

JAVA SE - Modèle Mémoire

Introduction au modèle mémoire de Java

On peut trouver (en particulier sur le Web) de nombreux exemples de programmes qui sont construits sur le principe suivant :

```
public class Tache { // éventuellement implements Runnable
```

```
    private boolean continuer ;
```

```
    public void run() {  
        while(continuer) { // faire des tas de chose  
        }  
    }
```

```
    public void arrêter() {  
        continuer = false ;  
    }  
    // autre codes
```

```
}
```

Le principe général de ces programmes est que la méthode `run` sera exécutée par un *Thread* et la méthode `arrêter()` permettra à un autre *Thread* de suspendre l'exécution.

Ce type de programme présente un défaut : il "tombe en marche"! C'est à dire que l'auteur pourra tester le fonctionnement de nombreuses fois et se satisfaire d'un fonctionnement apparemment normal...

Sauf que ces programmes sont **faux!**

Ce type d'erreur est particulièrement redoutable car il peut, à l'occasion, provoquer un dysfonctionnement qu'il sera très difficile de reproduire expérimentalement.

Il est tout à fait possible à l'instant T qu'un *Thread* (nommons-le "thread initiateur") demande au thread cible de s'arrêter. Ensuite à l'instant $T+n$ le "thread impliqué" (celui qui exécute le `run`) devient actif, teste la valeur de la variable `continuer` et la voit `false`.

On peut donc avoir à un moment de l'exécution :

- `continuer` à `true` pour le "thread initiateur"
- `continuer` à `false` pour le "thread impliqué"

Comment est-ce possible?

La gestion des *threads* par le système hôte obéit à diverses règles de fonctionnement et d'optimisation: l'objectif est d'obtenir un multi-tâches performant.

Java se veut neutre et ne veut pas interférer avec ces stratégies. Les spécifications du langage tentent d'avoir une vision unifiée (et forcément limitée) de ces comportements complexes.

Les spécifications du langage déterminent un "modèle mémoire" qui, entre autres choses, dit à peu près ceci :

- Un *thread* a la possibilité de copier des données depuis la "mémoire principale" vers la "mémoire" propre à ce *thread*. Ceci pour pouvoir travailler en local de manière efficace.
- La recopie de la mémoire du *thread* vers la mémoire principale se fait à la discrétion du gestionnaire de tâches **sauf** si le programme demande explicitement cette recopie.

Il faut bien comprendre que ces demandes explicites vont avoir pour effet de perturber les stratégies d'optimisation du système d'ordonnancement des tâches et vont donc impacter les performances. Ceci dit il va y avoir des cas où ces échanges entre les différentes parties de la mémoire vont être nécessaires pour le bon fonctionnement de l'application.

Dans notre exemple on a un objet Tache qui va être partagé par deux *threads*. Le "thread initiateur" a le droit d'en faire une copie et d'opérer des modifications dans cette copie.

Tant que la mémoire de ce *thread* n'aura pas été recopiée dans la mémoire principale **ET** que le "thread impliqué" n'aura pas lui même "lu" l'état modifié de la mémoire principale il n'y aura pas synchronisation sur l'état réel de la variable continuer.

Le "thread impliqué" ne va pas immédiatement réagir lorsqu'il teste l'état de la variable, cela se produira probablement quelque part dans le futur quand il y aura eu de la part des deux *threads* une synchronisation avec l'état de la mémoire principale. Avec un peu de chance, et surtout si le code à exécuter n'est pas critique, on finit par avoir satisfaction (et donc par avoir l'illusion que notre programme est correct). Mais si l'enchaînement des actions entre l'initiateur et l'impliqué est critique on peut se retrouver avec une erreur très sournoise!

Il y a plusieurs manières de traiter le problème : cela dépend en partie de ce que l'on veut faire dans la classe Tache (arrêter définitivement l'exécution, la suspendre en attente d'une reprise, etc).

De manière générale lorsqu'on a une variable partagée entre plusieurs *Threads* qui doivent successivement la modifier et la lire il faut connaître le mécanisme des variables volatile.

Une des rédactions possibles de la classe Tache devient alors :

```
public class Tache { // éventuellement implements Runnable  
  
    private volatile boolean continuer ;  
  
    public void run() {  
        while(continuer) { // faire des tas de chose  
        }  
    }  
  
    public void arrêter() {  
        continuer = false ;  
    }  
    // autre codes  
  
}
```

Formations JAVA sur ce sujet

Pour approfondir les notions :

- d'interruption
- d'antériorité des actions (*happens before*)
- de synchronisation de la mémoire (volatile, blocs synchronized)
- d'atomicité des actions

Telles qu'elles sont décrites par les spécifications du langage Java vous pouvez suivre les cours suivant :

- [USL275 - Java : Programmation pour Développeur Confirmé](#)
- [UJD110 - Java : Programmation pour Développeur Débutant](#)
- [IIN43 - mise à niveau et préparation à la certification programmeur](#)
- Nos cours "java avancé" adaptés à la demande.

[Toutes les formations Développement](#)

Bernard AMADE

© Demos