Java 多執行緒

讓我們深入探討 Java 的多執行緒以及如何有效地使用 `Executor` 框架。多執行緒允許您的程序同時運行多個任務，從而提高 I/O 密集型或 CPU 密集型操作的性能。`Executor` 框架通過抽象低級別的執行緒創建和處理來簡化執行緒管理。我將帶您了解基本知識，然後展示如何使用 `Executor` 及相關工具。

---

1. Java 多執行緒的基本知識

Java 提供了兩種主要方法來創建執行緒：- **擴展 `Thread` 類別**：

```
class MyThread extends Thread {
    public void run() {
        System.out.println(" 執行緒運行中: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // 啟動執行緒
    }
}
```

- **實現 `Runnable` 介面**：

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

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

- 請呼叫 `start()`（而不是 `run()`）來啟動執行緒——`run()` 只會在當前執行緒中執行代碼。

---

2. 為什麼使用 Executor？

手動創建執行緒對於簡單情況來說是可行的，但對於管理多個任務來說效率低下（例如，執行緒開銷、資源耗盡）。Executor 框架（在 Java 5 中引入，位於 java.util.concurrent）提供了一個執行緒池和任務管理系統，使多執行緒變得更加乾淨和可擴展。

---

3. 使用 ExecutorService

ExecutorService 介面（Executor 的子介面）是主要工具。以下是如何使用它：

第 1 步：創建一個 ExecutorService　使用 Executors 實用類來創建一個執行緒池：

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

public class Main {
    public static void main(String[] args) {
        // 固定執行緒池，4 個執行緒
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // 提交任務
        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                System.out.println(" 任務由執行緒執行: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // 模擬工作
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        // 關閉執行器
        executor.shutdown(); // 阻止新任務，等待現有任務完成
    }
}
```

- newFixedThreadPool(4) 創建一個包含 4 個執行緒的池。多餘的任務將在隊列中等待。
- submit() 接受 Runnable 或 Callable 任務（Callable 返回結果）。

常見的 Executor 類型

- `Executors.newSingleThreadExecutor()`：一個執行緒，按順序處理任務。
- `Executors.newCachedThreadPool()`：根據需要創建執行緒，重用空閒的執行緒（適合短暫任務）。
- `Executors.newScheduledThreadPool(n)`：用於安排具有延遲或週期性的任務。

---

4. 使用 `Callable` 和 `Future` 來處理結果

如果需要任務結果，請使用 `Callable` 而不是 `Runnable`：

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

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

        // 提交一個 Callable 任務
        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });

        // 在任務運行時進行其他工作
        System.out.println(" 任務已提交...");

        // 获取結果（如果任務未完成則阻塞）
        Integer result = future.get();
        System.out.println(" 結果: " + result);

        executor.shutdown();
    }
}
```

- `Future.get()` 獲取結果，如果任務未完成則阻塞。
- 使用 `future.isDone()` 來檢查完成情況而不阻塞。

---

5. 高級：執行緒池調整

對於實際應用，您可能需要自定義 `ThreadPoolExecutor`：

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

public class Main {
    public static void main(String[] args) {
        // 自定義執行緒池：2-4 個執行緒，隊列容量為 10
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2, // 核心池大小
            4, // 最大池大小
            60L, // 空閒執行緒超時
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(10) // 任務隊列
        );

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

        executor.shutdown();
    }
}
```

- 核心大小（2）：保持活躍的最小執行緒數。
- 最大大小（4）：如果隊列滿了則最大執行緒數。
- 队列：當所有執行緒都忙碌時保存任務。

---

6. 最佳實踐

- **正確關閉**：始終呼叫 `shutdown()` 或 `shutdownNow()` 以避免資源洩漏。

- **處理中斷**：在循環中檢查 `Thread.currentThread().isInterrupted()`。
- **避免過載**：過多的執行緒會降低性能——根據CPU核心數調整池大小(例如，`Runtime.getRuntime().availableProces`
- **使用 `try-finally` 來處理 `ExecutorService`**：

```
ExecutorService executor = Executors.newFixedThreadPool(4);
try {
    // 提交任務
} finally {
    executor.shutdown();
}
```

結合 Log4j 的範例（綜合應用）

將多執行緒與您之前的 Log4j 問題結合：

```
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(" 任務 {} 由 {} 啟動", taskId, Thread.currentThread().getName());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    logger.error(" 任務 {} 中斷", taskId, e);
                    Thread.currentThread().interrupt();
                }
                logger.info(" 任務 {} 完成", taskId);
            });
        }
```

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

使用之前的 `log4j2.xml`，這將把任務進度記錄到控制台和文件中。