markdown写出来怎么这么丑啊！(｡˘•ε•˘｡)

写在前面

一开始是打算用这个老接口做讯飞语音识别的程序，在转移到UWP时发现，这玩意在Windows Runtime中屏蔽（弃用）了，将来会更新使用WASAPI的程序

  [WaveRecorder类代码下载地址](http://download.csdn.net/detail/github_32886825/9309271" \t "_blank)

录音无非两种需求：   
       1. 非实时获得音频，也就是停止录音了你才需要处理它；   
       2. 实时获取音频，比如QQ电话这种这边讲话那边马上就听到的。   
       后者实现起来比较啰嗦，但也很定式。既然啰嗦那就乖乖地写成类吧，管别的大仙怎么说你low呢 ( ͡° ͜ʖ ͡°) 。   
              

  首先你要知道…

| **Windows API 可用的实现方式** | **用处** |
| --- | --- |
| Windows Multimedia API（WaveXxxAPI） | 定式且啰嗦地实现音频流的实时获取。 |
| Media Control Interface (MCI) | 让你简单地用字符串命令实现录音和放音 |

——**MMAPI可以让你访问音频设备的缓冲区，发挥比较自由，接近系统底层。**   
——**MCI是发送字符串指令给这个界面，内部的无法干涉，记得《少年电脑世界》之类的杂志教你打开关闭光驱什么的就是用写一行**   
          mciSendString("set cdaudio door open",NULL,0,NULL);//关闭把open改为close。   
          注意：MCI简称媒体控制接口，较为高层，为的是让你不用关心设备的具体操作，快速上手，简单地操作多媒体设备。WaveXxxAPI是接近底层的应用程序接口，为的是灵活地控制设备，但设备的操作还是比较定式的，灵活在于设备状态配置和数据处理的时机，所以M$给你了两个控制方式。

     下面我们来看看如何用这两种方式实现录音：

使用MMAPI(WaveXxxAPI)

  怎么和音频流打交道

       用MMAPI的估计都是想实时获得音频数据的，MMAPI可以把音频流缓冲起来并一块一块地发送给你，在这里我们暂把这种固定大小的音频裸数据简称为AudioFrame（数据块，代码中别名叫ChunkData）。这一块数据需要你一次性处理完（你甚至需要转移这个数据到另一个线程以保证缓冲区读写的稳定性），数据有多少字节可以根据实际情况来设定。在性能和延迟之间均衡考虑一下，200ms的数据可以应付大多数情况。   
       PS:用过Kinect V2麦克风的同学对此应该比有印象，AudioFrame顾名思义，音频帧。

  操作流程

       MMAPI的操作十分定式:   
       开始录音的流程为:以一种格式打开波形输入设备，发送WIM\_OPEN消息给回调函数，准备缓冲区，添加缓冲区到设备，告诉设备录音开始;   
       录音期间循环发送WIM\_DATA给回调函数;   
       结束录音的流程为:告诉设备录音结束并发送WIM\_DATA给回调函数让它处理最后的数据，重置录音设备，释放缓冲区，**至此可以重新设置缓冲区到设备并开始新的录音**，关闭设备并发送WIM\_CLOSE给回调函数。   
    
       关于MMAPI的回调函数：   
       这个回调函数是来处理消息的，一开始收到WIM\_OPEN，最后收到WIM\_CLOSE唯一频繁收到的消息是WIM\_DATA，得到这个消息时我们需要转移缓冲区里的数据并把缓冲区压入到设备缓冲队列中，你可以理解为自动pop手动push。   
       我写的类里面的回调函数是属于这个回调函数的，阻塞的话还是会直接影响MMAPI的回调函数   
    
       这里用到双缓冲乃至多缓冲技术:   
       假设一个实时接水的任务，听起来奇怪但与MMAPI的处理流程相似，这里需要你用杯子连续接水，杯子相当于你开辟的缓冲区：   
       根据上面的​流程，你的身份是MMAPI的回调函数，在饮水机面前要拿着杯子执行这个任务:任务的基本指标是**滴水不漏地连续用杯子把水接到一个存储区域里**，你要接指定量的水，还要负责转移走这杯水。为了能腾出时间把水倒在存储区域里，你肯定需要用不止一个杯子轮流接水。饮水机有一个功能：每当杯子灌满后，饮水机会通知你，并自动去接下一个杯子，如果后面没有杯子则终止任务。（不会讲故事的我啊TT，这个奇葩例子能看懂就行）可以见得：1.你会被及时地通知去转移数据2.缓冲区用完了要及时放回缓冲区队列后端以保证任务能够继续3.如果转移并处理数据的时间不是很稳定，你可能需要准备多个缓冲区而不是单纯增加缓冲区容量，为的是确保任务中能够预留足够多的容忍时间供你使用。   
       这里的多缓冲技术浅显地解释就是多个缓冲区排成一个队列（或者理解为放成一摞）来抵消这个任务中那些耗时不稳定的处理过程对整个实时处理任务的连续性带来的负面影响。其实生活中有很多事情也是用到多缓冲这个概念。

  代码

    这一部分我把自己写的类里面的函数拿了出来,完整代码请见链接，免下载积分。

    需要添加：

#include "mmsystem.h"

// using namespace std;

#pragma comment(lib, "winmm.lib")

    检查一下是否有音频输入设备：

if (!waveInGetNumDevs())

cout << "没有找到音频输入设备" << endl;

     写一个WaveXxxAPI的回调函数：

DWORD WaveRecorder::WaveXAPI\_Callback(HWAVEIN hwavein, UINT uMsg, DWORD dwInstance, DWORD dwParam1, DWORD dwParam2)

{

// 消息switch

switch (uMsg)

{

case WIM\_OPEN: // 设备成功已打开

// 在这里添加打开时要做的

break;

case WIM\_DATA: // 缓冲区数据填充完毕

// 把缓冲区数据拷贝出来

memcpy(ChunkData.data(), ((LPWAVEHDR)dwParam1)->lpData, CHUNCK\_SIZE);

waveInAddBuffer(hwavein, (LPWAVEHDR)dwParam1, sizeof(WAVEHDR));//用完了添加到缓冲区

// 没有录进去的被填充为0xcd,改成0来避免末尾出现爆音【只在结束录音时进行，不影响添加缓存效率】

// 你需要在这里检查dwBytesRecorded的数值，小于CHUNCK\_SIZE则需要你把后面的0xcd改为0或者在写入文件的时候忽略后面【只在刚执行stop的时候出现，不必担心占用时间】

// 拷贝出来供外部使用

RawData.push\_back(ChunkData);;

// 如果需要停止录音则不继续添加缓存

if (!stop)

{

waveInAddBuffer(hwavein, (LPWAVEHDR)dwParam1, sizeof(WAVEHDR));// 压入缓冲区

}

else

{

// 防止结束时重复记录数据

dat\_ignore = true;

}

break;

case WIM\_CLOSE: // 操作成功完成

// 在这里添加关闭设备时要做的

break;

default:

break;

}

return 0;

}

     配置Wave波形格式：

WAVEFORMATEX pwfx;

void WaveRecorder::WaveInitFormat(LPWAVEFORMATEX WaveFormat, WORD Ch, DWORD SampleRate, WORD BitsPerSample)

{

// 自动配置参数

WaveFormat->wFormatTag = WAVE\_FORMAT\_PCM; // 简直是废话，就一种类型

WaveFormat->nChannels = Ch; // 声道数

WaveFormat->nSamplesPerSec = SampleRate; // 采样率

WaveFormat->nAvgBytesPerSec = SampleRate \* Ch \* BitsPerSample / 8; // 每秒平均数据量

WaveFormat->nBlockAlign = Ch \* BitsPerSample / 8; // 单帧数据量

WaveFormat->wBitsPerSample = BitsPerSample; // 采样位深，也就是采样精度

WaveFormat->cbSize = 0; // 保留

}

     尝试打开输入设备并准备缓冲区：

HWAVEIN hwi; // 音频输入设备

static WAVEHDR pwh[BUFFER\_LAYER]; // 硬件缓冲区

waveInOpen(&hwi, WAVE\_MAPPER, &pwfx,(DWORD)WaveXAPI\_Callback,NULL,CALLBACK\_FUNCTION);

for (int layer = 0; layer < BUFFER\_LAYER; layer++)

{

// 配置缓冲区

pwh[layer].lpData = new char[CHUNCK\_SIZE]; // 8位缓冲区

pwh[layer].dwBufferLength = CHUNCK\_SIZE; // 缓冲区大小

pwh[layer].dwBytesRecorded = 0; // 这里是已填充字节的计数，对于结束时未填充的可以自行处理

pwh[layer].dwUser = layer; // 这是用户自定义数据，这里我将其设为缓冲层数的编号

pwh[layer].dwFlags = 0; // 用不着，保留

pwh[layer].dwLoops = 0; // 用不着，保留

pwh[layer].lpNext = NULL; // 用不着，保留

pwh[layer].reserved = 0; // 保留的保留（笑）

// 部署缓存

waveInPrepareHeader(hwi, &pwh[layer], sizeof(WAVEHDR)); // 配置头数据

waveInAddBuffer(hwi, &pwh[layer], sizeof(WAVEHDR)); // 压入缓冲区

}

// 初始化裸数据缓存

RawData.clear();

RawData.reserve(10);

// 发送录音开始消息

waveInStart(hwi);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

     这时我们可以干其他的事情，因为录音阶段所有的数据都由刚设置的回调函数处理：

     停止录制：

// 停止标记

stop = true;

// 设备停止

waveInStop(hwi);

waveInReset(hwi);

// 释放缓冲区

for (int layer = 0; layer<BUFFER\_LAYER; layer++)

{

waveInUnprepareHeader(hwi, &pwh[layer], sizeof(WAVEHDR));

delete pwh[layer].lpData;

}

// 关闭设备并发出WIM\_CLOSE

waveInClose(hwi);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

     我们还可以写入wav文件保存起来

/\* wav音频头部格式 \*/

typedef struct WAVEPCMHDR

{

char riff[4]; // = "RIFF"

UINT32 size\_8; // = FileSize - 8

char wave[4]; // = "WAVE"

char fmt[4]; // = "fmt "

UINT32 fmt\_size; // = PCMWAVEFORMAT的大小 :

//PCMWAVEFORMAT

UINT16 format\_tag; // = PCM : 1

UINT16 channels; // = 通道数 : 1

UINT32 samples\_per\_sec; // = 采样率 : 8000 | 6000 | 11025 | 16000

UINT32 avg\_bytes\_per\_sec; // = 每秒平均字节数 : samples\_per\_sec \* bits\_per\_sample / 8

UINT16 block\_align; // = 每采样点字节数 : wBitsPerSample / 8

UINT16 bits\_per\_sample; // = 量化精度: 8 | 16

char data[4]; // = "data";

//DATA

UINT32 data\_size; // = 裸数据长度

} WAVEPCMHDR;

/\* 默认wav音频头部数据 \*/

WAVEPCMHDR WavHeader =

{

{ 'R', 'I', 'F', 'F' },

0,

{ 'W', 'A', 'V', 'E' },

{ 'f', 'm', 't', ' ' },

sizeof(PCMWAVEFORMAT) ,

WAVE\_FORMAT\_PCM,

1,

SAMPLE\_RATE,

SAMPLE\_RATE\*(SAMPLE\_BITS / 8),

SAMPLE\_BITS / 8,

SAMPLE\_BITS,

{ 'd', 'a', 't', 'a' },

0

};

// 编辑并写入Wave头信息

WavHeader.data\_size = CHUNCK\_SIZE\*RawData.size();

WavHeader.size\_8 = WavHeader.data\_size + 32;

fwrite(&WavHeader, sizeof(WavHeader), 1, fp);

// 追加RawData

fwrite(RawData.data(), CHUNCK\_SIZE\*RawData.size(), 1, fp);

// 写入结束

fclose(fp);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46

    详细的函数说明及理解可以[参考另一人个的博客](http://www.cnblogs.com/matrix-r/p/3523303.html" \t "_blank)   
    自己很仓促地写了一个WaveRecorder类，想直接用的可以回到顶部下载，可以满足大多数人的需求，但里面用到了STL。   
    MMAPI的实时获取音频数据的实现就写到这

使用MCI（字符串命令）

MCI看起来比较文艺（笑），用起来也是简单到几句话而已，虽然是非实时的，但是行数跟MMAPI差距也太大了吧！

//　　设置位深：

mciSendString ("set wave bitpersample 8", "", 0, 0);

//　　设置采样率：

mciSendString ("set wave samplespersec 16000", "", 0, 0);

//　　设置声道数：

mciSendString ("set wave channels 1", "", 0, 0);

//　　设置WAVEPCM：

mciSendString ("set wave format tag pcm","", 0, 0);

//　　打开设备：

mciSendString ("open new type WAVEAudio alias WREC",0&,0,0); // 我见很多人写alias movie，这里就是alias个代号

//　　开始录音：

mciSendString ("record WREC",0&,0,0);

//　　停止录音：

mciSendString ("stop WREC",0&,0,0);

//　　保存录音

mciSendString ("save WREC C:\\test.wav",0&,0,0); // 绝对路径

//　　关闭设备：

mciSendString ("close WREC",0&,0,0);

