Optimisation & Benchmark

Généralités

-----------

Avant de commencer à optimiser une fonction, il est important de faire des mesures pour:

* Constater les performances actuels. Cela sera utile pour quantifier le gain lors de laisse d optimisation
* S assurer que la fonction a besoin d optimisation. En effet, il est important de s assurer qu il y a un gain potentiel
* Une méthode pouvant appeler un certain nombre d autres méthodes, il est important de localiser les zones exactes nécessitant une attention particulière pour l optimisation

Les optimisations peuvent être diverse et incluent:

* la cpu (code plus rapide)
* la mémoire permanente (limiter la consommation statique)
* la mémoire temporaire (limiter les pics mémoires qui engendreraient des GC trop fréquents)

Mesure de performance (benchmark)

---------------------------------

Lorsque l on mesure les performances lies à un bout de code (pour simplifier, on supposera que ce code est délimité par une fonction), il est important de tenir compte des facteurs suivants qui influent sur les résultats mesures:

* lors de la première exécution de la fonction, les classes nouvellement références vont être chargés (le fichier class est chargé en mémoire puis la structure est créé pour être accéder par le class manager) puis initialisé (initialisation des variables et bloc statiques). Bien que souvent cela est faible, mais dans certains cas cela peut être important
* l appel répété du même code va faire que la VM va le "jitter" c est a dire transformer le byte code à interpréter en code natif bien plus efficace

Ces 2 éléments vont expliquer que l appel à la même méthode va évoluer au cours du temps indépendamment du contexte.

La VM étant multi-threadé, il est aussi important de limiter l'activité externe et donc s assurer que les mesures sont faite de manière unitaire sur la fonction.

Effet de bord d'une optimisation:

---------------------------------

Mis à part dans le cas (pas si rare) ou le code utilise un mauvais algorithm ou est trop complexe, l optimisation d un code aura toujours une conséquence négative sur l application. Par exemple, dans le cas d'ptimisation de la CPU, ces conséquences peuvent être:

* une plus grande consommation mémoire permanente.
* ne plus grande consommation mémoire temporaire. Bien qu'on pourrait penser que cela n a pas de conséquences réelles, n oublions pas que cela va faire que le gc passera plus de temps à effectuer sa tâche.
* une moins bonne lisibilité du code ou une complexification du code qui entraînera une maintenance plus difficile.

Pour être plus concret, prenons le cas d'une fonction qui effectue la recherche d'un élément dans un conteneur. Pour implémenter le conteneur, plusieurs possibilités:

* une simple ArrayList. Les elements rajoutés sont simplement mis à la fin de la liste. La recherche d'un élément sera lente mais la place occupée en mémoire est minimale.
* une HashMap. les éléments rajoutés sont les clés, et en valeur n'importe quel objet fera l'affaire. Si le hashCode des objets rajoutés est correctement implémenté (et c'est un point important), la recherche sera extrêmement rapide mais cela va engendrer une surconsommation de mémoire (chaque élément rajouté fait parti d'un objet plus complexe permettant le stockage en mode hash). Le rajout d'un élément dans le conteneur est un peu plus lent que dans une liste et il est important que la fonction hashCode soit efficace pour limiter l'overhead
* une liste triée. Etant une liste cela prend juste la place nécessaire en mémoire. La rechercher est relativement rapide mais le rajout d'un élément va être assez long car l'élément doit être rajouté au "bon" endroit.

On voit bien que le choix de la structure va avoir un impact direct sur les performances (mémoire et cpu). Il est donc judicieux de prendre la bonne structure en fonction de la contrainte du projet concernant cette fonction.

Il est donc important de balancer le gain de l optimisation au coût engendré.

L optimisation par le multithreading:

-------------------------------------

Lorsque des tâches doivent été effectuées il peut être intéressant de les lancer dans des thread séparés pour gagner en performance. Par contre l exécution étant limité par le nombre de processeurs, si les taches n effectuent que des activités utilisant le cpu a 100 il n y a pas de grand intérêt à les mettre en parallèle pour gagner en temps d exécution. Par contre si les taches sont en grande partie constitues d activité d attente (attente de résultat asynchrone comme une requête ip), mettre les taches en parallèle aura un gain signification sur les performances. En pratique, les taches étant une combinaison de ces 2 cas, cela signifie que pour gagner en performance et de manière efficace il faut que les taches soient effectués par un thread pool (pour limiter le nombre de thread en parallèle) et la taille du pool dépendra de l activité moyenne effectuée pour être optimum.

Le coût de la synchronisation:

------------------------------

Synchroniser un bloc de code ou une fonction a un léger coût sur la performance. De plus trop synchroniser augmente la possibilité de créer des deadlock. Il est donc important de synchroniser au plus juste et de manière judicieuse.

String

------

Concaténation de String

-----------------------

Doit-on utiliser StringBuffer ou StringBuilder ou la simple opération sur String ?

Operateur ‘+’ sur String

------------------------

1. Calcul sur littéraux (constantes ou directement l’utilisation de String sous forme "…") : lorsque des littéraux sont utilisés (même via des constantes finales statiques initialisés avec des littéraux), la JVM va automatiquement optimiser en effectuant a la compilation le calcul de ces littéraux. Donc return “A”+ “B”;’ va automatiquement se transformer (dans le bytecode généré) ‘return new "AB";’

2. Calcul sur variable ‘simple’: Ce qu’il faut savoir c’est que l’opérateur ‘+’ pour les String n’est pas natif mais est « simulé » grace au StringBuilder.

Ainsi, lorsque l’on additionne 2 String A et B un StringBuilder est utilisé.

‘return A+B;’ est exactement équivalent a ‘return new StringBuilder(A).append(B).toString();’

Pour s’en convaincre, il suffit de regarder le bytecode généré (contenu du .class) quand le fichier java contient la méthode ‘return A+B;’ :

public static java.lang.String concat(java.lang.String A, java.lang.String B);

0 new java.lang.StringBuilder [101]

3 dup

4 aload\_0 [A]

5 invokestatic java.lang.String.valueOf(java.lang.Object) : java.lang.String [143]

8 invokespecial java.lang.StringBuilder(java.lang.String) [105]

11 aload\_1 [B]

12 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [111]

15 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [134]

18 areturn

Line numbers:

[pc: 0, line: 60]

Local variable table:

[pc: 0, pc: 19] local: A index: 0 type: java.lang.String

[pc: 0, pc: 19] local: B index: 1 type: java.lang.String

Le bytecode associé aux instructions ‘return new StringBuilder(A).append(B).toString();’ donne:

public static java.lang.String concat(java.lang.String A, java.lang.String B);

0 new java.lang.StringBuffer [143]

3 dup

4 aload\_0 [A]

5 invokespecial java.lang.StringBuffer(java.lang.String) [145]

8 aload\_1 [B]

9 invokevirtual java.lang.StringBuffer.append(java.lang.String) : java.lang.StringBuffer [146]

12 invokevirtual java.lang.StringBuffer.toString() : java.lang.String [149]

15 areturn

Line numbers:

[pc: 0, line: 60]

Local variable table:

[pc: 0, pc: 16] local: A index: 0 type: java.lang.String

[pc: 0, pc: 16] local: B index: 1 type: java.lang.String

}

On pourra toutefois noter que le code utilisant les StringBuilder contient une opération de moins.

Donc en général, pour des operations simples il est plutôt recommandé d’utiliser l’opérateur ‘+’ qui permettra d’avoir un code simple et lisible.

3. Calcul sur variable ‘complexe’

Dans le cas ou les operations sur String ne sont pas simple, par exemple incluant une boucle for, il sera plus judicieux d’utiliser des StringBuilder. En effet, a chaque calcul intermédiaire, un StringBuilder sera utilisé pour effectuer le calcul, puis stocker dans une String créer à partir du StringBuilder. Si le nombre d’itération est important, cela va entrainer une création importante d’objet temporaire (toutes les StringBuilder et String intermédiaire) et donc amener à une exécution plus fréquente ou plus longue du GC (tout en étant moins performant)

StringBuffer ou StringBuilder

-----------------------------

La seule différence entre ces 2 structures viens du fait que StringBuffer est synchronisé (et permet donc des accès concurrents) alors que StrinBuilder ne l’est pas. Il faudra de préférence utiliser le StringBuilder sauf si la variable est partagée.

instanceof ou cls.instanceOf()

------------------------------

Ces opérations permettent de vérifier si un objet implémente une interface ou étend une classe. La différence viens du fait que le premier est un mot clé alors que la 2 eme est une méthode faisant parti de la classe Object.

Le mot clé sera plus performant et fera un code plus lisible, néanmoins, la methode d’Object peux s’avérer utile quand la classe à laquelle on veut vérifier n’est pas connue à la compilation (ex : paramètre de fonction, classe identifiée par son nom seulement connu à l’exécution, …)

**public** **static** **boolean** testInstance(Object a, Class b) {

**return** a.getClass().isInstance(b);

}