The Invocation API

Chapter 5

The Invocation API allows software vendors to load the Java VM into an arbitrary native application. Vendors can deliver Java-enabled applications without having to link with the Java VM source code.

This chapter begins with an overview of the Invocation API. This is followed by reference pages for all Invocation API functions.

Overview

The following code example illustrates how to use functions in the Invocation API. In this example, the C++ code creates a Java VM and invokes a static method, called `Main.test`. For clarity, we omit error checking.

```
#include <jni.h>       /* where everything is defined */
...
JavaVM *jvm;       /* denotes a Java VM */
JNIEnv *env;       /* pointer to native method interface */
JavaVMInitArgs vm_args; /* JDK/JRE 6 VM initialization arguments */
JavaVMOption* options = new JavaVMOption[1];
options[0].optionString = "-Djava.class.path=/usr/lib/java";
vm_args.version = JNI_VERSION_1_6;
vm_args.nOptions = 1;
vm_args.options = options;
vm_args.ignoreUnrecognized = false;
/* load and initialize a Java VM, return a JNI interface
 * pointer in env */
JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
delete options;
/* invoke the Main.test method using the JNI */
jclass cls = env->FindClass("Main");
jmethodID mid = env->GetStaticMethodID(cls, "test", "(I)V");
env->CallStaticVoidMethod(cls, mid, 100);
/* We are done. */
jvm->DestroyJavaVM();
```

This example uses three functions in the API. The Invocation API allows a native application to use the JNI interface pointer to access VM features. The design is similar to Netscape's JRI Embedding Interface.

Creating the VM

The `JNI_CreateJavaVM()` function loads and initializes a Java VM and returns a pointer to the JNI interface pointer. The thread that called `JNI_CreateJavaVM()` is considered to be the *main thread*.

Attaching to the VM

The JNI interface pointer (`JNIEnv`) is valid only in the current thread. Should another thread need to access the Java VM, it must first call `AttachCurrentThread()` to attach itself to the VM and obtain a JNI interface pointer. Once attached to the VM, a native thread works just like an ordinary Java thread running inside a native method. The native thread remains attached to the VM until it calls `DetachCurrentThread()` to detach itself.

The attached thread should have enough stack space to perform a reasonable amount of work. The allocation of stack space per thread is operating system-specific. For example, using pthreads, the stack size can be specified in the `pthread_attr_t` argument to `pthread_create`.

Detaching from the VM

A native thread attached to the VM must call `DetachCurrentThread()` to detach itself before exiting. A thread cannot detach itself if there are Java methods on the call stack.

Unloading the VM

The `JNI_DestroyJavaVM()` function unloads a Java VM. As of JDK/JRE 1.1, only the main thread could unload the VM, by calling `DestroyJavaVM`. As of JDK/JRE 1.2, the restriction was removed, and any thread may call `DestroyJavaVM` to unload the VM.

The VM waits until the current thread is the only non-daemon user thread before it actually unloads. User threads include both Java threads and attached native threads. This restriction exists because a Java thread or attached native thread may be holding system resources, such as locks, windows, and so on. The `VM` cannot automatically free these resources. By restricting the current thread to be the only running thread when the VM is unloaded, the burden of releasing system resources held by arbitrary threads is on the programmer.

Library and Version Management

As of JDK/JRE 1.1, once a native library is loaded, it is visible from all class loaders. Therefore two classes in different class loaders may link with the same native method. This leads to two problems:

- A class may mistakenly link with native libraries loaded by a class with the same name in a different class loader.
- Native methods can easily mix classes from different class loaders. This breaks the name space separation offered by class loaders, and leads to type safety problems.

As of JDK/JRE 1.2, each class loader manages its own set of native libraries. **The same JNI native library cannot be loaded into more than one class loader.** Doing so causes `UnsatisfiedLinkError` to be thrown. For example, `System.loadLibrary` throws an `UnsatisfiedLinkError` when used to load a native library into two class loaders. The benefits of the new approach are:

- Name space separation based on class loaders is preserved in native libraries. A native library cannot easily mix classes from different class loaders.
- In addition, native libraries can be unloaded when their corresponding class loaders are garbage collected.

To facilitate version control and resource management, JNI libraries as of JDK/JRE 1.2 optionally export the following two functions:

JNI_OnLoad

```
jint JNI_OnLoad(JavaVM *vm, void *reserved);
```

The VM calls `JNI_OnLoad` when the native library is loaded (for example, through `System.loadLibrary`). `JNI_OnLoad` must return the JNI version needed by the native library.
In order to use any of the new JNI functions, a native library must export a `JNI_OnLoad` function that returns `JNI_VERSION_1_2`. If the native library does not export a `JNI_OnLoad` function, the VM assumes that the library only requires JNI version `JNI_VERSION_1_1`. If the VM does not recognize the version number returned by `JNI_OnLoad`, the native library cannot be loaded.

LINKAGE:
Exported from native libraries that contain native method implementation.

SINCE:
JDK/JRE 1.4

In order to use the JNI functions introduced in J2SE release 1.2, in addition to those that were available in JDK/JRE 1.1, a native library must export a `JNI_OnLoad` function that returns `JNI_VERSION_1_2`.

In order to use the JNI functions introduced in J2SE release 1.4, in addition to those that were available in release 1.2, a native library must export a `JNI_OnLoad` function that returns `JNI_VERSION_1_4`.

If the native library does not export a `JNI_OnLoad` function, the VM assumes that the library only requires JNI version `JNI_VERSION_1_1`. If the VM does not recognize the version number returned by `JNI_OnLoad`, the native library cannot be loaded.

JNI_OnUnload

```
void JNI_OnUnload(JavaVM *vm, void *reserved);
```

The VM calls `JNI_OnUnload` when the class loader containing the native library is garbage collected. This function can be used to perform cleanup operations. Because this function is called in an unknown context (such as from a finalizer), the programmer should be conservative on using Java VM services, and refrain from arbitrary Java call-backs.
Note that `JNI_OnLoad` and `JNI_OnUnload` are two functions optionally supplied by JNI libraries, not exported from the VM.

LINKAGE:
Exported from native libraries that contain native method implementation.

Invocation API Functions

The `JavaVM` type is a pointer to the Invocation API function table. The following code example shows this function table.

```
typedef const struct JNIInvokeInterface *JavaVM;


const struct JNIInvokeInterface ... = {
    NULL,
    NULL,
    NULL,
```

```
    DestroyJavaVM,
    AttachCurrentThread,
    DetachCurrentThread,

    GetEnv,

    AttachCurrentThreadAsDaemon
};
```

Note that three Invocation API functions, `JNI_GetDefaultJavaVMInitArgs()`, `JNI_GetCreatedJavaVMs()`, and `JNI_CreateJavaVM()`, are not part of the JavaVM function table. These functions can be used without a preexisting `JavaVM` structure.

JNI_GetDefaultJavaVMInitArgs

```
jint JNI_GetDefaultJavaVMInitArgs(void *vm_args);
```

Returns a default configuration for the Java VM. Before calling this function, native code must set the vm_args->version field to the JNI version it expects the VM to support. After this function returns, vm_args->version will be set to the actual JNI version the VM supports.

LINKAGE:

Exported from the native library that implements the Java virtual machine.

PARAMETERS:

`vm_args`: a pointer to a `JavaVMInitArgs` structure in to which the default arguments are filled.

RETURNS:

Returns `JNI_OK` if the requested version is supported; returns a JNI error code (a negative number) if the requested version is not supported.

JNI_GetCreatedJavaVMs

```
jint JNI_GetCreatedJavaVMs(JavaVM **vmBuf, jsize bufLen, jsize *nVMs);
```

Returns all Java VMs that have been created. Pointers to VMs are written in the buffer vmBuf in the order they are created. At most bufLen number of entries will be written. The total number of created VMs is returned in *nVMs.

As of JDK/JRE 1.2, creation of multiple VMs in a single process is not supported.

LINKAGE:

Exported from the native library that implements the Java virtual machine.

PARAMETERS:

`vmBuf`: pointer to the buffer where the VM structures will be placed.

`bufLen`: the length of the buffer.

`nVMs`: a pointer to an integer.

RETURNS:

Returns `JNI_OK` on success; returns a suitable JNI error code (a negative number) on failure.

JNI_CreateJavaVM

```
jint JNI_CreateJavaVM(JavaVM **p_vm, void **p_env, void *vm_args);
```

Loads and initializes a Java VM. The current thread becomes the main thread. Sets the `env` argument to the JNI interface pointer of the main thread.

As of JDK/JRE 1.2 , creation of multiple VMs in a single process is not supported.
The second argument to `JNI_CreateJavaVM` is always a pointer to `JNIEnv *`, while the third argument is a pointer to a `JavaVMInitArgs` structure which uses option strings to encode arbitrary VM start up options:

```
typedef struct JavaVMInitArgs {
    jint version;

    jint nOptions;
    JavaVMOption *options;
    jboolean ignoreUnrecognized;
} JavaVMInitArgs;
```

The `version` field must be set to at least `JNI_VERSION_1_2`. The `options` field is an array of the following type:

```
typedef struct JavaVMOption {
    char *optionString;  /* the option as a string in the default platform encoding */
    void *extraInfo;
} JavaVMOption;
```

The size of the array is denoted by the nOptions field in `JavaVMInitArgs`. If `ignoreUnrecognized` is `JNI_TRUE`, `JNI_CreateJavaVM` ignore all unrecognized option strings that begin with "`-X`" or "`_`". If `ignoreUnrecognized` is `JNI_FALSE`, `JNI_CreateJavaVM` returns `JNI_ERR` as soon as it encounters any unrecognized option strings. All Java VMs must recognize the following set of standard options:

optionString	meaning
`-D<name>=<value>`	Set a system property
`-verbose[:class|gc|jni]`	Enable verbose output. The options can be followed by a comma-separated list of names indicating what kind of messages will be printed by the VM. For example, "`-verbose:gc,class`" instructs the VM to print GC and class loading related messages. Standard names include: `gc`, `class`, and `jni`. All nonstandard (VM-specific) names must begin with "`X`".
`vfprintf`	`extraInfo` is a pointer to the `vfprintf` hook.
`exit`	`extraInfo` is a pointer to the `exit` hook.
`abort`	`extraInfo` is a pointer to the `abort` hook.

In addition, each VM implementation may support its own set of non-standard option strings. Non-standard option names must begin with "`-X`" or an underscore ("`_`"). For example, the JDK/JRE supports `-Xms` and `-Xmx` options to allow programmers specify the initial and maximum heap size. Options that begin with "`-X`" are accessible from the "`java`" command line.

Here is the example code that creates a Java VM in the JDK/JRE:

```
JavaVMInitArgs vm_args;
JavaVMOption options[4];

options[0].optionString = "-Djava.compiler=NONE";           /* disable JIT */
options[1].optionString = "-Djava.class.path=c:\myclasses"; /* user classes */
options[2].optionString = "-Djava.library.path=c:\mylibs";  /* set native library path */
options[3].optionString = "-verbose:jni";                   /* print JNI-related messages */

vm_args.version = JNI_VERSION_1_2;
vm_args.options = options;
vm_args.nOptions = 4;
vm_args.ignoreUnrecognized = TRUE;

/* Note that in the JDK/JRE, there is no longer any need to call
 * JNI_GetDefaultJavaVMInitArgs.
 */
res = JNI_CreateJavaVM(&vm, (void **)&env, &vm_args);
if (res < 0) ...
```

LINKAGE:

Exported from the native library that implements the Java virtual machine.

PARAMETERS:

`p_vm`: pointer to the location where the resulting VM structure will be placed.

`p_env`: pointer to the location where the JNI interface pointer for the main thread will be placed.

`vm_args`: Java VM initialization arguments.

RETURNS:

Returns `JNI_OK` on success; returns a suitable JNI error code (a negative number) on failure.

DestroyJavaVM
`jint DestroyJavaVM(JavaVM *vm);`

Unloads a Java VM and reclaims its resources.

The support for `DestroyJavaVM` was not complete in JDK/JRE 1.1. As of JDK/JRE 1.1 Only the main thread may call `DestroyJavaVM`. Since JDK/JRE 1.2, any thread, whether attached or not, can call this function. If the current thread is attached, the VM waits until the current thread is the only non-daemon user-level Java thread. If the current thread is not attached, the VM attaches the current thread and then waits until the current thread is the only non-daemon user-level thread. The JDK/JRE still does not support VM unloading, however.

LINKAGE:
Index 3 in the JavaVM interface function table.

PARAMETERS:
`vm`: the Java VM that will be destroyed.

RETURNS:
Returns `JNI_OK` on success; returns a suitable JNI error code (a negative number) on failure.

As of JDK/JRE 1.1.2 unloading of the VM is not supported.

AttachCurrentThread
`jint AttachCurrentThread(JavaVM *vm, void **p_env, void *thr_args);`

Attaches the current thread to a Java VM. Returns a JNI interface pointer in the `JNIEnv` argument.

Trying to attach a thread that is already attached is a no-op.

A native thread cannot be attached simultaneously to two Java VMs.

When a thread is attached to the VM, the context class loader is the bootstrap loader.

LINKAGE:
Index 4 in the JavaVM interface function table.

PARAMETERS:
`vm`: the VM to which the current thread will be attached.

`p_env`: pointer to the location where the JNI interface pointer of the current thread will be placed.

`thr_args`: can be NULL or a pointer to a `JavaVMAttachArgs` structure to specify additional information:

As of JDK/JRE 1.1, the second argument to `AttachCurrentThread` is always a pointer to `JNIEnv`. The third argument to `AttachCurrentThread` was reserved, and should be set to `NULL`.

As of JDK/JRE 1.2, you pass `NULL` as the third argument for 1.1 behavior, or pass a pointer to the following structure to specify additional information:

```
typedef struct JavaVMAttachArgs {
    jint version;  /* must be at least JNI_VERSION_1_2 */
    char *name;    /* the name of the thread as a modified UTF-8 string, or NULL */
    jobject group; /* global ref of a ThreadGroup object, or NULL */
} JavaVMAttachArgs
```

RETURNS:
Returns `JNI_OK` on success; returns a suitable JNI error code (a negative number) on failure.

AttachCurrentThreadAsDaemon
`jint AttachCurrentThreadAsDaemon(JavaVM* vm, void** penv, void* args);`

Same semantics as `AttachCurrentThread`, but the newly-created `java.lang.Thread` instance is a *daemon*.

If the thread has already been attached via either `AttachCurrentThread` or `AttachCurrentThreadAsDaemon`, this routine simply sets the value pointed to by `penv` to the `JNIEnv` of the current thread. In this case neither `AttachCurrentThread` nor this routine have any effect on the *daemon* status of the thread.

LINKAGE:
Index 7 in the JavaVM interface function table.

PARAMETERS:

`vm`: the virtual machine instance to which the current thread will be attached.

`penv`: a pointer to the location in which the `JNIEnv` interface pointer for the current thread will be placed.

`args`: a pointer to a `JavaVMAttachArgs` structure.

RETURNS

Returns `JNI_OK` on success; returns a suitable JNI error code (a negative number) on failure.

EXCEPTIONS

None.

SINCE:
JDK/JRE 1.4

DetachCurrentThread
`jint DetachCurrentThread(JavaVM *vm);`

Detaches the current thread from a Java VM. All Java monitors held by this thread are released. All Java threads waiting for this thread to die are notified.

As of JDK/JRE 1.2 , the main thread can be detached from the VM.

LINKAGE:
Index 5 in the JavaVM interface function table.

PARAMETERS:
`vm`: the VM from which the current thread will be detached.

RETURNS:
Returns `JNI_OK` on success; returns a suitable JNI error code (a negative number) on failure.

GetEnv
`jint GetEnv(JavaVM *vm, void **env, jint version);`

LINKAGE:
Index 6 in the JavaVM interface function table.

PARAMETERS:
`vm`: The virtual machine instance from which the interface will be retrieved.
`env`: pointer to the location where the JNI interface pointer for the current thread will be placed.
`version`: The requested JNI version.

RETURNS:
If the current thread is not attached to the VM, sets `*env` to `NULL`, and returns `JNI_EDETACHED`. If the specified version is not supported, sets `*env` to `NULL`, and returns `JNI_EVERSION`. Otherwise, sets `*env` to the appropriate interface, and returns `JNI_OK`.

SINCE:
JDK/JRE 1.2