Esercitazione di Sistemi Distribuiti e Pervasivi

Puntatori a strumenti moderni

Gabriele Civitarese

EveryWare Lab, Dept. of Computer Science, University of Milan

Premessa

- Nelle scorse lezioni abbiamo visto come gestire concorrenza e multi-threading in Java a grana fine (synchronized, wait, notify…)
- In questa lezione verranno presentate tecniche più recenti che astraggono su questi concetti
  - ...da **non** usare nel progetto!

Concurrency API

- Concurrency API (java.util.concurrent) presenti già da Java 5 e migliorate progressivamente
- Gestione dei problemi di concorrenza in ambito multi-thread semplificata e automatica
- Vedremo esempi in Java 8

Executors

- Gestire i thread usando direttamente la classe *Thread* è tedioso e spesso error-prone
- Java ci mette a disposizione degli *ExecutorService*, in modo da gestire task asincroni e pool di thread
- Non è più necessario far partire thread manualmente
- I task sono *Runnable* e vengono aggiunti tramite *submit()*

```
ExecutorService executor = Executors.newSingleThreadExecutor();


executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});
```

Tipi di executors

- **Fixed Thread Pool** - Riusa un numero fisso di thread che lavorano su una coda condivisa
- **Cached Thread Pool** - Crea nuovi thread quando serve, ma se sono disponibili vengono riusati thread precedentemente costruiti
- **Scheduled Thread Pool** - Schedula comandi da eseguire dopo un tempo predefinito, o da eseguire periodicamente.
- **Single Thread Executor** - Usa un solo worker thread che opera su una coda

Stoppare un executor

- Un *ExecutorService* va stoppato esplicitamente, altrimenti resta sempre in attesa di nuovi task da eseguire.
- Due modi:
  - *shutdown():* attende che i task in corso terminino
  - *shutdownNow():* termina interrompendo i task attualmente in esecuzione

Callables

- Oltre a *Runnable*, è anche possibile definire dei task *Callable*
  - la principale differenza è che **possono restituire un valore**
- Possono essere passati ad un *Executor*

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Future

- Quando si aggiunge un *Callable* ad un executor con *submit()*, questo non aspetta che il task sia terminato
- Viene restituito un oggetto *Future*
  - quando il task sarà completato restituirà il suo valore

```
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

Sincronizzazione

- L'uso di *Executor* non garantisce nulla rispetto alla sincronizzazione
- Ad esempio, cosa succede se definiamo

```
int count = 0;

void increment() {
    count = count + 1;
}
```

...e successivamente usiamo un *Executor* in questo modo?

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000).forEach(i -> executor.submit(this::increment));
```

Sincronizzazione (II)

- Se avete studiato, sapete che definendo il metodo *synchronized* il problema viene risolto...
  - ma le Concurrency API di Java ci possono semplificare ulteriormente la vita

```
ReentrantLock lock = new ReentrantLock();
int count = 0;

void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

Sincronizzazione (III)

- Le Concurrency API ci permettono quindi di avere strutture per la sincronizzazione
  - Lock
  - Read/write Lock
  - Semafori
  - ...
- Ma anche strutture dati già sincronizzate
  - ConcurrentMap
  - AtomicInteger
  - BlockingQueue
  - CopyOnWriteArrayList
  - ...

Conditions

- Visto che possiamo scegliere il nostro modo per gestire la sincronizzazione (reentrant/readwrite lock) astraiamo anche il concetto di *wait* e *notify*
- Anche in questo caso rimpiazziamo l'uso dell'intrinsic lock sfruttando oggetti delle Concurrency API
- I thread possono attendere su una condizione finchè un altro thread non li risveglia
- Analogo a wait e notify: *await()* e *signal()*
- Ogni condizione è legata ad uno specifico lock, e ne gestisce l'acquisizione e il rilascio ***atomico***

Conditions: example

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();
   ...
   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     } }
```

Conditions: example (cont)

```
…
public Object take() throws InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

Countdown latches

- Struttura usata per coordinare diversi thread
  - uno o più thread aspettano finchè le operazioni eseguite da un altro gruppo di thread non finisce
- Implementato come un contatore da decrementare
  - viene inizializzato con un intero N
  - un thread si mette in *await()*
  - altri N thread eseguono le proprie operazioni e alla fine decrementano il contatore
  - quando il contatore raggiunge 0, il thread in attesa si può risvegliare
  - Esempio di caso d'uso: attesa di N ack da altri processi

Fork/join

- Implementazione di *ExecutorService* per gestire il parallelismo hardware (multiprocessore)
  - tutta la capacità di processamento viene usata per velocizzare l'esecuzione
- Funzionamento ad alto livello:
  - divisione di task grandi in task più piccoli (*fork*)
  - ogni subtask viene processato in un singolo thread
  - i risultati intermedi vengono combinati (*join*)
- Uso di oggetti dedicati al posto di *Runnable* -> *RecursiveTask*

Fork/join: un esempio

```
public class MyRecursiveTask extends RecursiveTask<Long> {
   private long workLoad = 0;
   public MyRecursiveTask(long workLoad) {
        this.workLoad = workLoad;
    }
    protected Long compute() {
        if(this.workLoad > 16) {
            List<MyRecursiveTask> subtasks =  new ArrayList<MyRecursiveTask>();
            subtasks.addAll(createSubtasks());

            for(MyRecursiveTask subtask : subtasks){
                subtask.fork();
            }
            long result = 0;

            for(MyRecursiveTask subtask : subtasks) {
                result += subtask.join();
            }
            return result;
        } else {
                return workLoad * 3;
       }}
```

Fork/join: un esempio (cont)

...

```
private List<MyRecursiveTask> createSubtasks() {
        List<MyRecursiveTask> subtasks =
        new ArrayList<MyRecursiveTask>();

        MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);
        MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

        subtasks.add(subtask1);
        subtasks.add(subtask2);

        return subtasks;
    }
}
```

Fork/join: invocare il task

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);

MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);
```