Android Kernel (1) - Basic

AUGUST 11, 2014

android (/tags/android.html)

1. Android Architecture Introduction

Architecture

1. **Application Layer**: applications, using Java, and NDK
2. **Application Framework layer**: defines the policy of Android app. Java & JNI
3. **Libraries & Runtimes**: C++/C
4. **Linux Kernel**:

There is a layer, **hardware abstraction layer**. between linux kernel and libraries & runtimes. In HAL, only binary code is required. Hardware provider does not need to provide source code. HAL de-couples Android and Linux kernel, make it easier to

migrate system and do interface development.

From Dynamic perspective

Android consists of**User space** and **Kernel space**:

- User space
  - **Native Code** (NDK code) and **Dalvik runtime** (JDK code).
- Kernel space: Linux Kernel & Android extensions
  - Android extensions: Binder, Logger, OOM ...

Compile module in Android

Compile application layer app

For application layer app, check the `Android.mk` file, there is a variable inside:

```
LOCAL_PACKAGE_NAME := Phone
```

With `LOCAL_PACKAGE_NAME`, we can build a module seperately:

```
$ make Phone
```

Compile framework layer & runtime libraries source code

We need to know `LOCAL_MODULE` variable.

> **Example:** ***`app_process`*** **module**

`frameworks/base/cmds/app_process/Android.mk`:

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

LOCAL_SRC_FILES:= \
    app_main.cpp

LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    liblog \
    libbinder \
    libandroid_runtime

LOCAL_MODULE:= app_process

include $(BUILD_EXECUTABLE)


# Build a variant of app_process binary linked with ASan runtime.
# ARM-only at the moment.
ifeq ($(TARGET_ARCH),arm)

include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
    app_main.cpp

LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    liblog \
    libbinder \
    libandroid_runtime

LOCAL_MODULE := app_process__asan
LOCAL_MODULE_TAGS := eng
LOCAL_MODULE_PATH := $(TARGET_OUT_EXECUTABLES)/asan
LOCAL_MODULE_STEM := app_process
LOCAL_ADDRESS_SANITIZER := true

include $(BUILD_EXECUTABLE)

endif # ifeq($(TARGET_ARCH),arm)
```

To build `app_process`:

```
$ make app_process
```

`mmm` command

`mmm` specifies the path of module to be compiled:

```
$ mmm packages/apps/phone
```

`mm` command

`mm` compiles current module:

```
/Volumes/aosp/os/packages/apps/Phone  ➦ 1943cde  mm
```

2. JNI

JNI is between **Application layer** and **Framework layer**. Part of JNI source code is inside `frameworks/base`. Different mobules will be compiled into shared libraries, and put into `/system/lib` folder.

> **Difference between NDK and JNI is: NDK is a toolkit for using JNI. JNI is a programming interface, on top of application layer or framework layer, for Java code to call native code.**

To use JNI, there are 3 steps:

1. Declare `native` method in Java code
2. Implement native method in JNI layer. JNI layer wiil be compiled to shared library
3. Load shared library

Example of JNI programming

Let's look into the source code of `Log.d(TAG, "debug log")`:

1. `native` method in Java code

In `frameworks/base/core/java/android/util/Log.java`, part of the code about `Log`:

```
package android.util;

public final class Log {

    /**
     * Send a {@link #DEBUG} log message.
     * @param tag Used to identify the source of a log message.  It usually identifies
     *        the class or activity where the log call occurs.
     * @param msg The message you would like logged.
     */
    public static int d(String tag, String msg) {
        return println_native(LOG_ID_MAIN, DEBUG, tag, msg);
    }

    /**
     * Checks to see whether or not a log for the specified tag is loggable at the specifi
ed level.
     *
     *  The default level of any tag is set to INFO. This means that any level above and i
ncluding
     *  INFO will be logged. Before you make any calls to a logging method you should chec
k to see
     *  if your tag should be logged. You can change the default level by setting a system
property:
     *      'setprop log.tag.&lt;YOUR_LOG_TAG> &lt;LEVEL>'
     *  Where level is either VERBOSE, DEBUG, INFO, WARN, ERROR, ASSERT, or SUPPRESS. SUPP
RESS will
     *  turn off all logging for your tag. You can also create a local.prop file that with
the
     *  following in it:
     *      'log.tag.&lt;YOUR_LOG_TAG>=&lt;LEVEL>'
     *  and place that in /data/local.prop.
     *
     * @param tag The tag to check.
     * @param level The level to check.
     * @return Whether or not that this is allowed to be logged.
     * @throws IllegalArgumentException is thrown if the tag.length() > 23.
     */
    public static native boolean isLoggable(String tag, int level);

    /**
     * Low-level logging call.
     * @param priority The priority/type of this log message
     * @param tag Used to identify the source of a log message.  It usually identifies
     *        the class or activity where the log call occurs.
     * @param msg The message you would like logged.
     * @return The number of bytes written.
     */
    public static int println(int priority, String tag, String msg) {
        return println_native(LOG_ID_MAIN, priority, tag, msg);
    }

    /** @hide */ public static final int LOG_ID_MAIN = 0;
    /** @hide */ public static final int LOG_ID_RADIO = 1;
```

```
/** @hide */ public static final int LOG_ID_EVENTS = 2;
/** @hide */ public static final int LOG_ID_SYSTEM = 3;

/** @hide */ public static native int println_native(int bufID,
        int priority, String tag, String msg);
```

In java, we only need to define `native` method and not implement them.

2. JNI layer

JNI file name is based on the package name of Java code. Here it should be `android_util_Log`.

`framework/base/core/jni/android_util_Log.cpp`:

```
#include "jni.h"
#include "JNIHelp.h"
#include "utils/misc.h"
#include "android_runtime/AndroidRuntime.h"
#include "android_util_Log.h"

static jboolean android_util_Log_isLoggable(JNIEnv* env, jobject clazz, jstring tag, jint
level)
{
    if (tag == NULL) {
        return false;
    }

    const char* chars = env->GetStringUTFChars(tag, NULL);
    if (!chars) {
        return false;
    }

    jboolean result = false;
    if ((strlen(chars)+sizeof(LOG_NAMESPACE)) > PROPERTY_KEY_MAX) {
        char buf2[200];
        snprintf(buf2, sizeof(buf2), "Log tag \"%s\" exceeds limit of %d characters\n",
                chars, PROPERTY_KEY_MAX - sizeof(LOG_NAMESPACE));

        jniThrowException(env, "java/lang/IllegalArgumentException", buf2);
    } else {
        result = isLoggable(chars, level);
    }

    env->ReleaseStringUTFChars(tag, chars);
    return result;
}

/*
 * In class android.util.Log:
 *  public static native int println_native(int buffer, int priority, String tag, String m
sg)
 */
static jint android_util_Log_println_native(JNIEnv* env, jobject clazz,
        jint bufID, jint priority, jstring tagObj, jstring msgObj)
{
    const char* tag = NULL;
    const char* msg = NULL;

    if (msgObj == NULL) {
        jniThrowNullPointerException(env, "println needs a message");
        return -1;
    }

    if (bufID < 0 || bufID >= LOG_ID_MAX) {
        jniThrowNullPointerException(env, "bad bufID");
        return -1;
    }
```

```
    if (tagObj != NULL)
        tag = env->GetStringUTFChars(tagObj, NULL);
    msg = env->GetStringUTFChars(msgObj, NULL);

    int res = __android_log_buf_write(bufID, (android_LogPriority)priority, tag, msg);

    if (tag != NULL)
        env->ReleaseStringUTFChars(tagObj, tag);
    env->ReleaseStringUTFChars(msgObj, msg);

    return res;
}
```

`__android_log_buf_write` is the method writes the log message. JNI layer does a method mapping based on a naming convention. eg. `isLoggable` (in Java) -> `android_util_Log_isLoggable` (in JNI).

Compare the method arguments and return types:

Method name	arguments	return type
`isLoggable`	`(String tag, int level)`	`boolean`
`android_util_Log_isLoggable`	`(JNIEnv* env, jobject clazz, jstring tag, jint level)`	`jboolean`

3. Register JNI method

Now we need to connect `isLoggable` and `android_util_Log_isLoggable` togother. In `android_util_Log.cpp`:

```
/*
 * JNI registration.
 */
static JNINativeMethod gMethods[] = {
    /* name, signature, funcPtr */
    { "isLoggable",      "(Ljava/lang/String;I)Z", (void*) android_util_Log_isLoggable },
    { "println_native",  "(IILjava/lang/String;Ljava/lang/String;)I", (void*) android_uti
l_Log_println_native },
};
```

`JNINativeMethod` is defined in `development/ndk/platforms/android-3/include/jni.h`:

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

which means:

- Java layer method name is `isLoggable`
- Java layer method signature, based on JNI naming convention, is `(Ljava/lang/String;I)Z`
- JNI layer function pointer is `(void*) android_util_Log_isLoggable`

To tell VM about this mapping:

```
int register_android_util_Log(JNIEnv* env)
{
    jclass clazz = env->FindClass("android/util/Log");

    if (clazz == NULL) {
        ALOGE("Can't find android/util/Log");
        return -1;
    }

    levels.verbose = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "VERBOSE",
"I"));
    levels.debug = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "DEBUG",
"I"));
    levels.info = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "INFO",
"I"));
    levels.warn = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "WARN",
"I"));
    levels.error = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "ERROR",
"I"));
    levels.assert = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "ASSERT",
"I"));

    return AndroidRuntime::registerNativeMethods(env, "android/util/Log", gMethods, NELE
M(gMethods));
}
```

This method takes `gMethods`, Java class name and a `JNIEnv` pointer to `AndroidRuntime::registerNativeMethods`.

Question 1: `AndroidRuntime::registerNativeMethods`

This is defined in `framework/base/core/jni/AndroidRuntime.cpp`:

```
/*
 * Register native methods using JNI.
 */
/*static*/ int AndroidRuntime::registerNativeMethods(JNIEnv* env,
    const char* className, const JNINativeMethod* gMethods, int numMethods)
{
    return jniRegisterNativeMethods(env, className, gMethods, numMethods);
}
```

It is a wrapper method to `jniRegisterNativeMethods`, defined in `libnativehelper/include/nativehelper/JNIHelp.h`:

```
/*
 * Register one or more native methods with a particular class.
 * "className" looks like "java/lang/String". Aborts on failure.
 * TODO: fix all callers and change the return type to void.
 */
int jniRegisterNativeMethods(C_JNIEnv* env, const char* className, const JNINativeMethod*
gMethods, int numMethods);
```

Here is the implemetation:

```
extern "C" int jniRegisterNativeMethods(C_JNIEnv* env, const char* className,
    const JNINativeMethod* gMethods, int numMethods)
{
    JNIEnv* e = reinterpret_cast<JNIEnv*>(env);

    ALOGV("Registering %s's %d native methods...", className, numMethods);

    scoped_local_ref<jclass> c(env, findClass(env, className));
    if (c.get() == NULL) {
        char* msg;
        asprintf(&msg, "Native registration unable to find class '%s'; aborting...", class
Name);
        e->FatalError(msg);
    }

    if ((*env)->RegisterNatives(e, c.get(), gMethods, numMethods) < 0) {
        char* msg;
        asprintf(&msg, "RegisterNatives failed for '%s'; aborting...", className);
        e->FatalError(msg);
    }

    return 0;
}
```

The JNI registration process calls the `RegisterNatives` method of `JNIEnv` object, passes the `gMethods` to Dalvik VM. Here is the signature of `RegisterNatives`:

```
    jint        (*RegisterNatives)(JNIEnv*, jclass, const JNINativeMethod*,
                        jint);
```

Question 2: what is `JNIEnv`?

`JNIEnv` is a pointer to an array of JNI methods

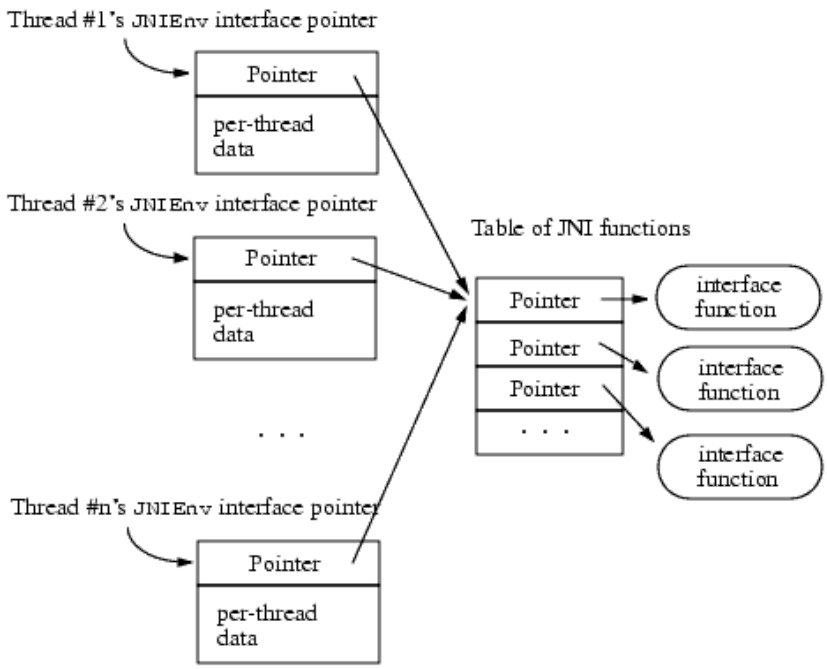
Question 3: where do we call `register_android_util_Log`?

It is called in Android bootstrap process, by `register_jni_procs` in `AndroidRuntime.cpp`

> **Two ways to use JNI: (1) follow JNI naming convention. (2) use function registration.**

JNIEnv

`JNIEnv` is the most important thing in JNI programming. Here is the structure of `JNIEnv`:



Elements within JNIEnv, points to a thread-related struct, the struct points to an array of pointers. Each of them points to a JNI method. In `libnativehelper/include/nativehelper/jni.h`:

```
struct _JNIEnv;
struct _JavaVM;
typedef const struct JNINativeInterface* C_JNIEnv;

#if defined(__cplusplus)
typedef _JNIEnv JNIEnv;
typedef _JavaVM JavaVM;
#else
typedef const struct JNINativeInterface* JNIEnv;
typedef const struct JNIInvokeInterface* JavaVM;
#endif

// ...

/*
 * C++ object wrapper.
 *
 * This is usually overlaid on a C struct whose first element is a
 * JNINativeInterface*.  We rely somewhat on compiler behavior.
 */
struct _JNIEnv {
    /* do not rename this; it does not seem to be entirely opaque */
    const struct JNINativeInterface* functions;

#if defined(__cplusplus)
    jclass FindClass(const char* name)
    { return functions->FindClass(this, name); }

    jint ThrowNew(jclass clazz, const char* message)
    { return functions->ThrowNew(this, clazz, message); }
```

We can see, `_JNIEnv` wraps `JNINativeInterface*`:

```
/*
 * Table of interface function pointers.
 */
struct JNINativeInterface {
    jclass      (*FindClass)(JNIEnv*, const char*);
    jint        (*ThrowNew)(JNIEnv *, jclass, const char *);

    // ...
}
```

Summary

In C++: `JNIEnv` is `struct _JNIEnv`. When calling, `env->FindClass(JNIEnv*, const char*)` calls the corresponding function pointer in `JNINativeInterface`.

In C: `JNIEnv` is `const struct JNINativeInterface*)`. `JNIEnv* env` is `const strcut JNINativeInterface ** env`. Call it using `(*env)->FindClass(JNIEnv*, const char*)`.

Calling JNI method in Java

We will discuss about JNI implementation, JNI data type convertion, JNI naming convention and JNI signature convention.

Data type

most of the Java primitive type, prefix with `j`, is the corresponding JNI type.

Java type	JNI type	length
boolean	jboolean	8 bits
byte	jbyte	8 bits
char	jchar	16 bits
...	...	....
void	void	

There is another macro

```
#define JNI_FALSE 0
#define JNI_TRUE 1
```

For reference type

Java type	JNI type
Class	jclass
String	jstring
Throwable	jthrowable
Object[], boolean[], byte[] ...	jobjectArray, jbooleanArray, jbyteArray ...
Object	jobject

Naming convention

eg. `isLoggable` -> `android_util_Log_isLoggaable`

Some methods do not follow naming convention, because they are using registration method.

Signature convention

By using method name cannot identify a method in Java, due to method overriding. JNI provides a set of signature rules, using a string to identify methods.

Java type	Signature
boolean	Z
byte	B
char	C
long	J
float	F
double	D
short	S
int	I
Class	L
array	[

Signature rule is: `(sig1 sig2 sig3 ...)sig_return`. No space in between.

> **Signature of class is** `L + full_class_name + ;` **. eg** `String` **->** `Ljava/lang/String;`

eg. `long fun(int n, String str, int[] arr);` -> `(ILjava/lang/String;[I)J)`

Calling Java in JNI method

JNI provides the way for Java and C/C++ to operate each other. We have seen how Java operate C/C++.

The second argument `jobject` can be operated by C/C++.

Access Java object

Most frequently used methods are `FindClass` and `GetObjectClass`.

In C++:

```
jclass FindClass(const char* name);
jclass GetObjectClass(jobject obj);
```

In C:

```
c jclass (*FindClass)(JNIEnv*, const char*); jclass (*GetObjectClass)(JNIEnv*, jobject);
```

In `android_util_Log.cpp`:

```
int register_android_util_Log(JNIEnv* env)
{
    jclass clazz = env->FindClass("android/util/Log");

    levels.verbose = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "VERBOSE",
"I"));
    levels.debug = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "DEBUG",
"I"));
    levels.info = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "INFO",
"I"));
    levels.warn = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "WARN",
"I"));
    levels.error = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "ERROR",
"I"));
    levels.assert = env->GetStaticIntField(clazz, env->GetStaticFieldID(clazz, "ASSERT",
"I"));
}
```

This is how C code access Java class. In `GetStaticFieldID`, second argument is the field name, third argument is the type signature.

```
jfieldId GetStaticFieldID(jclass clazz, const char* name, const char* sgi)
```

`FindClass -> GetMethodId` returns `(jmethodID) -> Call<Type>Method`

Global reference

JVM use reference counting strategy to do GC. If Java object uses Native method, how does the GC work?

1. Add global variable to a class, and assign value in a member method
2. Add static variable in a member method, and assign value

In these two situations, JVM cannot track reference counting. Some pointers may become **wild pointer** in C/C++, leads to memory leak.

JNIEnv provides a solution

1. Local Reference

Reference counting works, within current thread. It works like local variable

2. Global Reference

Reference counting works, within multiple thread. It requires explicit releasing. Use `NewGlobalRef` to create, and `DeleteGlobalRef` to release.

3. Weak Global Reference

Reference counting does NOT work. It's scope is multiple thread. It also requires explicit releasing. It's object life time depends on VM, which means, even though you haven't release weak global reference, it's referencing object may already been released. Use `NewWeakGlobalRef` to create, and `DeleteWeakGlobalRef` to release. It's advantage is, you can save object without blocking other thing to release it.

```
static jobject g_clazz_ref = NULL;
static jboolean android_util_Log_isLoggable(JNIEnv* env, jobject clazz, jstring tag, init
level) {
  // ...
  g_clazz_ref = env->NewGlobalRef(clazz);
  // ...
}
env->DeleteGlobalRef(g_clazz_ref);
```

For global reference, by default it cannot be created more than 200 global references.

JNI Exception Handling

The way JNI checks the exceptions:

1. check whether return value is NULL
2. call JNI method `ExceptionOccured`

Handling exception also got two ways:

1. Native method return immediately, then the exception will handled by Java code.
2. Use `ExceptionClear()` to remove exceptions

For native method, you must clear the exception before calling other JNI methods. Only `ExceptionOccurred()`, `ExceptionDescribe()`, `ExceptionClear()` can be called before exceptions are handled.

```
static jboolean android_util_Log_isLoggable(JNIEnv* env, jobject clazz, jstring tag, jint
level)
{
    if (tag == NULL) {
        return false;
    }

    const char* chars = env->GetStringUTFChars(tag, NULL);
    if (!chars) {
        return false;
    }

    jboolean result = false;
    if ((strlen(chars)+sizeof(LOG_NAMESPACE)) > PROPERTY_KEY_MAX) {
        char buf2[200];
        snprintf(buf2, sizeof(buf2), "Log tag \"%s\" exceeds limit of %d characters\n",
                chars, PROPERTY_KEY_MAX - sizeof(LOG_NAMESPACE));

        jniThrowException(env, "java/lang/IllegalArgumentException", buf2);
    } else {
        result = isLoggable(chars, level);
    }

    env->ReleaseStringUTFChars(tag, chars);
    return result;
}
```

The exception is manually, in `if (tag == NULL)` and `if ((strlen(chars)+sizeof(LOG_NAMESPACE)) > PROPERTY_KEY_MAX)`.

`jniThrowException` throws the exception, defined in `libnativehelper/JNIHelp.cpp`.

```
extern "C" int jniThrowException(C_JNIEnv* env, const char* className, const char* msg) {
    JNIEnv* e = reinterpret_cast<JNIEnv*>(env);

    if ((*env)->ExceptionCheck(e)) {
        /* TODO: consider creating the new exception with this as "cause" */
        scoped_local_ref<jthrowable> exception(env, (*env)->ExceptionOccurred(e));
        (*env)->ExceptionClear(e);

        if (exception.get() != NULL) {
            std::string text;
            getExceptionSummary(env, exception.get(), text);
            ALOGW("Discarding pending exception (%s) to throw %s", text.c_str(), classNam
e);
        }
    }
```

Share this article

Tweet Like 0 G+1 0 submit

0 Comments allenlsy Login

Recommend Share Sort by Best

Start the discussion…

Be the first to comment.

ALSO ON ALLENLSY

#1 - Rails Dev Environment & Vim

2 comments • 2 years ago

allenlsy — 一首随性的曲子

#2 - Getting Started With Rails 4

1 comment • 2 years ago

eclili — Very good!

WHAT'S THIS?

Google+ 团队的 Android UI 测试 — Simple, Not Easy

7 comments • 10 months ago

Li7tleMK — http://ww2.sinaimg.cn/mw690/6b... You can see the image and what's going on. I

Effective Java 4 - Generics — Simple, Not Easy

2 comments • 2 years ago

allenlsy — Nah, just some notes of other good books. Joshua Bloch (author of this book) is the real master.

Subscribe Add Disqus to your site Add Disqus Add Privacy

喜欢

0 最新 最早 最热

还没有评论，沙发等你来抢

嘿嘿参北斗哇 (http://www.baidu.com/p/嘿嘿参北斗哇) 帐号管理

说点什么吧...

(http://duoshuo.com/settings/avatar/)

分享到: 发布

allenlsy正在使用多说 (http://duoshuo.com)

Blog (http://allenlsy.com) · Casts (http://cast.allenlsy.com) ·