

Android Public Tracker > App Development > Android Studio > Deployment > C++ Debugger


123001559


← ↻ ☆ Android Studio 3.3 Native debugger execution control

+1

Hotlists (3)


Mark as Duplicate






Comments (8)DependenciesDuplicates (0)Blocking (0)Resources (0)


InfeasibleBugP2+ Add Hotlist

 STATUS UPDATE No update yet. Edit

 DESCRIPTION

om...@gmail.com created issue #1

Jan 18, 2019 12:46AM



Build: 3.3, AI-182.5107.16.33.5199772, 201812250239,

AI-182.5107.16.33.5199772, JRE 1.8.0_152-release-1248-b01x64 JetBrains s.r.o, OS Windows 10(amd64) v10.0 , screens 1920x1080, 2560x1440, 1920x1080

Android Gradle Plugin: 3.3.0

Gradle: 4.10.1

NDK: from local.properties: 19.0.5232133; latest from SDK: 19.0.5232133;

LLDB: LLDB 3.1 (revision: 3.1.4508709)

CMake: from local.properties: (not specified); latest from SDK: 3.6.0-rc2; from PATH: (not found);

In my project I have Java and Native C++ code. Stepping through Java code works OK, then Stepping into the JNI layer and next into C++ code also works well. However as soon as the debugger steps into C++ code, when stepping out, the debugger is actually stepping through the callstack of the ART VM implementation. One has to press Shift-F8 about 6 to 8 times (depending on the type of call ART needs to make) before returning to the point that a developer would consider "One call" back the callstack. I don't remember this working in this way with some previous versions of AndroidStudio.


What makes the problem worse is that as soon as the debugger tasted a bit of C++ code, one has to press Shift-F8 about 18 times to even step out of a Java call that was directly called from Java. So regular Java code debugging starts to work in the same way as well. At least for a while. Then it starts to work as expected again after a few calls.

In a run I just did, even stepping into usual Java code behaves like the debugger would be stepping through the ART code. I gave up pressing F8 while stepping into a Java call. Even F9 does not respond now properly. Behavior seems a little non-deterministic, with other words I am unable to fully explain it.

In the attached image of the callstack I marked two callstack blocks needed to step out of when doing two of my native code step-outs.

Similar debugger behavior was observed on a BQ Aquarius X5 Plus (Build number: 2.7.0_20181002-1711, Android: 7.1.1) and on a Nexus 5 (Build number M4B30Z, Android version: 6.0.1).

So far this makes debugging in AS not useful. The only way to work is to set a breakpoint wherever you want to stop and just use F9 to control the step-in and step-outs. F8 obviously still works as long as you stay within the same function.

 AndroidStudio-callstack.PNG

93 KB [View](#) [Download](#)

Reporter

om...@gmail.com

Type

Bug

Priority

P2

Severity

S3

Status

Won't fix (Infeasible)

Access

Default access [View](#)



Assignee

em...@google.com



Verifier

--

Collaborators

 [om...@gmail.com](#) 

CC

 [om...@gmail.com](#) [rm...@google.com](#) 

AOSP ID

--

Blocking Release

--

Release Status

--

Found In

5199772

Targeted To

--


Verified In

--

In Prod

☐


Show 1 additional field



COMMENTS


All comments

↓ Oldest first




rm...@google.com <rm...@google.com> #2

Jan 22, 2019 10:25PM




Assigned to ar...@google.com.

Thank you for your feedback. Team may reach out for more feedback in reproducing or triaging this issue.




ar...@google.com <ar...@google.com>

Jan 23, 2019 06:07AM




Reassigned to em...@google.com.




em...@google.com <em...@google.com> #3

Jan 24, 2019 07:00AM




Can you provide a simple project that reproduces the issue?

Specifically, I'd like to see how I can reproduce the particular stack trace you provided in the attachment, with all the ART methods in the call stack.



om...@gmail.com <om...@gmail.com> #4

Jan 25, 2019 12:18AM



I am a little surprised you had trouble reproducing the issue. The problem was easily reproduced on a Virtual Machine with a clean install of Windows 10 and a clean install of Android Studio 3.3. Same behavior previously occurred on my main system, but I am reluctant... to break it again. I don't think the VM version behaves much differently.

Steps to reproduce:
- open attached sample project (TestApp.zip)

- set a breakpoint at MainActivity.java:22
- set a breakpoint at native-lib.cpp:7
- press Shift-F9 (Debug app)
- wait.... :D (sorry, I had to...)
- try to step in / step over / step out

Pressing F7 on a simple native call brings me to some code in the memory include. See: AS_stack1_n5.png






If I set a breakpoint in the actual C++ implementation and press F9 to hit it - the trampolines start to show. See: AS_stack2_N5.png

These two stacks were recorded on a Nexus 5.

Same test done on a BQ Aquaris X5 Plus yields the following two stacks: AS_stack1_x5+.PNG and AS_stack2_x5+.PNG

I deliberately attached a sample app with all the build logs, objects and everything as produced by AS as the extra files may help solving the issue.

In the worst case... I could somehow package the VMWare machine and send it to you... Let me know if that would help.

-  **AS_stack1_n5.PNG**
20 KB [View](#) [Download](#)
-  **AS_stack1_x5+.PNG**
105 KB [View](#) [Download](#)
-  **AS_stack2_n5.PNG**
12 KB [View](#) [Download](#)
-  **AS_stack2_x5+.PNG**
120 KB [View](#) [Download](#)
-  **TestApp.zip**
14 MB [Download](#)

em...@google.com <em...@google.com> [#5](#)

Jan 26, 2019 01:34PM

In this particular example TestApp,

1. I hit the Java breakpoint at line 22,
2. I press "Step Into"
3. The C++ debugger stops at some method inside the <memory> header.
4. Android Studio line gutter shows symbolic breakpoint Java_.*stringFromJNI.*
5. From LLDB command line, I run:

```
(lldb) bt
* thread #1, name = 'example.testapp', stop reason = breakpoint 6.1
* frame #0: 0xd2351b08 libnative-lib.so::Java_com_example_testapp_MainActivity_stringFromJNI(JNIEnv *, jobject) [inlined]
std::__ndk1::__compressed_pair<.....(this=0xff9876e8) at memory:2216
  frame #1: 0xd2351b08 libnative-lib.so::Java_com_example_testapp_MainActivity_stringFromJNI(JNIEnv *, jobject) [inlined]
std::__ndk1::basic_string<.....="Hello from C++." at string:791
  frame #2: 0xd2351b08 libnative-lib.so::Java_com_example_testapp_MainActivity_stringFromJNI(env=0xed719400,
instance=0xff98773c) at native-lib.cpp:7
  frame #3: 0xed580b98 libart.so::art_quick_generic_jni_trampoline + 72
  frame #4: 0xed580e00 libart.so
```

Which shows that there are two methods inlined into the native method (i.e., JNI implementation) we are looking for (Note: functions with `__always_inline__` attribute get inlined in debug builds as well).

I reproduced this with Android Studio 3.2.1 with r18, so I don't think it was introduced by any of the most recent changes (i.e., AS3.3 or NDK19; although NDK19 upgrades the clang compiler).

It's unlucky that the C++ example we provide also hits this issue. A simple change to the source code like the following alters the behavior (to_string is done first, which is not inlined):

```
std::string hello = "Hello from C++ side" + std::to_string(123);
```

And the "Step Into" from Java to C++ works as expected for that case.

I will (1) look into what we can do about this, and also (2) try to reproduce the second part of your bug (i.e., too many stack frames in the backtrace).

om...@gmail.com <om...@gmail.com> [#6](#)

Jan 26, 2019 10:40PM

I can confirm that I also see the issue in Android Studio 3.2.1 and 3.2 with NDK R19. Same behavior as I described initially.

I see what you mean. The debugger did stop in the inlined code and the guts of the std::string implementation appear to be showing. And you are right that adding the " + std::to_string(123);" also resulted in a more subtle "Step into" experience. It is curious that some code would be always inlined. Debugging such code obviously becomes much harder. But I suppose the concern is with speed. Let me know if a solution to this issue was found.

The second part of my report, the Step out (Shift-F8) issue, works the same in AS 3.2.1 and 3.2 so it has likely been there from the start. When stepping out, the debugger is probably in the native context and stepping through ART code is business as usual. I suppose it is stepping through the code that is trying to push the return values back to the JVM (correct me if I am wrong).

Thank you for looking into the matter and for explaining the issue. For now a workaround is to heavily rely on breakpoints which is not THAT terrible... for simple cases.

Re-reading comment#1, I noticed that the issue described in comment#1 and what we have discussed in [comment#4, comment#5, and comment#6] are different issues. (I see the issue explained in comments 4/5/6 as will-not-fix [infeasible], as it requires LLDB support for dropping inlined frames that have identical \$PC values from stack trace; although I acknowledge that this is some minor UX issue).

What's described in comment#1: when the user wants to step out from Native back to Java, they need to hit "step out" many times to go over all the ART frames.

- **If you entered C++ code by using "Step Into" then you can hit "Resume" on the Native side, and you will be taken back to the Java side.**
 - When when you hit "Step Into" on Java side, the IDE also sets a Java breakpoint on the next expression after that JNI method call. That Java breakpoint is not removed while you are interacting with the native debugger.
- **If you entered C++ code without a Step Into (e.g., Resume), then there is no way to go back to the JNI call site, unless you manually set a breakpoint.**
 - In this case, the IDE does not set a Java breakpoint. In fact, even if you hit "Step Out" N times, you will still not be taken to the JNI call site. You might hit one of your other Java breakpoints, though.

I don't think there's anything we can do here. Even if the IDE finds out where to jump back on the Java side when the user presses "Step Out" from a JNI call, it cannot set a Java breakpoint at that location reliably, because that "Set Breakpoint" operation will enter a race with the program execution, and most likely lose it. This is because the Java debugger is running in-process, and is fully-stopped while the Native debugger is running; so the Java breakpoint does not get registered into the JVM until the app is resumed.

I am impressed with the diligence and focus you have put into this somewhat aged issue. You are right. The issue in comment#1 is different than the one we discussed later.

What you wrote in the first point on the transition back from C++ code to Java is really useful. Thank you. It didn't occur to me to try and use "Resume"... It is not entirely intuitive - but I recognize the technical challenge on your end. Perhaps it is a question of interpretation of what the user really wants vs. to be technically correct and execute debugger operations as requested by the user. I suppose it should be possible to detect that the user entered the C++ code from Java by using "Step Into". You probably already detect such calls to be able to inject the IDE breakpoint after the JNI call. And then a reasonable interpretation of a "Step Over" (F8) at the end of the native function, would be to instead execute a "Resume" (F9) operation to stop on the IDE injected Java breakpoint. I am guessing that everything that would needed for such handling is in place and implemented... The down side is that such handling would not allow the user to actually step back through the ART frames should that be desirable. The vast majority of end users probably expect the debugger behavior I just described.

In the second point you describe, the challenge is a lot bigger than in the first point. It would require mapping all "points of entry" from the user's code (but not platform code) and interpret a "Step Over" (F8) at the end of a native function, that was called from user's Java code base, to then interpret that "Step Over" as a "Resume" (F9)... Unless the user did actually hit Resume (F9) in which case the IDE injected breakpoint would need to be ignored... It sounds very involved and computationally intensive.

I see what you mean on the race condition. The implementation of debugging method I tried to describe in my second point is certainly questionable. Perhaps even not expected by most users. But there might be an elegant solution for it that I cannot see with my limited perspective on the problem.

Let me know what you think.

Status: Won't Fix (Infeasible)