In the Dalvik

Dainius Jocas

May 27, 2013

The Dalvik Virtual Machine is the heart of Android. It's a fast, just-in-time compiled, optimized bytecode virtual machine. Android applications are compiled to Dalvik bytecode and run on the Dalvik VM. This section includes detailed information such as the Dalvik bytecode format specification, design information on the VM itself, and so on.

Dalvik is the managed runtime used by applications and some system services on Android. Dalvik was originally created specifically for the Android project.

Today we are going to start to look at the code which is at the very core of Dalvik VM (Virtual Machine). To be more specific, we are going to talk about Android's run-time system, whose core part is the Dalvik VM.

By definition run-time system is a software designed to support the execution of computer programs. Intuitively, we can think of a runtime as a set of libraries that is responsible for low level tasks, e.g. dynamic memory allocation in C. In addition to the basic low-level tasks of the program, a runtime system may also implement higher-level behaviour and even support type checking, debugging, or code generation and optimization. Dalvik as a runtime, besides providing low-level libraries, implements lots of higher-level functionality, e.g. type checking.

To investigate implementation of Dalvik, means to investigate an implementation of the run-time system. And, therefore, to investigate the implementation of the run-time system means to investigate the implementation of a set of tools of the run-time system, e.g. dexopt – a tool which verifies and optimizes all of the classes in the DEX file.

Dalvik project, at the source code level, is a rich set of tools:

- dalvikvm – program to support a command-line invocation of the Dalvik VM.
- dexdump – this tool is intended to mimic "objdump". Objdump displays information about one or more object files.
- dexgen – the dex code generator project. It provides API for creating dex classes in runtime which is needed e.g. for class mocking.

- dexlist – tool that lists all methods in all concrete classes in one or more DEX files.
- dexopt – a tool which verifies and optimizes all of the classes in the DEX file.
- dx – Dalvik eXchange, the thing that takes in class files and reformulates them for consumption in the VM.
- libdex – tool which is responsible for accessing .dex (Dalvik Executable Format) files.
- opcode-gen – set of scripts for modification of opcodes.
- tools – This tool runs a host build of dalvikvm in order to preoptimize dex files that will be run on a device.
  - dexdeps – DEX external dependency dump. This tool dumps a list of fields and methods that a DEX file uses but does not define.
  - dmtracedump – is a tool that gives you an alternate way of generating graphical call-stack diagrams from trace log files (instead of using Traceview).
  - gdbjithelper – kind of disassembler.
  - hprof-conv – tool to strip Android-specific records out of hprof data.
- vm – actual implementation of a VM:
  - alloc – Garbage-collecting memory allocator.
  - analysis – Dalvik bytecode structural verifier.
  - compiler –
  - hprof – Preparation and completion of hprof data generation.
  - interp – Main interpreter entry point and support functions.
  - jdwp – Java Debug Wire Protocol support. Prints a list of available JDWP processes on a given device.
  - mterp – the opcode interpreter
  - static opcode check;
  - ...

1 Second Experiment

My task is to find the code which is executed in order to launch Android application. To do so I did an experiment.

The experiment is very trivial: I'm calling an activity which is not registered in the manifest file. This causes runtime crash, therefore, I can exploit stack trace to track the execution of the program. This simulates the overall execution of Android app because, this involves low level functionality, like finding code at runtime which needs to be executed.

To remaining part of the section would be organized in a following manner: first two subsection are the problem definition and the remaining part is a top-down approach of tracing the root cause of the runtime crash. Also, additional comments on the involved Android libraries will be provided.

1.1 Who Calls onCreate()?

Every activity that we are creating has a method with this notation:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // place for your code
}
```

Who calls that method?

1.2 Official documentation on the issue

Figure from official Android documentation[1]:

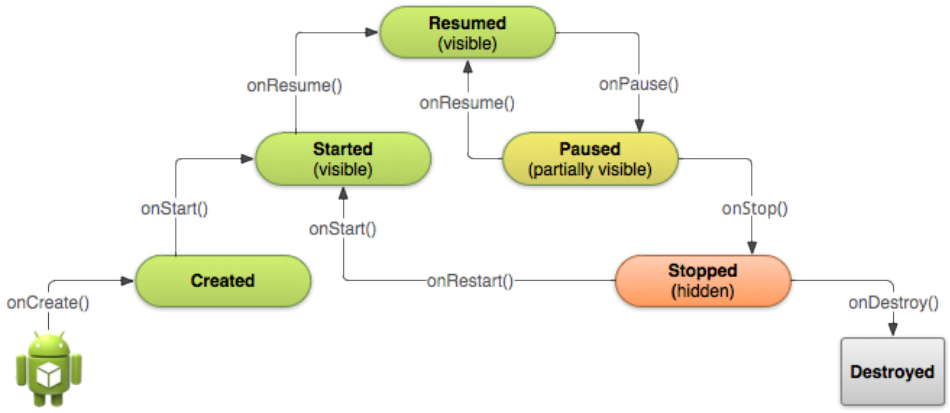

Figure 1: Activity lifecycle.

[1] http://developer.android.com/training/basics/activity-lifecycle/starting.html

1.3 Stack Trace

Here I provide stack trace of the runtime error while execution call of an activity which does not exist.

```
05-01 22:44:49.911: E/AndroidRuntime(1089): FATAL EXCEPTION: main
05-01 22:44:49.911: E/AndroidRuntime(1089): java.lang.RuntimeException: Unable to start activity ComponentInfo{com.example.test/com.example.test.MainAc
 {com.example.test/com.example.test.DummyActivity}; have you declared this activity in your AndroidManifest.xml?
05-01 22:44:49.911: E/AndroidRuntime(1089):  at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2663)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2679)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at android.app.ActivityThread.access$2300(ActivityThread.java:125)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at android.app.ActivityThread$H.handleMessage(ActivityThread.java:2033)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at android.os.Handler.dispatchMessage(Handler.java:99)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at android.os.Looper.loop(Looper.java:123)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at android.app.ActivityThread.main(ActivityThread.java:4627)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at java.lang.reflect.Method.invokeNative(Native Method)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at java.lang.reflect.Method.invoke(Method.java:521)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:868)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:626)
05-01 22:44:49.911: E/AndroidRuntime(1089):  at dalvik.system.NativeStart.main(Native Method)
```

1.4 class: Activity

An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View).

It's not that much surprising that the call is from the extended class:

```
final void performCreate(Bundle icicle) {
    onCreate(icicle);
    mVisibleFromClient = !mWindow.getWindowStyle().getBoolean(
            com.android.internal.R.styleable.Window_windowNoDisplay, false);
    mFragments.dispatchActivityCreated();
}
```

File: ./frameworks/base/core/java/android/app/Activity.java

Line: 5108

Good start!

1.5 class: Instrumentation

Base class for implementing application instrumentation code. When running with instrumentation turned on, this class will be instantiated for you before any of the application code, allowing you to monitor all of the interaction the system has with the application.

To executed code:

```
/**
 * Perform calling of an activity's {@link Activity#onCreate}
 * method.  The default implementation simply calls through to that method.
 *
 * @param activity The activity being created.
 * @param icicle The previously frozen state (or null) to pass through to
 *               onCreate().
 */
public void callActivityOnCreate(Activity activity, Bundle icicle) {
    ...
    activity.performCreate(icicle);
    ...
}
```

File: ./frameworks/base/core/java/android/app/Instrumentation.java
Line: 1080

1.6 class: ActivityThread

This manages the execution of the main thread in an application process, scheduling and executing activities, broadcasts, and other operations on it as the activity manager requests.

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    mInstrumentation.callActivityOnCreate(activity, r.state); // line: 2144
    ...
}
private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    Activity a = performLaunchActivity(r, customIntent); // line: 2230
    ...
}
```

File: ./frameworks/base/core/java/android/app/ActivityThread.java
Line: 2144

```
public void handleMessage(Message msg) {
switch (msg.what) {
    case LAUNCH_ACTIVITY: {
        Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
        ActivityClientRecord r = (ActivityClientRecord)msg.obj;

        r.packageInfo = getPackageInfoNoCheck(
                    r.activityInfo.applicationInfo, r.compatInfo);
        handleLaunchActivity(r, null); // line: 1234
        Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    } break;
    ...
}
```

1.7 class: Handler

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler, it is bound to the thread / message queue of the thread that is creating it – from that point on, it will deliver messages and runnables to that message queue and execute them as they come out of the message queue.

There are two main uses for a Handler: (1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be performed on a different thread than your own.

```
    /**
     * Handle system messages here.
     */
```

```
    public void dispatchMessage(Message msg) {
        if (msg.callback != null) {
            handleCallback(msg);
        } else {
            if (mCallback != null) {
                if (mCallback.handleMessage(msg)) {
                    return;
                }
            }
            handleMessage(msg);
        }
    }
```

Callback or not?
File: ./frameworks/base/core/java/android/os/Handler.java
Line: 90

1.8 class: Looper

Class used to run a message loop for a thread. Threads by default do not have a message loop associated with them; to create one, call prepare() in the thread that is to run the loop, and then loop() to have it process messages until the loop is stopped.

Most interaction with a message loop is through the Handler class.

```
/**
 * Run the message queue in this thread. Be sure to call
 * {@link #quit()} to end the loop.
 */
public static void loop() {
   ...
   msg.target.dispatchMessage(msg);
   ...
}
```

File: ./frameworks/base/core/java/android/app/ActivityThread.java
Line: 137

1.9 class: ActivityThread

Once again we are in this class, but it is because of the experiment.

```
public static void main(String[] args) {
    ...
    Looper.loop(); // line: 5048
    ...
}
```

1.10 class Method

This class represents a method. Information about the method can be accessed, and the method can be invoked dynamically.

File: ./libcore/luni/src/main/java/java/lang/reflect/Method.java

```
/**
 * This class represents a method. Information about the method can be accessed,
 * and the method can be invoked dynamically.
 */
...
public Object invoke(Object receiver, Object... args)
        throws IllegalAccessException, IllegalArgumentException, InvocationTargetException {
    if (args == null) {
        args = EmptyArray.OBJECT;
    }
    return invokeNative(receiver, args, declaringClass, parameterTypes, returnType, slot, flag);
}

private native Object invokeNative(Object obj, Object[] args, Class<?> declaringClass,
        Class<?>[] parameterTypes, Class<?> returnType, int slot, boolean noAccessCheck)
                throws IllegalAccessException, IllegalArgumentException,
                        InvocationTargetException; // line: 528
```

So far, so good, but now the real mess begins.

1.11 Native Method

```
File: ./dalvik/vm/native/java_lang_reflect_Method.cpp

/*
 * private Object invokeNative(Object obj, Object[] args, Class declaringClass,
 *   Class[] parameterTypes, Class returnType, int slot, boolean noAccessCheck)
 *
 * Invoke a static or virtual method via reflection.
 */
static void Dalvik_java_lang_reflect_Method_invokeNative(const u4* args,
    JValue* pResult)
{
```

What happens here is that runtime didn't find a class to which the needed method belongs.

1.12 class: ZygoteInit

Startup class for the zygote process.

Pre-initializes some classes, and then waits for commands on a UNIX domain socket. Based on these commands, forks off child processes that inherit the initial state of the VM.

```
file: ./frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
/**
 * Startup class for the zygote process.
 *
 * Pre-initializes some classes, and then waits for commands on a UNIX domain
 * socket. Based on these commands, forks off child processes that inherit
 * the initial state of the VM.
 ...
public static void main(String argv[]) {
    ...
    } catch (MethodAndArgsCaller caller) { //line 559
...
```

1.13 Grey zone

file: ./libcore/dalvik/src/main/java/dalvik/system/NativeStart.java

```
/**
 * Dummy class used during JNI initialization.  The JNI functions want
 * to be able to create objects, and the VM needs to discard the references
 * when the function returns.  That gets a little weird when we're
 * calling JNI functions from the C main(), and there's no Java stack frame
 * to hitch the references onto.
 *
 * Rather than having some special-case code, we create this simple little
 * class and pretend that it called the C main().
 *
 * This also comes in handy when a native thread attaches itself with the
 * JNI AttachCurrentThread call.  If they attach the thread and start
 * creating objects, we need a fake frame to store stuff in.
 */
```

dalvik.system.NativeStart.main(Native Method)
In other words: smoke and mirrors.

1.14 Conclusion

I have discovered how the Android framework deals with invoking new activity.
TODO: Why looper? What is H class? JPEG in the latex.