JNI: Подружим Java и C++

Введение

Бывают моменты, когда в Java некоторые действия выполняются за пределами обычных Java-классов. Например, необходимо исполнить код, написанный на C/C++ или другом каком-нибудь языке.  
  
В данной статье рассмотрим данный вопрос с практической точки зрения, а именно напишем простой пример взаимодействия кода Java с кодом C++, используя JNI. Статья не содержит чего-то сверхестественного, это скорее памятка для тех, кто с этим не работал.  
  
Для наших целей существует возможность динамической загрузки файлов в виде нативных библиотек, вызываемая методом System.load(), о чем более подробно можно прочитать внизу.

Постановка задачи

1. Пусть нам необходимо реализовать класс, содержащий в себе нативный метод, выводящий на экран “Hello world”.  
  
JNIHelloWorld.java  
  
package ru.forwolk.test;

public class JNIHelloWorld {

native void printHelloWorld(); //в «С» нужно будет реализовать этот метод

}

Генерация заголовков

2. Сгенерируем заголовки данного класса для C++.  
Сначала создадим папку в корне проекта, где будем собирать бинарники:  
  
mkdir bin

Затем, скомпилируем наш класс в данную директорию

javac -d bin/ src/ru/forwolk/test/JNIHelloWorld.java

В папке bin у нас появился class-файл. Вернее, в bin/ru/forwolk/test/. Переидем в папку bin и сгенерируем заголовки. Где **javah** производит заголовочные файлы C/С++ и исходные файлы C/С++ от Java class. Эти файлы обеспечивают соединительное связующее звено, которые позволяют Вашему Java и коду C/С++ взаимодействовать.

cd bin/

javah ru.forwolk.test.JNIHelloWorld

3. Как видно, в нашей папке bin появился файл ru\_forwolk\_test\_JNIHelloWorld.h. Для простоты переименуем его в JNIHelloWorld.h

mv ru\_forwolk\_test\_JNIHelloWorld.h JNIHelloWorld.h

Открыв его, видим следующую картину:

JNIHelloWorld.h

/\* DO NOT EDIT THIS FILE - it is machine generated \*/

#include <jni.h>

/\* Header for class ru\_forwolk\_test\_JNIHelloWorld \*/

#ifndef \_Included\_ru\_forwolk\_test\_JNIHelloWorld

#define \_Included\_ru\_forwolk\_test\_JNIHelloWorld

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*

\* Class: ru\_forwolk\_test\_JNIHelloWorld

\* Method: printHelloWorld

\* Signature: ()V

\*/

JNIEXPORT void JNICALL Java\_ru\_forwolk\_test\_JNIHelloWorld\_printHelloWorld

(JNIEnv \*, jobject);

#ifdef \_\_cplusplus

}

#endif

#endif

Реализация на C++

4. Создадим файл с исходниками JNIHelloWorld.cpp. Я для этой цели создал проект в Clion, в который вставил необходимый файл. Реализуем наш метод.

JNIHelloWorld.cpp

#include <iostream>

#include "JNIHelloWorld.h"

JNIEXPORT void JNICALL Java\_ru\_forwolk\_test\_JNIHelloWorld\_printHelloWorld

(JNIEnv \*, jobject) {

std::cout << "Hello world!";

}

5. Чтобы в Clion все работало корректно, необходимо в файл CMakeLists.txt добавить библиотеки Java:

// Вместо $JAVA\_HOME -- путь до Java

include\_directories($JAVA\_HOME/include)

include\_directories($JAVA\_HOME/include/linux)

link\_directories($JAVA\_HOME/include)

link\_directories($JAVA\_HOME/include/linux)

6. Далее компилируем  
  
g++ -I"$JAVA\_HOME/include" -I"$JAVA\_HOME/include/linux" -fPIC JNIHelloWorld.cpp -shared -o helloworld.so -Wl,-soname -Wl,--no-whole-archive

Загрузка в Java

7. В корневой папке проекта появился файл helloworld.so. Переместим его в папку bin/ проекта Java.  
  
8. Теперь необходимо загрузить нашу библиотеку. Хорошей практикой будет статическая загрузка библиотеки прямо в классе. Добавим загрузку прямо в класс JNIHelloWorld  
  
static {

// $PROJECT\_ROOT -- абсолютный путь до библиотеки

System.load("$PROJECT\_ROOT/bin/helloworld.so");

}

9. Теперь мы можем полноценно использовать данный класс. Давайте проверим.

public static void main(String[] args) {

JNIHelloWorld p = new JNIHelloWorld();

p.printHelloWorld();

}

На выходе получаем

Hello world!

Process finished with exit source 0

Передача параметров

А что делать, если нам надо не только выполнять какой-то код, а также передавать параметры и получать ответ? Рассмотрим еще один метод, выполняющий умножение двух чисел. Добавим в класс JNIHelloWorld метод

native int multiply(int a, int b);

Выполняем те же самые действия, описанные выше по генерации заголовков. Как видим, сгенерировалось следующее

/\*

\* Class: ru\_forwolk\_test\_JNIHelloWorld

\* Method: multiply

\* Signature: (II)I

\*/

JNIEXPORT jint JNICALL Java\_ru\_forwolk\_test\_JNIHelloWorld\_multiply

(JNIEnv \*, jobject, jint, jint);

Реализуем метод в JNIHelloWorld.cpp

JNIEXPORT jint JNICALL Java\_ru\_forwolk\_test\_JNIHelloWorld\_multiply

(JNIEnv \*, jobject, jint a, jint b) {

return a \* b; // добавляем строку

}

Опять же произведем описанные выше действия по подтягиванию библиотеки, добавим строку в main по выводу результата произведения двух чисел и запустим

public static void main(String[] args) {

JNIHelloWorld p = new JNIHelloWorld();

System.out.println(p.multiply(2, 2));

p.printHelloWorld();

}

Что получаем в консоли

4

Hello world!

Process finished with exit source 0

Заключение

Мы с вами рассмотрели возможность Java использовать код, написанный на C/C++. Это можно применять в различных целях, например, для увеличения скорости исполнения кода, для защиты кода от прямого вмешательства и для прочих целей. Очень надеюсь, что данная статья поможет вам разобраться в основах JNI.  
  
Весь код я выложил в [открытый доступ](https://bitbucket.org/Forwolk/jnihelloworld). В директории cpp разместил класс C++ без лишних файлов проекта Clion.

JNI, загрузка нативных библиотек. Меняем java.library.path на лету

**Ниже будет рассмотрен специфический случай, когда будут менять путь к загрузке системных библиотек методом System.loadlibrary().**

**Хотя обычно динамические библиотеки лучше менять через, описанный выше, System.load() *(предусмотренный, как я понял, как раз для таких случаев)*.**

В подмножестве экосистемы Java, относящейся в основном к JNI (без которого никуда не деться, если приходиться интегрироваться с каким-то legacy или просто редким и специфическим кодом, написанном на С или каком-то другом языке), есть такое понятие, как java.library.path. Вкратце, это в некотором роде аналог classpath, только не для Java классов и \*.jar файлов, а для нативных библиотек — системное свойство, которое указывает JVM, где искать эти самые нативные библиотеки (.dll в винде или .so под юниксами).  
  
Свойство это устанавливается один раз, перед запуском JVM, через глобальные system properties, или как ключ -Dname=value для JVM, и после этого оно становится read-only. Точнее, менять-то его можно, но никакого эффекта на работу программы это не окажет, т.к. после того как вы обновите это свойство, JVM не перечитает его и не будет использовать новое значение.  
  
Под катом — про то, как все таки поменять это свойство в рантайме, и немного о том, как собственно работает загрузка нативных библиотек в Java.  
  
Однако, возможность менять java.library.path на лету была бы очень кстати — тогда бы не пришлись много раз генерить, переписывать и перезаписывать скрипты для запуска JBoss-a, например, чтобы отразить в них все нужные пути ДО старта аппсервера.  
  
И такая возможность, изменять эти пути, по которым JVM ищет нативные библиотеки, на самом деле есть, изложенное в [этом посте на форумах Sun](http://forums.sun.com/thread.jspa?threadID=707176) , которое вращается вокруг изменения поля *usr\_paths,* хранящегося в классах Java.

public static void addDir(String **s**) throws IOException {

Создание объекта класса Class для ClassLoader

try {

Field field = ClassLoader.class.getDeclaredField("usr\_paths"); // получаем доступ к полю "usr\_paths" ClassLoader-а. В полях ClassLoader-а static private String usr\_paths[]/sys\_paths[] кешируются проинициализированные пути поиска

field.setAccessible(true); //получаем разрешение

String[] paths = (String[])field.get(null);

for (int i = 0; i < paths.length; i++) {

if (**s**.equals(paths[i])) {

return;

}

}

String[] tmp = new String[paths.length+1];

System.arraycopy(paths,0,tmp,0,paths.length); // копируем массив из исходного массива “paths”, начиная с указанной позиции “0”, в указанную позицию “0” другого массива “tmp”.

tmp[paths.length] = s;

field.set(null,tmp);

System.setProperty("java.library.path",System.getProperty("java.library.path") + File.pathSeparator + s); // тут мы в системный файл “properties” записываем ключ ("java.library.path") и значение (все остальное).

//Где System.getProperty("java.library.path") – ищет в системном файле “properties” поле со значением "java.library.path" и получает его значение или “null”. А если написать System.getProperty("а", “bbb”), то не найдя "а", вернется не “null”, а значение по умолчанию “bbb”.

} catch (IllegalAccessException e) {

throw new IOException("Failed to get permissions to set

library path");

} catch (NoSuchFieldException e) {

throw new IOException("Failed to get field handle to set

library path");

}

}

*Коменты:*

*- Я предпочитаю загружать нативные библиотеки из известного места, например, из подпапки 'lib' или, что еще лучше, из каталога jar, из которого был загружен ваш класс.*

*Смотрите System.load (String). (как в примере в предыдущей главе)*

*- Да, это было бы разумно. Однако в этом случае я использовал предварительно упакованную библиотеку (JNotify), которая использовала System.loadLibrary (String) и поэтому требовала, чтобы библиотека находилась в пути к библиотеке.*

Дальше я опишу сам механизм загрузки, и почему то, что описано по ссылкам, работает. Обычно, нативные библиотеки загружаются через статический инициализатор:

static {  
 try {  
  System.loadLibrary(«dp-integ»); //загружает системную библиотеку из “usr\_paths = “java.library.path””  
 }  
}  
  
Который выглядит так:

public static void loadLibrary(String libname) {  
 Runtime.getRuntime().loadLibrary0(getCallerClass(), libname);  
}  
  
И далее в классе Runtime:

synchronized void loadLibrary0(Class fromClass, String libname) {  
 // Проверяем, разрешено ли загружать данную конкретную библиотеку  
 SecurityManager security = System.getSecurityManager(); // получаем объект менеджера безопасности,а если он не установлен, то “null”

 if (security != null) {  
  security.checkLink(libname); //проверяем, можно ли динамически работать с библиотекой  
 }  
 if (libname.indexOf((int)File.separatorChar) != -1) {  
  throw new UnsatisfiedLinkError("Directory separator" +  
   "should not appear in library name: " + libname);  
 }  
 ClassLoader.loadLibrary(fromClass, libname, false); //это не тот-же метод “loadLibrary()”, что вверху. Этот метод описан ниже  
}

Т.е. в итоге, нативные библиотеки загружаются, так же как и обычные классы, через ClassLoader. У класса ClassLoader есть два свойства, в которых кешируются проинициализированные пути поиска.

// The paths searched for libraries  
static private String usr\_paths[];  
static private String sys\_paths[];

Код метода ClassLoader.loadLibrary(fromClass, libname, false), довольно длинный, и загроможденный многочисленными проверками, в сокращенном виде выглядит это так.

// Invoked in the java.lang.Runtime class to implement load and loadLibrary.  
static void loadLibrary(Class fromClass, String name,  
    boolean isAbsolute) {  
    
ClassLoader loader = (fromClass == null) ? null : fromClass.getClassLoader();  
  if (sys\_paths == null) {  
  // это то, что нам нужно  
  usr\_paths = initializePath("java.library.path");  
    
  // а это для тех библиотек, которые загружаются из классов,  
  // загруженных из boot classpath.  
  sys\_paths = initializePath("sun.boot.library.path");  
 }  
  
 // Дальше попытка загрузить библиотеку, и дальше,  
 // если найти ее так и не удалось, то -   
 // Oops, it failed  
 throw new UnsatisfiedLinkError("no " + name + " in java.library.path");  
}

Таким образом, теперь механизм загрузки нативной библиотеки стал более понятен.  
  
Вы можете либо выставить в null свойство sys\_paths у класслоадера, либо просто поменять свойства sys\_paths / usr\_paths, добавив к ним нужные пути к вашим нативным библиотекам.

«Секретное» расширение JNI

Дорог ли native метод?

Ниже будет приведен метод ускоряющий работу DNI для массивов данных малых размеров и большой скорости работы, т.к. на время работы такого метода отключается сборщик мусора. *(имеет ли это смысл на современных компах х.з.)*

Для чего Java-программисты прибегают к native методам? Иногда, чтобы воспользоваться сторонней DLL библиотекой. В других случаях, чтобы ускорить критичный алгоритм за счет оптимизированного кода на C или ассемблере. Например, для обработки потокового медиа, для сжатия, шифрования и т.п.  
  
Но вызов native метода не бесплатен. Порой, накладные расходы на JNI оказываются даже больше, чем выигрыш в производительности. А всё потому, что они включают в себя:

1. создание stack frame;
2. перекладывание аргументов в соответствии с [ABI](http://en.wikipedia.org/wiki/Application_binary_interface);
3. оборачивание ссылок в JNI хендлы (jobject);
4. передачу дополнительных аргументов JNIEnv\* и jclass;
5. захват и освобождение монитора, если метод synchronized;
6. «ленивую» линковку нативной функции;
7. трассировку входа и выхода из метода;
8. перевод потока из состояния in\_Java в in\_native и обратно;
9. проверку необходимости safepoint;
10. обработку возможных исключений.

Но зачастую native методы просты: они не бросают исключений, не создают новые объекты в хипе, не обходят стек, не работают с хендлами и не синхронизованы. Можно ли для них не делать лишних действий?  
  
Да, и сегодня я расскажу о недокументированных возможностях HotSpot JVM для ускоренного вызова простых JNI методов. Хотя эта оптимизация появилась еще с первых версий Java 7, что удивительно, о ней еще никто нигде не писал.

JNI, каким мы его знаем

Рассмотрим для примера простой native метод, получающий на вход массив byte[] и возвращающий сумму элементов. Есть несколько способов работы с массивом в JNI:

* GetByteArrayRegion – копирует элементы Java массива в указанное место нативной памяти;

JNIEXPORT jint JNICALL

Java\_bench\_Natives\_arrayRegionImpl(JNIEnv\* env, jclass cls, jbyteArray array) {

static jbyte buf[1048576];

jint length = (\*env)->GetArrayLength(env, array);

(\*env)->GetByteArrayRegion(env, array, 0, length, buf);

return sum(buf, length);

}

* GetByteArrayElements – то же самое, только JVM сама выделяет область памяти, куда будут скопированы элементы. По окончании работы с массивом необходимо вызвать ReleaseByteArrayElements.

JNIEXPORT jint JNICALL

Java\_bench\_Natives\_arrayElementsImpl(JNIEnv\* env, jclass cls, jbyteArray array) {

jboolean isCopy;

jint length = (\*env)->GetArrayLength(env, array);

jbyte\* buf = (\*env)->GetByteArrayElements(env, array, &isCopy);

jint result = sum(buf, length);

(\*env)->ReleaseByteArrayElements(env, array, buf, JNI\_ABORT);

return result;

}

* Зачем, спросите вы, делать копию массива? Но ведь работать с объектами в Java Heap напрямую из натива нельзя, так как они могут перемещаться сборщиком мусора прямо во время работы JNI метода. Однако есть функция GetPrimitiveArrayCritical, которая возвращает прямой адрес массива в хипе, но при этом запрещает работу GC до вызова ReleasePrimitiveArrayCritical.

JNIEXPORT jint JNICALL

Java\_bench\_Natives\_arrayElementsCriticalImpl(JNIEnv\* env, jclass cls, jbyteArray array) {

jboolean isCopy;

jint length = (\*env)->GetArrayLength(env, array);

jbyte\* buf = (jbyte\*) (\*env)->GetPrimitiveArrayCritical(env, array, &isCopy);

jint result = sum(buf, length);

(\*env)->ReleasePrimitiveArrayCritical(env, array, buf, JNI\_ABORT);

return result;

}

Critical Native

А вот и наш секретный инструмент. Внешне он похож на обычный JNI метод, только с приставкой JavaCritical\_ вместо Java\_. Среди аргументов отсутствуют JNIEnv\* и jclass, а вместо jbyteArray передаются два аргумента: jint length – длина массива и jbyte\* data – «сырой» указатель на элементы массива. Таким образом, Critical Native методу не нужно вызывать дорогие JNI функции GetArrayLength и GetByteArrayElements – можно сразу работать с массивом. На время выполнения такого метода GC будет отложен.

JNIEXPORT jint JNICALL

JavaCritical\_bench\_Natives\_javaCriticalImpl(jint length, jbyte\* buf) {

return sum(buf, length);

}

Как видим, в реализации не осталось ничего лишнего.  
Но чтобы метод мог стать Critical Native, он должен удовлетворять строгим ограничениям:

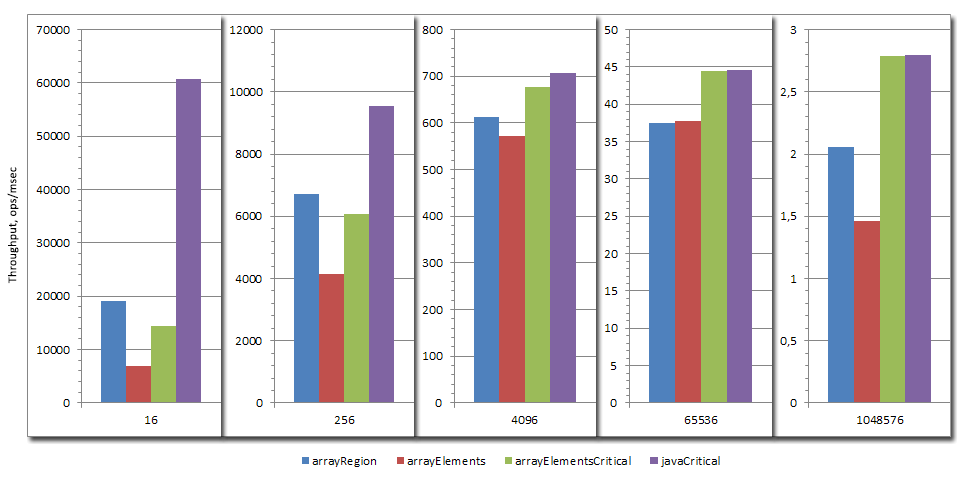
* метод должен быть static и не synchronized;
* среди аргументов поддерживаются только примитивные типы и массивы примитивов;
* Critical Native не может вызывать JNI функции, а, следовательно, аллоцировать Java объекты или кидать исключения;
* и, самое главное, метод должен **завершаться за короткое время**, поскольку на время выполнения он блокирует GC.

Critical Natives задумывался как приватный API Хотспота для JDK, чтобы ускорить вызов криптографических функций, реализованных в нативе. Максимум, что можно найти из описания – [комментарии к задаче в багтрекере](https://bugs.openjdk.java.net/browse/JDK-7013347). Важная особенность: JavaCritical\_ функции вызываются только из горячего (скомилированного) кода, поэтому помимо JavaCritical\_ реализации у метода должна быть еще и «запасная» традиционная JNI реализация. Впрочем, для совместимости с другими JVM так даже лучше.  
  
Сколько будет в граммах?

Давайте, измерим, какова же экономия на массивах разной длины: 16, 256, 4KB, 64KB и 1MB. Естественно, с помощью [JMH](http://openjdk.java.net/projects/code-tools/jmh/).

Бенчмарк

Результаты

  
Оказывается, для маленьких массивов стоимость JNI вызова в разы превосходит время работы самого метода! Для массивов в сотни байт накладные расходы сравнимы с полезной работой. Ну, а для многокилобайтных массивов способ вызова не столь важен – всё время тратится собственно на обработку.  
  
Выводы

Critical Natives – приватное расширение JNI в HotSpot, появившееся с JDK 7. Реализовав JNI-подобную функцию по определенным правилам, можно значительно сократить накладные расходы на вызов native метода и обработку Java-массивов в нативном коде. Однако для долгоиграющих функций такое решение не подойдет, поскольку GC не сможет запуститься, пока исполняется Critical Native.