

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:

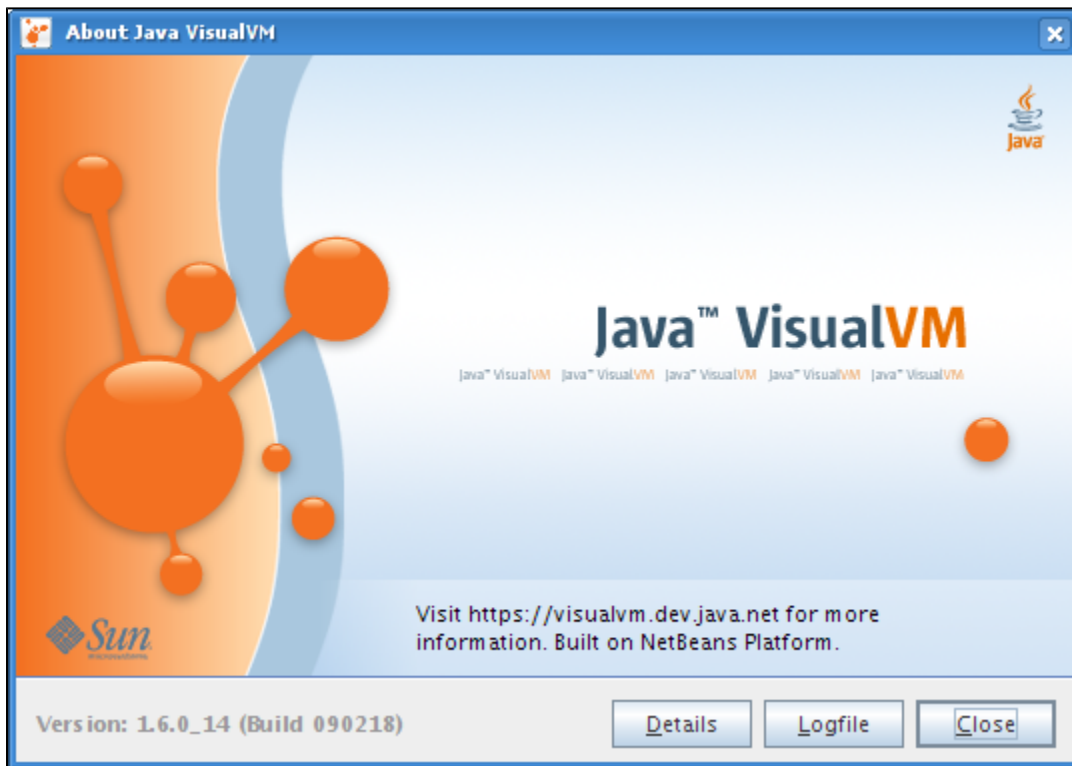
1. Как узнать «горячие места» в коде?
2. Как узнать, сколько занимает и что делает GC?
3. Как узнать, что происходит с памятью и где она «течёт»?

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

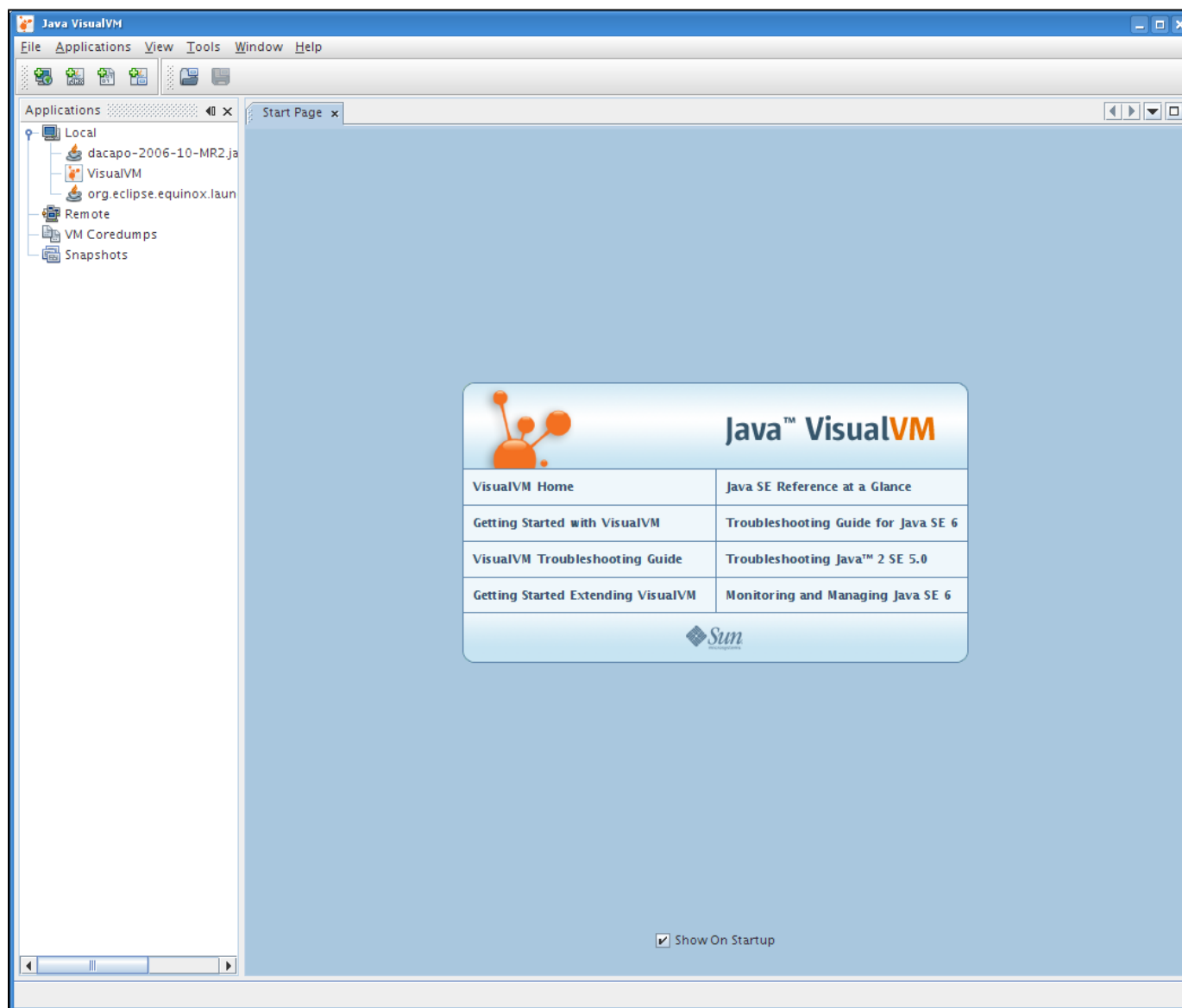


Подразумевается, что интерфейс VisualVM интуитивно понятен, поэтому подробно разбирать его я не буду. Постараюсь вместо этого рассказать о философии, стоящей за этим инструментом. В качестве тестируемого приложения я выбрал свободно распространяемый набор бенчмарков DaCapo, конкретно тест `lusearch`, 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.

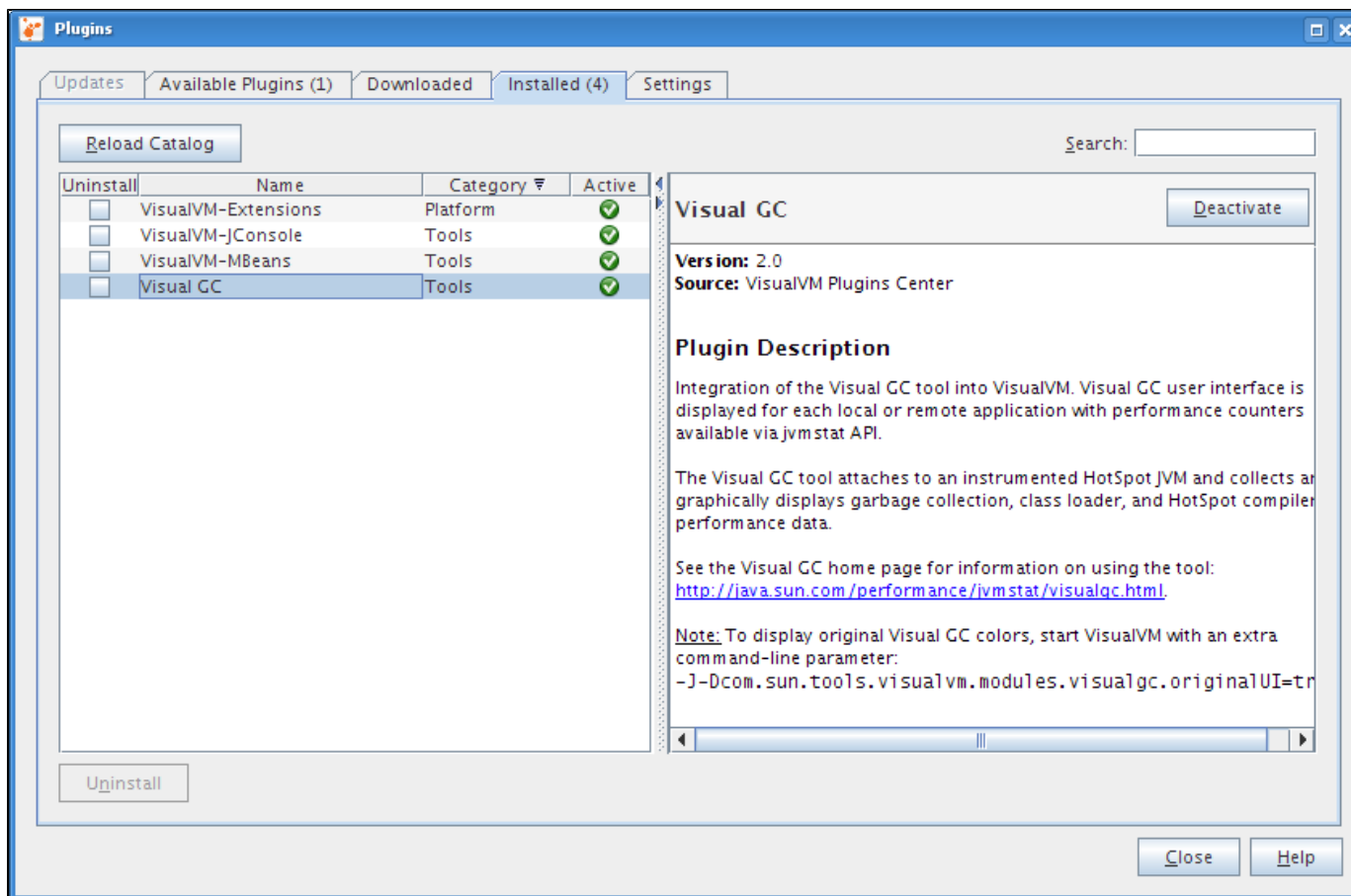
Перед тем, как начать саму демонстрацию, я скачал и установил несколько дополнений к VisualVM, которые покажу ниже. Для этого я запустил VisualVM:

`$JAVA_HOME/bin/jvisualvm`Если 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](#).

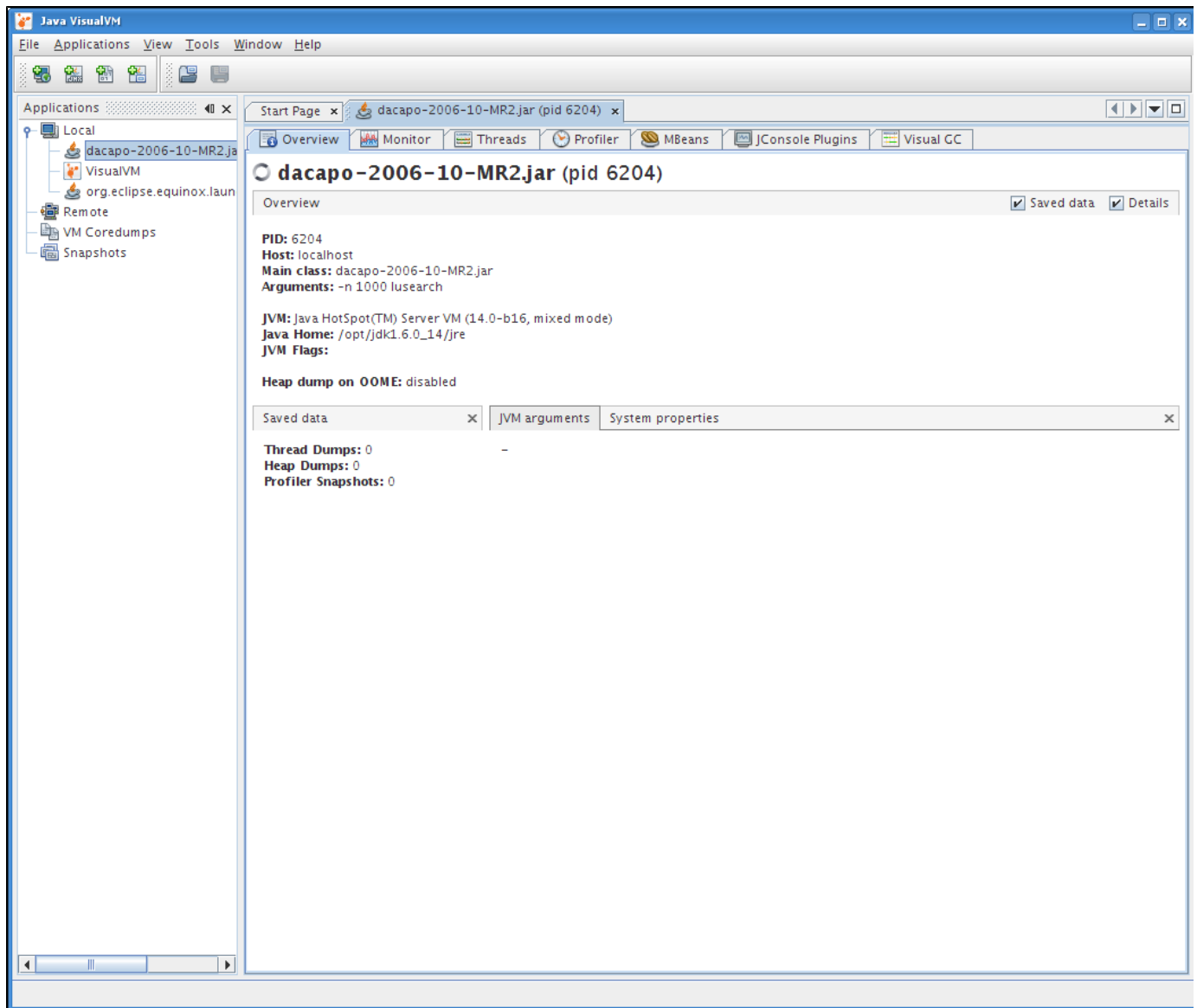
Вот так выглядит этот притаившийся зверь



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



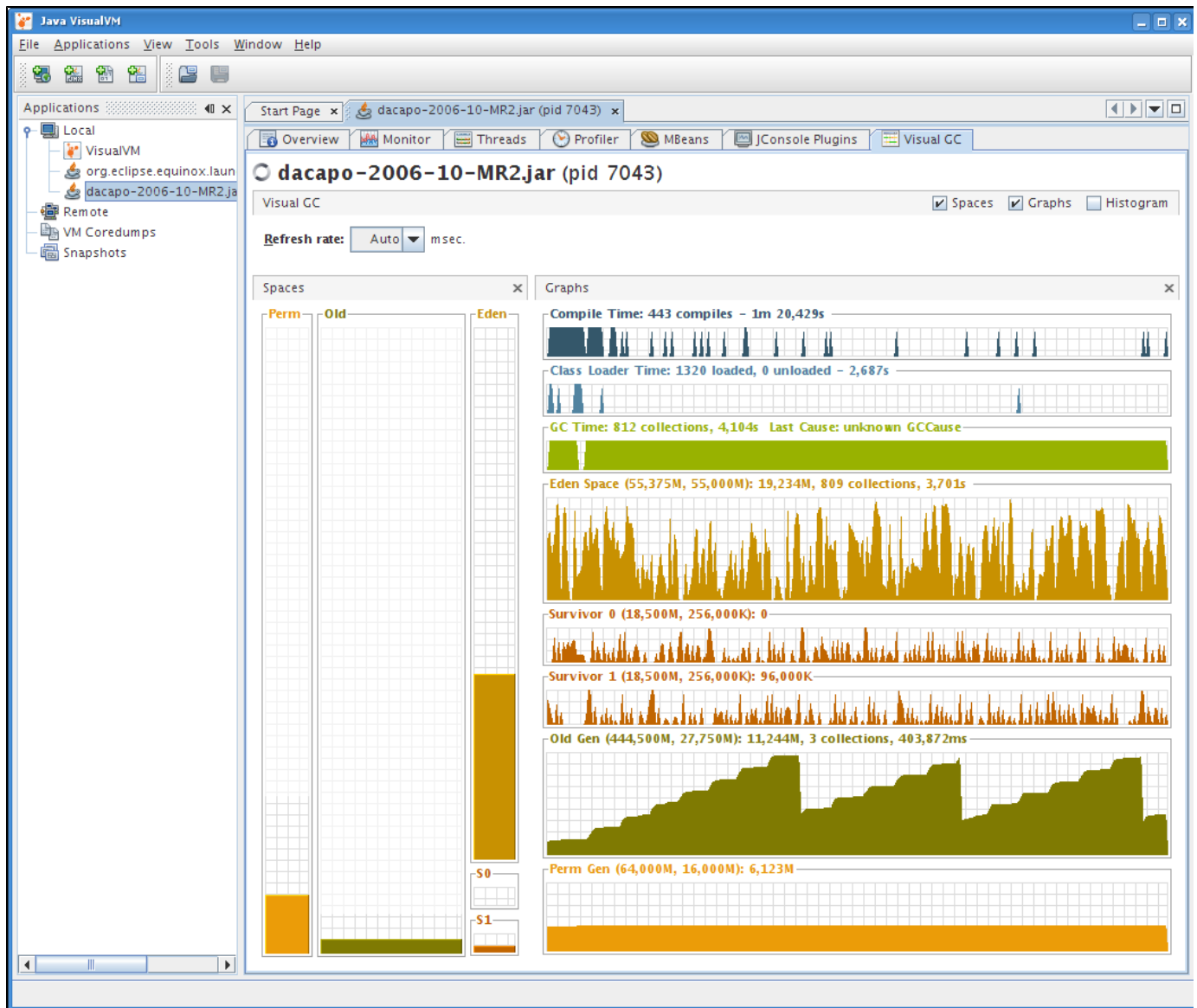
Всё готово! Время опробовать его в действии. Я запустил фоном DaCapo:lusearch
/opt/jdk1.6.0_14/bin/java -server -jar dacao-2006-10-MR2.jar -n 100 lusearchи отправился в VisualVM. Нашёл
в дереве слева интересующий меня java-процесс (можно увидеть, что там торчат ещё сам VisualVM и мой
Eclipse), сделал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

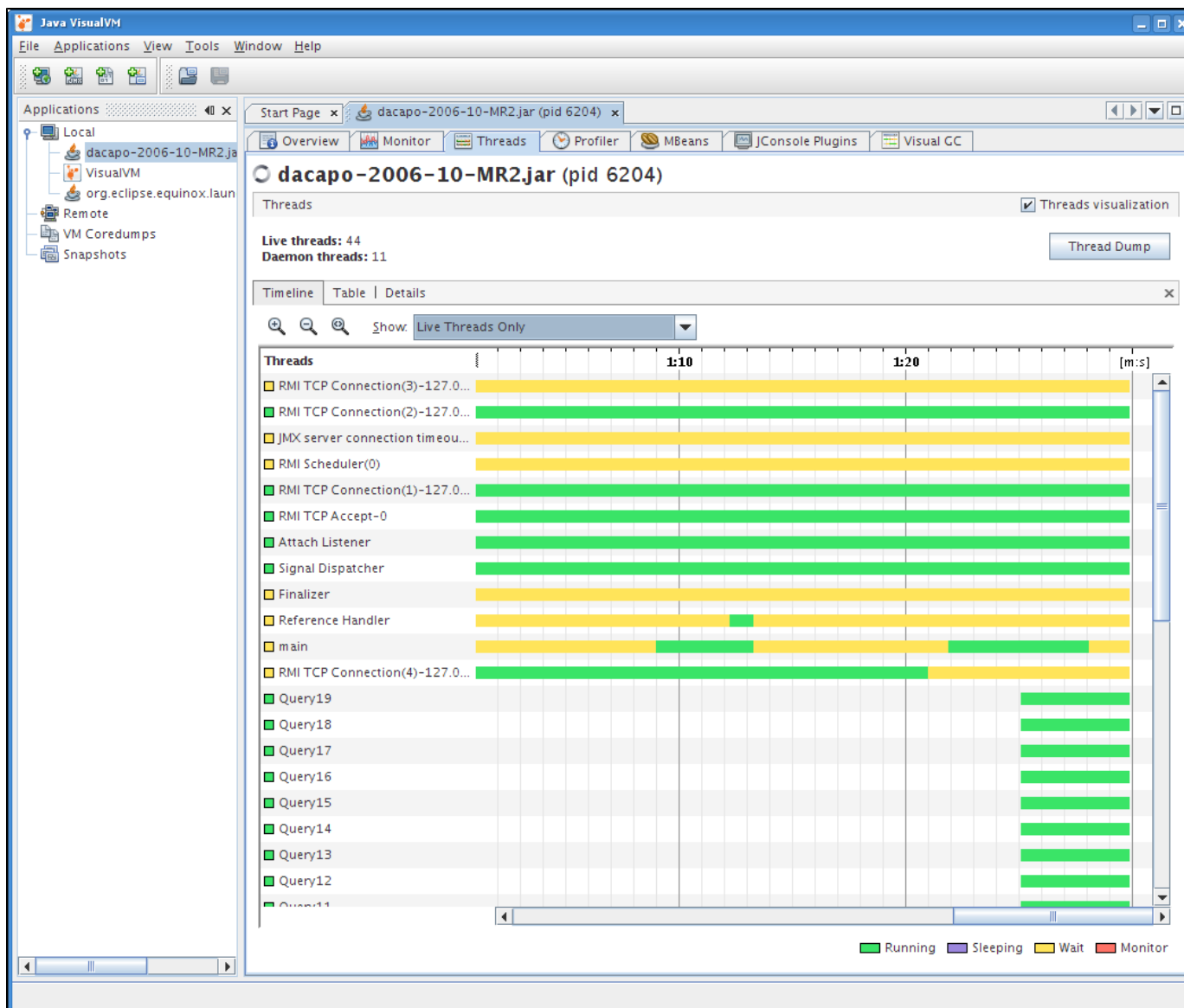
На скриншоте легко можно увидеть графики утилизации CPU (включая активность GC), количество загруженных классов и созданных нитей, состояние памяти.

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

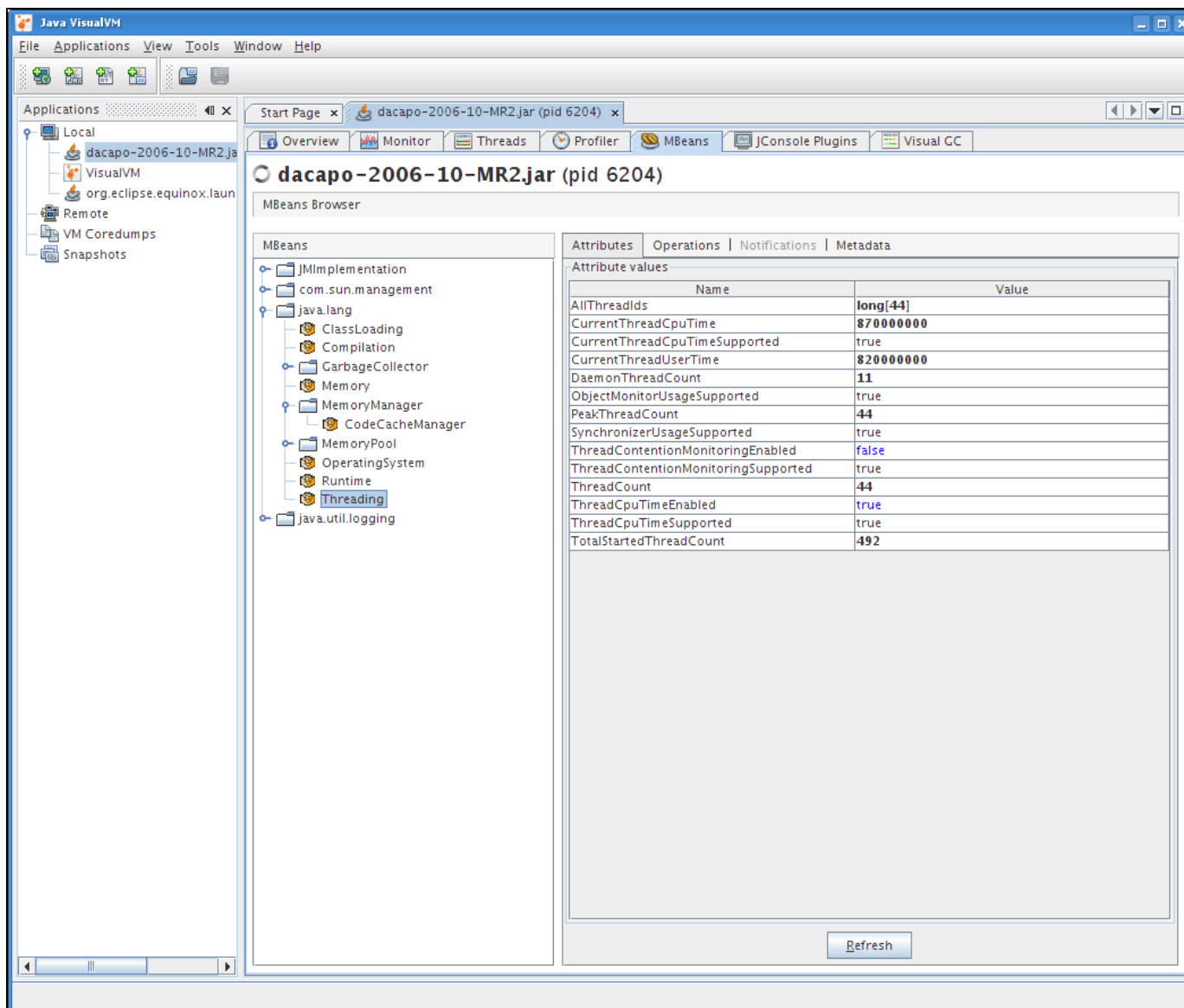


То, что здесь происходит, будет понятно тем, кто знает, как устроен сборщик мусора. Конкретно на этом скриншоте можно увидеть три пространства для объектов: постоянное, старое и молодое. Последнее разбито на ещё на три подпространства: эдем и два пространства под выжившие объекты. Данные обновляются 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?

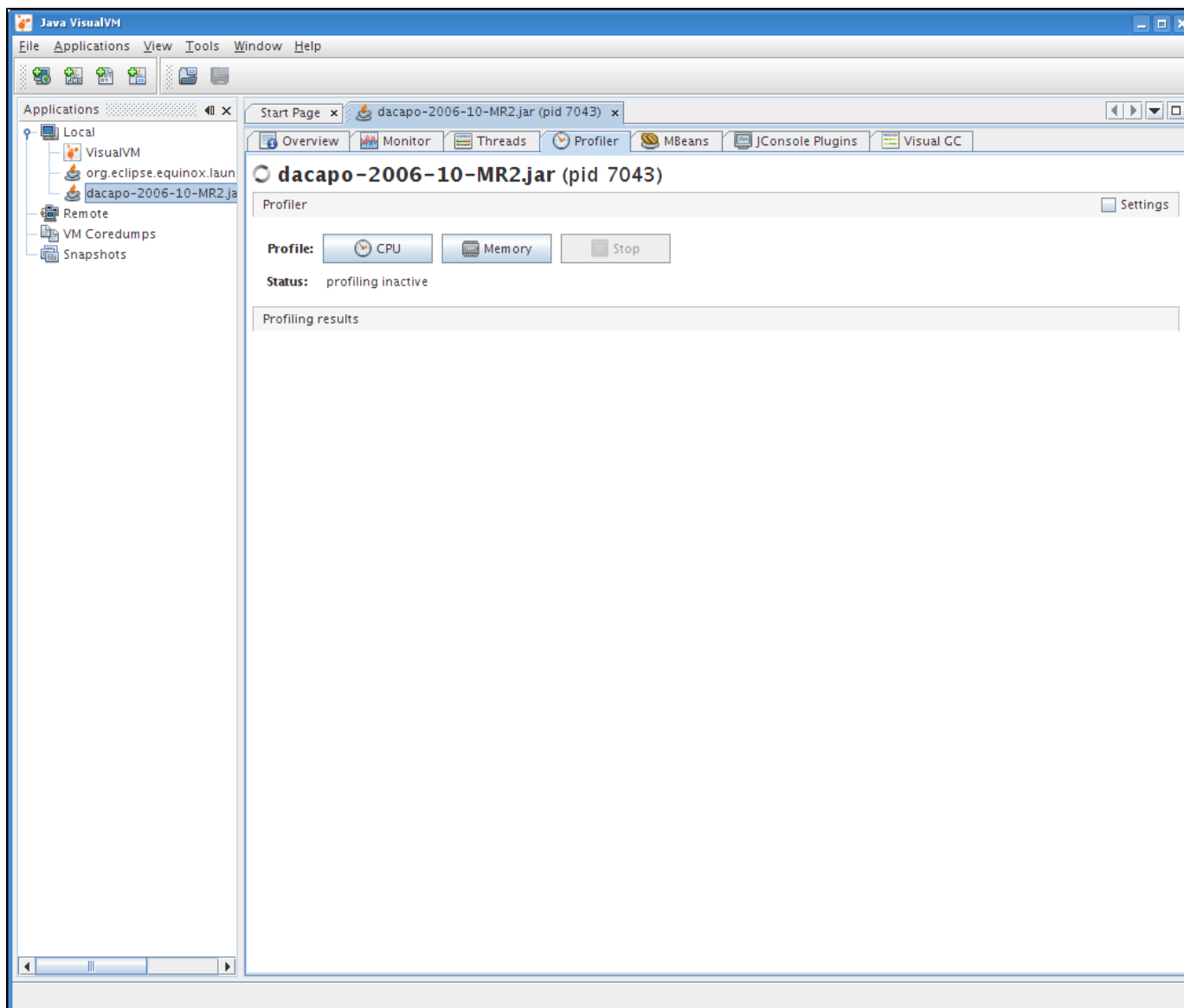
Вернёмся пока к закладкам. На закладке Threads можно увидеть подробную статистику по нитям. Там есть несколько режимов просмотра, оставляю их любопытному читателю.



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»:

Java VisualVM

File Applications View Tools Window Help

Applications

Local

- VisualVM
- org.eclipse.equinox.launcher
- dacapo-2006-10-MR2.jar

Remote

- VM CoreDumps
- Snapshots

Start Page x dacapo-2006-10-MR2.jar (pid 7951) x

Overview Monitor Threads Profiler MBeans JConsole Plugins Visual GC

dacapo-2006-10-MR2.jar (pid 7951)

Profiler Settings

Profile: CPU Memory Stop

Status: profiling running (318 methods instrumented)

Profiling results

Take Snapshot of Collected Results

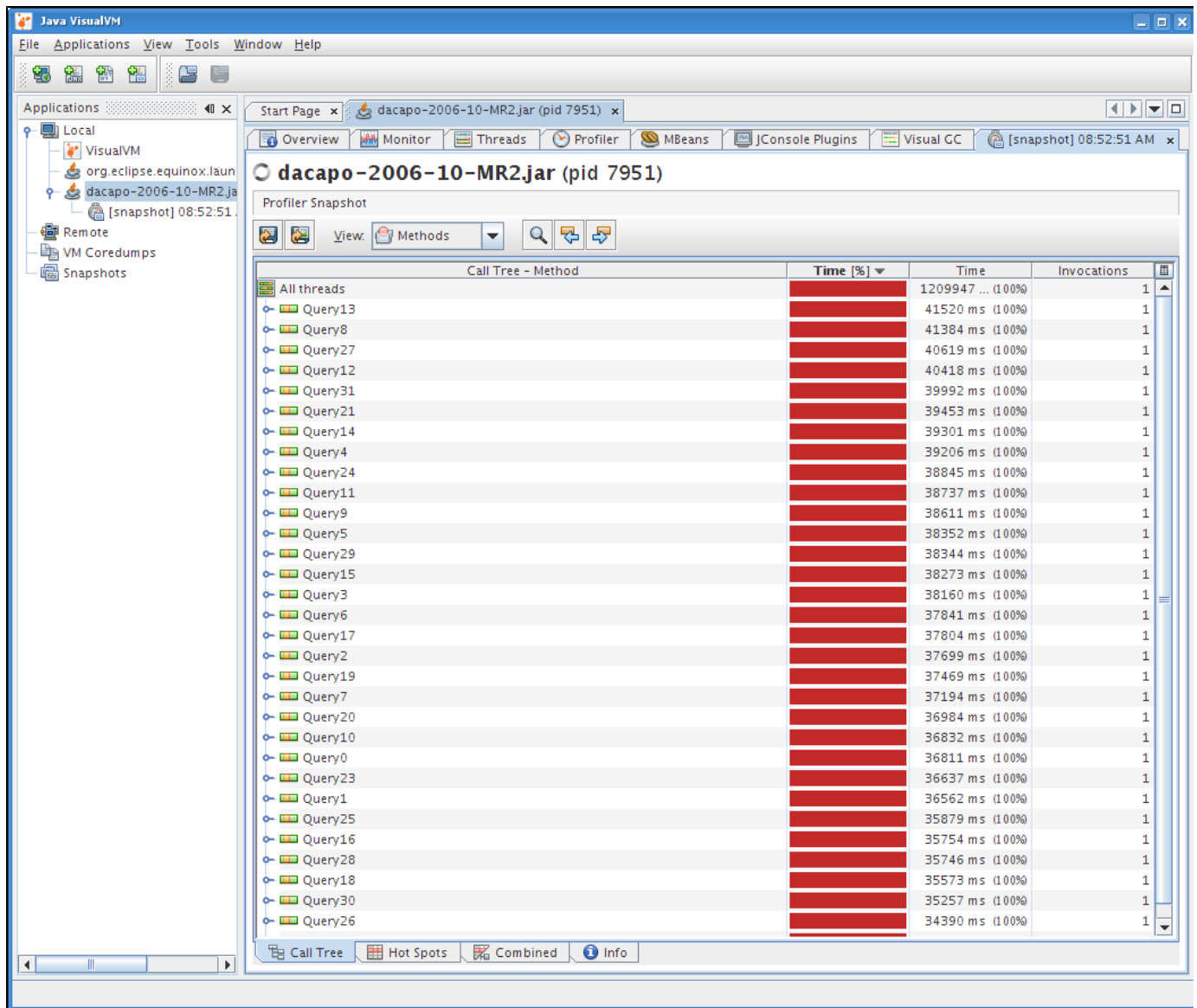
Hot Spots - Method

	Self time [%]	Self time	Invocations
org.apache.lucene.store.BufferedIndexInput.readByte ()	12,7%	136921 ms	262267
org.apache.lucene.store.IndexInput.readVInt ()	10,5%	113207 ms	1114525
org.apache.lucene.index.TermBuffer.read (org.apache.lucene.store.IndexInput, org....	8,3%	90138 ms	269896
org.apache.lucene.index.SegmentTermEnum.next ()	8,2%	88519 ms	269931
org.apache.lucene.analysis.standard.StandardTokenizerTokenManager.jjCheckNAddT...	5,6%	60984 ms	343670
org.apache.lucene.store.IndexInput.readVLong ()	5%	53806 ms	546423
org.apache.lucene.store.IndexInput.readChars (char[], int, int)	4,6%	49987 ms	272069
org.apache.lucene.analysis.standard.StandardTokenizerTokenManager.jjCheckNAdd...	4,5%	48723 ms	921529
org.apache.lucene.index.TermBuffer.compareTo (org.apache.lucene.index.TermBuff...	2,9%	31501 ms	271245
org.apache.lucene.index.TermBuffer.setTextLength (int)	2,7%	29517 ms	551827
org.apache.lucene.index.SegmentTermEnum.scanTo (org.apache.lucene.index.Term)	2,6%	28044 ms	8020
org.apache.lucene.analysis.standard.StandardTokenizerTokenManager.jjMoveNfa_0...	2,6%	27816 ms	4031
org.apache.lucene.index.TermBuffer.set (org.apache.lucene.index.TermBuffer)	2,5%	26670 ms	269899
org.apache.lucene.index.FieldInfos.fieldName (int)	2,5%	26590 ms	269889
org.apache.lucene.index.FieldInfos.fieldInfo (int)	1,4%	14755 ms	272062
org.apache.lucene.queryParser.QueryParserTokenManager.jjCheckNAdd (int)	1,3%	14473 ms	269137
org.apache.lucene.search.Similarity.tf (int)	1,3%	14021 ms	128086
org.apache.lucene.index.TermBuffer.compareChars (char[], int, char[], int)	1,2%	13287 ms	271235
org.apache.lucene.analysis.standard.StandardTokenizerTokenManager.jjCheckNAddS...	1,2%	12713 ms	46459
org.apache.lucene.queryParser.QueryParserTokenManager.jjMoveNfa_3 (int, int)	1,1%	11574 ms	4033
org.apache.lucene.store.BufferedIndexInput.readBytes (byte[], int, int)	0,7%	7873 ms	5573
org.apache.lucene.search.DefaultSimilarity.tf (float)	0,7%	7726 ms	128086
org.apache.lucene.queryParser.QueryParserTokenManager.jjCheckNAddStates (int, ...	0,6%	6946 ms	38448
org.apache.lucene.queryParser.QueryParserTokenManager.jjCheckNAddTwoStates ...	0,6%	6910 ms	68832
org.apache.lucene.index.TermInfosReader.get (org.apache.lucene.index.Term)	0,6%	6188 ms	8020
org.apache.lucene.search.TermScorer.<init> (org.apache.lucene.search.Weight, or...	0,6%	6033 ms	4001

[Method Name Filter]

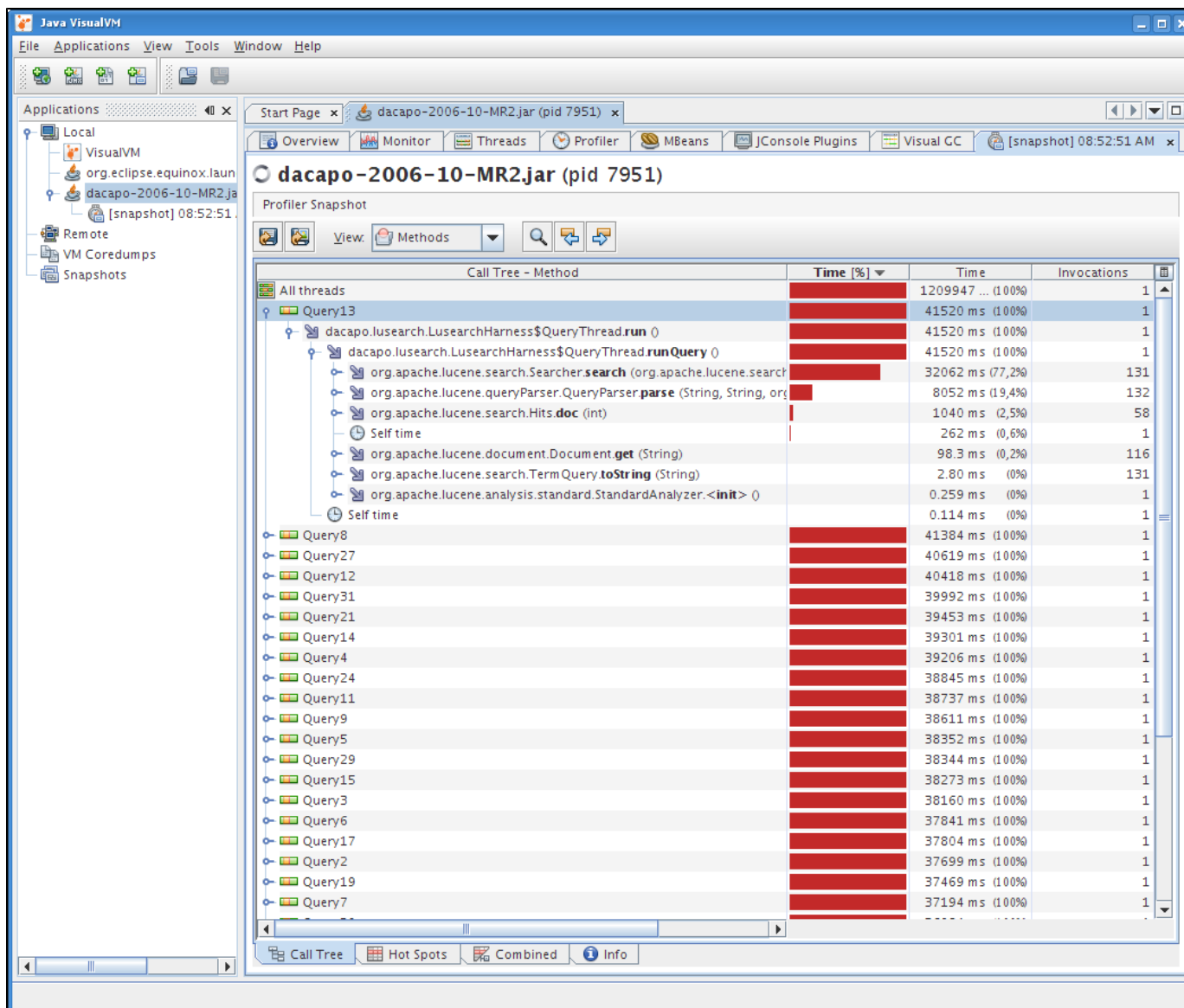
Интересная и полезная особенность состоит в том, что приложение всё ещё работает, профайлер инструментировал приложение на лету. Вот и первые результаты, видно, какой метод занимает больше всего времени. Но это только часть всего дела: неясно ведь, откуда его зовут.

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



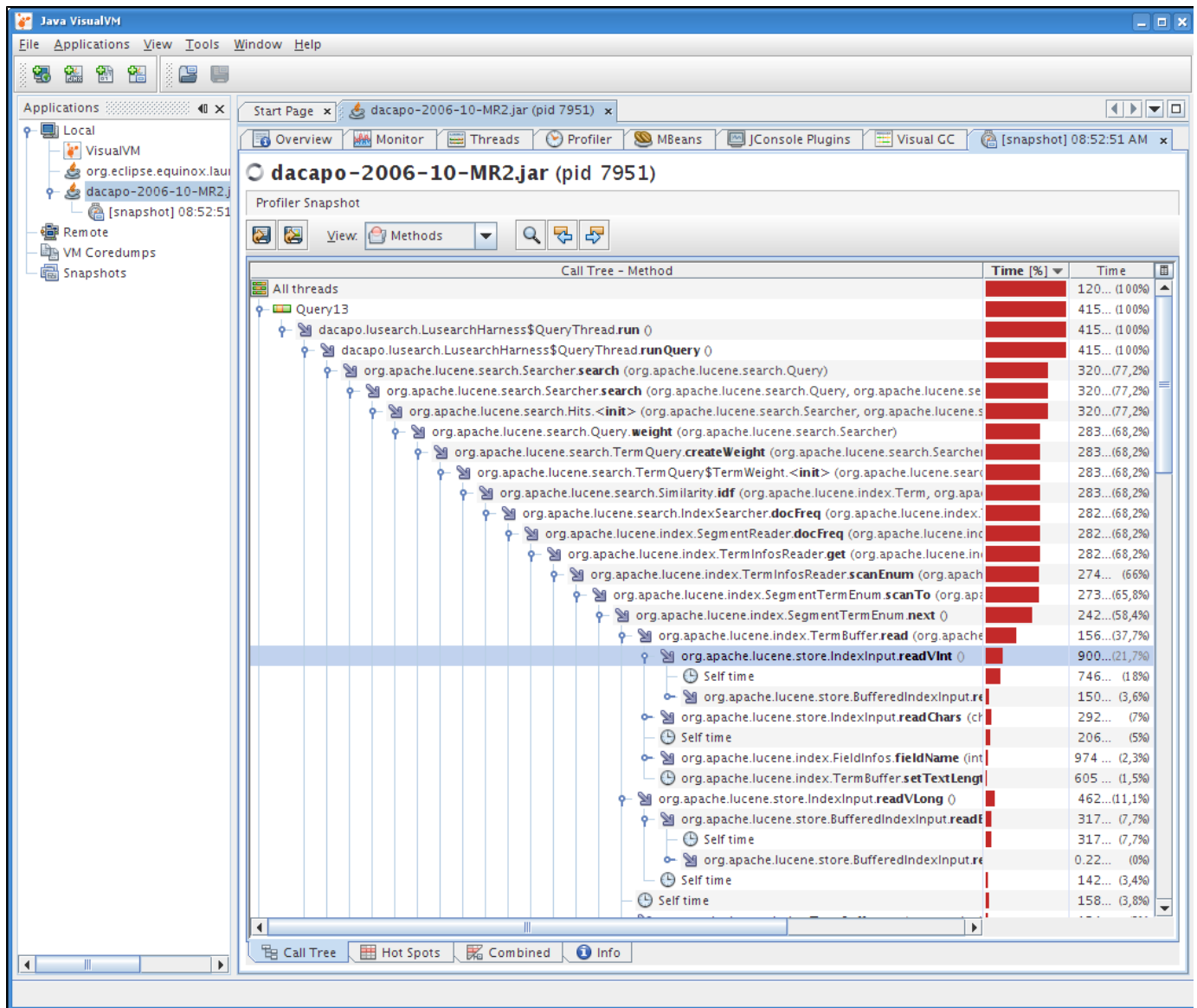
Это — все нитки, которые были пойманы профайлером.

Развернём одну из них:



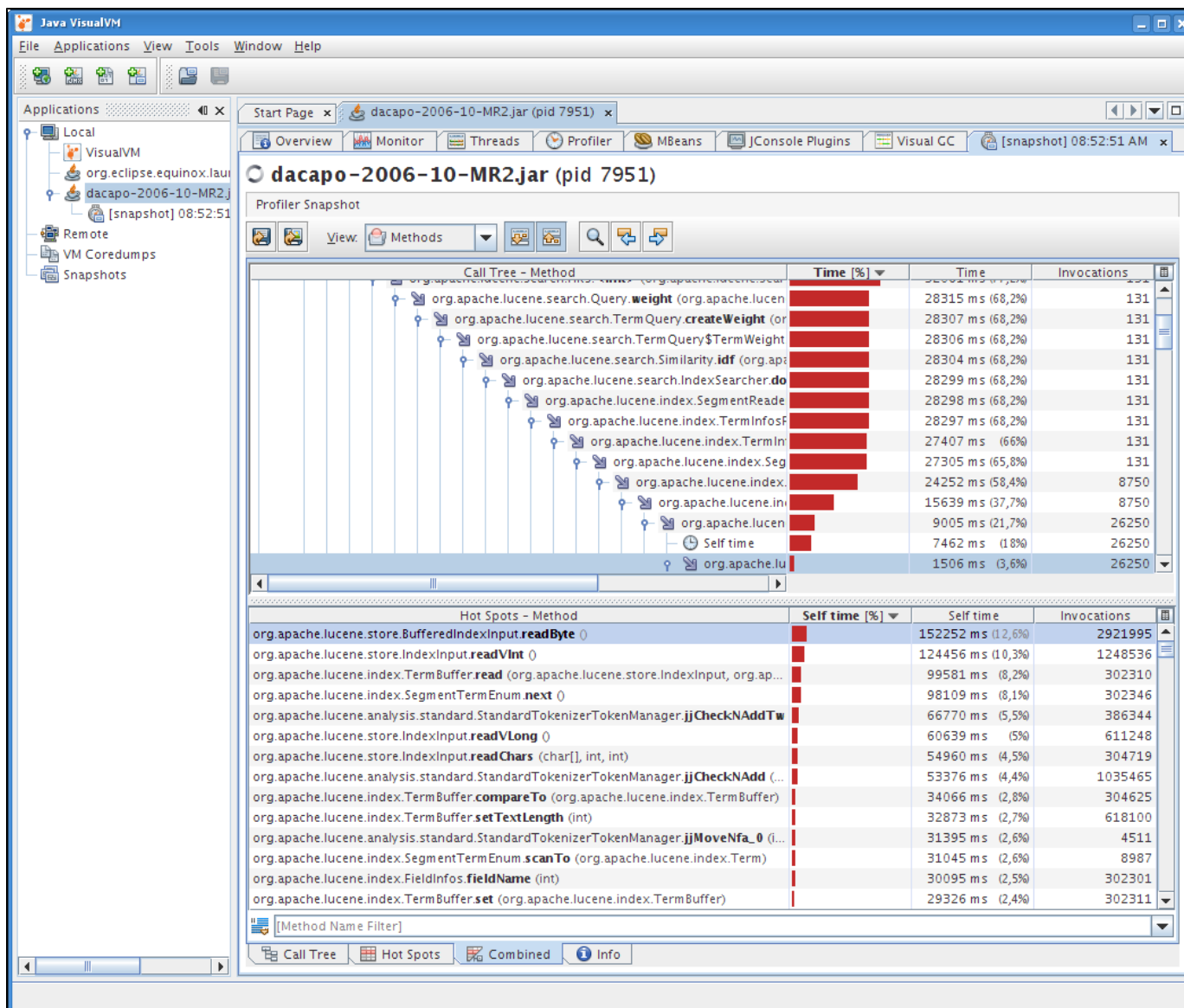
Видно, что исполнение началось с вызова метода `QueryThread.run()` и дальше оно разбилось на две стадии: `QueryParser.parse(...)` и `Searcher.search(...)`. Кажется, это правильно.

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:

Java VisualVM

File Applications View Tools Window Help

Applications

- Local
 - VisualVM
 - org.eclipse.equinox.lau...
 - dacapo-2006-10-MR2.j...
- Remote
 - VM CoreDumps
 - Snapshots

Start Page x dacapo-2006-10-MR2.jar (pid 8141) x

Overview Monitor Threads Profiler MBeans JConsole Plugins Visual GC

dacapo-2006-10-MR2.jar (pid 8141)

Profiler Settings

Profile: CPU Memory Stop

Status: profiling running (1 610 classes instrumented, tracking each 10th object)

Profiling results

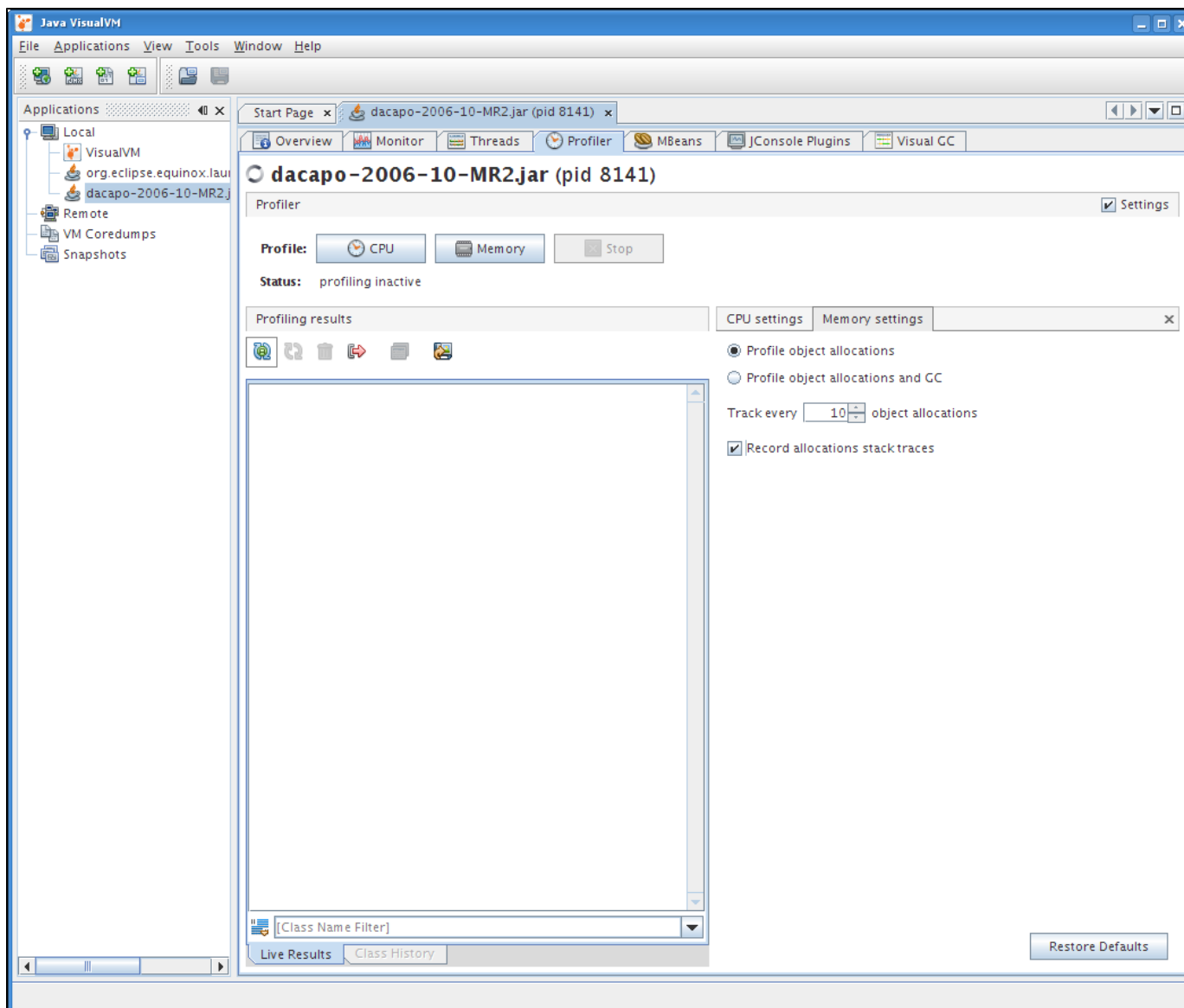
Class Name - Live Allocated Objects

Class Name - Live Allocated Objects	Live Bytes	Live Bytes	Live Objects	Generations
char[]	916 04...	(64,5%)	2 048 (19,9%)	4
int[]	214 16...	(15,1%)	900 (8,8%)	4
java.lang.Object[]	71 008...	(5%)	763 (7,4%)	4
java.lang.String	24 768...	(1,7%)	1 032 (10%)	4
org.apache.lucene.queryParser.Token	15 800...	(1,1%)	395 (3,8%)	4
float[]	12 672...	(0,9%)	88 (0,9%)	1
java.util.Vector	12 408...	(0,9%)	517 (5%)	4
org.apache.lucene.analysis.standard.Token	11 600...	(0,8%)	290 (2,8%)	3
org.apache.lucene.queryParser.QueryParser	10 416...	(0,7%)	93 (0,9%)	2
java.io.IOException	9 144 8	(0,6%)	381 (3,7%)	1
java.io.StringReader	9 120 8	(0,6%)	285 (2,8%)	3
byte[]	8 712 8	(0,6%)	23 (0,2%)	2
org.apache.lucene.index.SegmentTermDocs	8 184 8	(0,6%)	93 (0,9%)	2
java.util.TreeMap\$Entry	7 616 8	(0,5%)	238 (2,3%)	4
org.apache.lucene.queryParser.FastCharStream	6 208 8	(0,4%)	194 (1,9%)	3
org.apache.lucene.search.TermScorer	5 544 8	(0,4%)	99 (1%)	2
org.apache.lucene.analysis.standard.StandardTokenizerTokenManager	5 432 8	(0,4%)	97 (0,9%)	2
org.apache.lucene.queryParser.QueryParserTokenManager	5 152 8	(0,4%)	92 (0,9%)	3
org.apache.lucene.index.Term	4 816 8	(0,3%)	301 (2,9%)	4
org.apache.lucene.search.Hits	4 608 8	(0,3%)	96 (0,9%)	1
org.apache.lucene.queryParser.QueryParser\$JJCalls	4 584 8	(0,3%)	191 (1,9%)	2
org.apache.lucene.analysis.standard.StandardTokenizer	4 416 8	(0,3%)	92 (0,9%)	1
java.io.ObjectStreamClass\$WeakClassKey	3 648 8	(0,3%)	114 (1,1%)	1
org.apache.lucene.search.TermQuery\$TermWeight	3 072 8	(0,2%)	96 (0,9%)	2
org.apache.lucene.analysis.standard.FastCharStream	3 008 8	(0,2%)	94 (0,9%)	2

[Class Name Filter]

Live Results Class History

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:

Java VisualVM

File Applications View Tools Window Help

Applications

- Local
 - VisualVM
 - org.eclipse.equinox.lau
 - dacapo-2006-10-MR2.j
- Remote
 - VM CoreDumps
 - Snapshots

Start Page x dacapo-2006-10-MR2.jar (pid 8141) x

Overview Monitor Threads Profiler MBeans JConsole Plugins Visual GC

dacapo-2006-10-MR2.jar (pid 8141)

Profiler Settings

Profile: CPU Memory Stop

Status: profiling running (1 610 classes instrumented, tracking each 10th object)

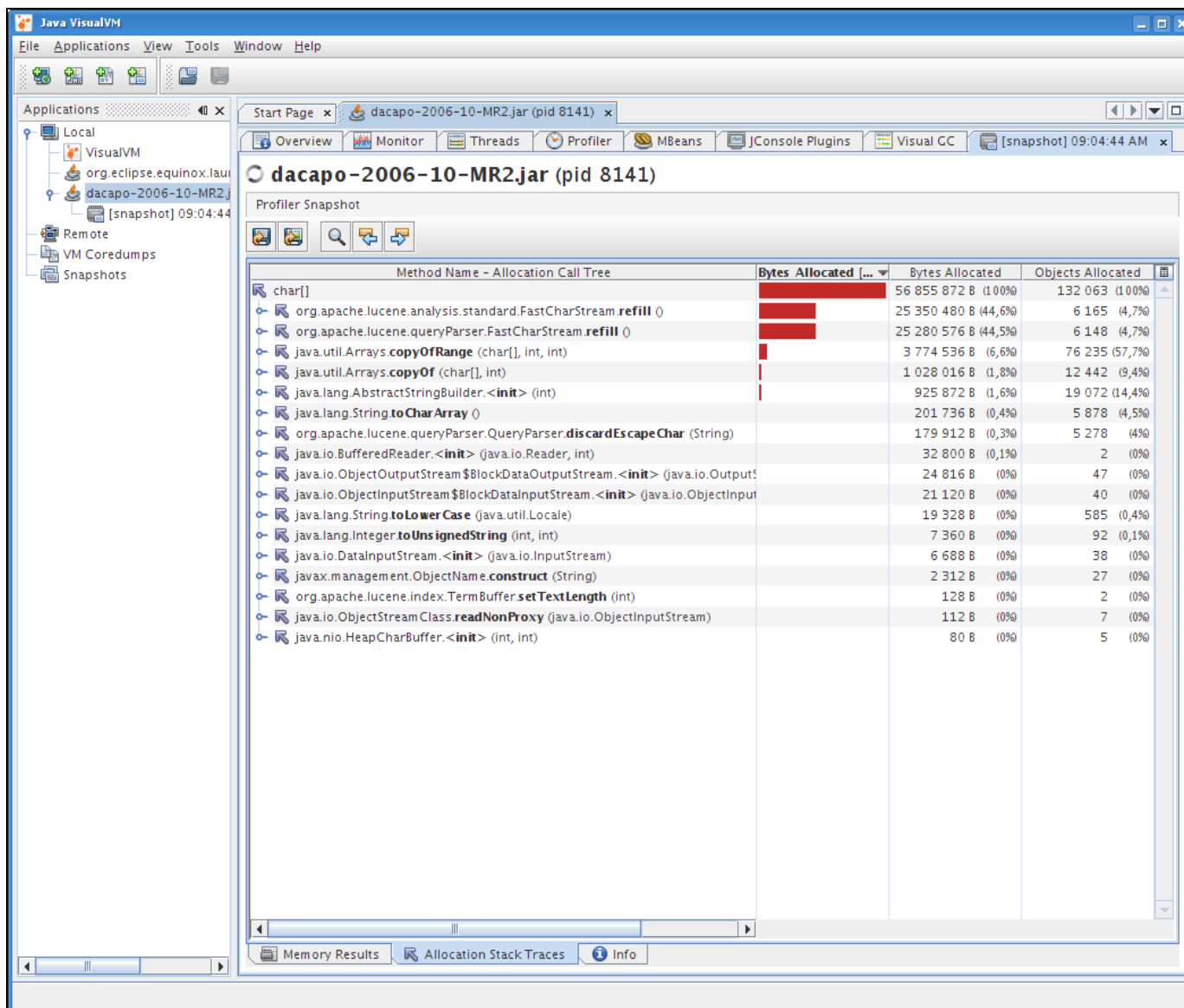
Profiling results

Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]	29 837 256 B (67%)	535 406 (19,8%)	
int[]	6 272 496 B (14,1%)	246 538 (9,1%)	
java.lang.Object[]	2 159 040 B (4,8%)	186 241 (6,9%)	
java.lang.String	707 568 B (1,6%)	273 010 (10,1%)	
org.apache.lucene.queryParser.Token	462 920 B (1%)	109 168 (4%)	
float[]	412 448 B (0,9%)	27 300 (1%)	
java.util.Vector	365 472 B (0,8%)	143 609 (5,3%)	
org.apache.lucene.analysis.standard.Token	347 720 B (0,8%)	81 870 (3%)	
org.apache.lucene.queryParser.QueryParser	319 872 B (0,7%)	27 290 (1%)	
java.io.IOException	280 464 B (0,6%)	109 158 (4%)	
java.io.StringReader	277 952 B (0,6%)	81 877 (3%)	
org.apache.lucene.index.SegmentTermDocs	255 816 B (0,6%)	27 295 (1%)	
byte[]	246 184 B (0,6%)	3 983 (0,1%)	
org.apache.lucene.queryParser.Fast CharStream	182 656 B (0,4%)	54 582 (2%)	
org.apache.lucene.analysis.standard.StandardTokenizerTokenManager	161 336 B (0,4%)	27 293 (1%)	
org.apache.lucene.search.TermScorer	160 664 B (0,4%)	27 295 (1%)	
org.apache.lucene.queryParser.QueryParserTokenManager	159 208 B (0,4%)	27 290 (1%)	
org.apache.lucene.index.CompoundFileReader\$CSIndexInput	146 608 B (0,3%)	24 229 (0,9%)	
org.apache.lucene.index.Term	139 488 B (0,3%)	81 592 (3%)	
org.apache.lucene.queryParser.QueryParser\$JJCalls	137 328 B (0,3%)	54 581 (2%)	
org.apache.lucene.analysis.standard.StandardTokenizer	137 328 B (0,3%)	27 292 (1%)	
org.apache.lucene.search.Hits	136 368 B (0,3%)	27 299 (1%)	
org.apache.lucene.search.TermQuery\$TermWeight	92 352 B (0,2%)	27 294 (1%)	
org.apache.lucene.analysis.Token	91 584 B (0,2%)	27 286 (1%)	
org.apache.lucene.analysis.standard.Fast CharStream	91 232 B (0,2%)	27 295 (1%)	

[Class Name Filter]

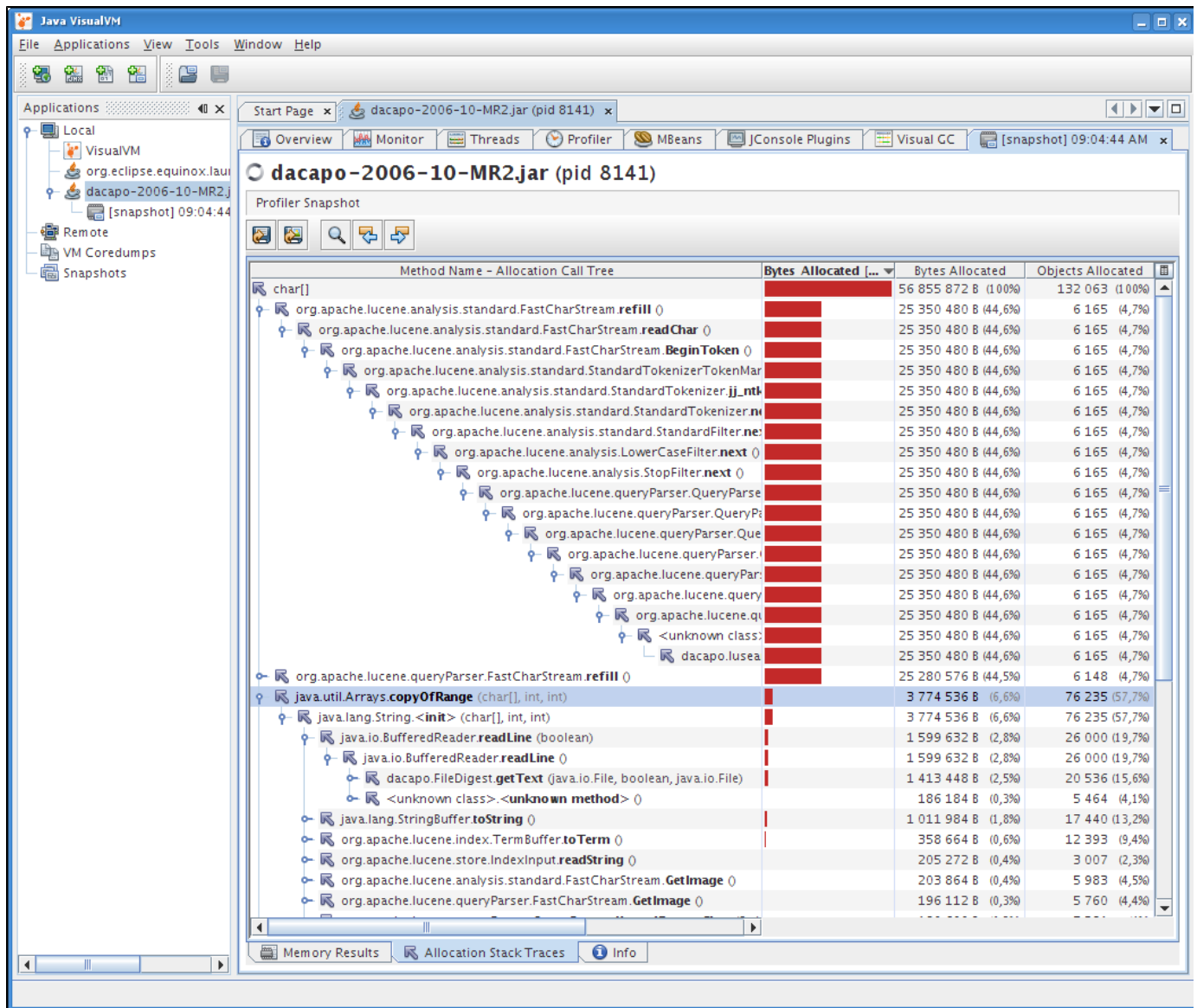
Live Results Class History

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? Откуда:



Несмотря на то, что Мемогу 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.launcher
- dacapo-2006-10-MR2.jar
- [snapshot] 09:04:44 AM
- [snapshot] 09:06:27 AM
- [heapdump] 09:06:59 AM

Remote

- VM CoreDumps
- Snapshots

Start Page x dacapo-2006-10-MR2.jar (pid 8141) x

[snapshot] 09:04:44 AM x [snapshot] 09:06:27 AM x [heapdump] 09:06:59 AM x

Overview Monitor Threads Profiler MBeans JConsole Plugins Visual GC

dacapo-2006-10-MR2.jar (pid 8141)

Summary Classes Instances

Class Name	Instances [%]	Instances	Size
org.apache.lucene.index.TermInfo	13847 (22%)	332328 (9%)	
org.apache.lucene.index.Term	13739 (22%)	629358 (16%)	
short[]	6688 (11%)	214016 (6%)	
byte[]	6656 (11%)	106496 (3%)	
int[]	2093 (3%)	127212 (3%)	
java.util.LinkedList\$Entry	2083 (3%)	1702285 (44%)	
java.util.HashMap\$Entry	1631 (3%)	140964 (4%)	
java.lang.Object[]	1556 (2%)	31120 (1%)	
java.util.LinkedList	1293 (2%)	31032 (1%)	
java.lang.Class[]	792 (1%)	34504 (1%)	
java.lang.reflect.Method	520 (1%)	10400 (0%)	
java.util.HashMap\$Entry[]	491 (1%)	5608 (0%)	
java.util.HashMap	487 (1%)	37499 (1%)	
java.lang.ref.SoftReference	410 (1%)	34232 (1%)	
java.lang.Long	397 (1%)	15880 (0%)	
java.lang.Integer	359 (1%)	11488 (0%)	
java.util.Hashtable\$Entry	314 (1%)	5024 (0%)	
java.lang.String[]	307 (0%)	3684 (0%)	
java.util.LinkedHashMap\$Entry	303 (0%)	7272 (0%)	
org.apache.lucene.index.CompoundFileReader\$CSIndexInput	284 (0%)	6868 (0%)	
java.util.concurrent.locks.ReentrantLock\$NonfairSync	262 (0%)	8384 (0%)	
java.lang.ref.Finalizer	256 (0%)	13312 (0%)	
java.util.concurrent.ConcurrentHashMap\$Segment	218 (0%)	5232 (0%)	
java.util.concurrent.ConcurrentHashMap\$HashEntry[]	210 (0%)	6720 (0%)	
java.lang.reflect.Constructor	208 (0%)	6656 (0%)	
java.lang.ThreadLocal\$ThreadLocalMap\$Entry	208 (0%)	2984 (0%)	
java.util.WeakHashMap\$Entry	207 (0%)	12627 (0%)	
java.lang.ref.WeakReference	181 (0%)	5068 (0%)	
	175 (0%)	6300 (0%)	
	175 (0%)	4200 (0%)	

[Class Name Filter]

Java VisualVM

File Applications View Tools Window Help

Applications

- Local
 - VisualVM
 - org.eclipse.equinox.launcher
 - dacapo-2006-10-MR2.jar
 - [snapshot] 09:04:44
 - [snapshot] 09:06:27
 - [heapdump] 09:06:59
 - Remote
 - VM CoreDumps
 - Snapshots

Start Page x dacapo-2006-10-MR2.jar (pid 8141) x

[snapshot] 09:04:44 AM x [snapshot] 09:06:27 AM x [heapdump] 09:06:59 AM x

Overview Monitor Threads Profiler MBeans JConsole Plugins Visual GC

dacapo-2006-10-MR2.jar (pid 8141)

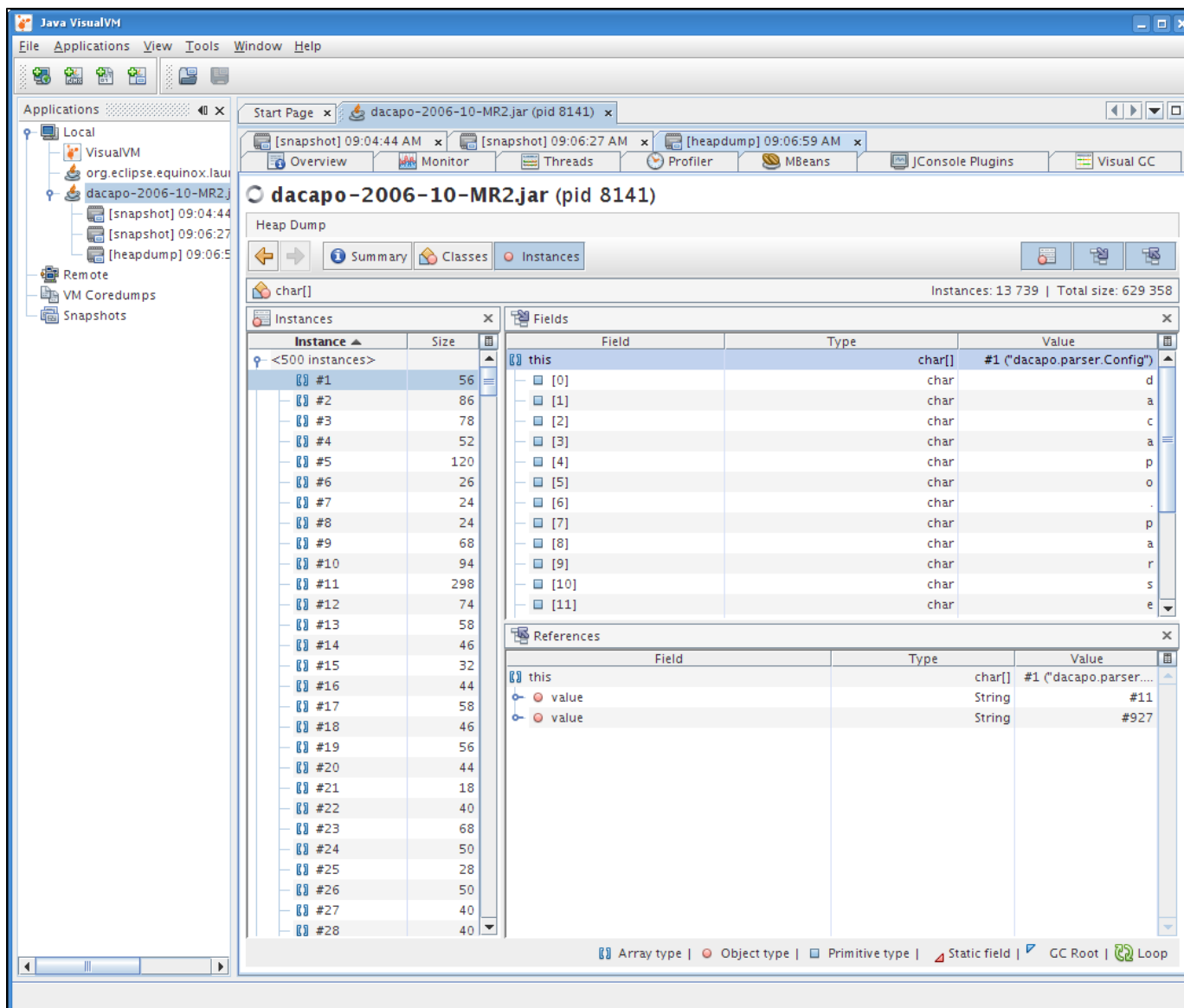
Heap Dump

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%)	

[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».