理返回已过期的资源
111	Revalidation failed（再验证失败）	代理再验证资源有效性时失败（服务器无法到达等原因）
112	Disconnection operation（断开连接操作）	代理与互联网连接被故意切断
113	Heuristic expiration（试探性过期）	响应的使用期超过24小时（有效缓存的设定时间大于24小时的情况下）
199	Miscellaneous warning（杂项警告）	任意的警告内容
214	Transformation applied（使用了转换）	代理对内容编码或媒体类型等执行了某些处理时
299	Miscellaneous persistent warning（持久杂项警告）	任意的警告内容

请求标头

请求标头用于客户端发送 HTTP 请求到服务器中所使用的字段，下面我们一起来看一下 HTTP 请求标头都包含哪些字段，分别是什么意思。下面会介绍

- Accept

- Accept-Charset
- Accept-Encoding
- Accept-Language
- Authorization
- Expect
- From
- Host
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Proxy-Authorization
- RangeReferer
- TE
- User-Agent

下面分别来介绍一下

Accept

HTTP 请求标头会告知客户端能够接收的 MIME 类型是什么

那么什么是 MIME 类型呢？在回答这个问题前你应该先了解一下什么是 MIME

> MIME: MIME (Multipurpose Internet Mail Extensions) 是描述消息内容类型的因特网标准。MIME 消息能包含文本、图像、音频、视频以及其他应用程序专用的数据。

也就是说，MIME 类型其实就是一系列消息内容类型的集合。那么 MIME 类型都有哪些呢?

`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

`应用程序二进制文件`： application/octet-stream、application/zip

比如，如果浏览器不支持 PNG 图片的显示，那 Accept 就不指定image/png，而指定可处理的 image/gif 和 image/jpeg 等图片类型。

一般 MIME 类型也会和 `q` 这个属性一起使用，q 是什么？q 表示的是权重，来看一个例子

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

这是什么意思呢？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，没有显示权重的时候默认值是1.0 ，我给你列个表格你就明白了

q	MIME
1.0	text/html
1.0	application/xhtml+xml
0.9	application/xml
0.8	* / *

也就是说，这是一个放置顺序，权重高的在前，低的在后，`application/xml;q=0.9` 是不可分割的整体。

Accept-Charset

`Accept-Charset` 表示客户端能够接受的字符编码。Accept-Charset 也是属于 `内容协商` 的一部分，它和

`Accept` 一样，也可以用 q 来表示字符集，用 `逗号` 进行分割，例如

```
Accept-Charset: iso-8859-1
Accept-Charset: utf-8, iso-8859-1;q=0.5
Accept-Charset: utf-8, iso-8859-1;q=0.5, *;q=0.1
```

> 事实上，很多以 `Accept-*` 开头的标头，都是属于内容协商的范畴，关于内容协商我们下面会说。

Accept-Encoding

表示 HTTP 标头会标明客户端希望服务端返回的内容编码，这通常是一种压缩算法。Accept-Encoding 也是属于 `内容协商` 的一部分，使用并通过客户端选择 `Content-Encoding` 内容进行返回。

即使客户端和服务器都能够支持相同的压缩算法，服务器也可能选择不压缩并返回，这种情况可能是由于这两种情况造成的:

- 要发送的数据已经被压缩了一次，第二次压缩并不会导致发送的数据更小
- 服务器过载，无法承受压缩带来的性能开销，通常，如果服务器使用 CPU 超过 80% ，`Microsoft` 则建议不要使用压缩

下面是 Accept-Encoding 的使用方式

```
Accept-Encoding: gzip
Accept-Encoding: compress
Accept-Encoding: deflate
Accept-Encoding: br
Accept-Encoding: identity
Accept-Encoding: *
Accept-Encoding: deflate, gzip;q=1.0, *;q=0.5
```

上面的几种表述方式就已经把 Accept-Encoding 的属性列全了

- `gzip`：由文件压缩程序 gzip 生成的编码格式，使用 `Lempel-Ziv编码（LZ77）` 和32位CRC的压缩格式，感兴趣的同学可以读一下 （**https://en.wikipedia.org/wiki/LZ77_and_LZ78#LZ77**）
- `compress`：使用 `Lempel-Ziv-Welch（LZW）` 算法的压缩格式，有兴趣的同学可以读 （**https://en.wikipedia.org/wiki/LZW**）
- `deflate`：使用 zlib 结构和 deflate 压缩算法的压缩格式，参考 （**https://en.wikipedia.org/wiki/Zlib**） 和 （**https://en.wikipedia.org/wiki/DEFLATE**）
- `br`：使用 Brotli 算法的压缩格式，参考 （**https://en.wikipedia.org/wiki/Brotli**）
- 不执行压缩或不会变化的默认编码格式
- `*`：匹配标头中未列出的任何内容编码，如果没有列出 `Accept-Encoding` ，这就是默认值，并不意味着支

  持任何算法，只是表示没有偏好
- `;q=` 采用权重 q 值来表示相对优先级，这点与首部字段 Accept 相同。

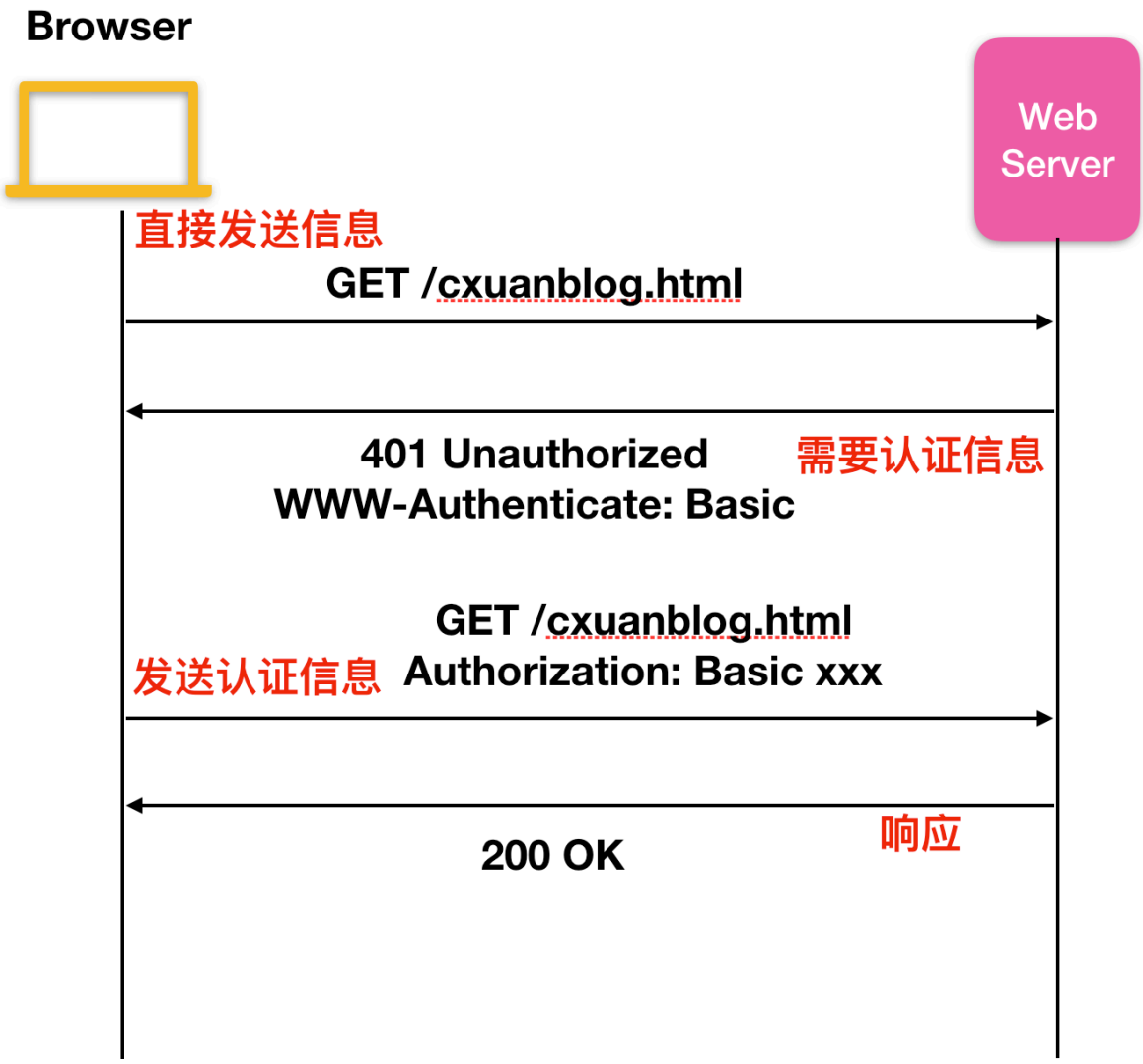
Accept-Language

`Accept-Language` 请求表示客户端需要服务端返回的语言类型，Accept-Language 也属于内容协商的范畴。服务端通过 `Content-Language` 进行响应，和 Accept 首部字段一样，按权重值 `q` 来表示相对优先级。例如

```
Accept-Language: de
Accept-Language: de-CH
Accept-Language: en-US,en;q=0.5
```

Authorization

HTTP `Authorization` 请求头用于向服务器认证用户代理的凭据，通常用在服务器以401未经授权状态和WWW-Authenticate标头响应之后，啥意思呢？你不明白的话我画张图给你看



请求标头 `Authorization` 是用来告知服务器，用户的认证信息，服务器在只有收到认证后才会返回给客户端 200 OK 的响应，如果没有认证信息，则会返回 401 并告知客户端需要认证信息。详细关于 Authorization 的信息，后面也会详细解释

Expect

Expect HTTP 请求标头指示服务器需要满足的期望才能正确处理请求。如果服务器没有办法完成客服端所期望完成的事情并且服务端存在错误的话，会返回 `417 Expectation Failed` 。HTTP 1.1 只规定了 `100-continue` 。

- 如果服务器能正常完成客户端所期望的事情，会返回 100
- 如果不能满足期望或返回任何其他 `4xx` 的状态码，会返回 417

例如

```
PUT /somewhere/fun HTTP/1.1
Host: origin.example.com
Content-Type: video/h264
Content-Length: 1234567890987
Expect: 100-continue
```

From

`From` 请求头用来告知服务器使用用户代理的电子邮件地址。通常情况下，其使用目的就是为了显示搜索引擎等用户代理的负责人的电子邮件联系方式。我们在使用代理的情况下，应尽可能包含 From 首部字段。例如

```
From: webmaster@example.org
```

> 你不应该将 From 用在访问控制或者身份验证中

Host

`Host` 请求头指明了服务器的域名（对于虚拟主机来说），以及（可选的）服务器监听的TCP端口号。如果没有给定端口号，会自动使用被请求服务的默认端口（比如请求一个 HTTP 的 URL 会自动使用80作为端口）。

```
Host: developer.mozilla.org
```

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。

If-Match

If-Match 后面可以跟一大堆属性，形式像 If-Match 这种的请求头称为 `条件请求` ，服务器接收到条件请求后，需要判定条件请求是否满足，只有条件请求为真，才会执行条件请求

类似的还有 **If-Match、If-Modified-Since、If-None-Match、If-Range、If-Unmodified-Since**

对于 `GET` 和 `POST` 方法，服务器仅在与列出的 `ETag（响应标头）` 之一匹配时才返回请求的资源。这里又多了一个新词 `ETag` ，我们稍后再说 ETag 的用法。对于像是 `PUT` 和其他非安全的方法，在这种情况下，它仅仅将上传资源。

下面是两种常见的案例

- 对于 `GET` 和 `POST` 方法，会结合使用 `Range` 标头，它可以确保新发送请求的范围与上一个请求的资源相同，如果不匹配的话，会返回 `416` 响应。
- 对于其他方法，特别是 `PUT` 方法， `If-Match` 可以防止丢失更新，服务器会比对 If-Match 的字段值和资源的 ETag 值，仅当两者一致时，才会执行请求。反之，则返回状态码 412 Precondition Failed 的响应。例如

```
If-Match: "bfc13a64729c4290ef5b2c2730249c88ca92d82d"
If-Match: *
```

If-Modified-Since

`If-Modified-Since` 是 HTTP 条件请求的一部分，只有在给定日期之后，服务端修改了请求所需要的资源，才会返回 200 OK 的响应。如果在给定日期之后，服务端没有修改内容，响应会返回 `304` 并且不带任何响应体。If-Modified-Since 只能使用 `GET` 和 `HEAD` 请求。

If-Modified-Since 与 If-None-Match 结合使用时，它将被忽略，除非服务器不支持 If-None-Match。一般表示如下

```
If-Modified-Since: Wed, 21 Oct 2015 07:28:00 GMT
```

> 注意：这是格林威治标准时间。 HTTP 日期始终以格林尼治标准时间表示，而不是本地时间。

If-None-Match

条件请求，它与 `If-Match` 的作用相反，仅当 `If-None-Match` 的字段值与 `ETag` 值不一致时，可处理该请求。对于 `GET` 和 `HEAD` ，仅当服务器没有与给定资源匹配的 `ETag` 时，服务器将返回 200 作为响应。对于其他方法，仅当最终现有资源的 ETag 与列出的任何值都不匹配时，才会处理请求。

当 `GET` 和 `POST` 发送的 `If-None-Match` 与 `ETag` 匹配时，服务器会返回 `304` 。

```
If-None-Match: "bfc13a64729c4290ef5b2c2730249c88ca92d82d"
If-None-Match: W/"67ab43", "54ed21", "7892dd"
If-None-Match: *
```

有同学可能会好奇 `W/` 是什么意思，这其实是 ETag 的弱匹配，关于 ETag 我们会在响应标头中详细讲述。

If-Range

`If-Range` 也是条件请求，如果满足条件（If-Range 的值和 ETag 值或者更新的日期时间一致），则会发出范围请求，否则将会返回全部资源。它的一般表示如下

```
If-Range: Wed, 21 Oct 2015 07:28:00 GMT
```

If-Unmodified-Since

`If-Unmodified-Since` HTTP 请求标头也是一个条件请求，服务器只有在给定日期之后没有对其进行修改时，服务器才返回请求资源。如果在指定日期时间后发生了更新，则以状态码 `412 Precondition Failed` 作为响应返回。

```
If-Unmodified-Since: Wed, 21 Oct 2015 07:28:00 GMT
```

Max-Forwards

MDN 把这个标头置灰了，所以下面内容取自《图解 HTTP》

`Max-Forwards` 一般用于 `TRACE` 和 `OPTION` 方法，发送包含 `Max-Forwards` 的首部字段时，每经过一个服务器，Max-Forwards 的值就会 -1，直到 Max-Forwards 为0时返回。Max-Forwards 是一个十进制的整数值。

```
Max-Forwards: 10
```

可以灵活使用首部字段 Max-Forwards，针对以上问题产生的原因展开调查。由于当 Max-Forwards 字段值为 0 时，服务器就会立即返回响应，由此我们至少可以对以那台服务器为终点的传输路径的通信状况有所把握。

Proxy-Authorization

`Proxy-Authorization` 是属于请求与认证的范畴，我们在上面提到一个认证的 HTTP 标头是 Authorization，不同于 Authorization 发生在客户端 - 服务器之间；`Proxy-Authorization` 发生在代理服务器和客户端之间。它表示接收到从代理服务器发来的认证时，客户端会发送包含首部字段 Proxy-Authorization 的请求，以告知服务器认证所需要的信息。

```
1   Proxy-Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
```

Range

`Range` HTTP 请求标头指示服务器应返回文档指定部分的资源，可以一次请求一个 Range 来返回多个部分，服务器会将这些资源返回各个文档中。如果服务器成功返回，那么将返回 206 响应；如果 Range 范围无效，服务器返回 `416 Range Not Satisfiable` 错误；服务器还可以忽略 Range 标头，并且返回 200 作为响应。

```
1   Range: bytes=200-1000, 2000-6576, 19000-
```

Referer

HTTP Referer 属性是请求标头的一部分，当浏览器向 web 服务器发送请求的时候，一般会带上 Referer，告诉服务器该网页是从哪个页面链接过来的，服务器因此可以获得一些信息用于处理。

```
1   Referer: https://developer.mozilla.org/testpage.html
```

TE

首部字段 `TE` 会告知服务器客户端能够处理响应的传输编码方式及相对优先级。它和首部字段 Accept-Encoding 的功能很相像，但是用于传输编码。

```
1   TE: gzip, deflate;q=0.5
```

首部字段 TE 除指定传输编码之外，还可以指定伴随 trailer 字段的分块传输编码的方式。应用后者时，只需把 trailers 赋值给该字段值。

```
1   TE: trailers, deflate;q=0.5
```

User-Agent

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。

```
1   Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:47.0) Gecko/20100101 Firefox/47.0
```

响应标头

刚刚我们的着重点一直放在客户端请求，现在我们把关注点转换一下放在服务器上。响应首部字段是由服务器发送给客户端响应中所包含的字段，用于补充相应信息等，这部分标头也是非常多，我们先一起来看一下

- Accept-Ranges
- Age
- ETag
- Location

- Proxy-Authenticate
- Retry-After
- Server
- Vary
- www-Authenticate

Accept-Ranges

Accept-Ranges HTTP 响应标头，这个标头有两个值

- 当服务器能够处理客户端发送过来的请求时，使用 `bytes` 来指定
- 当服务器不能处理客户端发来的请求时，使用 `none` 来指定

```
1   Accept-Ranges: bytes
2   Accept-Ranges: none
```

Age

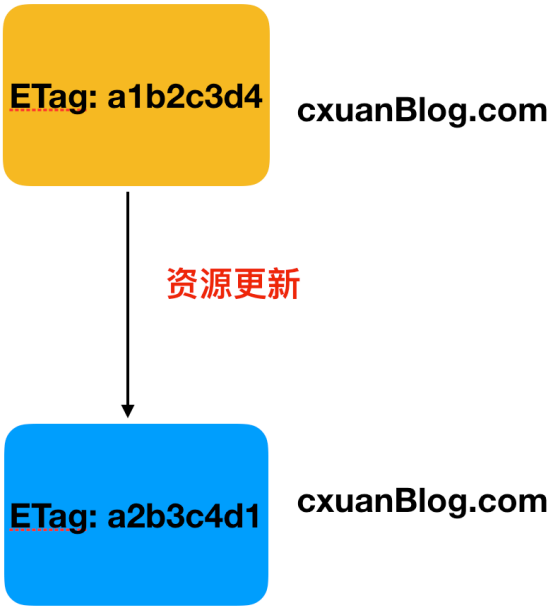
Age HTTP 响应标头告诉客户端源服务器在多久之前创建了响应，它的单位为 `秒`，Age 标头通常接近于0，如果是0则可能是从源服务器获取的，如果不是表示可能是由代理服务器创建，那么 Age 的值表示的是缓存后的响应再次发起认证到认证完成的时间值。代理创建响应时必须加上首部字段 Age。一般表示如下

```
1   Age: 24
```

ETag

ETag 对于条件请求来说真是太重要了。因为条件请求就是根据 ETag 的值进行匹配的，下面我们就来详细了解一下。

ETag 响应头是 `特定版本` 的标识，它能够使缓存变得更高效并能够节省带宽，因为如果缓存内容未发生变更，Web 服务器则不需要重新发送完整的响应。除此之外，ETag 能够防止资源同时更新互相覆盖。



如果给定 URL 上的资源发生变更，必须生成一个新的 `ETag` 值，通过比较它们可以确定资源的两个表示形式是否相同。

ETag 值有两种，一种是强 ETag，一种是弱 ETag；

- 强 ETag 值，无论实体发生多么细微的变化都会改变其值，一般的表示如下

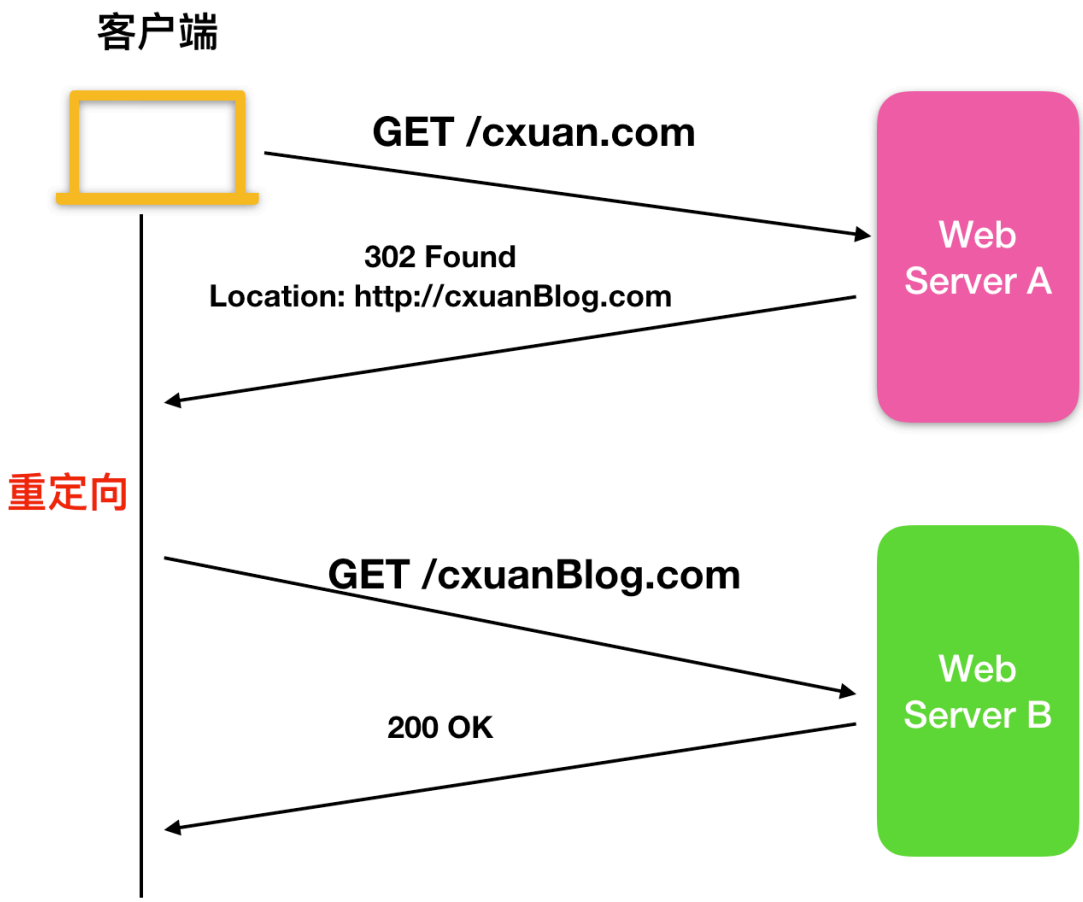
```
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

- 弱 ETag 值，弱 ETag 值只用于提示资源是否相同。只有资源发生了根本改变，产生差异时才会改变 ETag 值。这时，会在字段值最开始处附加 W/。

```
ETag: W/"0815"
```

Location

Location 响应标头表示 URL 需要重定向页面，它仅仅与 `3xx(重定向)` 或 `201(已创建)` 状态响应一起使用。下面是一个页面重定向的过程



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

`Location` 和 `content-Location` 是不一样的：Location 表示目标的重定向（或新创建资源的 URL）。然而 Content-Location 表示发生内容协商时用于访问资源的直接 URL，而无须进一步协商。Location 是与响应相关联的标头，而 Content-Location 与返回的实体相关联。

```
Location: /index.html
```

Proxy-Authenticate

HTTP 响应标头 `Proxy-Authenticate` 会定义认证方法，应该使用身份验证方法来访问代理服务器后面的资源即客户端。

它与 HTTP 客户端和服务端之间的访问认证行为相似，不同之处在于 `Proxy-Authenticate` 的认证双方是客户端与代理之间。它的一般表示形式如下

```
Proxy-Authenticate: Basic
Proxy-Authenticate: Basic realm="Access to the internal site"
```

Retry-After

HTTP 响应标头 Retry-After 告知客户端需要在多久之后重新发送请求，使用此标头主要有如下三种情况

- 当发送 `503(服务不可用)` 响应时，这表示该服务预计无法使用多长时间。
- 当发送 `429(太多请求)` 响应时，这表示发出新请求之前要等待多长时间。
- 当发送重定向的响应像是 `301(永久移动)` ，这表示在发出重定向请求之前要求用户客户端等待的最短时间。

字段值可以指定为具体的日期时间，也可以是创建响应后所持续的秒数，例如

```
Retry-After: Wed, 21 Oct 2015 07:28:00 GMT
Retry-After: 120
```

Server

服务器标头包含有关原始服务器用来处理请求的软件的信息。

应该避免使用过于冗长和详细的 Server 值，因为它们可能会泄露内部实施细节，这可能会使攻击者容易地发现并利用已知的安全漏洞。例如下面这种写法

```
Server: Apache/2.4.1 (Unix)
```

Vary

Vary HTTP 响应标头确定如何匹配请求标头，以决定是否可以使用缓存的响应，而不是从原始服务器请求一个新的响应。

```
Vary: User-Agent
```

www-Authenticate

HTTP `WWW-Authenticate` 响应标头定义了应用于获得对资源的访问权限的身份验证方法。WWW-Authenticate标头与401未经授权的响应一起发送。它的一般表示形式如下

```
WWW-Authenticate: Basic
WWW-Authenticate: Basic realm="Access to the staging site", charset="UTF-8"
```

Access-Control-Allow-Origin

一个返回的 HTTP 标头可能会具有 Access-Control-Allow-Origin ， `Access-Control-Allow-Origin` 指定一个来源，它告诉浏览器允许该来源进行资源访问。 否则-对于没有凭据的请求 `*` 通配符，告诉浏览器允许任何源访问资源。例如，要允许源 `https://mozilla.org` 的代码访问资源，可以指定：

```
Access-Control-Allow-Origin: https://mozilla.org
Vary: Origin
```

如果服务器指定单个来源而不是 `*` 通配符的话 ，则服务器还应在 Vary 响应标头中包含 `Origin` ，以向客户端指示 服务器响应将根据原始请求标头的值而有所不同。

实体标头

实体标头用于HTTP请求和响应中，例如 Content-Length，Content-Language，Content-Encoding 的标头是实体标头。实体标头不局限于请求标头或者响应标头，下面例子中， `Content-Length` 是一个实体标头，但是却出现在了请求报文中

```
POST /myform.html HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Content-Length: 128
```

下面就来说一下实体标头都包含哪些

- Allow
- Content-Encoding
- Content-Language
- Content-Length
- Content-Location
- Content-MD5
- Content-Range
- Content-Type
- Expires
- Last-Modified

下面来分开说一下

Allow

HTTP 实体标头 `Allow` 列出了资源支持的方法集合。如果服务器响应 `405 Method Not Allowed` 状态码以指示可以使用哪些请求方法，则必须发送此标头。例如

```
Allow: GET, POST, HEAD
```

这段代码表示服务器允许支持 `GET` 、 `POST` 和 `HEAD` 方法。当服务器接收到不支持的 HTTP 方法时，会以状态码 `405 Method Not Allowed` 作为响应返回。

Content-Encoding

我们上面讲过 `Accept-Encoding` 是客户端希望服务端返回的内容编码，但是实际上服务端返回给客户端的内容编码实际上是通过 `Content-Encoding` 返回的。内容编码是指在不丢失实体信息的前提下所进行的压缩。主要也是四种，和 Accept-Encoding 相同，它们是 **gzip**、**compress**、**deflate**、**identity**。下面是一组请求/响应内容压缩编码

```
Accept-Encoding: gzip, deflate
Content-Encoding: gzip
```

Content-Language

首部字段 Content-Language 会告知客户端，服务器使用的自然语言是什么，它与 Accept-Language 相对，下面是一组请求/响应使用的语言类型

```
Content-Language: de-DE, en-CA
```

Content-Length

Content-Length 的实体标头指服务器发送给客户端的实际主体大小，以字节为单位。

```
Content-Length: 3000
```

如上，服务器返回给客户端的主体大小是 3000 字节。

Content-Location

Content-Location 可不是对应 Accept-Location，因为没有这个标头哈哈哈哈。实际上 Content-Location 对应的是 `Location` 。

Location 和 Content-Location 是不一样的，Location 表示重定向的 URL，而 Content-Location 表示用于访问资源的直接 URL，以后无需进行进一步的内容协商。Location 是与响应关联的标头，而 Content-Location 是与返回的数据相关联的标头，如果你不好理解，看一下下面的表格

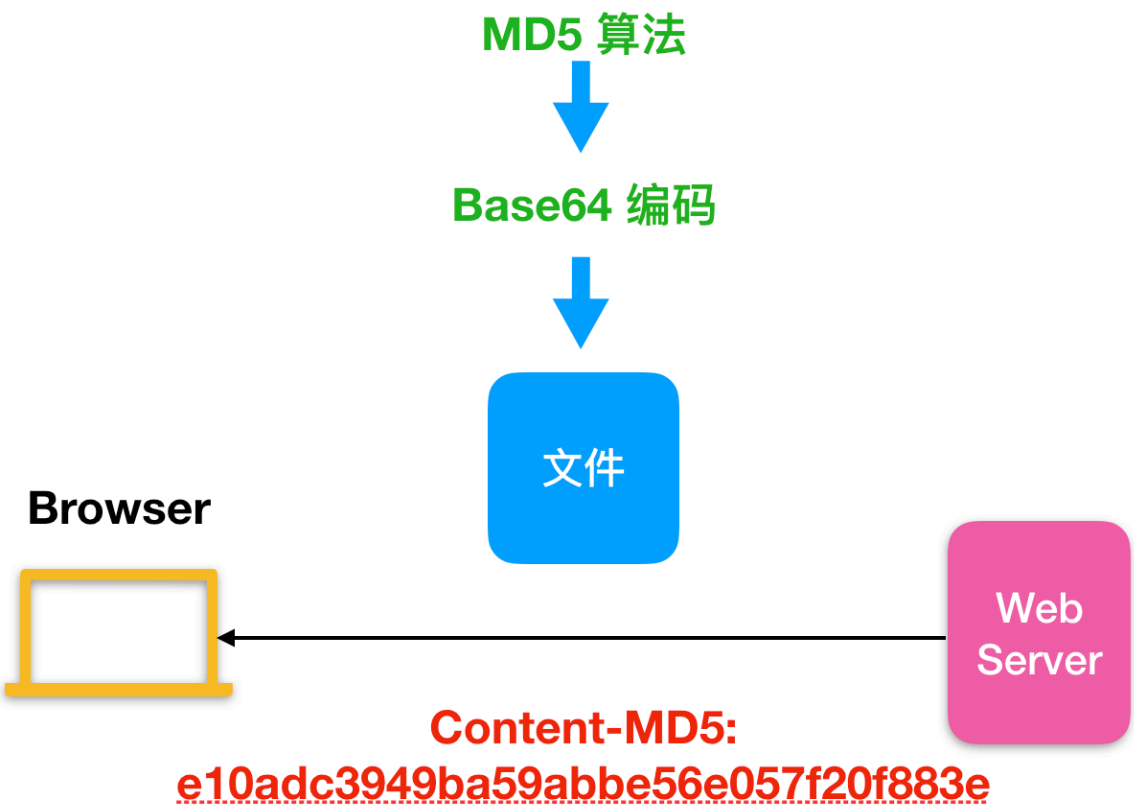
Request header	Response header
`Accept: application/json, text/json`	`Content-Location: /documents/foo.json`
`Accept: application/xml, text/xml`	`Content-Location: /documents/foo.xml`
`Accept: text/plain, text/*`	`Content-Location: /documents/foo.txt`

Content-MD5

客户端会对接收的报文主体执行相同的 MD5 算法，然后与首部字段 Content-MD5 的字段进行比较。

```
Content-MD5: e10adc3949ba59abbe56e057f20f883e
```

首部字段 Content-MD5 是一串由 MD5 算法生成的值，其目的在于检查报文主体在传输过程中是否保持完整，有无被修改的情况，以及确认传输到达。



Content-Range

HTTP 的 Content-Range 响应标头是针对范围请求而设定的，返回响应时使用首部字段 `Content-Range`，能够告知客户端响应实体的哪部分是符合客户端请求的，字段以字节为单位。它的一般表示如下

```
1    Content-Range: bytes 200-1000/67589
```

上段代码表示从所有 `67589` 个字节中返回 `200-1000` 个字节的内容

Content-Type

HTTP 响应标头 Content-Type 说明了实体内对象的媒体类型，和首部字段 Accept 一样使用，表示服务器能够响应的媒体类型。

Expires

HTTP Expires 实体标头包含 `日期/时间`，在该日期/时间之后，响应被认为过期；在响应时间之内被认为有效。特殊的值比如0表示过去的日期，表示资源已过期。

```
1    Expires: Wed, 21 Oct 2015 07:28:00 GMT
```

源服务器会将资源失效的日期或时间发送给客户端，缓存服务器在接受到 Expires 的响应后，会判断是否把缓存返回给客户端。

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。但是，当首部字段 Cache-Control 有指定 max-age 指令时，比起首部字段 Expires，会优先处理 max-age 指令。

Last-Modified

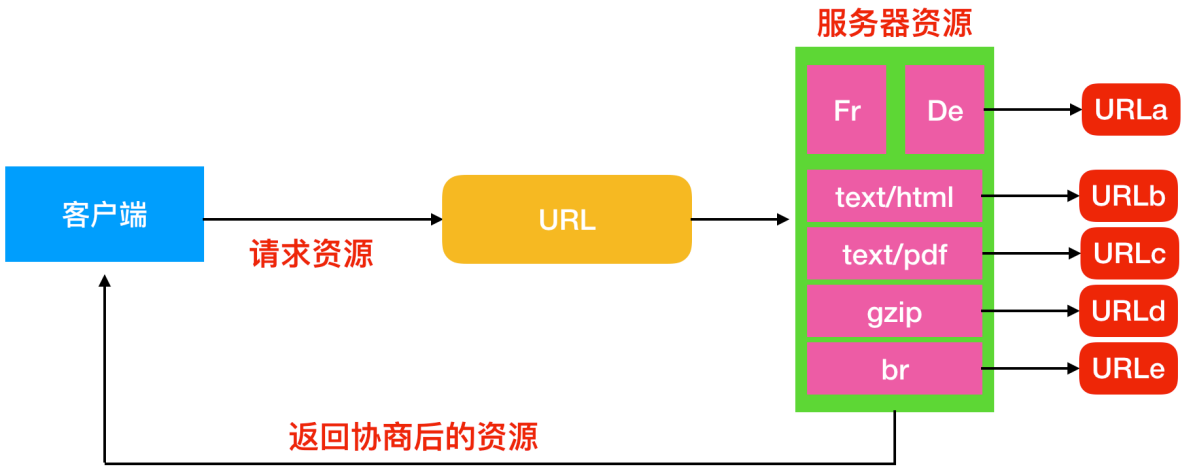
实体字段 `Last-Modified` 指明资源的最后修改时间，它用作验证器来确定接收或存储的资源是否相同。它的作用不如 `ETag` 那么准确，它可以作为一种后备机制，包含 `If-Modified-Since` 或 `If-Unmodified-Since` 标头的条件请求将使用此字段。它的一般表示如下

```
1   Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT
```

HTTP 内容协商

什么是内容协商

在 HTTP 中， `内容协商` 是一种用于在同一 URL 上提供资源的不同表示形式的机制。内容协商机制是指客户端和服务器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标准。



内容协商功能图

内容协商的种类

内容协商主要有以下3种类型：

- `服务器驱动协商（Server-driven Negotiation）`

这种协商方式是由服务器端进行内容协商。服务器端会根据请求首部字段进行自动处理

- `客户端驱动协商（Agent-driven Negotiation）`

这种协商方式是由客户端来进行内容协商。

- `透明协商（Transparent Negotiation）`

是服务器驱动和客户端驱动的结合体，是由服务器端和客户端各自进行内容协商的一种方法。

内容协商的分类有很多种，主要的几种类型是 **Accept、Accept-Charset、Accept-Encoding、Accept-Language、Content-Language**。

一般来说，客户端用 Accept 头告诉服务器希望接收什么样的数据，而服务器用 Content 头告诉客户端实际发送了什么样的数据。

为什么需要内容协商

我们为什么需要内容协商呢？在回答这个问题前我们先来看一下 TCP 和 HTTP 的不同。

在 TCP / IP 协议栈里，传输数据基本上都是 `header+body` 的格式。但 TCP、UDP 因为是传输层的协议，它们不会关心 body 数据是什么，只要把数据发送到对方就算是完成了任务。

而 HTTP 协议则不同，它是应用层的协议，数据到达之后需要告诉应用程序这是什么数据。当然不告诉应用这是哪种类型的数据，应用也可以通过不断尝试来判断，但这种方式无疑十分低效，而且有很大几率会检查不出来文件类型。

所以鉴于此，浏览器和服务器需要就数据的传输达成一致，浏览器需要告诉服务器自己希望能够接收什么样的数据，需要什么样的压缩格式，什么语言，哪种字符集等；而服务器需要告诉客户端自己能够提供的服务是什么。

所以我们就引出了内容协商的几种概念，下面依次来进行探讨

内容协商标头

Accept

接受请求 HTTP 标头会通告客户端自己能够接受的 `MIME` 类型

那么什么是 MIME 类型呢？在回答这个问题前你应该先了解一下什么是 MIME

> MIME: MIME (Multipurpose Internet Mail Extensions) 是描述消息内容类型的因特网标准。MIME 消息能包含文本、图像、音频、视频以及其他应用程序专用的数据。

也就是说，MIME 类型其实就是一系列消息内容类型的集合。那么 MIME 类型都有哪些呢?

`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

`应用程序二进制文件`： application/octet-stream、application/zip

比如，如果浏览器不支持 PNG 图片的显示，那 Accept 就不指定image/png，而指定可处理的 image/gif 和 image/jpeg 等图片类型。

一般 MIME 类型也会和 `q` 这个属性一起使用，q 是什么？q 表示的是权重，来看一个例子

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

这是什么意思呢？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，没有显示权重的时候默认值是1.0 ，我给你列个表格你就明白了

q	MIME
1.0	text/html
1.0	application/xhtml+xml
0.9	application/xml
0.8	* / *

也就是说，这是一个放置顺序，权重高的在前，低的在后，`application/xml;q=0.9` 是不可分割的整体。

Accept-Charset

Accept-charset 属性规定服务器处理表单数据所接受的字符编码；Accept-charset 属性允许你指定一系列字符集，服务器必须支持这些字符集，从而得以正确解释表单中的数据。

Accept-Charset 没有对应的标头，服务器会把这个值放在 `Content-Type` 中用 **charset=xxx**来表示，

例如，浏览器请求 GBK 或 UTF-8 的字符集，然后服务器返回的是 UTF-8 编码，就是下面这样

```
1   Accept-Charset: gbk, utf-8
2   Content-Type: text/html; charset=utf-8
```

Accept-Language

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。

```
1   Accept-Language: en-US,en;q=0.5
```

Accept-Encoding

表示 HTTP 标头会标明客户端希望服务端返回的内容编码，这通常是一种压缩算法。Accept-Encoding 也是属于 `内容协商` 的一部分，使用并通过客户端选择 `Content-Encoding` 内容进行返回。

即使客户端和服务器都能够支持相同的压缩算法，服务器也可能选择不压缩并返回，这种情况可能是由于这两种情况造成的:

- 要发送的数据已经被压缩了一次，第二次压缩并不会导致发送的数据更小
- 服务器过载，无法承受压缩带来的性能开销，通常，如果服务器使用 CPU 超过 80% ，`Microsoft` 则建议不要使用压缩

下面是 Accept-Encoding 的使用方式

```
Accept-Encoding: gzip
Accept-Encoding: compress
Accept-Encoding: deflate
Accept-Encoding: br
Accept-Encoding: identity
Accept-Encoding: *
Accept-Encoding: deflate, gzip;q=1.0, *;q=0.5
```

上面的几种表述方式就已经把 Accept-Encoding 的属性列全了

- `gzip`：由文件压缩程序 gzip 生成的编码格式，使用 `Lempel-Ziv编码（LZ77）` 和32位CRC的压缩格式，感兴趣的同学可以读一下 （**https://en.wikipedia.org/wiki/LZ77_and_LZ78#LZ77**）
- `compress`：使用 `Lempel-Ziv-Welch（LZW）` 算法的压缩格式，有兴趣的同学可以读 （**https://en.wikipedia.org/wiki/LZW**）
- `deflate`：使用 zlib 结构和 deflate 压缩算法的压缩格式，参考 （**https://en.wikipedia.org/wiki/Zlib**） 和 （**https://en.wikipedia.org/wiki/DEFLATE**）
- `br`：使用 Brotli 算法的压缩格式，参考 （**https://en.wikipedia.org/wiki/Brotli**）
- 不执行压缩或不会变化的默认编码格式
- `*`：匹配标头中未列出的任何内容编码，如果没有列出 `Accept-Encoding` ，这就是默认值，并不意味着支

  持任何算法，只是表示没有偏好
- `;q=` 采用权重 q 值来表示相对优先级，这点与首部字段 Accept 相同。

Content-Type

Content-Type 实体标头用于指示资源的 MIME 类型。作为响应，Content-Type 标头告诉客户端返回的内容的内容类型实际上是什么。Content-type 有两种值：MIME 类型和字符集编码，例如

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

> 在某些情况下，浏览器将执行 MIME 嗅探，并且不一定遵循此标头的值；为防止此行为，可以将标头 X-Content-Type-Options 设置为 nosniff。

Content-Encoding

Content-Encoding 实体标头用于压缩媒体类型，它让客户端知道如何进行解码操作，从而使客户端获得 Content-Type 标头引用的 MIME 类型。表示如下

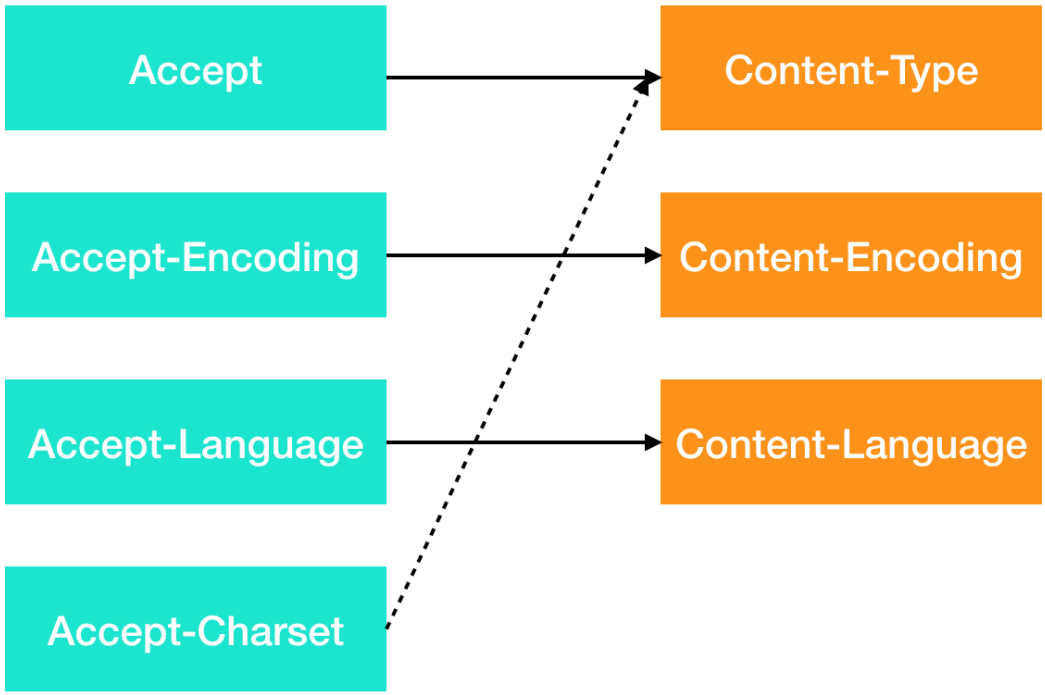
```
Content-Encoding: gzip
Content-Encoding: compress
Content-Encoding: deflate
Content-Encoding: identity
Content-Encoding: br
Content-Encoding: gzip, identity
Content-Encoding: deflate, gzip
```

Content-Language

Content-Language 实体标头用于描述面向受众的语言，以便使用户根据用户自己的首选语言进行区分。例如

```
Content-Language: de-DE
Content-Language: en-US
Content-Language: de-DE, en-CA
```

下面根据内容协商对应的请求/响应标头，我列了一张图供你参考，注意其中 Accept-Charset 没有对应的 Content-Charset ， 而是通过 Content-Type 来表示。



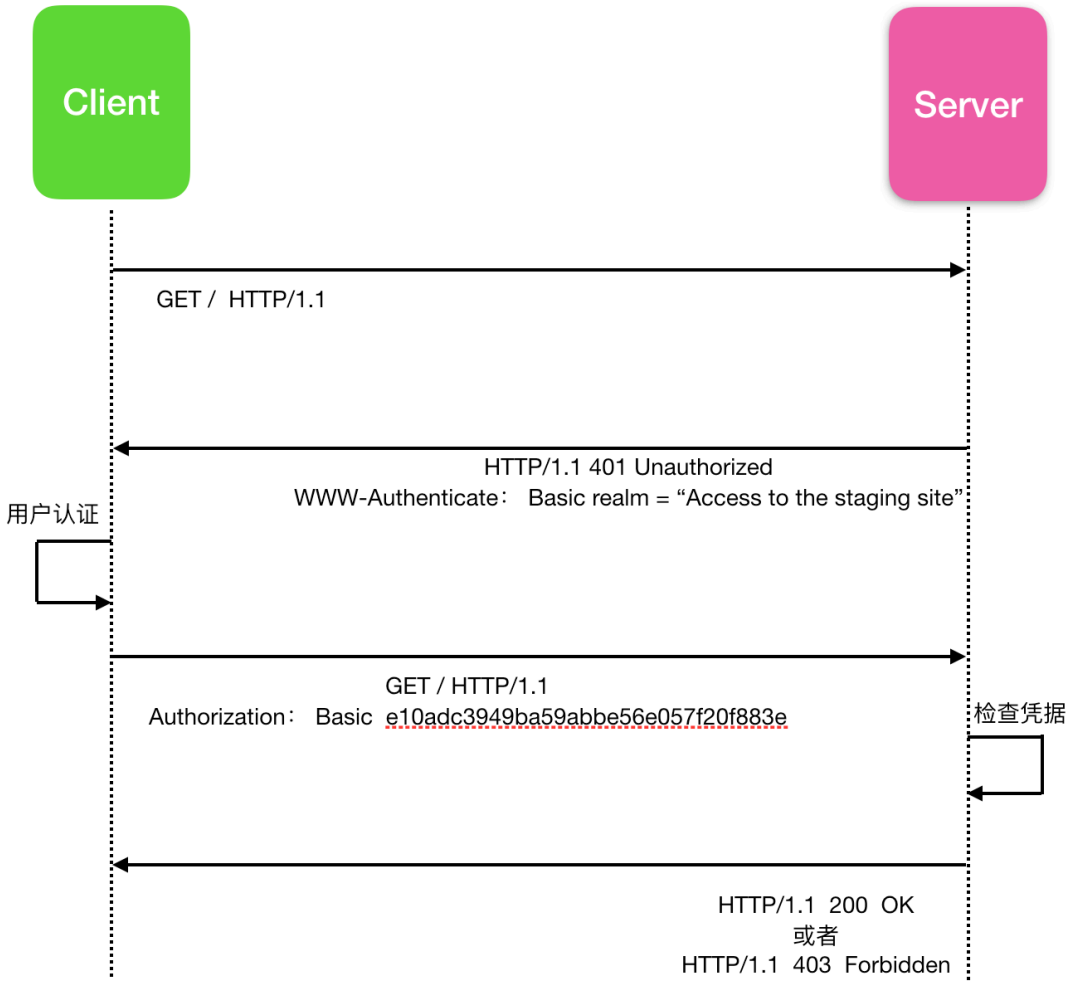
HTTP 认证

HTTP 提供了用于访问控制和身份认证的功能，下面就对 HTTP 的权限和认证功能进行介绍

通用 HTTP 认证框架

RFC 7235 定义了 HTTP 身份认证框架，服务器可以根据其文档的定义来检查客户端请求。客户端也可以根据其文档定义来提供身份验证信息。

请求/响应的工作流程如下：服务器以 `401(未授权)` 的状态响应客户端告诉客户端服务器需要认证信息，客户端提供至少一个 `www-Authenticate` 的响应标头进行授权信息的认证。想要通过服务器进行身份认证的客户端可以在请求标头字段中添加认证标头进行身份认证，一般的认证过程如下



首先客户端发起一个 HTTP 请求，不带有任何认证标头，服务器对此 HTTP 请求作出响应，发现此 HTTP 信息未带有认证凭据，服务器通过 `www-Authenticate` 标头返回 401 告诉客户端此请求未通过认证。然后客户端进行用户认证，认证完毕后重新发起 HTTP 请求，这次 HTTP 请求带有用户认证凭据（注意，整个身份认证的过程必须通过 HTTPS 连接保证安全），到达服务器后服务器会检查认证信息，如果不符合服务器认证信息，会返回 `403 Forbidden` 表示用户认证失败，如果满足认证信息，则返回 `200 OK` 。

我们知道，客户端和服务器之间的 HTTP 连接可以被代理缓存重新发送，所以认证信息也适用于代理服务器。

代理认证

由于资源认证和代理认证可以共存，因此需要不同的头和状态码，在代理的情况下，会返回状态码 `407(需要代理认证)` ， `Proxy-Authenticate` 响应头包含至少一个适用于代理的情况， `Proxy-Authorization` 请求头用于将证书提供给代理服务器。下面分别来认识一下这两个标头

Proxy-Authenticate

HTTP `Proxy-Authenticate` 响应标头定义了身份验证方法，应使用该身份验证方法来访问代理服务器后面的资源。它将请求认证到代理服务器，从而允许它进一步发送请求。例如

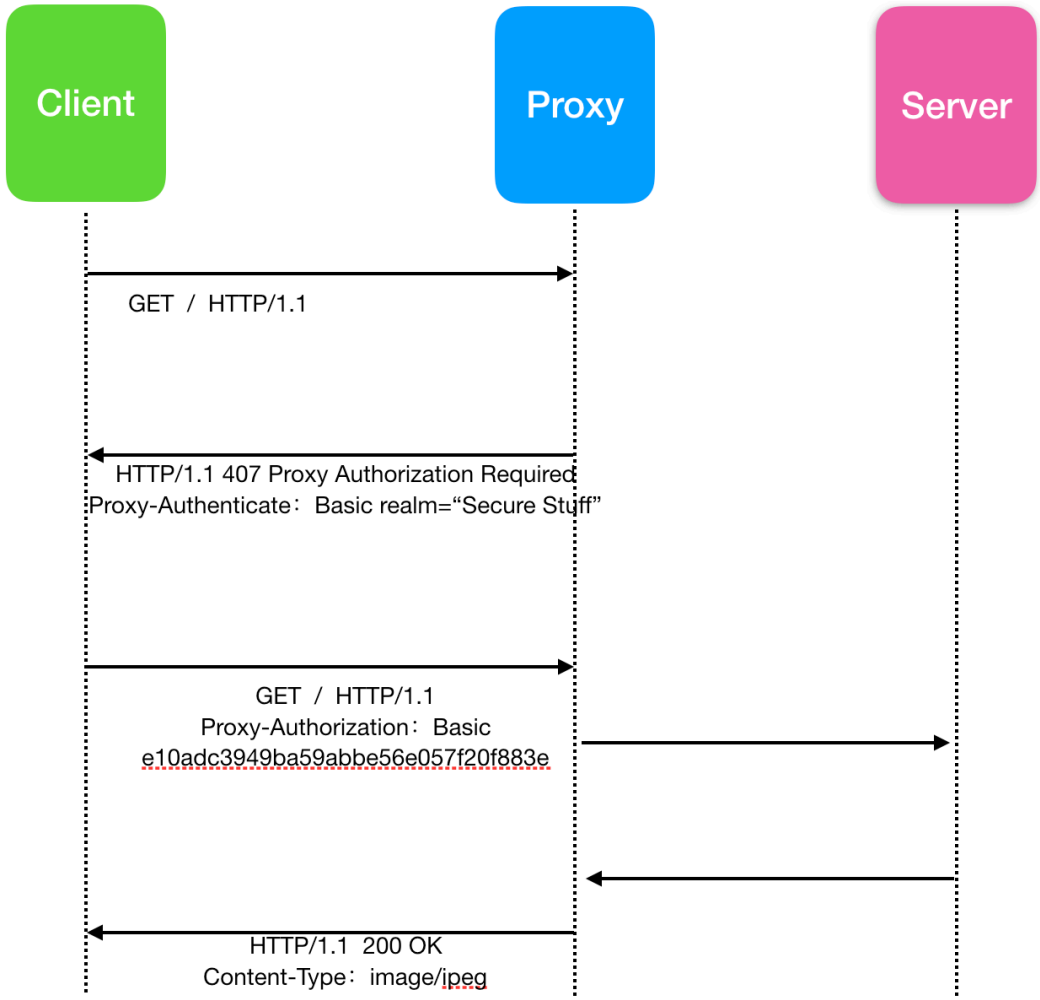
```
Proxy-Authenticate: Basic
Proxy-Authenticate: Basic realm="Access to the internal site"
```

Proxy-Authorization

这个 HTTP `请求` 标头和上面的 `Proxy-Authenticate` 拼接很相似，但是概念不同，这个标头用于向代理服务器提供凭据，例如

```
1   Proxy-Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
```

下面是代理服务器的请求/响应认证过程



这个过程和通用的过程类似，我们就不再详细展开描述了。

禁止访问

如果 `代理服务器` 收到的有效凭据不足以获取对给定资源的访问权限，则服务器应使用 `403 Forbidden` 状态代码进行响应。与 `401 Unauthorized` 和 `407 Proxy Authorization Required` 不同，该用户无法进行身份验证。

WWW-Authenticate 和 Proxy-Authenticate 头

`WWW-Authenticate` 和 `Proxy-Authenticate` 响应头定义了获得对资源访问权限的身份验证方法。他们需要指定使用哪种身份验证方案，以便希望授权的客户端知道如何提供凭据。它们的一般表示形式如下

```
WWW-Authenticate: <type> realm=<realm>
Proxy-Authenticate: <type> realm=<realm>
```

我想你从上面看到这里一定会好奇 `<type>` 和 `realm` 是什么东西，现在就来解释下。

- `<type>` 是认证协议， `Basic` 是下面协议中最普遍使用的

> RFC 7617 中定义了 `Basic` HTT P身份验证方案，该方案将凭据作为用户ID /密码对传输，并使用 base64 进行编码。(感兴趣的同学可以看看 **https://tools.ietf.org/html/rfc7617**)

其他的认证协议主要有

认证协议	参考来源
Basic	查阅 **RFC 7617**，base64编码的凭据
Bearer	查阅 **RFC 6750**，承载令牌来访问受 OAuth 2.0保护的资源
Digest	查阅 **RFC 7616**，Firefox仅支持md5哈希，请参见错误 **bug 472823** 以获得SHA加密支持
HOBA	查阅 **RFC 7486**
Mutual	查阅 **RFC 8120**
AWS4-HMAC-SHA256	查阅 **AWS docs**

- `realm` 用于描述保护区或指示保护范围，这可能是诸如 **Access to the staging site(访问登陆站点)** 或者类似的，这样用户就可以知道他们要访问哪个区域。

Authorization 和 Proxy-Authorization 标头

Authorization 和 Proxy-Authorization 请求标头包含用于通过代理服务器对用户代理进行身份验证的凭据。在此，再次需要类型，其后是凭据，取决于使用哪种身份验证方案，可以对凭据进行编码或加密。一般表示如下

```
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
Proxy-Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
```

HTTP 缓存

通过把 `请求/响应` 缓存起来有助于提升系统的性能， `Web 缓存` 减少了延迟和网络传输量，因此减少资源获取锁需要的时间。由于链路漫长，网络时延不可控，浏览器使用 HTTP 获取资源的成本较高。所以，非常有必要把数据缓存起来，下次再请求的时候尽可能地复用。当 Web 缓存在其存储中具有请求的资源时，它将拦截该请求并直接返回资源，而不是到达源服务器重新下载并获取。这样做可以实现两个小目标

- 减轻服务器负载

- 提升系统性能

下面我们就一起来探讨一下 HTTP 缓存都有哪些

不同类型的缓存

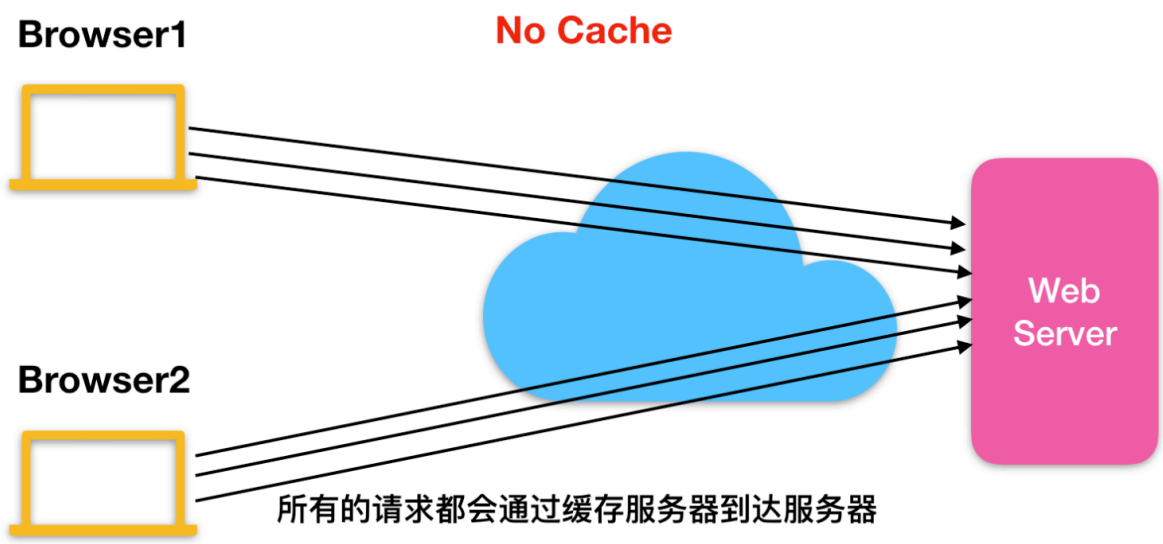
HTTP 缓存有几种不同的类型，这些可以分为两个主要类别：`私有缓存` 和 `共享缓存`。

- 共享缓存：共享缓存是一种缓存，它可以存储多个用户重复使用的请求/响应。
- 私有缓存：私有缓存也称为 `专用缓存`，它只适用于单个用户。
- 不缓存过期资源：所有的请求都会直接到达服务器，由服务器来下载资源并返回。

> 我们主要探讨 `浏览器缓存` 和 `代理缓存`，但真实情况不只有这两种缓存，还有网关缓存，CDN，反向代理缓存和负载平衡器，把它们部署在 Web 服务器上，可以提高网站和 Web 应用程序的可靠性，性能和可伸缩性。

不缓存过期资源

不缓存过期资源即浏览器和代理不会缓存过期资源，客户端发起的请求会直接到达服务器，可以使用 `no-cache` 标头代表不缓存过期资源。



no-cache 示意图

no-cache 属于 Cache-Control 通用标头，其一般的表示方法如下

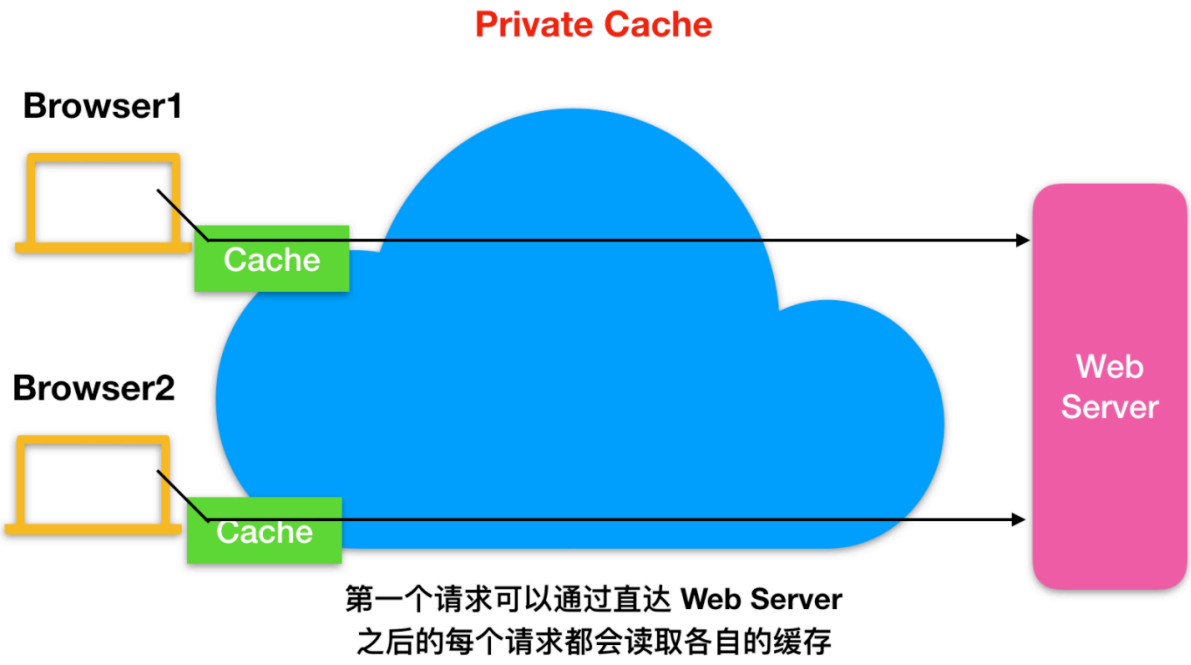
```
Cache-Control: no-cache
```

也可以使用 `max-age = 0` 来实现不缓存的效果。

```
Cache-Control: max-age=0
```

私有缓存

私有缓存只用来缓存单个用户，你可能在浏览器设置中看到了 `缓存`，浏览器缓存包含服务器通过 HTTP 下载下来的所有文档。这个高速缓存用于使访问的文档可以进行前进/后退，保存操作而无需重新发送请求到源服务器。



私有缓存示意图

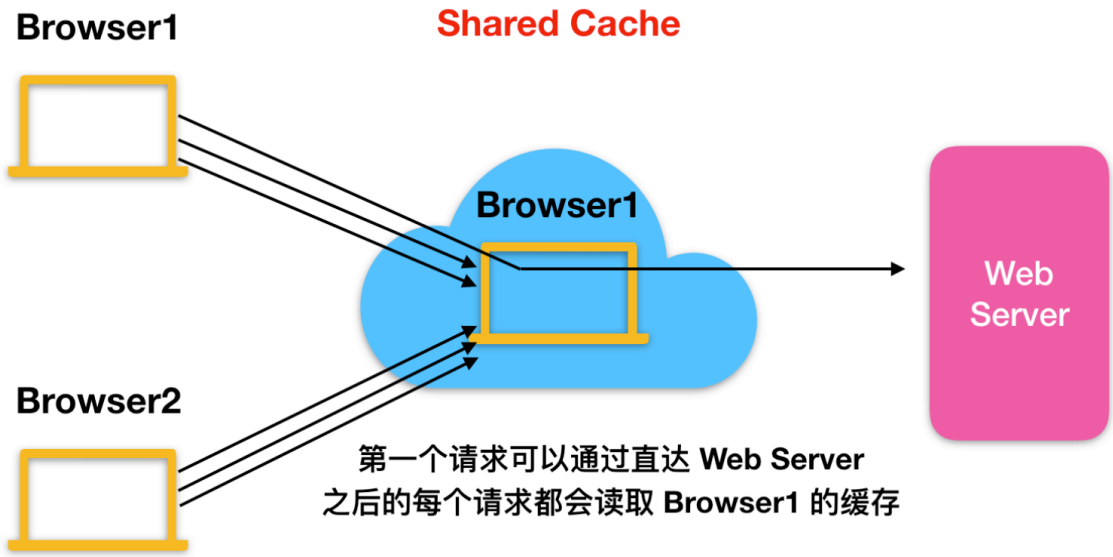
可以使用 `private` 来实现私有缓存，这与 `public` 的用法相反，缓存服务器只对特定的客户端进行缓存，其他客户端发送过来的请求，缓存服务器则不会返回缓存。它的一般表示方法如下

```
1   Cache-Control: private
```

共享缓存

共享缓存是一种用于存储要由多个用户重用的响应缓存。共享缓存一般使用 `public` 来表示， `public` 属性只出现在客户端响应中，表示响应可以被任何缓存所缓存。一般表示方法如下

```
1   Cache-Control: public
```



共享缓存示意图

缓存控制

HTTP/1.1 中的 `Cache-Control` 常规标头字段用于执行缓存控制，使用此标头可通过其提供的各种指令来定义缓存策略。下面我们依次介绍一下这些属性

不缓存

`no-store` 才是真正意义上的 `不缓存`，每次服务器接受到客户端的请求后，都会返回最新的资源给客户端。

```
Cache-Control: no-store
```

缓存但需要验证

同上面的 不缓存过期资源

私有和共享缓存

同上

缓存过期

缓存中一个很重要的指令就是 `max-age`，这是资源被视为 `新鲜` 的最长时间，与 `Expires` 相反，此指令是相对于请求时间的。对于应用程序中不会更改的文件，通常可以添加主动缓存。下面是 mag-age 的表示

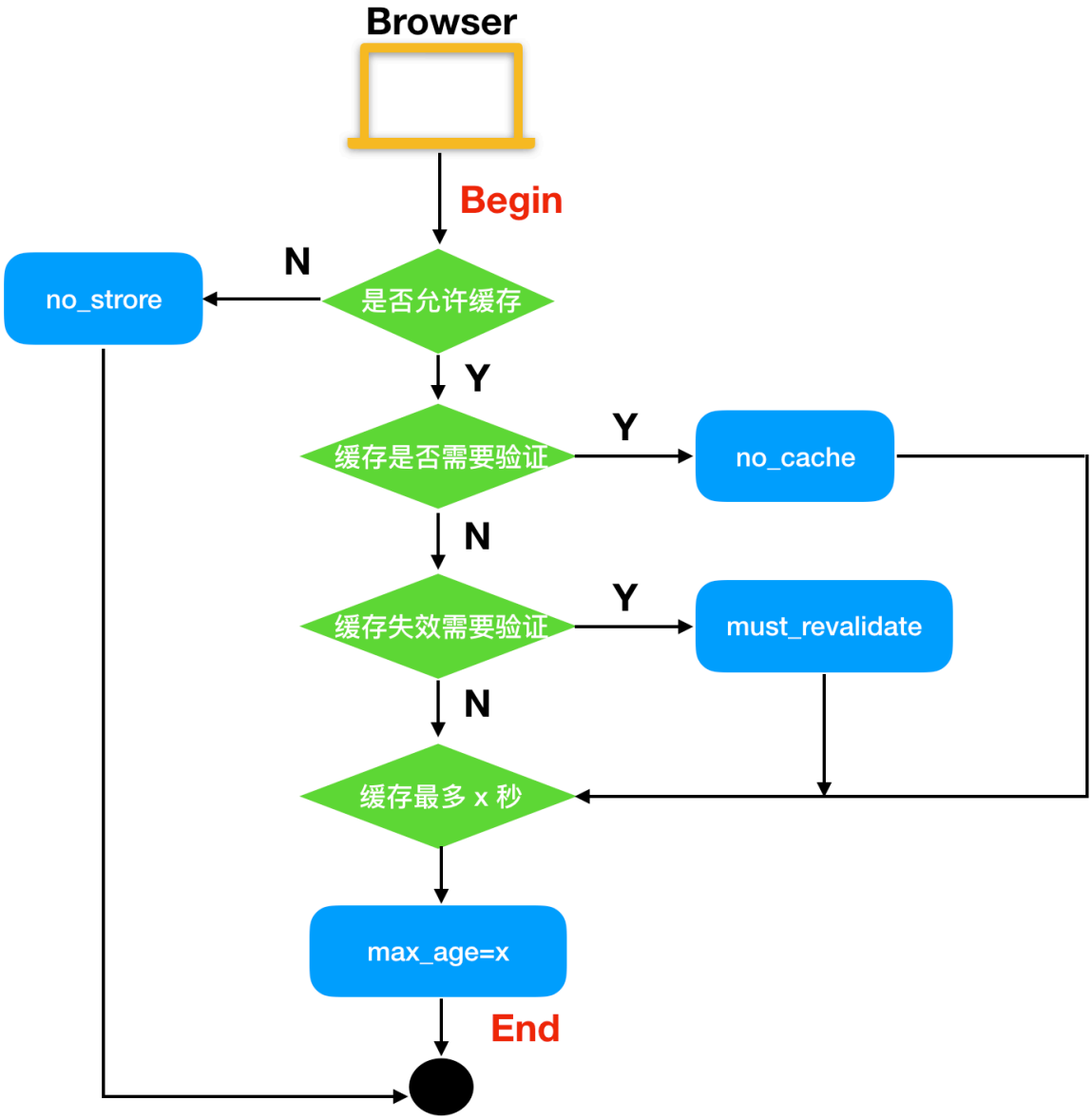
```
Cache-Control: max-age=31536000
```

缓存验证

`must-revalidate` 表示缓存必须在使用之前验证过时资源的状态，并且不应使用过期的资源。

```
Cache-Control: must-revalidate
```

下面是一个缓存验证图

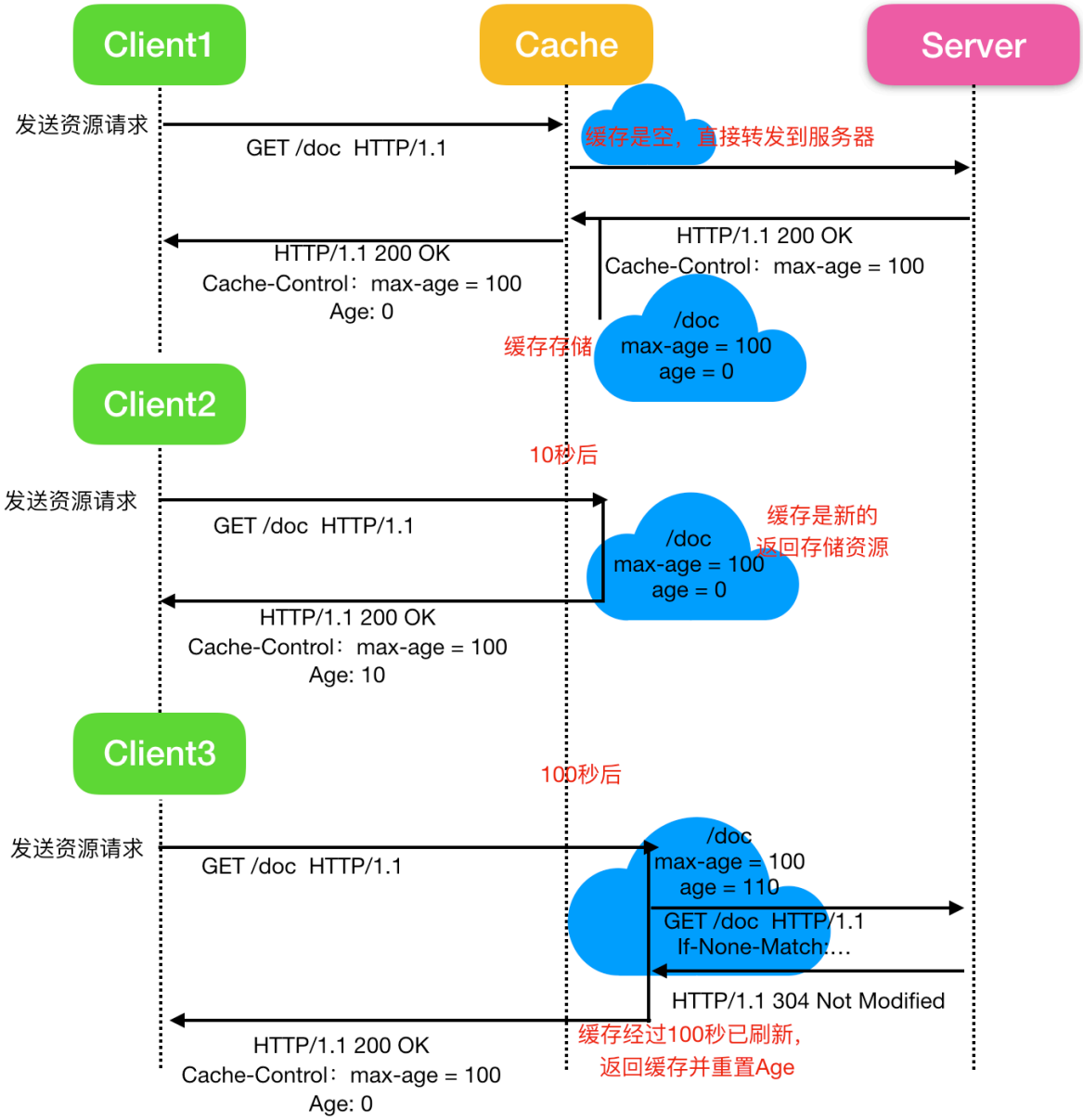


什么是新鲜的数据

一旦资源存储在缓存中，理论上就可以永远被缓存使用。但是不管是浏览器缓存还是代理缓存，其存储空间是有限的，所以缓存会定期进行清除，这个过程叫做 `缓存回收(cache eviction)` （自译）。另一方面，服务器上的缓存也会定期进行更新，HTTP 作为应用层的协议，它是一种 `客户-服务器` 模式，HTTP 是无状态的协议，因此当资源发生更改时，服务器无法通知缓存和客户端。因此服务器必须通过某种方式告知客户端缓存已经被更新。服务器会提供 `过期时间` 这个概念，告知客户端在此到期时间之前，资源是 `新鲜的` ，也就是未更改过的。在此到期时间的范围之外，资源已过时。 `过期算法(Eviction algorithms)` 通常会将新资源优先于陈旧资源使用。

这里需要注意一下，过期的资源并不会被回收或忽略，当高速缓存接收到过期资源时，它会使用 `If-None-Match` 转发此请求，以检查它是否仍然有效。如果有效，服务器会返回 `304 Not Modified` 响应头并且没有任何响应体，从而节省了一些带宽。

下面是使用共享缓存代理的过程



这个图应该比较好理解，只说一下 Age 的作用，Age 是 HTTP 响应标头告诉客户端源服务器在多久之前创建了响应，它的单位为 `秒`，Age 标头通常接近于0，如果是0则可能是从源服务器获取的，如果不是表示可能是由代理服务器创建，那么 Age 的值表示的是**缓存后的响应再次发起认证到认证完成的时间值**。

缓存的有效性是由多个标头来共同决定的，而并非某一个标头来决定。如果指定了 `Cache-control:max-age=N` ，那么缓存会保存 N 秒。如果这个通用标头不存在的话，则会检查是否存在 `Expires` 标头。如果 Exprires 标头存在，那么它的值减去 Date 标头的值就可以确定其有效性。最后，如果 `max-age` 和 `expires` 都不存在，就去寻找 `Last-Modified` 标头，如果存在此标头，则高速缓存的有效性等于 Date 标头的值减去 Last-modified 标头的值除以10。

缓存验证

当到达缓存资源的有效期时，将对其进行验证或再次获取。仅当服务器提供了 `强验证器` 或 `弱验证器` 时，才可以进行验证。

当用户按下重新加载按钮时，将触发重新验证。如果缓存的响应包含 `Cache-control: must-revalidate` 标头，则在正常浏览下也会触发该事件。另一个因素是 高级 -> 缓存首选项 面板中的缓存验证首选项。有一个选项可在每次加载文档时强制进行验证。

Etag

我们上面提到了强验证器和弱验证器，实现验证器功能的标头正式 Etag 的作用，这意味着 HTTP 用户代理（例如浏览器）不知道该字符串表示什么，并且无法预测其值。如果 Etag 标头是资源响应的一部分，则客户端可以在未来请求的标头中发出 `If-None-Match` ，以验证缓存的资源。

`Last-Modified` 响应标头可以用作弱验证器，因为它只有1秒可以分辨的时间。如果响应中存在 `Last-Modified` 标头，则客户端可以发出 `If-Modified-Since` 请求标头来验证缓存资源。（关于 Etag 更多我们会在条件请求介绍）

避免碰撞

通过使用 Etag 和 If-Match 标头，你可以检测避免碰撞。

例如，在编辑 MDN 时，将对当前 Wiki 内容进行哈希处理并将其放入响应中的 Etag 中

```
1   Etag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

当将更改保存到 Wiki 页面（发布数据）时，POST 请求将包含 If-Match 标头，其中包含 Etag 值以检查有效性。

```
1   If-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

如果哈希值不匹配，则表示文档已在中间进行了编辑，并返回 `412 Precondition Failed` 错误。

缓存未占用资源

Etag 标头的另一个典型用法是缓存未更改的资源，如果用户再次访问给定的 URL（已设置Etag），并且该 URL过时，则客户端将在 If-None-Match 标头字段中发送其 Etag 的值

```
1   If-None-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

服务器将客户端的 Etag（通过 If-None-Match 发送）与 Etag 进行比较，以获取其当前资源版本，如果两个值都匹配（即资源未更改），则服务器会发回 `304 Not Modified` 状态，没有主体，它告诉客户端响应的缓存仍然可以使用。

HTTP CROS 跨域

CROS 的全称是 `Cross-Origin Resource Sharing(CROS)` ，中文译为 `跨域资源共享` ，它是一种机制。是一种什么机制呢？它是一种让运行在一个 `域(origin)` 上的 Web 应用被准许访问来自不同源服务器上指定资源的机制。在搞懂这个机制前，你需要线了解什么是 `域(origin)`

Origin

Web 概念中 `域(Origin)` 的内容由 `scheme(protocol) - 协议` ， `host(domain) - 主机` 和用于访问它的 URL `port - 端口` 定义。仅仅当 scheme 、host、port 都匹配时，两个对象才有相同的来源。这种协议相同，域名相同，端口相同的安全策略也被称为 `同源策略（Same Origin Policy）` 。某些操作仅限于具有相同来源的内容，可以使用 CORS 取消此限制。

跨域的特点

- 下面是跨域问题的例子，看看你是否清楚什么是跨域了

```
(1) http://example.com/app1/index.html
(2) http://example.com/app2/index.html
```

上面这两个 URL 是否具有跨域问题呢?

上面两个 URL 是不具有跨域问题的，因为这两个 URL 具有相同的 `协议(scheme)` 和 `主机(host)`

- 那么下面这两个是否具有跨域问题呢?

```
http://Example.com:80
http://example.com
```

这两个 URL 也不具有跨域问题，为什么不具有，端口不一样啊。其实它们两个端口是一样的。

或许你会认为这两个 URL 是不一样的，放心，关于一样不一样的论据我给你抛出来了

> 协议和域名部分是不区分大小写的，但是路径部分则根据服务器平台而定。Windows 和 Mac OS X 系统是不区分大小写的，而采用UNIX和Linux系的服务器系统是区分大小写的，

也就是说上面的 `Example.com` 和 `example.com` 其实是一个网址，并且由于两个地址具有相同的 scheme 和 host ，默认情况下服务器通过端口80传递 HTTP 内容，所以上面这两个地址也是相同的。

- 下面这两个 URL 地址是否具有跨域问题?

```
http://example.com/app1
https://example.com/app2
```

这两个 URL 的 scheme 不同，所以这两个 URL 具有跨域问题

- 再看下面这三个 URL 是否具有跨域问题

```
http://example.com
http://www.example.com
http://myapp.example.com
```

这三个 URL 也是具有跨域问题的，因为它们隶属于不通服务器的主机 host。

- 下面这两个 URL 是否具有跨域问题

```
http://example.com
http://example.com:8080
```

这两个 URL 也是具有跨域问题，因为这两个 URL 的默认端口不一样。

同源策略

处于安全的因素，浏览器限制了从脚本发起跨域的 HTTP 请求。 `XMLHttpRequest` 和其他 `Fetch 接口` 会遵循 `同源策略(same-origin policy)` 。也就是说使用这些 API 的应用程序想要请求相同的资源，那么他们应该具有相同的来源，除非来自其他来源的响应包括正确的 CORS 标头也可以。

同源策略是一种很重要的安全策略，它限制了从一个来源加载的文档或脚本如何与另一个来源的资源进行交互。 它有助于隔离潜在的恶意文档，减少可能的攻击媒介。

我们上面提到，如果两个 URL 具有相同的协议、主机和端口号（如果指定）的话，那么两个 URL 具有相同的来源。下面有一些实例，你判断一下是不是具有相同的来源

目标来源 `http://store.company.com/dir/page.html`

URL	Outcome	Reason
http://store.company.com/dir2/other.html	相同来源	只有path不同
http://store.company.com/dir/inner/another.html	相同来源	只有path不同
https://store.company.com/page.html	不同来源	协议不通
http://store.company.com:81/dir/page.html	不同来源	默认端口不同
http://news.company.com/dir/page.html	不同来源	主机不同

现在我带你认识了两遍不同的源，现在你应该知道如何区分两个 URL 是否属于同一来源了吧！

好，你现在知道了什么是跨域问题，现在我要问你，哪些请求会产生跨域请求呢？这是我们下面要讨论的问题

跨域请求

跨域请求可能会从下面这几种请求中发出：

1. 调用 `XMLHttpRequest` 或者 `Fetch` api。

XMLHttpRequest 是什么？（我是后端程序员，前端不太懂，简单解释下，如果解释的不好，还请前端大佬们不要胖揍我）

所有的现代浏览器都有一个内置的 `XMLHttpReqeust` 对象，这个对象可以用于从服务器请求数据。

XMLHttpReqeust 对于开发人员来说很重要，XMLHttpReqeust 对象可以用来做下面这些事情

- 更新网页无需重新刷新页面
- 页面加载后从服务器请求数据
- 页面加载后从服务端获取数据
- 在后台将数据发送到服务器

使用 XMLHttpRequest(XHR) 对象与服务器进行交互，你可以从 URL 检索数据从而不必刷新整个页面，这使网页可以更新页面的一部分，而不会中断用户的操作。XMLHttpRequest 在 `AJAX` 异步编程中使用很广泛。

再来说一下 Fetch API 是什么，Fetch 提供了请求和响应对象（以及其他网络请求）的通用定义。它还提供了相关概念的定义，例如 CORS 和 HTTP Origin 头语义，并在其他地方取代了它们各自的定义。

2. Web 字体（用于 CSS 中@ font-face中的跨域字体使用），以便服务器可以部署 TrueType 字体，这些字体只能由允许跨站点加载和使用的网站使用。
3. WebGL 纹理
4. 使用 `drawImage()` 绘制到画布上的图像/视频帧
5. 图片的 CSS 形状

跨域功能概述

跨域资源共享标准通过添加新的 HTTP 标头来工作，这些标头允许服务器描述允许哪些来源从 Web 浏览器读取信息。另外，对于可能导致服务器数据产生副作用的 HTTP 请求方法（尤其是 GET 或者具有某些 MIME 类型 POST 方法以外 HTTP 方法），该规范要求浏览器 `预检` 请求，使用 HTTP OPTIONS 请求方法从服务器请求受支持的方法，然后在服务器 `批准` 后发送实际请求。服务器还可以通知客户端是否应与请求一起发送 `凭据` （例如 Cookies 和 HTTP 身份验证）。

> 注意：CORS 故障会导致错误，但是出于安全原因，该错误的详细信息不适用于 JavaScript。 所有代码都知道发生了错误。 确定具体出问题的唯一方法是查看浏览器的控制台以获取详细信息。

访问控制

下面我会和大家探讨三种方案，这些方案都演示了跨域资源共享的工作方式。所有这些示例都使用 XMLHttpRequest，它可以在任何支持的浏览器中发出跨站点请求。

简单请求

一些请求不会触发 `CORS预检` （关于预检我们后面再介绍）。 `简单请求` 是满足一下所有条件的请求

- 允许以下的方法： `GET` 、 `HEAD` 和 `POST`
- 除了由用户代理自动设置的标头（例如 Connection、User-Agent 或者在 Fetch 规范中定义为禁止标头名称的其他标头）外，唯一允许手动设置的标头是那些 Fetch 规范将其定义为 `CORS安全列出的请求标头` ，它们是：
    - Accept
    - Accept-Language
    - Content-Language
    - Content-Type（下面会介绍）
    - DPR
    - Downlink
    - Save-Data
    - Viewport-Width
    - Width
- Content-Type 标头的唯一允许的值是
    - application/x-www-form-urlencoded
    - multipart/form-data
    - text/plain

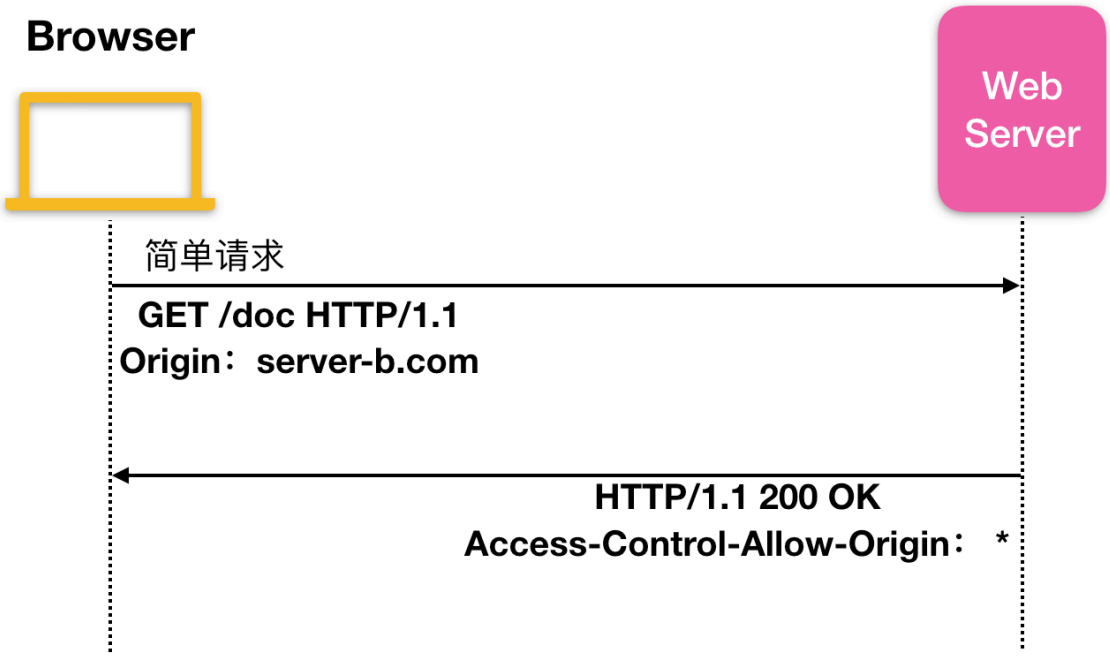
- 没有在请求中使用的任何 XMLHttpRequestUpload 对象上注册事件侦听器；这些可以使用 XMLHttpRequest.upload 属性进行访问。
- 请求中未使用 ReadableStream对象。

  例如，假定 web 内容 `https://foo.example` 想要获取 `https://bar.other` 域的资源，那么 JavaScript 中的代码可能会像下面这样写

```
const xhr = new XMLHttpRequest();
const url = 'https://bar.other/resources/public-data/';

xhr.open('GET', url);
xhr.onreadystatechange = someHandler;
xhr.send();
```

这使用 CORS 标头来处理特权，从而在客户端和服务器之间执行某种转换。



让我们看看在这种情况下浏览器将发送到服务器的内容，并让我们看看服务器如何响应：

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: https://foo.example
```

注意请求的标头 Origin ， 它表明调用来自于 `https://foo.example` 。让我们看看服务器是如何响应的

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[…XML Data…]
```

服务端发送 `Access-Control-Allow-Origin` 作为响应。使用 `Origin` 标头和 `Access-Control-Allow-Origin` 展示了最简单的访问控制协议。在这个事例中，服务端使用 `Access-Control-Allow-Origin` 作为响应，也就说明该资源可以被任何域访问。

如果位于 `https://bar.other` 的资源所有者希望将对资源的访问限制为仅来自 `https://foo.example` 的请求，他们应该发送如下响应

```
Access-Control-Allow-Origin: https://foo.example
```

现在除了 `https://foo.example` 之外的任何域都无法以跨域方式访问到 `https://bar.other` 的资源。

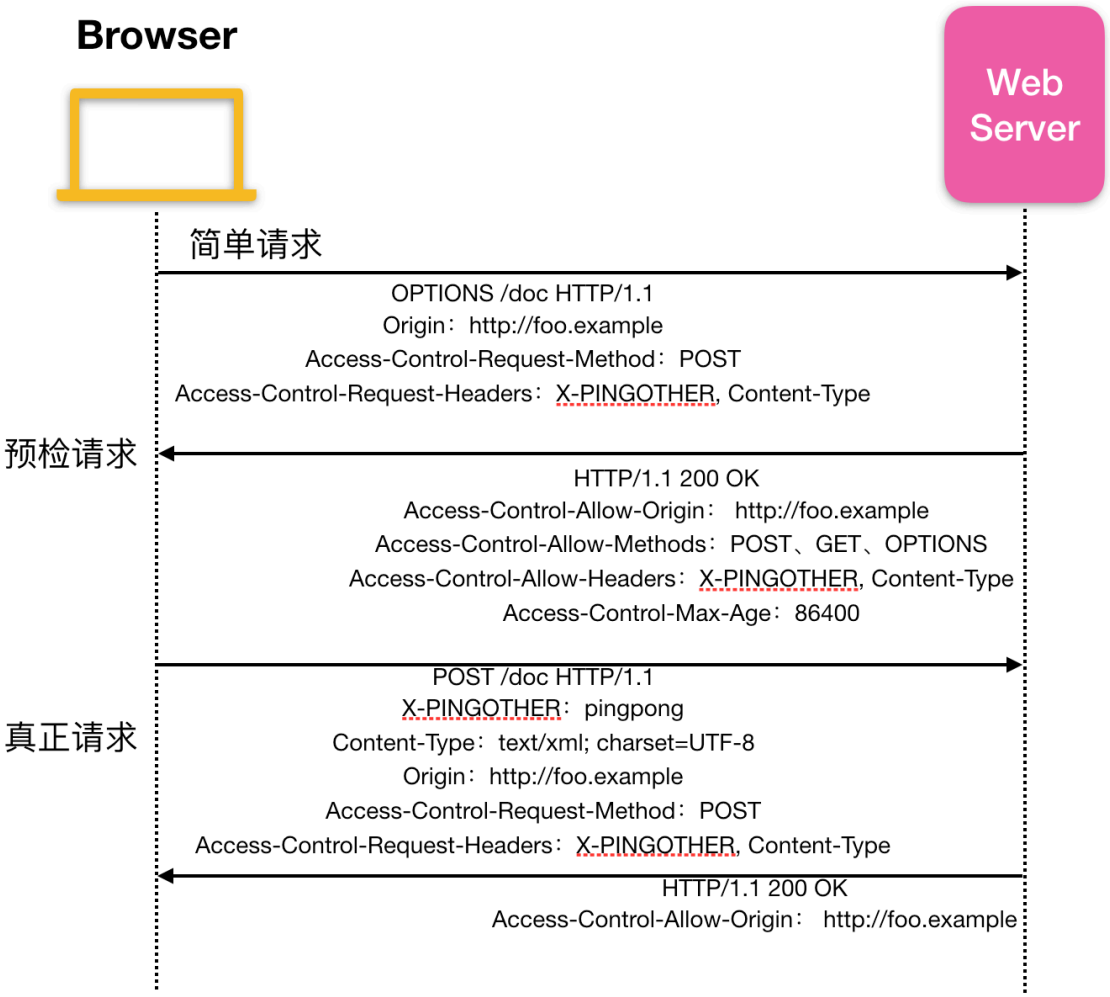
预检请求

和上面探讨的简单请求不同， `预检` 请求首先通过 `OPTIONS` 方法向另一个域上的资源发送 HTTP 请求，用来确定实际请求是否可以安全的发送。跨站点这样被 `预检` ，因为它们可能会影响用户数据。

下面是一个预检事例

```
const xhr = new XMLHttpRequest();
xhr.open('POST', 'https://bar.other/resources/post-here/');
xhr.setRequestHeader('X-PINGOTHER', 'pingpong');
xhr.setRequestHeader('Content-Type', 'application/xml');
xhr.onreadystatechange = handler;
xhr.send('<person><name>Arun</name></person>');
```

上面的事例创建了一个 XML 请求体用来和 POST 请求一起发送。此外，设置了非标准请求头 `X-PINGOTHER` ，这个标头不是 HTTP/1.1 的一部分，但通常对 Web 程序很有用。由于请求的 `Content-Type` 使用 `application/xml` ，并且设置了自定义标头，因此该请求被 `预检` 。如下图所示



> 如下所述，实际的 POST 请求不包含 Access-Control-Request- * 标头；只有 OPTIONS 请求才需要它们。

下面我们来看一下完整的客户端/服务器交互，首先是预检请求/响应

```
OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type

```

```
HTTP/1.1 204 No Content
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
```

上面的1 -11 行代表预检请求，预检请求使用 `OPYIIONS` 方法，浏览器根据上面的 JavaScript 代码段所使用的请求参数确定是否需要发送此请求，以便服务器可以响应是否可以使用实际请求参数发送请求。OPTIONS 是一种 HTTP / 1.1方法，用于确定来自服务器的更多信息，并且是一种安全的方法，这意味着它不能用于更改资源。请注意，与 OPTIONS 请求一起，还发送了另外两个请求标头（分别是第9行和第10行）

```
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

`Access-Control-Request-Method` 标头作为预检请求的一部分通知服务器，当发送实际请求时，将使用 `POST` 请求方法发送该请求。

`Access-Control-Request-Headers` 标头通知服务器，当发送请求时，它将与X-PINGOTHER 和 Content-Type 自定义标头一起发送。服务器可以确定这种情况下是否接受请求。

下面的 1 - 11行是服务器发回的响应，表示 `POST` 请求和 `X-PINGOTHER` 是可以接受的，我们着重看一下下面这几行

```
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
```

服务器完成响应表明源 `http://foo.example` 是可以接受的 URL，能够允许 `POST、GET、OPTIONS` 进行请求，允许自定义标头 `X-PINGOTHER, Content-Type` 。最后， `Access-Control-Max-Age` 以秒为单位给出一个值，这个值表示对预检请求的响应可以缓存多长时间，在此期间内无需发送其他预检请求。

完成预检请求后，将发送实际请求：

```
POST /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Referer: https://foo.example/examples/preflightInvocation.html
```

```
Content-Length: 55
Origin: https://foo.example
Pragma: no-cache
Cache-Control: no-cache

<person><name>Arun</name></person>
```

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
Keep-Alive: timeout=2, max=99
Connection: Keep-Alive
Content-Type: text/plain

[Some GZIP'd payload]
```

正式响应中很多标头我们在之前的文章已经探讨过了，本篇不再做详细的介绍，读者可以参考 **你还在为 HTTP 的这些概念头疼吗？** 查阅

带凭证的请求

XMLHttpRequest 或 Fetch 和 CORS 最有趣的功能就是能够发出知道 HTTP Cookie 和 HTTP 身份验证的 `凭证` 请求。默认情况下，在跨站点 XMLHttpRequest 或 Fetch 调用中，浏览器将不发送凭据。调用 XMLHttpRequest对象或 Request 构造函数时必须设置一个特定的标志。

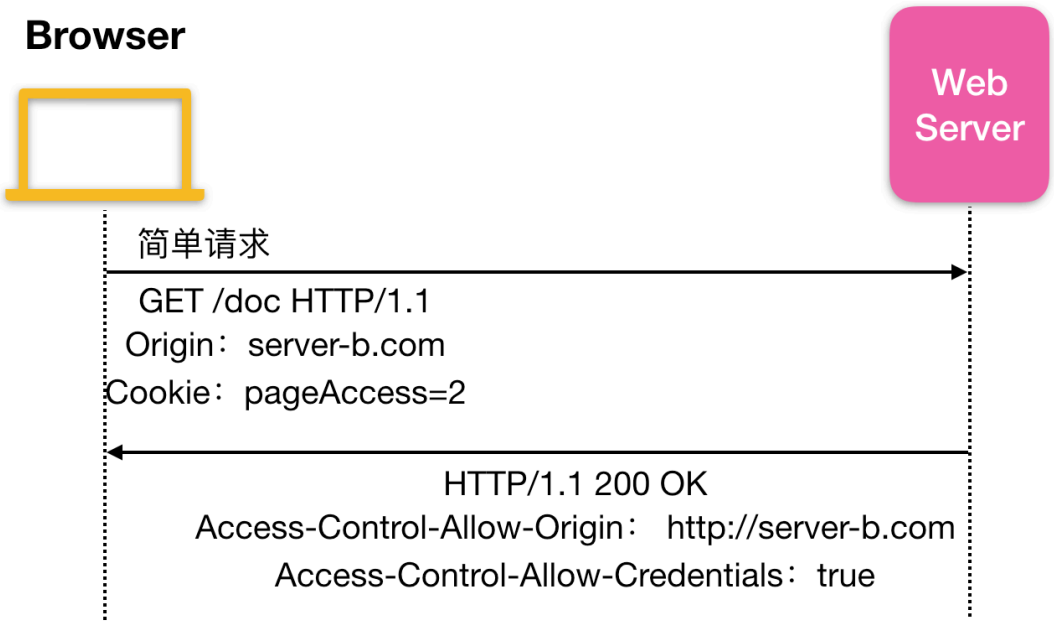
在下面这个例子中，最初从 `http://foo.example` 加载的内容对设置了 Cookies 的 `http://bar.other` 上的资源进行了简单的 GET 请求， foo.example 上可能的代码如下

```
const invocation = new XMLHttpRequest();
const url = 'http://bar.other/resources/credentialed-content/';

function callOtherDomain() {
  if (invocation) {
    invocation.open('GET', url, true);
    invocation.withCredentials = true;
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```

第7行显示 XMLHttpRequest 上的标志，必须设置该标志才能使用 Cookie 进行调用。默认情况下，调用是不在使用 Cookie 的情况下进行的。由于这是一个简单的 GET 请求，因此不会进行预检，但是浏览器将拒绝任何没有 Access-Control-Allow-Credentials 的响应：标头为true，指的是响应不会返回 web 页面的内容。

上面的请求用下图可以表示



这是客户端和服务器之间的示例交换：

```
GET /resources/access-control-with-credentials/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Referer: http://foo.example/examples/credential.html
Origin: http://foo.example
Cookie: pageAccess=2
```

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://foo.example
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=3; expires=Wed, 31-Dec-2008 01:34:53 GMT
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 106
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain


[text/plain payload]
```

上面第10行包含指向 `http://bar.other` 上的内容 Cookie，但是如果 bar.other 没有以 `Access-Control-Allow-Credentials:true` 响应（下面第五行），响应将被忽略，并且不能使用网站返回的内容。

请求凭证和通配符

当回应凭证请求时，服务器必须在 `Access-Control-Allow-Credentials` 中指定一个来源，而不能直接写 `*` 通配符

因为上面示例代码中的请求标头包含 Cookie 标头，如果 `Access-Control-Allow-Credentials` 中是指定的通配符 `*` 的话，请求会失败。

注意上面示例中的 `Set-Cookie` 响应标头还设置了另外一个值，如果发生故障，将引发异常（取决于所使用的API）。

HTTP 响应标头

下面会列出一些服务器跨域共享规范定义的 HTTP 标头，上面简单概述了一下，现在一起来认识一下，主要会介绍下面这些

- Access-Control-Allow-Origin
- Access-Control-Allow-Credentials
- Access-Control-Allow-Headers
- Access-Control-Allow-Methods
- Access-Control-Expose-Headers
- Access-Control-Max-Age
- Access-Control-Request-Headers
- Access-Control-Request-Method
- Origin

Access-Control-Allow-Origin

`Access-Control-Allow-Origin` 是 HTTP 响应标头，指示响应是否能够和给定的源共享资源。Access-Control-Allow-Origin 指定单个资源会告诉浏览器允许指定来源访问资源。对于没有凭据的请求 `*` 通配符，告诉浏览器允许任何源访问资源。

例如，如果要允许源 `https://mozilla.org` 的代码访问资源，可以使用如下的指定方式

```
1   Access-Control-Allow-Origin: https://mozilla.org
2   Vary: Origin
```

如果服务器指定单个来源而不是 `*` 通配符，则服务器还应在 Vary 响应标头中包含该来源。

Access-Control-Allow-Credentials

`Access-Control-Allow-Credentials` 是 HTTP 的响应标头，这个标头告诉浏览器，当包含凭证请求（Request.credentials）时是否将响应公开给前端 JavaScript 代码。

这时候你会问到 `Request.credentials` 是什么玩意？不要着急，来给你看一下，首先来看 Request 是什么玩意，

实际上，Request 是 Fetch API 的一类接口代表着资源请求。一般创建 Request 对象有两种方式

- 使用 Request() 构造函数创建一个 Request 对象
- 还可以通过 FetchEvent.request api 操作来创建

再来说下 Request.credentials 是什么意思，Request 接口的凭据只读属性指示在跨域请求的情况下，用户代理是否应从其他域发送 cookie。（其他 Request 对象的方法详见 **https://developer.mozilla.org/en-US/docs/Web/API/Request**）

当发送的是凭证模式的请求包含 （Request.credentials）时，如果 Access-Control-Allow-Credentials 值为 true，浏览器将仅向前端 JavaScript 代码公开响应。

```
1   Access-Control-Allow-Credentials: true
```

凭证一般包括 **cookie、认证头和 TLS 客户端证书**

> 当用作对预检请求响应的一部分时，这表明是否可以使用凭据发出实际请求。注意简单的 `GET` 请求不会进行预检。

可以参考一个实际的例子 **https://www.jianshu.com/p/ea485e5665b3**

Access-Control-Allow-Headers

`Access-Control-Allow-Headers` 是一个响应标头，这个标头用来响应预检请求，它发出实际请求时可以使用哪些HTTP标头。

示例

- 自定义标头

这是 Access-Control-Allow-Headers 标头的示例。它表明除了像 CROS 安全列出的请求标头外，对服务器的 CROS 请求还支持名为 `X-Custom-Header` 的自定义标头。

```
1   Access-Control-Allow-Headers: X-Custom-Header
```

- 多个标头

这个例子展示了 Access-Control-Allow-Headers 如何使用多个标头

```
1   Access-Control-Allow-Headers: X-Custom-Header, Upgrade-Insecure-Requests
```

- 绕过其他限制

尽管始终允许使用 CORS 安全列出的请求标头，并且通常不需要在 Access-Control-Allow-Headers 中列出这些标头，但是无论如何列出它们都将绕开适用的其他限制。

```
1   Access-Control-Allow-Headers: Accept
```

这里你可能会有疑问，哪些是 CORS 列出的安全标头？（别嫌累，就是这么麻烦）

有下面这些 **Accep、Accept-Language、Content-Language、Content-Type** ，当且仅当包含这些标头时，无需在 CORS 上下文中发送预检请求。

Access-Control-Allow-Methods

`Access-Control-Allow-Methods` 也是响应标头，它指定了哪些访问资源的方法可以使用预检请求。例如

```
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Methods: *
```

Access-Control-Expose-Headers

Access-Control-Expose-Headers 响应标头表明哪些标头可以作为响应的一部分公开。默认情况下，仅公开6个CORS安全列出的响应标头，分别是

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

如果希望客户端能够访问其他标头，则必须使用 Access-Control-Expose-Headers 标头列出它们。下面是示例

要公开非 CORS 安全列出的请求标头，可以像如下这样指定

```
Access-Control-Expose-Headers: Content-Length
```

要另外公开自定义标头，例如 X-Kuma-Revision，可以指定多个标头，并用逗号分隔

```
Access-Control-Expose-Headers: Content-Length, X-Kuma-Revision
```

在不是凭证请求中，你还可以使用通配符

```
Access-Control-Expose-Headers: *
```

但是，这不会通配 `Authorization` 标头，因此如果需要公开它，则需要明确列出

```
Access-Control-Expose-Headers: *, Authorization
```

Access-Control-Max-Age

Access-Control-Max-Age 响应头表示预检请求的结果可以缓存多长时间，例如

```
Access-Control-Max-Age: 600
```

表示预检请求可以缓存10分钟

Access-Control-Request-Headers

浏览器在发出预检请求时使用 Access-Control-Request-Headers 请求标头，使服务器知道在发出实际请求时客户端可能发送的 HTTP 标头。

```
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

Access-Control-Request-Method

同样的，Access-Control-Request-Method 响应标头告诉服务器发出预检请求时将使用那种 HTTP 方法。此标头是必需的，**因为预检请求始终是 OPTIONS**，并且使用的方法与实际请求不同。

```
Access-Control-Request-Method: POST
```

Origin

Origin 请求标头表明匹配的来源，它不包含任何信息，仅仅包含服务器名称，它与 CORS 请求以及 POST 请求一起发送，它类似于 `Referer` 标头，但与此标头不同，它没有公开整个路径。例如

```
Origin: https://developer.mozilla.org
```

HTTP 条件请求

HTTP 具有条件请求的概念，通过比较资源更新生成的值与验证器的值进行比较，来确定资源是否进行过更新。这样的请求对于验证缓存的内容、条件请求、验证资源的完整性来说非常重要。

原则

HTTP 条件请求是根据特定标头的值执行不同的请求，这些标头定义了一个前提条件，如果前提条件匹配或不匹配，则请求的结果将有所不同。

- 对于 `安全` 的方法，像是 `GET` 、用于请求文档的资源，仅当条件请求的条件满足时发回文档资源，所以，这种方式可以节约带宽。

> 什么是安全的方法，对于 HTTP 来说，**安全的方法是不会改变服务器状态的方法**，换句话说，如果方法只是只读操作，那么它肯定是安全的方法，比如说 GET 请求，它肯定是安全的方法，因为它只是请求资源。几种常见的方法肯定是安全的，它们是 **GET、HEAD和 OPTIONS**。所有安全的方法都是 `幂等的` （这他妈幂等又是啥意思？）但不是所有幂等的方法都是安全的，例如 PUT 和 DELETE 都是幂等的，但不安全。
>
> 幂等性：如果相同的客户端发起一次或者多次 HTTP 请求会得到相同的结果，则说明 HTTP 是幂等的。（我们这次不深究幂等性）

- 对于 `非安全` 的方法，像是 PUT，只有原始文档与服务器上存储的资源相同时，才可以使用条件请求来传输文档。（PUT 方法通常用来传输文件，就像 FTP 协议的文件上传一样）

验证

所有的条件请求都会尝试检查服务器上存储的资源是否与某个特定版本的资源相匹配。为了满足这种情况，条件请求需要指示资源的版本。由于无法和整个文件逐个字符进行比较，因此需要把整个文件描绘成一个值，然后把此值和服务器上的资源进行比较，这种方式称为比较器，比较器有两个条件

- 文档的最后修改日期
- 一个不透明的字符串，用于唯一标识每个版本，称为实体标签或 `Etag` 。

比较两个资源是否时相同的版本有些复杂，根据上下文，有两种相等性检查

- 当期望的是字节对字节进行比较时，例如在恢复下载时，使用 `强 Etag` 进行验证

- 当用户代理需要比较两个资源是否具有相同的内容时，使用 `若 Etag` 进行验证

HTTP 协议默认使用 `强验证`，它指定何时进行弱验证

强验证

强验证保证的是 `字节` 级别的验证，严格的验证非常严格，可能在服务器级别难以保证，但是它能够保证任何时候都不会丢失数据，但这种验证丢失性能。

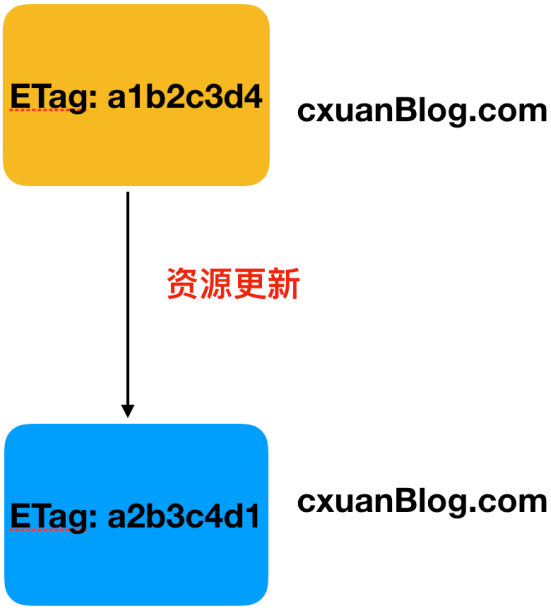
要使用 `Last-Modified` 很难实现强验证，通常，这是通过使用带有资源的 MD5 哈希值的 `Etag` 来完成的。

弱验证

弱验证不同于强验证，因为如果内容相等，它将认为文档的两个版本相同，例如，一个页面与另一个页面的不同之处仅在于页脚的日期不同，因此该页面被认为与其他页面相同。而使用强验证时则被认为这两个版本是不同的。构建一个若验证的 Etag 系统可能会非常复杂，因为这需要了解每个页面元素的重要性，但是对于优化缓存性能非常有用。

下面介绍一下 Etag 如何实现强弱验证。

Etag 响应头是 `特定版本` 的标识，它能够使缓存变得更高效并能够节省带宽，因为如果缓存内容未发生变更，Web 服务器则不需要重新发送完整的响应。除此之外，Etag 能够防止资源同时更新互相覆盖。



如果给定 URL 上的资源发生变更，必须生成一个新的 `Etag` 值，通过比较它们可以确定资源的两个表示形式是否相同。

Etag 值有两种，一种是强 Etag，一种是弱 Etag；

- 强 Etag 值，无论实体发生多么细微的变化都会改变其值，一般的表示如下

```
Etag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
```

- 弱 Etag 值，弱 Etag 值只用于提示资源是否相同。只有资源发生了根本改变，产生差异时才会改变 Etag 值。这时，会在字段值最开始处附加 W/。

```
Etag: W/"0815"
```

下面就来具体探讨一下条件请求的标头和 Etag 的关系

条件请求

条件请求主要包含的标头如下

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

If-Match

对于 `GET` 和 `POST` 方法，服务器仅在与列出的 `Etag（响应标头）` 之一匹配时才返回请求的资源。这里又多了一个新词 `Etag`，我们稍后再说 Etag 的用法。对于像是 `PUT` 和其他非安全的方法，在这种情况下，它仅仅将上传资源。

下面是两种常见的案例

- 对于 `GET` 和 `POST` 方法，会结合使用 `Range` 标头，它可以确保新发送请求的范围与上一个请求的资源相同，如果不匹配的话，会返回 `416` 响应。
- 对于其他方法，特别是 `PUT` 方法，`If-Match` 可以防止丢失更新，服务器会比对 If-Match 的字段值和资源的 Etag 值，仅当两者一致时，才会执行请求。反之，则返回状态码 412 Precondition Failed 的响应。例如

```
If-Match: "bfc13a64729c4290ef5b2c2730249c88ca92d82d"
If-Match: *
```

If-None-Match

条件请求，它与 `If-Match` 的作用相反，仅当 `If-None-Match` 的字段值与 `Etag` 值不一致时，可处理该请求。对于 `GET` 和 `HEAD` ，仅当服务器没有与给定资源匹配的 `Etag` 时，服务器将返回 `200 OK` 作为响应。对于其他方法，仅当最终现有资源的 Etag 与列出的任何值都不匹配时，才会处理请求。

当 `GET` 和 `POST` 发送的 `If-None-Match` 与 `Etag` 匹配时，服务器会返回 `304` 。

```
If-None-Match: "bfc13a64729c4290ef5b2c2730249c88ca92d82d"
If-None-Match: W/"67ab43", "54ed21", "7892dd"
If-None-Match: *
```

If-Modified-Since

`If-Modified-Since` 是 HTTP 条件请求的一部分，只有在给定日期之后，服务端修改了请求所需要的资源，才会返回 200 OK 的响应。如果在给定日期之后，服务端没有修改内容，响应会返回 `304` 并且不带任何响应体。If-Modified-Since 只能使用 `GET` 和 `HEAD` 请求。

If-Modified-Since 与 If-None-Match 结合使用时，它将被忽略，除非服务器不支持 If-None-Match。一般表示如下

```
If-Modified-Since: Wed, 21 Oct 2015 07:28:00 GMT
```

> 注意：这是格林威治标准时间。 HTTP 日期始终以格林尼治标准时间表示，而不是本地时间。

If-Range

`If-Range` 也是条件请求，如果满足条件（If-Range 的值和 Etag 值或者更新的日期时间一致），则会发出范围请求，否则将会返回全部资源。它的一般表示如下

```
If-Range: Wed, 21 Oct 2015 07:28:00 GMT
If-Range: bfc13a64729c4290ef5b2c2730249c88ca92d82d
```

If-Unmodified-Since

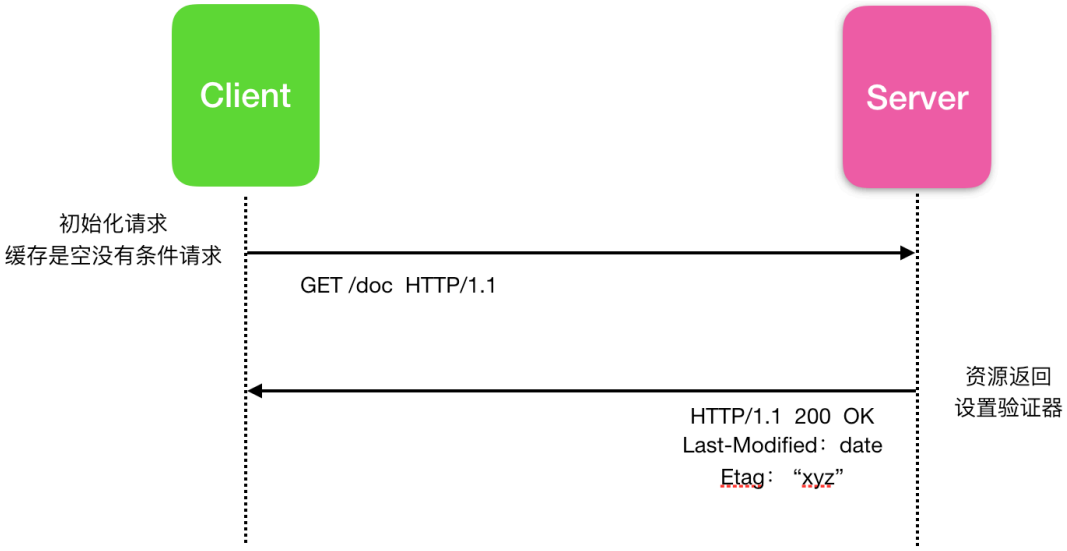
`If-Unmodified-Since` HTTP 请求标头也是一个条件请求，服务器只有在给定日期之后没有对其进行修改时，服务器才返回请求资源。如果在指定日期时间后发生了更新，则以状态码 `412 Precondition Failed` 作为响应返回。

```
If-Unmodified-Since: Wed, 21 Oct 2015 07:28:00 GMT
```

条件请求示例

缓存更新

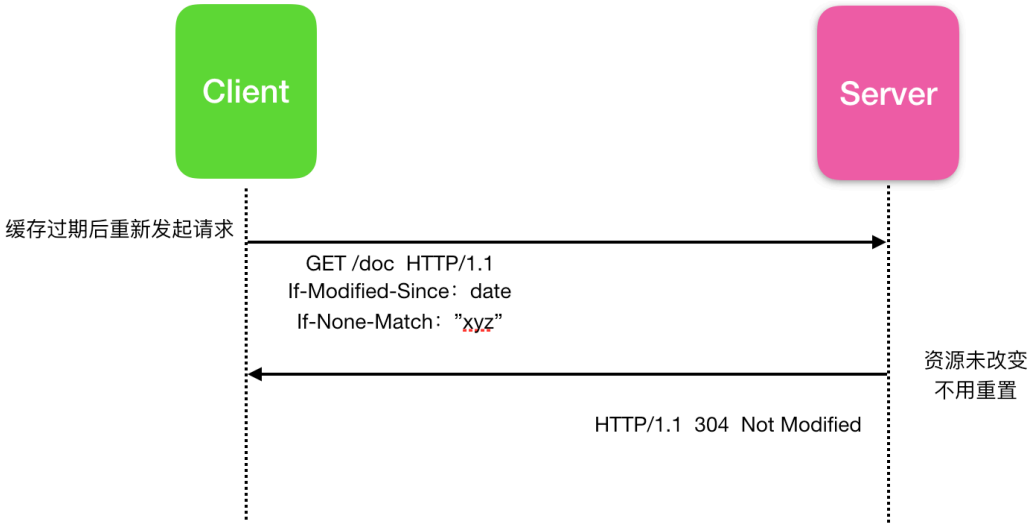
条件请求最常见的示例就是更新缓存，如果缓存是空或没有缓存，则以 `200 OK` 的状态发送回请求的资源。如下图所示



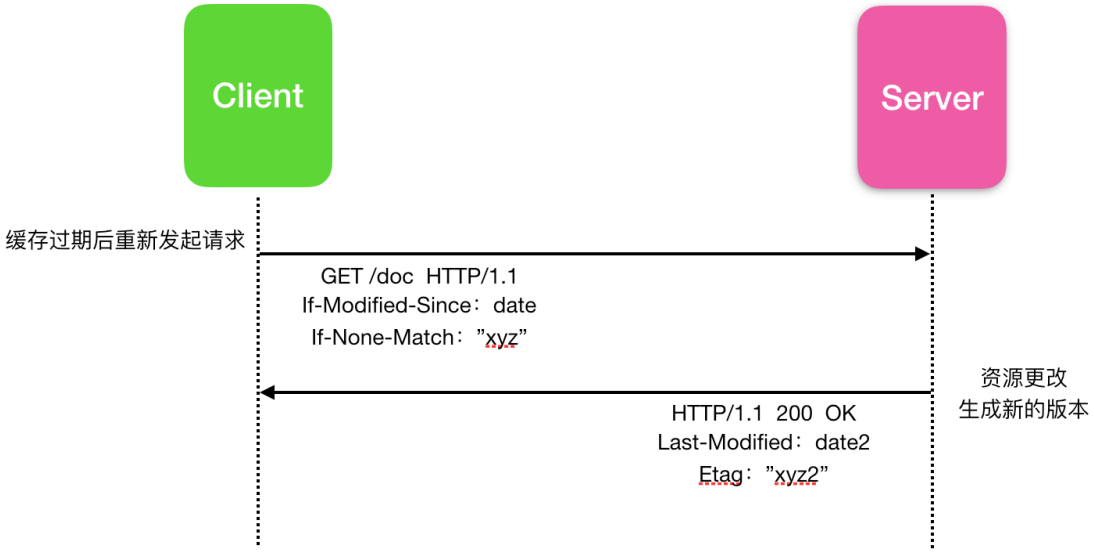
客户端第一次发送请求没有，缓存为空并且没有条件请求，服务器在收到客户端请求后，设置验证器 `Last-Modified` 和 `Etag` 标签，并把这两个标签随着响应一起发送回客户端。

下一次客户端再发送相同的请求后，会直接从缓存中提取，只要缓存没有过期，就不会有任何新的请求到达服务器重新下载资源。但是，一旦缓存过期，客户端不会直接使用缓存的值，而是发出条件请求。验证器的值用作 `If-Modified-Since` 和 `If-Match` 标头的参数。

缓存过期后客户端重新发起请求，服务器收到请求后发现如果资源没有更改，服务器会发回 `304 Not Modified` 响应，这使缓存再次刷新，并让客户端使用缓存的资源。 尽管有一个响应/请求往返消耗一些资源，但是这比再次通过有线传输整个资源更有效。

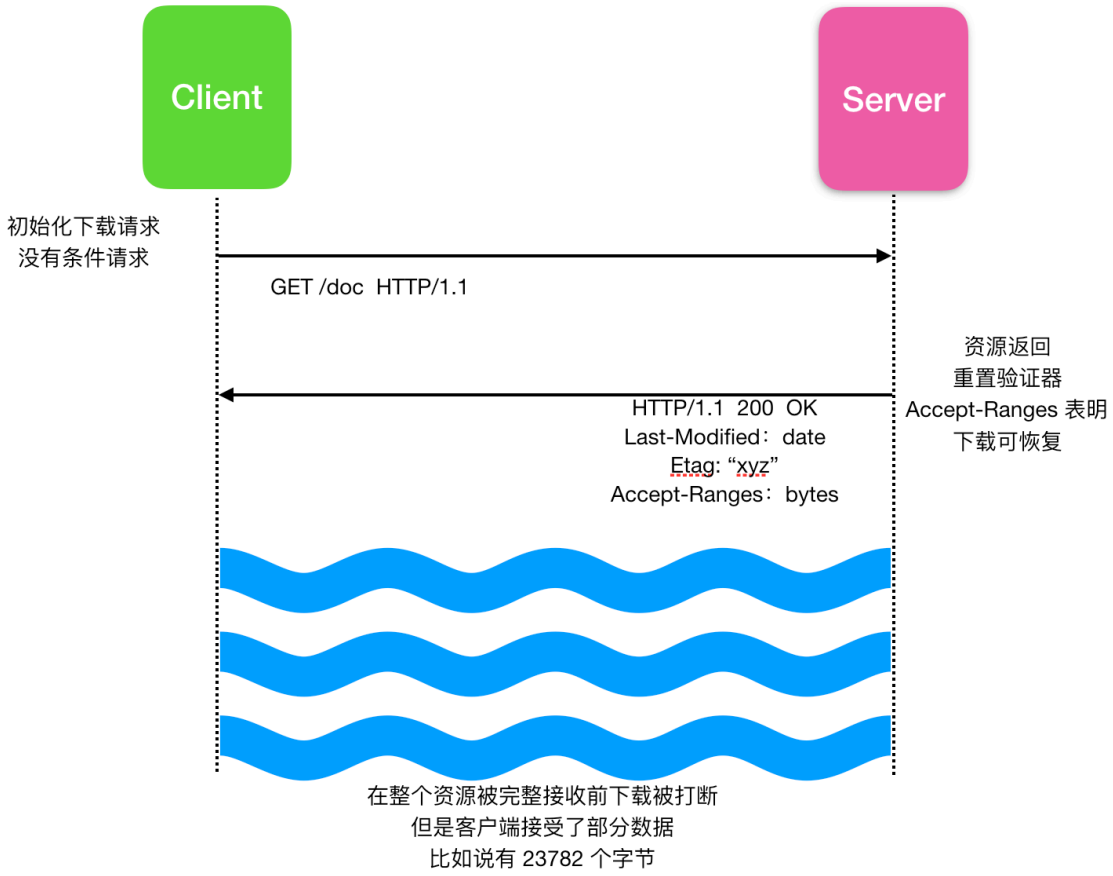


如果资源已经发生更改，则服务器仅使用新版本的资源返回 200 OK 响应，就像没有条件请求，并且客户端会重新使用新的资源，从这个角度来讲，**缓存是条件请求的前置条件**。

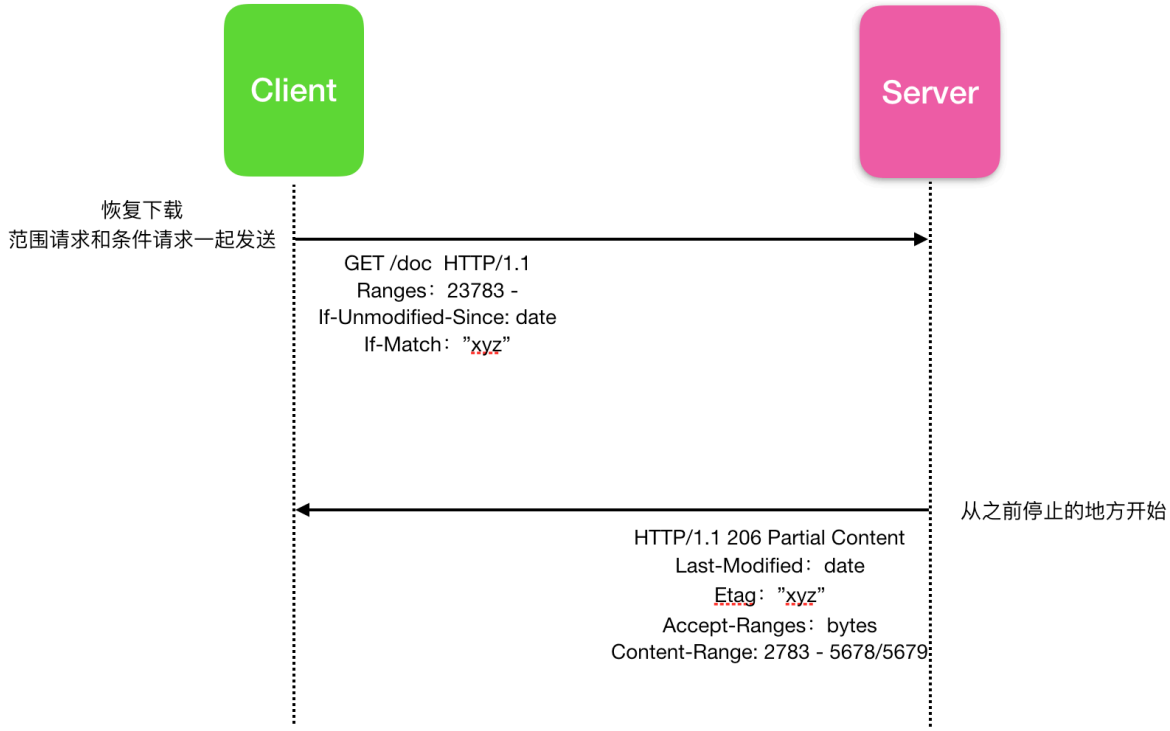


断点续传

HTTP 可以支持文件的部分下载，通过保留已获得的信息，此功能允许恢复先前的操作，从而节省带宽和时间。



支持断点续传的服务器通过发送 `Accept-Ranges` 标头广播此消息，一旦发生这种情况，客户端可以通过发送缺少范围的 `Ranges` 标头来恢复下载



这里你可能有疑问 `Ranges` 和 `Content-Range` 是什么，来解释一下

Range

`Range` HTTP 请求标头指示服务器应返回文档指定部分的资源，可以一次请求一个 Range 来返回多个部分，服务器会将这些资源返回各个文档中。如果服务器成功返回，那么将返回 206 响应；如果 Range 范围无效，服务器返回 `416 Range Not Satisfiable` 错误；服务器还可以忽略 Range 标头，并且返回 200 作为响应。

```
Range: bytes=200-1000, 2000-6576, 19000-
```

还有一种表示是

```
Range: bytes=0-499, -500
```

它们分别表示请求前500个字节和最后500个字节，如果范围重叠，则服务器可能会拒绝该请求。

Content-Range

HTTP 的 Content-Range 响应标头是针对范围请求而设定的，返回响应时使用首部字段 `Content-Range`，能够告知客户端响应实体的哪部分是符合客户端请求的，字段以字节为单位。它的一般表示如下

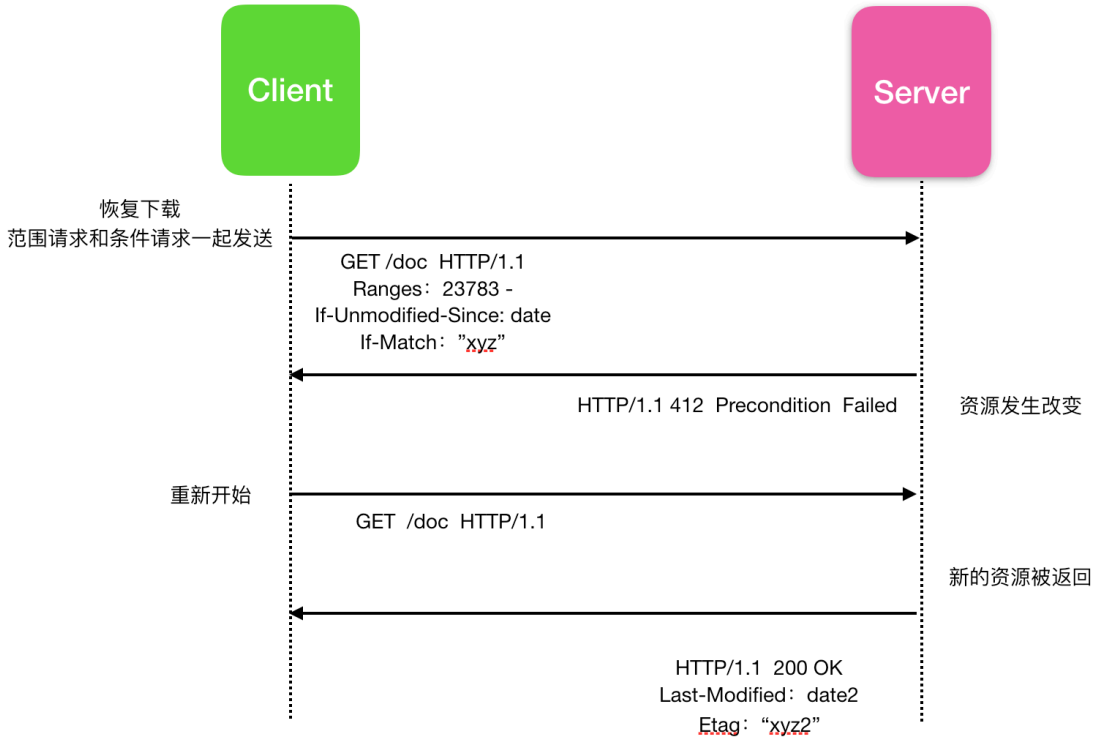
```
Content-Range: bytes 200-1000/67589
```

上段代码表示从所有 `67589` 个字节中返回 `200-1000` 个字节的内容

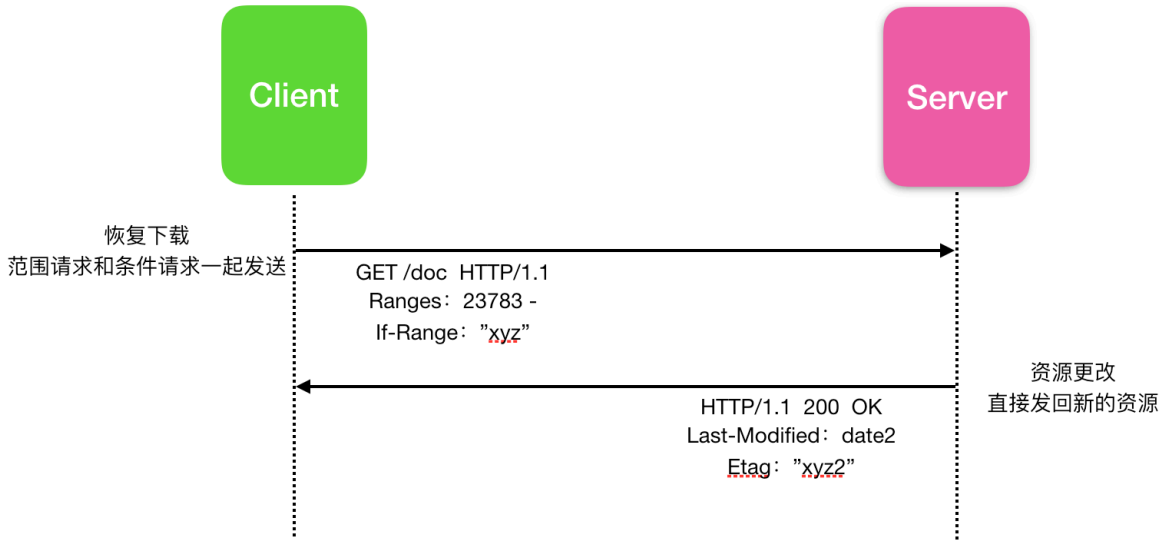
那么上面的 `Content-Range` 你也应该知道是什么意思了

`断点续传` 的原理比较简单，但是这种方式存在潜在的问题：如果在两次下载资源的期间进行了资源更新，那么获得的范围将对应于资源的两个不同版本，并且最终文档将被破坏。

为了阻止这种情况的出现，就会使用 `条件请求` 。对于范围来说，有两种方法可以做到这一点。一种方法是使用 `If-Modified-Since` 和 `If-Match`，如果前提条件失败，服务器将返回错误；然后客户端从头开始重新下载。



即使此方法有效，当文档资源发生改变时，它也会添加额外的 `响应/请求` 交换。这会降低性能，并且 HTTP 具有特定的标头来避免这种情况 `If-Range` 。

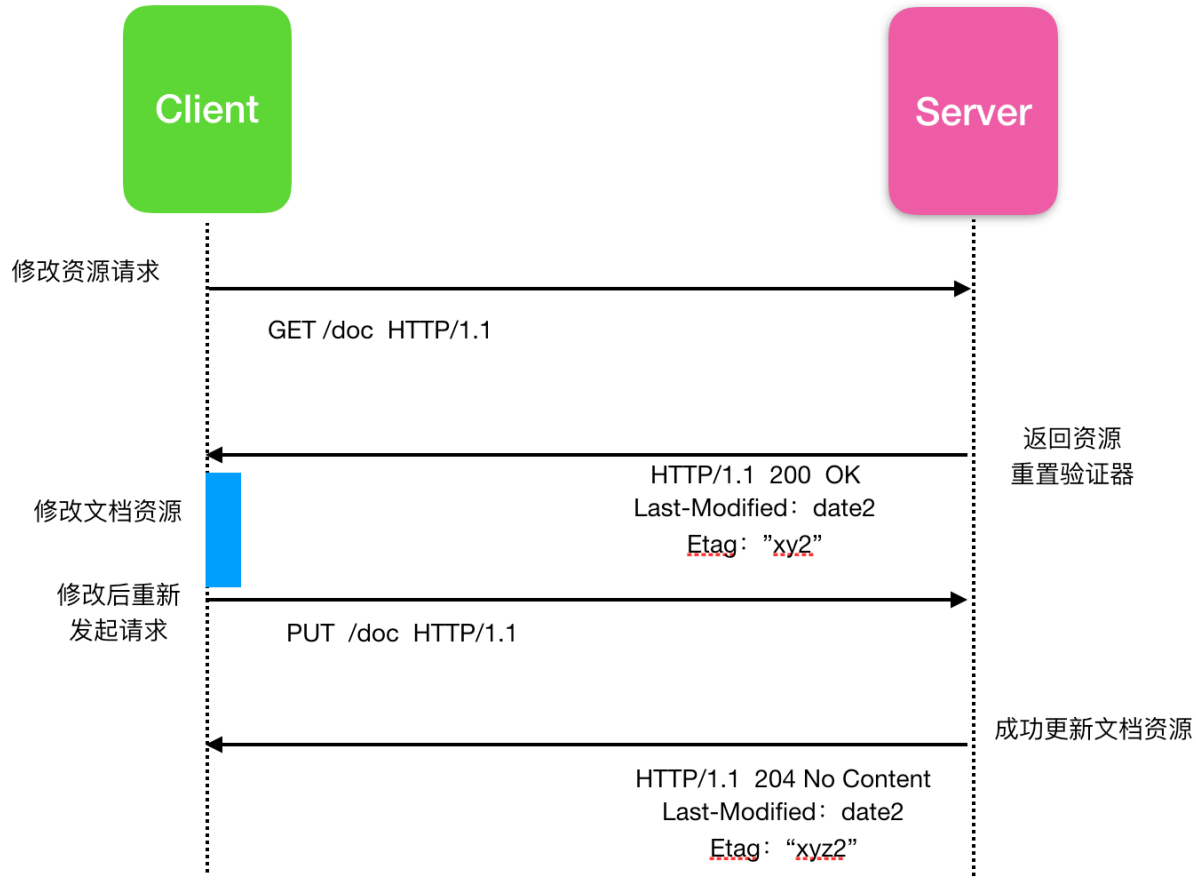


该解决方案效率更高，但灵活性稍差一些，因为在这种情况下只能使用一个 Etag。

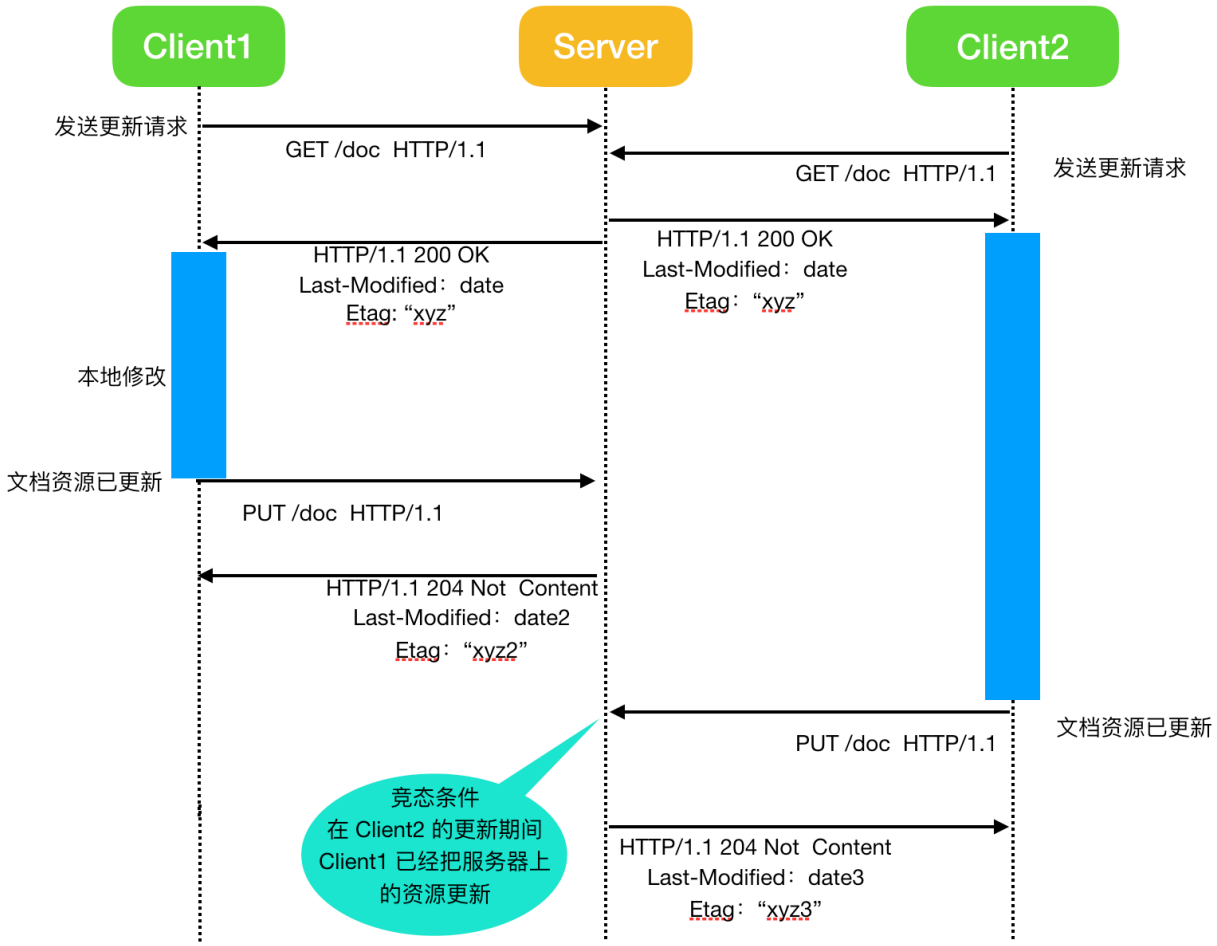
通过乐观锁避免丢失更新

Web 应用程序中最普遍的操作是资源更新。这在任何文件系统或应用程序中都很常见，但是任何允许存储远程资源的应用程序都需要这种机制。

使用 `put` 方法，你可以实现这一点，客户端首先读取原始文件对其进行修改，然后把它们发送到服务器。

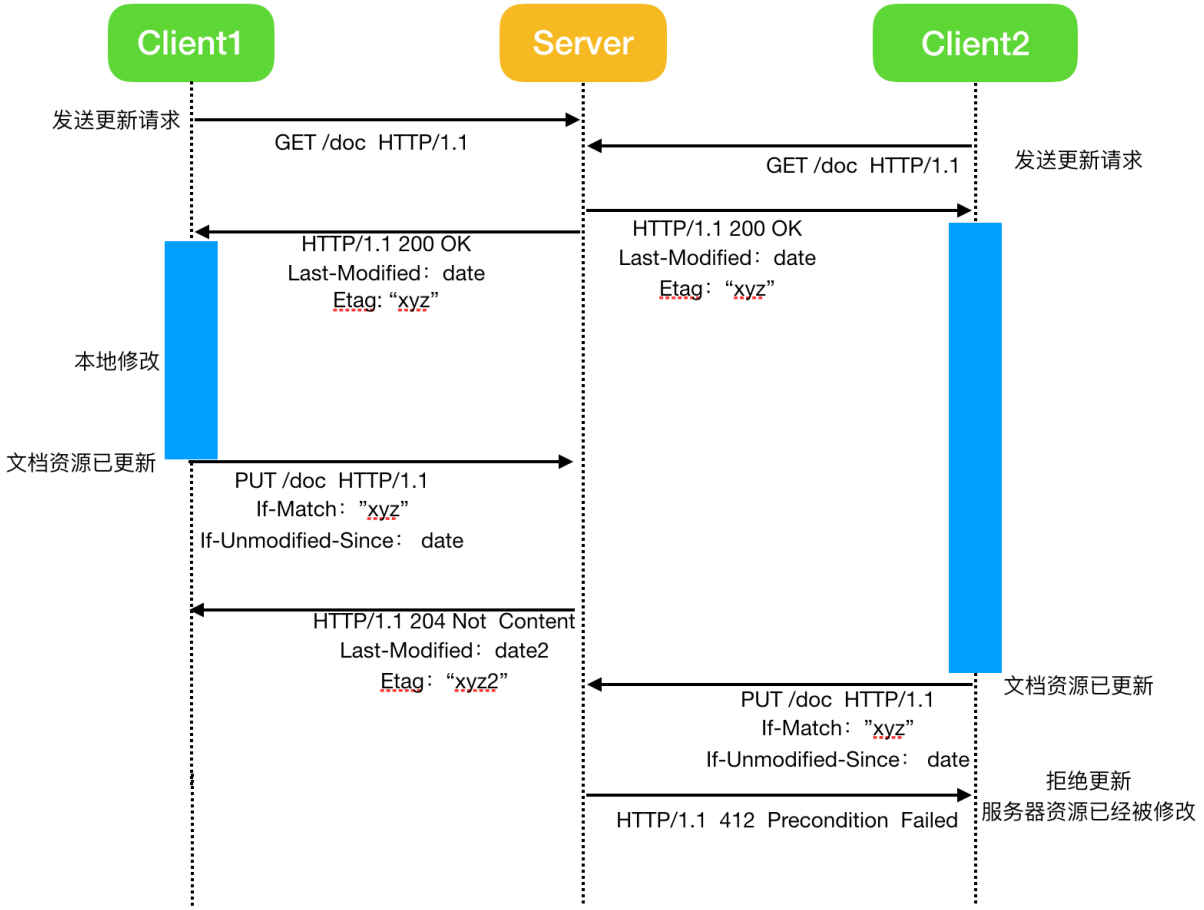


上面这种请求响应存在问题，一旦考虑到并发性，事情就会变得不准确。当客户端在本地修改资源打算重新发送之前，第二个客户端可以获取相同的资源并对资源进行修改操作，这样就会造成问题。当它们重新发送请求到服务器时，第一个客户端所做的修改将被第二次客户端的修改所覆盖，因为第二次客户端修改并不知道第一次客户端正在修改。资源提交并更新的一方不会传达给另外一方，所以要保留哪个客户的更改，将随着他们提交的速度而变化； 这取决于客户端，服务器的性能，甚至取决于人工在客户端编辑文档的性能。 例如下面这个流程



如果没有两个用户同时操作服务器，也就不存在这个问题。但是，现实情况是不可能只有单个用户出现的，所以为了规避或者避免这个问题，我们希望客户端资源在更新时进行提示或者修改被拒绝时收到通知。

条件请求允许实现乐观锁算法。这个概念是允许所有的客户端获取资源的副本，然后让他们在本地修改资源，并成功通过允许第一个客户端提交更新来控制并发，基于此服务端的后面版本的更新都将被拒绝。



这是使用 `If-Match` 或 `If-Unmodified-Since` 标头实现的。如果 Etag 与原始文件不匹配，或者自获取以来已对文件进行了修改，则更改为拒绝更新，并显示 `412 Precondition Failed` 错误。

HTTP Cookies

HTTP 协议中的 Cookie 包括 `Web Cookie` 和 `浏览器 Cookie`，它是服务器发送到 Web 浏览器的一小块数据。服务器发送到浏览器的 Cookie，浏览器会进行存储，并与下一个请求一起发送到服务器。通常，它用于判断两个请求是否来自于同一个浏览器，例如用户保持登录状态。

> HTTP Cookie 机制是 HTTP 协议无状态的一种补充和改良

Cookie 主要用于下面三个目的

- `会话管理`

登陆、购物车、游戏得分或者服务器应该记住的其他内容

- `个性化`

用户偏好、主题或者其他设置

- `追踪`

记录和分析用户行为

Cookie 曾经用于一般的客户端存储。虽然这是合法的，因为它们是在客户端上存储数据的唯一方法，但如今建议使用现代存储 API。Cookie 随每个请求一起发送，因此它们可能会降低性能（尤其是对于移动数据连接而言）。客户端存储的现代 API 是 Web 存储 API（localStorage 和 sessionStorage）和 IndexedDB。

创建 Cookie

当接收到客户端发出的 HTTP 请求时，服务器可以发送带有响应的 `Set-Cookie` 标头，Cookie 通常由浏览器存储，然后将 Cookie 与 HTTP 标头一同向服务器发出请求。可以指定到期日期或持续时间，之后将不再发送Cookie。此外，可以设置对特定域和路径的限制，从而限制 cookie 的发送位置。

Set-Cookie 和 Cookie 标头

`Set-Cookie` HTTP 响应标头将 cookie 从服务器发送到用户代理。下面是一个发送 Cookie 的例子

```
HTTP/2.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry

[page content]
```

此标头告诉客户端存储 Cookie

现在，随着对服务器的每个新请求，浏览器将使用 Cookie 头将所有以前存储的 cookie 发送回服务器。

```
GET /sample_page.html HTTP/2.0
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Cookie 主要分为三类，它们是 `会话Cookie` 、 `永久Cookie` 和 `Cookie的 Secure 和 HttpOnly 标记` ，下面依次来介绍一下

会话 Cookies

上面的示例创建的是会话 Cookie ， 会话 Cookie 有个特征，客户端关闭时 Cookie 会删除，因为它没有指定Expires 或 Max-Age 指令。 这两个指令你看到这里应该比较熟悉了。

但是，Web 浏览器可能会使用会话还原，这会使大多数会话 Cookie 保持永久状态，就像从未关闭过浏览器一样

永久性 Cookies

永久性 Cookie 不会在客户端关闭时过期，而是在特定日期（Expires）或特定时间长度（Max-Age）外过期。例如

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
```

Cookie的 Secure 和 HttpOnly 标记

安全的 Cookie 需要经过 HTTPS 协议通过加密的方式发送到服务器。即使是安全的，也不应该将敏感信息存储在cookie 中，因为它们本质上是不安全的，并且此标志不能提供真正的保护。

HttpOnly 的作用

- 会话 cookie 中缺少 HttpOnly 属性会导致攻击者可以通过程序(JS脚本、Applet等)获取到用户的 cookie 信息，造成用户cookie 信息泄露，增加攻击者的跨站脚本攻击威胁。
- HttpOnly 是微软对 cookie 做的扩展，该值指定 cookie 是否可通过客户端脚本访问。
- 如果在 Cookie 中没有设置 HttpOnly 属性为 true，可能导致 Cookie 被窃取。窃取的 Cookie 可以包含标识站点用户的敏感信息，如 ASP.NET 会话 ID 或 Forms 身份验证票证，攻击者可以重播窃取的 Cookie，以便伪装成用户或获取敏感信息，进行跨站脚本攻击等。

Cookie 的作用域

`Domain` 和 `Path` 标识定义了 Cookie 的作用域：即 Cookie 应该发送给哪些 URL。

`Domain` 标识指定了哪些主机可以接受 Cookie。如果不指定，默认为当前主机(**不包含子域名**）。如果指定了 `Domain`，则一般包含子域名。

例如，如果设置 `Domain=mozilla.org`，则 Cookie 也包含在子域名中（如 `developer.mozilla.org`）。

例如，设置 `Path=/docs`，则以下地址都会匹配：

- `/docs`
- `/docs/Web/`
- `/docs/Web/HTTP`

HTTP 的优点和缺点

HTTP 的优点

简单灵活易扩展

HTTP 最重要也是最突出的优点是 **简单、灵活、易于扩展**。

HTTP 的协议比较简单，它的主要组成就是 `header + body`，头部信息也是简单的文本格式，而且 HTTP 的请求报文根据英文也能猜出来个大概的意思，降低学习门槛，能够让更多的人研究和开发 HTTP 应用。

所以，在简单的基础上，HTTP 协议又多了 `灵活` 和 `易扩展` 的优点。

HTTP 协议里的请求方法、URI、状态码、原因短语、头字段等每一个核心组成要素都没有被制定死，允许开发者任意定制、扩充或解释，给予了浏览器和服务器最大程度的信任和自由。

应用广泛、环境成熟

因为过于简单，普及，因此应用很广泛。因为 HTTP 协议本身不属于一种语言，它并不限定某种编程语言或者操作系统，所以天然具有**跨语言、跨平台**的优越性。而且，因为本身的简单特性很容易实现，所以几乎所有的编程语言都有 HTTP 调用库和外围的开发测试工具。

随着移动互联网的发展， HTTP 的触角已经延伸到了世界的每一个角落，从简单的 Web 页面到复杂的 JSON、XML 数据，从台式机上的浏览器到手机上的各种 APP、新闻、论坛、购物、手机游戏，你很难找到一个没有使用 HTTP 的地方。

无状态

无状态其实既是优点又是缺点。因为服务器没有记忆能力，所以就不需要额外的资源来记录状态信息，不仅实现上会简单一些，而且还能减轻服务器的负担，能够把更多的 CPU 和内存用来对外提供服务。

HTTP 的缺点

无状态

既然服务器没有记忆能力，它就无法支持需要连续多个步骤的 `事务` 操作。每次都得问一遍身份信息，不仅麻烦，而且还增加了不必要的数据传输量。由此出现了 `Cookie` 技术。

明文

HTTP 协议里还有一把优缺点一体的双刃剑，就是**明文传输**。明文意思就是协议里的报文（准确地说是 header 部分）不使用二进制数据，而是用简单可阅读的文本形式。

对比 TCP、UDP 这样的二进制协议，它的优点显而易见，不需要借助任何外部工具，用浏览器、Wireshark 或者 tcpdump 抓包后，直接用肉眼就可以很容易地查看或者修改，为我们的开发调试工作带来极大的便利。

当然缺点也是显而易见的，就是 `不安全` ，可以被监听和被窥探。因为无法判断通信双方的身份，不能判断报文是否被更改过。

性能

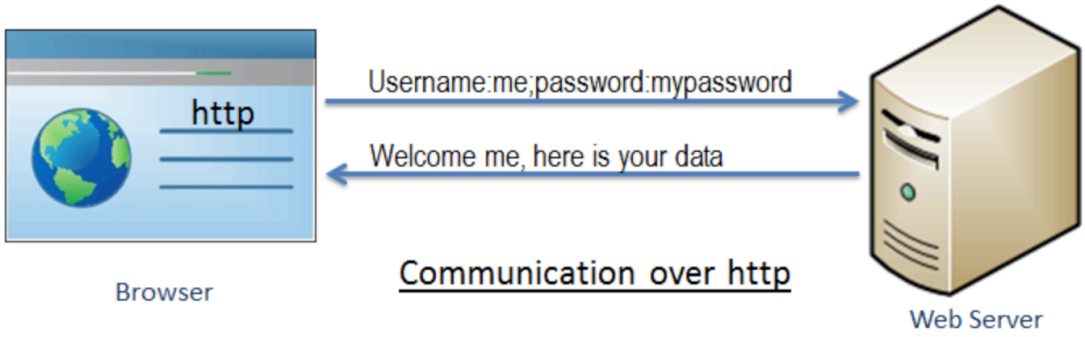
HTTP 的性能不算差，但不完全适应现在的互联网，还有很大的提升空间。

HTTPS 为什么会出现

一个新技术的出现必定是为了解决某种问题的，那么 HTTPS 解决了 HTTP 的什么问题呢?

HTTPS 解决了什么问题

一个简单的回答可能会是 `HTTP` 它不安全。由于 HTTP 天生明文传输的特性，在 HTTP 的传输过程中，任何人都有可能从中截获、修改或者伪造请求发送，所以可以认为 HTTP 是不安全的；在 HTTP 的传输过程中不会验证通信方的身份，因此 HTTP 信息交换的双方可能会遭到伪装，也就是 `没有用户验证` ；在 HTTP 的传输过程中，接收方和发送方并 `不会验证报文的完整性` ，综上，为了结局上述问题，HTTPS 应用而生。



什么是 HTTPS

你还记得 HTTP 是怎么定义的吗？HTTP 是一种 `超文本传输协议(Hypertext Transfer Protocol)` 协议，**它是一个在计算机世界里专门在两点之间传输文字、图片、音频、视频等超文本数据的约定和规范**，那么我们看一下 HTTPS 是如何定义的

`HTTPS` 的全称是 `Hypertext Transfer Protocol Secure`，它用来在计算机网络上的两个端系统之间进行 `安全的交换信息(secure communication)`，它相当于在 HTTP 的基础上加了一个 `Secure 安全` 的词眼，那么我们可以给出一个 HTTPS 的定义：**HTTPS 是一个在计算机世界里专门在两点之间安全的传输文字、图片、音频、视频等超文本数据的约定和规范。** HTTPS 是 HTTP 协议的一种扩展，它本身并不保传输的证安全性，那么谁来保证安全性呢？在 HTTPS 中，使用 `传输层安全性(TLS)` 或 `安全套接字层(SSL)` 对通信协议进行加密。也就是 HTTP + SSL(TLS) = HTTPS。

HTTPS 做了什么

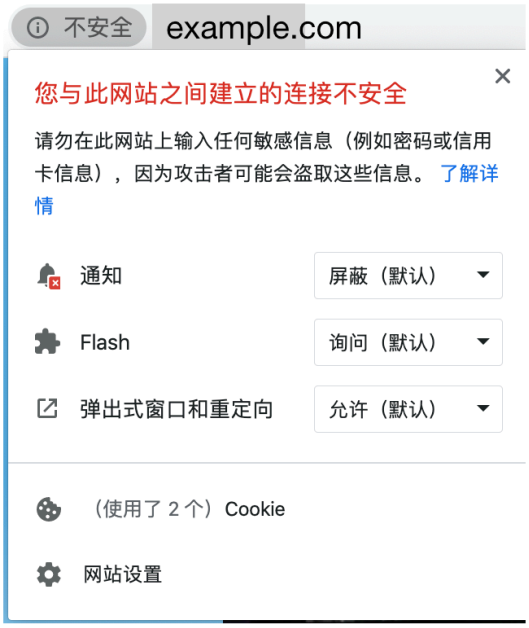
HTTPS 协议提供了三个关键的指标

- `加密(Encryption)`， HTTPS 通过对数据加密来使其免受窃听者对数据的监听，这就意味着当

用户在浏览网站时，没有人能够监听他和网站之间的信息交换，或者跟踪用户的活动，访问记录等，从而窃取用户信息。

- `数据一致性(Data integrity)`，数据在传输的过程中不会被窃听者所修改，用户发送的数据会`完整`的传输到服务端，保证用户发的是什么，服务器接收的就是什么。
- `身份认证(Authentication)`，是指确认对方的真实身份，也就是`证明你是你`（可以比作人脸识别），它可以防止中间人攻击并建立用户信任。

有了上面三个关键指标的保证，用户就可以和服务器进行安全的交换信息了。那么，既然你说了 HTTPS 的种种好处，那么我怎么知道网站是用 HTTPS 的还是 HTTP 的呢？给你两幅图应该就可以解释了。



HTTPS 协议其实非常简单，RFC 文档很小，只有短短的 7 页，里面规定了新的协议名，默认`端口号443`，至于其他的**应答模式、报文结构、请求方法、URI、头字段、连接管理**等等都完全沿用 HTTP，没有任何新的东西。

也就是说，除了协议名称和默认端口号外（HTTP 默认端口 80），HTTPS 协议在语法、语义上和 HTTP 一样，HTTP 有的，HTTPS 也照单全收。那么，HTTPS 如何做到 HTTP 所不能做到的`安全性呢`？关键在于这个 `S` 也就是 `SSL/TLS` 。

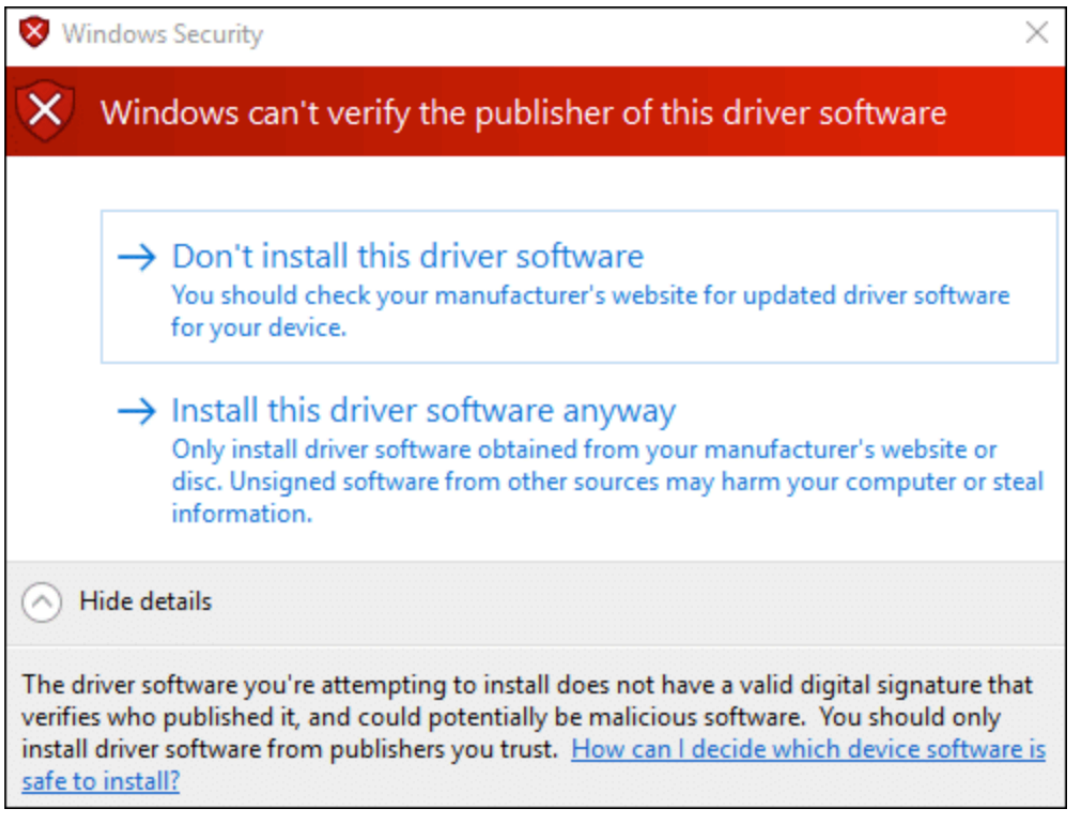
什么是 SSL/TLS

认识 SSL/TLS

`TLS(Transport Layer Security)` 是 `SSL(Secure Socket Layer)` 的后续版本，它们是用于在互联网两台计算机之间用于`身份验证`和`加密`的一种协议。

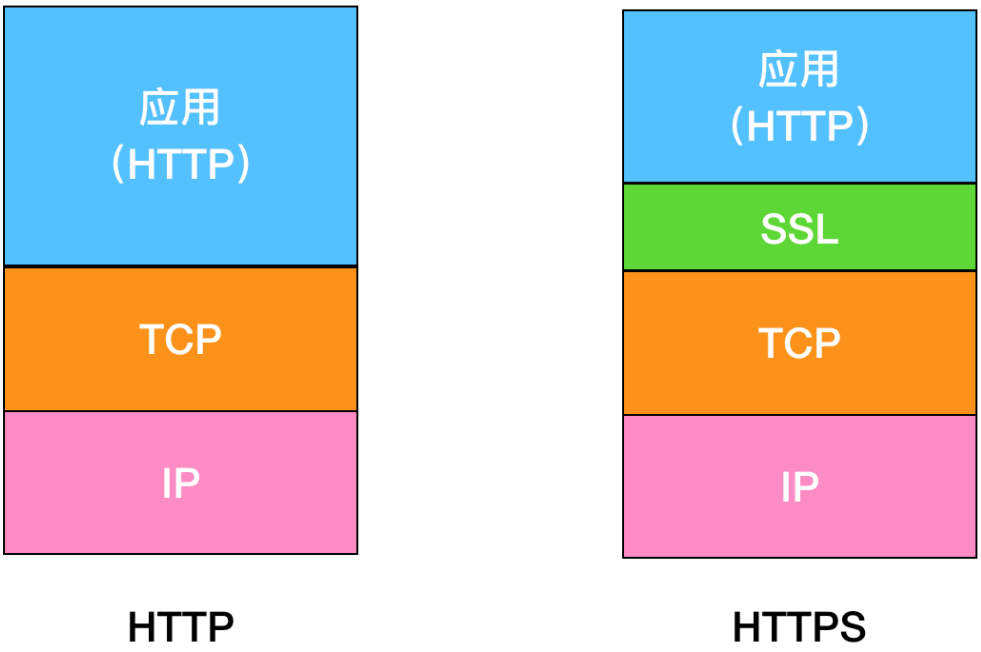
> 注意：在互联网中，很多名称都可以进行互换。

我们都知道一些在线业务（比如在线支付）最重要的一个步骤是创建一个值得信赖的交易环境，能够让客户安心的进行交易，SSL/TLS 就保证了这一点，SSL/TLS 通过将称为 `X.509` 证书的数字文档将网站和公司的实体信息绑定到 `加密密钥` 来进行工作。每一个 `密钥对(key pairs)` 都有一个 `私有密钥(private key)` 和 `公有密钥(public key)`，私有密钥是独有的，一般位于服务器上，用于解密由公共密钥加密过的信息；公有密钥是公有的，与服务器进行交互的每个人都可以持有公有密钥，用公钥加密的信息只能由私有密钥来解密。

> 什么是 `X.509`：X.509 是 `公开密钥` 证书的标准格式，这个文档将加密密钥与（个人或组织）进行安全的关联。
>
> X.509 主要应用如下
>
> - SSL/TLS 和 HTTPS 用于经过身份验证和加密的 Web 浏览
> - 通过 **S/MIME** 协议签名和加密的电子邮件
> - 代码签名：它指的是使用数字证书对软件应用程序进行签名以安全分发和安装的过程。
>
> 
>
>
> 通过使用由知名公共证书颁发机构（例如SSL.com）颁发的证书对软件进行数字签名，开发人员可以向最终用户保证他们希望安装的软件是由已知且受信任的开发人员发布；并且签名后未被篡改或损害。
>
> - 还可用于文档签名
> - 还可用于客户端认证
> - 政府签发的电子身份证（详见 **https://www.ssl.com/article/pki-and-digital-certificates-for-government/**）
>
> 我们后面还会讨论。

HTTPS 的内核是 HTTP

HTTPS 并不是一项新的应用层协议，只是 HTTP 通信接口部分由 SSL 和 TLS 替代而已。通常情况下，HTTP 会先直接和 TCP 进行通信。在使用 SSL 的 HTTPS 后，则会先演变为和 SSL 进行通信，然后再由 SSL 和 TCP 进行通信。也就是说，**HTTPS 就是身披了一层 SSL 的 HTTP**。（我都喜欢把骚粉留在最后。。。）



SSL 是一个独立的协议，不只有 HTTP 可以使用，其他应用层协议也可以使用，比如 `SMTP(电子邮件协议)`、`Telnet(远程登录协议)` 等都可以使用。

探究 HTTPS

我说，你起这么牛逼的名字干嘛，还想吹牛批？你 HTTPS 不就抱上了 TLS/SSL 的大腿么，咋这么牛批哄哄的，还想探究 HTTPS，瞎胡闹，赶紧改成 TLS 是我主，赞美我主。

SSL 即 `安全套接字层`，它在 OSI 七层网络模型中处于第五层，SSL 在 1999 年被 `IETF(互联网工程组)` 更名为 TLS，即 `传输安全层`，直到现在，TLS 一共出现过三个版本，1.1、1.2 和 1.3，目前最广泛使用的是 1.2，所以接下来的探讨都是基于 TLS 1.2 的版本上的。

TLS 用于两个通信应用程序之间提供保密性和数据完整性。TLS 由**记录协议、握手协议、警告协议、变更密码规范协议、扩展协议**等几个子协议组成，综合使用了**对称加密、非对称加密、身份认证**等许多密码学前沿技术（如果你觉得一项技术很简单，那你只是没有学到位，任何技术都是有美感的，牛逼的人只是欣赏，并不是贬低）。

说了这么半天，我们还没有看到 TLS 的命名规范呢，下面举一个 TLS 例子来看一下 TLS 的结构（可以参考 **https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml**）

```
1   ECDHE-ECDSA-AES256-GCM-SHA384
```

这是啥意思呢？我刚开始看也有点懵啊，但其实是有套路的，因为 TLS 的密码套件比较规范，基本格式就是 **密钥交换算法 - 签名算法 - 对称加密算法 - 摘要算法** 组成的一个密码串，有时候还有 `分组模式`，我们先来看一下刚刚是什么意思

使用 ECDHE 进行密钥交换，使用 ECDSA 进行签名和认证，然后使用 AES 作为对称加密算法，密钥的长度是 256 位，使用 GCM 作为分组模式，最后使用 SHA384 作为摘要算法。

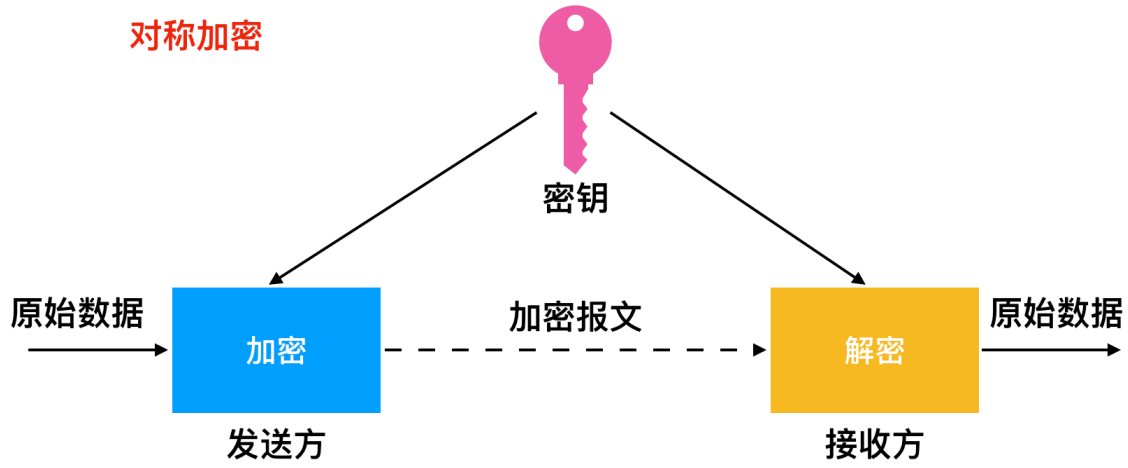
TLS 在根本上使用 `对称加密` 和 `非对称加密` 两种形式。

对称加密

在了解对称加密前，我们先来了解一下 `密码学` 的东西，在密码学中，有几个概念：**明文、密文、加密、解密**

- `明文(Plaintext)`，一般认为明文是有意义的字符或者比特集，或者是通过某种公开编码就能获得的消息。明文通常用 m 或 p 表示
- `密文(Ciphertext)`，对明文进行某种加密后就变成了密文
- `加密(Encrypt)`，把原始的信息（明文）转换为密文的信息变换过程
- `解密(Decrypt)`，把已经加密的信息恢复成明文的过程。

`对称加密(Symmetrical Encryption)` 顾名思义就是指**加密和解密时使用的密钥都是同样的密钥**。只要保证了密钥的安全性，那么整个通信过程也就是具有了机密性。



TLS 里面有比较多的加密算法可供使用，比如 DES、3DES、AES、ChaCha20、TDEA、Blowfish、RC2、RC4、RC5、IDEA、SKIPJACK 等。目前最常用的是 AES-128, AES-192、AES-256 和 ChaCha20。

`DES` 的全称是 `Data Encryption Standard(数据加密标准)`，它是用于数字数据加密的对称密钥算法。尽管其 56 位的短密钥长度使它对于现代应用程序来说太不安全了，但它在加密技术的发展中具有很大的影响力。

`3DES` 是从原始数据加密标准（DES）衍生过来的加密算法，它在 90 年代后变得很重要，但是后面由于更加高级的算法出现，3DES 变得不再重要。

AES-128, AES-192 和 AES-256 都是属于 AES，AES 的全称是 `Advanced Encryption Standard(高级加密标准)`，它是 DES 算法的替代者，安全强度很高，性能也很好，是应用最广泛的对称加密算法。

`ChaCha20` 是 Google 设计的另一种加密算法，密钥长度固定为 256 位，纯软件运行性能要超过 AES，曾经在移动客户端上比较流行，但 ARMv8 之后也加入了 AES 硬件优化，所以现在不再具有明显的优势，但仍然算得上是一个不错算法。

（其他可自行搜索）

加密分组

对称加密算法还有一个 `分组模式` 的概念，对于 GCM 分组模式，只有和 AES，CAMELLIA 和 ARIA 搭配使用，而 AES 显然是最受欢迎和部署最广泛的选择，它可以让算法用固定长度的密钥加密任意长度的明文。

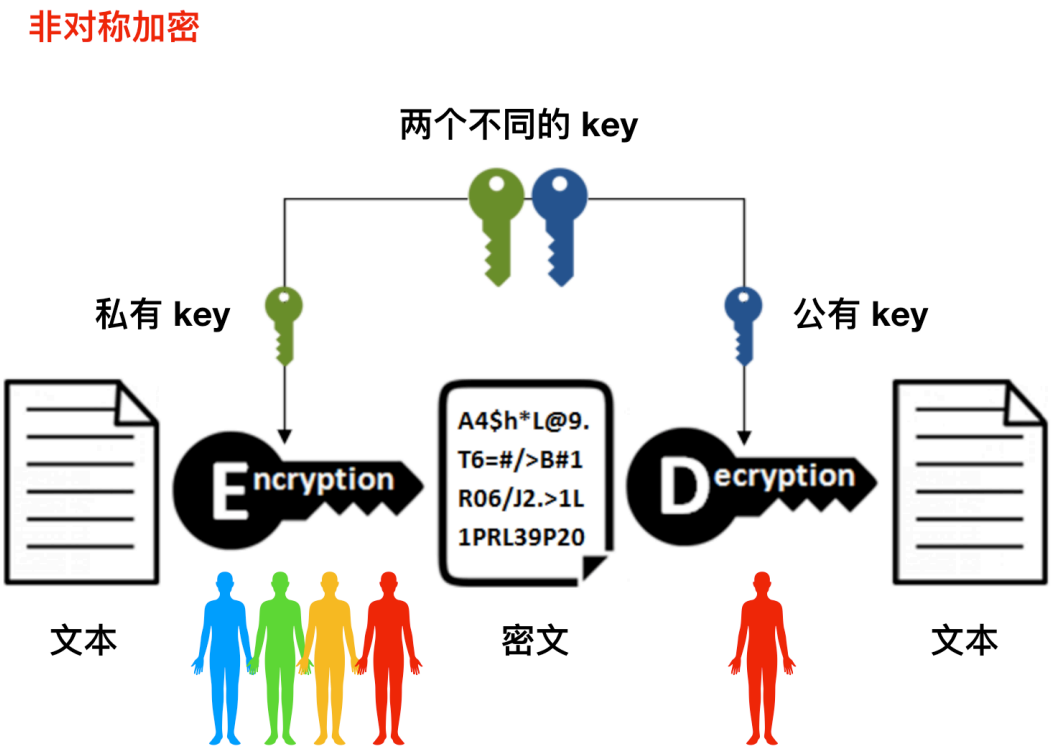
最早有 ECB、CBC、CFB、OFB 等几种分组模式，但都陆续被发现有安全漏洞，所以现在基本都不怎么用了。最新的分组模式被称为 `AEAD (Authenticated Encryption with Associated Data)` ，在加密的同时增加了认证的功能，常用的是 GCM、CCM 和 Poly1305。

比如 `ECDHE_ECDSA_AES128_GCM_SHA256` ，表示的是具有 128 位密钥， AES256 将表示 256 位密钥。GCM 表示具有 128 位块的分组密码的现代认证的关联数据加密（AEAD）操作模式。

我们上面谈到了对称加密，对称加密的加密方和解密方都使用同一个 `密钥` ，也就是说，加密方必须对原始数据进行加密，然后再把密钥交给解密方进行解密，然后才能解密数据，这就会造成什么问题？这就好比《小兵张嘎》去送信（信已经被加密过），但是嘎子还拿着解密的密码，那嘎子要是在途中被鬼子发现了，那这信可就是被完全的暴露了。所以，对称加密存在风险。

非对称加密

`非对称加密(Asymmetrical Encryption)` 也被称为 `公钥加密` ，相对于对称加密来说，非对称加密是一种新的改良加密方式。密钥通过网络传输交换，它能够确保及时密钥被拦截，也不会暴露数据信息。非对称加密中有两个密钥，一个是公钥，一个是私钥，公钥进行加密，私钥进行解密。公开密钥可供任何人使用，私钥只有你自己能够知道。



使用公钥加密的文本只能使用私钥解密，同时，使用私钥加密的文本也可以使用公钥解密。公钥不需要具有安全性，因为公钥需要在网络间进行传输，非对称加密可以解决 `密钥交换` 的问题。网站保管私钥，在网上任意分发公钥，你想要登录网站只要用公钥加密就行了，密文只能由私钥持有者才能解密。而黑客因为没有私钥，所以就无法破解密文。

非对称加密算法的设计要比对称算法难得多（我们不会探讨具体的加密方式），常见的比如 DH、DSA、RSA、ECC 等。

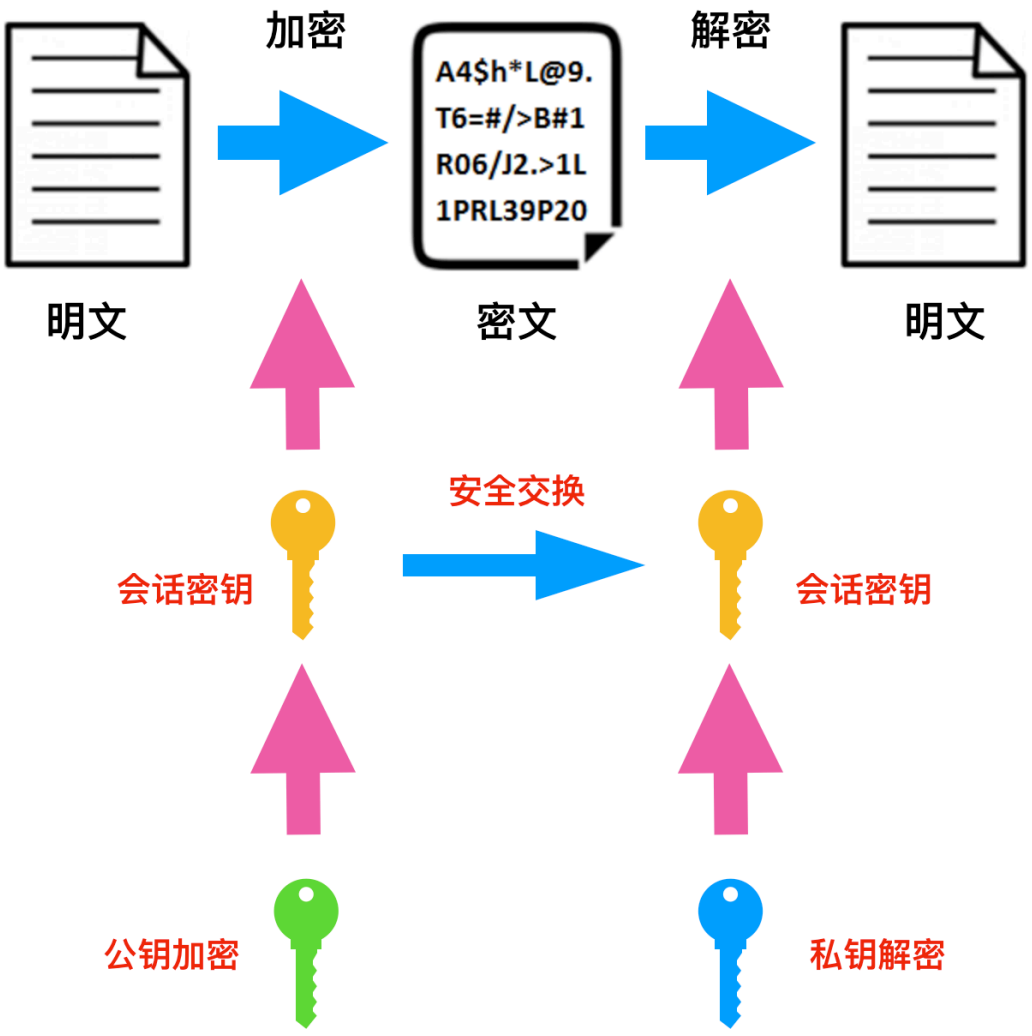
其中 `RSA` 加密算法是最重要的、最出名的一个了。例如 `DHE_RSA_CAMELLIA128_GCM_SHA256` 。它的安全性基于 `整数分解` ，使用两个超大素数的乘积作为生成密钥的材料，想要从公钥推算出私钥是非常困难的。

`ECC (Elliptic Curve Cryptography)` 也是非对称加密算法的一种，它基于 `椭圆曲线离散对数` 的数学难题，使用特定的曲线方程和基点生成公钥和私钥， ECDHE 用于密钥交换，ECDSA 用于数字签名。

TLS 是使用 `对称加密` 和 `非对称加密` 的混合加密方式来实现机密性。

混合加密

RSA 的运算速度非常慢，而 AES 的加密速度比较快，而 TLS 正是使用了这种 `混合加密` 方式。在通信刚开始的时候使用非对称算法，比如 RSA、ECDHE ，首先解决 `密钥交换` 的问题。然后用随机数产生对称算法使用的 `会话密钥（session key）` ，再用 `公钥加密` 。对方拿到密文后用 `私钥解密` ，取出会话密钥。这样，双方就实现了对称密钥的安全交换。



现在我们使用混合加密的方式实现了机密性，是不是就能够安全的传输数据了呢？还不够，在机密性的基础上还要加上 `完整性` 、 `身份认证` 的特性，才能实现真正的安全。而实现完整性的主要手段是 `摘要算法(Digest Algorithm)`

摘要算法

如何实现完整性呢？在 TLS 中，实现完整性的手段主要是 `摘要算法(Digest Algorithm)` 。摘要算法你不清楚的话，MD5 你应该清楚，MD5 的全称是 `Message Digest Algorithm 5` ，它是属于 `密码哈希算法(cryptographic hash algorithm)` 的一种，MD5 可用于从任意长度的字符串创建 128 位字符串值。尽管 MD5 存在不安全因素，但是仍然沿用至今。MD5 最常用于 `验证文件` 的完整性。但是，它还用于其他安全协议和应用程序中，例如 SSH、SSL 和 IPSec。一些应用程序通过向明文加盐值或多次应用哈希函数来增强 MD5 算法。

> 什么是加盐？在密码学中， `盐` 就是一项随机数据，用作哈希数据，密码或密码的 `单向` 函数的附加输入。盐用于保护存储中的密码。例如
>
> | Username | Salt value |
> | --- | --- |
> | user1 | E1F53135E559C253 |
> | user2 | 84B03D034B409D4E |
>
> 什么是单向？就是在说这种算法没有密钥可以进行解密，只能进行单向加密，加密后的数据无法解密，不能逆推出原文。

我们再回到摘要算法的讨论上来，其实你可以把摘要算法理解成一种特殊的压缩算法，它能够把任意长度的数据 `压缩` 成一种固定长度的字符串，这就好像是给数据加了一把锁。

除了常用的 MD5 是加密算法外， `SHA-1(Secure Hash Algorithm 1)` 也是一种常用的加密算法，不过 SHA-1 也是不安全的加密算法，在 TLS 里面被禁止使用。目前 TLS 推荐使用的是 SHA-1 的后继者： `SHA-2` 。

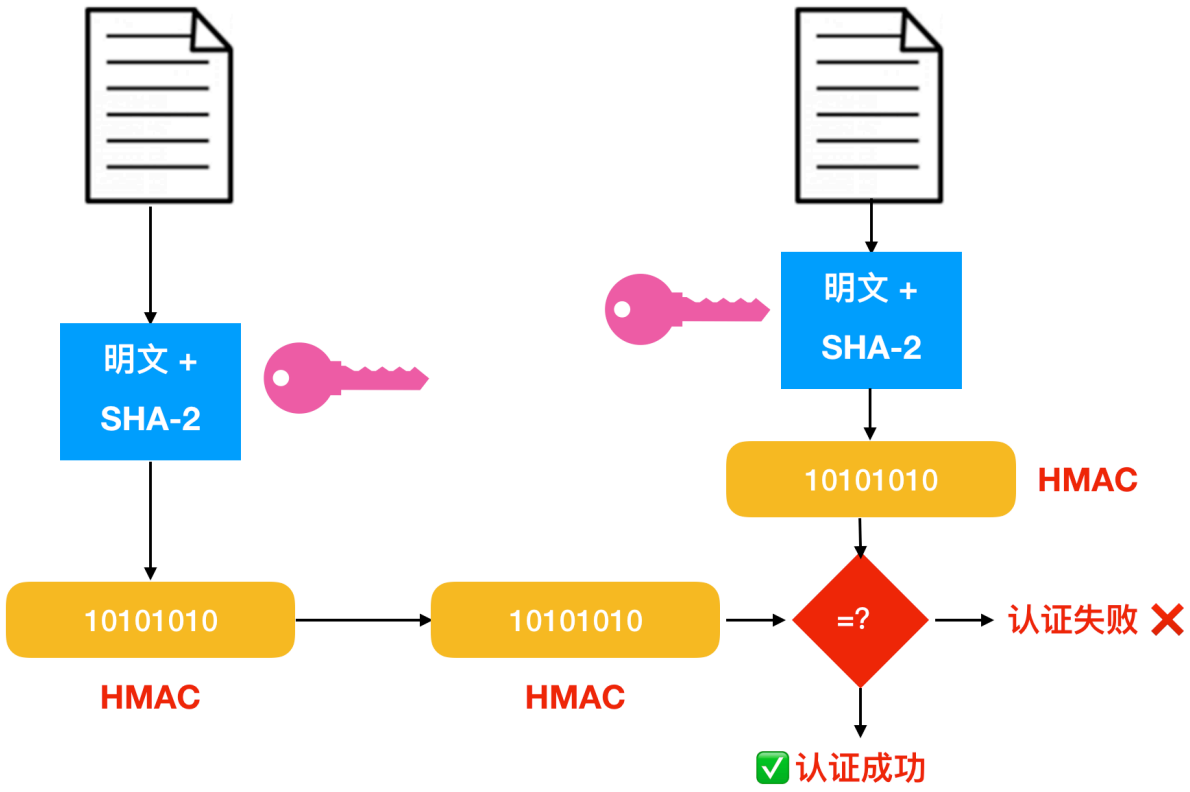
SHA-2 的全称是 `Secure Hash Algorithm 2` ，它在 2001 年被推出，它在 SHA-1 的基础上做了重大的修改，SHA-2 系列包含六个哈希函数，其摘要（哈希值）分别为 224、256、384 或 512 位：**SHA-224, SHA-256, SHA-384, SHA-512**。分别能够生成 28 字节、32 字节、48 字节、64 字节的摘要。

有了 SHA-2 的保护，就能够实现数据的完整性，哪怕你在文件中改变一个标点符号，增加一个空格，生成的文件摘要也会完全不同，不过 SHA-2 是基于明文的加密方式，还是不够安全，那应该用什么呢?

安全性更高的加密方式是使用 `HMAC`，在理解什么是 HMAC 前，你需要先知道一下什么是 MAC。

MAC 的全称是 `message authentication code`，它通过 MAC 算法从消息和密钥生成，MAC 值允许验证者（也拥有秘密密钥）检测到消息内容的任何更改，从而保护了消息的数据完整性。

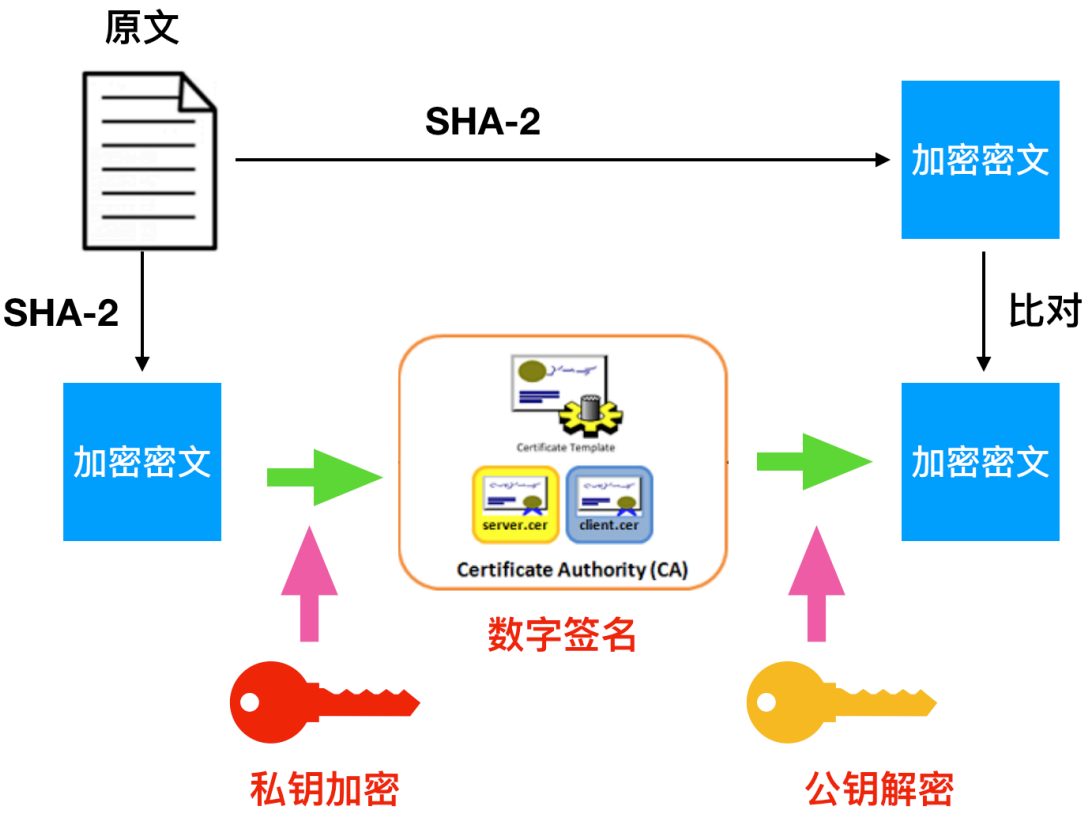
HMAC 是 MAC 更进一步的拓展，它是使用 MAC 值 + Hash 值的组合方式，HMAC 的计算中可以使用任何加密哈希函数，例如 SHA-256 等。



现在我们又解决了完整性的问题，那么就只剩下一个问题了，那就是 `认证`，认证怎么做的呢？我们再向服务器发送数据的过程中，黑客（攻击者）有可能伪装成任何一方来窃取信息。它可以伪装成你，来向服务器发送信息，也可以伪装称为服务器，接受你发送的信息。那么怎么解决这个问题呢?

认证

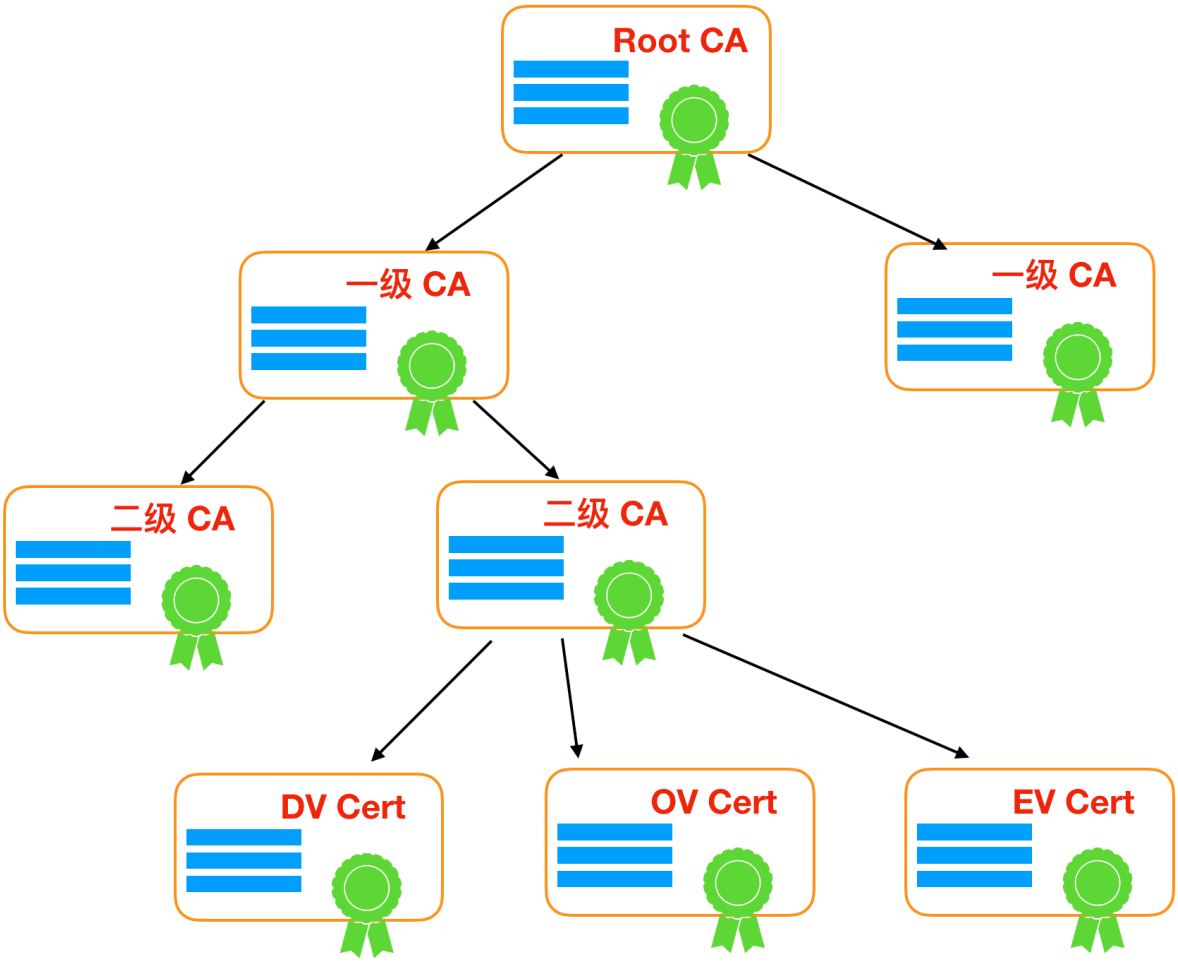
如何确定你自己的唯一性呢？我们在上面的叙述过程中出现过公钥加密，私钥解密的这个概念。提到的私钥只有你一个人所有，能够辨别唯一性，所以我们可以把顺序调换一下，变成私钥加密，公钥解密。使用私钥再加上摘要算法，就能够实现 `数字签名`，从而实现认证。



到现在，综合使用对称加密、非对称加密和摘要算法，我们已经实现了**加密、数据认证、认证**，那么是不是就安全了呢？非也，这里还存在一个**数字签名的认证问题**。因为私钥是是自己的，公钥是谁都可以发布，所以必须发布经过认证的公钥，才能解决公钥的信任问题。

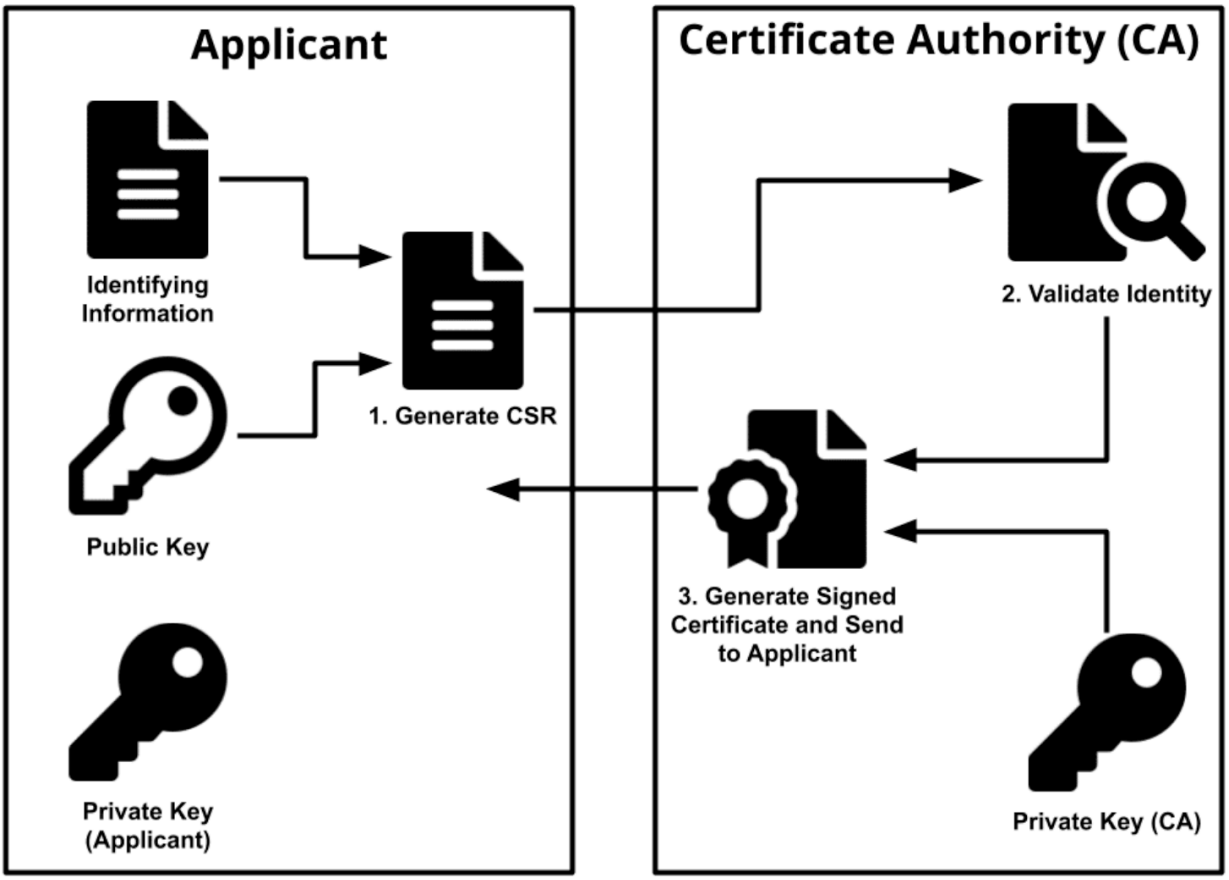
所以引入了 `CA`，CA 的全称是 `Certificate Authority`，证书认证机构，你必须让 CA 颁布具有认证过的公钥，才能解决公钥的信任问题。

全世界具有认证的 CA 就几家，分别颁布了 DV、OV、EV 三种，区别在于可信程度。DV 是最低的，只是域名级别的可信，EV 是最高的，经过了法律和审计的严格核查，可以证明网站拥有者的身份（在浏览器地址栏会显示出公司的名字，例如 Apple、GitHub 的网站）。不同的信任等级的机构一起形成了层级关系。



通常情况下，数字证书的申请人将生成由私钥和公钥以及证书 `签名请求（CSR）` 组成的密钥对。CSR是一个编码的文本文件，其中包含公钥和其他将包含在证书中的信息（例如域名，组织，电子邮件地址等）。密钥对和 CSR生成通常在将要安装证书的服务器上完成，并且 CSR 中包含的信息类型取决于证书的验证级别。与公钥不同，申请人的私钥是安全的，永远不要向 CA（或其他任何人）展示。

生成 CSR 后，申请人将其发送给 CA，CA 会验证其包含的信息是否正确，如果正确，则使用颁发的私钥对证书进行数字签名，然后将其发送给申请人。



Cookie 和 Session

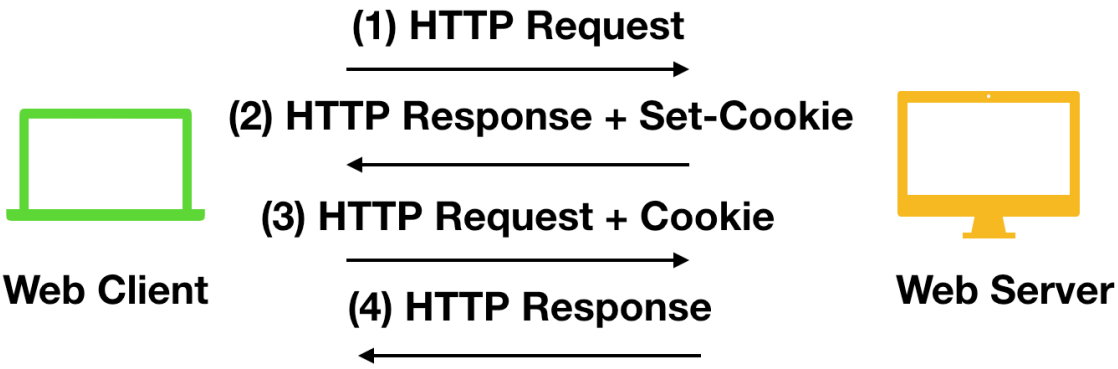
HTTP 协议是一种 `无状态协议` ，即每次服务端接收到客户端的请求时，都是一个全新的请求，服务器并不知道客户端的历史请求记录；Session 和 Cookie 的主要目的就是为了弥补 HTTP 的无状态特性。

Session 是什么

客户端请求服务端，服务端会为这次请求开辟一块 `内存空间` ，这个对象便是 Session 对象，存储结构为 `ConcurrentHashMap` 。Session 弥补了 HTTP 无状态特性，服务器可以利用 Session 存储客户端在同一个会话期间的一些操作记录。

Session 如何判断是否是同一会话

服务器第一次接收到请求时，开辟了一块 Session 空间（创建了Session对象），同时生成一个 sessionId ，并通过响应头的 **Set-Cookie：JSESSIONID=XXXXXXX** 命令，向客户端发送要求设置 Cookie 的响应； 客户端收到响应后，在本机客户端设置了一个 **JSESSIONID=XXXXXXX** 的 Cookie 信息，该 Cookie 的过期时间为浏览器会话结束；

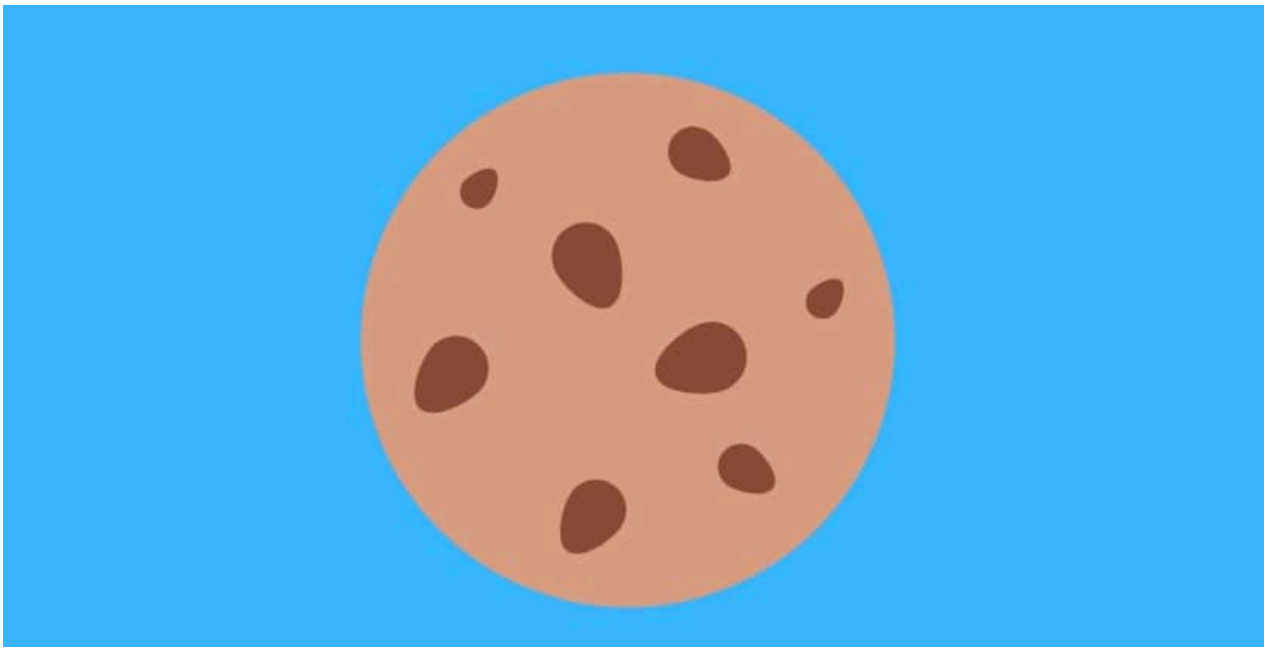


接下来客户端每次向同一个网站发送请求时，请求头都会带上该 Cookie信息（包含 sessionId ）， 然后，服务器通过读取请求头中的 Cookie 信息，获取名称为 JSESSIONID 的值，得到此次请求的 sessionId。

Session 的缺点

Session 机制有个缺点，比如 A 服务器存储了 Session，就是做了负载均衡后，假如一段时间内 A 的访问量激增，会转发到 B 进行访问，但是 B 服务器并没有存储 A 的 Session，会导致 Session 的失效。

Cookies 是什么



HTTP 协议中的 Cookie 包括 `Web Cookie` 和 `浏览器 Cookie`，它是服务器发送到 Web 浏览器的一小块数据。服务器发送到浏览器的 Cookie，浏览器会进行存储，并与下一个请求一起发送到服务器。通常，它用于判断两个请求是否来自于同一个浏览器，例如用户保持登录状态。

> HTTP Cookie 机制是 HTTP 协议无状态的一种补充和改良

Cookie 主要用于下面三个目的

- `会话管理`

登陆、购物车、游戏得分或者服务器应该记住的其他内容

- `个性化`

用户偏好、主题或者其他设置

- `追踪`

记录和分析用户行为

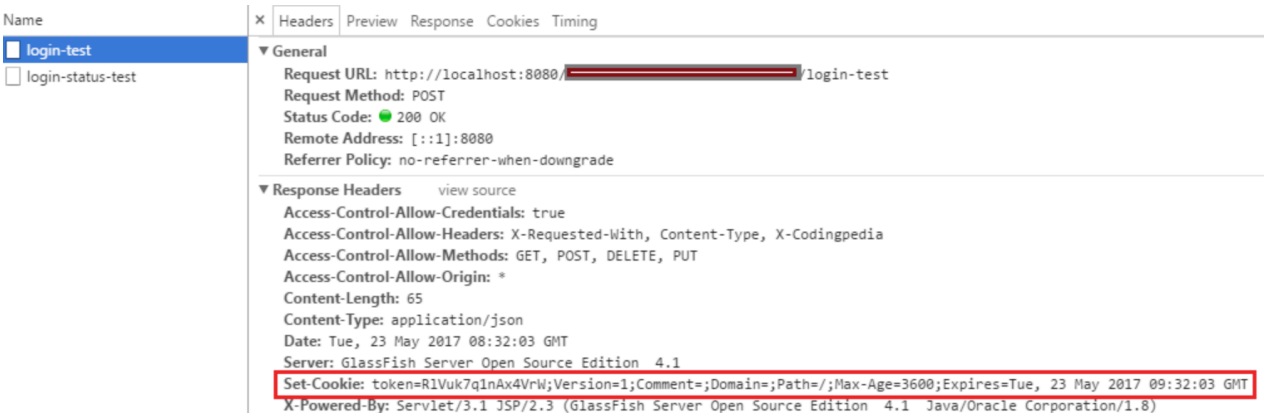
Cookie 曾经用于一般的客户端存储。虽然这是合法的，因为它们是在客户端上存储数据的唯一方法，但如今建议使用现代存储 API。Cookie 随每个请求一起发送，因此它们可能会降低性能（尤其是对于移动数据连接而言）。

创建 Cookie

当接收到客户端发出的 HTTP 请求时，服务器可以发送带有响应的 `Set-Cookie` 标头，Cookie 通常由浏览器存储，然后将 Cookie 与 HTTP 标头一同向服务器发出请求。

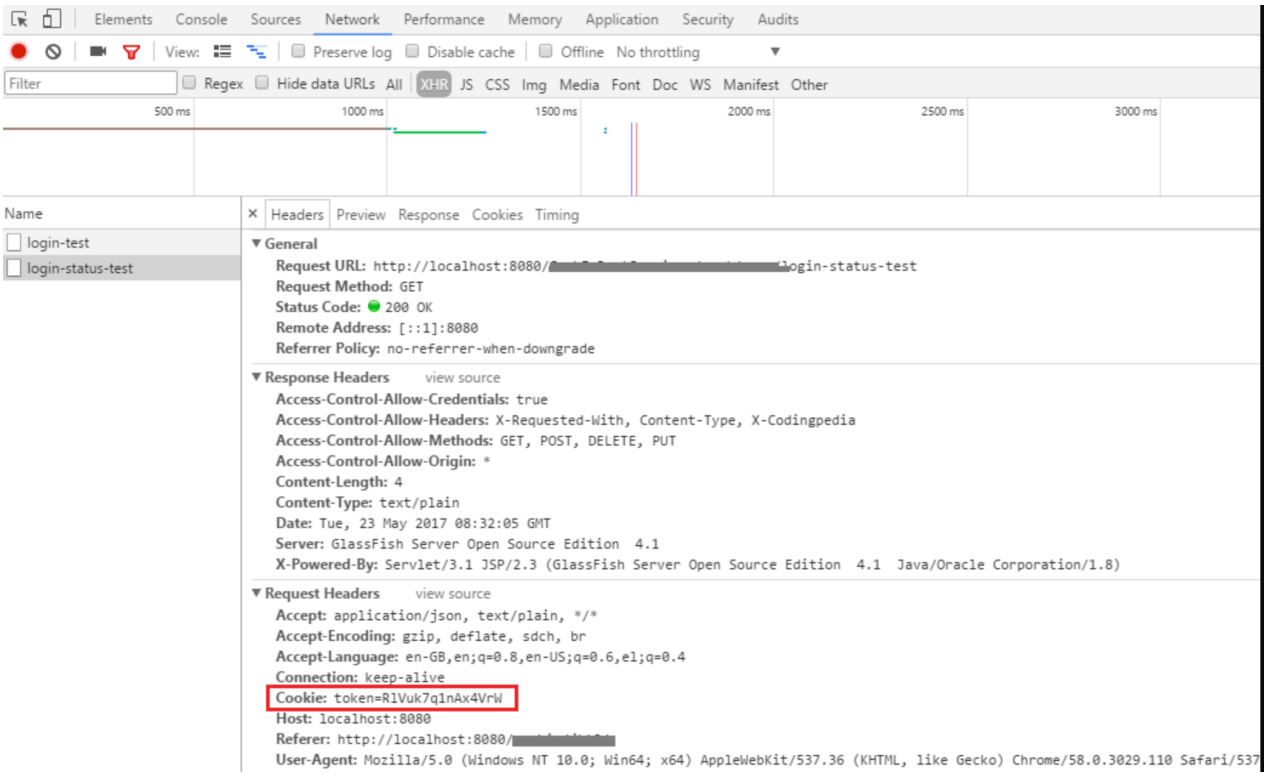
Set-Cookie 和 Cookie 标头

`Set-Cookie` HTTP 响应标头将 cookie 从服务器发送到用户代理。下面是一个发送 Cookie 的例子



此标头告诉客户端存储 Cookie

现在，随着对服务器的每个新请求，浏览器将使用 Cookie 头将所有以前存储的 Cookie 发送回服务器。



有两种类型的 Cookies，一种是 Session Cookies，一种是 Persistent Cookies，如果 Cookie 不包含到期日期，则将其视为会话 Cookie。会话 Cookie 存储在内存中，永远不会写入磁盘，当浏览器关闭时，此后 Cookie 将永久丢失。如果 Cookie 包含 `有效期` ，则将其视为持久性 Cookie。在到期指定的日期，Cookie 将从磁盘中删除。

还有一种是 `Cookie的 Secure 和 HttpOnly 标记` ，下面依次来介绍一下

会话 Cookies

上面的示例创建的是会话 Cookie ，会话 Cookie 有个特征，客户端关闭时 Cookie 会删除，因为它没有指定 `Expires` 或 `Max-Age` 指令。

但是，Web 浏览器可能会使用会话还原，这会使大多数会话 Cookie 保持永久状态，就像从未关闭过浏览器一样。

永久性 Cookies

永久性 Cookie 不会在客户端关闭时过期，而是在 `特定日期（Expires）` 或 `特定时间长度（Max-Age）` 外过期。例如

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
```

Cookie 的 Secure 和 HttpOnly 标记

安全的 Cookie 需要经过 HTTPS 协议通过加密的方式发送到服务器。即使是安全的，也不应该将敏感信息存储在cookie 中，因为它们本质上是不安全的，并且此标志不能提供真正的保护。

HttpOnly 的作用

- 会话 Cookie 中缺少 HttpOnly 属性会导致攻击者可以通过程序(JS脚本、Applet等)获取到用户的 Cookie 信息，造成用户 Cookie 信息泄露，增加攻击者的跨站脚本攻击威胁。
- HttpOnly 是微软对 Cookie 做的扩展，该值指定 Cookie 是否可通过客户端脚本访问。

- 如果在 Cookie 中没有设置 HttpOnly 属性为 true，可能导致 Cookie 被窃取。窃取的 Cookie 可以包含标识站点用户的敏感信息，如 ASP.NET 会话 ID 或 Forms 身份验证票证，攻击者可以重播窃取的 Cookie，以便伪装成用户或获取敏感信息，进行跨站脚本攻击等。

Cookie 的作用域

`Domain` 和 `Path` 标识定义了 Cookie 的作用域：即 Cookie 应该发送给哪些 URL。

`Domain` 标识指定了哪些主机可以接受 Cookie。如果不指定，默认为当前主机(**不包含子域名**）。如果指定了 `Domain`，则一般包含子域名。

例如，如果设置 `Domain=mozilla.org`，则 Cookie 也包含在子域名中（如 `developer.mozilla.org`）。

例如，设置 `Path=/docs`，则以下地址都会匹配：

- `/docs`
- `/docs/Web/`
- `/docs/Web/HTTP`

JSON Web Token 和 Session Cookies 的对比

`JSON Web Token，简称 JWT`，它和 `Session` 都可以为网站提供用户的身份认证，但是它们不是一回事。

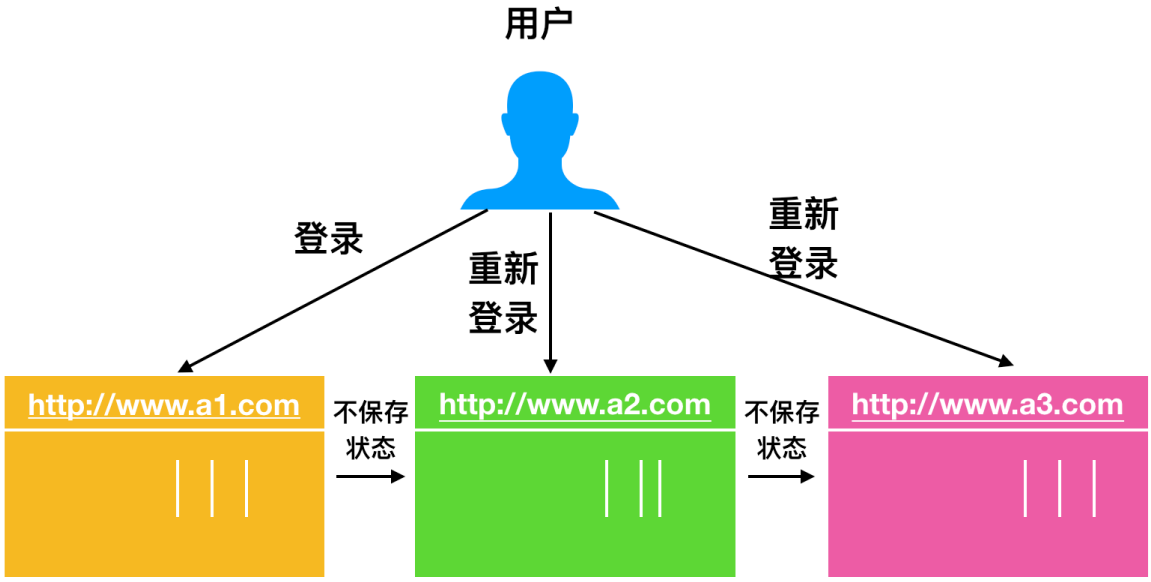
下面是 JWT 和 Session 不同之处的研究

JWT 和 Session Cookies 的相同之处

在探讨 JWT 和 Session Cookies 之前，有必要需要先去理解一下它们的相同之处。

它们既可以对用户进行身份验证，也可以用来在用户单击进入不同页面时以及登陆网站或应用程序后进行身份验证。

如果没有这两者，那你可能需要在每个页面切换时都需要进行登录了。因为 HTTP 是一个无状态的协议。这也就意味着当你访问某个网页，然后单击同一站点上的另一个页面时，服务器的 `内存中` 将不会记住你之前的操作。



因此，如果你登录并访问了你有权访问的另一个页面，由于 HTTP 不会记录你刚刚登录的信息，因此你将再次登录。

JWT 和 Session Cookies 就是用来处理在不同页面之间切换，保存用户登录信息的机制。

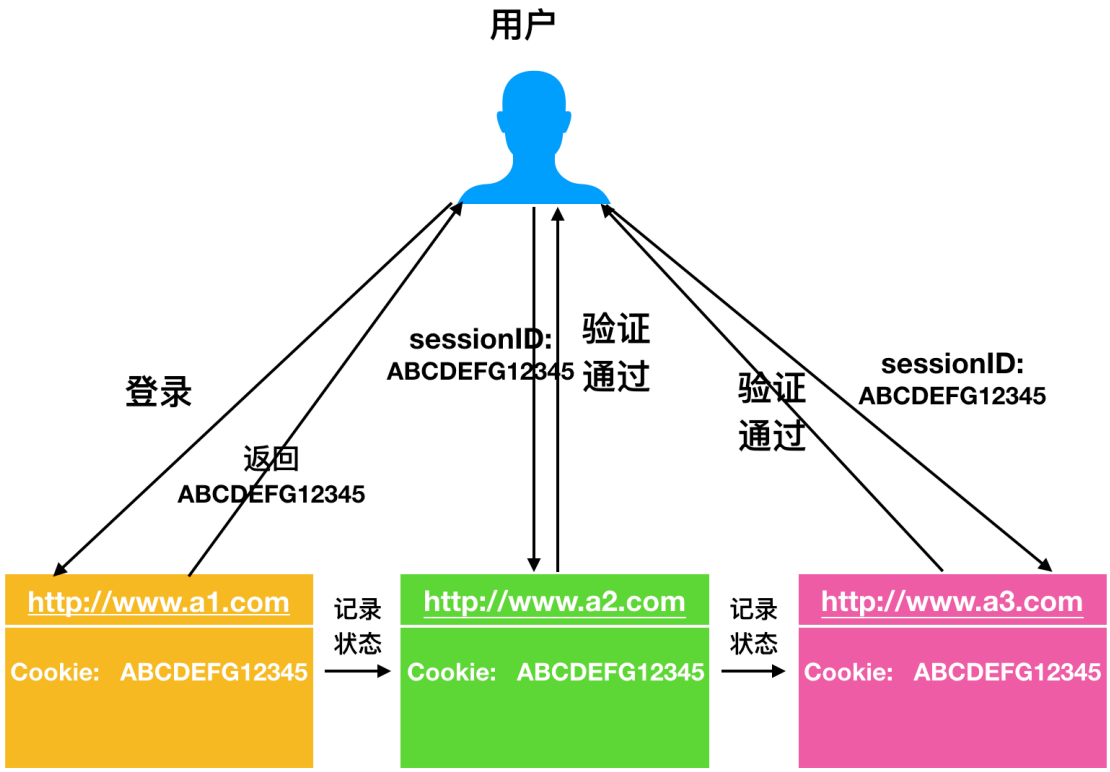
也就是说，这两种技术都是用来保存你的登录状态，能够让你在浏览任意受密码保护的网站。通过在每次产生新的请求时对用户数据进行身份验证来解决此问题。

所以 JWT 和 Session Cookies 的相同之处是什么？那就是它们能够支持你在发送不同请求之间，记录并验证你的登录状态的一种机制。

什么是 Session Cookies

Session Cookies 也称为 `会话 Cookies`，在 Session Cookies 中，用户的登录状态会保存在 `服务器` 的 `内存` 中。当用户登录时，Session 就被服务端安全的创建。

在每次请求时，服务器都会从会话 Cookie 中读取 SessionId，如果服务端的数据和读取的 SessionId 相同，那么服务器就会发送响应给浏览器，允许用户登录。



什么是 Json Web Tokens

Json Web Token 的简称就是 JWT，通常可以称为 `Json 令牌` 。它是 `RFC 7519` 中定义的用于 `安全的` 将信息作为 `Json 对象` 进行传输的一种形式。JWT 中存储的信息是经过 `数字签名` 的，因此可以被信任和理解。可以使用 HMAC 算法或使用 RSA/ECDSA 的公用/专用密钥对 JWT 进行签名。

使用 JWT 主要用来下面两点

- `认证(Authorization)` ：这是使用 JWT 最常见的一种情况，一旦用户登录，后面每个请求都会包含 JWT，从而允许用户访问该令牌所允许的路由、服务和资源。 `单点登录` 是当今广泛使用 JWT 的一项功能，因为它的开销很小。
- `信息交换(Information Exchange)` ：JWT 是能够安全传输信息的一种方式。通过使用公钥/私钥对 JWT 进行签名认证。此外，由于签名是使用 `head` 和 `payload` 计算的，因此你还可以验证内容是否遭到篡改。

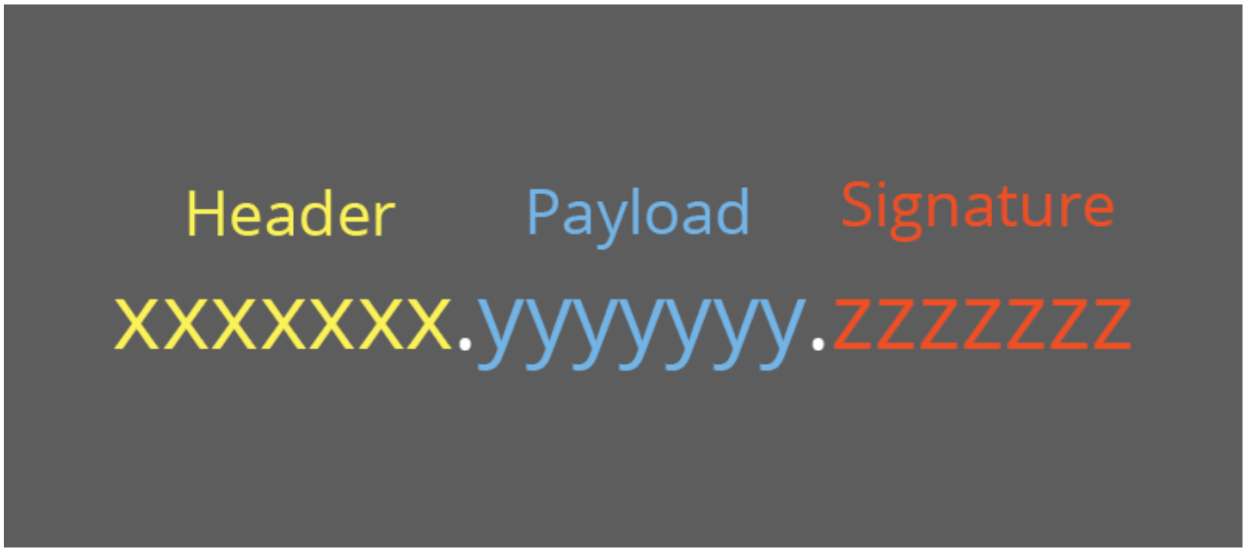
JWT 的格式

下面，我们会探讨一下 JWT 的组成和格式是什么

JWT 主要由三部分组成，每个部分用 `.` 进行分割，各个部分分别是

- `Header`
- `Payload`
- `Signature`

因此，一个非常简单的 JWT 组成会是下面这样



然后我们分别对不同的部分进行探讨。

Header

Header 是 JWT 的标头，它通常由两部分组成：`令牌的类型(即 JWT)`和使用的`签名算法`，例如 HMAC SHA256 或 RSA。

例如

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

指定类型和签名算法后，Json 块被`Base64Url`编码形成 JWT 的第一部分。

Payload

Token 的第二部分是`Payload`，Payload 中包含一个声明。声明是有关实体（通常是用户）和其他数据的声明。共有三种类型的声明：**registered, public 和 private** 声明。

- `registered 声明`： 包含一组建议使用的预定义声明，主要包括

ISS	签发人
iss (issuer)	签发人
exp (expiration time)	过期时间
sub (subject)	主题
aud (audience)	受众
nbf (Not Before)	生效时间
iat (Issued At)	签发时间
jti (JWT ID)	编号

- `public 声明`：公共的声明，可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息，但不建议添加敏感信息，因为该部分在客户端可解密。
- `private 声明`：自定义声明，旨在在同意使用它们的各方之间共享信息，既不是注册声明也不是公共声明。

例如

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

然后 payload Json 块会被 `Base64Url` 编码形成 JWT 的第二部分。

signature

JWT 的第三部分是一个签证信息，这个签证信息由三部分组成

- header (base64后的)
- payload (base64后的)
- secret

比如我们需要 HMAC SHA256 算法进行签名

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

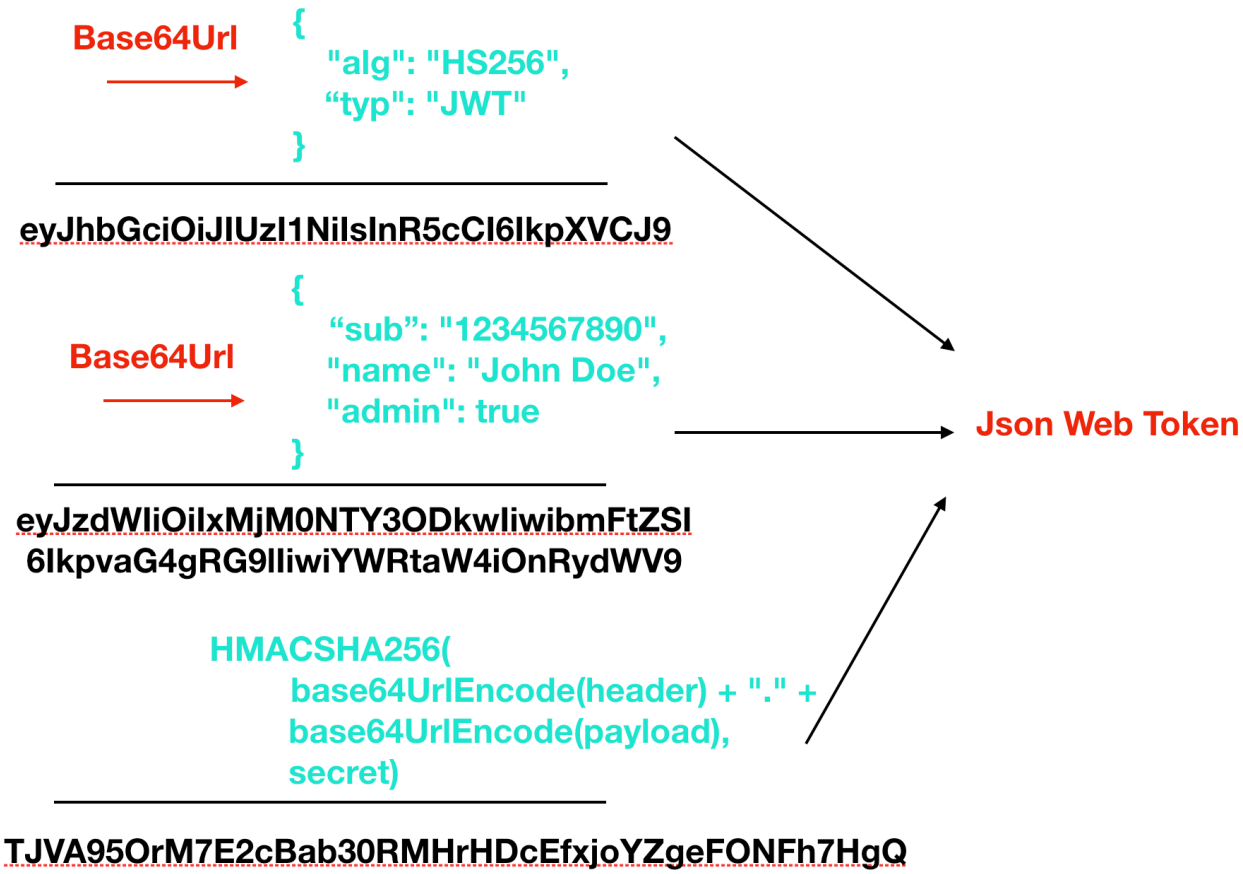
签名用于验证消息在此过程中没有更改，并且对于使用私钥进行签名的令牌，它还可以验证 JWT 的发送者的真实身份

拼凑在一起

现在我们把上面的三个由点分隔的 Base64-URL 字符串部分组成在一起，这个字符串可以在 HTML 和 HTTP 环境中轻松传递这些字符串。

下面是一个完整的 JWT 示例，它对 header 和 payload 进行编码，然后使用 signature 进行签名

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```



如果想自己测试编写的话，可以访问 JWT 官网 **https://jwt.io/#debugger-io**

JWT 和 Session Cookies 的不同

JWT 和 Session Cookies 都提供安全的用户身份验证，但是它们有以下几点不同

密码签名

JWT 具有加密签名，而 Session Cookies 则没有。

JSON 是无状态的

JWT 是 `无状态` 的，因为声明被存储在 `客户端` ，而不是服务端内存中。

身份验证可以在 `本地` 进行，而不是在请求必须通过服务器数据库或类似位置中进行。 这意味着可以对用户进行多次身份验证，而无需与站点或应用程序的数据库进行通信，也无需在此过程中消耗大量资源。

可扩展性

Session Cookies 是存储在服务器内存中，这就意味着如果网站或者应用很大的情况下会耗费大量的资源。由于 JWT 是无状态的，在许多情况下，它们可以节省服务器资源。因此 JWT 要比 Session Cookies 具有更强的 `可扩展性` 。

JWT 支持跨域认证

Session Cookies 只能用在 `单个节点的域` 或者它的 `子域` 中有效。如果它们尝试通过第三个节点访问，就会被禁止。如果你希望自己的网站和其他站点建立安全连接时，这是一个问题。

使用 JWT 可以解决这个问题，使用 JWT 能够通过 `多个节点` 进行用户认证，也就是我们常说的 `跨域认证` 。

JWT 和 Session Cookies 的选型

我们上面探讨了 JWT 和 Cookies 的不同点，相信你也会对选型有了更深的认识，大致来说

对于只需要登录用户并访问存储在站点数据库中的一些信息的中小型网站来说，Session Cookies 通常就能满足。

如果你有企业级站点，应用程序或附近的站点，并且需要处理大量的请求，尤其是第三方或很多第三方（包括位于不同域的API），则 JWT 显然更适合。

后记

前两天面试的时候问到了这个题，所以写篇文章总结一下，还问到了一个面试题，**禁用 Cookies，如何使用 Session** ？网上百度了一下，发现这是 PHP 的面试题.....

但还是选择了解了一下，如何禁用 Cookies 后，使用 Session

- 如果禁用了 Cookies，服务器仍会将 sessionId 以 cookie 的方式发送给浏览器，但是，浏览器不再保存这个cookie (即sessionId) 了。
- 如果想要继续使用 session，需要采用 `URL 重写` 的方式来实现，可以参考 **https://www.cnblogs.com/Renyi-Fan/p/11012086.html**

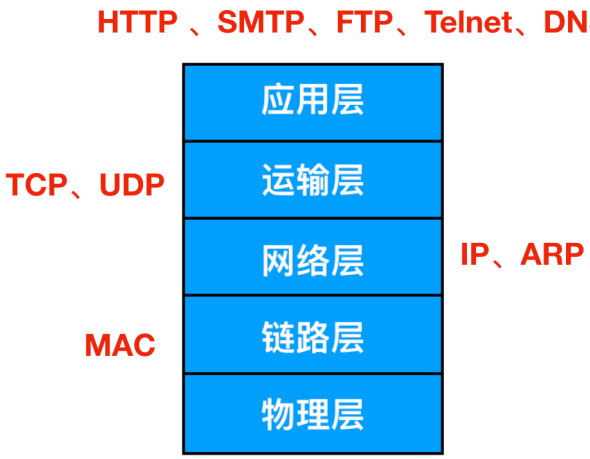
HTTP 和 HTTPS 的区别

HTTP 是一种 `超文本传输协议(Hypertext Transfer Protocol)` ，**HTTP 是一个在计算机世界里专门在两点之间传输文字、图片、音频、视频等超文本数据的约定和规范**

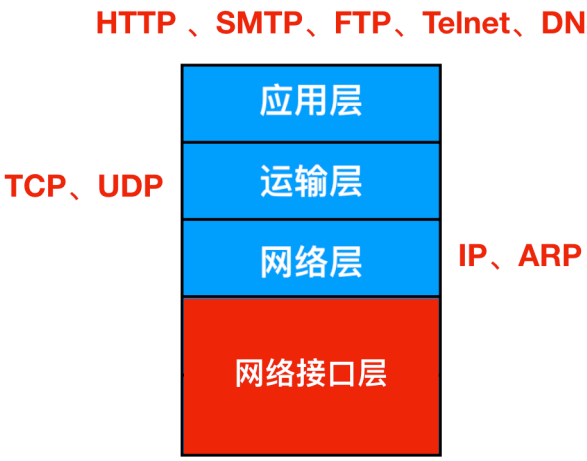
HTTP 主要内容分为三部分，**超文本（Hypertext）、传输（Transfer）、协议（Protocol）**。

- 超文本就是不单单只是本文，它还可以传输图片、音频、视频，甚至点击文字或图片能够进行 `超链接` 的跳转。
- 上面这些概念可以统称为数据，传输就是数据需要经过一系列的物理介质从一个端系统传送到另外一个端系统的过程。通常我们把传输数据包的一方称为 `请求方`，把接到二进制数据包的一方称为 `应答方`。
- 而协议指的就是是网络中(包括互联网)传递、管理信息的一些规范。如同人与人之间相互交流是需要遵循一定的规矩一样，计算机之间的相互通信需要共同遵守一定的规则，这些规则就称为协议，只不过是网络协议。

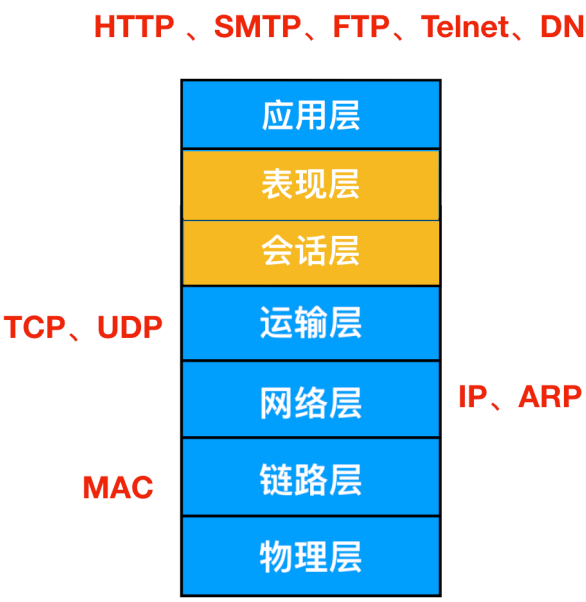
说到 HTTP，不得不提的就是 TCP/IP 网络模型，一般是五层模型。如下图所示



但是也可以分为四层，就是**把链路层和物理层都表示为网络接口层**

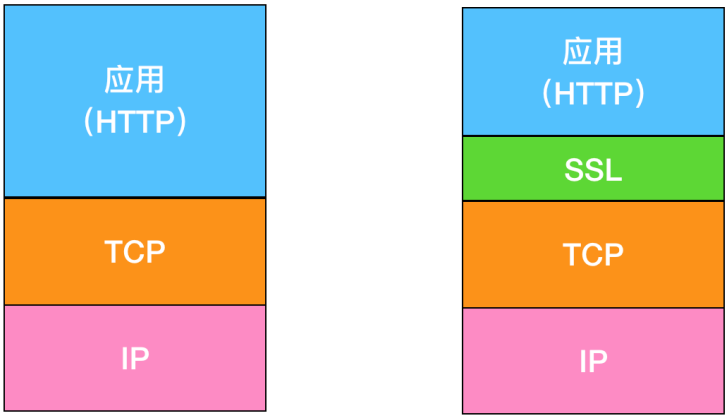


还有一种就是 OSI 七层网络模型，它就是在五层协议之上加了**表示层和会话层**



而 HTTPS 的全称是 `Hypertext Transfer Protocol Secure`，从名称我们可以看出 HTTPS 要比 HTTPS 多了 secure 安全性这个概念，实际上， HTTPS 并不是一个新的应用层协议，它其实就是 HTTP + TLS/SSL 协议组合而成，而安全性的保证正是 TLS/SSL 所做的工作。

也就是说，**HTTPS 就是身披了一层 SSL 的 HTTP**。

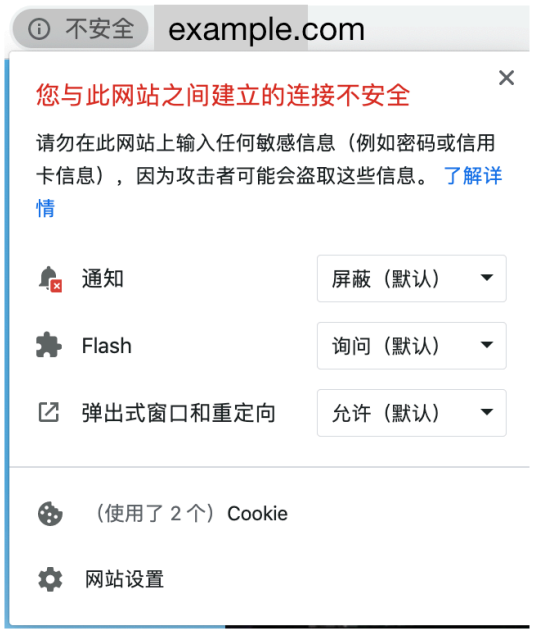


那么，HTTP 和 HTTPS 的主要区别是什么呢?

- 最简单的，HTTP 在地址栏上的协议是以 `http://` 开头，而 HTTPS 在地址栏上的协议是以 `https://` 开头

```
http://www.cxuanblog.com/
https://www.cxuanblog.com/
```

- HTTP 是未经安全加密的协议，它的传输过程容易被攻击者监听、数据容易被窃取、发送方和接收方容易被伪造；而 HTTPS 是安全的协议，它通过 **密钥交换算法 - 签名算法 - 对称加密算法 - 摘要算法** 能够解决上面这些问题。



- HTTP 的默认端口是 80，而 HTTPS 的默认端口是 443。

HTTP Get 和 Post 区别

HTTP 中包括许多方法，**Get 和 Post 是 HTTP 中最常用的两个方法**，基本上使用 HTTP 方法中有 99% 都是在使用 Get 方法和 Post 方法，所以有必要我们对这两个方法有更加深刻的认识。

- get 方法一般用于请求，比如你在浏览器地址栏输入 `www.cxuanblog.com` 其实就是发送了一个 get 请求，它的主要特征是请求服务器返回资源，而 post 方法一般用于 `<form> 表单` 的提交，相当于是把信息提交给服务器，等待服务器作出响应，get 相当于一个是 pull/拉的操作，而 post 相当于是一个 push/推的操作。
- get 方法是不安全的，因为你在发送请求的过程中，你的请求参数会拼在 URL 后面，从而导致容易被攻击者窃取，对你的信息造成破坏和伪造；

```
/test/demo_form.asp?name1=value1&name2=value2
```

而 post 方法是把参数放在请求体 body 中的，这对用户来说不可见。

```
POST /test/demo_form.asp HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2
```

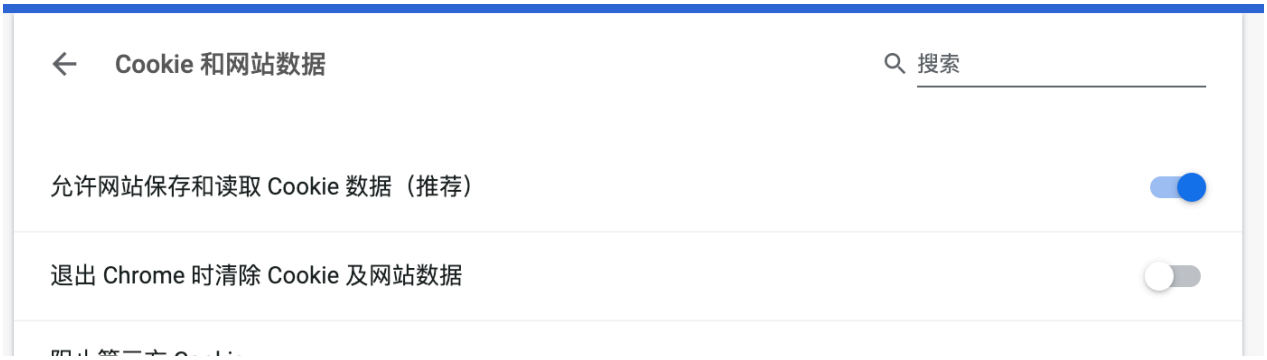
- get 请求的 URL 有长度限制，而 post 请求会把参数和值放在消息体中，对数据长度没有要求。
- get 请求会被浏览器主动 cache，而 post 不会，除非手动设置。
- get 请求在浏览器反复的 `回退/前进` 操作是无害的，而 post 操作会再次提交表单请求。
- get 请求在发送过程中会产生一个 TCP 数据包；post 在发送过程中会产生两个 TCP 数据包。对于 get 方式的请求，浏览器会把 http header 和 data 一并发送出去，服务器响应 200（返回数据）；而对于 post，浏览器先发送 header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok（返回数据）。

什么是无状态协议，HTTP 是无状态协议吗，怎么解决

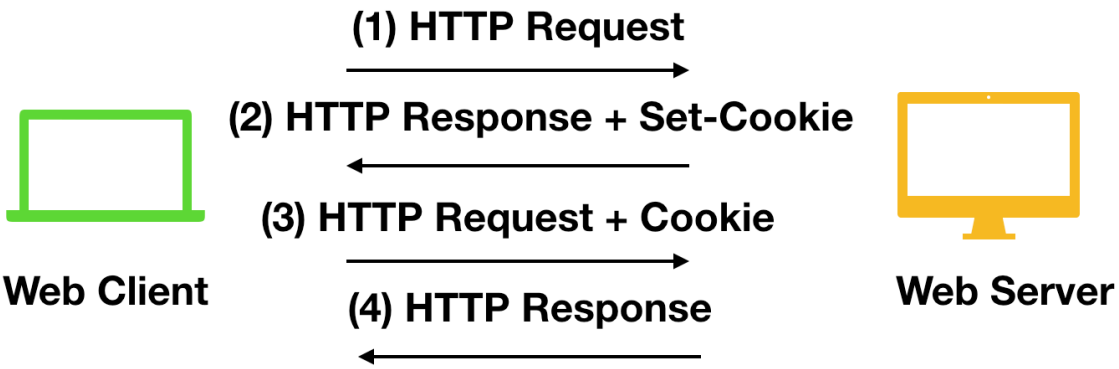
`无状态协议(Stateless Protocol)` 就是指**浏览器对于事务的处理没有记忆能力**。举个例子来说就是比如客户请求获得网页之后关闭浏览器，然后再次启动浏览器，登录该网站，但是服务器并不知道客户关闭了一次浏览器。

HTTP 就是一种无状态的协议，他对用户的操作没有记忆能力。可能大多数用户不相信，他可能觉得每次输入用户名和密码登陆一个网站后，下次登陆就不再重新输入用户名和密码了。这其实不是 HTTP 做的事情，起作用的是一个叫做 `小甜饼(Cookie)` 的机制。它能够让浏览器具有 `记忆` 能力。

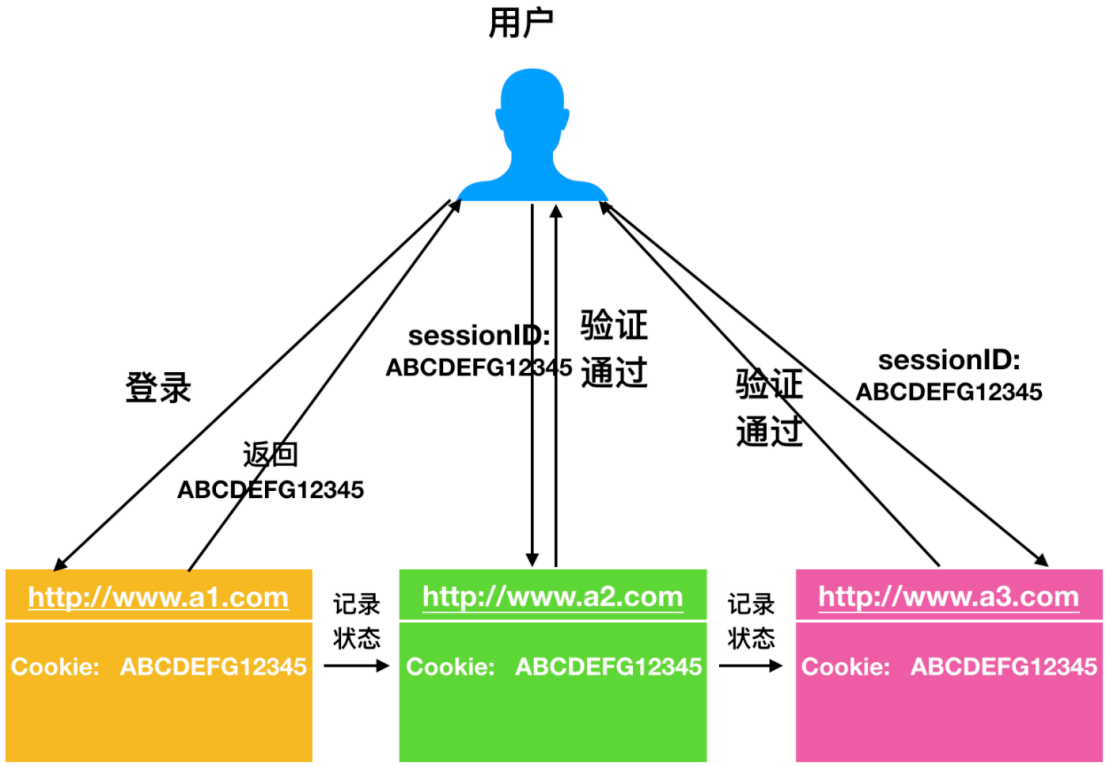
如果你的浏览器允许 cookie 的话，查看方式 **chrome://settings/content/cookies**



也就说明你的记忆芯片通电了..... 当你想服务端发送请求时，服务端会给你发送一个认证信息，服务器第一次接收到请求时，开辟了一块 Session 空间（创建了Session对象），同时生成一个 sessionId，并通过响应头的 **Set-Cookie：JSESSIONID=XXXXXXX** 命令，向客户端发送要求设置 Cookie 的响应； 客户端收到响应后，在本机客户端设置了一个 **JSESSIONID=XXXXXXX** 的 Cookie 信息，该 Cookie 的过期时间为浏览器会话结束；



接下来客户端每次向同一个网站发送请求时，请求头都会带上该 Cookie信息（包含 sessionId ）， 然后，服务器通过读取请求头中的 Cookie 信息，获取名称为 JSESSIONID 的值，得到此次请求的 sessionId。这样，你的浏览器才具有了记忆能力。



还有一种方式是使用 JWT 机制，它也是能够让你的浏览器具有记忆能力的一种机制。与 Cookie 不同，JWT 是保存在客户端的信息，它广泛的应用于单点登录的情况。JWT 具有两个特点

- JWT 的 Cookie 信息存储在 `客户端` ，而不是服务端内存中。也就是说，JWT 直接本地进行验证就可以，验证完毕后，这个 Token 就会在 Session 中随请求一起发送到服务器，通过这种方式，可以节省服务器资源，并且 token 可以进行多次验证。
- JWT 支持跨域认证，Cookies 只能用在 `单个节点的域` 或者它的 `子域` 中有效。如果它们尝试通过第三个节点访问，就会被禁止。使用 JWT 可以解决这个问题，使用 JWT 能够通过 `多个节点` 进行用户认证，也就是我们常说的 `跨域认证` 。

UDP 和 TCP 的区别

TCP 和 UDP 都位于计算机网络模型中的运输层，它们负责传输应用层产生的数据。下面我们就来聊一聊 TCP 和 UDP 分别的特征和他们的区别

UDP 是什么

UDP 的全称是 `User Datagram Protocol` ，用户数据报协议。它不需要所谓的 `握手` 操作，从而加快了通信速度，允许网络上的其他主机在接收方同意通信之前进行数据传输。

> 数据报是与分组交换网络关联的传输单元。

UDP 的特点主要有

- UDP 能够支持容忍数据包丢失的带宽密集型应用程序
- UDP 具有低延迟的特点
- UDP 能够发送大量的数据包

- UDP 能够允许 DNS 查找，DNS 是建立在 UDP 之上的应用层协议。

TCP 是什么

TCP 的全称是 `Transmission Control Protocol` ，传输控制协议。它能够帮助你确定计算机连接到 Internet 以及它们之间的数据传输。通过三次握手来建立 TCP 连接，三次握手就是用来启动和确认 TCP 连接的过程。一旦连接建立后，就可以发送数据了，当数据传输完成后，会通过关闭虚拟电路来断开连接。

TCP 的主要特点有

- TCP 能够确保连接的建立和数据包的发送
- TCP 支持错误重传机制
- TCP 支持拥塞控制，能够在网络拥堵的情况下延迟发送
- TCP 能够提供错误校验和，甄别有害的数据包。

TCP 和 UDP 的不同

下面为你罗列了一些 TCP 和 UDP 的不同点，方便理解，方便记忆。

TCP	UDP
TCP 是面向连接的协议	UDP 是无连接的协议
TCP 在发送数据前先需要建立连接，然后再发送数据	UDP 无需建立连接就可以直接发送大量数据
TCP 会按照特定顺序重新排列数据包	UDP 数据包没有固定顺序，所有数据包都相互独立
TCP 传输的速度比较慢	UDP 的传输会更快
TCP 的头部字节有 20 字节	UDP 的头部字节只需要 8 个字节
TCP 是重量级的，在发送任何用户数据之前，TCP需要三次握手建立连接。	UDP 是轻量级的。没有跟踪连接，消息排序等。
TCP 会进行错误校验，并能够进行错误恢复	UDP 也会错误检查，但会丢弃错误的数据包。
TCP 有发送确认	UDP 没有发送确认
TCP 会使用握手协议，例如 SYN，SYN-ACK，ACK	无握手协议
TCP 是可靠的，因为它可以确保将数据传送到路由器。	在 UDP 中不能保证将数据传送到目标。

TCP 三次握手和四次挥手

TCP 三次握手和四次挥手也是面试题的热门考点，它们分别对应 TCP 的连接和释放过程。下面就来简单认识一下这两个过程

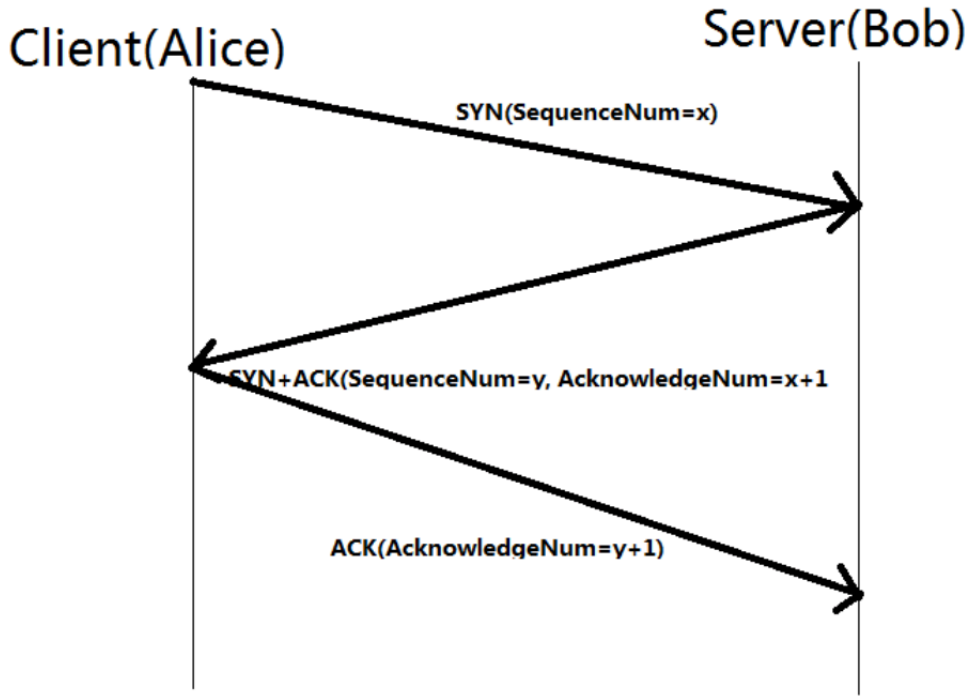
TCP 三次握手

在了解具体的流程前，我们需要先认识几个概念

消息类型	描述
SYN	这个消息是用来初始化和建立连接的。
ACK	帮助对方确认收到的 SYN 消息
SYN-ACK	本地的 SYN 消息和较早的 ACK 数据包
FIN	用来断开连接

- SYN：它的全称是 `Synchronize Sequence Numbers` ，同步序列编号。是 TCP/IP 建立连接时使用的握手信号。在客户机和服务器之间建立 TCP 连接时，首先会发送的一个信号。客户端在接受到 SYN 消息时，就会在自己的段内生成一个随机值 X。

- SYN-ACK：服务器收到 SYN 后，打开客户端连接，发送一个 SYN-ACK 作为答复。确认号设置为比接收到的序列号多一个，即 X + 1，服务器为数据包选择的序列号是另一个随机数 Y。
- ACK： `Acknowledge character` ，确认字符，表示发来的数据已确认接收无误。最后，客户端将 ACK 发送给服务器。序列号被设置为所接收的确认值即 Y + 1。



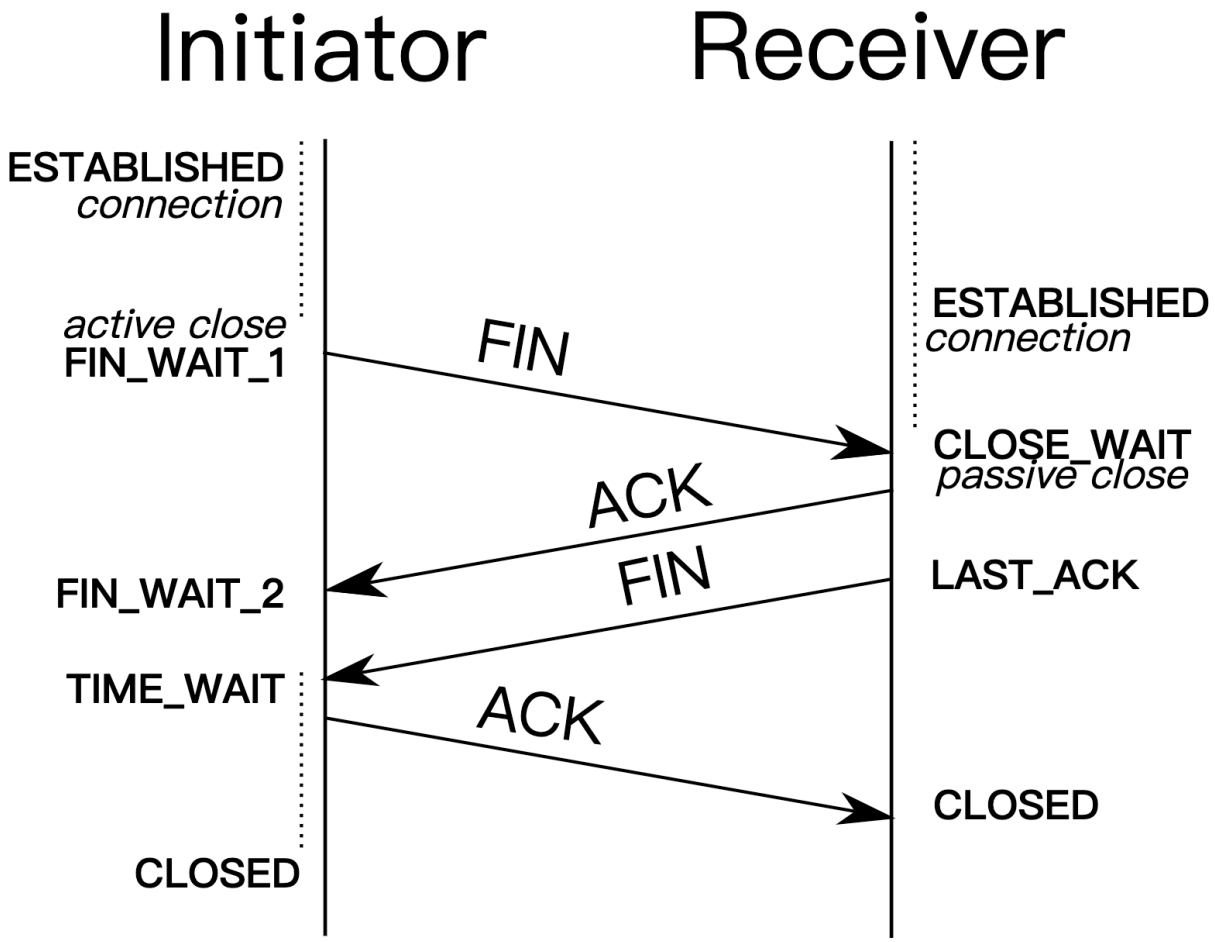
如果用现实生活来举例的话就是

小明 - 客户端 小红 - 服务端

- 小明给小红打电话，接通了后，小明说**喂，能听到吗**，这就相当于是连接建立。
- 小红给小明回应，**能听到，你能听到我说的话吗**，这就相当于是请求响应。
- 小明听到小红的回应后，**好的**，这相当于是连接确认。在这之后小明和小红就可以通话/交换信息了。

TCP 四次挥手

在连接终止阶段使用四次挥手，连接的每一端都会独立的终止。下面我们来描述一下这个过程。



- 首先，客户端应用程序决定要终止连接(这里服务端也可以选择断开连接)。这会使客户端将 FIN 发送到服务器，并进入 `FIN_WAIT_1` 状态。当客户端处于 FIN_WAIT_1 状态时，它会等待来自服务器的 ACK 响应。
- 然后第二步，当服务器收到 FIN 消息时，服务器会立刻向客户端发送 ACK 确认消息。
- 当客户端收到服务器发送的 ACK 响应后，客户端就进入 `FIN_WAIT_2` 状态，然后等待来自服务器的 `FIN` 消息
- 服务器发送 ACK 确认消息后，一段时间（可以进行关闭后）会发送 FIN 消息给客户端，告知客户端可以进行关闭。
- 当客户端收到从服务端发送的 FIN 消息时，客户端就会由 FIN_WAIT_2 状态变为 `TIME_WAIT` 状态。处于 TIME_WAIT 状态的客户端允许重新发送 ACK 到服务器为了防止信息丢失。客户端在 TIME_WAIT 状态下花费的时间取决于它的实现，在等待一段时间后，连接关闭，客户端上所有的资源（包括端口号和缓冲区数据）都被释放。

还是可以用上面那个通话的例子来进行描述

- 小明对小红说，我所有的东西都说完了，我要挂电话了。
- 小红说，收到，我这边还有一些东西没说。
- 经过若干秒后，小红也说完了，小红说，我说完了，现在可以挂断了
- 小明收到消息后，又等了若干时间后，挂断了电话。

请你说一下 HTTP 常见的请求头

这个问题比较开放，因为 HTTP 请求头有很多，这里只简单举出几个例子，具体的可以参考我的另一篇文章

https://mp.weixin.qq.com/s/XZZR0945IcI6X4S0g5fZXg

HTTP 标头会分为四种，分别是 `通用标头` 、 `实体标头` 、 `请求标头` 、 `响应标头` 。分别介绍一下

通用标头

通用标头主要有三个，分别是 `Date` 、 `Cache-Control` 和 `Connection`

Date

Date 是一个通用标头，它可以出现在请求标头和响应标头中，它的基本表示如下

```
Date: Wed, 21 Oct 2015 07:28:00 GMT
```

表示的是格林威治标准时间，这个时间要比北京时间慢八个小时

Cache-Control

Cache-Control 是一个通用标头，他可以出现在 `请求标头` 和 `响应标头` 中，Cache-Control 的种类比较多，虽然说这是一个通用标头，但是又一些特性是请求标头具有的，有一些是响应标头才有的。主要大类有 `可缓存性` 、 `阈值性` 、 `重新验证并重新加载` 和 `其他特性`

Connection

Connection 决定当前事务（一次三次握手和四次挥手）完成后，是否会关闭网络连接。Connection 有两种，一种是 `持久性连接` ，即一次事务完成后不关闭网络连接

```
Connection: keep-alive
```

另一种是 `非持久性连接` ，即一次事务完成后关闭网络连接

```
Connection: close
```

HTTP1.1 其他通用标头如下

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

实体标头

实体标头是描述消息正文内容的 HTTP 标头。实体标头用于 HTTP 请求和响应中。头部 `Content-Length` 、 `Content-Language` 、 `Content-Encoding` 是实体头。

- Content-Length 实体报头指示实体主体的大小，以字节为单位，发送到接收方。
- Content-Language 实体报头描述了客户端或者服务端能够接受的语言。
- Content-Encoding 这又是一个比较麻烦的属性，这个实体报头用来压缩媒体类型。Content-Encoding 指示对实体应用了何种编码。

  常见的内容编码有这几种： **gzip、compress、deflate、identity** ， 这个属性可以应用在请求报文和响应报文中

```
1   Accept-Encoding: gzip, deflate //请求头
2   Content-Encoding: gzip  //响应头
```

下面是一些实体标头字段

首部字段名	说明
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

请求标头

Host

Host 请求头指明了服务器的域名（对于虚拟主机来说），以及（可选的）服务器监听的 TCP 端口号。如果没有给定端口号，会自动使用被请求服务的默认端口（比如请求一个 HTTP 的 URL 会自动使用 80 作为端口）。

```
1   Host: developer.mozilla.org
```

上面的 `Accpet` 、 `Accept-Language` 、 `Accept-Encoding` 都是属于内容协商的请求标头。

Referer

HTTP Referer 属性是请求标头的一部分，当浏览器向 web 服务器发送请求的时候，一般会带上 Referer，告诉服务器该网页是从哪个页面链接过来的，服务器因此可以获得一些信息用于处理。

```
1   Referer: https://developer.mozilla.org/testpage.html
```

If-Modified-Since

If-Modified-Since 通常会与 If-None-Match 搭配使用，If-Modified-Since 用于确认代理或客户端拥有的本地资源的有效性。获取资源的更新日期时间，可通过确认首部字段 `Last-Modified` 来确定。

大白话说就是如果在 `Last-Modified` 之后更新了服务器资源，那么服务器会响应 200，如果在 `Last-Modified` 之后没有更新过资源，则返回 304。

```
1   If-Modified-Since: Mon, 18 Jul 2016 02:36:04 GMT
```

If-None-Match

If-None-Match HTTP 请求标头使请求成为条件请求。 对于 GET 和 HEAD 方法，仅当服务器没有与给定资源匹配的 `ETag` 时，服务器才会以 200 状态发送回请求的资源。 对于其他方法，仅当最终现有资源的 `ETag` 与列出的任何值都不匹配时，才会处理请求。

```
1   If-None-Match: "c561c68d0ba92bbeb8b0fff2a9199f722e3a621a"
```

Accept

接受请求 HTTP 标头会通告客户端其能够理解的 MIME 类型

Accept-Charset

accept-charset 属性规定服务器处理表单数据所接受的字符集。

常用的字符集有： UTF-8 - Unicode 字符编码 ； ISO-8859-1 - 拉丁字母表的字符编码

Accept-Language

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。

请求标头我们大概就介绍这几种，后面会有一篇文章详细深挖所有的响应头的，下面是一个响应头的汇总，基于 HTTP 1.1

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）
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
TE	传输编码的优先级
User-Agent	HTTP 客户端程序的信息

响应标头

Access-Control-Allow-Origin

一个返回的 HTTP 标头可能会具有 Access-Control-Allow-Origin ， `Access-Control-Allow-Origin` 指定一个来源，它告诉浏览器允许该来源进行资源访问。

Keep-Alive

Keep-Alive 表示的是 Connection 非持续连接的存活时间，可以进行指定。

Server

服务器标头包含有关原始服务器用来处理请求的软件的信息。

应该避免使用过于冗长和详细的 Server 值，因为它们可能会泄露内部实施细节，这可能会使攻击者容易地发现并利用已知的安全漏洞。例如下面这种写法

```
1   Server: Apache/2.4.1 (Unix)
```

Set-Cookie

Set-Cookie 用于服务器向客户端发送 sessionID。

Transfer-Encoding

首部字段 Transfer-Encoding 规定了传输报文主体时采用的编码方式。

HTTP /1.1 的传输编码方式仅对分块传输编码有效。

X-Frame-Options

HTTP 首部字段是可以自行扩展的。所以在 Web 服务器和浏览器的应用上，会出现各种非标准的首部字段。

首部字段 `X-Frame-Options` 属于 HTTP 响应首部，用于控制网站内容在其他 Web 网站的 Frame 标签内的显示问题。其主要目的是为了防止点击劫持（clickjacking）攻击。

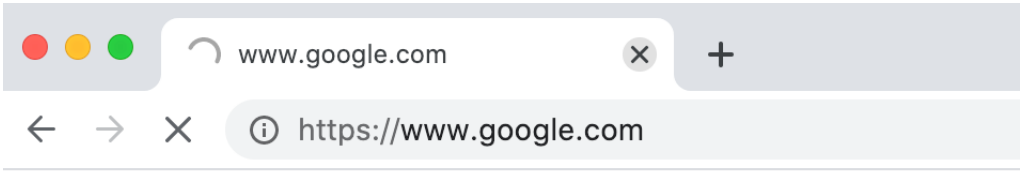
下面是一个响应头的汇总，基于 HTTP 1.1

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

地址栏输入 URL 发生了什么

这道题也是一道经常会考的面试题。那么下面我们就来探讨一下从你输入 URL 后到响应，都经历了哪些过程。

- 首先，你需要在浏览器中的 URL 地址上，输入你想访问的地址，如下

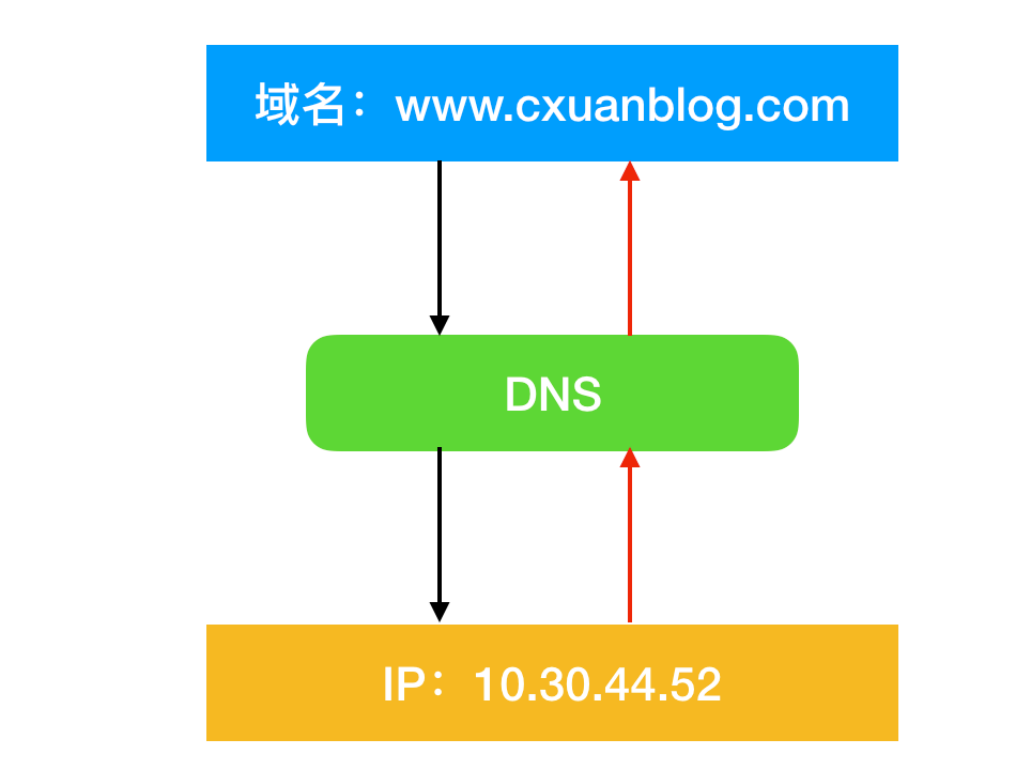


你应该访问不到的，对不对~

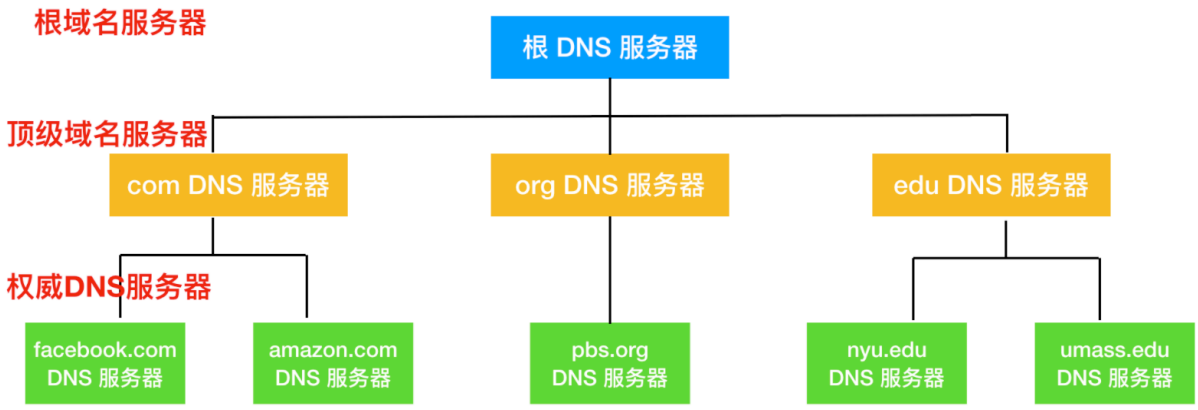
- 然后，浏览器会根据你输入的 URL 地址，去查找域名是否被本地 DNS 缓存，不同浏览器对 DNS 的设置不同，如果浏览器缓存了你想访问的 URL 地址，那就直接返回 ip。如果没有缓存你的 URL 地址，浏览器就会发起系统调用来查询本机 `hosts` 文件是否有配置 ip 地址，如果找到，直接返回。如果找不到，就向网络中发起一个 DNS 查询。

> 首先来看一下 DNS 是啥，互联网中识别主机的方式有两种，通过 `主机名` 和 `IP 地址` 。我们人喜欢用名字的方式进行记忆，但是通信链路中的路由却喜欢定长、有层次结构的 IP 地址。所以就需要一种能够把主机名到 IP 地址的转换服务，这种服务就是由 DNS 提供的。DNS 的全称是 `Domain Name System` 域名系统。DNS 是一种由分层的 DNS 服务器实现的分布式数据库。

> DNS 运行在 UDP 上，使用 53 端口。



DNS 是一种分层数据库，它的主要层次结构如下



一般域名服务器的层次结构主要是以上三种，除此之外，还有另一类重要的 DNS 服务器，它是 `本地 DNS 服务器(local DNS server)` 。严格来说，本地 DNS 服务器并不属于上述层次结构，但是本地 DNS 服务器又是至关重要的。每个 `ISP(Internet Service Provider)` 比如居民区的 ISP 或者一个机构的 ISP 都有一台本地 DNS 服务器。当主机和 ISP 进行连接时，该 ISP 会提供一台主机的 IP 地址，该主机会具有一台或多台其本地 DNS 服务器的 IP地址。通过访问网络连接，用户能够容易的确定 DNS 服务器的 IP地址。当主机发出 DNS 请求后，该请求被发往本地 DNS 服务器，它起着代理的作用，并将该请求转发到 DNS 服务器层次系统中。

首先，查询请求会先找到本地 DNS 服务器来查询是否包含 IP 地址，如果本地 DNS 无法查询到目标 IP 地址，就会向根域名服务器发起一个 DNS 查询。

> 注意：DNS 涉及两种查询方式：一种是 `递归查询(Recursive query)` ，一种是 `迭代查询(Iteration query)` 。《计算机网络：自顶向下方法》竟然没有给出递归查询和迭代查询的区别，找了一下网上的资料大概明白了下。

> 如果根域名服务器无法告知本地 DNS 服务器下一步需要访问哪个顶级域名服务器，就会使用递归查询；
>
> 如果根域名服务器能够告知 DNS 服务器下一步需要访问的顶级域名服务器，就会使用迭代查询。

在由根域名服务器 -> 顶级域名服务器 -> 权威 DNS 服务器后，由权威服务器告诉本地服务器目标 IP 地址，再有本地 DNS 服务器告诉用户需要访问的 IP 地址。

- 第三步，浏览器需要和目标服务器建立 TCP 连接，需要经过三次握手的过程，具体的握手过程请参考上面的回答。
- 在建立连接后，浏览器会向目标服务器发起 `HTTP-GET` 请求，包括其中的 URL，HTTP 1.1 后默认使用长连接，只需要一次握手即可多次传输数据。
- 如果目标服务器只是一个简单的页面，就会直接返回。但是对于某些大型网站的站点，往往不会直接返回主机名所在的页面，而会直接重定向。返回的状态码就不是 200 ，而是 301,302 以 3 开头的重定向码，浏览器在获取了重定向响应后，在响应报文中 Location 项找到重定向地址，浏览器重新第一步访问即可。
- 然后浏览器重新发送请求，携带新的 URL，返回状态码 200 OK，表示服务器可以响应请求，返回报文。

HTTPS 的工作原理

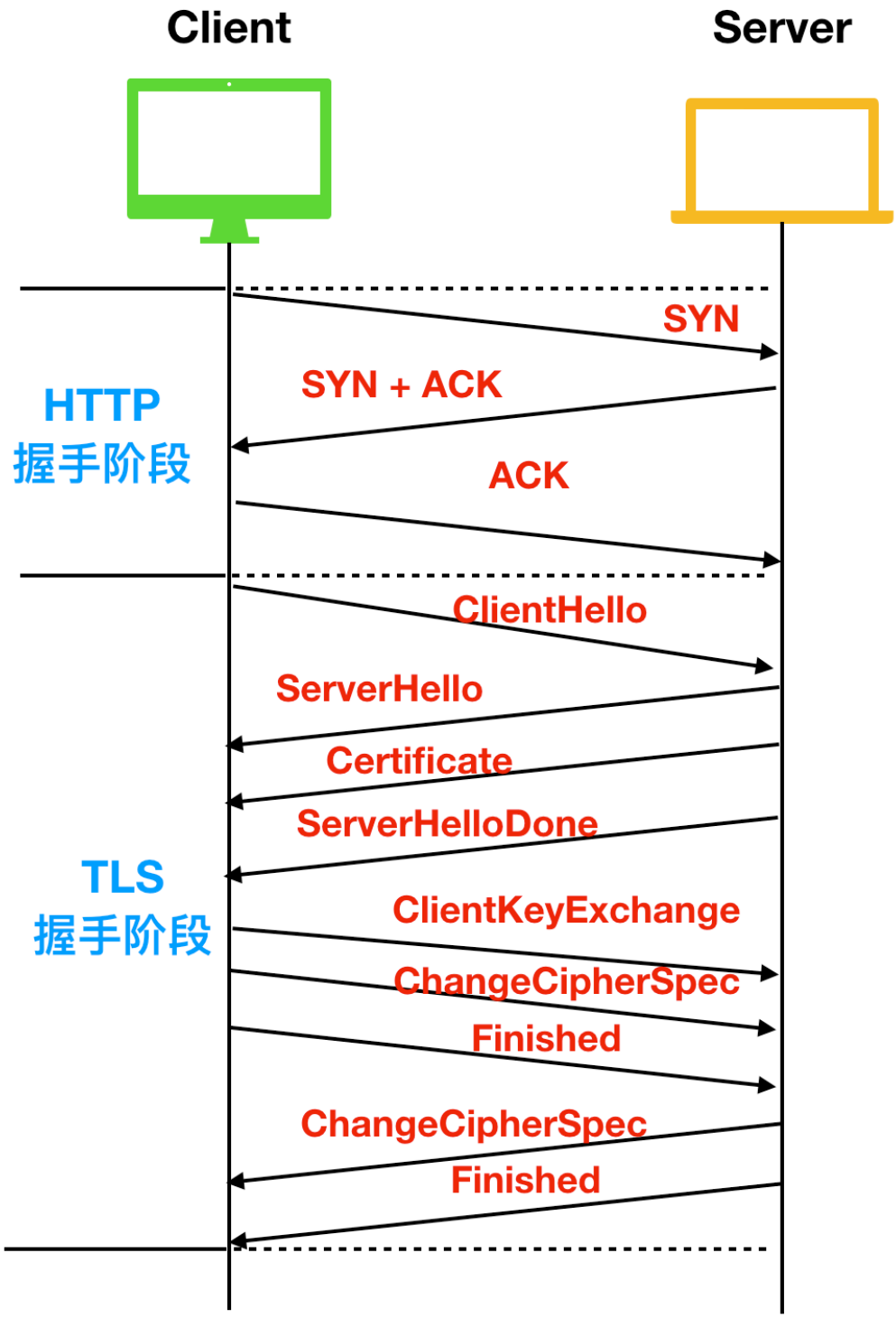
我们上面描述了一下 HTTP 的工作原理，下面来讲述一下 HTTPS 的工作原理。因为我们知道 HTTPS 不是一种新出现的协议，而是

所以，我们探讨 HTTPS 的握手过程，其实就是 SSL/TLS 的握手过程。

TLS 旨在为 Internet 提供通信安全的加密协议。TLS 握手是启动和使用 TLS 加密的通信会话的过程。在 TLS 握手期间，Internet 中的通信双方会彼此交换信息，验证密码套件，交换会话密钥。

每当用户通过 HTTPS 导航到具体的网站并发送请求时，就会进行 TLS 握手。除此之外，每当其他任何通信使用HTTPS（包括 API 调用和在 HTTPS 上查询 DNS）时，也会发生 TLS 握手。

TLS 具体的握手过程会根据所使用的 `密钥交换算法的类型` 和双方支持的 `密码套件` 而不同。 我们以 `RSA 非对称加密` 来讨论这个过程。整个 TLS 通信流程图如下



- 在进行通信前，首先会进行 HTTP 的三次握手，握手完成后，再进行 TLS 的握手过程
- ClientHello：客户端通过向服务器发送 `hello` 消息来发起握手过程。这个消息中会夹带着客户端支持的 `TLS 版本号(TLS1.0 、TLS1.2、TLS1.3)` 、客户端支持的密码套件、以及一串 `客户端随机数` 。
- ServerHello：在客户端发送 hello 消息后，服务器会发送一条消息，这条消息包含了服务器的 SSL 证书、服务器选择的密码套件和服务器生成的随机数。
- 认证(Authentication)：客户端的证书颁发机构会认证 SSL 证书，然后发送 `Certificate` 报文，报文中包含公开密钥证书。最后服务器发送 `ServerHelloDone` 作为 `hello` 请求的响应。第一部分握手阶段结束。
- `加密阶段` ：在第一个阶段握手完成后，客户端会发送 `ClientKeyExchange` 作为响应，这个响应中包含了一种称为 `The premaster secret` 的密钥字符串，这个字符串就是使用上面公开密

钥证书进行加密的字符串。随后客户端会发送 `ChangeCipherSpec`，告诉服务端使用私钥解密这个 `premaster secret` 的字符串，然后客户端发送 `Finished` 告诉服务端自己发送完成了。

> Session key 其实就是用公钥证书加密的公钥。

- `实现了安全的非对称加密`：然后，服务器再发送 `ChangeCipherSpec` 和 `Finished` 告诉客户端解密完成，至此实现了 RSA 的非对称加密。

欢迎关注作者的微信公众号 Java建设者 和 程序员cxuan