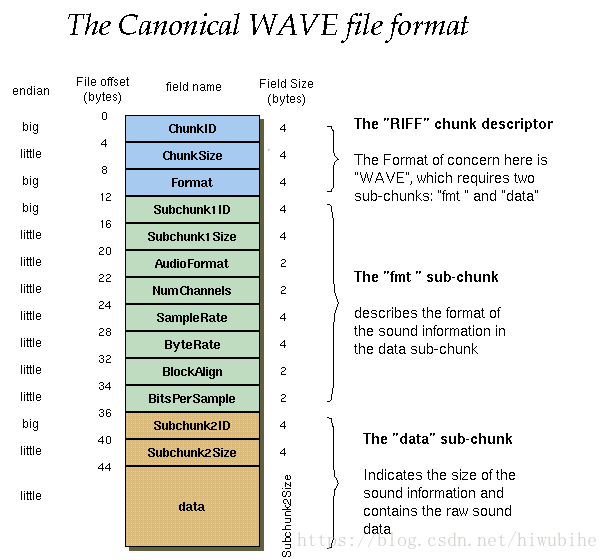
WAV格式介绍

WAV是微软的RIFF文件的一个特例，通常由一个文件头和若干个CHUNK组成，通常是由RIFF文件头类型“WAVE”子chunk为“fmt”,“data”,和可选数量的chunk组成。format如下：







typedef struct \_TWavHeader {

int rId; //标志符（RIFF）

int rLen; //数据大小,包括数据头的大小和音频文件的大小

int wId; //格式类型（"WAVE"）

int fId; //"fmt"

int fLen; //Sizeof(WAVEFORMATEX)

short wFormatTag; //编码格式，包括WAVE\_FORMAT\_PCM，WAVEFORMAT\_ADPCM等

short nChannels; //声道数，单声道为1，双声道为2

int nSamplesPerSec; //采样频率

int nAvgBytesPerSec; //每秒的数据量

short nBlockAlign; //块对齐

short wBitsPerSample; //WAVE文件的采样大小

short nbSize;

short nsamplesperblock;

int f2id; //"fact"

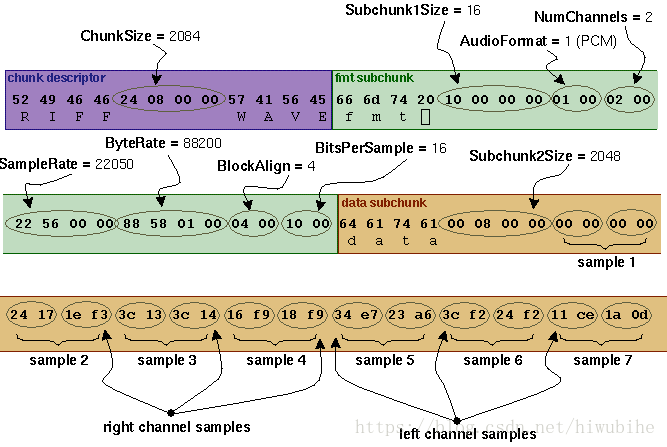
int f2len;

int dataLen; //解码成PCM后数据长度

int dId; //"data"

int wSampleLength; //音频数据的大小

} TWavHeader;



注意事项

1.二进制数据都是"小端"存储方式。

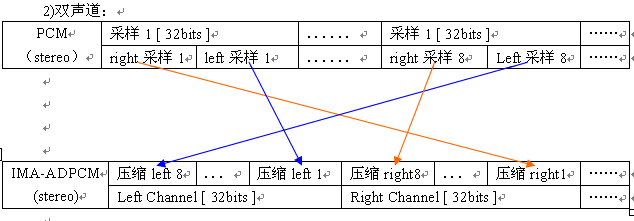
1. 样本存储8位范围为(0-255),样本存储格式16位范围(-32768-32767)

“data”chunk的内部组织

从“data”chunk的第9个字节开始，存储的就是声音信息的数据了，(前八个字节存储的是标志符“data”和后接数据大小size(DWORD)。这些数据可能是压缩的，也可能是没有压缩的。

PCM中的声音数据没有被压缩，如果是单声道的文件，采样数据按时间的先后顺序依次存入。（它的基本组织单位是BYTE(8bit)或WORD(16bit)）如果是双声道的文件，采样数据按时间先后顺序交叉地存入。

IMA-ADPCM是压缩格式，它是从PCM的16位采样压缩成4位的。对于单声道的IMA-ADPCM来说，它是将PCM的数据按时间次序依次压缩并写入文件中的，每个byte中含两个采样，低四位对应第一个采样，高四位对应第二个采样。而对于双声道的IMA-ADPCM来说，它的存储相对就麻烦一些了，它是将PCM的左声道的前8个采样依次压缩并写入到一个DWORD中，然后写入“data”chunk里。紧接着是右声道的前8个采样。以此循环，当采样数不足8时（到数据尾端），应该把多出来的采样用0填充。



在IMA-ADPCM中，“data”chuck中的数据是以block形式来组织的，我把它叫做“段”，也就是说在进行压缩时，并不是依次把所有的数据进行压缩保存，而是分段进行的，这样有一个十分重要的好处：那就是在只需要文件中的某一段信息时，可以在解压缩时可以只解所需数据所在的段就行了，没有必要再从文件开始起一个一个地解压缩。这对于处理大文件将有相当的优势。同时，这样也可以保证声音效果。

Block一般是由block header (block头) 和data 两者组成的。其中block header是一个结构，它在单声道下的定义如下：

Typedef struct

{

short  sample0;    //block中第一个采样值（未压缩）

BYTE  index;     //上一个block最后一个index，第一个block的index=0;

BYTE  reserved;   //尚未使用

}MonoBlockHeader;

有了blockheader的信息后，就可以不需要知道这个block前面和后面的数据而轻松地解出本block中的压缩数据。对于双声道，它的blockheader应该包含两个MonoBlockHeader其定义如下：

typedaf struct

{

MonoBlockHeader leftbher;

MonoBlockHeader rightbher;

}StereoBlockHeader;

在解压缩时，左右声道是分开处理的，所以必须有两个MonoBlockHeader;

注1：上述的index是解压缩算法中必须用到的一个参数。详见后面。

注2: 关于block的大小,通常会有以下几种情况:

对于单声道,大小一般为512byte,显然这里面可以保存的sample个数为(512-sizeof(MonoBlockHeader))/4 + 1 = 1017个<其中"+1"是第一个存在头结构中的没有压缩的sample.

对于双声道,大小一般为1024byte,按上面的算法可以得出,其中的sample个数也是1017个.

４．读取WAVE文件的方法．

在知道了WAVE文件的内部数据组织后，可以直接通过FILE或HFILE来实现文件的读取。但由于WAVE文件是以RIFF格式来组织的，所以用多媒体输入输出流来操作将更加方便，可以直接在文件中查找chunk并定位数据。

三、IMA-ADPCM 编码和解码算法

IMA-ADPCM 是Intel公司首先开发的是一种主要针对16bit采样波形数据的有损压缩算法, 压缩比为4:1.它与通常的DVI-ADPCM是同一算法。(对8bit数据压缩时是3.2:1,也有非标准的IMA-ADPCM压缩算法,可以达到5:1甚至更高的压缩比)4:1的压缩是目前使用最多的压缩方式。

　　ADPCM(Adaptive Differential Pulse Code Modulation 差分脉冲编码调制)主要是针对连续的波形数据的, 保存的是相临波形的变化情况, 以达到描述整个波形的目的。算法中必须用到两个一维数组，setptab[] 和index\_adjust[]，附在下面的代码之后。

--------------------------------------------------------------------------------

IMA-ADPCM 压缩过程

首先我们认为声音信号都是从零开始的,那么需要初始化两个变量

int index = 0,prev\_sample = 0;

但在实际使用中，prev\_sample的值是每个block中第一个采样的值。（这点在后面的block中会详细介绍）

假设已经写好了两个函数：

GetNextSamp() ——得到一个16bit 的采样数据；

SaveComCode() ——保存一个4bit 的压缩样品；

下面的循环将依次压缩声音数据流：

while (还有数据要处理) {

cur\_sample = GetNextSamp();         // 得到PCM中的当前采样数据

diff = cur\_sample-prev\_sample;      // 计算出和上一个的增量

if (diff<0)

{

diff=-diff;

fg=8;

}

else fg=0;                          // fg 保存的是符号位

code = 4\*diff / steptab[index];

if (code>7) code=7;                 // 根据steptab[] 得到一个0~7 的值，它描述了采样振幅的变化量

index+=index\_adjust[code];          // 根据声音强度调整下次取steptab 的序号，便于下次得到更精确的变化量的描述

if (index<0) index=0;               // 调整index的值

else if (index>88) index=88;

prev\_sample=cur\_sample;

SaveComCode(code|fg);                 // 加上符号位保存起来

}

--------------------------------------------------------------------------------

IMA-ADPCM 解压缩过程

解压缩实际是压缩的一个逆过程,假设写好了以下两个函数：

GetNextCode() ——得到一个编码（4bit）

OutputSamp() ——将解码出来的声音信号保存起来（16bit）.

int index=0,cur\_sample=0;

while (还有数据要处理) {

code=GetNextCode();           // 得到下一个压缩样品Code 4bit

if ((code & 8) != 0) fg=1 else fg=0;

code&=7;                      // 将code 分离为数据和符号

diff = (steptab[index]\*code) /4 + steptab[index] / 8;   // 后面加的一项是为了减少误差

if (fg==1) diff=-diff;

cur\_sample+=diff;            // 计算出当前的波形数据

if (cur\_sample>32767) OutputSamp(32767);

else if (cur\_sample<-32768) OutputSamp(-32768);

else OutputSamp(cur\_sample);

index+=index\_adjust[code];

if (index<0) index=0;

if (index>88) index=88;

}

1. 关于DPCM

DPCM是differential pulse code modulation的缩写，也就是差分脉冲编码调制的意思。他的主要思想是通过已知的数据预测下一个数据，然后传递预测值与实际值之间的差值。具体的细节可以在很多信号处理相关的书上找到。

一般的DPCM编码器都是采用的线性预测。假设传递的数据是X1,X2,...Xn,而下一个数据，Xn+1还是未知。可以通过前面的X1,X2,...Xn的加权和来预测Xn+1，也就是

Xn+1 = ∑(Ai\*Xi),其中i属于1...n

为了简化计算，大部分编码的实现只取前两项，也就是，Xn+1 = a\*Xn + b\*Xn-1, 现在，最主要的事情就是如何对a,b进行取值，才能使得Xn+1的误差最小。

如果假设 x~i 是预测值，xi是实际值，那么，∑(x~i-xi)^2 最小的时候，a,b就是最优的。设 F=∑(X~i-Xi)^2,因为 X~i = a\*X~i-1 + b\*X~i-2,可以得出，F是关于a,b的二元函数.也就是 F=f(a,b) 。可以分别对a和b求偏导数,求出它的极值点。

f<sub>a</sub>(a,b) = 0 ;

f<sub>b</sub>(a,b) = 0 ;

可以得到

a \* ∑(Xi-1)^2 + b \* ∑(Xi-1)\*(Xi-2) = ∑Xi\*Xi-1

a \* ∑(Xi-1)\*(Xi-2) + b \* ∑(Xi-2)^2 = ∑Xi\*Xi-2

如果设

alpha = ∑(Xi-1)^2

beta = ∑(Xi-1)\*(Xi-2)

gama = ∑(Xi-2)^2

m = ∑Xi\*Xi-1

n = ∑Xi\*Xi-2

上面的式子就可以写成

a\*alpha + b\*beta = m

a\*beta + b\*gama = n

算出alpha,beta,gama,m,n以后，a和b的值就可以计算出来了,实际上我们只需要一个循环遍历前n个数就能把它们都求出来。

2. ADPCM的思想

如果直接使用DPCM进行编码的话，是得不到什么压缩的效率的。缘故是，需要传输或保存的是预测值后的值与实际值之间的差值，这与原来的数据占用同样的空间。

为了满足我们原始的压缩数据的动机，你可以对这些差值进行各种各样的编码。因为，大部分情况下，差值都是像1,1,1,2,3,5,5,5之类的数。可以对它们进行通常的游程编码或者huffman编码，运气好的话能够得到很大的压缩比。

这样做会有一个很大的弊端。因为有些数据可能之间的联系会呈线性或者某种连续函数的性质。但是大部分情况下，数据的分布还是有一定的离散性的。当数据之间出现很大的跳跃的时候，这种方法就显得很苍白无力了。

我们可以这么做，每次对得到的差值用一个随着差值大小变化的数来除。这样就可以随着差值的变化，不断调整比例因子。这样出现较大的跳跃时也能把我们要存储的差值限定在一个较小的范围之内。

如果你现在有些迷惑，没事，我们换种方式来说明一下。

假设差值是 diff，也就是 diff = X~i - Xi，那么，diff就有可能变动很大，如果引入一个不断变化的因子iDelta,那么，diff' = diff / iDelta，而对于iDelta，每当diff变大的时候，他就变大比较大，当diff变得比较小的时候，他就相应的减小。这样，我们的diff'就能保持相对的稳定了。通过iDelta的引入，可以使得我们的DPCM编码自动的适应数据间大幅度的跳跃。这就是自适应脉冲编码调制，ADPCM的主要思想。

你现在可能会想，iDelta到底怎么变化，才能自动的匹配diff的变化? 一种可行的方法就是，把它定义为diff的一个函数，这个函数根据不同的diff的值的大小取不同大小的值。通常我们会做一个iDelta值的表，通过diff作为索引，这样，就可以根据不同的diff值，iDelta就可以作相应的变化了

编码：

对每一个block，编码的过程是通过下面的几个步骤进行的

确定需要使用的系数predictor

确定初始的iDelta值

输出block header

编码并输出数据

其中predictor的确定已经在开头部分详细的描写了。我们只要找出与a和b最相近的系数表的索引值就行了。

初始的idelta的值确定有很多种，你可以对每一个block都使用相同的iDelta值。也可以通过开始的几个数据来估计iDelta，比如，通过算出第一个sample的预测值跟实际值之间的diff，可以使用diff/4来表示iDelta，这样iErrorDelta的值就能刚好被限制在-8到7的比较靠中间的位置

每一个block的初始iDelta的值也可以使用前一个block的最后一个iDelta的值来确定。不过需要注意，第一个block的初始iDelta的值需要单独考虑。

当predictor和初始的iDelta确定之后，block header就可以写出了。

首先将每一个channel的predictor输出

再将每一个channel的iDelta输出

接着将16bit的第二个sample的PCM值输出(iSample1)

最后将16bit的第一个sample的PCM值输出(iSample2)

下面就是对剩下的数据编码了。

如果block里还有sample尚待编码

// 通过前两个sample预测下一个sample的值

lPredSamp = ((iSamp1 \* iCoef1) + (iSamp2 \*iCoef2)) / 256

// 计算iErrorDelta

iErrorDelta = (Sample(n) - lPredSamp) / iDelta

如果iErrorDelta大于7，把它设成7

如果iErrorDelta小于-8，把它设成-8

把iErrorDelta作为一个nibble输出

//算出使用iDelta和iErrorDelta预测得到的新的sample的值

lNewSamp = lPredSample + iDelta \* iErrorDelta

把lNewSamp限定到16bit所允许的大小

//调整iDelta的值

iDelta = iDelta \* AdaptionTable[ iErrorDelta] / 256

把iDelta限定到16bit所允许大小的范围内，并确保它不为0

//更新预测用的两个sample的值

iSamp2 = iSamp1;

iSamp1 = lNewSample.

重复上面的过程，直到block里没有需要再编码的sample

解码实际上就是上述过程的逆过程。如果你把上边的编码过程搞明白了，解码的过程就很简单，这里就不详细说了。