NDK 开发入门

---周绍卿

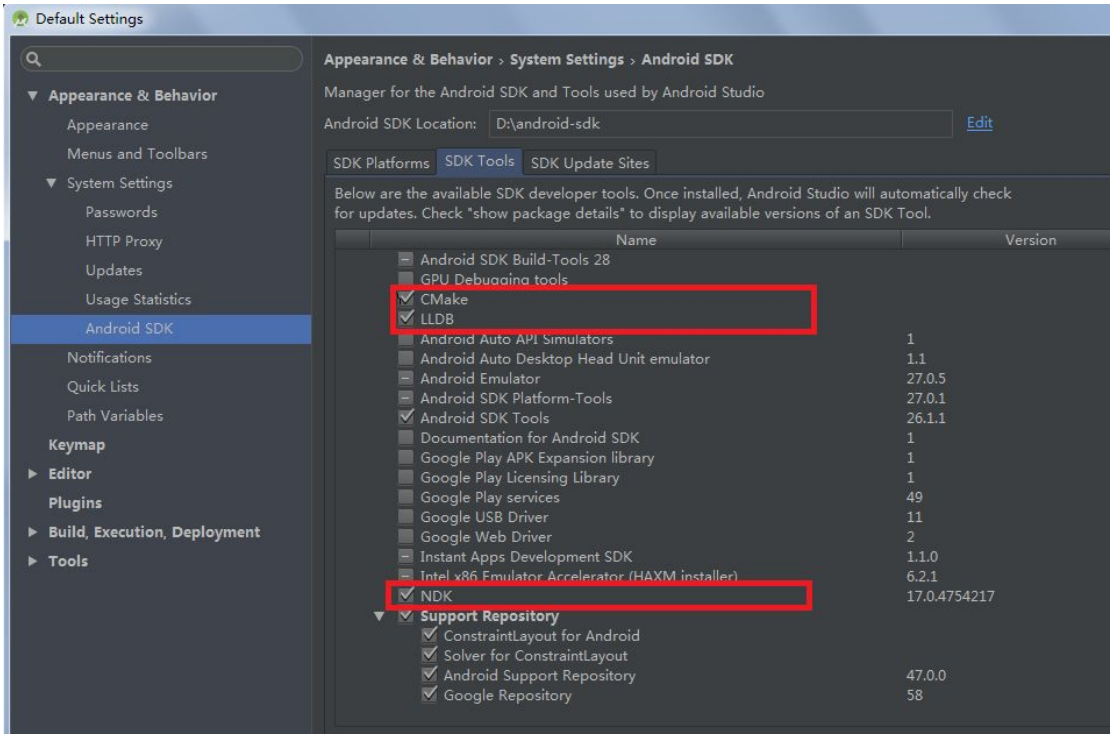
一. JNI 概述

JNI（Java Native Interface Java 本地调用）是一种技术，该技术可以实现在 Java 程序中的函数可以调用 Native 语言写的函数（如 C/C++），Native 层中的函数可以调用 Java 层的函数，也就是在 C/C++程序中调用 Java 的函数。

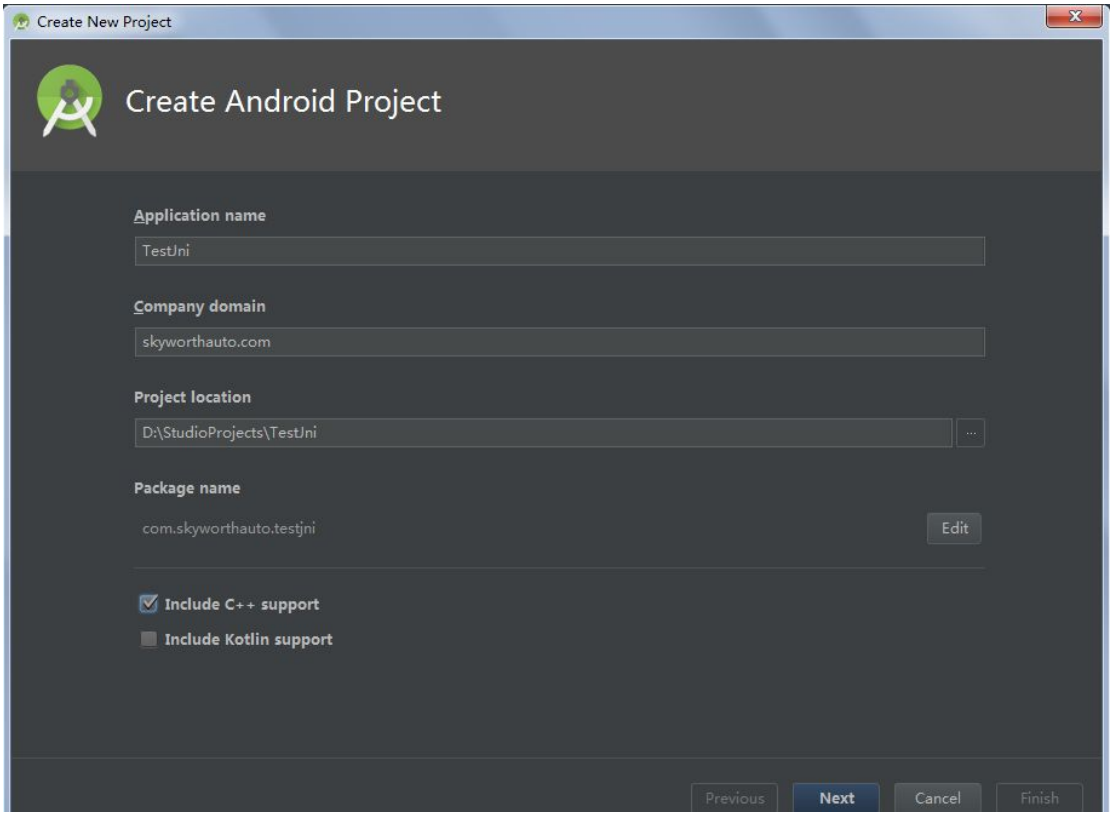
JNI 的主要用途是：1.应用于对执行速度要求比较高的场合　2.为了重用现有的 C/C++代码。主要缺点有使用起来复杂、兼容性差、调试麻烦。不是必要的情况下，最好不使用。

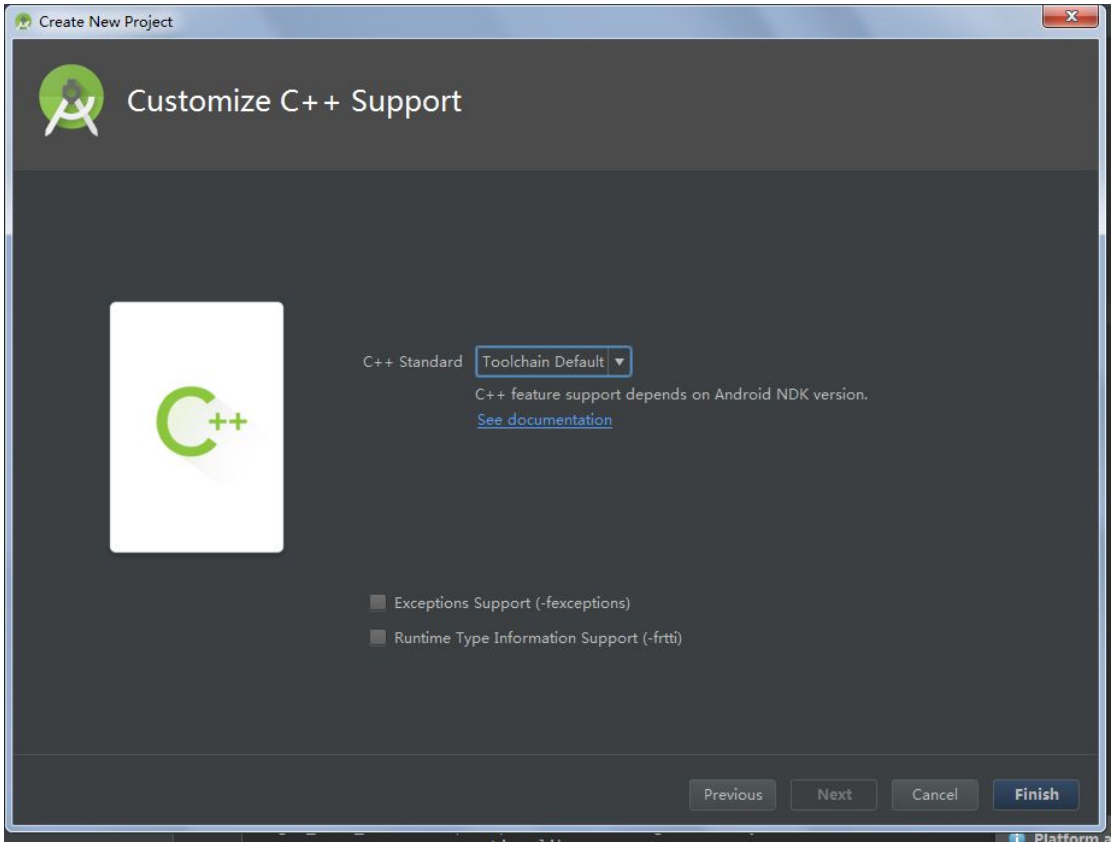
二. JNI 项目配置流程

1. 先安装 SDK Tools 中的 CMake，LLDB，NDK 等开发包。



2. 新建 AndroidStudio 项目。新建命名为 TestJni 项目，在“Create Android Project”向导界面勾选 Include C++ support，在“Customize C++ support”向导界面，C++ Standard 选择默认选项 Toolchain Default，按 FInish 完成新建项目。





3. 新建 Java 类 TestJni，其中 TestJni 类是需要实现 JNI 的类。

【TestJni\app\src\main\java\com\skyworthauto\testjni\TestJni.java】

```
package com.skyworthauto.testjni;
public class TestJni {
    public static native int funStatic(int a, int b);
    public native int fun(int a, int b);
}
```

4. 根据 java 类，用 javah 命令生成 TestJni 类的 TestJni.h 头文件。

```
javah com.skyworthauto.testjni.TestJni
```

5. 在 app/src/main/cpp/native-lib.cpp 中实现 TestJni.h 中的函数

【TestJni2\app\src\main\cpp\native-lib.cpp】

```
JNIEXPORT jint JNICALL
Java_com_skyworthauto_testjni_TestJni_funStatic
        (JNIEnv *env, jclass clazz, jint a, jint b){
    return a + b;
```

```
}

JNIEXPORT jint JNICALL Java_com_skyworthauto_testjni_TestJni_fun
        (JNIEnv *env, jobject thiz, jint a, jint b){
    return a + b;
}
```

6. 在 MainActivity 中调用 System.loadLibrary("native-lib");加载库文件，然后调用 TestJni 相关 native 方法

【TestJni\app\src\main\java\com\skyworthauto\testjni\MainActivity.java】

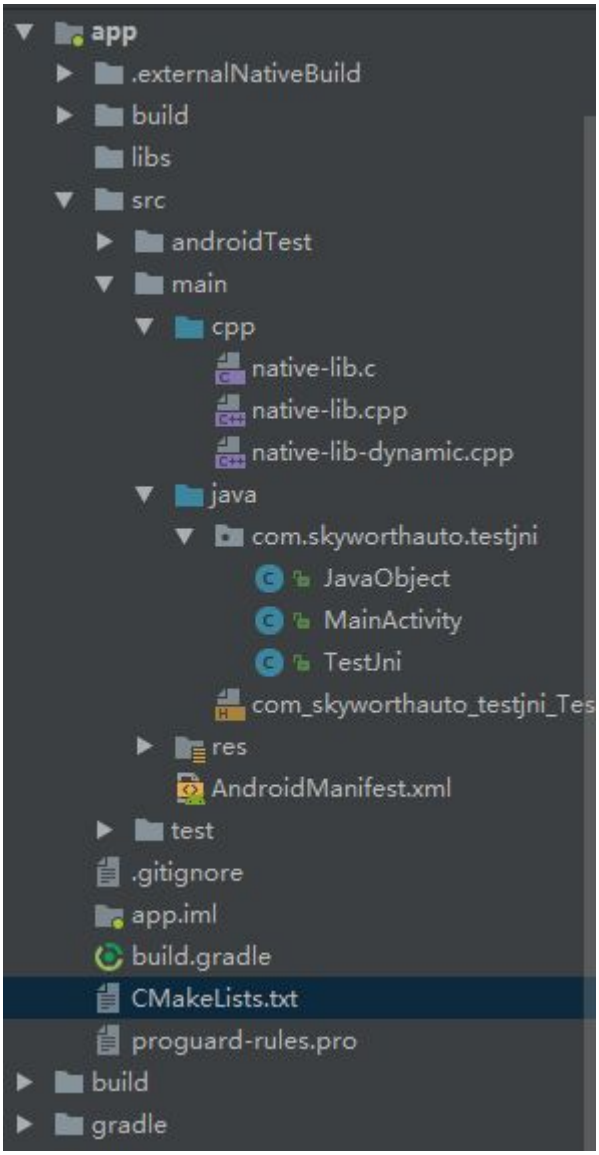
```
package com.skyworthauto.testjni;

public class MainActivity extends AppCompatActivity {

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

    ...
    TestJni mJni = new TestJni();
    mJni.fun(1, 2);
    //...
    TestJni.funStatic(2, 2);
    ...
}
```

7. 按需要对 app 根目录下的 CMakeLists.txt 文件进行修改和定制

【TestJni\app\CMakeLists.txt】

```
add_library( # Sets the name of the library.
             native-lib
             # Sets the library as a shared library.
             SHARED
             # Provides a relative path to your source file(s).
```

```
                src/main/cpp/native-lib-dynamic.cpp )
```



8. 编译运行

三. JNI 相关概念

1. Java 和 JNI 的数据类型转换

1）基本数据类型

Java 类型	JNI 类型	C/C++类型

boolean	jboolean	uint8_t
byte	jbyte	int8_t
char	jchar	uint16_t
short	jshort	int16_t
int	jint	int32_t
long	jlong	int64_t
float	jfloat	float
double	jdouble	double

2）对象数据类型

Java 类型	JNI 类型
java.lang.Object	jobject
java.lang.Class	jclass
java.lang.String	jstring
java.lang.Throwable	jthrowable
Object[]	objectArray
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray

double[]	jdoubleArray

2. JNI 类型表示符

Java 类型	标识符
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
类	Lpath/to/class;
基本类型数组	[type
对象数组	[Lpath/to/class;
void	V

3. Java 的方法签名

由于 Java 支持方法重载，所以需要使用返回类型和参数类型的类型标识符的组合来唯一定位各个方法

举例：

String fun2(int num);　　方法签名为 (I)Ljava/lang/String;

int[] fun2(String[] arr); 方法签名为 ([Ljava/lang/String)[I

4. JNIEXPORT 和 JNICALL

```
#define JNIIMPORT
```

```
#define JNIEXPORT  __attribute__ ((visibility ("default")))
#define JNICALL
```

__attribute__ 可以指定编译时的相关信息

5. FieldID

从 native 层访问或设置一个 Java 对象的属性时，需要先获取 FieldID

```
struct _jfieldID;                    /* 不透明数据类型 */
typedef struct _jfieldID* jfieldID;
```

6. MethodID

从 native 层调用一个 Java 对象的方法时，需要先获取其 MethodID

```
struct _jmethodID;                   /* 不透明数据类型 */
typedef struct _jmethodID* jmethodID;
```

7. JNINativeMethod

用于动态绑定时，描述一个 native 层函数和 Java 层方法的对应关系的结构体，包含 java 中的方法名，java 中方法签名，和 native 层对应的函数指针

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

8. JNIEnv

与线程相关的代表 JNI 环境的结构体，包括了大部分 JNI 的方法

在 C 语言中的定义：

```
struct JNIInvokeInterface{
 ......
}
typedef const struct JNINativeInterface* JNIEnv;
```

在 C++语言中的定义：

```
struct _JNIEnv {
    const struct JNINativeInterface* functions;
     ......
}
typedef _JNIEnv JNIEnv;
```

9. JavaVM

是 Java 虚拟机在 native 层的代表

在 C 语言中的定义：

```
struct JNINativeInterface{
}
typedef const struct JNIInvokeInterface* JavaVM;
```

在 C++语言中的定义：

```
struct _JavaVM {
    const struct JNIInvokeInterface* functions;
    ......
}
typedef _JavaVM JavaVM;
```

10. JavaVMOption

```
typedef struct JavaVMOption {
    const char* optionString;
    void*       extraInfo;
} JavaVMOption;
```

11. JavaVMInitArgs

```
typedef struct JavaVMInitArgs {
    jint        version;    /* use JNI_VERSION_1_2 or later */
```

```
    jint        nOptions;
    JavaVMOption* options;
    jboolean    ignoreUnrecognized;
} JavaVMInitArgs;
```

12. JavaVMAttachArgs

```
typedef struct JavaVMAttachArgs {
    jint        version;    /* must be >= JNI_VERSION_1_2 */
    const char* name;       /* NULL or name of thread as modified UTF-8
str */
    jobject     group;      /* global ref of a ThreadGroup object, or NULL
*/
}JavaVMAttachArgs;
```

四. JNI 深入了解

1. 静态绑定和动态绑定

1）静态绑定

静态绑定是通过”Java_包名_类名_函数名”的规则来定位 JNI 函数，当 Java 调用 native 方法时，会去 so 库里搜索。通过 Java 类生成.h 头文件时，会自动将 Java 的方法名转换为 JNI 方法名，规则如下：将包名、类名、方法名中的“.”替换成“_”，如果类名或方法名中，已有下划线，则在下划线后增加数字 1。如：

```
package com.skyworthauto.testjni;
public class My_class {
   public native int my_fun();
}
```

转换为 JNI 的函数名如下：

```
Java_com_skyworthauto_testjni_My_1class_my_1fun (){}
```

2）动态绑定

动态注册是 JNI 层主动告诉 Java 层函数对应关系，让 Java 层不必去库里搜索 。Java 中调用 System.loadLibrary("native-lib"); 时，加载完成后会在加载的库中查找 jint JNI_OnLoad(JavaVM* vm, void* reserved) 函数，如果找到，即调用该函数，在该函数可以实现动态绑定。

首先定义一个 JNINativeMethod 结构体数组，里面包含了，所有的 Java 方法和 JNI 函数的对应关系。

```
JNINativeMethod method[]={
        {"funStatic","(II)I",(void*)native_funStatic},
        {"fun","(II)I",(int*)native_fun}
}
```

然后定义 JNI_OnLoad（） 函数，在函数内调用 env->RegisterNatives(cl,method,sizeof(method)/sizeof(method[0])) 即可完成动态绑定。

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return -1;
    }
    if(registerNativeMethod(env)!=JNI_OK){
        return -1;
    }
    return JNI_VERSION_1_4;
}

jint registerNativeMethod(JNIEnv *env){
    jclass cl=env->FindClass("com/skyworthauto/testjni/TestJni");

if((env->RegisterNatives(cl,method,sizeof(method)/sizeof(method[0])))
<0){
        return -1;
    }
    return 0;
}
```

2. Java 代码调用 C/C++代码

直接调用 Java 代码中 native 方法，即可调用相应的 C/C++ native 方法。

1)调用普通方法

```
JniTest mJni = new JniTest();
mJni.fun();
```

2)调用静态方法

```
TestJni.fun();
```

3. C/C++代码调用 Java 代码

访问或者调用 Java 对象中的属性和方法，需要先根据 jclass 获取其 属性 ID 和方法 ID。

1)访问 Java 对象的属性

```
    jfieldID fId = (env)->GetFieldID(targetClass, "mNum", "I");
    jint result = (env)->GetIntField( newObject, fId);
```

2)访问 Java 对象的 static 属性

```
    jfieldID fId = (env)->GetStaticFieldID(targetClass, "sNum", "I");
    jint result = (env)->GetStaticIntField(targetClass, fId);
```

3)设置 Java 对象的属性

```
    jfieldID fId = (env)->GetFieldID(targetClass, "mNum", "I");
    (env)->SetIntField(newObject, fId, 30);
```

4)设置 Java 对象的 static 属性

```
    jfieldID fIdStatic = (env)->GetStaticFieldID(targetClass, "sNum",
"I");
    (env)->SetStaticIntField(targetClass, fIdStatic, 3000);
```

5)调用 Java 对象的方法

```
    jmethodID mId = (env)->GetMethodID(targetClass,"method", "()I");
    jint result = (env)->CallIntMethod(newObject, mId);
```

6)调用 Java 对象的 static 方法

```
    jmethodID mId =
(env)->GetStaticMethodID(targetClass,"methodStatic", "()I");
    jint result = (env)->CallStaticIntMethod(targetClass, mId);
```

4. JNI 中的函数重载

假如 Java 对象中有下两个重载方法：

```
public native int fun(int a, int b);
public native float fun(float a, float b);
```

1) 静态绑定时，在 native 层，2 个函数名自动转换成

```
Java_com_skyworthauto_testjni_TestJni_fun__II
Java_com_skyworthauto_testjni_TestJni_fun__FF
```

2)动态绑定时，可以分别指定 2 个函数名来和 Java 中的 2 个重载方法绑定，名称可以不一致

5. JNI 的全局引用

JNI 中的引用默认是局部引用，函数退出，指向的 Java 对象会被回收。使用全局引用可以使 Java 不会被回收

```
jclass globalTargetClass = 0;

//创建全局引用
jint native_fun1(JNIEnv *env, jobject thiz){
jclass targetClass =
env->FindClass("com/skyworthauto/testjni/JavaObject");
if(globalTargetClass == 0){
globalTargetClass = (jclass)env->NewGlobalRef(targetClass);
}
}
//销毁全局引用
void native_fun2(JNIEnv *env, jobject thiz){
env->DeleteGlobalRef(targetClass);
}
```

6.JNI 中 C 和 C++的差异

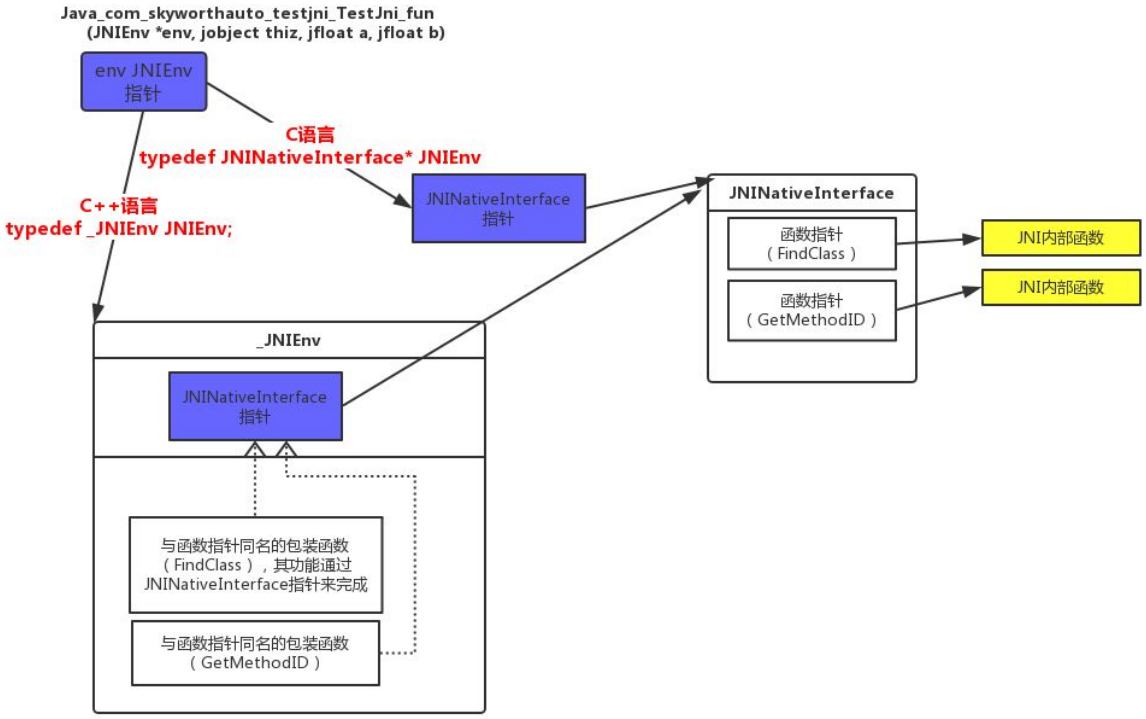
C 中函数调用方式

```
jclass targetClass = (*env)->FindClass(env,
"com/skyworthauto/testjni/JavaObject");
```

C++中函数调用方式：

```
jclass targetClass =
(env)->FindClass("com/skyworthauto/testjni/JavaObject");
```

这种差别是因为在 jni.h 中的 JNIEnv 在 C 和 C++中定义的不一致。（见 JNI 相关概念中的 **8.JNIEnv**）



7. JNI 在 Android 中的应用

1）SystemServier 的应用

【..\frameworks\base\services\java\com\android\server\SystemServer.java 】

```
    private static native void nativeInit();
```

【 ..\frameworks\base\services\jni\com_android_server_SystemServer.cpp 】

```
static void android_server_SystemServer_nativeInit(JNIEnv* env,
jobject clazz) {
    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsensorservice", propBuf, "1");
```

```
    if (strcmp(propBuf, "1") == 0) {
        // Start the sensor service
        SensorService::instantiate();
    }
}

static JNINativeMethod gMethods[] = {
    /* name, signature, funcPtr */
    { "nativeInit", "()V", (void*)
android_server_SystemServer_nativeInit },
};

int register_android_server_SystemServer(JNIEnv* env)
{
    return jniRegisterNativeMethods(env,
"com/android/server/SystemServer",
            gMethods, NELEM(gMethods));
}
```

2）Zygote 的应用

Zygote 的启动过程中，通过 AppRuntime 调用 ZygoteInit 类的 main 方法，进入 Java 层。

【frameworks/base/cmds/app_process/app_main.cpp】

```
class AppRuntime : public AndroidRuntime
{
    ...
}

int main(int argc, char* const argv[])
{
    AppRuntime runtime;
```

```
    ...

    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit", args,
zygote);//1
    } else if (className) {
        runtime.start("com.android.internal.os.RuntimeInit", args,
zygote);
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote
supplied.");
        return 10;
    }
}
```

【frameworks/base/core/jni/AndroidRuntime.cpp】

```
void AndroidRuntime::start(const char* className, const char* options)
{

    /* start the virtual machine */
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);

    /*
```

```
     * Register android functions.
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }


    jclass stringClass;
    jobjectArray strArray;
    jstring classNameStr;
    jstring optionsStr;

    stringClass = env->FindClass("java/lang/String");
    assert(stringClass != NULL);
    strArray = env->NewObjectArray(2, stringClass, NULL);
    assert(strArray != NULL);
    classNameStr = env->NewStringUTF(className);
    assert(classNameStr != NULL);
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    optionsStr = env->NewStringUTF(options);
    env->SetObjectArrayElement(strArray, 1, optionsStr);


    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {

    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass,
"main",
            "([Ljava/lang/String;)V");
```

```
        if (startMeth == NULL) {

        } else {
            env->CallStaticVoidMethod(startClass, startMeth,
strArray);
        }

        ...
    }
}
```

五. ndk-build 的使用

Ndk-build 工具，可以在 Windows 平台下生成其他平台的运行库，如 armeabi-v7a, arm64-v8a, x86,x86_64 等。除了可以在 AndroidStudio 中直接将 C/C++文件编译成 so 库，还可以先使用 ndk-build 工具单独生成 so 库，再将 so 库直接导入 AndroidStudio 使用。

1.ndk-build 环境变量配置

将 Android SDK 目录下的 ndk-bundle 加入 Path 全局变量

2.在任意位置新建一个目录，如：hello-jni

3.在 hello-jni 目录下新建一个 jni 目录

4.在 jni 目录下放入需要编译的源文件，如：hello-jni.c

5.在 jni 目录下新建 Android.mk 文件，内容如下：

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := hello-jni
LOCAL_SRC_FILES := hello-jni.c

include $(BUILD_SHARED_LIBRARY)
```

6.在 jni 目录下新建 Application.mk 文件，内容如下：

```
APP_ABI := all
```

7.在 cmd 窗口，cd 进入到 hello-jni 目录下，执行 ndk-build 命令，即可编译出 libhello-jni.so

```
G:\hello-jni>ndk-build
[arm64-v8a] Install        : libhello-jni.so => libs/arm64-v8a/libhello-jni.so
[x86_64] Install        : libhello-jni.so => libs/x86_64/libhello-jni.so
[mips64] Install        : libhello-jni.so => libs/mips64/libhello-jni.so
[armeabi-v7a] Install        : libhello-jni.so => libs/armeabi-v7a/libhello-jni.so
[armeabi] Install        : libhello-jni.so => libs/armeabi/libhello-jni.so
[x86] Install        : libhello-jni.so => libs/x86/libhello-jni.so
[mips] Install        : libhello-jni.so => libs/mips/libhello-jni.so
```

8.在 AndroidStudio 的 app/src/main 下建立 jniLibs 目录，放入 so 文件，即可编译