# 初探ffmpeg、jni

## 前言

本教程针对android系统ffmpeg移植，并在Linux系统上编译，如果要在windows上用cygwin+ndk进行编译，大概率会因为系统环境或者cygwin出现各种各样的问题，可以先尝试windows编译，如果遇到问题，请果断安装Linux虚拟机，否则可能会在编译问题上消耗大量时间最终却得不到想要的结果。

# 编译

如果是windows系统，需要用到VMware+ubuntu-12.04.3-desktop-amd64+ndk

## 第一步：下载

1. ubuntu-12.04.3-desktop-amd64（可以使用最新版本）
2. NDK Linux版本我用的是android-ndk-r9d-linux-x86\_64.tar.bz2
3. 下载ffmpeg-2.8.tar.gz

## 第二步：安装以及配置

1、安装VMware+ubuntu不做过多描述，如果遇到文件夹无法共享的问题，在此建议windows共享文件夹，ubuntu通过smb://局域网访问。

2、解压android-ndk-r9d-linux-x86\_64.tar.bz2于目录/home/chenran/ndk/android-ndk-r9d

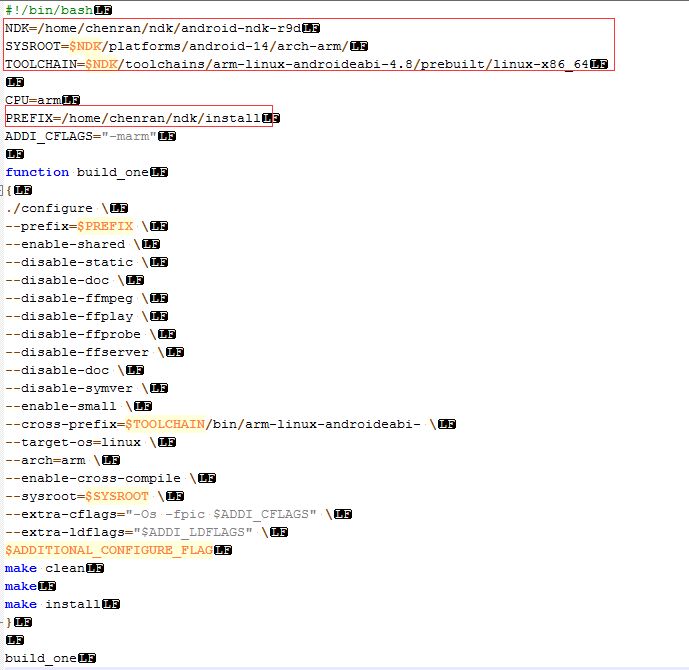
3、解压ffmpeg-2.8.tar.gz于目录/home/chenran/ndk/android-ndk-r9d/samples/ffmpeg-2.8

4、将配置文件build\_android\_arm.sh放于ffmpeg的根目录中



修改路径：

将NDK、SYSROOT、TOOLCHAIN路径修改为本地路径，并配置安装路径PREFIX，此路径为最终so文件生成路径



1. 打开configure文件

将该文件中的如下四行：

SLIBNAME\_WITH\_MAJOR='$(SLIBNAME).$(LIBMAJOR)'

LIB\_INSTALL\_EXTRA\_CMD='$$(RANLIB)"$(LIBDIR)/$(LIBNAME)"'

SLIB\_INSTALL\_NAME='$(SLIBNAME\_WITH\_VERSION)'

SLIB\_INSTALL\_LINKS='$(SLIBNAME\_WITH\_MAJOR)$(SLIBNAME)'

替换为：

SLIBNAME\_WITH\_MAJOR='$(SLIBPREF)$(FULLNAME)-$(LIBMAJOR)$(SLIBSUF)'

LIB\_INSTALL\_EXTRA\_CMD='$$(RANLIB)"$(LIBDIR)/$(LIBNAME)"'

SLIB\_INSTALL\_NAME='$(SLIBNAME\_WITH\_MAJOR)'

SLIB\_INSTALL\_LINKS='$(SLIBNAME)'

## 第三步：生成.so文件

1、给build\_arm.sh和configure文件增加可执行权限

终端进入ffmpeg目录并执行：

**$chmod +x ./** **build\_android\_arm.sh**

**$chmod +x ./configure**

2、给build\_android\_arm.sh和configure转换格式:

终端执行：（需要先安装 dos2unix）

**$dos2unix ./build\_android\_arm.sh**

**$dos2unix ./configure**

3、输入以下指令开始编译

**$./build\_android\_arm.sh**

整个编译过程需要几分钟，请耐心等候

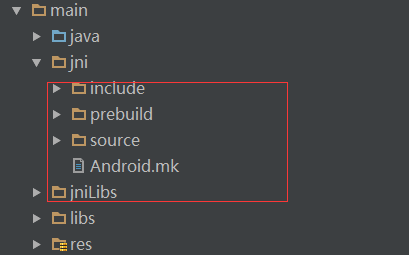
1. 编译完成后在PREFIX所对应的目录下可以找到相应的.so文件，以及include文件

# JNI

ffmpeg的功能很强大，能帮助解决的事情也很多，在这里，我们只用到ffmpeg的解码功能，不需要用到所有的库。

## 第一步：android项目配置

1. 创建一个安卓项目，并且在main目录下创建一个jni目录以及一个jniLibs目录，并且在jni目录下创建一个prebuild目录和一个source目录以及一个Android.mk文件。将所生成的include文件夹全部拷贝至jni目录下，将带有版本号的.so文件拷贝至prebuild目录下，例如：libavcodec-56.so而不是libavcodec.so，如图所示：

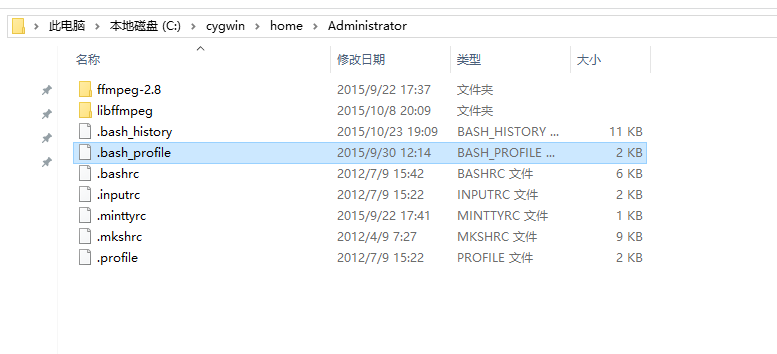


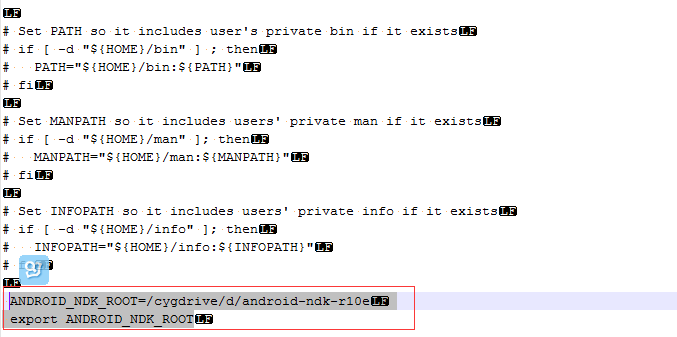
2、一个Android.mk文件向编译系统描述你的源代码。具体来说：该文件是GNU Makefile的一小部分，会被编译系统解析一次或多次。你可以在每一个Android.mk file中定义一个或多个模块，你也可以在几个模块中使用同一个源代码文件。说白了Android.mk文件就是将自己的原代码编译为.so文件的入口，上述创建的Android.mk文件只有一行代码：

**include $(all-subdir-makefiles)**

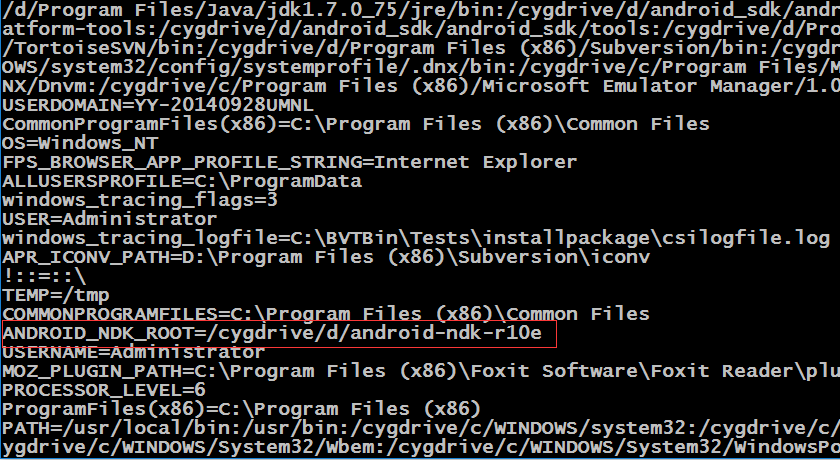
## 第二步：cygwin配置

因为本例只是用的Linux虚拟机去编译ffmpeg库的，但是开发的话，仍然使用windows，所以cygwin必不可缺。下载cygwin并且下载所需要的扩展包，如果不嫌麻烦安装所有的扩展包吧，可以找本人拿到cygwin压缩包，这样就不用下载这个庞然大物了。下载安装完成后，在安装目录下增加ndk环境变量，如下图所示：



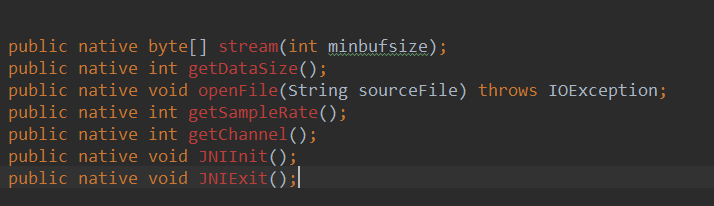


打开cygwin，输入命令env，如果有NDK的配置路径，那么表示配置成功

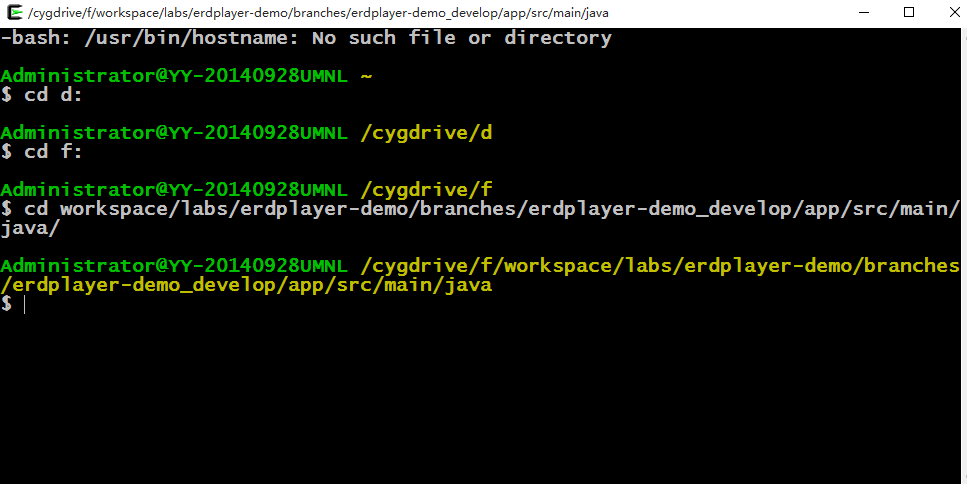


## 第三步：生成.h头文件和.cpp文件

1、创建一个带有native方法的java类，如图所示：

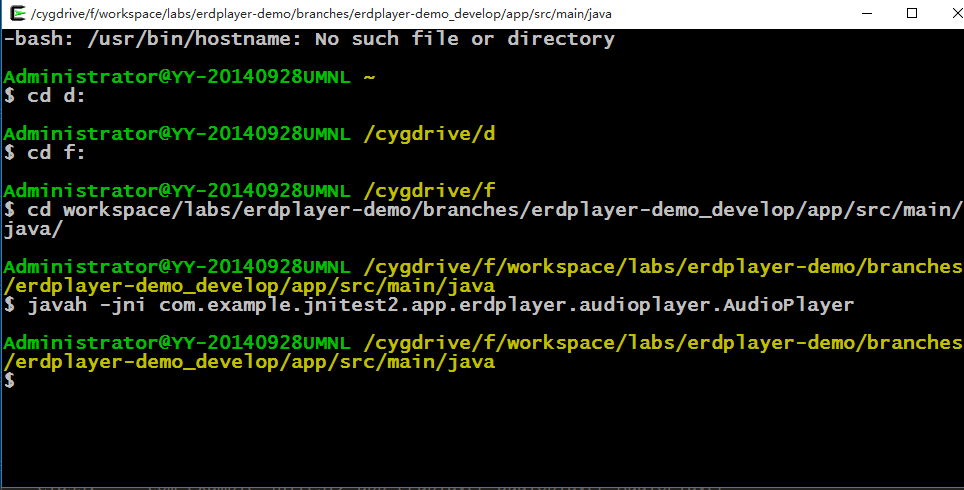


打开cygwin，并且将目录定位到项目所对应的java代码根目录下，如图所示：

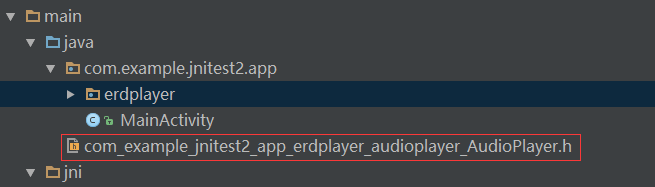


2、输入：

j**avah –jni 包名.类名**

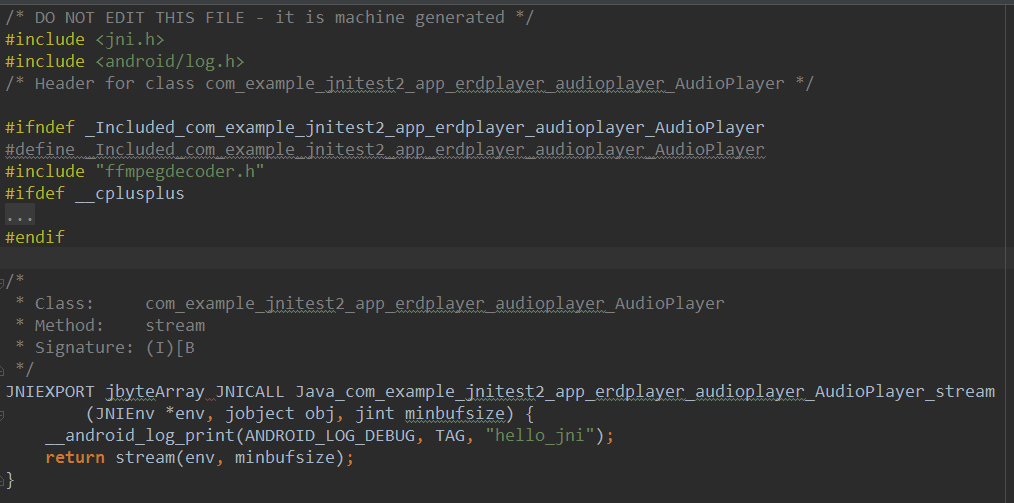


此时会自动生成一个.h文件，将此文件拷贝至source文件夹下



在source文件夹下创建一个与.h文件名相同的.cpp文件，直接拷贝.h文件的代码，并给每个方法加入参数名、方法实现、返回值。

3、导入android/log.h文件，并在一些方法上输出一些打印语句，然后在java上调用此方法，如图所示：



## 第四步：编写Android.mk文件和Application.mk文件

1、在source文件夹下创建Android.mk文件和Application.mk文件，Android.mk文件主要用作编译自己.so库的源代码配置信息，而Application.mk主要用于编译环境配置信息，附上这个这两个文件，



2、在Android.mk文件中：

**LOCAL\_PATH := $(call my-dir)**

此句话定义跟路径，这里是jni跟目录

**include $(CLEAR\_VARS)  
LOCAL\_MODULE := avcodec  
LOCAL\_SRC\_FILES := ../prebuild/libavcodec-56.so  
LOCAL\_EXPORT\_C\_INCLUDES := $(LOCAL\_PATH)/include  
include $(PREBUILT\_SHARED\_LIBRARY)  
  
include $(CLEAR\_VARS)  
LOCAL\_MODULE := avutil  
LOCAL\_SRC\_FILES := ../prebuild/libavutil-54.so  
LOCAL\_EXPORT\_C\_INCLUDES := $(LOCAL\_PATH)/include  
include $(PREBUILT\_SHARED\_LIBRARY)  
  
include $(CLEAR\_VARS)  
LOCAL\_MODULE := avformat  
LOCAL\_SRC\_FILES := ../prebuild/libavformat-56.so  
LOCAL\_EXPORT\_C\_INCLUDES := $(LOCAL\_PATH)/include  
include $(PREBUILT\_SHARED\_LIBRARY)  
  
include $(CLEAR\_VARS)  
LOCAL\_MODULE := swscale  
LOCAL\_SRC\_FILES := ../prebuild/libswscale-3.so  
LOCAL\_EXPORT\_C\_INCLUDES := $(LOCAL\_PATH)/include  
include $(PREBUILT\_SHARED\_LIBRARY)  
  
include $(CLEAR\_VARS)  
LOCAL\_MODULE := swresample  
LOCAL\_SRC\_FILES := ../prebuild/libswresample-1.so  
LOCAL\_EXPORT\_C\_INCLUDES := $(LOCAL\_PATH)/include  
include $(PREBUILT\_SHARED\_LIBRARY)**

以上代码定义了我们所需要预编译的动态库，也就是ffmpeg所对应的动态库，一定要放在自己需要编译的动态库前，否则将无法找到所对应的方法，导致编译无法通过。因为我们只需要用到ffmpeg的编码库，所以以上四个动态库是需要用到的。其他库不做预编译

**include $(CLEAR\_VARS)  
LOCAL\_C\_INCLUDES += \  
 $(LOCAL\_PATH)/../prebuild \  
 $(LOCAL\_PATH)/../include**

以上代码用于指定根目录下的子目录路径，必须指定，否则将无法找到所需要的.so文件和ffmpeg头文件

**LOCAL\_SRC\_FILES := \  
 com\_example\_jnitest2\_app\_erdplayer\_audioplayer\_AudioDecoder.cpp \  
 ffmpegdecoder.cpp  
LOCAL\_MODULE := libaudioplayer****LOCAL\_SHARED\_LIBRARIES := avcodec avutil swresample avformat swscale  
include $(BUILD\_SHARED\_LIBRARY)**

以上代码为自己编写的c++代码所需要编译为.so文件的配置信息，LOCAO\_MODULE为编译出来的文件名例如：libaudioplayer.so，LOCAL\_SHARED\_LIBRARIES代表代码中所需要引用到的动态库

1. 在Application.mk文件中

**APP\_ABI := armeabi**

表示编译平台，目前我们只需要arm平台即可，因为市面上的x86机型很少，所以无需支持，如果一定要支持x86平台，之前ffmpeg在编译的时候也要编译出x86平台对应的so库.

**APP\_PLATFORM := android-14**

指定了NDK编译所用到的版本，按照安卓项目的最低版本配置，具有向下兼容性

**APP\_STL := gnustl\_shared**

指定stl标准库的.so文件名称

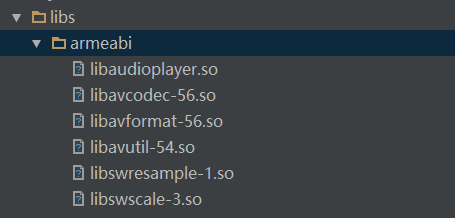
**APP\_CFLAGS += -Wno-error=format-security**

这一行将cygwin编译报出的字符串错误当做警告处理

1. 配置文件处理完成后，cygwin回到jni目录下，输入以下命令，进行编译：

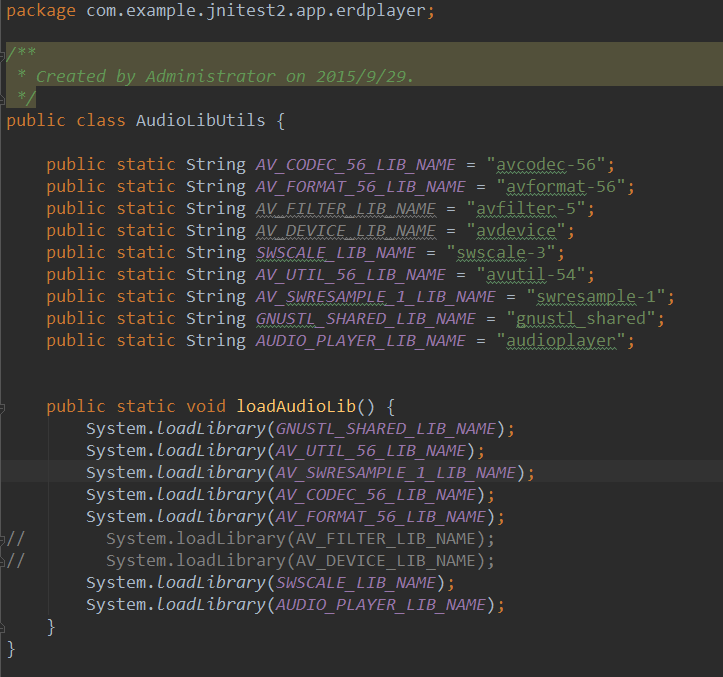
**$ANDROID\_NDK\_ROOT/ndk-build**

编译成功，将在jni目录下面自动生成一个libs文件夹，其中包含了下图所示几个.so文件，在main目录下创建一个jniLibs文件夹，将文件.so文件拷贝到这个文件夹下



如果编译出错，根据报错信息，自行查找错误原因。

1. 回到java代码中，在调用jni方法之前必须load这几个so库，如图所示：



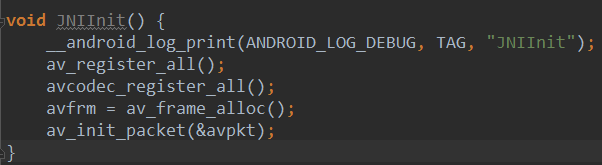
注意load顺序，因为ffmpeg中的几个库存在依赖关系，所以必须按照顺序load否则会运行报错

# FFMpeg使用

ffmpeg中包含了很多文件接口，我们需要使用的解码功能主要位于avcodec、avformat和avutil包下，其中avformat用于各种音视频[封装格式](http://baike.baidu.com/view/1942911.htm)的生成和解析，包括获取解码所需信息以生成解码上下文结构和读取音视频帧等功能，avcodec用于各种类型声音/图像编解码，avutil是一个工具包

## 第一步：初始化

在使用ffmpeg之前需要初始化一些参数以及配置信息



avfrm表示AVFrame结构体，位于frame.h文件下，是用来描述解码后或者编码前的数据，即存储了原始数据，对于音频文件来说就是PCM，AVFrame中可能包含了一个音频帧的数据也可能包含了多个音频帧的,nb\_samples可获取帧数。uint8\_t \*data[AV\_NUM\_DATA\_POINTERS]表示了解码后的原始数据。

avpkt表示AVPacket结构体，这个结构体是ffmpeg解码之前或者编码后的原始数据

## 第二步：打开输出文件或远程URL地址

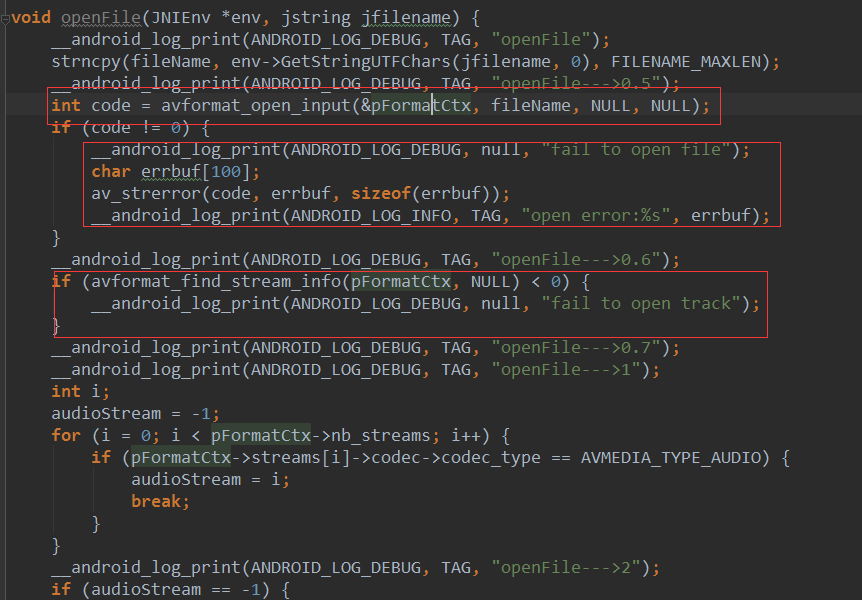
1、avformat\_open\_input方法将打开一个本地文件或者远程流媒体，注意：在android项目中必须开放网络以及本地读写权限，否则会报错。

2、pFormatCtx为AVFormatContext结构体，此结构体包含了流媒体的上下文信息，通过此上下文信息能够找到对应解码器的AVCodecContext，而AVCodecContext包含了一个AVCodecID，此ID是选择编码器的唯一判断条件。

3、avformat\_open\_input FFMPEG打开媒体的的过程开始于此函数，在该函数中，FFMPEG完成了：输入输出结构体AVIOContext的初始化；输入数据的协议（例如RTMP，或者file）的识别；判断文件名的后缀；读取文件头的数据进行比对；使用获得最高分的文件协议对应的URLProtocol，通过函数指针的方式，与FFMPEG连接等过程。

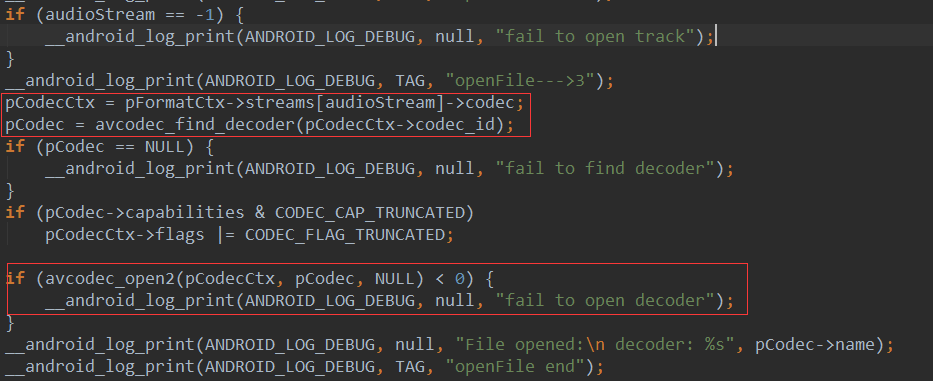
4、avformat\_find\_stream\_info函数主要用于给每个媒体流（音频/视频）的AVStream结构体赋值。它其实已经实现了解码器的查找，解码器的打开，视音频帧的读取，视音频帧的解码等工作。换句话说，该函数实际上已经“走通”的解码的整个流程。

5、av\_strerror函数用于错误信息打印，ffmpeg每个方法都将会返回一个int值，该值可以作为错误码传入此函数中，便于调试。



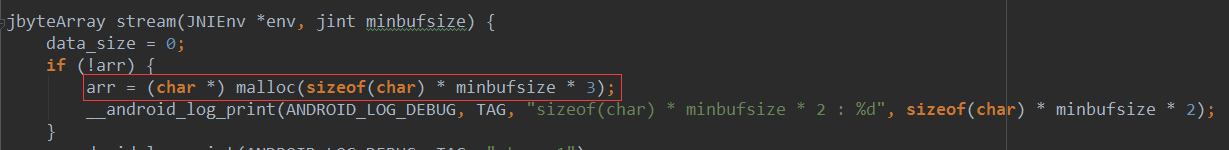
6、avcodec\_find\_decoder函数通过AVCodecContext获取的解码器ID返回一个AVCodec指针，AVCodec包含了对应解码器的一些信息，比如解码器名称、判断是音频还是视频、支持的采样率等。

7、avcodec\_open2函数用于初始化一个视音频编解码器的AVCodecContext



## 第三步：解码

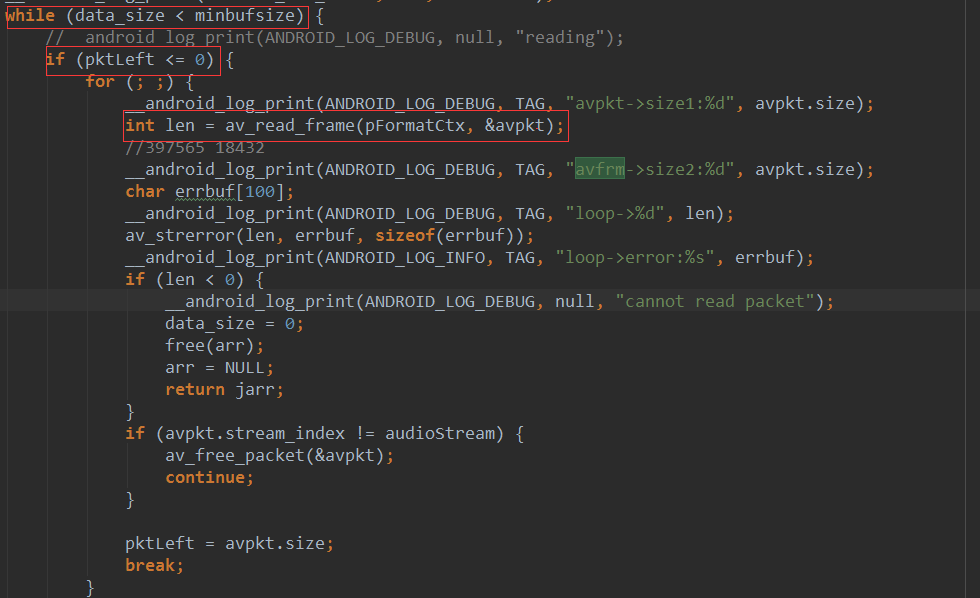
1、分配一块缓冲区内存，该内存大小可由上层需求自定义，因为此项目最终解码得到的PCM数据将写入audiotrack中，由audiotrack定义一块最小区域缓冲大小，而解码器的缓冲大小必须是这个最小值的2倍以上，因为一帧的数据如果大于这个最小值，就会内存溢出。



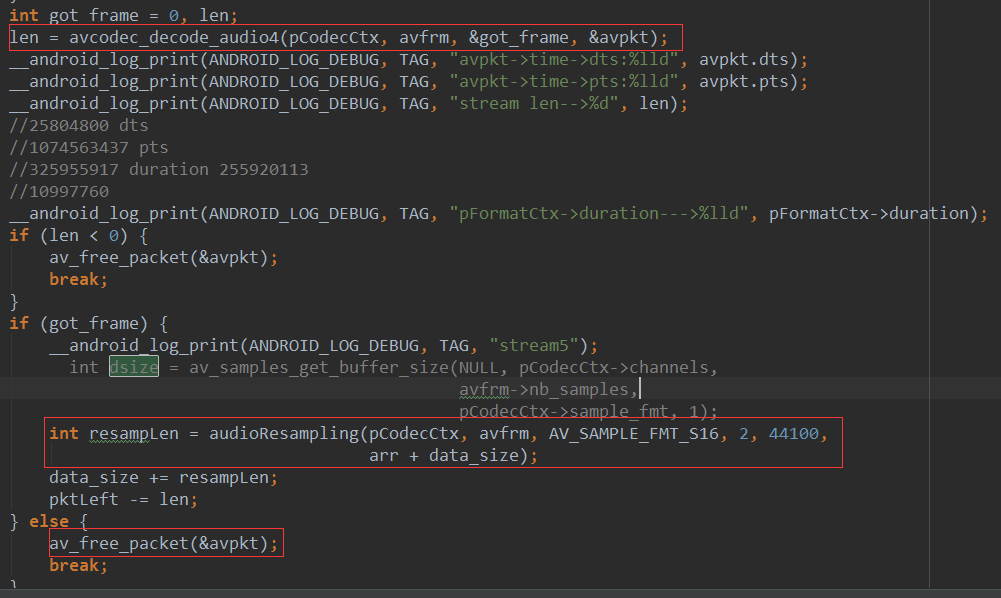
2、data\_size是每一帧数据缓存后的一个累加值，当该值大于最小缓冲值时，返回给上层，小于则循环解码。

3、av\_read\_frame用于读取码流中的音频若干帧或者视频一帧，正因为读取出来的音频帧可能是一帧也可能是多帧，而我们在做解码的时候，必须一帧一帧的解，所以添加一个pktLeft的变量控制，如果不加这个条件，将会导致跳帧的情况发生，因为每av\_read\_frame一次，

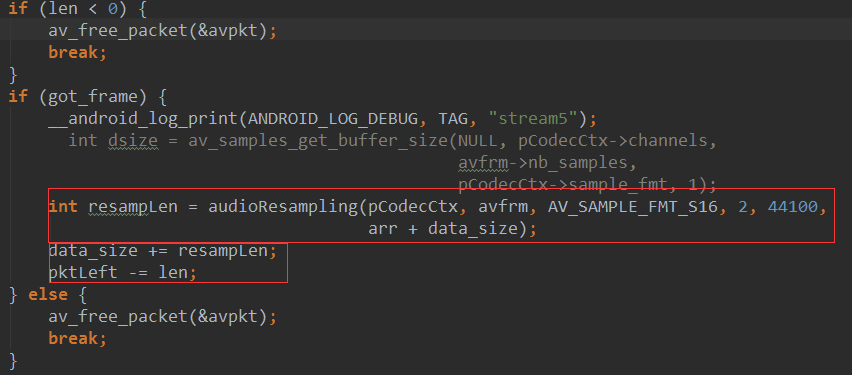
将重置AVPacket一次，如果此AVPacket包含了多帧，而只解码了一帧，其它帧将被丢弃。



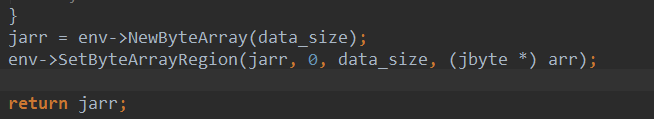
4、avcodec\_decode\_audio4()方法是解码的核心方法，调用此方法将得到最终我们需要的一帧PCM数据，并保存在AVFrame中，注意：我所用的ffmpeg2.8版本的解码情况是这样，在这之前的版本，似乎不需要做循环解码，结果就是ape无损码流各种漏帧。。。



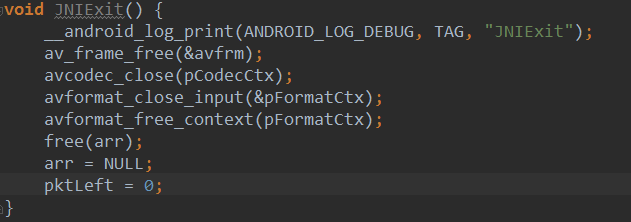
5、audioResampling是一个重新采样的函数，调用此方法的目的就是将二维平面数据格式转为一维线性数据格式，这种数据的处理方式对于mp3、ape格式是必须的，否则将出现严重的杂音，因为android自带的audiotrack支持的音频格式为PCM\_16BIT线性格式，而mp3、ape转换出来的数据为PCM\_16BIT平面格式，核心的转换函数为swr\_convert，在audioResampling函数中还会将转换后的数据拷贝到数组中。



6、最后将拿到的PCM数据返回给上层audiotrack



## 第四步：释放资源



# 阶段小结

FFmpeg神秘而庞大，目前只是用到了它的解码功能并且还只是皮毛而已，还有许多待研究的地方，比如网络音频缓冲进度、seek进度、以及效率最优化等，深入源代码才能明白这套框架其中的奥妙，另外它还提供了很多高效而实用的功能，待开发。。。