Creating Your Own Debugging Tools
Why Build a Custom Tool?
jvmstat Performance Counters
Dynamic Attach and Instrumentation API
Serviceability Agent
JVM Tool Interface
Changes in JDK 9
Conclusion

DEBUGGING JVM INTERNALS

Creating Your Own Debugging Tools

JDK serviceability technologies allow you into the JVM to solve difficult debugging problems.

by Andrei Pangin

Java is more than just a programming language; it is a comprehensive platform for developing and running application software. One of the most recognized advantages of the Java platform is the large set of serviceability and maintainability features that are especially important for enterprise applications. Besides the numerous built-in tools, there is a lot of third-party software to assist with troubleshooting.

Every decent IDE for Java has a powerful debugger that supports step-by-step execution, breakpoints, watches, and so on. To address performance issues, there are well-known profilers such as Oracle Java Mission Control, JProfiler, and YourKit. Memory leaks are another prevalent problem. There are appropriate tools to deal with memory issues, too—for example, VisualVM and Eclipse Memory Analyzer. General-purpose tools are good for solving typical problems developers often face. However, they do not always fit uncommon situations. I believe most of you know how to use standard tools, so here I discuss how to create new tools customized for your specific cases.

Why Build a Custom Tool?

Imagine a situation in which something has gone wrong with an application on a production server while you are out of the office. It would be helpful to take a heap dump for analysis, but the internet connection might not be good enough to download a multigigabyte dump file. The only thing you can do is run something small remotely. But what tool would you run? Sometimes it is faster to make a specialized tool by yourself rather than search for an existing one that might or might not help in a particular case.

Another instance where a custom tool can be effective is with the problem of ignored exceptions. It is a common mistake to catch declared exceptions and discard them without handling them. The problem is worse when this happens in a third-party library that you cannot modify. I encountered this bug in a proprietary JDBC driver that did not handle an unexpected error from a database server and, thus, could not properly invalidate a stale connection. Even when you have no control over exceptions in a third-party library, you can still intercept them with a specialized tool. I show how to do that in this article.

If you ever wanted to patch a running application without creating a service interruption, you might be interested in a tool capable of modifying loaded code. Of course, there are commercial solutions that can do the job, but why not fix the problem yourself with just a few lines of code?

The list of tasks that might benefit from a custom tool is endless. Thanks to serviceability components included in Java SE Development Kit (JDK), the creation of such tools is much easier. While each of these

components deserves a separate article, the following overview sheds light on what these technologies can do.

jvmstat Performance Counters

Monitoring the JVM is one of the key approaches to ensure that a system works well. Java HotSpot VM provides a huge amount of telemetry data through jvmstat performance counters. This data includes several hundred indicators covering nearly all JVM areas: class loading, garbage collection, multithreading, just-in-time compilation, and more. Despite the name, the tools are not all actually counters, and not all of them are about performance; nevertheless, they are very useful for monitoring JVM health. You might think of performance counters as gauges and dials in the cockpit of an aircraft.

jvmstat counters are available at no cost; that is, Java HotSpot VM exports them anyway, whether you read them or not. The JVM publishes the telemetry data onto the file system as a memory-mapped file in a temporary directory, often called /tmp/hsperfdata_{user}/{pid}, where *{pid}* is a Java process ID. This naming convention makes it possible for tools to find running Java processes in the system.

You can ask the JVM to dump a Java heap, print stack traces, change certain VM flags, load an agent library, and so on.

Fortunately, there is no need to replicate the directory scanning logic, because there is already a convenient Java API for that. Although the jvmstat API is not standard, it is supplied with the standard JDK bundle. You only need to include {JAVA_HOME} `/lib/tools.jar` in the classpath.

Here is how to get the process IDs of all running Java HotSpot VMs in the system:

```
import sun.jvmstat.monitor.MonitoredHost;
...
MonitoredHost host =
    MonitoredHost.getMonitoredHost((String) null);

Set<Integer> processIds = host.activeVms();
```

In this code, `null` stands for the local host. It is also possible to monitor remote virtual machines if a remote host runs the `jstatd` utility. Once you have a process ID, you can obtain an instance of `MonitoredVm` and read its jvmstat counters (or monitors). The `Monitor` type can be Integer, Long, or String. For example, a monitor named `sun.rt.javaCommand` contains the main class and the arguments used to start the given Java application. To get all the monitors matching the specified regular expression, use the `findByPattern` method, as shown next:

```
MonitoredVm vm = host.getMonitoredVm(
    new VmIdentifier(processId.toString()));

vm.findByPattern(".*").forEach(monitor -> {
    System.out.println(monitor.getName() + " = " +
                       monitor.getValue());
});
```

That simple code lists all available monitors with their values. In its output, you can find interesting metrics that are hardly shown by standard tools. For example:

```
// Total time spent in class initializers
sun.cls.classInitTime = 2545394
```

```
// Number of contended synchronizations
sun.rt._sync_ContendedLockAttempts = 55
// Duration of stop-the-world VM pauses
sun.rt.safepointTime = 811588
```

For instance, safepoint time is critical for low-latency applications, because it shows how long the application threads were forcibly stopped by the VM. If you choose to monitor safepoint pauses or any other of the 250+ counters, this is already a good monitoring tool, isn't it? It could be further improved to show a dynamic profile collected over time.

Unfortunately, you cannot do much just with read-only counters. Two-way communication with the JVM requires a different technology.

Dynamic Attach and Instrumentation API

The Dynamic Attach mechanism provides the means to connect to a running VM and execute one of several predefined commands. You can ask the JVM to dump a Java heap, print stack traces, change certain VM flags, load an agent library, and so on. The VM executes commands on its own, so it must be alive and healthy in order to respond.

The Java API for Dynamic Attach is available in the same tools.jar file. Note that this is a vendor-specific API applicable only to OpenJDK and Oracle's JDK.

Attaching to a running Java process is straightforward; you need to know only the target process ID (pid) as shown in the following code. (Dynamic Attach requires no special VM options. It can connect to any local HotSpot JVM unless it is started with the `-XX:+DisableAttachMechanism` flag.)

```
import com.sun.tools.attach.VirtualMachine;
...
VirtualMachine vm = VirtualMachine.attach(pid);
try {
    vm.loadAgent(agentJarPath, options);
} finally {
    vm.detach();
}
```

This shows how to inject a Java agent into a running VM. A *Java agent* is a utility program for instrumenting an application. It should be packed into a JAR file and contain a class with an `agentmain` method.

The instrumentation API enables Java agents to transform the bytecode of existing classes. When used together with Dynamic Attach, it enables you to change the code of a running application, even if the application is started without any debugging facilities. Here is a simple agent that installs a new version of `MyClass`:

```
public static void agentmain(String args,
            Instrumentation instr) throws Exception

    Class oldClass = Class.forName("org.pkg.MyClass"
    Path newFile = Paths.get("/path/to/MyClass.class
    byte[] newData = Files.readAllBytes(newFile);

    instr.redefineClasses(
            new ClassDefinition(oldClass, newData))
}
```

Remember the limitations of the `redefineClasses` API: a new version of a class file cannot add new methods or fields, nor can it remove existing members. Basically, only method bodies can be changed, but this is often enough for a hot fix.

The ability to instrument running Java processes makes the Attach API an important tool for maintenance of enterprise applications. The

Dynamic Attach mechanism requires full cooperation from the JVM. It becomes useless if the JVM has hung or become too busy. When this happens, it is time to call for brute force, such as the serviceability agent.

Serviceability Agent

The HotSpot Serviceability Agent (SA) provides a low-level view of a Java process from a VM perspective. It knows everything about Java HotSpot VM internal structures, including the heap layout, the system dictionaries, the compiled code, the threads, and the stacks. Moreover, this information is available through a clear and simple Java API, so developers can benefit from it without having experience in disassemblers and other hacker facilities.

The SA was originally invented by Java HotSpot VM engineers for debugging crashes inside the JDK. However, they later realized that it could be helpful for a wider group of developers, and now it is bundled with the regular JDK. Start using the SA by including {JAVA_HOME} `/lib/sa-jdi.jar` in the classpath, but remember that the API is not standard and is subject to change in any future JDK release.

Custom tools typically extend an existing `Tool` class, which is already capable of parsing arguments and attaching to a running VM. You just need to implement custom logic inside the overridden `run` method.

```
import sun.jvm.hotspot.runtime.VM;
import sun.jvm.hotspot.tools.Tool;

public class MyTool extends Tool {

    @Override
    public void run() {
        // Actual implementation
        VM.getVM()...
    }

    public static void main(String[] args) {
        new MyTool().execute(args);
    }
}
```

`VM.getVM()` is the starting point to access Java HotSpot VM internal structures. The next example employs `SystemDictionary` to traverse all loaded classes with their class loaders. A similar technique might be useful in detecting memory leaks related to class loading.

```
VM.getVM().getSystemDictionary()
        .classesDo((klass, loader) -> {
    String className = klass.getName().asString();
    System.out.print(className);

    String loaderName = (loader == null)
            ? "Bootstrap ClassLoader"
            : loader.getKlass().getName().asString(
    System.out.println(" loaded by " + loaderName);
});
```

That was rather simple. The real power of the SA is to restore VM structures, either from the memory of a live Java process or from the core dump of an abnormally terminated process when the operating system is configured to create such dumps. The SA provides the reflection-like API to inspect Java objects and to extract the required fields. Unlike the reflection, which works from within the same process, the SA reads memory of a different process or parses a core dump file. Tools based on this feature can do impressive tricks such as stealing private keys from a running web server. The following code scans the heap of a target process looking for the instances of `java.security.PrivateKey` and printing their contents.

```
Klass keyClass = VM.getVM().getSystemDictionary()
        .find("java/security/PrivateKey", null, null

VM.getVM().getObjectHeap()
      .iterateObjectsOfKlass(new DefaultHeapVisitor
   @Override
   public boolean doObj(Oop obj) {
      InstanceKlass c = (InstanceKlass) obj.getKlass
      OopField f = (OopField) c.findField("key", "[I
      TypeArray key = (TypeArray) f.getValue(obj);
      key.printOn(System.out);
      return false;
   }
}, keyClass);
```

The SA needs no cooperation from the target JVM, and there is no way to protect against SA interactions. This is not a reason to worry, though. The SA typically requires root privileges to attach to a running process. Also, keep in mind that the target JVM remains suspended while the SA is attached.

So far, I have focused on JDK internal APIs. If you are looking for a more standard way to build your own tool, consider using the JVM tool interface.

JVM Tool Interface

The JVM tool interface (JVM TI) is a standard programming interface designed especially for debugging, monitoring, and profiling software intended to run on top of the JVM. The best thing about JVM TI is its public specification, which is not tied to any particular implementation. It is not required that every JVM provide all JVM TI functionality; however, most popular JVMs do.

> The JVM tool interface (JVM TI) is a standard programming interface designed especially for debugging, monitoring, and profiling software intended to run on top of the JVM.

The interface is exposed through the C header file `jvmti.h`. JVM TI–based tools, called agents, are typically written in C or C++. They run within the same process and communicate with the JVM directly by calling JVM TI functions. The interface looks somewhat similar to Java Native Interface (JNI), so if you have ever written JNI code, you will easily grasp the principles of using JVM TI.

An agent may start at JVM bootstrap (when specified in `-agentlib` or `-agentpath` JVM arguments), or it can be loaded later at runtime using the Dynamic Attach mechanism. To support these options, an agent should define one or several entry points:

- `Agent_OnLoad`, which is called automatically by the JVM early at startup time
- `Agent_OnAttach`, which is called whenever the library is loaded at runtime

The first thing an agent typically does is get the reference to the JVM TI environment (`jvmtiEnv`), which is necessary for calling JVM TI functions.

```
#include <jvmti.h>

JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *args, void *unused)
    jvmtiEnv *jvmti;
    vm->GetEnv((void**)&jvmti, JVMTI_VERSION_1_0);

    // Initialization code
```

```
    return 0;
}
```

JVM TI has functions for everything debuggers usually do. You can manage threads, walk through their stacks, iterate through the Java heap, query local variables, set breakpoints, manipulate Java classes, intercept native methods, and do many other things. Besides that, an agent may subscribe to event notifications: the JVM will invoke a provided callback function whenever an event occurs.

The access to JVM TI functionality is capability-based; that is, an agent must explicitly request the capabilities it is going to use. Most of the capabilities are available at runtime, but some can be requested only during the `OnLoad` phase (the time when no classes have been loaded and no bytecodes have been executed). For example, the `can_access_local_variables` capability is available only at startup, because the JVM needs to disable certain optimizations beforehand in order to retain information about all local variables.

The following example requests a capability to generate exception events and sets up the callback to receive notifications about all thrown Java exceptions, both caught and uncaught.

```
jvmtiCapabilities capabilities = {0};
capabilities.can_generate_exception_events = 1;
jvmti->AddCapabilities(&capabilities);

jvmtiEventCallbacks cb = {0};
cb.Exception = ExceptionCallback;

jvmti->SetEventCallbacks(&cb, sizeof(cb));
jvmti->SetEventNotificationMode(
JVMTI_ENABLE, JVMTI_EVENT_EXCEPTION, NULL);
```

The callback function receives all details about an exception: a thread, a method, and the bytecode index for the thrown exception. The callback also has a reference to the JNI environment. This means you can invoke any JNI function from within. For instance, you can use JNI to call `Throwable.printStackTrace()`. Thus, an agent will print all the exceptions, including ignored exceptions, just before they are caught.

```
void JNICALL ExceptionCallback(
        jvmtiEnv *jvmti, JNIEnv *env, jthread threa
        jmethodID method, jlocation location,
        jobject exception, jmethodID catch_method,
        jlocation catch_location)
{
    jclass cls = env->FindClass("java/lang/Throwable
    jmethodID print_method = env->
        GetMethodID(cls, "printStackTrace", "()V");
    env->CallVoidMethod(exception, print_method);
}
```

You can do a lot more useful things with the JVM TI. Besides exceptions, it is possible to trace class loading, garbage collection, lock contention, thread activity, and more.

JVM TI is often confused with the Java debugger agent. There is a popular misconception that JVM TI compromises security and degrades the performance of Java applications. However, the Java Debug Wire Protocol (JDWP) agent is just one example of a JVM TI–based tool; the technology itself does not imply security or performance consequences. Whether an application will suffer from agent overhead solely depends on what the agent does and which capabilities it requests. Consider JVM TI as a sort of extension to JNI. This technology is definitely worth trying.

Changes in JDK 9

All the technologies discussed previously, including private APIs, will remain functional in the upcoming JDK 9. However, the new module system imposes certain restrictions on how you can access these APIs. No longer will tools.jar and sa-jdi.jar be separate libraries. JDK 9 serviceability features are supplied in the dedicated modules. Table 1 shows the location of key JAR files in Java 9.

FEATURE	MODULE	PUBLIC
JVMSTAT PERFORMANCE COUNTERS	jdk.jvmstat	NO
DYNAMIC ATTACH API	jdk.attach	YES*
INSTRUMENTATION AP	java.instrument	YES
SERVICEABILITY AGENT	jdk.hotspot.agent	NO

*`com.sun.tools.attach.VirtualMachine` is accessible from outside, but `sun.tools.attach.HotSpotVirtualMachine` is not.

Table 1. The location of serviceability APIs in Java 9

By default, applications cannot access an API from the modules that do not export packages externally. In order to use the private APIs, you need to explicitly break the encapsulation with the `--add-exports` JVM command-line argument. For example, to run a tool that depends on the sun.jvmstat.monitor package, use:

```
java --add-exports jdk.jvmstat/sun.jvmstat.monitor=
     ALL-UNNAMED MyTool
```

[The previous line should be written as a single line with no space after the = sign.—*Ed.*]

Feel free to use the private APIs for your own purposes, but do it with care: there are no guarantees that the APIs will continue to work in future JDK updates.

Conclusion

The Java platform comes with a set of versatile technologies for building custom debugging, monitoring, and troubleshooting tools. Some of them are covered by Java SE standards, while others are specific to OpenJDK and Oracle's JDK. Despite the lack of thorough documentation on private APIs, the source code of the OpenJDK project (particularly, the source code of JDK built-in tools) might serve as a good starting point for learning serviceability technologies.

Software development and maintenance can hardly succeed without proper tools. Although many tools exist in the market, there is no silver bullet to address all problems. As a Java developer, you can create your own tools to solve tasks that no other software solves. JDK serviceability technologies are your friends.

This article was originally published in the January/February 2017 issue of *Java Magazine*.

Andrei Pangin

Andrei Pangin (@AndreiPangin) leads the development of the Odnoklassniki social network. He previously worked on the HotSpot JVM, which became his favorite topic and area of expertise. Pangin is a frequent speaker at Java conferences and one of the top JVM answerers on Stack Overflow.