Профайлер в каждый дом

Владимир Ситников

NetCracker®

Кто я?

Владимир Ситников

System Performance @ NetCracker

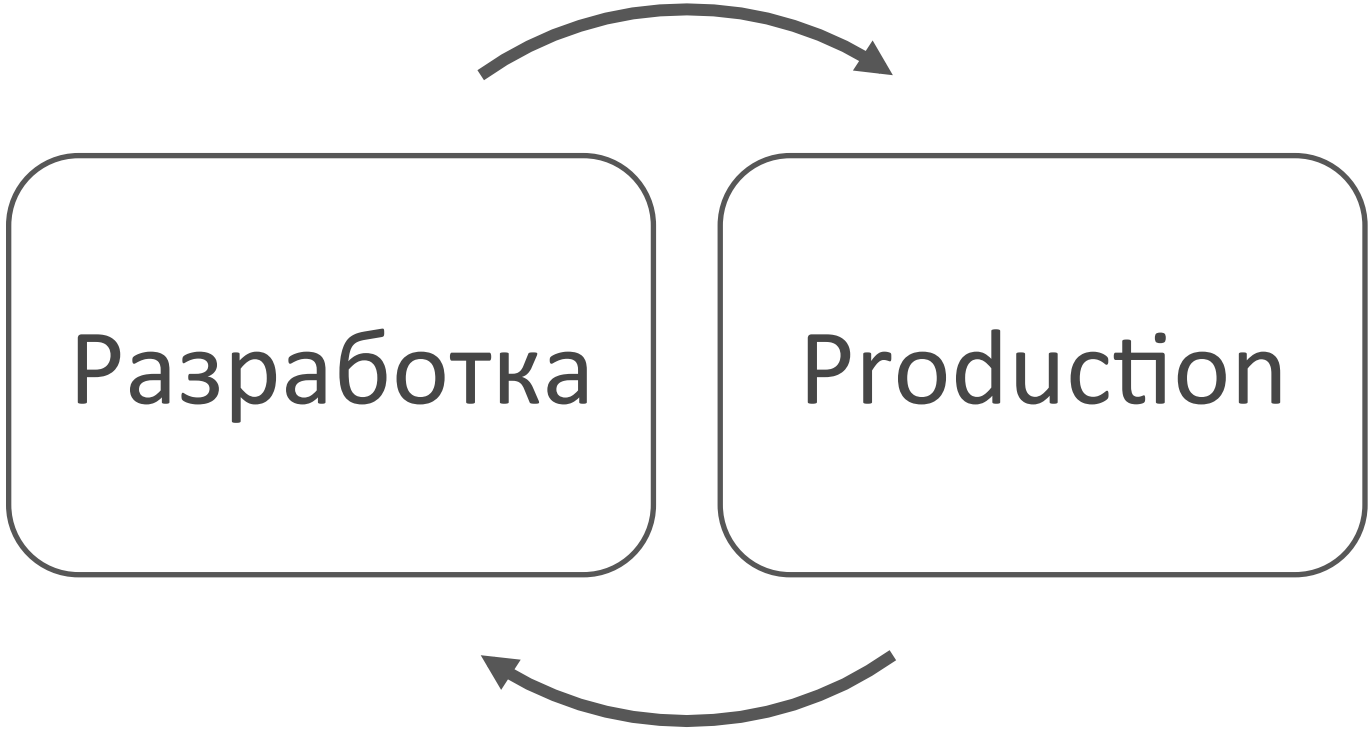
sitnikov@netcracker.com

@VladimirSitnikv

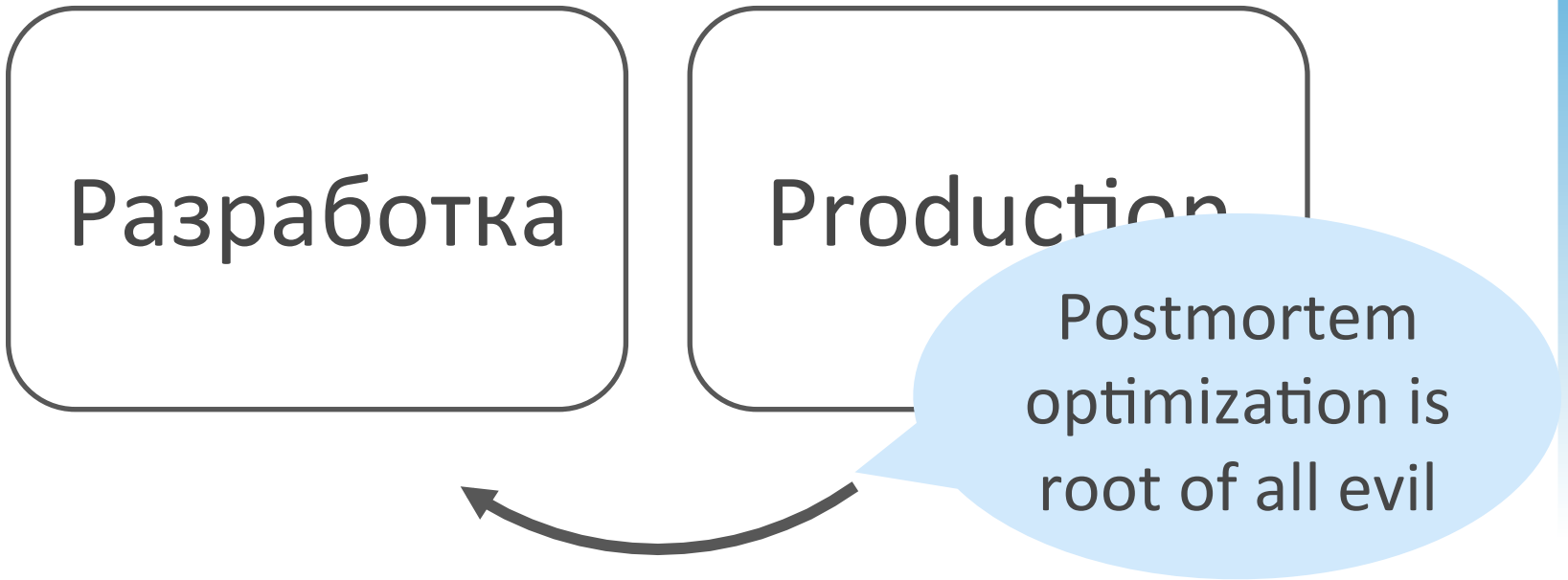
План

- Зачем нужны профайлеры
- Зачем создавать свой профайлер, если есть готовое?
- Выбираем подход
  - Sampling vs instrumentation
- Пишем
- Визуализация и анализ результатов

Типичный цикл разработки

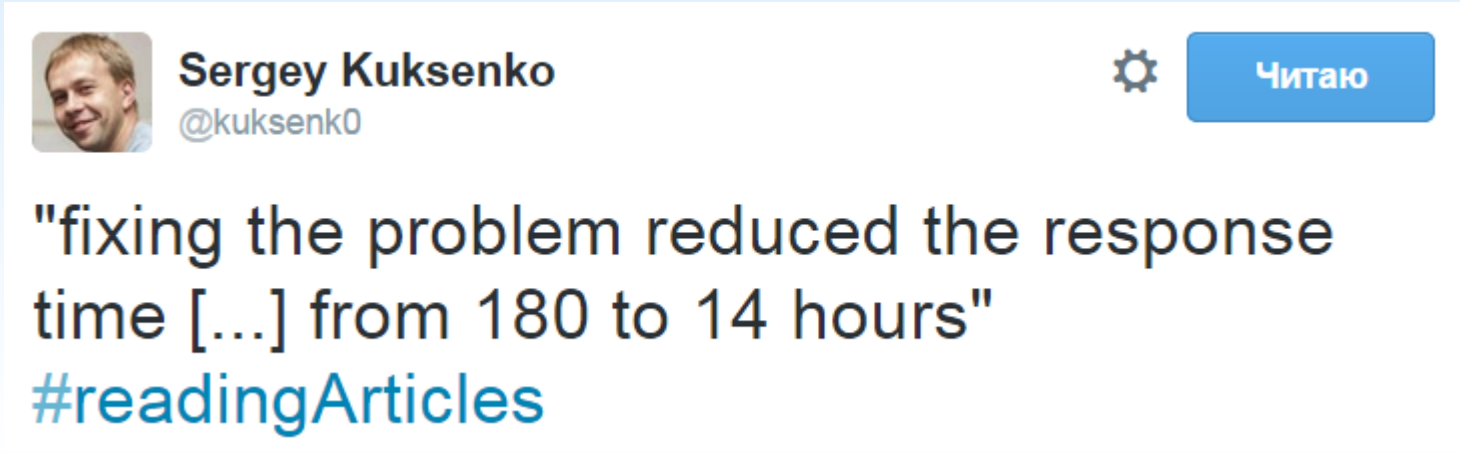


Типичный цикл разработки



Критерии оптимизации

Мы будем обсуждать операции по 0.05..5+ сек



Зачем ~~жениться~~ профилировать?

- Надо, чтобы работало гораздо быстрее
- Сделано должно быть ещё вчера
- Ты попал даже если там всего 1 твоя строка кода

Что ждём от профайлера?

- Нужно знать какие методы тормозили
  - MyDAO.loadObject
  - SocketOutputStream.write
- Нужно знать какие модули тормозили
  - Шапка, таблица с результатами

Что нужно получать помимо длительности?

- Нужно знать что и когда происходило
  - Имя нажатой кнопки, URL
- Нужно знать контекст
  - customer ID / order ID / user ID, SQL запросы, параметры к ним и так далее
- Нужно уметь заглядывать в прошлое
  - «У нас вчера было лучше/хуже»

Так ведь есть же профайлеры?

- Нужно чтобы профайлер был на каждом сервере
  - DEV/QA/PROD
- Профайлер может потребоваться каждому разработчику
  - Тут и на лицензиях разориться недолго
- Профайлер должен понимать суть профилируемого
  - Иначе получим «всё время потрачено в HashMap.get»

Интеграция профайлера и профилируемого кода

Код должен быть написан с оглядкой на профайлер

```
class UserServlet {
  private String userName;
  void doPost() {
    String userId = ...;
```

Как значения **userName** и **userId** попадут в результаты?

Интеграция профайлера и профилируемого кода

- Нельзя просто взять и поправить старые версии кода
  - А профилировать всё равно нужно
- Пока не получишь результат, непонятно где и какие значения нужно захватывать
  - Цикл «поправили->собрали->выдали» играет против нас

Интеграция профилируемого кода и профайлера

- Передаём из JMeter специальный HTTP header
- Ловим его в профайлере
- Анализ результатов нагрузочного теста сильно упрощается

Есть профайлеры в java мире!

Измеряем время ответа на клиенте

- JavaScript наше измерительное всё
  - G/Y analytics, boomerang, W3C Navigation timing API, W3C Mutation Observer API
- HTTP
  - access.log

```
CLIENT     DATE       TIME     METHOD      URL CODE SIZE DURATION
10.116.0.9 2011-06-14 17:28:24 GET /logout.jsp 302  279  0.022
```

Измеряем время ответа из самого приложения

Немного perf4j.log и у нас лог с замерами

```
StopWatch watch = new LoggingStopWatch("login");
runnable.run();
watch.stop(); // how much watch?
```

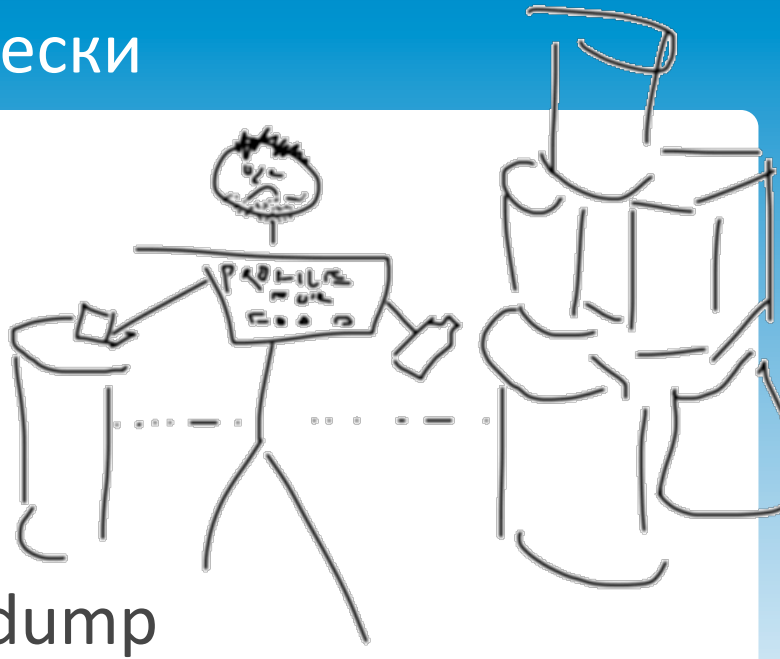
В логе будет как-то так:

```
start[1415926535898] time[42] tag[login]
```

Измеряем время ответа стохастически

Sampling: poor man's profiler

```
watch -n 0.1 kill -3 pid
```



- Каждые 0.1сек снимаем thread dump
- И смотрим чем чаще всего занимаются потоки

Измеряем время ответа стохастически

Смотрим ворох thread dump'ов

```
"[ACTIVE] ExecuteThread: '0' ... runnable ...
  at java.util.zip.ZipFile.getEntry(Native Method)
  at java.util.zip.ZipFile.getEntry
  - locked <0x789162b8> (a java.util.zip.ZipFile)
  at weblogic.utils.classloaders.ZipClassFinder.getSource
  at weblogic.utils.classloaders.JarClassFinder.getSource
  at weblogic.utils.classloaders.MultiClassFinder.getSource
```

И понимаем, что у нас safepoint bias

Safepoint

- Нельзя просто остановить поток в произвольном месте
  - Чтобы узнать stacktrace нужно-таки остановить
  - В вершине стека может оказаться случайный метод
- Тем не менее, снимать thread dump'ы с production раз в 5 минут весьма полезно

Измеряем время ответа на короткой ноге с JVM

honest-profiler

- Решает проблему safepoint bias
- Использует недокументированное API: AsyncGetCallTrace
- Пока не готово к production ☹

Может, купить какой-нибудь профайлер?

Клиент не всегда согласен устанавливать сторонние инструменты

- Прошлый отрицательный опыт
- Цена вопроса
- Чрезмерная общность
- Сложная адаптация кода к внешнему профайлеру

А что если создать профайлер для себя?

Создать профайлер совсем несложно

- Java byte code един для всех платформ
- Добавляем methodEnter/methodExit в нужные методы и дело в шляпе!
  - Instrumentation подход очень хорош для 10мс+ методов

MethodEnter/methodExit

Варианты добавить профилирующие вызовы:

- Вручную править исходный код
- Javassist
- Aspect oriented programming
- ASM Objectweb + Instrumentation API

Пишем профайлер за 4 шага

Шаг 1 – пишем логгер данных

Пишем логгер

```
class Profiler {
  public static State methodEnter(int id);
}
interface State {
  public void methodExit();
  public void logArg(int id, String value);
}
```

Шаг 1 – что нужно от profiler logger?

- Хорошая скорость потоковой записи
- Компактность данных
- По возможности, random-access
- Возможность экспорта данных за нужный интервал времени

Шаг 1 – где взять логгер?

- Взять готовый
  - Chronicle-Queue / Chronicle
- Создать самим
  - `DataOutputStream(new GZIPOutputStream(…))`

Шаг 2 – добавляем вызовы логгера в код

Нужно примерно следующее:

```
class MyDAO extends NoSQL {
  Object loadObject() {
    State s = Profiler.methodEnter(42);
    try {
      // прежний код метода loadObject
    } finally {
      s.methodExit(); ...
```

Шаг 2 – добавляем вызовы логгера в код

Байткод совсем не страшен: ASM и вперёд

```
class ProfilingAdapter extends AdviceAdapter
{
  protected void onMethodEnter()
  {
    добавляем вызов Profiler.methodEnter();
  }
}
```

Шаг 3 – активируем инструментацию

```
META-INF/MANIFEST.MF
```

- `Agent-Class: com.acme.Profiler`
- `Can-Redefine-Classes: true`
- `Boot-Class-Path: profiler-boot.jar`

И добавляем параметр запуска JVM

```
-javaagent:path-to-profiler.jar
```

Шаг 4 – Пишем UI для просмотра данных

Web приложение для просмотра данных очень удобно:

- Не требуется ставить клиент
- Без проблем работает отдельно от самого приложения
- Немного вспоминается «как рисовать сову», но SlickGrid и jQuery творят чудеса

Подводные камни

Подводные камни

- Web browser
- Contention / Concurrency
- JVM
- OS / Hardware

Подводные камни - Browser

JavaScript работает весьма быстро, но

- Выделить более 1Гб данных в javascript сложно
  - Нужно представлять данные компактно (ваш Кэп)
- Глубина иерархии массивов ограничена ([1,2,[3,4],…])
  - Нужно разбивать сильно вложенные структуры
- Более 100 уровней на экран просто не поместится
  - Нужно показывать только важные ветви дерева

Подводные камни - Contention

Profiler вызывается из разных потоков (surprise!), а значит:

- Нужно следить за `synchronized/volatile`
  - `ArrayBlockingQueue` (ABQ) вполне хорошо
- Стоит накопить пачку данных и уже её засылать в ABQ
- Не забываем про Single Writer Principle и писателю логов живётся хорошо

Подводные камни – JVM

- В java 1.4 нет Instrumentation API
- В java 1.5 Instrumentation не поддерживает native методы
- В java 1.6 уже можно инструментировать native методы
- В java 1.7 появился Split Verifier
  - Добавлять try-catch блоки стало сложнее
  - Обновляем ASM и следим за EXPAND_FRAMES

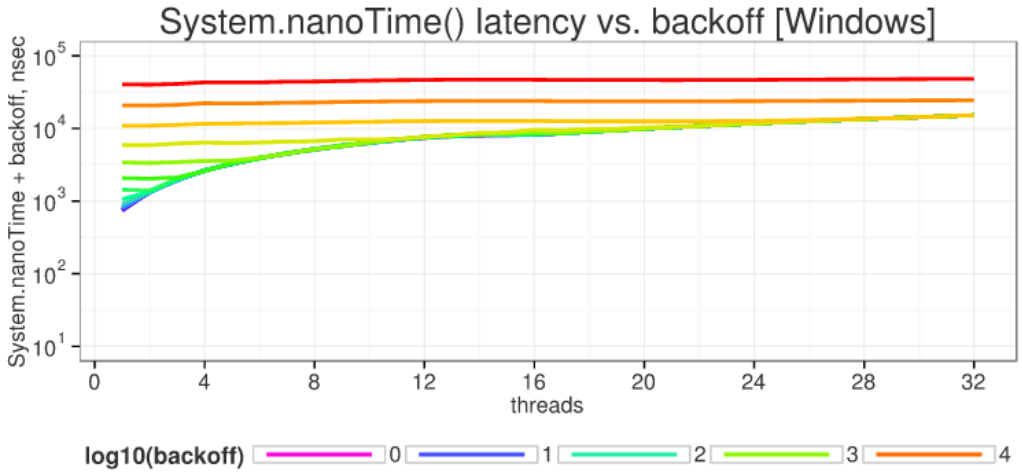
Подводные камни – OS / Hardware

Не пытайтесь покинуть Омск

Подводные камни – OS / Hardware

Не пытайтесь использовать System.nanoTime и currentTimeMillis

- Эти методы очень плохо работают под нагрузкой: точность/скорость проседает в 1000 раз
- Nanotrusting Nanotime:



Есть ли жизнь без currentTimeMillis?

Запускаем поток, и пусть он следит за временем:

```
class IAmTimer extends Thread {
  public static volatile long now;
  public run() {
    while(true) {
      now = System.currentTimeMillis();
      sleep(1); // На Solaris спит по 10+мс
```

Есть ли жизнь без currentTimeMillis?

- Если sleep(1) длится гораздо больше 1мс, то мы что-то подозреваем:

  GC пауза, Swap-in, safepoint, гранулярность таймера

- Такие события полезно записывать и отображать
  - jHiccup работает как раз таким образом

Складываем вместе и смотрим

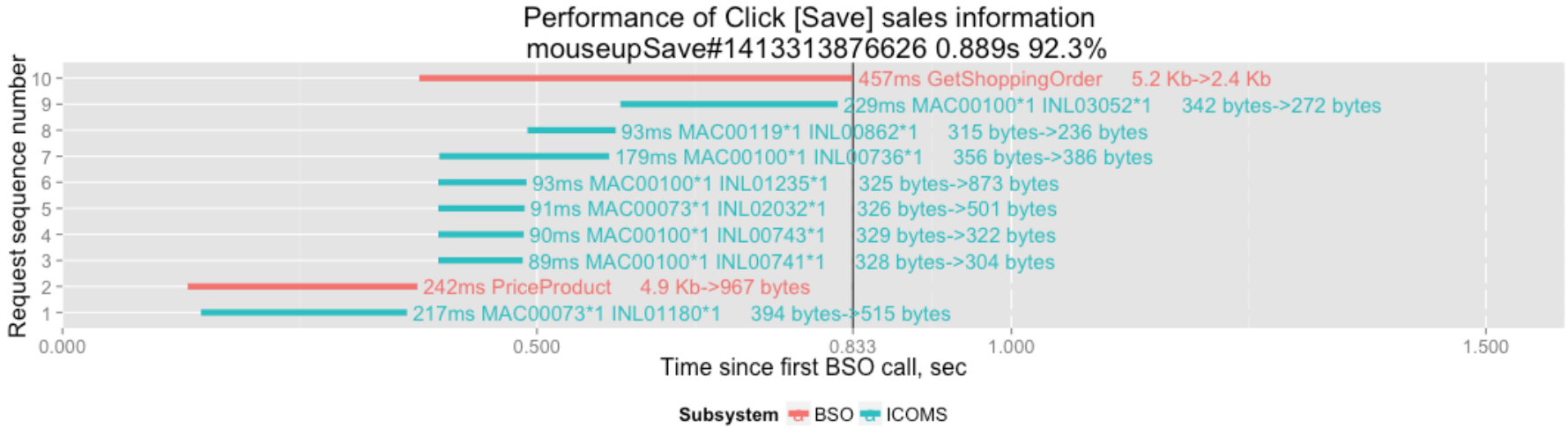
Что получилось у нас

- Мы используем профайлер в production
- Менее 1% увеличение времён отклика
- 500MiB gzip логов в час с одного узла (4-8 CPU core)
- Одного log writer’a на JVM пока хватает
- Профайлер активно используется разработчиками

Perf4j + ggplot

С помощью ggplot можно построить хорошие диаграммы из простых perf4j / access.log данных

```
ggplot(logs) + geom_segment(aes(x=start, xend=end), size=2)
```



Свой профайлер всегда доступен

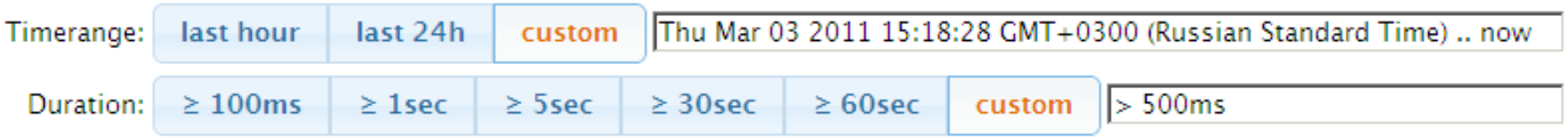
Заходим на /profiler и видим было ли приложению «плохо»

Date	Duration	Suspension	Calls	Title Search: Type filter string
13:16:34.257	1.260s	0.000s	27'090	JMS: **WfSubscriberSysJMSBean** wf.process: 912
13:16:34.007	0.240s	0.000s	4'499	JMS quartz job Scheduled workflow event – orc
13:16:00.058	0.210s	0.000s	4'794	Class quartz job Queue Processor (JMSQueueAc
13:15:00.559	0.620s	0.000s	42'960	GET /report/reports.jsp?report=603065407701
13:15:00.469	0.940s	0.000s	119'173	GET /report/reports.jsp?report=603065407701

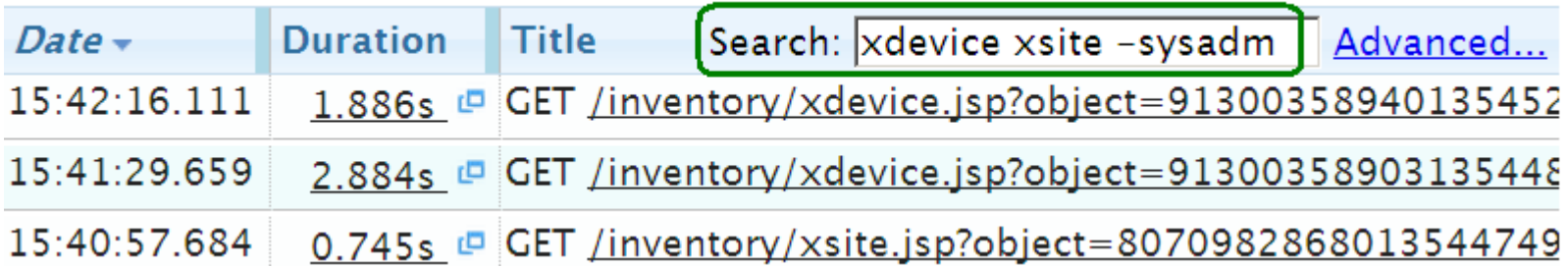
Поиск вызова

Поиск нужного вызова:

- По диапазону дат, длительности

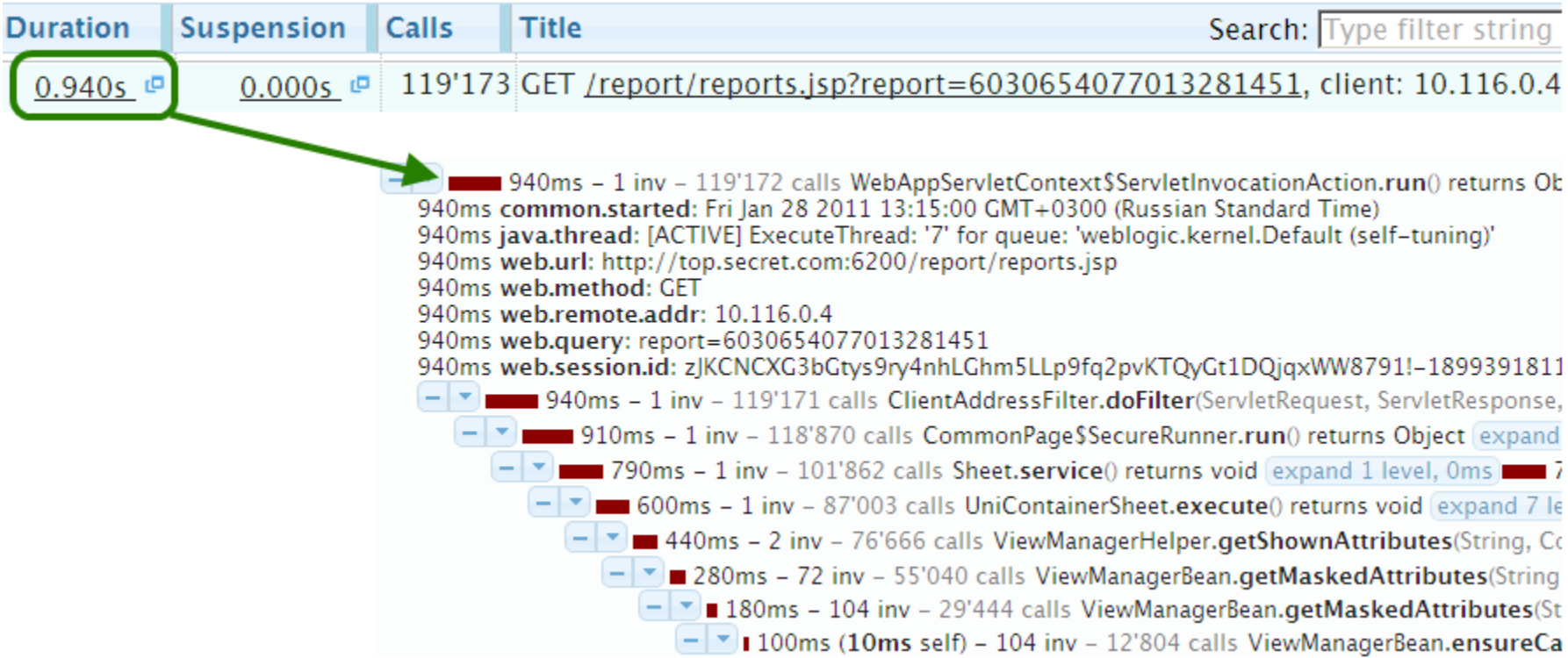


- По подстроке



Разные страницы – разные деревья вызовов

При instrumentation подходе можно рассмотреть каждое дерево независимо



Дерево вызовов курильщика

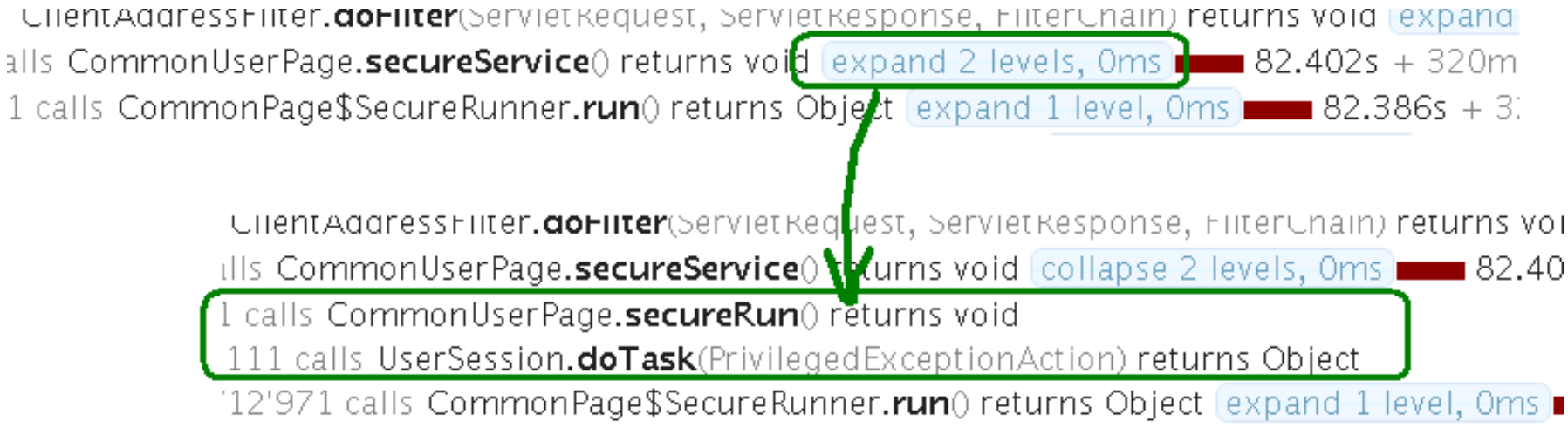
В «обычном» профайлере у дерева нет конца и края:

Sample Count	Percentage	Stack Trace
11 402	67,54%	~ UNCLASSIFIABLE ~.()
1 592	9,43%	weblogic.servlet.internal.FilterChainImpl.doFilter(ServletReq
167	0,99%	com.netcracker.platform.core.filters.clientaddress.Client/
167	0,99%	weblogic.servlet.internal.FilterChainImpl.doFilter(Serv
167	0,99%	com.netcracker.platform.core.filters.security.HttpS
167	0,99%	com.netcracker.platform.core.filters.security.Ht
167	0,99%	weblogic.servlet.internal.FilterChainImpl.dol
167	0,99%	com.netcracker.presentation.process.Se

В случае JBoss AS доходило до 1’400 глубины стека ☺

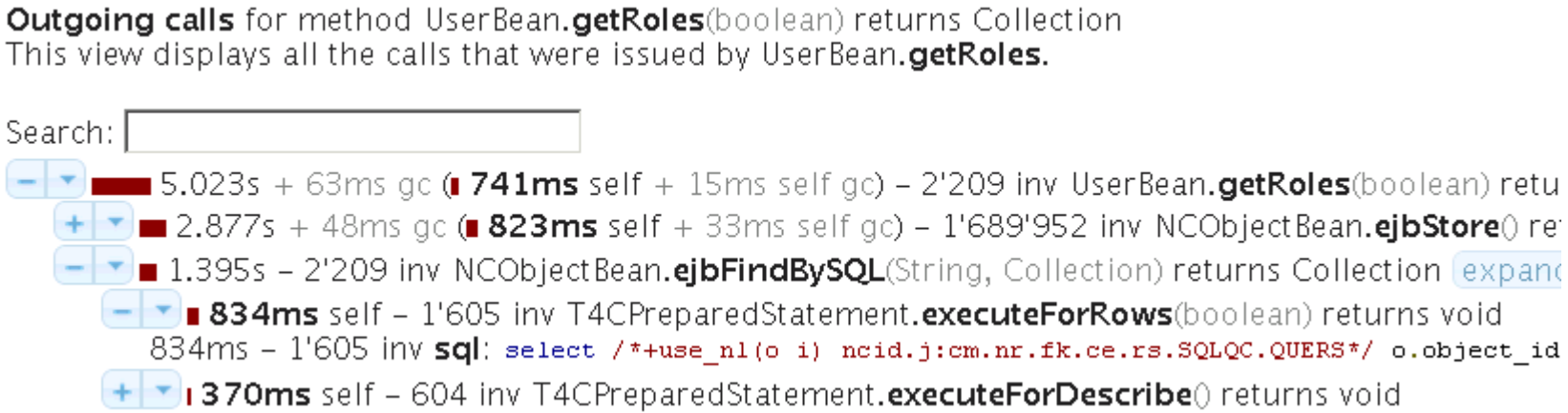
Дерево вызовов здорового человека

Мы скрываем незначимые уровни и всё помещается на экран-два-три:



Легко найти свой код

Каждый разработчик может в 1 click найти свой код



Видно:

- Общее количество вызовов getRoles
- Каждый getRoles всегда вызывает findBySQL
- Из 5-и секунд getRoles 2.9 сек тратилось на ejbStore

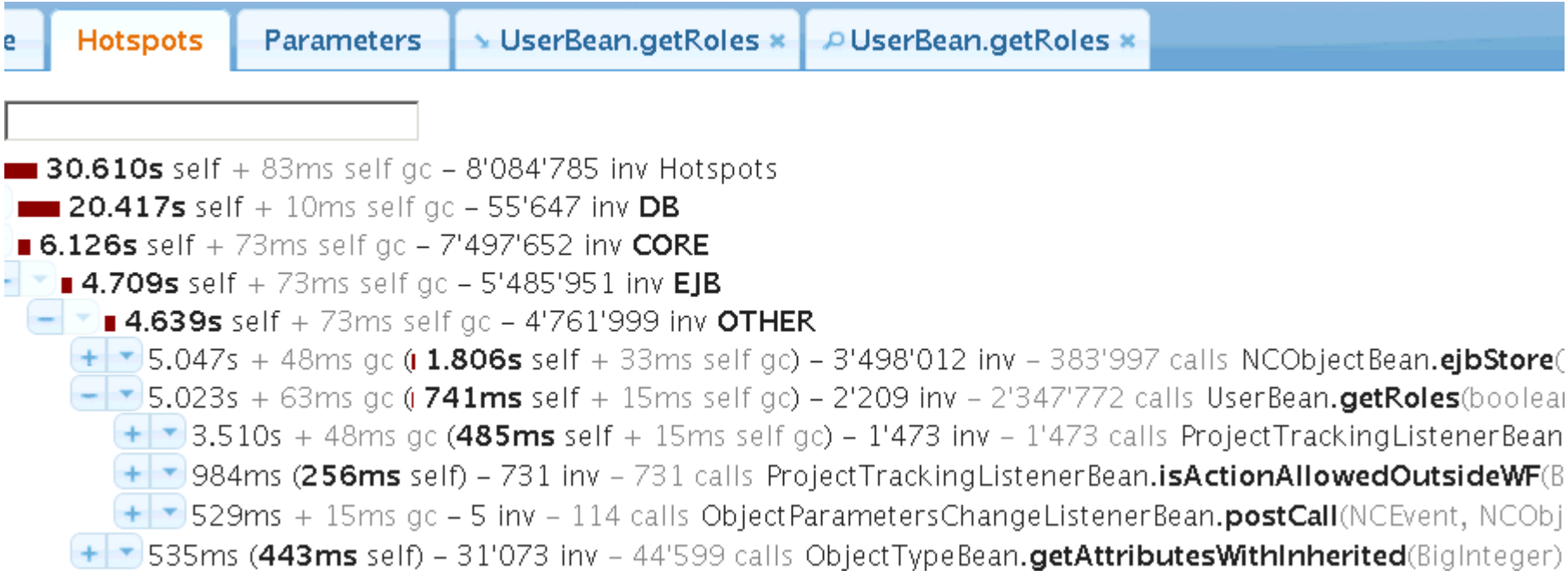
Черновик экрана hotspots

Первая версия окна hotspots: просто время, проведённое в методе

Self Time	Self Time	Invocations	Cummulative Time	Hotspot
	357ms	17	357ms	PreparedStatement.**execute**() returns boolean
	337ms	160	337ms	PreparedStatement.**executeQuery**() returns ResultSet
	320ms	6	320ms	PreparedStatement.**executeUpdate**() returns int
	33ms	1	1.112s	ProcessManagerBean.**execute**(BigInteger) returns void
	15ms	1	286ms	SecurityBulkManager.**setGrantsForAll**(BigInteger) retu
	14ms	1	843ms	InterprocessOperations.**clearDetailedGrants**(BigInteg
	10ms	3	10ms	WfJMSSender.**getRootElement**(String) returns Element

Hotspots v2

Попробовали и поняли, что нужно показывать источник вызовов



Поиск источника вызовов

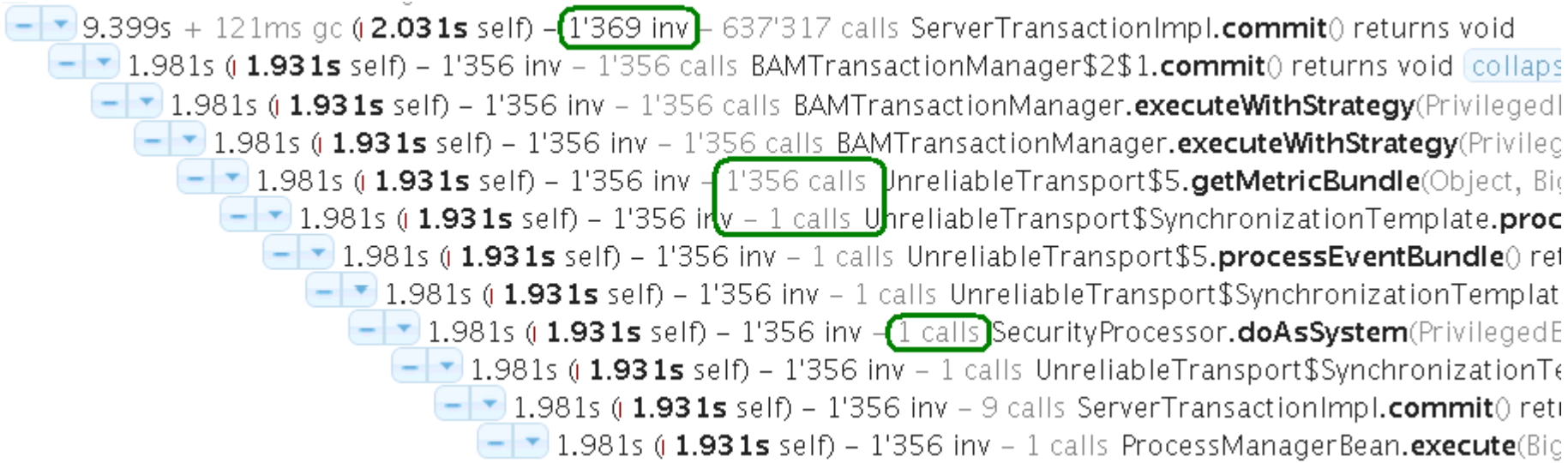
Немного развернём и уже понятна причина

Видно:

- Из ProjectTrackingListener’а было 1473+731 вызов getRoles
- Ещё 5 из ObjectParamersChangeListener’а

Циклы наше всё

В этом же hotspot view видно на каком уровне появляется цикл



Видно, что проблема в количестве вызовов getMetricBundle

Отчёты для PM

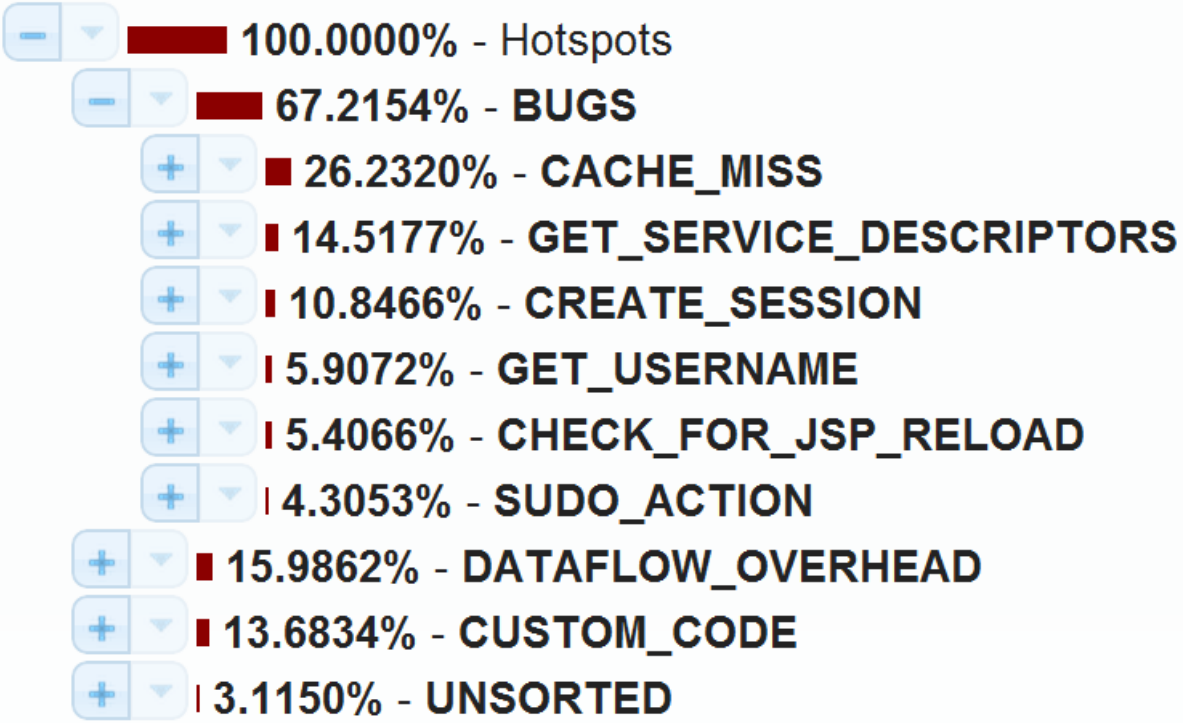
Дерево java методов это круто, но в конце концов нужно разделять по видимым клиенту результатам

Примеры:

- Шапка – 1 сек, основная таблица – 2 сек, copyright – 1 сек
- Проблема А – 2сек, проблема Б – 1 сек, остальное – 1 сек
- Debug логи – 4 сек, всё остальное – 0 сек

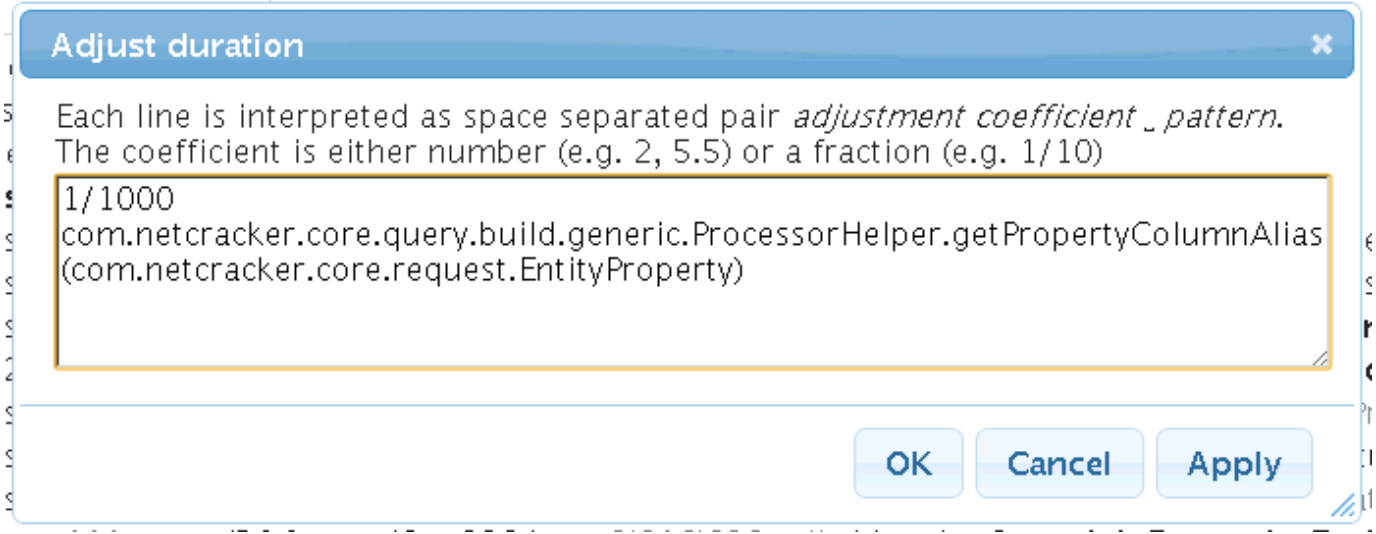
Отчёты для PM

Если сделать логическую разбивку, то сразу понятно что стоит исправлять сразу, а что может подождать:



«Известные проблемы»

- В одном дереве часто встречается несколько проблем сразу
- В каждой ветке top 1 проблема светится больше всего
- Выход – берём и немного правим уже собранные данные



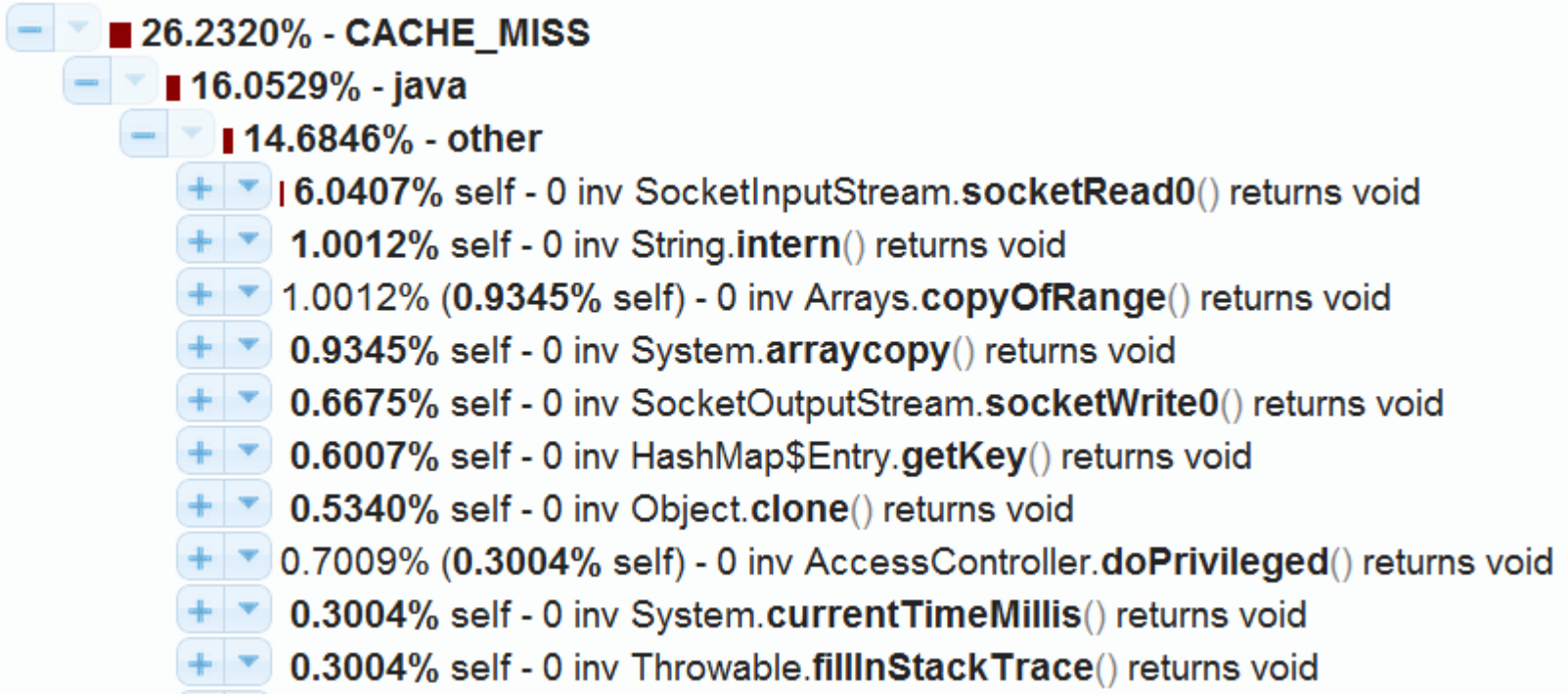
Что делать с Map.get?

Хочется смотреть и быстрые методы, а инструментировать слишком накладно

- Попытка записать каждое обращение к .get обречена на провал
- Если профайлер занимает 90% времени, то результат имеет небольшой смысл
- Особенно для production

Что делать с Map.get?

Во многих случаях 10-100ms проблемы можно анализировать по простым thread dump’ам



Открытые вопросы

- Профилирование памяти
  - java 1.7u40: Java Mission Control
  - java 1.6u26: ThreadMXBean.getThreadAllocatedBytes
- JITted/native code
  - Solaris Studio Performance Analyzer

Выводы

- Создать профайлер это просто
- Сделать UI чуть сложнее
- Профайлер без UI – время на ветер

Вопросы?

Список вызовов

Список вызовов: http://server:port/profiler

Date	Duration	Suspension	Calls	Transactions	Title Search: xdevice
15:42:16.111	1.886s	0.010s	1'008'187	2	GET /inventory/xdevice.js
15:41:29.659	2.884s	0.000s	291'397	2	GET /inventory/xdevice.js

- Duration: длительность вызова
- CPU time: время, потраченное на CPU java потоком
- Suspension: длительность торможения сервера (gc, swap, …)
- Calls: количество вызовов java методов в дереве
- Transactions: количество транзакций с участием JDBC
- Title: описание вызова: его тип и основные признаки
- Disk IO: количество записанных/прочитанных с диска данных
- Network IO: количество принятых/переданных по сети данных