



Lezione n.2
LPR-A-09
Thread Pooling

29/9/2009

V. Gervasi

ATTENDERE LA TERMINAZIONE DI UN THREAD: METODO `join()`

- Un thread `J` può invocare il metodo `join()` su un oggetto `T` di tipo thread
- `J` rimane sospeso sulla `join()` fino alla terminazione di `T`.
- Quando `T` termina, `J` riprende l'esecuzione con l'istruzione successiva alla `join()`.
- Un thread sospeso su una `join()` può essere interrotto da un altro thread che invoca su di esso il metodo `interrupt()`.
- Il metodo può essere utilizzato nel `main` per attendere la terminazione di tutti i threads attivati.

JOINING A THREAD

```
public class Sleeper extends Thread {
    private int period;
    public Sleeper (String name, int sleepPeriod){
        super(name);
        period = sleepPeriod;
        start( );
    }
    public void run( ){
        try{ sleep (period); }
        catch (InterruptedException e)
            {System.out.println(getName( )+" e' stato interrotto"); return;}
        System.out.println(getName()+" e' stato svegliato normalmente");}}}
```

JOINING A THREAD

```
public class Joiner extends Thread {
    private Sleeper sleeper;
    public Joiner(String name, Sleeper sleeper){
        super(name);
        this.sleeper = sleeper;
        start( ); }
    public void run( ){
        try{
            sleeper.join( );
        }catch(InterruptedException e) {
            System.out.println("Interrotto"); return; }
        System.out.println(getName( )+" join completed"); }}
```

JOINING A THREAD

```
public class Joining {  
    public static void main(String[ ] args){  
        Sleeper assonnato = new Sleeper("Assonnato", 1500);  
        Sleeper stanco = new Sleeper("Stanco", 1500);  
        new Joiner("WaitforAssonnato", assonnato);  
        new Joiner("WaitforStanco", stanco);  
        stanco.interrupt(); } }
```

Output:

Stanco è stato interrotto

WaitforStanco join completed

Assonnato è stato svegliato normalmente

WaitforAssonnato join completed

THREAD POOLING: CONCETTI FONDAMENTALI

- L'utente struttura l'applicazione mediante un insieme di tasks.
- **Task** = segmento di codice che può essere eseguito da un "esecutore". Può essere definito come un oggetto di tipo **Runnable**
- **Thread** = **esecutore** in grado di eseguire tasks.
- Uno stesso thread può essere utilizzato per eseguire diversi tasks, durante la sua vita.
- **Thread Pool** = Struttura dati (normalmente con dimensione massima prefissata), che contiene riferimenti ad un insieme di threads
- I thread del pool vengono utilizzati per eseguire i tasks sottomessi dall'utente.

THREAD POOLING: MOTIVAZIONI

- Tempo stimato per la creazione di un thread: qualche centinaio di microsecondi.
- La creazione di un alto numero di threads può non essere tollerabile per certe applicazioni.
- **Thread Pooling**
 - Diminuisce l'overhead dovuto alla creazione di un gran numero di thread: lo stesso thread può essere riutilizzato per l'esecuzione di più di un tasks
 - Permette una semplificazione e una migliore strutturazione del codice dell'applicazione: tutta la gestione dei threads può essere delegata al gestore del pool (che va può essere riutilizzato in altre applicazioni...)

THREAD POOLING: USO

L'utente

- Definisce i tasks dell'applicazione
- Crea un **pool di thread** e stabilisce una **politica per la gestione dei threads all'interno del pool**. La politica stabilisce:
 - **quando** i threads del pool **vengono attivati**: (al momento della creazione del pool, on demand, in corrispondenza dell'arrivo di un nuovo task,...)
 - **se e quando** è opportuno terminare l'esecuzione di un thread (ad esempio se non c'è un numero sufficiente di tasks da eseguire)
- Sottomette i tasks per l'esecuzione al pool di thread.

THREAD POOLING: IL GESTORE

- L'applicazione sottomette un task T al gestore del pool di thread
- Il gestore sceglie un thread dal pool per l'esecuzione di T.
Le scelte possibili sono:
 - utilizzare un thread attivato in precedenza, ma inattivo al momento dell'arrivo del nuovo task
 - creare un nuovo thread, purchè non venga superata la dimensione massima del pool
 - memorizzare il task in una struttura dati, in attesa di eseguirlo
 - respingere la richiesta di esecuzione del task
- Il numero di threads attivi nel pool può variare dinamicamente

LIBRERIA `java.util.concurrent`

- L'implementazione del thread pooling:
 - Fino a **J2SE 1.4** doveva essere realizzata a livello applicazione
 - **J2SE 5.0** introduce la libreria `java.util.concurrent` che contiene metodi per
 - Creare un pool di thread e il gestore associato
 - Definire la struttura dati utilizzata per la memorizzazione dei tasks in attesa
 - Decidere specifiche politiche per la gestione del pool

CREARE UN THREADPOOL EXECUTOR

Il package `java.util.concurrent` definisce:

- Alcune interfacce che definiscono servizi generici di esecuzione...

```
public interface Executor {  
    public void execute (Runnable task); }
```

```
public interface ExecutorService extends Executor{... }
```

- diverse classi che implementano `ExecutorService` (`ThreadPoolExecutor`, `ScheduledThreadPoolExecutor`, ...)
- la classe `Executors` che opera come una `Factory` in grado di generare oggetti di tipo `ExecutorService` con comportamenti predefiniti.

I tasks devono essere incapsulati in oggetti di tipo `Runnable` e passati a questi esecutori, mediante invocazione del metodo `execute()`

ESEMPI: IL TASK

```
public class TakeOff implements Runnable{
    int countDown = 3; // Predefinito
    public String status( ){
        return "#" + Thread.currentThread() +
            "(" + (countDown > 0 ? countDown: "Via!!!") + ")," ;
    }
    public void run( ) {
        while (countDown-- > 0){
            System.out.println(status());
            try{ Thread.sleep(100);}
            catch(InterruptedException e){ }
        }
    }
}
```

THREAD POOLING: ESEMPIO 1

```
import java.util.concurrent.*;

public class Esecutori1 {

    public static void main(String[ ] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i=0; i<3; i++) {
            exec.execute(new TakeOff( )); } } }
```

`newCachedThreadPool ()` crea un pool in cui quando viene sottomesso un task

- viene creato un nuovo thread se tutti i thread del pool sono occupati nell'esecuzione di altri tasks.
- viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente, se disponibile

Se un thread rimane inutilizzato per **60 secondi**, la sua esecuzione termina e viene rimosso dalla cache.

ESEMPIO1: OUTPUT

Output del programma:

```
#Thread[pool-1-thread-2,5,main](2)
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-3,5,main](2)
#Thread[pool-1-thread-3,5,main](1),
#Thread[pool-1-thread-2,5,main](1),
#Thread[pool-1-thread-1,5,main](1)
#Thread[pool-1-thread-2,5,main](Via!!!),
#Thread[pool-1-thread-1,5,main](Via!!!)
#Thread[pool-1-thread-3,5,main](Via!!!),
```

THREAD POOLING: ESEMPIO 2

```
import java.util.concurrent.*;
public class Esecutori2 {
    public static void main(String[] args){
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i=0; i<3; i++){
            exec.execute(new TakeOff( ));
            try {Thread.sleep (4000);}
                catch(InterruptedException e) { } } } }
```

La sottomissione di tasks al pool viene distanziata di 4 secondi. In questo modo l'esecuzione precedente è terminata ed è possibile riutilizzare un thread attivato precedentemente

ESEMPIO 2: OUTPUT

```
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-1,5,main](1),
#Thread[pool-1-thread-1,5,main](Via!!!),
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-1,5,main](1)
#Thread[pool-1-thread-1,5,main](Via!!!)
#Thread[pool-1-thread-1,5,main](2)
#Thread[pool-1-thread-1,5,main](1)
#Thread[pool-1-thread-1,5,main](Via!!!),
```


THREAD POOLING: ESEMPIO 3

```
import java.util.concurrent.*;

public class Esecutori3 {

    public static void main(String[] args){

        ExecutorService exec = Executors.newFixedThreadPool(2);

        for (int i=0; i<3; i++){

            exec.execute(new TakeOff());} } }
```

`newFixedThreadPool (int i)` crea un pool in cui, quando viene sottomesso un task

- Viene riutilizzato un thread del pool, se inattivo
- Se tutti i thread sono occupati nell'esecuzione di altri tasks, il task viene inserito in una coda, gestita dall'`ExecutorService` e condivisa da tutti i thread.

ESEMPIO 3: OUTPUT

```
#Thread[pool-1-thread-1,5,main](2),  
#Thread[pool-1-thread-2,5,main](2),  
#Thread[pool-1-thread-2,5,main](1),  
#Thread[pool-1-thread-1,5,main](1),  
#Thread[pool-1-thread-1,5,main](Via!!!),  
#Thread[pool-1-thread-2,5,main](Via!!!),  
#Thread[pool-1-thread-1,5,main](2),  
#Thread[pool-1-thread-1,5,main](1),  
#Thread[pool-1-thread-1,5,main](Via!!!),
```

THREAD POOL EXECUTOR

```
package java.util.concurrent;
```

```
public class ThreadPoolExecutor implements ExecutorService{
```

```
    public ThreadPoolExecutor ( int corePoolSize,
```

```
        int maximumPoolSize,
```

```
        long keepAliveTime,
```

```
        TimeUnit unit,
```

```
        BlockingQueue <Runnable> workqueue);
```

```
    ... }
```

- crea un oggetto di tipo `ExecutorService`
- consente di definire **gestori di thread pool** con una **politica di gestione personalizzata**

THREAD POOL EXECUTOR

```
public ThreadPoolExecutor ( int corePoolSize,  
                           int maximumPoolSize,  
                           long keepAliveTime,  
                           TimeUnit unit,  
                           BlockingQueue <Runnable> workqueue);  
... }
```

- **corePoolSize**, **maximumPoolSize**, e **keepAliveTime** controllano la gestione dei threads del pool
- **workqueue** è una struttura dati usata per memorizzare gli eventuali tasks in attesa di esecuzione

THREAD POOL: GESTIONE DINAMICA

- **corePoolSize**: dimensione minima del pool
 - È possibile allocare **corePoolSize** thread al momento della creazione del pool mediante il metodo **prestartAllCoreThreads()**. I thread creati rimangono inattivi in attesa di tasks da eseguire.
 - Oppure i thread possono essere creati "on demand". Quando viene sottomesso un nuovo task, viene creato un nuovo thread, anche se alcuni dei threads già creati sono inattivi. L'obiettivo è di riempire il "core" del pool prima possibile.
- **maximumPoolSize**: dimensione massima del pool

THREAD POOL: GESTIONE DINAMICA

- Se sono in esecuzione tutti i core thread, un nuovo task sottomesso viene inserito in una coda **Q**.
 - **Q** deve essere una istanza di **BlockingQueue<Runnable>**
 - **Q** viene passata al momento della costruzione del **ThreadPoolExecutor** (ultimo parametro del costruttore)
 - E' possibile scegliere diversi tipi di coda (sottoclassi di **BlockingQueue**). Il tipo di coda scelto **influisce sullo scheduling**.
- I task vengono poi prelevati da **Q** e inviati ai threads che si rendono disponibili
- Solo quando **Q** risulta piena si crea un nuovo thread attivando così **k** threads, **corePoolSize ≤ k ≤ maxPoolSize**

THREAD POOL: GESTIONE DINAMICA

Da questo punto in poi, quando viene sottomesso un nuovo task **T**

- se esiste un thread **th** inattivo, **T** viene assegnato a **th**
- se non esistono threads inattivi, si preferisce sempre accodare un task piuttosto che creare un nuovo thread
- solo se la coda è piena, si attivano nuovi threads
- Se la coda è piena e sono attivi *MaxPoolSize* threads, il task **viene respinto** e viene sollevata un'eccezione

THREAD POOL: GESTIONE DINAMICA

Supponiamo che un thread **th** termini l'esecuzione di un task, e che il pool contenga **k** threads

- Se **k** \leq **core**: il thread **si mette in attesa** di nuovi tasks da eseguire. L'attesa è indefinita.
- Se **k** $>$ **core**, si considera il **timeout** definito al momento della costruzione del thread pool
 - se nessun task viene sottomesso **entro il timeout**, **th** termina la sua esecuzione, riducendo così il numero di threads del pool
- Il **timeout** è determinato dai parametri **long keepAliveTime** e **TimeUnit unit** del costruttore, quindi consiste di:
 - un valore (es: 50000L) e
 - l'unità di misura utilizzata (es: **TimeUnit.MILLISECONDS**)

THREAD POOL: TIPI DI CODA

- **SynchronousQueue**: dimensione uguale a 0. Ogni nuovo task T
 - viene eseguito immediatamente oppure respinto.
 - T viene eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (numero di threads \leq **maxPoolSize**)
- **LinkedBlockingQueue**: dimensione illimitata
 - E' sempre possibile accodare un nuovo task, nel caso in cui tutti i tasks attivi nell'esecuzione di altri tasks
 - La dimensione del pool di **non supererà mai core**
- **ArrayBlockingQueue**: dimensione limitata, stabilita dal programmatore

THREAD POOLING: UN ESEMPIO

- Dato un intero **K**, si vuole calcolare, per ogni valore $n < K$ il valore dell'**n**-esimo numero di Fibonacci
- Si definisce un task **T** che effettua il calcolo del numero di Fibonacci di **n** (valore passato come parametro)
- Si attiva un **ThreadPoolExecutor**, definendo la politica di gestione del pool di thread mediante i parametri passati al costruttore
- Si passa all'esecutore una istanza di **T** per ogni $n < K$, invocando il metodo **execute()**

FIBONACCI TASK (I)

```
public class FibTask implements Runnable{
    int n; String id;
    public FibTask(int n, String id){
        this.n = n; this.id = id;
    }
    private int fib(int n){
        if (n == 0 || n == 1) return n;
        if (n==1) return 1;
        return fib(n - 1) + fib(n - 2);
    }
}
```

FIBONACCI TASK (II)

```
public void run( ){
    try{
        Thread t = Thread.currentThread( );
        System.out.println("Starting task " + id + " su " + n +
            " eseguito dal thread " + t.getName( ));
        System.out.println("Risultato " + fib(n) + " da task " + id +
            " eseguito dal thread " + t.getName( ));
    } catch (Exception e){
        e.printStackTrace();
    }
}
}
```

FIBONACCI THREAD POOL

```
import java.util.concurrent.*;

public class ThreadPoolTest {

    public static void main (String [] args){
        int nTasks = Integer.parseInt(args[0]); // # di tasks da eseguire
            // dimensione del core pool

        int corePoolSize = Integer.parseInt(args[1]);
            // massima dimensione del pool

        int maxPoolSize = Integer.parseInt(args[2]);

        ThreadPoolExecutor tpe = new ThreadPoolExecutor (corePoolSize,
            maxPoolSize,
            50000L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>( ));
```

FIBONACCI THREAD POOL

```
FibTask [ ] tasks = new FibTask[nTasks];
```

```
for (int i=0; i< nTasks; i++){
```

```
    tasks[i] = new FibTask(i, "Task" + i);
```

```
    tpe.execute(tasks[i]);
```

```
    System.out.println("dimensione del pool " + tpe.getPoolSize());
```

```
}
```

```
tpe.shutdown();
```

```
}
```

```
}
```

GESTIONE DINAMICA: ESEMPI

Parametri: tasks= 8, core = 3, MaxPoolSize= 4, SynchronousQueue, timeout=50000msec

dimensione del pool 1

Starting task Task0 eseguito da pool-1-thread-1

Risultato 0 da task Task0 eseguito da pool-1-thread-1

dimensione del pool 2

dimensione del pool 3

dimensione del pool 3

Starting task Task3 eseguito da pool-1-thread-1

Starting task Task1 eseguito da pool-1-thread-2

Starting task Task2 eseguito da pool-1-thread-3

dimensione del pool 4

Risultato 1 da task Task1 eseguito da pool-1-thread-2

Starting task Task4 eseguito da pool-1-thread-4

Risultato 2 da task Task3 eseguito da pool-1-thread-1

Risultato 1 da task Task2 eseguito da pool-1-thread-3

[java.util.concurrent.RejectedExecutionException](#)

Risultato 3 da task Task4 eseguito da pool-1-thread-4

GESTIONE DINAMICA: ESEMPI

Tutti I threads attivati inizialmente mediante `tpe.prestartAllCoreThreads();`

Parametri: tasks= 8, core = 3, MaxPoolSize= 4, SynchronousQueue

dimensione del pool 3

Starting task Task0 eseguito da pool-1-thread-3

dimensione del pool3

Risultato 0 da task Task0 eseguito da pool-1-thread-3

dimensione del pool 3

Starting task Task2 eseguito da pool-1-thread-1

Starting task Task1 eseguito da pool-1-thread-2

Risultato 1 da task Task2 eseguito da pool-1-thread-1

Risultato 1 da task Task1 eseguito da pool-1-thread-2

Starting task Task3 eseguito da pool-1-thread-3

dimensione del pool 3

Risultato 2 da task Task3 eseguito da pool-1-thread-3

dimensione del pool 3

GESTIONE DINAMICA: ESEMPI

(CONTINUA)

Starting task Task4 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task5 eseguito da pool-1-thread-3

Risultato 3 da task Task4 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task6 eseguito da pool-1-thread-1

Risultato 5 da task Task5 eseguito da pool-1-thread-3

dimensione del pool 3

Starting task Task7 eseguito da pool-1-thread-2

Risultato 8 da task Task6 eseguito da pool-1-thread-1

Risultato 13 da task Task7 eseguito da pool-1-thread-2

GESTIONE DINAMICA: ESEMPI

**Parametri: tasks= 10, core = 3, MaxPoolSize= 4, SynchronousQueue,
timeout=0msec**

dimensione del pool 1

Starting task Task0 eseguito da pool-1-thread-1

Risultato 0 da task Task0 eseguito da pool-1-thread-1

dimensione del pool 2

Starting task Task1 eseguito da pool-1-thread-2

Risultato 1 da task Task1 eseguito da pool-1-thread-2

dimensione del pool 3

dimensione del pool 3

Starting task Task2 eseguito da pool-1-thread-3

Starting task Task3 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task4 eseguito da pool-1-thread-1

Risultato 1 da task Task2 eseguito da pool-1-thread-3

Risultato 2 da task Task3 eseguito da pool-1-thread-2

dimensione del pool 4

GESTIONE DINAMICA:ESEMPI

(CONTINUA)

```
Risultato 3 da task Task4 eseguito dapool-1-thread-1
Starting task Task5 eseguito da pool-1-thread-4
dimensione del pool 3
Starting task Task6 eseguito da pool-1-thread-2
Risultato 5 da task Task5 eseguito dapool-1-thread-4
Starting task Task7 eseguito da pool-1-thread-1
dimensione del pool 3
Risultato 8 da task Task6 eseguito dapool-1-thread-2
dimensione del pool 3
Starting task Task8 eseguito da pool-1-thread-4
dimensione del pool 3
Starting task Task9 eseguito da pool-1-thread-2
Risultato 13 da task Task7 eseguito dapool-1-thread-1
Risultato 21 da task Task8 eseguito dapool-1-thread-4
Risultato 34 da task Task9 eseguito dapool-1-thread-2
```

GESTIONE DINAMICA: ESEMPI

Parametri: tasks= 10, core = 3, MaxPoolSize= 4,LinkedBlockingQueue

dimensione del pool 1

Starting task Task0 eseguito da pool-1-thread-1

Risultato 0 da task Task0 eseguito da pool-1-thread-1

dimensione del pool 2

dimensione del pool 3

Starting task Task1 eseguito da pool-1-thread-2

Risultato 1 da task Task1 eseguito da pool-1-thread-2

Starting task Task3 eseguito da pool-1-thread-2

dimensione del pool 3

Risultato 2 da task Task3 eseguito da pool-1-thread-2

dimensione del pool 3

Starting task Task2 eseguito da pool-1-thread-3

Starting task Task4 eseguito da pool-1-thread-1

Starting task Task5 eseguito da pool-1-thread-2

dimensione del pool 3

GESTIONE DINAMICA:ESEMPI

(CONTINUA)

```
Risultato 1 da task Task2 eseguito dapool-1-thread-3
Risultato 3 da task Task4 eseguito dapool-1-thread-1
Risultato 5 da task Task5 eseguito dapool-1-thread-2
dimensione del pool 3
Starting task Task6 eseguito da pool-1-thread-3
dimensione del pool 3
Starting task Task7 eseguito da pool-1-thread-1
Risultato 8 da task Task6 eseguito dapool-1-thread-3
dimensione del pool 3
Starting task Task8 eseguito da pool-1-thread-2
Risultato 13 da task Task7 eseguito dapool-1-thread-1
dimensione del pool 3
Starting task Task9 eseguito da pool-1-thread-3
Risultato 21 da task Task8 eseguito dapool-1-thread-2
Risultato 34 da task Task9 eseguito dapool-1-thread-3
```

TERMINAZIONE DI THREADS

- La JVM termina la sua esecuzione quando **tutti i thread (non demoni) terminano la loro esecuzione**
- Poiché un **ExecutorService** esegue i tasks in modo asincrono rispetto alla loro sottomissione, è necessario ridefinire il concetto di terminazione, nel caso si utilizzi un **ExecutorService**
- Un **ExecutorService** mette a disposizione del programmatore diversi metodi per effettuare lo 'shutdown' dei thread del pool
- La terminazione può avvenire
 - in **modo graduale**. Si termina l'esecuzione dei tasks già sottomessi, ma non si inizia l'esecuzione di nuovi tasks
 - in **modo istantaneo**. Terminazione immediata

TERMINAZIONE DI EXECUTORS

Alcuni metodi definiti dalla interfaccia **ExecutorService**

- **void shutdown()**
- **List<Runnable> shutdownNow()**
- **boolean isShutdown()**
- **boolean isTerminated()**
- **boolean awaitTermination(long timeout, TimeUnit unit)**

TERMINAZIONE DI EXECUTORS

- **void shutdown()**: graceful termination.
 - non accetta ulteriori task
 - i tasks sottomessi in precedenza vengono eseguiti, compresi quelli la cui esecuzione non è ancora iniziata (quelli accodati).
 - tutti i threads del pool terminano la loro esecuzione
- **List<Runnable> shutdownNow()**: immediate termination
 - non accetta ulteriori tasks,
 - elimina dalla coda i tasks la cui esecuzione non è ancora iniziata, e li restituisce in una lista
 - **tenta di terminare** l'esecuzione dei thread che stanno eseguendo i tasks (tipicamente con **interrupt()**): quindi non può garantire la terminazione dei thread.