AIUI 移动解决方案深度体验

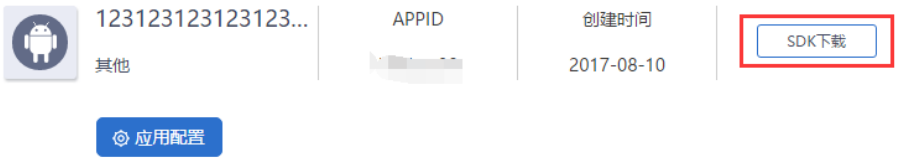
——Android

上一期和大家分享了 AIUI 移动解决方案初体验报告，基本上就演示了如何把官网的 demo 跑起来，看到效果。这一期要详细讲一下你可能还不知道的更多细节。

我们先来简单看一下 AIUI 平台业务架构图：

在 AIUI 概念面世之前，讯飞开放平台就已经有语义理解的服务了，就是【osp 开放语义平台】，讯飞内部版本号是 2.0，当时 SDK 也只有普通的 MSC SDK（就是你在讯飞开放平台 SDK 下载中心能够下载到的听写合成命令词唤醒等等语音相关的 SDK），这些 SDK 里面都有一个 SpeechUnderstander 类（Android 平台）用来访问【osp 开放语义平台】的语义服务。这里该类的 nlp_version（语义版本）参数无需设置，因为它默认就是 2.0。即上图中①的访问路径。

后来出现了【AIUI 开放平台】和相应的 AIUI SDK，使用 AIUI SDK 里面的 AIUIAgent 类的相关方法可以访问 AIUI 开放平台的语义服务。AIUI SDK 既可以在 SDK 下载中心下载，也可以在 AIUI 开放平台应用列表页下载：



这里的 AIUIAgent 之所以可以访问 AIUI 的语义服务，我们可以简单的认为它底层设置了 nlp_version=3.0 的参数，即架构图中③的访问路径。

那么问题来了，部分同学由于某些原因无法使用 AIUI SDK，只能用普通的 MSC SDK，怎么办呢？那就是上面架构图中②的访问路径。

机智的同学可以已经猜到了，对！

就是手动给 SpeechUnderstander 设置一个 nlp_version=3.0，另外还有一个 scene 参数（AIUI 的情景模式参数）：

```
    // 设置语义情景
    mSpeechUnderstander.setParameter("nlp_version", "3.0");
    mSpeechUnderstander.setParameter(SpeechConstant.SCENE, "main");
}
```

这样 MSC SDK 就可以和 AIUI SDK 一样使用 AIUI 开放平台的语义理解服务了，那么这两种方式（②和③）有什么区别吗？

当然有，但是区别在于交互形式上，而不在语义服务本身。前者是 SDK 的工作，后者是 AIUI 云端后台的工作；前者不过是因为它们用的 SDK 是不一样的，后者相同是因为它们实际访问的是同一个 AIUI 云端服务。

AIUI SDK 里面的 AIUIAgent 有个独门绝技——【连续交互】。

我们先回忆一下 SpeechUnderstander 的交互形式，语义 demo 中有一个【语音语义】的按钮，按一下按钮说一句话，进行一次交互，完了再按一下进行第二次交互。也就是说每次交互之前都需要一个特定的操作来触发识别功能，可以是按一下按钮，也可以是说一个唤醒词等等，这个就是不连续的交互。

使用过或者看过我们 AIUI 硬件评估版（AIUI 智能硬件解决方案）演示效果的同学应该知道，AIUI 可以一次唤醒后进行无数轮的语义交互，每轮之间无需再说唤醒词或者按一下什么按钮来触发。评估板上的这个优点，AIUI SDK（AIUI 移动端解决方案）很好的继承了过来，后面再介绍如何体验 AIUI SDK demo 里的这个功能。

说到这里恐怕有部分同学将之类的【连续交互】与【上下文理解】划等号了，后者的意思就是：

问：合肥明天天气怎么样？
答：合肥明天晴天，阵风 3 级……
问：后天呢？
答：合肥后天阴天，……

其实二者完全不是一个概念，如上面所说的，前者是客户端 SDK 实现的功能，后者是云端语义后台实现的功能，也就是说只要你使用的是 AIUI SDK 就有【连续交互】的功能，只要你用的是 AIUI 的语义后台就有【上下文理解】的功能。

综上，架构图中：

① 路径无【连续交互】、无【上下文】；
② 路径无【连续交互】、有【上下文】；
③ 路径有【连续交互】、有【上下文】。

下面再来看看 AIUI SDK 是如何调用 AIUI 语义服务的。

MSC SDK 的方式和老的语义调用过程类似，这里就不介绍了。

之前在《AIUI 移动解决方案初体验》中，已经报告了如何下载导入 AIUI SDK demo，还特别提到了运行 demo 之前，一定记得拷贝一个叫 meta_vad_16k.jet 的 VAD 资源文件。偷偷的告诉大家，这个东西就是 SDK 可以实现【连续交互】的关键资源。

有了它，SDK 就可以自信的说：主人你闭着眼只管说，我自己判断从哪到哪是一句话，从哪到哪是另一句话。

《初体验》中 demo 的交互其实还是单次交互模式，想要达到连续交互效果，还需要小小的改动一下 aiui_phone.cfg 配置文件：

interact_mode	交互模式，取值：oneshot（唤醒后一次交互后即休眠），continuous（默认，唤醒后可以持续交互）。

所以，我们打开 cfg 文件在 speech 节点下添加 interact_mode 参数：

```
{
    /*  交互参数  */
    "interact":{
        "interact_timeout":"60000",
        "result_timeout":"5000"
    },
    /*  全局设置  */
    "global":{
        "scene":"main",
        "clean_dialog_history":"auto"
    },
    //  本地 vad 参数
    "vad":{
        "vad_enable":"1",
        "engine_type":"meta",
        "res_type":"assets",
        "res_path":"vad/meta_vad_16k.jet"
     },
     //  识别（音频输入）参数
     "iat":{
         "sample_rate":"16000"
     },
     //  语音业务流程控制
     "speech":{
         "data_source":"sdk",
         "interact_mode":"continuous"
     }
}
```

这里多一句废话，添加 interact_mode 之前，一定记得在"data_source":"sdk"后面加一个英文半角的逗号哈⊙__⊙！不然就是一个非法的 json 结构了。

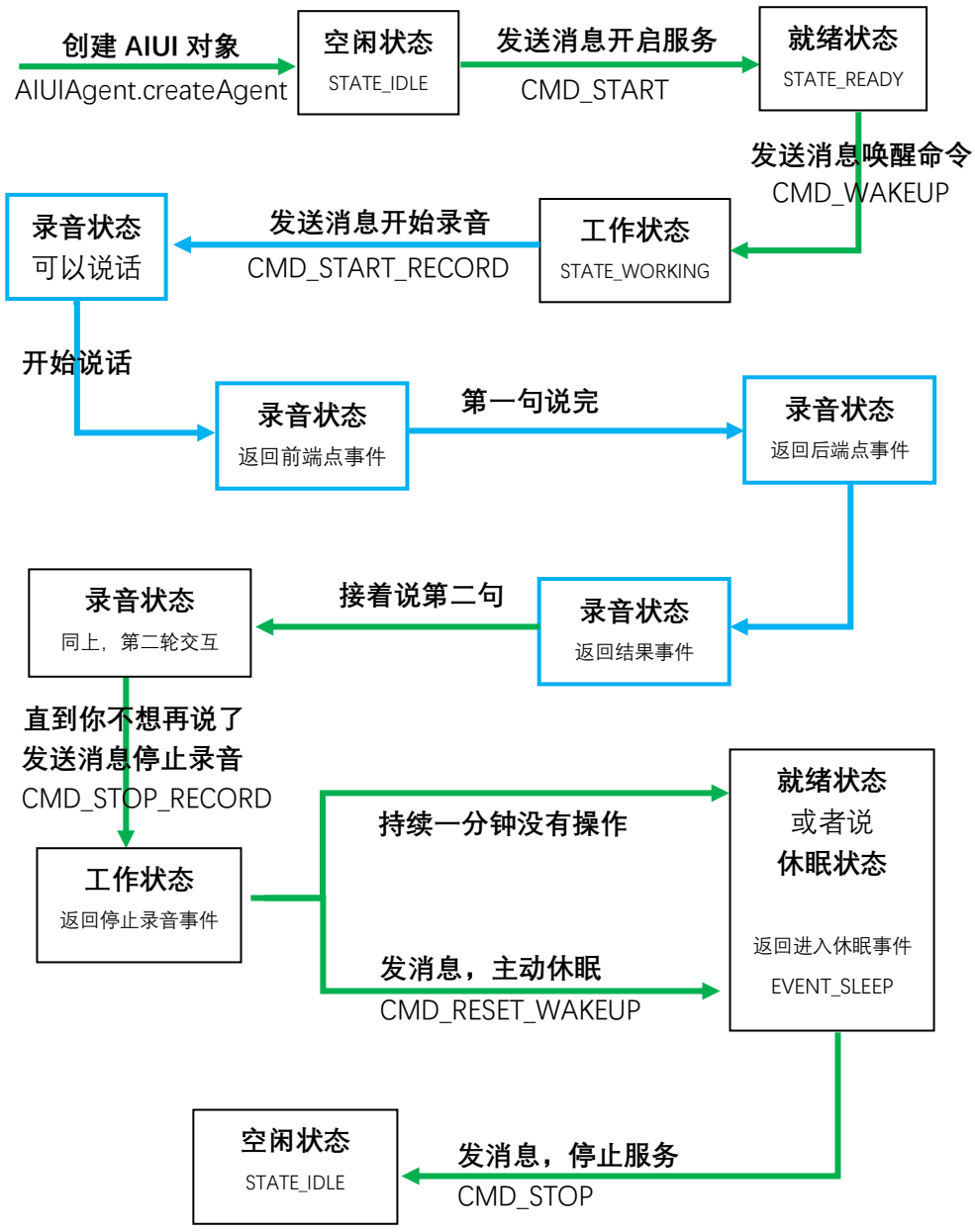
然后就可以官网的 demo 就变成连续交互模式啦。

如上图所示：

- 点击【语音语义】按钮后，启动语义识别，
- 说：明天天气怎么样
- 再说：后天呢
- 再再说：大后天呢

我们发现，整个交互过程中只在第一次说话前按了一下按钮，后面都是无需任何其他操作，直接说话就能连续识别。

看完了体验效果，下面来深入解剖一下上面的交互过程中 AIUI SDK 的工作流程：



上图中，有几个地方需要特殊说明一下：

- 蓝色部分为一轮完整的交互，后面可以接上无限个这样的交互循环下去。
- 工作状态下持续一分钟没有任何操作自动进入休眠状态，这里一分钟是可以设置的，默认是最大值一分钟，想短一点可以在 cfg 配置文件中找 interact_timeout。另外在录音状态下如果你超过一分钟没有说话也会自动进入休眠状态。

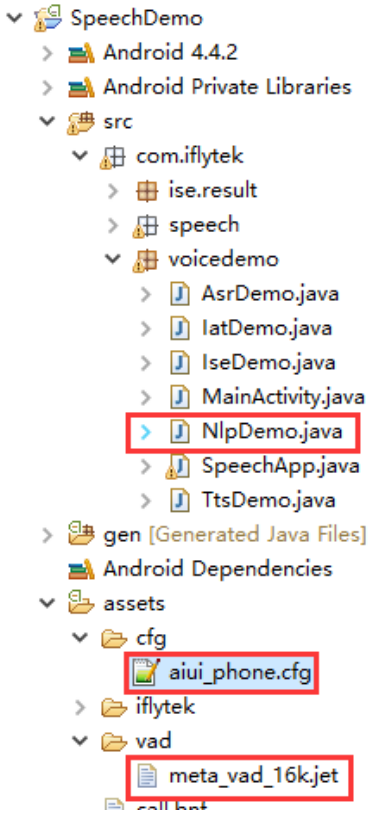
✧ 前后端点事件都属于 VAD 事件，VAD 时间还包括音量消息，AIUIEvent 中的 arg1 表示 VAD 消息类型，取值：0（前端点），1（音量消息），2（后端点）； arg2 表示音量，范围[0-30]，其在 arg1 为 1 时才有意义。
Demo 的 onEvent 回调代码中处理 VAD 事件的代码在 232 行至 240 行：

```
232                 case AIUIConstant.EVENT_VAD: {
233                     if (AIUIConstant.VAD_BOS == event.arg1) {
234                         showTip("找到vad_bos");
235                     } else if (AIUIConstant.VAD_EOS == event.arg1) {
236                         showTip("找到vad_eos");
237                     } else {
238                         showTip("" + event.arg2);
239                     }
240                 } break;
```

✧ 在一轮交互中，结果事件会回调多次，他们有的是听写结果，有的是语义结果，如果你勾选了后处理服务，还会有后处理的结果消息。想要区分它们，只需要把 event.info 打印出来看一下就明白了，info 是 json 字符串，里面有个 sub 字段，该字段值为 iat、nlp、tpp 时分别表示听写、语义、后处理结果，结果的内容在 event.data 里面，具体可以参考 Demo 中代码 NlpDemo.java 第 195 行起。

✧ 实际测试中，后端点事件回调和结果回调的先后顺序不一定想上图中那样有明确的先后关系。大部分情况下后端点事件回调会夹杂在多次结果回调之间，原因是由于基于能量（音量）的后端点检测都有滞后性（即你停止说话后过一小段时间才会触发后端点），而我们的语义识别是流式的，即一边录音一边上传音频一边拉去识别结果，这就导致了语义结果很可能会早于后端点返回。上图为了看起来简洁明朗，就简单的划成了先后关系。

上面铺垫了那么多，最后终于可以愉快的来看看代码啦！



如上图所示，和语义理解相关的文件主要就这三个，cfg 配置文件如何修改、jet 资源怎么拷贝前面已经介绍过了，这里在看一下 NlpDemo.java，看看实现代码。

```
public void onClick(View view)
{
    if( !checkAIUIAgent() ){
        return;
    }
    switch (view.getId()) {
        // 开始文本理解
        case R.id.text_nlp_start:
            startTextNlp();
            break;
        // 开始语音理解
        case R.id.nlp_start:
            startVoiceNlp();
            break;
        // 停止语音理解
        case R.id.nlp_stop:
            // AIUI 是连续会话，一次 start 后，可以连续的录音并返回结果；要停止需要调用 stop
            stopVoiceNlp();
            break;
        case R.id.update_lexicon:
            updateLexicon();
            break;
        default:
            break;
    }
}
```

第一步进入 onClick 中，点击【语音语义】按钮的时候，在 checkAIUIAgent()中初始化了 AIUIAgent 对象，然后发了一个 CMD_START 消息，此时进入就绪状态 STATE_READY。

```
private boolean checkAIUIAgent(){
  if( null == mAIUIAgent ){
    Log.i( TAG, "create aiui agent" );
    mAIUIAgent = AIUIAgent.createAgent( this, getAIUIParams(), mAIUIListener );
    AIUIMessage startMsg=new AIUIMessage(AIUIConstant.CMD_START,0,0,null,null);
    mAIUIAgent.sendMessage( startMsg );
    }
    ……
}
```

然后进入 startVoiceNlp()，先后发送了 CMD_WAKEUP 消息和 CMD_START_RECORD 消息，使 AIUI 进入工作状态 STATE_WORKING，然后进入录音状态，测试用户就可以开始说话进行交互了。

```
private void startVoiceNlp(){
  Log.i( TAG, "start voice nlp" );
  mNlpText.setText("");
  // 先发送唤醒消息，改变AIUI内部状态，只有唤醒状态才能接收语音输入
  if( AIUIConstant.STATE_WORKING != this.mAIUIState ){
    AIUIMessage wakeupMsg = new AIUIMessage(AIUIConstant.CMD_WAKEUP, 0, 0, "", null);
    mAIUIAgent.sendMessage(wakeupMsg);
  }
  // 打开AIUI内部录音机，开始录音
  String params = "sample_rate=16000,data_type=audio";
  AIUIMessage writeMsg=new AIUIMessage(AIUIConstant.CMD_START_RECORD,0,0, params,null);
  mAIUIAgent.sendMessage(writeMsg);
}
```

然后就可以从监听器里接收处理各种事件了，监听器就是 AIUIListener，细心的你一定已经发现了在 AIUIAgent 初始化的时候就传另一个 mAIUIListener ，没错就是它。他的实现代码在 185 行处，代码非常多就不贴出来了。

监听器就一个回调函数——onEvent(AIUIEvent event)，里面按照 event 的 eventType 来区分不同的事件类型分别处理：

- EVENT_ERROR：错误事件。
- EVENT_RESULT：结果事件。
- EVENT_SLEEP：休眠事件。
- EVENT_STATE：状态事件。
- EVENT_VAD：VAD 事件。
- EVENT_WAKEUP：唤醒事件。
- EVENT_START_RECORD：开始录音事件。
- EVENT_STOP_RECORD：停止录音事件。
- EVENT_CMD_RETURN：上传热词是否成功事件。

其中状态事件中的状态又分为：

- STATE_IDLE：空闲状态，AIUI 未开启。
- STATE_READY：就绪状态，等待唤醒。
- STATE_WORKING：工作状态，已唤醒，开始交互。

上述事件和状态中大部分已经在上面的 AIUI 工作流程图中有所体现了，所以这里不打算每个都详细的介绍，只选取一个最复杂的 EVENT_RESULT 事件处理为例来分析一下。

结果事件的事件实体 event 包括以下属性：

限定符和类型	字段和说明
int	arg1
int	arg2
Bundle	data
int	eventType
java.lang.String	info

arg1、arg2 为附属参数，部分事件才有，例如 VAD 事件中 arg1=1 时，arg2 表示音量；data 为事件消息的具体内容，例如 RESULT 事件中，data 就是语义结果数据；info 是

事件的说明性信息；eventType 就是前面介绍的消息的类型。

下面是一轮语义交互中返回的结果事件的 logcat 日志，分别打印了 eventType（int 类型，RESULT 类型就是 1）和 info：

```
NlpDemo          on event: 1
NlpDemo          {"data":[{"params":{"qisr_handle":"6f0d00010f90","rss":"0","sub":"iat","lrst" ⏎
                 :false,"rstid":1},"content":[{"dte":"utf8","dtf":"json","cnt_id":"0"}]}]}
NlpDemo          on event: 1
NlpDemo          {"data":[{"params":{"sub":"nlp","lrst":false,"rstid":1},"content":[{"dte":"ut ⏎
                 f8","dtf":"json","cnt_id":"0"}]}]}
NlpDemo          {"answer":{"answerType":"openQA","emotion":"default","question":{"question":" ⏎
                 你好","question_ws":"你好\/VI\/\/"},"text":"你好，又见面了真开心。","topic":321841...
                 0,"topicID":32184199507836300,"type":"T"},"man_intv":"","no_nlu_result":0,"op ⏎
                 eration":"ANSWER","rc":0,"service":"openQA","status":0,"text":"你好","uuid":"at ⏎
                 n00652751@ch46b50d1487e96f2a01","sid":"cid6f1ca71b@ch00270d1487e8010001"}
NlpDemo          on event: 1
NlpDemo          {"data":[{"params":{"sub":"tpp","lrst":false,"rstid":1},"content":[{"dte":"cu ⏎
                 stom","dtf":"custom"}]}]}
NlpDemo          on event: 12
NlpDemo          on event: 1
NlpDemo          {"data":[{"params":{"qisr_handle":"6f0d00010f90","rss":"5","sub":"iat","lrst" ⏎
                 :true,"rstid":2},"content":[{"dte":"utf8","dtf":"json","cnt id":"0"}]}]}
NlpDemo          on event: 1
NlpDemo          {"data":[{"params":{"sub":"nlp","lrst":true,"rstid":2},"content":[{"dte":"utf ⏎
                 8","dtf":"json","cnt_id":"0"}]}]}
NlpDemo          {}
NlpDemo          on event: 1
NlpDemo          {"data":[{"params":{"sub":"tpp","lrst":true,"rstid":2},"content":[{"dte":"cus ⏎
                 tom","dtf":"custom"}]}]}
```

可以清楚的看到一轮交互返回了 6 次 RESULT 结果事件，从打印出来的 info 说明信息发现他们的 sub 字段依次是 iat、nlp、tpp、iat、nlp、tpp。前文已经说明它们分别是什么意思，作为语义服务，其中我们可能最关注是 nlp 信息，的确，日志中在第一个 nlp 消息中也的确包含了语义结果数据。附上打印上述日志的代码：

```
case AIUIConstant.EVENT_RESULT: {
    Log.i( TAG,  "on event: "+ event.eventType );
    Log.i( TAG, event.info );
    try {
        JSONObject bizParamJson = new JSONObject(event.info);
        JSONObject data = bizParamJson.getJSONArray("data").getJSONObject(0);
        JSONObject params = data.getJSONObject("params");
        JSONObject content = data.getJSONArray("content").getJSONObject(0);

        if (content.has("cnt_id")) {
            String cnt_id = content.getString("cnt_id");
            JSONObject cntJson = new JSONObject(new String(event.data.getByteArray(cnt_id), "utf-8"));

            mNlpText.append( "\n" );
            mNlpText.append(cntJson.toString());

            String sub = params.optString("sub");
            if ("nlp".equals(sub)) {
                // 解析得到语义结果
                String resultStr = cntJson.optString("intent");
                Log.i( TAG, resultStr );
            }
        }
    } catch (Throwable e) {
        e.printStackTrace();
        mNlpText.append( "\n" );
        mNlpText.append( e.getLocalizedMessage() );
    }

    mNlpText.append( "\n" );
} break;
```

然后，连续交互结束后，点击【停止录音】，发送了一个 CMD_STOP_RECORD 消息，此时 AIUI 停止录音，进入工作状态。然后我们等待一分钟不进行任何操作会发现，收到一个 EVENT_SLEEP 事件，即 AIUI 自动进入了就绪状态（休眠状态）。

```
private void stopVoiceNlp(){
  Log.i( TAG, "stop voice nlp" );
  // 停止录音
  String params = "sample_rate=16000,data_type=audio";
  AIUIMessage stopWriteMsg = new AIUIMessage(AIUIConstant.CMD_STOP_RECORD, 0, 0,
params, null);
  mAIUIAgent.sendMessage(stopWriteMsg);
}
```

最后，在我们退出应用，效果 AIUIAgent 对象之前，发送了一个 CMD_STOP 消息，此时,AIUI 进入 STATE_IDLE 空闲状态。

```
protected void onDestroy() {
  super.onDestroy();
  if( null != this.mAIUIAgent ){
    AIUIMessage stopMsg = new AIUIMessage(AIUIConstant.CMD_STOP,0,0, null, null);
    mAIUIAgent.sendMessage( stopMsg );
    this.mAIUIAgent.destroy();
    this.mAIUIAgent = null;
   }
}
```

至此，Demo 完成了一个 AIUI 服务的完整的生命周期。