Java Performance

Florent Clarret - Equens Worldline

Département Informatique
École Polytechnique de l'Université de Tours

Plan du cours

Organisation de l'enseignement

A propos de moi

Qui suis-je ?

- Florent Clarret
- Ingénieur Étude et Développement chez Worldline, filiale du groupe Atos spécialisée dans le paiement.
- Ancien de Polytech' Tours (promotion 2015) et DUT Informatique (IUT de Lens)
- Contact : florent.clarret@gmail.com

Répartition des heures de cours

- 4h de cours et 20h de TPs

Mode d'évaluation

- Contrôle continu à 100% via des comptes rendus de TPs
  - Découverte des outils de monitoring et d'analyse.
  - Diagnostic et amélioration d'applications.
  - Mise en place de benchmarks.

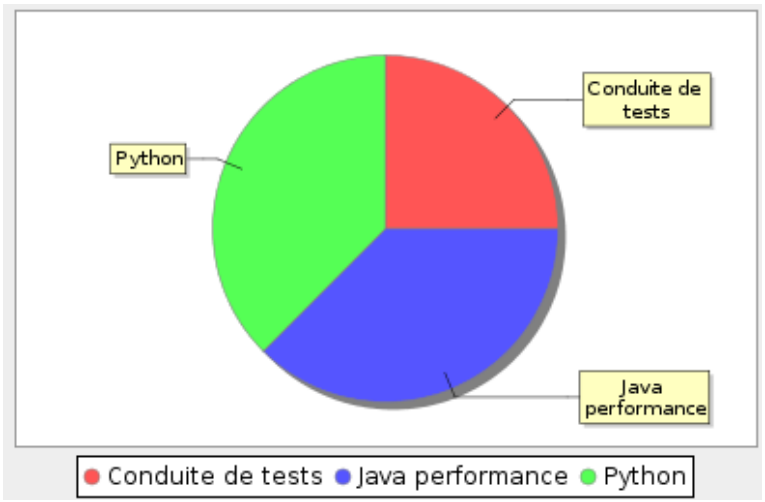


Figure – Poids dans l'UE : Programmation impérative et mise en œuvre

Description des TPs (provisoire)

1. Prise en main des outils de monitoring (4h).
2. Prise en main et mise en place de mini-benchmarks (2h).
3. Résolution de problème de performance sur des exemples simples (2h).
4. Mini-présentation sur le thème "Java Performance" (2h).
5. Étude et amélioration d'une application existante (8h).
6. Découverte de bibliothèques tierces axées sur les performances (2h).

Organisation de l'enseignement

Remarques préalables

Remarques préalables

Sur le cours

- Cours en réécriture permanente
- Basé sur le cours réalisé par Sébastien Aupetit
- Le cours nécessite des bonnes bases en Java

Le périmètre

- Java 8
- Oracle Hotspot JVM
- Système d'exploitation Linux

N'hésitez pas à me faire des retours pour l'améliorer !

Remarques préalables

Au cours des TPs, nous allons utiliser divers outils :

- CLI : tous les TPs sont réalisés sous linux (en l'occurence, une ubuntu)
- Un peu de script shell
- Maven
- Git
- Un IDE (Eclipse/Intellij Idea/Netbeans...) mais c'est facultatif. L'utilisation de vim est autorisée

Les maitriser est un plus, mais ce n'est pas obligatoire dans le cadre de ce cours.

Organisation de l'enseignement

Bibliographie et liens utiles

Bibliographie et liens utiles

- S. Oaks, Java Performance : The Definitive Guide, O'Reilly, Sebastopol, CA, ISSN :978-1-449-35845-7, 2014.
- D.R. Nadeau Java tip: How to get CPU, system, and user time for benchmarking
- DZone.com / Java Zone
- Le site officiel d'Oracle
- Effective Java - Joshua Bloch

Introduction

Introduction

Quelques idées reçues :

- Java est un langage lent
- C++ permet de faire plus et de faire mieux que Java
- Etc...

L'objectif du cours est de vous montrer que c'est faux. Pour maximiser les performances d'une application, il faut développer plusieurs qualités fondamentales :

1. Avoir des connaissances solides
2. Être très rigoureux
3. Maitriser les outils
4. Avoir de l'expérience, cela vient avec le temps
5. (Avoir une bonne dose d'intuition)

Introduction

Connaissances nécessaires

- La JVM (Java Virtual Machine) : le réglage de la JVM influence la performance des programmes.
  - C++ == code compilé => bien choisir les flags de compilation
  - PHP => fichier php.ini
  - Java == bytecode indépendant de la machine => compilation native/optimisation à l'exécution, exécution => réglage JVM et GC
- La plateforme Java : exploiter les meilleurs mécanismes lorsque plusieurs solutions existent pour coder

Améliorer les performances

- Réglage de la JVM => rôle d'ingénieur performance
- Choix judicieux d'implémentation => rôle de développeur

Les deux groupes doivent comprendre ce que chacun peut faire gagner en performance.

Pourquoi est-il nécessaire de faire des réglages en Java ?

- Java est compilé en bytecode indépendant de l'OS. Peu d'optimisations sont possibles à ce moment.
- Pour fonctionner, le code a besoin d'être exécuté par la Java Virtual Machine.
- Lors de l'exécution, la JVM peut transformer ce code (ou non) en instruction machine pour l'optimiser en fonction du système.
- L'utilisation de telle ou telle option peut donner des résultats différents. Il est possible de définir finement ce que l'on souhaite faire faire à la JVM.
- La gestion de la mémoire ainsi que du Garbage Collector peut aussi être modifiée.

Comparaison avec le C++

- Le binaire compilé est spécifique à une plateforme.
- Il est possible de réaliser toutes les optimisations lors de la compilation.
- Cela nécessite aussi de bien configurer le compilateur pour maximiser les performances.

Introduction

Comment améliorer les performances d'une application ?

Comment améliorer les performances d'une application ?

Leviers permettant d'influer sur les performances

1. L'implémentation : peu importe les réglages réalisés, améliorer les performances passe avant tout par une meilleure implémentation.
   - Rôle du développeur
2. Une bonne connaissance du fonctionnement de la JVM.
3. Les réglages de la JVM.
   - Rôle de l'ingénieur performance

En général, un code rapide respecte le principe du KISS.

Keep It simple, stupid

Les principes du KISS :

- Plus le code est long, plus il met de temps à s'exécuter.
- Plus le code est long, moins il a de chance de pouvoir être stocké dans le cache du processeur.
- Plus le code effectue d'allocation mémoire, plus le Garbage Collector a de travail.
- Plus le code alloue des objets et les garde actifs, plus le Garbage Collector aura besoin de temps par cycle.
- Plus le nombre de classes est important, plus le programme sera long à démarrer.
- Etc...

Comme le dit un grand philosophe : "Herta, le goût des choses simples"

Exemple de différentes implémentations

Alogorithme : recherche d'une chaîne dans un ensemble de chaînes.

- cas 1 : l'ensemble est un tableau => La recherche parcourt le tableau et à tester l'égalité
  - Optimisation de la JVM : élimination du test des indices, déroulement de boucle...
  - Dans le pire des cas, il faudra parcourir tout le tableau
- cas 2 : l'ensemble est une table de hashage (HashMap) i.e. tableau de liste chaînée
  - Calcul clé de hachage (indice), parcourt de la liste
  - Tableau suffisamment grand et fonction de hachage bien choisi => liste globalement de taille 2
  - Performance meilleure que cas 1 si le nombre de chaînes est grand
- cas 3 : découpe le tableau en morceau et traite chaque morceau en parallèle
  - performance dépend du nombre de threads exécutable en parallèle

Introduction

Que faut-il optimiser ?

Que faut-il optimiser ?

Les points à optimiser dépendent de vos besoins

- Minimiser l'occupation mémoire.
- Améliorer les temps de traitements.
- Minimiser l'utilisation du CPU.
- Réduire la taille des données émises sur le réseau.

En général, il n'est pas possible de gagner sur tous les tableaux. Il faut faire des compromis selon les objectifs que l'on souhaite atteindre.

Les formes d'optimisations

Diagnostique par profiling

- Se concentrer sur les parties du programmes les plus longues

Dégradation des performances

- Se concentrer sur les derniers ajouts/modifications
- L'environnement n'en est probablement pas la cause

Ecriture de code

- Considérer les paramètres les plus probables et optimiser pour ceux-ci
- Ex : cas du logger : il est plus probable que les log soient désactivés qu'activés
- Ex : tri d'un tableau : petit tableau = classique, grand tableau = tri parallèle + fusion triée

Introduction

Quand faut-il optimiser ?

Quand faut-il optimiser ?

L'optimisation doit être avant tout curative

- Si les performances sont satisfaisantes, il est inutile d'optimiser.
- Optimiser prend du temps, cela coûte donc de l'argent et peut comporter des risques.
- Le modèle peut être dégrader lors des optimisations. Cela ne doit être fait que si les performances sont une réelles contraintes, la qualité du modèle est la priorité.

Limite de l'approche curative

- Attention, cela ne signifie par pour autant que l'on peut faire n'importe quoi lors de l'implémentation sous prétexte que les performances ne sont pas un problème.

Exemple de code

Cas que l'on peut trouver souvent dans une application :

```
1 Logger logger = LoggerFactory.getLogger(this.getClass());
2 logger.debug("Method result is : {}", this.heavyMethod());
```

Même si elle paraît étrange au premier abord, voici une solution plus efficace :

```
1 Logger logger = LoggerFactory.getLogger(this.getClass());
2 if(logger.isDebugEnabled()) {
3    logger.debug("Method result is : {}", this.heavyMethod());
4 }
```

Introduction

Faire le bon diagnostique

Faire le bon diagnostique

Mettre en place un diagnostique précis est primordial.

Dans de nombreux cas, l'application interagit avec d'autres systèmes. Dans ce cas, ces interactions peuvent grandement influer sur vos performances.

- Interactions avec une base de données.
- Alimentation via un load balancer (en amont).
- Délégation d'une partie des tâches à un tiers (en aval).
- Utilisation d'entrées/sorties (sollicitation de disques).
- D'autres programmes utilisant qui partagent une même ressource avec le votre (type base de données).

Quelques mises en garde

Optimiser un système peut dégrader les performances

- Augmentation du nombre de requêtes sur une BDD saturée.
  - Problèmes liés à la concurrence, I/O...
- Augmentation du nombre de paquets envoyés sur un réseau saturé.
  - Capacité, collisions...
- Augmentation du nombre de threads sur un CPU trop faiblard.
  - Pas assez de cœurs CPU, changement de contexte...

Il faut alors trouver d'autres solutions : soulager la BDD, améliorer les requêtes réalisées, compresser les données envoyées sur le réseau...

Le changement de matériel peut être une solution, mais il doit être fait en dernier recours.

Les benchmarks

Les benchmarks

L'objectif des benchmarks :

- Ils permettent de mesurer les performances d'une application, d'un module ou d'une sous-partie d'une application.

Les différents types de benchmarks :

- Le micro-benchmark
- Le meso-benchmark
- Le macro-benchmark

Attention cependant :

Pour mesurer les performances d'une partie d'une application ou d'une application complète, il faut prendre des précautions sinon les résultats peuvent être faussés et aboutir à de mauvaises conclusions. Il faut toujours tenir compte de l'environnement du programme lors d'un benchmark.

Les benchmarks

Le micro-benchmark

Le micro-benchmark

L'objectif du micro-benchmark :

- Mesurer les performances d'une petite portion de code.

Quelques exemples :

- StringBuffer vs StringBuilder vs concat
- Synchronized vs non-synchronized
- AtomicInteger vs Integer
- De manière générale : implémentation A vs implémentation B

Il paraît simple ...

- C'est le type de benchmark le plus difficile à réaliser correctement.

Exemple : Suite de Fibonacci

```
1  private double fibImpl1(int n) {
2    if (n < 0) {
3      throw new IllegalArgumentException("Must be > 0");
4    } else if (n == 0) {
5      return 0d;
6    } else if (n == 1) {
7      return 1d;
8    }
9    double d = fibImpl1(n - 2) + fibImpl1(n - 1);
10   if (Double.isInfinite(d)) {
11     throw new ArithmeticException("Overflow");
12   }
13   return d;
14 }
```

Exemple : Le premier micro-benchmark

```
1  public void fibonacciMicroBenchmark(final int nLoops) {
2    double l;
3    long then = System.currentTimeMillis();
4
5    for (int i = 0; i < nLoops; ++i) {
6      l = fibImpl1(50);
7    }
8    long now = System.currentTimeMillis();
9    System.out.println("Elapsed time : " + (now - then));
10 }
```

Les microbenchmarks doivent utiliser leurs résultats

Le compilateur Java est intelligent

- Le résultat de **fibImpl1** n'est jamais utilisé
- => l'appel est donc supprimé
- => la boucle **for** est supprimé

Ce qui est testé ici :

- Le temps entre les deux appels de **System.currentTimeMillis()**

Solution

- Il faut que le résultat de **fibImpl1** soit lu
- => transformer la variable **l** en variable d'instance avec le mot clé **volatile**

Bencher des méthodes avec paramètres I

Le compilateur peut être très intelligent

- Le calcul de **nLoops** fois **fibImpl1(50)** peut être remplacé par 1 seul calcul + utilisation de la valeur
- Il est peu probable que la méthode soit toujours appelée avec la valeur 50 comme paramètre
  - Sinon, la meilleure optimisation possible est de remplacer par le résultat directement

Pour comparer deux implémentations

- Utiliser différentes valeurs d'entrée

```
1
2 for (int i = 0; i < nLoops; ++i) {
3   l = fibImpl1(random.nextInt());
4 }
```

Bencher des méthodes avec paramètres II

Mais...

- Temps de génération des nombres aléatoires est inclus dans la mesure
- Il faut que les valeurs d'entrée soient les mêmes pour comparer, sinon les tests ne sont pas identiques

Solution

- Précalculer l'ensemble des valeurs des paramètres avant de lancer le test
- Utiliser ces valeurs lors de chacun des tests

Bencher des méthodes avec paramètres III

```
1  public void fibonacciMicroBenchmark(final int nLoops) {
2    Random random = new Random();
3    int[] input = new int[nLoops];
4    for (int i = 0; i < nLoops; ++i)
5      input[i] = random.nextInt();
6
7    long then = System.currentTimeMillis();
8    double l;
9    for (int i = 0; i < nLoops; ++i) {
10     try {
11       l = fibImpl1(input[i]);
12     } catch (IllegalArgumentException ex) { }
13   }
14   long now = System.currentTimeMillis();
15   System.out.println("Elapsed time : " + (now - then));
16 }
```

Bencher des méthodes avec paramètres IV

Microbenchmark avec les bonnes valeurs d'entrées I

Remarques

- La valeur est calculable ssi $n \in [0 : 1476]$
- Ajouter un test de n au début rend automatiquement une implémentation plus rapide sans réelle amélioration

```
1  private double fibImpl1(int n) {
2    if (n < 0)
3      throw new IllegalArgumentException("Must be > 0");
4    else if (n > 1476)
5      throw new IllegalArgumentException("Must be <= 1476");
6    else if (n == 0)
7      return 0d;
8    else if (n == 1)
9      return 1d;
10   return fibImpl1(n - 2) + fibImpl1(n - 1);
```

Microbenchmark avec les bonnes valeurs d'entrées II

```
11  }
```

On optimise les performances dans le cas général !

- Quels sont les valeurs d'une utilisation réelle ?
- Seule les valeurs réelles sont pertinentes (ex : $n \in [0 : 100]$)

```
1  public void fibonacciMicroBenchmark(final int nLoops) {
2    Random random = new Random(1); int[] input = new int[nLoops];
3    for (int i = 0; i < nLoops; ++i)
4      input[i] = random.nextInt(100);
5    long then = System.currentTimeMillis(); double l;
6    for (int i = 0; i < nLoops; ++i) {
7        l = fibImpl1(input[i]);
8    }
```

Microbenchmark avec les bonnes valeurs d'entrées III

```
9     System.out.println("Elapsed time : " +
          (System.currentTimeMillis() - then));
10  }
```

Autres remarques I

Warm up

- Plus un code est exécuté, plus il est optimisé par la JVM (JIT compiler)
- Il faut prévoir une période de warm up
- Sans warm-up : la mesure se fait sur la compilation/optimisation du bytecode

Le mot-clef volatile

- Le recours au mot clé **volatile** introduit un biais qui peut être non négligeable

Autres remarques II

Optimisation du bytecode

- Dépend de l'environnement, du profile d'exécution, de la profondeur de pile, des types, des classes, des sous-classes des données manipulées...
- Dans un programme réel, il y a peu de chance que les appels à la fonction soient en rafales : répartition dans le temps

Conclusion

- Les performances obtenues peuvent être très différentes dans un programme réel
- Se focaliser sur un microbenchmark ne permet en général pas d'améliorer significativement les performances d'un programme

Autres remarques III

Si toutefois ...

- Il peut parfois être utile de réaliser ce type de test
- Pour cela, il existe des framework permettant de simplifier l'écriture de ce type de test
  - Par exemple JMH
  - Nous aborderons ce framework dans le cadre d'un TP

Les benchmarks

Le macro-benchmark

Le macro-benchmark I

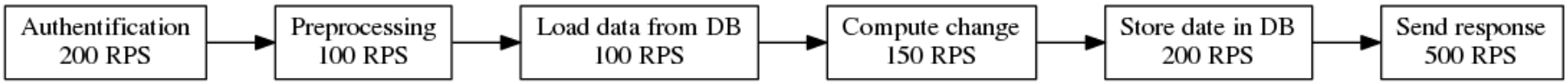
Objectifs

- Prendre en compte l'application dans son intégralité et dans son environnement
- Peut de précautions à prendre pour obtenir des informations fiables

Le macro-benchmark II

Analyse d'une application

- Identification de blocs/flux
- Les performances sont insuffisantes



Analyse par les développeurs de compute change

- Mise en place de mocks (composant simulant) pour être indépendant de la base de données
- Étape limitée à 150 RPS
- Décision d'améliorer les performances de cette partie :
  - Cout important en heures/jours
  - Etape capable de traiter 250 RPS

Le macro-benchmark III

L'application n'a obtenu aucune amélioration de performances !

- Capacité de traitement limité par le maillon le plus faible : 100 RPS
- Il aurait fallu concentrer les efforts sur le preprocessing et l'accès à la BD
- Analyser l'application dans son intégralité et dans son environnement est essentiel !

Les benchmarks

Le meso-benchmark

Mesobenchmark I

Objectifs

- Benchmark intermédiaire
- Mesure d'une portion de l'application ne se réduisant pas à quelques lignes de code (microbenchmarks) mais impliquant une quantité importante de code sans pour autant inclure la totalité des traitements normalement inclues dans l'application

Exemple

Performances d'accès à une page web (ex : JSP+traitement) mais en oblitérant la partie authentification ou la partie gestion de session.

Mesobenchmark II

Propriétés

- Moins sensibles aux erreurs d'interprétation que des microbenchmarks
- Plus sensibles aux erreurs d'interprétation que des macrobenchmarks
- Plus facile à tester que des macrobenchmarks
- Très largement utilisés

Attention

- Les parties omises peuvent avoir beaucoup d'influence (cf. macrobenchmark)
- Le contexte d'exécution (stack depth...) influence les optimisations de la JVM

Les benchmarks

Mesurer les performances

Mesurer le temps de traitement I

Mesurer le temps moyen

- Difficile de mesurer le temps avec précision si temps est court
- On utilise du batching pour augmenter la précision
- JIT compiler optimise le code => prévoir une phase de warm up
- Les données externes peuvent introduire un biais : JPA, IO... => cache application/système et remplissage lors du warm up

Mise en œuvre

- On exécute le traitement par batch (lot)
- On l'exécute X fois
- On calcule le temps moyen d'un traitement (temps du batch / X)

Mesurer le temps de traitement II

Mesure du temps du début à la fin du programme

- Incompatible avec le warm up
- Inclus le temps d'optimisation par le JIT compiler
- Premiers traitements plus lents que les derniers
- Les temps initiaux peuvent être ceux qui ont le plus d'importance
- Pas pertinent sur les micro-benchmarks

Mise en œuvre

- On sauvegarde le timestamp avant et après le traitement

Mesure de débit I

Mesure de débit

- Mesure de la quantité de traitements qui peuvent être effectués par unité de temps
- Requêtes, transactions, opérations par second (RPS/TPS/OPS)

Cas client/server

- 1 ou plusieurs clients font des requêtes au serveur
- Dès réception de la réponse, une nouvelle requête doit être faite
- => Il faut que le client soit le plus rapide possible
- => à défaut, c'est le débit du client qui est mesuré

La mesure de débit est un comportement stable

- Il faut prévoir une phase de warm up

Mesure du temps de réponse I

Mesure du temps de réponse

- Mesurer le temps écoulé entre l'envoi d'une requête et la réception d'une réponse
- Il faut refléter le comportement réel des requêtes
  - Client du type think-time
  - Il faut prévoir un temps d'analyse de la réponse (ex : consultation du contenu de la page web)
- Le débit à peu d'importance
- La mesure s'effectue en moyenne et/ou au centile

Décomposition du traitement

1. Envoi de la requête
2. Traitement par le client
3. Réception de la réponse

Mesure du temps de réponse II

Mesure par la moyenne

- Somme des temps de réponse divisée par le nombre de requêtes
- C'est une méthode très simple...
- Mais peu cacher des disparités très importantes dans les temps de réponse

Les disparités

- Un ou plusieurs traitement du lot peuvent être plus longs
- Ces traitements ne sortiront pas clairement dans le temps moyen
- Ces traitements cachent peut-être le vrai problème
- Solution simpliste : on peut aussi sauvegarder le maximum et le minimum

Mesure du temps de réponse III

Mesure au centile/percentile

- Mesure statistique consistant à diviser l'ensemble des mesures en 100 paquets
- Le centile à x% est le maximum du temps de réponse sur les x% plus petites valeurs de l'échantillon
- Les centiles courants sont 90%, 95%, 99%
- Peu sensible aux données aberrantes (outliers)
- Ex : 20 mesures dont 19 mesures à 1s et 1 mesure à 100s
  - Moyenne : 5.95s. Centile à 90% : 1s.
- Pour le diagnostic, on s'intéresse en priorité aux cas extrêmes

Idéal

- Utiliser tous les outils à notre disposition
- Considérer la moyenne et le(s) centile(s)

Variabilité de la mesure I

Les temps d'un traitement peuvent varier

- Dépend de la plateforme (architecture, dimensionnement...)
- De manière générale : de l'environnement d'exécution

Pour comparer fiablement des mesures

Il est parfois nécessaire de passer par d'autres tests dits statistiques => (Test de Student notamment)

Cycle de développement et mesure de performance

Mesure de performance

- Dans l'idéal, fait partie intégrante du cycle de développement

Règles générales à appliquer

- Tests automatiques donc scriptés : mise en place d'environnements, conduite, remonté des résultats, rapport d'analyse
- Enregistrer tout ce qui est utile : sampling CPU, RAM, threads dump, IO usage, network usage, GC logs..., informations externes associées
- Tests en conditions réelles
- Réaliser au minimum des mesobenchmarks
- Tests réguliers mais non systématique à chaque commit

Problématiques

- Pas toujours évident à mettre en place
- Coût de mise en place et de maintient

Remarques

La phase de warm-up

- Selon le cas d'usage, elle doit être réalisée ou non
- Cette phase peut induire des optimisations qui n'apparaitront que dans un bench (mise en cache, génération du code machine)
- Sans elle, les temps des premiers appels seront toujours plus longs que les temps des derniers

Sur l'utilité des benchs

- Une méthode qui n'est appelée que rarement n'aura pas un bench par lot représentatif. En production, la JVM ne lui appliquera que peu d'optimisations.
- Quelque soit le type de bench, il faut être au plus près des conditions normales de fonctionnement

Les outils de mesures

Les outils de mesures

Les outils de mesures

- De nombreux outils sont disponibles
- Nous nous limitrons à JVM Oracle Hotspot sous GNU/Linux
- Automatisation des mesures => outils en ligne de commande majoritairement => permet une analyse ultérieure
- Il y a aussi de nombreux outils graphiques

D'autres outils existent...

- Parfois payants
- Peuvent être très efficaces. Exemple : DynaTrace
  - Se connecte directement à la JVM à tester
  - Peut tester plusieurs applications en même temps
  - Nombreuses métriques, requêtes BDD effectuées, messaging (ex : interactions avec des ActiveMQ), temps par méthodes...
- Nécessite parfois une architecture lourde

Les outils de mesures

CPU monitoring

CPU monitoring I

Temps CPU

- temps en espace utilisateur (user time) : temps passé à exécuter le code de l'application
- temps en espace noyau (system time) : temps passé à exécuter le code du noyau (I/O, network...)

vmstat -w 1

- Mesure moyenne sur 1s, 5s, 30s
- CPU 50% sur 10 min vs CPU 100% sur 5 min
- Objectif : utiliser 100% sur la période la plus courte possible

CPU monitoring II

```
1 procs ---------------memory-------------- ---swap-- -----io---- -system--
      ------cpu-----
2  r  b     swpd     free     buff    cache   si   so    bi    bo   in   cs us sy id wa st
3  0  0        0 11085396   522172 1475392    0    0     0    92 1119 4283 2  1 97  0  0
4  0  0        0 11084736   522172 1475460    0    0     0     0  955 1994 2  0 98  0  0
5  2  1        0 11083920   522180 1476444    0    0   144    24 1882 32090 10 2 88 0  0
6  1  1        0 10968900   522924 1563284    0    0 87800     4 3944 72934 18 4 75 3  0
7  0  0        0 10886428   523300 1628908    0    0 12004 1040 1711 7682 7  1 90  2  0
8  0  0        0 10885972   523300 1629608    0    0     0     0  994 2976 1  0 98  0  0
9  0  0        0 10883840   523444 1631136    0    0   140   132 1267 5507 3  1 96  0  0
```

CPU monitoring III

Votre CPU ne fait rien ?

- L'application est bloquée sur une primitive de synchronisation (mutex, lock, semaphore...)
- L'application est en attente de quelques choses (données disponibles sur un socket, retour de la base de données...)
- L'application n'a rien à faire

=> On doit pouvoir faire mieux si ce n'est pas voulu

Traitement à la demande (server web...)

- CPU max le temps du traitement
- CPU min en attente
- CPU à 100% pendant 250ms, puis à 0% pendant 750ms => CPU à 25% en 1s
- Attention : temps de capture vs temps des actions

CPU monitoring IV

Augmenter l'utilisation du CPU sur CPU multi-coeurs

- Avoir des threads avec des traitements à faire
- Ne pas surcharger le CPU

```
1 procs ---------------memory-------------- ---swap-- -----io---- -system--
      ------cpu-----
2  r  b     swpd     free     buff    cache   si   so    bi    bo   in   cs us sy id wa st
3  3  0        0  9489292   533864 2054224    0    0  5976     0 4480 10172 54 5 40 1  0
4 11  1        0  9439056   534116 2079852    0    0  9048    12 8749 19488 71 4 24 1  0
5  9  4        0  9362680   534236 2121864    0    0 13900   856 6789 17150 72 4 19 5  0
6 12  2        0  9332696   535032 2130388    0    0  8288   892 16771 13894 81 4 9  6  0
7 10  1        0  9277808   536068 2159040    0    0 13704   160 7404 24937 83 4 11 3  0
8  5  0        0  9216752   536120 2164180    0    0  5056    24 9562 22662 72 3 23 2  0
```

Les outils de mesures

Disk usage

Disk usage I

iostat -xm 5

- Diagnosique des entrées/sorties sur disques

```
avg-cpu: %user   %nice %system %iowait %steal %idle
         11,50    0,03    0,53    1,61    0,00  86,33

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz await
     r_await w_await svctm %util
sda               0,00     7,20   20,00    0,60     0,18     0,04    21,36     0,00    0,12
     0,12    0,00   0,12   0,24
sdb               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00
     0,00    0,00   0,00   0,00
sdc               0,00    63,20   27,60   39,80     0,14     0,48    18,71     3,09   45,80
     6,14   73,31   2,05  13,84
```

Disk usage II

Analyse fortement dépendante du comportement de l'application

- cf. exemple sur le site web
- Problèmes principaux :
  - Beaucoup d'E/S mais débit faible
  - Peu d'E/S mais %util élevé
- Solutions principales :
  - Ajout de buffers
  - Regroupement des informations
  - Upgrade du matériel

E/S causée par l'utilisation de la mémoire virtuelle

- Mémoire virtuelle vs mémoire physique
- Utilisation normale, marche bien en général
- Cas programme Java : GC
- Diagnostique : colonne si/so de wmstat

Les outils de mesures

Network usage

Network usage I

nicstat 5

Diagnostique du traffic réseau

```
1    Time      Int   rKB/s   wKB/s   rPk/s   wPk/s    rAvs    wAvs %Util   Sat
2  16:28:45    eth0  16.62    1.81   16.59   10.57  1026.2   175.2  0.14  0.00
3  16:28:45   wlan0   4.09    0.13   39.64    0.77   105.6   174.8  0.00  0.00
4  16:28:45      lo   0.35    0.35    1.33    1.33   271.6   271.6  0.00  0.00
5  16:28:45  lxcbr0   0.00    0.01    0.00    0.07    0.00   107.9  0.00  0.00
```

Principes

- Similaire aux E/S
- Détecter la sur ou la sous utilisation du réseau
- Indicateurs les plus importants : %util et Sat

Network usage II

Problèmes

- Impossible d'atteindre 100% d'utilisation
- Dès 40% les problèmes apparaissent (dépend du type de matériel)
- Impossible de modifier cette limite via l'application

Solutions

- Upgrade matériel
- Optimiser l'utilisation du réseau (données plus compactes, étalement...)

Les outils de mesures

Les outils Java de monitoring

Les outils Java de monitoring

Principaux outils

- jps : liste les processus Java
- jcmd : permet d'obtenir diverses informations à partir du numéro de processus Java
- jconsole : outil graphique pour suivre l'activité d'un programme
- jhat : analyse des sauvegardes de la mémoire (heap dump)
- jmap : sauvegarde de la mémoire (heap dump) et obtention d'informations sur la mémoire d'une programme Java
- jinfo : permet de modifier des flags d'un processus
- jstack : affichage les piles d'appel d'un processus Java
- jstat : affiche des informations sur le Garbage Collector et le chargemnt des classes
- jvisualvm : outil graphique de monitoring, de profiling et d'analyse
- Java Mission Control : outil graphique de monitoring, de profiling et d'analyse

Les outils de mesures

Profling

Profiling

Profilers

- Fonctionnement classique : joint (via un socket) la JVM exécutant l'application puis échange des informations avec celle-ci
- Profilers sujet à des problèmes de performances (rare, mais ça arrive) : mémoire insuffisante, GC parallélisé...

Deux types de profilers

- Les profilers par échantillonnage (sampling profiler)
- Les profilers par instrumentation (intrumented profiler)
- Peuvent détecter des problèmes différents => on en utilise plusieurs en général

Les outils de mesures

Profling

Sampling profiler

Sampling profiler I

Propriétés

- Peu d'impact sur la charge CPU du système
- Peu de modification des performances et du comportement de l'application par les mesures
- Comptabilise périodiquement les méthodes en cours d'exécution (via call stack thread dumps)
- Sur timer, les fonctions actives sont considérées actives sur la période (1s en général)

Sampling profiler II

Inconvénients

- Risque de faux diagnostique
- Exécution alternative de deux méthodes : A pendant 750ms, B pendant 250ms
  - Si le timer se déclenche uniquement pendant que B s'exécute : diagnostique = B utilise beaucoup le CPU
  - Si le timer se déclenche uniquement pendant que A s'exécute : diagnostique = A utilise beaucoup le CPU

Remarques

- Chaque profiler utilise une stratégie de sampling différent => diagnostiques différents
- Bien profiler :
  - Profiler sur longue période
  - Varier l'intervalle d'échantillonnage

Sampling profiler III

call stack thread dump

- Ne peut pas s'obtenir n'importe quand
- uniquement lorsqu'un thread est à un safe-point i.e. lors d'une allocation mémoire
- Une méthode n'effectuant aucune allocation mémoire ne sera jamais vue !

Optimiser les performances

- Ne pas se concentrer sur les méthodes les plus consommatrices de CPU
  - Gain très faible (en général)
  - Risque de faux diagnostique
- Se concentrer sur la partie de l'application dont font partie ces méthodes
  - Changer la logique plutôt que quelques lignes de code
  - Gain important (en général)

Les outils de mesures

Profling

Instrumented profilers

Instrumented profilers

Propriétés

- Modifie le bytecode des classes lors de leurs chargements afin d'ajouter diverses opérations de mesure (comptabilisation du nombre d'appel, durée de l'appel...)
- Mesure précisément les appels des méthodes

Inconvénients

- Modifie le comportement de l'application et son optimisation par le JIT compiler (inlining...)
- Met en évidence des méthodes différentes d'un sampling profiler
- Surestimation des temps d'exécution des méthodes

Optimisation de l'application

- Identification des méthodes potentiellement les plus consommatrices de temps CPU
- Réduire le nombre d'appel aux méthodes

Méthodes bloquantes et threads

En fonction du profiler et de sa configuration

- Les méthodes bloquantes peuvent ressortir comme problématiques
- Ces méthodes sont bloquées et ne s'exécutent pas => pas de temps CPU

Mais si en attente et ...

- Que c'est normal (ex : serveur web),, alors pas de problème
- Sinon, il y a un problème ...
  - Lock contention
  - Données produites/consommées à un rythme insuffisant
  - E/S lentes
  - ...

  Souvent plus visible sur les graphiques des threads

Les outils de mesures

Mesurer les temps

Les outils de mesures

Mesurer les temps

Mesurer les temps réels

Mesurer les temps réels I

Temps réels

- Temps perçu par l'utilisateur
- Dépendant des autres activités du système

Attention

- Sous windows, la priorité est ajusté si l'application est au premier plan ou ddnon

Remarques

- Il faut moyenner sur plusieurs exécutions
- Recherche de problèmes de performances plus complexes

Mesurer les temps réels II

En millisecondes

```
1 long start = System.currentTimeMillis( );
2 // do something
3 long end = System.currentTimeMillis( );
4 long elapsed =end - start;
```

En nanosecondes

```
1 long start = System.nanoTime( );
2 // do something
3 long end = System.nanoTime( );
4 long elapsed = end - start;
```

Les outils de mesures

Mesurer les temps

Mesurer les temps CPU

Temps CPU, Système et Utilisateur avec un seul thread

Mesurer le temps CPU (CPU time), Système (System time) et Utilisateur (User time)

- A partir de Java 5
- **ManagementFactory.getThreadMXBean()** donne une instance de **ThreadMXBean**
- **getCurrentThreadCpuTime()** : mesure temps CPU du thread
- **getCurrentThreadUserTime()** : mesure temps en espace utilisateur du thread
- Temps en espace système = temps CPU - temps en espace utilisateur

Remarques

- Mesure en nanosecondes
- Peut ne pas être supporté par la JVM

Temps CPU, Système et Utilisateur de n'importe quel thread

A partie du numéro de thread

- **threadMXBean.getThreadCpuTime(id) ;**
- **threadMXBean.getThreadUserTime(id) ;**

Les outils de mesures

Mesurer les temps

Les temps de l'application

Les temps de l'application I

Avec des classes et méthodes standards

- Sommer les temps de tous les threads
- **ThreadMXBean.getAllThreadIds() ;**
- Des threads peuvent disparaitre donc ne pas être comptabilisé
- Une solution : faire des relevés périodiques (cf. site web, D. Robert Nadeau)

Les temps de l'application II

Avec une classe interne de Sun

```
1  import java.lang.management.ManagementFactory;
2  import java.lang.management.OperatingSystemMXBean;
3
4  public class ApplicationTimes {
5    public long getJVMCpuTime() {
6      OperatingSystemMXBean bean =
           ManagementFactory.getOperatingSystemMXBean();
7      if (!(bean instanceof
           com.sun.management.OperatingSystemMXBean)) {
8        return 0L;
9      }
10     return ((com.sun.management.OperatingSystemMXBean)
           bean).getProcessCpuTime();
11   }
12 }
```

Mesure du temps et précision

Mesure du temps

- S'exprime en nanosecondes (1 000 000 000ème de seconde)
- La précision réelle n'est pas en ns
- Le matériel, l'OS et la JVM n'offre aucune garantie
- Précision classique : quelques milliers de ns jusqu'à plusieurs ms
- Mesure en ms souvent suffisante + batching

Just in time compiler

JIT compiler I

Langages interprétés

- Les instructions sont interprétées une par une
- Indépendant de l'architecture matérielle
- Performances faibles

Langages compilés

- Exécution d'instructions machine obtenues par compilation du code source
- Flags + compilation = dépendance à une architecture matérielle
- Flags "moyens" pour un socle commun pour réduire le nombre de binaire
- Performances bonnes mais pas optimales

JIT compiler II

Approche hybride

- Code source compilé en un code intermédiaire indépendant de la machine
- Traduction/compilation/optimisation du code lors de l'exécution
- Indépendant de l'architecture matérielle
- Performances bonnes (sur moyen/long terme), capable d'exploiter les spécificités du matériel

Java

- Approche hybride
- Code intermédiaire = bytecode
- JIT Compiler

HotSpots et Just In Time compilation I

JIT Compiler d'Oracle s'appuit sur la notion de HotSpot

- Un HotSpot (ou point chaud) est une méthode (ou portion de méthode) exécutée un grand nombre de fois en un court lapse de temps.

Principes généraux pour l'exécution

- Le bytecode est interprété : s'il n'est pas déjà compilé en instruction machine et si ce n'est pas un HotSpot
- Le bytecode est compilé en code machine dès que possible s'il est détecté comme un HotSpot
- Le code machine est exécuté si le bytecode est déjà compilé (donc disponible).

HotSpots et Just In Time compilation II

Détection des HotSpot

- Comptabilisation du nombre d'appels des méthodes ou d'exécution des itérations des boucles
- Décrémentation des compteurs de façon périodique
- HotSpot = compteur élevé

La compilation du bytecode n'est pas systématique

- La compilation prend du temps : interpréter est plus rapide si le nombre d'utilisation est faible
- Pour optimiser, le JIT compiler a besoin d'information : profile feedback receuilli lors de l'interprétation

HotSpots et Just In Time compilation III

Exemple de profile feedback

- Une méthode admet en paramètre un objet du type **Object**
- L'appel de la méthode **toString()** sur l'objet implique une recherche de l'implémentation réelle de la méthode (table des méthodes dynamiques)
- Cette méthode est toujours appelé avec des objets du type **X** et **X** redéfinit **toString()**
- L'appel à **toString()** peut être simplifié en un appel direct à l'implémentation de **X** tant que cette hypothèse est vrai
- Le coût de recherche de l'implémentation est éliminé
- Si l'hypothèse devient fausse, la méthode rebasculera en mode interprété puis compilée selon le même principe

HotSpots et Just In Time compilation

Les compilateurs

Les compilateurs I

3 types de JIT compiler

- Client (C1) : destiné au poste client mono-coeur
- Server (C2) : destiné au serveur multi-coeur
- Tiered (à étage) : à partie de Java 7

Différence de destination moins importante de nos jours mais performances différentes

Mode client

- Compilation du bytecode plus tôt qu'en mode server
- Démarrage du programme plus rapide (premières requêtes. . .)
- Code machine sous optimal car il manque un profile feedback

Les compilateurs II

Mode server

- Compilation dy bytecode uniquement lorsque la collecte d'information est suffisante et HotSpot
- Démarrage du programme plus lent que le mode client
- Code machine plus rapide que mode client

Mode tiered

- Hybride entre les deux modes
- Démarrage rapide
- Code machine optimal sur le long terme
- Principes :
  - Compilation en mode client,
  - Identification des HotSpot et collecte des profile feedbacks
  - Recompilation en mode server des hotSpots
- Par défaut en mode server depuis Java 8

Les compilateurs III

Exemple de temps de démarrage

Type d'application	-client	-server	-server -XX :+TieredCompilation
HelloWorld	0.08 s	0.08 s	0.08 s
Netbeans	2.83 s	3.92 s	3.07 s
BigApp	51.5 s	54.0 s	52.0 s

Les compilateurs IV

Exemple de temps d'exécution d'une série de requêtes depuis le démarrage de l'application

Nombre de raquêtes	-client	-server	-server -XX :+TieredCompilation
1	0.142 s	0.176 s	0.165 s
10	0.211 s	0.348 s	0.226 s
100	0.454 s	0.674 s	0.472 s
1000	2.556 s	2.158 s	1.910 s
10000	23.78 s	14.03 s	13.56 s

- Tiered plus rapide car du code non HotSpot a été compilé en mode client
- Long terme : tiered identique ou plus rapide que mode server seul

32 bits ou 64 bits I

Les compilateurs C1 et C2 existent en 3 version

- C1 : 32 bits uniquement
- C2 : 32 bits ou 64 bits

JVM installable

- Machine 32 bits : JVM 32 bits (C1 ou C2 32 bits)
- Machine 64 bits : JVM 32 bits (C1 ou C2 32 bits), JVM 64 bits (C2 64 bits)

32 bits ou 64 bits II

Performances

- JVM 32 bits plus rapide qu'une JVM 64 bits (pointeurs plus petits...)
- JVM 32 bits limités à 4Go de mémoire tout compris (JVM + heap + stack + ...)
- Calcul intensif sur long/double plus rapide avec JVM 64 bits : exploite instructions spécifiques du CPU
- JVM 64 bits : utilisation des pointeurs compressés

Mode réel en fonction de la version de la JVM

Version installée	-client	-server	-d64	Tiered server par défaut ?
JVM Linux 32 bits	32b client	32b server	Erreur	
JVM Linux 64 bits	64b server	64b server	64b server	oui
JVM Max OS X	64b server	64b server	64b server	oui
JVM Windows 32 bits	32b client	32b server	Erreur	
JVM Windows 64 bits	64b server	64b server	64b server	oui

32 bits ou 64 bits III

Autoconfiguration de la JVM

Système d'exploitation	Mode par défaut
Linux 32 bits + 1 CPU	client
Linux 32 bits + 2 ou plus CPU	-server
Linux 64 bits	-server
Mac OS X	-server
Windows 32 bits	-client
Windows 64 bits	-server

Connaître la version utilisée

```
1 [florentclarret@imac-de-florent-clarret:~] : java -version
2 java version "1.8.0_121"
3 Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
4 Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)
```

Les paramètres de la JVM

Les paramètres -X

- Ce sont des paramètres non-standards
- Ils ne sont pas forcément implémentés sur toutes les JVM
- Ils ne sont pas forcément implémentés de la même façon
- Ils peuvent disparaitre d'une version à une autre

La liste des paramètres est longues

- Options qui agissent sur le comportement
- Options pour le garbage collector
- Options plus typées "performance"
- Options de debug
- => Consulter le site d'oracle pour en avoir la liste

Code cache

Code cache

- Zone mémoire où est stocké le code machine après compilation

Performances

- Si le code cache est plein : plus aucun code ne peut être compilé => interprété
- Problème plus fréquent en mode client ou en mode tiered : plus de compilation
- Par défaut depuis Java 8 :
  - Mode client taille fixée à 32 Mo
  - Mode server taille fixée à 240 Mo
- Taille réglable mais fixée au démarrage du programme
- Mode 32 bits : compte dans la limite des 4Go du processus
- Mode 64 bits : peu de risque de spécifier plus grand que nécessaire

Seuil de compilation du bytecode I

Utilisation de compteurs

- Compte le nombre d'appel d'une méthode
- Compte le nombre d'itération d'une boucle
- Décrémentation régulière des compteurs

Si la somme des compteurs dépasse le seuil

- Ajout du bytecode dans la file d'attente du code à compiler
- Lorsque le code est disponible, il remplace l'interprétation du bytecode

Seuil de compilation du bytecode II

Cas d'une boucle longue

- Méthode détectée comme un Hotspot
  - Mise en attente de compilation
  - Lors du prochain appel, si la version compilée existe, elle est utilisée
- Si la boucle est longue, ce n'est pas efficace :
  - JVM détecte que la boucle est longue
  - Le bytecode de la boucle est mis en attente de compilation
  - Dès que le code est disponible, à la prochaine itération, il y a remplacement par le code compilé
  - On-Stack Replacement (OSR)

Remarque

- Seuil de compilation réglable
- Peu d'effet en pratique

Inspecter le processus de compilation I

Argument -XX+PrintCompilation

```
timestamp compilation_id attributes (tiered_level) method_name size deopt
```

- Timestamp : le temps de la fin de la compilation exprimé depuis le démarrage de la JVM
- Compilation_id : identifiant de la tache de compilation (évolue de façon croissante).
- Attributes :
  - % : compilation d'un OSR (On-Stack Replacement)
  - s : la méthode est synchronisée
  - ! : la méthode à un gestionnaire d'exception
  - b : la compilation a bloqué les autres compilations (normalement impossible sur le JVM récente)
  - n : compilation de code permettant l'exécution de méthode native.

Inspecter le processus de compilation II

Argument -XX+PrintCompilation

```
timestamp compilation_id attributes (tiered_level) method_name size deopt
```

- tiered_level : le niveau de compilation en mode tiered ou rien si le mode tiered n'est pas activé
- method_name : le nom de la méthode au format className : :method
- size : la taille du bytecode compilé
- deopt : message indiquant s'il y a eu desoptimisation du code

Desoptimisation I

Desoptimisation

- Décision de la JVM de rebasculer un code machine vers sa version bytecode interprétée
- Apparait dans les logs : Not entrant code

Desoptimisation II

Causes

- Hypothèses d'optimisation ne sont plus vraies (ex : X.toString())
  - desoptimisation => baisse des performances
  - si c'est toujours un HotSpot => compilation avec les nouvelles hypothèses
  - Baisse de performances temporaires
- En mode tiered
  - Compilation en mode client => code machine sous optimal
  - Détection de HotSpot => compilation en mode server avec le profile feedback
  - Remplacement du code :
    - Ancien code marqué non entrant
    - Remplacement OSR

Desoptimisation III

Code non entrant

- Code qui ne peut plus être utilisé
- Élimination du code cache : Zombie code

Tiered compilation levels I

Niveaux d'exécution du code

- 0 : code interprété
- 1 : C1 simple
- 2 : C1 limité (information de profile non utilisée)
- 3 : C1 complet (information de profile utilisée)
- 4 : C2 (HotSpot)

Fonctionnement normal

- Méthode commence au niveau 0
- Compilation au niveau 3 (client complet)
- Détection en HotSpot : compilation au niveau 4 (server) puis niveau 3 marqué non entrant

Tiered compilation levels II

Si la file d'attente du niveau 4 (server) est pleine

- Compilation au niveau 2
- Avec profile feed-back : compilation niveau 3, puis 4 si besoin

Si la file d'attente client est pleine

- Compilation au niveau 2 (rapide)
- puis requalification au niveau 4

Méthodes triviales et erreur de compilation

- Compilation au niveau 1

Performances

- Si nombreuses compilations au niveau 2 et qu'il reste du temps CPU alors il faut augmenter le nombre de threads de compilation.

HotSpots et Just In Time compilation

Remarques supplémentaires

Quelques optimisations du JIT compiler I

Soit la classe Point suivante :

```
1 public class Point {
2   private int x, y;
3   public int getX() { return x; }
4   public void setX(int v) { x = v; }
5 }
```

Inlining avant :

```
1 public void f() {
2   Point p = getPoint();
3   p.setX(p.getX() * 2);
4 }
```

Quelques optimisations du JIT compiler II

Inlining après :

```
1 public void f() {
2   Point p = getPoint();
3   p.x = p.x * 2;
4 }
```

Remarques sur les méthodes courtes :

- Ne pas avoir peur des getters/setters, des méthodes courtes

Quelques optimisations du JIT compiler III

Escape analysis : simplifie le code en fonction du contexte

```
1  public class Factorial {
2    private BigInteger factorial;
3    private int n;
4    public Factorial(int n) { this.n = n;}
5    public synchronized BigInteger getFactorial() {
6      if (factorial == null)
7        factorial = ...;
8      return factorial
9    }
10 }
11 public void f() {
12   ArrayList<BigInteger> list = new ArrayList<>();
13   for (int i = 0; i < 100; ++i) {
14     Factorial factorial = new Factorial(i);
15     list.add(factorial.getFactorial());
16   }
17 }
```

Mot clé final

Ancienne croyance

- **final** permettrait de gagner en performance
- Faux avec JDK 1.5+
- A utiliser si besoin dans le code, mais forcément pas pour la performance

Toutefois

- J'ai tendance à l'utiliser souvent
- Permet de "verrouiller" les pointeurs

Garbage collection

Garbage collection

Garbage Collector

- Libération automatique de la mémoire inutile
- Pas de GC universel, les algo appraissent/disparaissent
- On s'intéresse aux GC disponibles en Java 8+

Opérations de Garbage Collection

- Identification des objets non accessibles
- Libération de la mémoire des objets non accessibles
- Compactage de la mémoire

Chaque GC réalise ces opérations différemment avec différentes performances

Les pauses stop-the-world

Détection des instances non accessibles

- Utilisation de compteurs de référence => pas suffisant (ex : liste double chaînée)
- Utilisation d'algorithmes plus complexes => les références ne doivent pas changer pendant la détection
- Blocage de tous les threads de la JVM = pause stop-the-world

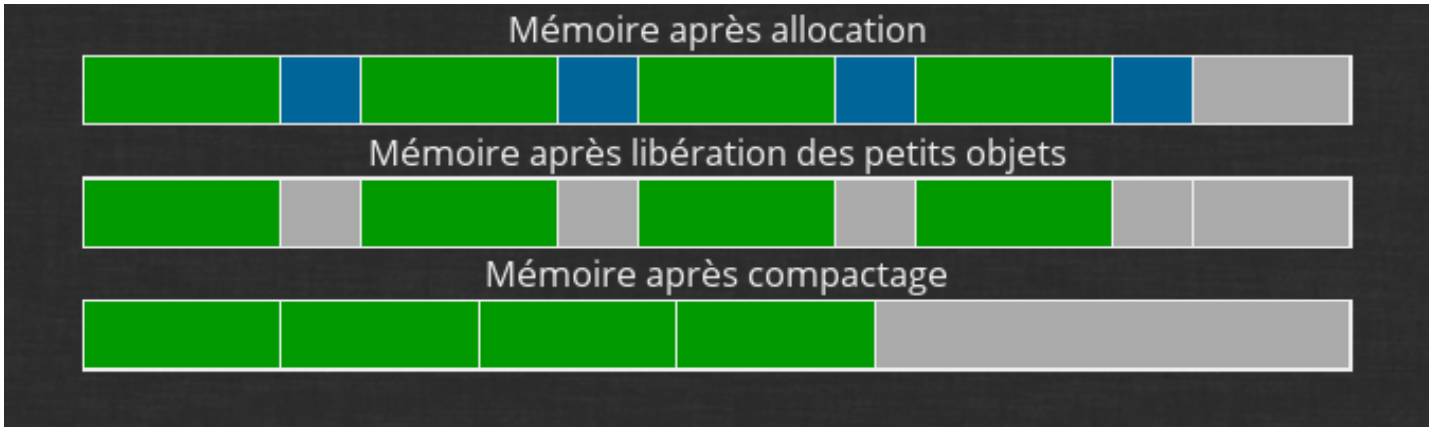
GC

- Cherche à minimiser cette pause
- Différentes stratégies : moment, durée
- Dépendant des conditions d'exécution : nombre CPU, taille mémoire...

GC compactant

GC Compactant

- Objectif : compacter la mémoire pour éviter la fragmentation



Propriétés

- Pas de problème d'allocation mémoire si mémoire disponible mais fragmentée
- Nécessité de copier/déplacer les objets en mémoire => pause stop-the-world

Tous les GC sont générationnels I

Mémoire divisée en deux zones

- Young generation :
  - Par défaut
  - Les nouvelles instances sont créées dans cette zone
  - Si leur durée de vie est courte, ils seront libérées en étant encore dans cette zone
- old generation :
  - Zone de stockage des objets en mémoire depuis "longtemps"

Tous les GC sont générationnels II

Performances

- En Java (fréquent) : création d'objets à durée de vie courte et très courte
- Ces objets sont dans la young generation
- Si la young generation est pleine, le GC se déclenche
  - Uniquement sur la young generation
  - Objets inaccessibles => libérés
  - Objets encore accessibles => déplacés ailleurs
- Gc sur la young generation = minor GC
  - Pause stop-the-world très courte
  - Pauses plus fréquentes
  - Compactage implicite (objets déplacés ailleurs)

Tous les GC sont générationnels III

Structure de la young generation

- 3 parties : eden + surviror 1 + survivor 2
- Nouvelle instance => eden
- Instances encore accessibles (minor GC) => survivor à tour de rôle si possible

Déclanchement d'un minor GC

- Lorsque l'eden est pleine
- GC sur eden + surviror actif
- Déplacement des instances encore accessibles :
  - Dans l'autre survivor : objets "jeunes"
  - Dans la old generation : objets "trop vieux" ou survivor plein
- Fin du minor GC : eden vide donc compacté implicitement

Tous les GC sont générationnels IV

Déclanchement d'un full GC

- Olg generation pleine
- GC sur toute la mémoire
- Etape stop-the-world beaucoup plus longue qu'un minor GC

Performance : GC bien choisi, programme bien conçu, mémoire taille adaptée

- Minor GC : plus ou moins fréquent mais court
- Full GC : rare

Garbage collection

Les différents algorithmes de Garbage Collection

Les différents algorithmes de Garbage Collection I

Serial garbage collector

- Le plus simple
- Mono-threadé
- Pendant un minor GC ou un full GC, il bloque tous les threads (stop-the-world pause)
- Pendant un full GC, la old generation est compactée entièrement
- Par défaut sur machine à 1 CPU ou sous Windows 32 bits

Les différents algorithmes de Garbage Collection II

Throughput collector

- Utilise plusieurs threads pour collecter la young generation
- Utilise plusieurs threads pour collecter la old generation
- Pendant un minor GC ou un full GC, il bloque tous les threads (stop-the-world pause)
- Les pauses sont plus courtes qu'un serial GC en raison de la parallélisation
- Par défaut sur machine Unix multi CPU ou JVM 64 bits

Les différents algorithmes de Garbage Collection III

CMS Collector (Concurrent Mark Sweep)

- Objectif : éliminer les longues pauses d'un full GC (low-pause collector)
- Utilise plusieurs threads pour collecter la young generation
- Pendant un minor GC, il bloque tous les threads (stop-the-world)
- Full GC remplacé par 1 ou plusieurs threads (background threads) :
  - Balayent la old generation pour libérer les objets inaccessibles
  - Pauses stop-the-world très courtes
  - Temps total des pauses stop-the-world < throughput collector
  - Inconvénients :
    - Background threads utilisent une partie des ressources CPU
    - Il faut donc que du temps CPU soit disponible pour que ça marche
    - Pas de compactage de la old generation => fragmentation
  - Pas assez de temps CPU ou mémoire trop fragmentée pour une allocation :
    - Basculement vers un serial GC
    - Tous les threads sont stopés (stop-the-world)
    - Young + Old generation analysées, libérées et compactées avec un seul thread
    - Rebasculement vers leCMS collector

Déclenchement forcé d'un Full GC

Full GC manuel

- En Java : **System.gc() ;**
- En ligne de commande : **jcmd process_id GC.run**
- Via GUI : jconsole, jvisualvm...

Une mauvaise idée sauf

- Dans un benchmark, après le warm up et avant la mesure
- Lors d'un dump mémoire, permet de réduire l'analyse au contenu utile
- RMI effectue un Full GC périodiquement pour gérer le GC distribué

Garbage collection

Réglages principaux

Taille de la mémoire I

Performance

- Performance du GC dépend de la taille de la mémoire
- Pas assez de mémoire => GC trop fréquent => moins de CPU pour l'application
- Trop de mémoire => Collecte et pauses plus longues, pauses moins fréquentes
- Taille mémoire > mémoire physique disponible => GC+SWAP ou pire GC multithread+SWAP

Taille de la mémoire II

Dimensionnement automatique

OS et JVM	Initial heap	Maximum heap
Linux 32 bits client	16Mo	256Mo
Linux 32 bits server	64Mo	min(1Go,1/4 mémoire physique)
Linux 64 bits server	min(512Mo,1/64 mémoire physique)	min(32Go,1/4 mémoire physique)
Mac OS X 64 bits server	64Mo	min(1Go,1/4 mémoire physique)
Windows 32 bits client	16Mo	256Mo
Windows 64 bits server	64Mo	min(1Go,1/4 mémoire physique)

Règles de dimensionnement

- Ne jamais dépasser la quantité de mémoire physique (tout compris et toutes JVM comprises)
- Trop de Full GC :
  - Mémoire insuffisante
  - Règle empirique : après un Full GC, la mémoire doit être occupée à 30% (application à l'état stable)

Dimensionnement des young et old generations

Dimensionnement par défaut

- Young = 1/3 de la mémoire
- Ratio réglable
- Young et Old s'agrandissent automatiquement

Metaspace I

Metaspace (Java 8+)

- Zone mémoire interne à la JVM pour conserver un certain nombre d'informations sur les classes chargées
- Pas de taille max par défaut
- Taille min : 12Mo, 16Mo ou 20.75Mo

Performances

- JVM 32 bits : compte dans la limite des 4Go : problème avec ClassLoader leak
- Agrandissement de la metaspace => Full GC
- Augmenter la taille initiale si provoque plusieurs Full GC au démarrage

Nombre de threads du GC I

Mode automatique : N nombre de CPU

- N si $N \leq 8$
- $8 + ((N - 8) * 5/8)$ si $N > 8$

Mode manuel

- **-XX :ParallelGCThreads=N**
- Très utile si plusieurs JVM sur même machine : surcharge lors des phases de GC

Adaptive sizing

Adaptive sizing

- JVM adapte automatiquement la taille des mémoires en fonction des besoins (Young, Eden, Survisor, Old, Metaspace...)
- Au moment des GC
- Utilisation des informations collectées lors des précédents GC

Performances

- Utilise uniquement la mémoire dont elle a besoin en fonction des objectifs de performance

Thread local allocation buffers I

Thread local allocation buffer

- Région dans l'eden
- 1 TLAB par thread (dédié)
- Permet de faire des allocations sans synchronisation de l'accès
- Transparent pour le développeur
- Dimensionnement du TLAB dépend des caractéristiques d'allocation du thread

Thread local allocation buffers II

Principe : lors d'une allocation par un thread

- Allocation dans le TLAB dédié
- TLAB plein :
  - Déclasser le TLAB, dédier un autre TLAB au thread et réaliser l'affectation => le TLAB déclassé est considéré plein même si ce n'est pas le cas
  - Ou allouer la mémoire dans la old generation avec synchronisation de l'accès
- Allocation de grands objets :
  - TLAB de 100ko
  - 75ko utilisés
  - Allocation de 30ko
  - Si déclassement alors :
    - Déclassement du TLAB
    - 25ko d'eden perdu
    - Déclenchement prématuré d'un minor GC, voire d'un full GC
  - Problème : si ça se répète

Thread local allocation buffers III

Que faire ?

- Régler la taille initiale des TLAB et/ou Régler l'autoadaptation de la taille des TLAB => complexe
- Monitorer les allocations hors TLAB, déterminer l'origine, essayer d'allouer de plus petits objets

Garbage collection

Monitoring des performances du GC

Monitoring des performances du GC I

Activation des logs du GC :

- Rajouter l'option à la JVM :
  - -verbose :gc ou **-XX :+PrintGC**
- Pour avoir plus d'infos :
  - **-XX :+PrintGCDetails**
- Pour horodater les logs :
  - Temps relatif au démarrage de la JVM : **-XX :+PrintGCTimeStamps**
  - Date exacte : **-XX :+PrintGCDateStamps**

Outils d'analyse du GC :

- En mode GUI :
  - jconsol ou jvisualvm
- En ligne de commande :
  - jstat

Monitoring des performances du GC II

Jstat : Usage

```
jstat -gcutil process_id interval
```

Jstat : Exemple

```
[florentclarret@imac-de-florent-clarret:~] : jstat -gcutil 1132 1000
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
 22,88   0,00   4,26  43,81  98,36  96,53    304    4,552    70    2,354    6,905
 22,88   0,00  38,46  43,81  98,36  96,53    304    4,552    70    2,354    6,905
 22,88   0,00  71,83  43,81  98,36  96,53    304    4,552    70    2,354    6,905
 69,32   0,00  11,68  43,78  98,36  96,53    306    4,567    71    2,391    6,959
  0,00 100,00  73,78  43,37  98,36  96,53    307    4,577    71    2,391    6,969
  0,00  89,72  38,82  44,12  98,37  96,53    309    4,600    72    2,394    6,994
  0,00  84,04  15,15  44,81  98,37  96,53    311    4,621    72    2,394    7,014
  0,00  92,06   7,26  45,39  98,37  96,53    313    4,642    72    2,394    7,036
100,00   0,00  79,75  42,66  98,37  96,53    314    4,652    73    2,431    7,083
  0,00  98,65  11,52  43,40  98,38  96,53    317    4,680    74    2,433    7,113
```

La mémoire

Shallow, Retained, Deep size

Les différentes size rencontrées

- Shallow size : c'est la taille en octets d'un objet en mémoire
- Deep size : c'est la taille en octets d'un objet et des objets qu'il référence
- Retained size : correspond à deep size mais en retirant les objets référencés par d'autres objets (donc partagés)

Il est important de confronter ces différentes tailles

Améliorer les performances du GC

- Il faut réduire son utilisation ou de mieux l'utiliser
- Un objet avec une retained size élevée maintient de nombreux objets en vie
- Les outils d'analyses peuvent mettre en évidence ces liens et donc remonter au objets racines

Point sur l'Out Of Memory

Il arrive pour plusieurs raisons :

- Il n'y a plus de mémoire native (ex : limite des 4Go atteinte, mémoire de l'OS pleine...)
- Metaspace plein (spécification d'une taille max trop petite, ClassLoader leak)
- Il n'y a plus de mémoire Java (taille max trop petite, memory leak (ex :collection de plus en plus grande...))
- Le coût du GC est trop important (GC prend trop de CPU sans réussir à récupérer de la mémoire)

La mémoire

Amélioration de l'utilisation de la mémoire

Amélioration de l'utilisation de la mémoire

Plusieurs stratégies possibles

- Réduction de la taille des instances
- Lazy initialization
- Eager deinitialization
- Immutable and canonical objets
- String interning

Les principaux effets visibles sont sur le comportement du GC :

- Moins de minor GC,
- Maintient prolongé d'instances dans les survivors avant basculement vers la old generation
- Moins de full GC ou de concurrent GC
- Les minor et full GC libéreront plus de mémoire à chaque fois

Réduction de la taille des instances I

Réduire la taille des objets

- Deux manières de réduire la taille :
  1. Réduire le nombre d'attributs
  2. Réduire la taille mémoire des attributs

La taille des objets

- La taille d'une objet est la somme de :
  - La taille des attributs
  - La taille de l'en-tête (8 octets pour JVM 32bits, 16 pour les JVM 64bits)
  - Les octets de paddings pour atteindre une taille multiple de 8
- La taille d'un tableau est la sommes de :
  - La taille de ces éléments
  - Une en-tête de 16 octets (JVM 32bits ou moins de 32 Go de mémoire) ou 24 octets

Réduction de la taille des instances II

Taille des types primitifs en Java

Type	Taille en octets	Type	Taille en octets
byte	1	**float**	4
char	2	**long**	8
short	2	**double**	8
int	4		

Taille des références en Java

- JVM 32 bits : 4 octets
- JVM 64 bits et moins de 32 Go de mémoire : 4 octets
- JVM 64 bits avec plus de 32 Go de mémoire : 8 octets

Réduction de la taille des instances III

Impact de la réduction/augmentation de la taille d'un objet

- Réduction de la taille :
  - Possible nécessité de devoir recalculer la même valeur, donc un coût en terme de CPU.
- Augmentation de la taille :
  - Le GC se déclenche plus souvent, donc un coût aussi en terme de CPU.

Conclusion

- Il faut peser le pour et le contre à chaque fois.
- Le résultat de la méthode *hashCode* de la classe *String* est mis en cache dans une variable.

Lazy initialization I

Principe

- Initialiser un objet uniquement lorsque l'on s'en sert (à sa première utilisation)

Exemple : Singleton sans lazy initialization

```
1 public class Singleton {
2    private Singleton() {}
3
4    private static Singleton INSTANCE = new Singleton();
5
6    public static Singleton getInstance() {
7       return INSTANCE;
8    }
9 }
```

Lazy initialization II

Exemple : Singleton avec lazy initialization

```
1  public class Singleton {
2     private Singleton() {}
3
4     private static Singleton INSTANCE = null;
5
6     public static Singleton getInstance() {
7        if (INSTANCE == null)
8           INSTANCE = new Singleton();
9        return INSTANCE;
10    }
11 }
```

Eager deinitialization

Principe

- Consiste à mettre une référence à null afin de faciliter le travail du Garbage Collector

=> **Dans la majorité des cas, c'est inutile**

Exception : la classe ArrayList

- Le tableau interne n'est pas redimensionné automatiquement
- La méthode remove force la dernière référence du tableau à null
- Dans le cas contraire, des objets zombis ne serait pas ramassés par le garbage collector

Threading et synchronization

Threading et synchronization I

Principe et contexte

- Apparition de CPU multi-cœurs
- L'idée est de réaliser des tâches en parallèle
  - Soit pour aller plus vite à réaliser un même traitement
  - Soit pour réaliser plus de traitements
- Cela induit que le traitement peut être découpé en différentes tâches plus petite

Exemple de programme parallélisé

- Une application server
  - Sans thread, il ne pourrait traiter qu'une seule requête à la fois
  - Les clients devraient alors attendre leur tour
- Un traitement sur une grande quantité de données
  - On peut assigner une partie des données à chacun des threads

Threading et synchronization II

Parallélisation en Java

- Possibilité de créer des processus légers => threads
- Ils partagent le même espace mémoire
  - **Attention aux problèmes de concurrence**
- La création d'un thread entraîne un surcout
- Pas de processus lourd en java => "fork"

Mise en œuvre pour une application server I

Création systématique de threads

- Principe : Créer un thread à chaque nouvel appel d'un client
- Avantages :
  - Pas de thread en attente pour rien si aucun appel n'est réalisé
  - Ressources disponibles pour d'autres processus
- Inconvénients :
  - La création d'un thread est processus couteux (temps, ressources)
  - Le temps de traitement de la requête augmente

Mise en œuvre pour une application server II

Utilisation d'un pool de threads

- Principe :
  - Créer un ensemble de thread au départ, et les réutiliser
  - On peut fixer un nombre minimum et maximum de threads
- Avantages :
  - Pas de surcout à chaque appel client lié à la création
  - Maîtrise du nombre de threads
- Inconvénients :
  - Les threads sont créés au lancement de l'application, entrainant un surcout à ce moment
  - Il faut bien calibrer la taille du pool (min et max)

Mise en œuvre pour une application server III

Fonctionnement d'un pool de threads

- Création du nombre minimum de threads
- Lorsqu'un appel est fait, on utilise un thread disponible, s'ils sont tous occupés, on en crée un nouveau dans le pool
  - Jusqu'à la taille maximum
  - Ce fonctionnement peut varier selon d'autres paramètres
- Après un certain temps d'inactivité, on détruit les threads pour revenir à la valeur minimale
  - De même, ce comportement n'est pas systématique

Pool de threads I

La taille minimum

- Elle peut varier selon les besoins
- Une taille minimum élevée permet d'être plus réactif si beaucoup d'appels sont réalisés
- Cependant l'empreinte mémoire du programme est plus élevée
- => Pas de solution miracle, il faut faire des tests

La taille maximum

- Varie selon les conditions d'exécution, vos besoins la machine...
- Un trop grand nombre de threads peut mener à une perte de performance
  - Les synchronisations bloquent les threads
  - La machine n'a pas assez de CPU
  - Un service tiers ne tient pas la charge
- => Pas de solution miracle non plus, il faut faire des tests

Pool de threads II

Exemple de surcharge du nombre de threads

Table – Tests sur une machine 4 cœurs pour le même traitement

Number of threads	Seconds required	Pourcentage of baseline
1	255,6	100
2	134,8	52,7
4	77,0	30,1
8	81,7	31,9
15	85,6	33,5

Remarques

- Le gain n'est ni linéaire ni proportionnel
- L'augmentation du nombre de threads peut ralentir le traitement

Synchronization

Pourquoi synchronisez les threads ?

- Les threads partagent le même espace mémoire
- Ils peuvent modifier/lire les mêmes données
- Cela peut mener à des problèmes de concurrence
  - => Inconsistance des données traitées

Synchronisation en Java

- Passe par le mot-clef synchronize pour sécuriser l'accès à une zone de code
- Ce mécanisme entraine un surcout à l'utilisation => pose et vérification des locks
- Si tout est synchronisé, le multithreading n'a plus d'intérêt car un seul thread pourra travailler
- Il faut l'utiliser uniquement si nécessaire

Monitorer les threads en Java

Monitorer les threads en Java

- jconsole permet de suivre l'évolution des threads d'un programme
- Java Mission Control
- jstack et jcmd peuvent aussi être utiles

Java Performance

Florent Clarret - Equens Worldline

Département Informatique
École Polytechnique de l'Université de Tours