ALIBABA CLOUD

Alibaba Cloud

智能语音交互
实时语音识别

文档版本：20201103

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.接口说明

对长时间的语音数据流进行识别，适用于会议演讲、视频直播等长时间不间断识别的场景。

使用须知

- 支持的输入格式：PCM编码、16bit采样位数、单声道（mono）。
- 支持的音频采样率：8000Hz/16000Hz。
- 支持设置返回结果：是否返回中间识别结果，在后处理中添加标点，将中文数字转为阿拉伯数字输出。
- 支持设置多语言识别：在控制台编辑项目中进行模型选择，详情请参见管理项目。

服务地址

访问类型	说明	URL
外网访问	所有服务器均可使用外网访问URL（SDK中默认设置了外网访问URL）。	wss://nls-gateway.cn-shanghai.aliyuncs.com/ws/v1
上海ECS内网访问	使用阿里云上海ECS（即ECS地域为华东2（上海）），可使用内网访问URL。ECS的经典网络不能访问AnyTunnel，即不能在内网访问语音服务；如果希望使用AnyTunnel，需要创建专有网络在其内部访问。 **说明** • 使用内网访问方式，将不产生ECS实例的公网流量费用。 • 关于ECS的网络类型请参见网络类型。	ws://nls-gateway.cn-shanghai-internal.aliyuncs.com:80/ws/v1

交互流程

说明

所有服务端的响应都会在返回信息的header包含用于表示本次识别任务的task_id参数，请记录下该值，如果发生错误，请将task_id和错误信息提交到工单。

1. 鉴权

客户端与服务端建立WebSocket连接时，使用Token进行鉴权。关于Token获取请参见获取Token。

2. 开始识别

客户端发起请求，服务端确认请求有效。其中在请求消息中需要进行参数设置，各参数由SDK中SpeechTranscriber对象的set方法设置，各参数含义如下：

参数	类型	是否必选	说明
appkey	String	是	管控台创建的项目appkey。
format	String	否	音频编码格式，默认是PCM（无压缩的PCM文件或WAV文件），16bit采样位数的单声道。
sample_rate	Integer	否	音频采样率，默认是16000Hz，根据音频采样率在管控台对应项目中配置支持该采样率及场景的模型。
enable_intermediate_result	Boolean	否	是否返回中间识别结果，默认是False。
enable_punctuation_prediction	Boolean	否	是否在后处理中添加标点，默认是False。
enable_inverse_text_normalization	Boolean	否	是否在后处理中执行ITN，设置为true时，中文数字将转为阿拉伯数字输出，默认是False。 **说明** 不对词信息进行ITN转换。
customization_id	String	否	自学习模型ID。
vocabulary_id	String	否	定制泛热词ID。

参数	类型	是否必选	说明
max_sentence_silence	Integer	否	语音断句检测阈值，静音时长超过该阈值会被认为断句，参数范围200ms～2000ms，默认值800ms。
enable_words	Boolean	否	是否开启返回词信息，默认是False。
enable_ignore_sentence_timeout	Boolean	否	是否忽略实时识别中的单句识别超时，默认是False。
disfluency	Boolean	否	是否对识别文本进行顺滑（去除语气词，重复说等），默认是False。
vad_model	String	否	服务端的vad模型id，默认无需设置。
speech_noise_threshold	float	否	噪音参数阈值，参数范围：[-1,1]。取值说明如下： • 取值越趋于-1，噪音被判定为语音的概率越大。 • 取值越趋于+1，语音被判定为噪音的概率越大。 **说明** 该参数属高级参数，调整需慎重并重点测试。

3. 接收识别结果

客户端循环发送语音数据，持续接收识别结果：

- SentenceBegin事件表示服务端检测到了一句话的开始。实时语音识别服务的智能断句功能会判断出一句话的开始与结束，举例如下：

```
{
 "header": {
  "namespace": "SpeechTranscriber",
  "name": "SentenceBegin",
  "status": 20000000,
  "message_id": "a426f3d4618447519c9d85d1a0d1****",
  "task_id": "5ec521b5aa104e3abccf3d361822****",
  "status_text": "Gateway:SUCCESS:Success."
 },
 "payload": {
  "index": 1,
  "time": 0
 }
}
```

header对象参数说明：

参数	类型	说明
namespace	String	消息所属的命名空间。
name	String	消息名称，SentenceBegin表示一个句子的开始。
status	Integer	状态码，表示请求是否成功，见服务状态码。
status_text	String	状态消息。
task_id	String	任务全局唯一ID，请记录该值，便于排查问题。
message_id	String	本次消息的ID。

payload对象参数说明：

参数	类型	说明
index	Integer	句子编号，从1开始递增。
time	Integer	当前已处理的音频时长，单位是毫秒。

- TranscriptionResultChanged事件表示识别结果发生了变化。仅当enable_intermediate_result取值为true时会多次返回此消息，即一句话的中间识别结果，举例如下：

```
{
 "header": {
  "namespace": "SpeechTranscriber",
  "name": "TranscriptionResultChanged",
  "status": 20000000,
  "message_id": "dc21193fada84380a3b6137875ab****",
  "task_id": "5ec521b5aa104e3abccf3d361822****",
  "status_text": "Gateway:SUCCESS:Success."
 },
 "payload": {
  "index": 1,
  "time": 1835,
  "result": "北京的天",
  "confidence": 1.0,
  "words": [{
   "text": "北京",
   "startTime": 630,
   "endTime": 930
  }, {
   "text": "的",
   "startTime": 930,
   "endTime": 1110
  }, {
   "text": "天",
   "startTime": 1110,
   "endTime": 1140
  }]
 }
}
```

header对象参数同上述表格说明，name为TranscriptionResultChanged：表示句子的中间识别结果。

payload对象参数说明：

参数	类型	说明
index	Integer	句子编号，从1开始递增。
time	Integer	当前已处理的音频时长，单位是毫秒。
result	String	当前句子的识别结果。
words	List< Word >	当前句子的词信息，需要将enable_words设置为true。
confidence	Double	当前句子识别结果的置信度，取值范围：[0.0,1.0]。值越大表示置信度越高。

- SentenceEnd事件表示服务端检测到了一句话的结束，并附带返回该句话的识别结果，举例如下：

```
{
 "header": {
  "namespace": "SpeechTranscriber",
  "name": "SentenceEnd",
  "status": 20000000,
  "message_id": "c3a9ae4b231649d5ae05d4af36fd****",
  "task_id": "5ec521b5aa104e3abccf3d361822****",
  "status_text": "Gateway:SUCCESS:Success."
 },
 "payload": {
  "index": 1,
  "time": 1820,
  "begin_time": 0,
  "result": "北京的天气。",
  "confidence": 1.0,
  "words": [{
   "text": "北京",
   "startTime": 630,
   "endTime": 930
  }, {
   "text": "的",
   "startTime": 930,
   "endTime": 1110
  }, {
   "text": "天气",
   "startTime": 1110,
   "endTime": 1380
  }]
 }
}
```


header对象参数同上述表格说明，name为SentenceEnd表示识别到句子的结束。

payload对象参数说明：

参数	类型	说明
index	Integer	句子编号，从1开始递增。
time	Integer	当前已处理的音频时长，单位是毫秒。

参数	类型	说明
begin_time	Integer	当前句子对应的SentenceBegin事件的时间，单位是毫秒。
result	String	当前的识别结果。
words	List< Word >	当前句子的词信息，需要将enable_words设置为true。
confidence	Double	当前句子识别结果的置信度，取值范围：[0.0,1.0]。值越大表示置信度越高。

Word对象参数说明：

参数	类型	说明
text	String	文本。
startTime	Integer	词开始时间，单位为毫秒。
endTime	Integer	词结束时间，单位为毫秒。

4. 结束识别

通知服务端语音数据发送完成，服务端识别结束后通知客户端识别完毕。

服务状态码

在服务的每一次响应中，都包含status字段，即服务状态码，此处列举通用错误码、网关错误码、配置错误码表格，如下表所示。

通用错误码：

错误码	原因	解决办法
40000001	身份认证失败	检查使用的令牌是否正确，是否过期。
40000002	无效的消息	检查发送的消息是否符合要求。

错误码	原因	解决办法
403	令牌过期或无效的参数	首先检查使用的令牌是否过期，然后检查参数值设置是否合理。
40000004	空闲超时	确认是否长时间（10秒）没有发送数据到服务端。
40000005	请求数量过多	检查是否超过了并发连接数或者每秒钟请求数。如果超过并发数，建议从免费版升级到商用版，或者商用版扩容并发资源。
40000000	默认的客户端错误码	查看错误消息或提交工单。
41010120	客户端超时错误	客户端连续10秒及以上没有发送数据，导致客户端超时错误。
50000000	默认的服务端错误	如果偶现可以忽略，重复出现请提交工单。
50000001	内部调用错误	如果偶现可以忽略，重复出现请提交工单。
52010001	内部调用错误	如果偶现可以忽略，重复出现请提交工单。

网关错误：

错误码	原因	解决办法
40010001	不支持的接口	使用了不支持的接口，如果使用SDK请提交工单。
40010002	不支持的指令	使用了不支持的指令，如果使用SDK请提交工单。
40010003	无效的指令	指令格式错误，如果使用SDK请提交工单。

错误码	原因	解决办法
40010004	客户端提前断开连接	检查是否在请求正常完成之前关闭了连接。
40010005	任务状态错误	发送了当前任务状态不能处理的指令。

配置错误：

错误码	原因	解决办法
40020105	应用不存在	解析路由时找不到应用。
40020106	appkey和token不匹配	检查应用apkey是否正确，是否与令牌归属同一个账号。
40020503	子账户鉴权失败	使用父账户对调用的子账户授权POP API的访问权限。

实时语音识别：

错误码	原因	解决办法
41040201	客户端10s内停止发送数据	检查网络问题，或者检查业务中是否存在不发数据的情况。
41040202	客户端发送数据过快，服务器资源已经耗尽	检测客户端发包是否过快，是否按照1:1的实时率来发包。
41040203	客户端发送音频格式不正确	请将音频数据的格式转换为SDK目前支持的音频格式发包。
41040204	客户端调用方法异常	客户端应该先调用发送请求接口，在发送请求完毕后再调用其他接口。
41040205	客户端设置MAXSILENCE_PARAM方法异常	参数MAXSILENCE_PARAM的取值范围200～2000。

错误码	原因	解决办法
41050008	采样率不匹配	检查调用时设置的采样率和管控台上appkey绑定的ASR模型采样率是否一致。
51040101	服务端内部错误	未知错误。
51040103	实时语音识别服务不可用	查看实时语音识别服务是否有任务堆积等导致任务提交失败。
51040104	请求实时语音识别服务超时	排查实时语音识别日志。
51040105	调用实时语音识别服务失败	检查实时语音识别服务是否启动，端口是否正常开启。
51040106	实时语音识别服务负载均衡失败，未获取到实时语音识别服务的IP地址	检查VPC中的实时语音识别服务机器是否有异常。
51070103	后处理服务参数配置错误	若使用的模型为非电话通用行业英语模型，请通过管控台再次选择模型刷新服务端参数。若仍无法解决问题，请提交工单。

2.Java SDK

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

前提条件

- 在使用SDK之前，请先阅读接口说明，详情请参见接口说明。
- 从2.1.0版本开始原有nls-sdk-long-asr更名为nls-sdk-transcriber。升级时需确认已删除nls-sdk-long-asr，并按编译提示添加相应回调方法。

下载安装

从maven服务器下载最新版本SDK，下载Demo源码ZIP包。

```
<dependency>
    <groupId>com.alibaba.nls</groupId>
    <artifactId>nls-sdk-transcriber</artifactId>
    <version>2.1.6</version>
</dependency>
```

Demo解压后，在pom目录运行 `mvn package` ，会在target目录生成可执行JAR：nls-example-transcriber-2.0.0-jar-with-dependencies.jar，将JAR包拷贝到目标服务器，用于快速验证及压测服务。

服务验证：

运行如下代码，并按提示提供相应参数。

运行后在命令执行目录生成logs/nls.log。

```
java -cp nls-example-transcriber-2.0.0-jar-with-dependencies.jar com.alibaba.nls.client.SpeechTranscriberDemo
```

服务压测：

运行如下代码，并按提示提供相应参数。

其中阿里云服务url参数为： `wss://nls-gateway.cn-shanghai.aliyuncs.com/ws/v1` ，语音文件为16k采样率PCM格式文件，并发数根据您的购买情况进行选择。

```
java -jar nls-example-transcriber-2.0.0-jar-with-dependencies.jar
```

> **说明**
>
> 自行压测超过2并发将产生费用。

关键接口

- 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。

- SpeechTranscriber：实时语音识别类，通过该接口设置请求参数，发送请求及声音数据。非线程安全。
- SpeechTranscriberListener：实时语音识别结果监听类，监听识别结果。非线程安全。

更多介绍，请参见Java API接口说明。

> **注意**
>
> SDK调用注意事项：
>
> - NlsClient使用了Netty框架，NlsClient对象的创建会消耗一定时间和资源，一经创建可以重复使用。建议调用程序将NlsClient的创建和关闭与程序本身的生命周期相结合。
> - SpeechTranscriber对象不可重复使用，一个识别任务对应一个SpeechTranscriber对象。例如，N个音频文件要进行N次识别任务，创建N个SpeechTranscriber对象。
> - SpeechTranscriberListener对象和SpeechTranscriber对象是一一对应的，不能在不同SpeechTranscriber对象使用同一个SpeechTranscriberListener对象，否则不能将各识别任务区分开。
> - Java SDK依赖Netty网络库，如果您的应用依赖Netty，其版本需更新至4.1.17.Final及以上。

示例代码

> **说明**
>
> - 下载nls-sample-16k.wav。
>
>   示例中使用的音频文件为16000Hz采样率，请在管控台中将appkey对应项目的模型设置为**通用**模型，以获取正确的识别结果；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。
> - 示例中使用了SDK内置的默认外网访问服务端URL，如果您需要使用阿里云上海ECS内网访问服务端URL，则在创建NlsClient对象时，设置内网访问的URL：
>
>   ```
>   client = new NlsClient("ws://nls-gateway.cn-shanghai-internal.aliyuncs.com/ws/v1", accessToken);
>   ```

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import com.alibaba.nls.client.protocol.InputFormatEnum;
import com.alibaba.nls.client.protocol.NlsClient;
import com.alibaba.nls.client.protocol.SampleRateEnum;
import com.alibaba.nls.client.protocol.asr.SpeechTranscriber;
import com.alibaba.nls.client.protocol.asr.SpeechTranscriberListener;
import com.alibaba.nls.client.protocol.asr.SpeechTranscriberResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
/**
 * 此示例演示了：
```

```
 * ASR实时识别API调用。
 * 动态获取token。
 * 通过本地模拟实时流发送。
 * 识别耗时计算。
 */
public class SpeechTranscriberDemo {
  private String appKey;
  private NlsClient client;
  private static final Logger logger = LoggerFactory.getLogger(SpeechTranscriberDemo.class);

  public SpeechTranscriberDemo(String appKey, String id, String secret, String url) {
    this.appKey = appKey;
    //应用全局创建一个NlsClient实例，默认服务地址为阿里云线上服务地址。
    //获取token，实际使用时注意在accessToken.getExpireTime()过期前再次获取。
    AccessToken accessToken = new AccessToken(id, secret);
    try {
      accessToken.apply();
      System.out.println("get token: " + ", expire time: " + accessToken.getExpireTime());
      if(url.isEmpty()) {
        client = new NlsClient(accessToken.getToken());
      }else {
        client = new NlsClient(url, accessToken.getToken());
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  private static SpeechTranscriberListener getTranscriberListener() {
    SpeechTranscriberListener listener = new SpeechTranscriberListener() {
      //识别出中间结果。仅当setEnableIntermediateResult为true时，才会返回该消息。
      @Override
      public void onTranscriptionResultChange(SpeechTranscriberResponse response) {
        System.out.println("task_id: " + response.getTaskId() +
          ", name: " + response.getName() +
          //状态码“20000000”表示正常识别。
          ", status: " + response.getStatus() +
          //句子编号，从1开始递增。
          ", index: " + response.getTransSentenceIndex() +
          //当前的识别结果。
```

```
            ", result: " + response.getTransSentenceText() +
            //当前已处理的音频时长，单位为毫秒。
            ", time: " + response.getTransSentenceTime());
    }

    @Override
    public void onTranscriberStart(SpeechTranscriberResponse response) {
      //task_id是调用方和服务端通信的唯一标识，遇到问题时，需要提供此task_id。
      System.out.println("task_id: " + response.getTaskId() + ", name: " + response.getName() + ", status: "
+ response.getStatus());
    }

    @Override
    public void onSentenceBegin(SpeechTranscriberResponse response) {
      System.out.println("task_id: " + response.getTaskId() + ", name: " + response.getName() + ", status: "
+ response.getStatus());

    }

    //识别出一句话。服务端会智能断句，当识别到一句话结束时会返回此消息。
    @Override
    public void onSentenceEnd(SpeechTranscriberResponse response) {
      System.out.println("task_id: " + response.getTaskId() +
          ", name: " + response.getName() +
          //状态码“20000000”表示正常识别。
          ", status: " + response.getStatus() +
          //句子编号，从1开始递增。
          ", index: " + response.getTransSentenceIndex() +
          //当前的识别结果。
          ", result: " + response.getTransSentenceText() +
          //置信度
          ", confidence: " + response.getConfidence() +
          //开始时间
          ", begin_time: " + response.getSentenceBeginTime() +
          //当前已处理的音频时长，单位为毫秒。
          ", time: " + response.getTransSentenceTime());
    }

    //识别完毕
    @Override
    public void onTranscriptionComplete(SpeechTranscriberResponse response) {
```

```
            System.out.println("task_id: " + response.getTaskId() + ", name: " + response.getName() + ", status: "
+ response.getStatus());
        }

        @Override
        public void onFail(SpeechTranscriberResponse response) {
          //task_id是调用方和服务端通信的唯一标识，遇到问题时，需要提供此task_id。
          System.out.println("task_id: " + response.getTaskId() +  ", status: " + response.getStatus() + ", status
_text: " + response.getStatusText());
        }
      };

      return listener;
    }

    //根据二进制数据大小计算对应的同等语音长度。
    //sampleRate：支持8000或16000。
    public static int getSleepDelta(int dataSize, int sampleRate) {
      // 仅支持16位采样。
      int sampleBytes = 16;
      // 仅支持单通道。
      int soundChannel = 1;
      return (dataSize * 10 * 8000) / (160 * sampleRate);
    }

    public void process(String filepath) {
      SpeechTranscriber transcriber = null;
      try {
        //创建实例、建立连接。
        transcriber = new SpeechTranscriber(client, getTranscriberListener());
        transcriber.setAppKey(appKey);
        //输入音频编码方式。
        transcriber.setFormat(InputFormatEnum.PCM);
        //输入音频采样率。
        transcriber.setSampleRate(SampleRateEnum.SAMPLE_RATE_16K);
        //是否返回中间识别结果。
        transcriber.setEnableIntermediateResult(false);
        //是否生成并返回标点符号。
        transcriber.setEnablePunctuation(true);
        //是否将返回结果规整化，比如将一百返回为100。
        transcriber.setEnableITN(false);
```

```
//设置vad断句参数。默认值：800ms，有效值：200ms～2000ms。
//transcriber.addCustomedParam("max_sentence_silence", 600);
//设置是否语义断句。
//transcriber.addCustomedParam("enable_semantic_sentence_detection",false);
//设置是否开启顺滑。
//transcriber.addCustomedParam("disfluency",true);
//设置是否开启词模式。
//transcriber.addCustomedParam("enable_words",true);
//设置vad噪音阈值参数，参数取值为-1～+1，如-0.9、-0.8、0.2、0.9。
//取值越趋于-1，判定为语音的概率越大，亦即有可能更多噪声被当成语音被误识别。
//取值越趋于+1，判定为噪音的越多，亦即有可能更多语音段被当成噪音被拒绝识别。
//该参数属高级参数，调整需慎重和重点测试。
//transcriber.addCustomedParam("speech_noise_threshold",0.3);
//设置训练后的定制语言模型id。
//transcriber.addCustomedParam("customization_id","你的定制语言模型id");
//设置训练后的定制热词id。
//transcriber.addCustomedParam("vocabulary_id","你的定制热词id");
//设置是否忽略单句超时。
transcriber.addCustomedParam("enable_ignore_sentence_timeout",false);
//vad断句开启后处理。
//transcriber.addCustomedParam("enable_vad_unify_post",false);

//此方法将以上参数设置序列化为JSON发送给服务端，并等待服务端确认。
transcriber.start();

File file = new File(filepath);
FileInputStream fis = new FileInputStream(file);
byte[] b = new byte[3200];
int len;
while ((len = fis.read(b)) > 0) {
  logger.info("send data pack length: " + len);
  transcriber.send(b, len);
  //本案例用读取本地文件的形式模拟实时获取语音流并发送的，因为读取速度较快，这里需要设置sleep。
  //如果实时获取语音则无需设置sleep, 如果是8k采样率语音第二个参数设置为8000。
  int deltaSleep = getSleepDelta(len, 16000);
  Thread.sleep(deltaSleep);
}

//通知服务端语音数据发送完毕，等待服务端处理完成。
```

```
            long now = System.currentTimeMillis();
            logger.info("ASR wait for complete");
            transcriber.stop();
            logger.info("ASR latency : " + (System.currentTimeMillis() - now) + " ms");
        } catch (Exception e) {
            System.err.println(e.getMessage());
        } finally {
            if (null != transcriber) {
                transcriber.close();
            }
        }
    }

    public void shutdown() {
        client.shutdown();
    }

    public static void main(String[] args) throws Exception {
        String appKey = "填写appkey";
        String id = "填写AccessKey Id";
        String secret = "填写AccessKey Secret";
        String url = ""; // 默认值：wss://nls-gateway.cn-shanghai.aliyuncs.com/ws/v1。

        if (args.length == 3) {
            appKey  = args[0];
            id      = args[1];
            secret  = args[2];
        } else if (args.length == 4) {
            appKey  = args[0];
            id      = args[1];
            secret  = args[2];
            url     = args[3];
        } else {
            System.err.println("run error, need params(url is optional): " + "<app-key> <AccessKeyId> <AccessKeyS
ecret> [url]");
            System.exit(-1);
        }
        //本案例使用本地文件模拟发送实时流数据。您在实际使用时，可以实时采集或接收语音流并发送到ASR服务端。
        String filepath = "nls-sample-16k.wav";
        SpeechTranscriberDemo demo = new SpeechTranscriberDemo(appKey, id, secret, url);
        demo.process(filepath);
```

```
        demo.shutdown();
    }
}
```

3.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。
10000010	please check if the memory is enough.	内存不足，请检查本地机器内存。

错误码	错误描述	解决方案
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，请参见服务状态码。

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。
> - 示例中使用了SDK内置的默认实时语音识别服务的外网访问服务端URL，如果您使用阿里云上海ECS且需要使用内网访问服务端URL，则在创建SpeechRecognizerRequest的对象中设置内网访问的URL。
>
>   ```
>   request->setUrl("ws://nls-gateway.cn-shanghai-internal.aliyuncs.com/ws/v1");
>   ```
>
> - 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->get
TaskId());
  // 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->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。
        );
  // 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, result=%s\n", c
bEvent->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, result=%s\n", c
bEvent->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->getStatusCode(), cbEv
ent->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
```

```
  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，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800ms。
  //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 firewalld\n");
    NlsClient::getInstance()->releaseTranscriberRequest(request); // start()失败，释放request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
    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 AccessKey-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 AccessKey-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;
}
```

4.Python SDK

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

前提条件

使用SDK前，请先阅读接口说明，详情请参见接口说明。

下载安装

> 说明
>
> - SDK仅支持Python3.4及以上版本，暂不支持Python 2。
> - 确认已安装Python包管理工具setuptools，如果没有安装，请在Linux终端使用以下命令安装：
>
> ```
> pip install setuptools
> ```

1. 下载Python SDK。
2. 安装SDK。

   在Linux终端的SDK目录下，执行以下命令：

   ```
   # 打包
   python setup.py bdist_egg
   # 安装
   python setup.py install
   ```

关键接口

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。

- SpeechTranscriber：实时语音识别类，通过该接口设置请求参数、发送请求及语音数据。非线程安全。
  - start方法：建立与服务端的连接。默认参数 `ping_interval` 表示自动发送 `ping` 命令的间隔， `ping_timeout` 表示等待 `ping` 命令响应的 `pong` 消息的超时时间，需要满足 `ping_interval` 大于 `ping_timeout` 。
  - send方法：发送语音数据到服务端。
  - stop方法：结束识别并关闭与服务端的连接。
  - close方法：关闭与服务端的网络链接。
- SpeechTranscriberCallback：回调事件函数集合类，语音结果、异常等回调的统一入口。
  - on_started方法：客户端与服务端建立连接成功的回调。
  - on_result_changed方法：客户端接收到中间识别结果的回调。
  - on_sentence_begin方法：客户端接收到一句话开始的回调。
  - on_sentence_end方法：客户端接收到一句话结束的回调。

- on_completed方法：客户端接收到识别结束的回调。
- on_task_failed方法：客户端接收到异常错误的回调。
- on_channel_closed方法：客户端接收到断开网络连接成功的回调。

SDK调用注意事项

- NlsClient对象创建一次可以重复使用。
- 一个识别任务对应一个SpeechTranscriber对象。例如，有N个音频文件，则要进行N次识别任务，创建N个SpeechTranscriber对象。
- SpeechTranscriberCallback对象和SpeechTranscriber对象是一一对应的，不能将一个实现的SpeechTranscriberCallback对象设置到多个SpeechTranscriber对象中，否则无法将各识别任务区分开。

示例代码

> 说明
>
> - 下载nls-sample-16k.wav。
>
> 示例使用的音频文件为16000Hz采样率，请在智能语音交互管控台中将appkey对应项目的模型设置为**通用**模型，以获取正确的识别结果；如果使用其他音频，请设置为支持该音频场景的模型，模型设置请参见管理项目。
> - 示例使用SDK内置的默认外网访问服务端URL，如果您使用阿里云上海ECS并需要使用内网访问服务端URL，则在创建NlsClient对象时，设置内网访问的URL：
>
>  ```
>  transcriber = client.create_transcriber(callback, "ws://nls-gateway.cn-shanghai-internal.aliyuncs.com/ws/v1")
>  ```

```
# -*- coding: utf-8 -*-
import os
import time
import threading
import ali_speech
from ali_speech.callbacks import SpeechTranscriberCallback
from ali_speech.constant import ASRFormat
from ali_speech.constant import ASRSampleRate

class MyCallback(SpeechTranscriberCallback):
    """
    构造函数的参数没有要求，可根据需要设置添加。
    示例中的name参数可作为待识别的音频文件名，用于在多线程中进行区分。
    """
    def __init__(self, name='default'):
        self._name = name

    def on_started(self, message):
```

```
    def on_started(self, message):
        print('MyCallback.OnRecognitionStarted: %s' % message)

    def on_result_changed(self, message):
        print('MyCallback.OnRecognitionResultChanged: file: %s, task_id: %s, result: %s' % (
            self._name, message['header']['task_id'], message['payload']['result']))

    def on_sentence_begin(self, message):
        print('MyCallback.on_sentence_begin: file: %s, task_id: %s, sentence_id: %s, time: %s' % (
            self._name, message['header']['task_id'], message['payload']['index'], message['payload']['time']))

    def on_sentence_end(self, message):
        print('MyCallback.on_sentence_end: file: %s, task_id: %s, sentence_id: %s, time: %s, result: %s' % (
            self._name,
            message['header']['task_id'], message['payload']['index'],
            message['payload']['time'], message['payload']['result']))

    def on_completed(self, message):
        print('MyCallback.OnRecognitionCompleted: %s' % message)

    def on_task_failed(self, message):
        print('MyCallback.OnRecognitionTaskFailed-task_id:%s, status_text:%s' % (
            message['header']['task_id'], message['header']['status_text']))

    def on_channel_closed(self):
        print('MyCallback.OnRecognitionChannelClosed')

def process(client, appkey, token):
    audio_name = 'nls-sample-16k.wav'
    callback = MyCallback(audio_name)
    transcriber = client.create_transcriber(callback)
    transcriber.set_appkey(appkey)
    transcriber.set_token(token)
    transcriber.set_format(ASRFormat.PCM)
    transcriber.set_sample_rate(ASRSampleRate.SAMPLE_RATE_16K)
    transcriber.set_enable_intermediate_result(False)
    transcriber.set_enable_punctuation_prediction(True)
    transcriber.set_enable_inverse_text_normalization(True)
// transcriber.add_payload_param("max_sentence_silence", 800)
    try:
        ret = transcriber.start()
```

```
        if ret < 0:
            return ret

        print('sending audio...')
        with open(audio_name, 'rb') as f:
            audio = f.read(3200)
            while audio:
                ret = transcriber.send(audio)
                if ret < 0:
                    break
                time.sleep(0.1)
                audio = f.read(3200)

        transcriber.stop()
    except Exception as e:
        print(e)
    finally:
        transcriber.close()


def process_multithread(client, appkey, token, number):
    thread_list = []
    for i in range(0, number):
        thread = threading.Thread(target=process, args=(client, appkey, token))
        thread_list.append(thread)
        thread.start()

    for thread in thread_list:
        thread.join()


if __name__ == "__main__":
    client = ali_speech.NlsClient()
    # 设置输出日志信息的级别：DEBUG、INFO、WARNING、ERROR。
    client.set_log_level('INFO')

    appkey = '您的Appkey'
    token = '您的Token'

    process(client, appkey, token)
```

```
# 多线程示例
# process_multithread(client, appkey, token, 2)
```

5.Android SDK

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

> 说明
>
> 建议您使用新版本Android SDK，本版本后续将不再更新。详情请参见Android SDK。

前提条件

- 首先阅读接口说明，详情请参见接口说明。
- 已在智能语音管控台创建项目并获取Appkey，详情请参见创建项目。
- 已获取智能语音服务访问令牌，详情请参见获取Token。

下载安装

1. 下载SDK和示例代码。
2. 解压ZIP包得到nls-sdk-android文件夹，即是示例代码工程，在工程app/libs目录下是AAR格式的SDK包。
3. 使用Android Studio打开此工程查看示例代码，其中实时语音识别示例有两个Activity结尾的代码文件。
   i. SpeechTransciberActivity.java演示了采集音频发送给SDK功能。
   ii. （推荐）SpeechTransciberWithRecorderActivity.java演示了如何使用SDK内置录音功能。

SDK关键接口

- 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。
- SpeechTranscriber：代表一次实时语音流识别请求，需要将录制的音频或从文件读取的音频数据，发送给SDK。
- SpeechTranscriberWithRecorder：代表一次实时语音识别请求，在SpeechTranscriber的基础上内置录音功能，调用更简便。
- SpeechTranscriberCallback：语音识别回调接口，在获得识别结果、发生错误等事件发生时会触发回调。您可参照demo在回调方法中加入需要的处理逻辑。
- SpeechTranscriberWithRecorderCallback：语音识别回调接口，在SpeechTranscriberCallback基础上增加了回调语音数据和音频音量的方法。

调用步骤

下面以SpeechTransciberActivity.java为例，为您介绍实时语音识别请求的调用步骤：

1. 创建NlsClient的实例。
2. 定义SpeechTranscriberCallback类，根据您的业务需要，处理识别结果或错误情况。
3. 调用NlsClient.createTranscriberRequest()方法得到SpeechTranscriber的实例。
4. 设置SpeechTranscriber参数，主要是Access Token和Appkey。
5. 调用SpeechTranscriber.start()方法启动与云端服务连接。
6. 采集语音并调用SpeechTranscriber.sendAudio()方法发送至云端服务。

7. 在回调中处理识别结果或错误。
8. 调用SpeechTranscriber.stop()方法结束识别。
9. 如果需要发起新的请求，请重复第3～8步。
10. 调用NlsClient.release()方法释放客户端实例。

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.util.*{*;}
```

代码示例

1. 创建识别请求。

```
// 新建识别回调类
SpeechTranscriberCallback callback = new MyCallback();
// 创建识别request
speechTranscriber = client.createTranscriberRequest(callback);
speechTranscriber.setToken("");
speechTranscriber.setAppkey("");
// 设置返回中间结果
speechTranscriber.enableIntermediateResult(true);
// 启动语音识别
int code = speechTranscriber.start();
```

2. 采集音频并发送给识别服务。

   音频数据也可以是从文件或其他来源获取。如果使用SpeechTranscriberWithRecorder接口，则SDK内部会处理录音并发送数据，此时可以省略该步骤。

```
ByteBuffer buf = ByteBuffer.allocateDirect(SAMPLES_PER_FRAME);
while(sending){
  buf.clear();
  // 采集语音
  int readBytes = mAudioRecorder.read(buf, SAMPLES_PER_FRAME);
  byte[] bytes = new byte[SAMPLES_PER_FRAME];
  buf.get(bytes, 0, SAMPLES_PER_FRAME);
  if (readBytes>0 && sending){
    // 发送语音数据到识别服务
    int code = recognizer.sendAudio(bytes, bytes.length);
    if (code < 0) {
      Log.w(TAG, "Failed to send audio!");
      break;
    }
  }
  buf.position(readBytes);
  buf.flip();
}
```

3. 处理回调结果。

```
// 识别到一个新句子开始
@Override
public void onSentenceBegin(String msg, int code)
{
  Log.i(TAG, "Sentence begin");
}
// 当前句子识别结束，得到完整的句子文本。
@Override
public void onSentenceEnd(final String msg, int code)
{
  Log.d(TAG,"OnSentenceEnd " + msg + ": " + String.valueOf(code));
}

// 识别返回中间结果，只有开启相关选项时才会回调。
@Override
public void onTranscriptionResultChanged(final String msg, int code)
{
  Log.d(TAG,"OnTranscriptionResultChanged " + msg + ": " + String.valueOf(code));
}
```

6.iOS SDK

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

> 说明
>
> 建议您使用新版本iOS SDK，本版本后续将不再更新。详情请参见iOS SDK。

前提条件

- 首先阅读接口说明，详情请参见接口说明。
- 已在智能语音管控台创建项目并获取Appkey，详情请参见创建项目。
- 已获取智能语音服务访问令牌，详情请参见获取Token。

下载安装

1. 下载iOS SDK。

   解压后，NlsDemo目录即为Demo工程目录。

2. 双击NlsDemo.xcodeproj，使用Xcode打开Demo工程。
   - 导入的iOS SDK即NlsDemo/AliyunNlsSdk.framework。SDK支持 x86_64/armv7/arm64架构。
   - 如果您要将应用程序提交发布至苹果应用商店，请您使用该版本SDK：NlsDemo-iOS/dynamic-verison-framework/AliyunNlsSdk.framework。
   - Demo代码中包含2个实时语音识别实例代码。
     - Transcriber.m 演示采集音频发送给SDK的功能。
     - TranscriberwithRecorder.m 演示内置录音发送给SDK功能。

调用步骤

> 说明
>
> 请使用Embedded Binaries方式导入SDK到工程中。

1. 导入NlsSdk中的AliyunNlsClientAdaptor.h、NlsSpeechTranscriberRequest.h以及TranscriberRequestParam.h头文件。
2. 实现NlsSpeechTranscriberRequest的NlsSpeechTranscriberDelegate回调方法。
3. 创建一个AliyunNlsClientAdaptor对象nlsClient，该对象全局只需创建一次。
4. 通过调用nlsClient对象的createTranscriberRequest方法获得一个NlsSpeechTranscriberRequest对象。该NlsSpeechTranscriberRequest对象不可重复使用，一个请求需要创建一个对象。
5. 通过TranscriberRequestParam设置参数，如Access Token、Appkey等。
6. 通过NlsSpeechTranscriberRequest的setTranscriberParams传入**步骤5**中设置的TranscriberRequestParam对象。
7. 分别调用NlsSpeechTranscriberRequest对象的start方法和stop方法，完成开始识别和结束识别操作。

8. 通过NlsSpeechTranscriberRequest对象的 `sendAudio:(NSData *)audioData length:(int)len` 方法传入识别数据。
9. 如有识别结果，则会触发**步骤3**中设置的相关回调函数，通过文本形式返回结果。

关键接口

- AliyunNlsClientAdaptor：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。
- NlsSpeechTranscriberRequest：语音识别处理的请求对象，用来完成语音识别等功能，线程安全。
- TranscriberRequestParam：语音识别相关参数。
- NlsSpeechTranscriberDelegate：定义了语音识别相关回调函数。在获得结果、遇到错误等事件发生时会触发回调。

代码调用示例

```
#import <Foundation/Foundation.h>
#import "Transcriber.h"
@interface Transcriber()<NlsSpeechTranscriberDelegate,NlsVoiceRecorderDelegate>{
  IBOutlet UITextView *textViewTranscriber;
  IBOutlet UISwitch *switchTranscriber;
  Boolean transcriberStarted;
}
@end
@implementation Transcriber
-(void)viewDidLoad {
  [super viewDidLoad];
  //1. 全局参数初始化操作
  //1.1 初始化识别客户端，将transcriberStarted状态置为false。
  _nlsClient = [[NlsClientAdaptor alloc]init];
  transcriberStarted = false;
  //1.2 初始化录音recorder工具
  _voiceRecorder = [[NlsVoiceRecorder alloc]init];
  _voiceRecorder.delegate = self;
  //1.3 初始化识别参数类
  _transRequestParam = [[TranscriberRequestParam alloc]init];
  //1.4 设置log级别
  [_nlsClient setLog:NULL logLevel:1];
}
- (IBAction)startTranscriber {
  //2. 创建请求对象和开始识别
  if(_transcriberRequest!= NULL){
    [_transcriberRequest releaseRequest];
    _transcriberRequest = NULL;
  }
```

```
    }
    //2.1 创建请求对象，设置NlsSpeechTranscriberDelegate回调。
    _transcriberRequest = [_nlsClient createTranscriberRequest];
    _transcriberRequest.delegate = self;
    //2.2 设置TranscriberRequestParam请求参数
    [_transRequestParam setFormat:@"opu"];
    [_transRequestParam setEnableIntermediateResult:true];
    [_transRequestParam setToken:@""];
    [_transRequestParam setAppkey:@""];
    //2.3 传入请求参数
    [_transcriberRequest setTranscriberParams:_transRequestParam];
    //2.4 启动录音和识别，将transcriberStarted置为true。
    [_voiceRecorder start];
    [_transcriberRequest start];
    transcriberStarted = true;
    //2.5 更新UI
    dispatch_async(dispatch_get_main_queue(), ^{
        // UI更新代码
        [self->switchTranscriber setOn:true];
        self->textViewTranscriber.text = @"start Recognize！";
    });
}
- (IBAction)stopTranscriber {
    //3 结束识别，停止录音，停止识别请求。
    [_voiceRecorder stop:true];
    [_transcriberRequest stop];
    transcriberStarted= false;
    _transcriberRequest = NULL;
    dispatch_async(dispatch_get_main_queue(), ^{
        // UI更新代码
        [self->switchTranscriber setOn:false];
    });
}
/**
 *4. NlsSpeechTranscriberDelegate接口回调方法
 */
//4.1 识别回调，本次请求失败。
- (void)OnTaskFailed:(NlsDelegateEvent)event statusCode:(NSString *)statusCode errorMessage:(NSString
*)eMsg {
    NSLog(@"OnTaskFailed, statusCode is: %@ error message： %@",statusCode,eMsg);
}
```

```
//4.2 识别回调，服务端连接关闭。
- (void)OnChannelClosed:(NlsDelegateEvent)event statusCode:(NSString *)statusCode errorMessage:(NSString *)eMsg {
    NSLog(@"OnChannelClosed, statusCode is: %@",statusCode);
}
//4.3 实时语音识别开始
- (void)OnTranscriptionStarted:(NlsDelegateEvent)event statusCode:(NSString *)statusCode errorMessage:(NSString *)eMsg {
}
//4.4 识别回调，一句话的开始。
- (void)OnSentenceBegin:(NlsDelegateEvent)event result:(NSString*)result statusCode:(NSString *)statusCode errorMessage:(NSString *)eMsg {
    dispatch_async(dispatch_get_main_queue(), ^{
        // UI更新代码
        self->textViewTranscriber.text = result;
        NSLog(@"%@", result);
    });
}
//4.5 识别回调，一句话的结束。
- (void)OnSentenceEnd:(NlsDelegateEvent)event result:(NSString*)result statusCode:(NSString *)statusCode errorMessage:(NSString *)eMsg {
    dispatch_async(dispatch_get_main_queue(), ^{
        // UI更新代码
        self->textViewTranscriber.text = result;
        NSLog(@"%@", result);
    });
}
//4.5 识别回调，一句话识别的中间结果。
- (void)OnTranscriptionResultChanged:(NlsDelegateEvent)event result:(NSString *)result statusCode:(NSString *)statusCode errorMessage:(NSString *)eMsg {
    dispatch_async(dispatch_get_main_queue(), ^{
        // UI更新代码
        self->textViewTranscriber.text = result;
        NSLog(@"%@", result);
    });
}
//4.6 识别回调，实时识别完全结束。
- (void)OnTranscriptionCompleted:(NlsDelegateEvent)event statusCode:(NSString *)statusCode errorMessage:(NSString *)eMsg {
}
/**
```

```
 *5. 录音相关回调
 */
- (void)recorderDidStart {
   NSLog(@"Did start recorder!");
}
- (void)recorderDidStop {
   NSLog(@"Did stop recorder!");
}
- (void)voiceDidFail:(NSError *)error {
   NSLog(@"Did recorder error!");
}
//5.1 录音数据回调
- (void)voiceRecorded:(NSData *)frame {
   if (_transcriberRequest != nil && transcriberStarted) {
      //录音线程回调的数据传给识别服务。
      if ([_transcriberRequest sendAudio:frame length:(short)frame.length] == -1) {
         NSLog(@"connect closed ,stop transcriberRequest!");
         [self stopTranscriber];
      }
   }
}
@end
```

7.移动端NUI SDK