计算机网络目录

1. 网络的字节序 4

2. 网络知识 tcp三次握手 各种细节 timewait状态 4

3. Tcp三次握手一共传输了多少包，有什么缺陷？（任意包丢失的解决方法） 6

4. TCP四次挥手讲一下过程，最后一次ack如果客户端没收到怎么办，为什么挥手不能只有三次，为什么time\_wait。 6

5. 对于socket编程，accept方法是干什么的，在三次握手中属于第几次，可以猜一下，为什么这么觉得。 8

6. tcp怎么保证有序传输的，讲下tcp的快速重传和拥塞机制，知不知道time\_wait状态，这个状态出现在什么地方，有什么用？ 8

7. TCP如何实现可靠性传输？ 9

8. TCP队头阻塞和HTTP队头阻塞（字节跳动） 10

9. 知道udp是不可靠的传输，如果你来设计一个基于udp差不多可靠的算法，怎么设计？（扩展在不调整链路层的情况下，怎么提高udp的可靠性：发标记的冗余包） 11

10. HTTP协议有哪些类型请求？ 11

11. HTTPS的加密过程 11

12. http与https有啥区别？说下https解决了什么问题，怎么解决的？说下https的握手过程。 12

13. 对称加密有哪些算法？MD5算法 13

14. http 都基于Tcp么、服务器如果同时返回文件和图片 客户端 怎么处理 13

15. tcp 粘包半包、拆包问题怎么处理？ 14

16. HTTP请求报文和响应格式？（HTTP请求头每个字段的作用） 15

16

17. keepalive 是什么东东？如何使用？ 18

18. 列举你所知道的tcp选项，并说明其作用。 19

19. 流量控制，拥塞控制 19

20. TCP和UDP区别，TCP如何保证可靠性，对方是否存活(心跳检测) 22

21. TCP报文格式 23

22. http报文格式 24

23. http中get和post的区别 26

24. http1.0与http1.1区别，http2.0特性，http3.0特性 26

21. Client和server建立tcp长连接，第一种情况服务器直接挂了，第二种情况服务器进程挂了，两种情况下client端会做出什么反应 30

22. 孤儿进程和守护进程 30

23. 僵尸进程，介绍一下僵尸进程，怎么避免僵尸进程 31

24. 网络层、数据链路层、传输层的设备有哪些？ 31

25. OSI七层模型及其设备，五层模型、四层模型 32

26. 网络层、传输层协议有哪些 32

27. tcp如何设定超时时间 32

28. SYN Flood攻击 32

29. Nagle 算法与延迟确认 34

30. 从输入URL到页面显示的过程（延伸TCP和UDP的区别） 35

31. TCP的time\_wait状态出现在什么时候？有什么作用？ 36

33. TCP具有超时重传策略？如果一直超时怎么办？如何解决？ 37

34. TCP的快速重传机制？什么时候发生重传？重传哪些数据包？ 37

35. IO多路复用如何实现的？epoll相对于select有什么优点？epoll的垂直和水平触发方式？ 38

36. UDP最大包长多少？首部的字段和长度？为什么是1500字节呢？ 38

37. 包丢了，tcp的发送端措施，出了重传机制，还有别的吗？ 38

38. Tcp/ip的5层，osi的七层 38

39. 说一下osi七层协议，强调可靠性问题 39

40. http在哪一层，tcp在哪一层，arp在哪一层 39

41. 路由表的建表过程，如何根据网络的变化进行更新 39

42. DNS查询过程 40

43. arp协议 40

44. Ping原理 41

45. 子网内计算机通信需要用到路由器吗？ 41

46. 交换机和路由器的区别 工作方式 41

47. http响应返回码 44

48. 滑动窗口及其作用 44

49. Cookie和Session（与各自应用场景） 44

50. 客户端禁用了cookie、session还能用吗？ 47

51. TCP为什么数据量比UDP大？如果有1M数据，两者会差多少？ 47

52. IEG光子 47

53. TCP为什么比UDP慢 47

54. MSS 和 MTU 47

55. UDP接受到包不是按顺序的，怎么办 47

（发送端+包标号 接收端+允许短暂时延） 48

56. Udp怎么保证分的包能被正确接受？ 48

57. Udp，可靠实现，应用场景（qq聊天为啥要用udp而非tcp） 48

58. TCP/UDP的各自应用场景 48

59. TCP延迟确认机制 49

60. HTTP长连接短链接 49

61. 高并发问题的解决历程 49

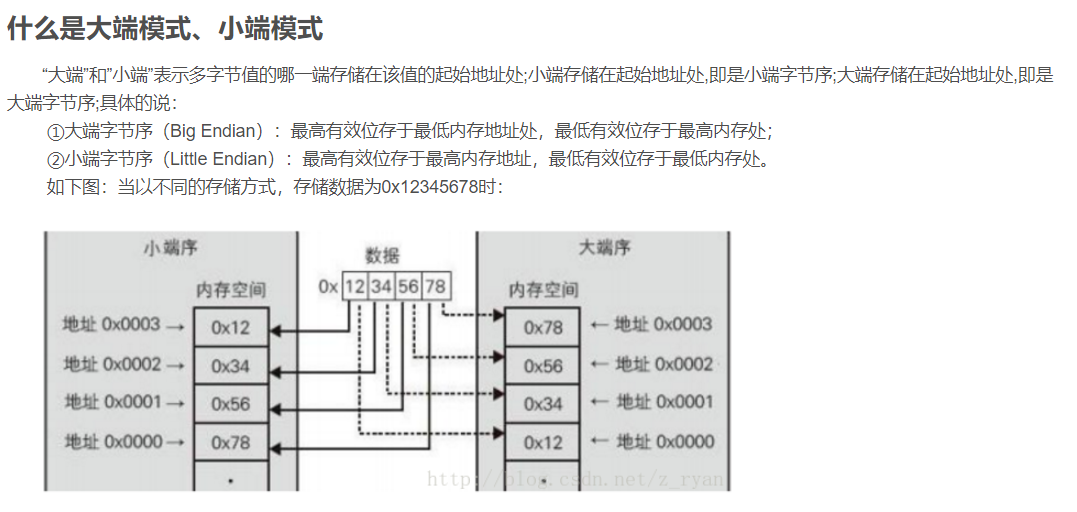
62. 假设我们写个服务器，估计要支持几万个人同时在线，这时候应该怎么来做？（负载均衡+IO多路复用+数据库扩展（+微服务+消息队列+）多台云服务器） 51

63. 你的项目如何实现负载均衡 51

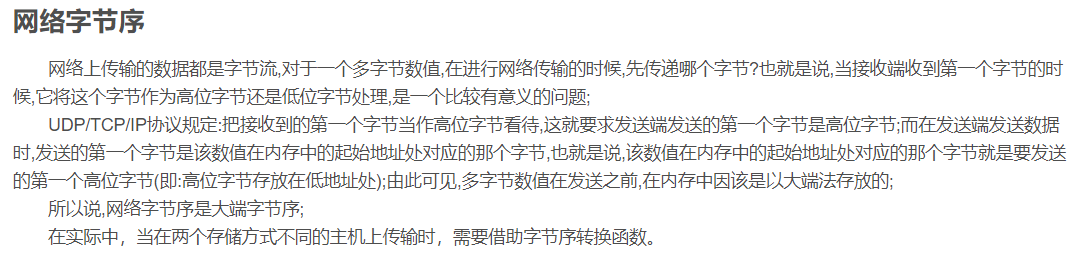
64. 概括性问题：如何设计对战服务器（详细） 53

65. 53

1. 网络的字节序



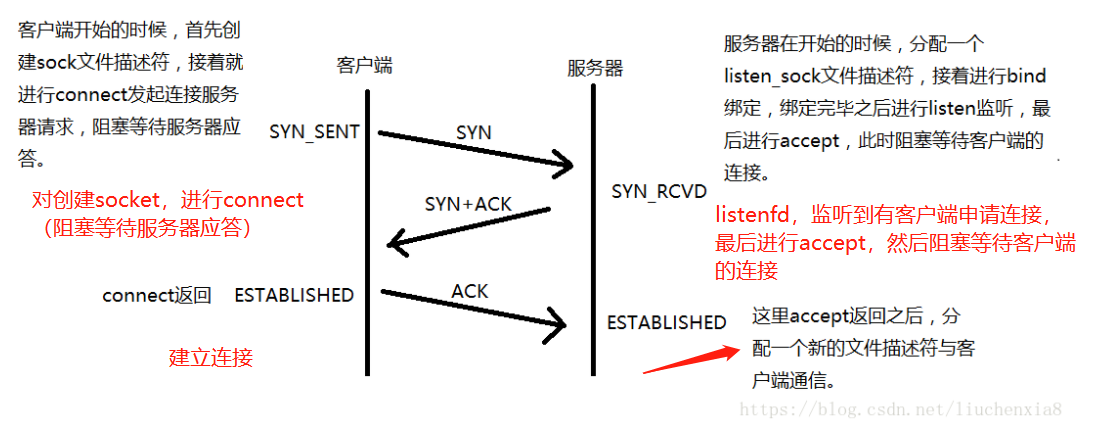
1.判断大小端方法：(1)union 所占内存法(因为union型数据内存空间为其最大成员的内存空间) (2)int->char强转保存法



主机字节序大部分为小端字节序，故在UDP/TCP/IP协议规定：把接受到的第一个字节当作高位字节看待，这就要求发送端的第一个字节是高位字节。（故网络编程中，需对传递进行hton()处理，host->to->net），网络字节序即为大端字节序。

1. 网络知识 tcp三次握手 各种细节 timewait状态

在TCP/IP协议中，TCP协议提供可靠的连接服务，why？肯定有其原因。即：采用三次握手建立一个连接



* 第一次握手：Client(客户端)先产生一个初试序列号Seq = ISN(C)，作为SYN(同步序列编号。是TCP/IP建立连接时使用的握手信号)并将数据包发送给Server(服务器端)，Client进入SYN\_SENT状态，等待Server确认。
* 第二次握手：Server收到数据包后也发送自己的SYN报文作为相应，并初始化序列号Seq = ISN(S)，为了确认Client的Seq，Server将Client发送的Seq加1，作为ACK(即是确认字符，在数据通信中，接收站发给发送站的一种传输类控制字符)发送给Client，Server进入SYN\_RCVD状态。
* 第三次握手：为了确认Server的SYN，Client将Server发送的Seq加1，作为ACK发送给Server。Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

通过这样的三次握手，客户端和服务器端建立起可靠的双工连接，开始传送数据。三次握手的最主要目的是保证连接是双工的，**可靠更多的是通过重传机制**来保证的。

引申：半连接、半打开、半关闭状态

* 半连接状态：发生在TCP三次握手过程中，客户端向服务器端发起连接，服务器也进行了回应，但是客户端却不进行第三次握手
* 半打开状态：在TCP连接中，如果某一端关闭了连接或者是异常关闭，则该连接处于半打开状态。解决半打开问题：引入心跳机制就可以察觉半打开状态
* 半关闭状态：当TCP连接中客户端向服务器端发送FIN请求关闭，服务器端回应ACK之后，并没有立即发送FIN给客户端，客户端就处于半关闭状态，此时客户端可以接收服务器发送的数据，但是客户端已经不能再向服务器发送数据。

原因：为什么需要三次握手？

为了保证服务端能接收到客户端的信息并能做出正确的应答而进行前两次(第一次和第二次)握手，为了保证客户端能够接收到服务端的信息并能做出正确的应答而进行后两次(第二次和第三次)握手。

* 如：甲—>乙：你吃饭了吗？
* 已—>甲：我吃了，你呢？
* 甲—>乙：我也吃了。

问题展开：

1. 在三次握手过程中，如果服务器一直收不到客户端的ACK会发生什么？

服务器端会给每个待完成的半连接都设一个定时器，如果超过时间还没有收到客户端的ACK信息，则重新发送一次SYN-ACK消息给客户端，直到重试超过一定次数时才会放弃。这个时候服务器需要分配内核资源维护半连接。

1. 初始化序列号Seq为什么要随机初始化？

这样做主要为了保证网络安全，如果不是随机产生初始序列号，黑客将会以很容易的方式获取到你与其它主机之间通信的初始化序列号，并且伪造序列号进行攻击，这已经称为一种很常见的网络攻击手段

* TCP 的可靠性含义： 接收方收到的数据是完整， 有序， 无差错的。
* UDP 不可靠性含义： 接收方接收到的数据可能存在部分丢失， 顺序也不一定能保证。

为什么不可以是两次握手？

当客户端向服务器端发送一个连接请求时，由于某种原因长时间驻留在网络节点中，无法到达服务器端，由于TCP的超时重传机制，当客户端在特定的时间内没有收到服务器端的的确认应答时，就会重新向服务器端发送连接请求，该请求到达服务器端并建立连接，进行数据传输，当数据传输完成时，释放了TCP连接。

若此时第一次的连接请求报文段延迟了一段时间后到达了服务器端，本来这是一个很早到达的失效的报文段，但是服务器端收到了该链接请求后误以为是客户端重新又发起了一次连接请求，于是服务器端发出确认应答报文段，并表示同意建立连接。如果没有第三次握手，由于服务器端发送了确认应答信息，则表示新的连接建立成功，但是客户端并没有向服务器端发送任何建立请求，客户端将忽略服务器端的确认报文，更不会发送任何请求或数据。而服务器端认为建立成功了，并一直在等待建立连接，直到超出计数器的设定值，则认为服务器端出现了异常，并关闭此链接。这个等待的过程中，浪费了服务器端的资源。

如果是三次握手将不会出现不该建立的连接。

1. Tcp三次握手一共传输了多少包，有什么缺陷？（任意包丢失的解决方法）

当Client 端 三次握手是收到Server 的SYN + ACK 应答后， 其状态变为established，并发送ACK包给Server

此时ACK在网络中丢失，会怎么样？

Server端该TCP连接的状态为SYN\_RECV, 并且依次等待3s,6s,12s后重新发送SYN+ACK

包，以便Client 重新发送ACK包，Server重发SYN+ACK的次数，可以通过修改配置文件修改（/proc/sys/net/ipv4/tcp\_synack\_retries），默认是5；

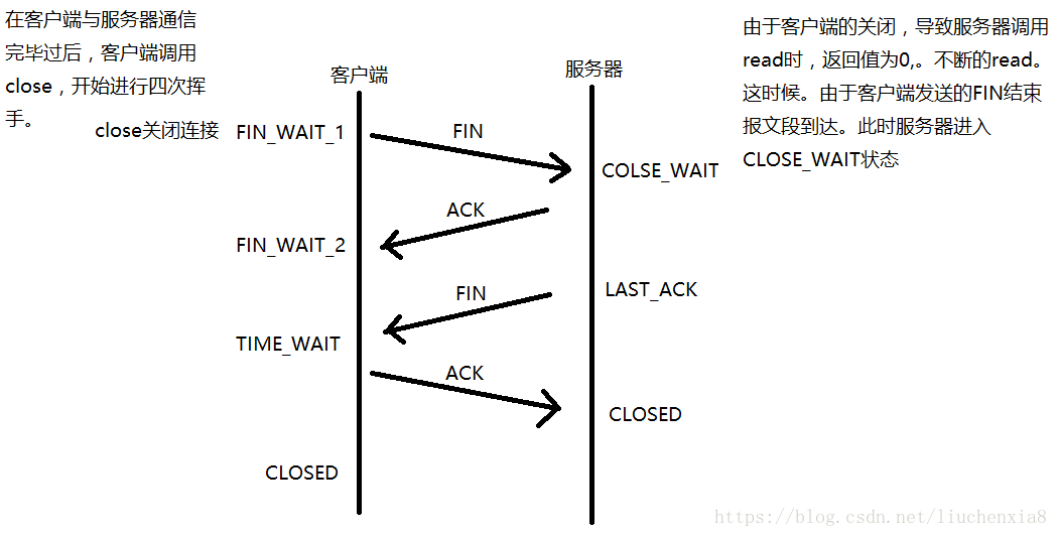
如果指定的次数后，仍然未收到ACK应答， 那么一段时间后，Server会自动关闭这个连接

但是Client，认为这个连接已经建立，如果Client向Server继续发送数据，Server端将以RST包（用于强制关闭tcp连接）响应，方能感知到Server出现错误。

拓展

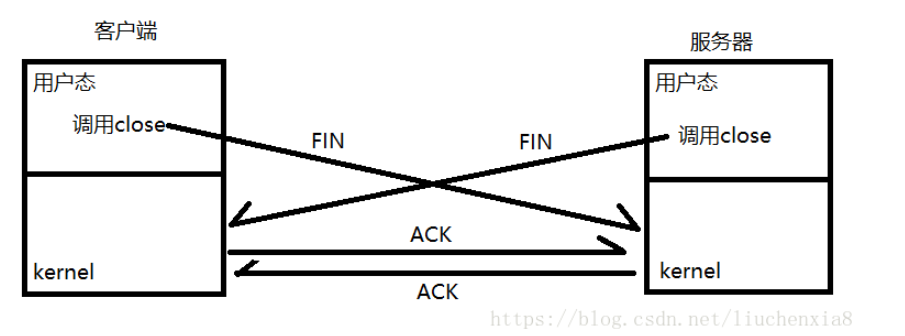
如果是故意不发ACK，就是大量连接只发送握手信号SYN，不发第三个SYN+ACK，也不发数据，那这个就是简单直接的SYN洪水攻击了。

1. TCP四次挥手讲一下过程，最后一次ack如果客户端没收到怎么办，为什么挥手不能只有三次，为什么time\_wait。



* 第一次挥手：Client发送一个FIN，Seq=K，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态
* 第二次挥手：Server收到FIN后，发送一个ACK(K+1)给Client，确认序号为收到序号+1(与SYN相同，一个FIN占用一个序号)，Server进入CLOSE\_WAIT状态
* 第三次挥手：Server发送一个FIN，Seq=L，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态
* 第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK(L+1)给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

问题：为什么建立连接是三次握手，而关闭连接却是四次挥手呢？



在三次握手时，服务端可以把ACK和SYN(ACK起应答作用，而SYN起同步作用)放在一个报文里来发送。但是四次挥手时，服务端发送的FIN与ACK是分开发送的。

原因在于：**首先FIN信号是由于调用close所以才发送的，而ACK是由内核发送的，所以ACK报文和FIN报文在发送的时间上都是分开的，不一定能同时发送。但是三次握手发送SYN是由内核直接完成的，所以这就可以达到一个同步发送的情况。**

TCP连接时是同步的，但结束时是不同步的，当挥手第二次后宣告的了主动关闭方不会再主动发送数据，但仍然可以接收数据，此时处于半关闭状态。这样被动关闭方有足够的时间去处理以前没有处理完的数据，它可能还有一部分数据没发送出去需要处理，在此之后提出主动关闭连接。所以4次挥手的设计为连接双方都提供了一定的处理扫尾工作的时间，从而显的是必要的。 细比一下，这很人性化，连接不是你想关就关的，就仿佛，你说要停电就立马把电停了。这样让我处于一个窘境，提前为手机充满电的时间都没有。故4次挥手的设计这样看来是极有益的

TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。如果要正确的理解四次分手的原理，就需要了解四次分手过程中的状态变化。

引申：

CLOSE\_WAIT状态有什么影响？

如果服务器代码没有调用close，那么意味着并没有发送FIN结束报文段。那么也就是说，此连接的服务器长期保持在CLOSE\_WAIT状态，这会又什么影响？

服务器长期保持在CLOSE\_WAIT状态，也就是说分配的文件描述符并没有关闭并归还。那么大量的CLOSE\_WAIT存在的话，就会导致一种资源的泄露，可能到最后就没有可分配的文件描述符，那么就会使一些客户端无法连接，从而造成不可估量的影响。

socket中的close是一次就关闭的吗？半关闭状态是怎么产生的？

使用close中止一个连接，但它只是减少文件描述符的引用计数，并不直接关闭连接，只有当描述符的引用计数为0时才关闭连接。

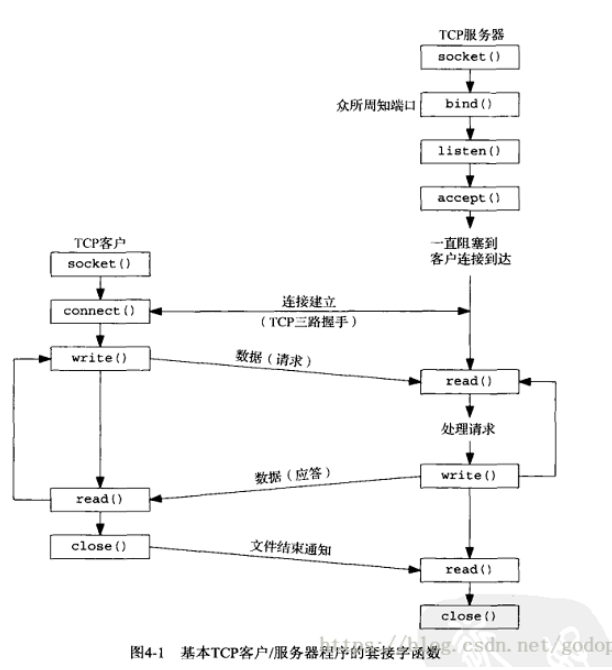
客户端的半关闭状态：收到服务器的ACK后，暂时关闭写端，但是读端并没有关闭，依然可以接受来自服务器的数据。

如果已经建立了连接，但是客户端突发故障了怎么办？

TCP设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端请求都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75分钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

1. 对于socket编程，accept方法是干什么的，在三次握手中属于第几次，可以猜一下，为什么这么觉得。

基本TCP客户端/服务器程序的套接字函数



  accept应该位于第三次握手之后，我们来看看TCP连接过程：

  初始：客户端处于CLOSED状态，而服务器端从CLOSED状态变为LISTEN状态

  第一步：客户端发送一个SYN序列号

  第二步：服务器端接收来自客户端的连接请求，并返回ACK+SYN

  第三步：客户端接收消息，返回ACK

  在第三步之后，服务器接收到消息，连接就完成了。这样就可以调用accept函数获得此连接。

1. tcp怎么保证有序传输的，讲下tcp的快速重传和拥塞机制，知不知道time\_wait状态，这个状态出现在什么地方，有什么用？

tcp保证有序传输：

主机每次发送数据时，TCP就给每个数据包分配一个序列号并且在一个特定的时间内等待接收主机对分配的这个序列号进行确认，如果发送主机在一个特定时间内没有收到接收主机的确认，则发送主机会重传此数据包。接收主机利用序列号对接收的数据进行确认，以便检测对方发送的数据是否有丢失或者乱序等，接收主机一旦收到已经顺序化的数据，它就将这些数据按正确的顺序重组成数据流并传递到高层进行处理。

具体步骤如下：

* 1. 为了保证数据包的可靠传输，发送方必须把已发送的数据包保留在缓冲区
  2. 并为每个已发送的数据包启动一个超时定时器
  3. 如在定时器超时之前收到了对方发来的应答消息(可能是对本包的应答，也可以是对本包后续包的应答)，则释放该数据包占用的缓冲区
  4. 否则，重传该数据包，直到收到应答或重传次数超过规定的最大次数为止
  5. 接收方收到数据包后，先进行CRC循环冗余校验，如果正确则把数据交给上层协议，然后给发送方发送一个累计应答包，表明该数据已收到，如果接收方正好也有数据要发给发送方，应答包也可放在数据包中捎带过去。

TCP的拥塞控制机制：

拥塞其实是一个动态问题，我们没有办法用一个静态方案去解决，从这个意义上来说，拥塞是不可避免的。

为了防止网路的拥塞现象，TCP提出了一系列拥塞控制机制。

拥塞控制也就是考虑当前的网络环境，动态调整窗口大小，没有发生拥塞情况，则窗口增大，拥塞的窗口减少，如此往复，最终应该接近接收端的窗口大小。

1. 慢启动和拥塞避免（门限值初始化为16，之后每次发生拥塞，门限值更改为发生超时或者出现拥塞的一半，门限值是慢启动和拥塞避免算法的分割点）

在开始发送信息时，由于不知道具体网络环境，为避免大量信息造成拥塞现象，此时的拥塞窗口以最小值(即拥塞窗口和接受端窗口中的较小值)进行数据发送，并设定门限值作为慢启动算法和拥塞避免算法的分割点。慢启动是指以最小的拥塞窗口按照指数形式递增，达到门限值后，以拥塞避免算法，即线性递增方式增大拥塞窗口。慢开始门限的初始值设置为16个报文段，即 cwnd = 16

在上述过程中，无论是窗口大小指数递增或者线性递增，当发生拥塞现象，则**门限值更新为当前窗口大小的一半**，拥塞窗口大小变为最小值，重复上述递增过程(此时属于网络环境限制，所以在接收端和拥塞窗口两个限制条件中选择拥塞窗口作为限制)。



1. 快重传和快恢复

在发送端连续收到了三个重复的ack时，表示该数据段已经丢失，需要重发。当收到三个表示同一个数据段的ack时，不需要等待计时器超时，立即重新发送数据段(当然这三个ack要在超时之前到达发送端)，**因为能够收到接收端的ack确认信息，所以数据段只是单纯的丢失，而不是因为网络阻塞导致，所以此时不需要拥塞窗口更新为最小值进行慢启动**(如果这样的话，反倒因为拥塞窗口的增长需要时间，可能导致性能降低)，此时需要设置拥塞窗口大小为：门限值大小+3，当然此处门限值已经更新到拥塞窗口值的一半大小，该行为也就是所谓的“乘法减少”，更新之后按照拥塞避免算法继续进行。

不连续的数据段会严重影响TCP的传输效率。而快速修复这种不连续，会释放掉占用的空间，加快发送方的传输速率。

TCP的快速重传：

拥塞窗口大小为什么先以指数增加再以线性增加？

**窗口大小首先以指数递增去探测一下网络的拥塞程度，执行拥塞避免算法后，拥塞窗口线性缓慢增大，防止网络过早出现拥塞**。

1. TCP如何实现可靠性传输？
2. CRC校验和
3. 确认应答和序列号。

序列号：TCP传输时将每个字节的数据都进行了编号，这就是序列号

确认应答：TCP传输的过程中，每次接收方收到数据后，都会对传输方进行确认应答。也就是发送ACK报文。这个ACK报文中带有对应的确认序列号，告诉发送方，接收到了哪些数据，下一次的数据从哪里发。

序列号的作用不仅仅是应答的作用，有了序列号能够将接收到的数据根据序列号排序，并且去掉重复序列号的数据。这也是TCP传输可靠性的保证之一。

1. 超时重传

TCP提供可靠的运输层。它使用的方法之一就是确认从另一端收到数据。但数据和确认都可能丢失。TCP通过在发送时设置一个定时器来解决这种问题。如果当定时器溢出时还没有收到确认，它就重传该数据。对任何实现而言，关键之处就在于超时和重传的策略，即怎样决定超时间隔和如何确定重传的频率。

对每个连接，TCP管理4个不同的定时器：

1. 重传定时器适用于当希望收到另一端的确认。
2. 坚持(persist)定时器使窗口大小信息保持不断流动，即时另一端关闭了其接收窗口
3. 保活(keepalive)定时器可检测到一个空闲连接的另一端何时崩溃或重启
4. 2MSL定时器测量一个连接处于TIME\_WAIT状态的时间
5. TCP队头阻塞和HTTP队头阻塞（字节跳动）

TCP协议下每个数据包都有唯一的序号，协议能够保证这些包的可靠、按顺序传输。但如果多个包中某个序号比较靠前的在传输过程中发生了丢包，其它序号靠后的包即使已顺利到达，也需要在接收方的缓冲区中等待，直到队头包重传成功才能将报文组装并返回给应用层。

* 队头阻塞发生在TCP协议层，应用层无法感知，应用层只能感知到总体的传输延迟
* 队头阻塞的负面影响在音、视频应用这种不要求所有包都到达即可正常运行的场景下尤其明显

如何解决TCP队头阻塞

TCP中的队头阻塞的产生是由TCP自身实现机制决定的，无法避免。想要在应用程序中避免TCP队头阻塞带来的影响，只有舍弃TCP协议。

比如Google推出的quic协议，在某种程度上可以说避免了TCP中的队头阻塞，因为它根本不使用TCP协议，而是在UDP协议的基础上实现可靠传输。而UDP是面向数据报的协议，数据报之间不会有阻塞约束。

此外还有一个SCTP(流控制传输协议)，它是和TCP、UDP在同一个层次的传输协议。SCTP的多流特征也可以尽可能的避免队头阻塞的情况。

HTTP队头阻塞：

在HTTP/1.1中客户端发送请求，服务器回应客户端的请求，**客户端发送的每一个请求，需要等待服务器回应后才能发送下一个请求**，即所有请求在一个FIFO队列里，如果队头请求意外阻塞，这样就会造成队头阻塞。即如果队头的请求响应时间太长，会影响后面的请求的处理。类比现实中，去银行办理业务，队列前面有个行动迟缓的老太太，崩不崩溃？

在HTTP/1.1中有3个方法可以缓解，但不能解决队头阻塞问题。

* + - 1. 管道：即可以并发多个请求，无需等待上一个请求返回，即现在的FIFO的长度可以>1。但它仍然期待按发送次序收到响应信息，如果某个先发请求阻塞，则后续报文即使处理也不能发出，所以仍然存在队头阻塞问题。管道仅在理想情况下，提高收发速率。而且目前浏览器对管道支持得不好，可以认为是废弃技术
      2. 并发TCP连接，对于同一个域名，浏览器允许同时开6-8个TCP连接，因为每个TCP连接独立，所以某个TCP中的HTTP请求阻塞，不影响其它TCP连接，所以虽然队头阻塞依然存在，但因为存在多个并发TCP连接，所以队头阻塞问题得到一定程度的缓解。
      3. 域名分片，即多个域名映射到一个IP，既可以并发6\*域名的连接，进一步缓解了队头阻塞的负面影响

HTTP/2 解决了用流和分帧的方式HTTP的队头阻塞问题

HTTP/3 解决了用UDP+QUIC解决了TCP队头阻塞问题

1. 知道udp是不可靠的传输，如果你来设计一个基于udp差不多可靠的算法，怎么设计？（扩展在不调整链路层的情况下，怎么提高udp的可靠性：发标记的冗余包）
2. HTTP协议有哪些类型请求？

http请求方法有八种，（其中比较常用的是GET、POST、HEAD）：

OPTIONS

返回服务器针对特定资源所支持的HTTP请求方法，也可以利用web服务器发送 ’\*’ 的请求来测试服务器的功能性。允许客户端查看服务器的性能。

HEAD

向服务器与GET请求相一致的响应，只不过返回的响应中没有具体内容，用于获取报头。

GET

向特定的资源发出请求。它本质就是发送一个请求来取得服务器上的某一资源。资源通过一组HTTP头和呈现数据（如HTML文本，或者图片或者视频等）返回给客户端。GET请求中，永远不会包含呈现数据。

POST

向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 Loadrunner中对应POST请求函数：web\_submit\_data,web\_submit\_form

PUT

向指定资源位置上传其最新内容。从客户端像服务器传送的数据取代指定文档的内容。

DELETE

请求服务器删除Request-URL所标识的资源。请求服务器删除指定的页面。

TRACE

回显服务器收到的请求，主要用于测试或诊断

CONNECT

HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

1. HTTPS的加密过程

Https的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机制。即对称加密与非对称加密混合的加密机制。

1. http与https有啥区别？说下https解决了什么问题，怎么解决的？说下https的握手过程。

http(Hyper Text Transfer Protocol) (超文本传输协议)的缩写

HTTPS(Secure Hypertext Transfer Protocol) 安全超文本传输协议

* HTTP： 直接通过明文在浏览器和服务器之间传递信息。
* HTTPS： 采用 对称加密 和 非对称加密 结合的方式来保护浏览器和服务端之间的通信安全。

它是一个安全通信通过，它基于HTTP开发，用于在客户计算机和服务器之间交换信息。它使用安全套接字层(SSL)进行信息交换，简单来说它是HTTP的安全版。

HTTPS使用端口443，HTTP使用端口80来和TCP/IP进行通信

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，信息是明文传输的，https则是具有安全性的ssl加密传输协议

http和https使用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

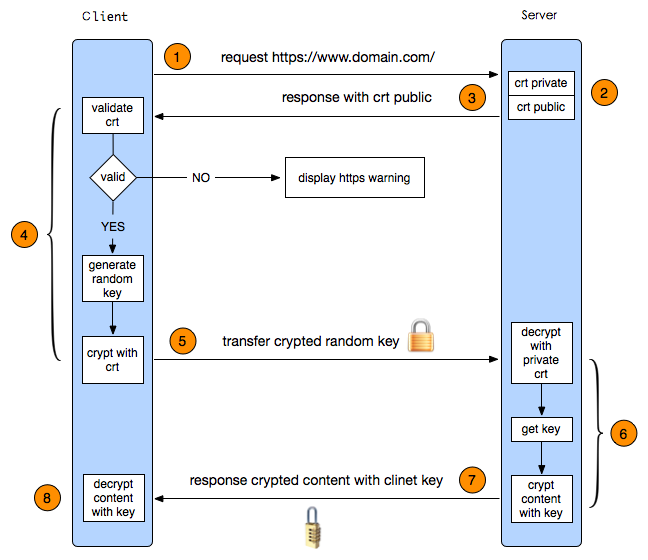
https协议是由ssl+http协议构建的可进行**加密传输、身份确认**的网络协议 要比http协议安全

https解决的问题：

1. 信任主机的问题。采用https的server端必须从ca申请一个用于证明服务器用途的证书
2. 通讯过程中数据的泄密和被篡改。一般意义上的https，就是server有一个证书。主要目的是保证server就是他声称的server，这个跟上一点一样。
3. 服务端和客户端之间所有通讯，都是加密的。具体讲，是客户端产生一个对称的密钥，通过server的证书来交换密钥，一般意义上的握手过程。
4. 加下来所有信息往来都是加密的。第三方即时捕获，也没有任何意义，因为他没有密钥，当然篡改也就没有什么意义了

https的握手过程：

HTTPS在HTTP的基础上加入了SSL协议，SSL依靠证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。具体是如何进行加密，解密，验证的，且看下图，下面的称为一次握手。



1.客户端发起https请求

2.服务器端的配置

采用HTTPS协议的服务器必须有一套数字证书，可以是自己制作或者CA证书。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用CA证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。公钥给别人加密使用，私钥给自己解密使用。

3.传送证书

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发结构，过期时间等。

4.客户端解析证书

这部分工作是由客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发即构、过期时间等，如果发生异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随机值，然后用证书对该随机值进行加密

5.传送加密信息

这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值进行加密解密。

6.服务端解密信息

服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密，所以对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

7.传输加密后的信息

这部分信息是服务端用私钥加密后的信息，可以在客户端被还原

8.客户端解密信息

客户端用之前生成的密钥解密服务端传过来的消息，于是获取解密后的内容。

1. 对称加密有哪些算法？MD5算法
2. http 都基于Tcp么、服务器如果同时返回文件和图片 客户端 怎么处理
3. tcp 粘包半包、拆包问题怎么处理？

TCP是基于字节流传输的，只维护发送出去多少，确认了多少，没有维护消息与消息之间的边界，因而可能导致粘包问题。

TCP传输中，客户端发送数据，实际上把数据写入到了TCP缓存中，粘包和半包也就会在此产生。

其实从上面的定义，我们就可以大概知道产生的原因了。

粘包的主要原因：

* **发送方每次写入数据 < 套接字(Socket)缓冲区大小**
* **接收方读取套接字(Socket)缓冲区数据不够及时**

半包的主要原因：

* **发送方每次写入数据 > 套接字(Socket)缓冲区大小**
* **发送的数据大于协议的 MTU (Maximum Transmission Unit，最大传输单元)，因此必须拆包**

根本原因：其实就是TCP是流式协议，消息无边界

（PS:UDP虽然也可以一次传输多个包或者多次传输一个包，但每个消息都是有边界的，因此不会有粘包和半包问题）

归纳为：

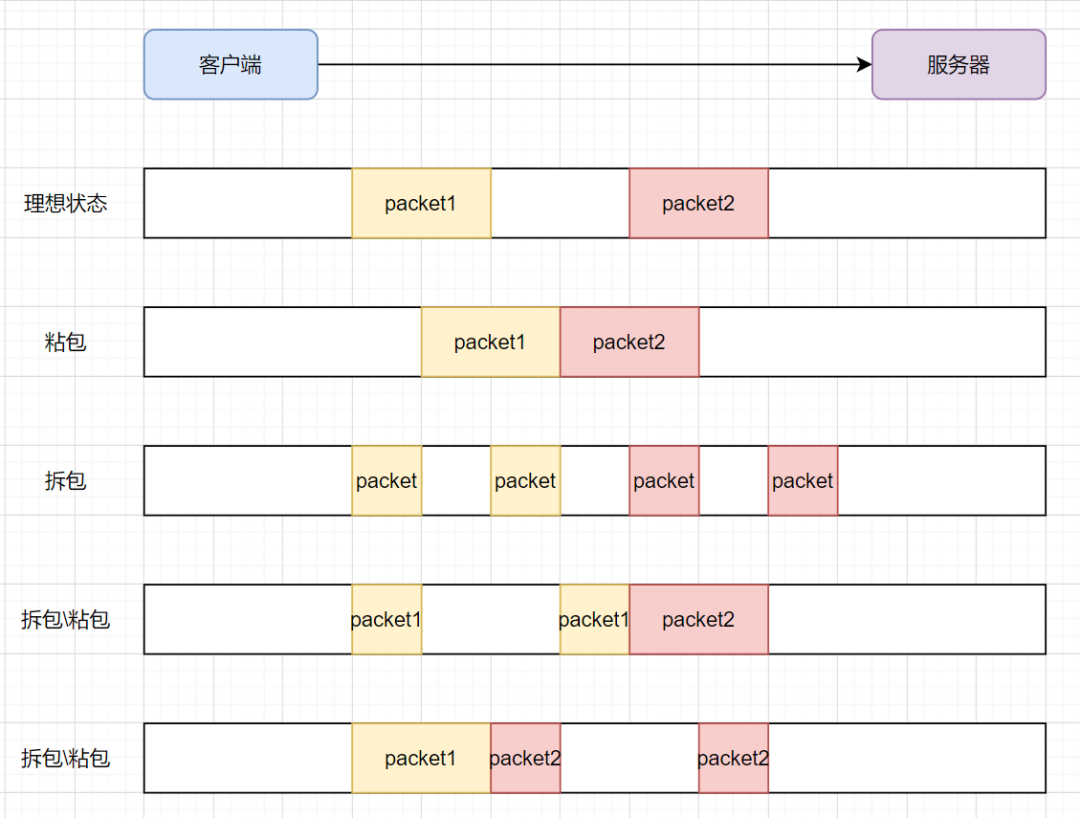
* 1. **消息定长**，例如每个报文大小固定200字节，如果不够，空位补空格
  2. **在包尾增加分隔符**，比如回车换行符进行分割，例如FTP协议：linebase包和delimiter包下，分别使用LineBasedFrameDecoder和DelimiterBasedFrameDecoder,如果超过规定字节长度，会报错
  3. **将消息分为消息头和消息体**，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思路为消息头的第一个字段使用int32来表示消息的总长度。

粘包问题本质上要在应用层维护信息与信息的边界。解决方案如下：

* 在接收端接收的时候采用定长的方式接收
* 在数据包尾添加一些分隔符
* 在数据包头部加上数据包长度
* 更复杂的应用层协议

### TCP的粘包和拆包

TCP是面向流，没有界限的一串数据。TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的实际情况进行包的划分，所以在业务上认为，一**个完整的包可能会被TCP拆分成多个包进行发送**，**也有可能把多个小的包封装成一个大的数据包发送**，这就是所谓的TCP粘包和拆包问题。

TCP的粘包和拆包

**为什么会产生粘包和拆包呢?**

* 要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包；
* 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包；
* 要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包；
* 待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。即TCP报文长度-TCP头部长度>MSS。

**解决方案：**

* 发送端将每个数据包封装为固定长度
* 在数据尾部增加特殊字符进行分割
* 将数据分为两部分，一部分是头部，一部分是内容体；其中头部结构大小固定，且有一个字段声明内容体的大小。

1. HTTP请求报文和响应格式？（HTTP请求头每个字段的作用）



起始行：

在请求报文中，起始行包括了3个部分：

* + - 1. 请求的方法
      2. 请求的URL（/cgi-bin/qqshow\_user\_props\_info）
      3. 协议类型及版本（HTTP/1.1）

请求方法：

http请求方法有八种，（其中比较常用的是GET、POST、HEAD）：

OPTIONS

返回服务器针对特定资源所支持的HTTP请求方法，也可以利用web服务器发送 ’\*’ 的请求来测试服务器的功能性。允许客户端查看服务器的性能。

**HEAD**

**向服务器与GET请求相一致的响应，只不过返回的响应中没有具体内容，用于获取报头。**

GET

向特定的资源发出请求。它本质就是发送一个请求来取得服务器上的某一资源。资源通过一组HTTP头和呈现数据（如HTML文本，或者图片或者视频等）返回给客户端。GET请求中，永远不会包含呈现数据。

POST

向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 Loadrunner中对应POST请求函数：web\_submit\_data,web\_submit\_form

PUT

向指定资源位置上传其最新内容。从客户端像服务器传送的数据取代指定文档的内容。

DELETE

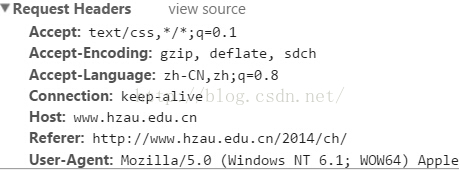
请求服务器删除Request-URL所标识的资源。请求服务器删除指定的页面。

TRACE

回显服务器收到的请求，主要用于测试或诊断

CONNECT

HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。



Accept

作用：浏览器端可以接受的媒体类型

例如：Accept：text/html 代表浏览器可以接受服务器回发的类型为text/html，也就是html文档。如果服务器无法返回text/html类型的数据，服务器就应该返回一个406错误

Accept-Encoding:

作用：浏览器申请自己接收的编码方法，通常指定压缩方法，是否支持压缩，支持扫码压缩方法

Accept-Language:

作用：浏览器申明自己接收的语言

语言跟字符集的区别：中文是语言，中文有多种字符集

Accept-Language:en-us

Connection:

例如：　Connection: keep-alive   当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接

例如：  Connection: close  代表一个Request完成后，客户端和服务器之间用于传输HTTP数据的TCP连接会关闭， 当客户端再次发送Request，需要重新建立TCP连接。

**Host**（发送请求时，该报头域是必需的）

作用: 请求报头域主要用于指定被请求资源的Internet主机和端口号，它通常从HTTP URL中提取出来的

例如: 我们在浏览器中输入：http://www.hzau.edu.cn

浏览器发送的请求消息中，就会包含Host请求报头域，如下：

Host：www.hzau.edu.cn

此处使用缺省端口号80，若指定了端口号，则变成：Host：指定端口号

Referer:

当浏览器向web服务器发送请求的时候，一般会带上Referer，告诉服务器我是从哪个页面链接过来的，服务器籍此可以获得一些信息用于处理。比如从我主页上链接到一个朋友那里，他的服务器就能够从HTTP Referer中统计出每天有多少用户点击我主页上的链接访问他的网站。

User-Agent:

作用：告诉HTTP服务器， 客户端使用的[操作系统](http://lib.csdn.net/base/operatingsystem)和浏览器的名称和版本.

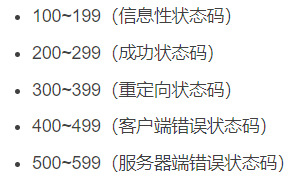
Cookie:

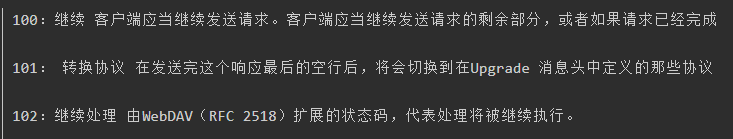
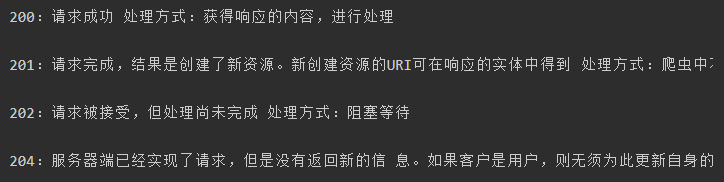
Cookie是用来存储一些用户信息以便让服务器辨别用户身份的（大多数需要登录的网站上面会比较常见），比如cookie会存储一些用户的用户名和密码，当用户登录后就会在客户端产生一个cookie来存储相关信息，这样浏览器通过读取cookie的信息去服务器上验证并通过后会判定你是合法用户，从而允许查看相应网页。当然cookie里面的数据不仅仅是上述范围，还有很多信息可以存储是cookie里面，比如sessionid等。

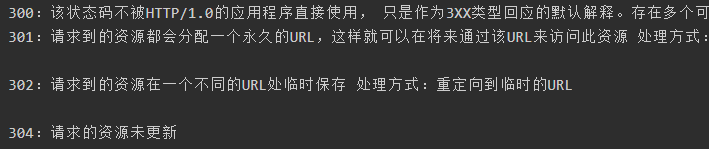
应答报文：

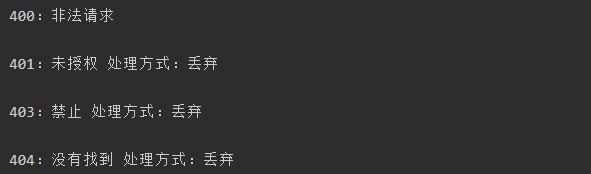
1. 协议类型及版本号
2. 状态码
3. 状态码文字描述

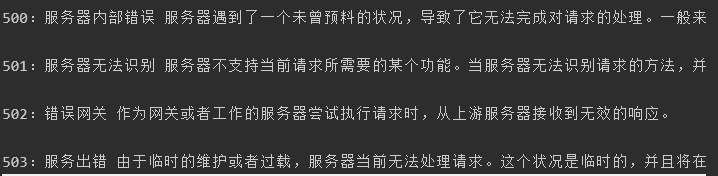
在**http**协议中，状态码被分为了5大类









1. keepalive 是什么东东？如何使用？

后来的HTTP/1.0中以及HTTP/1.1中，引入了重用连接的机制，就是在http请求头中加入Connection: keep-alive来告诉对方这个请求响应完成后不要关闭，下一次咱们还用这个请求继续交流。协议规定HTTP/1.0如果想要保持长连接，需要在请求头中加上Connection: keep-alive，而HTTP/1.1默认是支持长连接的，有没有这个请求头都行。

当然了，协议是这样规定的，至于支不支持还得看服务器（比如tomcat）和客户端（比如浏览器）的具体实现。在实践过程中发现谷歌浏览器使用HTTP/1.1协议时请求头中总会带上Connection: keep-alive，另外通过httpclient使用HTTP/1.0协议去请求tomcat时，即使带上Connection: keep-alive请求头也保持不了长连接。如果HTTP/1.1版本的http请求报文不希望使用长连接，则要在请求头中加上Connection: close，接收到这个请求头的对端服务就会主动关闭连接。

1. 流量控制，拥塞控制

流量控制：如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。

TCP的流量控制是利用滑动窗口机制实现的，接收方在返回的数据中会包含自己的接收窗口大小，以控制发送方的数据发送

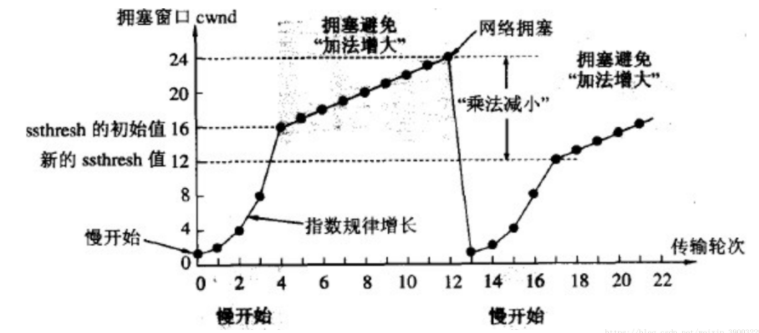
拥塞控制：拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。

两者的区别：流量控制为了预防拥塞。如：在马路上行车，交警跟红绿灯是流量控制，当发生拥塞时，如何进行疏散，是拥塞控制。流量控制指点对点通信量的控制。而拥塞控制是全局性的，涉及到所有主机和降低网络性能的因素。

拥塞解决的两种方法：

发送方控制拥塞窗口的原则是：主要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去，但只要网络出现拥塞，拥塞窗口就减小一些，以减少注入网络中的分组数。

慢开始+拥塞避免



一开始的慢开始算法指数增长很夸张，所以为了防止拥塞窗口增长过快需要设置一个门限值。无论再慢开始阶段还是在拥塞避免阶段，只要发送方没有收到确认，就认为这时候拥塞了

1. 就要把慢开始门限设置为当前窗口值的一半
2. 就要把拥塞窗口重新设置为拥塞窗口以最小值(即拥塞窗口和接受端窗口中的较小值)

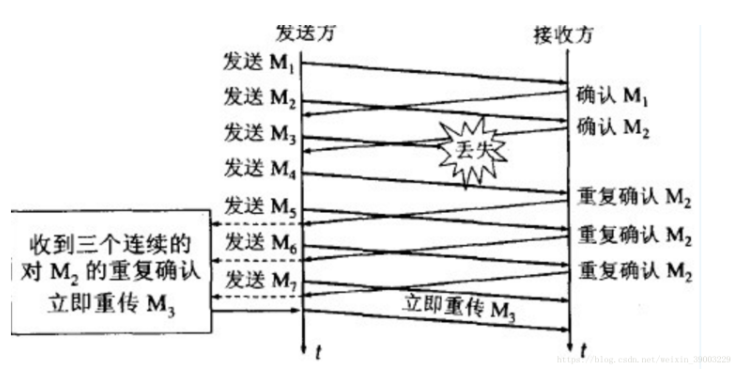
如何设置阈值：

慢启动的阈值和网络质量是有关系的

快重传和快恢复

快重传要求接收方在收到一个失序的报文段后就立即发送重复确认

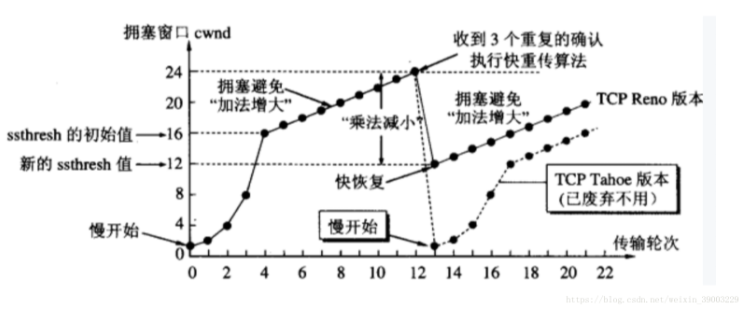
如果没有快速重传和快速恢复，TCP将会使用定时器来要求传输暂停。在暂停这段时间内，没有数据包被发送。所以快速重传和快速回复旨在快速恢复丢失的数据包。



快速重传机制：接收方发现M3丢失，则立即发送对M2的重复确认。一旦发送方一连收到三个M2的重复确认就应该立即重传M3，也就是发送方收到第四对M2的确认时。

与快重传配合使用的还有快恢复算法，结合上图的实例分析，其过程有一下两个要点。

1. 当发送方连续收到三个重复确认时，**就把门限减半**
2. 考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，**所以发送方认为网络可能没有出现拥塞**。**所以此时不执行慢开始算法**，**而实将拥塞窗口设置为门限大小，然后执行拥塞避免算法。**



拥塞控制主要是四个算法：1）慢启动，2）拥塞避免，3）拥塞发生，4）快速恢复

## 慢热启动算法 – Slow Start

所谓慢启动，也就是TCP连接刚建立，一点一点地提速，试探一下网络的承受能力，以免直接扰乱了网络通道的秩序。

慢启动算法：

1) 连接建好的开始先初始化拥塞窗口cwnd大小为1，表明可以传一个MSS大小的数据。 2) 每当收到一个ACK，cwnd大小加一，呈线性上升。 3) 每当过了一个往返延迟时间RTT(Round-Trip Time)，cwnd大小直接翻倍，乘以2，呈指数让升。 4) 还有一个ssthresh（slow start threshold），是一个上限，当cwnd >= ssthresh时，就会进入“拥塞避免算法”（后面会说这个算法）

## 拥塞避免算法 – Congestion Avoidance

如同前边说的，当拥塞窗口大小cwnd大于等于慢启动阈值ssthresh后，就进入拥塞避免算法。算法如下：

1) 收到一个ACK，则cwnd = cwnd + 1 / cwnd 2) 每当过了一个往返延迟时间RTT，cwnd大小加一。

过了慢启动阈值后，拥塞避免算法可以避免窗口增长过快导致窗口拥塞，而是缓慢的增加调整到网络的最佳值。

## 拥塞状态时的算法

一般来说，TCP拥塞控制默认认为网络丢包是由于网络拥塞导致的，所以一般的TCP拥塞控制算法以丢包为网络进入拥塞状态的信号。对于丢包有两种判定方式，一种是超时重传RTO[Retransmission Timeout]超时，另一个是收到三个重复确认ACK。

超时重传是TCP协议保证数据可靠性的一个重要机制，其原理是在发送一个数据以后就开启一个计时器，在一定时间内如果没有得到发送数据报的ACK报文，那么就重新发送数据，直到发送成功为止。

但是如果发送端接收到3个以上的重复ACK，TCP就意识到数据发生丢失，需要重传。这个机制不需要等到重传定时器超时，所以叫 做快速重传，而快速重传后没有使用慢启动算法，而是拥塞避免算法，所以这又叫做快速恢复算法。

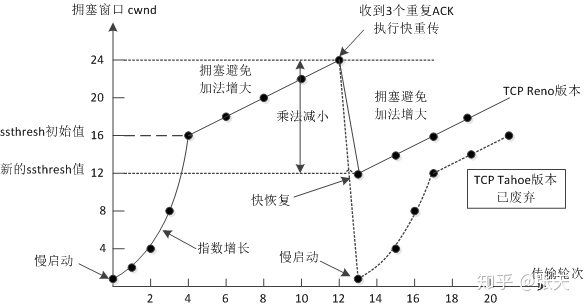
超时重传RTO[Retransmission Timeout]超时，TCP会重传数据包。TCP认为这种情况比较糟糕，反应也比较强烈：

* 由于发生丢包，将慢启动阈值ssthresh设置为当前cwnd的一半，即ssthresh = cwnd / 2.
* cwnd重置为1
* 进入慢启动过程

最为早期的TCP Tahoe算法就只使用上述处理办法，但是由于一丢包就一切重来，导致cwnd又重置为1，十分不利于网络数据的稳定传递。

所以，TCP Reno算法进行了优化。当收到三个重复确认ACK时，TCP开启快速重传Fast Retransmit算法，而不用等到RTO超时再进行重传：

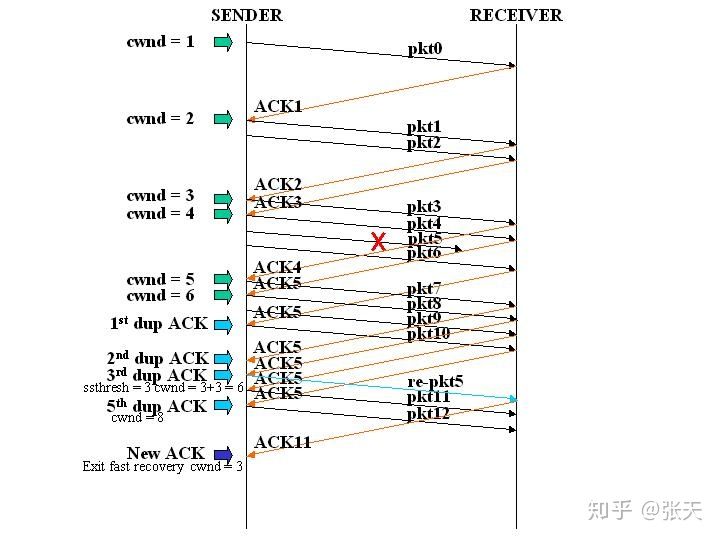
* cwnd大小缩小为当前的一半
* **ssthresh设置为缩小后的cwnd大小**
* 然后进入快速恢复算法Fast Recovery。



## 快速恢复算法 – Fast Recovery

TCP Tahoe是早期的算法，所以没有快速恢复算法，而Reno算法有。在进入快速恢复之前，cwnd和ssthresh已经被更改为原有cwnd的一半。快速恢复算法的逻辑如下：

* cwnd = cwnd + 3 \* MSS，加3 \* MSS的原因是因为收到3个重复的ACK。
* 重传DACKs指定的数据包。
* 如果再收到DACKs，那么cwnd大小增加一。
* 如果收到新的ACK，表明重传的包成功了，那么退出快速恢复算法。将cwnd设置为ssthresh，然后进入拥塞避免算法。



如图所示，第五个包发生了丢失，所以导致接收方接收到三次重复ACK，也就是ACK5。所以将ssthresh设置当当时cwnd的一半，也就是6/2 = 3，cwnd设置为3 + 3 = 6。然后重传第五个包。当收到新的ACK时，也就是ACK11，则退出快速恢复阶段，将cwnd重新设置为当前的ssthresh，也就是3，然后进入拥塞避免算法阶段。

1. TCP和UDP区别，TCP如何保证可靠性，对方是否存活(心跳检测)

TCP和UDP区别：

1.基于连接与无连接；  
2.对系统资源的要求（TCP较多，UDP少）；  
3.UDP程序结构较简单；  
4.**流模式与数据报**模式 ；

5.TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证。

tcp协议和udp协议的差别   
TCP UDP   
是否连接 面向连接 面向非连接   
传输可靠性 可靠 不可靠   
应用场合 少量数据 传输大量数据   
速度 慢 快

TCP如何保证可靠性：

* 首先，TCP的连接是基于三次握手，而断开则是四次挥手。确保连接和断开的可靠性。
* 其次，TCP的可靠性，还体现在有状态;TCP会记录哪些数据发送了，哪些数据被接受了，哪些没有被接受，并且保证数据包按序到达，保证数据传输不出差错。
* 再次，TCP的可靠性，还体现在可控制。它有**报文校验**、**ACK应答**、**超时重传**(发送方)、**失序数据重传**（接收方）、**丢弃重复数据**、**流量控制**（滑动窗口）和**拥塞控制**等机制。

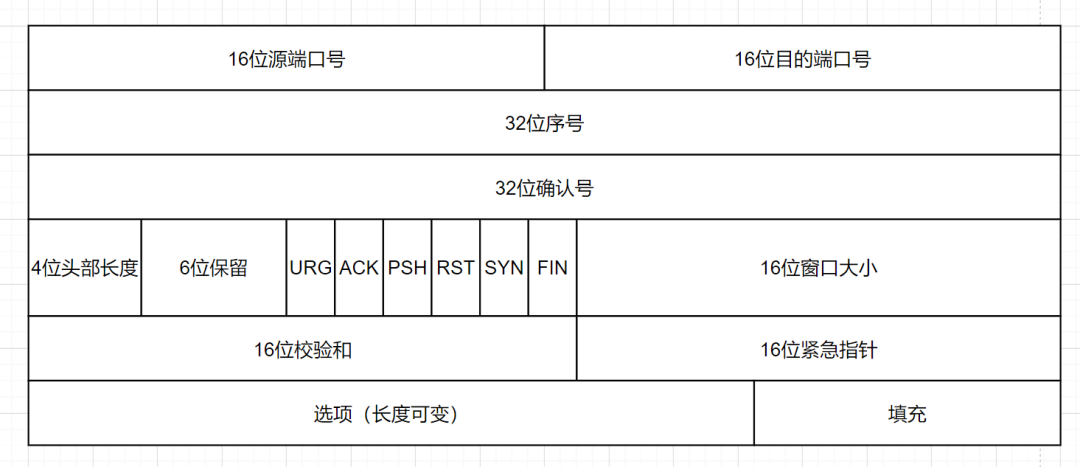
1. 校验和
2. 确认应答和序列号
3. 超时重传
4. 连接管理
5. 流量控制
6. 拥塞控制

心跳检测：判断对方（设备，进程或其它网元）是否正常动行，一般采用定时发送简单的通讯包，如果在指定时间段内未收到对方响应，则判断对方已经宕掉了。用于检测TCP的异常断开。  
  
基本原因是服务器端不能有效的判断客户端是否在线也就是说，服务器无法区分客户端是长时间在空闲，还是已经掉线的情况。所谓的心跳包就是客户端定时发送简单的信息给服务器端告诉它我还在而已。  
  
代码就是每隔几分钟发送一个固定信息给服务端，服务端收到后回复一个固定信息。如果服务端几分钟内没有收到客户端信息则视客户端断开。比如有些通信软件长时间不使用，要想知道它的状态是在线还是离线就需要心跳包，定时发包收包。

发包方可以是客户也可以是服务端，看哪边实现方便合理。一般是客户端。服务器也可以定时轮询发心跳下去。  
  
一般来说，出于效率的考虑，是由客户端主动向服务器端发包，而不是相反。

1. TCP报文格式

### TCP报文首部有哪些字段，说说其作用

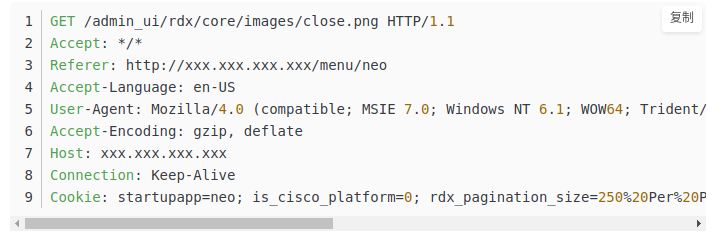


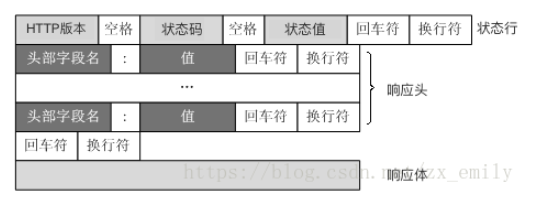
* **16位端口号**：源端口号，主机该报文段是来自哪里；目标端口号，要传给哪个上层协议或应用程序
* **32位序号**：一次TCP通信（从TCP连接建立到断开）过程中某一个传输方向上的字节流的每个字节的编号。
* **32位确认号**：用作对另一方发送的tcp报文段的响应。其值是收到的TCP报文段的序号值加1。
* **4位头部长度**：表示tcp头部有多少个32bit字（4字节）。因为4位最大能标识15，所以TCP头部最长是60字节。
* **6位标志位**：URG(紧急指针是否有效)，ACk（表示确认号是否有效），PSH（缓冲区尚未填满），**RST（表示要求对方重新建立连接）**，SYN（建立连接消息标志接），FIN（表示告知对方本端要关闭连接了）
* **16位窗口大小**：是TCP流量控制的一个手段。这里说的窗口，指的是接收通告窗口。它告诉对方本端的TCP接收缓冲区还能容纳多少字节的数据，这样对方就可以控制发送数据的速度。
* **16位校验和**：由发送端填充，接收端对TCP报文段执行CRC算法以检验TCP报文段在传输过程中是否损坏。注意，这个校验不仅包括TCP头部，也包括数据部分。这也是TCP可靠传输的一个重要保障。
* **16位紧急指针**：一个正的偏移量。它和序号字段的值相加表示最后一个紧急数据的下一字节的序号。因此，确切地说，这个字段是紧急指针相对当前序号的偏移，不妨称之为紧急偏移。TCP的紧急指针是发送端向接收端发送紧急数据的方法。

1. http报文格式

HTTP报文传过来都是一堆的0x ASCII码，诸如“41 63 63 65 70 74” 对应的是“accept” 单词的十六进制ASCII码。

以下是wireshark抓出来的一段HTTP请求报文

  
HTTP 的请求报文包括：请求行（request line）、请求头部（header）、空行和请求数据（request data）四个部分组成。



请求行包括：请求方法，URL(包括参数信息)，协议版本信息

请求头部(header)是一个个key-value值，比如

Accept-Encoding:gzip、deflate

User-Agent:Mozilla/4.0…

空行（CR+LF）: 请求报文用空行表示header和请求数据的分隔

请求数据：GET方法没有携带数据，POST方法会携带一个body

Response报文



HTTP的响应报文包括：状态行、响应头、空行，数据(响应体)



状态行包括：HTTP版本号，状态码和状态值组成

响应头类似请求头，是一系列key-value值

Cache-Control：private

Content-Encoding：gzip

空白行：同上，响应报文也用空白行来分隔header和数据

响应体：响应的data，本例是一段HTML

HTTP请求的一次完整过程

* + 1. 建立TCP连接(之前可能还有一次DNS域名解析)
    2. 三次握手建立TCP连接后，客户端向服务器发送请求命令，比如GET https://www.baidu.com?name=xx&addr=xx HTTP1.1
    3. 客户端发送请求头信息，发送完了header后会接着发送一个空白行，GET请求没有数据，post请求要发送body数据
    4. 服务器接收到以上信息后，开始处理业务，处理完有结果后，服务器开始应答
    5. 服务器返回响应头信息，发送response header以后，再发送一个空白行
    6. 然后服务器向客户端发送数据
    7. 发送完了服务器四次挥手关闭TCP连接

1. http中get和post的区别

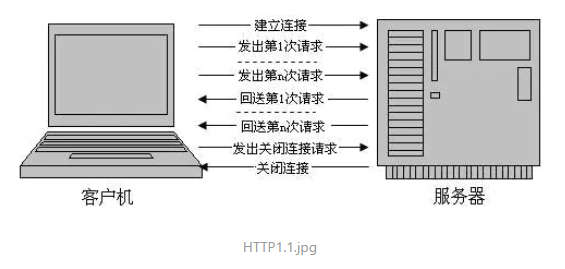


1. http1.0与http1.1区别，http2.0特性，http3.0特性
2. 长连接

HTTP/1.0协议使用非持久连接，即在非持久连接下，一个tcp连接只传输一个web对象。

HTTP/1.1支持持久连接，也就是说长连接，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。

一个包含有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还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分每次请求的相应内容，这样也显得减少了整个下载过程所需要的时间。基于HTTP1.1协议的客户机与服务器的信息交互过程。



HTTP1.0需要使用keep-alive参数来告知服务器要建立一个长连接，而HTTP1.1默认支持长连接

1. 影响一个HTTP网络请求的因素主要有两个：带宽和延迟。
2. 上述提到HTTP1.0和HTTP1.1的一些区别

当前使用最为广泛的HTTP协议，主要区别体现在：

* + - 1. 缓存处理，在HTTP1.0主要使用header里的if-modified-since，expires来作为缓存判断的标准，HTTP1.1则引入了更多缓存控制策略例如：Entity tag、if-unmodified-since，if-match，if-none-match等更多可供选择的缓存头控制缓存策略
      2. 带宽优化以及网路的使用
      3. 错误通知的管理
      4. Host头处理：在http1.0中认为每台服务器都绑定一个唯一的io地址，因此，请求消息中url没有传递主机名（hostname）。但是随着虚拟主机发展，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应支持host头域，且请求消息中如果没有host头域会报告一个错误（400 BAD request）
      5. 长连接

https://www.jianshu.com/p/be29d679cbff

HTTP 2.0（二进制分帧、首部压缩、多路复用、请求优先级、服务器推送）

HTTP2.0大幅度的提高了web性能，在HTTP1.1完全语义兼容的基础上，进一步减少了网络的延迟。实现低延迟高吞吐量。

HTTP/2是HTTP协议自1999年HTTP1.1发布后的首个更新，主要基于SPDY协议。

什么是SPDY协议？

SPDY是Speedy的昵音，意为“更快”。它是Google开发的**基于TCP协议的应用层协议**。目标是优化HTTP协议的性能，**通过压缩、多路复用和优先级**等技术，缩短网页的加载时间并提高安全性。SPDY协议的核心思想是**尽量减少TCP连接数**。SPDY并不是一种用于替代HTTP的协议，而是对HTTP协议的增强。

HTTP2.0和SPDY的区别：

HTTP2.0 支持明文 HTTP 传输，而 SPDY 强制使用 HTTPS。与压缩算法的区别。

HTTP 1.x有主要一下缺点：

HTTP/1.0一次只允许在一个TCP连接上发起一个请求，HTTP/1.1使用的流水线技术也只能部分处理请求并发，仍然会**存在队列头阻塞**问题，因此客户端在需要发起多次请求时，**通常会采用建立多连接来减少延迟**。

单向请求，只能由客户端发起

请求报文与响应报文首部**信息冗余量大**

**数据未压缩，导致数据的传输量大**

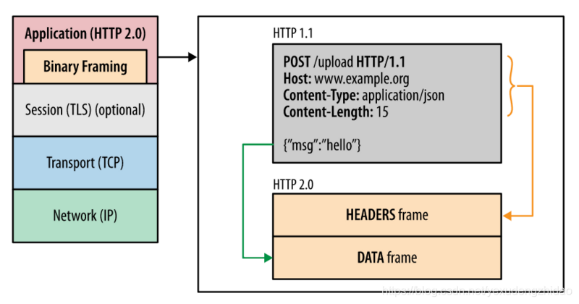
HTTP2.0如何突破HTTP1.1的性能限制，改进传输性能，实现低延迟高吞吐量呢？

一、 **二进制分帧**

关键之一就是在应用层（HTTP）和传输层（TCP）之间**增加一个二进制分帧层。**

* 帧：HTTP2.0通信的最小单位，所有帧都共享一个8字节的首部，其中包含帧的长度、类型、标志、还有一个保留位，并且至少有标识出当前帧所属的流的标识符，帧承载着特定类型的数据，如HTTP首部、负荷、等等。
* 消息：比帧大的通讯单位，是指逻辑上的HTTP消息，比如请求、响应等。由一个或多个帧组成
* 流：比消息大的通讯单位。是TCP连接中的一个虚拟通道，可以承载双向的消息。每个流都有一个**唯一的整数标识符**

HTTP2.0中所有**加强性能的核心是二进制传输**，在HTTP1.x中，我们是通过文本的方式传输数据。基于文本的方式传输数据存在很多缺陷，文本的表现形式有多样性，因此要做到健壮性考虑的场景必然有限，但是**二进制则不同，只有0和1的组合，因此选择了二进制传输，实现方便且健壮**。  
在HTTP2.0中引入了新的编码机制，所有传输的数据都会被分割，并采用二进制格式编码。



为了保证HTTP不受影响，那就需要在应用层（HTTP2.0）和传输层（TCP or UDP）之间增加一个二进制分帧层。**在二进制分帧层上，HTTP2.0会将所有传输的信息分为更小的消息和帧**，并采用二进制格式编码，其中HTTP1.x的**首部信息会被封装到Headers帧**，**而Request Body则封装到Data帧**。

二、**首部压缩**

HTTP1.1并不支持HTTP首部压缩，为此SPDY和HTTP2.0出现了。SPDY是用的是DEFLATE算法（15年 Google 宣布移除对 SPDY 的支持，压缩基于霍夫曼编码），而HTTP2.0则使用了专门为首部压缩设计的HPACK算法。

HTTP每次通讯（请求或响应）都会携带首部信息用于描述资源属性。

在HTTP1.0中，我们使用文本的形式传输header，在header中携带cookie的话，每次都需要重复传输几百到几千的字节，这着实是一笔不小的开销。

在HTTP2.0中，我们使用了HPACK（HTTP2头部压缩算法）压缩格式对传输的header进行编码，减少了header的大小。并在**两端维护了索引表，用于记录出现过的header，后面在传输过程中就可以传输已经记录过的header的键名，对端收到数据后就可以通过键名找到对应的值。**

三、**多路复用**

同个域名只需要占用一个 TCP 连接，使用一个连接并行发送多个请求和响应,这样整个页面资源的下载过程只需要一次慢启动，**同时也避免了多个TCP连接竞争带宽所带来的问题**。

在HTTP1.x中，我们经常会使用到雪碧图、使用多个域名等方式来进行优化，都是因为浏览器限制了同一个域名下的请求数量，当页面需要请求很多资源的时候，队头阻塞（Head of line blocking）会导致在达到最大请求时，资源需要等待其他资源请求完成后才能继续发送。

HTTP2.0中,基于二进制分帧层，HTTP2.0可以在共享TCP连接的基础上同时发送请求和响应。HTTP消息**被分解为独立的帧**，而不破坏消息本身的语义，**交错发出去**，在另一端根据流标识符和首部将他们重新组装起来。 通过该技术，可以避免HTTP旧版本的队头阻塞问题，极大提高传输性能

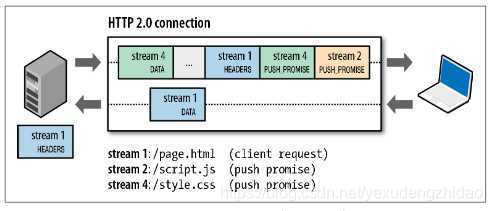
四、请求优先级

把HTTP消息分为很多独立帧之后，就可以通过优化这些帧的交错和传输顺序进一步优化性能。

五、服务器推送

HTTP2.0新增的一个强大的新功能，就是服务器可以对一个客户端请求发送多个响应。**服务器向客户端推送资源无需客户端明确的请求**。

服务端根据客户端的请求，提前返回多个响应，推送额外的资源给客户端。如下图，客户端请求stream 1(/page.html)。服务端在返回stream 1的消息的同时推送了stream 2(/script.js)和stream 4(/style.css)



服务端推送是一种在客户端请求之前发送数据的机制。在HTTP2.0中，服务器可以对一个客户端的请求发送多个响应。如果一个请求是由你的主页发送的，服务器可能会响应主页内容、logo以及样式表，因为他知道客户端会用到这些东西。这样不但**减轻了数据传送冗余步骤**，也**加快了页面响应的速度**，提高了用户体验。

主流的浏览器Chrome、Firefox等都公开宣布只支持加密的HTTP/2，**所以“事实上”的HTTP/2是加密的**。

HTTP 3.0

HTTP2.0的缺点

虽然 HTTP/2 解决了很多之前旧版本的问题，但是它还是存在一个巨大的问题，**主要是底层支撑的 TCP** 协议造成的。HTTP/2的缺点主要有以下几点：

* + - 1. TCP 以及 TCP+TLS建立连接的延时
      2. TCP的队头阻塞并没有彻底解决

在HTTP/2中，多个请求是跑在一个TCP管道中的。但当出现了丢包时，HTTP/2 的表现反倒不如 HTTP/1 了。**因为TCP为了保证可靠传输，有个特别的“丢包重传”机制，丢失的包必须要等待重新传输确认，HTTP/2出现丢包时，整个 TCP 都要开始等待重传，那么就会阻塞该TCP连接中的所有请求。而对于 HTTP/1.1 来说，可以开启多个 TCP 连接，出现这种情况反到只会影响其中一个连接，剩余的 TCP 连接还可以正常传输数据**。

又是Google又是Google，推完SPDY协议又来QUIC协议完美解决队头阻塞问题

HTTP/3简介：

Google 在推SPDY的时候就已经意识到了这些问题，于是就另起炉灶搞了一个基于 UDP 协议的“QUIC”协议，让HTTP跑在QUIC上而不是TCP上。而这个“HTTP over QUIC”就是HTTP协议的下一个大版本，HTTP/3。它在HTTP/2的基础上又实现了质的飞跃，真正“完美”地解决了“队头阻塞”问题。

QUIC新功能：

上面我们提到QUIC基于UDP，而UDP是“无连接”的，根本就不需要“握手”和“挥手”，所以就比TCP来得快。此外QUIC也实现了可靠传输，保证数据一定能够抵达目的地。它还引入了类似HTTP/2的“流”和“多路复用”，单个“流"是有序的，可能会因为丢包而阻塞，但其他“流”不会受到影响。具体来说QUIC协议有以下特点：

* 实现了类似TCP的**流量控制、传输可靠性**的功能。

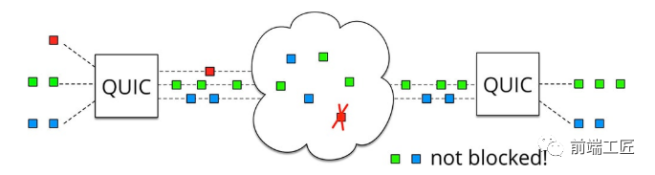
虽然UDP不提供可靠性的传输，但QUIC**在UDP的基础之上增加了一层来保证数据可靠性传输。它提供了数据包重传、拥塞控制以及其他一些TCP**中存在的特性。

* 实现了快速握手功能。

由于QUIC是基于UDP的，所以QUIC可以实现使用0-RTT或者1-RTT来建立连接，这意味着QUIC可以用最快的速度来发送和接收数据，这样可以大大提升首次打开页面的速度。**0RTT 建连可以说是 QUIC 相比 HTTP2 最大的性**能优势。

* 集成了TLS加密功能。
* **多路复用，彻底解决了TCP中队头阻塞问题。**

和TCP不同，QUIC实现了在同一物理连接上可以有**多个独立的逻辑数据流**（如下图）。实现了**数据流的单独传输**，就解决了TCP中队头阻塞的问题。



HTTP/1.1有两个主要的缺点：安全不足和性能不高。

HTTP/2完全兼容HTTP/1，是“更安全的HTTP、更快的HTTPS"，头部压缩、多路复用等技术可以充分利用带宽，降低延迟，从而大幅度提高上网体验；

QUIC 基于 UDP 实现，是 HTTP/3 中的底层支撑协议，该协议基于 UDP，又取了 TCP 中的精华，实现了即快又可靠的协议。

1. Client和server建立tcp长连接，第一种情况服务器直接挂了，第二种情况服务器进程挂了，两种情况下client端会做出什么反应
2. 孤儿进程和守护进程

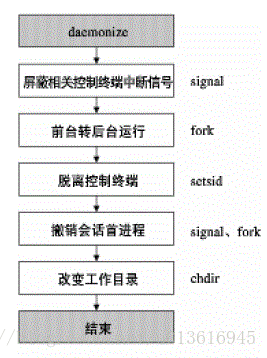
什么是孤儿进程？

如果一个子进程的父进程先于子进程结束，子进程就成为一个孤儿进程 ，它有init进程收养，成为init进程的子进程。危害不大。

代码实现<https://blog.csdn.net/believe_s/article/details/77040494>

什么是守护进程？

守护进程(daemon)，是一种运行在后台 的特殊进程，它独立于控制终端 ，并 周期性地执行某项任务或等待处理某些发生的事件。 。守护进程可以由一个普通的进程按照守护进程的特性改造而来。基本的流程如下：



守护进程是在后台运行不受终端控制的进程（如输入、输出等），一般的网络服务都是以守护进程的方式运行。守护进程脱离终端的主要原因有两点：（1）用来启动守护进程的终端在启动守护进程之后，需要执行其他任务。（2）（如其他用户登录该终端后，以前的守护进程的错误信息不应出现）由终端上的一些键所产生的信号（如中断信号），不应对以前从该终端上启动的任何守护进程造成影响。**要注意守护进程与后台运行程序（即加＆启动的程序）**的区别。

个人理解守护进程和普通进程区别是指：**将后台程序变成一种服务，比如说，用命令行输入启动程序，如果不是守护进程的话，一旦命令行窗口关闭，程序就终止了；而如果启动守护进程，则退出命令行窗口之后，服务一直处于运行状态**。

1. 僵尸进程，介绍一下僵尸进程，怎么避免僵尸进程

僵尸进程的产生：

当一个进程创建了一个子进程时，他们的运行是异步的。即父进程无法预知子进程是在什么时候结束，那么如果父进程很繁忙来不及wait子进程时，那么当子进程结束时，会不会丢失子进程的结束时的状态信息呢？

在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存。但是仍然保留了一些信息（如进程号pid 退出状态 运行时间等）。这些保留的信息直到进程通过调用wait/waitpid时才会释放。这样就导致了一个问题，如果没有调用wait/waitpid的话，那么保留的信息就不会释放。**比如进程号就会被一直占用了**。但系统所能使用的进程号的有限的，如果产生大量的僵尸进程，将导致系统没有可用的进程号而导致系统不能创建进程。所以我们应该避免僵尸进程

如果子进程先结束而父进程后结束，即子进程结束后，父进程还在继续运行但是并未调用wait/waitpid那子进程就会成为僵尸进程。

同样的在产生僵尸进程的那种情况下，即子进程结束了但父进程还在继续运行（并未调用wait/waitpid）这段期间，假如父进程异常终止了，那么该子进程就会自动被init接管。那么它就不再是僵尸进程了。应为init会发现并释放它所占有的资源。（当然如果进程表越大，init发现它接管僵尸进程这个过程就会变得越慢，所以在init为发现他们之前，僵尸进程依旧消耗着系统的资源）

避免僵尸进程的产生：

如果父进程不是很繁忙我们就可以通过直接调用wait/waitpid来等待子进程的结束。当然这会导致父进程被挂起。

但是如果父进程很忙。不希望父进程一直被挂起直到子进程结束

那么我们可以使用信号函数sigaction位SIGCHLD设置wait处理函数。这样子进程结束后，父进程就会收到子进程结束的信号。并调用wait回收子进程的资源。

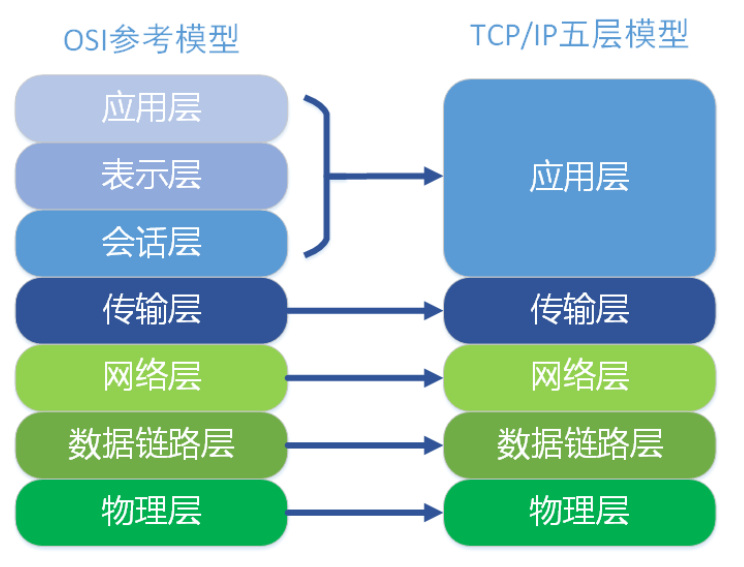
1. 网络层、数据链路层、传输层的设备有哪些？

网络层：路由器

数据链路层：网桥

传输层：网关

1. OSI七层模型及其设备，五层模型、四层模型



1. 网络层、传输层协议有哪些

**传输层协议：**

* + - 1. **传输控制协议TCP**
      2. **用户数据报协议UDP**
      3. **UGP**

**数据链路层：**

**ARP**

**RARP**

**网络层协议：**

* + - 1. **IP**
      2. **ICMP**
      3. **IGMP**

**应用层协议：**

**Telnet**

**FTP**

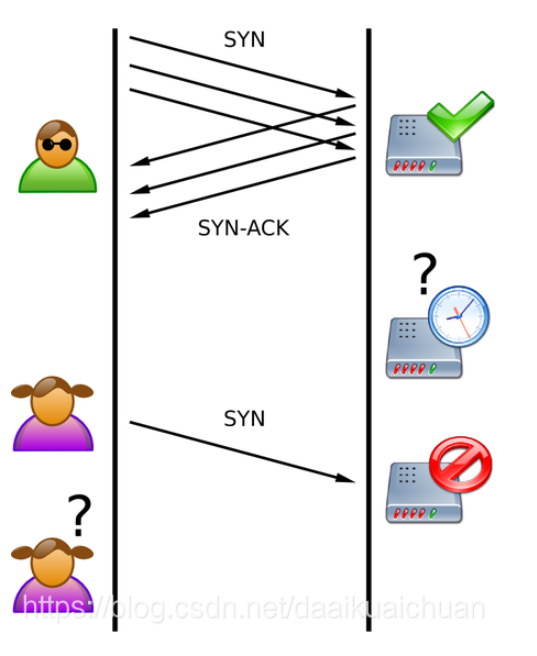
**SMTP**

**SNMP**

**HTTP**

1. tcp如何设定超时时间
2. SYN Flood攻击

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连接称为半连接状态。此时服务器处于SYN\_RCVD状态。当收到ACK后，服务器转入ESTABLISHED状态



Syn攻击就是攻击客户端在短时间内伪造大量不存在的IP地址，向服务器不断发送syn包，服务器回复确认包，并等待客户的确认，由于源地址不存在的，服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，目标系统运行缓慢，严重引起网络堵塞甚至系统瘫痪

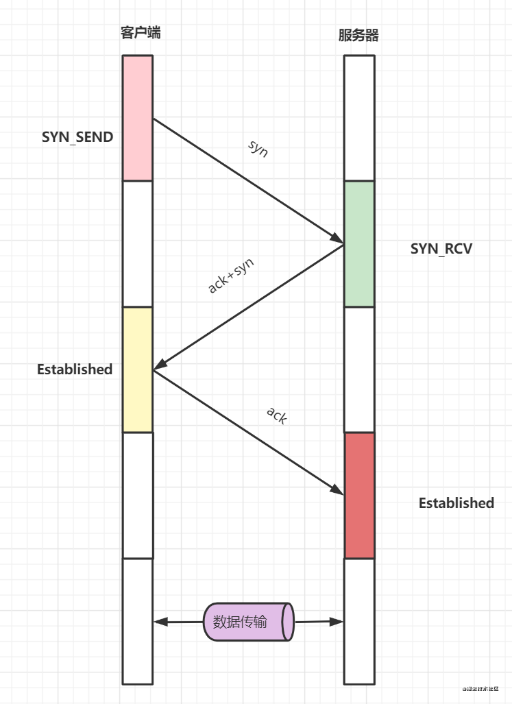
一般较新的TCP/IP协议栈都对这一过程进行修正来防范Syn攻击，修改tcp协议实现。

主要方法由**SynAttackProtect**保护机制、**SYN cookies**技术、**增大最大半连接**和**缩短超时时间**等。但是不能完全防范syn攻击。

### 半连接队列和 SYN Flood 攻击的关系

TCP进入三次握手前，服务端会从**CLOSED**状态变为**LISTEN**状态,同时在内部创建了两个队列：半连接队列（SYN队列）和全连接队列（ACCEPT队列）。

什么是**半连接队列（SYN队列）** 呢? 什么是**全连接队列（ACCEPT队列）** 呢？回忆下TCP三次握手的图：

三次握手

* TCP三次握手时，客户端发送SYN到服务端，服务端收到之后，便回复**ACK和SYN**，状态由**LISTEN变为SYN\_RCVD**，此时这个连接就被推入了**SYN队列**，即半连接队列。
* 当客户端回复ACK, 服务端接收后，三次握手就完成了。这时连接会等待被具体的应用取走，在被取走之前，它被推入ACCEPT队列，即全连接队列。

SYN Flood是一种典型的DoS (Denial of Service，拒绝服务) 攻击，它在短时间内，伪造**不存在的IP地址**,向服务器大量发起SYN报文。当服务器回复SYN+ACK报文后，不会收到ACK回应报文，导致服务器上建立大量的半连接半连接队列满了，这就无法处理正常的TCP请求啦。

主要有 **syn cookie**和**SYN Proxy防火墙**等方案应对。

* **syn cookie**：在收到SYN包后，服务器根据一定的方法，以数据包的源地址、端口等信息为参数计算出一个cookie值作为自己的SYNACK包的序列号，回复SYN+ACK后，服务器并不立即分配资源进行处理，等收到发送方的ACK包后，重新根据数据包的源地址、端口计算该包中的确认序列号是否正确，如果正确则建立连接，否则丢弃该包。
* **SYN Proxy防火墙**：服务器防火墙会对收到的每一个SYN报文进行代理和回应，并保持半连接。等发送方将ACK包返回后，再重新构造SYN包发到服务器，建立真正的TCP连接。

1. Nagle 算法与延迟确认

#### Nagle算法

如果发送端疯狂地向接收端发送很小的包，比如就1个字节，那么亲爱的小伙伴，你们觉得会有什么问题呢？

**★**

TCP/IP协议中，无论发送多少数据，总是要在数据前面加上协议头，同时，对方接收到数据，也需要发送ACK表示确认。为了尽可能的利用网络带宽，TCP总是希望尽可能的发送足够大的数据。**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算法的实现规则：

* **如果包长度达到MSS，则允许发送；**
* **如果该包含有FIN，则允许发送；**
* **设置了TCP\_NODELAY选项，则允许发送；**
* **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；**
* **上述条件都未满足，但发生了超时（一般为200ms），则立即发送。**

#### 延迟确认

如果接受方刚接收到发送方的数据包，在很短很短的时间内，又接收到第二个包。那么请问接收方是一个一个地回复好点，还是合并一起回复好呢？

**★**

接收方收到数据包后，如果暂时没有数据要发给对端，它可以等一段时再确认（Linux上默认是40ms）。如果这段时间刚好有数据要传给对端，ACK就随着数据传输，而不需要单独发送一次ACK。如果超过时间还没有数据要发送，也发送ACK，避免对端以为丢包。

”

但是有些场景不能延迟确认，比如发现了**乱序包**、**接收到了大于一个 frame 的报文，且需要调整窗口大小**等。

一般情况下，**Nagle算法和延迟确认**不能一起使用，Nagle算法意味着延迟发，**延迟确认**意味着延迟接收，酱紫就会造成更大的延迟，会产生性能问题。

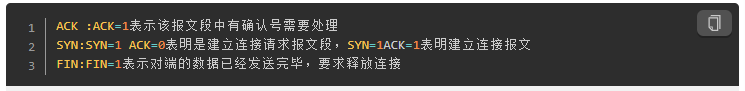
1. [从输入URL到页面显示的过程](https://www.jianshu.com/p/d9361badf49a)（延伸TCP和UDP的区别）
   * 1. 发送URL，请求IP地址

当发送一个URL请求时，不管这个URL是web页面的url还是web页面上每个资源的url，浏览器都会开启一个线程来处理这个请求，同时在远程DNS服务器上启动一个DNS查询，让浏览器获得请求对应的IP地址

* + 1. TCP三次握手

浏览器和远程WEB服务器通过TCP三次握手来协商建立一个TCP/IP连接。该握手包括一个同步报文、一个同步-应答报文、一个应答报文。这三个报文在浏览器和服务器之间传递。该握手首先由客户端尝试建立通信，然后服务端响应回答并接受客户端的请求，最后由客户端发出该请求已经被接受的报文。

再度复习TCP三次握手：



第一次握手：建立连接

客户端发送链接请求报文段，将SYN值设置为1，Sequence Number设为初始化序列号。客户端进入SYN\_SENT状态，等待服务器的确认

第二次握手：服务器收到SYN报文段

服务器收到客户端SYN报文段，需要对这个SYN报文段进行确认，设置ACK = 初始化序号+1，同时自己还要发送SYN请求信息，将SYN值设位1，并再次随机初始化序号y(服务端的初始化序列号)，服务器将上述信息放到一个报文段（即SYN+ACK报文段）中，一并发送给客户端，服务器进入SYN\_RECV装填。

第三次握手：客户端收到SYN+ACK报文段

客户端收到服务端的SYN+ACK报文段后将ACK设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕之后，客户端和服务器都进入ESTABLISHED状态，完成TCP的三次握手

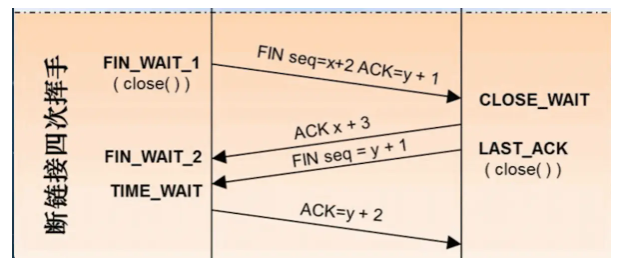
* + 1. 服务器响应200

TCP/IP连接建议后，浏览器通过该连接向远程服务器发送HTTP的GET请求，远程服务器找到资源并使用HTTP响应返回该资源，值为200的HTTP响应状态表示一个正确的响应

* + 1. 生成Render Tree

客户端下载完资源后，便进入了我们关注的前端模块了。浏览器会解析HTML成树形的数据结构DOM，生成DOM Tree，浏览器将css代码解析成树形的数据结构CSSOM，生成CSS Rule Tree。DOM Tree和CSS Rule Tree结合生成Render Tree

* + 1. 渲染页面



延申：TCP四次挥手（复习）

第一次挥手：客户端申请挥手

假设客户端想关闭连接，客户端会发送一个FIN标志位置为1的包(FIN=1,clientseq = x随机初始化序列)，表示自己已经没有数据可以发送了，但是可以接受数据，进入FIN\_WAIT\_1状态

第二次挥手：服务器想分手

服务器端确认客户端的FIN包，发送一个确认包（ACK=1，ACKnum = clientseq+1），表示自己已经接收到了客户端关闭连接的请求名单上还没准备好关闭连接。发送包后，服务器进入CLOSE\_WAIT状态，客户端收到确认包后，进入FIN\_WAIT\_2状态，等待服务器关闭连接

第三次挥手：服务器准备好分手

服务器准备好关闭连接后，向客户端发送结束请求，FIN置为1，（FIN=1，serverseq=y），发送完毕后进入LAST\_ACK状态，等待客户端的最后一个ACK

第四次挥手：分手

客户端接收来自服务器端的请求后，发送一个确认包（ACK=1,acknum = serverseq+1）进入TIME\_WAIT状态，等待可能出现的要求重传的ACK包。

服务器端接受到这个确认包之后，关闭连接，进入CLOSED状态

客户端等待2MSL之后，没有收到回复，确保服务器端确实为关闭了，客户端也关闭连接，进入CLOSED状态。

MSL是Maximum Segment Lifetime英文的缩写，中文可以译为“报文最大生存时间”，他是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

1. TCP的time\_wait状态出现在什么时候？有什么作用？

客户端收到了服务器端的FIN报文后进入此状态，此时并不是直接进入CLOSED状态，还需要等待一个时间计时器设置的时间（一般为2MSL）。这样做有两个理由：

* + - 1. **确保最后一个确认报文段能够到达。如果B没收到A发送来的确认报文段，那么就会重新发送连接释放请求报文段，A等待一段时间就是为了处理这种情况的发送**

因为 client 可能面临重发最终 ACK 的情形。先调用 close() 的一方会进入 TIME\_WAIT 状态

* + - 1. 可能存在“已失效的连接请求报文段”，为了防止这种报文段出现在本次连接之外，需要等待一段时间

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：

* + - 1. 可靠地进行TCP全双工连接的中止
      2. 允许老的重复分节在网络中消逝

1. TCP具有超时重传策略？如果一直超时怎么办？如何解决？

我们需要一个方法来[猜测]是否发生了丢包，最简单的想法就是，接收方每收到一个包，就向发送方返回一个 ACK，表示自己已经收到了这段数据，反过来，如果发送方一段时间内没有收到 ACK，就知道很可能是数据包丢失了，紧接着就重发该数据包，直到收到 ACK 为止。

你可能注意到我用的是「猜测」，因为即使是超时了，这个数据包也可能并没有丢，它只是绕了一条远路，来的很晚而已。毕竟 TCP 协议是位于传输层的协议，不可能明确知道数据链路层和物理层发生了什么。但这并不妨碍我们的超时重传机制，因为接收方会自动忽略重复的包。

超时是怎么确定的？

* + - 1. 将超时时间设置为固定值（不可取）

我的个人博客搭在国内，延迟大概 30ms，也就是说正常情况下的数据包，60ms 左右就已经能收到 ACK 了，但是按照我们的方法，200ms 才能确定丢包（正常可能是 90 到 120 ms），这效率实在是有点低。

假设你访问某国外网站，延迟有 130 ms，这就麻烦了，正常的数据包都可能被认为是超时，导致大量数据包被重发，可以想象，重发的数据包也很容易被误判为超时。。。雪崩效应的感觉

* + - 1. 根据网络延迟，动态调整超时时间，延迟越大，超时时间越长。因为网络流量和路由器在包的传输过程中可能改变，因此RTT也会变化。

引入概念：

RTT（Round Trip Time）:往返时延，也就是数据包从发出去到收到对应的ACK的时间。RTT是针对连接的，每一个连接都有各自独立返回的RTT。

**RTO(Retransmission Time Out)：**重传超时时间，即从数据发送时刻算起，超过这个时间便执行重传。

**缺点：**

当一个报文段丢失时，会等待一定的超时周期然后才重传分组，增加了端到端的时延。

当一个报文段丢失时，在其等待超时的过程中，可能会出现这种情况：其后的报文段已经被接收端接收但却迟迟得不到确认，发送端会认为也丢失了，从而引起不必要的重传，既浪费资源也浪费时间。

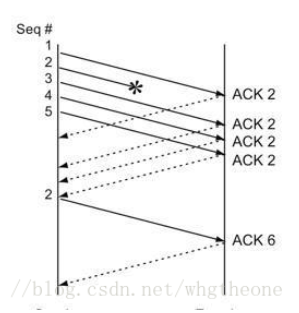
**如果一直超时怎么办：**

一直到75s（超时）还没有收到任何回应，**便会放弃，终止连接的创建**。但是在Linux实现中，并不是依靠超时总时间来判断是否终止连接。**而是依赖重传次数**：

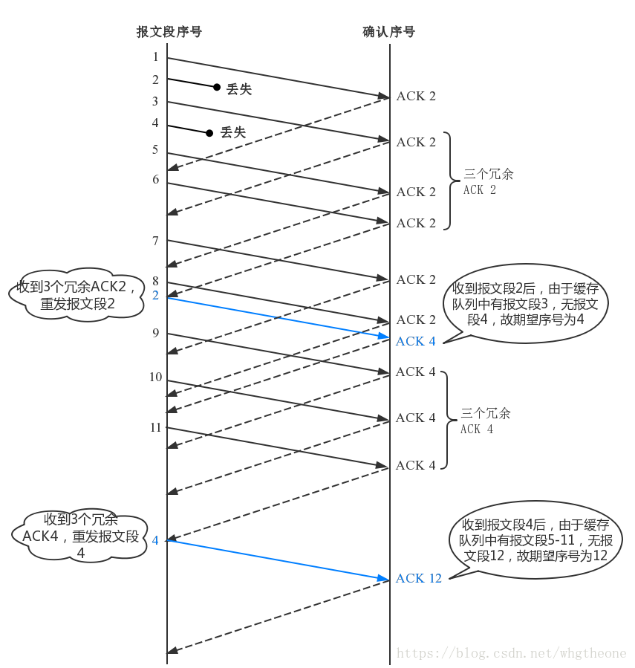
1. TCP的快速重传机制？什么时候发生重传？重传哪些数据包？

幸运的是，由于TCP采用的是累计确认机制，即当接收端收到比期望序号大的报文段时，便会重复发送最近一次确认的报文段的确认信号，我们称之为冗余ACK（duplicate ACK）。

如图所示，报文段1成功接收并被确认ACK 2，接收端的期待序号为2，当报文段2丢失，报文段3失序到来，与接收端的期望不匹配，接收端重复发送冗余ACK 2。



如果在超时重传定时器溢出之前，接收到连续的三个重复冗余ACK（其实是收到4个同样的ACK，第一个是正常的，后三个才是冗余的），发送端便知晓哪个报文段在传输过程中丢失了，于是重发该报文段，不需要等待超时重传定时器溢出，大大提高了效率。这便是快速重传机制。



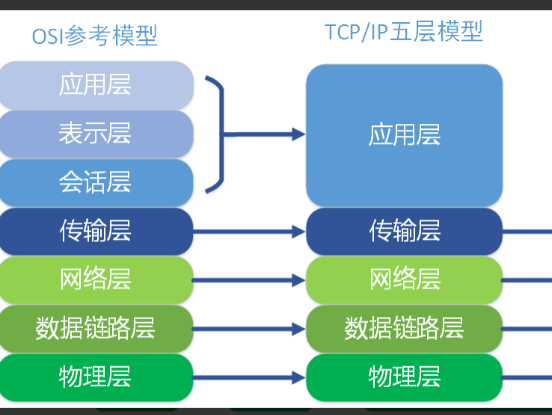
1. IO多路复用如何实现的？epoll相对于select有什么优点？epoll的垂直和水平触发方式？
2. UDP最大包长多少？首部的字段和长度？为什么是1500字节呢？

1500-8-20 = 1472字节

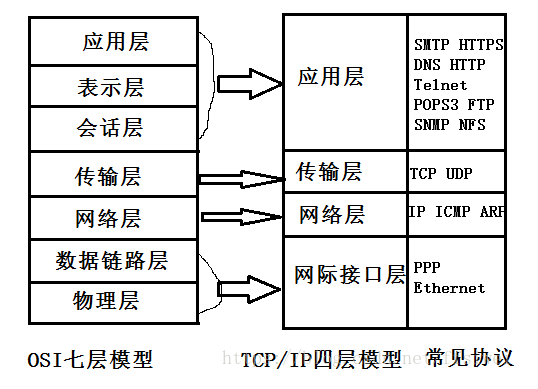
1. 包丢了，tcp的发送端措施，出了重传机制，还有别的吗？

假设TCP在内部使用ack机制来检测丢失的数据，并自动重新发送它们。因此，丢失的数据对用户来说时透明的，并由协议处理。

1. Tcp/ip的5层，osi的七层



1. 说一下osi七层协议，强调可靠性问题
2. http在哪一层，tcp在哪一层，arp在哪一层



1. 路由表的建表过程，如何根据网络的变化进行更新

路由表的构成：

•网络号

•下一跳地址

•接口

•Metric：跳数、延迟、费用



* + - 1. 静态路由：由网络管理员设置并随时更新。

缺点：网络管理员的工作负担重，容易出错，无法根据网络状态，进行调整。

优点：简单、开销小，只适用于小型网络

* + - 1. 动态路由：路由器运行过程中根据网络情况动态地维护减轻了网络管理员的工作负担重

缺点：实时性好，适应性好。能够满足大型网络的需要

缺点：因要搜集网络运行状态，网络开销有所增加，实现也比较复杂

路由器在刚刚开始工作时，**只知道到直接连接的网络 的距离（此距离定义为1）。**

以后，每一个路由器也只和数目非常有限的相邻路由 器交换并更新路由信息。

经过若干次更新后，所有的路由器最终都会知道到达本自治系统中任何一个网络的最短距离和下一跳路由器的地址。

RIP 协议的收敛(convergence)过程较快，即在自治系统中所有的结点都得到正确的路由选择信息的过程。

1. DNS查询过程

为什么要ＤＮＳ查询？

因为 **http 是基于 tcp 连接的，而 tcp 则是通过 ip 地址去识别访问的**。DNS 解析就是域名转化成 ip 地址的过程。

当我们在浏览器的地址栏输入网址（譬如： www.linux178.com） ，然后回车，回车这一瞬间到看到页面到底发生了什么呢？

域名解析 –> 发起TCP的3次握手 –> 建立TCP连接后发起http请求 –> 服务器响应http请求，浏览器得到html代码 –> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） –> 浏览器对页面进行渲染呈现给用户。

域名解析是页面加载的第一步，那么域名是如何解析的？

1. Chrome浏览器会**首先搜索自身浏览器的DNS缓存**
2. 如果浏览器自身的缓存里面没有找到对应的条目，那么Chrome会**搜索操作系统自身的DNS缓存**,如果找到且没有过期则停止搜索解析到此结束.
3. 如果在操作系统的DNS缓存也没有找到，那么**尝试读取hosts文件**（位于C:\Windows\System32\drivers\etc），看看这里面有没有该域名对应的IP地址，如果有则解析成功。
4. 如果在hosts文件中也没有找到对应的条目，浏览器就会发起一个DNS的系统调用，就会向**本地配置的首选DNS服务器**（一般是电信运营商提供的，也可以使用像Google提供的DNS服务器）发起域名解析请求（通过的是UDP协议向DNS的53端口发起请求，这个请求是递归的请求，也就是运营商的DNS服务器必须得提供给我们该域名的IP地址），运营商的DNS服务器首先查找自身的缓存，找到对应的条目，且没有过期，则解析成功。

浏览器缓存->操作系统自身DNS缓存->本地host文件->运营商缓存

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。

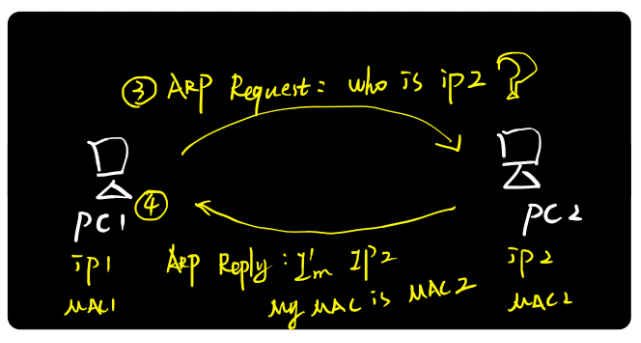
域名解析时使用UDP协议：

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。

1. arp协议
2. **ARP（Address Resolution Protocol）即地址解析协议， 用于实现从 IP 地址到 MAC 地址的映射，即询问目标IP对应的MAC地址**。
3. 在网络通信中，主机和主机通信的数据包需要依据OSI模型从上到下进行数据封装，当数据封装完整后，再向外发出。所以在局域网的通信中，不仅需要源目IP地址的封装，也需要源目MAC的封装。
4. 一般情况下，上层应用程序更多关心IP地址而不关心MAC地址，所以需要通过ARP协议来获知目的主机的MAC地址，完成数据封装。

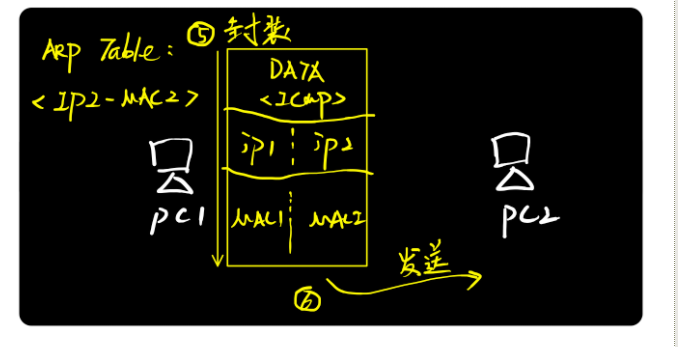
**ARP原理之请求应答**

**同一个局域网内，当p1需要与p2进行通信时，ping指令是由osi顶层到底层之间的封装数据。此时pc1已经有了pc2的ip地址信息，如何获得pc2的mac地址信息呢？此时ARP协议派上用场。我们看这张图。**



通过步骤3和4，我们看到了PC1和pc2进行一次ARP请求和回复的过程，通过这个交互过程PC1便具备了pc2的mac地址信息。

接下来pc会怎么做？在真正通信时，pc1会把pc2的mac信息放入本地的ARP缓存表中，表内放置了ip和MAC地址的映射信息，例如ip2<->mac2.接下来pc1再次进行数据封装，正式进入ping通信，如下



经过上述步骤处理pc1终于把数据包发送出去了，之后便可以进行正常的通信了。ARP的功能和实现过程时如此简单：它在发送方需要目标地址时候及时出手，通过“一问一答”的方式获取特定ip对应的MAC地址，然后存储到本地，后续需要就在这里查找。

1. Ping原理

icmp

1. 子网内计算机通信需要用到路由器吗？

不需要。用交换机、ARP协议广播即可

1. 交换机和路由器的区别 工作方式

路由器工作在第三层（网络层），而我们常说的交换机工作在第二层（数据链路层）。

两者主要的工作：

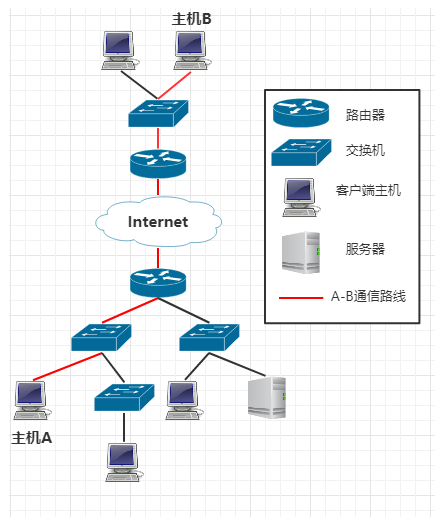
路由器：寻址、转发（依靠IP地址）

交换机：过滤、转发（依靠MAC地址）

两者的主要工作都是转发数据，但不同之处在于，依靠的地址不同，这是一个根本的区别。

路由器内有一份路由表，里面有它的寻址信息（就像一张地图），它收到网络层的数据报后，会根据路由表和选路算法将数据转发到下一站（可能是路由器、交换机、目的主机）。

交换机内有一张MAC表，里面存放着和它相连的所有设备的MAC地址，它会根据收到的数据帧的首部信息内的目的MAC地址在自己表中查找，如果有就转发，如果没有就放弃。



通过拓扑图知道：

每一个路由器与其之下连接的设备，其实构成一个局域网

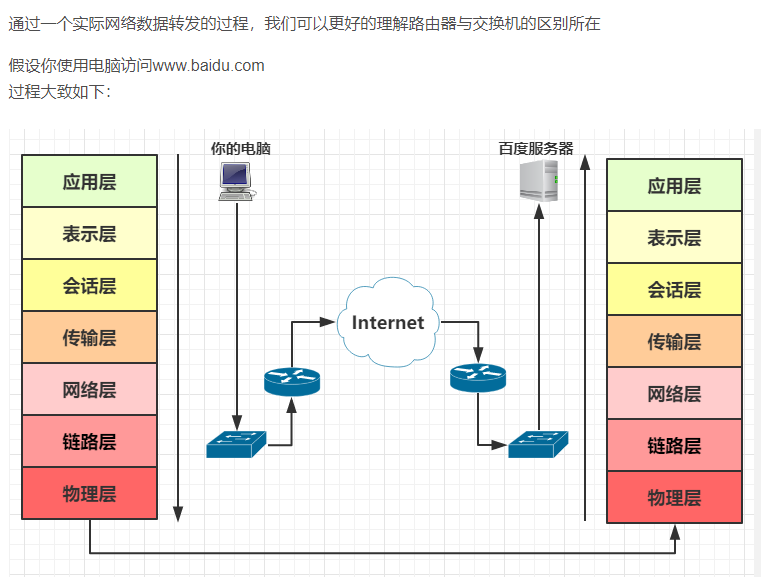
交换机工作在路由器之下，也就是交换机工作在局域网内

交换机用于局域网内网的数据转发

路由器用户连接局域网和外网

非常形象的例子：

我们每个人相当于主机，路由器相当于快递员，宿管大爷相关于交换机，学校是一个局域网。快递员根据学校地址（IP）把包裹送到学校，再根据公寓号（子网IP）把快递交给这个公寓的宿管大爷，宿管大爷根据你的名字（MAC）交给你



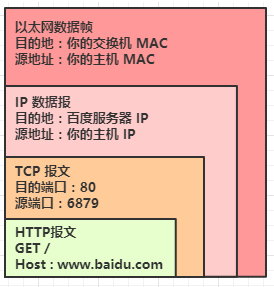
交换机在局域网内工作，它根据MAC地址转发数据，如果没有了路由器在网络层寻址，那么我们的数据就不能发送到其他网络终端上。

路由器内集成了交换机的功能，主机与路由器相连也可以实现数据转发，但是不足之处在于：

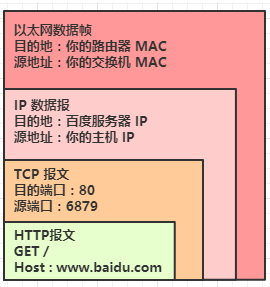
* + - 1. 可扩展的接口不如交换机多
      2. 交换通常由硬件加速转发，路由器主要靠软件寻址，速度慢

**应用层打包数据送到目的主机过程！！！！**

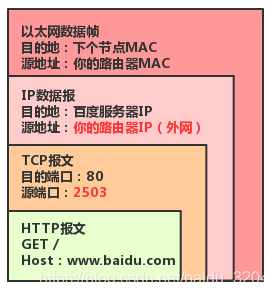
* + 1. 电脑现在应用层打包一个HTTP报文，然后在传输层再打包成TCP报文，然后再根据DNS查到的IP在网络层打包成IP数据报，然后在通过链路层打包成以太网数据帧，发送给你的交换机：



* + 1. 你的交换机收到后，重新包装数据帧，再发送给你的路由器



* + 1. 你的路由器利用NAT(Network Address Translation)，将你的主机IP（局域网IP）转换为外网IP，还会修改端口号，对外完全隐藏你的主机，再根据路由表选择一条合适的路径进行转发



在接下来的过程中，每个节点都只改变MAC地址，然后在网络中一路向着目的地发送

关于NAT:

NAT是一种网络隐藏技术，它通过建立IP地址映射来隐藏内部的网络

它的主要功能有：

提高内部网络安全性

共享网络地址，减少地址消耗

NAT主要有三种实现方式：

静态NAT：最基本的网络转换实现，只转换IP地址，建立IP地址的一对一映射，不支持端口转换。

网络地址端口转换：这种方式支持端口的映射，并允许多台主机共享一个公网IP地址

端口多路复用：是指改变外出数据包的源端口并进行端口转换，即端口转换采用端口多路复用的关系。

1. http响应返回码

http状态码，301，302的区别，499什么意思，502呢？

1. 滑动窗口及其作用

TCP的优势：

出众的一点就是提供一个可靠的，流控的数据传输，所以实现起来要比其他协议复杂的多，先来看下这两个修饰词的意义：

 1. Reliability ，提供TCP的可靠性，TCP的传输要保证数据能够准确到达目的地，如果不能，需要能检测出来并且重新发送数据。

 2. Data Flow Control，提供TCP的流控特性，管理发送数据的速率，不要超过设备的承载能力

为了能够实现以上2点，TCP实现了很多细节的功能来保证数据传输，比如说 滑动窗口适应系统，超时重传机制，累计ACK等，这次先介绍一下滑动窗口的一些知识点。

1. Cookie和Session（与各自应用场景）

什么是Cookie

　由于HTTP是一种无状态的协议，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是Cookie的工作原理。

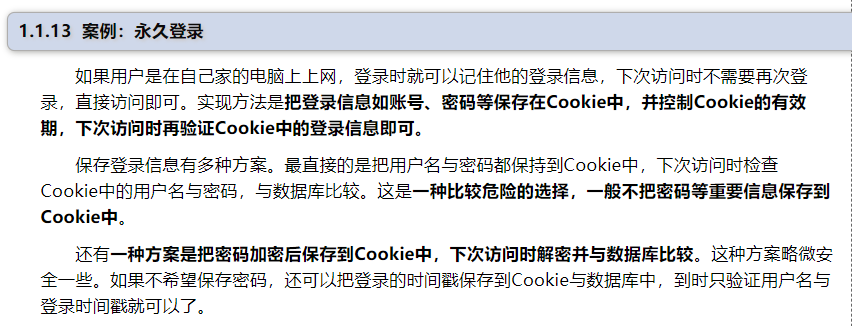
Cookie实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。**客户端浏览器会把Cookie保存起来**。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。

注意：Cookie功能需要浏览器的支持。

如果浏览器不支持Cookie（如大部分手机中的浏览器）或者把Cookie禁用了，Cookie功能就会失效。

### Cookie的不可跨域名性

**Cookie具有不可跨域名性**。根据Cookie规范，浏览器访问Google只会携带Google的Cookie，而不会携带Baidu的Cookie。Google也只能操作Google的Cookie，而不能操作Baidu的Cookie。



什么是Session

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

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

　　如果说Cookie机制是通过检查客户身上的“通行证”来确定客户身份的话，那么Session机制就是通过检查服务器上的“客户明细表”来确认客户身份。Session相当于程序在服务器上建立的一份客户档案，客户来访的时候只需要查询客户档案表就可以了。

Session对应的类为javax.servlet.http.HttpSession类。每个来访者对应一个Session对象，所有该客户的状态信息都保存在这个Session对象里。**Session对象是在客户端第一次请求服务器的时候创建的**。Session也是一种key-value的属性对，通过getAttribute(Stringkey)和setAttribute(String key，Objectvalue)方法读写客户状态信息。Servlet里通过request.getSession()方法获取该客户的Session

### Session的生命周期

Session保存在服务器端。为了获得更高的存取速度，服务器一般把Session放在内存里。每个用户都会有一个独立的Session。如果Session内容过于复杂，当大量客户访问服务器时可能会导致内存溢出。因此，Session里的信息应该尽量精简。

　　Session在用户第一次访问服务器的时候自动创建。需要注意只有访问JSP、Servlet等程序时才会创建Session，只访问HTML、IMAGE等静态资源并不会创建Session。如果尚未生成Session，也可以使用request.getSession(true)强制生成Session。

　　Session生成后，只要用户继续访问，服务器就会更新Session的最后访问时间，并维护该Session。用户每访问服务器一次，无论是否读写Session，服务器都认为该用户的Session“活跃（active）”了一次。

由于会有越来越多的用户访问服务器，因此Session也会越来越多。为防止内存溢出，服务器会把长时间内没有活跃的Session从内存删除。这个时间就是Session的超时时间。如果超过了超时时间没访问过服务器，Session就自动失效了。

　　Session的超时时间为maxInactiveInterval属性，可以通过对应的getMaxInactiveInterval()获取，通过setMaxInactiveInterval(longinterval)修改。

　　Session的超时时间也可以在web.xml中修改。另外，通过调用Session的invalidate()方法可以使Session失效。

虽然Session保存在服务器，对客户端是透明的，它的正常运行仍然需要客户端浏览器的支持。这是因为Session需要使用Cookie作为识别标志。HTTP协议是无状态的，Session不能依据HTTP连接来判断是否为同一客户，因此服务器向客户端浏览器发送一个名为JSESSIONID的Cookie，它的值为该Session的id（也就是HttpSession.getId()的返回值）。Session依据该Cookie来识别是否为同一用户。

　　该Cookie为服务器自动生成的，它的maxAge属性一般为–1，表示仅当前浏览器内有效，并且各浏览器窗口间不共享，关闭浏览器就会失效。

　　因此同一机器的两个浏览器窗口访问服务器时，会生成两个不同的Session。但是由浏览器窗口内的链接、脚本等打开的新窗口（也就是说不是双击桌面浏览器图标等打开的窗口）除外。这类子窗口会共享父窗口的Cookie，因此会共享一个Session。

　　注意：新开的浏览器窗口会生成新的Session，但子窗口除外。子窗口会共用父窗口的Session。例如，在链接上右击，在弹出的快捷菜单中选择“在新窗口中打开”时，子窗口便可以访问父窗口的Session。

　　如果客户端浏览器将Cookie功能禁用，或者不支持Cookie怎么办？例如，绝大多数的手机浏览器都不支持Cookie。Java Web提供了另一种解决方案：URL地址重写。

Cookie和Session的区别

* + - 1. cookie数据存放在客户的浏览器上，session数据放在服务器上。

简单的来说，当你登录一个网站的时候，如果web服务器端使用的是session，那么所有的数据都保存在服务器上面，

客户端每次请求服务器的时候会发送当前会话的session\_id，服务器根据当前session\_id判断相应的用户数据标志，以确定用户是否登录，或具有某种权限

由于数据是存储在服务器上面，所以你不能伪造，但是如果你能够获取某个登录用户的session\_id，用特殊的浏览器伪造该用户的请求也是能成功的。

Session\_id是服务器和客户端链接时候随机分配的，一般来说不会有重复，但是如果有大量并发请求，也不是没有重复的可能性

Session是由应用服务器维持的一个服务器端的存储空间，用户在链接服务器时，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的，用cookie保存的，用户提交页面时，会将这一sessionId提交给服务器端，来存取session数据。这一过程，是不用开发者干预的。所以一旦客户端禁用cookie，那么session也会失效。

* + - 1. cookie不是很安全，别人可以分析存放在本地的cookie并进行cookie欺骗，考虑安全应当使用session
      2. 设置cookie时间可以使cookie过期，但是使用session-destory，我们将销毁对话
      3. Session会在一定时间内保存在服务器上，当访问增多，会比较占用你服务器的性能，考虑到减轻服务器性能方面，应当使用cookie
      4. 单个cookie保存的数据不能超过4k，很多浏览器都限制一个站点最多保存20个cookie（Session没有对存储的数据量的限制，其中可以保存更为复杂的数据类型）

 注意:      session很容易失效,用户体验很差;      虽然cookie不安全,但是可以加密 ;      cookie也分为永久和暂时存在的;      浏览器 有禁止cookie功能 ,但一般用户都不会设置;     一定要设置失效时间,要不然浏览器关闭就消失了;

Session信息是存放在server端，但session id是存放在client cookie的

Cookie是完全保持在客户端的如：IE firefox 当客户端禁止cookie时将不能再使用

Session和cookie的使用场景：

1. 客户端禁用了cookie、session还能用吗？

一般默认情况下，在会话中，服务器存储 session 的 sessionid 是通过 cookie 存到浏览器里。

如果浏览器禁用了 cookie，浏览器请求服务器无法携带 sessionid，服务器无法识别请求中的用户身份，session失效。

但是可以通过其他方法在禁用 cookie 的情况下，可以继续使用session。

1. 通过url重写，把 sessionid 作为参数追加的原 url 中，后续的浏览器与服务器交互中携带 sessionid 参数。

2. 服务器的返回数据中包含 sessionid，浏览器发送请求时，携带 sessionid 参数。

3. 通过 Http 协议**其他 header 字段**，服务器每次返回时设置该 header 字段信息，浏览器中 js 读取该 header 字段，请求服务器时，js设置携带该 header 字段。

1. TCP为什么数据量比UDP大？如果有1M数据，两者会差多少？
2. IEG光子
3. TCP为什么比UDP慢

TCP需要三次握手才能建立链接

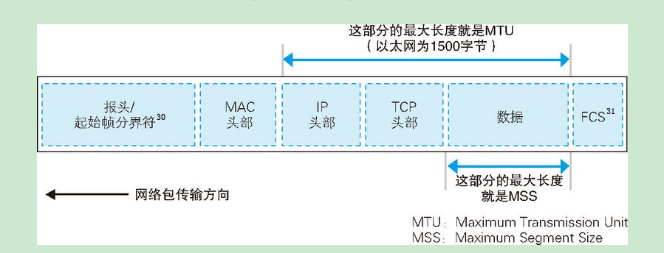
TCP是面向连接的，UDP是无连接的。

TCP提供一堆重传机制和校验和等保证数据可靠性。

Udp面向报文、面向无连接

TCP面向字节流、面向连接的

1. MSS 和 MTU



MTU： Maximum Transmit Unit，最大传输单元，即物理接口（数据链路层）提供给其上层（通常是IP层）最大一次传输数据的大小；以普遍使用的以太网接口为例，缺省MTU=1500 Byte，这是以太网接口对IP层的约束，如果IP层有<=1500 byte 需要发送，只需要一个IP包就可以完成发送任务；如果IP层有> 1500 byte 数据需要发送，需要分片才能完成发送，这些分片有一个共同点，即IP Header ID相同。

MSS：Maximum Segment Size ，TCP提交给IP层最大分段大小，不包含TCP Header和 TCP Option，只包含TCP Payload ，MSS是TCP用来限制application层最大的发送字节数。如果底层物理接口MTU= 1500 byte，则 MSS = 1500- 20(IP Header) -20 (TCP Header) = 1460 byte，如果application 有2000 byte发送，需要两个segment才可以完成发送，第一个TCP segment = 1460，第二个TCP segment = 540。

1. UDP接受到包不是按顺序的，怎么办

（发送端+包标号 接收端+允许短暂时延）

1. Udp怎么保证分的包能被正确接受？

在UDP通讯中，当你的数据包发出去后，至于对方有没有正确收到数据，并不知道，那么，如何保证你发出去的数据，对方一定能收到呢？？？我们可以借鉴TCP协议的做法（**回复+重发+编号 机制**）

在应用层上实现。

1）接收方收到数据后，回复一个确认包，如果你不回复，那么发送端是不会知道接收方是否成功收到数据的

      比如A要发数据“{data}”到B，那B收到后，可以回复一个特定的确认包“{OK}”，表示成功收到。

    但是如果只做上面的回复处理，还是有问题，比如B收到数据后回复给A的数据"{OK}"的包，A没收到，怎么办呢？？？

2）当A没有收到B的"{OK}"包后，要做定时重发数据，直到成功接收到确认包为止，再发下面的数据，当然，重发了一定数量后还是没能收到确认包，可以执行一下ARP的流程，防止对方网卡更换或别的原因。

但是这样的话，B会收到很多重复的数据，假如每次都是B回复确认包A收不到的话。

3）发送数据的包中加个标识符，比如A要发送的数据"{标识符|data}"到B，B收到后，先回复“{OK}"确认包，再根据原有的标识符进行比较，如果标识符相同，则数据丢失，如果不相同，则原有的标识符 = 接收标识符，且处理数据。

  当A发送数据包后，没有收到确认包，则每隔x秒，把数据重发一次，直到收到确认包后，更新一下标识符，再进行后一包的数据发送。

经过上面1），2），3）点的做法，则可以保证数据百分百到达对方，当然，标识符用ID号来代替更好。

1. Udp，可靠实现，应用场景（qq聊天为啥要用udp而非tcp）

https://blog.csdn.net/junjunba2689/article/details/82593124?utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-2.control&dist\_request\_id=1328767.80821.16177676304560943&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-2.control

1. TCP/UDP的各自应用场景

TCP：

对数据传输的质量有较高要求，但对实时性不高。比如HTTP、https等传输文件的协议。应选用TCP协议。

UDP：

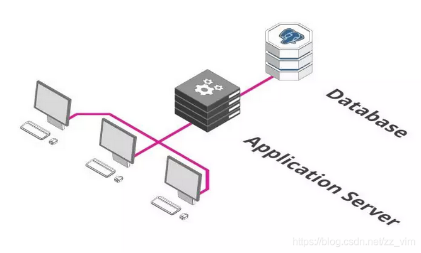
只对数据传输的实时性要求较高，但不对传输质量有要求。比如视频传输、实时通信等，应选用udp协议。

1. TCP延迟确认机制
2. HTTP长连接短链接

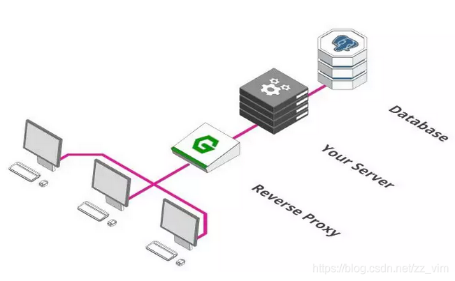
短连接：浏览器和服务器每进行一次HTTP操作，就建立一次连接，任务结束后就中断连接。

长连接：当一个网页打开完成后，客户端和服务器之间用于HTTP传输的tcp连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这条已经建立的连接。

1. 高并发问题的解决历程
2. 单服务器+数据库（原始）



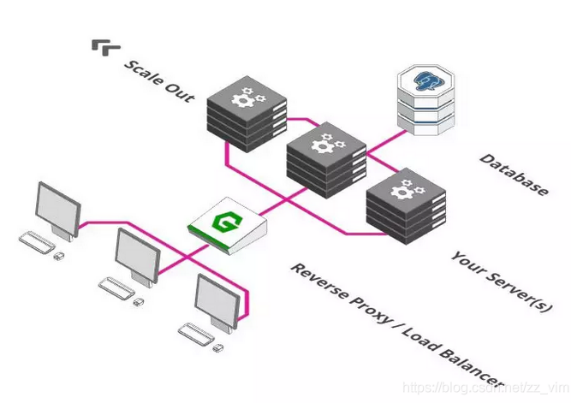
1. 增加反向代理



代理是一个接受和转发请求的过程。正常情况下，「正向代理」代理的对象是客户端，「反向代理」代理的对象是服务端，它完成这些功能：

* + - 1. 健康检查功能，确保服务器一直处于运行状态
      2. 路由转发功能，把请求转发到正确的服务路径上
      3. 认证功能：确保用户有权限访问后端服务器
      4. 防火墙功能：确保用户只能访问允许使用的网络部分等等

1. 引入负载均衡器



反向代理还有另外一个功能：它们也可以充当负载均衡器

比如Nginx经过配置，可以反向代理多台服务器。

部署多台Nginx（反向代理服务器），防止宕机，提升系统运行稳定性。

1. 扩展数据库

负载均衡的使用使得我们在多个服务器之间分配负载，但是现在所有的服务器还是使用一个数据库进行存储和检索数据。我们可以用同样的方式，再扩展几台数据库出来，减轻存储检索压力，但是这里存在一个数据一致性问题。

1. 主从模式或者单实例写多副本读

即其中一台数据库负责数据写入修改，其他服务器负责读，这个方案的好处是保证了一致性，因为数据只能被单实例一台服务器写入，之后把写入数据同步到其他部分即可，所以该方案适合读多写少的情景。

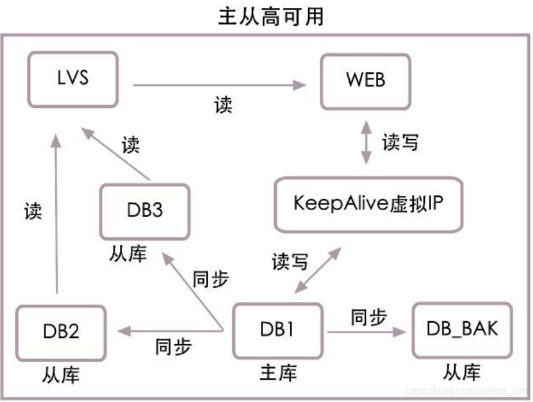
1. 如何进行同步

通过使用消息队列进行异步数据同步，来实现数据的最终一致性。当然消息队列的各种异常也会造成数据不一致。所以我们又引入了实时监控服务，实时计算两个集群的数据差距，并进行一致性同步。

1. 主从模式弊病

 当主库DB1出现问题时，DBA会将DB2切换为主库，并通知项目组，项目组使用DB2替换原有的主库DB1，重启web服务器，这样web服务将使用新的主库DB2，而DB1将不再被访问，整个数据库服务得到恢复，等DBA修复DB1时，再将DB1作为DB2的从库即可。这里有个很大的问题，就是不管主库或从库出现问题，都需要DBA和项目组协同完成数据库服务恢复，这很难做到自动化，而且恢复工程也过于缓慢。（即之前项目提到的数据库连接池的维护，切换主库，再次创建连接池，并重启）

所以数据库如何做到高可用呢？

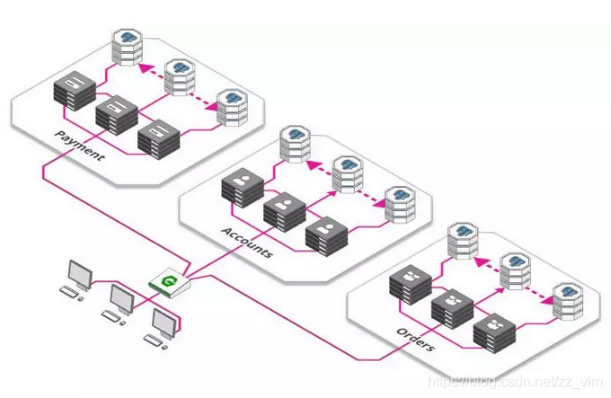


 如上图所示，web服务器将不再直接连接主库DB1，而是连接KeepAlive**虚拟出的一个虚拟ip**，再将此虚拟ip映射到主库DB1上，同时添加DB\_bak从库，实时同步DB1中的数据。正常情况下web还是在DB1中读写数据，**当DB1宕机后，脚本会自动将DB\_bak设置成主库**，并将虚拟ip映射到DB\_bak上，web服务将使用健康的DB\_bak作为主库进行读写访问。这样只需几秒的时间，就能完成主数据库服务恢复。（再进行实时监控服务，实现数据一致性）

**同样的，web服务器将不再直接连接从库DB2和DB3，而是连接LVS负载均衡，由LVS连接从库。这样做的好处是LVS能自动感知从库是否可用，从库DB2宕机后，LVS将不会把读数据请求再发向DB2。同时DBA需要增减从库节点时，只需独立操作LVS即可，不再需要项目组更新配置文件，重启服务器来配合。**

1. 微服务

  将所有服务放在一个服务器上，放在一个JAR包，降低了复杂性，但是随着规模的增加，事情会变得复杂和低效，更多的开发人员加入进来，在同一台服务器同一个项目代码里面进行开发，造成的冲突会越来越多，互相依赖性太高，同时不利于新入开发人员阅读理解代码。



1. 缓存和内容分发网络（CDN）

内容分发网络（Content Delivery Network，简称CDN）是建立并覆盖在承载网之上，由分布在不同区域的边缘节点服务器群组成的分布式网络。

如果客户端远离数据中心，服务器到客户的分组将跨越许多通信链路并很可能通过许多ISP，其中某些ISP可能位于不同的达州。很可能会给用户带来停滞时延。

为了应对向分布于全世界的用户分发巨量视频数据挑战，记录所有主要的视频流公司都利用内容分发网（CDN）。CDN管理分布在多个地理位置上的服务器，在它的服务器中存储视频的副本，并且所有试图将每个用户请求定向到一个能将提供最好的用户体验的CDN位置。

1. 消息队列

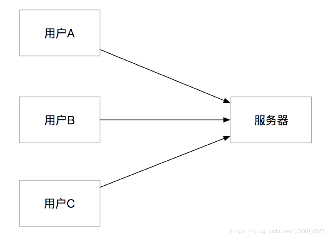
 将客户的任务请求放到一个队列当中，进行管理任务，优点：

* 解耦了任务和处理过程。有时需要处理大量的图片，有时很少。有时有大量服务可用，有时很少可用。简单地把任务添加到待办事项而不是直接处理它们，这确保了系统保持响应并且任务也不会丢失。
* 可以按需扩展。启动大量的服务比较耗时，所以当有大量用户上传图片时再去启动服务，这已经太晚了。我们把任务添加到队列中，我们可以推迟提供额外的处理能力。

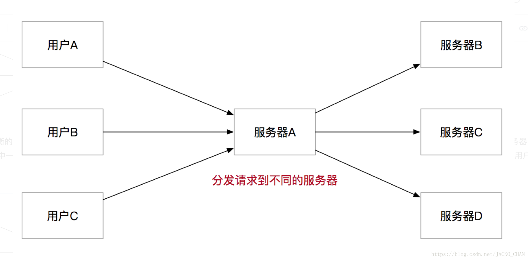
1. 假设我们写个服务器，估计要支持几万个人同时在线，这时候应该怎么来做？（负载均衡+IO多路复用+数据库扩展（+微服务+消息队列+）多台云服务器）
2. 你的项目如何实现负载均衡

什么是负载均衡？

这是没有用负载均衡的情况，用户都是直接请求到一台服务器上，并发量不大的时候不建议采用负载均衡，毕竟需要多台服务器来支撑，成本也是非常高昂的；但是用户量大的时候，服务器承载压力就会变大，如果出现宕机的情况，那就直接GG了；



采用负载均衡的话，用户请求的时候，服务器A会根据不同的策略（具体策略下面详细剖解），分配到服务器（B、C、D）其中一台；这样的话，每台服务器都可以承担压力，就算其中一台服务器宕机的时候，还有其他服务器来支持，也不会导致系统停运，请求地址在用户看来都是一样的，只是后台请求的是不同服务器，用户无感知变化；



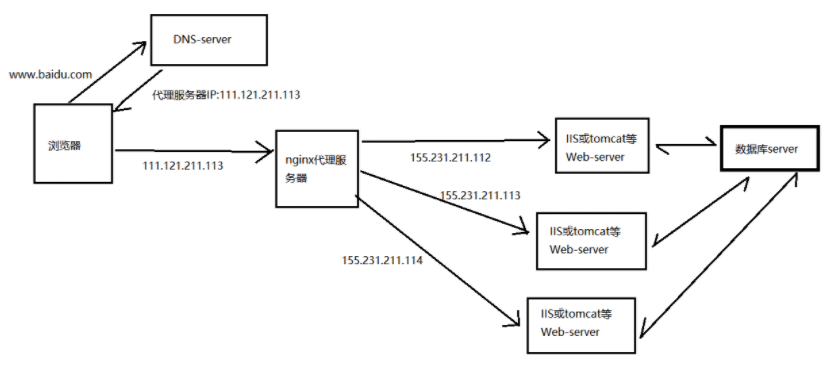
实现负载均衡方案（首先先先有多个服务器）

* 1. DNS轮询实现负载均衡
     + 浏览器通过DNS-server，域名解析到ip，这一步我们可以给DNS-server添加多条解析记录绑定多个服务器IP，然后通过配置记录的权重来实现负载均衡
     + 浏览器通过ip访问web-server
     + web-server处理访问数据库数据处理数据并返回到浏览器

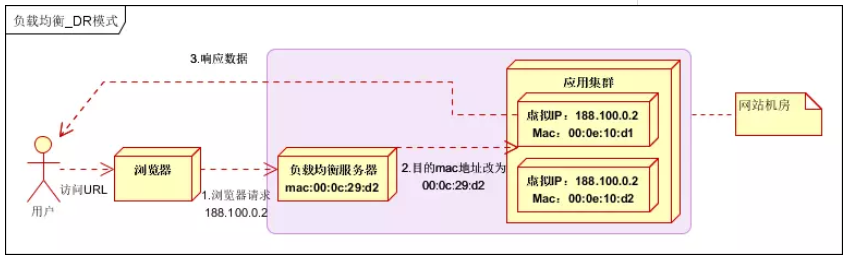
优点：零成本（域名注册免费提供解析服务）、部署简单

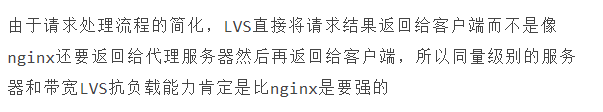
缺点：可靠性低、负载分配不均匀、暴露太多的外网ip

* 1. nginx反向代理实现负载均衡
     + 浏览器通过DNS-server，域名解析到Nginx代理服务器IP
     + 浏览器通过ip访问代理服务器web-server
     + 代理服务器web-server根据配置文件分发请求给实际真实的后端服务器
     + 后端实际web-server处理访问数据库数据处理数据并返回到代理服务器，最后通过代理服务器返回给浏览器（返回也得经过nginx代理服务器）



* 1. LVS调度实现负载均衡





* 1. 负载均衡器硬件实现负载均衡（硬件方面）

1. 概括性问题：如何设计对战服务器（详细）
3. **socket什么情况下可读？**
4. **nginx的epoll模型的介绍以及io多路复用模型**
5. **tcpdump抓包，如何分析数据包**
6. **tcp如何设定超时时间**
7. **基于socket网络编程和tcp/ip协议栈，讲讲从客户端send()开始，到服务端recv()结束的过程，越细越好**
8. **http3了解吗**
9. **http1.1长连接时，发送一个请求阻塞了，返回什么状态码？**
10. **udp调用connect有什么作用？**