Programación Multihilo en Java

Vamos a sumergirnos en el multihilo de Java y cómo usar el marco `Executor` de manera efectiva. El multihilo permite que tu programa ejecute múltiples tareas de manera concurrente, mejorando el rendimiento para operaciones intensivas en I/O o en CPU. El marco `Executor` simplifica la gestión de hilos abstrayendo la creación y manipulación de hilos a bajo nivel. Te guiaré a través de los conceptos básicos y luego te mostraré cómo usar `Executor` y herramientas relacionadas.

---

1. Conceptos Básicos del Multihilo en Java

Java proporciona dos formas principales para crear hilos: - **Extender la clase** `Thread`:

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

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

- **Implementar la interfaz** `Runnable`:

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

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

- Llama a `start()` (no a `run()`) para lanzar el hilo—`run()` solo ejecuta el código en el hilo actual.

---

2. ¿Por Qué Usar `Executor`?

Crear hilos manualmente funciona para casos simples, pero es ineficiente para gestionar muchas tareas (por ejemplo, sobrecarga de hilos, agotamiento de recursos). El marco `Executor` (introducido en Java 5 bajo `java.util.concurrent`) proporciona un grupo de hilos y un sistema de gestión de tareas, haciendo que el multihilo sea más limpio y escalable.

---

3. Usando `ExecutorService`

La interfaz `ExecutorService` (una subinterfaz de `Executor`) es la herramienta principal. Aquí te muestro cómo usarla:

Paso 1: Crear un `ExecutorService` Usa la clase utilitaria `Executors` para crear un grupo de hilos:

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

public class Main {
    public static void main(String[] args) {
        // Grupo de hilos fijo con 4 hilos
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // Enviar tareas
        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                System.out.println("Tarea ejecutada por: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simular trabajo
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }
```

```
        // Cerrar el executor
        executor.shutdown(); // Impide nuevas tareas, espera a que las existentes terminen
    }
}
```

- `newFixedThreadPool(4)` crea un grupo con 4 hilos. Las tareas adicionales esperan en una cola.
- `submit()` acepta tareas `Runnable` o `Callable` (`Callable` devuelve un resultado).

Tipos Comunes de Executor

- `Executors.newSingleThreadExecutor()`: Un hilo, procesa tareas secuencialmente.
- `Executors.newCachedThreadPool()`: Crea hilos según sea necesario, reutiliza los ociosos (bueno para tareas de corta duración).
- `Executors.newScheduledThreadPool(n)`: Para programar tareas con retrasos o periodicidad.

---

4. Manejo de Resultados con `Callable` y `Future`

Si necesitas resultados de tareas, usa `Callable` en lugar de `Runnable`:

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

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

        // Enviar una tarea Callable
        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });

        // Hacer otro trabajo mientras la tarea se ejecuta
        System.out.println("Tarea enviada...");

        // Obtener resultado (bloquea hasta que termine)
        Integer result = future.get();
        System.out.println("Resultado: " + result);

        executor.shutdown();
```

```
    }
}
```

- `Future.get()` recupera el resultado, bloqueando si la tarea no ha terminado.
- Usa `future.isDone()` para verificar la finalización sin bloquear.

---

5. Avanzado: Ajuste del Grupo de Hilos

Para aplicaciones del mundo real, podrías querer un `ThreadPoolExecutor` personalizado:

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

public class Main {
    public static void main(String[] args) {
        // Grupo de hilos personalizado: 2-4 hilos, capacidad de cola de 10
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2, // Tamaño del grupo principal
            4, // Tamaño máximo del grupo
            60L, // Tiempo de espera del hilo ocioso
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(10) // Cola de tareas
        );

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

        executor.shutdown();
    }
}
```

- Tamaño principal (2): Mínimo de hilos mantenidos vivos.

- Tamaño máximo (4): Máximo de hilos si la cola está llena.
- Cola: Mantiene tareas cuando todos los hilos están ocupados.

---

6. Mejores Prácticas

- **Cerrar Correctamente**: Siempre llama a `shutdown()` o `shutdownNow()` para evitar fugas de recursos.
- **Manejar Interrupciones**: Verifica `Thread.currentThread().isInterrupted()` en bucles.
- **Evitar Sobrecarga**: Demasiados hilos pueden degradar el rendimiento—ajusta el tamaño del grupo según los núcleos de la CPU (por ejemplo, `Runtime.getRuntime().availableProcessors()`).
- **Usar `try-finally` con `ExecutorService`**:

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

Ejemplo con Log4j (Uniendo Todo)

Combinando multihilo con tu pregunta anterior sobre 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("Tarea {} iniciada por {}", taskId, Thread.currentThread().getName());
                try {
```

```
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    logger.error("Tarea {} interrumpida", taskId, e);
                    Thread.currentThread().interrupt();
                }
                logger.info("Tarea {} completada", taskId);
            });
        }

        executor.shutdown();
    }
}
```

Con el `log4j2.xml` anterior, esto registra el progreso de la tarea en la consola y en el archivo.