Java Native Interface

C für Java

Marco Kraus <kraus@tfh-berlin.de>

Was ist JNI ?

- Ein Programmier-Interface im Sun JDK

- Ermöglicht u.a. C Code in Java

- Ermöglicht Aufruf von Java in den nativen Methoden

Warum C-Code in Java ?

- Performance

- Programme / Bibliotheken in C vorhanden

- Plattformspezifische Funktionen (z.B. Dateiberechtigungen)

Die offizielle Java-Dokumentation über JNI:

> [...] It is recommended that only experienced programmers should attempt to write native methods or use the Invocation API!

Intermezzo 0:

Erstellen eines C-Programmes:

- Programmcode schreiben

- Programm compilieren => Objectfile

- Linken : Binden der Objectfiles, Referenzerzeugung
  (=> Executable, z.B. a.out, ELF, win32 exe, ...)

Intermezzo I:

Definition und Deklaration von Funktionen in C

```
#include <stdio.h>

void PrintHello(){
    printf("HelloWorld\n");
}

int main(){
    PrintHello();
    return 0;
}
```

```
marco@discordia:~/uni/PR2$ gcc -Wall -o test1 test1.c
marco@discordia:~/uni/PR2$ ./test1
HelloWorld
```

Intermezzo I:

Definition und Deklaration von Funktionen in C

```
#include <stdio.h>

int main(){
    PrintHello();
    return 0;
}

void PrintHello(){
    printf("HelloWorld\n");
}
```

```
marco@discordia:~/uni/PR2$ gcc -Wall -o test1 test1.c
test1.c: In function `main':
test1.c:4: Warnung: implicit declaration of function `PrintHello'
```

Intermezzo I:

Definition und Deklaration von Funktionen in C

Lösung: Funktion vorher „bekannt machen“ (Funktionsdeklaration)

```
#include <stdio.h>

void PrintHello();        //Funktionsdeklaration

int main(){
    PrintHello();
    return 0;
}

void PrintHello(){       // Funktionsdefinition
    printf("HelloWorld\n");
}
```

Intermezzo II:

File-Struktur eines C-Programmes:

Funktions-Deklaration:
- Die Signaturen der Funktionen
- Wird in eine Header-File ausgelagert (*.h)
- Wird dann den Date-Files (dem eigentlichen Code) includiert

Funktions-Definition:
- Die eigentliche Funktions-Implementierung

Intermezzo II:

File-Struktur eines C-Programmes:

```
// ---   funct.c ---
#include <stdio.h>
void PrintHelloWorld(){
    printf("HelloWorld\n");
}

// --- funct.h ---
void PrintHelloWorld();

// --- main.c ---
#include <stdio.h>
#include "funct.h"
int main(){
    PrintHelloWorld();
    return 0;
}
```

Intermezzo II:

File-Struktur eines C-Programmes:

Gründe:

- Übersichtlichkeit

- Nur geänderte Files werden neu compiliert

- Distribution von Funktionen als
  a) Source
  b) Bibliothek (dynamisch und statisch)

Intermezzo III:

Shared Object Files (dynamische Bibliotheken)

- Funktions-Bibliotheken (=> können von mehrere Applikationen auf dem System gemeinsam genutzt werden)

- Speicherpositionen der Bibliotheksfunktionen werden erst beim Zulinken in die Endanwendung berechnet
  (in der Praxis meist Position-Independent-Code in der Library)

- Dynamisches Linken der Endapplikation : Libs werden dann zur Laufzeit der Anwendung in den Speicher geladen (Speicherabbild)

Schnittstelle zwischen Java und C:

Shared Library

- C-Funktionen werden in Bibliothek realisiert
- Bibliothek wird in Java geladen
- Bibliotheksfunktionen können in Java verwendet werden

Vorgehen (Teil 1) :

1. Java Programm schreiben und native Methode deklarieren

2. Java-Klassen compilieren

Implementierung:

Java-Klasse:

```
class HelloWorld {

    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

Vorgehen (Teil 2) :

1. Java Programm schreiben und native Methode deklarieren

2. Java-Klassen compilieren

3. Mit javah Headerfile für die C-Funktion erzeugen (mit -jni flag)

Implementierung:

Header-File (javah -jni <java_class_name>):

```
#include <jni.h>
#ifndef _Included_HelloWorld
 #define _Included_HelloWorld
 #ifdef __cplusplus
    extern "C" {
 #endif

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *, jobject);

 #ifdef __cplusplus
    }
 #endif
#endif
```

Vorgehen (Teil 3) :

1. Java Programm schreiben und native Methode deklarieren

2. Java-Klassen compilieren

3. Mit javah Headerfile für die C-Funktion erzeugen (mit -jni flag)

4. C-Implementierung schreiben

Implementierung:

C-Code:

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloWorld_
        displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

Vorgehen (Teil 4) :

1. Java Programm schreiben und native Methode deklarieren

2. Java-Klassen compilieren

3. Mit javah Headerfile für die C-Funktion erzeugen (mit -jni flag)

4. C-Implementierung schreiben

5. C-Files als Bibliothek übersetzen

Bibliothek erzeugen:

(am Beispiel für Linux mit gcc):

```
gcc -Wall -fPIC -dPIC -shared -o libhello.so HelloC.c -ldl \
-I/usr/lib/j2se/1.4/include/linux/ -I/usr/lib/j2se/1.4/include/
```

- Nach Lib-Standards mit lib-Prefix (wird beim Laden in Java dann nicht mit angegeben).

- LD_LIBRARY_PATH exportieren:
  export LD_LIBRARY_PATH=$PWD

Interaktion: Java in C

Java (native-Deklaration):

```
class HelloWorld {

   public native void displayHelloWorld(int zahl);
   static {
        System.loadLibrary("hello");
   }

   public static void main(String[] args) {
          new HelloWorld().displayHelloWorld(42);
   }
}
```

Interaktion: Java in C

C (Funktions-Definition):

```
JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env,
jobject obj, jint zahl)
{
        printf("Hello world: %d\n", zahl);
        return;
}
```

Name-Mapping:

```
private native String getLine(String prompt);
```

```
JNIEXPORT jstring JNICALL Java_Prompt_getLine(JNIEnv *, jobject, jstring);
```

- für primitive Typen (int => jint, float => jfloat, ...)
- für Objekte (String => jstring, class => jclass, ...)

(noch mehr) Interaktion:

- *Zugriff auf Java Strings*
- Zugriff auf Java Arrays
- Aufruf von Java Methods
- Zugriff auf Java Member Variablen
- Exception-Handling
- Threads and Native Methods

Jstring:

NICHT SO:

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    printf("%s", prompt);
    ...
```

Jstring:

KORREKT:

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
    printf("%s", str);
    (*env)->ReleaseStringUTFChars(env, prompt, str);
    .....
    char buf[128]
    scanf("%s", buf);
    return (*env)->NewStringUTF(env, buf);
}
```

Jstring: Weitere Methoden zum jstring-Handling

- GetStringChars
- GetStringLength
- GetStringUTFChars
- GetStringUTFLength
- NewString
- NewStringUTF
- ReleaseStringChars
- ReleaseStringUTFChars

(Nachzulesen in der Java API Referenz)

Array-Zugriffe:

- GetBooleanArrayElements
- GetByteArrayElements
- GetCharArrayElements
- GetShortArrayElements
- GetIntArrayElements
- GetLongArrayElements
- GetFloatArrayElements
- GetDoubleArrayElements

(Nachzulesen in der Java API Referenz)

JNI : C für Java