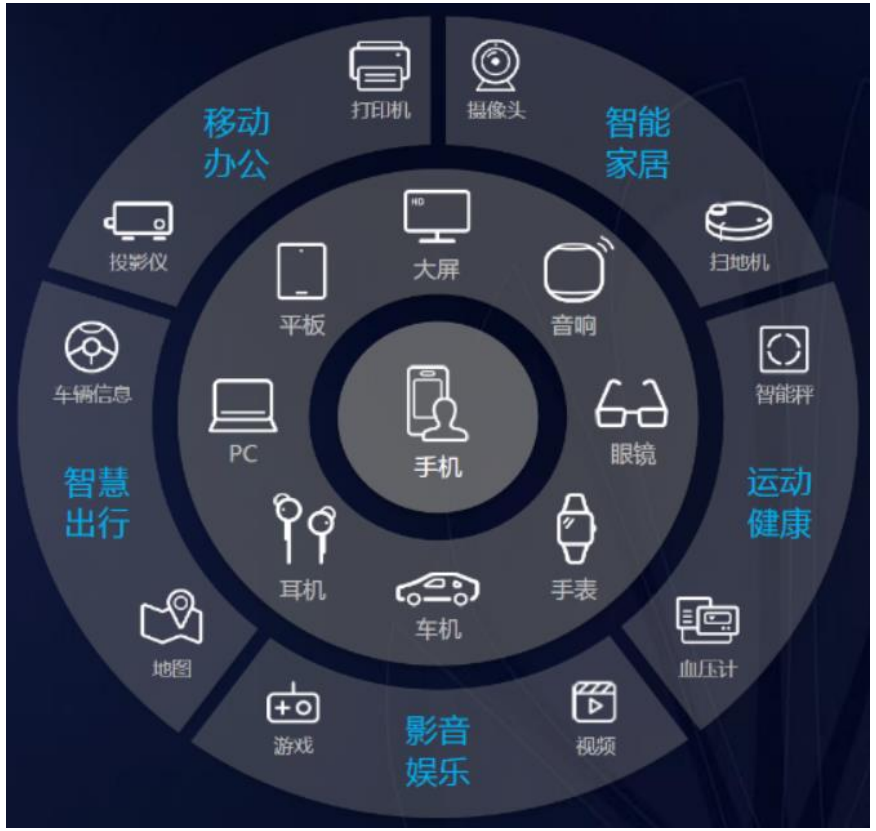
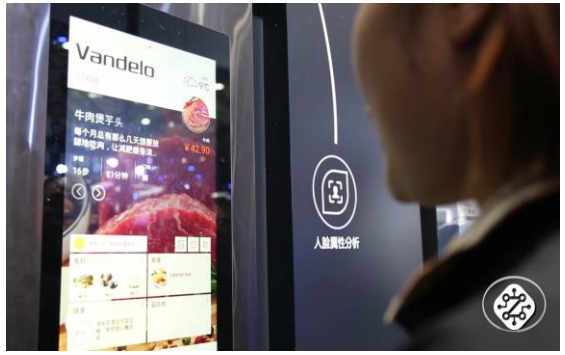
工业应用实践

洪青阳

纲要

15.1 应用场景



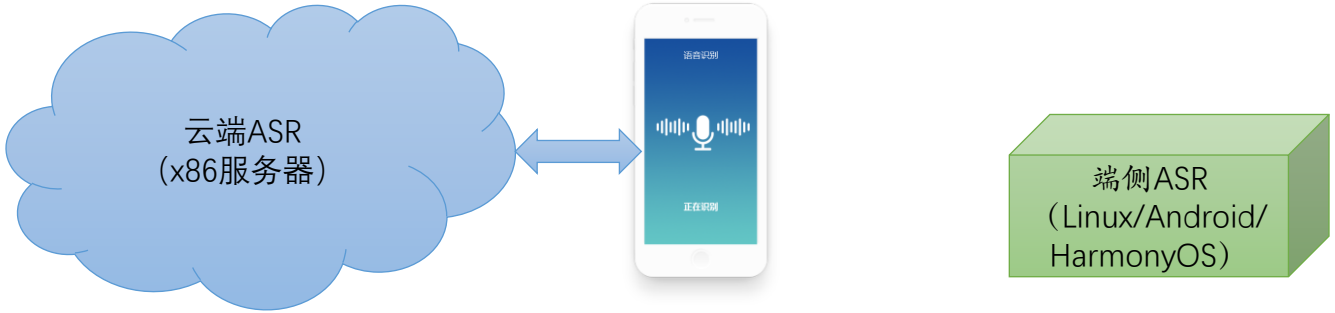


未来是万物互联的全场景智慧时代，需要适应不同场景多终端的语音交互技术

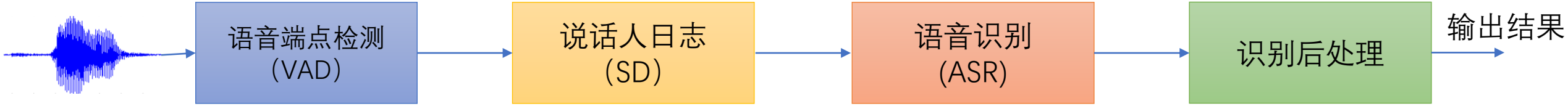
15.1 应用场景

- **语音算法需求**
  - 自动角色分离
  - 更好的识别效果
  - 快速定制能力
    - 新场景支持
    - 热词功能

- **工业落地需求**
  - 云端部署
  - 端侧部署

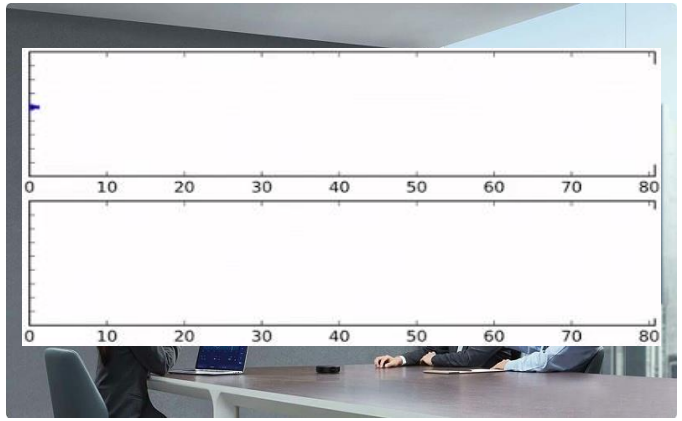


15.1 应用场景



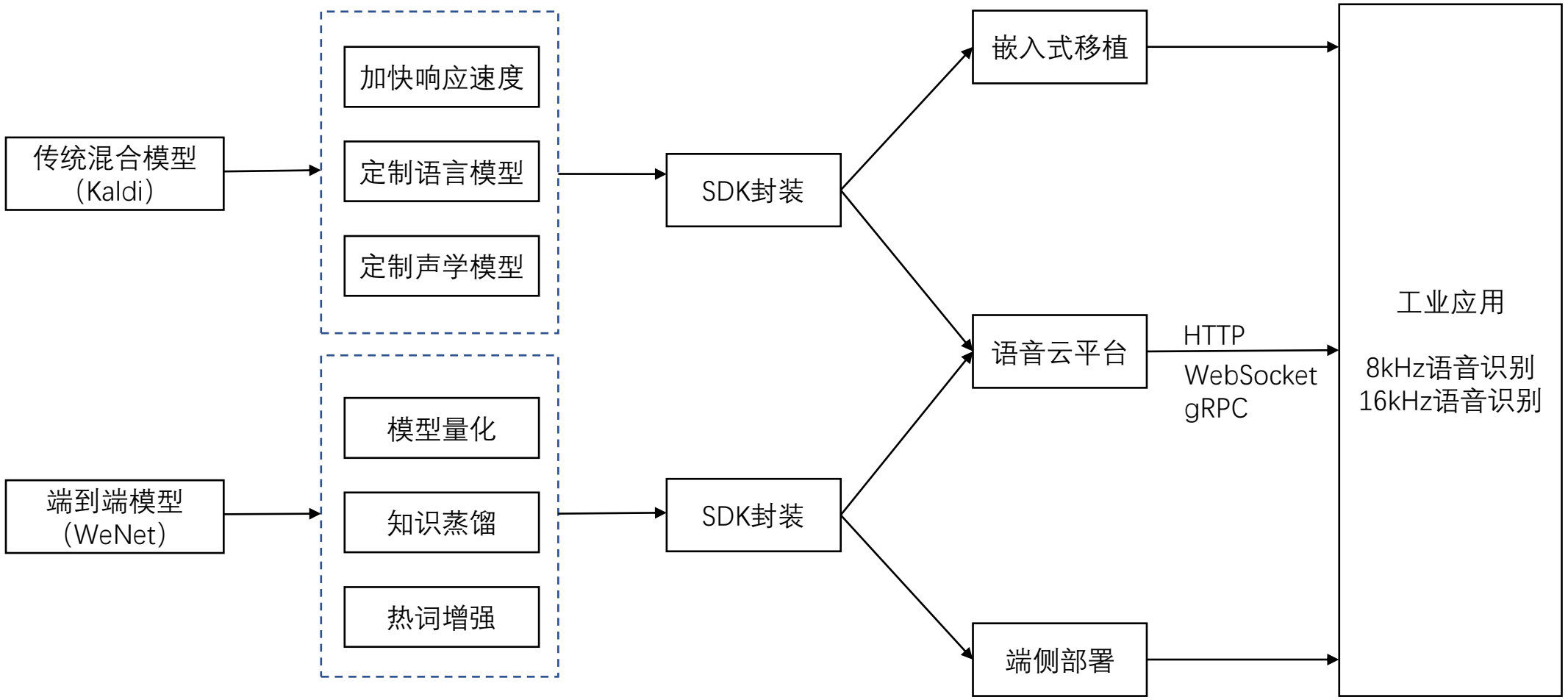
说话人分割聚类：给定一个包含多人交替说话的语音，系统需要判断每个时间段是谁在说话。

音频



分割信息

15.1 应用场景


传统混合模型
(Kaldi)
加快响应速度
定制语言模型
定制声学模型
端到端模型
(WeNet)
模型量化
知识蒸馏
热词增强
SDK封装
SDK封装
嵌入式移植
语音云平台
端侧部署
HTTP
WebSocket
gRPC
工业应用
8kHz语音识别
16kHz语音识别

15.2 引擎优化

Kaldi方案

1）加快响应速度

2）定制语言模型

3）定制声学模型

1）加快响应速度

- 语音识别的响应速度可用实时率（RTF）来衡量，即识别时间与语音时长的比值，越低越好。响应速度也可采用倍实时指标，倍实时跟RTF正好相反，是语音时长与识别时间的比值，越高越好。
- 为加快响应速度，可从算法优化和工程优化两方面入手。算法优化主要针对WFST解码器优化。WFST的令牌传播机制和Lattice解码都有剪枝过程，默认的最大活跃节点数和剪枝阈值为：

  max-active=7000

  beam=15.0

  lattice-beam=8.0
- 对响应速度影响较大的是前两个参数，实验表明，识别率略微变差情况下，max-active和beam值可以减小如下：

  max-active=3000

  beam=10.0

  lattice-beam=8.0
- 修改后响应速度可大幅提升，比如3秒语音只要0.5秒即可识别出结果，即RTF<0.2。

2）定制语言模型

- 通用语言模型一般使用日常生活、工作、新闻等范畴的文本语料训练而成，对书面语或日常用语识别较好，但针对口语化表达或特定行业，如司法、证券、电力、医疗等，因为有其专业的术语，往往识别不好。

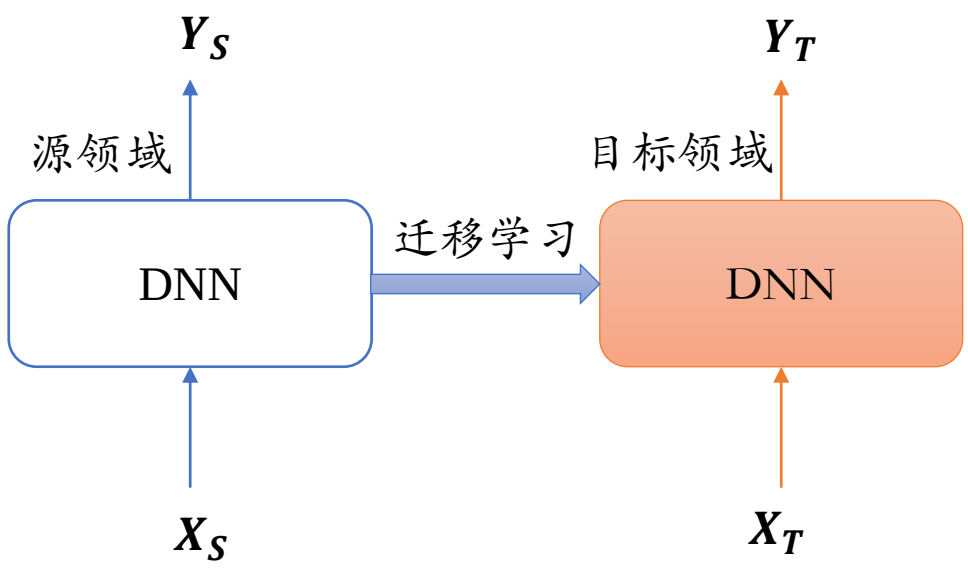
- 专用语言模型可以只用特定行业的句子表达来训练，但语料规模一般偏少，需要人为地设计一些类似的句子，尽可能覆盖到实际可能用的表达，使训练出来的统计模型覆盖更全面。

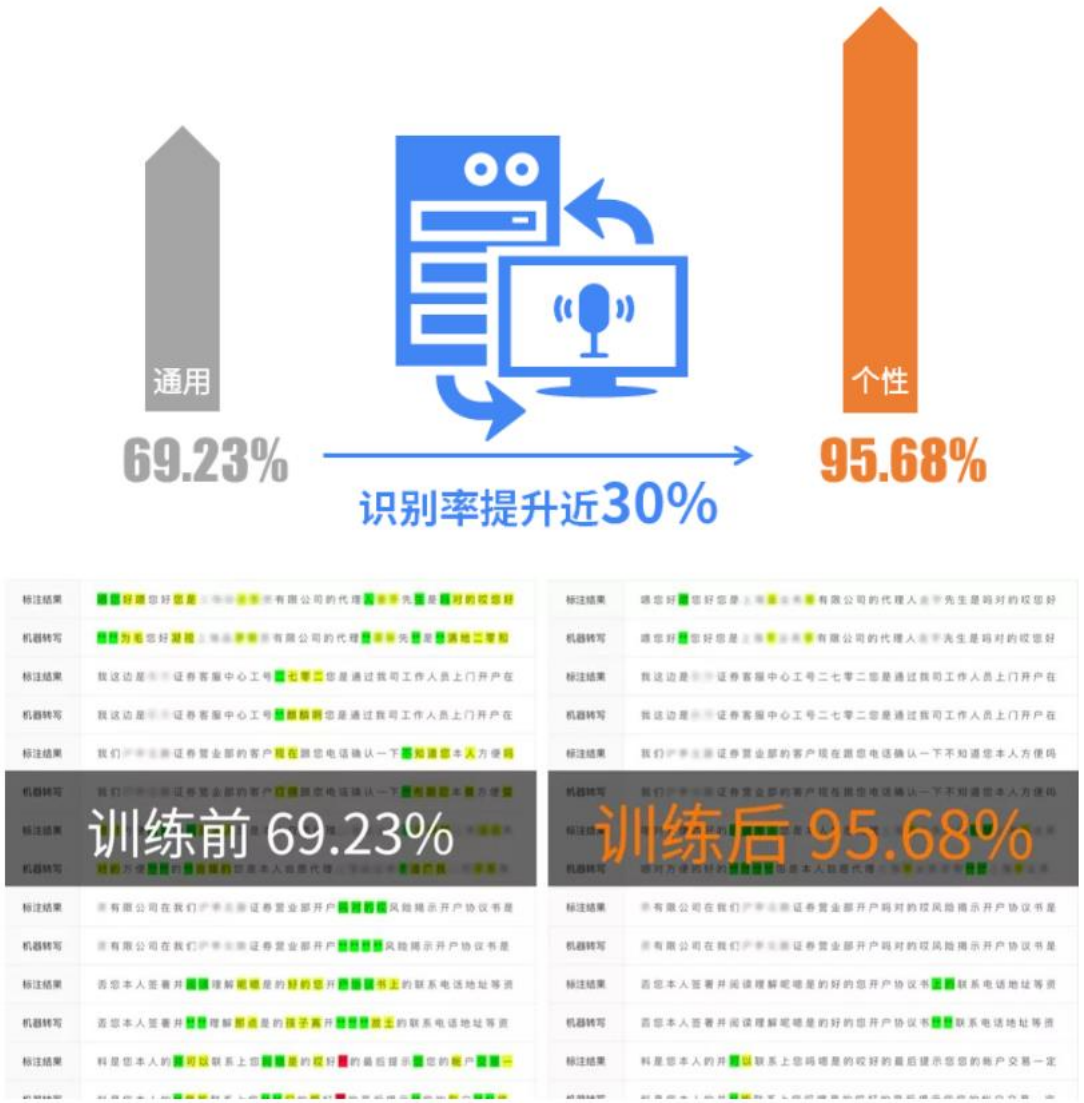
3）定制声学模型

- 如果采集设备比较特殊，跟常用的PC或手机麦克风差异较大，而声学模型的训练数据没有覆盖到这种录音，则识别性能会急剧下降，这也是一种跨信道问题。

3）定制声学模型—迁移学习

DNN迁移学习：适配不同场景（包括不同终端，不同格式等）





8k电话语音识别

工业界：识别准确率普遍在80~90%，与手机APP识别（16k）差距较大。

声学模型存在问题：

- 采样率8kHz：采集信道与麦克风差异很大，不能直接用16k模型识别；
- 特种设备：量化编码存在失真；
- 快语速：识别变差；

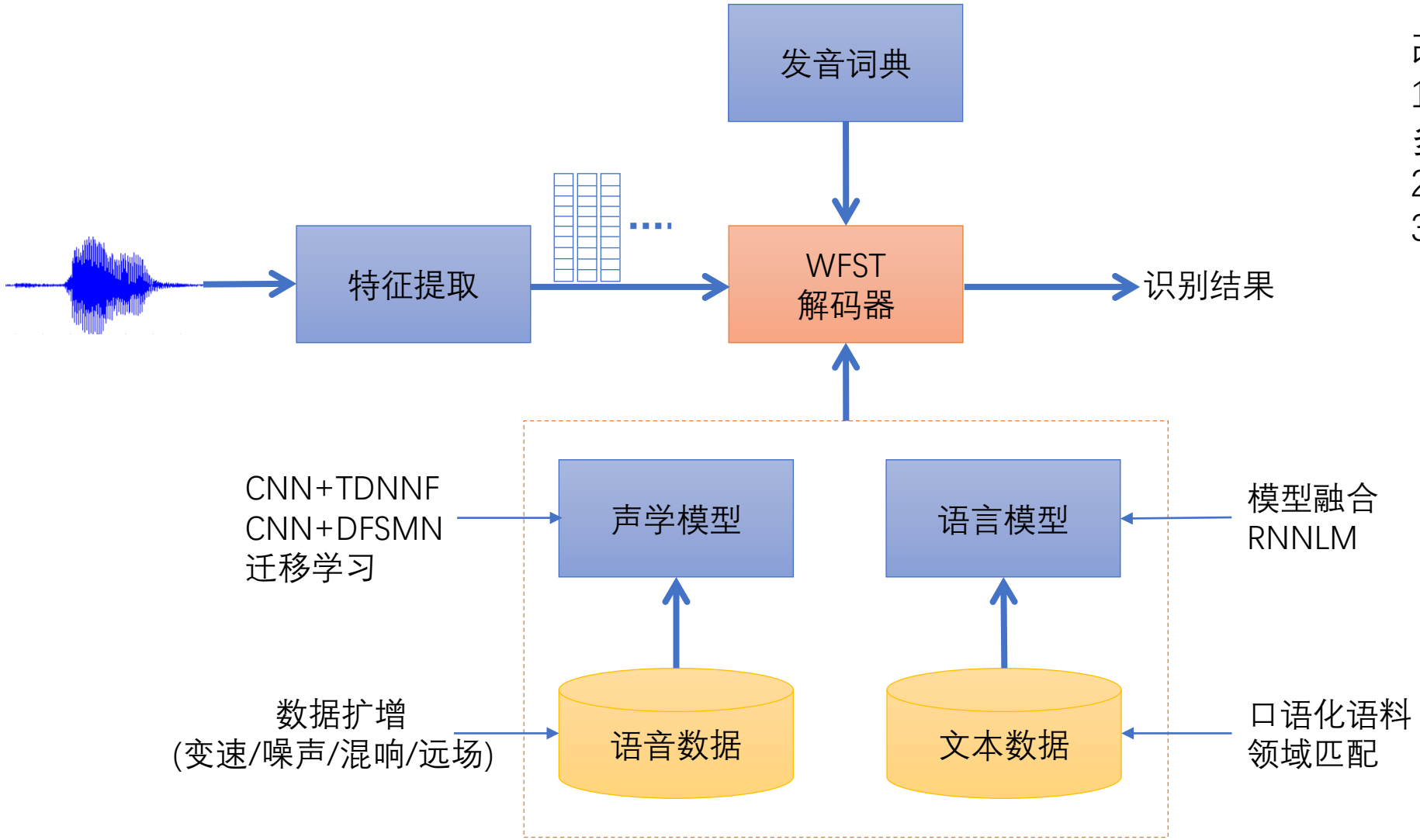
语言模型存在问题：

- 口语化表达难以覆盖；
- 通用领域与专业领域较难平衡；

可归结为两大原因：

1. 信道差异
2. 口语化识别

模型优化方案



改进方案：
1.采用数据扩增，进行多类型、多条件训练。
2. 采用更先进的声学模型。
3. DNN迁移学习。

15.2 引擎优化

WeNet方案

1）模型量化

2）知识蒸馏

3）热词增强

1）模型量化

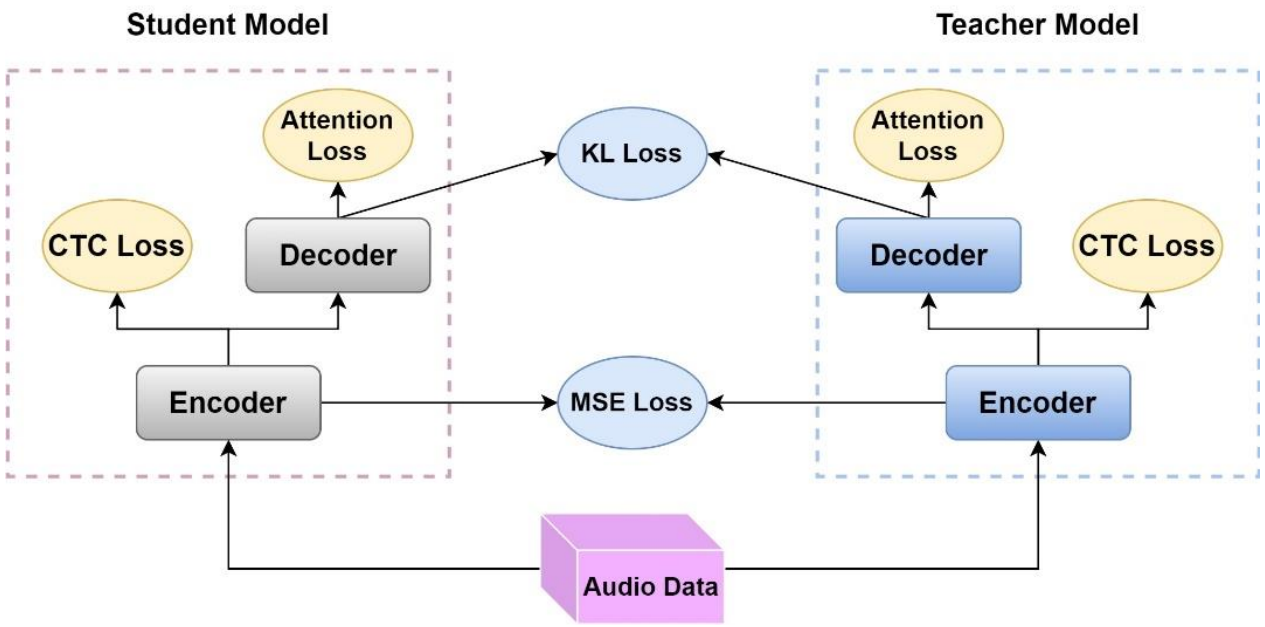
- 量化是把浮点（float）模型转为整型（int）模型，包含动态量化和静态量化两种方式。动态量化的缩放因子和零点是在推理时计算的，因此更准确，但引入额外的计算开销。静态量化采用固定的缩放因子和零点，计算更快，但需要额外的校准数据集，通过离线计算得到参数值。

- WeNet训练的端到端模型，可导出动态量化版本，把32位的float数据转成8位的int数据，对模型进行压缩，加快推理速度，同时准确率损失较小。

- 量化版本通过export_jit.py导出，默认是LibTorch模型，具体操作在WeNet实践一章有介绍。

2）知识蒸馏

- 实际应用尤其是端侧部署，希望模型越小越好，但模型变小，识别效果往往变差。为提升小模型的识别效果，可采用知识蒸馏方法，用大模型做教师模型，指导小模型（学生模型）训练，如图所示。



3）热词增强

◆拼音替换方案：采用模糊音判断，对近似音或同音字进行替换，例如：

“安想混合”=>“安享混合”

其中“安享”是设定的热词，其带声调拼音为an1 xiang3，只要发音为an xiang（不限声调）的词汇均转为该热词。

◆语言模型（LM）方案：可以支持更多词汇，在解码过程中，通过语言模型分（LM cost），提高特定词（热词）的概率，使之更容易被识别出来。语言模型方案实现过程如下：

- Unigram：首先根据热词集构建contextual biasing的G2.fst。
- on-the-fly Rescoring：在解码过程中，每当解码出现热词时，立即再加上G2.fst中的LM weight。

```
context_score=5.0
context_path=/work/hotword.txt

#Decoding with runtime
./tools/decode.sh --nj 16 \
    --context_score $context_score \
    --context_path $context_path \
    --ctc_weight 0.5 --rescoring_weight 1.0 --chunk_size -1 --reverse_weight 0.0 \
    /work/wenet/data/$dataset/wav.scp \
    /work/wenet/data/$dataset/text $dir/final_avg.zip $dir/lang_char.txt \
    $dir/${dataset}_context_${context_score}
```

15.3 SDK封装

1）函数接口
2）动态库编译
3）动态库调用

1）函数接口

```
/*****************************************************************************************************/
/*
                                    基于语音缓冲区的语音识别（非特定人）
                        @ handle        : 线路资源标识（必须是已打开）
                        @ buffer        : 待识别语音缓冲区
                        @ length        : 待识别语音缓冲区长度
                        @ text          : 识别的文本内容
                        @ scoreASR      : 语音识别得分
                        @ return        : 成功识别返回SUCCESS
                                                                                                    */
/*****************************************************************************************************/
return_ASR_Code ASR_recSpeechBuf(Handle handle, short* buffer, unsigned long length, char* text, float &scoreASR);
```

1）函数接口

- 根据ASR_recSpeechBuf函数的输入和输出参数，我们改写了Kaldi的在线解码程序，包括以下函数：
  - **ASR_recSpeechBuf**函数：与外面调用程序交互，首先判断分配到的句柄handle是否空闲，如果忙则返回ASR_STATE_ERROR，表示已被占用；如果检查通过，则调用KaldiDecode函数进行解码，并把词序列索引转化为文本内容，保存到输出参数text，即为识别后的句子。
  - **KaldiDecode**函数：实现从语音缓冲buffer到识别结果的具体解码过程，首先完成输入buffer到SubVector<BaseFloat> wave_part的转化过程，然后调用feature_pipeline.AcceptWaveform(samp_freq, wave_part)进行声学特征提取，注意声学特征一般有做倒谱均值减（CMN），因此在函数内部还要加上这步操作，接着调用decoder.AdvanceDecoding()进行分片段识别，得到中间解码结果，保存在Lattice里，随后采用decoder.FinalizeDecoding()进行Lattice解码，修正中间部分结果。最后调用GetDiagnosticsAndPrintOutput函数得到解码后的词序列索引。
  - **GetDiagnosticsAndPrintOutput**函数：根据输入的CompactLattice进行Lattice最优路径搜索并返回得到词序列和基于最小贝叶斯风险算出来的置信度，分别存放在输出参数words和words_conf。

1）函数接口—返回值

```
enum return_ASR_Code
{
        ASR_SUCCEEDED_OK=0,          //      0:操作成功
        ASR_WORKINGDIR_NOT_FIND,     //      1:工作目录不存在
        ASR_CONFIG_FILE_NOT_FOUND,   //      2:配置文件未找到
        ASR_MODEL_FILE_NOT_FOUND,    //      3:模型文件未找到
        ASR_LICENSE_ERROR,           //      4:授权有误
        ASR_HANDLE_ERROR,            //      5:句柄标识有误
        ASR_STATE_ERROR,             //      6:句柄状态有误
        ASR_TOO_SHORT_BUFFER,        //      7:语音太短
        ASR_EXTRACT_FEAT_ERROR,      //      8:特征提取出错
        ASR_MODEL_LOAD_ERROR,        //      9:模型加载出错
        ASR_MODEL_SAVE_ERROR,        //      10:模型保存出错
        TSASR_BUSY,                  //      11:线路正忙
        TSASR_IDLE,                  //      12:线路空闲
        TSASR_CLOSE,                 //      13:线路关闭
        ASR_OTHER_ERROR              //      14:其他错误
};
```

1）函数接口—引擎初始化和关闭

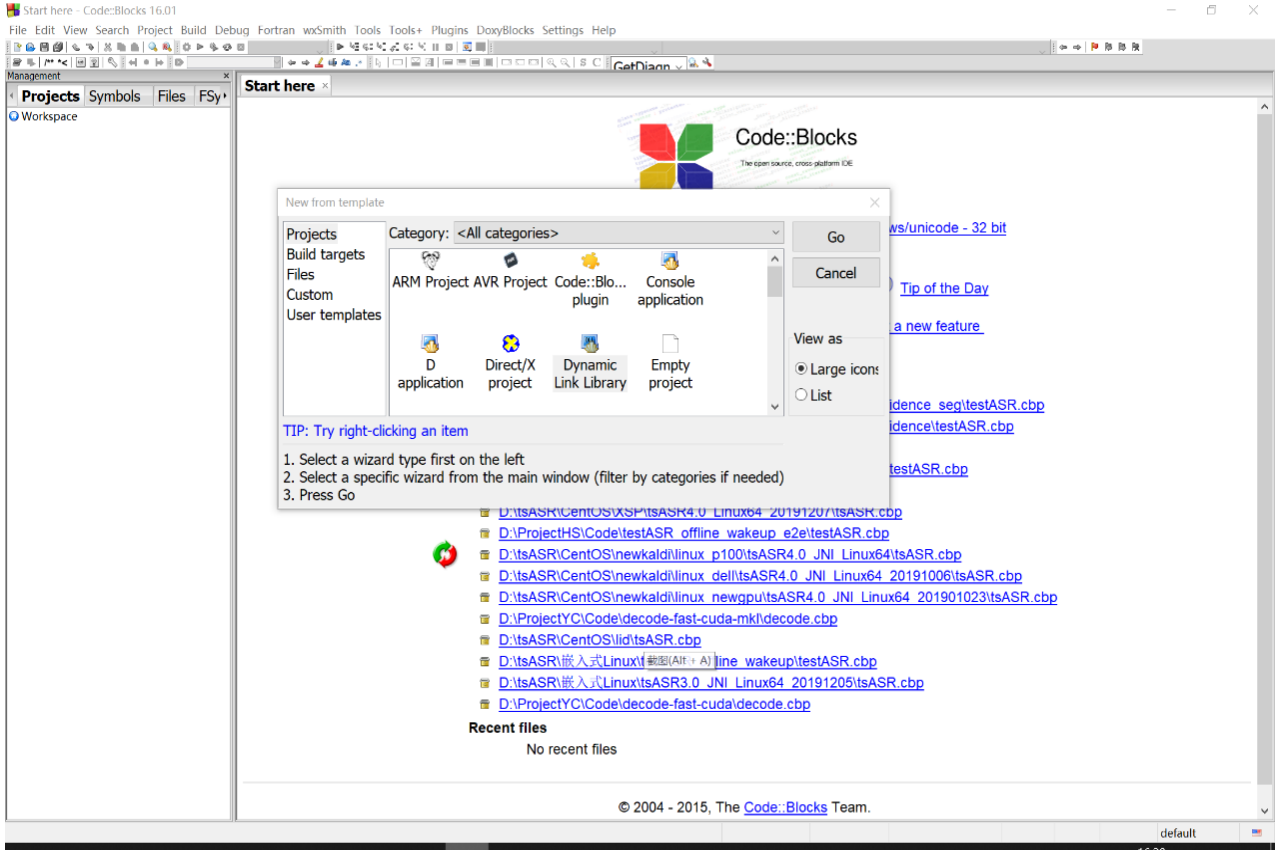
```
/*****************************************************************************************************/
/*
                                    初始化引擎
                        @ working_dir          : 工作目录下有引擎工作所必需的文件
                        @ max_lines            : 引擎支持最大线数
                        @ return               : 成功初始化返回SUCCESS
                                                                                                    */
/*****************************************************************************************************/
return_ASR_Code ASR_Init(const char* config_file,int max_lines);

/*****************************************************************************************************/
/*
                                    关闭引擎：改变各线路状态为CLOSE
                                                                                                    */
/*****************************************************************************************************/
return_ASR_Code ASR_Release();
```

1）函数接口—句柄打开和关闭

```
/*****************************************************************************************************/
/*
                                        打开线路：h是线路资源标识
                                                                                                    */
/*****************************************************************************************************/
return_ASR_Code ASR_Open(Handle &outh);


*****************************************************************************************************/
/*
                                        关闭线路：h是线路资源标识
                                                                                                    */
/*****************************************************************************************************/
return_ASR_Code ASR_Close(Handle &h);
```

2）动态库编译—Linux环境

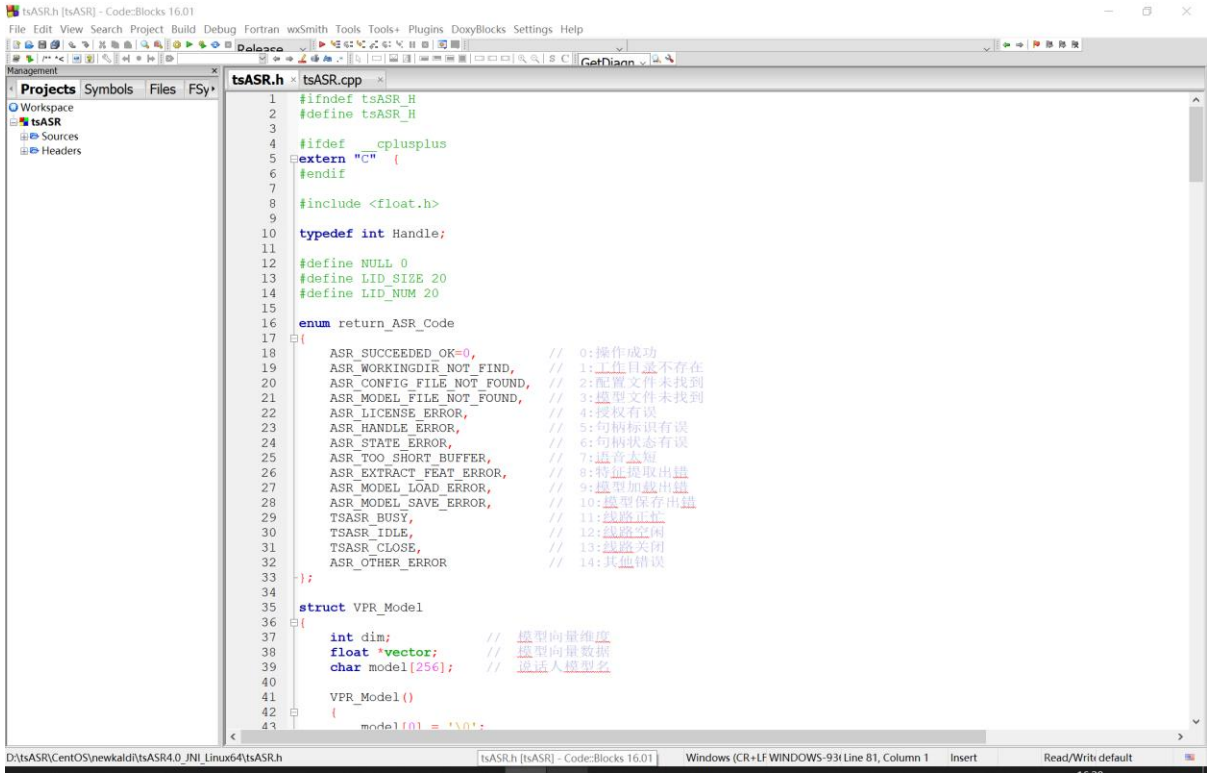
Linux环境不方便修改及调试代码，为便于操作，我们建议采用跨平台工具，开发环境可采用CodeBlocks，读者可下载最新版本并安装到Windows系统。以CodeBlocks16.01版本为例，一旦安装完成后，在文件菜单里选择新建->工程->动态库，如图所示。



2）动态库编译—Linux环境

厦门大学 XIAMEN UNIVERSITY

然后根据提示一步步创建，选择存放的目录，输入工程名，直到工程环境创建成功，如图所示。这时在工程目录会生成一个.cbp的工程文件，如asr.cbp。根据Kaldi函数调用关系，我们需要把在线解码需要的源程序全部加载到工程，并加入必要的外部支撑文件，用来读取配置文件，输出日志信息等。



2）动态库编译—Linux环境

工程配置保存完，把整个工程目录传到Linux环境，Linux的编译需要Makefile配置文件。为提高效率，可采用cbp2make工具（可网上下载）把asr.cbp工程文件转化为Makefile文件。有了Makefile文件，即可在Linux环境进行make编译。

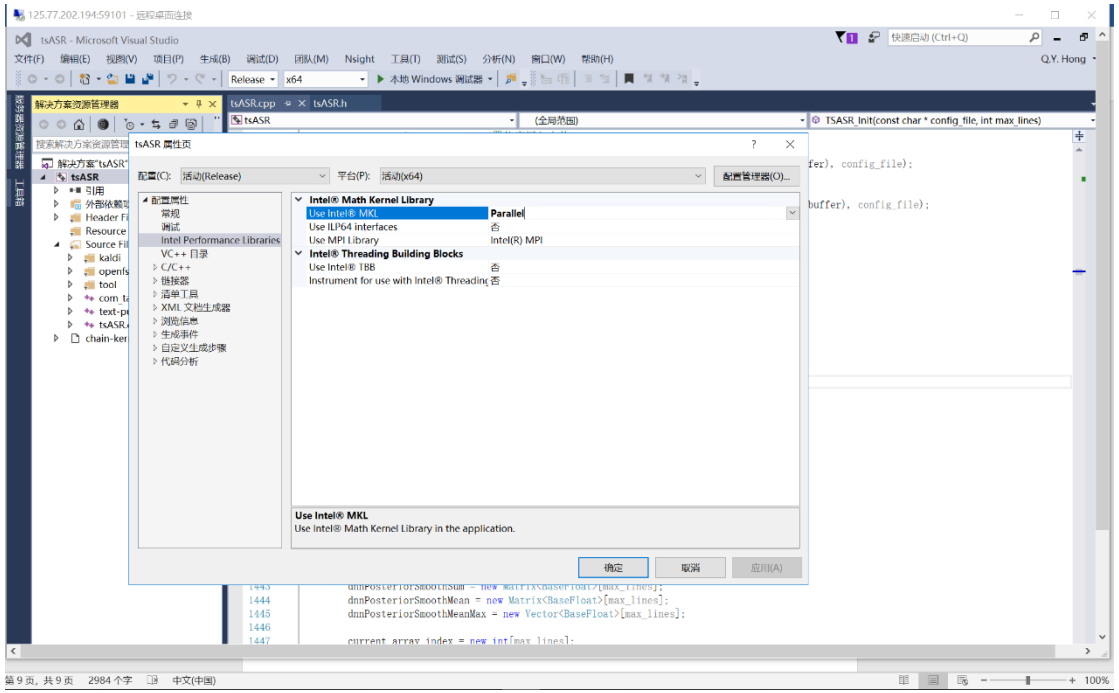
```
[采用Atlas库]
INC = -Isrc -Iopenfst/include -I/home/kaldi/tools/ATLAS/include -I/usr/local/cuda/include/ -I/home/kaldi/tools/portaudio/include -
I/home/kaldi/tools/portaudio/install/include -I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/libxml2/
CFLAGS = -std=c++11 -DHAVE_ATLAS -DHAVE_POSIX_MEMALIGN
RESINC =
LIBDIR =
LIB = lib/libatlas.so lib/liblapack.so
LDFLAGS = -ldl
```

由于Kaldi代码众多，包含很多子模块，函数互相之间关联度较强，加载的文件可能存在冗余或冲突，导致各种编译错误，读者需要根据报错信息一一修正，直至编译成功，最后生成so动态库文件。

这个so动态库需配套相应的头文件，包含可调用的函数接口及参数说明，供外部调用参照。

2）动态库编译—Windows环境

Windows环境编译的是dll动态库，主要采用Visual Studio开发工具。由于Kaldi代码采用C++ 11标准，需要安装Visual Studio 2015或更新的版本。另外，Windows环境的加速库只能采用Intel MKL或OpenBLAS。MKL集成相对容易，但需要安装Intel的配套工具，安装完MKL与Visual Studio 2015集成环境如图所示。



3）动态库调用

外部程序调用编译好的动态库，要先集成到工程里，如Linux在Makefile里配置如下：

```
CC = gcc
CXX = g++
AR = ar
LD = g++
WINDRES = windres

INC =
CFLAGS = -Wall -fexceptions
RESINC =
LIBDIR =
LIB = lib/libatlas.so lib/libtsASR.so lib/liblapack.so
LDFLAGS =
```

3）动态库调用

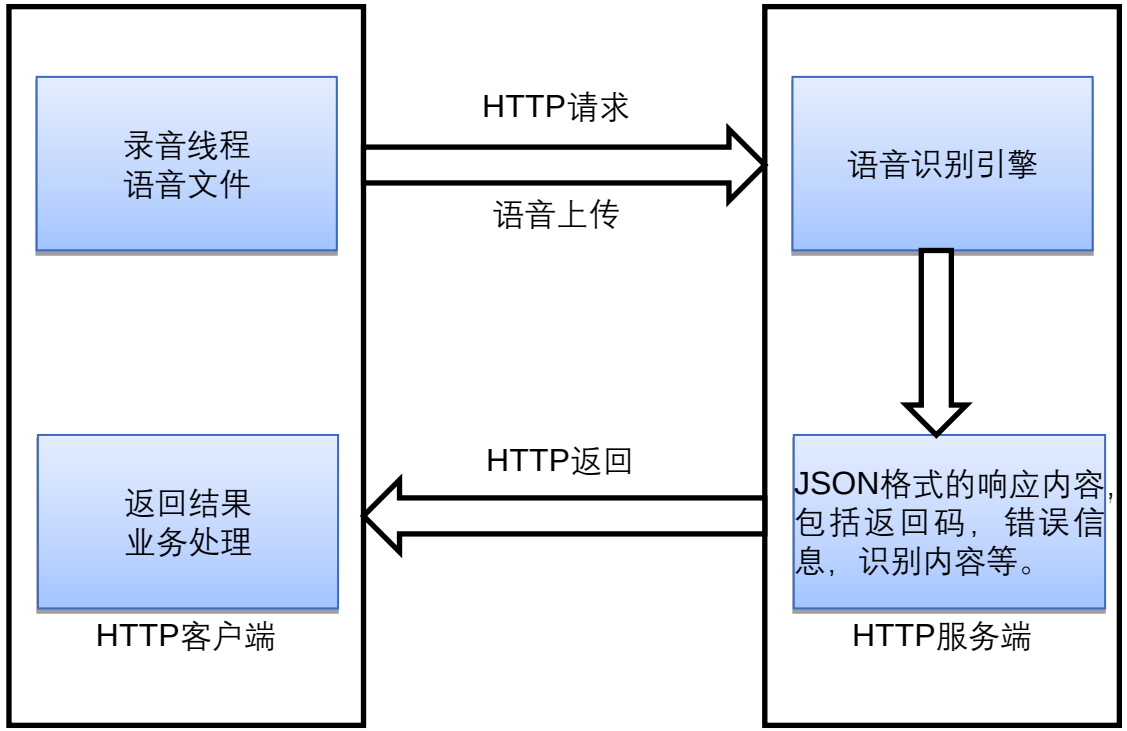
程序要调用时，先初始化引擎，然后分配句柄，再调用相关的识别函数，识别完关闭句柄。程序到最后还要关闭引擎，释放资源。

```
//初始化引擎
if(ASR_SUCCEEDED_OK == ASR_Init(config_asr_file.c_str(),3))
{
    cout<<"ASR init success!"<<endl;
}
else
{
    cout<<"ASR init fails!"<<endl;
    return -1;
}

//打开句柄
Handle tsASR;
ASR_Open(tsASR);

float scoreASR;
char rec_text[10240];

//语音识别
if(ASR_SUCCEEDED_OK == ASR_recSpeechBuf(tsASR,pWavBuffer,length,rec_text,scoreASR))
{
    cout<<"Recognized text: "<<rec_text<<endl;
    cout<<"scoreASR: "<<scoreASR<<endl;
}
else
{
    cout<<"Recognize "<<wave_full_file.c_str()<<"error!"<<endl;
}

//关闭句柄
ASR_Close(tsASR);

//关闭引擎
ASR_Release();
```

15.3.2 语音云平台

语音云平台可通过RESTful的方式给开发者提供一个通用的HTTP接口。

如图所示，系统通过HTTP的协议来进行调用。客户端采用HTTP Post，发Post请求到服务器，然后获取服务器的响应，根据响应的代码，判断操作是否成功。客户端负责语音的采集，并将采集后的语音上传到服务端，由服务端进行语音识别，并将结果返回到客户端。



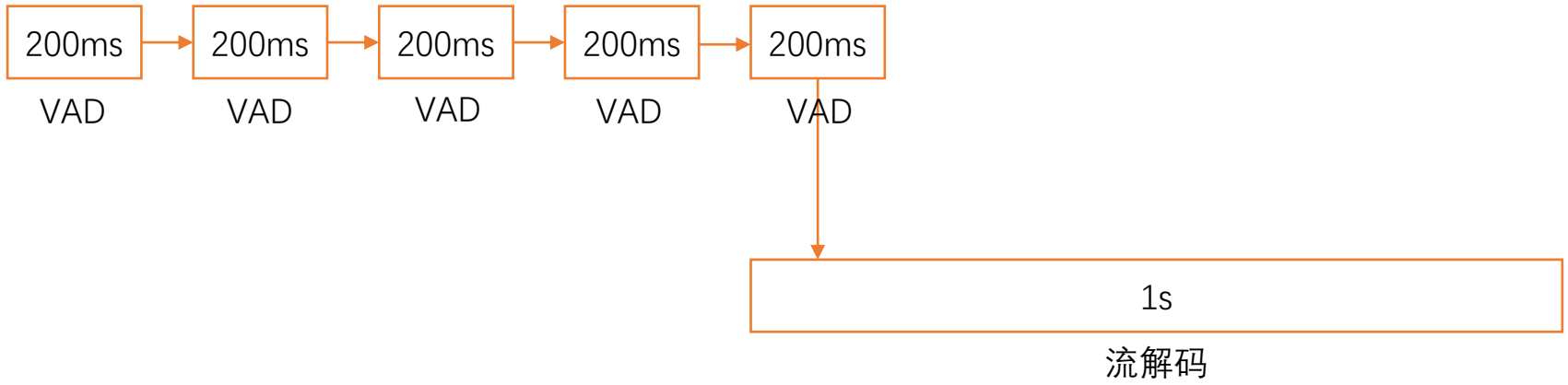
15.3.2 语音云平台

- HTTP协议服务一般通过高级语言，如go、Python或Java等语言实现，接收HTTP多路并发请求，使用多线程技术调用引擎进行识别，并用JSON格式返回识别结果。
- HTTP接口协议包括如下：

```
传送字节流
必选字段:
userid: 用户名称，可使用用户手机号码
token: 系统分配

file: 文件标识
语音缓冲（可以合并传，也可分段传，但不能有间隔符）：
buffer1(录音缓冲区）+
buffer2(录音缓冲区）+
buffer3(录音缓冲区）+
...
bufferN(录音缓冲区)
识别成功服务器返回:
{"result":"语音识别识别内容文本","errCode":"0","wavurl":"xxx.wav"}
wavurl是识别结果文本对应的语音文件url地址.默认是空""
识别结果是utf-8 编码。

识别失败服务器返回:
类似下面这个:{"result":"","errCode":"-1","AsrRetCode":"5"}
识别失败,会返回errCode!=0
说明:file内容字节流长度不能小于4000Byte
```

15.3.2 语音云平台—流识别

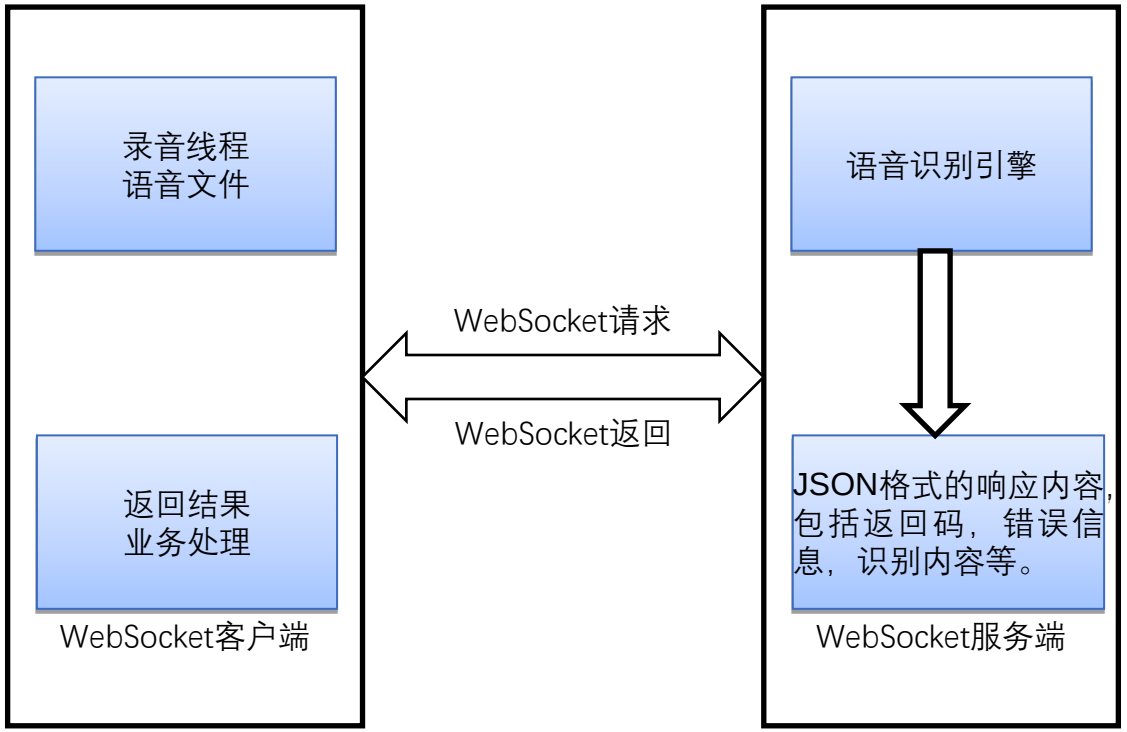
- 语音识别过程还可通过流的方式实现，如语音云平台的客户端一次可送200ms左右的片段，服务端接收后拼接，累计到1s时长即可开始识别，这个识别过程也是部分解码，只是令牌传播过程，可以显示中间结果。等句子结束后，再进行Lattice回溯解码，得到最后识别结果。



- 流识别结果跟整句识别略微有差别，但影响不大，能够保证实用性能。流识别方式可以边说边识别，大大缩短时延，有效提升响应速度。
- 现在工业界部署的云平台，普遍采用流识别方式。

15.3.2 语音云平台—WebSocket

为更好地支持流识别，另一种方案是采用WebSocket协议，其框架如图所示。WebSocket使用长链接，实现客户端和服务端双向数据传输，请求和返回效率更高。



15.3.3 Kaldi嵌入式移植

- 大部分的ARM平台都带有Linux系统，但用的是裁剪过的版本，里面的很多库跟CentOS或Ubuntu不兼容或缺少，因此移植起来较为繁琐。

- 以Kaldi的移植为例，其依赖的Atlas加速库包含libatlas.so和liblapack.so两个动态库，需要在ARM Linux环境重新编译。但单独编译这两个库有很多繁琐的配置，一种简便办法是通过Kaldi集成编译后得到，因为其工具包tools自带的安装脚本extras/install_atlas.sh会生成这两个so动态库。基于这两个so，重新编译引擎动态库。

15.3.3 Kaldi嵌入式移植

- 由于内存空间受限，很多裁剪过的ARM Linux系统，不带编译环境，即只能运行不能编译程序。为编译引擎动态库，我们需要通过交叉编译方式，在服务器另外搭一套编译的环境，所采用的gcc/g++编译器要跟ARM Linux一致，编译成功后再移植到ARM Linux环境。

```
CC = arm-buildroot-linux-gnueabihf-gcc
CXX = arm-buildroot-linux-gnueabihf-g++
AR = arm-buildroot-linux-gnueabihf-ar
LD = arm-buildroot-linux-gnueabihf-g++

WINDRES = windres

INC = -Isrc -Iopenfst/include -IOpenBLAS/include
CFLAGS = -std=c++11 -DHAVE_OPENBLAS -DHAVE_POSIX_MEMALIGN
RESINC =
LIBDIR =
LIB = lib/libopenblas.so
LDFLAGS = -ldl lib/libclapack.a lib/libblas.a lib/libf2c.a
```

15.3.3 Kaldi嵌入式移植

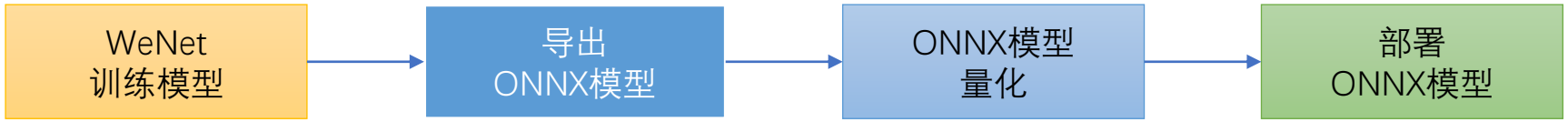
- 嵌入式语音识别一般只支持命令词识别或小范围的“随便说”，如机器人的动作命令和简单的对话。对话内容可能只有几百句，语言模型就可以做得很小，而声学模型本身不会太大，因此编译出来的HCLG也就很小，文件只有几百KByte甚至更小。这样更容易部署到嵌入式平台，识别速度也更快。

- 针对更低端的DSP平台，嵌入式移植可能还涉及定点化问题，需要把解码器特别是特征提取和DNN的前向传播重新改写，即把浮点运算改为定点运算，而且不再包含加速库，这部分工作量比较大。

15.3.4 WeNet端侧部署

端侧语音识别

- 端侧设备：X86、ARM
- 要求：实时率、模型大小
- 端到端架构：Conformer模型
- 模型压缩方案：量化模型、知识蒸馏、网络权重共享



15.3.4 WeNet端侧部署

在端侧安装部署WeNet完，先通过export_onnx_cpu.py导出ONNX模型，然后再量化，调用脚本如下：

```
# pytorch模型导出onnx
python wenet/bin/export_onnx_cpu.py \
  --config export_onnx_test/train.yaml \
  --checkpoint export_onnx_test/20.pt \
  --chunk_size -1 \
  --num_decoding_left_chunk -1 \
  --output_dir export_onnx_test/onnx

# onnx模型量化
output_path=export_onnx_test/quant_onnx
mkdir -p $output_path
python wenet/bin/onnx_quantize.py \
  --encoder_onnx_file export_onnx_test/onnx/encoder.onnx \
  --decoder_onnx_file export_onnx_test/onnx/decoder.onnx \
  --ctc_onnx_file export_onnx_test/onnx/ctc.onnx \
  --output_quant_dir $output_path \
  --quantize_dynamic
```

15.3.4 WeNet端侧部署

输出ONNX量化模型，我们就可以测试语音识别的RTF:

```
export GLOG_logtostderr=1
export GLOG_v=2
./decoder_main  --onnx_dir ./onnx --dict_path ./words.txt --wav_scp ./wav.scp --num_threads 4
```

15.4 本章小结

- 本章面向工业应用实践，针对引擎优化，特别是声学模型和语言模型的改进，做了较为详细的介绍。

- 针对工程部署，本章详细讲解引擎动态库的封装编译过程，并给出外部程序调用引擎的例子。针对工业界最主流的产品形态——语音云平台，本章介绍了客户端和服务端之间的交互流程，并给出了具体的CURL调用例子。

- 最后还介绍了Kaldi嵌入式移植和WeNet端侧部署过程。