一、谈谈你对 JNI 和 NDK 的理解

JNI：

JNI 是 `Java Native Interface` 的缩写，即 Java 的本地接口。

目的是使得 Java 与本地其他语言（如 C/C++）进行交互。

JNI 是属于 Java 的，与 Android 无直接关系。

NDK：

NDK 是 `Native Development Kit` 的缩写，是 Android 的工具开发包。

作用是更方便和快速开发 C/C++ 的动态库，并自动将动态库与应用一起打包到 apk。

NDK 是属于 Android 的，与 Java 无直接关系。

总结：

JNI 是实现的目的，NDK 是 Android 中实现 JNI 的手段。

二、谈谈你对 JNIEnv 和 JavaVM 理解

JavaVM

JavaVM 是虚拟机在 JNI 层的代表。

一个进程只有一个 JavaVM。（重要！）

所有的线程共用一个 JavaVM。（重要！）

JNIEnv

JNIEnv 表示 Java 调用 native 语言的环境，封装了几乎全部 JNI 方法的指针。

JNIEnv 只在创建它的线程生效，不能跨线程传递，不同线程的 JNIEnv 彼此独立。（重要！）

注意：

在 native 环境下创建的线程，要想和 java 通信，即需要获取一个 JNIEnv 对象。我们通过 `AttachCurrentThread` 和 `DetachCurrentThread` 方法将 native 的线程与 JavaVM 关联和解除关联。

三、解释一下 JNI 中全局引用和局部引用的区别和使用

全局引用

通过 `NewGlobalRef` 和 `DeleteGlobalRef` 方法创建和释放一个全局引用。

全局引用能在多个线程中被使用，且不会被 GC 回收，只能手动释放。

局部引用

通过 `NewLocalRef` 和 `DeleteLocalRef` 方法创建和释放一个局部引用。

局部引用只在创建它的 native 方法中有效，包括其调用的其它函数中有效。因此我们不能寄望于将一个局部引用直接保存在全局变量中下次使用（请使用全局引用实现该需求）。

我们可以不用删除局部引用，它们会在 native 方法返回时全部自动释放，但是建议对于不再使用的局部引用手动释放，避免内存过度使用。

扩展：弱全局引用

通过 `NewWeakGlobalRef` 和 `DeleteWeakGlobalRef` 创建和释放一个弱全局引用。

弱全局引用类似于全局引用，唯一的区别是它不会阻止被 GC 回收。

四、JNI 线程间数据怎么互相访问

考察点和上体类似，线程本来就是共享内存区域的，因此我们需要使用 `全局引用`

动态注册

原理：利用 RegisterNatives 方法来注册 java 方法与 JNI 函数的一一对应关系；

实现流程：

1. 利用结构体 JNINativeMethod 数组记录 java 方法与 JNI 函数的对应关系；
2. 实现 JNI_OnLoad 方法，在加载动态库后，执行动态注册；
3. 调用 FindClass 方法，获取 java 对象；
4. 调用 RegisterNatives 方法，传入 java 对象，以及 JNINativeMethod 数组，以及注册数目完成注册；

优点：

1. 流程更加清晰可控；
2. 效率更高；

```
jstring stringFromJNI(JNIEnv *env, jobject thiz){
    std::string hello = "Hello from C++";
    return env->NewStringUTF(hello.c_str());
}

static const JNINativeMethod gMethods[] = {
        {"stringFromJNI", "()Ljava/lang/String;", (jstring*)stringFromJNI}
};

JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved){

    __android_log_print(ANDROID_LOG_INFO, "native", "Jni_OnLoad");
    JNIEnv* env = NULL;
    if(vm->GetEnv((void**)&env, JNI_VERSION_1_4) != JNI_OK) //从JavaVM获取JNIEnv，
一般使用1.4的版本
        return -1;
```

```
    jclass clazz = env->FindClass("com/example/efan/jni_learn2/MainActivity");
    if (!clazz){
        __android_log_print(ANDROID_LOG_INFO, "native", "cannot get class:
com/example/efan/jni_learn2/MainActivity");
        return -1;
    }
    if(env->RegisterNatives(clazz, gMethods,
sizeof(gMethods)/sizeof(gMethods[0])))
    {
        __android_log_print(ANDROID_LOG_INFO, "native", "register native method
failed!\n");
        return -1;
    }
    return JNI_VERSION_1_4;
}
12345678910111213141516171819202122232425262728
```

JNINativeMethod

在动态注册的过程中使用到了结构体 JNINativeMethod 用于记录 java 方法与 jni 函数的对应关系

```
typedef struct {
    const char* name;
    const char* signature;
    void*       fnPtr;
} JNINativeMethod;12345
```

结构体的第一个参数 name 是java 方法名；

第二个参数 signature 用于描述方法的参数与返回值；

第三个参数 fnPtr 是函数指针，指向 jni 函数；

其中，第二个参数 signature 使用字符串记录方法的参数与返回值，具体格式形如“()V”、“(II)V”，其中分为两部分，括号内表示的是参数，括号右侧表示的是返回值；

数据类型映射

1. 基本数据类型

java 类型	native 类型	域描述符	补充
boolean	jboolean	Z	
byte	jbyte	B	
char	jchar	C	
short	jshort	S	
int	jint	I	
long	jlong	J	
float	jfloat	F	
double	jdouble	D	
void	void	V	

\2. 数组引用类型

如果是一维数组则遵循下表，如果是二维数组或更高维数组则对应的 native 类型为 jobjectArray，域描述符中使用 '[' 的个数表示维数

java 类型	native 类型	域描述符	补充
int[]	jintArray	[I	
float[]	jfloatArray	[f	
byte[]	jbyteArray	[B	
char[]	jcharArray	[C	
short[]	jshortArray	[S	
double[]	jdoubleArray	[D	
long[]	jlongArray	[F	
boolean[]	jbooleanArray	[Z	

\3. 对象引用类型

对于其它引用类型，即 java 中的对象，其映射规则为

java 类型	native 类型	域描述符	补充
类名（如 Surface）	通常是 jobject，仅有一种例外，如果 java 类型是 String，则对应的native 类型是 jstring	以"L"开头，以";"结尾中间是用"/" 隔开的包及类名（如 Landroid/view/Surface;）如果内部类则使用$连接内部类；	

\4. 对象数组引用类型

如果是一维数组则遵循下表，如果是二维数组或更高维数组则对应的 native 类型为 jobjectArray，域描述符中使用 '[' 的个数表示维数

java 类型	native 类型	域描述符	补充
类名（如 Surface）	通常是 jobject，仅有一种例外，如果 java 类型是 String，则对应的native 类型是 jstring	在对象引用类型的域描述符的基础上在左边添加'['字符	

jni 函数默认参数

在 jni 函数中有两个默认参数

```
JNIEnv *env, jobject thiz1
```

其中 JNIEnv 指代的是当前 java 环境，可以利用 JNIEnv 可以操作 java 层代码；jobject 指代的是 jni 函数对应的 java native 方法的类实例，如果 java 方法是 static，则代表的是 class 对象；