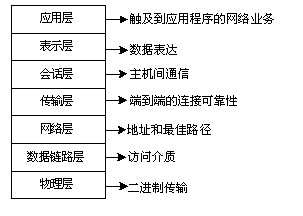
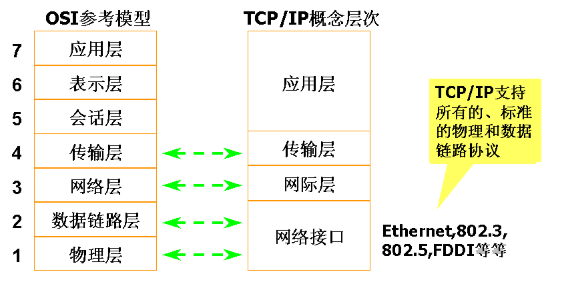
**1、网络体系结构**

有两种模型，一种是7层网络模型，也就是OSI模型，一种是4层模型，也就是TCP/IP模型。

前者模型是从底向上一步一步进行抽象，分别是物理层、数据链路层、网络层、传输层、会话层、表示层，应用层。具体如下图表示，高层是底层的一种抽象。



4层模型是将7层模型进行了浓缩，将底下两层，合并成为了数据链路层，网络层和传输层单独，最上面3层合并成为了应用层，具体对应关系如下图所示



一般现在用到的都是TCP/IP 4层网络模型，和7层模型相比，要简单不少，同时7层模型分层分的太过于细腻，导致实现起来也比较麻烦，而4层模型已经可以较好的满足现实世界中的业务模型，因此TCP/TP 4 层模型成为了事实上的标准。

**2、TCP和UDP有什么区别**

最基本的是基于连接和无连接的区别，这是因为协议的目的是不一样的。TCP实现的是可靠传输，因此就需要进行拥塞控制和流量控制，超时重传，乱序分包进行重排等相关实现。而UDP是没有连接的概念，只管发送出去，也不管是否可达或者是是否可以收到回应报文。因此这就体现在TCP报文首部需要20个字节以上，而UDP报文首都需要8个字节就可以了。

TCP与UDP区别总结：  
1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接  
2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保   证可靠交付  
3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的。UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）  
4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信  
5、TCP首部开销20字节;UDP的首部开销小，只有8个字节  
6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

**3、编写socket套接字的步骤**

基于TCP的socket通信模型

服务器：  
1.创建套接字描述符（socket）  
2.设置服务器的IP地址和端口号（需要转换为网络字节序的格式）  
3.将套接字描述符绑定到服务器地址（bind）  
4.将套接字描述符设置为监听套接字描述符（listen），等待来自客户端的连接请求，监听套接字维护未完成连接队列和已完成连接队列  
5.从已完成连接队列中取得队首项，返回新的已连接套接字描述符（accept），如果已完成连接队列为空，则会阻塞  
6.从已连接套接字描述符读取来自客户端的请求（read）  
7.向已连接套接字描述符写入应答（write）  
8.关闭已连接套接字描述符（close），回到第5步等待下一个客户端的连接请求  
客户端：  
1.创建套接字描述符（socket）  
2.设置服务器的IP地址和端口号（需要转换为网络字节序的格式）  
3.请求建立到服务器的TCP连接并阻塞，直到连接成功建立（connect）  
4.向套接字描述符写入请求（write）  
5.从套接字描述符读取来自服务器的应答（read）  
6.关闭套接字描述符（close）

基于UDP套接字编程  
服务器：  
1.创建套接字描述符（socket）  
2.设置服务器的IP地址和端口号（需要转换为网络字节序的格式）  
3.将套接字描述符绑定到服务器地址（bind）  
4.从套接字描述符读取来自客户端的请求并取得客户端的地址（recvfrom）  
5.向套接字描述符写入应答并发送给客户端（sendto）  
6.回到第4步等待读取下一个来自客户端的请求  
客户端：  
1.创建套接字描述符（socket）  
2.设置服务器的IP地址和端口号（需要转换为网络字节序的格式）  
3.向套接字描述符写入请求并发送给服务器（sendto）  
4.从套接字描述符读取来自服务器的应答（recvfrom）  
5.关闭套接字描述符（close）

**4、TCP三次握手和四次挥手，以及各个状态的作用**

三次握手用于进行连接

第一次握手：建立连接时，客户端发送SYN包，例如seq = 200 ，SYN = 1 （200是随机产生的一个值，TCP规定SYN = 1的时候不能携带数据）给服务器，然后Socket进入SYN\_SENT状态

第二次握手：服务器收到SYN报文段进行确认，发一个将SYN=1，ACK = 1，seq = 300，ack = 201（ack确认号为收到的序列号+1，ACK=1表示确认号有效，建立成功全程都是1，300是随机生成的）

第三次握手：客户端再进行一次确认，发一个ACK = 1，seq = 201，SYN = 0，ack = 301，ACK = 1（seq为第一次发送的seq+1，SYN从此都是0，ack为服务器确认的seq+1，ACK从此都是1）

建立成功 之后就可以相互发送数据报文了。

四次挥手用于断开连接

tcp四次挥手，由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

第一次挥手: 机1向主机2发送FIN报文段,表示关闭数据传送,并主机1进入FIN\_WAIT\_1状态,表示没有数据要传输了

第二次挥手：机2接受到FIN报文后进入CLOSE\_WAIT状态(被动关闭),然后发送ACK确认,表示同意了主机1的关闭请求，这个时候主机1进入到FIN\_WAIT\_2状态，这个状态是相对来讲稍微持久一点的。

第三次挥手：机2等待主机1发送完数据,发送FIN到主机1请求关闭,主机2进入LAST\_ACK状态,

第四次挥手：机1收到主机2发送的FIN后,回复ACK确认到主机2,主机1进入TIME\_WAIT状态,主机2收到主机1的ACK后就关闭连接了,状态为CLOSED,主机1等待2MSL,仍然没有收到主机2的回复,说明主机2已经正常关闭了,主机1关闭连接

MSL（Maximum Segment Lifetime）：报文最大生存时间，是任何报文段被丢弃前在网络内的最长时间。当主机1回复主机2的FIN后，等待(2-4分钟)，即使两端的应用程序结束。

**5、Http协议和TCP的区别**

　TPC/IP协议是传输层协议，主要解决数据如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据。Http协议底层就是使用TCP协议来进行实现的。Http协议帮我们实现了消息边界的划分，因为TCP协议是流式协议，消息是没有边界的，如果希望消息有边界，就需要自己进行消息边界的划分，常用的有两种方式，一种是以\r\n作为消息结束的标志，FTP协议就是使用这种方式，另外一种是包长+包体的形式。HTTP协议采用的混合模式，头部信息是通过\n来进行消息边界的划分，数据信息采用的是包长+包体的格式来进行消息边界的划分。

**6、同步IO和异步IO的区别**

同步IO，就是，如果有数据就返回，没有数据就等待,直至数据到来这种模式。网络编程中的read函数就是同步IO。

异步IO是，你只管调用，调用完走就可以了，操作系统会帮你把数据处理好，然后在数据处理好之后会通知你，这就是异步IO。Linux系统中的aio\_read函数就是异步IO函数。

**7、什么是IOCP**

IOCP全称IO Completion Port, 是window上的一种高效的异步通许模型，是真正的异步通许模型。提到IOCP就要提到epool了。以epoll为核心的reactor模型和以IOCP为核心的proactor模型，是两种不同的通讯模型。Reactor核心就是监听事件的发生，如果事件发生会通知你进行后续的处理，而Proactor模型的核心是，在你监听事件的同时，告诉它应该怎么处理，比如说读数据应该读取到哪里，这就需要你提前开辟好内存空间，然后当事件结束完成以后，会通知你，这个时候你就可以直接操作数据了，因为内核已经将数据从内核空间拷贝到了用户态空间。ASIO网络框架就是Proactor模型。

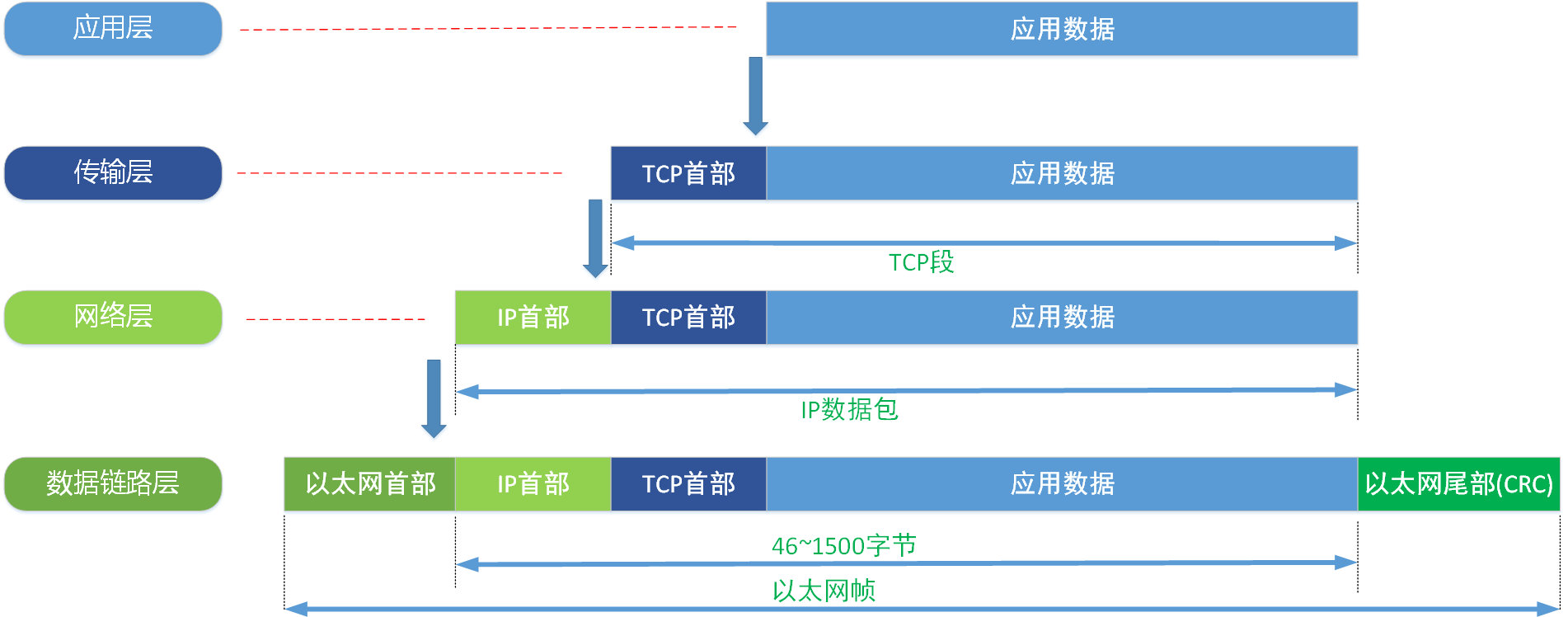
**8、什么是网络套接字(Socket)?流套接字(SOCK\_STREAM)基于什么协议。**

网络套接字是一个5元组，<源IP, 源端口，目的IP, 目的端口，协议>。只要这个5元组中有任何一个元素不相同个就认为是不同的部分，其中协议常见的是TCP或者是UDP协议。

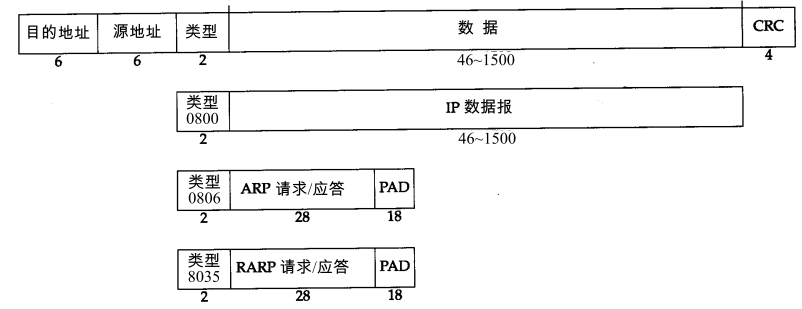
流套接字是基于TCP协议的，数据报套接字是基于UDP协议的。还有一种是raw socket，是可以自己进行协议包装的，比如说syn flood中ip地址伪造就是用的raw socket进行的。

**9、IP首部、TCP首部、UDP首部、以太网首部**

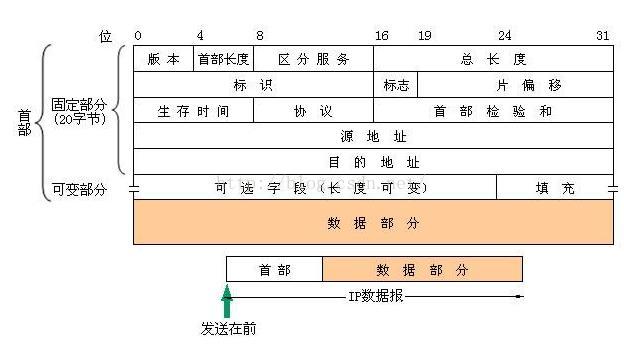
**首先是各种首部的关系：**



下面是以太网首部报文格式图：



下面是IP首部报文格式图：**20字节**



**1、第一个4字节（也就是第一行）：**

（1）版本号（Version），4位；用于标识IP协议版本，IPv4是0100，IPv6是0110，也就是二进制的4和6。

（2）首部长度（Internet Header Length），4位；用于标识首部的长度，单位为4字节，所以首部长度最大值为：(2^4 - 1) \* 4 = 60字节，但一般只推荐使用20字节的固定长度。

（3）服务类型（Type Of Service），8位；用于标识IP包的优先级，但现在并未使用。

（4）总长度（Total Length），16位；标识IP数据报的总长度，最大为：2^16 -1 = 65535字节。

**2、第二个四字节：**

（1）标识（Identification），16位；用于标识IP数据报，如果因为数据链路层帧数据段长度限制（也就是MTU，支持的最大传输单元），IP数据报需要进行分片发送，则每个分片的IP数据报标识都是一致的。

（2）标识（Flag），3位，但目前只有2位有意义；最低位为MF，MF=1代表后面还有分片的数据报，MF=0代表当前数据报已是最后的数据报。次低位为DF，DF=1代表不能分片，DF=0代表可以分片。

（3）片偏移（Fragment Offset），13位；代表某个分片在原始数据中的相对位置。

**3、第三个四字节：**

（1）生存时间（TTL），8位；以前代表IP数据报最大的生存时间，现在标识IP数据报可以经过的路由器数。

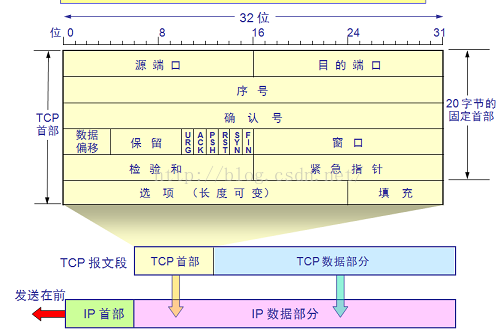
（2）协议（Protocol），8位；代表上层传输层协议的类型，1代表ICMP，2代表IGMP，6代表TCP，17代表UDP。

（3）校验和（Header Checksum），16位；用于验证数据完整性，计算方法为，首先将校验和位置零，然后将每16位二进制反码求和即为校验和，最后写入校验和位置。

**4、第四个四字节：源IP地址**

**5、第五个四字节：目的IP地址**

下面是TCP首部报文格式图：**20个字节**



**1、第一个4字节：**

（1）源端口，16位；发送数据的源进程端口

（2）目的端口，16位；接收数据的进程端口

**2、第二个4字节与第三个4字节**

（1）序号，32位；代表当前TCP数据段第一个字节占整个字节流的相对位置；

（2）确认号，32位；代表接收端希望接收的数据序号，为上次接收到数据报的序号+1，当ACK标志位为1时才生效。

**3、第四个4字节：**

（1）数据偏移，4位；实际代表TCP首部长度，最大为60字节。

（2）6个标志位，每个标志位1位；

SYN，为同步标志，用于数据同步；

ACK，为确认序号，ACK=1时确认号才有效；

FIN，为结束序号，用于发送端提出断开连接；

URG，为紧急序号，URG=1是紧急指针有效；

PSH，指示接收方立即将数据提交给应用层，而不是等待缓冲区满；

RST，重置连接。

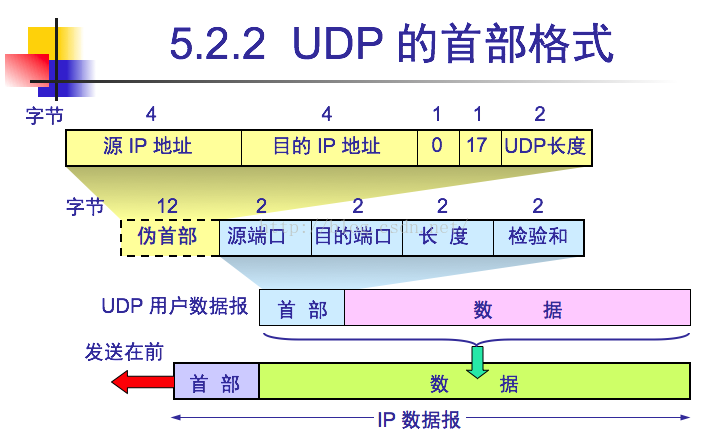
（3）窗口值，16位；标识接收方可接受的数据字节数。详解可参看：

**4、第五个4字节**

（1）校验和，16位；用于检验数据完整性。

（2）紧急指针，16位；只有当URG标识位为1时，紧急指针才有效。紧急指针的值与序号的相加值为紧急数据的最后一个字节位置。用于发送紧急数据。

下面是UDP首部报文格式图：8字节



**1. 源端口：**源端口号。在需要对方回信时选用，不需要时可用全0.   
**2. 目的端口：**目的端口号。这在终点交付报文时必须要使用到。   
**3. 长度：** UDP用户数据报的长度，其最小值是8（仅有首部）。   
**4. 检验和：**检测UDP用户数据报在传输中是否有错。有错就丢弃

**10、****TCP和UDP的应用场景**

从特点上，TCP 是可靠的但传输速度慢 ，UDP 是不可靠的但传输速度快。因此在选用具体协议通信时，应该根据通信数据的要求而决定。   
　　**若通信数据完整性需让位与通信实时性**，则应该选用 TCP 协议（如文件传输、重要状态的更新等）；反之，则使用 UDP 协议（如视频传输、实时通信等）。

**11、****如何实现可靠的UDP**

众所周知UDP通信协议是不可靠的通信协议，其可靠性必须由上层应用实现。一般都会采用**消息重传**来实现其可靠性，**采用消息重传的时候有两种方式，一种是发送者发起，另一种是接收者发起。**  
  
对于发送者发起的方式，一般情况下接收者会发送一个消息包的确认。发送者维护一个计时器并重传那些在某个确定的时间段里没有收到确认的消息包。这一类型的协议容易引起发送者溢出，因为要确认每一个发送的消息包。这种溢出现象被称为发送者（或者ACK）内爆。  
  
对于接收者发起的方式，通信双方的接收者负责错误检测。在这个方式里，序列号被用于检测消息包丢失。当检测到消息包丢失，接收者请求发送者重传消息包。采用这种方法，如果消息包没有到达任何一个接收者，发送者容易因NACK溢出。这会引起发送者的负载过高和过多的重传。这种现像被称为NACK内爆。Ramakrishnan et al.在1987年提出可以使用定时器来限制消息包重传，从而避免NACK内爆。在现实应用中这种方式使用得较多。

**12、详细说明TCP状态迁移过程**

**11种状态：**

1.CLOSED: 这个没什么好说的了，表示**初始状态**。

2.LISTEN: 这个也是非常容易理解的一个状态，表示**服务器端的某个SOCKET处于监听状态，可以接受连接**了。

3.SYN\_RCVD: 这个状态表示**接受到了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。

4.SYN\_SENT: 这个状态与SYN\_RCVD遥想呼应，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，并等待服务端的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。**

5.ESTABLISHED：这个容易理解了，表示**连接已经建立**了。

6.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。

7.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，稍后再关闭连接。

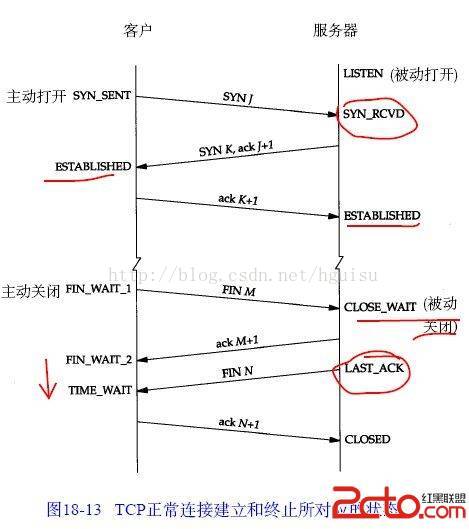
8.TIME\_WAIT: 表示收到了对方的FIN报文，并发送出了ACK报文，就等**2MSL后即可回到CLOSED可用状态**了。如果FIN\_WAIT\_1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。

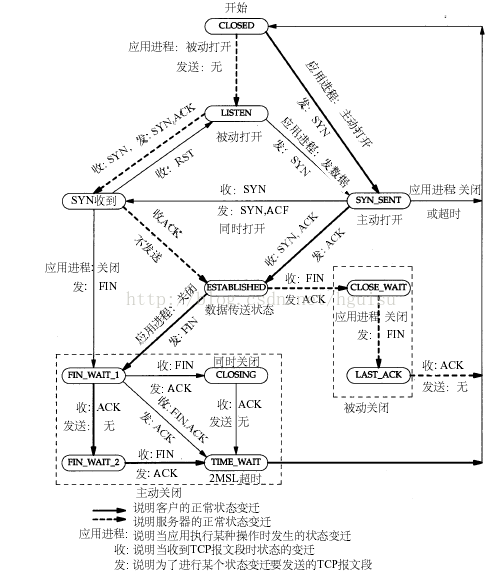
9.CLOSING: 这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

10.CLOSE\_WAIT: 这种状态的含义其实是表示**在等待关闭**。怎么理解呢？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。接下来呢，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。

11.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。

**3次握手4次挥手状态转移图：**





**客户端应用程序的状态迁移图**

客户端的状态可以用如下的流程来表示：

CLOSED->SYN\_SENT->ESTABLISHED->FIN\_WAIT\_1->FIN\_WAIT\_2->TIME\_WAIT->CLOSED

以上流程是在程序正常的情况下应该有的流程，从书中的图中可以看到，在建立连接时，当客户端收到SYN报文的ACK以后，客户端就打开了数据交互地连接。而结束连接则通常是客户端主动结束的，客户端结束应用程序以后，需要经历FIN\_WAIT\_1，FIN\_WAIT\_2等状态，这些状态的迁移就是前面提到的结束连接的四次握手。

**服务器的状态迁移图**

服务器的状态可以用如下的流程来表示：

CLOSED->LISTEN->SYN收到->ESTABLISHED->CLOSE\_WAIT->LAST\_ACK->CLOSED

    在建立连接的时候，服务器端是在第三次握手之后才进入数据交互状态，而关闭连接则是在关闭连接的第二次握手以后（注意不是第四次）。而关闭以后还要等待客户端给出最后的ACK包才能进入初始的状态。

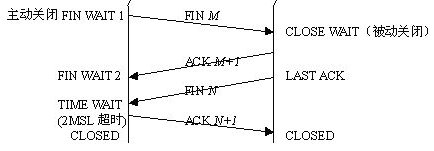
**13、****2MSL是什么状态？作用是什么（TIME\_WAIT状态是什么）**

**2MSL是什么状态？（TIME\_WAIT状态是什么）**

主动关闭的Socket端会进入**TIME\_WAIT状态**，并且持续2MSL时间长度，MSL就是maximum segment lifetime(最大分节生命期），这是一个IP数据包能在互联网上生存的最长时间，超过这个时间将在网络中消失。MSL在RFC 1122上建议是2分钟，而源自berkeley的TCP实现传统上使用30秒，因而，TIME\_WAIT状态一般维持在1-4分钟。

**作用是什么？**

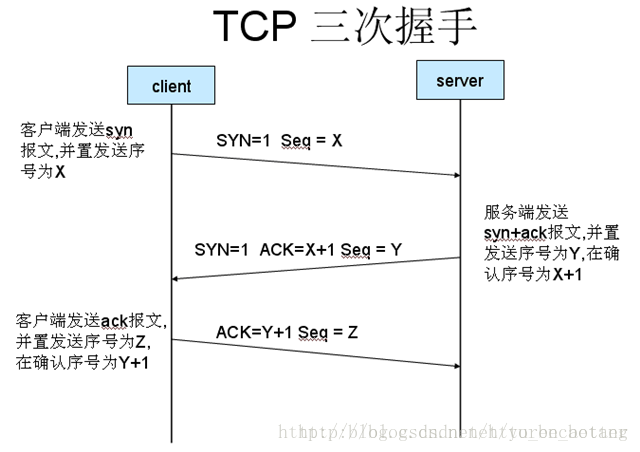
这是因为虽然双方都同意关闭连接了，而且握手的 4 个报文也都协调和发送完毕，按理可以直接回到 CLOSED 状态（就好比从 SYN\_SEND 状态到 ESTABLISH 状态那样）；但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的 ACK 报文会一定被对方收到，因此对方处于 LAST\_ACK 状态下的 SOCKET 可能会因为超时未收到 ACK 报文，而重发 FIN 报文，所以这个 TIME\_WAIT 状态的作用就是用来重发可能丢失的 ACK 报文。



附：

**TIME\_WAIT状态存在的理由：**  
**1）可靠地实现TCP全双工连接的终止**  
在进行关闭连接四路握手协议时，最后的ACK是由主动关闭端发出的，如果这个最终的ACK丢失，服务器将重发最终的FIN，因此客户端必须维护状态信息允许它重发最终的ACK。如果不维持这个状态信息，那么客户端将响应RST分节，服务器将此分节解释成一个错误（在java中会抛出connection reset的SocketException)。因而，要实现TCP全双工连接的正常终止，必须处理终止序列四个分节中任何一个分节的丢失情况，主动关闭 的客户端必须维持状态信息进入TIME\_WAIT状态。  
  
**2）允许老的重复分节在网络中消逝**  
TCP分节可能由于路由器异常而“迷途”，在迷途期间，TCP发送端可能因确认超时而重发这个分节，迷途的分节在路由器修复后也会被送到最终目的地，这个 原来的迷途分节就称为lost duplicate。在关闭一个TCP连接后，马上又重新建立起一个相同的IP地址和端口之间的TCP连接，后一个连接被称为前一个连接的化身（incarnation)，那么有可能出现这种情况，前一个连接的迷途重复分组在前一个连接终止后出现，从而被误解成从属于新的化身。为了避免这个情况，TCP不允许处于TIME\_WAIT状态的连接启动一个新的化身，因为TIME\_WAIT状态持续2MSL，就可以保证当成功建立一个TCP连接的时候，来自连接先前化身的重复分组已经在网络中消逝。

**14、三次握手为什么不是两次或者四次**



首先，我们要知道**TCP是全双工**的，即客户端在给服务器端发送信息的同时，服务器端也可以给客户端发送信息。

S:服务端 C:客户端

### 为什么不是两次

因为我们不进行第三次握手，所以在S对C的请求进行回应(第二次握手)后，就会理所当然的认为连接已建立，而如果C并没有收到S的回应呢第二次握手丢失)？此时，C仍认为连接未建立，S会对已建立的连接保存必要的资源，如果大量的这种情况，S会崩溃。

### 为什么不是四次

既然没法确认第二次的握手，C是否可以收到，

那么怎么确定第三次握手S就可以收到呢？

而三次握手后，C和S至少可以确认之前的通信情况，因为在三次结束之后，就已经可以保证C可以给S发信息，C可以收到S的信息； S可以给C发信息，S可以收到C的信息但无法确认之后的情况。

在这个道理上说，无论是四次还是五次或是更多次都是徒劳的。

**15、****TCP重发机制**

TCP的重传机制有两种：超时重传和快速重传。这两种重发策略一般是同时使用，它们是互补的。

**为什么需要重传?**

任何信息在介质中传输可能丢失，这是由于传输介质的物理特性决定的，所以网络不可能被设计为“可靠的”（不是由于考虑“性能”原因而是压根做不到）。既然物理层无法提供可靠数据传输那么只能由**协议**提供可靠传输了，其中最有名的协议就是TCP了。

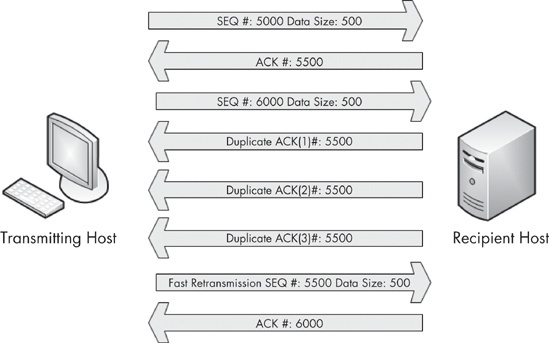
TCP是基于IP的网络协议，它提供可靠、有序的数据传输。在数据传输之前客户端和服务器端通过**三次握手**建立连接，建立连接的就是双方交换Seq（数据包序号）、MSS（每个TCP数据包大小） 、Win（滑动窗口，一次可以确认多少个TCP数据包），连接建立完成后每个TCP数据包都要被ACK（确认）。简单来说TCP通过确认/重传机制实现了“数据包可靠传输”

**超时重传**

说白了就是在请求包发出去的时候，开启一个计时器，当计时器达到时间之后，没有收到ACK，则就进行重发请求的操作，一直重发直到达到重发上限次数或者收到ACK。发送端的等待的时间叫RTO（Retransmission TimeOut）.

**快速重传**

还有一种机制就是快速重传，Fast Retransmit是由接收端**主动要求**重发的，当接收方收到的数据包是不正常的序列号，那么接收方会重复把应该收到的那一条ACK重复发送，这个时候，如果发送方收到连续3条的同一个序列号的ACK，那么就会启动快速重传机制，把这个ACK对应的发送包重新发送一次。具体可以参考：



**16、说说Nagle算法**

## Nagle算法

如果互联网上传递的都是小包，那绝对是个灾难，每个网络请求都耗费比较大的资源，如果一份数据分为零零散散很多份小包，每个网络传输都只传输一个小包，那么是典型的浪费资源，增加拥堵。糊涂窗口综合症就是发送方和接收方糊里糊涂达成的协商是传送小包。

Nagle算法主要是避免发送小的数据包，要求**TCP连接上最多只能有一个未被确认的小分组，在该分组的确认到达之前不能发送其他的小分组**。相反，TCP收集这些少量的小分组，并在确认到来时以一个分组的方式发出去。

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算法的基本定义是任意时刻，最多只能有一个未被确认的小段。 所谓“小段”，指的是小于MSS尺寸的数据块，所谓“未被确认”，是指一个数据块发送出去后，没有收到对方发送的ACK确认该数据已收到。  
　　**Nagle算法的规则**（可参考tcp\_output.c文件里tcp\_nagle\_check函数注释）：

（1）如果包长度达到MSS，则允许发送；

（2）如果该包含有FIN，则允许发送；

（3）设置了TCP\_NODELAY选项，则允许发送；

（4）未设置TCP\_CORK选项时，若所有发出去的小数据包（包长度小于MSS）均被确认，则允许发送；

（5）上述条件都未满足，但发生了超时（一般为200ms），则立即发送。

Nagle算法规定了，发送方网络链路上一个连接只能有一个未获得ACK的请求包。这个就意味着，发送方只有等待上一个请求的ACK回来之后才能发送下一个请求，这样两个请求过程中间，发送方的缓存区就存储了足够滑动窗口大小的包进行传递，这样就有效避免了大量的小包产生。

**17、TCP拥塞控制**

TCP 的拥塞控制方法主要有四种：**慢开始、拥塞避免、快重传和快恢复。**

为了简化问题，在下面的讨论中，我们作出如下两个假设：

**a 数据单向传输；**

**b 接收方的缓存是无限大的，窗口大小仅由网络的拥塞程度决定；**

**(1) 慢开始和拥塞避免**

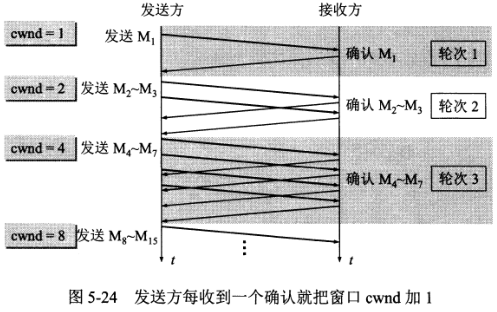
这里存在这样一个原则：**只要网络没有出现拥塞，拥塞窗口 cwnd（可以理解为发送窗口）就可以无限增大，以便于单次发送更多的数据，但是只要网络发生了拥塞，就必须把拥塞窗口减小，以减少涌入网络的数据量，从而减轻网络的拥塞程度。**

但是发送方如何知道网络发生了拥塞呢？当网络发生拥塞时，路由器的缓存便处于爆满状态，此时它不得不丢弃一部分分组。这样发送方就不可能收到确认报文，最终导致等待超时。现代网络线路的传输质量都很好，因为硬件层面的问题导致丢弃分组的情况是很少发生的。也就是说，**出现超时便代表着网络出现了拥塞。**

本来是通过控制拥塞窗口的大小来实现 TCP 的拥塞控制，但是**为了更好地解释慢开始的原理，我们将报文段的个数作为拥塞窗口的单位，即从原来的增加窗口大小变为了增加报文段的个数。**这样可以用较小的数字来说明拥塞控制的原理。

**慢开始算法**

一开始的时候设置 A 的窗口 cwnd 为 1，然后发送报文段 M1，B 收到报文段之后进行确认，接着 A 将窗口设置为 2（按照我们之间的假设，此时应该发送 2 个报文段），然后发送 2 个报文段 M2 和 M3，B 继续返回对 M2 和 M3 的确认，A 再将窗口增大为 4，发送 4 份报文段给 B……，流程图如下所示：

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E6%85%A2%E5%BC%80%E5%A7%8B.png)

可以看到，**每经过一个传输轮次（一个传输轮次是指拥塞窗口内的所有字节都发送出去，并且收到了最后一个发送的字节的确认），拥塞窗口的大小就翻倍，这样就会导致在慢开始阶段，窗口的大小是呈指数增长的，这是为了快速达到网络发生阻塞的临界点。显然，慢开始算法的“慢”不是指增长慢，而是指最开始的时候，从一个较低的起点开始增长。**

值得注意的是，上面的结论并不具有实际意义，因为拥塞窗口只要收到一个新的确认报文就会增大（假如这个确认报文段使得窗口增大），并且立即发送新窗口中的数据，而不必等到该轮次中所有的报文段都成功确认。

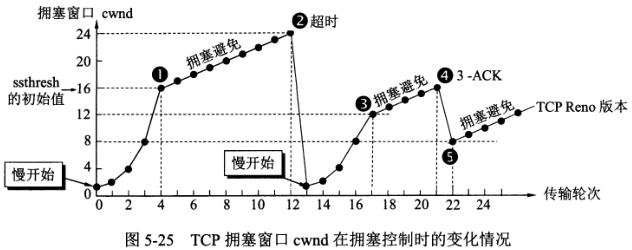
同时为了避免拥塞窗口无限增大造成网络拥塞，需要设置一个**慢开始门限 ssthresh**。拥塞窗口 cwnd 的大小和慢开始门限 ssthresh 之间存在如下的关系：

**a 当 cwnd < ssthresh 时，使用慢开始算法；**

**b 当 cwnd > ssthresh 时，停止使用慢开始算法而改用拥塞避免算法；**

**(2) 拥塞避免算法**

**不同于慢开始算法的指数级增长，拥塞避免算法是让窗口线性增大，即每经过一个往返时间 RTT 就让窗口增大 1。拥塞避免算法肯定不能“避免”网络拥塞的发生，只是使网络比较不容易发生拥塞。**下图很好的说明了该规律。

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E6%8B%A5%E5%A1%9E%E6%8E%A7%E5%88%B6.png)

**图中 X 轴为传播轮次而非时间，Y 轴为窗口的大小（单位为报文段的个数而非字节数），慢开始门限为 16。**下面对图中信息做如下解释：

**在慢开始阶段，拥塞窗口的大小呈指数增加；**

**到达慢开始门限后，采用拥塞避免算法，窗口大小依次加 1，呈线性增长趋势；**

**当窗口 cwnd = 24 时，出现了超时，此时发送方判断网络发生了阻塞，于是调整 ssthresh = cwnd/2 = 12，同时重新设置窗口 cwnd = 1；**

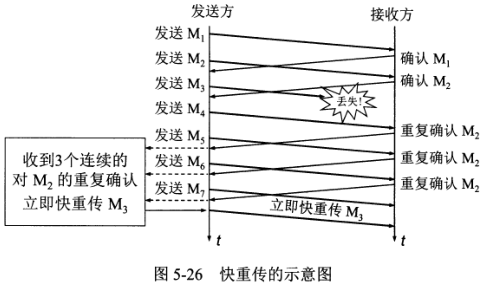
**接下来继续慢开始算法，达到新的门限 ssthresh = 12 时改用拥塞避免算法，窗口值呈线性增大；**

**当窗口值 cwnd = 16 时，出现了新的情况：发送方连续收到了 3 个对于同一报文段的重复确认（图中的 3-ACK），这个问题将在快重传和快恢复中继续讨论；**

**(3) 快重传和快恢复**

我们先考虑这样一种情况：**有时候，确认报文段只是在网络传输中丢失了，但是其实网络并没有发生拥塞。由于发送方接收不到确认报文段而导致超时的发生，此时便会开启慢开始算法，然后将窗口置为 1，导致传输效率降低。**

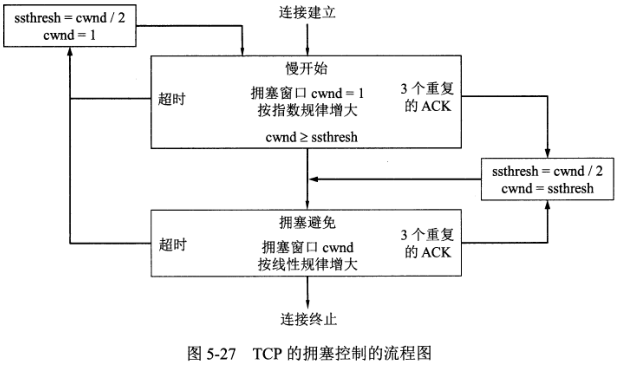
在下面的图中，发送方先发送了 M1 和 M2，并且成功收到了它们的确认报文段，但是接下来发送的 M3 却丢失了。由于接受方此时并不知道发送方向其发送了 M3，所以此时它做出任何响应。当接受方接收到发送方继续发送的 M4 的时候，经过的简单对比，它知道 M3 肯定是丢失了，于是**立刻向发送方发出对 M2 的重复确认**，对于后面的 M5、M6 也是如此。于是接收方一共收到了 4 个对 M2 的确认，除去第一个之外，**后面的三个都是重复确认**，也即图中的 **3-ACK**。此时，**快重传算法规定，当发送方连续收到 3 个重复确认的时候，就必须立即对丢失的报文段进行重传（即快重传）**，这样便不会导致超时的发生，从而避免了之前的情况。

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E5%BF%AB%E9%87%8D%E4%BC%A0.png)

接下来对于图中的 **3-ACK** 点来说，发送方知道只是丢失了部分报文段，于是执行**快恢复**算法，**调整 ssthresh = cwnd/2 = 8，同时设置 cwnd = ssthresh = 8，然后接着开始执行拥塞避免算法。**

**(4) 完整流程**

为了更好的分析整个流程，我们之前的讨论将实际过程可能遇到的问题进行了简化（比如没有考虑在慢开始阶段就遇到超时的情况），下面是 TCP 拥塞控制的完整流程图。

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/TCP%E6%8B%A5%E5%A1%9E%E6%8E%A7%E5%88%B6.png)

另外，我们在刚开始的时候，假定发送窗口的大小只会受到网络的影响。但是接收方的缓存肯定是有限的，也就是说：**接收窗口 rwnd 和拥塞窗口 cwnd 二者共同制约着发送窗口的大小。**显然，三者之间存在下图中的关系：

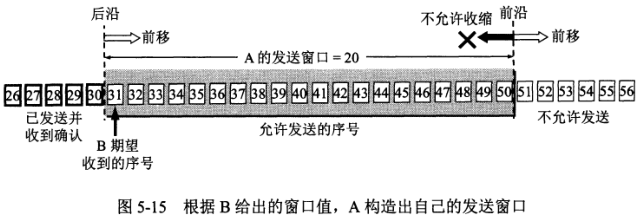
[发送窗口上限](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E5%8F%91%E9%80%81%E7%AA%97%E5%8F%A3%E4%B8%8A%E9%99%90.png)

上述公式说明，**每次对发送窗口进行调整时，需要根据当前的网络状况和接收窗口的大小而定。**

**18、TCP滑动窗口**

滑动窗口是发送方通过接受方的确认报文段中描述的缓冲区的大小来计算自己最多能发送多长的数据。如果发送方收到接受方的窗口大小为0的TCP数据报，那么发送方将停止发送数据，等到接受方发送窗口大小不为0的数据报的到来。

下面我们以一个完成的例子演示滑动窗口的工作原理。**假设发送方 A 再给 B 发送一个报文段之后，收到了 B 发来的确认报文段，其中窗口为 20，确认号为 31。**这表示 B 希望下一次收到的字节流的序号应该从 31 开始，并且最大只能接收 20 个字节。于是 A 根据确认报文段构建出如下的发送窗口：

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E7%AA%97%E5%8F%A3%E7%A4%BA%E4%BE%8B1.png)

值得注意的是，实际中发送窗口的大小需要根据确认报文段中的窗口值以及当前网络的拥塞情况来确定，在这里我们假设网络是畅通的，只考虑窗口值的影响。

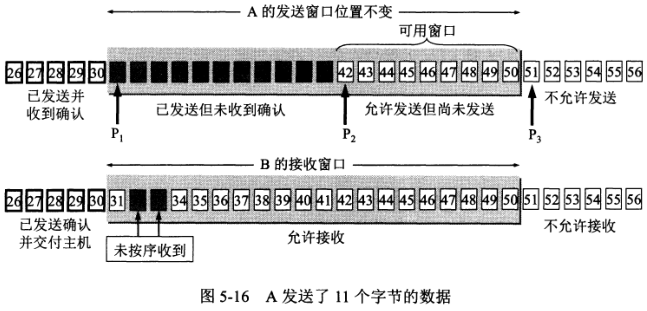
可以看到，在发送窗口中存在一个前沿和后沿，后沿之后是已经发送且得到确认的数据，无需保留；前沿的数据禁止在本次发送中发送。此时，A 可以将发送窗口中的数据全部发送出去，并且在没有收到 B 的确认报文段的情况下，不能将已发送的数据从缓存中清除，这是为了超时重传的需要。

发送窗口的位置由前沿和后沿共同确定，后沿可以前移（收到确认）或者不动（没有收到确认），同时前沿也只能前移或者不动。前沿不动对应着两种情况：

(1) 没有收到确认；

(2) 收到了确认，但是窗口变小了（比如 A 下一次收到的确认号为 41，但是窗口为 10）；

**现在假设 A 将序号为 31~41 的数据发送过去了，但是还没有收到 B 的确认报文段，**这个时候发送窗口如下所示：

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E7%AA%97%E5%8F%A3%E7%A4%BA%E4%BE%8B2.png)

图中的黑色字节分别表示已发送核已接受的字节，这个时候我们可以用三个指针 P1、P2 和 P3 对发送窗口的状态进行描述：

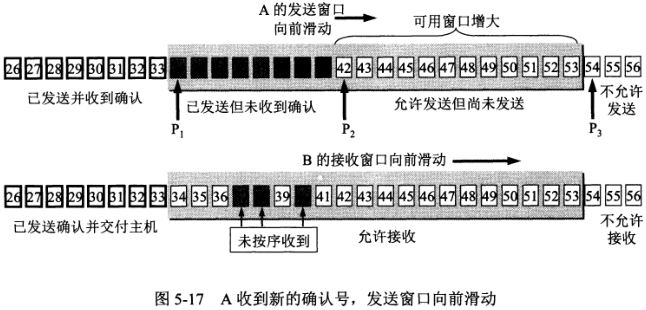
(1) P1 和 P2 之间为已发送但未确认的数据；

(2) P2 和 P3 之间为待发送数据（**可用窗口**）；

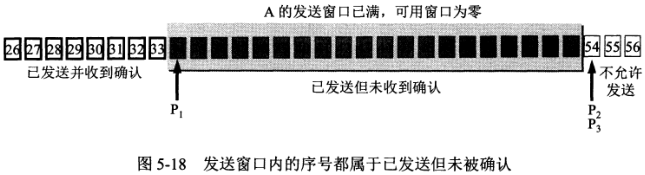
(3) P1 和 P3 之间为整个发送窗口；

**假设由于某种原因，B 未能收到序号为 31 的字节，但是却收到了 32、33 号字节。这个时候显然数据传输出现了问题，B 会重新向 A 发送窗口为 20，确认号为 31 的确认报文，示意 A 将序号为 31~50 内的字节重新发送。**

接着 A 又重新发送了一段报文段，**先假设 B 顺利收到了序号为 31~33 的字节，并把它们交付给了应用进程。此时，B 将接收窗口向前移动 3 个位置，然后给 A 发送一段窗口为 20，确认号为 34 的报文段，A 收到报文段后，将发送创口向前移动 3 个位置，**如下图所示：

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E7%AA%97%E5%8F%A3%E7%A4%BA%E4%BE%8B3.png)

从图中可以看到，指针 P1 和 P3 都向前移动了 3 个位置，但是 P2 保持不动，这是因为此时还没有发送新的字节，但是可用窗口却变大了。**接着 A 将发送窗口内所有未发送的数据发送出去，此时指针 P2 和 P3 重合，可用窗口为 0，发送停止。**示意图如下：

[](https://blog.huster.site/2018/07/15/TCP-IP-Transport/%E7%AA%97%E5%8F%A3%E7%A4%BA%E4%BE%8B4.png)

此时 A 便进入等待状态，假如迟迟收不到确认报文段，A 会在一段时间之后进行**超时重传**（A 认为数据已经丢失了），但是重传时发送窗口还是不会移动。也就是说**只有当 A 等到 B 的确认报文之后，发送窗口才能向前移动。**

**缓存与滑动窗口的关系**

我们在上面的例子中所提到的字节流实际上是位于发、接双方的缓存中的，窗口在字节流上滑动，对发送和接收的字节流进行控制。

在发送方这边，发送窗口后沿的字节会被删去，应用进程写入的新的字节又会添加到字节流的最前头。因此必须对应用程序写入字节的速度进行控制，否则会爆缓存。

在接收方这边，**接收窗口后沿的数据只是在确认报文段中已经确认的数据，等待被应用进程读取，成功读取后才能删除。**如果应用进程来不及读取，接收缓存就会被填满，最终导致接收窗口为 0，然后又会反馈到发送窗口那边，使发送停止。

值得说明的是，在上面的例子中，我们为了解释的方便，只考虑了一方发送、一方接收的情况。但是由于 TCP 是全双工通信，**实际中任意一方都是发送窗口与接收窗口并存的。**

**详见：**[TCP/IP详解：滑动窗口](https://blog.csdn.net/wdscq1234/article/details/52444277)

**19、域名解析的过程**

DNS域名解析系统是基于UDP协议，将域名解析为IP地址。域名解析全过程如下：

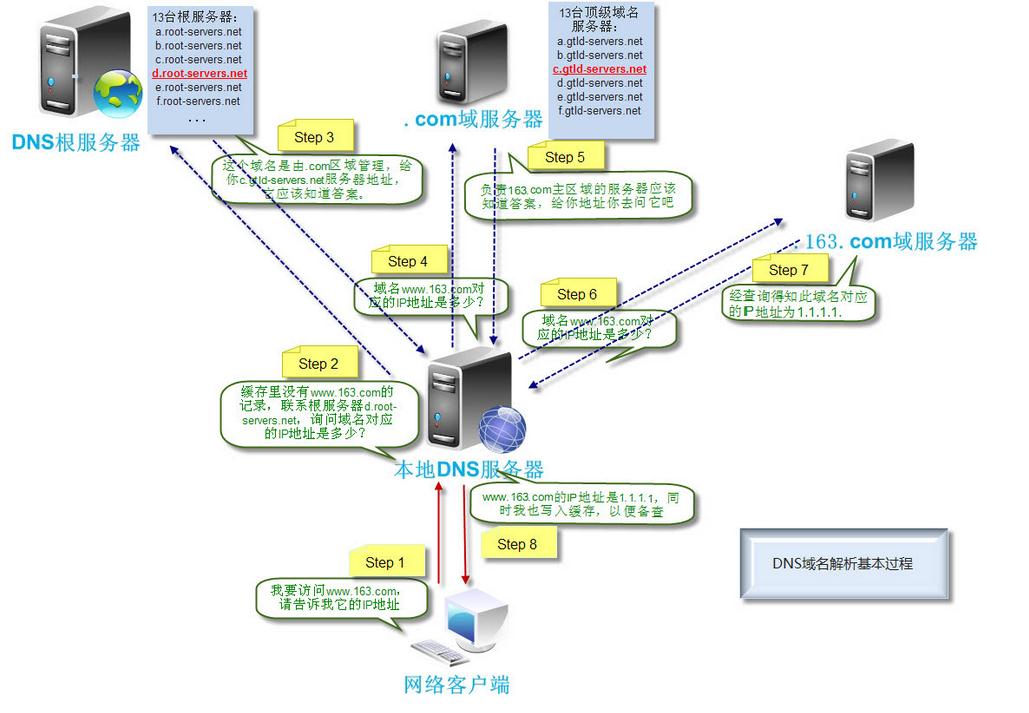
(1) 打开浏览器，输入一个域名如[www.163.com](http://www.163.com)，此时主机会发送一个DNS请求到本地浏览器（本地DNS服务器一般由ISP提供，如中国移动）。

(2) 本地DNS服务器查询缓存记录，有则直接返回结果。没有则向DNS根服务器进行查询。

(3) 如果根服务器没有记录具体的域名和IP地址的对应关系，就告知本地DNS服务器向域服务器进行查询。

(4) 域服务器dns.com告知本地DNS服务器所查域名对相应的IP地址。

(5) 本地DNS服务器最后将查询结果告知浏览器。



**20、ARP的实现机制**

ARP：地址解析协议，是将IP地址解析为物理地址(MAC地址)的协议，用来解决**同一个局域网**上的主机或路由器的IP地址和硬件地址的映射问题。**ARP高速缓存**用来存放本局域网中的各主机和路由器中的IP地址到MAC地址的映射表。

ARP的工作原理：

(1) 当源主机要发送数据时，首先检查ARP高速缓存列表中是否有对应IP地址的目的主机的MAC地址，若有，则直接发送数据；若没有，就在本局域网上广播发送ARP请求分组，包括源主机的IP地址、源主机的MAC地址和目的主机的IP地址。

(2) 当本局域网中的所有主机收到该ARP请求分组时，首先检查数据包中的目的主机IP地址和自己的IP地址是否一致，若不同，则忽略该数据包；若相同，则将数据包中的源主机的IP地址和MAC地址写入到ARP高速缓存列表中，然后向源主机发送ARP响应分组，同时在此ARP响应包中写入自己的MAC地址。

(3) 源主机收到ARP响应分组后，将目的主机的IP地址和MAC地址写入ARP高速缓存列表中，并利用此映射发送数据。

**广播发送ARP请求分组，单播发送ARP响应分组。**

**21、RARP的实现机制**

RARP：逆地址解析协议，作用与ARP相反，是将MAC地址解析为IP地址的协议。主要用于无盘工作站引导时获取IP地址。

(1) 源端发送一个本地的RARP广播包，在此广播包中声明自己的MAC地址，并且请求任何收到此请求的RARP服务器分配一个IP地址。

（2）本地网段上的RARP服务器收到此请求后，检查其RARP列表，查找该MAC地址对应的IP地址。如果存在，RARP服务器就给源端发送一个响应数据包，并将此IP地址提供给对方主机使用；如果不存在，RARP服务器对此不做任何响应。

（3）源端在收到从RARP服务器来的响应信息后，利用得到的IP地址进行通信；如果一直没有收到RARP服务器的响应信息，则表示初始化失败。

**广播发送RARP请求分组，单播发送RARP响应分组。**

**22、HTTP/HTTPS 1.0 /1.1 /2.0的特点和区别**

HTTPS是基于安全套接字层的超文本传输协议，HTTPS在HTTP应用层的基础上使用安全套接字层作为子层。HTTPS = HTTP + SSL。

**HTTP和HTTPS的区别如下：**

(1) 安全性：HTTP是超文本传输协议，信息是明文传输，容易发生流量劫持和HTTP攻击，HTTPS进行SSL加密传输、身份认证，比HTTP安全；

(2) 费用：HTTP免费，HTTPS的CA证书需要一定费用；

(3) 端口：HTTP标准端口为80，HTTPS标准端口号为443。

**HTTP1.0/1.1/2.0的特点和区别如下：**

(1) HTTP1.0规定浏览器与服务器保持较短时间的链接，链接无法复用，易产生线头阻塞；

(2) HTTP1.1支持持久链接，增加请求头和响应头来扩充功能，包括断点续传、缓存处理和host头处理；

(3) HTTP2.0支持多路复用、首部压缩、流量控制和服务端推送。

**详见链接**：[HTTP1.0、HTTP1.1和HTTP2.0的区别](https://www.cnblogs.com/heluan/p/8620312.html)

**23、get/post的区别**

get和post是客户端和服务器之间进行HTTP请求-响应时的两种基本的请求方法。主要区别如下：

(1) get一般用于查询获取信息，效率较高；post一般用于更新修改信息。

(2) get参数放在URL之后，明文显示；post参数放在HTTP请求报文的报主体中，安全性相对较高；

(3) get请求会被浏览器主动缓存，而post不会，除非手动设置。

(4) 请求传输数据的大小不同，对于get，由于浏览器对URL的长度有限制，请求的数据较小，常见在1K以内；post请求的数据较大，大小由服务器的设定值限制。

(4) get产生一个TCP数据包，浏览器会把HTTP header和data一并发送出去，服务器响应200(返回数据)； post产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。

**应用场景**：get适合于查询区分某些特定数据的页面，如商品展示(对同一 URL 的多个请求应该返回同样的结果)；post适合用于登录注册功能页。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **请求方式** | **数据位置** | **明文密文** | **数据安全** | **长度限制** | **应用场景** |
| get | HTTP报头 | 明文 | 不安全 | 长度较小 | 查询数据 |
| post | HTTP正文 | 可明可密 | 相对安全 | 支持较大数据传输 | 修改数据 |

**24、HTTP返回状态码**

状态码由三位数字组成，其中第一个数字定义了响应的类别，共分为五种：

1xx：通知信息，表示请求已接收，继续处理；

2xx：成功，表示请求已被服务器端成功接收；

3xx：重定向，要完成请求必须进行更进一步的操作；

4xx：客户端错误，请求有语法错误或请求无法实现；

5xx：服务器端错误，服务器失效未能实现合法的请求。

常见的HTTP状态码如下：

|  |  |
| --- | --- |
| **HTTP状态码** | **描述** |
| 100 Continue | 服务器接受到部分请求，等待客户端继续发送 |
| 200 OK | 服务器接收到全部请求，请求成功 |
| 301 Moved Permanently | 永久重定向 |
| 302 Temporarily Moved | 临时重定向 |
| 400 Bad Request | 请求存在语法错误，服务器无法识别 |
| 403 Forbidden | 请求的对应资源禁止被访问 |
| 404 Not Found | 找不到指定的资源文件 |
| 500 Internet Server Error | 服务器内部发生错误 |

**25、Http协议相关头**

有 4 种类型的首部字段：通用首部字段、请求首部字段、响应首部字段和实体首部字段

各种首部字段及其含义如下（不需要全记，仅供查阅,[MarkDowm链接](https://github.com/CyC2018/Interview-Notebook/blob/master/notes/HTTP.md#%E5%9B%9Bhttp-%E9%A6%96%E9%83%A8)）：

1. **通用首部字段**

|  |  |
| --- | --- |
| **首部字段名** | **说明** |
| Cache-Control | 控制缓存的行为 |
| Connection | 控制不再转发给代理的首部字段、管理持久连接 |
| Date | 创建报文的日期时间 |
| Pragma | 报文指令 |
| Trailer | 报文末端的首部一览 |
| Transfer-Encoding | 指定报文主体的传输编码方式 |
| Upgrade | 升级为其他协议 |
| Via | 代理服务器的相关信息 |
| Warning | 错误通知 |

1. **请求首部字段**

|  |  |
| --- | --- |
| **首部字段名** | **说明** |
| Accept | 用户代理可处理的媒体类型 |
| Accept-Charset | 优先的字符集 |
| Accept-Encoding | 优先的内容编码 |
| Accept-Language | 优先的语言（自然语言） |
| Authorization | Web 认证信息 |
| Expect | 期待服务器的特定行为 |
| From | 用户的电子邮箱地址 |
| Host | 请求资源所在服务器 |
| If-Match | 比较实体标记（ETag） |
| If-Modified-Since | 比较资源的更新时间 |
| If-None-Match | 比较实体标记（与 If-Match 相反） |
| If-Range | 资源未更新时发送实体 Byte 的范围请求 |
| If-Unmodified-Since | 比较资源的更新时间（与 If-Modified-Since 相反） |
| Max-Forwards | 最大传输逐跳数 |
| Proxy-Authorization | 代理服务器要求客户端的认证信息 |
| Range | 实体的字节范围请求 |
| Referer | 对请求中 URI 的原始获取方 |
| TE | 传输编码的优先级 |
| User-Agent | HTTP 客户端程序的信息 |

1. **响应首部字段**

|  |  |
| --- | --- |
| **首部字段名** | **说明** |
| Accept-Ranges | 是否接受字节范围请求 |
| Age | 推算资源创建经过时间 |
| ETag | 资源的匹配信息 |
| Location | 令客户端重定向至指定 URI |
| Proxy-Authenticate | 代理服务器对客户端的认证信息 |
| Retry-After | 对再次发起请求的时机要求 |
| Server | HTTP 服务器的安装信息 |
| Vary | 代理服务器缓存的管理信息 |
| WWW-Authenticate | 服务器对客户端的认证信息 |

1. **实体首部字段**

|  |  |
| --- | --- |
| **首部字段名** | **说明** |
| Allow | 资源可支持的 HTTP 方法 |
| 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 | 实体主体的大小 |
| Content-Location | 替代对应资源的 URI |
| Content-MD5 | 实体主体的报文摘要 |
| Content-Range | 实体主体的位置范围 |
| Content-Type | 实体主体的媒体类型 |
| Expires | 实体主体过期的日期时间 |
| Last-Modified | 资源的最后修改日期时间 |

**26、浏览器中输入一个URL发生什么，用到那些协议**

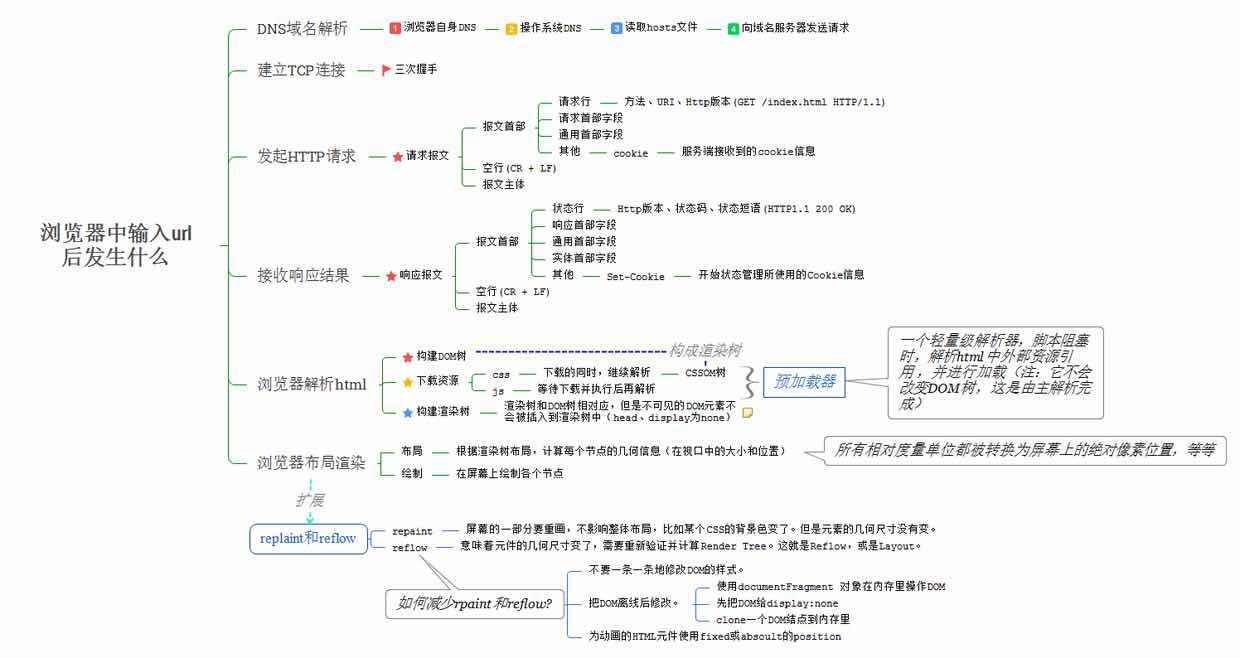
1） DNS域名解析: 浏览器缓存->系统缓存->路由器-> ISP的DNS服务器->根域名服务器。把域名转化成IP地址。

2）与IP地址对应的服务器建立TCP连接，经历三次握手：SYN，ACK、SYN，ACK

3）以get，post方式发送HTTP请求，get方式发送主机，用户代理，connection属性，cookie等

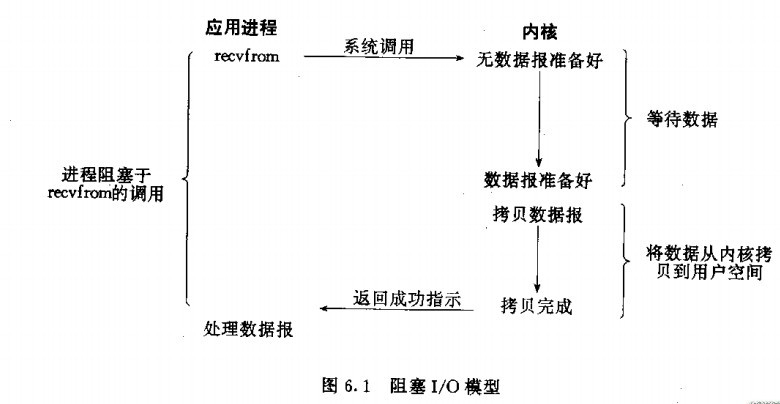
4）获得服务器的响应，显示页面

**用到的协议：**DNS协议，TCP/IP协议，http协议



**27、五种IO模型**

**1）阻塞IO（bloking IO）**



说明：用户进程process在blocking IO读recvfrom操作的两个阶段都是等待的。在数据没准备好的时候，process原地等待kernel准备数据。kernel准备好数据后，process继续等待kernel将数据copy到自己的buffer。在kernel完成数据的copy后process才会从recvfrom系统调用中返回

**2）非阻塞IO（non-blocking IO）**



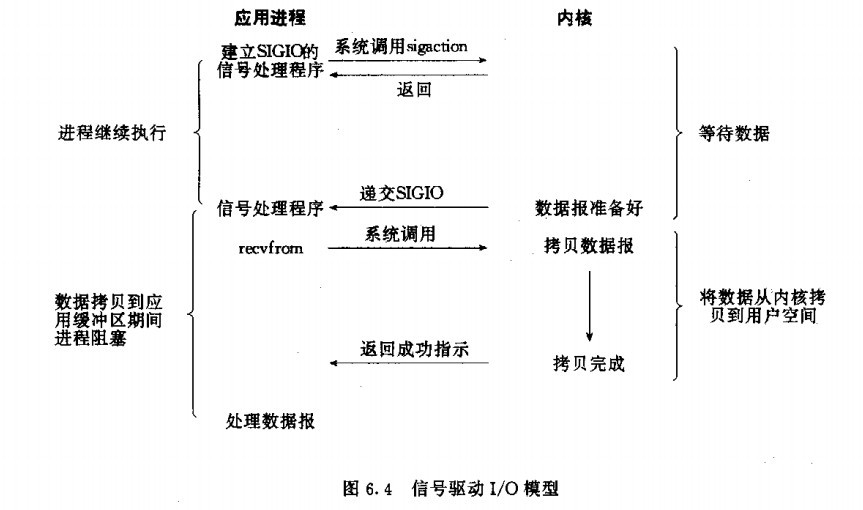
说明：用户进程process在NonBlocking IO读recvfrom操作的第一个阶段是不会block等待的，如果kernel数据还没准备好，那么recvfrom会立刻返回一个EWOULDBLOCK错误。当kernel准备好数据后，进入处理的第二阶段的时候，process会等待kernel将数据copy到自己的buffer，在kernel完成数据的copy后process才会从recvfrom系统调用中返回

**3）多路复用IO（multiplexing IO）**



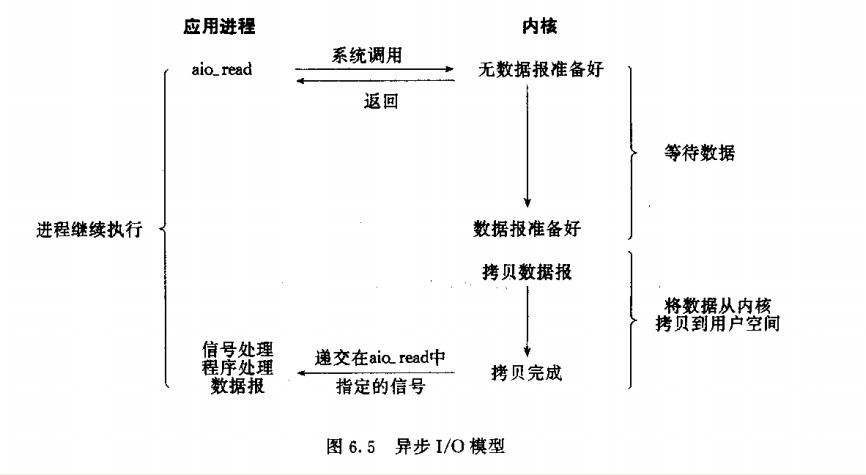
说明：IO多路复用，就是我们熟知的select、poll、epoll模型。从图上可见，在IO多路复用的时候，process在两个处理阶段都是block住等待的。初看好像IO多路复用没什么用，其实select、poll、epoll的优势在于可以以较少的代价来同时监听处理多个IO

**4）信号驱动式IO（signal-driven IO）**



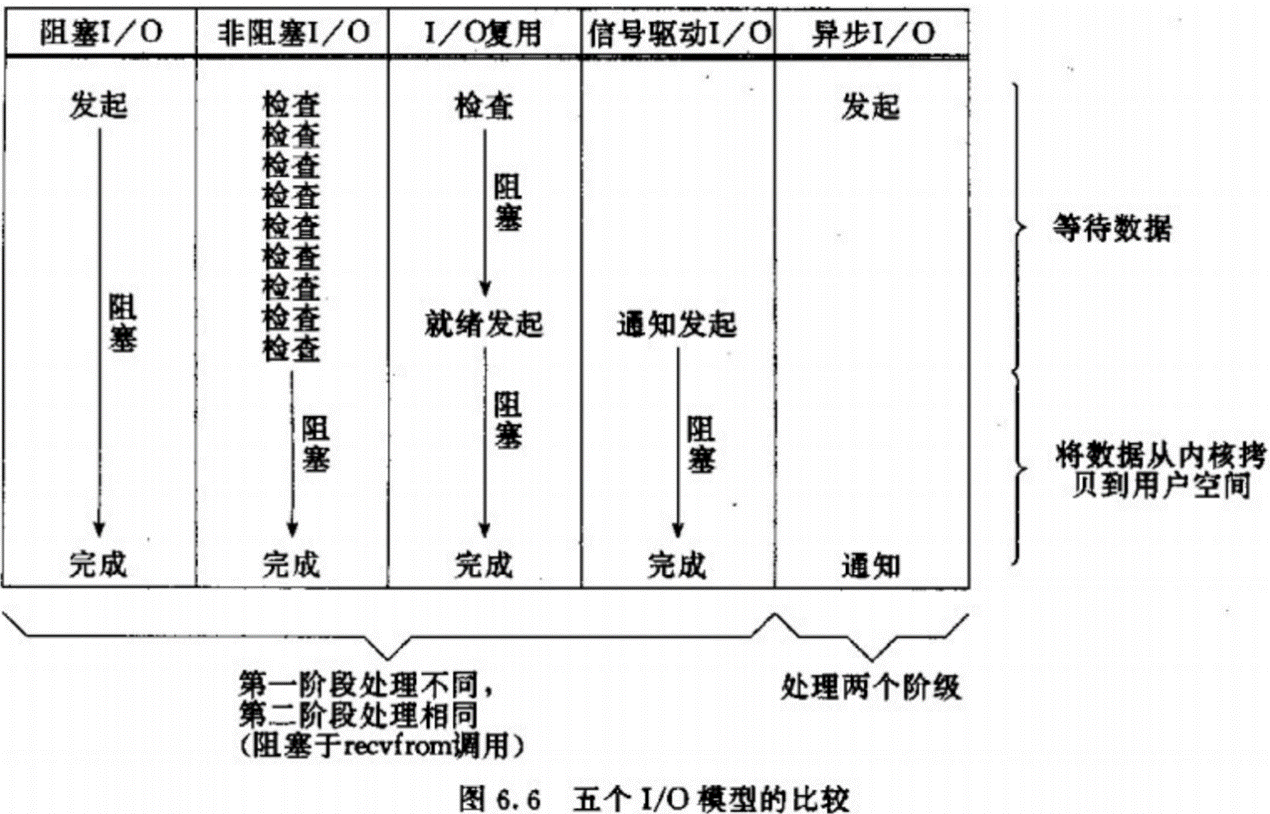
说明：**两次调用，两次返回**； 首先我们允许套接口进行信号驱动I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。

**5）异步IO（asynchronous IO）**



说明：从上图看，异步IO要求进程在recvfrom操作的两个处理阶段上都不能等待，也就是进程调用recvfrom后立刻返回，kernel自行去准备好数据并将数据从kernel的buffer中copy到process的buffer在通知process读操作完成了，然后process在去处理。遗憾的是，linux的网络IO中是不存在异步IO的，linux的网络IO处理的第二阶段总是阻塞等待数据copy完成的。真正意义上的网络异步IO是Windows下的IOCP（IO完成端口）模型

**五种IO模型的比较如下：**

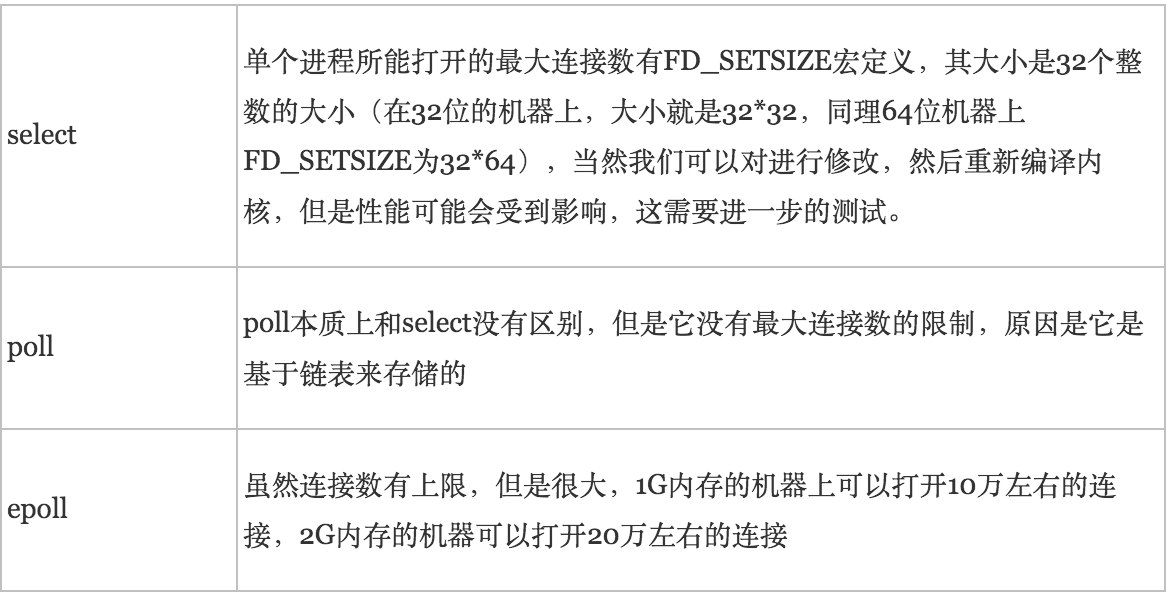


**28、select,poll,epoll的区别**

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的。**对他们的实现和对比如下：

select,poll,epoll主要**区别**如下

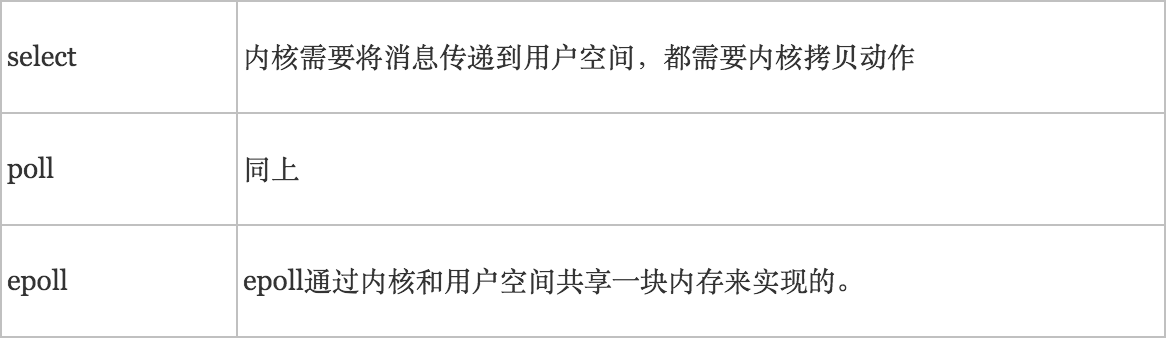
**1.支持一个进程所能打开的最大连接数**



**2.FD剧增后带来的IO效率问题**



**3.消息传递方式**



综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点：

（1）表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

（2）select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善。

**29、UDP中一个包的大小最大能多大？**

相对于不同的系统,不同的要求,其得到的答案是不一样的。

**局域网环境下，建议将UDP数据控制在1472字节以下。**

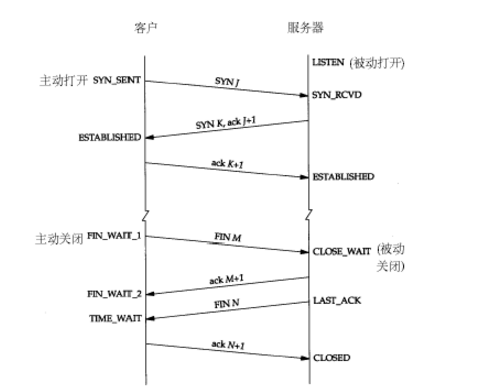
以太网(Ethernet)数据帧的长度必须在46-1500字节之间,这是由以太网的物理特性决定的，这个1500字节被称为链路层的MTU(最大传输单元)。但这并不是指链路层的长度被限制在1500字节，其实这个MTU指的是链路层的数据区，并不包括链路层的首部和尾部的18个字节。所以，事实上这个1500字节就是网络层IP数据报的长度限制。因为IP数据报的首部为20字节，所以IP数据报的数据区长度最大为1480字节。而这个1480字节就是用来放TCP传来的TCP报文段或UDP传来的UDP数据报的。又因为UDP数据报的首部8字节，所以UDP数据报的数据区最大长度为1472字节。这个1472字节就是我们可以使用的字节数。

当我们发送的UDP数据大于1472的时候会怎样呢？ 这也就是说IP数据报大于1500字节，大于MTU，这个时候发送方IP层就需要分片(fragmentation)。把数据报分成若干片，使每一片都小于MTU，而接收方IP层则需要进行数据报的重组。这样就会多做许多事情，而更严重的是，由于UDP的特性，当某一片数据传送中丢失时，接收方无法重组数据报，将导致丢弃整个UDP数据报。因此，在普通的局域网环境下，建议将UDP的数据控制在1472字节以下为好。

**Internet编程时，建议将UDP数据控制在548字节以下**

进行Internet编程时则不同，因为Internet上的路由器可能会将MTU设为不同的值。如果我们假定MTU为1500来发送数据，而途经的某个网络的MTU值小于1500字节，那么系统将会使用一系列的机制来调整MTU值，使数据报能够顺利到达目的地，这样就会做许多不必要的操作。鉴于Internet上的标准MTU值为576字节，所以我建议在进行Internet的UDP编程时，最好将UDP的数据长度控件在548字节以内。

**30、time\_wait close\_wait产生的原因（keepalive机制）**



TCP协议规定，对于已经建立的连接，网络双方要进行四次握手才能成功断开连接，如果缺少了其中某个步骤，将会使连接处于假死状态，连接本身占用的资源不会被释放。网络服务器程序要同时管理大量连接，所以很有必要保证无用连接完全断开，否则大量僵死的连接会浪费许多服务器资源。在众多TCP状态中，最值得注意的状态有两个：CLOSE\_WAIT和TIME\_WAIT。

**TIME\_WAIT 是主动关闭链接时形成的，等待2MSL(Max Segment Lifetime)时间，约4分钟。**主要是防止最后一个ACK丢失。由于TIME\_WAIT 的时间会非常长，因此server端应尽量减少主动关闭连接。为什么TIME\_WAIT状态停留2MSL（max segment lifetime）时间也就是TCP/IP设计者本来是这么设计的主要有两个原因：1. 防止上一次连接中的包，迷路后重新出现，影响新连接（经过2MSL，上一次连接中所有的重复包都会消失）；2.可靠的关闭TCP连接,在主动关闭方发送的最后一个ack(fin)，有可能丢失，这时被动方会重新发fin, 如果这时主动方处于 CLOSED 状态 ，就会响应 rst 而不是 ack。所以主动方要处于 TIME\_WAIT 状态，而不能是 CLOSED 。

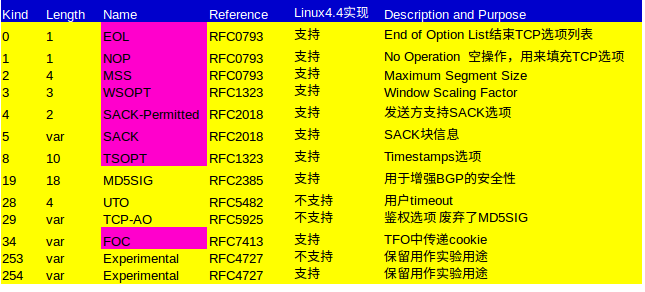
**CLOSE\_WAIT是被动关闭连接是形成的。**根据TCP状态机，服务器端收到客户端发送的FIN，则按照TCP实现发送ACK，因此进入CLOSE\_WAIT状态。但如果服务器端不执行close()，就不能由CLOSE\_WAIT迁移到LAST\_ACK，则系统中会存在很多CLOSE\_WAIT状态的连接。此时，可能是系统忙于处理读、写操作，而未将已收到FIN的连接，进行close。此时，recv/read已收到FIN的连接socket，会返回0。

解决CLOSE\_WAIT的方法:1 一般原因都是TCP连接没有调用关闭方法。需要应用来处理网络链接关闭。2 对于Web请求出现这个原因，经常是因为Response的BodyStream没有调用Close.比如Widnows下:使用HttpWebRequest 一定要保证GetRequestStream和GetResponse对象关闭，否则容易造成连接处于CLOSE\_WAIT状态。3. TCP的KeepLive功能，可以让操作系统替我们自动清理掉CLOSE\_WAIT的连接。但是KeepLive在Windows操作系统下默认是7200秒，也就是2个小时才清理一次。往往满足不了要求。可以调小该数值。

Close\_Wait引发的问题:Close\_Wait会占用一个连接，网络可用连接小。数量过多，可能会引起网络性能下降，并占用系统非换页内存。尤其是在有连接池的情况下(比如HttpRequest)会耗尽连接池的网络连接数，导致无法建立网络连接。

**31、列举你所知道的TCP选项**

tcp基本头下面可以带有tcp选项，其中有些选项只能在连接过程中随着SYN包发送，有些可以延后。下表汇总了一些tcp选项



**32****、connect会阻塞检测以及防止，socket什么情况下可读**

建立socket后默认connect()函数为阻塞连接状态，在大多数实现中，connect的超时时间在75s至几分钟之间，想要缩短超时时间，可解决问题的两种方法：方法一、将socket句柄设置为非阻塞状态，方法二、采用信号处理函数设置阻塞超时控制。

下列四个条件中的任何一个满足时,socket准备好读:

1. socket的接收缓冲区中的数据字节大于等于该socket的接收缓冲区低水位标记的当前大小。对这样的socket的读操作将不阻塞并返回一个大于0的值(也就是返回准备好读入的数据)。我们可以用SO\_RCVLOWATsocket选项来设置该socket的低水位标记。对于TCP和UDP .socket而言，其缺省值为1.

2. 该连接的读这一半关闭(也就是接收了FIN的TCP连接)。对这样的socket的读操作将不阻塞并返回0

3.socket是一个用于监听的socket,并且已经完成的连接数为非0.这样的soocket处于可读状态,是因为socket收到了对方的connect请求,执行了三次握手的第一步:对方发送SYN请求过来,使监听socket处于可读状态;正常情况下,这样的socket上的accept操作不会阻塞;

4.有一个socket有异常错误条件待处理.对于这样的socket的读操作将不会阻塞,并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可通过指定socket选项SO\_ERROR调用getsockopt来取得并清除;

下列三个条件中的任何一个满足时,socket准备好写:

1. socket的发送缓冲区中的数据字节大于等于该socket的发送缓冲区低水位标记的当前大小。对这样的socket的写操作将不阻塞并返回一个大于0的值(也就是返回准备好写入的数据)。我们可以用SO\_SNDLOWAT socket选项来设置该socket的低水位标记。对于TCP和UDPsocket而言，其缺省值为2048

2. 该连接的写这一半关闭。对这样的socket的写操作将产生SIGPIPE信号，该信号的缺省行为是终止进程。

3.有一个socket异常错误条件待处理.对于这样的socket的写操作将不会阻塞并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可以通过指定socket选项SO\_ERROR调用getsockopt函数来取得并清除;

**33、如果select返回可读，结果只读到0字节，什么情况**

通常情况下，select返回可读事件，但是读取到0，针对于文件描述符，是说明文件已经读取到结尾；针对于套接字描述符，通常是因为对方已经至少关闭了写端，不能再从套接字读取数据。

补充知识点：

假设server和client 已经建立了连接，server调用了close, 发送FIN 段给client（其实不一定会发送FIN段，后面再说），此时server不能再通过socket发送和接收数据，此时client调用read，如果接收到FIN 段会返回0，但client此时还是可以write 给server的，write调用只负责把数据交给TCP发送缓冲区就可以成功返回了，所以不会出错，而server收到数据后应答一个RST段，表示服务器已经不能接收数据，连接重置，client收到RST段后无法立刻通知应用层，只把这个状态保存在TCP协议层。如果client再次调用write发数据给server，由于TCP协议层已经处于RST状态了，因此不会将数据发出，而是发一个SIGPIPE信号给应用层，SIGPIPE信号的缺省处理动作是终止程序。

shutdown 可以选择关闭某个方向或者同时关闭两个方向，shutdown how = 0 or how = 1 or how = 2 (SHUT\_RD or SHUT\_WR or SHUT\_RDWR)，后两者可以保证对等方接收到一个EOF字符（即发送了一个FIN段），而不管其他进程是否已经打开了这个套接字。而close不能保证，只有当某个sockfd的引用计数为0，close 才会发送FIN段，否则只是将引用计数减1而已。也就是说只有当所有进程（可能fork多个子进程都打开了这个套接字）都关闭了这个套接字，close 才会发送FIN 段。

如果是调用shutdown how = 1 ，则意味着往一个已经发送出FIN的套接字中写是允许的，接收到FIN段仅代表对方不再发送数据，但对方还是可以读取数据的，可以让对方可以继续读取缓冲区剩余的数据。

**34、socket什么情况下可读**

一、 下列四个条件中的任何一个满足时,socket准备好读:   
 1.socket接收缓冲区中已经接收的数据的字节数大于等于socket接收缓冲区低潮限度的当前值;对这样的socket的读操作不会阻塞,并返回一个大于0的值(即:准备好读入的数据的字节数).我们可以用socket选项SO\_RCVLOWAT来设置此低潮限度,对于TCP和UDPsocket,其缺省值为1;   
 2.连接的读这一半关闭(即:接收到对方发过来的FIN的TCP连接).对于这样的socket的读操作将不阻塞,并且返回0(即:文件结束符,FIN包体长度为0字节);   
 3.socket是一个用于监听的socket,并且已经完成的连接数为非0.这样的soocket处于可读状态,是因为socket收到了对方的确认,执行了三次握手的第三步:对方发送ACK过来,使监听socket处于可读状态;正常情况下,这样的socket上的accept操作不会阻塞;   
 4.有一个socket有异常错误条件待处理.对于这样的socket的读操作将不会阻塞,并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可通过指定socket选项SO\_ERROR调用getsockopt来取得并清除;   
二、 下列三个条件中的任何一个满足时,socket准备好写 :   
 1.socket发送缓冲区中的可用空间字节数大于等于socket发送缓冲区低潮限度的当前值,且

(i):socket已连接(TCP socket)

(ii):socket不要求连接(如:UDP socket).这意味着,如果我们将这样的socket设置为非阻塞模式,写操作将不会阻塞,并且返回一个正值(如:由传输层接收的字节数).我们可以用socket选项SO\_SNDLOWAT来设置此低潮限度,对于TCP和UDP socket,其缺省值一般是2048Bytes;   
 2.连接的写这一半关闭.对于这样的socket的的写操作将产生信号SIGPIPE;   
 3.有一个socket异常错误条件待处理.对于这样的socket的写操作将不会阻塞并且返回一个错误(-1),errno则设置成明确的错误条件.这些待处理的错误也可以通过指定socket选项SO\_ERROR调用getsockopt函数来取得并清除;

**35、Keepalive是什么东西，如何使用**

keepalive，是在TCP中一个可以检测死连接的机制。

keepalive原理很简单，TCP会在空闲了一定时间后发送数据给对方:

1.如果主机可达，对方就会响应ACK应答，就认为是存活的。

2.如果可达，但应用程序退出，对方就发FIN应答，发送TCP撤消连接。

3.如果可达，但应用程序崩溃，对方就发RST消息。

4.如果对方主机不响应ack, rst，继续发送直到超时，就撤消连接。这个时间就是默认的二个小时。

**36、UDP使用connect的好处**

1:UDP中可以使用connect系统调用

2:UDP中connect操作与TCP中connect操作有着本质区别.TCP中调用connect会引起三次握手,client与server建立连结.UDP中调用connect内核仅仅把对端ip&port记录下来.

3:UDP中可以多次调用connect,TCP只能调用一次connect.UDP多次调用connect有两种用途:1,指定一个新的ip&port连结.2,断开和之前的ip&port的连结.指定新连结,直接设置connect第二个参数即可.断开连结,需要将connect第二个参数中的sin\_family设置成 AF\_UNSPEC即可.

4:UDP中使用connect可以提高效率.原因如下:普通的UDP发送两个报文,内核做了如下:#1:建立连结#2:发送报文#3:断开连结#4:建立连结#5:发送报文#6:断开连结

采用connect方式的UDP发送两个报文内核如下处理:#1:建立连结#2:发送报文#3:发送报文另外一点,每次发送报文内核都由可能要做路由查询.

5:采用connect的UDP发送接受报文可以调用send,write和recv,read操作.当然也可以调用sendto,recvfrom.调用sendto的时候第五个参数必须是NULL,第六个参数是0.调用recvfrom,recv,read系统调用只能获取到先前connect的ip&port发送的报文.  
UDP中使用connect的好处:1:会提升效率.前面已经描述了.2:高并发服务中会增加系统稳定性.原因:假设client A 通过非connect的UDP与server B,C通信.B,C提供相同服务.为了负载均衡,我们让A与B,C交替通信.A与 B通信IPa:PORTa <----> IPb:PORTbA与 C通信IPa:PORTa' <---->IPc:PORTc  
假设PORTa与 PORTa'相同了(在大并发情况下会发生这种情况),那么就有可能出现A等待B的报文,却收到了C的报文.导致收报错误.解决方法内就是采用connect的UDP通信方式.在A中创建两个udp,然后分别connect到B,C。

**37、各层对应的网络设备（路由器、交换机、网关、网桥、集线器等），各层对应的协议**

#### OSI参考模型（开放系统互连参考模型）

#### 物理层

* 作用：实现比特流的透明传输，为数据链路层提供数据传输服务。
* 传输单位：比特（byte）
* 设备：网线，集线器

数据链路层

* 作用：建立数据链路链接，采用差错控制和流量控制，ARP（地址解析协议）解析获取MAC地址。
* 传输单位：帧
* 设备：二层交换机（网桥）、网卡。驱动程序
* PPP（端对端协议，point-to-point protocal）

网络层

* 作用：通过路由算法为数据报文选择最合适的传输路径，实现流量控制、拥塞控制。
* 传输单位：数据报（分组、IP报）
* 设备：路由器
* IP（网络协议）、ICMP（网际控制报文协议）、IGMP（网际组管理协议）、

传输层

* 作用：提供可靠的端对端的链接和数据传输服务，传输层队高层屏蔽了底层数据通信的细节
* 传输单位：报文、数据段
* 设备：计算机
* TCP（传输控制协议）、UDP（用户数据报协议）

##### 会话层

* 维护两个会话主机之间的链接
* 设备：计算机

##### 表示层

* 作用：负责数据的加密解密、数据压缩、恢复、根据不同应用要求将文件处理陈不同格式，体现在文件扩展名上

##### 应用层：

* 作用：完成应用程序之间的通信
* 设备：应用程序
* Telnet（远程终端协议）、FTP（文件传输协议）、SMTP（简单邮件传输协议）、HTTP（超文本传输协议）

**38、数据链路层的CSMA/CD协议**

.CSMA/CD（带冲突检测的载波侦听多路访问协议）

　　由于以太网（Ethernet）成为现存局域网络结构的绝大多形式，CSMA/CD（Carrier Sense Multiple ACcess/Collision Detection）载波监测多址接人协议也成为局域网采用最多的MAC协议。CSMA/CD适宜于总线型局域网拓扑结构的随机竞争型媒体访问控制。总线型网络允许同一时刻只有一个节点（Node）发送数据，一旦两个或以上节点同时发送数据，则会发生数据碰撞，数据不能正常发送和接收。CSMA/CD协议就是尽可能保证网络上同时只有一个节点发送数据，减小数据“碰撞”概率。

　　CSMA/CD工作过程

　　当MAC收到LLG发来的数据以后，首先监测网络电缆上是否具有数据，即载波传送。

　　发送数据前 先侦听信道是否空闲 ,若空闲 则立即发送数据.在发送数据时,边发送边继续侦听.若侦听到冲突,则立即停止发送数据。等待一段随机时间（通过二进制指数回退算法）,再重新尝试。

　　先听后发，边发边听，冲突停发，随即延迟后重发

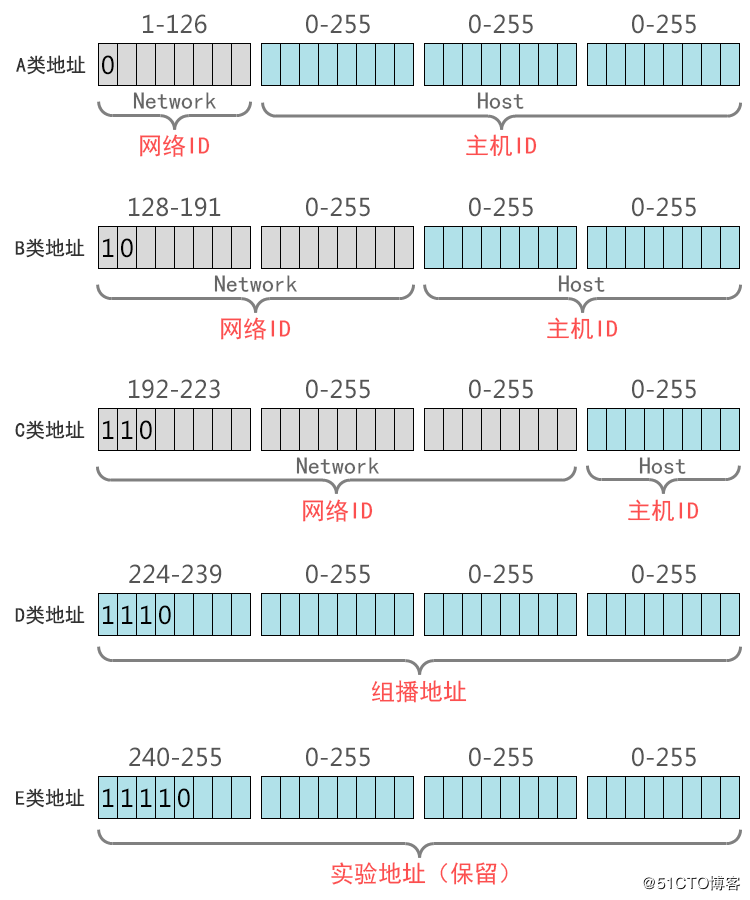
　　CSMA/CD使用条件

L/R >= 2\*T（L为发送数据包的大小，R为带宽，T为从发送端到接收端的传播时延）。

**39、IP地址的分类，子网划分**

### IP地址分类详解

IP地址分A、B、C、D、E五类，其中A、B、C这三类是比较常用的IP地址，D、E类为特殊地址。



①、A类地址

A类地址第1字节为网络地址（最高位固定是0），另外3个字节为主机地址。

A类地址范围：1.0.0.0 - 126.255.255.255，其中0和127作为特殊地址。

A类网络默认子网掩码为255.0.0.0，也可写作/8。

A类网络最大主机数量是256×256×256-2=166777214（减去1个主机位为0的网络地址和1个广播地址）。

在计算机网络中，主机ID全部为0的地址为网络地址，而主机ID全部为1的地址为广播地址，这2个地址是不能分配给主机用的。

②、B类地址

B类地址第1字节（最高位固定是10）和第2字节为网络地址，另外2个字节为主机地址。

B类地址范围：128.0.0.0 - 191.255.255.255。

B类网络默认子网掩码为255.255.0.0，也可写作/16。

B类网络最大主机数量256×256-2=6554。

③、C类地址

C类地址第1字节（最高位固定是110）、第2字节和第3个字节，另外1个字节为主机地址。

C类地址范围：192.0.0.0 - 223.255.255.255。

C类网络默认子网掩码为255.255.255.0，也可写作/24。

C类网络最大主机数量256-2=254。

④、D类地址

D类地址不分网络地址和主机地址，它的第1个字节的最高位固定是1110。

D类地址用于组播（也称为多播）的地址，无子网掩码。

D类地址范围：224.0.0.0 - 239.255.255.255。

⑤、E类地址

E类地址也不分网络地址和主机地址，它的第1个字节的最高位固定是11110。

E类地址范围：240.0.0.0 - 255.255.255.255。

其中240.0.0.0-255.255.255.254作为保留地址，主要用于Internet试验和开发，255.255.255.255作为广播地址。

VLSM子网划分的基本思想

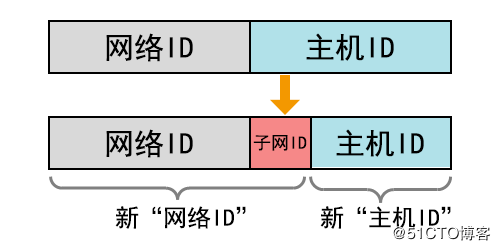
通过VLSM实现子网划分的基本思想很简单：就是借用现有网段的主机位的最左边某几位作为子网位，划分出多个子网。

①、把原来有类网络IPv4地址中的“网络ID”部分向“主机ID”部分借位

②、把一部分原来属于“主机ID”部分的位变成“网络ID”的一部分（通常称之为“子网ID”）。

③、原来的“网络ID”+“子网ID”=新“网络ID”。“子网ID”的长度决定了可以划分子网的数量。

如下示例图：



C类网络子网划分示例

①、等分为两个子网

将192.168.0.0 255.255.255.0这个网络等分成2个子网，并写出每个子网的地址信息？

分析：

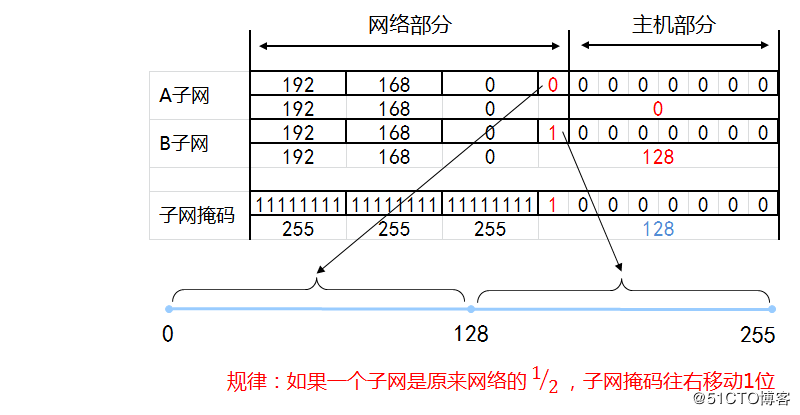
该网络子网掩码为/24，要划分为2个子网，要借用主机位1位作为子网位。

因为二进制数0和1按一位排列组合，只有这2种，分别为：0，1，如下图所示。

0是A子网  
1是B子网

借用主机1位，所以子网掩码+1位，由原来的255.255.255.0 （/24）变为255.255.255.128（/25）

结论：C类网络等分成2个子网，子网掩码往右移动1位，就能等分成2个子网，即2^1。

  
  
最终结果：

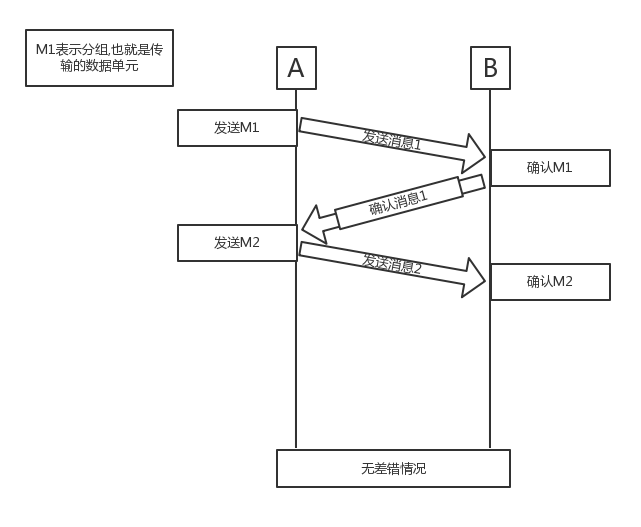
A子网的网络地址：192.168.0.0/25，可用地址（192.168.0.1～192.168.0.126），广播地址：192.168.0.127。

B子网的网络地址：192.168.0.128/25，可用地址（192.168.0.129～192.168.0.254），广播地址：192.168.0.255。

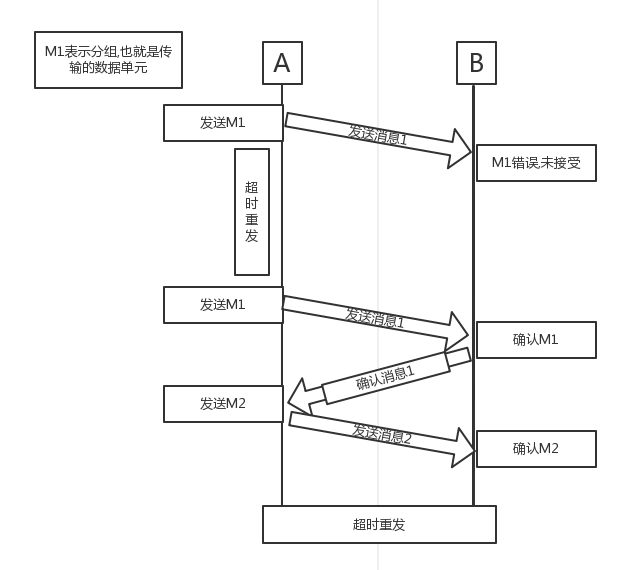
**40、TCP精髓问题：停止等待协议、连续ARQ协议**

**停止等待协议**   
停止等待协议是tcp保证传输可靠的重要途径,”停止等待”就是指发送完一个分组就停止发送,等待对方的确认,只有对方确认过,才发送下一个分组.

1:无差错情况:发送方发送分组,接收方在规定时间内收到,并且回复确认.发送方再次发送……



2:超时重传有以下三种情况:   
(1)分组丢失:发送方发送分组,接收方没有收到分组,那么接收方不会发出确认,只要发送方过一段时间没有收到确认,就认为刚才的分组丢了,那么发送方就会再次发送.   
(2):确认丢失:发送方发送成功,接收方接收成功,确认分组也被发送,但是分组丢失,那么到了等待时间,发送方没有收到确认,又会发送分组过去,此时接收方前面已经收到了分组,那么此时接收方要做的事就是:丢弃分组,重新发送确认.   
(3):传送延迟:发送方发送成功,接收方接收成功,确认分组也被发送,没有丢失,但是由于传输太慢,等到了发送方设置的时间,发送方又会重新发送分组,此时接收方要做的事情:丢弃分组,重新发送确认. 发送方如果收到两个或者多个确认,就停止发送,丢弃其他确认.



停止等待协议的优点是简单,但是缺点是信道的利用率太低。

**连续ARQ协议**

连续重发请求ARQ方案是指发送方可以连续发送一系列[信息帧](https://baike.so.com/doc/5305575-5540450.html)，即不用等前一帧被确认便可继续发送下一帧，效率大大提高。但在这种重发请求方案中，需要在发送方设置一个较大的缓冲存储空间(称作重发表)，用以存放若干待确认的以及待发送信息帧。当发送方收到对某信息帧的确认帧后，便可从重发表中将该信息帧删除。所以，连续重发请求ARQ方案的链路[传输效率](https://baike.so.com/doc/343547-363871.html)大大提高，但相应地需要更大的缓冲存储空间。

在这一协议中，当发送站点发送完一个[数据帧](https://baike.so.com/doc/665857-704844.html)后，不是停下来等待应答帧，而是可以连续再发送若干个数据帧。如果在此过程中又收到了接收端发来的应答帧，那么还可以接着发送数据帧。由于减少了等待时间，整个通信的吞吐量就提高了。

ARQ代表的是[自动重传请求](https://baike.so.com/doc/3853468-4045871.html)(Auto Repeat reQuest,ARQ)，而GBN与选择重传都属于其中。其中GBN的发送窗口>1，接收窗口=1，选择重传协议:发送窗口大小>1，接收窗口大于1。GBN协议中接收方可以发送累计确认帧ACK，而选择重传没有累计确认的特点。

**41、HTTP缓存机制（cache-control、Expires之类的一系列请求与相应的报头字段**

浏览器缓存，又称 HTTP 缓存，指的是：当我们浏览网站的时候，器存储会在本地存储一个副本，以便下次访问同个网址的时候可以不再连接服务器，直接使用本地的缓存。服务器端程序可以通过 HTTP Cache Headers 来控制缓存行为，减轻服务器的负担，缩短了响应时间，显著得提高网站的性能。

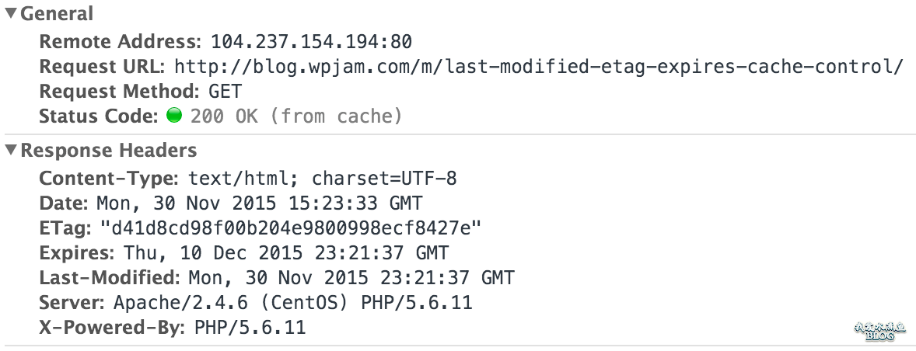
**HTTP Cache Headers**：

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（浏览器）如何处理缓存：

第一种是 **Expires**，比如：

Expires: Thu, 10 Dec 2015 23:21:37 GMT

在此日期之前，客户端都会认为缓存是有效的，第二次请求，浏览器不会连接服务器，直接从本地缓存中读取，比如下图：



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=3600

这里声明的是一个相对的秒数，表示从现在起，3600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，不适用与 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。

**条件请求：**

**Last-Modified / If-Modified-Since**

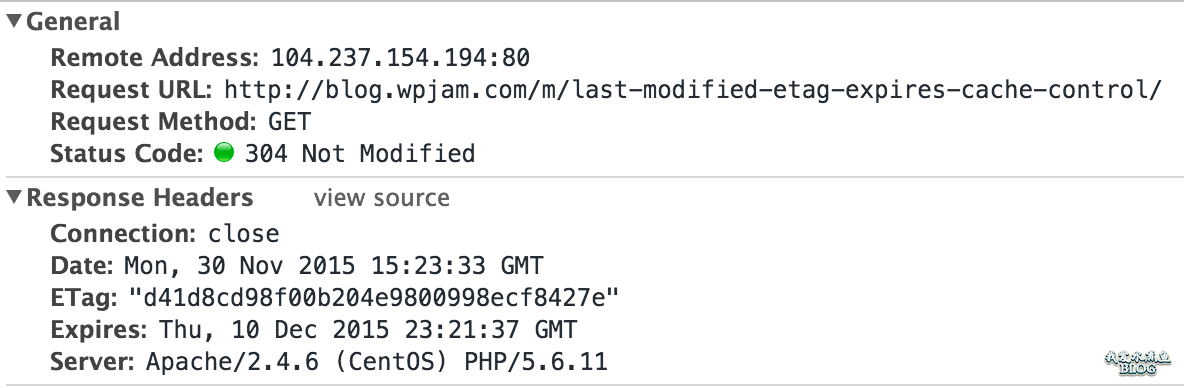
在浏览器第一次请求某一个 URL 时，服务器端的返回状态会是200，内容是你请求的资源，同时有一个 Last-Modified 的属性标记此文件在服务期端最后被修改的时间，格式类似这样：

Last-Modified: Mon, 30 Nov 2015 23:21:37 GMT

浏览器第二次请求此 URL 时，根据 HTTP 协议的规定，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: Mon, 30 Nov 2015 23:21:37 GMT

如果服务器端的资源没有变化，则自动返回 HTTP 304 （Not Changed）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。



**ETag / If-None-Match**

HTTP 协议规格说明定义 ETag 为“被请求变量的实体值”。 服务器单独负责判断记号是什么及其含义，并在 HTTP 响应头中将其传送到客户端，以下是服务器端返回的格式：

ETag: "d41d8cd98f00b204e9800998ecf8427e"

客户端的查询更新格式是这样的：

If-None-Match: W/"d41d8cd98f00b204e9800998ecf8427e"

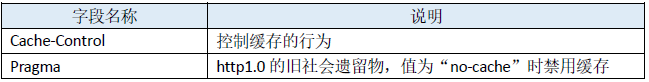
如果ETag没改变，则返回状态304，内容为空，这也和Last-Modified一样。

**如何使用 Last-Modified 和 Etags 如何帮助提高性能?**

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缓存是否过期。

**相应的报头字段：与缓存相关的HTTP首部字段**

1.通用首部字段



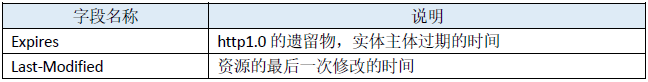
2.请求首部字段



3. 响应首部字段

http://imweb-io-1251594266.file.myqcloud.com/FgbZCbHn1jXX1zgb7xl9EzUNOC9q

4. 实体首部字段



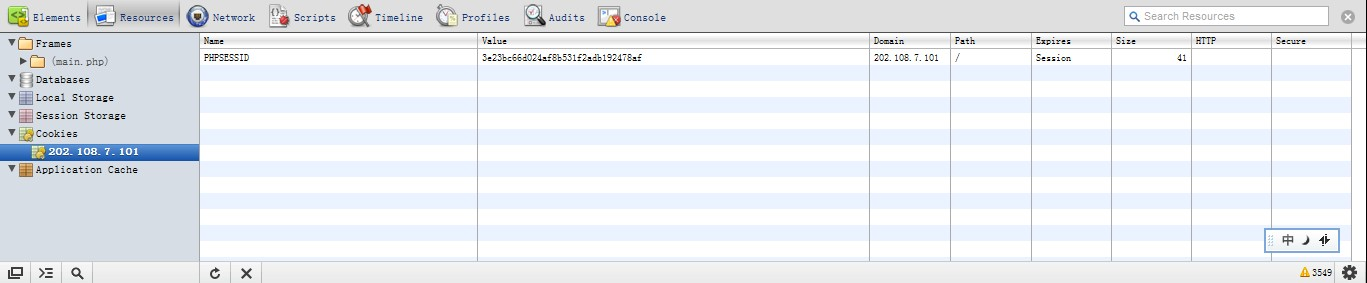
**42、session和cookied 区别，禁用cookie后怎么办**

Session是由应用服务器维持的一个服务器端的存储空间，用户在连接服务器时，会由服务器生成一个唯一的SessionID,用该SessionID 为标识符来存取服务器端的Session存储空间。而SessionID这一数据则是保存到客户端，用Cookie保存的，用户提交页面时，会将这一 SessionID提交到服务器端，来存取Session数据。这一过程，是不用开发人员干预的。所以一旦客户端禁用Cookie，那么Session也会失效。

　　服务器也可以通过URL重写的方式来传递SessionID的值，因此不是完全依赖Cookie。如果客户端Cookie禁用，则服务器可以自动通过重写URL的方式来保存Session的值，并且这个过程对程序员透明。

　　可以试一下，即使不写Cookie，在使用request.getCookies();取出的Cookie数组的长度也是1，而这个Cookie的名字就是JSESSIONID，还有一个很长的二进制的字符串，是SessionID的值。

来看一下，即使不写Cookie，通过chrome的开发者工具，也可以看到这么一组cookie的键值｛PHPSESSID : 3e23bc66d024af8b531f2adb192478af｝。



Session与Cookie区别-禁用cookie会不会影响session?

　　为什么会有cookie呢,大家应该知道，http是无状态的协议，客户每次读取web页面时，服务器都会去打开新的会话，而且服务器也不会自动维护客户的上下文信息(就是说你在这个页面存储的变量的值不会带到下一个页面去)，那么要怎么才能实现网上商店中的购物车机制呢，session就是一种保存上下文信息的机制，它是针对每一个用户的，变量的值保存在服务器端，通过SessionID来区分不同的客户,session是以cookie或URL重写为基础的，默认使用cookie来实现，系统会创造一个名为PHPSESSIONID的输出cookie，我们叫做session cookie,以区别persistent cookies,也就是我们通常所说的cookie,注意session cookie是存储于浏览器内存中的，并不是写到硬盘上的，这也就是我们刚才看到的PHPSESSIONID，我们通常情是看不到PHPSESSIONID的，但是当我们把浏览器的cookie禁止后，web服务器会采用URL重写的方式传递Sessionid，我们就可以在地址栏看到sessionid=3e23bc66d024af8b531f2adb192478af之类的字符串。

　　明白了原理，我们就可以很容易的分辨出persistent cookies和session cookie的区别了，网上那些关于两者安全性的讨论也就一目了然了，session cookie针对某一次会话而言，会话结束session cookie也就随着消失了，而persistent cookie只是存在于客户端硬盘上的一段文本（通常是加密的），而且可能会遭到cookie欺骗以及针对cookie的跨站脚本攻击，自然不如session cookie安全了。

　　正统的cookie分发是通过扩展HTTP协议来实现的，服务器通过在HTTP的响应头中加上一行特殊的指示以提示浏览器按照指示生成相应的cookie。从网络服务器观点看所有HTTP请求都独立于先前请求。就是说每一个HTTP响应完全依赖于相应请求中包含的信息状态管理机制克服了HTTP的一些限制并允许网络客户端及服务器端维护请求间的关系。在这种关系维持的期间叫做会话(session)。

　　Cookies是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器。IETF RFC 2965 HTTP State Management Mechanism 是通用cookie规范。网络服务器用HTTP头向客户端发送cookies，在客户终端，浏览器解析这些cookies并将它们保存为一个本地文件，它会自动将同一服务器的任何请求缚上这些cookies。

　　那么，在客户端禁用Cookie的时候，我们要怎么做呢，可以有以下两种方法

1.设置php.ini中的session.use\_trans\_sid = 1或者在PHP编译时打开–enable-trans-sid选项，让PHP自动通过URL传递session id。

2.如果是虚拟主机或者租用的服务器，无法去修改PHP.ini，那么可以手动通过URL传值，或者通过隐藏表单传递session id。说简单些就是自己去操纵sessionid这个唯一标识符，去鉴别用户即可。

**43、常用协议的端口**

使用TCP协议的常见端口主要有以下几种：  
（1）FTP：定义了文件传输协议，使用21端口。常说某某计算机开了FTP服务便是启动了文件传输服务。下载文件，上传主页，都要用到FTP服务。  
（2）Telnet：它是一种用于远程登陆的端口，用户可以以自己的身份远程连接到计算机上，通过这种端口可以提供一种基于DOS模式下的通信服务。如以前的BBS是纯字符界面的，支持BBS的服务器将23端口打开，对外提供服务。  
（3）SMTP：定义了简单邮件传送协议，现在很多邮件服务器都用的是这个协议，用于发送邮件。如常见的免费邮件服务中用的就是这个邮件服务端口，所以在电子邮件设置中常看到有这么SMTP端口设置这个栏，服务器开放的是25号端口  
（4）POP3：它是和SMTP对应，POP3用于接收邮件。通常情况下，POP3协议所用的是110端口。  
使用UDP协议端口常见的有：  
（1）HTTP：超文本传输协议。上网浏览网页时，就得在提供网页资源的计算机上打开80号端口以提供服务。常说"WWW服务"、"Web服务器"用的就是这个端口。  
（2）DNS：用于域名解析服务，这种服务在Windows NT系统中用得最多的。DNS用的是53号端口。  
（3）SNMP：简单网络管理协议，使用161号端口，是用来管理网络设备的。  
另外代理服务器常用以下端口：  
（1）HTTP协议代理服务器常用端口号：80/8080/3128/8081/9080  
（2）SOCKS代理协议服务器常用端口号：1080  
（3）FTP协议代理服务器常用端口号：21  
（4）Telnet协议代理服务器常用端口：23

**44、三层交换机和路由器的区别**

1. 主要功能不同

虽然三层交换机与路由器都具有路由功能，但我们不能因此而把它们等同起来，正如现在许多网络设备同时具备多种传统网络设备功能一样，就如现在有许多宽带路 由器不仅具有路由功能，还提供了交换机端口、硬件防火墙功能，但不能把它与交换机或者防火墙等同起来一样。因为这些路由器的主要功能还是路由功能，其它功 能只不过是其附加功能，其目的是使设备适用面更广、使其更加实用。这里的三层交换机也一样，它仍是交换机产品，只不过它是具备了一些基本的路由功能的交换机，它的主要功能仍是数据交换。也就是说它同时具备了数据交换和路由转发两种功能，但其主要功能还是数据交换；而路由器仅具有路由转发这一种主要功能。

2. 主要适用的环境不一样

三层交换机的路由功能通常比较简单，因为它所面对的主要是简单的局域网连接。正因如此，三层交换机的路由功能通常比较简单，路由路径远没有路由器那么复杂。它用在局域网中的主要用途还是提供快速数据交换功能，满足局域网数据交换频繁的应用特点。

而路由器则不同，它的设计初衷就是为了满足不同类型的网络连接，虽然也适用于局域网之间的连接，但它的路由功能更多的体现在不同类型网络之间的互联上，如 局域网与广域网之间的连接、不同协议的网络之间的连接等，所以路由器主要是用于不同类型的网络之间。它最主要的功能就是路由转发，解决好各种复杂路由路径 网络的连接就是它的最终目的，所以路由器的路由功能通常非常强大，不仅适用于同种协议的局域网间，更适用于不同协议的局域网与广域网间。它的优势在于选择 最佳路由、负荷分担、链路备份及和其他网络进行路由信息的交换等等路由器所具有功能。为了与各种类型的网络连接，路由器的接口类型非常丰富，而三层交换机 则一般仅同类型的局域网接口，非常简单。

3. 性能体现不一样

从技术上讲，路由器和三层交换机在数据包交换操作上存在着明显区别。路由器一般由基于微处理器的[软件](http://software.it168.com/)路 由引擎执行数据包交换，而三层交换机通过硬件执行数据包交换。三层交换机在对第一个数据流进行路由后，它将会产生一个MAC地址与IP地址的映射表，当同 样的数据流再次通过时，将根据此表直接从二层通过而不是再次路由，从而消除了路由器进行路由选择而造成网络的延迟，提高了数据包转发的效率。同时，三层交 换机的路由查找是针对数据流的，它利用缓存技术，很容易利用ASIC技术来实现，因此，可以大大节约成本，并实现快速转发。而路由器的转发采用最长匹配的方式，实现复杂，通常使用软件来实现，转发效率较低。

正因如此，从整体性能上比较的话，三层交换机的性能要远优于路由器，非常适用于数据交换频繁的局域网中；而路由器虽然路由功能非常强大，但它的数据包转发 效率远低于三层交换机，更适合于数据交换不是很频繁的不同类型网络的互联，如局域网与互联网的互联。如果把路由器，特别是高档路由器用于局域网中，则在相 当大程度上是一种浪费（就其强大的路由功能而言），而且还不能很好地满足局域网[通信](http://tele.it168.com/)性能需求，影响子网间的正常通信。

综上所述，三层交换机与路由器之间还是存在着非常大的本质区别的。无论从哪方面来说，在局域网中进行多子网连接，最好还选用三层交换机，特别是在不同子网 数据交换频繁的环境中。一方面可以确保子网间的通信性能需求，另一方面省去了另外购买交换机的投资。当然，如果子网间的通信不是很频繁，采用路由器也无可 厚非，也可达到子网安全隔离相互通信的目的。具体要根据实际需求来定。

**45、什么是TCP的自连接，如何解决**

tcp自连接，就是出现源ip和源端口通目的ip和目的端口完全相同的情况

## 如何解决：服务端的监听端口避免使用客户端的 ephemeral port

**46、单播、多播（组播）和广播的区别**

一、单播：   
主机之间“一对一”的通讯模式，网络中的交换机和路由器对数据只进行转发不进行复制。如果10个客 户机需要相同的数据，则服务器需要逐一传送，重复10次相同的工作。但由于其能够针对每个客户的及 时响应，所以现在的网页浏览全部都是采用IP单播协议。网络中的路由器和交换机根据其目标地址选择传输路径，将IP单播数据传送到其指定的目的地。   
单播的优点：   
1. 服务器及时响应客户机的请求   
2. 服务器针对每个客户不通的请求发送不通的数据，容易实现个性化服务。   
单播的缺点：   
1. 服务器针对每个客户机发送数据流，服务器流量＝客户机数量×客户机流量；在客户数量大、每个客户机流量大的流媒体应用中服务器不堪重负。   
2. 现有的网络带宽是金字塔结构，城际省际主干带宽仅仅相当于其所有用户带宽之和的5％。如果全部使用单播协议，将造成网络主干不堪重负。现在的P2P应用就已经使主干经常阻塞，只要有5％的客户在全速使用网络，其他人就不要玩了。而将主干扩展20倍几乎是不可能。   
二、 广播：   
 主机之间“一对所有”的通讯模式，网络对其中每一台主机发出的信号都进行无条件复制并转发，所有主机都可以接收到所有信息（不管你是否需要），由于其不用路径选择，所以其网络成本可以很低廉。有线电视网就是典型的广播型网络，我们的电视机实际上是接受到所有频道的信号，但只将一个频道的信号还原成画面。在数据网络中也允许广播的存在，但其被限制在二层交换机的局域网范围内，禁止广播数据穿过路由器，防止广播数据影响大面积的主机。   
广播的优点：   
 1. 网络设备简单，维护简单，布网成本低廉   
 2. 由于服务器不用向每个客户机单独发送数据，所以服务器流量负载极低。   
广播的缺点：   
 1.无法针对每个客户的要求和时间及时提供个性化服务。   
 2. 网络允许服务器提供数据的带宽有限，客户端的最大带宽＝服务总带宽。例如有线电视的客户端的线路支持100个频道（如果采用数字压缩技术，理论上可以提供500个频道），即使服务商有更大的财力配置更多的发送设备、改成光纤主干，也无法超过此极限。也就是说无法向众多客户提供更多样化、更加个性化的服务。   
3. 广播禁止在Internet宽带网上传输。   
三、组播：   
 主机之间“一对一组”的通讯模式，也就是加入了同一个组的主机可以接受到此组内的所有数据，网络中的交换机和路由器只向有需求者复制并转发其所需数据。主机可以向路由器请求加入或退出某个组，网络中的路由器和交换机有选择的复制并传输数据，即只将组内数据传输给那些加入组的主机。这样既能一次将数据传输给多个有需要（加入组）的主机，又能保证不影响其他不需要（未加入组）的主机的其他通讯。   
组播的优点：   
1. 需要相同数据流的客户端加入相同的组共享一条数据流，节省了服务器的负载。具备广播所具备的优点。   
2. 由于组播协议是根据接受者的需要对数据流进行复制转发，所以服务端的服务总带宽不受客户接入端带宽的限制。IP协议允许有2亿6千多万个（268435456）组播，所以其提供的服务可以非常丰富。   
3. 此协议和单播协议一样允许在Internet宽带网上传输。   
组播的缺点：   
1．与单播协议相比没有纠错机制，发生丢包错包后难以弥补，但可以通过一定的容错机制和QOS加以弥补。   
2．现行网络虽然都支持组播的传输，但在客户认证、QOS等方面还需要完善，这些缺点在理论上都有成熟的解决方案，只是需要逐步推广应用到现存网络当中。

47、当应用程序调用Send之后怎么判断对方是否成功接收

要应用层发送一个确认。TCP 的 ACK 表示对方的协议栈已经收到了你发的数据，不代表对方的应用程序收到了你发的消息。

**48、静态路由和动态路由各自的优缺点**  
**1.静态路由的优点和缺点**

静态路由的优点：  
占用的 CPU 处理时间少。  
便于管理员了解路由。  
易于配置。  
静态路由的缺点：  
配置和维护耗费时间。  
配置容易出错，尤其对于大型网络。  
需要管理员维护变化的路由信息。  
不能随着网络的增长而扩展；维护会越来越麻烦。  
需要完全了解整个网络的情况才能进行操作。  
**2.动态路由的优点和缺点**  
动态路由的优点：  
增加或删除网络时，管理员维护路由配置的工作量较少。  
网络拓扑结构发生变化时，协议可以自动做出调整。  
配置不容易出错。  
扩展性好，网络增长时不会出现问题。  
动态路由的缺点：  
需要占用路由器资源（CPU 时间、内存和链路带宽）。  
管理员需要掌握更多的网络知识才能进行配置、验证和故障排除工作。

**49、交换机和路由器的区别**

1. 交换机是用来连接局域网的，路由器是可以跨越局域网的。
2. 路由器会在局域网自动分配IP，实现虚拟拨号，交换机只是用来分配网络数据的。
3. 路由器在网络层，路由器根据IP地址寻址，路由器可以处理TCP/IP协议，交换机不可以。交换机在中继层，交换机根据MAC地址寻址。
4. 路由器能将一个IP分配给超多个主机使用，主机对外ip也是同一个。而交换机是可以将不同的主机连接起来，对外表现的IP也可各有不同。
5. 路由器提供防火墙的服务，交换机不能提供该功能。
6. **为什么有时ping服务器第一包丢失**

RFC 826号文档中有如下描述，如果没有ARP映射，则地址解析模块将通知调用方（在这里也就是上层的IP）它将会丢弃报文（当然在这里地址解析模块假设的是这个被丢弃的分组将会被高层重传，然而IP并没有数据恢复这个功能）。

有时发送ping命令，并没有网关的ARP地址，所以会产生丢包现象。

**51、tcp 阻塞socket send recv需要注意的操作**

**在阻塞条件下，read/recv/msgrcv的行为**::

  1、如果没有发现数据在网络缓冲中会一直等待，

  2、当发现有数据的时候会把数据读到用户指定的缓冲区，但是如果这个时候读到的数据量比较少，比参数中指定的长度要小，read 并不会一直等待下去，而是立刻返回。

**read 的原则**::是数据在不超过指定的长度的时候有多少读多少，没有数据就会一直等待。

 所以一般情况下::我们读取数据都需要采用循环读的方式读取数据，因为一次read 完毕不能保证读到我们需要长度的数据，read 完一次需要判断读到的数据长度再决定是否还需要再次读取。

写(send/write/msgsnd)::

  写的本质也不是进行发送操作,而是把用户态的数据copy 到系统底层去,然后再由系统进行发送操作,send，write返回成功，只表示数据已经copy 到底层缓冲,而不表示数据已经发出,更不能表示对方端口已经接收到数据.  
**对于write(或者send)而言**

**阻塞情况下:**: //阻塞情况下，write会将数据发送完。(不过可能被中断)

**在阻塞的情况下**，是会一直等待，直到write 完，全部的数据再返回．这点行为上与读操作有所不同。

原因::

读，究其原因主要是读数据的时候我们并不知道对端到底有没有数据，数据是在什么时候结束发送的，如果一直等待就可能会造成死循环，所以并没有去进行这方面的处理；写，而对于write, 由于需要写的长度是已知的，所以可以一直再写，直到写完．不过问题是write 是可能被打断吗，造成write 一次只write 一部分数据, 所以write 的过程还是需要考虑循环write, 只不过多数情况下一次write 调用就可能成功.

**52、TCP封包和拆包**

封包就是给一段数据加上包头，这样一来数据包就分为包头和包体两部分内容了。包头其实上是个大小固定的结构体，其中有个结构体成员变量表示包体的长度，这是个很重要的变量，其他的结构体成员可根据需要自己定义。根据包头长度固定以及包头中含有包体长度的变量就能正确的拆分出一个完整的数据包。

拆包就是根据包头以及包头中表示包长的结构体的变量，去掉包头的过程。

大概过程描述如下:  
 A,为每一个连接动态分配一个缓冲区,同时把此缓冲区和SOCKET关联,常用的是通过结构体关联.  
  B,当接收到数据时首先把此段数据存放在缓冲区中.  
  C,判断缓存区中的数据长度是否够一个包头的长度,如不够,则不进行拆包操作.  
  D,根据包头数据解析出里面代表包体长度的变量.  
  E,判断缓存区中除包头外的数据长度是否够一个包体的长度,如不够,则不进行拆包操作.  
  F,取出整个数据包.这里的"取"的意思是不光从缓冲区中拷贝出数据包,而且要把此数据包从缓存区中删除掉.删除的办法就是把此包后面的数据移动到缓冲区的起始地址.

1. **怎样理解阻塞非阻塞与同步异步的区别**

**1，同步与异步**  
同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)

所谓同步，就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。

而异步则是相反，\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在\*调用\*发出后，\*被调用者\*通过状态、通知来通知调用者，或通过回调函数处理这个调用。

**2，阻塞与非阻塞**

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。  
非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

1. **epoll：EPOLLLT和EPOLLET的区别**

Level-triggered ：水平触发，缺省模式。

edge-triggered ：边缘触发。

通知模式：

LT模式时，事件就绪时，假设对事件没做处理，内核会反复通知事件就绪

ET模式时，事件就绪时，假设对事件没做处理，内核不会反复通知事件就绪

**1.调用epoll\_ctl，ADD或者MOD事件EPOLLIN**

LT：如果此时缓存区没有可读数据，则epoll\_wait不会返回EPOLLIN，如果此时缓冲区有可读数据，则epoll\_wait会持续返回EPOLLIN

ET：如果此时缓存区没有可读数据，则epoll\_wait不会返回EPOLLIN，如果此时缓冲区有可读数据，则epoll\_wait会返回一次EPOLLIN

**2.调用epoll\_ctl，ADD或者MOD事件EPOLLOUT**

LT：如果不调用epoll\_ctl将EPOLLOUT修改为EPOLLIN，则epoll\_wait会持续返回EPOLLOUT(前提条件是写缓冲区未满)

ET：epoll\_wait只会返回一次EPOLLOUT

1. **epoll事件驱动框架使用注意事项**

epoll的坑点主要是其最初设计和实现的时候，没有对多线程、多进程这种scale-up和load-balance问题进行考虑，所以随着互联网并发和流量越来越大，越来越多的epoll flag和kernel flag被引入来修补相关问题；而来epoll的用户态空间操作接口是file descriptor，内核态管理接口是file descripton，有些情况下两者不是对应关系，会导致程序的行为很奇怪。假设通过dup用户态就有两个fd共享同一个底层的file description，此时关闭原先注册epoll event事件的fd而不调用epoll\_ctl取消事件侦听，那么底层的epoll event事件订阅就没有真正被取消；此时上层的应用程序看来现象就是即便关闭了fd，但是epoll\_wait()还是会不断返回关闭了的fd的事件信息，更糟糕的是fd已经关闭，我们无法通过epoll\_ctl再次取消这个事件侦听了，因为fd是epoll控制底层事件的唯一入口，即便相同引用底层的其他fd也不行，对此你无能为力。

1. **Linux-socket的close和shutdown区别及应用场景**

**Shutdown的定义**

#include<sys/socket.h>

int shutdown(int sockfd,int how);

**how的方式有三种分别是**：  
　　　　SHUT\_RD（0）：关闭sockfd上的读功能，此选项将不允许sockfd进行读操作。即该套接字不再接受数据，任何当前在套接字接受缓冲区的数据将被丢弃。进程将不能对该套接字发出任何读操作。对TCP套接字该调用之后接受到的任何数据将被确认然后无声的丢弃掉。  
　　　　SHUT\_WR（1）：关闭sockfd的写功能，此选项将不允许sockfd进行写操作，即进程不能在对此套接字发出写操作。  
　　　　SHUT\_RDWR（2）：关闭sockfd的读写功能，相当于调用shutdown两次：首先是以SHUT\_RD,然后以SHUT\_WR。

　　成功则返回0，错误返回-1，错误码errno：EBADF表示sockfd不是一个有效描述符；ENOTCONN表示sockfd未连接；ENOTSOCK表示sockfd是一个文件描述符而不是socket描述符。

shutdown()的效果是累计的，不可逆转的。既如果关闭了一个方向数据传输，那么这个方向将会被关闭直至完全被关闭或删除，而不能重新被打开。如果第一次调用了shutdown(0)，第二次调用了shutdown(1)，那么这时的效果就相当于shutdown(2)，也就是双向关闭socket。

**close的定义**

#include<unistd.h>

int close(int fd);

　　关闭读写。成功则返回0，错误返回-1，错误码errno：EBADF表示fd不是一个有效描述符；EINTR表示close函数被信号中断；EIO表示一个IO错误。

多进程

1.如果有多个进程共享一个套接字，close每被调用一次，计数减1，直到计数为0时，也就是所用进程都调用了close，套接字将被释放。

2.在多进程中如果一个进程中shutdown(sfd, SHUT\_RDWR)后其它的进程将无法进行通信。如果一个进程close(sfd)将不会影响到其它进程，得自己理解引用计数的用法了。有Kernel编程

使用场景：

通常来说，socket是双向的，即数据是双向通信的。但有些时候，你会想在socket上实现单向的socket，即数据往一个方向传输。单向的socket便称为半开放Socket。要实现半开放式，需要用到shutdown()函数。

一般来说，半开放socket适用于以下场合:  
　　1.当你想要确保所有写好的数据已经发送成功时。如果在发送数据的过程中，网络意外断开或者出现异常，系统不一定会返回异常，这是你可能以为对端已经接收到数据了。这时需要用shutdown()来确定数据是否发送成功，因为调用shutdown()时只有在缓存中的数据全部发送成功后才会返回。

2.想用一种方法来捕获程序潜在的错误，这错误可能是因为往一个不能写的socket上写数据，也有可能是在一个不该读操作的socket上读数据。当程序尝试这样做时，将会捕获到一个异常，捕获异常对于程序排错来说是相对简单和省劲的。

**57、TCP带外数据**

答: 传输层协议使用带外数据(out-of-band,OOB)来发送一些重要的数据,如果通信一方有重要的数据需要通知对方时,协议能够将这些数据快速地发送到对方.为了发送这些数据,协议一般不使用与普通数据相同的通道,而是使用另外的通道实现.

带外数据即就是优先数据，linux系统的套接字机制支持低层协议发送和接受带外数据.但是TCP协议没有真正意义上的带外数据.为了发送重要协议,TCP提供了一种称为紧急模式(urgent mode)的机制。TCP在报文头中设置URG位,表示进入紧急模式.接收方可以对紧急模式采取特殊的处理.



很容易看出来,这种方式数据不容易被阻塞,可以通过在我们的服务器端程序里面捕捉SIGURG信号来及时接受数据或者使用带OOB标志的recv函数来接受.使用send和recv函数时，可以指定最后一个参数flags为MSG\_OOB来发送接收带外数据。

带外数据发送过程：和普通数据发送过程一样，只是在数据字符串最后一个字符处设置一个标记，目标端接收到字符串，也是存放在一个描述符号中，但是会将指针指向最后一个字符上；接收进程发现字符上有带外数据标记，会发送信号SIGURG（紧急数据），即带外数据的优先级体现在信号的优先。  
 另一处解释:

<https://www.cnblogs.com/c-slmax/p/5553857.html>

**58、网桥的作用**

网桥是一个局域网与另一个局域网之间建立连接的桥梁.

[网桥](http://www.so.com/s?q=%E7%BD%91%E6%A1%A5&ie=utf-8&src=internal_wenda_recommend_textn)工作在[数据链路层](http://www.so.com/s?q=%E6%95%B0%E6%8D%AE%E9%93%BE%E8%B7%AF%E5%B1%82&ie=utf-8&src=internal_wenda_recommend_textn)，将两个[LAN](http://www.so.com/s?q=LAN&ie=utf-8&src=internal_wenda_recommend_textn)(局域网)连起来，根据[MAC地址](http://www.so.com/s?q=MAC%E5%9C%B0%E5%9D%80&ie=utf-8&src=internal_wenda_recommend_textn)来转发帧，可以看作一个“低层的路由器”（路由器工作在[网络层](http://www.so.com/s?q=%E7%BD%91%E7%BB%9C%E5%B1%82&ie=utf-8&src=internal_wenda_recommend_textn)，根据[网络地址](http://www.so.com/s?q=%E7%BD%91%E7%BB%9C%E5%9C%B0%E5%9D%80&ie=utf-8&src=internal_wenda_recommend_textn)如IP地址进行转发）。   
  
网桥的功能在延长网络跨度上类似于[中继器](http://www.so.com/s?q=%E4%B8%AD%E7%BB%A7%E5%99%A8&ie=utf-8&src=internal_wenda_recommend_textn)，然而它能提供智能化连接服务，即根据帧的终点地址处于哪一[网段](http://www.so.com/s?q=%E7%BD%91%E6%AE%B5&ie=utf-8&src=internal_wenda_recommend_textn)来进行转发和滤除。网桥对[站点](http://www.so.com/s?q=%E7%AB%99%E7%82%B9&ie=utf-8&src=internal_wenda_recommend_textn)[所处](http://www.so.com/s?q=%E6%89%80%E5%A4%84&ie=utf-8&src=internal_wenda_recommend_textn)网段的了解是靠“自学习”实现的。   
使用网桥进行互连克服了[物理](http://www.so.com/s?q=%E7%89%A9%E7%90%86&ie=utf-8&src=internal_wenda_recommend_textn)限制，这意味着构成LAN的数据站总数和网段数很容易扩充。   
　　 网桥纳入存储和转发功能可使其适应于连接使用不同[MAC协议](http://www.so.com/s?q=MAC%E5%8D%8F%E8%AE%AE&ie=utf-8&src=internal_wenda_recommend_textn)的两个LAN。 因而构成一个不同LAN混[连在一起](http://www.so.com/s?q=%E8%BF%9E%E5%9C%A8%E4%B8%80%E8%B5%B7&ie=utf-8&src=internal_wenda_recommend_textn)的[混合网络](http://www.so.com/s?q=%E6%B7%B7%E5%90%88%E7%BD%91%E7%BB%9C&ie=utf-8&src=internal_wenda_recommend_textn)环境。   
　　 网桥的中继功能仅仅依赖于MAC帧的地址，因而对高层协议完全透明。   
　　 网桥将一个较大的LAN分成段，有利于改善可靠性、[可用性](http://www.so.com/s?q=%E5%8F%AF%E7%94%A8%E6%80%A7&ie=utf-8&src=internal_wenda_recommend_textn)和安全性。

**59、防火墙的端口防护**

指通过对防火墙的端口开关的设置，关闭一些非必需端口，达到一定安全防护目的行为.

对某一个端口(例如telnet的端口为23)进行基于IP地址,方向,关键字等的检查过滤,判断是否准入或准出,以保护防火墙内的系统.

**60、有哪些私有保留地址**

A类：

10.X.X.X是私有地址（私有地址就是在互联网上不使用，而被用在局域网络中的地址）。

127.X.X.X是保留地址，用做循环测试用的。

B类：

172.16.0.0---172.31.255.255是私有地址。

169.254.X.X是保留地址。如果你的IP地址是自动获取IP地址，而你在网络上又没有找到可用的DHCP服务器。就会得到其中一个IP。

c类：

192.168.X.X是私有地址。（192.168.0.0---192.168.255.255)

总结：

**私有地址（内部局域网可以使用的）**

**A级：10.0.0.0 - 10.255.255.255**

**B级：172.16.0.0 - 172.31.255.255**

**C级：192.168.0.0 - 192.168.255.255**

**保留地址（特殊用途的）**

**A类：127.X.X.X**

**B类：169.254.X.X**

**61、Ping命令使用的那种报文**

Ping利用的就是ICMP ECHO和ICMP ECHO REPLY包来探测主机是否存在，这两个分别使用：ICMP ECHO(Type 8) 和ECHO Reply (Type 0)。

**62、两台笔记本电脑连起来后ping不通，你觉得有哪些问题造成的**

1 IP是否在[同一网段](https://www.baidu.com/s?wd=%E5%90%8C%E4%B8%80%E7%BD%91%E6%AE%B5&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)（ip地址配置有问题）

2 防火墙是否禁用了PING回复

3 网卡是否启用（tcp/ip安装的不完整）

4 还有就是网线具体接触是否正常

**63、Vlan的特点**

VLAN（Virtual Local Area Network）即虚拟局域网，是一种通过将局域网内的设备逻辑地而不是物理地划分成一个个网段从而实现虚拟工作组的新兴技术。IEEE于1999年颁布了用以标准化VLAN实现方案的802.1Q协议标准草案。

1. 限制广播域。广播域被限制在一个VLAN内，节省了带宽，提高了网络处理能力。

2. 增强局域网的安全性。不同VLAN内的报文在传输时是相互隔离的，即一个VLAN内的用户不能和其它VLAN内的用户直接通信，如果不同VLAN要进行通信，则需要通过路由器或三层交换机等三层设备。

3. 灵活构建虚拟工作组。用VLAN可以划分不同的用户到不同的工作组，同一工作组的用户也不必局限于某一固定的物理范围，网络构建和维护更方便灵活。

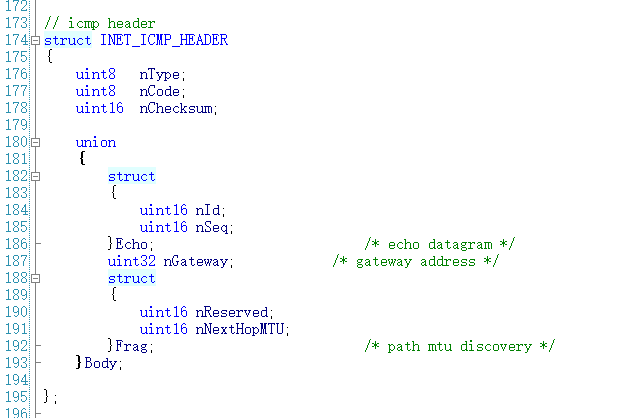
**64、ICMP是属于什么协议，处于那一层**

该协议是[TCP/IP协议](https://www.baidu.com/s?wd=TCP%2FIP%E5%8D%8F%E8%AE%AE&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)集中的一个子协议，属于网络层协议。

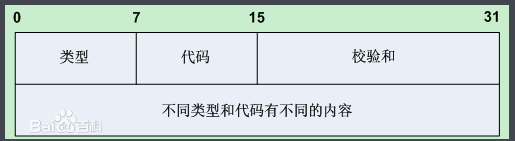
主要用于在主机与路由器之间传递控制信息，包括报告错误、交换受限控制和状态信息等。

当遇到IP数据无法访问目标、IP路由器无法按当前的[传输速率](https://www.baidu.com/s?wd=%E4%BC%A0%E8%BE%93%E9%80%9F%E7%8E%87&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)转发数据包等情况时，会自动发送ICMP消息。

我们可以通过[Ping命令](https://www.baidu.com/s?wd=Ping%E5%91%BD%E4%BB%A4&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)发送ICMP回应请求消息并记录收到ICMP回应回复消息，通过这些消息来对网络或主机的故障提供参考依据。



ICMP包头结构



ICMP包头格式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TYPE** | **CODE** | **Description** | **Query** | **Error** |
| 0 | 0 | Echo Reply——回显应答（Ping应答） | x |  |
| 3 | 0 | Network Unreachable——网络不可达 |  | x |
| 3 | 1 | Host Unreachable——主机不可达 |  | x |
| 3 | 2 | Protocol Unreachable——协议不可达 |  | x |
| 3 | 3 | Port Unreachable——端口不可达 |  | x |
| 3 | 4 | Fragmentation needed but no frag. bit set——需要进行分片但设置不分片比特 |  | x |
| 3 | 5 | Source routing failed——源站选路失败 |  | x |
| 3 | 6 | Destination network unknown——目的网络未知 |  | x |
| 3 | 7 | Destination host unknown——目的主机未知 |  | x |
| 3 | 8 | Source host isolated (obsolete)——源主机被隔离（作废不用） |  | x |
| 3 | 9 | Destination network administratively prohibited——目的网络被强制禁止 |  | x |
| 3 | 10 | Destination host administratively prohibited——目的主机被强制禁止 |  | x |
| 3 | 11 | Network unreachable for TOS——由于服务类型TOS，网络不可达 |  | x |
| 3 | 12 | Host unreachable for TOS——由于服务类型TOS，主机不可达 |  | x |
| 3 | 13 | Communication administratively prohibited by filtering——由于过滤，通信被强制禁止 |  | x |
| 3 | 14 | Host precedence violation——主机越权 |  | x |
| 3 | 15 | Precedence cutoff in effect——优先中止生效 |  | x |
| 4 | 0 | Source quench——源端被关闭（基本流控制） |  |  |
| 5 | 0 | Redirect for network——对网络重定向 |  |  |
| 5 | 1 | Redirect for host——对主机重定向 |  |  |
| 5 | 2 | Redirect for TOS and network——对服务类型和网络重定向 |  |  |
| 5 | 3 | Redirect for TOS and host——对服务类型和主机重定向 |  |  |
| 8 | 0 | Echo request——回显请求（Ping请求） | x |  |
| 9 | 0 | Router advertisement——路由器通告 |  |  |
| 10 | 0 | Route solicitation——路由器请求 |  |  |
| 11 | 0 | TTL equals 0 during transit——传输期间生存时间为0 |  | x |
| 11 | 1 | TTL equals 0 during reassembly——在数据报组装期间生存时间为0 |  | x |
| 12 | 0 | IP header bad (catchall error)——坏的IP首部（包括各种差错） |  | x |
| 12 | 1 | Required options missing——缺少必需的选项 |  | x |
| 13 | 0 | Timestamp request (obsolete)——时间戳请求（作废不用） | x |  |
| 14 |  | Timestamp reply (obsolete)——时间戳应答（作废不用） | x |  |
| 15 | 0 | Information request (obsolete)——信息请求（作废不用） | x |  |
| 16 | 0 | Information reply (obsolete)——信息应答（作废不用） | x |  |
| 17 | 0 | Address mask request——地址掩码请求 | x |  |
| 18 | 0 | Address mask reply——地址掩码应答 |  |  |

icmp类型+code列表

**65、IP组播有什么好处**

组播是指主机之间“一对一组”的通讯模式，也就是加入了同一个组的主机可以接受到此组内的所有数据，网络中的交换机和路由器只向有需求者复制并转发其所需数据。主机可以向路由器请求加入或退出某个组，网络中的路由器和交换机有选择的复制并传输数据，即只将组内数据传输给那些加入组的主机。这样既能一次将数据传输给多个有需要（加入组）的主机，又能保证不影响其他不需要（未加入组）的主机的其他通讯。

组播的优点：

1. 需要相同数据流的客户端加入相同的组共享一条数据流，节省了服务器的负载。具备广播所具备的优点。

2. 由于组播协议是根据接受者的需要对数据流进行复制转发，所以服务端的服务总带宽不受客户接入端带宽的限制。IP协议允许有2亿6千多万个（268435456）组播，所以其提供的服务可以非常丰富。

3. 此协议和单播协议一样允许在Internet宽带网上传输。

组播的缺点：

1．与单播协议相比没有纠错机制，发生丢包错包后难以弥补，但可以通过一定的容错机制和QOS加以弥补。

2．现行网络虽然都支持组播的传输，但在客户认证、QOS等方面还需要完善，这些缺点在理论上都有成熟的解决方案，只是需要逐步推广应用到现存网络当中。

**66、DNS欺骗的方式**

1>.DNS欺骗

在DNS的缓存还没有过期之前,如果在DNS的缓存中已经存在的记录,一旦有客户查询,DNS服务器将会直接返回缓存中的记录.

下面我们来看一个例子:

一台运行着Lunix的Internet主机,并且提供rlogin服务,它的IP地址为123.45.67.89,它使用的DNS服务器(即/etc/resolv.conf中指向的DNS服务器)的IP地址为98.76.54.32,某个客户端(IP地址为38.222.74.2)试图连接到unix主机的rlogin端口,假设unix主机的/etc/hosts.equiv文件中使用的是dns名称来允许目标主机的访问,那么unix主机会向IP为98.76.54.32的DNS服务器发出一个PTR记录的查询:

123.45.67.89 -> 98.76.54.32 [Query]

NQY: 1 NAN: 0 NNS: 0 NAD: 0

QY: 2.74.222.38.in-addr.arpa PTR

IP为98.76.54.32的DNS服务器中没有这个反向查询域的信息,经过一番查询,这个DNS服务器找到38.222.74.2和38.222.74.10为74.222.38.in-addr.arpa.的权威DNS服务器,所以它会向38.222.74.2发出PTR查询:

98.76.54.32 -> 38.222.74.2 [Query]

NQY: 1 NAN: 0 NNS: 0 NAD: 0

QY: 2.74.222.38.in-addr.arpa PTR

请注意,38.222.74.2是我们的客户端IP,也就是说这台机子是完全掌握在我们手中的.我们可以更改它的DNS记录,让它返回我们所需要的结果:

38.222.74.2 -> 98.76.54.32 [Answer]

NQY: 1 NAN: 2 NNS: 2 NAD: 2

QY: 2.74.222.38.in-addr.arpa PTR

AN: 2.74.222.38.in-addr.arpa PTR trusted.host.com

AN: trusted.host.com A 38.222.74.2

NS: 74.222.38.in-addr.arpa NS ns.sventech.com

NS: 74.222.38.in-addr.arpa NS ns1.sventech.com

AD: ns.sventech.com A 38.222.74.2

AD: ns1.sventech.com A 38.222.74.10

当98.76.54.32的DNS服务器收到这个应答后,会把结果转发给123.45.67.98,就是那台有rlogin服务的unix主机(也是我们的目标 :) ),并且98.76.54.32这台DNS服务器会把这次的查询结果缓存起来.

这时unix主机就认为IP地址为38.222.74.2的主机名为trusted.host.com,然后unix主机查询本地的/etc/hosts.equiv文件,看这台主机是否被允许使用rlogin服务,很显然,我们的欺骗达到了.

在unix的环境中,有另外一种技术来防止这种欺骗的发生,就是查询PTR记录后,也查询PTR返回的主机名的A记录,然后比较两个IP地址是否相同:

123.45.67.89 -> 98.76.54.32 [Query]

NQY: 1 NAN: 0 NNS: 0 NAD: 0

QY: trusted.host.com A

很不幸,在98.76.54.32的DNS服务器不会去查询这个记录,而会直接返回在查询2.74.222.38.in-addr.arpa时得到的并且存在缓存中的信息:

98.76.54.32 -> 123.45.67.89 [Query]

NQY: 1 NAN: 1 NNS: 2 NAD: 2

QY: trusted.host.com A

AN: trusted.host.com A 38.222.74.2

NS: 74.222.38.in-addr.arpa NS ns.sventech.com

NS: 74.222.38.in-addr.arpa NS ns1.sventech.com

AD: ns.sventech.com A 38.222.74.2

AD: ns1.sventech.com A 38.222.74.10

那么unix主机就认为38.222.74.2就是真正的trusted.host.com了,我们的目的达到了!

这种IP欺骗的条件是:你必须有一台Internet上的授权的DNS服务器,并且你能控制这台服务器,至少要能修改这台服务器的DNS记录,我们的欺骗才能进行.

2>.拒绝服务攻击 Denial of service

还是上面的例子,如果我们更改位于38.222.74.2的记录,然后对位于98.76.54.32的DNS服务器发出2.74.222.38.in-addr.arpa的查询,并使得查询结果如下:

因为74.222.38.in-addr.arpa完全由我们控制,所以我们能很方便的修改这些信息来实现我们的目的.

38.222.74.2 -> 98.76.54.32 [Answer]

NQY: 1 NAN: 2 NNS: 2 NAD: 2

QY: 2.74.222.38.in-addr.arpa PTR

AN: 2.74.222.38.in-addr.arpa PTR trusted.host.com

AN: www.company.com A 0.0.0.1

NS: 74.222.38.in-addr.arpa NS ns.sventech.com

NS: 74.222.38.in-addr.arpa NS ns1.sventech.com

AD: ns.sventech.com A 38.222.74.2

AD: ns1.sventech.com A 38.222.74.10

这样一来,使用98.76.54.32这台DNS服务器的用户就不能访问www.company.com了,因为这个IP根本就不存在!

3>.偷取服务 Theft of services

还是上面的例子,只是更改的查询结果如下:

38.222.74.2 -> 98.76.54.32 [Answer]

NQY: 1 NAN: 3 NNS: 2 NAD: 2

QY: 2.74.222.38.in-addr.arpa PTR

AN: 2.74.222.38.in-addr.arpa PTR trusted.host.com

AN: www.company.com CNAME www.competitor.com

AN: company.com MX 0 mail.competitor.com

NS: 74.222.38.in-addr.arpa NS ns.sventech.com

NS: 74.222.38.in-addr.arpa NS ns1.sventech.com

AD: ns.sventech.com A 38.222.74.2

AD: ns1.sventech.com A 38.222.74.10

这样一来,一个本想访问http://www.competitor.com的用户会被带到另外一个地方,甚至是敌对的公司的主页(想想把华为和北电联起来是什么样的感觉. :) ).并且发给company.com的邮件会被发送给mail.compertitor.com.(越来越觉得在网络上的日子不踏实! xxbin这样想)。

**67、reactor和proactor的区别**

两种I/O多路复用模式：Reactor和Proactor

一般地,I/O多路复用机制都依赖于一个事件多路分离器(Event Demultiplexer)。分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write事件处理器(Event Handler)。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。两个与事件分离器有关的模式是Reactor和Proactor。Reactor模式采用同步IO，而Proactor采用异步IO。

在Reactor中，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

而在Proactor模式中，处理器--或者兼任处理器的事件分离器，只负责发起异步读写操作。IO操作本身由操作系统来完成。传递给操作系统的参数需要包括用户定义的数据缓冲区地址和数据大小，操作系统才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获IO操作完成事件，然后将事件传递给对应处理器。比如，在windows上，处理器发起一个异步IO操作，再由事件分离器等待IOCompletion事件。典型的异步模式实现，都建立在操作系统支持异步API的基础之上，我们将这种实现称为“系统级”异步或“真”异步，因为应用程序完全依赖操作系统执行真正的IO工作。

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（类操作类似）。

在Reactor中实现读：

- 注册读就绪事件和相应的事件处理器

- 事件分离器等待事件

- 事件到来，激活分离器，分离器调用事件对应的处理器。

- 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

在Proactor中实现读：

- 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。

- 事件分离器等待操作完成事件

- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成。

- 事件分离器呼唤处理器。

- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

可以看出，两个模式的相同点，都是对某个IO事件的事件通知(即告诉某个模块，这个IO操作可以进行或已经完成)。在结构上，两者也有相同点：demultiplexor负责提交IO操作(异步)、查询设备是否可操作(同步)，然后当条件满足时，就回调handler；不同点在于，异步情况下(Proactor)，当回调handler时，表示IO操作已经完成；同步情况下(Reactor)，回调handler时，表示IO设备可以进行某个操作(can read or can write)。

**68、两台机器A-B进行TCP通讯，进程崩溃会怎么样进程死锁会怎么样进程或机器过载，反应变慢会怎么样进程死循环，拼命发消息会怎么样机器重启会怎么样机器死机会怎么样机器网卡抽风，丢包严重会怎么样交换机或路由器坏了或过载会怎么样路由器过热重启会怎么样A和B之间的带宽被别的服务占用了会怎么样如何诊断以上这些情况。如果A和B之间有防火墙，还会出哪些情况。**

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程

**69、Linux 中每个 TCP 连接最少占用多少内存**

维持一个tcp连接需要占用哪些资源，下面就总结一下最近学习的内容，不足之处，请读者多多指正。

**一个tcp连接需要：1，socket文件描述符；2，IP地址；3，端口；4，内存**

**TCP连接的四元组：源IP 源端口 目标IP 目标端口，这四元组构成了一个唯一的tcp连接。**

对于一台服务器，我们假设只有一个网卡，那么就对应一个唯一的IP地址，而**监听端口**，我们可以在1024-65535之间任选一个。通过这个监听端口，我们接收来自客户端的连接请求。那么，它的IP、端口已经确定了，下面就是讨论socket文件描述符合内存了。

**对于文件描述符fd，**每个tcp连接占用一个，那么一个文件描述符下的文件大约占1K字节，而内核对这块也有说明，文件描述符建议最多占用10%的内存，如果是8G内存，那么就相当于800M即80000,80万个文件描述符，当然，这个数据也可以通过linux参数调优进行调节，我在之前的一篇章节中也有讨论到，请大家参考：http://blog.csdn.net/fox\_hacker/article/details/41148115

**而对于内存，tcp连接归根结底需要双方接收和发送数据，那么就需要一个读缓冲区和写缓冲区**，这两个buffer在linux下最小为4096字节，可通过cat /proc/sys/net/ipv4/tcp\_rmem和cat /proc/sys/net/ipv4/tcp\_wmem来查看。所以，一个tcp连接最小占用内存为4096+4096 = 8k，那么对于一个8G内存的机器，在不考虑其他限制下，最多支持的并发量为：8\*1024\*1024/8 约等于100万。此数字为纯理论上限数值，在实际中，由于linux kernel对一些资源的限制，加上程序的业务处理，所以，8G内存是很难达到100万连接的，当然，我们也可以通过增加内存的方式增加并发量。

网上也有人做过相关试验，程序接收1024000个连接，共消耗7,5G内存，即每个连接消耗在8K左右。

**70、TCP 能否发送0字节的数据包**

1、tcp和udp都能发送0字节的数据包，windows下通过GetLastError查看返回值为0

2、tcp发送0字节的数据包后，接收方调用recv不会接收到该数据包

3、udp发送0字节的数据包后，接收方调用recvfrom能够接收到该数据包

**71、tcp 协议中为什么syn会消耗一个序号**

同步位syn=1，不携带数据段，所以需要消耗一个序列号

**72、tcp/ip数据包在互联网传输过程中，有哪些头会保持不变**

只说TCP。如果穿过了一层四层以上的设备，比如负载均衡器，那么什么保证都没有了，什么都会变，本质上来说是两条粘接在一起的TCP连接。还有一些介于这两者之间的功能，比如说FreeBSD有一个保护内部串号的模式，在TCP穿过防火墙的时候按特定规则修改TCP的序号。排除这些以外的话，穿过常见的防火墙、NAT一类设备之后，最不会改变的是ACK和SEQ号，这个大部分设备没有勇气去动，一动连接就可能出异常。TCP Flags自然也是不能动的，SYN、ACK、RST都是非常重要的标识。Data Offset、Window也不说了，跟数据和控制直接相关，能自动重新分割TCP的设备我没见过。Urgent Pointer，这个字段压根就没有人用。大部分TCP选项也能完好地穿过，但是偶尔会有设备会丢掉部分选项。总结来说TCP层不能穿过的通常只有端口号，然后Checksum会被重新计算。如果没有PAT设备（修改四层端口号的NAT设备）在中间，一般四层也是完好的。再说IP层，IP层其实能保持不变的字段比较少，有些设备会丢弃IP选项（虽然按协议本来应该是不允许的），Total Length跟数据有关的一般不会变，Id一般不变但偶尔也有重写的，Flag和Fragments和分片有关，重新分片的话会变。TTL每一条都会变，ToS本来设计上就是会被重写的。Protocol倒是不会变。IP源地址和目的地址会被NAT改变。

73、**Linux/unix socket编程并发的时候什么时候用进程（fork），什么时候用线程（池）？**

**答案一:**

1、进程：子进程是父进程的复制品。子进程获得父进程数据空间、堆和栈的复制品。

2、线程：相对与进程而言，线程是一个更加接近与执行体的概念，它可以与同进程的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

两者都可以提高程序的并发度，提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源管理和保护；而进程正相反。同时，线程适合于在SMP(Symmetric Multi-Processing，对称多处理结构的简称，是指在一个计算机上汇集了一组处理器(多CPU),各CPU之间共享内存子系统以及总线结构。)机器上运行，而进程则可以跨机器迁移。

**答案二:**

根本区别就一点：用多进程每个进程有自己的地址空间(address space)，线程则共享地址空间。所有其它区别都是由此而来的：

1、速度：线程产生的速度快，线程间的通讯快、切换快等，因为他们在同一个地址空间内。

2、资源利用率：线程的资源利用率比较好也是因为他们在同一个地址空间内。

3、同步问题：线程使用公共变量/内存时需要使用同步机制还是因为他们在同一个地址空间内。

**74.、如何测量网络发送速度？**

1、ping命令可以通过向目标网站发送数据包，从数据包的平均达到时间和丢包率来判断，从本地到目标网站的网络情况。

2、tracert命令是使用从本地到目标网站所在网络服务器的一系列网络节点的访问速度，网络节点最多支持显示30个。

**75.、如果将同一个listening socket加入多个epoll，是不是一个合理的设计？**

并不是很合理的设计，首先，一般来说，一个epoll单属于一个线程，多个线程中每个线程拥有一个epoll。考虑一种情况，多个epoll中都存在同一个监听套接字，那么当监听套接字连接队列不为空，原本多个线程阻塞在epoll\_wait，现在都被唤醒，但是实际上，仅仅一个线程才能从accept返回，其他的线程则会继续被阻塞。所有线程的唤醒和阻塞，导致了无用的开销。针对这种惊群现象，可以考虑nginx的做法，具体见链接。[Nginx中利用锁解决惊群现象的做法](https://www.cnblogs.com/549294286/p/6058811.html)

**76、Tcp客户端的正确关闭方式？**

1.1、客户端调用close()

1.2、客户端进程关闭

1.3、客户端调用shutdown()

1.4、客户端调用close()+SO\_LINGER选项

1.5、客户端崩溃

1.1与1.2等价，就算客户端进程关闭，系统内核也会自动close(socket)，且注意，当socket引用为0时才会真正调用close()，close()总是立即返回的，然后由系统尝试发送完内核缓冲区内的所有数据，接着才发送FIN。

说到这里，不得不谈谈TCP连接关闭的四次握手。可以看成是2组FIN, ACK。主动关闭的一方先发送FIN，收到ACK后，进入FIN\_WAIT2状态，此时也叫做“半关闭”状态，特别须要注意的是，此时客户端套接字依然可以接收数据包，但是不能发送数据包。 被动关闭的一方，此时收到FIN了，一般情况下都是由于read(socket)返回0，然后得知对方关闭，close(socket)后，另外一组FIN，ACK随之产生，此时主动方进入TIME\_WAIT状态。即四次握手完成。

以上即是正常情况下连接关闭的情形。

再看看1.3，shutdown()与close()主要有3点区别：shutdown()不理会引用计数与内核缓冲区内剩余待发数据包，直接发送FIN；shutdown()可以只关闭套接字某个方向的连接，例如关闭发送，关闭接收，或者2者都关闭；实际上shutdown(write)后，就是上面说的半关闭情形，依然可以完成四次握手。

再看看1.4，为什么要设置SO\_LINGER呢，SO\_LINGER的目的就是改变close()的默认行为，可以决定close()在哪个状态返回，或者让套接字立即发送RST，从而没有FIN的发送。接收方返回ECONNRESET错误，连接直接关闭。

再来总结下1.1-1.4，这么多关闭连接的方式，那么什么方式才是最好的呢？

择优选择的方式当然是考虑最恶劣的情况，对方主机崩溃或网络故障导致数据包传输停滞。RST不用考虑了，直接TIME\_WAIT状态都没，如果有网络故障，可能下次创建的套接字还会接收到已经被销毁的套接字的数据报。close()不能保证对方一定收到FIN。

close()+SO\_LINGER虽然能控制close()在收到ACK后返回，依然不能保证四次握手完成。

shutdown()先进入半关闭状态，再调用read()，返回0（收到对方FIN）则说明四次握手正常进行，此为最优方式。

**77、对一个已经关闭了的socket的server调用write操作**

当服务器close一个连接时，若client端接着发数据。根据TCP协议的规定，会收到一个RST响应，client再往这个服务器发送数据时，系统会发出一个SIGPIPE信号给进程，告诉进程这个连接已经断开了，不要再写了。

根据信号的默认处理规则SIGPIPE信号的默认执行动作是terminate(终止、退出),所以client会退出。若不想客户端退出可以把SIGPIPE设为SIG\_IGN 如: signal(SIGPIPE,SIG\_IGN); 这时SIGPIPE交给了系统处理。

**78、怎样实时判断socket链接状态**

对端正常close socket，或者进程退出（正常退出或崩溃），对端系统正常关闭这种情况下，协议栈会走正常的关闭状态转移，使用epoll的话，一般要判断如下几个情况：

1、处理可读事件时，在循环read后，返回结果为0。

2、处理可写事件时，write返回-1，errno为EPIPE。

3、EPOLLERR或EPOLLHUP或事件。

对端非正常断开，比如服务器断电，网线被拔掉这种情况下，协议栈无法感知，SO\_KEEPALIVE这个选项的超时事件太长并不实用，一般还是以应用层的heartbeat来及时发现。

**79、socket套接字在多线程发送数据时需要加锁吗**

socket是全双工的, 多线程同时 send 和 recv 没问题。

在操作系统层面, send 和 recv 均是原子操作, 多线程对同一个socket 发送数据, 无需加锁。

**80、TCP中已有SO\_KEEPALIVE选项，为什么还要在应用层加入心跳包机制？**

答案：主要是因为TCP协议中的SO\_KEEPALIVE有几个致命的缺陷：

1、keepalive只能检测连接是否存活，不能检测连接是否可用。比如服务器因为负载过高导致无法响应请求但是双方的连接仍然存在，此时keepalive无法判断该连接是否可用。

2、如果TCP连接中的另一方因为停电突然断网等非正常断开的现象，由于服务器端（被动连接/断开的一方）并不知道客户端已断开连接，此时若服务器正在发送数据，那么会导致数据发送失败并进行数据重传，由于重传包的优先级要高于keepalive的数据包，因此keepalive的数据包无法及时发送出去。

3、当重传超过一定次数，TCP协议会发送keepalive探测包到客户端，一旦探测包没有返回，服务器端会以keepaliveinterval的频率继续发送探测包，经过若干次重试，若服务器一直没有收到应答就会认为该TCP连接已经断开（默认时长是2小时），而2小时以内这个连接一直不会断开，浪费系统资源。

**相关知识点：**

一、保获的必要性：

1、许多防火墙等会自动关闭空闲的socket连接，TCP保活可以防止某些短时间的空闲socket不会被关闭；

2、对于一些非正常断开的连接，如断电，服务器端并不能检测到，为了回收资源，必须提供一种检测机制；

二、保活的两种方式

1、TCP协议自带的KeepAlive

2、应用层面的心跳包

**81、 TCP 协议下 socket 有可能丢包吗？**

**答案**：有可能。

TCP是一种可靠的、面向连接的字节流服务，可靠的服务是指，在网络连接正常的情况下，保证用户数据能够按照顺序的完整到达。但是，TCP是基于不可靠的网络来实现可靠的传输，肯定也会存在丢包的情况，一般在通信中发现缺少数据或者发生丢包，很可能是由于程序在发送或者接收的过程中出现了问题。

**举例1：**客户端A与服务器B建立TCP连接，A到B的传输过程中，A发送的字节流按顺序完整的到达B，但是由于B在对数据处理的过程中，导致数据的部分丢失（例如A发送大量的数据，且频率很高，B接收数据的时候可能由于缓冲区溢出、多线程同步等问题），那么B收到的数据就比理论上应该收到的少，产生丢包现象。

附上知乎上的一个讨论帖：<https://www.zhihu.com/question/53960871>

**82、如何理解TCP/IP, SPDY, WebSocket三者之间的关系**

答案：按照OSI网络分层模型，IP是网络层协议，TCP是传输层协议，而HTTP是应用层的协议。在这三者之间，SPDY和WebSocket都是与HTTP相关的协议，而TCP是HTTP底层的协议。

1、HTTP的不足（主要是性能方面）

1.1 HTTP的连接问题，HTTP客户端和服务器之间的交互是采用请求/应答模式，在客户端请求时，会建立一个HTTP连接，然后发送请求消息，服务端给出应答消息，然后连接就关闭了。（HTTP1.1开始支持长连接）

1.2 因为TCP连接的建立过程是有开销的，如果使用了SSL/TLS开销就更大。

1.3 在浏览器里，一个网页包含许多资源，包括HTML，CSS，JavaScript，图片等等，这样在加载一个网页时要同时与同一服务器建立多个连接。

1.4 HTTP消息头问题，现在的客户端会发送大量的HTTP消息头，由于一个网页可能需要50-100个请求，就会有相当大的消息头的数据量。

1.5 HTTP通信方式问题，HTTP的请求/应答方式的会话都是客户端发起的，缺乏服务器通知客户端的机制，在需要通知的场景，如聊天室，游戏，客户端应用需要不断地轮询服务器。

SPDY和WebSocket是从不同的角度来解决这些不足中的一部分。除了这两个技术，还有其他技术也在针对这些不足提出改进。

2、SPDY

SPDY的主要目的是减少50%以上的页面加载时间，但是不增加部署的复杂性，不影响客户端和服务端的Web应用，只需要浏览器和Web服务器支持SPDY。主要有以下几点：

2.1 多路复用，一个TCP连接上同时跑多个HTTP请求。请求可设定优先级。

2.2 去除不需要的HTTP头，压缩HTTP头，以减少需要的网络带宽。

2.3 使用了SSL作为传输协议提供数据安全。

2.4 对传输的数据使用gzip进行压缩

2.5 提供服务方发起通信，并向客户端推送数据的机制。

实质上，SPDY就是在不影响HTTP语义的情况下，替换HTTP底层传输的协议来加快页面加载时间。

SPDY的解决办法就是设计了一个会话层协议--帧协议，解决多路复用，优先级等问题，然后在其上实现了HTTP的语义。

3、WebSocket

WebSocket则提供使用一个TCP连接进行双向通讯的机制，包括网络协议和API，以取代网页和服务器采用HTTP轮询进行双向通讯的机制。

本质上来说，WebSocket是不限于HTTP协议的，但是由于现存大量的HTTP基础设施，代理，过滤，身份认证等等，WebSocket借用HTTP和HTTPS的端口。

由于使用HTTP的端口，因此TCP连接建立后的握手消息是基于HTTP的，由服务器判断这是一个HTTP协议，还是WebSocket协议。 WebSocket连接除了建立和关闭时的握手，数据传输和HTTP没一丁点关系。

4、SPDY和WebSocket的关系

SPDY和WebSocket的关系比较复杂。

        补充关系，二者侧重点不同。SPDY更侧重于给Web页面的加载提速，而WebSocket更强调为Web应用提供一种双向的通讯机制以及API。

        竞争关系，二者解决的问题有交集，比如在服务器推送上SPDY和WebSocket都提供了方案。

        承载关系，试想，如果SPDY的标准化早于WebSocket，WebSocket完全可以侧重于API，利用SPDY的帧机制和多路复用机制实现该API。 Google提出草案，说WebSocket可以跑在SPDY之上。WebSocket的连接建立在SPDY的流之上，将WebSocket的帧映射到SPDY的帧上。

        融合关系，如微软在HTTP Speed+Mobility中所做的。

**83、 为什么每台电脑都要设置子网掩码？**

答案：在正规网络环境中，设备，如电脑等，都是连接在交换机上的，交换机再往上接到路由器上。子网掩码用来判断是否在一个网段。

如果掩码位相同，则判定为同一局域网，则直接将数据包发给交换机，通过MAC地址表转发就可以通信了。如果掩码位不同，则判定不在同一局域网，需要路由出去，所以就需要把数据包发给网关，由 路由器根据路由表转发出去，进行外网路由通信。

所以每台设备上都要配子网掩码，这样设备通信的时候才知道这个数据包要发给路由器还是直接交换机就能处理。

同一个局域网内掩码都是一样的，只有一样才是同一个局域网。这是协议机制。

下面遵循OSI模型，举个例子：

设A：10.1.1.10 /24

B：10.1.1.20 /24

C：50.1.1.80 /24

AB在同一局域网，C位于外网。

三个表：

ARP表：主机维护，存放IP地址和MAC地址对应关系。

MAC地址表：交换机维护，存放MAC地址和交换机端口对应关系。

路由表：路由器维护，存放IP地址和路由器端口对应关系。

首先AB通信，例如A要给B发送一个数据包，目前A知道B的IP地址，根据掩码规则判定B和自己在同一个局域网，同一个广播域。接下来A通过广播方式获取B的MAC地址，添加到自己的ARP表中。然后把要发送的包封装，然后发送给交换机，交换机收到数据包后解封装得到B的MAC地址，根据MAC地址表转发到B所连接的交换机端口，完成发送。

如果A要和C通信，发送一个包给C的话，也只知道C的IP地址，然后A根据掩码规则发现C和自己不是同一个局域网的，广播不到C，所以A只能把数据包发给网关，由网关发出去给到C。A同样通过广播方式获取网关的MAC地址，然后把C的IP地址和网关的MAC地址封装到数据包后发给交换机，交换机解封装后对比MAC地址表，发现是发给网关的包，就转发到网关即路由器所在的交换机端口。路由器收到包之后再解封装，得到C的IP地址，然后根据自己的路由表转发到相应的端口。完成通信。

所以如果计算机上不设置子网掩码，从第一步就不能完成，下面就更不能继续了。

如果同一个广播域里有机器设置不同的子网掩码，依然能够通信，只不过有的内网包需要到网关绕一圈。外网包的话只要网关设置对了就没问题。

**84、 为什么以太网无法接收大于1500字节的数据包？**

答案：综合考虑信道利用率和信道状态的折中方法，因为帧长太小信道利用率低，而帧长太大，出错重传效率会降低，具体地：

1、数据包越大，出错的可能性越大；

2、数据包一旦出错就需要重传，大的数据包重传花费的时间多，从而导致网速的下降；

3、时延增大，数据包只有在被完整收到的时候才会开始做校验检查，确认收到的每个bit都没有出错，如果frame包太大，将花费较长时间等待数据包进行校验，从而不能将包及时交给上层；

4、所有用户共享一个公共节点的资源，每个用户的每个资源切片一定要有所限制，不然其他人就只能干等着。这个切片规定多大要考虑效率问题 。

5、最初以太网是在共享介质上面而且是半双工的所有机器共享10Mbps的带宽，如果发送的帧太长，就会占用太久的网络，不仅会影响其他机器的通信，而且会让更多的机器等待到这一帧发送结束之后一齐发送，造成更多的冲突。现在其实已经没有这个限制了，许多交换机都支持更大的MTU，不过单纯提高MTU也不能提高性能。

6、交换机的成本和造价。单个数据块越大，交换机就需要越大的缓存。为什么这么说呢？交换机的设计，并不是像应用程序一样，数据包多大就申请多大的缓存。他一般是硬件加速的方法，为每个数据块分配固定长度的缓存。不论你发送的数据块多大，100或1500，交换机都按1500去为每个数据包分配相同大小的缓存。这样的实现有3个好处：一个是快，相同的操作执行速度高；二是内存无碎片，不规则的内存申请释放，最后会留下很多内存碎片，不需要复杂的内存管理；三是防止错误和攻击。如果根据数据内容的“长度”去申请内存，如果长度域出错怎么办，如果有人故意把长度域设置非常大攻击怎么办，如果有人故意把长度域设置小但是数据块却故意设置大怎么办。

**85、 OSI七层模型中，每一层的数据包都是谁生成和解包的？**

答案：应用层由具体的额服务进程来生成和解析数据包，例如http应该是类似apache、tomcat、nginx等web服务器进程负责；

tcp/ip应该是有操作系统的tcp/ip协议栈实现的，这操作系统的一部分；

链路层是由网卡驱动+网卡实现；

物理层由网卡+网线实现；

**86、为什么网关和主机可以不在一个网段？**

答案：要深入理解这个问题，首先要搞清楚网络通讯的原理，网络上通讯工作在物理层和数据链路层，源地址和目标地址是通过源和目的的mac地址进行通讯的。

当源主机访问目标主机时，首先看两者的IP在不在同一网段，结果是：

1、两者在同一网段，就会直接把包发向目标IP，这时要做：

1.1 查本地arp缓存，看看是否有IP和Mac的对应表.

1.1.1 有，直接向网络上发包，包中包括原mac及目标mac。

1.1.2 没有，则向网络发arp广播，用来查找与目标IP对应的mac地址（ARP发送的是广播数据，电缆上的每个以太网接口都要接收广播的数据帧）。

1.1.2.1 如果查到了，则向网络发包。

1.1.2.2 没查到，则不通讯。

2、两者不在同一网段，则把目标地址转为网关地址（也就是平时说的向网关发包），然后查找本地arp缓存，继续1.1 。

由此可以看出，源主机和网关的通讯过程中，并不会检查两者是不是同一网段，而是直接去查arp缓存或者发送ARP广播。所以是可能通讯的。

**87、NAT和DHCP 的区别是什么**

答案：DHCP 用于为网络内的主机分配 IP 地址和一些其他与网络配置有关的信息；NAT 用于在网络间翻译地址，以达到同一个网络内的多个主机（对应多个地址）通过同一个网关（一个地址）来访问外部网络

简单来说：

DHCP是维护私网（局域网）秩序，而NAT是两个网之间（私网到私网、私网到公网）联系的桥梁，这是区别；

DHCP维护的IP可以通过NAT访问不在本网的服务器或其他客户端，这是联系。

**88、OSI模型中，一个协议应该属于哪一层是以什么为标准划分的**

答案：一个协议归属于OSI参考模型哪一层，主要根据该层可以提供什么样的服务。这个服务如果拘泥于一条链路，则为数据链路层；如果服务可以让终端的流量可以跨越路由器的不同接口在互联网穿梭，则为网络层。如果可以提供或可靠、或不可靠端对端服务的，则为传输层。可以给用户提供服务的则为应用层。

**89、TCP三次握手过程中，seq的变化一直都是加1的吗？**

举个例子

1、-> syn  
TCP: Sequence number = x  
TCP: Acknowledgement number = 0  
2、<- ack syn  
TCP: Sequence number = y  
TCP: Acknowledgement number = x+1  
3、-> ack  
TCP: Sequence number = x+1  
TCP: Acknowledgement number = y+1

第一次握手阶段，客户端（假设客户端主动发起连接）给自己随机初始化一个序列号x，服务端收到之后，也要进行第二次握手，发回SYN+ACK，其中的序列号y也是随机分配的，同时对客户端SYN确认，因此确认号为x+1，表明序号为x以及之前的字节都已经收到。

第三次握手时候，客户端返回一个ACK，确认号为y+1，表明收到了服务端的SYN+ACK，同时自己的序列号更新为x+1，表明接下来的数据，从这个序列号开始。

因此，从整体来看，seq值并不是一直+1的。但是但从客户端角度（主动连接一方），seq值是+1的。

**90、使用Linux epoll模型，水平触发模式下，当socket可写时，会不停的触发可写事件，如何处理？**

**第一种最普遍的方式：**  
需要向socket写数据的时候才把socket加入epoll，等待可写事件。接受到可写事件后，调用write或者send发送数据。当所有数据都写完后，把socket移出epoll。

这种方式的缺点是，即使发送很少的数据，也要把socket加入epoll，写完后在移出epoll，有一定操作代价。

**一种改进的方式：**  
开始不把socket加入epoll，需要向socket写数据的时候，直接调用write或者send发送数据。如果返回EAGAIN，把socket加入epoll，在epoll的驱动下写数据，全部数据发送完毕后，再移出epoll。

这种方式的优点是：数据不多的时候可以避免epoll的事件处理，提高效率