Java Flight Recorder

Der Flugdatenschreiber

Der Java Flight Recorder ist mit einem echten Flugdatenschreiber vergleichbar. Die auf ihm gespeicherten Informationen werden wichtig, wenn etwas schief gelaufen ist. Mit seiner einheitlichen Datenbasis hilft der Flight Recorder außerdem Administratoren und Entwicklern dabei, gemeinsam an Problemlösungen zu arbeiten und Konflikte besser und schneller bewältigen zu können.

von Wolfgang Weigend

Der Java Flight Recorder bildet zusammen mit der Managementkonsole Java Mission Control ein Toolset für Java. Java Mission Control (JMC) ist ein etabliertes Werkzeug für das Monitoring, Profiling und die Diagnose von Java-Anwendungen. Es ist im Oracle JDK enthalten und lässt sich mit beliebigen Java-Anwendungen und Applikationsservern verwenden, von der Entwicklung bis zum produktiven Betrieb. Als Bundle mit der HotSpot Java Virtual Machine ist JMC als Client oder als Eclipse-Version seit dem Oracle JDK 7 Update 40 verfügbar. Es basiert auf dem Eclipse RCP.

Mit dem Kommando *C:\JAVA_HOME\bin\jmc.exe* wird Java Mission Control beispielsweise unter Microsoft Windows gestartet. JMC lässt sich über ein Oracle-Plug-in auch direkt in die Eclipse-IDE integrieren. Die auf dem System laufenden Java-Prozesse werden in einer Baumstruktur im JVM-Browser angezeigt, und mittels Doppelklick wird der darunter liegende MBean-Server oder der Flight Recorder ausgewählt. Der MBean-Server vom Clock Applet Viewer zeigt allgemeine Informationen zum Prozessor- und Heap-Verbrauch, Memory, Code, Threads, I/O, Systeminformationen und Eventtypen an (**Abb. 1**). Wird der Flight Recorder per Doppelklick ausgewählt, bekommt man zuerst den Hinweis, dass die kommerziellen Features nicht freigegeben sind. Seit JDK 8 Update 40 wird gefragt, ob man die kommerziellen Features zum Start des Flight Recording automatisch freigeben möchte, mit dem Hinweis, dass die kommerziellen Features dem Oracle Binary Code License Agreement unterliegen [5].

Nach einer positiven Antwort wird der Flight Recorder Wizard aktiviert und bietet die Aufnahmedauer von beispielsweise einer Minute an. Im darauf folgenden Wizard-Dialog können die Aufzeichnungsparameter ausgewählt werden. Sämtliche Konfigurationsparameter der grafischen Oberfläche lassen sich auch über die Kommandozeilenebene *JDK/bin/jcmd <pid> <cmd>* einstellen (Listing 1). Existierende JFR-Plug-ins gibt es für WebLogic und JavaFX. Andere Java-Programme lassen sich über die Kommandozeilenebene mit dem Flight Recorder ebenfalls aufzeichnen. Die JFR-Freigabeparameter können dauerhaft in der Konfigurationsdatei *C:\JAVA_HOME\bin\jmc.ini* für die JVM gesetzt werden:

```
-XX:+UnlockCommercialFeatures
-XX:+FlightRecorder
-XX:FlightRecorderOptions=defaultrecording=true
```

Entstehung der Werkzeuge

Die ursprüngliche Entwicklung der Werkzeuge JRockit Mission Control Console, JRockit Runtime Analyzer, JRockit Memory Leak Detector und JRCMD (CLI für Diagnosekommandos) fand im JRockit Virtual Machine Team statt. Unter BEA Systems wurde der Runtime Analyzer durch den JRockit Flight Recorder ersetzt. Mit der Firmenübernahme von BEA Systems und Sun Microsystems durch Oracle wurde Java Mission Control (JMC) und Java Flight Recorder (JFR) für die HotSpot JVM im Oracle JDK 7 Update 40 weiterentwickelt. Es ist mit der Version JMC und JFR 5.5, basierend auf Eclipse 4.4, im Oracle JDK 8 enthalten [1] und für die Betriebssysteme Microsoft Windows, Linux und Mac OS X zertifiziert [2]. JMC und JFR sind als eingebundene Werkzeuge mit Java SE Advanced Support paketiert [3], [4], sodass sie nur über einen expliziten Java-SE-Advanced-Support-Vertrag kommerziell verwendet werden dürfen. Die kostenfreie Verwendung von JMC und JFR ist nur zum Selbststudium oder zur privaten Eigenentwicklung erlaubt. JMC ermöglicht das direkte JVM-Online-Monitoring mittels JMX und visualisiert die vom JFR gesammelten Daten. JFR übernimmt die Offline-Profiling- und Diagnosefunktion während des gesamten Applikationslebenszyklus, von der Anwendungsentwicklung über Test, Integration bis zum Anwendungsbetrieb mit einer kontinuierlichen Optimierung.

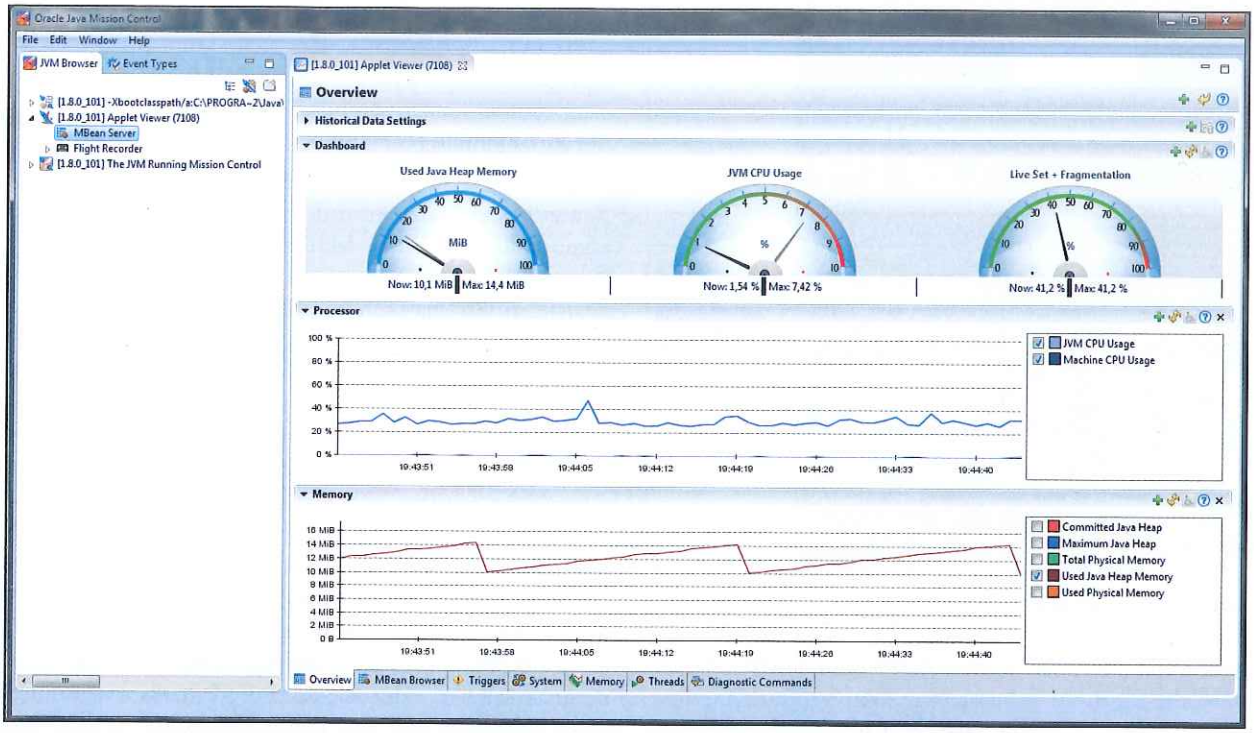

Abb. 1: Beim Start zeigt der MBean-Server vom Clock Applet Viewer allgemeine Informationen an, z. B. zum Prozessor- und Heap-Verbrauch, Memory oder Threads

Architektur des Java Flight Recorders

Der Java Flight Recorder ist ein Event Recorder, der direkt mit der Java Virtual Machine (JVM) verzahnt ist und alle verfügbaren JVM-Laufzeitparameter liefert [6]. Die Messung zeigt das tatsächliche Verhalten der eingeschalteten JVM-Optimierungsparameter. Die Aufzeichnung erfolgt in einem kompakten Binärformat, das alle notwendigen Informationen zur späteren Analyse und Diagnose enthält. Das proprietäre Binärformat ist für schnelles Schreiben und Lesen ausgelegt und lässt sich komprimieren. Der Mehraufwand für die Aufzeichnung beträgt rund drei Prozent einer bestehenden Anwendungsinfrastruktur. Der Einsatz vom JFR wurde für den dauerhaften Betrieb konzipiert, eine JFR-Datei lässt sich jederzeit erzeugen. Kontinuierliche Aufnahmen haben kein definiertes Ende, und man muss eine JFR-Datei über *dumponexit=true* explizit erzeugen. Es lassen sich aber auch zeitlich fest definierte Aufnahmen durchführen.

Der Flight Recorder sammelt über Events sowohl die Informationen der JVM durch interne APIs als auch die Daten der darauf laufenden Java-Anwendung über das Java-API. Die erzeugten Daten werden in kleine lokale Threadbuffer gespeichert. Nach dem Erreichen der maximalen Threadbufferkapazität werden sie in globale In-Memory-Buffer kopiert und schließlich auf die Festplatte geschrieben. Dabei gibt es keine überlappenden Informationen der Threadbuffer und Globalbuffer, sodass sich ein betreffendes Datensegment entweder noch im Speicherbereich der Bufferstruktur befindet oder bereits auf die Festplatte gespeichert worden ist (**Abb. 2**). Durch die gleichzeitige Aufnahme von Java-Anwendungsinformationen und den JVM-Informationen, wie Class Loading, Code-Cache-Compiler, Garbage Col-

Listing 1: Kommandozeilenbeispiel „jcmd“ mit Flight Recording für Clock Applet

```
C:\>jcmd
1936
7108 sun.plugin2.main.client.PluginMain
        read_pipe_name=jpi2_pid1936_pipe1,
        write_pipe_name=jpi2_pid1936_pipe2
4264
7512 sun.tools.jcmd.JCmd

C:\>jcmd 7108 help
7108:
The following commands are available:
JFR.stop
JFR.start
JFR.dump
JFR.check
VM.check_commercial_features
VM.unlock_commercial_features
..
For more information about a specific command
        use 'help <command>'.

C:\>jcmd 7108 VM.unlock_commercial_features
7108:
Commercial Features now unlocked.

C:\>jcmd 7108 JFR.start name=MyRecording
        settings=profile delay=20s duration=2m
        filename=c:\temp\myrecording.jfr
7108:
Recording 3 scheduled to start in 20 s. The
        result will be written to:

C:\temp\myrecording.jfr
```

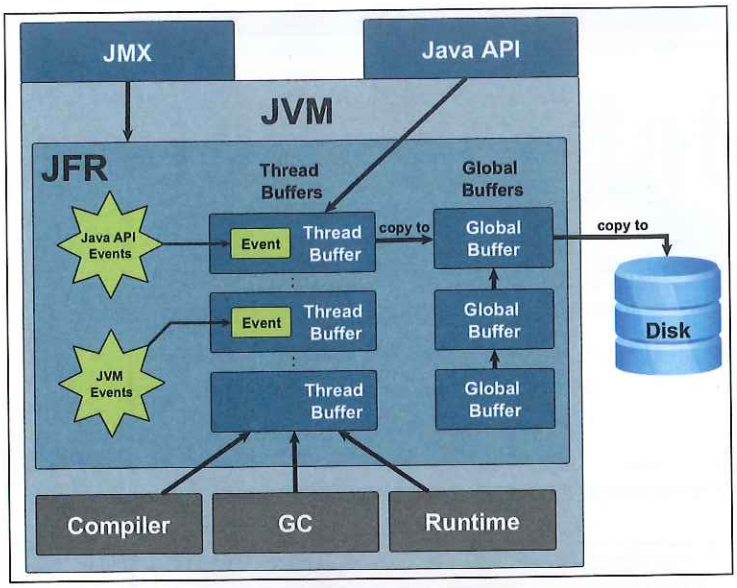

Abb. 2: Die Architektur des Java Flight Recordes

lector und Laufzeitumgebung, bekommt man eine einheitliche Sicht zur fokussierten Analyse und Diagnose. Aus der JMC-Konsole lässt sich über CREATE A NEW CONNECTION die Remote-Verbindung zu einer entfernten JVM herstellen, unter Angabe von Hostnamen, Port und Benutzer mit Passwort.

JFR-Aufzeichnungsbeispiele und Analyse

Es gibt zwei Arten von Aufzeichnungen. Die erste Variante besteht aus kontinuierlichen JFR-Aufnahmen ohne definiertes Ende. Man muss durch ein explizites Dump-Kommando oder mit dem Diagnostic-Kommando *JFR.dump* das laufende Flight Recording beenden. Dabei können die letzten Minuten vor einem Fehler ausgegeben werden. Die zweite Aufzeichnungsvariante besitzt eine feste Aufnahmezeit und wird als Profiling Recording bezeichnet. Falls diese aus der Java Mission Control Console gestartet wurde, wird die grafische Oberfläche automatisch mit der aufgezeichneten JFR-Datei geöffnet. Diese Variante wird bei Performance- und Lasttests empfohlen, um beispielsweise 30 Minuten aufzuzeichnen. Zum besseren Analyseverständnis ist es notwendig, die Anwendung zu kennen und zu wissen, ob hoher Durchsatz oder niedrige Latenzzeiten im Vordergrund stehen, oder wie hoch der Prozessorverbrauch tatsächlich sein soll. Zur Flight-Recording-Analyse be-

Listing 2: Codebeispiel „02_JFR_HotMethods HotMethods.java“

```
import java.io.IOException;
import com.oracle.jrockit.jfr.EventToken;
import com.oracle.jrockit.jfr.InstantEvent;
import com.oracle.jrockit.jfr.Producer;

public class HotMethods {
  // Hint: You may want to set NUMBER_OF_THREADS
  // close to the number of hardware threads for
  // maximum saturation
  private static final int NUMBER_OF_THREADS = 4;
  private static final String PRODUCER_URI = "http://
          www.oracle.com/jmc/tutorial/example2";
  static final Producer PRODUCER;
  static final EventToken WORK_EVENT_TOKEN;
  static {
    PRODUCER = createProducer();
    WORK_EVENT_TOKEN = createToken(WorkEvent.
                                    class);
    PRODUCER.register();
  }

  /**
   * Creates our event token.
   */
  public static EventToken createToken(Class<?
                    extends InstantEvent> clazz) {
    try {
      return PRODUCER.addEvent(clazz);
    } catch (Exception e) {
      // Add proper exception handling.
      e.printStackTrace();
    }
    return null;
  }

  /**
   * Creates our producer.
   */
  private static Producer createProducer() {
    try {
      return new Producer("Tutorial Example 2",
      "A demo event producer for the HotMethods
                          Example.", PRODUCER_URI);
    } catch (Exception e) {
      // Add proper exception handling.
      e.printStackTrace();
    }
    return null;
  }

  public static void main(String[] args) throws
                                      IOException {
    Thread[] threads = new Thread[NUMBER_OF_
                                      THREADS];
    for (int i = 0; i < threads.length; i++ ) {
      threads[i] = new Thread(new Worker(), "Worker
                                      Thread " + i);
      threads[i].setDaemon(true);
      threads[i].start();
    }
    System.out.print("Press enter to quit!");
    System.out.flush();
    System.in.read();
  }
}
```

Problem	Lösung
Lock contention	+ Monitor enter events
Excessive class loading	+ Class loading events
Expenses of redundant exception logging	+ Through method profiling information and lock contention information
Context switches	+ Through context switch rate events
Processor burn rate	+ Method profiling events
GC thrashing	+ All the various GC events

Tabelle 1: Beispielszenarien zur Diagnose interner JVM-Information mit JMC/JFR 5.5

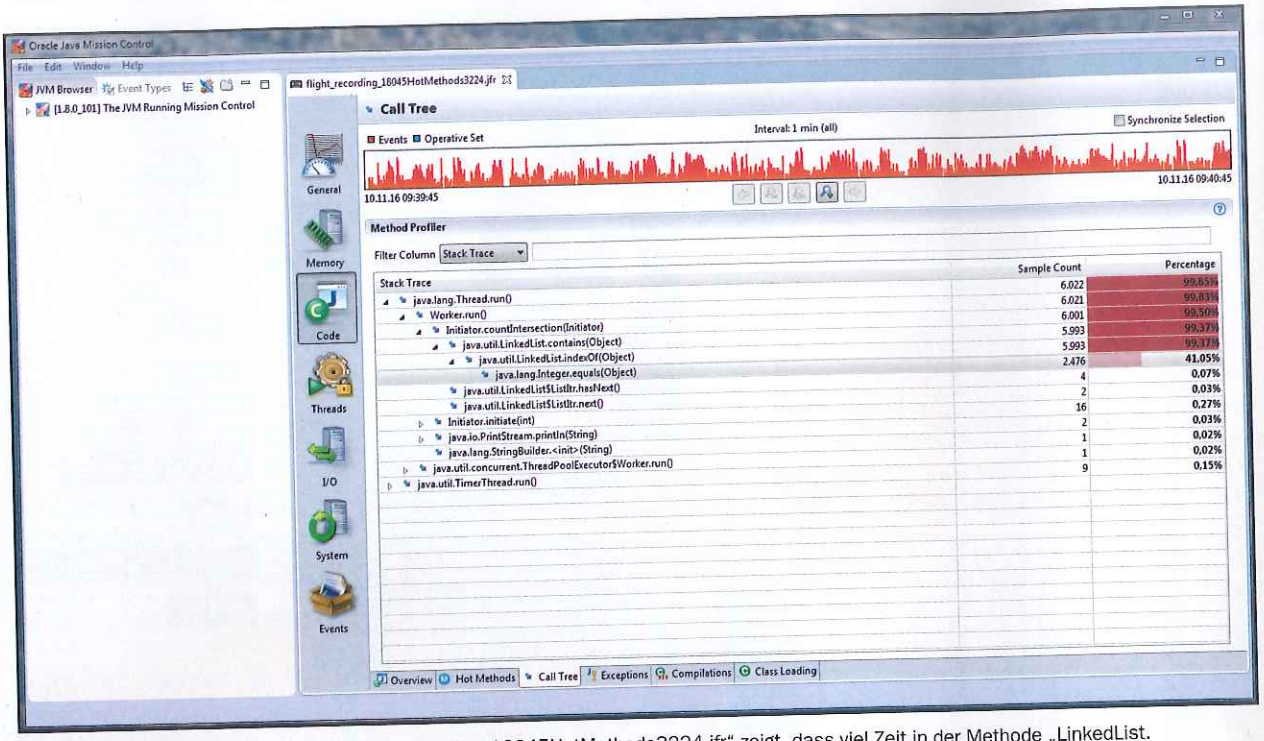

Abb. 3: Die JMC-Call-Tree-Ansicht für „flight_recording_18045HotMethods3224.jfr" zeigt, dass viel Zeit in der Methode „LinkedList.indexOf(Object)" verbracht wird

nötigt man ein gewisses Maß an Erfahrung und Übung im Umgang mit den Werkzeugen (Tabelle 1). Die praktischen Übungen aus dem Java Mission Control Tutorial [7] von Marcus Hirt [8] verdeutlichen den Umgang mit dem Java Flight Recorder.

Lässt man das Beispiel *02_JFR_HotMethods* mit Eclipse laufen (Listing 2) und startet den Flight Recorder, erkennt man in der Aufzeichnung *flight_recording_18045HotMethods3224.jfr* (**Abb. 3**), dass viel Zeit in der Methode *LinkedList.indexOf(Object)* verbracht wird. Schaut man sich die Aufzeichnung in der JMC-Konsole unter dem Menüpunkt CODE und HOT METHODS an, erkennt man, dass die Methode *contains* durchlaufen wird. *contains* verhält sich in einer *LinkedList* proportional zur Anzahl der Einträge. Durch ein *HashSet* könnte dies in kürzerer Zeit ablaufen. Mit einem Blick auf die JMC-Konsole unter dem Menüpunkt CODE und der Ansicht CALL TREE kann man herausfinden, wo *contains* aufgerufen wird. Folgt man im STACK TRACE dem größten Ressourcenverbrauch von 99,85 Prozent bei *java.lang.Thread.run()*, gelangt man zu *java.util.LinkedList.indexOf(Object)* mit einem hohen Ressourcenverbrauch von 99,37 Prozent. Ersetzt man im Programmcode *Initiator.java* (Listing 3) die Zeile *collection = new LinkedList<Integer>();* mit *collection = new HashSet<Integer>();*, läuft das Programmbeispiel *JFR_HotMethods* viel schneller und spart Ressourcen, wie in der Aufzeichnung *flight_recording_18045HotMethods948.jfr* in **Abbildung 4** dargestellt.

Das nächste Codebeispiel verdeutlicht das Allokationsverhalten, hervorgerufen durch eine ungünstige Auswahl von Datentypen. Wird das Beispiel *04_JFR_GC* mit dem Programm *Allocator.java* (Listing 4) in der

Listing 3: Codebeispiel „02_JFR_HotMethods Initiator.java"

```
import java.util.*;

public class Initiator {
  private Collection<Integer> collection;

  public Initiator() {
    collection = new LinkedList<Integer>();
  // collection = new HashSet<Integer>();
  }

  // Hint: This creates a list of unique
  // elements!

  public void initiate(int seed) {
    collection.clear();
    for (int i = 1; i < 10000; i++) {
      if ((i % seed) != 0)
        collection.add(i);
    }
  }

  protected Collection<Integer> getCollection()
  {
    return collection;
  }

  public int countIntersection(Initiator other) {
    int count = 0;
    for (Integer i : collection) {
      if (other.getCollection().contains(i)) {
        count++;
      }
    }
    return count;
  }
}
```

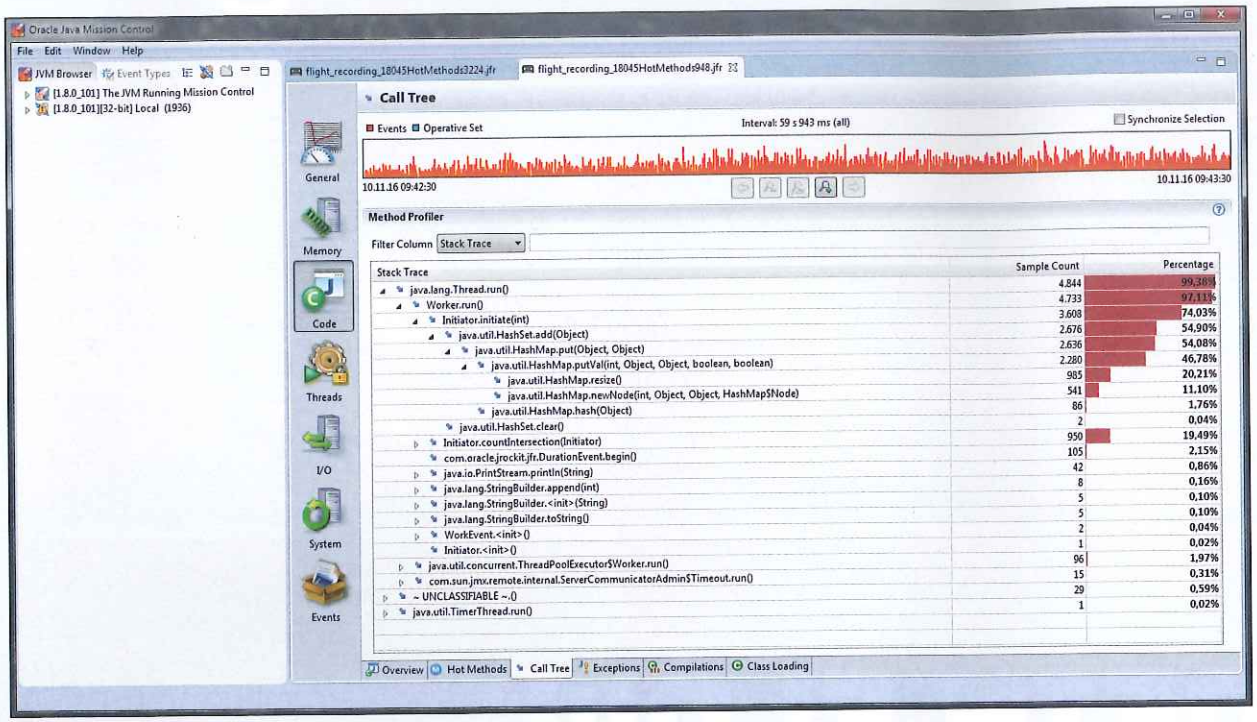

Abb. 4: Nach der Codeänderung ist das Programm schneller und spart Ressourcen, wie die Call-Tree-Ansicht für „flight_recording_18045HotMethods948.jfr" zeigt

Eclipse-IDE gestartet und aktiviert man den Flight Recorder, erkennt man in der JMC-Konsole unter dem Menüpunkt MEMORY und GARBAGE COLLECTION einen extrem breiten ansteigenden Sägezahnverlauf in der Aufzeichnung *flight_recording_18045Allocator6764.jfr* (**Abb. 5**), der den Used-Heap-Verbrauch mit ca. 22 MB zeigt. Nun stellt sich die Frage, welche Objekte am häufigsten allokiert werden. Es sind reichlich Integer-Allokationen vorhanden. Daraus ergibt sich die nächste Frage: Was hat die Allokationen und die Garbage Collections verursacht? Die JMC-Console-Ansicht CODE, CALL TREE und STACK TRACE mit *java.util.HashMap$ValueIterator.next()* lässt erahnen, dass die Allokationen durch *Collection<MyAlloc> myAllocSet = map.values();* verursacht worden sein könnten, weil *Allocator* ständig den Primitive Type *int* zum Object Type *Integer* umgewandelt hat, siehe auch Codezeile *if (!map.containsKey(c.getId()))* in Listing 4. Falls Primitive Types als Index in HashMaps verwendet werden, ist es besser, die Object-Type-Version zu speichern als ständig zwischen dem Primitive Type *int* und dem Object Type *Integer* hin und her zu wechseln. Ersetzt man jeden Primitive Type *int* mit dem Object oder Reference Type *Integer* in der Klasse *MyAlloc*, reduziert sich die Anzahl der Garbage Collections erheblich. Nach der Codeänderung in *Allocator.java* (Listing 5) zeigt die Aufzeichnung *flight_recording_18045Allocator1432.jfr* (**Abb. 6**) den ansteigenden Sägezahnverlauf, aber ohne die zuvor

Listing 4: Codebeispiel „04_JFR_GC Allocator.java" mit „int" und „Integer"

```
import java.util.Collection;
import java.util.HashMap;

public class Allocator {

  private HashMap<Integer, MyAlloc> map = new
                    HashMap<Integer, MyAlloc>();

  private static class MyAlloc {
    private int id;

    private MyAlloc(int id) {
      this.id = id;
    }
    private int getId() {
      return id;
    }
  }

  public static void main(String[] args) {
    new Allocator().go();
  }

  private void go() {
    alloc(10000);
    long yieldCounter = 0;
    while(true) {
      Collection<MyAlloc> myAllocSet = map.values();
      for (MyAlloc c : myAllocSet ) {
        if (!map.containsKey(c.getId()))
          System.out.println("Now this is strange!");
        if (++yieldCounter % 1000 == 0)
          Thread.yield();
      }
    }
  }

  private void alloc(int count) {
    for (int i = 0; i < count; i++) {
      map.put(i, new MyAlloc(i));
    }
  }
}
```

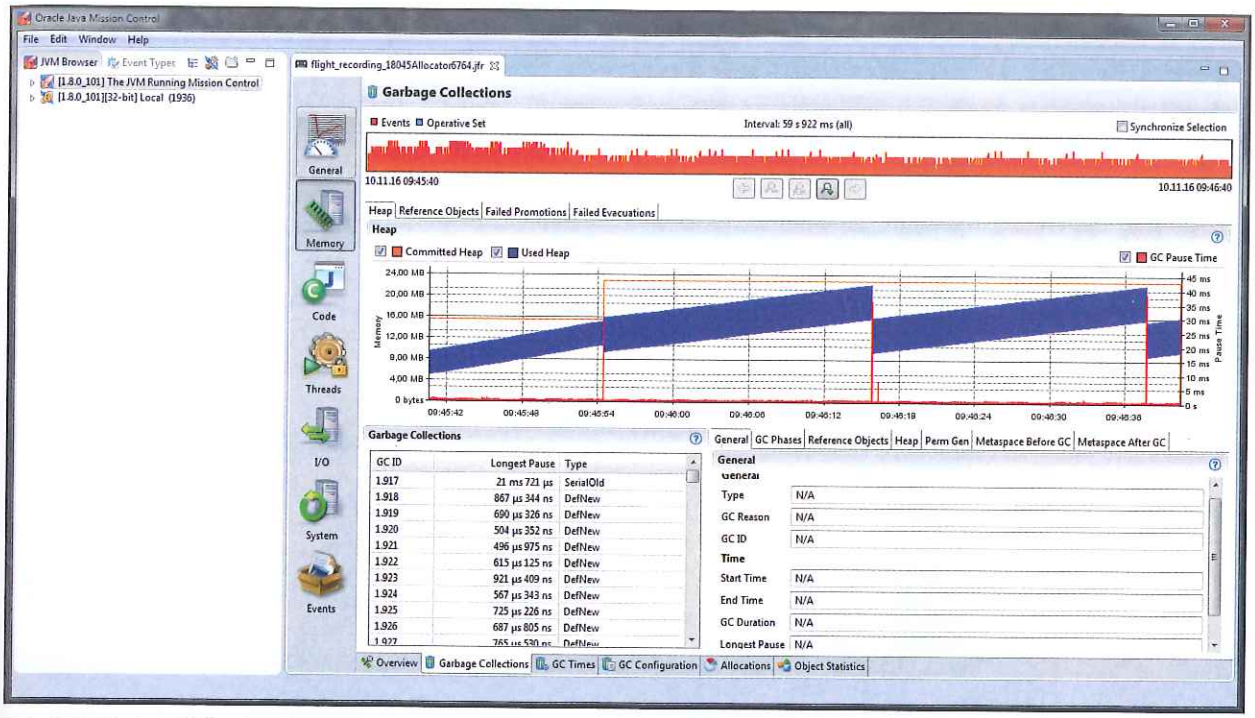

Abb. 5: Die Garbage-Collections-Ansicht für „flight_recording_18045Allocator6764.jfr" zeigt einen steilen Sägezahnanstieg

sichtbare breite Verzerrung, und der Used-Heap-Verbrauch hat sich auf rund 8 MB verringert.

Fazit und Ausblick

Die Verwendung der Werkzeuge Java Mission Control und Java Flight Recorder bringt einige Vorteile, weil man schneller und strukturiert potenzielle Fehler erkennt. Aufgrund der gemeinsamen Datenbasis mit JFR-Dateien können Entwickler, Administratoren und Supportmitarbeiter während des gesamten Applikationslebenszyklus besser zusammenarbeiten, ohne lästiges Verschieben von Zuständigkeiten oder abgegrenzten Verantwortungsbereichen. Voraussetzung für ein erfolgreiches Zusammenspiel der unterschiedlichen IT-Rollen ist, dass jeder in seinem Gebiet genügend Erfahrungen mit den eingesetzten Unternehmensanwendungen inklusive den passenden Werkzeugen sammeln kann und sich strukturiert mit seinen Kollegen austauschen kann. Mit der zunehmenden Komplexität der Anwendungslandschaft wächst auch die Werkzeugausstattung von Entwicklern und Administratoren in den Unternehmen. Dass die Werkzeuge Bestandteil vom Oracle-Java-SE-Advanced-Support sind [3], ist oft eine Hürde. Die Unternehmen, die sich einer notwendigen Java-Werkzeugausstattung für ihre IT-Umgebung bewusst sind, wählen zwischen Third-Party-Werkzeugen und den JMC-/JFR-Werkzeugen aus und haben auch ein Budget dafür eingeplant. Im anderen Fall versuchen Un-

Listing 5: Codebeispiel „04_JFR_GC Allocator.java" mit Integer

```
import java.util.Collection;
import java.util.HashMap;

public class Allocator {

  private HashMap<Integer, MyAlloc> map = new
                    HashMap<Integer, MyAlloc>();

  private static class MyAlloc {
    private Integer id;

    private MyAlloc(Integer id) {
      this.id = id;
    }
    private Integer getId() {
      return id;
    }
  }

  public static void main(String[] args) {
    new Allocator().go();
  }

  private void go() {
    alloc(10000);
    long yieldCounter = 0;
    while(true) {
      Collection<MyAlloc> myAllocSet = map.values();
      for (MyAlloc c : myAllocSet ) {
        if (!map.containsKey(c.getId()))
          System.out.println("Now this is strange!");
        if (++yieldCounter % 1000 == 0)
          Thread.yield();
      }
    }
  }

  private void alloc(Integer count) {
    for (Integer i = 0; i < count; i++) {
      map.put(i, new MyAlloc(i));
    }
  }
}
```

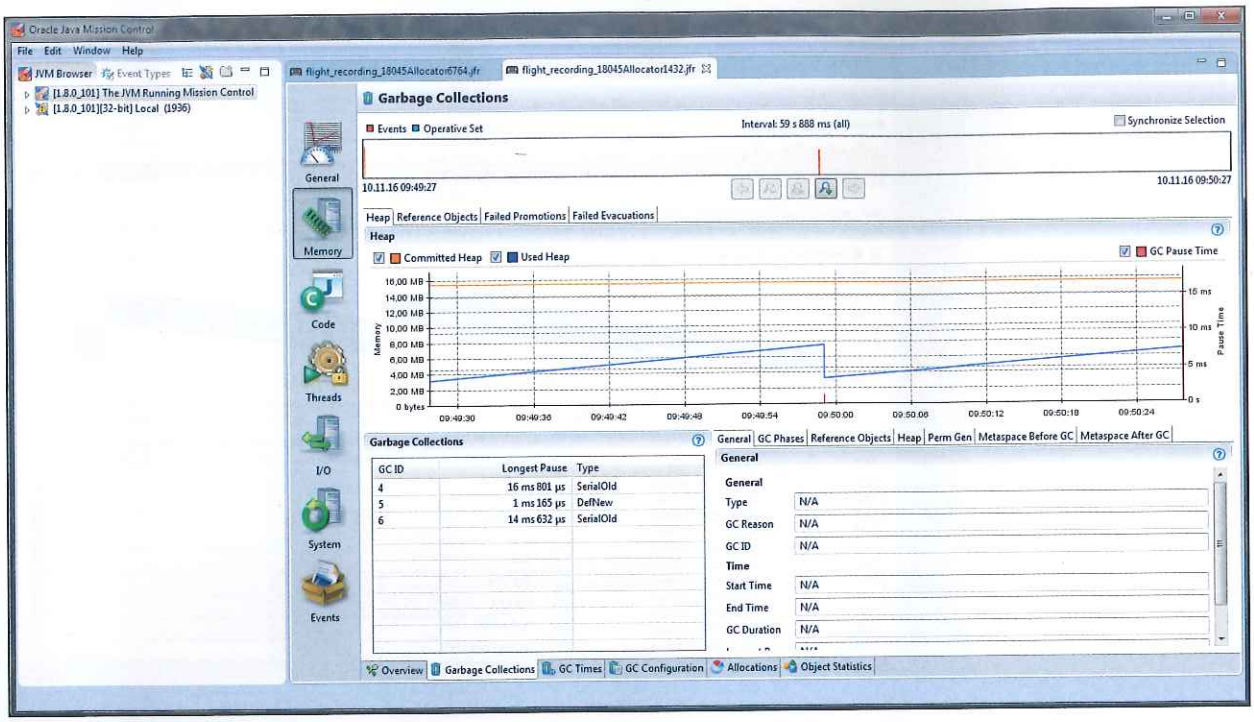

Abb. 6: „flight_recording_18045Allocator1432.jfr" zeigt den Sägezahnverlauf, aber ohne Verzerrungen, und der Used Heap hat sich verringert

ternehmen ohne Werkzeugbeschaffung auszukommen oder jeweils die einzelnen Entwickler und Administratoren in die Pflicht zu nehmen, oft mit positiven Effekten beim Einsatz von Open-Source-Werkzeugen. Aber auch mit den damit verbundenen Mehraufwendungen bei fehlenden Merkmalen oder kritischen Problemfällen, falls Performanceexperten als Feuerwehr ad hoc in die Unternehmen kommen müssen. In jedem Fall sind Kosten zu berücksichtigen, einerseits für die nötigen Werkzeuge und andererseits für Consulting.

Hinter den Werkzeugen Java Mission Control und Java Flight Recorder steht ein Entwicklungsteam, das größtenteils in Stockholm arbeitet und die Werkzeuge kontinuierlich weiterentwickelt. Die Roadmap sieht für JMC 6.0 mit JDK 9 eine automatische Analyse der Flight Recordings vor, und die Benutzeroberfläche soll modernisiert werden. Außerdem sollen neue Eventtypen kommen:

- Detaillierte Safe-Point-Informationen
- Detaillierte Code-Cache-Informationen
- Compilerevents mit detaillierten Inlining-Informationen
- Garbage-First-Informationen mit verbesserter Zustandsvisualisierung der Memory-Regionen
- Modulevents für geladene Module
- Unterstützung von nativen geladenen Bibliotheken mit periodischen Events

Zu den Verbesserungen zählen auch neue APIs, die bei der Verwendung von Custom-Events unterstützen. Der programmatische Zugriff zum Lesen von Flight Recordings und zur Kontrolle vom Flight Recorder sowie die Modularisierungsunterstützung mit kleineren Profilen werden einfacher zu benutzen sein. Zudem wird der Namespace von *oracle.jrockit.** nach *jdk.jfr.** geändert. Letztlich steht der JMC-JFR-Memory-Leak-Detektor unter Vorbehalt auf der Wunschliste für die Roadmap. Die geplante Auslieferung von JMC 6.0 ist mit der JDK-9-Releasefreigabe [9] für den 27. Juli 2017 vorgesehen.

Wolfgang Weigend arbeitet als Sen. Leitender Systemberater bei der Oracle Deutschland B.V. & Co. KG. Er beschäftigt sich mit Java-Technologie und Architektur für unternehmensweite Anwendungsentwicklung.

Links & Literatur

[1] Java Mission Control 5.5 Release Notes: http://www.oracle.com/technetwork/java/javase/jmc55-release-notes-2412446.html

[2] Oracle Java Mission Control 5.5 Certified System Configurations: http://www.oracle.com/technetwork/java/javaseproducts/documentation/jmc-5-5-certified-system-config-2432060.html

[3] Java Components: https://docs.oracle.com/javacomponents/index.html

[4] Oracle Java SE and Oracle Java Embedded Products: http://www.oracle.com/technetwork/java/javase/terms/products/index.html

[5] Oracle Binary Code License Agreement for the Java SE Platform Products and JavaFX: http://www.oracle.com/technetwork/java/javase/terms/license/index.html

[6] Java Platform, Standard Edition Java Flight Recorder Runtime Guide: https://docs.oracle.com/javacomponents/jmc-5-5/jfr-runtime-guide/toc.htm

[7] JMC Tutorial: http://hirt.se/downloads/oracle/jmc_tutorial.zip

[8] Blog von Marcus Hirt: http://hirt.se/blog/

[9] JDK 9: http://openjdk.java.net/projects/jdk9/