JMH

В первую очередь остановимся на основной части наших замеров — использовании [JMH](http://openjdk.java.net/projects/code-tools/jmh/). Java Microbenchmark Harness — набор библиотек для тестирования производительности небольших функций (то есть тех, где пауза GC увеличивает время работы в разы).

Перед запуском теста JMH перекомпилирует код, так как:

1. Для уменьшения погрешности вычисления времени работы функции необходимо запустить её N раз, подсчитать общее время работы, а потом поделить его на N.
2. Для этого требуется обернуть запуск в виде цикла и вызова необходимого метода. Однако в этом случае на время работы функции повлияет сам цикл, а также сам вызов замеряемой функции. А потому вместо цикла будет вставлен непосредственно код вызова функции, без reflection или генерации методов в runtime.

После переделки байткода тестирование можно запустить командой вида java -jar benchmarks.jar, так как все необходимые компоненты уже будут запакованы в один jar файл.

JMH Gradle Plugin

Как понятно из описания выше, для тестирования производительности кода недостаточно просто добавить необходимые библиотеки в classpath и запустить тесты в стиле JUnit. А потому, если мы хотим делать дело, а не разбираться в особенности написания билд скриптов, нам не обойтись без плагина к maven/gradle. Для новых проектов преимущество остается за gradle, потому выбираем его.

Для JMH есть полуофициальный плагин для gradle — [jmh-gradle-plugin](https://github.com/melix/jmh-gradle-plugin). Добавляем его в проект:

buildscript {

repositories {

mavenCentral()

maven {

url "https://plugins.gradle.org/m2/"

}

}

dependencies {

classpath "me.champeau.gradle:jmh-gradle-plugin:$jmh\_gradle\_plugin\_version"

}

}

apply plugin: "me.champeau.gradle.jmh"

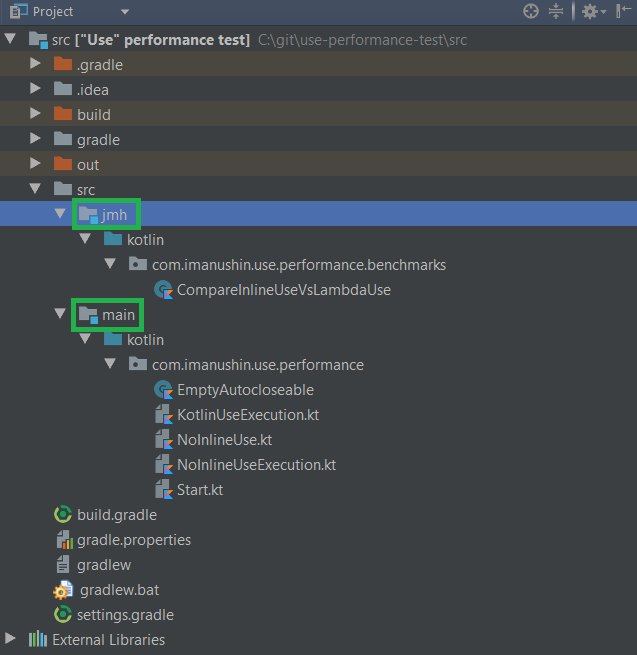
Плагин автоматом создаст новый source set (это "набор файлов и ресурсов, которые должны компилироваться и запускаться вместе", прочитать можно или [статью на хабре](https://habrahabr.ru/post/107085/) за авторством [svartalfar](https://habr.com/en/users/svartalfar/), или же [в официальной документации gradle](https://docs.gradle.org/current/userguide/artifact_dependencies_tutorial.html)). jmh source set автоматически ссылается на main, то есть получаем короткий алгоритм работы:

1. Код, который мы будем изменять, пишем в стандартном main source set, там же, где и всегда.
2. Код с настройкой и прогревом тестов пишем в отдельном source set. Именно его byte code и будет перезаписываться, сюда плагин добавит необходимые зависимости, в которых есть определения аннотация и тд.

Получаем следующую иерархию каталогов:

* src
  + jmh / kotlin/ <Имя java пакета> / <код, запускающий тесты (и аннотированный JMH аттрибутами)>
  + main / kotlin / <Имя java пакета> / <код для тестирования>

Или как это выглядит в IntelliJ Idea:



В итоге, после настройки проекта, можно запускать тесты простым вызовом .\gradlew.bat jmh (или .\gradlew jmh для Linux, Mac, BSD)

С плагином есть пара интересных особенностей на Windows:

* JMH использует fork java процесса. В случае Windows это сделать так просто нельзя, а потом новый процесс просто запускается с тем же classpath. И весь список jar файлов передается через командную строку, размер которой ограничен. В итоге, если GRADLE\_USER\_HOME (папка, внутри которой лежит в том числе кеш gradle) находится в глубине файловой структуры, список jar файлов для fork становится настолько большим, что Windows отказывается запускать процесс с таким громадным число аргументов командной строки. Следовательно, **если JMH отказывается делать fork — просто переместите кеши Gradle в папку с коротким именем, т.е. запишите в environment variable *GRADLE\_USER\_HOME* что-то вроде c:\gradle**
* Иногда предыдущий процесс JMH делает lock на файле (возможно, это делает byte code rewrite). В итоге, повторная компиляция может не работать, так как файл с нашим benchmark открыт кем-то на запись. Чтобы исправить эту проблему необходимо просто остановить deamon процессы gradle (которые уже запущены, чтобы ускорить работу компилятора): .\gradlew.bat --stop
* Для чистоты экспериментов, лучше отказаться от инкрементальной сборки для наших тестов. Отсюда, перед тестированием всегда вызываем .\gradlew.bat clean

Тестирование

В качестве примера я возьму вопрос (ранее заданный на [kotlin discussions](https://discuss.kotlinlang.org/t/why-autocloseable-use-method-is-inlined/5407)), который мучал меня ранее — зачем в конструкции use используется inline метод?

О конструкции use

Итак, необходимо сделать два метода:

1. Первый будет просто использовать use, который поставляется в библиотеке kotlin
2. Второй будет использовать те же методы, однако без inline. В итоге на каждый вызов в куче будет создаваться объект с параметрами для лямбды.

Код с JMH аттрибутами, который будет запускать разные функции:

@BenchmarkMode(Mode.All) *// тестируем во всех режимах*

@Warmup(iterations = 10) *// число итераций для прогрева нашей функции*

@Measurement(iterations = 100, batchSize = 10) *// число проверочных итераций, причем в каждой из них будет по четыре вызова функции*

open class CompareInlineUseVsLambdaUse {

@Benchmark

fun inlineUse(blackhole: Blackhole) {

NoopAutoCloseable(blackhole).use {

blackhole.consume(1)

}

}

@Benchmark

fun lambdaUse(blackhole: Blackhole) {

NoopAutoCloseable(blackhole).useNoInline {

blackhole.consume(1)

}

}

}

Dead Code Elimination

Java Compiler & JIT довольно умные и имеют ряд оптимизаций, как в compile time, так и в runtime. Метод ниже, например, вполне может свернуться в одну строку (как для kotlin, так и для java):

fun sum() : Unit {

val a = 1

val b = 2

a + b;

}

И в итоге мы будем тестировать метод:

fun sum() : Unit {

3;

}

Однако результат ведь никак не используется, потому компиляторы (byte code + JIT) в итоге вообще выкинут метод, так как он в принципе не нужен.

Чтобы избежать этого, в JMH существует специальный класс "черная дыра" — Blackhole. В нем есть методы, которые с одной стороны не делают ничего, а с другой стороны — не дают JIT выкинуть ветку с результатом.

А для того, чтобы javac не пытался сложить-таки a и b в процессе компиляции, нам требуется определить объект state, в котором будут храниться наши значения. В итоге в самом тесте мы будем использовать уже подготовленный объект (то есть не тратим время на его создание и не даем компилятору применить оптимизации).

В итоге для грамотного тестирования нашей функции требуется её написать вот в таком виде:

fun sum(blackhole: Blackhole) : Unit {

val a = state.a *// компилятор не знает заранее значение a*

val b = state.b

val result = a + b;

blackhole.consume(result) *// JIT не может выкинуть сложение, так как результат кому-то все-таки нужен*

}

Здесь мы взяли a и b из некоторого state, что помешает компилятору сразу посчитать выражение. А результат мы отправили в черную дыру, что помешает JIT выкинуть последнюю часть функции.

Возвращаясь к моей функции:

1. Объект для вызова метода close я создам в самом тесте, так как практически всегда при вызове метода close у нас до этого создавался объект.
2. Внутри нашего метода придется вызывть функцию из blackhole, чтобы спровоцировать создание лямбды в куче (и не дать JIT выкинуть потенциально ненужный код).

Результат теста

Запустив ./gradle jmh, а потом подождав два часа, я получил следующие результаты работы на моем mac mini:

# Run complete. Total time: 01:51:54

Benchmark Mode Cnt Score Error Units

CompareInlineUseVsLambdaUse.inlineUse thrpt 1000 11689940,039 ± 21367,847 ops/s

CompareInlineUseVsLambdaUse.lambdaUse thrpt 1000 11561748,220 ± 44580,699 ops/s

CompareInlineUseVsLambdaUse.inlineUse avgt 1000 ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.lambdaUse avgt 1000 ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.inlineUse sample 21976631 ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p0.00 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p0.50 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p0.90 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p0.95 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p0.99 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p0.999 sample ≈ 10⁻⁵ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p0.9999 sample ≈ 10⁻⁵ s/op

CompareInlineUseVsLambdaUse.inlineUse:inlineUse·p1.00 sample 0,005 s/op

CompareInlineUseVsLambdaUse.lambdaUse sample 21772966 ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p0.00 sample ≈ 10⁻⁸ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p0.50 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p0.90 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p0.95 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p0.99 sample ≈ 10⁻⁷ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p0.999 sample ≈ 10⁻⁵ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p0.9999 sample ≈ 10⁻⁵ s/op

CompareInlineUseVsLambdaUse.lambdaUse:lambdaUse·p1.00 sample 0,010 s/op

CompareInlineUseVsLambdaUse.inlineUse ss 1000 ≈ 10⁻⁵ s/op

CompareInlineUseVsLambdaUse.lambdaUse ss 1000 ≈ 10⁻⁵ s/op

Benchmark result is saved to /Users/imanushin/git/use-performance-test/src/build/reports/jmh/results.txt

Или, если сократить таблицу:

Benchmark Mode Cnt Score Error Units

inlineUse thrpt 1000 11689940,039 ± 21367,847 ops/s

lambdaUse thrpt 1000 11561748,220 ± 44580,699 ops/s

inlineUse avgt 1000 ≈ 10⁻⁷ s/op

lambdaUse avgt 1000 ≈ 10⁻⁷ s/op

inlineUse sample 21976631 ≈ 10⁻⁷ s/op

lambdaUse sample 21772966 ≈ 10⁻⁷ s/op

inlineUse ss 1000 ≈ 10⁻⁵ s/op

lambdaUse ss 1000 ≈ 10⁻⁵ s/op

В результате есть две самые важные метрики:

1. Inline метод показал производительность 11,6 \* 10^6 ± 0,02 \* 10^6 операций в секунду.
2. Lambda-based метод показал производительность 11,5 \* 10^6 ± 0,04 \* 10^6 операций в секунду.
3. Inline метод в итоге работает быстрее и стабильнее по скорости. Возможно, увеличенная погрешность для *lambdaUse* связана с более активной работой с памятью.
4. Я таки был неправ на [том форуме](https://discuss.kotlinlang.org/t/why-autocloseable-use-method-is-inlined/5407) — в стандартной библиотеке kotlin лучше оставить текущую реализацию метода.