Java e Serializzazione
dalla A all'XML

di Leonardo Puleggi

Indice generale

Introduzione

La **serializzazione** è un'attività che permette di salvare un oggetto o un grafo di oggetti in uno stream di byte che successivamente può essere salvato in maniera persistente ad esempio in un file oppure inviato sulla rete.

La **deserializzazione** è il processo inverso, che permette, cioè, di ricostruire l'oggetto o il grafo di oggetti dallo stream di byte.

Per capire come funziona iniziamo subito con un esempio molto semplice; abbiamo una classe Punto che rappresenta un punto:) nello spazio euclideo in base alle due variabili di istanza x e y.

Ecco il codice di questa semplicissima classe con evidenziate in rosso le parti relative alla serializzazione.

```
import java.io.Serializable;

public class Punto implements Serializable{
	private static final long serialVersionUID = 1;

	private int x;
	private int y;


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

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

Come è possibile vedere l'unica cosa che dobbiamo fare è :

1. far implementare alla classe l'interfaccia Serializable
2. definire una versione relativa alla classe (se non inserito la JVM provvede a calcolarlo in maniera autonoma, e questo potrebbe portare a errori)

La prima delle due richieste è ovvia (anche se ci potrebbe sorprendere il fatto che non abbiamo definito nessun metodo relativo all'interfaccia implementata, ma vedremo dopo che questo sarà nostro compito solo in casi particolari); in pratica si tratta di una interfaccia marker.

La versione serve in fase di deserializzazione per verificare che la classi usate

da chi ha serializzato l'oggetto e chi lo sta deserializzando siano compatibili; se, infatti, le versioni non corrispondono sarà sollevata una InvalidClassException.
Una classe può dichiarare la propria versione usando un campo chiamato serialVersionUID che deve essere static, final e di tipo long, quindi:

<Ogni-Modificatore-di-Accesso> **static final long serialVersionUID=1L;**

Vediamo ora il codice per serializzare un oggetto Punto in uno stream di byte (vedi esempio_01):

```
import java.io.*;

public class SerializzaPunto {
	public static void main(String[] args) {

		Punto p=new Punto(2,3);
		ObjectOutputStream output=null;
		try{
			output=new ObjectOutputStream(new FileOutputStream("dati.dat"));
		}
		catch (FileNotFoundException e) {
			.....
		}
		catch(IOException ioe){
			....
		}
		try {
			output.writeObject(p);
		} catch (IOException e1) {
			....
		}

		try {
			output.close();
		} catch (IOException e2) {
			....
		}
		System.out.println("Serializzazione completata.");
	}
}
```

Come è possibile vederei passi base per la serializzazione sono:

1. Definire l'oggetto da serializzare :)
2. Definire un ObjectOutputStream su cui andremo a salvare il nostro flusso di byte (in questo esempio lo stream sarà salvato in un file ma niente vieta di usare anche altri strumenti,vedi esempi con la rete).
3. Scrivere l'oggetto.
4. Chiudere lo stream.

Vediamo ora il procedimento inverso, ovvero la deserializzazione:

```
import java.io.*;

public class DeSerializzaPunto {

	public static void main(String[] args) {

		ObjectInputStream ois = null;
		try {
			ois = new ObjectInputStream(new FileInputStream("dati.dat"));
		} catch (FileNotFoundException e) {
			...
		} catch (IOException e) {
			...
		}

		Punto p = null;
		try {
			p = (Punto) ois.readObject();
		} catch (IOException e1) {
			.....
		} catch (ClassNotFoundException e1) {
			.....
		}
		System.out.println(p.toString());
	}
}
```

Come possiamo vedere il procedimento è banalmente simile:

1. Si definisce un ObjectInputStream (che, in questo caso, prenderà il flusso dal file creato in precedenza)
2. Definiamo un Punto e proviamo a leggerlo dallo stream

Grafo di Riferimenti

Abbiamo visto come sia possibile serializzare un oggetto semplice che ha come variabili di istanza solo dati primitivi, ma cosa succede se invece la nostra classe definisce variabili di istanza?
Vedremo che viene seguito il grafo dei riferimenti in modo che sia completamente ricostruibile lo stato dell'oggetto al momento della serializzazione.
Vediamo un esempio in cui abbiamo due classi:

- Un Padre che ha come variabile di istanza una lista dei suoi Figli
- Una classe Figlio

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

public class Padre implements Serializable {
	private static final long serialVersionUID = 1L;
	private String nome;
	private String cognome;
	private Collection<Figlio> figli;

	public Padre(String nome,String cognome){
		this.nome=nome;
		this.cognome=cognome;
		figli=new ArrayList<Figlio>();
	}
	.....ecc
}
```

```
import java.io.Serializable;

public class Figlio implements Serializable{
	private static final long serialVersionUID = 1L;
	private String nome;
	private String cognome;

	public Figlio(String nome,String cognome){
		this.nome=nome;
		this.cognome=cognome;
	}

	public String toString(){
		return "nome: "+nome+" cognome: "+cognome;
	}
}
```

Per serializzare un oggetto di tipo Padre non dobbiamo fare altro che richiamare il metodo writeObject esattamente come abbiamo fatto prima; se tutti gli oggetti che è possibile trovare nel grafo dei riferimenti sono serializzabili allora non ci saranno problemi.
Vediamo come è possibile fare facilmente quanto indicato (vedi esempio_02):

```
public class SerializzaPadre {
	public static void main(String[] args) {
		Figlio a=new Figlio("leo","puleggi");
		Figlio b=new Figlio("ale","puleggi");

		Padre p=new Padre("claudio","puleggi");
		p.aggiungiFiglio(a);
		p.aggiungiFiglio(b);

		ObjectOutputStream output=null;
```

```
		try{
			output=new ObjectOutputStream(new FileOutputStream("dati.dat"));
		}
		catch (FileNotFoundException e) {
			....
		}
		catch(IOException ioe){
			....
		}

		try {
			output.writeObject(p);
		} catch (IOException e1) {
			....
		}
		try {
			output.close();
		} catch (IOException e2) {
			....
		}
		System.out.println("Serializzazione completata.");
	}
}
```

La deserializzazione è simile a quanto già visto in precedenza.
Come abbiamo visto è possibile ricostruire completamente il grafo dei riferimenti dell'oggetto Padre; per realizzarlo si sfrutta la chiusura transitiva, cioè vengono serializzati anche tutti gli oggetti referenziati.
E' banale individuare alcuni problemi :

- Serializzando un oggetto si rischia di serializzare tutta l'applicazione
- Cosa accade se uno degli oggetti referenziati non è serializzabile??

Per quanto riguarda il secondo problema viene sollevata una eccezione NotSerializableException che porta il programma a fallire (vedi l'esempio 03, uguale al 02 tranne per il fatto che Figlio non è serializzabile).
Per il primo problema la questione è più complessa e sarà analizzata nel prossimo capitolo.

Attributi Transient

Per far si che nella serializzazione non vengano coinvolti particolari oggetti istanza si sfrutta la parola chiava **transient** che specifica di non serializzare quell'attributo.

Vediamo come è possibile modificare l'esempio 02 per ottenere che la collezione contenente i figli non sia serializzata con l'oggetto Padre.

```
public class Padre implements Serializable {
        private static final long serialVersionUID = 1L;
        private String nome;
        private String cognome;
        transient private Collection<Figlio> figli;
```

Andando ad eseguire l'esempio_04 notiamo infatti che il riferimento alla collezione è un null, cosa che ci aspettavamo.

Con la parola chiave transient siamo riusciti ad impedire di seguire tutto il grafo dei riferimenti, ma si pone il problema di ricostruire, in fase di deserializzazione, le informazioni che avevamo volutamente omesso.

Metodi writeObject e readObject

Per risolvere il problema con il quale abbiamo concluso il capitolo precedente possiamo adottare due tecniche (che consistono sostanzialmente nella definizione dei metodi di serializzazione e deserializzazione che finora abbiamo sempre ignorato).

Le due tecniche sono:

1. Ridefinire i due metodi privati responsabili della scrittura e lettura degli oggetti sugli stream
2. Implementare l'interfaccia Externalizable e definire in modo manuale come salvare e leggere da stream i vari oggetti definendo due metodi :

   void writeExternal(ObjectOutput out)
   void readExternal(ObjectInput in)

Vediamo prima un esempio per capire la prima tecnica; ipotizziamo di aver bisogno di un particolare programma che si sposta su vari computer e a seconda del sistema operativo deve eseguire determinate operazioni.
Abbiamo quindi una variabile di istanza che per semplicità è una stringa che rappresenta il sistema operativo (vedi esempio_05) e che dobbiamo escludere dalla serializzazione e ricostruire in fase di deserializzazione.

```
import java.io.*;

public class Agente implements Serializable{
	private static final long serialVersionUID = 1L;
	private String nome;
	transient private String piattaforma;

	public Agente(String nome){
		this.nome=nome;
		this.piattaforma=System.getProperty("os.name");
	}
	private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
		ois.defaultReadObject();
		piattaforma = System.getProperty("os.name");
	}
}
```

Come è possibile vedere ridefinendo il metodo è possibile specificare come ricostruire eventuali informazioni che non era possibile inserire nella serializzazione; in questo modo sarà possibile ad esempio far agire l'Agente su un certo sistema, quando questo avrà completato sarà serializzato insieme alle informazioni accumulate e sarà spostato su un nuovo sistema dove in comportamento dovrà essere dato ad esempio dal sistema operativo.
Questa tecnica può essere usata anche per ricostruire eventuali variabili statiche della classe che non vengono salvate dalla serializzazione (vedi esempio_06).
In questo esempio la classe Padre mantiene in una variabile statica il numero dei figli. Purtroppo se proviamo a serializzare semplicemente l'oggetto padre otteniamo un errore molto simpatico.
Eseguendo infatti l'esempio_06 otteniamo come output della serializzazione

```
Padre
nome: claudio
cognome: puleggi
Figli: [nome: leo cognome: puleggi, nome: ale cognome: puleggi]
Numero Figli: 0
```

Ovviamente si tratta di un errore; per risolverlo bisogna anche qui ridefinire il metodo per leggere l'oggetto dallo stream in modo che ricostruisca correttamente l'informazione mancante.
Vediamo il codice importante per questa operazione dall'esempio_07

```
private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
		ois.defaultReadObject();
		numero_figli=figli.size();
	}
```

La seconda tecnica, che consente di risolvere sia i problemi di eventuali variabili non serializzabili sia quelli legati a variabili statiche, consiste nell'implementare un'altra interfaccia, la Externalizable e definire in modo manuale come salvare e leggere da stream i vari oggetti definendo due metodi:

void writeExternal(ObjectOutput out)
void readExternal(ObjectInput in)

La differenza sostanziale con la prima tecnica è che definendo noi i metodi per la serializzazione possiamo ottenere un notevole vantaggio in termini di lunghezza dello stream; a fronte di questa maggiore efficienza dobbiamo abbandonare la semplicità del procedimento standard.
Questa soluzione può essere usata ad esempio per salvare le informazioni in un documento XML.
Vediamo l'esempio_08 in cui abbiamo una classe che rappresenta un proprietà dotata di nome e di un valore(entrambi stringhe):

```
import java.io.*;

public class Proprietà implements Externalizable{
	private static final long serialVersionUID = 1L;

	private String nome;
	private String valore;

	public Proprietà(){

	}

	public Proprietà(String nome,String valore){
		....
	}

	public String toString(){
		....
	}

	public void writeExternal(ObjectOutput out) throws java.io.IOException{
		out.writeObject(nome);
		out.writeObject("=");
		out.writeObject(valore);
	}

	public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException{
		nome=(String)in.readObject();
		String temp=(String)in.readObject();
		valore=(String)in.readObject();
	}
```

```
}
```

Come è possibile vedere dal codice, dobbiamo definire ovviamente i metodi richiesti dalla nuova interfaccia che indicano come andranno salvate le informazioni sullo stream (in questo caso abbiamo deciso di scrivere solamente la coppia con in mezzo un carattere =) ma anche un costruttore no-args(vedremo al momento della deserializzazione il perché).
Vediamo come è possibile serializzare un oggetto Proprietà.

```
import java.io.*;

public class SerializzaProprietà {

	public static void main(String[] args) {

		Proprietà p=new Proprietà("password","leonardo");

		ObjectOutputStream output=null;

		try{
			output=new ObjectOutputStream(new FileOutputStream("dati.dat"));
		}
		catch (FileNotFoundException e) {
			....
		}
		catch(IOException ioe){
			....
		}
		try {
			p.writeExternal(output);
		} catch (IOException e1) {
			....
		}

		try {
			output.close();
		} catch (IOException e2) {
			....
		}
		System.out.println("Serializzazione completata.");
	}
}
```

Più interessante è la deserializzazione:

```
import java.io.*;
public class DeSerializzaProprietà {

	public static void main(String[] args) {

		ObjectInputStream ois = null;
		try {
			ois = new ObjectInputStream(new FileInputStream("dati.dat"));
		} catch (FileNotFoundException e) {
			....
```

```
                } catch (IOException e) {
                        ....
                }

                Proprietà p=new Proprietà();
                try {
                        p.readExternal(ois);
                } catch (IOException e1) {
                        ....
                } catch (ClassNotFoundException e1) {
                        ....
                }
                System.out.println(p.toString());
        }
}
```

Ora è possibile capire la necessità del costruttore no-args che avevamo precedentemente definito; nella deserializzazione infatti creiamo un oggetto Proprietà che poi avrà la responsabilità di istanziare correttamente le sue variabile di istanza; infatti, come è possibile vedere dal codice, noi passiamo al metodo readExternal il flusso di byte in modo che poi lui possa usarlo per istanziare correttamente le sue variabile di istanza .

Ereditarietà e Serializzazione

Un altro uso tipico dell'interfaccia Externalizable è nel momento in cui ho la necessità di serializzare una classe che estende un'altra che non è serializzabile.
Il problema che si pone è quello illustrato nell'esempio_09 (che è la copia dell'esempio 12 in [1]).
In questo esempio abbiamo una classe Persona non serializzabile (e supponiamo non modificabile) e una classe Studente serializzabile che estende Persona. Nel momento della deserializzazione, bisogna quindi usare o il campo serialPersistentFields (che però non tratteremo ulteriormente) oppure usare l'interfaccia Externalizable.
L'output dell'esempio 09 è questo:

```
Serializzazione:
Creazione studente in corso...
Studente creato:
John Smith nazionalita Inglese matricola N=AA07594
Inizio serializzazione di John
Oggetto John serializzato
Deserializzazione:
Inizio ripristino studente
Ripristino completato
Info Studente:
```

null null nazionalita Italiana matricola N=AA07594

Come è possibile vedere vengono perse tutte le informazioni che riguardano la classe Persona che non è serializzabile.

Vediamo come sia possibile tramite l'interfaccia Externalizable con l'esempio_10:

```
package application;


public class Persona {

	private String nome;
	private String cognome;
	private String nazionalita;

	private static final String NAZIONALITA_DEFAULT = "Italiana";

	public Persona() {
		this.nazionalita = NAZIONALITA_DEFAULT;
	}


	public String getCognome() {
		return cognome;
	}
	public void setCognome(String cognome) {
		this.cognome = cognome;
	}
	public String getNazionalita() {
		return nazionalita;
	}
	public void setNazionalita(String nazionalita) {
		this.nazionalita = nazionalita;
	}
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}

	public String toString() {
		return nome + " "+ cognome + " nazionalita " + nazionalita;
	}
}

package application;

import java.io.*;

public class Studente extends Persona implements Externalizable{

	private String matricola;


	public String getMatricola() {
		return matricola;
```

```
	}
	public void setMatricola(String matricola) {
		this.matricola = matricola;
	}

	public String toString() {
		return super.toString() + " matricola N=" + matricola;
	}

	public void writeExternal(ObjectOutput out) throws java.io.IOException{
		out.writeObject(super.getNome());
		out.writeObject(super.getCognome());
		out.writeObject(super.getNazionalita());
		out.writeObject(matricola);
	}

	public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException{
		super.setNome((String)in.readObject());
		super.setCognome((String)in.readObject());
		super.setNazionalita((String)in.readObject());
		matricola=(String)in.readObject();
	}
}
```

Come è possibile vedere dal codice non abbiamo fatto altro che ottenere dalla superclasse le variabili di istanza, salvarle sullo stream e in fase di deserializzazione ricostruire lo stato della superclasse.

Serializzazione su Canali diversi

In questo capitolo vedremo come, tramite la serializzazione, è possibile “condividere”, tramite la rete, oggetti tra programmi differenti.
Vedremo come è possibile inviare lo stream di byte che rappresenta la serializzazione dell'oggetto su un pacchetto UDP.
Definiamo quindi le classi che andremo ad usare; una classe Figlio banale:

```
import java.io.Serializable;

public class Figlio implements Serializable{
	private static final long serialVersionUID = 1L;
	private String nome;
	private String cognome;

	public Figlio(String nome,String cognome){
		this.nome=nome;
		this.cognome=cognome;
	}

	public String toString(){
		return "nome: "+nome+" cognome: "+cognome;
	}
}
```

Vediamo il codice relativo al client che invierà il pacchetto contenente la serializzazione di una istanza di Figlio

```
import java.io.*;
import java.net.*;

public class FiglioUDPClient {
	public static void main(String[] args) {

		String nome_server=new String("localhost");
		int serverPort = 6789;

		try {
			ByteArrayOutputStream byteStrem=new ByteArrayOutputStream();
			ObjectOutputStream out=new ObjectOutputStream(byteStrem);
			out.writeObject(new Figlio("leo","puleggi"));
			out.close();
			byte[] dati=byteStrem.toByteArray();
			DatagramSocket udp_socket=new DatagramSocket();
			InetAddress server = InetAddress.getByName(nome_server);
			DatagramPacket pct_udp=new DatagramPacket(dati, dati.length, server,
serverPort);
			udp_socket.send(pct_udp);
		}
		catch(Exception e){
			....
		}
	}
}
```

Come è possibile vedere dal codice bisogna solo creare un ByteArrayOutputStream serializzare su questo l'oggetto e poi inviarlo com un qualunque array di byte all'interno del datagramma.

Il codice del server che si occuperà di ricevere il pacchetto e di estrarre le informazioni :

```
import java.io.*;
import java.net.*;

public class FiglioUDPServer {
	public static void main(String[] args) {

		int serverPort = 6789;

		try {
			DatagramSocket socket_ascolto=new DatagramSocket(serverPort);
			byte[] dati=new byte[1000];
			while(true){
				DatagramPacket udp=new DatagramPacket(dati,dati.length);
				socket_ascolto.receive(udp);
				ByteArrayInputStream byteStream=new ByteArrayInputStream(dati);
				ObjectInputStream ois=new ObjectInputStream(byteStream);
				Figlio f=(Figlio)ois.readObject();
				ois.close();
				System.out.println("Il Figlio letto dal pacchetto UDP: "+f.toString());
			}
```

```
                }
                catch(Exception e){
                        ....
                }

        }

}
```

Serializzazione in XML

da completare

Riferimenti

[1] Esercitazione di Sistemi Distribuiti di Leonardo Mariani