目录

[网络 2](#_Toc498587454)

[【网络基础】OSI TCP/IP体系结构 2](#_Toc498587455)

[以太网帧格式 4](#_Toc498587456)

[IP数据报格式 5](#_Toc498587457)

[TCP结构 11](#_Toc498587458)

[TCP之Nagle算法&&延迟ACK 14](#_Toc498587459)

[TCP分段与ip分片 15](#_Toc498587460)

[TCP拥塞控制-慢启动、拥塞避免、快重传、快启动 17](#_Toc498587461)

[UDP结构 21](#_Toc498587462)

[粘包与分包 21](#_Toc498587463)

[NTP时间同步 23](#_Toc498587464)

[全双工半双工 24](#_Toc498587465)

[socket 25](#_Toc498587466)

[Socket level options 30](#_Toc498587467)

[多播/组播 31](#_Toc498587468)

[RTMP 35](#_Toc498587469)

[RTSP 37](#_Toc498587470)

[P2P 41](#_Toc498587471)

[中继（TURN基于此） 43](#_Toc498587472)

[MPEG 43](#_Toc498587473)

[H264基本概念之 宏块、片和片组 44](#_Toc498587474)

[VCL存在三种封装形式：SODB, RBSP, EBSP. 45](#_Toc498587475)

[阐述H264裸流与TS流之间的关系(一) 47](#_Toc498587476)

[RTP 48](#_Toc498587477)

[c++语言 49](#_Toc498587478)

[关键字 49](#_Toc498587479)

[智能指针 49](#_Toc498587480)

[知识点 50](#_Toc498587481)

[webrtc 50](#_Toc498587482)

[Sigslot：信号槽 51](#_Toc498587483)

[talk\_base::Event 51](#_Toc498587484)

[CriticalSection:临界区 52](#_Toc498587485)

[talk\_base::MessageHandler 53](#_Toc498587486)

[messagequeue 54](#_Toc498587487)

[voice\_engine 58](#_Toc498587488)

[NACK与重传 64](#_Toc498587489)

[jitterbuffer 65](#_Toc498587490)

[带宽自适应 65](#_Toc498587491)

[前向纠错编码 66](#_Toc498587492)

[OpenGL 67](#_Toc498587493)

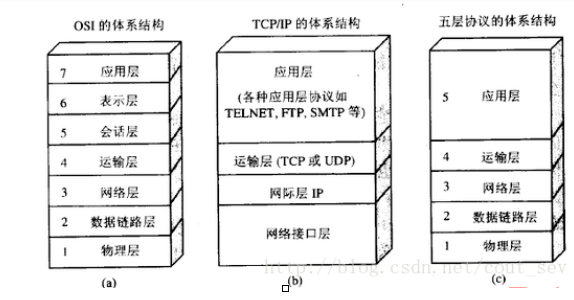
[C程序编译过程浅析 67](#_Toc498587494)

[Linux内存分布 69](#_Toc498587495)

[mach-o 72](#_Toc498587496)

# 网络

## 【网络基础】OSI TCP/IP体系结构



①物理层：指的是电信号传递方式，比如现在以太网通用的网线(双绞线)、早期以太网采用的同轴电缆(现在主要用于有线电视)、光纤等都属于物理层的概念。 物理层的能力决 定了最大传输速率、传输距离、抗干扰性等。集线器(Hub)是工作在物理层的网络设备，用于双绞线的连接和信号中继(将已衰减的信号再次放大使之传得更远)。

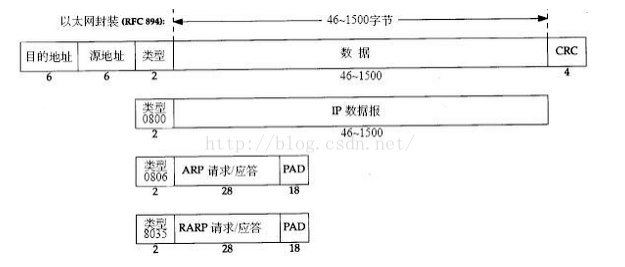
②链路层：链路层有以太网、令牌环网等标准。链路层负责网卡设备的驱动、帧同步(就是说从网线上检测到什么信号算作新帧的开始)、冲突检测(如果检测到冲突就自动重发)、 数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不 同的链路层网络之间转发数据帧(比如十兆以太网和百兆以太网之间、以太网和令牌环网之间)，由 于不同链路层的帧格式不同，交换机要将进来的数据包拆掉链路层首部重新封装之后再转发。网络层的IP协议是构成Internet的基础。Internet上的主机通过IP地址来标识，Internet上有大量路由器负责根据IP地址选择合适的路径转发数据包，数据包从Internet上的源主机到目的主机往往要经过十多个路由器。路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包。因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP 协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。 争论：为什么以太网要mac寻址，而不是用ip寻址。

③网络层：负责点到点(point-to-point)的传输(这里的“点”指主机或路由器)，而传输层负责端到 端 (end-to-end)的传输(这里 的“端”指源主机和目的主机)。

④传输层：该层定义了两个重要的协议，即传输控制协议（TCP）或用户数据报协议（UDP）。TCP是一种面向连接的、可靠的协议。就像打电话，双方拿起电话互通身份之后就建立了连接，然后说话就行了，这边说的话那边保证听得到，并且是按说话的顺序听到的，说完话挂机断开连接。也就是说TCP传输的双方需要首先建立连接，之后由TCP协议保证数据收发的可靠性,丢失的数据包自动重发，上层应用程序收到的总是可靠的数据流,通讯之后关闭连接。UDP协议不面向连接，也不保证可靠性，有点像寄信，写好信放到邮筒里，既不能保证信件在邮递过程中不会丢失，也不能保证信件是按顺序寄到目的地的。使用UDP协议的应用程序需要自己完成丢包重发、消息排序等工作。

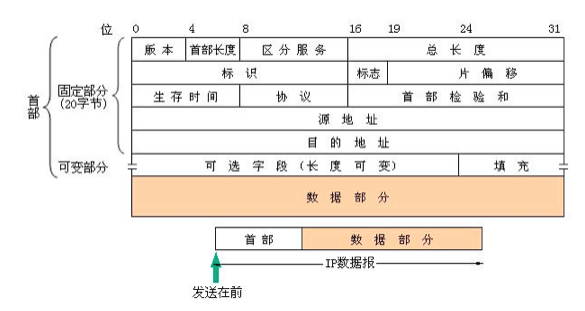
⑤应用层：为用户提供各种所需的服务。比如：FTP，DNS，Telnet，SMTP等。

## 以太网帧格式



帧格式：其中源地址和目的地址指的是网卡的硬件地址（也就是MAC地址），长度为12个字节（48位），由生产厂家烧制具有唯一性的地址标识，出厂时既已固化。在centos下利用config命令可以查看MAC地址。帧协议类型包括了IP、ARP、RARP这三种协议，其类型值分别为0800的IP协议，0806的ARP协议，0835的RARP协议。而后是有效载荷的大小（数据长度），最后是CRC校验码，用于差错校验的。

## IP数据报格式



首部解释（http://blog.sina.com.cn/s/blog\_5ceeb9ea0100wy0z.html）

* IP结构体（http://blog.sina.com.cn/s/blog\_5ceeb9ea0100wy0z.html）

(1)版本(Iphdr->version)　占4位，指IP协议的版本。通信双方使用的IP协议版本必须一致。目前广泛使用的IP协议版本号为4（即IPv4）。关于IPv6，目前还处于草案阶段。

(2)首部长度(Iphdr->ihl)　　占4位，可表示的最大十进制数值是15。请注意，这个字段所表示数的单位是32位字长（1个32位字长是4字节），因此，当IP的首部长度为1111时（即十进制的15），首部长度就达到60字节。当IP分组的首部长度不是4字节的整数倍时，必须利用最后的填充字段加以填充。因此数据部分永远在4字节的整数倍开始，这样在实现IP协议时较为方便。首部长度限制为60字节的缺点是有时可能不够用。但这样做是希望用户尽量减少开销。最常用的首部长度就是20字节（即首部长度为0101），这时不使用任何选项。

(3)区分服务(Iphdr->tos)　　占8位，用来获得更好的服务。这个字段在旧标准中叫做服务类型( 服务类型字段8位: 服务类型(TOS)字段包括一个3 bit的优先权子字段(现在已被忽略)，4 bit的TOS子字段和1 bit未用位但必须置0。4 bit的TOS子字段分别代表:最小时延、最大吞吐量、最高可靠性和最小费用。4 bit中只能设置其中1 bit。如果所有4 bit均为0,那么就意味着是一般服务)，但实际上一直没有被使用过。1998年IETF把这个字段改名为区分服务DS(Differentiated Services)。只有在使用区分服务时，这个字段才起作用。

(4)总长度(Iphdr->tot\_len)　总长度指首部和数据之和的长度，单位为字节。总长度字段为16位，因此数据报的最大长度为65535字节。在IP层下面的每一种数据链路层都有自己的帧格式，其中包括帧格式中的数据字段的最大长度，这称为最大传送单元MTU(Maximum Transfer Unit)。当一个数据报封装成链路层的帧时，此数据报的总长度（即首部加上数据部分）一定不能超过下面的数据链路层的MTU值。

利用首部长度字段和总长度字段,就可以知道 IP数据报中数据内容的起始位置和长度。由于该字段长16比特,所以IP数据报最长可达65535字.总长度字段是IP首部中必要的内容,因为一些数据链路(如以太网)需要填充一些数据以达到最小长度。尽管以太网的最小帧长为46字节,但是IP数据可能会更短。如果没有总长度字段,那么IP层就不知道46字节中有多少是IP数据报的内容。

(5)标识(Iphdr->id)　占16位。IP软件在存储器中维持一个计数器，每产生一个数据报，计数器就加1，并将此值赋给标识字段。但这个“标识”并不是序号，因为IP是无连接服务，数据报不存在按序接收的问题。当数据报由于长度超过网络的MTU而必须分片时，这个标识字段的值就被复制到所有的数据报的标识字段中。相同的标识字段的值使分片后的各数据报片最后能正确地重装成为原来的数据报。

(6)片偏移(Iphdr->frag\_off)　占16位。前3位是标识位，后13位是偏移位。

前三位：

第1位是保留的，必须为0,；

第2位是“更多分片”(MF -- More Fragment)标志。除了最后一片外，其他每个组成数据报的片都要把该比特置1。

第3位是“不分片”(DF -- Don't Fragment)标志,如果将这一比特置1，IP将不对数据报进行分片,这时如果有需要进行分片的数据报到来，会丢弃此数据报并发送一个ICMP差错报文给起始端。

后13位：片偏移位

片偏移指出：较长的分组在分片后，某片在原分组中的相对位置。也就是说，相对用户数据字段的起点，该片从何处开始。片偏移以8个字节为偏移单位。这就是说，每个分片的长度一定是8字节（64位）的整数倍

　(7)生存时间(Iphdr->ttl)　占8位，生存时间字段常用的的英文缩写是TTL(Time To Live)，表明是数据报在网络中的寿命。由发出数据报的源点设置这个字段。其目的是防止无法交付的数据报无限制地在因特网中兜圈子，因而白白消耗网络资源。最初的设计是以秒作为TTL的单位。每经过一个路由器时，就把TTL减去数据报在路由器消耗掉的一段时间。若数据报在路由器消耗的时间小于1秒，就把TTL值减1。当TTL值为0时，就丢弃这个数据报。

TTL(Time to live)域是一个用于限制分组生存期的计数器。这里的计数时间单位为秒，因此最大的生存期为255秒。在每一跳上该计数器必须被递减，而且，当数据报在一台路由器上排队时间较长时，该计数器必须被多倍递减。在实践中，它只是跳计数器，当它递减到0的时候，分组被丢弃，路由器给源主机发送一个警告分组。此项特性可以避免数据报长时间地逗留在网络中，有时候当路由表被破坏之后，这种事情是有可能发生的。

(8)协议(Iphdr-> protocol)　占8位，协议字段指出此数据报携带的数据是使用何种协议，以便使目的主机的IP层知道应将数据部分上交给哪个处理过程。

当网络层组装完成一个完整的数据报之后，它需要知道该如何对它进行处理。协议(Protocol)域指明了该将它交给哪个传输进程。TCP是一种可能，但是UDP或者其他的协议也是可能的。

(9)首部检验和(Iphdr->check)　占16位。这个字段只检验数据报的首部，但不包括数据部分。这是因为数据报每经过一个路由器，路由器都要重新计算一下首部检验和（一些字段，如生存时间、标志、片偏移等都可能发生变化）。不检验数据部分可减少计算的工作量。

首部检验和字段(16位)是根据IP首部计算的检验和码。它不对首部后面的数据进行计算。 ICMP、IGMP、UDP和TCP在它们各自的首部中均含有同时覆盖首部和数据检验和码。

为了计算一份数据报的IP检验和,首先把检验和字段置为0。然后,对首部中每个16 bit进行二进制反码求和(整个首部看成是由一串16 bit的字组成),结果存在检验和字段中。当收到一份IP数据报后,同样对首部中每个16 bit进行二进制反码的求和。由于接收方在计算过程中包含了发送方存在首部中的检验和,因此,如果首部在传输过程中没有发生任何差错,那么接收方计算的结果应该为全1。如果结果不是全1(即检验和错误),那么IP就丢弃收到的数据报。但是不生成差错报文,由上层去发现丢失的数据报并进行重传。

(10)源地址(Iphdr->saddr)　占32位(1111 1111=255)。

(11)目的地址(Iphdr->daddr)　占32位。

* MTU MSS:mtu是网络传输最大报文包。 mss是网络传输数据最大值。 mss加包头数据就等于mtu. 简单说拿TCP包做例子。 报文传输1400字节的数据的话，那么mss就是1400，再加上20字节IP包头，20字节tcp包头，那么mtu就是1400+20+20. 当然传输的时候其他的协议还要加些包头在前面，总之mtu就是总的最后发出去的报文大小。mss就是你需要发出去的数据大小。1.MSS: Maxitum Segment Size 最大分段大小 2.MSS最大传输大小的缩写，是TCP协议里面的一个概念。 3.MSS就是TCP数据包每次能够传输的最大数据分段。为了达到最佳的传输效能TCP协议在建立连接的时候通常要协商双方的MSS值，这个值TCP协议在实现的时候往往用MTU值代替（需要减去IP数据包包头的大小20Bytes和TCP数据段的包头20Bytes）所以往往MSS为1460。通讯双方会根据双方提供的MSS值得最小值确定为这次连接的最大MSS值 。以太网的MTU一般是1500，MSS一般是1500-20（IP包头）-20（TCP包头）。

## TCP结构



TCP协议头最少20个字节，包括以下的区域

TCP源端口(Source Port)：16位的源端口其中包含初始化通信的端口。源端口和源IP地址的作用是

标示报问的返回地址。

　　TCP目的端口(Destination port)：16位的目的端口域定义传输的目的。这个端口指明报文接收计算

机上的应用程序地址接口。

　　TCP序列号（序列码,Sequence Number）：32位

　　TCP应答号(Acknowledgment Number)：32位的序列号由接收端计算机使用，重组分段的报文成最初形式。，如果设置了ACK控制位，这个值表示一个准备接收的包的序列码。

　　数据偏移量(HLEN)：4位包括TCP头大小，指示何处数据开始。

　　保留(Reserved)：6位值域，这些位必须是0。为了将来定义新的用途所保留。

标志(Code Bits)：6位标志域。表示为：紧急标志、有意义的应答标志、推、重置连接标志、同步序列号标志、完成发送数据标志。按照顺序排列是：URG、ACK、PSH、RST、SYN、FIN。

1. URG：紧急标志

紧急(The urgent pointer) 标志有效。紧急标志置位，

2. ACK：确认标志

确认编号(Acknowledgement Number)栏有效。大多数情况下该标志位是置位的。TCP报头内的确认编号栏内包含的确认编号(w+1，Figure：1)为下一个预期的序列编号，同时提示远端系统已经成功接收所有数据。

3. PSH：推标志

该标志置位时，接收端不将该数据进行队列处理，而是尽可能快将数据转由应用处理。在处理 telnet 或 rlogin 等交互模式的连接时，该标志总是置位的。

4. RST：复位标志

复位标志有效。用于复位相应的TCP连接。例如有一端重启时会出现

5. SYN：同步标志

同步序列编号(Synchronize Sequence Numbers)栏有效。该标志仅在三次握手建立TCP连接时有效。它提示TCP连接的服务端检查序列编号，该序列编号为TCP连接初始端(一般是客户端)的初始序列编号。在这里，可以把TCP序列编号看作是一个范围从0到4，294，967，295的32位计数器。通过TCP连接交换的数据中每一个字节都经过序列编号。在TCP报头中的序列编号栏包括了TCP分段中第一个字节的序列编号。

6. FIN：结束标志

带有该标志置位的数据包用来结束一个TCP回话，但对应端口仍处于开放状态，准备接收后续数据。

　　窗口(Window)：16位，用来表示想收到的每个TCP数据段的大小。

　　校验位(Checksum)：16位TCP头。源机器基于数据内容计算一个数值，收信息机要与源机器数值 结果完全一样，从而证明数据的有效性。

　　优先指针（紧急,Urgent Pointer）：16位，指向后面是优先数据的字节，在URG标志设置了时才有效。如果URG标志没有被设置，紧急域作为填充。加快处理标示为紧急的数据段。

选项(Option)：长度不定，但长度必须以字节。如果 没有 选项就表示这个一字节的域等于0，当窗口大小需要更大时可能出现。

数据（Date）：应用程序的数据。

TCP判断数据丢包有两种，在rto时间内没收到ack或者收到重复ack三次。

## [TCP之Nagle算法&&延迟ACK](http://www.cnblogs.com/wanpengcoder/p/5366156.html)

1. Nagle算法：（针对小包）

是为了减少广域网的小分组数目，从而减小网络拥塞的出现；

该算法要求一个tcp连接上最多只能有一个未被确认的未完成的小分组，在该分组ack到达之前不能发送其他的小分组，tcp需要收集这些少量的分组，并在ack到来时以一个分组的方式发送出去；其中小分组的定义是小于MSS的任何分组；

该算法的优越之处在于它是自适应的，确认到达的越快，数据也就发哦送的越快；而在希望减少微小分组数目的低速广域网上，则会发送更少的分组；

2. 延迟ACK：

如果tcp对每个数据包都发送一个ack确认，那么只是一个单独的数据包为了发送一个ack代价比较高，所以tcp会延迟一段时间，如果这段时间内有数据发送到对端，则捎带发送ack，如果在延迟ack定时器触发时候，发现ack尚未发送，则立即单独发送；

延迟ACK好处：

(1) 避免糊涂窗口综合症；

(2) 发送数据的时候将ack捎带发送，不必单独发送ack；

(3) 如果延迟时间内有多个数据段到达，那么允许协议栈发送一个ack确认多个报文段；

3. 当Nagle遇上延迟ACK：

试想如下典型操作，写-写-读，即通过多个写小片数据向对端发送单个逻辑的操作，两次写数据长度小于MSS，当第一次写数据到达对端后，对端延迟ack，不发送ack，而本端因为要发送的数据长度小于MSS，所以nagle算法起作用，数据并不会立即发送，而是等待对端发送的第一次数据确认ack；这样的情况下，需要等待对端超时发送ack，然后本段才能发送第二次写的数据，从而造成延迟；

4. 关闭Nagle算法：

使用TCP套接字选项TCP\_NODELAY可以关闭套接字选项;

## TCP分段与ip分片

1，MTU（Maximum Transmission Unit，MTU），最大传输单元   
 （1）以太网和802.3对数据帧的长度都有一个限制，其最大值分别是1500和1492个字节。链路层的这个特性称作MTU。不同类型的网络大多数都有一个上限。如果IP层有一个数据要传，且数据的长度比链路层的MTU还大，那么IP层就要进行分片（fragmentation），把数据报分成若干片，这样每一个分片都小于MTU。   
 （2）把一份[IP数据报](https://www.baidu.com/s?wd=IP%E6%95%B0%E6%8D%AE%E6%8A%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PWR3m1R4njbsuWuWnhwB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjbvrjf1nW6kPHRYPWczPj6Y)进行分片以后，由到达目的端的IP层来进行重新组装，其目的是使分片和重新组装过程对运输层（TCP/UDP）是透明的。由于每一分片都是一个独立的包，当这些数据报的片到达目的端时有可能会失序，但是在IP首部中有足够的信息让接收端能正确组装这些数据报片。   
 （3）尽管IP分片过程看起来透明的，但有一点让人不想使用它：即使只丢失一片数据也要重新传整个数据报。why？因为IP层本身没有超时重传机制------由更高层（比如TCP）来负责超时和重传。当来自TCP报文段的某一片丢失后，TCP在超时后会重发整个TCP报文段，该报文段对应于一份[IP数据报](https://www.baidu.com/s?wd=IP%E6%95%B0%E6%8D%AE%E6%8A%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PWR3m1R4njbsuWuWnhwB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjbvrjf1nW6kPHRYPWczPj6Y)（而不是一个分片），没有办法只重传数据报中的一个数据分片。   
 （4）使用UDP很容易导致IP分片，TCP试图避免IP分片。那么TCP是如何试图避免IP分片的呢？其实说白了，采用[TCP协议](https://www.baidu.com/s?wd=TCP%E5%8D%8F%E8%AE%AE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PWR3m1R4njbsuWuWnhwB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjbvrjf1nW6kPHRYPWczPj6Y)进行数据传输是不会造成IP分片的，因为一旦TCP数据过大，超过了MSS，则在传输层会对TCP包进行分段（如何分，见下文！），自然到了IP层的数据报肯定不会超过MTU，当然也就不用分片了。而对于UDP数据报，如果UDP组成的[IP数据报](https://www.baidu.com/s?wd=IP%E6%95%B0%E6%8D%AE%E6%8A%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PWR3m1R4njbsuWuWnhwB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjbvrjf1nW6kPHRYPWczPj6Y)长度超过了1500，那么IP数据报显然就要进行分片，因为UDP不能像TCP一样自己进行分段。总结：UDP不会分段，就由我IP来分。TCP会分段，当然也就不用我IP来分了！   
2，MSS（Maxitum Segment Size）最大分段大小的缩写，是[TCP协议](https://www.baidu.com/s?wd=TCP%E5%8D%8F%E8%AE%AE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PWR3m1R4njbsuWuWnhwB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjbvrjf1nW6kPHRYPWczPj6Y)里面的一个概念 （1）MSS就是TCP数据包每次能够传输的最大数据分段。为了达到最佳的传输效能[TCP协议](https://www.baidu.com/s?wd=TCP%E5%8D%8F%E8%AE%AE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PWR3m1R4njbsuWuWnhwB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjbvrjf1nW6kPHRYPWczPj6Y)在建立连接的时候通常要协商双方的MSS值，这个值TCP协议在实现的时候往往用MTU值代替（需要减去IP数据包包头的大小20Bytes和TCP数据段的包头20Bytes）所以往往MSS为1460。通讯双方会根据双方提供的MSS值得最小值确定为这次连接的最大MSS值。   
 （2）相信看到这里，还有最后一个问题：TCP是如何实现分段的呢？其实TCP无所谓分段，因为每个TCP数据报在组成前其大小就已经被MSS限制了，所以TCP数据报的长度是不可能大于MSS的，当然由它形成的IP包的长度也就不会大于MTU，自然也就不用IP分片了

## TCP拥塞控制-慢启动、拥塞避免、快重传、快启动

一般原理：发生拥塞控制的原因：资源(带宽、交换节点的缓存、处理机)的需求>可用资源。

作用：拥塞控制就是为了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者链路不至于过载。拥塞控制要做的都有一个前提：就是网络能够承受现有的网络负荷。

对比流量控制：拥塞控制是一个全局的过程，涉及到所有的主机、路由器、以及降低网络相关的所有因素。流量控制往往指点对点通信量的控制。是端对端的问题。

**拥塞窗口**：发送方为一个动态变化的窗口叫做拥塞窗口，拥塞窗口的大小取决于网络的拥塞程度。发送方让自己的发送窗口=拥塞窗口，但是发送窗口不是一直等于拥塞窗口的，在网络情况好的时候，拥塞窗口不断的增加，发送方的窗口自然也随着增加，但是接受方的接受能力有限，在发送方的窗口达到某个大小时就不在发生变化了。

**发送方如果知道网络拥塞了呢？**发送方发送一些报文段时，如果发送方没有在时间间隔内收到接收方的确认报文段，则就可以人为网络出现了拥塞。

**慢启动算法的思路**：主机开发发送数据报时，如果立即将大量的数据注入到网络中，可能会出现网络的拥塞。慢启动算法就是在主机刚开始发送数据报的时候先探测一下网络的状况，如果网络状况良好，发送方每发送一次文段都能正确的接受确认报文段。那么就从小到大的增加拥塞窗口的大小，即增加发送窗口的大小。

**例子：**开始发送方先设置cwnd（拥塞窗口）=1,发送第一个报文段M1，接收方接收到M1后，发送方接收到接收方的确认后，把cwnd增加到2，接着发送方发送M2、M3，发送方接收到接收方发送的确认后cwnd增加到4，慢启动算法每经过一个传输轮次（认为发送方都成功接收接收方的确认），拥塞窗口cwnd就加倍。

**拥塞避免：**为了防止cwnd增加过快而导致网络拥塞，所以需要设置一个**慢开始门限**ssthresh状态变量（我也不知道这个到底是什么，就认为他是一个拥塞控制的标识）,它的用法：

                   1. 当cwnd < ssthresh,使用慢启动算法，

                   2. 当cwnd > ssthresh,使用拥塞控制算法，停用慢启动算法。

                   3. 当cwnd = ssthresh，这两个算法都可以。

**拥塞避免的思路**：是让cwnd缓慢的增加而不是加倍的增长，每经历过一次往返时间就使cwnd增加1，而不是加倍，这样使cwnd缓慢的增长，比慢启动要慢的多。

    无论是慢启动算法还是拥塞避免算法，只要判断网络出现拥塞，就要把慢启动开始门限(ssthresh)设置为设置为**发送窗口**的一半（>=2），cwnd(拥塞窗口)设置为1，然后在使用慢启动算法，这样做的目的能迅速的减少主机向网络中传输数据，使发生拥塞的路由器能够把队列中堆积的分组处理完毕。拥塞窗口是按照线性的规律增长，比慢启动算法拥塞窗口增长块的多。

  实例：1.TCP连接进行初始化的时候，cwnd=1,ssthresh=16。

             2.在慢启动算法开始时，cwnd的初始值是1，每次发送方收到一个ACK拥塞窗口就增加1，当ssthresh =cwnd时，就启动拥塞控制算法，拥塞窗口按照规律增长，

             3.当cwnd=24时，网络出现超时，发送方收不到确认ACK，此时设置ssthresh=12,(二分之一cwnd),设置cwnd=1,然后开始慢启动算法，当cwnd=ssthresh=12,慢启动算法变为拥塞控制算法，cwnd按照线性的速度进行增长。

快重传：

快重传算法要求首先接收方收到一个失序的报文段后就立刻发出重复确认，而不要等待自己发送数据时才进行捎带确认。接收方成功的接受了发送方发送来的M1、M2并且分别给发送了ACK，现在接收方没有收到M3，而接收到了M4，显然接收方不能确认M4，因为M4是失序的报文段。如果根据可靠性传输原理接收方什么都不做，但是按照快速重传算法，在收到M4、M5等报文段的时候，不断重复的向发送方发送M2的ACK,如果接收方一连收到三个重复的ACK,那么发送方不必等待重传计时器到期，由于发送方尽早重传未被确认的报文段。

**快恢复：**

       1. 当发送发连续接收到三个确认时，就执行乘法减小算法，把慢启动开始门限（ssthresh）减半，但是接下来并不执行慢开始算法。

       2. 此时不执行慢启动算法，而是把cwnd设置为ssthresh的一半， 然后执行拥塞避免算法，使拥塞窗口缓慢增大。

## UDP结构



typedef struct \_UDP\_HEADER

{

unsigned short m\_usSourPort; 　　　// 源端口号16bit

unsigned short m\_usDestPort; 　　　// 目的端口号16bit

unsigned short m\_usLength; 　　　　// 数据包长度16bit

unsigned short m\_usCheckSum; 　　// 校验和16bit

}\_\_attribute\_\_((packed))UDP\_HEADER, \*PUDP\_HEADER;

## 粘包与分包

粘包产生原因：

先说TCP：由于TCP协议本身的机制（面向连接的可靠地协议-三次握手机制）客户端与服务器会维持一个连接（Channel），数据在连接不断开的情况下，可以持续不断地将多个数据包发往服务器，但是如果发送的网络数据包太小，那么他本身会启用Nagle算法（可配置是否启用）对较小的数据包进行合并（基于此，TCP的网络延迟要UDP的高些）然后再发送（超时或者包大小足够）。那么这样的话，服务器在接收到消息（数据流）的时候就无法区分哪些数据包是客户端自己分开发送的，这样产生了粘包；服务器在接收到数据库后，放到缓冲区中，如果消息没有被及时从缓存区取走，下次在取数据的时候可能就会出现一次取出多个数据包的情况，造成粘包现象（确切来讲，对于基于TCP协议的应用，不应用包来描述，而应 用 流来描述），个人认为服务器接收端产生的粘包应该与linux内核处理socket的方式 select轮询机制的线性扫描频度无关。

再说UDP：本身作为无连接的不可靠的传输协议（适合频繁发送较小的数据包），他不会对数据包进行合并发送（也就没有Nagle算法之说了），他直接是一端发送什么数据，直接就发出去了，既然他不会对数据合并，每一个数据包都是完整的（数据+UDP头+IP头等等发一次数据封装一次）也就没有粘包一说了。

分包产生的原因就简单的多：可能是IP分片传输导致的，也可能是传输过程中丢失部分包导致出现的半包，还有可能就是一个包可能被分成了两次传输，在取数据的时候，先取到了一部分（还可能与接收的缓冲区大小有关系），总之就是一个数据包被分成了多次接收。

解决办法：

粘包与分包的处理方法：

我根据现有的一些开源资料做了如下总结（常用的解决方案）：

一个是采用分隔符的方式，即我们在封装要传输的数据包的时候，采用固定的符号作为结尾符（数据中不能含结尾符），这样我们接收到数据后，如果出现结尾标识，即人为的将粘包分开，如果一个包中没有出现结尾符，认为出现了分包，则等待下个包中出现后 组合成一个完整的数据包，这种方式适合于文本传输的数据，如采用/r/n之类的分隔符；

另一种是采用在数据包中添加长度的方式，即在数据包中的固定位置封装数据包的长度信息（或可计算数据包总长度的信息），服务器接收到数据后，先是解析包长度，然后根据包长度截取数据包（此种方式常出现于自定义协议中），但是有个小问题就是如果客户端第一个数据包数据长度封装的有错误，那么很可能就会导致后面接收到的所有数据包都解析出错（由于TCP建立连接后流式传输机制），只有客户端关闭连接后重新打开才可以消除此问题，我在处理这个问题的时候对数据长度做了校验，会适时的对接收到的有问题的包进行人为的丢弃处理（客户端有自动重发机制，故而在应用层不会导致数据的不完整性）；

## NTP时间同步

NTP时间戳格式：指从1900年（1970年需要- 0x83aa7e80）到现在的时间，单位为秒，高32位表示整数部分，低32位表示小数部分，都是大端表示。

系统时钟同步的工作过程如下：

   Device A发送一个NTP报文给Device B，该报文带有它离开Device A时的时间戳，该时间戳为10:00:00am（T1）。

  当此NTP报文到达Device B时，Device B加上自己的时间戳，该时间戳为11:00:01am（T2）。

  当此NTP报文离开Device B时，Device B再加上自己的时间戳，该时间戳为11:00:02am（T3）。

  当Device A接收到该响应报文时，Device A的本地时间为10:00:03am（T4）。

至此，Device A已经拥有足够的信息来计算两个重要的参数：

 NTP报文的往返时延Delay=（T4-T1）-（T3-T2）=2秒。

  Device A相对Device B的时间差offset=（（T2-T1）+（T3-T4））/2=1小时。

## 全双工半双工

单工就是在同一时间只允许一方向另一方传送信息，而另一方不能向一方传送

全双工（Full Duplex）是指在发送数据的同时也能够接收数据，两者同步进行，这好像我们平时打电话一样，说话的同时也能够听到对方的声音。目前的网卡一般都支持全双工。

半双工（Half Duplex），所谓半双工就是指一个时间段内只有一个动作发生，举个简单例子，一条窄窄的马路，同时只能有一辆车通过，当目前有两量车对开，这种情况下就只能一辆先过，等到头儿后另一辆再开，这个例子就形象的说明了半双工的原理。早期的对讲机、以及早期集线器等设备都是基于半双工的产品。随着技术的不断进步，半双工会逐渐退出历史舞台.

全双工以太网使用两对电缆线，而不是像半双工方式那样使用一对电缆线。全双工方式在发送设备的发送方和接收设备的接收方之间采取点到点的连接，这意味着在全双工的传送方式下，可以得到更高的数据传输速度。

## socket

socket有五元组标志协议，本地IP和端口号，目的IP和端口号，五元组相同则相同。

Tcp accect 产生的新socket与源socket的目的IP和端口号不同。

struct socket

{

socket\_state state;

unsigned long flags;

const struct proto\_ops \*ops;

struct fasync\_struct \*fasync\_list;

struct file \*file;

struct sock \*sk;

wait\_queue\_head\_t wait;

short type;

};

其中，struct sock 包含有一个 sock\_common 结构体，而sock\_common结构体又包含有struct inet\_sock 结构体，而struct inet\_sock 结构体的部分定义如下：

struct inet\_sock

{

struct sock sk;

#if defined(CONFIG\_IPV6) || defined(CONFIG\_IPV6\_MODULE)

struct ipv6\_pinfo \*pinet6;

#endif

\_\_u32 daddr; //IPv4的目的地址。

\_\_u32 rcv\_saddr; //IPv4的本地接收地址。

\_\_u16 dport; //目的端口。

\_\_u16 num; //本地端口（主机字节序）。

…………

}

socket 地址结构

struct sockaddr结构用来保存通用的套接字地址信息，其定义如下：

struct sockaddr {

unsigned short sa\_family; /\* 地址族, AF\_xxx \*/

char sa\_data[14]; /\* 14字节的协议地址\*/

};

ipv4 地址

struct sockaddr\_in {  /\*“in” 代表“Internet”\*/

short int sin\_family; /\* Internet地址族\*/

unsigned short int sin\_port; /\* 端口号\*/

struct in\_addr sin\_addr; /\* Internet地址\*/

unsigned char sin\_zero[8]; /\* 填充0（为了保持和struct sockaddr一样大小）\*/

};

ipv6地址

struct sockaddr\_in6 {

\_\_uint8\_t sin6\_len; /\* length of this struct(sa\_family\_t) \*/

sa\_family\_t sin6\_family; /\* AF\_INET6 (sa\_family\_t) \*/

in\_port\_t sin6\_port; /\* Transport layer port # (in\_port\_t) \*/

\_\_uint32\_t sin6\_flowinfo; /\* IP6 flow information \*/

struct in6\_addr sin6\_addr; /\* IP6 address \*/

\_\_uint32\_t sin6\_scope\_id; /\* scope zone index \*/

};

上面结构体大小都相同，一般对socket操作则直接使用sockaddr\_in，传递给内核时转换成sockaddr

1. closesocket（一般不会立即关闭而经历TIME\_WAIT的过程, 而被动关闭端会经历close\_wait阶段等待数据发送完成）后想继续重用该socket：

BOOL bReuseaddr=TRUE;

setsockopt (s,SOL\_SOCKET ,SO\_REUSEADDR,(const char\*)&bReuseaddr,sizeof(BOOL));

2. 如果要已经处于连接状态的soket在调用closesocket后强制关闭，不经历

TIME\_WAIT的过程：

BOOL bDontLinger = FALSE;

setsockopt (s,SOL\_SOCKET,SO\_DONTLINGER,(const char\*)&bDontLinger,sizeof(BOOL));

linger表示socket关闭之后等待一定的时间去发送没有完成的数据，如果超时了，则直接关闭，对主动关闭端容易产生timewait状态

3. 在send(),recv()过程中有时由于网络状况等原因，发收不能预期进行,而设置收发时限：

int nNetTimeout=1000;//1秒

//发送时限

setsockopt (socket，SOL\_S0CKET,SO\_SNDTIMEO，(char \*)&nNetTimeout,sizeof(int));

//接收时限

setsockopt (socket，SOL\_S0CKET,SO\_RCVTIMEO，(char \*)&nNetTimeout,sizeof(int));

4. 在send()的时候，返回的是实际发送出去的字节(同步)或发送到socket缓冲区的字节

(异步);系统默认的状态发送和接收一次为8688字节(约为8.5K)；在实际的过程中发送数据

和接收数据量比较大，可以设置socket缓冲区，而避免了send(),recv()不断的循环收发：

// 接收缓冲区

int nRecvBuf=32\*1024;//设置为32K

setsockopt (s,SOL\_SOCKET,SO\_RCVBUF,(const char\*)&nRecvBuf,sizeof(int));

//发送缓冲区

int nSendBuf=32\*1024;//设置为32K

setsockopt (s,SOL\_SOCKET,SO\_SNDBUF,(const char\*)&nSendBuf,sizeof(int));

5. 如果在发送数据的时，希望不经历由系统缓冲区到socket缓冲区的拷贝而影响

程序的性能：

int nZero=0;

setsockopt (socket，SOL\_S0CKET,SO\_SNDBUF，(char \*)&nZero,sizeof(nZero));

6. 同上在recv()完成上述功能(默认情况是将socket缓冲区的内容拷贝到系统缓冲区)：

int nZero=0;

setsockopt (socket，SOL\_S0CKET,SO\_RCVBUF，(char \*)&nZero,sizeof(int));

7. 一般在发送UDP数据报的时候，希望该socket发送的数据具有广播特性：

BOOL bBroadcast=TRUE;

setsockopt (s,SOL\_SOCKET,SO\_BROADCAST,(const char\*)&bBroadcast,sizeof(BOOL));

8. 在client连接服务器过程中，如果处于非阻塞模式下的socket在connect()的过程中可

以设置connect()延时,直到accpet()被呼叫(本函数设置只有在非阻塞的过程中有显著的

作用，在阻塞的函数调用中作用不大)

BOOL bConditionalAccept=TRUE;

setsockopt (s,SOL\_SOCKET,SO\_CONDITIONAL\_ACCEPT,(const char\*)&bConditionalAccept,sizeof(BOOL));

9 .如果在发送数据的过程中(send()没有完成，还有数据没发送)而调用了closesocket(),以前我们

一般采取的措施是"从容关闭"shutdown(s,SD\_BOTH),但是数据是肯定丢失了，如何设置让程序满足具体

应用的要求(即让没发完的数据发送出去后在关闭socket)？

struct linger {

u\_short l\_onoff;

u\_short l\_linger;

};

linger m\_sLinger;

m\_sLinger.l\_onoff=1;//(在closesocket()调用,但是还有数据没发送完毕的时候容许逗留)

// 如果m\_sLinger.l\_onoff=0;则功能和2.)作用相同;

m\_sLinger.l\_linger=5;//(容许逗留的时间为5秒)

setsockopt (s,SOL\_SOCKET,SO\_LINGER,(const char\*)&m\_sLinger,sizeof(linger));

//获得socket远端地址（只使用于tcp，下同）

getpeername(int, struct sockaddr \*, socklen\_t \*)

//获得socket本地地址

getsockname(int, struct sockaddr \*, socklen\_t \*)

int bind(int, const struct sockaddr \*, socklen\_t)

绑定一个本机的地址到socket，监听socket和连接socket都是本机地址

## Socket level options

int getsockopt(int s,int level,int optname,void \*optval,socklen\_t \*optlen);和

int setsockopt(int s,int level,int optname,void \*optval,socklen\_t \*optlen);

可以配置socket options,其中level必须为sol\_socket.

(以下默认数据均指ios)

SO\_SNDBUF:发送缓存区大小，当缓存区大于此时则发送失败或者阻塞（tcp:128KB,udp:9k）

SO\_RCVBUF:接受缓存区大小，当缓存区大于此时则接受失败或者阻塞(tcp:128k,udp:192k)

SO\_SNDLOWAT:发送最低标准，至少要缓冲区到达此才可以发送(2k),

SO\_RCVLOWAT:接受最低标准，至少要缓冲区到达此才可以接受(1B)

SO\_RCVTIMEO:接受超时，默认0，但是表示一直等待

SO\_SNDTIMEO:发送超时，默认0，但是表示一直等待

SO\_LABEL:本地mac地址

SO\_PEERLABEL:远端mac地址

## 多播/组播

IP隧道（IP tunneling）是将一个IP报文封装在另一个IP报文的技术，这可以使得目标为一个IP地址的数据报文能被封装和转发到另一个IP地址。IP隧道技术亦称为IP封装技术（IP encapsulation）。IP隧道主要用于移动主机和虚拟私有网络（Virtual Private Network），在其中隧道都是静态建立的，隧道一端有一个IP地址，另一端也有唯一的IP地址。

IGMP组管理协议

3.1 加入一个多播组

　　多播的基础就是一个进程的概念（使用的术语进程是指操作系统执行的一个程序），该进程在一个主机的给定接口上加入了一个多播组。在一个给定接口上的多播组中的成员是动态的—它随时因进程加入和离开多播组而变化。

　　这里所指的进程必须以某种方式在给定的接口上加入某个多播组。进程也能离开先前加入的多播组。这些是一个支持多播主机中任何API所必需的部分。使用限定词“接口”是因为多播组中的成员是与接口相关联的。一个进程可以在多个接口上加入同一多播组。

　　Stanford大学伯克利版Unix中的IP 多播详细说明了有关socket API的变化，这些变化在Solaris 2.x和ip（7）的文档中也提供了。

　　这里暗示一个主机通过组地址和接口来识别一个多播组。主机必须保留一个表，此表中包含所有至少含有一个进程的多播组以及多播组中的进程数量。

　　3.2 IGMP 报告和查询

　　多播路由器使用IGMP报文来记录与该路由器相连网络中组成员的变化情况。使用规则如下：

　　1） 当第一个进程加入一个组时，主机就发送一个IGMP报告。如果一个主机的多个进程加入同一组，只发送一个IGMP报告。这个报告被发送到进程加入组所在的同一接口上。

　　2） 进程离开一个组时，主机不发送IGMP报告，即便是组中的最后一个进程离开。主机知道在确定的组中已不再有组成员后，在随后收到的IGMP查询中就不再发送报告报文。

　　3） 多播路由器定时发送IGMP查询来了解是否还有任何主机包含有属于多播组的进程。多播路由器必须向每个接口发送一个IGMP查询。因为路由器希望主机对它加入的每个多播组均发回一个报告，因此IGMP查询报文中的组地址被设置为0.

　　4） 主机通过发送IGMP报告来响应一个IGMP查询，对每个至少还包含一个进程的组均要发回IGMP报告。

　　使用这些查询和报告报文，多播路由器对每个接口保持一个表，表中记录接口上至少还包含一个主机的多播组。当路由器收到要转发的多播数据报时，它只将该数据报转发到（使用相应的多播链路层地址）还拥有属于那个组主机的接口上。

　　图1 3 - 3显示了两个IGMP报文，一个是主机发送的报告，另一个是路由器发送的查询。该路由器正在要求那个接口上的每个主机说明它加入的每个多播组。

组播路由协议 （PIM-SM）

　　众多的组播路由协议中，目前应用最多的协议是 PIM-SM稀疏模式协议无关组播。

　　在PIM-SM域中，运行PIM-SM协议的路由器周期性的发送Hello消息，用以发现邻接的PIM路由器，并且负责在多路访问网络中进行指定路由器(DR)的选举。这里，DR负责为其直连组成员朝着组播分发树根节点的方向发送"加入/剪枝"消息，或是将直连组播源的数据发向组播分发树。

PIM-SM显式的加入机制

　　PIM-SM通过建立组播分发树来进行组播数据包的转发。组播分发树分为两种：以组G的RP为根的共享树（Shared Tree）和以组播源为根的最短路径树（Shortest Path Tree）。

PIM-SM通过显式的加入/剪枝机制来完成组播分发树的建立与维护。如上图所示：

当DR收到一个发自接收端的加入（Join），它就会向着组G的RP方向逐跳组播发出一个（\*，G）加入信息用以加入共享树；

源主机向组发送组播数据时，源的数据被封装在注册消息内，并由其DR 单播至RP，RP再将源的解封装数据包沿着共享树转发到各个组成员；

RP朝着源方向向第一跳路由器发送（S，G）加入信息，用以加入此源的最短路径树，这样源的数据包将沿着其最短路径树不加封装地发送到RP；

当第一个组播数据沿此树到达时，RP向源的DR发送注册-停止消息，以使DR停止注册封装过程。此后，这个源的组播数据不再注册封装，而是先沿着源的最短路径树发送到RP，再由RP 将其沿着共享树转发到各个组成员。

　　当不再需要组播数据时，DR向着组G的RP逐跳组播剪枝消息用以剪枝共享树。

　　PIM-SM中还涉及到其根节点RP的选择机制。PIM-SM域内配置了一个或多个候选自举路由器(Candidate-BSR)。应用一定的规则从中选出自举路由器(BSR)。PIM-SM域中还配置了候选 RP路由器(Candidate-RP)，这些候选 RP将包含了它们地址及可以服务的组播组等信息的包单播至自举路由器。BSR 定期生成包括一系列候选 RP以及相应的组地址的"自举"消息。"自举"消息在整个域中逐跳发送。路由器接收并保存这些"自举"消息。若 DR 从直连主机收到了某组的成员关系报告后，如果它没有这个组的路由项，DR 将使用一个hash算法将组地址映射至一个可以为该组服务的候选 RP。然后 DR 将朝RP方向逐跳组播"加入/剪枝"消息。若 DR从直连主机收到组播数据包，如果它没有这个组的路由项，DR 将使用hash算法将组地址映射至一个可以为该组服务的候选 RP。然后 DR将组播数据封装在注册消息中单播到RP。

## RTMP

握手

C0——>协议号8位

C1——>时间ct1+1536随机数

协议8位<——S0

时间st1+1536随机数<——S1

C2——>时间st1+ct2+S1的1536随机数

时间ct1+st2+C1的1536随机数<——S2

rtmp包由三部分组成，

Basic Header：base head的第一个字节比较重要，前3位fmt位决定Message header的类型和大小，后5位决定Basic Header的大小。

Fmt位后面存储着Chunk ID,但是0、1、2的Chunk ID保留，所以当fmt后5位为0时并不表示chunk id,而是表示chunkid存储在下一个字节，并且偏移为64，即后一个字节等于chunkid-64，当fmt后5位为1时，同样不是表示chunk id而是表示后面2个字节是chunkid -64，

3.1 Message(消息)

这里的Message是指满足该协议格式的、可以切分成Chunk发送的消息，消息包含的字段如下：

Timestamp（时间戳）：消息的时间戳（但不一定是当前时间，后面会介绍），4个字节

Length(长度)：是指Message Payload（消息负载）即音视频等信息的数据的长度，3个字节

TypeId(类型Id)：消息的类型Id，1个字节

Message Stream ID（消息的流ID）：每个消息的唯一标识，划分成Chunk和还原Chunk为Message的时候都是根据这个ID来辨识是否是同一个消息的Chunk的，4个字节，并且以小端格式存储,(应该就相当于标志一个流，一个传输应用流)

chunk header

包含了chunk stream ID（流通道Id）和chunk type（chunk的类型），chunk stream id一般被简写为CSID，用来唯一标识一个特定的流通道，chunk type决定了后面Message Header的格式。Basic Header的长度可能是1，2，或3个字节，其中chunk type的长度是固定的（占2位，注意单位是位，bit），Basic Header的长度取决于CSID的大小,在足够存储这两个字段的前提下最好用尽量少的字节从而减少由于引入Header增加的数据量。RTMP协议支持用户自定义［3，65599］之间的CSID，0，1，2由协议保留表示特殊信息。0代表Basic Header总共要占用2个字节，CSID在［64，319］之间，1代表占用3个字节，CSID在［64，65599］之间，2代表该chunk是控制信息和一些命令信息

RTMP流连接建立过程

C -> S connect 连接请求消息

S-> C set S BW消息

S->C set C BW消息(c每收到CBW字节就需要向S发送报告包)

S->C set **chunk size消息**

**S->C** RTMP\_PT\_INVOKE(invoke some stream action)消息，返回connect 结果

C->S Release Stream

C->S **FCPublish stream**

**C->S Creating stream**

**S->C** RTMP\_PT\_INVOKE(invoke some stream action)消息，返回**Creating stream**结果

c->s **Sending publish command**

**S->C publish result**

## RTSP

RTSP 会话不会绑定到传输层连接，如 TCP。它也可选择使用无连接传输协议，如 UDP。

RTSP 控制的流可能用到 RTP，但 RTSP 操作并不依赖用于传输连续媒体的传输机制。RTSP 在语法和操作上与 HTTP/1.1 类似，因此 HTTP 的扩展机制在多数情况下可加入 RTSP。然而，在很多重要方面 RTSP 仍不同于 HTTP ：

RTSP 引入了大量新方法并具有一个不同的协议标识符：

在大多数情况下，RTSP 服务器需要保持缺省状态，与 HTTP 的无状态相对；

RTSP 中客户端和服务器都可以发出请求；

在多数情况下，数据由不同的协议传输；

RTSP 使用 ISO 10646 （UTF-8）而并非 ISO 8859-1，与当前的国际标准 HTML 相一致；

URI 请求总是包含绝对 URI。为了与过去的错误相互兼容，HTTP/1.1 只在请求过程中传送绝对路径并将主机名置于另外的头字段。

该协议支持如下操作：

从媒体服务器上检索媒体：用户可通过 HTTP 或其它方法提交一个演示描述请求；

媒体服务器邀请进入会议： 媒体服务器可被邀请参加正进行的会议，或回放媒体，或记录部分或全部演示；

将新媒体加到现有演示中：如服务器能告诉客户端接下来可用的媒体内容，对现场直播显得尤其有用

sdp 的格式：

v=<version>

o=<username> <session id> <version> <network type> <address type> <address>

s=<session name>

i=<session description>

u=<URI>

e=<email address>

p=<phone number>

c=<network type> <address type> <connection address>

b=<modifier>:<bandwidth-value>

t=<start time> <stop time>

r=<repeat interval> <active duration> <list of offsets from start-time>

z=<adjustment time> <offset> <adjustment time> <offset> ....

k=<method>

k=<method>:<encryption key>

a=<attribute>

a=<attribute>:<value>

m=<media> <port> <transport> <fmt list>

v = （协议版本）

o = （所有者/创建者和会话标识符）

s = （会话名称）

i = \* （会话信息）

u = \* （URI 描述）

e = \* （Email 地址）

p = \* （电话号码）

c = \* （连接信息）

b = \* （带宽信息）

z = \* （时间区域调整）

k = \* （加密密钥）

a = \* （0 个或多个会话属性行）

时间描述：

t = （会话活动时间）

r = \* （0或多次重复次数）

媒体描述：

m = （媒体名称和传输地址）

i = \* （媒体标题）

c = \* （连接信息 — 如果包含在会话层则该字段可选）

b = \* （带宽信息）

k = \* （加密密钥）

a = \* （0 个或多个媒体属性行）

ijk缺点

read\_thred:start\_on\_prepared为false时，会一直循环等待start;

if (!ffp->start\_on\_prepared) {

while (is->pause\_req && !is->abort\_request) {

SDL\_Delay(100);

}

}

## P2P

NAT对待UDP的实现方式有两种种，分别如下：

* **锥形NAT（Cone NAT）**：象形理解为NAT入口一端有只有一点，出口一端有很多点，像横着的锥形。锥形又细分为一下三种：

1. 全锥形NAT（Full Cone NAT）

完全锥形NAT，所有从同一个内网IP和端口号发送过来的请求都会被映射成同一个外网IP和端口号，并且任何一个外网主机都可以通过这个映射的外网IP和端口号向这台内网主机发送包。

1. 受限锥形NAT（Restricted Cone NAT）

限制锥形NAT，它也是所有从同一个内网IP和端口号发送过来的请求都会被映射成同一个外网IP和端口号。与完全锥形不同的是，外网主机只能够向先前已经向它发送过数据包的内网主机发送包。

1. 端口受限锥形NAT（Port-Restricted Cone NAT）

端口限制锥形NAT，与限制锥形NAT很相似，只不过它包括端口号。也就是说，一台IP地址X和端口P的外网主机想给内网主机发送包，必须是这台内网主机先前已经给这个IP地址X和端口P发送过数据包。

* **对称NAT（Symmetric NAT）:**对称NAT，所有从同一个内网IP和端口号发送到一个特定的目的IP和端口号的请求，都会被映射到同一个IP和端口号。如果同一台主机使用相同的源地址和端口号发送包，但是发往不同的目的地，NAT将会使用不同的映射。此外，只有收到数据的外网主机才可以反过来向内网主机发送包

#### UDP打洞（UDP hole punching）（STUN 基于此）

第三种P2P通信技术，被广泛采用的，名为“P2P打洞“。P2P打洞技术依赖于通常防火墙和cone NAT允许正当的P2P应用程序在中间件中打洞且与对方建立直接链接的特性。

假设客户端A和客户端B的地址都是内网地址，且在不同的NAT后面。A、B上运行的P2P应用程序和服务器S都使用了UDP端口1234，A和B分别初始化了 与Server的UDP通信，地址映射如图所示:

现在假设客户端A打算与客户端B直接建立一个UDP通信会话。如果A直接给B的公网地址138.76.29.7:31000发送UDP数据，NAT B将很可能会无视进入的 数据（除非是Full Cone NAT），因为源地址和端口与S不匹配，而最初只与S建立过会话。B往A直接发信息也类似。

假设A开始给B的公网地址发送UDP数据的同时，给服务器S发送一个中继请求，要求B开始给A的公网地址发送UDP信息。A往B的输出信息会导致NAT A打开 一个A的内网地址与与B的外网地址之间的新通讯会话，B往A亦然。一旦新的UDP会话在两个方向都打开之后，客户端A和客户端B就能直接通讯， 而无须再通过引导服务器S了。

UDP打洞技术有许多有用的性质。一旦一个的P2P链接建立，链接的双方都能反过来作为“引导服务器”来帮助其他中间件后的客户端进行打洞， 极大减少了服务器的负载。应用程序不需要知道中间件具体是什么（如果有的话），因为以上的过程在没有中间件或者有多个中间件的情况下 也一样能建立通信链路。

## 中继（TURN基于此）

直接采用公网服务器打洞的方式，伪P2P，

## linux多线程

linux下**进程间通信**的几种主要手段：      
   **1.管道（Pipe）及有名管道（named pipe）**：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信；      
   **2.信号（Signal）**：信号是比较复杂的通信方式，用于通知接受进程有某种事件生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期 信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上， 该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，sigaction函数重新实现了signal函数）；      
   **3.报文（Message）队列（消息队列）**：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。  
   **4.共享内存**：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针其他通信机制运行效率较低设计的。往往与其它通信机制，如信号量结合使用， 来达到进程间的同步及互斥。     
   **5.信号量（semaphore）**：主要作为进程间以及同一进程不同线程之间的同步手段。               
   **6.套接字（Socket）**：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix 系统上：Linux和System V的变种都支持套接字。

# MPEG

MPEG是一套标准，包括MPEG-1，MPEG-2，MPEG-4。其中MPEG-4封装标志MP4,压缩标准包括mpeg4p2和H264/AVC（Advanced Video Codec/先进视频编码）,

数字信号实际传送的是数据流，一般数据流包括以下三种：

**ES流：**也叫基本码流（elementary stream），包含视频、音频或数据的连续码流。  
**PES流：**也叫打包的基本码流，是将基本的码流ES流根据需要分成长度不等的数据包，并加上包头就形成了打包的基本码流PES流。

**TS流：**也叫传输流，是由固定长度为188字节的包组成，含有独立时基的一个或多个节目，适用于误码较多的环境。

**PS流：**TS流与PS流的区别在于TS流的包结构是固定长度的，而PS流的包结构是可变长度。 PS包与TS包在结构上的这种差异，导致了它们对传输误码具有不同的抵抗能力，因而应用的环境也有所不同。TS码流由于采用了固定长度的包结构，当传输误码破坏了某一TS包的同步信息时，接收机可在固定的位置检测它后面包中的同步信息，从而恢复同步，避免了信息丢失。而PS包由于长度是变化的，一旦某一PS包的同步信息丢失，接收机无法确定下一包的同步位置，就会造成失步，导致严重的信息丢失。因此，在信道环境较为恶劣，传输误码较高时，一般采用TS码流；而在信道环境较好，传输误码较低时，一般采用PS码流由于TS码流具有较强的抵抗传输误码的能力，因此目前在传输媒体中进行传输的MPEG-2码流基本上都采用了TS码流的包格式。

# [H264基本概念之 宏块、片和片组](http://www.cnblogs.com/yinxiangpei/articles/2817675.html)

这几个概念对比音频信号处理可是全新的，下面简要介绍一下定义和作用：

1、宏块（Macro Block）：一个编码图像首先要划分成多个块（4x4 像素）才能进行处理，显然宏块应该是整数个块组成，通常宏块大小为16x16个像素。宏块分为I、P、B宏块，I宏块只能利用当前片中已解码的像素作为参考进行帧内预测；P宏块可以利用前面已解码的图像作为参考图像进行帧内预测；B宏块则是利用前后向的参考图形进行帧内预测；

2、片（Slice）：一帧视频图像可编码成一个或者多个片，每片包含整数个宏块，即每片至少一个宏块，最多时包含整个图像的宏块。

片的目的：为了限制误码的扩散和传输，使编码片相互间保持独立。片共有5种类型：I片（只包含I宏块）、P片（P和I宏块）、B片（B和I宏块）、SP片（用于不同编码流之间的切换）和SI片（特殊类型的编码宏块）。

H.264 的功能分为两层：视频编码层（VCL, Video Coding Layer）和网络提取层（NAL,Network Abstraction Layer）。VCL 数据即编码处理的输出，它表示被压缩编码后的视频数据序列。

http://blog.csdn.net/joeblackzqq/article/details/31377939

# NAL存在三种封装形式：SODB, RBSP, EBSP.

SODB：数据比特串，即编码后的最原始的数据；

RBSP：原始字节序列载荷，即在SODB的后面添加了trailing bits，即一个bit 1和若干个bit 0，以便字节对齐；

EBSP：扩展字节序列载荷，即在RBSP的基础上添加了仿校验字节0x03.----annexb

二、联系、区别与作用

SODB是编码收的原始数据，经过封装后为RBSP，RBSP是NAL单元的数据部分的封装格式。在NAL内部为了防止与起始码竞争，从而引入填充字节0x03，这样便形成了EBSP。

1、RBSP的形成过程

如果SODB的内容是空的，那么RBSP的内容也是空的；

否则，RBSP的第一个字节取自SODB的第1到第8个比特，RBSP字节内部按照从左到右从高到低的顺序排列。以此类推，RBSP中的每个字节都直接取自SODP的相应比特。RBSP的最后一个字节包含SODB的最后几个比特，以及trailing bits。其中，trailing bits的第一个比特为1，其余的比特为0，保证字节对齐。最后，在结尾添加0x0000，即CABAC ZERO WORD，从而形成RBSP。

2、EBSP的形成过程

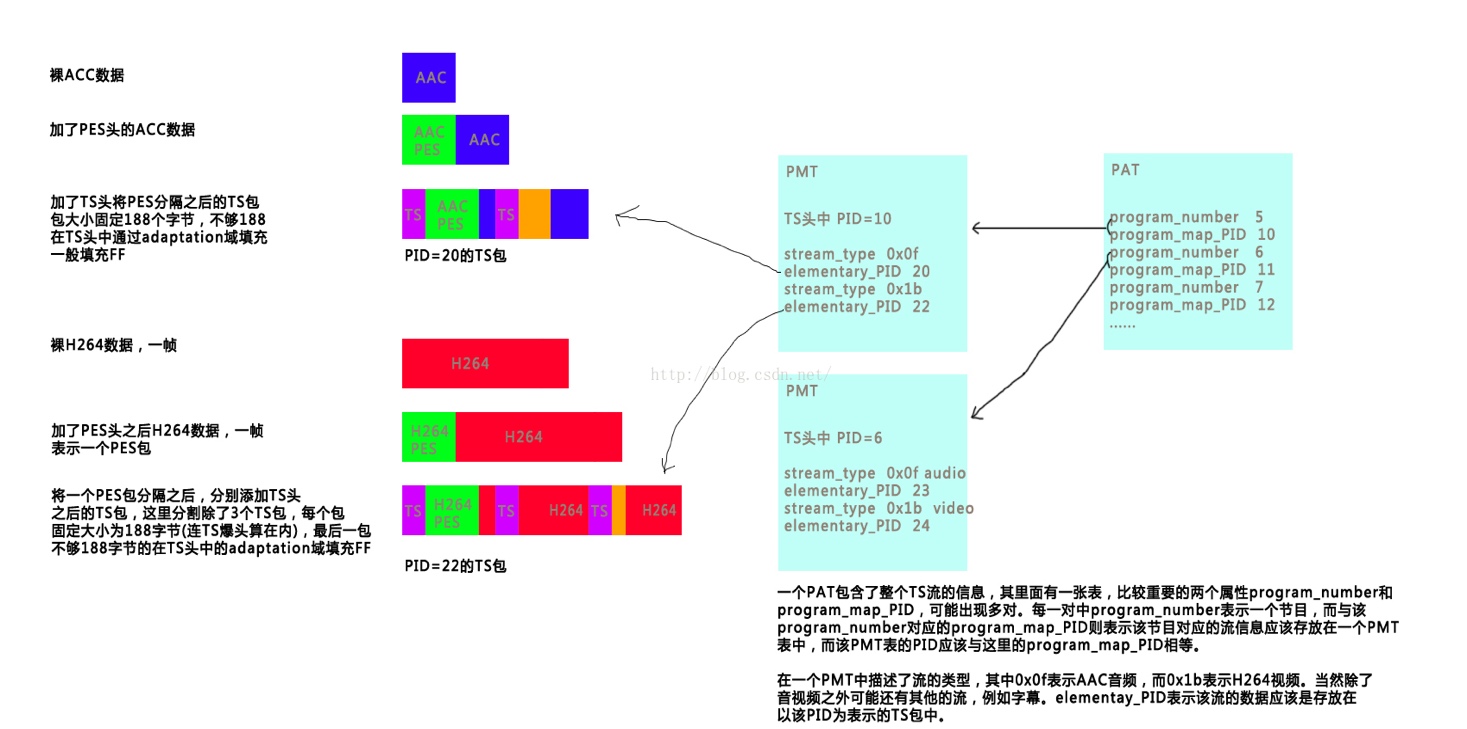
NALU数据+起始码就形成了AnnexB格式，起始码包括两种，0x00000001和0x000001。为了不让NALU的主体与起始码之间产生竞争，在对RBSP进行扫描时，如果遇到连续的两个0x00字节，则在该两个字节后面添加一个0x03字节。在解码的时候将该0x03字节去掉，也成为脱壳操作。

通过该种方式形成EBSP，这需要将近两倍的整帧图像码流大小。为了减少存储器需求，在每个macroblock结束后，即检查该macroblock的SODB的起始码竞争问题，并保留SODB的最后两个字节的零字节个数，以便与下一个macroblock的SODB的开始字节形成连续的起始码竞争检测。对一帧图像的最后一个macroblock，先添加结尾停止bit，在检查起始码竞争。

注：起始码包括两种情况，即4字节和3字节。在SPS、PPS和Access Unit的第一个NALU使用4字节起始码，其余情况均使用3字节起始码。

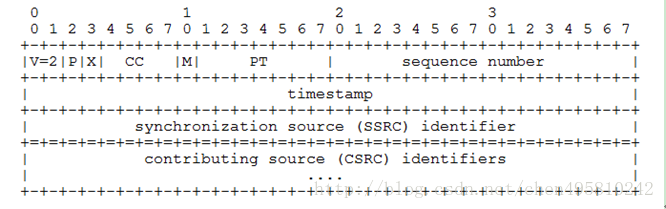
起始码：如果NALU 对应的Slice 为一帧的开始，则用4 字节表示，即0x00000001；否则用3 字节表示，0x000001。

# 阐述H264裸流与TS流之间的关系(一)



# YUV420数据格式

# RTP



1)        V：RTP协议的版本号，占2位，当前协议版本号为2

2)        P：填充标志，占1位，如果P=1，则在该报文的尾部填充一个或多个额外的八位组，它们不是有效载荷的一部分。

3)        X：扩展标志，占1位，如果X=1，则在RTP报头后跟有一个扩展报头

4)        CC：CSRC计数器，占4位，指示CSRC 标识符的个数

5)        M: 标记，占1位，不同的有效载荷有不同的含义，对于视频，标记一帧的结束；对于音频，标记会话的开始。

6)        PT: 有效荷载类型，占7位，用于说明RTP报文中有效载荷的类型，如GSM音频、JPEM图像等,在流媒体中大部分是用来区分音频流和视频流的，这样便于客户端进行解析。

7)        序列号：占16位，用于标识发送者所发送的RTP报文的序列号，每发送一个报文，序列号增1。这个字段当下层的承载协议用UDP的时候，网络状况不好的时候可以用来检查丢包。同时出现网络抖动的情况可以用来对数据进行重新排序，序列号的初始值是随机的，同时音频包和视频包的sequence是分别记数的。

8)        时戳(Timestamp)：占32位，必须使用90 kHz 时钟频率。时戳反映了该RTP报文的第一个八位组的采样时刻。接收者使用时戳来计算延迟和延迟抖动，并进行同步控制。

9)        同步信源(SSRC)标识符：占32位，用于标识同步信源。该标识符是随机选择的，参加同一视频会议的两个同步信源不能有相同的SSRC。

10)    特约信源(CSRC)标识符：每个CSRC标识符占32位，可以有0～15个。每个CSRC标识了包含在该RTP报文有效载荷中的所有特约信源。

注：基本的RTP说明并不定义任何头扩展本身，如果遇到X=1，需要特殊处理

# c++语言

## 关键字

explicit：用于只有一个参数的函数，表示参数不能隐式转换

## 智能指针

auto\_ptr<T> 指针只能被同一个auto\_ptr拥有， 允许赋值操作，但是右值会释放该指针。不能作为容器的成员，不能指向数组

unique\_ptr<T> 与auto\_ptr区别是不能直接赋值，需要转换使用权，需要显示使用move函数

scoped\_ptr<T> **永远不能被复制或被赋值**

## 知识点

1. 当没有构造函数时会有默认无参构造函数，有任意构造函数时就没有改默认函数

# webrtc

download\_from\_google\_storage错误：

使用脚本：

#!/bin/sh

keyStr="Failed to fetch file"

preStr="https://storage.googleapis.com"

input=$1

while read line

do

if [[ $line =~ $keyStr ]];then

line=${line#\*/}

line=$preStr$line

arry=($line)

source=${arry[0]}

path=${arry[2]}

path=${path%?}

if [ -f "$path" ];then

echo "exist file:$path"

continue

fi

echo "download $source to $path"

curl $source -o $path

else

continue

fi

done < $input

，然后将编译日志放到一个文件后，传递路径给该脚本作为参数运行。

gn gen out/ios --args='target\_os="ios" target\_cpu="arm64"' --complete\_static\_lib=true --ide=xcode

生成xcode静态库命令

## Sigslot：信号槽

所有的槽必须继承has\_slots。

signal\_with\_thread\_policy提供信号，sigslot.connect连接信号与槽，将slot对象与方法封存储到signal中，不同的signal具有不同的槽参数类型，所有连接时注意参数对应。

## talk\_base::Event

event.h/event.cc文件中只有talk\_base::Event类。

该类主要实现了跨平台的Win32 Event功能（正如前言中所说，本文假定读者已经很熟悉Win32平台的各种组件，如有不明白的地方可以参考MSDN）。talk\_base::Event类的各个成员函数与Win32 Event所提供的API几乎一致，所以不再多做解释。

在Linux系统中，WebRTC使用了mutex和条件变量来实现Event的功能。首先，我将对Win 32 API和pthread API做一下类比：

Event::Event(bool manual\_reset, bool initially\_signaled)

创建event，

manual：wait后是否手动取消event状态

initially\_signaled：初始状态是否event状态

void Event::Set()

触发event状态，解除所有等待 event的等待

void Event::Reset()

设置为非event状态

bool Event::Wait(int milliseconds)

等待event状态的到来，最多等待milliseconds时长，如果当前是event状态，则直接返回。

## CriticalSection:临界区

CriticalSection()

创建临界区

CriticalSection::Enter()

进入临界区，相当于可递归版本的pthread\_mutex\_lock()，如果自己进入过，则可以继续进入，否则等待锁释放

CriticalSection::TryEnter()

相当于可递归版本的pthread\_mutex\_trylock()

CritScope, TryCritScope:

采用构造函数和析构函数对CriticalSection的Enter和TryEnter进行封装

SharedExclusiveLock

共享互斥锁，有时也被称为单写多读锁。   
简单点的应用场景就是：一个写（LockExclusive）多个读（LockShared）。

LockExclusive，UnlockExclusive：加锁结束私有的锁

LockShared，UnlockShared：加锁解锁共有的锁。

即有LockExclusive时，其他所有的锁都要等待，LockShared时其他LockShared不需要等待。

## talk\_base::MessageHandler

talk\_base::MessageHandler类的主要功能是定义了消息处理器的基本数据结构，子类在继承了该类之后要重载OnMessage函数，并实现消息响应的逻辑。

talk\_base::FunctorMessageHandler : public MessageHandler

 只是封装了一个函数

FunctorMessageHandler(const FunctorT& functor)

构造某个函数的Handler

OnMessage(Message\* msg)

执行函数

result()

返回执行的结果

talk\_base::FunctorMessageHandler类的主要功能是将一个函数投递到目标线程执行。该类主要通过functor模板实现（熟悉C++的读者应该不会对它陌生，不熟悉的可以上网查找），而且定义了一个针对返回值类型为void的函数的特化版本（模板的特化和偏特也应该是一个C++程序员掌握的一个知识点，该语法有些难度）。用户不需要创建或者继承talk\_base::FunctorMessageHandler类，仅需调用talk\_base::Thread::Invoke函数就能使用它的功能。

## messagequeue

messagequeue.h/messagequeue.cc文件是多路信号分离器的重要组成部分。它实现了消息一个完整地消息队列，该队列包括立即执行消息队列、延迟执行消息队列和具有优先级的消息队列。其中，talk\_base::MessageQueue类也是talk\_base::Thread类的基类。所以，所有的WebRTC的线程都是支持消息队列的。

#### MessageQueueManager

talk\_base::MessageQueueManager类是一个全局单例类。这个类看似比较复杂，但是功能其实非常简单——仅仅为了在所有的talk\_base::MessagerQueue中删除与指定的talk\_base::MessageHandler相关的消息。WebRTC的消息队列在发送消息的时候要指定消息处理器（talk\_base::MessageHandler）。如果某个消息处理器被析构，那么与之相关的所有消息都将无法处理。所以，创建了这个全局单例类来解决这个问题（见talk\_base::MessageHandler析构函数）。

#### 4.2 MessageData

这一节的内容将包括talk\_base::MessageData类以及多个它的子类和几个工具函数。这些类和函数都很简单，所以就不介绍代码和原理，仅仅罗列一下它们的功能。

4.2.1 talk\_base::MessageData

定义了基类，并将析构函数定义为虚函数。

4.2.2 talk\_base::TypedMessageData

使用模板定义的talk\_base::MessageData的一个子类，便于扩展。

4.2.3 talk\_base::ScopedMessageData

类似于talk\_base::TypedMessageData，用于指针类型。在析构函数中，自动对该指针调用delete。

4.2.4 talk\_base::ScopedRefMessageData

类似于talk\_base::TypedMessageData，用于引用计数的指针类型。

4.2.5 talk\_base::WrapMessageData函数

模板函数，便于创建talk\_base::TypedMessageData

4.2.6 talk\_base::UseMessageData函数

模板函数，用于将talk\_base::TypedMessageData中的Data取出

4.2.7 talk\_base::DisposeData

这是一个很特殊的消息，用以将某个对象交给消息引擎销毁。可能的用途有2个：1. 有些函数不便在当前函数范围内销毁对象，见范例talk\_base::HttpServer::Connection::~Connection；2.某对象属于某一线程，因此销毁操作应该交给所有者线程（未见范例）。WebRTC用户不需要自行使用该类，调用talk\_base::MessageQueue::Dispose函数即可使用它的功能。

以上7个类或函数的实现非常简单，有C++使用经验的读者非常容易就能理解（标准库中就有相似的类）

#### 4.3 Message

这一节将简单介绍一下3个类：talk\_base::Message、talk\_base::DelayedMessage和talk\_base::MessageList。

4.3.1 talk\_base::Message

定义了消息的基本数据结构。

  Location posted\_from：从哪里抛出的，便于调试

MessageHandler \*phandler：消息处理函数

MessageData \*pdata：数据

uint32\_t message\_id：消息id

int64\_t ts\_sensitive：敏感时间，如果大于0，则消息触发时间大于ts\_sensitive时，会发出警告日志

4.3.2 talk\_base::DelayedMessage

定义了延迟触发消息的数据结构。在talk\_base::MessageQueue中，延迟消息被存放在以talk\_base::DelayedMessage::msTrigger\_排序（talk\_base::DelayedMessage类定义了operator<）的队列中。如果2个延迟消息的触发时间相同，响应顺序按先进先出原则。

这里我将简单介绍一下各个成员变量的用途：

cmsDelay\_：延迟多久触发消息，仅作调试使用

msTrigger\_：触发消息的时间

num\_：添加消息的时间

msg\_：消息本身

在使用延迟消息时，不需要自行构建talk\_base::DelayedMessage实例。直接调用talk\_base::MessageQueue::PostDelayed或者talk\_base::MessageQueue::PostAt函数即可。

4.3.3 talk\_base::MessageList

消息列表，定义为std::list<talk\_base::Message>

## voice\_engine

**主要类：**

1. class AudioFrame;
2. class EchoCancellation;
3. class EchoControlMobile;
4. class GainControl;
5. class HighPassFilter;
6. class LevelEstimator;
7. class NoiseSuppression;
8. class VoiceDetection;

**主要名词解释：**

APM[（AudioProcessing module）:音频处理模块](http://blog.csdn.net/elesos/article/details/53446731)

AudioFrame：主要记录了通道基本信息，数据，VAD标志时间戳，采样频率，信道数等。

EchoCancellation：回声消除模块（AEC），在使用外置扬声器的时候应该使用，有些使用耳麦通讯的情况也会存在回声（因为麦克风与扬声器有空间或者电的弱耦合），如果影响了通话也应该开启。

AudioEchoControlMobile：回声抑制模块（AES），这个模块和回声消除模块功能相似，但是实现方法不一样。该模块使用定电话实现，运算量也远远小于回声消除模块。非常适合移动平台使用。但是对语音损伤大。

AudioGainControl：增益控制模块（AGC），这个模块使用了语音的特征对系统硬件音量和输出的信号大小进行调节。硬件上可以控制输入音量。软件上只能调节原来信号的幅度，如果对原来就已经破音的信号，或者本来输入就比较小的信号就无能为力了。

HighPassFilter：高通滤波器，抑制不需要的低频信号。内部是使用定化的IIR实现的。可以根据需要修改参数选择相应的截止频率。对于某些有工频干扰的设备需要使用高通滤波器。

LevelEstimator：估计信号的能量值。

NoiseSuppression：噪声抑制模块（NS/SE），该模块一般应用在有环境噪声的情况，或者是麦克风采集到的数据有明显噪声的情况。

VoiceDetection：语音激活检测模块（VAD），该模块用于检测语音是否出现。用于编解码以及后续相关处理。

Far-end:远端流

Far-end获得数据后主要有4个步骤的处理。

1、判断是否是32k信号，采取相应的分频策略；

2、AEC流程，记录AEC中的farend及其相关运算；

3、AES流程，记录AES中的farend及其相关运算；

4、AGC流程，计算farend及其相关特征。

Near-end：近端流

Near-end包括七个步骤：

1、分频；

2、高通滤波；

3、硬件音量控制；

4、AEC；

5、NS；

6、AES；

7、VAD；

8、AGC；

9、综合。

一，实例化和配置

AudioProcessing\* apm = AudioProcessing::Create(0); //这里的0指的是channelID，只是一个标注那个通道的表示

apm->level\_estimator()->Enable(true)；//启用  重试次数估计   组件

apm->echo\_cancellation()->Enable(true);//启用回声消除组件

apm->echo\_cancellation()->enable\_metrics(true);//

apm->echo\_cancellation()->enable\_drift\_compensation(true);//启用时钟补偿模块（声音捕捉设备的时钟频率和

                                                                                                                    播放设备的时钟频率可能不一样）

apm->gain\_control()->Enable(true);//启用增益控制组件，client必须启用哦！

apm->high\_pass\_filter()->Enable(true);//高通过滤器组件，过滤DC偏移和低频噪音，client必须启用

apm->noise\_suppression()->Enable(true);//噪声抑制组件，client必须启用

apm->voice\_detection()->Enable(true);//启用语音检测组件，检测是否有说话声

apm->voice\_detection()->set\_likelihood( VoiceDetection::kModerateLikelihood);//设置语音检测的阀值，阀值越大，

                                                语音越不容易被忽略，同样一些噪音可能被当成语音。

apm->Initialize();//保留所有用户设置的情况下重新初始化apm的内部状态，用于开始处理一个新的音频流。第一个流

                        创建之后不一定需要调用此方法。

二，工作流程

AudioProcessing也是事件驱动的，事件分为初始化事件、捕捉音频事件、渲染音频事件。

初始化事件：

apm->set\_sample\_rate\_hz(sample\_rate\_hz)；//设置本地和远程音频流的采样率。

apm->echo\_cancellation()->set\_device\_sample\_rate\_hz()；//设置音频设备的采样率，

我们假定音频采集和播放设备采用同样的采样率。（drift组件启用时必须调用）。

 apm->set\_num\_channels(num\_capture\_input\_channels, num\_capture\_output\_channels);//设置

本地和远程音频流的通道数。

播放事件：

apm->AnalyzeReverseStream(&far\_frame));//分析远端音频流的10ms的frame数据，这些数据为回声抑制提供参考。

（启用回声抑制的时候需要调用）

捕捉事件：

apm->gain\_control()->set\_stream\_analog\_level(capture\_level);

apm->set\_stream\_delay\_ms(delay\_ms + extra\_delay\_ms);//设置本地和远端音频流之间的延迟，单位毫秒。

这个延迟是远端音频流和本地音频流之间的时差，计算方法为：

delay = (t\_render - t\_analyze) + (t\_process - t\_capture)；在调用前记录即可。

其中

t\_analyze是远端音频流交给AnalyzeReverseStream（）方法的时间；

t\_render是与刚才同样的远端音频frame的播放时间；

elesos注：render-aanalyze感觉就是远端音频处理的时间，从其开始处理到最终渲染的这一段时间。而

process-capture感觉就是近端，从采集到开始处理的这段时间。时差就是双方的和。

t\_capture是本地音频frame捕捉的时间；

t\_process是同样的本地音频frame被交给ProcessStream（）方法的时间。

 apm->echo\_cancellation()->set\_stream\_drift\_samples(drift\_samples);//

设置音频设备捕捉和播放的采样率的差值。（drift组件启用时必须调用）

int err = apm->ProcessStream(&near\_frame);//处理音频流，包括各个环节的处理。（如增益调节、回声消除、噪声

抑制、语音检测、高通过率等，没有解码哦！是针对pcm数据做处理的）

capture\_level = apm->gain\_control()->stream\_analog\_level();//模拟模式下，必须在ProcessStream之后调用此方法，

获取新的音频HAL的推荐模拟值。

stream\_has\_voice =apm->voice\_detection()->stream\_has\_voice();//检测是否有语音，必须在ProcessStream之后调用

此方法，elesos注：可参考Audio\_processing.h

ns\_speech\_prob = apm->noise\_suppression()->speech\_probability();//返回内部计算出的当前frame的人声优先概率。

三，释放和资源回收

AudioProcessing::Destroy(apm);  
apm = NULL;

代码分析

webrtc编码之前的处理在TransmitMixer中完成

int32\_t PrepareDemux函数：主要通过AudioProcessing处理各种数据AEC,AGC等处理，另外可以播放文件声音，录制声音道文件等等。

Void ProcessAndEncodeAudio函数：主要分发到下一级做编码等处理

# **NACK与重传**

NACK（negative-acknowledge character)，就是否定应答，与之对应的是TCP中的ACK（acknowledge character）。我们知道在TCP中，接收端对于收到的包都要进行应答即发送ACK包，发送端通过接受ACK包，来确定发送的包已经被成功接收，以此来保证网络包的传输可靠。RTP协议不来保证传输的可靠性，所以接收端也就不会发送ACK包，但是对于‘丢失’的包，没有收到的包，如果觉得这个包比较重要，可以给对端发送NACK包，来告诉对端，这个包我没有收到，你如果‘还有’的话，就给我重新发送一遍（Retransmission）。我们可以理解为，对于ACK机制来说，发送端没有收到ACK，我就重新发送，是一种push的方式，对于NACK来说，接收端没有收到，主动请求，是一种pull的模式。如果对于每个‘丢失’的包，都发起NACK的话，那也就和ACK没有大的不同了，但是正如前面描述中各种模棱两可的话所说，我们可以根据需要、根据策略，选择性的发起丢包重传，甚至可以选择完全不进行重传，而是通过其他的容错机制来进行保障，因为发起重传，也就意味着接收端要等待这个包，等待就会增加延时。比如在WebRTC的实现中，对于音频的传输，在收发两端进行协商的时候就约定了，音频包就不进行重传了，丢了就丢了吧。说到这里插一句话，我们这里说到的网络反馈和控制的机制，基本都是要收发两端进行协商的，比如这里说的重传，如果要支持的话，对于发送端来说，就需要一个发送端缓存，发送出去的包，需要暂留一段时间，不然即便收到了NACK包，也没办法重传了。对于接收端来说，我们前面说到的丢包，都加了引号，其实是说，这个“丢包”，有可能是真的丢了，也有可能是顺序乱了。我们知道网络包的到达，有时候不是一定严格按照包序到达的，我收到了很多较新的包了，某个旧的包还没有收到，我就认为它丢了，也没准儿一会儿它又到了。那么对于接收端，我要等多久，才认为它‘丢包’了呢？可以是根据乱序的偏移来决定发起重传，也可以根据预计到达时间已经超过一定时间了来发起重传。发起重传后，我得等这个重传的包吧，那我等多久呢？一般是等待一个rtt＋jitter的时常，如果这个时间内还没有到达，可以选择丢弃不要了，也可以选择再次发起重传，可是也不能一直等下去吧，这个就要看下面要说到的jitterbuffer。

# **jitterbuffer**

在网络传输中总是存在着抖动，由于网络拥塞、路由变化等等原因，导致网络包不是按时间均匀到达的，还有上面说的包不是按照包序有序到达，这时候就需要一个缓冲区，来缓存网络包，进行等待和排序，将这个网路‘抖动’给过滤掉，从而可以稳定有序的将包再发给后面的处理逻辑。假如是个固定长度的buffer，那么当缓冲区满了的时候，如果一个包还没有收到，那么就不会再等待这个包了。试想，在一定的网络抖动条件下，这个buffer设置的过小，就会导致发起的重传比较多，如果设置的过大，必然又会增加jitterbuffer中等待的延时，从而导致整个音视频通信延时的增加。所以理想的做法是，根据网络的状况，按照一定的策略，动态的来调整jitterbuffer的大小。比如在网络比较好的时候，设置小一点，这时对于‘丢包’尽快的发起重传，我们也是可以期望在较快时间内就得到的，从而降低延时；网络不好的时候，可以适当设置大一点，因为这时本来网络条件就较差，更多的重传只能是更加恶化网络环境。当然，仅仅是动态的jitterbuffer也是无法完全解决这个问题的，根本上还是应该根据网络状况来调整发送的码率，网络环境差的情况下，主动降低码率，通过减少网络负荷，来保证传输的流畅性。这就是下面要说的带宽自适应。

# **带宽自适应**

主要包括REMB／TMMBR／TMMBN

TMMBR是Temporal Max Media Bitrate Request，表示临时最大码率请求。表明接收端当前带宽受限，告诉发送端控制码率。

REMB是ReceiverEstimated Max Bitrate，接收端估计的最大码率。

TMMBN是Temporal Max Media Bitrate Notification

带宽自适应是指在音视频的收发过程中，根据网络带宽的变化，自动的来调整发送码率，来适应带宽的变化。在带宽足够的情况下，增加帧率和码率，提高音视频的质量，带来更好的通信体验。在带宽不足的情况下，主动降低码率或者帧率，保证通信的流畅性和可用性，也是带来更好的通信体验。带宽自适应的核心，就是如何准确的估计带宽。WebRTC在实现带宽自适应时采用了Google提出一个称为REMB（Receiver Estimated Max Bitrate，最大接收带宽估计）的带宽估计算法。这个算法的大概思路是在接收端根据丢包率或者延时情况维护一个状态机（参见下图）。以根据丢包率为例，在判断为overuse时，就根据一定的系数减少当前remb值，当判断为underuse时又根据增加系数来增加remb值；然后将这个值通过rtcp包发送给发送端，发送端根据该值来动态的调整码率。在发送端，WebRTC的实现中在调整码率时还会参考rtcp中的丢包率，当然这些我们都可以根据自己的策略进行修改。关于接收端（服务端）REMB带宽估计的实现，则是需要自己来实现，可以实现得更复杂和完善，来实现更佳准确的带宽估计，更佳敏捷的带宽自适应。

# **前向纠错编码**

最后我们再来说一下FEC(Forward Error Correction，前向纠错)，也叫前向纠错码，是通过增加冗余来增强容错性的一种方法。没有FEC的情况下，当接收端发现有包丢失时，需要通过发送NACK来发起重传，前文中我们已经说过，重传是会影响延时性的，而FEC则是在发送通过纠删码来增加一些冗余数据，这样接收端在数据丢失的情况下可以根据冗余数据来重建丢失的数据。通过增加FEC来避免和减少NACK／重传，从而减少丢包导致的延时。其缺点也是显然的，增加的FEC冗余数据占据了有效带宽，这又是一种取舍了，不过FEC的冗余度也是可以根据网络状况来动态的调整的。我们这里说的FEC是整个RTP传输层的，在使用WebRTC的过程中，还会发现，在WebRTC使用的音频opus编码中还有一个称为inband的FEC，就是在opus的编码过程中增加一定的FEC冗余，我想这也就是为什么我们前面说到的WebRTC中音频在协商的时候就不进行NACK重传了。

# **OpenGL**

Opengl分为三块

1. 构建程序，编译shader。有点类似c的编译过程，先是读取shader编译成类似.o文件，然后是连接成program，然后就可以使用程序，并对里面的数据赋值了
2. 构建fbo，fbo只是管理各种buffer，组成一帧图片，不会保存buffer。创建好的renderbuffer，depthbuffer都可以挂在fbo上，纹理也可以。
3. 构建顶点数据，并传入顶点数据，然后就可以draw了。

# **C程序编译过程浅析**

其实编译（包括链接）的命令：gcc hello.c 可分解为如下4个大的步骤：

* + **预处理**(Preprocessing)
  + **编译**(Compilation)
  + **汇编**(Assembly)
  + **链接**(Linking)

**1.       预处理(Preproceessing)**

**预处理的过程主要处理包括以下过程：**

* 将所有的#define删除，并且**展开所有的宏定义**
* **处理**所有的**条件预编译指令**，比如#if #ifdef #elif #else #endif等
* **处理#include** 预编译指令，将被包含的文件插入到该预编译指令的位置。
* **删除所有注释**“//”和”/\* \*/”.
* **添加行号和文件标识**，以便编译时产生调试用的行号及编译错误警告行号。
* 保留所有的**#pragma编译器指令**，因为编译器需要使用它们

**2.       编译(Compilation)**

* 编译过程就是把预处理完的文件进行一系列的词法分析，语法分析，语义分析及优化后生成相应的汇编代码。
* **编译过程可分为6步：扫描（词法分析）、语法分析、语义分析、源代码优化、代码生成、目标代码优化。**
* 词法分析：扫描器（Scanner）将源代的字符序列分割成一系列的记号（Token）。lex工具可实现词法扫描。
* 语法分析：语法分析器将记号（Token）产生语法树（Syntax Tree）。yacc工具可实现语法分析(yacc: Yet Another Compiler Compiler)。
* 语义分析：静态语义（在编译器可以确定的语义）、动态语义（只能在运行期才能确定的语义）。
* 源代码优化：源代码优化器(Source Code Optimizer)，将整个语法书转化为中间代码（Intermediate Code）（中间代码是与目标机器和运行环境无关的）。中间代码使得编译器被分为前端和后端。编译器前端负责产生机器无关的中间代码；编译器后端将中间代码转化为目标机器汇编码。
* 目标代码生成：代码生成器(Code Generator).
* 目标代码优化：目标代码优化器(Target Code Optimizer)。
* 最后得到的目标代码是汇编文件，对如果开启bitcode后，中间码信息会以某种方式附加到库中，所以直接用汇编文件编译连接的的包是无法完整的完成bitcode打包的

**3.       汇编(Assembly)**

* 汇编器是将汇编代码转变成机器可以执行的命令，每一个汇编语句几乎都对应一条机器指令。汇编相对于编译过程比较简单，根据汇编指令和机器指令的对照表一一翻译即可。最后得到的是.o文件

**4.       链接(Linking)**

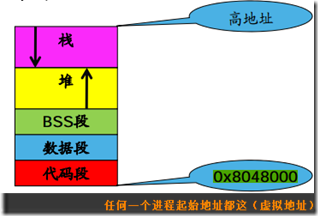
* 通过调用链接器ld来链接程序运行需要的一大堆目标文件，以及所依赖的其它库文件，最后生成可执行文件。
* 链接的主要内容是把各个模块之间相互引用的部分处理好，使得各个模块之间能够正确地衔接。
* 链接的主要过程包括：地址和空间分配（Address and Storage Allocation），符号决议（Symbol Resolution），重定位（Relocation）等。
* **链接分为静态链接和动态链接。**
* **静态链接**是指在编译阶段直接把静态库加入到可执行文件中去，这样可执行文件会比较大。
* 而**动态链接**则是指链接阶段仅仅只加入一些描述信息，而程序执行时再从系统中把相应动态库加载到内存中去。

# **Linux内存分布**

当一段程序被编译成为一个可执行的文件时，这个时候它已经被划分成代码段、数据段、栈段、.bss段、堆等部分。

各段的作用是：

1. 代码段（.text）：代码，全局常量（const），只读变量和字符串常量（有可能在代码段，一般被放在只读数据***".rodata"***段，还有可能就在“.data”段）。  
   2.数据段（.data）：全局变量（初始化以及未初始化的）、静态变量（全局的和局部的、初始化的以及未初始化的）。数据段包括初始化的数据和未初始化的数据（BSS）两部分。BSS段存放的是未初始化的全局变量和静态变量。BSS段在不占可执行文件空间，文件只记录BSS段的在内存中的开始和结束地址。  
   3.堆：动态分配的区域。  
   4.栈：局部变量（初始化以及未初始化的，但不包含静态变量）、局部只读变量（const）。



为什么需要分段呢，这是有理由的，总的来说，第一点是为了将程序的层次划分清晰，不过这点似乎不是最重要的原因才说得过去；第二点，① 由编译器负责挑选出数据具备的属性，从而根据属性将程序片段分类，比如，划分出了只读属性的代码段和可写属性的数据段；② 由内核分配段的属性（例如读、写、执行等）； ③ 这时候就可以将段描述符写进CPU的段寄存器中了，这就相当于操作系统定义段，然后要告知CPU，操作系统对段做了些什么，好让CPU做出相应的准备（例如只读段，CPU就不能去写，写操作就会产生异常）。

做过开发的同学都清楚，尽量把同一属性的数据放在一起，这样易于维护。这一点类似于MVC，在程序逻辑中把模型、视图、控制这三部分分开，这样更新各部分时，不会影响到其他模块。

将数据和代码分开的好处有三点。

第一，可以为它们赋予不同的属性。

例如数据本身是需要修改的，所以数据就需要有可写的属性，不让数据段可写，那程序根本就无法执行啦。程序中的代码是不能被更改的，这样就要求代码段具备只读的属性。真要是在运行过程中程序的下一条指令被修改了，谁知道会产生什么样的灾难。

第二，为了提高CPU内部缓存的命中率。

大伙儿知道，缓存起作用的原因是程序的局部性原理。在CPU内部也有缓存机制，将程序中的指令和数据分离，这有利于增强程序的局部性。CPU内部有针对数据和针对指令的两种缓存机制，因此，将数据和代码分开存储将使程序运行得更快。

第三，节省内存。

程序中存在一些只读的部分，比如代码，当一个程序的多个副本同时运行时（比如同时执行多个ls命令时），没必要在内存中同时存在多个相同的代码段，这将浪费有限的物理内存资源，只要把这一个代码段共享就可以了。

静态变量

静态变量只作用与本文件中，其他文件不能直接调用（非静态全局变量可以通过extern 申明然后使用），否则连接时找不到符号，但是可以通过函数传递地址来间接使用。静态变量更不能放到头文件做全局变量，否则每个引用的文件都会拷贝一份，注意结构体的静态成员变量只是声明，并没有定义和分配内存，所以每个引用的.cpp文件中使用前都必须在全局区域初始化（int Person::c = 0;） 所以每个cpp文件中该静态变量的地址也不一样

# **内存屏障（无锁循环队列）**

指令执行乱序包括cpu执行乱序（Memory barrier）和编译器优化导致代码乱序执行（\_\_volatile\_\_

），

例子（运行时的cpu乱序）：

一个线程写一个线程读的循环队列， 队列长度设置在65536来简化描述（省去求余操作）

struct {

char buf[65536];

unsigned short writer\_index;

unsigned short reader\_index;

}

用法：

0. writer\_index和reader\_index初始化为0

1.写者发现reader\_index-writer\_index != 1就可以从buf[writer\_index]写，写了之后writer\_index自增（写量当然也不能超过reader\_index-writer\_index - 1）。

2.读者发现writer\_index - reader\_index != 0 就可以从buf[reader\_index]读，读了之后reader\_index自增（读量当然也不能超过writer\_index - reader\_index）。

注意：

1.writer\_index 和 reader\_index都各自只有一个线程去写，所以不存在写冲突或者对读写顺序的依赖；

2.读者如果读到writer\_index更新前的旧值，只会少读一点，不会造成程序错误。 写者如果读到reader\_index更新前的旧值，只会少写一点，也不会造成程序错误。

3.综合1，2 我们这个循环队列是可以免锁的了。。然而这就够了吗？

不够。

剖析：

cpu完全可以执行一种乱序操作：在写者线程里，让writer\_index先自增， 再往buf里写（单线程看起来是交换这个顺序是完全没有问题的）。。此时读者如果读到writer\_index自增后的结果，认为可以读writer\_index自增那么多数据，而buf里实际还没有写入那些数据，就懵了。

同理，在读者线程里，cpu也可以让reader\_index先自增， 再从buf里读出来数据。。此时如果写者读到reader\_index自增后的结果，认为有reader\_index自增那么多空余空间可以写入，但是实际上那段空间的读者还未把他们读走，就懵了。

在多核处理器上，对另外一个核的缓存更新通知是可能有乱序的，这会造成和上面2点等价的错误，完全可能两个线程是在不同的核上执行的。

综上：修改reader\_index和writer\_index之前一定要加一个内存屏障来阻止这种cpu乱序执行（或通知）。修改流程如下：

1.写者发现reader\_index - writer\_index != 1就可以从buf[writer\_index]写，写了之后再放一个类似sfence指令一样的写屏障，然后再writer\_index自增。 这样就让buf的写和writer\_index的写不能错位了（也避免了用错误的顺序通知读者所在的核 关于writer\_index的移动和buf的写）。

2.读者发现writer\_index - reader\_index != 0 就可以从buf[reader\_index]读，读了之后再放一个类似mfence指令一样的指令屏障，然后再reader\_index自增。这样就让对buf的读和reader\_index的写不能错位了（也避免了过早的通知写者所在的核 关于reader\_index的移动 ）。

这样我们就成功实现了一个无锁但使用了内存屏障指令的 单写单读循环队列。

# **mach-o**

mach-o是ios和mac os特有的可执行文件格式，注意静态库不是，但是动态库是，但是fat静态库和可执行文件都会有fatheader字段。 fatheader包含不同架构的mach-o（即不同架构）的大小和偏移。

Mach-o由很多load command（加载命令）组成，最常见的是segment\_command（段命令，主要加载数据到内存，所以里面保存了数据在文件中的偏移和内存中的地址），

struct segment\_command\_64 { /\* for 64-bit architectures \*/

uint32\_t cmd; /\* 命令类型 \*/

uint32\_t cmdsize; /\* 命令大小 所以的load commad前面都是这两个属性，用于判断和解析不同的命令\*/

char segname[16]; /\* segment name \*/

uint64\_t vmaddr; /\*段的虚拟内存启始地址

\*/

uint64\_t vmsize; /\* 段的虚拟内存大小

\*/

uint64\_t fileoff; /\* 段在文件中的偏移量

\*/

uint64\_t filesize; /\* 段在文件中的大小

\*/

vm\_prot\_t maxprot; /\*段页面所需要的最高内存保护（4=r,2=w,1=x）

\*/

vm\_prot\_t initprot; /\* 段页面初始的内存保护

\*/

uint32\_t nsects; /\*段中包含section的数量

\*/

uint32\_t flags; /\* 其他杂项标志位

\*/

};

一般在\_\_TEXT和\_\_DATA段中有section header(注意，只是header,真正的section还在后面，header中有标记位置和大小)。这里注意下命名规范，大写代表segment,小写是section，例如 \_\_TEXT.\_\_text,指的是\_\_TEXT段的\_\_text节。

struct section { /\* for 32-bit architectures \*/

char sectname[16]; /\* 例子中是\_\_text ,就是主程序代码

\*/

char segname[16]; /\* 该section所属的 segment名，例子中是\_\_TEXT

\*/

uint32\_t addr; /\* 该section在内存的启始位置

\*/

uint32\_t size; /\* 该section的大小

\*/

uint32\_t offset; /\*  该section的文件偏移，根据例子中上下文，文件开始于0x4000位置，offset大小是0x7268，这样就得出绝对位置是0x4000+0x7268，和之前addr的值是一样。

\*/

uint32\_t align; /\* section alignment (power of 2) \*/

uint32\_t reloff; /\* 重定位入口的文件偏移

\*/

uint32\_t nreloc; /\* 需要重定位的入口数量

\*/

uint32\_t flags; /\* 包含section的type和attributes，具体看loader.h。)\*/

uint32\_t reserved1; /\* reserved (for offset or index) \*/

uint32\_t reserved2; /\* reserved (for count or sizeof) \*/

};

以下是\_\_TEXT段的section

\_\_text  主程序代码

\_\_stubs 和\_\_stub\_helper   用于动态链接库的stub

\_\_cstring    c语言字符串

\_\_const    const修饰的常量

\_\_objc\_methname    objc的方法名称

\_\_objc\_methtype    objc方法类型

\_\_objc\_classname    objc类方法

以下是\_\_DATA段的section

\_\_objc\_ivars   objc类的实例变量

\_\_objc\_classlist    objc类列表

\_\_objc\_protolist    objc协议列表

\_\_objc\_imageinfo    objc镜像信息

\_\_objc\_const    objc常量

\_\_objc\_selfrefs    objc自引用(self)

\_\_objc\_protorefs    objc协议引用

\_\_objc\_superrefs    objc超类引用

\_\_cfstring   使用Core Foundation字符串

\_\_bss   BSS

\_DWARF段不属于任何segment,用于调试信息，包含了调试信息的引用，当在本地调试时，可以直接引用.o文件的信息。但是发布版本时没有.o文件，所以一般发布版本需要保留dsym文件，dsym存储了调试信息

# 密码学

散列算法：MD5,SHA1

对称加密算法：DES,AES, 算法效率高

非对称算法：RSA，ECC椭圆曲线加密算法（新型算法）。算法效率低，不适合传输大数据，但是可以先通过RSA协商对称加密算法，然后再通过对称加密算法加密。

另外Base64编码，严格来说base64不能算是一种加密，只能说是编码转换。使用base64的初衷。是为了方便把含有不可见字符串的信息用可见字符串表示出来，以便其他处理；

关键词解释

秘钥（key）：用于加密解密的钥匙

初始化向量（iv）：用于结合秘钥加密第一个块，如下CBC

ECB(Electronic Code Book:电码本)：所有的明文用相同的密文加密，导致结果是如果明文相同那么密文也会相同，别人就可以看到这些密码特征

CBC(Cipher Block Chaining:密码分组连接)：将明文分成多个块，每次使用上一个块的信息加密下一个块的加密过程中，这样就会避免相同明文块输出相同密文块的缺点，而对于第一个块，我们引入IV向量来当做上一个块加密第一个块 。

缺点：

1. 不利于并行计算，
2. 会导致误差传递
3. 需要初始化向量IV

# 对称加密

# RSA原理

RSA是基于数论中大素数的乘积难分解理论上的非对称加密法。在此密码术中，使用公钥（public key）和私钥（private key）两个不同的密钥：公钥用于加密，它是向所有人公开的；私钥用于解密，只有密文的接收者持有。

举例：小红希望安全地发送一条消息给小明，消息明文为m，小明的公钥为K+，小明的私钥为K-。通信过程为，小红使用K+加密m，成为密文K+(m)，传送给小明，小明收到后使用K-解密这个密文得到原始消息明文，即m = K-(K+(m))。

具体的密钥生成算法如下。随机选择两个大素数p和q（比如每个都是1024 bit），计算n = pq, n’ = (p-1)(q-1). 选择一个e (e小于n) 与n’互质。计算d使得ed = 1 mod n’.则公钥为(n, e)，私钥为(n, d). 注意p和q都应该保密。

加密过程为c = m^e mod n  
解密过程为m = c^d mod n

**第一步，随机选择两个不相等的质数p和q。**

爱丽丝选择了61和53。（实际应用中，这两个质数越大，就越难破解。）

**第二步，计算p和q的乘积n。**

爱丽丝就把61和53相乘。

　　n = 61×53 = 3233

n的长度就是密钥长度。3233写成二进制是110010100001，一共有12位，所以这个密钥就是12位。实际应用中，RSA密钥一般是1024位，重要场合则为2048位。

**第三步，计算n的欧拉函数φ(n)。**

根据公式：

　　φ(n) = (p-1)(q-1)

爱丽丝算出φ(3233)等于60×52，即3120。

**第四步，随机选择一个整数e，条件是1< e < φ(n)，且e与φ(n) 互质。**

爱丽丝就在1到3120之间，随机选择了17。（实际应用中，常常选择65537。）

**第五步，计算e对于φ(n)的模反元素d。**

所谓["模反元素"](http://zh.wikipedia.org/wiki/%E6%A8%A1%E5%8F%8D%E5%85%83%E7%B4%A0)就是指有一个整数d，可以使得ed被φ(n)除的余数为1。

　　ed ≡ 1 (mod φ(n))

这个式子等价于

　　ed - 1 = kφ(n)

于是，找到模反元素d，实质上就是对下面这个二元一次方程求解。

　　ex + φ(n)y = 1

已知 e=17, φ(n)=3120，

　　17x + 3120y = 1

这个方程可以用["扩展欧几里得算法"](http://zh.wikipedia.org/wiki/%E6%89%A9%E5%B1%95%E6%AC%A7%E5%87%A0%E9%87%8C%E5%BE%97%E7%AE%97%E6%B3%95)求解，此处省略具体过程。总之，爱丽丝算出一组整数解为 (x,y)=(2753,-15)，即 d=2753。

至此所有计算完成。

**第六步，将n和e封装成公钥，n和d封装成私钥。**

在爱丽丝的例子中，n=3233，e=17，d=2753，所以公钥就是 (3233,17)，私钥就是（3233, 2753）。

实际应用中，公钥和私钥的数据都采用[ASN.1](http://zh.wikipedia.org/zh-cn/ASN.1)格式表达（[实例](http://hi.baidu.com/mathack/item/d0ad4cc1514a3663f7c95da2)）。

IOS 踩坑

UIImage不负责图像内存的保存，只是图像内存的引用，所以当cvpixelbuffer->ciimage->uiimage的转换时，如果对cvpixelbuffer的内存进行修改时，会直接影响到UIImage的显示。

UIImageview.image = image的形式赋值时，如果image的地址不变时，UIImageview不会更新，所以此时就算图像数据修改了，也不能反映在uiimageview上，需要先UIImageview.image = nil,然后UIImageview.image = image才能更新图像。