Welcome to the Machine

Die VM für Entwickler und hochverfügbare Applikationsserver

Ralf Schmelter, Michael Wintergerst

Die SAP JVM ist eine zertifizierte Java Virtual Machine (JVM) und beinhaltet ein komplettes Java Development Kit (JDK). Sie wurde unter dem Gesichtspunkt der Hochverfügbarkeit, Zuverlässigkeit und Plattformvielfalt entwickelt. Nahezu alle Java-basierten SAP-Produkte (wie der SAP NetWeaver Application Server Java oder die SAP HANA Cloud-Plattform) nutzen die SAP JVM als Fundament ihrer Laufzeitumgebung.

Die SAP JVM [SAPJVM] ist nicht nur „just another Java Virtual Machine", sondern beinhaltet eine große Anzahl von Hilfsmitteln und Funktionalitäten für Support-Mitarbeiter und Entwickler, die es in dieser Form in keiner anderen JVM gibt – dazu gehören Debugging on Demand, Profiling und Speicheranalyse sowie die verbesserten Wartungsmöglichkeiten. Darüber hinaus ist die SAP JVM auf fünfzehn verschiedenen Plattformen verfügbar. Da das Verhalten und die Hilfsmittel der SAP JVM auf allen Plattformen identisch sind, wird der Wartungsaufwand auf Entwicklerseite erheblich reduziert.

Plattformen

Die SAP JVM basiert auf der Technologie der Java HotSpot VM von Oracle und unterstützt somit alle Betriebssysteme und CPU-Architekturen, die auch von der Java HotSpot VM abgedeckt werden. Darunter fallen Windows, Linux, Solaris und Mac OS X auf Betriebssystemseite sowie die entsprechenden CPU-Architekturen Intel (x86_32, x86_64) und SPARC. Darüber hinaus ist die SAP JVM auf HP/UX und den IBM-Plattformen AIX und IBM i verfügbar. Die unterstützten CPU-Architekturen wurden um PowerPC, z/Arch, PA-RISC und Itanium erweitert (s. Tabelle 1).

	x86_32	x86_64	IA64	SPARC	z/Arch	Power	PA-RISC
Windows	✓	✓	✓				
Linux	✓	✓	✓		✓	✓	
Solaris		✓		✓			
Mac OS X		✓					
HP/UX			✓				✓
AIX					✓		
IBM i					✓		

Tabelle 1: Unterstützte Architekturen

SAP JVM-Technologie im OpenJDK

Mitte 2011 ist SAP dem OpenJDK-Projekt [OpenJDK] zur Entwicklung einer Java Standard Edition auf Open-Source-Basis beigetreten. Im Frühjahr 2012 hat SAP zusammen mit IBM ein Projekt [PowerPC/AIXPort] gestartet, um eine OpenJDK-Implementierung für AIX und PowerLinux zur Verfügung zu stellen. Mittlerweile wurde eine Java 7-Zertifizierung für diese Plattformen erfolgreich durchgeführt, und man arbeitet momentan an einer Java 8-Implementierung.

Vorteile der SAP JVM

Dadurch, dass die SAP ihren Kunden eine JVM für all ihre unterstützten Plattformen bieten kann, ist der Administrationsaufwand für SAP-Kunden deutlich reduziert worden. Denn die SAP JVM verhält sich auf allen Plattformen identisch. Es gibt nahezu keine Unterschiede zwischen den JVM-Konfigurationsparametern, dem Laufzeitverhalten (bspw. Garbage-Collection-Algorithmen, XML-Parser, Error/Trace-Meldungen) und den angebotenen Analysewerkzeugen. Insbesondere in einer heterogenen IT-Landschaft mit einer Vielzahl unterschiedlicher Hardware- und Betriebssystemkonfigurationen bietet die SAP JVM daher Administratoren einen enormen Vorteil.

Darüber hinaus beinhaltet die SAP JVM gerade im Bereich der Supportbarkeit eine Menge zusätzlicher Funktionalität und Werkzeuge für Administratoren und Entwickler, wie Tabelle 2 zeigt.

SAP JVM - Supportbarkeit	Beschreibung
Debugging on Demand	Startet den Debugging-Modus zu einem beliebigen Zeitpunkt ohne Durchstarten der JVM
Monitoring-Tool „jvmmon"	Administrationswerkzeug für den Zugriff auf laufende SAP JVM-Prozesse
SAP JVM Profiler	Grafisches Analysewerkzeug für das Laufzeitverhalten von Java-Applikationen
SAP JVM Debugger (in Kürze verfügbar)	Debugging-Lösung inklusive Java-Disassembler und Java-Decompiler – insbesondere für das Debugging von „remote" Java-Applikationen
Monitoring API	Java-API für den lokalen und „remote" Zugriff auf SAP JVM-Prozesse
Erweiterte Thread- und Heap-Dumps	Detaillierte Ressourcen-Informationen (CPU, Speicher, I/O) pro Thread. Heap-Dumps beinhalten zusätzlich zu den Objekt-Strukturen Informationen zum Füllstand der permanent Generation sowie Object Ids zum Vergleich
Detail-Informationen zu kritischen Java-Exceptions	Erweiterte Kontext-Informationen im Fall von `NullPointerException`, `ClassCastException`, `NoClassDefFoundError` und `OutOfMemoryError`

Tabelle 2: Werkzeuge für Administratoren und Entwickler

Download: SAP HANA Cloud

Zum lokalen Entwickeln in der SAP HANA Cloud [Hana] wird die SAP JVM zum freien Download zur Verfügung gestellt: `https://tools.hana.ondemand.com/#cloud`. Die entsprechenden Ent-

wicklerwerkzeuge wie den SAP-JVM-Profiler erhalten Sie über die SAP-Eclipse-Update-Site: `https://tools.hana.ondemand.com/juno`. Dokumentationen zum SAP JVM Profiler finden Sie integriert in der Eclipse-Hilfe.

Debugging on Demand

Jede JVM muss eine Infrastruktur anbieten, um Java-Applikationen zu debuggen. Mit Hilfe der folgenden JVM-Option kann eine JVM im Debugging-Modus gestartet werden:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n
```

Die JVM öffnet daraufhin einen TCP-Port und mittels eines Debuggers (z. B. Eclipse, NetBeans, jdb) kann sich der Entwickler mit der JVM verbinden und den Zustand der Java-Applikation betrachten. Der Debugging-Modus kann direkten, negativen Einfluss auf die Performance einer JVM haben, denn die JVM läuft dann teilweise interpretiert. Daher sollte der Debugging-Modus nicht in produktiven Umgebung „per default" eingeschaltet werden. Der Entwickler kann aber die JVM auch erst zur Laufzeit, „on the fly", in den Debugging-Modus versetzen.

Diese Funktionalität lässt sich hervorragend nutzen, um Fehlersituationen einer Java-Applikation zu analysieren. Lässt sich beispielsweise ein Problem nicht auf einfache Weise reproduzieren, versetzt man die JVM dann in den Debugging-Modus, wenn das Problem auftritt. Ein typisches Szenario ist die Analyse von Deadlock-Situationen im Bereich von Applikationsservern. Im Allgemeinen ist eine Reproduzierbarkeit solcher Probleme nur schwer möglich.

Die SAP JVM bietet zwei Möglichkeiten, um in den Debugging-Modus zu wechseln. Es existiert ein Java-API, das insbesondere Tool-Entwickler nutzen können, um eine SAP JVM zu debuggen. Daneben kommt die SAP JVM mit einem eigenen Monitoring-Tool „jvmmon", das die entsprechende Funktionalität bietet. Es wird in beiden Fällen ein TCP-Port geöffnet, an den sich ein Standard-Debugger verbinden kann.

Monitoring-Tool jvmmon

Die SAP JVM beinhaltet ein einfaches Monitoring-Tool namens „jvmmon". Es gibt zwei Varianten: eine Kommandozeilen-Version und eine Swing-basierte Oberfläche. Das Tool bietet eine Übersicht und Zugriff auf alle gestarteten SAP JVM-Prozesse auf einem Host. Für einen ausgewählten SAP JVM-Prozess erhält man grundlegende Informationen wie die Version der JVM, aktuelle Konfigurationseinstellungen und JVM-Parameter. Darüber hinaus werden dem Entwickler wichtige Status-Informationen zur Garbage Collection (Anzahl GCs, wie viel Zeit in der GC verbracht wurde) und zur aktuellen Heap-Belegung angezeigt.

Daneben bietet das Tool Entwicklern die Möglichkeit, Befehle auf die einzelnen SAP JVM-Prozesse anzuwenden. Beispielsweise können sie eine SAP JVM in den Debugging-Modus versetzen, Thread/Heap-Dumps triggern, sich Detail-Informationen zu den aufgetretenen GCs geben lassen oder sich eine Übersicht der geladenen Klassen und Instanzen ansehen. Abbildung 1 zeigt einen Ausschnitt der GUI-Variante und die entsprechende Funktionalität zum Einschalten des Debugging-Modus.

Generell gibt es zwei Modi für das Ausführen der Kommandos: „Print" und „Dump". Sollen alle Informationen wie Thread-Dumps, Klassen-Statistiken und GC-Informationen in Dateien gespeichert werden, können die entsprechenden Dump-Kommandos verwendet werden. Um sich die Daten direkt im GUI anzusehen, können die „Print"-Kommandos Anwendung finden.

Neben der lokalen Monitorbarkeit von SAP JVM-Prozessen besteht auch die Möglichkeit, Zugriff auf remote Instanzen zu bekommen. Der sogenannte jvmmond-Prozess muss auf dem System, das überwacht werden soll, gestartet werden. Dieser Prozess öffnet einen TCP-Port, an dem sich ein lokal laufender jvmmon-Prozess verbinden kann. Mittels dieser Funktionalität können beispielsweise SAP JVM-Prozesse, die auf AIX oder HP/UX laufen, von einem lokalen Entwickler-PC unter Windows oder Linux überwacht werden.

Exceptions

Exceptions sind einer der wichtigsten Eckpfeiler der Supportbarkeit in Java. Im Falle eines Fehlers bekommt man zum einen den Stacktrace zur Lokalisierung des Fehlers, zum anderen kann die Message der Exception weitere Hinweise zur Lokalisierung des Fehlers enthalten. Durch die Verbesserung der von der VM geworfenen Exceptions lässt sich also auch die Supportbarkeit der auf ihr laufenden Java-Anwendungen steigern. An einigen Beispielen sollen nun solche Optimierungen vorgestellt werden.

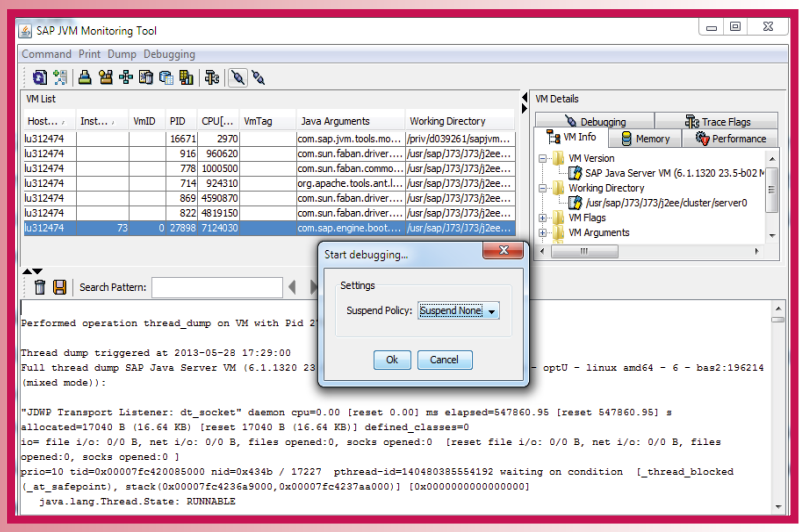

Abb. 1: SAP JVM Monitoring Tool: Start debugging

NullPointerException

Eine NullPointerException (kurz NPE genannt) ist eine der am häufigsten auftretenden Exceptions. Vielfach kann schon anhand des Stacktraces festgestellt werden, warum die Exception aufgetreten ist. Andererseits gibt es Fälle, in denen dies nicht so einfach zu erkennen ist. Dies liegt daran, dass eine NPE an vielen Stellen im Java-Code auftreten kann und die Angabe der Zeilennummer deshalb allein nicht ausreichend sein muss. Als Beispiel dient folgende Methode:

```
public int hashCode() {
  return field1.getId() ^ field2.getUniqueId();
}
```

Hier können mehrere NPEs auftreten:

- `field1` ist `null` und `getId()` wird dann auf `null` aufgerufen.

▼ `field2` ist `null` und `getUniqueId()` wird dann auf `null` aufgerufen. Zusätzlich zu diesen offensichtlichen Möglichkeiten ergeben sich noch weitere Problemfälle. Wenn `getId()` oder `getUniqueId()` beispielsweise `Integer`-Objekte zurückgeben, dann können durch Unboxing auch NPEs geworfen werden (an zwei weiteren Stellen). Aus diesem Grunde versucht die SAP JVM, den Grund für die NPE in der Message der Exception genauer zu beschreiben. Die zugrunde liegende Idee ist, dass durch Angabe des Bytecode-Index (bci) die Ursache der NPE genau beschrieben werden kann, da auf Bytecode-Ebene nur jeweils ein Grund für das Auftreten einer NPE existiert. Schauen wir uns einmal das Disassembly der Methode an:

```
0: aload_0
1: getfield #21; //Field field1:LNPEExample;
4: invokespecial #42; //Method getId:()Ljava/lang/Integer;
7: invokevirtual #46; //Method java/lang/Integer.intValue:()I
10: aload_0
11: getfield #23; //Field field2:LNPEExample;
14: invokespecial #49; //Method getUniqueId:()Ljava/lang/Integer;
17: invokevirtual #46; //Method java/lang/Integer.intValue:()I
20: ixor
21: ireturn
```

An den bcis 4, 7, 14 und 17 können NPEs auftreten. Da die SAP JVM den Stacktrace durch Methoden/bci-Paare speichert, ist der bci der NPE automatisch im Exception-Objekt gespeichert. Aus diesen Daten lässt sich dann eine detaillierte Message für die NPE generieren.

Betrachten wir dies am Beispiel einer NPE, die entsteht, wenn `getUniqueId()` den Wert `null` zurückliefert. Der bci der Exception ist dann 17. Durch den Bytecode bei bci 17 wissen wir schon, dass die NPE durch das Aufrufen der Methode `intValue()` auf einem `null`-Wert entsteht. Wie wir sehen, schränkt diese Information die potenziellen Auslöser der NPE ein, aber es bleiben noch zwei Möglichkeiten bestehen. Um diese zu unterscheiden, wird eine Datenflussanalyse auf der Methode ausgeführt. Damit lässt sich ermitteln, welche Instruktion die `null`-Referenz für den Methodenaufruf bei bci 17 erzeugt hat. In unserem Fall ist es der Rückgabewert des Methodenaufrufs am bci 14. So erzeugt die SAP JVM dann folgende Message für die NPE:

```
Exception in thread "main" java.lang.NullPointerException:
  while trying to invoke the method java.lang.Integer.intValue()
  of a null object returned from NPEExample.getUniqueId()
```

Da die Datenflussanalyse aufwendig sein kann, wird die Message nicht schon beim Erzeugen der NPE zusammengebaut, sondern erst wenn sie angefragt wird.

ClassCastException

Als weiteres Beispiel für erhöhte Supportbarkeit durch bessere Exception Messages dienen `ClassCassException`s. Eine `ClassCastException` gibt normalerweise den Klassennamen des Objekts zurück, auf das ein Cast fehlerhaft angewandt wurde. Für viele Java-Applikationen ist dies auch völlig ausreichend, nicht aber im Umfeld von Applikationsservern. Der Grund liegt in der Verwendung von Classloadern, um Applikationen voneinander zu isolieren und zum Beispiel die parallele Nutzung von Bibliotheken verschiedener Versionen zu ermöglichen. Dies kann dazu führen, dass Klassen gleichen Namens in der VM existieren, die allerdings von verschiedenen Classloadern definiert worden sind. Idealerweise sollten sich diese Klassen nie „sehen“, aber durch Fehler kann es doch dazu kommen. Diese Fehler äußern sich häufig in einer `ClassCastException`.

Als Beispiel seien `ClassA` und ihre Unterklasse `ClassB` angenommen. Diese seien in zwei Versionen von verschiedenen Classloadern geladen worden. Dabei bezeichne `ClassA:loader1` die Klasse `ClassA`, definiert von `loader1`. Nun soll `ClassB:loader2` (mit Oberklasse `ClassA:loader2`) zu `ClassA:loader1` gecastet werden. Da `ClassA:loader1` nicht in der Klassenhierarchie von `ClassB:loader2` enthalten ist, geht der Cast schief und eine `ClassCastException` mit der Message „ObjectB“ wird erzeugt. Diese Meldung ist allerdings verwirrend, denn `ClassB` scheint ja eine echte Unterklasse von `ClassA` zu sein. Aus diesem Grund gibt die Message einer `ClassCastException` in der SAP JVM sowohl den Klassennamen als auch den Classloader an:

```
Exception in thread "main" java.lang.ClassCastException:
  ClassB:loader2 incompatible with classA:loader1
```

„loader2“ bzw. „loader1“ sind dabei die Rückgabewerte von `toString()` der entsprechenden Classloader. Ein sinnvolle Implementierung dieser Methode erhöht die Aussagekraft der Meldung also deutlich.

Thread-Annotationen

Für die verbesserten Java-Exceptions sind im Großen und Ganzen keine Anpassungen der Anwendung notwendig, um von ihnen zu profitieren. Im Folgenden werden einige Features vorgestellt, die durch kleinere Code-Anpassungen deutlich an Wert gewinnen. Eine der wichtigsten sind die sogenannten Thread-Annotationen. Die SAP JVM bietet ein Java-API an, um den laufenden Thread zu annotieren. Es gibt vier vordefinierte Annotationen: *user*, *request*, *session* und *application*. Für jede dieser Annotationen kann der aktuelle Wert als String für den Thread gesetzt werden. Bei Bedarf können zusätzliche Annotationen definiert werden. Die Annotationen werden dann von der VM beispielsweise dafür verwendet, beim Profiling die Ressourcen pro Benutzer zu erfassen. Eine weitere Verwendung findet sich in Thread-Dumps.

Thread-Dumps

Jede Java VM biete die Möglichkeit, sich eine Liste aller aktuell laufenden Threads sowie ihrer Stacktraces ausgeben zu lassen, die sogenannten Thread-Dumps. Die SAP JVM hat diese Thread-Dumps an vielen Stellen erweitert – stellen sie doch häufig die einzige Information da, die für die Lösung eines Kundenproblems zur Verfügung steht. Hier ein Ausschnitt aus einem Thread-Dump:

```
"Application [10]" cpu=300.00 [reset 270.00] ms elapsed=4398.43
 [reset 4391.90] s allocated=10376456 B (9.90 MB)
 [reset 8460760 B (8.07 MB)] defined_classes=33
 io= file i/o: 115970/478 B, net i/o: 147022/54143 B,
 files opened:2, socks opened:1
 [reset file i/o: 36976/478 B, net i/o: 84518/50863 B,
 files opened:2, socks opened:1 ] user="test" application="test-app"
 prio=10 tid=0x00007fc41c02c800 nid=0x7579 / 30073
 pthread-id=140480345388816 runnable
 [_thread_in_native (_at_safepoint),
 stack(0x00007fc42105b000,0x00007fc42115c000)] [0x00007fc42115a000]
java.lang.Thread.State: RUNNABLE
at java.net.SocketInputStream.socketRead0(Ljava/io/FileDescriptor;
[BIII)I(Native Method)
at java.net.SocketInputStream.read([BII)I(SocketInputStream.java:129)
- additional info (remote: lu312474.wdf.sap.corp/10.17.90.30:3974,
 local: localhost/127.0.0.1:60844)
at java.io.BufferedInputStream.fill()V(BufferedInputStream.java:218
...
```

Als zusätzliche Informationen zu einem Thread sieht man die bisher verbrauchte CPU-Zeit (cpu=…), allokierte Bytes (allocated=…), die Zeit seit Start des Threads (elapsed=…) sowie die gelesenen/geschriebenen Bytes für Datei- und Netzwerk-

I/O (io=…). Darüber hinaus bietet die SAP JVM ein API an, mit dem diese Werte für den laufenden Thread zurückgesetzt werden. Die Werte seit dem letzten Reset sind auch angegeben (als [reset …]). Ein Reset wird zum Beispiel vom Applikationsserver gemacht, wenn er eine neue Anfrage bearbeitet. Außerdem sehen wir hier die Werte der aktuell gesetzten Thread-Annotationen. Der Stacktrace selbst ist ebenfalls mit zusätzlichen Informationen angereichert. Wenn ein Thread zum Beispiel in einem Netzwerkaufruf wartet, so werden der lokale und der remote Port der Verbindung ausgegeben.

SAP JVM Profiler

Die Aufgabe eines Java-Profilers ist es, Sie bei der Analyse Ihrer Applikation im Hinblick auf den Ressourcen-Verbrauch zu unterstützen. Die klassischen Ressourcen sind die zur Verfügung stehende CPU-Leistung, Speicher und I/O-Einheiten. Der SAP JVM Profiler ist ein Java-Profiler, der Ihnen einen detaillierten Einblick in den Zustand Ihrer SAP JVM gewährt und Ihnen hilft, Ressourcen-Engpässe aufzuspüren.

Architektur

Der SAP JVM Profiler gliedert sich in zwei wesentliche Komponenten: ein in die SAP JVM integriertes Backend für das Sammeln der Analysedaten sowie ein Eclipse-basiertes Frontend für die grafische Darstellung und Aufbereitung der Informationen. Das Eclipse-Plug-in können Sie direkt in Ihr SAP NetWeaver Developer Studio oder in eine gewöhnliche Eclipse-Entwicklungsumgebung für Java einbinden.

Profiling-Traces

Der Profiler bietet Ihnen zur Beobachtung der unterschiedlichen Ressourcen mehrere Traces an, die Sie zur Laufzeit je nach Bedarf aktivieren und nach dem gewünschten Analysezeitraum wieder deaktivieren können. Das Profiling-Backend der SAP JVM sammelt bei eingeschaltetem Trace die entsprechenden Analysedaten und schickt sie zur weiteren Verarbeitung zum Frontend.

Bitte beachten Sie, dass ein aktiver Trace zu einem gewissen Performance-Overhead in der JVM führt. Beispielsweise kann beim Performance-Hotspot-Trace, der zur Aufspürung von CPU-intensiven Methoden geeignet ist, der Durchsatz des Systems um 10 bis 20 Prozentpunkte sinken. Die Ursache dafür liegt darin, dass die JVM periodisch in kurzen Intervallen von wenigen Millisekunden angehalten werden muss, um die Anwendungs-Threads zu inspizieren.

Die durch einen Trace gesammelten Daten der JVM können dann mit Hilfe der im Frontend angebotenen Statistiken eingehend analysiert und ausgewertet werden. Es stehen Ihnen hierbei folgende Traces und Analysemöglichkeiten zur Verfügung:

- Der Allocation-Trace protokolliert sämtliche Objekt-Erzeugungen und bildet somit den Speicherverbrauch über die Zeit ab.
- CPU-intensive Methodenaufrufe können Sie mit dem Performance-Hotspot-Trace aufdecken.
- Mit Hilfe des Method-Parameter-Trace können Sie die Anzahl der Aufrufe und Über- oder Rückgabeparameter bestimmter Methoden überwachen.
- Der Synchronization-Trace informiert Sie über Blockierungen von Threads auf Grund von Sperren.
- I/O-Operationen der JVM werden vom Network-I/O- und Datei-I/O-Trace erfasst.
- Mit der Garbage Collection Analysis können Sie auffällige GCs bis ins kleinste Detail untersuchen.
- Die Class-Statistic und die Heap Dump Analysis verschaffen Ihnen einen Überblick über die geladenen Klassen und die Zusammensetzung des Java-Heaps zu einem bestimmten Zeitpunkt.

SAP JVM Debugger

Die Debugging-Funktionalität einer JVM bietet die Möglichkeit, eine Java-Applikation „remote“ über eine „Wide Area Network“-Verbindung zu debuggen. Der Java-Standard hat dazu das sogenannte Java Debug Wire Protocol (JDWP) spezifiziert, was eine Entkopplung zwischen Debugger und Debuggee ermöglicht. Das heißt, ein beliebiger Debugger, der das JDWP-Protokoll implementiert hat, kann eine beliebige JVM debuggen.

Besteht jedoch eine relativ große Latenz (>= 50 ms) zwischen Debugger und Debuggee, dann wird das Debugging quasi unbenutzbar und eine interaktive Debugging-Sitzung wird unmöglich. Grund dafür ist, dass eine große Anzahl von JDWP-Paketen zwischen Debugger und Debuggee gesendet wird. Gerade im Cloud-Umfeld, wo Applikationen gehostet werden und Entwickler über den Erdball verstreut sind, wird das Debugging von Java-Applikationen unmöglich.

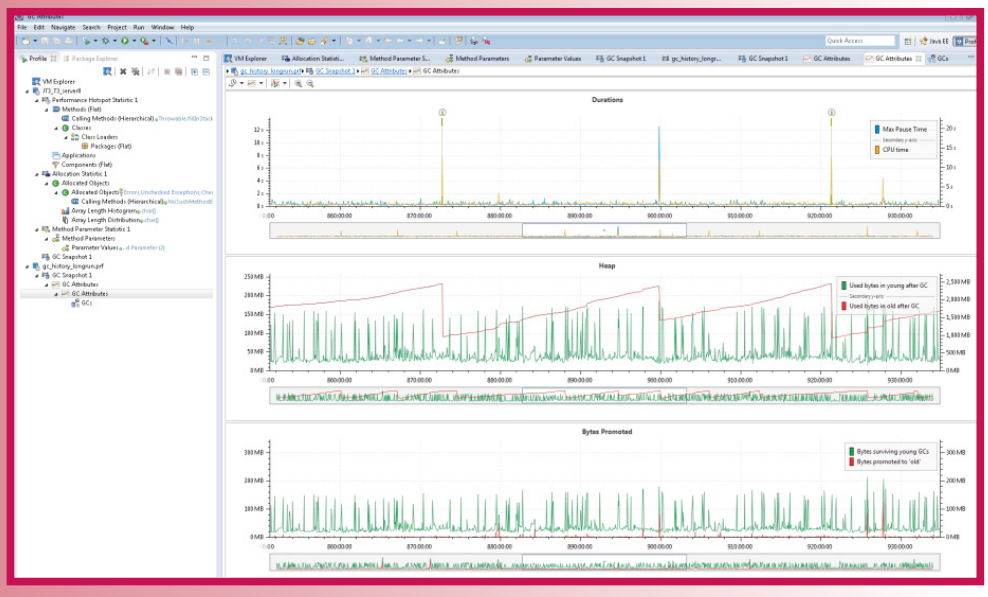

Abb. 2: SAP JVM Profiler – Eclipse Perspective

Als Lösung des Problems bietet die SAP JVM eine eigene Debugging-Lösung an, den sogenannten SAP JVM Debugger. Diese Lösung ist insbesondere zum Remote Debugging von Java-Applikationen über WAN-Verbindungen mit relativ großer Latenz geeignet. Der eigentliche Debugger ist wiederum in eine Frontend- und eine Backend-Komponente unterteilt. Das Frontend ist als Eclipse-Feature implementiert und kann in jede Eclipse-basierte Entwicklungsumgebung eingebettet werden. Es verwendet die üblichen Ec-

lipse-Debugging-Views wie Debug View, Thread View, Breakpoint und Variables View. Aus Entwicklersicht gibt es keinen Unterschied zum normalen Eclipse-Debugging.

Fazit

Der Artikel hat Ihnen einen ersten Einblick in die SAP JVM-Technologie und deren Werkzeuge gegeben. In weiteren Ausgaben von JavaSPEKTRUM werden wir im Detail auf den SAP JVM Profiler und den SAP JVM Debugger eingehen.

Links

[Hana] `https://help.hana.ondemand.com/`
[OpenJDK] `http://openjdk.java.net/`
[PowerPC/AIXPort]
`https://wiki.openjdk.java.net/pages/viewpage.action?pageId=13041681`
[SAPJVM] `https://help.hana.ondemand.com/help/frameset.htm?da030d10d97610149defa1084cb0b2f1.html`

Dr. Ralf Schmelter widmet sich seit seinem Chemie-Studium und seiner Promotion der JVM-Entwicklung innerhalb der SAP AG. Als führender Architekt in der SAP JVM-Entwicklung ist er auch seit über zehn Jahren in diesem Umfeld tätig.
E-Mail: ralf.schmelter@sap.com

Michael Wintergerst beschäftigt sich nunmehr seit über zehn Jahren mit dem Thema JVM innerhalb der SAP AG. Er leitet als Development Manager die Entwicklungsabteilung der SAP JVM.
E-Mail: michael.wintergerst@sap.com