Blaming bad code (or bad code monkeys) won’t help you find performance bottlenecks and improve the speed of your Java™ applications, and neither will guessing. Ted Neward directs your attention to tools for Java performance monitoring, starting with five tips for using Java 5's built-in profiler, JConsole, to collect and analyze performance data.

5 things you didn't know about ... Java performance monitoring, Part 1
Java performance profiling with JConsole and VisualVM

Skill Level: Introductory

**Ted Neward** (ted@tedneward.com)
Principal
Neward & Associates

29 Jun 2010

When application performance suffers, most developers panic, and with good reason. Tracking the source of Java application bottlenecks has historically been a major pain, both because the Java virtual machine has a black-box effect, and because profiling tools for the Java platform have traditionally fallen short.

All of that changed with the introduction of JConsole in Java 5, however. JConsole is a built-in Java performance profiler that works from the command-line and in a GUI shell. It’s not perfect, but it’s a more than adequate first line of defense when pointy-head boss comes at you with a performance problem — and it’s a whole lot
better than consulting Papa Google.

In this edition of the 5 things series, I'll show you five easy ways to use JConsole (or its visually sophisticated cousin, VisualVM) to monitor Java application performance and track bottlenecks in your Java code.

1. The JDK ships with a profiler

Many Java developers don't realize that a profiler tool has been included in the JDK since Java 5. JConsole (or, for more recent Java platform releases, VisualVM) is a built-in profiler that is as easy to launch as the Java compiler. From a command prompt that has the JDK on the PATH, just run `jconsole`. From a GUI shell, navigate to the JDK installation directory, open the bin folder, and double-click `jconsole`.

When the profiler tool pops up (depending on which version of Java is running and how many other Java programs are running at the moment), it either presents a dialog box asking for a URL of a process to connect to, or lists a number of different local Java processes to connect to — sometimes, including the JConsole process itself.

### JConsole or VisualVM?

JConsole has shipped with every Java platform release since Java 5. VisualVM is an updated profiler based on the NetBeans platform, which was first included somewhere around Java 6 update 12. Most shops haven't upgraded to Java 6 yet, so this article focuses on JConsole. Most tips should be relevant to both profilers, however. *(Note: In addition to being included in Java 6, VisualVM is a stand-alone download. See Resources to download VisualVM.)*

### Working with JConsole

In Java 5, Java processes are not set up by default to be profiled. Passing a command-line argument — `-Dcom.sun.management.jmxremote` — at startup tells the Java 5 VM to open the connections, so that the profiler can find them. Once a process is picked up by JConsole, you can just double-click it to start profiling.

Profiler have their own overhead, so it's a good idea to spend a few minutes figuring out what that is. The easiest way to discover JConsole's overhead is to first run an application by itself, then run it under the profiler, and measure the difference. (The app shouldn't be too large or too small; my favorite is the SwingSet2 demo app that ships with the JDK.) So, for instance, I tried running SwingSet2 with `-verbose:gc` to see garbage collection sweeps, then ran the same app and connected the JConsole profiler to it. When JConsole was connected, a steady stream of GC sweeps happened that didn't occur otherwise. That was the
performance overhead of the profiler.

2. Remotely connect to processes

Because Web application profilers assume connectivity across a socket for profiling, you only need a little configuration to set up JConsole (or any JVMTI-based profiler, for that matter) to monitor/profile applications running remotely.

For example, if Tomcat were running on a machine named “webserver” and that JVM had JMX enabled and listening on port 9004, connecting to it from JConsole (or any other JMX client) would require a JMX URL of "service:jmx:rmi:///jndi/rmi://webserver:9004/jmxrmi".

In essence, all you need to profile an application server running in a remote data center is the JMX URL. (See Resources for more about remote monitoring and management with JMX and JConsole.)

3. Track statistics

Don’t be typical

Common responses to discovering a performance problem in application code vary, but they’re predictable, too. Developers who have been programming since the early days of Java are likely to fire up the old IDE and start doing code reviews of major parts of the code base, looking for familiar “red flags” in the source like synchronized blocks, object allocations, and the like. With fewer years of programming, a developer will probably pore over the -X flags that the JVM supports, looking for ways to optimize the garbage collector. And newbies, of course, go straight to Google, hoping that somebody else out there has found the JVM's magical “make it go fast” switch, so that they can avoid having to rewrite any code.

There's nothing intrinsically wrong with any of these approaches, but they're all a crapshoot. The most effective response to a performance problem is to use a profiler — and now that they're built in to the Java platform, we really have no excuse not to!

JConsole has a number of tabs that are useful for collecting statistics, including:

- **Memory**: For tracking activity against the various heaps in the JVM’s garbage collector.
- **Threads**: For examining the current thread activity in the targeted JVM.
- **Classes**: For watching the total loaded class count for a VM.
These tabs (and the associated graphs) are all courtesy of the JMX objects that every Java 5-and-later VM registers with the JMX server, which is built-in to the JVM. The complete list of beans available within a given JVM is listed in the MBeans tab, complete with some metadata and a limited user interface for seeing that data or executing those operations. (Registering for notifications is beyond the JConsole user interface, however.)

**Using statistics**

Say a Tomcat process keeps dying from OutOfMemoryErrors. If you want to find out what’s going on, open JConsole, click the Classes tab, and keep a lazy eye on the class count as time goes by. If the count steadily rises, then you can assume that either the app server or your code has a ClassLoader leak somewhere and will run out of PermGen space before long. Check the Memory tab if you need to further confirm the problem.

**4. Create a heap dump for offline analysis**

Things often move quickly in a production environment, and you may not have quality time to spend with your application profiler. Instead, you can take a snapshot of everything in your Java environment and save it to look at later. You can do this in JConsole, and do it even better in VisualVM.

Start by navigating to the MBeans tab, where you'll open the com.sun.management node, followed by the HotSpotDiagnostic node. Now select Operations, and note the "dumpHeap" button that appears in the right-hand pane. If you pass dumpHeap a filename to dump to in the first ("String") input box, it will take a snapshot of the entire JVM heap and dump it to that file.

Later, you can use a variety of different commercial profilers to analyze the file, or use VisualVM to analyze the snapshot. (Remember that VisualVM is available in Java 6 and also as a stand-alone download.)

**5. JConsole isn't rocket science**

As a profiler utility, JConsole is nice, but other tools are nicer. Some profilers come with analysis add-ons or a slick user interface, and some track more data by default than JConsole does.

What's truly fascinating about JConsole is that the entire program is written in "plain old Java," meaning that any Java developer could write a utility like it. In fact, the JDK even includes an example of how to customize JConsole by creating a new plug-in for it (see Resources). VisualVM, being built on top of NetBeans, takes the plug-in concept much further.
If JConsole (or VisualVM, or any other tool) doesn't quite do what you want, or track what you're looking to track, or track in quite the way you want to track, you could write your own. And if Java code seems too cumbersome, there's always Groovy or JRuby or any of a dozen other JVM languages to help you get it done faster.

All you really need is a quick-and-dirty command-line tool connected via JMX, and you can track exactly the data you're interested in, exactly the way you want to.

In conclusion

Java performance monitoring doesn't end with JConsole or VisualVM — there's a whole raft of tools hiding out in the JDK that most developers don't know about. The next article in the series will dig into some experimental command-line tools that could help you dig out more of the performance data you need. Because these tools are generally focused on specific data, they're smaller and more lightweight than a complete profiler, and so they don't incur the same performance overhead.
Resources

Learn

• "5 things you didn't know about ...": Find out how much you don't know about the Java platform in this series dedicated to turning Java technology trivia into useful programming tips.

• "Monitoring and management using JMX" (Sun Microsystems): Learn more about using JMX and the JVM's built-in instrumentation tools to monitor and manage Java application performance.

• "Mustang JConsole" (Mandy Chung, Java.net, May 2008): A quick introduction to using the JConsole Plugin API to build custom plugins.

• "Acquiring JVM Runtime Information" (Dustin Marx, Dustin's Software Development Cogitations and Speculations, June 2009): Demonstrates the JDK's built-in monitoring and management tools, including JConsole and VisualVM.

• "Build your own profiler" (Andrew Wilcox, developerWorks, March 2006): The author shows how he used the Java 5 agent interface and AOP to build a custom profiler, the Java Interactive Profiler.

• IBM Monitoring and Diagnostic Tools for Java: Health Center is a low-overhead diagnostic tool for monitoring a running IBM Java Virtual Machine.

• The developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

• VisualVM is a visual tool integrating several commandline JDK tools and lightweight profiling capabilities.

Discuss

• Get involved in the My developerWorks community.

About the author

Ted Neward

Ted Neward is the principal of Neward & Associates, where he consults, mentors, teaches, and presents on Java, .NET, XML Services, and other platforms. He resides near Seattle, Washington.
Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.