Java Native Interface Programming学习记录

JNI概要

什么是JNI

java native interface (jni)是native programing interface.它允许运行在jvm虚拟机里的Java代码与其他语言（例如C/C++/汇编）编写的application和libraries进行相互操作。

JNI最重要的优势是它不会限制底层java VM的实现，因此Java VM供应商可以添加对JNI的支持又不会影响VM的其他部分。程序员期望可以写一个版本的的native application或者library运行在所有支持JNI的VM上。

什么时候需要使用JNI

有时候你的应用不能只使用java来实现所有的需求，需要使用JNI调用java native methods来实现。

下面是需要使用java native methods的几种情况：

1. 标准的java lib不能满足需求。
2. 已经有一个使用其它语言实现好的lib,我们希望直接使用jni来让这个库访问java code.
3. 你想使用更低级的语言(比如：汇编)来实现一小部分执行效率更高的code.

使用JNI编程，你可以使用native method做如下的事：

1. 创建、检查、更新java objects.
2. 调用Java methods.
3. 捕获和throw Exception.
4. 加载类和获取类的信息.
5. 执行运行时类型检查.

You can also use the JNI with the *Invocation API* to enable an arbitrary native application to embed the Java VM. This allows programmers to easily make their existing applications Java-enabled without having to link with the VM source code.

JNI的历史

不同VM的供应商提供了不同的native method interfaces.不同的interfaces要求工程师为不同的平台创建、维护、发布不同的native methods library.

下面是不同的native method interfaces:

1. JDK 1.0 native method interfaces.
2. Netscape's Java Runtime Interface.
3. Microsoft's Raw Native Interface and Java/COM interface.

JNI的目标

统一的、经过深思熟虑的interfaces标准会给每个人带来下面的好处：

1. Each VM vendor can support a larger body of native code.
2. Tool builders will not have to maintain different kinds of native method interfaces.
3. Application programmers will be able to write one version of their native code and this version will run on different VMs.

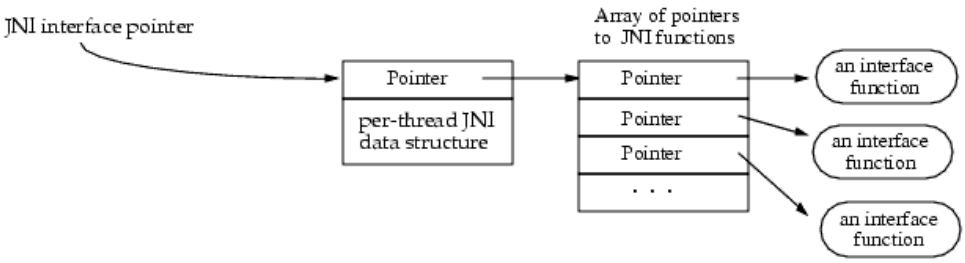
标准的native method interfaces必须满足下面的要求：

1. **Binary compatibility**（二进制兼容性）- The primary goal is binary compatibility of native method libraries across all Java VM implementations on a given platform. Programmers should maintain only one version of their native method libraries for a given platform.
2. **Efficiency**（效率） - To support time-critical code, the native method interface must impose little overhead. All known techniques to ensure VM-independence (and thus binary compatibility) carry a certain amount of overhead. We must somehow strike a compromise between efficiency and VM-independence.
3. **Functionality**（功能性） - The interface must expose enough Java VM internals to allow native methods to accomplish useful tasks.

JNI Design Overview

JNI Interface Functions and Pointers

native方法是通过JNI Functions来访问Java VM features,而JNI Functions需要通过Interface pointer来进行使用。Interface pointer是指向另一个指针（简称指针B）的指针。指针B指向的是一个指针数值。数组中的每个指针分别指向一个interface function.具体结构如下：



JNI interface pointer的组织结构类似于C++的virtual function table或COM interface.**使用interface table而不是采用"hard-wired function entries"的设计优势是这样设计JNI name space就跟native code独立起来（解耦）。一个VM可以容易地提供多个版本的JNI function tables。**比如，VM可以支持两种JNI function tables:

- one performs thorough(['θʌrə] 彻底的) illegal argument checks, and is suitable for debugging;
- the other performs the minimal amount of checking required by the JNI specification, and is therefore more efficient.

JNI interface pointer只在当前线程有效，native method不能将interface pointer从一个线程传到另外一个线程去。实现JNI的VM可以在JNI interface pointer指向的区域中分配和存储线程本地数据。

JNI interface pointer是作为参数传给native method的。VM在同一java 线程中多次调用某个native method时，VM会保证传递同一个interface pointer给该native method.然后同一个native method可能被多个不同的java线程调用，因此该native method可能会收到不同的JNI interface pointer.

Compiling,Loading and Linking Native Methods

Native method通过System.loadLibrary被加载。loadLibrary将接收lib name的参数，不同的平台会有相应的规则将lib name转换成对应的native library lib.比如linux系统将lib name转换成lib_(lib name).so，windows系统将lib name转换成"lib name".dll。

```
public class HelloWordJNISample {
    static{
        System.loadLibrary("hello");//linux系统加载libhello.so库，windows系统加载hello.dll
                                    //该库包含了native方法sayHello的实现
    }

    public static native String sayHello(String name); // 1.声明这是一个native函数，由本地代码实现

    public static void main(String[] args) {
        String text = sayHello("jni world");  // 3.调用本地函数
        System.out.println(text);
    }
}
```

A native library may be *statically linked* with the VM. The manner in which the library and VM image are combined is implementation dependent. A `System.loadLibrary` or equivalent API must succeed for this library to be considered loaded.

native library可以与vm静态进行链接。库和VM的组合方式取决于实现

A library L whose image has been combined with the VM is defined as statically linked if and only if the library exports a function called `JNI_OnLoad_L`.

if and only if:当且仅当

当且仅当library L exports a JNI_OnLoad_L函数，library L和vm 的组合方式才被认定为statically linked.

如果library L是静态链接库,有如下特点:

1. 如果静态链接库export JNI_OnLoad_L 和JNI_OnLoad方法，JNI_OnLoad将会被忽略。
2. 如果library L是静态链接库，当第一次调用System.loadLibrary("L")或通过等效的api来加载L库，会执行JNI_OnLoad_L。
3. 静态链接的库L将禁止动态加载同名的库。
4. 当包含静态链接的本机库L的类加载器被垃圾回收时，VM将会调用该库的JNI_OnUnload_L函数
5. 当静态链接库L export JNI_OnUnLoad_L和JNI_OnUnLoad函数，JNI_OnUnLoad将会被忽略。

程序员可以调用 JNI function RegisterNatives()去注册native method. RegisterNatives()函数对于静态链接的函数特别有用。

native method的注册有两种方式，下面是https://developer.android.com/training/articles/perf-jni中的描述

```
There are two ways that the runtime can find your native
methods. You can either explicitly register them with
`RegisterNatives`, or you can let the runtime look them up
dynamically with `dlsym`.

```

Resolving Native Method Names

动态链接器是通过native method name来解析entries的。native method name是通过下面的部分连接而成：

- 前缀Java_
- a mangled fully-qualified class name ('/'--> "_")
- an underscore (“_”) separator
- a mangled method name
- for overloaded native methods, two underscores (“__”) followed by the mangled argument signature

  Take note of the native function naming convention: `Java_<*fully-qualified-name*>_methodName`, with dots replaced by underscores.

概念补充：

1. The **fully-qualified** name for a class or interface is the package name followed by the class/interface name, separated by a period (.). Code outside the package can reference public classes and interfaces of a package using their **fully-qualified** names.
2. **Name mangling** is the encoding of **function** and variable **names** into unique **names** so that linkers can separate common **names** in the language. Type

> **names** may also be **mangled**. **Name mangling** is commonly used to facilitate the overloading feature and visibility within different scopes.

short name:the name without the argument signature

long name: the name with the argument signature

VM检查会根据name到native library进行匹配，VM先按short name进行查询，然后在按照long name进行查询。

We adopted a simple name-mangling scheme to ensure that all Unicode characters translate into valid C function names. We use the underscore (“_”) character as the substitute for the slash (“/”) in fully qualified class names.

根据上面的描述获得如下信息：

1. **这里的fully qualified names是用"/"来分隔的。**
2. **采取的name mangling策略是将full qualified class names的"/"替换成"_"。**
3. **采用这种name mangling策略的原因是为了确保所有的Unicode字符可以转换成有效的C function names。**

Since a name or type descriptor never begins with a number, we can use `_0`, ..., `_9` for escape sequences, as the following table illustrates:

使用`_0`, ..., `_9` 当做转义符。

	UNICODE CHARACTER TRANSLATION
Escape Sequence	Denotes
`_0XXXX`	a Unicode character `XXXX`. Note that lower case is used to represent non-ASCII Unicode characters, for example, `_0abcd` as opposed to `_0ABCD`.
`_1`	the character “_”
`_2`	the character “;” in signatures
`_3`	the character “[“ in signatures

jni转义test demo：

java 文件如下

```
package  com.practise.jni;
public class JniEscapeSequences {
    static{
        System.loadLibrary("helloescape");
    }

    public native String test_escape_sequences(String name); //1.声明这是一个native函数，在函数名中添加”_“

    public static void main(String[] args) {
        JniEscapeSequences jniEscapeSequenceObj = new JniEscapeSequences();
```

```
        String text =
jniEscapeSequenceObj.test_escape_sequences("test jni escape
sequences");  // 3.调用本地函数
        System.out.println(text);
    }
}

```

生成的.h头文件如下：

```
android@jinyang:~/local/practise_smaple/jni_test2$ cat
com_practise_jni_JniEscapeSequences.h
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_practise_jni_JniEscapeSequences */

#ifndef _Included_com_practise_jni_JniEscapeSequences
#define _Included_com_practise_jni_JniEscapeSequences
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_practise_jni_JniEscapeSequences
 * Method:    test_escape_sequences
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL
Java_com_practise_jni_JniEscapeSequences_test_1escape_1sequence
s
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif

```

**test_escape_sequences被转义成了
Java_com_practise_jni_JniEscapeSequences_test_1escape_1sequences。**

Native Method Arguments

native method的第一个参数是Jni interface pointer,该参数的类型为JNIEnv

native method的第二个参数依赖于该native method是static还是非static:

- **static native method对应的第二个参数为jclass**
- **non-static native method对应的第二个参数为jobject**

native method的剩余参数是将java 层接收的参数类型转换成jni 的数据类型。java 数据类型与jni 数据类型有对应的映射表。

Referencing Java Objects

Accessing Java Objects

Java Exceptions

JNI Types and Data Structures

该章节主要介绍jni将java类型map到native C类型。（java层数据类型与native 数据类型的映射关系）

JNI Type主要有基本类型（primitive types）和 引用类型（Reference Types）两大类。

JNI Types and Native Equivalents

JAVA TYPE	NATIVE TYPE	DESCRIPTION
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits
void	void	not applicable
Object	jobject	
Class	jclass	
String	jstring	
Object[]	jobjectArray	
boolean[]	jbooleanArray	
byte[]	jbyteArray	
char[]	jcharArray	

JAVA TYPE	NATIVE TYPE	DESCRIPTION
short[]	jshortArray	
int[]	jintArray	
long[]	jlongArray	
float[]	jfloatArray	
double[]	jdoubleArray	

jni reference types类层次结构如下：

jobject

- `jclass` (`java.lang.Class` objects)
- `jstring` (`java.lang.String` objects)
- ```
  1 jarray(arrays)
  ```

  - `jobjectArray` (object arrays)
  - `jbooleanArray` (`boolean` arrays)
  - `jbyteArray` (`byte` arrays)
  - `jcharArray` (`char` arrays)
  - `jshortArray` (`short` arrays)
  - `jintArray` (`int` arrays)
  - `jlongArray` (`long` arrays)
  - `jfloatArray` (`float` arrays)
  - `jdoubleArray` (`double` arrays)
- `jthrowable` (`java.lang.Throwable` objects)

Field and Method IDs

jfieldID和jmethodID这两种类型都是指针。

```
1 struct _jfieldID;              /* opaque structure */
2 typedef struct _jfieldID *jfieldID;   /* field IDs */
3
4 struct _jmethodID;              /* opaque structure */
5 typedef struct _jmethodID *jmethodID; /* method IDs */
```

jni具体有哪些数据类型可以才看如下文件android/prebuilts/jdk/jdk11/linux-x86/include/jni.h
```

Type Signatures

jni使用的是jvm的type signature，所以我们可以通过javap工具来进行转换。不用死记硬背。

TYPE SIGNATURE	JAVA TYPE
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L fully-qualified-class ;	fully-qualified-class
[type	type[]
(arg-types) ret-type	method type

通过javap工具转换得到的type signatures如下：

```
android@jinyang:~/local/practise_smaple/jni_test2$ javap -s
com/practise/jni/JniEscapeSequences.class
Compiled from "JniEscapeSequences.java"
public class com.practise.jni.JniEscapeSequences {
  public com.practise.jni.JniEscapeSequences();
    descriptor: ()V

  public native java.lang.String
test_escape_sequences(java.lang.String);
    descriptor: (Ljava/lang/String;)Ljava/lang/String;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V

  static {};
    descriptor: ()V
}
android@jinyang:~/local/practise_smaple/jni_test2$
```

根据上面的信息我们可以得知JNI中的fully-qualified-class是使用“/”进行分隔的。

JNI Functions

JNI Functions包含如下的内容：

- Interface Function Table
- Version Information
- Class Operations
- Exceptions

- Global and Local References
- Weak Global References
- Object Operations
- Accessing Fields of Objects
- Calling Instance Methods
- Accessing Static Fields
- Calling Static Methods
- String Operations
- Array Operations
- Registering Native Methods
- Monitor Operations
- NIO Support
- Reflection Support
- Java VM Interface

这一章的内容其实就是Jni Api手册，我们根据自己的需求查找并使用对应的JNI functions。

The Invocation API

The Invocation API allows software vendors to load the Java VM into an arbitrary native application. Vendors can deliver Java-enabled applications without having to link with the Java VM source code.

The Invocation API allows a native application to use the JNI interface pointer to access VM features.

这章的内容主要介绍了如何创建VM,lib库的管理以及Invocation API的介绍。通过这章的学习知道在native应用中是如何创建vm,获取jni interface等。

小结

1. 在native应用里面通过Invocation API来创建VM并拿到Interface pointer。
2. 通过拿到的Interface pointer调用JNI Funtions来访问Java VM features.native方法通过jni interface从而实现c调用java.
3. native method的注册方式有两种：静态注册和动态注册

   静态注册：使用javah工具生成头文件(有 JNIEXPORT 和 JNICALL 两个宏定义声明)，实现的方法名格式 `Java_<*fully-qualified-name*>_methodName`

   ```
   JNIEXPORT jstring JNICALL
   Java_com_practise_jni_HelloWordJNISample_sayHello
     (JNIEnv *, jclass, jstring);
   ```

   动态注册：库的实现中重写JNI_OnLoad方法并通过调用RegisterNatives()进行注册。

```
见android/development/samples/SimpleJNI/jni/native.cpp

 static JNINativeMethod methods[] = {
40    {"add", "(II)I", (void*)add },
41  };


46  static int registerNativeMethods(JNIEnv* env, const char* className,
47      JNINativeMethod* gMethods, int numMethods)
48  {
49      jclass clazz;
50
51      clazz = env->FindClass(className);
52      if (clazz == NULL) {
53          ALOGE("Native registration unable to find class '%s'", className);
54          return JNI_FALSE;
55      }
56      if (env->RegisterNatives(clazz, gMethods, numMethods) < 0) {
57          ALOGE("RegisterNatives failed for '%s'", className);
58          return JNI_FALSE;
59      }
60
61      return JNI_TRUE;
62  }
```

如何使用JNI

创建HelloWordJNISample.java类并定义native方法：

```
package  com.practise.jni;
public class HelloWordJNISample {
    static{
        System.loadLibrary("hello");//linux系统加载libhello.so库，windows系统加载hello.dll
                                    //该库包含了native方法sayHello的实现
    }

    public static native String sayHello(String name); // 1.声明这是一个native函数，由本地代码实现

```

```
    public static void main(String[] args) {
        String text = sayHello("jni world");  // 3.调用本地函数
        System.out.println(text);
    }
}
```

生成相应的C/C++头文件的方法：

```
方法一：
通过javac -h来生成头文件（从java8开始支持该命令）。
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 12
drwxrwxr-x 2 android android 4096 12月 28 14:47 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  389 12月 28 14:47 HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$
android@jinyang:~/local/practise_smaple/jni_test2$
//如果java source中定义了相应的packname,下面的命令生成的class路径会有问题，具体问题见“**执行class文件，查看运行结果”
android@jinyang:~/local/practise_smaple/jni_test2$ javac -h ./ HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 20
drwxrwxr-x 2 android android 4096 12月 28 14:47 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  554 12月 28 14:47 com_practise_jni_HelloWordJNISample.h
-rw-rw-r-- 1 android android  629 12月 28 14:47 HelloWordJNISample.class
-rw-rw-r-- 1 android android  389 12月 28 14:47 HelloWordJNISample.java

方法二（javah命令从jdk10开始不支持）：
Before JDK 8, you need to compile the Java program using javac and generate C/C++ header using a dedicated javah utility, as follows. The javah utility is no longer available in JDK 10.

android@jinyang:~/local/practise_smaple/jni_test2$ javac -d ./ HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 16
drwxrwxr-x 3 android android 4096 12月 28 15:13 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
drwxrwxr-x 3 android android 4096 12月 28 15:13 com
-rw-rw-r-- 1 android android  389 12月 28 15:13 HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ javah -jni -classpath ./ com.practise.jni.HelloWordJNISample
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 20
```

```
drwxrwxr-x 3 android android 4096 12月 28 15:14 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
drwxrwxr-x 3 android android 4096 12月 28 15:13 com
-rw-rw-r-- 1 android android  554 12月 28 15:14 com_practise_jni_HelloWordJNISample.h
-rw-rw-r-- 1 android android  389 12月 28 15:13 HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ tree ./
./
├── com
│   └── practise
│       └── jni
│           └── HelloWordJNISample.class
├── com_practise_jni_HelloWordJNISample.h
└── HelloWordJNISample.java

3 directories, 3 files
android@jinyang:~/local/practise_smaple/jni_test2$

或者是：
android@jinyang:~/local/practise_smaple/jni_test2$ javac -d ./ HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ tree ./
./
├── com
│   └── practise
│       └── jni
│           └── HelloWordJNISample.class
└── HelloWordJNISample.java

3 directories, 2 files
android@jinyang:~/local/practise_smaple/jni_test2$ javah HelloWordJNISample
Error: Could not find class file for 'HelloWordJNISample'.
android@jinyang:~/local/practise_smaple/jni_test2$ javah com.practise.jni.HelloWordJNISample
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 20
drwxrwxr-x 3 android android 4096 12月 28 15:23 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
drwxrwxr-x 3 android android 4096 12月 28 15:22 com
-rw-rw-r-- 1 android android  554 12月 28 15:23 com_practise_jni_HelloWordJNISample.h
-rw-rw-r-- 1 android android  389 12月 28 15:13 HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ tree ./
./
├── com
│   └── practise
│       └── jni
│           └── HelloWordJNISample.class
├── com_practise_jni_HelloWordJNISample.h
```

```
└── HelloWordJNISample.java

3 directories, 3 files


下面的方式不可行：
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 12
drwxrwxr-x 2 android android 4096 12月 28 15:16 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  389 12月 28 15:13
HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ javac
HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 16
drwxrwxr-x 2 android android 4096 12月 28 15:16 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  629 12月 28 15:16
HelloWordJNISample.class
-rw-rw-r-- 1 android android  389 12月 28 15:13
HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ javah
HelloWordJNISample
Error: Could not find class file for 'HelloWordJNISample'.
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 16
drwxrwxr-x 2 android android 4096 12月 28 15:16 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  629 12月 28 15:16
HelloWordJNISample.class
-rw-rw-r-- 1 android android  389 12月 28 15:13
HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ javah
HelloWordJNISample
Error: Could not find class file for 'HelloWordJNISample'.

android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 12
drwxrwxr-x 2 android android 4096 12月 28 15:30 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  389 12月 28 15:13
HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ javac
HelloWordJNISample.java
lsandroid@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 16
drwxrwxr-x 2 android android 4096 12月 28 15:30 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  629 12月 28 15:30
HelloWordJNISample.class
```

```
-rw-rw-r-- 1 android android  389 12月 28 15:13 HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ javah com.practise.jni.HelloWordJNISample
Error: Could not find class file for 'com.practise.jni.HelloWordJNISample'.
android@jinyang:~/local/practise_smaple/jni_test2$


```

上面javac和javah命令的含义直接通过javac -help和javac -help查看。默认生成的头文件的命名方式为packageName+className（"."替换成"_"）.我们也可以指定头文件的名称：如下我们将生成的头文件直接命名成HelloWordJNISample.h

```
android@jinyang:~/local/practise_smaple/jni_test2$ javah -o HelloWordJNISample.h com.practise.jni.HelloWordJNISample
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 24
drwxrwxr-x 3 android android 4096 12月 28 15:52 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
drwxrwxr-x 3 android android 4096 12月 28 15:34 com
-rw-rw-r-- 1 android android  554 12月 28 15:35 com_practise_jni_HelloWordJNISample.h
-rw-rw-r-- 1 android android  554 12月 28 15:52 HelloWordJNISample.h
-rw-rw-r-- 1 android android  389 12月 28 15:13 HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ tree ./
./
├── com
│   └── practise
│       └── jni
│           └── HelloWordJNISample.class
├── com_practise_jni_HelloWordJNISample.h
├── HelloWordJNISample.h
└── HelloWordJNISample.java

3 directories, 4 files


```

生成的头文件内容如下：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_practise_jni_HelloWordJNISample */

#ifndef _Included_com_practise_jni_HelloWordJNISample
#define _Included_com_practise_jni_HelloWordJNISample
#ifdef __cplusplus
extern "C" {
```

```
#endif
/*
 * Class:     com_practise_jni_HelloWordJNISample
 * Method:    sayHello
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL
Java_com_practise_jni_HelloWordJNISample_sayHello
  (JNIEnv *, jclass, jstring);

#ifdef __cplusplus
}
#endif
#endif

```

实现头文件

```
#include <jni.h>
#include "com_practise_jni_HelloWordJNISample.h"
JNIEXPORT jstring JNICALL
Java_com_practise_jni_HelloWordJNISample_sayHello(JNIEnv *env,
jclass className, jstring printContent){
    const char *c_str = NULL;
    c_str = (*env)->GetStringUTFChars(env, printContent, NULL);
    return (*env)->NewStringUTF(env,c_str);
}
```

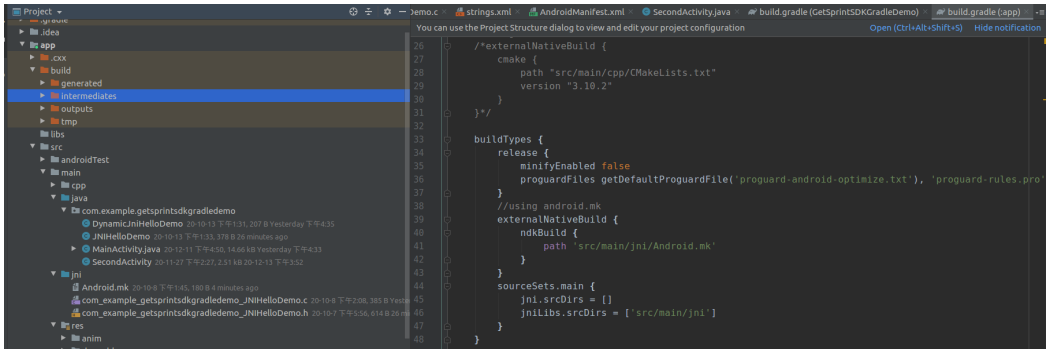
使用gcc命令生成libhello.so

```
android@jinyang:~/local/practise_smaple/jni_test2$ gcc -fPIC -
I"$JAVA_HOME/include" -I"$JAVA_HOME/include/linux" -shared -o
libhello.so HelloWordJNISample.c
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 32
drwxrwxr-x 2 android android 4096 12月 28 16:51 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
-rw-rw-r-- 1 android android  554 12月 28 16:38
com_practise_jni_HelloWordJNISample.h
-rw-rw-r-- 1 android android  335 12月 28 16:46
HelloWordJNISample.c
-rw-rw-r-- 1 android android  629 12月 28 16:38
HelloWordJNISample.class
-rw-rw-r-- 1 android android  389 12月 28 15:13
HelloWordJNISample.java
-rwxrwxr-x 1 android android 7914 12月 28 16:51 libhello.so

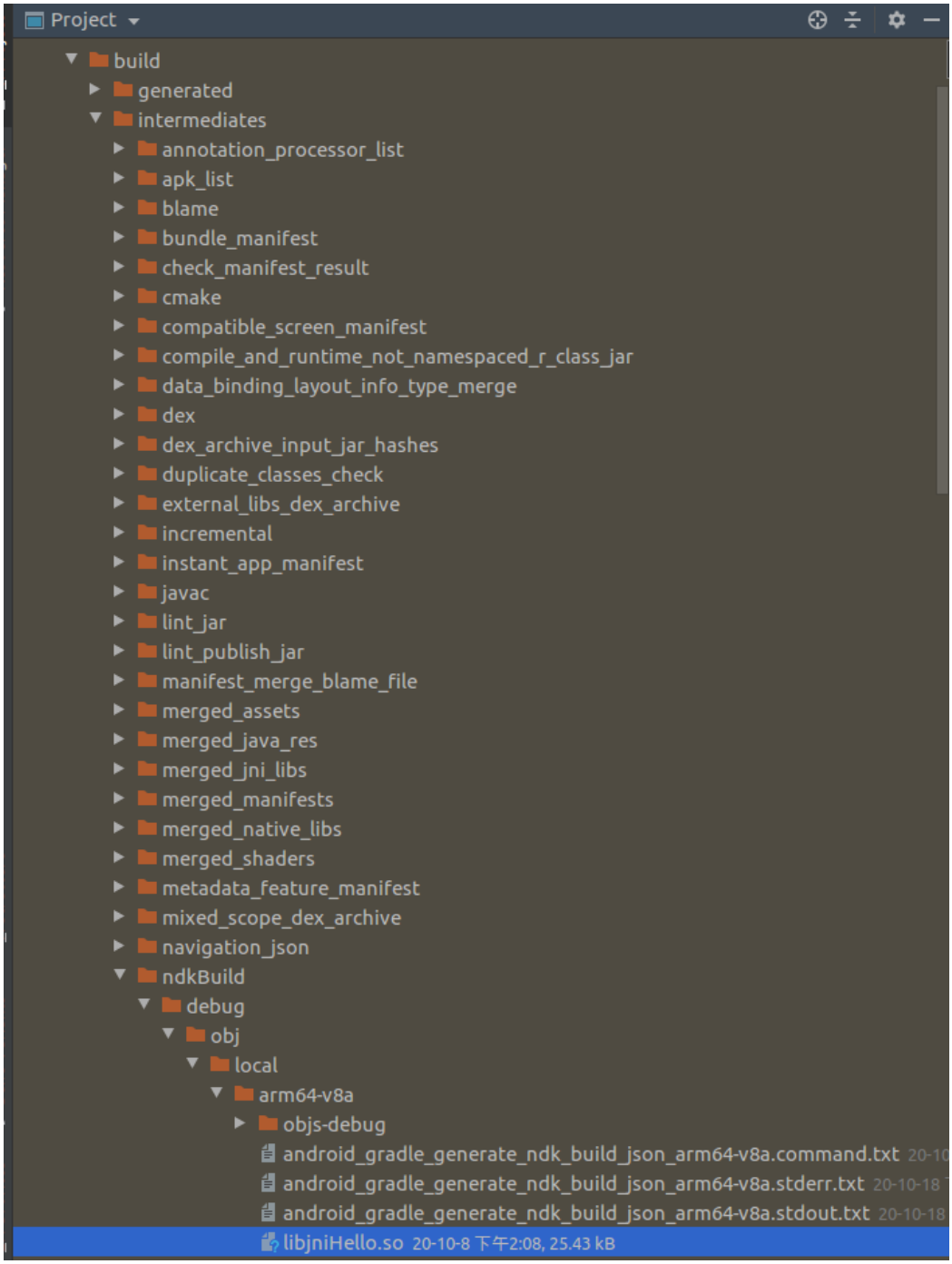

gcc的具体操作命令可以通过如下命令学习：
man gcc
```

除了使用gcc，在Android Studio中还可以通过gradle+Android.mk/gradle+CMake来编译相应的so文件.

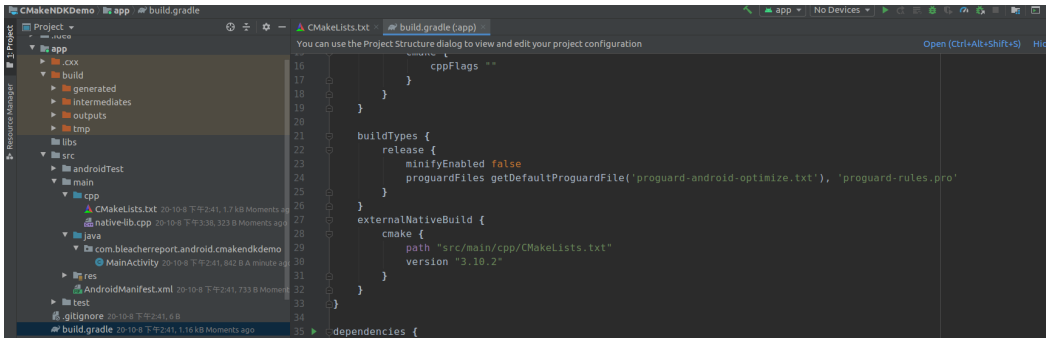
gradle+Android.mk的目录结构如下：



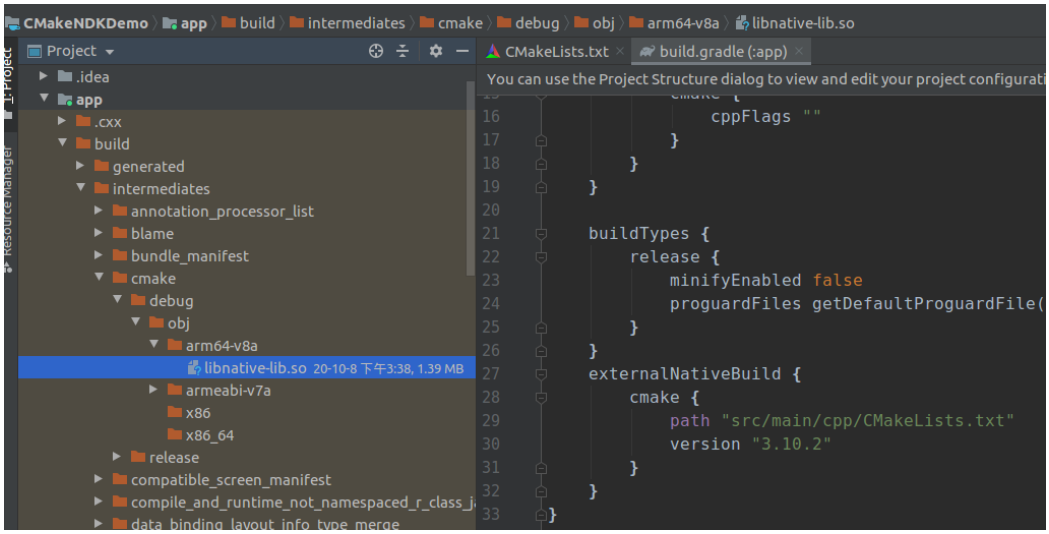
在gradle文件中根据path指定Android.mk的路径以及so库编译出来的位置。



gradle+CMake的目录结构如下：



在gradle文件中根据path指定Android.mk的路径以及so库编译出来的位置。



执行class文件，查看运行结果

```
遇到如下错误：
android@jinyang:~/local/practise_smaple/jni_test2$ java -Djava.library.path=. HelloWordJNISample
Error: Could not find or load main class HelloWordJNISample

开始怀疑跟包名有关，进行如下尝试：
android@jinyang:~/local/practise_smaple/jni_test2$ java -Djava.library.path=. com.practise.jni.HelloWordJNISample
Error: Could not find or load main class com.practise.jni.HelloWordJNISample

根据生成头文件过程的经验，可能和目录有关。根据packagename创建相应的目录，将class移动到packagename目录下进行如下尝试：
android@jinyang:~/local/practise_smaple/jni_test2$ mkdir -p com/practise/jni/
android@jinyang:~/local/practise_smaple/jni_test2$ ls -al
total 36
drwxrwxr-x 3 android android 4096 12月 28 17:06 .
drwxrwxr-x 5 android android 4096 12月 28 14:41 ..
drwxrwxr-x 3 android android 4096 12月 28 17:06 com
-rw-rw-r-- 1 android android  335 12月 28 16:46 com_practise_jni_HelloWordJNISample.c
```

```
-rw-rw-r-- 1 android android  554 12月 28 17:00 com_practise_jni_HelloWordJNISample.h
-rw-rw-r-- 1 android android  629 12月 28 17:00 HelloWordJNISample.class
-rw-rw-r-- 1 android android  389 12月 28 15:13 HelloWordJNISample.java
-rwxrwxr-x 1 android android 7931 12月 28 16:56 libhello.so
android@jinyang:~/local/practise_smaple/jni_test2$ mv HelloWordJNISample.class com
com/                                  com_practise_jni_HelloWordJNISample.c
com_practise_jni_HelloWordJNISample.h
android@jinyang:~/local/practise_smaple/jni_test2$ mv HelloWordJNISample.class com/practise/jni/
android@jinyang:~/local/practise_smaple/jni_test2$ tree ./
./
├── com
│   └── practise
│       └── jni
│           └── HelloWordJNISample.class
├── com_practise_jni_HelloWordJNISample.c
├── com_practise_jni_HelloWordJNISample.h
├── HelloWordJNISample.java
└── libhello.so

3 directories, 5 files
android@jinyang:~/local/practise_smaple/jni_test2$ java -Djava.library.path=. com.practise.jni.HelloWordJNISample
jni world
android@jinyang:~/local/practise_smaple/jni_test2$
```

解决上面错误的关键是将class文件放到相应的packageName路径下，根据这种思路尝试如下方案：

```
android@jinyang:~/local/practise_smaple/jni_test2$ javac -d ./ -h ./ HelloWordJNISample.java
android@jinyang:~/local/practise_smaple/jni_test2$ tree ./
./
├── com
│   └── practise
│       └── jni
│           └── HelloWordJNISample.class
├── com_practise_jni_HelloWordJNISample.h
├── HelloWordJNISample.c
└── HelloWordJNISample.java

3 directories, 4 files
android@jinyang:~/local/practise_smaple/jni_test2$ gcc -fPIC -I"$JAVA_HOME/include" -I"$JAVA_HOME/include/linux" -shared -o libhello.so HelloWordJNISample.c
```

```
android@jinyang:~/local/practise_smaple/jni_test2$ tree ./
./
├── com
│   └── practise
│       └── jni
│           └── HelloWordJNISample.class
├── com_practise_jni_HelloWordJNISample.h
├── HelloWordJNISample.c
├── HelloWordJNISample.java
└── libhello.so

3 directories, 5 files
android@jinyang:~/local/practise_smaple/jni_test2$ java com.practise.jni.HelloWordJNISample
Exception in thread "main" java.lang.UnsatisfiedLinkError: no hello in java.library.path
    at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1867)
    at java.lang.Runtime.loadLibrary0(Runtime.java:870)
    at java.lang.System.loadLibrary(System.java:1122)
    at com.practise.jni.HelloWordJNISample.<clinit>(HelloWordJNISample.java:4)
android@jinyang:~/local/practise_smaple/jni_test2$ java -Djava.library.path=. com.practise.jni.HelloWordJNISample
welcome to jni world（修改了传入的字符串，所以和上次输出的不一样）
android@jinyang:~/local/practise_smaple/jni_test2$
```

相关参考资料

https://www.baeldung.com/jni

https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

https://developer.android.com/training/articles/perf-jni

https://www.codetd.com/en/article/7397888

http://gityuan.com/2016/05/28/android-jni/

https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html

https://www.baeldung.com/java-could-not-find-load-main-class

https://en.wikipedia.org/wiki/Name_mangling

https://stackoverflow.com/questions/1314743/what-is-name-mangling-and-how-does-it-work

https://astro.uni-bonn.de/~sysstw/CompMan/gnu/gcc/gxxint_15.html

https://blog.csdn.net/xinkuang126/article/details/107578201

https://www.cs.vu.nl/~eliens/design/hush/scratch/hush/documents/java/tutorial/native1.1/implementing/index.html