Mr. Joel Kemp
Tech articles and poetry

Getting Started with JNI and C++ on OSX Lion

Posted on January 4, 2012 by Joel Kemp

This tutorial combines a few key ideas to getting started with a Hello World program using the Java Native Interface (JNI) on Mac OSX. JNI allows Java code to utilize C++ code (i.e., native code).

This technology is very important if you want to do any computationally intensive operations (games, video processing, audio processing, etc) in mobile (Android) apps or your everyday Java applications.

The important point to this article is that most tutorials only talk about compilation for Windows and Solaris machines — I will focus exclusively on OSX. Knowing these subtleties can save you a lot of time and headaches!

Creating the Java Class

Let's start by creating a very simple Hello World program!

In a file HelloWorld.java

```
class HelloWorld {
    private native void print();
    public static void main(String[] args) {
        new HelloWorld().print();
    }
    static {
        System.loadLibrary("HelloWorld");
    }
}
```

The above code details a simple Java class that has a very important native method print(). The fact that this method is native signifies that it should be implemented in C++. We'll get to that implementation in a bit.

The static section gets executed first, which expects to load a JNI shared or dynamic library known as HelloWorld.

Compiling the Java Class

Compiling the newly created HelloWorld class involves opening up the terminal and issuing the command:

```
javac HelloWorld.java
```

We use the *javac* command to generate the class file that is needed to generate the appropriate JNI C++ classes. You can't skip the generation of the class file!

To create the JNI C++ files, you can just type the following command into the terminal:

```
javah -jni HelloWorld
```

The above command will generate the following header file:

In HelloWorld.h

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

```
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    print
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_print
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

This file contains a lot of seemingly obscure boiler-plate specific to JNI. The part we care about is the definition of the print() function that was declared to be native in the HelloWorld.java file.

By convention, native functions will be named "Java_" followed by the class name "HelloWorld_" and the function name "print". Thus, we obtain the native function: Java_HelloWorld_print().

Let's write an implementation file to define the behavior of the Java_HelloWorld_print() function.

In HelloWorld.cpp

```
#include <jni.h>
#include <iostream>
#include "HelloWorld.h"
using namespace std;

JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *, jobject){
    cout << "Oh JNI, how cumbersome you are!\n";
    return;
}
```

The implementation of the native function simply prints out a literal string.

Compiling the C++ Code

To include the native code in the compilation of the Java program, we need to compile our native C++ code into a dynamic library. On OSX, JNI shared/dynamic libraries have the extension .jnilib. This is a huge difference from the extensions on Windows and Solaris machines: .dll and .so, respectively.

Before we get to the command, there's a big problem: your compiler (g++) will need to know the location of the jni.h library. You don't get this linking for free, so you have to specify the location of the header file within the compilation command.

The locations of the the jni.h library that need to be included are:

```
"-I/System/Library/Frameworks/JavaVM.framework/Versions/CurrentJDK/Headers"
"-
I/Developer/SDKs/MacOSX10.6.sdk/System/Library/Frameworks/JavaVM.framework/Versions/A/Headers"
```

Unfortunately, leaving off either of these results in the compiler error:
HelloWorld.cpp:1:17: error: jni.h: No such file or directory

In your terminal, issue the command (*all part of a single command*):

```
g++ "-I/System/Library/Frameworks/JavaVM.framework/Versions/CurrentJDK/Headers"
"-
I/Developer/SDKs/MacOSX10.6.sdk/System/Library/Frameworks/JavaVM.framework/Versions/A/Headers"
-c HelloWorld.cpp
```

Notice that we include the -c option to generate the object file for our native code: HelloWorld.o. We also include the implementation file for our native class: HelloWorld.cpp.

The final step for compiling the native source is to issue the command:

```
g++ -dynamiclib -o libhelloworld.jnilib HelloWorld.o
```

This command uses the dynamiclib option to specify that the compiler shouldn't produce a standard executable, but should produce a library. The -o option is used to name the library with the appropriate extension: libhelloworld.jnilib. We also include the object file generated in the previous step.

The result of this process is the compiled library libhelloworld.jnilib.

Compiling the Native Library and Java Program

The final piece to this painstaking puzzle is to connect all of the pieces! We're going to finally compile our Java program which will look for the jnilib library containing our compiled native code.

In your terminal, issue the following command:

```
java HelloWorld
```

This will compile and run our HelloWorld program and display the output:

```
Oh JNI, how cumbersome you are!
```

One final note

If you see the error:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no HelloWorld in java.library.path
at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1758)
at java.lang.Runtime.loadLibrary0(Runtime.java:823)
at java.lang.System.loadLibrary(System.java:1045)
at HelloWorld.(HelloWorld.java:7)
```

Then you've mistakenly used a different shared/dynamic library extension than the necessary .jnilib! This is the root of all evil in this exercise!

Happy coding!

This tutorial is based on the JNI tutorial provided by Sun.

Comments

11 comments

Saravanan Ganesan ·
Software Developer at CountryNet Software

Helpful article. It works for me all in the flat structure. Do you have article written for JNI and Maven integration build on Mac