Javas neue Gesprächskultur

Bernd Müller, Ostfalia

Java und C können sich über das Java Native Interface (JNI) unterhalten. Das geht ganz gut, Spaß macht es jedoch sicher nicht. Das Panama-Projekt versucht mit einer ganzen Reihe von JEPs, die Gesprächskultur von Java und C auf ein neues Niveau zu heben, und erlaubt es, nativen Code über pures Java ohne zusätzlichen C-Code aufzurufen.

Dieser Artikel ist eine kurze Einführung in das Foreign Function & Memory API mit dem Schwerpunkt auf die Funktionsschnittstelle. Er soll dem Leser einen ersten Eindruck vermitteln, wie das Zusammenspiel von Java und C funktioniert. Das API ist in Java 17 zwar noch im Inkubator-Modus, aber schon durchaus verwendungsfähig.

Die Entstehung von Panama

Leider können wir es nicht mit letzter Sicherheit sagen, doch die Entstehung des Panama-Projekts scheint seinen Ursprung im JEP 191 (Foreign Function Interface, *[1]*) zu haben. Dieser hatte das Ziel, native Methoden des Betriebssystems direkt durch Java-Methoden aufrufen und nativen Speicher direkt manipulieren zu können. Das JEP wurde kurz nach dem Erscheinen wieder zurückgezogen. Die zu lösenden Probleme waren wohl zu umfangreich und zu kompliziert für ein einzelnes JEP, sodass eine breitere Basis etabliert werden musste: das Panama-Projekt *[2]*, dessen Untertitel *„Interconnecting JVM and native code"* lautet.

Neben der Schnittstelle zu nativen Funktionen und nativem Speicher sind auch `jextract`, ein Werkzeug zur Generierung von Java-Interfaces aus C-Header-Dateien, und das Vector-API Teil von Panama. Wir gehen auf die beiden Letzteren an dieser Stelle nicht weiter ein.

Die Panama zuzurechnenden JEPs sind, was die Quantität angeht, mittlerweile recht umfangreich und umfassen die JEPs 370 *[3]*, 383 *[4]*, 389 *[5]*, 393 *[6]* und 412 *[7]*. Das JEP 370 wurde bereits mit Java 14 ausgeliefert, das JEP 412 mit Java 17. Dieses letzte JEP vereint auch das Funktions- und das Speicher-API. Da es das aktuellste Panama-JEP ist, soll hier dessen Zusammenfassung erwähnt werden:

„Introduce an API by which Java programs can interoperate with code and data outside of the Java runtime. By efficiently invoking foreign functions (i.e., code outside the JVM), and by safely accessing foreign memory (i.e., memory not managed by the JVM), the API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI."

Um die genannten negativen Eigenschaften von JNI ins Gedächtnis zurückzurufen, wollen wir ein typisches Hello-World mit JNI realisieren.

Das Java Native Interface

Java 1.0 enthielt bereits ein *Native Method Interface*. Dieses war jedoch an Interna der JVM gebunden und somit nicht plattformunabhängig. So waren etwa die Schnittstellen der jeweiligen JVMs von Sun und Microsoft verschieden. Mit Java 1.1 wurde eine plattformunabhängige Schnittstelle eingeführt, das *Java Native Interface (JNI) [8]*.

Die Verwendung sieht die Deklaration der nativen Methode mit dem `native` Modifier sowie das Laden der nativen Bibliothek mit `System.loadLibrary()` vor. Ein einfaches Hello-World auf dieser Grundlage zeigt *Listing 1*. Das JDK-Programm `javah`, seit Java 8 überführt in `javac` mit der Option `-h`, erzeugt aus dem Java-Code eine C-Include-Datei, die in *Listing 2* dargestellt ist.

Zu guter Letzt muss noch der C-Code erstellt werden, der aufgerufen wird. Dieser ist in *Listing 3* wiedergegeben. Der Code muss nun in eine Bibliothek des zugrunde liegenden Betriebssystems übersetzt werden, also eine Shared Library unter Linux beziehungsweise eine DLL unter Windows. Wir verzichten auf eine Darstellung der Übersetzungsbefehle sowie der Bibliothekserstellung und verweisen auf das GitHub-Projekt *[9]*, das den gesamten Code dieses Artikels enthält. Es bleibt noch anzumerken, dass Shared Libraries unter Linux die Dateinamenserweiterung `.so` und den Präfix `lib` haben, sodass die mit `System.loadLibrary("hello")` geladene Bibliothek den Namen `libhello.so` tragen muss.

Eine oberflächliche Analyse des Quellcodes zeigt, dass beim Aufruf einer C-Funktion durch eine Java-Methode immer eine Indirektion im Sinne einer zusätzlichen C-Funktion benötigt wird, die die Schnittstelle mithilfe der Include-Datei `jni.h` durch verschiedene Typdefinitionen und Datenstrukturen erst ermöglicht. Dies machte die Java-Entwicklergemeinde auf Dauer nicht glücklich, sodass das Projekt Java Native Access (JNA) *[10]* entstand, um diesen in jedem Projekt strukturell identischen Code überflüssig zu machen.

```
package de.pdbm;

public class HelloWorldJni {

  public static void main(String[] args) {
     System.loadLibrary("hello");
     new HelloWorldJni().sayHello();
  }

  private native void sayHello();

}
```

Listing 1

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class de_pdbm_HelloWorldJni */

#ifndef _Included_de_pdbm_HelloWorldJni
#define _Included_de_pdbm_HelloWorldJni
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     de_pdbm_HelloWorldJni
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_de_pdbm_HelloWorldJni_sayHello
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 2

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

JNIEXPORT void JNICALL Java_de_pdbm_HelloWorldJni_sayHello(JNIEnv *, jobject) {
  printf("Hello World from C with JNI\n");
}
```

Listing 3

Das Projekt Panama ist ein weiterer Versuch in dieser Richtung, der allerdings im Gegensatz zu JNA in die Java-Sprachdefinition einfließen wird beziehungsweise schon als Inkubator eingeflossen ist.

JEP 412: Foreign Function & Memory API

Das JEP 412, Foreign Function & Memory API *[7]*, kurz FFM, definiert Schnittstellen, um Speicher außerhalb der VM zu allozieren, zu lesen und zu schreiben, den Lebenszyklus dieses Speichers zu beschreiben und letztendlich auch, um Funktionen außerhalb der VM aufzurufen. Damit dies mit verschiedenen Sprachen – nicht nur C und C++ – und verschiedenen Betriebssystemen funktioniert, ergibt sich ein recht hohes Abstraktionsniveau dieser Schnittstellen. Eine vollständige Darstellung verbietet sich daher in dieser kurzen Einführung und wird Thema eines zukünftigen Artikels.

Zunächst ist interessant, wie dem Anspruch einer einfacheren Verwendung als JNI Genüge getan wird. *Listing 4* zeigt die Panama-Version des Hello-World aus *Listing 1*. Der Leser wird recht erstaunt sein zu sehen, dass der Code umfangreicher ist. Dies ist dem Umstand geschuldet, dass die Methode `downcallHandle()` explizit verwendet werden muss, während das JNI-Äquivalent in der Include-Datei und der JVM-Implementierung versteckt ist. Das Ziel des `downcallHandle()`-Aufrufs ist es, ein Method-Handle (eingeführt in Java 1.7) zu bestimmen, das neben dem Bezeichner beziehungsweise der Adresse (`MemoryAddress`) sowohl der Java-seitigen (`MethodType`) als auch der C-seitigen Signatur (`FunctionDescriptor`) entspricht. Der Methodenaufruf erfolgt also ausschließlich mit Java-Bordmitteln und lässt damit ein wenig den Mehraufwand verschmerzen. Recht positiv fällt dagegen die Code-Reduktion auf der C-Seite ins Gewicht. *Listing 5* zeigt die vollständige C-Implementierung. Eine Include-Datei wird nicht benötigt.

Auch hier verweisen wir bezüglich des Bauens sowie des Aufrufs der Anwendung auf das GitHub-Projekt *[9]*. Hier soll lediglich angemerkt werden, dass das FFM-API auch unsichere (unsafe) Methoden verwendet und sich im Augenblick noch im Inkubator-Modus befindet.

```
package de.pdbm;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodType;
import java.util.Optional;

import jdk.incubator.foreign.CLinker;
import jdk.incubator.foreign.SymbolLookup;
import jdk.incubator.foreign.FunctionDescriptor;
import jdk.incubator.foreign.MemoryAddress;

public class HelloWorldPanama {

    public static void main(String[] args) throws Throwable {
  new HelloWorldPanama().sayHello();
    }

    public void sayHello() throws Throwable {
     System.loadLibrary("hello"); // libhello.so
     Optional<MemoryAddress> address = SymbolLookup.loaderLookup().lookup("hello");
     MethodHandle hello = CLinker.getInstance().downcallHandle(
            address.get(),
            MethodType.methodType(void.class),
            FunctionDescriptor.ofVoid());
     hello.invokeExact();
    }

}
```

Listing 4

```
#include <stdio.h>

void hello() {
    printf("Hello World from C with Panama\n");
}
```

Listing 5

Beim Aufruf der JVM sind daher die Optionen `--enable-native-access=ALL-UNNAMED` und `--add-modules jdk.incubator.foreign` zu verwenden.

Gesprächskultur ist immer beidseitig

Mit dem FFM-API ist auch die umgekehrte Richtung möglich: C ruft Java auf. Die Funktion `qsort()` der Standard-C-Bibliothek (`stdlib.h`) sortiert ein Array auf Basis eines Quicksort-Algorithmus. Die Signatur der Funktion stellt sich folgendermaßen dar:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Der Parameter `base` ist ein Zeiger auf das erste Element des Arrays. Da C nicht weiß, wie groß ein Array und seine Elemente sind, müssen diese Informationen angegeben werden. `nmemb` gibt die Anzahl der Elemente, `size` die Größe eines einzelnen Elements an. Der Parameter `compar` ist die Vergleichsfunktion, ganz analog zu der in Java etwa von `Collections.sort()` benötigten Comparator-Klasse.

```
package de.pdbm;

import jdk.incubator.foreign.CLinker;
import jdk.incubator.foreign.FunctionDescriptor;
import jdk.incubator.foreign.MemoryAccess;
import jdk.incubator.foreign.MemoryAddress;
import jdk.incubator.foreign.MemorySegment;
import jdk.incubator.foreign.ResourceScope;
import jdk.incubator.foreign.SegmentAllocator;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.util.Arrays;

import static jdk.incubator.foreign.CLinker.*;

/**
 * Simple example to call C from Java as well as Java from C
 *
 * Based on https://github.com/openjdk/panama-foreign/blob/foreign-jextract/doc/panama_ffi.md
 *
 * @author bernd
 *
 */
public class Quicksort {

    public static void main(String[] args) throws Throwable {
        qsort();
    }

    static class Comparator {
        static int compare(MemoryAddress addr1, MemoryAddress addr2) {
            int v1 = MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(), addr1.toRawLongValue());
            int v2 = MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(), addr2.toRawLongValue());
            return v1 - v2;
        }
    }

    public static void qsort() throws Throwable {
        MethodHandle qsort = CLinker.getInstance().downcallHandle(
                CLinker.systemLookup().lookup("qsort").get(),
                MethodType.methodType(void.class, MemoryAddress.class, long.class, long.class, MemoryAddress.
class),
                FunctionDescriptor.ofVoid(C_POINTER, C_LONG, C_LONG, C_POINTER)
        );

        MethodHandle comparHandle = MethodHandles.lookup()
                .findStatic(Comparator.class, "compare",
                        MethodType.methodType(int.class, MemoryAddress.class, MemoryAddress.class));

        try (ResourceScope scope = ResourceScope.newConfinedScope()) {
            MemoryAddress comparFunc = CLinker.getInstance().upcallStub(
                comparHandle,
                FunctionDescriptor.of(C_INT, C_POINTER, C_POINTER), scope);

            MemorySegment array = SegmentAllocator.ofScope(scope)
                                                  .allocateArray(C_INT, new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7
});
            qsort.invokeExact(array.address(), 10L, 4L, comparFunc);
            int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
            System.out.println(Arrays.toString(sorted));
        }
    }

}
```

Listing 6

Im GitHub-Repository des Panama-Projekts *[11]* findet man ein Beispiel zur Verwendung der `qsort`-Funktion, die als `compar`-Funktion eine Java-Methode verwendet. Java ruft also die C-Funktion `qsort()`, diese wiederum für jede Vergleichsoperation des Sortieralgorithmus eine Java-Methode auf. Das *Listing 6* zeigt eine Überarbeitung dieses Beispiels, dessen Code auch in unserem GitHub-Projekt *[9]* zu finden ist. Die Klasse `Comparator` mit der einzigen Methode `compare()` ist intuitiv gut zu verstehen, wenn wir von den FFM-Typen `MemoryAddress`, `MemoryAccess` und `MemorySegment` absehen.

In der Methode `qsort()` wird zunächst in bekannter Manier ein Method-Handle auf die C-Funktion `qsort()` erstellt. Im Gegensatz zu *Listing 4* wird nicht die Methode `SymbolLookup.loaderLookup()`, sondern `Clinker.systemLookup()` verwendet, die Symbole in der Standard-C-Bibliothek sucht. Das zweite Method-Handle `comparHandle` ist die `compare()`-Methode der Klasse `Comparator`. Im `try`-Block erfolgt letztendlich die Verdrahtung des Ganzen und der eigentliche Aufruf der Sortierfunktion.

Im Gegensatz zu `downcallHandle()` für die Aufrufrichtung *Java ruft C* verwendet man die Methode `upcallStub()` für die Richtung *C ruft Java*. Ebenfalls noch interessant ist die Speicherverwaltung. Da das zu sortierende Array außerhalb der JVM liegt, wird dessen Speicher nach Verwendung nicht „garbage-collected". Ein Speicherbereich mit sogenanntem *confined Scope* muss explizit durch Aufruf der `close()`-Methode wieder freigegeben werden. Da die Klasse `ResourceScope` das Interface `AutoCloseable` implementiert, wird das angelegte Array nach Beendigung des `try`-Blocks automatisch wieder freigegeben.

Wir wollen an dieser Stelle nicht zu sehr in die Details gehen, die durchaus recht komplex sind. Wir hoffen jedoch, dass der Leser mit unserer Einführung einen kleinen Einblick gewinnen konnte, wie sich die Verwendung des Foreign Function & Memory API anfühlt.

Zusammenfassung

Das FFM-API erlaubt es, dass C-Funktionen direkt von Java aufgerufen werden können. Es werden kein weiterer Glue-Code und keine Include-Datei benötigt. Diese Pure-Java-Lösung erkauft man sich mit einem nicht ganz trivialen API, das von den Implementierungsdetails des C-Compilers, des Betriebssystems und letztendlich auch von der konkreten Hardware abstrahiert. Das API befindet sich in Java 17 noch im Inkubator-Modus, scheint sich aber mit dem während des Entstehens dieses Artikels veröffentlichten JEP 419 *[12]* zu stabilisieren.

Referenzen

[1] JEP 191: Foreign Function Interface, *https://openjdk.java.net/jeps/191*

[2] Project Panama: Interconnection JVM and native Code, *https://openjdk.java.net/projects/panama/*

[3] JEP 370: Foreign-Memory Access API (Incubator), *https://youtu.be/0HXID3XZuOQ*

[4] JEP 383: Foreign-Memory Access API (Second Incubator), *https://openjdk.java.net/jeps/383*

[5] JEP 389: Foreign Linker API (Incubator), *https://openjdk.java.net/jeps/389*

[6] JEP 393: Foreign-Memory Access API (Third Incubator), *https://openjdk.java.net/jeps/393*

[7] JEP 412: Foreign Function & Memory API (Incubator), *https://openjdk.java.net/jeps/412*

[8] Java Native Interface Specification Contents, *https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html*

[9] *https://github.com/BerndMuller/Panama*

[10] Java Native Access, *https://github.com/java-native-access/jna*

[11] Panama-Foreign Repository, *https://github.com/openjdk/panama-foreign*

[12] JEP 419: Foreign Function & Memory API (Second Incubator), *https://openjdk.java.net/jeps/419*

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.