# JEP 389: Foreign Linker API (Incubator)

| | |
|---|---|
| *Owner* | Maurizio Cimadamore |
| *Type* | Feature |
| *Scope* | JDK |
| *Status* | Closed / Delivered |
| *Release* | 16 |
| *Component* | core-libs |
| *Discussion* | panama dash dev at openjdk dot java dot net |
| *Effort* | L |
| *Duration* | L |
| *Relates to* | JEP 393: Foreign-Memory Access API (Third Incubator)<br>JEP 412: Foreign Function & Memory API (Incubator) |
| *Reviewed by* | Brian Goetz, Jorn Vernee, Paul Sandoz |
| *Created* | 2020/07/20 11:19 |
| *Updated* | 2022/03/02 17:08 |
| *Issue* | 8249755 |

## Summary

Introduce an API that offers statically-typed, pure-Java access to native code. This API, together with the Foreign-Memory API (JEP 393), will considerably simplify the otherwise error-prone process of binding to a native library.

## History

The Foreign-Memory Access API, which provides the foundations for this JEP, was first proposed by JEP 370 and targeted to Java 14 in late 2019 as an incubating API, and then subsequently refreshed by JEP 383 and JEP 393, which were targeted to Java 15 and 16, respectively. Together, the Foreign-Memory Access API and the Foreign Linker API constitute key deliverables of Project Panama.

## Goals

- *Ease of use:* Replace JNI with a superior pure-Java development model.
- *C support:* The initial scope of this effort aims at providing high quality, fully optimized interoperability with C libraries, on x64 and AArch64 platforms.
- *Generality:* The Foreign Linker API and implementation should be flexible enough to, over time, accommodate support for other platforms (e.g., 32-bit x86) and foreign functions written in languages other than C (e.g. C++, Fortran).
- *Performance:* The Foreign Linker API should provide performance that is comparable to, or better than, JNI.

## Non-Goals

It is not a goal to:

- Drop, re-implement, or improve JNI,
- Provide a tool to mechanically generate Java code from native-code header files, or
- Change or improve the way in which Java applications interacting with native libraries are packaged and deployed (e.g., multi-platform JAR files).

## Motivation

Java has supported native method calls via the Java Native Interface (JNI) since Java 1.1, but this path has always been hard and brittle. Wrapping a native function with JNI requires developing multiple artifacts: a Java API, a C header file, and a C implementation. Even with tooling help, Java developers must work across multiple toolchains to keep multiple platform-dependent artifacts in sync. This is hard enough with stable APIs, but when trying to track APIs in progress, it is a significant maintenance burden to update all of these artifacts each time the API evolves. Finally, JNI is largely about code, but code always exchanges data, and JNI offers little help in accessing native data. For this reason, developers often resort to workarounds (such as direct buffers or `sun.misc.Unsafe`) which make the application code harder to maintain or even less safe .

Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
ThreeTen
Tiered Attribution
Tsan
Type Annotations
XRender Pipeline
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

Over the years, numerous frameworks have emerged to fill the gaps left by JNI, including JNA, JNR and JavaCPP. JNA and JNR generate wrappers dynamically from a user-defined interface declaration; JavaCPP generates wrappers statically driven by annotations on JNI method declarations. While these frameworks are often a marked improvement over the JNI experience, the situation is still less than ideal, especially when compared with languages which offer first-class native interoperation. For instance, Python's ctypes package can dynamically wrap native functions without any glue code. Other languages, such as Rust, provide tools which mechanically derive native wrappers from C/C++ header files.

Ultimately, Java developers should be able to (mostly) *just use* any native library that is deemed useful for a particular task — and we have seen how the status quo gets in the way of achieving that. This JEP rectifies this imbalance by introducing an efficient and supported API — the Foreign Linker API — which provides foreign-function support without the need for any intervening JNI glue code. It does this by exposing foreign functions as method handles which can be declared and invoked in pure Java code. This greatly simplifies the task of writing, building and distributing Java libraries and applications which depend upon foreign libraries. Moreover, the Foreign Linker API, together with the Foreign-Memory Access API, provides a solid and efficient foundation which third-party native interoperation frameworks — both present and future — can reliably build upon.

## Description

In this section we dive deeper into how native interoperation is achieved using the Foreign Linker API. The various abstractions described in this section will be provided as an incubator module named `jdk.incubator.foreign`, in a package of the same name, side-by-side with the existing Foreign Memory Access API.

### *Symbol lookups*

The first ingredient of any foreign-function support is a mechanism to look up symbols in native libraries. In traditional Java/JNI scenarios, this is done via the `System::loadLibrary` and `System::load` methods, which internally map into calls to `dlopen`. The Foreign Linker API provides a simple library-lookup abstraction via the `LibraryLookup` class (similar to a method-handle lookup), which provides capabilities to look up named symbols in a given native library. We can obtain a library lookup in three different ways:

- `LibraryLookup::ofDefault` — returns the library lookup which can *see* all the symbols that have been loaded with the VM.
- `LibraryLookup::ofPath` — creates a library lookup associated with the library found at the given absolute path.
- `LibraryLookup::ofLibrary` — creates a library lookup associated with the library with given name (this might require setting the `java.library.path` variable appropriately).

Once a lookup is obtained, a client can use it to retrieve handles to library symbols, either global variables or functions, using the `lookup(String)` method. This method returns a fresh `LibraryLookup.Symbol`, which is just a proxy for a memory address and a name.

For instance, the following code looks up the `clang_getClangVersion` function provided by the `clang` library:

```
LibraryLookup libclang = LibraryLookup.ofLibrary("clang");
LibraryLookup.Symbol clangVersion = libclang.lookup("clang_getClangVersion");
```

One crucial distinction between the library loading mechanism of the Foreign Linker API and that of JNI is that loaded JNI libraries are associated with a class loader. Furthermore, to preserve class loader integrity, the same JNI library cannot be loaded into more than one class loader. The foreign-function mechanism described here is more primitive: The Foreign Linker API allows clients to target native libraries directly, without any intervening JNI code. Crucially, Java objects are never passed to and from native code by the Foreign Linker API. Because of this, libraries loaded via `LibraryLookup` are not tied to any class loader and can be (re)loaded as many times as needed.

### *The C linker*

The `CLinker` interface is the foundation of the API's foreign function support.

```
interface CLinker {
    MethodHandle downcallHandle(LibraryLookup.Symbol func,
```

```
                                    MethodType type,
                                    FunctionDescriptor function);
    MemorySegment upcallStub(MethodHandle target,
                             FunctionDescriptor function);
}
```

This abstraction plays a dual role. First, for *downcalls* (e.g. calls from Java to native code), the `downcallHandle` method can be used to model native functions as plain `MethodHandle` objects. Second, for *upcalls* (e.g. calls from native back to Java code), the `upcallStub` method can be used to convert an existing `MethodHandle` (which might point to some Java method) into a `MemorySegment`, which can then be passed to a native function as a function pointer. Note that, while the `CLinker` abstraction is mostly focused on providing interoperation support for the C language, the concepts in this abstraction are general enough to be applicable, in the future, to other foreign languages.

Both `downcallHandle` and `upcallStub` take a `FunctionDescriptor` instance, which is an aggregate of memory layouts which is used to describe the signature of a foreign function in full. The `CLinker` interface defines many layout constants, one for each main C primitive type. These layouts can be combined using a `FunctionDescriptor` to describe the signature of a C function. For instance, we can model a C function taking a `char*` and returning a `long` with the following descriptor:

```
FunctionDescriptor func
    = FunctionDescriptor.of(CLinker.C_LONG, CLinker.C_POINTER);
```

The layouts in this example map to the layout appropriate to the underlying platform, so these layouts are platform dependent: `C_LONG` will, e.g., be a 32 bit value layout on Windows, but a 64-bit value on Linux. To target a specific platform, specific sets of platform-dependent layout constants are available (e.g., `CLinker.Win64.C_LONG`).

Layouts defined in the `CLinker` class are convenient, since they model the C types we want to work with. They also contain, via layout *attributes*, hidden pieces of information which the foreign linker uses in order to compute the calling sequence associated with a given function descriptor. For instance, the two C types `int` and `float` might share a similar memory layout (they are both 32-bit values), but are typically passed using different processor registers. The layout attributes attached to the C-specific layouts in the `CLinker` class ensure that arguments and return values are handled in the correct way.

Both `downcallHandle` and `upcallStub` also accept (either directly or indirectly) a `MethodType` instance. The method type describes the Java signatures that clients will use when interacting with the generated downcall handles or upcall stubs. The argument and return types in the `MethodType` instance are validated against the corresponding layouts. For instance, the linker runtime checks that the size of the Java carrier associated to a given argument/return value is equal to that of the corresponding layout. The mapping of primitive layouts to Java carriers can vary from one platform to another (e.g., `C_LONG` maps to `long` on Linux/x64, but to `int` on Windows), but pointer layouts (`C_POINTER`) are always associated with a `MemoryAddress` carrier and structs (whose layouts are defined by a `GroupLayout`) are always associated with a `MemorySegment` carrier.

***Downcalls***

Assume we want to call the following function defined in the standard C library:

```
size_t strlen(const char *s);
```

To do that, we have to:

- Lookup the `strlen` symbol,
- Describe the signature of the C function using the layouts in the `CLinker` class,
- Select the Java signature to *overlay* on the native function (this will be the signature that clients of the native method handle will interact with), and
- Create a downcall native method handle with the above information, using `CLinker::downcallHandle`.

Here's an example of how to do that:

```
MethodHandle strlen = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().lookup("strlen"),
        MethodType.methodType(long.class, MemoryAddress.class),
```

```
        FunctionDescriptor.of(C_LONG, C_POINTER)
    );
```

The `strlen` function is part of the standard C library, which is loaded with the VM, so we can just use the default lookup to look it up. The rest is pretty straightforward. The only tricky detail is how we model `size_t` — typically this type has the size of a pointer, so we can use `C_LONG` on Linux, but we would have to use `C_LONG_LONG` on Windows. On the Java side, we model the `size_t` using a `long` and the pointer is modeled using a `MemoryAddress` parameter.

Once we have obtained the downcall native method handle, we can just use it as any other method handle:

```
try (MemorySegment str = CLinker.toCString("Hello")) {
   long len = strlen.invokeExact(str.address()); // 5
}
```

Here we use one of the helper methods in `CLinker` to convert a Java string into an off-heap memory segment which contains a `NULL` terminated C string. We then pass that segment to the method handle and store the result in a Java `long`.

Observe that all this has been possible without any intervening native code — all of the interoperation code can be expressed in (low level) Java.

***Upcalls***

Sometimes it is useful to pass Java code as a function pointer to some native function. We can achieve that by using the foreign-linker support for upcalls. To demonstrate this, consider the following function defined in the standard C library:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

This is a function that can be used to sort the contents of an array, using a custom comparator function, `compar`, which is passed as a function pointer. To be able to call the `qsort` function from Java we have first to create a downcall native method handle for it:

```
MethodHandle qsort = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().lookup("qsort"),
        MethodType.methodType(void.class, MemoryAddress.class, long.class,
                              long.class, MemoryAddress.class),
        FunctionDescriptor.ofVoid(C_POINTER, C_LONG, C_LONG, C_POINTER)
    );
```

As before, we use `C_LONG` and `long.class` to map the C `size_t` type, and we use `MemoryAddess.class` both for the first pointer parameter (the array pointer) and the last parameter (the function pointer).

This time, in order to invoke the `qsort` downcall handle, we need a *function pointer* to pass as the last parameter. This is where the upcall support of the foreign-linker abstraction comes in handy, since it allows us to create a function pointer from an existing method handle. First, we write a static method that can compare two int elements, passed as pointers:

```
class Qsort {
    static int qsortCompare(MemoryAddress addr1, MemoryAddress addr2) {
            return MemoryAccess.getIntAtOffset(MemorySegment.ofNativeRestricted(),
                                               addr1.toRawLongValue()) -
                   MemoryAccess.getIntAtOffset(MemorySegment.ofNativeRestricted(),
                                               addr2.toRawLongValue());
    }
}
```

Then we create a method handle pointing to the above comparator function:

```
MethodHandle comparHandle
    = MethodHandles.lookup()
                   .findStatic(Qsort.class, "qsortCompare",
                               MethodType.methodType(int.class,
                                                     MemoryAddress.class,
                                                     MemoryAddress.class));
```

Now that we have a method handle for our Java comparator we can create a function pointer. Just as for downcalls, we describe the signature of the foreign-function pointer using the layouts in the `CLinker` class:

```
MemorySegment comparFunc
    = CLinker.getInstance().upcallStub(comparHandle,
                                       FunctionDescriptor.of(C_INT,
                                                             C_POINTER,
                                                             C_POINTER));
);
```

We finally have a memory segment, `comparFunc`, whose base address points to a stub that can be used to invoke our Java comparator function, and so we now have all we need to invoke the `qsort` downcall handle:

```
try (MemorySegment array = MemorySegment.allocateNative(4 * 10)) {
    array.copyFrom(MemorySegment.ofArray(new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 }));
    qsort.invokeExact(array.address(), 10L, 4L, comparFunc.address());
    int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
}
```

This code creates an off-heap array, copies the contents of a Java array into it, and then passes the array to the `qsort` handle along with the comparator function we obtained from the foreign linker. As a side effect, after the invocation the contents of the off-heap array will be sorted according to our comparator function, written in Java. We then extract a new Java array from the segment, which contains the sorted elements.

This advanced example shows the full power of the foreign-linker abstraction, with full bidirectional interoperation of both code and data across the Java/native boundary.

### Alternatives

Keep using JNI, or other third-party native interoperation frameworks.

### Risks and Assumptions

- The JIT implementations will require some work to ensure that uses of the native method handles retrieved form the API are at least as efficient and optimizable as uses of existing JNI native methods.
- Allowing foreign function calls always implies relaxing some of the safety requirements typically associated with the Java Platform. (This is already the case when invoking JNI native methods, although developers might not be aware of that). For instance, there is no way for the Foreign Linker API to validate, e.g., that the number of arguments in a function descriptor matches that of the symbol being linked. To help troubleshoot some of the most common failure causes, additional debugging capabilities may be provided, similar to the existing `-Xcheck:jni` option.
- Since the Foreign Linker API is intrinsically unsafe, obtaining a foreign linker instance is a privileged, restricted operation which requires the `-Dforeign.restricted=permit` flag.

### Dependencies

- The API described in this JEP represents a significant milestone towards the native interoperation support that is a goal of Project Panama, and builds heavily upon the Foreign-Memory Access API described in JEP 370 and JEP 383.
- The work described in this JEP will likely enable subsequent work to provide a tool, `jextract`, which, starting from the header files for a given native library, mechanically generates the native method handles required to interoperate with that library. This will further reduce the overhead of using native libraries from Java.