一、TCP、UDP、HTTP

1.TCP 和 UDP 区别

	TCP	UDP
可靠性	可靠	不可靠
连接性	面向连接	无连接
报文	面向字节流	面向报文（保留报文的边界）
效率	传输效率低	传输效率高
双工性	全双工	一对一、一对多、多对一、多对多
流量控制	有（滑动窗口）	无
拥塞控制	有（慢开始、拥塞避免、快重传、快恢复）	如果要快的话就用UDP，准确性高的话就要求TCP 无
传输速度	慢	快
应用场合	对效率要求相对低，但对准确性要求相对高，或者是要求有连接的场景	对效率要求相对高，对准确性要求相对低的场景
应用示例	TCP一般用于文件传输（FTP http 对数据准确性要求高，速度可以相对慢），发送或接收邮件（pop imap SMTP 对数据准确性要求高，非紧急应用），远程登录（telnet SSH 对数据准确性有一定要求，有连接的概念）等等；	UDP一般用于即时通信（QQ聊天 对数据准确性和丢包要求比较低，但速度必须快），在线视频（rtsp 速度一定要快，保证视频连续，但是偶尔花了一个图像帧，人们还是能接受的），网络语音电话（VoIP 语音数据包一般比较小，需要高速发送，偶尔断音或串音也没有问题）等等。

1. 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。

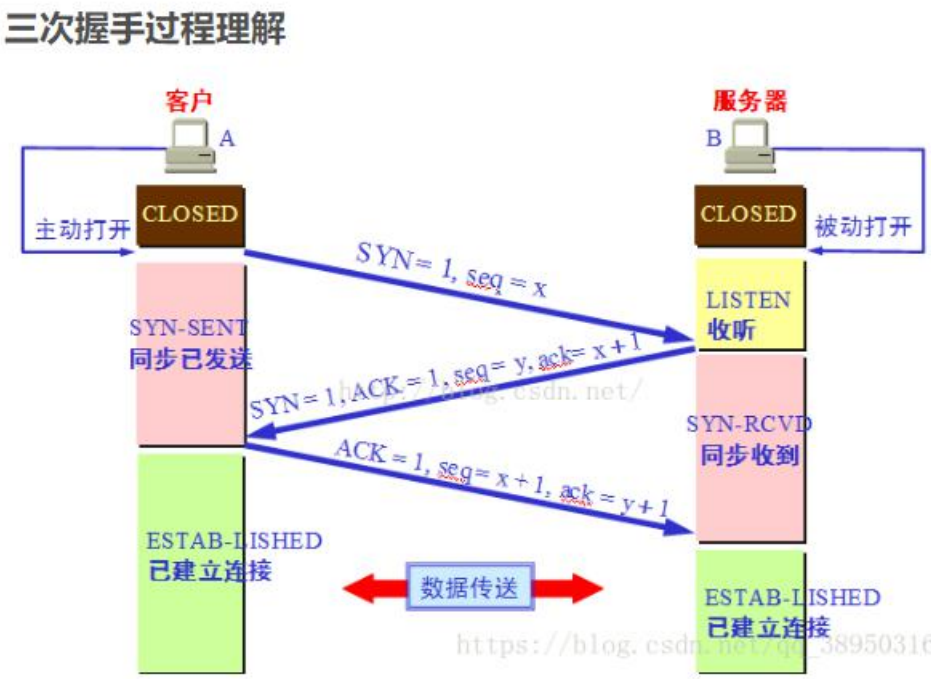
譬如发起请求遇到类似这样的情况 :客户端发出去的第一个连接请求由于某些原因在网络节点中滞留了导致延迟，直到连接释放的某个时间点才到达服务端，**这是一个早已失效的报文，**但是此时服务端仍然认为这是客户端的建立连接请求第一次握手，于是服务端回应了客户端，第二次握手。

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；序列号 seq 表示报文段携带数据的第一个字节的编号；而确认号指的是期望接收到下一个字节的编号；因此当前报文段最后一个字节的编号+1 即为确认号。

确认 ACK：占 1 位，仅当 ACK=1 时，确认号字段才有效。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

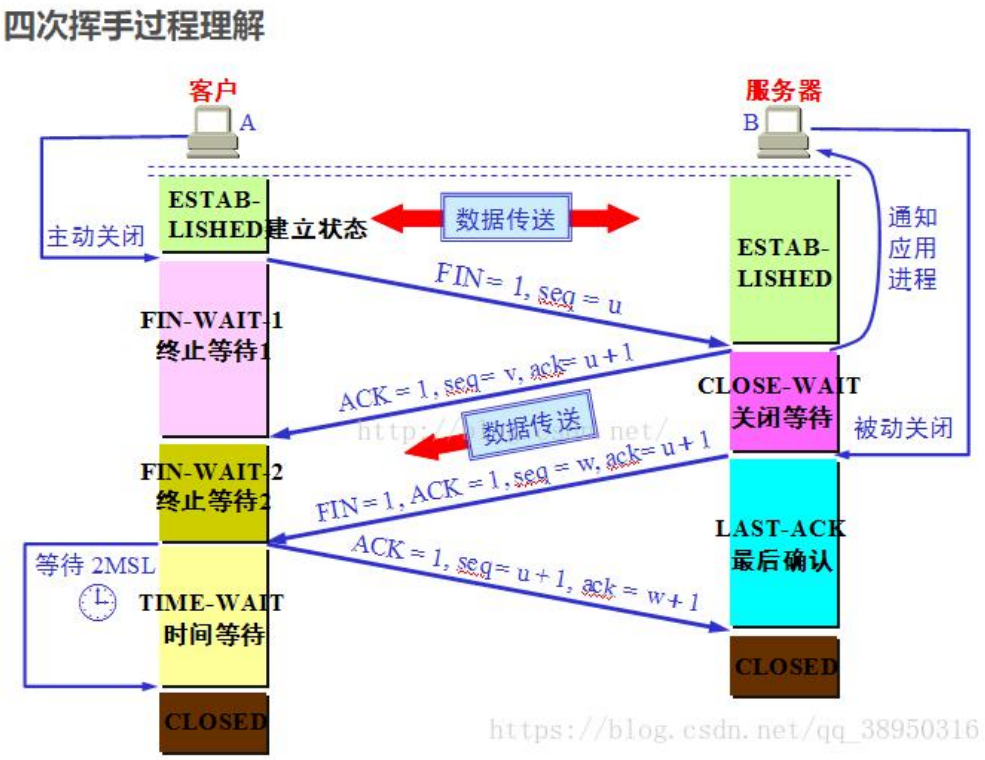
PS：ACK、SYN 和 FIN 这些大写的单词表示标志位，其值要么是 1，要么是 0；ack、seq 小写的单词表示序号。

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。

2. 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。**拥塞控制所要做的都有一个**前提：网络能够承受现有的网络负荷。**拥塞控制是一个**全局性的过程**，涉及到所有的主机、路由器，以及与降低网络传输性能有关的所有因素

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)

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。

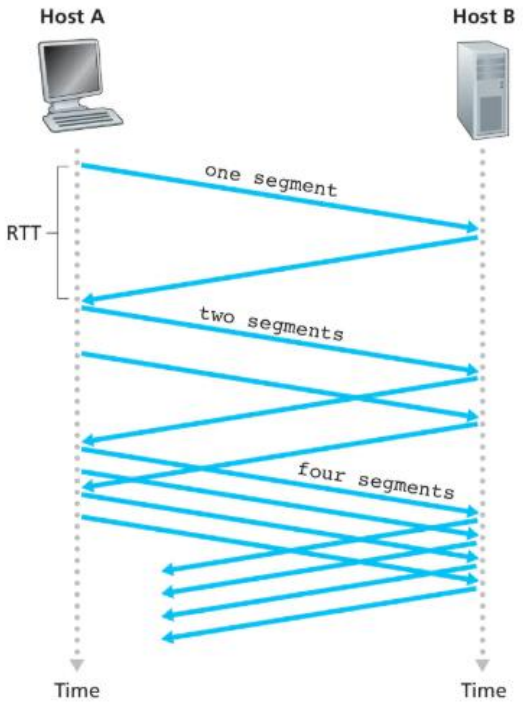


Figure 3.51 ♦ TCP slow start

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

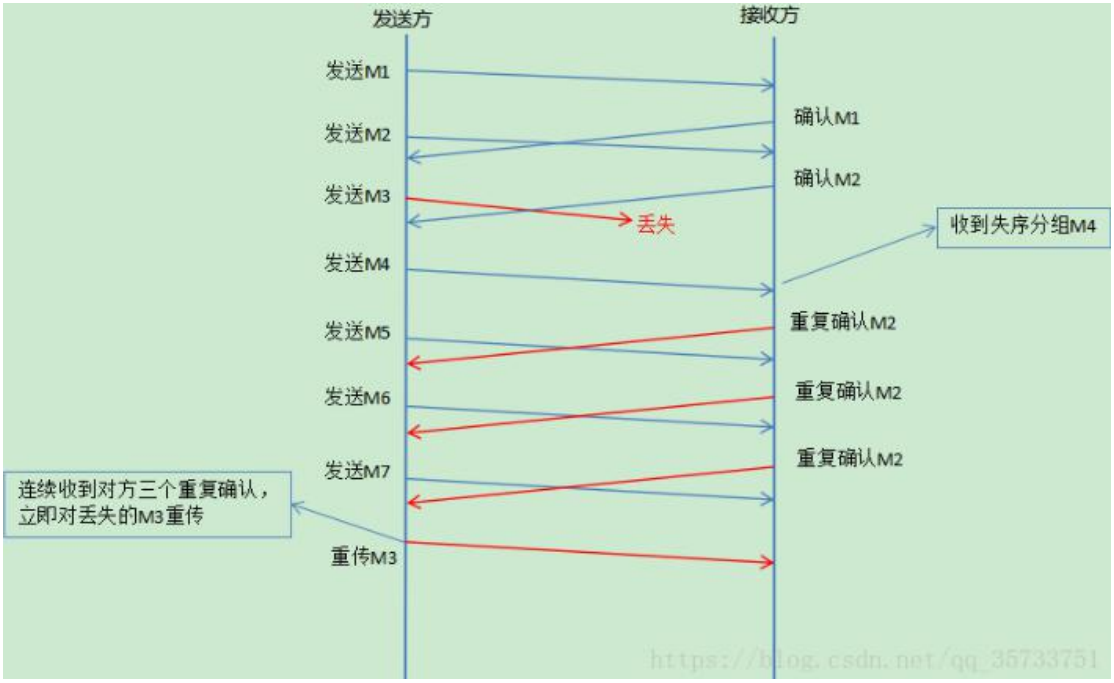
考虑这么一种情况：假设现在网络没有发生拥塞，但是发送方发送的个别数据包却在网络中某一处丢失了，而此时发送方的超时计时器超时了，又没有收到确认。

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



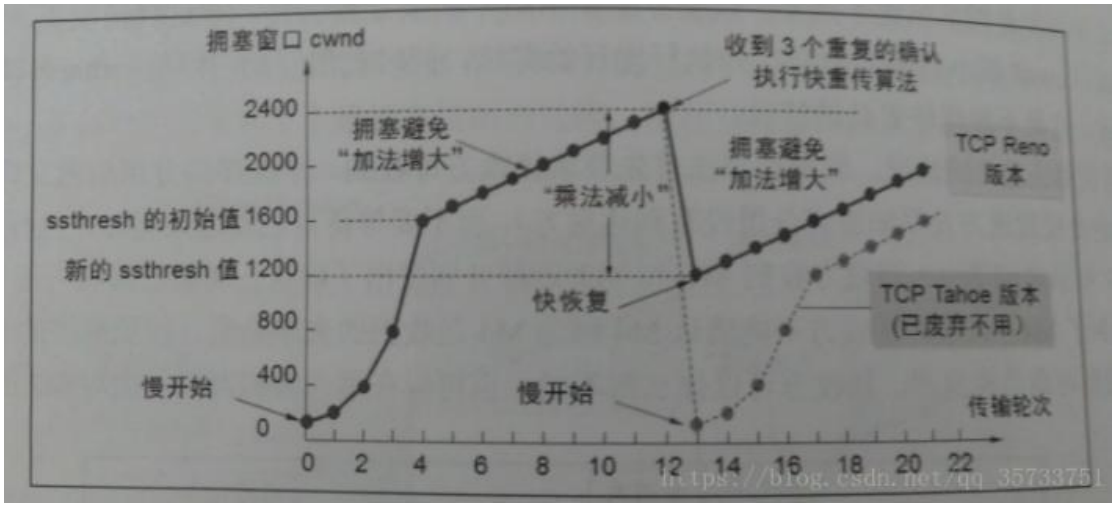
规定：接收方收到失序的 M4 分组后，应该立即对 M2 发出重复确认（总共发送三次重复确认），也就是告诉发送方：“我还没收到 M3 分组”。

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，知道达到慢启动上限 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）

1. 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.

2. 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

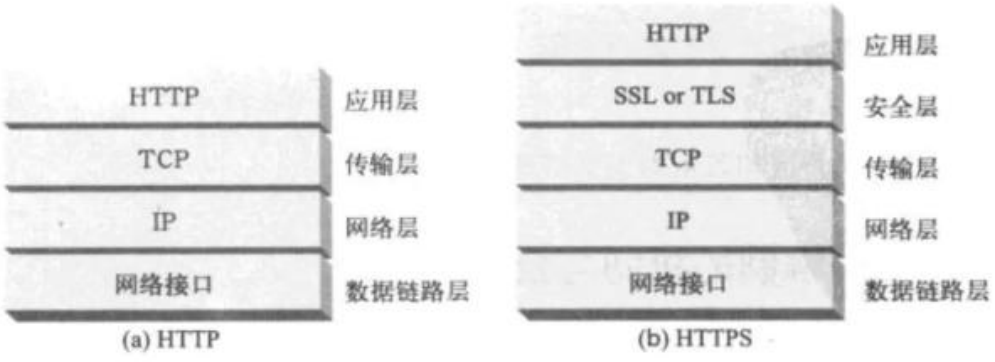
HTTP 协议通常承载于 TCP 协议之上，有时也承载于 TLS 或 SSL 协议层之上，这个时候，就成了我们常说的 HTTPS。默认 HTTP 的端口号为 80， HTTPS 的端口号为 443。

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。



HTTPS是位于安全层之上的HTTP，这个安全层位于TCP之上

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：

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

3. 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.

4. 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。

5.

5. 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。

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

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。

2、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。

3、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

4、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 是一个顶级域名 ,而 www.baidu.com 和 map.baidu.com 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。

图 15-4　HTTP 请求报文

6. 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

分类 分类描述

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

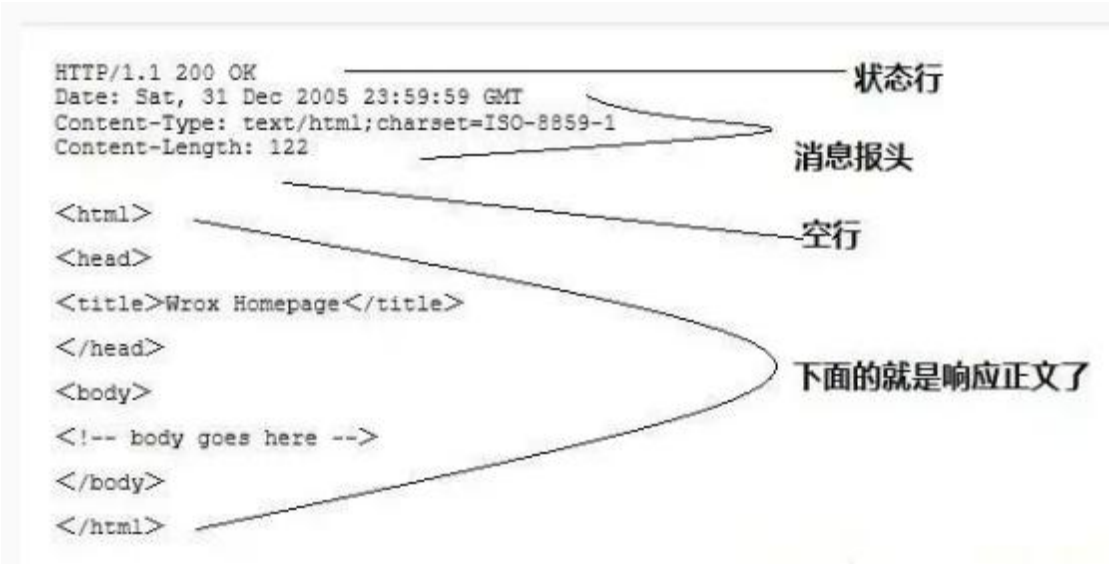
用于重定向一个新的位置，包含新的 URL 地址。表示客户应当到哪里去提取文档。

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。



7. 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.

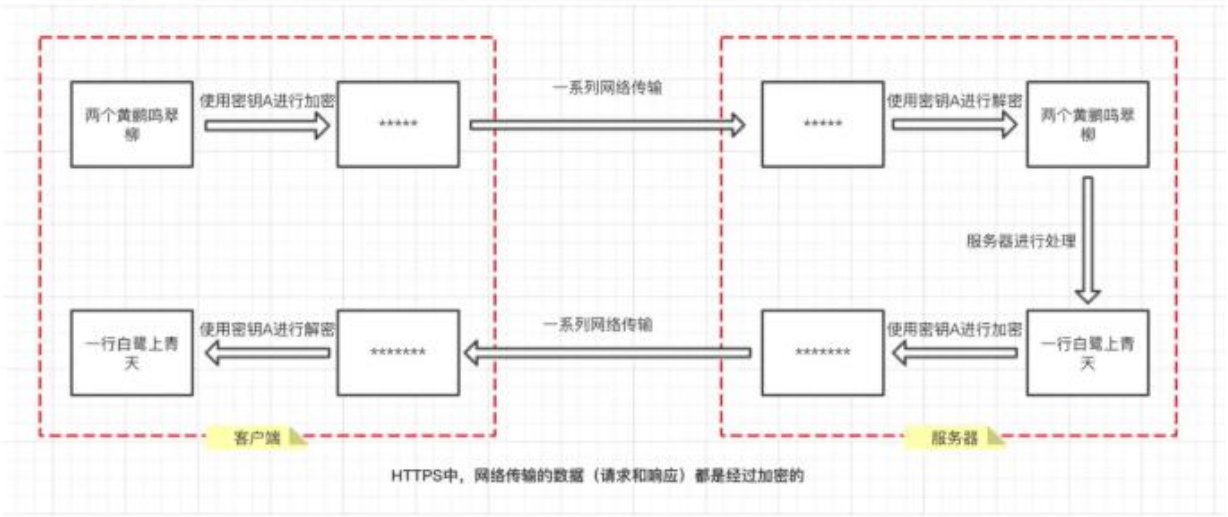
4.释放连接 TCP 连接

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



HTTPS中，网络传输的数据（请求和响应）都是经过加密的

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

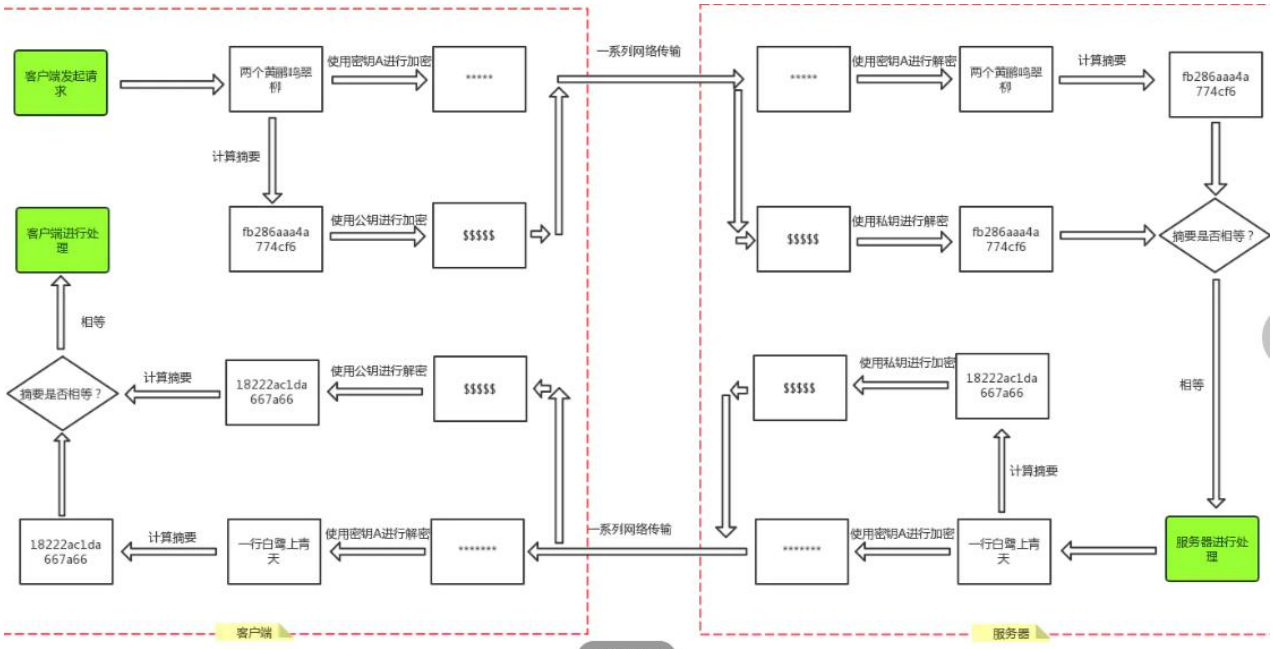
拥有一个密钥）。使用密钥 A 加密的内容，只能用密钥 A 来解密，其他的密钥都无法解密。常用的对称加密算法有 DES,3DES,AES。

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。

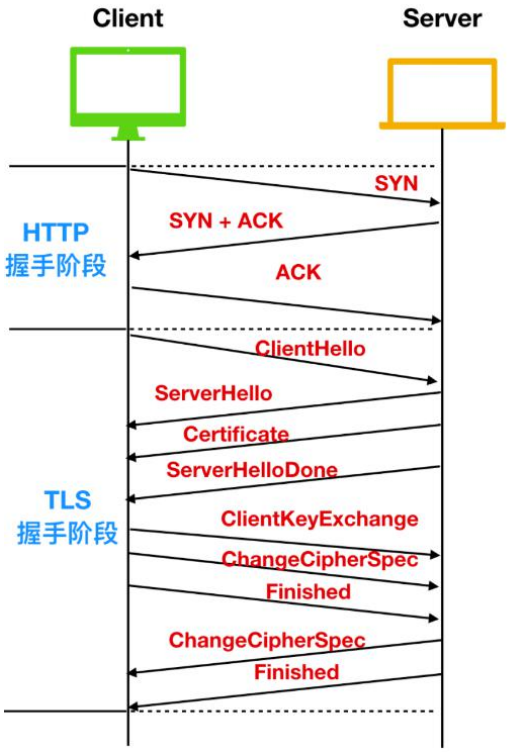
如果要使用 HTTPS，**服务器管理人员需要向 CA（权威的证书颁发机构）购买证书。CA 会将该服务器的域名、公钥、公司信息等内容封装到证书中。**

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；