Azzolini Riccardo

Correttezza dei programmi concorrenti – Problemi tipici e come evitarli

1 Nondeterminismo

Come mostrato negli esempi visti in precedenza, anche solo un programma concorrente semplice, con pochi thread, può avere un numero molto elevato di diversi possibili percorsi di esecuzione (non controllati dal programmatore). Questo aspetto prende il nome di **nondeterminismo**: il programmatore non può determinare l'ordine assoluto di esecuzione delle istruzioni.

A causa del nondeterminismo, testare un programma concorrente è solitamente difficile.

2 Race condition

Si dicono **race condition** (in italiano *corse critiche*, *condizioni di corsa* o, genericamente, *errori dipendenti dal tempo*), tutte quelle situazioni in cui thread diversi operano su una risorsa comune, in modo tale che il risultato dipenda dall'ordine (nondeterministico) in cui essi effettuano le loro operazioni.

2.1 Esempio

```
public class Counter {
    private long count = 0;

    public void add(long value) {
        long tmp = this.count;
        tmp = tmp + value;
        this.count = tmp;
    }

    public long getValue() {
        return count;
    }
}
```

Nonostante questo codice sia molto semplice, esso può produrre risultati sbagliati, se eseguito da più thread contemporaneamente. Ad esempio, se un singolo oggetto `Counter` viene condiviso tra due thread, che effettuano ciascuno 10000 chiamate `add(1)`,

```
public class RaceExample extends Thread {
    Counter myCounter;

    public RaceExample(Counter myCounter) {
        this.myCounter = myCounter;
    }

    public void run() {
        for (int i = 0; i < 10000; i++) {
            myCounter.add(1);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        RaceExample t1 = new RaceExample(counter);
        RaceExample t2 = new RaceExample(counter);
        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("counter = " + counter.getValue());
    }
}
```

ci si aspetterebbe che, alla fine, il valore del contatore sia 20000. In realtà, è invece possibile che alcuni degli incrementi del contatore si "perdano", se si hanno sequenze di esecuzione simili a questa:

Thread 1	Thread 2
`tmp = this.count;  //tmp=119`	
	`tmp = this.count;  //tmp=119`
	`tmp = tmp + value; //tmp=120`
	`this.count = tmp;  //count=120`
`tmp = tmp + value; //tmp=120`	
`this.count = tmp;  //count=120`	

Osservazione: Siccome `tmp` è una variabile locale del metodo `add`, ne esiste un'istanza *separata* per ciascun thread che esegue tale metodo. Invece, `count` appartiene a un

oggetto di classe `Counter`, quindi è condivisa tra tutti i thread che accedono a tale oggetto.

2.2 Quando si possono verificare

Perché si possa verificare una race condition, cioè il programma possa fornire risultati diversi in diverse esecuzioni, sono necessarie due condizioni:

- Una risorsa deve essere condivisa tra almeno due thread (nell'esempio precedente, questa è la variabile `count` dell'oggetto `Counter`).
- Deve esistere almeno un percorso di esecuzione tra i thread nel quale tale risorsa è condivisa in modo non sicuro. Nell'esempio precedente, il codice di `add` condivide in modo non sicuro la variabile `count` perché effettua una lettura e una scrittura su di essa,

  ```
  public void add(long value) {
      long tmp = this.count; // Lettura
      tmp = tmp + value;
      this.count = tmp; // Scrittura
  }
  ```

  ma, tra queste due operazioni, può essere eseguito il codice di un altro thread.

Anche riscrivendo il codice in modo da effettuare l'aggiornamento di `count` con un'unica istruzione Java,

```
public void add(long value) {
    this.count += value;
}
```

non cambierebbe nulla: i thread vengono interrotti a livello delle istruzioni di byte code, non delle istruzioni sorgente Java, e `this.count += value` corrisponde a più istruzioni di byte code, cioè, in altre parole, *non è atomica*.

3 Come assicurare l'assenza di race condition

In generale, non si può dimostrare l'assenza di race condition attraverso dei test fatti nel modo classico. Infatti, anche se il programma testato si comportasse correttamente N volte, la $(N+1)$-esima volta potrebbe verificarsi una sequenza di istruzioni diversa da tutte le N precedenti, ed essa potrebbe essere proprio quella in cui si manifesta l'errore.

Esistono invece due modi per assicurare la correttezza di un programma concorrente (dal punto di vista delle race condition):

- Dimostrare formalmente che il programma soddisfa le condizioni sufficienti perché possa essere considerato sicuro (ma effettuare questa dimostrazione a posteriori è spesso troppo costoso).
- Sviluppare il programma utilizzando strategie note che permettano di evitare le race condition. Alla base di tali strategie si ha l'idea di fare in modo che una risorsa condivisa sia in uno stato sicuro prima di consentire a un altro thread di accedervi, cioè di bloccare l'accesso a essa mentre è in uno stato non sicuro.

Nell'esempio, le istruzioni del metodo `add` costituiscono una **sezione critica**, nella quale avvengono, in tempi successivi, la lettura e la scrittura di una variabile condivisa (`count`). Per rendere sicuro il programma, è necessario un meccanismo che *blocchi* l'accesso alla sezione critita quando un thread vi entra, e lo *sblocchi* quando il thread ne esce: in questo modo, solo un thread alla volta può essere nella sezione critica.

```
public void add(long value) {
    // Qui bisogna bloccare
    long tmp = this.count;
    tmp = tmp + value;
    this.count = tmp;
    // Qui bisogna sbloccare
}
```

4 Semafori

Uno dei modi per bloccare l'accesso a una risorsa condivisa è un **semaforo**:

1. il primo thread che accede alla risorsa blocca l'accesso mediante il semaforo;
2. quando ha finito di usare la risorsa, sblocca l'accesso, consentendo a un altro thread di accedere;
3. questo, a sua volta, bloccherà eventuali altri thread, e così via.

Esistono due tipi di semafori: **contatori** e **binari**. Un semaforo contatore viene inizializzato a un valore intero positivo, poi:

- a ogni richiesta di accesso alla risorsa viene decrementato, finché non arriva a 0, poi i thread che fanno ulteriori richieste vengono bloccati;
- a ogni uscita viene incrementato, sbloccando un thread in attesa, se c'è.

Un semaforo binario, detto anche **mutex**, funziona allo stesso modo, ma viene inizializzato a 1, e può assumere solo i valori 0 e 1, quindi lascia "entrare" solo un thread alla volta, bloccando tutti gli altri finché questo non esce.

Java fornisce un semaforo generico n-ario (cioè contatore), mediante la classe `java.util.concurrent.Semaphore`, che comunque può essere usata anche per implementare un mutex, inizializzando il semaforo a 1. I principali metodi di questa classe sono:

- `acquire()`, per richiedere l'accesso alla risorsa condivisa;
- `release()`, che rilascia la risorsa.

4.1 Uso dei semafori nell'esempio

Ci sono vari modi per rendere sicuro il codice dell'esempio precedente attraverso un semaforo. Uno di questi è modificare il codice del metodo `run`, aggiungendo:

1. una chiamata `acquire()` prima di `add`;
2. una chiamata `release()` dopo `add`.

In questo modo, la classe `Counter` non ha bisogno di alterazioni.

```
import java.util.concurrent.Semaphore;

public class RaceExample extends Thread {
    Counter myCounter;
    Semaphore semaphore;

    public RaceExample(Counter myCounter, Semaphore semaphore) {
        this.myCounter = myCounter;
        this.semaphore = semaphore;
    }

    public void run() {
        for (int i = 0; i < 10000; i++) {
            try {
                semaphore.acquire();
            } catch (InterruptedException e) {
                // Terminare il thread
            }
            myCounter.add(1);
            semaphore.release();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Semaphore mutex = new Semaphore(1);
        RaceExample t1 = new RaceExample(counter, mutex);
```

```
        RaceExample t2 = new RaceExample(counter, mutex);
        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("counter = " + counter.getValue());
    }
}
```

Osservazione: Anche l'ultima istruzione del `main` accede alla variabile condivisa `count`, ma, siccome effettua solo una lettura, non abbinata a una scrittura, essa non costituisce una sezione critica, e non è quindi necessario l'utilizzo del semaforo.

In alternativa, la classe `Counter` può creare internamente un proprio semaforo, e fornire dei metodi (qui `lock` e `unlock`) che ne permettano la gestione. Il programmatore dovrà comunque ricordarsi di chiamare correttamente tali metodi ogni volta che vuole invocare `add`.

```
import java.util.concurrent.Semaphore;

public class Counter {
    private long count = 0;
    private Semaphore mutex = new Semaphore(1);

    public void add(long value) {
        this.count += value;
    }

    public long getValue() {
        return count;
    }

    public void lock() {
        try {
            mutex.acquire();
        } catch (InterruptedException e) {
            // Terminare il thread
        }
    }

    public void unlock() {
        mutex.release();
    }
}
```

```java
public class RaceExample extends Thread {
    Counter myCounter;

    public RaceExample(Counter myCounter) {
        this.myCounter = myCounter;
    }

    public void run() {
        for (int i = 0; i < 10000; i++) {
            myCounter.lock();
            myCounter.add(1);
            myCounter.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        RaceExample t1 = new RaceExample(counter);
        RaceExample t2 = new RaceExample(counter);
        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("counter = " + counter.getValue());
    }
}
```

Infine, una soluzione ancora più pratica e sicura è rendere **thread-safe** la classe `Counter`, cioè incorporare direttamente nella classe dei meccanismi per la gestione delle sezioni critiche. A tale scopo, ogni istanza di `Counter` viene dotate di un semaforo privato, come nell'esempio precedente, ma con la differenza che questo viene utilizzato direttamente nel metodo `add`. In questo modo, il codice del chiamante può essere scritto senza preoccuparsi di possibili race condition.

```java
import java.util.concurrent.Semaphore;

public class Counter {
    private long count = 0;
    private Semaphore mutex = new Semaphore(1);

    public void add(long value) {
        try {
```

```
            mutex.acquire();
        } catch (InterruptedException e) {
            // Terminare il thread
        }
        this.count += value;
        mutex.release();
    }

    public long getValue() {
        return count;
    }
}

public class RaceExample extends Thread {
    Counter myCounter;

    public RaceExample(Counter myCounter) {
        this.myCounter = myCounter;
    }

    public void run() {
        for (int i = 0; i < 10000; i++) {
            myCounter.add(1);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        RaceExample t1 = new RaceExample(counter);
        RaceExample t2 = new RaceExample(counter);
        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("counter = " + counter.getValue());
    }
}
```

4.2 Pericolo dell'uso dei semafori

Soprattutto nei programmi complessi, può capitare di commettere errori nell'uso dei semafori: ad esempio, dimenticare di rilasciare un semaforo, o rilasciarlo più di una

volta. Tali errori portano solitamente a problemi quali race condition o deadlock.

5 Modificatore `synchronized`

Per prevenire gli errori legati all'uso dei semafori, in Java è stato introdotto un metodo alternativo per il controllo dell'accesso alle sezioni critiche: a *ogni istanza* della classe `Object` (e quindi di tutte le classi, in quanto specializzazioni di `Object`) è associato un semaforo binario, chiamato **intrinsic lock**. Per utilizzarlo, si contrassegna con la parola chiave `synchronized` un metodo, oppure una qualunque sezione di codice.

1. Quando un thread entra in un metodo/area `synchronized`, cerca di acquisire il lock associato all'oggetto: se tale lock è già stato acquisito da un altro thread, il thread corrente si mette in attesa, altrimenti esso acquisisce il lock, e può allora iniziare a eseguire il codice di tale metodo/area.
2. Quando si esce dal metodo/area `synchronized`, si rilascia automaticamente il lock associato all'oggetto. Di conseguenza, un eventuale altro thread in attesa passa allo stato ready, e può acquisire il lock ed entrare.

Con questo meccanismo, non ci possono mai essere due o più thread che eseguono contemporaneamente metodi/aree `synchronized` dello stesso oggetto, ovvero si ha la **mutua esclusione** su tali metodi/aree.

5.1 Sintassi

La sintassi per contrassegnare come `synchronized` una sezione di codice qualunque è

```
synchronized (obj) { // Acquisizione del lock
    // Sezione critica su obj...
} // Rilascio del lock
```

dove:

- `obj` è l'oggetto di cui si vuole ottenere il lock (spesso si usa `this`, cioè l'oggetto al quale appartiene il metodo contenente questo frammento di codice);
- il lock viene acquisito prima di eseguire il codice racchiuso tra le parentesi graffe, e rilasciato quando tale codice termina.

Anche applicare il modificatore `synchronized` a un intero metodo, con la sintassi

```
public synchronized void example() {
    // Codice del metodo
}
```

equivale semplicemente a:

```
public void example() {
    synchronized (this) {
        // Codice del metodo
    }
}
```

Nota: Se uno stesso oggetto ha più metodi/blocchi sincronizzati, potrà esserne eseguito solo uno di essi alla volta, dato che ogni oggetto ha un unico lock, e non uno separato per ciascun blocco sincronizzato.

5.2 Applicazione all'esempio

La race condition dell'esempio precedente può semplicemente essere risolta dichiarando come `synchronized` il metodo `add`, senza bisogno di usare esplicitamente un oggetto `Semaphore`:

```
public class Counter {
    private long count = 0;

    public synchronized void add(long value) {
        this.count += value;
    }

    public long getValue() {
        return count;
    }
}
```

5.3 Altro esempio

```
public class AssegnatorePosto {
    private int postiDisponibili = 20;

    public boolean assegnaPosti(int postiRichiesti) {
        if (postiRichiesti <= postiDisponibili) { // Sezione
            postiDisponibili -= postiRichiesti;   // critica
            return true;
        } else {
            return false;
        }
    }

    public int getPostiDisponibili() {
```

```
        return postiDisponibili;
    }
}
```

Le prime due righe del codice di `assegnaPosti` costituiscono una sezione critica, dato che eseguono una lettura e, successivamente, una scrittura della variabile `postiDisponibili`: se più thread eseguissero `assegnaPosti` in parallelo, il numero di posti assegnati alla fine potrebbe essere maggiore di quello realmente disponibile, ovvero si potrebbe avere una race condition.

Ad esempio, nel seguente codice vengono creati quattro thread concorrenti, che condividono una stessa istanza di `AssegnatorePosto`, alla quale ciascuno richiede un determinato numero di posti:

```
public class Richiedente extends Thread {
    private int posti;
    private AssegnatorePosto assegnatore;

    public Richiedente(
        String nome,
        int posti,
        AssegnatorePosto assegnatore
    ) {
        super(nome);
        this.posti = posti;
        this.assegnatore = assegnatore;
    }

    public void run() {
        System.out.println(getName() + " richiede " + posti);
        if (assegnatore.assegnaPosti(posti)) {
            System.out.println(getName() + " ottiene " + posti);
        } else {
            System.out.println(getName() + " NON ottiene " + posti);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        AssegnatorePosto assegnatore = new AssegnatorePosto();
        Richiedente[] clienti = new Richiedente[4];
        clienti[0] = new Richiedente("cliente 1", 3, assegnatore);
        clienti[1] = new Richiedente("cliente 2", 5, assegnatore);
        clienti[2] = new Richiedente("cliente 3", 3, assegnatore);
        clienti[3] = new Richiedente("cliente 4", 10, assegnatore);
        for (int i = 0; i < clienti.length; i++) {
```

```
            clienti[i].start();
        }

        for (int i = 0; i < clienti.length; i++) {
            clienti[i].join();
        }
        System.out.println(
            "Al termine restano disponibili "
            + assegnatore.getPostiDisponibili()
            + " posti"
        );
    }
}
```

Un possibile output corretto di questo programma è:

```
cliente 2 richiede 5
cliente 4 richiede 10
cliente 2 ottiene 5
cliente 4 ottiene 10
cliente 3 richiede 3
cliente 1 richiede 3
cliente 3 ottiene 3
cliente 1 NON ottiene 3
Al termine restano disponibili 2 posti
```

Invece, un esempio di output errato, dovuto a una race condition, è:

```
cliente 4 richiede 10
cliente 2 richiede 5
cliente 3 richiede 3
cliente 1 richiede 3
cliente 4 ottiene 10
cliente 2 ottiene 5
cliente 1 ottiene 3
cliente 3 ottiene 3
Al termine restano disponibili -1 posti
```

Per correggere questo esempio, è necessario assicurarsi che la sezione critica, contenuta nel metodo `assegnaPosti`, possa essere eseguita da un solo thread alla volta, contrassegnando come `synchronized`:

- l'intero metodo:

```
public synchronized boolean assegnaPosti(int postiRichiesti) {
    if (postiRichiesti <= postiDisponibili) { // Sezione
        postiDisponibili -= postiRichiesti;   // critica
        return true;
    } else {
        return false;
    }
}
```

- in alternativa, solo la porzione di codice che costituisce la sezione critica:

```
public boolean assegnaPosti(int postiRichiesti) {
    synchronized (this) {
        if (postiRichiesti <= postiDisponibili) { // Sezione
            postiDisponibili -= postiRichiesti;   // critica
            return true;
        }
    }
    return false;
}
```

5.4 Accesso ai dati

I metodi `synchronized` hanno **accesso esclusivo** ai dati incapsulati in un oggetto solo se a tali dati si accede esclusivamente, appunto, con metodi `synchronized`. Invece, i metodi non `synchronized` non sono esclusivi, quindi permettono l'**accesso concorrente** ai dati.

Tali accorgimenti non interessano le variabili locali: poiché ogni thread ha il proprio stack, anche quando più thread eseguono contemporaneamente lo stesso metodo, ciascuno di essi avrà una copia separata delle variabili locali, senza pericolo di "interferenza".

5.5 Variabili statiche

Normalmente, i metodi/blocchi `synchronized` *non assicurano* l'accesso mutualmente esclusivo ai dati statici, che sono condivisi da tutti gli oggetti della stessa classe: siccome ognuno di questi oggetti ha un lock separato, thread diversi potrebbero accedere contemporaneamente a tali dati, mediante oggetti diversi.

A ogni classe Java, però, è associato un oggetto di tipo `Class`: per accedere in modo sincronizzato ai dati statici, si sfrutta il lock di questo oggetto. Ci sono due modi equivalenti di farlo:

- dichiarando un *metodo statico* come `synchronized`;

- contrassegnando un blocco come `synchronized` sull'oggetto di tipo `Class`, il quale è accessibile mediante la sintassi `NomeClasse.class`.

Importante: Il lock a livello di classe non si ottiene quando ci si sincronizza su un oggetto di tale classe, e viceversa.

5.5.1 Esempio

```
public class StaticSharedVariable {
    private static int shared;

    public int read() {
        synchronized (StaticSharedVariable.class) {
            return shared;
        }
    }

    public static synchronized void write(int i) {
        shared = i;
    }
}
```

5.6 Ereditarietà

La specifica `synchronized` non fa propriamente parte della segnatura di un metodo. Di conseguenza, una classe derivata può ridefinire un metodo `synchronized` come non `synchronized`, e viceversa. Ciò è molto comodo, in quanto consente di:

1. definire classi adatte all'uso sequenziale, senza preoccuparsi dei problemi legati alla concorrenza;
2. successivamente, derivare da esse delle sottoclassi che vengono rese adatte all'uso concorrente mediante `synchronized`.

5.6.1 Esempio

```
public class AssegnatoreSequenziale {
    private int postiDisponibili = 20;

    public boolean assegnaPosti(int postiRichiesti) {
        if (postiRichiesti <= postiDisponibili) {
            postiDisponibili -= postiRichiesti;
            return true;
```

```
        } else {
            return false;
        }
    }

    public int getPostiDisponibili() {
        return postiDisponibili;
    }
}

public class AssegnatoreConcorrente extends AssegnatoreSequenziale {
    public synchronized boolean assegnaPosti(int postiRichiesti) {
        return super.assegnaPosti(postiRichiesti);
    }
}
```