ALIBABA CLOUD

Alibaba Cloud

智能语音交互

实时语音识别

文档版本：20220516

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.C++ SDK（新） 05

1.C++ SDK（新）

本文介绍了如何使用阿里云智能语音服务提供的C++ SDK，包括SDK的安装方法及SDK代码示例。

> **说明**
>
> - 当前最新版本：3.0.8。发布日期：2020年1月9日。
> 该版本只支持Linux平台，暂不支持Windows平台。
> - 请先阅读接口说明，详情请参见接口说明。
> - 该版本C++ SDK API和上一版本API（已下线）定义有区别，本文以当前版本为例进行介绍。

下载安装

SDK下载：

下载CppSdk，文件包含如下几部分：

- CMakeLists.txt：示例代码工程的CMakeList文件。
- readme.txt：SDK说明。
- release.log：版本说明。
- version：版本号。
- build.sh：demo编译脚本。
- lib：SDK库文件。
- build：编译目录。
- demo：示例，语音服务配置文件，如下表所示。

文件名	描述
speechRecognizerDemo.cpp	一句话识别示例
speechSynthesizerDemo.cpp	语音合成示例
speechTranscriberDemo.cpp	实时语音识别示例
speechLongSynthesizerDemo.cpp	长文本语音合成示例
test0.wav/test1.wav	测试音频（16k采样频率、16bit采样位数的音频文件

- include：SDK头文件，如下表所示。

文件名	描述
nlsClient.h	SDK实例
nlsEvent.h	回调事件说明
speechRecognizerRequest.h	一句话识别
speechSynthesizerRequest.h	语音合成/长文本语音合成
speechTranscriberRequest.h	实时音频流识别

编译运行：

1. 安装工具的最低版本要求如下：
   i. Cmake 3.1
   ii. Glibc 2.5
   iii. Gcc 4.1.2
2. 在Linux终端运行如下脚本。

```
mkdir build
cd build && cmake .. && make
cd ../demo  #生成示例可执行程序：srDemo（一句话识别）、stDemo（实时语音识别）、syDemo（语音合成）、syLongDemo（长文本语音合成）。
./stDemo appkey  <yourAccessKey Id>  <yourAccessKey Secret>   #测试使用。
```

关键接口

- 基础接口
  - NlsClient：语音处理客户端，利用该客户端可以进行一句话识别、实时语音识别和语音合成的语音处理任务。该客户端为线程安全，建议全局仅创建一个实例。
  - NlsEvent：事件对象，您可以从中获取Request状态码、云端返回结果、失败信息等。
- 识别接口

  SpeechTranscriberRequest：实时语音识别请求对象，用于长语音实时识别。

C++ SDK错误码

错误码	错误描述	解决方案

错误码	错误描述	解决方案
10000001	`SSL: couldn’t create a ……!`	建议重试。
10000002	openssl官方错误描述	根据描述提示处理之后，重试。
10000003	系统错误描述	根据系统错误描述提示处理。
10000004	`URL: The url is empty.`	检查是否设置云端URL地址。
10000005	`URL: Could not parse WebSocket url.`	检查是否正确设置云端URL地址。
10000006	`MODE: unsupport mode.`	检查是否正确设置了语音功能模式。
10000007	`JSON: Json parse failed.`	服务端发送错误响应内容，请提供task_id，并反馈至阿里云。
10000008	`WEBSOCKET: unkown head type.`	服务端发送错误WebSocket类型，请提供task_id，并通过工单系统反馈给我们。
10000009	`HTTP: connect failed.`	与云端连接失败，请检查网络后，重试。
HTTP协议官方状态码	`HTTP: Got bad status.`	根据HTTP协议官方描述提示处理。
系统错误码	`IP: ip address is not valid.`	根据系统错误描述提示处理。
系统错误码	`ENCODE: convert to utf8 error.`	根据系统错误描述提示处理。

错误码	错误描述	解决方案
10000010	`please check if the memory is enough.`	内存不足，请检查本地机器内存。
10000011	`Please check the order of execution.`	接口调用顺序错误（接收到Failed/Complete事件时，SDK内部会关闭连接。此时再调用send会上报错误。）。
10000012	`StartCommand/StopCommand Send failed.`	参数错误。请检查参数设置是否正确。
10000013	`The sent data is null or dataSize <= 0.`	发送错误。请检查发送参数是否正确。
10000014	`Start invoke failed.`	调用start方法超时错误。请调用stop方法释放资源，重新开始识别流程。
10000015	`connect failed.`	调用connect方法失败。请调用stop方法释放资源，重新开始识别流程。

服务端响应状态码

关于服务状态码，请参见接口说明。

代码示例

说明

- 示例中使用的音频文件为16000Hz采样率，管控台设置的模型为**通用**模型。如果使用其他音频，请设置为支持该音频场景的模型。关于模型设置，请参见管理项目。
- 完整示例参见SDK压缩包中Demo目录的speechTranscriberDemo.cpp文件。

```
#include <pthread.h>
#include <unistd.h>
#include <ctime>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
#include <fstream>
#include "nlsClient.h"
```

```
#include "nlsEvent.h"
#include "speechTranscriberRequest.h"
#include "nlsCommonSdk/Token.h"
#define FRAME_SIZE 3200
#define SAMPLE_RATE 16000
using namespace AlibabaNlsCommon;
using AlibabaNls::NlsClient;
using AlibabaNls::NlsEvent;
using AlibabaNls::LogDebug;
using AlibabaNls::LogInfo;
using AlibabaNls::SpeechTranscriberRequest;
// 自定义线程参数。
struct ParamStruct {
    std::string fileName;
    std::string token;
    std::string appkey;
};
// 自定义事件回调参数。
struct ParamCallBack {
    int userId;
    char userInfo[10];
};
//全局维护一个服务鉴权token和其对应的有效期时间戳，
//每次调用服务之前，首先判断token是否已经过期。
//如果已经过期，则根据AccessKey ID和AccessKey Secret重新生成一个token，并更新这个全局的token和其有效期时间戳。
//说明：只需在token即将过期时进行重新生成。所有的服务并发可共用一个token。
std::string g_akId = "";
std::string g_akSecret = "";
std::string g_token = "";
long g_expireTime = -1;
int generateToken(std::string akId, std::string akSecret, std::string* token, long* expireTime) {
    NlsToken nlsTokenRequest;
    nlsTokenRequest.setAccessKeyId(akId);
    nlsTokenRequest.setKeySecret(akSecret);
    if (-1 == nlsTokenRequest.applyNlsToken()) {
        // 获取失败原因。
        printf("generateToken Failed: %s\n", nlsTokenRequest.getErrorMsg());
        return -1;
    }
    *token = nlsTokenRequest.getToken();
    *expireTime = nlsTokenRequest.getExpireTime();
    return 0;
}
//@brief 获取sendAudio发送延时时间
//@param dataSize 待发送数据大小
//@param sampleRate 采样率：16k/8K
//@param compressRate 数据压缩率，例如压缩比为10:1的16k OPUS编码，此时为10，非压缩数据则为1。
//@return 返回sendAudio之后需要sleep的时间。
//@note 对于8k PCM编码数据，16位采样，建议每发送1600字节sleep 100ms。
// 对于16k PCM编码数据，16位采样，建议每发送3200字节sleep 100ms。
// 对于其它编码格式的数据，您可以根据压缩比自行估算。比如，压缩比为10:1的16k OPUS编码数据，
// 需要每发送3200/10=320 sleep 100ms。
```

```
unsigned int getSendAudioSleepTime(int dataSize, int sampleRate, int compressRate) {
    // 仅支持16位采样
    const int sampleBytes = 16;
    // 仅支持单通道
    const int soundChannel = 1;
    // 当前采样率，采样位数下每秒采样数据的大小。
    int bytes = (sampleRate * sampleBytes * soundChannel) / 8;
    // 当前采样率，采样位数下每毫秒采样数据的大小。
    int bytesMs = bytes / 1000;
    // 待发送数据大小除以每毫秒采样数据大小，以获取sleep时间。
    int sleepMs = (dataSize * compressRate) / bytesMs;
    return sleepMs;
}
//@brief 调用start()，成功与云端建立连接，SDK内部线程上报started事件。
//@param cbEvent 回调事件结构，详见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL。可以根据需求自定义参数。
void onTranscriptionStarted(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    // 演示如何打印/使用用户自定义参数示例。
    printf("onTranscriptionStarted: %d\n", tmpParam->userId);
    // 当前任务的task id，方便定位问题。
    printf("onTranscriptionStarted: status code=%d, task id=%s\n", cbEvent->getStatusCode()
, cbEvent->getTaskId());
    // 获取服务端返回的全部信息
    //printf("onTranscriptionStarted: all response=%s\n", cbEvent->getAllResponse());
}
//@brief 服务端检测到了一句话的开始，SDK内部线程上报SentenceBegin事件。
//@param cbEvent 回调事件结构，详情参见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL，可以根据需求自定义参数。
void onSentenceBegin(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    // 演示如何打印/使用用户自定义参数示例。
    printf("onSentenceBegin: %d\n", tmpParam->userId);
    printf("onSentenceBegin: status code=%d, task id=%s, index=%d, time=%d\n", cbEvent->get
StatusCode(), cbEvent->getTaskId(),
                cbEvent->getSentenceIndex(), //句子编号，从1开始递增。
                cbEvent->getSentenceTime() //当前已处理的音频时长，单位：毫秒。
                );
    // 获取服务端返回的全部信息
    //printf("onTranscriptionStarted: all response=%s\n", cbEvent->getAllResponse());
}
//@brief 服务端检测到了一句话结束，SDK内部线程上报SentenceEnd事件。
//@param cbEvent 回调事件结构，详情参见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL，可以根据需求自定义参数。
void onSentenceEnd(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    // 演示如何打印/使用用户自定义参数示例。
    printf("onSentenceEnd: %d\n", tmpParam->userId);
    printf("onSentenceEnd: status code=%d, task id=%s, index=%d, time=%d, begin_time=%d, re
sult=%s\n", cbEvent->getStatusCode(), cbEvent->getTaskId(),
                cbEvent->getSentenceIndex(), //句子编号，从1开始递增。
                cbEvent->getSentenceTime(), //当前已处理的音频时长，单位：毫秒。
                cbEvent->getSentenceBeginTime(), // 对应的SentenceBegin事件的时间。
                cbEvent->getResult()    // 当前句子的完整识别结果。
```

```
                );
    // 获取服务端返回的全部信息
    //printf("onTranscriptionStarted: all response=%s\n", cbEvent->getAllResponse());
}
//@brief 识别结果发生了变化，SDK在接收到云端返回的最新结果时，其内部线程上报ResultChanged事件。
//@param cbEvent 回调事件结构，详情参见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL，可以根据需求自定义参数。
void onTranscriptionResultChanged(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    // 演示如何打印/使用用户自定义参数示例。
    printf("onTranscriptionResultChanged: %d\n", tmpParam->userId);
    printf("onTranscriptionResultChanged: status code=%d, task id=%s, index=%d, time=%d, re
sult=%s\n", cbEvent->getStatusCode(), cbEvent->getTaskId(),
                cbEvent->getSentenceIndex(), //句子编号，从1开始递增。
                cbEvent->getSentenceTime(), //当前已处理的音频时长，单位：毫秒。
                cbEvent->getResult()     // 当前句子的完整识别结果
                );
    // 获取服务端返回的全部信息
    //printf("onTranscriptionStarted: all response=%s\n", cbEvent->getAllResponse());
}
//@brief 服务端停止实时音频流识别时，SDK内部线程上报Completed事件。
//@note 上报Completed事件之后，SDK内部会关闭识别连接通道。此时调用sendAudio会返回-1，请停止发送。
//@param cbEvent 回调事件结构，详情参见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL，可以根据需求自定义参数。
void onTranscriptionCompleted(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    // 演示如何打印/使用用户自定义参数示例。
    printf("onTranscriptionCompleted: %d\n", tmpParam->userId);
    printf("onTranscriptionCompleted: status code=%d, task id=%s\n", cbEvent->getStatusCode
(), cbEvent->getTaskId());
}
//@brief 识别过程（包含start()、send()、stop()）发生异常时，SDK内部线程上报TaskFailed事件。
//@note 上报TaskFailed事件之后，SDK内部会关闭识别连接通道。此时调用sendAudio会返回-1，请停止发送。
//@param cbEvent 回调事件结构，详情参见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL，可以根据需求自定义参数。
void onTaskFailed(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    // 演示如何打印/使用用户自定义参数示例。
    printf("onTaskFailed: %d\n", tmpParam->userId);
    printf("onTaskFailed: status code=%d, task id=%s, error message=%s\n", cbEvent->getStat
usCode(), cbEvent->getTaskId(), cbEvent->getErrorMessage());
    // 获取服务端返回的全部信息
    //printf("onTaskFailed: all response=%s\n", cbEvent->getAllResponse());
}
//@brief SDK内部线程上报语音表单结果事件
//@param cbEvent 回调事件结构，详情参见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL，可以根据需求自定义参数。
void onSentenceSemantics(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    // 演示如何打印/使用用户自定义参数示例。
    printf("onSentenceSemantics: %d\n", tmpParam->userId);
    // 获取服务端返回的全部信息。
    printf("onSentenceSemantics: all response=%s\n", cbEvent->getAllResponse());
}
```

```
//@brief 识别结束或发生异常时，会关闭连接通道，SDK内部线程上报ChannelCloseed事件。
//@param cbEvent 回调事件结构，详情参见nlsEvent.h。
//@param cbParam 回调自定义参数，默认为NULL，可以根据需求自定义参数。
void onChannelClosed(NlsEvent* cbEvent, void* cbParam) {
    ParamCallBack* tmpParam = (ParamCallBack*)cbParam;
    delete tmpParam; //识别流程结束，释放回调参数。
}
// 工作线程
void* pthreadFunc(void* arg) {
    int sleepMs = 0;
    ParamCallBack *cbParam = NULL;
    //初始化自定义回调参数，以下两变量仅作为示例表示参数传递，在demo中不起任何作用。
    //回调参数在堆中分配之后，SDK在销毁request对象时会一并销毁。
    cbParam = new ParamCallBack;
    cbParam->userId = 1234;
    strcpy(cbParam->userInfo, "User.");
    // 0: 从自定义线程参数中获取token，配置文件等参数。
    ParamStruct* tst = (ParamStruct*)arg;
    if (tst == NULL) {
        printf("arg is not valid\n");
        return NULL;
    }
    /* 打开音频文件，获取数据 */
    std::ifstream fs;
    fs.open(tst->fileName.c_str(), std::ios::binary | std::ios::in);
    if (!fs) {
        printf("%s isn't exist..\n", tst->fileName.c_str());
        return NULL;
    }
    //2: 创建实时音频流识别SpeechTranscriberRequest对象。
    SpeechTranscriberRequest* request = NlsClient::getInstance()->createTranscriberRequest(
);
    if (request == NULL) {
        printf("createTranscriberRequest failed.\n");
        return NULL;
    }
    request->setOnTranscriptionStarted(onTranscriptionStarted, cbParam);                //
设置识别启动回调函数
    request->setOnTranscriptionResultChanged(onTranscriptionResultChanged, cbParam);    //
设置识别结果变化回调函数
    request->setOnTranscriptionCompleted(onTranscriptionCompleted, cbParam);            //
设置语音转写结束回调函数
    request->setOnSentenceBegin(onSentenceBegin, cbParam);                              //
设置一句话开始回调函数
    request->setOnSentenceEnd(onSentenceEnd, cbParam);                                  //
设置一句话结束回调函数
    request->setOnTaskFailed(onTaskFailed, cbParam);                                    //
设置异常识别回调函数
    request->setOnChannelClosed(onChannelClosed, cbParam);                              //
设置识别通道关闭回调函数
    request->setAppKey(tst->appkey.c_str());            // 设置appkey，必选参数。
    request->setFormat("pcm");                          // 设置音频数据编码格式，默认值PCM。
    request->setSampleRate(SAMPLE_RATE);                // 设置音频数据采样率，可选参数，目前支
持16000/8000，默认值16000。
    request->setIntermediateResult(true);               // 设置是否返回中间识别结果，可选参数
```

```
    request->setIntermediateResult(true);                    // 设置是否返回中间识别结果，可选参数，
默认false。
    request->setPunctuationPrediction(true);                 // 设置是否在后处理中添加标点，可选参数
，默认false。
    request->setInverseTextNormalization(true);              // 设置是否在后处理中执行数字转写，可选
参数，默认false。
    //语音断句检测阈值，一句话之后静音长度超过该值，即本句结束，合法参数范围200ms～2000ms，默认值800m
s。
    //request->setMaxSentenceSilence(800);
    //request->setCustomizationId("TestId_123"); //定制模型id，可选。
    //request->setVocabularyId("TestId_456"); //定制泛热词id，可选。
    // 用于传递某些定制化、高级参数设置，参数格式为JSON格式： {"key": "value"}
    //request->setPayloadParam("{\"vad_model\": \"farfield\"}");
    //设置是否开启词模式。
    request->setPayloadParam("{\"enable_words\": true}");
    //语义断句，默认false，非必需则不建议设置。
    //request->setPayloadParam("{\"enable_semantic_sentence_detection\": false}");
    //是否开启顺滑，默认不开启，非必需则不建议设置。
    //request->setPayloadParam("{\"disfluency\": true}");
    //设置vad的模型，默认不设置，非必需则不建议设置。
    //request->setPayloadParam("{\"vad_model\": \"farfield\"}");
    //设置是否忽略单句超时
    //request->setPayloadParam("{\"enable_ignore_sentence_timeout\": false}");
    //vad断句开启后处理，默认不设置，非必需则不建议设置。
    //request->setPayloadParam("{\"enable_vad_unify_post\": true}");
    request->setToken(tst->token.c_str());
    //3: start()为异步操作。成功返回started事件，失败返回TaskFailed事件。
    if (request->start() < 0) {
        printf("start() failed. may be can not connect server. please check network or fire
walld\n");
        NlsClient::getInstance()->releaseTranscriberRequest(request); // start()失败，释放re
quest对象。
        return NULL;
    }
    while (!fs.eof()) {
        uint8_t data[FRAME_SIZE] = {0};
        fs.read((char *)data, sizeof(uint8_t) * FRAME_SIZE);
        size_t nlen = fs.gcount();
        if (nlen <= 0) {
            continue;
        }
        //4: 发送音频数据。sendAudio返回-1表示发送失败，需要停止发送。
        int ret = request->sendAudio(data, nlen);
        if (ret < 0) {
            // 发送失败，退出循环数据发送。
            printf("send data fail.\n");
            break;
        }
        //语音数据发送控制：
        //语音数据是实时的，不需要sleep控制速率。
        //语音数据来自文件，发送时需要控制速率，使单位时间内发送的数据大小接近单位时间原始语音数据存储
的大小。
        sleepMs = getSendAudioSleepTime(nlen, SAMPLE_RATE, 1); // 根据发送数据大小、采样率、数
据压缩比，获取sleep时间。
        //5: 语音数据发送延时控制
```

```
        //5: 语音数据发送速率控制
        usleep(sleepMs * 1000);
    }
    // 关闭音频文件
    fs.close();
    //6: 通知云端数据发送结束
    //stop()为异步操作，失败返回TaskFailed事件。
    request->stop();
    //7: 识别结束，释放request对象。
    NlsClient::getInstance()->releaseTranscriberRequest(request);
    return NULL;
}
//识别单个音频数据
int speechTranscriberFile(const char* appkey) {
    // 获取当前系统时间戳，判断token是否过期。
    std::time_t curTime = std::time(0);
    if (g_expireTime - curTime < 10) {
        printf("the token will be expired, please generate new token by AccessKey-ID and Ac
cessKey-Secret.\n");
        if (-1 == generateToken(g_akId, g_akSecret, &g_token, &g_expireTime)) {
            return -1;
        }
    }
    ParamStruct pa;
    pa.token = g_token;
    pa.appkey = appkey;
    pa.fileName = "test0.wav";
    pthread_t pthreadId;
    // 启动一个工作线程，用于识别。
    pthread_create(&pthreadId, NULL, &pthreadFunc, (void *)&pa);
    pthread_join(pthreadId, NULL);
    return 0;
}
//识别多个音频数据
//SDK多线程指一个音频数据对应一个线程，多个音频数据对应多个线程。
//示例代码为同时开启2个线程识别2个文件。
//免费用户并发连接不能超过2个。
#define AUDIO_FILE_NUMS 2
#define AUDIO_FILE_NAME_LENGTH 32
int speechTranscriberMultFile(const char* appkey) {
    // 获取当前系统时间戳，判断token是否过期。
    std::time_t curTime = std::time(0);
    if (g_expireTime - curTime < 10) {
        printf("the token will be expired, please generate new token by AccessKey-ID and Ac
cessKey-Secret.\n");
        if (-1 == generateToken(g_akId, g_akSecret, &g_token, &g_expireTime)) {
            return -1;
        }
    }
    char audioFileNames[AUDIO_FILE_NUMS][AUDIO_FILE_NAME_LENGTH] = {"test0.wav", "test1.wav
"};
    ParamStruct pa[AUDIO_FILE_NUMS];
    for (int i = 0; i < AUDIO_FILE_NUMS; i ++) {
        pa[i].token = g_token;
        pa[i].appkey = appkey;
```

```
        pa[i].fileName = audioFileNames[i];
    }
    std::vector<pthread_t> pthreadId(AUDIO_FILE_NUMS);
    // 启动2个工作线程，同时识别2个音频文件。
    for (int j = 0; j < AUDIO_FILE_NUMS; j++) {
        pthread_create(&pthreadId[j], NULL, &pthreadFunc, (void *)&(pa[j]));
    }
    for (int j = 0; j < AUDIO_FILE_NUMS; j++) {
        pthread_join(pthreadId[j], NULL);
    }
    return 0;
}
int main(int arc, char* argv[]) {
    if (arc < 4) {
        printf("params is not valid. Usage: ./demo <your appkey> <your AccessKey ID> <your
AccessKey Secret>\n");
        return -1;
    }
    std::string appkey = argv[1];
    g_akId = argv[2];
    g_akSecret = argv[3];
    // 根据需要设置SDK输出日志，可选。此处表示SDK日志输出至log-Transcriber.txt，LogDebug表示输出所
有级别日志。
    int ret = NlsClient::getInstance()->setLogConfig("log-transcriber", LogDebug);
    if (-1 == ret) {
        printf("set log failed\n");
        return -1;
    }
    //启动工作线程
    NlsClient::getInstance()->startWorkThread(4);
    // 识别单个音频数据
    speechTranscriberFile(appkey.c_str());
    // 识别多个音频数据
    // speechTranscriberMultFile(appkey.c_str());
    // 所有工作完成，进程退出前释放nlsClient，releaseInstance()非线程安全。
    NlsClient::releaseInstance();
    return 0;
}
```