1. 从浏览器输入网址到显示经历了什么

递归查询和迭代查询

1. Udp与tcp的区别

在三次握手过程中，服务器发送SYN-ACK之后，收到客户端的ACK之前的TCP连接称为半连接(half-open connect).此时服务器处于Syn\_RECV状态.当收到ACK后，服务器转入ESTABLISHED状态.

udp的校验和：udp传输的不可靠性，会丢包。udp包的大小可以达到64k，但实际上mtu大小只有1k多，如果直接发一个超过mtu大小的包，就会在协议层被分片，这样的问题是，如果只要有一个分片在传输中出错了即校验不正确（这是较容易发生的），整个传输的udp包就被丢弃。

二进制反码求和 以后再看！！

1. http与https的区别

https是在http协议基础上增加了使用SSL加密传送信息的协议。

在http（应用层协议）和TCP协议之间有一个SSL加密协议

将自己需要传输的超文本协议通过SSL加密

HTTPS和HTTP的区别：

https协议需要到ca申请证书，一般免费证书很少，需要交费。

http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议。

http和https使用的是完全不同的连接方式用的端口也不一样，前者是80，后者是443。

http的连接很简单，是无状态的。

HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

https要解决的问题：

信任问题：

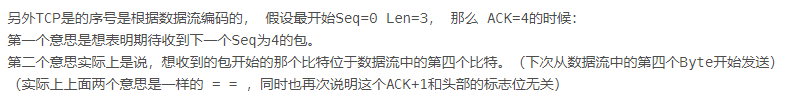
SSL协议

1. tcp三次握手与四次挥手过程：
2. http长连接与短链接：

Connection: Keep-alive

 TCP的keep alive是检查当前TCP连接是否活着；HTTP的Keep-alive是要让一个TCP连接活久点。它们是不同层次的概念。

TCP客户端和服务端两个不同的序列号，可以保证下个数据想要哪个



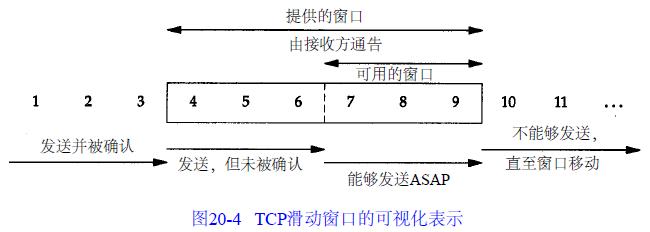
<https://blog.csdn.net/dangzhangjing97/article/details/81008836> 解释的很全面

TCP协议作为一个可靠的面向流的传输协议，其可靠性是由流量控制和滑动窗口协议保证，“窗口”对应的是一段可以被发送者发送的字节序列，其连续的范围称之为“窗口”；TCP是全双工协议，也就是说可以双向传输。因此A和B应该分别维护着一个独立的发送缓冲区和接收缓冲区

 所谓流量控制，主要是接收方传递信息给发送方，使其不要发送数据太快，是一种端到端的控制。主要的方式就是返回的ACK中会包含自己的接收窗口的大小，并且利用大小来控制发送方的数据发送：

所以我们的原则是尽可能一次多发送几个字节，或者窗口空余较多的时候通知发送方一次发送多个字节。对于前者我们广泛使用Nagle算法，即：  
\*1. 若发送应用进程要把发送的数据逐个字节地送到TCP的发送缓存，则发送方就把第一个数据字节先发送出去，把后面的字节先缓存起来；  
\*2. 当发送方收到第一个字节的确认后（也得到了网络情况和对方的接收窗口大小），再把缓冲区的剩余字节组成合适大小的报文发送出去；  
\*3. 当到达的数据已达到发送窗口大小的一半或以达到报文段的最大长度时，就立即发送一个报文段；  
     对于后者我们往往的做法是让接收方等待一段时间，或者接收方获得足够的空间容纳一个报文段或者等到接受缓存有一半空闲的时候，再通知发送方发送数据。

滑动窗口： 注意下面几个概念！！



拥塞控制：

常用的方法就是：  
1. 慢开始、拥塞控制 窗口大小从1开始  
2. 快重传、快恢复 三次收到一样的确认就重传  
     一切的基础还是慢开始，这种方法的思路是这样的：  
-1. 发送方维持一个叫做“拥塞窗口”的变量，该变量和接收端口共同决定了发送者的发送窗口；  
-2. 当主机开始发送数据时，避免一下子将大量字节注入到网络，造成或者增加拥塞，选择发送一个1字节的试探报文；  
-3. 当收到第一个字节的数据的确认后，就发送2个字节的报文；  
-4. 若再次收到2个字节的确认，则发送4个字节，依次递增2的指数级；  
-5. 最后会达到一个提前预设的“慢开始门限”，比如24，即一次发送了24个分组，此时遵循下面的条件判定：  cwnd\_拥塞窗口  ssthresh\_慢启动阈值

\*1. cwnd < ssthresh， 继续使用慢开始算法；  
\*2. cwnd > ssthresh，停止使用慢开始算法，改用拥塞避免算法；  
\*3. cwnd = ssthresh，既可以使用慢开始算法，也可以使用拥塞避免算法；  
-6. 所谓拥塞避免算法就是：每经过一个往返时间RTT就把发送方的拥塞窗口+1，即让拥塞窗口缓慢地增大，按照线性规律增长（由指数增长转为线性增长）；

-7. 当出现网络拥塞，比如丢包时，将慢开始门限设为原先的一半，然后将cwnd设为1，执行慢开始算法（较低的起点，指数级增长）；

慢启动和拥塞避免：

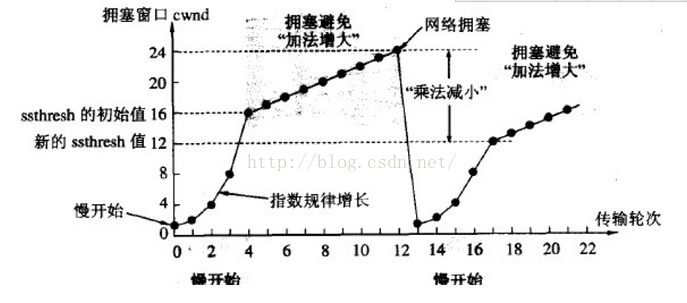
慢启动为发送方的TCP增加了另一个窗口：拥塞窗口(congestion window)，记为cwnd。当与另一个网络的主机建立TCP连接时，拥塞窗口被初始化为1个报文段（即另一端通告的报文段大小）。每收到一个ACK， 拥塞窗口就增加一个报文段（cwnd以字节为单位，但是慢启动以报文段大小为单位进行增加）。发送方取拥塞窗口与通告窗口中的最小值作为发送上限。

拥塞窗口是发送方使用的流量控制，而通告窗口则是接收方使用的流量控制。

拥塞控制： 发送端主动控制控制cwnd，有慢启动（从cwnd初始为1开始启动，指数启动），拥塞避免（到达ssthresh后，为了避免拥塞开始尝试线性增长），快重传（接收方每收到一个报文段都要回复一个当前最大连续位置的确认，发送方只要一连收到三个重复确认就知道接收方丢包了，快速重传丢包的报文，并TCP马上把拥塞窗口 cwnd 减小到1），快恢复（直接从ssthresh线性增长）。

拥塞发生有超时和收到重复确认两种情况：

超时会启用慢启动和避免拥塞算法

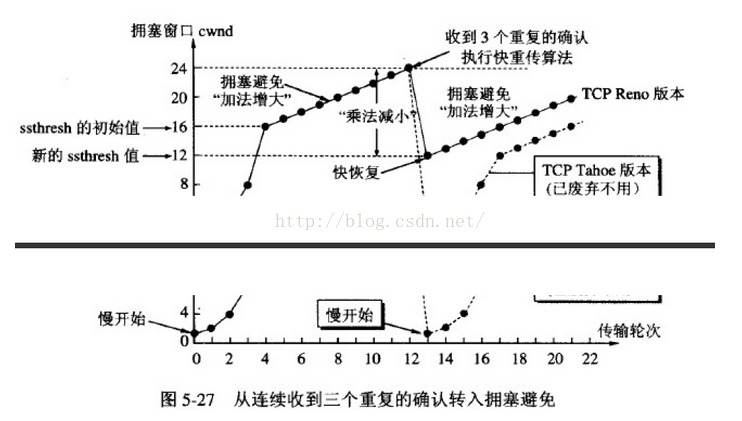


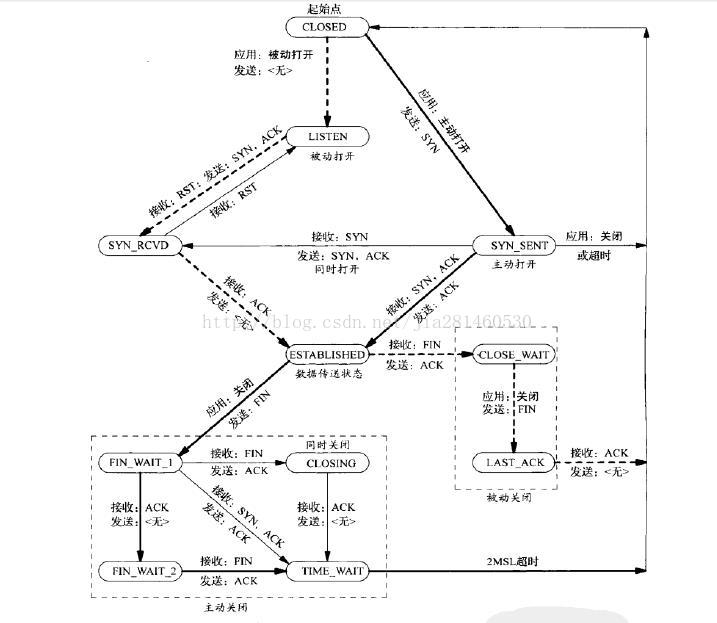
重复确认可以避免过长时间的等待，会启用快速重传和快速恢复

而快重传和快恢复则是为了减少因为拥塞导致的数据包丢失带来的重传时间，从而避免传递无用的数据到网络。快重传的机制是：

-1. 接收方建立这样的机制，如果一个包丢失，则对后续的包继续发送针对该包的重传请求；  
-2. 一旦发送方接收到三个一样的确认，就知道该包之后出现了错误，立刻重传该包；  
-3. 此时发送方开始执行“快恢复”算法：  
\*1. 慢开始门限减半；  
\*2. cwnd设为慢开始门限减半后的数值；  
\*3. 执行拥塞避免算法（高起点，线性增长）；

几个概念：慢开始、拥塞避免、快恢复、快重传





**为什么 TIME\_WAIT 状态需要保持 2MSL 的时间**

如果 TIME\_WAIT 状态保持时间不足够长 ( 比如小于 2MSL) ，第一个连接就正常终止了。 第二个拥有相同相关五元组的连接出现，而第一个连接的重复报文到达，干扰了第二个连接。 TCP 实现必须防止某个连接的重复报文在连接终止后出现，所以让 TIME\_WAIT 状态保持时间足够长 (2MSL) ，连接相应方向上的 TCP 报文要么完全响应完毕，要么被丢弃。建立第二个连接的时候，不会混淆。

1、保证可靠传输，ack可靠到达；2、保证上次连接的消息永远消逝

CP的终止通过双方的四次握手实现。发起终止的一方执行主动关闭，响应的另一方执行被动关闭。

发起方更改状态为FIN\_WAIT\_1，关闭应用程序进程，发出一个TCP的FIN段；

接收方收到FIN段，返回一个带确认序号的ACK，同时向自己对应的进程发送一个文件结束符EOF，同时更改状态为CLOSE\_WAIT，发起方接到 ACK后状态更改为FIN\_WAIT\_2；

接收方关闭应用程序进程，更改状态为LAST\_ACK，并向对方发出一个TCP的FIN段；

发起方接到FIN后状态更改为TIME\_WAIT，并发出这个FIN的ACK确认。ACK发送成功后(2MSL内)双方TCP状态变为CLOSED。

我们不难看出上面的显示的结果的意思。根据TCP协议，主动发起关闭的一方，会进入TIME\_WAIT状态(TCP实现必须可靠地终止连接的两个方向(全双工关闭))，持续2\*MSL (Max Segment Lifetime)，缺省为240秒。

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）

# 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。

**DNS用什么协议：**

DNS占用53号端口，同时使用TCP和UDP协议。那么DNS在什么情况下使用这两种协议？

DNS在区域传输的时候使用TCP协议，其他时候使用UDP协议。

**DNS区域传输的时候使用TCP协议：**

1.辅域名服务器会定时（一般3小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，会执行一次区域传送，进行数据同步。区域传送使用TCP而不是UDP，因为数据同步传送的数据量比一个请求应答的数据量要多得多。

2.TCP是一种可靠连接，保证了数据的准确性。

**域名解析时使用UDP协议：数据量较小较多**

客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。不用经过三次握手，这样DNS服务器负载更低，响应更快。理论上说，客户端也可以指定向DNS服务器查询时用TCP，但事实上，很多DNS服务器进行配置的时候，仅支持UDP查询包。

SSL协议：

SSL是一个介于HTTP协议与TCP之间的一个可选层

发送端：SSL层: 借助下层协议的的信道安全的协商出一份加密密钥，并用此密钥来加密HTTP请求。

接收端：SSL在TCP之上建立了一个加密通道，通过这一层的数据经过了加密，因此达到保密的效果。

SSL协议分为两部分：Handshake Protocol（握手协议）和Record Protocol（记录协议）,。其中Handshake Protocol用来协商密钥，协议的大部分内容就是通信双方如何利用它来安全的协商出一份密钥。 Record Protocol则定义了传输的格式。

密钥协商：用公钥算法协商密钥，用对称密码算法进行通信

简单的说便是：SSL客户端（也是TCP的客户端）在TCP链接建立之后，发出一个Clienth\*llo来发起握手，这个消息里面包含了自己可实现的算法列表和其它一些需要的消息，SSL的服务器端会回应一个Serverh\*llo，这里面确定了这次通信所需要的算法，然后发过去自己的证书（里面包含了身份和自己的公钥）。Client在收到这个消息后会生成一个秘密消息，用SSL服务器的公钥加密后传过去，SSL服务器端用自己的私钥解密后，会话密钥协商成功，双方可以用同一份会话密钥来通信了。

SSL协议要求建立在可靠的传输层协议(例如：TCP)之上。SSL协议的优势在于它是与应用层协议独立无关的

**SSL记录协议：**在SSL协议中，所有的传输数据都被封装在记录中。记录是由记录头和长度不为0的记录数据组成的。所有的SSL通信包括握手消息、安全空白记录和应用数据都使用SSL记录层。SSL记录协议包括了记录头和记录数据格式的规定。

SSL的记录数据包含三个部分：MAC数据、实际数据和粘贴数据。 MAC数据＝HASH[密钥，实际数据，粘贴数据，序号]。当会话的客户端发送数据时，密钥是客户的写密钥(服务器用读密钥来验证MAC数据)；而当会话的客户端接收数据时，密钥是客户的读密钥(服务器用写密钥来产生MAC数据)。序号是一个可以被发送和接收双方递增的计数器。每个通信方向都会建立一对计数器，分别被发送者和接收者拥有。

**SSL握手协议：**SSL握手协议包含两个阶段，第一个阶段用于建立私密性通信信道，第二个阶段用于客户认证。

HTTP+SSL = HTTPS

数字证书内容大体如下：

* 签发证书的机构
* 鲍勃的加密算法
* 鲍勃所使用的Hash算法
* 鲍勃的公钥
* 证书到期时间

数字证书是由权威机构——CA机构统一来进行发行，我们绝对信任这个机构。

好，我们来梳理下整个过程：

* **第一步：**首先，当爱丽丝开启一个新的浏览器第一次去访问鲍勃的时候，会先让爱丽丝安装一个**数字证书**，这个数字证书里包含的主要信息就是CA机构的公钥。
* **第二步：**鲍勃发送来了CA机构颁发给自己的数字证书，爱丽丝通过第一步中已经得到的公钥解密CA用私钥加密的Hash-a(**这个过程就是非对称加密**)，然后再用传递过来的HASH算法生成一个Hash-b，如果Hash-a === Hash-b就说明认证通过，确实是鲍勃发过来的。

### SSL协议的握手过程

先用语言来阐述下：

1. **第一步**：爱丽丝给出支持SSL协议版本号，一个客户端**随机数**(Client random，请注意这是第一个随机数)，客户端支持的加密方法等信息；
2. **第二步：**鲍勃收到信息后，确认双方使用的加密方法，并返回数字证书，一个服务器生成的**随机数**(Server random，注意这是第二个随机数)等信息；
3. **第三步：**爱丽丝确认数字证书的有效性，然后生成一个新的**随机数**(Premaster secret)，然后使用数字证书中的公钥，加密这个随机数，发给鲍勃。
4. **第四步：**鲍勃使用自己的私钥，获取爱丽丝发来的**随机数**(即Premaster secret)；(第三、四步就是非对称加密的过程了)
5. **第五步：**爱丽丝和鲍勃通过约定的加密方法(通常是[AES算法](https://zh.wikipedia.org/wiki/%E9%AB%98%E7%BA%A7%E5%8A%A0%E5%AF%86%E6%A0%87%E5%87%86))，使用前面三个随机数，生成**对话密钥**，用来加密接下来的通信内容；

计算机网络中常用协议：

物理层：

数据链路层：帧 **网卡的地址，就是数据包的发送地址和接收地址，这叫做MAC地址。**

有了MAC地址，就可以定位网卡和数据包的路径了。

我们现在有了网卡，也知道每一块网卡都有一个世界上独一无二的Mac地址，那发送者应该怎么去找接受者这台设备呢，换句话说发送者怎么才能知道接受者的Mac地址呢？这就需要另一个协议了叫做ARP协议，这个协议留在后面介绍。这里我们只需要知道，发送者必须要知道接受者的Mac地址才能准确的发送数据。

广播的方式，发送者将数据包发送给本网络内的所有计算机，然后接受者判断自己是不是接收方

网络层: 走到这里我们实现的只是在一个子网络里传送数据。但我们知道，互联网实际上是由大大小小的子网络组成的：广播的方式只能在子网络内进行，不同子网络之间广播方式是行不通的。因此我们需要一种方法能够判断两台主机是否在同一个子网络之内，如果在同一个子网络就以广播的方式传输数据，如果不在同一个子网络就以路由的方式传输

网络的作用是引进一套新的地址，使得我们能够区分不同的计算机是否属于同一个子网络。这套地址就叫做”网络地址”，简称”网址”。

网址判断两台计算机是否在同一个子网络，Mac地址则是将数据准确的传递到目标计算机中。因此逻辑上可以判断必定是先处理网络地址，再处理Mac地址。

IP协议： IP地址分为网络部分和主机部分，通过子网掩码来区分两部分，其网络部分全为1，主机部分全为0，看两个IP地址的网络部分是否相同来判断是否在同一个网络中

ARP协议：已知IP地址，获取对应的MAC地址。一种是两台计算机在同一个子网络，那么我们可以用ARP协议，得到对方的MAC地址。ARP协议也是发出一个数据包（包含在以太网数据包中），其中包含它所要查询主机的IP地址，在对方的MAC地址这一栏，填的是FF:FF:FF:FF:FF:FF，表示这是一个”广播”地址。它所在子网络的每一台主机，都会收到这个数据包，从中取出IP地址，与自身的IP地址进行比较。如果两者相同，都做出回复，向对方报告自己的MAC地址，否则就丢弃这个包。

传输层：端口号用于区分不同的进程。端口号其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。确切的说，传输层实现的是端对端的服务，网络层实现的仅仅是主机到主机之间的服务。只要确定主机和端口，我们就能实现程序之间的交流。因此，Unix系统就把主机+端口，叫做”套接字”（socket）。

UDP协议：

TCP协议：

应用层：“应用层”的作用，就是规定应用程序的数据格式。

DNS：协议：域名

网关：路由器的IP。如果两台电脑不在同一个子网络，就无法知道对方的MAC地址，必须通过网关（gateway）转发。网关通过路由协议查找所在的子网。

DHCP协议：动态IP地址上网。每一个子网络中，有一台计算机负责管理本网络的所有IP地址，它叫做”DHCP服务器”。新的计算机加入网络，必须向”DHCP服务器”发送一个”DHCP请求”数据包，申请IP地址和相关的网络参数。

接收方的MAC地址填入一个广播地址：FF-FF-FF-FF-FF-FF。

发出方的IP地址就设为0.0.0.0，接收方的IP地址设为255.255.255.255。

这个数据包构造完成后，就可以发出了。以太网是广播发送，同一个子网络的每台计算机都收到了这个包。因为接收方的MAC地址是FF-FF-FF-FF-FF-FF，看不出是发给谁的，所以每台收到这个包的计算机，还必须分析这个包的IP地址，才能确定是不是发给自己的。当看到发出方IP地址是0.0.0.0，接收方是255.255.255.255，于是DHCP服务器知道”这个包是发给我的”，而其他计算机就可以丢弃这个包。

通过以下四个地址才能上网：

本机的IP地址

子网掩码

网关的IP地址

DNS的IP地址

## 请你说说TCP/IP数据链路层的交互过程

### 参考回答：

网络层等到数据链层用mac地址作为通信目标，数据包到达网络等准备往数据链层发送的时候，首先会去自己的arp缓存表(存着ip-mac对应关系)去查找改目标ip的mac地址，如果查到了，就讲目标ip的mac地址封装到链路层数据包的包头。如果缓存中没有找到，会发起一个广播：who is ip XXX tell ip XXX,所有收到的广播的机器看这个ip是不是自己的，如果是自己的，则以单拨的形式将自己的mac地址回复给请求的机器

## 请你说说传递到IP层怎么知道报文该给哪个应用程序，它怎么区分UDP报文还是TCP报文

### 参考回答：

根据端口区分；

看ip头中的协议标识字段，17是udp，6是tcp

## 请问tcp握手为什么两次不可以？为什么不用四次？

### 参考回答：

两次不可以：tcp是全双工通信，两次握手只能确定单向数据链路是可以通信的，并不能保证反向的通信正常

不用四次：  
本来握手应该和挥手一样都是需要确认两个方向都能联通的，本来模型应该是：  
1.客户端发送syn0给服务器  
2.服务器收到syn0，回复ack(syn0+1)  
3.服务器发送syn1  
4.客户端收到syn1，回复ack(syn1+1)  
因为tcp是全双工的，上边的四部确认了数据在两个方向上都是可以正确到达的，但是2，3步没有没有上下的联系，可以将其合并，加快握手效率，所有就变成了3步握手。

首部开销

TCP首部开销大，首部20个字节。

UDP首部开销小，8字节。（源端口，目的端口，数据长度，校验和）

ACK SYN是TCP报文中的两个标志位

三次握手原因：

三次握手是为了防止，客户端的请求报文在网络滞留，客户端超时重传了请求报文，服务端建立连接，传输数据，释放连接之后，服务器又收到了客户端滞留的请求报文，建立连接一直等待客户端发送数据。而此时客户端已经关闭了

服务器对客户端的请求进行回应(第二次握手)后，就会理所当然的认为连接已建立，而如果客户端并没有收到服务器的回应呢？此时，客户端仍认为连接未建立，服务器会对已建立的连接保存必要的资源，如果大量的这种情况，服务器会崩溃

为什么TCP协议终止链接要四次？

1、当客户端确认发送完数据且知道服务器已经接收完了，想要关闭发送数据口（当然确认信号还是可以发），就会发FIN给服务器。

2、服务器收到客户端发送的FIN，表示收到了，就会发送ACK回复。

3、但这时候服务器可能还在发送数据，没有想要关闭数据口的意思，所以服务器的FIN与ACK不是同时发送的，而是等到服务器数据发送完了，才会发送FIN给客户端。

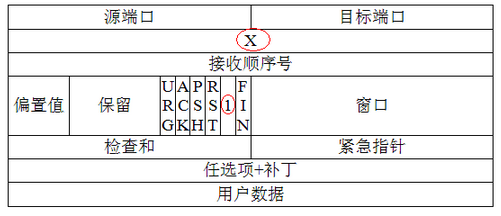
4、客户端收到服务器发来的FIN，知道服务器的数据也发送完了，回复ACK， 客户端等待2MSL以后，没有收到服务器传来的任何消息，知道服务器已经收到自己的ACK了，客户端就关闭链接，服务器也关闭链接了。

2MSL意义：

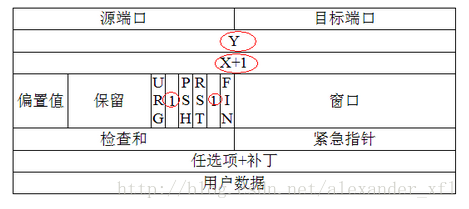
1、保证最后一次握手报文能到B，能进行超时重传。

2、2MSL后，这次连接的所有报文都会消失，不会影响下一次连接。

#### 第一次握手：

客户端发送一个TCP的SYN标志位置1的包指明客户打算连接的服务器的端口，以及初始序号X,保存在包头的序列号(Sequence Number)字段里。  


#### 第二次握手：

服务器发回确认包(ACK)应答。即SYN标志位和ACK标志位均为1同时，将确认序号(Acknowledgement Number)设置为客户的I S N加1以.即X+1。  


#### 第三次握手：

客户端再次发送确认包(ACK) SYN标志位为0,ACK标志位为1.并且把服务器发来ACK的序号字段+1,放在确定字段中发送给对方.并且在数据段放写ISN的+1  


HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

释放连接TCP连接

若connection 模式为close，则服务器主动关闭TCP连接，客户端被动关闭连接，释放TCP连接;若connection 模式为keepalive，则该连接会保持一段时间，在该时间内可以继续接收请求;

## 请你来说一下GET和POST的区别

### 参考回答：

1、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（返回数据）；数据放在url中 是幂等的，适合查询，幂等保证每次查询的结果一样。

而对于POST，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）将数据放在html header中，不是幂等的，适合新增操作。

2、区别：

1、get参数通过url传递，post放在request body中。

2、get请求在url中传递的参数是有长度限制的，而post没有。

3、get比post更不安全，因为参数直接暴露在url中，所以不能用来传递敏感信息。

4、get请求只能进行url编码，而post支持多种编码方式。

5、get请求会浏览器主动cache，而post支持多种编码方式。

6、get请求参数会被完整保留在浏览历史记录里，而post中的参数不会被保留。

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

8、GET产生一个TCP数据包；POST产生两个TCP数据包。因为post会有两次tcp的过程，一次发送header，一次发送data

,send的作用是将要发送的数据拷贝到缓冲区，协议负责传输。recv先等待s的发送缓冲中的数据被协议传送完毕，然后从缓冲区中读取接收到的内容给应用层。

## 请你说一下http协议会话结束标志怎么截出来？

### 参考回答：

看tcp连接是否有断开的四部挥手阶段。

Ssh免密登陆，生成.ssh文件

<https://www.cnblogs.com/xiaoaofengyue/p/8080639.html>

ssh远程登陆原理：

1. 客户端向服务器发送远程请求
2. 服务器接收到请求后，将自己的主机的公钥发送给客户端，公钥用于加密
3. 客户端将主机发送来的公钥通过hash算法得出主机公钥的公钥指纹，核对公钥指纹是否正确，以确认当前请求连接的的是我们想要登录的主机。
4. 核对过公钥后，人机交互界面输入yes,客户端会将服务端的公钥保存在$HOME/.ssh/know\_hosts文件中，可通过 cat ～/.ssh/know\_hosts文件中查看保存在客户端的公钥。
5. 客户端用服务端的公钥将密码进行加密并发送给服务端
6. 服务端接收到用自己公钥加密的客户端的密码，用在/etc/ssh下对应的私钥进行解密然后对比密码并返回登录结果，私钥用来解密。

利用ssh-copy-id等方法共享公钥

免密登陆步骤：

1. 在客户端生成一对密钥（公钥/私钥）
2. 将客户端公钥发送个服务端（其他客户端），使用ssh-copy-id。经过ssh-copy-id后接收公钥的服务端会把公钥追加到服务端对应用户的$HOME/.ssh/authorized\_keys文件中
3. 客户端请求（带有自己的用户名和主机名）
4. 服务端根据客户端的用户名和主机名查找对应的公钥，将一个随机的字符串用该公钥加密后发送给客户端
5. 客户端用自己的私钥进行解密收到的字符串，并将解密后的字符串发送给服务端
6. 服务端对比发送出去的和接收到的字符串是否相同，返回登录结果

主要步骤就是将客户端自己生产的公/私钥对copy到服务端

http的长链接与短链接：

  HTTP1.1规定了默认保持长连接（HTTP persistent connection ，也有翻译为持久连接），数据传输完成了保持TCP连接不断开（不发RST包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

TCP的keep alive是检查当前TCP连接是否活着；HTTP的Keep-alive是要让一个TCP连接活久点。它们是不同层次的概念。

    TCP keep alive的表现：

    当一个连接“一段时间”没有数据通讯时，一方会发出一个心跳包（Keep Alive包），如果对方有回包则表明当前连接有效，继续监控。

这个“一段时间”可以设置。

HTTP方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用

<https://blog.csdn.net/bieleyang/article/details/76272699>

基本的思想就是要检测出对方已经关闭的socket，然后关闭它。  
  
1.代码需要判断socket，一旦read返回0，断开连接，read返回负，检查一下errno，如果不是AGAIN，也断开连接。(注:在UNP 7.5节的图7.6中，可以看到使用select能够检测出对方发送了FIN，再根据这条规则就可以处理CLOSE\_WAIT的连接)  
2.给每一个socket设置一个时间戳last\_update，每接收或者是发送成功数据，就用当前时间更新这个时间戳。定期检查所有的时间戳，如果时间戳与当前时间差值超过一定的阈值，就关闭这个socket。  
3.使用一个Heart-Beat线程，定期向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。  
4.设置SO\_KEEPALIVE选项，并修改内核参数

TIME\_WAIT与CLOSE\_WAIT的产生原因和解决办法

<https://blog.csdn.net/superinzaghi747/article/details/80842504>

<https://blog.csdn.net/libaineu2004/article/details/78886182>

TCP中close与shutdown的区别

<https://www.cnblogs.com/kiven-code/p/3822252.html>

shutdown可以指定在某个方向上终止连接，通过指定标志：SHUT\_RD, SHUT\_WR, SHUT\_RDWR。比如指定SHUT\_WR后可以继续读数据，但不能写入了。而close是两个方向上终止连接。

close会将描述符的引用计数减一，如果引用计数变为0就关闭描述符，发送FIN。而shutdown不管引用计数，直接发送FIN终止连接。所以在多线程下操作同一个socket描述符下，一个线程调用shutdown会使其他线程无法使用这个描述符，而调用close就不会影响到其他线程

TCP心跳包 ，SO\_KEEPALIVE

<https://blog.csdn.net/callinglove/article/details/38380673>

TCP的RST报文段

<https://blog.csdn.net/BryantLmm/article/details/81671457>

TCP粘包

TCP no\_delay字段 不允许发送端先缓存数据到一定大小后才发送，也不允许接收端延迟ACK :

因为有时发送端的Nagle算法（可以认为是流量控制？）和接收端的延迟ACK会影响效率：

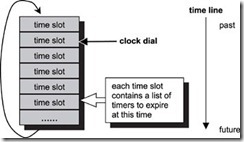
当一个消息被分成AB两部传输，但是B部分比较小，A传输过去，因为接收端有延迟ACK，可能没有返回ACK，而B数据量太小，首先加入缓冲区内，缓冲区内的数据量到达一定大小后再发送，这种情况下，可能会一直等到超时后才会发送B，从而影响了传输效率

TCP的延迟ACK

如何设计俄罗斯方块

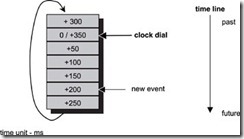
定时器如何实现（时间轮、时间堆）：<https://www.cnblogs.com/zhongwencool/p/timing_wheel.htm>

如下图Figure11.11所示：时间轮是一个固定大小的数组结构，这个数组的每一个槽（元素）代表着软定时器的精度，(类似于时钟的最小刻度).时间轮的优点:通过排序的时间列表来有效的更新timers.它能非常效率地安装（instaallation),取消(cancellation）timer.(这2个操作的时间复杂度o(1)).

[](https://images0.cnblogs.com/blog/629822/201406/271246275086694.jpg)

每个time slot包含当前时间的一系列的过期事件 这些time slot用一个固定的数组表示，并且是按时间排序的，所以安装、取消的时间复杂度是O(1) ，每个槽里的事件用一个双向链表存储 timer过期会触发callback调用

时钟转盘每过一个tick就会指向下一时间(next time)，当指针指到数组的最后一个槽时，下一时间又会回到指针最开始的槽。时间轮的概念就来自于此。因此：当安装一个新的事件(time event)时，转盘当前的位置决定了这个新事件到底应该放在哪一个槽，如下图Figure11.13所描述，每经过一个槽代表过去50ms

[](https://images0.cnblogs.com/blog/629822/201406/271246292114780.jpg)

上面这个时间轮方法存在的系列的问题：

**问题1：** 槽的数量是有限的（也许不同的系统会有不同的限制），比如：你可以在Figure11.13非常明显地看出来：最大的可处理长度(总槽度)是350ms,当要安装一个400ms的事件时怎么办？这个会引起时间轮溢出，为了解决这个问题：一种方法就是禁止超过范围的事件安装，另一个更好的方法：把这些溢出的事件放在一个统一的事件缓冲(event buffer)里面，等转盘转到下一刻度时就从buffer中取出符合范围的事件，这样这些事件也可以被处理了，你可仔细研究Figure11.14得到答案：

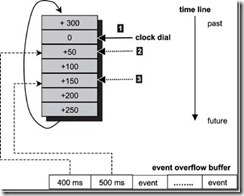
[](https://images0.cnblogs.com/blog/629822/201406/271246299924652.jpg)

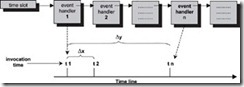
Figure 11.14: Timing wheel overflow event buffer.

比如：当转盘位置在0刻度(图中位置1)处时，同时要安装一个400ms的timeout，这个事件必须暂时存在溢出缓冲buffer里面，随着转盘转到+50ms(图中位置2处），就会从缓冲区取出这个事件安装. 同理：500ms的事件也只能是在转盘到+150ms(图中位置3)处才能安装。转盘每指向下一刻时，都会检查这个事件缓冲区，这就要求缓冲区里面的事件列表是正增长，如果这个列表很长，那么新的事件插入时代价会非常大。

比如：当转盘位置在0刻度(图中位置1)处时，同时要安装一个400ms的timeout，这个事件必须暂时存在溢出缓冲buffer里面，随着转盘转到+50ms(图中位置2处），就会从缓冲区取出这个事件安装. 同理：500ms的事件也只能是在转盘到+150ms(图中位置3)处才能安装。转盘每指向下一刻时，都会检查这个事件缓冲区，这就要求缓冲区里面的事件列表是正增长，如果这个列表很长，那么新的事件插入时代价会非常大。

**问题2：**这个时间轮的精度，试想一下：当tick指到time wheel 到开始指向下一个时间刻度前，如又安装一个150ms的事件，那么这个事件是安装在+150ms，还是在+200呢？按平均来讲，出错的概率均等的情况下，那么这个出错可能会延迟或提前最小刻度的一半，在这里就是50ms/2=25ms.

**问题3：**非常重要的问题：关于callbacks的安装.理论上，每一个Callback都应该在时间过期时同时发生，但是在现实中，这是不可能的，每一个Callback的工作状态都不可预测，因此，执行的每一个callback的长度也不可预测，导致没有方法可以保证一个在很长列表后面的callback会被马上执行，这个问题是不合需求的，不能引放到系统里面。Figure11.15描述了这个问题：

[](https://images0.cnblogs.com/blog/629822/201406/271246308673553.jpg)

### 层时间轮(Hierarchical Timing Wheels)

Figure11.14里面的**问题1**：溢出问题可以使用分层时间轮的方法解决。

软时间设备需要适应事件在跨越在不同范围的值，这个跨度可以非常大，比如：适应timers 范畴从100ms到5 分钟需要3000（(5 × 60 × 10）跨度的时间轮，因为这个时间轮的精度最少要100ms，这也是此时间轮的最小精度啦：

10 × 100ms = 1 sec

10 entries/sec

60 sec = 1 minute

60 × 10 entries / min

因此:

5 × 60 × 10 =需要3000个刻度(entries).

一个分层的时间轮就好像一个数字刻盘指针型时钟，用多个时间轮安装在这个分层结构里面，取代上面单一的时间轮。这里面每个时间轮都有自己的粒度(granularity)精度，时间转盘与所有的时间轮联系在一起，当最低层的时间轮转一轮时，上一层的时间轮就转一个单位。使用分层时间轮刚上面的需要3000entries的现在仅需要75(10 + 60 + 5)entries就可以保证timeout从100ms到5分钟。这里用到的多维数组：

[复制代码](javascript:void(0);)

10 × 100ms = 1 sec

10 entries/sec

60 sec = 1 minute

60 entries / min

5 entries for 5 minutes

因此：

5 + 60 + 10 =只需要75个刻度(entries)

[复制代码](javascript:void(0);)

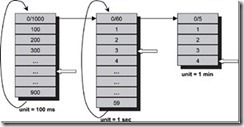
[](https://images0.cnblogs.com/blog/629822/201406/271246316791669.jpg)

Figure 11.16: A hierarchical timing wheel

这个模型不仅节省了大量的空间，并且保持着很高的精度和跨度，Figure11.16说明了这一点。

举个例子：它可能会安装一个2分4秒300ms处timeout事件。首先安装2min，当2分钟发生时，它检查还有4.3s的事件才能timeout,所以它又安装了4s的timeout事件槽，当4s过去后，检查到还有300ms才能timeout,又安装了一个300ms事件，再过300ms,真正的timeout才会被执行.

时间堆：<https://blog.csdn.net/liushall/article/details/81331591>

由于定时器的触发是由于时间到了，因此只有时间最短的定时器会首先被触发，通过这个原理，我们可以采用最小堆，将按时间顺序排序，堆顶元素是时间最短的定时器，因此只要判断堆顶元素是否被触发即可。只有堆顶定时器的时间到了，才会到其他时间较晚的定时器的时间。