Azzolini Riccardo

Serializzazione

1 Stream di oggetti e socket

Per trasferire oggetti (o, più in generale, grafi di oggetti) Java da un client a un server (e/o viceversa), si possono costruire su un socket degli stream di tipo `ObjectInputStream` e `ObjectOutputStream` (invece degli stream di caratteri utilizzati finora).

Questi stream inviano i dati in un formato binario, al quale gli oggetti vengono convertiti mediante la serializzazione. È quindi necessario che gli oggetti da trasferire siano appunto serializzabili (cioè che implementino l'interfaccia `Serializable`, e rispettino le altre regole della serializzazione).

2 Esempio: invio di una stringa

Nell'esempio che segue, il client si connette al server e riceve da esso un oggetto di tipo `String`. Il codice di questo client è in gran parte analogo a quello già visto in tanti altri esempi: cambia solo il tipo di stream "montato" sul socket.

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.net.InetAddress;
import java.net.Socket;

public class ReceiverClient {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        System.out.println("Receiver Start");
        InetAddress addr = InetAddress.getByName(null);
        try (
            Socket socket = new Socket(addr, SenderServer.PORT);
            ObjectInputStream in =
                new ObjectInputStream(socket.getInputStream())
        ) {
            System.out.println("socket = " + socket);
            String s = (String) in.readObject();
            System.out.println("String is: '" + s + "'");
```

```
            System.out.println("Closing...");
        }
        System.out.println("Receiver End");
    }
}
```

Anche nel codice del server, che si mette in attesa di un client al quale inviare la stringa, l'unica "novità" è il tipo di stream usato:

```
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class SenderServer {
    public static final int PORT = 9999;

    public static void main(String[] args) throws IOException {
        try (
            ServerSocket server = new ServerSocket(PORT);
            Socket socket = server.accept();
            ObjectOutputStream out =
                new ObjectOutputStream(socket.getOutputStream())
        ) {
            System.out.println("Connection accepted: " + socket);
            out.writeObject("test");
            System.out.println("Connection ended\nClosing...");
        }
    }
}
```

3 Esempio: invio di un array

Adesso, invece di inviare una singola stringa, si invia un oggetto di tipo `String[]`. In pratica, a parte il tipo dell'oggetto, il codice per l'invio e la ricezione rimane uguale, dato che Java è automaticamente in grado di serializzare sia singole stringhe che array di stringhe.

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.net.InetAddress;
import java.net.Socket;
```

```java
public class ReceiverClient {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        System.out.println("Receiver Start");
        InetAddress addr = InetAddress.getByName(null);
        try (
            Socket socket = new Socket(addr, SenderServer.PORT);
            ObjectInputStream in =
                new ObjectInputStream(socket.getInputStream())
        ) {
            System.out.println("socket = " + socket);
            String[] array = (String[]) in.readObject();
            System.out.print("Strings are:");
            for (String s : array) {
                System.out.print(" '" + s + "',");
            }
            System.out.println("\nClosing...");
        }
        System.out.println("Receiver End");
    }
}
```

```java
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class SenderServer {
    public static final int PORT = 9999;

    public static void main(String[] args) throws IOException {
        try (
            ServerSocket server = new ServerSocket(PORT);
            Socket socket = server.accept();
            ObjectOutputStream out =
                new ObjectOutputStream(socket.getOutputStream())
        ) {
            System.out.println("Connection accepted: " + socket);
            String[] obj = new String[] {"giallo", "rosso"};
            out.writeObject(obj);
            System.out.println("Connection ended\nClosing...");
        }
    }
}
```

4 ClassNotFoundException

Quanto si trasferisce un oggetto serializzato, viene ricevuta una sequenza di byte, che il ricevente deve sapere come interpretare per ricostruire l'oggetto originale.

Se l'oggetto serializzato è di tipo primitivo, la JVM sa sempre come interpretare i byte per deserializzarlo. Altrimenti, è necessario conoscere il codice della classe di cui l'oggetto è istanza. Se tale codice (in particolare, il bytecode della classe) non è disponibile, la deserializzazione solleva una `ClassNotFoundException`.

4.1 Esempio

Si consideri la classe `Punto`, già usata per esempi precedenti di serializzazione:

```
import java.io.Serializable;

public class Punto implements Serializable {
    private static final long serialVersionUID = 1;
    private int x, y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "Il punto ha coordinate " + x + " e " + y;
    }
}
```

Innanzitutto, si modifica il server in modo che invii al client un'istanza di `Punto`. Siccome `Punto` è serializzabile, questo codice funzionerà senza alcun errore.

```
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class SenderServer {
    public static void main(String[] args) throws IOException {
        try (
            ServerSocket server = new ServerSocket(9999);
            Socket socket = server.accept();
            ObjectOutputStream out =
```

```
                new ObjectOutputStream(socket.getOutputStream())
        ) {
            Punto p = new Punto(6, 11);
            out.writeObject(p);
        }
    }
}
```

Si suppone che invece il client *non* conosca la classe `Punto` (cioè, in particolare, che non abbia accesso al file `Punto.class`). Dunque, esso prova a leggere l'oggetto inviato dal server e salvarlo semplicemente in una variabile `Object`: non può eseguire un cast a `Punto` perché altrimenti, in assenza della definizione di tale classe, non si potrebbe neanche compilare il codice.

Anche senza il cast, però, l'oggetto da deserializzare è comunque di tipo `Punto`, quindi la JVM non riesce a ricostruirlo, e viene sollevata una `ClassNotFoundException`.

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.net.InetAddress;
import java.net.Socket;

public class ReceiverClient {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        InetAddress addr = InetAddress.getByName(null);
        try (
            Socket socket = new Socket(addr, 9999);
            ObjectInputStream in =
                new ObjectInputStream(socket.getInputStream())
        ) {
            Object p = in.readObject();
            System.out.println("Object is: " + p);
        } catch (ClassNotFoundException e) {
            System.err.println("Unknown object class");
        }
    }
}
```

La soluzione è fare in modo che il anche il programma client contenga la definizione della classe `Punto`, che deve essere la stessa usata dal server.

5 Quando la serializzazione di default non è abbastanza

Ci sono casi in cui la serializzazione base fornita automaticamente da Java non è sufficiente, ed è invece necessario definire manualmente alcuni dettagli relativi alle operazioni di serializzazione e/o deserializzazione.

Il seguente esempio illustra uno di questi casi. Esso è abbastanza complicato, poiché la serializzazione di default è adatta praticamente a tutte le situazioni più semplici.

5.1 Esempio: il problema

La classe che si vorrà serializzare è `PersistentClock`: quando viene istanziata, essa crea un thread che stampa la data e l'ora correnti a intervalli regolari.

```
import java.util.Date;

public class PersistentClock implements Runnable {
    private Thread animator;
    private long animationInterval;

    public PersistentClock(int animationInterval) {
        this.animationInterval = animationInterval;
        animator = new Thread(this);
        animator.start();
    }

    public void run() {
        while (true) {
            System.out.println(new Date());
            try {
                Thread.sleep(animationInterval);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        new PersistentClock(1000);
    }
}
```

L'esecuzione di questa classe produce un output del genere:

```
Thu Jun 04 10:32:41 CEST 2020
Thu Jun 04 10:32:42 CEST 2020
Thu Jun 04 10:32:43 CEST 2020
Thu Jun 04 10:32:44 CEST 2020
Thu Jun 04 10:32:45 CEST 2020
Thu Jun 04 10:32:46 CEST 2020
...
```

La serializzazione di un'istanza di `PersistentClock` viene eseguita come al solito. In particolare, il campo `animator` deve essere dichiarato `transient`, poiché contiene un riferimento a un oggetto `Thread`, che non è serializzabile. Per semplicità, l'oggetto serializzato viene scritto in un file, ma lo si potrebbe equivalentemente inviare su un socket.

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

public class PersistentClock implements Serializable, Runnable {
    private static final long serialVersionUID = 1;
    private transient Thread animator;
    private long animationInterval;

    public PersistentClock(int animationInterval) {
        // Uguale a prima
    }

    public void run() {
        // Uguale a prima
    }

    public static void main(String[] args) throws IOException {
        PersistentClock p = new PersistentClock(1000);
        try (
            ObjectOutput os =
                new ObjectOutputStream(new FileOutputStream("tmp.ser"))
        ) {
            os.writeObject(p);
            os.flush();
        }
    }
}
```

Adesso, si prova a deserializzare l'oggetto. Per comodità, la deserializzazione viene eseguita nel main della classe `PersistentClock` stessa, sostituendo il codice che crea e serializza un'istanza.

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

public class PersistentClock implements Serializable, Runnable {
    // ...uguale a prima

    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        try (
            ObjectInput is =
                new ObjectInputStream(new FileInputStream("tmp.ser"))
        ) {
            PersistentClock p = (PersistentClock) is.readObject();
        }
    }
}
```

Quando si deserializza il `PersistentClock` in questo modo, viene ricostruito l'oggetto, ma non il thread `animator`: in pratica, la ricostruzione è parziale. Infatti, in questo esempio, lo stato (e anche il comportamento) dell'oggetto dipende fortemente da ciò che viene fatto nel costruttore: è proprio il costruttore a creare e avviare il thread `animator`.

Il problema è che, quando si deserializza un oggetto, il costruttore *non* viene chiamato – come è giusto che sia, dato che non si sta creando una nuova istanza della classe. Allora, bisogna personalizzare la deserializzazione in modo da ricreare manualmente l'`animator`.

5.2 Serializzazione personalizzata

Java consente di personalizzare il meccanismo di serializzazione / deserializzazione: nelle classi da serializzare, si possono definire due metodi,

```
private void writeObject(ObjectOutputStream out)
    throws IOException;
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

che verranno chiamati invece di applicare la serializzazione di default.

All'interno di questi metodi, si può richiamare il comportamento di default, invocando rispettivamente `out.defaultWriteObject()` e `in.defaultReadObject()`. Ciò è utile,

ad esempio, se si vogliono solo aggiungere delle operazioni in più, senza invece cambiare come vengono scritti i campi (non `static` e non `transient`) dell'oggetto.

5.3 Esempio: la soluzione

Nel caso di `PersistentClock`, la serializzazione di default è sufficiente, mentre è necessario personalizzare la deserializzazione, aggiungendo la creazione e l'avviamento dell'`animator`:

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

public class PersistentClock implements Serializable, Runnable {
    private static final long serialVersionUID = 1;
    private transient Thread animator;
    private long animationInterval;

    public PersistentClock(int animationInterval) {
        this.animationInterval = animationInterval;
        startAnimation();
    }

    private void startAnimation() {
        animator = new Thread(this);
        animator.start();
    }

    public void run() {
        // Uguale a prima
    }

    private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        startAnimation();
    }

    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        try (
            ObjectInput is =
                new ObjectInputStream(new FileInputStream("tmp.ser"))
        ) {
```

```
            PersistentClock p = (PersistentClock) is.readObject();
        }
    }
}
```

Così, quando si deserializza il `PersistentClock`, l'output riparte, come desiderato.

5.4 Altro esempio: serializzazione di un campo statico

Come altro esempio di serializzazione / deserializzazione personalizzata, si suppone di voler salvare e ripristinare un campo statico (cosa che, di default, non avviene). Per fare ciò, si implementano sia `writeObject` che `readObject`, nei quali vengono rispettivamente scritti e letti i due campi `dat` e `sdat`, uno dopo l'altro:

```
import java.io.*;

public class DemoClass implements Serializable {
    private int dat = 3;
    private static int sdat = 2;

    private void writeObject(ObjectOutputStream out)
            throws IOException {
        out.writeInt(dat);
        out.writeInt(sdat);
    }

    private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException {
        dat = in.readInt();
        sdat = in.readInt();
    }

    public String toString() {
        return "DemoClass: " + dat + " " + sdat;
    }

    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        try (
            ObjectOutput os =
                new ObjectOutputStream(new FileOutputStream("tmp.ser"))
        ) {
            DemoClass obj = new DemoClass();
```

```
            os.writeObject(obj);
            os.flush();
        }

        try (
            ObjectInput is =
                new ObjectInputStream(new FileInputStream("tmp.ser"))
        ) {
            DemoClass obj = (DemoClass) is.readObject();
            System.out.println(obj);
        }
    }
}
```

6 Controllo di versione

Come già anticipato, l'identificatore univoco della versione `serialVersionUID` della classe usata per deserializzare un oggetto deve essere uguale a quello dell'oggetto serializzato.

Un tipico esempio di scenario in cui ciò non avviene è il seguente:

1. si definisce una classe, se ne crea un'istanza, la si serializza e la si salva in un file;
2. successivamente, si aggiorna il codice che definisce la classe, magari aggiungendo un nuovo campo, e, per indicare la modifica, si cambia il `serialVersionUID`;
3. quando si tenta di deserializzare l'oggetto salvato, siccome è cambiato l'identificatore della versione viene generata una `InvalidClassException`.

Un altro caso comune si verifica in ambito distribuito, quando programmi diversi hanno versioni diverse della stessa classe.

L'esistenza di questo meccanismo di controllo delle versioni è molto importante, poiché permette di accorgersi immediatamente di eventuali problemi legati a versioni diverse di una classe. Perciò, è buona norma definire sempre manualmente il `serialVersionUID`, e modificarlo quando si modifica la classe; altrimenti, la JVM assegna un valore di default.