Agon solutions gmbh

Mai 2013

PERFORMANCEFALLE BEIM LOGGING: Zeitstempel-Formatierung

Dr. Jürgen Lampe
Agon Solutions

Abstract

An Hand eines Beispiels aus dem Projektalltag wird beschrieben, wie eine als Folge umfangreicher und geschwätziger Log-Ausgaben hervorgerufene Belastungsspitze beseitigt werden kann, bevor sie zum ernsthaften Problem für die Gesamtanwendung wird.
Dieses Whitepaper besteht aus zwei Teilen: Durch das im ersten Teil erläuterte gleichwertige Profiling von Laufzeitverhalten und Speicherverbrauch werden kritische Code-Bereiche identifiziert und durch geringfügige Änderungen entschärft. Die dabei bestätigte Erfahrung, dass das Einsparen von Objekterzeugungen die Gesamtleistung überproportional verbessern kann, wird im zweiten Teil einer genaueren Analyse unterzogen.

Teil 1

Das praktische Problem

Irgendwann ist jede erfolgreiche Applikation so gewachsen, dass ihre Performance durch zunehmende Nutzung und Funktionserweiterungen an Grenzen stößt. Sobald die relativ kostengünstigen Möglichkeiten leistungsfähigere Hardware einzusetzen ausgereizt sind, bleibt nichts anderes übrig, als das gesamte System einer kritischen Schwachstellenanalyse zu unterziehen.

Im betrachteten Fall handelt es sich um einen großen kommerziellen Applikationsserver mit mehr als 100 Anwendungen, der bereits auf einer sehr elaborierten Hardware läuft. Wie so oft hatten sich die Performanceprobleme auf diesem Server schleichend entwickelt. Updates mit neuen Funktionen brachten zusätzliche Belastungen. Allerdings konnten durch Änderungen in der Konfiguration auch immer wieder Verbesserungen erreicht werden, ohne dabei jedoch den langfristigen Trend der wachsenden Systembelastung nachhaltig entgegen zu wirken.

Bei Beginn der Schwachstellenanalyse war bereits bekannt, dass die sich abzeichnenden Probleme mit hoher Wahrscheinlichkeit durch das Logging verursacht werden. Indikatoren für diese Vermutung waren:

- die beobachtete und mit der Serverlast korrelierte Verzögerung bei der Ausgabe der Log-Ereignisse
- die Tatsache, dass Änderungen in der Verwaltung der Log-Dateien (schnelleres Filesystem) bereits eine Verbesserung gebracht hatten

Auf Grund rechtlicher Bedingungen, die für diese Anwendung eine lückenlose Nachvollziehbarkeit aller relevanten Aktionen erfordern, müssen im laufenden Produktionsbetrieb des Fallbeispieles sehr umfangreiche Logs geschrieben werden. Das für den Debug-Modus bekannte Anti-Pattern, das darauf hinweist, `log.debug(…)`-Anweisungen in ein `if(log.debugEnabled()){log.debug(…);}` einzuschließen, wurde bereits berücksichtigt. Die Last wurde durch Info-Level-Ausgaben verursacht, die nicht abgeschaltet

werden dürfen. Betrachten wir nun die einzelnen Schritte der Schwachstellenanalyse genauer.

Laufzeit-Messung

Es ist ein bewährter Grundsatz, Code-Optimierungen nur auf der Basis von Messungen vorzunehmen und den Erfolg ebenfalls durch Messungen zu verifizieren. Das ist leichter gesagt als getan. Die heute üblichen Prozessoren mit mehreren Kernen, Berechnungs-Pipelines und hierarchischen Cache-Strukturen machen es nahezu unmöglich, reproduzierbare Daten zu gewinnen. Java-Entwickler sehen sich einer weiteren Schwierigkeit gegenüber, die durch die just-in-time (JIT) kompilierenden virtuellen Maschinen (VM) verursacht wird. Die bekannte HotSpot-VM führt außerdem Optimierungen wie Inlining (Ersetzung durch Einkopieren) von Methodenaufrufen oder Loop-unrolling auf der Basis statistischer Daten aus, die nicht nur zu stark variierenden Ausführungszeiten führen, sondern auch Profiler bei der Datengewinnung beeinträchtigen, da beispielsweise einkopierte Methodenaufrufe praktisch unsichtbar werden.
Weitere Probleme ergeben sich, wenn – wie häufig und auch im vorliegenden Fall – das Profiling nicht direkt in der Produktionsumgebung möglich ist. Ein erster Ansatz kann dann darin bestehen, häufig aufgerufene Codebereiche zu identifizieren. Zu diesem Zweck sollte auf dem Testsystem die JIT-Funktion deaktiviert werden, um Verfälschungen zu vermeiden. Mit etwas Glück liefert eine anschließende Code-Inspektion der gefundenen Bereiche Ansatzpunkte für Optimierungen. Man sollte diesen Weg aber nur dann weiter verfolgen, wenn die erkannten Defizite wirklich gravierend sind und prinzipiell nicht durch die VM behoben werden können. Ansonsten besteht die große Gefahr, Zeit und Kraft in Arbeiten zu stecken, deren Auswirkungen minimal sind, oder gleich null, wenn im Produktionsbetrieb eine gleichwertige Beschleunigung durch die JIT-VM bereits erreicht wird.
Im vorliegenden Fall weisen die mit einem kommerziellen Profiler-Programm gewonnenen Daten auf einen Schwerpunkt bei der Aufbereitung der Log-Ereignisse hin, insbesondere bei der Formatierung, ohne jedoch klare Favoriten für die weitere Analyse auszuweisen.

Objekterzeugung analysieren

In anderen durchgeführten Projekten hatte es sich dabei als überraschendes Ergebnis immer wieder gezeigt, dass Verbesserungen der Speicherökonomie gleichzeitig das Laufzeitverhalten verbessern. Dies steht im gewissen Widerspruch zu der verbreiteten Überzeugung, dass sich Speicherbedarf und Laufzeit antagonistisch verhalten und somit die Verbesserung des einen auf Kosten des anderen geht. Zumindest für Java-Programme gilt die letzt genannte Regel nicht mehr uneingeschränkt:

- Erstens kostet die Objekterzeugung Laufzeit, und ein Objekt kommt selten allein, oft müssen weitere (Member-)Objekte erzeugt und initialisiert werden.

- Zweitens bedeuten mehr Objekte auch mehr Arbeit für den Garbage-Collector (GC) – schon während ihrer Lebenszeit sind mehr erreichbare Objekte zu traversieren. Gleichzeitig wird der Heap schneller gefüllt, was häufigere GC-Aktivitäten verursacht.

Eine erfolgversprechende Strategie besteht deshalb darin, Hotspots der Objekterzeugung zu suchen. Das ist wesentlich einfacher als die im Teil 2 beschriebene direkte Laufzeitanalyse, weil Hardware und VM praktisch keinen Einfluss auf die Objekterzeugung nehmen. (Und wenn kurzlebige Objekte tatsächlich im Stack angelegt werden und uns entgehen, so ist das eine begrüßenswerte Optimierung, aber für diese Analyse unerheblich.)

Ein besonderes Augenmerk haben bei dieser Analyse String-Objekte verdient. Die bequeme Handhabung von Zeichenketten ist einer der großen Vorzüge von Java. Aber die Erzeugung von Strings erfordert (außer wenn sie als Teilstrings mittels *substring*-Methode entstehen) jeweils die Allokation eines eigenen internen *char*-Arrays sowie das Kopieren der Zeichen in dieses Feld. Letzteres ist eine Operation, deren Kosten mit der Länge des Strings zunehmen.

Engpass: Formatierung der Log-Ausgaben

Auch im vorgestellten Fall waren an dem Performanceproblem String-Objekte wesentlich beteiligt. Die Durchsicht der Allokation-Profile deutete auf hohe Last beim Formatieren des Zeitstempels der *LoggingEvent*s innerhalb des verwendeten Log4j-Frameworks [1] hin. Dabei geht es nicht um eine bestimmte Software, sondern darum, welche Auswirkungen kleine, scheinbar belanglose Unterschiede haben können.
Um nachzuweisen, dass die Formatierung des Zeitstempels für die beobachtete Last verantwortlich ist, wurde sie durch eine eigene Formatierungsroutine ersetzt. Die Erstellung eines eigenen Konverters ist recht einfach. Listing 1 zeigt die beiden benötigen Klassen *FastDatePatternParser* und *FastDatePatternLayout*, die fast nur aus Verwaltungsaktionen bestehen. Die eigentliche Funktion verbirgt sich ganz unten in der Methode *convert*.
Damit dieser Konverter verwendet wird, muss er in die Konfiguration aufgenommen werden, zum Beispiel durch die folgenden Zeilen in der *log4j.xml* innerhalb eines Appender-Elements:

```
<layout class="de.agon.util.log.FastDatePatternLayout">
  <param name="ConversionPattern" value=”%d - %m%n"/>
</layout>
```

Listing 1

```
public class FastDatePatternLayout extends PatternLayout {
  public FastDatePatternLayout() {
      this(DEFAULT_CONVERSION_PATTERN);
  }

  public FastDatePatternLayout(String pattern) {
      super(pattern);
  }

  @Override
  public PatternParser createPatternParser(String pattern) {
      if (pattern == null) {
          return new
FastDatePatternParser(DEFAULT_CONVERSION_PATTERN);
      } else {
          return new FastDatePatternParser(pattern);
      }
    }
}
```

```
////////////////////////////////////////////////////////////
```

```
public class FastDatePatternParser extends PatternParser {
  public FastDatePatternParser(String pattern) {
    super(pattern);
  }
  @Override
  protected void finalizeConverter(char c) {
  if (c == 'd') {
      String dateFormatStr= "yyyy-MM-dd HH:mm:ss,SSS";
      String dOpt= extractOption();
      if ("ABSOLUTE".equals(dOpt)) {
        dateFormatStr= "HH:mm:ss,SSS";
      } else if ("DATE".equals(dOpt)) {
        dateFormatStr= "dd MMM yyyy HH:mm:ss,SSS";
      } else if ("ISO8601".equals(dOpt)) {
      } else if (dOpt != null) {
        dateFormatStr= dOpt;
      }
      PatternConverter pc= new
FastDatePatternConverter(formattingInfo, dateFormatStr);
      currentLiteral.setLength(0);
```

```
      addConverter(pc);
    } else {
      super.finalizeConverter(c);
    }
  }
  private static class FastDatePatternConverter extends
PatternConverter {
    final private String format;

    FastDatePatternConverter(FormattingInfo formattingInfo, String
format) {
      super(formattingInfo);
      this.format= format;
    }
    @Override
    public String convert(LoggingEvent event) {
      return CurrentDateProvider.formatTimeMillis(
event.getTimeStamp(), format);
    }
  }
}
```

Immer gleich: das aktuelle Datum

Die Umwandlung der Millisekunden eines Zeitstempels erfolgt in zwei Schritten. Im ersten Schritt muss wegen der bekannten Unregelmäßigkeiten durch Schaltjahre und Zeitumstellungen der größere Aufwand getrieben werden, um das - immer wieder gleiche - Tagesdatum zu berechnen. Die Konvertierung der Uhrzeit ist der zweite Schritt und hierzu vergleichsweise einfach. und Daher wird zunächst am Datum angesetzt.
Im beschriebenen Fall existierte mit der Klasse *CurrentDateProvider* dafür bereits eine Lösung, da auch an anderen Stellen das aktuelle Datum als Zeichenkette, mal mit und mal ohne Uhrzeit, häufig benötigt wird.
Listing 2 zeigt die hier interessierenden Teile des Codes. Für den aktuellen Tag wird jeweils eine Instanz angelegt, die die Millisekundenwerte für den Tagesbeginn (*intervalStart*) und das Tagesende (*intervalEnd* = Tagesbeginn des Folgetages) und die numerischen Zeichen des Datums (*dateText*) enthält. Daneben wird aus der Tageslänge noch ein Indikator für die beiden Tage der Sommer-Winterzeit-Umschaltung bestimmt (*isDSTswitchDay* mit den möglichen Werten -1, 0 und +1).
Die Formatierung des Zeitstempels reduziert sich so auf das Kopieren des Datums. Lediglich die Uhrzeit muss jeweils neu konvertiert werden. Das leisten wenige Ganzzahloperationen. Zugunsten der Performance wurden die möglichen Formate auf rein numerische eingeschränkt. Außerdem muss der Formatstring genau so viele Zeichen enthalten wie das erwartete Ergebnis. Für den Einsatz beim Logging ergab sich aus diesen Einschränkungen

aber kein Hindernis. Die Methode ist wiedereintrittsfähig (re-entrant), d. h. *thread-safe*, und erzeugt beim Aufruf nur ein einziges Objekt – das Ergebnisarray.
Eine weitere Verbesserung für das Logging wäre eventuell dadurch zu erreichen, dass der *CurrentDateProvider* um eine Methode ohne Format-Parameter ergänzt wird, die nur das im Log benötigte Format liefert. (In der Praxis wird dieses Format fast nie geändert, da es beispielsweise Änderungen an den Skripten zur Auswertung zur Folge hätte.) Allerdings kann die Anzahl der erzeugten Objekte dadurch nicht mehr reduziert werden und die zu erwartende minimale Beschleunigung dürfte den Aufwand kaum rechtfertigen. Bei einer Neuimplementierung sollte dieser Ansatz aber in Erwägung gezogen werden.
Über den hier gezeigten Code hinaus enthält diese Klasse noch eine Reihe von Convenience-Methoden, die beispielsweise Instanzen des aktuellen Datums in verschiedenen Formaten aus dem Cache liefern. Schließlich wird beim Tageswechsel jeweils eine neue Instanz angelegt.

Listing 2

```
/** Tagesgrenzen */
private final long intervalStart, intervalEnd;
/** Cache fuer das aktuelle Datum in der Form ddMMyyyy */
private final char[] dateText = new char[8];

protected final char[] formatToCharArrayIntern(long time, String
format) {
 if (time < intervalEnd && time >= intervalStart) {
    int millisecs= (int) (time - intervalStart);
   if (isDSTswitchDay != 0) {
     if (isDSTswitchDay < 0 && millisecs > 7200000L) {
                millisecs+= 3600000L;
     } else if (isDSTswitchDay > 0 && millisecs > 10800000L) {
        millisecs-= 3600000L;
     }
  }
  char[] charArray= format.toCharArray();
  int yearCount= 0;
  int monthCount= 0;
  int dayCount= 0;
  int hours= millisecs / 3600000;
  millisecs%= 3600000;
  int minutes= millisecs / 60000;
  millisecs%= 60000;
  int secs= millisecs / 1000;
  int millis= millisecs % 1000;

  char[] dateAsText= dateText;
  for (int i= 0, len= charArray.length; i < len; i++) {
    switch (charArray[i]) {
```

```
      case 'y':
        if (yearCount < 2) {
           charArray[i]= dateAsText[6 + yearCount++];
        } else if (yearCount++ == 2) {
          charArray[i - 2]= dateAsText[4];
          charArray[i - 1]= dateAsText[5];
          charArray[i]= dateAsText[6];
        } else {
          charArray[i]= dateAsText[7];
        }
        break;
      case 'M':
        charArray[i]= dateAsText[2 + monthCount++];
        break;
      case 'd':
        charArray[i]= dateAsText[0 + dayCount++];
        break;
      case 'H':
        charArray[i]= (char) ('0' + hours / 10);
        hours= hours % 10 * 10;
        break;
      case 'm':
        charArray[i]= (char) ('0' + minutes / 10);
        minutes= minutes % 10 * 10;
        break;
      case 's':
        charArray[i]= (char) ('0' + secs / 10);
        secs= secs % 10 * 10;
        break;
      case 'S':
        charArray[i]= (char) ('0' + millis / 100);
        millis= millis % 100 * 10;
        break;
      default:
      }
    }
    return charArray;
  } else return null;
}
```

Ergebnis

Bei der Bewertung des Ergebnisses müssen die Ausgangsbedingungen in Betracht gezogen werden, da sie nach Erfahrungen des Autors repräsentativ sind:

- Die Entwicklung einschließlich der Entwickler- und Integrationstests erfolgt auf einem Standard-PC.
- In Produktion läuft die Anwendung auf gänzlich anderen Systemen bei einem externen Hoster.
- Die Überwachung der Leistung erfolgt anhand regelmäßig gelieferter und vertraglich vereinbarter Auswertungen der Logs. Daraus lassen sich Hinweise auf kritische Punkte gewinnen, ein zielgerichtetes Profiling unter Produktionsbedingungen ist jedoch unmöglich.
- Sich eventuell abzeichnende Probleme sollen so rechtzeitig beseitigt werden, dass eine Eskalation, die andererseits wiederum die Voraussetzung dafür wäre, den Hoster mit dem Profiling zu beauftragen, vermieden wird.

In dieser Situation kann nur auf der Basis von Erfahrungen und Tests auf den Entwicklungs- bzw. Testsystemen nach Verbesserungen gesucht werden. Dies war beim beschriebenen Performance-Problem auch der Fall.
Die Veränderungen zeigten sowohl im Performance- als auch im Speicherprofiling sichtbare Ergebnisse und wurden demzufolge in die Produktion übernommen. Dies ist immer risikobehaftet, allerdings war aufgrund der Tatsache, dass die Anzahl der auszuführenden Bytecode-Instruktionen und der erzeugten Objekte verringert wurden, zumindest keine Verschlechterung zu erwarten. Tatsächlich sind die Auswirkungen auf die Gesamt-Anwendung noch erfreulicher als erwartet. Die Serverlast hat sich messbar reduziert und Verzögerungen zwischen *LoggingEvent*-Generierung und der Protokollierung treten so gut wie nicht mehr auf. Mittlerweile ist dieser Fix über ein Jahr erfolgreich und völlig unauffällig im Einsatz.
Damit hat sich erneut gezeigt, dass Sparsamkeit beim Erzeugen von Objekten einen deutlich über die unmittelbar betroffenen Programmteile hinausgehenden positiven Effekt hat. Die Verschlechterung der Code-Dichte durch angepasstes Ausprogrammieren von eigentlich in Bibliotheksmethoden vorhandenen Funktionen konnte dabei auf einen sehr kleinen und gut austestbaren Bereich beschränkt werden (*CurrentDateProvider*). Tatsächlich war dieser Code ja sogar schon vorhanden.

Teil 2

Nachträgliche Analyse

Das im ersten Teil beschrieben Vorgehen zeigt die Arbeit im Projektalltag, wo es vorrangig darum geht, ein gestecktes Ziel mit beschränktem Aufwand zu erreichen. Im zweiten Teil sollen nun die erwähnten Erfahrungen und Heuristiken besser begründet werden, um hier die Teile der Analyse nachzuliefern, für die im Projektalltag meist die Zeit fehlt. Dabei wird der seinerzeit erzeugte Code durchaus kritisch inspiziert.

Im Folgenden werden vier Varianten der Formatierung verglichen.

- NORMAL

Als Ausgangspunkt mit dem Namen *NORMAL* dient die Implementierung, die jeden Zeitstempel mit Hilfe von *java.text.SimpleDateFormat* aufbereitet. Das ist die Log4j-Standardimplementierung, wenn im *ConversionPattern* explizit ein Muster für die Zeitstempelformatierung angegeben wird, das %d also nicht allein steht, sondern zum Beispiel in der Form *%d{dd.MM.yyyy}*.

- FAST

Unter dem Namen *FAST* wird die im Teil 1 beschriebene verbesserte Variante betrachtet. Auf Grund der erläuterten speziellen Implementierung ist es dabei unerheblich, in welcher Form das Format spezifiziert wird.

- ISO8601

Bei genauem Lesen der Log4j-API-Dokumentation findet sich ein wichtiger Hinweis: „*For better results it is recommended to use the log4j date formatters. These can be specified using one of the strings "ABSOLUTE", "DATE" and "ISO8601"…* “[2] Da das ISO8601-Format dem teilweise vor der Optimierung verwendeten entspricht (es ist das Defaultmuster, wenn *%d* allein steht), wurde die entsprechende Konvertierung als *ISO8601* in den Vergleich aufgenommen. Der zugehörige Konverter hält nicht nur den Datumsteil des formatierten Zeitstempels in einem Cache vom Typ *char[]*, sondern alle Zeichen außer den Sekundenbruchteilen, die bei jedem Aufruf neu konvertiert werden. Zusammen mit den Zeichen steht der dazugehörige Sekundenwert im Cache, um entscheiden zu können, wann eine neue Konvertierung des vorgehaltenen Teils notwendig ist. Diese Arbeitsweise ist für die Formatierung sehr schnell aufeinanderfolgender Zeitstempel günstig, verliert ihre Vorteile aber, wenn die Zeitstempel ungeordnet sind oder nur wenige pro Sekunde vorliegen. Nebenbei bemerkt, ergab eine kleine (nichtrepräsentative) Umfrage, dass unter den Log4j-Anwendern vielen unbekannt ist, dass zwischen den Formaten *%d*, bzw. *%d{ISO8601}* einerseits und *%d{yyyy-MM-dd HH:mm:ss,SSS}* andererseits ein erheblicher Performance-Unterschied besteht. (Im vorliegenden Projekt kamen diese Formen ebenfalls vermischt zur Anwendung.) Bisweilen ist Optimierung auch schon durch Auswahl der richtigen Schreibweise möglich.

- FAST2

Die abschließende vierte Variante *FAST2* kann wegen Zugriffsrestriktionen nicht vollständig realisiert werden. Mit ihr soll der Einfluss, den das Einsparen von String-Objekten hat, untersucht werden. Dafür wird ein weiterer *PatternConverter* definiert, der diesmal die *format*-Methode von *org.apache.log4j.helpers.PatternConverter* überschreibt. Im Wesentlichen wird in dieser Methode die in Listing 1 zu findende *convert*-Methode aufgerufen und deren Ergebnis-String in einen *StringBuffer* kopiert. Listing 3 zeigt die geänderte Methode. Die erste auskommentierte Zeile wurde durch die nachfolgende neue ersetzt. Die zweite auskommentierte Zeile müsste angepasst werden, wird für den Test aber nicht benötigt. Die aufgerufene statische Methode

CurrentDateProvider.formatTimeMillisToChars kapselt den Code für das Ermitteln der aktuellen Instanz des Providers und ruft deren *formatToCharArrayIntern*-Methode (vgl. Listing 2) auf. Durch das direkte Kopieren des *char*-Arrays in den *StringBuffer* wird pro Aufruf ein String eingespart. (Eigentlich ist diese Optimierung generell anwendbar, praktisch aber leider mit einigem Aufwand verbunden, da die für die Konfiguration des *PatternConverter* verwendete Klasse *FormattingInfo* package-private ist.)

Listing 3

```
@Override
public void format(StringBuffer sbuf, LoggingEvent e) {
//  String s = convert(e); wird ersetzt durch
  char[] s =
CurrentDateProvider.formatTimeMillisToChars(e.getTimeStamp(),
format);
  if (s == null) {
    if (0 < min) {
      spacePad(sbuf, min);
    }
    return;
  }
  int len = s.length;
  if (len > max) {
//  sbuf.append(s.substring(len - max));
  } else if (len < min) {
    if (leftAlign) {
      sbuf.append(s);
      spacePad(sbuf, min - len);
    } else {
      spacePad(sbuf, min - len);
      sbuf.append(s);
    }
  } else {
    sbuf.append(s);
  }
}
```

Einfache Messung

Auf die Problematik der Performance-Messung mit modernen Prozessoren und VM wurde bereits hingewiesen. Alle Messungen werden auf einem PC mit Intel 7-2600 CPU/3.40 GHz, 16 GB Hauptspeicher und HotSpot-64-Bit-Server-VM (1.6.0_31) ausgeführt. Die Tabelle zeigt repräsentative Ergebnisse für jeweils vier aufeinander folgende Formatierungen von *LogEvent*s, gemessen mit *System.nanoTime()*. Bei den ersten drei Aufrufen wird der gleiche Event verwendet, beim vierten ein solcher mit einem um mehr als eine Sekunde abweichenden Zeitstempel.

Durchlauf / Variante	1	2	3	4
NORMAL	417461 ns	29280 ns	21130 ns	28977 ns
FAST	897102 ns	12075 ns	9961 ns	10263 ns
ISO8601	55541 ns	14187 ns	10565 ns	20828 ns
FAST2	20224 ns	9659 ns	9659 ns	9357 ns

Tab.1: Ergebnisse von Formatierungen von LogEvents

Die Ergebnisse sind wenig überraschend. Die Dauer des ersten Aufrufs wird durch das Laden und Initialisieren der Klassen bestimmt. Da die Varianten von oben nach unten und in einem Lauf ausgeführt werden, profitieren die letzten natürlich davon, dass die meisten benötigten Klassen bereits geladen sind. Klar erkennbar ist die längere Laufzeit von *NORMAL*, während die anderen Varianten relativ dicht beieinander liegen.
Weitere Ergebnisse:

- Die Laufzeiten von NORMAL variieren wesentlich stärker als die der anderen Verfahren.
- Wenn *ISO8601* auf den Cache zurückgreifen kann, ist die Laufzeit mit *FAST* vergleichbar (Läufe 2 und 3).
- Der vierte Aufruf von *ISO8601* (Ereignis mit Zeitstempel, der nicht mit Hilfe des Cache formatiert werden kann) reicht erwartungsgemäß in der Dauer an *NORMAL* heran.
- *FAST2* ist fast immer (ein wenig) schneller als *FAST.*

Wie zu erwarten war, zeigen die Messungen eine große Streuung, wobei von einzelnen Ausreißern abgesehen, die aufgeführten Ergebnisse klar erkennbar sind.

Benchmark

Um einen Eindruck vom Einfluss des JIT-Compilings der VM zu bekommen, werden die Formatierungen mit Googles Benchmark-Framework „Caliper“ [3] untersucht. Das verfügbare Release 0.5 ist laut eigener Homepage noch “…*a little rough around the edges, but we have already found it quite useful*.” Implementiert wird ein statistischer Ansatz, der sich an Experimenten mit komplexen biologischen Systemen orientiert.

Ein Benchmark kann einfach als Erweiterung der Klasse *SimpleBenchmark* programmiert werden. Die Struktur gleicht der von JUnit-Tests, nur dass die Methodennamen mit *time* beginnen. Listing 4 zeigt den Code des Format-Benchmarks. Neben der Laufzeit kann der verbrauchte Speicher protokolliert werden. „Caliper“ bietet verschiedene Möglichkeiten, die Ergebnisse darzustellen, u.a. auch einen direkten Upload auf einen Web-Server. Im einfachsten Fall erhält man die folgende Ausgabe auf die Konsole:

```
  version instances B  ns linear runtime
   NORMAL     9,00  504 903 ==============================
     FAST     7,00  408 431 ==============
   ISO860     9,00  504 536 ================
    FAST2     5,00  312 425 ==============
```

Die Resultate korrespondieren gut mit denen der Einzelmessungen und es wird das erhebliche Beschleunigungspotential der JIT-Technologie gezeigt. Bei der Interpretation muss aber beachtet werden, dass die wiederholte Ausführung in Schleifen zu einer besonders aggressiven Optimierung führt. Im tatsächlichen Anwendungsszenario ist die Formatierung nur ein Schritt in einer umfassenderen Schleife, so dass sich durchaus andere Hotspots ergeben können. Wichtig ist die Erkenntnis, dass keine der untersuchten Varianten Code enthält, der die VM bei ihren Optimierungen behindert.

Listing 4

```
private PatternLayout layout;
private LoggingEvent[] events;

@Param Version version;
public enum Version {NORMAL, FAST, ISO860, FAST2}

@Override
protected void setUp() throws Exception {
  Logger logger= LogManager.getLogger("FormatLogger");
  switch (version) {
  case NORMAL:
   layout= new PatternLayout("%d{yyyy-MM-dd HH:mm:ss,SSS} [%t] -
%m%n");
   break;
  case FAST:
   layout= new FastDatePatternLayout("%d [%t] - %m%n");
   break;
  case ISO860:
   layout= new PatternLayout("%d [%t] - %m%n");
   break;
  case FAST2:
   layout= new FastDatePatternLayout2("%d [%t] - %m%n");
```

```
   break;
  }
  long ts= System.currentTimeMillis() - 8 * 3600L;
  events = new LoggingEvent[10000];
  for (int i= 0; i < events.length; i++) {
    events[i]= new LoggingEvent("fgnOf", logger, ts+600*i,
Level.INFO, "message", null);
  }
}

public String timePatternLayoutFormat(int reps) {
  String dummy= null;
  for (int i= 0; i < reps; i++) {
     dummy= layout.format(events[i % 10000]);
  }
  return dummy;
}
```

Fazit

Die zwischen den Varianten *ISO8601* und *FAST* beobachteten Laufzeitunterschiede sind, auch wenn man berücksichtigt, dass daneben noch teilweise *NORMAL* verwendet wurde, zu gering, um für sich allein die beim Gesamtsystem erreichten Verbesserungen zu erklären. Das gelingt nur, wenn man den Speicherverbrauch in die Betrachtung einbezieht. Wie im Benchmark-Versuch dokumentiert, erzeugt ein Aufruf der *FAST*-Formatierung zwei Objekte weniger (sieben statt neun) und verbraucht dadurch fast 20 Prozent weniger Platz (408 statt 504 Byte). *ISO8601* und *NORMAL* unterscheiden sich in dieser Beziehung nicht. Interessant wäre nun natürlich die Frage, ob beim Einsatz der *FAST2*-Variante in der Produktion eine weitere Verbesserung beobachtet werden kann.

Obwohl die Auswirkungen des Speichersparens deutlich spürbar sind, ist das Ausmaß quantitativ kaum exakt erfassbar, denn neben der Verringerung der GC-Aktivität wird unter Umständen durch größere Datenlokalität auch der Effekt des Cachings verbessert. Wegen der zahlreichen Einflussfaktoren lassen sich solche Effekte jedoch nicht isolieren. Offen bleibt gleichfalls, ob sich die sparsamere Verwendung des Speichers gerade (oder nur) bei Systemen so positiv bemerkbar macht, die vorher bereits eine gewisse Lastgrenze überschritten hatten.

Letztlich haben die zusätzlichen Untersuchungen keine wirklich neuen Fakten an den Tag gebracht. Das sollte niemanden davon abhalten, solche Messungen vorzunehmen. Das Performancemodell von Prozessoren und VM macht Vorhersagen zunehmend schwieriger. Umso wichtiger ist es, die bewährten Heuristiken immer wieder zu überprüfen, um unangenehme Überraschungen möglichst vermeiden zu können. Im vorliegenden Fall darf man nicht außer Acht lassen, dass Log4j eine Bibliothek mit bereits weitgehend optimiertem Code ist. Jedes andere als das vorgestellte Resultat wäre daher eine echte Überraschung gewesen.

Zusammenfassung

Performance-Verbesserungen sind eine wichtige und herausfordernde Aufgabe. In ordentlich gebauten Anwendungssystemen ist es oft schwierig, allein durch Laufzeitprofiling kritische Codestellen zu identifizieren. In diesen Fällen ist die Suche nach Allokations-Hotspots eine vielversprechende Alternative. Die vorliegenden Erfahrungen sprechen dafür, dass insbesondere beim Umgang mit Strings Sorgfalt geboten ist.

Quellen

[1] Apache log4j™, http://logging.apache.org/log4j
[2] http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html
[3] Caliper – Microbenchmarking framework for Java,
https://code.google.com/p/caliper/

Der Autor

Dr. Jürgen Lampe ist IT-Berater bei der Agon Solutions in Frankfurt. Vor seiner Tätigkeit als Berater wirkte er als Hochschullehrer an einer Technischen Universität. Seit mehr als 15 Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen im Banken-umfeld. An Fachsprachen (DSL) und Werkzeugen für deren Implementierung ist er seit seiner Studienzeit interessiert.

Email: juergen.lampe@agon-solutions.de

Agon Solutions

Die Agon Solutions, 2004 gegründet, ist ein unabhängiges IT-Dienstleistungsunternehmen mit Firmensitz in Eschborn bei Frankfurt und weiteren Standorten in Hamburg und Berlin. Das branchenübergreifende Dienstleistungsportfolio von Agon umfasst das „Agon-proven IT-Consulting“, eine bewährte, herstellerneutrale IT-Beratung; sowie die „Agon-tailored IT-Solutions“, zu denen maßgeschneiderte, individuelle Softwareentwicklung und passgenaue Softwareintegration gehören. Agon stellt sich mit professionellem Projektmanagement, proaktivem Anforderungsmanagement und änderbaren Softwarearchitekturen auf die individuellen Bedürfnisse seiner Kunden ein. In ausgewählten Branchen wie Banken, Versicherungen, Touristik, Aviation und Health Care bietet Agon gemeinsam mit seinen Partnern Lösungen, die auf fundiertem Geschäftsprozess-Know-how beruhen und speziell auf die jeweilige Branche zugeschnitten sind: die „Agon-tailored Business Solutions“. Die plattformübergreifende technologische Kompetenz bei Agon reicht von klassischen Mainframe-Architekturen bis hin zu modernen Java/JEE Web- und Portal-Architekturen. Zu den Referenzkunden von Agon gehören unter anderen die AOK Berlin-Brandenburg, die Commerzbank, die Deutsche Bank, die Deutsche Bank Bauspar, die Deutsche Börse, die Finanz Informatik und die Deutsche Lufthansa.

Frankfurter Strasse 71-75
D-65760 Eschborn
Telefon : +49 6196 80269 0
Telefax : +49 6196 80269 11
http://www.agon-solutions.de

Handelsregister Frankfurt HRB 58185
St.-Nr. 4022826171
Geschäftsführer: Udo Peters