

Change Variable Value in JVM with GDB

Asked 3 years, 4 months ago Active 3 years, 4 months ago Viewed 673 times

Imagine yourself at Surescripts LLC



[View all 4 job openings!](#)

Currently I have a simple Java program:

```

3 public class Test {
    public static void main(String[] args) {
        boolean test = true;
        while (test) {
            System.out.println("Hello World");
            try { Thread.sleep(1000); } catch (Exception e) {}
        }
        System.out.println("Bye-bye");
    }
}

```

It prints "Hello World" every second. I would like to use gdb to attach to the process and make a memory patch to stop it with "Bye-bye" printed.

I know GDB can get created VMs (JNI_GetCreatedVMs) from its console, the env object is also available via the API of GetEnv. **How can I** find the `test` variable address in JVM and set it to false (this is optional) to **make the program exit normally**? Not sure if API like `AttachCurrentThread`, class like `HotSpotVirtualMachine`, tools like `jmap` or `jstack` can help?

And there is no debug option, assume the simple program running in production with `java -cp . Test .`

Thanks in advance for any guidance. :)

additional info (track state)

- `jmap -dump:file=hex <pid> && jhat hex` and browse at <http://localhost:7000>; cannot find any reference to `test` (it is not an object and just an instance of `class Z`)
- `jstack <pid>` can get the tid of main thread (0x7fa412002000) and `jhat hex` has the object of the `java.lang.Thread` of main (0x76ab05c40)
- `java.lang.Thread` has a native method `start0` which invokes hotspot method of `JVM_StartThread` (hotspot/src/share/vm/prims/jvm.cpp), there is a class `JavaThread` may contain the memory structure for local variables in thread stack.
- if `private static boolean test = true; ; then JNI_GetCreatedJavaVMs ==> jvm , jvm->jvm_api->AttachCurrentThread ==> env , env->env_api->(FindClass, GetStaticFieldID, SetStaticBooleanField) ==> test[true ==> false]`

java

jvm

gdb

inject

edited Aug 14 '16 at 6:07

asked Aug 13 '16 at 9:25



Doz Parp

134 2 15

Generally this is not possible. `test` variable may not even exist in memory. As soon `main` method is JIT-compiled, there will be unconditional infinite loop, because the compiler realizes that `test` value never changes, and optimizes it out. – [apangin](#) Aug 13 '16 at 12:36

@apangin thx for the comment. how about the updated one? The real purpose is that I would like to patch memory of a local variable of a function call. – [Doz Parp](#) Aug 13 '16 at 13:22

have you considered using the debugger protocol instead? – [the8472](#) Aug 13 '16 at 14:00

@DozParp Even in the updated example `test` variable may not have a location in memory - it may be cached in CPU registers. JVM does not record a mapping between original variables and their actual locations in compiled code unless JVM is started with debug agent. – [apangin](#) Aug 14 '16 at 2:02 ✎

Are you sure you can't use the Java debugger or the protocol it uses. That will be a lot simpler. – [Peter Lawrey](#) Aug 14 '16 at 2:48

1 Answer



4



In some cases it is possible to get local variable addresses using HotSpot Serviceability Agent. I've made a sample agent that prints extended stack traces with local variable info:

```
import sun.jvm.hotspot.code.Location;
import sun.jvm.hotspot.code.LocationValue;
import sun.jvm.hotspot.code.NMethod;
import sun.jvm.hotspot.code.ScopeValue;
import sun.jvm.hotspot.code.VMRegImpl;
import sun.jvm.hotspot.debugger.Address;
import sun.jvm.hotspot.debugger.OopHandle;
import sun.jvm.hotspot.interpreter.OopMapCacheEntry;
import sun.jvm.hotspot.oops.Method;
import sun.jvm.hotspot.oops.Oop;
import sun.jvm.hotspot.runtime.CompiledVFrame;
import sun.jvm.hotspot.runtime.InterpretedVFrame;
import sun.jvm.hotspot.runtime.JavaThread;
import sun.jvm.hotspot.runtime.JavaVFrame;
import sun.jvm.hotspot.runtime.VM;
import sun.jvm.hotspot.runtime.VMReg;
import sun.jvm.hotspot.tools.Tool;

import java.util.List;

public class Frames extends Tool {

    @Override
    public void run() {
        for (JavaThread thread = VM.getVM().getThreads().first(); thread != null; thread =
thread.next()) {
            System.out.println(thread.getThreadName() + ", id = " +
thread.getOSThread().threadId());
        }
    }
}
```

```

        for (JavaVFrame vf = thread.getLastJavaVFrameDbg(); vf != null; vf =
vf.javaSender()) {
            dumpFrame(vf);
        }
        System.out.println();
    }

    private void dumpFrame(JavaVFrame vf) {
        Method method = vf.getMethod();
        String className = method.getMethodHolder().getName().asString().replace('/',
'.');
        String methodName = method.getName().asString() +
method.getSignature().asString();
        System.out.println("  # " + className + '.' + methodName + " @ " + vf.getBCI());

        if (vf.isCompiledFrame()) {
            dumpCompiledFrame(((CompiledVFrame) vf));
        } else {
            dumpInterpretedFrame(((InterpretedVFrame) vf));
        }
    }

    private void dumpCompiledFrame(CompiledVFrame vf) {
        if (vf.getScope() == null) {
            return;
        }

        NMethod nm = vf.getCode();
        System.out.println("    * code=[" + nm.codeBegin() + "-" + nm.codeEnd() + "], pc="
+ vf.getFrame().getPC());

        List locals = vf.getScope().getLocals();
        for (int i = 0; i < locals.size(); i++) {
            ScopeValue sv = (ScopeValue) locals.get(i);
            if (!sv.isLocation()) continue;

            Location loc = ((LocationValue) sv).getLocation();
            Address addr = null;
            String regName = "";

            if (loc.isRegister()) {
                int reg = loc.getRegisterNumber();
                addr = vf.getRegisterMap().getLocation(new VMReg(reg));
                regName = ":" + VMRegImpl.getRegisterName(reg);
            } else if (loc.isStack() && !loc.isIllegal()) {
                addr = vf.getFrame().getUnextendedSP().addOffsetTo(loc.getStackOffset());
            }

            String value = getValue(addr, loc.getType());
            System.out.println("    [" + i + "] " + addr + regName + " = " + value);
        }
    }

    private void dumpInterpretedFrame(InterpretedVFrame vf) {
        Method method = vf.getMethod();
        int locals = (int) (method.isNative() ? method.getNumberOfParameters() :
method.getMaxLocals());
        OopMapCacheEntry oopMask = method.getMaskFor(vf.getBCI());

        for (int i = 0; i < locals; i++) {
            Address addr = vf.getFrame().addressOfInterpreterFrameLocal(i);
            String value = getValue(addr, oopMask.isOop(i) ? Location.Type.OOP :
Location.Type.NORMAL);
            System.out.println("    [" + i + "] " + addr + " = " + value);
        }
    }

```

```

    }
}

private String getValue(Address addr, Location.Type type) {
    if (type == Location.Type.INVALID || addr == null) {
        return "(invalid)";
    } else if (type == Location.Type.OOP) {
        return "(oop) " + getOopName(addr.getOopHandleAt(0));
    } else if (type == Location.Type.NARROWOOP) {
        return "(narrow_oop) " + getOopName(addr.getCompOopHandleAt(0));
    } else if (type == Location.Type.NORMAL) {
        return "(int) 0x" + Integer.toHexString(addr.getJIntAt(0));
    } else {
        return "(" + type + ") 0x" + Long.toHexString(addr.getJLongAt(0));
    }
}

private String getOopName(OopHandle hadle) {
    if (hadle == null) {
        return "null";
    }
    Oop oop = VM.getVM().getObjectHeap().newOop(hadle);
    return oop.getKlass().getName().asString();
}

public static void main(String[] args) throws Exception {
    new Frames().execute(args);
}
}

```

To run it:

```
java -cp $JAVA_HOME/lib/sa-jdi.jar:. Frames PID
```

This will attach to Java process PID and print the stacktraces like

```

main, id = 30920
# java.lang.Thread.sleep(J)V @ 0
# Test.main([Ljava/lang/String;)V @ 15
[0] 0x00007f075a857918 = (oop) [Ljava/lang/String;
[1] 0x00007f075a857910 = (int) 0x1
[2] 0x00007f075a857908 = (int) 0x0

```

Here main is Java thread name; 30920 is native thread ID; @ 15 is bytecode index.

The line [1] 0x00007f075a857910 = (int) 0x1 means that the local variable #1 is located at address 0x00007f075a857910 and has the value 1. This is exactly the variable you are interested in.

The local variable information is reliable for interpreted methods, but not always for compiled methods. However, compiled methods will have an extra line with an address of the code, so you can disassemble and inspect it in gdb.

answered Aug 14 '16 at 23:13




apangin

61.6k 9 130 164

that's cool, thx so much, very detailed stack trace! :D just wonder how hotspot works there. do you have any book or document page about it? – Doz Parp Aug 15 '16 at 4:11

@DozParp blogs.oracle.com/thejavatutorials/entry/... – apangin Aug 15 '16 at 22:15

@apangin , Can you kindly help to give the details for "compiled methods will have an extra line with an address of the code, so you can disassemble and inspect it in gdb." ? As I know compiled Java methods cannot be display from the call stack trace as gdb just display the native methods, so how can it be disassemble and inspect it in gdb? It is better if you can kindly help to give an example for checking the variable value for jitted methods with GDB. Please let me know if I need to create a new question. – Jason Jan 28 '17 at 6:02 

@apangin , BTW, besides the variable value and the address, is it possible to get the variable name? For example `int abcd = 123`; how to get the name `abcd`? – Jason Jan 28 '17 at 7:10

@Jason Yes, if a class file contains debug information. See [Method.getLocalVariableName](#) – apangin Jan 31 '17 at 21:02

@apangin, I found the `getLocalVariableName(int bci, int slot)` need the bci, and I got it by `LocalVariableTable[i].getStartBCI()` , however if the `LocalVariableTable` info is turn off when compiling, this way doesn't work, and especially I found this way doesn't display the variable name for `Array`. So may I need if there are better way to get the bci or better way to display the variable name? – YuFeng Shen Feb 20 '17 at 3:58

@Jacky If Java sources are compiled without debug info, local variable names will not be available. – apangin Feb 20 '17 at 7:09

@apangin ,BTW , why the local variable information is reliable for interpreted methods, but not always for compiled methods ?It is a bug of SA or there is some technical barrier? – YuFeng Shen Mar 18 '17 at 8:00

@Jason Because compiled methods do not preserve local variables. Some variables may be optimized away altogether. SA can do nothing about that. – apangin Apr 2 '17 at 15:22
