语音服务Linux SDK

功能说明

ASR 自动语音识别技术（Automatic Speech Recognition）， 将人的语音转换为文本。

SDK支持对语音数据流进行识别，即边读取语音流边处理：

- 支持音频格式：PCM
- 采用率： 16K，8K
- 识别服务可同时返回中间结果，最终结果

编译运行环境

目前仅支持 x64 Linux 操作系统。

目前支持运行centos 7+ 版本，及Ubuntu 14+， 使用g++ xxx 编译。 其它Linux及g++版本暂时未做测试，请自行尝试编译，如果有报错请反馈。

版本历史

- 当前最新版本：1.0，发布日期：2019年4月25号。

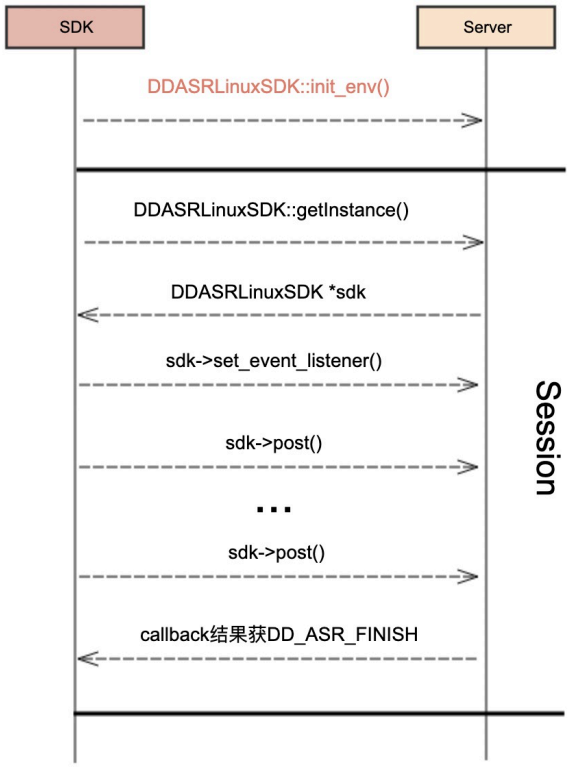
版本,DDASRLinuxSDK::get_sdk_version()	发布时间	libDDSpeechLinuxSDK.a md5值
BuildTime:2019-04-24 11:53:20,Version:1.0,GitVersion:045cd6	2019-4-25	6c2bf2a8a37e53370aaf0a82ef9dc5c0

SDK 结构

/libDDSpeechLinuxSDK

- include 头文件目录
  - utilcxx
  - vad
  - DDASRLinuxSDK.h SDK头文件，主要包含通用类型定义，SDK接口
  - DDASRMessage.h SDK头文件,主要包含消息结构体信息
- lib
  - libcurl.a
  - libcurlreq.a
  - libglog.a
  - libjson_libmt.a
  - liblog.a
  - libutilcxx.a
  - libvadbv.a
  - libvadbvopus.a
  - libDDSpeechLinuxSDK.a SDK库
- Makefile demo的编译文件
- demo.cpp 测试demo
- 8k-test.pcm 测试demo使用的pcm文件
- readme.txt 说明文件

流程图



数据结构说明

1.消息结构体

DDASRMessage ： 主要用于SDK参数设置以及相关操作

成员	类型	，描述
name	Msg_t	DD_ASR_CMD_PUSH_AUDIO 传输音频数据 DD_ASR_CMD_STOP 停止传输音频数据
void set_parameter(Param_t type,char *buf,int read_cnt);		type=DD_DATA_CHUNK，表示设置传递传递音频数据的信息 buf：音频缓冲区 read_cnt：音频长度（1280~PACK_LEN 字节）

ASRResultM ： ASR识别结果传输结构

成员 / 格式	类型	描述
Param_t get_status();		获取识别结果类型： DD_ASR_MIDDLE=105 中间结果 DD_ASR_ERR=108 没有识别出结果 DD_ASR_FINISH=107 最终识别结果
void get_asr_content(string &asr);		获取ASR识别结果，见ASR内容结果举例

ASR内容结果举例
 	
{
"header": {
"status": 107
},
"payload": {
"asr_content":
"{\"nbest\":[\"今天天气怎么样\"]}\n",
"begintime": 138,
"endtime": 144
}
	JSON
	字符串
		header：表示头信息说明
		
		status：表示识别结果状态，内容：DD_ASR_MIDDLE / DD_ASR_ERR / DD_ASR_FINISH
		
		payload：识别内容
		
		asr_content：asr文本，json格式
		
		begintime： 语音说话内容在语音中的开始位置偏移
		
		endtime： 语音说话内容在语音中的结束位置偏移

2.常量定义

名称	类型	描述
RATE_8K=8 RATE_16K=16	枚举	标识音频码率
DEBUG = 0, NOTICE , TRACE, INFO, WARNING, ERROR, FATAL	枚举	日志级别
PACK_LEN=3200	宏	
DD_DATA_CHUNK	枚举	参数消息结构体
DD_ASR_CMD_PUSH_AUDIO, DD_ASR_CMD_STOP, DD_ASR_CMD_CANCEL	枚举	参数消息结构体
SUCCESS = 0, ERR_MEM, ERR_INIT, ERR_SESSION, ERR_NUM_EXCEED, ERR_VAD, ERR_SERVER, ERR_PARAM, ERR_OTHER, VAD_END	枚举	SUCCESS 成功 ERR_MEM 内存错误 ERR_INIT SDK初始化错误 ERR_SESSION SDK内部 session创建失败 ERR_NUM_EXCEED 并发超过限制 ERR_VAD 音频VAD失败 ERR_PARAM 参数错误 ERR_OTHER 其他错误 VAD_END 当前语音检测到说话结束

3.SDK 接口

DDASRLinuxSDK：主要交互实体

成员	类型	说明
DDASRLinuxSDK::err_no	int ,SDK全局	SDK全局错误码，表示全局设置操作结果状态

int init_env(string log_path,string log_file,int log_level);	SDK全局函数	SDK日志初始化接口，每个进程只需要调用一次： log_path：日志目录 log_file：日志文件名称 log_level：日志等级
DDASRLinuxSDK *getInstance(int rate,int vad_on,int bv_on,string &err_msg);	SDK全局函数	获取SDK识别对象： rate：码率（RATE_8K,RATE_16K） vad_on:0表示关闭VAD，此时支持长语音文件（>60s）识别 1表示打开VAD，对实时语音流进行识别（短语音片段） vad_on:表示是否使用BV压缩 err_msg:错误信息提示 返回：成功返回SDK对象，识别时返回NULL，并可以通过 DDASRLinuxSDK::err_no 获取错误码 备注：当前测试环境允许最大并发：5
void clean_resource(bool force = false);	SDK全局函数	SDK全局清理资源函数，只需要在进程退出时调用一次
void set_event_listener(event_callback cb,void*arg);	SDK对象函数	设置每个SDK对象实例的回调函数： cb：回调函数 arg：用户自定义的参数，建议传递char*，arg会被当作回调函数的第二个参数使用
string get_sdk_version();	SDK全局函数	获取SDK的版本，
int post(DDASRMessage &msg,string &err_msg);	SDK对象函数	和SDK交互接口

调用流程

初始化环境

init_env（“./”,"asr.log",NOTICE）；

获取实例

```
std::string err_msg;
int vad_on = 0;
int bv_on = 0;
dd::DDASRLinuxSDK* sdk = dd::DDASRLinuxSDK::get_instance(RATE_8K,vad_on,bv_on, err_msg);
```

设置回调监听器

```
void asr_output_callback(DD::DDASRMessage& message, void* user_arg);
sdk->set_event_listener(&asr_output_callback, (void*)& thread_seq);
//  thread_sequser_argasr_output_callback
```

传递音频数据

```
dd::DDASRMessage push_params;
push_params.name = dd::DD_ASR_CMD_PUSH_AUDIO;
push_params.set_parameter(dd::DD_DATA_CHUNK, audio_buf, (int)read_cnt); // read_cnt=1280
....
int ret =sdk->post(push_params, err_msg);
```

停止音频流输入

告诉SDK 音频流已经输入完毕，不再有后续音频。 需要调用以下代码：

```
push_params.set_parameter(DD::DD_DATA_CHUNK, audio_buf, 0);

DD::DDASRMessage stop_params;
stop_params.name = DD::DD_ASR_CMD_STOP;
int ret = sdk->post(stop_params, err_msg));
```

释放资源

```
DDASRLinuxSDK::clean_resource();
```

代码示范

```
#include "DDASRLinuxSDK.h"
#include "DDASRMessage.h"
#include "string"
#include "stdio.h"
using namespace dd;
using namespace std;
#define MAX_WAV_SIZE 1024*1024*1024
int read_mch_wav( const char *mic_file, int *mic_len,char ** mData )
{
    cout << "read_mch_wav mic_file: " << mic_file << "  mic_len: " << mic_len <<  "  mData: " << mData
<< endl;
    FILE *fp_mic = fopen( mic_file, "rb" );
    if ( fp_mic == NULL )
    {
        printf( "Failed to open mch wav file[%s].\n", mic_file );
        return -1;
    }
    fseek( fp_mic, 0, SEEK_END );
    *mic_len = ftell( fp_mic );
    if ( *mic_len > MAX_WAV_SIZE )
    {
        printf( "mch_data overflows.\n" );
        return -1;
    }
    fseek( fp_mic, 0, SEEK_SET );
    fread( mData[0], sizeof(char), *mic_len, fp_mic );
    fclose( fp_mic );
    return 0;
}
//callbcak定义
void asr_output_callback(ASRResultM& message, void* user_arg){
 string asr ;
 message.get_asr_content(asr);
 //std::cout<<"recv asr content " << asr << "," <<message.get_status()<< endl;
```

```
 if( message.get_status() == DD_ASR_MIDDLE){
  //中间结果
  std::cout<<"中间结果asr:"<<asr<<","<<string((char*)user_arg)<<endl;
 }
 if( message.get_status() == DD_ASR_FINISH){
  //最后的一个结果
  std::cout<<"最后结果asr:"<<asr<<","<<string((char*)user_arg)<<endl;
 }
 if( message.get_status() == DD_ASR_ERR ){
  //报错
  //std::cout<<"未识别asr:"<<asr<<","<<string((char*)user_arg)<<endl;;
 }
}
int test(int,char*,int);
int main(){
 //初始化SDK日志环境，只需要配置一次
 dd::DDASRLinuxSDK::init_env("./","log.txt",NOTICE);
 std::cout<<"ver:"<<dd::DDASRLinuxSDK::get_sdk_version()<<endl;
 char * data = new char[MAX_WAV_SIZE];
        int len = 0;
        if( data == NULL ){
                cout << "mem error" <<endl;
                return -1;
        }
        //从文件读取PCM数据
        read_mch_wav("quanshi_meeting.pcm",&len,&data);
 for(int j = 30; j >= 0 ; j--){
  for ( int i= 0 ;i < 1;i++)
   test(0,data,len);
 sleep(5);
 }
 getchar();
 //程序结束的时候，清理SDK占有的资源
 dd::DDASRLinuxSDK::clean_resource();
 return 0;
}
int test (int idx,char *data,int len){
 string err_msg;
 int ret = 0;
 if( len <= 0 || data == NULL ){
  return 0;
 }
 //获取ASR SDK的识别的对象指针，支持 8k / 16k
 dd::DDASRLinuxSDK *sdk = DDASRLinuxSDK::getInstance(RATE_16K,err_msg);
 if(sdk == NULL){
  cout<<"sdk inst "<<idx<<" error:"<<err_msg<<endl;
  return -1;
 }
 int index = 0;
 int round = len/PACK_LEN ;
 char thread_seq[20]={0};
 sprintf(thread_seq,"idx:%d",idx);
 //设置识别结果的异步回调函数,当有识别结果时，回调函数会被不断触发
 sdk->set_event_listener(&asr_output_callback, (void*)thread_seq);
 cout<<"round:"<< (round)+(len/PACK_LEN==0?0:1) <<endl;
 //传输录音的内容，开始识别语音,当语音过长时，可以多次传输
 for(int i =0; i < round ;i++){
  dd::DDASRMessage push_params;
  push_params.name = dd::DD_ASR_CMD_PUSH_AUDIO;
```

```
  push_params.set_parameter(dd::DD_DATA_CHUNK, data+i*PACK_LEN, PACK_LEN);
  ret = sdk->post(push_params, err_msg);
  // VAD_END 表示自动检测到说话结束，此时会自动结束当前的识别
  // 重新开启一个会话，继续识别后续语音
  if( ret == VAD_END ){
   std::cout<<"detect speech end,continue" <<std::endl;
   test(idx+1,data+(i+1)*PACK_LEN,len-((i+1)*PACK_LEN));
   return 0;
  }else if( ret !=SUCCESS ){
   std::cout<<" post chunk failed"<< err_msg<<std::endl;
   return -1;
  }
 }
 if( len%PACK_LEN != 0 ){
  dd::DDASRMessage push_params;
                push_params.name = dd::DD_ASR_CMD_PUSH_AUDIO;
                push_params.set_parameter(dd::DD_DATA_CHUNK, data+(len/PACK_LEN)*PACK_LEN,
len%PACK_LEN);
  ret = sdk->post(push_params, err_msg);
  if( ret == VAD_END ){
                        std::cout<<"detect speech end" <<std::endl;
                }else if( ret != SUCCESS ){
                        std::cout<<" post chunk failed"<< err_msg<<std::endl;
                        return -1;
                }
 }
 //传输录音内容结束,手动结束录音传输
 dd::DDASRMessage push_params;
 push_params.name = DD_ASR_CMD_STOP;//dd::DD_ASR_CMD_PUSH_AUDIO;
 push_params.set_parameter(DD_DATA_CHUNK, NULL, 0);
 ret = sdk->post(push_params, err_msg);
 if( ret !=0 ){
                        std::cout<<" post chunk failed"<< err_msg<<std::endl;
                        return -1;
```

```
  }
  return 0;
}
```