JEP 191: Foreign Function Interface

Author	Charles Oliver Nutter
Owner	John Rose
Created	2014/01/28 20:00
Updated	2014/11/17 23:02
Type	Feature
Status	Draft
Scope	JDK
JSR	TBD
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	M
Duration	L
Priority	4
Issue	8046181

Summary

Define a Foreign Function Interface that can bind native functions, such as those found in shared libraries and operating-system kernels, to Java methods, and also directly manage blocks of native memory.

Goals

- Provide a foreign function interface at the Java level, similar to JNA (Java Native Access) or JNR (Java Native Runtime).
- Optimize calls to native functions and management of native memory at the JVM level (optional).
- Support a future JSR for a standard Java FFI.

Non-Goals

- Mechanisms for explicitly defining the managed and unmanaged structure of standard Java objects.
- Other CPU-related improvements to Java object management such as cache line control, GPU/vectorization, etc.
- Direct modifications or improvements to JNI itself. These may come during the JSR process, however, and JVM enhancements for FFI may require internal changes.

Success Metrics

Success will be having an FFI API at the Java level sufficient for implementing large-scale native-backed features like NIO, advanced filesystem metadata, process management, and the like. This FFI API should ideally become preferred over writing JNI backends for each platform when a new native-level feature is required.

Motivation

The enhancement of the JDK (and of Java in general) by adding native-level features (new filesystem structures, different I/O channels, new crypto backends) has been stymied by the need to write JNI code to back those features for every new platform. The expertise necessary to write proper JNI is orthogonal to the expertise needed to write good Java code or call native libraries, but JNI has been the only path toward bridging those two worlds. This API will provide a tool enabling any JDK developer with understanding of Java and of a specific native library to bind that library as a JDK API in furtherance of JDK and Java features.

This JEP should make it easier to add new OS, hardware, and native library-level features to Java and the JDK as well as provide a standard Java FFI for use in the wider Java world.

VisualVM
Zero

Tools
Java SE
Mercurial
NetBeans
jtreg harness

Community
Planet JDK blogs
The Aquarium
java.net
java.sun.com
Java Community Process

Related
JDK Snapshots (8u, 9)
GlassFish
IoT

Description

For years, Java and JDK developers have had only one trusted way to add native-level features to Java applications: the Java Native Interface (JNI). This interface governs the boundary between the managed environment of the JVM and the unmanaged environment of native code, providing explicit protocols for data marshaling, object lifecycle management, upcalls back into Java, and native JVM tooling and management APIs. JNI provides a strict, unmistakable demarcation between managed and native code; it also makes that boundary incredibly painful to cross, even for the most trivial cases.

The following aspects of JNI are most painful to developers:

- Requiring developers to write C code means they have to have expertise of a world completely different from Java. In many cases, the native binding required is a trivial function call with no side effects and little to no data-marshaling complications, but in every case JNI requires developers to be C programmers.
- Using JNI requires expertise usually not found in typical C and Java developers, since the developer must have at least some understanding of how the JVM manages memory and code (object lifecycles, GC complications, Java class layout, the JVM lifecycle). JNI makes it possible to do the right thing, but much easier to do the wrong thing.
- Beyond simply writing C code, developers must be able to build that code for each platform they wish to support, or provide appropriate tooling for end-users to do the same. This is required despite the fact that the JDK itself is provided across dozens of platforms and platform-specific experts have already done significant work to make the JVM run and access native code on those platforms. This requirement makes JNI more detrimental to write-once, run-anywhere than an FFI API, since the latter will be much more likely to work across many platforms with only a single binary, and fixing incompatibilities will usually still only require a new build and release of that one binary.
- Even after crafting perfect JNI code and providing builds for every platform, the performance of JNI-based libraries is usually very poor compared to the same library bound into a native application. This stems from the inescapable rigidity of JNI's managed/unmanaged demarcation and the complete inability of the JVM's own optimizations to see through JNI calls. In many cases, JNI downcalls to trivial functions could be done directly from jitted Java code, since they do not require memory gymnastics and do not interfere with JVM internals. JNI's guarantees make it impossible to make native calls as lightweight as they could be.
- Finally, JNI acts as an opaque boundary for security. The JDK only knows about permission to load a specific library; it does not know what calls the functions in that library might use or whether the code in that library could compromise the stability or security of the JVM. This invites mistakes from JNI developers who are not expert C programmers or who simply don't understand the JVM-level aspects of security.

All the difficulties of JNI would be addressed by providing a built-in FFI API at the JDK level. An FFI would be easier for Java developers to write, not require as much knowledge of JVM internals, favor correct implementation or fast-fail over latent bugs, and eliminate the requirement for per-platform build expertise.

The JDK FFI API will provide the following to JDK developers:

1. A metadata system to describe native library calls (call protocol, argument list structure, argument types, return type) and native memory structure (size, layout, typing, lifecycle).
2. Mechanisms for discovering and loading native libraries. These capabilities may be provided by current `System.loadLibrary` or may include additional enhancements for locating platform or version-specific binaries

appropriate to the host system.

3. Mechanisms for binding, based on metadata, a given library/function coordinate to a Java endpoint, likely via a user-defined interface backed by plumbing to make the native downcall.
4. Mechanisms for binding, based on metadata, a specific memory structure (layout, endianness, logical types) to a Java endpoint, either via a user-defined interface or a user-defined class, in both cases backed by plumbing to manage a real block of native memory.
5. Appropriate support code for marshaling Java data types to native data types and vice-versa. This will in some cases require the creation of FFI-specific types to support bit widths and numeric signs that Java can't represent.

Optionally, this JEP will build additional support for the above features via:

1. JVM-level awareness of FFI downcalls. This could include: JIT optimization of those calls, JVM/GC-level awareness of native memory, protection against illegal native memory accesses (SEGV faults), and mechanisms to opt out of JNI safeguards known to be unnecessary in specific cases (safepoint boundaries, blocking call guarantees, object lifecycle management, etc.).
2. Tooling at either build time or run time for reflectively gathering function and memory metadata from native libraries. This would aid the initial binding of a library by providing a way to generate that binding at the Java level rather than requiring hand-implementation and tweaking for each platform. Prior work here includes the ffi-gen library for (J)Ruby, which uses clang (LLVM C compiler) metadata APIs for generaing Ruby FFI code.
3. The JVM security subsystem should understand specific library/fuction coordinates. It should be possible to set up security policies that allow binding only specific functions in specific libraries, rather than just the coarse-grained library-level permissions that exist today.

The level of abstraction for the JDK FFI is TBD; at minimum, it must be able to:

1. Load a library;
2. Invoke a function at some offset in that library;
3. Pass bit-appropriate arguments to that library (width, endianness) and receive bit-appropriate values back (typing is not relevant at this level, beyond describing bit width and structure); and
4. Minimally manage native memory: allocation, deallocation, access, and passing to/receiving from native calls.

There will, no doubt, be further discussion on how far to go with this API.

Alternatives

The need for a Java FFI has spawned several libraries. Among these, Java Native Access (JNA) is the most pervasively used, and Java Native Runtime (JNR) is perhaps the most comprehensive and advanced. JNR is likely to form the basis for this JEP, since it implements various levels of abstraction, provides function and memory metadata, abstracts away library and function binding, and has been in heavy use by at least the JRuby project (and its users) for at least the last five years.

Testing

In order to test this API, we will need to add representative native endpoints (as C code) to provide sufficient coverage of all call protocols, type marshaling, and memory management features. Many of these capabilities may exist in current JDK and Java test suites. The JNR library also has an existing suite of tests that could (will) be incorporated into JDK.

Risks and Assumptions

JNR's original author has recently gone on hiatus from open-source development. However, he believes he will be able to assist us in this effort.

JNR's license may or may not be compatible with the OpenJDK Community's standard licensing terms, but it is negotiable.

Dependences

No known dependencies on other JEPs or JSRs. Not known to block any other JEPs or JSRs.

Impact

- Compatibility: FFI will increase the work required to guarantee the cross-platform compatibility of the JDK. However, the JDK's supported platforms already support FFI via JNR.
- Security: FFI opens up the potential for user-level code to access untrusted native functions and read or write normally-inaccessible areas of native memory. This capability is no more extreme than that provided by existing APIs (`sun.misc.Unsafe`, et al). It will be possible to explicitly define security controls based on library/function coordinates; this capability represents an improvement over coarse-grained `System.loadLibrary` controls.
- Performance/scalability: Performance of native bindings should improve, if those bindings are written to use FFI. It is an open question whether existing JNI-based features should be rewritten to use FFI, and in what timeframe that should happen.
- Portability: There are obvious portability concerns, but no more than exist in current builds of the JDK. JDK code that uses FFI will still need testing across supported platforms to ensure functionality.
- Documentation: Ideally this API will become the preferred way to bind native code and memory, and so developer documentation should provide everything needed for JDK developers to use this API instead of JNI.
- TCK: A future Java FFI JSR will obviously need additions to the TCK, but ideally this will be little more than the testing provided by the JDK-level FFI.