

La gestion de la mémoire en Java

(profilage avec Netbeans 5.x)

oschmitt@free.fr

Novembre 2006

Table des matières

1	Introduction.....	3
2	Gestion de la mémoire.....	4
2.1	Automatique ?.....	4
2.2	Vie et mort d'un objet.....	4
2.3	Mémoire générationnelle.....	4
2.3.1	La jeune génération.....	5
2.3.2	La génération tenured.....	5
2.3.3	La génération permanente.....	6
2.4	Les stratégies de gestion de la mémoire.....	6
2.4.1	Qu'est ce qu'une gestion efficace de la mémoire ?.....	6
2.4.2	Les collecteurs.....	7
2.4.3	Dimensionner les générations.....	7
2.4.3.1	La génération permanente.....	8
2.4.3.2	Le ratio Tenured/Young.....	8
2.4.3.3	Le ratio Eden/Survivor.....	9
2.5	Ebauche d'une méthodologie.....	10
2.5.1	Phase 0 : établir un scénario de test.....	10
2.5.2	Phase 1 : tester en activant la surveillance du ramasse-miettes.....	10
2.5.3	Phase 2 : interprétation des résultats.....	11
2.5.4	Phase 3 : calibrage.....	11
3	Profilage d'une application Java.....	12
3.1	Les outils inclus avec le JDK.....	12
3.1.1	Surveillance du ramasse-miettes.....	12
3.1.2	La console JMX : jConsole.....	12
3.2	GCViewer.....	15
3.3	Le profileur de Netbeans 5.0.....	16
3.3.1	Installation.....	16
3.3.2	Détection d'une fuite mémoire.....	16
4	Conclusion.....	22

1 Introduction

Java est un langage moderne.

Finies les migraines du développeur C à cause d'un pointeur mal initialisé ou d'un *malloc* caché dans un coin.

La machine virtuelle s'occupe des tâches ingrates, pendant que vous, développeur, développez en toute sérénité sans jamais soulever le capot.

Lorsque le temps de la mise en production arrive et que votre application subit l'assaut des utilisateurs, votre sérénité à toutes les chances de voler en éclat.

Application trop lente ? Trop gourmande ? Que se passe-t-il vraiment ? Comment surveiller l'activité de la JVM ?

La machine virtuelle a-t-elle toujours raison ou bien faut-il lui indiquer comment mieux faire son travail ?

Nous tenterons de répondre à ces angoissantes questions.

Cependant, le monde Java offre un nombre toujours plus important d'APIs ayant chacune leur spécificité en terme de performance, nous nous attacherons à présenter les aspects les plus importants du JDK 1.5 alias Tiger.

2 Gestion de la mémoire

2.1 Automatique ?

La gestion automatique de la mémoire est l'un des points forts les plus marquants de Java.

Java masque la complexité de cette tâche au développeur. L'acteur principal de cette tâche est le *garbage collector* ou ramasse-miettes.

A partir de la version 1.2 du JDK, le ramasse miette peut opérer suivant plusieurs algorithmes. Le développeur devait choisir quel était celui le plus adapté à son application.

Le JDK 1.5 et sa JVM va plus loin dans le sens où la JVM choisit l'algorithme en fonction de la machine qui l'héberge.

Cependant, l'activité du ramasse-miettes peut dégrader fortement les performances d'une application lorsque l'utilisation de la mémoire ne correspond pas aux réglages par défaut de la JVM.

Le symptôme le plus fréquent est le gel de l'application pendant plusieurs secondes voire plus (plusieurs minutes), sans que le système ne présente des signes de saturation.

2.2 Vie et mort d'un objet

Un objet vit dans la JVM lorsque le mot clef *new* est exécuté. La mémoire nécessaire est allouée sur le *heap* ou tas puis l'objet est consommé par l'application.

Sa mort est une chose plus mystérieuse. Le ramasse-miettes décide qu'un objet est mort lorsque il n'est référencé par aucun autre objet. Le ramasse-miettes le détruit et il disparaît à jamais du tas.

Le développeur se préoccupe de la création de l'objet mais jamais de sa mort.

2.3 Mémoire générationnelle

L'algorithme le plus simple pour collecter les objets morts est aussi le moins performant. Il consiste à parcourir l'ensemble du tas et à vérifier, pour chaque objet, la règle énoncée au point précédent.

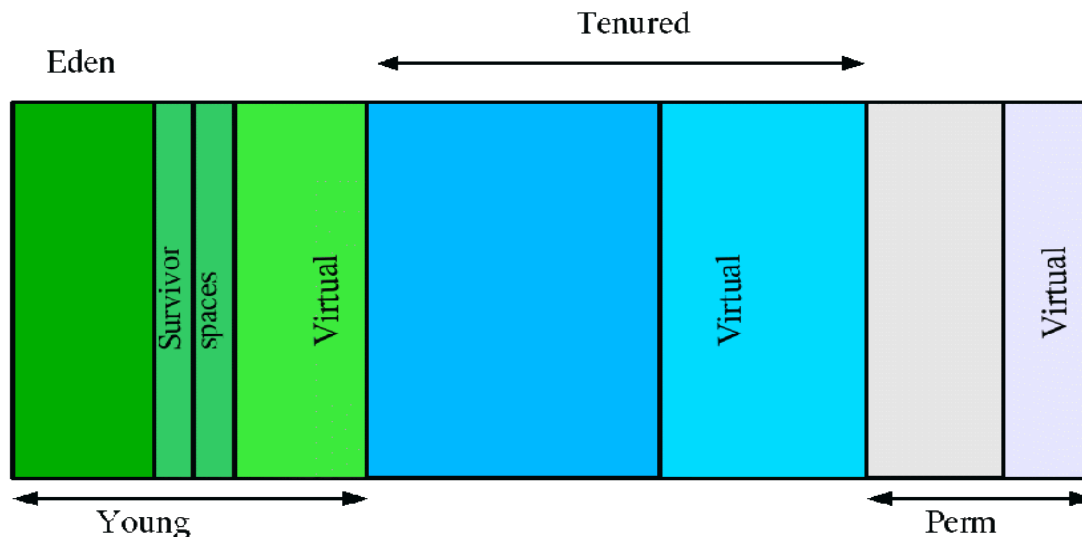
Cependant, plus la taille du tas augmente, plus le ramasse-miettes consomme de ressources pour parvenir à traiter l'ensemble des objets.

Le temps de traitement du tas peut être réduit en observant comment les applications utilisent les objets. Les ingénieurs de SUN se sont aperçus que la majorité des objets créés vivent peu de temps après leur allocation.

Ce sont des sortes d'éphémères : **la plupart des objets meurent jeunes.**

Prenez une classe Java quelconque de votre application et analysez là sous cet angle. Vous remarquerez alors le nombre impressionnant d'objets locaux dans vos méthodes, comme les *Iterator* par exemple, qui ne sont utiles que le temps d'une boucle.

Fort de ce constat, le tas a été découpé en différentes zones de mémoire. Ces zones correspondent à une étape de la vie d'un objet.



Les zones nommées *virtual* sont des zones de mémoire réservée auprès du système mais qui ne sont pas encore peuplées par des objets. Les zones *virtual* sont utiles lorsque les zones peuplées sont saturées.

L'enjeu d'un tel découpage est d'**optimiser la recherche d'objets morts**.

Remarque: l'agencement des générations peut varier d'un collecteur à un autre.

2.3.1 La jeune génération

Lorsqu'un objet est créé il débute sa vie dans la zone *young* et plus précisément dans la sous-zone *eden*. L'ensemble des objets présents dans cette zone est désigné par l'expression *young generation* ou jeune génération.

Le ramasse-miettes tente de nettoyer en priorité la jeune génération d'objets (*young*) car d'après les observations (la majorité des objets meurent jeunes), il est inutile de parcourir l'ensemble du tas.

Cette opération est appelée *minor collection* ou collecte mineure car elle implique un nombre limité d'objets. Le ramasse-miettes déclenche une collecte mineure lorsque la zone *young* est pleine (*eden*, *survivor*).

Tous les objets déjà mort situés dans l'*eden* disparaissent du tas. Les survivants sont copiés dans une des sous-zone *survivor space*. Il y a en fait deux *survivor space* mais une seule contient des objets.

Les objets présents dans un *survivor space* passe dans l'autre *survivor space* à chaque collecte mineure jusqu'à ce qu'ils aient effectué un nombre donné d'aller retour ou que la place viennent à manquer.

2.3.2 La génération tenured

Lorsque les objets présents dans le *survivor space* ont vécu assez longtemps, c'est à dire ont survécu à un nombre déterminé de collectes mineures, ils sont transférés dans la zone *tenured*.

Lorsque une application alloue brutalement un grand nombre d'objets, l'algorithme précédent n'est pas

forcément respecté. Le ramasse-miettes peut être contraint de transférer les objets vers la zone *tenured* plus vite que prévu et d'augmenter la taille de la zone *young*.

Une collecte majeure ou *major collection* intervient lorsque la zone *tenured* se remplit au delà d'une certaine limite.

Cette limite est en fait un ratio entre la mémoire libre et la mémoire occupée. Une collecte majeure est bien plus coûteuse qu'une collecte mineure. Les tailles des zones influent considérablement sur le coût d'une collecte. Or, la zone *tenured* est plus vaste que la zone *young*.

Le ramasse-miettes tente toujours de limiter la taille du tas mais lorsque la demande est trop forte, il étirera les différentes zones jusqu'à la limite du tas ou bien jusqu'à ce que la taille courante du tas suffise au besoin de l'application.

L'exception *java.lang.OutOfMemoryException* est levée lorsque le ramasse-miettes ne peut augmenter la taille du tas et qu'une collecte majeure ne permet pas de libérer suffisamment de place pour que l'application continue.

2.3.3 La génération permanente

La zone *perm* définit une zone de mémoire qui accueille des objets permanents. Il n'y a pas de collectes dans cette zone. Elle contient notamment les classes Java qui sont chargées au démarrage de la JVM et au cours de l'exécution de l'application. Il arrive parfois que cette zone soit trop petite et provoque un blocage de votre application. Dans ce cas le message de l'exception *OutOfMemoryException* dira *PermGen space*.

2.4 Les stratégies de gestion de la mémoire

2.4.1 Qu'est ce qu'une gestion efficace de la mémoire ?

Tout d'abord plusieurs points de vue sont possibles en fonction des contraintes de production. Il convient de choisir un objectif :

- l'application doit consommer un minimum de mémoire
- l'application ne doit pas geler

Ensuite, plusieurs critères vont permettre de mesurer la réalisation de l'objectif :

- *footprint* : indique l'empreinte mémoire de la JVM sur le système
- *throughput* : indique le pourcentage total du temps qui n'est pas utilisé par le garbage collector
- *pauses* : indique le temps de gel de l'application

Les objectifs précités correspondront à un type d'application. Par exemple, il est souhaitable de limiter les gels d'une l'application avec IHM : il est très agaçant d'utiliser une IHM qui gèle souvent.

Une application web peut tolérer des gels modérés grâce à Internet et sa latence. Un internaute accepte des temps de réponse allant jusqu'à plusieurs secondes.

2.4.2 Les collecteurs

La JVM offre trois types de collecteurs qui utilisent des stratégies différentes de collecte. Tous trois sont des collecteurs générationnels.

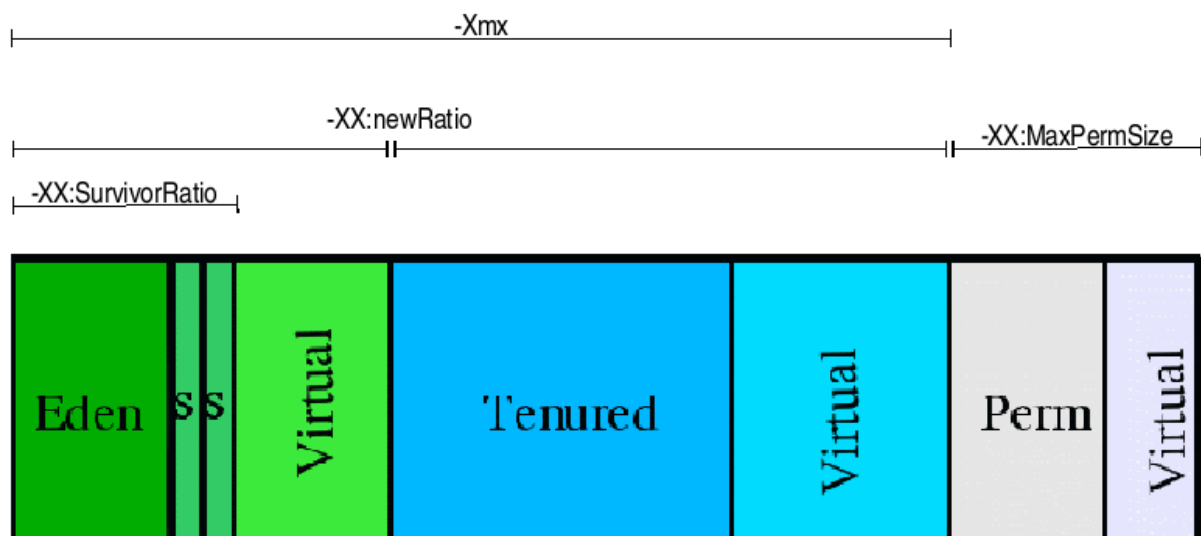
- *serial collector* : c'est le collecteur par défaut.
- *throughput collector* : la collecte de la *young generation* est parallélisée tandis que celle de la *tenured* est identique à celle du *serial collector*.
- *concurrent low pause* : les différentes collectes sont parallélisées afin de limiter les pauses.

Remarque : les explications de ce tutoriel concernent le *serial collector*.

2.4.3 Dimensionner les générations

La taille des générations influe de manière importante sur les performances de l'application.

Le schéma suivant montre les différentes options de la JVM qui permettent de modifier la taille des générations :



Voyons ensuite comment dimensionner ces zones et les impacts qui en découlent.

2.4.3.1 La génération permanente

La taille de la génération permanente est un élément de configuration important notamment si votre application charge des classes dynamiquement (serveur J2EE, plugins, ...).

En effet, les classes Java sont chargées dans cette zone. Cette zone est sensible lorsque vous utilisez un conteneur de Servlet/JSP comme Tomcat.

Dès que vous déployez de nouvelles applications webs, les classes Java du dossier WEB-INF seront chargées dans la génération permanente en fonction des besoins de l'application. La taille maximale par défaut est 64 mégas octets : `-XX:MaxPermSize=64m`.

Impacts :

la génération permanente n'est pas considérée comme faisant partie du *heap*. A ce titre l'activité du ramasse-miettes n'est pas perturbée par cet ajustement.

2.4.3.2 Le ratio Tenured/Young

Pour rappel, la *young generation* comprend l'*eden*, les deux *survivor space* et la mémoire virtuelle de réserve.

La taille de ces générations n'est pas paramétrable directement: un ratio est utilisé qui indique le rapport entre la taille de la *tenured gen* et la *young gen*. Ce ratio est ensuite appliqué sur la valeur de l'option `-Xmx` qui fixe la taille totale du *heap*.

Ainsi le ratio est précisé par l'option `-XX:NewRatio=n` ou *n* est un nombre entier.

Si *n* vaut 3 alors la *young gen* est 3 fois plus petite que la *tenured gen*.

La croissance de la *young gen* peut être maîtrisée grâce aux options `-XX:MaxNewSize=nm` et `-XX:NewSize=nm` où *n* est un nombre entier.

Dans la première option, *n* indique que la *young gen* grandira de *n* mégas octets maximum, tandis que dans la seconde *n* indique que l'accroissement par défaut est de *n* mégas octets.

Impacts :

Plus la *young gen* est vaste plus les collectes mineures sont rares. Cependant, lorsque la taille du *heap* est peu extensible, cela va diminuer la taille de la *tenured gen* ce qui aura pour conséquence d'augmenter la fréquence des collectes majeures.

Les options `-XX:MaxNewSize` et `-XX:NewSize` sont utiles lorsque l'application produit des objets avec une distribution de vie particulière (ou *lifetime distribution*).

L'outil Jmeter (<http://jakarta.apache.org/jmeter/>) est un cas d'utilisation très intéressant dans la mesure où il produit ce type d'objets et permet d'effectuer des tests de montée en charge d'applications Java et donc de mesurer la performance.

Voici un extrait du script de démarrage *bin/jmeter.sh* :

```
# This is the base heap size -- you may increase or decrease it to fit your
# system's memory availability:
HEAP="-Xms256m -Xmx256m"

# There's an awful lot of per-sample objects allocated during test run, so we
# need a large eden to avoid too frequent scavenges -- you'll need to tune this
# down proportionally if you reduce the HEAP values above:
NEW="-XX:NewSize=128m -XX:MaxNewSize=128m"
```

Le commentaire indique que Jmeter produit un nombre considérable d'objets qui ne survivent pas à leur première collecte mineure. Or, sans ce paramétrage, l'*eden space* se trouverait constamment saturé (max heap à 256m) ainsi que les *survivor space* associés ce qui impliquerait de déclencher fréquemment des collectes mineures tout en copiant les objets dans la *tenured gen*, cette dernière se trouvant à son tour saturée d'objets qui n'auraient pas dû échouer chez elle.

La croissance de la *young gen* se faisant par bonds de 128 mégas octets, cela assure que les collectes mineures interviendront moins fréquemment tout en étant efficaces car les objets ne seront plus copiés dans la *tenured gen* pour cause de sous dimensionnement des *survivor space* et de l'*eden space*. Ils seront très vite supprimés du *heap*.

Le cas Jmeter montre parfaitement que le développeur doit connaître la manière dont l'application consomme les objets. Toutefois, la gestion par défaut de la mémoire est basée sur des retours d'expérience accumulés au fil des ans.

2.4.3.3 *Le ratio Eden/Survivor*

Ce ratio détermine la taille des *survivor space* par rapport à l'*eden space*. L'option *-XX:SurvivorRatio=n* permet de spécifier ce ratio. La taille effective est ensuite calculée à partir de la taille de la *young gen*.

Impacts :

Normalement cette option à peu d'impact sur les performances, il faut garder à l'esprit que les différents espaces sont efficaces si ils sont correctement dimensionnés, c'est à dire ni trop vide ni trop plein.

2.5 Ebauche d'une méthodologie

L'optimisation de la gestion de la mémoire est une tâche qui peut s'avérer complexe.

En effet, une application est une entité dynamique qui répond aux sollicitations d'un ou plusieurs utilisateurs. Certaines applications seront mono utilisateur tandis que d'autres seront massivement multi utilisateurs. La construction de l'application reposera parfois sur un ensemble de librairies tierces dont l'impact sur les performances peut être déterminant.

La recherche de la bonne configuration implique :

- une connaissance des mécanismes de la gestion de la mémoire par la JVM
- la détermination du *lifetime distribution* des objets de l'application

2.5.1 Phase 0 : établir un scénario de test

Il convient d'établir un scénario de test qui devra reproduire un problème de performance où dont on veut vérifier qu'il n'a pas d'impacts.

2.5.2 Phase 1 : tester en activant la surveillance du ramasse-miettes

La JVM propose des informations sur l'activité du ramasse-miettes. Pour activer la surveillance, il suffit de lancer la JVM comme suit :

```
java -verbose:gc
```

Cette option produit des traces dans la console.

```
[GC 2653K->2242K(2928K), 0.0025180 secs]
[GC 2745K->2352K(2928K), 0.0018850 secs]
[GC 2864K->2416K(3056K), 0.0023560 secs]
[Full GC 2416K->2296K(3056K), 0.1035500 secs]
[GC 2808K->2336K(4408K), 0.0024140 secs]
[GC 2848K->2363K(4408K), 0.0022280 secs]
[GC 2875K->2417K(4408K), 0.0018240 secs]
```

Il y a deux types de traces, celle qui indique une collecte mineure et celle qui indique une collecte majeure.

Exemple de collecte mineure :

```
[GC 2653K->2242K(2928K), 0.0025180 secs]
```

Exemple de collecte majeure :

```
[Full GC 2416K->2296K(3056K), 0.1035500 secs]
```

Le nombre avant la flèche indique la taille totale des objets en kilo octets qui sont vivants dans le tas avant la collecte (sauf la génération permanente). Le nombre après la flèche indique la taille des objets vivants après la collecte. Le dernier nombre indique le temps nécessaire à la collecte.

Il est possible d'indiquer à la JVM de tracer ces informations dans un fichier. Ce fichier peut être ensuite exploité par différents outils.

```
java -verbose:gc -Xloggc:gc.log
```

D'autres options ajoutent des niveaux de détails comme `-XX:+PrintGCDetails` ou `-XX:+PrintGCTimeStamps`. L'option `-XX:+PrintGCTimeStamps` est indispensable pour estimer la fréquence des collectes.

2.5.3 Phase 2 : interprétation des résultats

Le fichier de logs produit par la JVM doit être analysé à la lumière des explications précédentes. Par exemple, les gels fréquents et inexplicables (non associés à un traitement coûteux) de l'application se traduira le plus souvent par des collectes majeures rapprochées et conséquentes (en terme de temps).

2.5.4 Phase 3 : calibrage

Fort de votre interprétation, il faudra ajuster la taille du *heap* ou dans des cas plus rares, ajuster les ratios comme pour JMeter. Ensuite, vous retournerez à la phase 1.

En effet, il est peu probable que vous parveniez à obtenir la configuration parfaite du premier coup: elle sera souvent affaire de compromis entre le débit et l'empreinte (*throughput* et *footprint*).

Lorsque, votre administrateur système et vos utilisateurs ne se plaignent plus : vous avez gagné !

La tâche du développeur est compliquée dans le cadre des applications J2EE car une part non négligeable des ressources est prise par le conteneur J2EE. Il convient d'étudier l'empreinte du conteneur à « vide » et de se plonger dans la documentation du conteneur pour en tirer les informations sur la performance.

Néanmoins, une panoplie d'outils aide le développeur dans sa démarche et permet de surpasser la méthode empirique.

3 Profilage d'une application Java

3.1 Les outils inclus avec le JDK

3.1.1 Surveillance du ramasse-miettes

Voir chapitre précédent.

3.1.2 La console JMX : jConsole

La console JMX est livrée avec à partir du JDK 1.5.

Java Management eXtension est une API qui permet de surveiller les applications Java à distance.

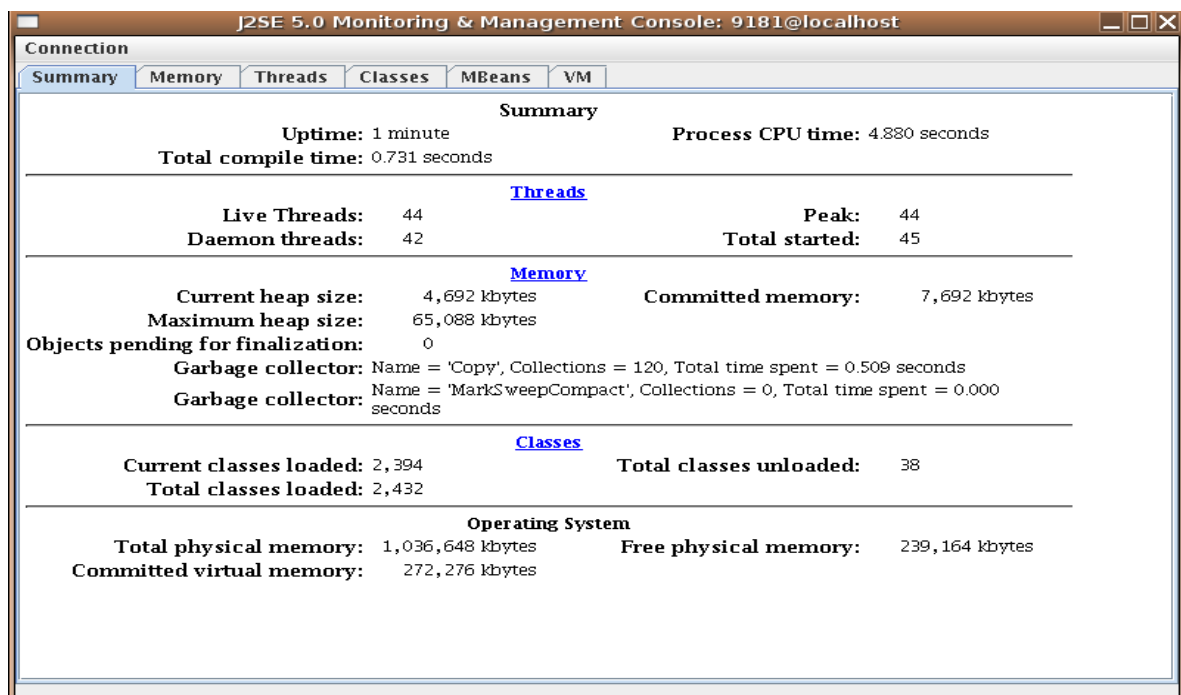
La console se présente sous la forme d'une application Swing dont le lanceur se trouve dans le dossier *bin* du JDK.

Pour lancer la console, exécutez le programme *JAVA_HOME/bin/jconsole*.

Votre application Java (serveur J2EE, ...) est démarrée avec quelques options supplémentaires pour activer le serveur JMX sur lequel la console se connectera.

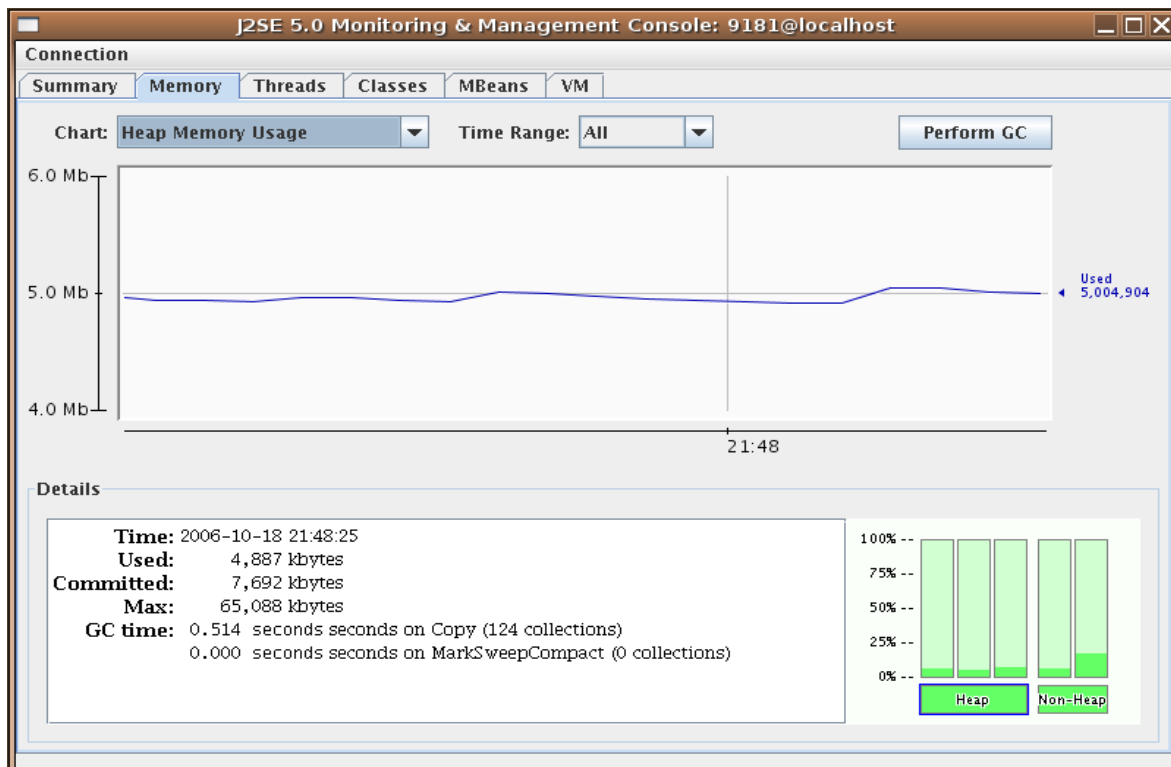
Par exemple, l'option *-Dcom.sun.management.jmxremote* active JMX sur la JVM.

Lorsque la console établit la connection, elle vous propose une interface en 6 onglets dont le premier présente un résumé de l'état de la JVM :



Les informations sur la mémoire sont intéressantes bien qu'incomplètes car la taille des différentes zones n'y figure pas. Les statistiques qui concernent le ramasse-miettes (*garbage collector* sur l'image) indique le nombre de collectes mineures (*Copy*) et le nombre de collectes majeures (*MarkSweepCompact*).

L'onglet **Memory** comble cette lacune. Cet onglet vous permet de suivre précisément l'évolution de la ressource mémoire :

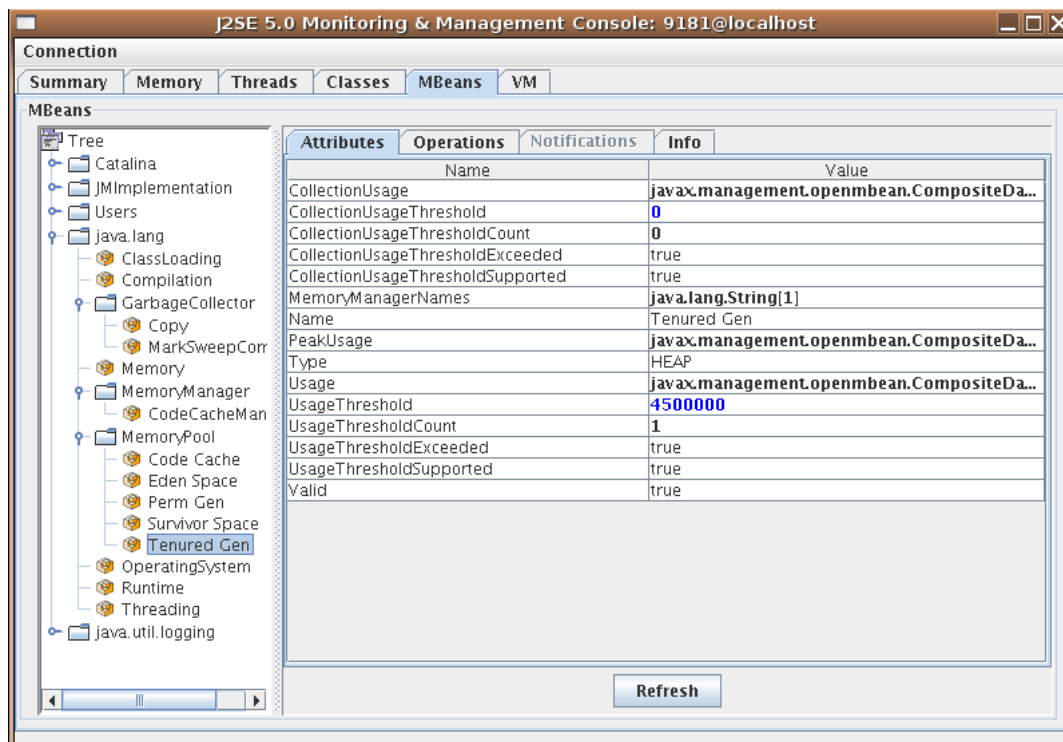


Il est possible de suivre l'évolution du tas dans sa globalité.

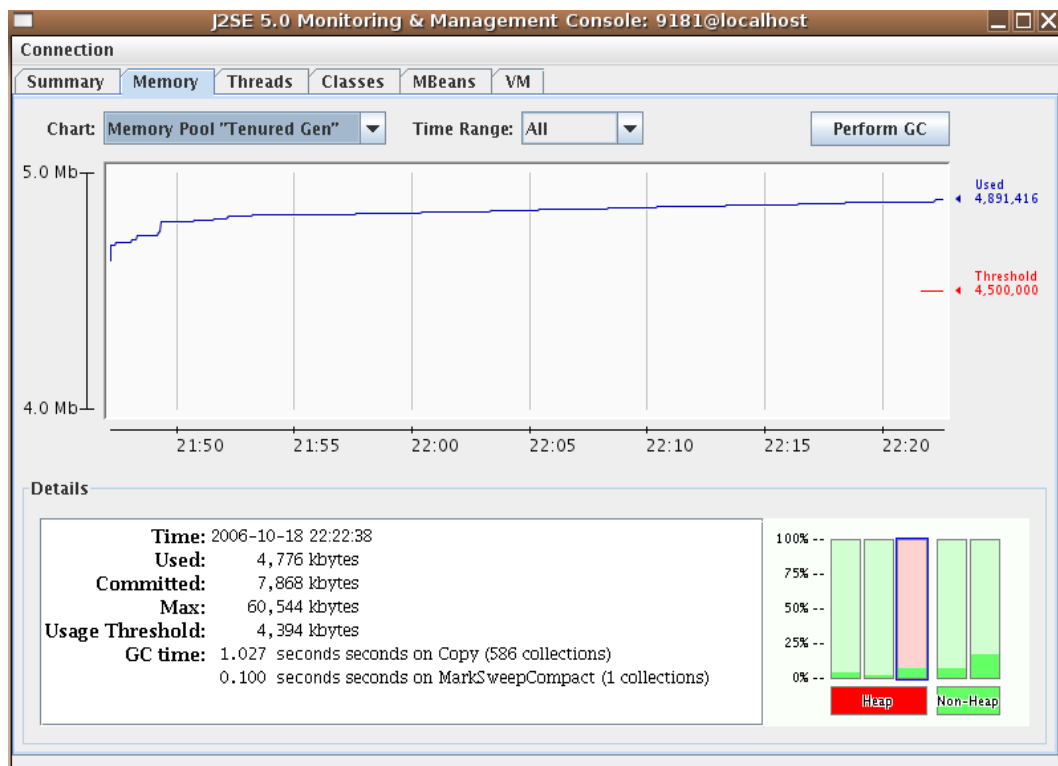
Si vous souhaitez connaître l'état d'une génération, il vous suffit de choisir dans la liste des graphiques disponibles ou bien de sélectionner une jauge (en bas à droite).

Il est possible d'invoquer le ramasse-miettes pour provoquer une collecte majeure : le bouton *Perform GC*.

Enfin, l'onglet Mbeans permet d'accéder à des beans de gestion de la JVM :



Il est possible d'activer des contrôles sur certaines zones de mémoire. La capture d'écran ci-dessus montre que l'attribut *UsageThreshold* est positionné à 4500000 octets. Si la quantité de mémoire utilisée pour la génération *tenured* dépasse cette limite alors la JConsole nous signalera l'anomalie :



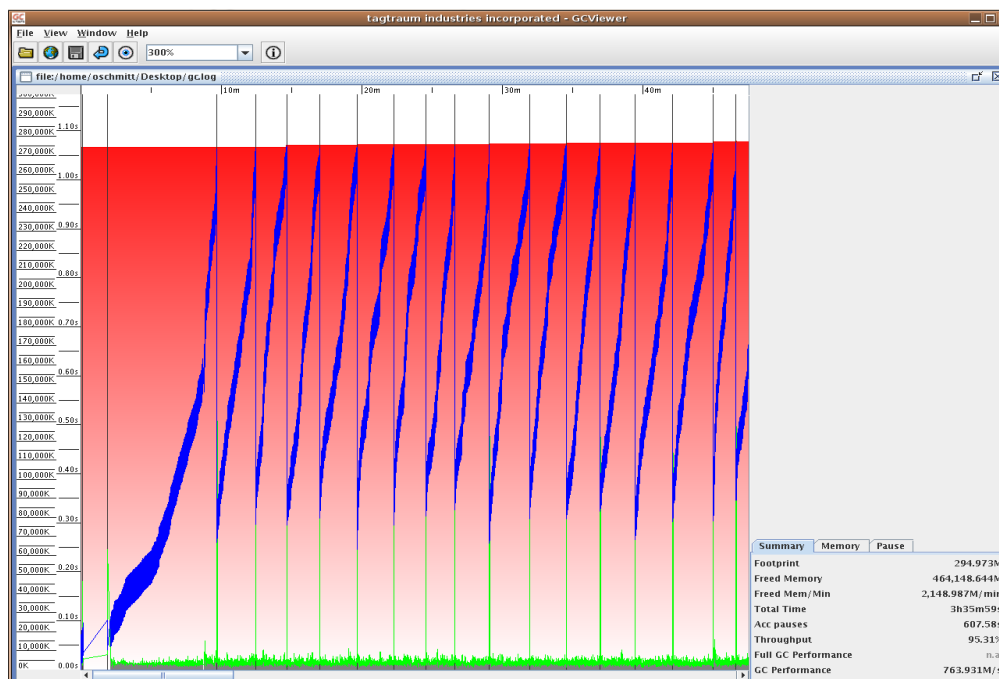
Cet outil est particulièrement utile pour suivre l'évolution du tas.

Il n'est pas obligatoire de lancer la console à partir du JDK : Netbeans offre un plugin très pratique qui permet de lancer la console à partir de l'IDE.

3.2 GCViewer

GCViewer (<http://www.tagtraum.com/gcviewer.html>) est un outil open source écrit en Java qui permet visualiser les logs produits par l'option `-verbose:gc`.

Exemple de restitution dans l'outil :



GCViewer permet de dégager une vue d'ensemble du comportement de la JVM lors de l'exécution de l'application. La restitution utilise une double échelle en ordonnée qui présente le temps de collecte et la quantité de mémoire consommée.

L'onglet en bas à droite présente des statistiques très intéressantes.

Consultez la documentation de GCViewer pour l'installation et le lancement de l'outil.

3.3 Le profileur de Netbeans 5.0

3.3.1 Installation

Le profileur nécessite Netbeans 5.0, vous devez installer Netbeans au préalable.

Téléchargez la distribution du profileur sur le site <http://www.netbeans.org/products/profiler/>.

Ensuite, il suffit d'exécuter l'installateur.

Lancez Netbeans.

Le menu *Profile* doit apparaître dans la liste des menus.

Le profileur doit passer par une phase de calibration afin de prendre en compte la puissance de votre machine.

Vous pouvez lancer cette opération via le menu *Profile > Advanced commands > Run profiler calibration*.

Remarque: la procédure est identique avec le récent Netbeans 5.5

3.3.2 Détection d'une fuite mémoire

Une fuite mémoire est le problème numéro un du développeur en quête de performance.

On pourrait définir une fuite mémoire comme la **consommation continue et inattendue de mémoire par un programme**.

La réalité est plus complexe. Une fuite mémoire peut provenir de votre code ou bien d'une librairie tierce utilisée par votre code.

Par exemple, le lien suivant recense une liste de librairies qui contiennent des fuites mémoires : <http://opensource.atlassian.com/confluence/spring/pages/viewpage.action?pageId=2669>.

Rappel: un objet est détruit par le ramasse-miettes si aucun autre objet vivant ne le référence.

Nous allons étudier comment repérer une fuite mémoire à partir d'une application Java sur mesure.

Cette application contient deux classes **Main** et **LeakThread**.


```

package dvp.performance;

/**
 * Exemple de code qui produit des objets jamais détruits
 * @author oschmitt
 */
public class LeakThread extends Thread {

    // Contient des couples float[1], double[1]
    private java.util.HashMap map = new java.util.HashMap();

    public void run() {
        while (true) {
            try {
                // Ajoute un couple float[1], double[1] : la clef est différente pour
chaque itération !
                map.put(new float[1], new double [1]);
                // Dort 100 ms
                Thread.sleep(100);
                // Lance le ramasse-miettes pour tenter de tuer les objets
                System.gc();
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}

package dvp.performance;

/**
 *
 * @author oschmitt
 */
public class Main {

    public static void main(String argv[]) throws InterruptedException{

        LeakThread leakThread = new LeakThread();
        leakThread.start();
        while(true){
            Thread.currentThread().sleep(200);
        }
    }
}

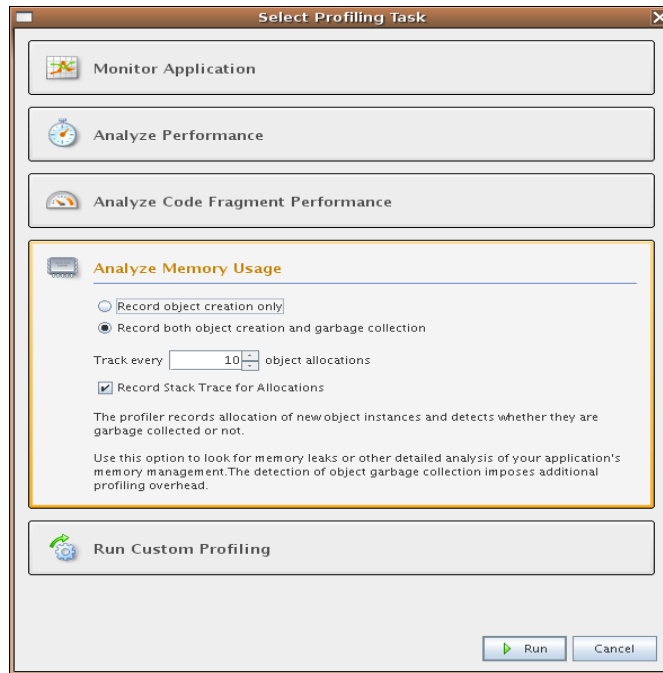
```

La classe **LeakThread** est un thread qui va consommer la mémoire sans jamais la restituer.

Pour s'en convaincre il suffit de lancer le profileur sur l'application.

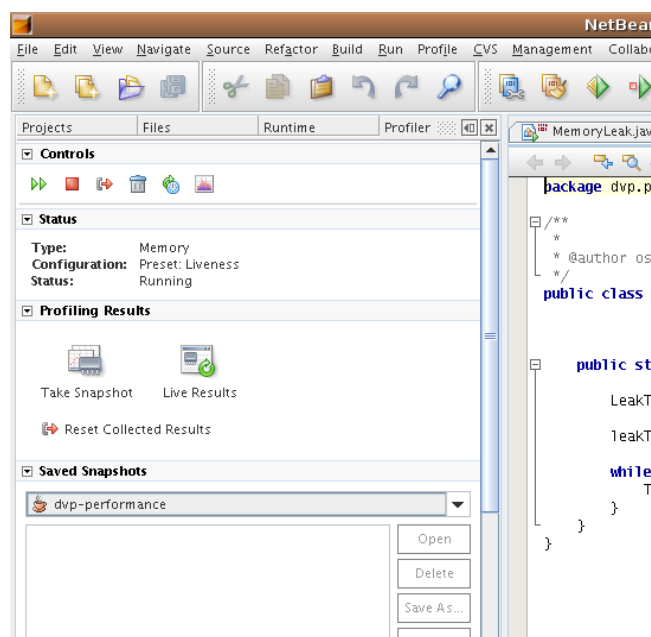
Pour cela, il suffit d'effectuer un click droit sur le projet Java puis de choisir *Profile project*.

La boîte de dialogue suivante s'affiche :




Choisissez *Analyze Memory Usage* puis cochez *Record both object creation and garbage collection*.

Ensuite cliquez sur le bouton *Run*.

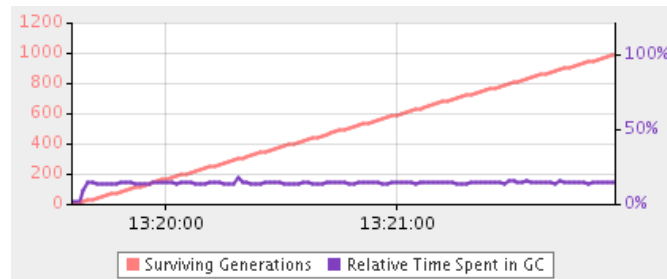


L'onglet *Profiler* s'ajoute aux autres onglets dans la marge gauche. Cet onglet permet de piloter le profileur.

La première chose à faire pour détecter une fuite mémoire est d'afficher les métriques globales du profileur : il suffit de cliquer sur l'icône  dans la zone de contrôle du profileur.

L'onglet *Telemetry Overview* s'affiche avec trois graphiques dont celui de l'activité du ramasse-miettes.

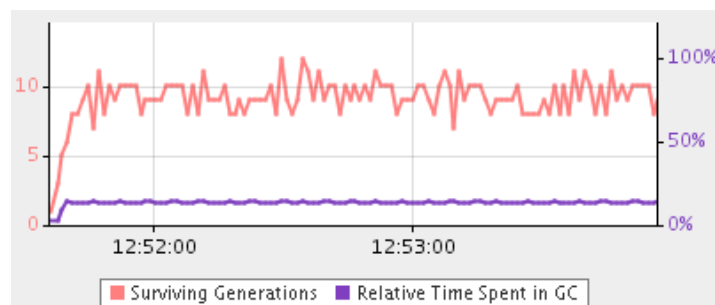
Ce graphique est très intéressant dans la mesure où il permet de détecter une fuite importante rapidement.



En effet, la graphique indique le temps passé dans le ramasse-miettes et le nombre de générations survivantes.

Cette dernière information est cruciale car dans le cas d'une fuite, ce nombre a tendance à augmenter. Plus un objet survit aux collectes successives plus ce nombre augmente.

Or dans le cas d'une application qui libère correctement les objets, le graphe prend la forme d'une dent de scie au gré des créations et libérations d'objets :



On voit d'emblée que le graphique obtenu à partir du profilage de notre application de test présente un nombre toujours plus important de générations survivantes.

Pour y voir plus clair, un clic sur *Live Results* fait apparaître un tableau de statistiques sur les classes Java de l'application et les instances associées. Ce tableau s'actualise automatiquement, vous pouvez ainsi suivre l'évolution des métriques.

Class Name - Live Allocated Objects	Live B...	Live Bytes	Live Obj...	Allocate...	Avg. Age	Generati...
double[]		7,248 B (36.3%)	302 (32.8%)	302	1431.9	302
java.util.HashMap\$Entry		7,176 B (36%)	299 (32.5%)	299	1418.1	299
float[]		4,816 B (24.1%)	301 (32.7%)	301	1416.9	301
java.util.HashMap\$Entry[]		144 B (0.7%)	1 (0.1%)	2	2822.0	1
dvp.performance.LeakThread		96 B (0.5%)	1 (0.1%)	1	2822.0	1
java.lang.Object[]		56 B (0.3%)	1 (0.1%)	1	2822.0	1
java.lang.Package		48 B (0.2%)	1 (0.1%)	1	2822.0	1
java.util.HashMap		40 B (0.2%)	1 (0.1%)	1	2822.0	1

La capture d'écran ci-dessus présente les résultats du profilage après plusieurs minutes d'exécution de l'application. On peut voir très nettement que 3 classes se détachent du lot : **float**, **double** et **HashMap\$Entry**.

Le profileur présente des informations très intéressantes pour chaque classe:

- *Live Bytes* : l'encombrement mémoire en octets de toutes les instances
- *Live Objects, Allocated Objects* : le nombre total d'instances vivantes
- *Avg. Age* : âge moyen des instances de la classe, c'est à dire la somme des âges de toutes les instances divisée par le nombre d'instances
- *Generations* : le nombre de génération d'instances pour une classe. La génération d'un objet renvoie à sa naissance, deux objets créés au même instant sont de la même génération.

Les instances de ces classes représentent $32.8 + 32.5 + 32.7$ soit presque 99 % des objets vivants de l'application soit $302 + 299 + 302$ objets vivants.

D'autre part, le nombre de générations qui cohabitent dans le tas est très élevé ainsi que l'âge moyen des instances.

Tout ceci indique une fuite mémoire.

Il suffit de cliquer droit sur la classe qui nous intéresse et de choisir *Take Snapshot And Show Allocation Stack Trace*, pour connaître la pile d'allocations des instances.


Method Name - Allocation Call Tree	Live B...	Live Bytes	Live Obj...	Allocate...	Avg. Age	Genera...
double[]		3,000 B (100%)	125 (100%)	125	631.8	
dvp.performance.LeakThread.run()		3,000 B (100%)	125 (100%)	125	631.8	

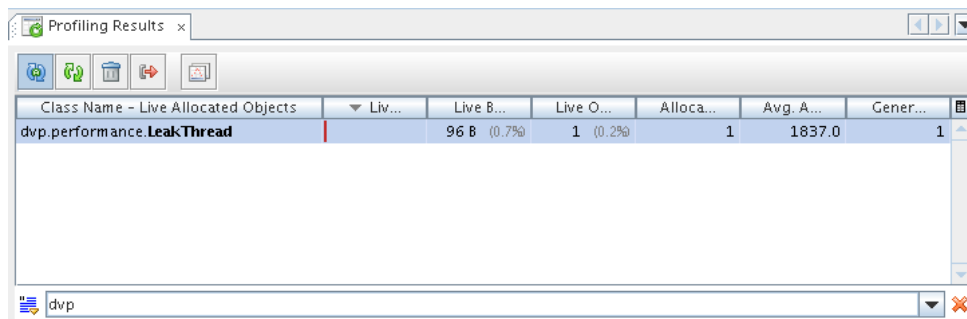
Enfin , encore un petit click droit sur la trace pour accéder au code source.

Vous pouvez sauvegarder un snapshot et le reprendre à tête reposée.

Astuce :

Lorsque vous utiliserez le profileur dans un environnement réel, les classes Java des différentes bibliothèques apparaîtront dans les résultats de profilage. Cela peut être gênant.

La vue *Profiling Results* vous permet de filtrer les classes grâce à la zone de filtre au pied de la vue. Un click sur  affiche un choix d'opérateurs.



Class Name - Live Allocated Objects	▼ Liv...	Live B...	Live O...	Alloca...	Avg. A...	Gener...
dvp.performance. LeakThread		96 B (0.7%)	1 (0.2%)	1	1837.0	1

4 Conclusion

Détecter un problème de mémoire sur une application Java n'est pas une chose aisée.

La plupart des applications réelles utilisent des bibliothèques tierces en plus du JDK : vous devrez prendre garde à votre propre code et à celui des autres.

Heureusement, les fuites mémoires les plus grossières (et leurs contournements) sont souvent connues et recensées dans les bibliothèques populaires.

L'analyse des métriques des outils de profilage suppose une bonne compréhension d'un petit nombre de concepts détaillés dans la première partie du tutoriel.

Ne vous découragez pas : il vous faudra un peu de pratique pour trouver facilement les fuites mémoires et calmer les appétits de vos applications.