JNI Interface

Friday, November 5, 2021 9:45 AM

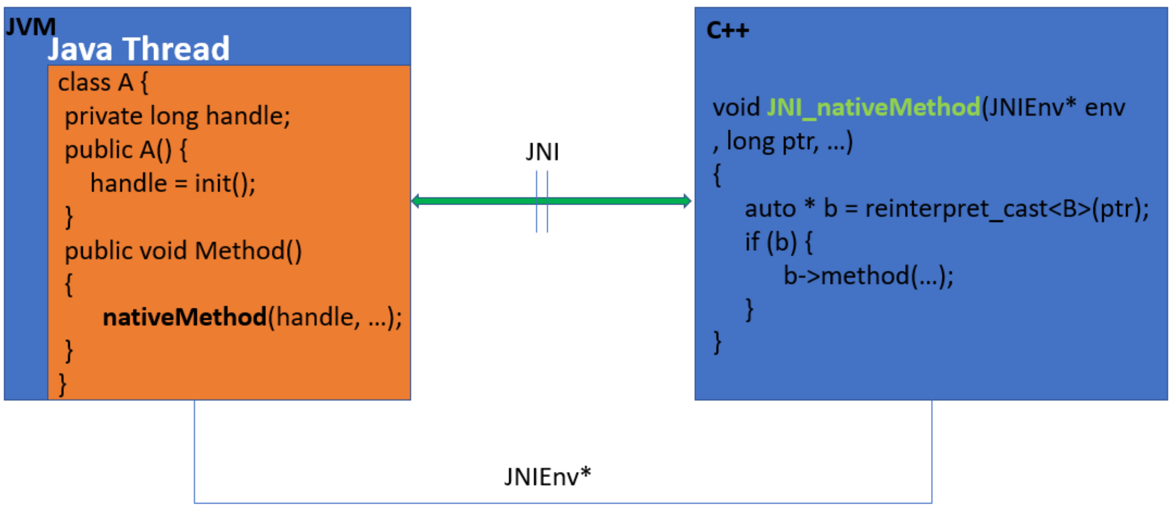
Author: Damir Ljubic

JNI (Java Native Interface) is used to marshal the calls (and data) between Java and native - C++ components.
This can be done **in both directions**:

1) Calling native (C++) implementation from Java method

For this use-case, the following should be considered:

- The Java method, as a wrapper around the calling native implementation, already runs in JVM context.
  No extra action is required.
- The JNI interface (Objective C) will be usually generated using some external tool (like *javah,* or swig)
- The first argument in each JNI call is *JNIEnv** - a pointer to the JNI functions table.
  It serves as a reference to the various getters/setters to manipulate with the custom Java types, in order to reach their accessible fields.
- In case that native implementation is encapsulated within the C++ user-defined type, as it usually is, at Java side we need to store the reference to the native implementation.
  We will usually store the raw pointer into *long* type member variable, and then pass it by to the each "*native*" call



1. **Raw pointer to native (C++) implementation**

```
<java>
     class A {
        private long handle;
        public A(){
            handle = init(); // store the reference to the C++ implementation
        }
        ...
        void method1(String s) {
           native_method1(handle, s);
        }

        private native long init();
        private native void native_method1(long ptr, String s);
      };

 <c++> Generated JNI interface: c-callbacks
      extern "C" JNIEXPORT jlong   JNICALL jni_init(JNIEnv* env, jobject obj)
      {
        cached_java_obj=env->NewGlobalRef(obj); // store the reference to the Java object
        auto * receiver = new Receiver(...); // raw pointer to the native implementation
        return reinterpret_cast<jlong>(receiver);
      }

      void jni_native_method1(JNIEnv* env, jobject obj, long ptr, jstring s)
      {
          auto * receiver = reinterpret_cast<Receiver*>(ptr);
          if (receiver) receiver->doSomething(…);
      }
```

At the same time, we can cache the Java object, for referencing it's methods (static and non-static) from native side of JNI interface.
This will be explained in next section.
The matching deinit() needs to be called, to release the memory on the heap

2. **std::shared_ptr<> to native (C++) implementation**

The approach with shared pointer, as a lifecycle manager over underlying pointer, with additional advantage: you don't need to store at Java side the reference of it, nor it should be addressed in each and every native method call.

```
<java>
private native void init(); // just initialize the static std::shared_ptr
private native void method1(String s); // no need for the reference to the raw non-static pointer on the heap

<c++>

static std::shared_ptr<Receiver> s_receiver = nullptr;

void jni_init(...)
{
    s_receiver = std::make_shared<Receiver>(...);
}

void jni_native_method1(JNIEnv* env, jobject object, jstring s)
{
    if (s_receiver) s_receiver->doSomething(…);
}
```

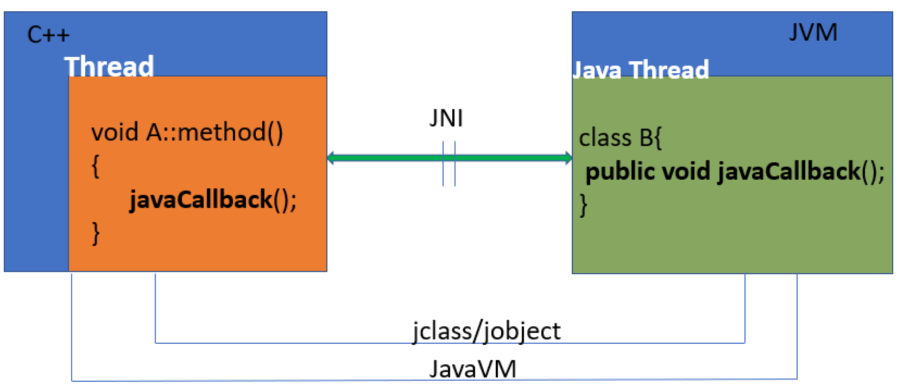
2) Calling Java callbacks from native (C++) side

This is much more interesting - demanding direction of calls.
The differences, compare with previous use-case:

- Native thread needs to be "attached" to JVM in order to be able to call java callback over *JNIEnv**.
  Attaching native thread to JVM (***AttachCurrentThread***) would effectively create a new Java thread at JVM side, in which context the Java callback will be actually invoked.
  Having that in mind, repeatedly attaching/detaching the very same native thread is expensive

  "*Attaching a natively-created thread causes a java.lang.Thread object to be constructed and added to the "main" ThreadGroup, making it visible to the debugger. Calling AttachCurrentThread() on an already-attached thread is a no-op*."
  https://developer.android.com/training/articles/perf-jni

- To retrieve the *JNIEnv** from native side, the reference to the JVM needs to be cached
- To call the static java callback from the native side, the *jclass* needs to be cached
  In case that we need to call the java non-static method, *jobject* needs to be cached
  (**@see Appendix A for a real-code example**)



*If you attach a native thread with ***AttachCurrentThread***, the code you are running **will never automatically free local references until the thread detaches**.
Any local references you create will have to be deleted manually.
In general, any native code that creates local references in a loop probably needs to do some manual deletion.
https://developer.android.com/training/articles/perf-jni#java

Implementation details

https://developer.android.com/training/articles/perf-jni
"The JNIEnv is used for **thread-local storage**. For this reason, **you cannot share a JNIEnv between threads**."

In order to fulfil this requirement, we can use one of these two approaches

 # Scoped wrapper

The implementation is based on RAII idiom and having the TLS - per-thread unique storage class.
The *scoped_env* wrapper around the JNIEnv* will attach in constructor the calling thread - only if this is not already attached, and release (detach) it at the point when destructor is called.

We will use the helper method, as a gateway for calling the Java callbacks, using ***thread_local*** as a thread specific storage variable, for holding the scoped_env, **which will be instantiated for each new (attached) thread**.

As with static function scope variables, starting with C++11, it's **thread-safe instantiation**

```
template <typename Callback, typename...Args>
auto ivoke_java_cbk(JavaVM * jvm, Callback callback, Args&&...args)
{
    // Implicit static, with thread lifetime duration
    thread_local utils::jni::scoped_env env {jvm, JNI_VERSION_1_6};

    return callback(env.get(), std::forward<Args>(args)...);
}
```

Be aware, since this is function template, for a different template arguments, the compiler will create different versions of a function, that will cause thread_local storage reinitializations: having multiple scoped_env per thread
<Compiler Explorer>: https://godbolt.org/z/x7rde85Yz

This could be acceptable, since constructing *scoped_env* on an attached thread (as well destructing on a detached thread) is relatively cheap (there is still guarding block, to ensure thread-safe initialization).
How the *callback* invocation itself can be simplified - using some generic code
(@see the **# Appendix A section**)

The implementation of *scoped_env* class is more than trivial

```
// header file
namespace utils::jni
{
     class scoped_env final
     {
         public:
             using pointer_type = JNIEnv *;

             explicit scoped_env(JavaVM * jvm, int version) noexcept;
             ~scoped_env();

             // Interface that allows the wrapper class to be used in the same manner as JNIEnv*

             pointer_type get() const { return m_env; }
             pointer_type operator->() const { return m_env; }

             operator JNIEnv*() const { return m_env; }
             explicit operator bool() const { return m_env != nullptr; }

         private:
             void attach();
             void detach();

         private:
             JavaVM * m_jvm;
             int m_version;
             JNIEnv* m_env = nullptr;
             bool m_attached = false;
             std::thread::id m_id;
     };
}

// source file

using namespace utils::jni;

scoped_env::scoped_env(JavaVM * jvm, int version) noexcept
        : m_jvm(jvm)
        , m_version(version)
        , m_id()
{
    attach();
}

scoped_env::~scoped_env()
{
    detach();
}

void scoped_env::attach()
{
    if (m_jvm == nullptr) return;

    if (const auto status = m_jvm->GetEnv(reinterpret_cast<void**>(&m_env), m_version); status == JNI_EDETACHED)
    {
        if (const auto result = m_jvm->AttachCurrentThread(&m_env, nullptr); result == JNI_OK)
        {
            m_attached = true;
            m_id = std::this_thread::get_id();
        }
    }
}

void scoped_env::detach()
{
   if (m_attached)
   {
```

```
        if (const auto id = std::this_thread::get_id(); m_id == id)
        {
            if (m_jvm != nullptr)
            {
                m_jvm->DetachCurrentThread();
                m_attached = false;
            }
        }
    }
}
```

 # Per-thread detacher

This approach enables better control over the thread local storage, avoiding the unnecessary reinitialization as result of using the generic gateway as entry point for handling all java callbacks invocations, and it's more in line, with the requirement that we try to satisfy (correctness).
This is accomplished having ***thread_detacher*** helper class, that will be "injected" into the stack of a newly attached thread.

```
class thread_detacher final
{
    public:
        explicit thread_detacher(JavaVM* jvm) noexcept
        : m_jvm(jvm)
        {}

        ~thread_detacher()
        {
            // Detach the current C++ thread from the JVM
            if (m_jvm) { m_jvm->DetachCurrentThread(); }
        }

        // Enable move operations as well (since custom destructor is provided)

        thread_detacher(thread_detacher&& other) = default;
        thread_detacher& operator=(thread_detacher&& other) = default;



    private:
        JavaVM* m_jvm;
};
```

Each attached thread will have its own instance of "detacher" that should be destroyed along with the thread, after thread is joined.
"Detacher" belongs to attached thread (not the class), and will be destroyed (calling destructor) at the point when thread is joined, as part of the cleaning threads stack process.
This will trigger detaching the calling thread from JVM - the matching Java thread will eventually terminate, and resources garbage-collected.

```
// header file

class thread_env
{

    inline static thread_local std::unique_ptr<thread_detacher> detacher;

   public:
    using pointer_type = JNIEnv*;


    /**
     * C-tor
     *
     * @param jvm The reference to the JVM (global cache variable)
     * @param version The JNI version
     */
    explicit thread_env(JavaVM* jvm, int version) noexcept;

   /**
     * Returns the JNIEnv reference, and attach the current native
     * thread to the JVM (if not already) - initiate per-thread instance of
     * the detacher: that will be called after thread stack is released,
     * detaching the native thread - releasing the matching Java thread
     *
     * @return JNIEnv reference
     */
    pointer_type get() const;
    operator pointer_type() const { return get(); }

    private:
        JavaVM* m_jvm;
        int m_version;
};

// source file

thread_env::thread_env(JavaVM* jvm, int version) noexcept
        : m_jvm(jvm)
```

```
        , m_version(version)
{}

thread_env::pointer_type thread_env::get() const
{
    if (m_jvm == nullptr) return nullptr;

    pointer_type env = nullptr;

    /*
     * Only if the calling thread is not attached - attach it once,
     * and initialize the per-thread "detacher".
     */
    if (const auto status = m_jvm->GetEnv(reinterpret_cast<void**>(&env), m_version); status == JNI_EDETACHED)
    {
        if (const auto result = m_jvm->AttachCurrentThread(&env, nullptr); result == JNI_OK)
        {
            detacher.reset(new (std::nothrow) thread_detacher{m_jvm});
        }
        else
        {
            env = nullptr;
        }
    }

    return env;
}
```

<Compiler explorer>: https://godbolt.org/z/GzadPcxGW

Appendix A

Helper methods, for calling the Java methods (static as well non-static) from C++ side, based on the cached global references (*jclass*/*jobjec*t) to the enclosing Java class.

```
namespace utils::jni
{
        /*
         * Helper methods.
         *
         * JNI interface for calling the Java methods
         * from native (C++) code
         */


        // Signature of JNI getters/setters methods

        template <typename R>
        using jni_non_static_method_t = R (JNIEnv::*)(jobject, jmethodID, ...);

        template <typename R>
        using jni_static_method_t = R (JNIEnv::*)(jclass, jmethodID, ...);

        template <typename R, typename... Args>
        R jni_non_static_method_call(
                jni_non_static_method_t<R> method,
                JNIEnv* env,
                jclass cls,
                jobject obj,
                const std::string& name,
                const std::string& signature,
                Args&&... args)
        {
           // Check input arguments

           if (env == nullptr) throw std::invalid_argument("<JNI> Invalid env reference!");
           if (cls == nullptr) throw std::invalid_argument("<JNI> Invalid class reference!");
           if (obj == nullptr) throw std::invalid_argument("<JNI> Invalid object reference!");

           R result;
           if constexpr (std::is_pointer<R>::value) result = nullptr;

           /*
            * Call the java method
            *
            */

           auto* id = env->GetMethodID(cls, name.c_str(), signature.c_str());
           if (id == nullptr) goto exit;

           result = (env->*method)(obj, id, std::forward<Args>(args)...); // @note: JNI APIs (Objective C) accept the value types

        exit:

           if (env->ExceptionCheck())
```

```
    {
        env->ExceptionDescribe();
        env->ExceptionClear();  // stop propagating exception
    }

    return result;
}

template <typename R, typename... Args>
R jni_static_method_call(
        jni_static_method_t<R> method,
        JNIEnv* env,
        jclass cls,
        const std::string& name,
        const std::string& signature,
        Args&&... args)
{
    // Check input arguments

    if (env == nullptr) throw std::invalid_argument("<JNI> Invalid env reference!");
    if (cls == nullptr) throw std::invalid_argument("<JNI> Invalid class reference!");

    /*
     * Call the java method
     *
     */
    R result;
    if constexpr (std::is_pointer<R>::value) result = nullptr;

    auto* id = env->GetStaticMethodID(cls, name.c_str(), signature.c_str());
    if (id == nullptr) goto exit;

    result = (env->*method)(id, std::forward<Args>(args)...);

exit:

    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();  // stop propagating exception
    }

    return result;
}

/*
 * Calling the arbitrary java non-static void method
 */
template <typename... Args>
void jni_non_static_method_void_call(
        jni_non_static_method_t<void> method,
        JNIEnv* env,
        jclass cls,
        jobject obj,
        const std::string& name,
        const std::string& signature,
        Args&&... args)
{
    // Check the input arguments

    if (env == nullptr) throw std::invalid_argument("<JNI> Invalid env reference!");
    if (cls == nullptr) throw std::invalid_argument("<JNI> Invalid class reference!");
    if (obj == nullptr) throw std::invalid_argument("<JNI> Invalid object reference!");

    /*
     * Call the java method
     *
     */
    auto* id = env->GetMethodID(cls, name.c_str(), signature.c_str());
    if (id == nullptr) goto exit;

    (env->*method)(obj, id, std::forward<Args>(args)...);

exit:
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();  // stop propagating exception
    }
}

template <typename... Args>
void jni_static_method_void_call(
        jni_static_method_t<void> method,
        JNIEnv* env,
        jclass cls,
        const std::string& name,
        const std::string& signature,
        Args&&... args)
{
    // Check input arguments
```

```
    if (env == nullptr) throw std::invalid_argument("<JNI> Invalid env reference!");
    if (cls == nullptr) throw std::invalid_argument("<JNI> Invalid class reference!");

    /*
     * Call the java method
     *
     */
    auto* id = env->GetStaticMethodID(cls, name.c_str(), signature.c_str());
    if (id == nullptr) goto exit;

    (env->*method)(cls, id, std::forward<Args>(args)...);

exit:
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();  // stop propagating exception
    }
}


inline static jbyteArray toByteArray(JNIEnv* env, const std::vector<std::uint8_t>& data)
{
    if (env == nullptr) throw std::invalid_argument("<JNI> Invalid env reference!");

    const auto size = static_cast<jsize>(data.size());
    auto* jdata = env->NewByteArray(size);
    if (jdata == nullptr) goto exit;

    env->SetByteArrayRegion(jdata, 0, size, reinterpret_cast<const jbyte*>(data.data()));

exit:

    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();  // stop propagating exception
    }

    return jdata;
}

inline static std::vector<std::uint8_t> fromByteArray(JNIEnv* env, jbyteArray data)
{
    if (env == nullptr) throw std::invalid_argument("<JNI> Invalid env reference!");

    const auto size = env->GetArrayLength(data);
    if (size == 0) return {};

    std::vector<std::uint8_t> out(size, 0);
    env->GetByteArrayRegion(data, 0, size, reinterpret_cast<jbyte*>(out.data()));

    return out;
}

/**
 * Converting the Java array into C++ collection
 * Constraint: Collection::emplace_back
 *
 * @param env JNI functions table reference
 * @param in  The Java array
 * @param map Transformation function: f: jobject->C++ matching object
 * @return    The resulting C++ collection
 */
template <class Collection, class Func>
auto convJNIArray(JNIEnv* env, jobjectArray in, Func map)
{
    if (env == nullptr) throw std::invalid_argument("<JNI> Invalid env reference!");

    Collection out;
    auto size = env->GetArrayLength(in);
    out.reserve(static_cast<std::size_t>(size));

    for (decltype(size) i = 0; i < size; ++i)
    {
        auto* el = env->GetObjectArrayElement(in, i);
        if (el == nullptr) continue;

        out.emplace_back(std::move(map(el)));

        env->DeleteLocalRef(el);
    }

    return out;
}

static inline jstring convertString(JNIEnv* env, const std::string& s)
{
```

```
        if (env == nullptr) throw std::runtime_error("Invalid JNIEnv reference!");

        auto js = env->NewStringUTF(s.c_str());
        if (js == nullptr) goto exit;

    exit:

        if (env->ExceptionCheck())
        {
            env->ExceptionDescribe();
            env->ExceptionClear();  // stop propagating exception
        }

        return js;
    }
}  // namespace utils::jni
```