Azzolini Riccardo

Paradigmi di comunicazione

1 Paradigmi di comunicazione tra thread

Esistono molti modi in cui i thread possono comunicare tra loro. Alcuni tra i più comuni sono:

- signal,
- buffer,
- blackboard,
- broadcast,
- barrier.

Note:

- Tutti questi paradigmi possono essere implementati utilizzando le primitive di comunicazione già viste.
- La comunicazione tra thread riguarda due tipi di elementi: i dati, e le informazioni di controllo (usate per informare mutuamente i thread riguardo al verificarsi di determinati eventi, ecc.).

2 Signal

Un **signal** consente a un thread di attendere un segnale inviato da un altro thread. Un segnale è solo un'indicazione di un evento, un'informazione di controllo: non contiene particolari dati.

Tradizionalmente, ci sono due tipi di segnali:

persistent signal: rimane impostato finché un singolo thread non lo riceve;

transient signal: rilascia uno o più thread in attesa, ma si perde se invece non ci sono thread in attesa.

Nel meccanismo dei signal, si identificano due ruoli ben distinti:

- chi aspetta un segnale;
- chi invia il segnale atteso.

Spesso si tratta di thread diversi. Perciò, a livello di implemntazione, è utile separare l'interfaccia del thread che invia dall'interfaccia di quello che aspetta. In questo modo, ogni thread può chiamare l'operazione più appropriata per il suo ruolo.

```
public interface SignalSender {
    void send();
}

public interface SignalWaiter {
    void waits() throws InterruptedException;
}
```

Osservazioni:

- Il metodo `waits()` è distinto dalla primitiva `wait()` di Java (ma questa verrà tipicamente usata nella sua implementazione).
- Se si volesse evitare un'attesa infinita, si potrebbe aggiungere un metodo `waits(t)` con un timeout.

È anche utile definire una classe astratta `Signal`, dalla quale si potranno poi derivare le classi concrete che realizzeranno i segnali persistenti e transienti:

```
public abstract class Signal implements SignalSender, SignalWaiter {
    protected boolean arrived = false;

    public abstract void waits() throws InterruptedException;

    public synchronized void send() {
        arrived = true;
        notify();
    }
}
```

2.1 Persistent signal

Siccome la classe `PersistentSignal` realizza un segnale persistente, la sua implementazione del metodo `waits()` è tale che:

- se il segnale è già impostato, viene azzerato immediatamente, senza bloccare il thread chiamante;
- se il segnale non è ancora arrivato, il thread chiamante si blocca;
- quando ci sono uno o più thread bloccati, e il segnale arriva, il primo thread che si sveglia azzera ("consuma") il segnale, quindi gli eventuali altri thread in attesa dovranno aspettare l'arrivo di ulteriori segnali per sbloccarsi.

Inoltre, questa classe mette anche a disposizione un metodo *non bloccante* (ma comunque `synchronized`) `watch()` (definito dall'interfaccia `SignalWaiterOrWatcher`), che indica al chiamante se il segnale è arrivato, e al tempo stesso lo azzera, in modo che solo un thread possa trovarlo impostato. Esso può essere usato per fare polling, invece di sospendersi in attesa dell'evento.

```
public interface SignalWaiterOrWatcher extends SignalWaiter {
    boolean watch();
}

public class PersistentSignal
        extends Signal
        implements SignalWaiterOrWatcher {
    public synchronized void waits() throws InterruptedException {
        while (!arrived) {
            wait();
        }
        arrived = false;
    }

    public synchronized boolean watch() {
        if (!arrived) {
            return false;
        }
        arrived = false;
        return true;
    }
}
```

2.1.1 Esempio: accesso al disco

```
public class DiskController {
    // Varie operazioni, compresa:
    public SignalWaiterOrWatcher asyncWrite(/* argomenti... */) {
        SignalWaiterOrWatcher reply = new PersistentSignal();
        // Avvia l'operazione di scrittura
        // (ma senza aspettare che termini)
        return reply;
    }
}
```

Il metodo `asyncWrite` permette di effettuare scritture asincrone: esso crea un'istanza di `PersistentSignal`, e la restituisce dopo aver iniziato la scrittura, ma *prima* che essa sia completata.

Quando la scrittura sarà completata, verrà invocato il metodo `send` del segnale. Inoltre, in caso di fallimento della scrittura, sarebbe opportuno interrompere l'attesa di eventuali thread bloccati sul segnale (mediante `interrupt`), in modo che questi possano gestire tale fallimento.

Il chiamante riceve un riferimento al segnale, e può usarlo per capire quando la scrittura sia completata,

- mettendosi in attesa:

```
public static void main(String[] args) {
    DiskController controller = new DiskController();

    // Preparazione dei dati da scrivere...

    SignalWaiterOrWatcher outputDone =
        controller.asyncWrite(/* ... */);

    // ...

    // Quando è il momento di controllare
    // che l'output sia completato:
    try {
        outputDone.waits();
    } catch (InterruptedException e) {
        // La scrittura non è andata a buon fine,
        // quindi si avvia un'operazione di recovery.
    }
}
```

- oppure, mediante polling:

```
public static void main(String[] args) {
    DiskController controller = new DiskController();

    // Preparazione dei dati da scrivere...

    SignalWaiterOrWatcher outputDone =
        controller.asyncWrite(/* ... */);

    // ...

    // Quando è il momento di controllare
    // che l'output sia completato:
    if (!outputDone.watch()) {
        // La scrittura non è completata,
```

```
            // quindi si avvia un'operazione di recovery.
        }
    }
```

Nota: In questo esempio, quando si controlla l'esito della scrittura, se questa non è ancora completata si avvia subito un'operazione di recovery, senza aspettare ulteriormente che termini). Alternativamente, si potrebbe eseguire il controllo in un ciclo (cioè fare un "vero e proprio" polling), e nel frattempo, ad esempio, continuare con altre elaborazioni.

2.1.2 Attesa con timeout

Per consentire ai thread che attendono un segnale di specificare un tempo massimo di attesa, si aggiunge a `SignalWaiter` un metodo `waits(t)`, che riceve come argomento un timeout, e restituisce un valore booleano per indicare se è arrivato il segnale (`true`) o se è scaduto il timeout (`false`):

```
public interface SignalWaiter {
    void waits() throws InterruptedException;
    boolean waits(long timeout) throws InterruptedException;
}
```

L'implementazione di questo nuovo metodo nella classe `PersistentSignal` è la seguente:

```
public class PersistentSignal
        extends Signal
        implements SignalWaiterOrWatcher {
    public synchronized boolean waits(long timeout)
            throws InterruptedException {
        long callTime = System.currentTimeMillis();
        long elapsed = 0;
        while (!arrived && elapsed < timeout) {
            wait(timeout - elapsed);
            if (arrived) {
                arrived = false;
                return true;
            } else {
                elapsed = System.currentTimeMillis() - callTime;
            }
        }
        return false;
    }
```

```
    // Altri metodi come prima
}
```

Riprendendo l'esempio di prima, l'attesa con timeout costituisce una terza opzione per il controllo del completamento di una scrittura su disco:

```
public static void main(String[] args) {
    DiskController controller = new DiskController();

    // Preparazione dei dati da scrivere...

    SignalWaiterOrWatcher outputDone = controller.asyncWrite(/* ... */);

    // ...

    // Quando è il momento di controllare che l'output sia completato:
    try {
        if (outputDone.waits(345)) {
            System.out.println("Write completed");
        } else {
            System.out.println("Write not completed");
        }
    } catch (InterruptedException e) {
        // La scrittura non è andata a buon fine,
        // quindi si avvia un'operazione di recovery.
    }
}
```

2.2 Transient signal

Nell'implementare i segnali transienti, bisogna decidere cosa succede quando arriva un segnale e ci sono in attesa più thread: si può scegliere di sbloccarne solo uno, oppure di sbloccarli tutti. La soluzione migliore è implementare entrambe le possibilità. Per ora, si crea la classe `TransientSignal`, che sveglia un solo thread:

```
public class TransientSignal extends Signal {
    protected int waiting = 0;

    public synchronized void send() {
        if (waiting > 0) {
            super.send();
        }
    }
```

```
    public synchronized void waits() throws InterruptedException {
        waiting++;
        try {
            while (!arrived) {
                wait();
            }
            arrived = false;
        } finally {
            waiting--;
        }
    }
}
```

Osservazioni:

- Il metodo `send` implementato dalla superclasse `Signal` (che contiene una `notify`, quindi sveglia un singolo thread) viene chiamato solo se ci sono effettivamente thread in attesa, per fare in modo che il segnale si perda se invece non ce ne sono. Altrimenti, in assenza di thread da svegliare, il flag `arrived` rimarrebbe impostato a `true`, e quindi il segnale si comporterebbe in modo persistente.
- Nell'implementazione di `waits`, è importante decrementare il numero di thread in attesa anche in caso di `InterruptedException`, quindi quest'operazione viene messa in un `finally`. Non è invece necessario impostare `arrived` a `false`, perché se l'attesa termina a causa di una `interrupt` significa che il segnale non è arrivato.
- `TransientSignal` non fornisce un metodo `watch`, perché non deve essere possibile controllare a posteriori se il segnale sia arrivato: come già detto, l'unico scopo di un segnale transiente è svegliare gli eventuali thread che sono in attesa al momento del suo arrivo.

Si definisce poi una classe `Pulse`, la quale estende `TransientSignal`, mettendo a disposizione un metodo aggiuntivo `sendAll()` che sveglia tutti i thread in attesa:

```
public class Pulse extends TransientSignal {
    public synchronized void sendAll() {
        if (waiting > 0) {
            arrived = true;
            notifyAll();
        }
    }

    public synchronized void waits() throws InterruptedException {
        waiting++;
        try {
            while (!arrived) {
```

```
                wait();
            }
        } finally {
            waiting--;
            if (waiting == 0) {
                arrived = false;
            }
        }
    }
}
```

`Pulse` ridefinisce anche il metodo `waits`, perché solo l'ultimo thread svegliato deve resettare il flag `arrived`:[1] altrimenti, una volta resettato, i thread usciti dalla `wait` penserebbero che il segnale non sia arrivato, e si rimetterebbero in attesa.

2.2.1 Esempio: attesa per i saldi

Durante la stagione dei saldi, un negozio decide di aprire le porte solo a intervalli di tempo di N secondi:

1. i clienti che arrivano quando le porte sono chiuse aspettano;
2. quando le porte vengono aperte, i clienti in attesa entrano;
3. le porte si richiudono subito dopo che i clienti sono entrati.

Nella simulazione di questa situazione, l'apertura delle porte è rappresentata da un segnale transiente, e i clienti sono thread che attendono tale segnale:

```
public class Cliente extends Thread {
    private Pulse portaNegozio;

    public Cliente(Pulse portaNegozio) {
        this.portaNegozio = portaNegozio;
        setName("Cliente " + getName());
    }
```

[1] Tecnicamente, questa ridefinizione significa che il risveglio di un singolo thread, attraverso il metodo `send()` ereditato da `TransientSignal`, non funziona più correttamente se ci sono thread multipli in attesa: il flag `arrived` rimane impostato, quindi i thread che eventualmente chiameranno `waits()` in seguito non si bloccheranno. Una possibile soluzione sarebbe tenere traccia di quale metodo di risveglio sia stato invocato (`send` o `sendAll`), e modificare `waits()` in modo che il singolo thread svegliato da `send` azzeri il flag anche se non è l'unico in attesa. In alternativa, `Pulse` potrebbe supportare solo il risveglio di tutti i thread in attesa (allora, non dovrebbe essere una sottoclasse di `TransientSignal`), evitando così qualsiasi possibilità di conflitto tra i due metodi di risveglio, a scapito della versatilità di questa classe.

```
    public void run() {
        System.out.println(getName() + ": mi metto in coda");
        try {
            portaNegozio.waits();
            System.out.println(getName() + ": entrato nel negozio");
        } catch (InterruptedException e) {}
    }
}
```

Anche il negozio è un thread, che periodicamente invia un segnale ("apre le porte"):

```
public class Negozio extends Thread {
    private Pulse portaNegozio;

    public Negozio(Pulse portaNegozio) {
        this.portaNegozio = portaNegozio;
        setName("Negozio " + getName());
    }

    public void run() {
        try {
            while (true) {
                Thread.sleep(1000);
                System.out.println(getName() + ": apro le porte");
                portaNegozio.sendAll();
                System.out.println(getName() + ": chiudo le porte");
            }
        } catch (InterruptedException e) {}
    }
}
```

Osservazione: Di fatto, la porta è chiusa dall'ultimo cliente che entra, il quale resetta il flag del segnale.

Infine, il main istanzia un oggetto `Pulse` condiviso, avvia il thread `Negozio`, e poi, periodicamente, crea un numero casuale di clienti, che si mettono in coda:

```
import java.util.Random;

public class Saldi {
    public static void main(String[] args) throws InterruptedException {
        Pulse portaNegozio = new Pulse();
        new Negozio(portaNegozio).start();

        Random rnd = new Random();
```

```
        while (true) {
            int numClienti = rnd.nextInt(5);
            for (int i = 0; i < numClienti; i++) {
                new Cliente(portaNegozio).start();
            }
            Thread.sleep(300);
        }
    }
}
```

Un esempio di output è il seguente:

```
Cliente Thread-2: mi metto in coda
Cliente Thread-3: mi metto in coda
Cliente Thread-1: mi metto in coda
Cliente Thread-4: mi metto in coda
Cliente Thread-5: mi metto in coda
Cliente Thread-6: mi metto in coda
Cliente Thread-7: mi metto in coda
Negozio Thread-0: apro le porte
Negozio Thread-0: chiudo le porte
Cliente Thread-2: entrato nel negozio
Cliente Thread-1: entrato nel negozio
Cliente Thread-5: entrato nel negozio
Cliente Thread-4: entrato nel negozio
Cliente Thread-7: entrato nel negozio
Cliente Thread-3: entrato nel negozio
Cliente Thread-6: entrato nel negozio
Cliente Thread-8: mi metto in coda
Cliente Thread-9: mi metto in coda
Cliente Thread-10: mi metto in coda
Cliente Thread-11: mi metto in coda
Cliente Thread-12: mi metto in coda
Cliente Thread-13: mi metto in coda
Negozio Thread-0: apro le porte
Negozio Thread-0: chiudo le porte
Cliente Thread-11: entrato nel negozio
Cliente Thread-9: entrato nel negozio
Cliente Thread-12: entrato nel negozio
Cliente Thread-13: entrato nel negozio
Cliente Thread-8: entrato nel negozio
Cliente Thread-10: entrato nel negozio
Cliente Thread-14: mi metto in coda
...
```

3 Buffer

Un **buffer** contiene dati che, una volta letti, verranno distrutti. Se i dati devono invece essere conservati, è più appropriato il paradigma Blackboard, che verrà presentato più avanti.

Di seguito, viene mostrata un'implementazione di un `BoundedBuffer` generico (in particolare, una coda con capacità limitata), di fatto uguale a quella già vista nell'ambito del problema del produttore-consumatore:

```
public class BoundedBuffer<Data> {
    private final int size;
    private final Data[] buffer;
    private int numItems = 0;
    private int first = 0;
    private int last = 0;

    public BoundedBuffer(int size) {
        this.size = size;
        buffer = (Data[]) new Object[size];
    }

    public synchronized void put(Data item) throws InterruptedException {
        while (numItems == size) {
            wait();
        }
        buffer[last] = item;
        last = (last + 1) % size;
        numItems++;
        notifyAll();
    }

    public synchronized Data get() throws InterruptedException {
        while (numItems == 0) {
            wait();
        }
        Data item = buffer[first];
        first = (first + 1) % size;
        numItems--;
        notifyAll();
        return item;
    }
}
```

3.1 Esempio: produttore e consumatore di frutta

L'applicazione tipica dei buffer è, appunto, il problema del produttore-consumatore. Come esempio, viene simulata la compravendita di frutta al mercato:

- Il prezzo E varia in funzione del peso P:

$$E = \frac{P^2}{2}$$

- Il peso può assumere valori nell'intervallo $[0.1, 2.5)$.
- Alla fine, il produttore stampa il proprio ricavo, con un output del tipo

  `NomeThread: Guadagnati in totale ... euro`

  mentre il consumatore stampa il peso totale della frutta comprata:

  `NomeThread: Comprati in totale ... kg`

- Si creano 3 produttori e 3 consumatori.

```
public class Frutta {
    private final double peso;

    public Frutta(double peso) {
        this.peso = peso;
    }

    public double getPeso() {
        return peso;
    }

    public double getPrezzo() {
        return peso * peso / 2;
    }
}

public class Consumatore extends Thread {
    private BoundedBuffer<Frutta> mercato;
    private double pesoTotale = 0;

    public Consumatore(String nome, BoundedBuffer<Frutta> mercato) {
        setName(nome);
        this.mercato = mercato;
    }
```

```
    public void run() {
        for (int i = 0; i < 20; i++) {
            Frutta f;
            try {
                f = mercato.get(); // Si blocca se il mercato è vuoto
            } catch (InterruptedException e) { break; }
            pesoTotale += f.getPeso();
            System.out.printf(
                "%s: Comprati %.2f kg\n",
                getName(), f.getPeso()
            );
        }

        System.out.printf(
            "%s: Comprati in totale %.2f kg\n",
            getName(), pesoTotale
        );
    }
}

public class Produttore extends Thread {
    private BoundedBuffer<Frutta> mercato;
    private double ricavoTotale = 0;

    public Produttore(String nome, BoundedBuffer<Frutta> mercato) {
        setName(nome);
        this.mercato = mercato;
    }

    public void run() {
        for (int i = 0; i < 20; i++) {
            Frutta f = new Frutta(Math.random() * 2.4 + 0.1);
            System.out.printf(
                "%s: Vende %.2f kg\n",
                getName(), f.getPeso()
            );
            try {
                mercato.put(f); // Si blocca se il mercato è pieno
            } catch (InterruptedException e) { break; }
            ricavoTotale += f.getPrezzo();
            System.out.printf(
                "%s: Guagadnati %.2f euro\n",
                getName(), f.getPrezzo()
            );
```

```
        }

        System.out.printf(
            "%s: Guadagnati in totale %.2f euro\n",
            getName(), ricavoTotale
        );
    }
}

public class ProdConsFrutta {
    public static void main(String[] args) {
        BoundedBuffer<Frutta> mercato = new BoundedBuffer<>(8);
        for (int i = 0; i < 3; i++) {
            new Produttore("P" + i, mercato).start();
            new Consumatore("C" + i, mercato).start();
        }
    }
}
```

Osservazione: Questo non è un bellissimo esempio: nella realtà, la compravendita è una transazione, quindi il produttore dovrebbe incassare solo nel momento in cui il consumatore preleva la merce dal buffer. Tuttavia, l'esempio regge se si interpretano:

- il produttore come colui che produce la frutta, e incassa nel momento in cui la porta al mercato;
- il consumatore come colui che compra dal mercato;
- il mercato come un intermediario, che, di fatto, compra dal produttore e vende al consumatore.