SAP Memory Analyzer: Multi-Gigabyte Java Heap Dumps Are Nothing to Fear Now

Applies to:
Java development in general

Summary
SAP has developed a tool for Java memory analysis which is helpful to all Java developers in general, not only to those building Software on top of SAP products like SAP NetWeaver for Java. It was presented at JavaOne 2007 in a technical session and is now available for free download on SDN. Please read this article to find out why we think this tool makes a difference and is worth working with. More technical and problem-oriented articles will follow soon.

Author(s): Vedran Lerenc
Company: SAP AG
Created on: 21 January 2007

Author Bio
Vedran is a Development Architect in the Java Server Technology group of SAP AG. He started to work professionally in 1994 with IBM at the European Networking Research Center in Heidelberg. He left and co-founded MetaKey Inc. in 1998 developing an OODBMS. In 2000 he joined SAP AG in Palo Alto, later in Walldorf where he is currently researching and developing tools for memory and thread analysis in Java.
Introduction

To put it short, the SAP Memory Analyzer is a tool for Java memory analysis or – more precisely – an Eclipse-based application which allows you to inspect a Java HPROF binary heap dump very efficiently and effectively, because it is both, fast in execution and helpful in analysis.

The tool was developed to analyze real productive heap dumps which tend to get enormous in size with hundreds of millions of objects. High Performance, low resource consumption and especially its analysis techniques make it a unique tool, powerful also to small application heap dumps. Please see our Java Memory Analysis page on the SDN Wiki for more information, performance numbers and a complete feature list.

Use Case

Java VMs of Sun, HP, SAP and some other vendors can be configured to write heap dumps on the first thrown OutOfMemoryError (OOM). Such a heap dump basically contains all the Java Objects including their field values and references among them and is helpful to understand the state of the application at the time the Java VM run out of memory.

You can open those heap dumps with the SAP Memory Analyzer and inspect them in detail. The SAP Memory Analyzer won’t tell you when and by whom the objects were allocated, because this information is not present in the heap dump. Also, you will only see the live objects, i.e. you won’t see the objects which have been allocated and garbage collected in the past. Again, this information is not present in the heap dump. What you will see are all the Java Objects which couldn’t be garbage collected at the time the heap dump was written, along with the information who keeps them alive, their size and more.

Procedure

The easiest way to get a heap dump is to get one on OutOfMemoryError (OOM). This feature is available since version 1.4.2_12, 5.0_7 and 6.0 and can be switched on with the following VM parameter:

-XX:+HeapDumpOnOutOfMemoryError

There are some more means to get a heap dump on demand, but they depend on the platform/version of the Java VM. Please have a look at our Wiki for more information on this topic.

Our Wiki also gives you the download link to the SAP Memory Analyzer and information on how to install it. You will find the download link at the end of this article, too. Even though we offer only a Windows 32 bit installer, you can read on the Wiki on how to set up the tool for your platform, as long as it is supported by the Eclipse platform.

When you have a heap dump and open it for the first time in the SAP Memory Analyzer, the tool will parse the heap dump. Parsing is in our case the process of building up index files for the most important properties of the heap dump, e.g. the size of each and every object, where it is found in the heap dump, which objects
reference it or which objects are referenced by it. This happens only once. When you reopen the heap dump, it opens instantly. Other tools read/parses the heap dump again and again when you reopen it.

Parsing is a relatively expensive operation. We optimized the parsing a lot, but for large heap dumps above 2 GB you will usually need a bigger box. However, as written above, this happens only once. If you have your index files, you can analyze even extremely large heap dumps on a smaller, let’s say 32 bit machine and hand the heap dump along with the index files around to other users, e.g. from support to development or from one developer to another.

After the parsing process the real intellectual work for the user starts: The analysis of the heap dump. The tool supports the user with a rich set of features, allowing also users not familiar with the source code to identify the problems.

Features

After opening a heap dump you will see some basic numbers about the heap dump itself and a basic class histogram on the first page. A class histogram lists all the classes which were found in the heap dump and the number and size of instances of them.

This is nothing special as all tools more or less offer such a view. You may jump to the list of objects, inspect the objects on the left hand side in the object inspector, but this is again boilerplate functionality – heap walking.
Only in less than 5% of the cases this view tells you what exactly is wrong, but it gives you an overview. E.g. you might get irritated if you see too many instances of java.lang.String, let's say 10 million of them, but it doesn't tell you much more. What can you do from there next? Follow each reference to each instance manually to detect a pattern? That would be pretty time consuming if not useless.

The SAP Memory Analyzer did something during parsing which will help you a lot during analysis. It transformed the graph of objects into a tree of objects – a dominator tree. This dominator tree has a very important property: Each object in the tree dominates its children, meaning, it keeps them alive. All children are alive because of this single object in the tree above them.

Dominator trees point immediately to the biggest distinct object sets. You can explore their content as you like. You can also do it vice versa, take the instances of java.lang.String I have mentioned above and look up their dominators and group them by class. An aggregation pattern, i.e. systematic behavior behind your memory will immediately become visible in the dominators. You can even filter out objects you are not interested in, e.g. to know that some or many Strings were hold in HashMaps is not interesting, but skipping those collection references or generally instances of classes from the java.* packages in the dominator chain until you reach an instance of a class of your company or your customers tells you a lot more.

It is important to note here that this approach is not to be confused with heap walking. Usually if you take an object and look what it references you will see that it references half of the heap through some reference chains, e.g. an object references implicitly its class which references implicitly its class loader which references the classes defined through it which reference static fields and so on. In reality following the object references doesn't help you for a set of objects. With the dominator tree, however, this problem is
solved elegantly. In a tree you have only one parent and in the dominator tree this parent is the single object which keeps all its children alive.

Dominators have some more very helpful properties which we have exploited in the tool, but we won’t go into more details in this article and conclude this brief feature tour with the most common final question in analysis: Who holds your problematic chunk of memory alive? Which key object or thread is it? Independent of the means you have used to spot the memory problem, through the dominator tree as you have seen above or through some different means, e.g. through our query language or our textual console, you are interested to see the real reference chain from the roots, e.g. from the thread(s). This can be done by a view which shows you all the paths from the roots to your object or object set of interest.

I picked those three features/screenshots just to highlight the basic approach in analysis, which is 1.) about getting an overview, 2.) finding your problematic memory and 3.) analyzing who keeps it alive, but there is much more to explore in the tool than this.

We will cover more functionality, tips and hints in further articles, blogs and posts.

**Leak or Footprint**

Having read the above you might think that memory leaks are the problem this tool addresses primarily, but in reality memory leaks are comparatively easy to detect and fix if the leak can grow over time. However, in a Java application server with hundreds or thousands of deployed applications memory footprint problems are far more threatening, as all deployed applications together burden the server. Wasted memory may be
hidden everywhere and the SAP Memory Analyzer helps to uncover most of them, e.g. by views showing the memory consumption per class loader which is usually the equivalent of a deployed application in an application server.

**Synopsis**

SAP Memory Analyzer supports Java developers in general with new analysis techniques, as well as SAP customers with specific SAP NetWeaver CE views. It helps to analyze small Java application heap dumps, as well as extremely large Java application stack heap dumps like those from productive JEE systems.

It is now available for free download on SDN. SAP doesn’t charge money for it, nor limits its functionality, but doesn’t offer support for it, except if you are a SAP NetWeaver CE customer.

However, we like to receive your feedback and will actively participate in discussions on the Internet. Please visit our [Wiki](https://wiki.sdn.sap.com/wiki/x/k2w) for more details.

**Related Content**

- Wiki Page: [https://wiki.sdn.sap.com/wiki/x/k2w](https://wiki.sdn.sap.com/wiki/x/k2w)
- Slides from our JavaOne 2007 Technical Session: [https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/00ca7f0d-8ee6-2910-5d82-fc3e8dd25300](https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/00ca7f0d-8ee6-2910-5d82-fc3e8dd25300)