# HTTP

## 简介

HTTP（Hyper Text Transformer Protocol，超文本传输协议）是一种通信协议，它允许将超文本标记语言（HTML）文档从Web服务器传送到客户端的浏览器。

它是一个应用层协议，**承载于TCP协议之上**。

由请求和响应构成，是一个标准的客户端服务器模型（典型的问答结构模型）。



注：客户端可以是浏览器等。

作用：规定了应用进程间通信的准则

特点：

1. 传输效率高

无连接：交换HTTP报文前，不需要建立HTTP连接

无状态：数据传输过程中，不保存任何历史和状态信息

传输格式简单：请求时，只需要传输请求方法和路径

1. 传输可靠性高

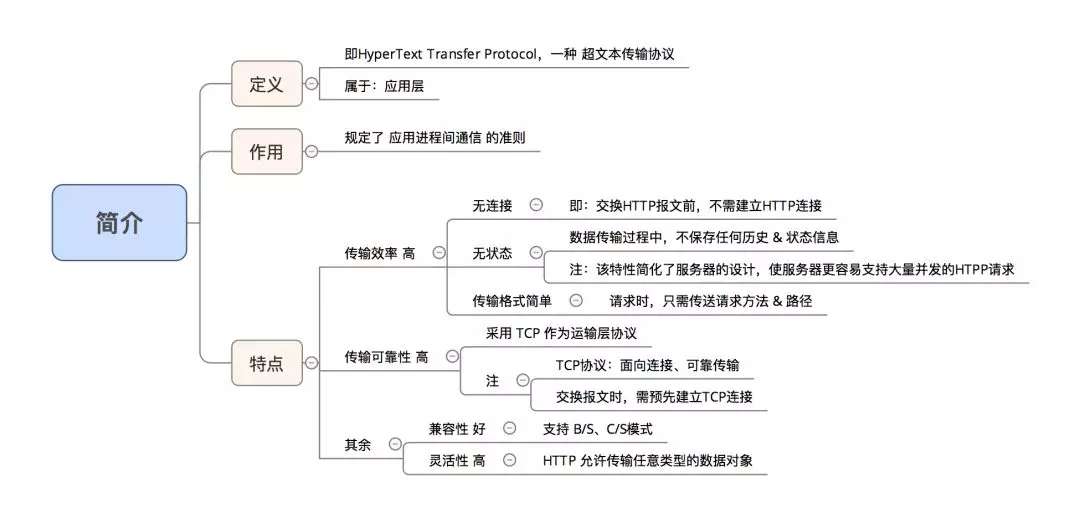
采用TCP作为传输层协议

1. 兼容性好

支持B/S、C/S模式

1. 灵活性高

HTTP允许传输任意类型的数据对象



## 版本



作为Web文档传输协议的HTTP，版本更新十分缓慢，目前只更新了三个版本：

### HTTP/0.9

GET方法拉取数据。

### HTTP1.0

HTTP/1.0：HTTP/1.0是第一个在通讯中指定版本号的HTTP协议版本，至今仍被广泛采用，特别是在代理服务器中。**默认采用短连接**。

在HTTP 1.0中, 没有官方的keepalive的操作。通常是在现有协议上添加一个指数。如果浏览器支持keep-alive，它会在请求的包头中添加：Connection: Keep-Alive

然后当服务器收到请求，作出回应的时候，它也添加一个头在响应中：Connection: Keep-Alive

这样做，连接就不会中断，而是保持连接。当客户端发送另一个请求时，它会使用同一个连接。 这一直继续到客户端或服务器端认为会话已经结束，其中一方中断连接。

HTTP1.0规定浏览器和服务器保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器处理完成后立即断开TCP连接（无连接），服务器不跟踪每个客户端也不记录过去的请求（无状态）。

这种无状态性可以借助cookie/session机制来做身份认证和状态记录。

而下面两个问题就比较麻烦：

首先，无连接的特性导致最大的性能缺陷就是无法复用连接。每次发送请求的时候，都需要进行一次TCP的连接，而TCP的连接释放过程又是比较费事的。这种无连接的特性会使得网络的利用率非常低。

其次就是队头阻塞（head of line blocking）。由于HTTP1.0规定下一个请求必须在前一个请求响应到达之前才能发送。假设前一个请求响应一直不到达，那么下一个请求就不发送，同样的后面的请求也给阻塞了。

为了解决这些问题，HTTP1.1出现了。

### HTTP/1.1

HTTP/1.1：HTTP/1.1是当前版本，**持久连接/长连接被默认采用**（默认模式使用带流水线的持久连接），并能很好地配合代理服务器工作，还支持以管道方式同时发送多个请求，以便降低线路负载，提高传输速度。

注：HTTP1.0和1.1在之后很长的一段时间内会一直并存，这是由于网络基础设施更新缓慢所决定的。

HTTP持久连接不使用独立的keepalive信息，而是仅仅允许多个请求使用单个连接。然而，Apache 2.0 httpd 的默认连接过期时间是仅仅15秒，对于 Apache 2.2 只有5秒。 短的过期时间的优点是能够快速的传输多个web页组件，而不会绑定多个服务器进程或线程太长时间。

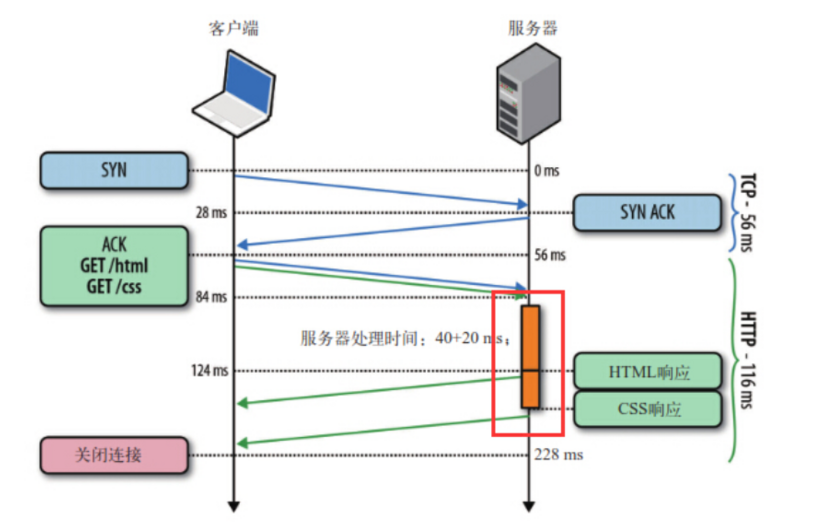
对于HTTP1.1，不仅继承了HTTP1.0简单的特点，还克服了诸多HTTP1.0性能上的问题。

首先是长连接，HTTP1.1增加了一个Connection字段，通过设置Keep-Alive可以保持HTTP连接不断开，避免了每次客户端与服务器请求都要重复建立释放建立TCP连接，提高了网络的利用率。如果客户端想关闭HTTP连接，可以在请求头中携带Connection: false来告知服务器关闭请求。

其次，是HTTP1.1支持请求管道化（pipelining）。基于HTTP1.1的长连接，使得请求管线化成为可能。管线化使得请求能够“并行”传输。举个例子来说，假如响应的主体是一个html页面，页面中包含了很多img，这个时候keep-alive就起了很大的作用，能够进行“并行”发送多个请求。

注意：这里的“并行”并不是真正意义上的并行传输，需要注意的是，服务器必须按照客户端请求的先后顺序依次回送相应的结果，以保证客户端能够区分出每次请求的响应内容。

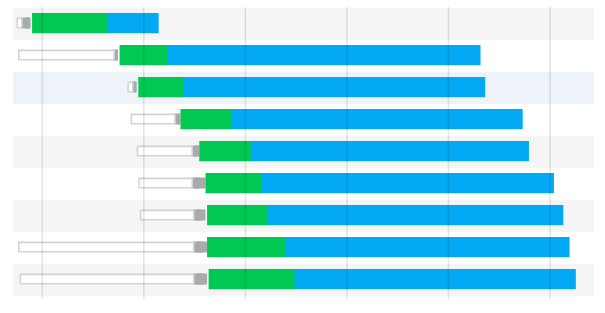
也就是说，HTTP管道化可以让我们把先进先出队列从客户端（请求队列）迁移到服务端（响应队列）。



如图所示，客户端同时发了两个请求分别来获取html和css，假如说服务器的css资源先准备就绪，服务器也会先发送html再发送css。

换句话来说，只有等到html响应的资源完全传输完毕后，css响应的资源才能开始传输。也就是说，不允许同时存在两个并行的响应。

可见，HTTP1.1还是无法解决队头阻塞（head of line blocking）的问题。同时“管道化”技术存在各种各样的问题，所以很多浏览器要么根本不支持它，要么就直接默认关闭，并且开启的条件很苛刻，而且实际上好像并没有什么用处。



如图所示，绿色部分代表请求发起到服务器响应的一个等待时间，而蓝色部分表示资源的下载时间。按照理论来说，HTTP响应理应当是前一个响应的资源下载完了，下一个响应的资源才能开始下载。而这里却出现了响应资源下载并行的情况。这又是为什么呢？

虽然HTTP1.1支持管道化，但是服务器也必须进行逐个响应的送回，这个是很大的一个缺陷。实际上，现阶段的浏览器厂商采取了另外一种做法，它允许我们打开多个TCP的会话。也就是说，上图我们看到的并行，其实是不同的TCP连接上的HTTP请求和响应。这也就是我们所熟悉的浏览器对同域下并行加载6~8个资源的限制。而这，才是真正的并行！

此外，HTTP1.1还加入了缓存处理（强缓存和协商缓存）新的字段如cache-control，支持断点传输，以及增加了Host字段（使得一个服务器能够用来创建多个Web站点）。

**HTTP1.0的缺点：**

1. 明文传输（明文即表示没有压缩，传输效率低）
2. 传输问题（没有采用乱序，乱序传输不用等待前面的块效率较高）
3. 请求头太长
4. Server端无法主动push（在HTTP2.0之前该问题已通过websocket解决）

**HHTP2.0针对上述问题，提出解决方案：**

1. 二进制传输（对原来的明文进行压缩）
2. 单连接+帧（HTTP1.0是多个连接，一个个刷新）
3. 头部压缩
4. Server端可以主动push

#### Keep-alive

##### 优势

较少的CPU和内存的使用（由于同时打开的连接的减少了）

允许请求和应答的HTTP管线化

减少了后续请求的延迟（无需再进行握手）

报告错误无需关闭

##### 劣势

对于现在的广泛普及的宽带连接来说，Keep-Alive也许并不像以前一样有用。web服务器会保持连接若干秒(Apache中默认15秒)，这与提高的性能相比也许会影响性能。对于单个文件被不断请求的服务(例如图片存放网站)，Keep-Alive可能会极大的影响性能，因为它在文件被请求之后还保持了不必要的连接很长时间。

根据RFC2616，用户客户端与任何服务器和代理服务器之间不应该维持超过2个链接。 代理服务器应该最多使用2×N个持久连接到其他服务器或代理服务器，其中N是同时活跃的用户数。这个指引旨在提高HTTP响应时间并避免阻塞。但由于，TCP连接减少了，对于静态资源（图片、JavaScript、CSS）较多的网站，性能反而可能会下降。

#### 动静分离

为了规避上面说的对图片等静态资源的影响，大多数商业网站会启用独立的静态资源域名。从而保证主站的动态资源请求和静态资源的请求不会互相挤占连接。动静分离同时还会有一个额外的好处：对于静态资源的请求，HTTP请求头里的Cookie等信息是没有用处的， 反而占用了宝贵的上行网络资源。用独立的域名存放静态资源后，请求静态资源域名就不会默认带上主站域的Cookie，从而解决了这个问题。如下表：



### HTTP2.0

HTTP/2.0：HTTP/2.0在HTTP 1.x的基础上，大幅度的提高了web性能，减少了网络延迟。

HTTP2.0的几个特征：

#### 多路复用

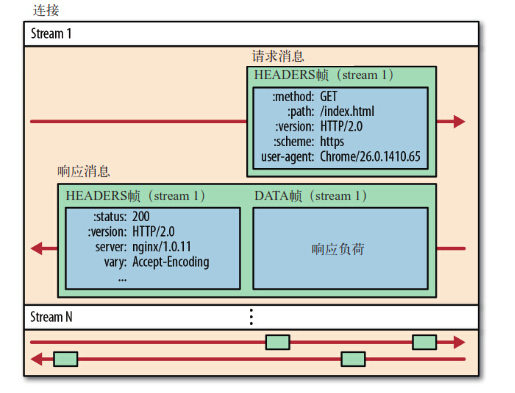
**多路复用 (Multiplexing)**

多路复用允许同时通过单一的HTTP/2连接发起多重的请求-响应消息。在 HTTP/1.1协议中浏览器客户端在同一时间，针对同一域名下的请求有一定数量限制。超过限制数目的请求会被阻塞。这也是为何一些站点会有多个静态资源CDN域名的原因之一，拿Twitter为例，http://twimg.com，目的就是变相的解决浏览器针对同一域名的请求限制阻塞问题。而HTTP/2的多路复用(Multiplexing) 则允许同时通过单一的HTTP/2连接发起多重的请求-响应消息。因此HTTP/2可以很容易的去实现多流并行而不用依赖建立多个TCP连接，HTTP/2把HTTP协议通信的基本单位缩小为一个一个的帧，这些帧对应着逻辑流中的消息。并行地在同一个TCP连接上双向交换消息。

流（stream）：已建立连接上的双向字节流。

消息：与逻辑消息对应的完整的一系列数据帧。

帧（frame）：HTTP2.0通信的最小单位，每个帧包含帧头部，至少也会标识出当前帧所属的流（stream id）。



从图中可见，所有的HTTP2.0通信都在一个TCP连接上完成，这个连接可以承载任意数量的双向数据流。

每个数据流以消息的形式发送，而消息由一或多个帧组成。这些帧可以乱序发送，然后再根据每个帧头部的流标识符（stream id）重新组装。

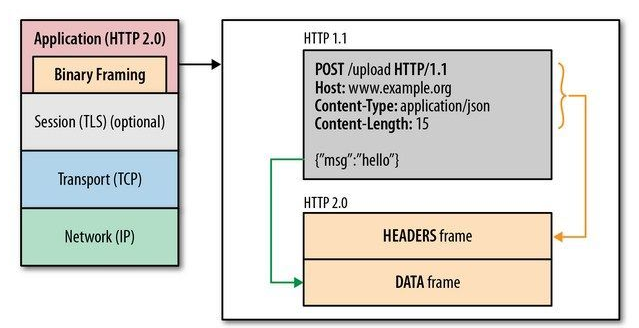
举个例子，每个请求是一个数据流，数据流以消息的方式发送，而消息又分为多个帧，帧头部记录着stream id用来标识所属的数据流，不同属的帧可以在连接中随机混杂在一起。接收方可以根据stream id将帧再归属到各自不同的请求当中去。

另外，多路复用（连接共享）可能会导致关键请求被阻塞。HTTP2.0里每个数据流都可以设置优先级和依赖，优先级高的数据流会被服务器优先处理和返回给客户端，数据流还可以依赖其他的子数据流。

可见，HTTP2.0实现了真正的并行传输，它能够在一个TCP上进行任意数量HTTP请求。而这个强大的功能则是基于“二进制分帧”的特性。

#### 二进制分帧

**二进制分帧**



HTTP/2在**应用层(HTTP/2)和传输层(TCP or UDP)**之间增加一个二进制分帧层。在不改动 HTTP/1.x 的语义、方法、状态码、URI以及首部字段的情况下, 解决了HTTP1.1 的性能限制，改进传输性能，实现低延迟和高吞吐量。在二进制分帧层中， HTTP/2会将所有传输的信息分割为更小的消息和帧（frame）,并对它们采用二进制格式的编码，其中HTTP1.x 的首部信息会被封装到HEADER frame，而相应的Request Body 则封装到 DATA frame 里面。

HTTP/2 通信都在一个连接上完成，这个连接可以承载任意数量的双向数据流。在过去， HTTP 性能优化的关键并不在于高带宽，而是低延迟。TCP 连接会随着时间进行自我调谐，起初会限制连接的最大速度，如果数据成功传输，会随着时间的推移提高传输的速度。这种调谐则被称为 TCP 慢启动。由于这种原因，让原本就具有突发性和短时性的 HTTP 连接变的十分低效。HTTP/2 通过让所有数据流共用同一个连接，可以更有效地使用 TCP 连接，让高带宽也能真正的服务于 HTTP 的性能提升。

这种单连接多资源的方式，减少服务端的链接压力,内存占用更少,连接吞吐量更大；而且由于 TCP 连接的减少而使网络拥塞状况得以改善，同时慢启动时间的减少,使拥塞和丢包恢复速度更快。

#### 首部压缩

**首部压缩（Header Compression）**

HTTP/1.1并不支持 HTTP 首部压缩，为此SPDY和HTTP/2应运而生，SPDY使用的是通用的DEFLATE算法，而 HTTP/2则使用了专门为首部压缩而设计的HPACK算法。

在HTTP1.x中，头部元数据都是以纯文本的形式发送的，通常会给每个请求增加500~800字节的负荷。

比如说cookie，默认情况下，浏览器会在每次请求的时候，把cookie附在header上面发送给服务器。（由于cookie比较大且每次都重复发送，一般不存储信息，只是用来做状态记录和身份认证）

HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。高效的压缩算法可以很大的压缩header，减少发送包的数量从而降低延迟。

#### 服务端推送

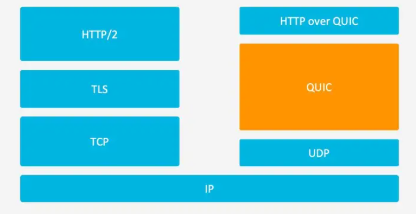
**服务端推送（Server Push）**

服务端推送是一种在客户端请求之前发送数据的机制。在 HTTP/2中，服务器可以对客户端的一个请求发送多个响应。Server Push让 HTTP1.x 时代使用内嵌资源的优化手段变得没有意义；如果一个请求是由你的主页发起的，服务器很可能会响应主页内容、logo 以及样式表，因为它知道客户端会用到这些东西。这相当于在一个 HTML 文档内集合了所有的资源，不过与之相比，服务器推送还有一个很大的优势：可以缓存！也让在遵循同源的情况下，不同页面之间可以共享缓存资源成为可能。

### HTTP3.0

1999年设计的HTTP/1.1已经不能满足需求，所以Google在2009年设计了基于TCP的 SPDY，后来SPDY的开发组推动SPDY成为正式标准，不过最终没能通过。不过SPDY的开发组全程参与了HTTP/2的制定过程，参考了SPDY的很多设计，所以我们一般认为SPDY就是HTTP/2的前身。无论SPDY还是HTTP/2，都是基于TCP的，TCP与UDP相比效率上存在天然的劣势，所以2013年Google开发了基于UDP的名为QUIC的传输层协议，QUIC全称Quick UDP Internet Connections，希望它能替代TCP，使得网页传输更加高效。后经提议，互联网工程任务组正式将基于QUIC协议的 HTTP（HTTP over QUIC）重命名为HTTP/3。

QUIC（Quick UDP Internet Connections，快速UDP网络连接） 基于UDP，正是看中了UDP的速度与效率。同时QUIC也整合了TCP、TLS和HTTP/2的优点，并加以优化。



QUIC是用来替代TCP、SSL/TLS的传输层协议，在传输层之上还有应用层，我们熟知的应用层协议有HTTP、FTP、IMAP等，这些协议理论上都可以运行在QUIC之上，其中运行在QUIC之上的HTTP协议被称为HTTP/3，这就是HTTP over QUIC，即HTTP/3的含义。

[https://mp.weixin.qq.com/s?\_\_biz=MzI1OTY2MzMxOQ==&mid=2247490673&idx=1&sn=b4ea1c2a47d2b76e9fba1e582b430c0e&chksm=ea7428c9dd03a1df2b119118b2f5bc875e8913d3b9d924188a7fe8d8e2bfb1f11e84ff780701&mpshare=1&scene=24&srcid=0918WqWVyoBTbeKiv0qsVzjT&sharer\_sharetime=1600388103192&sharer\_shareid=33f795d236f19ac7c128b2e279563f84#rd](https://mp.weixin.qq.com/s?__biz=MzI1OTY2MzMxOQ==&mid=2247490673&idx=1&sn=b4ea1c2a47d2b76e9fba1e582b430c0e&chksm=ea7428c9dd03a1df2b119118b2f5bc875e8913d3b9d924188a7fe8d8e2bfb1f11e84ff780701&mpshare=1&scene=24&srcid=0918WqWVyoBTbeKiv0qsVzjT&sharer_sharetime=1600388103192&sharer_shareid=33f795d236f19ac7c128b2e279563f84" \l "rd)

### 总结

**HTTP1.0：**

无状态、无连接

**HTTP1.1：**

持久连接

请求管道化

增加缓存处理（新的字段如cache-control）

增加Host字段、支持断点传输等

**HTTP2.0：**

二进制分帧

多路复用（或连接共享）

头部压缩

服务器推送

## 应用

HTTP是整个Web基础，许多应用都离不开对HTTP协议的认识：

WebService=HTTP+XML

Reset=HTTP+JSON

API的实现：HTTP+XML/JSON

采集、小偷站

QQ、迅雷等桌面应用软件

# HTTPS

## 背景

我们为什么需要HTTPS？主要有如下三个原因：

1、保护隐私（Privacy）：所有信息都是加密传播，第三方无法窃听数据。**如果使用HTTP明文传输数据的话，很可能被第三方劫持数据**，那么所输入的密码或者其他个人资料都被暴露在他人面前，后果可想而知。

2、数据完整性（Integraty）：一旦第三方篡改了数据，接收方会知道数据经过了篡改，这样便保证了数据在传输过程中不被篡改——数据的完整性。

3、身份认证（Identification）：第三方不可能冒充身份参与通信，因为服务器配备了由证书颁发机构（Certificate Authority，简称CA）颁发的安全证书，可以证实服务器的身份信息，防止第三方冒充身份。

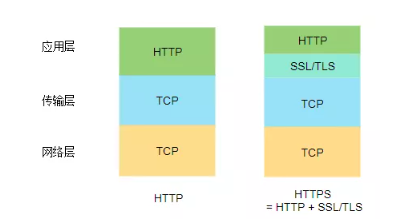
也有少数情况下，通信需要客户端提供证书，例如银行系统，需要用户在登录的时候，插入银行提供给用户的USB，就是需要客户端提供证书，用来验证客户的身份信息。

## 概述

HTTP协议传输的数据都是未加密的，也就是**明文**的，因此使用HTTP协议传输隐私信息非常不安全。为了保证这些隐私数据能加密传输，于是网景公司设计了**SSL（Secure Sockets Layer）**协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。现在的HTTPS都是用的**TLS协议**，但是由于SSL出现的时间比较早，并且依旧被现在浏览器所支持，因此SSL依然是HTTPS的代名词。

**HTTPS在内容传输的加密上使用的是对称加密，非对称加密只作用在证书验证阶段**。

IETF将SSL作了标准化，重新命名为TLS（Transport Layer Security）。在1999年，TLS 1.0诞生了（其实也就是SSL 3.1）。



HTTPS（HyperText Transfer Protocol Secure）是建立在SSL/TLS协议之上，信息通信通过SSL/TLS进行加密，最后一个S就是Secure的缩写，代表安全的意思，HTTPS = HTTP+SSL/TLS。

## SSL/TLS

### 加密算法

**加密算法有两种：**

对称加密

非对称加密

#### 对称加密

**对称加密：**通信双方使用相同的密钥进行加密。特点是加密速度快，但是缺点是需要保护好密钥，如果密钥泄露的话，那么加密就会被别人破解。常见的对称加密有AES，DES算法。

如果对称加密客户端都是用同样的key和加密算法，那么是不安全的。

#### 非对称加密

**非对称加密：**它需要生成两个密钥：公钥（Public Key）和私钥（Private Key）。公钥加密私钥解密，私钥加密公钥解密。

公钥顾名思义是公开的，任何人都可以获得，而私钥是私人（或服务端）保管的。

非对称加密的基本过程：

1. 客户端请求服务端获取服务端的公钥；
2. 服务端返回公钥给客户端；
3. 客户端利用获取的公钥对数据加密，传输给服务端；
4. 服务端通过私钥解密数据；
5. 服务端向客户端发送数据（不能使用公钥加密，因为客户端没有私钥解密，私钥只有服务端才有，如果使用私钥加密，那么黑客也可以获取到公钥，非对称加密也是不安全的）。

我们提交代码到Github的时候，就可以使用SSH key：在本地生成私钥和公钥，私钥放在本地.ssh目录中，公钥放在Github网站上。

这样每次提交代码，不用麻烦的输入用户名和密码了，Github会根据网站上存储的公钥来识别我们的身份。公钥负责加密，私钥负责解密；或者，私钥负责加密，公钥负责解密。

这种加密算法安全性更高，但是计算量相比对称加密大很多，加密和解密都很慢。

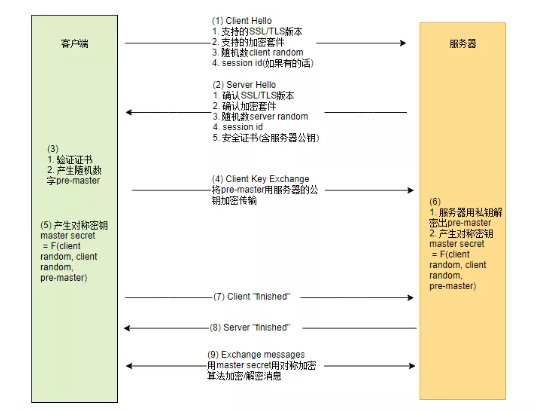
常见的非对称算法有RSA。

综上，对称加密和非对称加密都存在安全问题，对于对称加密，其缺点在在于key只有一个，如果每个客户端都有一个key，这样可以实现安全函数。非对称加密在客户端向服务端发送数据时是安全的，在服务端向客户端发送数据的时候是非安全的。可以利用非对称加密算法在客户端和服务端协商一个临时制定的key，然后使用这个key值进行对称加密数据交互（这个key是当前客户端和服务端协商的值，其他客户端还会协商另外的值，每个客户端就可以拿到不同的key）。

### 原理

SSL/TLS是利用了对称加密和非对称加密的特点。

先来看下整个SSL/TLS的握手过程：



1、当TCP建立连接之后，TLS握手的第一步由客户端发起，发送ClientHello的消息到服务器。

ClientHello 消息包含：

客户端支持的SSL/TLS版本

客户端支持的加密套件（Cipher Suites）

会话ID session id（如果有的值的话，服务器端会复用对应的握手信息，避免短时间内重复握手）

随机数client-random

2、然后服务器端在收到这个ClientHello，从中选择服务器支持的版本和套件，发送ServerHello消息：

服务器所能支持的最高SSL/TLS版本

服务器选择的加密套件

随机数server-random

会话ID session id（用于下次复用当前握手的信息，避免短时间内重复握手。）

随后服务器发送服务器的安全证书(含公钥)。如果需要客户端也提供证书的话，还会发出客户端证书请求（Client Certificate Request），只有少数金融机构才需要客户端也提供客户端证书。

此后客户端发送Server Hello Done消息表示Hello阶段完成。

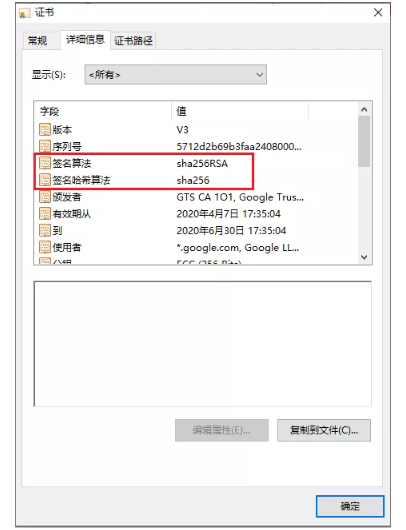
3、客户端收到ServerHello后，会对收到的证书进行验证。

我们来看一下为什么可以通过CA（Certificate Authority，证书颁发机构）签发的证书来确认网站的身份？

当我们安装操作系统或者浏览器的时候，会安装一组可信任的CA（根证书CA包括 GlobalSign、GeoTrust、Verisign等）列表。

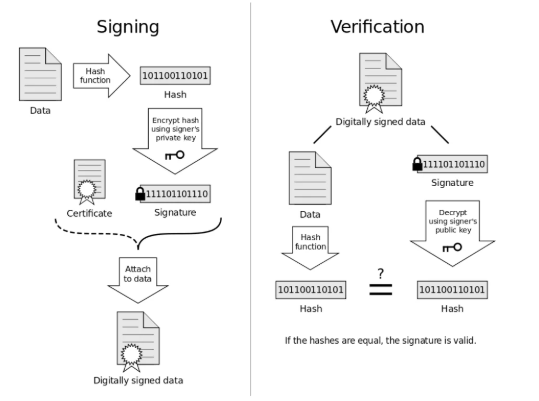
根CA如GlobalSign就在我们的可信任的CA列表里，你的浏览器或者操作系统含有 GlobalSign的公钥。

先来看一下Google的证书，当你访问Google的时候，Google会发给你它的证书。证书中包含颁发机构的签名以及服务器的公钥。



浏览器首先用哈希函数对明文信息的摘要做哈希得到一个哈希值（用到的就是证书中的签名哈希算法SHA256），然后用根CA的公钥对根证书的签名作解密得到另一个哈希值（用到的算法就是RSA非对称算法）。

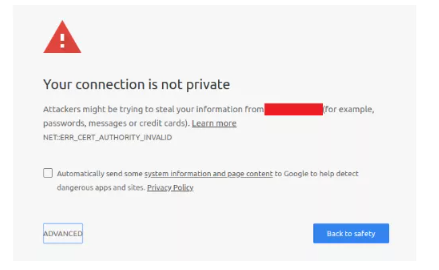
如果两个哈希值相等则说明证书没有被篡改过。当然还需校验证书中服务器名称是否合法以及验证证书是否过期。



这样就免受中间人攻击，因为假如有中间人修改了证书的内容（如将证书中的公钥替换成自己的公钥），那么将获得不同的哈希值，从而两个哈希值不匹配导致验证失败。

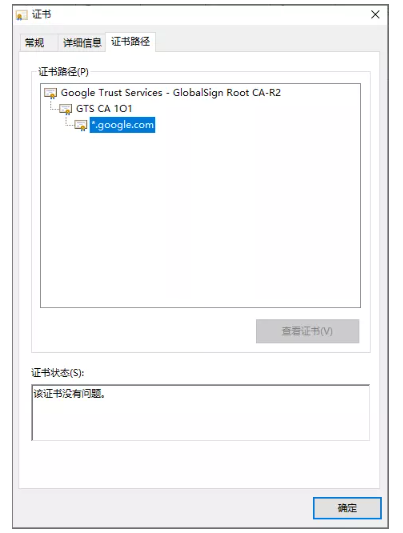
如果要绕过这个机制，中间人必须要也替换签名，使签名也相匹配。而做到这一点就需要破解到了根证书的密钥（而这是不可能的，中间人必然会失败）。

浏览器会出现以下画面，告诉你正在遭受中间人攻击，因为证书被篡改了：



如果你开发了一个系统还在测试阶段，还没有正式申请一张证书，那么你可以为服务器自签名一张证书，然后将证书导入客户端的CA信任列表中。

信任链机制如下图：



可以看到证书路径是：

GlobalSign Root CA-R2→GTS CA 1O1→\*.google.com

因为我们的浏览器信任GlobalSign Root CA，根据信任链机制，你相信了根CA颁发的证书，也要相信它签名的子CA颁发的证书，也要相信子CA签名的子CA的证书。

而我们通过一级级的校验，如果从根证书到最下层的证书都没有被篡改过，我们就相信最下层的这个服务器证书是合法的。所以在这个机制中，你就需要无条件的相信根证书的颁发机构。

如果通过验证，客户端生成一个随机数pre-master，用于密钥交换过程。

4、密钥交换过程：客户端用第三步中服务器的证书中拿到服务器的公钥，用这个公钥加密（算法是加密套件中的密钥交换算法，譬如ECDHE算法）生成密文发送给服务器。

5、客户端用server-random+client-random+pre-master一起计算出对称密钥master secret。

6、服务器收到第四步的信息之后，用服务器的私钥对密文进行解密得到密钥pre-master。

因为只有服务器有私钥，可以针对客户端发出的加密过的信息进行解密得到pre-master，这样就保证了只有服务器和客户端知道pre-master。

服务器端也可以用server-random+client-random+pre-master一起计算出对称密钥master secret。

现在客户端和服务器均有密钥 master secret了，后面就可以用它来进行加密和解密了。

**为什么不能只用一个pre-master作为之后加密的对称密钥？**

虽然只有服务器有私钥，能够解密pre-master，但仅用它作为master secret是不够安全的，这是因为要以防客户端的pre-master并不是随机数的情况。

加上另外两个随机数client-random以及server-random（而这两个随机数和时间有相关性），这样就能保证最后生成的master secret一定是随机数。

7、客户端用master secret加密了一条握手完成的消息发送给服务器。

8、服务器端也回发了一条用master secret加密的握手完成的消息。

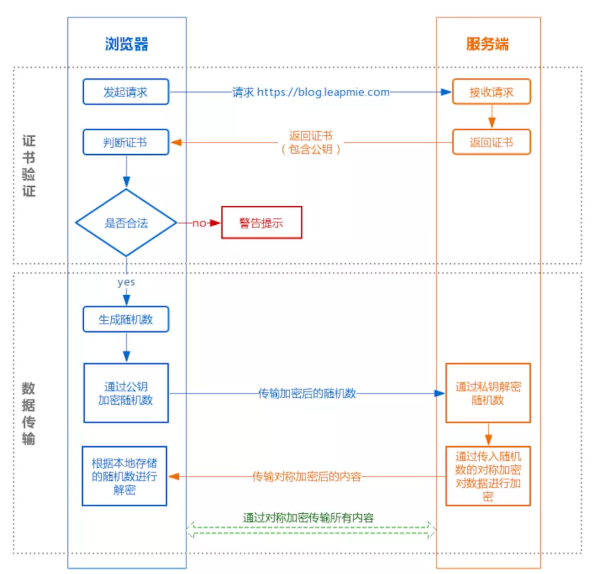
9、当两方都收到对方发送的握手消息之后，也成功解密后，就可以用master secret愉快的开始数据加密和解密了。

综上，整个握手过程主要是通过一系列步骤通过非对称加密的算法交换得到了master secret，这个步骤通常需要几百毫秒，但是就是这一顿猛操作之后使得只有服务器和客户端知道master secret。

之后的通信又利用了高效的对称算法对所有信息进行加密和解密，虽然加密和解密也需要耗时耗流量，不过信息是完全不可能被别人篡改和破解的，这一点损耗还是值得的。

## 原理

HTTPS的整体过程分为证书验证和数据传输阶段，具体的交互过程如下：



1、证书验证阶段

浏览器发起HTTPS请求

服务端返回HTTPS证书

客户端验证证书是否合法，如果不合法则提示告警

2、数据传输阶段

当证书验证合法后，在本地生成随机数

通过公钥加密随机数，并把加密后的随机数传输到服务端

服务端通过私钥对随机数进行解密

服务端通过客户端传入的随机数构造对称加密算法，对返回结果内容进行加密后传输

HTTPS在传输数据之前需要客户端（浏览器）与服务端（网站）之间进行**一次握手**，在握手过程中将**确立双方加密传输数据的密码信息**。TLS/SSL协议不仅仅是一套加密传输的协议，TLS/SSL中使用了**非对称加密，对称加密以及HASH算法**。握手过程的简单描述如下：

1、浏览器将自己支持的一套加密规则发送给网站（发送HTTP请求）。

2、网站从中选出一组加密算法与HASH算法，并将自己的身份信息以证书的形式发回给浏览器（返回HTTPS证书）。证书里面包含了网站地址，加密公钥，以及证书的颁发机构等信息。

3、获得网站证书之后浏览器要做以下工作：

a) 验证证书的合法性（颁发证书的机构是否合法，证书中包含的网站地址是否与正在访问的地址一致等），如果证书受信任，则浏览器栏里面会显示一个小锁头，否则会给出证书不受信的提示。

b) 如果证书受信任，或者是用户接受了不受信的证书，浏览器会生成一串随机数的密码，并用证书中提供的**公钥加密**随机数。

c) 使用约定好的HASH计算握手消息，并使用生成的随机数对消息进行加密，最后将之前生成的所有信息发送给网站。

4、网站接收浏览器发来的数据之后要做以下的操作：

a) 使用自己的**私钥**将信息解密取出密码，使用密码解密浏览器发来的握手消息，并验证HASH是否与浏览器发来的一致。

b) 使用密码加密一段握手消息，发送给浏览器。

5、浏览器解密并计算握手消息的HASH，如果与服务端发来的HASH一致，此时握手过程结束，之后所有的通信数据将由之前浏览器生成的随机密码并利用对称加密算法进行加密。

这里浏览器与网站互相发送加密的握手消息并验证，目的是为了保证双方都获得了一致的密码，并且可以正常的加密解密数据。其中非对称加密算法用于在握手过程中加密生成的密码，对称加密算法用于对真正传输的数据进行加密，而HASH算法用于验证数据的完整性。由于浏览器生成的密码是整个数据加密的关键，因此在传输的时候使用了非对称加密算法对其加密。非对称加密算法会生成公钥和私钥，公钥只能用于加密数据，因此可以随意传输，而网站的私钥用于对数据进行解密，所以网站都会非常小心的保管自己的私钥，防止泄漏。

TLS握手过程中如果有任何错误，都会使加密连接断开，从而阻止了隐私信息的传输。正是由于HTTPS非常的安全，攻击者无法从中找到下手的地方，于是更多的是采用了假证书的手法来欺骗客户端，从而获取明文的信息。

**默认HTTP的端口号为80，HTTPS的端口号为443。**

### 对称加密

**为什么数据传输是用对称加密？**

首先，非对称加密的加解密效率是非常低的，而http的应用场景中通常端与端之间存在大量的交互，非对称加密的效率是无法接受的；

另外，**在HTTPS的场景中只有服务端保存了私钥，一对公私钥只能实现单向的加解密，所以HTTPS中内容传输加密采取的是对称加密，而不是非对称加密**。

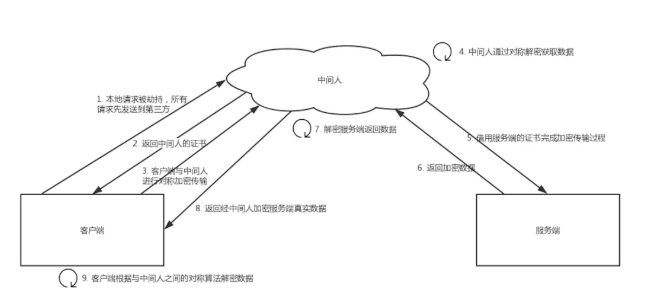
### 中间人攻击

**为什么需要 CA 认证机构颁发证书？**

HTTP协议被认为不安全是因为传输过程容易被监听者勾线监听、伪造服务器，而HTTPS协议主要解决的便是网络传输的安全性问题。

首先我们假设不存在认证机构，任何人都可以制作证书，这带来的安全风险便是经典的“中间人攻击”问题。

“中间人攻击”的具体过程如下：



过程原理：

1. 本地请求被劫持（如DNS劫持等），所有请求均发送到中间人的服务器
2. 中间人服务器返回中间人自己的证书
3. 客户端创建随机数，通过中间人证书的公钥对随机数加密后传送给中间人，然后凭随机数构造对称加密对传输内容进行加密传输
4. 中间人因为拥有客户端的随机数，可以通过对称加密算法进行内容解密
5. 中间人以客户端的请求内容再向正规网站发起请求
6. 因为中间人与服务器的通信过程是合法的，正规网站通过建立的安全通道返回加密后的数据
7. 中间人凭借与正规网站建立的对称加密算法对内容进行解密
8. 中间人通过与客户端建立的对称加密算法对正规内容返回的数据进行加密传输
9. 客户端通过与中间人建立的对称加密算法对返回结果数据进行解密

由于缺少对证书的验证，所以客户端虽然发起的是 HTTPS 请求，但客户端完全不知道自己的网络已被拦截，传输内容被中间人全部窃取。

为了解决中间人问题，引入CA。

### CA证书合法性

**浏览器是如何确保 CA 证书的合法性？**

1、证书包含什么信息？

颁发机构信息

公钥（一般都在客户端大量缓存CA的公钥，服务端存储非对称加密后的证书）

公司信息

域名

有效期

指纹

2、证书的合法性依据是什么？

首先，权威机构是要有认证的，不是随便一个机构都有资格颁发证书，不然也不叫做权威机构。另外，证书的可信性基于信任制，权威机构需要对其颁发的证书进行信用背书，只要是权威机构生成的证书，我们就认为是合法的。所以权威机构会对申请者的信息进行审核，不同等级的权威机构对审核的要求也不一样，于是证书也分为免费的、便宜的和贵的。

3、浏览器如何验证证书的合法性？

浏览器发起HTTPS请求时，服务器会返回网站的SSL证书，浏览器需要对证书做以下验证：

1. 验证域名、有效期等信息是否正确。证书上都有包含这些信息，比较容易完成验证；
2. 判断证书来源是否合法。每份签发证书都可以根据验证链查找到对应的根证书，操作系统、浏览器会在本地存储权威机构的根证书，利用本地根证书可以对对应机构签发证书完成来源验证；
3. 判断证书是否被篡改。需要与 CA 服务器进行校验；
4. 判断证书是否已吊销。通过CRL（Certificate Revocation List 证书注销列表）和 OCSP（Online Certificate Status Protocol 在线证书状态协议）实现，其中 OCSP 可用于第3步中以减少与 CA 服务器的交互，提高验证效率

以上任意一步都满足的情况下浏览器才认为证书是合法的。

**说明：**

既然证书是公开的，如果要发起中间人攻击，我在官网上下载一份证书作为我的服务器证书，那客户端肯定会认同这个证书是合法的，如何避免这种证书冒用的情况？

其实这就是非加密对称中公私钥的用处，虽然中间人可以得到证书，但私钥是无法获取的，一份公钥是不可能推算出其对应的私钥，中间人即使拿到证书也无法伪装成合法服务端，因为无法对客户端传入的加密数据进行解密。

4、只有认证机构可以生成证书吗？

如果需要浏览器不提示安全风险，那只能使用认证机构签发的证书。但浏览器通常只是提示安全风险，并不限制网站不能访问，所以从技术上谁都可以生成证书，只要有证书就可以完成网站的

HTTPS传输。例如早期的12306采用的便是手动安装私有证书的形式实现HTTPS访问。

### 本地随机数

证书验证是采用非对称加密实现，但是传输过程是采用对称加密，而其中对称加密算法中重要的随机数是由本地生成并且存储于本地的，HTTPS如何保证随机数不会被窃取？

其实HTTPS并不包含对随机数的安全保证，HTTPS保证的只是传输过程安全，而随机数存储于本地，本地的安全属于另一安全范畴，应对的措施有安装杀毒软件、反木马、浏览器升级修复漏洞等。

### HTTPS抓包

**用了HTTPS会被抓包吗？**

HTTPS的数据是加密的，常规下抓包工具代理请求后抓到的包内容是加密状态，无法直接查看。

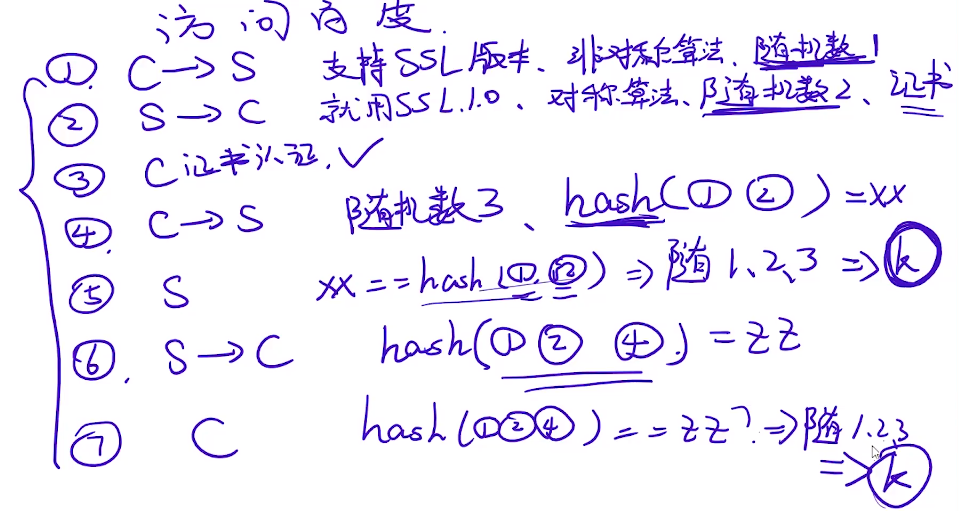
但是，正如前文所说，浏览器只会提示安全风险，如果用户授权仍然可以继续访问网站，完成请求。因此，只要客户端是我们自己的终端，我们授权的情况下，便可以组建中间人网络，而抓包工具便是作为中间人的代理。通常HTTPS抓包工具的使用方法是会生成一个证书，用户需要手动把证书安装到客户端中，然后终端发起的所有请求通过该证书完成与抓包工具的交互，然后抓包工具再转发请求到服务器，最后把服务器返回的结果在控制台输出后再返回给终端，从而完成整个请求的闭环。

既然HTTPS不能防抓包，那HTTPS有什么意义？

HTTPS可以防止用户在不知情的情况下通信链路被监听，对于主动授信的抓包操作是不提供防护的，因为这个场景用户是已经对风险知情。要防止被抓包，需要采用应用级的安全防护，例如采用私有的对称加密，同时做好移动端的防反编译加固，防止本地算法被破解。

### 协商过程

访问百度的大致协商过程：



1. client-->server发送请求（非对称加密过程），该请求包含支持SSL版本、非对称加密算法（证书验证）、随机数1
2. server-->client回送消息，包含SSL版本、对称加密算法（传输数据）、随机数2、证书
3. client证书认证
4. client-->server发送消息（对称加密过程），包含随机数3，散列算法hash(步骤1、步骤2)=XXX
5. server接收数据，验证XX=hash(步骤1、步骤2)？（server端页参与前面步骤，所以也会缓存步骤1、2的散列值），将随机数1、2、3生成key
6. server-->client发送消息，hash（步骤1、2、4）=YYY
7. client接收数据，验证YY=hash（步骤1、2、4）？，随机数1、2、3生成key

注：服务端和客户端至此完成key的协商。

## 权限认证

### Auth

Auth分为两种authorization（针对用户名密码的身份认证，错误码401）、authentication（权限不足，错误码403）。

### 方式

#### Session/Cookie

优点：比较高效

缺点：分布式情况可能出错

#### Authorization

##### Basic

BASE64

优点：

缺点：数据库认证耗时

##### Bearer

令牌Token

# 长连接/短连接

## HTTP与TCP/IP协议

**HTTP 的长连接和短连接本质上是TCP 长连接和短连接。**

HTTP属于应用层协议，在传输层使用TCP协议，在网络层使用IP协议。IP 协议主要解决网络路由和寻址问题，TCP协议主要解决如何在IP层之上可靠的传递数据包，使在网络上的另一端收到发端发出的所有包，并且顺序与发出顺序一致。TCP有可靠，面向连接的特点。

## HTTP协议是无状态的

**HTTP协议是无状态的，指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态**。也就是说，打开一个服务器上的网页和你之前打开这个服务器上的网页之间没有任何联系。

**HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（无连接）**。

## 概述

**在 HTTP/1.0 中，默认使用的是短连接**。也就是说，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。如果客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源，如JavaScript文件、图像文件、CSS文件等；当浏览器每遇到这样一个Web资源，就会建立一个HTTP会话。

但**从 HTTP/1.1 起，默认使用长连接**，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码：

Connection:keep-alive

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如 Apache ）中设定这个时间。实现长连接要客户端和服务端都支持长连接。

**HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接**。

### 连接池

在性能比较敏感的程序中，为了节省TCP网络调用建立连接三次握手的时间， 很多程序都会选择采用复用之前已经建立过的连接的方法来优化。再加上往往是“请求、响应、请求、响应”的模式，单个连接限制了QPS（Query Per Second）的提升。所以会采取开启多个连接组成一个“池子”的方式来优化性能，我们称之为连接池。

#### 概述

连接池是一个用来维护可复用连接的数据结构，正确地使用连接池可以达到减少网络往返损耗，降低系统资源占用，提升响应性能的目的。连接池主要的思想是把新建的连接暂存到池子中，当请求结束后不关闭连接，而是放回到连接池中，需要的时候从连接池中取出连接使用。

当一个请求结束后，如果没有连接池，那么会关闭这个TCP链接，再发起请求时需要走一遍重新建立连接，打开端口等流程。

而有了连接池，当一个请求结束后，可以在返回请求后，把这个连接放到内存队列中。再来请求时直接从队列获取一个连接即可。

最大连接数就是队列的长度。

#### 实现

开源的实现方案。

### TCP连接过程

当网络通信时采用TCP协议时，在真正的读写操作之前，server与client之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接 时它们可以释放这个连接，连接的建立是需要3次握手的，而释放则需要4次握手，所以说每个连接的建立都是需要资源消耗和时间消耗的。

经典的三次握手示意图：



经典的四次握手关闭图：



### TCP短连接

TCP短连接的情况：client向server发起连接请求，server接到请求，然后双方建立连接。client向server发送消息，server回应client，然后一次读写就完成了，这时候双方任何一个都可以发起close操作，不过一般都是client先发起 close操作。为什么呢，一般的server不会回复完client后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在client/server间传递一次读写操作。

短连接的优点是：管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段。

### TCP长连接

长连接的情况：client向server发起连接，server接受client连接，双方建立连接。Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

首先说一下《TCP/IP详解》上讲到的**TCP保活功能**（HTTP无状态，不具备保活功能），**保活功能**主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将应远等待客户端的数据，**保活功能就是试图在服务器端检测到这种半开放的连接**。

如果一个给定的连接在两小时内没有任何的动作，则服务器就向客户发一个探测报文段，客户主机必须处于以下 4 个状态之一：

1、客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。

2、客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在75秒后超时。服务器总共发送10个这样的探测，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

3、客户主机崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。

4、客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探查的响应。

### 长连接短连接操作过程

短连接的操作步骤是：

建立连接——数据传输——关闭连接…建立连接——数据传输——关闭连接

长连接的操作步骤是：

建立连接——数据传输…（保持连接）…数据传输——关闭连接

## 优点/缺点

由上可以看出，长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。**对于频繁请求资源的客户来说，较适用长连接**。

不过这里存在一个问题，存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。

在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server 早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但**如果客户请求频繁，将在TCP的建立和关闭操作上浪费时间和带宽**。

长连接和短连接的产生在于client和server采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

## 使用

长连接多用于**操作频繁，点对点的通讯**，而且**连接数不能太多**情况。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：**数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，而且频繁的socket创建也是对资源的浪费**。

而像**WEB网站的http服务一般都用短链接**，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以**并发量大，但每个用户无需频繁操作情况下需用短连好**。

# TCP/IP协议

协议是指计算机通信网络中，两台计算机进行通信，所必须遵循的规定或规则。

计算机网络中有各种各样的协议，比如TCP协议、IP协议、HTTP协议、FTP协议等等。像这样把与互联网相关联的协议集合起来，就称为TCP/IP协议族。



## 分层管理

TCP/IP协议族最重要的一点就是分层。按层次分别分为以下4层：

应用层：负责处理特定的应用程序，TCP/IP协议族内预存了各类通用的应用服务。比如FTP、DNS等等，HTTP也在该层。

传输层：对上层应用层提供处于网络连接中两台计算机之间的数据传输。该层有两个不同的协议：TCP和UDP。

网络层：处理网络上流动的数据包，该层协议有：IP、ICMP、IGMP。

数据链路层：处理连接网络的硬件部分。包括操作系统、硬件设备驱动、网卡、光纤等。

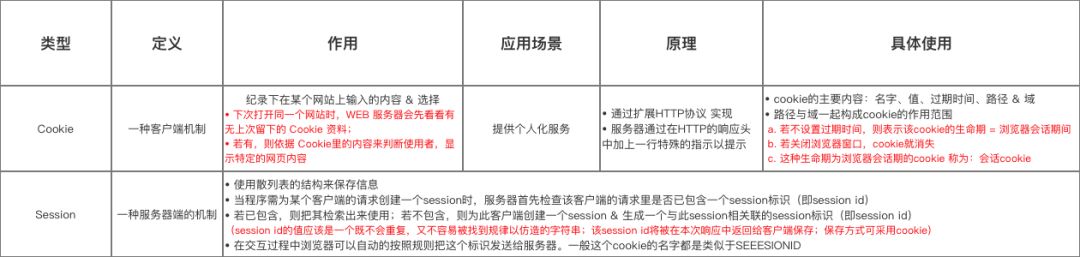
注：socket是对TCP/IP协议的封装。

## 流程图



# Cookie与Session

## 简介



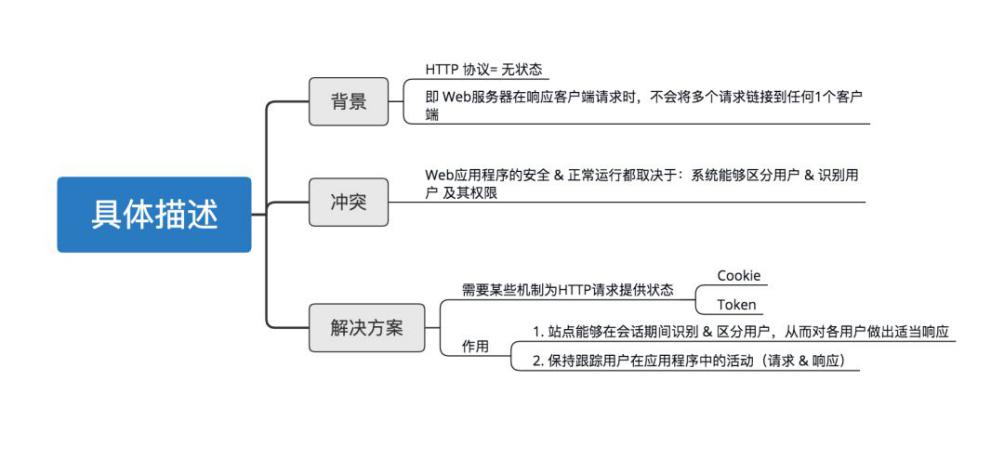
## 区别



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | cookie | localstorage | sessionstorage | session |
| 存储信息 | K-V存储，同域名可用 | | | |
| 存储位置 | 客户端 | | | 服务端 |
| 特点 | 随请求头每次提交 | 不随请求头提交  可长期保存 | 不随请求头提交  页面关闭即失效 | 安全 |
| 跨页 | 可跨页  不可跨域 | | 不可跨页  不可跨域 | 可跨页  不可跨域 |

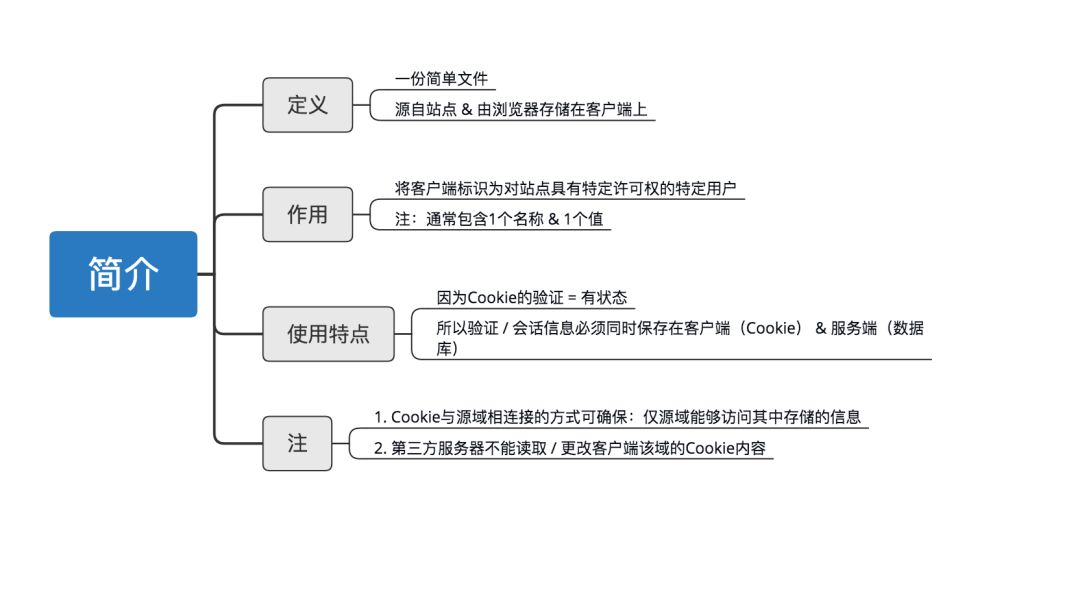
# Cookie与Token

## 简介



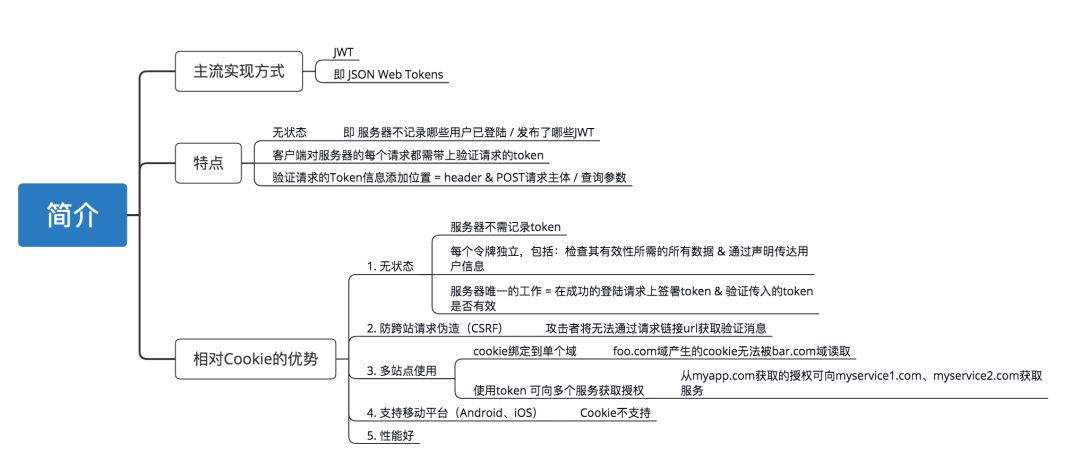
## 原理

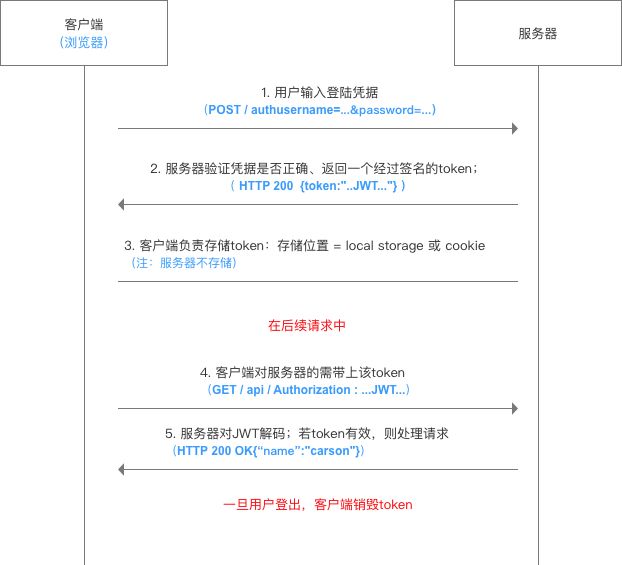
**基于Cookie的身份验证 & 验证流程：**





**基于Token的身份验证 & 验证流程：**





# IP/TCP/DNS

## IP协议

### 简介

IP（Internet Protocol），翻译成网络协议，位于网络层。

### 作用

IP的作用：把各种数据包传送给对方。如果要保证正确传送到对方那里，则需要满足各类条件，其中最重要的两个条件是IP地址和MAC地址。

IP地址：指明了节点被分配到的地址

MAC地址：是指网卡所属的固定地址

区别：IP地址可变换，但MAC地址基本上不会更改。

### ARP

使用ARP协议凭借MAC地址进行通信：

ARP是一种解析地址的协议，根据通信放的IP地址就可以反向查出对应的MAC地址。



## TCP协议

### 简介

TCP（Transmission Control Protocol），翻译成传输控制协议，位于传输层。

### 作用

提供可靠的字节流服务。

### 三次握手

TCP采用三次握手的策略保证提供可靠的服务。



注：SYN🡪SYN+ACK🡪ACK

## DNS

### 简介

DNS（Domain Name System），翻译成域名系统，位于应用层。

### 作用

提供域名到IP地址之间的解析服务。



注：客户端向服务器发送数据时，首先是通过DNS解析出IP地址，然后再发送HTTP请求。

## 联系

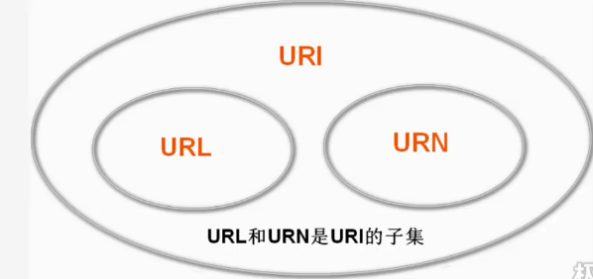


# URI与URL

URI：Uniform Resource Identifier，统一资源标识符；

URL：Uniform Resource Locator，统一资源定位符；

URN：Uniform Resource Name，统一资源名称。



注：URI是抽象的，URL和URN是具体的。

## URI

### 简介

URI（Uniform Resource Identifier），翻译为统一资源**标识符**，是一个用于标识某一互联网资源名称的字符串。

### URI格式

**格式：**

[http://user:pass@www.example.com:80/home/index.html?age=11#mask](http://user:pass@www.example.com:80/home/index.html?age=11" \l "mask)

**http：**协议方案名

**user:pass：**登录信息（认证）

[www.example.com](http://www.example.com)：服务器地址

**80：**端口号

/home/index.html：文件路径

age=11：查询字符串

**mask：**片段标识符

**协议方案名：**http:、https:、ftp:等，在获取资源时要指定协议类型

**登录信息（认证）：**指定用户名和密码作为从服务器端获取资源时必要的登录信息，此项是可选的。

**不带用户名和密码访问FTP服务器：**<ftp://test.com/>

**只带用户名访问FTP服务器：**<ftp://username@baidu.com>

**带用户名和密码访问FTP服务器：**ftp://username:password@baidu.com/

**服务器地址：**使用绝对URI必须指定待访问的服务器地址。

**服务器端口号：**指定服务器连接的网络端口号，此选项是可选的。常用默认端口：HTTP（80）、FTP（21）、Telnet（23）、HTTPS（443）、SMTP（25）、POP3（110）等等。

**路径：**指定服务器上的文件路径来定位特定资源。格式为：/home/index.html

**参数：**为应用程序提供访问资源所需的附加信息，例如：ftp:/127.27.27.27/pub/pic;type=d

**查询字符串：**针对已指定的文件路径内的资源，可以使用查询字符串传入任意参数，即传查询条件到服务器，来缩小请求资源的范围。此项是可选的。



**片段标识符：**通常可标记出以获取资源中的子资源（文档内的某一位置），此选项是可选的。

例如：http://www.baidu.com/path/php/#footer

## URL

### 简介

URL（Uniform Resource Location），翻译为统一资源**定位符**，它描述一台特定服务器上某特定资源的特定位置。

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

URI由两个主要的子集URL（通过位置识别资源）和URN（通过名字识别资源）构成。

注：URN（Uniform Resource Name，统一资源**名称**）没有得到广泛应用。

### 组件作用

大多数URL语法都建立在这个由9部分构成的通用格式上：

<schema>://<user>:<password>@<host>:<port>/<path>;<param>?<query>#<frag>



schema**协议方案名：**http:、https:、ftp:等，在获取资源时要指定协议类型

**登录信息（认证）：**指定用户名和密码作为从服务器端获取资源时必要的登录信息，此项是可选的。

host**服务器地址：**使用绝对URI必须指定待访问的服务器地址。

port**服务器端口号：**指定服务器连接的端口号（用于一台主机上区分不同的服务），此项是可选的。

path**路径：**指定服务器上的文件路径来定位特定资源，格式为：/home/index.html。

params**参数：**为应用程序提供访问资源所需的附加信息。例如：<ftp://127.0.0.1/pub/pic;type=d（FTP>传输格式有文本和二进制两种，type=d表示选择二进制传输FTP的文件）

query**查询字符串：**传查询条件到服务器，来缩小请求资源的范围。针对已指定的文件路径内的资源，可以使用查询字符串输入任意参数，此项是可选的。例如：http://www.baidu.com/course/php/?pageNum=2



**frag片段标识符：**通常可标记出以获取资源中的子资源（文档内的某一个位置），此项是可选的。例如：http://www.baidu.com/path/php/#footer

### 相对URL

相对URL：URL是不完整的

基础URL：必须要有一个参考点



#### 解析过程

1、找到基础URL



2、将相对URL转为绝对URL



#### 自动拓展URL

自动拓展URL的两种方式：

1. 主机名扩展

例如：输入baidu🡪浏览器自动拓展为www.baidu.com

1. 历史扩展

例如：输入bai🡪浏览器自动填充baidu.com

### 编码机制

用ASCII字符集中的有限子集对任意ASCII字符集外的字符值或数据进行编码。

格式：一个百分号（%）、后面跟两个表示字符ASCII码的十六进制数



### URL保留及受限字符

所谓保留字就是那些在URL中具有特定意义的字符。

受限（不安全）字符是指那些在URL中没有特殊含义，但在URL所在的上下文中可能具有特殊意义的字符，例如双引号。



# HTTP请求方法

## GET

### WampServer

Wamp就是Windows Apache MySQL PHP集成安装环境，即在windows下的apache、php和mysql的服务器软件。



### GET请求方法

请求指定的页面信息，并返回实体主体。

注：GET也可以将数据提交到服务器，不过一般都是使用POST提交数据到服务器。

应用：

exec 8<> /dev/tcp/www.baidu.com/80

注：建立TCP 3次握手（8<>表示输入输出都由数字8代表）

echo -e “GET /HTTP/1.0\n” >& 8

注：发送HTTP请求

cat <& 8

注：获取返回结果

## POST

### POST请求方法

POST请求方法的作用：向服务器发送数据



### 区别

POST和GET请求方法的本质区别：

1. GET用于信息**获取**，它是安全的（这里安全的含义是指非修改信息），而POST是用于**修改**服务器上资源的请求；
2. GET请求的数据会附在URL之后（然后提交到服务器），而POST把提交的数据则放置在HTTP实体的主体里，所以，**POST的安全性要比GET的安全性高**；
3. GET方式提交数据和POST方法提交数据并没有限制数据大小。

## 其他方法

### HEAD

HEAD请求方法作用：只请求头部，允许客户端在未获取资源实体的情况下，对资源的头部检查，了解资源的基本情况，比如类型，是否被修改过等。

HEAD请求方法只请求页面的首部：



HEAD和GET请求方法的区别：GET请求回来的报文有实体的主体部分，而HEAD请求回来的报文没有实体的主体部分。



### 其他

PUT：向指定资源位置上传其最新内容。

DELETE：请求服务器删除指定的页面。

OPTIONS：返回服务器针对特定资源所支持的 HTTP 请求方法。

TRACE：回显服务器收到的请求，主要用于测试或诊断。

PATCH：实体中包含一个表，表中说明与该URI所表示的原内容的区别。

MOVE：请求服务器将指定的页面移至另一个网络地址。

COPY：请求服务器将指定的页面拷贝至另一个网络地址。

LINK：请求服务器建立链接关系。

UNLINK：断开链接关系。

WRAPPED： 允许客户端发送经过封装的请求。

# HTTP状态码

## 信息性状态码

100 Continue

说明收到了请求的初始部分，请客户端继续。发送了这个状态码之后，服务器在收到请求之后必须进行相应。



注：这是一种优化手段，客户端在发送数据前确认一下服务器状态。

101 Switching Protocols

说明服务器正在根据客户端的指定，将协议切换成 Upgrade 首部所列的协议。



注：简单的理解就是切换协议版本，HTTP1.1中加入的。

## 成功状态码

200 OK

请求成功，实体的主体部分包含了所请求的资源。

201 Created

表示服务器在请求的响应中建立了新的文档。

202 Accepted

请求已被接受，但服务器还未对其执行任何操作。

203 Non-Authoritative Information

请求已成功，但实体首部包含的信息不是来自于源服务器，而是来自本地或者第三方的一份副本。

204 No Content

服务器成功处理了请求，但没有返回任何内容（实体的主体部分）。

205 Reset Content

告知浏览器清除当前页面中所有HTML表单元素

206 Partial Content

成功执行了一个部分或者Range（范围）请求。

## 重定向错误状态码

300 Multiple Choices

表示被请求的文档可以在多个地方找到，并将在返回的文档中列出来。服务器可以在Location首部包含首选URL。

301 Moved Permanently

被请求的资源已永久移动到新位置，新的永久性的URL在响应的Location首部返回，并且会自动重定向到该URL去。

302 Found

与301状态码类似，只是Location首部所给的URL被理解为临时重定向，而不是永久重定向。

303 See Other

与301、302状态码类似，其主要目的是允许POST请求的响应将客户端定位到某个资源上去。

304 Not Modified

客户端发起一个GET请求，如果最近资源未被修改的话，就可以用这个状态码说明资源未被修改。

305 Use Proxy

被请求的资源必须通过指定的代理才能被访问。

307 Temporary Redirect

请求的资源现在临时从不同的URI 响应请求，与302状态码类似。

## 客户端错误状态码

400 Bad Request

用于告知客户端它发送了一个错误的请求。

401 Unauthorized

当前请求需要用户验证。

403 Forbidden

服务器已经理解请求，但是拒绝执行它。

404 Not Found

请求失败，请求所希望得到的资源未被在服务器上发现。

405 Method Not Allowed

指出请求方法对某些特定的资源不允许使用。

406 Not Acceptable

当前请求需要用户验证。

407 Proxy Authentication Required

该状态指出客户端必须通过代理服务器的认证。

408 Request Timeout

指服务端等待客户端发送请求的时间过长。

409 Conflict

该状态通常与PUT请求一同使用，常被用于试图上传版本不正确的文件时。

411 Length Required

服务器不能处理请求，除非客户端发送Content-Length头信息指出发送给服务器的数据的大小。

412 Precondition Failed

指出请求头信息中的某些先决条件是错误的。

413 Request Entity Too Large

告诉客户端现在所请求的文档比服务器现在想要处理的要大。

414 Request URI Too Long

请求的URL长度超过了服务器能够解释的长度，因此服务器拒绝对该请求提供服务。

415 Unsupported Media Type

对于当前请求的方法和所请求的资源，请求中提交的实体并不是服务器中所支持的格式，因此请求被拒绝。

416 Requested Range Not Satisfiable

表示客户端包含了一个服务器无法满足的Range头信息的请求。

## 服务器错误状态码

500 Internal Server Error

服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。

501 Not Implemented

服务器不支持当前请求所需要的某个功能。

502 Bad Gateway

作为网关或者代理的服务器尝试执行请求时，从上游服务器接收到无效的响应。

503 Service Unavailable

由于临时的服务器维护或者过载，服务器当前无法处理请求。

504 Gateway Timeout

服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。

505 HTTP Version Not Supported

说服务器并不支持在请求中所标明 HTTP 版本。

# 报文格式

## HTTP事务

HTTP事务 = 请求命令 + 响应结果



注：报文是易于阅读的字符，而不是二进制格式。

## 报文流



报文都是向下游流动的：



## 请求报文



请求报文的格式：

**请求行：**请求方法 请求URL 版本

**首部：** 名：值

（空行）

**实体的主体部分：**（HTTP要传输的内容）

## 响应报文



响应报文的格式：

**响应行：**协议版本 状态码 原因短语

**首部：**名: 值

（空行）

**实体的主体部分：**（HTTP要传输的内容）

# 工作流程

## 整个流程步骤



## 域名解析过程



注：可以通过浏览器[chrome://net-internals/#dns](chrome://net-internals/" \l "dns)这个URL进行清除浏览器缓存的DNS。

查看操作系统自身的DNS缓存数据：ipconfig /displaydns，通过ipconfig flushdns刷新DNS缓存。

Windows主机本地的hosts文件：C:/windows/system32/drivers/etc/hosts

## 三次握手过程



## 发起HTTP请求

建立TCP连接后发起HTTP请求



## 响应HTTP请求并得到HTML代码

服务器端响应HTTP请求，浏览器得到HTML代码



## 浏览器解析HTML代码

浏览器拿到HTML文档后，开始解析HTML代码。

当遇到JS/CSS/图片等静态资源时，会自动向服务器端请求下载。



## 浏览器对页面进行渲染呈现给用户

最后一步，浏览器利用自己内部的工作机制，把请求到的HTML代码和静态资源进行渲染，渲染最后，呈现给用户。

# Curl

## 概述

Curl用来做HTTP请求。

## 指令

# Telnet

它是TCP/IP协议族中的一员，是Internet远程登陆服务的标准协议和主要方式。

为用户提供了在本地计算机上完成远程主机工作的能力。

在终端使用者的电脑上使用telnet程序，用它连接到服务器。

使用Telnet程序连接HTTP服务器（步骤）

第一步，打开命令窗口（按win+R键，在运行窗口输入cmd，运行弹出命令窗口）

第二步，输入 telnet www.jikexueyuan.com 80

第三步，按下 ctrl键 + }键

第四步，按下 enter键

第五步，可将http请求报文复制进去

最后看响应的内容是不是我们想要的。