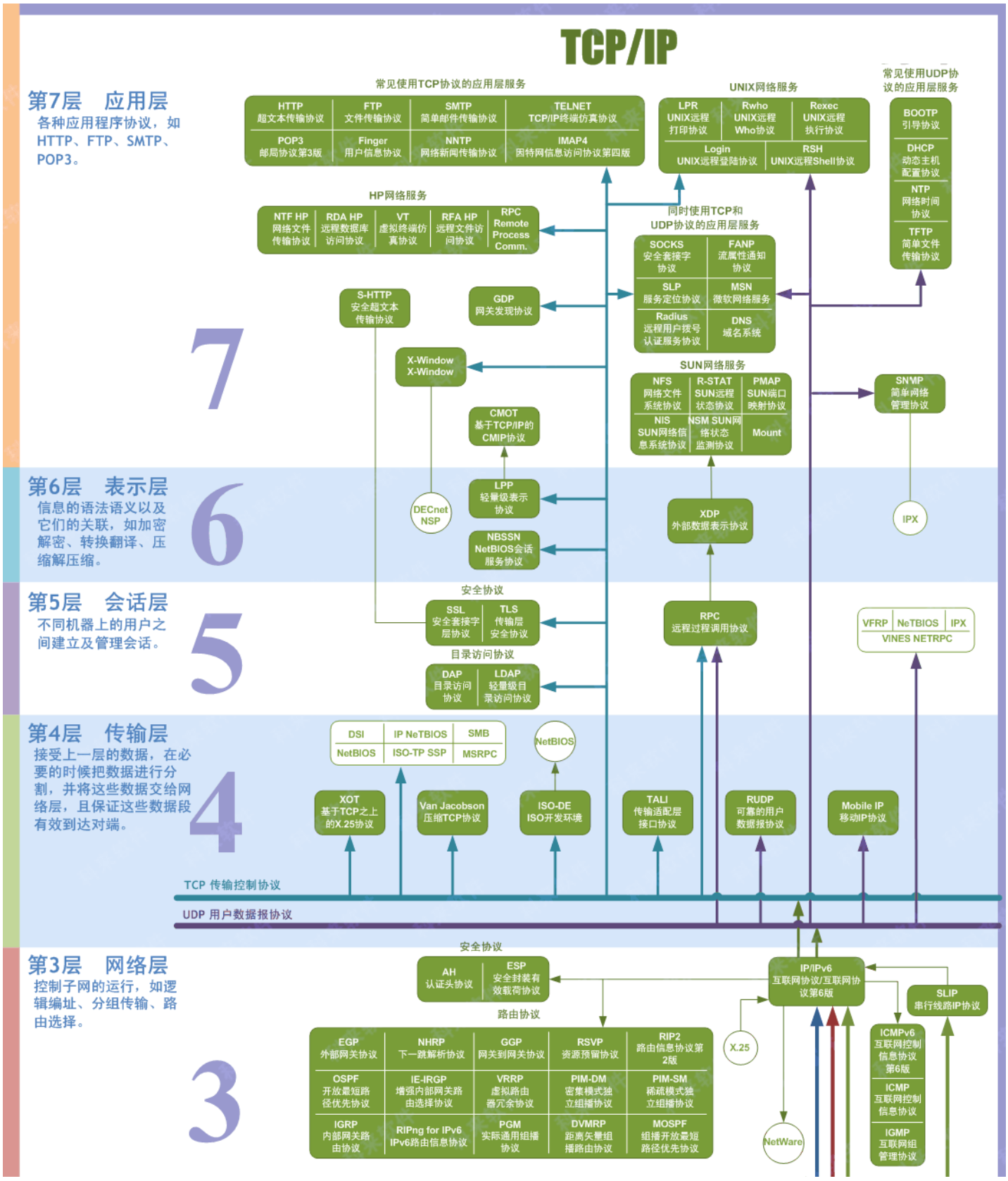
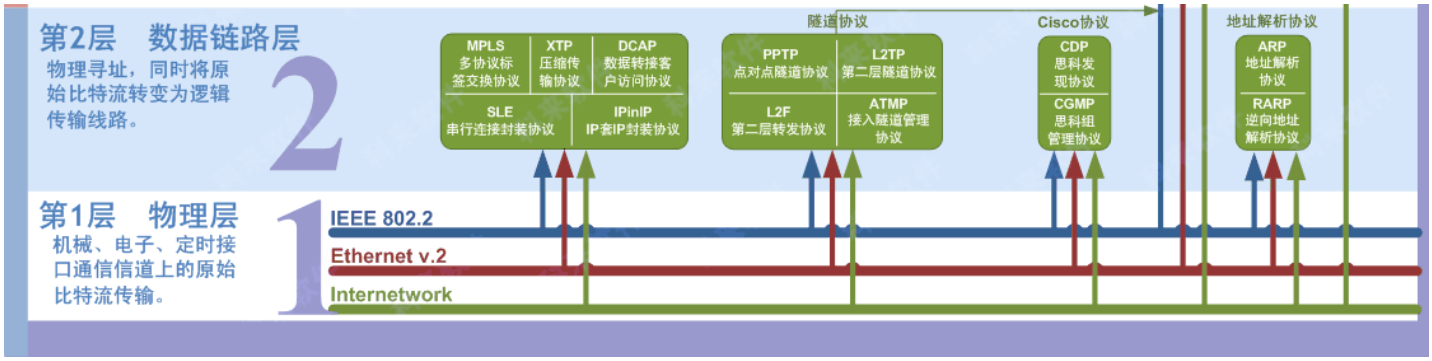
computer network

OSI 七层模型





传输层 Transport layer

- 传输层提供应用层提供端到端通信服务
- 有IP地址，通过ip地址能找到不同网络下的网络，结合mac地址就能找到对应主机，那么怎么找到主机应用进程呢，肯定也有一个东西来标识它，那就是我们常说的端口了
- 端口，占有16位，其大小也就有65536个，是从0~65535。
- 常用端口号 http:80 ssh:22 scp 21or23 https:443

端口分类

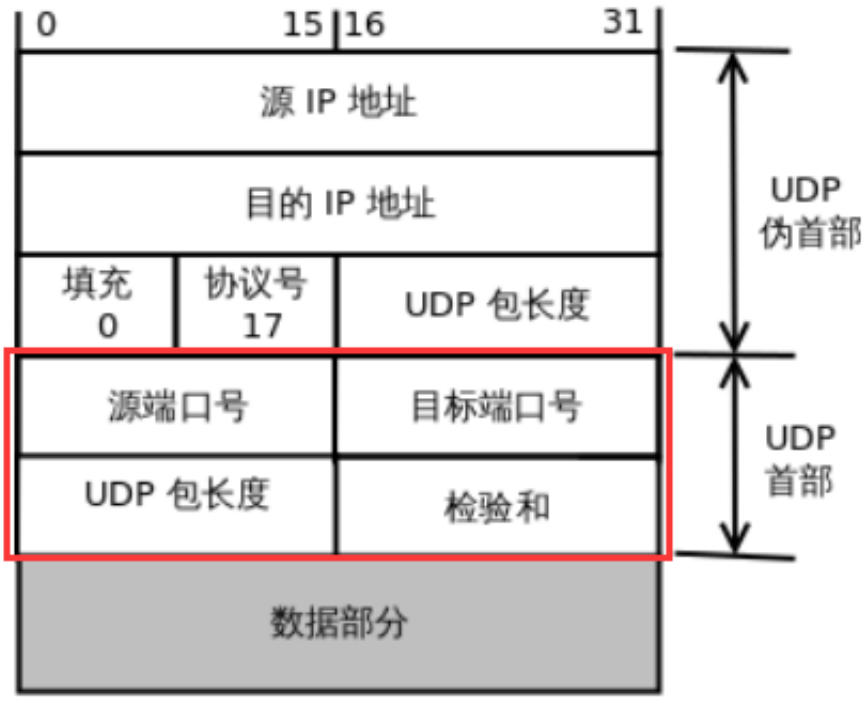
- 熟知端口：0-1023， 也就是一些固定的端口号，比如http使用的80端口，意思就是在访问网址时，我们访问服务器的端口就是80，然后服务器那边传网页的数据给我们。
- 登记端口：1024-49151，比如微软开发了一个系统应用，该应用在通讯或使用时，需要使用到xxx端口，那么就要去登记一下这个端口，以免有别人公司的应用使用同一个端口号，例如，windows系统中的3389端口，就是用来实现远程连接的，就固定了这台计算机如果要使用远程连接服务，就打开3389端口，别人就能使用远程连接连你了，默认是不打开的
- 客户端端口：49152-65535，一般我们使用某个软件，比如QQ，等其他服务，随机拿这个范围内的端口，而不是去拿前面哪些固定的，拿到等通讯结束后，就会释放该端口。

UDP协议

- 无连接，不可靠。

- 无连接：意思就是在通讯之前不需要建立连接，直接传输数据。
- 不可靠：是将数据报的分组从一台主机发送到另一台主机，但并不保证数据报能够到达另一端，任何必须的可靠性都由应用程序提供。在 UDP 情况下，虽然可以确保发送消息的大小，却不能保证消息一定会达到目的端。没有超时和重传功能，当 UDP 数据封装到 IP 数据报传输时，如果丢失，会发送一个 ICMP 差错报文给源主机。即使出现网络阻塞情况，UDP 也无法进行流量控制。此外，传输途中即使出现丢包，UDP 也不负责重发，甚至当出现包的到达顺序杂乱也没有纠正的功能

- UDP首部格式



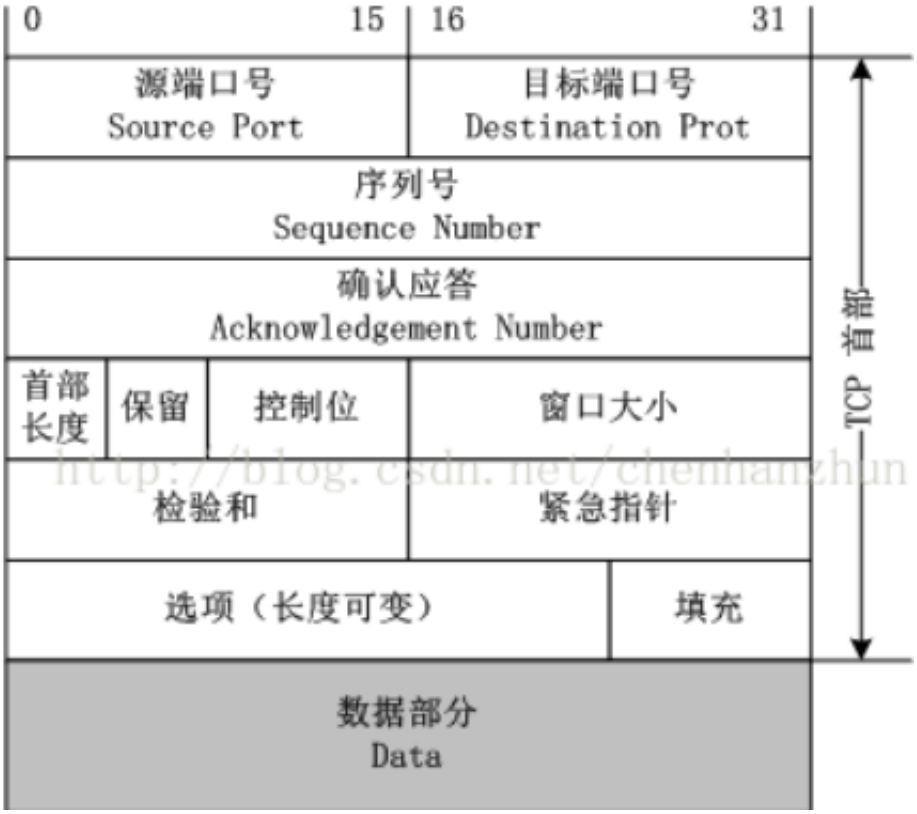
- 图 2 UDP检验和计算中使用的UDP伪首部
- 源端口号：占16位，源主机的应用进程所使用的端口号
- 目标端口号：占16位，目标主机的应用进程所使用的端口号，也就是我们需要通信的目标进程
- UDP报长度：UDP用户数据报的长度，数据部分+UDP首部之和为UDP报长度
- 检验和：检验和是为了提供可靠的 UDP 首部和数据而设计，这里不要和上面的不可靠传输搞混淆了，这里提供可靠的UDP首部，是因为一个进程可能接受多个进程过来的报文，那么如何区分他们呢，就是通过5个东西来进行区分的，“源 IP 地址”、“目的 IP 地址”、“协议号”、“源端口号”、“目标端口号”的，这个检测可靠，是检测接受哪个正确的报文，也就是说是哪个报文要进这个端口，那个不可靠，说的是这个报文可能丢失，可能其中数据损坏了我们不关心，但是这些的前提是，你得传输到正确的目的地去，不然乱出乱发数据报，岂不是乱套了。
- UDP伪首部 就是拿到IP层的一些数据，因为要进行检验和，就必须要有这些数据。其中检验的算法跟IP层中检验首部的办法是一样的。

使用udp协议的例子

- 应用层协议中DNS，也就是根据域名解析ip地址的一个协议，他使用的就是UDP
- DHCP,这个是给各电脑分配ip地址的协议，其中用的也是UDP协议
- IGMP，我们说的多播，也就是使用的UDP，在多媒体教室，老师拿笔记本讲课，我们在下面通过各自的电脑看到老师的画面，这就是通过UDP传输数据，所以会出现有的同学卡，有的同学很流畅，就是因为其不可靠传输，但是卡一下，对接下来的观看并没有什么映像

TCP 协议

- TCP协议是面向连接的、可靠传输、有流量控制，拥塞控制，面向字节流传输等很多优点的协议。其最终功能和UDP一样，在端和端之间进行通信，但是和UDP的区别还是很大的。



- TCP报文的结构：
- - 源端口号
- - 目标端口号
- - 序列号：因为在TCP是面向字节流的，他会将报文度分成一个个字节，给每个字节进行序号编写，比如一个报文有900个字节组成，那么就会编成1-900个序号，然后分几部分来进行传输，比如第一次传，序列号就是1，传了50个字节， 那么第二次传，序列号就为51，所以序列号就是传输的数据的第一个字节相对所有的字节的位置
- - 确认应答：如刚说的例子，第一次传了50个字节给对方，对方也会回应你，其中带有确认应答，就是告诉你下一次要传第51个字节来了，所以这个确认应答就是告诉对方要传第多少个字节了
- - 首部长度：就是首部的长度
- - 保留：给以后有需要在用，这个保留的位置放的东西是跟控制位类似的
- - 控制位：目前有的控制位为6个

URG:紧急，当URG为1时，表名紧急指针字段有效，标识该报文是一个紧急报文，传送到目标主机后，不用排队，应该让该报文尽量往下排，让其早点让应用程序给接受

ACK:确认，当ACK为1时，确认序号才有效。当ACK为0时，确认序号没用

PSH：推送，当为1时，当遇到此报文时，会减少数据向上交付，本来想应用进程交付数据是要等到一定的缓存大小才发送的，但是遇到它，就不用在等足够多的数据才向上交付，而是让应用进程早点拿到此报文，这个要和紧急分清楚，紧急是插队，但是提交缓存大小的数据不变，这个推送就要排队，但是遇到他的时候，会减少交付的缓存数据，提前交付

RST:复位，报文遇到很严重的差错时，比如TCP连接出错等，会将RST置为1，然后释放连接，全部重新来过。

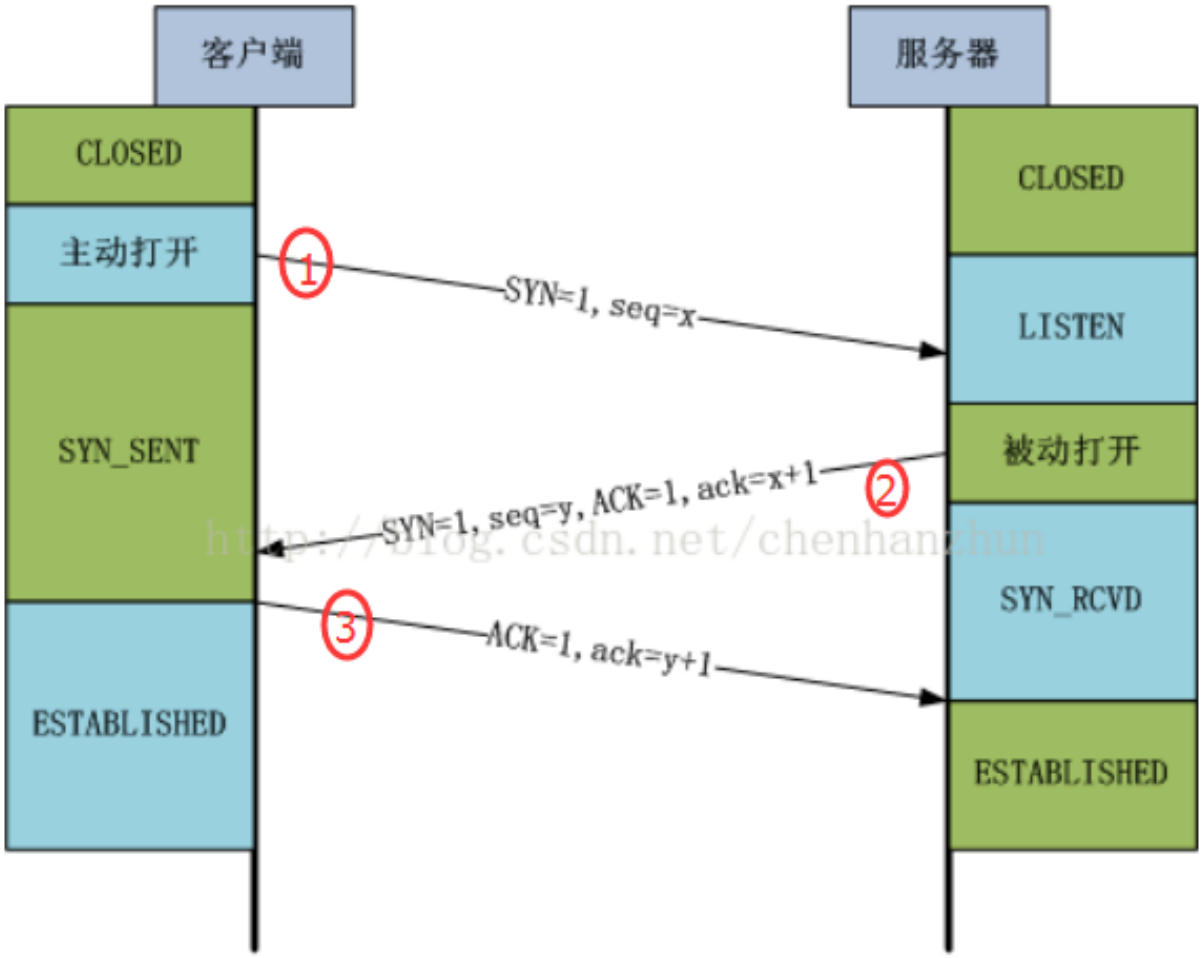
SYN：同步，在进行连接的时候，也就是三次握手时用得到，下面会具体讲到，配合ACK一起使用

FIN：终止，在释放连接时，也就是四次挥手时用的

- 窗口：指发送报文段一方的接受窗口大小，用来控制对方发送的数据量(从确认号开始，允许对方发送的数据量)。也就是后面需要讲的滑动窗口的窗口大小
- 检验和：检验首部和数据这两部分，和UDP一样，需要拿到伪首部中的数据来帮助检测
- 选项：长度可变，介绍一种选项，最大报文段长度，MSS。 能够告诉对方TCP，我的缓存能接受报文段的数据字段的最大长度是MSS个字节。如果没有使用选项，那么首部固定是20个字节。
- 填充：就是为了让其成为整数个字节

面向连接

三次握手



- 一开始客户端和服务端都市关闭状态，但是在某个时刻，客户端需要和服务端进行通信，此时双方都会各自准备好端口，服务器段的端口会处于监听状态，等待客户端的连接。客户端可会知道自己的端口号，和目的进程的端口号，这样才能发起请求。

- 第一次握手：客户端想与服务器进行连接了，所以状态变为主动打开，同时发送一个连接请求报文给服务器段SYN=1，并且会携带x个字节过去。发送完请求连接报文后，客户端的状态就变为了SYN_SENT，可以说这个状态是等待发送确认(为了发送第三次握手时的确认包)

- 第二次握手：服务端接收到连接请求报文后，从LSTTEN状态变为被动打开状态，然后给客户端返回一个报文。这个报文有两层意思，一是确认报文，而可以达到告诉客户端，我也打开连接了。发完后，变为SYN_RCVD状态(也可以说是等待接受确认状态，接受客户端发过来的确认包)

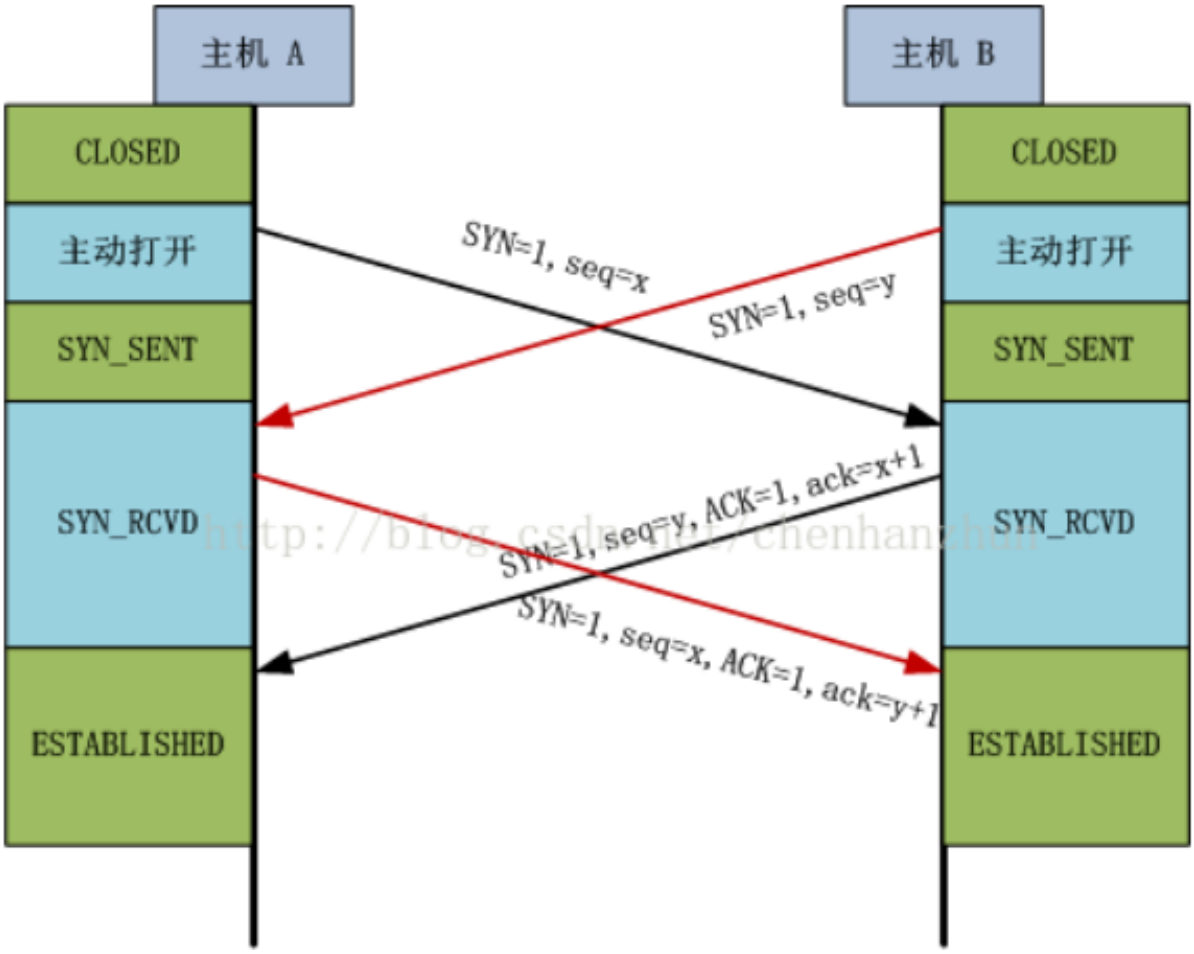
- 第三次握手：客户端得到服务器端的确认和知道服务器端也已经准备好了连接后，还会发一个确认报文到服务器端，告诉服务器端，我接到了你发送的报文，接下来就让我们两个进行连接了。客户端发送完确认报文后，进入ESTABLISHED，而服务器接到了，也变为ESTABLISHED

  进入到ESTABLISHED状态后，连接就已经完成了，可以进行通信了。

- **question：为什么需要第三次握手，有前面两次不就已经可以了吗?**
- 假设没有第三次握手，客户端发送一个连接请求报文过去，但是因为网络延迟，在等待了一个超时时间后，客户端就会在重新发一个请求连接报文过去，然后正常的进行，服务器端发回一个确认连接报文，然后就开始通讯，通讯结束后，那个第一次因为网络延迟的请求连接报文到了服务器端，服务器端不知道这个报文已经失效，也发回了一个确认连接报文，客户端接收后，发现自己并没有发送连接请求(因为超时了，所以就认为自己没有发)，所以对这个确认连接请求就什么也不做，但是此时服务器端不这么认为，他认为i连接已经建立了，就一直打开着等待客户端传数据过来，这就造成了极大的浪费。如果有了第三次握手，那么客户端就可以通知服务器了。所以第三次握手也很重要

- **同时建立连接**

- 正常情况下，通信一方请求建立连接，另一方响应该请求，但是如果出现，通信双方同时请求建立连接时，则连接建立过程并不是三次握手过程，而且这种情况的连接也只有一条，并不会建立两条连接。同时打开连接时，两边几乎同时发送 SYN，并进入 SYN*SENT 状态，当每一端收到 SYN 时，状态变为 SYN*RCVD，同时双方都再发 SYN 和 ACK 作为对收到的 SYN 进行确认应答。当双方都收到 SYN 及相应的 ACK 时，状态变为 ESTABLISHED



- **可靠传输**
- 通过1、数据编号和积累确认 2、以字节为单位的滑动窗口 3、超时重传时间 4、快速重传 这四个方面来达到可靠传输的目的。
- 1、数据编号：将每个字节进行编号，有900个字节，就从1到900进行编号
- 1、积累确认：服务器端不是接收到一个字节就发一个确认，那样效率太低，而是当接收到4，5个时，在发送一个确认，那么在之前的确认之前的数据就算发送成功了的
- 2、滑动窗口：这个跟在数据链路层讲个滑动窗口一样。每次能发送的数据是在此窗口中的，接到了多少数据，就往后滑多少数据
- 3、超时重传时间：这个也在链路层讲过，如果等待一段时间后，还没接收到确认报文，那么就重新
- 4、快速重传：在滑动窗口中的应用，比如传了1234 6到服务器端，老办法是在4之后的所有数据度要重新传，而这个快速重传就只需要等待传了5这个序号，就可以继续往下接收数据了。
- **流量控制**
- 在传输层中，有接受缓存和发送缓存这两个东西的存在，所以每次发送数据过去另一端时，都会把这些数据给带过去，让对方知道自己的这两个缓存的大小，然后来合理的设置自己的发送窗口的大小，如果

对方的缓存快满了，对方在传送数据过来的时候，就会告诉自己，少发一点数据过来，自己就设置滑动窗口小一点，让对方有缓冲的机会，而不会导致缓存溢出，不让自己的报文被丢弃。

- **拥塞控制**
- 其实跟流量控制差不多，但是站的角度更大，此时既考虑了对方接收不过来，缓存太多溢出导致，又考虑在线路中，线路上的传输速率就那么大，但是有很多人同时用，发送的数据太多，就会使线路发现拥塞，也就是路由器可能转发不过来，导致大量数据丢失，这两个问题。所以拥塞控制这个解决方案，大概意思就是当检测到有网络拥塞时，就会让自己的滑动窗口变小，但具体是怎么变化的，就是根据算法来算了，
- ◦ 发送窗口的上限值 = Min[rwnd，cwnd]

rwnd：接受窗口，根据接受缓存，而定的接受窗口，接收缓存还有很多，那么接收窗口就大

cwnd：拥塞窗口，根据线路中的拥塞状况来决定，线路中不拥塞，那么此窗口就大，

发送窗口是取两个中较小值。这个还是可以理解的。

慢启动算法、快速恢复算法、结合来达到对拥塞进行控制的，想了解这两种算法的，百度一下百度百科

- **TCP释放时的四次挥手**

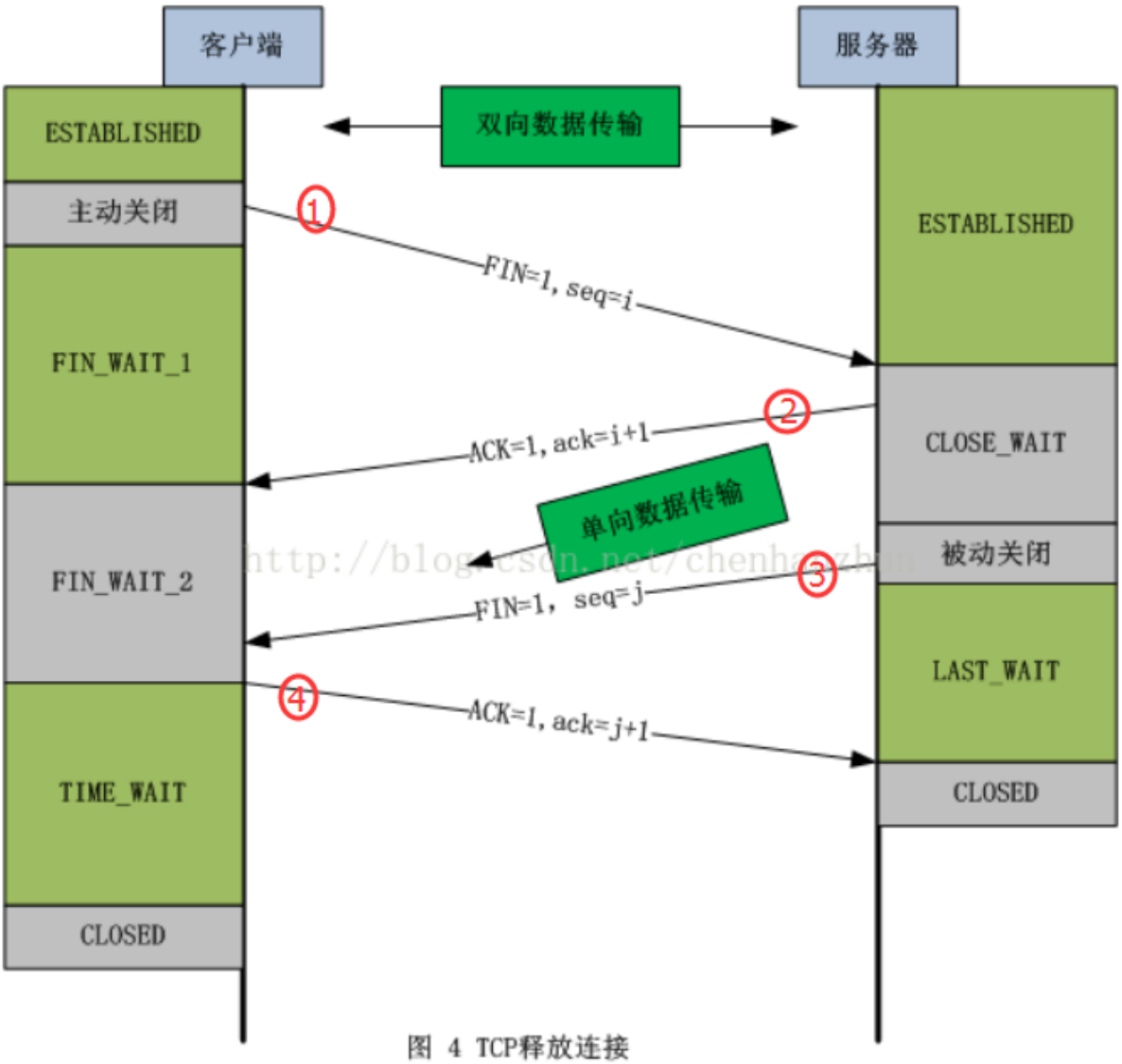


图 4 TCP释放连接

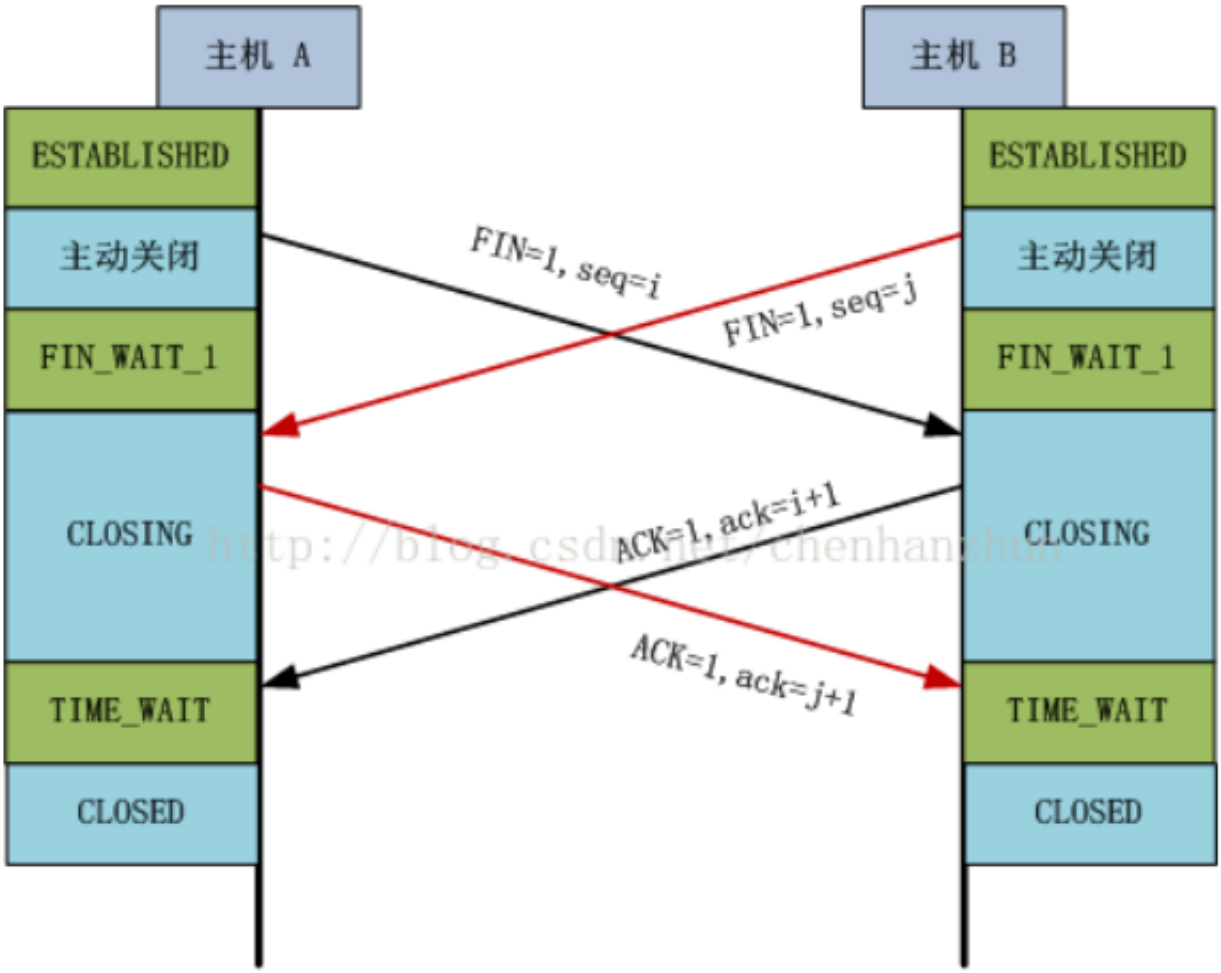
- 第一次挥手：从ESTABLISHED变为主动关闭状态，客户端主动发送释放连接请求给服务器端，FIN=1。发送完之后就变为FIN*WAIT*1状态，这个状态可以说是等待确认状态。

- 第二次挥手：服务器接收到客户端发来的释放连接请求后，状态变为CLOSE*WAIT，然后发送确认报文给客户端，告诉他我接收到了你的请求。为什么变为CLOSE*WAIT，原因是是客户端发送的释放连接请求，可能自己这端还有数据没有发送完呢，所以这个时候整个TCP连接的状态就变为了半关闭状态。服务器端还能发送数据，并且客户端也能接收数据，但是客户端不能在发送数据了，只能够发送确认报文。客户端接到服务器的确认报文后，就进入了FIN*WAIT*2

- 第三次挥手：服务器端所有的数据度发送完了，认为可以关闭连接了，状态变为被动关闭，所以向客户端发送释放连接报文，发完之后自己变为LAST_WAIT状态，也就是等待客户端确认状态

- 第四次挥手：客户端接到释放连接报文后，发送一个确认报文，然后自己变为TIME_WAIT,而不是立马关闭，因为客户端发送的确认报文可能会丢失，丢失的话服务器就会重传一个FIN，也就是释放连接报文，这个时候客户端必须还没关闭。 当服务器接受到确认报文后，服务器就进入CLOSE状态，也就是关闭了。但是由于上面说的这个原因，客户端必须等待一定的时间才能够进入CLOSE状态。

- **同时关闭连接**
- 正常情况下，通信一方请求连接关闭，另一方响应连接关闭请求，并且被动关闭连接。但是若出现同时关闭连接请求时，通信双方均从 ESTABLISHED 状态转换为 FIN*WAIT*1 状态。任意一方收到对方发来的 FIN 报文段后，其状态均由 FIN*WAIT*1转变到 CLOSING 状态，并发送最后的 ACK 数据段。当收到最后的 ACK 数据段后，状态转变化 TIME_WAIT，在等待 2MSL 时间后进入到 CLOSED 状态，最终释放整个 TCP 传输连接。其过程入下:



- **使用tcp协议的例子**
- http 协议

网络层

IP协议

- IP 协议位于 TCP/IP 协议的第三层——网络层。与传输层协议相比，网络层的责任是提供点到点(hop by hop)的服务，而传输层（TCP/UDP）则提供端到端(end to end)的服务。
- IP地址的分类

  A类地址 子网掩码 255.0.0.0

B类地址 子网掩码 255.255.0.0

C类地址 子网掩码 255.255.255.0

D类地址 多播地址

一、IP地址分类：

最初设计互联网络时，为了便于寻址以及层次化构造网络，每个IP地址包括两个标识码（ID），即网络ID和主机ID。同一个物理网络上的所有主机都使用同一个网络ID，网络上的一个主机（包括网络上工作站，服务器和路由器等）有一个主机ID与其对应。Internet委员会定义了5种IP地址类型以适合不同容量的网络，即A类~E类。

其中A、B、C3类（如下表格）由InternetNIC在全球范围内统一分配，D、E类为特殊地址。

类别	最大网络数	IP地址范围	最大主机数	私有IP地址范围
A	126（2^7-2)	0.0.0.0-127.255.255.255	16777214	10.0.0.0-10.255.255.255
B	16384(2^14)	128.0.0.0-191.255.255.255	65534	172.16.0.0-172.31.255.255
C	2097152(2^21)	192.0.0.0-223.255.255.255	254	192.168.0.0-192.168.255.255

- 特殊的IP地址 每一个字节都为0的地址（“0.0.0.0”）对应于当前主机；

  IP地址中的每一个字节都为1的IP地址（“255．255．255．255”）是当前子网的广播地址；

  IP地址中凡是以“11110”开头的E类IP地址都保留用于将来和实验使用。

  IP地址中不能以十进制“127”作为开头，该类地址中数字127．0．0．1到127．255．255．255用于回路测试，如：127.0.0.1可以代表本机IP地址，用“http://127.0.0.1”就可以测试本机中配置的Web服务器。

  网络ID的第一个8位组也不能全置为“0”，全“0”表示本地网络。

- 私有IP地址范围

  私有IP地址是一段保留的IP地址。只使用在局域网中，无法在Internet上使用。是为了解决IPv4地址不够用的问题，这类地址配合NAT可以缓解IPV4地址紧张的问题。

- 广播与多播

广播和多播仅用于UDP（TCP是面向连接的）

- 广播
- 多播

  多播又叫组播，使用D类地址，D类地址分配的28bit均用作多播组号而不再表示其他。

多播组地址包括1110的最高4bit和多播组号。它们通常可以表示为点分十进制数，范围从224.0.0.0到239.255.255.255。

多播的出现减少了对应用不感兴趣主机的处理负荷。

- 多播特点：

  允许一个或多个发送者（组播源）发送单一的数据包到多个接收者（一次的，同时的）的网络技术

  可以大大的节省网络带宽，因为无论有多少个目标地址，在整个网络的任何一条链路上只传送单一的数据包

  多播技术的核心就是针对如何节约网络资源的前提下保证服务质量。

BGP

- 边界网关协议（BGP）是运行于 TCP 上的一种自治系统的路由协议
- BGP 是唯一一个用来处理像因特网大小的网络的协议，也是唯一能够妥善处理好不相关路由域间的多路连接的协议
- BGP是一种外部网关协议（Exterior Gateway Protocol，EGP），与OSPF、RIP等内部网关协议（Interior Gateway Protocol，IGP）不同，BGP不在于发现和计算路由，而在于控制路由的传播和选择最佳路由
- BGP使用TCP作为其传输层协议（端口号179），提高了协议的可靠性
- BGP既不是纯粹的矢量距离协议，也不是纯粹的链路状态协议
- BGP支持CIDR（Classless Inter-Domain Routing，无类别域间路由）
- 路由更新时，BGP只发送更新的路由，大大减少了BGP传播路由所占用的带宽，适用于在Internet上传播大量的路由信息
- BGP路由通过携带AS路径信息彻底解决路由环路问题
- BGP提供了丰富的路由策略，能够对路由实现灵活的过滤和选择
- BGP易于扩展，能够适应网络新的发展

应用层

http协议

- HTTP构建于TCP/IP协议之上，默认端口号是80

- HTTP是无连接无状态的

http报文

- 请求报文
- HTTP 协议是以 ASCII 码传输，建立在 TCP/IP 协议之上的应用层规范。规范把 HTTP 请求分为三个部分：状态行、请求头、消息主体。类似于下面这样：

```
<method> <request-URL> <version>
<headers>


<entity-body>
```

HTTP定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而 HTTP 中的GET，POST，PUT，DELETE就对应着对这个资源的查，增，改，删4个操作。

GET用于信息获取，而且应该是安全的 和 幂等的。

所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

幂等的意味着对同一URL的多个请求应该返回同样的结果。

GET请求报文示例：

```
GET /books/?sex=man&name=Professional HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
Gecko/20050225 Firefox/1.0.1
Connection: Keep-Alive
```

POST表示可能修改变服务器上的资源的请求。

```
POST / HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
Gecko/20050225 Firefox/1.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Connection: Keep-Alive

sex=man&name=Professional
```

注意:

- GET 可提交的数据量受到URL长度的限制，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制 理论上讲，POST 是没有大小限制的，HTTP 协议规范也没有进行大小限制，出于安全考虑，服务器软件在实现时会做一定限制 参考上面的报文示例，可以发现 GET 和 POST 数据内容是一模一样的，只是位置不同，一个在URL里，一个在 HTTP 包的包体里 POST 提交数据的方式

- HTTP 协议中规定 POST 提交的数据必须在 body 部分中，但是协议中没有规定数据使用哪种编码方式或者数据格式。实际上，开发者完全可以自己决定消息主体的格式，只要最后发送的 HTTP 请求满足上面的格式就可以。

- 但是，数据发送出去，还要服务端解析成功才有意义。一般服务端语言如 php、python 等，以及它们的 framework，都内置了自动解析常见数据格式的功能。服务端通常是根据请求头（headers）中的 Content-Type 字段来获知请求中的消息主体是用何种方式编码，再对主体进行解析。所以说到 POST 提交数据方案，包含了 Content-Type 和消息主体编码方式两部分。下面就正式开始介绍它们：

- application/x-www-form-urlencoded 这是最常见的 POST 数据提交方式。浏览器的原生 `<form>` 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。上个小节当中的例子便是使用了这种提交方式。可以看到 body 当中的内容和 GET 请求是完全相同的。

- multipart/form-data 这又是一个常见的 POST 数据提交的方式。我们使用表单上传文件时，必须让 `<form>` 表单的 enctype 等于 multipart/form-data。直接来看一个请求示例：

```
POST http://www.example.com HTTP/1.1
Content-Type:multipart/form-data; boundary=----WebKitFormBoundaryrGKCBY7qhFd3TrwA

------WebKitFormBoundaryrGKCBY7qhFd3TrwA
Content-Disposition: form-data; name="text"

title
------WebKitFormBoundaryrGKCBY7qhFd3TrwA
Content-Disposition: form-data; name="file"; filename="chrome.png"
Content-Type: image/png

PNG ... content of chrome.png ...
------WebKitFormBoundaryrGKCBY7qhFd3TrwA--
```

这个例子稍微复杂点。首先生成了一个 boundary 用于分割不同的字段，为了避免与正文内容重复，boundary 很长很复杂。然后 Content-Type 里指明了数据是以 multipart/form-data 来编码，本次请求的 boundary 是什么内容。消息主体里按照字段个数又分为多个结构类似的部分，每部分都是以 --boundary 开始，紧接着是内容描述信息，然后是回车，最后是字段具体内容（文本或二进制）。如果传输的是文件，还要包含文件名和文件类型信息。消息主体最后以 --boundary-- 标示结束。关于 multipart/form-data 的详细定义，请前往 RFC1867 查看（或者相对友好一点的 MDN 文档）。

- 这种方式一般用来上传文件，各大服务端语言对它也有着良好的支持。
- 上面提到的这两种 POST 数据的方式，都是浏览器原生支持的，而且现阶段标准中原生

  表单也只支持这两种方式（通过 元素的 enctype 属性指定，默认为 application/x-www-form-urlencoded。其实 enctype 还支持 text/plain，不过用得非常少）。
- 随着越来越多的 Web 站点，尤其是 WebApp，全部使用 Ajax 进行数据交互之后，我们完全可以定义新的数据提交方式，例如 application/json，text/xml，乃至 application/x-protobuf 这种二进制格式，只要服务器可以根据 Content-Type 和 Content-Encoding 正确地解析出请求，都是没有问题的。

响应报文

- HTTP 响应与 HTTP 请求相似，

  HTTP响应也由3个部分构成，分别是：

  ```
  状态行
  响应头(Response Header)
  响应正文
  ```

  状态行由协议版本、数字形式的状态代码、及相应的状态描述，各元素之间以空格分隔

```
200 OK 客户端请求成功
301 Moved Permanently 请求永久重定向
302 Moved Temporarily 请求临时重定向
304 Not Modified 文件未修改，可以直接使用缓存的文件。
400 Bad Request 由于客户端请求有语法错误，不能被服务器所理解。
401 Unauthorized 请求未经授权。这个状态代码必须和WWW-Authenticate报头域一起使用
403 Forbidden 服务器收到请求，但是拒绝提供服务。服务器通常会在响应正文中给出不提供服务的原因
404 Not Found 请求的资源不存在，例如，输入了错误的URL
500 Internal Server Error 服务器发生不可预期的错误，导致无法完成客户端的请求。
503 Service Unavailable 服务器当前不能够处理客户端的请求，在一段时间之后，服务器可能会恢复正常。
```

- 下面是一个HTTP响应的例子：

```
HTTP/1.1 200 OK

Server:Apache Tomcat/5.0.12
Date:Mon,6Oct2003 13:23:42 GMT
Content-Length:112

<html>...
```

- **条件get**
- HTTP 条件 GET 是 HTTP 协议为了减少不必要的带宽浪费，提出的一种方案。

- **持久连接**

- 我们知道 HTTP 协议采用“请求-应答”模式，当使用普通模式，即非 Keep-Alive 模式时，每个请求/应答客户和服务器都要新建一个连接，完成之后立即断开连接（HTTP 协议为无连接的协议）；当使用 Keep-Alive 模式（又称持久连接、连接重用）时，Keep-Alive 功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接。

- 在 HTTP 1.0 版本中，并没有官方的标准来规定 Keep-Alive 如何工作，因此实际上它是被附加到 HTTP 1.0协议上，如果客户端浏览器支持 Keep-Alive ，那么就在HTTP请求头中添加一个字段 Connection: Keep-Alive，当服务器收到附带有 Connection: Keep-Alive 的请求时，它也会在响应头中添加一个同样的字段来使用 Keep-Alive 。这样一来，客户端和服务器之间的HTTP连接就会被保持，不会断开（超过 Keep-Alive 规定的时间，意外断电等情况除外），当客户端发送另外一个请求时，就使用这条已经建立的连接。

- 在 HTTP 1.1 版本中，默认情况下所有连接都被保持，如果加入 "Connection: close" 才关闭。目前大部分浏览器都使用 HTTP 1.1 协议，也就是说默认都会发起 Keep-Alive 的连接请求了，所以是否能完成一个完整的 Keep-Alive 连接就看服务器设置情况。

- 由于 HTTP 1.0 没有官方的 Keep-Alive 规范，并且也已经基本被淘汰，以下讨论均是针对 HTTP 1.1 标准中的 Keep-Alive 展开的。

- **注意：**

- HTTP Keep-Alive 简单说就是保持当前的TCP连接，避免了重新建立连接。
- HTTP 长连接不可能一直保持，例如 Keep-Alive: timeout=5, max=100，表示这个TCP通道可以保持5秒，max=100，表示这个长连接最多接收100次请求就断开。
- HTTP 是一个无状态协议，这意味着每个请求都是独立的，Keep-Alive 没能改变这个结果。另外，Keep-Alive也不能保证客户端和服务器之间的连接一定是活跃的，在 HTTP1.1 版本中也如此。唯一能保证的就是当连接被关闭时你能得到一个通知，所以不应该让程序依赖于 Keep-Alive 的保持连接特性，否则会有意想不到的后果。
- 使用长连接之后，客户端、服务端怎么知道本次传输结束呢？两部分：1. 判断传输数据是否达到了 Content-Length 指示的大小；2. 动态生成的文件没有 Content-Length ，它是分块传输（chunked），这时候就要根据 chunked 编码来判断，chunked 编码的数据在最后有一个空 chunked 块，表明本次传输数据结束，详见这里。什么是 chunked 分块传输呢？下面我们就来介绍一下。

http pipelining

- 默认情况下 HTTP 协议中每个传输层连接只能承载一个 HTTP 请求和响应，浏览器会在收到上一个请求的响应之后，再发送下一个请求。在使用持久连接的情况下，某个连接上消息的传递类似于请求1 -> 响应1 -> 请求2 -> 响应2 -> 请求3 -> 响应3。
- HTTP Pipelining（管线化）是将多个 HTTP 请求整批提交的技术，在传送过程中不需等待服务端的回应。使用 HTTP Pipelining 技术之后，某个连接上的消息变成了类似这样请求1 -> 请求2 -> 请求3 -> 响应1 -> 响应2 -> 响应3。
- 注意下面几点：
- 管线化机制通过持久连接（persistent connection）完成，仅 HTTP/1.1 支持此技术（HTTP/1.0不支持）
- 只有 GET 和 HEAD 请求可以进行管线化，而 POST 则有所限制
- 初次创建连接时不应启动管线机制，因为对方（服务器）不一定支持 HTTP/1.1 版本的协议 管线化不会影响响应到来的顺序，如上面的例子所示，响应返回的顺序并未改变
- HTTP /1.1 要求服务器端支持管线化，但并不要求服务器端也对响应进行管线化处理，只是要求对于管线化的请求不失败即可
- 由于上面提到的服务器端问题，开启管线化很可能并不会带来大幅度的性能提升，而且很多服务器端和代理程序对管线化的支持并不好，因此现代浏览器如 Chrome 和 Firefox 默认并未开启管线化支持
- cookie:Cookie是Web服务器发送给客户端的一小段信息，客户端请求时可以读取该信息发送到服务器端，进而进行用户的识别。对于客户端的每次请求，服务器都会将Cookie发送到客户端,在客户端可以进行保存,以便下次使用。

客户端可以采用两种方式来保存这个Cookie对象，一种方式是保存在客户端内存中，称为临时Cookie，浏览器关闭后这个Cookie对象将消失。另外一种方式是保存在客户机的磁盘上，称为永久Cookie。以后客户端只要访问该网站，就会将这个Cookie再次发送到服务器上，前提是这个Cookie在有效期内，这样就实现了对客户的跟踪。

Cookie是可以被禁止的。

- session:每一个用户都有一个不同的session，各个用户之间是不能共享的，是每个用户所独享的，在session中可以存放信息。

在服务器端会创建一个session对象，产生一个sessionID来标识这个session对象，然后将这个sessionID放入到Cookie中发送到客户端，下一次访问时，sessionID会发送到服务器，在服务器端进行识别不同的用户。

Session的实现依赖于Cookie，如果Cookie被禁用，那么session也将失效。