【Android 音视频开发打怪升级：FFmpeg音视频编解码篇】一、FFmpeg so库编译 - 简书

[Android audio and video development and upgrade: FFmpeg audio and video codec] 1. FFmpeg so library compilation

In this article you can learn

> Use `GCC` or `CLANG` to cross compile the FFmpeg so library that the Android platform can use. In order to take the first step of `FFmpeg` development , we must not only know what it is, but also know why it is. Not only to know how to compile successfully, but also to know why it compiles successfully. Before starting, it is recommended to read the entire article first, I believe this article will give you some insight.

I. Introduction

In fact, there are already a lot of sharing about FFmpeg so library compilation on the Internet, but most of them directly post the content of the configuration file. I think most people who search for "how to compile FFmpeg so library" are relatively unfamiliar with cross-compilation.

Especially for mobile developers, most people develop in the Java layer most of the time, and rarely touch the NDK layer. If you directly look at a cross-compiled configuration, it is estimated that it will be very high.

Usually, under an article compiled by FFmpeg, there will be a lot of comments like "Why can't it be compiled successfully according to the landlord's configuration?" So why can others compile successfully, but we can't copy it?

There are many reasons, most of them are actually concentrated in the following aspects:

1. 无脑copy，祈求有一个傻瓜式的配置可以成功编译；
2. FFmpeg版本和NDK版本很多，每一个版本都可能需要不一样的配置；
3. 不了解每个配置项的意义，即使好运配置对了， 但是稍微一修改，又无法正常编译了。

Why does FFmpeg feel so hard to do?

I think the main reason is that it is very difficult to take the first step, and even the so library cannot be compiled, and the rest is nonsense.

2. What is cross compilation

definition

> The definition quoted from Baidu Encyclopedia: Cross-compilation is the generation of executable code on one platform on another platform.

What does that mean? To put it bluntly, it is to generate a program on one machine, and this program can run on another machine. Example: Compile an apk on a PC, and this apk can run on an Android phone. This is actually a cross-compilation process.

Why cross-compile

We know that the software on the PC is compiled and generated directly on the PC, so why can't the software on Android be compiled and generated by itself on Android?

It is theoretically possible, but the resources on Android phones are limited. It takes so long to compile an apk on a PC. Can you imagine how long it takes to compile an apk on an Android phone? Or can you imagine typing a code on your phone?

Then we will think that since there are so many resources on the PC, can we use the PC to compile software that can run on the mobile phone?

Thus, cross-compilation appears.

What is needed for cross-compiling

Compiler Environment

We know that the environment on the PC and the running environment on the mobile phone are completely different. If you use the environment on the PC to compile directly, it is conceivable that the compiled App will hang in minutes.

Therefore, the most important thing for cross-compilation is to configure the relevant environment used in the compilation process, and this environment is actually the running environment of the target machine (such as an Android phone).

build toolchain

For C/C++ compilation, there are usually two tools `GCC` and `CLANG` .

`GCC` You may have heard that this is an old-fashioned compilation tool that can not only compile C/C++, but also Java, Object-C, Go and other languages.

`CLANG` It is a more efficient C/C++ compilation tool and is compatible with GCC. Google began to recommend the use of clang for compilation a long time ago, and `ndk 17` later , `GCC` removed and fully implemented the use `CLANG` .

3. How to cross compile FFmpeg

what is FFmpeg

The well-known FFmpeg, not to mention that it is very popular in the audio and video industry, even a developer who does not develop audio and video has heard a little.

Official Profile

> A complete, cross-platform solution to record, convert and stream audio and video.

The translation is: FFmpeg is a complete cross-platform solution for recording, converting and streaming audio and video.

From this introduction, we can see that FFmpeg has the following characteristics:

1. Powerful functions: recording, decoding, encoding, editing, streaming, etc.
2. Cross-platform

Compilation process

From the previous introduction, we can basically summarize the basic process of FFmepg compilation:

1. Choose build tool
2. Configure the cross-compilation environment
3. Configure compilation parameters (such as removing some unneeded features)
4. start compilation

The process is so simple, let's take a look at it in detail, `CLANG` how `GCC` to compile through and two ways.

Fourth, use CLANG to compile FFmpeg

> Note: The compilation platform of this article is Mac. It is recommended to use Mac or Linux for compilation. It is said that Windows has many pits.

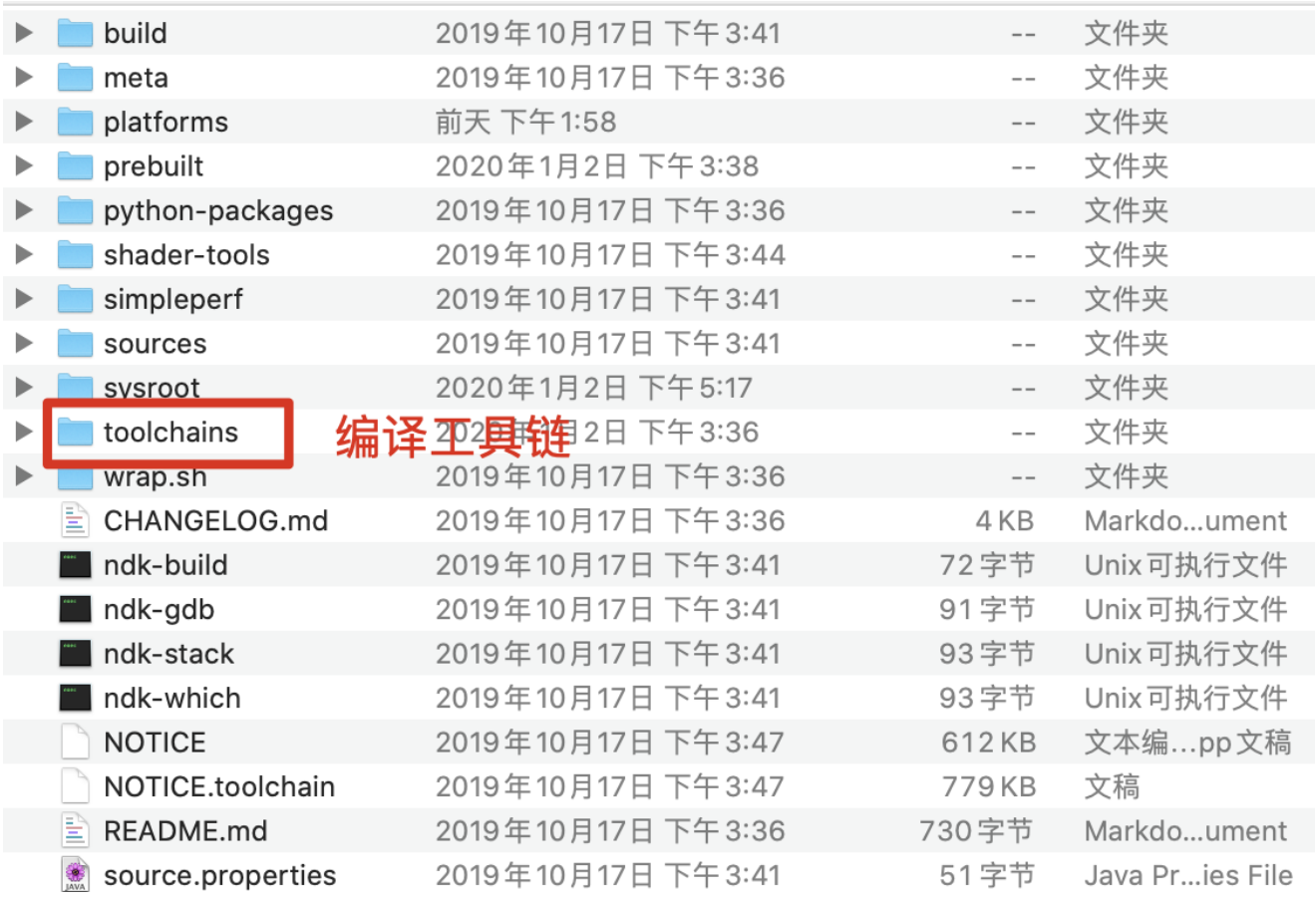
Download Android NDK

Android `NDK` has been iterated for many versions. After `r17c` that, Google officially removed `GCC` it and no longer supports `GCC` it. The new version `NDK` is `CLANG` compiled with .

Here we use the latest `NDK r20b` version to compile.

`NDK` Download address: Android-NDK

NDK directory



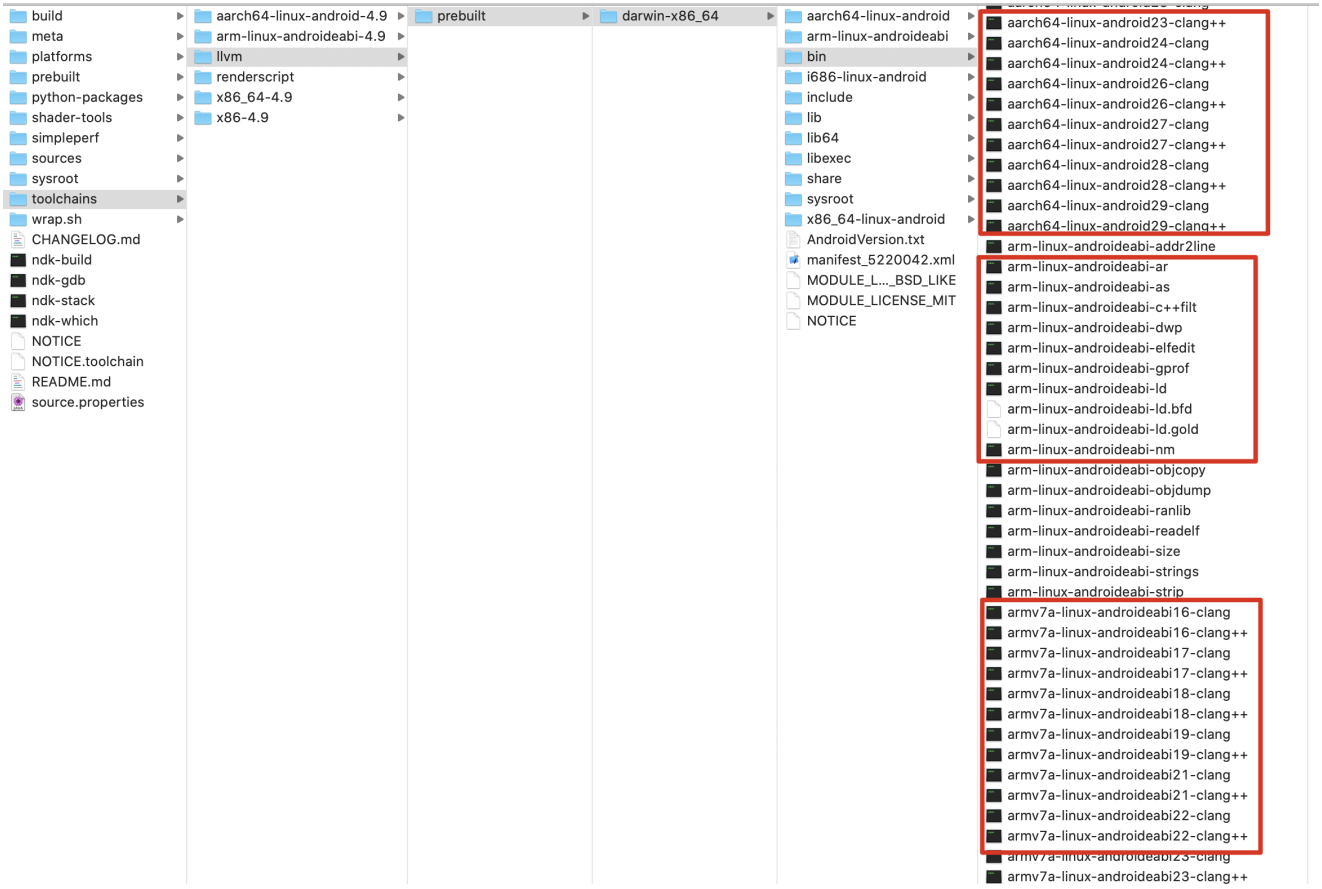
NDK r20b directory

The most important ones are these two paths:

编译工具链目录：
`toolchains/llvm/prebuilt/darwin-x86_64/bin`

交叉编译环境目录：
`toolchains/llvm/prebuilt/darwin-x86_64/sysroot`

> build tool path



build tool

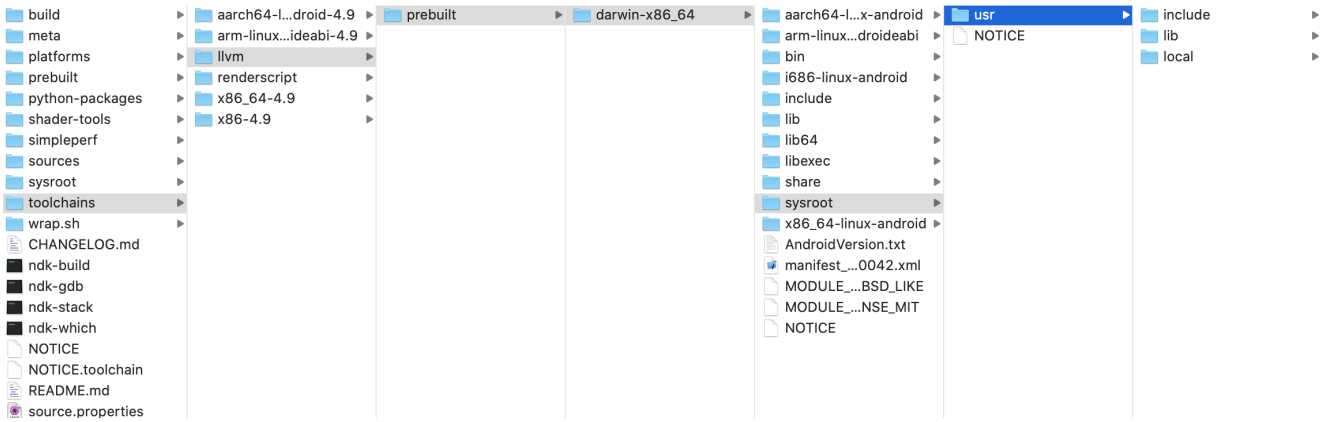
According to different CPU architecture areas and different Android versions, different clang tools are distinguished, just choose according to your own needs.

This article selects the CPU architecture `armv7a`, Android version `21`:

```
armv7a-linux-androideabi21-clang
armv7a-linux-androideabi21-clang++
```

build environment path

Under the `toolchains/llvm/prebuilt/darwin-x86_64/sysroot` directory, there are two directories: `usr/include`, `usr/lib`, corresponding to `头文件` and `库文件`.



Libraries and header files

Download FFmpeg source code

FFmpeg official website to download , just DownLoad directly.

This article uses the latest version `ffmpeg-4.2.2` .

After downloading the source code, go to the root directory and find `congfigure` a , which is a shell script used to generate some `FFmpeg` configuration files required for compilation.

> This file is very important, and `FFmpeg` the compilation configuration is done by it.
> Later we will analyze some of the important content, which is the key to understanding the `FFmpeg` compilation configuration.

With the above foundation, you can compile FFmpeg.

layout script

Modify the configure script

1. Add `cross_prefix_clang` parameter

Open (note: not double-click to run) `ffmpeg-4.2.2` the `configure` file , search `CMDLINE_SET` , you can find the following code, and then add a command line option: `cross_prefix_clang`

```
CMDLINE_SET="
    $PATHS_LIST
    ar
    arch
    as
    assert_level
    build_suffix
    cc
    objcc
    cpu
    cross_prefix
    # 新增命令行参数
    cross_prefix_clang
    custom_allocator
    cxx
    dep_cc
    # 省略其他.....
"
```

2. Modify the build tool path settings

Search `ar_default="${cross_prefix}${ar_default}"` , find the following code

```
ar_default="${cross_prefix}${ar_default}"
cc_default="${cross_prefix}${cc_default}"
cxx_default="${cross_prefix}${cxx_default}"
nm_default="${cross_prefix}${nm_default}"
pkg_config_default="${cross_prefix}${pkg_config_default}"
```

Change the middle two lines to

```
ar_default="${cross_prefix}${ar_default}"
#-----------------------------------------------
cc_default="${cross_prefix_clang}${cc_default}"
cxx_default="${cross_prefix_clang}${cxx_default}"
#-----------------------------------------------
nm_default="${cross_prefix}${nm_default}"
pkg_config_default="${cross_prefix}${pkg_config_default}"
```

As for why this modification is made, it will be explained in detail in the subsequent `configure` analysis .

Create a new build configuration script

Create a new script in the `ffmpeg-4.2.2` root directory `shell` and name it: `build_android_clang.sh`

```
#!/bin/bash
set -x
# 目标Android版本
API=21
CPU=armv7-a
#so库输出目录
OUTPUT=/Users/cxp/Desktop/FFmpeg/ffmpeg-4.2.2/android/$CPU
# NDK的路径，根据自己的NDK位置进行设置
NDK=/Users/cxp/Desktop/FFmpeg/android-ndk-r20b
# 编译工具链路径
TOOLCHAIN=$NDK/toolchains/llvm/prebuilt/darwin-x86_64
# 编译环境
SYSROOT=$TOOLCHAIN/sysroot

function build
{
  ./configure \
  --prefix=$OUTPUT \
  --target-os=android \
  --arch=arm \
  --cpu=armv7-a \
  --enable-asm \
  --enable-neon \
  --enable-cross-compile \
  --enable-shared \
  --disable-static \
  --disable-doc \
  --disable-ffplay \
  --disable-ffprobe \
  --disable-symver \
  --disable-ffmpeg \
  --sysroot=$SYSROOT \
  --cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
  --cross-prefix-clang=$TOOLCHAIN/bin/armv7a-linux-androideabi$API- \
  --extra-cflags="-fPIC"

  make clean all
  # 这里是定义用几个CPU编译
  make -j12
  make install
}

build
```

This shell script is generally very easy to understand, such as

`--disabble-static` `--enable-shared` They are respectively used to prohibit the output of static libraries and the output of dynamic libraries;

`--arch` `--cpu` What is the architecture of the so library used to configure the output;

`--prefix` The storage path of the so library used for configuration output.

Let's focus on a few options:

target-os

`--target-os=android` : In the old version `FFmpeg` of , the support for the Android platform is not very complete, and there is no `android` such target, so it will be mentioned in some older articles that to compile the so library of the Android platform, the following modifications need to `configure` be made, otherwise the following changes will be made. `linux` The standard

```
SLIBNAME_WITH_VERSION='$(SLIBNAME).$(LIBVERSION)'
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)'
```

修改为：

```
SLIBNAME_WITH_VERSION='$(SLIBNAME).$(LIBVERSION)'
SLIBNAME_WITH_MAJOR='$(SLIBPREF)$(FULLNAME)-$(LIBMAJOR)$(SLIBSUF)'
LIB_INSTALL_EXTRA_CMD='$$(RANLIB)"$(LIBDIR)/$(LIBNAME)"'
SLIB_INSTALL_NAME='$(SLIBNAME_WITH_MAJOR)'
SLIB_INSTALL_LINKS='$(SLIBNAME)'
```

But in the new version of FFmpeg, this problem is finally solved, FFmpeg added `android` this `target` . `所以我们再也不需要手动去修改了` .

sysroot

`--sysroot=$SYSROOT` : It is used to configure the cross-compilation environment `根路径` . When compiling, it will search for `usr/include` `usr/lib` these , and then find the relevant header files and library files.

`r20b` Versions of the `NDK` system 's header and library files are in and `$SYSYROOT/usr/include` . `$SYSYROOT/usr/lib`

> Basically, many novices will not find various header files when compiling, resulting in compilation failure. So when the compilation fails to find the header file, the first thing to check is this path.

little doubt

> When compiling with the latest `ndk r20b` version , it is found that even if it is not configured `sysroot` , it can be compiled normally. I doubt whether the Android `clang` tool has been processed, and will automatically find the corresponding path. No reason has been found from the `configure` documentation so far.
> If anyone knows, please let me know.

When it comes `sysroot` to , I have to mention another parameter `-isysyroot` . This parameter has also puzzled me for a long time, because few articles will mention the connection and difference between these two parameters. However, this parameter also leads to a very inexplicable compilation failure. .

extra-cflags

`-isysroot` Before introducing , let's take a look at this `extra-cflags` option.

The effect of this option is to specify a search path for header files `sysroot` other than . for example:

```
--extra-cflags="-I$SYSROOT/usr/include"

# 其中 -I 用于区分不同的路径
```

Rather, it `-isysroot` is a configuration for this option. for example

```
--extra-cflags="-isysroot $SYSROOT"
```

`-isysroot` The function is to set the following path as the default header file search path. At this time, the previous `sysroot` configuration path will no longer be used as `头文件` the default search path, but it is still the `库文件` default search path.

As you can see, these two configurations are the same to some extent:

```
--extra-cflags="-I$SYSROOT/usr/include"

约等于

--extra-cflags="-isysroot $SYSROOT"
```

extra-ldflags

This `extra-cflags` is , but is used to configure additional `库文件` search paths, such as

```
--extra-ldflags="-L$SYSROOT/usr/lib"
# 其中 -L 用于区分不同的路径
```

It can be seen that the `extra-cflags` `extra-ldflags` combination can be replaced `sysroot` .

cross-prefix

This option is literally translated `交叉编译前缀`, referring to the prefix of the cross-compiler tool.

This option is often used in `cc` conjunction .

What does it mean? Some articles on the Internet often show two configuration methods for `cc` this option:

One is only configuration `cross-prefix`, no configuration `cc`, such as this article.

The other is to configure `cross-prefix` and configure `cc`.

for example:

```
--cc=$TOOLCHAIN/bin/arm-linux-androideabi-gcc \
--cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
```

These are two completely different configuration methods, but the amazing thing is that sometimes they can compile successfully, and sometimes there will be an error that the compile chain tool cannot be found.

In order to understand what impact the configuration of `cross-prefix` `cc` these two options has, and how to use these two configurations, I carefully looked at `FFmpeg` the `configure` configuration and found some clues.

Analyze the configure configure script

> Note: The following analysis is based on ffmpeg-4.2.2 version, other versions may be different, just master the basic principles.

Get user configuration options

Open (note: not double-click to run) `configure` the shell script, first let's see how configure obtains the compilation options configured by the user.

Search `for opt do`, you can find the following code

```
for opt do
    optval="${opt#*=}"
    case "$opt" in
        --extra-ldflags=*)
            add_ldflags $optval
        ;;
        --extra-ldexeflags=*)
            add_ldexeflags $optval
        ;;
        --extra-ldsoflags=*)
            add_ldsoflags $optval
        ;;
        --extra-ldlibflags=*)
            warn "The --extra-ldlibflags option is only provided for compatibility
and will be\n"\
                 "removed in the future. Use --extra-ldsoflags instead."
            add_ldsoflags $optval
        ;;
        --extra-libs=*)
            add_extralibs $optval
        ;;
        --disable-devices)
            disable $INDEV_LIST $OUTDEV_LIST
        ;;
        --enable-debug=*)
            debuglevel="$optval"
        ;;

        # 省略中间一些代码...

        *)
            optname="${opt%%=*}"
            optname="${optname#--}"
            optname=$(echo "$optname" | sed 's/-/_/g')
            if is_in $optname $CMDLINE_SET; then
                eval $optname='$optval'
            elif is_in $optname $CMDLINE_APPEND; then
                append $optname "$optval"
            else
                die_unknown $opt
            fi
        ;;
    esac
done
```

The code of this shell script has a lot of unique syntax, and you don't need to go to the horns, you can roughly understand it.

The first line of the for loop `=` obtains `optval`.

In addition to some special options below, let's look at the last wildcard `*)`. The purpose of this code is to associate user-configured options and values.

For example `--cpu=armv7-a`, the first three lines are to `cpu` split out, assign to `optname`, and then `optval` assign to `cpu`, to put it bluntly, `cpu` this as `armv7-a`.

Android related configuration

Search `android` keywords , you can find the following code

```
# ffmpeg-4.2.2/configure

if test "$target_os" = android; then
    cc_default="clang"
fi

ar_default="${cross_prefix}${ar_default}"
cc_default="${cross_prefix}${cc_default}"
cxx_default="${cross_prefix}${cxx_default}"
nm_default="${cross_prefix}${nm_default}"
pkg_config_default="${cross_prefix}${pkg_config_default}"
```

When you configure , `--target-os=android` the default build tool for FFmpeg is `clang`.

`cc_default` In fact, it is `cc` the . You can see that it is spliced with `cc_default` here . `cross_prefix` Here is why it is said `cross_prefix` to be the cross-compiler tool prefix.

The stitching is like this:

```
cc_defalut=$TOOLCHAIN/bin/arm-linux-androideabi-$cc
```

See what `ar_default` `cc_default` `cxx_default` these default values are.

A search `cc_default` can find the following code

```
# ffmpeg-4.2.2/configure

ar_default="ar"
cc_default="gcc"
cxx_default="g++"
host_cc_default="gcc"
```

As you can see, the default compilation tool of FFmpeg is `GCC`.

When you compile the library for the Android platform, due to the `configure` mandatory settings `cc_default="clang"`:

1. When you use `GCC` as a build tool, you must configure the `cc` options, or modify `configure` the in `cc_default="clang"` to `cc_default="gcc"`;

2. When you use `CLANG` as a build tool, you can leave the `cc` option unconfigured.

If you think about it carefully, you will find that when the `cc` configuration is the following value, it can be compiled normally?

```
--cc=$TOOLCHAIN/bin/arm-linux-androideabi-gcc
```

At this time it is `cc_defalut` not equal to

```
cc_defalut=$TOOLCHAIN/bin/arm-linux-androideabi-$TOOLCHAIN/bin/arm-linux-androideabi-gcc
```

This path must be wrong!

It depends on how it `cc_default` is used.

Initialize variables

Search `set_default arch` , you can see the following code, `configure` reset `cc` the default value here.

```
set_default arch cc cxx doxygen pkg_config ranlib strip sysinclude \
    target_exec x86asmexe nvcc
```

Here `set_default` a to see the implementation of this function

```
set_default(){
    for opt; do
        eval : \${$opt:=\$${opt}_default}
    done
}
```

This is also an incomprehensible shell syntax, which roughly means: the for loop gets all the input parameter variables, and then assigns a value to this variable.

For example `set_default cc` , the meaning is `cc=cc_default` , but one thing to pay attention to is the symbol in the middle `:=` .

This notation is similar to the ternary operator in Java:

```
opt != null? opt:opt_defalut
```

That is, if the parameter is empty, `xx_default` assign to `xx` .

This can explain the above question.

1. When configuring

```
--cc=$TOOLCHAIN/bin/arm-linux-androideabi-gcc
```

`set_default cc` It's like useless. Because after the user's configuration is obtained through the `for` loop , `cc` it is not empty. `set_default` After that, `cc` the value will not change.

2. When `cc` not configured, FFmpeg sets the spliced path to according to the default splicing method `cc` .

3. However, `cc=gcc` this , so that the final `cc` value is only `gcc` , and the compilation tool cannot be found correctly.

   Why include `corss-prefix-clang` this option

Now it's time to explain why the `configure` configuration .

The original configuration is like this

```
ar_default="${cross_prefix}${ar_default}"
cc_default="${cross_prefix}${cc_default}"
cxx_default="${cross_prefix}${cxx_default}"
nm_default="${cross_prefix}${nm_default}"
pkg_config_default="${cross_prefix}${pkg_config_default}"
```

That is, the default `cc` `ar` `nm` path prefix is the same, but `Android NDK` the path of

NDK clang path

Did you see it? `ar` / and `nm` prefixes are not the same, the former is , the latter is . `cc` `arm-linux-androideabi-` `armv7a-linux-androideabi16-`

Therefore, the `cc` and `cxx` two prefixes need to be modified, and a new is `cross_prefix_clange` added for separate configuration.

> This is only for the case of NDK r20b, different NDK versions may be different, you can set it according to this principle.

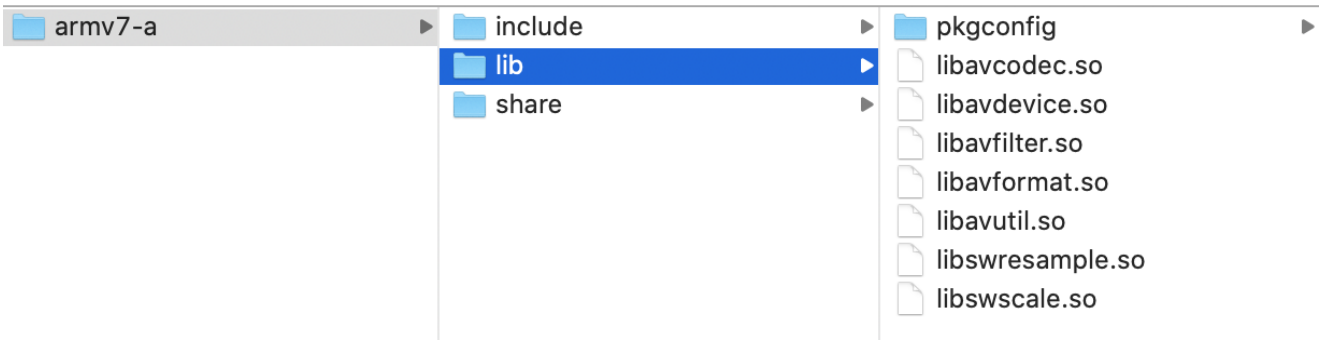
In summary, I explained some common configuration options for compiling FFmpeg, and in principle, I figured out why it should be configured in this way.

start compilation

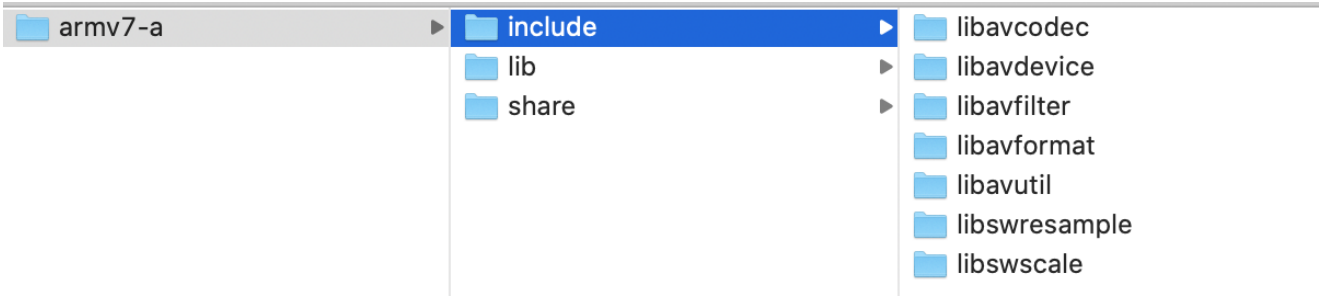
Open the cmd terminal and cd to the directory where FFmpeg is located

Enter ./build_android_clang.sh

Waiting for the compilation to complete, you will get and two directories `include` in the ffmpeg/android/armv7-a directory , respectively and `lib` `头文件` `so库文件`



generated so



generated header file

Five, use GCC to compile FFmpeg

At present, most of the articles on the Internet `GCC` are compiled `FFmpeg` using . Let's take a look at how to configure `GCC` the compilation parameters of .

Download Android NDK r17b

As mentioned earlier, after NDK r17c, Googole removed GCC, so to use GCC, only r17c and previous versions can be downloaded. This article uses r17c to compile.

Select the corresponding version according to your own compilation platform: NDK r17c

This article selects the Mac version: Mac OS X.

NDK-related environment paths

NDK r17c directory

`NDK r20b` Compared with , `NDK r17c` the directory of . is slightly changed.

cross compilation environment path

```
# 库文件路径
android-ndk-r17c/platforms/android-21/arch-arm/usr/lib

# 头文件路径
android-ndk-r17c/sysroot/usr/include
```

GCC toolchain path

```
android-ndk-r17c/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin
```

It can be seen that Google separated `头文件` and `库文件` , which is also the reason why many novices have not paired paths when compiling, resulting in compilation failures.

Create a new build configuration script

The version of FFmpeg still uses the above `ffmpeg-4.2.2` , of course, this time there is no need to modify `configure` it .

According to the knowledge introduced above, it is easy to write the compilation configuration

Create a new script in the `ffmpeg-4.2.2` root directory: build_android_gcc.sh

```
#!/bin/bash
set -x
API=21
CPU=armv7-a
#so库输出目录
OUTPUT=/Users/cxp/Desktop/FFmpeg/ffmpeg-4.2.2/android/$CPU
# NDK的路径，根据自己的安装位置进行设置
NDK=/Users/cxp/Desktop/FFmpeg/android-ndk-r17c
# 库文件
SYSROOT=$NDK/platforms/android-$API/arch-arm
# 头文件
ISYSROOT=$NDK/sysroot/usr/include
# 汇编头文件
ASM=$ISYSROOT/arm-linux-androideabi
TOOLCHAIN=$NDK/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64

function build
{
./configure \
  --prefix=$OUTPUT \
  --target-os=android \
  --arch=arm \
  --cpu=armv7-a \
  --enable-asm \
  --enable-cross-compile \
  --enable-shared \
  --disable-static \
  --disable-doc \
  --disable-ffplay \
  --disable-ffprobe \
  --disable-symver \
  --disable-ffmpeg \
  --sysroot=$SYSROOT \
  --cc=$TOOLCHAIN/bin/arm-linux-androideabi-gcc \
  --cross-prefix=$TOOLCHAIN/bin/arm-linux-androideabi- \
  --extra-cflags="-I$ISYSROOT -I$ASM -fPIC"

make clean all
# 这里是定义用几个CPU编译
make -j12
make
make install
}
build
```

As you can see, configuration is basically the same `CLANG` as .

There are the following differences:

1. more `cc` configuration . Because if not configured, the `cc` default is `clang` (refer to the previous analysis);

2. `extra-cflags` There is more configuration, because `SYSROOT` only contains the search path of , which needs `库文件` additional configuration ; the path of is not in , and additional configuration is also required . `头文件` `汇编头文件` `SYSROOT` `ASM`

start compilation

Open cmd terminal, cd to the ffmpeg-4.2.2 directory

Execute ./build_android_gcc.sh

Dang, this article only introduces the most basic configuration scheme, you can `--disable-xxx` also achieve the right `FFmpeg` crop through more `--enable-xxx` options enable some advanced functions through options.