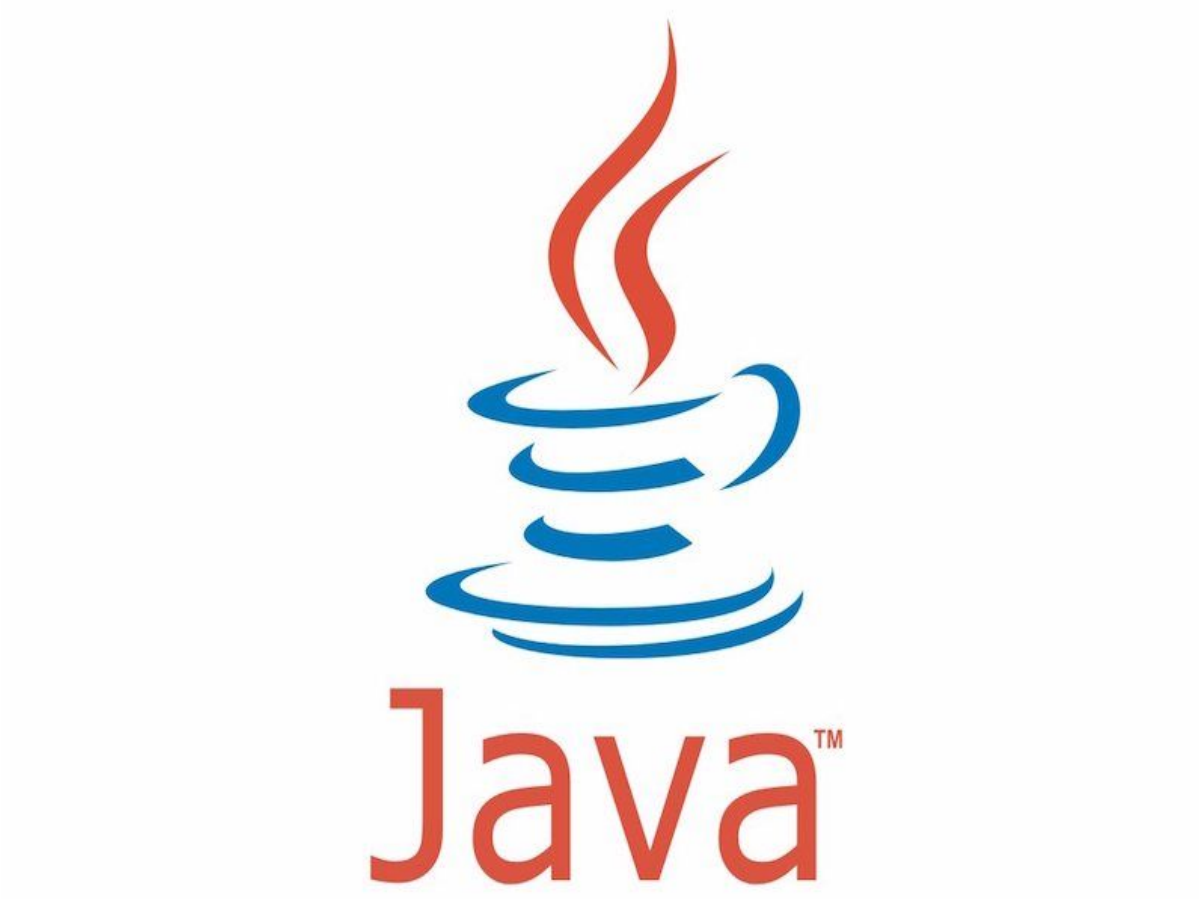
TP2

JMH et les micro-benchmarks

Cognard Cédric et Van-Elsue Paul



Sommaire

1. Génération et lancement du projet JMH

Pour créer un projet **JMH** la façon la plus simple est de passer par *Maven*, pour cela nous utilisons la ligne de commande suivante :

```
mvn archetype:generate -DinteractiveMode=false -DarchetypeGroupId=org.openjdk.jmh
-DarchetypeArtifactId=jmh-java-benchmark-archetype
-DgroupId=fr.polytechtours.javaperformance.tp2 -DartifactId=jmh -Dversion=1.0.0
```

Cela génère le projet qui contient donc tout ce qu'il faut pour lancer les tests **JMH**, il suffit ensuite seulement de coder les fonctions de benchmark.

2. Coder les fonctions de benchmark

Pour coder un benchmark, il suffit simplement d'ajouter l'annotation **@Benchmark** au dessus d'une méthode puis de choisir d'exécuter cette dernière dans l'IDE. On aura en sortie tout un rapport sur l'exécution de cette méthode. On peut ainsi comparer plusieurs tests.

De nombreuses annotations sont possibles pour personnaliser la méthode d'approche et la forme des résultats :

Benchmark : permet de lancer un Benchmark (wahou quelle surprise !);
BenchmarkMode : permet de choisir quel type de Benchmark effectué:

- `Mode.Throughput` : calcul le nombre d'opérations dans un temps donné;
- `Mode.AverageTime` : calcul le temps moyen d'exécution;
- `Mode.SampleTime` : calcul combien de temps prend une méthode à s'exécuter;
- `Mode.SingleShotTime` : lance une seule méthode à la fois (utile pour les tests à froid);
- `Mode.All` : tous les modes les uns après les autres;

Fork : lorsqu'on lance le Benchmark, plusieurs forks (5 par défauts) sont lancés, avec chacun plusieurs appels à la méthode de test. Cette annotation permet de préciser combien de forks lancer.

Group : on met @Group(name) sur tout ce qu'on veut tester et quand on lance le benchmark tout ce qui est dans le groupe est lancé

GroupThread : précise combien de threads sont alloués, @GroupThreads(threadsNumber)

Measurement : permet de fournir les paramètres réels de la phase de test. Il est possible de spécifier le nombre d'itérations, la durée d'exécution de chaque itération et le nombre d'appels de test dans l'itération (généralement utilisé avec @BenchmarkMode

(`Mode.SingleShotTime`) pour mesurer le coût d'un groupe d'opérations au lieu d'utiliser des boucles).
Warmup : Même chose que pour @Measurement mais pour la phase de warmup.

OutputTimeUnit : utilisé avec `TimeUnit` on choisit quel type d'output est choisi pour le temps d'exécution (secondes, millisecondes, etc...);

State : définit la portée dans laquelle une instance d'une classe donnée sera disponible. JMH permet d'exécuter des tests dans plusieurs threads simultanément:

- `Scope.Thread` : état par défaut, une instance sera allouée à chaque thread exécutant le test donné.
- `Scope.Benchmark` : Une instance sera partagée par tous les threads exécutant le même test. Peut être utilisé pour tester les performances multithread d'un objet d'état.
- `Scope.Group` : Une instance sera allouée par groupe de threads.

Setup : indique à JMH que la méthode doit être appelée pour configurer l'objet d'état avant d'être transmis à la méthode de référence

TearDown : indique à JMH que cette méthode doit être appelée pour nettoyer (teardown) l'objet d'état après l'exécution du test.

Timeout : s'arrête au bout d'un moment, exemple : `@Timeout(time = 10, timeUnit = TimeUnit.SECONDS);`

3. Benchmark : boucle while VS boucle for

Pour tester le principe du Benchmark et vérifier que tout fonctionne bien, nous avons travaillé avec deux fonctions stupides qui faisaient une multiplication dans une boucle. Dans un premier cas nous avons fait ce calcul dans une boucle *for* et dans l'autre cas dans une boucle *while*. Les approches cherchant à chaque fois à obtenir les mêmes résultats, il est cohérent de les comparer.

```
package fr.polytechtours.javaperformance.tp2;

public class BouclesNazes {
   public static int stupidFor(int stop) {
      int i, stupid_variable = 0;

      for(i = 0; i < stop; i++)
         stupid_variable += 10 * i;

      return stupid_variable;
   }

   public static int stupidWhile(int stop) {
      int i = 0, stupid_variable = 0;

      while(i < stop) {
         stupid_variable += 10 * i;
         i++;
      }

      return stupid_variable;
   }
}
```

Pour mettre en place le benchmark, rien de très compliqué. Nous avons choisi le flag @Fork(1) afin de n'effectuer qu'une phase de test. On sait à peu près à quoi s'attendre comme résultat et c'est juste pour vérifier qu'on a bien compris comment ça fonctionne, inutile de perdre du temps à faire plusieurs forks.

Un point important qu'il ne faut pas oublier : le Blackhole ! En effet, lorsque nous écrivons une fonction, le compilateur la compresse. Il supprime ainsi les variables qui sont inutiles, notre boucle en utilise une mais malgré les calculs faits avec et la valeur de retour, elle est très inutile et si on la supprime cela ne change rien à l'exécution, le compilateur choisit donc de s'en débarrasser. Une méthode simple pourrait être d'afficher la variable avec un *print*, mais c'est une erreur car c'est une fonction lourde à exécuter et qui pourrait altérer les résultats d'un test.

La solution consiste donc à utiliser un blackhole, on indique au compilateur qu'on utilise la variable, mais c'est un mensonge car cette dernière ne va nulle part, c'est juste pour la conserver elle ainsi que tous les calculs associés.

Voici un exemple de ce que ça donne :

```
@Benchmark @Fork(1)
public void testStupidWhile(Blackhole blackhole) {
   blackhole.consume(BouclesNazes.stupidWhile(10000));
}
```

On exécute ça et on peut analyser les résultats. Voici celui correspondant à la boucle *for* par exemple.

```
# Warmup Iteration   1: 186815.133 ops/s
# Warmup Iteration   2: 185498.506 ops/s
# Warmup Iteration   3: 185759.358 ops/s
# Warmup Iteration   4: 186827.004 ops/s
# Warmup Iteration   5: 186421.672 ops/s
Iteration   1: 187055.883 ops/s
Iteration   2: 186574.883 ops/s
Iteration   3: 155156.116 ops/s
Iteration   4: 176529.969 ops/s
Iteration   5: 170606.179 ops/s


Result "fr.polytechtours.javaperformance.tp2.MyBenchmark.testStupidFor":
  183790.208 ±(99.9%) 5474.796 ops/s [Average]
  (min, avg, max) = (155156.116, 183790.208, 187116.571), stdev = 7308.696
  CI (99.9%): [178315.412, 189265.005] (assumes normal distribution)


# Run complete. Total time: 00:08:23

REMEMBER: The numbers below are just data. To gain reusable insights, you need to
follow up on why the numbers are the way they are. Use profilers (see -prof, -lprof), design
factorial experiments, perform baseline and negative tests that provide experimental
control, make sure the benchmarking environment is safe on JVM/OS/HW level, ask for
reviews from the domain experts. Do not assume the numbers tell you what you want
them to tell.

Benchmark                Mode  Cnt       Score   Error  Units
MyBenchmark.testStupidFor  thrpt   25  183790.208 ± 5474.796  ops/s

Process finished with exit code 0
```

Et voilà, on observe que le calcul entier se fait en moyenne avec 183790.208 calculs par secondes, que tout le benchmark s'est effectué en 8 secondes (nous avions fais ce test

avec 5 forks ce qui explique le long temps d'exécution, mais nous n’en avons affiché qu’un seul sur le résultat pour des soucis de lisibilité).

Si on fait la même chose avec la boucle *while* on obtient à peu près les mêmes résultats.

Conclusion

Ces tests sont totalement inutiles en soit car les deux boucles s'exécutent à la même vitesse et ce n’est pas surprenant, mais nous avons choisi de faire ce TP vraiment dans l’optique de comprendre le benchmark en général et non pas de comparer réellement deux programmes. Maintenant que nous savons exactement comment ça fonctionne nous pourrions l’appliquer à tout un tas de choses utiles pour faire de vraies comparaisons.

Un test intéressant à mettre en place pourrait être de comparer nos propres programmes avant et après optimisation, voir le gain de performance et pourquoi pas… la perte de performance si on s’y est mal pris.