Memory leak detection and analysis in WebSphere Application Server: Part 1: Overview of memory leaks

Indrajit Poddar
Robbie John Minshall

June 28, 2006

This article provides a high level overview of the techniques available in IBM® WebSphere® Application Server to address Java™ heap memory leaks.

Introduction

The cost of memory leaks is significant and is often associated directly with production down time, or a slipped deployment schedule. Unfortunately, the costs of appropriate test solutions are also significant, and customers are often unwilling -- or unable -- to invest the resources necessary.

To be clear, the best approach for solving memory leaks is to detect and resolve them in test. Ideally, there should be a testing schedule, a test environment identical to the production environment that is able to drive representative and anomaly workloads, and technical resources with appropriate skill sets dedicated to system testing. This is the best way to ensure, as much as possible, a clean transition to production. However, designing and provisioning such an environment, along with the associated cultural changes, is not the focus of this article.

There are four common categories of memory usage problems in Java:

- Java heap memory leaks
- Heap fragmentation
- Insufficient memory resources
- Native memory leaks.

WebSphere Application Server has introduced two complementary technologies to help customers address Java heap memory leaks (to a certain extent these tools also address problems caused by insufficient memory resources; the similarities of these two problems is not addressed in this article).

In general, a Java heap memory leak results when an application unintentionally (due to program logic error) holds on to references to objects that are no longer required. These unintentional object references prevent the built-in Java garbage collection mechanism from freeing the memory used by these objects. Common causes for these memory leaks are:
• Insertion without deletion into collections
• Unbounded caches
• Un-invoked listener methods
• Infinite loops
• Too many session objects
• Poorly written custom data structures.

Memory usage problems due to insufficient memory resources can be caused by configuration issues or system capacity issues. For example, the maximum heap size allowable for the Java virtual machine may be configured using the -Xmx parameter at a value which is too low to accommodate the total number of user sessions in memory. Alternatively, the system may have too little physical memory to accommodate the current workload. Native memory leaks are memory leaks in non-Java code; for example, in Type-II JDBC drivers, or fragmentation in the non-heap segment of the Java process's address space.

This is an introductory article written for Java developers, system administrators, and problem determination consultants working with applications deployed on IBM WebSphere Application Server.

Problems with existing techniques for detecting root causes of memory leaks

There are a number of problems that cause memory leaks to be particularly troublesome for System Administrators:

1. Traditional Java memory leak root cause detection techniques have significant performance burdens and may not be suited for use in production environments. These techniques usually include heap dump analysis, attaching JVMPI (Java Virtual Machine Profiler Interface) or JVMTI (Java Virtual Machine Tools Interface) agents, or the use of byte code insertion to track insertions and deletions into collections.

2. Traditional redundancy methods such as clustering can only help to a certain extent. Memory Leaks will propagate throughout cluster members. An affected application server's response rate slows workload management techniques. This can cause requests to be routed to healthier servers and result in coordinated application server crashes.

A typical analysis solution should attempt to move the application onto an isolated test environment where the problem can be recreated and analysis may be performed without impacting the production servers. The cost of associated memory leaks is magnified by the difficulty of reproduction in these test environments.

The cause of these issues is that traditional techniques attempt to perform both detection and analysis.

The WebSphere solution

WebSphere Application Server V6.0.2 and higher provides a two-stage solution that separates the problem of detection from analysis.
The first stage is a lightweight memory leak detection mechanism running within the production WebSphere Application Server runtime. This lightweight detection technique uses inexpensive, universally available Java heap usage statistics to monitor memory usage trends, and provides early notification of memory leaks. This enables the administrator time to prepare appropriate back up solutions and to analyze the root cause of the problem in an offline manner without the expensive and frustrating problems associated with reproduction in test environments.

The second stage of the solution is an offline tool: Memory Dump Diagnostic for Java (MDD4J) which analyzes heap dumps outside the production application server. This is a heavyweight offline memory leak analysis tool that incorporates multiple existing heap dump analysis tools into a single user interface.

To bridge the gap between detection and analysis, an automated heap dump generation facility has been provided for WebSphere Application Server running on IBM JDKs. Upon detection of a memory leak pattern, this facility will generate multiple heap dumps that have been coordinated with sufficient memory leakage to facilitate comparative analysis using MDD4J. In addition, the IBM JDK is configured to generate a heap dump automatically if an OutOfMemoryError is detected. Administrators should set up load balancing, or generate heap dumps at low usage times, to avoid short periods of poor performance.

**Memory leak detection**

Lightweight memory leak detection is achieved by monitoring downward trends in free memory.

Leaks can be very fast or incredibly slow, so memory usage trends of both short and long intervals are analyzed. In addition, downward trends in approximated memory usage after a garbage collection cycle are analyzed to detect situations where the average free memory after the garbage collection cycle is below certain thresholds. Such a situation is either a sign of a memory leak or of running an application on an application server with too few resources. This lightweight memory leak detection facility is available on all versions of WebSphere Application Server starting with Version 6.0, for all platforms.

In addition, specifically for the iSeries® platform, WebSphere Application Server on iSeries includes additional functions to detect if the Java heap size is going to be expanded onto DASD, and issues an alert notifying the administrator that this is going to occur, and that there is either:

- Too small an effective memory pool size
- Too few resources
- A memory leak.

zSeries® support includes only single servant topologies in Version 6.0.2, but is extended in Version 6.1 to include multiple servant topologies. Single servant topologies limits the scope of memory leak detection in V6.0.2 to problem determination or test environments.

Figure 1 shows a sample notification generated by the memory leak detection feature. This notification is sent out via JMX, is displayed in the administrative console, and is recorded in the server logs.
The automated heap dump generation facility (available only on IBM JDKs) generates a heap dump after evidence of the memory leak is apparent, but before the application server crashes due to an OutOfMemoryError. This facility generates a second heap dump after enough memory leakage. The two heap dumps facilitate comparative analysis using MDD4J.

Automated heap dump generation can be enabled by default or can be initiated at the appropriate time using an MBean operation.

WebSphere Extended Deployment

Although lightweight memory leak detection is available and designed to work out of the box in WebSphere Application Server and WebSphere Application Server Network Deployment, it is also completely configurable and designed to interact with advanced autonomic managers or custom JMX clients. WebSphere Extended Deployment is an example of such a relationship.

WebSphere Extended Deployment provides a number of policies that configure memory leak detection. One policy might react to a memory leak notification by taking multiple heap dumps (using workload management to maintain the performance of the application) for analysis. Another policy might simply monitor when the application server's memory levels are critical in order to restart the application server prior to it being crippled.

Memory leak analysis

Once a memory leak has been detected and heap dumps have been generated, they can be transferred outside the production server and into a problem determination machine for analysis.

Memory Dump Diagnostic for Java (MDD4J) is an offline heap dump analysis tool that aids in the process of determining the root cause of the memory leak. The analysis mechanism identifies suspected leaking data structure classes and packages. This identification enables the system administrator to narrow down the root cause of the memory leak to a few component applications.

WebSphere Application Server provides containers for hosting J2EE applications from third parties. As the Java stack evolves with time, more and more layers of abstraction and componentization are being added on to the basic Java application and to the WebSphere Application Server stack. This poses an enormous challenge for problem determination when a memory leak occurs. To a system administrator encountering a memory leak, a WebSphere Application Server hosting a number of third party applications is like a black box. In such a situation, the first step is to narrow down the root cause of the memory leak to one or a few components. The root cause often lies in one faulty component application. With the aid of the analysis results from the Memory Dump Diagnostic for Java tool, a system administrator can now identify a faulty component much faster -- and without any support from IBM.
Once the faulty component is identified, the system administrator can approach a developer who might be able to replicate this problem in a smaller environment and use a debugger or specific trace statements in logging to identify the faulty source code method, and make the necessary changes either to the application code, or to the configuration to resolve the memory leak. Sometimes, knowing the faulty component or leaking object is enough for us to identify some common configuration problems. For example, finding lots of HTTP session objects would lead us to look at the HTTP session timeout configuration.

Two main types of analysis functions are provided by this tool:

- Single dump analysis is most commonly used with memory dumps that are triggered automatically by the IBM Developer Kit, Java Technology Edition with OutOfMemoryExceptions exceptions. This type of analysis uses a heuristic process to identify suspected data structures that have a container object with a large number of children objects. This heuristic is quite effective in detecting leaking Java collection objects which use an array internally to store the contained objects. This heuristic has been found to be effective in a large number of memory leak cases handled in IBM Support.

- Comparative analysis is used for comparing two memory dumps (the primary dump and the baseline dump) taken during a single run of a memory leaking application (that is, while the free Java heap memory is dropping). Comparative analysis is well suited to be used in conjunction with lightweight memory leak detection. For comparative analysis, the primary dump refers to the dump taken after the memory leak has progressed considerably (consuming a large amount of the maximum configured heap size). The baseline dump refers to the heap dump taken early on, when the heap has not yet been consumed significantly due to the memory leak. The greater the heap consumption in between the dumps, the better the analysis result.

The comparative analysis technique identifies a set of large sized data structures experiencing significant growth in the number of instances of constituent data types. Data structures are grouped together in each dump and then matched and compared across the primary and baseline dump to identify suspected data structures experiencing large amount of growth. This technique differs from basic heap dump differencing techniques available in many analysis tools in the market today by virtue of the fact that this technique identifies suspected leaking data structures at a higher level of granularity, as opposed to identifying suspected leaking data types (such as, strings at a much lower level of granularity). For example, MDD4J will tell you that a particular container, such as specific EJB object, is leaking a large number of strings, rather than simply tell you that a large number of strings are being leaked from some unknown source. Identification of suspected data structures aid in a better understanding of the root cause of a memory leak. A sample data structure is described in a tree view in Figure 3, where it is possible to see that not only string objects are leaking, but they are being referenced from a HashSet in a class by the name of MyClass.

The analysis results are shown in an interactive Web-based user interface with the following features:

- Lists a summary of analysis results, heap contents, size, and growth.
• Lists suspected data structures, data types, and packages contributing to the growth in heap usage for comparative analysis, and to the large size of the heap for single dump analysis.
• Ownership context view shows ownership relationship between major contributors to the footprint and significant constituent data types of the summarized set of major footprint contributors.
• Browsing capabilities in an interactive tree view displays the relevant portion of the heap dump, showing all incoming (only one reference shown in the tree, remaining ones shown separately) and outgoing references for all suspected container objects in the heap and child objects of the container object, sorted according to reach size.
• Navigation capabilities from suspect lists to the ownership context, and contents view to the browse view.
• Tabulated views of all the objects and data types in the memory dump with filters and sorted columns.

The MDD4J tool combines best of breed features from many existing tools. The comparative analysis techniques are based on the Leakbot research project (see Related topics). The single dump analysis feature uses analysis techniques which can also be found in the HeapAnalyzer tool downloadable from alphaWorks (see Related topics). The tabulated views are based on features from a command line tool, HeapRoots (see Related topics).

With WebSphere Application Server V6.1, the Memory Dump Diagnostic (Version 1.0) for Java tool is packaged with the IBM Support Assistant tool (Version 3.0), a standalone tool for system administrators that can also be installed separately (see Related topics) from WebSphere Application Server on any machine to which heap dumps can be transferred for offline analysis. (See Related topics for information on how to launch MDD4J Version 1.0 from IBM Support Assistant Version 3.0.)

After downloading the IBM Support Assistant, the Memory Dump Diagnostic for Java can be added separately by using the Updater mechanism: in IBM Support Assistant, select New Products and Tools => Common Component and Tools in the Updater view (also make sure that the WebSphere Application Server V6.0 or V5.0 product plug-ins are also installed).

This new version of MDD4J (Version 1.0) features a number of scalability-related improvements, and support for 64-bit heap dumps over the previous version (Version 0.97a). (This previous version of MDD4J can be downloaded as a technology preview developerWorks; see Related topics.)

The following example shows memory leak analysis results for a simple memory leaking Java application. This sample application leaks string objects into a static HashMap. Lightweight leak detection is able to identify the presence of the leak early on and automatically generate the appropriate heap dumps. These dumps are then analyzed offline using comparative analysis in Memory Dump Diagnostic for Java Version 1.0. The analysis results showing the memory leak suspects are shown in Figures 2, 3, and 4.
Figure 2. Memory leak suspects in the Memory Dump Diagnostic for Java analysis result

Figure 3. Browse suspected data structure in Memory Dump Diagnostic for Java
Conclusion

With WebSphere Application Server V6.1, system administrators are able to receive early notification of memory leaks in their production environments without the use of byte code insertion, or any attached agents. Lightweight memory leak detection has been designed to have minimal performance impact, and provides the ability to automatically generate heap dumps at appropriate times to ensure accurate analysis results.

Heap dumps generated manually, or with the help of memory leak detection, can be analyzed offline using Memory Dump Diagnostic for Java (MDD4J). This tool can help system administrators assign issues to the appropriate component where analysis results can easily be reproduced by developers on their machines. MDD4J provides a tool for developers and problem determination consultants to identify leaking candidates, browse the heap and ownership chains, to determine what data structure is leaking within their code.

This technology significantly helps in the diagnosis of Java heap memory leak detection and analysis, and can be used either in conjunction with good system test procedures or with production systems.

Acknowledgements

The authors would like to thank: Daniel Julin and Stan Cox for reviewing the paper; Nick Mitchell, lead researcher on the LeakBot project, for his leadership and innovation; Mark T Schleusner for his assistance and collaboration in developing MDD4J.
Related topics

- Java theory and practice -- A brief history of garbage collection: How does garbage collection work?, Brian Goetz
- Java theory and practice -- Garbage collection in the HotSpot JVM: Generational and concurrent garbage collection, Brian Goetz
- WebSphere Application Server Information Center
- IBM Java technology diagnosis documentation
- Launch MDD4J from IBM Support Assistant version 3.0 packaged with WebSphere V6.1
- LeakBot, a memory leak analysis tool.

© Copyright IBM Corporation 2006

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)