Azzolini Riccardo 2020-02-20

Thread

1 Programma, processo e thread

- Un **programma** è semplicemente un insieme di istruzioni (ad alto livello o in linguaggio macchina).
- Un **processo** è un programma in esecuzione. In particolare:
  - se i processi condividono lo stesso spazio degli indirizzi, vengono chiamati **processi leggeri** o **thread**;
  - invece, i processi che hanno ciascuno un proprio spazio degli indirizzi sono detti **processi pesanti**, o semplicemente processi.

In Java, i thread creano dei flussi di esecuzione concorrente all'interno del singolo processo rappresentato dal programma in esecuzione. Si indica allora con il termine **programmazione concorrente** la pratica di implementare programmi contenenti *più flussi di esecuzione* (cioè, appunto, thread).

2 Thread main

In Java, ogni programma in esecuzione è un thread. In particolare, il metodo `main` è associato al thread main.

Per accedere alle proprietà del thread main, è necessario ottenere un riferimento all'oggetto corrispondente, mediante il metodo `Thread.CurrentThread()`. Ad esempio:

```
public class ThreadMain {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.out.println("Thread corrente: " + t);
        t.setName("Mio Thread");
        System.out.println("Dopo cambio nome: " + t);
    }
}
```

L'output di questo programma è

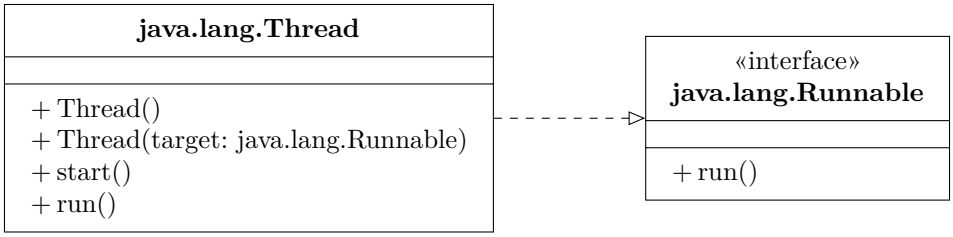
```
Thread corrente: Thread[main,5,main]
Dopo cambio nome: Thread[Mio Thread,5,main]
```

dove `Thread`[*nome*, *priorità*, *gruppo*] indica:

1. il nome del thread;
2. la priorità di scheduling;
3. il gruppo di appartenenza del thread.

3 Classe `Thread`

La classe principale per la gestione dei thread in Java è `java.lang.Thread`:



3.1 Estensione di `Thread`

Il modo più semplice per creare un thread è:

1. Estendere la classe `Thread`.
2. Nella sottoclasse, ridefinire (override) il metodo `run()`:[1] il codice che esso contiene (più eventuali altri metodi che questo invoca, direttamente o indirettamente) è quello che verrà eseguito in parallelo al codice degli altri thread.
3. Creare un'istanza della sottoclasse.
4. Richiamare il metodo `start()` sull'istanza creata.

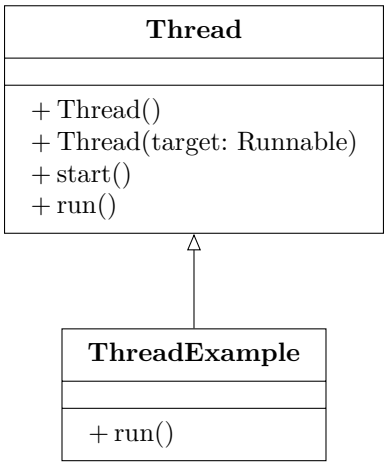
   Nota: Spesso si mette una chiamata `start()` direttamente nel costruttore, in modo che la creazione dell'istanza faccia anche già partire il thread.

[1] Nella classe `Thread`, il metodo `run()` è effettivamente vuoto. In realtà, esso contiene il codice per la gestione di un metodo alternativo di creazione di un thread (che verrà spiegato in seguito), ma tale codice non ha alcun effetto se si usa il costruttore di default di `Thread`.

3.1.1 Esempio

```java
public class ThreadExample extends Thread {
    public void run() {
        System.out.println("Running");
    }

    public static void main(String args[]) {
        ThreadExample t = new ThreadExample();
        t.start();
    }
}
```



3.2 Uso di `Runnable`

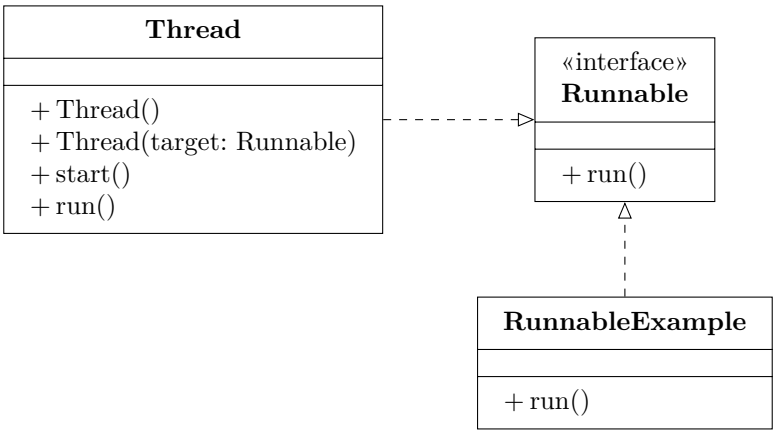
Un approccio alternativo alla creazione di un thread consiste nell'uso dell'interfaccia `java.lang.Runnable`:

1. Definire un'implementazione di `Runnable`.
2. Realizzare il metodo `run()` nella classe creata.
3. Creare un'istanza di questa classe.
4. Istanziare un nuovo `Thread`, passando al costruttore l'istanza della classe che implementa `Runnable` (creata al passo precedente).
5. Richiamare il metodo `start()` sull'istanza di thread.

3.2.1 Esempio

```
public class RunnableExample implements Runnable {
    public void run() {
        System.out.println("Running");
    }

    public static void main(String[] args) {
        RunnableExample r = new RunnableExample();
        Thread t = new Thread(r);
        t.start();
    }
}
```



4 Programmi concorrenti e sequenziali

Rispetto ai programmi sequenziali, con cui i programmatori hanno solitamente maggiore familiarità, i programmi concorrenti hanno delle proprietà molto diverse.

Ad esempio, un programma sequenziale eseguito ripetutamente con lo stesso input produce lo stesso risultato ogni volta, quindi eventuali bug saranno riproducibili. Lo stesso *non* vale, invece, per i programmi concorrenti, nei quali, infatti, il comportamento di un thread dipende fortemente dagli altri thread.

5 Metodi `run()` e `start()`

Se si invoca direttamente il metodo `run()` di un'istanza di `Thread`, il codice associato al thread viene eseguito in modo sequenziale (cioè *nello stesso thread* del chiamante).[2] Sono anche consentite più invocazioni di `run()`. Un esempio è

```
public class RunExample implements Runnable {
    public void run() {
        System.out.println("Ciao!");
    }

    public static void main(String[] args) {
        RunExample runnable = new RunExample();
        Thread t = new Thread(runnable);
        t.run();
        t.run();
    }
}
```

che produce in output:

```
Ciao!
Ciao!
```

Invece, il metodo `start()` avvia l'esecuzione concorrente del codice del thread. Esso può essere chiamato una sola volta: eventuali chiamate successive, anche dopo che il thread è terminato, generano un'eccezione di tipo `IllegalThreadStateException`. Ad esempio, il programma

```
public class StartExample implements Runnable {
    public void run() {
        System.out.println("Ciao!");
    }

    public static void main(String[] args) {
        StartExample runnable = new StartExample();
        Thread t = new Thread(runnable);
        t.start();
        t.start(); // genera IllegalThreadStateException
    }
}
```

[2] Chiaramente, non ha senso istanziare un `Thread` se si vuole eseguire il codice in modo sequenziale: solitamente, non c'è motivo di invocare direttamente il metodo `run()`. Piuttosto, è importante ricordare che, al contrario di quanto potrebbe suggerire il nome, `run()` non avvia l'esecuzione concorrente del thread.

produce in output:[3]

```
Ciao!
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.base/java.lang.Thread.start(Thread.java:790)
    at StartExample.main(StartExample.java:10)
```

Per eseguire un'altra volta lo stesso codice, è quindi necessario istanziare un nuovo `Thread`.

6 Flusso di controllo e thread daemon

Dal momento in cui un oggetto A crea e avvia un oggetto `Thread` B, si hanno due flussi di esecuzione concorrenti: l'oggetto A non aspetta che termini l'esecuzione dell'oggetto B.

Complessivamente, un programma non termina fintanto che ci sono thread *non-daemon* in esecuzione. In Java, un **thread daemon** è un particolare tipo di thread, che:

- ha tipicamente priorità molto bassa (esegue quando nessun altro thread dello stesso programma è in esecuzione);
- viene normalmente utilizzato come fornitore di servizi per i thread normali (l'esempio tipico è il garbage collector).

Quando gli unici thread in esecuzione in un programma sono daemon, la JVM li termina e arresta l'esecuzione del programma.

Per creare un thread daemon, si usa la chiamata `setDaemon(true)`, che viene tipicamente inserita nel costruttore del thread stesso.

6.1 Esempio

Il programma

```
public class DaemonThread extends Thread {
    public DaemonThread() {
        setDaemon(true);
    }

    public void run() {
```

[3] È anche possibile che la stampa di "Ciao!" e l'eccezione avvengano nell'ordine opposto, a seconda di quando il thread avviato dalla prima chiamata `start()` viene schedulato, rispetto al thread main (nel quale viene sollevata l'eccezione).

```
        int count = 0;
        while (true) {
            System.out.println("Hello " + count);
            count++;
            try {
                Thread.sleep(350);
            } catch (InterruptedException e) {}
        }
    }
}

public class DaemonExample {
    public static void main(String[] args) {
        DaemonThread dt = new DaemonThread();
        dt.start();
        try {
            Thread.sleep(750);
        } catch (InterruptedException e) {}
        System.out.println("Main thread ends.");
    }
}
```

produce in output:

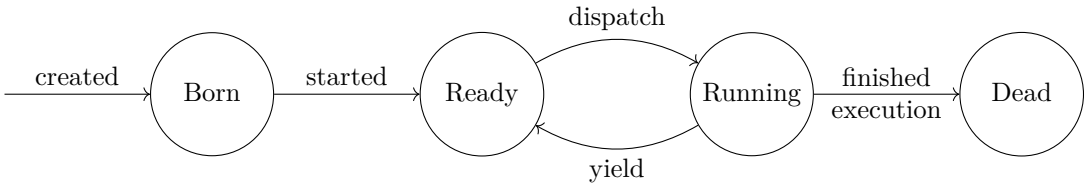
```
Hello 0
Hello 1
Hello 2
Main thread ends.
```

Se, invece, il thread creato non fosse un daemon, la sua esecuzione continuerebbe all'infinito, anche dopo la terminazione del thread main:

```
Hello 0
Hello 1
Hello 2
Main thread ends.
Hello 3
Hello 4
Hello 5
...
```

7 Stati di un thread

Semplificando, un thread si può trovare nei seguenti stati:



- Quando si invoca il metodo `start()` su un thread, esso non viene eseguito immediatamente, ma si porta solo nello stato **ready**.
- Successivamente, quando viene selezionato dallo **scheduler** il thread passa allo stato **running**, ed esegue il metodo `run()`, che è infatti l'*entry point* del thread (cioè il punto dal quale inizia l'esecuzione del codice).

  La prima volta che diventa running, un thread inizia a eseguire la prima istruzione di `run()`. Le volte successive, invece, l'esecuzione continua da dove era rimasta.
- Un thread è considerato **alive** finché il metodo `run()` non ritorna, poi diventa **dead** (e, come già detto, non può essere rieseguito: se ciò è necessario, bisogna invece creare una nuova istanza di `Thread`).

8 Scheduling

Il funzionamento esatto dello scheduler dipende dalla specifica piattaforma (sistema operativo, ecc.) sulla quale viene eseguita la JVM, ma, in generale, i thread Java vengono schedulati usando un algoritmo **preemptive** e **priority based**, cioè:

- tutti i thread hanno una *priorità*, e il thread con la priorità più alta tra quelli ready viene schedulato per essere eseguito;
- avendo il diritto alla *preemption*, lo scheduler può sottrarre forzatamente la CPU al thread che la sta usando (alla scadenza di un determinato quanto di tempo), per assegnarla a un altro thread.

Per la precisione, Java non specifica quale tipo di politica di scheduling debba essere adottata dalla macchina virtuale: essa varia in base al sistema operativo sottostante. Per verificare se la politica è preemptive, si può effettuare un test:

1. Creare due thread (entrambi con la stessa priorità):
   a) uno che procede indefinitamente, senza fare I/O né chiamate di sistema (cioè operazioni che possono far cedere la CPU a un altro);

b) uno che fa output.

2. Avviare per primo il thread a: se il sistema non è preemptive, questo non cederà mai la CPU, e allora il thread b non avrà mai modo di produrre output.

```
public class BusyThread extends Thread {
    public void run() {
        int a = 0;
        while (true) {
            a++;
        }
    }
}

public class OutputThread extends Thread {
    public void run() {
        String name = Thread.currentThread().getName();
        while (true) {
            System.out.println(name);
        }
    }
}

public class PreemptionTest {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("Main: inizio");
        Thread busy = new BusyThread();
        busy.start();
        // Dà tempo al primo thread di andare in esecuzione
        Thread.sleep(100);
        Thread out = new OutputThread();
        out.setName("Output");
        out.start();
    }
}
```

Con questo programma, in presenza di scheduling non preemptive, eseguirebbe solo il thread avviato per primo, quindi non sarebbe visibile l'output del secondo thread. Se, invece, venissero eseguiti entrambi, ciò indicherebbe che lo scheduling sia certamente preemptive.

Sulla maggior parte dei sistemi operativi moderni (ad esempio Windows e Linux), Java utilizza una politica preemptive, quindi il test produce l'output:

```
Main: inizio
```

```
Output
Output
Output
Output
Output
...
```

9 Esempio di nondeterminismo

Il seguente programma procedurale,[4] contenente due chiamate a `run()`,

```
import java.util.Random;

public class Procedural {
    private static final Random rand = new Random();

    private int num;

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

    public void run() {
        try {
            Thread.sleep(rand.nextInt(100));
            System.out.println("in run, num = " + num);
            Thread.sleep(rand.nextInt(100));
            System.out.println("in run, num = " + num);
        } catch (InterruptedException e) {}
    }

    public static void main(String[] args) {
        Procedural a = new Procedural(1);
        Procedural b = new Procedural(2);
        a.run();
        b.run();

        try {
            Thread.sleep(rand.nextInt(100));
            System.out.println("in main");
            Thread.sleep(rand.nextInt(100));
```

[4] Il termine *procedurale*, in questo caso, indica un programma con un singolo thread.

```
            System.out.println("in main");
        } catch (InterruptedException e) {}
    }
}
```

produce *sempre* lo stesso output:

```
in run, num = 1
in run, num = 1
in run, num = 2
in run, num = 2
in main
in main
```

Se, invece, si considera una versione concorrente di questo programma, che ha due thread (più il thread main),

```
import java.util.Random;

public class Concurrent extends Thread {
    private static final Random rand = new Random();

    private int num;

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

    public void run() {
        try {
            Thread.sleep(rand.nextInt(100));
            System.out.println("in run, num = " + num);
            Thread.sleep(rand.nextInt(100));
            System.out.println("in run, num = " + num);
        } catch (InterruptedException e) {}
    }

    public static void main(String[] args) {
        Concurrent a = new Concurrent(1);
        Concurrent b = new Concurrent(2);
        a.start();
        b.start();

        try {
```

```
            Thread.sleep(rand.nextInt(100));
            System.out.println("in main");
            Thread.sleep(rand.nextInt(100));
            System.out.println("in main");
        } catch (InterruptedException e) {}
    }
}
```

si possono ottenere sequenze di output diverse, a seconda dello scheduling dei thread. Ad esempio, due delle numerose possibili sequenze di output sono:

```
in main
in run, num = 2
in run, num = 1
in run, num = 1
in run, num = 2
in main

in main
in main
in run, num = 2
in run, num = 1
in run, num = 2
in run, num = 1
```

9.1 Possibile sequenza di esecuzione

L'output

```
in main
in run, num = 2
in run, num = 1
in run, num = 1
in run, num = 2
in main
```

potrebbe essere prodotto se le istruzioni dei thread vengono eseguite, ad esempio, in questa sequenza:

Stato main	Stato `a`	Stato `b`	Istruzione	Output
exec	new → ready	new	`a.start()`	
exec	ready	new → ready	`b.start()`	
exec → sleep	ready	ready → exec	sleep	
sleep	ready → exec	exec → sleep	sleep	
sleep	exec → sleep	sleep	sleep	
sleep	sleep	sleep		
sleep → ready → exec	sleep	sleep		
exec	sleep	sleep	stampa	`in main`
exec	sleep	sleep → ready		
exec → sleep	sleep	ready → exec	sleep	
sleep	sleep → ready	exec		
sleep	ready	exec	stampa	`in run, num = 2`
sleep	ready → exec	exec → sleep	sleep	
sleep	exec	sleep	stampa	`in run, num = 1`
sleep	exec → sleep	sleep	sleep	
sleep	sleep	sleep		
sleep	sleep → ready → exec	sleep		
sleep	exec	sleep	stampa	`in run, num = 1`
sleep	exec	sleep → ready		
sleep	exec → dead	ready → exec	*fine*	
sleep → ready	dead	exec		
ready	dead	exec	stampa	`in run, num = 2`
ready → exec	dead	exec → dead	*fine*	
exec	dead	dead	stampa	`in main`
exec → dead	dead	dead	*fine*	