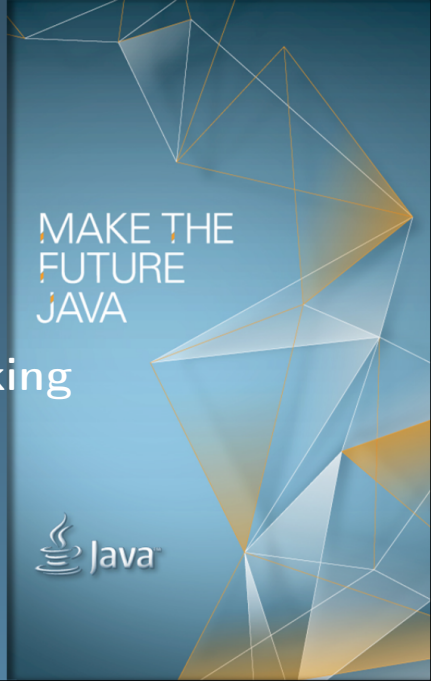
ORACLE



(The Art of) (Java) Benchmarking

Gentle Introduction in JMH

Алексей Шипилёв
aleksey.shipilev@oracle.com, @shipilev

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Введение

Введение: стандартный дисклеймер

1. Computer Science → Software Engineering
   - Строим приложения по функциональным требованиям
   - В большой степени абстрактно, в «идеальном мире»
   - Рассуждения при помощи формальных методов

2. Performance Engineering
   - «Real world strikes back!»
   - Исследуем взаимодействия софта с железом на типичных данных
   - Эффективно предсказывается уже мало что
   - Рассуждения при помощи формальных методов

Теория

Теория: бенчмарки

1. «Программа для измерения производительности»
   - Жило-было обычное приложение, добавили измерение времени – бац – уже бенчмарк
   - Каждый запуск бенчмарка – эксперимент

2. Типичные требования к эксперименту
   - Результат запуска – значение некоторой метрики
   - Должен быть объективным («test the right thing»)
   - Должен быть надёжным (воспроизводимым)

Теория: классификация бенчмарок

1. Реальные приложения
   - Запускаем руками, совершаем действия руками
   - Меряем секундомером, вольтметром, осциллографом
2. Автоматические сценарии приложений
   - Зафиксировали какой-нибудь сценарий
   - Автоматически измерили время, мощность, трафик
3. Синтетические (макро) бенчмарки
   - Написали приложение, похожее на типичное, эталонное
   - Автоматически измерили
4. Микро/нано-бенчмарки
   - Написали отдельную, маленькую часть
   - Выбросили всё остальное

Теория: зачем?

Высечь в граните

Эксперимент требует понимания объекта исследований.

Теория: зачем?

Высечь в граните

Эксперимент требует понимания объекта исследований.

1. Чем у́же область исследования, тем больше нужно знать
   - надёжная изоляция от посторонних эффектов
   - надёжный контроль, «проверки на дурака»

Теория: зачем?

Высечь в граните

Эксперимент требует понимания объекта исследований.

1. Чем у́же область исследования, тем больше нужно знать
   - надёжная изоляция от посторонних эффектов
   - надёжный контроль, «проверки на дурака»
2. Бенчмаркинг – это война против оптимизаций

Теория: зачем?

Высечь в граните

Эксперимент требует понимания объекта исследований.

1. Чем у́же область исследования, тем больше нужно знать
   - надёжная изоляция от посторонних эффектов
   - надёжный контроль, «проверки на дурака»
2. Бенчмаркинг – это война против оптимизаций
3. Нужно знать всё, вплоть до микроархитектуры

Теория: главный вопрос

Теория: главный вопрос

Всегда, всегда задавайте себе его!

Почему мой бенчмарк не может работать быстрее?

Ответ определяет качество эксперимента:

1. В какие ограничения упёрлись?
2. Работает та часть кода, которую мы «исследуем»?
3. Что сделать, чтобы исправить бенчмарк?

Практика

Практика: фреймворки

- Автоматизация
  - Килотонна инфраструктуры на грамм измеряемого кода
  - Фреймворк, который эту килотонну делает однажды?
  - a priori не известно, на какие эффекты мы наступим $\Rightarrow$ нужно учитывать все возможные
- Вечная борьба между удобством интерфейсов, их скоростью, и надёжностью:
  - звать бенчмарки через Reflection? (oh wow)
  - запускать любые JUnit-тесты? (oh wow x2)
  - (ещё какое-нибудь сумасшествие)

Практика: JMH

У нас тоже есть очень хороший харнесс:
`http://openjdk.java.net/projects/code-tools/jmh/`

- JMH is Serious Business:
  - Он учитывает тонну VM-ных эффектов
  - Мы его периодически допиливаем, когда меняется VM
  - Мы его периодически фиксим, как растёт наша экспертиза
  - Всякий внешний бенч валидируется переписыванием под JMH

Практика: JMH

У нас тоже есть очень хороший харнесс:
`http://openjdk.java.net/projects/code-tools/jmh/`

- JMH is Serious Business:
  - Он учитывает тонну VM-ных эффектов
  - Мы его периодически допиливаем, когда меняется VM
  - Мы его периодически фиксим, как растёт наша экспертиза
  - Всякий внешний бенч валидируется переписыванием под JMH
- Мы вынули столько неочевидных граблей из JMH, что не верим ни одному известному харнессу.

Hello World: демо

Базовый Hello World:

`http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_01_HelloWorld.java`

Warmup: замолвите слово

Запомните

Мы имеем дело с динамическими системами.
«Прогрев» – это выжидание характерного времени переходного процесса.

- У нас полно переходных процессов.

Warmup: замолвите слово

Запомните

Мы имеем дело с динамическими системами.
«Прогрев» – это выжидание характерного времени переходного процесса.

- У нас полно переходных процессов.
- Компиляция кода – не единственный переходный процесс!

Warmup: замолвите слово

Запомните

Мы имеем дело с динамическими системами.
«Прогрев» – это выжидание характерного времени переходного процесса.

- У нас полно переходных процессов.
- Компиляция кода – не единственный переходный процесс!
- «Мудрость»: «Следите за PrintCompilation». WTF?

States & Fixtures: многопоточные бенчмарки

1. Мало кто умеет писать многопоточные бенчмарки
2. Требуется изрядная ловкость, чтобы:
   - корректно инициализировать состояние (+ нужным потоком)
   - корректно состояние опубликовать (+ только нужным потокам)
   - корректно синхронизировать потоки (+ только те, что работают с состоянием)
3. И уж тем более сложно разделить во харнессе состояния
   - shared состояние можно сэмулировать static-полями
   - non-shared состояние можно сэмулировать instance-полями
   - а состояние, расшаренное только отдельной группой потоков?

States & Fixtures: демо

Про State-объекты:

http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_03_States.java

States & Fixtures: демо

Про Fixtures:

http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_05_StateFixtures.java

Dead Code Elimination: проблема

- Умные компиляторы давно умеют делать dead-code elimination:
  - результат не используется, нет side effect'ов?
  - никто не заметит, если мы удалим код?
- Кажется, что с этим легко бороться, используем результат!

Dead Code Elimination: проблема

- Умные компиляторы давно умеют делать dead-code elimination:
  - результат не используется, нет side effect'ов?
  - никто не заметит, если мы удалим код?
- Кажется, что с этим легко бороться, используем результат!
  - Q: Накапливать в аккумуляторе?
    A: DCE «протягивается» даже через очень сложные выражения.

Dead Code Elimination: проблема

- Умные компиляторы давно умеют делать dead-code elimination:
  - результат не используется, нет side effect'ов?
  - никто не заметит, если мы удалим код?
- Кажется, что с этим легко бороться, используем результат!
  - Q: Накапливать в аккумуляторе?
    A: DCE «протягивается» даже через очень сложные выражения.
  - Q: Сохранить результат в поле?
    A: False sharing, card marks...

Dead Code Elimination: проблема

- Умные компиляторы давно умеют делать dead-code elimination:
  - результат не используется, нет side effect'ов?
  - никто не заметит, если мы удалим код?
- Кажется, что с этим легко бороться, используем результат!
  - Q: Накапливать в аккумуляторе?
    A: DCE «протягивается» даже через очень сложные выражения.
  - Q: Сохранить результат в поле?
    A: False sharing, card marks...
  - Q: Сравнить с невероятным результатом и кинуть exception?
    A: Попробуй выбрать такое значение, чтобы оно не было предсказано.

Dead Code Elimination: демо

Про DCE:

http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_08_DeadCode.java

CSE, Folding: проблема

Умные компиляторы хорошо предвычисляют:

- вычисление над константами = константа
- вычисление над локалом = константа
- вычисление над полем = константа для цикла

С этим можно бороться:

- храним источники данных в полях
- делаем volatile read между чтениями источника

CSE, Folding: демо

Про constant fold:

`http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_10_ConstantFold.java`

Loops: Loop Unrolling

Мега-оптимизация!

После раскрутки циклов:

- склеиваются общие части итераций
- что-то даже hoist'ится наружу
- удаляются разного рода проверки, типа array bounds check

Если одна итерация реально стоит X ns, то
после раскрутки цикла она «эффективно» стоит
$\alpha\ X$, где $\alpha \in [0; 1]$.

Loops: демо

Про loop unrolling:

`http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_11_Loops.java`

Forking: смешать, но не взбалтывать

Многие оптимизации опираются на профиль.

Почти всегда тесты смешивать нельзя:

- смешаются профили, и привет
- обычно, первый тест ещё и шустрее работает

Почти все вменяемые тесты делаются в отдельных JVM.

Forking: демо

Про forking:

`http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_12_Forking.java`

Run-to-run Variance: бич сложных систем

Как и любой сложный ящик, JVM «шумит».

От запуска к запуску производительность может плавать[1]:

- шум в профилях, в порядке компиляции
- шум в трединге
- случайность в JVM/JDK, и в самом приложении

Пока не доказано обратное,
тесты нужно запускать в нескольких JVM подряд.

[1] http://dl.acm.org/citation.cfm?id=1297033

Run-to-run Variance: демо

Про run-to-run variance:

`http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_13_RunToRun.java`

Edge Effects: особый, уличный косяк

Warmup уводит нас в steady state.
Измеряем ли мы только в steady state?

Пример:

- реакция шедулинга не мгновенна
- пока стартуют все потоки, некоторые уже начнут работать
- (и это существенно исказит результат)

На больших машинах edge effect'ы от потоков
сильно искажают результаты!

Edge Effects: демо

Про sync iterations:

http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_17_SyncIterations.java

Edge Effects: демо

Про холодные потоки:

http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_07_FixtureLevelInvocation.java

Выводы

Выводы: Benchmarking is Serious Business

Огромное поле для ошибок.

- Написание тестов требует экспертизы
- Написание фреймворков требует ещё большей экспертизы
- Не верьте красивым репортам, верьте логичным результатам

Выводы: инструменты

1. Мозг
   - Плагин «данунеможетбыть» для перепроверок фактов
   - Плагин «щапридумаем» для построения гипотез и экспериментов
   - Плагин «чётоянепонял» для проверки консистентности гипотез
   - Плагин «ядурак» для лёгкого отвержения ложных гипотез
2. Руки
   - Прямые, для постановки аккуратных экспериментов
   - Сильные, для обработки тонн экспериментальных данных
3. Язык, уши, глаза и прочее I/O
   - Для обмена результатами и peer review
   - Для доступа к предыдущим экспериментам

Выводы: прочие инструменты

1. Фреймворки
   - JMH: `http://openjdk.java.net/projects/code-tools/jmh/`
2. Профилировщики
   - VisualVM, JRockit Mission Control, Oracle Studio Performance Analyzer
   - top, vmstat, mpstat, iostat, dtrace, strace
3. Дизассемблеры
   - `-XX:+PrintAssembly`
   - `https://wikis.sun.com/display/HotSpotInternals/PrintAssembly`