Chapter 5: The Invocation API

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. It covers the following topics:

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 19 VM initialization arguments */
JavaVMOption* options = new JavaVMOption[1];
options[0].optionString = "-Djava.class.path=/usr/lib/java";
vm_args.version = JNI_VERSION_19;
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*.

Note: Depending on the operating system, the primordial process thread may be subject to special handling that impacts its ability to function properly as a normal Java thread (such as having a limited stack size and being able to throw `StackOverflowError`). It is strongly recommended that the primordial thread is not used to load the Java VM, but that a new thread is created just for that purpose.

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, with the one exception that there is no Java caller when calling caller-sensitive methods. 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.

Terminating the VM

The `JNI_DestroyJavaVM()` function terminates a Java VM.

The VM waits until the current thread is the only non-daemon user thread before it actually terminates. 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 terminated, the burden of releasing system resources held by arbitrary threads is on the programmer.

Library and Version Management

A native library may be either dynamically linked or statically linked with the VM. The manner in which the library and VM image are combined is implementation dependent. A `System.loadLibrary` or equivalent API must succeed for a library to be considered loaded, this applies to both dynamically and even statically linked libraries.

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.

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 this 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.

Support for Statically Linked Libraries

The following rules apply to statically linked libraries, for the statically linked library given in these examples named 'L':

- A library 'L' whose image has been combined with the VM is defined as statically linked if and only if the library exports a function called `JNI_OnLoad_L`.
- If a statically linked library L exports a function called `JNI_OnLoad_L` and a function called `JNI_OnLoad`, the `JNI_OnLoad` function will be ignored.
- If a library L is statically linked, then upon the first invocation of `System.loadLibrary("L")` or equivalent API, a `JNI_OnLoad_L` function will be invoked with the same arguments and expected return value as specified for the `JNI_OnLoad function`.
- A library L that is statically linked will prohibit a library of the same name from being loaded dynamically.
- When the class loader containing a statically linked native library L is garbage collected, the VM will invoke the `JNI_OnUnload_L` function of the library if such a function is exported.
- If a statically linked library L exports a function called `JNI_OnUnload_L` and a function called `JNI_OnUnload`, the `JNI_OnUnload` function will be ignored.

The programmer can also call the JNI function `RegisterNatives()` to register the native methods associated with a class. The `RegisterNatives()` function is particularly useful with statically linked functions.

If dynamically linked library defines `JNI_OnLoad_L` and/or `JNI_OnUnload_L` functions, these functions will be ignored.

Library Lifecycle Function Hooks

To facilitate version control and resource management, JNI libraries may define *load* and *unload* function hooks. Naming of these of functions depends upon whether the library was dynamically or statically linked.

JNI_OnLoad

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

Optional function defined by dynamically linked libraries. The VM calls `JNI_OnLoad` when the native library is loaded (for example, through `System.loadLibrary`).

In order to make use of functions defined at a certain version of the JNI API, `JNI_OnLoad` must return a constant defining at least that version. For example, libraries wishing to use `AttachCurrentThreadAsDaemon` function introduced in JDK 1.4, need to return at least `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 VM will unload the library and act as if the library was never loaded.

LINKAGE:

Exported from dynamically linked native libraries that contain native method implementations.

PARAMETERS:

`vm`: a pointer to the current VM structure.

`reserved`: unused pointer.

RETURNS:

Return the required `JNI_VERSION` constant (see also `GetVersion`).

JNI_OnUnload

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

Optional function defined by dynamically linked libraries. 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.

LINKAGE:

Exported from dynamically linked native libraries that contain native method implementations.

PARAMETERS:

`vm`: a pointer to the current VM structure.

`reserved`: unused pointer.

JNI_OnLoad_L

```
jint JNI_Onload_<L>(JavaVM *vm, void *reserved);
```

Mandatory function that **must be defined by statically linked libraries** .

If a library, named 'L', is statically linked, then upon the first invocation of `System.loadLibrary("L")` or equivalent API, a `JNI_OnLoad_L` function will be invoked with the same arguments and expected return value as specified for the `JNI_OnLoad` function. `JNI_OnLoad_L` must return the JNI version needed by the native library. This version must be `JNI_VERSION_1_8` or later. If the VM does not recognize the version number returned by `JNI_OnLoad_L`, the VM will act as if the library was never loaded.

LINKAGE:

Exported from statically linked native libraries that contain native method implementations.

PARAMETERS:

`vm`: a pointer to the current VM structure.

`reserved`: unused pointer.

RETURNS:

Return the required `JNI_VERSION` constant (see also `GetVersion`). The minimum version returned being at least `JNI_VERSION_1_8`.

SINCE:

JDK/JRE 1.8

JNI_OnUnload_L

```
void JNI_OnUnload_<L>(JavaVM *vm, void *reserved);
```

Optional function defined by statically linked libraries. When the class loader containing a statically linked native library 'L' is garbage collected, the VM will invoke the `JNI_OnUnload_L` function of the library if such a function is exported.

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.

LINKAGE:

Exported from statically linked native libraries that contain native method implementations.

PARAMETERS:

`vm`: a pointer to the current VM structure.

`reserved`: unused pointer.

SINCE:

JDK/JRE 1.8

Informational Note:

The act of loading a native library is the complete process of making the library and its native entry points known and registered to the Java VM and runtime. Note that simply performing operating system level operations to load a native library, such as `dlopen` on a UNIX(R) system, does not fully accomplish this goal. A native function is normally called from the Java class loader to perform a call to the host operating system that will load the library into memory and return a handle to the native library. This handle will be stored and used in subsequent searches for native library entry points. The Java native class loader will complete the load process once the handle is successfully returned to register the library.

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, must not be `NULL`.

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.

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, must not be `NULL`.

`bufLen`: the length of the buffer.

`nVMs`: a pointer to an integer. May be a `NULL` value.

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.

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 `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` ignores 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:

Standard Options

optionString	meaning
-D<name>=<value>	Set a system property

optionString	meaning
-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.

The module related options, --add-reads, --add-exports, --add-opens, --add-modules, --limit-modules, --module-path, --patch-module, and --upgrade-module-path must be passed as option strings using their "option=value" format instead of their "option value" format. (Note the required = between "option" and "value".) For example, to export java.management/sun.management to ALL-UNNAMED pass option string "--add-exports=java.management/sun.management=ALL-UNNAMED".

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, must not be NULL.

p_env: pointer to the location where the JNI interface pointer for the main thread will be placed, must not be NULL.

vm_args: Java VM initialization arguments, must not be NULL.

RETURNS:

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

DestroyJavaVM

```
jint DestroyJavaVM(JavaVM *vm);
```

Terminates the operation of the JVM, making a best-effort attempt to release resources.

Any thread, whether attached or not, can invoke 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.

LINKAGE:

Index 3 in the JavaVM interface function table.

PARAMETERS:

`vm`: the Java VM that will be destroyed, must not be `NULL`.

RETURNS:

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

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, must not be `NULL`.

`p_env`: pointer to the location where the JNI interface pointer of the current thread will be placed, must not be `NULL`.

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

The second argument to `AttachCurrentThread` is always a pointer to `JNIEnv`. The third argument to `AttachCurrentThread` was reserved, and should be set to `NULL`.

You pass a pointer to the following structure to specify additional information:

```
typedef struct JavaVMAttachArgs {
    jint version;
    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, must not be `NULL`.

`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. May be a `NULL` value.

RETURNS

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

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.

The main thread can be detached from the VM.

Trying to detach a thread that is not attached is a no-op.

If an exception is pending when `DetachCurrentThread` is called, the VM may choose to report its existence.

LINKAGE:

Index 5 in the JavaVM interface function table.

PARAMETERS:

`vm`: the VM from which the current thread will be detached, must not be `NULL`.

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, must not be `NULL`.

`env`: pointer to the location where the JNI interface pointer for the current thread will be placed, must not be `NULL`.

`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`.
