

Java postmortem diagnostics, Part 1: Introduction to JSR 326 and Apache Kato

Learn how the Apache Kato project can help you analyse the causes of JVM failure

Skill Level: Intermediate

[Adam Pilkington \(apilkington@uk.ibm.com\)](mailto:apilkington@uk.ibm.com)
Software Engineer
IBM UK Ltd

05 May 2009

The artifacts produced when your Java™ application fails can help you analyse the root causes of the failure. A standard API to facilitate postmortem analysis is being developed by the Java Community process, and the Apache Kato project is under way to produce a reference implementation and tools for this API. This article, the first in a two-part series, introduces the Post mortem JVM Diagnostics API (JSR 326) and summarises the ways Kato will help you make good use of it. Part 2 will explore postmortem-diagnosis scenarios in greater depth.

This article is the first in a two-part series showing how you can analyse problems with your Java code using the artifacts produced when your application unexpectedly terminates. Java Specification Request (JSR) 326 is in progress to define an industry-standard API for this purpose. You'll learn how the associated Apache Kato project is working to deliver not only a reference implementation but also example tools that demonstrate how to use this API. With these tools, you'll be able to analyse problems such as out of memory errors, or simply browse through the available artifacts. This part gives you some background on JSR 326 and the Kato project, then describes some common postmortem-analysis scenarios and summarises the issues involved. Part 2 will examine each scenario in more detail, expand upon the problems to be solved, and show how you can use the API to solve them.

Live monitoring vs. postmortem analysis

Two broad categories define how you can try to diagnose problems with a running Java application. The first is *live monitoring*: you observe the behaviour of an in-flight virtual machine, for example monitoring the frequency at which your objects are being created or how long they live for before being garbage collected. A number of open source and commercial live-monitoring tools are available (see [Resources](#)). The second category is *postmortem analysis*. When a running JVM encounters a fatal error, the process it was running in terminates, and a number of artifacts are produced. Depending upon the error's severity, the JVM might create these artifacts as it shuts down, or the operating system creates them. Postmortem analysis is the act of analysing such artifacts after a JVM has "died."

Generating postmortem artifacts

Depending upon the JVM provider, it is not always true that the running process must be terminated in order to generate postmortem artifacts. Once you realise that in many cases it's the JVM itself that creates the artifacts, then it's not a particularly big leap to understand that this process can be triggered at any time. This opens up interesting diagnostic capabilities, such as instructing the JVM to create the required artifacts when a particular exception is thrown, and then using a suitable tool to diagnose the exception's root cause.

Live-monitoring tools enjoy the ability to connect to a number of standard APIs and interfaces — such as the JVM Tool Interface — that all JVM providers support because they are part of the Java language specification. Postmortem artifacts, in contrast, are normally opaque blobs of data. Typical examples include core files, heap dumps, and various other system dumps whose internal format and structure are usually known only to the vendors themselves, although some formats are public documents. This creates a general reluctance on the part of the people who write diagnostic tools to invest resources in producing tools that will work on only a limited number of JVM implementations. Fortunately, work is under way in the community to address this state of affairs.

JSR 326 and Apache Kato

JSR 326 is in progress to define a standard API for accessing the data contained in postmortem artifacts. It's intended to address the Java community's needs in four ways:

- Facilitate a vibrant tools ecosystem that will increase the number of available tools.

- Increase the quality and functionality of tools across different JVM providers.
- Allow developers to solve their own problems without being reliant on support from third parties.
- Address longer-term issues that can have an impact on the way that analysis is being performed now.

The JSR 326 Expert Group will initially define the problems and scenarios that the API will be used to solve. These user stories will then be used to shape the way that the API looks and should behave. Some of the more interesting design challenges that the Expert Group will be looking at include:

Future issues

One of the requirements of JSR 326 is that it address future issues or industry trends that will affect postmortem diagnostics. The first such issue is that existing postmortem artifacts will increase in size. As the computer industry moves towards 64-bit, multicore, multiprocessor servers, the amount of information that artifacts can contain is potentially far more than the available mass storage will allow. Even if that amount of data *can* be stored, subsequent issues are the ability to process that data or send it to someone who can. Another issue is how to handle situations when artifacts contain information from multiple JVMs that might come from different providers.

- Navigating through large data sets efficiently.
- Defining an exception-handling strategy and the rules governing the use of checked and unchecked exceptions.
- The logging mechanism to be used.
- Implementation optionality. When designing an API that is intended to interact with a large number of independently defined, and generated, postmortem artifacts, it is to be expected that there will be differences in the information present in each instance. This means that you must define a mechanism whereby implementations of the API can signal that the requested information is unavailable. This allows tools to be written that can gracefully deal with missing data.
- Backward compatibility between releases.

Whether or not you have a particular interest in postmortem diagnostics, the design fundamentals being discussed are of a more general nature and may well be of interest to you. Either way, see [Resources](#) for all the links necessary to be able to review, and possibly contribute to, JSR 326.

Future issues

One of the requirements of JSR 326 is that it address future issues or industry trends that will affect postmortem diagnostics. The first such issue is that existing postmortem artifacts will increase in size. As the computer industry moves toward 64-bit, multicore, multiprocessor servers, the amount of information that artifacts can contain is potentially far more than the available mass storage will allow. Even if that amount of data *can* be stored, subsequent issues are the ability to process that data or send it to someone who can. Another issue is how to handle situations when artifacts contain information from multiple JVMs that might come from different providers.

Apache Kato

Apache Kato is the open source project that will provide the reference implementation of JSR 326 for analysing artifacts produced by Sun's virtual machine. The code base is being seeded with IBM's Diagnostic Tool Framework for Java (DTFJ) (see [Resources](#)). Now an incubator project, Kato will become a fully fledged Apache Software Foundation project in due course.

Kato is more than just a reference implementation. It also contains:

- **Technology Compatibility Kit (TCK):** This test suite will be available for other JVM providers to check their level of compliance with JSR 326.
- **Documentation:** The API will be available to download and view as a Javadoc.
- **Wiki:** A wiki will cover all aspects of the project.
- **Example tools:** To demonstrate best practice when using the API, the project will provide a number of downloadable tools that can be used to solve real-world problems.

Diagnostic tools and scenarios for using them

Postmortem artifacts are analysed to determine a problem's underlying root cause, or to provide a view of a virtual machine's state. Next I'll look at the types of tools that can be used to carry out this analysis, then describe some common scenarios and summarise the issues involved.

Kato's diagnostic tools will exhibit some of the following characteristics:

- **Interactive or batch:** Will the tool be used in an interactive manner, such as a GUI, or would it be more likely as a command-line interface (CLI), which is typically found in batch processing?

- **Startup time or processing time:** Is it more important for the tool to start quickly, or to perform better when processing data?
- **Random or serial access:** Will the tool move at random through the postmortem artifact, or will it access the contents serially?
- **Lightweight or heavyweight:** Will the tool access a large proportion of the data available, or will it selectively interact with a smaller number of items?

Table 1 shows the tools the Apache Kato project intends to produce, categorised using the characteristics I just defined:

Table 1. Kato tools

Tool type or problem area	Interactive (GUI) or batch (CLI)	Startup time or processing time	Random or serial access	Lightweight or heavyweight
Java Debug Interface connector (for example, using Eclipse to debug an artifact)	Interactive	Startup	Random	Lightweight
Artifact explorers	Interactive	Startup	Random	Lightweight
Native-memory analysis	Batch	Processing	Serial	Heavyweight
Trend analysis	Batch	Startup	Random	Heavyweight
Artifact analysers that detect potential problems	Batch	Processing	Serial	Heavyweight
Java heap analyser	Interactive	Startup	Serial	Lightweight

Artifact explorers

Artifact explorers are intended to allow you to wander over the contents of the postmortem artifacts. A common implementation would be a GUI tool that has a tree on the left-hand side showing the artifact contents, with some panels on the right displaying more detail. You'd also be able to use the explorer to locate or search for known items in the artifact. The explorer should be lightweight because it will be dealing only with items in direct response to the user's input, rather than iterating through all of the items in the artifact. The explorer should also provide the following functionality:

- **Data type specification:** The ability to tell the UI to display the data at a given address as a specific data type — for example, *this address*

represents the start of a null-terminated string.

- **Data overlays:** The ability to override the byte values contained in a dump with user-defined values and have the UI work with the updated value. This means that you'd potentially be able to fix corrupt structures or lists by specifying the correct values. You should then be able to save and reload these overlays as required.
- **Automatic determination of artifact type.**
- **The ability to launch other tools from the explorer.** This can be simply starting the tool or launching with a set of command-line options that make sense within the current explorer context. For example, if the explorer shows a large heap occupancy, you'd launch an [artifact analyser](#) that starts a heap-occupancy analysis.

Native-memory analysis

In the past, Java developers didn't need to worry about native memory; in fact, it was often cited as an advantage that the JVM took care of these issues for you. However, the issue of native memory has been creeping back into the spotlight (see [Resources](#)). Initially, it was relatively easy to see when native memory was being used, because at some point you went through a Java Native Interface (JNI) call to access some functionality outside of the JVM. This could either be a direct call that you made yourself through JNI or that you coded to a specification that had well-defined links with JNI, such as the JVM Tool Interface (JVMTI). With Java 5 and the introduction of the `java.nio` package (and subpackages), native memory is used as the underlying storage for the buffers that this package provides. This sometimes unappreciated fact can easily lead to a situation in which a program has a large amount of free Java heap but throws an out-of-memory exception because it has run out of native memory. This means that native-memory analysis tools are becoming more important and need to be supported by JSR 326. Some example uses of such a tool include being able to:

- See the link between native memory and Java objects — that is, which Java objects are consuming the most native memory.
- List all the occurrences of JNI references in your application.
- Search the allocated memory for a known value.
- View a summary report showing how much native memory your application is using.

Artifact analysers

Artifact analysers process the entire contents of a given artifact looking for common errors such as deadlocked threads in a multithreaded application. They can build up a complete picture of the JVM to provide a statistical analysis of the JVM's state

when it terminated. This information can be used to show the number and types of Java objects, which can be useful if you need to determine why you ran out of space in the Java heap. Processing these large quantities of data presents its own set of challenges and will need to be taken account of in the final API design. One area that may be of particular interest is introducing support for an object query language that can be used to retrieve objects. The API might support the use of list objects to facilitate this approach.

Java Debug Interface (JDI) connector

A commonly generated postmortem artifact is a core file on the hard disk to which the entire contents of the process running your JVM is written. In some other programming languages, you can attach a debugger to this file and then use it to view the contents and possibly gain some insight as to the problem's cause. The Java language currently defines several interfaces and standards concerned with defining how to connect a debugger to a running JVM. The purpose of the postmortem JDI connector would be to allow you to connect to a core file, instead of a running JVM, using these standard interfaces. This would make it possible to debug artifacts in a standard Java debugger such as jdb, Eclipse, or NetBeans.

Help define the standard

Although work has already started on both the JSR and Apache incubator project, it is all at a very early stage. Both JSR 326 and the Apache Kato project are looking for people to either help define an API for analysing postmortem artifacts or contribute code for the reference implementation and tools. Plenty of opportunities are available to get involved and help shape the way the API looks or to contribute code. This is a rare chance to get involved at many different levels in a completely new open source project, as opposed to the more mature Apache Software Foundation projects.

Contributing to JSR 326

JSR 326 is all about solving real-world problems. If you have some, I urge you to share them with the Expert Group via the mailing list. (The discussions used to shape the API are being held on the mailing lists associated with the Apache Kato project; see [Resources](#)). That way, the group can make sure that the defined API will consider all uses and situations. It is vitally important for the Expert Group to have the best possible handle on this problem space. In particular, they would welcome feedback from:

- People who use postmortem diagnostics (such as dump viewers or heap explorers). The Expert Group would like to hear about the types of problems you need to solve and which tools you are using. It would also

be useful to know if the tool actually solved the problem, and if not, why not.

- Postmortem diagnostic tool providers (both commercial and open source applications). Feedback on your experiences with creating diagnostic tools for Java development and what you would like to see addressed in a standard API would be invaluable.

You are also free to apply to join the Expert Group, although you will be expected to have some knowledge of the area being defined by the specification.

Contributing to Apache Kato

As you now know, the Apache Kato project is more than just a reference implementation of JSR 326. It also includes best practice examples of using the API as well as a wiki and other documentation sources. If you want to contribute to this project, I recommend going to the Web site to see which items are currently under development and those areas that are specifically requesting help (see [Resources](#)).

What next?

I hope this article has given you an idea of the areas that are being covered in JSR 326 and Kato, and that some areas of interest extend beyond the projects' stated goals and are more general in nature. Some of the areas that need to be considered are:

- Manipulating large amounts of data, ensuring that good performance is achieved, and that the final solution performs as required. This will cover matters such as traversal and scanning of data.
- Developing caching strategies and their subsequent implementations.
- Ensuring performance using lightweight, lazily loaded objects.
- The possibility of incorporating an object query language.

The second article in this two-part series will explore various post mortem analysis scenarios and how they can be solved using the tools and technology demos provided as part of the Apache Kato project. It will also dive under the covers to see how the project is exploiting JSR 326 to achieve the desired results, while dealing with the fundamental issues discussed here in Part 1, such as large data set navigation.

Resources

Learn

- [JSR 326](#): Read the Post mortem JVM Diagnostics API specification proposal at the [Java Community Process](#) site.
- [Apache Kato](#): Visit the Kato Web site.
- [Java Platform Debugger Architecture](#): This overview includes information about the Java Debug Interface.
- *Thanks for the memory* (Andrew Hall, developerWorks, April 2009): Learn how the Java runtime uses native memory, what running out of native memory looks like, and how to debug a native `OutOfMemoryError` on [Windows®](#), [Linux®](#), and [AIX](#).
- [Diagnostic Tool Framework for Java](#): Learn more about DTFJ.
- [Java SE Monitoring and Management Guide](#): Find out how to use the monitoring tools shipped with Sun's JVM, including [JConsole](#).
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Java-Monitor](#): An online live monitoring service.
- [JXinsight](#): A commercial product for monitoring your Java application.
- [The IBM Monitoring and Diagnostic Tools for Java - Health Center](#): A lightweight monitoring tool that allows running Java VMs to be observed and health-checked. The Health Center enables insight into general system health, application activity, and garbage-collection activity.
- [VisualVM](#): A visual tool integrating several command-line tools and lightweight profiling capabilities.

Discuss

- View the [JSR 326 mailing archives](#) and [subscribe](#) to the mailing list.
- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Adam Pilkington

Adam Pilkington is a software engineer at the IBM Java Technology Centre working on JSR 326 and the Apache Kato project. Before joining IBM in 2006, he was a J2EE technical architect for a large financial-services organisation in the United Kingdom. He holds a degree in mathematics and computer science.

Trademarks

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