

Android 的 JAVA 虚拟机和 JAVA 环境

Android 的 JAVA 虚拟机和 JAVA 环境

- 第一部分 Dalvik 虚拟机
- 第二部分 Android 的 JAVA 程序环境
- 第三部分 JNI 的使用
- 第四部分系统服务的 JAVA 部分

第一部分 Dalvik 虚拟机

Dalvik 是 Android 程序的虚拟机，它执行（**.dex**）的 Dalvik 可执行文件，该格式文件针对小内存使用做了优化。同时虚拟机是基于寄存器的，所有的类都经由 **JAVA** 编译器编译，然后通过 **SDK** 中的 "dx" 工具转化成 **.dex** 格式由虚拟机执行。

Dalvik 虚拟机依赖于 **linux** 内核的一些功能，比如线程机制和底层内存管理机制。

第一部分 Dalvik 虚拟机

Dalvik 虚拟机的代码路径: [dalvik/](#) , 其中包含了目标机和主机的内容。

vm 目录中的内容是虚拟机的实现, 由本地代码实现 (包含了部分的汇编代码), 其编译的结果为共享库 **libdvm.co**。

libcore 目录是一个提供了对基础 **JAVA** 实现支持的代码目录, 包含了 **C** 语言代码和 **JAVA** 代码, 编译的结果为 **JAVA** 的包 **core.jar**。

第一部分 Dalvik 虚拟机

`nativehelper` 库是一个工具库，用于注册 **JAVA** 本地调用的函数，在其他的代码中需要使用 **JNI** 从本地层次向 **JAVA** 层次提供功能的时候，需要使用这个库。

`nativehelper` 库的代码路径为：

`dalvik/libnativehelper` 。连接静态库 `libjavacore.a` ，生成动态库 `libnativehelper.so` 。

`nativehelper` 个库的头文件的路径为：

[`libnativehelper/include/nativehelper/jni.h`](#) ：基于 **JNI** 标准的头文件

[`libnativehelper/include/nativehelper/JNIHelp.h`](#) ：提供 **JNI** 注册功能的头文件

第二部分 Android 的 JAVA 程序环境

Android 的 API 的层次结构:

- ❑ JAVA 标准 API
- ❑ JAVA 扩展 API (javax 包)
- ❑ 企业和组织提供的 java 类库 (org 包)
- ❑ Android 的各种包

相比标准 JAVA, Android 中的 JAVA API 名称相同的 API 功能, 但这些 API 并不是一个全集。

第二部分 Android 的 JAVA 程序环境

Android 中的 JAVA 的库主要为 android 包及其子包，其中核心的包的目录为：

[frameworks/base/core/java/](#)

其中，各个子目录和文件是按照 JAVA 包的关系来组织的，例如文件：

[android/app/Activity.java](#)

它表示 android.app 包中的类 Activity 。

第二部分 Android 的 JAVA 程序环境

Android 中 JAVA 类的 API 的描述文件包含在 [frameworks/base/api/](#) 目录的 [current.xml](#) 文件。

主要使用的标签:



```
<package> </package>  
<class> </class>  
<interface> </interface>  
<implements> </implements>  
<method> </method>  
<field> </field>
```

当注释中写入 **@hide** 的时候，就表示内容被隐藏了，即这个内容虽然出现在 **JAVA** 的源代码中，但是不被视为属于 **Android** 的系统 **API**。

第二部分 Android 的 JAVA 程序环境

android.app.Activity 类的定义:

```
public class Activity extends ContextThemeWrapper           // 定义 Activity 类
    implements LayoutInflater.Factory,
    Window.Callback, KeyEvent.Callback,
    OnCreateContextMenuListener, ComponentCallbacks {
    public Activity() {
        ++sInstanceCount;
    }
    // ... .. 省略
}
```



第二部分 Android 的 JAVA 程序环境

```
<package name="android.app">
<class name="Activity"
  extends="android.view.ContextThemeWrapper"
  abstract="false"
  static="false"
  final="false"
  deprecated="not deprecated"
  visibility="public">
<implements name="android.content.ComponentCallbacks"></implements>
<implements name="android.view.KeyEvent.Callback"></implements>
<implements name="android.view.LayoutInflater.Factory"></implements>
<implements name="android.view.View.OnCreateContextMenuListener"></implements>
<implements name="android.view.Window.Callback"></implements>
<constructor name="Activity"
  type="android.app.Activity"
  static="false"
  final="false"
  deprecated="not deprecated"
  visibility="public"
>
</constructor>
<!-- 省略内容 -->
</class>
<!-- 省略内容 -->
</package>
```

第三部分 JNI 的使用

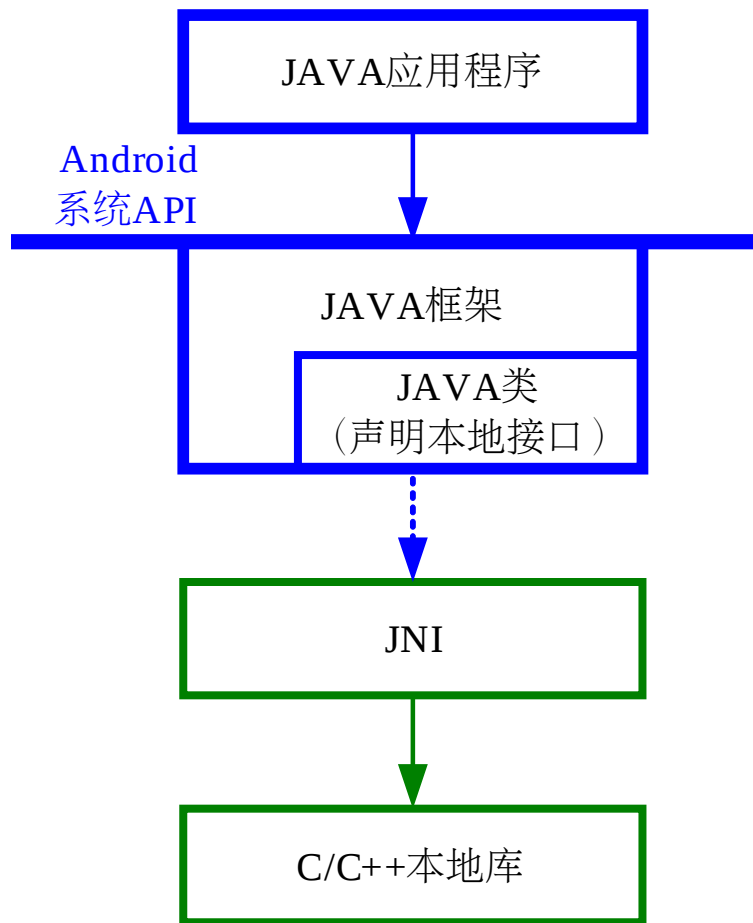
JNI 是 Java Native Interface 的缩写，中文为 JAVA 本地调用。从 Java 1.1 开始，Java Native Interface (JNI) 标准成为 java 平台的一部分，它允许 Java 代码和其他语言写的代码进行交互。JNI 是本地编程接口。它使得在 Java 虚拟机 (VM) 内部运行的 Java 代码能够与用其它编程语言 (如 C、C++ 和汇编语言) 编写的应用程序和库进行互操作。

在 Android 中提供 JNI 的方式，让 JAVA 程序可以调用 C 语言的程序。

第三部分 JNI 的使用

JAVA 的类型	JNI 的类型	对应的字母
JAVA 布尔类型 (boolean)	jboolean (8 位无符号)	Z
JAVA 字节 (byte)	jbyte (8 位有符号)	B
JAVA 字符 (char)	jchar (16 位无符号)	C
JAVA 短整型 (short)	jshort (16 位有符号)	S
JAVA 整型 (int)	jint (32 位有符号)	I
JAVA 长整型 (long)	jlong (64 位有符号)	J
JAVA 单精度浮点 (float)	jfloat (IEEE 754 , 32 位)	F
JAVA 双精度浮 (double)	jdouble (IEEE 754 , 64 位)	D
JAVA 对象	jobject	L
JAVA 的无返回值	void	V

第三部分 Android 中的 JNI



在 Android 中提供 JNI 的方式，让 JAVA 程序可以调用 C 语言的程序。很多 Android 中 JAVA 的类都具有 native 的接口，这些 native 接口就是由本地实现，然后注册到系统中的。

3.1 JNI 的实现方式

在 Android 中，主要的 JNI 的代码在以下的路径中：[frameworks/base/core/jni/](#)

这个路径中的内容将被编译成为库，[libandroid_runtime.so](#)，这就是一个普通的动态库，被放置在目标系统的 [/system/lib](#) 目录中。

除此之外，Android 还包含了其他的几个 JNI 的库，例如媒体部分的 JNI 在目录 [frameworks/base/media/jni/](#) 中，被编译成为库 [libmedia_jni.so](#)。

3.1 JNI 的实现方式

JNI 中各个文件的实际上就是 C++ 的普通源文件，其命名一般和对应支持的 JAVA 类有对应关系。这种关系是习惯上的写法，而不是强制的。

在 Android 中实现的 JNI 库，需要连接动态库 `libnativehelper.so`。

Android 中使用 JNI 主要有两种方式：

1. 在框架层实现，连接 JAVA 框架和本地框架
2. 在应用程序的 Apk 包中实现

3.2 在框架层实现 JNI

android.util.Log 类的情况:

```
public final class Log {  
    public static native boolean isLoggable(String tag, int level);  
    public static native int println(int priority, String tag, String msg);  
}
```

android_util_Log.cpp 中的方法列表:

```
static JNINativeMethod gMethods[] = {  
    { "isLoggable", "(Ljava/lang/String;I)Z",  
      (void*) android_util_Log_isLoggable },  
    { "println", "(ILjava/lang/String;Ljava/lang/String;)I",  
      (void*) android_util_Log_println },  
};
```


3.2 在框架层实现 JNI

注册 JNI 的情况:

[illegible]

3.3 在 Apk 中实现 JNI

JNI 的示例程序的路径:

[development/samples/SimpleJNI](#)

编译成的 JNI 动态库: **libsimplejni.so**

编译成的 JAVA 包: **SimpleJNI.apk**

```
TOP_LOCAL_PATH:= $(call my-dir)
# Build activity
LOCAL_PATH:= $(TOP_LOCAL_PATH)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := samples
LOCAL_SRC_FILES := $(call all-subdir-java-files)
LOCAL_PACKAGE_NAME := SimpleJNI
LOCAL_JNI_SHARED_LIBRARIES := libsimplejni
include $(BUILD_PACKAGE)

include $(call all-makefiles-under,$(LOCAL_PATH))
```

第三部分 JNI 的使用

JNI Code path:

[jni/native.cpp](#)

JNI Method:

```
static jint
add(JNIEnv *env, jobject thiz, jint a, jint b) {
    int result = a + b;
    LOGI("%d + %d = %d", a, b, result);
    return result;
}

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

3.3 在 Apk 中实现 JNI

Register Method:

```
static int registerNativeMethods(JNIEnv* env,
                                const char* className,
                                JNINativeMethod* gMethods, int numMethods)
{
    jclass clazz;
    clazz = env->FindClass(className);
    // .....
    if (env->RegisterNatives(clazz, gMethods, numMethods) < 0) {
        LOGE("RegisterNatives failed for '%s'", className);
        return JNI_FALSE;
    }
    return JNI_TRUE;
}

static int registerNatives(JNIEnv* env)
{
    if (!registerNativeMethods(env, classPathName,
                              methods, sizeof(methods) / sizeof(methods[0]))) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

3.3 在 Apk 中实现 JNI

初始化部分执行的代码：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    UnionJNIEnvToVoid uenv;
    uenv.venv = NULL;
    jint result = -1;
    JNIEnv* env = NULL;

    LOGI("JNI_OnLoad");
    if (vm->GetEnv(&uenv.venv, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed");
        goto bail;
    }
    env = uenv.env;
    if (registerNatives(env) != JNI_TRUE) {
        LOGE("ERROR: registerNatives failed");
        goto bail;
    }
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

3.3 在 Apk 中实现 JNI

JAVA Application Code path:

<src/com/example/android/simplejni/simplejni.java>

```
public class SimpleJNI extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        int sum = Native.add(2, 3);
        tv.setText("2 + 3 = " + Integer.toString(sum));
        setContentView(tv);
    }
}

class Native {
    static {
        System.loadLibrary("simplejni");
    }
    static native int add(int a, int b);
}
```

3.4 JNI 中的高级使用

JNI 的基本功能是让 **JAVA** 调用本地代码。除此之外，**JNI** 还可以实现更为高级的内容。

- 本地代码访问 **JAVA** 类中的属性
- 本地代码反向调用 **JAVA** 的方法

3.4 JNI 中的高级使用

JNI 代码如下所示:

```
struct fields_t {
    jfieldID  context;
    jmethodID post_event;
};
static fields_t fields;
/* 获取和调用 */
{
    fields.context = env->GetFieldID(clazz, "mNativeContext", "I");
    env->SetIntField(thiz, fields.context, (int)context);
    fields.post_event = env->GetStaticMethodID(clazz,
                                                "postEventFromNative",
                                                "(Ljava/lang/Object;IIILjava/lang/Object;)V");
}
```


3.4 JNI 中的高级使用

反向调用的过程。

```
static jint
test (JNIEnv *env, jobject thiz, jobject weak_this) {

    JNIEnv *env = AndroidRuntime::getJNIEnv();
    jclass clazz = env->GetObjectClass(thiz);
    jobject object = env->NewGlobalRef(weak_this);
    env->CallStaticVoidMethod(clazz, fields.post_event,
                             object, msgType, ext1, ext2, NULL);

    /* ..... */
}
```

3.4 JNI 中的高级使用

与之对应的 **JAVA** 代码:

```
public class TestClass {
    @SuppressWarnings("unused")
    private int mNativeContext;
    private EventHandler mEventHandler;
    TestClass() {
        /* ..... */
        test(new WeakReference<TestClass>(this));
    }
    private class EventHandler extends Handler
    {
        private TestClass mTestClass;
        public EventHandler(TestClass testclass, Looper looper) {
            super(looper);
            mTestClass = testclass;
        }
    }
}
```

3.4 JNI 中的高级使用

```
@Override
public void handleMessage(Message msg) {
    msg.what;
    msg.arg1;
    msg.arg2;
    /* ..... */
}
}
private static void postEventFromNative(Object TestClassref,
                                         int what, int arg1, int arg2, Object obj)
{
    TestClass TestClass = (TestClass)((WeakReference)TestClassref).get();
    if (TestClass == null)
        return;
    if (TestClass.mEventHandler != null) {
        Message m = TestClass.mEventHandler.obtainMessage(what, arg1, arg2, obj);
        TestClass.mEventHandler.sendMessage(m);
    }
}
}
```

第四部分 系统服务的 JAVA 部分

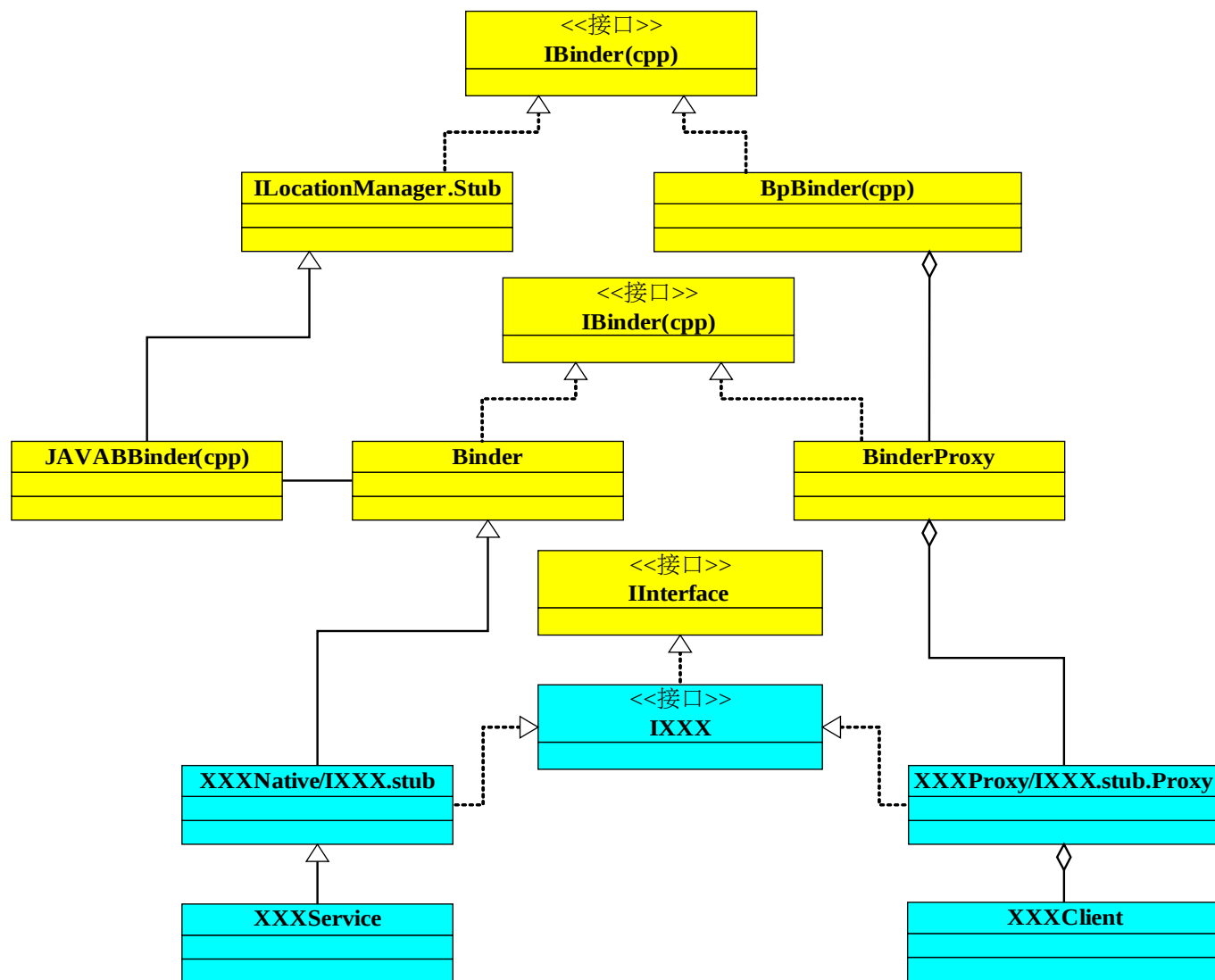
JAVA 层同样提供了一套 Binder 的相关函数，让 JAVA 代码可以直接进行 Binder 操作。实现在：

[frameworks/base/core/java/android/os/](#)

[frameworks/base/core/java/com/android/internal/os/](#)

[frameworks/base/core/jni/](#)

第四部分 系统服务的 JAVA 部分



第四部分 系统服务的 JAVA 部分

zygote 是通过 init 进程读取 init.rc 启动：

```
service zygote /system/bin/app_process -Xzygote  
/system/bin --zygote --start-system-server  
    socket zygote stream 666  
    onrestart write /sys/android_power/request_state wake  
    onrestart write /sys/power/state on
```

谢谢！