Autres modèles de concurrence en Java

Introduction

La gestion de la concurrence par `synchronized` pose plusieurs difficultés :

- La granularité est très fine : on synchronise à la ligne de code près, sur base d'un objet particulier.
- La syntaxe implique des restrictions, voir `java.util.concurrent.locks`.
- On peut facilement avoir un dead-lock.
- On peut facilement oublier de synchroniser un endroit et avoir une race condition très difficile à détecter et débugger.
- On peut facilement trop synchroniser et perdre en performance.

Historiquement, cette technique provient des années 1960 et est très proche de la manière dont les processeurs physiques gèrent les sections critiques. Il va sans dire qu'en 50 ans d'autres approches ont vu le jour et les concepteurs Java en ont reprises certaines à leur compte. Nous allons en explorer quelques-unes.

Objets immuables

C'est parce qu'une zone de mémoire est partagée entre plusieurs threads qu'ils peuvent modifier cette zone de mémoire en se marchant sur les pieds et terminer en un état incorrect.

Les ressources statiques sur le Web (pages html, images gif, etc.) ne posent pas de problème de concurrence car elles sont en lecture seule. Peu importe qui lit la ressource, le résultat est toujours le même.

En Java, on peut obtenir cet effet en programmant des objets immuables :

> L'état de l'objet est déterminé à la construction, tous ses attributs sont privés et aucune méthode ne permet de les modifier. Par exemple, tous les attributs sont finaux.
>
> Les instances de `String`, `Integer` et `Double` sont des exemples d'objets immuables. Les partager entre différents threads ne pose jamais de problèmes de concurrence. Ceci reste fort restrictif. Notez que cela reste le principe de base des sites web statiques.

Service Stateless

Un service est dit stateless si le résultat de l'appel d'une fonction du service dépend uniquement des paramètres de la fonction. Le service n'a pas d'état propre.

En Java, on peut implémenter un service par un objet dont les méthodes sont les fonctions de ce service.

Vu que le service n'a aucun état propre, les différents threads qui y font appel ne partageront pas d'état entre eux, sauf si les paramètres de ces services peuvent être des objets à état partagé.

On utilisera donc un service stateless en passant en paramètre uniquement des objets immuables, on n'aura alors aucun risque de problème de concurrence.

Exemple de service stateless :

```
public Integer racineCarrée(Integer in) {…}
```

`Integer` est immuable. L'implémentation de `racineCarrée` ne dépend que de `in`, donc il n'y a aucun risque de concurrence.

Ceci est encore restrictif. Notez que HTTP est basé sur cette approche.

Opérations atomiques

Comme nous l'avons vu une instruction aussi simple que `i++` peut provoquer un race condition. On se retrouve donc rapidement à devoir synchroniser toutes les méthodes du code. Java permet donc d'éviter cela pour les opérations les plus simples en fournissant des opérations atomiques. Chaque opération atomique est thread-safe, porte sur une seule variable et ne peut pas générer de dead-lock.

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicLong;

public class Fukushima {
	private AtomicLong degats = new AtomicLong();
	private AtomicBoolean seisme = new AtomicBoolean(false);

	class DegatsSismique extends Thread {
		@Override
		public void run() {
			while (true) {
				Fukushima.sleep(5000);
				if (Math.random() > 0.5) {
					System.out.println("Tremblement de terre !");
					degats.addAndGet((long)(Math.random()*10000));
					seisme.set(true);
				}
			}
		}
	}

	class DegatsNucleaire extends Thread {
		@Override
		public void run() {
			while (true) {
				Fukushima.sleep(1000);
				degats.addAndGet((long) (Math.random() * 100));
			}
		}
	}

	class DegatsTsunami extends Thread {
		@Override
		public void run() {
			Fukushima.sleep(2000);
			while (true) {
				Fukushima.sleep(5000);
				if (seisme.getAndSet(false)) {
					System.out.println("Tsunami !");
					degats.addAndGet( (long)(Math.random()*10000));
				}
			}
		}
	}
```

```
	class CNN extends Thread {
		@Override
		public void run() {
			while (true) {
				System.out.println("Degats actuels " +
							degats.get());
				try {
					Thread.sleep(5000);
				} catch (InterruptedException e) {
					Thread.currentThread().interrupt();
				}
			}
		}
	}

	public Fukushima() {
		new DegatsSismique().start();
		new DegatsNucleaire().start();
		new DegatsTsunami().start();
		new CNN().start();
	}

	public static void main(String[] args) {
		new Fukushima();
	}

	public static void sleep(int ms) {
		try {
			Thread.sleep(ms);
		} catch (InterruptedException e) {
			Thread.currentThread().interrupt();
		}
	}
}
```

Dans l'exemple ci-dessus, nous avons 4 threads. 3 de ces threads ajoutent des valeurs à la variable atomique `degats` tandis que le dernier thread affiche sa valeur toutes les 5 secondes. Le thread qui gère les séismes ne peut ajouter des dégâts que s'il suit un tremblement de terre. On utilise donc la variable booléenne atomique `seisme` lorsqu'il y en a un.

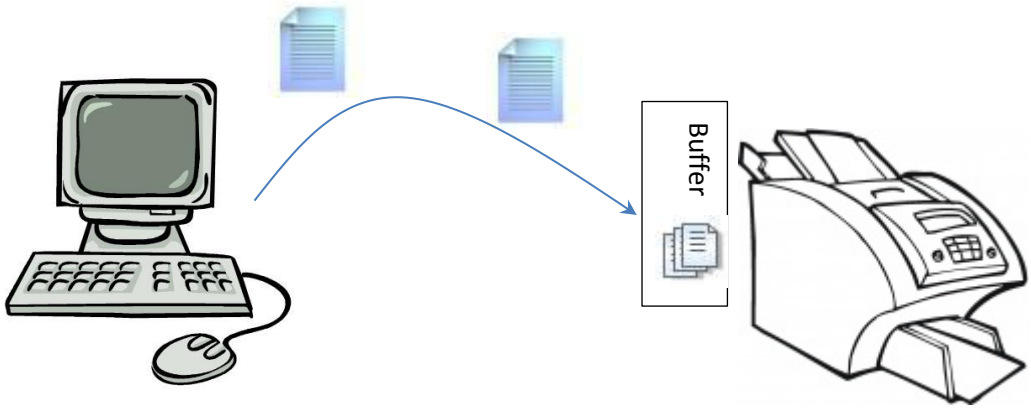
Remarquez l'usage de la méthode `getAndSet` de `seisme` (dans la classe `DegatsTsunami`) : en une seule opération atomique, on récupère le contenu de la variable tout en changeant son contenu. Si on avait utilisé une variable booléenne habituelle, il aurait fallu faire 2 opérations distinctes (obtenir l'ancienne valeur puis changer son contenu) ce qui est un risque de race condition.

Pour en savoir plus, confer la Javadoc :

http://download.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html

Le problème du producteur-consommateur

Imaginons un `Producteur` qui génère des pages à imprimer. Un `Consommateur` lit ces pages et les imprime. De plus, le `Consommateur` possède un buffer qui permet de retenir temporairement des pages en attente d'impression.



Il faudra un contrôle de flux entre le `Producteur` et le `Consommateur` :

- Le `Producteur` pourra envoyer une page au `Consommateur` s'il reste au moins une place de disponible dans son buffer, sinon il devra attendre (son thread s'endort). Si le `Consommateur` était endormi lorsqu'il reçoit cette page alors il se réveille.
- Lorsque le `Consommateur` a fini d'imprimer une page, il prend la page suivante du buffer. Ceci peut réveiller le `Producteur` ! Par contre, si le buffer est vide, alors le `Consommateur` devra attendre (son thread s'endort).

Ce genre de problème est récurrent en informatique. On pourra, par exemple, avoir un client web qui produit des requêtes avec un serveur web qui y répond. Si on ne traite qu'une seule requête à la fois, le client deviendra très lent : par exemple il pourrait demander une page html, puis demander simultanément les 3 images qui y sont incluses. Il vaut mieux traiter ces demandes en parallèle plutôt que séquentiellement. Par contre, si on accepte que le serveur traite toutes les requêtes sans limites, on risque très fort de dépasser ses propres capacités.

Java propose une solution directe à ce problème : `java.util.concurrent.BlockingQueue<E>`

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

class Producer implements Runnable {
	private final BlockingQueue<Integer> queue;

	Producer(BlockingQueue<Integer> q) {
		queue = q;
	}

	public void run() {
		try {
			while (true) {
				queue.put(produce());
			}
		} catch (InterruptedException ex) {
```

```
                    Thread.currentThread().interrupt();
                }
        }

        private Integer produce() {
                return (int)(Math.random()*100);
        }
}

class Consumer implements Runnable {
        private final BlockingQueue<Integer> queue;
        private int n = 0;
        private int total = 0;

        Consumer(BlockingQueue<Integer> q) {
                queue = q;
        }

        public void run() {
                try {
                        while (true) { consume(queue.take()); }
                } catch (InterruptedException ex) {
                        Thread.currentThread().interrupt();
                }
        }

        void consume(Integer x) {
                n++;
                total += x;
                if (total > 1000) {
                        System.out.println("Il a fallu "+n+
                                        " nombres pour que leur total dépasse 1000");
                        n = 0;
                        total = 0;
                }
        }
}

public class Setup {
        public static void main(String[] args) {
                BlockingQueue<Integer> q = new LinkedBlockingQueue<Integer>();
                Producer p = new Producer(q);
                Consumer c = new Consumer(q);
                new Thread(p).start();
                new Thread(c).start();
        }
}
```

Dans cet exemple, on a un `Producteur` et un `Consommateur` qui s'exécutent chacun dans leur propre thread. Le `Producteur` émet des entiers aléatoires. Le `Consommateur` les additionne et quand le total dépasse 1000 il affiche le nombre d'additions qu'il a dû effectuer pour en arriver là. Notez qu'il n'y a aucun contrôle de flux, tout est automatiquement géré par la `BlockingQueue`.

Pour en savoir plus, confer la Javadoc :

http://download.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html

Le thread pool et son exécuteur

La création d'un thread Java n'est pas une opération anodine : c'est un appel au système d'exploitation[1] qui prend un certain temps. Ce n'est donc pas parce qu'un algorithme peut facilement s'écrire en multi-thread que ceci sera nécessairement souhaitable.

```
public class Doubleur {
	private static class DoubleurThread extends Thread {
		private int index;
		private int[] tab;
		public DoubleurThread(int[] tab, int i) {
			index = i;
			this.tab = tab;
		}
		@Override
		public void run() {
			this.tab[index] = this.tab[index]*2;
		}

	}
	public static void doubleTout(final int[] tab) {
		for(int i = 0 ; i < tab.length ; i++) {
			new DoubleurThread(tab,i).start();
		}
	}
}
```

La méthode `doubleTout` multiplie par 2 chaque élément du tableau. Chaque multiplication est effectuée dans son propre thread.

D'une part ceci ne sera pas efficace : la création même d'un thread prend plus de temps que la multiplication et on perd donc plus de temps qu'on n'en gagne.

D'autre part on a un gros problème : on n'a aucune idée de quand tous les threads se sont exécutés et donc de quand les multiplications ont toutes été effectuées. Oups….

Il est donc très légitime de se poser la question du nombre de threads qu'il est optimal pour l'application d'utiliser. Une partie de la réponse tient au nombre de cœurs présents sur la machine physique : on peut certainement démarrer un thread par cœur. Mais ce n'est pas parce qu'un thread est en cours d'exécution sur un cœur qu'il va en utiliser tous les cycles. Chaque entrée-sortie avec la mémoire mais aussi et surtout avec le disque va suspendre le thread le temps que l'information soit récupérée. Suivant que le thread effectue beaucoup d'opérations d'entrée-sortie ou bien qu'il fait surtout du calcul intensif, on pourra donc démarrer ou pas plusieurs threads par cœur. La quantité optimale se détermine en général empiriquement par essai-erreur.

La question à se poser maintenant est : comment écrire son application Java de telle manière que le nombre de threads soit un paramètre du système et non pas une valeur figée arbitraire ?

Pour arriver à cet effet, on utilise la technique du *thread pool* (piscine à threads) : le thread pool contient un nombre déterminé et configurable de threads. Un *exécuteur* permet de soumettre des tâches à ce thread pool et orchestre leur exécution. Il peut par exemple recycler le même thread

[1] Sauf si la JVM est configurée pour utiliser des « greens » threads, mais dans ce cas elle ne pourra bénéficier des multiples cœurs présents sur la machine.

pour plusieurs tâches successives fournissant ainsi une implémentation très efficace de la concurrence. L'exécuteur aura aussi la responsabilité de la synchronisation sur la terminaison d'une tâche, et d'en récupérer le résultat le cas échéant. On appelle un *futur* le résultat d'une tâche soumise à un exécuteur.

Par exemple le scénario suivant :

- Le thread principal crée un thread pool d'une taille de 2.
- Il soumet la tâche `Marty` à l'exécuteur du thread pool qui confie cette tâche à son premier thread. Le thread principal récupère un futur sur `Marty`.
- Il soumet la tâche `Doc` à l'exécuteur du thread pool qui confie cette tâche à son deuxième thread. Le thread principal récupère un futur sur `Doc`.
- Plus tard, le thread principal effectue un retour vers le futur de `Marty` pour en obtenir le résultat. Supposons que cette tâche était déjà terminée et le thread principal obtient son résultat directement.
- Ensuite il effectue un retour vers le futur de `Doc`. Supposons cette fois par contre que la tâche ne soit pas encore terminée. Le futur suspend donc le thread principal et en attend la fin de l'autre. Une fois ceci terminé, le thread principal peut continuer avec le résultat.

```
public class BackToTheFuture {


	public static void main(String[] args)
			throws InterruptedException, ExecutionException {

		ExecutorService executor = Executors.newFixedThreadPool(2);

		Future<String> marty = executor.submit(new Callable<String>() {
			@Override
			public String call() throws Exception {
				Thread.sleep((long) (Math.random()*1000));
				return "Marty";
			}
		});

		Future<String> doc = executor.submit(new Callable<String>() {
			@Override
			public String call() throws Exception {
				Thread.sleep((long) (Math.random()*1000));
				return "Doc";
			}
		});

		System.out.println(marty.get()+" "+doc.get());

		executor.shutdown();
	}
}
```

L'interface `ExecutorService` définit la notion de cycle de vie pour un service d'exécution. Lorsque qu'il est créé, ce service est dans l'état ***en_cours***. Lorsque l'on veut l'éteindre, on le passe dans l'état ***en_cours_d_arrêt*** par appel à sa méthode `shutdown()`. Dans cet état, il ne peut plus accepter de nouvelles tâches, et le service tente d'éteindre les tâches qu'il est en train d'exécuter. Si l'on veut l'éteindre de façon urgente, alors on fait appel à sa méthode `shutdownNow()`, qui va tout arrêter sans attendre la fin normale de l'exécution des tâches en cours. Le dernier état est ***terminé*** lorsqu'il n'y a plus aucune tache à exécuter.

Pour en savoir plus, confer la Javadoc :

http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html