VisualVM. Мониторинг, профилировка и диагностика Java-приложений

Тихо и незаметно закончился на прошлой неделе крупнейшая конференция Java-разработчиков JavaOne 2009, и по сети начали распространяться некоторые слайды. Я, волею судеб, оказался специалистом по производительности Java SE, и бОльшую часть времени работал в павильоне на стенде «Java SE Platform: Performance, Java HotSpot VM Internals, and Diagnostics». Подавляющее большинство вопросов, заданных мне, сводилось к нескольким:

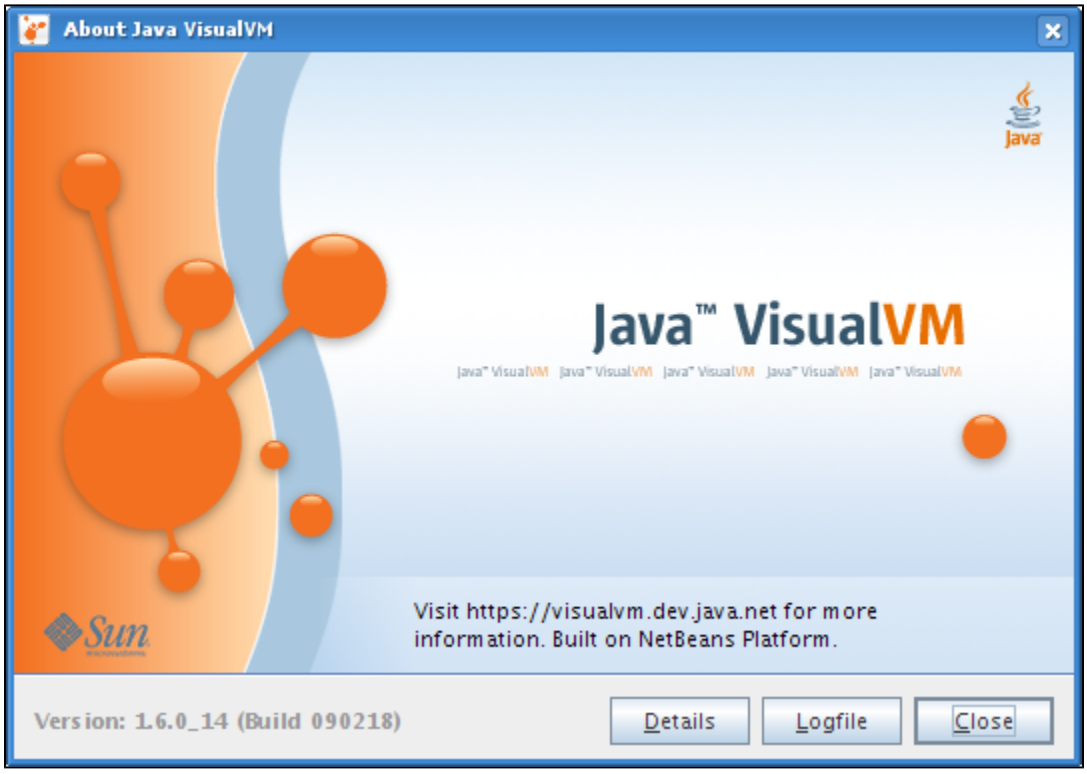
1. Как узнать «горячие места» в коде?
2. Как узнать, сколько занимает и что делает GC?
3. Как узнать, что происходит с памятью и где она «течёт»?

Добрых три-четыре десятка раз я отвечал одно и то же, и каждый раз удивлялся, что мои собеседники не владеют, как мне казалось, элементарным набором инструментов. Поэтому хочу зафиксировать это знание здесь, чтобы впредь не повторяться :)

Итак, в стародавние времена в составе JDK было несколько хороших инструментов для диагностики JVM: jps, jinfo, jstack, jstat, jmap, jhat, и т.п. Все они консольные, и тем или иным образом были удобны. До настоящего времени единственным включённым в состав JDK мониторинговым GUI-инструментом был jconsole. У jconsole, однако, есть один очень большой минус: она не предоставляет никаких возможностей профилировки.

Однако в NetBeans есть отличный профайлер! Кроме того, платформа NetBeans достаточно модульная, чтобы можно было выделить профайлер в отдельный инструмент и распространять его в составе JDK. А если ещё прикрутить туда сбор всей информации, доступной в jconsole, то получится универсальный инструмент для мониторинга, профилировки и диагностики Java-приложений.

Инженеры Sun это сделали, и так родился

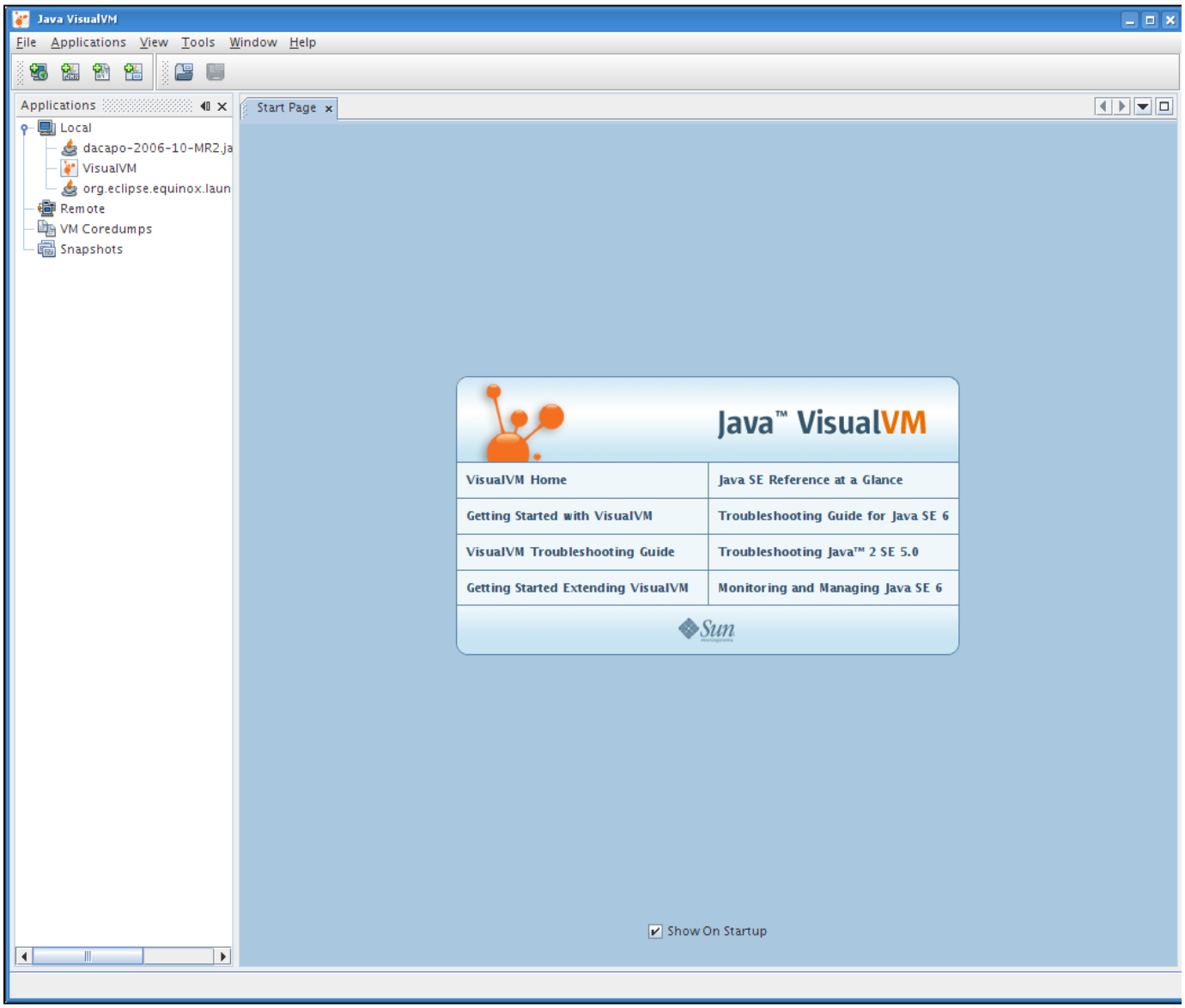


Подразумевается, что интерфейс VisualVM интуитивно понятен, поэтому подробно разбирать его я не буду. Постараюсь вместо этого рассказать о философии, стоящей за этим инструментом. В качестве тестируемого приложения я выбрал свободно распространяемый набор бенчмарок DaCapo, конкретно тест lusearch, который внутри гоняет тест производительности на Apache Lucene. Цель данного топика — показать возможности VisualVM, а не поведение конкретной виртуальной машины на конкретном оборудовании в конкретных условиях.

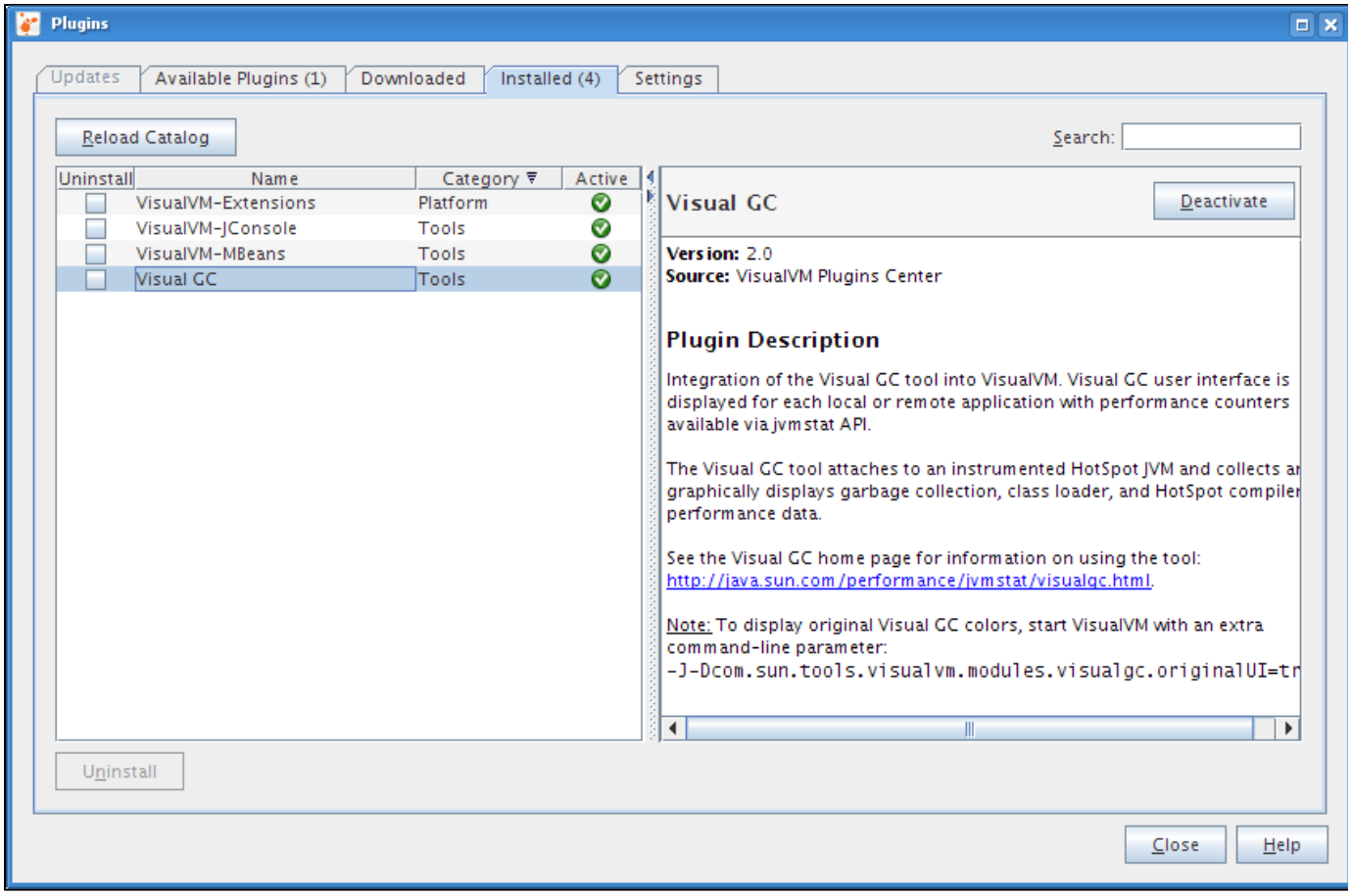
Перед тем, как начать саму демонстрацию, я скачал и установил несколько дополнений к VisualVM, которые покажу ниже. Для этого я запустил VisualVM:

$JAVA_HOME/bin/jvisualvmЕсли у вас стоит Sun JDK 6u7 и выше, то VisualVM у вас уже есть, и она доступна как и мне, из каталога с JRE. Если у вас более старая машина, то лучше проапгрейдиться :) Но можно скачать VisualVM отдельно.

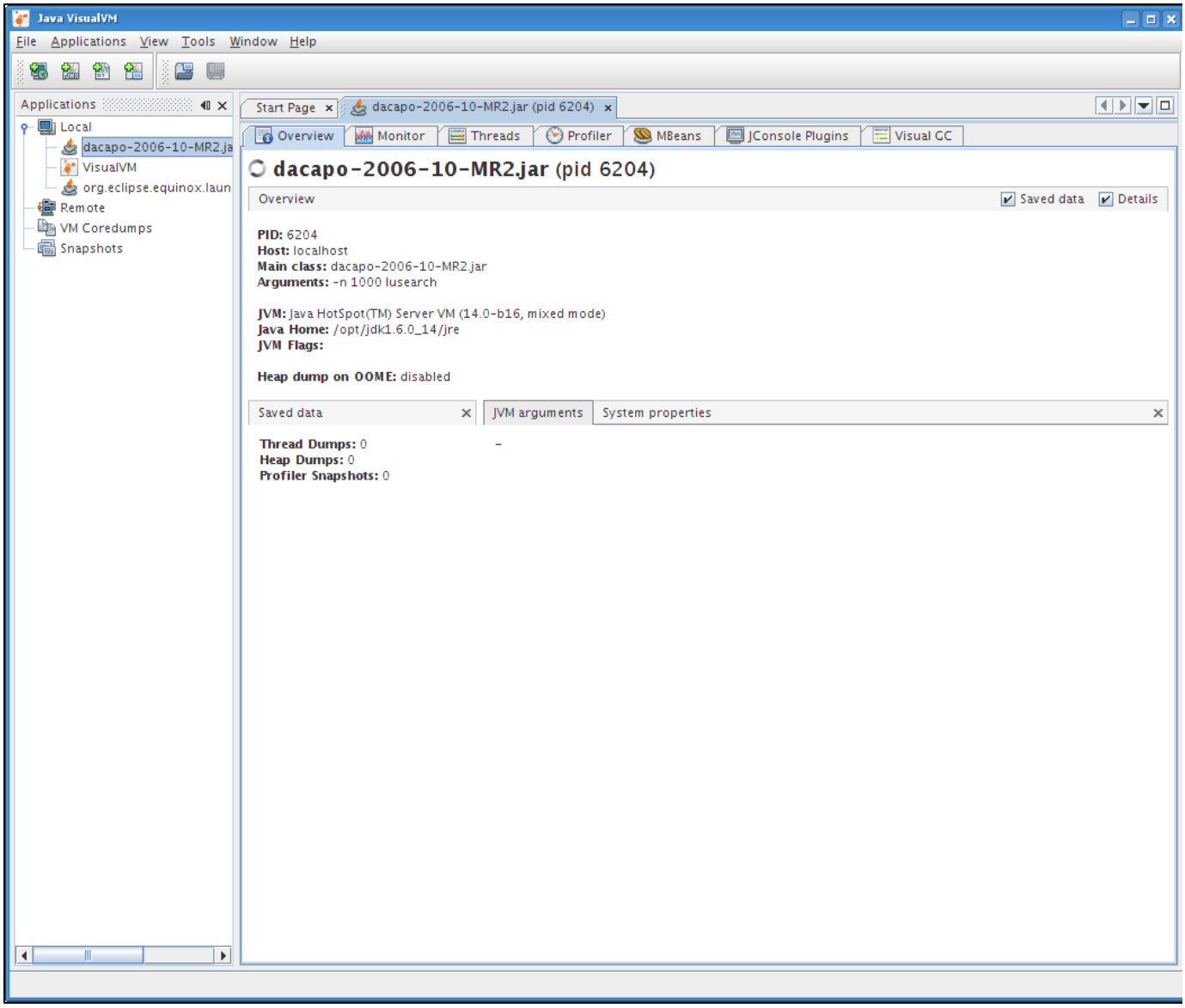
Вот так выглядит этот притаившийся зверь



В меню «Tools -> Plug-ins» установил VisualGC

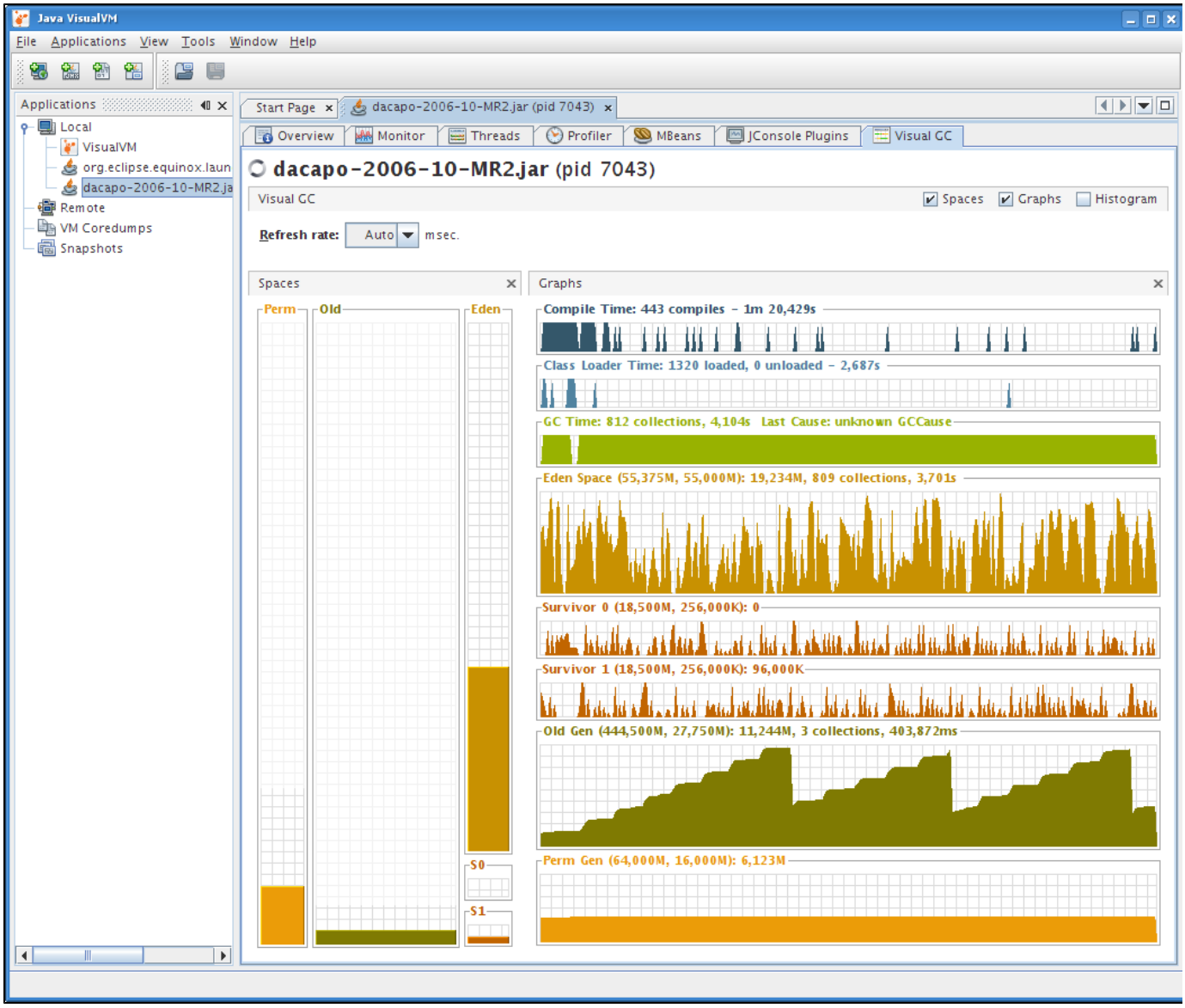


Всё готово! Время опробовать его в действии. Я запустил фоном DaCapo:lusearch `/opt/jdk1.6.0_14/bin/java -server -jar dacapo-2006-10-MR2.jar -n 100 lusearch`и отправился в VisualVM. Нашёл в дереве слева интересующий меня java-процесс (можно увидеть, что там торчат ещё сам VisualVM и мой Eclipse), сделал на нём Open и попал в Overview моего процесса:



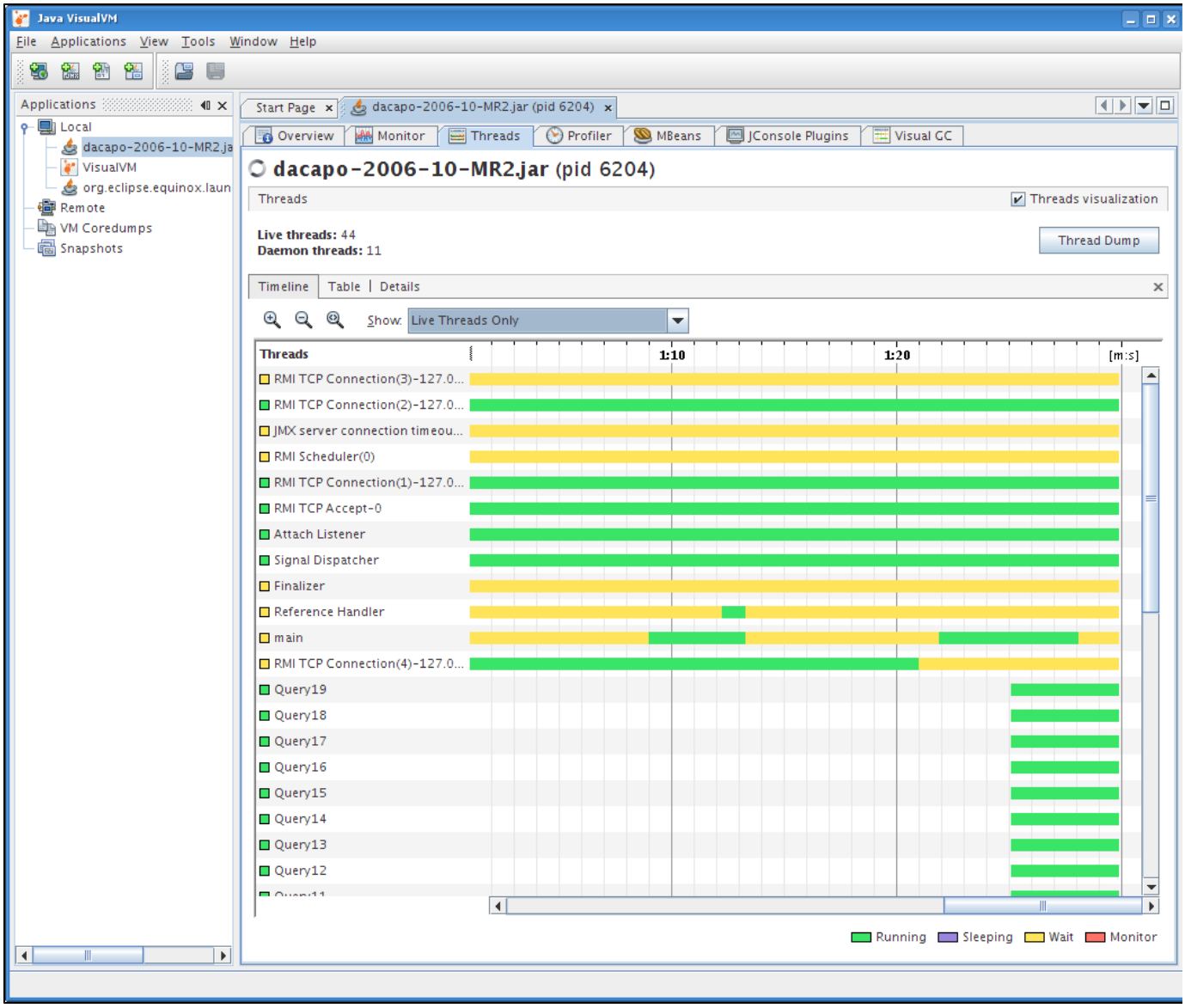
Здесь не так интересно, поэтому я поспешил перейти на вкладку Monitor и посмотреть на высокоуровневую статистику:
На скриншоте легко можно увидеть графики утилизации CPU (включая активность GC), количество загруженных классов и созданных нитей, состояние памяти.

Можно получить информацию о памяти чуть детальнее, здесь нам и пригодится VisualGC:

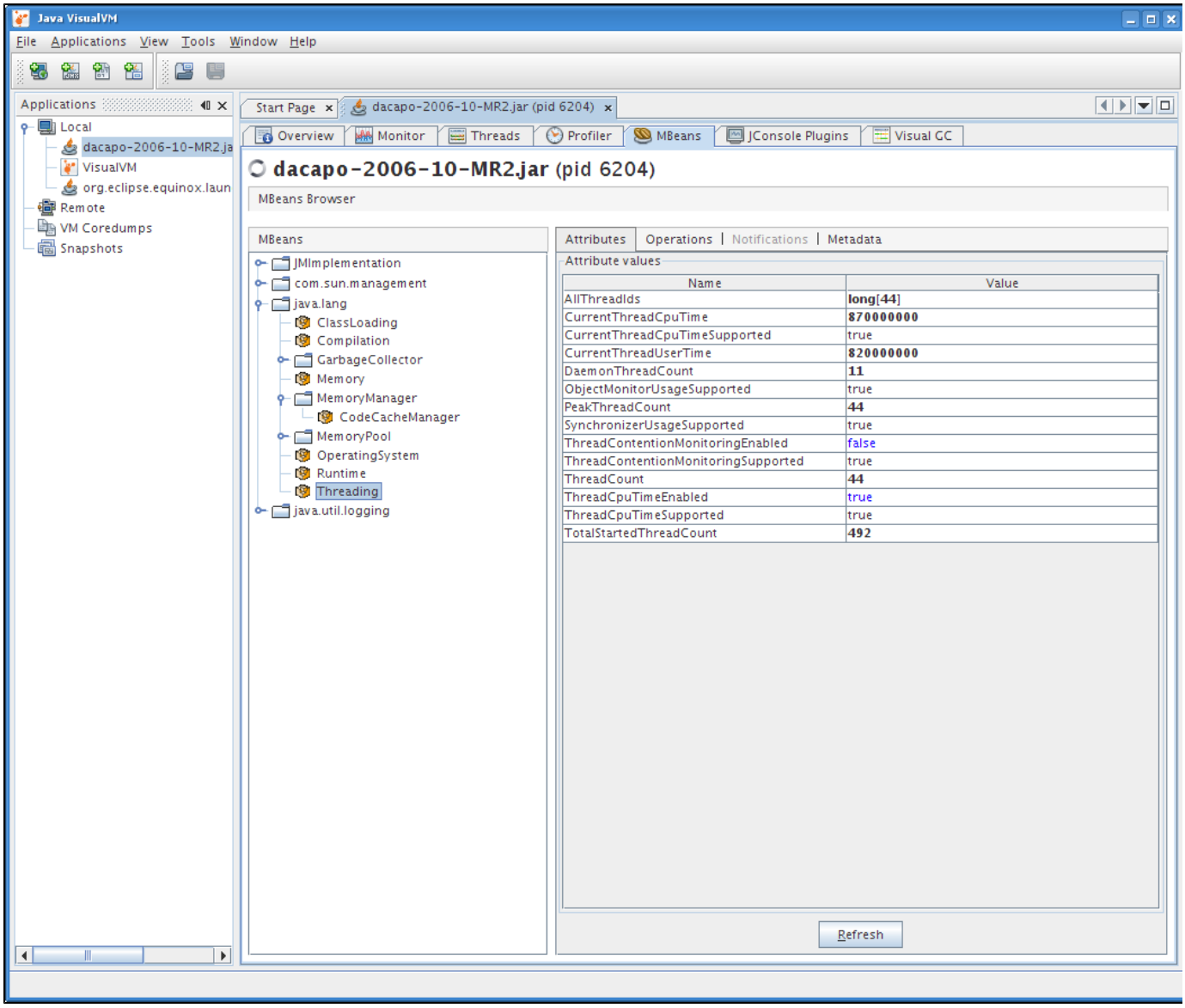


То, что здесь происходит, будет понятно тем, кто знает, как устроен сборщик мусора. Конкретно на этом скриншоте можно увидеть три пространства для объектов: постоянное, старое и молодое. Последнее разбито на ещё на три подпространства: эдем и два пространства под выжившие объекты. Данные обновляются вживую, а графики справа показывают, что происходило за последнее время. По этому скриншоту можно оценить, правильно ли GC работает на конкретном приложении. Здесь я вижу, что часть объектов перемещается в old space, а потом всё-таки убирается полным GC: может, стоит покрутить promotion thresholds? Или попробовать другой GC?

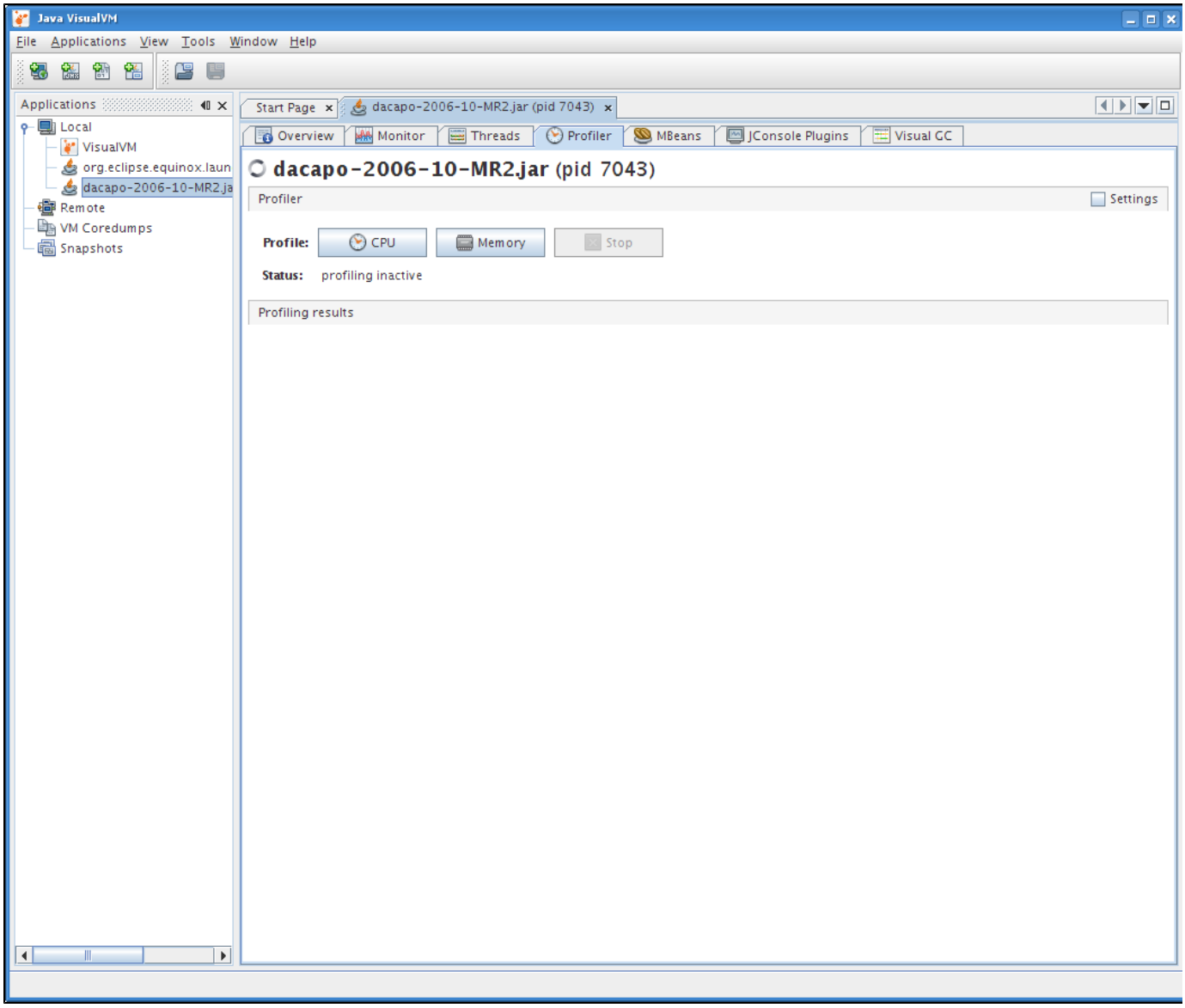
Вернёмся пока к закладкам. На закладке Threads можно увидеть подробную статистику по нитям. Там есть несколько режимов просмотра, оставляю их любопытному читателю.



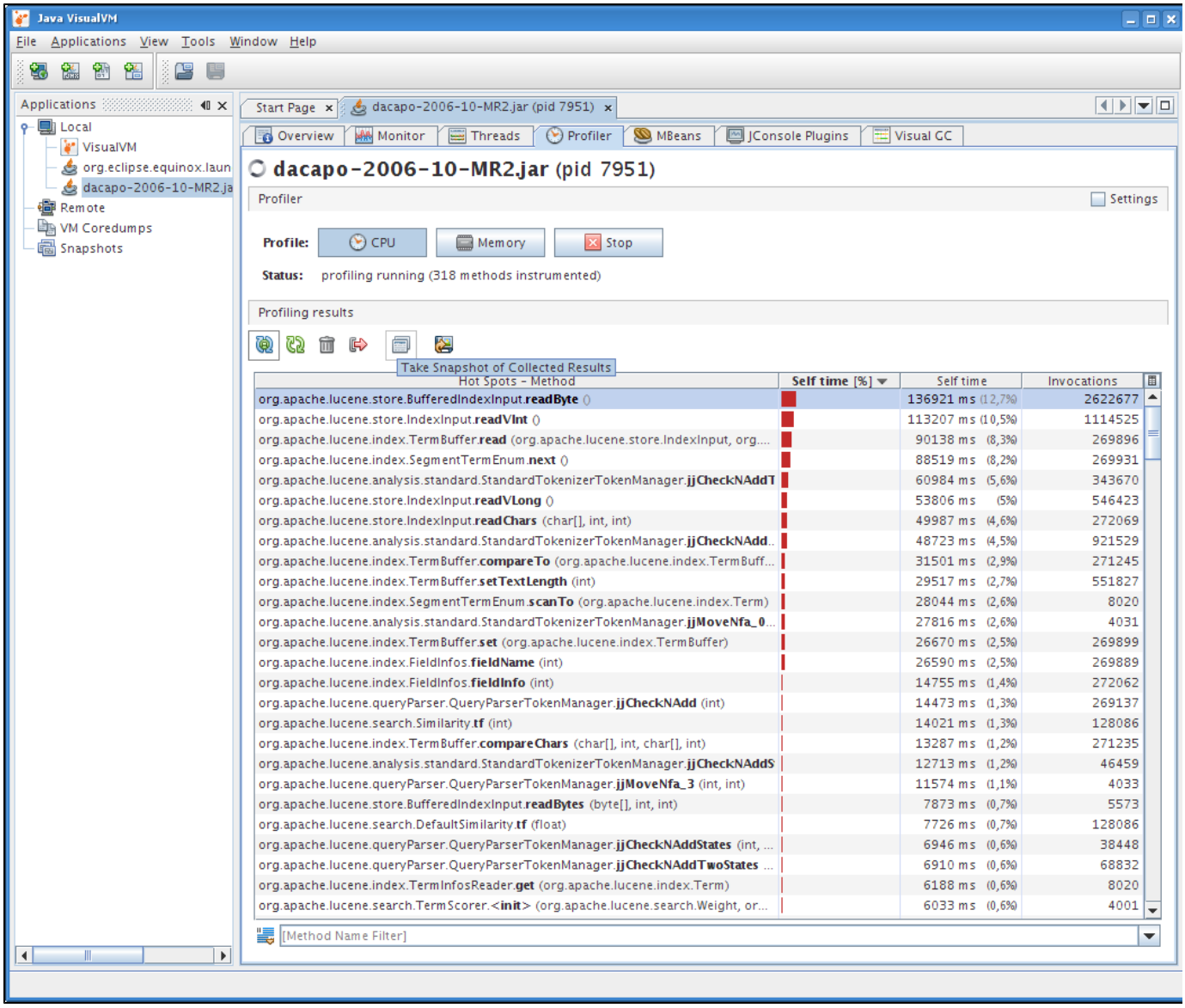
Как и в старой доброй jconsole, можно посмотреть на MBeans:



И наконец! Самое вкусное в этом посте: профайлер.

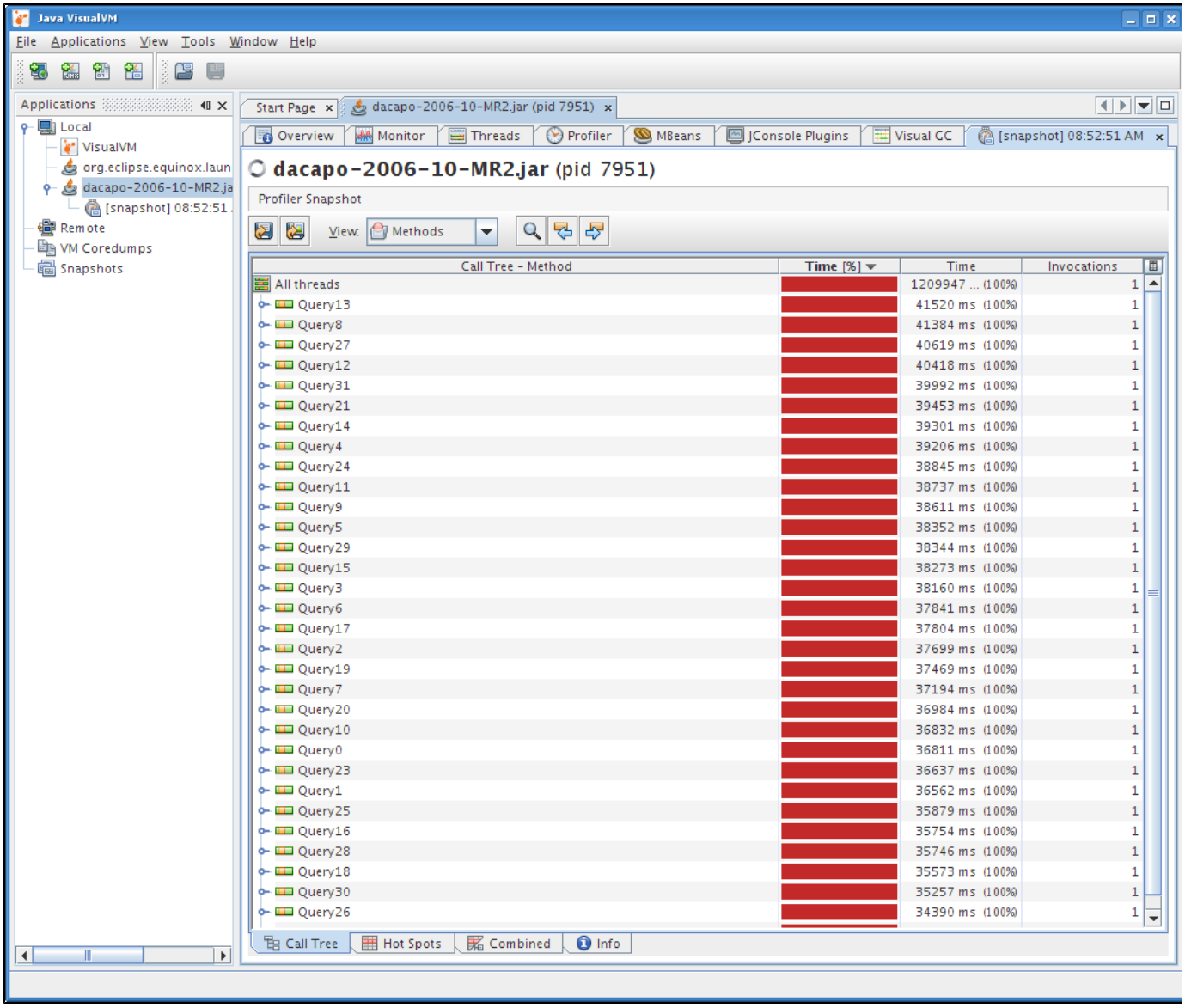


У профайлера есть два режима: CPU и Memory. Первый профилирует методы с точки зрения проведённого в них времени, второй — с точки зрения созданных и уничтоженных объектов. Попробуем для начала CPU. Для этого прямо здесь и сейчас нажимаем кнопку «CPU»:



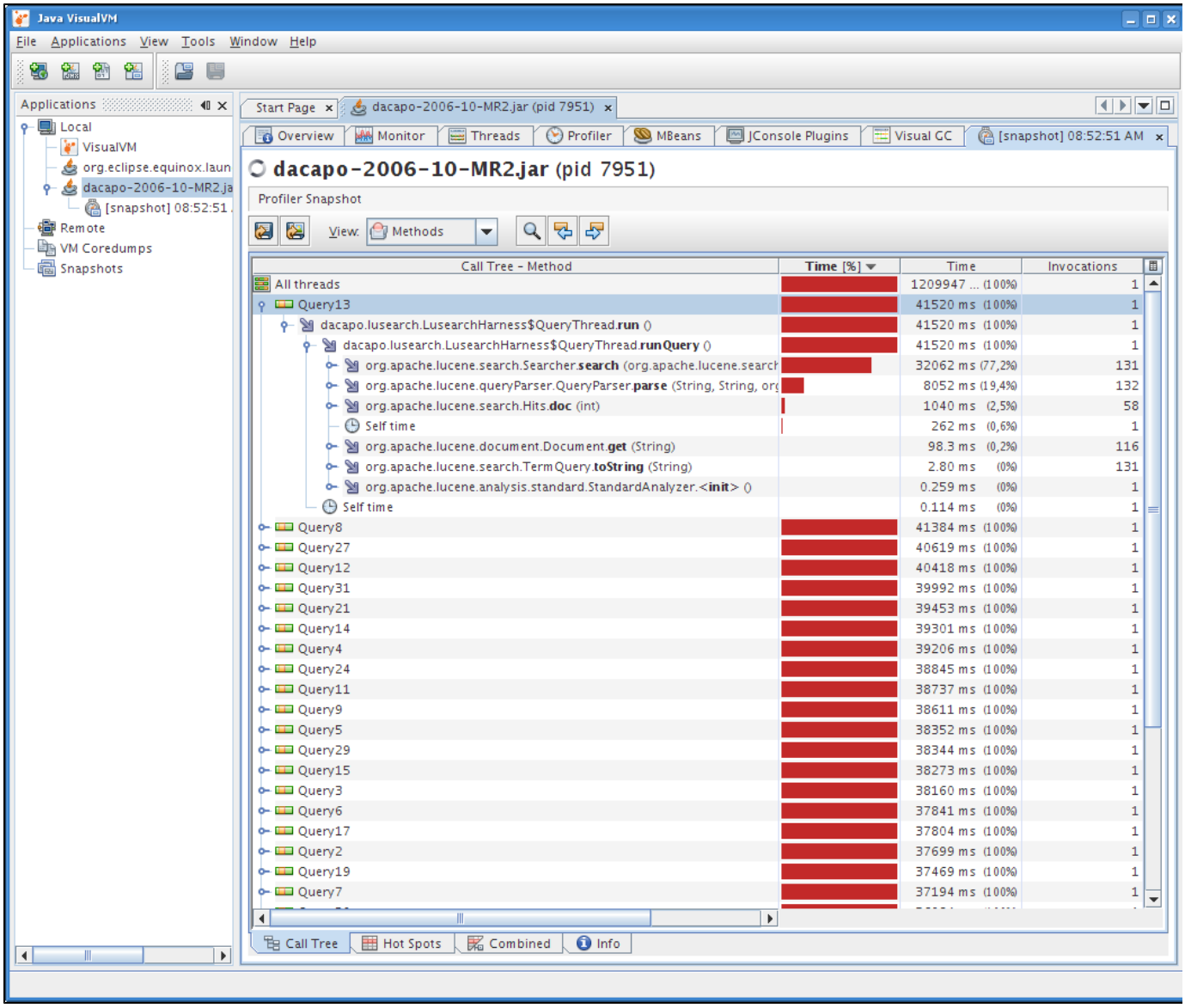
Интересная и полезная особенность состоит в том, что приложение всё ещё работает, профайлер инструментировал приложение на лету. Вот и первые результаты, видно, какой метод занимает больше всего времени. Но это только часть всего дела: неясно ведь, откуда его зовут.

Поэтому мы делаем snapshot при помощи соответствующей кнопки, и открываем его:



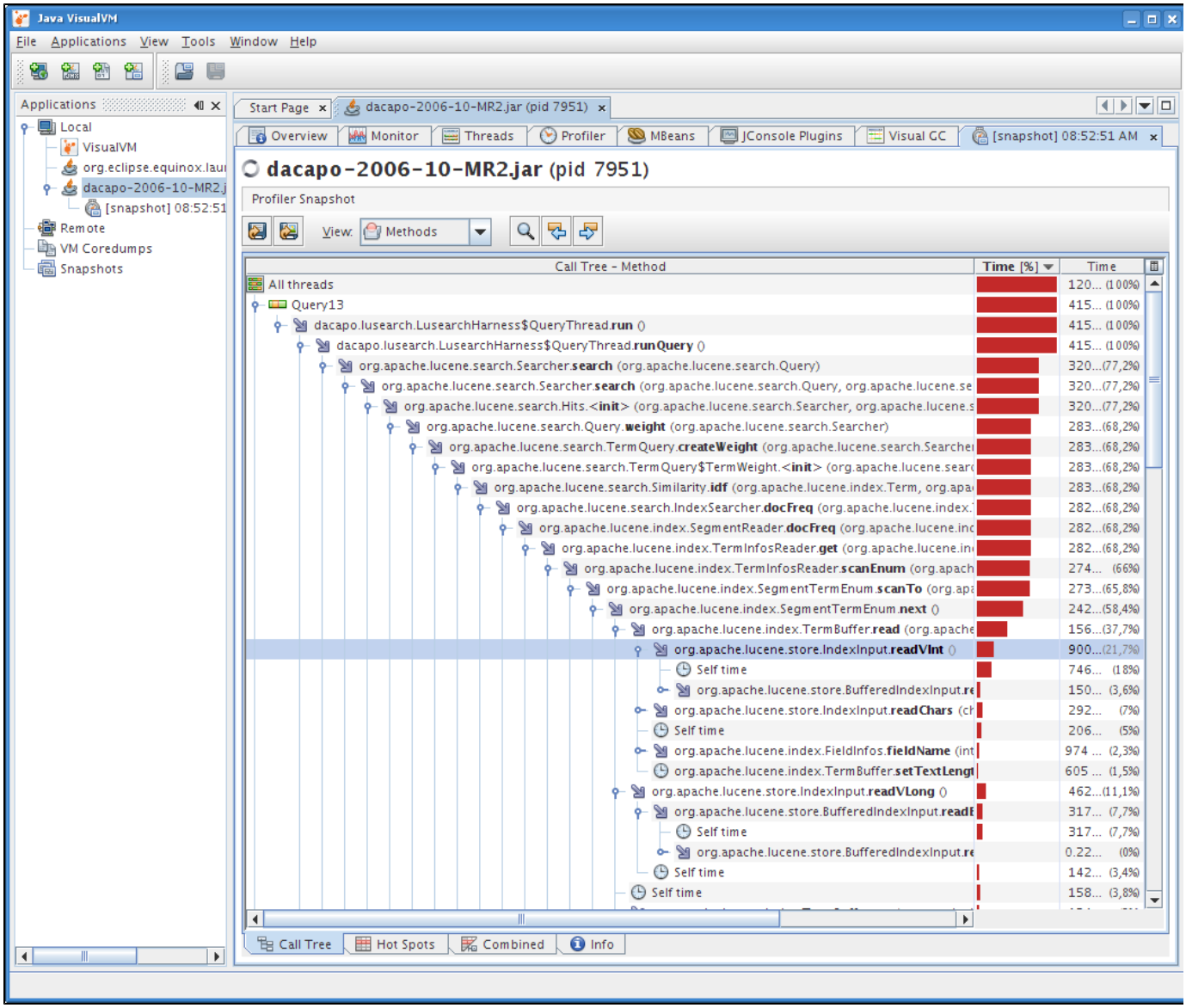
Это — все нитки, которые были пойманы профайлером.

Развернём одну из них:



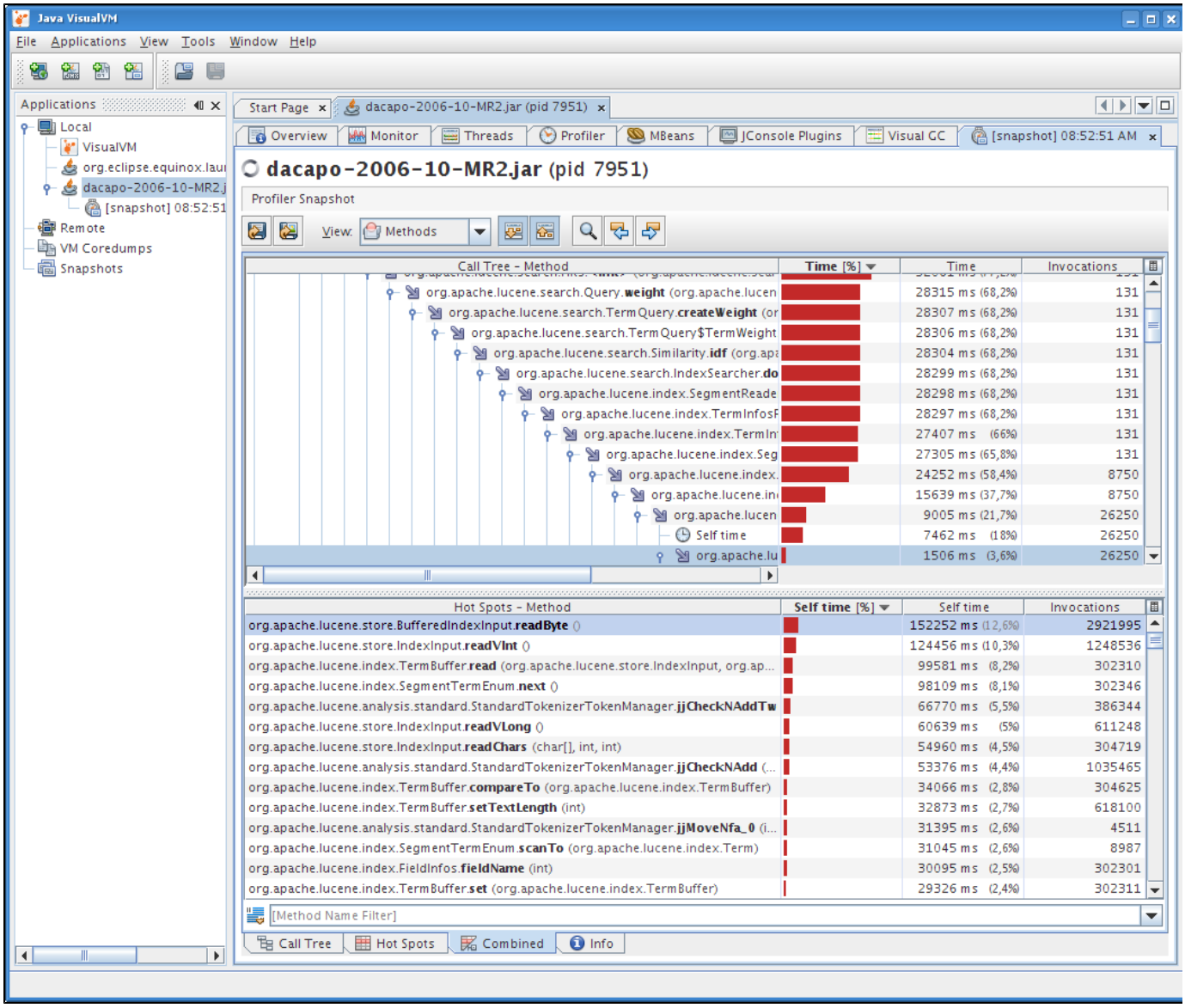
Видно, что исполнение началось с вызова метода QueryThread.run() и дальше оно разбилось на две стадии: QueryParser.parse(...) и Searcher.search(...). Кажется, это правильно.

Пойдём глубже:



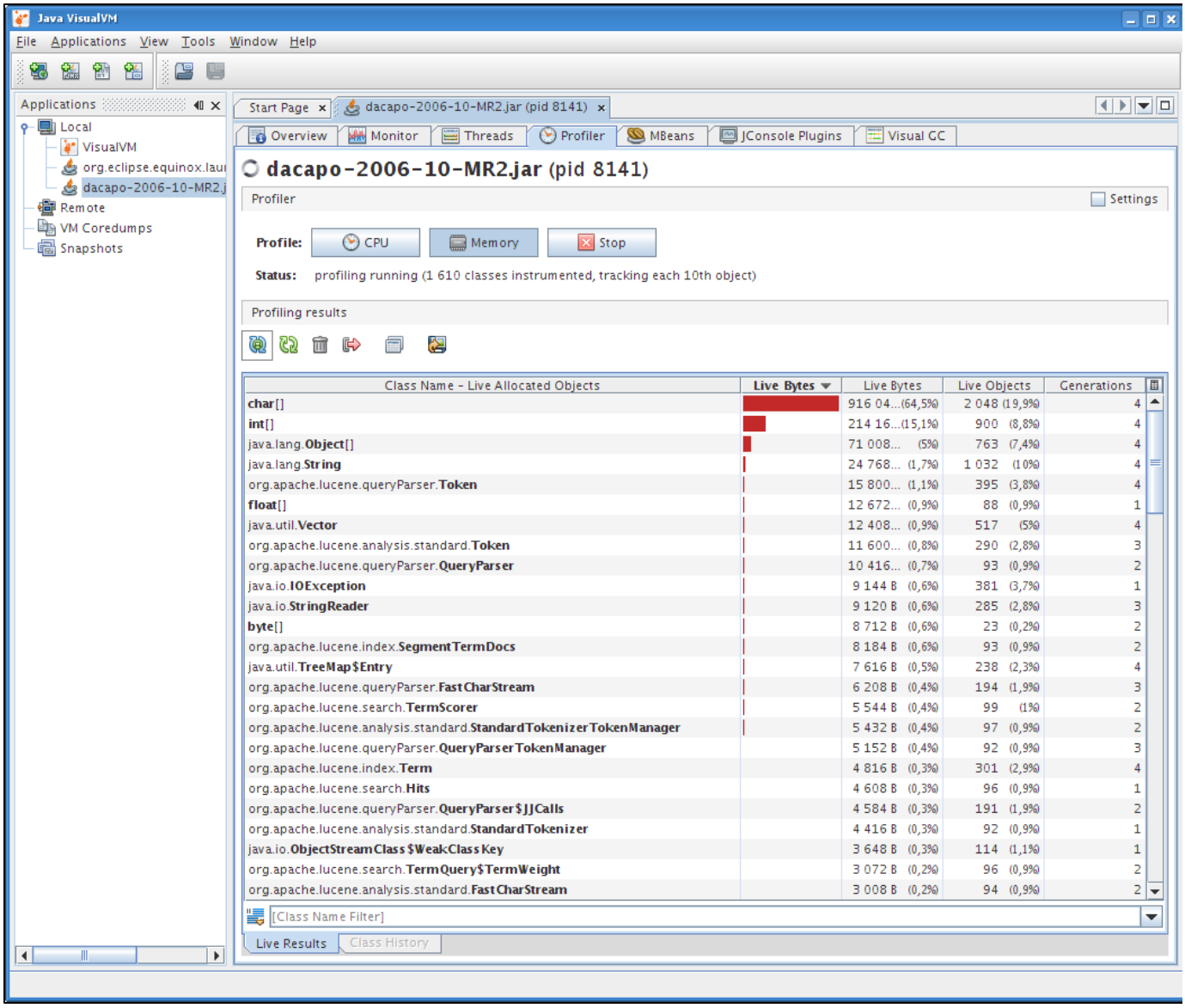
Видно всю глубокую цепочку вызовов, а так же конечных бойцов, которые светятся своим Self time'ом: IndexInput.readV*(). Если бы это было моё приложение, я бы начал задумываться, а нельзя ли на этих методах сэкономить, или как-нибудь их оптимизировать :)

А где же наши горячие места в этом стеке? Перейдём на combined view и выберем первый горячий метод:

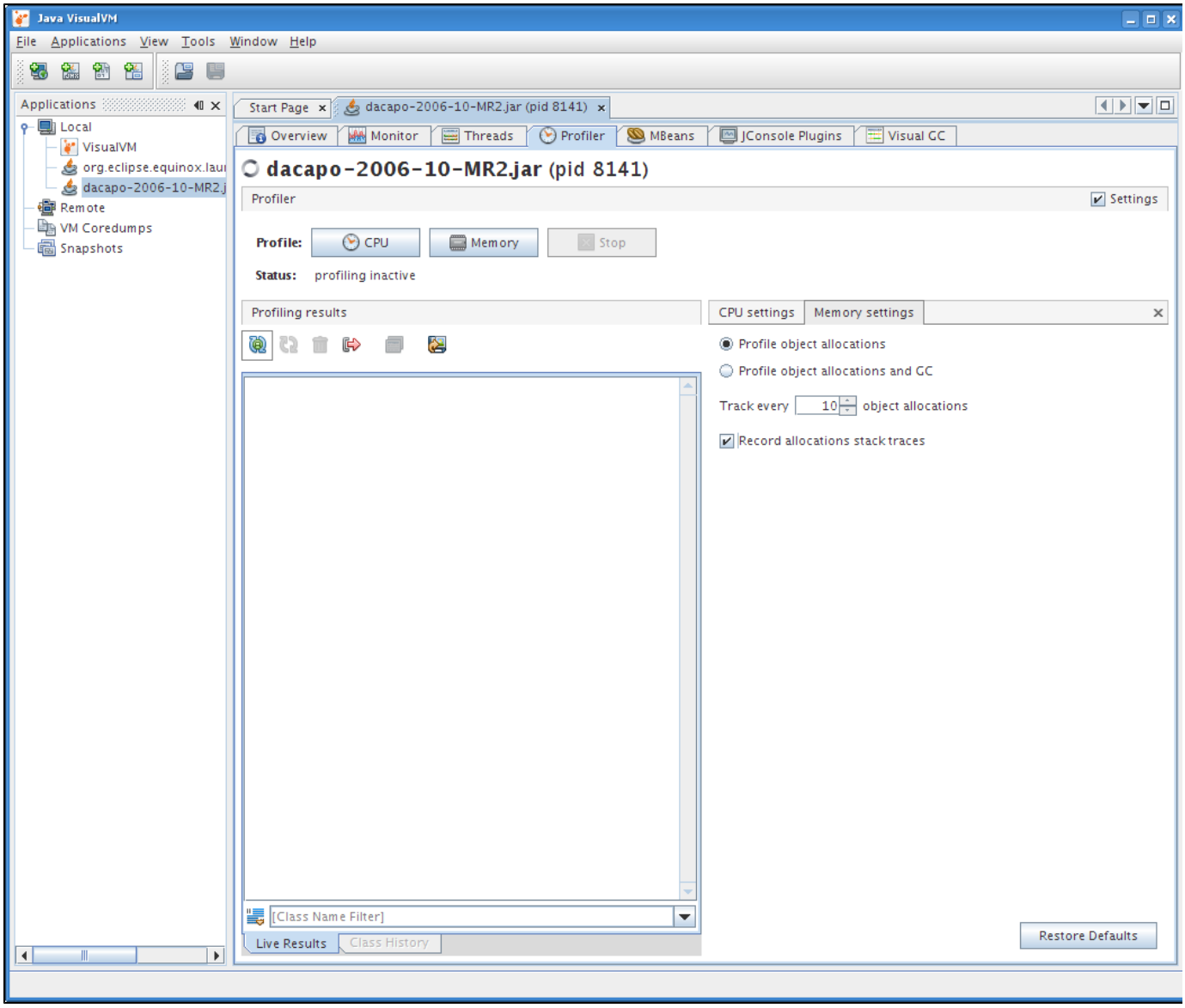


Вон он где.

На этом с CPU профайлером закончим: мы уже узнали, где приложение проводит бОльшую часть своего времени. Попробуем Memory профайлер. Для этого на вкладке Profiler я нажимаю кнопку «Stop», жду когда CPU профайлер отменит инструментацию, а потом жму «Memory». Через некоторое время мы видим это:

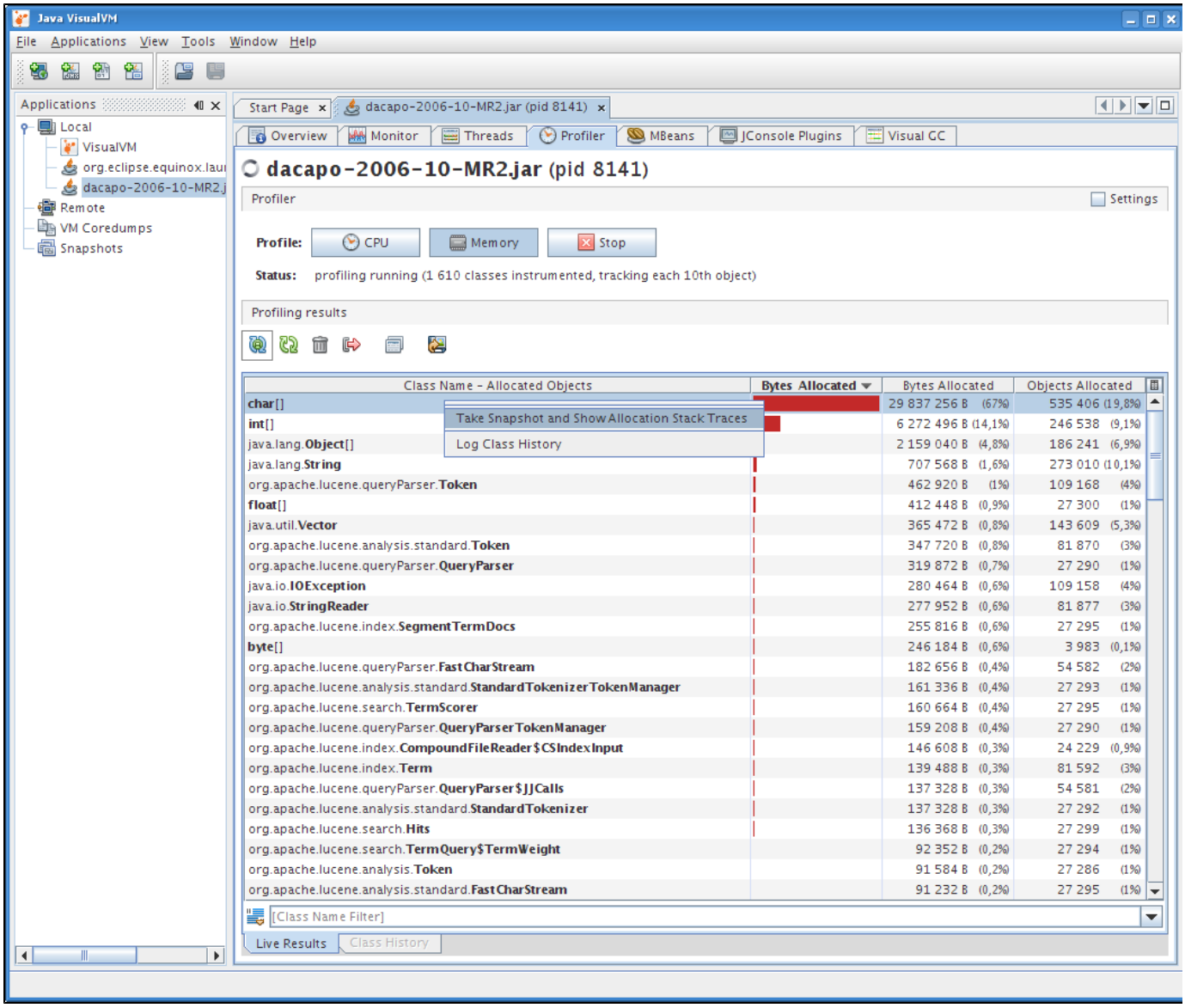


Ага, кажется, у нас преобладают массивы char[]. Неудивительно для Lucene. Однако, я хочу бОльшего: сейчас я смотрю на статистику живых объектов, а хочу узнать, какие объекты вообще создавались за время всего профилирования. Для этого я иду в Settings и прошу учитывать только аллокации объектов:

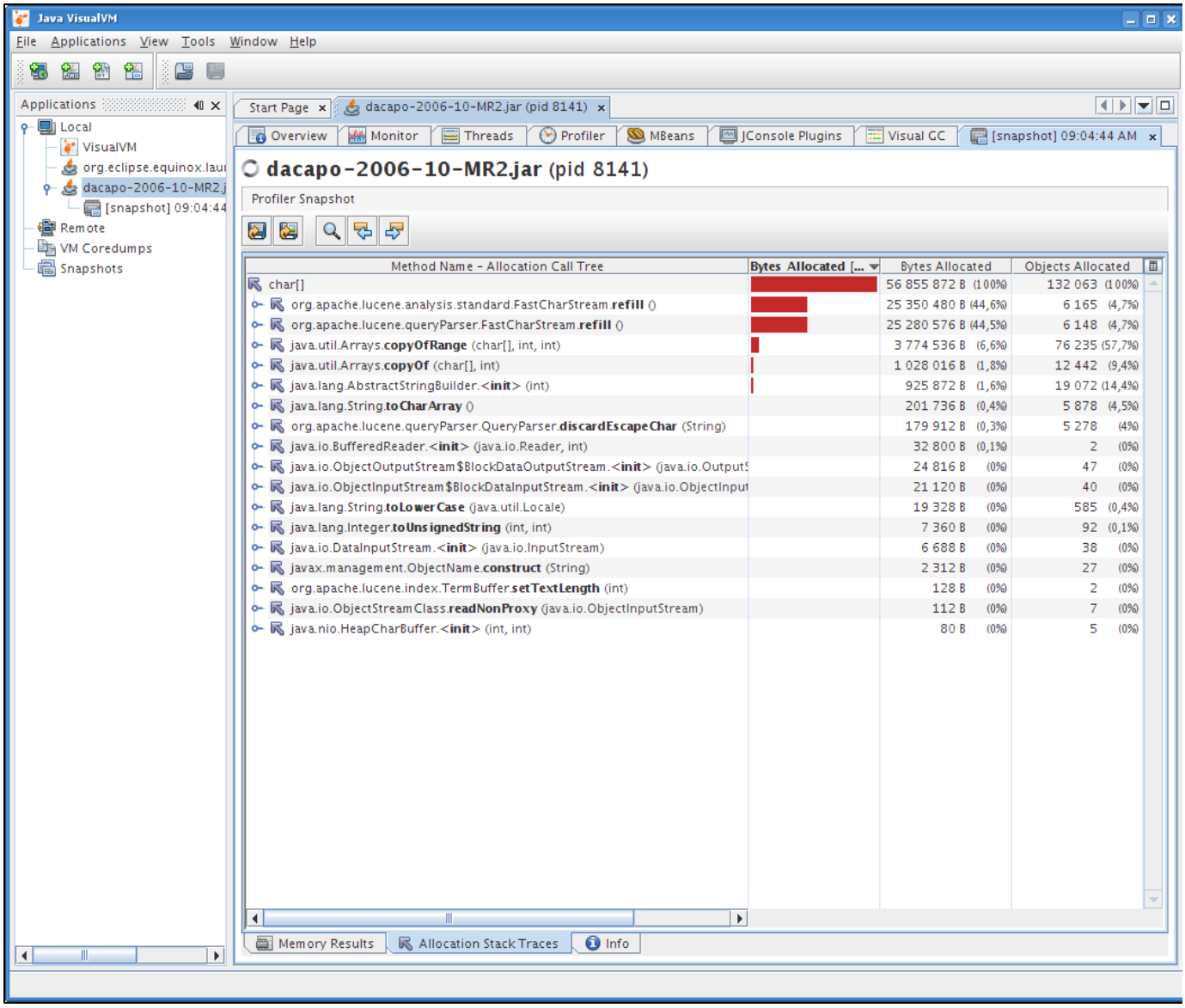


(вместе с этим я включил запись стеков для аллокаций)

Вот что у нас в длинном профиле доминирует:

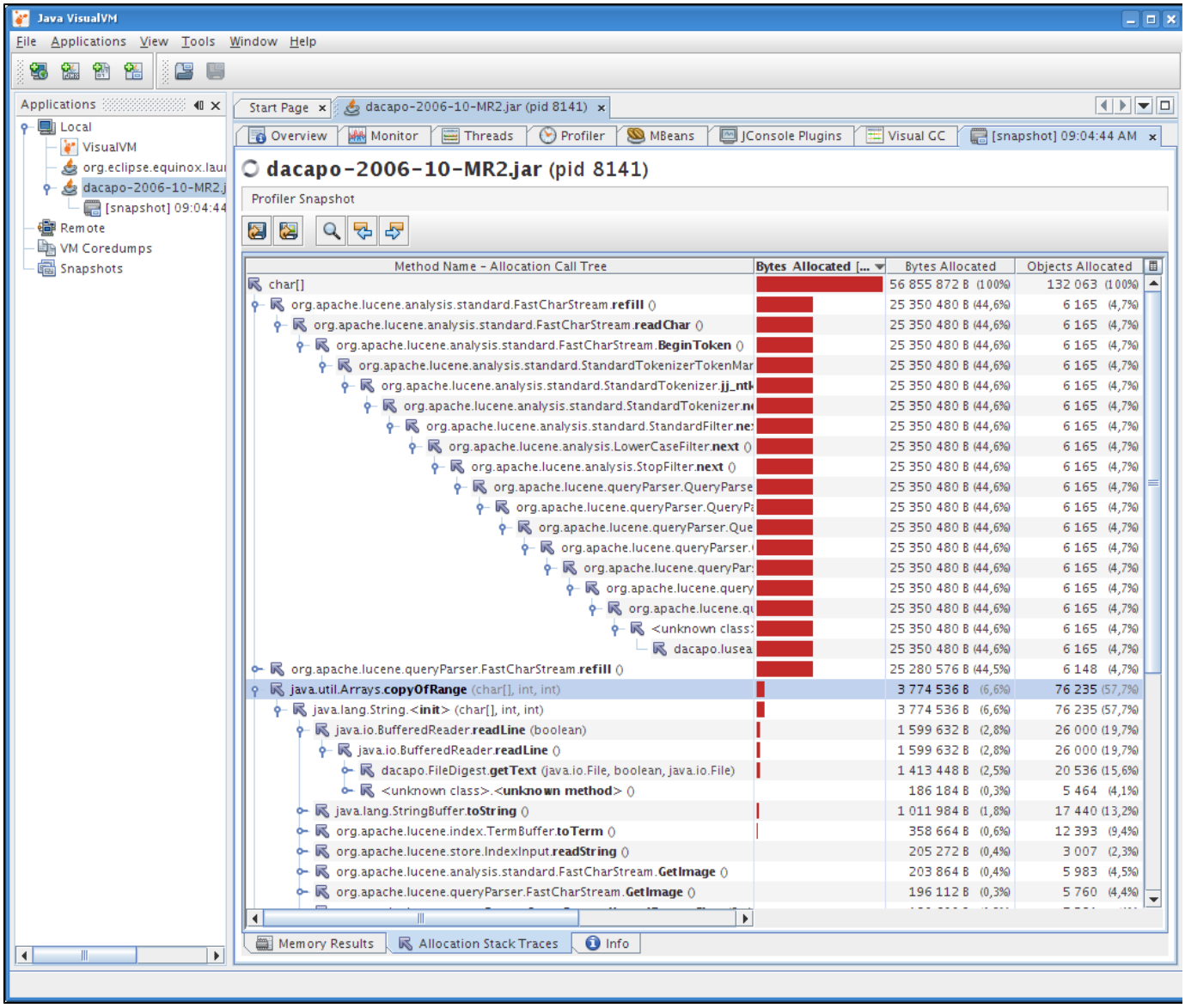


Те же самые парни. Ну что ж, теперь я хочу увидеть, откуда они взялись. Выбираю интересующий меня класс, и прошу показать мне stack trace.



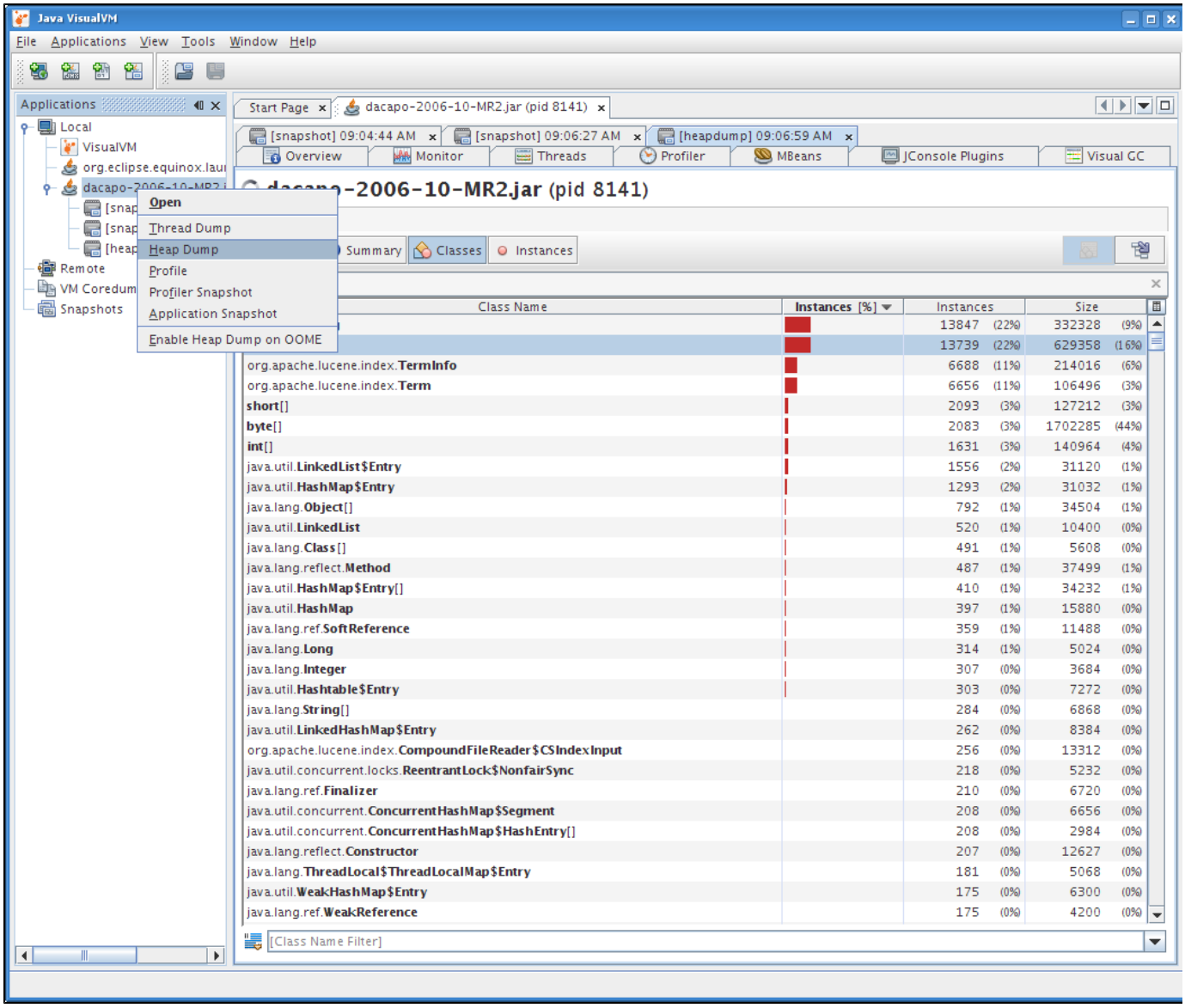
Вот и он! Видно, что подавляющая часть аллокаций происходит внутри Lucene, в методе FastCharStream.refill(). Интересно, в Lucene дубликатный класс в разных пакетах?

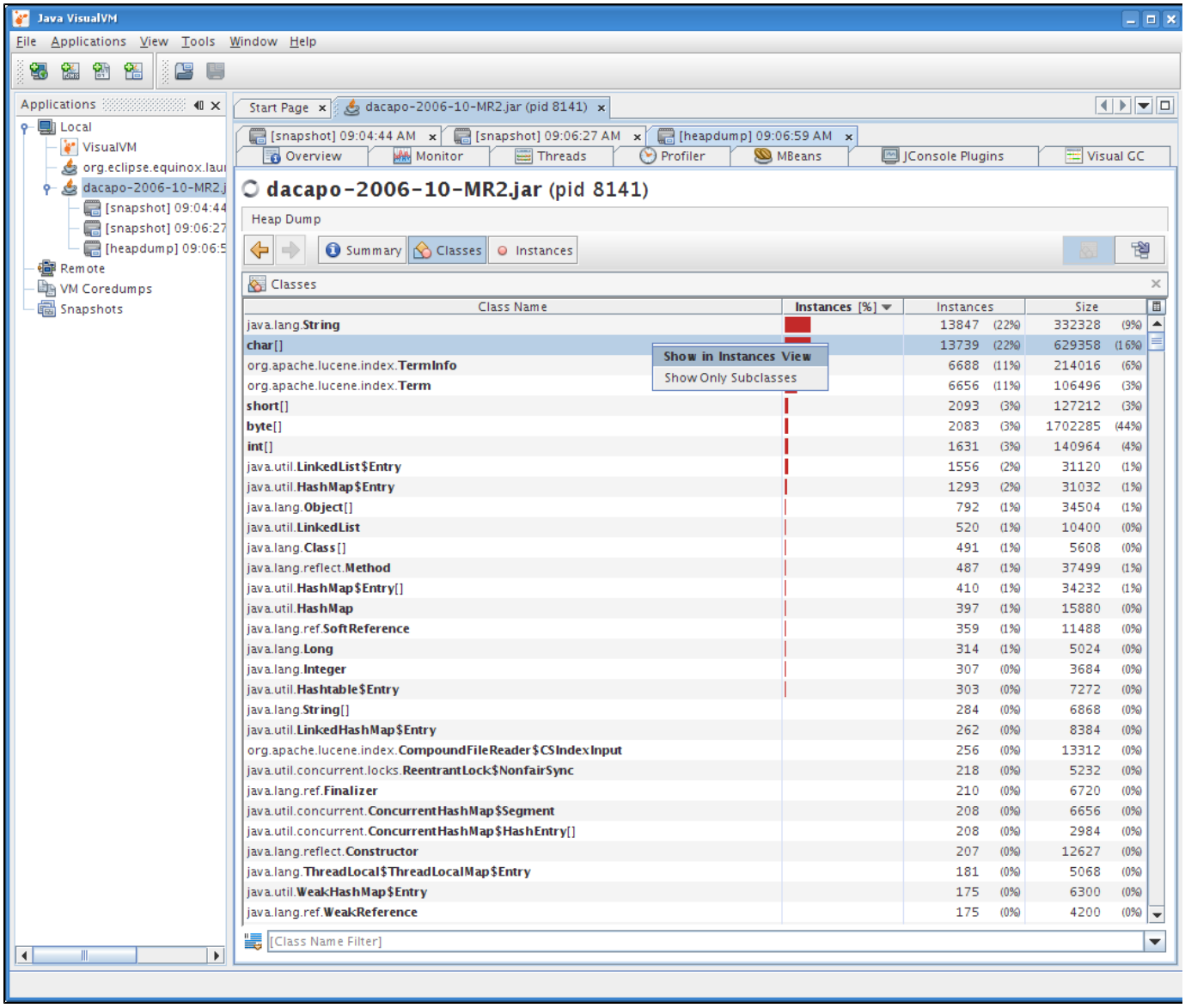
Откуда эти методы зовутся? Отсюда:

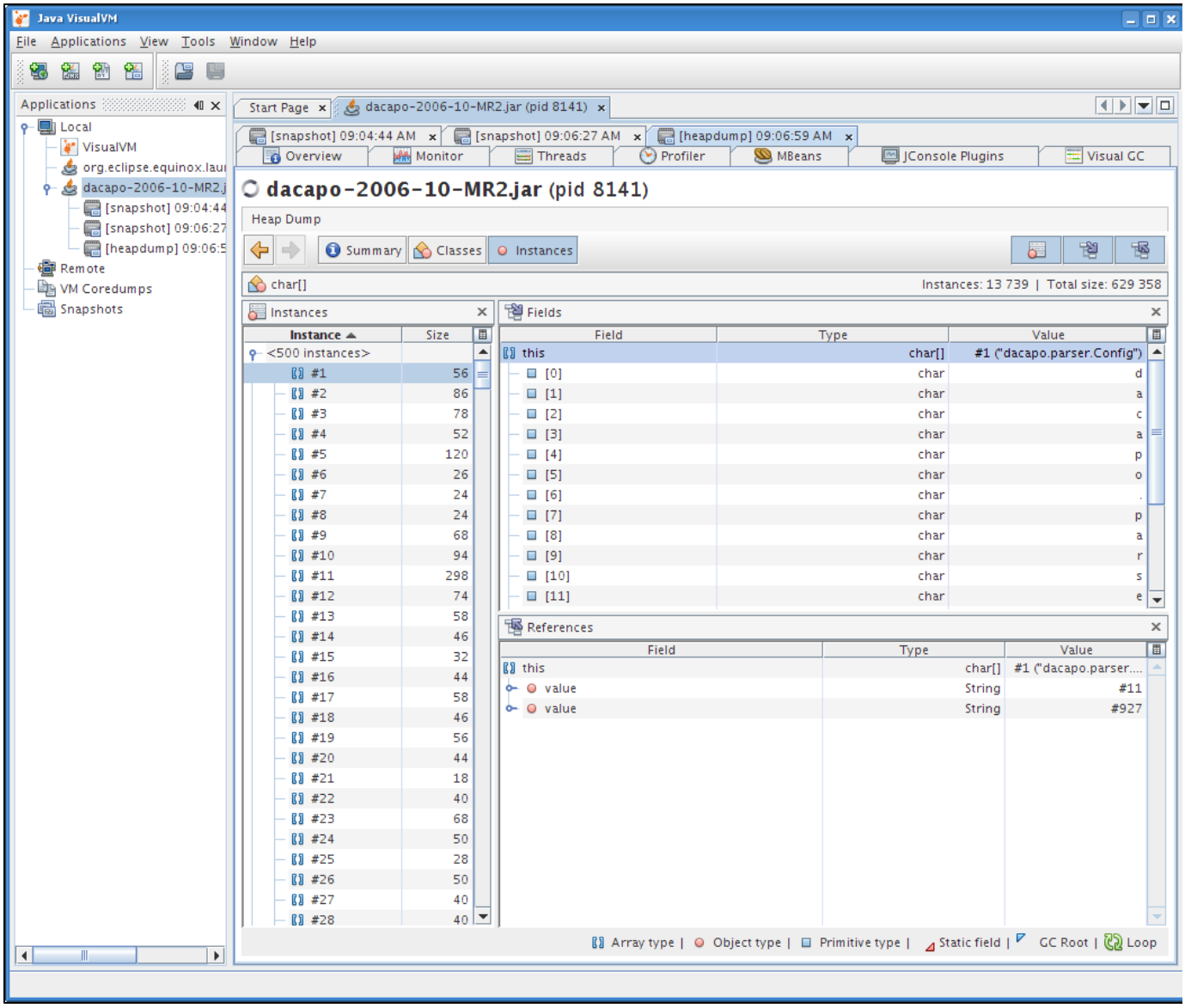


Несмотря на то, что Memory профайлер не делает статического анализа на предмет того, правильно ли вы обращаетесь с памятью, он даёт вам ценную информацию, а что же у вас в памяти творится. Сняв такой профиль в экстремальном режиме работы приложения, когда вот-вот будет OOME (кстати, VisualVM умеет это событие перехватывать), можно одним махом увидеть, какие и откуда объекты пришли.

Потом — классический heap dump.


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
org.eclipse.equinox.laur
dacapo-2006-10-MR2.j
[snap
[snap
[heap
Remote
VM Coredum
Snapshots
Open
Thread Dump
Heap Dump
Profile
Profiler Snapshot
Application Snapshot
Enable Heap Dump on OOME
Start Page
dacapo-2006-10-MR2.jar (pid 8141)
[snapshot] 09:04:44 AM
[snapshot] 09:06:27 AM
[heapdump] 09:06:59 AM
Overview
Monitor
Threads
Profiler
MBeans
JConsole Plugins
Visual GC
dacapo-2006-10-MR2.jar (pid 8141)
Summary
Classes
Instances
Class Name
Instances [%]
Instances
Size
13847 (22%) 332328 (9%)
13739 (22%) 629358 (16%)
org.apache.lucene.index.TermInfo 6688 (11%) 214016 (6%)
org.apache.lucene.index.Term 6656 (11%) 106496 (3%)
short[] 2093 (3%) 127212 (3%)
byte[] 2083 (3%) 1702285 (44%)
int[] 1631 (3%) 140964 (4%)
java.util.LinkedList$Entry 1556 (2%) 31120 (1%)
java.util.HashMap$Entry 1293 (2%) 31032 (1%)
java.lang.Object[] 792 (1%) 34504 (1%)
java.util.LinkedList 520 (1%) 10400 (0%)
java.lang.Class[] 491 (1%) 5608 (0%)
java.lang.reflect.Method 487 (1%) 37499 (1%)
java.util.HashMap$Entry[] 410 (1%) 34232 (1%)
java.util.HashMap 397 (1%) 15880 (0%)
java.lang.ref.SoftReference 359 (1%) 11488 (0%)
java.lang.Long 314 (1%) 5024 (0%)
java.lang.Integer 307 (0%) 3684 (0%)
java.util.Hashtable$Entry 303 (0%) 7272 (0%)
java.lang.String[] 284 (0%) 6868 (0%)
java.util.LinkedHashMap$Entry 262 (0%) 8384 (0%)
org.apache.lucene.index.CompoundFileReader$CSIndexInput 256 (0%) 13312 (0%)
java.util.concurrent.locks.ReentrantLock$NonfairSync 218 (0%) 5232 (0%)
java.lang.ref.Finalizer 210 (0%) 6720 (0%)
java.util.concurrent.ConcurrentHashMap$Segment 208 (0%) 6656 (0%)
java.util.concurrent.ConcurrentHashMap$HashEntry[] 208 (0%) 2984 (0%)
java.lang.reflect.Constructor 207 (0%) 12627 (0%)
java.lang.ThreadLocal$ThreadLocalMap$Entry 181 (0%) 5068 (0%)
java.util.WeakHashMap$Entry 175 (0%) 6300 (0%)
java.lang.ref.WeakReference 175 (0%) 4200 (0%)
[Class Name Filter]


Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
org.eclipse.equinox.laur
dacapo-2006-10-MR2.j
[snapshot] 09:04:44
[snapshot] 09:06:27
[heapdump] 09:06:5
Remote
VM Coredumps
Snapshots
Start Page
dacapo-2006-10-MR2.jar (pid 8141)
[snapshot] 09:04:44 AM
[snapshot] 09:06:27 AM
[heapdump] 09:06:59 AM
Overview
Monitor
Threads
Profiler
MBeans
JConsole Plugins
Visual GC
dacapo-2006-10-MR2.jar (pid 8141)
Heap Dump
Summary
Classes
Instances
Classes
Class Name
Instances [%]
Instances
Size
java.lang.String 13847 (22%) 332328 (9%)
char[] 13739 (22%) 629358 (16%)
org.apache.lucene.index.TermInfo 6688 (11%) 214016 (6%)
org.apache.lucene.index.Term 6656 (11%) 106496 (3%)
short[] 2093 (3%) 127212 (3%)
byte[] 2083 (3%) 1702285 (44%)
int[] 1631 (3%) 140964 (4%)
java.util.LinkedList$Entry 1556 (2%) 31120 (1%)
java.util.HashMap$Entry 1293 (2%) 31032 (1%)
java.lang.Object[] 792 (1%) 34504 (1%)
java.util.LinkedList 520 (1%) 10400 (0%)
java.lang.Class[] 491 (1%) 5608 (0%)
java.lang.reflect.Method 487 (1%) 37499 (1%)
java.util.HashMap$Entry[] 410 (1%) 34232 (1%)
java.util.HashMap 397 (1%) 15880 (0%)
java.lang.ref.SoftReference 359 (1%) 11488 (0%)
java.lang.Long 314 (1%) 5024 (0%)
java.lang.Integer 307 (0%) 3684 (0%)
java.util.Hashtable$Entry 303 (0%) 7272 (0%)
java.lang.String[] 284 (0%) 6868 (0%)
java.util.LinkedHashMap$Entry 262 (0%) 8384 (0%)
org.apache.lucene.index.CompoundFileReader$CSIndexInput 256 (0%) 13312 (0%)
java.util.concurrent.locks.ReentrantLock$NonfairSync 218 (0%) 5232 (0%)
java.lang.ref.Finalizer 210 (0%) 6720 (0%)
java.util.concurrent.ConcurrentHashMap$Segment 208 (0%) 6656 (0%)
java.util.concurrent.ConcurrentHashMap$HashEntry[] 208 (0%) 2984 (0%)
java.lang.reflect.Constructor 207 (0%) 12627 (0%)
java.lang.ThreadLocal$ThreadLocalMap$Entry 181 (0%) 5068 (0%)
java.util.WeakHashMap$Entry 175 (0%) 6300 (0%)
java.lang.ref.WeakReference 175 (0%) 4200 (0%)
Show in Instances View
Show Only Subclasses
[Class Name Filter]



Продолжал бы и дальше, но «поля здесь слишком узки» (с).

Напомню, что этот чудесный инструмент доступен большинству практически мгновенно (в $JAVA_HOME/bin/jvisualvm), а остальной части сообщества --- с небольшими телодвижениями. Надеюсь, теперь, когда отпрофилировать приложение будет так просто, культура performance-wise программирования привьётся куда лучше :)

Да, и помните завет Кнута: «Premature optimization is the root of all evil».