JNI: un ponte tra due mondi

DI FRANCESCO SBLENDORIO

Sviluppando in Java capita di aver bisogno di complesse funzionalità, magari già implementate da altri ma in linguaggi diversi: vediamo allora come sfruttare queste componenti preesistenti

Mi è capitato di lavorare ad un progetto in Java in cui, tra le altre cose, si doveva effettuare il *clustering* di tuple di un database. In quell'occasione, ho scoperto l'esistenza in rete di una implementazione che faceva proprio al caso mio: tuttavia, essa non era scritta in Java, bensì in C. Tradurre il tutto in Java non era proponibile (il sorgente, compresso in formato zip, occupava circa 300K), perciò la soluzione migliore è stata quella di compilare quella implementazione in una libreria nativa e di creare un "ponte" tra questa e la JVM. Una cosa del genere è stata resa possibile grazie a JNI (*Java Native Interface*) e in questo articolo vedremo, appunto, come affrontare problemi simili a quello appena esposto.

FIGURA 1 Rappresentazione grafica dell'invocazione di codice nativo (caso con sorgente)

LISTATO 1 Le due funzioni C da richiamare da Java

```
#include <string.h>

/* ordina l'array "inArray" ed inserisce l'array ordinato
   in "outArray". I due array hanno "length" elementi */
void sort(long* inArray, long* outArray, int length);

/* inverte la stringa "inString" e la inserisce in "outString" */
void invertString(char *inString, char *outString);


void sort(long* inArray, long* outArray, int length)
{
        int i,j;
        long temp;

        /* Copia "inArray" in "outArray" */
        for (i=0; i<length; i++)
                outArray[i] = inArray[i];

        for (i=0; i<(length-1); i++)
                for (j=i+1; j<length; j++)
                        if (outArray[i] > outArray[j])
                        {
                                temp = outArray[i];
                                outArray[i] = outArray[j];
                                outArray[j] = temp;
                        }
}

void invertString(char *inString, char *outString)
{
        int i;
        long length = strlen(inString);

        for (i=0; i<length; i++)
                outString[length-i-1] = inString[i];
        outString[length] = '\0';
}
```

CONSIDERAZIONI

Qualcuno, a questo punto, potrebbe già obiettare: "*se includiamo codice nativo, usare Java non ha più senso, perché così si perde la portabilità*". A queste obiezioni rispondo che non è detto che Java venga usato solo per la portabilità. La scelta di Java a livello progettuale può essere dettata anche da altre motivazioni, come la particolare pulizia della sua libreria di classi (si pensi a JDBC o alle Swing) o per la particolare attenzione che Java riserva alle problematiche di sicurezza o di gestione della memoria.

Inoltre non è detto che la portabilità sia così perduta, in quanto nel caso si abbia a disposizione il sorgente (e a patto che sia scritto in C/C++ standard), lo si può sempre ricompilare per qualsiasi sistema operativo target si desideri. Verrà analizzato comunque anche il caso (piuttosto frequente) in cui si abbia a che fare con codice nativo già compilato in una DLL ed il linguaggio usato per la sua costruzione sia totalmente sconosciuto. In questo articolo si assume come piattaforma di riferimento Win32 e come compilatore Microsoft Visual C++ 6. Tutti i concetti esposti sono comunque applicabili a tutti i sistemi operativi che supportino la JVM.

Gli esempi cui faremo riferimento sono disponibili per il download presso il sito ftp://ftp.infomedia.it/pub/DEV./Listati

L'ESEMPIO UTILIZZATO

Il sorgente C del componente da includere in un ipotetico progetto Java è presentato nel **Listato 1**. Si tratta di un programma C che contiene due banali funzioni:

- una funzione, *sort*, di ordinamento su interi lunghi, che ha come parametri un array di input (disordinato), un array di output (che sarà ordinato) e la lunghezza dell'array di input;
- una funzione (*invertString*) che prende in input una stringa (primo parametro) e la restituisce invertita nella variabile indicata nel secondo parametro.

Nei casi reali il codice C è ovviamente più complesso ed è composto da più file, ma il modo di trattare il problema rimane lo stesso. Ciò che interessa è come trattare i dati in input ed in output al codice C. Per questo motivo abbiamo usato array di long e stringhe: questi tipi in Java sono trattati in modo profondamente diverso rispetto al C/C++.

IL PRIMO PASSO

Per questa fase è indifferente il fatto di avere o meno a disposizione il sorgente del codice nativo.

Si consideri una classe Java, come quella del **Listato 2**: si è supposto che la DLL nativa abbia come nome "libreria" (con la "l" minuscola) e che Java richiamerà queste funzioni come metodi statici di una classe chiamata "Libreria" (con la "L" maiuscola) contenuta nel package "provaJni".

La DLL nativa (che deve essere creata successivamente, anche nel caso in cui il codice nativo sia già in una DLL) avrà nome "libreria.dll" e sarà caricata dall'"inizializzatore statico" (la sua strana sintassi indica che esso è eseguito nel momento in cui la classe è caricata in memoria dal ClassLoader e non al momento della istanziazione degli oggetti).

Il percorso della DLL è contenuto nella variabile d'ambiente PATH. Nel caso in cui si voglia specificare il percorso assoluto si può usare, al posto di System.loadLibrary, il metodo:

```
System.load("C:/percorso/della/dll/libreria.dll");
```

Successivamente vengono dichiarate le interfacce Java delle funzioni native, con la seguente sintassi:

```
[public|private|protected]  [static]  native
    <tipo-restituito>  <nome-funzione>  (<lista-parametri>);
```

È molto importante fissare in questo momento le interfacce delle funzioni, poiché questa fase influenzerà le successive.

Si proceda ora alla compilazione della classe. A meno di usare JBuilder o altri ambienti integrati, si imposti - da linea dei comandi - la directory del file ".java" come "corrente" e si lanci il classico

```
javac  Libreria.java
```

per ottenere il file "Libreria.class".

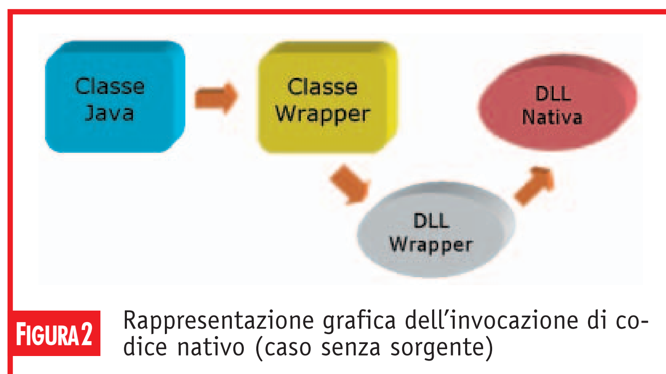

FIGURA 2 Rappresentazione grafica dell'invocazione di codice nativo (caso senza sorgente)

LISTATO 2 La classe-wrapper "Libreria"

```
//  Libreria.java

package  provaJni;

public  class  Libreria  {
   static  {  System.loadLibrary("libreria");  }
   public  static  native  int[]  sortArray(int[]  vet);
   public  static  native  String  invertString(String  s);
}
```

GENERAZIONE DELLE INTERFACCE C/C++

Anche per questo passo è indifferente avere o meno il sorgente del codice nativo. Da linea dei comandi, si entri nella directory che contiene la sotto-directory "provaJni" e si lanci:

```
javah  -jni  -classpath  .  provaJni.Libreria
```

(particolarmente importante è il parametro classpath, senza il quale la classe non sarebbe individuata).

Nella directory corrente viene generato un file, di nome "provaJni_Libreria.h" (dovrà essere copiato ed incluso nel progetto Visual C++ che creeremo tra poco). Aprendolo con un editor di testo, si scopre che contiene le interfacce di due strane funzioni C/C++ (che si vedrà a breve come implementare), rispettivamente:

```
JNIEXPORT  jintArray  JNICALL
                          Java_provaJni_Libreria_sortArray
        (JNIEnv  *,  jclass,  jintArray);
```

```
JNIEXPORT  jstring  JNICALL
                       Java_provaJni_Libreria_invertString
        (JNIEnv  *,  jclass,  jstring);
```

Ad un esame più attento si osserva che queste interfacce ricalcano, a parte qualche aggiunta, quelle delle funzioni indicate in Libreria.java; in particolare notiamo che:

1. i nomi delle funzioni sono gli stessi (sortArry e invertString), ma cominciano con Java_provaJni_Libreria_
2. entrambe le funzioni hanno, come primi due, parametri di tipo JNIEnv* e jclass
3. i tipi di ritorno e di input sono degli equivalenti Java in C/C++:
   – int[] (Java) corrisponde a jintArray (C/C++)
   – String (Java) corrisponde a jstring (C/C++)

Il punto 1 si spiega col fatto che JNI usa una convenzione, chiamata *name mangling*, che impone di far iniziare i nomi delle funzioni C/C++ con il prefisso "Java" seguito dal nome del package (se presente), da quello della classe ed infine da quello del metodo; questi token devono essere separati da un *underscore* (_). Se si ha *overloading*, il nome sarà completato con la *signature* della funzione (indicata nelle righe di commento). Relativamente al punto 2, il parametro di tipo JNIEnv* è un puntatore all'ambiente della JVM e serve a scambiare dati (oggetti, metodi) con essa, tramite le funzioni JNI. Il parametro di tipo jclass è un puntatore alla *classe* cui appartiene la funzione. Se il metodo Java *non* fosse stato statico, il secondo parametro sarebbe stato di tipo jobject, puntatore all'oggetto corrente (in pratica, corrisponde a *this*).

Infine, relativamente al punto 3, i tipi Java sono, come già detto, profondamente diversi da quelli del C/C++, quindi bisogna creare una sorta di "equivalenza" (dipendente ovviamente dalla piattaforma): le variabili di questi tipi, defi-

LISTATO 3 Implementazione delle funzioni-wrapper in C

```
  1: #include <string.h>
  2: #include <stdlib.h>
  3: #include "provaJni_Libreria.h"
  4:
  5: /*********************************
  6: ** Qui va incluso il listato 1 **
  7: *********************************/
  8:
  9:
 10: /* implementazione delle funzioni
 11: che verranno richiamate dalla JVM */
 12:
 13: JNIEXPORT jintArray JNICALL
             Java_provaJni_Libreria_sortArray
 14: (JNIEnv *env, jclass jc, jintArray inArray)
 15: {
 16: int nelem, i;
 17:
 18: /* qui ci sarà l'array in input
                           (array di jint) */
 19: jint* jInArray;
 20:
 21: /* qui ci sarà l'array in input
        con gli elementi convertiti in long */
 22: long* cInArray;
 23:
 24: /* qui ci sarà l'array in output
                      (con elementi long) */
 25: long* cOutArray;
 26:
 27: /* qui ci sarà l'array in output
                      (con elementi jint) */
 28: jint* jOutArray;
 29:
 30: /* qui ci sarà il risultato da restituire
                      alla JVM (IntArray) */
 31: jintArray jResult;
 32:
 33: nelem = (*env)->GetArrayLength(env, inArray);
 34: jInArray = (*env)->GetIntArrayElements
                       (env, inArray, NULL);
 35: jOutArray = (long*) malloc(nelem*sizeof(long));
 36: cInArray = (long*) malloc(nelem*sizeof(long));
 37: cOutArray = (long*) malloc(nelem*sizeof(long));
 38:
 39: /* copia gli elementi di tipo "jint"
                     nell'array di "long" */
 40: for (i=0; i<nelem; i++)
 41:    cInArray[i] = (long) jInArray[i];
 42:
 43: /**** esegue la funzione sort ****/
 44: sort(cInArray, cOutArray, nelem);
 45: /*********************************/
 46:
 47: /* copia il risultato nell'array di "jint" */
 48: for (i=0; i<nelem; i++)
 49: jOutArray[i] = (jint) cOutArray[i];
 50:
 51: /* copia l'array di "jint" in un array java */
 52: jResult = (*env)->NewIntArray(env, nelem);
 53: (*env)->SetIntArrayRegion(env, jResult, 0,
                              nelem, jOutArray);
 54:
 55: /* libera la memoria usata dal C */
 56: free(cInArray);
 57: free(cOutArray);
 58: free(jOutArray);
 59:
 60: /* libera la memoria occupata dal parametro */
 61: (*env)->ReleaseIntArrayElements
                 (env, inArray, jInArray, 0);
 62:
 63: /* ritorna il risultato alla JVM */
 64: return jResult;
 65: }
 66:
 67:
 68: JNIEXPORT jstring JNICALL
          Java_provaJni_Libreria_invertString
 69:    (JNIEnv *env, jclass jc, jstring inString)
 70: {
 71: int stringLength;
 72:
 73: /* qui sarà contenuta la stringa come array
                           di caratteri ASCII */
 74: char* cInString;
 75:
 76: /* qui sarà contenuta la stringa invertita
                    (array di caratteri ASCII) */
 77: char* cOutString;
 78:
 79: /* qui sarà contenuta la stringa Java
                           restituita alla JVM */
 80: jstring jResult;
 81:
 82: stringLength =
        (*env)->GetStringLength(env, inString);
 83: cInString = (char*) (*env)->GetStringUTFChars
                           (env, inString, NULL);
 84: cOutString = (char*)
           malloc(1 + stringLength*sizeof(char));
 85:
 86: /**** esegue la funzione invertString ****/
 87: invertString(cInString,cOutString);
 88: /****************************************/
 89:
 90: jResult =
           (*env)->NewStringUTF(env, cOutString);
 91:
 92: /* libera la memoria allocata dal C */
 93: free(cOutString);
 94:
 95: /* libera la memoria occupata dal parametro */
 96: (*env)->ReleaseStringUTFChars
                   (env, inString, cInString);
 97:
 98: /* ritorna il risultato alla JVM */
 99: return jResult;
100: }
```

niti da JNI, vengono manipolate tramite le funzioni JNI. Non si è ancora detto cosa siano in effetti le due funzioni appena viste: sono quelle effettivamente richiamate quando vengono invocati i metodi della classe "Libreria" (sono funzioni-wrapper). Pertanto dobbiamo implementarle, facendo in modo che richiamino opportunamente le due funzioni viste nel **Listato 1** (*sort* e *invertString*).

INTERFACCIAMENTO TRA C/C++ E JAVA

Il prossimo passo dipende dal fatto di avere o meno i sorgenti del codice nativo. Prendiamo in considerazione il caso in cui si abbiano i sorgenti a disposizione. Possono verificarsi ulteriori due sotto-casi: i sorgenti possono essere scritti in C o in C++. Segue un'analisi del caso in cui il linguaggio utilizzato sia C. Si apra l'IDE del Visual C++ e si selezioni "File / New… / Projects / Win32 Dynamic-Link Library" (chiamando il progetto "libreria") e poi "An empty DLL project". In questo progetto si aggiunga un file ".c" in cui va riportato il **Listato 3** (inserendovi il **Listato 1** ove indicato); infine si compili (Build / Build libreria.dll) e si provveda a copiare il file "libreria.dll" appena creato in una directory appartenete al PATH, tipicamente \WINDOWS\System.

Prima di procedere con i commenti su tale listato, ecco un piccolo suggerimento per evitare tanti grattacapi: dalla directory "C:\JDK\include" (supponendo che il JDK si trovi in "C:\JDK") e dalle sue sotto-directory si copino (e si includano) tutti i file ".h" nella directory del progetto Visual C++. Si apra ogni file, cercando le direttive #include che fanno riferi-

mento a questi stessi file, modificandole in modo che il nome del file header sia racchiuso tra virgolette anziché tra '<' e '>'. Il **Listato 3** mostra l'implementazione C delle funzioni viste nel paragrafo precedente. La prima cosa che può venire in mente è: "*per essere dei semplici wrapper, sembrano piuttosto lunghi!*". In realtà basta suddividere in zone il codice e si vede che la struttura di ogni funzione è la seguente:

- conversione dei parametri Java in tipi C (linee 18-41 nella prima funzione e linee 73-84 nella seconda funzione)
- elaborazione dei dati, tramite chiamate a codice C (linea 44 nella prima funzione e linea 87 nella seconda funzione)
- conversione dei risultati C in tipi Java (linee 47-64 nella prima funzione e linee 90-99 nella seconda funzione)

Come si vede, l'operazione più complessa è quella di conversione da tipi Java a tipi C e viceversa. I tipi Java infatti non sono trattabili direttamente da C. L'accesso alle variabili di questi tipi è regolato tramite le *funzioni JNI*. Basta aprire con un editor di testo il file "jni.h" per avere un'idea di quante e quali siano. Esse consentono di manipolare ogni tipo di oggetto e variabile Java; in particolare ci si soffermerà sul trattamento di stringhe e di array di tipi semplici, dato che questo è il modo più semplice di interfacciamento tra funzioni. Le funzioni JNI sono tante e consentono anche di creare, manipolare e restituire oggetti di qualunque classe Java. Per approfondimenti si vedano **[1]** e **[2]**.

INVOCAZIONE DELLA FUNZIONE "SORT"

Segue un'analisi della funzione-wrapper che richiama "sort": essa riceve in input un array Java (che JNI traduce nell'equivalente C "jintArray") il cui contenuto deve essere "riversato" in un classico array C per poterlo utilizzare nella funzione "sort" (richiamata alla linea 44). Si noti la dichiarazione di ben quattro puntatori (che sono intesi come array):

- jInArray: contiene gli elementi del "jintArray" in input sotto forma di un array C i cui elementi sono di tipo "jint".
- cInArray: contiene una copia degli elementi di "jInArray" convertiti in "long". Studieremo più avanti questo aspetto.

LISTATO 4 La classe Java di test per la classe "Libreria"

```
// MainClass.java

package provaJni;

public class MainClass {
   public static void main(String[] args) {
      int[] inVect =
                {98,23,50,20,15,14,10,5,4,3,2,1,0};
      int[] outVect;
      String str = "Stringa da invertire";

      outVect = Libreria.sortArray(inVect);

      System.out.println("Vettore originale");
      for (int i=0; i<inVect.length; i++)
            System.out.print(inVect[i]+" ");
      System.out.println("\n");

      System.out.println("Vettore ordinato");
      for (int i=0; i<inVect.length; i++)
            System.out.print(outVect[i]+" ");
      System.out.println("\n");

      System.out.println("Stringa originale:
                              \""+str+"\"");
      System.out.println("Stringa invertita:
                \""+Libreria.invertString(str)+"\"");
   }
}
```

- cOutArray: conterrà il risultato della funzione "sort" (un array con elementi "long").
- jOutArray: conterrà una copia degli elementi di "cOutArray" convertiti in "jint".

Perché questa "doppia conversione"? Il motivo è che la funzione "sort" richiede un array di long; a quale tipo equivalga "jint" dipende dal sistema operativo e dal compilatore, quindi bisogna effettuare una conversione. Discorso analogo per il ritorno dei valori: cOutArray contiene elementi long, convertiti in "jint" nell'array jOutArray (linee 48..49); successivamente (tramite una funzione JNI) si converte tutto l'array in jintArray (linee 52..53), in seguito restituito alla JVM. Si esaminino queste conversioni un po' più in dettaglio: per prima cosa viene calcolato il numero degli elementi dell'array Java (linea 33):

```
nelem = (*env)->GetArrayLength(env, inArray);
```

Notiamo una strana sintassi: l'ambiente della JVM (la variabile "env") viene referenziato due volte. Si tenga presente che la riga indicata è codice C e non C++: "*env" è un puntatore a *struct* e non a *class*, cioè "GetArrayLength" non è un metodo bensì un semplice puntatore a funzione facente parte della *struct*. Quindi l'ambiente JVM "env" va passato alla funzione puntata. Ogni funzione JNI, richiamata da C, ha come primo parametro "env". Il secondo parametro è l'array Java di cui si vuole la lunghezza. Subito dopo un'altra chiamata a funzione JNI provvede ad allocare un array C di elementi jint, copiandovi il contenuto dell'array Java (linea 34):

```
jInArray = (*env)->GetIntArrayElements(env, inArray, NULL);
```

Il terzo parametro, un puntatore a *jboolean*, ha il seguente significato: se non è NULL, la variabile puntata è settata a JNI_TRUE se viene effettuata una copia dell'array; a JNI_FALSE altrimenti. Nel caso considerato ciò non interessa e si può tranquillamente porre a NULL. Si noti, verso le ultime righe della funzione, la deallocazione di questo array, necessaria (salvo pericolosi *memory leaks*) e rigorosamente a carico del programmatore (linea 61: l'ultimo parametro posto a zero indica di salvare le eventuali modifiche sull'array e di deallocare la memoria prima occupata):

```
(*env)->ReleaseIntArrayElements(env, inArray, jInArray, 0)
```

Dopo aver copiato l'array Java (linea 34), viene allocata memoria (opportunamente liberata alle linee 56..58) per contenere gli array cInArray, cOutArray, jOutArray (linee 35..37) di cui si è discusso sopra; array che vengono riempiti (con eventuali conversioni di tipo tramite *cast* esplicito) con opportuni cicli prima (linee 40..41) e dopo (linee 48..49) l'esecuzione di "sort" (linea 44), ovvero la funzione che esegue l'elaborazione vera e propria. Come ultima operazione viene creato un array Java di *nelem* elementi (linea 52), riempito (linea 53) con gli elementi di jOutArray, a partire da quello di indice 0 e per una lunghezza di *nelem*. Questo array Java viene infine restituito dalla funzione alla JVM (linea 64). Esso non deve essere deallocato, in quanto soggetto a *garbage collection*.

INVOCAZIONE DELLA FUNZIONE "INVERTSTRING"

L'interfacciamento con la funzione "invertString" è del tutto analogo a quello di "sort", salvo le funzioni JNI per la gestione delle stringhe; una delle differenze tra stringhe C/C++ e stringhe Java è la codifica usata per i caratteri: C/C++ usa la codifica ASCII (8 bit) mentre Java usa Unicode (a 16 bit). Segue in dettaglio la descrizione delle sole parti che differi-

LISTATO 5 Una DLL che ne carica un'altra di cui non si ha il codice sorgente

```
1: // libreria.cpp
2:
3: #include "stdafx.h"
4:
5: // #include necessarie per l'implementazione
                    delle funzioni Java_....
6: #include "provaJni_Libreria.h"
7: #include <string.h>
8: #include <stdlib.h>
9:
10: // Definizione delle funzioni da importare
11: HINSTANCE hDLL;
12: typedef void (*SORT)(long*, long*, int);
13: typedef void (*INVERTSTRING)(char*, char*);
14: SORT sort;
15: INVERTSTRING invertString;
16:
17: // Carica la DLL che contiene le funzioni da
                           richiamare da Java
18: // ed inizializza i puntatori a funzione
19: BOOL APIENTRY DllMain( HANDLE hModule,
20:                        DWORD ul_reason_for_call,
21:                        LPVOID lpReserved
22:              )
23: {
24: hDLL = LoadLibrary("utility.dll");
25: sort         = (SORT) GetProcAddress
                              (hDLL,"sort");
26: invertString = (INVERTSTRING) GetProcAddress
                      (hDLL,"invertString");
27: return TRUE;
28: }
29:
30: // implementazione delle funzioni che
       verranno richiamate dalla JVM (Java_...)
31: JNIEXPORT jintArray JNICALL
                Java_provaJni_Libreria_sortArray
32: (JNIEnv *env, jclass jc, jintArray inArray)
33: {
34: int nelem, i;
35:
36: /* qui ci sarà l'array in input
                           (array di jint) */
37: jint* jInArray;
38:
39: /* qui ci sarà l'array in input
       con gli elementi convertiti in long */
40: long* cInArray;
41:
42: /* qui ci sarà l'array in output
                      (con elementi long) */
43: long* cOutArray;
44:
45: /* qui ci sarà l'array in output
                      (con elementi jint) */
46: jint* jOutArray;
47:
48: /* qui ci sarà il risultato da restituire
                      alla JVM (IntArray) */
49: jintArray jResult;
50:
51: nelem = env->GetArrayLength(inArray);
52: jInArray =
       env->GetIntArrayElements(inArray, NULL);
53: jOutArray = (long*) malloc(nelem*sizeof(long));
54: cInArray = (long*) malloc(nelem*sizeof(long));
55: cOutArray = (long*) malloc(nelem*sizeof(long));
56:
57: /* copia gli elementi di tipo "jint"
                     nell'array di "long" */
58: for (i=0; i<nelem; i++)
59:    cInArray[i] = (long) jInArray[i];
60:
61: /**** esegue la funzione sort ****/
62: sort(cInArray, cOutArray, nelem);
63: /*********************************/
64:
65: /* copia il risultato nell'array di "jint" */
66: for (i=0; i<nelem; i++)
67:    jOutArray[i] = (jint) cOutArray[i];
68:
69: /* copia l'array di "jint" in un array java */
70: jResult = env->NewIntArray(nelem);
71: env->SetIntArrayRegion(jResult, 0,
                                nelem, jOutArray);
72:
73: /* libera la memoria usata dal C */
74: free(cInArray);
75: free(cOutArray);
76: free(jOutArray);
77:
78: /* libera la memoria occupata dal parametro */
79: env->ReleaseIntArrayElements(inArray,
                                    jInArray, 0);
80:
81: /* ritorna il risultato alla JVM */
82: return jResult;
83: }
84:
85:
86: JNIEXPORT jstring JNICALL
            Java_provaJni_Libreria_invertString
87:     (JNIEnv *env, jclass jc, jstring inString)
88: {
89: int stringLength;
90:
91: /* qui sarà contenuta la stringa
                come array di caratteri ASCII */
92: char* cInString;
93:
94: /* qui sarà contenuta la stringa invertita
                  (array di caratteri ASCII) */
95: char* cOutString;
96:
97: /* qui sarà contenuta la stringa Java
                        restituita alla JVM */
98: jstring jResult;
99:
100: stringLength = env->GetStringLength(inString);
101: cInString = (char*)
          env->GetStringUTFChars(inString, NULL);
102: cOutString = (char*)
          malloc(1 + stringLength*sizeof(char));
103:
104: /**** esegue la funzione invertString ****/
105: invertString(cInString,cOutString);
106: /*****************************************/
107:
108: jResult = env->NewStringUTF(cOutString);
109:
110: /* libera la memoria allocata dal C */
111: free(cOutString);
112:
113: /* libera la memoria occupata dal parametro */
114: env->ReleaseStringUTFChars(inString,
                                   cInString);
115:
116: /* ritorna il risultato alla JVM */
117: return jResult;
118: }
```

scono rispetto alla funzione precedente. Si noti che non sono necessari cicli per convertire i caratteri a 16 bit nei caratteri a 8 bit, in quanto (linea 83) la funzione JNI *GetStringUTFChars* provvede da sola a convertire in ASCII la stringa Unicode di Java e ad inserirla in un array di caratteri:

```
cInString = (char*)
        (*env)->GetStringUTFChars(env, inString, NULL);
```

Il significato di NULL è quello spiegato nel paragrafo precedente circa la linea 34. Da notare la necessaria deallocazione da parte del programmatore, alla linea 96:

```
(*env)->ReleaseStringUTFChars(env, inString, cInString);
```

Da notare anche, riguardo cOutString, il numero di byte allocati (lunghezza della stringa + 1) che tiene conto dello '\0' di fine stringa. Dopo la chiamata della funzione C "invertString" viene allocata ed inizializzata una stringa Java a partire dalla stringa C "cOutString" (linea 90):

```
jResult = (*env)->NewStringUTF(env, cOutString);
```

Come per l'array restituito dalla funzione precedente, anche questa stringa (restituita dalla funzione alla JVM: linea 99) è soggetta a *garbage collection*.

RICHIAMO DELLA DLL DA JAVA

All'interno dello stesso package (provaJni) si inserisca un'altra classe, MainClass (**Listato 4**). Come si vede, le funzioni (statiche) della classe Libreria sono utilizzate come fossero normali metodi scritti in Java. Si compili (con il comando "javac *" nella directory in cui si trovano i file .java) e si esegua MainClass (posizionandosi nella directory dove si trova il package provaJni):

```
java –classpath . provaJni.MainClass
```

Tutto funziona perfettamente.

L'invocazione delle funzioni native (caso del sorgente a disposizione) è rappresentata graficamente in **Figura 1**.

CODICE NATIVO IN C++

Nel caso in cui il codice nativo sia scritto in C++ e faccia uso di costrutti di tale linguaggio, il progetto DLL che si crea in Visual C++ contiene file con estensione ".cpp": in tal caso la variabile "env" non è un puntatore a *struct*, ma ad un oggetto, perciò le uniche modifiche da fare al **Listato 3** sono le chiamate alle funzioni JNI. La generica chiamata

```
(*env)->FunzioneJNI(env, parametro1, parametro2)
```

diventa

```
env->FunzioneJNI(parametro1, parametro2)
```

Ad esempio

```
(*env)->ReleaseStringUTFChars(env, inString, cInString)
```

diventa

```
env->ReleaseStringUTFChars(inString, cInString)
```

Questo è il caso che si presenta nel paragrafo seguente.

CODICE NATIVO SENZA SORGENTI

Si esamina ora il caso in cui il codice nativo da richiamare da Java non sia corredato da sorgenti; si supponga quindi di avere a disposizione la sola DLL (compilata a partire da codice C, C++, Delphi o qualsiasi altro linguaggio) insieme ad un documento che specifichi le interfacce delle funzioni ivi contenute. Nei listati scaricabili dal sito ftp://ftp.infomedia.it/DEV./Listati c'è una directory chiamata "senza_sorgente" che contiene due progetti Visual C++:

- "utility" che contiene la DLL che si suppone senza sorgenti (contenente le funzioni "sort" ed "invertString"). Naturalmente sul sito ci sono solo i sorgenti di questo progetto: basta compilarli e copiare la DLL ("utility.dll") ottenuta in \WINDOWS\System; dopodiché questi sorgenti possono essere totalmente ignorati.
- "libreria" che, nelle funzioni-wrapper indicate da JNI, richiama "sort" ed "invertString" contenute in "utility.dll"

Per creare la DLL "libreria" che richiama le funzioni di quella nativa, si crei un nuovo progetto DLL da Visual C++ e, al momento di scegliere tra "Empty DLL / Simple DLL Project/ etc..", si opti per "Simple DLL Project". Sarà creata una DLL con una sola funzione, DLLMain(), richiamata al caricamento della DLL in memoria: al suo interno bisogna caricare la DLL nativa di cui non si ha il sorgente (in questo caso "utility.dll"). Il file che l'IDE ha creato automaticamente ha estensione ".cpp", pertanto bisogna utilizzare la sintassi C++ nel richiamare le funzioni JNI.

Nel **Listato 5** si può leggere il sorgente da creare.

Nelle linee 11..15 c'è la definizione di hDLL, l'handle della DLL da caricare ("utility.dll"), seguita dalla definizione di due tipi, entrambi puntatori a funzioni le cui interfacce sono quelle di "sort" e "invertString" (linee 12..13); sono poi istanziate due variabili dei due tipi appena illustrati: esse punteranno proprio a "sort" e "invertString".

Nelle linee 19..28 c'è la funzione DllMain, che si occupa di caricare "utility.dll" (linea 24) e di assegnare ai puntatori a funzione, gli indirizzi effettivi delle funzioni desiderate (linee 25..26) tramite "GetProcAddress".

Il resto del codice è identico al **Listato 3** salvo per la sintassi delle chiamate alle funzioni JNI, che stavolta seguono le regole del C++. A questo punto basta avere in \WINDOWS\System entrambe le DLL "utility.dll" e "libreria.dll" ed il programma Java funzionerà correttamente. La **Figura 2** illustra graficamente l'invocazione di metodi nativi di cui non si dispone del sorgente.

CONCLUSIONI

JNI, se usato in modo appropriato, è uno strumento potentissimo a disposizione degli sviluppatori Java: tutta la sua utilità si rivela quando si devono integrare porzioni di codice legacy di vecchie applicazioni in nuovi progetti, in quanto evita la completa riscrittura di codice già utilizzato ed il cui uso è ben acquisito dall'organizzazione.

BIBLIOGRAFIA

[1] Bruce Eckel, "*Thinking in Java – 2nd edition*", http://www.bruceeckel.com

[2] Sito ufficiale SUN su Java, http://java.sun.com

Francesco Sblendorio *è laureato in Informatica presso l'Università degli Studi di Bari. Si occupa di programmazione in C/C++, Java, Visual Basic e di problematiche legate al Web. Può essere contattato tramite e-mail all'indirizzo sblendorio@infomedia.it*

DEV • DEV