C/C++ Library für Java
by
TgZero
03.03.2013

Inhaltsverzeichnis

Intro

Okey liebe Coding-Freunde, heute dreht es sich mal ein wenig um C/C++ UND Java.
Ja ihr habt richtig gelesen Java.... ^^

In diesem kleinen Ausschnitt werde ich euch zeigen, wie ihr eine Library in C++ schreibt, und diese in eurem Java-Programm einbettet. *Dabei gehe ich nur auf JNI und nicht auf JNA ein!* Doch warum sollte man so etwas tun, immerhin ist Java plattformunabhängig und "mächtig genug", um C++ bezüglich Anwendungsentwicklung abzulösen.
Der Grund ist zum einen, Performance! Die Libs liegen immernoch in Binärcode und nicht im Byte-Code vor, das heißt, dass der Java-Interpreter diesen nicht erst umwandeln, sondern direkt ausführen kann.

Das Problem bei den vorhandenen Libs

Wie viele wohl schon wissen, benutzt Java sogenannte Referenzdatentypen für alle Datentypen (Klassen), außer eben den Basisdatentypen. (Integer Referenzdatentyp, int Basisdatentyp).
C bzw. C++ unterscheidet zwischen Zeigern, Referenzen und "normalen" Variablen (sprich Call-By-Pointer, Call-By-Reference, Call-By-Value). Damit wir also unsere Library in einem Java-Programm implementieren können, müssen bestimmte bedingungen erfüllt sein (das ist auch der Grund, warum man nicht einfach jede x-beliebige Library einbinden kann).
Hierbei bietet Java Basisdatentypen, welche für C zugeschnitten sind, an (*jstring, jclass, JNIEnv, jint...*)

Die Bedingungen für eine Library

Da die JVM fürs Laden der Bibliothek zuständig ist und für das einbinden der Methoden, müssen bestimmte Signaturen für die Header-Datei erfüllt sein. Und für genau diesen Zweck wird uns das Programm "javah" mitgeliefert, welches aus einer Klassenimplementation von Java, die eine Library einbindet, eine Header-Datei erstellt. Anschließend müssen wir uns nurnoch um eine Implementierung kümmern.

Eine Java-Klasse erstellen

Per *System.loadLibrary(Klassenname)* laden wir eine Library mit dem Namen "Klassenname". Diese Funktion kann dann in einer Java-Klasse (z.b. Plugin) im Konstruktor aufgerufen werden. Somit bindet man die Library an die Lebenszeit des Klassenobjekts. Wird also die Klasse von der GC gelöscht, so wird auch die Library aus dem virtuellem Speicher des Prozesses geladen. Natürlich könnte man den aufruf von loadLibrary statisch machen, somit wäre die Library von Anfang an bereits im Speicher. Da ich dieses Tutorial mit dem Ziel schreibe, dass der Leser Plugins entwickeln kann, wird hier nicht weiter drauf eingegangen. Um also Java mitzuteilen, dass Funktionen dieser Methode aus der Library kommen, müssen diese als "native" deklariert und nicht implementiert werden. Desweiteren sollte man weitesgehend auf Objekte verzichten und nur die Standarddatentypen verwenden.

Eine mögliche Plugin-Klasse könnte zum Beispiel so aussehen:

```
public class Plugin {
   public Plugin(String Libname)
   {
      System.loadLibrary(Libname);
   }

   private int MeinInt;
   public void PrintMembers()
   {
      System.out.println(this.MeinInt);
   }
   public native void          SetMeinInt(int value);
   public native String        GetPluginName();
}
```

Hierbei werden die Funktionen "SetMeinInt()" und "GetPluginName()" aus der Library "Libname" implementiert. Wenn wir also in unserem Java-Programm ein Objekt erstellen und von diesem dann die Funktionen aufrufen, wird die Funktion in der Library ausgeführt.

Zugreifen auf Objekte, Felder, Methoden und Funktionen von C / C++

Als erstes muss der Ordner der die Datei "jni.h" beinhaltet zu den Include-Verzeichnissen des C/C++-Projekts hinzugefügt werden und in unserer Header-Datei eingebunden werden (dies sollte allerdings "javah" schon für uns erledigt haben.

Wie schon kurz bei "Das Problem bei den vorhandenen Libs" angesprochen, liefert Java uns spezielle Basisdatentypen. Eine C-Funktion, die in einem Java-Programm implementiert werden soll bekommt als ersten Parameter immer einen Zeiger auf die JNI-Umgebung. Mit diesem Zeiger, könnt ihr neue Java-Klassen erstellen, oder statische Funktionen aufrufen.
jclass ist ebenfalls wie *JNIEnv* ein Zeiger und zeigt auf die "gefundene" Klasse. Bei einem Funktionsaufruf, zeigt das Objekt auf die eigentliche Klasse ("this"!) , andernfalls kann man diesen Zeiger auch auf ein vorhandenes Objekt zeigen lassen. Funktionen wie *GetFieldID, GetMethodID...* brauchen bei ihrem Aufruf als Parameter einen Datentyp als C-String kodiert.

C-String	Datentyp
Z	Boolean
B	Byte
C	Char
S	Short
I	Int
J	Long
F	Float
D	Double
V	Void

In C++ müssen wir Variablen,Felder und Funktionen von Klassen immer per Ids ansprechen. Dazu müssen wir zunächst die besagte Klasse finden, anschließend die Id der gewünschten Variable/Funktion und diese anschließend mit einer weiteren Funktion von JNIEnv aufrufen. (etwas kompliziert oder? Das habe ich mir auch gedacht und als Übung dieses Tutorial verfasst :D)

Auf Felder der Klasse zugreifen

Da wir ja mit Objekten arbeiten, und eine Klassenfunktion wohl auch auf dessen Felder zugreifen darf, haben wir einen weiteren Datentypen bekommen *jfieldID*. Dank diesem Datentyp und der Funktion *GetFieldID* können wir beliebige Felder aufrufen, wobei *GetFieldID* die folgenden Parameter benötigt:
jclass class, char* Feldname, char* Datentyp.
Aufgerufen wird das ganze dann so
jfieldID meineId = env->GetFieldID(meineKlasse,"IntFeld","I");
Doch woher bekommt man genau den Zeiger auf die richtige Klasse?
Wenn Klassenfunktionen als native deklariert wurden, so bekommen sie automatisch als zweiten Parameter einen Zeiger auf die eigene Klasse. Somit ersparen wir uns die große Suche ;-)
Anschließend können wir mit *Get---Field()* den Wert des Feldes auslesen. Hierbei steht "---" für einen beliebigen Datentypen.

Funktionsname	N-Datentyp	J-Datentyp
GetObjectField()	jobject	Object
GetBooleanField()	jboolean	boolean
GetByteField()	jbyte	byte
GetCharField()	jchar	char
GetShortField()	jshort	short

GetIntField()	jint	int
GetLongField()	jlong	long
GetFloatField()	jfloat	float
GetDoubleField()	jdouble	double

jint = env->GetIntField(obj,MeineId);

Das setzen von Feldern geschieht ähnlich wie das auslesen.
Wir holen uns zunächst die Klasse an sich von dem übergebenen *jobject,* anschließend suchen wir nach der ID und setzen nun die Variable mit *Set—Field()*
Eine Funktion namens SetMeinInt, welches von der eigenen Klasse den Wert der int-Variable "MeinInt" auf "value" setzt könnte nun so aussehen:

```
JNIEXPORT void JNICALL Java_Plugin_SetMeinInt(JNIEnv * env, jobject obj, jint value)
{
	jclass		MeineKlasse;
	jfieldID	MeinIntId;

	MeineKlasse =	env->GetObjectClass(obj);
	MeinIntId =	env->GetFieldID(MeineKlasse,"MeinInt","I");
	env->SetIntField(obj,MeinIntId,value);
}
```

Methoden und Funktionen

Zunächst brauchen wir eine Id. Bei den Feldern war dieses *jfieldID* und bei den Methoden heißt es nun *jmethodID.* Also wieder eine Instanz davon erstellt und per "GetMethodID" die Methode finden. Hierbei sind jedoch die Parameter etwas anders. Zunächst kommt ein Zeiger auf unserer Klasse, dann ein C-String mit dem Namen der Funktion, erneut ein C-String mit dem Rückgabewert, allerdings als String kodiert (z.B. "()Ljava/lang/String") und nun die Datentypen der Parameter (ebenfalls als String kodiert!).
Falls keine Methode/Funktion mit dieser Signatur existiert, wird uns 0 zurückgegeben.
Und jetzt wird es wieder etwas tricky... wir unterscheiden zunächst zwischen statischen und Klassenfunktionen. Hierbei gibt es CallStaticMethod() und CallObjectMethod().
Da wir aber unterschiedliche Rückgabewerte haben, Basisdatentypen oder sogar Objekte, müssen wir wieder eine kleine Unterscheidung machen. Anstatt das wir also CallObjectMethod() für eine Funktion benutzen, die uns einen booleschen Wert zurückgibt, rufen wir diese mit *env->CallBooleanMethod()* auf. Trotz dieser Unterscheidung fehlen uns aber noch die Prozeduren. Hierbei rufen wir die Funktion einfach mit CallVoidMethod() auf.
Es sei noch gesagt, dass man Objekte, also Felder von Nicht-primitiven Datentypen in einem jobject speichert.
Doch wie sieht es jetzt mit den Parametern der Methoden aus?
Die Parameter werden entweder mit einem jvalue-Zeiger auf einem jvalue-Array übergeben, oder per va_list. Um zu entscheiden, welche der beiden Möglichkeiten benutzt werden soll, hängen wir einfach an unsere Call-Funktion ein "A" für jvalue oder ein "V" für va_list an.

Ein Aufruf für eine Funktion mit booleschem Rückgabewert könnte nun so aussehen:
env->CallBooleanMethodA(class, methodID, args);

Was es noch zu beachten gibt

Zum einen arbeitet man normalerweise in C++ C-Strings, also mit chars, während man in Java standardmäßig mit Unicode und somit mit wide-chars arbeitet. Um trotzdem mit Strings zwischen Java und C++ zu arbeiten, bietet uns die JNIEnv ein paar Funktionen dafür.

NewString, GetStringLength,ReleaseStringChars,NewStringUTF, GetStringUTFLength, GetStringUTFChars, ReleaseStringUTFChars

Ich habe schon zu beginn gesagt, dass man (sofern möglich) auf Klassenobjekte soweit es geht verzichten sollte. Ich habe absichtlich auf Code in dieser Richtung verzichtet, wer sich aber durch mich ermutigt fühlt, diesen anzufertigen kann diesen gerne innerhalb des Threads posten, oder mir per Mail schicken. Ansonsten lege ich euch die Header-Datei "jni.h" ans Herz, die so oder so eingebunden werden muss.
Ich habe ja mehrfach auf das Programm javah verwiesen.
Angenommen wir hätten die Plugin-Klasse bei "Eine Java-Klasse erstellen" in der Datei Plugin.class, so können wir mit
"javah -jni -o Plugin.h Plugin" daraus einer Headerdatei erstellen. Somit spart man es sich, die Signaturen anzupassen. Anschließend nurnoch in Visual Studio oder einer beliebigen IDE unter Linux eine Shared-Library (oder DLL) erstellen und Plugin.h in das Projekt mit einbinden. Dann noch eine Plugin.cpp-Datei erstellen und dort die Implementierungen der Funktionen vornehmen. Fertig ;-)

Ein paar letzte Worte

Ich hoffe euch hat dieses Tutorial gefallen. Zum Schluss wurde es etwas kürzer, dennoch hoffe ich, dass alles verstanden wurde. Falls nicht, entweder mich fragen, oder eine Suchmaschine anschmeißen. Der Anlass für dieses Tutorial war im übrigen das Community-Projekt "SECSuite" vom Coding-Board "coderz.cc". Natürlich nur für den Fall,dass ihr Lust bekommen habt fleißig mitzuwirken. Zum Schluss noch ein paar Grüße an: gehaxelt, Easysurfer, cr4ck3r, Rasputin und Zer0Flag (denn die Zer0s müssen zusammen halten!)

mfg
euer TgZero