Multithreading in Java
II parte

Lorenzo Gallucci

Synchronized

- Parola chiave introdotta per gestire la mutua esclusione
- Utile per proteggere l'accesso alle strutture dati "critiche"
- Induce una sincronizzazione ("serializzazione") tra i thread che tentano di accedere

Synchronized

- Come funziona?
  - Ogni istanza della classe Object e di sue sottoclassi possiede una propria "regione critica"
    - 1 Object ←→ 1 Regione critica
  - Sono Object:
    - Tutte le classi create dall'utente
    - Tutti gli array (di qualsiasi tipo essi siano)
    - Tutte le classi "di sistema" (ad es. Thread)
  - Per accedere ad essa il nostro codice deve utilizzare la parola chiave *synchronized*

Richiamo …

- Come garantire la correttezza?
- Modifichiamo l'Incrementer tramite l'uso della parola chiave synchronized

```
public class SynchronizedIncrementer implements Runnable
{
   private final int[] data;

   public SynchronizedIncrementer(int[] data)
   {
      this.data = data;
   }

   public void run()
   {
      synchronized (data)
      {
         for (int i = 0; i < data.length; i++)
         {
            data[i] = data[i] + 1;
         }
      }
   }
}
```

Synchronized

- Regione critica
  - Entrare nella regione critica significa cominciare ad eseguire il codice avviluppato nel blocco synchronized
  - In un dato istante, la regione critica può essere controllata al più da un solo thread
    - In tal caso, può essere presente una lista di thread che attendono “pazientemente” il loro turno
  - La regione critica può anche essere libera
    - In tal caso, l’accesso sarà consentito al primo thread che lo richiede

Attività speciali nel blocco synchronized

- Una volta entrato in una sezione critica, un thread può anche decidere, non trovando condizioni favorevoli, di "ibernarsi"
  - Per far ciò, si può usare il metodo wait()
- Il risveglio potrà avvenire su sollecitazione di altri thread, se vi è possibilità che le condizioni siano cambiate
  - Il metodo notify() consente di svegliare il primo thread in attesa (se ve n'è uno)
  - Il metodo notifyAll(), invece, li sveglia TUTTI

Synchronized: sintassi speciali

- È anche possibile dichiarare un intero metodo come synchronized
  - Se il metodo è statico, la sincronizzazione avviene sull'oggetto che descrive la classe nel suo complesso (ne esiste uno per ogni oggetto creato, v. Class)
  - Se il metodo non è statico, la sincronizzazione avviene sulla particolare istanza considerata

Pro e contro di synchronized

- Controindicazione: basse prestazioni
  - In condizioni di elevata contesa fra thread, le prestazioni scendono drasticamente
  - L'aumento del costo di gestione delle code di thread "sospesi" può essere micidiale
- Pregio: semplicità, integrazione nel linguaggio
  - L'entrata e l'uscita dal blocco synchronized sono automatiche
  - Il costrutto è conciso
  - Il monitor è incluso in ogni oggetto creato

Concorrenza in Java5: regioni critiche

- La versione 1.5 di Java ha affiancato, al meccanismo dei synchronized, una nuova serie di API basate sul concetto di compare-and-swap
- Si tratta di un'operazione atomica implementata nei moderni processori
- Ricordate l'Incrementer della volta scorsa?

Sincronizzazione: un caso pratico (RICHIAMO)

- Si abbia la classe:

```
public class Incrementer implements Runnable
{
   private final int[] data;

   public Incrementer(int[] data)
   {
      this.data = data;
   }

   public void run()
   {
      for (int i = 0; i < data.length; i++)
      {
         data[i] = data[i] + 1;
      }
   }

}
```

- Scopo della classe: ad ogni invocazione del metodo run(), tutti gli interi dell'array vengono incrementati

Compare-and-swap

- L'operazione "critica" è:

```
data[i] = data[i] + 1;
```

- Tra la lettura del valore di data[i] e l'impostazione del nuovo valore passa un certo lasso di tempo
  - Quando reimpostiamo il valore nell'array, ad esso potrebbe essere successa qualsiasi cosa!

- È importante controllare quanto vale data[i] prima di reimpostarlo!

Compare-and-swap

- Disponendo di un metodo:

```
public boolean changeAtomically(int[] data, int pos, int oldValue, int newValue)
  {
     if (data[pos] == oldValue)
     {
        data[pos] = newValue;
        return true;
     }
     return false;
  }
```

Potremmo scrivere il nostro incremento come:

```
public void increment(int[] data, int pos)
  {
     while (true)
     {
        int oldValue = data[pos];
        if (changeAtomically(data, pos, oldValue, oldValue + 1)) return;
     }
  }
```

Note:

- L'operazione nel blocco for(;;) {…} può essere ripetuta diverse volte, a seconda di quanti sono i thread che tentano di eseguire la medesima operazione; comunque l'attesa è LIMITATA
- changeAtomically() è *realmente disponibile* come operazione atomica dei moderni processori!!

Atomic* in Java5

- Sulla scorta dell'operazione compare-and-swap, in Java sono stati aggiunti, recentemente:
  - Una serie di contenitori e strumenti per operare sui valori di un programma (int, long, array, oggetti generici, ecc.) in maniera atomica
    - AtomicInteger, AtomicBoolean, AtomicLong, ecc.
    - Ma anche: AtomicIntegerArray, AtomicLongArray, ecc.
- Sui meccanismi di base degli Atomic* sono state poi costruite diverse classi che supportano la programmazione concorrente
  - Ad esempio, alcune classi di base come: Lock
  - Ma anche metafore più complesse, come: ConcurrentLinkedList

Atomic* in Java5

- Incrementer con AtomicInteger:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerIncrementer implements Runnable
{
   private final AtomicInteger[] data;

   public AtomicIntegerIncrementer(AtomicInteger[] data)
   {
      this.data = data;
   }

   public void run()
   {
      for (int i = 0; i < data.length; i++)
      {
         data[i].incrementAndGet();
      }
   }
}
```

Atomic* in Java5

- Incrementer alternativo:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AlternateAtomicIntegerIncrementer implements Runnable
{
   private final AtomicInteger[] data;

   public AlternateAtomicIntegerIncrementer(AtomicInteger[] data)
   {
      this.data = data;
   }

   public void run()
   {
      for (int i = 0; i < data.length; i++)
      {
         while (true)
         {
            int initial = data[i].intValue();
            if (data[i].compareAndSet(initial, initial + 1)) break;
         }
      }
   }
}
```

(Qui esplicitiamo il meccanismo insito nell'incrementAndGet)

Atomic* in Java5

- Come nel caso del SynchronizedIncrementer, un uso reale non rileva discrepanze nei valori dell'array (non vi è alcun output):

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerIncrementerTest
{
   public static void main(String[] args) throws InterruptedException
   {
      AtomicInteger[] data = new AtomicInteger[10000];
      for (int i = 0; i < data.length; i++)
      {
         data[i] = new AtomicInteger(0);
      }
      Thread[] threads = new Thread[1000];
      for (int i = 0; i < threads.length; i++)
      {
         threads[i] = new Thread(new AtomicIntegerIncrementer(data));
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].start();
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].join();
      }
      for (int i = 0; i < data.length; i++)
      {
         int j = data[i].intValue();
         if (j != threads.length) System.out.println("[" + i + "] = " + j);
      }
   }
}
```

Atomic* in Java5

- Finora abbiamo utilizzato array di oggetti di tipo AtomicInteger (sostituendo il tipo originario degli elementi dell'array, che era int)
- Esiste un'alternativa?
- Sì! Disponiamo anche di classi che gestiscono la coerenza di un intero array di interi → AtomicIntegerArray

Atomic* in Java5

- Incrementer che sfrutta AtomicIntegerArray:

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class AtomicIntegerArrayIncrementer implements Runnable
{
   private final AtomicIntegerArray data;

   public AtomicIntegerArrayIncrementer(AtomicIntegerArray data)
   {
      this.data = data;
   }

   public void run()
   {
      for (int i = 0; i < data.length(); i++)
      {
         data.incrementAndGet(i);
      }
   }
}
```

Atomic* in Java5

- Esempio d'uso:

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class AtomicIntegerArrayIncrementerTest
{
   public static void main(String[] args) throws InterruptedException
   {
      AtomicIntegerArray data = new AtomicIntegerArray(10000);
      Thread[] threads = new Thread[1000];
      for (int i = 0; i < threads.length; i++)
      {
         threads[i] = new Thread(new AtomicIntegerArrayIncrementer(data));
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].start();
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].join();
      }
      for (int i = 0; i < data.length(); i++)
      {
         int j = data.get(i);
         if (j != threads.length) System.out.println("[" + i + "] = " + j);
      }
   }
}
```

Atomic* in Java5

- Quando usare AtomicInteger?
  - Per il codice che intendete scrivere è necessario gestire l'accesso concorrente a poche variabili intere (al limite, anche una soltanto)
    - Ogni variabile intera "protetta" diventa un AtomicInteger
- Quando usare AtomicIntegerArray?
  - Si intende gestire un accesso concorrente a una moltitudine di variabili intere, organizzate in uno più array
    - Ogni array di interi "protetto" diventa un AtomicIntegerArray

Lock in Java5

- Come già accennato, al blocco synchronized in Java5 è stata fornita un'alternativa, gli oggetti di tipo Lock ("lucchetto")

- A differenza dei meccanismi di Object, che sono disponibili per qualsiasi oggetto creato, i Lock devono essere creati esplicitamente

Lock in Java5: pro e contro

- Pro:
  - È possibile "tentare" l'accesso, in maniera non bloccante
    - Se l'accesso alla zona critica non è disponibile, il programma più prendere una strada differente
  - Migliori performance in presenza di forte contesa
  - È possibile gestire esplicitamente ed in maniera flessibile il concetto di "condizione" del monitor
- Contro:
  - L'uscita dalla zona critica dev'essere gestita esplicitamente
    - … ma, fortunatamente, esistono meccanismi Java che consentono di codificare tutto ciò in maniera robusta

Lock in Java5

- Cosa è possibile fare con un Lock
  - Accedere al lock, attendendo che sia libero, se non lo è
    - **void lock**();
  - Come sopra, accettando però un'interruzione dell'attesa
    - **void lockInterruptibly**() **throws** InterruptedException;
  - Tentare l'accesso immediato al lock
    - **boolean tryLock**();
  - Tentare l'accesso al lock, entro un certo lasso di tempo
    - **boolean tryLock**(**long** time, TimeUnit unit) **throws** InterruptedException;
  - Sbloccare il lock (consentendo l'accesso ad altri thread)
    - **void unlock**();
  - Creare una nuova condizione
    - Condition **newCondition**();
      - Ci ritorneremo….

Lock in Java5

- L'utilizzo tipico dei lock è, come nel caso della parola chiave synchronized, l'avvolgimento di un blocco di codice tra un lock() ed un unlock()

- Si utilizza il costrutto try{} finally{}, ponendo:
  - lock() immediatamente prima della parola chiave try
  - unlock() nel blocco finally

- Si ricordi che try/finally, in Java, assicura che ciò che è stato scritto in *finally* venga eseguito, in qualsiasi condizione, subito dopo l'uscita dal blocco di codice racchiuso nel *try*

Lock in Java5

- Esempio di blocco try/finally con lock/unlock:

```
 lock.lock();
try
{
    // operations…
}
finally
{
    lock.unlock();
}
```

Lock in Java5

- Quali tipi di lock sono disponibili:
  - ReentrantLock
    - Supporta l'esecuzione multipla (annidata) dell'operazione lock()
    - Supporta un'esecuzione "fair" (rispetta l'ordine di arrivo in fila) o "unfair" (fila "all'italiana")
  - ReentrantReadWriteLock
    - Specializzato per l'algoritmo dei lettori/scrittori
      - Ci ritorneremo…

Lock in Java5

- Come si modifica l'Incrementer per usare i Lock?

```
import java.util.concurrent.locks.Lock;

public class LockedIncrementer implements Runnable
{
   private final int[] data;

   private final Lock  lock;

   public LockedIncrementer(int[] data, Lock lock)
   {
      this.data = data;
      this.lock = lock;
   }

   public void run()
   {
      lock.lock();
      try
      {
         for (int i = 0; i < data.length; i++)
         {
            data[i] = data[i] + 1;
         }
      }
      finally
      {
         lock.unlock();
      }
   }
}
```

Questa volta la sincronizzazione avviene su un oggetto esterno, non ci basta l'int[] dunque, ma dobbiamo farci passare anche un Lock.

Lock in Java5

- Esempio d'uso:

```
import java.util.concurrent.locks.ReentrantLock;

public class LockedIncrementerTest
{
   public static void main(String[] args) throws InterruptedException
   {
      int[] data = new int[10000];
      ReentrantLock lock = new ReentrantLock(true);
      Thread[] threads = new Thread[1000];
      for (int i = 0; i < threads.length; i++)
      {
         threads[i] = new Thread(new LockedIncrementer(data, lock));
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].start();
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].join();
      }
      for (int i = 0; i < data.length; i++)
      {
         int j = data[i];
         if (j != threads.length) System.out.println("[" + i + "] = " + j);
      }
   }
}
```

Domande

- Quali aspetti differenziano i Lock dalla parola chiave synchronized?
- Com'è possibile, in Java5, gestire la concorrenza di strutture dati che comprendono interi (eventualmente, organizzati in array) ?
- Come si può assicurare il rilascio di un lock?