A Recipe For Platform-Dependent Java Builds

Stuart Maclean

Applied Physics Laboratory
University of Washington
stuart@apl.uw.edu

May 2015

Outline

Local Java Artifacts Needing Help From C

FUSE4J	Java bindings to Fileystem In User Space, C library. Used to expose virtual machine disk contents on the host (VM powered off). Git-cloned, build process altered.
TSK4J	Java bindings to The SleuthKit, a digital forensics library in C. Can add e.g. Java graphics to disk traversals. Created Java side in-house.
Tupelo	Cybersecurity tool for efficient storage of whole disk drives. Need disk drive serial numbers. Created in-house.

For each, the Java side is Maven-oriented, the C side is Make-oriented. How to solve (work around??) the thorny issue of platform dependency? How to make these builds follow a pattern? Need a recipe.

Split-Language Builds and Maven

- Unlike Java's build-once-run-anywhere, any C parts of a split-language product result in that product being tied to its targeted platform.
- Want products to align with the Maven philosophy of modular artifacts promoting re-use and composition via dependency management.
- If we 'tag' our Maven artifacts with some platform identifier, e.g. `group:artifact-x86_64:version`, then we need an artifact per platform??
- Depending on such an artifact results in the dependent program being platform-dependent also, yuk!

Dependency Management Problem

If our JNI artifact is this:

```
<groupId>seajug</groupId>
<artifactId>myLib-x86</artifactId>
<version>1.0</version>
```

then using that library in a larger application results in this:

```
<groupId>seajug</groupId>
<artifactId>myApp-x86</artifactId>

<dependency>
 <groupId>seajug</groupId>
 <artifactId>myLib-x86</artifactId>
 <version>1.0</version>
</dependency>
```

Intro to JNI: Java Classes

Start with Java code, with 1+ methods declared native:

```
package seajug;

public class Adder {

 // Java cannot do integer addition, need the power of C
 static native int sum( int i, int j );

 static {
  // Load the C code into the JVM, can now call Adder.sum
  System.loadLibrary( "Adder" );
 }
}
```

Intro to JNI: Compile Java

Build the Java code:

```
$ javac -d bin src/seajug/Adder.java
$ tree src bin
src
`-- seajug
    `-- Adder.java
bin
`-- seajug
    `-- Adder.class
```

Intro to JNI: Produce C Headers

JDK tool javah produces a C header file:

```
$ javah -d c -classpath bin seajug.Adder
$ tree bin c
bin
‘-- seajug
    ‘-- Adder.class
c
‘-- seajug_Adder.h
```

Intro to JNI: C Header Inspection

Generated C header file shows the signature of the method(s) to define:

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

/* Header for class seajug_Adder */

/*
 * Class:     seajug_Adder
 * Method:    sum
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_seajug_Adder_sum
  (JNIEnv*, jclass, jint, jint);
```

Intro to JNI: C Functions

Implement the required method, in e.g. `Adder.c`:

```
#include "seajug_Adder.h"

JNIEXPORT jint JNICALL Java_seajug_Adder_sum
 (JNIEnv* env, jclass clazz, jint a, jint b) {
  return a+b;
}
```

Intro to JNI: Compile C

Compile and link the C code. Your JDK comes with C headers in support of JNI programming.

```
$ JAVA_HOME=/usr/lib/jvm/java7
$ gcc -c Adder.c -I $JAVA_HOME/include
$ gcc -shared *.o -o libAdder.so

$ cl -c Adder.c -I $JAVA_HOME\include
$ cl *.o -o Adder.dll
```

Intro to JNI: Run the Program

```
$ java -Djava.library.path=c -classpath bin seajug.Adder
```

If built on e.g. 64-bit Linux, will run *only* on 64-bit Linux.

Intro to JNI: Considerations

- Decide on a home for the Java and C source and compiled code.
- Have to ensure C shared library file accompanies Java jars for distribution.
- Java issues solved with Maven, what about C?
- Platform dependencies??

Been struggling with these issues in my own Java/C projects for years.

Then Along Came Snappy

- Snappy is a compression algorithm by Google, written in C++.
- Wanted to consume Snappy-compressed data in a Java program.
- Search net for 'Snappy + Maven'.
- https://github.com/xerial/snappy-java.
- Declares itself 'JNI-based', so a Java wrapper around C code.
- Maven artifact is `org.xerial.snappy:snappy-java:1.1.1`.

How can a split Java/JNI codebase appear to be a regular Maven artifact?

Snappy-Java Maven Artifact Contents

```
org/xerial/snappy/native/AIX/ppc64/libsnappyjava.a
org/xerial/snappy/native/Linux/arm/libsnappyjava.so
org/xerial/snappy/native/Linux/x86/libsnappyjava.so
org/xerial/snappy/native/Linux/x86_64/libsnappyjava.so
org/xerial/snappy/native/Mac/x86/libsnappyjava.jnilib
org/xerial/snappy/native/Mac/x86_64/libsnappyjava.jnilib
org/xerial/snappy/native/SunOS/sparc/libsnappyjava.so
org/xerial/snappy/native/SunOS/x86/libsnappyjava.so
org/xerial/snappy/native/SunOS/x86_64/libsnappyjava.so
org/xerial/snappy/native/Windows/x86/snappyjava.dll
org/xerial/snappy/native/Windows/x86_64/snappyjava.dll
```

Approximates a C shared library with a Java class.

The Genius of Snappy-Java

- Snappy-Java bundles code for *all* platforms it supports into a *single* Maven artifact jar.
- So what if it bloats the jar? Disk is cheaper than developer time.
- Snappy-Java treats compiled C code as Maven resources, so they are built just once per platform.
- Once built, check into SCM (e.g. git). Odd, but works.

Seeing this approach inspired me to generalize it for my own builds.

Snappy-Java Features

- As seen, contains native libraries for e.g. Windows/Mac/Linux.
- Snappy-Java loader class loads one of these libraries, according to system properties `os.name` and `os.arch`.
- Defines canonical values for OS name and platform, then maps actual `os.arch`, `os.arch` strings to these canonical values. Example: openjdk7 on Mac gives `os.arch` of 'universal', maps to X64 for resource lookup.
- Uses `java.io.tmpdir` for unpacking native resources from the jar, cleans up at program end. Native libraries can *only* be loaded from filesystem.

Recipe Step 1: Generalized Loader

Generalize Snappy-Java's native library loader to load *any* C code bundled in a Maven artifact with an expected path structure:

```
public class NativeLoader {
 static void load( String group, String artifact ) {
  File nativeLibFile = findNativeLibrary( group, artifact );
  System.load( nativeLibFile.getPath() );
 }
 static File findNativeLibrary(String group,String artifact){
  String nativeLibraryName = System.mapLibraryName(artifact);
  String info = getNativeLibFolderPathForCurrentOS();
  String nativeLibraryPath = group + "/native/" + info +
    "/" + nativeLibraryName;
  return extract( nativeLibraryPath );
 }
```

Native Lib Loader

`NativeLibLoader` loads shared library code from the main Java jar:

- Calls `System.mapLibrary()` to convert e.g. "foo" into "foo.dll", "libfoo.so".
- Interrogates JVM for `os.arch, os.name`.
- Constructs a resource path to include supplied group name, `os.arch, os.name` and system-mapped artifact.
- Extracts located resource to filesystem.
- Calls `System.load()` on extracted resource.

Native Lib Loader Usage

```
package somePackage;

class WithNativeMethods {

 static {
  NativeLoader.load( WithNativeMethods.class.getName(),
                     "artifact" );
 }
}
```

Recipe Step 2: Filesystem Layout

```
pom.xml
src/main/java/package/Classes.java
src/test/java/
src/main/native/*.c  // lucky
src/test/native/*.c
src/main/native/Linux/*.c
src/main/native/Linux/Makefile
src/main/native/Linux/x86_64/*.c // unlucky
src/main/native/Linux/x86_64/Makefile
src/main/native/Mac/x86/*.c
src/main/native/Mac/x86/Makefile
src/main/resources/package/native/Linux/x86_64/.so
src/main/resources/package/native/Mac/x86/.dylib
```

Recipe Step 3: Maven POM

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>native-maven-plugin</artifactId>
 <executions>
  <execution>
   <phase>compile</phase>
   <goals>
    <goal>javah</goal>
   </goals>
   <configuration>
     <javahClassNames>
       <javahClassName>
     edu.uw.apl.tupelo.model.PhysicalDisk
   </configuration>
   </execution>
  </executions>
```

Recipe Step 3: Maven POM

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
   <executions><execution>
     <phase>compile</phase>
     <goals><goal>exec</goal></goals>
     <configuration>
      <workingDirectory>
       src/main/native/${os.name}/${os.arch}
      </workingDirectory>
      <executable>make</executable>
      <environmentVariables>
       <PACKAGE>edu.uw.apl.tupelo.model</PACKAGE>
       <ARTIFACT>${project.artifactId}</ARTIFACT>
      </environmentVariables></configuration></execution>
    </executions></plugin>
```

Recipe Step 4: OS+Arch Makefile

```
ARCH = x86_64

NATIVEHOME = $(abspath ../../)

include ../Makefile
```

Recipe Step 4: OS Makefile

```
OS = Linux
BASEDIR = $(abspath $(NATIVEHOME)/../../..)
CFLAGS += -Wall -Werror -fPIC -std=c99
CPPFLAGS += -I$(BASEDIR)/target/native/javah
CPPFLAGS += -I$(JAVA_HOME)/include

MAINSRCS = $(shell cd $(NATIVEHOME)/$(OS) && ls *.c)
LIB = lib$(ARTIFACT).so

PACKAGEASPATH = $(subst .,/,$(PACKAGE))
TARGET_DIR = $(BASEDIR)/src/main/resources/
 $(PACKAGEASPATH)/native/$(OS)/$(ARCH)
```

Recipe Step 5: The Build

The pom places the javah, exec plugins inside a "native" profile, so only the power dev actually builds the C code. The Java devs don't need any C tools at all.

```
$ mvn -Pnative

$ git add src/main/resources/package/native/Linux/x86/*.so

$ mvn package

$ jar tf artifact.jar

// aligns with NativeLoader
package/native/Linux/x86/*.so
```

Summary, Next Steps

- Snappy-Java idea generalizable to a recipe for Java/JNI builds.
- Planning to upload native loader artifact and recipe to github.
- For another day: Maken, a wannabe Maven for C codebases and Make.