第 3 章

NDK 开发场景

3.1 NDK 开发实际集成源码的场景


57min

在移动应用开发中，将 C/C++实现的功能整合至应用中常常采用两种主要方式。一种方式是通过集成开发环境（如 Android Studio）直接将源代码整合进项目中。另一种方式是利用预先编译好的库，在本章节将详细介绍这两种方式的实际应用，同时会深入探讨 FFmpeg 的编译过程。

3.1.1 使用 Android Studio 源码直接集成

1. 项目创建

首先，打开 Android Studio，选择 File→New→New Project 来建立一个新的工程。与非 NDK 应用不同之处在于要选择 Native C++项目类型。在项目类型选择界面，选择默认 Activity 类型为 Native C++，具体界面如图 3-1 所示。

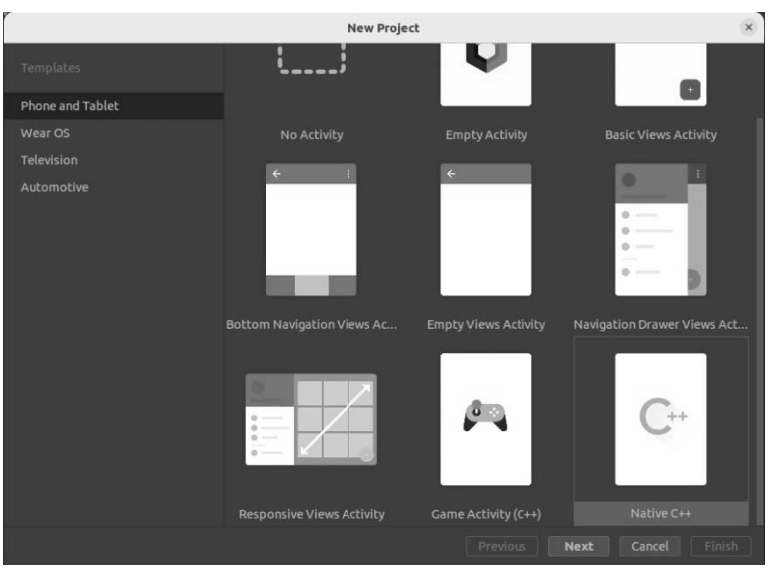

图 3-1　Native C++

单击 Next 按钮，输入项目名称、包名、语言及保存路径等信息。在这个示例中，将工程命名为 Ndk3_1，选择语言为 Java，界面如图 3-2 所示。

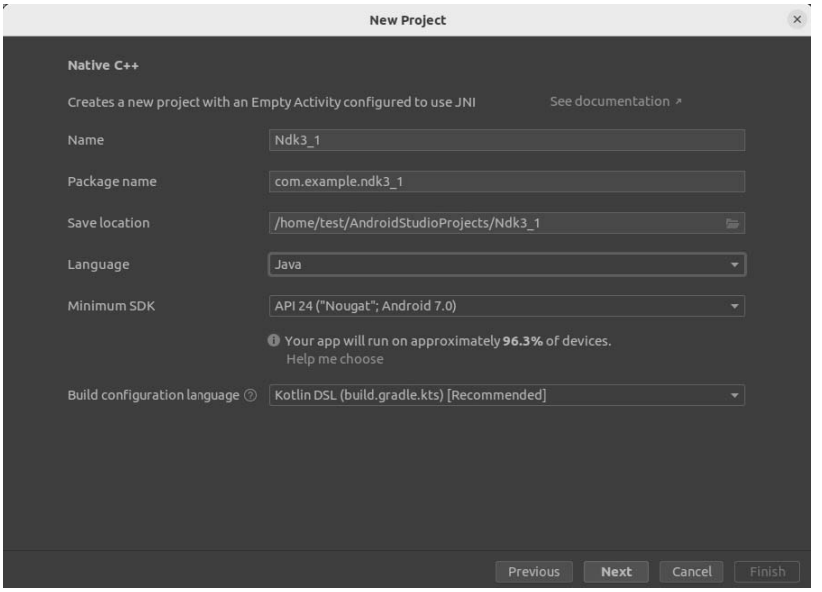

图 3-2　工程配置

单击 Next 按钮进入 C++配置界面，这个界面用于配置 C++的版本信息。通常可选择默认配置，或根据项目的特定要求进行调整，示例界面如图 3-3 所示。

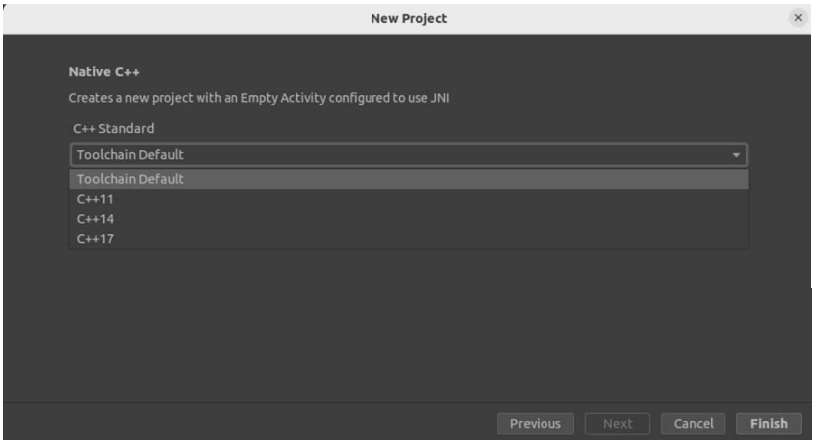

图 3-3　C++版本选择

本节的主要目标是演示如何使用 Android Studio 集成 C/C++源码，并没有对 C++版本有特定的要求，因此，在这个示例中选择默认 C++版本，然后单击 Finish 按钮完成项目的创建。

一旦项目创建完成便会对 gradle 等依赖进行同步。在这个过程中，可能会出现 gradle 下载超时的情况，类似于图 3-4 所展示的情况。

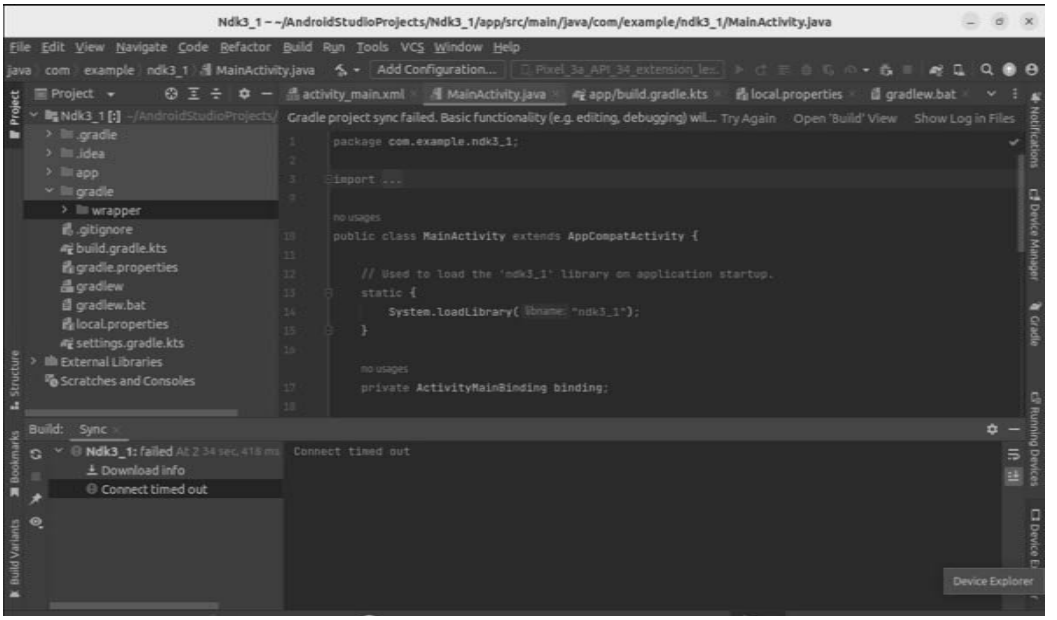

图 3-4　同步超时

这个问题通常与网络有关，可以通过将 gradle 地址更换为国内镜像网址来解决。首先，将项目视图切换为 Project 模式，然后选择 gradle→wrapper→gradle→wrapper.properties。在这个示例中，当前使用的 gradle 版本是 8.0，具体操作如图 3-5 所示。

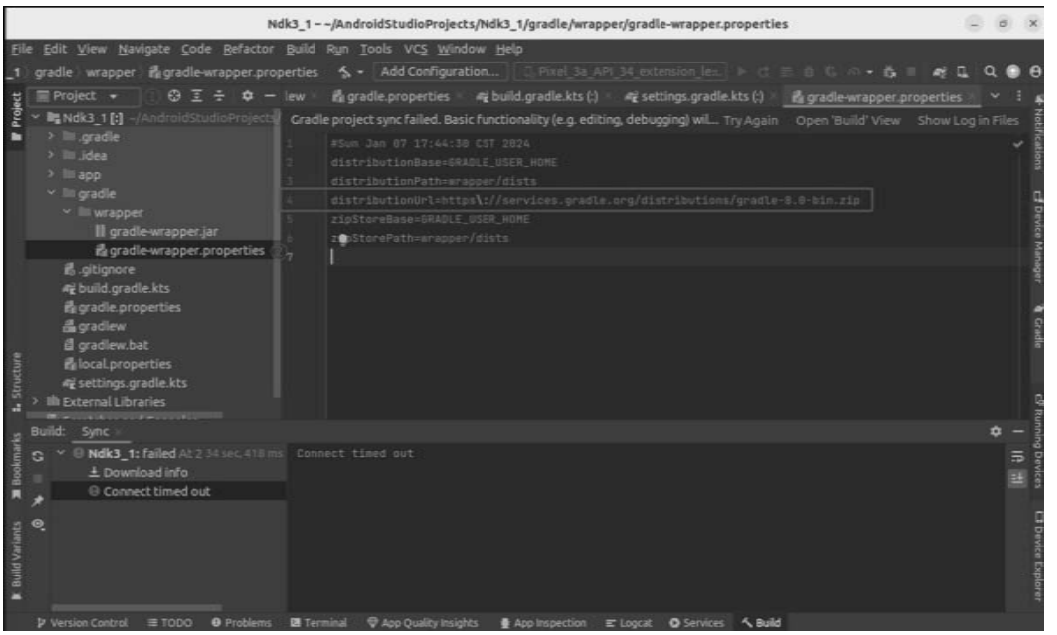

图 3-5　gradle 配置

只需将 gradle 地址更换为以下地址：distributionUrl=https://mirrors.cloud.tencent.com/gradle/gradle-8.0-bin.zip，然后单击 Try Again 按钮便可完成项目的同步。

2. 源码编写

为了深入学习源码集成，本节采用编写简单的 C 源码的方式进行学习。这样做的主要目的是凸显源码集成的核心内容，而非仅限于源代码本身。接下来我们将在 app/src/main/cpp 目录下创建 3 个文件：LogUtils.h、Add.c、Add.h。

LogUtils.h 文件的代码如下：

```
    //第 3 章/LogUtils.h
    #ifndef NDK3_1_LOGUTILS_H
    #define NDK3_1_LOGUTILS_H
    #include<android/log.h>
    #define LOGD(...)  __android_log_print(ANDROID_LOG_DEBUG,TAG,__VA_ARGS__)
//定义 LOGD 类型
    #define LOGI(...)  __android_log_print(ANDROID_LOG_INFO,TAG,__VA_ARGS__)
//定义 LOGI 类型
    #define LOGW(...)  __android_log_print(ANDROID_LOG_WARN,TAG,__VA_ARGS__)
//定义 LOGW 类型
    #define LOGE(...)  __android_log_print(ANDROID_LOG_ERROR,TAG,__VA_ARGS__)
//定义 LOGE 类型
    #endif //NDK3_1_LOGUTILS_H
```

这个头文件定义了常用的日志打印函数，通过宏定义的方式定义了 LOGD、LOGI、LOGW 及 LOGE 共 4 种常用的打印 log 的宏函数。在使用的地方包含 LogUtils.h 及定义 TAG 即可。

Add.c 文件的代码如下：

```
    //第 3 章/Add.c
    #include "Add.h"          //包含 Add.c 头文件
    #define TAG "Add"         //定义 TAG，以便区分不同文件打印
    /**
     * add 函数
     * @param a   第 1 个参数
     * @param b   第 2 个参数
     * @return 返回两数之和
     */
    int add(int a, int b){
       //打印调试日志
       LOGI("a = %d  b = %d\n", a, b);
       //返回两数之和
       return a + b;
    }
```

Add.h 文件的代码如下：

```
    //第 3 章/Add.h
```

```
#ifdef __cplusplus   //注意__cplusplus 编译器的保留宏定义，也就是说编译器认为这个宏
//已经定义了，一定要完全一样，否则会出问题
extern "C"{
#endif
#ifndef NDK3_1_ADD_H
#define NDK3_1_ADD_H
#include "LogUtils.h"
//函数声明
int add(int a, int b);
#endif //NDK3_1_ADD_H

#ifdef __cplusplus
}
#endif
```

注意：当代码包含 extern "C"声明时，它告诉编译器该部分代码应该按照 C 语言的标准来进行编译，而不是像 C++ 那样可能支持函数重载。这意味着编译器不会为函数的参数添加类型信息，因为它不适用在 C 语言中。如果代码可能会在 C++的环境下运行，则需要使用此方式定义头文件。

3. 编译配置

无论是直接集成源码还是使用预编译库文件，当源代码编写或库集成工作完成之后，都需要对项目的编译配置进行调整。这个过程通常会涉及修改项目的构建文件，其中在使用 CMake 作为构建系统时，这个文件通常是 CMakeLists.txt。

具体来讲，修改 CMakeLists.txt 文件是为了让构建系统知道存在新加入的源代码文件或库文件，并正确地将其包含到项目的构建过程中。这包括指定源代码文件的位置、编译选项、链接外部库等。以下是源码集成编译配置，代码如下：

```
#第 3 章/CMakeLists.txt
#声明使用的 CMake 的最低版本
cmake_minimum_required(VERSION 3.22.1)

#项目名称
project("ndk3_1")

#用来添加一个库文件，其中 SHARED 代表动态库
add_library(${CMAKE_PROJECT_NAME} SHARED
        #库生成所需的源文件
        Add.c
        native-lib.cpp)

#链接库，表示上面生成的库需要依赖 android 和 log 库
target_link_libraries(${CMAKE_PROJECT_NAME}
        #List libraries link to the target library
        android
```

```
        log)
```

4. 使用库函数

在集成外部库时，除功能源码外，通常还需一个接口文件，这个文件负责向外界提供接口函数。使其他代码可以调用库中的功能。接口函数的提供方式分为静态注册和动态注册两种。对于规模较小的项目，静态注册方式较为常见。这种方式在集成时，借助集成开发环境的帮助，相对更容易实现。静态注册方式通常在编译时确定接口函数的映射关系，因此其实现过程较为直观和简单。

然而，静态注册方式在移植性方面相较于动态注册会稍逊一筹。这是因为静态注册通常在编译时固定了接口函数的映射关系，在移植到不同的包名环境下可能需要重新修改和编译，而动态注册方式则可以在运行时动态地加载和注册接口函数，因此具有更好的一致性和灵活性。

本节将使用静态注册的方式来对外提供接口。这种方式虽然在一些方面可能不如动态注册方式，但对于小项目来讲，其简单易用的特点通常能够满足需求。

1）动态库加载

在创建原生（Native）工程时，集成开发环境通常会默认使用库的项目名称，在 MainActivity 的静态代码块中会自动加载生成的动态库。这个过程是 IDE 自动化配置的一部分，旨在简化原生库的集成过程。

静态代码块是在类加载时执行的代码段，通常用于执行只需执行一次的初始化操作。在这里，IDE 会在 MainActivity 的静态代码块中插入代码，以便在应用程序启动时加载并初始化动态库。

加载动态库的代码通常类似于 System.loadLibrary 方法，并传入库的名称。这会告诉系统去加载与设备架构匹配的动态库文件，例如，本节中动态库的名称为 ndk3_1，那么 IDE 会自动生成如下代码：

```
//第 3 章/MainActivity.java
public class MainActivity extends AppCompatActivity {

    //静态代码块，类加载时会自动执行
    static {
        System.loadLibrary("ndk3_1");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //以下代码为工程自动生成，读者可不必关心
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
 }
}
```

这段代码确保了MainActivity被加载到JVM中时，ndk3_1动态库也会被加载到进程中，从而可以供应用程序的其余部分使用。这是Android应用程序中加载和使用原生库的标准方式之一。

注意：类加载机制和库加载涉及Java虚拟机（JVM）和Android源码的深层次知识，对于初学者来讲，理解这些机制需要一定的实践和经验积累，因此在学习初期，读者不必过于深究相关细节，而应该从宏观上了解这些机制的基本概念和作用。随着学习的深入，逐渐掌握Java虚拟机和Android源代码的相关知识。

2）创建 Native 方法

细心的读者可能会注意到，在新建的工程中 MainActivity 中有一个名为 stringFromJni 的 native 方法，代码如下：

```
//第3章/MainActivity.java
public class MainActivity extends AppCompatActivity {
    //静态代码块，类加载时会自动执行
    static {
        System.loadLibrary("ndk3_1");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //以下代码为工程自动生成，读者可不必关心
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        //Example of a call to a native method
        TextView tv = binding.sampleText;
        tv.setText(stringFromJNI());
    }
    //native方法的声明
    public native String stringFromJNI();
}
```

此方法使用 native 关键字修饰，代表这是一个 native 方法，这是 Java 层的接口函数。对应地，在 native 层也有一个对应的 C/C++函数。在编译配置中，在 CMakeLists 中除了添加了自己编写的 Add.c 文件之外，还有一个 native-lib.cpp 文件，该文件就是 IDE 默认生成的用于向外界提供原生接口的接口文件，代码如下：

```
//第3章/native-lib.cpp
#include <jni.h>
#include <string>
#include "Add.h"

#define TAG "native-lib"
```

```
extern "C" JNIEXPORT jstring JNICALL
Java_com_example_ndk3_11_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    std::string hello = "Hello from C++";
    //调用源码库的功能函数
    int ret = add(1, 2);
    //打印执行结果
    LOGI("ret = %d", ret);
    return env->NewStringUTF(hello.c_str());
}
```

3）运行

当单击“运行”按钮时，将在虚拟机上观察到以下现象：首先，模拟器的屏幕中央会显示一行文字：“Hello from C++”。这段文字来源于 stringFromJNI()函数的返回值，该函数的返回值最终通过 TextView 组件在模拟器的界面上进行展示。这一功能展示了 C++代码与 Android 界面之间的交互，表明 C++代码能够成功地为 Android 应用提供数据。

其次，在 IDE 的右下角，将看到一条 Log 输出信息。这条信息记录了 add 函数的调用情况。add()函数在后台执行了加法运算，并通过 Log 系统将其执行结果输出到 IDE 的终端上。通过观察这条 Log 输出，开发人员可以了解 add()函数的调用情况，以及它在应用中的实际表现，实际运行情况如图 3-6 所示。

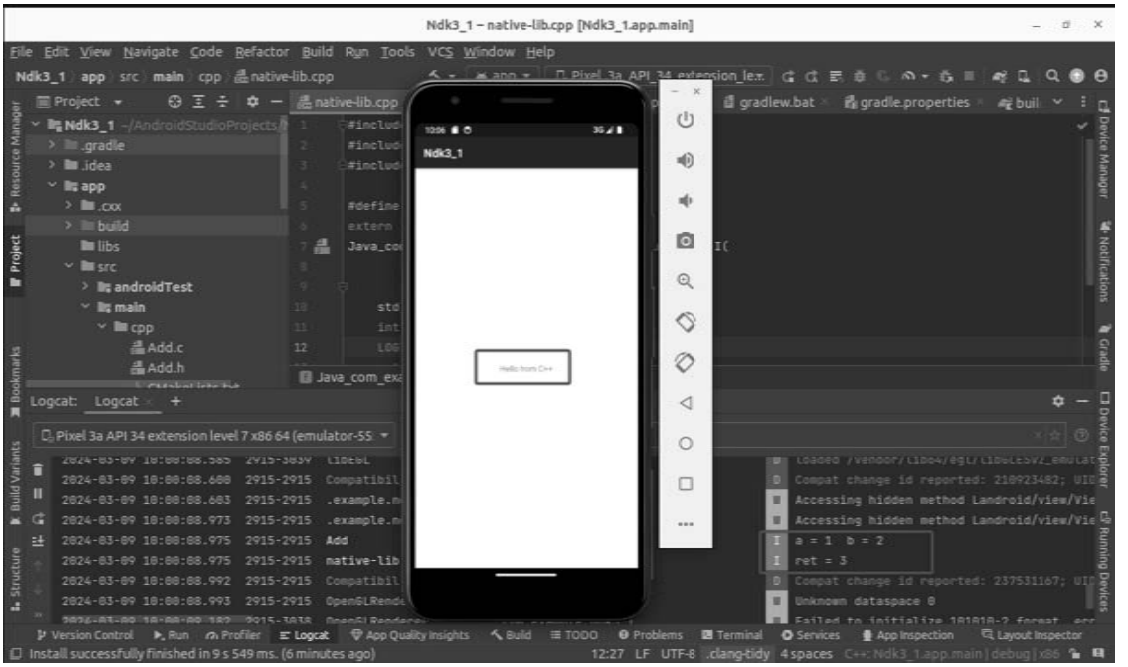

图 3-6　运行结果

3.1.2　使用命令编译出符合平台相关的预编译库

在软件开发的过程中，通常涉及多个小组乃至多个公司的协同合作。在这种合作模式下，往往有专门的小组或公司负责提供通用的功能组件，而非直接开发面向终用户端的软件。他们的主要任务是开发并提供 SDK（软件开发工具包），以供其他合作方使用。这些 SDK 是预先设计好的功能集合，旨在简化其他小组或公司的开发工作，使他们能够基于这些 SDK 进行二次开发，从而完成整个软件的开发流程。

通过正确的编译过程，可以确保这些库与目标平台兼容，并且具备所需的性能和稳定性。在这个过程中，理解并熟练运用工具链是关键。工具链包含了从源代码到最终产品的整个编译过程中所需的各种工具和技术。通过配置和使用这些工具，可以将源代码转换成可在目标平台上运行的预编译库。

因此，本章节将指导读者如何选择合适的工具链，以及如何编写和执行编译脚本，从而生成符合平台要求的预编译库。这对于任何参与软件开发合作的团队或公司来讲都是一项至关重要的技能。

1. CMAKE_TOOLCHAIN_FILE

在 1.2 节中深入地探讨了与编译相关的概念，包括预编译、编译、汇编和链接等重要步骤，其中，特别演示了在 Linux 平台上如何使用 gcc 和 ar 等工具来编译 x86 架构的动态库和静态库。

然而，在实际的 SDK 开发工作中，情况往往更为复杂。开发者通常需要面对多种平台，如 Linux、macOS、Windows、Android 及 iOS 等。每种平台或架构可能有其独特的编译器和工具链，这使配置工具链变得相当烦琐。

为了简化这一流程，CMake 引入了 CMAKE_TOOLCHAIN_FILE 这一变量。它允许开发者为每个目标平台预先定义一个工具链文件。当需要为不同平台编译时，只需指定相应的工具链文件，而无须手动配置每个工具链参数。这一特性的引入极大地提高了跨平台开发的效率和便捷性。通过合理利用 CMAKE_TOOLCHAIN_FILE，开发者能够更关注于代码本身，而无须在复杂的编译环境配置上花费过多的精力。这对于任何需要跨平台工作的 SDK 开发团队来讲都是一项极具价值的改进。

2. android.toolchain.cmake

在 ndk-r19 之前的版本中，为了生成自定义的工具链，开发者通常需要借助 make_standalone_toolchain.py 这一脚本。随着 Android NDK 的版本迭代，截至本书编写时，最新的 NDK 版本已经更新至 r26c。在这一过程中，旧的生成方式由于其使用过于烦琐，在实际开发中已经逐渐被淘汰，因此，这里不再深入介绍。

自 ndk-r19 起，Android 引入了名为 android.toolchain.cmake 的独立工具链文件，旨在进一步简化交叉编译的步骤。该文件的引入极大地提高了开发效率，使开发者能够轻松地处理不同架构的动态库编译任务。该文件可在 NDK 工具链中的 build/cmake 目录中找到。

在掌握了这些背景知识之后，接下来将详细介绍如何利用 android.toolchain.cmake 这一

工具链文件来进行 Android 中各架构的动态库交叉编译。

3. 源码准备

按照以下步骤完成所需任务：

首先，创建一个名为 3.1.2 的目录，这个目录将用于存放所有的源文件和编译脚本。在本示例中，将复用 3.1.1 节中的源文件，即 Add.c、Add.h 及 LogUtils.h。

这些源文件内容保持和 3.1.1 节中的内容一致。为了保持代码的规范性和组织性，接下来在 3.1.2 节目录中创建两个子目录：src 和 include，其中，将 Add.c 文件移动到 src 目录中，将 Add.h 和 LogUtils.h 文件移动到 include 目录中，完成后的目录结构如图 3-7 所示。

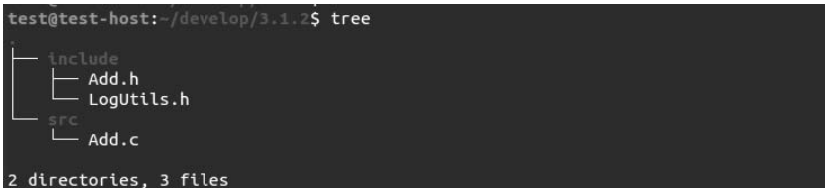

图 3-7　源文件目录结构（1）

4. CMakeLists 文件编写

在源文件就绪后，接下来开始编写 CMakeLists.txt，用来编译生成动态库，内容如下：

```
#第 3 章/CMakeLists.txt
#声明 CMake 的最低版本
cmake_minimum_required(VERSION 3.22.1)
#将项目名称设置为 add
project(add)
#设置 NDK 路径
set(ANDROID_NDK "${NDK_ROOT}" CACHE PATH "Android NDK path")
#设置 Android API 级别
set(ANDROID_API_LEVEL 22)
#设置头文件目录
include_directories(include)
#设置源文件列表
set(SOURCES src/Add.c)

#设置输出目录变量，ANDROID_ABI 为外部传入
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY_${ANDROID_ABI}
${CMAKE_CURRENT_SOURCE_DIR}/libs/${ANDROID_ABI})

#编译某个架构的
function(build_library)
    #使用 SOURCES 中的源文件生成库
    add_library(${CMAKE_PROJECT_NAME} SHARED ${SOURCES})
    #链接库，表示上面生成的库需要依赖 android 和 log 库
    target_link_libraries(${CMAKE_PROJECT_NAME}
        #依赖库列表
```

```
        android
        log)
    #设置库的输出目录
    set_target_properties(${CMAKE_PROJECT_NAME} PROPERTIES
LIBRARY_OUTPUT_DIRECTORY
${CMAKE_LIBRARY_OUTPUT_DIRECTORY_${ANDROID_ABI}})
endfunction()

message("-----------------------------------------   ${ANDROID_ABI}")
#调用函数开始编译
build_library()
```

以上展示的即为 CMakeLists.txt 文件的全部内容。对于不熟悉 CMake 语法的读者来讲，这些内容可能初看起来较为复杂，但可以先耐心阅读注释，有助于大致了解这个文件的主要作用。

总体而言，这份 CMake 配置文件主要实现了以下几个功能：首先，它导入了必要的头文件和源文件；其次，设置了编译输出的目录；接着，根据配置生成了相应的库文件；最后，还包含了链接库的操作指令。

特别需要强调的是，CMake 配置文件接收了一个关键的外部参数 ANDROID_ABI。该参数的主要作用是明确指定编译的目标架构，以便根据不同的架构设置合适的输出目录。此外，除了 ANDROID_ABI 变量外，外部还会传入一系列其他参数，如 ANDROID_NDK 和 ANDROID_PLATFORM 等。这些参数均是为 android.toolchain.cmake 这个编译工具链文件所设计的，它们共同协助配置出符合要求的编译环境。这些参数会通过 cmake 命令传入 CMake 文件中供其使用。

接下来编写一个脚本文件，用以执行动态库的编译工作。开发者可以通过这一脚本自动化地完成编译工作，确保生成的动态库符合预期的架构要求。

5. 编译脚本文件编写

编写脚本与手动执行命令在功能层面上没有本质区别，两者的核心都执行一系列的命令以完成特定的任务，然而，脚本文件的主要优势在于，它可以帮助我们自动化地完成那些需要重复执行的操作或者完成那些复杂的命令行操作，从而减少了手动输入时可能出现的错误，提高了工作效率。

首先，创建一个名为 build.sh 的文件。为了便于理解，以最简单的脚本开始编写，内容如下：

```
#第 3 章/build.sh
#使用 bash
#!/bin/bash

#删除编译目录，保持每次编译的干净环境
rm build -rf
```

```
#删除产物目录，保证每次编译出的产物都是最新的
rm libs -rf

#重新创建编译目录，CMake 编译会产生很多中间文件，在一个单独的目录中编译便于清理
mkdir build

#进入编译目录
cd build

#执行 cmake 命令，生成 Makefile
cmake -DCMAKE_TOOLCHAIN_FILE=$NDK_ROOT/build/cmake/android.toolchain.cmake \
      -DANDROID_ABI="armeabi-v7a" \
      -DANDROID_NDK=$ANDROID_NDK \
      -DANDROID_PLATFORM=android-22 \
      ..

#执行编译命令
make
```

脚本编写完成后，使用 chmod a+x build.sh 命令为 build.sh 文件增加可执行权限，此时的目录结构如图 3-8 所示。

```
test@test-host:~/develop/3.1.2$ tree
.
├── build.sh
├── CMakeLists.txt
├── include
│   ├── Add.h
│   └── LogUtils.h
└── src
    └── Add.c

2 directories, 5 files
```

图 3-8　源文件目录结构（2）

执行脚本，输出如图 3-9 所示。

```
test@test-host:~/develop/3.1.2$ ./build.sh
-- The C compiler identification is Clang 17.0.2
-- The CXX compiler identification is Clang 17.0.2
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /opt/android-ndk-r26b/toolchains/llvm/prebuilt/linux-x86_64/bin/clang - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /opt/android-ndk-r26b/toolchains/llvm/prebuilt/linux-x86_64/bin/clang++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
------------------------------------------   armeabi-v7a
-- Configuring done
-- Generating done
-- Build files have been written to: /home/test/develop/3.1.2/build
[ 50%] Building C object CMakeFiles/add.dir/src/Add.c.o
[100%] Linking C shared library ../libs/armeabi-v7a/libadd.so
[100%] Built target add
```

图 3-9　编译输出

此时再看目录结构，如图 3-10 所示。

图 3-10 源文件目录结构

在编译过程中，生成了两个重要的目录：build 和 libs，其中，build 目录主要用于存放编译过程中生成的中间文件，这些文件是构建系统在编译源代码时自动生成的，对于一般的用户来讲，通常不需要特别关注这些文件的内容。

而 libs 目录则包含了最终编译得到的库文件。特别地，在 libs 目录下的 armeabi-v7a 子目录中，可以发现名为 libadd.so 的动态链接库文件。这个文件是针对 ARMv7-A 架构编译得到的产物，是编译过程的最终成果。

接下来，回顾编译脚本中使用的 cmake 命令。这个命令在构建系统中扮演着至关重要的角色，它接收了 5 个参数，其中 CMAKE_TOOLCHAIN_FILE 在 3.1.2 节中已经讲解过，这里不再赘述，其余 4 个参数的作用如下。

1）ANDROID_ABI

ANDROID_ABI 指定了当前要编译的动态库的 ABI，脚本指定的内容会被 android.toolchain.cmake 文件解析，并根据传入的 ABI 选择合适的工具链。

2）ANDROID_NDK

ANDROID_NDK 指定了 ndk 的根目录，用来寻找相关工具链。

3）ANDROID_PLATFORM

ANDROID_PLATFORM 指定了动态库兼容的最小 Android 版本。

4）..

这个参数用来指定 CMakeLists.txt 所在的目录。..在 Linux 系统中代表上一级目录，告知 cmake 要执行的 CMakeLists.txt 所在的相对路径。

所有使用“-D”定义的变量均可在 android.toolchain.cmake 和 CMakeLists.txt 文件中通过“${使用-D 定义的变量}”来获得该变量的值，从而可以通过定义不同的变量来指定不同的编译目标。

6. 升级编译脚本

观察 3.1.2 节中的 build.sh 脚本文件可以发现，ANDROID_ABI 的值决定了编译出来动态库的架构。那么，只需在脚本中动态地改变此变量的值便可完成不同架构动态库的编译。

在脚本中，可以利用数组和循环结构来满足该需求。升级后的脚本如下：

```
#第 3 章/build.sh
```

```
#使用 bash
#!/bin/bash

#删除编译目录，保持每次编译的干净环境
rm build -rf

#删除产物目录，保证每次编译出的产物都是最新的
rm libs -rf

#重新创建编译目录，CMake 编译会产生很多中间文件，在一个单独的目录中编译便于清理
mkdir build

#进入编译目录
cd build

#定义一个数组
ARCHS=('armeabi-v7a' 'arm64-v8a' 'x86' 'x86_64' )


#定义一个函数
function compile(){
    #利用 for 循环来循环获取数组中的字符串，赋值给 ANDROID_ABI，再调用 make 命令
        for i in ${ARCHS[@]}; do
                cmake -DCMAKE_TOOLCHAIN_FILE= \
                        $NDK_ROOT/build/cmake/android.toolchain.cmake \
                       -DANDROID_ABI="$i" \
                       -DANDROID_NDK=$ANDROID_NDK \
                       -DANDROID_PLATFORM=android-22 \
                       ..
                make
        done
}



#函数调用，开始编译
compile
```

执行脚本，输出如图 3-11 所示。

此时再看目录结构，libs 目录下除 armeabi-v7a 外还多出了 arm64-v8a、x86 及 x86_64，如图 3-12 所示。

3.1.3 使用 Android Studio 直接集成预编译库

在 3.1.2 节中，已详细地阐述了如何利用命令行编译出与平台相匹配的预编译库。本节将聚焦于如何利用 Android Studio 这一集成开发环境，将预编译库集成至项目中，进而对相

```
test@test-host:~/develop/3.1.2$ ./build.sh
-- The C compiler identification is Clang 17.0.2
-- The CXX compiler identification is Clang 17.0.2
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /opt/android-ndk-r26b/toolchains/llvm/prebuilt/linux-x86_64/bin/clang - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /opt/android-ndk-r26b/toolchains/llvm/prebuilt/linux-x86_64/bin/clang++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
----------------------------------------------   armeabi-v7a
-- Configuring done
-- Generating done
-- Build files have been written to: /home/test/develop/3.1.2/build
[ 50%] Building C object CMakeFiles/add.dir/src/Add.c.o
[100%] Linking C shared library ../libs/armeabi-v7a/libadd.so
[100%] Built target add
----------------------------------------------   arm64-v8a
-- Configuring done
-- Generating done
-- Build files have been written to: /home/test/develop/3.1.2/build
Consolidate compiler generated dependencies of target add
[ 50%] Building C object CMakeFiles/add.dir/src/Add.c.o
[100%] Linking C shared library ../libs/arm64-v8a/libadd.so
[100%] Built target add
----------------------------------------------   x86
-- Configuring done
-- Generating done
-- Build files have been written to: /home/test/develop/3.1.2/build
Consolidate compiler generated dependencies of target add
[ 50%] Building C object CMakeFiles/add.dir/src/Add.c.o
[100%] Linking C shared library ../libs/x86/libadd.so
[100%] Built target add
----------------------------------------------   x86_64
-- Configuring done
-- Generating done
-- Build files have been written to: /home/test/develop/3.1.2/build
Consolidate compiler generated dependencies of target add
[ 50%] Building C object CMakeFiles/add.dir/src/Add.c.o
[100%] Linking C shared library ../libs/x86_64/libadd.so
[100%] Built target add
```

图 3-11　编译输出

```
test@test-host:~/develop/3.1.2$ tree
.
├── build
├── build.sh
├── CMakeLists.txt
├── include
│   ├── Add.h
│   └── LogUtils.h
├── libs
│   ├── arm64-v8a
│   │   └── libadd.so
│   ├── armeabi-v7a
│   │   └── libadd.so
│   ├── x86
│   │   └── libadd.so
│   └── x86_64
│       └── libadd.so
└── src
    └── Add.c

8 directories, 9 files
```

图 3-12　源文件列表

关功能进行二次开发。

通过 Android Studio 集成预编译库，开发者能够更便捷地扩展和优化现有功能，而无须从头编写全部代码。这不仅能提升开发效率，还能确保代码的稳定性和可靠性。

在集成预编译库时，需确保预编译库与项目在平台版本、架构类型等方面的兼容性。随后，按照 Android Studio 的导入流程，将预编译库添加到项目的依赖中。

一旦预编译库被成功地集成至项目中，便可开始功能的二次开发。这包括调用预编译库中的函数和方法，实现特定的业务逻辑，以及对现有功能进行优化和扩展。利用预编译库提供的强大功能，可以快速地构建出满足需求的应用程序。

综上所述，通过 Android Studio 集成预编译库进行功能的二次开发，能够充分地利用已有资源，提升开发效率，同时确保应用程序的质量和稳定性。这将为开发者带来更便捷和更高效的开发体验。

1. 工程创建

参考 3.1.1 节完成 Native 工程的创建和配置。将工程命名为 Ndk3_1.2，如图 3-13 所示。

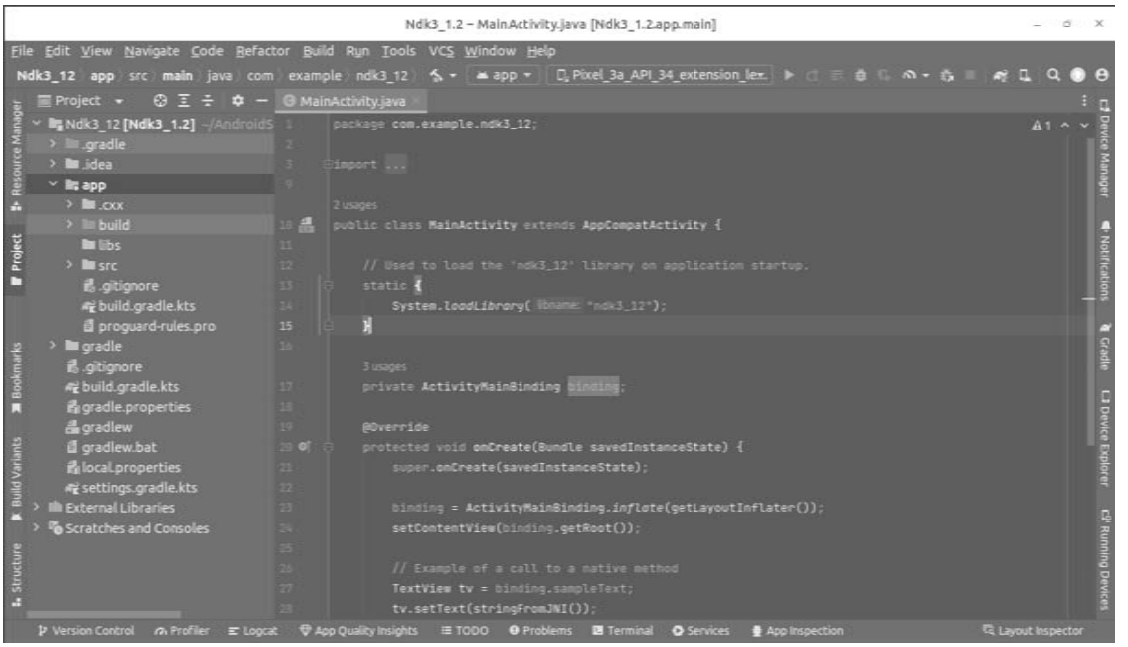

图 3-13　Native 工程

2. 导入库与头文件

将 3.1.2 节生成的预编译库复制到工程的 app/libs 目录中，并将头文件复制到 src/main/cpp 目录中，如图 3-14 所示。

3. 工程配置

在工程配置中，有两个关键的文件起到了至关重要的作用，分别是 CMakeLists.txt 和 build.gradle.kts。

1）CMakeLists.txt 配置

在 CMakeLists.txt 文件中添加对预构件库和头文件的依赖，修改后的 CMakeLists.txt 文件中的代码如下：

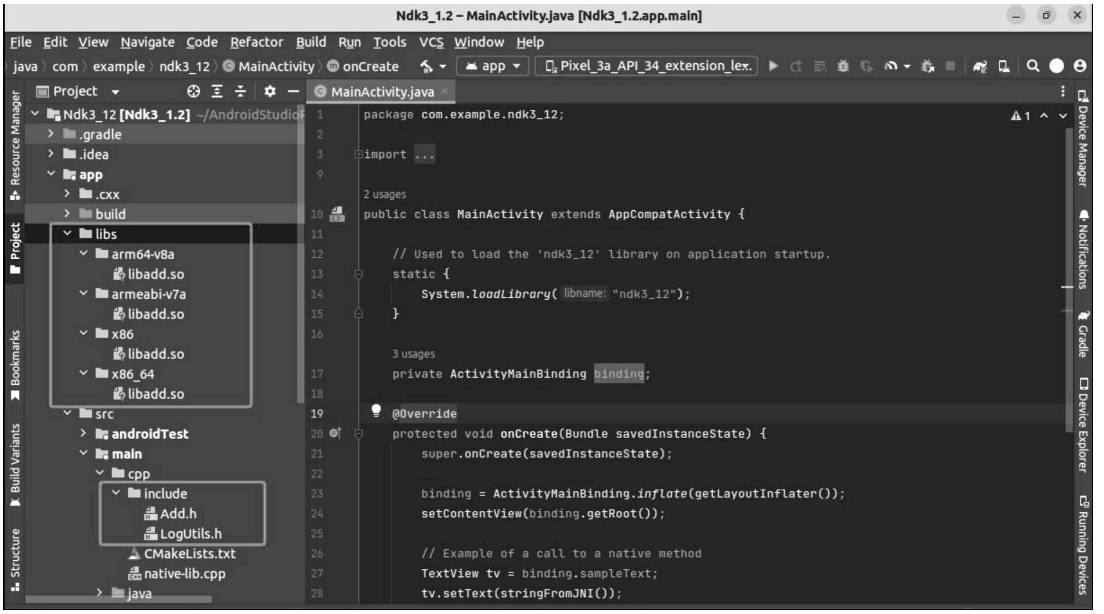

图 3-14　导入库与头文件

```
#第 3 章/CMakeLists.txt
#CMake 的最低版本
cmake_minimum_required(VERSION 3.22.1)
#项目名称
project("ndk3_12")

#设置依赖库路径，ANDROID_ABI 表示当前编译的指令集
set(LIBDIR ${CMAKE_CURRENT_SOURCE_DIR}/../../../libs/${ANDROID_ABI})
#包含头文件
include_directories(include)
#预构建库的导入
add_library(add SHARED IMPORTED)
#指定库的位置
set_target_properties(add PROPERTIES IMPORTED_LOCATION ${LIBDIR}/libadd.so)

#生成动态库
add_library(${CMAKE_PROJECT_NAME} SHARED
        #源文件列表
        native-lib.cpp)

#链接库
target_link_libraries(${CMAKE_PROJECT_NAME}
        #依赖的链接库列表
        android
        log
```

```
        #链接预编译库
        add
        )
```

与工程默认生成的 CMakeLists.txt 文件相比，我们对其进行了扩展和定制，添加了多个关键配置，以便更好地支持库的依赖路径设置、预编译库的导入、头文件的包含及对预编译库的链接操作。

首先，利用 ANDROID_ABI 这个内置变量的值动态地生成了库的路径，并将该路径保存到 LIBDIR 这个变量中，然后使用 add_library 命令的 IMPORTED 特性告知系统导入了一个预构建库。通过 set_target_properties 命令指定预构建库的路径，CMake 能够在构建过程中将这些库集成到项目中，使项目能够利用这些库提供的功能。

此外，通过 include_directories 命令配置了头文件的包含路径。这确保了项目中的源代码能够正确地包含和引用预构建库的头文件，从而实现了对库功能的调用和使用。

最后，通过 target_link_libraries 命令对预编译库进行了链接操作。这确保了项目在编译过程中能够正确地链接到预构建库，生成可执行文件或库时能够包含预编译库中的代码和数据。

2）build.gradle.kts 配置

在默认创建的 Native 工程构建脚本中，通常已经预设了对 CMakeLists.txt 的引用配置，这使开发者在大多数情况下无须进行过多的修改即可顺利地进行构建，然而，在软件开发的实际过程中，项目往往依赖于多种预构建库来实现其功能。

在默认情况下，工程构建脚本会尝试将所有预构建库打包进 APK 中，然而，当某个依赖库缺少对应指令集的库文件时，这可能会导致编译失败或在运行时出现错误。为了解决这一问题，开发者通常需要根据实际需求对库的依赖进行过滤。

通过对库的依赖进行过滤，可以确保只将必要的库文件打包进 APK 中，从而避免因为缺少对应指令集的库而导致出现编译或运行时问题。这一步骤对于保证工程的正常编译和运行至关重要。

因此，在开发过程中，开发者应该仔细审查并管理项目的依赖库，确保它们与目标平台的指令集兼容，并根据需要进行适当过滤和配置。这样做不仅可以提高构建的成功率，还可以减小 APK 的大小，优化应用的性能。

截至本书撰写之际，Android Studio 支持两种工程构建脚本，分别是基于 Groovy 语法的构建脚本和基于 Kotlin 语法的构建脚本。由于两者在实际开发中均有广泛应用，所以本书特提供两种配置方式，旨在满足不同读者的需求，使他们能够根据自己使用的脚本类型进行相应配置。

（1）使用 Groovy 语法构建脚本的开发者，可参考如下配置，代码如下：

```
defaultConfig {
    applicationId " com.example.ndk3_12"
    minSdk 24
    targetSdk 33
```

```
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
        ndk {
            //开发者根据需要过滤需要的指令集
            abiFilters 'armeabi', 'armeabi-v7a', 'arm64-v8a', 'x86', 'x86_64'
        }
    }
```

（2）使用 Kotlin 构建脚本的开发者，可参考如下配置，代码如下：

```
android {
    namespace = "com.example.ndk3_12"
    compileSdk = 33

    defaultConfig {
    applicationId = "com.example.ndk3_12"
    minSdk = 24
    targetSdk = 33
    versionCode = 1
    versionName = "1.0"

    testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"

    ndk{
        //仅打包过滤器中包含的架构的库
        abiFilters.addAll(arrayOf("armeabi-v7a", "arm64-v8a", "x86", "x86_64"))
    }
}
```

通过提供这两种配置方式，本书旨在为读者提供更全面和更灵活的指导，使读者能够根据自己的实际情况选择最适合的构建脚本配置方法，从而提高开发效率，确保项目的顺利进行。

注意：此配置特指 App 级别的 build.gradle 文件中的配置。在编写或修改配置文件时，可根据 build.gradle 文件的后缀来判断所使用的语法类型。若后缀为.gradle，则使用的是 Groovy 语法；若后缀为.kts，则使用的是 Kotlin 语法。区分不同的语法类型有助于开发者更准确地理解并应用配置文件中的各项设置，确保项目的构建过程能够顺利地进行，因此，在进行配置时，务必注意文件后缀，并根据实际情况选择合适的语法进行编写。

4. 使用预构建库提供的功能

在默认生成的 native-lib.cpp 文件中，添加对库函数的调用，代码如下：

```
//第 3 章/native-lib.cpp
#include <jni.h>
```

```
#include <string>
//包含库提供的头文件
#include "Add.h"

//定义 log 的 TAG
#define TAG "native-lib"
extern "C" JNIEXPORT jstring JNICALL
Java_com_example_ndk3_112_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    std::string hello = "Hello from C++";
    //调用库中的功能函数
    int ret = add(1,2);
    LOGI("ret = %d", ret);
    return env->NewStringUTF(hello.c_str());
}
```

上述代码成功地包含了预构建库的头文件 Add.h，并调用了该库提供的 add 方法。该方法被用于执行加法运算，并返回计算结果。单击“运行”按钮，运行结果如图 3-15 所示。

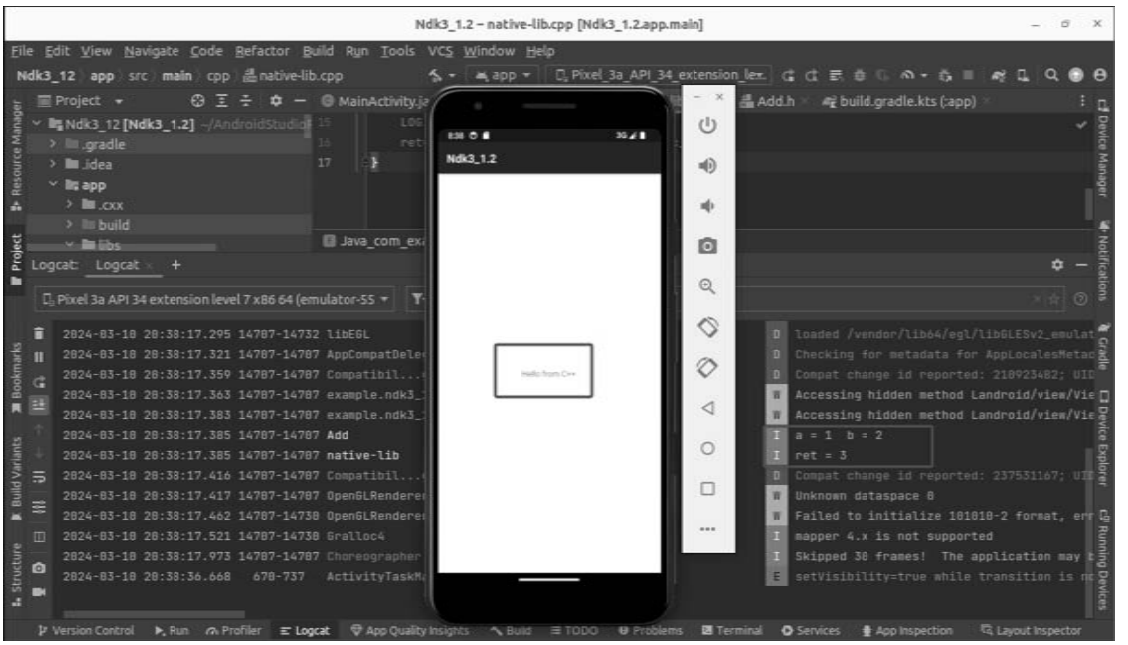

图 3-15　运行结果

通过上面的操作，成功地利用了预构建库的功能，实现了加法运算，并验证了代码的正确性，其结果和源码集成并无不同，这展示了在软件开发中，如何有效地利用现有的预构建库来提高开发效率和代码质量。