智能语音交互

语音识别(ASR)

语音识别(ASR)

语音识别服务，可提供语音转文本服务，包括：一句话识别、实时语音识别、录音文件识别。

- 一句话识别：即实时短语音识别，可提供Java、Android、iOS SDK，并提供python的批量识别工具。
- 实时语音识别：即实时长语音识别，可支持长时间语音识别。可提供Java SDK。
- 录音文件识别：可提供Restful API接口，支持录音文件的语音识别。

阿里云语音服务为用户提供语音识别的基础服务，Android、iOS SDK封装了录音(Recoder)、静音检测(VAD)、语音服务访问(WSAPI)等功能，可以极大的简化App开发。

阿里云语音识别技术是基于后台服务器的密集CPU计算，语音SDK负责在App端打开录音机，进行语音压缩后，传送到服务器端。服务器进行语音识别转成文字后，通常还需要进行自然语言处理，分析其语法结构，并把语意结果返回给SDK。

一句话识别

一句话识别：即实时短语音识别，可提供Java、Android、iOS SDK，并提供python的批量识别工具。

支持的app_key

一句话识别 app_key	语音数据格式	结果返回方式	领域
nls-service	16kHz采样 16bit 音频流	非流式	社交聊天
nls-service-streaming	16kHz采样 16bit 音频流	流式	社交聊天
nls-service-tv	16kHz采样 16bit 音频流	非流式	家庭娱乐
nls-service-shopping	16kHz采样 16bit 音频流	非流式	电商购物领域
nls-service-care	16kHz采样 16bit 音频流	非流式	智能客服服务领域

注：

(1)“支持的结果返回方式”式包括“流式”和“非流式”两种模式，“流式”模式下用户一边说话一边返回识别结果，“非流式”简单来说就是用户整句话说完后返回识别结果。

(2)“一句话识别”支持的领域包括：社交聊天、家庭娱乐、电商购物、智能客服等。用户可针对具体的使用场景选择对应领域的app_key。

功能介绍

语音Java SDK提供一句话识别服务，提供将实时短语音转成文字的功能，可直接用于语音搜索类应用。

SDK下载地址

一句话识别JavaSDK

请注意：sdk内部不自带语音采集的功能，只提供将语音流与文字互转和语意识别的功能。

示例说明

下载地址中包含了Java SDK的jar包和demo工程，用户只需在工程中通过“Java Build Path”->“Add External JARs”将jar包导入，即可运行。

SDK调用顺序

创建一个NlsClient的实例并调用init()方法来初始化客户端

提取语音数据并创建语音识别请求，至少填写appKey及需要识别的语音数据的格式。创建一个NlsListener的实现类。

调用NlsClient的createNlsFuture（第2步中的listener实例作为入参之一，用来处理返回结果）方法获取future，通过future的sendVoice方法来发送语音数据并在listener中处理返回结果。

通过future的sendFinishSignal来结束语音文件的发送，ASR服务收到这个结束信号后，会返回处理结果。

如有多个识别需要，重复步骤2-4。

调用NlsClient的close()方法来关闭客户端并释放资源。

SDK调用注意事项

NlsClient创建一次可以重复使用,每次创建消耗性能

NlsClient使用了netty的框架，创建时比较消耗时间和资源，但创建之后可以重复利用。建议调用程序将NlsClient的创建和关闭与程序本身的生命周期结合。

每一次调用语音识别请求时注入的NlsListener只对本次语音识别的生命周期负责。

每次调用createNlsFuture(NlsRequest req, com.alibaba.idst.nls.event.NlsListener listener)方法做语音识别和对话时，注入的listener只对本次识别起作用。其他的语音识别进程不会触发该listener。

NlsRequest里面的语音格式请与语音文件的格式保持一致。目前SDK支持pcm和opu格式，opu格式请参考opus编解码文档进行压缩。

sdk只会根据该格式来切分和发送语音文件，如果两个不一致，后续的一切都是错误的。

NlsListener的实现类里面，处理返回结果的代码尽量耗时剪短，最好不要涉及I/O操作。

NlsListener的实现类中的处理方法是在NlsClient的语音识别线程中调用的，太长的操作时间导致线程不能及时释放会影响其他识别进程的顺利进行，同时也会影响整个程序的loading。I/O操作应该禁止在该实现类中出现，最好使用触发的方式将操作转移到其他线程中去进行。

并发或多线程支持，如果需要在您的应用支持多个并发请求，请不要重复创建NlsClient对象。正确的做法是构建不同的NlsRequest对象，同时创建不同的NlsListener，并传入NlsRequest对象。这样就可以并发不同请求并且拿到正确的相应的结果。

重要接口说明

com.alibaba.idst.nls.NlsClient

语音sdk对外暴露的类，调用程序通过调用该类的init()、close()、createNlsFuture()等方法来打开、关闭或发送语音数据。

初始化NlsClient

public void init((boolean sslMode, int port)

- 说明 初始化NlsClient，创建好websocket client factory

参数

- sslMode 是否采用ssl模式，一般设置为true
- port 端口号，一般为443

返回值 null

设置服务器地址

public void setHost(String host)

- 说明 设置服务器地址

参数

返回值 null

实例化NlsFuture请求

public NlsFuture createNlsFuture(NlsRequest req, com.alibaba.idst.nls.event.NlsListener listener)

- 说明 实例化NlsFuture请求，传入请求和监听器

参数

- req 请求对象
- listener 回调数据的监听器

返回值 future

关闭NlsClient

public void close()

- 说明 关闭websocket client factory，释放资源

参数

null

返回值 null

com.alibaba.idst.nls.event.NlsListener

语音识别是一个非常漫长的过程，为了不影响服务端的正常工作，语音sdk被设计成异步模式，调用程序将语音客户端建立好并将语音数据交给sdk后就可以离开。在收到语音数据之后的处理需要通过实现该接口来完成。每

一个识别过程创建的时候都需要注入侦听者以响应语音识别的结果。

识别结果回调

void onMessageReceived(NlsEvent e);

- 说明 识别结果回调

参数

- NlsEvent Nls服务结果的对象 识别结果在e.getResponse()中

返回值 null

识别失败回调

void onOperationFailed(NlsEvent e);

- 说明 识别失败回调

参数

- NlsEvent Nls服务结果的对象 错误信息在e.getErrorMessage()中

返回值 null

socket 连接关闭的回调

void onChannelClosed(NlsEvent e);

- 说明 socket 连接关闭的回调

参数

- NlsEvent Nls服务结果的对象

返回值 null

com.alibaba.idst.nls.protocol.NlsRequest

语音识别的请求封装对象。调用程序需要至少在请求对象里设定好调用者的appKey和语音数据的格式以便后台服务能正确识别并识别语音。

初始化Nls 请求：NlsRequest mNlsRequest = new NlsRequest(proto)

public void setApp_key(String app_key)

- 说明 设置应用的appkey，appkey需要先申请再使用。
- 参数

- app_key

- 返回值 null

public void setAsr_sc(String sc)

说明 设置语音识别的语音格式，一般为pcm。若使用opu格式的语音，请参考opus编解码文档说明进行编码设置。

该项在使用语音识别服务时必须设置，起到初始化作用。

- 参数
  - sc 一般为pcm。
- 返回值 null

public void authorize(String id, String secret)

- 说明 数加认证模块，所有的请求都必须通过authorize方法认证通过，才可以使用。 id和secret需要申请获取。
- 参数
  - id 数加平台申请的Access Key ID。
  - secret 对应密钥Access Key Secret。
- 返回值 null

com.alibaba.idst.nls.internal.protocol.NlsRequestProto

NlsRequest对象初始化时需要注入的参数。

public void setApp_id(String app_id)

- 设置application_id.

public void setApp_user_id(String app_user_id)

- 设置app_user_id.

com.alibaba.idst.nls.NlsFuture

NlsFuture是具体操作语音的对象类，语音数据的发送/接收都通过这个类进行操作。**public NlsFuture sendVoice(byte[] data, int offset, int length)**

- 说明 语音识别时，发送语音文件的方法
- 参数
  - data byte[]类型的语音流
  - offset
  - length
- 返回值 NlsFuture

public NlsFuture sendFinishSignal()

- 说明 语音识别结束时，发送结束符

参数

返回值 NlsFuture

public boolean await(int timeout)

- 说明 请求超时时间
- 参数
  - timeout 请求发送出去后，等待服务端结果返回的超时时间，单位ms
- 返回值 true false

com.alibaba.idst.nls.protocol.NlsResponse

语音识别的返回结果封装对象。返回结果告知语音识别是否成功，语音识别结果或部分结果(视调用程序需要的返回方式而定)，语音识别是否已经结束。

public String getAsr_ret()

- 说明 获取语音识别的结果
- 参数
  - null
- 返回值 String

错误码

状态	status_code	CloseFrame状态码	HTTP语义
成功	200	1000	成功处理
请求格式有误	400	4400	错误请求
需要鉴权信息	401	4401	请求要求身份验证
鉴权失败	403	4403	服务器拒绝请求
超出最大并发量	429	4429	太多请求
请求超时	408	4408	处理请求超时
处理出错	500	4500	服务器内部错误
服务不可用	503	4503	服务不可用

完整示例

```
package com.demo;
```

```
import java.io.File;
import java.io.FileInputStream;

import com.alibaba.idst.nls.NlsClient;
import com.alibaba.idst.nls.NlsFuture;
import com.alibaba.idst.nls.event.NlsEvent;
import com.alibaba.idst.nls.event.NlsListener;
import com.alibaba.idst.nls.protocol.NlsRequest;
import com.alibaba.idst.nls.protocol.NlsResponse;

public class AsrDemo implements NlsListener {
private static NlsClient client = new NlsClient();

public AsrDemo() {
System.out.println("init Nls client...");
// 初始化NlsClient
client.init();
}

public void shutDown() {
System.out.println("close NLS client");
// 关闭客户端并释放资源
client.close();
System.out.println("demo done");
}

public void startAsr() {
//开始发送语音
System.out.println("open audio file...");
FileInputStream fis = null;
try {
String filePath = "src/test/resources/sample.pcm";
File file = new File(filePath);
fis = new FileInputStream(file);
} catch (Exception e) {
e.printStackTrace();
}

if (fis != null) {
System.out.println("create NLS future");
try {
NlsRequest req = new NlsRequest();
req.setApp_key(""); // appkey列表中获取，本demo可同时支持流式和非流式appkey
req.setAsr_sc("pcm"); // 设置语音文件格式为pcm,我们支持16k 16bit 的无头的pcm文件。
req.authorize("", ""); // 请替换为用户申请到的Access Key ID和Access Key Secret
NlsFuture future = client.createNlsFuture(req, this); // 实例化请求,传入请求和监听器
System.out.println("call NLS service");
byte[] b = new byte[8000];
int len = 0;
while ((len = fis.read(b)) > 0) {
future.sendVoice(b, 0, len); // 发送语音数据
//使用文件测试，需要每8000byte sleep250ms。
//如果发送实时语音流，不需要进行sleep操作。
Thread.sleep(250);
}
```

```
future.sendFinishSignal(); // 语音识别结束时，发送结束符
System.out.println("main thread enter waiting for less than 10s.");
future.await(10000); // 设置服务端结果返回的超时时间
} catch (Exception e) {
e.printStackTrace();
}
System.out.println("calling NLS service end");
}
}

public void onMessageReceived(NlsEvent e) {
//识别结果的回调
NlsResponse response = e.getResponse();
String result = "";
int statusCode = response.getStatus_code();
if (response.getAsr_ret() != null) {
result += "\nget asr result: statusCode=[" + statusCode + "], " + response.getAsr_ret();
}
if (result != null) {
System.out.println(result);
} else {
System.out.println(response.jsonResults.toString());
}
}

@Override
public void onOperationFailed(NlsEvent e) {
//识别失败的回调
String result = "";
result += "on operation failed: statusCode=[" + e.getResponse().getStatus_code() + "], " + e.getErrorMessage();
System.out.println(result);
}

@Override
public void onChannelClosed(NlsEvent e) {
//socket 连接关闭的回调
System.out.println("on websocket closed.");
}

public static void main(String[] args) {
AsrDemo asrDemo = new AsrDemo();
asrDemo.startAsr();
asrDemo.shutDown();
}
}
```

功能介绍

语音Android SDK提供一句话识别服务，提供将实时短语音转成文字的功能，可直接用于语音搜索类应用。

阿里云语音服务SDK（NLSClinet），是运行于android平台的基础语音识别、自然语言理解和语音合成的基础服务，本服务通过Jar包和.so库的形式导入，用户通过本指南可以方便的接入。

SDK下载地址

一句话识别AndroidSDK

开发包目录

文件夹	内容
libs/	
libs/NlsClientSdk.jar	NLSClinet的服务包
libs/armeabi	
libs/armeabi/libjoysecurity.so	
libs/armeabi/libzts2.so	
libs/armeabi-v7a	
libs/armeabi-v7a/libjoysecurity.so	
libs/armeabi-v7a/libzts2.so	
libs/x86	
libs/x86/libjoysecurity.so	
libs/x86/libzts2.so	

- 本服务使用了解析json的gson.jar,需要用户自行导入。

集成指南

导入服务包

将下载的服务包解压后，将NlsClientSdk.jar，以及对应架构的.so包导入你的项目中的libs/目录下。

Android权限管理

在AndroidManifest.xml文件中添加以下权限申请：

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
```

Proguard配置

如果代码使用了混淆，请在proguard-rules.pro中配置：

```
-keep class com.alibaba.idst.nls.** {*;}
-keep class com.google.**{*;}
```

重要接口说明

com.alibaba.idst.nls.NlsClient

语音服务的核心服务类，客户端程序通过调用该类的cancel()、start()、stop()等方法来打开、关闭或发送语音数据。

全局配置

public static void configure(Context appContext)

- 说明 全局配置，在初始化NlsClient前需要被调用，建议放在Application的onCreate()里
- 参数
  - appContext 传入ApplicationContext参数，用于识别引擎内部访问和Android上下文相关的资源
- 返回值 null

实例化NlsClient对象

public static NlsClient newInstance(Context context,NlsListener nlsListener,StageListener stageListener, NlsRequest nlsRequest)

- 说明 获得一个NlsClient, 通过该方法实例化NlsClient。
- 参数
  - context 传入Context参数，用于识别引擎内部访问和Android上下文相关的资源
  - nlsListener 识别相关的回调接口，用于通知客户端识别结果
  - stageListener 引擎状态回调接口，用于通知客户端当前的引擎状态以及录音音量等
- 返回值 NlsClient

打开语音识别引擎

public boolean start()

- 说明 打开语音识别引擎

  参数

  返回值 isOpen true:打印log false:关闭log

true:开始录音，并对录音进行语音识别 false:打开语音识别引擎失败，可能重复打开或者远程服务处于不可用状态。

关闭语音识别引擎

public void stop()

- 说明 关闭语音识别引擎

参数

返回值

判断是否正在进行语音识别

public boolean isStarted()

- 说明 返回当前引擎是否已启动

参数

返回值 true:引擎已启动 false:引擎未启动

取消此次语音识别

public void cancel()

- 说明 取消此次语音识别，识别结果为ErrorCode.USER_CANCEL

参数

返回值 null

其它接口：

Log开关

public static void openLog(boolean isOpen)

- 说明 log开关，标识是否需要通过logcat打印识别引擎的相关log
- 参数
    - isOpen true:打印log false:关闭log
- 返回值 null

设置VAD是否打开

public NlsClient setRecordAutoStop(boolean isAutoStop)

- 说明 VAD 是端点检测功能，打开VAD可以自动检测语音结束，并结束语音
- 参数
    - isAutoStop true:打开VAD false:关闭VAD
- 返回值 NlsClient

设置音量回调时长

public NlsClient setMinVoiceValueInterval(int interval)

- 说明 设置获取录音音量的最短时间间隔，防止录音音量回调过于频繁影响客户端对音量的展示逻辑
- 参数
    - interval 更新间隔 单位ms
- 返回值 NlsClient

设置最短录音时长

public NlsClient setMinRecordTime(int minRecordTime)

- 说明 用于设置最短录音时间，引擎开始时，用户在该时间内不说话，则会自动关闭引擎，识别结果为ErrorCode.NOTHING
- 参数
    - minRecordTime 最短录音时间 单位ms
- 返回值 NlsClient

设置最大录音时长

public NlsClient setMaxRecordTime(int maxRecordTime)

- 说明 用于设置最大录音时间
- 参数
    - maxRecordTime 最大录音时间 单位ms
- 返回值 NlsClient

设置最大录音中断时间

public NlsClient setMaxStallTime(int milliSeconds)

- 说明 打开VAD时， 设置录音中句子之间的最长停顿时间 ，录音过程中，如果用户停顿超过该时间则认为用户已经停止说话，停止录音
- 参数
    - milliSeconds 最大录音时间 单位ms
- 返回值 NlsClient

获取完整录音语音

public byte[] getObject()

- 说明
  - 得到录音样本PCM，该方法可以在mStageListener.onStopRecognizing()方法中调用。
- 参数
  - pcm bytes

com.alibaba.idst.nls.internal.protocol.NlsRequest

语音服务的请求封装对象。调用程序需要至少在请求对象里设定好调用者的appKey和语音数据的格式以便后台服务能正确识别并翻译语音。

初始化Nls 请求：NlsRequest mNlsRequest = new NlsRequest(proto)或缺省proto：NlsRequest mNlsRequest = new NlsRequest()

public void setApp_key(String app_key)

- 说明 设置应用的appkey，appkey请从 “快速开始” 帮助页面的appkey列表中获取。
- 参数
  - app_key
- 返回值 null

public void setAsr_sc(String sc)

- 说明 设置语音识别的语音格式，默认为opu。 该项在使用语音识别服务时必须设置，起到初始化作用。
- 参数
  - sc 设置为opu。
- 返回值 null

public void authorize(String id, String secret)

- 说明 数加认证模块，所有的请求都必须通过authorize方法认证通过，才可以使用。 id和secret需要申请获取。
- 参数
  - id id。
  - secret 对应密钥。
- 返回值 null

com.alibaba.idst.nls.internal.protocol.NlsRequestProto

NlsRequest对象初始化时需要注入的参数。

public void setApp_id(String app_id)

- 设置application_id.

public void setApp_user_id(String app_user_id)

- 设置app_user_id.

public void setReq_id(String req_id)

- 设置req_id.

public void setQuery_type(String query_type)

- 设置query_type.

com.alibaba.idst.nls.StageListener

语音服务引擎状态变更回调接口，服务状态的改变、音量大小的回调、语音文件的生成通过本接口获取。

录音开始的回调

public void onStartRecording(NlsClient recognizer)

- 说明 录音开始

  参数

  返回值 NlsClient

录音结束的回调

public void onStopRecording(NlsClient recognizer)

- 说明 录音结束

  参数

  返回值 NlsClient

识别开始的回调

public void onStartRecognizing(NlsClient recognizer)

- 说明 识别开始

  参数

  返回值 NlsClient

识别结束的回调

public void onStopRecognizing(NlsClient recognizer)

- 说明 识别结束

参数

返回值 NlsClient

音量大小回调

public void onVoiceVolume(int volume)

说明 获取音量

参数

- volume 录音音量，范围0-100

返回值 null

获取流式返回录音语音

public void onVoiceData(short[] data, int length)

- 说明 当引擎获取到每帧录音时回调，用于客户端应用获取录音数据，默认20ms每帧数据
- 参数
  - data 录音数据 原始未压缩 未判断VAD
  - length 该帧录音长度

com.alibaba.idst.nls.NlsListener

语音服务结果的回调类，语音服务是一个相对较长过程，为了不影响客户端的正常工作，语音sdk被设计成异步模式，调用程序将语音客户端建立好并将语音数据交给sdk后就可以离开。在收到语音数据之后的处理需要通过实现该接口来完成。每一个识别过程创建的时候都需要注入侦听者以响应语音服务的结果。

语音识别结果的回调接口

public void onRecognizingResult(int status, RecognizedResult result)

- 说明 语音识别结果回调接口，识别结束时回调。(使用流式返回的key时，会多次回调本接口得到实时识别结果)
- 参数
  - status 标识识别结果的状态，详情见下方。
  - result 返回识别结果的数据结构，详情见下方。
- 返回值 null

- result中的字段值：

字段名	含义	类型
asr_out	语音识别结果json	String
finish	流式返回时判断返回是否结束的字段，finish=true表示返回结束	Boolean
bstream_attached	判断后面是否跟有语音流	Boolean
status_code	状态码	int

错误码

- 客户端错误码

字段名	错误码	含义
SUCCESS	0	成功
RECOGNIZE_ERROR	1	识别失败
USER_CANCEL	520	用户取消
CONNECT_ERROR	530	网络及通讯异常
NOTHING	540	语音服务异常
RECORDING_ERROR	550	录音及语音识别异常
ERROR_CLICK_TOOMUCH	570	用户点击过快

- 服务端返回结果错误码

状态	status_code	CloseFrame状态码	HTTP语义
成功	200	1000	成功处理
请求格式有误	400	4400	错误请求
需要鉴权信息	401	4401	请求要求身份验证
鉴权失败	403	4403	服务器拒绝请求
超出最大并发量	429	4429	太多请求
请求超时	408	4408	处理请求超时
处理出错	500	4500	服务器内部错误
服务不可用	503	4503	服务不可用

完整示例

```
package com.alibaba.idst.nlsdemo;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

import com.alibaba.idst.R;
import com.alibaba.idst.nls.NlsClient;
import com.alibaba.idst.nls.NlsListener;
import com.alibaba.idst.nls.StageListener;
import com.alibaba.idst.nls.internal.protocol.NlsRequest;
import com.alibaba.idst.nls.internal.protocol.NlsRequestProto;

public class PublicAsrActivity extends Activity {

private boolean isRecognizing = false;
private EditText mFullEdit;
private EditText mResultEdit;
private Button mStartButton;
private Button mStopButton;
private NlsClient mNlsClient;
private NlsRequest mNlsRequest = initNlsRequest();

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_public_asr);

mFullEdit = (EditText) findViewById(R.id.editText2);
mResultEdit = (EditText) findViewById(R.id.editText);
mStartButton = (Button) findViewById(R.id.button);
mStopButton = (Button) findViewById(R.id.button2);

String appkey = ""; //请设置简介页面的Appkey

mNlsRequest.setApp_key(appkey); //appkey列表中获取
mNlsRequest.setAsr_sc("opu"); //设置语音格式

NlsClient.openLog(true);
NlsClient.configure(getApplicationContext()); //全局配置
mNlsClient = NlsClient.newInstance(this, mRecognizeListener, mStageListener,mNlsRequest); //实例化NlsClient

mNlsClient.setMaxRecordTime(60000); //设置最长语音
mNlsClient.setMaxStallTime(1000); //设置最短语音
mNlsClient.setMinRecordTime(500); //设置最大录音中断时间
mNlsClient.setRecordAutoStop(false); //设置VAD
mNlsClient.setMinVoiceValueInterval(200); //设置音量回调时长

initStartRecognizing();
initStopRecognizing();
}
```

```
private NlsRequest initNlsRequest(){
NlsRequestProto proto = new NlsRequestProto();
proto.setApp_user_id("xxx"); //设置在应用中的用户名，可选
return new NlsRequest(proto);

}

private void initStartRecognizing(){
mStartButton.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
isRecognizing = true;
mResultEdit.setText("正在录音，请稍候！");
mNlsRequest.authorize("", ""); //请替换为用户申请到的Access Key ID和Access Key Secret
mNlsClient.start();
mStartButton.setText("录音中。。。");
}
});
}

private void initStopRecognizing(){
mStopButton.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
isRecognizing = false;
mResultEdit.setText("");
mNlsClient.stop();
mStartButton.setText("开始 录音");

}
});
}

private NlsListener mRecognizeListener = new NlsListener() {

@Override
public void onRecognizingResult(int status, RecognizedResult result) {
switch (status) {
case NlsClient.ErrorCode.SUCCESS:
Log.i("asr", "[demo] callback onRecognizResult " + result.asr_out);
mResultEdit.setText(result.asr_out);
mFullEdit.setText(result.asr_out);
break;
case NlsClient.ErrorCode.RECOGNIZE_ERROR:
Toast.makeText(PublicAsrActivity.this, "recognizer error", Toast.LENGTH_LONG).show();
break;
case NlsClient.ErrorCode.RECORDING_ERROR:
Toast.makeText(PublicAsrActivity.this,"recording error",Toast.LENGTH_LONG).show();
break;
case NlsClient.ErrorCode.NOTHING:
Toast.makeText(PublicAsrActivity.this,"nothing",Toast.LENGTH_LONG).show();
break;
}
isRecognizing = false;
}
```

```
} ;

private StageListener mStageListener = new StageListener() {
@Override
public void onStartRecognizing(NlsClient recognizer) {
super.onStartRecognizing(recognizer); //To change body of overridden methods use File | Settings | File Templates.
}

@Override
public void onStopRecognizing(NlsClient recognizer) {
super.onStopRecognizing(recognizer); //To change body of overridden methods use File | Settings | File Templates.
}

@Override
public void onStartRecording(NlsClient recognizer) {
super.onStartRecording(recognizer); //To change body of overridden methods use File | Settings | File Templates.
}

@Override
public void onStopRecording(NlsClient recognizer) {
super.onStopRecording(recognizer); //To change body of overridden methods use File | Settings | File Templates.
}

@Override
public void onVoiceVolume(int volume) {
super.onVoiceVolume(volume);
}

};

}
```

功能介绍

语音iOS SDK提供一句话识别服务，提供将实时短语音转成文字的功能，可直接用于语音搜索类应用。

SDK下载地址

一句话识别iOSSDK

重要接口说明

发送语音请求的对象及方法NlsRequest.h

语音请求初始化方法

- (instancetype)init;

- 说明 语音识别、语音合成的语音请求初始化方法
- 返回值 self

设置语音请求的appkey

- (void)setAppkey:(NSString *)appKey;

- 说明 设置语音请求的appkey。
- 参数
  - appKey 用户申请的appKey
- 返回值 无

设置发送的请求是否需要带语音数据

- (void)setBstreamAttached:(BOOL)bstreamAttached;

- 说明 设置发送的请求是否需要带语音数据。若发送的是语音识别请求，则bstreamAttached为YES；若发送的是语音合成请求，则bstreamAttached为NO。
- 参数
  - bstreamAttached 请求是否需要带语音数据。
- 返回值 无

设置语音识别ASR请求

- (void)setAsrRequest:(NSString *)userId;

- 说明 设置语音识别ASR请求
- 参数
  - userId 用户id
- 返回值 无

数加验证

- (void)Authorize:(NSString *)authId withSecret:(NSString *)secret;

- 说明 数加验证，未经过数加验证的语音请求均为非法请求。
- 参数
  - authId 数加验证的ak_id
  - secret 数加验证的ak_secret
- 返回值 无

将语音请求NlsRequest对象转换成JSON字符串

+ (NSString *)getJSONStringfromNlsRequest:(NlsRequest *)nlsRequest;

- 说明 将语音请求NlsRequest对象转换成JSON字符串形式。

- 参数
  - nlsRequest NlsRequest对象
- 返回值 NlsRequest的JSON字符串

将对象转换成JSONString方法

+ (NSString *)getJSONString:(id)obj options:(NSJSONWritingOptions)options error:(NSError)error;**

- 说明 将object转换成JSONString。
- 参数
  - obj 被转化对象
  - options NSJSONWritingOptions
  - error NSError
- 返回值 NlsRequest的JSON字符串

将对象转换成NSDictionary方法

+ (NSDictionary *)getObjectData:(id)obj;

- 说明 将object转换成NSDictionary。
- 参数
  - obj 被转化对象
- 返回值 NSDictionary

语音服务SDK的核心类NlsRecognizer.h

语音服务SDK的核心类，封装了录音设备的初始化，压缩处理，语音检测（VAD）等复杂逻辑，自动的将语音数据同步传送到语音服务器上。开发者只需要传递正确的delegate，就能完成语音识别和语音合成。

语音识别的关键回调函数

-(void)recognizer:(NlsRecognizer *)recognizer *didCompleteRecognizingWithResult:(NlsRecognizerResult*)*result error:(NSError*)error;

- 说明 语音识别的关键回调函数，delegate必须实现。若appkey为流式返回appkey，将会多次回调该方法。
- 参数
  - recognizer NlsRecognizer
  - result 返回值对象 NlsRecognizerResult
  - error 语音识别错误和异常 NSError
- 返回值 无

返回录音的语音音量

-(void)recognizer:(NlsRecognizer *)recognizer recordingWithVoiceVolume:(NSUInteger)voiceVolume;

- 说明 返回录音的语音音量，调用频率取决于SDK内部设定。

- 参数
  - recognizer NlsRecognizer
  - voiceVolume 0-100的数值
- 返回值 无

返回录音的数据

-(void)recognizer:(NlsRecognizer *)recognizer recordingWithVoiceData:(NSData*)voiceData;

- 说明 返回录音的数据，调用频率取决于SDK内部设定。
- 参数
  - recognizer NlsRecognizer
  - frame 返回的语音
- 返回值 无

开始录音的回调通知

-(void)recognizerDidStartRecording:(NlsRecognizer *)recognizer;

- 说明 开始录音的回调通知
- 参数
  - recognizer NlsRecognizer
- 返回值 无

停止录音的回调通知

-(void)recognizerDidStopRecording:(NlsRecognizer *)recognizer;

- 说明 停止录音的回调通知
- 参数
  - recognizer NlsRecognizer
- 返回值 无

录音模式下停止的回调通知

-(void)recognizerDidStopRecording:(NlsRecognizer *)recognizer withRecorderData:(NSData*)data;

- 说明 录音模式下停止的回调通知，可得到录音数据（保留方法）
- 参数
  - recognizer NlsRecognizer
  - data 录音数据
- 返回值 无

设置SDK工作模式

NlsRecognizer @property(nonatomic,assign,readwrite) kNlsRecognizerMode mode;

- 说明 设置语音SDK的工作模式，若不设置，则为kMODE_RECOGNIZER

设置SDK是否监听App状态

NlsRecognizer @property(nonatomic,assign,readwrite) BOOL cancelOnAppEntersBackground;

- 说明 设置SDK是否监听App状态，缺省为NO。如果设为YES，则SDK会监听App状态，一旦切换到后台，就自动取消请求。

设置cancel时是否回调onRecognizeComplete方法

NlsRecognizer @property(nonatomic,assign,readwrite) BOOL enableUserCancelCallback;

- 说明 NlsRecognizer的cancel方法被调用的时候，会触发delegate的onRecognizeComplete方法，错误码为kERR_USER_CANCELED,错误信息为kNlsRecognizerErrorUserCanceled。如果App不想被回调，请设置为NO即可。参见 cancel方法。

设置是否记录语音

NlsRecognizer @property(nonatomic,assign,readwrite) BOOL enableVoiceLog;

- 说明 打开语音记录功能，SDK将会把语音识别的文件记录到当前App的document下面，调试用。缺省关闭，在DEBUG模式下有效，Release模式下无效。（保留字段）

配置语音服务模块的基础参数

+(void)configure;

- 说明 配置语音服务模块的基础参数，请在App启动的时候调用
- 参数 无
- 返回值 无

初始化NlsRecognizer

-(id)initWithNlsRequest:(NlsRequest *)nlsRequest svcURL:(NSString *)svcURL;

- 说明 初始化NlsRecognizer，注意: 通过该API使用的appKey必须提前通过configure函数预先配置好
- 参数
  - nlsRequest 语音请求NlsRequest
  - svcURL 语音服务地址
- 返回值 无

设置是否打开VAD开关

-(void)setVad;

- 说明 设置是否打开VAD开关，VAD默认关闭。
- 参数
  - isVad 是否打开静音检测开关

- 返回值 无

检测到用户语音后，自动停止的间隙

-(void)setVadAutoStopTimeInterval:(NSTimeInterval) timeinterval;

- 说明 检测到用户语音后，自动停止的间隙。
- 参数
  - timeinterval 静音触发自动停止的时长，以秒表示.缺省0.7s
- 返回值 无

打开VAD时，设置最短录音时间，缺省时为5s

-(void)setMinRecordTime:(NSTimeInterval) timeinterval;

- 说明 打开VAD时，设置最短录音时间，缺省时为5s
- 参数
  - timeinterval 最短录音时间，缺省时为5s
- 返回值 无

设置最长录音时间，缺省时为60s

-(void)setMaxRecordTime:(NSTimeInterval) timeinterval;

- 说明 设置最长录音时间，缺省时为60s
- 参数
  - timeinterval 最短录音时间，缺省时为60s
- 返回值 无

语音主服务是否可用

+(BOOL)isServiceAvailable;

- 说明 语音主服务是否可用，开发者可以根据测返回值，调整UI行为
- 返回值 返回语音主服务当前是否可用。

开始语音识别

-(void)start;

- 说明 开始语音识别。打开录音设备，同时开始识别。
- 返回值 无

停止语音识别

-(void)stop;

- 说明 停止语音识别。停止录音设备，delegate的didStopRecord会被回调。网络请求在后台继续，如果有识别结果返回，则会通过delegate的didCompleteRecognizingWithResult回调方法单独返

回。
- 返回值 无

取消语音识别

-(void)cancel;

- 说明 取消语音识别。取消语音识别，录音会停止，网络请求会取消。根据enableUserCancelCallback的设置，决定是否回调delegate方法。
- 返回值 无

语音识别是否已经开始

-(BOOL)isStarted;

- 说明 语音识别是否已经开始
- 返回值 YES表示语音识别已经开始，NO表示语音识别未开始。

参数和错误码说明

发送ASR语音请求的参数：

```
{
"requests" : {
"asr_in" : {
"version" : "3.0",// 协议版本号
"asrSC" : "opu",// 输入的语音格式，默认opu
"user_id" : ""// 用户名，可选项
},
"context" : {
"auth" : {} //requested
}
},
"app_key" : "",
"bstream_attached" : true,// 请求包的后面是不是还接着二进制语音流。语音识别时bstream_attached = YES 。
"version" : "4.0"// 协议版本号
}
```

返回的识别结果result是一个NlsRecognizerResult的对象：

```
{
"status" : "1",// 服务器状态，0为失败，非零为成功
"id" : "",// 透传系统始终的uuid,服务端配置是否返回
"finish" : "1",// 0为未结束，非零为结束，识别是否已经结束
"results" : {
"asr_out" : {
"result" : "",// 语音识别结果
"status" : 1,// 服务器状态，0为失败，非零为成功
"finish" : 1,// 0为未结束，非零为结束，识别是否已经结束
"version" : "4.0"
```

```
},
"out" : {}//保留字段
},
"bstream_attached" : false,// 应答包的后面是不是还接着二进制语音流。
"version" : "4.0"// 协议版本号
}
```

若识别发生错误，recognizer:didCompleteRecognizingWithResult:error:的回调函数中error不为nil。相应错误码的对应表如下所示。

- 客户端错误码

字段名	错误码	含义
kERR_NO_ERROR	0	成功
kERR_GENERIC_ERROR	1	识别失败
kERR_USER_CANCELED	520	用户取消
kERR_NETWORK_ERROR	530	网络及通讯异常
kERR_SERVICE_ERROR	540	语音服务异常或被降级
kERR_VOICE_ERROR	550	录音及语音识别异常
kERR_MIC_ERROR	560	Mic无法访问或硬件异常
kERR_TOOSHORT_ERROR	570	用户点击过快

- 服务端返回结果错误码

状态	status_code	CloseFrame状态码	HTTP语义
成功	200	1000	成功处理
请求格式有误	400	4400	错误请求
需要鉴权信息	401	4401	请求要求身份验证
鉴权失败	403	4403	服务器拒绝请求
超出最大并发量	429	4429	太多请求
请求超时	408	4408	处理请求超时
处理出错	500	4500	服务器内部错误
服务不可用	503	4503	服务不可用

完整示例

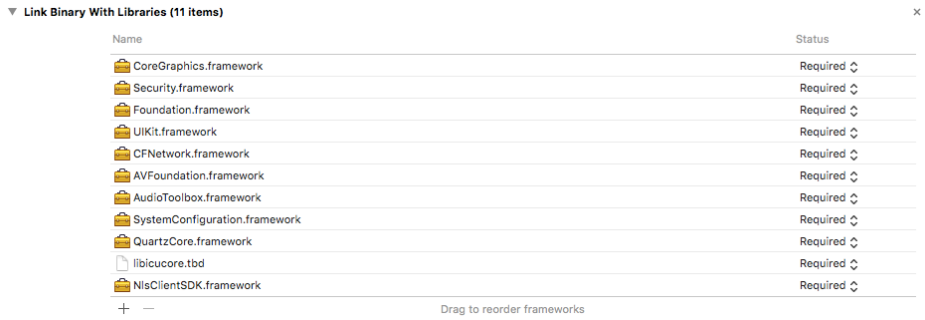
创建应用

使用Xcode创建iOS application应用工程。

添加Framework

在Xcode工程中需要引入所需要的framework，NlsClientSDK.framework。

添加方法：选中工程，点击TARGETS，在右侧的Build Phases中选择 Link Binary With Libraries，点击上图中左下角的+号，在弹出界面中依次添加所依赖的framework。




```
    $lipo -info NlsClientSDK
Architectures in the fat file: NlsClientSDK are: armv7 i386 x86_64 arm64
```

引入头文件

在需要调用SDK的文件中，添加如下头文件:

```
    #import <NlsClientSDK/NlsClientSDK.h>
```

语音服务注册

AppDelegate中注册语音服务：

```
    #import "AppDelegate.h"
#import "ViewController.h"
#import <NlsClientSDK/NlsClientSDK.h>

@interface AppDelegate ()
@end

@implementation AppDelegate
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
-
#warning configure语音服务，必须在调用语音服务前执行该方法。
[NlsRecognizer configure];
.....
 }
```

实现语音识别功能

ViewController中实现语音识别方法：

```
    #import "ViewController.h"
#import <NlsClientSDK/NlsClientSDK.h>

@interface ViewController ()<NlsRecognizerDelegate>

@property(nonatomic,strong) NlsRecognizer *recognizer;

@end

@implementation ViewController

- (void)viewWillAppear:(BOOL)animated
{
[super viewWillAppear:animated];

// 检查asr服务是否可用
if([NlsRecognizer isServiceAvailable]) {
NSLog(@"当前语音服务可用");
}
else {
NSLog(@"当前语音服务不可用");
}

// 监测语音服务状态
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(asrStatusChanged:)
name:kNlsRecognizerServiceStatusChanged object:nil];

}

#pragma mark - Actions
- (void)onStartAsrButtonClick:(id)sender {

// 初始化语音请求类
NlsRequest * nlsRequest = [[NlsRequest alloc] init];
#warning appkey请从 "快速开始" 帮助页面的appkey列表中获取
[nlsRequest setAppkey:@""]; // requested
[nlsRequest setBstreamAttached:YES]; // requested 语音识别，setBstreamAttached = YES
[nlsRequest setAsrRequest:@"input_userid"]; // requested 设置asr_in参数
#warning 请修改为您在阿里云申请的数字验证串Access Key ID和Access Key Secret
[nlsRequest Authorize:@"" withSecret:@""]; // requested

// 初始化语音服务核心类
NlsRecognizer *r = [[NlsRecognizer alloc] initWithNlsRequest:nlsRequest svcURL:nil]; // requested 采用默认svcURL
//vad：自动语音检测，可根据需要设置是否打开，YES为打开，NO为关闭。
[r setVad:self.setVadFlag];
r.delegate = self;
r.cancelOnAppEntersBackground = YES;
r.enableUserCancelCallback = YES;
self.recognizer = r;

NSString *nlsRequestJSONString = [NlsRequest getJSONStringfromNlsRequest:nlsRequest];
NSLog(@"setupAsrIn : %@",nlsRequestJSONString);

//开始语音识别
[self.recognizer start];
```

```
}

- (void)onStopAsrButtonClick:(id)sender {
//结束语音识别
[self.recognizer stop];
}

#pragma mark - Notification Callbacks
-(void)asrStatusChanged:(NSNotification*)notify{
//处理网络变化
}

#pragma mark - RecognizerDelegate
-(void) recognizer:(NlsRecognizer *)recognizer didCompleteRecognizingWithResult:(NlsRecognizerResult*)result
error:(NSError*)error{
//处理识别结果和错误信息
//若appkey为流式返回appkey，将会多次回调该方法
}

-(void) recognizer:(NlsRecognizer *)recognizer recordingWithVoiceVolume:(NSUInteger)voiceVolume{
//处理音量变化
}

-(void) recognizerDidStartRecording:(NlsRecognizer *)recognizer{
//处理开始识别事件
}

-(void) recognizerDidStopRecording:(NlsRecognizer *)recognizer{
//处理结束识别事件
}

@end
```

FAQ

问题1：bitcode。

```
ld: 'xxx/NlsClientSDK.framework/NlsClientSDK(NlsRecognizer.o)' does not contain bitcode. You must rebuild it with
bitcode enabled (Xcode setting ENABLE_BITCODE), obtain an updated library from the vendor, or disable bitcode
for this target. for architecture arm64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

解决1：打开项目 - targets - build settings - Enable Bitcode-设置为No。

问题2：数加验证失败4403。

```
{
NSLocalizedDescription = "server closed connection, code:4403, reason:Unauthorized AppKey [xxx], wasClean:1";
}
```

解决2：检查数加验证的 ak_id 和 ak_secret 是否正确;检查 appkey 填写是否正确。

```
#warning 请修改为您在阿里云申请的数字验证串Authorize withSecret
[nlsRequest Authorize:@"Access Key ID" withSecret:@"Access Key Secret"]; // requested Access Key ID和Access Key Secret


#warning appkey请从 "快速开始" 帮助页面的appkey列表中获取。
[nlsRequest setAppkey:@"your_appkey"]; // requested
```

问题3：Assertion failed。

解决3：检查在开始语音识别前，是否注册;检查NlsRecognizer初始化时svcURL是否为nil。

```
//Config appkey.
[NlsRecognizer configure];


NlsRecognizer *r = [[NlsRecognizer alloc] initWithNlsRequest:nlsRequest svcURL:nil];
```

问题4：错误码4400。

```
{
NSLocalizedDescription = "server closed connection, code:4400, reason:illegal params, operation forbidden, wasClean:1";
}
```

解决4：检查appkey是否为空，填写正确的appkey。

```
#warning 请修改为您在阿里云申请的APP_KEY
[nlsRequest setAppkey:@"your_appkey"]; // requested
```

简介

Opus编码器 是一个有损声音编码的格式，由互联网工程任务组（IETF）进来开发，适用于网络上的实时声音传输，标准格式为RFC 6716。Opus 格式是一个开放格式，使用上没有任何专利或限制。

- Bitrates from 6 kb/s to 510 kb/s
- Sampling rates from 8 kHz (narrowband) to 48 kHz (fullband)
- Frame sizes from 2.5 ms to 60 ms
- Support for both constant bitrate (CBR) and variable bitrate (VBR)
- Audio bandwidth from narrowband to fullband
- Support for speech and music
- Support for mono and stereo

Why Opus ?

OPUS 在NLS服务中的使用

目前我们的Android 和 iOS客户端sdk中集成了OPUS编码（encoder）功能。根据我们的参数进行语音编解码，可以做到大概9:1的语音压缩比，能够有效的节省传输带宽和响应时间。

在使用Java SDK时，SDK本身不支持OPUS的编解码，用户可以根据本文档的设置自行进行编码，将生成的opu格式的文件通过SDK做语音识别。

下载

opus更详细的说明请参考 http://www.opus-codec.org/ 。

opus版本请使用1.0.3以上版本 http://www.opus-codec.org/downloads/older.shtml.en ，下载libopus Source code。

接口说明： http://www.opus-codec.org/docs/html_api-1.0.3/index.html 。

主要参数配置

Encoder配置：

```
 //encoder's settings

OpusEncoder *pOpusEnc = opus_encoder_create(16000, 1, OPUS_APPLICATION_VOIP,&error); // 初始化操作.创建编码器


opus_encoder_ctl(pOpusEnc, OPUS_SET_VBR(1)); //可变比特率
opus_encoder_ctl(pOpusEnc, OPUS_SET_BITRATE(27800)); //比特率设置为27800
opus_encoder_ctl(pOpusEnc, OPUS_SET_COMPLEXITY(8));
opus_encoder_ctl(pOpusEnc, OPUS_SET_SIGNAL(OPUS_SIGNAL_VOICE)); //信号类型



opus_int32 opus_encode ( OpusEncoder * st,
const opus_int16 * pcm,
int frame_size,
unsigned char * data,
opus_int32 max_data_bytes
) //Encodes an Opus frame.

void opus_encoder_destroy ( OpusEncoder * st )//释放编码器
```

Decoder 配置

```


//decoder's settings
```

```
OpusDecoder* opus_decoder_create (16000,
int 1,
int * error
) //创建解码器

int opus_decode ( OpusDecoder * st,
const unsigned char * data,
opus_int32 len,
opus_int16 * pcm,
int frame_size,
int decode_fec
) //解码操作

void opus_decoder_destroy ( OpusDecoder * st ) //释放解码器
```

实时语音识别

- 实时语音转写服务适用场景如实时会议记录、视频直播实时字幕等。
- 本文档提供服务端程序的Java SDK，Java SDK内部不自带语音采集功能，只提供将语音流实时转写成文字的功能。
- 支持16k 16bit PCM、16k 16bit WAV的语音格式。
- sdk本身未设置超时时间，用户可以调用client.close()方法关闭链接。

支持的app_key

app_key	描述
nls-service-shurufa16khz	16kHz采样率的pcm、wav语音文件

SDK下载地址

实时语音识别JavaSDK&Demo

示例说明

下载地址中包含了Java SDK的jar包，以及测试用的demo工程，用户只需在工程中通过“Java Build Path”->“Add External JARs”将jar包导入，即可运行。推荐使用IntelliJ IDEA导入项目。

SDK调用顺序

创建一个NlsClient的实例并调用init()方法来初始化客户端

提取语音数据并创建语音识别请求，至少填写appKey及需要识别的语音数据的格式。创建一个NlsListener的实现类。

调用NlsClient的createNlsFuture（第2步中的listener实例作为入参之一，用来处理返回结果）方法获取future，通过future的sendVoice方法来发送语音数据并在listener中处理返回结果。

通过future的sendFinishSignal来结束语音文件的发送，ASR服务收到这个结束信号后，会返回处理结果。

调用NlsClient的close()方法来关闭客户端并释放资源。

重要接口说明

com.alibaba.idst.nls.NlsClient

语音sdk对外提供的类，调用程序通过调用该类的init()、close()、createNlsFuture()等方法来打开、关闭或发送语音数据。

初始化NlsClient

public void init()

- 说明 初始化NlsClient，创建好websocket client factory
- 返回值
  - null

实例化NlsFuture请求

public NlsFuture createNlsFuture(NlsRequest req, com.alibaba.idst.nls.event.NlsListener listener)

- 说明 实例化NlsFuture请求，传入请求和监听器
- 参数
  - req 请求对象
  - listener 回调数据的监听器
- 返回值
  - future

关闭NlsClient

public void close()

- 说明 关闭websocket client factory，释放资源
- 参数
  - null
- 返回值
  - null

com.alibaba.idst.nls.event.NlsListener

语音识别是一个相对较长的过程，为了不影响服务端的正常工作，语音sdk被设计成异步模式，调用程序将语音客户端建立好并将语音数据交给sdk后就可以离开。在收到语音数据之后的处理需要通过实现该接口来完成。每一个识别过程创建的时候都需要注入侦听者以响应语音识别的结果。

识别结果回调

void onMessageReceived(NlsEvent e);

- 说明 识别结果回调
- 参数
  - NlsEvent Nls服务结果的对象 识别结果在e.getResponse()中
- 返回值
  - null

识别失败回调

void onOperationFailed(NlsEvent e);

- 说明 识别失败回调
- 参数
  - NlsEvent Nls服务结果的对象 错误信息在e.getErrorMessage()中
- 返回值
  - null

socket 连接关闭的回调

void onChannelClosed(NlsEvent e);

- 说明 socket 连接关闭的回调
- 参数
  - NlsEvent Nls服务结果的对象
- 返回值
  - null

com.alibaba.idst.nls.protocol.NlsRequest

语音识别的请求封装对象。调用程序需要至少在请求对象里设定好调用者的appKey和语音数据的格式以便后台服务能正确识别并识别语音。

初始化Nls 请求：NlsRequest mNlsRequest = new NlsRequest(proto)

public void setAppkey(String appkey)

- 说明 业务方或者业务场景的标记。
- 参数
  - appkey
- 返回值
  - null

public void setFormat(String format)

- 说明 设置语音识别的语音格式，目前仅支持pcm。 该项在使用语音识别服务时必须设置，起到初始化作用。
- 参数
  - format 目前仅支持pcm。
- 返回值
  - null

public void setResponseMode(String responseMode)

- 说明 设置返回结果的模式，默认为streaming。
- 参数
  - responseMode 返回模式，目前支持normal和streaming, 默认为streaming，支持流式返回结果，多次调用结果返回方法；normal表示调用一次结果返回方法。
- 返回值
  - null

public void setSampleRate(int sampleRate)

- 说明 采样率，目前仅支持8000和16000，默认为16000。
- 参数
  - sampleRate 采样率，目前仅支持8000和16000，默认为16000。
- 返回值
  - null

public void authorize(String id, String secret)

- 说明 数加认证模块，所有的请求都必须通过authorize方法认证通过，才可以使用。 id和secret需要申请获取。
- 参数
  - id 申请的数加Access Key ID。
  - secret 申请的数加Access Key ID对应的密钥Access Key Secret。
- 返回值

- null

com.alibaba.idst.nls.NlsFuture

NlsFuture是具体操作语音的对象类，语音数据的发送/接收都通过这个类进行操作。

public NlsFuture sendVoice(byte[] data, int offset, int length)

- 说明 语音识别时，发送语音文件的方法
- 参数
  - data byte[]类型的语音流
  - offset
  - length
- 返回值
  - NlsFuture

public NlsFuture sendFinishSignal()

- 说明 语音识别结束时，发送结束符
- 返回值
  - NlsFuture

public boolean await(int timeout)

- 说明 请求超时时间
- 参数
  - timeout 请求发送出去后，等待服务端结果返回的超时时间，单位ms
- 返回值
  - true or false

com.alibaba.idst.nls.protocol.NlsResponse

语音识别的返回结果封装对象。返回结果告知语音识别是否成功，语音识别结果或部分结果(视调用程序需要的返回方式而定)，语音识别是否已经结束。

返回参数：

属性	值类型	是否必须	说明	取值方法
version	String	是	协议版本	getVersion()
request_id	String	是	当前请求id	getId()
status_code	Int	是	状态码	getStatus_code()
result	JsonObject	否	返回结果，当已有识别结果时返回	getResult()

- 其中请求结果result的字段如下:

属性	值类型	是否必须	说明	取值方法
sentence_id	Int	是	当前句子序号	getRequest_id()
begin_time	Int	是	当前句子开始时间	getBegin_time()
end_time	Int	是	当前句子结束时间，当为streaming模式时，中间结果返回 - 1	getEnd_time()
status_code	Int	是	状态码，normal模式时，结果为0；streaming模式时，最终结果为0，中间结果为1	getRecognizeStatusCode()
text	String	是	当前识别结果	getText()

错误码

状态	status_code	CloseFrame状态码	HTTP语义
成功	200	1000	成功处理
请求格式有误	400	4400	错误请求
需要鉴权信息	401	4401	请求要求身份验证
鉴权失败	403	4403	服务器拒绝请求
超出最大并发量	429	4429	太多请求
请求超时	408	4408	处理请求超时
处理出错	500	4500	服务器内部错误
服务不可用	503	4503	服务不可用

完整示例

```
package com.alibaba.idst.nls.demo;

import com.alibaba.idst.nls.NlsClient;
import com.alibaba.idst.nls.NlsFuture;
import com.alibaba.idst.nls.event.NlsEvent;
import com.alibaba.idst.nls.event.NlsListener;
import com.alibaba.idst.nls.protocol.NlsRequest;
```

```
import com.alibaba.idst.nls.protocol.NlsResponse;

import java.io.File;
import java.io.FileInputStream;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class NlsDemo implements NlsListener {
private NlsClient client = new NlsClient();
private static final String asrSC = "pcm";

static Logger logger = LoggerFactory.getLogger(NlsDemo.class);
public String filePath = "";
public String appKey = "";
String ak_id = ""; //Access Key ID
String ak_secret = ""; //Access Key Secret


public NlsDemo() {
}

public void shutDown() {
logger.debug("close NLS client manually!");
client.close();
logger.debug("demo done");
}

public void start() {
logger.debug("init Nls client...");
client.init();
}


public void hearIt() {
logger.debug("open audio file...");
FileInputStream fis = null;
try {
File file = new File(filePath);
fis = new FileInputStream(file);
} catch (Exception e) {
e.printStackTrace();
}

if(fis != null) {
logger.debug("create NLS future");
try {
NlsRequest req = new NlsRequest();
req.setAppkey(appKey);
req.setFormat(asrSC);
req.setResponseMode("streaming");
req.setSampleRate(16000);
// the id and the id secret
req.authorize(ak_id, ak_secret);
NlsFuture future = client.createNlsFuture(req,this);
logger.debug("call NLS service");
```

```
byte[] b = new byte[8000];
int len = 0;
while((len = fis.read(b)) > 0) {
future.sendVoice(b, 0, len);
//使用文件测试，需要每8000byte sleep250ms。
//如果发送实时语音流，不需要进行sleep操作。
Thread.sleep(250);
}
logger.debug("send finish signal!");
future.sendFinishSignal();

logger.debug("main thread enter waiting .");
future.await(10000);


} catch(Exception e) {
e.printStackTrace();
}
logger.debug("calling NLS service end");
}
}

@Override
public void onMessageReceived(NlsEvent e) {
NlsResponse response = e.getResponse();
String result = "";
if (response.result != null) {
logger.info("status code = {},get recognize result: {}",response.getStatus_code(),response.getResult().toString());
}else{
logger.info("get an acknowledge package from server.");
}
}

@Override
public void onOperationFailed(NlsEvent e) {
logger.error("on operation failed: {}",e.getErrorMessage());

}

@Override
public void onChannelClosed(NlsEvent e) {
logger.debug("on websocket closed.");
}

/**
* @param args
*/
public static void main(String[] args) {
NlsDemo lun = new NlsDemo();

logger.info("start ....");
if (args.length < 4) {
logger.debug("NlsDemo <app-key> <Id> <Secret> <opu-file>");
System.exit(-1);
}
```

```
lun.appKey = args[0];
lun.ak_id = args[1];
lun.ak_secret = args[2];
lun.filePath = args[3];


lun.start();
lun.hearIt();
lun.shutDown();
}

}
```

录音文件识别

录音文件识别

“录音文件识别”服务是以**Restful**方式提供长语音文件识别接口。基于该Restful API接口，开发者可以方便获取语音识别能力。

本文档描述了使用“录音文件识别”服务 Restful API 的方法，并提供了完整示例供开发者参考，适用使用 HTTPS 接口的开发人员。

功能介绍

- 支持**多轨wav格式**的长语音文件识别
- 目前只支持aLaw和linear两种音频格式；
- 目前只支持8k和16k的采样率；
- 支持普通话识别

支持的app_key

app_key	描述
nls-service-telephone8khz	8kHz采样率语音文件
nls-service-shurufa16khz	16kHz采样率语音文件

使用步骤

申请账号和开通服务。

用户把语音文件存放到OSS里，为了数据安全，需要设置数据为私有。当用户调用语音识别服务的时候，可以生成有过期时间的文件链接https://help.aliyun.com/knowledge_detail/5974651.html，也可以通过SDK生成有过期时间的文件链接https://help.aliyun.com/knowledge_detail/6716167.html， 这样我们的服务可以访问要识别的文件。

按照“请求调用接口”提交识别请求并获取id。

通过id调用“结果查询接口”获得识别的结果。

接口调用方式

“录音文件识别”API包括两部分：POST方式的“请求调用接口”，GET方式的“结果查询接口”。

请求调用接口

提交识别请求以获取id。

URL说明

协议	URL	方法	参数
HTTPS	nlsapi.aliyun.com/transcriptions	POST	JSON字符串

输入参数

“请求调用接口”的关键请求参数，以json格式放置于Https Body内，并作为数加验证的body参数进行加密。

```
{
"app_key": "***",
"oss_link": "***",
"valid_times": [
```

```
{
"begin_time": 200,
"end_time":2000,
"channel_id": 0
}
]
}
```

- 请求参数JSON字符串:

属性	值类型	是否必须	说明
app_key	String	是	业务方或者业务场景的标记
oss_link	String	是	语音文件存放的OSS link地址
valid_times	List< ValidTime >	否	有效时间段信息，用来排查一些不必要的时间段

- 有效时间段ValidTime描述:

属性	值类型	是否必须	说明
begin_time	Int	是	有效时间段的起始点时间偏移(单位: ms)
end_time	Int	是	有效时间段的结束点时间偏移(单位: ms)
channel_id	Int	是	有效时间段的作用音轨序号(从0开始)

- 数加验证字符串:

数加验证字符串以json格式放置于Https headers内。数加验证过程可参见“官方服务API校验规范”。注意：语音使用的数加验证与官方略有差异，详细见下文「完整示例」-「语音服务数加鉴权代码」。

```
{
"Content-Type": "application/json",
"Accept": "application/json",
"date": gmtTime,
"Authorization": authorization
}
```

输出参数

```
{
"id": "***"
```

```
}
```

参数说明：

- 返回HTTP状态： 201 Created
- 返回参数JSON字符串：

属性	值类型	是否必须	说明
id	String	是	识别任务ID

结果查询接口

在提交完识别请求后，调用方将可以按照如下接口轮询结果。

URL说明

协议	URL	方法	参数
HTTPS	nlsapi.aliyun.com/transcriptions/< id >	GET	无

输入参数

用户通过id调用“结果查询”接口可获取识别结果，在接口调用过程时，需要设置一定的查询时间间隔，可参考**完整示例**。

- 请求参数:

id为识别任务ID。

- 数加验证字符串:

数加验证字符串以json格式放置于Https headers内。数加验证过程可参见“官方服务API校验规范”。注意：语音使用的数加验证与官方略有差异，详细见下文「完整示例」-「语音服务数加鉴权代码」。

```
{
"Content-Type": "application/json",
"Accept": "application/json",
"date": gmtTime,
"Authorization": authorization
}
```

输出参数

正常返回

```
{
"id": "***",
"status": "SUCCEED",
"result": [
{
"channel_id": 0,
"begin_time": 700,
"end_time": 3120,
"text": "你好，很高兴为您服务"
}
.....
]
}
```

正在识别

```
{
"id":"2e2bf1fbe4d34dab8b9e36488d0fa1e2",
"status":"RUNNING"
}
```

异常返回

```
{
"error_message": "UNSUPPORTED_SAMPLE_RATE",
"id": "***",
"status": "FAILED",
"status_code": 400
}
```

参数说明：

- 返回HTTP状态： 200 OK
- 返回参数JSON字符串：

属性	值类型	是否必须	说明
id	String	是	识别任务ID
status	String	是	该识别任务的当前状态，三种取值：RUNNING, SUCCEED, FAILED
status_code	Int	否	错误码。当status为FAILED时存在。
error_message	String	否	对错误状态的进一步描述。当status为FAILED时存在
result	List<	否	识别的结果数据。当

	SentenceResult>		status为SUCCEED时存在

- 错误码status_code描述：

状态码	说明
200	成功
400	无效的请求
401	需要鉴权信息
403	鉴权失败
404	不存在
408	请求超时
422	请求内容有误
429	超出最大并发
500	服务器内部出错
503	服务不可用

- 单句结果SentenceResult描述：

属性	值类型	是否必须	说明
channel_id	Int	是	该句所属音轨ID
begin_time	Int	是	该句的起始时间偏移(单位: ms)
end_time	Int	是	该句的结束时间偏移(单位: ms)
text	String	是	该句的识别文本结果

完整示例

Java Demo 下载地址

- 语音文件识别服务demo入口

```
package com.alibaba.idst.nls;

import com.alibaba.fastjson.JSON;
import com.alibaba.idst.nls.utils.*;

public class TranscriptionDemo {
```

```
/**
* 服务url
* */
private static String url = "https://nlsapi.aliyun.com/transcriptions";
private static RequestBody body = new RequestBody();
private static HttpUtil request = new HttpUtil();

public static void main(String[] args) throws InterruptedException {

//设置请求参数
body.setApp_key("xxx");
body.setOss_link("http://aliyun-nls.oss.aliyuncs.com/asr/fileASR/examples/nls-sample.wav");
body.addValid_time(100,2000,0); //validtime 可选字段 设置的是语音文件中希望识别的内容,begintime,endtime以及
channel
body.addValid_time(100,2000,1);

String ak_id = "xxx";
String ak_secret = "xxx";

System.out.println("Recognize begin!");

/*
* 发送录音转写请求
* **/
String bodyString;
bodyString = JSON.toJSONString(body);
String postResult = request.sendPost(url,bodyString,ak_id,ak_secret);
System.out.println("response is:"+postResult);




/*
* 通过TaskId获取识别结果
* **/
String TaskId = JSON.parseObject(postResult).getString("id");
String status = "RUNNING";
String getResult = "";

while (!status.equals("RUNNING")){
Thread.sleep(3000);
getResult = request.sendGet(url,TaskId,ak_id,ak_secret);
status = JSON.parseObject(getResult).getString("status");
System.out.println("response is:"+getResult);
}

System.out.println("Recognize over!");

}
}
```

- RequestBody类，语音识别服务请求中数加鉴权所需的body参数，也是HTTPS请求的data部分。

```
package com.alibaba.idst.nls.utils;
```

```
import java.util.*;

public class RequestBody {

private String app_key = null; //appkey 应用的key
private String oss_link = null; //语音文件存储地址
private List<validTime> valid_times =null; //有效时间段ValidTime描述,可选字段


public class validTime{

private int begin_time;
private int end_time;
private int channel_id;

public int getBegin_time() {
return begin_time;
}

public void setBegin_time(int begin_time) {
this.begin_time = begin_time;
}

public int getEnd_time() {
return end_time;
}

public void setEnd_time(int end_time) {
this.end_time = end_time;
}

public int getChannel_id() {
return channel_id;
}

public void setChannel_id(int channel_id) {
this.channel_id = channel_id;
}

}

public void setApp_key(String appKey){
app_key = appKey;
}
public void setOss_link(String fileLine){
oss_link = fileLine;
}
public void addValid_time(int beginTime,int endTime,int channelId){

if(valid_times == null){
valid_times = new ArrayList<validTime>();
}

validTime valid_time = new validTime();
```

```
valid_time.setBegin_time(beginTime);
valid_time.setEnd_time(endTime);
valid_time.setChannel_id(channelId);

valid_times.add(valid_time);
}

public List<validTime> getValid_times() {
return valid_times;
}

public String getOss_link() {
return oss_link;
}

public String getApp_key() {
return app_key;
}

}
```

- 语音服务数加鉴权代码。HttpUtil类，实现数加鉴权过程，并将其作为HTTPS请求的header部分。实现POST方式的请求识别方法和GET方式的结果查询方法。

```
package com.alibaba.idst.nls.utils;

import java.io.*;
import java.net.URL;
import java.net.URLConnection;
import java.security.MessageDigest;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
import javax.crypto.spec.SecretKeySpec;
import sun.misc.BASE64Encoder;
import javax.crypto.Mac;

@SuppressWarnings("restriction")
public class HttpUtil {

/*
* 计算MD5+BASE64
*/
public static String MD5Base64(String s) throws UnsupportedEncodingException {
if (s == null)
return null;
String encodeStr = "";

//string 编码必须为utf-8
byte[] utfBytes = s.getBytes("UTF-8");

MessageDigest mdTemp;
try {
mdTemp = MessageDigest.getInstance("MD5");
```

```
mdTemp.update(utfBytes);
byte[] md5Bytes = mdTemp.digest();
BASE64Encoder b64Encoder = new BASE64Encoder();
encodeStr = b64Encoder.encode(md5Bytes);
} catch (Exception e) {
throw new Error("Failed to generate MD5 : " + e.getMessage());
}
return encodeStr;
}

/*
* 计算 HMAC-SHA1
*/
public static String HMACSha1(String data, String key) {
String result;
try {

SecretKeySpec signingKey = new SecretKeySpec(key.getBytes(), "HmacSHA1");
Mac mac = Mac.getInstance("HmacSHA1");
mac.init(signingKey);
byte[] rawHmac = mac.doFinal(data.getBytes());
result = (new BASE64Encoder()).encode(rawHmac);
} catch (Exception e) {
throw new Error("Failed to generate HMAC : " + e.getMessage());
}
return result;
}

/*
* 等同于javaScript中的 new Date().toUTCString();
*/
public static String toGMTString(Date date) {
SimpleDateFormat df = new SimpleDateFormat("E, dd MMM yyyy HH:mm:ss z", Locale.UK);
df.setTimeZone(new java.util.SimpleTimeZone(0, "GMT"));
return df.format(date);
}

/*
* 发送POST请求
*/
public static String sendPost(String url, String body, String ak_id, String ak_secret) {
PrintWriter out = null;
BufferedReader in = null;
String result = "";
try {
URL realUrl = new URL(url);

/*
* http header 参数
*/
String method = "POST";
String accept = "application/json";
String content_type = "application/json";
String path = realUrl.getFile();
String date = toGMTString(new Date());
```

```
// 1.对body做MD5+BASE64加密
String bodyMd5 = MD5Base64(body);
String stringToSign = method + "\n" + accept + "\n" + bodyMd5 + "\n" + content_type + "\n" + date ;
// 2.计算 HMAC-SHA1
String signature = HMACSha1(stringToSign, ak_secret);
// 3.得到 authorization header
String authHeader = "Dataplus " + ak_id + ":" + signature;

// 打开和URL之间的连接
URLConnection conn = realUrl.openConnection();
// 设置通用的请求属性
conn.setRequestProperty("accept", accept);
conn.setRequestProperty("content-type", content_type);
conn.setRequestProperty("date", date);
conn.setRequestProperty("Authorization", authHeader);
// 发送POST请求必须设置如下两行
conn.setDoOutput(true);
conn.setDoInput(true);
// 获取URLConnection对象对应的输出流
out = new PrintWriter(conn.getOutputStream());
// 发送请求参数
out.print(body);
// flush输出流的缓冲
out.flush();
// 定义BufferedReader输入流来读取URL的响应
in = new BufferedReader(new InputStreamReader(conn.getInputStream()));
String line;
while ((line = in.readLine()) != null) {
result += line;
}
} catch (Exception e) {
System.out.println("发送 POST 请求出现异常！" + e);
e.printStackTrace();
}
// 使用finally块来关闭输出流、输入流
finally {
try {
if (out != null) {
out.close();
}
if (in != null) {
in.close();
}
} catch (IOException ex) {
ex.printStackTrace();
}
}
return result;
}


/*
* GET请求
*/

public static String sendGet(String url, String task_id,String ak_id, String ak_secret) {
```

```
String result = "";
BufferedReader in = null;
try {
URL realUrl = new URL(url+"/"+task_id);
/*
* http header 参数
*/
String method = "GET";
String accept = "application/json";
String content_type = "application/json";
String path = realUrl.getFile();
String date = toGMTString(new Date());
// 1.对body做MD5+BASE64加密
//String bodyMd5 = MD5Base64("");
String stringToSign = method + "\n" + accept + "\n" + "" + "\n" + content_type + "\n" + date;
// 2.计算 HMAC-SHA1
String signature = HMACSha1(stringToSign, ak_secret);
// 3.得到 authorization header
String authHeader = "Dataplus " + ak_id + ":" + signature;
// 打开和URL之间的连接
URLConnection connection = realUrl.openConnection();
// 设置通用的请求属性
connection.setRequestProperty("accept", accept);
connection.setRequestProperty("content-type", content_type);
connection.setRequestProperty("date", date);
connection.setRequestProperty("Authorization", authHeader);
// 建立实际的连接
connection.connect();
// 定义 BufferedReader输入流来读取URL的响应
in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
String line;
while ((line = in.readLine()) != null) {
result += line;
}
} catch (Exception e) {
System.out.println("发送GET请求出现异常！" + e);
e.printStackTrace();
}
// 使用finally块来关闭输入流
finally {
try {
if (in != null) {
in.close();
}
} catch (Exception e) {
e.printStackTrace();
}
}
return result;
}
}
```