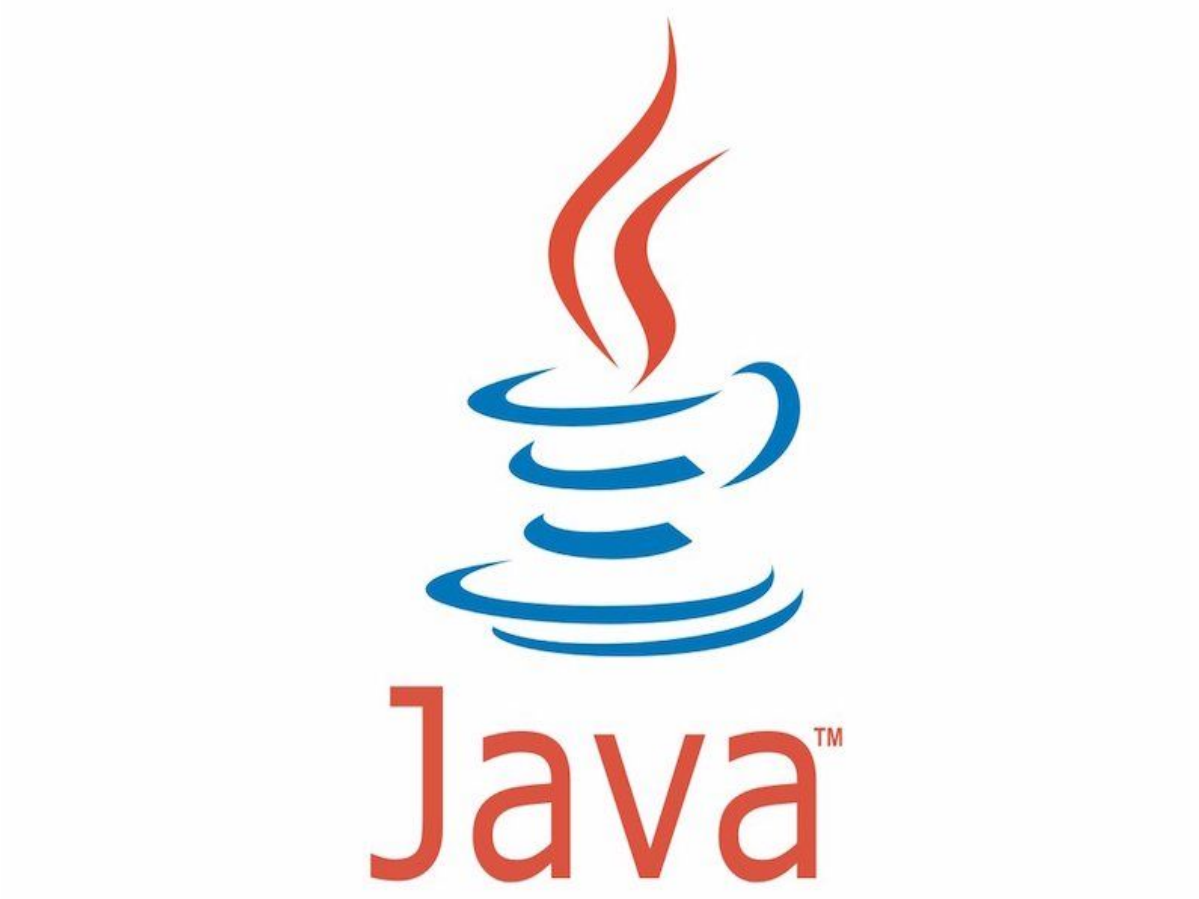
TP3

On heap VS Off heap

Cognard Cédric et Van-Elsue Paul



Sommaire

1. Introduction historique

Dans tout programme, la heap (en français le tas) correspond à la mémoire du programme qui contient diverses informations, et en particulier les variables mémoires ainsi que les objets pour les langages orientés objets.

Seulement la taille maximale allouée à la JVM a longtemps été problématique à cause des différentes architectures. Que ce soit à cause du 32 bits, à cause des différences entre les différents systèmes d'exploitation ou encore à cause du comportement du Garbage Collector, il fallait toujours trouver un compromis entre la mémoire allouée au heap pour pouvoir garder en mémoire nos objets, et la mémoire du Garbage Collector pour lui permettre d'agir assez rapidement et de ne pas provoquer de ralentissements.

Ce que nous appellerons On Heap est donc cette mémoire, la plus classique. C'est tout ce qui est sur le tas. Mais pour régler ces problèmes, une fonctionnalité spécifique au Java est la possibilité de gérer des objets en dehors de la tas. Ce n'est pas le seul langage à le faire mais par exemple le C et le C++ ont une utilisation de la mémoire plus classique.

```
Object myObject = new Object()
```

Allocation d'un objet sur le tas (On Heap), utilisé dans tout programme Java.

2. Apparition de la Off Heap

Pour régler les problèmes vu précédemment, Java a donc intégré la mémoire dite "Off Heap".

Cette mémoire est, contrairement à ce qui est plus classique, en dehors du tas. Elle fournit certains avantages supplémentaires telles que :

1. Une meilleur évolutivité des tailles de mémoire;
2. Un impact sur les temps de pauses du Garbage Collector;
3. Un partage de la mémoire entre les processus évitant le dédoublement des informations entre les JVMs;
4. Une persistance pour les redémarrages entraînant certaines réponses plus rapides (comme par exemples pour les données de tests).

Le plus gros problème de cette mémoire vient évidemment de la manière dont elle est stockée. En effet c'est une mémoire qui utilise la sérialisation pour sauvegarder les informations directement sur le disque dur. Les structures deviennent moins naturelles à traiter, on perd un peu de flexibilité qui est avant tout un des points forts du Java. Mais ce problème est plus une gêne qu'une limitation pour le développeur. Par contre il faut faire attention à l'utilisation du Off Heap, car la sérialisation est coûteuse, tout autant que les accès aux disques. C'est donc bien plus lent qu'une utilisation de la mémoire sur le tas. Il faut donc faire attention quant à son utilisation.

Pour être sûrs de bien comprendre ce qu'est la Off Heap, voici une petite phrase simple : tout ce qui n'est pas dans la Heap est Off Heap.

3. Les différences apportées

La première différence entre la On Heap et la Off Heap est bien évidement la différence de gestion à apporter de la part du développeur. Sur la On Heap, la JVM limite la quantité de mémoire utilisable, il n'y a pas à se soucier de problèmes telle qu'une allocation mémoire trop grande surchargeant le système, Java se limitera seulement à ce dont il a besoin.

Mais avec la Off Heap, certaines limitations sont levées et le développeur peut plus facilement faire n'importe quoi. C'est en soi une force car on peut utiliser des données bien plus volumineuses qu'avec la mémoire principale mais c'est aussi un problème, on ne veut pas par exemple écrire et réécrire un nombre trop important de fois des données sur un SSD car on pourrait endommager ce dernier. Alors que si on a le choix pour faire de la pagination, il vaudrait mieux le faire sur un SSD qui peut aller jusqu'à 80 000 IOPS plutôt qu'un disque dur lent avec 80 IOPS. Le développeur ne se limite plus à l'utilisation de son propre programme mais doit prêter attention à son intégration dans son environnement.

L'utilisation de la mémoire Off Heap peut donc présenter quelques difficultés, mais elle n'existerait pas sans son lot d'avantages.

Tout d'abord il est bien plus simple à utiliser que la majorité des autres mémoires en dehors du tas. Pas besoin d'écrire les données soi-même dans un fichier, pas besoin de mettre en place une base de donnés, etc…

Mais le gain le plus important est surtout le temps de démarrage. Les systèmes en productions sont plus rapides à démarrer, ré-exécuter le même événement pour des tests est aussi plus rapide à s'effectuer, etc...

Pour conclure nous pourrions dire que les données On Heap sont bien pour des informations moins lourdes, pour une utilisation classique. Ce sont des données plus sujettes au Garbage Collector. Mais si nous traitons des données lourdes et encore plus pour des données ré-utilisées régulièrement, alors les placer en dehors du tas, sur la Off Heap, est une très bonne solution. Cela libérera de la mémoire et offrira donc de meilleures performances mais il faut bien être sûr de comprendre comment fonctionnent ces deux solutions pour savoir laquelle sera la plus efficace à choisir.

4. Comment utiliser la Off Heap

L'utilisation dans un programme Java de mémoire hors heap devra répondre aux problématiques suivantes :

- Accéder à la mémoire hors heap.
- Gestion de la mémoire.
- Référencement et accès à la mémoire off-heap.

Pour cela nous allons utiliser **Java Native Access (JNA)**.
JNA permet l'accès aux librairies natives (dll, so) à partir de code Java uniquement (sans utiliser JNI ou du C).
Une interface java décrit les méthodes et les structures de la librairie native.
Il n'y a pas de génération de code tout se fait au Runtime.
JNA est fourni avec certaines librairies natives courantes ainsi que des classes utilitaires d'accès à la mémoire native.

L'utilisation de la mémoire native avec cette API est simple, voire transparente.
`ByteBuffer.allocateDirect()`

Remarque : Le contenu stocké doit être de type tableau de byte, ce qui explique que le contenu stocké hors heap doit être sérialisable.

Exemples de création d'objets hors heap:

```
int value = 12345;
int offset = 0;
int size = 128;

Memory m = new Memory(1024);// on crée un objet mémory

// on alloue l'espace mémoire
ByteBuffer buff = m.getByteBuffer(0, size);
offset = offset + size;

buff.putInt(value);// On insère l'objet

buff.flip(); // RAZ de la position courante du buffer

System.out.println(buff.getInt());// On récupère et affiche l'objet

ByteBuffer buff2 = m.getByteBuffer(offset, size);// On déclare un deuxième buffer

buff2.putInt(value + 1);// On insère l'objet

buff2.flip();// RAZ de la position courante du buffer

System.out.println(buff2.getInt());// On récupère et affiche l'objet

m.clear();// on libère la mémoire


// méthode 2

Memory m2 = new Memory(1024);// on crée un objet memory et on insère directement les objets

m2.setInt(offset, value);// On insère l'objet

System.out.println(m2.getInt(offset));// On récupère et affiche l'objet

m2.clear();// on libère la mémoire
```

Tout comme pour la heap, l'espace est libéré par le GC lorsque l'objet n'est plus référencé par le code. De plus, il n'y a pas de relation directe entre le moment ou l'objet est libérable et le moment ou il est effectivement libéré. Donc il n'y a pas de magie, les objets hors heap sont bien sensibles au GC.
Toutefois :

- Pas de phase de marquage des objets.
- Pas de phase de compaction (réorganisation de l'espace mémoire) pendant le passage du GC.
- Le nettoyage de la mémoire hors heap est donc plus rapide que son homologue de la heap.

Il est possible d'appeler la méthode de nettoyage à tout moment (encore une fois en fouillant dans les profondeurs de l'API) :

- `Method getCleanerMethod = buffer.getClass().getMethod("cleaner", new Class[0]);`
- `getCleanerMethod.setAccessible(true);`

- `sun.misc.Cleaner cleaner = sun.misc.Cleaner)getCleanerMethod.invoke(buffer, new Object[0]);`
- `cleaner.clean();`

5. Conclusion

Nous avons donc vu comment fonctionnent et comment utiliser les mémoires “On Heap” et “Off Heap”, chacun apportant son lot d’avantages et d’inconvénients. Désormais nous devrions être capables d’optimiser la gestion de notre tas pour obtenir de meilleure performances dans nos programmes Java. Peut-être pourrions nous tenter de comparer ces deux méthodes avec différents types de données dans des benchmarks pour être sûrs de bien savoir quand utiliser quoi…

6. Sources

VanillaJava, On Heap VS Off Heap :
https://vanillajava.blogspot.com/2014/12/on-heap-vs-off-heap-memory-usage.html?fbclid=IwAR2FH1EufzAndASm-WiFIbhKISaX4UreHfCZaiRECHqqx5BD1YkSEIXC1D4

Ippon.fr, utilisation de la Off Heap :
https://blog.ippon.fr/2011/11/03/java-acces-directs-a-la-memoire-off-heap/

StackOverflow, “Difference between ‘on-heap’ and ‘off-heap’ ” :
https://stackoverflow.com/questions/6091615/difference-between-on-heap-and-off-heap