Seminar Java Native Interface

# Tổng quan JNI

## JNI là gì?

* Java Native Interface (JNI) là phần nền tảng của Java, là một interface trung gian cho phép kết nối ứng dụng Java với các ứng dụng hoặc thư viện viết bằng ngôn ngữ khác (C, C++).
* JNI cho phép gọi các hàm và truyền các biến qua lại giữa chương trình Java & C/C++.
* Cung cấp quy định chung về dữ liệu và loại exception ...
* Một số khái niệm liên quan:
* Host Environment : là những thiết lập và cài đặt trên máy chủ để chạy ứng dụng  
  JNI.
* Thư viện .dll: thư viện .dll phải đạt được tính độc lập riêng .
* JNI : cầu nối trung gian giữa JAVA & .dll.

## Khi nào thì sử dụng?

* Ứng dụng phải truy cập vào các tính năng hệ thống hoặc cá thiệt bị không thể truy cập thông qua nền tảng Java.
* Có một số lượng lớn mã kiểm tra, sửa đổi trong một ngôn ngữ khác và bạn biết cách nào để port tất cả chứng đến nền tảng mong muốn.
* Bạn phát hiện một số trường hợp mã Java là chậm hơn các đoạn mã tương ứng trong các ngôn ngữ khác.

## Mục đích, lợi ích chính của việc dùng JNI

* Sử dụng một ứng dụng, thư viện liên kết động như C, C++ vào trong chương trình Java.
* Nhúng máy ảo Java vào các ứng dụng native (C, C++).

## Một số khó khăn, khuyết điểm

* Khi sử dụng JNI thì ứng dụng Java sẽ không còn dễ dàng chạy trên nhiều môi trường nữa.
* Trong khi ngôn ngữ Java là loại an toàn thì các ngôn ngữ native (C, C++) thì không. Do đó, khi có một phương thức trong ứng dụng hay dll được sử dụng bị hỏng thì sẽ làm cho toàn bộ hệ thống bị hỏng. Do vậy, cần phải kiểm tra các thư viện dll, ứng dụng C, C++ trước khi sử dụng JNI

# Các bước cơ bản để lập trình với JNI:

1. Khai báo một phương thức ***native*** trong một lớp Java.
2. Chạy ***javah*** để có được một ***header*** với một khai báo C cho phương thức.
3. Cài đặt phương thưc ***native*** trong C.
4. Biên dịch mã nguồn C thành thư viện.
5. Load thư viện trong chương trình Java của bạn và sử dụng (cần chỉ đến nơi chứa file đã biên dịch trên.

*Ghi chú*:

* Một số thư viện dll của phương thưc native phải chạy mã khởi tạo. Bạn có thể tạo phương thức JNI\_OnLoad để thực thi mã khởi tạo này. Tương tự như vậy khi máy ảo tắt thì sẽ gọi phương thức JNI\_OnUnload.
* *jint JNI\_OnLoad(JavaVM\* vm, void\* reserved);*
* *void JNI\_OnUnload(JavaVM\* vm, void\* reserved);*
* Trong đó, phương thức JNI\_OnLoad cần phải trả về phiên bản tối thiểu mà máy ảo cần. Ví dụ JNI\_VERSION\_1\_1
* **Ví dụ:** xem phụ lục kèm theo có các bước chi tiết để tạo một ứng dụng java dùng jni đơn giản.

# Các kiểu dữ liệu cơ bản

* Việc tryền và quản lí các biến trong JNI khá phức tạp . Nguyên nhân là   
  do cấu trúc xây dựng và quản lí biến giữa 2 ngôn ngữ C/C++ và Java   
  không tương đồng .
* Để biểu diễn kiểu dữ liệu chung JNI tạo ra một vài kiểu dữ liệu trung   
  gian như sau :
  + jobject jboolean jbyte
  + jchar jshort jint
  + jlong jstring jfloat
  + jdouble …

## Numeric Parameters

* Giữa Java và C có những kiểu dữ liệu đặc trưng khác nhau. Ví dụ trong C kiểu int có 2 nền tảng là 16bit và 32bit, còn trong java kiểu int luôn là 32bit. Do đó JNI định nghĩa ra các kiểu dữ liệu jint, jlong …
* Sự tương ứng trong kiểu dữ liệu:

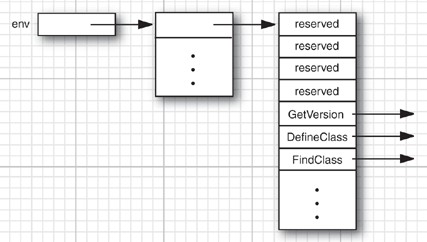
|  |  |  |
| --- | --- | --- |
| JAVA programming | C programming | Bytes |
| boolean | jboolean | 1 |
| byte | jbyte | 1 |
| char | jchar | 2 |
| short | jshort | 2 |
| Int | jint | 4 |
| long | jlong | 8 |
| float | jfloat | 4 |
| double | jdouble | 8 |

* Trong file jni.h cũng định nghĩa được các typedef, JNI\_TRUE=1, JNI\_FALSE=0

## String Parameters

* String trong các ngôn ngữ Java là trình tự các điểm mã UTF-16 trong khi ở ngôn ngữ C, String được chấm dứt bởi byte null.
* String trong jstring (JNI), là một chuỗi unicode 16 bit. Trong C String được xây dựng mặc định từ 8 ký tự bit. Vì vậy, để truy cập vào một đối tượng String trong Java thông qua một C/C++, hoặc trả lại một chuỗi C/C++ cho Java, ta cần phải sử dụng chức năng chuyển đổi JNI trong việc thực hiện.
* JNI tạo ra phương thức để thao tác chuyển đổi chuỗi jstring. Jstring không được sử dụng như 1 string bình thường trong C mà phải qua các phương thức trong JNI
* Jstring có các phương thức hỗ trợ căn bản sau:

|  |  |
| --- | --- |
| * **GetStringChars** * **ReleaseStringChars** | Lấy chuỗi String thường |
| * **GetStringUTFChars** * **ReleaseStringUTFChars** | Lấy ra chuỗi String UTF |
| * **GetStringLength** * **GetStringUTFLength** | Trả về độ dài chuỗi String |
| * **NewString** * **NewStringUTF** | Tạo chuỗi String mới |
| * **GetStringCritical** * **ReleaseStringCritical** | Lấy chuỗi String từ con trỏ truyền vào |

* Tất cả các lời gọi phương thức JNI đều sử dụng con trỏ env là tham số đầu tiên của các phương thức. Con trỏ env là 1 con trỏ đến bảng các con trỏ chức năng. Vì vậy phải thêm (\*env)-> trước các lời gọi hàm. Hơn nữa, env là tham số đầu tiên của các hàm JNI.

## Array

* JNI xử lý *các* mảng nguyên thủy và các mảng đối tượng khác nhau. Mảng nguyên thủy có chứa các yếu tố là các loại nguyên thủy như int và boolean. Mảng đối tượng chứa các yếu tố là các loại object chẳng hạn như trường hợp class và các mảng 2 chiều.
* JNI sử dụng jarray để tao tác với mảng
* Jarray không phải là 1 loại mảng trong C, vì vậy phải sử dụng các phương thức jarray trong JNI thích hợp để truy cập mảng.
* Các phương thức hỗ trợ căn bản:

|  |  |
| --- | --- |
| * **Get<Type>ArrayRegion** * **Set<Type>ArrayRegion** | Lấy ra mảng từ 1 mảng căn bản |
| * **Get<Type>ArrayElements** * **Release<Type>ArrayElement** | Nhận và hủy 1 phần tử mảng |
| * **GetArrayLength** | Trả về số phần tử mảng |
| * **New<Type>Array** | Tạo mảng mới |
| * **GetPrimitiveArrayCritical** * **ReleasePrimitiveArrayCritical** | Nhận và hủy 1 mảng từ biến con trỏ gửi vào |

# Field và Method

## Truy xuất Field

JNI cung cấp các hàm để native code truy xuất (get/set) các field của object và các static field của class.

### Truy xuất Field của đối tượng

Để truy xuất vào một Field JNI cung cấp các hàm sau:

(\*env)->GetAAAField(env, jclass, fieldID);

|  |  |
| --- | --- |
| **Get<Type>Field** | **<NativeType>** |
| GetObjectField | jobject |
| GetBooleanField | jboolean |
| GetByteField | jbyte |
| GetCharField | jchar |
| GetShortField | jshort |
| GetIntField | jint |
| GetLongField | jlong |
| GetFloatField | jfloat |
| GetDoubleField | jdouble |

**Parameters**

env: JNIEnv interface pointer.

jclass: trỏ đến class có field được truy xuất.

fieldID: một biến có kiểu jfieldID đại diện cho field được truy xuất.

**Return Values**

Trả về giá trị của field.

jclass là một biến có kiểu jclass có đượng bằng hàm:

jclass cls = (\*env)->GetObjectClass(env, obj);

Với env, obj là tham số đầu vào của hàm JNI.

Để có được fieldID ta gọi hàm:

jfieldID GetFieldID(JNIEnv \*env, jclass clazz,  
const char \*name, const char \*sig);

**Parameters**

env: JNIEnv interface pointer.

jclass: trỏ đến class có field được truy xuất.

name: chuỗi tên field trong java.

sig:  field descriptor – được nói rõ trong mục ngay phía sau.

**Return Values**

Trả về fieldID.

Để **set** giá trị field ta dùng các hàm sau:

**Prototype** void *Set<Type>Field*(JNIEnv \*env, jobject obj,   
jfieldID fieldID,*<NativeType>* value);

|  |  |
| --- | --- |
| **Set<Type>Field** | **<NativeType>** |
| SetObjectField | jobject |
| SetBooleanField | jboolean |
| SetByteField | jbyte |
| SetCharField | jchar |
| SetShortField | jshort |
| SetIntField | jint |
| SetLongField | jlong |
| SetFloatField | jfloat |
| SetDoubleField | jdouble |

Ý nghĩa các trường tương tự hàm get.

### Field Descriptors

Là một chuỗi dùng để biểu diễn kiểu dữ liệu của field.

Với các kiểu cơ bản mô tả sẽ đơn giản hơn như: int có chuỗi mô tả là “I”, float có “F”, double có “D”, boolean có “Z”.

Với các kiểu dữ liệu khác thì mô tả theo package name của nó. Chẳng hạn như kiểu String có thuộc package: java.lang.String thì chuỗi mô tả tương ứng là “Ljava/lang/String;”.

Có một cách khác để biết chuỗi mô tả của các filed trong lớp java là dùng tool javap đi kèm trong JDK với tham số -s -p TenClass

### Truy xuất static field

Static field truy xuất giống như field của đối tượng chỉ là tên các hàm có thay đổi chút ít như thay vì gọi GetFieldID thì ta gọi GetStaticFieldID tương tự đối với các hàm Get/Set dữ liệu.

## Gọi Method

Trong java cũng có vài loại method như method của đối tượng, static method, và các hàm dựng của đối tượng.

### Gọi method của object

Đầu tiên ta cần có MethodId của method cần gọi bằng cách gọi hàm:

jmethodID GetMethodID(JNIEnv \*env, jclass clazz,   
const char \*name, const char \*sig);

Các tham số giống như phần truy xuất Field. Tuy nhiên phần sig sẽ là method descriptor.

Gọi method bằng hàm sau:

**Prototype** <*NativeType>* *Call<Type>Method*(JNIEnv \*env,  
jobject obj, jmethodID methodID, ...);

|  |  |
| --- | --- |
| **Call<Type>Method** | **<NativeType>** |
| CallVoidMethod | void |
| CallObjectMethod | jobject |
| CallBooleanMethod | jboolean |
| CallByteMethod | jbyte |
| CallCharMethod | jchar |
| CallShortMethod | jshort |
| CallIntMethod | jint |
| CallLongMethod | jlong |
| CallFloatMethod | jfloat |
| CallDoubleMethod | jdouble |

Hàm trả về kết quả gọi hàm. Các tham số đã được nói ở phần trên. Tham số của method có thể truyền vào phần sau hàm.

### Method descriptor

Là một chuỗi tương tự như prototype của method nhưng có mô tả khác. Gồm 2 phần “(?)?” Với chấm hỏi thứ nhất là phần các tham số truyền vào method, Chấm hỏi thứ hay là giá trị trả về. Còn nội dụng của ? thì giống với Field descriptor. Lưu ý là void có ký hiệu là “V”. Một vài ví dụ:

“()V”

“(I)I”

“(Ljava/lang/String;)Ljava/lang/String;”

### Gọi static method

Hoàn toàn giống với gọi hàm của object. Chỉ khác nhau ở tên hàm trong JNI như thay vì gọi GetMethodID, CallVoidMethod… thì sẽ gọi GetStaticMethodID, CallStaticVoidMethod…

### Gọi method của một lớp cha của object

Trong JNI có thể gọi hàm của lớp cha (khi dùng đa hình) bằng hàm:

|  |  |
| --- | --- |
| **CallNonvirtual<Type>Method** | **<NativeType>** |
| CallNonvirtualVoidMethod | void |
| CallNonvirtualObjectMethod | jobject |
| CallNonvirtualBooleanMethod | jboolean |
| CallNonvirtualByteMethod | jbyte |
| CallNonvirtualCharMethod | jchar |
| CallNonvirtualShortMethod | jshort |
| CallNonvirtualIntMethod | jint |
| CallNonvirtualLongMethod | jlong |
| CallNonvirtualFloatMethod | jfloat |
| CallNonvirtualDoubleMethod | jdouble |

Việc gọi hàm này tương đương với lời gọi super.f() trong java.

### Tạo dựng đối tượng Java trong JNI

JNI cho phép ta lấy được hàm dựng của một đối tượng qua đó tạo dựng được đối tượng Java.

Cũng giống như gọi method bình thường ta cần có methodId, jclass…

Để có được methodId có thể gọi hàm GetMethodID với name là “<init>” mà chuỗi mô tả method là tùy theo tham số đầu vào của hàm dựng (giá trị trả về luôn là “V” vì hàm dựng không trả về).

Để tạo đối tượng với hàm dựng có được ta gọi hàm:

jobject NewObject(JNIEnv \*env, jclass clazz,   
jmethodID methodID, ...);

# Handing errors & Invocation API

## Handing errors

Các hàm native code được xem là làm mất an toàn cho chương trình java vì “C runtime system” không có cơ chế xử lý ngoại lệ như: truy cập ngoài vùng mảng, truy cập địa chỉ không hợp lệ… Do đó một vấn đề quan trọng trong lập trình native là phải kiểm tra các lỗi có thể phát sinh và quăng –“throw” về cho chương trình java bắt.

### Throw một ngoại lệ cho chương trình java

JNI cung cấp hai hàm để throw lỗi:

jint Throw(JNIEnv \*env, jthrowable obj);

**Parameters**

env: the JNIEnv interface pointer.

obj: a java.lang.Throwable object.

**Return Values**: trả về 0 nếu thành công, ngược lại trả về số âm.

Để dùng hàm này thì ta phải tạo một đối tượng có interface là Throwable (cũng lấy từ JNI). Sau đó mới gọi hàm Throw. Còn hàm ThrowNew thì ta chỉ cần truyền vào class cài đặt interface Throwable không cần phải tạo đối tượng.

jint ThrowNew(JNIEnv \*env, jclass clazz, const char \*message);

**Parameters**

env: the JNIEnv interface pointer.

clazz: một lớp cài đặt của java.lang.Throwable.

message: chuỗi thông báo dùng để dựng đối tượng Throwable kiểu clazz được truyền vô.

**Return Values**: trả về 0 nếu thành công, ngược lại trả về số âm.

Lưu ý là khi Throw một ngoại lệ thì native code vẫn tiếp tục chạy. Khi kết thúc hàm native thì JVM mới phát sinh ra ngoại lệ. Nêu nếu muốn native code không chạy tiếp khi có ngoại lệ thì cần gọi return ngay sau lệnh Throw.

### Nhận ngoại lệ từ chương trình Java

Khi gọi thực thi một method java từ navite code. Chúng ta sẽ cần biết method chạy thế nào, có bị ngoại lệ gì không. JNI có hỗ trợ cho chúng ta làm việc này.

JNI cũng cấp hai hàm kiểm tra xem có ngoại lệ xảy ra không và hàm lấy về đối tượng ngoại lệ:

jboolean ExceptionCheck(JNIEnv \*env);

**Return Values** trả về JNI\_TRUE nếu có ngoại lệ sảy ra, hoặc JNI\_FALSE nếu ngược lại.

jthrowable ExceptionOccurred(JNIEnv \*env);

**Return Values** trả về đối tượng có interface Throwable nếu có ngoại lệ sảy ra, hoặc NULL nếu ngược lại.

## Sử dụng Invocation API

Giả sử bạn có một chương trình C/C++ và muốn gọi Java code. Chính lúc này invocation API của JVM sẽ giúp bạn. Cho phép chúng ta tạo một Java Virtual Machine trong chương trình C/C++.

JNI cung cấp hai hàm để tạo và hủy máy ảo java.

### Tạo JVM

jint JNI\_CreateJavaVM(JavaVM\*\* p\_jvm, void\*\* p\_env, JavaVMInitArgs\* vm\_args)

Khởi tạo một JVM. Hàm trả về 0 nếu thành công, JNI\_ERR khi ngược lại.

| **Parameters:** | p\_jvm | Con trỏ có kiểu JavaVM, dùng biến này để thao tác lên JVM tạo ra. |
| --- | --- | --- |
|  | p\_env | Con trỏ kiểu JNIEnv, dùng biến này để thực hiện các lệnh JNI (như gọi chạy hàm main chẳng hạn). |
|  | vm\_args | Biến kiểu JavaVMInitArgs làm tham sô truyền vào cho JVM. |

Mô tả của cấu trúc tham số JavaVMInitArgs

typedef struct [JavaVMOption](http://xdprof.sourceforge.net/doxygen/structJavaVMOption.html) {

char \*optionString;

void \*extraInfo;

} [JavaVMOption](http://xdprof.sourceforge.net/doxygen/structJavaVMOption.html);

typedef struct [JavaVMInitArgs](http://xdprof.sourceforge.net/doxygen/structJavaVMInitArgs.html) {

jint version; // phiên bản của jni có JNI\_VERSION\_1\_2 hoặc JNI\_VERSION\_1\_1

jint nOptions;

[JavaVMOption](http://xdprof.sourceforge.net/doxygen/structJavaVMOption.html) \*options;

jboolean ignoreUnrecognized;

} [JavaVMInitArgs](http://xdprof.sourceforge.net/doxygen/structJavaVMInitArgs.html);

Một tham số quan trọng cần truyền vào khi tạo JVM là class\_path. Để thêm vị trí hiện tại vào class\_path ta tạo option sau:

JavaVMOption options[1];

options[0].optionString = "-Djava.class.path=.";

Sau đó biến options đẽ được gán vào trương option của struct [JavaVMInitArgs](http://xdprof.sourceforge.net/doxygen/structJavaVMInitArgs.html)

### Hủy JVM

Để hủy một JVM đã tạo ta dùng hàm:

jint DestroyJavaVM(JavaVM\* jvm)

Returns: trả về 0 nếu thành công, một số âm nếu thất bại. Hàm này cần phải được gọi thông qua một con trỏ JavaVM được truyền vô lúc tạo JVM.

# Jave Native Access

## Tổng quan Jave Native Access

* Java Native Access là một phần mở rộng của Java cho phép sử dụng các API bao gồm các tập tin thư viện động DLL trong Windows.
* JNA cho phép gọi hàm trực tiếp bằng các phương thức tự nhiên trong java mà không cần sử dụng JNI
* JNA bao gồm các thư viện nền tảng với nhiều native functions được ánh xạ và một tập hợp các tiện ích giúp đơn giản hóa việc truy cập
* Để sử dụng JNA ta cần download tập tin jna.jar và import và chương trình. Link down jna.jar: <https://github.com/twall/jna>

## Điểm đặc trưng của JNA

* Tự động ánh xạ từ java các native functions với các kiểu dữ liệu cơ bản.
* Chạy trên hầu hết các nền tảng có hỗ trợ java
* Tự động chuyển đổi chuỗi giữa C và Java
* Tự động tạo ra các con trỏ hàm
* Tự động hỗ trợ các hàm trong Windows về ASCII/UNICODE
* Tối ưu hóa việc ánh xạ trực tiếp giúp chương trình hoạt động tốt hơn

## Các thư viện tiêu biểu trong JNA

* Các thư viện cần khai báo khi sử dụng JNA:

|  |  |
| --- | --- |
| com.sun.jna | Cung cấp các truy cập đơn giản đến native library |
| com.sun.jna.ptr | Cung cấp các loại con trỏ khác nhau (<type> \*) |
| com.sun.jna.win32 | Cung cấp các kiểu dữ liệu và hàm tiêu chuẩn để làm việc trên nền tảng win32 |

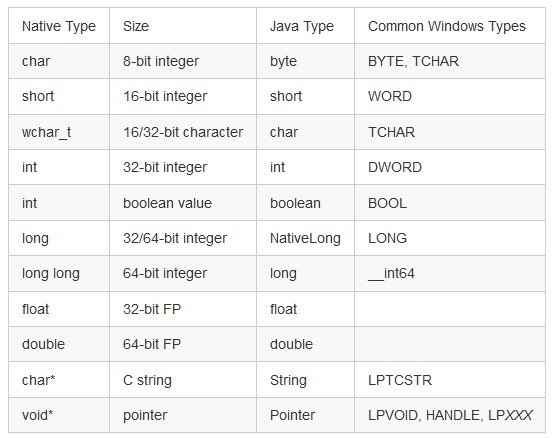
* Ngoài ra JNA còn cung cấp các thư viện nền tảng đặc trưng khác để làm việc với các môi trường như linux, macos

|  |  |
| --- | --- |
| com.sun.jna.platform.mac | Cung cấp thư viện hỗ trợ làm việc trên macos |
| com.sun.jna.platform.unix | Cung cấp thư viện hỗ trợ làm việc trên linux |
| com.sun.jna.platform.win32 | Cung cấp thư viện làm việc trên nền tảng windows |

* Cần chắc chắn là thư viện JNA đã được load trong chương trình, có 2 cách để làm là:
  + Thiết lập thuộc tính hệ thống jna.library.path chỉ tới thư viện bạn cần sử dụng
  + Thay đổi biến môi trường để tích hợp thư viện JNA. Trong windows là PATH, linux là LD\_LIBRARY\_PATH và DYLD\_LIBRARY\_PATH trên OSX

## Một số kiến thức cơ bản khi sử dụng JNA

* Các kiểu dữ liệu cơ bản được JNA ánh xạ trực tiếp từ native C với cùng kích thước



* Nếu tham số được sử dụng bởi các hàm native có nguồn gốc bên ngoài thì phải sử dụng 1 bộ nhớ đệm NIO. Bộ nhớ này được cung cấp như 1 mảng trong java và chỉ có giá trị trong suốt lời gọi hàm.
* Mảng chuỗi trong C được JNA biểu diễn là String[] với kí tự null ở cuối mảng
* Khi một hàm chấp nhận đối số là con trỏ ta sử dụng kiểu ByReference trong JNA để lấy giá trị trả về hoặc phân lớp cho hàm.
* Các hàm cross-platform và cấu trúc được cung cấp trong thư viện com.sun.jna.platform dùng để thao tác với các tập tin hệ thống, phần cứng và cửa sổ chương trình
* JNA hỗ trợ các phương pháp ánh xạ trực tiếp giúp tăng hiệu suất của chương trình bằng các tùy chỉnh trong JNI