2. Programación concurrente

Anexo II. Más sobre hilos y sincronización

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

1. Introducción

Una vez que cubiertos todos los conceptos básicos de la programación concurrente en Java, en esta sección se obtendrá una descripción general de algunas características adicionales que pueden ser útiles. La mayoría de ellas se han incluido desde Java 5 y versiones posteriores, y no forman parte del núcleo inicial explicado anteriormente. Algunos otros, como los grupos de hilos, existen desde el comienzo de Java, aunque apenas se utilizan.

Para ser más precisos, se hablará de:

- Cómo definir y usar grupos de hilos para manejar múltiples hilos como un todo.
- Cómo definir variables locales para cada hilo dentro del mismo contexto.
- Cómo usar hilos *daemon*.
- Cómo sincronizar bloques de código utilizando la interfaz *Lock* en lugar de sincronizar.
- Cómo paralelizar el trabajo de los hilos automáticamente con el framework *Fork/Join*.

2. Grupos de hilos y demonios

2.1 Grupos de hilos

Java permite poner algunos hilos en un grupo para poder tratar este grupo como una sola unidad. De esta forma, se podrán tener algunos hilos realizando una tarea y controlarlos independientemente del número total de hilos del grupo.

Para gestionar grupos, se utilizará la clase ***ThreadGroup***. Se puede crear un grupo básico con un nombre dado, e incluso un grupo dentro de otro grupo, con su propio nombre:

```
ThreadGroup g1 = new ThreadGroup("Main group");
ThreadGroup g2 = new ThreadGroup(g1, "Additional group inside main group");
```

Para añadir hilos a un grupo, se pueden usar algunos de los constructores disponibles en la clase *Thread*. Por ejemplo, si se crea un hilo extendiendo de la clase *Thread*, puede añadirse a un grupo con este constructor (y algunos otros, consulta la API para obtener más detalles):

```
public Thread(ThreadGroup group, String name);
```

Si se crea el hilo implementando interfaz *Runnable*, se puede añadir con estos constructores (y algunos otros, consulta la API para obtener más detalles):

```
public Thread(ThreadGroup group, Runnable target);
public Thread(ThreadGroup group, Runnable target, String name);
```

Una vez añadidos los hilos a un grupo, hay algunos métodos útiles dentro de la clase *ThreadGroup*, como:

- ***activeCount***: devuelve el número de hilos en el grupo (y sus subgrupos) que están actualmente activos (no terminados).
- ***enumerate(Thread[] array)***: copia en el *array* especificado cada hilo activo del grupo (y sus subgrupos).
- ***interrupt***: interrumpe todos los hilos del grupo.
- ***setMaxPriority()***/***getMaxPriority()***: establece/obtiene la máxima prioridad de los hilos del grupo.

2.2.1. Ejemplo

El siguiente ejemplo crea algunos hilos de una clase que implementa la interfaz *Runnable*. Se supone que estos hilos generan un número aleatorio entre 1 y 10, duermen el número de segundos especificado por este número aleatorio y luego imprimen un mensaje en la pantalla. Pero tan pronto como el primer hilo termina su tarea, todo el grupo es interrumpido. El código para el objeto *Runnable* es:

```
import java.util.Random;
import java.util.concurrent.TimeUnit;

public class MyRandomMessage implements Runnable {
    Random r = new Random(System.currentTimeMillis());

    @Override
    public void run() {
        int time = r.nextInt(10) + 1;

        try {
            TimeUnit.SECONDS.sleep(time);
            System.out.println("Thread waited " + time +
                    " seconds and finished.");
        } catch (Exception e) { }
    }
}
```

El programa principal quedará como sigue:

```
public static void main(String[] args) {
    ThreadGroup g = new ThreadGroup("Random messages");
    MyRandomMessage m = new MyRandomMessage();
    Thread t1 = new Thread(g, m);
    Thread t2 = new Thread(g, m);
    Thread t3 = new Thread(g, m);
    t1.start();
    t2.start();
    t3.start();

    while (g.activeCount() == 3) {
        try {
            Thread.sleep(100);
        } catch (Exception e) { }
    }
    g.interrupt();
}
```

Tan pronto como termine un hilo, el método *activeCount* devolverá un número menor que 3, y el hilo principal terminará su ciclo e interrumpirá todos los hilos. Si los otros hilos aún están esperando que expire el tiempo, se interrumpirán, se lanzará una excepción y no imprimirán su mensaje de finalización.

2.2. Hilos demonio

Un hilo demonio (*daemon*) es un tipo especial de hilo que ejecuta una tarea periódica de vez en cuando.

Sus principales propiedades son:

- Tienen una prioridad muy baja (es decir, se ejecutan cuando no es necesario ejecutar ningún otro hilo "normal").
- El programa principal no espera a que terminen. Esto es, si todos los hilos del programa han finalizado, pero todavía se está ejecutando un hilo demonio, también se terminará.

Por esta razón, un hilo demonio no debe realizar ninguna tarea crítica, ya que puede interrumpirse en cualquier momento y no se sabe cuándo podrá realizar esa tarea. Un buen ejemplo de un hilo demonio es el recolector de basura de Java.

Para crear un hilo demonio, solo hay que llamar al método ***setDaemon()*** de la clase *Thread* antes de comenzar el hilo:

```
Thread t = new MyThread();
t.setDaemon(true);
t.start();
```

También se puede utilizar el método ***isDaemon()*** de la clase *Thread* para comprobar si un hilo dado es un demonio o no.

2.3. Variables *ThreadLocal*

Se ha visto que, si se utiliza el mismo objeto en diferentes hilos (un objeto *Runnable* o cualquier otro objeto), todos comparten los datos de este objeto. Pero a veces se necesitará tener un atributo que no se comparta entre los hilos. Para hacer esto, se puede usar la clase ***ThreadLocal***, que permite especificar un tipo de datos para crear un atributo de este tipo, y crear múltiples valores de este atributo, cada uno asignado a un hilo diferente.

Por ejemplo, si se quiere que los hilos tengan su propia fecha de creación, se hará algo como esto:

```
public class MyRunnableClass implements Runnable {
    private static ThreadLocal<Date> creationDate = new ThreadLocal<Date>() {
        protected Date initialValue() {
            return new Date();
        }
    };
}
```

Si se quiere obtener el valor de este atributo para cada hilo, se llamará al método ***get()***, y si se quiere asignar un nuevo valor, se llamará a su método ***set()***. El método ***initialValue()*** en el código anterior se ejecuta cuando el atributo no tiene valor y el hilo está tratando de obtenerlo. También hay un método ***remove()*** que se puede usar para eliminar el valor de este atributo del hilo actual.

Entonces, se puede tener un método ***run()*** como este en la clase *MyRunnableClass*:

```
@Override
public void run() {
    System.out.println("This thread was created on " + creationDate.get());
    System.out.println("Updating creation date...");
    creationDate.set(new Date());
    System.out.println("Now the creation date is " + creationDate.get());
    System.out.println("Removing value...");
    creationDate.remove();
    System.out.println("Now the creation date is " + creationDate.get());
}
```

El código anterior mostrará tres fechas diferentes para el mismo hilo, una para cada llamada a *get()*, ya que se establece un nuevo valor entre la primera y la segunda llamada, y se elimina el valor después de la segunda llamada.

3. Sincronizar con *Lock*

Desde Java 5 hay otra forma de sincronizar hilos (además de usar la palabra clave *synchronized*) al intentar llegar a secciones críticas. Consiste en implementar una interfaz denominada ***Lock*** (del paquete ***java.util.concurrent.locks***). Esta interfaz proporciona métodos para bloquear y desbloquear un recurso determinado, de modo que se garantiza que las operaciones intermedias no se ejecutarán simultáneamente.

```
Lock myLock;
…
public void myMethod() {
    myLock.lock();
    … // Do some not concurrent tasks
    myLock.unlock();
}
```

Debería definirse una clase que implemente esta interfaz. Afortunadamente, Java proporciona esa clase: ***ReentrantLock*** (del mismo paquete), por lo que se puede usar esta clase directamente para crear el bloqueo:

```
Lock myLock = new ReentrantLock();
```

Ejercicio opcional 1

Crear un proyecto llamado ***BankAccountLock***, que es una copia del proyecto creado en el ejercicio 7 (*BankAccountSynchronized*). Reemplaza los métodos sincronizados por mecanismos *Lock* que acabas de ver, y comprobar que todo funciona bien.

3.1. Bloqueos de lectura / escritura

Además, hay una mejora aportada por esta interfaz *Lock*: la posibilidad de que las operaciones de lectura y escritura funcionen por separado, de modo que pueda haber varias operaciones de lectura ejecutándose al mismo tiempo en un archivo o recurso determinado, pero solo una operación de escritura (cuando un hilo está escribiendo, nadie más puede leer o escribir). Se puede conseguir esto con la interfaz ***ReadWriteLock*** y su implementación en la clase ***ReentrantReadWriteLock***. Esta clase tiene dos bloqueos, uno para operaciones de lectura y otro para operaciones de escritura, por lo que se puede usar cualquiera de ellos dependiendo de la operación que se quiera hacer.

```
ReadWriteLock lock = new ReentrantReadWriteLock();
…
public void readOperation() {
    lock.readLock().lock();
    … // This area can be achieved by multiple reading threads
    lock.readLock().unlock();
}

public void writeOperation() {
    lock.writeLock().lock();
    … // When a writing thread gets here, no one else can have the lock
    lock.writeLock().unlock();
}
```

Como puedes ver en el código a continuación, *readLock()* permite bloquear un objeto para lectura, de modo que cualquier otra operación de lectura pueda llegar también a la sección crítica. Sin embargo, cuando se configura un bloqueo de escritura, no se podrá aplicar ningún otro bloqueo. En otras palabras, se puede tener varios lectores ejecutando la sección crítica al mismo tiempo, pero siempre que un escritor ejecuta la sección crítica, ningún otro hilo puede ejecutarla.

Observa cómo funciona con el siguiente ejemplo: se creará una clase que almacene un valor entero:

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class MyData {
    int value;
    ReentrantReadWriteLock lock;

    public MyData(int value) {
        this.value = value;
        lock = new ReentrantReadWriteLock();
    }

    public int getValue() {
        lock.readLock().lock();
        try {
            Thread.sleep(2000);
        } catch (Exception e) { }

        System.out.println("Thread #" + Thread.currentThread().getId() +
                " reads value " + value);
        int v = value;
        lock.readLock().unlock();
```

```
        return v;
    }

    public void setValue(int value) {
        lock.writeLock().lock();
        try {
            Thread.sleep(2000);
        } catch (Exception e) { }

        System.out.println("Thread #" + Thread.currentThread().getId() +
                " sets value to " + (this.value + value));
        this.value += value;
        lock.writeLock().unlock();
    }
}
```

A continuación, define una clase hilo que intente leerlo y una clase de hilo que intente cambiar el valor:

```
public class ReadingThread extends Thread {
    MyData sharedData;

    public ReadingThread(MyData sharedData) {
        this.sharedData = sharedData;
    }
    @Override
    public void run() {
        int value = sharedData.getValue();
    }
}
```

```
public class WritingThread extends Thread {
    MyData sharedData;

    public WritingThread(MyData sharedData) {
        this.sharedData = sharedData;
    }

    @Override
    public void run() {
        sharedData.setValue(10);
    }
}
```

Si ejecutas un *main* como el siguiente...

```
public static void main(String[] args) {
    MyData mds = new MyData(10);
    ReadingThread[] threadsR = new ReadingThread[5];
    WritingThread[] threadsW = new WritingThread[2];

    for (int i = 0; i < threadsW.length; i++) {
        threadsW[i] = new WritingThread(mds);
    }

    for (int i = 0; i < threadsR.length; i++) {
        threadsR[i] = new ReadingThread(mds);
    }

    for (int i = 0; i < threadsW.length; i++) {
        threadsW[i].start();
    }
```

```
    for (int i = 0; i < threadsR.length; i++) {
        threadsR[i].start();
    }
}
```

Observarás que, todos los hilos de lectura imprimen sus resultados al mismo tiempo, mientras que los hilos de escritura imprimen sus mensajes por separado, después de 2 segundos. La salida puede ser algo como esto (puede diferir dependiendo del orden en el que los hilos comiencen):

```
Thread #13 sets value to 20
Thread #14 reads value 20
Thread #12 sets value to 30
Thread #18 reads value 30
Thread #15 reads value 30
Thread #17 reads value 30
Thread #16 reads value 30
```

Ejercicio opcional 2

Crea un proyecto llamado ***ReadersWritersLock*** y copia el ejemplo anterior en él. Realiza cambios en el código para que haya 10 hilos de lectura (en lugar de 5), y cada hilo (lector o escritor) dormirá un número aleatorio de segundos (entre 1 y 10), y luego hará su trabajo. De esta manera, debe haber algunas operaciones de lectura al principio, algunas en el medio de los dos escritos y algunas al final. Tu salida debería verse así:

```
Thread #20 reads value 10
Thread #12 sets value to 20
Thread #22 reads value 20
Thread #21 reads value 20
Thread #19 reads value 20
Thread #14 reads value 20
Thread #17 reads value 20
Thread #13 sets value to 30
Thread #15 reads value 30
…
```

4. El *framework Fork/Join*

El ejecutor de hilos que acabas de ver se añadió en Java 5, y permite separar la creación de hilos y su ejecución. Desde Java 7, se puede dar un paso más gracias al *framework Fork/Join*.

Con este *framewok*, se puede dividir problemas grandes o complejos en problemas más pequeños. Este *framework* se basa en dos operaciones: ***fork*** (dividir una tarea en tareas más pequeñas) y ***join*** (una tarea espera a que finalicen sus sub-tareas). Sin embargo, las tareas involucradas en el *framework Fork/Join* no tienen otro mecanismo de sincronización.

El *framework Fork/Join* se basa en dos clases: ***ForkJoinPool*** (gestionará las tareas y ofrecerá información sobre su ejecución) y ***ForkJoinTask*** (la clase base de cada tarea añadida a la *ForkJoinPool*). Esta clase tiene dos sub-clases implementadas:

RecursiveAction (para tareas que no devolverán ningún resultado) y ***RecursiveTask*** (para tareas que devolverán un resultado). Todas estas clases pertenecen al paquete ***java.util.concurrent***.

4.1. Ejemplo: tareas que no devuelven ningún resultado

Observa cómo se puede utilizar este *framework* con el siguiente ejemplo: se creará una lista de videojuegos, con sus títulos y precios. Luego, se buscará un título dado en la lista, de modo que, si el tamaño de la lista es menor a 5 videojuegos, solo será necesaria una tarea, pero si no, se creará una tarea para buscar un subconjunto de hasta 5 videojuegos de la lista.

La clase *VideoGame* sería como esta:

```
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;
    }
}
```

El hilo para buscar en la lista sería como sigue:

```
public class GameSearch extends RecursiveAction {
    /* How many video games will each task be in charge of? */
    public static final int MAX_GAMES = 5;
    /* List of video games */
    ArrayList<VideoGame> list;
    /* First index of the list to search */
    int first;
    /* Last index of the list to search */
    int last;
    /* Text to be searched in the list */
    String text;

    public GameSearch(ArrayList<VideoGame> list, String text, int first, int last) {
        this.list = list;
        this.text = text;
        this.first = first;
        this.last = last;
    }

    @Override
    protected void compute() {
        if (last - first <= MAX_GAMES)
            search();
        else {
            int middle = (last - first) / 2;
```

```
            System.out.println("Creating 2 subtasks...");
            GameSearch s1 = new GameSearch(list, text, first, middle + 1);
            GameSearch s2 = new GameSearch(list, text, middle + 1, last);
            invokeAll(s1, s2);
        }
    }

    public void search() {
        for (int i = first; i < last; i++) {
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (Exception e) { }

            if (list.get(i).getTitle().contains(text))
                System.out.println("Found at position " + i + ": " + list.get(i).getTitle());
        }
    }
}
```

Fíjate que, cuando se extiende de la clase *RecursiveAction*, se necesita definir un método *compute()*. Este sería el equivalente al método *run()* en hilos comunes. Dentro de este método, se comprueba el tamaño de la lista de juegos. Si es menor que 5, simplemente se llama al método *search()* para resolver el problema. De lo contrario, se divide la lista en dos partes y se crean dos sub-tareas; cada una se encargará de buscar en su mitad de la lista.

También se puede crear una lista de tareas y llamar al método *invokeAll()* al que se le pasa esa lista como parámetro:

```
ArrayList<GameSearch> subtasks = new ArrayList<>();
…
subtasks.add(new GameSearch(…));
subtasks.add(new GameSearch(…));
subtasks.add(new GameSearch(…));

invokeAll(subtasks);
```

Desde el programa principal, se creará la lista de videojuegos, una tarea *GameSearch* para buscar la palabra *“Assassin’s”* y ejecutarla mediante *Fork/Join*, como se hizo antes con los ejecutores de hilos:

```
public static void main(String[] args) {
    ArrayList<VideoGame> list = new ArrayList<VideoGame>();
    list.add(new VideoGame("Assassin's Creed", 19.95f));
    list.add(new VideoGame("The last of us", 49.90f));
    list.add(new VideoGame("Fifa 14", 39.95f));
    list.add(new VideoGame("Far Cry 2", 14.95f));
    list.add(new VideoGame("Watchdogs", 59.95f));
    list.add(new VideoGame("Assassin's Creed II", 24.90f));
    list.add(new VideoGame("Far Cry 3", 39.50f));
    list.add(new VideoGame("Borderlands", 19.90f));

    GameSearch v = new GameSearch(list, "Assassin's", 0, list.size());
    ForkJoinPool pool = new ForkJoinPool();
    pool.execute(v);

    do {
        try {
            Thread.sleep(100);
        } catch (Exception e) { }
    } while (!v.isDone());
```

```
    pool.shutdown();
}
```

El programa principal tiene que esperar hasta que finalice la tarea (usando su método *isDone()*), antes de terminar.

4.2. Ejemplo: tareas que devuelven un resultado

¿Cómo se podría adaptar el ejemplo anterior para que las tareas no impriman nada en la salida y devuelvan un conjunto o lista de resultados encontrados? Hay que utilizar una subclase de *RecursiveTask* en lugar de una subclase de *RecursiveAction*. Cuando se extiende de *RecursiveTask*, debe tenerse en cuenta que es una clase parametrizada, es decir, es necesario proporcionar el tipo de resultado que se devolverá. Por lo que la clase *GameSearch* del ejemplo anterior será como se muestra ahora:

```
public class GameSearch extends RecursiveTask<ArrayList<String>> {
    /* How many video games will each task be in charge of? */
    public static final int MAX_GAMES = 5;
    /* List of video games */
    ArrayList<VideoGame> list;
    /* First index of the list to search */
    int first;
    /* Last index of the list to search */
    int last;
    /* Text to be searched in the list */
    String text;

    public GameSearch(ArrayList<VideoGame> list, String text, int first, int last) {
        this.list = list;
        this.text = text;
        this.first = first;
        this.last = last;
    }

    @Override
    protected ArrayList<String> compute() {
        ArrayList<String> results = new ArrayList<String>();
        if (last - first <= MAX_GAMES)
            results = search();
        else {
            int middle = (first + last) / 2;
            System.out.println("Creating 2 subtasks...");
            GameSearch s1 = new GameSearch(list, text, first, middle + 1);
            GameSearch s2 = new GameSearch(list, text, middle + 1, last);
            invokeAll(s1, s2);
            try {
                results = s1.get();
                ArrayList<String> aux = s2.get();
                results.addAll(aux);
            } catch (Exception e) { }
        }
        return results;
    }

    public ArrayList<String> search() {
        ArrayList<String> results = new ArrayList<String>();
        for (int i = first; i < last; i++) {
            try {
                TimeUnit.SECONDS.sleep(1);
```

```
            } catch (Exception e) { }

            if (list.get(i).getTitle().contains(text))
                results.add("Found at " + i + ": " + list.get(i).getTitle());
        }
        return results;
    }
}
```

Se devolverá un *ArrayList* de *strings* como resultado, cada uno contiene cada ocurrencia del texto buscado. En el método *search()*, simplemente se creará la lista de juegos que coinciden con el texto y se devuelve. En el método *compute()*, se llama al método *search()* directamente si hay menos de 5 juegos para buscar, o se divide el trabajo en dos tareas y se unen sus resultados en el bloque *try…catch* (llamando al método *get()* se pueden producir excepciones).

El programa principal obtendrá los resultados una vez finalizada la tarea principal y los imprimirá por la salida estándar:

```
public static void main(String[] args) {
    …
    /* Main method is the same until we call the shutdown method, then we
    need to add some lines to get and print the results */
    try {
        ArrayList<String> results = v.get();
        for (int i = 0; i < results.size(); i++)
            System.out.println(results.get(i));
    } catch (Exception e) {
        System.out.println("Exception occurred: " + e.getMessage());
    }
    pool.shutdown();
}
```

Ejercicio opcional 3

Crea un proyecto llamado ***ForkJoinFile***. Crea un archivo de texto en él, con algunas líneas (al menos 50, puede copiarlas desde cualquier fuente). Utiliza el *framework Fork/Join* para crear tareas que verifiquen el contenido del archivo de texto (hasta 10 líneas para cada tarea). Las tareas deben reemplazar todas las apariciones de la palabra *"java"* por *"Java"* (por supuesto, intenta añadir la palabra *"java"* en el archivo de texto varias veces). Al final, el programa principal obtendrá los resultados de todas las sub-tareas (es decir, los fragmentos de texto con los reemplazos realizados), los unirá y reescribirá el archivo de texto con el texto actualizado.

4.3. Lanzar de sub-tareas asincrónicas

En los ejemplos que se han mostrado arriba, cuando se llamaba al método *invokeAll()*, la tarea que los llamaba esperaba hasta que las sub-tareas invocadas terminasen su trabajo. Es decir, se utiliza una forma síncrona de llamar tareas y sub-tareas.

También se puede llamar a las sub-tareas de forma asincrónica (es decir, la tarea principal continúa su trabajo mientras lanza las sub-tareas), utilizando los métodos ***fork*** y ***join***, en lugar de *invokeAll()* y otros métodos sincrónicos. Por lo que el código anterior del método *compute()* se podría cambiar a uno asincrónico de la siguiente manera:

```
@Override
protected void compute() {
    if (last - first <= MAX_GAMES)
        search();
    else {
        int middle = (first + last) / 2;
        System.out.println("Creating 2 subtasks...");
        GameSearch s1 = new GameSearch(list, text, first, middle + 1);
        GameSearch s2 = new GameSearch(list, text, middle + 1, last);
        s1.fork();
        s2.fork();
        // At this point, this task continues running its code
        // ...
        // Wait for the 1st subtask to finish
        s1.join();
        // ...
        // Wait for the 2nd subtask to finish
        s2.join();
    }
}
```