



Wenn's mal wieder länger dauert ...

Einführung in den NetBeans Profiler

Dominik Hofmann 14.01.2010

Themen

- Grundlagen und Features des NetBeans Profilers
- CPU Profiling
- Memory Profiling einer Web-Anwendung



Let's talk about IT



Teil I: Grundlagen und Features des NetBeans Profilers

Was ist ein Profiler?

- Werkzeug zur Analyse von Performance- und Memory-Problemen
- Profiler überwachen
 - CPU Performance
 - Verwendung von Speicherplatz
 - Zustand der Threads
- Grundlage: JVMTI (Java Virtual Machine Tool Interface) und Bytecode-Instrumentierung
- Nachteil/Einschränkung: Overhead beim Sammeln der Informationen
=> meist nicht für produktiven Umgebungen geeignet

Tool-/Marktübersicht

- jconsole und VisualVM in JDK 6 enthalten
- NetBeans: NetBeans Profiler
- Eclipse: Eclipse Test & Performance Tools Plattform (TPTP)
- JProfiler
- JProbe
- ...

Features des NetBeans Profilers

- Monitor Application: high-level Informationen über die Ausführung der Anwendung (Aktivitäten der Threads, verwendeten Speicher)
- Analyze CPU Performance: Ausführungszeiten der Methoden und die Anzahl der Aufrufe je Methode
- Analyze Memory Usage liefert detaillierte Daten zur Anlage von Objekten im Heap sowie zur Garbage Collection

Features des NetBeans Profilers

Feature	Beschreibung
Heap Dumps und HeapWalker	HeapWalker zur Analyse von HeapDumps
Snapshots	Snapshots liefern detaillierte Momentaufnahmen für die spätere Analyse
Profiling Points	Profiling Points können wie Debugger Break Points im Sourcecode platziert werden. Profiling Points triggern z. B. Snapshots oder Heapdumps.
Thread Timeline	Die Thread Timeline zeigt den zeitlichen Verlauf des Status aller Threads.
Remote Profiling	Mit dem Remote Pack des Profilers können Anwendungen analysiert werden, die auf einem anderen System laufen als die NetBeans IDE.
Lasttest-Unterstützung	z.B. Jmeter: Über Profiling Points wird die Ausführung von Lasttestskripten gesteuert.



Let's talk about IT



Teil II: CPU Profiling

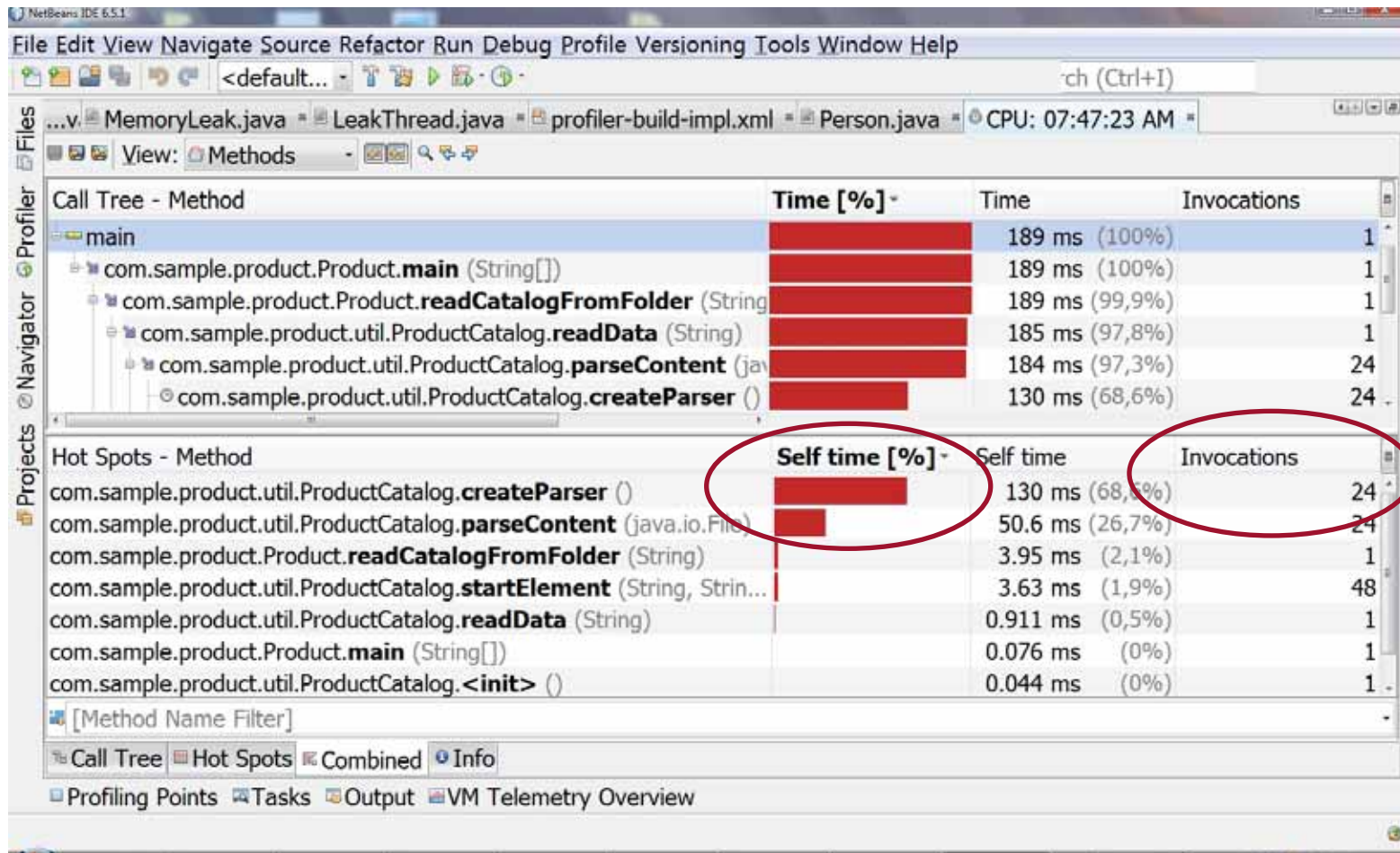
Beispiel: CPU Profiling

- Szenario: Einlesen von Dateien mit Produktinformationen mit SAX, Ausgabe von Informationen auf der Console
- Code:

```
protected void parseContent(final File file) throws Exception {  
  
    SAXParser parser = createParser();  
    InputStream is = new FileInputStream(file);  
    parser.parse(is, this);  
  
}  
  
protected SAXParser createParser() throws Exception {  
  
    SAXParserFactory f = SAXParserFactory.newInstance();  
    f.setValidating(false);  
    return f.newSAXParser();  
  
}
```

Beispiel: CPU Profiling

- CPU Snapshot der Beispielanwendung



Analyse der Profiling Ergebnisse

- Ansicht Hot Spots – Method: Self Time und Zahl der Ausführungen einer Methode
- Self Time: Zeit, die tatsächlich in einer Methode verbraucht wurde – also ohne Untermethodenaufrufe
- im Beispiel: `createParser()` wird 24 mal ausgeführt und verbraucht am meisten Zeit
- Grund (Codeanalyse): SAX Parser wird in jedem Schleifendurchlauf - also für jede Datei - neu instanziiert

Optimierung

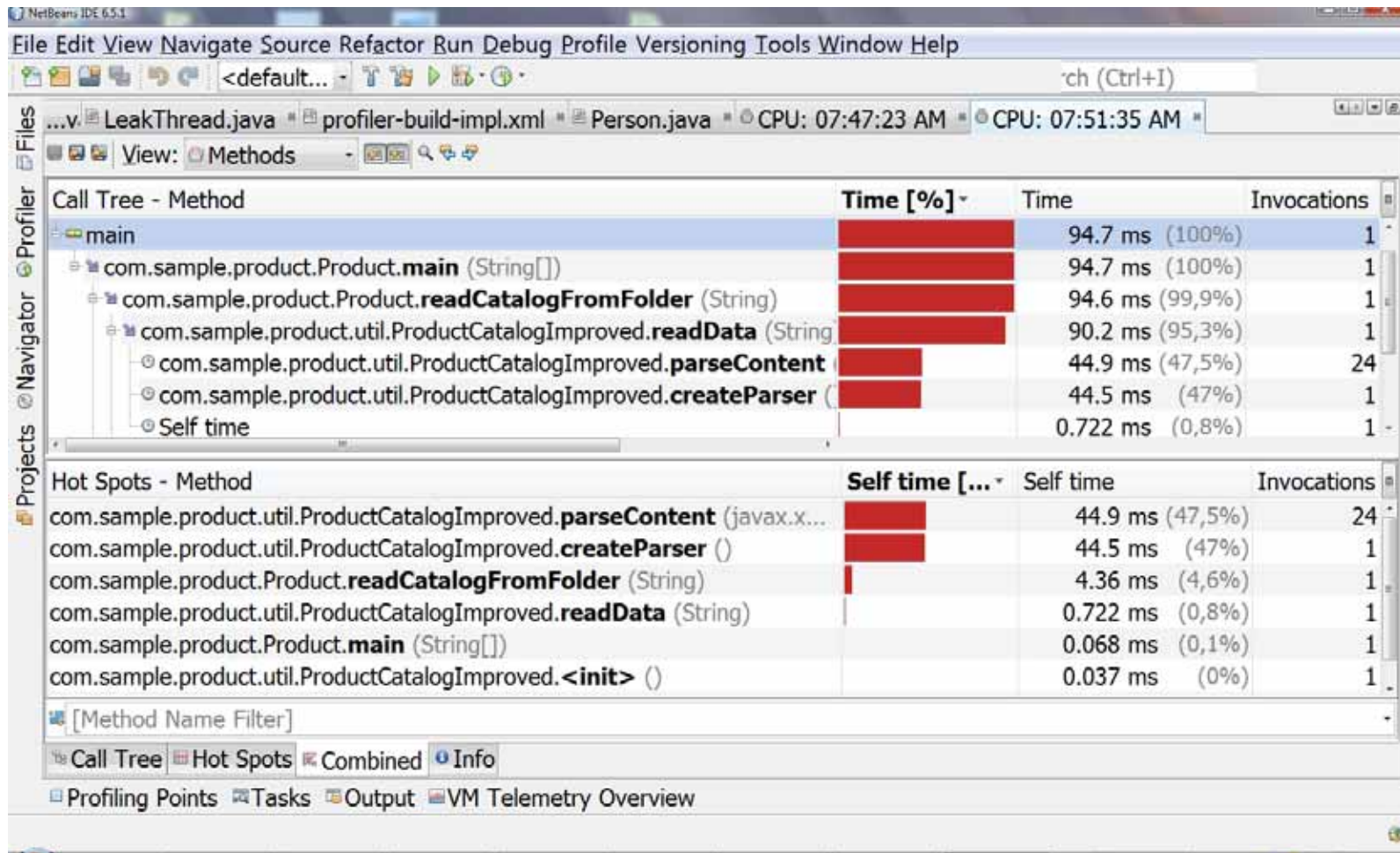
- Vorschlag: Parser nur einmal erzeugt und in den Schleifendurchläufen wiederverwendet
- Wichtig: Optimierung darf Korrektheit nicht gefährden (Unit-Tests)

```
public void readData(final String directoryName) throws
    ...
    SAXParser parser = createParser();
    for(int idx=0; idx<files.length; idx++) {
        parseContent(parser, files[idx]);
    }
}
```

```
protected void parseContent(final SAXParser parser, final
    InputStream is = new FileInputStream(file);
    parser.parse(is, this);
}
```

Profiling der optimierten Anwendung

- CPU Snapshot der Beispielanwendung nach Verbesserungen





Teil III: Memory Profiling einer Web-Anwendung

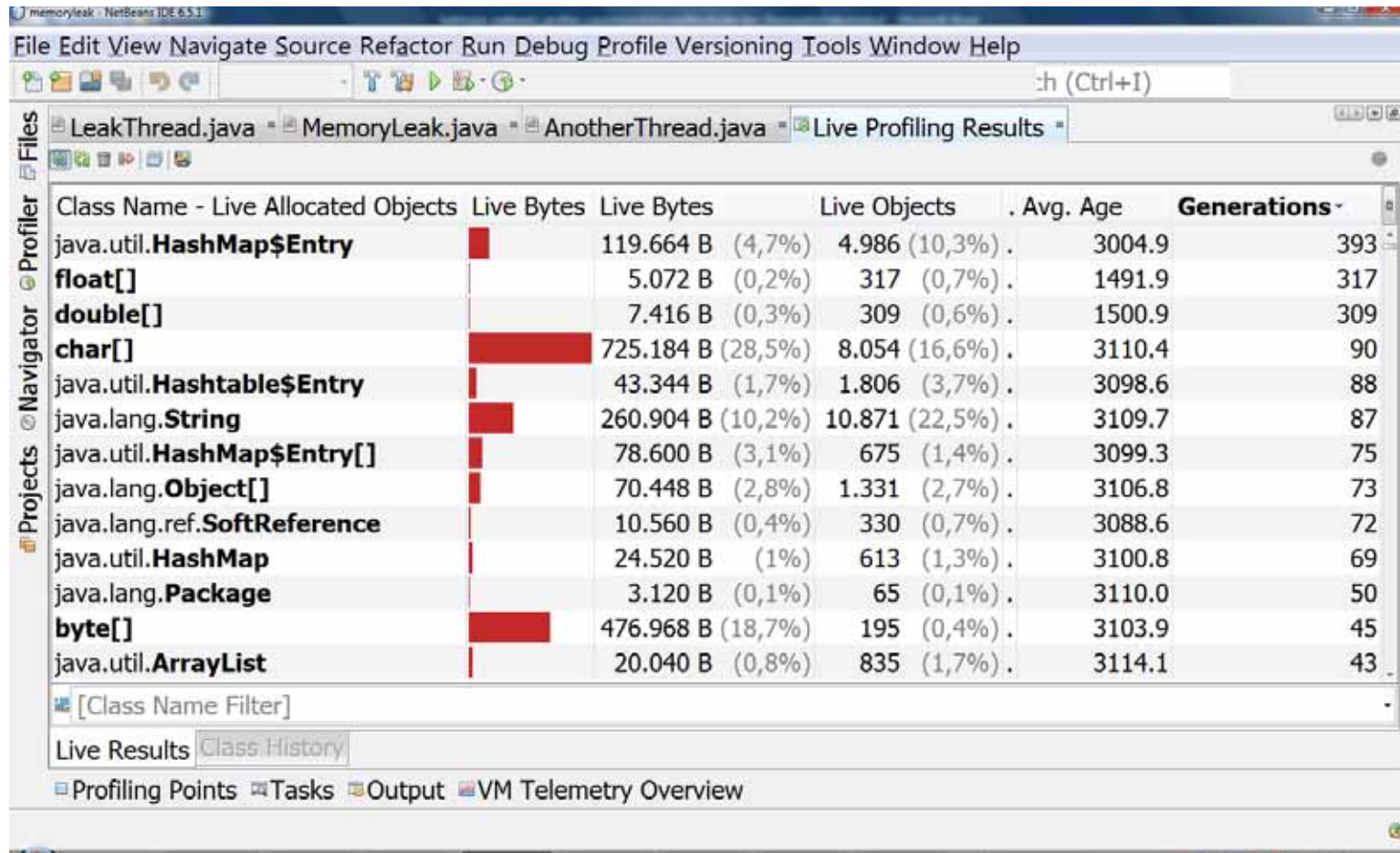
Memory Profiling einer Webanwendung

- Profiling von Applikationen auf GlassFish, JBoss oder Tomcat per Mausklick möglich
- Für andere Server steht Attach-Wizard zur Verfügung
- Beispielszenario: Webanwendung mit Memory Leak
- „böse“ Codestelle: *OutOfMemoryError* wenn Programm länger läuft

```
public class LeakThread extends Thread {  
  
    private Map<Integer, MyTestBean> map = HashMap<Integer,  
MyTestBean> ();  
  
    public void run () {  
        int counter = 1;  
        ...  
        while (!stop) {  
            map.put (counter, new MyTestBean(counter++));  
            Thread.sleep (100);  
            System.gc ();  
        }  
    }  
}
```

Memory Profiling liefert Informationen über Speicherverbrauch der Objekte

Live Results beim Memory Profiling



Live Results beim Memory Profiling

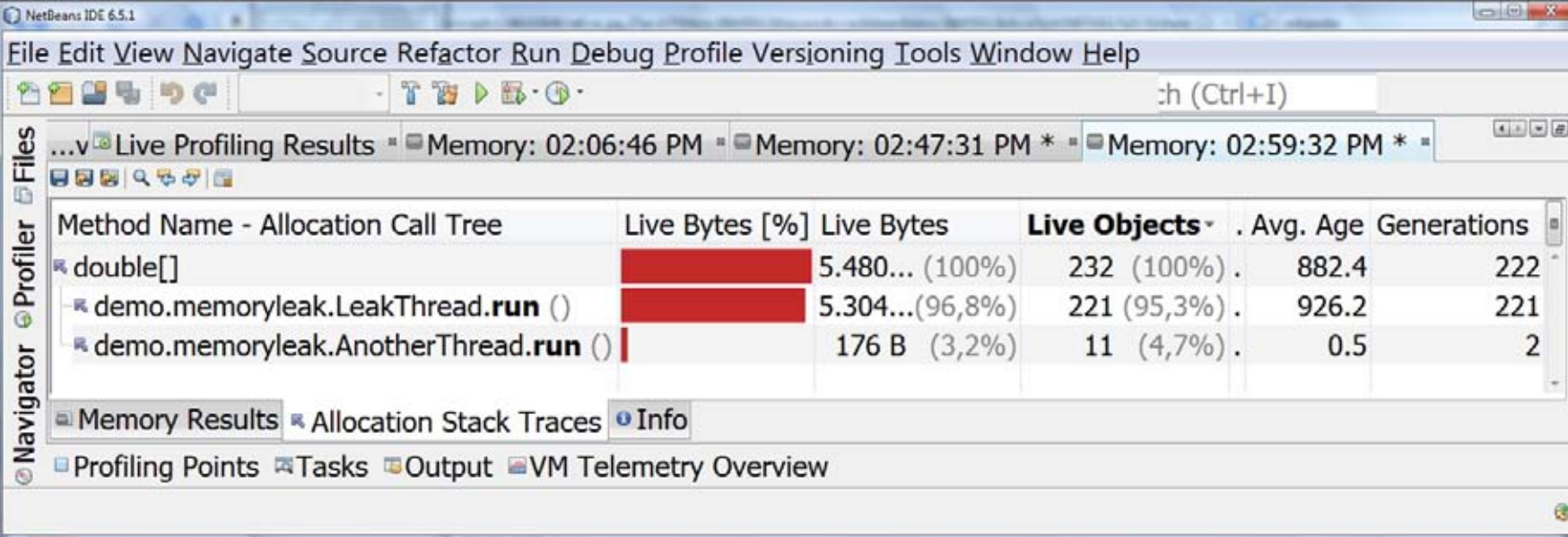
- Allocated Objects: Anzahl der Objekte, die der Profiler untersucht.
- Live Objects: Anzahl der Objekte im Heap der JVM residieren
- Live Bytes Spalten: Menge des Speichers, der von Live Objects belegt wird.
- Avg. Age: durchschnittliche Zahl der Garbage Collection Zyklen, die die Objekte überlebt haben.
- Generations: Anzahl der verschiedenen Alterswerte der Live Objects.

Surviving Generations Metrik

- Ansatzpunkt zur Identifikation von Memory Leaks
- Generation: Menge von Instanzen einer Klasse, die zwischen zwei Läufen des Garbage Collectors angelegt werden
- Surviving Generation: Generation, die Garbage Collection überlebt hat
- Alter einer Generation: Anzahl der überlebten Garbage Collection Zyklen
- Surviving Generations Metrik: Anzahl der verschiedenen Surviving Generations einer Klasse => Anzahl der Generationen mit verschiedenem Alter
- Verdächtig sind: Klassen, bei denen die Zahl der Surviving Generations stetig zunimmt \Leftrightarrow GC schafft es nicht Objekte einer Generation zu Löschen

Allocation Stack Trace des Memory Leaks

- Identifizieren von Codestellen, die verdächtige Objekte anlegen



The screenshot shows the NetBeans IDE 6.5.1 interface. The Profiler window is open, displaying the Allocation Call Tree for a memory leak. The table below summarizes the data shown in the Profiler window.

Method Name - Allocation Call Tree	Live Bytes [%]	Live Bytes	Live Objects	Avg. Age	Generations
double[]		5.480... (100%)	232 (100%)	882.4	222
demo.memoryleak.LeakThread.run ()		5.304... (96,8%)	221 (95,3%)	926.2	221
demo.memoryleak.AnotherThread.run ()		176 B (3,2%)	11 (4,7%)	0.5	2

The Profiler window also shows tabs for Memory Results, Allocation Stack Traces, and Info. The Allocation Stack Traces tab is currently selected, showing the call stack for the selected method.

HeapWalker

- HeapWalker stellt alle Objekte auf dem Heap sowie die Referenzen zwischen diesen Objekten dar

The screenshot shows the NetBeans IDE 6.5.1 interface with the HeapWalker tool. The main window displays 'Live Profiling Results' for 'Memory: 02:06:46 PM' and 'Heap: 02:12:11 PM'. The 'Instances' tab is active, showing a list of instances for 'double[]' with 3,966 instances and a total size of 63,768. The 'Fields' and 'References' panels are also visible, showing details for a selected instance.

Instance	Size	Field	Type	Value
<500 instances>				
#1	200	this	double[]	#14 (1 items)
#2	56	[0]	double	0.0
#3	56			
#4	56			
#5	56			
#6	56			
#7	24			
#8	24			
#10	24			
#13	24			
#14	24			

Field	Type	Value
this	double[]	#14 (1 items)
value	HashMap\$Entry	#45368
[2365]	HashMap\$Entry[]	#6543 (8.192 items)
table	HashMap	#6387
map (thread object)	LeakThread	#1

Legend: Array type | Object type | Primitive type | Static field | GC Root | Loop

Features des HeapWalker

- HeapDumps erstellen
 - per Knopfdruck während Profiling Session
 - automatisiert per Profiling Point
 - Heap Dumps bei *OutOfMemoryErrors* (z.B. mit der JVM Option *HeapDumpOnOutOfMemoryError*).
- CLASSES Ansicht: alle Klassen mit Zahl der Instanzen und belegtem Speicherplatz
- Instances View: Instanz und sich ihre Attribute sowie alle Objekte anzeigen lassen, welche die gewählte Instanz referenzieren.
- Garbage Collection (GC) Roots sind Objekte die niemals vom Heap entfernt werden => Ausgangspunkt der GC
- SHOW NEAREST GC ROOT: zeigt GC Root, die verhindert, dass Objekt vom Heap gelöscht wird
=> Ausgangspunkt für Codeanalyse

Profiling bei F&F

- Siemens/Pegasus: VisualVM und Jconsole, hauptsächlich bei Memory-Problemen, z.T. auch prophylaktisch (ASM)
- Jetstream: JProfiler zur Suche von Memory-Problemen, z.B. in XML-Bibliotheken (RL/FK)
- Ärztekammer: JProfiler zur Performanceoptimierung

Fazit / Zusammenfassung

- Profiler helfen, langsame Methoden und Memory Leaks zu identifizieren
- Leistungsfähige, kostenlose Profiler verfügbar: NetBeans Profiler und VisualVM
- Aber: nicht für jedes Performance-Problem ist Profiler der richtige Ansatzpunkt
 - ca. 80% der Performance Probleme in Informationssystemen durch langsame Datenbankzugriffe
 - => bei Informationssystemen Profiler tendenziell eher für Memory-Probleme
- Tool alleine hilft nicht weiter: Erfahrung bei der Interpretation der Ergebnisse nötig

Literatur

- NetBeans Profiler - Wenn's mal wieder länger dauert ..., Javamagazin 12/2009
- <http://profiler.netbeans.org/>
- Mirco Novakovic und Marc van den Bogaard: Profiling, Diagnose und Monitoring, Javamagazin 01/2008
- Gregg Sporar und Sang Shin: Finding Memory Leaks Using the NetBeans Profiler, <http://www.javapassion.com/handsonlabs/nbprofilermemory/>





Let's talk about IT



Let's talk about IT

F&F Computer Anwendungen und Unternehmensberatung GmbH

Westendstraße 195
D-80686 München

Tel.: + 49 89 51727-0
Fax: + 49 89 51727-111

kontakt@ff-muenchen.de
www.ffa-muenchen.de