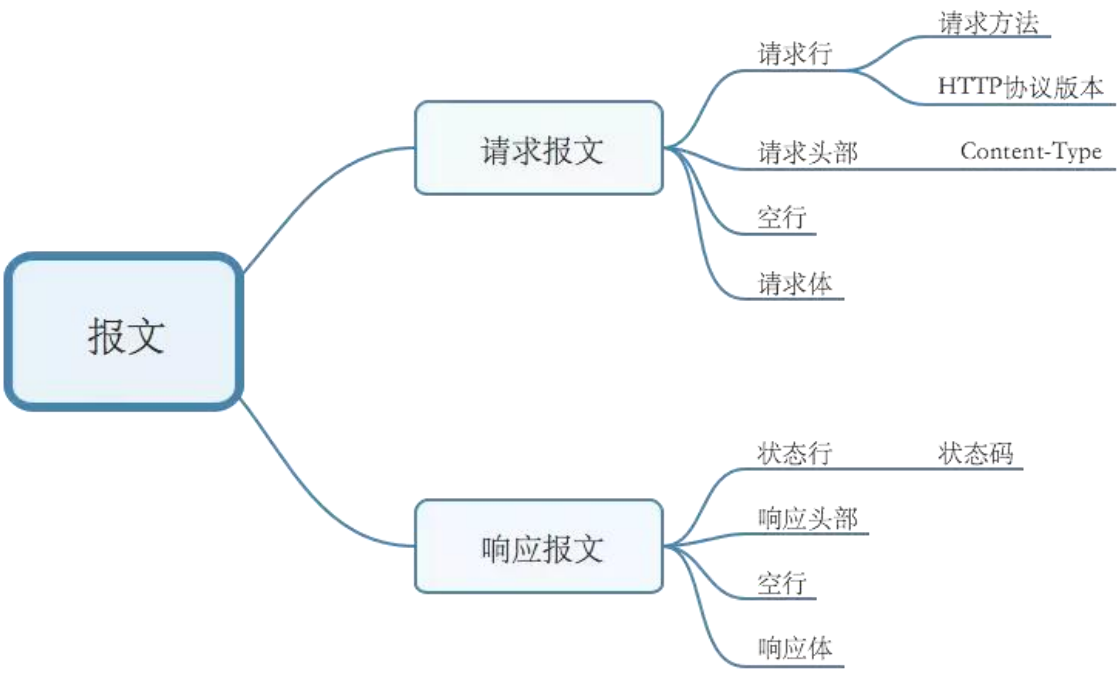
网络相关面试题

一、HTTP 协议

HTTP 协议：超文本传输协议

是一种详细规定了浏览器和万维网(WWW = World Wide Web)服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。
HTTP 是基于 TCP 的应用层协议
(OSI 网络七层协议从上到下分别是 应用层、表示层、会话层 、传输层、网络层 、数据链路层、物理层)

- 请求/响应报文
- 连接建立流程
- HTTP 的特点



A、请求报文和响应报文

1、请求报文

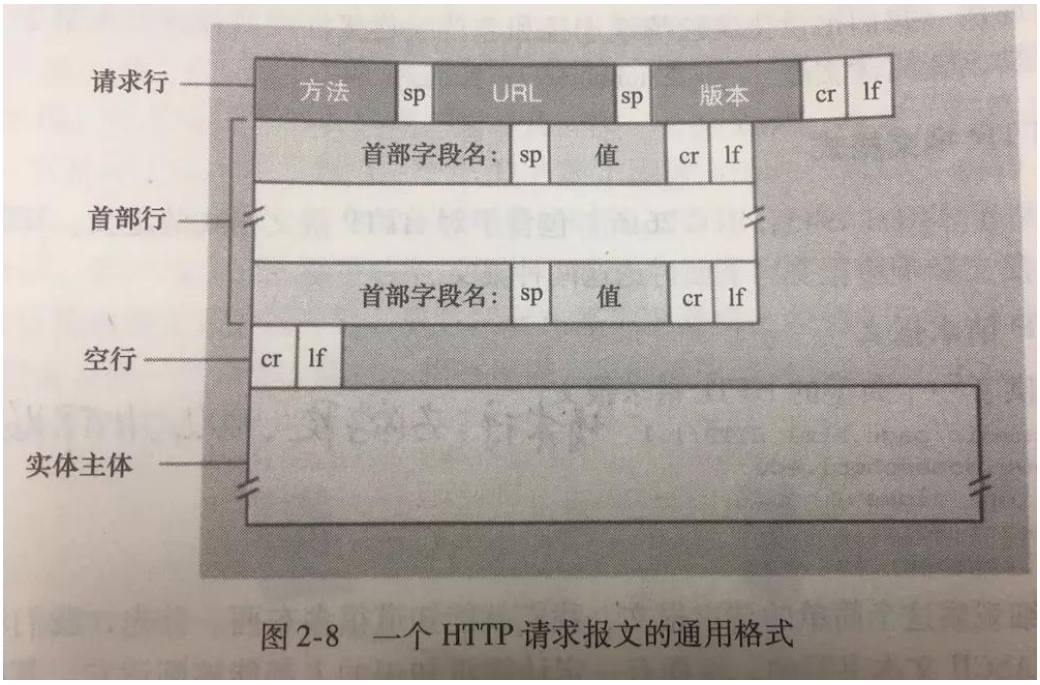


图 2-8　一个 HTTP 请求报文的通用格式

如下:

```
POST /somedir/page.html HTTP/1.1
//以上是请求行:方法字段、URL字段和HTTP版本字段
Host: www.user.com
Content-Type: application/x-www-form-urlencoded
Connection: Keep-Alive
User-agent: Mozilla/5.0.
Accept-lauguage: fr
//以上是首部行
（此处必须有一空行）  //空行分割header和请求内容
name=world    请求体
```

Host：指明了该对象所在的主机

Connection：Keep-Alive 首部行用来表明该浏览器告诉服务器使用持续连接

Content-Type: x-www-form-urlencoded 首部行用来表明 HTTP 会将请求参数用 key1=val1&key2=val2 的方式进行组织，并放到请求实体里面

User-agent：首部行用来指明用户代理，即向服务器发送请求的浏览器类型

Accept-lauguage：首部行表示用户想得到该对象的法语版本（如果服务器中有这样的对象的话），否则，服务器应发送它的默认版本

2、响应报文

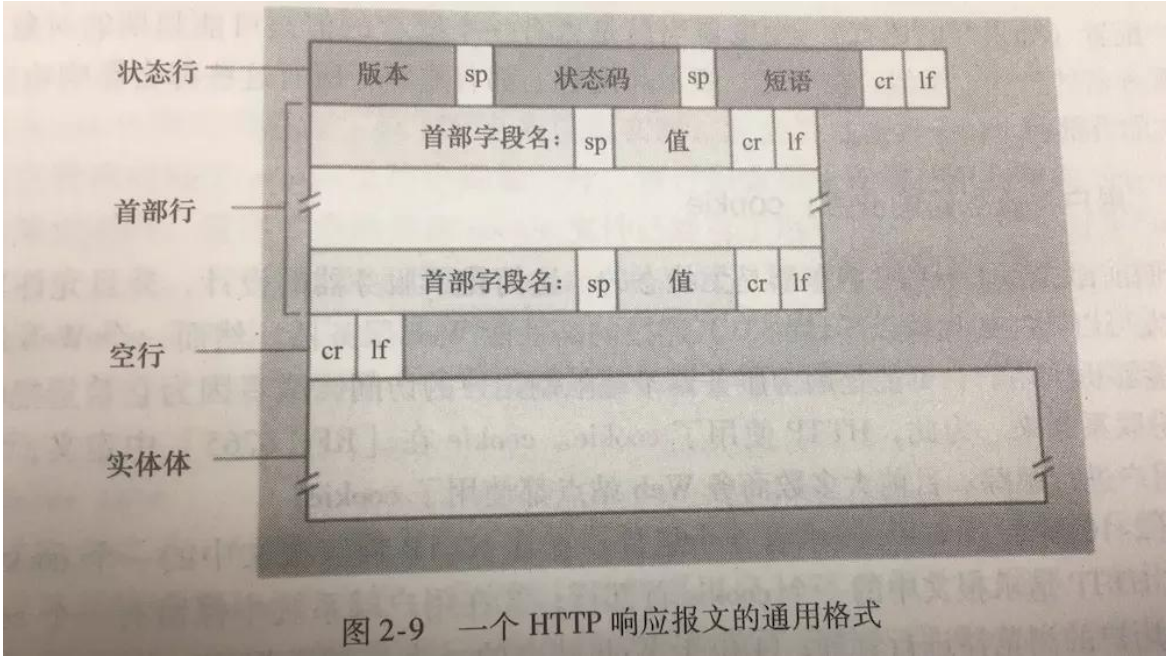


图 2-9　一个 HTTP 响应报文的通用格式

如下：

```
HTTP/1.1 200 OK
//以上是状态行：协议版本字段、状态码、相应状态信息
Connection：close
Server:Apache/2.2.3(CentOS)
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html
Content-Length: 122
//以上是首部行
（此处必须有一空行）  //空行分割header和实体主体
(data data data data)//响应实体主体
```

状态码及其相应的短语指示了请求的结果。

一些常见的状态码和对应的短语：

- **200 OK**：请求成功，信息在返回的响应报文中
- **301 Moved Permanently**：请求的对象已经被永久转移了，新的 URL 定义在响应报文中的 Location：首部行中。客户软件将自动获取新的 URL
- **400 Bad Request**：一个通用差错代码，指示该请求不能被服务器理解
- **404 Not Found**：被请求的文件不在服务器上
- **505 HTTP Version Not Supported**：服务器不支持请求报文使用的 HTTP 协议版本
  <4 开头的状态码通常是客户端的问题，5 开头的则通常是服务端的问题>

Connection：close 首部行告诉客户，发送完报文后将关闭 TCP 连接。
Date:指的不是对象创建或最后修改的时间，而是服务器从文件系统中检索到该对象，插入到响应报文，并发送该响应报文的时间。
Server：首部行指示该报文是由一台 Apache Web 服务器产生的，类似于 HTTP 请求报文里的 User-agent

Content-Length：首部行指示了被发送对象中的字节数
Content-Type：首部行指示了实体体中的对象是 HTML 文本

二、HTTP 的请求方式

GET、POST、PUT、DELETE、HEAD、OPTIONS

1、GET 和 POST 方式的区别

从语法角度来看，最直观的区别就是

- GET 的请求参数一般以?分割拼接到 URL 后面，POST 请求参数在 Body 里面
- GET 参数长度限制为 2048 个字符，POST 一般是没限制的
- GET 请求由于参数裸露在 URL 中， 是不安全的，POST 请求则是相对安全
  之所以说是相对安全，是因为，如果 POST 虽然参数非明文，但如果被抓包，GET 和 POST 一样都是不安全的。(HTTPS 该用还是得用)

而从语义的角度来看：

GET：获取资源是 安全的，幂等的(只读的，纯粹的)， 可缓存的
POST：获取资源是 非安全的，非幂等的，不可缓存的

- 这里的安全是指不应引起 Server 端的任何状态变化
  GET 的语义就是获取数据，是不会引起服务器的状态变化的，即是安全的。（HEAD，OPTIONS 也是安全的）
  而 POST 语义则是提交数据，是可能会引起服务器状态变化的，即是不安全的
- 幂等:同一个请求方法执行多次和执行一次的效果完全相同
  显然 GET 请求是幂等而 POST 请求是非幂等的。
  这里用幂等形容 GET 还不够，因为 GET 不止是执行多次和执行一次的效果完全相同，而且是执行一次和执行零次的效果也是完全相同的。
- 可缓存的
  请求是否可以被缓存。
  GET 请求会主动进行 Cache

以上特性，并非并列，正是因为 GET 是幂等的只读的，即 GET 请求除了返回数据不会有其他副作用，所以 GET 才是安全的，从而可以直接由 CDN 缓存，大大减轻服务器的负担，也就是可缓存的。
而 POST 是非幂等的，即除了返回数据还会有其他副作用，所以 POST 是不安全的，必须交由 web 服务器处理，即是 不可缓存的

GET 和 POST 本质上就是 TCP 链接，并无差别。但是由于 HTTP 的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。

在响应时，GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包:
对于 GET 方式的请求，浏览器会把 Header 和实体主体一并发送出去，服务器响应 200（返回数据）；

而对于 POST，浏览器先发送 Header，服务器响应 100 Continue，浏览器再发送实体主体，服务器响应 200 OK（返回数据）。

2、GET 相对 POST 的优势是什么？

1、最大的优势就是方便。GET 的 URL 可以直接手输，从而 GET 请求中的 URL 可以被存在书签里，或者历史记录里
2、可以被缓存，大大减轻服务器的负担
所以大多数情况下，还是用 GET 比较好。

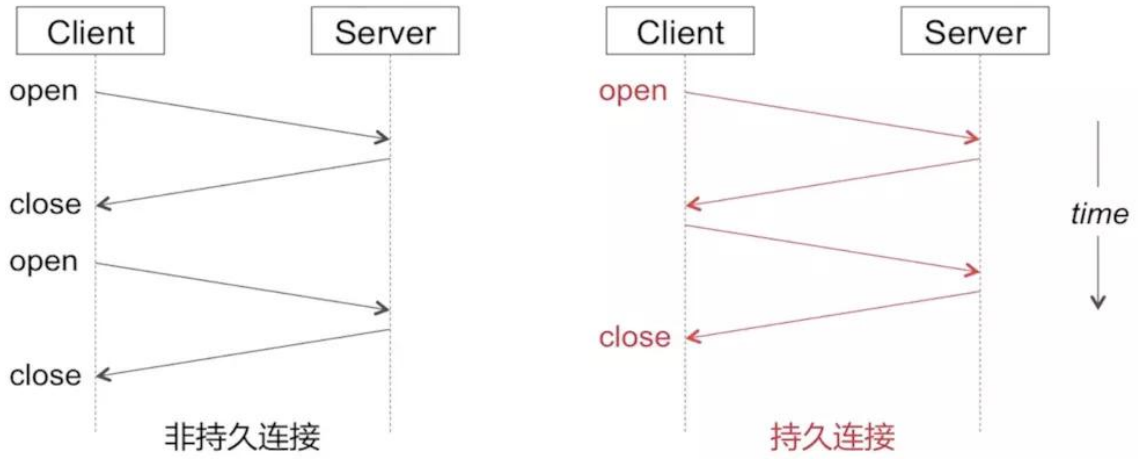
三、HTTP 的特点

无连接、 无状态
HTTP 的持久连接、Cookie/Session

1、HTTP 的无状态

即协议对于事务处理没有记忆能力。
每次的请求都是独立的，它的执行情况和结果与前面的请求和之后的请求时无直接关系的，它不会受前面的请求应答情况直接影响，也不会直接影响后面的请求应答情况
也就是说**服务器中没有保存客户端的状态，客户端必须每次带上自己的状态去请求服务器**
标准的 HTTP 协议指的是不包括 cookies，session，application 的 HTTP 协议

2、HTTP 的持久连接



- 非持久连接：每个连接处理一个请求-响应事务。
- 持久连接：每个连接可以处理多个请求-响应事务。

持久连接情况下，服务器发出响应后让 TCP 连接继续打开着。同一对客户/服务器之间的后续请求和响应可以通过这个连接发送。
HTTP/1.0 使用非持久连接。HTTP/1.1 默认使用持久连接<keep-alive>。

非持久连接的每个连接，TCP 得在客户端和服务端分配 TCP 缓冲区，并维持 TCP 变量，会严重增加服务器负担。而且每个对象都有 2 个 RTT(Round Trip Time，也就是一个数据包从发出去到回来的时间)的延迟，由于 TCP 的拥塞控制方案,每个对象都遭受 TCP 缓启动，因为每个 TCP 连接都起始于缓启动阶段

HTTP 持久连接怎么判断一个请求是否结束的？

- Content-length：根据所接收字节数是否达到 Content-length 值
- chunked(分块传输):Transfer-Encoding。当选择分块传输时，响应头中可以不包含 Content-Length，服务器会先回复一个不带数据的报文（只有响应行和响应头和\r\n），然后开始传输若干个数据块。当传输完若干个数据块后，需要再传输一个空的数据块，当客户端收到空的数据块时，则客户端知道数据接收完毕。

四、HTTPS、对称加密、非对称加密

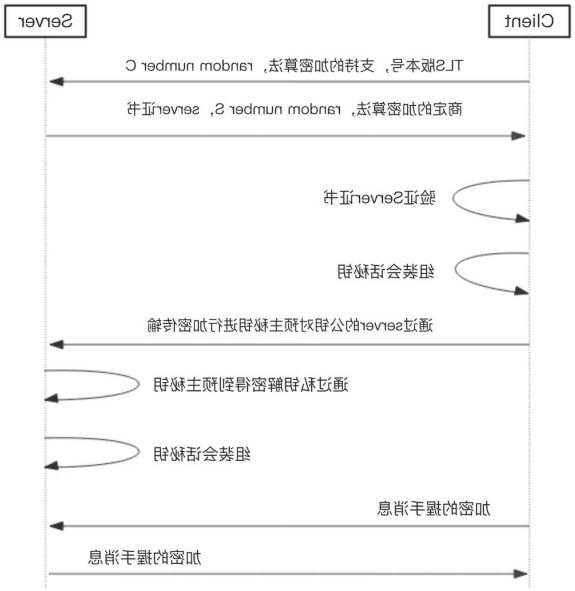
1、HTTPS 和 HTTP 的区别

HTTPS 协议 = HTTP 协议 + SSL/TLS 协议
SSL 的全称是 Secure Sockets Layer，即安全套接层协议，是为网络通信提供安全及数据完整性的一种安全协议。TLS 的全称是 Transport Layer Security，即安全传输层协议。
即 HTTPS 是安全的 HTTP。

2、HTTPS 的连接建立流程

HTTPS 为了兼顾安全与效率，同时使用了对称加密和非对称加密。在传输的过程中会涉及到三个密钥：

- 服务器端的公钥和私钥，用来进行非对称加密
- 客户端生成的随机密钥，用来进行对称加密



如上图，HTTPS 连接过程大致可分为八步：

1、客户端访问 HTTPS 连接。

客户端会把安全协议版本号、客户端支持的加密算法列表、随机数 C 发给服务端。

2、服务端发送证书给客户端

服务端接收密钥算法配件后，会和自己支持的加密算法列表进行比对，如果不符合，则断开连接。否则，服务端会在该算法列表中，选择一种对称算法（如 AES）、一种公钥算法（如具有特定秘钥长度的 RSA）和一种 MAC 算法发给客户端。
服务器端有一个密钥对，即公钥和私钥，是用来进行非对称加密使用的，服务器端保存着私钥，不能将其泄露，公钥可以发送给任何人。
在发送加密算法的同时还会把数字证书和随机数 S 发送给客户端

3、客户端验证 server 证书

会对 server 公钥进行检查，验证其合法性，如果发现发现公钥有问题，那么 HTTPS 传输就无法继续。

4、客户端组装会话秘钥

如果公钥合格，那么客户端会用服务器公钥来生成一个前主秘钥(Pre-Master Secret，PMS)，并通过该前主秘钥和随机数 C、S 来组装成会话秘钥

5、客户端将前主秘钥加密发送给服务端

是通过服务端的公钥来对前主秘钥进行非对称加密，发送给服务端

6、服务端通过私钥解密得到前主秘钥

服务端接收到加密信息后，用私钥解密得到主秘钥。

7、服务端组装会话秘钥

服务端通过前主秘钥和随机数 C、S 来组装会话秘钥。
至此，服务端和客户端都已经知道了用于此次会话的主秘钥。

8、数据传输

客户端收到服务器发送来的密文，用客户端密钥对其进行对称解密，得到服务器发送的数据。
同理，服务端收到客户端发送来的密文，用服务端密钥对其进行对称解密，得到客户端发送的数据。

总结：

会话秘钥 = random S + random C + 前主秘钥

- HTTPS 连接建立过程使用非对称加密，而非对称加密是很耗时的一种加密方式

- 后续通信过程使用对称加密，减少耗时所带来的性能损耗
- 其中，对称加密加密的是实际的数据，非对称加密加密的是对称加密所需要的客户端的密钥。

五、对称加密和非对称加密

1、对称加密

用同一套密钥来进行加密解密。
对称加密通常有 DES,IDEA,3DES 加密算法。

2、非对称加密

用公钥和私钥来加解密的算法。
公钥（Public Key）与私钥（Private Key）是通过一种算法得到的一个密钥对（即一个公钥和一个私钥），公钥是密钥对中公开的部分，私钥则是非公开的部分,私钥通常是保存在本地。

- 用公钥进行加密，就要用私钥进行解密；反之，用私钥加密，就要用公钥进行解密（数字签名）。
- 由于私钥是保存在本地的，所以非对称加密相对与对称加密是安全的。
  但非对称加密比对称加密耗时(100 倍以上),所以通常要结合对称加密来使用。

常见的非对称加密算法有：RSA、ECC（移动设备用）、Diffie-Hellman、El Gamal、DSA（数字签名用）

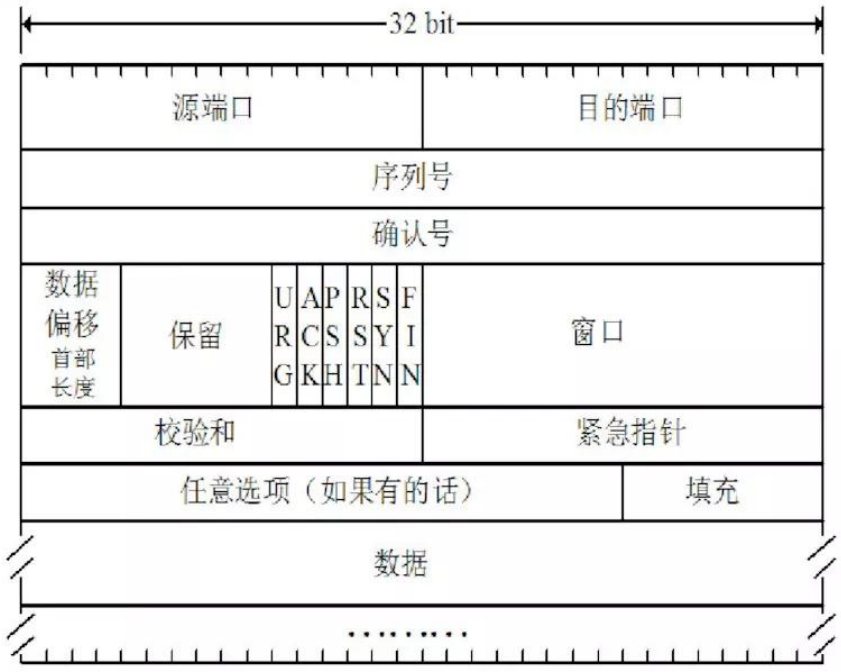
而为了确保客户端能够确认公钥就是想要访问的网站的公钥，引入了数字证书的概念，由于证书存在一级一级的签发过程，所以就出现了证书链，在证书链中的顶端的就是根 CA。

六、TCP 的特点和报文结构

1、面向连接、可靠传输、面向字节流、全双工服务

2、TCP 的报文结构

TCP 报文段由**首部字段**和一个**数据字段**组成。
数据字段包含一块应用数据。最大报文长度 MSS（Maximum Segment Size）限制了报文段数据字段的最大长度。MSS 选项用于在 TCP 连接建立时，收发双方协商通信时每一个报文段所能承载的最大数据长度。
所以当 TCP 发送一个大文件（比如一张高清图片）时，通常是将该文件划分为 MSS 长度的若干块（最后一块除外，通常会小于 MSS）。而实际交互式应用通常传送长度小于 MSS 的数据块。



如图，与 UDP 一样，首部包括**源端口号**和**目的端口号**，用于多路复用/分解来自上层或送到上层应用的数据。TCP 首部也同样包括**检验和字段**

TCP 首部还包含下列字段:

- 32 比特的**序号字段 Seq(sequence number field)** 和 32 比特的**确认号字段 Ack(acknowledge number field)**
- 16 比特的**接收窗口字段 RW(receive window field)**,该字段用于流量控制，用于指示接收方愿意接收的字节数量。
- 4 比特的**首部长度字段（header length field）**，该字段指示了以 32 比特的字为单位的 TCP 首部长度。由于 TCP 选项字段的原因，TCP 首部长度是可变的。（通常，选项字段为空，所以 TCP 首部的典型长度就是 20 字节）
- 可选和变长的**选项字段（option field）**，该字段用于发送方和接收方协商最大报文段长度（MSS）时，或用作窗口调节因子时使用。

- 6 比特的**标志字段（flag field）**。ACK 比特用于指示确认字段中的值是有效的，即该报文段包括一个对已被接收报文段的确认。**RST、SYN、FIN** 比特用于连接建立和拆除。
  PSH 比特指示接收方应立即将数据交给上层。**URG** 比特用于指示报文段里存在着被发送端的上层实体置为“紧急”的数据。紧急数据的最后一个字节由 16 比特的紧急数据指针字段指出。当紧急数据存在并给出指向紧急数据尾的指针的时候，TCP 必须通知接收端的上层实体。在实践中，PSH、URG 和紧急数据指针并没有使用。

3、序号字段 Seq 和确认号字段 Ack

- 在 TCP 通讯中，无论是建立连接，数据传输，友好断开，强制断开，都离不开 Seq 值和 Ack 值，它们是 TCP 传输的可靠保证。

序号 Seq：
TCP 把数据看成一个无结构的、有序的字节流。一个**报文段的序号**因此是该报文段的首字节的字节流编号。比如数据流由一个包含 100000 字节的文件组成，其 MSS 是 1000 字节，数据流的首字节编号是 0。该 TCP 将为该数据流构建 100 个报文段。给第一个报文段分配序号 0，第二个则是 1000，第三个是 2000，以此类推。每一个序号被填入到相应 TCP 报文段首部的序号字段中。

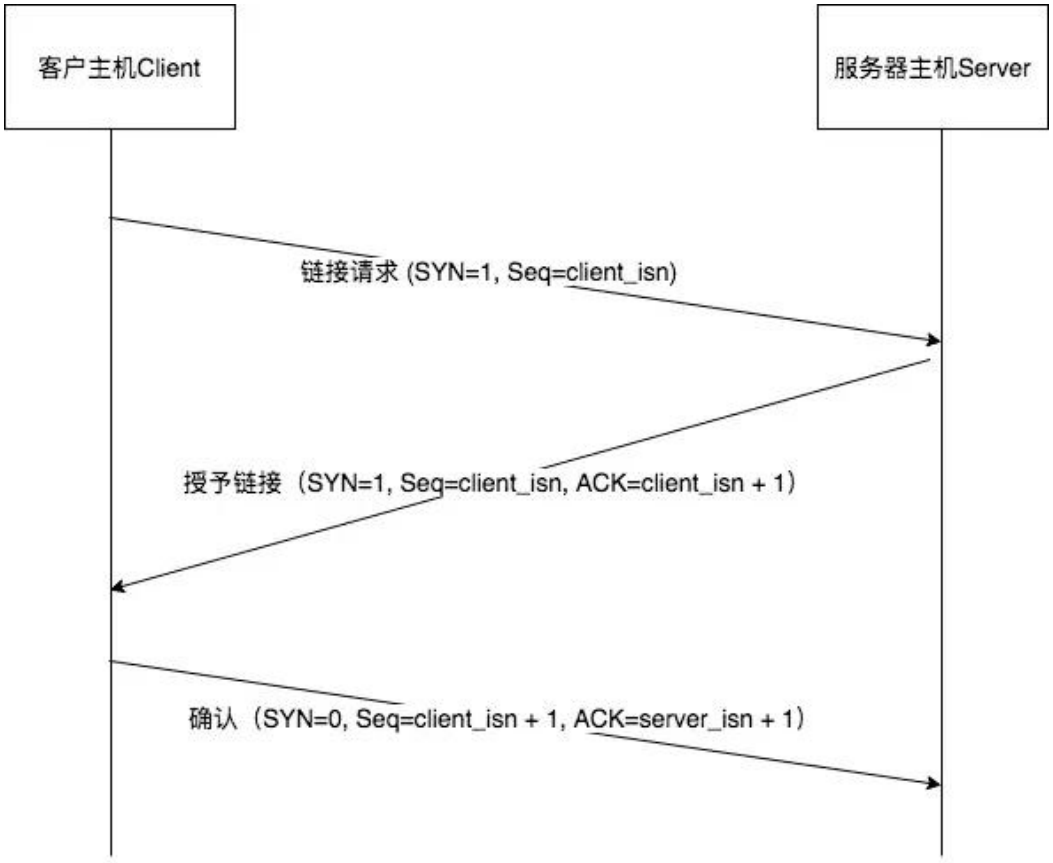
确认号 Ack：
TCP 是全双工服务的，因此主机 A 在向主机 B 发送数据的同时，也许也在接收主机 B 的数据。
主机 A 填充进报文段的确认号是主机 A 期望从主机 B 收到的下一个字节的序号。

在上个例子中，假如服务端已经接收包含字节 0-999 的报文段和包含字节 2000-2999 的报文段，但由于某种原因，还未收到包含字节 1000-1999 的报文段，那么将仍会等待字节 1000（及其后的字节）。因此服务端发给客户端的下一个报文段将在**确认号 Ack** 字段中包含 1000。
因为 TCP 只确认该流中至第一个丢失字节为止的字节，所以 TCP 被称为**累积确认**。

七、三次握手

-数据开始传输前，需要通过 三次握手来建立连接

事实上我认为，这里称呼三步握手（three-way handshake）才更贴切些



TCP 三次握手数据交互

第一步：

- 客户端的 TCP 首先向服务端的 TCP 发送一条特殊的 TCP 报文段。该报文段不包含应用层数据，该报文段首部中的一个标志位（SYN 比特）被置为 1，所以该报文段被称为 **SYN 报文段**。另外，客户会随机选择一个初始序号 client_isn，并将该序号放置于该起始的 TCP SYN 报文段的序号字段中。
- 客户端和服务端最开始都处于 CLOSED 状态，发送过该 SYN 报文段后，客户端 TCP 进入 SYN_SENT 状态，等待服务端确认并将 SYN 比特置为 1 的报文段。

第二步：

- 收到 SYN 报文段后，服务端会为该 TCP 连接分配 TCP 缓存和变量，服务端 TCP 会进入 SYN_RCVD 状态，等待客户端 TCP 发送确认报文段。
- 并向该客户端 TCP 发送允许连接的报文段，该报文段同样不包含应用层数据。该报文段首部的 SYN 比特被置为 1，确认号字段被置为 client_isn+1。服务端还会选择自己的初始序号 server_isn，放到报文段首部的序号段中。该连接被称为 **SYNACK 报文段**。

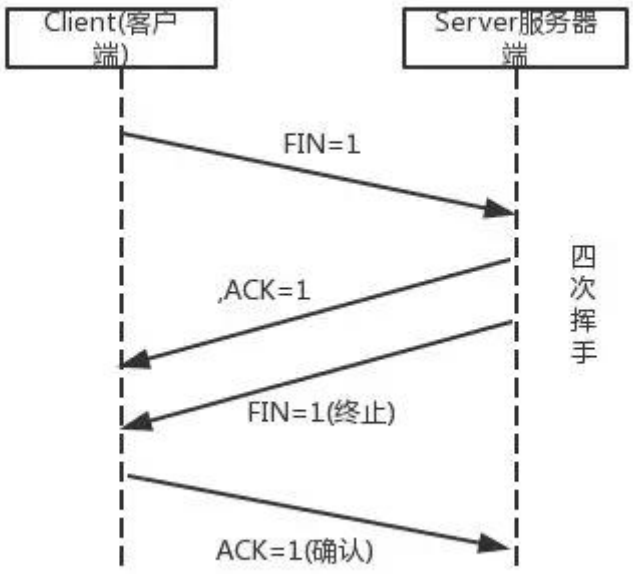
第三步：

- 收到 SYNACK 报文段后，客户端也要为该 TCP 连接分配缓存和变量，客户端 TCP 进入 ESTABLISHED 状态，在此状态，客户端就能发送和接收包含有效载荷数据的报文段了。
- 并向服务端 TCP 发送一个报文段：这最后一个报文段对服务端的允许连接的报文表示了确认（将 server_isn + 1 放到报文段首部的确认字段中）。因为连接已经建立了，所以该 SYN 比特被置为 0。这个阶段，可以在报文段负载中携带应用层数据。
- 收到客户端该报文段后，服务端 TCP 也会进入 ESTABLISHED 状态，可以发送和接收包含有效载荷数据的报文段。

八、四次挥手

参与 TCP 连接的两个进程中的任何一个都能终止该连接，当连接结束后，主机中的资源（缓存和变量）会被释放。

上边说到，SYN 和 FIN 标志位分别对应着 TCP 连接的建立和拆除。



第一步：

- 客户应用进程发出一个关闭连接的指令。会引起客户端 TCP 向服务端发送一个特殊的 TCP 报文段。该报文段即是将首部的一个标志位 FIN 比特置为 1。
- 同时，客户端进入 FIN_WAIT_1 状态，等待服务端的带有确认的 TCP 报文段。

第二步：

- 收到该报文段后会向客户端发送一个确认报文段。
- 服务端 TCP 进入 CLOSE_WAIT 状态，对应客户端的 TIME_WAIT，表示被动关闭。
- 客户端收到该报文段后，进入 FIN_WAIT_2 状态，等待服务端的 FIN 比特置为 1 的报文段。

第三步：

- 服务端发送自己的终止报文段，同样是把报文段首部的标志位 FIN 比特置为 1。
- 服务端 TCP 进入 LAST_ACK 状态，等待服务端最后的确认报文段。

第四步：

- 客户端收到服务端的终止报文段后，向服务端发送一个确认报文段。同时，客户端进入 TIME_WAIT 状态。
- 假如 ACK 丢失，TIME_WAIT 状态使 TCP 客户重传最后的确认报文，TIME_WAIT 通常会等待 2MSL（Maximum Segment Lifetime 最长报文段寿命）。经过等待后，连接就正式关闭，重新进入 CLOSED 状态，客户端所有资源将被释放。
- 服务端收到该报文段后，同样也会关闭，重新进入 CLOSED 状态，释放所有服务端 TCP 资源。

一些问题

1、问：为什么建立连接只用三次握手，而断开连接却要四次挥手？

- 首先，当客户端数据已发送完毕，且知道服务端也全部接收到了时，就会去断开连接即向服务端发送 FIN
- 服务端接收到客户端的 FIN，为了表示接收到了，就会向客户端发送 ACK
- 但此时，服务端可能还在发送数据，并没有关闭 TCP 窗口的意思，所以服务端的 FIN 和 ACK 并不是同步发的，只有当数据发送完了，才会发送 FIN
- **答：服务端的 FIN 和 ACK 需要分开发，并不是像三次握手中那样，SYN 可以和 ACK 同步发，所以就需要四次挥手**

2、在四次挥手中，客户端为什么在 TIME_WAIT 后必须等待 2MSL 时间呢？

这个 ACK 报文段有可能丢失，因而使处在 LAST_ACK 端的服务端收不到对已发送的 FIN 报文段的 ACK 报文段，从而服务端会去不断重传 FIN 报文段。
而客户端就能在 2MSL 时间内收到重传的 FIN 报文段。接着客户端重传一次确认，重新启动 2MSL 计时器。直至服务端收到后，客户端和服务端就都会进入 CLOSED 状态，关闭 TCP 连接。
而如果客户端不等待 2MSL 时间，而是在发送完 ACK 确认后立即释放资源，关闭连接，那么就无法收到服务端重传的 FIN 报文段，因而也不会再发送一次 ACK 确认报文段，这样，服务端就无法正常进入 CLOSED 状态，资源就一直无法释放了。

- 答：为了保证客户端发送的最后一个 ACK 报文段能够到达服务端。

3、TCP 在创建连接时，为什么需要三次握手而不是两次或四次？

一个简单的例子：

- 三次握手：
  “喂，你听得到吗？”
  “我听得到呀，你听得到我吗？”
  “我能听到你，今天 balabala……”
- 两次握手：
  “喂，你听得到吗？”
  “我听得到呀，你听得到我吗？”
  “喂，你听得到吗？”
  “……谁在说话？”
  “喂，你听得到吗？”
  “……”
- 四次握手：
  “喂，你听得到吗？”
  “我听得到呀”“你能听到我吗？”
  “……不想跟傻逼说话”

之所以不用四次握手的原因很容易理解，就是浪费资源，服务端的 SYN 和 ACK 可以一起发，完全没必要分开两次。

而如果是两次握手：
客户端发出的第一个连接请求 SYN 报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达服务端。本来这是一个早已失效的报文段。但服务端收到此失效的连接请求 SYN 报文段后，就误认为是客户端再次发出的一个新的连接请求 SYN 报文段。于是就向客户端发出 ACK 确认报文段，同意建立连接。假设不采用三次握手，那么只要服务端发出确认，新的连接就建立了。
由于现在客户端并没有发出建立连接的 SYN 请求，因此不会理睬服务端的确认，也不会向服务端发送数据。但服务端却以为新的运输连接已经建立，并一直等待客户端发来数据。这样，服务端的很多资源就白白浪费掉了。

事实上：TCP 对有数据的 TCP 报文段必须确认的原则，所以，客户端对服务端的 SYN 报文段必须回复一个 ACK 报文段表示确认。并且，TCP 不会为没有数据的 ACK 超时重传，那么当服务端没收到客户端的 ACK 确认报文段时，会超时重传自己的 SYN 报文段，一直到收到客户端的 ACK 为止。

- **答：两次握手会可能导致已失效的连接请求报文段突然又传送到了服务端产生错误，四次握手又太浪费资源**

代码实现

参考 UDP 的代码，其实 TCP 在代码实现上也很相似，首先·socket·初始化时不再用·SOCK_DGRAM·，而是用·SOCK_STREAM·

```
fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

TCP 服务端要多了一道监听、接受连接的过程：

```
int listen_ret = listen(fd,5);
int listen_socket = accept(_fd,(sockaddr *)&addr,&addr_len);
```

UDP 则是多了一道连接的过程：

```
int ret = connect(_fd, (struct sockaddr *) &addr, sizeof(addr));
```

然后就是在接收和发送数据时，不用再传主机和端口了。即由 recvfrom、sendto 改为 recv 和 send。

```
send(_fd, [buffer bytes], [buffer length], 0);
recv(_fd, receiveBuffer, sizeof(receiveBuffer), 0);
```

python 的客户端代码如下：

```
from socket import *
serverName = '127.0.0.1'
serverPort = 12000
clientSocket = socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase:\n')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1029)
print 'From server:\n',modifiedSentence
clientSocket.close()
```

服务端代码：

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'server is ready to receive'
connectionSocket,addr = serverSocket.accept()
sentence = connectionSocket.recv(1029)
capitalizeSentence = sentence.upper()
print capitalizeSentence
connectionSocket.send(capitalizeSentence)
connectionSocket.close()
```

九、可靠数据传输

网络层服务（IP 服务）是不可靠的。IP 不保证数据报的交付，不保证数据报的按序交付，也不保证数据报中数据的完整性。

TCP 则是在 IP 服务上创建了一种**可靠数据传输服务**
TCP 的**可靠数据传输服务**确保一个进程从其接收缓存中读出的数据流是无损坏、无间隔、无冗余、按序的数据流。即该字节流与连接的另一端发出的字节流是完全相同的。
作为 TCP 接收方，有三个与发送和重传有关的主要事件

1、从上层应用数据接收数据

将数据封装到一个报文段中，并把报文段交付给 IP。每个报文段都包含一个序号 Seq，即该报文段第一个数据字节的字节流编号。如果定时器还没有为其他报文段而运行，则启动定时器(即不是每条报文段都会启动一个定时器，而是一共只启动一个定时器)，定时器的过期间隔是 `TimeoutInterval`
是由 EstimatedRTT 和 DevRTT 计算得来的:TCP 的往返时间的估计与超时

2、超时

TCP 通过重传引起超时的报文段来响应超时事件，然后重启定时器。

而发送端超时有两种情况：发送数据超时，接收端发送 ACK 超时。这两种情况都会导致发送端在 `TimeoutInterval` 内接收不到 ACK 确认报文段。

- 1、如果是发送数据超时，直接重传即可。
- 2、而如果是接收端发送 ACK 超时，这种情况接收端实际上已经接收到发送端的数据了。那么当发送端超时重传时，接收端会丢弃重传的数据，同时再次发送 ACK。

而如果在 `TimeoutInterval` 后接收到了 ACK，会收下 ACK，但不做任何处理

- TCP 不会为没有数据的 ACK 超时重传

以下两种情况：

- 1、如果在发送两条或多条数据报文段都超时，那么只会重传序号最小的那个，并重启定时器。只要其余报文段的 ACK 在新重启的定时器超时前到达，就不会重传。
- 2、如果发送序号为 `100` 和 `120` 的两条数据报文段，序号 `100` 的 ACK 丢失，但收到了序号 `120` 的 ACK，由于累积确认机制，可以得出接收方已经接收到了序号 `100` 的报文段，这种情况也不会去重传。

3、接收到 ACK

用 TCP 状态变量 `SendBase` 指最早未被确认的字节的序号。则 `SendBase - 1` 指接收方已正确按序接收到的数据的最后一个字节的序号。
当收到 ACK 确认报文段后，会将 ACK 的值 Y 与 `SendBase` 比较。TCP 采用**累计确认**的方法，所以 `Y` 确认来字节编号在 Y 之前的所有字节都已经收到。如果 `Y` 比 `SendBase` 小，不用理会；而如果 `Y` 比 `SendBase`

大，则该 ACK 是在确认一个或多个先前未被确认的报文段，因此要更新 SendBase 变量，如果当前还有未被确认的报文段，TCP 还要重启定时器。
通过超时重传，能保证接收到的数据是无损坏、无冗余的数据流，但并不能保证按序。

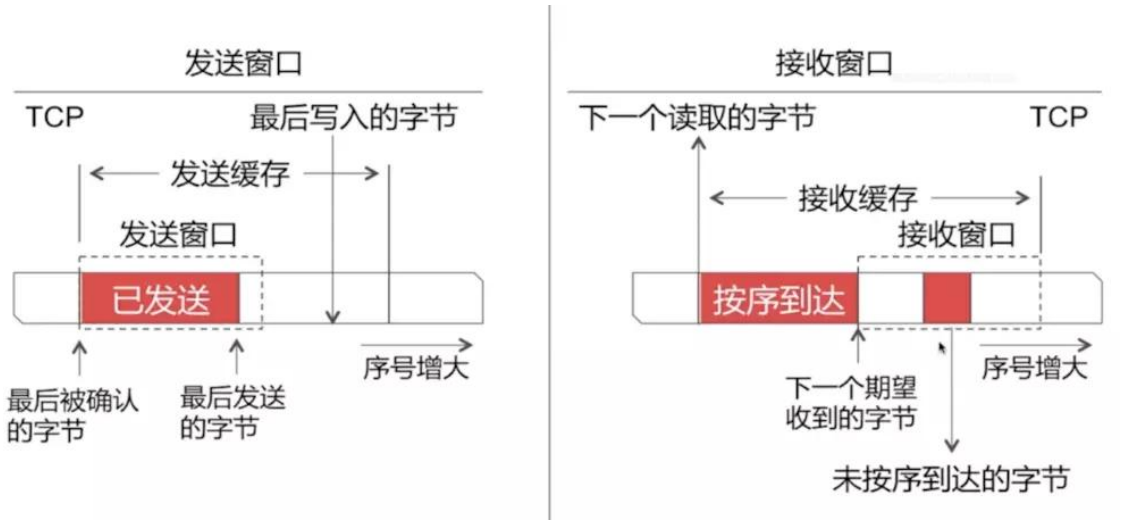
而通过 TCP 滑动窗口，能够有效保证接收数据有序

十、流量控制

TCP 连接的双方主机都会为该 TCP 连接分配缓存和变量。当该 TCP 连接收到正确、按序的字节后，就将数据放入接收缓存。上层的应用进程会从该缓存中读取数据，但不必是数据一到达就立即读取，因为此时应用程序可能在做其他事务。而如果应用层读取数据相对缓慢，而发送方发送得太多、太快，发送的数据就会很容易地使该连接的接收缓存溢出。

所以，TCP 为应用程序提供了**流量控制服务**（flow-control service），以消除发送方使接收方缓存溢出的可能性。

流量控制是一个速度匹配服务，即发送方的发送速率与接收方应用程序的读取速率相匹配。
作为全双工协议，TCP 会话的双方都各自维护一个**发送窗口**和一个**接收窗口**（receive window）的变量来提供流量控制。而发送窗口的大小是由对方接收窗口来决定的，接收窗口用于给发送方一个指示--该接收方还有多少可用的缓存空间。



1、发送窗口

发送方的发送缓存内的数据都可以被分为 4 类:

- 已发送，已收到 ACK
- 已发送，未收到 ACK
- 未发送，但允许发送
- 未发送，但不允许发送

则 2 和 3 属于发送窗口

- 发送窗口只有收到发送窗口内字节的 ACK 确认，才会移动发送窗口的左边界

2、接收窗口

接收方的缓存数据分为 3 类：
1.已接收
2.未接收但准备接收
3.未接收而且不准备接收
则 2 属于接收窗口（这里的接收指接收数据并确认）

- 接收窗口只有在前面所有的报文段都确认的情况下才会移动左边界。当在前面还有字节未接收但收到后面字节的情况下，会先接收下来，接收窗口不会移动，并不对后续字节发送 ACK 确认报文，以此确保发送端会对这些数据重传。

我们定义以下变量：

- LastByteRead：接收方应用程序读取的数据流的最后一个字节编号。可以得知，这是接收缓存的起点
- LastByteRcvd：从网络中到达的并且已放入接收缓存中的数据流的最后一个自己的的编号。

可以得知：LastByteRcvd - LastByteRead <= RcvBuffer(接收缓存大小)
那么接收窗口 rwnd =RcvBuffer - （LastByteRcvd - LastByteRead）
rwnd 是随时间动态变化的，如果 rwnd 为 0，则意味着接收缓存已经满了。
接收端在回复给发送端的 ACK 中会包含该 rwnd，发送端则会根据 ACK 中的接收窗口的值来控制发送窗口。

有一个问题，如果当发送 rwnd 为 0 的 ACK 后，发送端停止发送数据。等待一段时间后，接收方应用程序读取了一部分数据，接收端可以继续接收数据，于是给发送端发送报文告诉发送端其接收窗口大小，但这个报文不幸丢失了，我们知道，**不含数据的 ACK 是不会超时重传的**，于是就出现发送端等待接收端的 ACK 通知||接收端等待发送端发送数据的死锁状态。

为了处理这种问题，TCP 引入了持续计时器（Persistence timer），当发送端收到对方的 rwnd=0 的 ACK 通知时，就启用该计时器，时间到则发送一个 1 字节的探测报文，对方会在此时回应自身的接收窗口大小，如果结果仍未 0，则重设持续计时器，继续等待。

十一、拥塞控制

TCP 除了可靠传输服务外，另一个关键部分就是拥塞控制。
TCP 让每一个发送方根据所感知到的网络拥塞程度来限制其能向连接发送流量的速率。

可能有三个疑问：
1、TCP 发送方如何感知网络拥塞？
2、TCP 发送方如何限制其向连接发送流量的速率？
3、发送方感知到网络拥塞时，采用何种算法来改变其发送速率？

这就是 TCP 的**拥塞控制机制**。
前边说到，TCP 连接的每一端都是由一个接收缓存、一个发送缓存和几个变量（LastByteRead、LastByteRcvd、rwnd 等）组成。而运行在发送方的 TCP 拥塞控制机制会跟踪一个额外的变量，即**拥塞窗口 cwnd**（congestion window）。它对一个 TCP 发送方能向网络中发送流量的速率进行了限制。
发送方中未被确认的数据量不会超过 cwnd 和 rwnd 的最小值:min(rwnd,cwnd)

1、TCP 发送方如何感知网络拥塞？

冗余 ACK（duplicate ACK）：就是再次确认某个报文段的 ACK，而发送方先前已经收到对该报文段的确认。

冗余 ACK 的产生原因：

- 1.当接收端接收到失序报文段时，即该报文段序号大于下一个期望的、按序的报文段，检测到数据流中的间隔，即由报文段丢失，并不会对该报文段确认。TCP 不使用否定确认，所以不能向发送方发送显式的否定确认，为了使接收方得知这一现象，会对上一个按序字节数据进行重复确认，这也就产生了一个冗余 ACK。
- 2.因为发送方经常发送大量的报文段，如果其中一个报文段丢失，可能在定时器过期前，就会收到大量的冗余 ACK。一旦收到 3 个冗余 ACK（3 个以下很可能是链路层的乱序引起的，无需处理），说明在这个已被确认 3 次的报文段之后的报文段已经丢失，TCP 就会执行**快速重传**，即在该报文段的定时器过期之前重传丢失的报文段。

将 TCP 发送方的丢包事件**定义为：要么出现超时，要么收到来自接收方的 3 个**冗余 ACK**。**

当出现过度的拥塞时，路由器的缓存会溢出，导致一个数据报被丢弃。丢弃的数据报接着会引起发送方的丢包事件。那么此时，发送方就认为在发送方到接收方的路径上出现了网络拥塞。

2、TCP 发送方如何限制其向连接发送流量的速率？

- 当出现丢包事件时：应当降低 TCP 发送方的速率。
- 当对先前未确认报文段的确认到达时，即接收到非冗余 ACK 时，应当增加发送方的速率。

3、发送方感知到网络拥塞时，采用何种算法来改变其发送速率？

即 **TCP 拥塞控制算法（TCP congestion control algorithm）**

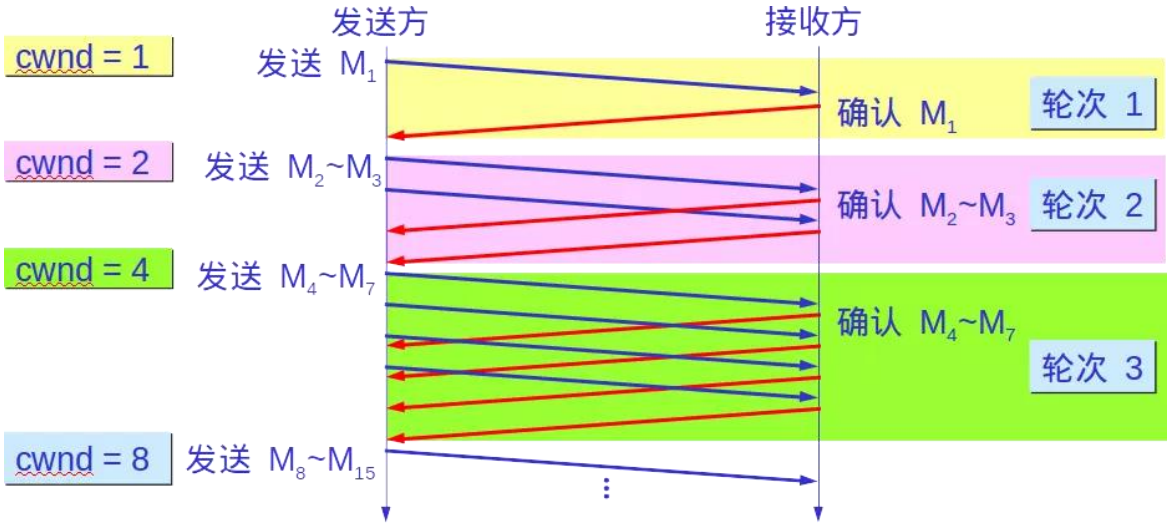
包括三个主要部分：**慢启动、拥塞避免、快速恢复**，其中快速恢复并非是发送方必须的，慢启动和拥塞避免则是 TCP 强制要求的

- **1、慢启动**

  当一条 TCP 连接开始时，拥塞窗口 cwnd 的值通常置为一个 MSS 的较小值，这就使初始发送速率大约为 **MSS/RTT**（RTT：往返时延，报文段从发出到对该报文段的确认被接收之间的时间量）。

  而对 TCP 发送方来说，可用带宽可能比 MSS/RTT 大得多,TCP 发送方希望迅速找到可用带宽的数量。

  因此，在慢启动状态，cwnd 以一个 MSS 的值开始并且每当收到一个非冗余 ACK 就增加一个 MSS。



最初 cwnd 值为 1MSS，发送一个报文段 M1。收到 M1 的确认后，cwnd 增加为 2MSS，这时可以发送两个报文段 M2，M3。收到这两个报文段的确认后，cwnd 则增加为 4MSS，可以发送四个报文段，以此类推...

因此，TCP 虽然发送速率起始慢，但在慢启动阶段以指数增长。

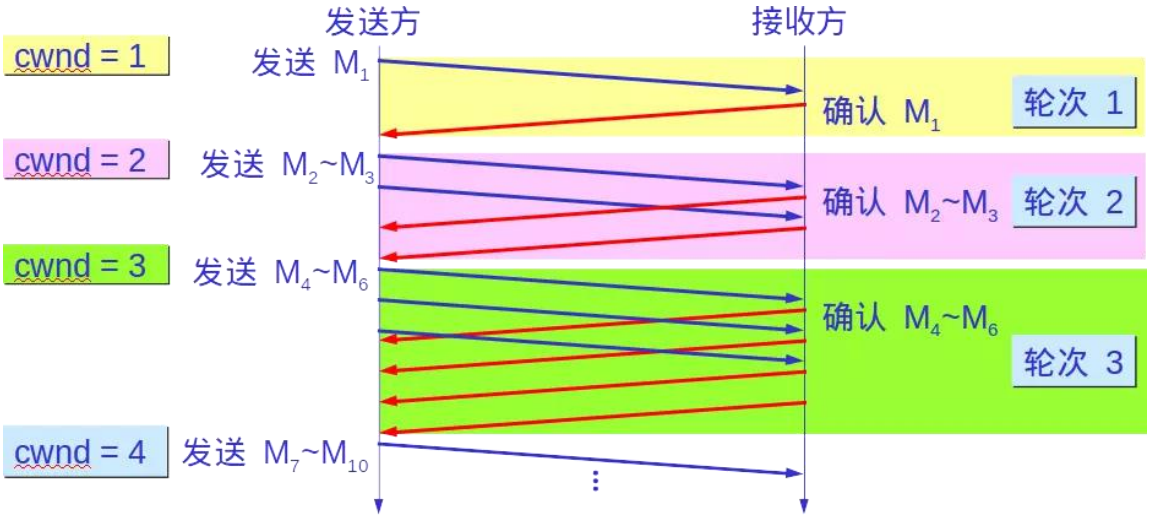
这种指数增长很显然不是无限制的，那么何时结束呢？

如果出现丢包事件，TCP 发送方将 ssthresh（慢启动阈值）设置为 cwnd/2

- 发生由超时引起的丢包事件，并将 cwnd 重置为 1MSS，重启慢启动
- 当 TCP 发送方的 cwnd 值达到或超过 ssthresh，再继续翻倍显然不合适。这时将结束慢启动转移到拥塞避免模式。
- TCP 发送方检测到 3 个冗余 ACK，会结束慢启动，并快速重传，即在该报文段的定时器过期之前重传丢失的报文段。且进入快速恢复状态。

十二、拥塞避免

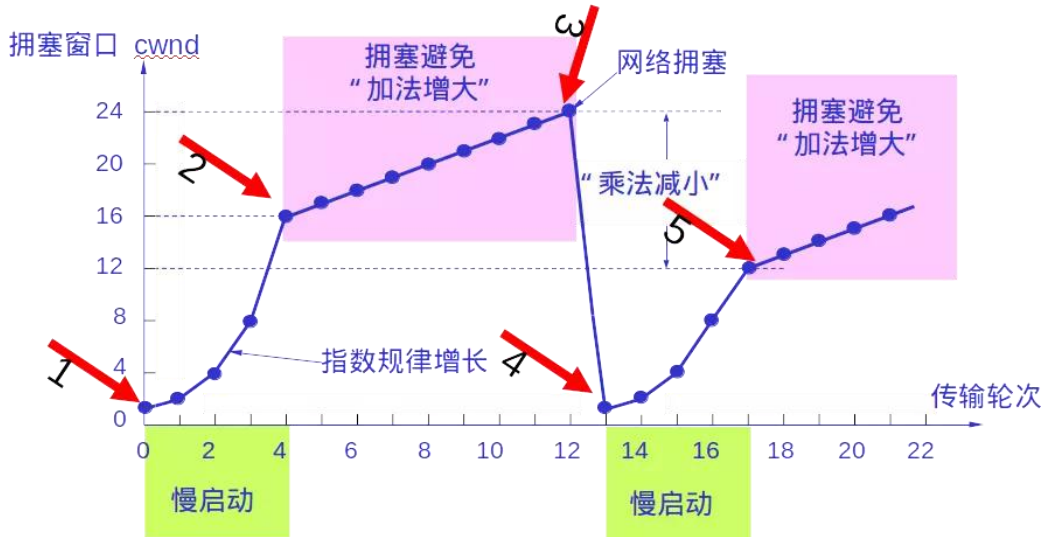
一旦进入拥塞避免状态，cwnd 的值大约是上次遇到拥塞时的值的一半，即距离拥塞并不遥远。因此，TCP 无法每过一个 RTT 就将 cwnd 翻倍。而是每个 RTT 只增加 1MSS，即每收到一个非冗余 ACK，就将 cwnd 增加 1/cwnd。即假如此时 cwnd 为 10MSS，则每收到一个非冗余 ACK，cwnd 就增加 1/10MSS，在 10 个报文段都收到确认后，拥塞窗口的值就增加了 1MSS。



那么何时结束拥塞避免的线性增长（每 RTT 1MSS）呢？

和慢启动一样，如果出现丢包事件，TCP 发送方将 ssthresh（慢启动阈值）设置为 cwnd/2（加法增大，乘法减小）

- 发生由超时引起的丢包事件，拥塞避免和慢启动处理的方式相同。即 TCP 发送方将 ssthresh（慢启动阈值）设置为 cwnd/2，并将 cwnd 重置为 1MSS，重启慢启动
- TCP 发送方检测到 3 个冗余 ACK，cwnd 为原来的一半加上 3MSS，进入快速恢复状态。



十三、快速恢复

快速恢复是由 3 个冗余 ACK 引起的。
在快速恢复中，对引起 TCP 进入快速恢复状态的缺失报文段，对收到的每个冗余 ACK，cwnd 增加 1 个 MSS。
最终，当对丢失报文段的一个 ACK 到达时，TCP 在降低 cwnd 后进入拥塞避免状态。
如果出现超时，和之前一样，即 TCP 发送方将 ssthresh（慢启动阈值）设置为 cwnd/2，并将 cwnd 重置为 1MSS，重启慢启动

快速恢复并非是必须的。

TCP 的拥塞控制是：每个 RTT 内 cwnd 线性（加性增）增加 1MSS，然后出现 3 个冗余 ACK 事件时 cwnd 减半（乘性减），因此 TCP 拥塞控制常被称为**加性增，乘性减**拥塞控制方式。

十四、DNS

因特网上的主机，可以使用多种方式标识，比如主机名或 IP 地址。

- 一种标识方法就是用它的**主机名（hostname）**，比如 www.baidu.com 、 www.google.com 、gaia.cs.umass.edu 等。这方式方便人们记忆和接受，但是这种长度不一、没有规律的字符串路由器并不方便处理。
- 还有一种方式，就是直接使用定长的、有着清晰层次结构的 IP 地址，路由器比较热衷于这种方式。

为了折衷这两种方式，我们需要一种能进行主机名到 IP 地址转换的目录服务。这就是***域名系统（Domain Name System，DNS）**的主要任务。

DNS 是：
1、一个由分层的 **DNS 服务器**实现的分布式数据库
2、一个使得主机能够查询分布式数据库的**应用层协议**

而 DNS 服务器通常是运行 BIND 软件的 UNIX 机器，DNS 协议运行在 **UDP** 上，使用 53 号端口
DNS 通常是由其他应用层协议所使用的，包括 HTTP、SMTP 等。其作用则是：**将用户提供的主机名解析为 IP 地址**

DNS 的一种简单设计就是在因特网上只使用一个 DNS 服务器，该服务器包含所有的映射。很明显这种设计是有很大的问题的：

单点故障：如果该 DNS 服务器崩溃，全世界的网络随之瘫痪

通信容量：单个 DNS 服务器必须处理所有 DNS 查询

远距离的集中式数据库：单个 DNS 服务器必须面对所有用户，距离过远会有严重的时延。

维护：该数据库过于庞大，还需要对新添加的主机频繁更新。

所以，DNS 被设计成了一个**分布式、层次数据库**

十五、DNS 服务器

为了处理扩展性问题，DNS 使用了大量的 DNS 服务器，它们以层次方式组织，并且分布在全世界范围内。

域名服务器是提供域名解析的服务器，在有基本的知识下，任何人都可以搭建域名服务器，甚至是根域名服务器，有名的软件有：BIND

目前的 DNS 服务器大致分为 3 种类型的 DNS 服务器：根 DNS 服务器、顶级域 DNS 服务器、权威 DNS 服务器

1、 根 DNS 服务器

主机名称	IP 地址	运营者	地区
a.root-servers.net	198.41.0.4, 2001:503:ba3e::2:30	VeriSign, Inc.	美国
b.root-servers.net	199.9.14.201, 2001:500:200::b	University of Southern California (ISI)	美国
c.root-servers.net	192.33.4.12, 2001:500:2::c	Cogent Communications	美国
d.root-servers.net	199.7.91.13, 2001:500:2d::d	University of Maryland	美国
e.root-servers.net	192.203.230.10, 2001:500:a8::e	NASA (Ames Research Center)	美国
f.root-servers.net	192.5.5.241, 2001:500:2f::f	Internet Systems Consortium, Inc.	美国
g.root-servers.net	192.112.36.4, 2001:500:12::d0d	US Department of Defense (NIC)	美国
h.root-servers.net	198.97.190.53, 2001:500:1::53	US Army (Research Lab)	美国
i.root-servers.net	192.36.148.17, 2001:7fe::53	Netnod	瑞典
j.root-servers.net	192.58.128.30, 2001:503:c27::2:30	VeriSign, Inc.	美国
k.root-servers.net	193.0.14.129, 2001:7fd::1	RIPE NCC	荷兰
l.root-servers.net	199.7.83.42, 2001:500:9f::42	ICANN	美国
m.root-servers.net	202.12.27.33, 2001:dc3::35	WIDE Project	日本

因特网上有 13 个根 DNS 服务器（标号 A 到 M），1 个为主根服务器在美国。其余 12 个均为辅根服务器，其中 9 个在美国，欧洲 2 个，位于英国和瑞典，亚洲 1 个位于日本。这里的个并不是指物理意义上的单个服务器，它是一个逻辑概念，根 DNS 服务器可以由分布在全球的多个服务器组成，形成一个集群，对外统一为 1 台逻辑的根 DNS 服务器（即每个标号下的根 DNS 服务器的 IP 地址是一样的）。而实际物理意义上的根 DNS 服务器，已超过千台

https://root-servers.org/ 这里可以查到所有根 DNS 服务器的分布

截止发这篇文章之前，中国有 19 台根 DNS 服务器，分别位于：

- 北京：F，I，J，L
- 杭州：J
- 澳门：F
- 香港：A、D、E、F、F、I、J
- 台北：E、F、F、I、K、L

2、顶级域（TLD）DNS 服务器

这些服务器负责顶级域名如 com、org、net、edu 和 gov，以及所有国家的顶级域名如 uk、fr、ca 和 cn

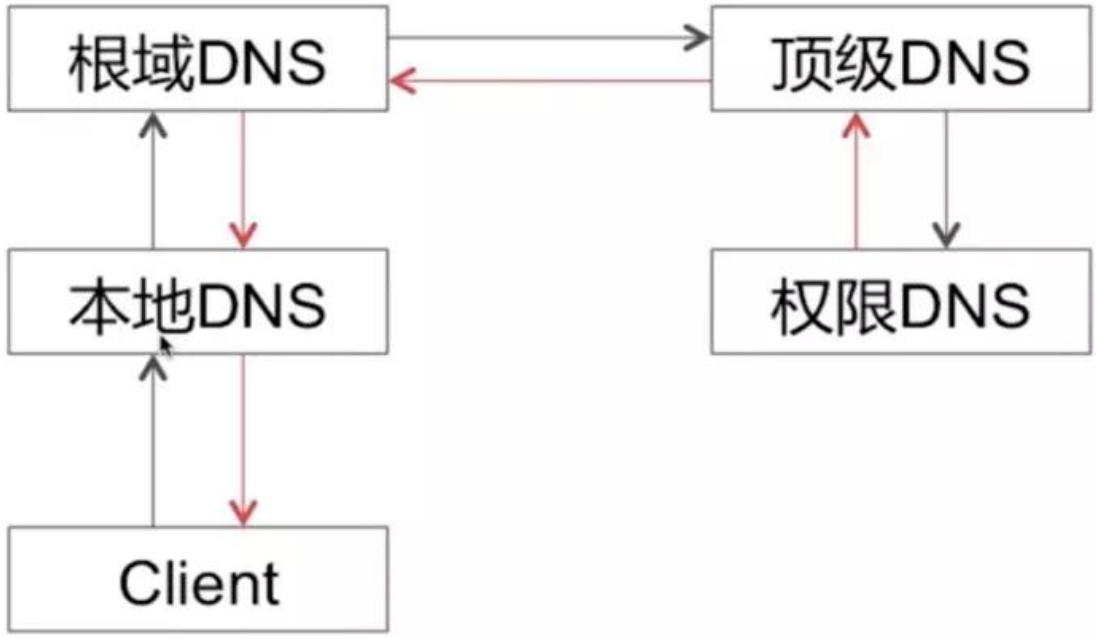
3、权威 DNS 服务器

在因特网上具有公共可访问主机的每个组织机构必须提供公共可访问的 DNS 记录，这些记录将这些主机的名字映射为 IP 地址。一个组织机构的权威 DNS 服务器收藏了这些 DNS 记录。

除此之外，还有一种很重要的 DNS，成为**本地 DNS 服务器**，其严格来说不属于该服务器的层次结构，但它对 DNS 层次结构是重要的。每个 **ISP**（互联网服务提供商），比如一个大学，一个公司或一个居民区的 ISP，都有一台本地 DNS 服务器。

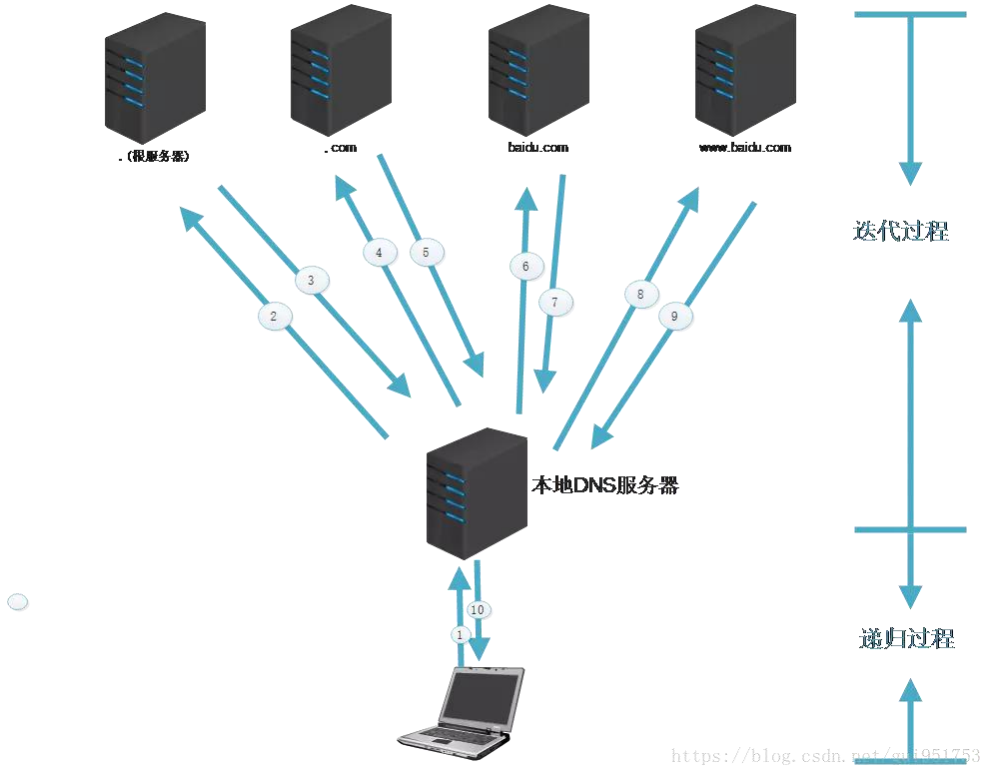
十六、DNS 解析过程

1、迭代查询和递归查询



很清晰地显示出了一条 DNS 查询链：本地 DNS 服务器-->根 DNS 服务器-->顶级域 DNS 服务器-->权威 DNS 服务器 ，所有查询都是递归的。

这是**递归查询**。

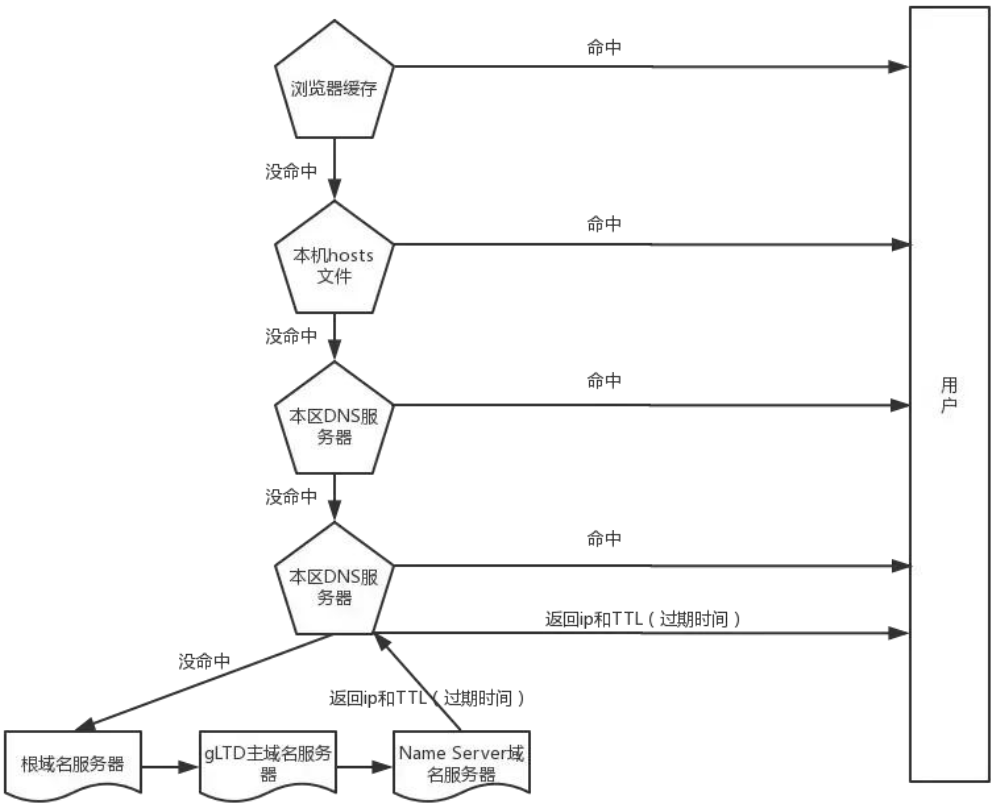


这种利用了**迭代查询**和**递归查询**，从 Client 与本地 DNS 之间是**递归查询**，其余则是**迭代查询**。
所谓 **递归查询**过程 就是“查询的递交者”更替，而 **迭代查询**过程 则是“查询的递交者”不变。
从理论上讲，任何 DNS 查询既可以是迭代的也能是递归的。
而在实际过程中，更常用的是图上 **从请求主机到本地 DNS 服务器的查询是递归，其余查询是迭代的**这种方式。

2、DNS 缓存

DNS 缓存（DNS Caching）：为了改善时延性能并减少在因特网上到处传输的 DNS 报文数量，DNS 广泛使用了缓存技术。
在一个请求链中，当某 DNS 服务器接收一个 DNS 回答时，它能将该回答中的信息缓存在本地存储器中。那么另一个对相同主机名的查询到达该 DNS 服务器时，该 DNS 服务器就可以提供所要求的 IP 地址，即使它不是该主机名的权威服务器。
由于 IP 和主机名的映射并不是永久的，DNS 服务器在一段时间后就会丢弃缓存的信息。
本地 DNS 服务器也能够缓存 TLD 服务器的 IP 地址，从而允许本地 DNS 绕过查询链中的根 DNS 服务器。
而事实上，有 DNS 的地方，就有缓存。浏览器、操作系统、本地 DNS 服务器、根 DNS 服务器，它们都会对 DNS 结果做一定程度的缓存。

3、DNS 解析过程



大致分为 8 步：

- 1、发起基于域名的请求后，首先检查本地缓存（浏览器缓存-->操作系统的 hosts 文件）
- 2、如果本地缓存中有，直接返回目标 IP 地址，否则将域名解析请求发送给本地 DNS 服务器
- 3、如果本地 DNS 服务器中有，直接返回目标 IP 地址，到这一步基本能解析 80%的域名。如果没有，本地 DNS 服务器将解析请求发送给根 DNS 服务器
- 4、根 DNS 服务器会返回给本地 DNS 服务器一个所查询的 TLD 服务器地址列表
- 5、本地 DNS 服务器再向上一步返回的 TLD 服务器发送请求，TLD 服务器查询并返回域名对应的权威域名服务器的地址
- 6、本地 DNS 服务器再向上一步返回的权威域名服务器发送请求，权威域名服务器会查询存储的域名和 IP 的映射关系表，将 IP 连同一个 TTL（过期时间）值返回给本地 DNS 服务器
- 7、本地 DNS 服务器会将 IP 和主机名的映射保存起来，保存时间由 TTL 来控制
- 8、本地 DNS 服务器把解析的结果返回给用户，用户根据 TTL 值缓存在本地系统缓存中，域名解析过程结束

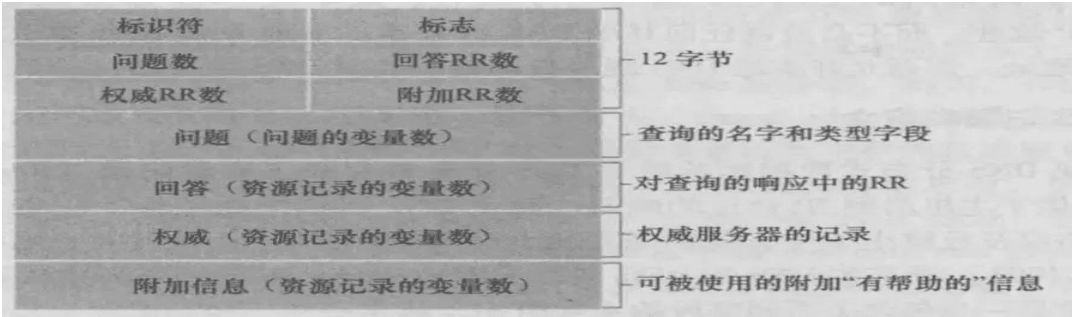
十七、DNS 记录和报文

1、资源记录

所有 DNS 服务器都存储了**资源记录（Resource Record，RR）**，其提供了主机名到 IP 的映射。
资源记录是一个包含以下字段的四元组：
（Name，Value，Type，TTL）
TTL 是该记录的生存时间，决定了资源记录应当从缓存中删除的时间。
Name 和 Value 的值取决于 Type(以下涉及的 foo,bar 均为伪变量)：

- Type = A，则 Name 是主机名，Value 使其对应的 IP 地址。这也是一个标准的主机名到 IP 地址的映射。如(replay1.bar.foo.com,145.37.93.126,A)
- Type = NS，则 Name 是个域（如 foo.com），而 Value 是个知道如何获取该域中主机 IP 地址的权威 DNS 服务器的主机名，如(foo.com,dns.foo.com,NS)
- Type = CNAME,则 Name 是别名为 Name 的主机对应的规范主机名。该记录能够向查询的主机提供一个主机名对应的规范主机名，如（foo.com,replay1.bar.foo.com,CNAME）
- Type = MX，则 Value 是个别名为 Name 的邮件服务器的规范主机名。如（foo.com,main.bar.foo.com,MX）

2、 DNS 报文



DNS 只有查询和回答两种报文，这两种报文格式是一样的。

- 前 12 个字节是**首部区域**。
  标识符用于标识该查询，这个标识符会被复制到对查询的回答报文中，以便让客户用它来匹配发送的请求和接收到的回答。
  标志字段中含有若干标志。1 比特的“查询/回答”标志位指出报文是查询报文（0）还是回答报文（1）。当某 DNS 服务器是所请求名字的权威 DNS 服务器时，1 比特的“权威的”标志位被置在回答报文中。此外，还有“希望递归”、“递归可用”等标志位。
  在首部中，还有 4 个数量相关的字段，指出来在首部后的 4 类数据区域出现的数量，其中 RR 是资源记录的意思。
- **问题区域**包含着正在进行的查询信息。该区域包括： 名字字段，指出正在被查询的主机名字； 类型字段，指出有关该名字的正被查询的问题类型，即上边说的四元组中的 Type
- **回答区域**包含了对最初请求的名字的资源记录。在回答区域中可以包含多条 RR，因此一个主机名理论上能够有多个 IP 地址（不同用户在不同地点访问同一个域名，可能会访问到不同的 IP 地址）
- **权威区域**中包含了其他权威服务器的记录
- **附加区域**包含了其他有帮助的记录

得知 DNS 的报文格式后，我们也就可以手动发送 DNS 查询包了。

十八、DNS 解析安全问题

1、DNS 劫持

一种可能的域名劫持方式即黑客侵入了宽带路由器并对终端用户的本地 DNS 服务器进行篡改，指向黑客自己伪造的本地 DNS 服务器，进而通过控制本地 DNS 服务器的逻辑返回错误的 IP 信息进行域名劫持。

另一方面，由于 DNS 解析主要是基于 UDP 协议，除了上述攻击行为外，攻击者还可以监听终端用户的域名解析请求，并在本地 DNS 服务器返回正确结果之前将伪造的 DNS 解析响应传递给终端用户，进而控制终端用户的域名访问行为。

2、缓存污染（DNS 污染）。

我们知道在接收到域名解析请求时，本地 DNS 服务器首先会查找缓存，如果缓存命中就会直接返回缓存结果，不再进行递归 DNS 查询。这时候如果本地 DNS 服务器针对部分域名的缓存进行更改，比如将缓存结果指向第三方的广告页，就会导致用户的访问请求被引导到这些广告页地址上。

3、如何解决 DNS 劫持？

DNS 解析发生在 HTTP 协议之前，DNS 解析和 DNS 劫持和 HTTP 没有关系，DNS 协议使用的是 UDP 协议向服务器的 53 端口进行请求。
要想解决 DNS 劫持：

- 可以使用 HttpDNS 的方案：**使用 HTTP 协议向 DNS 服务器的 80 端口进行请求**,来规避 DNS 劫持
  比如：`http://119.29.29.29/d?dn=domain&ip=clientIp`
- 在终端上，可以更换 DNS 服务器，不管手机还是电脑，都能手动配置 DNS

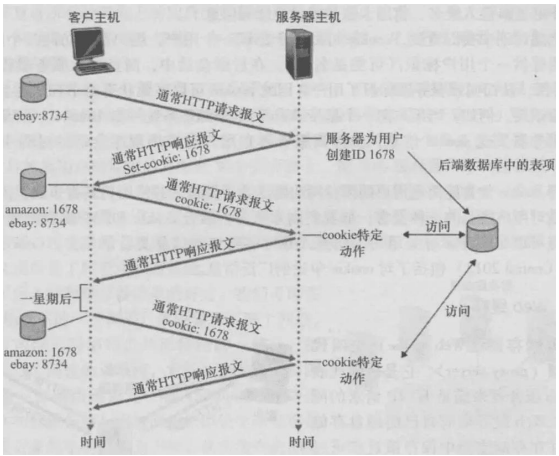
十九、Cookie

这里有说到，HTTP 协议是无状态的，服务器中没有保存客户端的状态，客户端必须每次带上自己的状态去请求服务器

基于 HTTP 这种特点，就产生了 cookie/session

1、用户与服务器的交互：Cookie

cookie 主要是用来记录用户状态，区分用户，**状态保存在客户端**。



- 1.首次访问 amazon 时，客户端发送一个 HTTP 请求到服务器端 。服务器端发送一个 HTTP 响应到客户端，其中包含 Set-Cookie 头部
- 2.客户端发送一个 HTTP 请求到服务器端，其中包含 Cookie 头部。服务器端发送一个 HTTP 响应到客户端
- 3.隔段时间再去访问时，客户端会直接发包含 Cookie 头部的 HTTP 请求。服务器端发送一个 HTTP 响应到客户端

cookie 技术有 4 个组件：

- 1.在 HTTP 响应报文中的一个 cookie 首部行
- 2.在 HTTP 请求报文中的一个 cookie 首部行
- 3.在用户端系统中保留一个 cookie 文件，并由用户的浏览器进行管理
- 4.位于 Web 站点的一个后端数据库

也就是说，cookie 功能需要浏览器的支持。如果浏览器不支持 cookie（如大部分手机中的浏览器）或者把 cookie 禁用了，cookie 功能就会失效。

2、cookie 的修改和删除

在修改 cookie 的时候，只需要新 cookie 覆盖旧 cookie 即可，在覆盖的时候，由于 Cookie 具有不可跨域名性，注意 name、path、domain 需与原 cookie 一致
删除 cookie 也一样，设置 cookie 的过期时间 expires 为过去的一个时间点，或者 maxAge = 0(Cookie 的有效期,单位为秒)即可

3、cookie 的安全

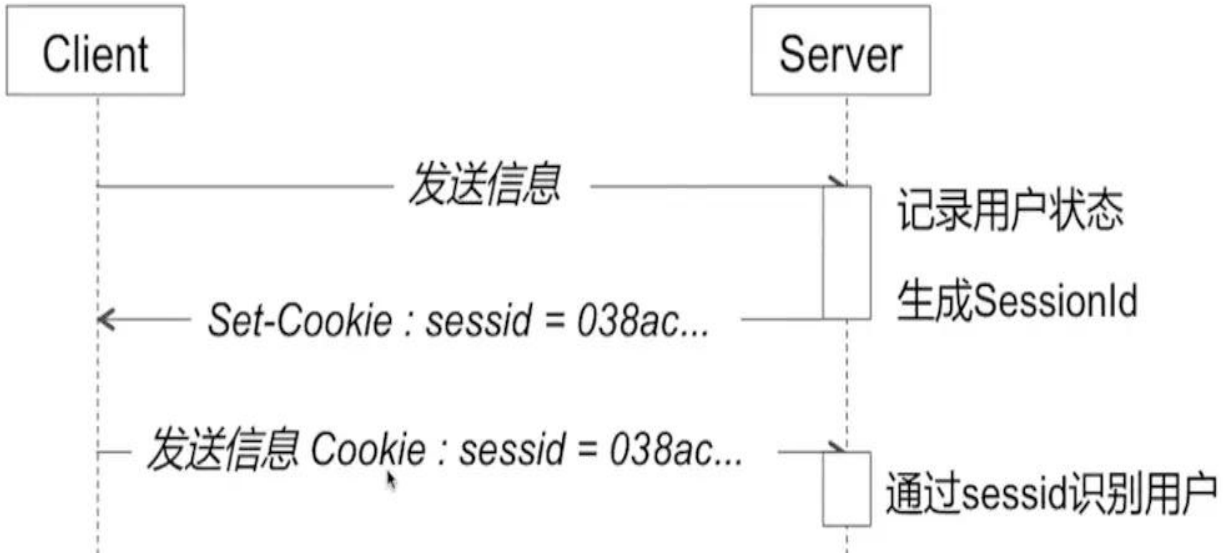
事实上，cookie 的使用存在争议，因为它被认为是对用户隐私的一种侵害，而且 cookie 并不安全
HTTP 协议不仅是无状态的，而且是不安全的。使用 HTTP 协议的数据不经过任何加密就直接在网络上传播，有被截获的可能。使用 HTTP 协议传输很机密的内容是一种隐患。

- 如果不希望 Cookie 在 HTTP 等非安全协议中传输，可以设置 Cookie 的 secure 属性为 true。浏览器只会在 HTTPS 和 SSL 等安全协议中传输此类 Cookie。
- 此外，secure 属性并不能对 Cookie 内容加密，因而不能保证绝对的安全性。如果需要高安全性，需要在程序中对 Cookie 内容加密、解密，以防泄密。
- 也可以设置 cookie 为 **HttpOnly**，如果在 cookie 中设置了 **HttpOnly** 属性，那么通过 js 脚本将无法读取到 cookie 信息，这样能有效的防止 **XSS（跨站脚本攻击）**攻击

二十、Session

除了使用 Cookie，Web 应用程序中还经常使用 **Session** 来记录客户端状态。Session 是服务器端使用的一种记录客户端状态的机制，使用上比 Cookie 简单一些，相应的也增加了服务器的存储压力。

Session 是另一种记录客户状态的机制，不同的是 **Cookie 保存在客户端浏览器中，而 Session 保存在服务器上**。
客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是 Session。客户端浏览器再次访问时只需要从该 Session 中查找该客户的状态就可以了。



- 当程序需要为某个客户端的请求创建一个 session 时，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 SessionId）
- 如果已包含则说明以前已经为此客户端创建过 session，服务器就按照 SessionId 把这个 session 检索出来，使用（检索不到，会新建一个）
- 如果客户端请求不包含 SessionId，则为此客户端创建一个 session 并且生成一个与此 session 相关联的 SessionId，SessionId 的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个 SessionId 将被在本次响应中返回给客户端保存。
- 保存这个 SessionId 的方式可以采用 cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。但 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 SessionId 传递回服务器。

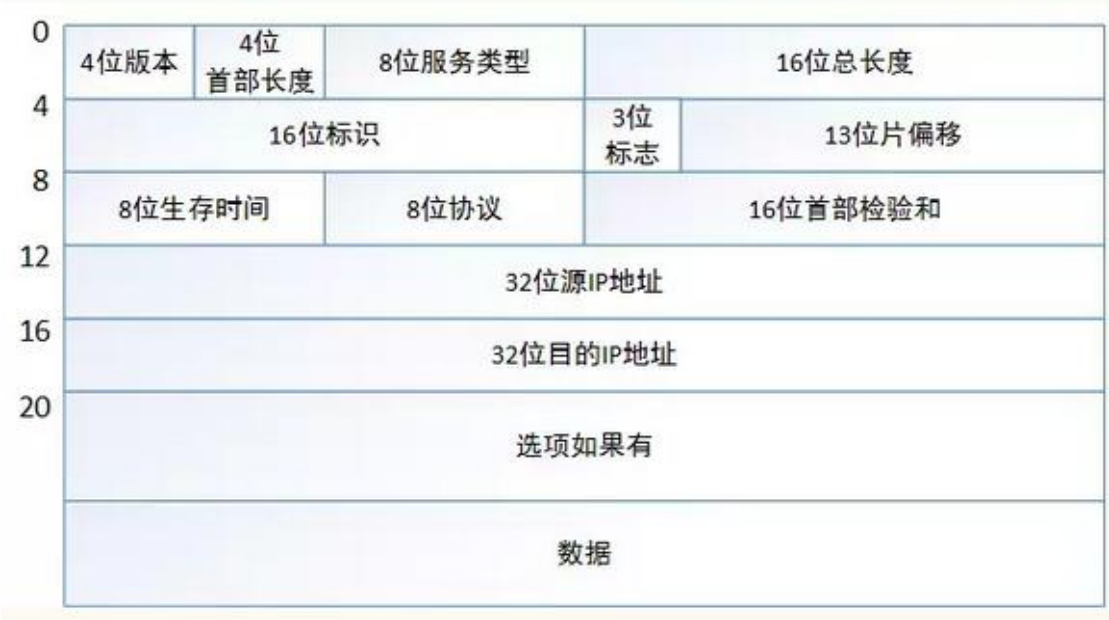
二十一、Cookie 和 Session 的区别：

1、cookie 数据存放在客户的浏览器上，session 数据放在服务器上。
2、cookie 相比 session 不是很安全，别人可以分析存放在本地的 cookie 并进行 cookie 欺骗,考虑到安全应当使用 session。
3、session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能,考虑到减轻服务器性能方面，应当使用 cookie。
4、单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。而 session 存储在服务端，可以无限量存储
5、所以：将登录信息等重要信息存放为 session;其他信息如果需要保留，可以放在 cookie 中

二十二、IP 协议

IP 协议是 TCP/IP 核心协议。

1、IP 协议的数据报格式（IPv4）



- 版本号
  规定了数据报的 IP 协议版本（IPv4 还是 IPv6）。不同的 IP 版本使用不同的数据报格式 ，
  上图是 IPv4 的数据报格式
- 首部长度
  大多数 IP 数据报不包含选项，所以一般 IP 数据报具有 20 字节的首部。
- 服务类型
  使不同类型的 IP 数据报能相互区别开来。
- 数据报长度
  整个 IP 数据报的长度，利用首部长度和总长度就可以是算出 IP 数据报中数据内容的起始地

址。该字段长度为 16 比特，所以 IP 数据报最长可达 2^16=65535 字节，而事实上，数据报很少有超过 1500 字节的

- 标识、标志、片偏移字段
  这三个字段与 IP 分片有关。此外，IPv6 不允许在路由器上对分组分片
- 生存时间 TTL
  用来确保数据报不会永远在网络中循环。设置该数据报可以经过的最多的路由器数。指定了数据报的生存时间，经过一个路由器，它的值就减 1，当该字段为 0 时，数据报就被丢弃
- 协议
  该字段只有在一个 IP 数据报到达其目的地才有用。该字段值指示了 IP 数据报的数据部分应交给哪个特定的传输层协议，比如，值为 6 表明要交给 TCP，而值为 17 则表明要交给 UDP
- 首部检验和
  与 UDP/TCP 的检验和不同，这个字段只检验数据报的首部，不包括数据部分。
- 选项字段
  是一个可变长字段，选项字段一直以 4 字节作为界限。这样就可以保证首部始终是 4 字节的整数倍。很少被用到
- 源 IP 和目的 IP

```
记录源IP地址，目的IP地址
```

二十三、IP 数据报分片

一个链路层帧能承载的最大数据量叫做**最大传送单元（Maximun Transmission Unit,MTU）**,即链路层的 MTU 限制着 IP 数据报的长度。
问题是在不同的链路上，可能使用不同的链路层协议，且每种协议可能具有不同的 MTU。
假定在某条链路上收到一个 IP 数据报，通过检查转发表确定出链路，且出链路的 MTU 比该 IP 数据报的长度要小，如何将这个过大的 IP 数据报压缩进链路层帧的有效载荷字段呢？

解决方案就是**分片**：将 IP 数据报中的数据分片为两个或更多个较小的 IP 数据报，用单独的链路层帧封装这些较小的 IP 数据报，然后向出链路上发送这些帧，每个这些较小的数据都称为**片（fragment）**。

片在到达目的地传输层前需要重新组装。

实际上，TCP 和 UDP 都希望从网络层上收到完整的未分片的报文。IPv4 的数据报重组工作是在端系统中，而不是在网络路由器中。
当一台目的主机从相同源收到一系列数据时，需要确定这些数据报中的某些是否是一些原来较大的数据报中的片。而如果是片的话，需要进一步确定何时收到最后一片，并且如何将这些片拼接到一起以形成初始的数据报。从而就用到了前边说到的 IPv4 数据报首部中的*标识*、*标志*、*片偏移* 字段。

- 1、当生成一个数据报时，发送主机在为该数据报设置源和目的地址的同时在贴上标识号,发送主机通常将为它发送的每个数据报标识号加 1
- 2、当某路由器需要对一个数据报分片时，形成的每个数据报（即片）具有初始数据报的源地址、目的地址和标识号
- 3、当目的地从同一发送主机收到一系列数据报时，它能够检查数据报的标识号以确定哪些数据报实际上是同一较大数据报的片
- 4、由于 IP 协议是不可靠服务，一个或者多个片可能永远到达不了目的地。为了让目的主机绝对相信它已收到初始数据报的最后一个片，最后一个片的标志比特被设为 0，其余被设为 1
- 5、为了让目的主机确定是否丢失了一个片，并且能按照正确的顺序重新组装片，使用偏移字段指定该片应放在初始 IP 数据报的哪个位置>
  此外，如果有一个片或者多个片未能到达，则该不完整的数据报将会被丢弃且不会交给传输层。但如果传输层正使用着 TCP，则 TCP 将通过让源以初始数据来重传数据。因为 IP 层没有超时重传机制，所以会重传整个数据报，而不是某个片

二十四、IPv4 编址

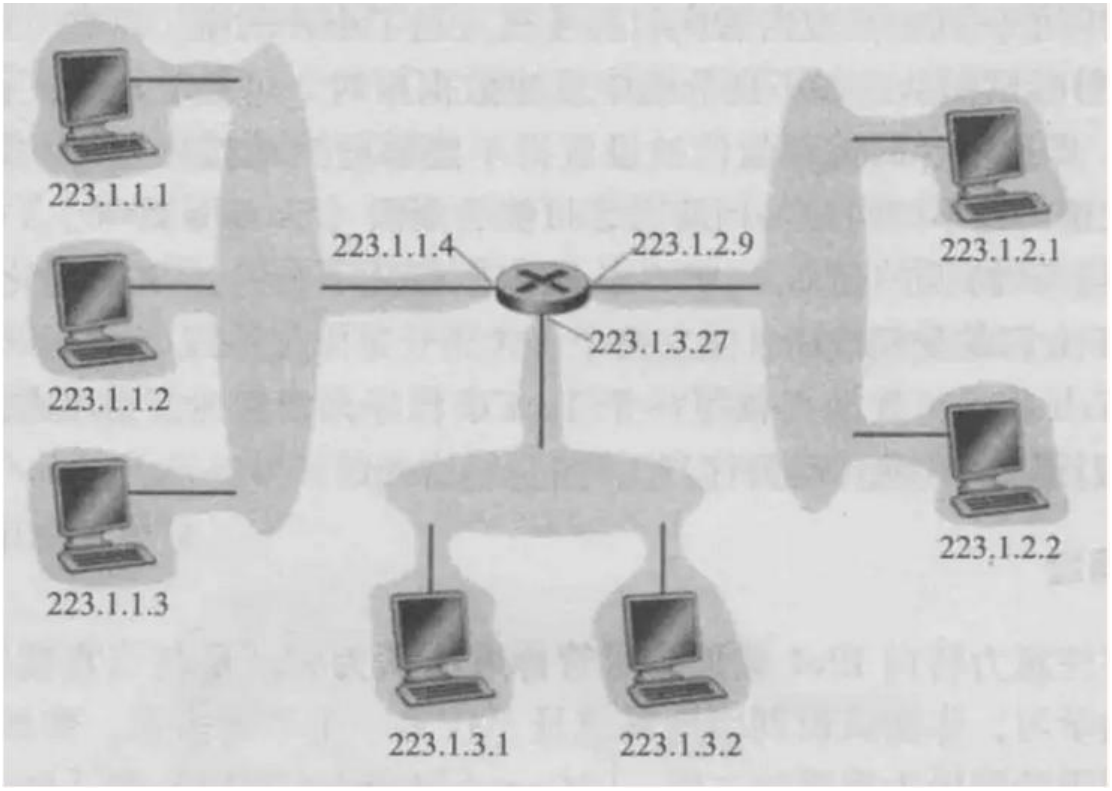
1、IP 地址

一台主机通常只有一条链路连接到网络，当主机上的 IP 想发送一条数据报时，就在该链路上发送。主机与物理链路之间的边界叫做**接口（interface）**。
而路由器的任务是从链路上接收数据报并从某些其他链路转发出去，路由器必须有两条或更多链路与其连接，路由器与它的任意一条链路之间的边界也叫做接口。即一台路由器会有多个接口，每个接口有其链路。

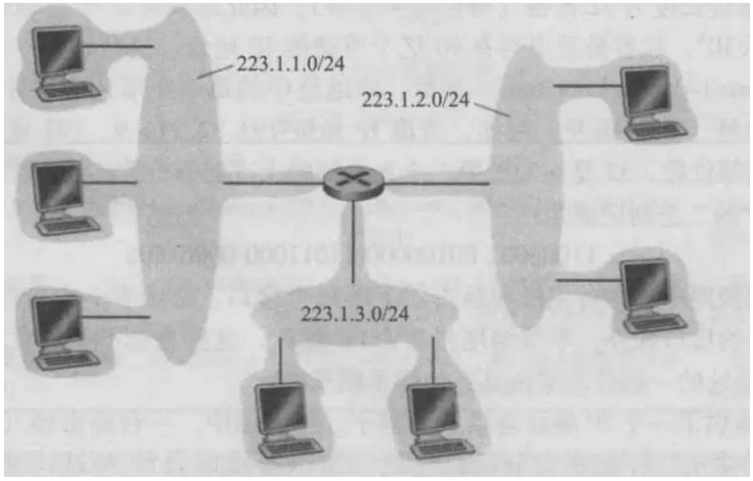
因为每台主机与路由器都能发送和接收 IP 数据报，IP 要求每台主机和路由器接口都有自己的 IP 地址。因此，一个 IP 地址技术上是与一个接口相关联的，而不是与包括该接口的主机或路由器相关联的。

2、子网

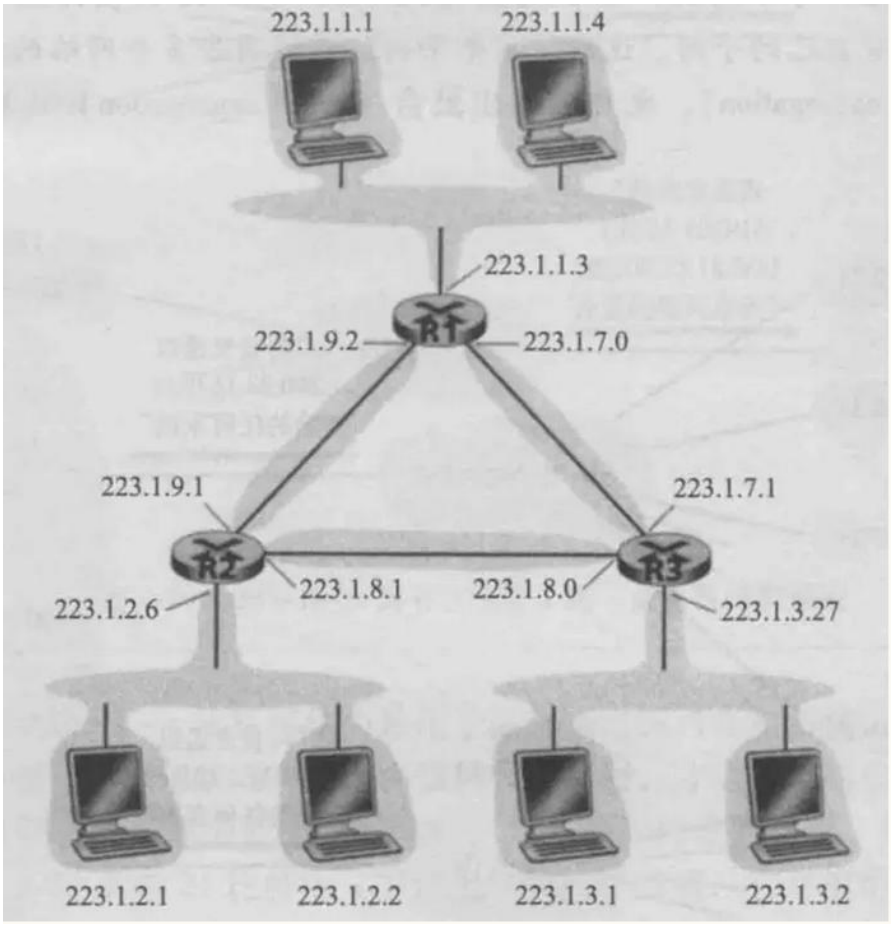
每个 IP 地址（IPv4）长度为 32 比特（4 字节），按**点分十进制记法**书写，即地址中的每个字节都用它的十进制形式书写，各字节间以点.隔开，比如 193.32.122.30
在因特网上的每台主机和路由器上的每个接口，必须有一个全球唯一的 IP 地址（NAT 后的接口除外）。这些地址不能自由选择，一个接口的 IP 地址的一部分需要由其连接的子网来决定。



如图，一台路由器有三个接口，连接 7 台主机。左侧的三台主机以及连接它们的路由器的接口，都有一个形如 223.1.1.x 的 IP 地址。即在它们的 IP 地址中，最左侧的 24 比特是相同的。



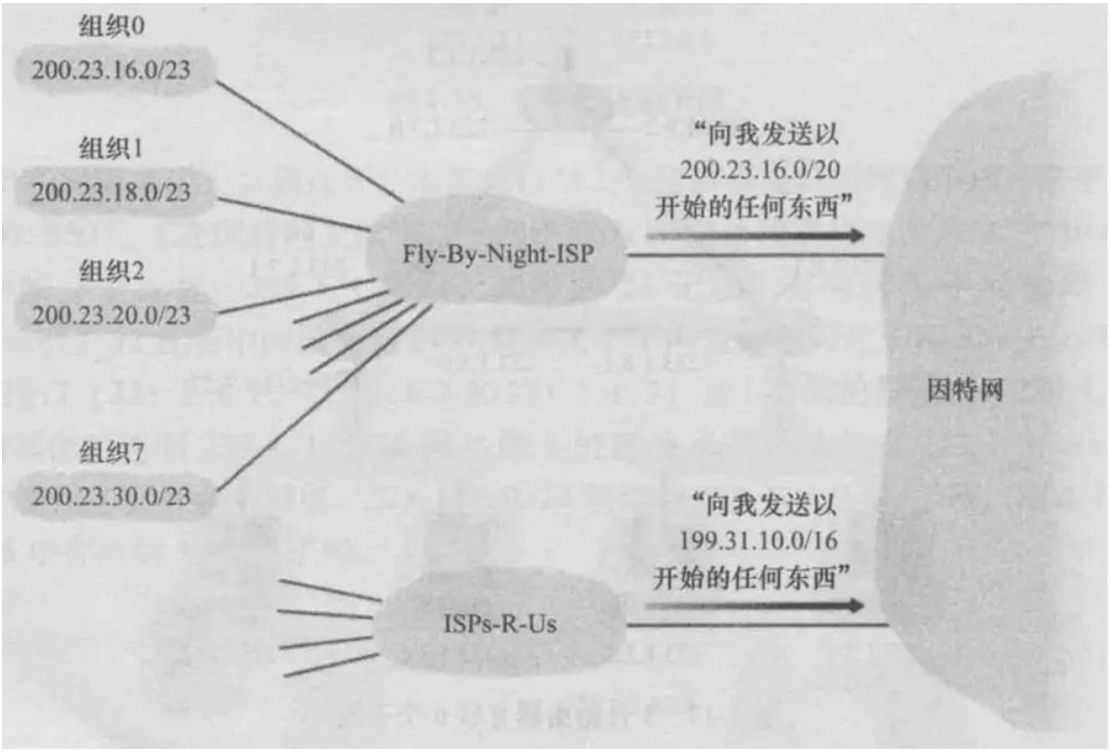
互联左侧这三个主机接口与 1 个路由器接口的网络形成 1 个**子网（subnet）**（也被称为 IP 网络或直接称为网络）。IP 编址为这个子网分配一个地址：223.1.1.0/24，其中的/24 记法，有时称为**子网掩码（network mask）**，指示了 32 比特中的最左侧 24 比特定义了子网地址。任何连接到该子网的主机都要求其地址具有 223.1.1.x 的形式。同样图中下侧和右侧也是子网，分别为 223.1.3.0/24 和 223.1.2.0/24



上图显示了 3 台通过点对点链路彼此互联的路由器，这里出现了 6 个子网。
一个具有多个以太网段和点对点链路的组织将具有多个子网，在给定子网上的所有设备都具有相同的子网地址。
虽然在理论上来说，不同子网可以有完全不同的子网地址。但上图可以看出，这 6 个子网在前 16 个比特是一致的，都是 223.1

3、无类别域间路由选择（CIDR）

因特网的地址分配策略被称为**无类别域间路由选择（CIDR）**(也被称为无分类编址，以区分于分类编址)。对于子网寻址，32 比特的 IP 地址被分为两部分，也是点分十进制数形式 a.b.c.d/x，其中 x 指示了地址的第一部分中的比特数，又被称为该地址的**前缀（prefix）**。
一个组织通常被分配一块连续的地址，即具有相同前缀的地址。在该组织外的的路由器仅考虑前面的前缀比特 x，这相当大地减少了在这些路由器中转发表的长度，形式为 a.b.c.d/x 单一表项足以将数据报转发到该组织内的任何目的地。



如图，200.23.16.0/20 下有 8 个组织，分别是 200.23.16.0/23 到 200.23.30.0/23，每个组织有自己的子网。而外界不需要知道这 8 个组织。这种使用单个网络前缀通告多个网络的能力通常称为**地址聚合**，也称为**路由聚合**或**路由摘要**

4、分类编址

在 CIDR 被采用之前，IP 地址的网络部分被限制长度为 8、16、24 比特，也就是**分类编址（classful addressing）**。具有 8、16、24 比特子网地址的子网被称为 A、B 和 C 类网络。
一个 C 类（/24）子网既能容纳 *2 的 8 次方 - 2 = 254* 台主机（其中两个地址预留用于特殊用途），这对于很多组织来说都太小了。
而一个 B 类（/16）子网可支持多达 *2 的 16 次方 - 2 = 65534*
台主机，又太大了。
在分类编址方法下，一个有 2000 台主机的组织通常被分给一个 B 类（/16）地址，那么剩下的 6 万多个地址就浪费掉了。这就会导致 **B 类地址空间的迅速损耗以及所分配的地址空间的利用率低**。

此外，255.255.255.255 是 IP 广播地址，当一台主机发出一个目的地址为该地址的数据报时，该报文会交付给同一个网络中的所有主机。

5、获取主机地址

某组织一旦获得了一块地址，它就可为本组织内的主机与路由器逐一分配 IP 地址。系统管理员通常手工配置路由器中的 IP 地址。主机地址也能手动配置，但更多使用的是**动态主机配置协议（DHCP）**。DHCP 允许主机自动获取 IP 地址。网络管理员可以配置 DHCP，以使某给定主机每次与网络连接时能得到一个相同的 IP 地址，或者某主机将被分配一个**临时的 IP 地址**，该地址在每次与网络连接时也许是不同的。

6、网络地址转换

每个 IP 地址（IPv4）长度为 32 比特（4 字节），因此总共有 2[图片上传失败...(image-524f6a-1565267828082)]

个可能的 IP 地址，约为 40 亿个。在互联网越来越普及的当下，个人计算机及智能手机等越来越多，这些 IP 地址显然无法满足人们的需求。
为了解决 IP 地址不足的问题，于是就有了**网络地址转换(Network Address Translation， NAT)**，它的思想就是给一个局域网络分配一个 IP 地址就够了，对于这个网络内的主机，则分配私有地址，这些私有地址对外是不可见的，他们对外的通信都要借助那个唯一分配的 IP 地址。

如果从广域网到达 NAT 路由器的所有数据报都有相同的目的 IP 地址，那么该路由器如何知道是发送给哪个内部主机的呢？其原理就是使用在 NAT 路由器上的一张 **NAT 转换表**，并在该表内包含了端口号及其 IP 地址。

- 假设一台主机向广域网请求数据，NAT 路由器收到该数据报，会为该数据报生成一个新的端口号替换掉源端口号，并将源 IP 替换为其广域网一侧接口的 IP 地址。当生成一个新的源端口号时，该端口号可以是任意一个当前未在 NAT 转换表中的源端口号（端口号字段是 16 比特，意味着 NAT 协议可以支持超过 60000 个并行使用路由器广域网一侧 IP 地址的连接），路由器中的 NAT 也在其 NAT 转换表中增加一表项。
- 该 NAT 路由器收到广域网返回的数据时，路由器使用目的 IP 地址与目的端口号从 NAT 转换表中检索出该主机使用的 IP 地址和目的端口号，改写该数据报的目的 IP 地址和目的端口号，并向该主机转发该数据报

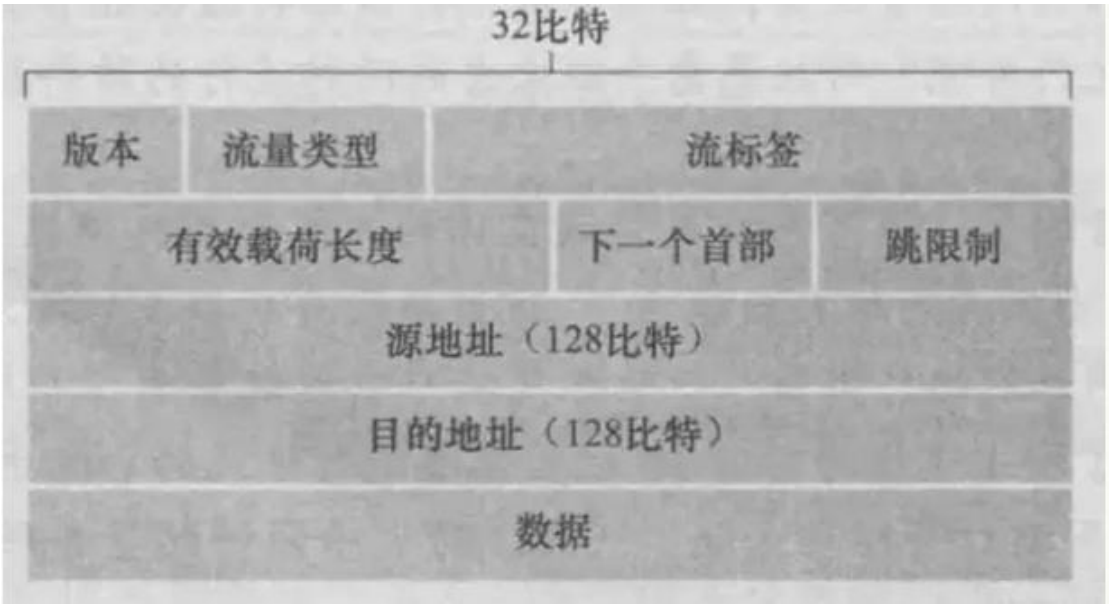
NAT 虽然在近几年得到了很广泛的应用，但也被很多人反对。
主要是：

- 1、端口号是用来进程编址的，而不是主机编址的（NAT 协议类似 NAT 路由器将该家庭网络的主机都当做进程处理，并通过 NAT 转换表为其分配端口号）
- 2、路由器通常仅应当处理高达第三层的分组
- 3、违背端到端原则，即主机彼此之间应当相互直接对话，结点不应当介入修改 IP 地址与端口号。
- 4、应当用 IPv6 来解决 IP 地址短缺问题

但不管反对与否，NAT 终究已成为当今因特网的一个重要组件

二十五、IPv6 数据报格式

1、IPv6 数据报格式



- 版本（4 比特）
  该字段用于标识 IP 版本号，IPv6 将该字段值设为 6。而如果将该字段设为 4 并不能创建一个合法的 IPv4 数据报
- 流量类型（8 比特）
  类似于 IPv4 数据报中的服务类型（TOS）
- 流标签（20 比特）
  流标签字段是 IPv6 数据报中新增的一个字段，用来标识一条数据报的流类型，以便在网络层区分不同的报文。
- 有效载荷长度（16 比特）

IPv6 数据报中在 40 定长字节数据报首部后的字节数量，即除了 IPv6 的数据报首部以外的其他部分的总长度

- 下一个首部（8 比特）
  当 IPv6 没有扩展报头时，该字段的作用和 IPv4 的协议字段一样。当含有扩展报头时，该字段的值即为第一个扩展报头的类型
- 跳限制（8 比特）
  与 IPv4 报文中的 TTL 字段类似，转发数据报的每台路由器将对该字段的内容减 1.如果跳限制计数到达 0，则该数据报将被丢弃
- 源地址和目的地址（各 128 比特）
  记录源 IP 地址，目的 IP 地址
- 数据

- 下一个首部（8 比特）
  当 IPv6 没有扩展报头时，该字段的作用和 IPv4 的协议字段一样。当含有扩展报头时，该字段的值即为第一个扩展报头的类型
- 跳限制（8 比特）
  与 IPv4 报文中的 TTL 字段类似，转发数据报的每台路由器将对该字段的内容减 1.如果跳限制计数到达 0，则该数据报将被丢弃
- 源地址和目的地址（各 128 比特）
  记录源 IP 地址，目的 IP 地址
- 数据

可以看出，在 IPv4 数据报中出现的几个字段在 IPv6 数据报中已不复存在：

- 分片/重新组装
  IPv6 不允许在中间路由器上进行分片和重新组装。这种操作只能在源与目的地上执行。如果路由器收到的 IPv6 数据报因太大不能转发出链路上的话，路由器会丢掉该数据报，并回一个“分组太大”的 ICMP 差错报文
- 首部检验和
  因为运输层和数据链路层协议执行了检验操作，该项功能在网络层就没有必要了，从而更快速处理 IP 分组
- 选项
  选项字段不再是标准 IP 首部的一部分了。但并没有消失，而是可能出现在 IPv6 首部中由“下一个首部”指出的位置上。即就像 TCP 或 UDP 协议首部能够是 IP 分组中的“下一个首部”，选项字段也能是“下一个首部”

IPv6 相对 IPv4 最重要的变化如下：

- 扩大的地址容量
  IPv6 将 IP 地址长度由 32 比特增加到 128 比特，这使得理论可存在的 IP 地址增加到 2[图片上传失败...(image-a67865-1565268140183)]
- 个，约 340 万亿亿亿亿个，这是一个非常大的数字，确保全世界再也不会用尽 IP 地址，甚至可以为地球上每一粒沙子都分配一个唯一的 IP 地址
  除了单播和多播地址外，IPv6 没有广播这一说法，而是引入了一种称为**任播地址**的新型地址，这种地址可以使数据报交付给一组主机中的任意一个
- 简化高效的 40 字节首部
  除去共 32 字节的源地址和目标地址外，首部其余字段只占了 8 字节
- 流标签与优先级
  给属于特殊流的分组打上标签，这些特殊流是发送方要求进行特殊处理的流，如一种非默认服务质量或需要实时服务的流

2、IPv6 书写和表达方式

表述和书写时，把长度为 128 比特的 IPv6 地址分成 8 个 16 位的二进制段、每一个 16 位的二进制段用 4 位的 16 进制数表示，段间用“：”（冒号）隔开（其书写方法和 IPv4 的十进制数加“.”不同）。

例如：1000:0000:0000:0000:000A:000B:000C:000D 就是每一个 16 位的二进制数的段用 4 位 16 进制数的段来表示、段间用“：”（冒号）隔开的一个 IPv6 地址；其中：各个 4 位 16 进制数的段中的高位 0 允许省略；因此，上面的 IPv6 地址也可以缩写成：1000:0:0:0:A:B:C:D。

为了更进一步简化，IPv6 的地址规范中还规定，可以在一个 IPv6 地址中**最多使用一次双冒号**（::）来取代 IPv6 地址中紧密相连的多个全 0 的 16 进制数的段（因为如果允许在一个 IPv6 地址中使用一次以上的双冒号时将无法判断 IPv6 地址的长度，所以 IPv6 的地址规范中才规定：在一个 IPv6 地址中最多只能使用一次双冒号），这样上面的 IPv6 地址还可以缩写成：1000::A:B:C:D。

双冒号使用的地点可以在 IPv6 地址的前面、后面或者是中间；例如：对于 1000:0:0:0:A:B:0:0 这样的一个 IPv6 地址，可以写成 1000::A:B:0:0，也可以写成 1000:0:0:0:A:B::；但是不能写成 1000::A:B::。

带有端口号的 IPV6 地址字符串形式，地址部分应当用“[]”括起来，在后面跟着‘：’带上端口号，如 [A01F::0]:8000

二十六、从 IPv4 到 IPv6 的迁移

基于 IPv4 的公共因特网如何迁移到 IPv6 呢？这是个非常现实的问题
虽然 IPv6 使能系统可做成向后兼容，即能接收、发送和路由 IPv4 数据报，但已部署的 IPv4 使能系统却不能处理 IPv6 数据报

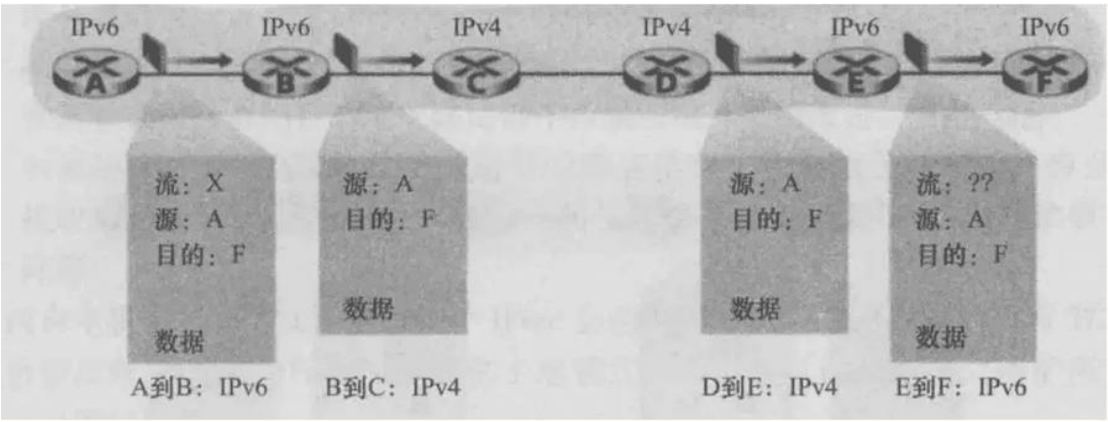
1、双协议栈

引入 IPv6 使能结点的最直接方式是**双栈**方法，即使用该方法的 IPv6 结点还有完整的 IPv4 实现，即 **IPv6/IPv4 结点**，具有接收和发送 IPv4 和 IPv6 两种数据报的能力。
当与 IPv4 结点互操作时，IPv6/IPv4 结点可使用 IPv4 数据报；当与 IPv6 结点互操作时，IPv6/IPv4 结点又可使用 IPv6 数据报。

IPv6/IPv4 结点必须有 IPv6 与 IPv4 两种地址。此外，它们还必须能确定另一个结点是否是 IPv6 使能的或仅 IPv4 使能的。

可以使用 **DNS** 来解决，若要解析的结点名字是 IPv6 使能的，则 DNS 会返回一个 IPv6 地址，否则返回一个 IPv4 地址。如果发出 DNS 请求的结点是仅 IPv4 使能的，则只返回一个 IPv4 地址。

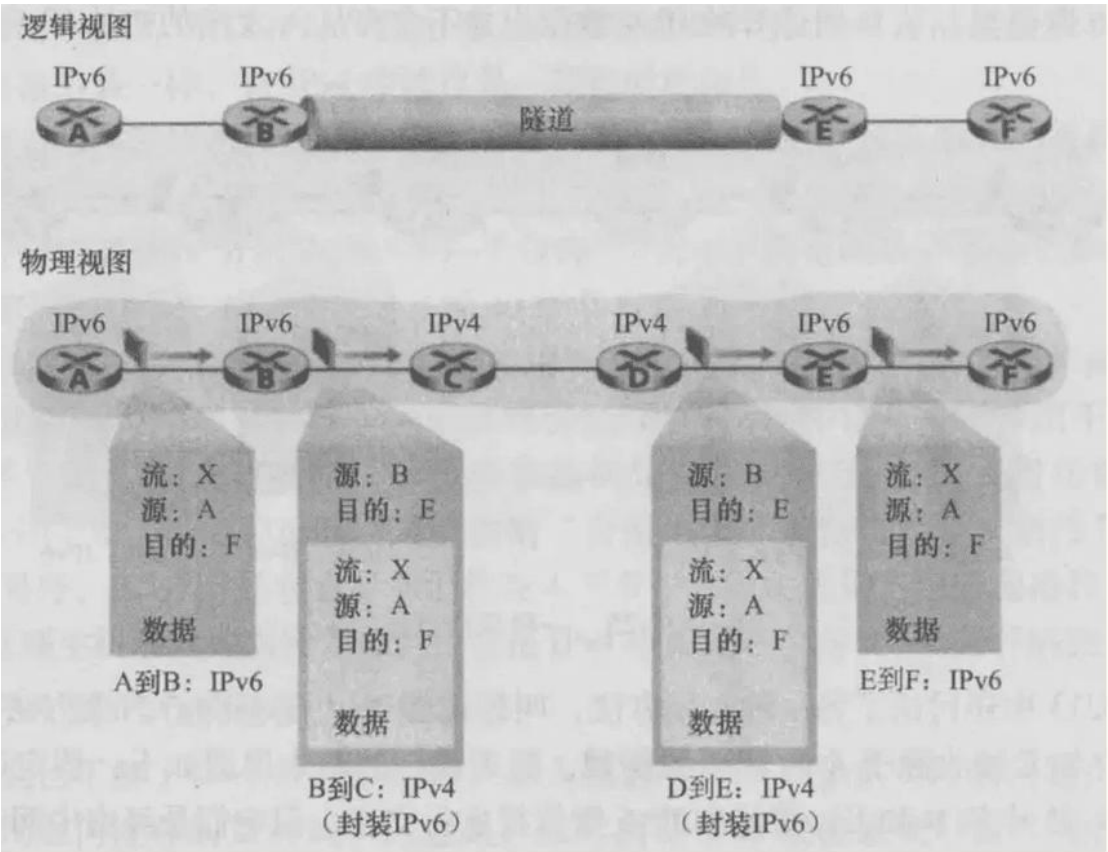
两个 IPv6 使能的结点不应相互发送 IPv4 数据报，而如果发送方或接收方任意一个仅为 IPv4 使能的,则必须使用 IPv4 数据报。
这样就会有下面这种情况：



如图，假如结点 A、B、E、F 都是 IPv6 使能的结点，而结点 C 和 D 是仅 IPv4 使能的结点，那么当按 A->B->C->D->E->F 顺序发送数据报时,AB 之间会发 IPv6 数据报，BC 会发 IPV4 数据报， 由于 IPv6 数据报特定的字段在 IPv4 数据报中无对应的部分，这些字段将会丢失。因此，即使 E 和 F 之间能发 IPv6 数据报，从 D 到达 E 的 IPv4 数据报并未含有从 A 发出的初始 IPv6 数据报中的所有字段。

2、隧道

建隧道是另一种双栈方法，该方法能解决上述问题。
假定两个 IPv6 结点要使用 IPv6 数据报进行交互，但是它们是经由中间 IPv4 路由器互联的。将两台 IPv6 路由器中间的 IPv4 路由器的集合成为一个隧道，如 B->C->D->E。



如图，借助于隧道，在隧道发送端的 IPv6 结点可将整个 IPv6 数据报放到一个 IPv4 数据报的数据字段中。于是，该 IPv4 数据报的地址设为指向隧道接收端的 IPv6 结点，再发送给隧道中的第一个结点。而隧道中的 IPv4 路由器在它们之间为该数据报提供路由，就像对待其他 IPv4 数据报一样，完全不知道该数据报自身就含有一个完整的 IPv6 数据报。而隧道接收端的 IPv6 结点最终收到该 IPv4 数据报，并确定该 IPv4 数据报中含有一个 IPv6 数据报，于是提取出该 IPv6 数据报，然后再为该 IPv6 数据报提供路由

3、NAT-PT

除了双栈和隧道方案外，还有一种 **NAT-PT(Network Address Translator - Protocol Translator)附带协议转换器的网络地址转换器**方案