android NDK

**ABI**

不同的android手机使用不同的CPU，因为支持不同的指令集。

**NEON**

NENO 是浮点的协处理器，用来计算算法。

-mfpu= vfp（浮点协处理器）

-mpu= neon

-march = armv7-a

**🡪使用NDK编译FFmpeg的库**

1. 准备ubuntu开发环境

进入linux,

sudo passwd 密码

sudo apt-get update // 更新软件源

apt-get openssh-server // 下载openssh,这个工具是ssh的服务端

wget <https://dl.google.com> // 使用wget下载

使用xshell连接到linux主机

二.下载ffmpeg源码

git clone <https://git.ffmpeg.org/ffmpeg.git> ffmpeg, 使用git下载ffmpeg源码

wget <http://ffmpeg.org/releases/ffmpeg-3.4.tar.bz2>

<http://www.ffmpeg.club/ffmpeg_download.html>

解压编译ffmpeg

tar –xvf ffmpeg-3.4.tar.bz2

apt-get install make

./config // 配置makefile

make –j16 // 编译的线程数量(8核16线程就开了16线程)

make install

config文件配置

--prefix 输出目录

--enable 开启模块，硬解码，neon

--disable 禁止模块，禁止ffmpeg工具

交叉编译参数 cross-prefix arch target cpu sysroot extra-cflags

🡪创建支持FFmpeg的项目

◆权限

需要读写SD卡权限：WRITE\_EXTERNAL\_STORAGE

为了避免Android6.0动态权限检测，可以将targetSdkVersion = 21

◆JNI库的路径

build.gradle中添加

sourceSets {

main {

jniLibs.srcDirs = [‘libs’]

}

}

◆配置CMake参数

externalNativeBuild {

cmake {

cppFlags “-std=c++11”

path = “CMakeLists.txt”

}

ndk {

abiFilters “armeabi-v7a”// 只加载armv7版本

}

}

◆CMake项目配置app/CMakeLists.txt

include\_directories(include) 引用的头文件路径

-DANDROID-ABI = armeabi-v7a 设置FFmpeg的库路径

set (FF\_DIR${CMAKE\_CURRENT\_SOUCE\_DIR}/libs/${ANDROID\_ABI})

创建FF\_DIR这样一个变量，括号内的大写字母也是变量，语法格式：${变量名}

cmake添加动态库 add\_library

添加源码类型的库

add\_library(native-lib SHARED src/main/cpp/native-lib.cpp)

添加已经编好的so库(如ffmpeg)

add\_library(avformat) SHARED IMPORTED // 注意：avformat 并不是so库的文件名，仅是一个符号

set\_target\_properties(avformat PROPERTIES IMPORTED\_LOCATON${FF\_DIR}/libavformat.so)

◆CMake链接动态so库（将所有用到的库全部链接到一个库当中，这样做的好处是在java中只需要引入一个native-lib库就行了）

target\_link\_libraries(#specifies the target library.

native-lib

android

avformat avcodec swscale avtuil

)

新建项目

注意点：勾选支持

一.添加头文件路径。在native-lib.cpp中引入FFmpeg的头文件。

1.1 将编译生成的include目录添加到项目的app目录下。

1.2 在native-lib.cpp中添加下列代码：

extern “C”{

#include <libavcodec/avcodec.h>

}

// 必须添加extern “C”字样，c++对函数名有重载

// 还需要指定头文件路径，否则项目找不到指定的头文件。

1.3 在CMakelist.txt中添加头文件路径(相对于本文件的路径)。

include\_directories(include) // 指定路径的名称是include

1.4 调用函数测试ffmpeg是否可以正常工作

hello += avcodec\_configuration();

1.5 加载ffmpeg的库文件，否则如果调用avcodec\_configuration()会报函数未定义。

将编译生成的so库拷贝到 项目的app/libs/armeabi-v7a目录下。

1.6 在CMakeList.txt中添加FFMpeg库所在路径的变量

set {FF ${CMAKE\_CURRENT\_SOURCE\_DIR}/libs/${ANDROID\_ABI}}

add\_library(avcodec SHARED IMPORTED )

set\_target\_properties(avcodec PROPERTIES IMPORTED\_locaton ${FF} /libavcodec.so) // 与Android.mk不同，这里还需要加lib前缀和.so后缀

1.7 在CMakeList.txt中对将要添加的库进行链接

target\_Link\_librarise

native\_lib

avcodec

1.8 在app/build.gralde中defaultConfig这一层的externNativeBuild中添加

ndk {

abiFilters “armeabi-v7a”

}

1.9 指定指定JN目录

main {

jniLibs.srcDirs = [‘libs’]

}

2.0 测试JN函数

public native boolean Open(String url, Object handle)

FILE \*fp = foen(url, “rb”);

if (!fp) {

}

FFmpeg SDK 软硬件解码基础

**解封装**

pts 是显示的时间 dts是解码的时间， 这个时间是用来做同步

1. av\_register\_all(), 注册所有的格式。包括解封装格式和加封装格式。
2. avformat\_network\_init(), FFmpeg本身也支持解封装RTSP的数据,如果要解封装网络数据格式。
3. avformat\_open\_input()。打开一个文件，并解析，可解析的内容包括：视频流、音频流、视频流参数、音频流参数、视频帧索引。
4. avformat\_find\_stream\_info()，查找格式和索引。有些早期格式它的索引并没有放到头当中，需要你到后面探测，就会用到此函数。
5. av\_find\_best\_stream(), 当视频被解封装出来后，需要分开处理音频和视频，需要找到对应的音频流和视频流
6. 解封装后涉及到的结构体：

AVFormatContext: 封装的上下文，

AVStream : 存放的是音频流或视频流的参数信息

AVPacket: 针对于具体的解封装完后的一个一个的数据包。

av\_read\_frame() 用于读取一个AVPacket,AVPacket里面包含了这个包的pts,dts,还有这个包的stream index(它是音频还是视频),是否是关键帧，AVPacket把h264数据的间隔符去掉了，

**avformat\_open\_input() 函数详解并示例打开mp4文件。int avformat\_open\_input()**

打开一个输入的封装器。在调用该函数之前，须确保av\_register\_all()和avformat\_network\_init()已调用。

参数说明：

AVFormatContext \*\*ps, 格式化的上下文。要注意，如果传入的是一个AVFormatContext\*的指针，则该空间须自己手动清理，若该指针传空，则FFmpeg会内部自己创建。

const char \*url, 传入的地址。支持http,RTSP,普通的本地文件。地址最终会存入到AVFormatContext结构体当中。

AVInputFormat \*fmt, 一般不用，指定输入的封装格式，一般传NULL，由FFmpeg自行探测。

AVDictionary \*\*options, 字典数组。用于参数传递，不传则写NULL。参见：libavformat/options\_table.h,其中包含了它支持的参数设置。

代码参考：

#include <libavformat/avformat.h>

stringFromJNI() {

// 初始化解封装

av\_register\_all()

// 初始化网络

avformat\_network\_init()

// 打开文件

AVFormatContext \*ic = NULL;

char path[] = “sdcard/1080.mp4”;

int ret = avformat\_open\_input(&ic, path, 0, 0);

if (ret == 0) {

LOGI(“avformat input open success”);

} else {

LOGE(“avformat\_open\_input failed. %s”av\_err2str(ret;)

}

}

提示函数未声明：

在CMakeLists.txt中添加avcodec, 和 avutil。

==============================================================================

**AVFormatContext成员分析**

AVIOContext \*pb IO Context,, 自定义一些读写格式或者从内存当中读，

char filename[1024]; 文件名，它会把打开的文件名存下来(经常会用到断开重连的操作)

unsigned int nb\_streams 代表流的数量

AVStream \*\*streams 流，一般存放的是音频或者是视频数据的信息，正常情况下0是视频下标，1是音频下标

int64\_t duration 总长度。表示我们整个媒体文件的长度，以AV\_TIME\_BASE这个时间基数为单位，1000000，实际上它是一个微秒值，除以1000就是毫秒，这个值可能无法获取到。

int64 bit\_rate 比特率

void avformat\_close\_input(AVFormatContext \*\*s) 关闭输入的上下文

**avformat\_find\_stream\_info**

**第一个参数是ic的指针，第二个参数传NULL，主要用于做探测。**

获取视频流信息。在一些格式当中没有头部信息，如flv格式，h264格式，这个时候调用avformat\_open\_input()在打开之后就没有参数，也就无法获取到里面的信息。这个时候就可以调用此函数，因为它会试着去探测文件的格式，但是如果格式当中没有头部信息，那么它只能获取到编码、宽高这些信息。还是无法获得总时长。如果总时长无法获取到，则仍需要把整个文件读一遍，计算一下它的总帧数。

读取流的信息和流的数量

flv 格式无法读取到，

**AVStream**

AVCodecContext \*codec; // 过时了，以它作解码器

AVRational time\_base // 时间基数，其实就是一个分数，因为用浮点数运算有精度损失,

int64\_t duration; // 转换成毫秒，duration\*( time\_base.num / time\_base.dem) 参看ppt,总时长，有时候不一定有,不可靠

AVRational avg\_frame\_rate, 帧率，同样是用分数表示

AVCodecParameters \*codecpar; // 音视频参数 （宽度，高度）用来替代上面的AVCodecContext

AVCodecParameters 详解

enum AVMediaType codec\_type; // 编码类型（音频/视频）

enum AVCodecID codec\_id // 编码格式 (内部是一个枚举h264，mpeg4, mjpeg)

uint32\_t codecTag; // 可以不用

int format; // 格式。对于视频来说指的是像素格式，对于音频来说，是音频的采样格式

int width, int height; // 视频的宽高，只有视频有

uint64\_t channel\_layout; // 取默认值即可

int channels; // 声道数

int sample\_rate; // 样本率

int frame\_size; // 只针对音频，一帧音频的大小

打印音视频对应的格式

static double

通过av\_find\_best\_stream获取音视频流索引

av\_find\_best\_stream

AVFormatContext \*ic打开的上下文

enum AVMediaType type // 我们要获取的类型。是获取音频流信息还是视频流信息。

int wanted\_stream\_nb // 指定流的信息，取哪一个流。可以传-1

int related\_stream // 相关的流信息

AVCodec \*\*decoder\_ret // 最终的解码器信息，传NULL即可

int flags // 暂时不用

**av\_read\_frame**

读取具体的音视频帧数据

AVFormatContext \*s // 文件格式上下文

AVPacket \*pkt // 这个值不能传NULL，必须是一个空间，

return 0 is OK, <0 on error or end of file

**AVPacket**

AVBufferRef buf; // 指向一个buf空间，该空间是用来存储引用计数的。有可能多个AVPacket指向的都是这个buf的空间。因此要在每次拷贝的时候对其进行+1，删除的时候对其进行-1.

int64\_t pts // 显示时间。是一个long long 的类型。因为它有时候是用微秒来做的。

int64\_t des // 解码时间。同上。

uint8\_t data; int size // 指向的是AVBufferRef里面再分配的空间。该空间无法手动删除，因为是在动态库中创建的，需要调用FFmpeg提供的接口来删除。

**AVPacket相关函数**

AVPacket \*av\_packet\_alloc(void) // AVPacket空间的创建和初始化。创建一个AVPacket对象，它会在堆上面申请空间，因此还需要去手动释放。

AVPacket \*av\_packet\_clone(const AVPacket \*src); // AVPacket空间的复制。创建并应用引用计数， 分配空间&增加引用

int av\_packet\_ref(AVPacket \*dst, const AVPacket \*src); // 手动对AVPacket的引用计数+1

int av\_packet\_unref(AVPacket \*pkt);

void av\_packet\_free(AVPacket \*\*pkt); // 清空对象并减引用计数

void av\_init\_packet(AVPacket \*pkt); // 默认值。方便初始化的一个接口，如果只是在栈中或者是new了一个对象，可以利用此函数进行初始化。

int av\_packet\_from\_data(AVPacket \*pkt, uint8\_t \*data, int size); // 分配空间的函数。如果创建了一个AVPacket,需要分配空间的话，可以调用该函数来分配空间。

int av\_copy\_packet(AVPacket \*dst, const AVPacket \*src); // 早期函数，已不推荐使用。

**av\_seek\_frame**

当滑动条拖动的时候，我们需要做一个seek操作，也就是说我们要把它移动到相应的位置。此时就用到了这个av\_seek\_frame()函数。

AVFormatContext \*s // 封装格式上下文

int streamIndex // -1 default。流索引。你的媒体文件中既包含视频又包含音频，可以指定究竟是视频来移还是音频来移。-1默认一般是视频。针对于视频，必须要移到关键帧的位置。所以一般要选视频要做seek。

int64\_t timestamp. // 时间戳。你要移动到哪个时间位置。

int flag // 标识位。表示我们移的方法。

av\_seek\_frame flag

#define AVSEEK\_FLAG\_BACKGROUND 1 ///<<Seek Background 往后走,

#define AVSEEK\_FALG\_BYTE ///<<<seeking based on position in bytes

#define AVSEEK\_FLAG\_ANY ///<<<seek to any frame, even non-keyframes // 任意帧

#define AVSEEK\_FLAG\_FRAME ///<<<seeking based on frame number // 找关键帧，一般与AVSEEK\_FLAG\_BACKGROUND一起用