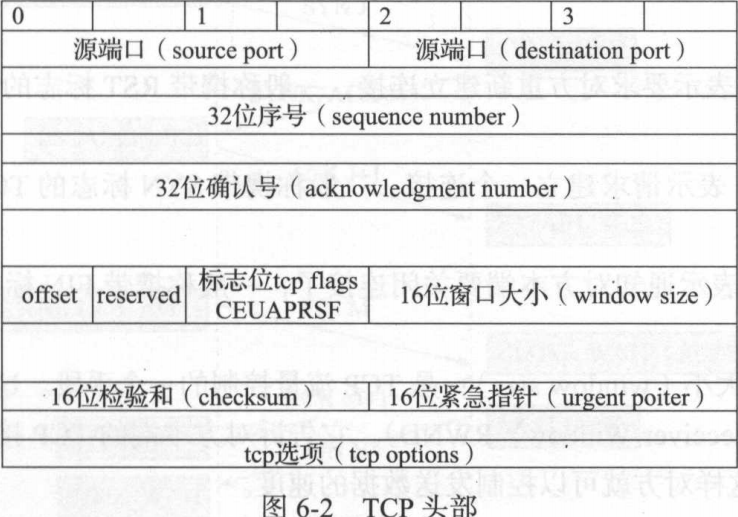
###### tcp头部：



16位源端口号(从哪里来) 16位目的端口号

32位序号：一次TCP通信(从TCP连接建立到断开)过程中某一个传输方向上的字节流的每个字节的编号。用来解决网络包乱序问题

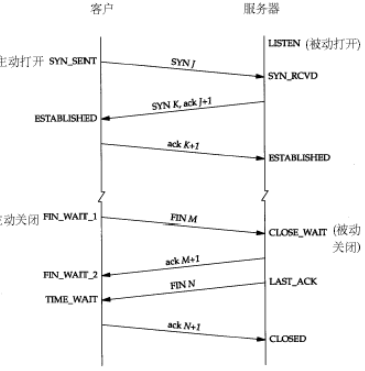
32位确认号: 对另一方发送来的TCP报文段序号值编号。用来解决网络包乱序问题。

6位标志位：ACK标志(确认号是否有效;用来解决不丢包的问题) ，SYN标志(表示请求建立一个连接) FIN标志(通知对方自己要关闭连接)

16位窗口大小:TCP流量控制手段。告诉对方本端的TCP接收缓冲区还能容纳多少字节的数据，让对方控制发送数据的速度

服务器常用的三个状态:established(正在通信), close\_wait(收到客户端的FIN报文，被动关闭)，last\_ack(服务器发送一个FIN报文，等待客户端的最后一个确认)

客户端常用的状态:established(正在通信);FIN\_WAIT\_1(主动关闭)，FIN\_WAIT\_2(客户端收到服务器对自己FIN报文的确认后，等待server的关闭)，TIME\_WAIT(客户端收到服务器的FIN报文，给服务器发送一个ACK报文，确认序号为收到的序号+1，然后进入TIME\_WAIT状态，等待2MSL，然后关闭连接)



TCP有一个特别的概念叫做半关闭：在关闭连接时，必须关闭传和送两个方向上的连接

为什么TCP建立连接要3次握手,断开连接要4次挥手

通信的双方要互相通知对方自己的初始化序列号(sequence number)SYN,以确保收到的数据不会乱序(TCP用序号拼接数据)

对于4次挥手：因为发送方和接收方都需要FIN和ACK

TIME\_WAIT(又称作2MSL状态):表示收到了server的FIN报文，并向server发出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果在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，而无需经过FIN\_WAIT\_2状态

TIME\_WAIT作用(为什么要等待2MSL(最大报文生存时间)?):首先明确time\_wait时期干了什么？server等待client发送对自己FIN报文的确认(ack)

如果没有time\_wait,可以在最后一个ack还未到达server的时候，就建立一个新的连接。此时，如果server收到了这个ack，就乱套了。必须保证这个ack完全死掉后，才能建立新的连接。也就是说，time\_wait允许老的重复分节在网络中消逝。

保证当最后一个client发出的ack包丢失后，能在2MSL时间内收到server重传的FIN包(可靠的实现TCP连接的终止)server收不到最后一个ack包，将不能释放连接，占用资源

问题来了！！！！！！在2MSL时间内无法再次使用同一个端口来进行服务(端口被占用)。设置SO\_REUSEADDR(重用本地地址)

SO\_REUSEADDR这个套接字选项通知内核，如果端口忙，但TCP状态处于TIME\_WAIT状态,则可以重用端口；如果端口忙，而TCP状态位于其他状态,重用端口时依旧会得到一个错误信息，指出地址已经使用中。 检查端口:netstat –anpt | grep “80”

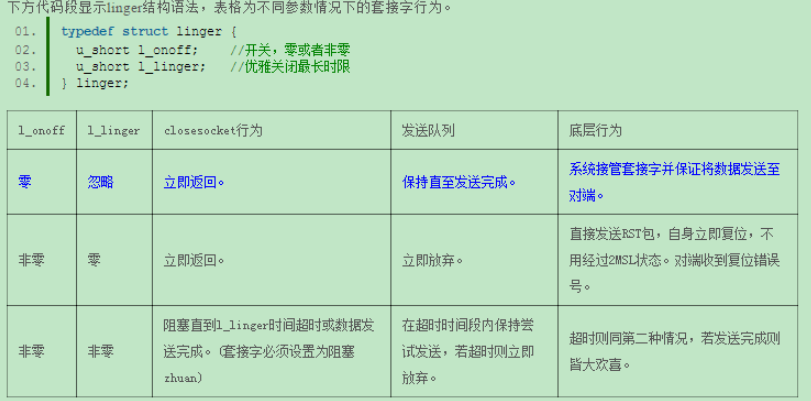
为什么server程序重启之后，无法连接，需要过一段时间才能连接上？

server一直停留在time\_wait状态(服务进程一直在为客户端服务，旧的连接还没释放)。需要等待2MSL时间才能重新连接上。

SO\_REUSEADDR允许一个server程序listen并且bind到一个端口，即使这个端口的连接仍存在。因此所有的TCP server都必须设定此选项，应对server重启现象。在bind前,setsockopt(sol\_socket,SO\_REUSEADDR….)

SO\_LINGER(延迟关闭):

Linger是延迟延缓的意思,这里的延缓是指面向连接的socket的close操作。默认close立即返回，但是当发送缓冲区中还有一部分数据的时候，系统将会尝试将数据发送给对端。SO\_LINGER可以改变close的行为



默认情况下，当调用close关闭socket的使用,close会立即返回。此时若send buffer还有数据，系统会先把send buffer中的数据发送出去，然后close才返回。正常的4次挥手。

linger开关打开,延迟时间设为0(或linger值设置太小，发送缓冲区来不及发送):调用close时,tcp连接会立即断开,send buffer中未被发送出去的数据被丢弃,并向对方发送rst信息，强制释放TCP连接。tcp不会进入time\_wait状态

linger开关打开，延迟时间非0：调用close(),若send buffer中还有数据尚未发送，该进程会休眠直到超时(超时后,send buffer中所有数据被丢弃)或send buffer中的所有数据都被发送并且得到对方tcp的应答消息(并不意味着对方程序已经收到数据，原因shutdown)

so\_linger的目的：强可靠性的需求。必须确保发出的消息，FIN都被对方收到

TCP异常终止(reset报文：rst报文)：TCP正常释放连接是通过4次挥手来完成，但有些情况下，TCP交互过程中产生意外情况，导致TCP无法正常释放连接，如果不通过其他方式来释放TCP连接的话，这个TCP连接将会一直存在，占用系统的资源。通过TCP的reset报文(TCP 报头的标志字段)

TCP 连接关闭 close和shutdown的区别：

在多进程并发中,close套接字描述符引用计数记录着共享的进程个数。当父进程或某一子进程close套接字时，描述符引用计数减1。当引用计数大于0时，close调用不会引发tcp的4次挥手断连过程。shutdown会切断进程共享套接字的所有连接。不论这个套接字的引用计数是否为0，同时shutdown的第二个参数可以选择断连的方式(优雅的关闭:关闭读或写或两者);close()除了关闭连接外，还释放套接字fd。shutdown()只关闭连接，不会释放占用的fd,即使用了shutdown关闭了连接，也要调用close()来释放fd

int shutdown(int sockfd, int howto);函数行为依赖于howto的值。0:关闭连接的读 1:关闭连接的写 2:关闭连接的读和写

O\_NONBLOCK：在读取不到数据或是缓冲区已满会马上return，而不会阻塞等待。读取不到数据返回-1，设置errno为EAGIN

SO\_KEEPALIVE:用于保持连接；检测对方主机是否崩溃，避免(server)永远阻塞于TCP连接的输入

设置该选项后,如果2h内再次套接口的任一方向都没有数据交换，TCP就自动给对方(server给client )发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的TCP分节。1)对方接收一切正常：以期望的ACK相应，2h后，TCP将发出另一个探测分节；2)对方无任何相应，以后就每隔75s发送一个探测分节，11min后就放弃。3)对方已崩溃且重新启动，套接口被关闭。只有设置了SO\_KEEPALIVE套接口选项后才会发送保活探测消息。

SO\_SNDTIMEO和SO\_RCVTIMEO:分别设置socket的发送和接收超时时间；如果超时，就会断开连接。

setsockopt(fd,SOL\_SOCKET,SO\_SNDTIMEO,&stTimeValStruct,sizeof(stTimeValStruct));

setsockopt(fd,SOL\_SOCKET,SO\_RCVTIMEO,&stTimeValStruct,sizeof(stTimeValStruct));

SO\_RCVBUF和SO\_SNDBUF(改变默认缓冲区大小):TCP的可靠性在于容错能力强；容错强就要有备份，也就是说要有缓存，这样才能支持重传等功能

每个TCP套接字都有一个发送缓冲区和一个接收缓冲区，每个udp套接字都有一个接收缓冲区。

当进行send和recv操作时，立即返回(非阻塞)，其实数据并没有发送出去，而是存放在内核的send buffer和内核的receive buffer中

接收缓冲区

接收缓冲区把数据缓存入内核，应用进程一直没调用read进行读取的话，数据就一直缓存在相应socket的接收缓冲区内。read的工作就是把内核缓冲区的数据拷贝到应用层的buffer里面。

接收缓冲区buffer满之后的处理策略：

TCP：对于TCP，如果进程一直没有读取,buffer满了之后，通知对端TCP窗口关闭(滑动窗口的实现，TCP的可靠传输)

UDP:当套接口接收缓冲区满时,新来的数据无法进入接收缓冲区，此数据报就被丢弃(udp是没有流量控制的，快的发送这很容易就淹没慢的接受者，导致接收方的UDP丢包)

发送缓冲区：

进程调用send发送数据时,将数据拷贝到socket的内核发送缓冲区内,然后send便返回。即send返回时,数据不一定会发送到对端去，send仅仅是把应用层buffer的数据拷贝到socket的内核中。

UDP： 没有发送缓冲区：只要有数据就发，不管对方是否正确接收，所以不缓冲，不需要发送缓冲区

为了达到最大网络吞吐,SO\_SNDBUF不应该小于带宽\*延迟；设的太大，浪费内存

网络字节序与主机序 ntohl htonl

主机序：小端(低字节存储在内存的低地址); 大端(高字节存储在内存的低地址)

网络字节序:所有网络协议都是采用大端的方式来传输数据的。所以把大端方式称之为网络字节序。当两台采用不同字节序的主机通信时，在发送数据之前都必须经过字节序的转换成为网络字节序后再进行传输

TCP是个流协议，没有界限的一串数据。常说的”粘包(不知道消息之间的边界)”，就需要把接收到的数据进行拆包，拆成一个个独立的数据包；而为了拆包就必须在发送端进行封包。

粘包原因：Nagle算法造成发送端的粘包(发送端发送不及时)。接收端接收不及时造成接收端粘包(TCP把接收到的数据保存在自己的接收缓冲区中，应用层没能及时取出TCP的数据，造成TCP接收缓冲区存放了多段数据)。

解决粘包方法(封包+拆包) #pragma pack(1) 发送，接收结构体

封包:给一段数据加上包头(将数据包分成包头和包体head+body)，包头是个大小固定的结构体，其中有个变量表示包体的长度(very important).固定的包头长度以及包头中含有的包体长度的变量值能正确拆分出一个完整的数据包。

数据包之间添加特殊字符;将数据包封装成固定长度(不够的用0填充)

对于UDP来说不存在拆包问题，不使用块合并优化算法。因为UDP是个数据报协议，也就是两段数据间是有界限的，在接收端要么接收不到数据要么就是接收一段完整的数据，不能一次提取任意字节数据。

TCP\_NODELAY/TCP\_CORK Nagle算法(默认情况下，发送数据采用Nagle算法，虽然提高了网络吞吐率，实时性却降低了。)

TCP中的Nagle算法默认是开启的，但并不适用任何情况，对于telnet或rlogin这样的远程登入比较合适

Nagle算法：如果包长度达到MSS，则允许发送。如果包含有FIN，则允许发送；如果设置了TCP\_NODELAY，则允许发送；未设置TCP\_CORK选项时，若所有发送出去的小数据包(包长度小于MSS)均被确认，则允许发送。发生了超时，则立即发送。

TCP\_NODELAY：禁止Nagle算法，有数据发送就立即发送

TCP\_CORK：cork就是塞子的意思，形象的理解就是用cork将连接塞住，使得数据先发不出去，等到拔去塞子再发出去。设置该选项后，内核会尽力把小数据包拼接成一个大的数据包(一个MTU)再发送出去。当然若一定时间后，内核仍然没有组成一个MTU时也必须发送现有的数据(不能不能让数据一直等)。

是一种加强的Nagle算法，过程和Nagle算法类似，都是累计数据后发送；区别在于Nagle算法中发送小包后若收到ack，则可以继续发送(Nagle关心网络拥塞问题)；而设置TCP\_CORK后，即使所有的ack都已经收到，但我还是不想发送数据(CORK关心发送内容)，想继续等待应用层更多的数据

CLOSE\_WAIT：表示等待服务器关闭自己一端的连接。当对方关闭一个socket后发送FIN报文给自己时，系统毫无疑问回应一个ACK报文给对方，此时进入CLOSE\_WAIT状态。等待自己去关闭连接。

TCP主要两个特性：可靠性(超时重传，回复确认，序号)，流控特性(拥塞机制，流量控制)

TCP滑动窗口(进行流量控制)：TCP的窗口是一个16bit位字段(TCP报文头中字段)，代表窗口的字节容量。TCP是全双工协议，会话双方都可以同时接收，同时发送数据。各自维护一个发送窗口，接收窗口。用于网络传输时流量控制，以避免拥塞的发生。该协议允许发送方一次发送多个数据分组，不必每次发一个分组就停下来等待确认，因此该协议可以加速数据的传输，提高网络吞吐量。Tcp利用滑动窗口告诉发送端所能发送的数据的缓冲区大小(滑动窗口本质上描述接收方的TCP数据缓冲区大小的数据，发送方根据这个数据来计算自己最多能发送多长的数据)

tcp发送方：发送缓冲区内数据分为:已发送并且已得到ack；已发送还未得到ack；未发送且允许发送；未发送且不允许发送。其中已发送还未收到ack和未发送允许发送的数据称为发送窗口。

tcp接收方：接收缓冲区数据分为：已接收；未接收准备接收；未接收且未准备接收。其中未接收且准备接收称为接收窗口

可靠性来自于”确认和重传”机制：发送窗口只有收到对发送的确认(ack)后，窗口才移动。接收窗口只有前面的段都确认后才会移动，前面字节未收到但收到后面字节，窗口不移动，并且不对后续字节确认。

窗口大小：指无需等待应答而可以继续发送数据的最大值。窗口中的数据即使没有收到确认应答也可以被发送出去。

滑动窗口以外的部分包括未发送的数据以及已经确认对端已收到的数据。

如果服务器出了异常,80%都是下面这种情况：

1. 服务器保持了大量time\_wait状态 2.服务器保持了大量close\_wait状态

服务器保持了大量time\_wait状态

首先time\_wait是主动关闭连接的一方保持的状态，对于爬虫服务器来说，在完成一个爬虫任务之后,它就发起主动关闭连接,从而进入time\_wait的状态,然后在保持这个状态2MSL(max segment lifetime)时间之后，彻底收回资源。为什么这么做?明明已经主动关闭连接为啥还要保持资源一段时间？这个是Tcp/IP的设计

Time\_wait状态如何产生？

首先调用close()发起主动关闭的一方，在发送最后一个ack之后会进入time\_wait的状态，发送方保持2MSL时间之后才会回到初始状态。MSL是数据包在网络中的最大生存时间，在这个时间里面这个连接的四元组(客户端IP和PORT，服务端IP和PORT)不能被使用

close\_wait

什么情况下，连接处于close\_wait状态？

在被动关闭连接状态下，在已经接收到FIN，但是还没有发送自己的FIN的时候，连接处于close\_wait状态。通常讲，close\_wait状态的持续时间很短，但是一些情况会出现连接长时间处于close\_wait状态

出现大量close\_wait的现象，主要原因是某种情况下对方关闭了socket连接,但是我方忙于读或者写，没有关闭连接。代码需要判断socket，一旦读到0，断开连接，read返回负，检查一下errno如果不是AGAIN,就断开连接

解决close\_wait方法：

1. 代码需要判断socket,一旦read返回0，断开连接，read返回负,检查errno，如果不是eagain，断开连接
2. 使用一个heart-beat线程，定期想socket发送指定格式的心跳数据包。如果接收到对方的RST报文，说明对方已经关闭了socket,那么我们也关闭这个socket
3. 设置so\_keepalive选项，并修改内核

tcp/udp

为什么不能两次握手进行链接？

client给server发送一个链接请求(SYN=1),server收到了这个分组，并发送了确认应答分组。按照2次握手协定,server认为链接已经成功建立，开始发送数据分组。可是server给client的应答分组丢失,client不清楚是否server准备好，认为server还未建立链接成功(server到client这段不成功)，将忽略server发来的任何数据分组，只等待连接确认应答分组。而server在发出的分组超时后,重复发送同样的分组，形成了死锁。

如果已经建立了链接，但是客户端突然出现故障怎么办？

Tcp设有一个保活计时器。如果client挂了，server不会一直等下去，白白浪费资源；若一段时间没收到client的任何数据，发送一个探测报文仍没反应，就认为client挂了，接着关闭链接

TCP和UDP的区别： tcp适合可靠传输但速度慢(文件传输，重要状态更新)，udp不可靠但速度快(视频传输，实时通信)

tcp面向连接，可靠(不重复，不丢失，无差错，按序到达)，基于字节流；而udp不面向连接，不可靠，基于数据包

tcp是点对点的两点间服务，即一条TCP连接只有两个端点；udp支持一对一，一对多，多对一，多对多的交互通信。

tcp需要三次握手，重新确认等连接过程,实时性差，过程复杂；而udp无连接，因而实时性较强

tcp有拥塞控制和流量控制保证数据传输的安全性；udp没有拥塞控制，网络拥塞不会影响源主机的发送效率。

报文长度: tcp是动态报文长度，即tcp报文长度是根据接收方的窗口大小和当前网络拥塞情况决定的；udp面向报文，不合并，不拆分，保留上面传下来的报文边界。

首部开销：tcp报头20字节,udp首部8字节；tcp无丢包,udp有丢包

使用TCP协议：FTP（读写20端口，控制21端口）TELNET (23端口) http(80端口)

使用udp协议：DNS （53端口） DHCP（67端口）

路由协议（RIP ，OSPF）

内部网关协议(IGP)是在自治系统内部运行的路由协议，主要包括:RIP,OSPF

外部网关协议(EGP)是在自治系统之间使用的路由协议,有BGP

内部网关协议划分为两类：1）距离矢量路由协议(RIP) 2)链路状态路由协议(OSPF)

tcp保证可靠性：

序列号，确认应答，超时重传(TCP每发送一个报文段，就对这个报文段设置一次计时器。只要计时器设置的重传时间到了，但没收到确认，就要重传这个报文段)

数据到达接收方，接收方需要发出一个确认应答，表示已经收到该数据段，并且确认序号会说明它下一次需要接收的数据序列号。如果发送方迟迟未收到确认应答，那么可能是发送的数据丢失，也可能是确认应答丢失，这时发送方在在等待一段时间超时重传。这个时间一般是2\*RTT(报文往返时间)

tcp流控特性:窗口控制与快速重传

TCP会利用窗口控制来提高传输速度，意思是在一个窗口大小内，不用一定要等到应答才能发送下一段数据，窗口大小就是无需等待确认而可以继续发送数据的最大值。如果不使用窗口控制，每一个没收到确认应答的数据都要重发

使用窗口控制，如果数据段1001-2000丢失，后面数据每次传输，确认应答都会不停的发送序列号为1001的应答，发送端如果收到3次相同应答，就会立刻进行重发

拥塞控制：

为了在发送端调节所要发送的数据量,定义了一个拥塞窗口，在发送数据时，将拥塞窗口的大小与接收端确认窗口大小做比较，取较小者作为发送数据量的上限

拥塞避免: 设置慢启动阈值，一般开始都设为65536.拥塞避免是指当拥塞窗口大小达到这个阈值，拥塞窗口的值不再指数上升(慢启动)，而是加法增加(拥塞窗口大小+1)，以此来避免拥塞。当出现网络拥塞(三个重复的ack或者超时)时候,将慢启动阈值设置为出现拥塞时大小的一半，cwnd的大小重新从0开始进入慢起动阶段。

快重传：

在遇到3次重复确认应答时(3个ack)，代表收到了3个报文段，但是这之前的1个段丢失了，便对它进行立即重传(而不是等待计时器时间到了再重传)

快恢复

在遇到连续3次重复确认应答，将慢开始门限减半，将当前的拥塞窗口设置为慢开始门限，并且执行拥塞避免算法(连续收到三个重复确认，说明当前网络可能没有拥塞)

什么情况开始减慢增长的速度？

一旦拥塞窗口>慢开始门限，就采用拥塞避免算法，减慢增长速度。一旦出现丢包的情况，就重新进行慢开始，减慢增长速度。一旦发送方接收到3个重复确认，就采用拥塞避免算法，减慢增长速度

IP地址作用/MAC地址作用

MAC地址是一个硬件地址，用来定义网络设备的位置，由数据链路层负责。而IP地址为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。

TCP/IP数据链路层的交互过程

网络层到数据链路层用mac地址作为通信目标。数据包到达网络层准备往数据链路层发送时，首先会去自己的ARP缓存表(存放IP-MAC对应关系)去查找目标IP的MAC地址。如果查到了，就将目标IP的MAC地址封装到链路层数据包的包头。如果缓存中没找到，发起广播(发起ARP广播报文who has 192.168.0.8 tell 192.168.0.5)，所有收到广播的机器看这个IP是不是自己的，如果是自己的，以单播的形式将自己的MAC地址回复给请求的机器(192.16回复)

udp的connect函数:

http(默认端口80,HTTPS默认端口443):

一次HTTP操作称为一个事务，工作过程:1)client和server建立连接2）建立连接后,client发送一个请求给server，请求的格式为:URI,http1.1,MIME信息(包括请求修饰符，客户机信息和可能的内容) 3)server接到请求后，给予响应信息，其格式为一个状态行：协议版本号，一个成功或错误的代码，后边是MIME信息(包括服务器信息，实体信息和可能的内容) 4)客户端收到服务器所返回的信息显示在浏览器上，然后与服务器断开连接。

HTTP协议结构：

http协议无论是请求报文还是回应报文，都分为4个部分：1)报文头():GET,POST,….URI, HTTP/1.1

1. 请求头(header line):0个或多个3)空行(作为header lines的结束) 4)可选的消息体(body)

HTTP常见请求头:

Host(发送请求时,该请求头是必需的)主要用于被请求资源的Internet主机和端口号，通常从HTTP URL中提取出来。HTTP/1.1请求必需包含主机头域，否则系统以400状态码返回。Connection：值通常只有2个,keep-alive(HTTP/1.1默认功能)和close(本次请求后关闭连接)。Keep-alive被用来解决效率低的问题。Keep-alive使client到server的连接持续有效，当出现对server的后继请求时,keep-alive避免了建立或者重新建立连接。这个功能很有用，但同样影响了性能(在处理暂停期间，本来可以释放的资源仍旧被占用)。

ACCEPT:浏览器可以接收的MIME类型(Accept: text/html表示浏览器可以接收服务器发回的类型为text/html,即html文档)

Range:可以请求实体的范围 响应状态码:206

HTTP回应报文：

返回码:2XX:成功 3XX：重定向(要完成请求必须更进一步的操作) 4XX：客户端错误(请求有语法错误或请求无法实现) 400:bad request(语法错误，服务器不能理解) 403:网站拒接访问 401:unauthorized 发送的请求需要有通过http认证的认证信息；404:not found 501：服务端无法识别请求的方法(不支持当前请求需要的功能) 503：service unavailable(服务器维护或过载导致压力过大，无法处理请求)

HTTPS：SSL+HTTP 加密传输；身份认证

安全版本的HTTP，在http下加入SSL层，因此加密的详细内容就需要SSL。HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

HTTP和HTTPS的区别：

1)HTTP的信息是明文传输，HTTPS则是具有安全性的加密传输协议;2)HTTP的端口是80，HTTPS的端口是443；3)HTTPS比较耗性能，对于安全没那么高要求的应用来说，用HTTP就已经够了。

HTTP的工作原理：一次HTTP操作称为一个事务，其工作过程分为4步：

Client和Server建立连接；连接建立后，客户发送一个请求给服务端，请求格式为：URL,协议版本号,body等。服务端接到请求后，给予响应的响应信息，格式为状态行，包括协议的版本号，代码，body等内容；客户收到信息后，断开连接。

HTTPS的工作原理：

客户使用HTTPS的URL访问服务器，要求与服务器建立SSL连接；服务器收到客户端的请求后，会将网站的证书信息传送一份给客户端；客户端与服务器开始协商SSL连接的安全等级；客户端根据双方同意的安全等级，建立会话密钥，然后利用服务器的公钥将会话密钥加密，并传送给服务器；服务器利用自己的私钥解密出会话密钥；服务器利用会话密钥加密与客户端的通信。

http的基本优化(延迟):

浏览器阻塞:浏览器会因为一些原因阻塞请求：超过浏览器最大连接数限制,后续请求就会被阻塞。

DNS查询: 浏览器需要知道目标服务器的IP才能建立连接。将域名解析为IP的这个系统就是DNS。通常可以利用DNS缓存结果达到减少这个时间的目的

建立连接:http是基于tcp协议的,浏览器最快也要三次握手达到真正的建立连接，但这些连接无法复用，每次请求都要经历三次握手和慢起动

http1.0和http1.1的区别

http1.1支持长连接和请求的流水线处理；http1.0规定浏览器与服务器只保持短暂的连接,浏览器的每次请求都需要与服务器建立一个tcp连接，服务器完成请求处理后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求

http1.1则支持持久连接，并且默认使用持久连接。在同一个tcp连接中可以传送多个http请求和相应。多个请求和相应可以重叠，多个请求和响应可以同时进行。更加多的请求头和响应头(比如http1.0没有host字段)

在1.0时的会话方式：1.建立连接2.发出请求信息3.回送相应信息4.关掉连接

http1.1的持续连接,也需要增加新的请求头来帮助实现，例如,connection请求头的值为keep-alive时，客户端通知服务器返回本次请求头结果后保持连接；connection请求头的值为close时,客户端通知服务器返回本次请求结果后关闭连接.http1.1还提供了与身份认证,状态管理和cache缓存等机制相关的请求头和响应头。

请求的流水线：在一个tcp连接上可以传送多个http请求和响应，减少了建立和关闭连接的消耗和延迟。例如：一个包含许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接

http1.1增加host字段

在http1.0中认为每台服务器都绑定一个唯一的IP地址。因此，请求消息中的URL并没有传递主机名(hostname)。随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机，并且它们共享一个IP地址。http1.1的请求消息和响应消息都支持host头域，且消息中如果没有host头域会报告一个错误(400 bad request)

响应码100(continue)

http1.1加入了一个新的状态码100(continue).客户端先发送一个只带头域的请求，如果服务器因为权限拒绝了请求，就回送响应码401（unauthorized）；如果服务器接收此请求就回送响应码100，客户端就可以继续发送带实体的完整请求了。100(continue)状态码的使用，允许客户端在发送request消息body之前先用request header试探一下server看server要不要接收request body再决定要不要发request body

chunked transfer-coding

http1.1引入了chunked transfer-coding来解决上面这个问题，发送方将消息分割成若干个任意大小的数据块，每个数据块在发送时都会附上块的长度，最后用一个零长度的块作为消息结束的标志。这种方法允许发送方只缓冲消息的一个片段，避免缓冲整个消息带来的过载

GET和POST的区别：本质都是TCP链接；GET参数通过URL传递,POST放在request body中；GET请求在URL中传递的参数是有长度限制的，而POST没有；GET请求只能进行URL编码，而POST支持多种编码方式；GET请求参数会被完整保留在浏览历史纪录里(浏览器主动cache),而POST中的参数不会被保留。

概括:对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200。

而对于POST，浏览器先发送header,服务器响应100 continue(http1.1新增的状态码),浏览器再发送data,服务器响应200

请求方法：指定了client想对指定的资源/server做何种操作

GET：获取资源；GET方法用来请求已被URI识别的资源。指定的资源经服务器端程序解析后返回响应的内容

POST：传输数据给服务器处理，数据包含在http的body中；post请求会向指定资源提交数据，请求服务器进行处理，如:表单数据提交，文件上传，请求数据会被包含在请求体中。Post方法可能会创建新的资源/或修改现有资源

HEAD：获得首部(用于确认URI的有效性及资源更新的日期时间等)：判断类型；看对象是否存在；测试资源是否被修改过。 HEAD和GET区别：GET方法有body，HEAD方法无body

DELETE删除文件：用于请求服务器给出所请求的URI

PUT: 传输文件

Put方法用来传输文件，就像FTP协议的文件上传一样；要求在请求报文主体中包含文件内容，然后保存在请求URI指定的位置；但存在安全问题，一般不用

浏览器中输入一个URL发生什么，用到哪些协议？

输入URL，首先浏览器要将URL解析为IP地址，解析域名用到DNS协议。首先主机会查询DNS缓存，如果没有就给本地DNS发送查询请求。DNS服务器是基于UDP协议的，因此用到UDP协议。

得到IP地址后，浏览器向服务器发送HTTP的GET请求(或者https)。该报文传给TCP层处理，TCP层可能将HTTP数据包分片，分片后传给IP层，用到IP协议。IP层通过路由选路，发送到目的地址。以太网需要目的IP的地址，需要用到ARP协议。

从浏览器输入地址到呈现页面中间发生了什么事情

客户端进行DNS(udp协议)解析；(DNS协议)

解析域名得到ip地址和arp地址，找到目标服务器(IP协议， ARP协议)

进行tcp三次握手，建立tcp连接。(TCP三次握手)

浏览器发送数据，等待服务器响应

服务器处理请求，并对请求做出响应。(HTTP请求响应)

浏览器收到服务器响应，得到html代码(浏览器渲染) (TCP四次挥手)

TCP如何实现可靠传输？序列号，超时重传，确认应答

序号：TCP给每个发送的包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。

校验和：TCP首部校验和，检测数据在传输过程中的任何变化。校验和若有错，丢弃这个报文，不做确认此报文。

流量控制(利用滑动窗口实现)：接收方发送的确认报文中的窗口字段用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为0，发送方不能发送数据。

拥塞控制：当网络拥塞时，减少数据的发送。慢开始，拥塞避免，快重传，快恢复。

停止等待协议：基本原理是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。若收到重复分组，就丢弃该分组，但同时还要发送确认。

超时重传：当TCP发出一个报文后，它启动一个定时器，等待对端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

自动重传请求ARQ协议：停止等待协议超时重传被称为自动重传ARQ

连续ARQ协议：提高信道利用率。发送方维持一个发送窗口(滑动窗口)，位于发送窗口的分组可以连续发送，不必等待对方确认。接收方采用累计确认，对按序到达的最后一个分组发送确认。

udp不提供控制机制，利用ip提供面向无连接的通信服务。将应用程序发送的数据包，立刻原样发送(不用Nagle算法)；即使网络拥堵的情况下，udp也无法进行流量控制等避免网络阻塞的行为。此外传输过程中,udp不负责重发，数据包乱序也不能纠正顺序的功能。需要有这些细节控制的话，需要在应用层做出处理。

如何设计udp的可靠传输？ Tcp速度慢(Nagle,超时重传，流量控制等严重影响了速度和实时性)

传输层无法保证数据的可靠传输，只能通过应用层来实现。实现不在传输层，实现转移到了应用层。实现功能：发送方：包的分片，包确认，包的重发 接收方：包的调序，包的序号确认。

rudp(可靠udp:reliable udp):实现udp可靠传输

由于udp是一种无连接，不可靠，基于数据报的传输层协议，提供面向事务的简单不可靠信息传送服务。所以udp的可靠性由上层应用实现，所以要实现udp可靠性传输，必须通过应用层来实现和控制。

设计方法：添加seq/ack机制，确保数据发送到对端。添加发送和接收缓冲区，主要是用户超时重传。添加超时重传机制。

Linux

EWOULDBLOCK:非阻塞socket读写未就绪时，调用读写函数将返回EWOULDBLOCK或EAGAIN

EWOULDBLOCK(windows)等价于EAGAIN(linux)

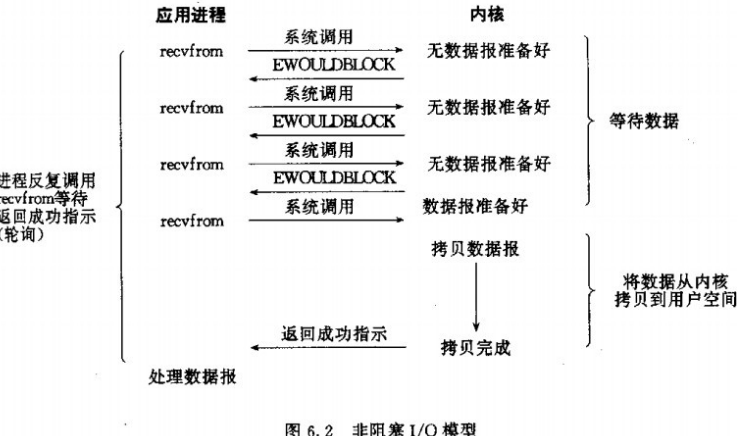
EINTR

非阻塞socket+epoll(select)：既然epoll(select)能够发现socket可读或可写，为什么socket要设置为非阻塞？

Epoll(select)能够说明socket可读或可写，不能说明能读写多少字节：socket写缓冲区有10字节空间，但如果要写100字节，若没设置非阻塞，线程就阻塞了。

5种IO模型

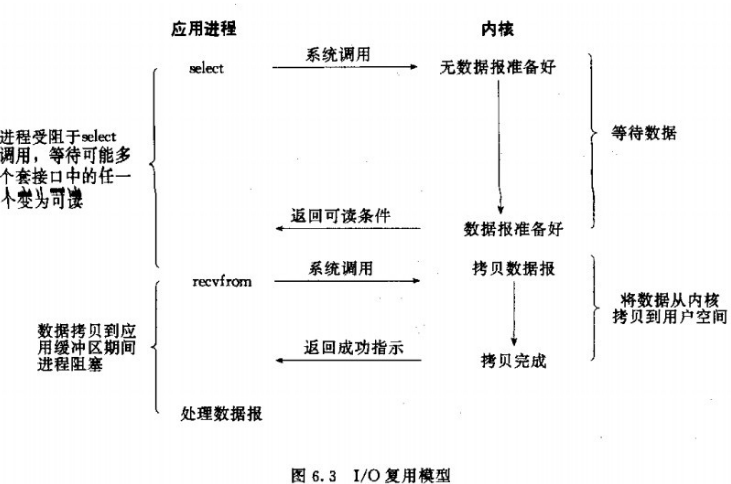
阻塞IO：应用程序调用一个IO 函数(recv() recvfrom())，导致应用程序阻塞，等待数据准备好；数据准备好了，从内核拷贝到用户空间，IO 函数返回成功指示

非阻塞IO：非阻塞IO通过进程反复调用IO 函数(多次系统调用，并马上返回EWOULDBLOCK)； 

IO复用：复用指单个线程或进程同时管理多个socket。

主要是select和epoll;能实现同时对多个IO端口进行监听；IO复用模型会用到select,poll,epoll函数，这几个函数会使进程阻塞(直接调用select,epoll若没数据到会阻塞)，但是和阻塞IO所不同的，这两个函数可以同时阻塞多个IO操作。而且可以同时对多个读操作，多个写操作的IO函数进行检测，直到有数据可读或可写，才真正调用IO操作函数

信号驱动IO



select,poll,epoll

IO复用就是通过一种机制，一个进程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

与多进程和多线程技术相比,IO多路复用技术的最大优势是系统开销小,系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减少了系统的开销

select(apache常用)

基本原理:select函数监视的文件描述符分为3类，分别是writefds,readfds,exceptfds.调用select会阻塞，直到有描述符就绪(有数据可读，可写，或者有except)，或者超时，函数返回。

特点：最大并发数限制；因为一个进程所打开的FD是有限制的，因此select模型的最大并发数被相应的限制了。

效率问题：select每次调用都会从头至尾线性扫描全部的fd集合(无论socket是否活跃,都遍历一遍)，测试每个fd是否有事件发生。

内核/用户空间 内存拷贝问题：如何让内核把fd消息通知给用户空间的一个存放大量fd的数据结构？select采取了内存拷贝方法(将fd消息从内核态内存拷贝到用户态)，非常耗时。

poll

解决了select最大并发数的限制(socket通过链表连接，没有数量上的限制)，但依然没有解决掉select的效率问题和内存拷贝问题

epoll(nginx的使用)select的缺点就是epoll的优点

epoll没有最大并发数连接的限制(上限是整个系统最大可以打开的文件数目)

效率提升：只触发活跃的socket连接

内存共享：使用mmap共享内存，不用内存拷贝。

用法总结：

epoll API:

int epoll\_create(int size); 创建一个epoll句柄(epfd),用完epoll后必须调用close函数关闭

int epoll\_ctl(int epfd,int op,int fd,struct epoll\_event\* event);

op: EPOLL\_CTL\_ADD; EPOLL\_CTL\_MOD;EPOLL\_CTL\_DEL

event:告诉内核需要监听什么事件：EPOLLIN(读),EPOLLOUT(写)EPOLLET(设置为边缘触发)EPOLLONESHOT(只监听一次事件，当监听完这次事件后，如果还需要继续监听这个socket，需要再次加入epoll\_ctl\_add)

int epoll\_wait(int epfd, struct epoll\_event\* events,int maxevent,int timeout);

收集在epoll监控的事件中已经发生的事件。参数events是分配好的epoll\_event结构体数组，epoll将发生的事件赋值到events数组中。maxevents告诉内核这个events数组多大。如果调用成功，返回对应IO已准备好的文件描述符数目。返回0表示已经超时。

epoll\_wait的工作流程：

epoll\_wait调用时，当链表为空(无就绪fd)时挂起当前进程,直到链表不为空时进程才被唤醒。

文件状态改变(读写缓冲区由不可读变为可读，不可写变为可写)，导致相应fd上的回调函数被调用。

回调函数将相应的fd加入链表，导致链表不为空，进程被唤醒，epoll\_wait可以继续运行。

epoll高效原理：红黑树，链表，少量的内核cache(内核使用slab机制，为epoll提供了快速的数据结构)，mmap

执行epoll\_create时，创建红黑树(epoll采用红黑树去储存所有套接字)和就绪list链表

执行epoll\_ctl时，如果增加fd,则检查在红黑树中是否存在，存在则立即返回，不存在则添加到红黑树中，然后向内核注册回调函数，用于当事件来时往链表中插入数据。

执行epoll\_wait时立即返回链表里的数据即可。

epoll优点：

没有最大并发连接的限制，能打开的FD 的上限远大于1024

效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数（和连接总数无关），因此在实际网络环境中,epoll的效率远高于select和poll

内存拷贝,利用mmap ()文件映射内存加速和内核空间的消息传递(把fd消息通知给用户空间);即epoll使用mmap减少复制开销

epoll对文件描述符的操作: LT(level trigger默认模式)和ET(edge trigger)

LT模式:当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不处理该事件。下次调用epoll\_wait，会再次响应应用程序并通知此事件

ET模式:当epoll\_wait检测到描述符事件发生并将此事件通知应用程序,应用程序应该完整的处理该事件(在et模式下，应用程序若只读取了一部分数据，其将再也得不到通知了。正确做法是应用程序确认读完了所有的字节，一直调用read/write直到出错eagain为止)，因为下一次epoll\_wait将不再通知这一事件。应用程序若只写了一部分数据，即使写缓冲区仍有空间，也不能写(ET模式下不返回写就绪。所以要尽可能写完)。直到写缓冲区有数据出去了(可写空间变大了)，才能写。

LT模式：效率低于ET模式；只要有数据没被获取，内核就不断通知，因此不必担心事件丢失。

ET模式：效率高，容易发生丢失事件的情况。从本质上讲：与LT相比，ET是通过减少系统调用来提高并行效率的。如何处理ET模式下的数据读写？要用while循环来处理读写，直到没有数据可以读写，fd要设置为非阻塞。

ET模式下的accept问题：某一时刻，多个连接同时到达，服务器tcp就绪队列中瞬间积累多个就绪连接，由于是边缘触发模式，epoll只会通知一次，accept只处理一个连接，导致tcp就绪队列中剩下的连接都得不到处理。该如何处理？

在while循环中调用acept，处理完就绪队列后退出循环。当accept返回-1,且errno设置为eagain就表示所有连接处理完了。

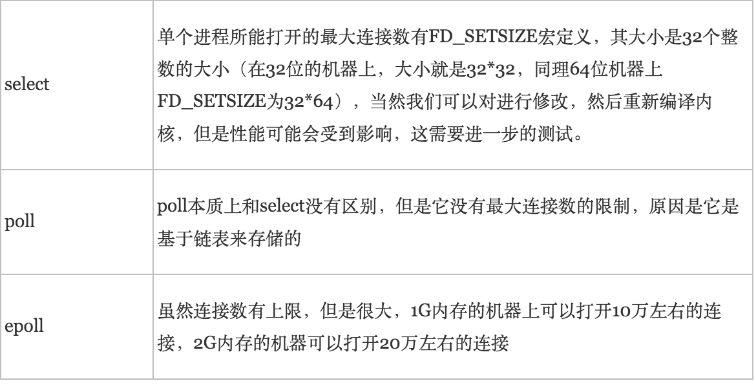
Et模式为什么设置为非阻塞模式下工作？

因为et模式下读写要一直读或写直到出错(eagain)，而如果fd不是非阻塞的，那么一直读或一直写必定会在最后一次阻塞，造成不能阻塞在epoll\_wait上了。

注意：如果没有大量的idle-connection或者dead-connection，epoll的效率并不会比select/poll高很多，但是当遇到大量的idle-connection，就会发现epoll的效率大大高于select/poll

select,poll,epoll区别

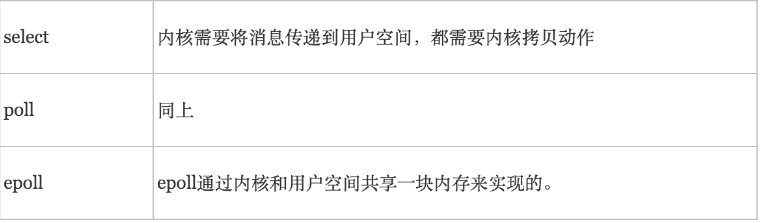
1. 支持一个进程所能打开的最大连接数



2.FD剧增后带来的IO效率问题



3.消息传递方式



http1.0和http1.1的区别

http1.0,每对request/response都使用一个新的连接；http1.1支持持久连接

host域 http1.1在request消息头里面多了一个host域；如GET /www/theproject.html http/1.1 Host:www.w3.org http1.0没有这个域

Transfer Coding:分块传输编码是一种超文本传输协议(HTTP)中的一种数据传输机制(持久连接)，允许HTTP由应用服务器发送给客户端应用的数据可以分成多个部分。通常，HTTP应答消息中发送的数据是整个发送的，content-length消息头字段表示数据的长度(数据长度很重要，因为客户端需要知道哪里是应答消息的结束，以及后续应答消息的开始)。然而使用分块传输编码，数据分解成一系列数据块，并且一个或多个块发送，这样服务器可以发送数据而不需要预先知道发送内容的总大小。数据块的大小可以不一致。

http1.1支持chunked transfer，所以可以有Transfer Encoding 头部域 http1.0没有

100(continue) Status

100(continue)状态码的使用，允许client在发request消息body之前先用request header试探一下server，看server要不要接收request body，再决定要不要发request body

Client在request头部中包含: expect:100-continue server看到偶如果回100(continue)则个状态码,client就继续发request body

http1.1增加了OPTIONS,PUT, DELETE..这些request方法

http1.1增加了status code: 100(continue) 206(partial content)

http2.0的三大特性：头部压缩；多路复用；服务器推送；

http2.0头部压缩：假设一个页面有100个资源需要加载，而每次请求都有1KB的消息头，则至少需要多消耗100kb来获取这些消息头。http2.0通过维护一个字典，差量更新http头部，大大降低因头部传输产生的流量。

http2.0多路复用：多路复用允许同时通过单一的连接发起多重请求-相应消息。

Http1.1 流水线解决方法为：若干个请求串行化单线程处理。后面的请求等待前面请求的返回才能获得执行机会一旦有某个请求超时，后续请求只能被阻塞。

http2.0多个请求可同时在一个连接上并行执行。某个请求任务耗时严重，不会影响到其他连接的正常执行。

服务器推送：推送把客户端所需要的资源伴随着index.html一起发送到客户端，省去了客户端重复请求的步骤

一些问题：

tcp三次握手的过程，accept发生在三次握手的哪一个阶段？

accept过程发生在三次握手之后，三次握手完成后，客户端和服务端就建立了tcp连接并可以进行数据交互了。这时可以调用accept函数获得此连接。accept函数会从已经建立连接的队列中取出第一个连接，并创建一个新的socket,新的socket的类型和地址参数要和原来的那个指定的socket的地址一样，并且还要为这个新的socket分配文件描述符。