|  |
| --- |
| Optimisation & Benchmark |
| A quoi ça sert et comment faire ? |
| NABIL GUISSOUMA |

Contenu

[Généralités 2](#_Toc430163093)

[Mesure de performance (benchmark) 2](#_Toc430163094)

[Problème de la mesure 2](#_Toc430163095)

[Effet de bord d'une optimisation 3](#_Toc430163096)

[L’optimisation par le multithreading: 3](#_Toc430163097)

[Optimisation du code généré 3](#_Toc430163098)

[Analyses & Exemple 4](#_Toc430163099)

[Concaténation de String (operateur ‘+ ‘) 4](#_Toc430163100)

[Calcul sur littéraux 4](#_Toc430163101)

[Calcul sur variable ‘simple’ 4](#_Toc430163102)

[Calcul sur variable ‘complexe’ 5](#_Toc430163103)

[StringBuffer ou StringBuilder 5](#_Toc430163104)

[Comparaison détaillé d’operations de String/StringBuilder : 6](#_Toc430163105)

[instanceof ou cls.instanceOf() 9](#_Toc430163106)

[Outils de profiling: 9](#_Toc430163107)

[JDK Profiling Tools 9](#_Toc430163108)

[Outils de profiling gratuit : 10](#_Toc430163109)

[Outils de profiling payant : 10](#_Toc430163110)

# Généralités

Lors de la vie d’un projet il peut être nécessaire de faire une analyse détaillée de l’application pour :

* Améliorer les performances (CPU ou mémoire)
* Corriger les fuites mémoires (objets créés et qui ne sont pas normalement « garbage collecté »)
* Corriger les problèmes de deadlock

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

* Constater les performances actuelles. Cela sera utile pour quantifier le gain lors de la phase 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 diverses et incluent:

* Le code exécuté (pour avoir un 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)

## Problème de la mesure

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 à 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 algorithme ou est trop complexe, l’optimisation d’un code aura toujours une conséquence « négative » sur l’application. Par exemple, dans le cas d'optimisation de l’exécution du code, 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 souvent et prendra 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 éléments 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 partie d'un objet plus complexe permettant le stockage en mode hash). De plus, 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. Par contre, cela suppose que les éléments peuvent être ordonné (il faudra donc aussi créer la fonction de comparaison).

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

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

## L’optimisation par le multithreading:

Lorsque plusieurs 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%, le gain sera limité par le nombre de processeurs présents (en espérant que chaque thread sera exécuté par différent processeur). Par contre si les taches sont en grande partie constituées 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 effectuées 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.

## Optimisation du code généré

Il peut aussi être intéressant de diminuer le code généré (code compilé dans le .class) car il est très probable qu’un code ayant moins d’instructions (pour un algorithme identique) a exécuté sera plus performant (cf «Comparaison détaillé d’opérations de String/StringBuilder »)

## 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.

# Analyses & Exemples

## Concaténation de String (operateur ‘+ ‘)

Lorsque l’on doit faire des opérations sur les « String », la question se pose toujours si l’opérateur ‘+’ ou une des classes « StringBuffer » ou « StringBuilder » doit être utilisée.

En fait, cela dépend du contexte. En fonction des cas, l’une des structures sera plus appropriée.

### Calcul sur littéraux

Un littéral est une utilisation de String sous forme "…", soit directement soit indirectement grâce à une constante java (« final static »).

Lorsque des littéraux sont utilisés (même via des constantes finales statiques initialisés avec des littéraux), la JVM va automatiquement optimiser en effectuant à la compilation le calcul de ces littéraux.

Donc

**public** **static** String concat() {

**return** "A" + "B";

}

est exactement équivalent à (dans le bytecode généré)

**public** **static** String concat() {

**return** "AB";

}

// Method descriptor #148 ()Ljava/lang/String;

// Stack: 1, Locals: 0

public static java.lang.String concat();

0 ldc <String "AB"> [178]

2 areturn

Line numbers:

[pc: 0, line: 88]

}

### 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.

### 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.

## Comparaison détaillé d’opérations de String/StringBuilder :

On pourra remarquer que l’ajout de variable temporaire intermédiaire va potentiellement impacter les performances puisque le compilateur va générer des .class contenant le même séquencement que le fichier d’origine. Voici quelques bout de code donnant tous le même résultat en sortie mais avec de légères différences :

* Simple concaténation de String

**public** **static** String concat(String A, String B) {

**return** A + B + A;

}

// Method descriptor #185 (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

// Stack: 3, Locals: 2

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

0 new java.lang.StringBuilder [146]

3 dup

4 aload\_0 [A]

5 invokestatic java.lang.String.valueOf(java.lang.Object) : java.lang.String [192]

8 invokespecial java.lang.StringBuilder(java.lang.String) [150]

11 aload\_1 [B]

12 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

15 aload\_0 [A]

16 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

19 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [179]

22 areturn

Line numbers:

[pc: 0, line: 99]

Local variable table:

[pc: 0, pc: 23] local: A index: 0 type: java.lang.String

[pc: 0, pc: 23] local: B index: 1 type: java.lang.String

* Concaténation de String avec résultat final mis dans une variable locale

**public** **static** String concat(String A, String B) {

String str = A + B + A;

**return** str;

}

// Method descriptor #185 (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

// Stack: 3, Locals: 3

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

0 new java.lang.StringBuilder [146]

3 dup

4 aload\_0 [A]

5 invokestatic java.lang.String.valueOf(java.lang.Object) : java.lang.String [192]

8 invokespecial java.lang.StringBuilder(java.lang.String) [150]

11 aload\_1 [B]

12 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

15 aload\_0 [A]

16 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

19 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [179]

22 astore\_2 [str]

23 aload\_2 [str]

24 areturn

Line numbers:

[pc: 0, line: 103]

[pc: 23, line: 104]

Local variable table:

[pc: 0, pc: 25] local: A index: 0 type: java.lang.String

[pc: 0, pc: 25] local: B index: 1 type: java.lang.String

[pc: 23, pc: 25] local: str index: 2 type: java.lang.String

* Concaténation de String avec résultat intermédiaire

**public** **static** String concat(String A, String B) {

String str = A + B;

str += A;

**return** str;

}

// Method descriptor #185 (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

// Stack: 3, Locals: 3

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

0 new java.lang.StringBuilder [146]

3 dup

4 aload\_0 [A]

5 invokestatic java.lang.String.valueOf(java.lang.Object) : java.lang.String [192]

8 invokespecial java.lang.StringBuilder(java.lang.String) [150]

11 aload\_1 [B]

12 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

15 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [179]

18 astore\_2 [str]

19 new java.lang.StringBuilder [146]

22 dup

23 aload\_2 [str]

24 invokestatic java.lang.String.valueOf(java.lang.Object) : java.lang.String [192]

27 invokespecial java.lang.StringBuilder(java.lang.String) [150]

30 aload\_0 [A]

31 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

34 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [179]

37 astore\_2 [str]

38 aload\_2 [str]

39 areturn

Line numbers:

[pc: 0, line: 93]

[pc: 19, line: 94]

[pc: 38, line: 95]

Local variable table:

[pc: 0, pc: 40] local: A index: 0 type: java.lang.String

[pc: 0, pc: 40] local: B index: 1 type: java.lang.String

[pc: 19, pc: 40] local: str index: 2 type: java.lang.String

* Concaténation de String avec StringBuilder

**public** **static** String concat(String A, String B) {

**return** **new** StringBuilder(A).append(B).append(A).toString();

}

// Method descriptor #185 (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

// Stack: 3, Locals: 2

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

0 new java.lang.StringBuilder [146]

3 dup

4 aload\_0 [A]

5 invokespecial java.lang.StringBuilder(java.lang.String) [150]

8 aload\_1 [B]

9 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

12 aload\_0 [A]

13 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

16 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [179]

19 areturn

Line numbers:

[pc: 0, line: 108]

Local variable table:

[pc: 0, pc: 20] local: A index: 0 type: java.lang.String

[pc: 0, pc: 20] local: B index: 1 type: java.lang.String

* Concaténation de String avec StringBuilder en plusieurs étapes

**public** **static** String concat(String A, String B) {

StringBuilder sb = **new** StringBuilder(A);

sb.append(B);

sb.append(A);

**return** sb.toString();

}

// Method descriptor #185 (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

// Stack: 3, Locals: 3

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

0 new java.lang.StringBuilder [146]

3 dup

4 aload\_0 [A]

5 invokespecial java.lang.StringBuilder(java.lang.String) [150]

8 astore\_2 [sb]

9 aload\_2 [sb]

10 aload\_1 [B]

11 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

14 pop

15 aload\_2 [sb]

16 aload\_0 [A]

17 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [156]

20 pop

21 aload\_2 [sb]

22 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [179]

25 areturn

Line numbers:

[pc: 0, line: 86]

[pc: 9, line: 87]

[pc: 15, line: 88]

[pc: 21, line: 89]

Local variable table:

[pc: 0, pc: 26] local: A index: 0 type: java.lang.String

[pc: 0, pc: 26] local: B index: 1 type: java.lang.String

[pc: 9, pc: 26] local: sb index: 2 type: java.lang.StringBuilder

## 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);

}

# Techniques sur les optimisations

## Optimisation de l’exécution

Une fois que l’on a fait des mesures et que l’on a identifiés les fonctions qui nécessitent une attention particulière, il sera nécessaire de faire une relecture de code poussée des parties impliquées. En fonction des cas, plusieurs solutions sont possibles, à appliquer dans les fonctions traversées:

* Changer l’algorithmique de la fonction pour en trouver une plus efficace. Pour pouvoir faire cela, il faudra déjà comprendre ce que la fonction réalise fonctionnellement pour voir si cela peut être effectué différemment.
* Si une fonction est énormément appelé, voir si le nombre d’appel peut être diminué (voir si l’appel peut être « factorisé »).
* Si une fonction est énormément appelé, voir si le contenu ne peut pas être optimisé (même un petit gain de quelques ms sera visible en final vu que c’est une fonction appelée très souvent)
* Le rajout d’un cache peut être utile dans le cas d’opérations répétitives qui prennent du temps. Par contre, le premier appel d’une donnée qui n’est pas dans le cache sera tout aussi couteux.
* Il pourra aussi être intéressant d’analyser le bytecode effectivement généré par le compilateur java pour voir ce qui va réellement être exécuté. Pour cela il suffit de prendre le .class généré et de le déplacer dans la fenêtre des éditeurs d’Eclipse.

## Optimisation de la consommation mémoire (permanente ou temporaire)

De même que pour l’optimisation de l’exécution, les mesures de performances effectuées seront nécessaires ainsi qu’une relecture approfondie du code impacté.

Pour limiter la consommation mémoire, il sera nécessaire de retravailler les structures de données voir l’algorithmique. En choisissant un algorithme différent, des structures de données différentes pourront être utilisées.

Voici quelques remarques générales qui pourront vous aider lorsque vous essaierez de limiter la consommation mémoire :

* Utiliser les types de base (int, long, short, …) de préférence plutôt que les types Objet (Integer, Short, …). Ces types ne sont pas des objets et consomment juste les quelques octets (1 à 4 en général, voir 8 pour les long et double) servant à les stocker. Attention à l’auto-boxing/unboxing qui est automatiquement effectué par le compilateur (conversion automatique des types de base vers les types Objet : int <-> Integer, short <-> Short, …). Il faut savoir que la plupart des structures de la JDK prennent des Object en paramètre et même si un type de base est fourni, le compilateur pourrait automatiquement le transformer en Object (style Integer, …).
* Utiliser la bonne structure de donnée. Il pourrait être souhaité de créer la sienne si les classes « standards » sont moins efficace : Par exemple, une LinkedList consomme beaucoup plus de mémoire qu’un ArrayList en général, car pour chaque objet stocké, un autre objet est créé pour faire les liens entre les éléments. Mais attention à l’ArrayList qui a un tableau en interne qui peut excéder les besoins et du coup faire une surconsommation. Initialisé l’ArrayList à la bonne « taille » peut s’avérer judicieux, surtout que cela évitera les redimensionnements internes.
* Utiliser des « pools » d’objet. Cela permettra d’éviter une trop grande consommation d’objet temporaire (ayant une durée de vie limitée dans l’algorithme mais utilisé énormément). Par contre cela nécessite la création et l’utilisation explicite de ce pool d’objet dans tous les codes dépendants.

## Recherche de fuite mémoire

La recherche de fuite mémoire est un peu différent des activités précédentes car dans les cas précédents, le code marche bien et il est juste nécessaire d’améliorer les performances, alors qu’ici le code, bien qu’ayant l’air de bien marcher, a en fait un problème à corriger, qui est la fuite mémoire.

Une fuite mémoire est due au fait que certains objets restent référencé par le système par erreur. Par exemple, un code, le temps de son traitement a besoin d’être à l’écoute de certains événements. Pour cela un addXXXListener est nécessaire. Dès que le traitement est terminé, le composant est sensé faire un removeXXXListener mais a été oublié. Soit dans le cas nominal, soit dans certains cas aux limites ou cas d’erreurs. Un autre cas classique est l’ouverture d’une ressource en début de traitement mais sans sa fermeture (close) à la fin du traitement.

De même que pour les activités précédentes, un profilage (des objets en mémoire) sera nécessaire pour identifier les objets en cause. Sachant qu’à chaque exécution d’un code, il va générer toujours la même fuite, il suffit de faire des images des objets mémoires sur plusieurs exécutions successives (pour éviter d’être parasité par les objets temporaires qui seront garbage collectés, il sera souhaitable d’exécuté un GC avant de faire les mesures). Une fois ces images disponibles, il suffit d’identifier les objets dont le nombre augmente de manière linéaire en fonction des exécutions.

Une fois la liste identifiée, il reste à identifier le code qui crée cette fuite. Cette liste contiendra beaucoup d’objets « basiques » utilisés dans la plupart des codes (List, Map, Integer, tableaux, …) qui ne permettront pas de localiser le code en cause. Par contre, un de ces éléments sera en général moins basique et donnera une indication (par le nom de la classe) du package ou des classes en cause. Connaissant les objets qui ne sont pas normalement libéré, une relecture de code permettra d’identifier et de remédier au problème.

**Attention :** Il faut faire attention aux « faux positifs ». En effet, lors des analyses, il peut s’avérer que certaines parties du code consomment continuellement et de plus en en plus d’objets, mais cela peut être un fonctionnement normal. Par exemple, un code qui effectue du caching aura ce genre de comportement mais ce n’est pas un réel problème, voir c’est même un comportement voulu dans le but d’améliorer les performances. En général, dans la mesure du possible, lors de la recherche de fuite mémoire, il est souhaitable de désactiver les caches.

# Outils de « profiling »:

Pour pouvoir optimiser du code, il est nécessaire de connaitre les performances du code. Pour cela, il sera nécessaire d’utiliser des outils de profiling pour faire ces mesures.

## Outils de « profiling » du JDK

De nombreux outils sont directement disponible dans les JDK récents (JDK 1.5 et supérieurs). La documentation d’Oracle concernant ces outils est disponible ici :

<http://www.oracle.com/technetwork/java/javase/tooldescr-136044.html>

Tous ces outils utilisent la JVMTI qui est l’interface offerte par la JDK pour monitorer l’exécution.

Voici une partie des outils disponibles et leur utilité :

* HPROF - Heap Profiler -> profiler agent)
* Java VisualVM -> GUI
* JConsole Utility -> GUI: affiche et surveille l’évolution des informations globales (mémoire, thread, classes, ...) à l’aide de graphique
* jdb Utility -> débuggeur en commande en ligne
* jhat Utility -> permet de naviguer dans la topologie des objets à partir des d’une image de la « heap » : Cela permet de savoir pourquoi un objet est référencé et par qui. Cela peut être très utile lors de la recherche de fuite mémoire.
* jinfo Utility -> récupère les informations d'une jvm (propriétés diverses)
* jmap Utility -> affiche des statistiques mémoires
* jstack Utility -> affiche la stacktrace de tous les threads actifs. Très utile pour la recherche de deadlock
* jstat Utility -> outil donnant des informations sur la mémoire et les GC qui s'effectuent
* visualgc Tool -> GUI: c'est la version graphique de jstat

## Outils de profiling gratuit :

De nombreux outils gratuits existent pour profiler votre application et tous utilisent la même interface JVMTI fournie par la JDK. Par contre la plupart n’ont pas évolué depuis très longtemps.

* Profiler4j (2006):

<http://profiler4j.sourceforge.net/>

* JIP ou Java Interactive Profiler (2010) :

<http://sourceforge.net/projects/jiprof/>

* JMH : cela permet d’utiliser des annotations pour profiler spécifiquement certaines parties du code. Par contre cela ne donnera que des informations relatives à la CPU.

Site : <http://openjdk.java.net/projects/code-tools/jmh/>

Tutoriel : <http://soat.developpez.com/tutoriels/java/mesurer-performances-jmh/>

## Outils de profiling payant :

* JProfile de EJ Technologies : <http://www.ej-technologies.com/products/jprofiler/overview.html>

Après une utilisation rapide (il peut être utilisé pendant 10 jours pour évaluation), ce profiler semble complet et permet d’avoir des informations détaillés sur l’exécution (CPU), les threads et la mémoire (y compris sur les objets en particulier).