Guía de Concurrencia Avanzada en Java

---

1. Interfaces Funcionales para Tareas Concurrentes

Runnable y Callable

Runnable: Interfaz para tareas que ejecutan un hilo sin devolver valor. Define el método `run()`.

```
Runnable runnable = () -> System.out.println("Thread ejecutado.");
new Thread(runnable).start();
```

Callable: Similar a Runnable, pero permite devolver un valor y lanzar excepciones. Define el método `call()`.

```
Callable<String> callable = () -> "Resultado de Callable";
String result = callable.call();
```

2. Ejecución de Tareas: Executor y ExecutorService

***Executor**:* Ejecuta tareas de `Runnable`, separando el envío de tareas de su ejecución.

***ExecutorService**:* Extiende `Executor`, permitiendo ejecutar `Callable` y gestionar threads.

```
ExecutorService executor = Executors.newFixedThreadPool(4);
executor.submit(runnable);                // Ejecuta Runnable
Future<String> future = executor.submit(callable);  // Ejecuta
Callable
executor.shutdown();
```

shutdown(): Detiene el `ExecutorService` después de completar las tareas pendientes.

shutdownNow(): Intenta detener las tareas activas y apagar el `ExecutorService` inmediatamente.

***Future**:* Representa el resultado de una tarea asíncrona.

`get()`: Espera y obtiene el resultado.
`cancel()`: Intenta cancelar la tarea si no se ha completado.

invokeAny e invokeAll

invokeAny(): Ejecuta una lista de `Callable` y devuelve el resultado del primero que termine exitosamente.

invokeAll(): Ejecuta una lista de `Callable` y devuelve una lista de `Future` con los resultados.

```
List<Callable<String>> tasks = Arrays.asList(callable1, callable2);
String result = executor.invokeAny(tasks);
List<Future<String>> futures = executor.invokeAll(tasks);
```

3. Problemas de Concurrencia

- **Deadlock**: Ocurre cuando dos o más hilos esperan mutuamente los recursos que el otro posee, bloqueándose indefinidamente.
- **Starvation**: Un hilo no puede obtener recursos porque otros hilos acaparan continuamente.
- **Livelock**: Hilos que siguen ejecutándose sin hacer progresos debido a la dependencia mutua de sus acciones.
- **Race Condition**: Conflicto por acceso simultáneo de hilos a un recurso compartido, causando resultados impredecibles.

4. Sincronización de Hilos

Locks Intrínsecos y Bloques Synchronized

Locks Intrínsecos: Controlan el acceso a un recurso compartido. Utilizados en métodos o bloques `synchronized`.

Métodos Synchronized: Aseguran que solo un hilo acceda a un método a la vez.

Bloques Synchronized: Permiten especificar el objeto sobre el cual se adquiere el lock.

```
synchronized (monitor) {
    // Código sincronizado
}
```

5. Clases Atómicas

Clases como `AtomicInteger`, `AtomicBoolean`, `AtomicLong`, `AtomicReference` permiten realizar operaciones atómicas sobre tipos primitivos y referencias, evitando `synchronized`.

Métodos Clave

`compareAndSet(expectedValue, newValue)`: Cambia el valor solo si coincide con el esperado.

`getAndSet(newValue)`: Actualiza el valor y devuelve el antiguo.

`incrementAndGet()`, `decrementAndGet()`: Incrementa o decrementa y devuelve el valor actualizado.

```
AtomicInteger atomicInt = new AtomicInteger(10);
atomicInt.incrementAndGet();
```

Operaciones Atómicas Avanzadas

Funciones de Actualización:

`updateAndGet()`: Actualiza usando una función y devuelve el valor nuevo.

`accumulateAndGet()`: Usa una función binaria para actualizar y devolver el valor.

```
atomicInt.accumulateAndGet(10, (a, b) -> a + b); // Suma 10 al valor
actual
```

6. CyclicBarrier

Permite que un grupo de threads espere hasta que todos lleguen a un "punto de barrera".

```
CyclicBarrier barrier = new CyclicBarrier(2, () ->
System.out.println("Threads released"));
barrier.await(); // Espera a que ambos threads lleguen
```

Métodos Clave

- **await()**: Espera a que todos los threads invoquen `await()`.
- **getNumberWaiting()**: Número de threads esperando en la barrera.
- **reset()**: Reinicia la barrera.

7. CopyOnWriteArrayList

Lista segura para múltiples threads; crea una copia de la lista al modificarla, evitando conflictos.

```
List<String> list = new CopyOnWriteArrayList<>();
list.add("Hello");
list.addIfAbsent("Hello");
```

8. Fork/Join Framework

Framework para tareas grandes divididas en subtareas que se ejecutan en paralelo. Utiliza `ForkJoinPool` y tareas de tipo `RecursiveTask` o `RecursiveAction`.

Componentes Principales

- **ForkJoinPool**: Ejecuta tareas en paralelo.
- **ForkJoinTask**: Tareas especializadas en dividirse en subtareas.
    - `fork()`: Ejecuta la tarea de forma asíncrona.
    - `join()`: Espera el resultado de la tarea.

Ejemplo de RecursiveTask

```
ForkJoinPool pool = new ForkJoinPool(); MyTask task = new
MyTask(listaNumeros); int resultado = pool.invoke(task);
```

9. Streams Paralelos y Operaciones Concurrentes

Streams Paralelos ejecutan operaciones de forma paralela, dividiendo el procesamiento en varios threads.

```
Stream<String> stream = Arrays.asList("a","b","c").parallelStream();
String resultado = stream.reduce("", String::concat,
String::concat);
```

Collectors Concurrentes

`Collectors.groupingByConcurrent(predicate)`: Agrupa elementos en un mapa concurrente.

`Collectors.toConcurrentMap(keyMapper, valueMapper)`: Convierte elementos a valores en un mapa concurrente.

```
Stream<String> stream = Arrays.asList("java",
"oracle").parallelStream();
ConcurrentMap<String, Integer> map =
stream.collect(Collectors.toConcurrentMap(String::toUpperCase,
String::length));
```

Resumen Visual

- **Interfaces Funcionales**: `Runnable` (sin valor de retorno), `Callable` (con valor de retorno).
- **Gestión de Ejecución**: `ExecutorService` y métodos `submit`, `invokeAny`, `invokeAll`.
- **Problemas de Concurrencia**: Deadlock, Starvation, Livelock, Race Conditions.
- **Sincronización**: Locks y bloques `synchronized`.
- **Clases Atómicas**: Operaciones atómicas seguras para tipos primitivos y referencias.
- **Herramientas para Concurrencia**: `CyclicBarrier`, `CopyOnWriteArrayList`, y `Fork/Join Framework`.
- **Streams y Collectors Paralelos**: Procesamiento concurrente eficiente.

Este esquema te brinda una referencia rápida de los principales conceptos y herramientas para trabajar con concurrencia en Java.