Android Studio NDK开发

概述

- 在Eclipse的时代，我们进行NDK的开发一般需要通过手动执行NDK脚本生成*.so文件，再将.so文件放到对应的目录之后，之后再进行打包。
- AS + Gradle的NDK开发
    - 不需要再去通过javah根据java文件生成头文件，并根据头文件生成的函数声明编写cpp文件
    - 当在Java文件中定义完native接口，可以在cpp文件中自动生成对应的native函数，所需要做的只是补全函数体中的内容
    - 不需要手动执行ndk-build命令得到so，再将so拷贝到对应的目录
    - 在编写cpp文件的过程中，可以有提示了

创建支持C/C++的项目

安装组件

- NDK
- CMake
- LLDB

创建工程

- C++ Standard：选择C++的标准，Toolchain Default表示使用默认的CMake配置，这里我们选择默认。
- Excptions Support：如果您希望启用对C++异常处理的支持，请选中此复选框。如果启用此复选框，Android Studio会将-fexceptions标志添加到模块级 build.gradle文件的cppFlags中，Gradle会将其传递到CMake。
- Runtime Type information Support：如果您希望支持RTTI，请选中此复选框。如果启用此复选框，Android Studio会将-frtti标志添加到模块级 build.gradle文件的cppFlags中，Gradle会将其传递到CMake

cpp 文件夹

用于存放C/C++的源文件，在磁盘上对应于app/src/main/cpp文件夹，当新建工程时，它会生成一个native-lib.cpp的事例文件

增加 CMakeList.txt 脚本

构建脚本，在磁盘上对应于app/目录下的txt文件，其内容为如下图所示，这里面涉及到的CMake语法包括下面四种，关于CMake的语法，可以查看 官方的API 说明

cmake_minimum_required add_library find_library target_link_libraries

build.gradle 脚本

```
android {
    ...
        externalNativeBuild {
            cmake {
                cppFlags ""
            }
        }
    }
    buildTypes {
       ...
    }
    externalNativeBuild {
        cmake {
            path "CMakeLists.txt"
        }
    }
}
```

加载so库

```
 static {
        System.loadLibrary("native-lib");
    }
```

步骤原理

1. 首先，在构建时，通过build.gradle中path所指定的路径，找到CMakeList.txt，解析其中的内容。
2. 按照脚本中的命令，将src/main/cpp/native-lib.cpp编译到共享的对象库中，并将其命名为libnative-lib.so，随后打包到APK中。
3. 当应用运行时，首先会执行MainActivity的static代码块的内容，使用System.loadLibrary()加载原生库。
4. 在onCreate()函数中，调用原生库的函数得到字符串并展示

在现有的项目中添加C/C++代码

创建一个Android工程

定义native接口

定义 cpp 文件

在模块根目录下的src/main/新建一个文件夹cpp，在其中新增一个cpp文件

定义 CMakeLists.txt

在模块根目录下新建一个CMakeLists.txt文件

在 build.gradle 中进行配置

- 我们需要让Gradle脚本确定CMakeLists.txt所在的位置，我们可以在CMakeLists.txt上点击右键，之后选择Link C++ Project with Gradle
- 要手动配置 Gradle 以关联到我们的原生库，我们需要将 externalNativeBuild {} 块添加到 模块（组件）级 build.gradle 文件中，并使用 cmake {} 或 ndkBuild {} 对其进行配置：

```
android {
  ...
  defaultConfig {...}
  buildTypes {...}

  // 封装您的外部本地构建配置.
  externalNativeBuild {

    // 封装您的 CMake 构建配置.
    cmake {

      // 为CMake 构建脚本提供一个相对路径（这个相对路径是相对于当前
build.gradle的路径）.
      path "CMakeLists.txt"
    }
  }
}
```

> *注：如果你想要将 Gradle 关联到现有 ndk-build 项目，请使用 ndkBuild {} 块而不是 cmake {}，并提供 Android.mk 文件的相对路径。如果 Application.mk 文件与您的 Android.mk 文件位于相同目录下，Gradle 也要包含此文件*

- 指定可选配置

我们可以在 模块(组件)级 build.gradle 文件的 defaultConfig {} 块中配置另一个 externalNativeBuild {} 块，为 CMake 或 ndk-build 指定可选参数和标志。与 defaultConfig {} 块中的其他属性类似，我们也可以在构建配置中为每个产品风味（productFlavors ）重写这些属性。

例如，如果我们的 CMake 或 ndk-build 项目定义多个原生库，我们可以使用 targets 属性仅为给定产品风味（productFlavors ）构建和打包这些库中的一部分。以下代码示例说明了我们可以配置的部分属性：

```
android {
  ...
  defaultConfig {
    ...
    // 这个代码块不同于我们关联到Gradle的CMake或ndk构建脚本的那个
块.
    externalNativeBuild {

      // For ndk-build, instead use ndkBuild {}
      // 用于配置Cmake构建参数
      cmake {

        // 将参数传递给变量时，请使用以下语法：
        // arguments "-DVAR_NAME=ARGUMENT".
        arguments "-DANDROID_ARM_NEON=TRUE",
        // 如果要将多个参数传递给变量，使用以下语法一起传递:
        // arguments "-DVAR_NAME=ARG_1 ARG_2"
        // 下面一行将 'rtti' 和 'exceptions' 传递给
'ANDROID_CPP_FEATURES'.
                  "-DANDROID_CPP_FEATURES=rtti exceptions"

        arguments "-DANDROID_ARM_NEON=TRUE", "-
DANDROID_TOOLCHAIN=clang"

        // 为C编译器设置可选标志.
        cFlags "-D_EXAMPLE_C_FLAG1", "-D_EXAMPLE_C_FLAG2"

        // 设置一个标志使C++编译器的format宏常量 生效.
        cppFlags "-D__STDC_FORMAT_MACROS"
      }
    }
  }

  buildTypes {...}

  productFlavors {
    ...
    demo {
      ...
      externalNativeBuild {
        cmake {
          ...
          // 为这个product flavor 指定要构建和打包的本地库，如果
您不配置这个属性，
          // Gradle 将构建和打包所有您在CMake 或 ndk-build项目
中定义的共享对象库
          targets "native-lib-demo"
        }
      }
    }

    pad {
```

```
        ...
        externalNativeBuild {
          cmake {
            ...
            targets "native-lib-pad"
          }
        }
      }
    }

    // 使用下面的代码块链接Gradle到我们的CMake或ndk-build脚本
    externalNativeBuild {
      cmake {...}
      // or ndkBuild {...}
    }
}
```

CMake部分构建变量列表：

变量名	参数	描述
ANDROID_TOOLCHAIN	clang(默认)	指定CMake应该使用的编译器工具链
ANDROID_PLATFORM	android-19	指定Android的目标平台
ANDROID_CPP_FEATURES	默认为空，可配置： rtti（RunTime Type Information）：运行时类型信息 exceptions: 指示代码使用C++异常	指定CMake编译时需要使用某些C++特性
ANDROID_ARM_MODE	thumb（默认） arm	指定是arm还是以thumb模式生成ARM目标二进制库

CMake构建命令

Android Studio在`cmake_build_command.txt`文件中保存用于执行CMake构建的构建参数。

Android Studio会为每个ABI和每个构建类型创建`cmake_build_command.txt`，放置在如下目录：

> *//.externalNativeBuild/cmake///*

CMake构建参数列表：

构建参数	描述
-G	Android Gradle - Ninja是Android Studio唯一支持的C/C++构建系统.CMake会生成`android_gradle_build.json`文件。 其中包含有关CMake构建的Gradle插件的元数据，例如编译器标志和目标名称。
-DANDROID_ABI	目标ABI
-DCMAKE_LIBRARY_OUTPUT_DIRECTORY	CMake生成的库的位置
-DCMAKE_TOOLCHAIN_FILE	CMake用于交叉编译的`android.toolchain.cmake`文件的路径

如果要显示执行构建过程中的详细信息，比如为了得到更详细的出错信息。

运行后
在`.externalNativeBuild/cmake/debug/{abi}/cmake_build_output.txt`查看log

```
# 开启输出详细的编译和链接信息
set(CMAKE_VERBOSE_MAKEFILE on)
message(STATUS "要打印的信息")
```

自定义变量

```
set(变量名 变量值)
```

常用变量

```
# 引用变量格式：${变量名}
# 工程的源文件目录PROJECT_SOURCE_DIR
# CMakeList.txt文件所在的目录CMAKE_SOURCE_DIR
```

- 指定 ABI 默认情况下，Gradle 会针对 NDK 支持的 ABI 将我们的原生库构建到单独的 .so 文件中，并将其全部打包到我们的 APK 中。如果我们希望 Gradle 仅构建和打包原生库的特定 ABI 配置，我们可以在模块级 build.gradle 文件中使用 ndk.abiFilters 标志指定这些配置，如下所示：
- ABI 是什么

  `ABI`（Application binary interface）应用程序二进制接口。不同的 CPU 与指令集的每种组合都有定义的 `ABI` (应用程序二进制接口)，一段程序只有遵循这个接口规范才能在该 CPU 上运行，所以同样的程序代码为了兼容多个不同的CPU，需要为不同的 `ABI` 构建不同的库文

件。当然对于CPU来说，不同的架构并不意味着一定互不兼容。

- armeabi设备只兼容armeabi；
- armeabi-v7a设备兼容armeabi-v7a、armeabi；
- arm64-v8a设备兼容arm64-v8a、armeabi-v7a、armeabi；
- X86设备兼容X86、armeabi；
- X86_64设备兼容X86_64、X86、armeabi；
- mips64设备兼容mips64、mips；
- mips只兼容mips；

```
android {
  ...
  defaultConfig {
    ...
    externalNativeBuild {
      cmake {...}
      // or ndkBuild {...}
    }

    ndk {
      // Specifies the ABI configurations of your native
      // libraries Gradle should build and package with
your APK.
      abiFilters 'x86', 'x86_64', 'armeabi', 'armeabi-
v7a',
                 'arm64-v8a'
    }
  }
  buildTypes {...}
  externalNativeBuild {...}
}
```

在大多数情况下，我们只需要在 ndk {} 块中指定 abiFilters（如上所示），因为它会指示 Gradle 构建和打包原生库的这些版本。不过，如果我们希望控制 Gradle 应当构建的配置，并独立于我们希望其打包到 APK 中的配置，请在 defaultConfig.externalNativeBuild.cmake{} 块（或 defaultConfig.externalNativeBuild.ndkBuild{} 块中）配置另一个 abiFilters 标志。Gradle 会构建这些 ABI 配置，不过仅会打包我们在 defaultConfig.ndk{} 块中指定的配置。

为了进一步降低 APK 的大小，请考虑 配置 ABI APK 拆分，而不是创建一个包含原生库所有版本的大型 APK，Gradle 会为我们想要支持的每个 ABI 创建单独的 APK，并且仅打包每个 ABI 需要的文件。如果我们配置 ABI 拆分，但没有像上面的代码示例一样指定 abiFilters 标志，Gradle 会构建原生库的所有受支持 ABI 版本，不过仅会打包我们在 ABI 拆分配置中指定的版本。为了避免构建我们不想要的原生库版本，请为 abiFilters 标志和 ABI 拆分配置提供相同的 ABI 列表。

实现 C++

> *JNIEXPORT和JNICALL： 它们是JNI中所定义的宏， 可以在jni.h这个头文件中查找到；*

> *JNIEnv*：表示一个指向JNI环境的指针，可以通过它来访问JNI提供的接口方法；*

> *jobject：表示Java对象中的this；*

编写CMakeLists.txt脚本

- Cmake常用命令

命令	含义
cmake_minimum_required	指定需要CMAKE的最小版本
include_directories	指定 原生代码 或 so库 的头文件路径
add_library	添加 源文件或库
set_target_properties(<> PROPERTIES IMPORTED_LOCATION)	指定 导入库的路径
set_target_properties(<> PROPERTIES LIBRARY_OUTPUT_DIRECTORY)	指定生成的目标库的导出路径
find_library	添加NDK API
target_link_libraries	将预构建库关联到原生库
aux_source_directory	查找在某个路径下的所有源文件

cmake_minimum_required用于指定CMake的最低版本信息，不加入会收到警告。

```
cmake_minimum_required(VERSION 3.4.1)
```

从原生代码构建一个原生库

我们通过add_library让CMake根据native-lib.cpp源文件构建一个名为native-lib的共享库

- 静态库：以.a结尾。静态库在程序链接的时候使用，链接器会将程序中使用到函数的代码从库文件中拷贝到应用程序中。一旦链接完成，在执行程序的时候就不需要静态库了。
- 共享库：以.so结尾。在程序的链接时候并不像静态库那样在拷贝使用函数的代码，而只是作些标记。然后在程序开始启动运行的时候，动态地加载所需模块

```
add_library(
        #第一个参数，决定了最终生成的共享库的名字
        native-lib #最终生成的so文件libnative-lib.so
        #第二个参数，我们可以指定根据源文件编译出来的是静态库还是共享库，分别对应STATIC/SHARED关键字
        SHARED
        #指定源文件
        native-lib.cpp)
```

添加 NDK API

在Android系统当中，预制了一些标准的NDK库，这些库函数的目的就是让开发者能够在原生方法中实现之前在Java层开发的一些功能

> *android-ndk-r20\build\cmake\system_libs.cmake*

在 CMakeLists.txt 引入 Android NDK 的 log 库

- find_library：将一个变量和Android NDK的某个库建立关联关系。该函数的第二个参数为Android NDK中对应的库名称，而调用该方法之后，它就被和第一个参数所指定的变量关联在一起。 在这种关联建立以后，我们就可以使用这个变量在构建脚本的其它部分引用该变量所关联的NDK库。
- target_link_libraries：把NDK库和我们自己的原生库native-lib进行关联，这样，我们就可以调用该NDK库中的函数了

```
find_library(
 # 定义路径变量的名称 并用这个变量存储 NDK库的位置。
        log-lib
         # 指定 CMake 需要定位的NDK库的名称
        log)
# 将一个或多个 其他本地库 链接到我们的本地库上。
target_link_libraries(# 指定目标库（native-lib是我们自己创建的
原生库）.
        native-lib
          # 将日志库链接到目标库.
        ${log-lib})
```

- NDK 还以源代码的形式包含一些库，我们在构建和关联到我们的原生库时需要使用这些代码。我们可以使用 CMake 构建脚本中的 add_library() 命令，将源代码编译到原生库中。要提供本地 NDK 库的路径，我们可以使用 ANDROID_NDK 路径变量，Android Studio 会自动为您定义此变量。以下命令可以指示 CMake 构建 android_native_app_glue.c，后者会将 NativeActivity 生命周期事件和触摸输入置于静态库中并将静态库关联到 native-lib：

```
add_library( app-glue
             STATIC

${ANDROID_NDK}/sources/android/native_app_glue/android_nat
ive_app_glue.c )

# 您需要将 静态（STATIC）库 与 共享(SHARED )的本地库链接起来.
target_link_libraries( native-lib
                       app-glue
                       ${log-lib} )
```

- 在代码中引入头文件，并调用Log函数

```
#include <android/log.h>
...
 __android_log_write(ANDROID_LOG_DEBUG, tag, log);
...
```

引入第三方so库

将 so 库和头文件拷贝到对应的目录

> */app/src/main/jniLibs/arm/libxxx.so*

修改 CMakeLists.txt 文件

- 第三方so库 这里和之前在第二步中介绍的创建一个新的原生库类似，区别在于最后一个参数，我们通过IMPORTANT标志告知CMake只希望将库导入到项目中。
- 目标库的路径 这里有几点需要说明：
- CMAKE_SOURCE_DIR}表示的是CMakeLists.txt所在的路径，我们指定第三方so所在路径时，应当以这个常量为起点。
- 来说，我们应当为每种ABI接口提供单独的软件包，那么，我们就可以在jinLibs下建立多个文件夹，每个文件夹对应一种ABI接口类型，之后再通过${ANDROID_ABI}来泛化这一层目录的结构，这样将有助于充分利用特定的CPU架构。
- 三方的库关联到原生库 这里和将NDK库关联到原生库的原理是一样的。
- 过，为了确保 CMake 可以在编译时定位 我们的 头文件，我们需要将 **include_directories()** 命令添加到 CMake 构建脚本中并指定 头文件的路径

```
add_library( # 指定目标导入库.
             imported-lib

             # 设置导入库的类型（静态或动态） 为 shared
library.
             SHARED

             # 告知 CMake imported-lib 是导入的库
             IMPORTED )


set_target_properties( # 指定目标导入库
                       imported-lib

                       # 指定属性（本地导入的已有库）
                       PROPERTIES IMPORTED_LOCATION

                       # 指定你要导入库的路径.
                       # ${CMAKE_SOURCE_DIR}
                       imported-
lib/src/${ANDROID_ABI}/libimported-lib.so )
```

```
#为了确保 CMake 可以在编译时定位到我们的 头文件，我们需要使用
include_directories() 命令，并包含 头文件的路

include_directories( imported-lib/include/ )

#要将预构建库关联到我们的原生库，请将其添加到 CMake 构建脚本的
target_link_libraries() 命令中
target_link_libraries(  # 指定了三个库，分别是native-lib、
imported-lib和log-lib.
                        native-lib
                        imported-lib
                        # log-lib是包含在 NDK 中的一个日志库
                        ${log-lib} )
```

在代码中引入第三方库的头文件，调用函数

3 资料文献

首推 Android NDK 官方文档，虽然很多都不完整，但是绝对是必须看一遍的东西。

当初次接触 `NDK` 开发又觉得新建的 Hello World 项目过于简单时。建议把 googlesamples - android-ndk 项目拉下来。里面有多个实例参考，比官方文档完整很多。

Q1：怎么指定 C++标准？

A：在 `build_gradle` 中，配置 `cppFlags -std`

```
externalNativeBuild {
  cmake {
    cppFlags "-frtti -fexceptions -std=c++14"
    arguments '-DANDROID_STL=c++_shared'
  }
}复制代码
```

Q2：add_library 如何编译一个目录中所有源文件？

A： 使用 `aux_source_directory` 方法将路径列表全部放到一个变量中。

```
# 查找所有源码 并拼接到路径列表
aux_source_directory(${CMAKE_HOME_DIRECTORY}/src/api
SRC_LIST)
aux_source_directory(${CMAKE_HOME_DIRECTORY}/src/core
CORE_SRC_LIST)
list(APPEND SRC_LIST ${CORE_SRC_LIST})
add_library(native-lib SHARED ${SRC_LIST})复制代码
```

Q3：怎么调试 CMakeLists.txt 中的代码？

A：使用 `message` 方法

```
cmake_minimum_required(VERSION 3.4.1)
message(STATUS "execute CMakeLists")
...复制代码
```

然后运行后在 `.externalNativeBuild/cmake/debug/{abi}/cmake_build_output.txt` 中查看 log。

Q4：什么时候 CMakeLists.txt 里面会执行？

A：测试了下，好像在 sync 的时候会执行。执行一次后会生成 `makefile` 的文件缓存之类的东西放在 `externalNativeBuild` 中。所以如果 `CMakeLists.txt` 中没有修改的话再次同步好像是不会重新执行的。（或者删除 `.externalNativeBuild` 目录）

真正编译的时候好像只是读取`.externalNativeBuild` 目录中已经解析好的 `makefile` 去编译。不会再去执行 `CMakeLists.txt`

gcc/clang编译器的编译命令

编译命令： gcc/clang -g -O2 -o log ffmpeg_log.c -I -L -l(第一竖线是大写的i，第三个竖线是小写的L) 示例clang -g -O2 -o log ffmpeg_log.c -I .../ffmpeg -L .../ffmpeg/libavutil -lavutil

解析： -g 输出文件中的调试信息 -O2 对输出文件做指令优化（默认是-O1是不对指令进行优化，-O2编译器会按照自己的理解优化指令，让指令运行的更快） -o 输出文件的名字 -o后面跟的.c文件就是要编译的文件的名字 -I 指定头文件的位置 -L 指定库文件的位置 -l 指定引用的库文件名字 示例命令是使用ffmpeg的日志系统。

```
#include <stdio.h>
#include <libavutil/log.h>
int main(int argc,char* argv[]){
    av_log_set_level(AV_LOG_DEBUG);
    av_log(NULL,AV_LOG_DEBUG,"Hello World\n");
    av_log(NULL,AV_LOG_INFO,"Hello World\n");
    av_log(NULL,AV_LOG_WARNING,"Hello World\n");
    av_log(NULL,AV_LOG_ERROR,"Hello World\n");
    return 0;
}
```

1. 添加头文件目录 include_directories 命令：
   include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...]) 使用：
   include_directories(${CMAKE_CURRENT_LIST_DIR}/calculate) 作用： 把当前目录(CMakeLists.txt所在目录)下的calculate文件夹加入到包含路径
2. 找到我的库文件 link_directories 命令： link_directories(directory1 directory2 ...) 使用：
   link_directories(${CMAKE_CURRENT_LIST_DIR}/calculate) 作用：
   与include_directories()类似，这个命令添加了库包含路径。
3. 添加源文件目录 aux_source_directory 命令： aux_source_directory(

) 使用： aux_source_directory(./calculate calculateLib) 作用：
   发现一个目录下所有的源代码文件并将列表存储在一个变量中，这个指令临时被用来 自动构建源文件列表。因为目前 cmake 还不能自动发现新添加的源文件;此例子，即是 查找 calculate路径下的所有源文件，保存到calculateLib变量中.

注意：不会递归包含子目录，仅包含指定的dir目录

4. 添加源文件目录 add_executable 命令： add_executable([WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] source1 [source2 ...]) 使用： add_executable(calculate $calculateLib$)作用：使用{calculateLib}里面的源文件来生成一个可执行文件，起名叫calculate

注意：使用进行变量的引用。在IF等语句中，是直接使用变量名而不通过{}取值

5. 添加源文件目录 add_library 命令： add_library([STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] [source1] [source2[...]) 使用：
   add_library(jason-lib SHARED src/main/cpp/Jason-lib.cpp) 作用： 该指令的主要作用就是将指定的源文件生成链接文件，然后添加到工程中去