目录

[网络 2](#_Toc493515714)

[【网络基础】OSI TCP/IP体系结构 2](#_Toc493515715)

[以太网帧格式 4](#_Toc493515716)

[IP数据报格式 5](#_Toc493515717)

[TCP结构 11](#_Toc493515718)

[粘包与分包 14](#_Toc493515719)

[全双工半双工 16](#_Toc493515720)

[socket 17](#_Toc493515721)

[多播/组播 22](#_Toc493515722)

[RTMP 26](#_Toc493515723)

[RTSP 27](#_Toc493515724)

[NAT 32](#_Toc493515725)

[STUN 32](#_Toc493515726)

[c++语言 33](#_Toc493515727)

[关键字 33](#_Toc493515728)

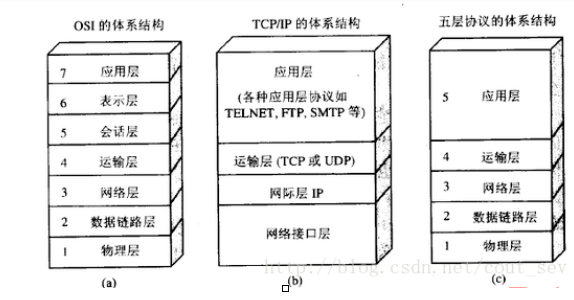
[智能指针 33](#_Toc493515729)

[知识点 33](#_Toc493515730)

[webrtc 33](#_Toc493515731)

# 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

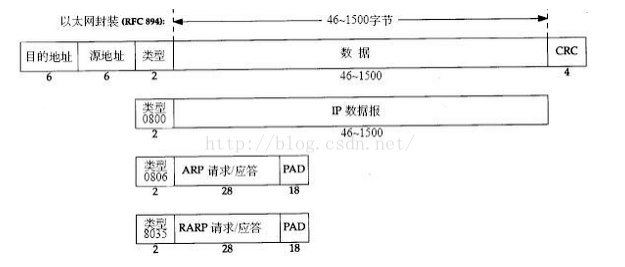
能，可以在不同的链路层接口之间转发数据包。因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP 协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。

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。

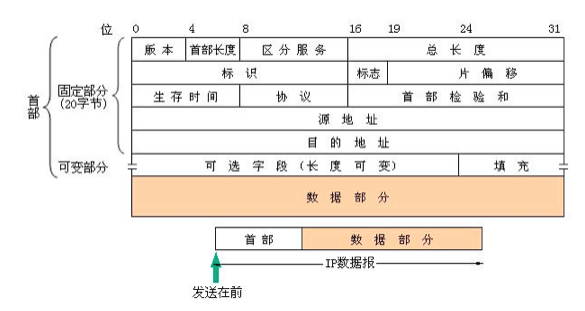
⑤应用层：为用户提供各种所需的服务。比如：FTP，DNS，Telnet，SMTP等。

## 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。

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。

数据（Date）：应用程序的数据。

TCP

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）；

## 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）。

…………

}

1. closesocket（一般不会立即关闭而经历TIME\_WAIT的过程）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));

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 \*)

## 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

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

## 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，

# 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