Azzolini Riccardo 2020-03

Paradigmi di comunicazione

1 Barrier

Una **barrier** (barriera) blocca un insieme di thread fino a quando non raggiungono tutti tale barriera:

1. i thread che desiderano bloccarsi a una barriera devono chiamare il metodo `waitB`;
2. tutti i thread che arrivano, tranne l'ultimo, vengono trattenuti;
3. quando l'ultimo thread arriva, tutti i thread vengono rilasciati (possono procedere).

Nota: Questo è un costrutto che serve puramente per la sincronizzazione, senza alcuna comunicazione di dati (se necessaria, questa dovrà essere effettuata tramite altri mezzi). In particolare, esso è utile quando bisogna essere certi che il sistema sia arrivato in una situazione nota.

Una possibile implementazione di una barriera è la seguente:

```
public class Barrier {
    // Numero di thread che devono arrivare perché la barriera si apra
    private final int need;
    // Numero di thread attualmente presenti alla barriera
    private int arrived = 0;
    // Stato della barriera (true significa aperta, false indica chiusa)
    private boolean releasing = false;

    public Barrier(int need) {
        this.need = need;
    }

    public synchronized int getArrived() {
        return arrived;
    }

    public int capacity() {
        return need;
    }

    public synchronized void waitB() throws InterruptedException {
```

```
        while (releasing) {
            // La barriera è ancora aperta (per far passare un gruppo
            // precedente di thread).
            // Bisogna aspettare che si richiuda, altrimenti potrebbero
            // passare più thread del numero previsto.
            wait();
        }

        arrived++;
        try {
            while (arrived < need && !releasing) {
                wait();
            }
            if (arrived == need) {
                releasing = true;
                System.out.println(
                    "All " + need + " parties arrived at barrier"
                );
                notifyAll();
            }
        } finally {
            arrived--;
            if (arrived == 0) {
                // L'ultimo thread che lascia la barriera la richiude,
                releasing = false;
                // e sveglia i nuovi thread che stavano aspettando la
                // chiusura (i quali andranno alla seconda wait).
                notifyAll();
            }
        }
    }
}
```

1.1 Esempio con 3 thread

In questo esempio ci sono 3 thread, ciascuno dei quali attraversa 4 volte la barriera, aspettando un tempo casuale da quando passa a quando si rimette in attesa di passare.

```
import java.util.concurrent.ThreadLocalRandom;

public class BarrierExample {
    private static class Task implements Runnable {
        private final Barrier barrier;
```

```
        public Task(Barrier barrier) {
            this.barrier = barrier;
        }

        public void run() {
            try {
                for (int i = 0; i < 4; i++) {
                    System.out.println(
                        Thread.currentThread().getName()
                        + " arrived at the barrier"
                    );
                    barrier.waitB();
                    System.out.println(
                        Thread.currentThread().getName()
                        + " has crossed the barrier"
                    );
                    Thread.sleep(
                        ThreadLocalRandom.current().nextInt(30)
                    );
                }
            } catch (InterruptedException e) {}
        }
    }

    public static void main(String[] args) {
        Barrier b = new Barrier(3);
        for (int i = 0; i < 3; i++) {
            new Thread(new Task(b), "Thread " + (i + 1)).start();
        }
    }
}
```

Un possibile output è il seguente:

```
Thread 2 arrived at the barrier
Thread 1 arrived at the barrier
Thread 3 arrived at the barrier
All 3 parties arrived at barrier
Thread 3 has crossed the barrier
Thread 2 has crossed the barrier
Thread 1 has crossed the barrier
Thread 1 arrived at the barrier
Thread 3 arrived at the barrier
```

```
Thread 2 arrived at the barrier
All 3 parties arrived at barrier
Thread 2 has crossed the barrier
Thread 1 has crossed the barrier
Thread 3 has crossed the barrier
Thread 2 arrived at the barrier
Thread 1 arrived at the barrier
Thread 3 arrived at the barrier
All 3 parties arrived at barrier
Thread 3 has crossed the barrier
Thread 2 has crossed the barrier
Thread 1 has crossed the barrier
Thread 3 arrived at the barrier
Thread 2 arrived at the barrier
Thread 1 arrived at the barrier
All 3 parties arrived at barrier
Thread 1 has crossed the barrier
Thread 3 has crossed the barrier
Thread 2 has crossed the barrier
```

1.2 Esempio: corsa di cavalli

Durante una corsa di cavalli, ogni cavallo deve entrare nella barriera (che ha una postazione per ogni cavallo) prima di partire. Inoltre, a ogni giro:

- i cavalli devono fermarsi alla barriera e aspettare tutti gli altri;
- un cronista annuncia il numero di giri fatti.

Ci sono N cavalli/postazioni, e bisogna fare M giri.

Il main crea semplicemente la barriera e i thread cavalli e cronista:

```
public class CorsaDiCavalli {
    private static final int NUM_CAVALLI = 3;
    private static final int NUM_GIRI = 2;

    public static void main(String[] args) throws InterruptedException {
        Barrier barriera = new Barrier(NUM_CAVALLI);
        for (int i = 0; i < NUM_CAVALLI; i++) {
            new Cavallo(barriera, NUM_GIRI).start();
        }
        new Cronista(barriera, NUM_GIRI).start();
    }
}
```

Il thread `Cronista` usa un apposito metodo `waitCycle` della barriera per attendere ogni volta l'inizio del giro successivo:

```
public class Cronista extends Thread {
    private final Barrier barriera;
    private final int numGiri;

    public Cronista(Barrier barriera, int numGiri) {
        this.barriera = barriera;
        this.numGiri = numGiri;
    }

    public void run() {
        for (int i = 0; i < numGiri; i++) {
            try {
                int g = barriera.waitCycle(i + 1);
                System.out.println("Siamo al giro numero " + g);
            } catch (InterruptedException e) { return; }
        }
        System.out.println("Siamo all'ultimo giro!");
    }
}
```

Per ogni giro, il thread `Cavallo` esegue una `sleep` (che simula il tempo necessario ad arrivare alla barriera), e poi si blocca per aspettare gli altri:

```
import java.util.concurrent.ThreadLocalRandom;

public class Cavallo extends Thread {
    private final Barrier barriera;
    private final int numGiri;

    public Cavallo(Barrier barriera, int numGiri) {
        this.barriera = barriera;
        this.numGiri = numGiri;
        setName("Cavallo " + getName());
    }

    public void run() {
        for (int i = 0; i < numGiri; i++) {
            try {
                System.out.println(
                    getName() + " sta per entrare nella barriera"
                );
```

```
                Thread.sleep(ThreadLocalRandom.current().nextInt(100));
                barriera.waitB();
                System.out.println(getName() + " è partito!");
            } catch (InterruptedException e) {}
        }
    }
}
```

Rispetto alla classe `Barrier` mostrata prima, quella usata in questo esempio ha in più il metodo `waitCycle`, che mette il chiamante in attesa fino al ciclo di funzionamento della barriera specificato come argomento:

```
public class Barrier {
    private final int need;
    private int arrived = 0;
    private int numCycles = 0;
    private boolean releasing = false;

    public Barrier(int need) {
        this.need = need;
    }

    public synchronized int getArrived() {
        return arrived;
    }

    public int capacity() {
        return need;
    }

    public synchronized int waitCycle(int cycleNum)
            throws InterruptedException {
        while (cycleNum != numCycles) {
            wait();
        }
        return numCycles;
    }

    public synchronized void waitB() throws InterruptedException {
        while (releasing) {
            wait();
        }

        arrived++;
```

```
        try {
            while (arrived < need && !releasing) {
                wait();
            }
            if (arrived == need) {
                releasing = true;
                System.out.println(
                    "All " + need + " parties arrived at barrier"
                );
                numCycles++;
                notifyAll();
            }
        } finally {
            arrived--;
            if (arrived == 0) {
                releasing = false;
                notifyAll();
            }
        }
    }
}
```

Un esempio di output è:

```
Cavallo Thread-2 sta per entrare nella barriera
Cavallo Thread-1 sta per entrare nella barriera
Cavallo Thread-0 sta per entrare nella barriera
All 3 parties arrived at barrier
Cavallo Thread-0 è partito!
Cavallo Thread-0 sta per entrare nella barriera
Cavallo Thread-2 è partito!
Cavallo Thread-2 sta per entrare nella barriera
Siamo al giro numero 1
Cavallo Thread-1 è partito!
Cavallo Thread-1 sta per entrare nella barriera
All 3 parties arrived at barrier
Cavallo Thread-1 è partito!
Siamo al giro numero 2
Cavallo Thread-2 è partito!
Siamo all'ultimo giro!
Cavallo Thread-0 è partito!
```

2 Classe `CyclicBarrier`

A partire dalla versione 1.5, la libreria standard di Java mette a disposizione un'implementazione di una barriera: la classe `java.util.concurrent.CyclicBarrier`.

Il funzionamento di questa barriera è analogo a quello descritto finora, ma con un elemento in più: opzionalmente, è possibile associare alla barriera un comando `Runnable`, che viene eseguito quando sono arrivati tutti i thread, subito prima che la barriera si apra. Questo permette di eseguire delle operazioni nell'istante in cui tutti i thread sono fermi.

2.1 Costruttori

I costruttori di `CyclicBarrier` sono due: quello "classico", nel quale si specifica solo il numero di thread che devono essere attesi prima che la barriera si apra,

```
public CyclicBarrier(int parties);
```

e uno con il quale si specifica anche il comando `Runnable`:

```
public CyclicBarrier(int parties, Runnable barrierAction);
```

2.2 Principali metodi

```
public int await() throws InterruptedException, BrokenBarrierException;
```

Il metodo di attesa (analogo al `waitB` delle classi `Barrier` di prima).

```
public int await(long timeout, TimeUnit unit)
    throws InterruptedException,
           BrokenBarrierException,
           TimeoutException;
```

Una versione di `await` che permette di specificare un tempo di attesa massimo, allo scadere del quale il thread viene sbloccato anche se la barriera non si è aperta (e viene sollevata un'eccezione `TimeoutException`).

```
public int getNumberWaiting();
```

Restituisce il numero di thread attualmente in attesa alla barriera (come il metodo `getArrived` di `Barrier`).

```
public int getParties();
```

Restituisce il numero totale di thread necessari per aprire la barriera (come il metodo `capacity` di `Barrier`).

```
public boolean isBroken();
```

Indica se la barriera è in uno stato inconsistente (ciò può avvenire solo in alcuni casi particolari, dovuti a delle eccezioni).

```
public void reset();
```

Resetta la barriera allo stato iniziale.

2.3 Esempio: corsa di cavalli

L'esempio della corsa di cavalli può essere riscritto usando la classe `CyclicBarrier`. La principale differenza è che il cronista, invece di essere un thread, diventa il comando `Runnable` associato alla barriera, garantendo così una sincronizzazione ottimale: il cronista dirà che inizia l'$(n+1)$-esimo giro esattamente quando tutti i cavalli hanno completato l'n-esimo, ma non sono ancora ripartiti per l'$(n+1)$-esimo.

```
import java.util.concurrent.CyclicBarrier;

public class CorsaDiCavalli {
    private static final int NUM_CAVALLI = 3;
    private static final int NUM_GIRI = 2;

    public static void main(String[] args) throws InterruptedException {
        CyclicBarrier barriera =
            new CyclicBarrier(NUM_CAVALLI, new Cronista());

        for (int i = 0; i < NUM_CAVALLI; i++) {
            new Cavallo(barriera, NUM_GIRI).start();
        }
    }
}
```

Il `Cavallo` è sostanzialmente uguale a prima: viene solo sostituito l'uso di `Barrier` con `CyclicBarrier`.

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ThreadLocalRandom;

public class Cavallo extends Thread {
    private final CyclicBarrier barriera;
    private final int numGiri;

    public Cavallo(CyclicBarrier barriera, int numGiri) {
```

```
        this.barriera = barriera;
        this.numGiri = numGiri;
        setName("Cavallo " + getName());
    }

    public void run() {
        for (int i = 0; i < numGiri; i++) {
            try {
                System.out.println(
                    getName() + " sta per entrare nella barriera"
                );
                Thread.sleep(ThreadLocalRandom.current().nextInt(3000));
                barriera.await();
                System.out.println(getName() + " è partito!");
            } catch (InterruptedException | BrokenBarrierException e) {}
        }
    }
}
```

Il codice del `Cronista` si semplifica notevolmente, dato che, come già detto, esso non è più un thread a sé stante, bensì solo un comando `Runnable` (in pratica, un singolo metodo) associato alla barriera. In particolare, il `Cronista` non ha più bisogno di preoccuparsi della sincronizzazione.

```
public class Cronista implements Runnable {
    private int giro = 0;

    public void run() {
        System.out.println("I cavalli partono per il giro " + giro);
        giro++;
    }
}
```

Un possibile output di questa nuova versione del programma è:

```
Cavallo Thread-0 sta per entrare nella barriera
Cavallo Thread-2 sta per entrare nella barriera
Cavallo Thread-1 sta per entrare nella barriera
I cavalli partono per il giro 0
Cavallo Thread-0 è partito!
Cavallo Thread-1 è partito!
Cavallo Thread-2 è partito!
Cavallo Thread-2 sta per entrare nella barriera
Cavallo Thread-0 sta per entrare nella barriera
```

```
Cavallo Thread-1 sta per entrare nella barriera
I cavalli partono per il giro 1
Cavallo Thread-0 è partito!
Cavallo Thread-1 è partito!
Cavallo Thread-2 è partito!
```

2.4 Esempio: somma delle righe di una matrice

Sia A una matrice di dimensione $n \times m$. Si vuole ottenere il vettore C, i cui elementi sono definiti come

$$c_i = \sum_{k=0}^{m-1} a_{ik}$$

dove $0 \leq i < n$.

Per farlo, si implementa un programma concorrente che calcoli gli elementi c_i in parallelo: un thread si occupa di c_0, un altro di c_1, ecc. Alla fine dei calcoli, si visualizza C. Quest'ultima parte verrà eseguita mediante una `CyclicBarrier`.

Ciascun thread `RowSummer` riceve

- l'indice della riga da sommare (`row`),
- un riferimento alla matrice (`matrix`),
- un riferimento al vettore in cui inserire la somma (`results`),
- un riferimento alla barriera (`barrier`),

calcola la somma della sua riga, e poi si mette in attesa sulla barriera per segnalare che ha finito.

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class RowSummer extends Thread {
    private final int row;
    private final int[][] matrix;
    private final int[] results;
    private final CyclicBarrier barrier;

    public RowSummer(
        int row,
        int[][] matrix,
        int[] results,
        CyclicBarrier barrier
    ) {
```

```
        this.row = row;
        this.matrix = matrix;
        this.results = results;
        this.barrier = barrier;
    }

    public void run() {
        int sum = 0;
        for (int element : matrix[row]) {
            sum += element;
        }
        results[row] = sum;
        System.out.println("Result for row " + row + " is: " + sum);
        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {}
    }
}
```

Quando tutti i `RowSummer` hanno finito, viene eseguito il metodo `run` di `ResultUser`, che visualizza gli elementi vettore C e la loro somma:

```
public class ResultUser implements Runnable {
    private final int[] results;

    public ResultUser(int[] results) {
        this.results = results;
    }

    public void run() {
        System.out.print("[");
        int total = 0;
        for (int element : results) {
            total += element;
            System.out.print(" " + element);
        }
        System.out.println(" ]");
        System.out.println("total = " + total);
    }
}
```

Nel main viene creata una `CyclicBarrier`, che aspetta tanti thread quante sono le righe della matrice, e alla quale si associa, come comando, un'istanza di `ResultUser`. Infine, viene avviato un `RowSummer` per ogni riga.

```java
import java.util.concurrent.CyclicBarrier;

public class ParMatSum {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3, 4, 5},
            {2, 2, 2, 2, 2},
            {3, 3, 3, 3, 3},
            {4, 4, 4, 4, 3},
            {5, 5, 5, 5, 5}
        };
        int rows = matrix.length;
        int[] results = new int[rows];

        ResultUser finisher = new ResultUser(results);
        CyclicBarrier barrier = new CyclicBarrier(rows, finisher);

        for (int i = 0; i < rows; i++) {
            new RowSummer(i, matrix, results, barrier).start();
        }
    }
}
```

L'output di questo programma è il seguente:

```
Result for row 3 is: 19
Result for row 4 is: 25
Result for row 1 is: 10
Result for row 0 is: 15
Result for row 2 is: 15
[ 15 10 15 19 25 ]
total = 84
```

2.5 Esempio: MapReduce

MapReduce è un framework definito da Google per supportare l'esecuzione in parallelo di calcoli su di grandi quantità di dati (tipicamente sfruttando cluster di computer).

Il principio di base è analogo a quello dell'esempio precedente:

1. si "spezza" un calcolo complesso in tanti calcoli più semplici;
2. ciascuno di questi calcoli viene eseguito in parallelo (*map*);

3. alla fine, i risultati parziali vengono raccolti per costruire il risultato complessivo (*reduce*).

Una semplice implementazione può allora essere realizzata mediante la programmazione concorrente, usando un thread per ogni calcolo da eseguire nella fase di map, e una `CyclicBarrier` per la fase di reduce.

Come esempio, si considera il problema di contare il numero di parole presenti in un insieme, potenzialmente anche molto grande, di file di testo. In questo caso, ciascun thread conterà le parole di un piccolo insieme di file (al limite, anche solo di un singolo file), e alla fine tutti i conteggi verranno sommati per ottenere il totale.

La classe `WordCounter` contiene il codice dei thread che contano le parole in un piccolo insieme di file. Per semplicità, si suppone che questi file siano chiamati `file_0.txt`, `file_1.txt`, ecc., e si assegna a ogni istanza di `WordCounter` un gruppo contiguo di file, definito indicando un range di numeri dei file.

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

public class WordCounter implements Runnable {
    private final CyclicBarrier barrier;
    private final int startFile, endFile;
    private int wordCount = 0;

    public WordCounter(
        CyclicBarrier barrier,
        int startFile,
        int endFile
    ) {
        this.barrier = barrier;
        this.startFile = startFile;
        this.endFile = endFile;
    }

    public int getWordCount() {
        return wordCount;
    }

    public void run() {
        for (int i = startFile; i < endFile; i++) {
            String fileName = "file_" + i + ".txt";
            System.out.println("reading " + fileName);
            try (
                FileReader fr = new FileReader(fileName);
```

```
                BufferedReader br = new BufferedReader(fr)
            ) {
                countWords(br);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        System.out.println("GOING TO WAIT");
        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {}
    }

    private void countWords(BufferedReader reader) throws IOException {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] words = line.split("\\s+");
            wordCount += words.length;
        }
    }
}
```

L'azione eseguita dalla `CyclicBarrier` quando tutti i `WordCounter` hanno finito legge i valori di tutti i contatori e li somma; inoltre, essa misura anche il tempo impiegato complessivamente per l'elaborazione (compreso il calcolo del totale).

```
import java.util.List;

public class BarrierReachedAction implements Runnable {
    private final List<WordCounter> wordCounters;
    private final long startTime;

    public BarrierReachedAction(
        List<WordCounter> wordCounters,
        long startTime
    ) {
        this.wordCounters = wordCounters;
        this.startTime = startTime;
    }

    public void run() {
        System.out.println("Barrier reached!");
        int totalWords = 0;
```

```
        for (WordCounter counter : wordCounters) {
            totalWords += counter.getWordCount();
        }
        long totalTimeElapsed = System.currentTimeMillis() - startTime;
        System.out.println("Elapsed time = " + totalTimeElapsed + " ms");
        System.out.println("Word count = " + totalWords);
    }
}
```

Osservazione: Mentre, nell'esempio precedente, i thread avevano dei riferimenti a un array condiviso in cui depositare i propri risultati, qui li tengono in dei loro attributi, che vengono poi letti dall'esterno.

Infine, nel main, i file da leggere vengono suddivisi tra (ad esempio) 8 thread:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CyclicBarrier;

public class WordCountExample {
    private static final int NUM_FILES = 279;
    private static final int NUM_THREADS = 8;

    public static void main(String[] args) {
        List<WordCounter> counters = new ArrayList<>();
        long startTime = System.currentTimeMillis();
        BarrierReachedAction reduce =
            new BarrierReachedAction(counters, startTime);
        CyclicBarrier barrier = new CyclicBarrier(NUM_THREADS, reduce);

        int increment =
            (int) Math.ceil((double) NUM_FILES / NUM_THREADS);
        for (int i = 0; i < NUM_THREADS; i++) {
            int startFile = i * increment;
            int endFile = Math.min(NUM_FILES, (i + 1) * increment);
            WordCounter counter =
                new WordCounter(barrier, startFile, endFile);
            counters.add(counter);
            new Thread(counter).start();
        }
    }
}
```

Un possibile output di questo programma è:

```
...
reading file_277.txt
reading file_278.txt
reading file_242.txt
GOING TO WAIT
reading file_243.txt
reading file_244.txt
GOING TO WAIT
Barrier reached!
Elapsed time = 267 ms
Word count = 436075
```

3 Conclusioni

Nella programmazione concorrente, esistono molti modi per far comunicare i thread, e diversi paradigmi di comunicazione.

Quelli appena presentati sono alcuni dei più comunemente usati, ma ce ne sono comunque diversi altri, e chiunque può inventarsi delle varianti che meglio si adattino a specifiche situazioni.