Build and debug OpenJDK

Jul 8, 2019

I recently started reading the source code of OpenJDK 11, so I will summarize the build and debugging that I did.

1. Build

The official documentation for the build is Building the JDK . The amount is large, but the formula is also known and there is TL; DR first, so I think that you should check only there first. This time, I did the following procedure until the debug build.

```
# ソースを落とす。
# 今回は jdk ではなく jdk-updates というリポジトリを使用している。
# 過去のこのバージョンでビルドしたいというのがあればこのリポジトリを使うのが良さそう。
#
# jdk < 10 と jdk >= 10 でリポジトリ構成が変わっているので注意。
# http://openjdk.java.net/jeps/296
#
# "stream ended unexpectedly" で失敗しまくるときは少しずつ changeset を持ってくる手もある
# https://stackoverflow.com/questions/21655574/stream-ended-unexpectedly-on-clone
$ hg clone http://hg.openjdk.java.net/jdk-updates/jdk11u

# デバッグ情報付きでビルドできるように configure
# https://programmer.help/blogs/compile-and-debug-openjdk.html
# ビルドには jdk >= 10 を要求される
$ bash ./configure --disable-warnings-as-errors --enable-debug --with-native-debug-symbols=internal

# ビルド
$ make

# ビルドした JDK の確認
$ build/linux-x86_64-normal-server-fastdebug/jdk/bin/java -version
openjdk version "11.0.4-internal" 2019-07-16
OpenJDK Runtime Environment (fastdebug build 11.0.4-internal+0-adhoc.nm.openjdk11u)
OpenJDK 64-Bit Server VM (fastdebug build 11.0.4-internal+0-adhoc.nm.openjdk11u, mixed mode)
```

At the beginning, as for TL; DR, image instead of default was selected for make target. However, doing so from a clean state worked fine, but incremental compile failed (cpu usage increased abnormally and the PC stuck). According to Running Make , for the default target,

> This will build a minimal (or roughly minimal) set of compiled output (known as an “exploded image”) needed for a developer to actually execute the newly built JDK.The idea is that in an incremental development fashion, when doing a normal make , you should only spend time recompiling what's changed (making it purely incremental) and only do the work that's needed to actually run and test your code.

And for the image,

> Build the JDK image

Is written. To be honest, even if you read this, you do not know the specific difference between default and image target, but for debugging purposes, default seems to be sufficient, so I built with default from the middle.

2. Cooperation with CLion

Since I use CLion as C and C ++ IDE, I added CMakeLists.txt manually so that the source could be read there. The contents refer to Debugging C ++ code of OpenJDK with CLion .

CmakeLists.txt:

```
cmake_minimum_required(VERSION 3.7)
project(hotspot)
include_directories(
        src/hotspot/cpu
        src/hotspot/os
        src/hotspot/os_cpu
        src/hotspot/share
        src/hotspot/share/precompiled
        src/hotspot/share/include
        src/java.base/unix/native/include
        src/java.base/share/native/include
)
file(GLOB_RECURSE SOURCE_FILES "*.cpp" "*.hpp" "*.c" "*.h")
add_executable(hotspot ${SOURCE_FILES})
```

When CMakeLists.txt was placed in the project root and selected from File-> Open from CLion, some errors remained, but somehow the completion worked.

3. gdb debugging

Since it is built with debug information, you can set breakpoints and watchpoints based on the source code (the output below is partially omitted).

```
$ gdb --args build/linux-x86_64-normal-server-fastdebug/jdk/bin/java Hello
(gdb) b main
Breakpoint 1 at 0x10d0: file src/java.base/share/native/launcher/main.c, line 141.

(gdb) r
Starting program: build/linux-x86_64-normal-server-fastdebug/jdk/bin/java Hello
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffe618)
    at src/java.base/share/native/launcher/main.c:141
141         JLI_InitArgProcessing(jargc > 0, const_disable_argfile);

(gdb) b JavaMain
Breakpoint 2 at 0x7ffff7d6a060: file src/java.base/share/native/libjli/java.c, line 395.

(gdb) c
Continuing.
[New Thread 0x7ffff59ef700 (LWP 3535)]
[Switching to Thread 0x7ffff59ef700 (LWP 3535)]

Thread 2 "java" hit Breakpoint 2, JavaMain (_args=0x7fffffffa320)
    at src/java.base/share/native/libjli/java.c:395
395         int argc = args->argc;

(gdb) watch StubRoutines::_call_stub_entry
Hardware watchpoint 3: StubRoutines::_call_stub_entry

(gdb) c
Continuing.

Thread 2 "java" received signal SIGSEGV, Segmentation fault.
0x00007fffe100053b in ?? ()

(gdb) c
Continuing.
```

```
Thread 2 "java" hit Hardware watchpoint 3: StubRoutines::_call_stub_entry

Old value = (address) 0x0
New value = (address) 0x7fffe10008e4 "UH\213\354H\203\354`L\211M\370L\211E\360H\211M\350\211U\340H\211u\330H\2
StubGenerator::generate_initial (this=0x7ffff59ee8a0)
    at src/hotspot/cpu/x86/stubGenerator_x86_64.cpp:5593
5593        StubRoutines::_catch_exception_entry = generate_catch_exception();
```

In the above example, a Segmentation fault occurs on the way, but it seems that Hotspot VM may generate (?) As a normal system, so ignore it during debugging and proceed to c (continue) to reach the target location.

The Hotspot VM also makes extensive use of dynamic native code generation (other than JIT compilation), and watchpointing was `StubRoutines::_call_stub_entry`one such code. The dynamically generated code does not exist on the source, of course, but if you set a breakpoint at the address, you can check the register and stack just after the function call.

```
(gdb) b *0x7fffe10008e4
Breakpoint 4 at 0x7fffe10008e4

(gdb) c
...
Continuing.
[New Thread 0x7fffcd479700 (LWP 3804)]

Thread 2 "java" hit Breakpoint 4, 0x00007fffe10008e4 in ?? ()
(gdb) p /x $rbx
$6 = 0x7ffff59ee580
```

Another problem (not limited to OpenJDK) is that when playing with gdb, it is sometimes said that the target you want to print is optimized out.

```
(gdb) b generate_call_stub(unsigned char*&)
Breakpoint 3 at 0x7ffff73181a0: file src/hotspot/cpu/x86/stubGenerator_x86_64.cpp, line 213.

(gdb) c
Continuing.
...

Thread 2 "java" hit Breakpoint 3, StubGenerator::generate_call_stub (this=this@entry=0x7ffff59ee8a0,
    return_address=@0x7ffff7b69908: 0x0)
    at src/hotspot/cpu/x86/stubGenerator_x86_64.cpp:213
213         StubCodeMark mark(this, "StubRoutines", "call_stub");

(gdb) n
214         address start = __ pc();

(gdb) p start
$1 = <optimized out>
```

This looks like Stack Overflow is a situation where it is not possible to refer to it with gdb as a result of compile-time optimization. How to avoid optimization

- Make the variable volatile
- Tune optimization with compilation options (eg `-O0`)

There seems to be a method. This time, the latter is more convenient, so I reconfigured it as follows, but no change was seen. `make LOG=TRACE`Shows that the compile options are properly `-O0`added ...

```
$ bash ./configure --disable-warnings-as-errors --enable-debug --with-native-debug-symbols=internal --with-ext
$ make clean
$ make
```

4. Prepare hsdis

You mentioned that the Hotspot JVM generates native code dynamically, but you can use a tool called hsdis (hotspot diassembly) to dump the generated code (assembly) to standard output.

You need to build hsdis yourself, the procedure is as follows.

1. `cd hotspot/src/share/tools/hsdis`
2. Get the source from binutils (the version may be up to date)
3. Unpack and let it `hsdis/build/binutils`be placed as
4. `make all64`. `make all`If you don't need it, maybe all64 is good
5. `build/linux-amd64/hsdis-amd64.so<built_jdk_path>/jdk/lib/`Put the build products around

For the current jdk11, the fix mentioned here was necessary.

When hsdis is prepared and you pass development options `-XX:+PrintInterpreter`, `-XX:+PrintStubCode`such as to the java command, dynamically generated code will be output.

```
$ build/linux-x86_64-normal-server-fastdebug/jdk/bin/java -XX:+PrintInterpreter Hello

...
----------------------------------------------------------------------
iadd  96 iadd  [0x00007f86c1021b60, 0x00007f86c1021ba0]  64 bytes

  0x00007f86c1021b60: mov    (%rsp),%eax
  0x00007f86c1021b63: add    $0x8,%rsp
  0x00007f86c1021b67: mov    (%rsp),%edx
  0x00007f86c1021b6a: add    $0x8,%rsp
  0x00007f86c1021b6e: add    %edx,%eax
  0x00007f86c1021b70: movzbl 0x1(%r13),%ebx
  0x00007f86c1021b75: inc    %r13
  0x00007f86c1021b78: movabs $0x7f86d9a15d60,%r10
  0x00007f86c1021b82: jmpq   *(%r10,%rbx,8)
  0x00007f86c1021b86: nop
  0x00007f86c1021b87: nop
  0x00007f86c1021b88: int3
  0x00007f86c1021b89: int3
  0x00007f86c1021b8a: int3
  0x00007f86c1021b8b: int3
  0x00007f86c1021b8c: int3
  0x00007f86c1021b8d: int3
  0x00007f86c1021b8e: int3
  0x00007f86c1021b8f: int3
  0x00007f86c1021b90: int3
  0x00007f86c1021b91: int3
  0x00007f86c1021b92: int3
  0x00007f86c1021b93: int3
  0x00007f86c1021b94: int3
  0x00007f86c1021b95: int3
  0x00007f86c1021b96: int3
  0x00007f86c1021b97: int3
  0x00007f86c1021b98: int3
  0x00007f86c1021b99: int3
  0x00007f86c1021b9a: int3
  0x00007f86c1021b9b: int3
  0x00007f86c1021b9c: int3
  0x00007f86c1021b9d: int3
  0x00007f86c1021b9e: int3
  0x00007f86c1021b9f: int3
...
```

For example, this is the native code generated for a Java byte code called iadd. I thought it was interesting to know for the first time this time, but Hotspot VM generated native code for each Java byte code immediately after startup if it was the default behavior, and generated this code as much as possible when executing Java byte code. It seems to be an implementation that can be processed only with code. Until now, there was an image that Java byte code was read and the processing was separated by switch branch, etc., and executed using the stack area prepared for Hotspot VM.However, frequently-used methods were JIT-compiled. But it looks like they are doing something more aggressive. That's reasonable given the good performance of the JVM.

The mechanism around this is called Template interpreter in Hotspot VM. Now that the debugging environment has been set up, I will try to understand this Template interpreter in the future.