RGB、YUV像素数据处理

YUV 420 大小： w\*h\*3/2 Y：w\*h U：w\*h/4 V：w\*h/4

YUV 444 大小： w\*h\*3

YUV420P 大小：

1. 分离YUV420数据

for(int i=0;i<num;i++){

fread(pic,1,w\*h\*3/2,fp);

//Y

fwrite(pic,1,w\*h,fp1);

//U

fwrite(pic+w\*h,1,w\*h/4,fp2);

//V

fwrite(pic+w\*h\*5/4,1,w\*h/4,fp3);

}

1. 将YUV420P像素数据去掉颜色（变成灰度图）

如果想把 YUV 格式像素数据变成灰度图像，只需要将U、V分量设置成 128 即可。这是因为U、V是图像中的经过偏置处理的色度分量。色度分量在偏置处理前的取值范围是-128至127，这时候的无色对应的是“0”值。经过偏置后色度分量取值变成了0至255，因而此时的无色对应的就是128了。

1. 将YUV420P像素数据的亮度减半

Y 值变为之前的一半

1. 将YUV420P像素数据的周围加上边框

将边框范围内的数据，亮度设置为 255。

1. 计算两个YUV420P像素数据的PSNR

PSNR是最基本的视频质量评价方法。

mse\_sum+=pow((pic1[j]-pic2[j]),2);

mse = mse\_sum /(w\*h);

psnr = 10\*log10(255.0\*255.0/mse);

对于 8bit 量化的像素数据来说，PSNR 的计算公式：

https://img-blog.csdn.net/20160117233527240

上述公式中mse的计算公式如下所示。

https://img-blog.csdn.net/20160117233543104

其中M，N分别为图像的宽高，xij和yij分别为两张图像的每一个像素值。PSNR通常用于质量评价，就是计算受损图像与原始图像之间的差别，以此来评价受损图像的质量。

PSNR取值通常情况下都在**20-50**的范围内，取值越高，代表两张图像越接近，反映出受损图像质量越好。

1. 分离RGB24 的R、G、B等元素

for(int j=0;j<w\*h\*3;j=j+3){

//R

fwrite(pic+j,1,1,fp1);

//G

fwrite(pic+j+1,1,1,fp2);

//B

fwrite(pic+j+2,1,1,fp3);

}

RGB24格式的每个像素的三个分量是连续存储的。一帧宽高分别为w、h的RGB24图像一共占用w\*h\*3 Byte的存储空间。RGB24格式规定首先存储第一个像素的R、G、B，然后存储第二个像素的R、G、B…以此类推。类似于YUV420P的存储方式称为Planar方式，而类似于RGB24的存储方式称为Packed方式。

1. 将RGB24格式像素数据封装为BMP图像

1)将RGB数据前面加上文件头。

2)将RGB数据中每个像素的“B”和“R”的位置互换。

BMP文件是由BITMAPFILEHEADER、BITMAPINFOHEADER、RGB像素数据共3个部分构成，它的结构如下图所示。



BMP采用的是小端（Little Endian）存储方式。这种存储方式中“RGB24”格式的像素的分量存储的先后顺序为B、G、R。由于RGB24格式存储的顺序是R、G、B，所以需要将“R”和“B”顺序作一个调换再进行存储。

BMP文件头

typedef struct tagBITMAPFILEHEADER

{

unsigned short int bfType; //位图文件的类型，必须为BM

unsigned long bfSize; //文件大小，以字节为单位

unsigned short int bfReserverd1; //位图文件保留字，必须为0

unsigned short int bfReserverd2; //位图文件保留字，必须为0

unsigned long bfbfOffBits; //位图文件头到数据的偏移量，以字节为单位

}BITMAPFILEHEADER;

typedef struct tagBITMAPINFOHEADER

{

long biSize; //该结构大小，字节为单位

long biWidth; //图形宽度以象素为单位

long biHeight; //图形高度以象素为单位

short int biPlanes; //目标设备的级别，必须为1

short int biBitcount; //颜色深度，每个象素所需要的位数

short int biCompression; //位图的压缩类型

long biSizeImage; //位图的大小，以字节为单位

long biXPelsPermeter; //位图水平分辨率，每米像素数

long biYPelsPermeter; //位图垂直分辨率，每米像素数

long biClrUsed; //位图实际使用的颜色表中的颜色数

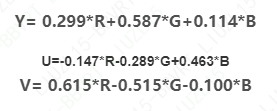
long biClrImportant; //位图显示过程中重要的颜色数

}BITMAPINFOHEADER;

BMP采用的是小端（Little Endian）存储方式。这种存储方式中“RGB24”格式的像素的分量存储的先后顺序为B、G、R。由于RGB24格式存储的顺序是R、G、B，所以需要将“R”和“B”顺序作一个调换再进行存储。

1. 将RGB24格式像素数据转换为YUV420P格式像素数据

RGB到YUV的转换公式：



1、RGB24存储方式是Packed，YUV420P存储方式是Packed。

2、U，V在水平和垂直方向的取样数是Y的一半

for (int j = 0; j<h;j++){

ptrRGB = RgbBuf + w\*j\*3 ;

for (int i = 0;i<w;i++){

r = \*(ptrRGB++);

g = \*(ptrRGB++);

b = \*(ptrRGB++);

y = (unsigned char)( ( 66 \* r + 129 \* g + 25 \* b + 128) >> 8) + 16 ;

u = (unsigned char)( ( -38 \* r - 74 \* g + 112 \* b + 128) >> 8) + 128 ;

v = (unsigned char)( ( 112 \* r - 94 \* g - 18 \* b + 128) >> 8) + 128 ;

\*(ptrY++) = clip\_value(y,0,255);

if (j%2==0&&i%2 ==0){

\*(ptrU++) =clip\_value(u,0,255);

}

else{

if (i%2==0){

\*(ptrV++) =clip\_value(v,0,255);

}

}

}

}

PCM音频数据处理

16LE。“16”代表采样位数是16bit。由于1Byte=8bit，所以一个声道的一个采样值占用2Byte。“LE”代表Little Endian，代表2 Byte采样值的存储方式为高位存在高地址中。

1. 分离PCM16LE双声道音频采样数据的左声道和右声道

while(!feof(fp)){

fread(sample,1,4,fp);

//L

fwrite(sample,1,2,fp1);

//R

fwrite(sample+2,1,2,fp2);

}

CM16LE双声道数据中左声道和右声道的采样值是间隔存储的。每个采样值占用2Byte空间。

1. 将PCM16LE双声道音频采样数据中左声道的音量降一半

unsigned char \*sample=(unsigned char \*)malloc(4);

while(!feof(fp)){

short \*samplenum=NULL;

fread(sample,1,4,fp);

samplenum=(short \*)sample; //读取了4个字节，但是 short 是2个字节，强制类型转换了

\*samplenum=\*samplenum/2;

//L

fwrite(sample,1,2,fp1);

//R

fwrite(sample+2,1,2,fp1);

cnt++;

}

读出左声道的2 Byte的取样值之后，将其当成了C语言中的一个short类型的变量。将该数值除以2之后写回到了PCM文件中。

1. 将PCM16LE双声道音频采样数据的声音速度提高一倍

unsigned char \*sample=(unsigned char \*)malloc(4);

while(!feof(fp)){

fread(sample,1,4,fp);

if(cnt%2!=0){

//L

fwrite(sample,1,2,fp1);

//R

fwrite(sample+2,1,2,fp1);

}

cnt++;

}

只采样了每个声道奇数点的样值。处理完成后，原本22秒左右的音频变成了11秒左右。音频的播放速度提高了2倍，音频的音调也变高了很多。

1. 将PCM16LE双声道音频采样数据转换为PCM8音频采样数据

unsigned char \*sample=(unsigned char \*)malloc(4);

while(!feof(fp)){

short \*samplenum16=NULL;

char samplenum8=0;

unsigned char samplenum8\_u=0;

fread(sample,1,4,fp);

//(-32768-32767)

samplenum16=(short \*)sample;

samplenum8=(\*samplenum16)>>8;

//(0-255)

samplenum8\_u=samplenum8+128;

//L

fwrite(&samplenum8\_u,1,1,fp1);

samplenum16=(short \*)(sample+2);

samplenum8=(\*samplenum16)>>8;

samplenum8\_u=samplenum8+128;

//R

fwrite(&samplenum8\_u,1,1,fp1);

cnt++;

}

PCM16LE格式的采样数据的取值范围是-32768到32767，而PCM8格式的采样数据的取值范围是0到255。所以PCM16LE转换到PCM8需要经过两个步骤：第一步是将-32768到32767的16bit有符号数值转换为-128到127的8bit有符号数值，第二步是将-128到127的8bit有符号数值转换为0到255的8bit无符号数值。

1. 将从PCM16LE单声道音频采样数据中截取一部分数据

unsigned char \*sample=(unsigned char \*)malloc(2);

int cnt=0;

while(!feof(fp)){

fread(sample,1,2,fp);

if(cnt>start\_num&&cnt<=(start\_num+dur\_num)){

fwrite(sample,1,2,fp1);

}

cnt++;

}

1. 将PCM16LE双声道音频采样数据转换为WAVE格式音频数据

WAVE格式音频（扩展名为“.wav”）是Windows系统中最常见的一种音频。该格式的实质就是在PCM文件的前面加了一个文件头。

typedef struct WAVE\_HEADER{

char fccID[4];

unsigned long dwSize;

char fccType[4];

}WAVE\_HEADER;

typedef struct WAVE\_FMT{

char fccID[4];

unsigned long dwSize;

unsigned short wFormatTag;

unsigned short wChannels;

unsigned long dwSamplesPerSec;

unsigned long dwAvgBytesPerSec;

unsigned short wBlockAlign;

unsigned short uiBitsPerSample;

}WAVE\_FMT;

typedef struct WAVE\_DATA{

char fccID[4];

unsigned long dwSize;

}WAVE\_DATA;

通过在PCM文件前面加一个WAVE文件头从而封装为WAVE格式音频。

//WAVE\_HEADER

memcpy(pcmHEADER.fccID,"RIFF",strlen("RIFF"));

memcpy(pcmHEADER.fccType,"WAVE",strlen("WAVE"));

fseek(fpout,sizeof(WAVE\_HEADER),1);

//WAVE\_FMT

pcmFMT.dwSamplesPerSec=sample\_rate;

pcmFMT.dwAvgBytesPerSec=pcmFMT.dwSamplesPerSec\*sizeof(m\_pcmData);

pcmFMT.uiBitsPerSample=bits;

memcpy(pcmFMT.fccID,"fmt ",strlen("fmt "));

pcmFMT.dwSize=16;

pcmFMT.wBlockAlign=2;

pcmFMT.wChannels=channels;

pcmFMT.wFormatTag=1;

fwrite(&pcmFMT,sizeof(WAVE\_FMT),1,fpout);

//WAVE\_DATA;

memcpy(pcmDATA.fccID,"data",strlen("data"));

pcmDATA.dwSize=0;

fseek(fpout,sizeof(WAVE\_DATA),SEEK\_CUR);

fread(&m\_pcmData,sizeof(unsigned short),1,fp);

while(!feof(fp)){

pcmDATA.dwSize+=2;

fwrite(&m\_pcmData,sizeof(unsigned short),1,fpout);

fread(&m\_pcmData,sizeof(unsigned short),1,fp);

}

pcmHEADER.dwSize=44+pcmDATA.dwSize;

rewind(fpout);

fwrite(&pcmHEADER,sizeof(WAVE\_HEADER),1,fpout);

fseek(fpout,sizeof(WAVE\_FMT),SEEK\_CUR);

fwrite(&pcmDATA,sizeof(WAVE\_DATA),1,fpout);

WAVE文件是一种RIFF格式的文件。其基本块名称是“WAVE”，其中包含了两个子块“fmt”和“data”。从编程的角度简单说来就是由WAVE\_HEADER、WAVE\_FMT、WAVE\_DATA、采样数据共4个部分组成。它的结构如下所示：



有一点需要注意：WAVE\_HEADER和WAVE\_DATA中包含了一个文件长度信息的dwSize字段，该字段的值必须在写入完音频采样数据之后才能获得。因此这两个结构体最后才写入WAVE文件中。

H.264视频码流解析

H.264原始码流（又称为“裸流”）是由一个一个的NALU组成的。他们的结构如下图所示：

https://img-blog.csdn.net/20160118001549018

其中每个NALU之间通过startcode（起始码）进行分隔，起始码分成两种：0x000001（3Byte）或者0x00000001（4Byte）。如果NALU对应的Slice为一帧的开始就用0x00000001，否则就用0x000001。

H.264码流解析的步骤就是**首先从码流中搜索0x000001和0x00000001，分离出NALU**；然后再分析NALU的各个字段。

typedef enum {

NALU\_TYPE\_SLICE = 1,

NALU\_TYPE\_DPA = 2,

NALU\_TYPE\_DPB = 3,

NALU\_TYPE\_DPC = 4,

NALU\_TYPE\_IDR = 5,

NALU\_TYPE\_SEI = 6,

NALU\_TYPE\_SPS = 7,

NALU\_TYPE\_PPS = 8,

NALU\_TYPE\_AUD = 9,

NALU\_TYPE\_EOSEQ = 10,

NALU\_TYPE\_EOSTREAM = 11,

NALU\_TYPE\_FILL = 12,

} NaluType;

typedef enum {

NALU\_PRIORITY\_DISPOSABLE = 0,

NALU\_PRIRITY\_LOW = 1,

NALU\_PRIORITY\_HIGH = 2,

NALU\_PRIORITY\_HIGHEST = 3

} NaluPriority;

typedef struct

{

int startcodeprefix\_len;

//! 4 for parameter sets and first slice in picture, 3 for everything else (suggested)

unsigned len;

//! Length of the NAL unit (Excluding the start code, which does not belong to the NALU)

unsigned max\_size; //! Nal Unit Buffer size

int forbidden\_bit; //! should be always FALSE

int nal\_reference\_idc; //! NALU\_PRIORITY\_xxxx

int nal\_unit\_type; //! NALU\_TYPE\_xxxx

char \*buf; //! contains the first byte followed by the EBSP

} NALU\_t;

//寻找 NALU 的起始码

static int FindStartCode2 (unsigned char \*Buf){

if(Buf[0]!=0 || Buf[1]!=0 || Buf[2] !=1) return 0; //0x000001?

else return 1;

}

static int FindStartCode3 (unsigned char \*Buf){

if(Buf[0]!=0 || Buf[1]!=0 || Buf[2] !=0 || Buf[3] !=1) return 0;//0x00000001?

else return 1;

}

int GetAnnexbNALU (NALU\_t \*nalu){

int pos = 0;

int StartCodeFound, rewind;

unsigned char \*Buf;

if ((Buf = (unsigned char\*)calloc (nalu->max\_size , sizeof(char))) == NULL)

printf ("GetAnnexbNALU: Could not allocate Buf memory\n");

nalu->startcodeprefix\_len=3;

if (3 != fread (Buf, 1, 3, h264bitstream)){

free(Buf);

return 0;

}

info2 = FindStartCode2 (Buf);

if(info2 != 1) {

if(1 != fread(Buf+3, 1, 1, h264bitstream)){

free(Buf);

return 0;

}

info3 = FindStartCode3 (Buf);

if (info3 != 1){

free(Buf);

return -1;

}

else {

pos = 4;

nalu->startcodeprefix\_len = 4;

}

}

else{

nalu->startcodeprefix\_len = 3;

pos = 3;

}

StartCodeFound = 0;

info2 = 0;

info3 = 0;

while (!StartCodeFound){

if (feof (h264bitstream)){

nalu->len = (pos-1)-nalu->startcodeprefix\_len;

memcpy (nalu->buf, &Buf[nalu->startcodeprefix\_len], nalu->len);

nalu->forbidden\_bit = nalu->buf[0] & 0x80; //1 bit

nalu->nal\_reference\_idc = nalu->buf[0] & 0x60; // 2 bit

nalu->nal\_unit\_type = (nalu->buf[0]) & 0x1f;// 5 bit

free(Buf);

return pos-1;

}

Buf[pos++] = fgetc (h264bitstream);

info3 = FindStartCode3(&Buf[pos-4]);

if(info3 != 1)

info2 = FindStartCode2(&Buf[pos-3]);

StartCodeFound = (info2 == 1 || info3 == 1);

}

// Here, we have found another start code (and read length of startcode bytes more than we should

// have. Hence, go back in the file

rewind = (info3 == 1)? -4 : -3;

if (0 != fseek (h264bitstream, rewind, SEEK\_CUR)){

free(Buf);

printf("GetAnnexbNALU: Cannot fseek in the bit stream file");

}

// Here the Start code, the complete NALU, and the next start code is in the Buf.

// The size of Buf is pos, pos+rewind are the number of bytes excluding the next

// start code, and (pos+rewind)-startcodeprefix\_len is the size of the NALU excluding the start code

nalu->len = (pos+rewind)-nalu->startcodeprefix\_len;

memcpy (nalu->buf, &Buf[nalu->startcodeprefix\_len], nalu->len);//

nalu->forbidden\_bit = nalu->buf[0] & 0x80; //1 bit

nalu->nal\_reference\_idc = nalu->buf[0] & 0x60; // 2 bit

nalu->nal\_unit\_type = (nalu->buf[0]) & 0x1f;// 5 bit

free(Buf);

return (pos+rewind);

}

int simplest\_h264\_parser(char \*url){

NALU\_t \*n;

int buffersize=100000;

//FILE \*myout=fopen("output\_log.txt","wb+");

FILE \*myout=stdout;

h264bitstream=fopen(url, "rb+");

if (h264bitstream==NULL){

printf("Open file error\n");

return 0;

}

n = (NALU\_t\*)calloc (1, sizeof (NALU\_t));

if (n == NULL){

printf("Alloc NALU Error\n");

return 0;

}

n->max\_size=buffersize;

n->buf = (char\*)calloc (buffersize, sizeof (char));

if (n->buf == NULL){

free (n);

printf ("AllocNALU: n->buf");

return 0;

}

int data\_offset=0;

int nal\_num=0;

printf("-----+-------- NALU Table ------+---------+\n");

printf(" NUM | POS | IDC | TYPE | LEN |\n");

printf("-----+---------+--------+-------+---------+\n");

while(!feof(h264bitstream))

{

int data\_lenth;

data\_lenth=GetAnnexbNALU(n);

char type\_str[20]={0};

switch(n->nal\_unit\_type){

case NALU\_TYPE\_SLICE:sprintf(type\_str,"SLICE");break;

case NALU\_TYPE\_DPA:sprintf(type\_str,"DPA");break;

case NALU\_TYPE\_DPB:sprintf(type\_str,"DPB");break;

case NALU\_TYPE\_DPC:sprintf(type\_str,"DPC");break;

case NALU\_TYPE\_IDR:sprintf(type\_str,"IDR");break;

case NALU\_TYPE\_SEI:sprintf(type\_str,"SEI");break;

case NALU\_TYPE\_SPS:sprintf(type\_str,"SPS");break;

case NALU\_TYPE\_PPS:sprintf(type\_str,"PPS");break;

case NALU\_TYPE\_AUD:sprintf(type\_str,"AUD");break;

case NALU\_TYPE\_EOSEQ:sprintf(type\_str,"EOSEQ");break;

case NALU\_TYPE\_EOSTREAM:sprintf(type\_str,"EOSTREAM");break;

case NALU\_TYPE\_FILL:sprintf(type\_str,"FILL");break;

}

char idc\_str[20]={0};

switch(n->nal\_reference\_idc>>5){

case NALU\_PRIORITY\_DISPOSABLE:sprintf(idc\_str,"DISPOS");break;

case NALU\_PRIRITY\_LOW:sprintf(idc\_str,"LOW");break;

case NALU\_PRIORITY\_HIGH:sprintf(idc\_str,"HIGH");break;

case NALU\_PRIORITY\_HIGHEST:sprintf(idc\_str,"HIGHEST");break;

}

fprintf(myout,"%5d| %8d| %7s| %6s| %8d|\n",nal\_num,data\_offset,idc\_str,type\_str,n->len);

data\_offset=data\_offset+data\_lenth;

nal\_num++;

}

//Free

if (n){

if (n->buf){

free(n->buf);

n->buf=NULL;

}

free (n);

}

return 0;

}

AAC音频码流解析

AAC原始码流（又称为“裸流”）是由一个一个的ADTS frame组成的。他们的结构如下图所示。

https://img-blog.csdn.net/20160118101611729

其中每个ADTS frame之间通过syncword（同步字）进行分隔。同步字为0xFFF（二进制“111111111111”）。AAC码流解析的步骤就是首先从码流中搜索0x0FFF，分离出ADTS frame；然后再分析ADTS frame的首部各个字段。

首部各个字段：（0~6 共7字节）

* 1. : AAC码流标识

2：

最高两位表示协议，有 Main、LC、SSR等

后6位的4倍表示采样率，如：44100HZ

3-5：size

6：暂不知

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int getADTSframe(unsigned char\* buffer, int buf\_size, unsigned char\* data ,int\* data\_size){

int size = 0;

if(!buffer || !data || !data\_size ){

return -1;

}

while(1){

if(buf\_size < 7 ){

return -1;

}

//Sync words

if((buffer[0] == 0xff) && ((buffer[1] & 0xf0) == 0xf0) ){

size |= ((buffer[3] & 0x03) <<11); //high 2 bit

size |= buffer[4]<<3; //middle 8 bit

size |= ((buffer[5] & 0xe0)>>5); //low 3bit

break;

}

--buf\_size;

++buffer;

}

if(buf\_size < size){

return 1;

}

memcpy(data, buffer, size);

\*data\_size = size;

return 0;

}

int simplest\_aac\_parser(char \*url)

{

int data\_size = 0;

int size = 0;

int cnt=0;

int offset=0;

//FILE \*myout=fopen("output\_log.txt","wb+");

FILE \*myout=stdout;

unsigned char \*aacframe=(unsigned char \*)malloc(1024\*5);

unsigned char \*aacbuffer=(unsigned char \*)malloc(1024\*1024);

FILE \*ifile = fopen(url, "rb");

if(!ifile){

printf("Open file error");

return -1;

}

printf("-----+- ADTS Frame Table -+------+\n");

printf(" NUM | Profile | Frequency| Size |\n");

printf("-----+---------+----------+------+\n");

while(!feof(ifile)){

data\_size = fread(aacbuffer+offset, 1, 1024\*1024-offset, ifile);

unsigned char\* input\_data = aacbuffer;

while(1)

{

int ret=getADTSframe(input\_data, data\_size, aacframe, &size);

if(ret==-1){

break;

}else if(ret==1){

memcpy(aacbuffer,input\_data,data\_size);

offset=data\_size;

break;

}

char profile\_str[10]={0};

char frequence\_str[10]={0};

unsigned char profile=aacframe[2]&0xC0;

profile=profile>>6;

switch(profile){

case 0: sprintf(profile\_str,"Main");break;

case 1: sprintf(profile\_str,"LC");break;

case 2: sprintf(profile\_str,"SSR");break;

default:sprintf(profile\_str,"unknown");break;

}

unsigned char sampling\_frequency\_index=aacframe[2]&0x3C;

sampling\_frequency\_index=sampling\_frequency\_index>>2;

switch(sampling\_frequency\_index){

case 0: sprintf(frequence\_str,"96000Hz");break;

case 1: sprintf(frequence\_str,"88200Hz");break;

case 2: sprintf(frequence\_str,"64000Hz");break;

case 3: sprintf(frequence\_str,"48000Hz");break;

case 4: sprintf(frequence\_str,"44100Hz");break;

case 5: sprintf(frequence\_str,"32000Hz");break;

case 6: sprintf(frequence\_str,"24000Hz");break;

case 7: sprintf(frequence\_str,"22050Hz");break;

case 8: sprintf(frequence\_str,"16000Hz");break;

case 9: sprintf(frequence\_str,"12000Hz");break;

case 10: sprintf(frequence\_str,"11025Hz");break;

case 11: sprintf(frequence\_str,"8000Hz");break;

default:sprintf(frequence\_str,"unknown");break;

}

fprintf(myout,"%5d| %8s| %8s| %5d|\n",cnt,profile\_str ,frequence\_str,size);

data\_size -= size;

input\_data += size;

cnt++;

}

}

fclose(ifile);

free(aacbuffer);

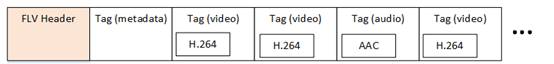
free(aacframe);

return 0;

}

FLV封装格式解析

FLV封装格式是由一个FLV Header文件头和一个一个的Tag组成的。Tag中包含了音频数据以及视频数据。FLV的结构如下图所示。



FlvHeader:

struct{

byte Signature[3];

byte Version;

byte Flags;

uint DataOffsets;

}FLV\_HEADER;

TagHeader:

typedef struct{

byte TagType;

byte DataSize[3];

byte Timestamp[3];

uint Reserved;

}

TagType:

TAG\_TYPE\_AUDIO

TAG\_TYPE\_VIDEO

TAG\_TYPE\_SCRIPT

DataSize： DataSize[0]\*65536+ DataSize[1]\*256+ DataSize[2]

Timestamp[3]：Timestamp[0]\*65536+ Timestamp[1]\*256+ Timestamp[2]

数据首字节：

TagType：

TAG\_TYPE\_ADUIO:

0：声道

1：位数

2-3：采样率

4-7：协议格式

TAG\_TYPE\_VIDEO:

0-3位：

0： UNKNOWN

1： JPEG(currently unused)

2： Sorenson H.263

3： Screen video

4： On2 VP6

5： On2 VP6 with alpha channel

6： Screen video version 2

7： AVC

4-7位：

1：key frame

2：inter frame

3：disposable inter frame

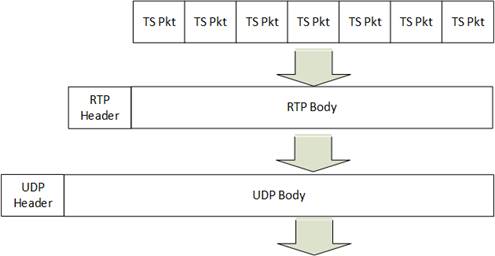
4：generated keyframe

5：video info/command frame

other：UNKNOWN

UDP-RTP协议解析：

首先每7个MPEG-TS Packet打包为一个RTP，然后每个RTP再打包为一个UDP。其中打包RTP的方法就是在MPEG-TS数据前面加上RTP Header，而打包RTP的方法就是在RTP数据前面加上UDP Header。



typedef struct RTP\_FIXED\_HEADER{

/\* byte 0 \*/

unsigned char csrc\_len:4; /\* expect 0 \*/

unsigned char extension:1; /\* expect 1 \*/

unsigned char padding:1; /\* expect 0 \*/

unsigned char version:2; /\* expect 2 \*/

/\* byte 1 \*/

unsigned char payload:7;

unsigned char marker:1; /\* expect 1 \*/

/\* bytes 2, 3 \*/

unsigned short seq\_no;

/\* bytes 4-7 \*/

unsigned long timestamp;

/\* bytes 8-11 \*/

unsigned long ssrc; /\* stream number is used here. \*/

} RTP\_FIXED\_HEADER;

typedef struct MPEGTS\_FIXED\_HEADER {

unsigned sync\_byte: 8;

unsigned transport\_error\_indicator: 1;

unsigned payload\_unit\_start\_indicator: 1;

unsigned transport\_priority: 1;

unsigned PID: 13;

unsigned scrambling\_control: 2;

unsigned adaptation\_field\_exist: 2;

unsigned continuity\_counter: 4;

} MPEGTS\_FIXED\_HEADER;

RTP:

payload:

0-18: "Audio"

31: "H.261"

32: "MPV"

33: "MP2T"

34: "H.263"

96: "H.264"

default:"other"

MPEGTS:

起始字节为 0x47

长度为 188

RTMP:

最简单的Flash流媒体系统由两部分组成：

客户端：

Flash Player

服务器端：

Flash Media Server(FMS)

Red5

libRTMP(RTMP流媒体客户端)

rtmpdump 是一个可以通过RTMP协议下载流媒体的工具.  
rtmpdump 通过给定的URL建立连接到指定的RTMP服务器. 这URL应该如下：  
  rtmp[t][e]://hostname[:port][/app[/playpath]]  
支持一般以及加密的RTMP会话

选项

网络参数

这些选项定义了如何连接到服务器

--rtmp -r url

连接到服务器的url

--host -n hostname

覆盖 RTMP URL的主机地址.

--port -c port

覆盖 RTMP URL的端口号.

--protocol -l number

覆盖 RTMP URL的协议.

0 = rtmp

1 = rtmpt

2 = rtmpe

3 = rtmpte

4 = rtmps

5 = rtmpts

--socks -S host:port

使用指定 SOCKS4 代理.

连接参数

这些选项定义了RTMP请求连接的数据包内容，如果不提供正确的值，服务器将拒绝连接尝试

--app -a app

连接到RTMP的应用名，覆盖RTMP URL中的app。有时rtmpdump URL 无法正确自动解析app名称。这时必须使用该选项。

--tcUrl -t url

目标流的URL。默认格式 rtmp[e]://host[:port]/app/playpath.

--pageUrl -p url

媒体嵌入网页的URL，默认情况下没有被发送的价值

--swfUrl -s url

该媒体swf播放器的URL. 默认情况下没有被发送的价值.

--flashVer -f version

swf播放器使用的Flash版本. 默认是"LNX 10,0,32,18".

--auth -u string

增加到connect的验证字符串，使用此选项将增加一个bool值TRUE，然后是指定的字符串，此选项仅用于一些特定的服务器，已经过时了，更一般情况，--conn选项应该被代替使用

--conn -C type:data

任意AMF数据追加到connect，类型说明，

B 布尔型

N 数字

S 字符串

O 对象

Z 空

对于布尔型必须是0或1作为FALSE或TRUE，

对于对象数据必须以0或1分别作为结束和开始的标制，子对象中数据项前加前缀类型N，并指定值名称，例如：

-C B:1 -C S:authMe -C O:1 -C NN:code:1.23 -C NS:flag:ok -C O:0

会话选项

这些选项在连接成功后生效

--playpath -y path

覆盖RTMP URL解析的playpath，rtmpdump有时不能正确解析，通过该选项明确

--playlist -Y

在play命令之前发生set\_playlist命令。否则播放列表将会值包含playpath

--live -v

指定媒体是实时流。在实时流中没有恢复和搜索。

--subscribe -d stream

订阅的实时流名称。默认 playpath.

--realtime -R

实时下载时试图使用暂停和恢复命令("the BUFX hack")，服务器可以调到后面再使用恢复命令，恢复和搜索在实时流中还是可以实现的

--resume -e

恢复一个不完整的rtmp流下载

--skip -k num

恢复时寻找最后一个关键帧跳过的关键帧数。如果经常尝试恢复失败，这可能有用，默认0。

--start -A num

开始到流的秒数，实时流无效

--stop -B num

该流停止秒数

--buffer -b num

设置缓冲时间，单位毫秒. The default is 36000000.

--timeout -m num

num秒后没有收到任何数据会话超时，默认值120。

安全参数

这些选项处理额外的身份验证，来自服务器的请求

--token -T key

输入安全令牌响应，如果服务器需要使用安全令牌验证

--jtv -j JSON

JSON令牌用于传统Justin.tv服务器 ，调用NetStream.Authenticate.UsherToken

--swfhash -w hexstring

swf文件的SHA256 hash 数值. 如果服务器使用swf验证，该选项会被用到, 看下面 --swfVfy 选项. 该哈希值是32字节，并且必须以十六进制表示. 始终与 --swfsize 选项一起使用.

--swfsize -x num

解压后SWF文件大小. 服务器swf验证时可能需要此选项，看下面--swfVfy 选项.始终与 --swfhash 选项一起使用.

--swfVfy -W url

swf播放器的URL，此选项将替换所以三个--swfUrl, --swfhash, and --swfsize选项。使用此选项时，swf播放器将从指定URL检索，并自动计算哈希和大小。此外信息缓存在一个swfinfo文件在用户主目录，所以它在每次rtmpdump运行时，并不需要检索和重新计算。swfinfo记录URL，生成时间，修改SWF文件时间，它的大小，它的哈希，默认情况下，缓冲信息用于30天，然后重新检测。

--swfAge -X days

指定使用缓存的swf信息天数，然后重新检查，使用0为经常检查，如果检查显示swf具有相同的修改时间戳，它不会被再次检索

杂项

--flv -o output

指定输出文件名，如果这个名字是-或被省略，流写入到stdout

--hashes -#

显示流进度，一个哈希标记没1%进度，而不是字节计数器

--quiet -q

制止所以命令的输出

--verbose -V

详细输出.

--debug -z

debug输出等级，最详细可输出所有分组数据

--help -h

打印命令选项摘要

退出状态

0

程序成功运行

1

不可知错误

2

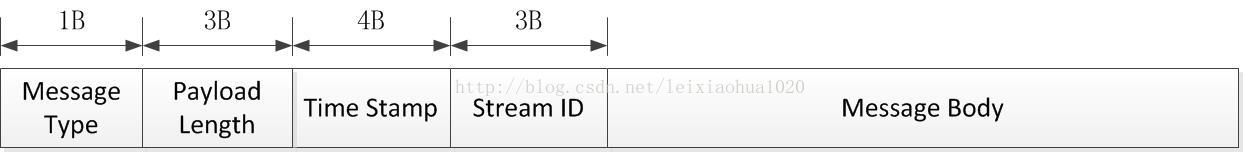
不完全转移，可能会得到进一步恢复

RTMP规范：

RTMP协议是一个互联网TCP/IP五层体系结构中应用层的协议。RTMP协议中基本的数据单元称为消息（Message）。当RTMP协议在互联网中传输数据的时候，消息会被拆分成更小的单元，称为消息块（Chunk）。

消息：

消息是RTMP协议中基本的数据单元。不同种类的消息包含不同的Message Type ID，代表不同的功能。RTMP协议中一共规定了十多种消息类型，分别发挥着不同的作用。例如，Message Type ID在1-7的消息用于协议控制，这些消息一般是RTMP协议自身管理要使用的消息，用户一般情况下无需操作其中的数据。Message Type ID为8，9的消息分别用于传输音频和视频数据。Message Type ID为15-20的消息用于发送AMF编码的命令，负责用户与服务器之间的交互，比如播放，暂停等等。消息首部（Message Header）有四部分组成：标志消息类型的Message Type ID，标志消息长度的Payload Length，标识时间戳的Timestamp，标识消息所属媒体流的Stream ID。消息的报文结构如图所示：



Message Type：

消息的类型Id，1个字节

Payload Length：

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

Time Stamp：

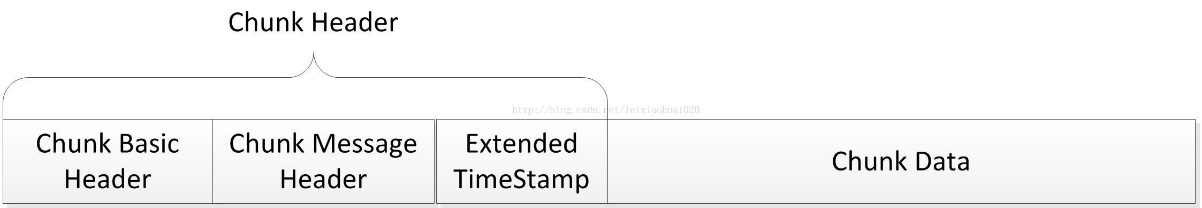
消息的时间戳（但不一定是当前时间），4个字节

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。

消息块

在网络上传输数据时，消息需要被拆分成较小的数据块，才适合在相应的网络环境上传输。RTMP协议中规定，消息在网络上传输时被拆分成消息块（Chunk）。消息块首部（Chunk Header）有三部分组成：用于标识本块的Chunk Basic Header，用于标识本块负载所属消息的Chunk Message Header，以及当时间戳溢出时才出现的Extended Timestamp。消息块的报文结构如图所示：



在消息被分割成几个消息块的过程中，消息负载部分（Message Body）被分割成大小固定的数据块（默认是128字节，最后一个数据块可以小于该固定长度），并在其首部加上消息块首部（Chunk Header），就组成了相应的消息块。

Chunk的默认大小是128字节，在传输过程中，通过一个叫做Set Chunk Size的控制信息可以设置Chunk数据量的最大值，在发送端和接受端会各自维护一个Chunk Size，可以分别设置这个值来改变自己这一方发送的Chunk的最大大小。大一点的Chunk减少了计算每个chunk的时间从而减少了CPU的占用率，但是它会占用更多的时间在发送上，尤其是在低带宽的网络情况下，很可能会阻塞后面更重要信息的传输。小一点的Chunk可以减少这种阻塞问题，但小的Chunk会引入过多额外的信息（Chunk中的Header），少量多次的传输也可能会造成发送的间断导致不能充分利用高带宽的优势，因此并不适合在高比特率的流中传输。在实际发送时应对要发送的数据用不同的Chunk Size去尝试，通过抓包分析等手段得出合适的Chunk大小，并且在传输过程中可以根据当前的带宽信息和实际信息的大小动态调整Chunk的大小，从而尽量提高CPU的利用率并减少信息的阻塞机率。

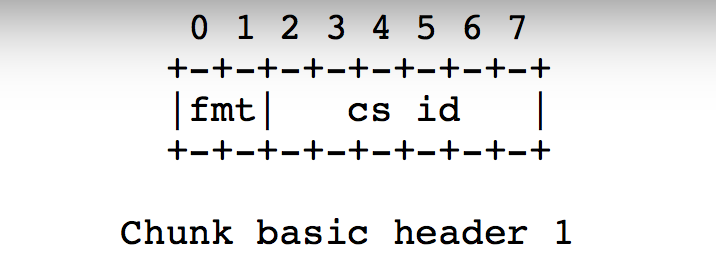
Basic Header(基本的头信息)：

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。

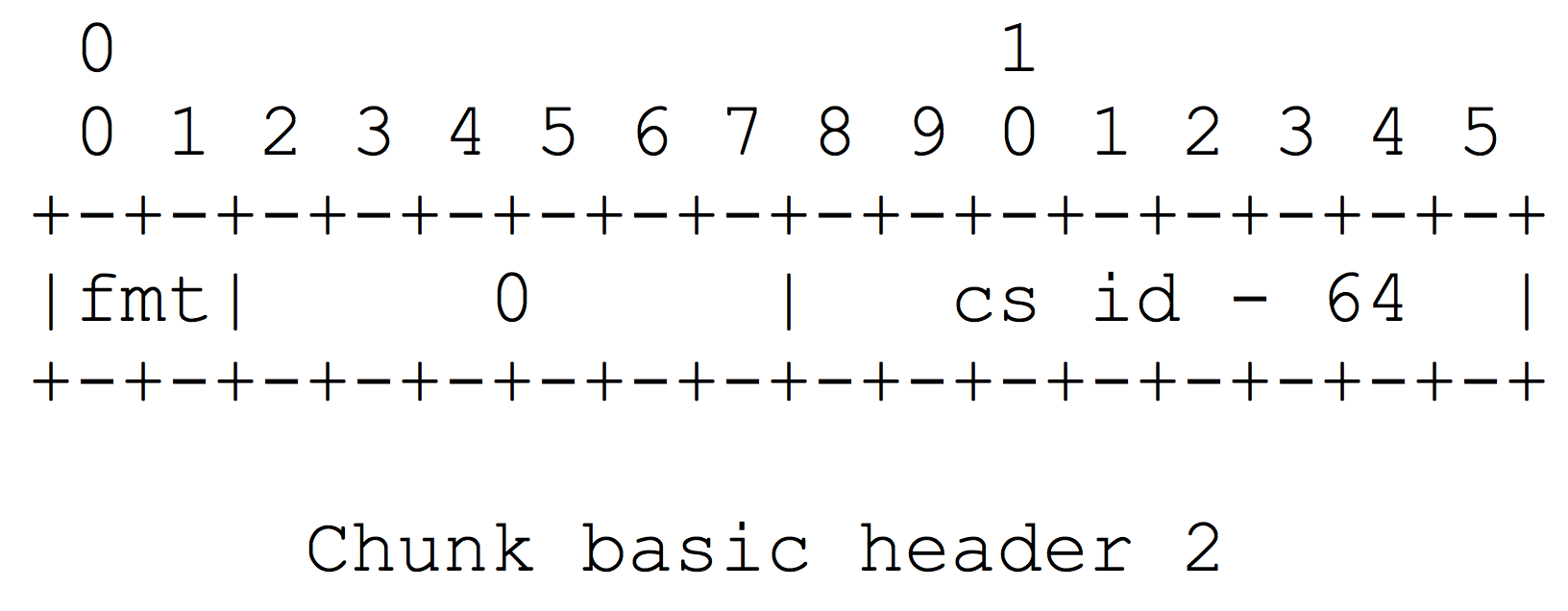
RTMP协议支持用户自定义［3，65599］之间的CSID，0，1，2由协议保留表示特殊信息。0代表Basic Header总共要占用2个字节，CSID在［64，319］之间，1代表占用3个字节，CSID在［64，65599］之间，2代表该chunk是控制信息和一些命令信息，后面会有详细的介绍。

chunk type的长度固定为2位，因此CSID的长度是（6=8-2）、（14=16-2）、（22=24-2）中的一个。

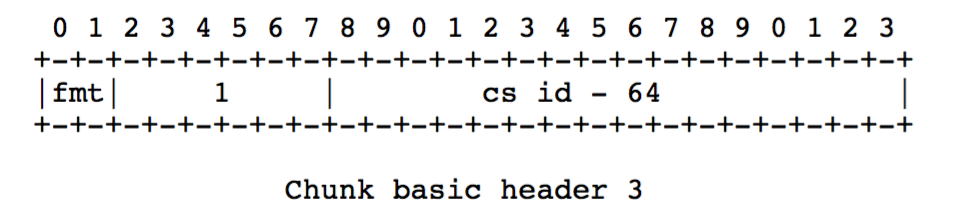
1. 当Basic Header为1个字节时，CSID占6位，6位最多可以表示64个数，因此这种情况下CSID在［0，63］之间，其中用户可自定义的范围为［3，63］。



1. 当Basic Header为2个字节时，CSID占14位，此时协议将与chunk type所在字节的其他位都置为0，剩下的一个字节来表示CSID－64，这样共有8个二进制位来存储CSID，8位可以表示［0，255］共256个数，因此这种情况下CSID在［64，319］，其中319=255+64。



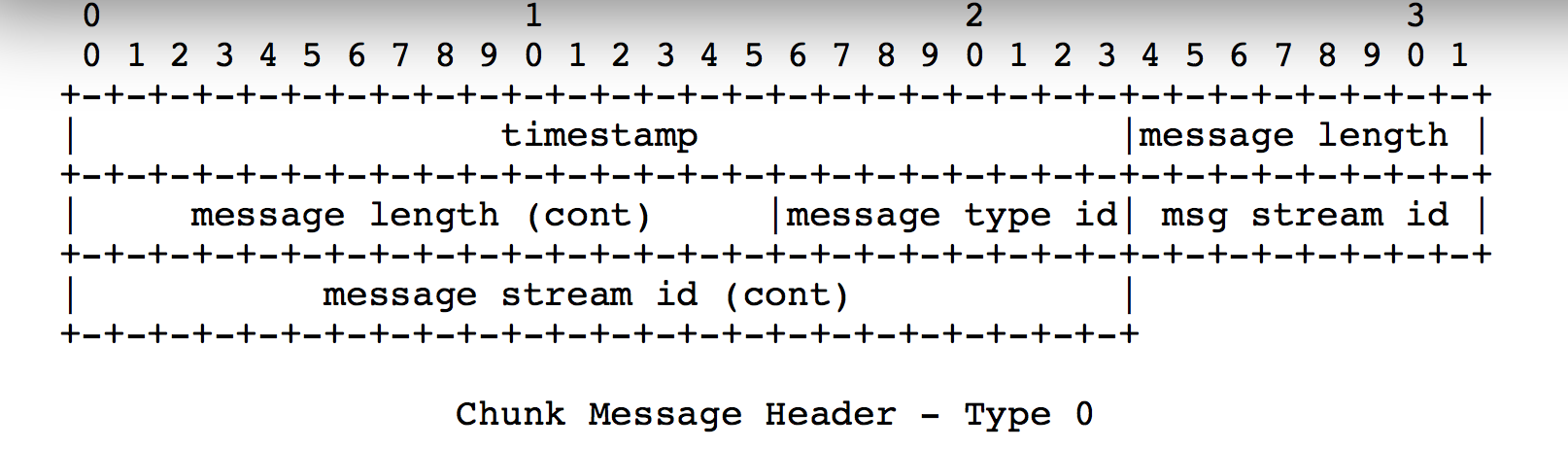
1. 当Basic Header为3个字节时，CSID占22位，此时协议将［2，8］字节置为1，余下的16个字节表示CSID－64，这样共有16个位来存储CSID，16位可以表示［0，65535］共65536个数，因此这种情况下CSID在［64，65599］，其中65599=65535+64，需要注意的是，Basic Header是采用小端存储的方式，越往后的字节数量级越高，因此通过这3个字节每一位的值来计算CSID时，应该是:<第三个字节的值>x256+<第二个字节的值>+64。



Message Header（消息的头信息）

包含了要发送的实际信息（可能是完整的，也可能是一部分）的描述信息。Message Header的格式和长度取决于Basic Header的chunk type，共有4种不同的格式，由上面所提到的Basic Header中的fmt字段控制。其中第一种格式可以表示其他三种表示的所有数据，但由于其他三种格式是基于对之前chunk的差量化的表示，因此可以更简洁地表示相同的数据，实际使用的时候还是应该采用尽量少的字节表示相同意义的数据。以下按照字节数从多到少的顺序分别介绍这4种格式的Message Header。

**Type＝0**：



type=0时Message Header占用11个字节，其他三种能表示的数据它都能表示，但在chunk stream的开始的**第一个chun**k和**头信息中的时间戳后退**（即值与上一个chunk相比减小，通常在回退播放的时候会出现这种情况）的时候必须采用这种格式。

**timestamp（时间戳）**：占用3个字节，因此它最多能表示到16777215=0xFFFFFF=2

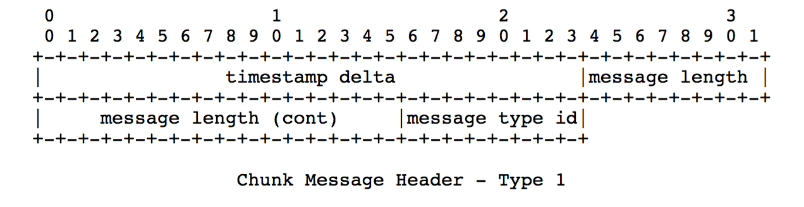
的24次减1, 当它的值超过这个最大值时，这三个字节都置为1，这样实际的timestamp会转存到Extended Timestamp字段中，接受端在判断timestamp字段24个位都为1时就会去Extended timestamp中解析实际的时间戳。

**message length（消息数据的长度）**：占用3个字节，表示实际发送的消息的数据如音频帧、视频帧等数据的长度，单位是字节。注意这里是Message的长度，也就是chunk属于的Message的总数据长度，而不是chunk本身Data的数据的长度。

**message type id(消息的类型id)**：占用1个字节，表示实际发送的数据的类型，如8代表音频数据、9代表视频数据。

**msg stream id（消息的流id）**：占用4个字节，表示该chunk所在的流的ID，和Basic Header的CSID一样，它采用小端存储的方式。

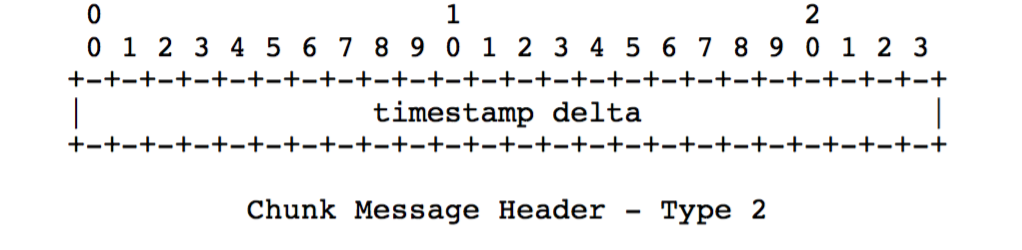
**Type = 1:**



**type=1时Message Header占用7个字节，省去了表示msg stream id的4个字节，表示此chunk和上一次发的chunk所在的流相同，如果在发送端只和对端有一个流链接的时候可以尽量去采取这种格式。**

**timestamp delta：占用3个字节，注意这里和type＝0时不同，存储的是和上一个chunk的时间差。类似上面提到的timestamp，当它的值超过3个字节所能表示的最大值时，三个字节都置为1，实际的时间戳差值就会转存到Extended Timestamp字段中，接受端在判断timestamp delta字段24个位都为1时就会去Extended timestamp中解析时机的与上次时间戳的差值。**

**Type = 2：**



**type=2时Message Header占用3个字节，相对于type＝1格式又省去了表示消息长度的3个字节和表示消息类型的1个字节，表示此chunk和上一次发送的chunk所在的流、消息的长度和消息的类型都相同。余下的这三个字节表示timestamp delta，使用同type＝1。**

**Type = 3：**

**0字节！！！好吧，它表示这个chunk的Message Header和上一个是完全相同的，自然就不用再传输一遍了。当它跟在Type＝0的chunk后面时，表示和前一个chunk的时间戳都是相同的。什么时候连时间戳都相同呢？就是一个Message拆分成了多个chunk，这个chunk和上一个chunk同属于一个Message。而当它跟在Type＝1或者Type＝2的chunk后面时，表示和前一个chunk的时间戳的差是相同的。比如第一个chunk的Type＝0，timestamp＝100，第二个chunk的Type＝2，timestamp delta＝20，表示时间戳为100+20=120，第三个chunk的Type＝3，表示timestamp delta＝20，时间戳为120+20=140。**

**Extended Timestamp（扩展时间戳）：**

**扩展时间戳占4个字节，能表示的最大数值就是0xFFFFFFFF＝4294967295。当扩展时间戳启用时，timestamp字段或者timestamp delta要全置为1，表示应该去扩展时间戳字段来提取真正的时间戳或者时间戳差。注意扩展时间戳存储的是完整值，而不是减去时间戳或者时间戳差的值。**

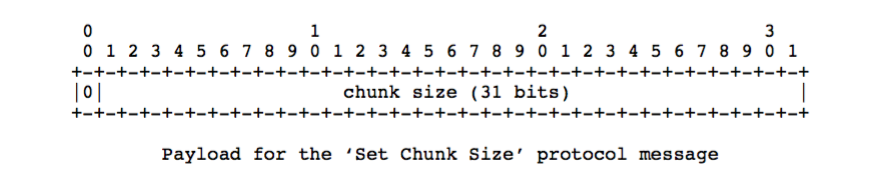
**Chunk Data（块数据）：**

**用户层面上真正想要发送的与协议无关的数据，长度在(0,chunkSize]之间。**

**协议控制消息（Protocol Control Message）**

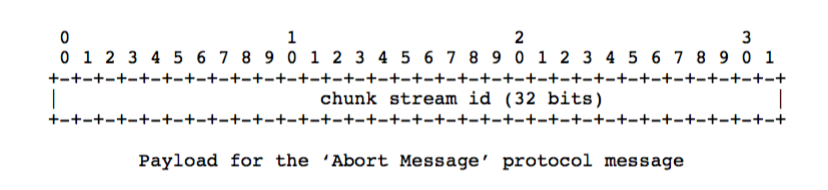
**在RTMP的chunk流会用一些特殊的值来代表协议的控制消息，它们的Message Stream ID必须为0（代表控制流信息），CSID必须为2，Message Type ID可以为1，2，3，5，6，具体代表的消息会在下面依次说明。控制消息的接受端会忽略掉chunk中的时间戳，收到后立即生效。**

**Set Chunk Size(Message Type ID=1):设置chunk中Data字段所能承载的最大字节数，默认为128B，通信过程中可以通过发送该消息来设置chunk Size的大小（不得小于128B），而且通信双方会各自维护一个chunkSize，两端的chunkSize是独立的。**

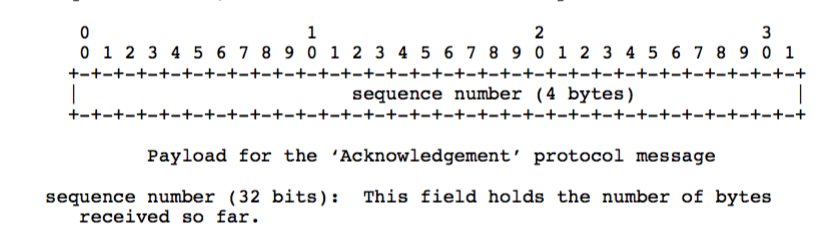


**其中第一位必须为0，chunk Size占31个位，最大可代表2147483647＝0x7FFFFFFF＝231-1，但实际上所有大于16777215=0xFFFFFF的值都用不上，因为chunk size不能大于Message的长度，表示Message的长度字段是用3个字节表示的，最大只能为0xFFFFFF。**

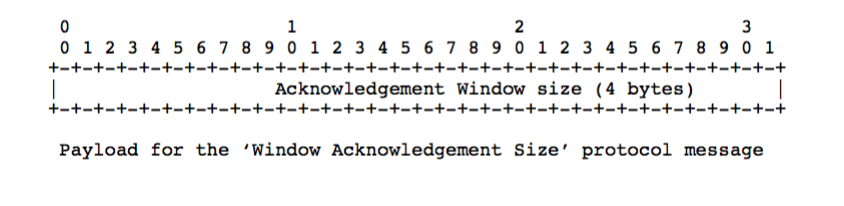
**Abort Message(Message Type ID=2):当一个Message被切分为多个chunk，接受端只接收到了部分chunk时，发送该控制消息表示发送端不再传输同Message的chunk，接受端接收到这个消息后要丢弃这些不完整的chunk。Data数据中只需要一个CSID，表示丢弃该CSID的所有已接收到的chunk。**



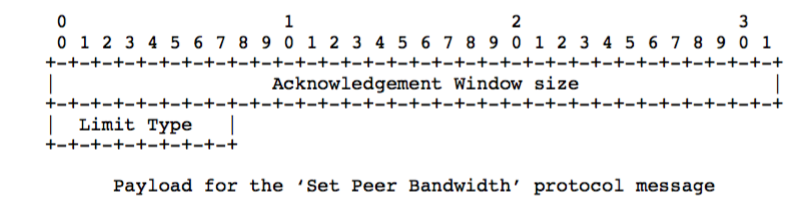
**Acknowledgement(Message Type ID=3):当收到对端的消息大小等于窗口大小（Window Size）时接受端要回馈一个ACK给发送端告知对方可以继续发送数据。窗口大小就是指收到接受端返回的ACK前最多可以发送的字节数量，返回的ACK中会带有从发送上一个ACK后接收到的字节数。**



**Window Acknowledgement Size(Message Type ID=5):发送端在接收到接受端返回的两个ACK间最多可以发送的字节数。**



**Set Peer Bandwidth(Message Type ID=6):限制对端的输出带宽。接受端接收到该消息后会通过设置消息中的Window ACK Size来限制已发送但未接受到反馈的消息的大小来限制发送端的发送带宽。如果消息中的Window ACK Size与上一次发送给发送端的size不同的话要回馈一个Window Acknowledgement Size的控制消息。**



**Hard(Limit Type＝0):接受端应该将Window Ack Size设置为消息中的值**

**Soft(Limit Type=1):接受端可以讲Window Ack Size设为消息中的值，也可以保存原来的值（前提是原来的Size小与该控制消息中的Window Ack Size）**

**Dynamic(Limit Type=2):如果上次的Set Peer Bandwidth消息中的Limit Type为0，本次也按Hard处理，否则忽略本消息，不去设置Window Ack Size。**

**不同类型的RTMP Message**

**Command Message(命令消息，Message Type ID＝17或20)：表示在客户端和服务器间传递的在对端执行某些操作的命令消息，如connect表示连接对端，对端如果同意连接的话会记录发送端信息并返回连接成功消息，publish表示开始向对方推流，接受端接到命令后准备好接受对端发送的流信息，后面会对比较常见的Command Message具体介绍。当信息使用AMF0编码时，Message Type ID＝20，AMF3编码时Message Type ID＝17。**

**Data Message（数据消息，Message Type ID＝15或18）：传递一些元数据（MetaData，比如视频名，分辨率等等）或者用户自定义的一些消息。当信息使用AMF0编码时，Message Type ID＝18，AMF3编码时Message Type ID＝15。**

**Shared Object Message(共享消息，Message Type ID＝16或19)：表示一个Flash类型的对象，由键值对的集合组成，用于多客户端，多实例时使用。当信息使用AMF0编码时，Message Type ID＝19，AMF3编码时Message Type ID＝16。**

**Audio Message（音频信息，Message Type ID＝8）：音频数据。**

**Video Message（视频信息，Message Type ID＝9）：视频数据。**

**Aggregate Message (聚集信息，Message Type ID＝22)：多个RTMP子消息的集合。**

**User Control Message Events(用户控制消息，Message Type ID=4):告知对方执行该信息中包含的用户控制事件，比如Stream Begin事件告知对方流信息开始传输。和前面提到的协议控制信息（Protocol Control Message）不同，这是在RTMP协议层的，而不是在RTMP chunk流协议层的，这个很容易弄混。该信息在chunk流中发送时，Message Stream ID=0,Chunk Stream Id=2,Message Type Id=4。**

**---------下面对以上7种信息具体介绍----------**

**Command Message(命令消息，Message Type ID＝17或20)**

**发送端发送时会带有命令的名字，如connect，TransactionID表示此次命令的标识，Command Object表示相关参数。接受端收到命令后，会返回以下三种消息中的一种：\_result 消息表示接受该命令，对端可以继续往下执行流程，\_error消息代表拒绝该命令要执行的操作，method name消息代表要在之前命令的发送端执行的函数名称。这三种回应的消息都要带有收到的命令消息中的TransactionId来表示本次的回应作用于哪个命令。**

**可以认为发送命令消息的对象有两种，一种是NetConnection，表示双端的上层连接，一种是NetStream，表示流信息的传输通道，控制流信息的状态，如Play播放流，Pause暂停。**

**NetConnection Commands(连接层的命令)**

**用来管理双端之间的连接状态，同时也提供了异步远程方法调用（RPC）在对端执行某方法，以下是常见的连接层的命令：**

**connect:用于客户端向服务器发送连接请求，消息的结构如下：**

**字段**

**类型**

**说明**

**Command Name(命令名字)**

**String**

**命令的名字，如"connect"**

**Transaction ID(事务ID)**

**Number**

**恒为1**

**Command Object(命令包含的参数对象)**

**Object**

**键值对集合表示的命令参数**

**Optional User Arguments（额外的用户参数)**

**Object**

**用户自定义的额外信息**

**第三个字段中的Command Object中会涉及到很多键值对，这里不再一一列出，使用时可以参考协议的官方文档。**

**消息的回应有两种，\_result表示接受连接，\_error表示连接失败**

**4.1.1.2 Call:用于在对端执行某函数，即常说的RPC：远程进程调用，消息的结构如下：**

**字段**

**类型**

**说明**

**Procedure Name(进程名)**

**String**

**要调用的进程名称**

**Transaction ID**

**Number｜如果想要对端响应的话置为非0值，否则置为0**

**Command Object**

**Object**

**命令参数**

**Optional Arguents**

**Object**

**用户自定义参数**

**如果消息中的TransactionID不为0的话，对端需要对该命令做出响应，响应的消息结构如下：**

**字段**

**类型**

**说明**

**Command Name(命令名)**

**String**

**命令的名称**

**TransactionID**

**Number**

**上面接收到的命令消息中的TransactionID**

**Command Object**

**Object**

**命令参数**

**Optional Arguments**

**Object**

**用户自定义参数**

**4.1.1.3 Create Stream：创建传递具体信息的通道，从而可以在这个流中传递具体信息，传输信息单元为Chunk。**

**字段**

**类型**

**说明**

**Command Name(命令名)**

**String**

**"createStream"**

**TransactionID**

**Number**

**上面接收到的命令消息中的TransactionID**

**Command Object**

**Object**

**命令参数**

**Optional Arguments**

**Object**

**用户自定义参数**

**4.1.2 NetStream Commands(流连接上的命令)**

**Netstream建立在NetConnection之上，通过NetConnection的createStream命令创建，用于传输具体的音频、视频等信息。在传输层协议之上只能连接一个NetConnection，但一个NetConnection可以建立多个NetStream来建立不同的流通道传输数据。**

**以下会列出一些常用的NetStream Commands，服务端收到命令后会通过onStatus的命令来响应客户端，表示当前NetStream的状态。**

**onStatus命令的消息结构如下：**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"onStatus"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL**

**对onSatus命令来说不需要这个字段**

**Info Object**

**Object**

**AMF类型的Object，至少包含以下三个属性：1，“level”，String类型，可以为“warning”、"status"、"error"中的一种；2，"code",String类型，代表具体状态的关键字,比如"NetStream.Play.Start"表示开始播流；3，"description"，String类型，代表对当前状态的描述，提供对当前状态可读性更好的解释，除了这三种必要信息，用户还可以自己增加自定义的键值对**

**4.1.2.1 play(播放):由客户端向服务器发起请求从服务器端接受数据（如果传输的信息是视频的话就是请求开始播流），可以多次调用，这样本地就会形成一组数据流的接收者。注意其中有一个reset字段，表示是覆盖之前的播流（设为true）还是重新开始一路播放（设为false）。**

**play命令的结构如下：**

**字段**

**类型**

**说明**

**命令名**

**String**

**"play"**

**事务ID**

**Number**

**恒为0**

**命令参数对象**

**Null**

**不需要此字段，设为空**

**流名称**

**String**

**要播放的流的名称**

**开始位置**

**Number**

**可选参数，表示从何时开始播流，以秒为单位。默认为－2，代表选取对应该流名称的直播流，即当前正在推送的流开始播放，如果对应该名称的直播流不存在，就选取该名称的流的录播版本，如果这也没有，当前播流端要等待直到对端开始该名称的流的直播。如果传值－1，那么只会选取直播流进行播放，即使有录播流也不会播放；如果传值或者正数，就代表从该流的该时间点开始播放，如果流不存在的话就会自动播放播放列表中的下一个流**

**周期**

**Number**

**可选参数，表示回退的最小间隔单位，以秒为单位计数。默认值为－1，代表直到直播流不再可用或者录播流停止后才能回退播放；如果传值为0，代表从当前帧开始播放**

**重置**

**Boolean**

**可选参数，true代表清除之前的流，重新开始一路播放，false代表保留原来的流，向本地的播放列表中再添加一条播放流**

**4.1.2.2 play2（播放）：和上面的play命令不同的是，play2命令可以将当前正在播放的流切换到同样数据但不同比特率的流上，服务器端会维护多种比特率的文件来供客户端使用play2命令来切换。**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"play2"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL,对onSatus命令来说不需要这个字段**

**parameters**

**Object**

**AMF编码的Flash对象，包括了一些用于描述flash.net.NetstreamPlayOptions ActionScript obejct的参数**

**4.1.2.3 deleteStream(删除流)：用于客户端告知服务器端本地的某个流对象已被删除，不需要再传输此路流。**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"deleteStream"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL,对onSatus命令来说不需要这个字段**

**Stream ID（流ID)**

**Number**

**本地已删除，不再需要服务器传输的流的ID**

**4.1.2.4 receiveAudio(接收音频)：通知服务器端该客户端是否要发送音频**

**receiveAudio命令结构如下：**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"receiveAudio"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL**

**对onSatus命令来说不需要这个字段**

**Bool Flag**

**Boolean**

**true表示发送音频，如果该值为false，服务器端不做响应，如果为true的话，服务器端就会准备接受音频数据，会向客户端回复NetStream.Seek.Notify和NetStream.Play.Start的Onstatus命令告知客户端当前流的状态**

**4.1.2.5 receiveVideo(接收视频)：通知服务器端该客户端是否要发送视频**

**receiveVideo命令结构如下：**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"receiveVideo"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL**

**对onSatus命令来说不需要这个字段**

**Bool Flag**

**Boolean**

**true表示发送视频，如果该值为false，服务器端不做响应，如果为true的话，服务器端就会准备接受视频数据，会向客户端回复NetStream.Seek.Notify和NetStream.Play.Start的Onstatus命令告知客户端当前流的状态**

**4.1.2.6 publish(推送数据)：由客户端向服务器发起请求推流到服务器。**

**publish命令结构如下：**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"publish"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL,对onSatus命令来说不需要这个字段**

**Publishing Name（推流的名称）**

**String**

**流名称｜**

**Publishing Type（推流类型）**

**String**

**"live"、"record"、"append"中的一种。live表示该推流文件不会在服务器端存储；record表示该推流的文件会在服务器应用程序下的子目录下保存以便后续播放，如果文件已经存在的话删除原来所有的内容重新写入；append也会将推流数据保存在服务器端，如果文件不存在的话就会建立一个新文件写入，如果对应该流的文件已经存在的话保存原来的数据，在文件末尾接着写入**

**4.1.2.7 seek(定位流的位置)：定位到视频或音频的某个位置，以毫秒为单位。**

**seek命令的结构如下：**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"seek"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL,对onSatus命令来说不需要这个字段**

**milliSeconds**

**Number**

**定位到该文件的xx毫秒处｜**

**4.1.2.8 pause（暂停）：客户端告知服务端停止或恢复播放。**

**pause命令的结构如下：**

**字段**

**类型**

**说明**

**Command Name**

**String**

**"pause"**

**TransactionID**

**Number**

**恒为0**

**Command Object**

**NULL,对onSatus命令来说不需要这个字段**

**Pause/Unpause Flag**

**Boolean**

**true表示暂停，false表示恢复**

**milliSeconds**

**Number**

**暂停或者恢复的时间，以毫秒为单位｜**

**如果Pause为true即表示客户端请求暂停的话，服务端暂停对应的流会返回NetStream.Pause.Notify的onStatus命令来告知客户端当前流处于暂停的状态，当Pause为false时，服务端会返回NetStream.Unpause.Notify的命令来告知客户端当前流恢复。如果服务端对该命令响应失败，返回\_error信息。**

**5. 代表流程**

**5.1 推流流程**

**5.2 播流流程**

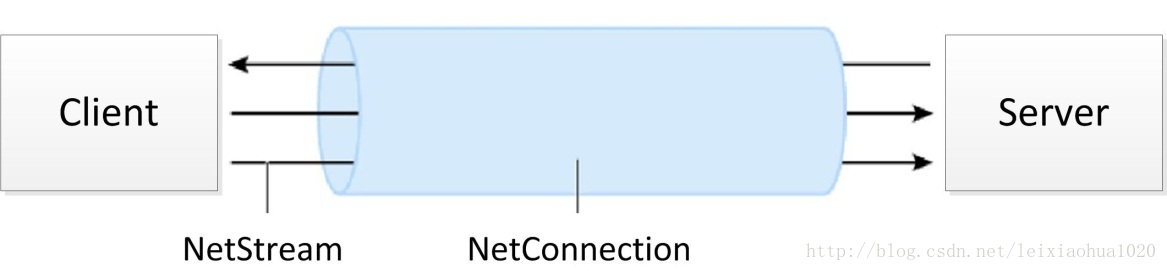
RTMP流媒体播放过程

RTMP协议规定，播放一个流媒体有两个前提步骤：

第一步，建立一个网络连接（NetConnection）；

第二步，建立一个网络流（NetStream）。

其中，网络连接代表服务器端应用程序和客户端之间基础的连通关系。网络流代表了发送多媒体数据的通道。服务器和客户端之间只能建立一个网络连接，但是基于该连接可以创建很多网络流。他们的关系如图所示：



播放一个RTMP协议的流媒体需要经过以下几个步骤：握手，建立连接，建立流，播放。RTMP连接都是以握手作为开始的。建立连接阶段用于建立客户端与服务器之间的“网络连接”；建立流阶段用于建立客户端与服务器之间的“网络流”；播放阶段用于传输视音频数据。

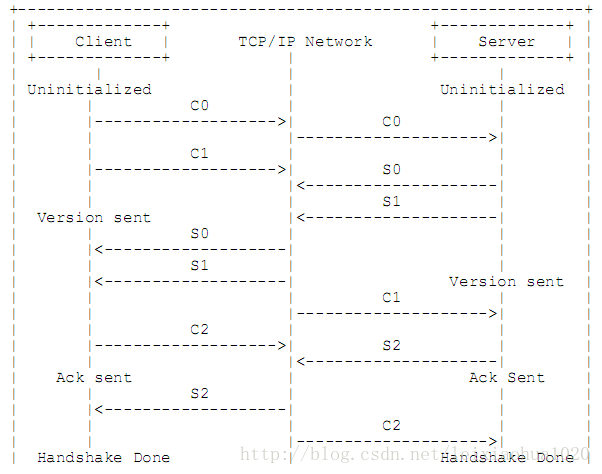
握手（HandShake）

一个RTMP连接以握手开始，双方分别发送大小固定的三个数据块

a） 握手开始于客户端发送C0、C1块。服务器收到C0或C1后发送S0和S1。

b） 当客户端收齐S0和S1后，开始发送C2。当服务器收齐C0和C1后，开始发送S2。

c） 当客户端和服务器分别收到S2和C2后，握手完成。



建立网络连接（NetConnection）

a) 客户端发送命令消息中的“连接”(connect)到服务器，请求与一个服务应用实例建立连接。

b) 服务器接收到连接命令消息后，发送确认窗口大小(Window Acknowledgement Size)协议消息到客户端，同时连接到连接命令中提到的应用程序。

c) 服务器发送设置带宽协议消息到客户端。

d) 客户端处理设置带宽协议消息后，发送确认窗口大小(Window Acknowledgement Size)协议消息到服务器端。

e) 服务器发送用户控制消息中的“流开始”(Stream Begin)消息到客户端。

f ) 服务器发送命令消息中的“结果”(\_result)，通知客户端连接的状态。



建立网络流（NetStream）

1. 客户端发送命令消息中的“创建流”（createStream）命令到服务器端。
2. 服务器端接收到“创建流”命令后，发送命令消息中的“结果”(\_result)，通知客户端流的状态。



播放（Play）

1. 客户端发送命令消息中的“播放”（play）命令到服务器。
2. 接收到播放命令后，服务器发送设置块大小（ChunkSize）协议消息。
3. 服务器发送用户控制消息中的“streambegin”，告知客户端流ID。
4. 播放命令成功的话，服务器发送命令消息中的“响应状态” NetStream.Play.Start & NetStream.Play.reset，告知客户端“播放”命令执行成功。
5. 在此之后服务器发送客户端要播放的音频和视频数据。

