[Android audio and video development and upgrade: FFmpeg audio and video codec] 2. Android introduces FFmpeg

简 jianshu.com/p/2c9918546edc

In this article you can learn

> This article will introduce how to introduce the `FFmpeg so` library into the `Android` project and verify `so` whether it can be used normally.

1. Enable Android native C/C++ support

In the past, the `makefile` usual way of introducing `C/C++` code `Android Studio` support into a project `makefile` has been largely `CMake` replaced .

With `Android` official support, `NDK` the development of layer code becomes easier. When I talked about it before `Android NDK` , many people will be shocked and feel that it is something unfathomable. On the one hand, it `makefile` is difficult to write, and on the other hand, it is relatively obscure compared to `C/C++` . `Java`

But don't worry, one is there `CMake` , and the other is that `C/C++` the basic usage of is almost `Java` the same as that of . This series involves the basic usage of . `C/C++` hahaha~~

1. Install CMake

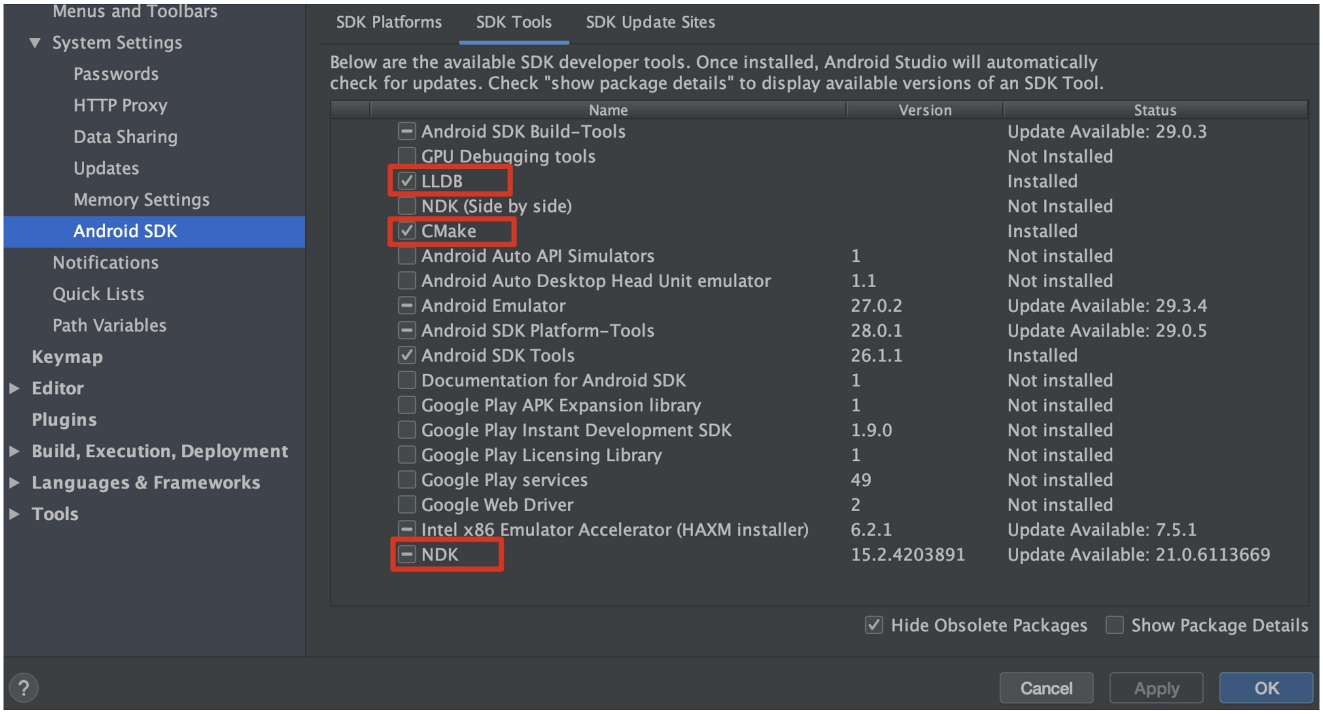
First, you need to download `CMake` related tools, `Android Studio` click in turn in `Tools->SDK Manager->SDK Tools` , and then tick

`CMake` : CMake build tool

`LLDB` : C/C++ code debugging tool

`NDK` : NDK environment

Finally, click `OK->OK->Finish` one by one to start the download (the file is relatively large and may be slow, please wait patiently).



Download CMake Tools

2. Add C/C++ support

There are two ways:

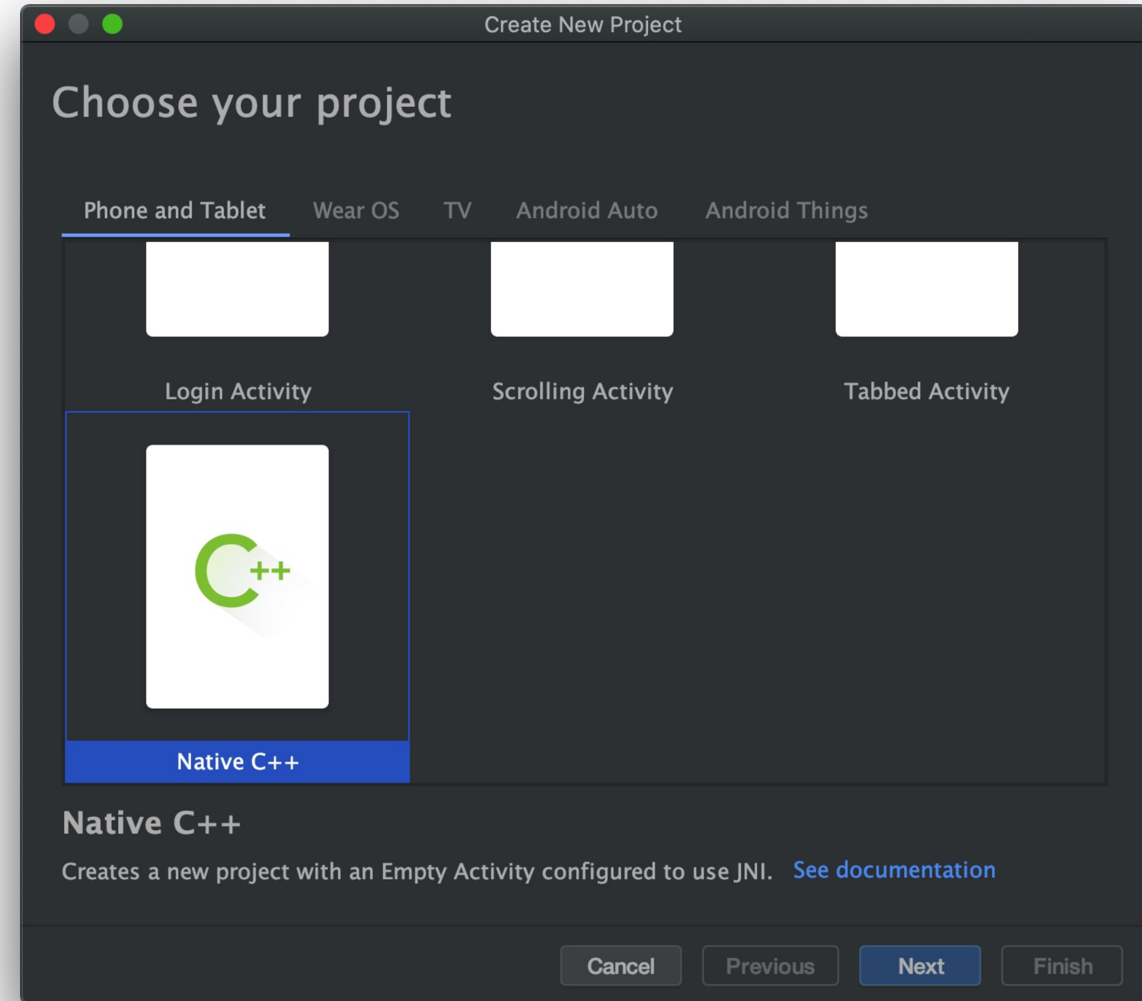
> First, create a new project and check the `C/C++` Support option, the system will automatically create a `C/C++` project that supports encoding.

> Second, on an existing project, manually adding all the additions to support `C/C++` coding is actually the things that are automatically created for `「第一种方式」` us in the manual addition. `Android Studio`

First, by creating a new project, let's see what is generated `IDE` for us.

1) Create a new C/C++ project

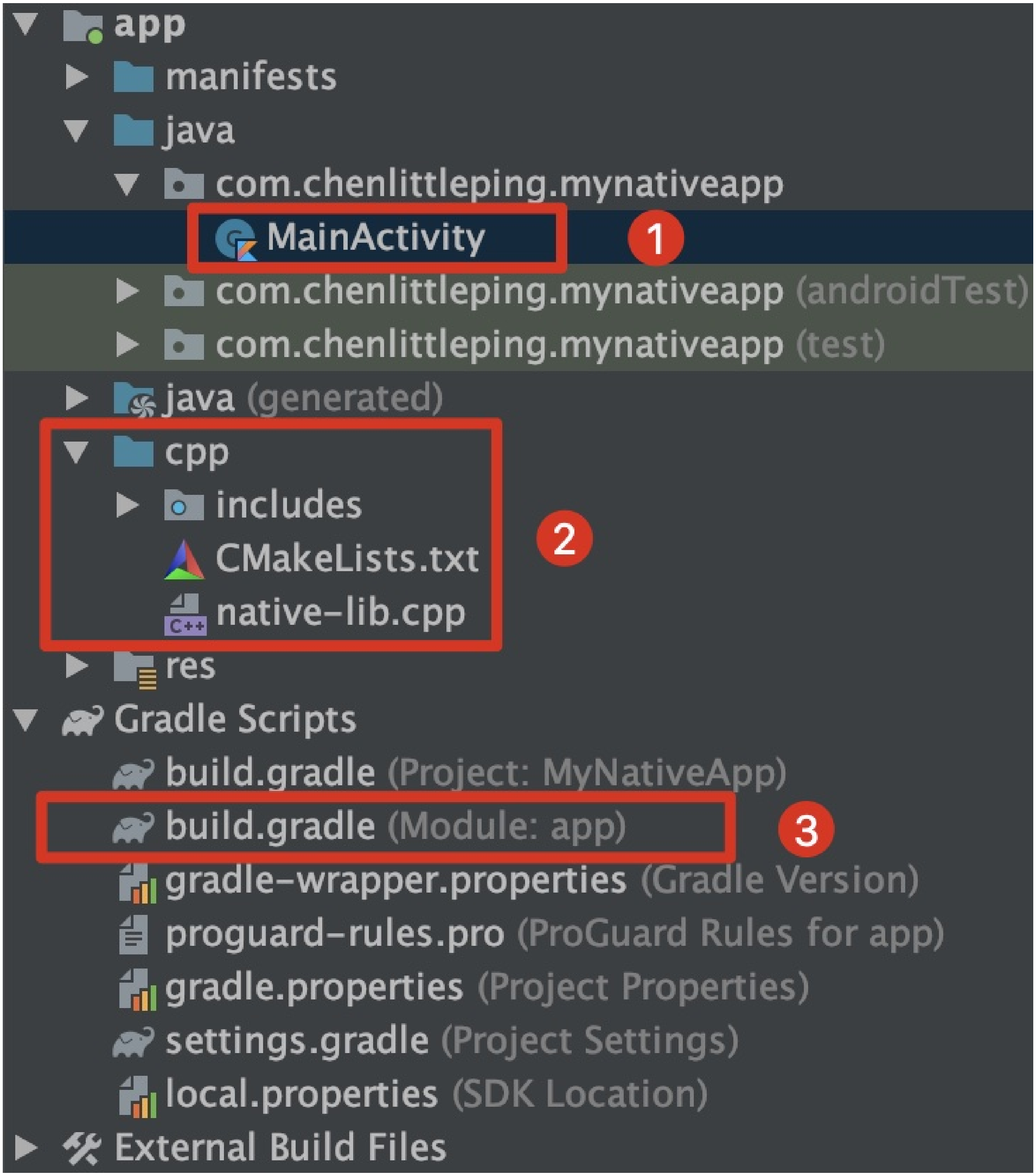
Click one by one `File -> New -> New Project` to enter the new project page, pull to the end, select `Native C++` and then follow the default configuration, all the way `Next -> Next -> Finish` .



Create a new C++ project

2) What does Android Studio automatically generate?

The generated project directory is as follows:



Project directory

Focus on the 3 places marked in the picture above:

> **First, the top** `MainActivity`

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Example of a call to a native method
        sample_text.text = stringFromJNI()
    }

    /**
     * A native method that is implemented by the 'native-lib' native library,
     * which is packaged with this application.
     */
    external fun stringFromJNI(): String

    companion object {

        // Used to load the 'native-lib' library on application startup.
        init {
            System.loadLibrary("native-lib")
        }
    }
}
```

It's very simple. Anyone who has used the `so` library should understand it. Let me briefly talk about it here.

At the bottom of the code, in `companion object` represents a static code block, similar to in , the code in it has and will only be executed once. `Kotlin` `Java` `static { }`

Then in the `init{}` method , `C/C++` the `so` library loaded: `native-lib` .

In the previous code, an external reference method is `external` declared with `stringFromJNI()` , and this method corresponds to the code of the `C/C++` layer .

Finally in the topmost `onCreate` , `C/C++` returned from the layer is `String` displayed .

Second, create `cpp` a package

Among them, two files are very important, namely `native-lib.cpp` , `CMakeLists.txt` .

i. `native-lib.cpp` : is a C++ interface file `MainActivity` where external methods declared in will be implemented.

`native-lib.cpp` The content automatically generated is as follows:

```
#include <jni.h>
#include <string>

extern "C" JNIEXPORT jstring JNICALL
Java_com_chenlittleping_mynativeapp_MainActivity_stringFromJNI(
        JNIEnv *env,
        jobject /* this */) {
    std::string hello = "Hello from C++";
    return env->NewStringUTF(hello.c_str());
}
```

As you can see, the method names in this cpp file are very long, but they are actually very simple.

The first is the fixed writing of the head `extern "C" JNIEXPORT jstring JNICALL` :

`extern "C"` means to compile in `C语言` the way;

`jstring` Indicates that the return type of the method is the type of `Java` layer `String` , similar to: `void` `jint` etc.;

Then there is the mapping of the corresponding method of the Java layer , that is, the entire method name is actually the absolute path of the corresponding method of the `Java` layer .

Among them, the first `Java_` is the fixed writing method;

`com_chenlittleping_mynativeapp_MainActivity_` : Correspondingly `com.chenlittleping.mynativeapp.MainActivity.` , it is actually `.` replaced by `_` ;

`stringFromJNI` Consistent with the approach of the Java layer.

Finally there are two parameters , `JNIEnv *env` and `jobject` , which represent `JNI` the context of and `Java` the instance of the class that calls this interface.

Calling this method will create a string at the `C++` layer and `Java#String` return it as type .

ii. **`CMakeLists.txt`** : That is, the build script. The content is as follows:

```
# cmake 最低版本
cmake_minimum_required(VERSION 3.4.1)

# 配置so库编译信息
add_library(
        # 输出so库的名称
        native-lib

        # 设置生成库的方式，默认为SHARE动态库
        SHARED

        # 列出参与编译的所有源文件
        native-lib.cpp)

# 查找代码中使用到的系统库
find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)

# 指定编译目标库时，cmake要链接的库
target_link_libraries(
        # 指定目标库，native-lib 是在上面 add_library 中配置的目标库
        native-lib

        # 列出所有需要链接的库
        ${log-lib})
```

This is the simplest build configuration, see the comments above for details.

`CMakeLists.txt` `native-lib` The purpose is to configure the build information that can compile the so library.

To put it bluntly, it is to tell the compiler:

```
- 编译的目标是谁
- 依赖的源文件在哪里找
- 依赖的 `系统或第三方` 的 `动态或静态` 库在哪里找。
```

Third, register the CMake script in the Gradle file

`第二步` In , `so` the information for building the library has been configured, and then the information must be registered `Gradle` in , and the compiler will compile it.

The `build.gradle` content as follows:

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 28
    buildToolsVersion "29.0.1"
    defaultConfig {
        applicationId "com.chenlittleping.mynativeapp"
        minSdkVersion 19
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"

        // 1) CMake 编译配置
        externalNativeBuild {
            cmake {
                cppFlags ""
            }
        }
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
        }
    }

    // 2) 配置 CMakeLists 路径
    externalNativeBuild {
        cmake {
            path "src/main/cpp/CMakeLists.txt"
            version "3.10.2"
        }
    }
}

dependencies {
    // 省略无关代码
    //......
}
```

The two main places are two `externalNativeBuild` .

In the first **`externalNativeBuild`** one , you can do some optimized configuration, such as only packaging `armeabi` the architecture that includes `so` :

```
externalNativeBuild {
    cmake {
        cppFlags ""
    }
    ndk {
        abiFilters  "armeabi" //, "armeabi-v7a"
    }
}
```

The second one `externalNativeBuild` is mainly `CMakeLists.txt` the .

> `Android Studio` The `C/C++` support is mainly the above three places. With the above configuration, it can be `MainActivity` displayed normally on the page `Hello from C++` .

3) Add `C/C++` support

As mentioned earlier, adding `C/C++` support adding the entire configuration manually by ourselves. Then according to the three steps introduced by the signature, follow the gourd and draw the scoop, and you can add it.

Here is just a demonstration of adding `FFMpeg so` it to the existing Demo project of this series of articles.

2. Introducing FFmpeg so

1. Create a new cpp directory

First, in the `app/src/main/` directory , create a new folder and name it `cpp` .

Next, in the `cpp` directory , right-click `New -> C/C++ Source File` and create a new `native-lib.cpp` file .

Next, in the `cpp` directory , right-click `New -> File` , create a new `CMakeLists.txt` one, and paste the code `IDE` generated . The configuration of FFmpeg will be explained in detail later.

```
# CMakeLists.txt

# cmake 最低版本
cmake_minimum_required(VERSION 3.4.1)

# 配置so库编译信息
add_library(
        # 输出so库的名称
        native-lib

        # 设置生成库的方式，默认为SHARE动态库
        SHARED

        # 列出参与编译的所有源文件
        native-lib.cpp)

# 查找代码中使用到的系统库
find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)

# 指定编译目标库时，cmake要链接的库
target_link_libraries(
        # 指定目标库，native-lib 是在上面 add_library 中配置的目标库
        native-lib

        # 列出所有需要链接的库
        ${log-lib})
```

2. Configure CMakeLists into build.gradle

```
android {
    // ...

    defaultConfig {
    // ...

    // 1) CMake 编译配置
    externalNativeBuild {
            cmake {
                cppFlags ""
            }
        }
    }

    // ...

    // 2) 配置 CMakeLists 路径
    externalNativeBuild {
        cmake {
            path "src/main/cpp/CMakeLists.txt"
            version "3.10.2"
        }
    }
}

// ...
```

If you just simply write `C/C++` code , the above basic configuration is enough.

Then let's take a look at the focus of this article, how to `CMakeLists.txt` use `FFmpeg` the imported dynamic library.

3. Put the FFmpeg so library in the corresponding CPU architecture directory

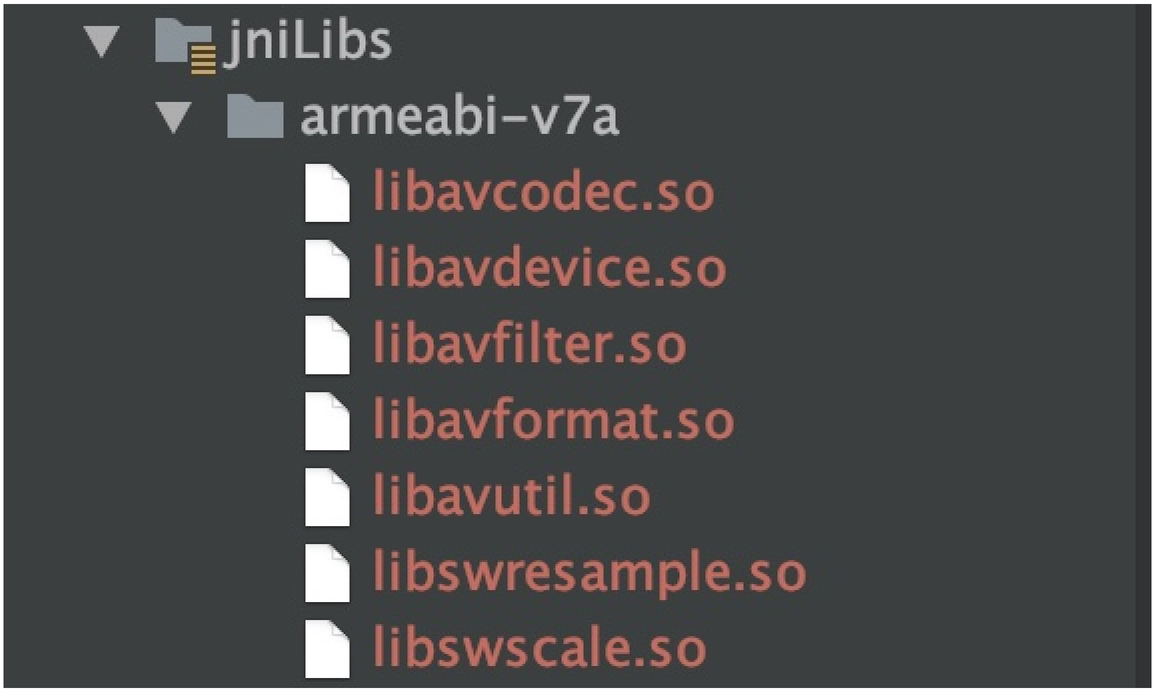
In the previous article, the architecture of the `FFmpeg` `so` library was , so, we need to put all the libraries into the directory. `CPU` `armv7-a` `so` `armeabi-v7a`

First, in the `app/src/main/` directory , create a new folder and name it `jniLibs` .

> `app/src/main/jniLibs` It is the default directory of Android Studio where the so dynamic library is placed.

Next, under the `jniLibs` directory , create a new `armeabi-v7a` directory.

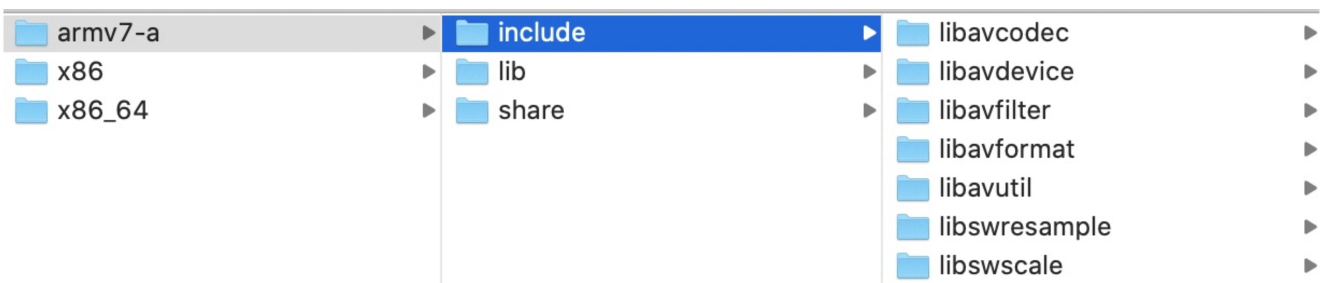
`FFmpeg` Finally `so` paste all the compiled libraries into the `armeabi-v7a` directory . as follows:



so directory

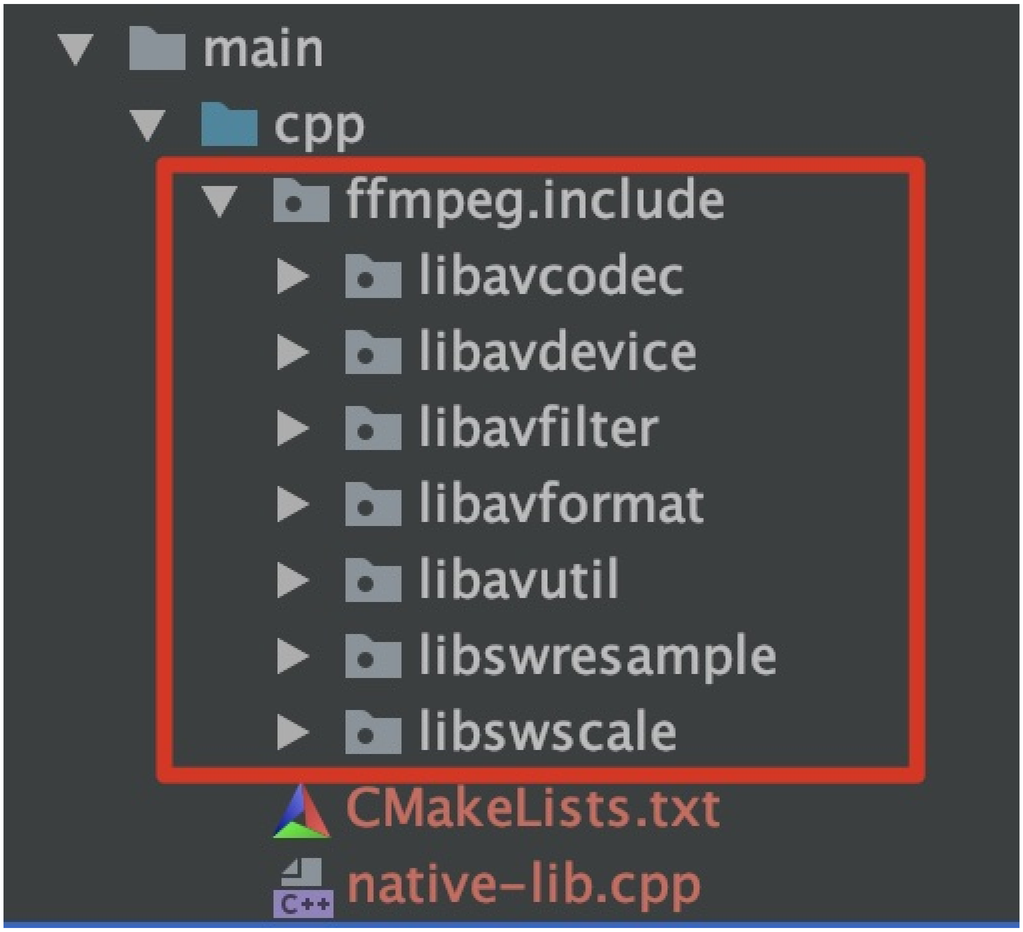
4. Add the header file of FFmpeg so

When `FFmpeg` compiling , in addition to generating `so` , it will also generate the corresponding `.h` header files, that is `FFmpeg` , all the interfaces exposed to the outside world.



FFmpeg compile output

Under the `cpp` directory , create a new `ffmpeg` directory and paste the `include` files into it.



header file directory

5. Add and link FFmpeg so library

The above has placed `so` and `头文件` into the corresponding directory, but the compiler will not compile, link, and package them `Apk` into . We also need `CMakeLists.txt` to explicitly `so` add in . The complete is `CMakeLists.txt` as follows :

```
cmake_minimum_required(VERSION 3.4.1)

# 支持gnu++11
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=gnu++11")

# 1. 定义so库和头文件所在目录，方面后面使用
set(ffmpeg_lib_dir ${CMAKE_SOURCE_DIR}/../jniLibs/${ANDROID_ABI})
set(ffmpeg_head_dir ${CMAKE_SOURCE_DIR}/ffmpeg)

# 2. 添加头文件目录
include_directories(${ffmpeg_head_dir}/include)

# 3. 添加ffmpeg相关的so库
add_library( avutil
        SHARED
        IMPORTED )
set_target_properties( avutil
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libavutil.so )

add_library( swresample
        SHARED
        IMPORTED )
set_target_properties( swresample
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libswresample.so )

add_library( avcodec
        SHARED
        IMPORTED )
set_target_properties( avcodec
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libavcodec.so )

add_library( avfilter
        SHARED
        IMPORTED)
set_target_properties( avfilter
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libavfilter.so )

add_library( swscale
        SHARED
        IMPORTED)
set_target_properties( swscale
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libswscale.so )

add_library( avformat
        SHARED
        IMPORTED)
set_target_properties( avformat
```

```
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libavformat.so )

add_library( avdevice
        SHARED
        IMPORTED)
set_target_properties( avdevice
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libavdevice.so )

# 查找代码中使用到的系统库
find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log )

# 配置目标so库编译信息
add_library( # Sets the name of the library.
        native-lib

        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        native-lib.cpp
        )

# 指定编译目标库时，cmake要链接的库
target_link_libraries(

        # 指定目标库，native-lib 是在上面 add_library 中配置的目标库
        native-lib

# 4. 连接 FFmpeg 相关的库
        avutil
        swresample
        avcodec
        avfilter
        swscale
        avformat
        avdevice

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib} )
```

Mainly look at the newly added `1~4` points .

1) `set` The and `so` directory are defined by the method, which is convenient for later use. `头文件`

> where `CMAKE_SOURCE_DIR` is a system variable, pointing to `CMakeLists.txt` the directory where is located. `ANDROID_ABI` It is also a system variable, pointing to the `CPU` framework : armeabi, armeabi-v7a, x86...

```
set(ffmpeg_lib_dir ${CMAKE_SOURCE_DIR}/../jniLibs/${ANDROID_ABI})
set(ffmpeg_head_dir ${CMAKE_SOURCE_DIR}/ffmpeg)
```

2) Find the directory by `include_directories` setting the header file

```
include_directories(${ffmpeg_head_dir}/include)
```

3) By `add_library` adding FFmpeg-related `so` libraries, and `set_target_properties` setting `so` the corresponding directory.

> Among them, the first parameter of add_library is the so name, `SHARED` indicating that the import method is dynamic library import.

```
add_library( avcodec
        SHARED
        IMPORTED )
set_target_properties( avcodec
        PROPERTIES IMPORTED_LOCATION
        ${ffmpeg_lib_dir}/libavcodec.so )
```

4) Finally, `target_link_libraries` link the previously added `FFMpeg so` libraries to the target `native-lib` library .

In this way, we have introduced the `FFMpeg` relevant `so` libraries into the current project. Next, let's test whether the `FFmpeg` related .

3. Use FFmpeg

To check `FFmpeg` if it can be used, you can `FFmpeg` verify by getting the basic information.

1. Add an external method ffmpegInfo to FFmpegAcrtivity

Display the obtained `FFmpeg` information .

```
class FFmpegActivity: AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_ffmpeg_info)

        tv.text = ffmpegInfo()
    }

    private external fun ffmpegInfo(): String

    companion object {
        init {
            System.loadLibrary("native-lib")
        }
    }
}
```

2. Add the corresponding JNI layer method in native-lib.cpp

```
#include <jni.h>
#include <string>
#include <unistd.h>

extern "C" {
    #include <libavcodec/avcodec.h>
    #include <libavformat/avformat.h>
    #include <libavfilter/avfilter.h>
    #include <libavcodec/jni.h>

    JNIEXPORT jstring JNICALL
    Java_com_cxp_learningvideo_FFmpegActivity_ffmpegInfo(JNIEnv *env, jobject  /*
this */) {

        char info[40000] = {0};
        AVCodec *c_temp = av_codec_next(NULL);
        while (c_temp != NULL) {
            if (c_temp->decode != NULL) {
                sprintf(info, "%sdecode:", info);
                switch (c_temp->type) {
                    case AVMEDIA_TYPE_VIDEO:
                        sprintf(info, "%s(video):", info);
                        break;
                    case AVMEDIA_TYPE_AUDIO:
                        sprintf(info, "%s(audio):", info);
                        break;
                    default:
                        sprintf(info, "%s(other):", info);
                        break;
                }
                sprintf(info, "%s[%10s]\n", info, c_temp->name);
            } else {
                sprintf(info, "%sencode:", info);
            }
            c_temp = c_temp->next;
        }
        return env->NewStringUTF(info);
    }
}
```

First, we see that the code is wrapped in `extern "C" { }`, which is slightly different from the one created by the previous system. By wrapping it in curly braces, we don't need to add a separate `extern "C"` opening for each method.

In addition, since it `FFmpeg` is `C` written in the language `C++`, `#include` when it is referenced in the file, it also needs to be wrapped in `extern "C" { }` order to compile correctly.

Needless to say, the new method is the same as the naming method introduced earlier.

In the method, use the method `FFmpeg` provided by `av_codec_next` to get the codec of FFmpeg, and then through the loop, concatenate all the audio and video codec information, and finally return it to the `Java` layer .

At this point, it `FFmpeg` is added to the project and called.

If everything is normal, after the App runs, it will display the information of the `FFmpeg` audio and video codec.

> If prompted by `so` or `头文件` cannot be found, you need to `CMakeLists.txt` check whether the and `so` paths set in are correct. `头文件`