Concurrency, Multithreading and Parallel Computing in Java

Jacopo De Angelis

20 marzo 2022

Indice

Capitolo 1

Multhithreading theory

1.1 Processi e thread

I processi e i thread sono indipendenti tra di loro:

- i **processi** sono l'istanza di esecuzione: ogni processo ha registri indipendenti, uno stack di memoria e un heap per ogni procesos. In java si crea con **ProcessBuilder**.

- i **thread** sono l'unità di esecuzione di un processo: un processo può avere più thread, ogni thread condivide memoria e risorse.

1.2 Time slicing

Il tempo di funzionamento di un core è diviso tra i suoi thread, il modo nel quale il tempo è diviso tra di essi è deciso tramite un algoritmo di time slicing.

1.3 Vantaggi del multithreading

La possibilità di svolgere più lavori contemporaneamente senza dovere attendere la fine di uno per iniziare l'altro.

1.4 Svantaggi del multithreading

Problemi di sincronizzazione e di utilizzo delle risorse possono rendere il multithreading inefficiente. Il debug è più complicato.

1.5 Ciclo di vita di un thread

- instanziazione (new): quando viene creato
- in funzione (runnable): quando viene eseguito tramite start()
- in attesa (waiting): passaggio ad uno stato di attesa tramite wait() e sleep()
- fine vita (dead): quando il thread termina la sua funzione

Capitolo 2

Manipolazione dei thread

2.1 Interfaccia Runnable

Runnable porta ad implementare una funzione run() dentro alla quale viene posta la parte logica del thread. Il thread a questo punto viene eseguito tramite start()

2.2 Classe Thread

Estendendo la classe Thread si ottiene lo stesos risultato ma con il limite di non potere estendere altre classi, si ottengono però i suoi metodi.

2.3 join()

thread.join() attende la fine dell'esecuzione del thread. Serve per segnalare quando il thread muore.

2.4 Deamon e Worker

- **Deamon**: è un thread che vive durante tutta la vita del **processo**. Sono thread di aiuto, ad esempio il garbage collector.
- **Worker**: è un thread che vive durante la propria vita

Capitolo 3

Comunicazione tra thread

Le variabili locali vivono sullo stack, gli oggetti che sono instanziati vivono sull'heap. **Ogni thread ha il suo stack ma tutti i thread condividono l'heap.**

3.1 Sincronizzazione

Se due thread devono agire sullo stesso campo si crea un problema di accesso e scrittura su di esso. Per questo si usa l'accessore *synchronized* sul metodo, in questo modo solo un thread alla volta può chiamarlo.

Il problema è che definire così un metodo porta il thread ad acquisire il lock sulla classe, non sul metodo, quindi anche se ci fossero due metodi indipendenti synchronized, comunque non potrebbero essere eseguiti contemporaneamente. è meglio non mettere come synchronized il metodo ma un blocco all'interno del codice, ad esempio

```
public static void increment1() {
  synchronized(App.class) {
    counter++
  }
}
```

Questo comunque non risolve il problema del lock sulla classe, per quello possiamo usare un oggetto come lock:

```
public class App {
  public static int counter1 = 0;
  public static int counter2 = 0;

  private static final Object lock1 = new Object();
  private static final Object lock2 = new Object();

  public static void increment1() {
    synchronized(lock1) {
      counter1++
    }
  }

  public static void increment2() {
    synchronized(lock2) {
      counter2++
    }
  }
```

In questo modo si ha un lock a livello di oggetto.

3.2 wait() e notify()

Due thread con lo stesso lock possono comunicare tra di loro con due metodi:

- wait(): il thread entra in stato di attesa fino a quando il token non viene rilasciato
- notify(): rilascia il lock e avvisa la thread pool che può essere preso

3.3 Reentrant lock

Un thread non può acquisire un lock più volte a meno che questo non sia un ReentrantLock

Capitolo 4

Concetti di multithreading

4.1 Volatile

volatile è un accesso che segnala di NON inserire all'interno della cache una variabile. Per quanto la cache sia più veloce, bisogna scegliere accuratamente cosa inserirci in quanto non si possono svolgere azioni di riorganizzazione automatica della memoria sulla cache, uno strumento di potenziamento delle performance.

4.2 Fermare un thread

Se si vuole interrompere un thread in maniera sicura non bisogna usare stop() ma interrupt().

4.3 Deadlock e Livelock

- **Deadlock**: quando due thread attendono che l'altro finisca e quindi nessuno prende il comando
- **Livelock**: due thread sono troppo impegnati a passare il controllo all'altro per eseguire le proprie funzioni.

Per gestirli è meglio usare tryLock() come metodo.

4.4 Variabili atomiche

Sono variabili che consentono di rendere l'accesso ad essere non interrompibile, in questo modo la sincronizzazione è fatta a livello di variabile.

4.5 Semafori

I semafori sono dei tipi astratti che sono usati per regolare l'accesso a risorse condivise. Possono essere:

- a conteggio: contiene un conteggio di risorse arbitrario
- binario: è acceso o spento

4.6 Mutex

Sono semafori binari, quindi un solo thread può accedere al blocco controllato dalla mutex.

Capitolo 5

Creazione di threads con l'Executor Framework

L'utilizzo delle thread pool e l'Executor Framework aiuta a rendere gestibile il multithreading. Il vantaggio delle thread pool è il riutilizzo dei thread, questo perchè la loro creazione è costosa. Ci sono 4 tipi di Executor:

- **SingleThreadExecutor**: ha un solo thread
- **FixedThreadPool(n)**: permette di decidere un numero n di thread, solitamente il numero di core della CPU
- **CachedThreadPool**: il numero di thread non è limitato, se sono tutti occupai e arriva un nuovo task la pool creerà un nuovo thread, se rimane in idle 60 secondi viene rimosso, è usato solitamente per compiti paralleli brevi
- **ScheduledExecutor**: Possiamo eseguire una data operazione e intervalli regolari o possiamo usarlo per definire un ritardo nell'esecuzione di certi task

5.1 Interfacce Runnable e Callable

- **Runnable**: run-and-forget. Dopo essere stata eseguita non ritorna un valore. Si chiama con executorService.execute()

- **Callable**<**T**>: ritorna un valore di tipo **Futuree**<**T**>. Si chiama con executorService.submit()

Capitolo 6

Concurrent Collections

Possiamo rendere una collezione sincronizzata tramite *Collections.syncronizedList(<lista>), Collections.syncronizedMap(<map>), Collections.syncronizedSet(<set>).*

6.1 CountDownLatch

Il countdown del Latch è legato al numero di esecuzioni di run vengono completate.

6.2 CyclicBarrier

Permette di costringere un set di thread di attendere fino a quando tutti sono arrivati alla barriera tramite await().

6.3 BlockingQueue

Ha come sempre una struttura FIFO ma in più è sincronizzata.

6.4 DelayQueue

è una BlockingQueue con l'interfaccia aggiuntiva Delay, quindi si può prendere un oggetto dalla coda solo se il delay è terminato.

6.5 PriorityBlockingQueue

è un'implementazione della BlockingQueue e una versione thread safe della PriorityQueue.

6.6 ConcurrentMap

Non è molto efficiente perchè così la mappa ottiene un lock intrinseco anche per azioni non concorrenziali. Possiamo, però, rendere una mappa sincronizzata a segmenti, dividendola così in n segmenti da m elementi, ognuno con un suo lock.

6.7 Exchanger

è un punto di scambio tra thread per scambiarsi elementi.