Wrapping Java with C++ objects - JNI OOP

Yochai Timmer, 1 Jun 2015 CPOL

A way to avoid JNI's reflection oriented programming. Wrapping Java with C++

- Download JNIWrapper.zip - 11.7 KB

Introduction

JNI was introduced as an interface for allowing Java code tunning on a JVM virtual machine to interact with Native code (C++). The JNI framework's implementation for interacting with Java code is strongly reflection oriented. This means that you get a "pointer" to an object that represents the Virtual Machine (JavaVM*), or a pointer to the "Environment" (JNIEnv*). After that you need to "ask" the virtual machine or the environment for function handles or method handles on which you perform operations.

This gets very annoying very fast. The amount of repetitive copy-pasted code gets very dificult to handle, especially because the use of strings in the reflection. This usually prevents programmers to scale-up the code and use C++ and JNI to call java methods freely.

JNI C++ wrappers

We need to bring back OOP to JNI. Represent the java objects as Objects. This way we can regain all the advantages of object oriented programming, including the reusability and code maintainability which we lost in the reflection-based JNI.

The ability to have a C++ style wrapper to handle Java objects can make JNI production much easier.

Proper interaction with java classes should be easy to read and intuitive to C++ developers.

Hide Copy Code

```
CPoint3D myPoint(env);

myPoint.x = 5;
myPoint.y = 4;
mypoint.z = 3;
```

Just a reminder of the JNI method of doing stuff:

Hide Copy Code

```
jclass classObj =  env->FindClass("javax/vecmath/Point3d");
jmethodID classConstructor = env->GetMethodID(classObj, "<init>", "()V");
jobject myPoint = env->NewObject(classObj , classConstructor);

jfieldID fidx = env->GetFieldID(classObj , "x", "I");
env->SetIntField(myPoint , fidx , 5);

jfieldID fidy = env->GetFieldID(classObj , "y", "I");
env->SetIntField(myPoint , fidy , 4);

jfieldID fidz = env->GetFieldID(classObj , "z", "I");
env->SetIntField(myPoint , fidz , 3);
```

Creating a C++ wrapper

To create a C++ class wrapper for a Java object, just inherit from `CJavaObject`. Define all the class members and function members as `CJavaMember` objects.

Static members and methods can be defined as CJavaStaticMember, and initialize it with `mClassTypeReflector`instead of this.

Hide Shrink Copy Code

```
class CPoint3D : public CJavaObject
{
public:

    CPoint3D(JNIEnv* env, jclass iClassLoader = NULL) :
        CJavaObject(env, "javax/vecmath/Point3d", iClassLoader),
        x(this, "x", "I"),
        y(this, "y", "I"),
        z(this, "z", "I"),
        distanceFunc(this, "distance", "(Ljavax/vecmath/Point3d;)D")
    {    }

    CPoint3D(JNIEnv* env, jobject obj) :
        CJavaObject(env, obj),
        x(this, "x", "I"),
        y(this, "y", "I"),
        z(this, "z", "I")
        distanceFunc(this, "distance", "(Ljavax/vecmath/Point3d;)D")
    {    }

    CJavaMember x;
    CJavaMember y;
    CJavaMember z;

    double distance(CPoint3D  p1)
```

```
    {
       double res;
       distance(&res, (jobject)p1);
       return res;
    }

private:
    CJavaMember distanceFunc;
};
```

A C++ Java class wrapper will either create a java object when constructed, or wrap an existing object instance if it's passed in the `CJavaObject` constructor.

Using an anonymous object

CJavaObject can anonymously access object members.
You can just instantiate a CJavaObject with a java object (or create a new one by naming the type).
To access the object member fields or object member methods use the overloaded operator[] with the field or method name and type.
Primitive member fields don't really need the type, it can be infered by type of the rvalue.

Hide Copy Code

```
CJavaObject myPoint(env, "javax/vecmath/Point3d");
myPoint["x"] = 1;
myPoint["y"] = 2;
myPoint["z"] = 3;

double res;
myPoint["distance"]["(Ljavax/vecmath/Point3d;)D"](&res, (jobject)myPoint);

//or:
myPoint["distance", "(Ljavax/vecmath/Point3d;)D"](&res, (jobject)myPoint);
```

Implementation

Download the source header files. Include the JniWrapper.h file wherever you need the wrappers.
The wrappers use C++11 features like variadic templates, make sure you add the right compilation flags for that.

Considerations

1. DeleteLocalRef - When constructing a Java object in JNI, you generally need to call DeleteLocalRef. I haven't added that to the destructor (like in the RAII idiom). This is because it is rarely really needed (unless it's a callback native thread). In a normal JNI call from Java, the local refrerences are stored on the stack and therefor removed when you go back to the Java layer. And besides that, it can cause problems if the object is the return

value of a JNI function.

2. Class loader - The CJavaObject class can receive a class loader object so you can create new Java objects with a different loader context. (For example: if you're using a native callback function on a newely attached thread with an empty class loader). Here's what google have to say about caching a class loader: http://developer.android.com/training/articles/perf-jni.html#native_libraries

3. Error checking - The goal is to wrap Java classes statically. So, hopefully once you got it running once properly, with the right names JNI names etc, the wrapper will always work for that Java class. So I don't believe in adding redundant failure checking in the wrapper code itself.

License

About the Author

Yochai Timmer

Software Developer (Senior)
Israel
http://stackoverflow.com/users/536086/yochai-timmer

Follow on LinkedIn

Private members of javaObject might refer to discarded objects **cth027** **2-Jun-15 2:41**
new

I like your `CJavaObject` : it makes it very comfortable to get rid of a lot of repetitive code. I think however that there is a nasty bug:
If you'd create a local `CJavaObject` inside of a native function called by Java, everything is fine. But if you'd keep this object in a static variable or on the free store, there could be some issue: any references that you get from JNI including references to classes and methods are considered to be local references. As soon as you return to Java, their reference count is decreased. They could hence become a victim of the garbage collector.
Unfortunately, you store these references in the private member variables of your C++ object at construction. Later invocation could perfectly work. But it might also fail if the class or the object would be garbage collected in the meantime. I think the probabilities are low, but if this would occur, it would have extremely nasty consequences.
I'd therefore strongly suggest to use `NewGlobalRef()` in the constructor to make sure the object and its class reference are kept alive, and `DeleteGlobalRef()` in the destructor. Note that the documentation indicates a`jobject` as parameter for the NewGlobalRef(), but it works also for `jclass`.

Yochai Timmer
For the same reason that I avoided DeleteLocalRef, I thought it would be safer to keep the objects local. Holding global references by default would be dangerous.Besides, making a GlobalRef is much more "expensive" and we

wouldn't want that with local variables that aren't meant for "global" use. In this case it would be just an extra call to the virtual machine. So creating a global object should be something you specify, not default.
In JniWrapper.h you can find CJavaGlobalRef that can be used to store global references. That is a RAII object that insures DeleteGlobalRef in the end.

cth027
That's a valid point ! Thanks ! I didn't notice CJavaGlobalRef.