20. Comunicazione tra thread e mutua esclusione

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

Comunicazione tra thread

- Piuttosto che evolvere in maniera del tutto indipendente tra loro, è spesso utile che due thread possano **comunicare**

- Siccome thread dello stesso processo condividono lo spazio di memoria, il modo più semplice per farli comunicare consiste nell'utilizzare degli **oggetti condivisi**

- Si tratta semplicemente di oggetti ai quali entrambi i thread posseggono un riferimento

- Ad esempio, supponiamo di voler creare due thread che **condividano un numero intero**: un thread ne modificherà il valore, mentre l'altro ne leggerà solo il contenuto
- Supponiamo di creare due classi MyThread1 e MyThread2, che estendono Thread e rappresentano le nostre due tipologie di thread
- Come fare in modo che condividano un numero intero?

- Proviamo nel seguente modo:

```
int n = 0;
Thread t1 = new MyThread1(n);
Thread t2 = new MyThread2(n);
t1.start();
t2.start();
```

- E' possibile che i due thread, così creati, comunichino tra loro tramite la variabile “n”?
- La risposta è sulla slide successiva

Comunicazione tra thread

- Naturalmente, **non** è possibile che i due thread della slide precedente comunichino tramite la variabile “n”
- Difatti, essendo “n” del tipo base “int”, essa verrà passata ai due costruttori per **valore**, e non per riferimento
- Quindi, i due costruttori riceveranno entrambi zero, e, soprattutto, due copie completamente indipendenti del valore zero
- Seppure il primo thread memorizzasse e successivamente modificasse tale valore, l'operazione non avrebbe nessun effetto sulla variabile “n”, né tantomeno sul secondo thread

- Visto che il problema è dovuto al fatto che la variabile “n” è di un tipo base, proviamo con la seguente variante:

```
Integer n = 0;
Thread t1 = new MyThread1(n);
Thread t2 = new MyThread2(n);
t1.start();
t2.start();
```

- Neanche questo può funzionare, ma per un altro motivo...

- Gli oggetti di tipo Integer (come tutti i tipi wrapper) sono **immutabili**
- Quindi, i thread non possono modificare il valore della variabile condivisa "n"
- Se ad esempio un thread esegue

```
n++
```

questo non ha nessun effetto sull'altro thread, perché quell'istruzione è equivalente (tramite il meccanismo dell'**autoboxing**) a:

```
n = Integer.valueOf(n.intValue()+1)
```

- Ovvero, viene restituito un **nuovo oggetto** Integer, che nulla ha a che vedere col vecchio
- Dunque, come fare?

- Ci servirebbe una classe che contenga un intero e sia modificabile
- Possiamo facilmente creare una classe **MyInt**, simile ad Integer, ma dotata di un metodo modificatore come "setValue(int m)"
- In effetti, una classe del genere esiste già e si chiama java.util.concurrent.atomic.**AtomicInteger**

Oppure...

...possiamo usare una collezione fornita da Java, a partire da un semplice **array**

```
int[] a = new int[1];
Thread t1 = new MyThread1(a);
Thread t2 = new MyThread2(a);
t1.start();
t2.start();
```

- Per quanto possa sembrare anomalo creare un array di un solo elemento, questa soluzione è corretta e rappresenta un comodo escamotage per evitare di creare un'intera classe
- In pratica, è più comune che due thread debbano condividere un **insieme di valori**
- In questi casi, sarà del tutto naturale utilizzare un array, o una collezione del Java Collection Framework

- Fin qui, abbiamo ignorato i ben noti problemi di sincronizzazione che possono insorgere con l'accesso concorrente a variabili condivise
- Le prossime slide introducono l'argomento

Esempio: contatore condiviso

```
class Task implements Runnable {
   public void run() {
      …
      counter++;
   }
}
```

```
main(…) {
   Task task = new Task();
   Thread t1 = new Thread(task);
   Thread t2 = new Thread(task);
   Thread t3 = new Thread(task);
   t1.start();
   t2.start();
   t3.start();
}
```

- Vogliamo contare il numero di volte in cui viene eseguito il task

- Dove collochiamo la variabile counter?
- Come affrontiamo la race condition?
- Cosa cambia se ogni thread usa un’istanza diversa di Task?

Sincronizzazione tra thread

- Se due thread tentano di modificare contemporaneamente lo stesso oggetto, l'interleaving arbitrario stabilito dallo scheduler può far si che l'operazione lasci l'oggetto in uno stato incoerente
- E' necessario garantire che solo un thread alla volta possa modificare tale oggetto
- Questa proprietà prende il nome di *mutua esclusione*
- La soluzione classica al problema prevede l'uso di *mutex*

- Un mutex è un semaforo binario che supporta le operazioni base di *lock* e *unlock*
- A differenza di un semaforo, un mutex può essere rilasciato solo dal thread che l'ha acquisito
- Si consulti un libro sui sistemi operativi per ulteriori informazioni sui mutex

- Java integra i mutex nel linguaggio stesso:
  - Ad ogni oggetto, indipendentemente dal tipo, è associato un mutex (chiamato *monitor*) e una corrispondente lista d'attesa
  - La parola chiave ***synchronized*** permette di utilizzare implicitamente tali mutex

I monitor Java sono ispirati all'omonimo costrutto proposto da Brinch-Hansen e Hoare nel 1973/74:

Hoare, C. A. R. (1974). *"Monitors: an operating system structuring concept"*. Comm. ACM. 17 (10): 549–557.

- In queste lezioni, talvolta chiameremo "x.mutex" il mutex associato all'oggetto "x"
- Questa è una notazione puramente didattica, che non trova corrispondenza nel linguaggio Java

- synchronized si può applicare ad un metodo come modificatore, oppure può introdurre un blocco di codice
- Si noti che **non** si può applicare synchronized come modificatore di un campo o altra variabile

Metodi sincronizzati

- Consideriamo il caso di un metodo a cui sia applicato il modificatore synchronized
- In tal caso, diremo che il metodo è *sincronizzato*

```
public synchronized int f(int n) { ... }
```

- Supponiamo che “x.f(3)” sia una chiamata a tale metodo

- L'effetto del modificatore synchronized è il seguente:
  - Prima di entrare nel metodo “f”, il thread corrente tenta di acquisire il mutex di “x”
    - informalmente, è come se il thread chiamasse x.mutex.lock()
  - Se il mutex è già impegnato, il thread viene messo in attesa che si liberi
  - Quando esce dal metodo “f”, il thread rilascia il mutex di “x”
    - informalmente, è come se il thread chiamasse x.mutex.unlock()

Metodi sincronizzati

- In altre parole, quando un thread invoca un metodo sincronizzato “f” di un dato oggetto, altri thread che invochino **qualunque** metodo sincronizzato dello **stesso oggetto** devono aspettare che il primo thread esca dalla chiamata a “f”
- Questo garantisce che solo un thread alla volta possa eseguire i metodi sincronizzati di ciascun oggetto
- Se un metodo **statico** di una classe “A” è sincronizzato, il thread che lo invoca acquisirà il mutex dell'oggetto **Class** corrispondente alla classe “A”

synchronized e overriding

In caso di overriding, un metodo che era sincronizzato può diventare non sincronizzato, e viceversa

- Quindi, un'interfaccia non può forzare le sue implementazioni ad avere metodi sincronizzati
- Per questo, non è possibile applicare synchronized ai metodi astratti di un'interfaccia

Blocchi sincronizzati

- La parola chiave synchronized può anche introdurre un blocco di codice
- In questo caso, parleremo di **blocco** (di codice) **sincronizzato**
- Usato in questo modo, synchronized richiede come argomento l'oggetto del quale vogliamo acquisire il mutex
- Ad esempio, il seguente frammento di codice:

```
Employee emp = ...;
synchronized (emp) { // sezione critica rispetto a emp
   ...
}
```

- corrisponde informalmente a (il codice seguente non è Java, ma è solo esemplificativo):

```
Employee emp = ...;
emp.mutex.lock();
...
emp.mutex.unlock();
```

- Si noti che i mutex acquisiti dai blocchi sincronizzati sono gli **stessi** che sono utilizzati anche dai metodi sincronizzati
- Quindi, se un thread sta eseguendo un blocco che è sincronizzato sull'oggetto “x”, gli altri thread devono aspettare per eseguire eventuali metodi sincronizzati di “x”

Esempio

Supponiamo che la classe A abbia i seguenti metodi:

```
synchronized void f()
synchronized void g()
void h()
static synchronized void i()
```

Consideriamo due oggetti distinti a e b di tipo A e un thread che invoca a.f()

Che succede se nel frattempo un altro thread invoca...

1) a.f()

2) b.f()

3) a.g()

4) a.h()

5) A.i()

Esempio

Supponiamo che la classe A abbia i seguenti metodi:

```
synchronized void f()
synchronized void g()
void h()
static synchronized void i()
```

Consideriamo due oggetti distinti a e b di tipo A e un thread che invoca a.f()

Che succede se nel frattempo un altro thread invoca...

1) a.f() deve aspettare

2) b.f() non deve aspettare

3) a.g() deve aspettare

4) a.h() non deve aspettare

5) A.i() non deve aspettare

Monitor in C#

- C# offre un meccanismo di monitor molto simile a Java

- Ad ogni oggetto è associato un monitor
- La sintassi per un blocco sincronizzato è la seguente

```
lock (obj) {
     ...
}
```

- A differenza di Java, è possibile acquisire e rilasciare *esplicitamente* un monitor:

```
Monitor.Enter(obj);
...
Monitor.Exit(obj);
```

Osservazioni

- I monitor di Java sono *rientranti* (*reentrant*)

- Ciò vuol dire che un thread può acquisire lo *stesso* mutex più volte
- Questo accade comunemente, ogni qual volta un metodo sincronizzato ne chiama un altro, anch'esso sincronizzato
- Se i monitor non fossero rientranti, un metodo sincronizzato che ne chiamasse un altro sullo stesso oggetto andrebbe immediatamente in deadlock

- Internamente, un monitor rientrante ricorda quante volte è stato acquisito dallo stesso thread
  - quindi, il monitor contiene un contatore, che viene incrementato ad ogni acquisizione (lock) e decrementato ad ogni rilascio (unlock)
  - il monitor risulta libero quando il contatore vale zero

- Per certi versi, un mutex (o monitor) rientrante è simile ad un semaforo (*counting semaphore*)
  - tuttavia, un semaforo può essere incrementato e decrementato da diversi thread, mentre un thread non può né acquisire né rilasciare un mutex che in quel momento risulti acquisito (una o più volte) da un altro thread

Esercizio

[simile a 11/2/2013, #3]

Data la seguente interfaccia:

```
public interface Predicate<T> {
   boolean test(T x);
}
```

implementare il metodo (statico) ***concurrentFilter***, che prende come argomenti un *Set X* e un Predicate *p*, di tipi compatibili, e restituisce un nuovo insieme *Y* che contiene quegli elementi di *X* per i quali la funzione *test* di *p* restituisce il valore *true*.

Inoltre, il metodo deve invocare la funzione *test* **in parallelo** su tutti gli elementi di X (dovrà quindi creare tanti thread quanti sono gli elementi di X).

[Esempio d'uso sulla slide successiva]

Esempio d'uso:

```
Set<Integer> x = new HashSet<Integer>();
x.add(1); x.add(2); x.add(5);

Predicate<Integer> isOdd = new Predicate<Integer>() {
   public boolean test(Integer n) {
      return (n%2 != 0);
   }
};

Set<Integer> y = concurrentFilter(x, isOdd);
for (Integer n: y)
   System.out.println(n);
```

Output:

```
1
5
```

Classi *thread-safe*

- Dal libro *Java Concurrency in Practice* (Brian Goetz et al.):

  *A class is thread–safe if it **behaves correctly** when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with **no additional synchronization** or other coordination on the part of the calling code.*

- Parafrasando, una classe thread-safe mantiene il proprio contratto anche se utilizzata da diversi thread contemporaneamente, senza sincronizzazione da parte del chiamante

Esempio di *thread-safety*

- Consideriamo la solita classe Employee
- Stabiliamo questo *invariante* per la classe:

  Il salario è un numero non negativo

- Consideriamo un metodo **incrementSalary**, che aggiunge al salario un valore dato
- Il metodo sarà thread-safe se potrà essere invocato contemporaneamente da diversi thread, mantenendo sempre rispettato il suo contratto e l'invariante di classe

Esempio di *thread-safety*

Quali di queste versioni sono thread-safe?

1.
```
public void incrementSalary(int delta) {
    if (delta >= 0)
        salary += delta;
}
```

2.
```
public void incrementSalary(int delta) {
    if (salary + delta >= 0)
        salary += delta;
}
```

3.
```
public synchronized void incrementSalary(int delta) {
    if (salary + delta >= 0)
        salary += delta;
}
```

Esempio di *thread-safety*

Quali di queste versioni sono thread-safe?

1. *Non thread-safe:* può violare il contratto di incrementSalary

```
public void incrementSalary(int delta) {
    if (delta >= 0)
        salary += delta;
}
```

2. *Non thread-safe:* può violare l'invariante e il contratto di incrementSalary

```
public void incrementSalary(int delta) {
    if (salary + delta >= 0)
        salary += delta;
}
```

3. *Thread-safe*

```
public synchronized void incrementSalary(int delta) {
    if (salary + delta >= 0)
        salary += delta;
}
```

Collezioni standard e thread safety

- La maggior parte delle collezioni standard non è thread safe, per motivi di efficienza
- LinkedList, ArrayList, HashSet/Map e TreeSet/Map non sono thread safe
- Se più thread condividono una di queste collezioni, e almeno uno dei thread modifica la collezione (è uno "scrittore"), tutti i thread devono accedere alla collezione condivisa in mutua esclusione
- Ad esempio, acquisendo il monitor di quella collezione
- In una lezione successiva esamineremo alcune collezioni standard thread safe

Supporto al multi-threading in Java

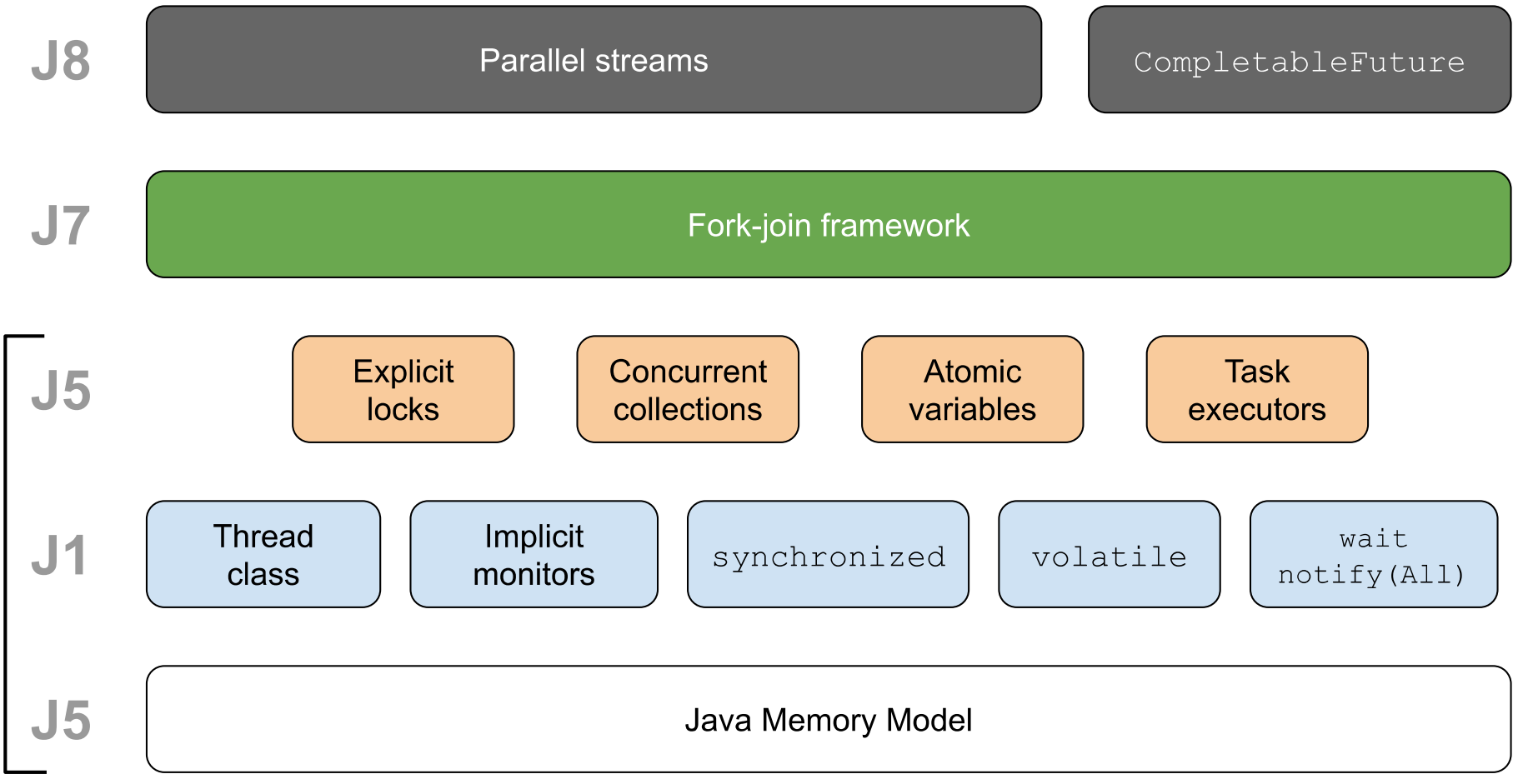

Immagine da: Seriously Good Software, di M. Faella © 2020 Manning Publications