Djinni, I wish for easy integration of C++ in my Java Android app!

1 December 2015 | **Software Engineering** | Aaron Wyder
Reading time: 5 minutes

Have you ever had the pleasure of integrating native (i.e. C/C++) code into your Java application using JNI? Well, to use the word “pleasure” and “JNI” in one sentence is probably not a good idea, right? At least that is my own Impression.

Typical tasks in JNI binding code are:

- generating the interface headers from the Java methods declared as native using javah
- looking up Java methods by the exact signature
```
jniGetMethodID(aClazz, "printHelloWorldNTimes",
"(ILjava/lang/String;)Z")
```
- data type mapping to and from Java
- handling of local and global references
- thread handling (attaching to and detaching from a java thread)
- storing pointers to C++ object instances in Java code and passing them via JNI

Sounds like a lot of repetitive and error prone work. Would it not be nice to have all that dreadful JNI code generated for you? Actually there is a trend in mobile app development to write the common platform independent logic in native code which can be shared between Android and iOS. Therefore it would also be nice to generate Objective-C bindings as well.

In a current project I am exactly in this position. We have a native framework which needs to be integrated into an Android App. I started writing the JNI binding code by hand and even began to maintain relationships between Java and native objects by storing the memory addresses (stored as long) in my Java objects which were then passed through the JNI interface and resolved (using static_cast) to native C++ classes. It all worked quite well except that some small errors crept into the code now, which resulted in many app crashes. Furthermore the ratio of JNI binding code to production code was extremely high.

At this years droidcon in Berlin a talk about platform independent app development mentioned the tool “Djinni” [1] to generates binding code for Java and Objective-C which immediately caught my attention. Djinni is actively developed by Dropbox Inc. which are known to implement their core business logic in native code. Djinni is basically a command line tool which parses an interface description language (IDL) containing the interface declaration between native code and Java or Objective-C. It also allows complex data structures to be passed through the interface.

Show me code!

Let’s have a look at a simple example and consider an Android app to compute the Fibonacci sequence as fast as possible. In order to achieve the speed needed, a decision has been made to implement the business logic using native C++ code, which also makes it possible to use this highly complex algorithm on multiple platforms (this is a hypothetical example, Java is of comparable speed with regard to simple algorithms). There is also a requirement that the UI needs to be informed about the progress of the computation of a given sequence.

First, we need to create a fresh Android project using Android Studio and implement a simple activity which allows us to enter the number up to which the sequence needs to be computed. The activity contains a button to start the computation and outputs each part of the sequence (e.g. every 10 computed numbers) in a listview as they are computed by the algorithm. You can look at the sample code at [2]. Please note that the sample app uses the newest NDK capabilities which are only available in the 1.5 preview builds of Android Studio [3]. But it is worth it since in the preview versions it is finally possible to debug native code and have complete C/C++ code completion [4].

Next we define the interface from Java to C++ and vice versa. The corresponding Djinni input file (fibonacci.djinni) is pretty straightforward

```
# definition of the C++ interface to be called from Java
fibonacci_engine_djinni = interface +c {
  # factory method to create a fibonacci engine with a callback to
Java
  static create_with_callback(callback: fibonacci_callback_djinni):
fibonacci_engine_djinni;
  computeFibonacci(amount : i64) : i64;
}

# definition of the Java callback interface
fibonacci_callback_djinni = interface +j {
  reportProgress(fibonacciSequenceChunk : list<i64>);
}
```

This looks pretty simple, right? Now we let Djinni generate the code needed to make this magic work. See the shell script runDjinni.sh in the sample app on how to use the Djinni command line.

In this simple example we are only scratching the surface of what Djinni is capable of. Djinni can handle many common data types, records (i.e. complex objects), enums and of course lists, sets and maps of all those data types.

Djinni will generate the following classes from this in src/main/java/ch.fork.djinnisample/djinni_generated:

- **FibonacciEngineDjinni.java**: Java facade for the native Fibonacci-Engine
- **FibonacciCallbackDjinni.java**: abstract Java class which we can extend in order to receive callbacks from the native code
  in src/main/jni/djinni_generated/cpp:

- **fibonacci_engine_djinni.hpp**: abstract C++ class to extend with our implementation of the Fibonacci-Engine
- **fibonacci_callback_djinni.hpp**: C++ facade for the Java Fibonacci-Callback
  in src/main/jni/djinni_generated/jni:

- various **Native*.cpp** and **Native*.hpp** files containing the actual JNI code which connects Java and C++
  Furthermore the support_lib folder, containing setup and mapping code, needs to be copied manually from the Djinni source tree.

C++ Implementation
The actual implementation of the Fibonacci engine is in the C++ class FibonacciEngine which inherits from FibonacciEngineDjinni. I artificially slow the computation down by adding some sleep calls during the computation in order to make the calculation steps visible in the UI.

```
# Headerfile FibonacciEnding.hpp omitted for brevity

std::shared_ptr<FibonacciEngineDjinni>
FibonacciEngineDjinni::create_with_callback(
  const std::shared_ptr<FibonacciCallbackDjinni> &callback)
{
  return std::make_shared<FibonacciEngine>(callback);
}

FibonacciEngine::FibonacciEngine(const
std::shared_ptr<FibonacciCallbackDjinni> &callback)
{
    this->m_callback = callback;
}

int64_t FibonacciEngine::computeFibonacci(int64_t amount)
{
  int64_t input(0), a(0), b(1), total(1);
  std::vector<int64_t>
  for (int64_t i = 0; i <= amount; i++) {
    total = a + b;
    a = b;
    b = total;
    chunk.push_back(total);
    if (chunk.size() == 10) {
      m_callback->reportProgress(chunk);
      chunk.clear();
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(100));


  }
  m_callback->reportProgress(chunk);
  return total;
}
```

Accessing the native components from Java
All code accessing the native layer is in the class FibonacciCalculator.

After loading the native library with

```
static {
  System.loadLibrary("fibonacci-jni");
}
```

we can now create an instance of our FibonacciEngine:

```
FibonacciEngineDjinni.createWithCallback(new FibonacciCallbackDjinni()
{
  @Override
  public void reportProgress(ArrayList<Long> fibonacciSequenceChunk) {
    // do something
  }
});
```

and invoke the computation with

```
fibonacciEngineDjinni.computeFibonacci(amount);
```

And that’s all there is to it! Nice, right?

The remaining code of the sample application is just Android-specific and should be self-explanatory.

Conclusion

Migrating from the custom-built solution in my current real world project over to Djinni was a substantial initial effort as the API of the native code is quite large. However it was well worth it. The custom-built solution was just not maintainable as the API of the native framework got more and more complex. As I am not an iOS developer and we are not using the iOS code generation in this Project, I cannot say anything about the quality of the iOS generator. So far I have had no run time problems caused by Djinni. An additional bonus was that the somehow hidden structure of the API suddenly became apparent when we formulated it in the Djinni IDL, which is also of documentary value for the Project.

To migrate a pre-existing API, I would strongly suggest implementing the Djinni interface as a wrapper around the original API. Since you have to inherit from the classes generated by Djinni and convert to and from its type system, this would lead otherwise to a very strong coupling to Djinni.

References

[1] Djinni project site: https://github.com/dropbox/djinni

[2] Sample code: https://github.com/forkch/DjinniSample

[3] Android Studio 1.5 preview:
http://tools.android.com/recent/androidstudio15preview2available
(or http://tools.android.com/recent for recent news about Android Studio)

[4] Android NDK Preview: http://tools.android.com/tech-docs/android-ndk-preview