Java-Mehrfachverarbeitung

Lass uns in die Java-Multithreading und die effektive Nutzung des `Executor`-Frameworks eintauchen. Multithreading ermöglicht es Ihrem Programm, mehrere Aufgaben gleichzeitig auszuführen, wodurch die Leistung für I/O-gebundene oder CPU-intensive Operationen verbessert wird. Der `Executor`-Framework vereinfacht die Threadverwaltung, indem er die niedrigstufige Thread-Erstellung und -Verarbeitung abstrahiert. Ich werde Ihnen die Grundlagen erklären und Ihnen dann zeigen, wie Sie `Executor` und verwandte Tools verwenden.

---

1. Grundlagen des Multithreadings in Java

Java bietet zwei Hauptmethoden zur Erstellung von Threads: - **Erweitern der `Thread`-Klasse**:

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

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

- **Implementieren der `Runnable`-Schnittstelle**:

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

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

- Rufen Sie `start()` (nicht `run()`) auf, um den Thread zu starten—`run()` führt den Code im aktuellen Thread aus.

---

2. Warum `Executor` verwenden?

Manuelles Erstellen von Threads funktioniert für einfache Fälle, ist aber ineffizient bei der Verwaltung vieler Aufgaben (z. B. Thread-Overhead, Ressourcenerschöpfung). Der `Executor`-Framework (eingeführt in Java 5 unter `java.util.concurrent`) bietet einen Thread-Pool und ein Aufgabenverwaltungssystem, wodurch Multithreading sauberer und skalierbarer wird.

---

3. Verwendung von `ExecutorService`

Das `ExecutorService`-Interface (ein Unterinterface von `Executor`) ist das Werkzeug der Wahl. Hier ist, wie man es verwendet:

Schritt 1: Erstellen eines `ExecutorService` Verwenden Sie die `Executors`-Hilfsklasse, um einen Thread-Pool zu erstellen:

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

public class Main {
    public static void main(String[] args) {
        // Fester Thread-Pool mit 4 Threads
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // Aufgaben einreichen
        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                System.out.println("Aufgabe ausgeführt von: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Arbeit simulieren
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
```

```
        }

        // Beenden des Executors
        executor.shutdown(); // Verhindert neue Aufgaben, wartet auf bestehende
    }
}
```

- `newFixedThreadPool(4)` erstellt einen Pool mit 4 Threads. Überschüssige Aufgaben warten in einer Warteschlange.
- `submit()` akzeptiert `Runnable`- oder `Callable`-Aufgaben (`Callable` gibt ein Ergebnis zurück).

Gängige Executor-Typen

- `Executors.newSingleThreadExecutor()`: Ein Thread, verarbeitet Aufgaben sequenziell.
- `Executors.newCachedThreadPool()`: Erstellt Threads nach Bedarf, wiederverwendet inaktive (gut für kurzlebige Aufgaben).
- `Executors.newScheduledThreadPool(n)`: Zum Planen von Aufgaben mit Verzögerungen oder Periodizität.

---

4. Ergebnisse mit `Callable` und `Future` verarbeiten

Wenn Sie Ergebnisse von Aufgaben benötigen, verwenden Sie `Callable` anstelle von `Runnable`:

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

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

        // Eine Callable-Aufgabe einreichen
        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });

        // Andere Arbeit während der Ausführung der Aufgabe
        System.out.println("Aufgabe eingereicht...");

        // Ergebnis abrufen (blockiert, bis fertig)
        Integer result = future.get();
```

```
        System.out.println("Ergebnis: " + result);

        executor.shutdown();
    }
}
```

- `Future.get()` holt das Ergebnis ab, blockiert, wenn die Aufgabe noch nicht abgeschlossen ist.
- Verwenden Sie `future.isDone()`, um die Fertigstellung ohne Blockierung zu überprüfen.

---

5. Fortgeschritten: Thread-Pool-Tuning

Für Echtanwendungen möchten Sie möglicherweise einen benutzerdefinierten `ThreadPoolExecutor`:

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

public class Main {
    public static void main(String[] args) {
        // Benutzerdefinierter Thread-Pool: 2-4 Threads, Warteschlangenkapazität von 10
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2, // Kern-Pool-Größe
            4, // Maximale Pool-Größe
            60L, // Timeout für inaktive Threads
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(10) // Aufgabenwarteschlange
        );

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

        executor.shutdown();
```

```
    }
}
```

- Kerngröße (2): Mindestanzahl der Threads, die am Leben erhalten werden.
- Maximalgröße (4): Maximale Anzahl der Threads, wenn die Warteschlange voll ist.
- Warteschlange: Hält Aufgaben, wenn alle Threads beschäftigt sind.

---

6. Best Practices

- **Ordentliches Herunterfahren**: Rufen Sie immer `shutdown()` oder `shutdownNow()` auf, um Ressourcenlecks zu vermeiden.
- **Unterbrechungen behandeln**: Überprüfen Sie `Thread.currentThread().isInterrupted()` in Schleifen.
- **Überlastung vermeiden**: Zu viele Threads können die Leistung beeinträchtigen—passen Sie die Poolgröße basierend auf CPU-Kernen an (z. B. `Runtime.getRuntime().availableProcessors()`).
- **Verwenden Sie `try-finally` mit `ExecutorService`**:

```
ExecutorService executor = Executors.newFixedThreadPool(4);
try {
    // Aufgaben einreichen
} finally {
    executor.shutdown();
}
```

Beispiel mit Log4j (Alles zusammenfassen)

Kombinieren Sie Multithreading mit Ihrer früheren Log4j-Frage:

```
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("Aufgabe {} gestartet von {}", taskId, Thread.currentThread().getName());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    logger.error("Aufgabe {} unterbrochen", taskId, e);
                    Thread.currentThread().interrupt();
                }
                logger.info("Aufgabe {} abgeschlossen", taskId);
            });
        }

        executor.shutdown();
    }
}
```

Mit der `log4j2.xml` aus dem vorherigen Beispiel protokolliert dies den Fortschritt der Aufgaben in der Konsole und in einer Datei.