’Java Multithreading

Plongeons dans le multithreading Java et l’utilisation efficace du framework `Executor`. Le multithreading permet à votre programme d’exécuter plusieurs tâches simultanément, améliorant ainsi les performances pour les opérations liées à l’E/S ou intensives en CPU. Le framework `Executor` simplifie la gestion des threads en abstraisant la création et la gestion des threads de bas niveau. Je vais vous guider à travers les bases, puis vous montrer comment utiliser `Executor` et les outils associés.

---

1. Bases du multithreading en Java

Java propose deux moyens principaux de créer des threads : - **Étendre la classe** `Thread` :

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Démarre le thread
    }
}
```

- **Implémenter l’interface** `Runnable` :

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable running: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();
    }
}
```

- Appelez `start()` (et non `run()`) pour lancer le thread—`run()` exécute simplement le code dans le thread actuel.

---

2. Pourquoi utiliser `Executor` ?

La création manuelle de threads fonctionne pour des cas simples, mais elle est inefficace pour gérer de nombreuses tâches (par exemple, surcharge de threads, épuisement des ressources). Le framework `Executor` (introduit dans Java 5 sous `java.util.concurrent`) fournit un pool de threads et un système de gestion des tâches, rendant le multithreading plus propre et plus évolutif.

---

3. Utilisation de `ExecutorService`

L'interface `ExecutorService` (une sous-interface de `Executor`) est l'outil à utiliser. Voici comment l'utiliser :

Étape 1 : Créer un `ExecutorService` Utilisez la classe utilitaire `Executors` pour créer un pool de threads :

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
    public static void main(String[] args) {
        // Pool de threads fixe avec 4 threads
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // Soumettre des tâches
        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                System.out.println("Task executed by: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simuler le travail
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }
```

```
        // Arrêter l'exécuteur
        executor.shutdown(); // Empêche les nouvelles tâches, attend que les tâches existantes se terminent
    }
}
```

- `newFixedThreadPool(4)` crée un pool avec 4 threads. Les tâches en excès attendent dans une file d'attente.
- `submit()` accepte des tâches `Runnable` ou `Callable` (`Callable` retourne un résultat).

Types d'exécuteurs courants

- `Executors.newSingleThreadExecutor()` : Un thread, traite les tâches séquentiellement.
- `Executors.newCachedThreadPool()` : Crée des threads selon les besoins, réutilise les threads inactifs (bon pour les tâches à courte durée de vie).
- `Executors.newScheduledThreadPool(n)` : Pour planifier des tâches avec des délais ou une périodicité.

---

4. Gestion des résultats avec `Callable` et `Future`

Si vous avez besoin des résultats des tâches, utilisez `Callable` au lieu de `Runnable` :

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Soumettre une tâche `Callable`
        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });

        // Faire d'autres travaux pendant que la tâche s'exécute
        System.out.println("Task submitted...");

        // Obtenir le résultat (bloque jusqu'à ce que ce soit terminé)
        Integer result = future.get();
        System.out.println("Result: " + result);
```

```
        executor.shutdown();
    }
}
```

- `Future.get()` récupère le résultat, bloquant si la tâche n'est pas terminée.
- Utilisez `future.isDone()` pour vérifier l'achèvement sans bloquer.

---

5. Avancé : Ajustement du pool de threads

Pour des applications réelles, vous pourriez vouloir un `ThreadPoolExecutor` personnalisé :

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        // Pool de threads personnalisé : 2-4 threads, capacité de la file d'attente de 10
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2, // Taille du pool de base
            4, // Taille maximale du pool
            60L, // Délai d'inactivité du thread
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(10) // File d'attente des tâches
        );

        for (int i = 0; i < 15; i++) {
            executor.execute(() -> {
                System.out.println("Task by: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        executor.shutdown();
    }
}
```

- Taille de base (2) : Nombre minimum de threads maintenus en vie.
- Taille maximale (4) : Nombre maximum de threads si la file d'attente est pleine.
- File d'attente : Contient les tâches lorsque tous les threads sont occupés.

---

6. Bonnes pratiques

- **Arrêter correctement** : Appelez toujours `shutdown()` ou `shutdownNow()` pour éviter les fuites de ressources.
- **Gérer les interruptions** : Vérifiez `Thread.currentThread().isInterrupted()` dans les boucles.
- **Éviter la surcharge** : Trop de threads peut dégrader les performances—ajustez la taille du pool en fonction des cœurs CPU (par exemple, `Runtime.getRuntime().availableProcessors()`).
- **Utiliser `try-finally` avec `ExecutorService`** :

```
ExecutorService executor = Executors.newFixedThreadPool(4);
try {
    // Soumettre des tâches
} finally {
    executor.shutdown();
}
```

Exemple avec Log4j (Tout assembler)

Combiner le multithreading avec votre question précédente sur Log4j :

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import java.util.concurrent.*;

public class Main {
    private static final Logger logger = LogManager.getLogger(Main.class);

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            final int taskId = i;
            executor.submit(() -> {
```

```
                logger.info("Task {} started by {}", taskId, Thread.currentThread().getName());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    logger.error("Task {} interrupted", taskId, e);
                    Thread.currentThread().interrupt();
                }
                logger.info("Task {} completed", taskId);
            });
        }

        executor.shutdown();
    }
}
```

Avec le `log4j2.xml` précédent, ceci enregistre la progression des tâches dans la console et le fichier.