Android NDK 开发笔记

deepwaterooo

May 16, 2019

Contents

1 Android 之 JNI 开发详解 - NDK 从入门到精通

- https://my.oschina.net/sfshine/blog/781644

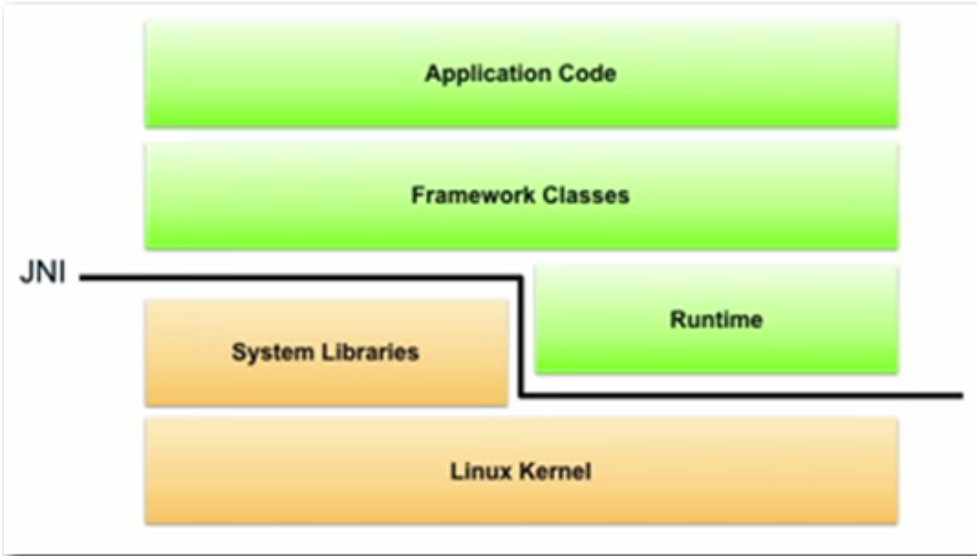
1.1 JNI 引入

- JNI 概念: Java 本地接口, Java Native Interface, 它是一个协议, 该协议用来沟通 Java 代码和外部的本地 C/C++ 代码, 通过该协议 Java 代码可以调用外部的本地代码, 外部的 C/C++ 代码可以调用 Java 代码;
- C 和 Java 的侧重:
  - **C 语言**: C 语言中最重要的是函数 function;
  - **Java 语言**: Java 中最重要的是 JVM, class 类, 以及 class 中的方法;
- C 与 Java 如何交流:

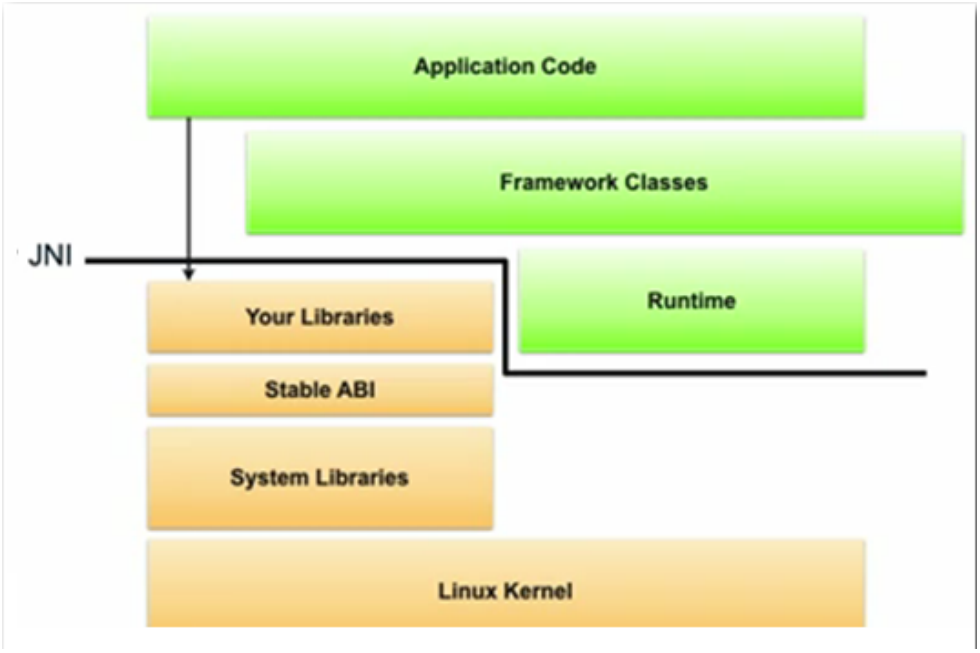
- **JNI 规范**: C 语言与 Java 语言交流需要一个适配器, 中间件, 即 JNI, JNI 提供了一种规范;
- **C 语言中调用 Java 方法**: 可以让我们在 C 代码中找到 Java 代码 class 中的方法, 并且调用该方法;
- **Java 语言中调用 C 语言方法**: 同时也可以在 Java 代码中, 将一个 C 语言的方法映射到 Java 的某个方法上;
- **JNI 桥梁作用**: JNI 提供了一个桥梁, 打通了 C 语言和 Java 语言之间的障碍;

1.2 Android 中的应用程序框架

- 正常情况下的 Android 框架:
  - 最顶层是 Android 的应用程序代码, 是纯 Java 代码, 中间有一层的 Framework 框架层代码是 C/C++ 代码, 通过 Framework 进行系统调用, 调用底层的库和 linux 内核;



- 使用 JNI 时的 Android 框架:
  - 绕过 Framework 提供的调用底层的代码, 直接调用自己写的 C 代码, 该代码最终会编译成为一个库, 这个库通过 JNI 提供的一个 Stable 的 ABI 调用 linux kernel;ABI 是二进制程序接口 application binary interface.



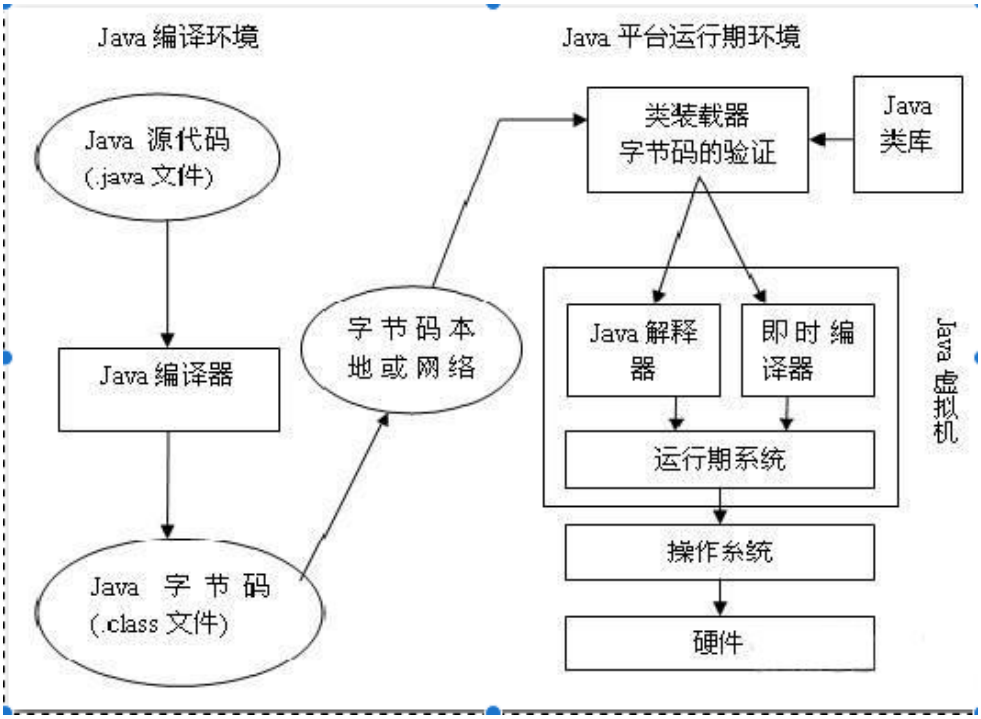
1.3 JNI 作用

1.3.1 JNI 作用:

- **扩展**: JNI 扩展了 JVM 能力, 驱动开发, 例如开发一个 wifi 驱动, 可以将手机设置为无限路由;
- **高效**: 本地代码效率高, 游戏渲染, 音频视频处理等方面使用 JNI 调用本地代码, C 语言可以灵活操作内存;
- **复用**: 在文件压缩算法 7zip 开源代码库, 机器视觉 openCV 开放算法库等方面可以复用 C 平台上的代码, 不必在开发一套完整的 Java 体系, 避免重复发明轮子;
- **特殊**: 产品的核心技术一般也采用 JNI 开发, 不易破解;

1.3.2 Java 语言执行流程:

- **编译字节码**: Java 编译器编译.java 源文件, 获得.class 字节码文件;
- **装载类库**: 使用类装载器装载平台上的 Java 类库, 并进行字节码验证;
- **Java 虚拟机**: 将字节码加入到 JVM 中, Java 解释器和即时编译器同时处理字节码文件, 将处理后的结果放入运行时系统;
- **调用 JVM 所在平台类库**: JVM 处理字节码后, 转换成相应平台的操作, 调用本平台底层类库进行相关处理;



- **Java 一次编译到处执行**: JVM 在不同的操作系统都有实现, Java 可以一次编译到处运行, 字节码文件一旦编译好了, 可以放在任何平台的虚拟机上运行;

1.4 NDK 详解: 交叉编译库文件

1.4.1 C 代码执行:

- C 代码被编译成库文件之后, 才能执行, 库文件分为动态库和静态库两种;
  - **动态库**: unix 环境下.so 后缀的是动态库, windows 环境下.dll 后缀的是动态库; 动态库可以依赖静态库加载一些可执行的 C 代码;
  - **静态库**: .a 后缀是静态库的扩展名;

1.4.2 库文件来源:

- C 代码进行编译链接操作之后, 才会生成库文件, 不同类型的 CPU 操作系统生成的库文件是不一样;
  - **CPU 分类**: arm 结构, 嵌入式设备处理器; x86 结构, pc 服务器处理器; 不同的 CPU 指令集不同;
  - **交叉编译**: windows x86 编译出来的库文件可以在 arm 平台运行的代码;
  - **交叉编译工具链**: Google 提供的 NDK 就是交叉编译工具链, 可以在 linux 环境下编译出在 arn 平台下执行的二进制库文件;
- **NDK 作用**:
  - 是 Google 提供了交叉编译工具链, 能够在 linux 平台编译出在 arm 平台下执行的二进制库文件;
- **NDK 版本介绍**:
  - android-ndk-windows 是在 windows 系统中的 cygwin 使用的, android-ndk-linux 是在 linux 下使用的;

2 Android studio NDK ndk-build 编译 C 生成.so 文件步骤

- https://blog.csdn.net/Mr_55/article/details/79773728

2.1 android.useDeprecatedDdk

- 新建一个 demo 项目 jnidemo 来记录 JNI 开发流程，项目创建完毕，打开 gradle.properties 文件，输入

```
android.useDeprecatedDdk=true
```

, 否者后面编译时会提示相关 (so 库兼容错误) 错误。

2.2 配置 NDK

- 配置 gradle, 打开 app 的 gradle，在 defaultConfig 下面加入 NDK 的编译配置，这里的 moduleName 跟 loadLibrary 的时候用的名字必须相同：

```
  android {
      ndk {
          moduleName "myfirstndk" // 指定生成的 so 文件名
          abiFilters "x86", "x86_64", "arm64-v8a" // cpu 的类型
      }
      sourceSets.main {
          jni.srcDirs = [] // 屏蔽掉默认的 jni 编译生成过程
          jniLibs.srcDir "src/main/libs"
      }
  }
  externalNativeBuild {
      ndkBuild {
          path "src/main/jni/Android.mk"
      }
  }
```

2.3 JniUtil.java

- 源文件

```
package com.myfirstndk;
public class JniUtil {
    static {
        // 加载生成 .so 文件名称
        System.loadLibrary("myfirstndk"); // 名字必须和 build.gradle 中的 moduleName
    }
    public static native String sayHello(); // 底层映射
    // native 为本地的意思，顾名思义就是被表示为 sayHello() 函数是从本地映射上来的函数
}
```

- System.loadLibrary() 方法加载动态库。(如果动态库的名字为 libwgr.so，那么我们应该去掉文件名的 lib 和.so，只把中间的 wgr 作为参数传递到方法中进行加载)

2.4 编译

- 用 AS 自带的命令行，进入到项目文件夹目录，输入命令 javac JniUtil.java，将 java 文件编译成.class 文件：

```
javac JniUtil.java
```

2.5 生成头文件

- 输入命令

```
javah -jni com.myfirstndk.JniUtil // （包名 + 类名）
```

- 如果报找不到该类的错误，用

```
javah -classpath . -jni com.myfirstndk.JniUtil
```

- 此命令将生成与 JniUtil 类对应的.h 文件。成功后将在包目录下生成一个 com_myfirstndk_JniUtil.h 文件。
- javah -jni -d /自己想放入的目录 (但是要方便自己寻找) -classpath . jni.JniPlug(切记 JniPlug 这个 class 文件在命令中是不要加.class 后缀否则出错)
- 这样我们就有了需要的头文件了，不过我们要将这个头文件拷贝到我们项目中的一个制定文件夹，我的文件夹在 src\jni 文件夹内为什么要建立这么一个文件夹呢？我们还需要配置 Android.mk 与 Application.mk 否则我们会在 NDK-BUILD 过程中报错。

2.6 jni 文件夹

- 新建一个 jni 文件夹，将刚刚生产的.h 文件剪切到 jni 文件夹下面，并创建一个 C/C++ 文件 hello.c
- hello.c 的方法名必须跟.h 文件中的方法名一致。这里的方法内容返回“HelloWorld!”

```
#include "com_myfirstndk_JniUtil.h"
JNIEXPORT jstring JNICALL Java_com_myfirstndk_JniUtil_sayHello(JNIEnv *env, jclass j
    return (*env)->NewStringUTF(env, "Hello World~!");
}
```

2.7 建立 android.mk

```
LOCAL_PATH := $(call my-dir)  # 为调用当前目录

include $(CLEAR_VARS)         # 清空无用变量

LOCAL_MODULE    := myfirstndk # 指定的是: 生成的库文件的名字（动态? 静态?)
LOCAL_SRC_FILES := hello.c    # 关联的是 jni 目录下的.c文件

# for logging
LOCAL_LDLIBS    += -llog

include $(BUILD_SHARED_LIBRARY) # 构建后成为共享库
```

2.8 建立 Application.mk

```
APP_ABI := all  # ABI全部构建
```

2.9 通过 ndk-build 来生产库文件

- 通过 alt+F12 进入我们存放.h 与 Android.mk Application.mk 文件的目录后，执行下面的命令
- 命令行执行:

```
ndk-build NDK_PROJECT_PATH=. NDK_APPLICATION_MK=Application.mk APP_BUILD_SCRIPT=Andr
```

- 解析
  - ndk-build 为 ndk 命令方法, 默认的不用管
  - NDK_PROJECT_PATH=. 这是当前目录了，因为我们切换进来了在这里生产共享库文件
  - NDK_APPLICATION_MK=Application.mk 我们之前建立的文件
  - APP_BUILD_SCRIPT=Android.mk 我们之前建立的文件

- 然后我们需要将生产的好库的上级目录一起拷贝到 build.gradle 文件中关键字 sourceSets.main 设置的目录中就大功告成了

2.10 Java 配置项目

```java
package com.myfirstndk;

import android.widget.TextView;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        TextView tv = new TextView(this);
        tv.setText("the following came from jni: " + JniUtil.sayHello());
    }
}
```

3 Android NDK 开发（1）—— Java 与 C 互相调用

- https://abc20899.iteye.com/blog/1861121

- callback.c 文件如下：

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <jni.h>
#include <android/log.h>

#define LOGI(...) ((void)__android_log_print(
ANDROID_LOG_INFO, "native-activity", __VA_ARGS__))
#define LOGW(...) ((void)__android_log_print(
ANDROID_LOG_WARN, "native-activity", __VA_ARGS__))

/********** 传输整数 ************* */
JNIEXPORT void JNICALL Java_com_nan_callback_MyCallbackActivity_callJNIInt(
```

```
    JNIEnv* env, jobject obj , jint i) {
    // 找到 java 中的类
    jclass cls = (*env)->FindClass(env, "com/nan/callback/MyCallbackActivity");
    // 再找类中的方法
    jmethodID mid = (*env)->GetMethodID(env, cls, "callbackInt", "(I)V");
    if (mid == NULL)    {
        LOGI("int error");
        return;
    }
    // 打印接收到的数据
    LOGI("from java int: %d",i);
    // 回调 java 中的方法
    (*env)->CallVoidMethod(env, obj, mid ,i);

}

/******** 传输字符串 **************/
JNIEXPORT void JNICALL Java_com_nan_callback_MyCallbackActivity_callJNIString(
    JNIEnv* env, jobject obj , jstring s)   {
    // 找到 java 中的类
    jclass cls = (*env)->FindClass(env, "com/nan/callback/MyCallbackActivity");
    // 再找类中的方法
    jmethodID mid = (*env)->GetMethodID(
        env, cls, "callbackString", "(Ljava/lang/String;)V");
    if (mid == NULL)    {
        LOGI("string error");
        return;
    }
    const char *ch;
    // 获取由 java 传过来的字符串
    ch = (*env)->GetStringUTFChars(env, s, NULL);
    // 打印
    LOGI("from java string: %s",ch);
    (*env)->ReleaseStringUTFChars(env, s, ch);
    // 回调 java 中的方法
    (*env)->CallVoidMethod(env, obj, mid ,(*env)->NewStringUTF(env," 你好 haha"));

}

/******** 传输数组 (byte[])**************/
JNIEXPORT void JNICALL Java_com_nan_callback_MyCallbackActivity_callJNIByte(
    JNIEnv* env, jobject obj , jbyteArray b)   {
    // 找到 java 中的类
    jclass cls = (*env)->FindClass(env, "com/nan/callback/MyCallbackActivity");
    // 再找类中的方法
    jmethodID mid = (*env)->GetMethodID(env, cls, "callbackByte", "([B)V");
    if (mid == NULL)    {
        LOGI("byte[] error");
        return;
    }

    // 获取数组长度
    jsize length = (*env)->GetArrayLength(env,b);
```

```
73      LOGI("length: %d",length);
74      // 获取接收到的数据
75      int i;
76      jbyte* p = (*env)->GetByteArrayElements(env,b,NULL);
77      // 打印
78      for(i=0;i<length;i++)   {
79          LOGI("%d",p[i]);
80      }
81
82      char c[5];
83      c[0] = 1;c[1] = 2;c[2] = 3;c[3] = 4;c[4] = 5;
84      // 构造数组
85      jbyteArray carr = (*env)->NewByteArray(env,length);
86      (*env)->SetByteArrayRegion(env,carr,0,length,c);
87      // 回调 java 中的方法
88      (*env)->CallVoidMethod(env, obj, mid ,carr);
89  }
```