01-2015 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Fünf Jahre Java aktuell

Praxis
Mehr Typsicherheit mit Java 8

Programmieren
Der selbe GUI-Code für Desktop, Web und native mobile App

SAP HANA
Was für Java-Anwendungen drin ist

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 01

iJUG Verbund

Happy Birthday Java aktuell

Wege zum Aufbau von modernen Web-Architekturen

Groovy-Entwickler argumentieren, dass Java in die Jahre gekommen sei

Java ist aktuell ein sicherer Hafen

Java hat in den letzten Jahren in der Tat einen größeren Aufschwung erlebt und wird in allen Bereichen der IT, von Kleinst-Systemen bis hin zum Großrechner, von kleinen Tools bis hin zu geschäftskritischen 24x7-Anwendungen, eingesetzt. Die hohe Geschwindigkeit, mit der Java-Anwendungen entwickelt werden können, haben jedoch auch Spuren hinterlassen. Im Artikel „Estimating the Principal of an Application's Technical Debt" von Bill Curtis, Jay Sappidi und Alexandra Szynkarski wird Java-EE-Anwendungen ein um 40 Prozent höherer Nachbesserungsbedarf bescheinigt als durchschnittlichen Anwendungen. Wir müssen hier aufpassen, dass wir die nächsten Jahre nicht damit beschäftigt sind, die Nachlässigkeiten der Vergangenheit nachzupflegen.

Dennoch, die Tatsache, dass viele große Unternehmen die Ablösung ihrer Altsysteme von Cobol und PL/1 nach Java planen, spricht für sich. Java ist aktuell der sichere Hafen für einen zukunftssicheren Betrieb einer Anwendungsplattform. Entwickler und Betriebspersonal sind in ausreichendem Maß vorhanden.

http://ja.ijug.eu/15/1/8

Marc Bauer
marc.bauer@de.ibm.com

Marc Bauer arbeitet im Software Service der IBM Deutschland GmbH. Er berät und unterstützt Kunden bei Modernisierungen von Mainframeanwendungen mit Hilfe von WebSphere und Java-Applikationen

Unbekannte Kostbarkeiten des SDK Heute: Informationen über die Virtual Machine und die Programmausführung

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir stellen in dieser Reihe derartige Features des SDK vor: die unbekannten Kostbarkeiten.

Application-Server stellen in Management-Seiten vielerlei Informationen über die Virtual Machine (Executable, Version etc.) sowie die Programmausführung (durchgeführte Garbage Collections, Anzahl Threads etc.) bereit. Diese Informationen können jedoch auch in eigenen Anwendungen über einfach zu verwendende APIs erfragt und verwendet werden. Der Artikel stellt einige davon vor.

System-Properties

Die Methode „System.getproperties()" liefert die System-Properties der aktuellen Programmausführung. Im API-Dokument der Methode sind die Schlüssel aufgeführt, die standardmäßig vorhanden sind. Sie werden aber durch weitere Schlüssel ergänzt. Bei Java 8 sind dies im Augenblick 52, von denen einige in *Tabelle 1* mit ihren Werten beispielhaft dargestellt sind.

Exkurs: Abwärts- und Aufwärtskompatibilität

Java ist generell nicht aufwärtskompatibel. Eine Klasse, die mit Java 8 kompiliert wurde, ist unter Java 7 und kleiner nicht lauffähig. Bei der Ausführung wird die Exception „UnsupportedClassVersionError" geworfen und die nicht unterstützte Version angegeben. Im Falle von Java 8 ist dies der Fehlertext „Unsupported major.minor version 52.0", wobei 52.0 der Wert des System-Property „java.class.version" ist.

Java ist jedoch prinzipiell abwärtskompatibel. Klassen, die mit früheren Java-Versionen kompiliert wurden, sind auf neueren JVMs ausführbar. Kürzlich gab es ein damit zusammenhängendes Problem *[1]*. In den JVMs für 8u11 und 7u65 wurde ein bestimmter Punkt der Byte-Code-Verifikation erstmalig der Sprachdefinition entsprechend durchgeführt. Java verlangt, dass die erste Anweisung in einem Konstruktor der Aufruf des Konstruktors der Oberklasse ist. Dies wird selbstverständlich von allen Java-Compilern überprüft und entsprechender Byte-Code erzeugt.

Bei der nachträglichen Instrumentierung durch Byte-Code-Manipulationswerkzeuge wie „Javass" ist wird dies jedoch scheinbar von einigen Frameworks missachtet, sodass bei derartigen Klassen die Byte-Code-Verifikation

diesen Umstand erkennt und diese Klassen nicht ausgeführt werden können. Das Interessante an diesem Beispiel ist die Tatsache, dass rund zwanzig Jahre nach der ersten Java-Version in der JVM nun tatsächlich etwas verifiziert wird, was schon immer erfüllt sein musste.

Soll verhindert werden, dass ein Java-Programm mit einer höheren Java-Version ausgeführt wird, muss die Version der verwendeten Klassen mit der JVM-Version aus obiger Tabelle verglichen werden. Dies für die Praxis wenig relevante Problem ist so einfach zu lösen, dass wir es nicht vorenthalten wollen. *Listing 1* zeigt den Zugriff auf die ersten Bytes einer Java-Klasse, die die Java-Identifizierung („CAFEBABE") sowie die Minor- und Major-Version enthalten. *Listing 2* zeigt die Ausgabe, wenn die Klasse mit Java 8 kompiliert wird.

Javas Management-Interface

Das Package „java.lang.management" enthält eine Reihe von Schnittstellen zu MXBeans und eine Fabrikklasse zur Erzeugung von Instanzen dieser MXBeans. Es existieren MXBeans, die Informationen zum Laufzeitsystem, zu Speicherbereichen und zur Speicherverwaltung, zur Thread-Verwaltung und weiteren Themen bereitstellen.

Betrachten wir zunächst die Speicherbereiche. Die Fabrikmethode „ManagementFactory.getMemoryPoolMXBeans()" liefert eine Liste von MemoryPoolMXBeans, die die verschiedenen Speicherbereiche einer JVM repräsentieren. *Tabelle 2* zeigt diese Speicherbereiche in verschiedenen JVM-Versionen. Die Namen der Speicherbereiche wurden durch das folgende Code-Segment erzeugt *(siehe Listing 3)*.

Die ersten beiden Spalten der Tabelle wurden durch Aufruf der JVM ohne weitere Option, die dritte Spalte mit der Option „-XX:+UseG1GC" erzeugt. Man erkennt, dass in Java 8 der Permanent-Generation-Speicherbereich durch den Metaspace-Speicherbereich ersetzt sowie ein zusätzlicher Speicherbereich für die Repräsentation von Klassen eingeführt wurde.

Eine weitere wichtige Information ist die Größe einzelner Speicherbereiche. Das Interface „MemoryPoolMXBeans" stellt hierzu die Methoden „getUsage()", „getPeakUsage()" und „getCollectionUsage()" mit der offensichtlichen Semantik bereit, die jeweils eine Instanz von „MemoryUsage" zurückgeben. Diese Instanz enthält die vier Werte „init", „used", „committed" und „max", die den initialen, aktuell benutzten, verfügbaren und maximalen Speicherbereich in Bytes angeben.

Die JVM in ihrer heutigen Form erlaubt die Verwendung mehrerer Hundert Kommandozeilen-Optionen, von denen wir bereits „-XX:+UseG1GC" eingesetzt haben. Der Garbage Collector ist, wie im Beispiel, indirekt über die „MemoryPoolMXBeans" in Erfahrung zu bringen. Die Fabrikmethode „ManagementFactory.getGarbageCollectorMXBeans()" liefert eine Liste von „GarbageCollectorMXBeans" und erlaubt so den direkten Zugriff. Bei vielen weiteren Optionen gestaltet sich dies jedoch nicht so einfach. Abhilfe schafft hier die Methode „ManagementFactory.getRuntimeMXBean()", die Laufzeit-Informationen in Form des Interface „RuntimeMXBean" zurückgibt. Deren Methode „getInputArguments()" liefert die Liste aller Kommandozeilen-Optionen, also neben Standard-, „-X"- und „-XX"-Optionen auch Property-Definitionen mit „-D".

Fazit

Java und die JVM stellen eine Reihe von Schnittstellen zur Verfügung, um Informationen über die virtuelle Maschine und die Programmausführung zu erlangen. Es wurden die beiden großen Alternativen über Properties und MXBeans eingeführt und mit kleinen Beispielen vorgestellt.

Literatur/Videos

[1] http://www.infoq.com/news/2014/08/Java8-U11-Broke-Tools

Schlüssel	Wert
java.vm.version	25.11-b03
java.runtime.version	1.8.0_11-b12
java.class.version	52.0
java.specification.version	1.8
java.vm.specification.version	1.8
java.home	/java/jdk/jdk1.8.0_11/jre
os.name	Linux
user.home	/home/bernd
file.separator	/

Tabelle 1

Java 5/6/7	Java 8 (Default)	Java 8 (-XX:+UseG1GC)
PS Eden Space	PS Eden Space	G1 Eden Space
PS Survivor Space	PS Survivor Space	G1 Survivor Space
PS Old Gen	PS Old Gen	G1 Old Gen
Code Cache	Code Cache	Code Cache
PS Perm Gen	Metaspace	Metaspace
	Compressed Class Space	Compressed Class Space

Tabelle 2

Alle Listings finden Sie online unter:
http://www.doag.org/go/javaaktuell/link2

Bernd Müller
bernd.mueller@ostfalia.de

Bernd Müller ist Professor für Software-Technik an der Ostfalia. Er ist Autor des Buches „JavaServer Faces 2.0" und Mitglied in der Expertengruppe des JSR 344 (JSF 2.2).

http://ja.ijug.eu/15/1/9