Introducción a la Programación Concurrente en Java: Hilos

Agustín Alejandro Mota Hinjosa

10 de noviembre de 2023

1. Concepto de un Hilo

Normalmente, podemos definir subprocesos como un subproceso liviano con la unidad de proceso más pequeña y que también tiene rutas de ejecución separadas. Estos subprocesos usan memoria compartida pero actúan de forma independiente, por lo tanto, si hay una excepción en los subprocesos que no afectan el funcionamiento de otros subprocesos a pesar de que comparten la misma memoria.

1.1. Multitarea

Para ayudar a los usuarios, el sistema operativo les brinda el privilegio de realizar múltiples tareas, donde los usuarios pueden realizar múltiples acciones simultáneamente en la máquina. Esta Multitarea se puede habilitar de dos maneras:

- Multitarea basada en procesos
- Multitarea basada en subprocesos

1.1.1. Multitarea basada en procesos (multiprocesamiento)

En este tipo de multitarea, los procesos son pesados y a cada proceso se le asignó un área de memoria separada. Y como el proceso es pesado, el costo de comunicación entre procesos es alto y lleva mucho tiempo cambiar entre procesos, ya que implica acciones como cargar, guardar en registros, actualizar mapas, listas, etc.

1.1.2. Multitarea basada en subprocesos

Como comentamos anteriormente, los subprocesos son livianos y comparten el mismo espacio de direcciones, y el costo de comunicación entre subprocesos también es bajo.

2. Ciclo de vida del hilo

Hay diferentes estados a los que se transfiere el hilo durante su vida; háganos saber acerca de esos estados en las siguientes líneas: durante su vida, un hilo pasa por los siguientes estados, a saber:

- Nuevo Estado - Estado activo - Estado de espera/bloqueado - Estado de espera cronometrado - Estado terminado

2.1. Nuevo Estado

De forma predeterminada, un subproceso estará en un estado nuevo; en este estado, el código aún no se ha ejecutado y el proceso de ejecución aún no se ha iniciado.

2.2. Estado activo

Un subproceso que es un estado nuevo por defecto se transfiere al estado Activo cuando invoca el método start(), su estado Activo contiene dos subestados, a saber:

Estado ejecutable: en este estado, el subproceso está listo para ejecutarse en cualquier momento dado y es trabajo del Programador de subprocesos proporcionar el tiempo del subproceso para los subprocesos conservados en estado ejecutable. Un programa que ha obtenido subprocesos múltiples comparte intervalos de tiempo que se comparten entre subprocesos, por lo tanto, estos subprocesos se ejecutan durante un breve lapso de tiempo y esperan en el estado ejecutable para obtener su intervalo de tiempo programado. Estado de ejecución: cuando el subproceso recibe la CPU asignada por el Programador de subprocesos, se transfiere del estado .^{Ej}ecutable.^{a}l estado .^{En} ejecución". y después de la expiración de su sesión de intervalo de tiempo determinada, vuelve nuevamente al estado .^{Ej}ecutableτ espera su siguiente intervalo de tiempo.

2.3. Estado de espera/bloqueado

Si un hilo está inactivo pero en un tiempo temporal, entonces está en estado de espera o bloqueado, por ejemplo, si hay dos hilos, T1 y T2, donde T1 necesita comunicarse con la cámara y el otro hilo, T2, ya usa una cámara. para escanear, T1 espera hasta que T2 Thread complete su trabajo, en este estado T1 está estacionado esperando el estado y, en otro escenario, el usuario llamó a dos Threads T2 y T3 con la misma funcionalidad y ambos tuvieron el mismo intervalo de tiempo proporcionado por Thread. El Programador entonces ambos subprocesos T1 y T2 están en un estado bloqueado. Cuando hay varios subprocesos estacionados en estado Bloqueado/En espera, el Programador de subprocesos borra la cola rechazando subprocesos no deseados y asignando CPU con prioridad.

2.4. Estado de espera cronometrado

A veces, la mayor duración de la espera de los subprocesos provoca inanición, si tomamos un ejemplo como que hay dos subprocesos T1, T2 esperando la CPU y T1 está experimentando una operación de codificación crítica y si no existe la CPU hasta que se ejecute su operación, entonces T2 Estará expuesto a esperas más largas con certeza indeterminada. Para evitar esta situación de inanición, tuvimos una espera programada para que el estado evite ese tipo de escenario, ya que en la espera programada, cada subproceso tiene un período de tiempo durante el cual se invoca el método sleep(). y una vez transcurrido el tiempo, Threads comienza a ejecutar su tarea.

2.5. Estado terminado

Un hilo estará en estado Terminado, debido a las siguientes razones:

La terminación la logra un Thread cuando finaliza su tarea normalmente. A veces, los subprocesos pueden finalizar debido a eventos inusuales como fallas de segmentación, excepciones, etc. y este tipo de Terminación puede denominarse Terminación Anormal. Un hilo terminado significa que está muerto y ya no está disponible.

2.5.1. Qué es el hilo principal?

Como sabemos, creamos un método principal en todos y cada uno de los programas Java, que actúa como un punto de entrada para que JVM ejecute el código. De manera similar, en este concepto de subprocesos múltiples, cada programa tiene un subproceso principal que JVM proporcionó de forma predeterminada. Por lo tanto, cada vez que se crea un programa en Java, JVM proporciona el hilo principal para su ejecución.

2.6. ¿Cómo crear subprocesos utilizando el lenguaje de programación Java?

Podemos crear subprocesos en Java de dos formas, a saber:

2.7. Clase de hilo extendida

Implementación de una interfaz ejecutable 1. Heredando la clase de subproceso

Podemos ejecutar Threads en Java usando Thread Class, que proporciona constructores y métodos para crear y realizar operaciones en un Thread, que extiende una clase Thread que puede implementar Runnable Interface. Usamos los siguientes constructores para crear el hilo:

- Hilo
- Hilo (ejecutable r)

- Hilo (nombre de cadena)
- Hilo (Runnable r, nombre de cadena)

3. Ejemplo de creación de un hilo

```
import java.io.*;
import java.util.*;

public class GFG extends Thread {
    // initiated run method for Thread
    public void run()
    {
        System.out.println("Thread Started Running...");
    }
    public static void main(String[] args)
    {
        GFG g1 = new GFG();

        // Invoking Thread using start() method
        g1.start();
    }
}
```

Implementando la intefaz:

```
import java.io.*;
import java.util.*;

public class GFG implements Runnable {
    // method to start Thread
    public void run()
    {
        System.out.println(
            "Thread is Running Successfully");
    }

    public static void main(String[] args)
    {
        GFG g1 = new GFG();
        // initializing Thread Object
        Thread t1 = new Thread(g1);
        t1.start();
    }
}
```

4. Ejemplos prácticos

4.1. Tareas Concurrentes Simples

```
class Tarea implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Tarea en ejecución: " + i);
            try {
                Thread.sleep(1000); // Simula una tarea que toma tiempo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class EjemploHilos {
    public static void main(String[] args) {
        // Crear dos hilos que ejecutan la misma tarea
        Thread hilo1 = new Thread(new Tarea());
        Thread hilo2 = new Thread(new Tarea());

        // Iniciar los hilos
        hilo1.start();
        hilo2.start();
    }
}
```

4.2. Sincronización de Hilos

```
class Contador {
    private int count = 0;

    public synchronized void increment() {
        for (int i = 0; i < 5; i++) {
            count++;
            System.out.println(Thread.currentThread().getName() + ": " + count);
        }
    }
}

public class EjemploSincronizacion {
    public static void main(String[] args) {
        Contador contador = new Contador();
```

```
        // Crear dos hilos que incrementan el contador simultáneamente
        Thread hilo1 = new Thread(() -> contador.increment());
        Thread hilo2 = new Thread(() -> contador.increment());

        // Iniciar los hilos
        hilo1.start();
        hilo2.start();
    }
}
```

4.3. Productor-Consumidor con Hilos

```
import java.util.LinkedList;

class Buffer {
    private LinkedList<Integer> queue = new LinkedList<>();
    private final int CAPACIDAD = 2;

    public void producir() throws InterruptedException {
        int valor = 0;
        while (true) {
            synchronized (this) {
                while (queue.size() == CAPACIDAD) {
                    wait();
                }
                System.out.println("Productor produce: " + valor);
                queue.add(valor++);
                notify();
                Thread.sleep(1000);
            }
        }
    }

    public void consumir() throws InterruptedException {
        while (true) {
            synchronized (this) {
                while (queue.isEmpty()) {
                    wait();
                }
                int valor = queue.poll();
                System.out.println("Consumidor consume: " + valor);
                notify();
                Thread.sleep(1000);
            }
        }
    }
```

```
}

public class ProductorConsumidor {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();

        // Crear un hilo productor y un hilo consumidor
        Thread hiloProductor = new Thread(() -> {
            try {
                buffer.producir();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread hiloConsumidor = new Thread(() -> {
            try {
                buffer.consumir();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        // Iniciar los hilos
        hiloProductor.start();
        hiloConsumidor.start();
    }
}
```

4.4. Pool de Hilos con ExecutorService

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

class TareaSimple implements Runnable {
    private int id;

    public TareaSimple(int id) {
        this.id = id;
    }

    public void run() {
        System.out.println("Tarea " + id + " en ejecución");
        try {
            Thread.sleep(2000); // Simula una tarea que toma tiempo
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
}

public class PoolDeHilos {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 5; i++) {
            Runnable tarea = new TareaSimple(i);
            executor.execute(tarea);
        }

        executor.shutdown();
    }
}
```