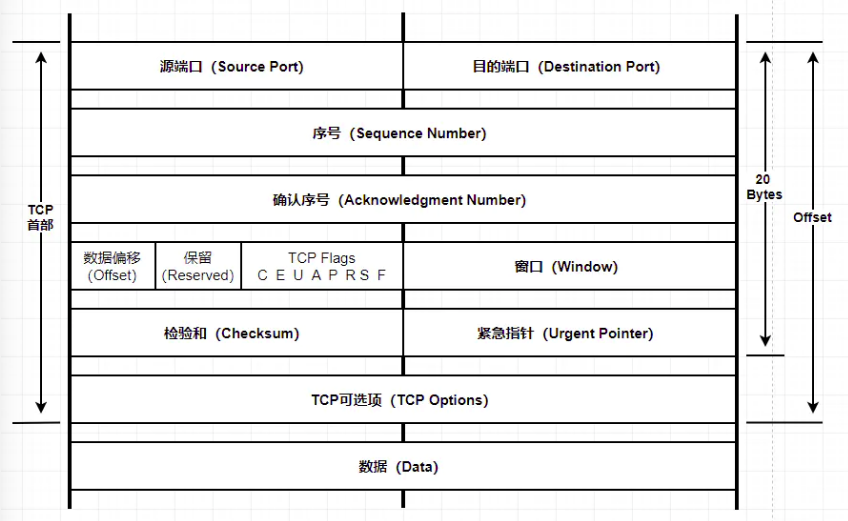
## 一、TCP、UDP、HTTP

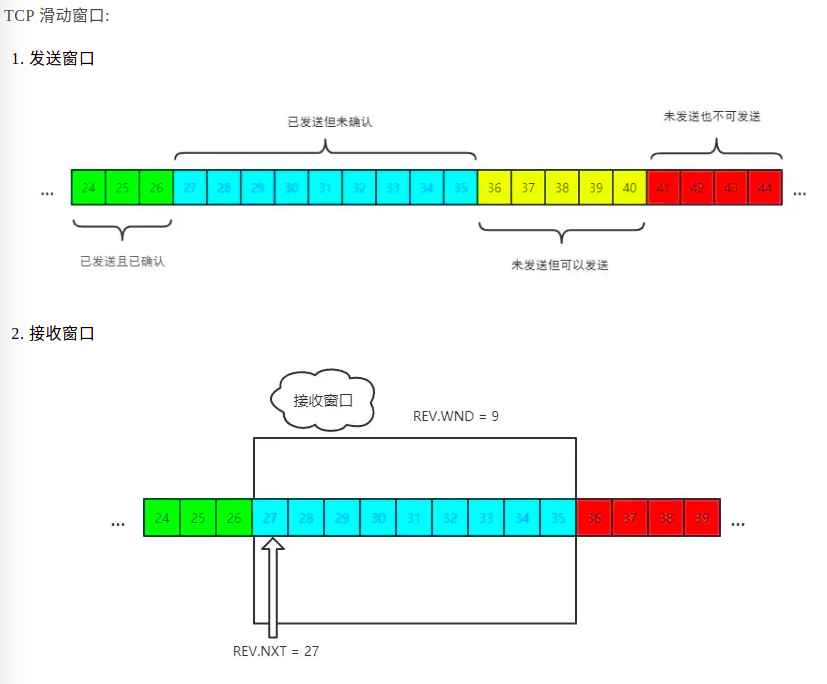
### 1.TCP和UDP区别



Tcp报文结构



Tcp滑动窗口



### TCP三次握手

#### 为什么建立连接需要三次握手？

首先非常明确的是两次握手是最基本的。第一次握手，客户端发了个连接请求消息到服务端，服务端收到信息后知道自己与客户端是可以连接成功的，但此时客户端并不知道服务端是否已经接收到了它的请求，所以服务端接收到消息后的应答，客户端得到服务端的反馈后，才确定自己与服务端是可以连接上的，这就是第二次握手。

**第三次握手是为了防止已经失效的连接请求报文段突然又传到服务端，因而产生错误。**

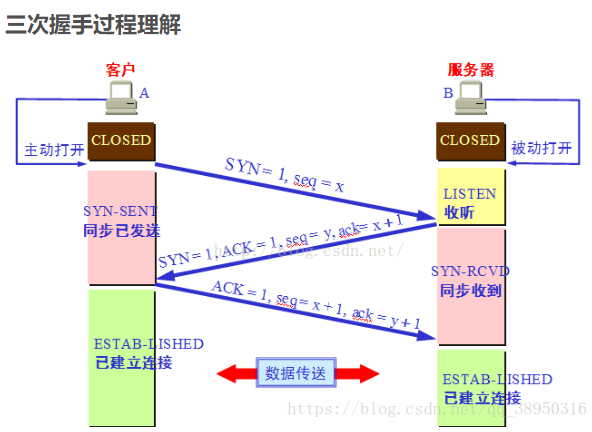
譬如发起请求遇到类似这样的情况：客户端发出去的第一个连接请求由于某些原因在网络节点中滞留了导致延迟，直到连接释放的某个时间点才到达服务端，**这是一个早已失效的报文，**但是此时服务端仍然认为这是客户端的建立连接请求第一次握手，于是服务端回应了客户端，第二次握手。

如果只有两次握手，那么到这里，连接就建立了，**但是此时客户端并没有任何数据要发送**，而服务端还在傻傻的等候佳音，造成很大的资源浪费。所以需要第三次握手，只有客户端再次回应一下，就可以避免这种情况。

**如果是三次握手，当客户端没有数据要进行发送，则不会发送第三次握手信息，那么连接也就不会建立，服务端也不会等待接收数据。**

**也就是第三次握手是为了保证第一次握手的有效性。**

**客户端在第二次握手时连接建立，而服务端在第三次握手时连接建立。**



**序列号seq：**占4个字节，用来标记数据段的顺序，TCP把连接中发送的所有数据字节都编上一个序号，第一个字节的编号由本地随机产生；给字节编上序号后，就给每一个报文段指派一个序号；序列号seq就是这个数据报文段中的第一个字节的数据编号。

**确认号ack**：占4个字节，期待收到对方下一个报文段的第一个数据字节的序号；

**确认ACK：**占1位，仅当ACK=1时，确认号字段才有效，确认可建立连接。ACK=0时，确认号无效。

**同步SYN：**连接建立时用于同步序号。**当SYN=1，ACK=0时表示**：这是一个连接请求报文段。若同意连接，则在响应报文段中使得**SYN=1，ACK=1。**因此，SYN=1表示这是一个连接请求，或连接接受报文。SYN这个标志位只有在TCP建产连接时才会被置1，握手完成后SYN标志位被置0。

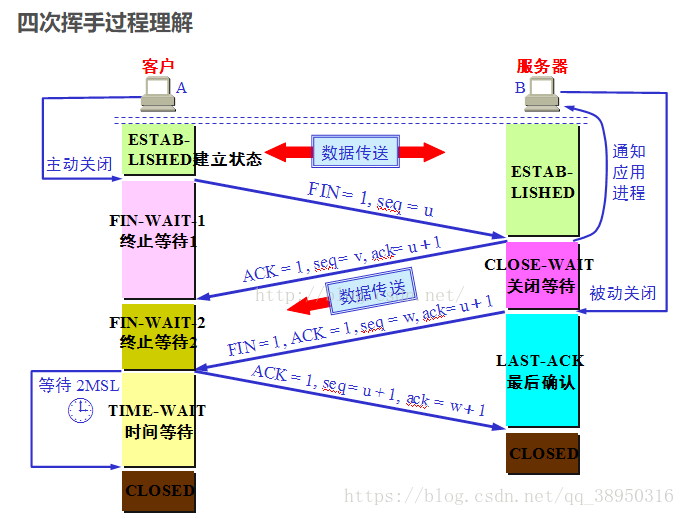
**终止FIN：**用来释放一个连接。FIN=1表示：**此报文段的发送方的数据已经发送完毕**，并要求释放运输连接。

**PS：ACK、SYN和FIN这些大写的单词表示标志位，其值要么是1，要么是0；ack、seq小写的单词表示序号。**

#### 如果已经建立了连接，但是客户端突然出现故障了怎么办？

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

### TCP四次挥手



#### 为什么建立连接是三次握手，而断开连接是四次握手？

其实三次握手是两个阶段：

1. A向B发送请求建立连接消息，如果B确认能和A建立连接，B发送确认消息给A，A接收到后，A端的连接就建立了；
2. B再发送请求建立连接消息，问是否真的有数据发送，如果有，A就发送确认消息给B，这个时候B端的连接建立

而四次握手其实也是两个阶段：

1. A向B发送数据传输结束的断开连接消息，B端确认接收到连接，确认可以断开连接，但是不能马上断开，要等B端的数据传送完，但是会先给A发送可以断开连接的确认消息。
2. B端的数据传送完了，那么向A端发送可以断开连接的消息，A端接收到确认消息，如果真的要断开连接，就发送确认消息，B端接收到消息后，马上断开连接，不会一直等着。A端经过2MSL的等待时间后，自动断开连接。

所以先建立连接的是客户端，因为是客户端发起的请求建立连接，

但是先断开连接的是服务端，也是因为是客户端发起请求断开连接，这样的话就能保证不会漏掉服务端传过来的数据。

所以为什么一个三次一个是四次的原因就是，三次握手的B端往A端发送确认消息和建立连接消息可以一起发，即ACK+SYN，而四次挥手，只能先发ACK，等到B端数据传输完成后，再发送FIN。也就是说FIN和ACK不能一起发送，导致的多了一次挥手过程。

#### **为什么不是两次握手和三次挥手？**

第三次握手是客户端向服务端发送确实有数据发送消息，你之前接收的消息是即时的。

第四次挥手是客户端向服务端发送确实没有数据发送了消息，你之前接收的消息是即时的。

**也就是说为了服务端确认接收的消息是即时的**

**两次握手和三次挥手其实只有一方确认了对方。**

#### 大白话总结三次握手和四次挥手

**三次握手**

1. **客户端：我有要传送的数据段，我要建立连接（SYN = 1），我的数据段第一个字节的数据编号是x（seq=x）。**
2. **服务端：好的我收到了（ACK = 1），你确定你有要传送的数据段吗（SYN=1），如果你有的话，我希望你下一个数据段的第一个字节的数据编号是x+1（ack = x+1），对了，这是我的数据段的第一个字节的数据编号（seq=y）**
3. **客户端：是的，我有要传送的数据段（ACK=1），这是我下一个数据段的第一个字节的数据编号是x+1（seq=x+1），我想要你下一个数据段的第一个字节的数据编号是y+1（ack=y+1），开始建立连接咯！**
4. **服务端：开始建立连接。**

**四次挥手**

1. **客户端：我的数据发送完咯（FIN = 1），这是我最后一个数据段了（seq=u）**
2. **服务端：好的（ACK=1），那你等我一会啊（seq=v，ack=u+1）**
3. **服务端：好了（ACK=1），我的数据也发送完了，你确定你没有数据要发送了吗？（FIN = 1，seq=w, ack=u+1）**
4. **客户端：是的，我确认没了（ACK=1,seq=u+1,ack=w+1）,你不用回我了，我等你2MSL时间，你要是要事就快点说，时间过了，我自己断开连接咯。**
5. **服务端：好的，那我断开连接了。**

#### 为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可能最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。在Client发送出最后的ACK回复，但该ACK可能丢失。**Server如果没有收到ACK，将不断重复发送FIN片段。**所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME\_WAIT状态。**Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。**所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

### 4.TCP拥塞控制

拥塞：**即对资源的需求超过了可用的资源**。若网络中许多资源同时供应不足，网络的性能就要明显变坏，整个网络的吞吐量随之负荷的增大而下降

拥塞控制：**防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。**

拥塞控制的方法： 慢启动( slow-start )、拥塞避免( congestion avoidance )、快重传( fast retransmit )和快恢复( fast recovery )

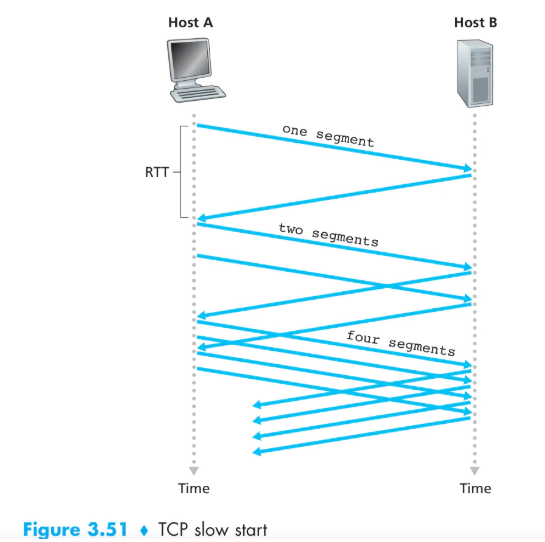
上述方法都是针对拥塞窗口的，拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口。如果再考虑到接收方的接收能力，那么发送窗口还可能小于拥塞窗口。

#### 慢启动

思路：主机开始发送数据报时，如果立即将大量的数据注入到网络中，可能会出现网络的拥塞。**慢启动算法就是因为现在并不清楚网络的负荷情况。因此，先探测一下，再逐渐增大拥塞窗口的数值**

通常在刚刚开始发送报文段时，先把拥塞窗口 cwnd 设置为一个最大报文段MSS的数值。而在每收到一个对新的报文段的确认后，**将cwnd值加倍。**

**别被“慢启动”这个名字所迷惑了，实际上这是cwnd增长最快的阶段。**



#### 拥塞避免

思路：**让拥塞窗口cwnd缓慢地增大，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍。**这样拥塞窗口cwnd按线性规律缓慢增长，比慢启动算法的拥塞窗口增长速率缓慢得多

为了防止cwnd增加过快而导致网络拥塞，所以需要设置一个**慢开启门限ssthresh状态变量**,它的用法：

1. 当cwnd < ssthresh,使用慢启动算法

2.当cwnd > ssthresh,使用拥塞避免算法，停用慢启动算法

3. 当cwnd = ssthresh，这两个算法都可以

**拥塞避免算法不能够完全的避免网络拥塞，通过控制拥塞窗口的大小只能使网络不易出现拥塞**

#### 快重传和快恢复

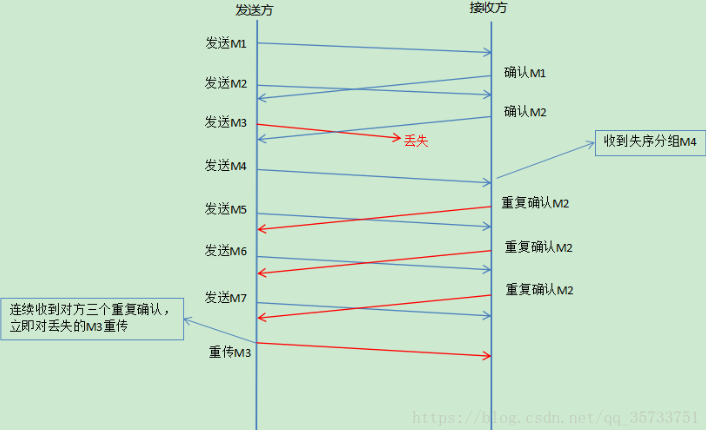
考虑这么一种情况：假设现在网络没有发生拥塞，但是发送方发送的个别数据包却在网络中某一处丢失了，而此时发送方的超时计时器超时了，又没有收到确认。

在这种情况，tcp拥塞控制会误认为出现网络拥塞，然后马上把拥塞窗口cwnd减小到1（即一个mss），并启动慢开始算法，同时把门限值ssthresh减半。

这样就降低了通信效率，**所以基于这种情况就有了快重传和快恢复两个算法。**

**快重传：**

快重传算法首先要求接收方每收到一个失序的报文段后**就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时才进行捎带确认**。快重传算法还规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器到期

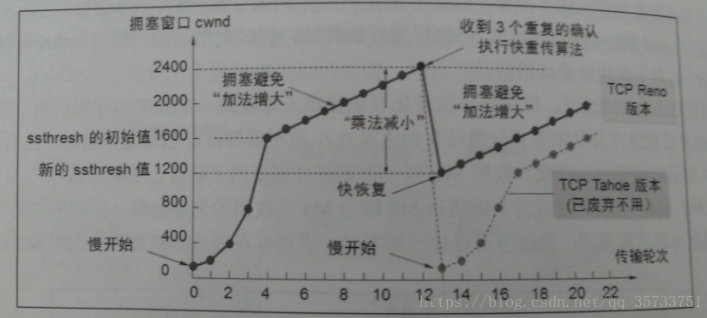


规定：接收方收到失序的M4分组后，应该立即对M2发出重复确认（总共发送三次重复确认），也就是告诉发送方：“我还没收到M3分组”。

然后发送方又接着发送了M5，M6，接收方收到后就会继续发送重复确认M2。这样发送方收到了接收方三个重复的M2确认。快重传算法规定，只要发送方一连收到了3个重复确认就应该立即重传对方尚未收到的M3，而不是等待超时计时器超时后再进行重传，这样发送方及时重传了丢失的分组，因此快重传算法提高了通信效率。

**快恢复：**

为了配合使用快重传算法，因为毕竟发送了三个重复确认，还要重传一个消息，可能消息过多，发生拥塞，所以当发送方连续接收到三重复个确认时，就执行乘法减小算法，**把慢启动开始门限（ssthresh）减半**， 然后执行拥塞避免算法，使拥塞窗口缓慢增大。



快恢复是之前采用的TCP Tahoe版本，发生快重传后，立即恢复慢开始状态；已经废弃不用，恢复的是拥塞避免状态。

所以快恢复是指快速恢复到当前窗口的一半，以防发生拥塞。

#### 总结

TCP三次握手建立之后，就要开始传输数据了，传输数据的通道可以称为窗口，为防止大量数据传输超过服务器和客户端的负荷，那么这个窗口的大小cwnd 就要慢慢增大；

1. 刚开始时将cwnd 设置为一个最大报文段MSS的大小。
2. 然后采用慢启动算法，即每次收到一个新的报文段的确认后，将cwnd翻倍，直到达到慢启动上限ssthresh。
3. 慢启动因为将cwnd翻倍，所以是最快的，很容易超负荷，所以cwnd过了慢启动上限后，就采用拥塞避免算法，将cwnd += 1。

通过慢启动和拥塞避免，可以保证拥塞窗口的大小不会超负荷运转。

但如果在传送数据报的时候，数据包丢失怎么办呢？

传统的方法是：tcp拥塞控制会认为这是因为窗口拥塞了导致的，那么就会直接将cwnd还原为一个MSS大小，然后将ssthresh减半，重新执行慢启动和拥塞避免。

而这很明显很耗费时间和资源，所以有了快重传和快恢复。

当发送方发送的一个数据报文丢失的时候，接收方会发现自己接收到了失序报文，会发送一个重复确认的响应，当接到三个失序报文时，那么就发送了三个重复确认，而快重传就是当发送方收到了三个重复确认后，就会重传丢失的那个数据报。

因为重传了数据报文，那么cwnd就增加了一个MSS的大小，可能会超负荷，所以就减小拥塞窗口的大小，原本是将cwnd恢复到1个MSS的大小，将ssthresh减半，而快恢复是将cwnd和sshthred同时减半。也可以这么认为，以前恢复的是慢启动状态，而现在恢复的是拥塞避免状态。

## 二、HTTP、TLS（SSL）

### HTTP基本概念

HTTP(Hyper Text Transfer Protocol)<超文本传输协议>的缩写.**是用于从WWW服务器传输超文本到本地浏览器的传输协议，也就是传送页面回来**

HTTP是一个应用层协议,**由请求和响应构成**,是一个标准的个客户端和服务器模型.

### HTTPS基本概念

网景（Netscape）在 1994 年设计了 HTTPS 协议，使用安全套接字层（Secure Sockets Layer，SSL）保证数据传输的安全，随着传输层安全协议（Transport Layer Security，TLS）的发展，目前我们已经使用 TLS 取代了废弃的 SSL 协议，不过仍然使用 SSL 证书一词

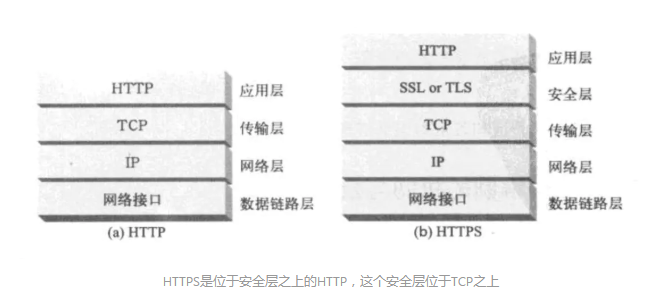
HTTP协议通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了我们常说的HTTPS。默认HTTP的端口号为80， HTTPS的端口号为443。

HTTPS在应用层（HTTP）和传输层（TCP）之间加入了一个安全层（SSL或TLS）。目的 是为了解决HTTP协议的几个缺点：

1、通信使用明文（不加密），内容可能会被窃听。

2、无法证明报文的完整性，所以有可能早已篡改。

3、不验证通信方的身份，因此有可能遭遇伪装。



**为了解决上述三个问题，在HTTP基础上加了加密防窃听，加了完整性保护防篡改，加了身份认证防伪装，前两者都是建立在后者的基础之上的。**

**HTTP+加密（对称秘钥）+防篡改（摘要算法-非对称秘钥）+防伪装（证书）=HTTPS**

### 无连接的概念

无连接的含义是限制每次连接只处理一个请求.服务器处理完客户端的请求,然后响应,并收到应答之后,就断开连接.这种方式可以节省传输时间.

### 无状态的概念

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**.这种方式的一个坏处就是,如果后续的处理需要用到之前的信息,则必须要重传,这样就导致了每次连接传输的数据量增大.好处就是,如果后续的连接不需要之前提供的信息,响应就会比较快.而为了解决HTTP的无状态特性,出现了Cookie和Session技术.

这样设计的原因是因为Web服务器一般需要面对很多浏览器的并发访问，为了提高Web服务器对并发访问的处理能力，在设计HTTP协议时规定Web服务器发送HTTP应答报文和文档时，不保存发出请求的Web浏览器进程的任何状态信息。

### HTTP请求

每一个HTTP请求都由三部分组成，分别是：请求行、请求报头、请求正文。

#### 5.1. 请求行

请求行一般由**请求方法、url路径、协议版本**组成

虽然HTTP请求方法有这么多种，但是我们平常使用的基本只有GET和POST两种方法，而且大部分网站都是禁用掉了除GET和POST外其他的方法。

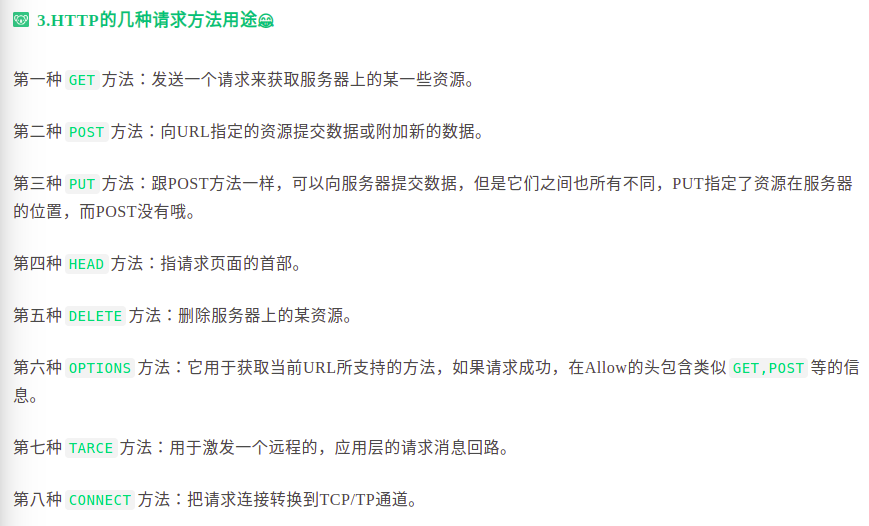
**因为其他几种方法通过GET或者POST都能实现，而且对于网站来说更加的安全和可控。**

**GET**

其实简单来说，GET方法一般用来负责获取数据，或者将一些简短的数据放到URL参数中传递到服务器。比POST更加高效和方便。

**POST**

**由于GET方法最多在url中携带1024字节数据，且将数据放到URL中传递太不安全，数据量大时URL也会变得冗长**。所以传递数据量大或者安全性要求高的数据的时候，最好使用POST方法来传递数据。



1. GET方法获取资源
2. POST方法传输资源
3. PUT方法更新资源
4. DELETE方法删除资源
5. HEAD方法获得报文首部
6. OPTIONS方法，用来跨域请求

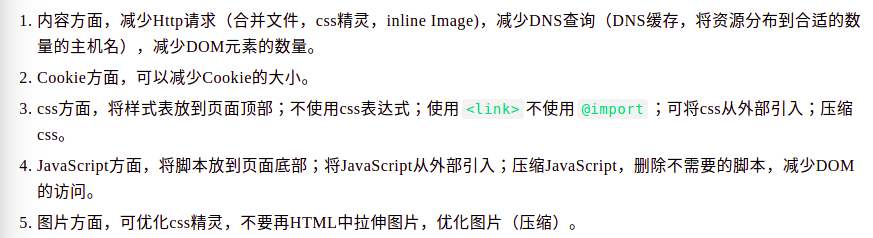
CONNECT方法，用来建立连接，用于代理服务器

#### 5.2. 请求报头

**请求行下方的是则是请求报头，HTTP消息报头包括普通报头、请求报头、响应报头、实体报头。每个报头的形式如下：**

**名字 + : + 空格 + 值**

**请求报头不像请求正文，它不能压缩，所以可能占据着大量带宽。因此就可以进行网站的性能优化：**



##### Host

指定的请求资源的域名（主机和端口号）。HTTP请求必须包含HOST，否则系统会以400状态码返回。

##### User-Agent

简称UA，内容包含发出请求的用户信息，通常UA包含浏览者的信息，主要是浏览器的名称版本和所用的操作系统。这个UA头不仅仅是使用浏览器才存在，只要使用了基于HTTP协议的客户端软件都会发送，无论是手机端还是PDA等，这个**UA头是辨别客户端所用设备的重要依据。**

##### Accept

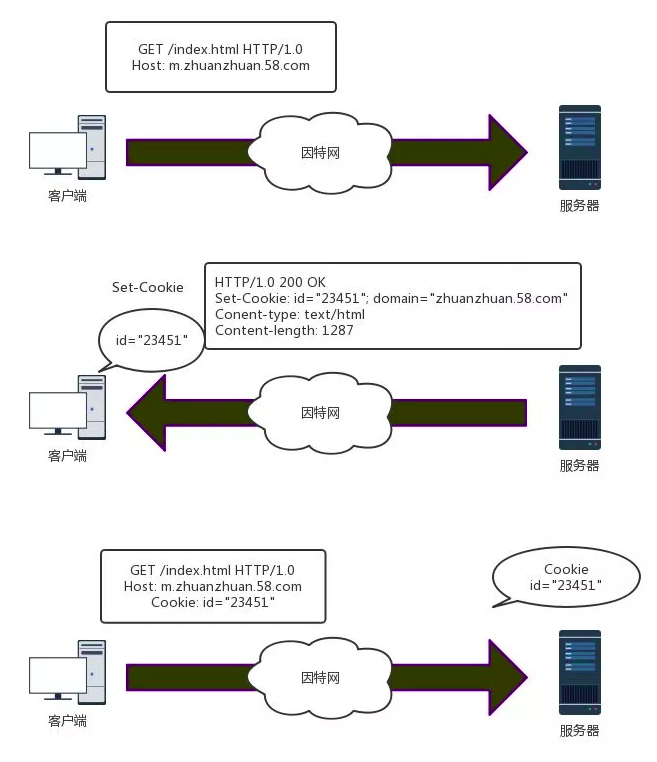
**告诉服务器可以接受的文件格式。**通常这个值在各种浏览器中都差不多，不过WAP浏览器所能接受的格式要少一些，这也是用来区分WAP和计算机浏览器的主要依据之一，随着WAP浏览器的升级，其已经和计算机浏览器越来越接近，因此这个判断所起的作用也越来越弱。

##### Cookie

**1、概念：**

Cookie其实就是**由服务器发给客户端的特殊信息，而这些信息以文本文件的方式存放在客户端，然后客户端每次向服务器发送请求的时候都会带上这些特殊的信息。** 服务器在接收到Cookie以后，会验证Cookie的信息，以此来辨别用户的身份。

Cookie可以理解为一个临时通行证。



1. **用处：**

Cookie其实是HTTP请求头的扩展部分，由于HTTP协议是无状态的协议，所以为了在网页上实现登陆之类的需求，所以扩展了Cookie这样的功能。

每一次HTTP请求在数据交换完毕之后就会关闭连接，所以下一次HTTP请求就无法让服务端得知你和上一次请求的关系。而使用了Cookie之后，**你在第一次登陆之类的请求成功之后，服务器会在Response的头信息中给你返回Cookie信息**，你下一次访问的时候带上这个Cookie信息，则服务器就能识别你为上一次成功登陆的用户。

1. **内容：**

Cookie一般保存的格式为json格式，由一些属性组成：

name：Cookie的名称

value：Cookie的值

domain：可以使用此Cookie的域名

path：可以使用此Cookie的页面路径

expires/Max-Age：此Cookie的超时时间，不设置的话默认值是Session，意思是cookie会和session一起失效，session的有效期一般是浏览器打开和关闭这段时间，所以一般而言Cookie是设置七天的，与session一样的话，时间太短了。

secure：设置是否只能通过https来传递此条Cookie

1. **domain属性**

域名一般来说分为顶级域名，二级域名，三级域名等等。

**例如[baidu.com](http://baidu.com" \t "_blank)是一个顶级域名，而[www.baidu.com](http://www.baidu.com" \t "_blank)和[map.baidu.com](http://map.baidu.com" \t "_blank)就是二级域名，**依次类推。

而在我们的Cookie来说，都有一个domain属性，这个属性限制了访问哪些域名时可以使用这一条Cookie。因为每个网站基本上都会分发Cookie，所以domain属性就可以让我们在访问新浪时不会带上百度分发给我们的Cookie。

而在同一系的域名中，**顶级域名是无法使用其二级域名的Cookie的**，也就是说访问baidu.com的时候是不会带上map.baidu.com分发的Cookie的，二级域名之间的Cookie也不可以共享。但访问二级域名时是可以使用顶级域名的Cookie的。**也就是能访问具体域名的肯定能访问抽象点的域名。**

##### Session（不包含在请求报头中）

Session，中文经常翻译为会话，**其本来的含义是指有始有终的一系列动作/消息**，比如打电话时从拿起电话拨号到挂断电话这中间的一系列过程可以称之为一个session。这个词在各个领域都有在使用。

而我们web领域，一般使用的是其本义，**一个浏览器窗口从打开到关闭这个期间。**

**Session的目的则是，**在一个客户从打开浏览器到关闭浏览器这个期间内，发起的所有请求都可以被识别为同一个用户。

**而实现的方式则是**，在一个客户打开浏览器开始访问网站的时候，会生成一个SessionID，这个ID每次的访问都会带上，**而服务器会识别这个SessionID并且将与这个SessionID有关的数据保存在服务器上**。由此来实现客户端的状态识别。

Session与Cookie相反，**Session是存储在服务器上的数据**，只由客户端传上来的SessionId来进行判定，**所以相对于Cookie，Session的安全性更高。**

一般SessionID会在浏览器被关闭时丢弃，或者服务器会验证Session的活跃程度，例如30分钟某一个SessionID都没有活跃，那么也会被识别为失效。

##### Cache-Control

**指定请求和响应遵循的缓存机制**。在请求消息或响应消息中设置Cache-Control并不会修改另一个消息消息处理过程中的缓存处理过程。**请求时的缓存指令**包括no-cache、no-store、man-age、max-stake、min-fresh、only-if-cached；**响应消息中的指令**包括 public、privete、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age。

##### Referer

页面跳转处，表明产生请求的网页来自于哪个URL，用户是从该 Referer页面访问到当前请求的页面。**这个属性可以用来跟踪Web请求来自哪个页面，是从什么网站来的**。

##### Content-Length

内容长度。

##### Content-Range

**响应的资源范围**。可以在每次请求中标记请求的资源范围，**在连接断开重连时，客户端只请求该资源未下载的部分，而不是重新请求整个资源，实现断点续传。**迅雷就是基于这个原理，使用多线程分段读取网络上的资源，最后再合并。

##### Accept-Encoding

**指定所能接收的编码方式，通常服务器会对页面进行GZIP压缩后再输出以减少流量，**一般浏览器均支持对这种压缩后的数据进行处理，但对于我们来说，如果不想接收到这些看似乱码的数据，可以指定不接收任何服务器端压缩处理，要求其原样返回。

##### Accept-Language

**指浏览器可以接受的语言种类** en、en-us指英语 zh、zh-cn指中文。

##### Connection

客户端与服务器链接类型，**keep-alive:保持链接，close:关闭链接。**

#### 5.3. 请求正文

请求正文通常是使用POST方法进行发送的数据，**GET方法是没有请求正文的。**

**请求正文跟上面的消息报头一般由一个空行隔开。**



### HTTP响应

HTTP响应同样也是由状态行、响应报头、报文主体三部分组成。

#### 6.1. 状态行

状态行由**HTTP协议版本号， 状态码， 状态消息**三部分组成。如下所示：

HTTP的状态码是由三位数字来表示的，由第一位数字来表示状态码的类型，一般来说有五种类型：

| 分类 | 分类描述 |
| --- | --- |
| 1\*\* | 信息，服务器收到请求，需要请求者继续执行操作 |
| 2\*\* | 成功，操作被成功接收并处理 |
| 3\*\* | 重定向，需要进一步的操作以完成请求 |
| 4\*\* | 客户端错误，请求包含语法错误或无法完成请求 |
| 5\*\* | 服务器错误，服务器在处理请求的过程中发生了错误 |

#### 6.2. 响应报头

##### Allow

服务器支持哪些请求方法（如GET、POST等）。

##### Date

表示消息发送的时间，时间的描述格式由rfc822定义。例如，Date:Mon,31Dec200104:25:57GMT。Date描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。

##### Set-Cookie

非常重要的header, **用于把cookie发送到客户端浏览器**，每一个写入cookie都会生成一个Set-Cookie。

##### Expires

指明应该在什么时候认为文档已经过期，从而不再缓存它，重新从服务器获取，会更新缓存。过期之前使用本地缓存。

##### Content-Type

WEB服务器告诉客户端自己响应的对象的类型和字符集。

##### Content-Encoding

文档的编码（Encode）方法。只有在解码之后才可以得到Content-Type头指定的内容类型。利用gzip压缩文档能够显著地减少HTML文档的下载时间。

##### Content-Length

指明实体正文的长度，以字节方式存储的十进制数字来表示。

##### Location

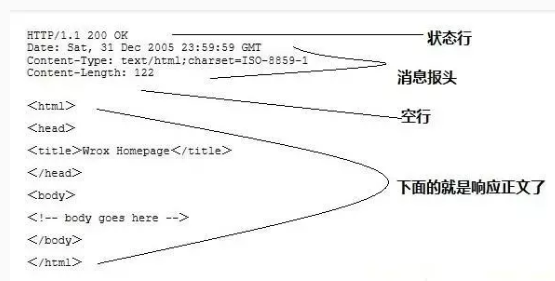
用于重定向一个新的位置，包含新的URL地址。表示客户应当到哪里去提取文档。

##### Refresh

表示浏览器应该在多少时间之后刷新文档，以秒计。

#### 6.3. 响应正文

**服务器返回的数据。**



### HTTP工作原理

**1.客户端连接到web服务器**

一个HTTP客户端,通常是浏览器,与Web服务器的HTTP端口(默认是80)建立一个TCP套接字连接.

**2.发送HTTP请求**

通过TCP套接字,客户端向Web服务器发送一个文本的请求报文,一个请求报文由请求行,请求头部,空行和请求体4个部分构成.

**3.服务区接收解释请求并返回HTTP响应**

Web解析请求,定位请求资源.服务器将资源复本写到TCP套接字,由客户端获取.一个响应由状态行,响应 头,空行和响应数据4部分组成.

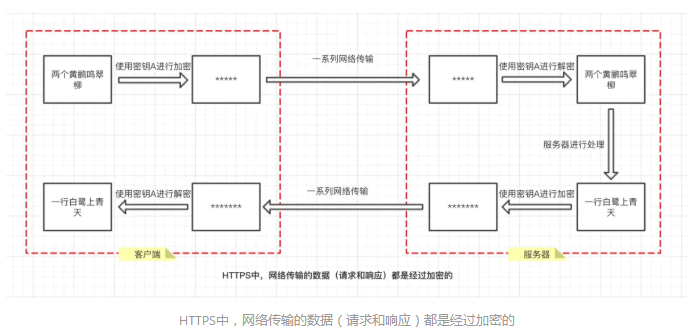
**4.释放连接TCP连接**

若Connection模式为close,则服务器主动关闭TCP连接,客户端被动关闭TCP连接,释放TCP连接.若Connection为keepalive,则该连接会保持一段时间,该时间内可以持续使用该连接接收请求,做出响应。

**5.客户端浏览器解析HTML内容**

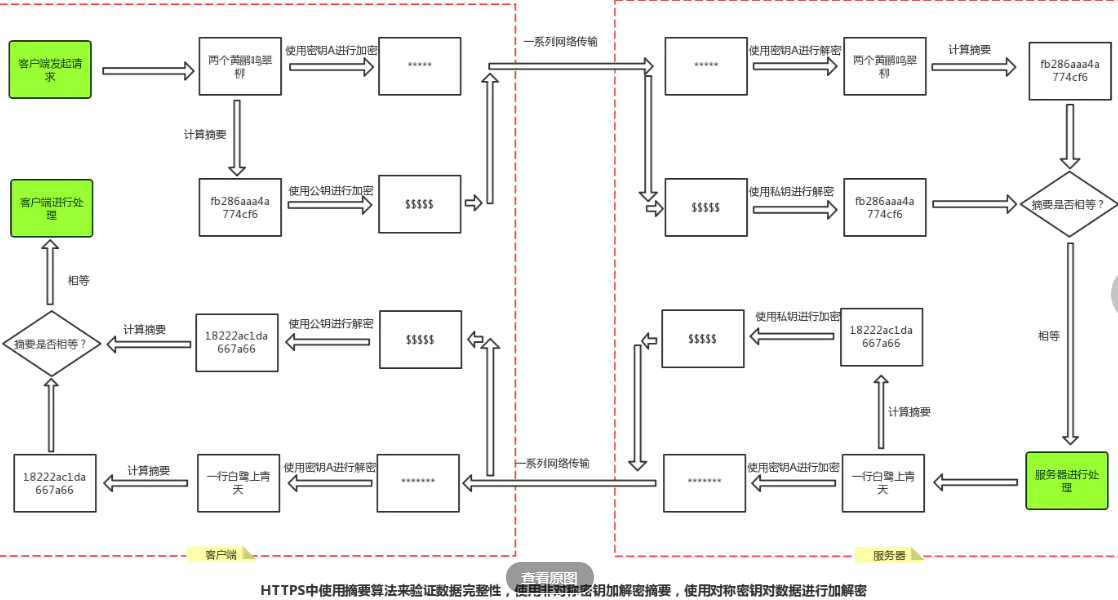
### **8.HTTPS = HTTP + 加密 + 完整性保护 + 认证**

#### 1.加密



像这种加密和解密都是使用同一个密钥的加密方式叫做对称加密（也叫共享加密，共同拥有一个密钥）。使用密钥A加密的内容，只能用密钥A来解密，其他的密钥都无法解密。常用的对称加密算法有DES,3DES,AES。  
 还有一种加密方式叫非对称加密（也叫公开密钥加密）。非对称加密需要使用两个密钥，公开密钥（public key）和私有密钥（private key）。公开密钥是公开的，所有人都知道。私有密钥是保密的，除了自己，不让任何人知道。  
 使用公开密钥加密的数据，只有使用私有密钥才能解密。  
 使用私有密钥加密的数据，只有使用公开密钥才能解密。

#### 2.完整性保护



在客户端用对称加密加密数据，对数据进行摘要计算，用非对称加密加密摘要，可以防止篡改，将加密后的数据和摘要发到服务器，用对称秘钥解密数据，并对数据进行摘要生成，跟用非对称秘钥解密的摘要对比，看数据是否是否被篡改。

上诉存在的问题是，服务器用私钥加密，而客户端用服务器的公钥解密，怎么确保别人不会用服务端的公钥解密呢？还有服务端的公钥是如何发给客户端的呢。

就是利用公钥私钥加密解密摘要，通过共享秘钥加密的字符串再次计算摘要，看二者是否相等。

这里的公钥和共享秘钥都是认证过程中获得的。

#### 3.认证

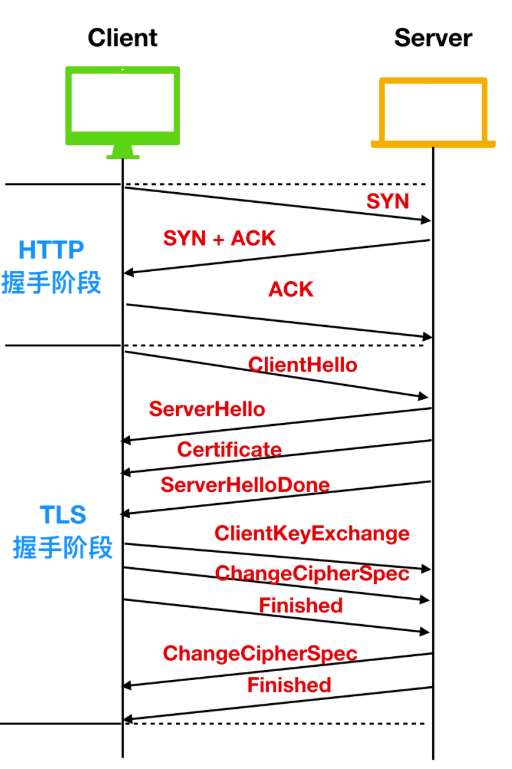
##### SSL协议大白话：

HTTPS引入了权威的第三方机构来确保这个公钥确实是该服务器的。  
 如果要使用HTTPS，**服务器管理人员需要向CA（权威的证书颁发机构）购买证书**。**CA会将该服务器的域名、公钥、公司信息等内容封装到证书中**。

**并且使用CA自己的私钥对证书进行签名。**那么，服务器将证书发给客户端，客户端如果验证出证书是有效的。

问题又来了，客户端如何验证服务器发过来证书是有效的呢？  
 证书上有CA用其私钥做的签名，而**一般客户端都会内置这些权威机构（CA）的公钥的**，所以能够直接拿到CA的公钥对证书上的签名进行解密，然后自己根据证书上的说明计算摘要，两个摘要一致的话，代表证书是有效。因为只有CA自己才有私钥，别人是不可能冒充这个签名的。

##### HTTPS握手过程（七次握手）：



**TLS握手阶段：**

1. 客户端向服务端发送 Client Hello 消息，其中携带客户端支持的协议版本、加密算法、压缩算法以及客户端生成的随机数；

2、服务端收到客户端支持的协议版本、加密算法等信息后；

A向客户端发送 Server Hello 消息，并携带选择特定的协议版本、加密方法、会话 ID 以及服务端生成的随机数；

B向客户端发送 Certificate 消息，即服务端的证书链，其中包含证书支持的域名、发行方和有效期等信息；

C向客户端发送 Server Key Exchange 消息，传递公钥以及签名等信息；

D向客户端发送可选的消息 CertificateRequest，验证客户端的证书；

E向客户端发送 Server Hello Done 消息，通知服务端已经发送了全部的相关信息；

(CD两步只有当客户端要验证服务器的身份时才会用到）？

3、客户端收到服务端的协议版本、加密方法、会话 ID 以及证书等信息后，验证服务端的证书；

A向服务端发送 Client Key Exchange 消息，包含使用服务端公钥加密后的随机字符串，即预主密钥（Pre Master Secret）；

B向服务端发送 Change Cipher Spec 消息，通知服务端后面的数据段会加密传输；

C向服务端发送 Finished 消息，其中包含加密后的握手信息；

4、服务端收到 Change Cipher Spec 和 Finished 消息后；

A向客户端发送 Change Cipher Spec 消息，通知客户端后面的数据段会加密传输；

B向客户端发送 Finished 消息，验证客户端的 Finished 消息并完成 TLS 握手；

### 9.HTTP2.0

**不同之处**

1、相比http1协议的文本传输格式，http2是以二进制（帧）的格式进行数据传输的，具有更小的传输体积以及负载。

2、http2.0分层，分为

a、帧头（http2多路复用能力的核心部分),

b、数据正文（包含传统上被认为是 HTTP 及其关联数据的部分）。

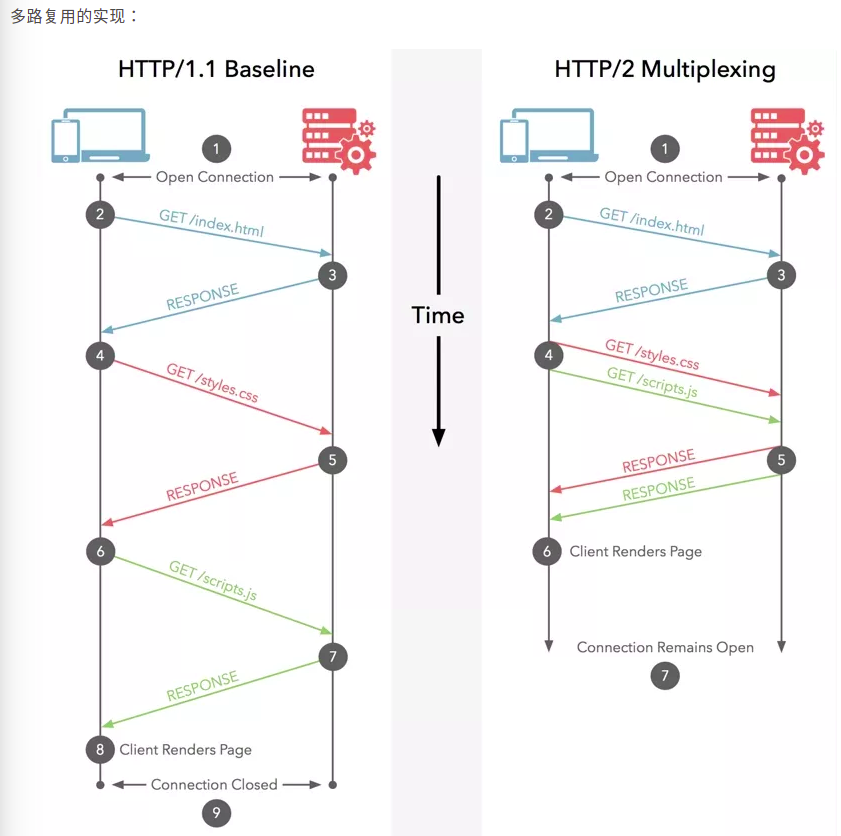
而http1.0分为请求行，请求头和请求正文

**优势**

1、多路复用机制，引入了二进制的分帧层机制来实现多路复用。（分帧层是基于帧的二进制协议，可以将帧）

2、可以设置请求的优先级（客户端的分帧层对分割块标上请求的优先级）。

3、头部压缩 请求头压缩，增加传输效率。



在单个域名下仍可以建立一个TCP管道，使用一个TCP长连接，下载整个资源页面，只需要一次慢启动，并且避免了竞态，浏览器发起请求，分帧层会对每个请求进行分割，将同一个请求的分割块打上相同的id编号，然后通过协议栈将所有的分割体发送给服务器，然后通过服务器的分帧层根据id编号进行请求组装，服务器的分帧层将回应数据分割按同一个回应体进行ID分割回应给客户端，客户端拼装回应。

**HTTP帧结构**

对于http2中的帧（frame），http1不是基于帧（frame）的，是文本分隔的。

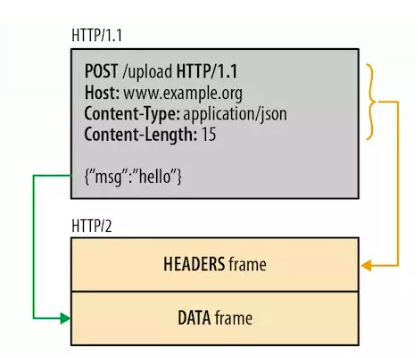


这样，对于http1的请求或者是响应可能有的问题：

1、一次只能处理一个请求或者是响应，完成之前是不能停止解析的。

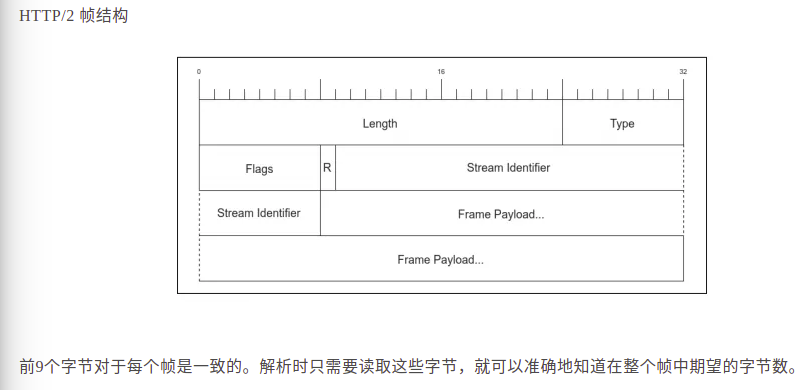
2、无法预判解析需要多少内层。

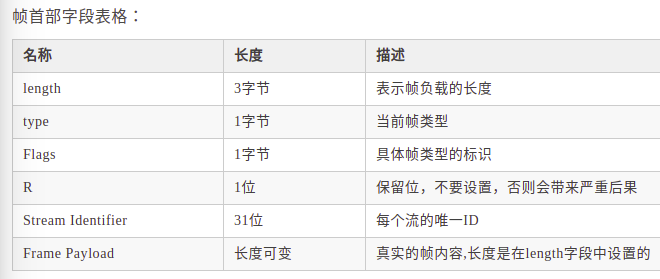
HTTP/1 的请求和响应报文，是由起始行、首部和正文组成，换行符分隔；HTTP/2是将请求和响应数据分割成更小的帧，采用二进制编码，易于解析的。





帧结构总结 所有的帧都包含一个9 byte的帧头 + 可变长的正文部分。根据帧的类型不同，正文部分的结构也不一样。

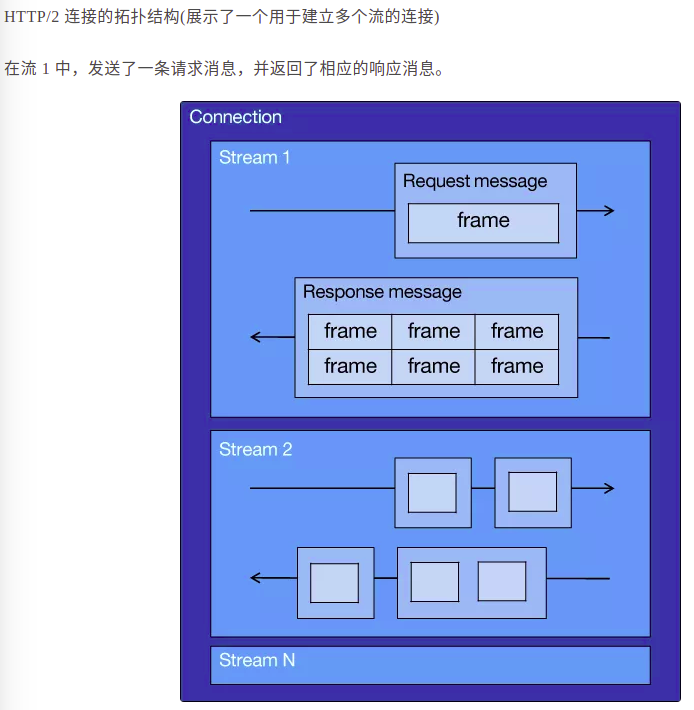




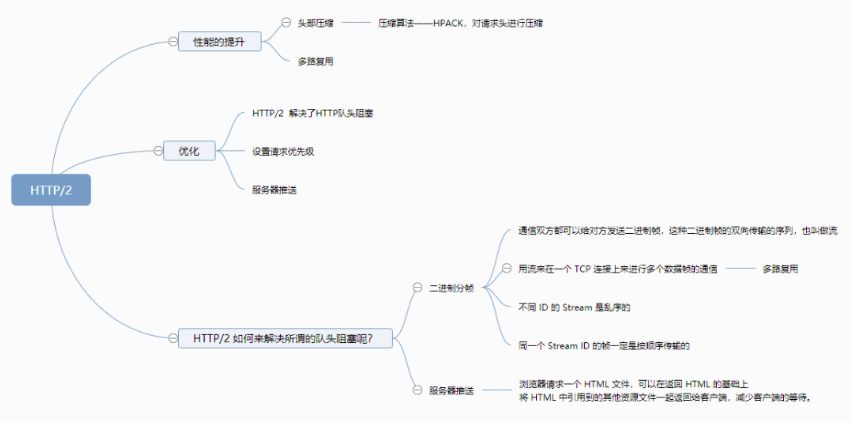
备注：流Id是用来标识帧所属的流。流看作在连接上的一系列帧，它们构成了单独的 HTTP 请求和响应。



**HTTP2连接拓扑结构**



流是多路复用的体现，因为划分成帧，那么请求报文和响应报文可以复用一个流。



# UDP

UPD面向报文的协议，就是UDP只是报文的搬运工，不会对报文进行任何拆分和拼接操作；

在发送端，应用层将数据传给传输层的UDP协议，UDP会给数据加一个UDP头标识下是UUDP协议，然后传给网络层。

在接收端，网络层将数据传给传输层，UDP只去除IP报文头就传给应用层，不会任何拼接操作。

UDP是无连接，通信不需要建立和断开连接，UDP是不可靠的，不关心数据的安全等问题，UDP是没有拥塞控制，在网络条件不好的情况下可能会导致丢包。

传输：UDP 支持一对一，一对多，多对多，多对一的的传输方式， UDP 提供了单播，多播，广播的功能

# 安全问题

1. **CSRF：**

CSRF的基本概念，攻击原理，防御措施

CSRF（Cross-site request forgery）：跨站请求伪造

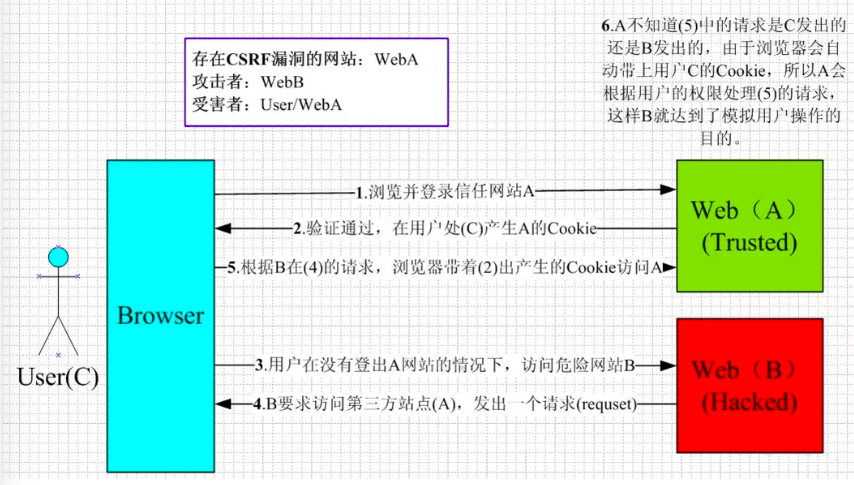
理解CSRF攻击：攻击者盗用了你的身份，以你的名义发送恶意请求。

以你名义发送邮件，发消息，盗取你的账号，甚至于购买商品，虚拟货币转账……造成的问题包括：个人隐私泄露以及财产安全。

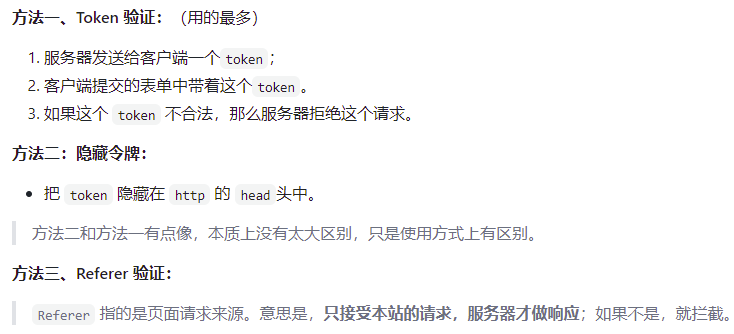
CSRF的原理:(要完成一次CSRF攻击)

登录受信任网站A，并在本地生成Cookie。

在不登出A的情况下，访问危险网站B。



解决：



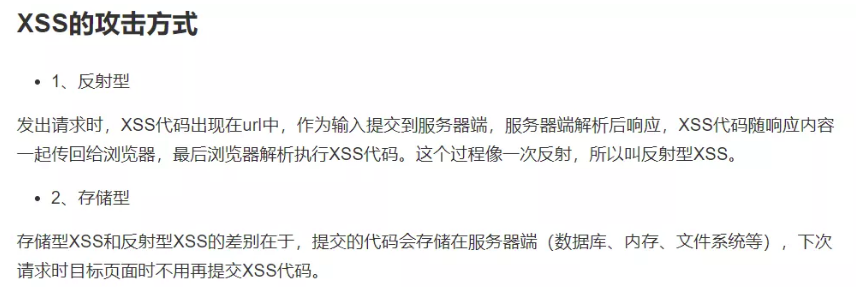
1. **XSS**

XSS的基本概念,跨域脚本攻击。

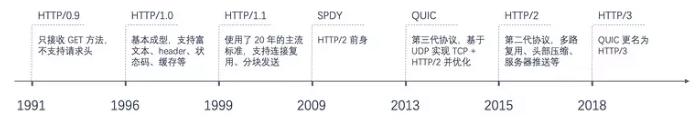
xss是一种发生在web前端的漏洞，所以其危害的对象也主要是前端用户。

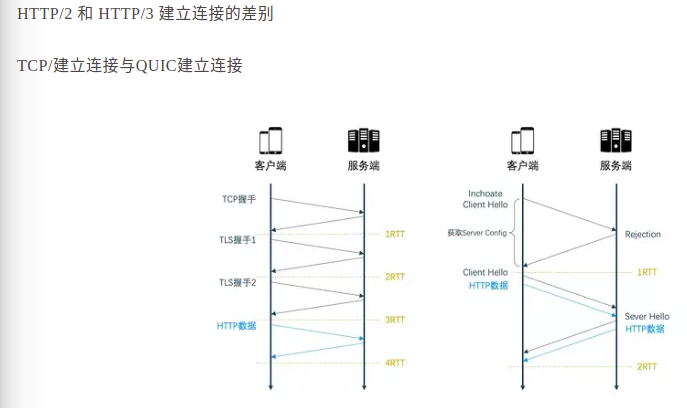
跨域脚本攻击是，恶意攻击者往web页面里插入恶意的script代码，在浏览器中运行script代码，达到恶意攻击用户的目的。

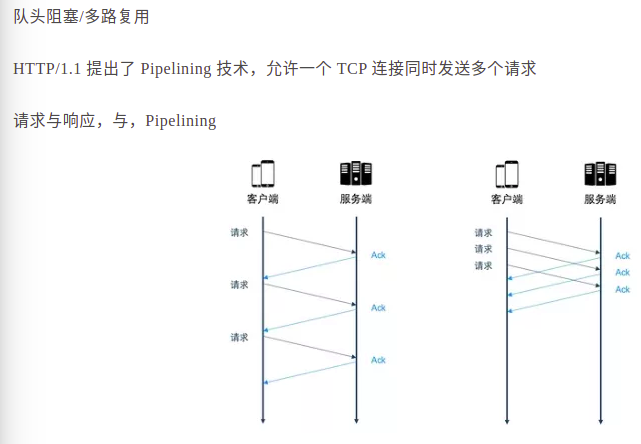
so，实现xss攻击具备2个条件，第一需要向web页面注入恶意的代码，第二，这些恶意代码被浏览器成功的执行。

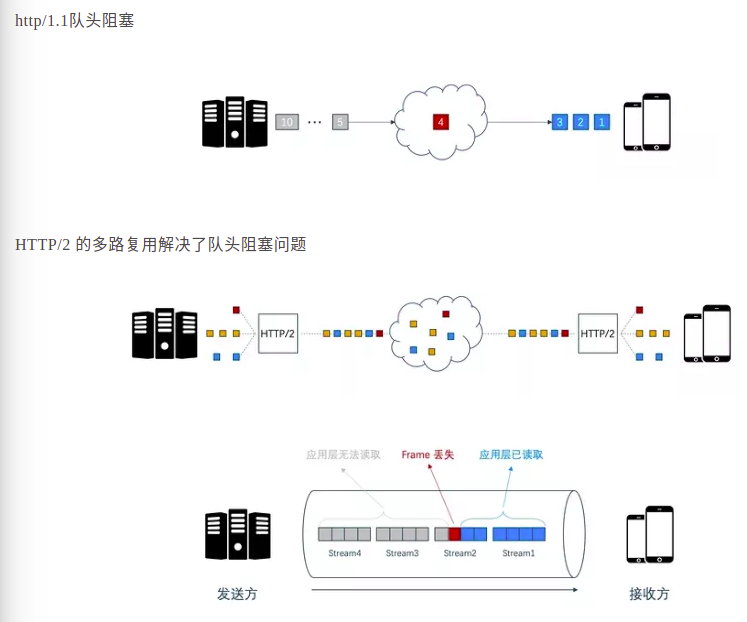


# HTTP历史





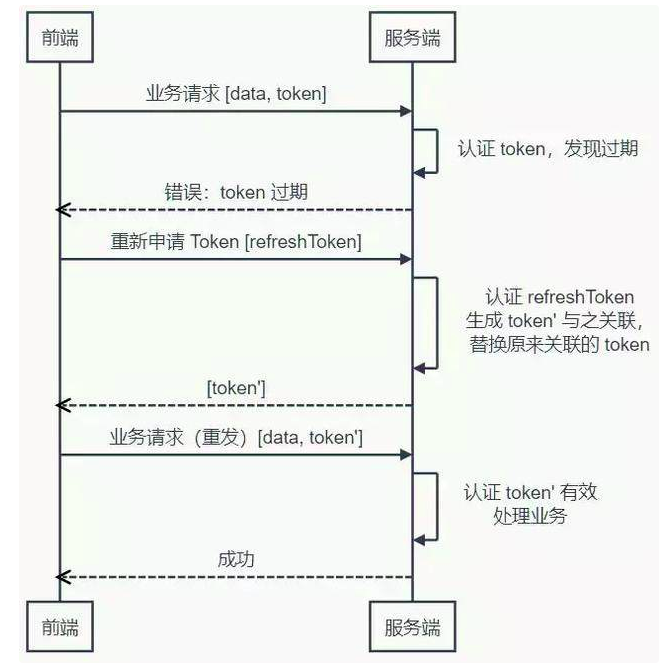
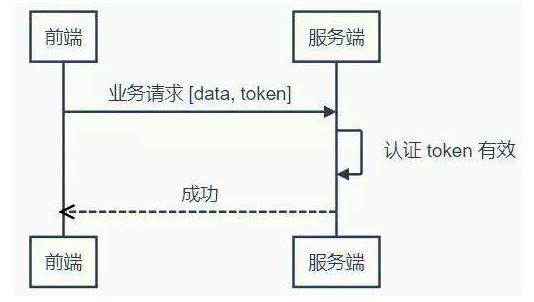
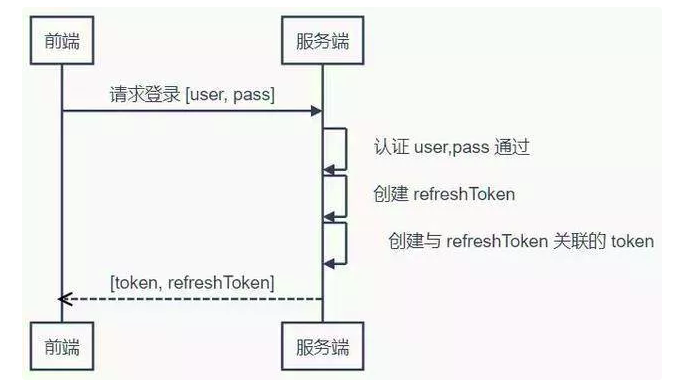




# Token

token的出现，是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示。token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，第一登录时，服务器生成一个token，将此token返回给客户端，客户端带上这个token，无需再次带上用户名和密码了。

token的出现减轻了服务器的压力，减少频繁地数据库查询。





cookie，记录访问过的网站或正在访问的网站，对于HTTP 协议是无状态的，服务器不知道浏览器上一次访问做了什么，也无法对用户会话进行跟踪连接，所以，cookie是由服务器发送到客户端浏览器的一段文本文件，包含了网站访问活动信息。Cookie 存放在客户端，用来保存客户端会话信息；由于存储在客户端，它的安全性不能完全保证。

session表示是c/s架构中服务器和客户端一次会话的过程，用来保存认证用户信息。session是一种HTTP存储机制，提供持久机制。Session存放在服务器端，用户验证客户端信息。由于存储在服务器，安全性有一定的保证。

token是一种认证方式(是“令牌”的意思，主要是用于身份的验证方式。)

# TCP/IP协议

1. 应用层决定了向用户提供应用服务时通信的活动。
2. 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。
3. 网络层用来处理在网络上流动的数据包。
4. 链路层，用来处理连接网络的硬件部分。

HTTP协议的职责，生成对目标web服务器的HTTP请求报文

tcp协议的职责，为了方便通信，将HTTP请求报文分割成报文段

IP协议的职责，搜索对方的地址，一边中转一边传送

TCP协议的职责，从对方那里接收到的报文段，重组到达的报文段，按序号以原来的顺序重组请求报文