2. Programación concurrente

Parte II. Simultaneidad de hilos y datos compartidos

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco


Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional.

Tabla de contenido

3. Utilizar *thread executors*

Cuando se trata con varios hilos en una aplicación, te puedes enfrentar dos problemas:

- El rendimiento de la aplicación no es tan bueno como se esperaba, ya que hay demasiados hilos ejecutándose al mismo tiempo.
- El código se vuelve un poco confuso, porque hay que crear e iniciar cada hilo que se necesite.

Estos problemas se pueden evitar parcialmente utilizando **ejecutores de hilos** (*thread executors*). Estas estructuras permiten crear un conjunto de hilos y dejar que un objeto especial maneje los hilos en nuestro lugar. Por ejemplo, si definimos un hilo como el siguiente:

```
public class MyThread implements Runnable {
    …
    @Override
    public void run() {
        …
    }
}
```

Después se puede crear un ejecutor de hilos que maneje objetos de tipo *MyThread* (y cualquier otro objeto *Runnable*), de esta manera:

```
ThreadPoolExecutor executor = (ThreadPoolExecutor)Executors.newCachedThreadPool();

MyThread t1 = new MyThread();
MyThread t2 = new MyThread();

executor.execute(t1);
executor.execute(t2);
executor.shutdown();
```

Por lo general, se puede definir un bucle para crear hilos usando una expresión lambda y agregarlos al ejecutor, de esta manera:

```
ThreadPoolExecutor executor = …

for (int i = 0; i < N; i++) {
    executor.execute(() -> {
    // Thread code
    });
}

executor.shutdown();
```

En ambos casos, se utiliza la clase ***ThreadPoolExecutor*** (del paquete *java.util.concurrent*) para manejar el grupo de hilos. Hay muchas formas de conseguir un objeto de este tipo, pero el utilizado en el código anterior es bastante simple. Luego, se pueden crear tantos hilos como se necesiten y llamar al método *execute()* del ejecutor del hilo. A partir de ese momento, el ejecutor del hilo se encargará de llamar al método *start()* de cada hilo. Cuando todos los hilos se han añadido al grupo, deberá llamarse al método *shutdown()* del ejecutor para dejar que el programa finalice cuando todos los hilos terminen su tarea.

¿Qué ventajas tiene esto?

- El ejecutor de hilos puede reutilizar un hilo (o una posición en el grupo) para poner un hilo nuevo si algún hilo anterior ha terminado su tarea, por lo que puede ahorrar algo de espacio en la memoria.
- También se puede decir al ejecutor cuántos hilos se quieren ejecutar al mismo tiempo, usando esta instrucción en lugar de la primera utilizada antes:

  ```
  ThreadPoolExecutor executor = (ThreadPoolExecutor)Executors.newFixedThreadPool(10);
  ```

  Si se limita el tamaño del grupo, cada hilo que exceda este tamaño estará esperando un espacio libre antes de comenzar su tarea. Esto puede resultar especialmente útil si se ajusta el tamaño total al número total de núcleos del procesador, de esta forma:

  ```
  ThreadPoolExecutor executor =(ThreadPoolExecutor)
      Executors.newFixedThreadPool(
          Runtime.getRuntime().availableProcessors()
      );
  ```

  También se puede utilizar este método desde la clase *Executors* para ajustar el tamaño del grupo a la cantidad de procesadores disponibles.

  ```
  ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool();
  ```

- Además, existen algunos métodos útiles en la clase *ThreadPoolExecutor*, como ***getPoolSize()*** (devuelve cuántos hilos hay añadidos actualmente al grupo), ***getActiveCount()*** (devuelve cuántos hilos en la cola todavía están vivos) o ***shutdownNow()*** (fuerza a todos los hilos de la cola a terminar de inmediato).

3.1.1. Mas opciones

Hay más opciones con ejecutores de hilos que no se tratarán en esta unidad. Algunas de las más útiles son:

- Añadir un retraso a una tarea en un ejecutor, para que comience a ejecutarse después de ese retraso. Para hacer esto, puedes usar la clase *ScheduledThreadPoolExecutor*.
- Ejecutar una tarea periódicamente en un ejecutor, también a través de la clase *ScheduledThreadPoolExecutor*.

4. Acceso a datos simultáneamente

En esta sección se verá qué tipo de objetos se utilizarán para tratar con valores simples o colecciones de forma segura en una aplicación multi-hilo.

4.1. Acceder a valores únicos

En esta sección se verá cómo resolver algunos problemas asociados con valores individuales (primitivos y objetos), cómo escribirlos (actualizar su valor) y leerlos de forma segura cuando muchos hilos están accediendo al mismo tiempo.

4.1.1. Presentando el problema

Supón que tienes una clase para administrar una variable (entero) y algunos métodos para modificarla y obtener su valor.

```
public class SimpleInteger {
    int num;

    public SimpleInteger(int num) {
        this.num = num;
    }

    public int getNum() {
        return num;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public void increment() {
        num++;
    }
}
```

Ahora, crea algunos hilos que modifiquen los valores de ese número llamando al método de incremento disponible (se gestionará con un *Executor*):

```
public static SimpleInteger simpleInt = new SimpleInteger(0);

public static void main(String[] args) {
    ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();

    for (int i = 0; i < 10000; i++) {
        executor.execute(() -> simpleInt.increment());
    }
    executor.shutdown();

    while (!executor.isTerminated()) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) { }
    }
    System.out.println("Expected: 10000, Result: " + simpleInt.getNum());
}
```

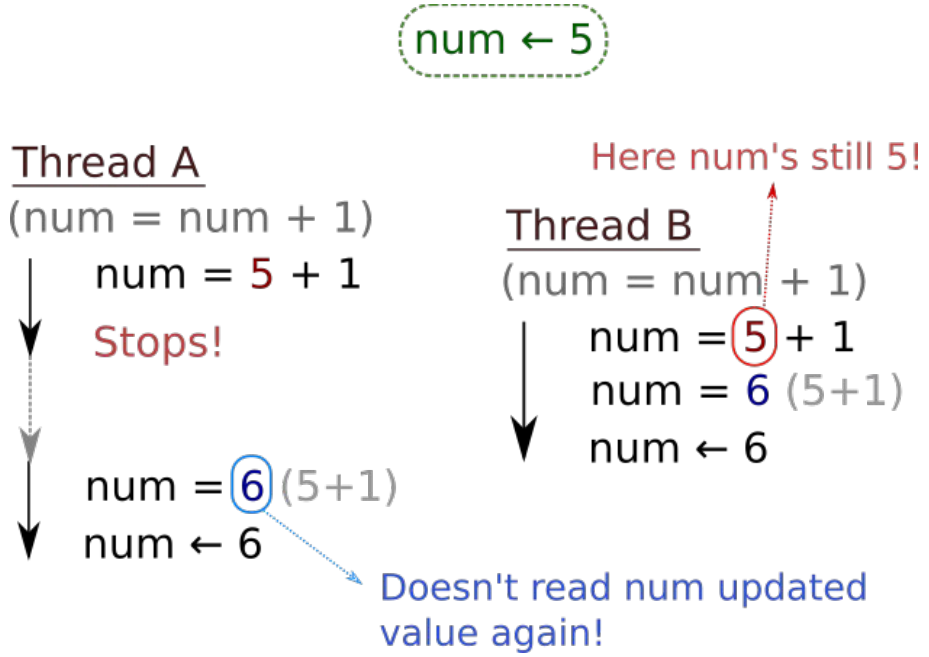
¿Qué pasa cuando se ejecuta este código? Es posible que se obtengan resultados como estos:

```
Expected: 10000, Result: 9961
Expected: 10000, Result: 9983
```

Podrás ver que hay un problema. 10,000 hilos incrementando cada uno la misma variable debería obtener siempre 10,000, pero se obtiene un resultado diferente cada vez y ninguno es correcto. He aquí por qué: examina esta operación: **num++**. Para el compilador, este incremento se traduce así → **num = num + 1**. Existen tres operaciones teniendo lugar aquí (imagina que el valor actual de **num** es 5):

- Sustitución de variable: **num = 5 + 1**
- Operación añadir: **num = 6**
- Asignar el nuevo valor a num: **num ← 6**

Si el hilo (cuando el valor actual es 5) se detiene **justo después** de hacer el primer o segundo paso, el siguiente hilo que acceda a esa variable no verá la actualización del valor y verá que sigue siendo 5. Este segundo hilo incrementará 5 y asignará 6 a la variable *num*. Cuando el primer hilo despierta, continuará desde el último paso antes de detenerse, por lo que asignará 6, como el otro hilo (¡¡se ha perdido un incremento!!).



4.1.2. Resolver el problema con sincronización

Como se ha visto, un bloque o método sincronizado solo permite acceder a un hilo a la vez. Si un hilo quiere entrar mientras otro está dentro, será interrumpido hasta que salga el hilo dentro del bloque o método.

Se puede utilizar de varias formas:

- Fuera de la clase (en el método principal, por ejemplo):

```
executor.execute(() -> {
    synchronized (simpleInt) {
        simpleInt.increment();
    }
});
```

- Dentro de los métodos del objeto, siempre que se intente cambiar el valor de la variable que puede ser accesible por más de 1 hilo:

```
public void increment() {
    synchronized (this) {
        num++;
    }
}
```

- En la declaración del método (para cada método que cambie el valor de la variable):

```
public synchronized void increment() {
    num++;
}
```

Deberás utilizar solo una solución de las mostradas arriba (no todas). Ahora, funcionará correctamente y producirá → 10,000 como el resultado.

4.1.3. Resolverlo con objetos atómicos

En lugar de sincronización, también se pueden usar variables atómicas (desde Java 5), un mecanismo para verificar que la operación aplicada a una variable dada se realiza en un solo paso. De esta forma, se pueden utilizar tipos de datos como ***AtomicLong***, ***AtomicInteger*** o ***AtomicBoolean*** para tratar con tipos de datos simples largos, enteros o booleanos. Por tanto, en lugar de hacer esto (que no es seguro para hilos):

```
long num;
num = 10;
num++;
```

Se puede hacer esto:

```
AtomicLong num = new AtomicLong();
num.set(10);
num.getAndAdd(1);
```

Al aplica esto al problema, se puede usar una variable *AtomicInteger* en lugar de solo un *int*:

```
public class SimpleInteger {
    AtomicInteger num;

    public SimpleInteger(int num) {
        this.num = new AtomicInteger(num);
    }

    public int getNum() {
        int localReturn = num.get();
        return localReturn;
    }
```

```
    public void setNum(int num) {
        this.num.set(num);
    }

    public void increment() {
        num.incrementAndGet();
    }
}
```

Existe un tipo adicional llamado ***AtomicReference***, que está parametrizado, de modo que se pueda usar para hacer que cualquier otro tipo de datos sea atómico:

```
AtomicReference<String> name = new AtomicReference<String>();
name.set("Javi");
System.out.println("My name is " + name.get());
```

4.2. Acceso a *arrays*

Con respecto a los ***arrays***, Java también proporciona tipos de datos atómicos para poder tratarlos. Por ejemplo, puedes utilizar ***AtomicIntegerArray*** para manejar *arrays* de enteros, o ***AtomicReferenceArray*** para muchos otros tipos de datos.

```
// Create an array of strings with size 10
AtomicReferenceArray<String> names = new AtomicReferenceArray<String>(10);

// Add names to some positions
names.set(0, "Javi");
names.set(1, "Nacho");

// Get names at given positions
System.out.println("Name at 1st position is " + names.get(0));
```

Ejercicio 9

Crea un proyecto llamado ***AtomicCounter*** a partir del ejemplo que se muestra en la sección 2.7.2 (unidad 2, parte I), sin mecanismo de sincronización. Ahora deberás usar un atributo *AtomicInteger* (en lugar del atributo *int* del ejemplo), para garantizar que el incremento y decremento de las operaciones contra este objeto sean atómicas y, por tanto, seguras para los hilos.

4.3. Colecciones concurrentes y sincronizadas

Cuando se quiere utilizar colecciones de datos en un programa concurrente, se debe tener mucho cuidado con la forma en que se manejen los datos. Muchas de estas colecciones, como *ArrayList*, no están preparadas para trabajar con varios hilos y, si varios de ellos intentan modificar el mismo elemento, se le puede asignar un valor final incorrecto. Pero hay otras colecciones que se pueden utilizar de forma segura.

4.3.1. Colecciones sincronizadas

Las colecciones sincronizadas se generan a partir de la clase estática *Collections*[1]. Estas colecciones sincronizan el acceso a la lista (todos los métodos excepto los iteradores), bloqueando la lista completa para un solo hilo a la vez.

```
List<String> list = new ArrayList<>(); // Not synchronized
List<String> syncList = Collections.synchronizedList(list); // Synchronized
```

4.3.2. Colecciones concurrentes

Respecto a las colecciones concurrentes, estas pueden ser:

- ***Blocking***: cuando se intenta añadir o eliminar algo de la colección y no se puede, el hilo se bloquea hasta que se pueda realizar la operación. En este grupo se puede utilizar, por ejemplo, la clase ***LinkedBlockingDeque***. Tiene un método *put()* para añadir elementos a la lista, *getFirst()*/*getLast()* para obtenerlos desde el principio/final de la lista y los métodos *takeFirst()*/*takeLast()* para obtenerlos y eliminarlos del principio/final de la lista. Consulta la API para obtener una explicación completa de estos y otros métodos.

  ```
  LinkedBlockingDeque<String> data = new LinkedBlockingDeque<String>();
  data.put("One element");
  data.put("Another element");
  String first = data.takeFirst();
  ```

- ***Non-blocking***: si la operación de añadir/eliminar no se puede realizar, se devuelve un valor nulo o se lanza una excepción. En este grupo se puede utilizar, por ejemplo, la clase ***ConcurrentLinkedDeque***. Tiene un método *add()* para añadir elementos a la lista, *getFirst()*/*getLast()* para obtenerlos desde el principio/final de la lista, *removeFirst()*/*removeLast()* para eliminar el primer/último elemento de la lista, y así sucesivamente (consulta la API para obtener una explicación completa de sus métodos).

  ```
  ConcurrentLinkedDeque<String> data = new ConcurrentLinkedDeque<String>();
  data.add("One element");
  data.add("Another element");
  String first = data.getFirst();
  data.removeFirst();
  ```

1 Collections (https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html)

Con respecto a los mapas, se puede utilizar, por ejemplo ***ConcurrentSkipListMap***, que es una implementación *non-blocking* de un tipo de tabla *hash*. Por ejemplo, se podría almacenar un conjunto de títulos de libros, identificados por su ISBN:

```
ConcurrentSkipListMap<String, String> map = new ConcurrentSkipListMap<String, String>();
data.put("1122", "Ender's game");
data.put("3344", "The Da Vinci's code");

Map.Entry<String, String> element = map.firstEntry();

String isbn = element.getKey();
String title = element.getValue();

System.out.println("First element is " + isbn + " - " + title);
```

4.3.3. Sincronización vs concurrencia

Existe una diferencia importante entre usar una colección sincronizada y usar una concurrente:

- Llamar a un método en una colección sincronizada bloquea todo el objeto de la colección, por lo que ningún otro hilo puede acceder a la colección, incluso si se quiere llegar a un índice diferente.
- Llamar a un método en una colección concurrente generalmente bloquea solo la posición o clave a la que se accede. En conclusión, estas colecciones consiguen un mejor rendimiento que las sincronizadas cuando se trabaja con muchos hilos.

Además, al usar colecciones sincronizadas, iterar sobre ellas no es seguro para los hilos, por lo que deberá implementarse manualmente:

```
synchronized (syncList) {
    Iterator i = syncList.iterator(); // Must be in synchronized block

    while (i.hasNext()) {
        // Do something
    }
}
```

Este problema no ocurre con las colecciones concurrentes. Sus iteradores son seguros, pero no garantizan que si otro hilo cambia un elemento mientras se itera, el hilo actual verá esa modificación inmediatamente.

4.3.4. Actualizar objetos internos

Cuando desees modificar un objeto interno de la colección, si lo haces obteniendo el objeto y luego realizas alguna modificación en sus propiedades, deberás sincronizar esa operación (o los métodos del objeto).

```
ConcurrentLinkedDeque<String> data = new ConcurrentLinkedDeque<String>();
data.add("One element");
data.add("Another element");
data.getFirst().replace("One", "First");
```

Cuando se concatena la operación *getFirst()* con el método *replace()*, el primero es seguro para hilos, pero una vez que se obtiene el *string*, el segundo método (el reemplazo) no es seguro para hilos. Entonces, se podría, por ejemplo, usar un método sincronizado que reemplace el texto, o métodos como *compute()* que se ejecutan de forma atómica. Por ejemplo, si quieres reemplazar el texto del primer elemento (índice 0) de la lista anterior, se podría hacer algo como esto:

```
data.compute(0, value -> {
    value = value.replace("One", "First");
    return value;
});
```

Se toma el valor de esa posición (posición 0), se almacena el reemplazo y luego se devuelve, de modo que el valor anterior sea reemplazado automáticamente por este nuevo valor devuelto.

Ejercicio 10

Crea un proyecto llamado ***ConcurrentVideoGames***. Deberás manejar una lista de videojuegos con un título y un precio, según esta estructura:

```
public class VideoGame {
    String title;
    float price;

    public VideoGame(String title, float price) {
        this.title = title;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public float getPrice() {
        return price;
    }
}
```

En el programa principal, se creará una lista de 100 videojuegos.

Para añadir videojuegos automáticamente, simplemente añadelos con un patrón determinado (como *“Videogame 1”*, *“Videogame 2”*, etc) y el mismo precio (o uno aleatorio).

Luego, se lanzarán dos hilos:

- Uno de ellos sumará 1 al precio de cada videojuego de la lista (durmiendo 50ms después de cada operación).
- El otro restará 1 al precio de cada videojuego de la lista (durmiendo 50ms después de cada operación).

Prueba en primer lugar con un simple *ArrayList*, y observa cómo el precio total de la lista es diferente cada vez que ejecuta el programa. Después, cambia el *ArrayList* por una estructura adecuada para hilos seguros, como *LinkedBlockingDeque*, y observa cómo funciona ahora perfectamente.