Concurrent collections - TP3

November 13, 2018

Thomas COUCHOUD
thomas.couchoud@etu.univ-tours.fr
Victor COLEAU
victor.coleau@etu.univ-tours.fr

Chapter 1

Introduction

Ce TP a pour but de démystifier certains aspects de Java et de les mettre en lien avec la notion de performances.

Notre sujet porte sur les Concurrent Collections. Les collections sont des éléments antiques (JDK 1.2) de Java, importants et fréquemment utilisés.

Nous allons donc détailler plus en détail ce qu'est une Collections. Puis nous introduirons la notion de concurrence au sein de celles-ci et verrons les outils disponibles afin de la prendre en comtpe.

Chapter 2

Qu'est-ce que c'est ?

2.1 Collection

En Java Collection désigne une interface. Cette dernière a pour but de désigner un groupe d'objets appelés éléments. Cette définition est très générique, ce qui laisse les implémentations variées. Certaines peuvent être ordonnées, d'autres non ; certaines avec duplicatas, d'autres non.

Dans le JDK on ne trouve pas d'implémentation directe de cette interface, seulement des implémentations d'interfaces plus spécifiques telles que List ou Set.

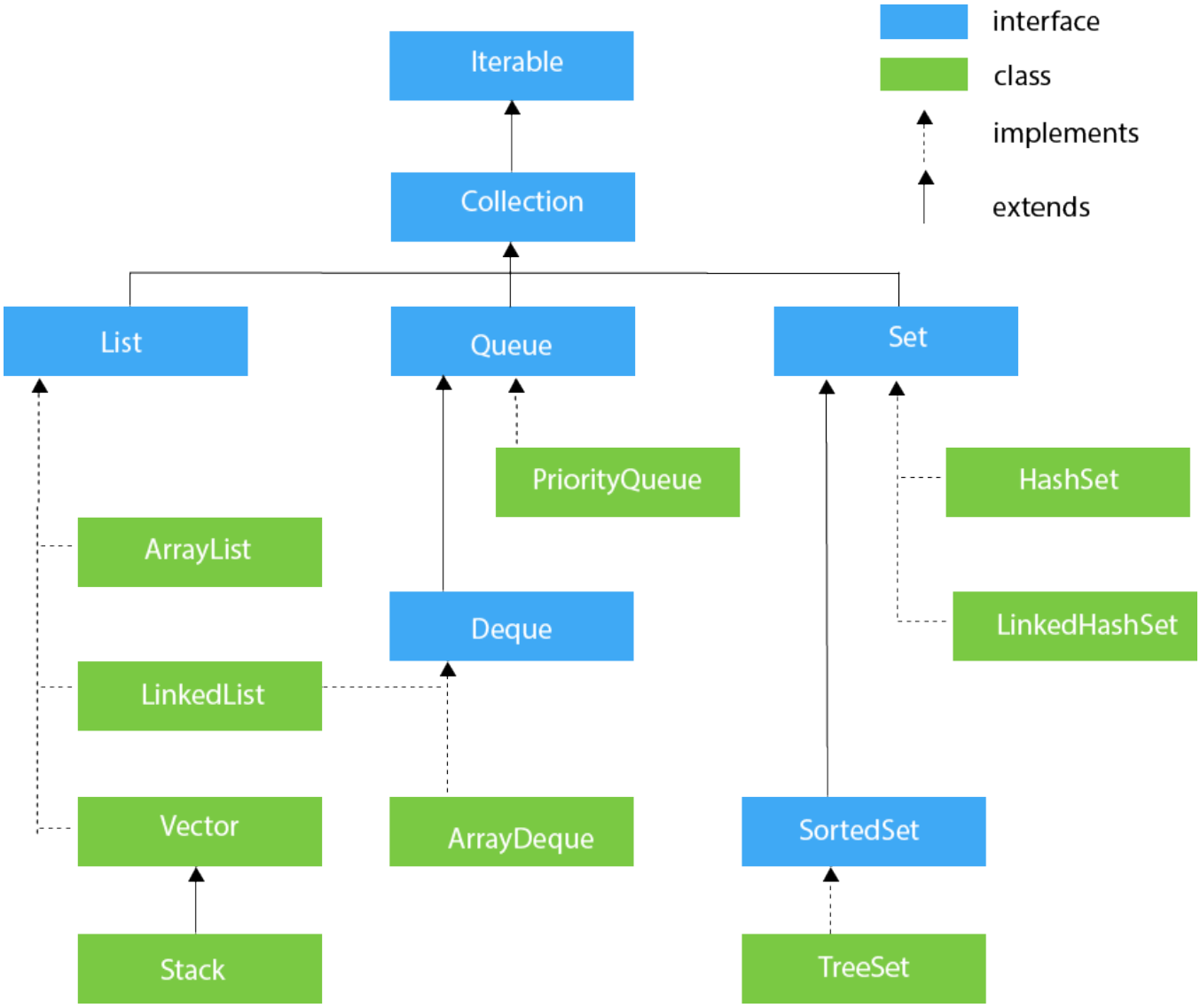

Figure 2.1

Collection définie notamment les méthodes add(element), contains(element) et remove(element). Parlons donc rapidement des 3 grandes extensions de Collection: List, Set, Queue.

- List définie une collection d'objets ordonnée. On peut avoir des éléments dupliqués et il y a du sens de parler de l'élément à l'indice k. De ce fait la méthode remove(index) est aussi définie dans ce cas.

  Exemple: $[1,2,3,4,3,5,4]$

- Queue désigne une queue. Cette dernière n'est pas très différente d'une list: une collection d'éléments ordonnés. Cependant on ne peux que manipuler les objets se trouvant uniquement aux extrémités (on ajoute d'un coté, et on retire d'un autre). Parler de l'élément à l'indice k n'a donc aucun sens, on le verra quand il est en tête de queue.

  On retrouve les fonctions offer(element), poll et peek.

- Set définie une collection d'objets n'ayant pas d'ordre et pas de duplicata.

2.2 Concurrence

Listing 2.1 – Main.java

```
1  import java.util.*;
2  class Main extends Thread {
3      static ArrayList l = new ArrayList();
4      public void run()
5      {
6          try {
7              Thread.sleep(2000);
8          }
9          catch (InterruptedException e) {
10         }
11         l.add("E4");
12     }
13
14     public static void main(String[] args) throws InterruptedException
15     {
16         l.add("E1");
17         l.add("E2");
18         l.add("E3");
19         new Main().start();
20
21         Iterator i = l.iterator();
22         while (i.hasNext()) {
23             String s = (String)i.next();
24             System.out.println(s);
25             Thread.sleep(6000);
26         }
27         System.out.println(l);
28     }
29 }
```

Si on lance le code ci-dessus, nous allons obtenir une exception java.util.ConcurrentModificationException. En effet, nous avons deux threads qui tentent d'accéder à la même liste en même temps (le while et notre add ligne 11).

Afin de pouvoir réaliser ces opérations, deux possibilités s'offrent à nous : soit on gère cela dans notre code pour être sûr qu'une seule opération est réalisée à la fois, soit on utilise des implémentations de collections qui gère la concurrence.

Deux façons de procéder reposent sur Collections.synchronizedXxx() et le package java.util.concurrent.

Chapter 3

Comment ça marche ?

Le but de ces solutions est de créer des objets ThreadSafe. C'est à dire qu'ils peuvent être manipulés par plusieurs Threads parallèles sans provoquer d'erreur.

Il est à noter que certaines classes spécifiques de Java.util sont nativement ThreadSafe telles que Vector et Stack.

3.1 Collections.synchronizedXxx()

Ces méthodes static prennent toutes en paramètres un objet de leur type (un Set pour Collections.synchronizedSet, une List pour Collections.synchronizedList, etc.) et renvoient un objet "synchronisé" de même type.

Toute interaction avec la Collection devra s'effectuer au travers de ce nouvel objet. Il est donc recommandé de supprimer la référence à l'objet "non-synchronisé". De plus, s'il l'on souhaite itérer sur la dite Collection, il faut le faire dans un block *synchronized* avec comme verrou la Collection.

Cette solution, bien que simple à mettre en place, comprend un défaut majeur. En effet, un objet "synchronisé" est un objet dont toutes les méthodes sont synchronisées. Cela signifie qu'à chaque appelle d'une de ses méthodes, tout l'objet est verrouillé, même dans le cas de méthodes non-problèmatiques telles que Get (dans le cas d'une List).

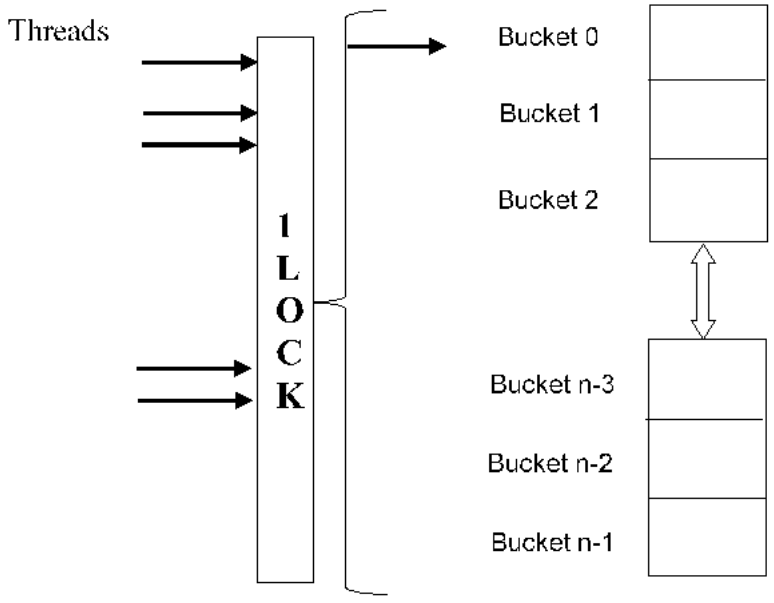

Figure 3.1

De ce fait, l'accès à l'objet est très ralenti car les processus de verrouillage et déverrouillage sont couteux en temps, ce qui provoque un effet de congestion.

3.2 java.util.concurrent

Ce package contient des classes crées spécifiquement pour gérer la concurrence et les problèmes qu'elle soulève. On y trouve par exemple ConcurrentHashMap, CopyOnWriteArrayList.

L'avantage de cette solution par rapport à la précédente est que celle-ci ne bloque pas tout l'objet mais utilise la technique du Lock Striping.

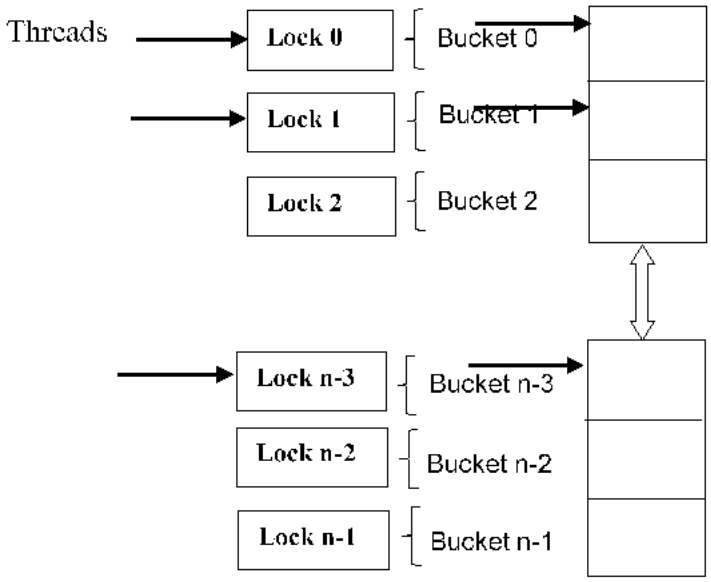

Figure 3.2

Cette technique consiste à verrouiller une portion d'une structure de données où chaque lock verrouille un ensemble d'objets indépendants (appelé Bucket). Dans le cas d'un ConcurrentHashMap on dispose par défaut de 16 Buckets donc de 16 locks différents. Grâce à cela, jusqu'à environ 16 opérations normalement concurrentes peuvent être ici réalisées en parallèle.

A noter que les méthodes Get de chaque objet ne sont pas bloquantes. Cela implique qu'elles peuvent être exécuter en même temps qu'une méthode Update (Add ou Remove). Dans ce cas, le Get renverra la Collection la plus récemment mise à jour (Update fini).

Dans la cas de CopyOnWriteArrayList, une autre technique de synchronisation est employée. Cette dernière consiste à autoriser toutes les opérations de lectures en parallèle tandis que les opérations d'écriture sont réalisées grâce à une copie intégrale de la Collection. La modification est alors effectuée sur la copie, ne génant par les lectures en cours. Dès que cela est possible, l'ancienne Collection est remplacée par la nouvelle (mise à jour).

Chapter 4

Quels sont les impacts sur les performances?

La gestion de la concurrence n'est pas optionnelle dans un programme nécessitant plusieurs Threads. En effet, si rien n'est mis en oeuvre, des exception seront levées ainsi que des états de Collections incohérents.

Le choix d'utilisation des objets "synchronisés" ou des méthodes du package Concurrent se fait sur la taille estimée des Collections et le nombre de Threads accédant à ces Collections.

Dans le cas de petites instances, les objets "synchronisés" peuvent être suffisants car le faible nombre d'éléments et de demandes d'accès en parallèles réduisent l'impacts du bloquage total des objets.

A l'inverse, dans le cas de grandes instances, le bloquage complet de la Collection ralentirait bien trop le programme. Il est donc recommandé d'utiliser les objets du package java.util.concurrent. Ces derniers autorisant l'écriture et la lecture en parallèle ainsi que le bloquage partiel permettent au taux d'accès supérieur.

De plus, certaines classes ayant des implémentations spécifiques (telle que CopyOnWriteArrayList) s'adaptent mieux à certains types de programmes en fonction de ce que l'on souhaite faire.

Chapter 5

Conclusion

Les collections sont présentes dans Java depuis un long moment et font donc partie intégrante du développement. Cependant savoir quelle implémentation utiliser est une tâche plus délicate. En effet chacune à ses propres spécificités et ne sont pertinentes que dans certains cas précis.

La notion de concurrence rajoute une complexité supplémentaire car une méthode d'accès aux données doit être implémentée. Sous sa forme la plus basique (lock d'objet) les performances sont drastiquement diminuées sur des volumes importants. Des alternatives plus sophistiquées sont disponibles et ont une meilleure scalabilité.

Il est donc important de garder à l'esprit que chaque problème à son implémentation qui lui correspond le mieux et qu'aucune solution n'est parfaite.