Algorithmique parallèle et distribuée :
Quelques outils du package *java.util.concurrent*

Julien Rossit

IUT Paris Descartes

Un problème de partage de mémoire

Un problème d'accès concurrent

```
class Counter {
  private int c = 0;

  public void increment() {
      c++;
  }

  public void decrement() {
      c--;
  }

  public int value() {
      return c;
  }
}
```

Détails du problème

Comportement :

1. récupère la valeur de c ;
2. incrémente la valeur récupérée de 1 ;
3. sauvegarde la valeur de retour dans c.

Un scénario catastrophe :

1. le Thread A récupère la valeur de c ;
2. le Thread B récupère la valeur de c ;
3. le Thread A incrémente la valeur récupérée, le résultat est 1 ;
4. le Thread B décrémente la valeur récupérée, le résultat est -1 ;
5. le Thread A sauvegarde la valeur de c, c vaut 1 ;
6. le Thread B sauvegarde la valeur de c, c vaut -1.

Problème

Cette opération n'est pas atomique !

Détails du problème

Comportement :

1. récupère la valeur de c ;
2. incrémente la valeur récupérée de 1 ;
3. sauvegarde la valeur de retour dans c.

Un scénario catastrophe :

1. le Thread A récupère la valeur de c ;
2. le Thread B récupère la valeur de c ;
3. le Thread A incrémente la valeur récupérée, le résultat est 1 ;
4. le Thread B décrémente la valeur récupérée, le résultat est -1 ;
5. le Thread A sauvegarde la valeur de c, c vaut 1 ;
6. le Thread B sauvegarde la valeur de c, c vaut -1.

Problème

Cette opération n'est pas atomique !

Détails du problème

Comportement :

1. récupère la valeur de c ;
2. incrémente la valeur récupérée de 1 ;
3. sauvegarde la valeur de retour dans c.

Un scénario catastrophe :

1. le Thread A récupère la valeur de c ;
2. le Thread B récupère la valeur de c ;
3. le Thread A incrémente la valeur récupérée, le résultat est 1 ;
4. le Thread B décrémente la valeur récupérée, le résultat est -1 ;
5. le Thread A sauvegarde la valeur de c, c vaut 1 ;
6. le Thread B sauvegarde la valeur de c, c vaut -1.

Problème

Cette opération n'est pas atomique !

Les collections non synchronisées

Interface *List<E>* : listes

- *ArrayList<E>* : tableau ;
- *LinkedList<E>* : liste chainée ;
- *CopyOnWriteArrayList<E>* (threadsafe mais modifications coûteuses !).

Interface *Set<E>* : ensembles

- *HashSet<E>* ;
- *TreeSet<E>* ;
- *CopyOnWriteSet<E>* (threadsafe mais modifications très coûteuses !).

Interface *Map<K, V>* : table d'association clef-valeur

- *HashMap<K, V>* ;
- *TreeMap<K, V>*.

Le problème

Comportement :

1. vérifie la capacité du tableau représentant la liste ;
2. redimensionne le tableau si nécessaire ;
3. ajoute l'élément à la dernière position ;
4. incrémente l'index de la dernière position.

Un scénario catastrophe :

1. il reste une place dans le tableau représentant la liste ;
2. le Thread 1 vérifie la capacité de la liste et ne redimensionne pas ;
3. le Thread 2 vérifie la capacité de la liste et ne redimensionne pas ;
4. le Thread 2 ajoute l'élément à la dernière place, incrémente l'index ;
5. le Thread 1 ajoute l'élément à une position incorrecte.

Dans tous ces cas

A protéger avec *synchronized()* !

Le problème

Comportement :

1. vérifie la capacité du tableau représentant la liste ;
2. redimensionne le tableau si nécessaire ;
3. ajoute l'élément à la dernière position ;
4. incrémente l'index de la dernière position.

Un scénario catastrophe :

1. il reste une place dans le tableau représentant la liste ;
2. le Thread 1 vérifie la capacité de la liste et ne redimensionne pas ;
3. le Thread 2 vérifie la capacité de la liste et ne redimensionne pas ;
4. le Thread 2 ajoute l'élément à la dernière place, incrémente l'index ;
5. le Thread 1 ajoute l'élément à une position incorrecte.

Dans tous ces cas

A protéger avec *synchronized()* !

Le problème

Comportement :

1. vérifie la capacité du tableau représentant la liste ;
2. redimensionne le tableau si nécessaire ;
3. ajoute l'élément à la dernière position ;
4. incrémente l'index de la dernière position.

Un scénario catastrophe :

1. il reste une place dans le tableau représentant la liste ;
2. le Thread 1 vérifie la capacité de la liste et ne redimensionne pas ;
3. le Thread 2 vérifie la capacité de la liste et ne redimensionne pas ;
4. le Thread 2 ajoute l'élément à la dernière place, incrémente l'index ;
5. le Thread 1 ajoute l'élément à une position incorrecte.

Dans tous ces cas

A protéger avec *synchronized()* !

Les collections synchronisées

Vector<E> : tableau dynamique

- *boolean add(E elt)* : ajoute en fin de tableau ;
- *boolean add(int i,E elt)* : ajoute en position i du tableau ;
- *E get(int index)* : retourne l'élément en position i du tableau ;
- *boolean remove(E elt)* : retire la première occurrence de *elt*, retourne vrai s'il elle existe ;
- *E remove(int i)* : retire l'élément en position i et le retourne.

Stack<E> extends Vector<E> : pile

- *E peek()* : renvoie le premier élément de la pile sans l'enlever ;
- *E pop()* : dépile le premier élément ;
- *E push(E elt)* : empile et retourne l'élément ajouté.

Hashtable<K,V> : table de hachage

- *V get(Object clef)* : renvoie l'objet indexé par clef ;
- *V put(K clef, V valeur)* : associer valeur à clef dans la table.

L'implémentation de ces objets utilisent des verrous (implicites).

Le package *java.util.concurrent*

Un peu d'histoire

Les méthodes de synchronisation que nous connaissons (objet verrou, *synchronised()*, etc) :

- sont **de bas niveau** ;
- sont efficaces ;
- mais rendent le code peu lisible.

Motivations :

Une synchronisation de haut niveau rend le code :

- plus clair ;
- plus rapide ;
- plus facile à écrire ;
- et donc plus fiable.

le package *concurrent*, développé par Doug Léa (professeur au State University of New York (SUNY) College d'Oswego) est intégré à partir de Java 5

variables atomiques

Variables mises à jour de façon **atomique** :

- *AtomicBoolean* ;
- *AtomicInteger* ;
- *AtomicLong* ;
- *AtomicReference<V>* ;
- etc.

Quelques méthodes communes :

- *E get()* : retourne la valeur ;
- *void set(E elt)* : met à jour la valeur ;
- *boolean compareAndSet(E old, E new)* : compare à une une valeur attendue, puis met à jour ;
- *String toString()* : fournit une représentation de la valeur.
- méthodes d'incrémentation, etc.

Ne remplace pas systématiquement efficacement les types primitifs classiques

L'interface *BlockingQueue<E>*

Une file qui réagit :

- si elle est vide lors de la récupération d'un élément ;
- si elle est pleine lors de l'effilement d'un élément.

Différents comportements sont possibles selon la méthode utilisée :

- lance une exception ;
- retourne une valeur spéciale (*null* ou *false*) ;
- bloque le thread courant indéfiniment jusqu'à ce que l'opération soit possible ;
- bloque le thread courant jusqu'à un timeout.

Les implémentations classiques :

- *ArrayBlockingQueue* : file de taille bornée ;
- *LinkedBlockingQueue* : liste chainée ;
- *PriorityBlockingQueue* : liste à priorité ;
- *SynchronousQueue* : une file de capacité nulle ;
- etc.

L'interface *BlockingQueue<E>*

Une file qui réagit :

- si elle est vide lors de la récupération d'un élément ;
- si elle est pleine lors de l'effilement d'un élément.

Différents comportements sont possibles selon la méthode utilisée :

- lance une exception ;
- retourne une valeur spéciale (*null* ou *false*) ;
- bloque le thread courant indéfiniment jusqu'à ce que l'opération soit possible ;
- bloque le thread courant jusqu'à un timeout.

Les implémentations classiques :

- *ArrayBlockingQueue* : file de taille bornée ;
- *LinkedBlockingQueue* : liste chainée ;
- *PriorityBlockingQueue* : liste à priorité ;
- *SynchronousQueue* : une file de capacité nulle ;
- etc.

Quelques méthodes usuelles

Method Summary :

- *boolean add(E e)* : enfile l'élément si possible, lance une exception et retourne *false* sinon ;
- *boolean offer(E e)* : enfile l'élément si possible et retourne *true*, retourne *false* sinon ;
- *void put(E e)* : enfile l'élément si possible, attend une place disponible si nécéssaire ;
- *boolean remove(Object o)* : retire une instance unique de l'objet, si présente dans la queue ;
- *E poll(long timeout, TimeUnit unit)* : retire et retourne l'élément en tête, attend un temps défini qu'il soit disponible ;
- *E take()* : retire et retourne l'élément en tête, attend indéfiniment qu'il soit disponible.

D'autres collections concurrentes

ConcurrentLinkedQueue<E> : une file d'attente non bornée, threadsafe et sans attente.

- *boolean add(E elt)* : insert en queue l'élément et renvoie *true* ;
- *E poll()* : retire et retourne l'élément en tête ;
- *E peek()* : retourne sans enlever le premier élément.

ConcurrentHashMap<K,V> : une table de hachage dotée d'opération atomiques :

- *V putIfAbsent(K key, V value)* : si la clef est libre, associe la valeur à la clef ;
- *boolean remove(Object key, Object value)* : retire l'entrée associée à la clef ;
- *V replace(K key, V value)* : remplace l'entrée associée à la clef ;

Et le reste du package ?

Le package *java.util.concurrent.locks*

La classe *ReentrantLock* : verrou aux fonctionnalités étendues (re-entrant)

- *Thread getOwner()* :retourne le thread possédant le verrou ;
- *Collection<Thread> getQueuedThreads()* : retourne la collection de threads en attente ;
- *boolean hasQueuedThreads()* : indique si quelqu'un attend le verrou ;
- *boolean isLocked()* : indique si un verrou est pris ;
- *void lock()* : tente d'acquérir le verrou, attend sinon ;
- *boolean tryLock()* : tente d'acquérir le verrou, retourne faux sinon ;
- etc.

La classe ReentrantReadWriteLock : un couple de verrou (en lecture/écriture)

```
private final ReentrantReadWriteLock rwl =
                                  new ReentrantReadWriteLock();
private final Lock r = rwl.readLock();
private final Lock w = rwl.writeLock();
```

Semaphore

Un sémaphore encapsule un entier, avec une contrainte de positivité.

Deux opérations possibles :

- opération P (*acquire()*) : décrémente le compteur s'il est strictement positif, bloque en attente sinon ;
- opération V (*release()*) : incrémente le compteur.

Peut être vu comme un **ensemble de jetons**, avec deux opérations :

- prendre un jeton, en attendant si nécessaire qu'il y en ait ;
- déposer un jeton.

Les exécuteurs de tache

Permet la gestion d'un *pool de thread* :

- de nouveaux threads peuvent être créés au fil des besoins ;
- un thread ayant terminé sa tâche reste vacant ;
- de nouvelles tâches sont peuvent être allouées à des threads vacants.

La création de threads est couteuse !

La classe *Executors* :

- *static ExecutorService newFixedThreadPool(int nThreads)* : créé un nombre fixe de threads réutilisés au fil des besoins ;
- *static ExecutorService newCachedThreadPool()* : utilise un nombre non borné de threads ;
- *static ExecutorService newSingleThreadExecutor()* : utilise un thread unique.

L'interface *Executor* :

void execute(Runnable command) : exécutera la commande à un moment dans le future.

Un exemple ?

```
public class monLanceur {
  private final ExecutorService pool;

  public monLanceur(int poolSize) throws IOException {
    pool = Executors.newFixedThreadPool(poolSize);
  }

  public void run() {
    try {
      for (;;) {
        pool.execute(new maTache());
      }
    } catch (IOException ex) {
      pool.shutdown();
    }
  }
}
```